Парадигма развития науки

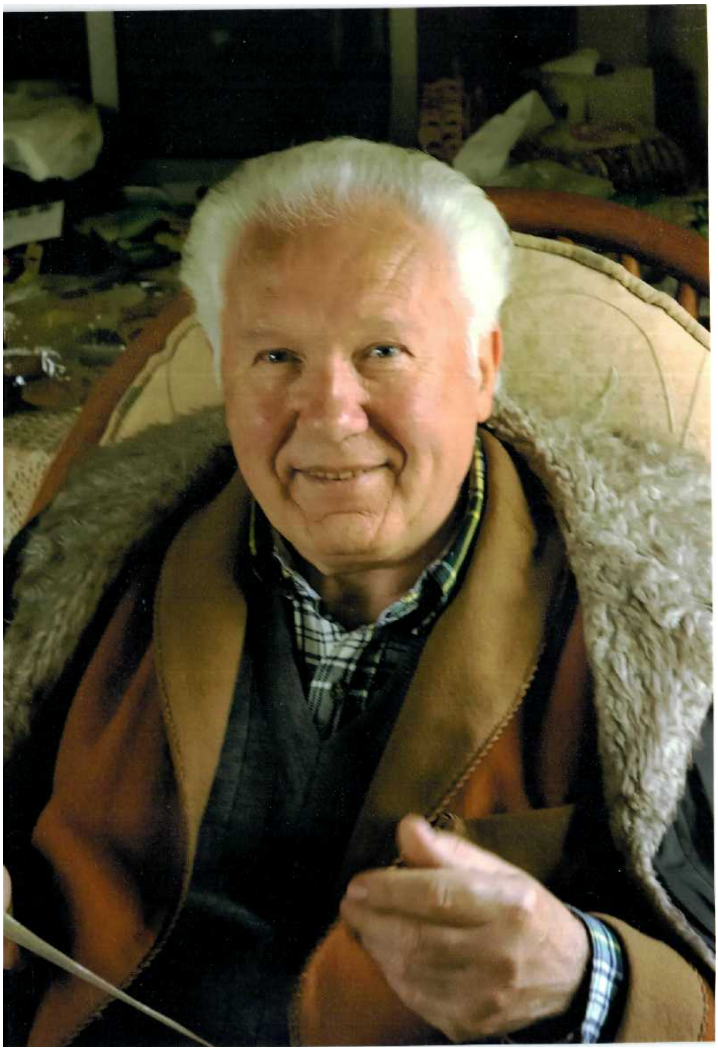
А. Е. Кононюк

Основы фундаментальной теории искусственного интеллекта

Книга 9

Представление данных в искусственном интеллекте

Киев
«Освіта України»
2018



Кононюк Анатолий Ефимович

Структурная схема
парадигмы развития науки

УДК 51 (075.8)
ББК В161.я7
К65

Рецензент:
Н.К.Печурин - д-р техн. наук, проф. (Национальный авиационный университет).

Кононюк А. Е.
К213 Основы фундаментальной теория искусственного интеллекта. — В 20-и кн. Кн.9. — К.:Освіта України. 2018.— 620 с.

ISBN 978-966-373-693-8 (многотомное издание)
ISBN 978-966-373-694-19 (книга 9)

Многотомная работа посвящена систематическому изложению общих формализмов, математических моделей и алгоритмических методов, которые могут быть исспользуемых при моделировании и исследованиях математических моделей объектов искусственного интеллекта.

Развиваются представления и методы решения, основанные на теориях эвристического поиска и автоматическом доказательстве теорем, а также процедуральные методы, базирующиеся на классе проблемно-ориентированных языков, сочетающих свойства языков программирования и автоматических решателей задач отображения искусственного интеллекта различными математическими средствами.

В работе излагаются основы теории отображения искусственного интеллекта такими математическими средствами как: множества, отношения, поверхности, пространства, алгебраические системы, матрицы, графы, математическая логика и др.

Для бакалавров, специалистов, магистров, аспирантов, докторантов всех специальностей.

УДК 51 (075.8)
ББК В161.я7
ISBN 978-966-373-693-8 (многотомное издание)
ISBN 978-966-373-694-19 (книга 9)

Оглавление

1. Методы представление данных

1.1. Основные понятия и определения

Да́нные — зарегистрированная информация; представление фактов, понятий или инструкций в форме, приемлемой для общения, интерпретации, или обработки человеком или с помощью автоматических средств (ISO/IEC/IEEE 24765-2010).

В информатике и информационных технологиях:

- Данные — поддающееся многократной интерпретации представление информации в формализованном виде, пригодном для передачи, связи или обработки (ISO/IEC 2382:2015).
- Данные — формы представления информации, с которыми имеют дело информационные системы и их пользователи (ISO/IEC 10746-2:1996).

В метрологии:

- Данные — совокупность значений, сопоставленных основным или производным мерам и/или показателям (ISO/IEC 15939:2007, ISO/IEC 25000:2005).

Хотя информация должна обрести некоторую *форму представления* (то есть превратиться в данные), чтобы ей можно было обмениваться, информация есть в первую очередь интерпретация (смысл) такого представления (ISO/IEC/IEEE 24765:2010). Поэтому в строгом смысле *информация* отличается от *данных*, хотя в неформальном контексте эти два термина очень часто используют как синонимы.

Метаданные (от лат. *meta* — цель, конечный пункт, предел, край и данные) — информация о другой информации, или данные, относящиеся к дополнительной информации о содержимом или объекте. Метаданные раскрывают сведения о признаках и свойствах, характеризующих какие-либо сущности, которые позволяют

автоматически искать и управлять ими в больших информационных потоках.

Различие между данными и метаданными

Обычно невозможно провести однозначное разделение на данные и **метаданные** в документе, поскольку:

- Что-то может являться как данными, так и метаданными. Так, заголовок статьи можно одновременно отнести как к метаданным (как элемент метаданных — заголовок), так и к собственно данным (поскольку заголовок является частью самого текста).
- Данные и метаданные могут меняться ролями. На стихотворение, рассматриваемое как данные, может быть написана музыка, в этом случае всё стихотворение может быть «прикреплено» к музыкальному файлу и в этом случае рассматриваться как метаданные. Таким образом, отнесение к одной или другой категории зависит от точки зрения (или пространства имён, системы отсчёта).
- Возможно создание мета-мета-…-метаданных (см. аксиома выбора). Поскольку, в соответствии с обычным определением, метаданные являются данными, то можно создать метаданные на метаданные, метаданные для вывода на специальные устройства, либо чтения их описания с использованием программного обеспечения, преобразующего текст в речь.

Больши́е да́нные (англ. *big data*, [ˈbɪg ˈdeɪtə]) — обозначение структурированных и неструктурированных данных огромных объёмов и значительного многообразия, эффективно обрабатываемых горизонтально масштабируемыми (англ. *scale-out*) программными инструментами, появившимися в конце 2000-х годов и альтернативных традиционным системам управления базами данных и решениям класса Business Intelligence.

Пространственные данные (**географические данные**, **геоданные**) — данные о пространственных объектах и их наборах. Пространственные данные составляют основу информационного обеспечения геоинформационных систем.

1.2. Большие данные

Больши́е да́нные (англ. *big data*, [ˈbɪg ˈdeɪtə]) — обозначение структурированных и неструктурированных данных огромных объёмов и значительного многообразия, эффективно обрабатываемых горизонтально масштабируемыми (англ. *scale-out*) программными инструментами, появившимися в конце 2000-х годов и альтернативных традиционным системам управления базами данных и решениям класса Business Intelligence.

В широком смысле о «больших данных» говорят как о социально-экономическом феномене, связанном с появлением технологических возможностей анализировать огромные массивы данных, в некоторых проблемных областях — весь мировой объём данных, и вытекающих из этого трансформационных последствий.

В качестве определяющих характеристик для больших данных традиционно выделяют **«три V»: объём (англ. *volume*, в смысле величины физического объёма), скорость (*velocity* в смыслах как скорости прироста, так и необходимости высокоскоростной обработки и получения результатов), многообразие (*variety*, в смысле возможности одновременной обработки различных типов структурированных и полуструктурированных данных)**; в дальнейшем возникли различные вариации и интерпретации этого признака.

С точки зрения **информационных технологий** в совокупность подходов и инструментов изначально включались средства массово-параллельной обработки неопределённо структурированных данных, прежде всего, системами управления базами данных категории NoSQL, алгоритмами MapReduce и реализующими их программными каркасами и библиотеками проекта Hadoop. В дальнейшем к серии технологий больших данных стали относить разнообразные информационно-технологические решения, в той или иной степени обеспечивающие сходные по характеристикам возможности по обработке сверхбольших массивов данных.

Широкое введение термина «большие данные» связывают с Клиффордом Линчем, редактором журнала Nature, подготовившим к 3 сентября 2008 года специальный выпуск с темой *«Как могут повлиять*

на будущее науки технологии, открывающие возможности работы с большими объёмами данных?», в котором были собраны материалы о феномене взрывного роста объёмов и многообразия обрабатываемых данных и технологических перспективах в парадигме вероятного скачка «от количества к качеству»; термин был предложен по аналогии с расхожими в деловой англоязычной среде метафорами *«большая нефть»*, *«большая руда»*.

Несмотря на то, что термин вводился в академической среде и прежде всего разбиралась проблема роста и многообразия научных данных, начиная с 2009 года термин широко распространился в деловой прессе, а к 2010 году относят появление первых продуктов и решений, относящихся исключительно и непосредственно к проблеме обработки больших данных. К 2011 году большинство крупнейших поставщиков информационных технологий для организаций в своих деловых стратегиях используют понятие о больших данных, в том числе IBM, Oracle, Microsoft, Hewlett-Packard, EMC, а основные аналитики рынка информационных технологий посвящают концепции выделенные исследования.

В 2011 году Gartner отметил большие данные как тренд номер два в информационно-технологической инфраструктуре (после виртуализации и как более существенный, чем энергосбережение и мониторинг). В это же время прогнозировалось, что внедрение технологий больших данных наибольшее влияние окажет на информационные технологии в производстве, здравоохранении, торговле, государственном управлении, а также в сферах и отраслях, где регистрируются индивидуальные перемещения ресурсов[20].

С 2013 года большие данные как академический предмет изучаются в появившихся вузовских программах по науке о данных и вычислительным наукам и инженерии.

В 2015 году Gartner исключил большие данные из цикла зрелости новых технологий и прекратил выпускать выходивший в 2011—2014 годы отдельный цикл зрелости технологий больших данных, мотивировав это переходом от этапа шумихи к практическому применению. Технологии, фигурировавшие в выделенном цикле зрелости, по большей части перешли в специальные циклы по продвинутой аналитике и науке о данных, по BI и анализу данных,

корпоративному управлению информацией, резидентным вычислениям, информационной инфраструктуре.

1.2.1. VVV

Набор признаков ***VVV*** (*volume*, *velocity*, *variety*) изначально выработан Meta Group в 2001 году вне контекста представлений о больших данных как об определённой серии информационно-технологических методов и инструментов, в нём, в связи с ростом популярности концепции центрального хранилища данных для организаций, отмечалась равнозначимость проблематик управления данными по всем трём аспектам. В дальнейшем появились интерпретации с «четырьмя V» (добавлялась *veracity* — достоверность, использовалась в рекламных материалах IBM), «пятью V» (в этом варианте прибавляли *viability* — жизнеспособность, и *value* — ценность), и даже «семью V» (кроме всего, добавляли также *variability* — переменчивость, и *visualization*). IDC интерпретирует «четвёртое V» как *value* с точки зрения важности экономической целесообразности обработки соответствующих объёмов в соответствующих условиях, что отражено также и в определении больших данных от IDC. Во всех случаях в этих признаках подчёркивается, что определяющей характеристикой для больших данных является не только их физический объём, но другие категории, существенные для представления о сложности задачи обработки и анализа данных.

Источники

Классическими источниками больших данных признаются интернет вещей и социальные медиа, считается также, что большие данные могут происходить из внутренней информации предприятий и организаций (генерируемой в информационных средах, но ранее не сохранявшейся и не анализировавшейся), из сфер медицины и биоинформатики, из астрономических наблюдений.

В качестве примеров источников возникновения больших данных приводятся непрерывно поступающие данные с измерительных устройств, события от радиочастотных идентификаторов, потоки сообщений из социальных сетей, метеорологические данные, данные дистанционного зондирования Земли, потоки данных о местонахождении абонентов сетей сотовой связи, устройств аудио- и видеорегистрации. Ожидается, что развитие и начало широкого

использования этих источников инициирует проникновение технологий больших данных как в научно-исследовательскую деятельность, так и в коммерческий сектор и сферу государственного управления.

1.2.2. Методы анализа

Методы и техники анализа, применимые к большим данным, выделенные в отчёте McKinsey:

- методы класса Data Mining: обучение ассоциативным правилам (англ. *association rule learning*), классификация (методы категоризации новых данных на основе принципов, ранее применённых к уже наличествующим данным), кластерный анализ, регрессионный анализ;
- краудсорсинг — категоризация и обогащение данных силами широкого, неопределённого круга лиц, привлечённых на основании публичной оферты, без вступления в трудовые отношения;
- смешение и интеграция данных (англ. *data fusion and integration*) — набор техник, позволяющих интегрировать разнородные данные из разнообразных источников для возможности глубинного анализа, в качестве примеров таких техник, составляющих этот класс методов приводятся цифровая обработка сигналов и обработка естественного языка (включая тональный анализ);
- машинное обучение, включая обучение с учителем и без учителя, а также Ensemble learning (англ.) — использование моделей, построенных на базе статистического анализа или машинного обучения для получения комплексных прогнозов на основе базовых моделей (англ. *constituent models*, ср. со статистическим ансамблем в статистической механике);
- искусственные нейронные сети, сетевой анализ, оптимизация, в том числе генетические алгоритмы;
- распознавание образов;
- прогнозная аналитика;
- имитационное моделирование;
- пространственный анализ (англ. *Spatial analysis*) — класс методов, использующих топологическую, геометрическую и географическую информацию в данных;

- статистический анализ, в качестве примеров методов приводятся A/B-тестирование и анализ временных рядов;
- визуализация аналитических данных — представление информации в виде рисунков, диаграмм, с использованием интерактивных возможностей и анимации как для получения результатов, так и для использования в качестве исходных данных для дальнейшего анализа.

1.2.3. Технологии

Наиболее часто указывают в качестве базового принципа обработки больших данных горизонтальную масштабируемость, обеспечивающую обработку данных, распределённых на сотни и тысячи вычислительных узлов, без деградации производительности; в частности, этот принцип включён в определение больших данных от NIST. При этом McKinsey, кроме рассматриваемых большинством аналитиков технологий NoSQL, MapReduce, Hadoop, R, включает в контекст применимости для обработки больших данных также технологии Business Intelligence и реляционные системы управления базами данных с поддержкой языка SQL.

1.2.3.1. NoSQL

NoSQL (от англ. *not only SQL — не только SQL*) — термин, обозначающий ряд подходов, направленных на реализацию хранилищ баз данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных СУБД с доступом к данным средствами языка SQL. Применяется к базам данных, в которых делается попытка решить проблемы масштабируемости и доступности за счёт атомарности (англ. *atomicity*) и согласованности данных (англ. *consistency*).

Происхождение

Изначально слово NoSQL являлось акронимом из двух слов английского языка: No («Не») и SQL (сокращение от англ. *Structured Query Language* — «структурированный язык запросов»), что даёт термину смысл «отрицающий SQL». Возможно, что первые, кто стал употреблять этот термин, хотели сказать «No RDBMS» («не

реляционная СУБД») или «no relational» («не реляционный»), но NoSQL звучало лучше и в итоге прижилось (в качестве альтернативы предлагалось также NonRel). Позднее для NoSQL было придумано объяснение «Not Only SQL» («не только SQL»). **NoSQL стал общим термином для различных баз данных и хранилищ, но он не обозначает какую-либо одну конкретную технологию или продукт**.

Развитие идеи

Сама по себе идея нереляционных баз данных не нова, а использование нереляционных хранилищ началось ещё во времена первых компьютеров. Нереляционные базы данных процветали во времена мэйнфреймов, а позднее, во времена доминирования реляционных СУБД, нашли применение в специализированных хранилищах, например, иерархических службах каталогов. Появление же нереляционных СУБД нового поколения произошло из-за необходимости создания параллельных распределённых систем для высокомасштабируемых интернет-приложений, таких как поисковые системы.

В начале 2000-х годов Google построил свою высокомасштабируемую поисковую систему и приложения: GMail, Google Maps, Google Earth и т. п., решая проблемы масштабируемости и параллельной обработки больших объёмов данных. В результате была создана распределённая файловая система и распределённая система координации, хранилище семейств колонок (англ. *column family store*), среда выполнения, основанная на алгоритме MapReduce. Публикация компанией Google описаний этих технологий привела к всплеску интереса среди разработчиков открытого программного обеспечения, в результате чего был создан Hadoop и запущены связанные с ним проекты, призванные создать подобные Google технологии. Через год, в 2007 году, примеру Google последовал Amazon.com, опубликовав статьи о высокодоступной базе данных Amazon DynamoDB.

Поддержка гигантов индустрии менее чем за пять лет привела к широкому распространению технологий NoSQL (и подобных) для управления «большими данными», а к делу присоединились другие большие и маленькие компании, такие как: IBM, Facebook, Netflix, EBay, Hulu, Yahoo!, со своими проприетарными и открытыми решениями.

Основные черты

Традиционные СУБД ориентируются на требования ACID к транзакционной системе: атомарность (англ. *atomicity*), согласованность (англ. *consistency*), изолированность (англ. *isolation*), надёжность (англ. *durability*), тогда как в NoSQL вместо ACID может рассматриваться набор свойств BASE:

- базовая доступность (англ. *basic availability*) — каждый запрос гарантированно завершается (успешно или безуспешно).
- гибкое состояние (англ. *soft state*) — состояние системы может изменяться со временем, даже без ввода новых данных, для достижения согласования данных.
- согласованность в конечном счёте (англ. *eventual consistency*) — данные могут быть некоторое время рассогласованы, но приходят к согласованию через некоторое время.

Термин «BASE» был предложен Эриком Брюером, автором теоремы CAP, согласно которой в распределённых вычислениях можно обеспечить только два из трёх свойств: согласованность данных, доступность или устойчивость к разделению.

Разумеется, системы на основе BASE не могут использоваться в любых приложениях: для функционирования биржевых и банковских систем использование транзакций является необходимостью. В то же время, свойства ACID, какими бы желанными они ни были, практически невозможно обеспечить в системах с многомиллионной веб-аудиторией, вроде amazon.com. Таким образом, проектировщики NoSQL-систем жертвуют согласованностью данных ради достижения двух других свойств из теоремы CAP. Некоторые СУБД, например, Riak, позволяют настраивать требуемые характеристики доступности-согласованности даже для отдельных запросов путём задания количества узлов, необходимых для подтверждения успеха транзакции.

Решения NoSQL отличаются не только проектированием с учётом масштабирования. Другими характерными чертами NoSQL-решений являются:

- Применение различных типов хранилищ.

- Возможность разработки базы данных без задания схемы.
- Линейная масштабируемость (добавление процессоров увеличивает производительность).
- Инновационность: «не только SQL» открывает много возможностей для хранения и обработки данных.

Типы хранилищ данных

Описание схемы данных в случае использования NoSQL-решений может осуществляться через использование различных структур данных: хеш-таблиц, деревьев и других.

В зависимости от модели данных и подходов к распределённости и репликации можно выделить четыре типа хранилищ: «ключ-значение» (key-value store), документно-ориентированные (document store), хранилища семейств колонок (column database), графовые базы данных (graph database).

Хранилище «ключ-значение»

Хранилище «ключ-значение» является простейшим хранилищем данных, использующим ключ для доступа к значению. Такие хранилища используются для хранения изображений, создания специализированных файловых систем, в качестве кэшей для объектов, а также в системах, спроектированных с прицелом на масштабируемость. Примеры таких хранилищ — Berkeley DB, MemcacheDB (англ.)русск., Redis, Riak, Amazon DynamoDB.

Хранилище семейств колонок (или Bigtable-подобные базы данных)

В этом хранилище данные хранятся в виде разреженной матрицы, строки и столбцы которой используются как ключи. Типичным применением этого вида СУБД является веб-индексирование, а также задачи, связанные с большими данными, с пониженными требованиями к согласованности данных. Примерами СУБД данного типа являются: Apache HBase, Apache Cassandra, Apache Accumulo (англ.)русск., Hypertable (англ.)русск., SimpleDB (Amazon.com).

Хранилища семейств колонок и документно-ориентированные хранилища имеют близкие сценарии использования: системы управления содержимым, блоги, регистрация событий. Использование отметок времени (timestamp) позволяет использовать этот вид хранилища для организации счётчиков, а также регистрации и обработки различных данных, связанных со временем.

Хранилища семейств колонок (англ. *column family stores*) не следует путать с колоночными хранилищами (англ. *column stores*). Последние являются реляционными СУБД с раздельным хранением колонок (в отличие от более традиционного построчного хранения данных).

Документоориентированная СУБД

Документоориентированные СУБД служат для хранения иерархических структур данных. Находят своё применение в системах управления содержимым, издательском деле, документальном поиске и т. п. Примеры СУБД данного типа — CouchDB, Couchbase, MarkLogic, MongoDB, eXist, Berkeley DB XML.

Базы данных на основе графов

Графовые базы данных применяются для задач, в которых данные имеют большое количество связей, например, социальные сети, выявление мошенничества. Примеры: Neo4j, OrientDB, AllegroGraph (англ.)русск., Blazegraph (RDF-хранилище, ранее называлось Bigdata), InfiniteGraph, FlockDB, Titan.

Так как рёбра графа материализованы (англ. *materialized*), то есть, являются хранимыми, обход графа не требует дополнительных вычислений (как JOIN в SQL), но для нахождения начальной вершины обхода требуется наличие индексов. Графовые базы данных как правило поддерживают ACID, а также имеют различные языки запросов, вроде Gremlin и Cypher (Neo4j).

UnQL

В июле 2011 компания Couchbase, разработчик CouchDB, Memcached и Membase, анонсировала создание нового SQL-подобного языка

запросов — UnQL (Unstructured Data Query Language). Работы по созданию нового языка выполнили создатель SQLite Ричард Гипп (англ. *Richard Hipp*) и основатель проекта CouchDB Дэмиен Кац (англ. *Damien Katz*). Разработка передана сообществу на правах общественного достояния.

1.2.3.2. MapReduce

MapReduce — модель распределённых вычислений, представленная компанией Google, используемая для параллельных вычислений над очень большими, вплоть до нескольких петабайт, наборами данных в компьютерных кластерах.

MapReduce — это фреймворк для вычисления некоторых наборов распределенных задач с использованием большого количества компьютеров (называемых «нодами»), образующих кластер.

Работа MapReduce состоит из двух шагов: Map и Reduce, названных так по аналогии с одноименными функциями высшего порядка, map и reduce.

На Map-шаге происходит предварительная обработка входных данных. Для этого один из компьютеров (называемый главным узлом — master node) получает входные данные задачи, разделяет их на части и передает другим компьютерам (рабочим узлам — worker node) для предварительной обработки.

На Reduce-шаге происходит свёртка предварительно обработанных данных. Главный узел получает ответы от рабочих узлов и на их основе формирует результат — решение задачи, которая изначально формулировалась.

Преимущество MapReduce заключается в том, что он позволяет распределенно производить операции предварительной обработки и свертки. Операции предварительной обработки работают независимо друг от друга и могут производиться параллельно (хотя на практике это ограничено источником входных данных и/или количеством используемых процессоров). Аналогично, множество рабочих узлов может осуществлять свертку — для этого необходимо только чтобы все результаты предварительной обработки с одним конкретным

значением ключа обрабатывались одним рабочим узлом в один момент времени. Хотя этот процесс может быть менее эффективным по сравнению с более последовательными алгоритмами, MapReduce может быть применен к большим объёмам данных, которые могут обрабатываться большим количеством серверов. Так, MapReduce может быть использован для сортировки петабайта данных, что займет всего лишь несколько часов. Параллелизм также дает некоторые возможности восстановления после частичных сбоев серверов: если в рабочем узле, производящем операцию предварительной обработки или свертки, возникает сбой, то его работа может быть передана другому рабочему узлу (при условии, что входные данные для проводимой операции доступны).

Фреймворк в большой степени основан на функциях map и reduce, широко используемых в функциональном программировании, , хотя фактически семантика фреймворка отличается от прототипа.

Пример

Канонический пример приложения, написанного с помощью MapReduce, — это процесс, подсчитывающий, сколько раз различные слова встречаются в наборе документов:

```
// Функция, используемая рабочими нодами на Map-
шаге
// для обработки пар ключ-значение из входного
потока
void map(String name, String document):
    // Входные данные:
    //   name - название документа
    //   document - содержимое документа
    for each word w in document:
        EmitIntermediate(w, "1");

// Функция, используемая рабочими нодами на Reduce-
шаге
// для обработки пар ключ-значение, полученных на
Map-шаге
void reduce(String word, Iterator partialCounts):
    // Входные данные:
    //   word - слово
```

```
    //   partialCounts - список группированных
промежуточных результатов. Количество записей в
partialCounts и есть
    //     требуемое значение
    int result = 0;
    for each v in partialCounts:
        result += parseInt(v);
    Emit(AsString(result));
```

В этом коде на Map-шаге каждый документ разбивается на слова, и возвращаются пары, где ключом является само слово, а значением — «1». Если в документе одно и то же слово встречается несколько раз, то в результате предварительной обработки этого документа будет столько же этих пар, сколько раз встретилось это слово.
Сформированные пары отправляются на дальнейшую обработку, система группирует их по ключу (в данном случае - ключом является само слово) и распределяет по множеству процессоров. Наборы объектов с одинаковым ключом в группе попадают на вход функции reduce, которая перерабатывает поток данных, сокращая его объёмы. В данном примере функция reduce просто складывает вхождения данного слова по всему потоку, и результат - только одна сумма - отправляется дальше в виде выходных данных.

1.2.3.3. Hadoop

Apache Hadoop

Тип	фреймворк и decentralized computing[d]
Автор	Дуг Каттинг[d] и Майк Кафарела[d]
Разработчик	Apache Software Foundation
Написана на	Java
Операционная система	кроссплатформенное программное обеспечение и POSIX
Первый выпуск	2005
Аппаратная	Java Virtual Machine

платформа	
Последняя версия	• 3.0.0 (13 декабря 2017)
Лицензия	Apache License 2.0 и GNU GPL
Сайт	hadoop.apache.org

Apache Hadoop на Викискладе

Hadoop — проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах из сотен и тысяч узлов. Используется для реализации поисковых и контекстных механизмов многих высоконагруженных веб-сайтов, в том числе, для Yahoo! и Facebook. Разработан на Java в рамках вычислительной парадигмы MapReduce, согласно которой приложение разделяется на большое количество одинаковых элементарных заданий, выполнимых на узлах кластера и естественным образом сводимых в конечный результат.

По состоянию на 2014 год, проект состоит из четырёх модулей — Hadoop Common[⇨] (связующее программное обеспечение — набор инфраструктурных программных библиотек и утилит, используемых для других модулей и родственных проектов), HDFS[⇨] (распределённая файловая система), YARN[⇨] (система для планирования заданий и управления кластером) и Hadoop MapReduce[⇨] (платформа программирования и выполнения распределённых MapReduce-вычислений), ранее в Hadoop входил целый ряд других проектов, ставших самостоятельными в рамках системы проектов Apache Software Foundation.

Считается одной из основополагающих технологий «больших данных». Вокруг Hadoop образовалась целая экосистема[⇨] из связанных проектов и технологий, многие из которых развивались изначально в рамках проекта, а впоследствии стали самостоятельными. Со второй половины 2000-х годов идёт процесс активной коммерциализации технологии[⇨], несколько компаний строят бизнес целиком на создании коммерческих дистрибутивов Hadoop и услуг по технической поддержке экосистемы, а практически все крупные

поставщики информационных технологий для организаций в том или ином виде включают Hadoop в продуктовые стратегии и линейки решений.

Разработка была инициирована в начале 2005 года Дугом Каттингом (англ. *Doug Cutting*) с целью построения программной инфраструктуры распределённых вычислений для проекта Nutch — свободной программной поисковой машины на Java, её идейной основой стала публикация сотрудников Google Джеффри Дина и Санжая Гемавата о вычислительной концепции MapReduce. Новый проект был назван в честь игрушечного слонёнка ребёнка основателя проекта.

В течение 2005—2006 годов Hadoop развивался усилиями двух разработчиков — Каттинга и Майка Кафареллы (*Mike Cafarella*) в режиме частичной занятости, сначала в рамках проекта Nutch, затем — проекта Lucene. В январе 2006 года корпорация Yahoo пригласила Каттинга возглавить специально выделенную команду разработки инфраструктуры распределённых вычислений, к этому же моменту относится выделение Hadoop в отдельный проект. В феврале 2008 года Yahoo запустила кластерную поисковую машину на 10 тыс. процессорных ядер, управляемую средствами Hadoop.

В январе 2008 года Hadoop становится проектом верхнего уровня системы проектов Apache Software Foundation. В апреле 2008 года Hadoop побил мировой рекорд производительности в стандартизованном бенчмарке сортировки данных — 1 Тбайт был обработан за 209 сек. на кластере из 910 узлов. С этого момента начинается широкое применение Hadoop за пределами Yahoo — технологию для своих сайтов внедряют Last.fm, Facebook, The New York Times, проводится адаптация для запуска Hadoop в облаках Amazon EC2.

В апреле 2010 года корпорация Google предоставила Apache Software Foundation права на использование технологии MapReduce, через три месяца после её защиты в патентном бюро США, тем самым избавив организацию от возможных патентных претензий.

Начиная с 2010 года Hadoop неоднократно характеризуется как ключевая технология «больших данных», прогнозируется его широкое распространение для массово-параллельной обработки данных, и, наряду с Cloudera, появилась серия технологических стартапов,

целиком ориентированных на коммерциализацию Hadoop. В течение 2010 года несколько подпроектов Hadoop — Avro, HBase, Hive, Pig, Zookeeper — последовательно стали проектами верхнего уровня фонда Apache, что послужило началом формирования экосистемы вокруг Hadoop[⇨]. В марте 2011 года Hadoop удостоен ежегодной инновационной награды медиагруппы Guardian, на церемонии вручения технология была названа *«швейцарским армейским ножом XXI века»*.

Реализация в вышедшем осенью 2013 года Hadoop 2.0 модуля YARN оценена как значительный скачок, выводящий Hadoop за рамки парадигмы MapReduce и ставящая технологию на уровень универсального решения для организации распределённой обработки данных.

Hadoop Common

В **Hadoop Common** входят библиотеки управления файловыми системами, поддерживаемыми Hadoop, и сценарии создания необходимой инфраструктуры и управления распределённой обработкой, для удобства выполнения которых создан специализированный упрощённый интерпретатор командной строки (*FS shell, filesystem shell*), запускаемый из оболочки операционной системы командой вида: `hdfs dfs -command URI`, где *`command`* — команда интерпретатора, а *`URI`* — список ресурсов с префиксами, указывающими тип поддерживаемой файловой системы, например `hdfs://example.com/file1` или `file:///tmp/local/file2`. Бóльшая часть команд интерпретатора реализована по аналогии с соответствующими командами Unix (таковы, например, `cat`, `chmod`, `chown`, `chgrp`, `cp`, `du`, `ls`, `mkdir`, `mv`, `rm`, `tail`, притом, поддержаны некоторые ключи аналогичных Unix-команд, например ключ рекурсивности `-R` для `chmod`, `chown`, `chgrp`), есть команды, специфические для Hadoop (например, `count` подсчитывает количество каталогов, файлов и байтов по заданному пути, `expunge` очищает корзину, а `setrep` модифицирует коэффициент репликации для заданного ресурса).

HDFS

HDFS (*Hadoop Distributed File System*) — файловая система, предназначенная для хранения файлов больших размеров, поблочно распределённых между узлами вычислительного кластера. Все блоки в HDFS (кроме последнего блока файла) имеют одинаковый размер, и каждый блок может быть размещён на нескольких узлах, размер блока и коэффициент репликации (количество узлов, на которых должен быть размещён каждый блок) определяются в настройках на уровне файла. Благодаря репликации обеспечивается устойчивость распределённой системы к отказам отдельных узлов. Файлы в HDFS могут быть записаны лишь однажды (модификация не поддерживается), а запись в файл в одно время может вести только один процесс. Организация файлов в пространстве имён — традиционная иерархическая: есть корневой каталог, поддерживается вложение каталогов, в одном каталоге могут располагаться и файлы, и другие каталоги.

Развёртывание экземпляра HDFS предусматривает наличие центрального *узла имён* (англ. *name node*), хранящего метаданные файловой системы и метаинформацию о распределении блоков, и серии *узлов данных* (англ. *data node*), непосредственно хранящих блоки файлов. Узел имён отвечает за обработку операций уровня файлов и каталогов — открытие и закрытие файлов, манипуляция с каталогами, узлы данных непосредственно отрабатывают операции по записи и чтению данных. Узел имён и узлы данных снабжаются веб-серверами, отображающими текущий статус узлов и позволяющими просматривать содержимое файловой системы. Административные функции доступны из интерфейса командной строки.

HDFS является неотъемлемой частью проекта, однако, Hadoop поддерживает работу и с другими распределёнными файловыми системами без использования HDFS, поддержка Amazon S3 и CloudStore реализована в основном дистрибутиве. С другой стороны, HDFS может использоваться не только для запуска MapReduce-заданий, но и как распределённая файловая система общего назначения, в частности, поверх неё реализована распределённая NoSQL-СУБД HBase, в её среде работает масштабируемая система машинного обучения Apache Mahout.

YARN

YARN (англ. *Yet Another Resource Negotiator* — «*ещё один ресурсный посредник*») — модуль, появившийся с версией 2.0 (2013), отвечающий за управление ресурсами кластеров и планирование заданий. Если в предыдущих выпусках эта функция была интегрирована в модуль MapReduce, где была реализована единым компонентом (*JobTracker*), то в YARN функционирует логически самостоятельный демон — планировщик ресурсов (*ResourceManager*), абстрагирующий все вычислительные ресурсы кластера и управляющий их предоставлением приложениям распределённой обработки. Работать под управлением YARN могут как MapReduce-программы, так и любые другие распределённые приложения, поддерживающие соответствующие программные интерфейсы; YARN обеспечивает возможность параллельного выполнения нескольких различных задач в рамках кластера и их изоляцию (по принципам мультиарендности). Разработчику распределённого приложения необходимо реализовать специальный класс управления приложением (*ApplicationMaster*), который отвечает за координацию заданий в рамках тех ресурсов, которые предоставит планировщик ресурсов; планировщик ресурсов же отвечает за создание экземпляров класса управления приложением и взаимодействия с ним через соответствующий сетевой протокол.

YARN может быть рассмотрен как кластерная операционная система в том смысле, что ведает интерфейсом между аппаратными ресурсами кластера и широким классом приложений, использующих его мощности для выполнения вычислительной обработки.

Hadoop MapReduce

Hadoop MapReduce — программный каркас для программирования распределённых вычислений в рамках парадигмы MapReduce. Разработчику приложения для Hadoop MapReduce необходимо реализовать базовый обработчик, который на каждом вычислительном узле кластера обеспечит преобразование исходных пар «ключ — значение» в промежуточный набор пар «ключ — значение» (класс, реализующий интерфейс `Mapper`, назван по функции высшего порядка Map), и обработчик, сводящий промежуточный набор пар в окончательный, сокращённый набор (*свёртку*, класс, реализующий

интерфейс `Reducer`). Каркас передаёт на вход свёртки отсортированные выводы от базовых обработчиков, сведе́ние состоит из трёх фаз — *shuffle* (*масовка*, выделение нужной секции вывода), *sort* (*сортировка*, группировка по ключам выводов от распределителей — досортировка, требующаяся в случае, когда разные атомарные обработчики возвращают наборы с одинаковыми ключами, при этом, правила сортировки на этой фазе могут быть заданы программно и использовать какие-либо особенности внутренней структуры ключей) и собственно *reduce* (*свёртка списка*) — получения результирующего набора. Для некоторых видов обработки свёртка не требуется, и каркас возвращает в этом случае набор отсортированных пар, полученных базовыми обработчиками.

Hadoop MapReduce позволяет создавать задания как с базовыми обработчиками, так и со свёртками, написанными без использования Java: утилиты *Hadoop streaming* позволяют использовать в качестве базовых обработчиков и свёрток любой исполняемый файл, работающий со стандартным вводом-выводом операционной системы (например, утилиты командной оболочки UNIX), есть также SWIG-совместимый прикладной интерфейс программирования *Hadoop pipes* на C++. Также, в состав дистрибутивов Hadoop входят реализации различных конкретных базовых обработчиков и свёрток, наиболее типично используемых в распределённой обработке.

В первых версиях Hadoop MapReduce включал планировщик заданий (*JobTracker*), начиная с версии 2.0 эта функция перенесена в YARN[⇨], и начиная с этой версии модуль Hadoop MapReduce реализован поверх YARN. Программные интерфейсы по большей части сохранены, однако полной обратной совместимости нет (то есть для запуска программ, написанных для предыдущих версий API, для работы в YARN в общем случае требуется их модификация или рефакторинг, и лишь при некоторых ограничениях возможны варианты обратной двоичной совместимости).

Масштабируемость

Одной из основных целей Hadoop изначально было обеспечение горизонтальной масштабируемости кластера посредством добавления недорогих узлов (оборудования массового класса, англ. *commodity hardware*), без прибегания к мощным серверам и дорогим сетям

хранения данных. Функционирующие кластеры размером в тысячи узлов подтверждают осуществимость и экономическую эффективность таких систем, так, по состоянию на 2011 год известно о крупных кластерах Hadoop в Yahoo (более 4 тыс. узлов с суммарной ёмкостью хранения 15 Пбайт), Facebook (около 2 тыс. узлов на 21 Пбайт) и Ebay (700 узлов на 16 Пбайт). Тем не менее, считается, что горизонтальная масштабируемость в Hadoop-системах ограничена, для Hadoop до версии 2.0 максимально возможно оценивалась в 4 тыс. узлов при использовании 10 MapReduce-заданий на узел. Во многом этому ограничению способствовала концентрация в модуле MapReduce функций по контролю за жизненным циклом заданий, считается, что с выносом её в модуль YARN в Hadoop 2.0 и децентрализацией — распределением части функций по мониторингу на узлы обработки — горизонтальная масштабируемость повысилась.

Ещё одним ограничением Hadoop-систем является размер оперативной памяти на узле имён (*NameNode*), хранящем всё пространство имён кластера для распределения обработки, притом общее количество файлов, которое способен обрабатывать узел имён — 100 млн. Для преодоления этого ограничения ведутся работы по распределению узла имён, единого в текущей архитектуре на весь кластер, на несколько независимых узлов. Другим вариантом преодоления этого ограничения является использование **распределённых СУБД** поверх HDFS, таких как HBase, роль файлов и каталогов в которых с точки зрения приложения играют записи в одной большой таблице базы данных.

По состоянию на 2011 год типичный кластер строился из однопроцессорных многоядерных x86-64-узлов под управлением Linux с 3—12 дисковыми устройствами хранения, связанных сетью с пропускной способностью 1 Гбит/с. Существуют тенденции как к снижению вычислительной мощности узлов и использованию процессоров с низким энергопотреблением (ARM, Intel Atom), так и применения высокопроизводительных вычислительных узлов одновременно с сетевыми решениями с высокой пропускной способностью (InfiniBand в Oracle Big Data Appliance, высокопроизводительная сеть хранения данных на Fibre Channel и Ethernet пропускной способностью 10 Гбит/с в шаблонных конфигурациях FlexPod для «больших данных»).

Масштабируемость Hadoop-систем в значительной степени зависит от характеристик обрабатываемых данных, прежде всего, их внутренней

структуры и особенностей по извлечению из них необходимой информации, и сложности задачи по обработке, которые, в свою очередь, диктуют организацию циклов обработки, вычислительную интенсивность атомарных операций, и, в конечном счёте, уровень параллелизма и загруженность кластера. В руководстве Hadoop (первых версий, ранее 2.0) указывалось, что приемлемым уровнем параллелизма является использование 10—100 экземпляров базовых обработчиков на узел кластера, а для задач, не требующих значительных затрат процессорного времени — до 300; для свёрток считалось оптимальным использование их по количеству узлов, умноженному на коэффициент из диапазона от 0,95 до 1,75 и константу `mapred.tasktracker.reduce.tasks.maximum`. С бóльшим значением коэффициента наиболее быстрые узлы, закончив первый раунд сведения, раньше получат вторую порцию промежуточных пар для обработки, таким образом, увеличение коэффициента избыточно загружает кластер, но при этом обеспечивает более эффективную балансировку нагрузки. В YARN вместо этого используются конфигурационные константы, определяющие значения доступной оперативной памяти и виртуальных процессорных ядер, доступных для планировщика ресурсов, на основании которых и определяется уровень параллелизма.

Коммерциализация

На фоне популяризации Hadoop в 2008 году и сообщениях о построении Hadoop-кластеров в Yahoo и Facebook, в октябре 2008 года была создана компания Cloudera во главе с Майклом Ольсоном, бывшим генеральным директором Sleepycat (фирмы-создателя Berkeley DB), целиком нацеленная на коммерциализацию Hadoop-технологий. В сентябре 2009 года в Cloudera из Yahoo перешёл основной разработчик Hadoop Дуг Каттинг, и благодаря такому переходу комментаторы охарактеризовали Cloudera как «нового знаменосца Hadoop», несмотря на то, что основная часть проекта была создана всё-таки сотрудниками Facebook и Yahoo. В 2009 году основана компания MapR, поставившая целью создать высокопроизводительный вариант дистрибутива Hadoop, и поставлять его как собственническое программное обеспечение. В апреле 2009 года Amazon запустил облачный сервис Elastic MapReduce, предоставляющий подписчикам возможность создавать кластеры Hadoop и выполнять на них задания с повременной оплатой. Позднее, в качестве альтернативы, подписчики Amazon Elastic

MapReduce получили выбор между классическим дистрибутивом от Apache и дистрибутивами от MapR.

В 2011 году Yahoo выделила подразделение, занимавшееся разработкой и использованием Hadoop, в самостоятельную компанию — Hortonworks, вскоре новой компании удалось заключить соглашение с Microsoft о совместной разработке дистрибутива Hadoop для Windows Azure и Windows Server. В том же году со становлением представлений о Hadoop как одной из базовых технологий «больших данных» фактически все крупные производители технологического программного обеспечения для организаций в том или ином виде включили Hadoop-технологии в стратегии и продуктовые линейки. Так, Oracle выпустила аппаратно-программный комплекс Big Data appliance (заранее собранный в телекоммуникационном шкафе и предконфигурированный Hadoop-кластер с дистрибутивом от Cloudera), IBM на основе дистрибутива Apache создала продукт BigInsights, EMC лицензировала у MapR их высокопроизводительный Hadoop для интеграции в продукты незадолго до этого поглощённой Greenplum (позднее это бизнес-подразделение было выделено в самостоятельную компанию Pivotal, и она перешла на полностью самостоятельный дистрибутив Hadoop на базе кода Apache), Teradata заключила соглашение с Hortonworks по интеграции Hadoop в аппаратно-программный комплекс массово-параллельной обработки Aster Big Analytics appliance. В 2013 году собственный дистрибутив Hadoop создала Intel, год спустя отказавшись от его развития в пользу решений от Cloudera, в которой приобрела долю в 18 %.

Объём рынка программного обеспечения и услуг вокруг экосистемы Hadoop на 2012 год оценён в размере $540 млн с прогнозом роста к 2017 году до $1,6 млрд, лидеры рынка — калифорнийские стартапы Cloudera, MapR и Hortonworks. Кроме них отмечены также компании Hadapt (поглощена в июле 2014 корпорацией Teradata), Datameer, Karmasphere и Platfora, как строящие целиком свой бизнес на создании продуктов для обеспечения Hadoop-систем аналитическими возможностями.

1.2.3.4. R (язык программирования)

R	
Класс языка	мультипарадигмальный
Тип исполнения	интерпретируемый
Появился в	1993[1]
Автор	Росс Айхэка Роберт Джентлмен
Выпуск	• 3.5.0 «Joy in Playing» (23 апреля 2018)[2]
Система типов	динамическая
Испытал влияние	S, Scheme
Лицензия	GNU GPL 2[3]
Сайт	r-project.org

R — язык программирования для статистической обработки данных и работы с графикой, а также свободная программная среда вычислений с открытым исходным кодом в рамках проекта GNU. Язык создавался как аналогичный языку S, разработанному в Bell Labs, и является его альтернативной реализацией, хотя между языками есть существенные отличия, но в большинстве своём код на языке S работает в среде R. Изначально R был разработан сотрудниками статистического факультета Оклендского университета Россом Айхэкой (англ. *Ross Ihaka*) и Робертом Джентлменом (англ. *Robert Gentleman*) (первая буква их имён — R); язык и среда поддерживаются и развиваются организацией *R Foundation*.

R широко используется как статистическое программное обеспечение для анализа данных и фактически стал стандартом для статистических программ.

R доступен под лицензией GNU GPL. Распространяется в виде исходных кодов, а также откомпилированных приложений под ряд операционных систем: FreeBSD, Solaris и другие дистрибутивы Unix и Linux, Microsoft Windows, Mac OS X.

В R используется интерфейс командной строки, хотя доступны и несколько графических интерфейсов пользователя, например пакет R Commander, RKWard, RStudio, Weka, Rapid Miner, KNIME, а также средства интеграции в офисные пакеты.

В 2010 году R вошёл в список победителей конкурса журнала Infoworld в номинации на лучшее открытое программное обеспечение для разработки приложений.

Особенности

R поддерживает широкий спектр статистических и численных методов и обладает хорошей расширяемостью с помощью пакетов. Пакеты представляют собой библиотеки для работы специфических функций или специальных областей применения. В базовую поставку R включен основной набор пакетов, а всего по состоянию на 2017 год доступно более 11778 пакетов.

Ещё одна особенность R - возможность создания качественной графики, которая может включать математические символы.

Примеры

Базовый синтаксис

```
> x <- c(1,2,3,4,5,6)   # Создать упорядоченную
коллекцию
> y <- x^2              # Возвести в квадрат
элементы из x
> print(y)              # Вывести y
[1]  1  4  9 16 25 36
> mean(y)               # Рассчитать среднее
арифметическое y; результат - число
[1] 15.16667
```

```
> var(y)                  # Рассчитать дисперсию
[1] 178.9667
```

Средний балл выпускника вуза

```
# В переменную a поместить список всех оценок:
a <- c(4,3,3,3,3,4,4,4,4,4,5,4,4,4,5,5,5,5,+
3,5,5,4,4,3,3,4,4,3,5,5,4,3,3,4,4,3,3,5,4,5,5)

# В переменную n поместить количество оценок:
length(a) -> n

# Средний балл:
m <- mean(a)

# Таблица (горизонтальная) с подсчётом количества
оценок:
t <- table(a)

# Преобразование в более удобный формат данных
(вертикальную таблицу):
f <- as.data.frame(t)

# Вычисление процентной доли и запись её в третий
столбец:
mapply(function(r) r*100/n, f[,2]) -> f[,3]

# Заголовки столбцов:
colnames(f) <- c("Оценка", "Кол-во", "%")

# Вывод результатов:
a
n
m
f
```

Результат:

```
[1] 4 3 3 3 3 4 4 4 4 4 5 4 4 4 5 5 5 5 3 5 5 4 4 3
3 4 4 3 5 5 4 3 3 4 4 3 3 5
[39] 4 5 5
[1] 41
```

```
[1] 4
  Оценка Кол-во        %
1      3     12 29.26829
2      4     17 41.46341111
3      5     12 29.26829
```

Инструменты

Для удобства работы с R разработан ряд графических интерфейсов, в том числе RStudio, JGR, RKWard, SciViews-R, Statistical Lab, R Commander, Rattle.

Кроме того, в ряде текстовых и кодовых редакторов предусмотренные специальные режимы для работы с R, в частности в ConTEXT, Emacs (Emacs Speaks Statistics), jEdit, Kate, Syn, TextMate, Tinn-R, Vim, Bluefish, WinEdt (с пакетом RWinEdt), Gedit (с пакетом rgedit/gedit-r-plugin). Для среды разработки Eclipse существует специализированный R-плагин; доступ к функциям и среде выполнения R возможен из Python с использованием пакета RPy; работать с R можно из эконометрического пакета Gretl.

Коммерциализация

Компания Revolution Analytics, основанная в 2007 году, целиком свой бизнес посвящает коммерциализации языка программирования R, в её коммерческом пакете Revolution R примечательны такие компоненты (не распространяемые со свободной версией языка), как *ParallelR* (поддержка многопоточности среды выполнения), *R Productivity Environment* (интегрированная среда разработки), *RevoScaleR* (поддержка массово-параллельной обработки в рамках концепции «больших данных»), *RevoDeployR*, библиотеки по интеграции с веб-службами, поддержка форматов статистических пакетов корпорации SAS Institute.

В октябре 2011 года корпорация Oracle выпустила аппаратно-программный комплекс Big Data Appliance — NoSQL-кластер серверов массово-параллельной обработки, с интегрированным программными средствами на основе языка R и Apache Hadoop, а в феврале 2012 года язык встроен в Oracle Database. В 2011 году массово-параллельный

анализ средствами R реализован в аппаратно-программных комплексах Netezza корпорации IBM; позднее язык поддержан в аппаратно-программном комплексе SAP Hana.

Также язык R поддерживают коммерческие программные среды Tibco Spotfire, SPSS (начиная с версии 16.0), Statistica (начиная с версии 9.0), Platform Symphony, SAS, Tableau.

CRAN

R и дополнительные пакеты распространяются через *CRAN* (акроним *Comprehensive R Archive Network*). В настоящее время в мире доступны более 60 зеркал CRAN. Головной узел — (http://cran.r-project.org/) расположен в Вене (Австрия).

Информационный бюллетень R

Два-три раза в год выходит свободно распространяемый информационный журнал R Journal. Он содержит информацию по статистической обработке данных и разработке, что может быть интересно как пользователям, так и разработчикам R. С января 2001 года по октябрь 2008 года он выходил в качестве бюллетеня R News.

Конференции

Одна из самых популярных конференций, посвящённых языку — *useR!* (*The R User Conference*), проходит ежегодно, начиная с 2004 года, собирает специалистов в различных областях.

Начиная с 2009 года каждой весной в Чикаго проводится конференция, посвящённая применению R в финансах (*R/Finance: Applied Finance with R*). В 2013 году прошла первая конференция, посвящённая применению R в страховании (*R in Insurance*).

1.2.4. Аппаратные решения

Существует ряд аппаратно-программных комплексов, предоставляющих предконфигурированные решения для обработки больших данных: Aster MapReduce appliance (корпорации Teradata), Oracle Big Data appliance, Greenplum appliance (корпорации EMC, на основе решений поглощённой компании Greenplum). Эти комплексы поставляются как готовые к установке в центры обработки данных телекоммуникационные шкафы, содержащие кластер серверов и управляющее программное обеспечение для массово-параллельной обработки.

Аппаратные решения для резидентных вычислений, прежде всего, для баз данных в оперативной памяти и аналитики в оперативной памяти, в частности, предлагаемой аппаратно-программными комплексами Hana (предконфигурированное аппаратно-программное решение компании SAP) и Exalytics (комплекс компании Oracle на основе реляционной системы Timesten (англ.) и многомерной Essbase), также иногда относят к решениям из области больших данных, несмотря на то, что такая обработка изначально не является массово-параллельной, а объёмы оперативной памяти одного узла ограничиваются несколькими терабайтами.

Кроме того иногда к решениям для больших данных относят и аппаратно-программные комплексы на основе традиционных реляционных систем управления базами данных — Netezza, Teradata, Exadata, как способные эффективно обрабатывать терабайты и эксабайты структурированной информации, решая задачи быстрой поисковой и аналитической обработки огромных объёмов структурированных данных. Отмечается, что первыми массово-параллельными аппаратно-программными решениями для обработки сверхбольших объёмов данных были машины компаний Britton Lee (англ.), впервые выпущенные в 1983 году, и Teradata (начали выпускаться в 1984 году, притом в 1990 году Teradata поглотила Britton Lee).

Аппаратные решения DAS — систем хранения данных, напрямую присоединённых к узлам — в условиях независимости узлов обработки в SN-архитектуре также иногда относят к технологиям больших данных. Именно с появлением концепции больших данных связывают всплеск интереса к DAS-решениям в начале 2010-х годов, после

вытеснения их в 2000-е годы сетевыми решениями классов NAS и SAN.

1.3. Метаданные

Метаданные (от лат. *meta* — цель, конечный пункт, предел, край и данные) — информация о другой информации, или данные, относящиеся к дополнительной информации о содержимом или объекте. Метаданные раскрывают сведения о признаках и свойствах, характеризующих какие-либо сущности, которые позволяют автоматически искать и управлять ими в больших информационных потоках.

Различие между данными и метаданными

Обычно невозможно провести однозначное разделение на данные и **метаданные** в документе, поскольку:

- Что-то может являться как данными, так и метаданными. Так, заголовок статьи можно одновременно отнести как к метаданным (как элемент метаданных — заголовок), так и к собственно данным (поскольку заголовок является частью самого текста).
- Данные и метаданные могут меняться ролями. На стихотворение, рассматриваемое как данные, может быть написана музыка, в этом случае всё стихотворение может быть «прикреплено» к музыкальному файлу и в этом случае рассматриваться как метаданные. Таким образом, отнесение к одной или другой категории зависит от точки зрения (или пространства имён, системы отсчёта).
- Возможно создание мета-мета-…-метаданных (см. аксиома выбора). Поскольку, в соответствии с обычным определением, метаданные являются данными, то можно создать метаданные на метаданные, метаданные для вывода на специальные устройства, либо чтения их описания с использованием программного обеспечения, преобразующего текст в речь.

Другие описательные метаданные могут использоваться автоматизированными рабочими потоками. Например, если некоторая «умная» программа «знает» содержимое и структуру данных, то

данные могут быть автоматически преобразованы и переданы другой «умной» программе как входные данные. В результате, пользователи будут освобождены от необходимости выполнения множества рутинных операций, если данные предоставлены для работы такими «немногословными» программами.

Метаданные становятся важны в World Wide Web по причине необходимости обеспечения поиска полезной информации среди огромного количества доступной. Метаданные, созданные вручную, имеют большую ценность, поскольку это гарантирует осмысленность. Если веб-страница на какую-то определённую тему содержит слово или фразу, то все другие веб-страницы на эту тему могут содержать такое же слово или фразу. Метаданные также обладают разнообразием, поэтому если с какой-то темой связаны два значения, то каждое из них может быть использовано. Например, статья про Живой Журнал может быть обозначена с помощью нескольких значений: «Живой Журнал», «ЖЖ», «LiveJournal».

Метаданные используются для хранения информации о записях audio CD. Аналогично MP3 файлы хранят метаданные в формате ID3.

Редактировать метаданные графических файлов можно в специальных программах для работы с метаданными.

1.3.1. Классификация метаданных

Метаданные можно классифицировать по

- Содержанию. Метаданные могут либо описывать сам ресурс (например, название и размер файла), либо содержимое ресурса (например, «в этом видеофайле показано как парень играет в футбол»).
- По отношению к ресурсу в целом. Метаданные могут относиться к ресурсу в целом или к его частям. Например, «Title» (название фильма) относится к фильму в целом, а «Scene description» (описание эпизода фильма) отдельное для каждого эпизода фильма.
- По возможности логического вывода. Метаданные можно подразделить на **три слоя**: **нижний слой** — это «сырые» данные сами по себе; **средний слой** — метаданные,

описывающие указанные «сырые» данные; и **верхний слой** — метаданные, которые позволяют делать логический вывод, используя второй слой.

Тремя наиболее используемыми классами метаданных являются:

- ***Внутренние метаданные***, описывающие структуру или составные части вещи, то, чем вещь является. Например, формат и размер файла.
- ***Административные метаданные***, требующиеся для процессов обработки информации, назначение вещи. Например, информация об авторе, редакторе, дата публикации и т. п.
- ***Описательные метаданные***, которые описывают природу вещи, её признаки. Например, набор связанных с информацией категорий, ссылки на другие вещи, связанные с данной.

1.3.2. Формат метаданных

Метаданными на практике обычно называют данные, представленные в соответствии с одним из форматов метаданных.

Формат метаданных — представляет собой стандарт, предназначенный для формального описания некоторой категории ресурсов (объектов, сущностей и т. п.). Такой стандарт обычно включает в себя набор **полей (атрибутов, свойств, элементов метаданных**), позволяющих характеризовать рассматриваемый объект. Например, формат MARC позволяет описывать книги (и не только книги), содержит поля для описания названия, автора, тематики и огромного множества других характеристик (формат MARC позволяет описать сотни характеристик).

Форматы можно классифицировать, во-первых, по охвату и подробности типов описываемых ресурсов. Во-вторых, по ширине и подробности области описания ресурсов и мощности структуры элементов метаданных. Кроме этого, можно классифицировать по предметным областям, или целям разработки и использования формата метаданных.

Форматы метаданных часто разрабатываются международными организациями или консорциумами, включающими в себя заинтересованные в появлении стандарта государственные организации и частные компании. Разработанный формат часто закрепляется как стандарт в одной или нескольких организациях, занимающихся разработкой и принятием стандартов (например W3C, ISO, ANSI и т. п.).

Классификация форматов метаданных по описываемой предметной области:

- DCMI является одним из наиболее распространённых в интернет форматов метаданных для описания ресурсов любого типа (как электронных документов, так и реальных физических объектов). Другие форматы метаданных, предназначенные для описания архивов и электронных ресурсов GILS, EAD.
- для описания персон и организаций vCard и FOAF
- для описания библиографических ресурсов предназначены форматы семейства MARC (MARC 21, используемый в США и Великобритании, и UNIMARC, используемый в Европе и Азии); UNIMARC в свою очередь подразделяется на национальные расширения этого формата (так, в России используется RUSMARC); в силу большой сложности форматов семейства MARC для решения задач интеграции данных был разработан формат MODS.
- для описания музейных и исторических ценностей используется формат CDWA
- для описания издательской продукции используются PRISM и ONIX
- для кристаллографической информации CIF
- для работы с изображениями со спутников VICAR
- для описания новостей NewsXML

1.4. Пространственные данные

Пространственные данные (**географические данные**, **геоданные**) — данные о пространственных объектах и их наборах. Пространственные данные составляют основу информационного обеспечения геоинформационных систем.

Совокупность пространственных данных, записанных (сохранённых) тем или иным образом, называется пространственной базой данных (англ. *spatial database*). Современные пространственные БД организовываются на платформе специализированного программного обеспечения, позволяющего сохранять, накапливать и обрабатывать (включая, пространственный анализ) все компоненты пространственных данных в виде логически единой БД.

Большинство современных СУБД поддерживают т. н. пространственные расширения — геометрические типы данных и пространственные индексы. К примеру, MySQL поддерживает следующие типы данных:

- Точка (Point)
- Отрезок (Linestring)
- Многоугольник (Polygon)
- Набор точек (MultiPoint)
- Набор отрезков (MultiLinestring)
- Набор многоугольников (MultiPolygon)
- Коллекция типов (GeometryCollection)

Геометрические примитивы хранятся в БД в бинарном виде, однако могут быть представлены в текстовом виде в формате Well-Known Text (WKT), например:

```
POINT(15 20)
LINESTRING(0 0, 10 10, 20 25, 50 60)
POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7,
5 5))
MULTILINESTRING((10 10, 20 20), (15 15, 30 15))
GEOMETRYCOLLECTION(POINT(10 10), POINT(30 30),
LINESTRING(15 15, 20 20))
```

Структура пространственных данных

Пространственные данные обычно состоят из двух взаимосвязанных частей: **координатных и атрибутивных данных**. Установление связи между этими частями называется геокодированием.

1.4.1. Координатные данные

Координатные данные определяют позиционные характеристики пространственного объекта. Они описывают его местоположение в установленной системе координат.

Геометрически информация, содержащаяся на карте, может быть определена как совокупность наборов точек, линий, контуров и площадей, имеющих метрические значения, отражающие трехмерную реальность. Эта информация образует *класс* ***координатных данных*** ГИС.

Такое определение широты не годится для эллипсоида**.**

Основные типы координатных моделей

Класс координатных моделей можно разбить на типы. При этом следует учесть, что попытка включить в описание широкий набор групп приводит к усложнению базы данных и процессов обработки. В ГИС используют меньшее число атомарных моделей по сравнению с САПР.

В ГИС, как и в САПР, применяют набор базовых геометрических типов моделей, из которых создают все остальные, более сложные. С учетом предметной области карт ограничиваются лишь описанием таких типов (**структур географических данных**), которые относятся к представлению топографии и к тематическому упорядочению.

В ГИС включают следующие основные типы координатных данных (рис. 1):

• точка (узлы, вершины);

• линия незамкнутая;

• контур (замкнутая линия);

• полигон (ареал, район) - группы примыкающих друг к другу замкнутых участков.

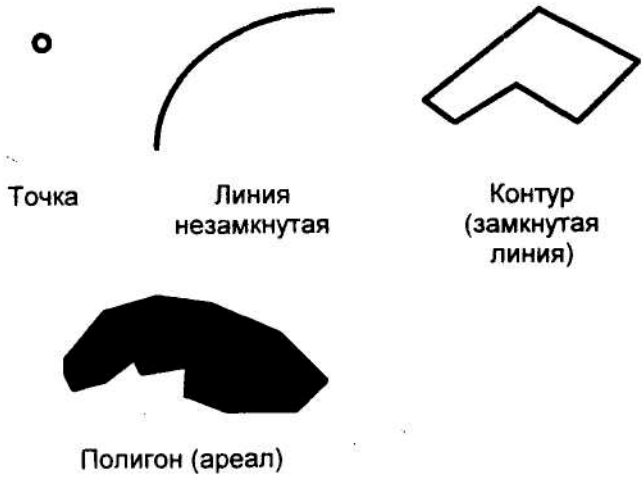


Рис. 1. Основные типы координатных данных

В некоторых системах в описание основных типов моделей включают понятие *пространственная сеть,* которая является развитием типа данных *район.* Контуры и линии часто объединяют общим термином - "линейные объекты". Таким образом, в разных ГИС число основных

типов координатных моделей меняется от трех до пяти. Проводя сравнение с технологиями САПР, отметим, что основные типы координатных данных являются *аналогами* атомарных моделей в САПР, которые называют *примитивами.*

Приведенные выше понятия носят концептуальный характер. На практике для построения реальных объектов используют большее число составных координатных моделей. В разных ГИС они незначительно отличаются, поэтому рассмотрим в качестве примера набор данных в системе ГеоДраф:

• точка - пара координат X, Y;

• отрезок - линия, соединяющая две точки;

• вершина (вертекс) - начальная или конечная точка отрезка;

• дуга (линия) - упорядоченный набор связных отрезков (или вершин);

• узел - начальная или конечная вершина дуги;

• висячий узел - узел, принадлежащий только одной дуге, у которой начальная и конечная вершины не совпадают;

• псевдоузел - узел, принадлежащий только двум дугам либо одной замкнутой дуге, у которой начальная и конечная вершины совпадают. Исключением является узел, принадлежащий двум дугам, одна из которых самозамкнута в этом узле, а другая примыкает к ней (такой узел является нормальным);

• нормальный узел - узел, принадлежащий трем (и более) дугам. Нормальным также является узел, принадлежащий двум дугам, одна из которых самозамкнута в этом узле, а другая примыкает к ней;

• висячая дуга - дуга, имеющая висячий узел;

• замкнутая дуга - дуга, у которой совпадают начальная и конечная вершины (у такой дуги имеется только один узел);

• полигон - единичная область, ограниченная (находящаяся внутри) замкнутой дугой или упорядоченным набором связных дуг, которые образуют замкнутый контур;

• покрытие - набор файлов, фиксирующий в виде цифровых записей пространственные объекты (точки, дуги , полигоны) и структуру отношений между ними;

• пустое покрытие - покрытие, в котором отсутствуют пространственные объекты;

• слой - покрытие, рассматриваемое в контексте его содержательной определенности (растительность, рельеф, административное деле-йие и т.п.) или его статуса в среде редактора (активный слой, пассивный слой);

• внутренний идентификатор пространственного объекта - целое число, являющееся служебным идентификатором системы (уникальное для йждого объекта данного покрытия и назначаемое автоматически в процессе работы редактора). Может изменяться системой в процессе работы;

• пользовательский идентификатор (внутренний ключ) пространственного объекта - целое число, служащее для связи объектов цифровой карты с базой (таблицами) тематических данных. Назначается и изменяется только пользователем.

На рис. 2 показаны основные из рассмотренных элементы векторных данных ГИС.

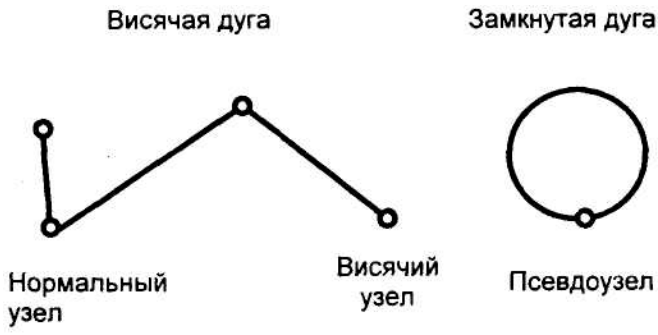


Рис. 2. Основные элементы векторных данных ГИС

Точечные объекты. Простейший тип пространственного объекта задают точечные данные, к которым относятся не только точки, но и все точечные условные знаки. Выбор объектов, представляемых в виде точек, зависит от масштаба карты или исследования. Например, на крупномасштабной карте точками показываются отдельные строения, а на мелкомасштабной карте - города.

Особенность точечных объектов состоит в том, что они хранятся и в виде графических файлов, как другие пространственные объекты, и в виде таблиц, как атрибуты. Последнее обусловлено тем, что координаты каждой точки описывают как два дополнительных атрибута. В силу этого информацию о наборе точек можно представить в виде развернутой таблицы или таблицы, содержащей помимо координат наборы атрибутов (идентификационные номера, тематические характеристики и т.д.). В таких таблицах каждая строка соответствует точке - в ней собрана вся информация о данной точке. Каждый столбец - это признак, содержащий типизированные данные: координаты или атрибуты. Каждая точка независима от всех остальных точек, представленных отдельными строками.

Линейные объекты. Они широко применяются для описания сетей, для которых в отличие от точечных объектов характерно присутствие топологических признаков.

Любая сеть состоит из узлов (вершин) - соединений, концов обособленных линий и звеньев (дуг) - цепей в модели базы данных.

Для каждого узла существует специальная характеристика, называемая ***валентностью,*** определяемая количеством звеньев в нем. Концы обособленных линий одновалентны. Для уличных сетей (пересечения типа "крест") наиболее характерны четырехвалентные узлы. В гидрологии чаще всего встречаются трехвалентные узлы.

В древовидной сети (Е-дерево) каждая пара узлов имеет лишь одно соединение, не допускаются петли и замкнутые контуры, большая часть речных сетей имеет древовидную структуру.

Линейные объекты, как и точечные, имеют свои атрибуты, причем разные для дуг (звеньев) и узлов. Атрибутами для дуг являются:

- направление движения, интенсивность движения, протяженность;
- количество полос, время пути вдоль звена;
- диаметр трубы, направление движения газа;
- напряжение в ЛЭП, высота опор;
- количество путей, уклон, ширина тоннеля, грузоподъемность и др. Атрибуты для узла:
- наличие перехода, названия пересекающихся улиц;
- наличие автоматического регулирования перекрестков;
- тип (ручной или автоматический) перевода стрелок;
- характеристики трансформаторов ЛЭП;
- мощность компрессора.

Некоторые атрибуты (например, названия пересекающихся улиц) служат для связи одного типа объектов с другими (узлы со звеньями), другие характеризуют только участки звеньев сети.

Во многих ГИС для включения дополнительных атрибутов в сеть необходимо разбиение существующих звеньев и создание новых узлов. Например, звено улицы, часть которой ремонтируется, разрывается на месте начального и конечного участка ремонта, его атрибуты присваиваются новому (двухвалентному) узлу. Другой пример: для отрезка дороги, проходящей через мост, создаются новое звено и два новых узла. Такой подход может привести к появлению недопустимо большого числа звеньев и двухвалентных узлов, поэтому он имеет ограничение, определяемое ресурсами конкретной ГИС.

Сети часто используют как системы линейной адресации. В этих случаях точки размещают в сети по данным о номере звена и о расстоянии от его начала. Это более удобно, чем использовать *X, Y* координаты **точки из** таблицы, поскольку такие данные непосредственно указывают положение точки в сети.

Данный подход определяет метод присвоения атрибутов отдельным участкам звеньев. При этом линейные объекты (здания, тоннели) хранятся в отдельных таблицах, а с сетью они увязаны путем указания номера звена и расстояния от его начала.

Для точечных объектов необходимо указать одно значение координат, для линейных - два (для начальной и конечной точек). Это позволяет при необходимости рассчитать *X, Y* координаты этих объектов и исключает необходимость дробить звенья и вводить двухвалентные узлы.

Ареалы. В настоящее время в ГИС может быть представлено несколько типов ареалов: зоны в приложении к окружающей среде или природным ресурсам, социально-экономические зоны, данные об угодьях и др.

Для ареальных объектов границы могут определяться свойством или явлением, а также независимо от явления (затем перечисляются значения атрибутов). Кроме того, границы могут устанавливаться искусственно, например для микрорайонов.

Взаимосвязи между координатными моделями

В общем случае пространственные данные могут иметь большое число разнообразных связей. Эти связи играют важную роль для пространственного анализа данных. Например, связь типа "содержится в" позволяет соотносить объекты с их окружением, связь "пересекает" между двумя линиями важна для анализа маршрутов в сетях.

Взаимосвязи могут существовать между объектами одного типа или разных типов.

Исходя из критерия построения моделей можно выделить три основных типа взаимосвязей между координатными объектами.

Первый тип - взаимосвязи для построения сложных объектов из простых элементов, например, взаимосвязи между дугой и упорядоченным набором определяющих ее вершин, взаимосвязи между полигоном и упорядоченным набором определяющих его линий. При этом используют процедуры агрегации и обобщения *(см.* разд. 3).

Второй тип - взаимосвязи, которые можно вычислить по координатам объектов. Например, координаты точки пересечения двух линий определяют взаимосвязь типа "скрещивается" и наличие четырехвалентного узла. Табличные координаты отдельной точки и данные о границах полигонов позволяют найти полигон, включающий данную точку. Этим определяется взаимосвязь типа "содержится в". Используя данные о границах полигонов, можно выяснить, перекрываются ли полигоны, и тем самым установить взаимосвязь типа "перекрывает". Другими словами, второй тип связи содержится в атрибутивных данных в неявном виде.

Третий тип - "интеллектуальный". Эти взаимосвязи нельзя вычислить по координатам, они должны получать специальное описание и семантику при вводе данных. Например, можно вычислить пересечение двух линий, но, если этими линиями являются автодороги, нельзя сказать, пересекаются они или в этом месте находится развязка автодорог. Следовательно, для решения дополнительных задач необходима информация о связях. Учет связей происходит при кодировании данных, т.е. в подсистемах семантического моделирования.

1.4.2. Атрибутивные данные

Атрибутивные данные представляют собой совокупность непозиционных характеристик (атрибутов) пространственного объекта. Атрибутивные данные определяют смысловое содержание (семантику) объекта и могут содержать качественные или количественные значения.

Одних координатных данных недостаточно для описания картографической или сложной графической информации. Картографические объекты кроме метрической обладают некоторой присвоенной им опи-

сательной информацией (названия политических единиц, городов и рек). Характеристики объектов, входящие в состав этой информации, называют атрибутами. Совокупность возможных атрибутов определяет класс атрибутивных моделей ГИС.

Выше отмечалось, что атрибутивные данные описывают тематические и временные характеристики. Таблица, содержащая атрибуты объектов, называется ***таблицей атрибутов.***

Атрибуты, соответствующие тематической форме данных и определяющие различные признаки объектов, также хранятся в таблицах. Каждому объекту соответствует строка таблицы, каждому тематическому признаку - столбец таблицы. Каждая клетка таблицы отражает значение определенного признака для определенного объекта.

Временная характеристика может отражаться несколькими способами:

• путем указания временного периода существования объектов;

• путем соотнесения информации с определенными моментами времени;

• путем указания скорости движения объектов. В зависимости от способа отражения временной характеристики она может размещаться в одной таблице или в нескольких таблицах атрибутов данного объекта для различных временных этапов.

Применение атрибутов позволяет осуществлять анализ объектов базы данных с использованием стандартных форм запросов и разного рода фильтров, а также выражений математической логики. Последнее эффективно при тематическом картографировании.

Кроме того, с помощью атрибутов можно типизировать данные и упорядочивать описание для широкого набора некоординатных данных.

Таким образом, атрибутивное описание дополняет координатное, совместно с ним создает полное описание моделей ГИС и решает зада-

чи типизации исходных данных, что упрощает процессы классификации и обработки.

Атрибутами могут быть символы (названия), числа (статистическая информация, код объекта) или графические признаки (цвет, рисунок, заполнения контуров).

Числовые значения в ГИС могут относиться как к координатным данным, так и к атрибутивным. Для пояснения этого напомним, что основной формой представления атрибутивных данных в БД является таблица, а в таблице могут храниться как координаты объектов (координат-ные данные), так и описательные характеристики (атрибутивные данные).

Можно по-разному организовывать взаимосвязь координатного и атрибутивного описания. Например, В. Вебером было предложено специфическое сочетание координатного и атрибутивного классов для описания картографических данных. Для построения общей модели данных ГИС он вводит четырехмерное пространство объекта, где первые дав (плановые) размера присваиваются данным *X, Y,* атрибуты располагаются в третьем измерении, а четвертое измерение резервируется для временных наборов данных.

Такой подход не нов, он заимствован из методов релятивистской механики и теории N-мерных пространств. По Веберу, данные по координате Z следует обрабатывать как атрибуты, помещая их в одну и ту же иггегорию наряду с описательными текстами и значениями.

Существуют различные методы хранения атрибутивной информации в ГИС:

• хранение для всех объектов системы 1-2 стандартных атрибутов;

• хранение таблицы атрибутов, связанных с пространственными объектами, и информации о реляциях;

• хранение ссылок на элементы данных иерархической или сетевой БД;

• хранение атрибутивной информации может вообще не применяться, если система опирается на классификатор.

1.4.3. Вопросы точности координатных и атрибутивных данных

Использование любой информации допустимо, если она удовлетворяет определенным критериям и стандартам. Одним из критериев применимости пространственно-временных данных в системах ГИС является ***точность*** - близость результатов, расчетов или оценок к истинным значениям (или значениям, принятым за истинные). Например, точность горизонтали в цифровой базе данных, полученной на основе дигитализации по карте, можно оценить сравнением ее с горизонталью на исходной карте.

Рассмотрим несколько показателей точности в ГИС: точность вычисления, точность измерения, точность представления.

Точность вычисления определяется количеством значимых цифр после запятой, *точность измерений* - количеством значимых цифр при измерениях, *точность представления* - количеством разрядов, описывающих координатные данные.

Точность вычислений и измерений не адекватна точности представления. Большое количество значимых цифр не всегда гарантирует точность вычислений или измерений.

Точность вычисления в ГИС велика, обычно она намного выше, чем точность самих данных. Более того, набор специальных методов и алгоритмов в ряде случаев позволяет повысить точность первичных измерений.

Точность входит в комплекс данных, определяющий важный показатель - ***качество данных.***

В США разработаны национальные стандарты для цифровых картографических данных, которые применяются при оценке точности цифровых данных. Стандарт выделяет несколько компонентов качества данных:

- позиционную точность;
- точность атрибутов;
- логическую непротиворечивость;

- полноту;
- происхождение.

Позиционная точность выражается степенью отклонения данных ГИС о местоположении от истинного положения объекта на местности. Обычно точность карт приблизительно определяется толщиной линии, или 0,4 мм. Это соответствует 10 м в масштабе 1 : 25 000.

Для проверки позиционной точности используют независимые более точные источники, например карту более крупного масштаба, систему глобального позиционирования (GPS) и др.

Можно на основе известного в статистике правила "переноса ошибок" оценить точность, зная погрешности, вносимые различными источниками. Например, при создании цифровой модели имели место следующие погрешности: 1 мм в исходном материале, 0,4 мм на карте, предназначенной для цифрования, 0,1 мм при цифровании.

При независимой комбинации источников ошибок общую погрешность А Можно оценить, суммируя квадраты отдельных погрешностей и извлекая квадратный корень из суммы:

$$\Delta = \sqrt{1^2 + 0,4^2 + 0,1^2} = 1,08\ (мм)$$

Точность атрибутов определяется близостью значений атрибута к его истинной величине. Атрибуты могут со временем меняться: довольно часто по сравнению с координатными данными.

В зависимости от типов данных точность атрибутов может быть измерена разными способами. Для непрерывных атрибутов (поверхностей), например в полигонах Тиссена, точность выражается как погрешность измерений. Для атрибутов категорий объектов, например классифицированных полигонов, точность зависит от того, являются ли категории подходящими, достаточно подробными и определенными, и от того, какова вероятность наличия в данных грубых ошибок.

Точность атрибута может быть различной в разных частях карты, поэтому полезнее рассчитывать пространственную вариацию

вероятности ошибки в классификации, чем пользоваться обобщенными статистическими показателями.

Понятие ***логической непротиворечивости*** связано с непротиворечивостью данных в базах данных.

В среде ГИС это понятие распространяется на внутреннюю непротиворечивость структур данных и внутреннюю топологическую непротиворечивость векторных данных. В частности, это определяет такие требования, как замкнутость полигонов, уникальность идентификатора полигона, наличие или отсутствие узлов на пересечениях дуг.

Понятие ***полноты*** (достаточности) данных связано со степенью охвата данными множества соответствующих объектов. В зависимости от правил отбора, генерализации и масштаба определяют число соответствующих объектов для полного описания ситуации, картографической композиции, явления и т.п. .

Несколько специфический показатель ***происхождение*** включает сведения об источниках данных и операциях по созданию базы данных, о методах кодирования данных, времени сбора данных, методе обработки данных, точности результатов вычислений и т.п.

2. Типы данных

Тип данных (*тип*) — множество значений и операций на этих значениях (IEEE Std 1320.2-1998).

Другие определения:

- Тип данных — класс данных, характеризуемый членами класса и операциями, которые могут быть к ним применены (ISO/IEC/IEEE 24765-2010).
- Тип данных — категоризация абстрактного множества возможных значений, характеристик и набор операций для некоторого атрибута (IEEE Std 1320.2-1998).
- Тип данных — категоризация аргументов операций над значениями, как правило, охватывающая как поведение, так и представление (ISO/IEC 19500-2:2003).
- Тип данных — допустимое множество значений.

Тип определяет возможные значения и их смысл, операции, а также способы хранения значений типа. Изучается теорией типов.

2.1. Элементы теории типов

В математике, логике и компьютерных науках **теорией типов** считается какая-либо формальная система, являющаяся альтернативой наивной теории множеств, сопровождаемая классификацией элементов такой системы с помощью типов, образующих некоторую иерархию. Также под **теорией типов** понимают изучение подобных формализмов.

Теория типов — математически формализованная база для проектирования, анализа и изучения систем типов данных в теории языков программирования (раздел информатики). Многие программисты используют это понятие для обозначения любого аналитического труда, изучающего системы типов в языках программирования. В научных кругах под **теорией типов** чаще всего понимают более узкий раздел дискретной математики, в частности λ-исчисление с типами.

Современная теория типов была частично разработана в процессе разрешения парадокса Рассела и во многом базируется на работе Бертрана Рассела и Альфреда Уайтхеда «Principia mathematica».

Доктрина типов

Доктрина типов восходит к Б. Расселу, согласно которому всякий тип рассматривается как диапазон значимости пропозициональной (высказывательной) функции. Помимо того, считается, что у всякой функции имеется тип (её домен, область определения). В доктрине типов соблюдается выполнимость *принципа замены типа (высказывания) на дефинициально эквивалентный тип (высказывание)*.

Теория типов в логике

В основе этой теории лежит принцип иерархичности. Это означает, что логические понятия — высказывания, индивиды, пропозициональные функции — располагаются в иерархию типов. Существенно, что

произвольная функция в качестве своих аргументов имеет лишь те понятия, которые предшествуют ей в иерархии.

Некоторая теория типов

Под **некоторой** теорией типов обычно понимают прикладную логику высших порядков, в которой имеется тип *N* натуральных чисел и в которой выполняются аксиомы арифметики Пеано.

Разветвлённая теория типов

Разветвлённая теория типов (*Ramified Theory of Types*, ***RTT***)

Интуиционистская теория типов

Интуиционистскую теорию типов (*Intuitionistic Type Theory*, ***ITT***) построил Пер Мартин-Лёф.

Чистые системы типов

Теория чистых систем типов (англ. *pure type systems*, ***PTS***) обобщает все исчисления лямбда-куба и формулирует правила, позволяющие вычислить их как частные случаи. Её независимо построили Берарди (*Berardi*) и Терлоу (*Terlouw*). Чистые системы типов оперируют *только* понятием типа, рассматривая все понятия других исчислений только в виде типов — потому они и называются «*чистыми*». Не производится разделения между термами и типами, между различными слоями (т.е. родá типов также называются типами, только относящимися к другой вселенной), и даже сами слои называются не сортами, а типами (точнее, вселенными типов). В общем виде, чистая система типов задаётся понятием спецификации, пятью жёсткими правилами и двумя гибкими (меняющимися от системы к системе). Спецификация чистой системы типов представляет собой тройку (S,A,R), где S — множество сортов (**S**orts), A — множество аксиом (**A**xioms) над этими сортами и R — множество правил (**R**ules).

Многомерные теории типов

Теории типов высших размерностей (англ. *higher-dimensional type theories* или просто *higher type theories*, ***HTT***) обобщают традиционные

теории типов, разрешая устанавливать нетривиальные отношения равенства между типами. Например, если взять множество пар (декартовых произведений) натуральных чисел `nat × nat` и множество функций, возвращающих натуральное число `nat -> nat`, то нельзя утверждать, что элементы этих множеств попарно равны, но можно утверждать, что эти множества эквивалентны. Изоморфизмы между типами и изучаются в двухмерной, трёхмерной и т.д. теориях типов. Весь необходимый базис для формулировки этих теорий был заложен ещё Жираром — Рейнольдсом, но сами теории были сформулированы много позже.

Неотъемлемой частью большинства языков программирования являются системы типов, использующие типы для обеспечения той или иной степени типобезопасности.

2.2. Система типов

Система типов — совокупность правил в языках программирования, назначающих свойства, именуемые типами, различным конструкциям, составляющим программу — таким как переменные, выражения, функции или модули. Основная роль системы типов заключается в уменьшении числа багов (в программировании **баг** — жаргонное слово, обычно обозначающее ошибку в программе) в программах посредством определения интерфейсов между различными частями программы и последующей проверки согласованности взаимодействия этих частей. Эта проверка может происходить статически (на стадии компиляции) или динамически (во время выполнения), а также быть комбинацией обоих видов.

По Пирсу, **система типов** — разрешимый синтаксический метод доказательства отсутствия определённых поведений программы путём классификации конструкций в соответствии с видами вычисляемых значений.

2.2.1. Описание

Пример простой системы типов можно видеть в языке Си. Части программы на Си оформляются в виде определений функций. Функции вызывают друг друга. Интерфейс функции задаёт имя функции и список значений, которые передаются её телу. Вызывающая функция

постулирует имя функции, которую хочет вызвать, и имена переменных, хранящих значения, которые требуется передать. Во время исполнения программы значения помещаются во временное хранилище, и затем исполнение передаётся в тело вызываемой функции. Код вызываемой функции осуществляет доступ к значениям и использует их. Если инструкции в теле функции написаны в предположении, что им на обработку должно быть передано целое значение, но вызывающий код передал число с плавающей точкой, то вызванная функция вычислит неверный результат. Компилятор Си проверяет типы, заданные для каждой переданной переменной, в отношении к типам, заданным для каждой переменной в интерфейсе вызываемой функции. Если типы не совпадают, компилятор порождает ошибку времени компиляции.

Технически, система типов назначает тип каждому вычисленному значению и затем, отслеживая последовательность этих вычислений, предпринимает попытку проверить или доказать отсутствие ошибок согласования типов. Конкретная система типов может определять, что именно приводит к ошибке, но обычно целью является предотвращение того, чтобы операции, предполагающие определённые свойства своих параметров, получали параметры, для которых эти операции не имеют смысла — предотвращение **логических ошибок**. Дополнительно это также предотвращает ошибки адресации памяти.

Системы типов обычно определяются как часть языков программирования и встраиваются в их интерпретаторы и компиляторы. Однако система типов языка может быть расширена посредством специальных инструментов, осуществляющих дополнительные проверки на основе исходного синтаксиса типизации в языке.

Компилятор также может использовать статический тип значения для оптимизации хранилища и для выбора алгоритмической реализации операций над этим значением. Например, во многих компиляторах Си тип `float` представляется 32 битами, в соответствии со спецификацией IEEE для операций с плавающей точкой одинарной точности. На основании этого будет использоваться специальный набор инструкций микропроцессора для значений этого типа (сложение, умножение и другие операции).

Количество налагаемых на типы ограничений и способ их вычисления определяют типизацию языка. Помимо этого, в случае полиморфизма типов, язык может также задать для каждого типа операцию варьирования конкретных алгоритмов. Изучением систем типов занимается теория типов, хотя на практике конкретная система типов языка программирования основывается на особенностях архитектуры компьютера, реализации компилятора и общем образе языка.

2.2.2. Формальное обоснование

Формальным обоснованием для систем типов служит теория типов. В состав языка программирования включается *система типов* для осуществления проверки типов во время компиляции или во время выполнения, требующая явного провозглашения типов или выводящая их самостоятельно. Марк Мэнесси (англ. *Mark Manasse*) сформулировал проблему так:

Основная проблема, решаемая теорией типов, состоит в том, чтобы убедиться, что программы являются осмысленными. Основная проблема, порождаемая теорией типов, состоит в том, что осмысленные программы могут не соответствовать поведению, ожидаемому от них по замыслу. Следствием этой напряжённости является поиск более мощных систем типов.

Операция назначения типа, называемая типизацией, придаёт смысл цепочкам бит, таким как значение в памяти компьютера, или объектам, таким как переменная. Компьютер не имеет возможности отличить, к примеру, адрес в памяти от инструкции кода, или символ от целого числа или числа с плавающей запятой, поскольку цепочки бит, представляющие эти различные по смыслу значения, не имеют каких-либо явных особенностей, позволяющих делать предположения об их смысле. Назначение цепочкам бит типа предоставляет это осмысление, превращая тем самым программируемое аппаратное обеспечение в символьную систему, состоящую из этого аппаратного обеспечения и программы.

2.2.3. Проверка согласования типов

Процесс проверки и установления ограничений для типов — контроль типов или проверка соответствия типов — может осуществляться как

на стадии компиляции (статическая типизация), так и во время выполнения (динамическая типизация). Если спецификация языка требует, чтобы правила типизации исполнялись строго (то есть допуская в той или иной мере лишь те автоматические преобразования типов, которые не теряют информацию), такой язык называется *сильно типизированным* (англ. *strongly typed*; в русской литературе преобладает вариант перевода *строго типизированным*), в противном случае — *слабо типизированным*. Эти термины являются условными и не используются в формальных обоснованиях.

Статическая проверка типов

Стати́ческая типиза́ция — приём, широко используемый в языках программирования, при котором переменная, параметр подпрограммы, возвращаемое значение функции связывается с типом в момент объявления и тип не может быть изменён позже (переменная или параметр будут принимать, а функция — возвращать значения только этого типа). Примеры статически типизированных языков — Ада, C++, C#, D, Java, ML, Паскаль, Nim.

Противоположный приём — динамическая типизация.

Некоторые статически типизированные языки позже получили возможность также использовать динамическую типизацию при помощи специальных подсистем. Например, `Variant` в Delphi, пакеты в AliceML[1], `Data.Dynamic` в Haskell[2].

- Статическая типизация даёт самый простой машинный код. Поэтому она удобна для языков, дающих исполняемые файлы ОС или JIT-компилируемые промежуточные коды.
- Многие ошибки исключаются уже на стадии компиляции.
- Поэтому статическая типизация хороша для написания сложного, но быстрого кода.
- В интегрированной среде разработки осуществимо более релевантное автодополнение, особенно если типизация — сильная статическая: множество вариантов можно отбросить как не подходящие по типу.
- Чем больше и сложнее проект, тем большее преимущество дает статическая типизация, и наоборот.

Недостатки

- Языки с недостаточно проработанной математической базой оказываются довольно многословными: каждый раз надо указывать, какой тип будет иметь переменная. В некоторых языках есть автоматическое выведение типа, однако оно может привести к трудноуловимым ошибкам. Нивелируется функциями IDE вроде quick fix.
    - Сказанное не верно для языков семейства ML, основанных на т. н. «главной типизации» (англ. *principal typing scheme*), которая одновременно автоматически выводит наиболее общий тип всякого выражения и строго проверяет согласование типов зависимостей. Это придаёт языку выразительность динамически типизируемых, но обеспечивает лучшее быстродействие и типобезопасность.
- Тяжело работать с данными из внешних источников (например, в реляционных СУБД / десериализация данных).

Динамическая проверка типов и информация о типах

Динами́ческая типиза́ция — приём, широко используемый в языках программирования и языках спецификации, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов. Примеры языков с динамической типизацией — Smalltalk, Python, Objective-C, Ruby, PHP, Perl, JavaScript, Lisp, xBase, Erlang.

Противоположный приём — статическая типизация.

В некоторых языках со слабой динамической типизацией стоит проблема сравнения величин, так, например, PHP имеет операции сравнения «==», «!=» и «===», «!==», где вторая пара операций сравнивает и значения, и типы переменных. Операция «===» даёт true только при полном совпадении, в отличие от «==», который считает верным такое выражение: (`1=="1"`). Стоит отметить, что это проблема не динамической типизации в целом, а конкретных языков программирования.

Преимущества

- Упрощается написание несложных программ, например, скриптов.

(**Сценарный язык** (**язык сценариев**, жарг. ***скриптовый язык***; англ. *scripting language*) — высокоуровневый язык сценариев (англ. *script*) — кратких описаний действий, выполняемых системой. Разница между программами и сценариями довольно размыта. Сценарий — это программа, имеющая дело с готовыми программными компонентами).

- Облегчается работа прикладного программиста с СУБД, которые принципиально возвращают информацию в «динамически типизированном» виде. Поэтому динамические языки ценны, например, для программирования веб-служб.
- Иногда требуется работать с данными переменного типа. Например, функция поиска подстроки возвращает позицию найденного символа (число) или маркер «не найдено». В PHP этот маркер — булевое `false`. В языках со статической типизацией это особая константа (0 в Паскале, `std::string::npos` в C++).

Недостатки

- Статическая типизация позволяет уже при компиляции заметить простые ошибки «по недосмотру». Для динамической типизации требуется как минимум выполнить данный участок кода.
- В объектно-ориентированных языках не действует, либо действует с ограничениями, автодополнение: трудно или невозможно понять, к какому типу относится переменная, и вывести набор её полей и методов.
- Развитая статическая система типов (такая как Хиндли-Милнер) играет значительную роль в самодокументировании программы; динамическая типизация по определению не проявляет этого свойства, что затрудняет разработку структурно сложных программ.
- Снижение производительности из-за трат процессорного времени на динамическую проверку типа, и излишние расходы

памяти на переменные, которые могут хранить «что угодно». К тому же большинство языков с динамической типизацией интерпретируемые, а не компилируемые.

Примеры

PHP

```
$res  = "string1"; echo $res."\n";  // выводит
"string1" - переменная имеет строковый тип.
$res  = 1;         echo $res."\n";  // выводит "1"
- переменная преобразуется в целочисленный тип.
$res += 2;         echo $res."\n";  // выводит "3"
- результат операции: целочисленная переменная.
$res .= "string2"; echo $res."\n";  // выводит
"3string2" - переменная преобразуется в строковый
тип и выполняется конкатенация.
```

Python

```
var = "string1"
print(var) # Выведет "string1"
var = 1
print(var) # Выведет "1"
var += 2
print(var) # Выведет "3"

# Однако, такой ход в Python невозможен:
var += "string2" # Порождается исключение
TypeError: unsupported operand type(s) for +=:
'int' and 'str'

var = str(var)   # и потребуется явное
преобразование типов
var += "string2" # теперь ОК
print(var) # Выведет "3string2"
```

JavaScript

```
var res = "string1"; alert(res);  // выводит
'string1'
res = 1;             alert(res);  // выводит 1
res += 2;            alert(res);  // выводит 3
```

```
res += 'string2';    alert(res);  // выводит
'3string2'
```

Object Pascal

```
program Project2;
{$APPTYPE CONSOLE}
Uses   SysUtils;
Var    V1, V2: Variant;
begin
    V1  := 'string1';  WriteLn(V1);           //
выводит "string1"
    V2  := 1;          WriteLn(V2);           //
выводит "1"
    Inc(V2,2);         WriteLn(V2);           //
выводит "3"
                       WriteLn(V2,'string2'); //
выводит "3string2"
end.
```

Object Pascal: Другие способы использования динамической типизации

```
procedure TForm1.Myproc(Obj: TObject);
begin
If (Obj is TButton)
   then (Obj as TButton).Click;
end;
function Something (A: array of const)
begin
  // ...
end;
{Вызов: } Something ( [5,'Hello',3.14159, True,
TForm] );

procedure TForm1.DisplayValue(const AValue:
TValue);
begin
Memo1.Lines.Append(AValue.ToString);
end;

procedure TForm1.btn1Click(Sender: TObject);
var
list: TStrings;
```

```
begin
 list := TStringList.Create();
 list.Text := 'Foo';
 try
   DisplayValue(list);
   DisplayValue(list.Count);
   DisplayValue(list.Capacity * 8.964);
   DisplayValue(list is TStringList);
   DisplayValue(list.Text);
 finally
   list.Free;
 end;
end;
{Вывод:
(TStringList @ 0166E460)
1
35,856
True
Foo
}
```

Lua

```
var="string" -- Переменная строкового типа
var={} -- Переменная стала таблицей

oldprint=print
print=1 -- Такое тоже возможно!
print("somestring") -- attempt to call a nil value
oldprint("somestring") -- somestring
```

2.3. Числовые данные

2.3.1. Целочисленный тип данных

Целое, **целочисленный тип данных** (англ. *Integer*), в информатике — один из простейших и самых распространённых типов данных в языках программирования. Служит для представления целых чисел.

Множество чисел этого типа представляет собой конечное подмножество бесконечного множества целых чисел, ограниченное максимальным и минимальным значениями.

В программировании различают беззнаковые целые числа и целые числа со знаком. Знак числа обычно кодируется старшим битом машинного слова. Традиционно, если старший бит равен 1, то число считается отрицательным, только, если оно не определено как беззнаковое.

Количество чисел в машинном изображении множества целых чисел зависит от длины машинного слова, обычно выражаемой в битах. Например, при длине машинного слова 1 байт (8 бит) диапазон представимых целых чисел со знаком от -128 до 127. В беззнаковом формате байтовое представление числа будет от 0 до 255 (2^8 - 1). Если используется 32-разрядное машинное слово, то целое со знаком будет представлять значения от −2 147 483 648 (-2^{31}) до 2 147 483 647 (2^{31}–1); всего $1\ 0000\ 0000_{16}$ ($4\ 294\ 967\ 296_{10}$) возможных значений.

Ограничение длины машинного слова, обусловленное конкретной аппаратной реализацией того или иного компьютера, не препятствие для обработки ими весьма длинных в битах представлений целых чисел, достигаемое усложнением программных алгоритмов. Естественное ограничение - конечность ёмкости памяти и разумное время на исполнение.

Целые числа и вычисления с целыми числами в современных компьютерах имеют очень важное значение (в подавляющем количестве приложений занимают меньше ресурсов процессора, чем, арифметика с плавающей точкой). Вся адресная арифметика и операции с индексами массивов основаны на целочисленных операциях.

Представление

В памяти типовой компьютерной системы целое число представлено в виде цепочки битов фиксированного (кратного 8) размера. Эта последовательность нулей и единиц — не что иное, как двоичная запись числа, поскольку обычно для представления чисел в современной компьютерной технике используется позиционный двоичный код. Диапазон целых чисел, как правило, определяется

количеством байтов в памяти компьютера, отводимых под одну переменную.

Многие языки программирования предлагают выбор между **короткими** (англ. *short*), **длинными** (англ. *long*) и целыми стандартной длины. Длина *стандартного целого типа*, как правило, совпадает с размером машинного слова на целевой платформе. Для 16-разрядных операционных систем — этот тип (int) составляет 2 байта и совпадает с типом short int (можно использовать как short, опуская слово int), для 32-разрядных операционных систем он будет равен 4 байтам и совпадает с длинным целым long int (можно использовать как long, опуская слово int), и в этом случае будет составлять 4 байта. Короткое целое short int, для 16-разрядных операционных систем, 32-разрядных операционных систем, и для большинства 64-разрядных операционных систем составляет — 2 байта. Также в некоторых языках может использоваться тип данных двойное длинное long long, который составляет 8 байт.

Для 64-разрядных операционных систем учитывая разность моделей данных (LP64, LLP64, ILP64), представление целого типа на разных моделях данных может отличаться между собой. Тип int и long может составлять как 4, так и 8 байт.

Стоит отметить, что каждый язык программирования реализует свою сигнатуру представления целых чисел, которая может отличатся от международных стандартов, но обязана его/их поддерживать. К примеру можно отнести кросс-платформенную библиотеку Qt, где целое представляется типом qintX и quintX, где X-8,16,32,64.

Целые типы подразделяются на *беззнаковые* (англ. *unsigned*) и *знаковые* (англ. *signed*).

Беззнаковые целые

Беззнаковые целые представляют только неотрицательные числа, при этом все разряды кода используются для представления значения числа и максимальное число соответствует единичным значениям кода во всех разрядах: 111…111. m-байтовая переменная целого типа без знака, очевидно, принимает значения от 0 до $+2^{8m}-1$.

В C и C++ для обозначения беззнаковых типов используется префикс `unsigned`. В C# в качестве показателя беззнаковости используется префикс `u` (*англ. unsigned*). Например, для объявления беззнакового целого, равного по размеру одному машинному слову используется тип `uint`.

Беззнаковые целые, в частности, используются для адресации памяти, представления символов.

Иногда в литературе встречаются рекомендации не использовать тип беззнаковые целые, поскольку он может быть не реализован процессором компьютера, однако вряд ли этот совет следует считать актуальным — большинство современных процессоров (в том числе x86-совместимые) одинаково хорошо работают как со знаковыми, так и с беззнаковыми целыми.

В некоторых языках, например Java, беззнаковые целые типы (за исключением символьного) отсутствуют.

Неправильное использование беззнаковых целых может приводить к неочевидным ошибкам из-за возникающего переполнения. В приведённом ниже примере использование беззнаковых целых в цикле в C и C++ превращает этот цикл в бесконечный:

```
char ar[N];
for (unsigned int i = N-1; i >= 0; --i){
      ar[i] = i;
}
```

Целые со знаком

Существует несколько различных способов представления целых значений в двоичном коде в виде величины со знаком (англ.)русск.. В частности можно назвать прямой и обратный коды. Знак кодируется в старшем разряде числа: 0 соответствует положительным, а 1 отрицательным числам.

Могут быть использованы и более экзотические представления отрицательных чисел, такие, как, например, система счисления по основанию −2.

Однако для большинства современных процессоров обычным представлением чисел со знаком является дополнительный код. Максимальное положительное число представляется двоичным кодом 0111…111, максимальное по модулю отрицательное кодом 1000…000, а код 111…111 соответствует −1. Такое представление чисел соответствует наиболее простой реализации арифметических логических устройств процессора на логических вентилях и позволяет использовать один и тот же алгоритм сложения и вычитания как для беззнаковых чисел, так и для чисел со знаком (отличие — только в условиях, при которых считается, что наступило арифметическое переполнение).

m-байтовая переменная целого типа со знаком принимает значения от -2^{8m-1} до $+2^{8m-1}-1$.

Предельные значения для разных разрядностей

Ниже представлена таблица предельных значений десятичных чисел для разных разрядностей при кодировании дополнительным кодом. В столбце «Максимальное десятичное» сначала идёт максимальное значение целого без знака, а под ним минимальное и максимальное целое со знаком.

Битов	Байтов	Дв. слов	Максимальное десятичное	Дес. цифр
4	½	⅛	15	2
			-8	1
			+7	1
8	1	¼	255	3
			-128	3
			+127	3
16	2	½	65535	5
			-32768	5
			+32767	5
24	3	¾	16777215	8
			-8388608	7
			+8388607	7
32	4	1	4294967295	10
			-2147483648	10
			+2147483647	10

48	6	1½	281474976710655 -140737488355328 +140737488355327	15 15 15
64	8	2	18446744073709551615 -9223372036854775808 +9223372036854775807	20 19 19
96	12	3	79228162514264337593543950335 -39614081257132168796771975168 +39614081257132168796771975167	29 29 29
128	16	4	340282366920938463463374607431768211455 -170141183460469231731687303715884105728 +170141183460469231731687303715884105727	39 39 39
256	32	8	115792089237316195(...)584007913129639935 -57896044618658097(...)792003956564819968 +57896044618658097(...)792003956564819967	78 77 77
512	64	16	13407807929942597099(...)946433649006084095 -6703903964971298549(...)973216824503042048 +6703903964971298549(...)973216824503042047	155 154 154
1024	128	32	179769313486231590(...)356329624224137215 -89884656743115795(...)678164812112068608 +89884656743115795(...)678164812112068607	309 308 308
2048	256	64	32317006071311007(...)853611059596230655 -	617 617 617

			16158503035655503(...)926805529798115328	
			+16158503035655503(...)92680552979811532 7	
			1044388881413152506(...)708340403154190335	
			-	1234
4096	512	128	522194440706576253(...)3541702015770951 68	1233 1233
			+522194440706576253(...)354170201577095167	
			1090748135619415929(...)505665475715792895	
			-	2467
8192	1024	256	545374067809707964(...)2528327378578964 48	2466 2466
			+545374067809707964(...)252832737857896447	
			1189731495357231765(...)027290669964066815	
			-	4933
16384	2048	512	594865747678615882(...)5136453349820334 08	4932 4932
			+594865747678615882(...)513645334982033407	
			1415461031044954789(...)668104633712377855	
			-	9865
32768	4096	1024	707730515522477394(...)3340523168561889 28	9864 9864
			+707730515522477394(...)334052316856188927	
			2003529930406846464(...)587895905719156735	
			-	19729
65536	8192	2048	1001764965203423232(...)793947952859578 368	19729 19729
			+1001764965203423232(...)79394795285957 8367	
131072	16384	4096	4014132182036063039(...)812318570934173	39457

			695	39457
			2007066091018031519(...)906159285467086848	- 39457
			+2007066091018031519(...)90615928546708 6847	
			16113257174857604736(...)60534993429830 0415	
262144	32768	8192	8056628587428802368(...)302674967149150 208	- 78914 78913 78913
			+8056628587428802368(...)30267496714915 0207	
			259637056783100077(...)364528226185773055	157827
524288	65536	16384	129818528391550038(...)1822641130928865 28	- 157827
			+129818528391550038(...)182264113092886 527	157827
			67411401254990734(...)11906894033557913 5	315653
1048576	131071	32767	33705700627495367(...)559534470167789568	- 315653
			+33705700627495367(...)5595344701677895 67	315653

Операции над целыми

Арифметические операции

К целочисленным значениям применимы в первую очередь арифметические операции. Ниже приведены самые часто используемые (в скобках указаны их обозначения в различных языках программирования и им аналогичным средствах).

- **Сравнение** (англ. *comparison*). Здесь применимы соотношения «равно» («=»; «==»; «eq»), «не равно» («!=»; «<>»; «ne»),

«больше» («>»; «gt»), «больше или равно» («>=»; «ge»), «меньше» («<»; «lt») и «меньше или равно» («<=»; «le»).

- **Инкремент** (англ. ***inc**rement*; «++») и **декремент** (англ. ***dec**rement*; «--») — арифметическое увеличение или уменьшение числа на единицу. Выделено в отдельные операции из-за частого использования с переменными-счётчиками в программировании.
- **Сложение** (англ. ***add**ition*; «+») и **вычитание** (англ. ***sub**traction*; «-»).
- **Умножение** (англ. ***mul**tiplication*; «*»).
- **Деление** (англ. ***div**ision*; «/»; «\») и получение **остатка от деления** (англ. ***mod**ulo*; «%»). Некоторые процессоры (например, архитектуры x86) позволяют производить обе эти операции за одну инструкцию.
- **Инверсия знака** (англ. ***neg**ation*) и **получение абсолютного значения** (англ. ***abs**olute*).
- **Получение знака**. Результатом такой операции обычно является 1 для положительных значений, −1 — для отрицательных и 0 — для нуля.
- **Возведение в степень** («^»).

В некоторых языках программирования для лаконичности есть операторы, которые позволяют производить арифметическую операцию с присвоением. Например, «+=» складывает текущее значение переменной слева с выражением справа и помещает результат в исходную переменную. Так же в некоторых языках и средах доступна совмещённая операция **MulDiv**, которая умножает на одно число, а потом делит результат на второе.

Обычно самыми дорогими по скорости операциями являются умножение и деление (получение остатка от деления).

В памяти компьютера для хранения целых чисел обычно отводится ячейки фиксированного объёма. Из-за этого операции увеличения и уменьшения значений могут приводить к переполнению, что оборачивается искажением результата. Некоторые языки программирования позволяют производит вызов исключения в таких случаях. Кроме этого можно определять поведение при переполнении:

- Циклическая операция (обычно происходит по умолчанию). Например, если сделать инкремент 8-битного беззнакового значения 255, то получится 0.
- Операция с насыщением. Если будет достигнут предел, то конечным значением будет это предельное. Например, если к 8-битному беззнаковому числу 250 прибавить 10, то получится 255. Сложение, вычитание и умножение с насыщением обычно применяется при работе с цветом.

Побитовые операции

Помимо математических, к целым числам применимы битовые операции, которые основаны на особенностях позиционного двоичного кодирования. Обычно они выполняются значительно быстрее арифметических операций и поэтому их используют как более оптимальные аналоги.

- **Битовый сдвиг влево** с дополнением нулями аналогичен умножению числа на степень двойки (количество бит сдвига соответствует степени двойки).
- **Битовый сдвиг вправо** аналогичен делению на степень двойки (количество бит сдвига соответствует степени двойки). Некоторые языки программирования и процессоры поддерживают арифметический сдвиг, который позволяет сохранять знак у целых со знаком (сохраняется значение старшего бита).
- У целых со знаком знак можно узнать по **старшему биту** (у отрицательных он установлен).
- Чтение и установка **младшего бита** позволяет управлять чётностью (у нечётных чисел он установлен).
- **Побитовое «И»** над определённым количеством младших бит позволяет узнать остаток от деления на степень двойки (степень соответствует количеству бит).
- **Побитовое «ИЛИ»** над определённым количеством младших бит и последующий инкремент округляет число на значение, равное степени двойки (степень соответствует количеству бит) — используется для выравнивания адресов и размеров на определённое значение.

Работа со строками

Довольно частыми операциями являются получение строки из числового значения во внутреннем представлении и обратно — число из строки. При преобразовании в строку обычно доступны средства задания форматирования в зависимости от языка пользователя.

Ниже перечислены некоторые из представлений чисел строкой.

- **Десятичное число** (англ. *decimal*). При получении строки обычно можно задать разделители разрядов, количество знаков (добавляются лидирующие нули если их меньше) и обязательное указание знака числа.
- Число в системе счисления, которое является степенью двойки. Самые частые: **двоичное** (binary англ. *binary*), **восьмеричное** (англ. *octal*) и **шестнадцатеричное** (англ. *hexadecimal*). При получении строки обычно можно задать разделители групп цифр и минимальное количество цифр (производится дополнение нулями, если их меньше). Так как эти представления чаще всего используются в программировании, то здесь обычно доступны соответствующие опции. Например, указание префикса и постфикса для получения значения в соответствии с синтаксисом языка. Для 16-ричных актуально указание регистра символов, а также обязательное добавление нуля, если первая цифра представлена буквой (чтобы число не определялось как строковый идентификатор).
- **Римское число** (англ. *roman*).
- **Словесное представление** (в том числе **сумма прописью**) — число представляется словами на указанном натуральном языке.

Перечислимый тип

К целым относится также перечислимый тип. Переменные перечислимого типа принимают конечный заранее заданный набор значений. Размер набора не определяется числом байтов, используемых для представления целочисленных значений переменных такого типа.

Например, в языке Python логический тип является подтипом целого и использует имена False и True, которые при приведении к целому получают значения 0 и 1 соответственно.

2.3.2. Число с фиксированной запятой

Число с фиксированной запятой (англ. *fixed-point number*) — формат представления вещественного числа в памяти ЭВМ в виде целого числа. При этом само число x и его целочисленное представление x' связаны формулой

$$x = x' \cdot z,$$

где z — цена (вес) младшего разряда.

В случае, если $z<1$, для удобства расчётов делают, чтобы целые числа кодировались без погрешности. Другими словами, выбирают целое число u (*машинную единицу*) и принимают $z=1/u$. В случае, если $z>1$, его делают целым.

Если не требуется, чтобы какие-либо конкретные дробные числа входили в разрядную сетку, программисты обычно выбирают $z=2^{-f}$ — это позволяет использовать в операциях умножения и деления битовые сдвиги. Про такую арифметику говорят: «f битов на дробную часть, $i=n-f$ — на целую» и обозначают как «i,f», «$i.f$» или «$Qi.f$». Например: арифметика 8,24 отводит на целую часть 8 бит и 24 — на дробную. Соответственно, она способна хранить числа от -128 до $128-z$ с ценой (весом) младшего разряда $z=2^{-24}=5{,}96\cdot10^{-8}$.

Для угловых величин зачастую делают $z=2\pi\cdot2^{-f}$ (особенно если тригонометрические функции вычисляются по таблице).

Название

Название «фиксированная запятая» (или «фиксированная точка»; далее — ФЗ) произошло из-за простой метафоры: между двумя заранее определёнными разрядами ставится запятая для превращения целого числа в дробное. Например, целое число 1234 после вставки запятой превращается в дробное 12,34.

В Великобритании, США и других странах вместо запятой для отделения целой части числа от дробной используется точка, поэтому понятия «фиксированная точка» и «фиксированная запятая» эквивалентны.

Применение

- Чтобы обеспечить минимальную поддержку дробных чисел на целочисленном процессоре: микроконтроллера, мобильного телефона, приставок вплоть до Playstation и т. д. Если не решаются некорректные задачи и СЛАУ высокого порядка, фиксированной запятой зачастую достаточно — важно только подобрать подходящую цену (вес) младшего разряда для каждой из величин.
- Для ускорения вычислений в местах, где не требуется высокая точность. В большинстве современных процессоров ФЗ аппаратно не реализована, но даже программная ФЗ очень быстра — поэтому она применяется в разного рода игровых движках, растеризаторах[1] и т. д. Например, движок Doom для измерения расстояний использует фиксированную запятую 16,16, для измерения углов — 360°=65536.
- Для записи чисел, которые по своей природе имеют постоянную абсолютную погрешность: координаты в программах вёрстки, отметки времени, денежные суммы. Например, и сдачу в супермаркете, и налоги в стране вычисляют с точностью до копейки. А файлы метрики шрифтов TeX используют 32-битный знаковый тип с фиксированной запятой (12,20). На подобные величины можно отдать и плавающую запятую с достаточным количеством знаков мантиссы — но тогда поле порядка становится излишним.
- Кроме того, фиксированная запятая ведёт себя абсолютно предсказуемо — при подсчёте денег это позволяет наладить разные виды округления, а в играх — наиболее простой способ реализовать многопользовательский режим и запись повторов.

Недостаток фиксированной запятой — очень узкий диапазон чисел, с угрозой переполнения на одном конце диапазона и потерей точности вычислений на другом. Эта проблема и привела к изобретению плавающей запятой. Например: если нужна точность в 3 значащих цифры, 4-байтовая фиксированная запятая даёт диапазон в 6 порядков

(то есть, разница приблизительно 10^6 между самым большим и самым маленьким числом), 4-байтовое число одинарной точности — в 70 порядков.

Реализации

Немногие языки программирования предоставляют встроенную поддержку чисел с фиксированной запятой, поскольку для большинства применений двоичное или десятичное представление чисел с плавающей запятой проще и достаточно точно. Числа с плавающей запятой проще из-за их большего динамического диапазона, для них не нужно предварительно задавать количество цифр после запятой. Если же потребуется арифметика с фиксированной запятой, она может быть реализована программистом даже на языках типа C и C++, которые обычно не включают в себя такой арифметики.

Числа с фиксированной запятой в формате BCD часто используются для хранения денежных величин — неточности от форматов с плавающей запятой недопустимы, а простеньким микроконтроллерам платёжных терминалов BCD предпочтительнее двоичного представления. Исторически, числа с фиксированной точкой часто использовались для десятичных типов данных, например в языках PL/I и COBOL. Язык программирования Ada 2012 включает встроенную поддержку чисел с фиксированной запятой (как двоичных, так и десятичных) и чисел с плавающей запятой. JOVIAL и Coral 66 предоставляли оба формата.

Стандарт ISO/IEC TR 18037 добавляет поддержку чисел с фиксированной запятой в язык C. Разработчики компилятора GCC уже реализовали эту поддержку.

Практически все СУБД и язык SQL поддерживают арифметику с фиксированной запятой и хранение таких данных. Например, PostgreSQL имеет специальный численный тип для точного хранения чисел до 1000 цифр.

Видео-сопроцессоры приставок PlayStation (Sony), Saturn (Sega), Game Boy Advance (Nintendo), Nintendo DS, GP2X используют арифметику с фиксированной запятой для того, чтобы увеличить пропускную способность на архитектурах без FPU.

Стандарт OpenGL ES 1.x включает поддержку чисел с фиксированной запятой, так как он создан для встраиваемых систем, у которых часто нет FPU.

Операции

- Сложение и вычитание чисел с фиксированной запятой — это обычные сложение и вычитание: $(x \pm y)' = x' \pm y'$.
- Аналогично с умножением и делением на целочисленную константу: $(cx)' = c \cdot x'$.
- Умножение и деление отличаются от целочисленных на константу.

$$(x \bullet y)' = [x' \bullet y' \bullet z] = \left[\frac{x' \bullet y'}{u}\right]$$

$$\left(\frac{x}{y}\right)' = \left[\frac{x'}{z \bullet y'}\right] = \left[\frac{x' \bullet u}{y'}\right]$$

где [ ] — операция округления до целого. В частности, если в дробной части f бит:

$$(x \bullet y)' = (x' \bullet y') shr\ f, \left(\frac{x}{y}\right)' = \frac{x'\ shl\ f}{y'}.$$

- Для других операций, помимо обычных рядов Тейлора и итерационных методов, широко применяются вычисления по таблице.

Если операнды и результат имеют разную цену (вес) младшего разряда, формулы более сложны — но иногда такое приходится делать из-за большой разницы в порядке величин.

Для перевода чисел из формата с фиксированной запятой в человекочитаемый формат и наоборот применяются обычные правила перевода дробных чисел из одной позиционной системы счисления в другую.

2.3.3. Число с плавающей запятой

Число с плавающей запятой (или **число с плавающей точкой**) — форма представления вещественных (действительных) чисел, в которой число хранится в форме мантиссы и показателя степени. При этом число с плавающей запятой имеет фиксированную относительную точность и изменяющуюся абсолютную. Используемое наиболее часто представление утверждено в стандарте IEEE 754. Реализация математических операций с числами с плавающей запятой в вычислительных системах может быть как аппаратная, так и программная.

«Плавающая запятая» и «плавающая точка»

Так как в некоторых, преимущественно англоязычных и англофицированных, странах (см. подробный список Decimal separator (англ.)) при записи чисел целая часть отделяется от дробной точкой, то в терминологии этих стран фигурирует название «плавающая точка» (floating point (англ.)). Так как в России целая часть числа от дробной традиционно отделяется запятой, то для обозначения того же понятия исторически используется термин «плавающая запятая», однако в настоящее время в русскоязычной литературе и технической документации можно встретить оба варианта.

Происхождение названия

Название «плавающая запятая» происходит от того, что запятая в позиционном представлении числа (десятичная запятая, или, для компьютеров, двоичная запятая — далее по тексту просто запятая) может быть помещена где угодно относительно цифр в строке. Это положение запятой указывается отдельно во внутреннем представлении. Таким образом, представление числа в форме с плавающей запятой может рассматриваться как компьютерная реализация экспоненциальной записи чисел.

Преимущество использования представления чисел в формате с плавающей запятой над представлением в формате с фиксированной запятой (и целыми числами) состоит в том, что можно использовать существенно больший диапазон значений при неизменной относительной точности. Например, в форме с фиксированной запятой число, занимающее 6 разрядов в целой части и 2 разряда после

запятой, может быть представлено в виде 123 456,78. В свою очередь, в формате с плавающей запятой в тех же 8 разрядах можно записать числа 1,2345678; 1 234 567,8; 0,000012345678; 12 345 678 000 000 000 и так далее, но для этого необходимо иметь дополнительное двухразрядное поле для записи показателей степени 10 от 0 до 16, при этом общее число разрядов составит 8+2=10.

Скорость выполнения компьютером операций с числами, представленными в форме с плавающей запятой, измеряется во FLOPS (от англ. *floating-point operations per second* — «[количество] операций с плавающей запятой в секунду»), и является одной из основных единиц измерения быстродействия вычислительных систем.

Структура числа

Число с плавающей запятой состоит из следующих частей:

- знак мантиссы (указывает на отрицательность или положительность числа),
- мантисса (выражает значение числа без учёта порядка),
- знак порядка,
- порядок (выражает степень основания числа, на которое умножается мантисса).

Нормальная и нормализованная формы

Нормальной формой числа с плавающей запятой называется такая форма, в которой мантисса (без учёта знака) находится на полуинтервале [0 1), то есть 0 □ a < 1.

Такая форма записи имеет недостаток: некоторые числа записываются неоднозначно (например, 0,0001 можно записать как $0{,}000001 \times 10^{2}$, $0{,}00001 \times 10^{1}$, $0{,}0001 \times 10^{0}$, $0{,}001 \times 10^{-1}$, $0{,}01 \times 10^{-2}$, и так далее), поэтому распространена (особенно в информатике) также другая форма записи — *нормализованная*, в которой мантисса десятичного числа принимает значения от 1 (включительно) до 10 (не включительно), то есть 1 □ a < 10 (аналогично мантисса двоичного числа принимает значения от 1 до 2). В такой форме любое число (кроме 0) записывается единственным образом. Недостаток заключается в том, что в таком виде невозможно представить 0,

поэтому представление чисел в информатике предусматривает специальный признак (бит) для числа 0.

Старший разряд (целая часть числа) мантиссы двоичного числа (кроме 0) в нормализованном виде равен 1 (так называемая *неявная* единица), поэтому при записи мантиссы числа в ЭВМ старший разряд можно не записывать, что и используется в стандарте IEEE 754. В позиционных системах счисления с основанием большим, чем 2 (в троичной, четверичной и др.), этого свойства нет.

Способы записи

При ограниченных возможностях оформления (например, отображение числа на семисегментном индикаторе), а также при необходимости обеспечить быстрый и удобный ввод чисел, вместо записи вида $m \cdot b^e$ (m — мантисса; b — основание, чаще всего 10; e — экспонента), записывают лишь мантиссу и показатель степени, разделяя их буквой «E» (от англ. *exponent*). Основание при этом неявно полагают равным 10. Например, число $1{,}528535047 \times 10^{-25}$ в этом случае записывается как 1.528535047E-25.

Существует несколько способов того, как строки из цифр могут представлять числа:

- Наиболее распространённый путь представления значения числа из строки с цифрами — в виде целого числа — запятая (radix point) по умолчанию находится в конце строки.
- В общем математическом представлении строка из цифр может быть сколь угодно длинной, а положение запятой обозначается путём явной записи символа запятой (или, на Западе, точки) в нужном месте.
- В системах с представлением чисел в формате с фиксированной запятой существует определённое условие относительно положения запятой. Например, в строке из 8 цифр условие может предписывать положение запятой в середине записи (между 4-й и 5-й цифрой). Таким образом, строка «00012345» обозначает число 1,2345 (нули слева всегда можно отбросить).
- В экспоненциальной записи используют стандартный (*нормализованный*) вид представления чисел. Число считается записанным в стандартном (нормализованном) виде, если оно

записано в виде aq^n, где a, называемое мантиссой, такое, что $1 \leq a < q$, n — целое, называется показатель степени и q — целое, основание системы счисления (на письме это обычно 10). То есть в мантиссе запятая помещается сразу после первой значащей (не равной нулю) цифры, считая слева направо, а дальнейшая запись даёт информацию о действительном значении числа. Например, период обращения (на орбите) спутника Юпитера Ио, который равен 152 853,5047 с, в стандартном виде можно записать как $1{,}528535047 \times 10^5$ с. Побочным эффектом ограничения на значения мантиссы является то, что в такой записи невозможно изобразить число 0.

- Запись в форме с плавающей запятой похожа на запись чисел в стандартном виде, но мантисса и экспонента записываются раздельно. Мантисса записывается в *нормализованном* формате — с фиксированной запятой, подразумеваемой после первой значащей цифры. Возвращаясь к примеру с Ио́, запись в форме с плавающей запятой будет 1528535047 с показателем 5. Это означает, что записанное число в 10^5 раз больше числа 1,528535047, то есть для получения подразумеваемого числа запятая сдвигается на 5 разрядов вправо. Однако, запись в форме с плавающей запятой используется в основном в электронном представлении чисел, при котором используется основание системы счисления 2, а не 10. Кроме того, в двоичной записи мантисса обычно денормализована, то есть запятая подразумевается до первой цифры, а не после, и целой части вообще не имеется в виду — так появляется возможность и значение 0 сохранить естественным образом. Таким образом, десятичная 9 в двоичном представлении с плавающей запятой будет записана как мантисса +1001000…0 и показатель +0…0100. Отсюда, например, беды с двоичным представлением чисел типа одной десятой (0,1), для которой двоичное представление мантиссы оказывается периодической двоичной дробью — по аналогии с 1/3, которую нельзя конечным количеством цифр записать в десятичной системе счисления.

Запись числа в форме с плавающей запятой позволяет производить вычисления над широким диапазоном величин, сочетая фиксированное количество разрядов и точность. Например, в десятичной системе представления чисел с плавающей запятой (3 разряда) операцию умножения, которую мы бы записали как

$$0{,}12 \times 0{,}12 = 0{,}0144$$

в нормальной форме представляется в виде

$$(1{,}20 \cdot 10^{-1}) \times (1{,}20 \cdot 10^{-1}) = (1{,}44 \cdot 10^{-2}).$$

В формате с фиксированной запятой мы бы получили вынужденное округление

$$0{,}120 \times 0{,}120 = 0{,}014.$$

Мы потеряли крайний правый разряд числа, так как данный формат не позволяет запятой «плавать» по записи числа.

Диапазон чисел, представимых в формате с плавающей запятой

Диапазон чисел, которые можно записать данным способом, зависит от количества бит, отведённых для представления мантиссы и показателя. На обычной 32-битной вычислительной машине, использующей двойную точность (64 бита), мантисса составляет 1 бит знак + 52 бита, показатель — 1 бит знак + 10 бит. Таким образом получаем диапазон точности примерно от $4{,}94 \cdot 10^{-324}$ до $1.79 \cdot 10^{308}$ (от $2^{-52} \times 2^{-1022}$ до $\sim 1 \times 2^{1024}$). В стандарте IEEE 754 несколько значений данного типа зарезервировано для обеспечения возможности представления специальных значений. К ним относятся значения NaN (Not a Number, «не число») и +/-INF (Infinity, бесконечность), получающихся в результате операций деления на ноль или при превышении числового диапазона. Также сюда попадают денормализованные числа, у которых мантисса меньше единицы. В специализированных устройствах (например, GPU) поддержка специальных чисел часто отсутствует. Существуют программные пакеты, в которых объём памяти выделенный под мантиссу и показатель задаётся программно, и ограничивается лишь объёмом доступной памяти ЭВМ (см. *Арифметика произвольной точности*).

Точность	Одинарная	Двойная	Расширенная
Размер (байты)	4	8	10
Число десятичных знаков	~7.2	~15.9	~19.2
Наименьшее значение (>0), denorm	$1{,}4\cdot10^{-45}$	$5{,}0\cdot10^{-324}$	$1{,}9\cdot10^{-4951}$
Наименьшее значение (>0), normal	$1{,}2\cdot10^{-38}$	$2{,}3\cdot10^{-308}$	$3{,}4\cdot10^{-4932}$
Наибольшее значение	$3{,}4\times10^{+38}$	$1{,}7\times10^{+308}$	$1{,}1\times10^{+4932}$
Поля	S-E-F	S-E-F	S-E-I-F
Размеры полей	1-8-23	1-11-52	1-15-1-63

- S — знак, E — показатель степени, I — целая часть, F — дробная часть
- Так же, как и для целых, знаковый бит — старший.

Машинный эпсилон

В отличие от чисел с фиксированной запятой, сетка чисел, которые способна отобразить арифметика с плавающей запятой, неравномерна: она более густая для чисел с малыми порядками и более редкая — для чисел с большими порядками. Но относительная погрешность записи чисел одинакова и для малых чисел, и для больших. ***Машинным эпсилоном*** называется наименьшее положительное число ε такое, что $1 \oplus \varepsilon \neq 1$ (знаком $\oplus$ обозначено машинное сложение). Грубо говоря, числа a и b, соотносящиеся так, что $1<a/b<+\varepsilon$, машина не различает.

2.3.4. Комплексный тип данных

Некоторые языки программирования предоставляют специальный **тип данных для комплексных чисел**. Наличие встроенного типа упрощает хранение комплексных величин и выполение операций над ними.

Арифметика над комплексными

Комплексные переменные и значения обычно хранятся как пара чисел с плавающей запятой. Языки, поддерживающие встроенный тип для комплексных величин, обычно предоставляют специальный синтаксис

для инициализации комплексных переменных (например, `CMPLX(R, I)` в фортране) и расширяют действие основных арифметических операций ('+', '−', '×', '/'). Эти операции обычно транслируются компилятором в последовательность инструкций по обработке чисел с плавающей запятой или в вызовы функций специальной библиотеки. Иногда также предоставляются функции вывода комплексных, сравнения их на равенство и другие. Как и в математике, языки с поддержкой комплексных типов могут использовать обычные числа с плавающей запятой как комплексные с нулевой мнимой частью.

Поддержка в языках

- FORTRAN, тип `COMPLEX` поддерживается начиная с FORTRAN IV. В FORTRAN II существовала поддержка с иным синтаксисом и возможностями.
- Язык Си, начиная со стандарта C99 включительно. Комплексный тип обозначается ключевым словом `_Complex`. Реализовано множество математических функций над комплексными числами. Требуется использование заголовочного файла `<complex.h>`
- Язык C++ включает поддержку шаблонного класса `complex` и математических функций (заголовочный файл `<complex>`)
- Perl предоставляет модуль `Math::Complex`, включенный во все поставки
- Python поддерживает встроенный тип `complex`. Мнимые константы обозначаются добавлением суффикса «j». Комплексные математические функции реализованы в стандартном библиотечном модуле `cmath`
- Ruby поддерживает класс `Complex` (стандартный библиотечный модуль `complex`)
- OCaml поддерживает комплексные в стандартном библиотечном модуле `Complex`
- Haskell — стандартная библиотека `Complex`
- Apache Commons Math предоставляет поддержку для ЯП Java, класс `Complex`
- Common Lisp: Стандарт ANSI Common Lisp описывает работу с комплексными числами над типами float и над типами с произвольной точностью. Базовые математические функции определены также и для комплексных

- .NET Framework поддерживает System.Numerics.Complex с версии 4.0.

Тип данных `COMPLEX` широко используется с версии FORTRAN IV.

2.3.5. Интервальная арифметика

Интервальная арифметика — математическая структура, которая для вещественных интервалов определяет операции, аналогичные обычным арифметическим. Эту область математики называют также **интервальным анализом** или **интервальными вычислениями**. Данная математическая модель удобна для исследования различных прикладных объектов:

- Величины, значения которых известны только приближённо, то есть определён конечный интервал, в котором эти значения содержатся.
- Величины, значения которых в ходе вычислений искажены ошибками округления.
- Случайные величины.

Объекты и операции интервальной арифметики можно рассматривать как обобщение модели вещественных чисел, поэтому интервалы в ряде источников называются **интервальными числами**. Практическая важность этой модели связана с тем, что результаты измерений и вычислений почти всегда имеют некоторую погрешность, которую необходимо учесть и оценить.

Операции над интервалами

Мы будем рассматривать всевозможные конечные вещественные интервалы $[a,b]$ ($a \le b$). Операции над ними определяются следующим образом:

- Сложение: $[a,b] + [c,d] = [a + c, b + d]$
- Вычитание: $[a,b] - [c,d] = [a - d, b - c]$
- Умножение: $[a,b] \times [c,d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$
- Деление: $[a,b] / [c,d] = [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$

Из определения видно, что интервал-сумма содержит всевозможные суммы чисел из интервалов-слагаемых и определяет границы множества таких сумм. Аналогично трактуются прочие действия. Отметим, что операция деления определена только в том случае, когда интервал-делитель не содержит нуля.

Вырожденные интервалы, у которых начало и конец совпадают, можно отождествить с обычными вещественными числами. Для них данные выше определения совпадают с классическими арифметическими действиями.

Свойства операций

Сложение и умножение интервалов коммутативны и ассоциативны. Дистрибутивное свойство имеет место в ослабленном виде:

$$X(Y+Z) \subset XY+XZ$$

2.4. Текстовые данные

Те́кстовые да́нные (также **те́кстовый форма́т**) — представление информации строкового типа (то есть, последовательности печатных символов) в вычислительной системе. В MIME закодированным таким образом данным соответствует тип `text/plain`.

Часто текстовые данные понимаются в более узком смысле — как текст на каких-либо языках (формальных или естественных), который может быть прочитан и понят человеком.

Текстовому формату противопоставляются «двоичные данные», информация в которых закодирована произвольным образом, не рассчитанном на восприятие человеком.

Для большей части компьютерного оборудования и программ неважно, являются ли данные текстовыми. Однако многие сетевые протоколы рассчитаны на работу только с текстовыми данными и не могут обрабатывать произвольную последовательность байтов. Также,

некоторые программы обрабатывают текстовые и двоичные данные по-разному, а некоторые предназначены для обработки именно текстовых данных. Программы для создания и редактирования текстовых данных называются текстовыми редакторами.

Структура

Текстовыми данными как правило называются последовательности из подмножества знаков, включающего только печатные знаки (буквы, цифры, знаки препинания) и некоторые управляющие знаки (пробелы, табуляции, переводы строки). Существуют методы (например, UUENCODE или Base64), позволяющие закодировать в текстовом формате произвольные данные любого формата, что часто используется для кодирования бинарных данных.

Требование к возможности понимания содержимого человеком вносит дополнительную избыточность в представление данных. К примеру, число 123, для кодирования которого достаточно одного 8-битного байта, в текстовом виде кодируется несколькими цифровыми символами — так, в десятичной системе счисления для этого требуется три знака («123»), в двоичной — семь знаков («1111011»), в шестнадцатеричной — два («7B»).

Текстовый формат не позволяет использовать команды форматирования текста, управлять атрибутами шрифтов, размечать содержимое.

2.4.1. Символьный тип

Символьный тип (Char) — тип данных, предназначенный для хранения одного символа (управляющего или печатного) в определённой кодировке. Может являться как однобайтовым (для стандартной таблицы символов), так и многобайтовым (к примеру, для Юникода). Основным применением является обращение к отдельным знакам строки.

Язык C

В языке Си размер типа равен одному байту. В общем случае размер типа `char` на конкретной платформе регулируется значением

константы `CHAR_BIT`, определённой в заголовочном файле limits.h. По умолчанию и на платформах x86 она равна 8. `Char` может вмещать максимум один символ ASCII.

Если `char` определён как `signed` (знаковый), то его диапазон значений составляет от −128 до 127 (может быть на единицу дальше в положительную сторону, в зависимости от реализации). Если он определён как `unsigned` (беззнаковый), то его значения могут составлять от 0 до 255. Значение, содержащееся в этом типе, можно всегда безопасно привести к значению типа int. В Си нет примитивных типов для работы со строками, поэтому для работы с ними используется указатель `char *`.

Управляющие символы

Управля́ющие си́мволы — символы в кодировке, которым не приписано графическое представление, но которые используются для управления устройствами, организации передачи данных и других целей.

Сейчас для этих целей применяются форматы файлов, языки управления устройствами (такие как Postscript) и сетевые протоколы.

Стандарт POSIX требует обязательного наличия лишь восьми управляющих символов — \0, \a, \b, \t, \n, \v, \f, \r (см. переносимый набор символов).

Управляющие символы ASCII

Номер	Английское название	Русское название	Сочетание клавиш[1]	Escape последовательность	Назначение
00	NULL	пустой символ	^@	\0	Этот символ ничего не делает. Некоторые терминалы изображают его как пробел, но это неправильно. Часто NULL используют для обозначения конца цепочки символов (например, в языке C).
01	START OF HEADING	начало заголовка	^A		В настоящее время используется в консоли маршрутизаторов Cisco.
02	START OF TEXT	начало текста	^B		В настоящее время используется в консоли маршрутизаторов Cisco.
03	END OF TEXT	конец текста	^C		При вводе на терминале обычно интерпретируется как сигнал прерывания.
04	END OF TRANSMISSION	конец передачи	^D		При вводе на терминале в UNIX-системах интерпретируется

					как конец вводимых данных. Если текущая программа брала данные с терминала, то она завершается, как только обработает всё, что было до символа ^D.
05	ENQUIRY	запрос	^E		Использовался в телетайпной связи. В ответ предполагалось получить идентификационную строку удалённого аппарата.
06	ACKNOWLEDGE	подтверждение	^F		Использовался в телетайпной связи.
07	BELL	звуковой сигнал	^G	\a	Если этот символ послать на принтер или на терминал, то ничего не напечатается, но послышится звуковой сигнал.
08	BACKSPACE	возврат на шаг	^H	\b	Перемещает позицию печати на один символ назад. На принтерах может использоваться для наложения одного символа на другой, например **a** BS ^ = **â**. При вводе с терминала иногда используется для стирания

					предшествующего символа («забой»).
09	CHARACTER TABULATION (horizontal tabulation)	горизонтальная табуляция	^I	\t	Перемещает позицию печати к следующей позиции горизонтальной табуляции.
0A	LINE FEED	перевод строки	^J	\n	Перемещает позицию печати на одну строку вниз (исходно — без *возврата каретки*). Разделяет строки текстовых файлов в Unix-системах.
0B	LINE TABULATION (vertical tabulation)	вертикальная табуляция	^K	\v	Перемещает позицию печати к следующей позиции вертикальной табуляции. На терминалах этот символ обычно эквивалентен переводу строки.
0C	FORM FEED	прогон страницы, смена страницы	^L	\f	Выбрасывает текущую страницу и начинает печать со следующей. На терминалах этот символ обычно эквивалентен переводу строки (хотя в принципе можно было бы его использовать для очистки экрана).
0D	CARRIAGE RETURN	Возврат каретки	^M	\r	Перемещает позицию печати в

				крайнее левое положение (исходно — без *перевода на следующую строку*'). Разделяет строки текстовых файлов в некоторых ОС (например Mac OS, но не в Mac OS X). Во многих других ОС (CP/M, MS-DOS и Microsoft Windows), для разделения строк используется сочетание кодов возврата каретки (CARRIAGE RETURN) и перевода строки (LINE FEED) — $0D_{16}$ + $0A_{16}$, то есть в том виде в котором файл можно отправить непосредственно на принтер.
0E	SHIFT OUT (locking-shift one)	режим национальных символов	^N	В КОИ-7 включает режим национальных символов. На некоторых принтерах включает режим символов двойной ширины.
0F	SHIFT IN (locking-shift zero)	режим обычного ASCII	^O	В КОИ-7 включает латинский режим. На некоторых

				принтерах включает режим узких символов.
10	DATA LINK ESCAPE	освобождение канала данных	^P	Означает, что следующий за ним управляющий символ должен восприниматься как данные, а не как управляющий символ.
11	DEVICE CONTROL ONE	1-й код управления устройством	^Q	На терминалах разрешает продолжить вывод данных.
12	DEVICE CONTROL TWO	2-й код управления устройством	^R	
13	DEVICE CONTROL THREE	3-й код управления устройством	^S	На терминалах временно прерывает (приостанавливает) вывод данных.
14	DEVICE CONTROL FOUR	4-й код управления устройством	^T	
15	NEGATIVE ACKNOWLEDGE	отрицательное подтверждение	^U	Использовался в телетайпной связи.
16	SYNCHRONOUS IDLE	пустой символ для синхронного режима передачи	^V	Некоторые линии связи устроены так, что требуют непрерывной передачи данных. Если передавать нечего, то передают этот символ.
17	END TRANSMISSION BLOCK	конец блока передаваемых данных	^W	

18	CANCEL	отмена	^X		Данные, которые идут перед ним, некорректны. (Обычно речь идёт об одной строке.)
19	END OF MEDIUM	конец носителя	^Y		Использовался, напр., если закончилась перфолента и т. п.
1A	SUBSTITUTE	символ замены	^Z		Ставится на месте символов, значения которых были потеряны при передаче. В CP/M и MS-DOS использовался для обозначения конца текстовых файлов и конца вводимых с консоли данных (хотя для этого были предназначены символы ^C и ^D). Некоторые текстовые редакторы под DOS автоматически добавляли в конце файла ^Z.
1B	ESCAPE	Альтернативный регистр № 2 (AP2)	^[	\e	Означает, что следующие за ним символы имеют какое-то другое значение, отличное от того, которое определено в ASCII. Обычно начинает управляющие последовательност

				и. См. также ANSI.SYS.
1C	INFORMATION SEPARATOR FOUR (file separator)	разделитель данных № 4 (разделитель файлов)	^\	
1D	INFORMATION SEPARATOR THREE (group separator)	разделитель данных № 3 (разделитель групп)	^]	
1E	INFORMATION SEPARATOR TWO (record separator)	разделитель данных № 2 (разделитель записей)	^^	Видимо, предназначался для разделения записей в базах данных, но практически никогда не используется для этого.
1F	INFORMATION SEPARATOR ONE (unit separator)	разделитель данных № 1 (разделитель полей)	^_	Видимо, предназначался для разделения полей в базах данных, но практически никогда не используется для этого.
7F	DELETE	удаление	^?	Предназначен для забивания ошибочно пробитых символов на семидорожечных перфолентах (поскольку обозначается пробитием дырочек во всех дорожках), поэтому там он эквивалентен пустому символу

(\0). На терминалах может генерироваться нажатием либо кнопки Backspace, либо кнопки Delete.

Управляющие символы ISO 8859

80, PADDING CHARACTER, символ-заполнитель.

81, HIGH OCTET PRESET, ???.

82, BREAK PERMITTED HERE, здесь разрешён разрыв строки.

83, NO BREAK HERE, здесь не разрешён разрыв строки.

84, INDEX, ???.

85, NEXT LINE, следующая строка. Одновременно переводит строку и возвращает позицию печати к началу строки (эквивалентно \r\n).

86, START OF SELECTED AREA, начало выделенной области.

87, END OF SELECTED AREA, конец выделенной области.

88, CHARACTER TABULATION SET, установка позиций горизонтальной табуляции.

89, CHARACTER TABULATION WITH JUSTIFICATION, установка позиций и выравнивания горизонтальной табуляции.

8A, LINE TABULATION SET, установка позиций вертикальной табуляции.

8B, PARTIAL LINE FORWARD, частичный перевод строки вперёд.

8C, PARTIAL LINE BACKWARD, частичный перевод строки назад.

8D, REVERSE LINE FEED, обратный перевод строки.

8E, SINGLE SHIFT TWO, 2-е значение для следующего символа.

8F, SINGLE SHIFT THREE, 3-е значение для следующего символа.

90, DEVICE CONTROL STRING, строка управления устройством.

91, PRIVATE USE ONE, пользовательский символ № 1.

92, PRIVATE USE TWO, пользовательский символ № 2.

93, SET TRANSMIT STATE, установка режима передачи.

94, CANCEL CHARACTER, символ отмены.

95, MESSAGE WAITING, есть сообщение.

96, START OF GUARDED AREA, начало защищённой области.

97, END OF GUARDED AREA, конец защищённой области.

98, START OF STRING, начало строки.

99, SINGLE GRAPHIC CHARACTER INTRODUCER, следующий символ интерпретируется как специальный графический.

9A, SINGLE CHARACTER INTRODUCER, следующий символ интерпретируется как управляющий.

9B, CONTROL SEQUENCE INTRODUCER, начало управляющей последовательности. Обычно этот символ эквивалентен Escape+[.

9C, STRING TERMINATOR, окончание строки.

9D, OPERATING SYSTEM COMMAND, команда операционной системы.

9E, PRIVACY MESSAGE, секретное сообщение.

9F, APPLICATION PROGRAM COMMAND, команда прикладной программы.

Управляющие символы Unicode

034F, COMBINING GRAPHEME JOINER. Объединить символы, стоящие слева и справа (создать лигатуру).

200B, ZERO-WIDTH SPACE, пробел нулевой ширины. При выравнивании по ширине может расширяться.

200C, ZERO WIDTH NON-JOINER. Запрещает образование лигатур.

200D, ZERO WIDTH JOINER. Разрешает образование лигатур.

200E, LEFT-TO-RIGHT MARK. Писать слева направо.

200F, RIGHT-TO-LEFT MARK. Писать справа налево.

2028, LINE SEPARATOR, разделитель строк. Разделяет строки текста, но не абзацы.

2029, PARAGRAPH SEPARATOR, разделитель абзацев. Разделяет абзацы текста.

202A, LEFT-TO-RIGHT EMBEDDING. Начало текста, написанного слева направо, внутри текста, написанного справа налево.

202B, RIGHT-TO-LEFT EMBEDDING. Начало текста, написанного справа налево, внутри текста, написанного слева направо.

202C, POP DIRECTIONAL FORMATTING. Конец вставленного текста с другим направлением.

202D, LEFT-TO-RIGHT OVERRIDE. Заменить текст, написанный слева направа, текстом, написанным справа налево.

202E, RIGHT-TO-LEFT OVERRIDE. Заменить текст, написанный справа налево, текстом, написанным слева направо.

2060, WORD JOINER, соединитель слов.

FE01 … **FE0F**, VARIATION SELECTOR-1…16, выбор варианта начертания № 1 … № 16.

FEFF, ZERO WIDTH NO-BREAK SPACE / BYTE ORDER MARK, неразрывный пробел нулевой ширины / индикатор порядка байтов. Этот символ используется для указания того, что данный файл записан в UTF-16 или UTF-32 с определённым порядком байтов (поскольку символа FFFE нет, а в UTF-8 байты FE и FF не используются). Использование этого символа в качестве неразрывного пробела нулевой ширины не рекомендуется; для этого есть символ U+2060 (word joiner).

FFFD, REPLACEMENT CHARACTER, заменяющий символ. Используется, когда значение символа неизвестно или не может быть выражено в Юникоде (см. также символ 1A).

E0100 … **E01EF**, VARIATION SELECTOR-17…256, выбор варианта начертания № 17 … № 256.

Кодировки символов

Основы		алфавит • текст (файл • данные) • набор символов • конверсия
Исторические кодировки	Докомп.:	семафорная (Макарова) • Морзе • Бодо • МТК-2
	Комп.:	6-битная • УПП • RADIX-50 • EBCDIC (ДКОИ-8) • КОИ-7 • ISO 646
современное 8-битное представление	символы	ASCII (управляющие • печатные) • не-ASCII (псевдографика)
	8-битные код.стр.	Кириллица: КОИ-8 • Основная кодировка • MacCyrillic

	ISO 8859	1 (лат.) • 2 • 3 • 4 • 5 (кир.) • 6 • 7 • 8 • 9 • 10 • 11 • 12 • 13 • 14 • 15 (€) • 16
	Windows	1250 • 1251 (кир.) • 1252 • 1253 • 1254 • 1255 • 1256 • 1257 • 1258 • WGL4
	IBM & DOS	437 • 850 • 852 • 855 • 866 «альт.» • МИК
	Традиционные	DBCS (GB2312) • HTML
Многобайтные	Unicode	UTF-32 • UTF-16 • UTF-8 • список символов (кириллица)
Связанные темы		интерфейс пользователя • раскладка клавиатуры • локаль • перевод строки • шрифт • транслит • нестандартные шрифты
Утилиты		iconv • recode

2.4.2. Строковый тип данных

В программировании, **строковый тип** (англ. *string* «нить, вереница») — тип данных, значениями которого является произвольная последовательность (строка) символов алфавита. Каждая переменная такого типа (**строковая переменная**) может быть представлена фиксированным количеством байтов либо иметь произвольную длину.

2.4.2.1. Представление в памяти

Некоторые языки программирования накладывают ограничения на максимальную длину строки, но в большинстве языков подобные ограничения отсутствуют. При использовании Unicode каждый символ строкового типа может требовать двух или даже четырёх байтов для своего представления.

Основные проблемы в машинном представлении строкового типа:

- строки могут иметь достаточно существенный размер (до нескольких десятков мегабайтов);
- изменяющийся со временем размер — возникают трудности с добавлением и удалением символов.

В представлении строк в памяти компьютера существует два принципиально разных подхода.

Представление массивом символов

В этом подходе строки представляются *массивом символов*; при этом размер массива хранится в отдельной (служебной) области. От названия языка Pascal, где этот метод был впервые реализован, данный метод получил название *Pascal strings*.

Слегка оптимизированным вариантом этого метода является т. н. формат *c-addr u* (от англ. *character-aligned address + unsigned number*), применяемый в Форте. В отличие от Pascal strings, здесь размер массива хранится не совместно со строковыми данными, а является частью указателя на строку.

Преимущества

- программа в каждый момент времени содержит сведения о размере строки, поэтому операции добавления символов в конец, копирования строки и собственно получения размера строки выполняются достаточно быстро;
- строка может содержать любые данные;
- возможно на программном уровне следить за выходом за границы строки при её обработке;
- возможно быстрое выполнение операции вида «взятие N-ого символа с конца строки».

Недостатки

- проблемы с хранением и обработкой символов произвольной длины;
- увеличение затрат на хранение строк — значение «длина строки» также занимает место и в случае большого количества

строк маленького размера может существенно увеличить требования алгоритма к оперативной памяти;
- ограничение максимального размера строки. В современных языках программирования это ограничение скорее теоретическое, так как обычно размер строки хранится в 32-битовом поле, что даёт максимальный размер строки в 4 294 967 295 байт (4 гигабайта);
- при использовании алфавита с переменным размером символа (например, UTF-8), в размере хранится не количество символов, а именно размер строки в байтах, поэтому количество символов необходимо считать отдельно.

Метод «завершающего байта»

Второй метод заключается в использовании *«завершающего байта»*. Одно из возможных значений символов алфавита (как правило, это символ с кодом 0) выбирается в качестве признака конца строки, и строка хранится как последовательность байтов от начала до конца. Есть системы, в которых в качестве признака конца строки используется не символ 0, а байт 0xFF (255) или код символа «$».

Метод имеет три названия — *ASCIIZ* (или asciz, символы в кодировке ASCII с нулевым завершающим байтом), *C-strings* (наибольшее распространение метод получил именно в языке Си) и метод нуль-терминированных строк.

Нуль-терминированная строка

Нуль-терминированная строка или **C-строка** (от названия языка Си) или ASCIIZ-строка — способ представления строк в языках программирования, при котором вместо введения специального строкового типа используется массив символов, а концом строки считается первый встретившийся специальный *нуль-символ* (NUL из кода ASCII, со значением 0).

Например, в *строковом буфере* (области памяти, выделенной для хранения строки) размером 11 байт нуль-терминированная строка «СТРОКА» в кодировке Windows-1251 может представляться следующим образом:

C	T	P	O	K	A	**NUL**	F	%	NUL	4
0xD1	0xD2	0xD0	0xCE	0xCA	0xC0	**0x00**	0x46	0x25	0x00	0x34

В данном примере представлена область памяти из 11 байт, хотя на самом деле строка занимает всего 7. Символы после нуль-символа (8 - 11 байты) называются *мусором* — это данные, которые могли остаться в буфере от предыдущих строк или от других использований памяти. Среди них также могут находиться нулевые символы.

При использовании однобайтных кодировок (ASCII) объём памяти, требуемый для представления строки из N символов, равен N + 1 байт. В том случае, когда для кодирования символов применяется Юникод, длина строки зависит от используемого представления Юникода (например, 2N + 2 байта для UCS-2).

Такие строки являются стандартом в Си и некоторых других языках программирования. Поскольку они используются для передачи строковых аргументов в стандартные функции во многих операционных системах, операции для работы с нуль-терминированными строками появились в Паскале и других языках.

Для ссылки на нуль-терминированную строку применяется указатель на первый её символ. Это простой, быстрый и гибкий подход, но чреватый ошибками. Программист постоянно должен следить за своим кодом, а именно:

- быть уверенным, что не случаются переполнения буфера;
- аккуратно проводить управление памятью, выделяемой под строки;
- следить за корректной нуль-терминацией строк при использовании функций, которые её не гарантируют (например, `strncpy`);
- в редких случаях, когда размер строки может быть очень велик, следить, что не происходит переполнение целого при подсчёте длины и прочих связанных с длиной вычислениях.

Кроме того, некоторые операции со строками, например, конкатенация, для нуль-терминированных строк выполняются медленнее, чем для других типов строк.

Сравнение с альтернативами

Альтернативой нуль-терминированным строкам являются способы, принятые в Паскале и современных ООП-языках. В Паскале строка начинается с первого элемента массива, а в нулевом элементе хранится длина строки. В этом случае не требуется специального терминатора для обозначения конца строки. С другой стороны, здесь на длину строки накладывается ограничение, связанное с вместимостью нулевого элемента массива, то есть в случае с однобайтовыми элементами длина строки не может превышать 255 символов. Нуль-терминированные строки такому ограничению не подвержены и теоретически могут хранить строки любой длины. В объектно-ориентированных языках применяется хранение записи с длиной строки и ссылкой (или указателем) на массив символов. Эти способы не подвержены недостатку нуль-терминированных строк: они могут хранить в себе нуль-символы без искажений и специального кодирования.

В ряде интерфейсов применяются дважды-нуль-терминированные строки, признаком завершения которых является два последовательных нуль-терминатора.

В языке Си

Для работы с нуль-терминированными строками в языке программирования Си используется ряд функций:

- strcpy, wcscpy — копирование строк;
- strlen, wcslen — вычисление длины строки;
- strchr — поиск символа в строке;
- strdup — дублирование строк;
- strstr — поиск подстроки;
- strtok — разделение строки через разделители на подстроки;
- strbrk — поиск первого вхождения в строку одного из символов другой строки.

В языке ассемблера

В некоторых разновидностях языка ассемблера для определения NUL-терминированных строк используется специальная директива. Так в GNU Assembler-е для этого есть директива `.asciz`.

Преимущества метода «завершающего байта»

- отсутствие дополнительной служебной информации о строке (кроме завершающего байта);
- возможность представления строки без создания отдельного типа данных;
- отсутствие ограничения на максимальный размер строки;
- экономное использование памяти;
- простота получения суффикса строки;
- простота передачи строк в функции (передаётся указатель на первый символ);

Недостатки метода «завершающего байта»

- долгое выполнение операций получения длины и конкатенации строк;
- отсутствие средств контроля за выходом за пределы строки, в случае повреждения завершающего байта возможность повреждения больших областей памяти, что может привести к непредсказуемым последствиям — потере данных, краху программы и даже всей системы;
- невозможность использовать символ завершающего байта в качестве элемента строки.
- невозможность использовать некоторые кодировки с размером символа в несколько байт (например, UTF-16), т.к. во многих таких символах, например Ā (0x0100), один из байтов равен нулю (в то же время, кодировка UTF-8 свободна от этого недостатка).

Использование обоих методов

В таких языках, как, например, Оберон, строка размещается в массиве символов определённой длины, причём её конец обозначается нулевым символом. По умолчанию, весь массив заполнен нулевыми символами.

Такой способ позволяет объединить многие преимущества обоих подходов, а также избежать большинство их недостатков.

Представление в виде списка

Языки Erlang, Haskell, Пролог используют для строкового типа список символов. Этот метод делает язык более «теоретически элегантным» за счёт соблюдения ортогональности в системе типов, но приносит существенные потери быстродействия.

2.4.2.2. Реализация в языках программирования

- В первых языках программирования вообще не было строкового типа; программист должен был сам строить функции для работы со строками того или другого типа.
- В Си используются нуль-терминированные строки с полным ручным контролем со стороны программиста.
- В стандартном Паскале строка выглядит как массив из 256 байтов; первый байт хранил длину строки, в остальных хранится её тело. Таким образом, длина строки не может превышать 255 символов. В Borland Pascal 7.0 также появились строки «а-ля Си» — очевидно, из-за того, что в число поддерживаемых платформ вошла Windows.
- В Object Pascal и C++ STL строка является «чёрным ящиком», в котором выделение/высвобождение памяти происходит автоматически — без участия программиста. При создании строки память выделяется автоматически; как только на строку не останется ни одной ссылки, память возвращается системе. Преимущество этого метода в том, что программист не задумывается над работой строк. С другой стороны, программист имеет недостаточный контроль над работой программы в критичных к скорости участках; также трудно реализуется передача таких строк в качестве параметра в DLL. Также Object Pascal автоматически следит, чтобы в конце строки был символ с кодом 0. Поэтому если функция требует на входе нуль-терминированную строку, для конвертации надо просто написать `PAnsiChar(`*`строковая_переменная`*`)` или `PWideChar(`*`строковая_переменная`*`)` (для Pascal), *`переменная`*`.c_str()` (для Builder/STL).

- В C# и других языках со сборкой мусора строка является неизменяемым объектом; если строку нужно модифицировать, создаётся другой объект. Этот метод медленный и расходует немало временной памяти, но хорошо сочетается с концепцией сборки мусора. Преимущество этого метода в том, что присваивание происходит быстро и без дублирования строк. Также имеется некоторый ручной контроль над конструированием строк (в Java, например, через классы `StringBuffer и StringBuilder`) — это позволяет уменьшить количество выделений и высвобождений памяти и, соответственно, увеличить скорость.
    - В некоторых языках (например, Standard ML) кроме этого, есть дополнительный модуль для обеспечения ещё большей эффективности — «подстрока» (SubString). Его использование позволяет выполнять операции над строками без копирования их тел посредством манипулирования индексами начала и конца подстроки; физическое копирование происходит лишь при необходимости преобразовании подстрок в строки.

2.4.2.3. Операции

Простейшие операции со строками

- получение символа по номеру позиции (индексу) — в большинстве языков это тривиальная операция;
- конкатенация (соединение) строк.

Производные операции

- получение подстроки по индексам начала и конца;
- проверка вхождения одной строки в другую (поиск подстроки в строке);
- проверка на совпадение строк (с учётом или без учёта регистра символов);
- получение длины строки;
- замена подстроки в строке.

Операции при трактовке строк как списков

- свёртка;
- отображение одного списка на другой;
- фильтрация списка по критерию.

Более сложные операции

- нахождение минимальной надстроки, содержащей все указанные строки;
- поиск в двух массивах строк совпадающих последовательностей *(задача о плагиате)*.

Возможные задачи для строк на естественном языке

- сравнение на близость указанных строк по заданному критерию;
- определение языка и кодировки текста на основании вероятностей символов и слогов.

2.4.2.4. Представление символов строки

До последнего времени один символ всегда кодировался одним байтом (8 двоичных битов; применялись также кодировки с 7 битами на символ), что позволяло представлять 256 (128 при семибитной кодировке) возможных значений. Однако для полноценного представления символов алфавитов нескольких языков (многоязыковых документов, типографских символов — несколько видов кавычек, тире, нескольких видов пробелов и для написания текстов на иероглифических языках — китайском, японском и корейском) 256 символов недостаточно. Для решения этой проблемы существует несколько методов:

- Переключение языка управляющими кодами. Метод не стандартизирован и лишает текст самостоятельности (то есть последовательность символов без управляющего кода в начале теряет смысл); использовался в некоторых ранних русификациях ZX-Spectrum и БК.
- Использование двух или более байт для представления каждого символа (UTF-16, UTF-32). Главным недостатком этого метода является потеря совместимости с предыдущими библиотеками для работы с текстом при представлении строки

как ASCIIZ. Например, концом строки должен считаться уже не байт со значением 0, а два или четыре подряд идущих нулевых байта, в то время как одиночный байт «0» может встречаться в середине строки, что сбивает библиотеку «с толку».

- Использование кодировки с переменным размером символа. Например, в UTF-8 часть символов представляется одним байтом, часть двумя, тремя или четырьмя. Этот метод позволяет сохранить частичную совместимость со старыми библиотеками (нет символов 0 внутри строки и поэтому 0 можно использовать как признак конца строки), но приводит к невозможности прямой адресации символа в памяти по номеру его позиции в строке.

Пустая строка

Пустая строка *(в информатике)* — это термин, обозначающий значение строкового типа, не содержащее символов (то есть содержащее 0 символов, нулевой длины).

Несмотря на то, что *пустая строка* не содержит символьных данных, тем не менее её представление в памяти занимает определенное место (см. Строковый тип). Например, *пустое строковое* значение может содержать маркер длины или терминальный символ. В частности, в языках программирования в качестве терминального символа часто используется символ с кодом 0, а в обычном текстовом файле символы CR (ASCII 0x0D), LF (ASCII 0x0A) или их комбинация CR + LF (ASCII 0x0D0A), обозначающие конец предыдущей и начало следующей строки.

Пустую строку не следует путать со строкой, состоящей из управляющих символов или пробелов, которые, хоть и не отображаются при выводе строки на печать или на экран, тем не менее являются символьными данными.

Использование

- Во многих (особенно нетипизированных) языках программирования *пустая строка* может быть интерпретирована как логическое отрицание.

- В текущих версиях СУБД Oracle пустая строка эквивалентна NULL

- Пустая строка может быть использована:
  - для обозначения отсутствия каких-либо данных,
  - в качестве терминального значения при перечислении.

Разбиение на строки

Текстовые данные могут разделяться на строки. На некоторых платформах (в основном, в операционных системах семейства UNIX) разбиение на строки кодируется одним управляющим знаком с кодом 10 в таблице ASCII (наименование — Line Feed, LF), на других (к примеру, в MS-DOS и Microsoft Windows) — парой управляющих знаков с кодами 13 и 10 (Carriage Return и Line Feed, CR/LF). В Mac OS (но не Mac OS X) разбиение кодируется одним знаком с кодом 13.

Такое разбиение управляющим знаком или знаками продиктовано тем, как работали пишущие машинки, через которые осуществлялся ввод в некоторых первых компьютерах — позиция ввода там указывалась положением валика с бумагой, и для поворота валика и перехода к следующей строке требовалось нажатие одной или двух клавиш или рычажков.

Также, знаки разбиения строк использовались для управления механическими принтерами (в качестве которых могли выступать те же печатные машинки, используемые и для ввода) — знак LF вызывал прокрутку рулона с бумагой, а знак CR вызывал возврат печатной каретки (там, где они были) в начало строки. Отсюда и название знаков — англ. *Line Feed* (перевод строки) и англ. *Carriage Return* (возврат каретки).

На некоторых платформах разбиение на строки делалось иначе — текст представлялся в виде последовательности записей фиксированной длины, для чего более короткие строки дополнялись нужным количеством пробелов. Это соответствовало представлению данных на перфокартах, которые служили средством ввода и даже хранения данных, имевших фиксированную ширину (например, 80 позиций - колонок).

Использование

Основная цель применения текстовых данных — «общий знаменатель», независимость от отдельных программ, требующих собственного кодирования или форматирования и несовместимых с другими программами. Текстовые файлы (файлы в текстовом формате) могут быть открыты, прочитаны и отредактированы в любых текстовых редакторах, таких как MS-DOS Editor (англ.) (DOS), Блокнот (Windows), ed, vi и vim (UNIX, Linux), SimpleText (англ.), TextEdit (Mac OS X) и т. п. Другие программы также как правило умеют читать и импортировать текстовые данные. Просмотреть текстовые файлы можно также встроенными командами (`type` в DOS и Windows) и утилитами (`cat` в Unix).

Текстовый формат часто используются для представления данных, которые сами не являются чисто текстовыми. В этом случае другие форматы данных «надстраиваются» над простым текстом, для чего их управляющие конструкции выражаются посредством печатных слов и знаков препинания. Это обеспечивает удобство работы с данными на двух уровнях — например, данные HTML и XML можно просматривать и редактировать с показом форматирования в режиме WYSIWYG, а можно их открыть в обычном текстовом редакторе и иметь доступ ко всем тонкостям языка разметки. При хранении данных в «двоичном» виде (как это делается, например, в Microsoft Word ранних версий) с ними нередко нельзя работать в других программах (из-за недоступности информации о структуре формата) или даже в разных версиях одной и той же программы.

В большинстве языков программирования предполагается использование текстового формата для исходного кода программ. Помимо прочего, это позволяет применять к исходным кодам разнообразные утилиты для преобразований, оформления, поиска, статистики, анализа и т. п.

В файлах конфигурации многих программ применяется текстовый формат, даже если там представлены числа и двоичные переключатели (да/нет). Это несколько усложняет программы из-за необходимости преобразования текстовых данных во внутренний формат и обратно, но появляется возможность править конфигурацию вручную, без использования средств настройки самой программы.

Затруднительным является указание на какую-то определенную часть текста, хранящегося в формате текстовых данных. В качестве указателей могут использоваться номера строк или номера символов[2].

Близкие термины

Термин *открытый текст* (англ. *plaintext*; выглядит очень похоже на термин англ. *plain text*, используемый для обозначения текстовых данных) широко применяется в криптографии и означает любые незашифрованные данные, в том числе и нетекстовые. Термин *чистый текст* (англ. *cleartext*) также применяется в криптографии и означает незашифрованные данные, к тому же понятные человеку и незащищённые от «подслушивания» при передаче.

2.5. Сильная и слабая типизация

По одной из классификаций, языки программирования неформально делятся на **сильно и слабо типизированные** (англ. *strongly and weakly typed*), то есть обладающие сильной или слабой системой типов. Эти термины не являются однозначно трактуемыми, и чаще всего используются для указания на достоинства и недостатки конкретного языка. Существуют более конкретные понятия, которые и приводят к называнию тех или иных систем типов «*сильными*» или «*слабыми*».

Примечание: В русскоязычной литературе часто встречается некорректный перевод термина «*strong typing*» как «*строгая типизация*»; корректный вариант «*сильная типизация*» используется лишь при противопоставлении «*слабой типизации*». Следует иметь в виду, что использование термина «*строгий*» в отношении системы типов языка может вызвать путаницу со строгой семантикой вычислений языка (англ. *strict evaluation*).

В 1974 году Лисков и Зиллес (англ. *Liskov and Zilles*) назвали сильно типизированными те языки, в которых «*при передаче объекта из вызывающей функции в вызываемую тип этого объекта должен быть совместим с типом, определённым в вызываемой функции*». Джексон писал: «*В сильно типизированном языке всякая ячейка данных будет иметь уникальный тип и всякий процесс будет провозглашать свои требования по взаимосвязи в терминах этих типов*».

В статье Луки Карделли «Полнотиповое программирование», система типов называется «сильной», если она исключает возможность возникновения ошибки согласования типов времени выполнения. Иначе говоря, отсутствие непроконтролированных ошибок времени выполнения называется типобезопасностью; ранние работы Хоара называют это свойство безопасностью (англ. *security*).

Определение «сильной» и «слабой» типизации

«Сильной» и «слабой» типизацией называется продукт множества решений, принятых при разработке языка. Более точно языки характеризуются наличием или отсутствием безопасности согласования типов и безопасности доступа к памяти, а также

характерным временем осуществления такого контроля (в статике или в динамике).

Например, яркими примерами **слабой** системы типов являются те, что лежат в основе языков Си и C++. Их характерными атрибутами являются понятия приведения типов и каламбуров типизации. Эти операции поддерживаются на уровне компилятора и часто вызываются неявно. Операция `reinterpret_cast` в C++ позволяет представить элемент данных любого типа как принадлежащий любому другому типу при условии равенства длины их низкоуровневой реализации (битового представления) и изменить его состояние образом, недопустимым для исходного типа. Неосторожное использование таких операций нередко является источником крахов программ. Несмотря на это, в учебной литературе по C++ его система типов описывается как «*сильная*», что, с учётом тезисов Луки Карделли и других, следует понимать как «*более сильная, чем в Си*». В противоположность этому, в языках, типизированных по Хиндли — Милнеру, понятие о *приведении* типов отсутствует в принципе. Единственным способом «преобразовать» тип является написание функции, которая алгоритмически *строит* значение требуемого типа на основе значения исходного типа. Для тривиальных случаев, таких как «преобразование» целого без знака в целое со знаком и наоборот, такие функции обычно входят в состав стандартных библиотек. Наиболее часто используемым случаем такого рода функций являются специальные определяемые функции с пустым телом, называемые конструирующими функциями или просто конструкторами.

При этом система типов Хиндли — Милнера обеспечивает чрезвычайно высокий показатель повторного использования кода за счёт параметрического полиморфизма. Сильная, но не полиморфная система типов может затруднить решение многих алгоритмических задач, как это было отмечено в отношении языка Pascal.

В теории программирования сильная типизация является непременным элементом обеспечения надёжности разрабатываемых программных средств. При правильном применении (подразумевающем, что в программе объявляются и используются отдельные типы данных для логически несовместимых значений) она защищает программиста от простых, но труднообнаруживаемых ошибок, связанных с совместным использованием логически несовместимых значений, возникающих

иногда просто из-за элементарной опечатки.

Подобные ошибки выявляются ещё на этапе компиляции программы, тогда как при возможности неявного приведения практически любых типов друг к другу (как, например, в классическом языке Си) эти ошибки выявляются только при тестировании, причём не все и не сразу, что порой очень дорого обходится на этапе промышленной эксплуатации.

Python является одним из примеров языка с сильной динамической типизацией.

2.6. Алгебраический тип данных

Алгебраи́ческий тип да́нных — в информатике наиболее общий составной тип, представляющий собой тип-сумму из типов-произведений. Алгебраический тип имеет набор конструкторов, каждый из которых принимает на вход значения определённых типов и возвращает значение конструируемого типа. Конструктор представляет собой функцию, которая строит значение своего типа на основе входных значений. Для последующего извлечения этих значений из алгебраического типа используется сопоставление с образцом.

Простым примером алгебраического типа данных является список. Действительно, список имеет два конструктора — конструктор пустого списка и конструктор пары, первым элементом которой является значение определённого типа, а вторым — список. Пример определения списка на языке Haskell:

```
data List a = Nil
            | Cons a (List a)
```

Так что видно, что алгебраические типы данных являются контейнерными типами — они содержат внутри себя значения других типов (или того же самого типа). То, что у списка первый конструктор не принимает на вход каких-либо параметров, не должно вводить в сомнение. Такая форма конструктора является необходимой для

создания значений, которые внутри себя не содержат ничего, но являются «единичными» элементами алгебраических типов данных.

Специальными разновидностями алгебраических типов данных являются декартовы типы (они имеют только один конструктор) и перечисления (у них все конструкторы аргументов не имеют вовсе, хотя самих конструкторов может быть несколько). Так простейшим, но очень широко используемым перечислением является логический тип. Код на Haskell:

```
data Bool = False | True
```

Также алгебраический тип данных может быть абстрактным, если такой тип определён в некотором модуле, из которого не экспортируются конструкторы соответствующего типа, а доступ к значениям внутри алгебраического типа данных осуществляется при помощи специальных методов — селекторов. Особо стоит отметить так называемые «обобщённые алгебраические типы данных», которые реализованы в языках Haskell и ML.

Остаётся отметить, что с точки зрения синтаксически-ориентированного конструирования данных алгебраическим типом данных является размеченное объединение декартовых произведений типов. Каждое слагаемое в размеченном объединении соответствует одному конструктору, а каждый конструктор в свою очередь определяет декартово произведение типов, соответствующих параметрам конструктора. Конструкторы без параметров являются пустыми произведениями. Если алгебраический тип данных является рекурсивным, всё размеченное объединение обёртывается рекурсивным типом, и каждый конструктор возвращает рекурсивный тип.

Реализация

Язык Haskell

В языке Haskell любой тип данных, который не является примитивным, является алгебраическим. Все возможные виды значений (перечисления, объекты, структуры и т. д.) строятся при помощи конструкторов алгебраических типов данных. Поэтому

рассматриваемая тема является чрезвычайно важной для понимания системы типизации языка Haskell.

Язык Nemerle

В языке Nemerle существует ключевое слово «variant», с помощью которого можно описать алгебраический тип данных. Все созданные таким путём варианты могут быть сопоставлены с образцом через ключевое слово «match».

Язык Haxe

В языке Haxe алгебраический тип данных реализуется при помощи анонимных типов и перечислений. В языке предусмотрено сопоставление с образцом, которое так же можно применить для работы с алгебраическим типом данных.

2.6.1. Конструктор (функциональное программирование)

В теории типов и функциональных языках программирования **конструктор алгебраического типа данных** или просто **конструктор** представляет собой функцию с пустым телом, конструирующую объект **алгебраического типа данных**. Оптимизирующие компиляторы исполняют эти функции статически, т.е. на этапе компиляции.

Алгебраические типы данных являются важным элементов языков, типизированных по Хиндли — Милнеру.

Пример

Простейшую структуру XML-документа в Standard ML можно определить следующим образом:

```
datatype simple_xml = Empty
                    | Word of string
```

```
                    | Tagged of string * simple_xml
list
```

Это определение алгебраического типа данных. Оно вводит в программу четыре идентификатора: нуль-арный конструктор типов `simple_xml` и три **конструктора** объектов этого алгебраического типа: нуль-арный `Empty`, унарный `Word` и бинарный `Tagged`. Последний принимает два параметра (в данном случае в виде кортежа), второй из которых имеет тип `simple_xml list` (т.е. список объектов определяемого здесь типа). Таким образом, `simple_xml` представляет собой рекурсивный тип данных[en].

Конструкторы обладают всеми правами функций (например, конструктор `Word` имеет функциональный тип «`string -> simple_xml`»), и в частности, могут использоваться в абстракции функций.

```
fun listOfWords s =
   map Word (String.tokens Char.isSpace s)

fun toString e =
   let val scat = String.concat in
      case e of
         Empty => ""
       | Word s => s ^ " "
       | Tagged (tag, contents) => scat [
"<",tag,">", scat (map toString contents),
"</",tag,">" ]
   end
```

В теле функции `listOfWords` можно видеть как конструктор `Word` передаётся в качестве параметра функции `map`, и та применяет его к каждому элементу списка строк, который она получает вторым параметром. Список строк, в свою очередь, получен токенизацией (в данном случае просто разбиением на слова) строки, которую получила функция `listOfWords` входным параметром.

Каждое применение конструктора `Word` к объекту типа «строка» порождает объект типа `simple_xml`. Эти порождённые объекты затем используются для построения списка (это происходит внутри

функции `map`) — таким образом, результатом функции `listOfWords` будет список объектов типа `simple_xml`. Это подтверждается её функциональным типом, который выводит компилятор: «`string -> simple_xml list`». Соответственно, результат функции может непосредственно использоваться в качестве параметра для другого конструктора данного типа — `Tagged` — что породит новый объект типа `simple_xml`:

```
fun mkXmlFile s = Tagged ( "main", listOfWords s )
```

Таким образом, XML-документ строится посредством рекурсивной композиции конструкторов алгебраического типа (отсюда и название «*рекурсивный тип данных*»). Например, такой документ

```
<main>Here is some text</main>
```

будет представлен в программе такой структурой данных:

```
Tagged ( "main", [Word "Here", Word "is", Word
"some", Word "text"] )
```

В этой записи перемешивается использование конструкторов двух типов — `simple_xml` и `list`. Синтаксис «`[ , , ]`», конструирующий список, является на самом деле синтаксическим сахаром над цепочкой конструкторов типа `list`:

```
Tagged ( "main", Word "Here" :: Word "is" :: Word
"some" :: Word "text" :: nil )
```

Хотя тип `list` и встроен во все X-M-типизированные языки, но формально он определяется как рекурсивный тип данных[en] с нульарным конструктором `nil` и бинарным конструктором `cons`, который обычно имеет инфиксное символьное имя (два двоеточия в классических диалектах ML или одно в Хаскеле):

```
datatype 'a list = nil | :: of 'a * 'a list
infixr 5 ::
```

2.6.2. Типобезопасность и защита адресации памяти

В информатике **типобезопасность** (англ. *type safety*) языка программирования означает безопасность (или надёжность) его системы типов.

Система типов называется **безопасной** (англ. *safe*) или **надежной** (англ. *sound*), если в программах, прошедших проверку согласования типов, (англ. *well-typed programs* или *well-formed programs*) исключена возможность возникновения ошибок согласования типов во время выполнения.

Ошибка согласования типов или **ошибка типизации** (англ. *type error*) представляет собой несогласованность типов компонентов выражений в программе, например попытку использовать целое число в роли функции. Пропущенные ошибки согласования типов на этапе выполнения могут приводить к багам и даже крахам программ. Безопасность языка не является синонимом полного отсутствия багов, но, по меньшей мере, они становятся исследуемы в рамках семантики языка (формальной или неформальной).

Надёжные системы типов также называют **сильными** (англ. *strong*), но трактовка этого термина часто смягчается, кроме того, его часто применяют к языкам, осуществляющим динамическую проверку согласования типов (см. сильная и слабая типизация).

Иногда безопасность рассматривается как свойство конкретной программы, а не языка, на котором она написана — по той причине, что некоторые типобезопасные языки разрешают обойти или нарушить систему типов, если программист практикует скудную типобезопасность. Распространено мнение, что такие возможности на практике оказываются необходимостью, но это вымысел. Понятие о «безопасности программы» важно в том смысле, что реализация безопасного языка сама может быть небезопасной. Раскрутка компилятора решает эту проблему, обеспечивая языку безопасность не только в теории, но и на практике.

Робину Милнеру принадлежит выражение «Программы, прошедшие проверку типов, не могут „сбиться с пути истинного“». Иначе говоря, безопасная система типов предотвращает заведомо ошибочные операции, связанные с неверными типами. Например, выражение `3 / "Hello, World"` очевидно является ошибочным, поскольку арифметика не определяет операцию деления числа на строку. Формально, безопасность означает гарантию того, что значение любого выражения, прошедшего проверку типов, и имеющего тип `T`, является *истинным* элементом множества значений `T`, то есть будет лежать в границах диапазона значений, допустимого статическим типом этого выражения. На самом деле, в этом требовании есть нюансы — см. подтипы и полиморфизм для сложных случаев.

Кроме того, при интенсивном использовании механизмов определения новых типов предотвращаются логические ошибки, проистекающие из семантики различных типов. Например, и миллиметры, и дюймы могут представляться целыми числами, но будет ошибкой вычитать дюймы из миллиметров, и развитая система типов не допустит этого (разумеется, при условии, что программист описал различие на уровне типов, а не на уровне имён переменных целого типа). Другими словами, безопасность языка означает, что язык защищает программиста от его собственных возможных ошибок.

Многие языки системного программирования (например, Ada, Си, C++) предусматривают *ненадёжные* (англ. *unsound*) или *небезопасные* (англ. *unsafe*) операции, предназначенные для возможности нарушить (англ. *violate*) систему типов — см. приведение типа и каламбур типизации. В одних случаях это допускается лишь в ограниченных частях программы, в других — неотличимо от хорошо типизированных операций.

В мейнстриме нередко встречается сужение понятия типобезопасности до «*безопасности типов в отношении доступа к памяти*» (англ. *memory type safety*), означающее, что компоненты объектов одного агрегатного типа не могут обращаться к ячейкам памяти, выделенным под объекты другого типа. Безопасность доступа к памяти означает запрещение возможности скопировать произвольную цепочку бит из одной области памяти в другую. Например, если язык предусматривает тип `t`, имеющий ограниченный спектр допустимых значений, и предоставляет возможность скопировать нетипизированные данные в переменную этого типа, то это не является

типобезопасным, поскольку потенциально допускает, что переменная типа `t` будет иметь значение, не являющееся допустимым для типа `t`. И, в частности, если такой небезопасный язык позволяет записать произвольное целое значение в переменную, имеющую тип «указатель», то небезопасность доступа к памяти очевидна (см. Висячий указатель). Примерами небезопасных языков служат Си и C++[4]. В сообществах этих языков часто называют «безопасными» любые операции, непосредственно не приводящие к краху программы. Безопасность доступа к памяти также означает предотвращение возможности переполнения буфера, например, попытки записи крупноразмерных объектов в ячейки, выделенные для объектов другого типа меньшего размера.

Надёжные статические системы типов **консервативны** (избыточны) в том смысле, что отвергают даже программы, которые исполнились бы корректно. Причина этого заключается в том, что для любого Тьюринг-полного языка, множество программ, которые могут порождать ошибки согласования типов во время выполнения, алгоритмически неразрешимо (см. проблема остановки). Как следствие, такие системы типов обеспечивают степень защиты, существенно более высокую, чем это необходимо для обеспечения безопасности доступа к памяти. С другой стороны, безопасность некоторых действий не может быть доказана статически и должна контролироваться динамически — например, индексация массива с произвольным доступом. Такой контроль может осуществляться либо рантайм-системой самого языка, либо непосредственно функциями, реализующими подобные потенциально небезопасные операции.

(**Среда выполнения** (англ. *execution environment*, иногда *«рантáйм»* от англ. *runtime* — «время выполнения») в информатике — вычислительное окружение, необходимое для выполнения компьютерной программы и доступное во время выполнения компьютерной программы. В среде выполнения, как правило, невозможно изменение исходного текста программы, но может наличествовать доступ к переменным окружения операционной системы, таблицам объектов и модулей разделяемых библиотек.

Взаимодействие со средой выполнения для интерпретируемых языков программирования реализуется непосредственно в интерпретаторе, обеспечивающим взаимодействие конструкций языка с окружением, в котором он запущен. Для компилируемых языков взаимодействие с

вычислительным окружением может реализовываться набором подключаемых разделяемых библиотек среды выполнения либо целиком в виртуальной машине, выполняющей промежуточный код, в который компилируется программа.)

Сильно динамически типизируемые языки (например, Lisp , Smalltalk) не допускают повреждения данных за счёт того, что программа, пытающаяся преобразовать значение к несовместимому типу, порождает исключение. К достоинствам сильной динамической типизации перед типобезопасностью можно отнести отсутствие консервативности, и, как следствие, расширение спектра решаемых задач программирования. Ценой этого становится неизбежное снижение быстродействия программ, а также необходимость существенно бóльшего количества пробных запусков для выявления возможных ошибок. Поэтому многие языки комбинируют возможности статического и динамического контроля согласования типов тем или иным образом.

2.6.3. Примеры безопасных языков

Ada

Ada (наиболее типобезопасный язык в семействе Pascal) ориентирована на разработку надёжных встраиваемых систем, драйверов и других задач системного программирования. Для реализации критичных секций Ada предоставляет ряд небезопасных конструкций, имена которых обычно начинаются с `Unchecked_`.

Язык SPARK является подмножеством Ады. Из него устранены небезопасные возможности, но добавлены возможности проектирования по контракту. SPARK исключает возможность возникновения висячих указателей посредством исключения самой возможности динамического выделения памяти. Статически проверяемые контракты были добавлены в Ada2012.

Хоар в своей лекции на премию Тьюринга утверждал, что для обеспечения надёжности мало статических проверок — надёжность в первую очередь является следствием *простоты*. Тогда же он предсказал, что сложность Ады станет причиной катастроф.

BitC

BitC представляет собой гибридный язык, комбинирующий низкоуровневые возможности Си с безопасностью Standard ML и лаконичностью Scheme. BitC ориентирован на разработку надёжных встраиваемых систем, драйверов и решение других задач системного программирования.

Cyclone

Cyclone является безопасным диалектом языка Си, заимствующим многие идеи из ML (включая типобезопасный параметрический полиморфизм). Cyclone предназначен для тех же задач, что и Си, и после осуществления всех проверок компилятор порождает код на ANSI C. Cyclone не требует виртуальной машины или сборки мусора для обеспечения динамической безопасности — вместо этого он основан на управлении памятью посредством регионов. Cyclone устанавливает более высокую планку требований безопасности исходного кода, и из-за небезопасной природы Си портирование даже простых программ с Си на Cyclone может потребовать определённой работы, хотя немалая её часть может быть проделана в рамках Си до смены компилятора. Поэтому Cyclone часто определяют не как диалект Си, а как «*язык, синтаксически и семантически похожий на Си*».

Haskell

Haskell (потомок ML) изначально разрабатывался как полнотиповый чистый язык, что должно было сделать поведение программ на нём ещё более предсказуемым, чем на ранних диалектах ML . Однако, позже в *стандарте языка* были предусмотрены небезопасные операции, необходимые в повседневной практике, хотя и отмеченные соответствующими идентификаторами (начинающимися с `unsafe`).

Haskell также предоставляет возможности слабой динамической типизации, и в стандарт языка была включена реализация механизма обработки исключений посредством этих возможностей. Её использование может приводить к аварийному завершению программ, за что Роберт Харпер назвал Хаскел «*исключительно небезопасным*». Он считает неприемлемым тот факт, что исключения имеют тип, определённый пользователем в контексте класса типов `Typeable`, с учётом того, что исключения генерируются безопасным кодом (за

пределами монады `IO`); и классифицирует выдаваемое компилятором сообщение о внутренней ошибке как несоответствующее слогану Милнера . В последовавшем обсуждении было показано, как можно было бы реализовать в Хаскеле статически типобезопасные исключения в стиле Standard ML .

Lisp

«Чистый» (минимальный) Lisp представляет собой однотиповый язык (то есть любая конструкция принадлежит к типу «S-выражение»), хотя даже первые промышленные реализации Lisp предусматривали как минимум определённое количество атомов. Семейство потомков языка Lisp представлено по большей степени сильно динамически типизируемыми языками, но существуют статически типизируемые и сочетающие обе формы типизации.

Common Lisp является сильно динамически типизируемым языком, но предусматривает возможность явно (манифестно) назначать типы (а компилятор SBCL способен сам их выводить), однако, эта возможность используется для оптимизации и усиления динамических проверок и не означает статическую типобезопасность. Программист может установить для компилятора сниженный уровень динамических проверок с целью повышения быстродействия, и скомпилированная таким образом программа уже не может считаться безопасной.

Язык Scheme также является сильно динамически типизируемым языком, но компилятор Stalin статически выводит типы, используя их для агрессивной оптимизации программ. Языки «Typed Racket» (расширение Racket Scheme) и Shen типобезопасны. Clojure сочетает сильный динамический и статический контроль типов.

ML

Язык ML изначально разрабатывался в качестве интерактивной системы доказательства теорем, и лишь впоследствии стал самостоятельным компилируемым языком общего назначения. Много усилий было уделено доказательству надёжности параметрически полиморфной системы типов Хиндли-Милнера, лежащей в основе ML. Доказательство надёжности построено для чисто функционального подмножества («Fuctional ML»), расширения ссылочными типами («Reference ML»), расширения исключениями («Exception ML»), для

языка, объединяющего все эти расширения («Core ML») и наконец, его расширения первоклассными продолжениями («Control ML»), сперва мономорфными, затем полиморфными.

Следствием этого стало то, что потомки ML зачастую априори считаются типобезопасными, даже несмотря на то, что некоторые из них откладывают значимые проверки на этап выполнения программы (OCaml), либо отклоняются от семантики, для которой построено доказательство надёжности, и содержат небезопасные возможности явным образом (Haskell). Для языков семейства ML характерна развитая поддержка алгебраических типов данных, использование которых существенно способствует предотвращению логических ошибок, что также поддерживает впечатление типобезопасности.

Некоторые потомки ML так же являются инструментами интерактивного доказательства (Idris, Mercury, Agda). Многие из них, хотя и могли бы использоваться для непосредственной разработки программ с доказанной надёжностью, чаще используются для верификации программ на других языках — из-за таких причин как высокая трудоёмкость использования, низкое быстродействие, отсутствие FFI и прочее. Среди потомков ML с доказанной надёжностью выделяются как ориентированные на промышленное применение языки Standard ML и прототип его дальнейшего развития successor ML (ранее известный как «ML2000»).

Standard ML

Язык Standard ML (старший в семействе языков ML) ориентирован на разработку крупномасштабных программ промышленного быстродействия. Язык имеет строгое формальное определение и его статическая и динамическая безопасность доказана. После статической проверки согласованности интерфейсов компонентов программы (в том числе порождаемых — см. функторы ML), дальнейший контроль безопасности поддерживается рантайм-системой. Как следствие, даже содержащая ошибку программа на Standard ML всегда продолжает вести себя как ML-программа: она может навечно уйти в расчёты или выдать пользователю сообщение об ошибке, но она *не может* обрушиться.

Однако, некоторые реализации (SML/NJ, Mythryl, MLton) включают *нестандартные библиотеки*, предоставляющие определённые

небезопасные операции (их идентификаторы начинаются с `Unsafe`). Эти возможности необходимы для внешнеязыкового интерфейса этих реализаций, обеспечивающего взаимодействие с не-ML-кодом (обычно это код на Си, реализующий критичные по скорости компоненты программ), который может требовать своеобразного битового представления данных. Кроме того, многие реализации Standard ML, хотя сами написаны на нём самом, используют рантайм-систему, написанную на Си. Другим примером является режим REPL компилятора SML/NJ, использующий небезопасные операции для исполнения интерактивно вводимого программистом кода.

Язык Alice является расширением Standard ML, предоставляя возможности сильной динамической типизации.

2.6.4. Типизированные и бестиповые языки

Язык называют типизированным, если спецификация каждой операции определяет типы данных, к которым эта операция может применяться, подразумевая её неприменимость к иным типам. Например, данные, которые представляет «*этот текст в кавычках*», имеют тип «`строка`». В большинстве языков программирования деление числа на строку не имеет смысла, и большинство современных языков отвергнет программу, которая пытается выполнить такую операцию. В одних языках бессмысленная операция будет выявлена в процессе компиляции (статическая типизация), и отвергнута компилятором. В других она будет выявлена во время выполнения программы (динамическая типизация), порождая исключительную ситуацию.

Особый случай типизированных языков представляют однотиповые языки (англ. *single-type language*), то есть языки с единственным типом. Обычно это языки сценариев или разметки, такие как REXX и SGML, единственным типом данных в которых является символьная строка, используемая для представления как символьных, так и числовых данных.

Бестиповые языки, в противоположность типизированным, позволяют осуществлять любую операцию над любыми данными, которые в них представляются цепочками бит произвольной длины. Бестиповыми является большинство языков ассемблера. Примерами высокоуровневых бестиповых языков служат BCPL, BLISS, Forth, Рефал.

На практике, лишь некоторые языки могут считаться типизированными с точки зрения теории типов (разрешая или отвергая все операции), большинство современных языков предлагают лишь некую степень типизированности. Многие промышленные языки предоставляют возможность обойти или нарушить систему типов, поступаясь типобезопасностью ради более точного контроля над исполнением программы (каламбур типизации).

2.7. Абстрактрый тип данных

2.7.1. Список

В информатике, **спи́сок** (англ. *list*) — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза. Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности. Экземпляры значений, находящихся в списке, называются **элементами** списка (англ. *item, entry* либо *element*); если значение встречается несколько раз, каждое вхождение считается отдельным элементом.

Структура односвязного списка из трёх элементов

Термином *список* также называется несколько конкретных структур данных, применяющихся при реализации абстрактных списков, особенно связных списков.

При помощи нотации метода синтаксически-ориентированного конструирования Ч. Хоара определение списка можно записать следующим образом:

$$List(A) = NIL + (A \times List(A))$$
prefix=constructor *List(A)*
head,tail=selectors *List(A)*
nul, nonnul=predicates *List(A)*
NIL , non NIL =past *List(A)*

Первая строка данного определения обозначает, что список элементов типа A(говорят: «список над A») представляет собой размеченное объединение пустого списка и декартова произведения атома типа A со

списком над A. Для создания списков используются два конструктора (вторая строка определения), первый из которых создаёт пустой список, а второй — непустой соответственно. Вполне понятно, что второй конструктор получает на вход в качестве параметров некоторый атом и список, а возвращает список, первым элементом которого является исходный атом, а остальными — элементы исходного списка. То есть получается префиксация атома к списку, с чем и связано такое наименование конструктора. Здесь необходимо отметить, что пустой список не является атомом, а потому не может префиксироваться. С другой стороны, пустой список является как бы нулевым элементом для конструирования списков, поэтому любой список содержит в самом своём конце именно пустой список — с него начинается конструирование.

Третья строка определяет селекторы для списка, то есть операции для доступа к элементам внутри списка. Селектор *head* получает на вход список и возвращает первый элемент этого списка, то есть типом результата является тип A. Этот селектор не может получить на вход пустой список — в этом случае результат операции неопределён. Селектор *tail* возвращает список, полученный из входного в результате отсечения его головы (первого элемента). Этот селектор также не может принимать на вход пустой список, так как в этом случае результат операции неопределён. При помощи этих двух операций можно достать из списка любой элемент. Например, чтобы получить третий элемент списка (если он имеется), необходимо последовательно два раза применить селектор *tail*, после чего применить селектор *head*. Другими словами, для получения элемента списка, который находится на позиции *n* (начиная с 0 для первого элемента, как это принято в программировании), необходимо *n* раз применить селектор *tail*, после чего применить селектор *head*.

Четвёртая строка определения описывает предикаты для списка, то есть функции, возвращающие булевское значение в зависимости от некоторых условий. Первый предикат возвращает значение *true* в случае, если заданный список пуст. Второй предикат действует наоборот. Наконец, пятая строка описывает части списка, которые, как уже сказано, представляют собой пустой и непустой списки.

Свойства

У определённой таким образом структуры данных имеются некоторые свойства:

- **Размер списка** — количество элементов в нём, исключая последний «нулевой» элемент, являющийся по определению пустым списком.

- **Тип элементов** — тот самый тип , над которым строится список; все элементы в списке должны быть этого типа.
- **Отсортированность** — список может быть отсортирован в соответствии с некоторыми критериями сортировки (например, по возрастанию целочисленных значений, если список состоит из целых чисел).
- **Возможности доступа** — некоторые списки в зависимости от реализации могут обеспечивать программиста селекторами для доступа непосредственно к заданному по номеру элементу.
- **Сравниваемость** — списки можно сравнивать друг с другом на соответствие, причём в зависимости от реализации операция сравнения списков может использовать разные технологии.

Как должно быть понятно из названия рассматриваемой структуры данных, списки используются для хранения наборов однотипных элементов. Такие элементы могут быть отсортированы для использования в функциях поиска или функциях для быстрой вставки новых элементов в список.

Списки в языках программирования

Функциональные языки

Списки в функциональных языках являются фундаментальной структурой. Большинство функциональных языков имеет встроенные средства для работы со списками вроде получения длины списка, головы (первый элемент списка), хвоста (часть списка, идущая за первым элементом), применения функции к каждому элементу списка (Map), свертки списка и пр.

Язык Haskell

Язык Lisp

Императивные языки

2.7.2. XOR-связный список

XOR-связный список — структура данных, похожая на обычный двусвязный список, однако в каждом элементе хранится только *один* адрес — результат выполнения операции XOR над адресами предыдущего и следующего элементов списка. Для того, чтобы перемещаться по списку, необходимо взять два последовательных адреса и выполнить над ними операцию XOR, которая и даст реальный адрес следующего элемента.

Сравнения с двусвязным списком

Классический двусвязный список хранит отдельно адреса предыдущего и следующего элемента списка, для хранения которых требуется два указателя:

```
...  A       B         C         D         E  ...
       ->  next  ->  next  ->  next  ->
       <-  prev  <-  prev  <-  prev  <-
```

Накладные расходы XOR-связного списка в два раза меньше, так как в нём хранится только один «адрес» — XOR указателей на предыдущий и следующий элементы:

```
...  A       B         C         D         E  ...
       <->  A⊕C  <->  B⊕D  <->  C⊕E  <->
```

2.7.3. Связный список

Свя́зный спи́сок — базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость:

порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Виды связных списков

Линейный связный список

Односвязный список (однонаправленный связный список)

Разновидность связного списка — односвязный список, содержащий 3 элемента

Линейный однонаправленный список — это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на NULL. Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

В информатике **линейный список** обычно определяется как абстрактный тип данных (АТД), формализующий понятие упорядоченной коллекции данных. На практике линейные списки обычно реализуются при помощи массивов и связных списков. Иногда термин «список» неформально используется также как синоним понятия «связный список». К примеру, АТД нетипизированного изменяемого списка может быть определён как набор из конструктора и основных операций:

- Операция, проверяющая список на пустоту.
- Три операции добавления объекта в список (в начало, конец или внутрь после любого (n-го) элемента списка);
- Операция, вычисляющая первый (головной) элемент списка;
- Операция доступа к списку, состоящему из всех элементов исходного списка, кроме первого.

Характеристики

- **Длина списка**. Количество элементов в списке.
- Списки могут быть **типизированными** или **нетипизированными**. Если список типизирован, то тип его элементов задан, и все его элементы должны иметь типы, совместимые с заданным типом элементов списка. Чаще списки типизированы.
- Список может быть **сортированным** или **несортированным**.
- В зависимости от реализации может быть возможен **произвольный доступ** к элементам списка.

Односвязный список в языках программирования

Си

```
typedef struct s_list
{
  int field; // поле данных
  struct s_list *next; // указатель на следующий
элемент
} list;
```

применение односвязного списка:

```
list* l1 = (list*)malloc(sizeof(list));
l1->field = 1;
l1->next = (list*)malloc(sizeof(list));
l1->next->field = 2;
l1->next->next = (list*)malloc(sizeof(list));
/* и т.д. */
```

Двусвязный список (двунаправленный связный список)

Здесь ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом дает возможность перемещения в обе стороны. В этом списке проще производить удаление и перестановку элементов, так как легко

доступны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

Кольцевой связный список

Разновидностью связных списков является кольцевой (циклический, замкнутый) список. Он тоже может быть односвязным или двусвязным. Последний элемент кольцевого списка содержит указатель на первый, а первый (в случае двусвязного списка) — на последний.

Как правило, такая структура реализуется на базе линейного списка. С каждым кольцевым списком дополнительно хранится указатель на первый элемент. В этом списке ссылки на NULL не встречается.

Также существуют циклические списки с выделенным головным элементом, облегчающие полный проход через список.

Достоинства

- эффективное (за константное время) добавление и удаление элементов
- размер ограничен только объёмом памяти компьютера и разрядностью указателей
- динамическое добавление и удаление элементов

Недостатки

Недостатки связных списков вытекают из их главного свойства — последовательного доступа к данным:

- сложность прямого доступа к элементу, а именно определения физического адреса по его индексу (порядковому номеру) в списке
- на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны)
- некоторые операции со списками медленнее, чем с массивами, так как к произвольному элементу списка можно обратиться, только пройдя все предшествующие ему элементы

- соседние элементы списка могут быть распределены в памяти нелокально, что снизит эффективность кэширования данных в процессоре
- над связными списками, по сравнению с массивами, гораздо труднее (хоть и возможно) производить параллельные векторные операции, такие, как вычисление суммы: накладные расходы на перебор элементов снижают эффективность распараллеливания

Список с пропусками

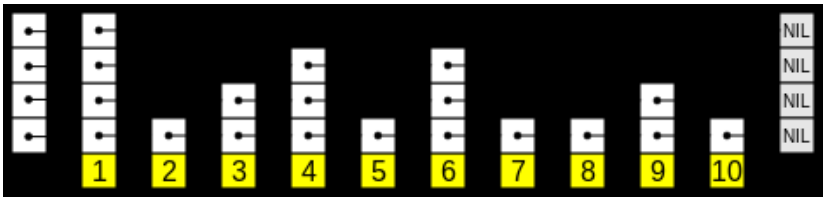
Материал из Википедии — свободной энциклопедии
Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 2 апреля 2017; проверки требует 1 правка.

Список с пропусками (англ. *Skip List*) — вероятностная структура данных, основанная на нескольких параллельных отсортированных связных списках с эффективностью, сравнимой с двоичным деревом (порядка O(log n) среднее время для большинства операций).

В основе списка с пропусками лежит расширение отсортированного связного списка дополнительными связями, добавленными в случайных путях с геометрическим/негативным биномиальным распределением[1], таким образом, чтобы поиск по списку мог быстро пропускать части этого списка. Вставка, поиск и удаление выполняются за логарифмическое случайное время.

Описание

Список с пропусками — это несколько слоев. Нижний слой — это обычный упорядоченный связный список. Каждый более высокий слой представляет собой «выделенную полосу движения» для списков ниже, где элемент в i-ом слое появляется в i+1-м слое с некоторой фиксированной вероятностью p (два наиболее часто используемых значений для p — 1/2 и 1/4). В среднем каждый элемент встречается в $1/(1-p)$ списках, и верхний элемент (обычно специальный головной элемент в начале списка с пропусками) в $\log_{1/p} n$ списках.



Поиск нужного элемента начинается с головного элемента верхнего списка, и выполняется горизонтально до тех пор, пока текущий элемент не станет больше либо равен целевому. Если текущий элемент равен целевому, он найден. Если текущий элемент больше, чем целевой, процедура повторяется после возвращения к предыдущему элементу и спуска вниз вертикально на следующий нижележащий список. Ожидаемое число шагов в каждом связном списке $1/p$, что можно увидеть, просматривая путь поиска назад с целевого элемента, пока не будет достигнут элемент, который появляется в следующем более высоком списке. Таким образом, общие *ожидаемые* затраты на поиск — $(\log_{1/p} n)/p$, равные $O(\log n)$ в случае константного p. Выбирая разные значения p, возможно выбирать необходимый компромисс между затратами на время поиска и затратами памяти на хранение списка.

Детали реализации

Элементы, используемые в списке с пропусками, могут содержать более одного указателя, таким образом они могут состоять в более чем одном списке.

Операции удаления и вставки реализованы весьма похоже на аналогичные операции связного списка, с тем исключением, что «высокие» должны быть вставлены или удалены более чем из одного связного списка.

Однако без рандомизации эти операции приводили бы к очень низкой производительности, так как необходимо было бы полностью перетасовывать список при добавлении нового элемента, чтобы сохранить число пропусков на верхнем уровне константным. William Pugh предложил следующий алгоритм для решения, на какую высоту должен быть продвинут новый элемент. Этот алгоритм требует лишь локальных изменений списка при добавлении новых элементов и позволяет сохранять эффективность «экспресс-линий» (l —

результирующее значение уровня, на который нужно помещать элемент):

```
l = 1
пока случайное значение в диапазоне [0,1] < p и l < максимального уровня
  l = l + 1
```

В итоге элемент вставляет на l-ый уровень и, соответственно, на все уровни меньше l.

Θ(n) операций, которые необходимы нам для посещения каждого узла в возрастающем порядке (например, печать всего списка), предоставляют возможность выполнить незаметную дерандомизацию структуры уровней списка с пропусками оптимальным путём, достигая $O(\log n)$ времени поиска для связного списка. (выбирая уровень i-го конечного узла 1 плюс количество раз, которое мы можем поделить i на 2, пока оно не станет нечетным. Также i=0 для отрицательно бесконечного заголовка, как мы имеем, обычный специальный случай, выбирая максимально возможный уровень для отрицательных и/или положительных бесконечных узлов.) Тем не менее, это позволяет узнать кому-нибудь, где все узлы с уровнем более 1, и затем удалить их.

В качестве альтернативы мы можем сделать структуру уровней квази-случайной следующим путём:

```
создать все узлы уровня 1
j = 1
пока количество узлов на уровне j > 1
  для каждого i-го узла на уровне j
    если i нечетное
      если i не последний узел на уровне j
        случайно выбираем, продвигать ли его на уровень j+1
      иначе
        не продвигать
      конец условия
    иначе, если i четный узел i-1 не продвинут
      продвинуть его на уровень j+1
    конец условия
```

```
  конец цикла для
  j = j + 1
конец цикла пока
```

Так же, как дерандомизированная версия, квази-рандомизация выполняется только, когда есть какая-то другая причина выполнять Θ(n) операций (которые посетят каждый узел).

Пример реализации

[показать]
Код класса на C++

Развёрнутый связный список

Развёрнутый связный список — список, каждый физический элемент которого содержит несколько логических элементов (обычно в виде массива, что позволяет ускорить доступ к отдельным элементам).

Позволяет значительно уменьшить расход памяти и увеличить производительность по сравнению с обычным списком. Особенно большая экономия памяти достигается при малом размере логических элементов и большом их количестве — так, односвязный список из 10 тысяч четырёхбайтных целых чисел при четырёхбайтной же адресации памяти займет 40 тысяч байт под собственно значения, плюс 40 тысяч байт под адреса, итого 80 тысяч байт; если же объединить числа в 100 массивов по 100 элементов, расход памяти на адреса упадёт до 400 байт, и суммарный расход составит 40400 байт.

Прирост производительности достигается за счёт того, что большая часть операций проводится над относительно небольшими массивами, которые обычно целиком помещаются в кэш-памяти. Благодаря этому, быстродействие программы может быть даже выше, чем при работе с обычными массивами. В развёрнутый список легко можно добавлять новые элементы — без необходимости переписывать весь массив, что является большой проблемой при работе с обычными массивами.

При реализации необходимо тщательно выбирать размер «блока» (количество элементов в массивах). При слишком большом размере

блока список начинает страдать от тех же проблем, что и обыкновенный массив: долгая вставка элементов в начало или середину, долгое удаление элементов оттуда же, и т.п. При слишком маленьком — увеличивается расход памяти.

2.7.4. Очередь (программирование)

О́чередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, англ. *first in, first out*). Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.

Способы реализации очереди

Существует несколько способов реализации очереди в языках программирования.

Массив

Первый способ представляет очередь в виде массива и двух целочисленных переменных start и end.

Обычно `start` указывает на голову очереди, `end` — на элемент, который заполнится, когда в очередь войдёт новый элемент. При добавлении элемента в очередь в `q[end]` записывается новый элемент очереди, а `end` уменьшается на единицу. Если значение end становится меньше 1, то мы как бы циклически обходим массив, и значение переменной становится равным n. Извлечение элемента из очереди производится аналогично: после извлечения элемента `q[start]` из очереди переменная `start` уменьшается на 1. С такими алгоритмами одна ячейка из `n` всегда будет незанятой (так как очередь с `n` элементами невозможно отличить от пустой), что компенсируется простотой алгоритмов.

Преимущества данного метода: возможна незначительная экономия памяти по сравнению со вторым способом; проще в разработке.

Недостатки: максимальное количество элементов в очереди ограничено размером массива. При его переполнении требуется перевыделение памяти и копирование всех элементов в новый массив.

Связный список

Второй способ основан на работе с динамической памятью. Очередь представляется в качестве линейного списка, в котором добавление/удаление элементов идет строго с соответствующих его концов.

Преимущества данного метода: размер очереди ограничен лишь объёмом памяти.

Недостатки: сложнее в разработке; требуется больше памяти; при работе с такой очередью память сильнее фрагментируется; работа с очередью несколько медленнее.

Реализация на двух стеках

Методы очереди могут быть реализованы на основе двух стеков `S1` и `S2`, как показано ниже:

```
Процедура enqueue(x):
    S1.push(x)
Функция dequeue():
    если S2 пуст:
        если S1 пуст:
            сообщить об ошибке: очередь пуста
        пока S1 не пуст:
            S2.push(S1.pop())
    вернуть S2.pop()
```

Такой способ реализации наиболее удобен в качестве основы для построения персистентной очереди.

Очереди в различных языках программирования

Практически во всех развитых языках программирования реализованы очереди. В CLI для этого предусмотрен класс System.Collections.Queue

с методами Enqueue и Dequeue. В STL также присутствует класс queue<>, определённый в заголовочном файле queue. В нём используется та же терминология (push и pop), что и в стеках.

Применение очередей

Очередь в программировании используется, как и в реальной жизни, когда нужно совершить какие-то действия в порядке их поступления, выполнив их последовательно. Примером может служить организация событий в Windows. Когда пользователь оказывает какое-то действие на приложение, то в приложении не вызывается соответствующая процедура (ведь в этот момент приложение может совершать другие действия), а ему присылается сообщение, содержащее информацию о совершенном действии, это сообщение ставится в очередь, и только когда будут обработаны сообщения, пришедшие ранее, приложение выполнит необходимое действие.

Клавиатурный буфер BIOS организован в виде кольцевого массива, обычно длиной в 16 машинных слов, и двух указателей: на следующий элемент в нём и на первый незанятый элемент.

Двухсторонняя очередь

Материал из Википедии — свободной энциклопедии
Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 20 марта 2015; проверки требуют 2 правки.

Двусвязная очередь (жарг. *дэк, дек* от англ. *deque — double ended queue*; двухсторонняя очередь, очередь с двумя концами) — структура данных, в которой элементы можно добавлять и удалять как в начало, так и в конец, то есть дисциплинами обслуживания являются одновременно FIFO и LIFO.

Типовые операции

- PushBack — добавление в конец очереди.
- PushFront — добавление в начало очереди.
- PopBack — выборка с конца очереди.
- PopFront — выборка с начала очереди.

- Проверка наличия элементов.
- Очистка.

Очередь с приоритетом (программирование)

Очередь с приоритетом (англ. *priority queue*) — абстрактный тип данных в программировании, поддерживающий две обязательные операции — добавить элемент и извлечь максимум (минимум). Предполагается, что для каждого элемента можно вычислить его *приоритет* — действительное число или в общем случае элемент линейно упорядоченного множества.

Описание

Основные методы, реализуемые очередью с приоритетом, следующие:

- `insert(ключ, значение)` — добавляет пару `(ключ, значение)` в хранилище;
- `extract_minimum()` — возвращает пару `(ключ, значение)` с минимальным значением ключа, удаляя её из хранилища.

При этом меньшее значение ключа соответствует более высокому приоритету.

В некоторых случаях более естественен рост ключа вместе с приоритетом. Тогда второй метод можно назвать `extract_maximum()`.

Примеры

В качестве примера очереди с приоритетом можно рассмотреть список задач работника. Когда он заканчивает одну задачу, он переходит к очередной — самой приоритетной (ключ будет величиной, обратной приоритету) — то есть выполняет операцию извлечения максимума. Начальник добавляет задачи в список, указывая их приоритет, то есть выполняет операцию добавления элемента.

Расширения очереди с приоритетом

На практике интерфейс очереди с приоритетом нередко расширяют другими операциями:

- вернуть минимальный элемент без удаления из очереди
- изменить приоритет произвольного элемента
- удалить произвольный элемент
- слить две очереди в одну

В ***индексированных очередях с приоритетом*** (адресуемых) возможно обращение к элементам по индексу. Такие очереди могут быть использованы, например, для слияния нескольких отсортированных последовательностей (multiway merge).

Реализации

Очередь с приоритетами может быть реализована на основе различных структур данных.

Простейшие (и не очень эффективные) реализации могут использовать неупорядоченный или упорядоченный массив, связный список, подходящие для небольших очередей. При этом вычисления могут быть как «ленивыми» (тяжесть вычислений переносится на извлечение элемента), так и ранними (eager), когда вставка элемента сложнее его извлечения. То есть, одна из операций может быть произведена за

время , а другая — в худшем случае за , где N — длина очереди.

Более эффективными являются реализации на основе кучи, где обе

операции можно производить в худшем случае за время . К ним относятся двоичная куча, биномиальная куча, фибоначчиева куча.

Абстрактный тип данных (АТД) для очереди с приоритетом получается из АТД кучи переименованием соответствующих функций. Минимальное (максимальное) значение находится всегда на вершине кучи.

Пример на Python

Стандартная библиотека Python содержит модуль `heapq`, реализующий очередь с приоритетом:

```
# импорт двух функций очереди под именами,
принятыми в данной статье
from heapq import heappush as insert, heappop as
extract_maximum
pq = []  # инициализация списка
insert(pq, (4, 0, "p"))   # вставка в очередь
элемента "p" с индексом 0 и приоритетом 4
insert(pq, (2, 1, "e"))
insert(pq, (3, 2, "a"))
insert(pq, (1, 3, "h"))
# вывод четырёх элементов в порядке возрастания
приоритетов
print(extract_maximum(pq)[-1] +
extract_maximum(pq)[-1] + extract_maximum(pq)[-1] +
extract_maximum(pq)[-1])
```

Этот пример выведет слово «heap».

2.7.5.Стек

Стек (англ. *stack* — стопка; читается *стэк*) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу *LIFO* (англ. *last in — first out*, «последним пришёл — первым вышел»).

Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

В цифровом вычислительном комплексе стек называется магазином — по аналогии с магазином в огнестрельном оружии (стрельба начнётся с патрона, заряженного последним).

В 1946 Алан Тьюринг ввёл понятие стека. А в 1957 году немцы Клаус Самельсон и Фридрих Л. Бауэр запатентовали идею Тьюринга.

В некоторых языках (например, Lisp, Python) стеком можно назвать любой список, так как для них доступны операции pop и push. В языке C++ стандартная библиотека имеет класс с реализованной структурой и методами. И т. д.

Программный стек

Организация в памяти

Организация стека в виде одномерного упорядоченного по адресам массива. Показаны операции вталкивания и выталкивания данных из стека операциями *push* и *pop*.

Зачастую стек реализуется в виде однонаправленного списка (каждый элемент в списке содержит помимо хранимой информации в стеке указатель на следующий элемент стека).

Но также часто стек располагается в одномерном массиве с упорядоченными адресами. Такая организация стека удобна, если элемент информации занимает в памяти фиксированное количество слов, например, 1 слово. При этом отпадает необходимость хранения в элементе стека явного указателя на следующий элемент стека, что экономит память. При этом указатель стека (*Stack Pointer*, — **SP**) обычно является регистром процессора и указывает на адрес головы стека.

Предположим для примера, что голова стека расположена по меньшему адресу, следующие элементы располагаются по нарастающим адресам. При каждом вталкивании слова в стек, SP сначала уменьшается на 1 и затем по адресу из SP производится запись в память. При каждом извлечении слова из стека (выталкивании) сначала производится чтение по текущему адресу из SP и последующее увеличение содержимого SP на 1.

При организации стека в виде однонаправленного списка значением переменной стека является указатель на его вершину — адрес вершины. Если стек пуст, то значение указателя равно NULL.

Пример реализации стека на языке C:

```
struct stack
```

```
{
    char *data;
    struct stack *next;
};
```

Операции со стеком

Возможны три операции со стеком: добавление элемента (иначе проталкивание, *push*), удаление элемента (*pop*) и чтение головного элемента (*peek*).

При проталкивании (*push*) добавляется новый элемент, указывающий на элемент, бывший до этого головой. Новый элемент теперь становится головным.

При удалении элемента (*pop*) убирается первый, а головным становится тот, на который был указатель у этого объекта (следующий элемент). При этом значение убранного элемента возвращается.

```
void push( STACK *ps, int x ) // Добавление в стек
нового элемента
{
    if ( ps->size == STACKSIZE ) // Не переполнен
ли стек?
    {
        fputs( "Error: stack overflow\n", stderr );
        abort();
    }
    else
    {
        ps->items[ps->size++] = x;
    }
}

int pop( STACK *ps ) // Удаление из стека
{
    if ( ps->size == 0 ) // Не опустел ли стек?
    {
        fputs( "Error: stack underflow\n", stderr
);
        abort();
    }
```

```
    else
    {
        return ps->items[--ps->size];
    }
}
```

Область применения

Программный вид стека используется для обхода структур данных, например, дерево или граф. При использовании рекурсивных функций также будет применяться стек, но аппаратный его вид. Кроме этих назначений, стек используется для организации стековой машины, реализующей вычисления в обратной инверсной записи.

Для отслеживания точек возврата из подпрограмм используется стек вызовов.

Арифметические сопроцессоры, программируемые микрокалькуляторы и язык Forth используют стековую модель вычислений.

Идея стека используется в стековой машине среди стековых языков программирования.

Аппаратный стек

При вызове подпрограммы (процедуры) процессор помещает в стек адрес команды, следующей за командой вызова подпрограммы «адрес возврата» из подпрограммы. По команде возврата из подпрограммы из стека извлекается адрес возврата в вызвавшую подпрограмму программу и осуществляется переход по этому адресу.

Аналогичные процессы происходят при аппаратном прерывании (процессор X86 при аппаратном прерывании сохраняет автоматически в стеке ещё и регистр флагов). Кроме того, компиляторы размещают локальные переменные процедур в стеке (если в процессоре предусмотрен доступ к произвольному месту стека).

В архитектуре X86 аппаратный стек — непрерывная область памяти, адресуемая специальными регистрами ESP (указатель стека) и SS (селектор сегмента стека).

До использования стека он должен быть инициализирован так, чтобы регистры SS:ESP указывали на адрес головы стека в области физической оперативной памяти, причём под хранение данных в стеке необходимо зарезервировать нужное количество ячеек памяти (очевидно, что стек в ПЗУ, естественно, не может быть организован). Прикладные программы, как правило, от операционной системы получают готовый к употреблению стек. В защищённом режиме работы процессора сегмент состояния задачи содержит четыре селектора сегментов стека (для разных уровней привилегий), но в каждый момент используется только один стек.

2.7.6. Ассоциативный массив

Ассоциативный массив — абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- `INSERT(ключ, значение)`
- `FIND(ключ)`
- `REMOVE(ключ)`

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

В паре значение называется значением, ассоциированным с ключом . Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

Операция `FIND(ключ)` возвращает значение, ассоциированное с заданным ключом, или некоторый специальный объект UNDEF, означающий, что значения, ассоциированного с заданным ключом, нет. Две другие операции ничего не возвращают (за исключением,

возможно, информации о том, успешно ли была выполнена данная операция).

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов — например, строки.

Поддержка ассоциативных массивов есть во многих интерпретируемых языках программирования высокого уровня, таких, как Perl, PHP, Python, Ruby, Tcl, JavaScript и др. Для языков, которые не имеют встроенных средств работы с ассоциативными массивами, существует множество реализаций в виде библиотек.

Примеры

Примером ассоциативного массива является телефонный справочник. Значением в данном случае является совокупность «Ф. И. О. + адрес», а ключом — номер телефона. Один номер телефона имеет одного владельца, но один человек может иметь несколько номеров.

Расширения ассоциативного массива

Указанные три операции часто дополняются другими. Наиболее популярные расширения включают следующие операции:

- `CLEAR` — удалить все записи
- `EACH` — «пробежаться» по всем хранимым парам
- `MIN` — найти пару с минимальным значением ключа
- `MAX` — найти пару с максимальным значением ключа

В последних двух случаях необходимо, чтобы на ключах была определена операция сравнения.

Реализации ассоциативного массива

Существует множество различных реализаций ассоциативного массива.

Самая простая реализация может быть основана на обычном массиве, элементами которого являются пары (ключ, значение). Для ускорения операции поиска можно упорядочить элементы этого массива по ключу и осуществлять нахождение методом бинарного поиска. Но это увеличит время выполнения операции добавления новой пары, так как необходимо будет «раздвигать» элементы массива, чтобы в образовавшуюся пустую ячейку поместить новую запись.

Наиболее популярны реализации, основанные на различных деревьях поиска. Так, например, в стандартной библиотеке STL языка C++ контейнер `map` реализован на основе красно-чёрного дерева. В языках Ruby, Tcl, Python используется один из вариантов хэш-таблицы. Есть и другие реализации.

У каждой реализации есть свои достоинства и недостатки. Важно, чтобы все три операции выполнялись как в среднем, так и в худшем

случае за время , где — текущее количество хранимых пар. Для сбалансированных деревьев поиска (в том числе для красно-чёрных деревьев) это условие выполнено.

В реализациях, основанных на хэш-таблицах, среднее время

оценивается как , что лучше, чем в реализациях, основанных на деревьях поиска. Но при этом не гарантируется высокая скорость выполнения отдельной операции: время операции `INSERT` в худшем

случае оценивается как . Операция `INSERT` выполняется долго, когда коэффициент заполнения становится высоким и необходимо перестроить индекс хэш-таблицы.

Хэш-таблицы плохи также тем, что на их основе нельзя реализовать быстро работающие дополнительные операции MIN, MAX и алгоритм обхода всех хранимых пар в порядке возрастания или убывания ключей.

Поддержка ассоциативных массивов в языках программирования

Библиотека STL языка C++

Здесь приведено простейшее консольное приложение, предоставляющее интерфейс телефонной книжки. Оно реализовано на основе контейнера `map`.

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
  {
    string cmd, name, phone;
    map< string, string > book;

    while( cin >> cmd )
      {
        if ( cmd == "add" )
          {
            cin >> name >> phone;
            book[ name ] = phone;
            cout << "Added" << endl;
          }
        else if ( cmd == "find" )
          {
            cin >> name;
            try {
              string v = book.at( name );
              cout << name << "'s phone is " << v
<< endl;
            }
            catch (const out_of_range& e) {
              cout << name << " not found" << endl;
            }
          }
        else if ( cmd == "del" )
          {
            cin >> name;
            book.erase( name );
            cout << "Deleted" << endl;
```

```
        }
      else if ( cmd == "view" )
        {
          for( auto& kv : book )
            cout << kv.first << "\t " <<
kv.second << endl;
        }
      else if ( cmd == "quit" )
        return 0;
      else
        cerr << "Bad command '" << cmd << "'" <<
endl;
    }

    return 0;
  }
```

C#

В C# для организации ассоциативного массива используется тип Dictionary:

```
Dictionary<string, string> dic = new
Dictionary<string, string>();
dic.Add("Sally Smart", "555-9999");
dic.Add("John Doe", "555-1212");
dic.Add("J. Random Hacker", "553-1337");

// доступ к значению и показ сообщения
MessageBox.Show(dic["Sally Smart"]);
```

Для перебора элементов можно использовать цикл foreach. Порядок элементов не гарантируется. Если порядок важен, можно использовать SortedDictionary либо использовать метод-расширение Sort из LINQ.

```
// цикл по элементам с показом каждого элемента
foreach(KeyValuePair<string,string> kvp in dic)
{
    MessageBox.Show(String.Format("Phone number for
{0} is {1}", kvp.Key, kvp.Value));
}
```

Java

В языке Java ассоциативный массив именуется отображением (map) и имеет соответствующий интерфейс в стандартном Java API: java.util.Map Стандартный Java SDK включает в себя ряд реализаций этого интерфейса: HashMap, LinkedHashMap, ConcurrentHashMap, EnumMap, TreeMap и другие.

```
Map<String, String> map = new HashMap<String,
String>();
map.put("a","apricot");
map.put("b","banana");
map.put("c","cherry");
String s = map.get("b");
```

Перебор элементов коллекции:

```
  for (Map.Entry<String,String> pair :
map.entrySet()){
            System.out.printf("Ключ %s - Значение
%s" , pair.getKey() ,pair.getValue());
        }
```

Kotlin

В отличие от многих языков, в Kotlin существуют неизменяемые и изменяемые ассоциативные массивы: Map<K, out V> и MutableMap<K, V>. Это относится к любым коллекциям - спискам, множествам и т.д.. Создание ассоциативного массива осуществляется с помощью идиомы mapOf(a to b, c to d).

```
val customer = mapOf(
        "name" to "Dmitrii",
        "age" to 33,
        "languages" to listOf("russian", "english",
"japanese"),
        "address" to mapOf(
                "city" to "Khabarovsk",
                "street" to "Muraviev-Amurskiy",
                "zipCode" to "680000"
        )
```

```
)
```

Перебор элементов коллекции:

```
val map = mapOf("api.domain.io" to 9389,
"localhost" to 8080)
for ((host, port) in map){
    //...
}
```

Ruby

Класс Hash из стандартной библиотеки Ruby поддерживает операции `[]` (find), `[]=` (insert), `delete`, `each`, `keys`, `values`, а также множество других.

Ниже приведён код с примерами выполнения отдельных операций.

```
# телефонная книга
phone_book = {'Ivan'=>'+74951234567',
'Anna'=>'+74951112233'}
phone_book['Ivan'] # равно '+74951234567'
phone_book['Peter'] = '+74952223344'  # добавили
новую пару
phone_book.delete('Anna') # удалили пару ('Anna',
'+74951112233')

phone_book.each do |key, value| # выведем все
записи
 puts "%20s %10s" % [key, value]
end

puts phone_book.values # вывести все номера
телефонов
```

Ниже приведён код с реализацией консольного приложения «телефонная книжка».

```
require 'yaml'
book = {}
while line = STDIN.readline
  cmd, name, phone = line.split
  case cmd
  when 'insert'
    book[name] = phone
  when 'find'
    puts "#{name}'s phone is #{book[name]}"
  when 'del'
    book.delete(name)
  when 'view'
    book.each {|n,p| puts "#{n}\t #{p}" }
  when 'save'
    File.open(name,'w+'){|f| f.write(book.to_yaml)}
  when 'load'
    book = YAML.load_file(name)
  when 'quit'
    exit 0;
  else
    puts "Bad command '#{cmd}'";
  end
end
```

Python

Встроенный в Python тип ассоциативного массива называется словарём, элементами которого являются пары ключей и соответствующих им значений.

```
d1 = dict(a=10, b=20)
d2 = {'a': 10, 'b': 20}
d1[100] = 123
d2['c'] = 321
d1[100] = 1023
```

Здесь были показаны два способа написания литерала словаря и продемонстрировано, что ключом может быть объект любого неизменяемого (в нотации python) типа. Добавление нового объекта в словарь не требует предварительных проверок: если ранее ключу уже соответствовало некоторое значение, оно будет перезаписано

(Подробнее см. Python Tutorial, Dictionaries (англ.)). Другие операции со словарем:

```
if 'a' in d1:    # проверка наличия ключа
    del d1['a']        # удаление ключа (и
значения)
val = d1.get('a', 'default value')   # получение
значения по ключу или значения
                                     # по умолчанию
в случае отсутствия ключа
val = d1.setdefault('a', 'default value')   #
получение значения по ключу или значения
                       # по умолчанию в случае
отсутствия ключа (при этом
                       # значение записывается в
словарь)
d1.keys()              # список ключей
d1.values()            # список значений
d1.items()             # список пар ключ-значение
```

На Python весьма просто можно написать свой класс, который будет вести себя подобно словарю. Для этого необходимо лишь определить в своем классе соответствующие методы (см. Python Reference Manual, Emulating container types (англ.)).

Расширить свойства встроенного типа словаря (dict) можно путём наследования класса, см. пример.

Perl

Ассоциативный массив (в Perl принято называть его хешем — англ. *hash*[2]) является встроенным типом данных. Хеш можно создавать поэлементно либо целиком, присвоив ему значения списка, в котором элементы записаны в виде пар «ключ — значение», внутри пары элементы могут разделяться как традиционным путём (например, запятой), так и при помощи оператора =>:

```
# Поэлементное присваивание
$hash{'horse'} = 'colt';
$hash{'sheep'} = 'lamb';
```

```
# Вывод количества ключей хеша. Напечатает 2
print scalar keys %hash;

# Присваивание списка
# Значения для ключей 'horse' и 'sheep' будут
потеряны
%hash = (
    'cat' => 'kitten',
    'dog' => 'puppy',
    'cow' => 'calf',
);

print $hash{'cat'}; # Напечатает kitten
print keys %hash; # Вывод всех ключей. Напечатает
catdogcow
print values %hash; # Вывод всех значений.
Напечатает kittenpuppycalf
print %hash; # Напечатает catkittencowcalfdogpuppy
```

Delphi

Delphi до 2007 версии не имело прямых средств работы с ассоциативными массивами. Однако вы можете имитировать ассоциативные массивы, используя различного рода списковые классы для этого: TBucketList, TObjectBucketList, THashedStringList, TStringList (как и все другие потомки TStrings, а также Memo, ListBox и др.). Например:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i : Integer;
  s : string;
begin
  with TStringList.Create do //создание анонимного
списка (без объявления в var-секции)
    try
{1}   Values['Sally Smart'] := '555-9999'; // -
добавление новой записи
{2}   Values['John Doe'   ] := '555-1212'; // -
добавление ещё одной записи
{3}   Values['Sally Smart'] := '111-9999'; // -
замена значения у существующей записи
```

```
{4}   Values['John Doe'   ] := '';         // -
удаление записи со сдвигом списка

      SaveToFile(  'restore.txt'); // - сохранения
списка во внешний файл
      LoadFromFile('restore.txt'); // -
восстановление списка из файла

      s := Text;  // - сохранения списка в строке
(единым текстом с CR/LF-ами)
      Text := s;  // - восстановление списка из
строки (единым текстом с CR/LF-ами)

   // очистка и наполнение списка из специальной
строки с настраиваемыми разделителями
   // Delimiter, QuoteChar - тремя записями пар
[строковый ключ=значение ключа]
      DelimitedText :=
      '"Sally Smart=555-9999" "John Doe=555-1212"
"J. Random Hacker=553-1337"';

   // TStringList, как и все потомки TStrings,
наделены этими 4 свойствами (см. выше),
   // поэтому возможно их взаимодействие, в том
числе с врапперами над Ассоциативными
   // массивами, хранящимися в системных списках:
Memo-полях, ListBox'ах и т.д.:

      Assign(ListBox1.Items);  // - очистка и
восстановление списка из ListBox1
      Memo1.Lines.Values['J. Random Hacker'] :=
'553-1337'; // - в Memo-поле
      AddStrings(Memo1.Lines); // - добавление
строчек из списка Memo-поля

      Clear;                   // - просто очистка
списка

   // доступ к значению и вывод его в message box
      ShowMessage(Values['Sally Smart']);

   // проход по ассоциативному массиву и доступ к
ключам и значениям по индексу
```

```
      for i := 0 to Count - 1 do
        ShowMessage(Format('Number for %s:
%s',[Names[i],ValueFromIndex[i]]));
    finally
      Free // автоуничтожение списка -
Ассоциативного массива
    end;
end;
```

PL/SQL

СУБД Oracle начиная с версии 9.2.0 позволяет использовать в качестве ключей, помимо binary_integer и pls_integer, также и строки varchar2 с длиной до 32767:

```
declare
      type year_type is table of number index by
varchar2(4000);
      year_sales year_type;
      tot_sales   number;
begin
      year_sales('1990') := 34000;
      year_sales('1991') := 45000;
      year_sales('1992') := 43000;

      tot_sales := year_sales('1990') +
year_sales('1991') +
year_sales('1992');
      dbms_output.put_line('Total sales: ' ||
tot_sales);
end;
```

PHP

```
$hash = array(
    'cat' => 'kitten',
    'dog' => 'puppy',
    'cow' => 'calf'
);
print $hash['cat']; # Напечатает kitten
print_r( array_keys($hash) ); # Вывод всех ключей.
print_r( array_values($hash) ); # Вывод всех
значений.
```

```
print_r($hash); # Напечатает Array(cat=>'kitten',
dog=>'puppy', cow=>'calf');
```

PureBasic

В PureBasic, начиная с версии 4.40, появилась встроенная поддержка ассоциативных массивов. Его называют картой (*map*). Пример обычного ассоциативного массива:

```
; Создали ассоциативный массив.
NewMap Country.s()

; Заполнение массива данными.
Country("GE") = "Germany"
Country("FR") = "France"
Country("UK") = "United Kingdom"

Debug Country("FR") ; Вывод результата "FR" в
отладочное окно.

; Перебор всех элементов с отображением их значений
в отладочном окне.
ForEach Country()
  Debug Country()
Next
```

Пример структурированного (каждым элементом является структура данных) ассоциативного массива.

```
  Structure Car ; Описание структуры.
    Weight.l
    Speed.l
    Price.l
  EndStructure

  NewMap Cars.Car() ; Создание структурированного
ассоциативного массива.

  ; Заполнение массива данными.
  Cars("Ferrari F40")\Weight = 1000
  Cars()\Speed = 320
  Cars()\Price = 500000
```

```
  Cars("Lamborghini Gallardo")\Weight = 1200
  Cars()\Speed = 340
  Cars()\Price = 700000

  ; Перебор всех элементов с отображением их
значений в отладочном окне.
  ForEach Cars()
    Debug "Car name: "+MapKey(Cars()) ; Имя ключа
текущего элемента массива.
    Debug "Weight: "+Str(Cars()\Weight)
  Next
```

JavaScript

В ECMAScript 6 есть специальный объект Map, но он не везде поддерживается. Обычные массивы могут иметь только числовые индексы, потому для эмуляции ассоциативных массивов, ключами которых могут быть в том числе и строковые значения, можно использовать объекты.

Конструкция вида myVar = { key1: value1, key2: value2, … } создает объект myVar с набором полей, каждое из которых имеет свой ключ и значение. В дальнейшем доступ к элементам этого объекта может выполняться как с использованием нотации объектов и полей (myVar.key1), так и в нотации массивов и ключей (myvar['key1']).

```
var hash = {
   cat : 'kitten',
   'my-dog' : 'puppy', // если ключ содержит
символы, отличные от алфавитно-цифровых, он
заключается в кавычки
   cow : 'calf'
};

document.write(hash.cat);
document.write(hash['my-dog']);

hash.parrot = 'Kesha'; // добавление элемента в хэш
выполняется присваиванием нужного значения по (пока
еще) не существующему ключу
document.write(hash.parrot);
```

```
delete hash.parrot;  // для удаления элемента
используется оператор delete
delete hash['my-dog'];
hash['my-parrot'] = 'Kesha';

document.write(hash['my-parrot']);

for (var p in hash) document.write(p + ' name is '
+ hash[p]); // для обхода элементов хэша в
JavaScript есть специальный вид цикла for .. in
```

Go

В компилируемом языке программирования Go ассоциативные массивы названы картами (map) и их поддержка, в отличие от языков C и C++, встроена в язык, то есть не требует подключения каких-либо модулей (package).

```
package main

import "fmt"

func main() {
        elements := make(map[string]string)
        elements["H"] = "Hydrogen"
        elements["He"] = "Helium"
        fmt.Println(elements["He"])

        x := make(map[string]int)
        x["aaa"] = 10
        x["bbb"] = 20
        fmt.Println(x["aaa"])
}
```

2.7.7. Множество (тип данных)

Множество — тип и структура данных в информатике, которая является реализацией математического объекта множество.

Данные типа множество позволяют хранить ограниченное число значений определённого типа без определённого порядка. Повторение значений, как правило, недопустимо. За исключением того, что множество в программировании конечно, оно в общем соответствует концепции математического множества. Для этого типа в языках программирования обычно предусмотрены стандартные операции над множествами.

В зависимости от идеологии, разные языки программирования рассматривают множество как простой или сложный тип данных.

Реализации

Множество в Паскале

В языке Паскаль множество — составной тип данных, хранящий информацию о присутствии в множестве объектов любого счетного типа. Мощность этого типа определяет размер множества — 1 бит на элемент. В Turbo Pascal есть ограничение на 256 элементов, в некоторых других реализациях это ограничение ослаблено.

Пример работы с множествами:

```
type
  {определяем базовые для множеств перечислимый тип
и тип-диапазон}
  colors = (red,green,blue);
  smallnumbers = 0..10;
  {определяем множества из наших типов}
  colorset = set of colors;
  numberset = set of smallnumbers;
  {можно и не задавать тип отдельно}
  anothernumberset = set of 0..20;

{объявляем переменные типа множеств}
var
  nset1,nset2,nset3:numberset;
  cset:colorset;
begin
  nset1 := [0,2,4,6,8,10]; {задаем в виде
конструктора множества}
```

```
  cset  := [red,blue];      {простым перечислением
элементов}
  nset2 := [1,3,9,7,5];     {порядок перечисления
неважен}
  nset3 := [];              {пустое множество}
  include(nset3,7);         {добавление элемента}
  exclude(nset3,7);         {исключение элемента}
  nset1 := [0..5];          {возможно задавать
элементы диапазоном}
  nset3 := nset1 + nset2;   {объединение}
  nset3 := nset1 * nset2;   {пересечение}
  nset3 := nset1 - nset2;   {разность}
  if (5 in nset2) or        {проверка на вхождение
элемента}
    (green in cset) then
    {…}
end.
```

Множество в C++

Пример программы, использующей структуру set для реализации каталогов:

```
vector <string> vec;

void f(vector <string> vec1, int level) {
        set <string> sett;
        for (int i = 0; i < vec1.size(); i++) {
                int k = vec1[i].find('/');
                if (k != string::npos) {
                        string s1 =
vec1[i].substr(0, k);
                        sett.insert(s1);
                }
        }

        for (set <string> :: iterator it =
sett.begin(); it != sett.end(); it++) {
                vector <string> vec2;
                for (int i = 0; i < vec1.size();
i++) {
                        int k = vec1[i].find('/');
```

```
                        if (k != string::npos &&
vec1[i].substr(0, k) == (*it)) {
                                string s1 = vec1[i];
                                s1.erase(0, k + 1);
                                vec2.push_back(s1);
                        }
                }
                for (int i = 0; i < level; i++) {
                        cout << '+';
                }
                cout << *it << endl;
                f(vec2, level + 1);
        }
}


int main() {
        int n;
        cin >> n;

        for (int i = 0; i < n; i++) {
                string s;
                cin >> s;
                s += '/';
                vec.push_back(s);
        }

        f(vec, 0);
        return 0;
}
```

Множество в JavaScript

Множество в JavaScript - это структура данных, служащая для хранения набора значений, которые не имеют индексов или ключей (но внутри структуры данных они должны иметь порядок, например, индекс в массиве, однако, множество абстрагирует нас от этой особенности реализации).

Объект Set

Set – коллекция для хранения множества значений, причём каждое значение может встречаться лишь один раз.

Методы

set.has(item) - возвращает true если item есть в коллекции, иначе - false;

set.add(item) - добавляет элемент item в набор, возвращает set. Если пытаться добавить существующий, ничего не произойдет;

set.clear() - очищает set;

set.delete(item) - удаляет item из множества, возвращает true, если он там был, иначе false.

Перебор элементов

осуществляется через for..of либо forEach

```
'use strict';

let set = new Set(['first', 'second', 'third']);

for(let value of set) {
  console.log(value);
};
// first, second, third

// то же самое
set.forEach((value, valueAgain, set) => {
  console.log(value);
});
// first, second, third
```

Почему значение в аргументах повторяется два раза?

Так сделано для совместимости с Map, где у .forEach-функции также три аргумента.Но в Set первые два всегда совпадают и содержат очередное значение множества

Пример использования Set

```
const union = (s1, s2) => new Set([...s1, ...s2]);
// множество из всех значений s1 и s2

const intersection = (s1, s2) => new Set(
  [...s1].filter(v => s2.has(v))
);
// множество из значений которые есть и в s1 и в s2

const difference = (s1, s2) => new Set(
  [...s1].filter(v => !s2.has(v))
);
// множество из значений, которые есть в s1 но нет
в s2

const complement = (s1, s2) => difference(s2, s1);
// множество из значений коротые есть в s2 но нет в
s1

const cities1 = new Set(['Beijing', 'Kiev']);
const cities2 = new Set(['Kiev', 'London',
'Baghdad']);

const operations = [union, intersection,
difference, complement];

const results = operations.map(operation => ({
  [operation.name]: operation(cities1, cities2)
}));


console.dir({ cities1, cities2 });
console.dir(results);
```

3. Типы и полиморфизм

Термин «полиморфизм» означает способность кода выполняться над значениями множества разных типов, или возможность разных экземпляров одной и той же структуры данных содержать элементы разных типов. Некоторые системы типов допускают полиморфизм для потенциального повышения коэффициента повторного использования кода: в языках с полиморфизмом программистам достаточно реализовать структуры данных, такие как список или ассоциативный массив, лишь единожды, и не требуется разрабатывать по одной реализации для каждого типа элементов, которые планируется хранить в этих структурах. По этой причине некоторые учёные в области информатики иногда называют использование определённых форм полиморфизма «обобщённым программированием». Обоснования полиморфизма с точки зрения теории типов практически те же, что и для абстракции, модульности и в ряде случаев выделения подтипов данных.

В языках программирования и теории типов **полиморфизмом** называется способность функции обрабатывать данные разных типов.

Существует несколько разновидностей полиморфизма. Две наиболее различных из них были описаны Кристофером Стрэчи в 1967 году: это ad hoc полиморфизм и параметрический полиморфизм.

Параметрический полиморфизм подразумевает исполнение **одного и того же** кода для всех допустимых типов аргументов, тогда как ad hoc полиморфизм подразумевает исполнение потенциально **разного** кода для каждого типа или подтипа аргумента.

Бьерн Страуструп определил полиморфизм как «*один интерфейс — много реализаций*», но это определение не относится к параметрическому полиморфизму.

Полиморфизм является фундаментальным свойством системы типов. Различают статическую неполиморфную типизацию (потомки Алгола и BCPL), динамическую типизацию (потомки Lisp, Smalltalk, APL) и

статическую полиморфную типизацию (потомки ML). Использование ad hoc полиморфизма наиболее характерно для неполиморфной типизации. Параметрический полиморфизм и динамическая типизация намного существеннее, чем ad hoc полиморфизм, повышают коэффициент повторного использования кода, поскольку определенная единственный раз функция реализует без дублирования заданное поведение для бесконечного множества вновь определяемых типов, удовлетворяющих требуемым в функции условиям. С другой стороны, временами возникает необходимость обеспечить различное поведение функции в зависимости от типа параметра, и тогда необходимым оказывается специальный полиморфизм.

Параметрический полиморфизм является синонимом **абстракции типа**. Он повсеместно используется в функциональном программировании, где он обычно обозначается просто как «полиморфизм».

В сообществе объектно-ориентированного программирования под термином «полиморфизм» обычно подразумевают наследование, а использование параметрического полиморфизма называют обобщённым программированием, или иногда «статическим полиморфизмом».

3.1. Классификация

Впервые классификацию разновидностей полиморфизма осуществил Кристофер Стрэчи.

Если параметру функции сопоставлен ровно один тип, то такая функция называется **мономорфной**. Многие языки программирования предоставляют синтаксический механизм для назначения нескольким мономорфным функциям единого имени (идентификатора). В этом случае, в исходном коде становится возможным осуществлять вызов функции с фактическими параметрами разных типов, но в скомпилированном коде фактически происходит вызов *различных* функций (см. перегрузка процедур и функций). Стрэчи назвал такую возможность «*ad hoc полиморфизмом*».

Если параметру функции сопоставлено более одного типа, то такая функция называется **полиморфной**. Разумеется, с каждым фактическим значением может быть связан лишь один тип, но

полиморфная функция рассматривает параметры на основе внешних свойств, а не их собственной организации и содержания. Стрэчи назвал такую возможность «*параметрическим полиморфизмом*».

В дальнейшем классификацию уточнил Лука Карделли, выделив четыре разновидности полиморфизма:

- универсальный
  - параметрический
  - включения (или подтипов)
- ad hoc
  - перегрузка
  - приведение типов

Джон Митчел выделяет три разновидности полиморфизма как независимые: параметрический, ad hoc и подтипов.

Латинский фразеологизм «ad hoc» (буквально «*к случаю*») имеет двойственное значение:

1. спонтанный, непродуманный, сделанный «на коленке»
2. специальный, устроенный конкретно для данной цели или данного случая

Двойственность содержания термина «*ad hoc полиморфизм*» долгие годы была заслуженной. Стрэчи выбрал этот термин, руководствуясь первым значением — в работе он подчеркивает, что при ad hoc полиморфизме нет единого систематичного способа вывести тип результата из типа аргументов, и хотя возможно построение определённого набора правил для сужения спектра его поиска, но эти правила по своей природе являются спонтанными как по содержанию, так и по контексту применения.

Действительно, ad hoc полиморфизм не является *истинным* полиморфизмом. Перегрузка функций даёт не «***значение**, имеющее множество типов*», а «***символ**, имеющий множество типов*», но значения, идентифицируемые этим символом, имеют *разные* (потенциально не совместимые) типы. Аналогично, приведение типов не является истинным полиморфизмом: кажется, будто оператор принимает значения множества типов, но значения должны быть

преобразованы к некоторому представлению до того, как он сможет их использовать, так что на самом деле оператор работает лишь над одним типом (то есть имеет один тип). Более того, тип возвращаемого значения здесь не зависит от типа входного параметра, как в случае параметрического полиморфизма.

Тем не менее, определение специальных реализаций функций для разных типов в некоторых случаях является *необходимостью*, а не случайностью. Классическими примерами служат реализация арифметических операций (физически различная для целых и чисел с плавающей запятой) и равенства типов, которые на протяжении десятилетий не имели общепринятой универсальной формализации. Решением стали классы типов, представляющие собой механизм явного дискретного перечисления допустимых значений переменных типа для статической диспетчеризации в слое типов. Они сводят воедино две разновидности полиморфизма, разделённые Стречи, «*делая ad hoc полиморфизм менее ad hoc*» (игра на двойственности смысла). Дальнейшее обобщение классов типов привело к построению теории квалифицированных типов, единообразно формализующей самые экзотичные системы типов, включая расширяемые записи и подтипы.

В отличие от перегрузки и приведения типов, полиморфизм подтипов является *истинным* полиморфизмом: объектами подтипа можно манипулировать единообразно, как если бы они принадлежали к своим супертипам (но сказанное не верно для языков, реализующих «*полиморфизм при наследовании*» посредством приведения типов и перегрузки функций, как в случае C++). Наиболее *чистым* является параметрический полиморфизм: один и тот же объект или функция может единообразно использоваться в разных контекстах типизации без изменений, приведений типов или любых других проверок времени исполнения или преобразований. Однако, для этого требуется некое единообразное представление данных (например, посредством указателей).

3.2. Параметрический полиморфизм

Параметрический полиморфизм в языках программирования и теории типов представляет собой свойство семантики системы типов, позволяющее обрабатывать значения разных типов *идентичным*

образом, то есть исполнять физически *один и тот же* код для данных разных типов.

Параметрический полиморфизм является **истинной** формой полиморфизма, делая язык более выразительным и существенно повышая коэффициент повторного использования кода. Традиционно ему противопоставляется ad hoc полиморфизм, предоставляющий единый интерфейс к потенциально *различному* коду для разных допустимых в данном контексте типов, потенциально не совместимых (статически или динамически). В ряде исчислений, например, в теории квалифицированных типов, ad hoc полиморфизм рассматривается как частный случай параметрического.

Параметрический полиморфизм лежит в основе систем типов языков семейства ML; такие системы типов называют полиморфными. В сообществах языков с неполиморфными системами типов (потомки Алгола и BCPL) параметрически полиморфные функции и типы называют «обобщёнными».

3.3. Полиморфизм типов

Термин «*параметрический полиморфизм*» традиционно используется для обозначения типобезопасного параметрического полиморфизма, хотя существуют и нетипизированные его формы (см. параметрический полиморфизм в Си и C++). Ключевым понятием типобезопасного параметрического полиморфизма, помимо полиморфной функции, является полиморфный тип.

Полиморфный тип (англ. *polymorphic type*), или политип (англ. *polytype*) — это тип, параметризованный другим типом. Параметр в слое типов называется переменной типа (или ти́повой переменной).

Формально полиморфизм типов изучается в полиморфно типизированном лямбда-исчислении, называемом Системой F.

Например, функция `append`, сцепляющая два списка в один, может быть построена независимо от типа элементов списка. Пусть ти́повая переменная `a` описывает тип элементов списка. Тогда функция `append` может быть типизирована как «`forall a. [a] × [a] ->`

`[a]`» (здесь конструкция `[a]` означает тип «*список, каждый элемент которого имеет тип* `a`» — синтаксис, принятый в языке Haskell). В этом случае говорят, что тип параметризован переменной `a` для всех значений `a`. В каждом месте применения `append` к конкретным аргументам значение `a` разрешается, причём **каждое** её упоминание в сигнатуре типа подменяется значением, соответствующим контексту применения. Таким образом, в данном случае сигнатура функционального типа требует *идентичности* типов элементов обоих списков и результата.

Множество допустимых значений переменной типа задаётся посредством ***квантификации***. Простейшими кванторами являются универсальный (как в примере с `append`) и экзистенциальный (см. далее).

Квалифицированный тип (англ. *qualified type*) — это полиморфный тип, дополнительно снабжённый набором предикатов, регламентирующих спектр допустимых значений параметра этого типа. Иначе говоря, *квалификация* типа позволяет управлять квантификацией произвольным образом. Теорию квалифицированных типов построил Марк Джонс (*Mark P. Jones*) в 1992 году. Она предоставляет общее обоснование для самых экзотичных систем типов, включая классы типов, расширяемые записи и подтипы и позволяет точно формулировать особые ограничения для конкретных полиморфных типов, устанавливая таким образом отношения между параметрическим и ad hoc полиморфизмом (перегрузкой), а также между явной и неявной перегрузкой. Связь типа с предикатом в этой теории называется *свидетельством* (англ. *evidence*). Для свидетельств сформулирована алгебра, аналогичная лямбда-исчислению, включающая абстракцию свидетельств, применение свидетельств и т. д. Соотнесение терма этой алгебры с явно перегруженной функцией называется *трансляцией свидетельства* (англ. *evidence translation*).

Мотивирующими примерами для разработки обобщённой теории послужили классы типов Вадлера — Блотта и типизация расширяемых записей посредством предикатов Харпера — Пирса.

3.4. Классификация полиморфных систем

Имеется викиучебник по теме
«Параметрический полиморфизм»

Параметрически полиморфные системы типов принципиально классифицируются по **рангу** и по свойству **предикативности**. Кроме того, различаются явный и неявный полиморфизм и ряд других свойств. Неявный полиморфизм обеспечивается за счёт выведения типов, что существенно повышает удобство использования, но имеет ограниченную выразительность. Многие практические параметрически полиморфные языки разделяют фазы *внешнего* неявно типизированного языка (англ. *external implicitly typed language*) и *внутреннего* явно типизированного (англ. *internal explicitly typed language*).

Наиболее общей формой полиморфизма является «*импредикативный полиморфизм высших рангов*». Наиболее популярными ограничениями этой формы являются полиморфизм 1-го ранга, называемый «*пренексным*», и предикативный полиморфизм. Вместе они образуют «*предикативный пренексный полиморфизм*», близкий к реализованному в ML и в ранних версиях Хаскела.

С усложнением систем типов сигнатуры типов становятся *настолько* сложными, что полное или почти полное их выведение начинает рассматриваться многими исследователями как критичное свойство, отсутствие которого сделает язык непригодным для практики. Например, для традиционного комбинатора `map` полная сигнатура типа (с учётом родовой квантификации) в условиях типобезопасного отслеживания потока исключений принимает следующий вид (как и выше, `[a]` означает список элементов типа `a`):

$$val\ map : \forall \alpha : *. \forall \beta : *. \forall \gamma : \varnothing. \forall \delta : \varnothing. (\alpha \xrightarrow{\gamma} \beta) \xrightarrow{\delta} ([\alpha] \xrightarrow{\gamma} [\delta])$$

Ранг

Ранг полиморфизма показывает *допустимую* в рамках системы глубину вложения кванторов переменных типа. «*Полиморфизм ранга k*» (при $k > 1$) позволяет конкретизировать переменные типа полиморфными типами ранга не выше (k — 1). «*Полиморфизм высших рангов*» позволяет ставить кванторы переменных типа слева от *произвольного* числа стрелок в типах высших порядков.

Джо Уэллс (англ. *Joe Wells*) доказал, что выведение типов для Системы F, типизированной по Карри, неразрешимо для рангов выше 2-го, так что при использовании более высоких рангов необходимо использовать явное аннотирование типами.

Существуют системы типов с частичным выведением, требующие аннотирования только невыводимых ти́повых переменных.

Пренексный полиморфизм

Полиморфизм ранга 1 часто называется ***пренексным*** (от слова «пренекс» — см. пренексная нормальная форма). В пренексно полиморфной системе переменные типа не могут конкретизироваться полиморфными типами. Это ограничение делает различие между мономорфными и полиморфными типами существенным, из-за чего в пренексной системе полиморфные типы нередко называют «*схемами типизации*» (англ. *type schemas*) для отличения их от «обычных» (мономорфных) типов (монотипов). Как следствие, все типы могут быть записаны в форме, когда все кванторы переменных типа вынесены в самую внешнюю (пренексную) позицию, что и называется пренексной нормальной формой. Проще говоря, разрешается полиморфное определение функций, но запрещается передавать полиморфные функции в качестве аргументов другим функциям — полиморфно определённые функции должны быть инстанцированы монотипом перед использованием.

Близким эквивалентом является так называемый «***Let-полиморфизм***» или «***полиморфизм в стиле ML***» Дамаса — Милнера. Технически, Let-полиморфизм в ML имеет дополнительные синтаксические ограничения, такие как «ограничение на значения» (*value restriction*),

связанное с проблемой типобезопасности при использовании ссылок (не возникающих в чистых языках, таких как Haskell и Clean).

3.5. Предикативность

Предикативный полиморфизм

Предикативный (ограниченный условием) полиморфизм требует, чтобы переменная типа была конкретизирована монотипом (не политипом).

К предикативным системам относятся интуиционистская теория типов и Nuprl.

Импредикативный полиморфизм

Импредикативный (безусловный) полиморфизм разрешает конкретизировать переменную типа *произвольным* типом — как мономорфным, так и полиморфным, включая сам определяемый тип. (Разрешение в рамках некоего исчисления рекурсивного включения системы в саму себя называется импредикативностью. Потенциально это может приводить к парадоксам типа Расселовского или Канторовского, но в случае с тщательно продуманной системой типов этого не происходит.)

Импредикативный полиморфизм позволяет передавать полиморфные функции другим функциям в качестве параметров, возвращать их в качестве результата, хранить их в структурах данных и т. д., поэтому его также называют **полиморфизмом первого класса**. Это наиболее мощная форма полиморфизма, но, с другой стороны, представляющая серьёзную проблему для оптимизации и делающая выведение типов неразрешимым.

Примером импредикативной системы является Система F и её расширения (см. лямбда-куб).

Поддержка рекурсии

Основная статья: **Полиморфная рекурсия (англ.)**

Традиционно в потомках ML функция может быть полиморфной только при взгляде «извне» (то есть её можно применять к аргументам различных типов), но её определение может содержать только мономорфную рекурсию (то есть разрешение типов осуществляется до вызова). Распространение реконструкции типов по ML на *рекурсивные типы* не представляет серьезных трудностей. С другой стороны, сочетание реконструкции типов с *рекурсивно определенными термами* порождает сложную проблему, известную под названием ***полиморфной рекурсии***. Майкрофт (*Mycroft*) и Мейртенс (*Meertens*) предложили полиморфное правило типизации, позволяющее конкретизировать различными типами рекурсивные вызовы рекурсивной функции из ее собственного тела. В этом исчислении, известном как исчисление Милнера — Майкрофта, выведение типов неразрешимо.

3.6. Полиморфизм структурных типов

Типы-произведения (также известные как «записи») служат формальной базой для объектно-ориентированного и модульного программирования. Их двойственную пару составляют типы-суммы (также известные как «варианты»):

$$\neg\{a \wedge b\} = \langle \neg a \vee \neg b \rangle$$
$$\neg\langle a \vee b \rangle = \{\neg a \wedge \vee b\}$$

Вместе они являются средством выражения любых сложных структур данных и некоторых аспектов поведения программ.

Исчисление записей (англ. *record calculi*) — обобщённое название проблемы и ряда её решений, касающихся вопросов гибкости типов-произведений в языках программирования при условии типобезопасности. Термин нередко распространяется и на типы-суммы, а границы понятия «тип записи» могут варьироваться от исчисления к исчислению (как и само понятие «тип»). Применяются также термины «**полиморфизм записей**» (что, опять же, зачастую включает в себя полиморфизм вариантов), «**исчисление модулей**» и «**структурный полиморфизм**».

Произведения и суммы типов (записи и варианты) обеспечивают гибкость при построении сложных структур данных, но ограничения многих реальных систем типов зачастую не позволяют использовать их по-настоящему гибко. Эти ограничения обычно возникают в связи с вопросами эффективности, выведения типов или просто удобства использования.

Классическим примером может служить язык Standard ML, система типов которого была умышленно ограничена ровно настолько, чтобы сочетать простоту реализуемости с выразительностью и математически доказуемой типобезопасностью.

Пример определения записи:

```
> val r = {name = "Foo", used = true};
(* val r : {name : string, used : bool} = {name =
"Foo", used = true} *)
```

Доступ к значению поля по его имени осуществляется префиксной конструкцией вида `#field record`:

```
> val r1 = #name r;
(* val r1 : string = "Foo" *)
```

Следующее определение функции над записью является корректным:

```
> fun name1 (x: {name : string, age : int}) = #name
x
```

А следующее порождает ошибку компилятора о том, что тип не разрешён полностью:

```
> fun name2 x = #name x
(* unresolved type in declaration:
      {name : '1, ...} *)
```

Мономорфизм записей делает их негибким, но эффективным средством: поскольку фактическое расположение в памяти каждого поля записи известно заранее (на этапе компиляции), обращение к нему по имени компилируется в прямую индексацию, что обычно

вычисляется за одну машинную инструкцию. Однако, при разработке сложных масштабируемых систем желательно иметь возможность абстрагировать поля от конкретных записей — например, определить универсальный селектор полей:

```
fun select r l = #l r
```

Но компиляция полиморфного обращения к полям, которые могут располагаться в разном порядке в разных записях, представляет сложную проблему, как с точки зрения контроля безопасности операций на уровне языка, так и с точки зрения быстродействия на уровне машинного кода. Наивным решением может быть динамический поиск по словарю при каждом обращении (и скриптовые языки его применяют), однако, очевидно, что это чрезвычайно неэффективно.

Суммы типов составляют основу **выражения ветвления**, причём за счёт строгости системы типов компилятор обеспечивает контроль за полнотой разбора. Например, для следующего типа-суммы:

```
datatype 'a foo = A of 'a
                | B of ('a * 'a)
                | C
```

всякая функция над ним будет иметь вид

```
fun bar (p:'a foo) =
   case p of
        A x => ...
      | B (x,y) => ...
      | C => ...
```

и при удалении любого из предложений компилятор выдаст предупреждение о неполноте разбора («`match nonexhaustive`»). Для случаев, когда из множества вариантов лишь некоторые требуют анализа в данном контексте, можно организовать `default`-ветвление при помощи т. н. «джокера» — универсального образца, с которым сопоставимы все допустимые (согласно типизации) значения. Для его записи используется символ подчёркивания («_»). Например:

```
fun bar (p:'a foo) =
   case p of
        C => ...
      | _ => ...
```

Следует отметить, что в некоторых языках (в Standard ML, в Haskell) даже «более простая» конструкция `if-then-else` является лишь синтаксическим сахаром над частным случаем ветвления, то есть выражение

```
if A
   then B
   else C
```

уже на этапе грамматического разбора представляется в виде

```
case A of
     true => B
   | false => C
```

либо

```
case A of
     true => B
   | _ => C
```

Синтаксический сахар

Синтаксический сахар (англ. *syntactic sugar*) в языке программирования — это синтаксические возможности, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека.

Это может быть любой элемент синтаксиса, который даёт программисту альтернативный способ записи уже имеющейся в языке синтаксической конструкции, и при этом является более удобным, или более кратким, или похожим на другой распространённый способ записи, или помогает писать программы в хорошем стиле.

Определение

Под «синтаксическим сахаром» понимается любой имеющийся в языке программирования синтаксический элемент, механизм, способ описания, который дублирует другой, имеющийся в языке элемент или механизм, но является более удобным в использовании, или более краток, или выглядит естественнее, или более привычен (похож на аналогичные элементы других языков), или просто лучше воспринимается при чтении программы человеком. Принципиально то, что синтаксический сахар, теоретически, всегда можно удалить из языка без потери его возможностей — всё, что можно написать с применением синтаксического сахара, может быть написано на этом же языке и без него. Таким образом, синтаксический сахар предназначен лишь для того, чтобы сделать более удобным для программиста написание программы.

Понятие синтаксического сахара во многом условно. Его использование предполагает, что из всего множества синтаксических конструкций, имеющихся в языке, можно выделить некоторый «базовый набор», обеспечивающий функциональность языка, и дополнительные синтаксические средства, которые при желании можно выразить с помощью базового набора; последние и будут для данного языка синтаксическим сахаром. Однако многие конструкции являются взаимозаменяемыми, и далеко не всегда можно определённо сказать, какие именно из них являются базовыми, а какие — дополнительными. Например, в языке Модула-2 есть четыре вида циклов: цикл с предусловием, цикл с постусловием, цикл с шагом и безусловный цикл. Теоретически, первые три вида циклов могут быть легко выражены через последний. Являются ли они, в таком случае, синтаксическим сахаром? Обычно так не говорят, хотя формально под вышеприведённое определение они попадают.

Отнесение некоторых конструкций к синтаксическому сахару неоднозначно в силу исторических причин. Например, в языке C и его потомках имеются операторы увеличения и уменьшения (++, --, +=, -= и т. п.). Введение этих операторов в язык было вызвано необходимостью поддержки ручной оптимизации программ, так как код с их использованием мог транслироваться в более эффективные машинные команды («++a» транслировалось в одну команду INC, а аналогичное выражение «a=a+1» — в целую группу команд). Развитие технологии оптимизации кода сделало подобную ручную оптимизацию

бессмысленной; сейчас компиляторы порождают одинаковый код для «длинного» и «короткого» варианта операции. В результате сокращённые операторы превратились в синтаксический сахар, хотя изначально им не были.

«Синтаксический сахар» или «Лексический мусор»?

В некоторых случаях понятие «синтаксический сахар» трактуют более широко, нежели «другой способ записи для уже имеющихся синтаксических конструкций». Джек Креншоу в книге «Давайте создадим компилятор!» применяет этот термин к синтаксическим элементам, которые не нужны для правильной компиляции программы, а включаются в язык исключительно для обеспечения удобства программиста и для удобочитаемости программы:

В конце концов, люди тоже должны читать программы… Сахарные токены служат в качестве полезных ориентиров, помогающих вам не сбиться с пути…

В качестве примера такого синтаксического сахара приводится «then» в операторе «if» или «do» в операторе «while», а также точка с запятой: компилятор и без них однозначно определяет конец условия и место завершения оператора, но наличие этих конструкций делает программу более удобочитаемой. Очевидно, узкая трактовка понятия «синтаксический сахар» несовместима с широкой: в Си или Паскале невозможно записать операторы иным способом, без «then», «do» и точки с запятой. В таком случае уместно говорить о «**синтаксическом мусоре**». Учитывая же, что лишние в языке программирования слова — это лишние лексемы, то правильнее было бы применять термин «**лексический мусор**». С другой стороны, называть такие «лишние» элементы языка «мусором» не вполне корректно, ведь в действительности они могут существенно влиять на качество программирования, поскольку наличие избыточности в синтаксисе упрощает компилятору локализацию ошибок в коде. Рассмотрим пример на некоем условном Бейсик-подобном языке, где необязательно слово then в условном операторе и точка с запятой между операторами, а знак равенства может обозначать, в зависимости от положения, как логическое равенство, так и присваивание:

```
if a > b and k = 20 f = 10
```

Здесь «a>b and k=20» — это условие, а «f=10» — ветвь «то». Однако если программист пропустит или случайно удалит оператор «and», конструкция превратится в:

```
if a > b k = 20 f = 10
```

Программа останется синтаксически корректной, но условием будет уже просто «a>b», а ветвью «то», в зависимости от правил языка, либо «k=20», превратившееся из условия в присваивание, либо оба оператора «k=20 f=10». В результате ошибки будет нарушено условие и произойдёт разрушение значения переменной k. Так как при внесении логической ошибки программа останется синтаксически верной, компилятор ошибки не заметит. Требование обязательного наличия служебного слова «then» между условием и оператором приведёт к тому, что компилятор обнаружит синтаксическую ошибку в условии. Обязательность точки с запятой между операторами также позволит компилятору обнаружить ошибку — отсутствие точки с запятой после оператора «k=20». Таким образом, наличие «сахарных» токенов, как и вообще любая избыточность в языке, приводит к тому, что логические ошибки в коде превращаются в синтаксические и могут быть обнаружены компилятором.

Термин «синтаксический сахар» (англ. *syntactic sugar*) был введён Питером Лэндином (Peter J. Landin) в 1964 году для описания поверхностного синтаксиса простого алголоподобного языка, семантически определяемого в терминах аппликативных выражений лямбда-исчисления с последующей чисто лексической заменой λ на `where`.

Примеры

Массивы в Си

Массивы в Си представляют собой блоки в памяти. Доступ к элементам массива производится через указатель на начало блока памяти (то есть, на начало массива) и смещение элемента относительно начального адреса. Это может быть записано без использования специального синтаксиса для массивов (`a` — указатель на начало массива, `i` — индекс элемента): `*(a + i)`, но язык предоставляет специальный синтаксис: `a[i]`. Интересно, что можно использовать и

форму `i[a]`, совершенно логичную вследствие коммутативности операции сложения, но практически нечитаемую.

Переопределение операторов

К синтаксическому сахару можно отнести и переопределение операторов, поддерживаемое многими языками программирования. В принципе, любая операция может быть оформлена как процедура (функция, метод). Переопределение операторов позволяет выполнять операции, созданные программистом, внешне так же, как и встроенные в язык.

Свойства

Ещё одним примером синтаксического сахара является концепция «свойств», поддерживаемая многими современными языками программирования. Имеется в виду объявление в классе псевдополей, которые внешне ведут себя как поля класса (имеют имя, тип, допускают присваивание и чтение), но в действительности таковыми не являются. Каждое обращение к свойству преобразуется компилятором в вызов метода доступа. Свойства совершенно не являются необходимыми (методы доступа можно вызывать и непосредственно) и используются исключительно для удобства, поскольку код с использованием свойств выглядит несколько проще и понятнее.

Критика

Не все программисты считают наличие синтаксического сахара в языках программирования и использование его программистами благом. Известна точка зрения Никлауса Вирта, которую разделяет часть программистского сообщества: согласно ей, любое расширение языка, не вызванное необходимостью, ухудшает его, так как приводит к усложнению транслятора и, соответственно, к понижению его надёжности и производительности. Одновременно возрастает сложность изучения языка и сложность сопровождения программ. Кроме того, сам факт наличия дополнительных синтаксических средств часто играет провоцирующую роль: он побуждает программиста прибегать к различным синтаксическим трюкам вместо того, чтобы глубже анализировать задачу и реализовывать более

эффективные алгоритмы. Эти взгляды отразились в языках семейства Оберон, очень простых и практически лишённых синтаксического сахара.

Известен афоризм Алана Перлиса: *«Синтаксический сахар вызывает рак точек с запятой»*. Точка с запятой (;), являясь обязательной частью большинства популярных языков программирования, даже если в новом языке бесполезна, оставляется как необязательный элемент, так как большинство программистов имеют прочную привычку её использования. В оригинале афоризм обыгрывает созвучие английских слов *semicolon* («точка с запятой») и *colon*, последнее из которых означает не только двоеточие, но и толстый кишечник (*colon cancer* — «рак толстого кишечника»).

Чаще критика направляется на отдельные, часто встречающиеся виды синтаксического сахара: переопределение операций, свойства, сложные операции (вроде тернарной операции `?:` в Си). Доводы критиков, в основном, сводятся к тому, что подобные средства, в действительности, не делают программу ни проще, ни понятнее, ни эффективнее, ни короче, но приводят к дополнительной трате ресурсов и усложняют восприятие, а значит и сопровождение программы.

Синтаксическая соль

В противоположность «синтаксическому сахару» в понятие «**синтаксическая соль**» (англ. *syntactic salt*) на жаргоне хакеров обозначает дополнительные технически бесполезные конструкции в языке программирования, которые правила языка требуют употреблять при выполнении потенциально небезопасных действий. Они вводятся в язык только для того, чтобы, используя их, программист тем самым подтверждал, что сомнительное действие предпринято им сознательно, а не является случайной ошибкой или результатом непонимания. Так же, как «синтаксический сахар» не добавляет языку выразительности, «синтаксическая соль» не расширяет возможности языка и не нужна транслятору для корректной компиляции программы; она предназначена исключительно для людей, пользующихся данным языком. Классическим примером общеизвестной и широко применяемой «синтаксической соли» являются имеющиеся почти в любом языке со статической типизацией встроенные команды преобразования типов данных. Формально эти команды излишни (что

доказывает классический язык Си, где любое преобразование типов допустимо и выполняется автоматически), но в языках, где их применение обязательно, программист вынужден каждый раз обращать внимание на то, что он выполняет потенциально опасное смешение типов, которое часто указывает на логическую ошибку в программе. В зависимости от строгости языка программирования использование «синтаксической соли» может быть обязательным или факультативным. В первом случае транслятор воспринимает её отсутствие как синтаксическую ошибку, во втором — выдаёт при трансляции предупреждение, которое программист может проигнорировать. В отличие от «синтаксического сахара», который расширяет свободу выражения программиста, «синтаксическая соль» её сужает, требуя «без причины» писать длинные конструкции.

В Jargon File написано: «синтаксическая соль вредна, поскольку повышает артериальное давление хакера». Действительно, при написании небольших программ, создаваемых и поддерживаемых одним человеком, предосторожности могут показаться излишними, однако при промышленной разработке крупных программных комплексов, поддерживаемых большими коллективами программистов, зачастую, к тому же, не самой высокой квалификации, «синтаксическая соль» помогает не ошибаться в разработке и эффективнее разбираться в коде, написанном другими разработчиками.

Примеры:

- Директива `override` в Delphi явно указывает на то, что помеченный ею метод подменяет виртуальный метод класса-родителя. Наличие этой директивы требует от компилятора проверить соответствие сигнатуры подменяющего и подменяемого метода, так что при изменении в базовом классе программист будет вынужден внести те же изменения в классы-потомки, иначе программа не будет компилироваться.
- Операция `reinterpret_cast` в C++ обеспечивает небезопасное преобразование типа. Операция не производит никакого кода, она лишь позволяет программисту обойти контроль типов. Единственный смысл её использования — прямое указание на то, что небезопасное преобразование типов использовано намеренно.

В Standard ML записи и варианты являются мономорфными, однако, поддерживается синтаксический сахар для работы с записями со множеством полей, называемый «*универсальным образцом*»:

```
> val r = {name = "Foo", used = true};
(* val r : {name : string, used : bool} = {name =
"Foo", used = true} *)
> val {used = u, ...} = r;
(* val u : bool = true *)
```

Многоточие в типе «`{used, ...}`» означает, что в данной записи существуют и другие поля, помимо упомянутых. Однако полиморфизм записей как таковой отсутствует (даже пренексный): требуется полное статическое разрешение информации о мономорфном типе записи (посредством выведения или явного аннотирования).

3.7. Расширение и функциональное обновление записей

Термин **расширяемые записи** (*extensible records*) используется для обобщённого обозначения таких операций, как *расширение* (построение новой записи на основе имеющейся с добавлением новых полей), *обрезание* (взятие указанной части от имеющейся записи) и др. В частности, он подразумевает возможность так называемого «**функционального обновления записей**» (*functional record update*) — операции построения нового значения записи на основе имеющегося путём копирования имён и типов его полей, при которой одно или несколько полей в новой записи получают новые значения, отличающиеся от исходных (и, возможно, имеющие другой тип).

Сами по себе операции функционального обновления и расширения ортогональны полиморфизму записей, но их полиморфное использование представляет особый интерес. Даже для мономорфных записей приобретает большое значение возможность опускать явное упоминание полей, копируемых без изменений, а это фактически представляет собой полиморфизм записей в чисто синтаксической форме. С другой стороны, если рассматривать все записи в системе как

расширяемые, то это позволяет типизировать функции над записями как полиморфные.

Пример простейшего варианта конструкции реализован в Alice ML (согласно действующим соглашениям по successor ML). Универсальный образец (многоточие) имеет расширенные возможности: посредством его можно осуществлять «захват ряда» с тем, чтобы работать с «оставшейся» частью записи как со значением:

```
> val r = {a = 1, b = true, c = "hello"}
(* r = {a = 1, b = true, c = "hello"} *)
> val {a = n, ... = r1} = r
(* r1 = {b=true, c="hello"} *)
> val r2 = {d = 3.14, ... = r1}
(* r2 = {b=true, c="hello", d=3.14} *)
```

Функциональное обновление реализуется как производная форма от захвата ряда с помощью служебного слова `where`. Например, следующий код:

```
> let
     val r = { a = 1, c = 3.0, d = "not a list", f
= [1], p = ["not a string"], z = NONE }
  in
     { r where d = nil, p = "hello" }
  end
```

будет автоматически переписан в форме:

```
> let
     val r = { a = 1, c = 3.0, d = "not a list", f
= [1], p = ["not a string"], z = NONE }
     val { d = _, p = _, ... = tmp } = r
  in
     { ... = tmp, d = nil, p = "hello" }
  end
```

Конкатенация записей

Одними из первых (конец 1980-х — начало 1990-х) были предложены различные варианты формализации расширяемых записей через операции конкатенации над неполиморфными записями (Харпер — Пирс, Ванд, Сальцманн). Карделли даже использовал операции над записями *вместо* полиморфизма в языке Amber. Для этих исчислений нет известного способа эффективной компиляции; кроме того, эти исчисления весьма сложны и с точки зрения теоретико-ти́пового анализа. Например:

```
val car = { name = "Toyota"; age = "old"; id = 6678
}
val truck = { name = "Toyota"; id = 19823235 }
...
val repaired_truck = { car and truck }
```

Ванд (автор рядного полиморфизма) показал, что посредством конкатенации записей можно моделировать множественное наследование.

Структурная подтипизация Карделли

Лука Карделли (Luca Cardelli (англ.)) исследовал возможность формализовать «полиморфизм записей» посредством отношений ***подтипизации*** на записях: для этого запись рассматривается как «универсальный подтип», то есть разрешается отнесение её значения ко всему множеству её супертипов. Этот подход также поддерживает наследование методов и служит теоретико-ти́повой базой для наиболее распространённых форм объектно-ориентированного программирования.

Карделли представил вариант метода компиляции полиморфизма записей через их подтипы посредством предопределения смещения всех возможных меток в потенциально огромной структуре со множеством пустых слотов.

Метод имеет существенные недостатки. Поддержка подтипизации в системе типов существенно усложняет механизм проверки согласования типов. Кроме того, в её присутствии статический тип

выражения перестаёт предоставлять полную информацию о динамической структуре значения записи. Например, при использовании только подтипов, следующий терм:

```
> if true then {A = 1, B = true} else {B = false, C
= "Cat"}
(* val it : {B : bool} *)
```

имеет тип `{B : bool}`, но его динамическое значение равно `{A = 1, B = true}`, то есть информация о типе расширяемой записи теряется, что представляет серьёзную проблему для проверки операций, требующих для своего исполнения полной информации о структуре значения (таких как сравнение на равенство). Наконец, при наличии подтипов выбор между упорядоченным и неупорядоченным представлением записей серьёзно влияет на производительность.

Популярность подтипизации обусловлена тем, что она предоставляет простые и наглядные решения для многих задач. Например, объекты разных типов можно помещать в единый список, если они имеют общий супертип.

3.8. Рядный полиморфизм Ванда

Митчел Ванд (Mitchell Wand (англ.)) в 1987 году предложил идею захватывать информацию об «оставшейся» (не указанной явно) части записи посредством того, что он назвал рядной ти́повой переменной (*row type variable*).

Рядная переменная — это переменная типа, пробегающая множество конечных наборов (рядов) типизированных полей (пар «`(значение : тип)`»). В результате появляется возможность реализовать «наследование вширину» непосредственно на основе параметрического полиморфизма, лежащего в основе ML,— без усложнения системы типов правилами подтипизации. Получаемую разновидность полиморфизма называют *рядным полиморфизмом* (*row polymorphism*). Рядный полиморфизм распространяется как на произведения типов, так и на суммы типов.

***Замечание** — Ванд заимствовал термин «row» (рус. ряд, цепочка, строка) из Алгола-68, где он означал набор представлений. В*

русскоязычной литературе этот термин в контексте Алгола традиционно переводился как «мультивид». Встречается также вариант перевода «row variables» как «строчные переменные», который может вызвать путаницу со строчными буквами в строковых типах.

Пример (язык OCaml; синтаксис постфиксный, `record#field`):

```
# let send_m a = a#m ;;
(* value send_m : < m : a; .. > -> a = <fun> *)
```

Здесь многоточие (из двух точек) — это принятый в OCaml синтаксис для безымянной рядной ти́повой переменной. За счёт такой типизации, функция `send_m` может быть применена к объекту *любого* (заранее не известного) объектного типа, в состав которого входит метод `m` соответствующей сигнатуры.

Выведение типов для исчисления Ванда в первоначальной версии было неполным: из-за отсутствия ограничений на расширение ряда, добавление поля при совпадении имени подменит существующее — в результате не все программы имеют главный тип. Однако, эта система была первым конкретным предложением по расширению ML записями, поддерживающими наследование. В последующие годы был предложен целый ряд различных доработок, в том числе, делающих его полным.

Наиболее заметный след оставил Дидье́ Реми́ (фр. *Didier Rémy*). Он построил практичную систему типов с расширяемыми записями, включающую полный и эффективный алгоритм реконструкции типов. Реми расслаивает язык типов на сорта, формулируя сортированную алгебру типов (англ. *sorted algebra of types, sorted language of types*). Различаются сорт собственно *типов* (в том числе типов полей) и сорт *полей*; вводятся отображения между ними и на их основе формулируются правила типизации расширяемых записей как простое расширение классических правил ML. Информация о *присутствии* (англ. *presence*) поля определяется как отображение из сорта *типов* в сорт *полей*. Рядные ти́повые переменные переформулируются как переменные, принадлежащие сорту *полей* и равные константе *отсутствия* (англ. *absence*), являющейся элементом сорта *полей*, не имеющим соответствия в сорте *типов*. Операция вычисления типа для записи из `n` полей определяется как отображение *n-арного поля* в *тип*

(где каждое поле в кортеже либо вычисляется функцией *присутствия*, либо задаётся константой *отсутствия*).

Упрощённо идею исчисления можно трактовать как расширение типа всякого поля записи *флагом присутствия/отсутствия* и представление записи в виде кортежа со слотом для каждого возможного поля. В прототипе реализации синтаксис языка типов был сделан приближенным к теоретико-ти́повой формулировке, например:

```
# let car = { name = "Toyota"; age = "old"; id =
7866 } ;;
(* car : ∏ (name : pre (string); id : pre (num);
age : pre (string); abs) *)

# let truck = { name = "Blazer"; id = 6587867567 }
;;
(* truck : ∏ (name : pre (string); id : pre (num);
abs) *)

# let person = { name = "Tim"; age = 31; id =
5656787 } ;;
(* person : ∏ (name : pre (string); id : pre (num);
age : pre (num); abs) *)
```

(символ ∏ у Реми означает операцию вычисления типа)

Добавление нового поля записывается с помощью конструкции `with`:

```
# let driver = { person with vehicle = car } ;;
(* driver : ∏ (vehicle : pre (∏ (name : pre
(string); id : pre (num); age : pre (string);
abs));
              name : pre (string); id : pre (num);
age : pre (num); abs) *)
```

Функциональное обновление записывается идентично, с той разницей, что упоминание уже существующего поля переопределяет его:

```
#let truck_driver = { driver with vehicle = truck
};;
```

```
(* truck driver : ∏ (vehicle : pre (∏ (name : pre
(string); id : pre (num); abs));
                    name : pre (string); id : pre
(num); age : pre (num); abs) *)
```

Эта схема формализует ограничение, необходимое для проверки операций над записями и выведения главного типа, но не ведёт к очевидной и эффективной реализации. Реми использует хеширование, что довольно эффективно в среднем, но неизбежно накладывает оверхед на райнтайм даже для исходно мономорфных программ и плохо подходит для записей с большим числом полей.

В дальнейшем Реми исследовал использование рядного полиморфизма совместно с выделением подтипов данных, подчеркнув, что это ортогональные понятия, и показав, что записи становятся наиболее выразительными при их одновременном использовании. На этой основе он совместно с Жеромом Вуйоном (фр. *Jérôme Vouillon*) предложил легковесное объектно-ориентированное расширение для ML. Это расширение было реализовано в языке «`Caml Special Light`» Ксавье Леруа (Xavier Leroy (англ.)), превратив его в OCaml. Объектная модель OCaml тесно сплетает использование структурной подтипизации и рядного полиморфизма, из-за чего порой их ошибочно отождествляют. Рядный полиморфизм произведений в OCaml лежит в основе выведения типов; отношения подтипизации не являются необходимыми в языке с его поддержкой, но дополнительно повышают гибкость на практике. В OCaml реализован более простой и наглядный синтаксис для информации о типах.

Жак Гарри́га (фр. *Jacques Garrigue*) реализовал практичную систему полиморфных сумм. Он совместил теоретические работы Реми и Охори, построив систему, пролегающую посередине: информация о *наличии* меток в записи представляется посредством использования родо́в, а информация об *их типах* использует рядные переменные. Чтобы компилятор мог отличать полиморфные суммы от мономорфных, Гаррига использует специальный синтаксис (обратный апостроф, предваряющий тег). При этом исчезает необходимость в объявлении типа — можно сразу писать функции над ним, и компилятор будет выводить минимальный список тегов по мере композиции этих функций. Эта система стала частью OCaml около 2000 года, но не *вместо*, а *в дополнение* к мономорфным суммам, так как они несколько менее эффективны, и из-за невозможности контроля

полноты разбора затрудняют поиск ошибок (в отличие от решения Блюма).

Из недостатков рядного полиморфизма Ванда можно отметить неочевидность реализации (нет единого систематичного способа его компилировать, каждая конкретная система типов на основе рядных переменных имеет свою реализацию) и неоднозначное соотношение с теорией (нет единообразной формулировки для проверки и выведения типов, поддержка выведения решалась отдельно и потребовала экспериментов с наложением различных ограничений).

3.9. Просвечивающие суммы Харпера — Лилибриджа

Наиболее сложной разновидностью записей являются зависимые записи. Такие записи могут включать в себя типы наравне с «обычными» значениями (*материализованные типы*, reified (англ.) types), причём термы и типы, следующие далее по порядку в теле записи, могут быть определены на основе предшествующих им. Такие записи соответствуют «*слабым суммам*» из теории зависимых типов, также известным как «*экзистенциалы*», и служат наиболее общим обоснованием систем модулей языков программирования. Карделли рассматривал аналогичные по свойствам типы как один из основных типов в полнотиповом программировании (но называл их «кортежами»).

Роберт Харпер (Robert Harper (англ.)) и Марк Лилибридж (*Mark Lillibridge*) построили исчисление **просвечивающих сумм** (англ. *translucent sums*) для формального обоснования языка модулей первого класса высшего порядка — наиболее развитой системы модулей среди известных. Это исчисление, в том числе, применяется в семантике Харпера — Стоуна, представляющей теоретико-ти́повое обоснование для Standard ML.

Просвечивающие суммы обобщают *слабые суммы* за счёт меток и набора равенств, описывающих конструкторы типов. Термин «*просвечивающие*» (англ. *translucent*) означает, что в составе типа записи могут присутствовать как типы с явно экспортированной структурой, так и полностью абстрактные. Слой родо́в в исчислении имеет простой классический состав: различаются род всех типов * и

функциональные рода́ вида $k_1 \Rightarrow k_2$, типизирующие конструкторы типов (ML не поддерживает полиморфизма в высших рода́х, все переменные типа принадлежат к роду *, и абстракция конструкторов типов возможна лишь посредством функторов). Исчисление различает правила подтипизации для *записей* как основных типов и для *полей записей* как их составляющих, соответственно, рассматривая «подтипы» и «подполя», а затемнение (абстрагирование) сигнатур полей является отдельным от подтипизации понятием. Подтипизация здесь формализует сопоставление модулей с интерфейсами.

Исчисление Харпера — Лилибриджа неразрешимо даже в части проверки согласования типов (диалекты языка модулей, реализованные в Standard ML и OCaml, используют ограниченные подмножества этого исчисления). Однако позже Андреас Россберг (англ. *Andreas Rossberg*) на основе их идей построил язык «**1ML**», в котором традиционные записи уровня ядра языка и стуктуры уровня модулей являются одной и той же первоклассной конструкцией (существенно более выразительной, чем у Карделли — см. критика языка модулей ML). За счёт подключения идеи Харпера и Митчела о подразделении всех типов на вселенные «маленьких» и «больших» типов (упрощённо, это похоже на фундаментальное разделение типов на простые и агрегатные, с неодинаковыми правилами типизации), Россберг обеспечил разрешимость не только проверки согласования, но и почти полного выведения типов. Более того, 1ML допускает импредикативный полиморфизм. При этом внутренний язык в 1ML основан на плоской Системе F_ω и не требует использования зависимых типов в качестве метатеории. На 2015 год Россберг оставил открытым вопрос о возможности добавления в 1ML рядного полиморфизма, отметив лишь, что это должно обеспечить более полное выведение типов. Спустя год он добавил в 1ML полиморфизм эффектов.

3.10. Полиморфное исчисление записей Охори

Ацуси Охори (англ. *Atsushi Ohori*) совместно со своим научным руководителем Питером Бьюнеманом (Peter Buneman (англ.)) в 1988 году предложил идею ограничивать спектр возможных значений обычных типовых переменных в полиморфной типизации самих записей. В дальнейшем Охори формализовал эту идею посредством родо́в записей, построив к 1995 году полноценное исчисление и способ его эффективной компиляции. Прототип реализации был создан в 1992 году как расширение компилятора SML/NJ, затем Охори возглавил

разработку собственного компилятора SML#, реализующего одноимённый диалект языка Standard ML. В SML# полиморфизм записей служит основой для бесшовного встраивания конструкций на SQL в программы на SML. SML# применяется крупными японскими компаниями для решения бизнес-задач, связанных с нагруженными базами данных. Пример такого рода сессии (REPL):

```
fun wealthy { Salary = s, ... } = s > 100000;
(* val wealthy = fn : 'a#{ Salary:int, ... } ->
bool *)

fun young x = #Age x < 24;
(* val young = fn : 'a#{ Age:int, ... } -> bool *)

fun youngAndWealthy x = wealthy x andalso young x;
(* val youngAndWealthy = fn : 'a#{ Age:int,
Salary:int, ... } -> bool *)

fun select display l pred = fold (fn (x,y) => if
pred x then (display x)::y else y) l nil;
(* val select = fn : ('a -> 'b) -> 'a list -> ('a -
> bool) -> 'b list *)

fun youngAndWealthyEmployees l = select #Name l
youngAndWealthy;
(* val youngAndWealthyEmployees = fn : 'b#{
Age:int, Name:'a, Salary:int, ... } list -> 'a list
*)
```

Охори назвал своё исчисление «*полиморфизмом записей*» (англ. *record polymorphism*) или «*полиморфным исчислением записей*» (англ. *polymorphic record calculus*), одновременно подчеркнув, что он, как и Ванд, рассматривает полиморфизм не только типов-произведений, но и типов-сумм.

Исчисление Охори выделяется наиболее интенсивным использованием слоя родóв. В записи $t::k$ (отнесение типа t к роду k) символ k *означает* либо род всех типов U; либо род записей, имеющих форму $\{\{l_1:t_1,\ldots, l_n:t_n\}\}$, обозначающий множество всех записей, содержащих как минимум указанные поля; либо род вариантов, имеющих форму $<< l_1:t_1,\ldots, l_n:t_n >>$, обозначающий множество всех вариантных типов, содержащих как минимум указанные конструкторы. В плоском

синтаксисе языка ограничение типа некоторым родом записи записывается как `t#{...}` (см. примеры выше). Решение в некоторой степени схоже с ограниченной квантификацией Карделли — Вегнера.

Единственная *полиморфная* операция, предусмотренная этим исчислением — операция извлечения поля. Однако Охори был первым, кто представил для полиморфизма записей простую и эффективную схему компиляции. Он назвал её «исчислением реализаций» (*implementation calculus*). Запись представляется вектором, упорядоченным лексикографически по именам полей исходной записи; обращение к полю по имени транслируется в вызов промежуточной функции, возвращающей номер данного поля в данном векторе по запрашиваемому имени, и последующую индексацию вектора по вычисленному номеру позиции. Функция вызывается только при инстанцировании полиморфных термов, что накладывает минимальный оверхед на рантайм при использовании полиморфизма и не накладывает никакого оверхеда при работе с мономорфными типами. Метод работает одинаково хорошо с произвольно большими записями и вариантами. Исчисление обеспечивает выведение типов и находит строгое соответствие с теорией (родовая квантификация напрямую соотносится с обычной индексацией вектора), представляя собой непосредственно расширение лямбда-исчисление второго порядка Жирара — Рейнольдса, что позволяет переносить различные известные свойства полиморфной типизации на полиморфизм записей.

На практике, поддержка полиморфных вариантов в SML# не была реализована из-за её несовместимости с механизмом определения типов-сумм в Standard ML (требуется синтаксическое разделение сумм и рекурсивных типов).

Недостатком исчисления Охори является отсутствие поддержки операций расширения или обрезания записей.

3.11. Первоклассные метки Гастера — Джонса

В теории квалифицированных типов расширяемые записи описываются предикатами *отсутствия* поля (*«lacks» predicate*) и *присутствия* поля (*«has» predicate*). Бенедикт Гастер (*Benedict R. Gaster*) совместно с автором теории Марком Джонсом (*Mark P. Jones*) доработал её части расширяемых записей до практичной системы типов неявно типизированных языков, в том числе, определив способ

компиляции. Они вводят термин **первоклассные метки** (*first-class labels*), подчёркивающий возможность абстрагировать операции над полями от статически известных меток. В дальнейшем Гастер защитил на построенной системе диссертацию.

Исчисление Гастера — Джонса не обеспечивает полное выведение типов. Кроме того, из-за проблем разрешимости было наложено искусственное ограничение: запрет на пустые ряды. Сульцманн предпринял попытку реформулирования исчисления, однако построенная им система не может быть расширена до поддержки полиморфного расширения записей, и не имеет универсального метода эффективной компиляции.

Да́ан Ле́йен (*Daan Leijen*) добавил в исчисление Гастера — Джонса предикат *рядного равенства* (или *равенства ряда*, англ. *row equality predicate*) и переместил язык рядов в язык предикатов — это обеспечило полное выведение типов и сняло запрет на пустые ряды. При компиляции записи преобразуются в лексикографически упорядоченный кортеж и применяется *трансляция свидетельств* по схеме Гастера — Джонса. Система Лейена позволяет выражать такие идиомы, как сообщения высшего порядка[en] (наиболее мощную форму объектно-ориентированного программирования) и первоклассные ветвления.

На основе этих работ реализованы расширения языка Haskell.

Результаты Гастера — Джонса очень близки к результатам Охори, несмотря на существенные различия в теоретико-ти́повом обосновании, и основным преимуществом является поддержка операций расширения и обрезания записей. Недостатком исчисления является то, что оно опирается на свойства системы типов, которые отсутствуют в большинстве языков программирования. Кроме того, выведение типов для него представляет серьёзную проблему, из-за чего авторы наложили дополнительные ограничения. И хотя Лейен устранил многие недостатки, использование `default`-ветвления невозможно.

3.12. Полиморфизм управляющих конструкций

С развитием программных систем может увеличиваться количество вариантов в типе-сумме, и добавление каждого варианта требует добавления соответствующего ветвления в каждую функцию над этим типом, даже если эти ветвления в разных функциях идентичны. Таким образом, трудоёмкость наращивания функциональности в большинстве языков программирования нелинейно зависит от декларативных изменений в техническом задании. Эта закономерность известна как **проблема выражения**. Другой известной проблемой является обработка исключений: на протяжении десятилетий исследования систем типов, все языки, относимые к типобезопасным, могли, тем не менее, завершать работу порождением непойманного исключения — поскольку, несмотря на типизацию самих исключений, механизм их порождения и обработки не типизировался. И хотя были построены *средства анализа* потока исключений, эти средства всегда являлись внешними по отношению к языку.

Матиас Блюм (англ. *Matthias Blume*, коллега Эндрю Аппеля, работающий над проектом successor ML), его аспирант Вонсёк Чэй (*Wonseok Chae*) и коллега Юмат Эйкар (*Umut Acar*) решили обе проблемы, основываясь на математической двойственности произведений и сумм. Воплощением их идей стал язык **MLPolyR**, основанный на простейшем подмножестве Standard ML и дополняющий его несколькими уровнями типобезопасности. Позже Вонсёк Чэй защитил на этих достижениях диссертацию.

Решение состоит в следующем. Согласно принципу двойственности, *вводная форма* (англ. *introduction form*) для некоего понятия соответствует *устраняющей форме* (англ. *elimination form*) двойственного ему. Таким образом, устраняющая форма сумм (разбор ветвлений) соответствует вводной форме записей. Это побуждает наделить ветвления теми же свойствами, что уже доступны для записей — сделать их первоклассными объектами и допустить их расширение.

Например, простейший интерпретатор языка выражений:

```
fun eval e = case e of
                  `Const i => i
```

```
                | `Plus (e1,e2) => eval e1 + eval
e2
```

с введением первоклассной конструкции `cases` может быть переписан в виде:

```
fun eval e = match e with
                cases `Const i => i
                    | `Plus (e1,e2) => eval e1 +
eval e2
```

после чего `cases`-блок может быть вынесен:

```
fun eval_c eval = cases `Const i => i
                      | `Plus (e1,e2) => eval e1 +
eval e2

fun eval e = match e with (eval_c eval)
```

Это решение допускает `default`-ветвления (в отличие от исчисления Гастера — Джонса), что важно для *композиции* первоклассных ветвлений. Завершение композиции ряда осуществляется с помощью слова `nocases`.

```
fun const_c d =
    cases `Const i => i
    default: d

fun plus_c eval d =
    cases `Plus (e1,e2) => eval e1 + eval e2
    default: d

fun eval e = match e with
    const_c (plus_c eval nocases)


fun bind env v1 x v2 =
    if v1 = v2 then x else env v2

fun var_c env d =
    cases `Var v => env v
```

```
    default: d

fun let_c eval env d =
    cases `Let (v,e,b) => eval (bind env v (eval
env e)) b
    default: d

fun eval_var env e = match e with
    const_c (plus_c (eval_var env) (var_c env
(let_c eval_var env nocases)))
```

Как видно, новый код, который необходимо дописывать при качественном усложнении системы, не требует изменения уже написанного кода (функции `const_c` и `plus_c` «ничего не знают» о последующем добавлении в интерпретатор языка поддержки переменных и `let`-блоков). Таким образом, *первоклассные расширяемые ветвления* (*first-class extensible cases*) являются принципиальным решением проблемы выражения, позволяя говорить о парадигме ***расширяемого программирования***. По словам Блюма, это является не чем-то принципиально новым, а просто интересным способом применения рядного полиморфизма, который уже поддерживается в системе типов, и в этом смысле достоинством такого технического решения является его концептуальная простота.

Однако расширение программных систем требует также контроля над обработкой исключений, которые могут порождаться на произвольной глубине вложения вызовов. И здесь Блюм с коллегами провозглашают новый слоган типобезопасности в развитие слогана Милнера: «*Программы, прошедшие проверку типов,* ***не порождают необработанных исключений***». Проблема состоит в том, что если сигнатура функционального типа включает информацию о типах исключений, которые эта функция потенциально может порождать, и эта информация в сигнатуре передаваемой функции должна быть строго согласована с информацией о параметре функции высшего порядка (в том числе, если это пустое множество) — типизация *механизма обработки* исключений немедленно требует *полиморфности* типов самих исключений — в противном случае функции высшего порядка перестают быть полиморфными. В то же время, в безопасном языке исключения являются *расширяемой суммой*, то есть вариантным типом, конструкторы которого добавляются по ходу программы. Соответственно, типобезопасность потока исключений означает необходимость поддержки типов-сумм, которые

являются одновременно расширяемыми и полиморфными. И здесь вновь решением является рядный полиморфизм.

Как и в исчислении Гарриги, для полиморфных сумм в MLPolyR используется специальный синтаксис (обратный апостроф, предваряющий тег), и нет нужды в предварительном объявлении типа-суммы (то есть вышеприведённый код — это *вся* программа, а не фрагмент). Преимущество состоит в том, что проблемы с контролем полноты разбора не возникает: семантика MLPolyR определена через преобразование во внутренний язык с доказанной надёжностью, не поддерживающий ни полиморфизма сумм, ни исключений (не говоря уже о непойманных исключениях), так что необходимость их удаления на этапе компиляции сама по себе является доказательством надёжности.

MLPolyR использует нетривиальное сочетание нескольких исчислений и двухстадийную трансляцию. Для выведения и согласования типов он использует исчисление Реми, одновременно используя принцип математической двойственности для представления сумм как произведений, далее транслирует язык в промежуточный явно типизированный язык с полиморфными записями, и затем использует эффективный способ компиляции, построенный Охори. Иначе говоря, модель компиляции Охори была обобщена до поддержки исчисления Реми. На теоретико-ти́повом уровне Блюм вводит сразу несколько новых синтаксических нотаций, позволяющих записывать правила для типизации исключений и первоклассных ветвлений. Система типов MLPolyR обеспечивает полное выведение типов, так что авторы отказались от разработки плоского синтаксиса для явной записи типов и от поддержки сигнатур в языке модулей.

В системе типов Лейена также возникает вариант полиморфизма ветвлений: конструкция `case` может быть представлена в виде функции высшего порядка, получающей запись из функций, каждая из которых соответствует определённой ветви вычислений (синтаксис Хаскела подходит для этого изменения и не требует пересмотра). Например:

```
data List a = nil :: {}
            | cons :: { hd :: a, tl :: List a }

snoc xs r = case (reverse xs) r
```

```
last xs = snoc xs { nil = \r -> _|_,
                    cons = \r -> r.hd }
```

Поскольку записи в системе Лейена являются расширяемыми, разбор ветвлений обретает гибкость на уровне динамических решений (например, цепочки проверок или использования ассоциативного массива), но обеспечивает гораздо более эффективную реализацию (метка варианта соответствует смещению в записи). Однако, от поддержки ветвления по умолчанию (`default`) в данном случае приходится отказаться, поскольку единственный `default`-образец соответствовал бы множеству полей (и, соответственно, множеству смещений). Лейен называет эту конструкцию «*первоклассными образцами для сопоставления*» (*first-class patterns*).

3.13. Полиморфизм в высших рода́х

Полиморфизм в высших рода́х (англ. *higher-kinded polymorphism*) означает абстракцию над конструкторами типов высших порядков, то есть ти́повыми операторами вида

```
* -> * -> ... -> *
```

Поддержка этой возможности поднимает полиморфизм на более высокий уровень, обеспечивая абстракцию как над типами, так и над конструкторами типов — подобно тому как функции высших порядков обеспечивают абстракцию как над значениями, так и над другими функциями. Полиморфизм в высших рода́х является естественным компонентом многих идиом функционального программирования, включая монады, свёртки и встраиваемые языки.

Например, если определить следующую функцию (язык Haskell):

```
when b m = if b then m else return ()
```

то для неё будет выведен такой функциональный тип:

```
when :: forall (m :: * -> *). Monad m => Bool -> m
() -> m ()
```

Сигнатура `m :: * -> *` говорит о том, ти́повая переменная `m` является переменной *типа, принадлежащего к высшему роду* (англ. *higher-kinded type variable*). Это значит, что она абстрагируется над конструкторами типов (в данном случае унарными, такими как `Maybe` или `[]`), которые могут применяться к конкретным типам, таким как `Int` или `()`, для построения новых типов.

В языках, поддерживающих полную абстракцию типа (Standard ML, OCaml), все ти́повые переменные должны принадлежать к роду `*`, в противном случае система типов была бы небезопасной. Полиморфизм в высших родáх, таким образом, обеспечивается за счёт самого механизма абстракции в сочетании с явным аннотированием при инстанцировании, что несколько неудобно. Тем не менее, возможна идиоматическая реализация полиморфизма в высших родáх, не требующая явного аннотирования — для этого на уровне типов применяется техника, аналогичная дефункциализации.

3.14. Полиморфизм родóв

Системы родóв (англ. *kind systems*) обеспечивают безопасность самих систем типов, позволяя контролировать смысл ти́повых выражений.

Например, пусть требуется реализовать вместо обычного типа «*вектор*» (линейный массив) семейство типов «*вектор длиной* `n`», иначе говоря, определить тип «*вектор, индексированный длиной*». Классическая реализация на Haskell выглядит так:

```
data Zero
data Succ n
data Vec :: * -> * -> * where
    Nil  :: Vec a Zero
    Cons :: a -> Vec a n -> Vec a (Succ n)
```

Здесь вначале определяются *фантомные типы*, то есть типы, не имеющие динамического представления. Конструкторы `Zero` и `Succ` служат «значениями слоя типов», а переменная `n` обеспечивает неравенство разных конкретных типов, построенных конструктором

Succ. Тип Vec определён как обобщённый алгебраический тип данных (GADT).

Обобщённый алгебраический тип данных

Обобщённый алгебраи́ческий тип да́нных (GADT, англ. *generalized algebraic data type*) — один из видов алгебраических типов данных, который характеризуется тем, что его конструкторы могут возвращать значения не своего типа, связанного с ним. Обобщённые АТД были придуманы под влиянием работ об индуктивных семействах в среде исследователей зависимых типов.

Такие типы реализованы в нескольких языках программирования, в частности в языках OCaml (начиная с версии 4), Idris, Agda и Haskell, причём в последнем оно не входит в стандарт языка **Haskell-98**, а реализовано только в одном из расширений компилятора GHC. Язык Haskell имитирует индуктивное семейство (англ. *inductive family*), представляя их типами, индексированными другими типами.

Ранняя версия обобщённых АТД была описана Леннартом Аугустсоном и Кентом Петерсоном в 1994 году и основывалась на сопоставлении с образцом в системе доказательства теорем ALF.

Обобщённые АТД были введены в 2003 году независимо Чейни (Cheney) и Хинце (Hinze) и до этого Си (Xi), Ченом (Chen) и Ченом (Chen) как расширения алгебраических типов данных ML и Haskell. Введённые обобщённые АТД оказались эквивалентны друг другу и похожи на индуктивные семейства типов данных (или индуктивные типы данных), используемые в Coq в исчислении индуктивных конструкций.

В 2006 году Sulzmann, Wazny и Stuckey ввели ***расширенные алгебраические типы данных***, сочетающие обобщённые АТД с экзистенциальными типами данных (англ.)русск. и ограничениями класса типов (англ.)русск., введёнными Перри (Perry), Ляуфером (Läufer) и Одерски (Odersky) в середине 1990-х.

Вывод типов при отсутствии деклараций типов, заданных программистом, является алгоритмически неразрешимой задачей, а

функции, определённые на обобщённых АТД, в общем случае могут не принимать основные типы (англ.)русск.(principal types).

Реконструкция типа требует при проектировании нескольких компромиссов и является по состоянию на 2011 год темой исследований.

Обобщённые АДТ применяются в обобщённом программировании, моделировании абстрактного синтаксиса высшего порядка (англ. *higher-order abstract syntax*) языков программирования и моделировании объектов, сохранении инвариантов структур данных, выражении ограничений во встроенных предметно-ориентированных языках.

Пример на Haskell

В следующем примере определяется обобщённый тип `Type`, в котором представлены несколько типов:

```
data Type :: * -> * where
    Char :: Type Char
    Int :: Type Int
    List :: Type a -> Type [a]
```

Для этого типа можно составить ad hoc-полиморфную функцию суммирования:

```
sum :: Type a -> a -> Int
sum Char _ = 0
sum Int n = n
sum (List a) xs = foldr (+) 0 (map (sum a) xs)
```

Которую можно применять для типов, поддерживаемых `Type`, например, для типа `[Int]`:

```
sum (List Int) [1, 2, 4]
```

Решение условно предполагает, что фантомный тип `n` будет использоваться для моделирования целочисленного параметра вектора на основе аксиом Пеано — то есть будут строиться только такие

выражения, как `Succ Zero`, `Succ Succ Zero`, `Succ Succ Succ Zero` и т. д. Однако, хотя определения записаны на языке типов, сами по себе они сформулированы бестиповым образом. Это видно по сигнатуре `Vec :: * -> * -> *`, означающей, что конкретные типы, передаваемые в качестве параметров, принадлежат роду `*`, а значит, могут быть любым конкретным типом. Иначе говоря, здесь не запрещаются бессмысленные ти́повые выражения вроде `Succ Bool` или `Vec Zero Int`.

Более развитое исчисление позволило бы задать область значений параметра типа более точно:

```
data Nat = Zero | Succ Nat
data Vec :: * -> Nat -> * where
   VNil  :: Vec a Zero
   VCons :: a -> Vec a n -> Vec a (Succ n)
```

Но обычно такой выразительностью обладают лишь системы с зависимыми типами, реализованные в таких языках, как Agda, Coq и др., которые имеют очень высокий порог вхождения и трудоёмки в использовании. Например, с позиции языка Agda запись `Vec :: * -> Nat -> *` означала бы, что род типа `Vec` *зависит* от типа `Nat` (то есть элемент одного сорта зависит от элемента другого, более низкого сорта).

Зависимый тип

Зависимый тип в информатике и логике — тип, который зависит от некоторого значения. Зависимые типы играют ключевую роль в интуиционистской теории типов и построении функциональных языков программирования таких как ATS, Agda и Epigram.

К примеру, тип, описывающий *n*-кортежи действительных чисел является зависимым, так как он «зависит» от величины *n*.

Решение о равенстве зависимых типов в программе может потребовать вычислений. Если в зависимых типах допущено использование произвольных значений, то решение о равенстве типов может включать в себя проверку равенства результата работы двух

произвольных программ. Таким образом проверка типа становится неразрешимой задачей.

Изоморфизм Карри — Ховарда позволяет строить типы для выражения сколь угодно сложных математических свойств. Если предоставлено конструктивное доказательство того, что тип «заселён» (то есть, существует хотя бы одно значение этого типа), компилятор сможет проверить это доказательство и превратить его в исполняемый код, вычисляющий значение. Наличие проверки доказательств делает зависимо-типизированные языки схожими с программным обеспечением автоматизации доказательств (например, интерактивный доказатель теорем Coq).

3.15. Системы лямбда-куба

Хенк Барендрегт разработал лямбда-куб в качестве средства классификации систем типов по трем осям. Каждый из восьми углов кубической диаграммы соответствует системе типов. В наиболее бедной вершине куба находится просто типизированное лямбда-исчисление, а в наиболее богатой — исчисление конструкций. Трём осям куба соответствуют три различных дополнения к просто типизированному лямбда-исчислению: дополнение зависимых типов, дополнение полиморфизма и дополнение конструкторов типов высшего порядка.

Очень упрощённо зависимый тип можно представить как тип индексированного семейства множеств. Более формально, для типа $A:U$ (где U — вселенная типов), можно определить семейство типов B, сопоставляющее каждому терму $\alpha:A$ тип $B(\alpha):U$, что записывается как $B:A\rightarrow U$. Функция, чья область значений варьируется в зависимости от её аргумента, называется **зависимой функцией**. Тип этой функции называется **зависимым произведением типов**, **пи-типом** или просто **зависимым типом**. Зависимый тип записывается для данного случая как

$$\Pi_{(x:A)}\, B(x)$$

или

$$\Pi(x:A),\, B(x)$$

Если A является константой, то зависимый тип упрощается до обычной функции $A \to B$. Иначе говоря, $\Pi_{(x:A)}\ B$ равен функциональному типу $A \to B$. Название «*пи-тип*» подчёркивает, что такой тип является декартовым произведением типов. Пи-типы также могут быть представлены как модели кванторов всеобщности.

Например, если Vec(**R**,*n*)— кортеж из *n* вещественных чисел, то $\Pi_{(n:\mathbf{N})}$ Vec(**R**,*n*)— тип функций, которые для всякого натурального *n* возвращают кортеж вещественных чисел размера *n*. Обычное Функциональное пространство является тем частным случаем, когда область значений не зависит от входного параметра: например, $\Pi_{(n:\mathbf{N})}\ \mathbf{R}$ — тип функций из натуральных в вещественные, обозначаемых $\mathbf{N} \to \mathbf{R}$ в просто типизированном лямбда-исчислении.

Полиморфные функции являются важным примером зависимых функций, т.е. функций, имеющих зависимый тип. Для некоторого заданного типа такие функции обрабатывают значения этого типа, или значения типа, построенного на основе этого типа. Полиморфная функция, возвращающая значения типа C будет иметь полиморфный тип, записываемый как

$\Pi_{(A:U)}\ A \to C$.

В 2012 году было построено расширение языка Haskell, реализующее более развитую систему родóв и делающее вышеприведённый код корректным кодом на Хаскеле. Решение состоит в том, что все типы (за определёнными ограничениями) автоматически «*продвигаются*» (англ. *promote*) на уровень выше, формируя одноимённые родá, которые можно использовать явным образом. С этой точки зрения, запись `Vec :: * -> Nat -> *` не является зависимой — она означает лишь, что второй параметр вектора должен принадлежать к именованному роду `Nat`, а в данном случае единственным элементом этого рода является одноимённый тип.

Решение является весьма простым, как с точки зрения реализации в компиляторе, так и с точки зрения практической доступности. А поскольку полиморфизм типов изначально является естественным элементом семантики Хаскела, продвижение типов приводит к ***полиморфизму родóв*** (англ. *kind polymorphism*), который одновременно повышает коэффициент повторного использования кода

и обеспечивает более высокий уровень типобезопасности. Например, следующий GADT, используемый для верификации равенства типов:

```
data EqRefl a b where
   Refl :: EqRefl a a
```

в классическом Хаскеле имеет род `* -> * -> *`, что не позволяет использовать его для проверки равенства конструкторов типов, таких как `Maybe`. Система родов, основанная на продвижении типов, выводит **полиморфный** род `forall X. X -> X -> *`, делая тип `EqRefl` более универсальным. Это можно записать явно:

```
data EqRefl (a :: X) (b :: X) where
   Refl :: forall X. forall(a :: X). EqRefl a a
```

3.16. Полиморфизм эффектов

Системы эффектов (англ. *effect systems*) были предложены Гиффордом и Лукассеном во второй половине 1980-х с целью обособления побочных эффектов для более тонкого контроля за безопасностью и эффективностью в конкурентном программировании.

Побочный эффект (программирование)

Побо́чные эффе́кты (англ. *side effects*) — любые действия работающей программы, изменяющие среду выполнения (англ. *execution environment*). Например, к побочным эффектам относятся:

- доступ (чтение или запись) к объекту, определённому с модификатором volatile (англ.);
- изменение (запись) объекта;
- изменение файла;
- изменение поведения инструкций процессора, обрабатывающих числа с плавающей точкой (см. floating-point environment (англ.));
- вызов функции, выполняющей любое из перечисленных выше действий.

Побочный эффект функции — возможность в процессе выполнения своих вычислений: читать и модифицировать значения глобальных переменных, осуществлять операции ввода-вывода, реагировать на исключительные ситуации, вызывать их обработчики. Если вызвать функцию с побочным эффектом дважды с одним и тем же набором значений входных аргументов, может случиться так, что в качестве результата будут возвращены разные значения. Такие функции называются недетерминированными функциями с побочными эффектами.

Полиморфизм эффектов (англ. *effect polymorphism*) при этом означает квантификацию над чистотой конкретной функции, то есть включение в функциональный тип флага, характеризующего функцию как чистую либо нечистую. Такое расширение типизации позволяет абстрагировать чистоту функций высшего порядка от чистоты функций, передаваемых им в качестве аргументов.

Это приобретает особое значение при переходе к функциям над модулями (записями, включающими в свой состав абстрактные типы) — функторам — поскольку в условиях чистоты они имеют право быть аппликативными, но в присутствии побочных эффектов они обязаны быть порождающими для обеспечения типобезопасности (подробнее об этом см. эквивалентность типов в языке модулей ML). Таким образом, в языке модулей первого класса высшего порядка полиморфизм эффектов оказывается необходимой основой для поддержки ***полиморфизма порождаемости*** (англ. *generativity polymorphism*): передача флага чистоты в функтор обеспечивает выбор между аппликативной и порождающей семантикой в единой системе.

3.17. Поддержка в языках программирования

Типобезопасный параметрический полиморфизм доступен в языках, типизированных по Хиндли — Милнеру — в диалектах ML (Standard ML, OCaml, Alice, F#) и их потомках (Haskell, Clean, Idris, Mercury, Agda) — а также в наследованных от них гибридных языках (Scala, Nemerle).

Обобщённые типы данных (дженерики) отличаются от параметрически полиморфных систем тем, что используют ограниченную квантификацию, и потому не могут иметь ранг выше 1-го. Они

доступны в Ada, Eiffel, Java, C#, D; а также в Delphi, начиная с 2009-й версии. Впервые они появились в CLU.

Обобщённое программирование (англ. *generic programming*) — парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание. В том или ином виде поддерживается разными языками программирования. Возможности обобщённого программирования впервые появились в виде дженериков (обобщённых функций) в 1970-х годах в языках Клу и Ада, затем в виде параметрического полиморфизма в ML и его потомках, а затем во многих объектно-ориентированных языках, таких как C++, Java, Object Pascal[1], D, Eiffel, языках для платформы .NET и других.

Методология обобщённого программирования

Обобщённое программирование рассматривается как методология программирования, основанная на разделении структур данных и алгоритмов через использование абстрактных описаний требований. Абстрактные описания требований являются расширением понятия абстрактного типа данных. Вместо описания отдельного типа в обобщённом программировании применяется описание семейства типов, имеющих общий интерфейс и семантическое поведение (англ. *semantic behavior*). Набор требований, описывающий интерфейс и семантическое поведение, называется *концепцией* (англ. *concept*). Таким образом, написанный в обобщённом стиле алгоритм может применяться для любых типов, удовлетворяющих его своими концепциями. Такая возможность называется **полиморфизмом**.

Говорят, что тип *моделирует* концепцию (является моделью концепции), если он удовлетворяет её требованиям. Концепция является *уточнением* другой концепции, если она дополняет последнюю. Требования к концепциям содержат следующую информацию:

- **Допустимые выражения** (англ. *valid expressions*) — выражения языка программирования, которые должны успешно компилироваться для типов, моделирующих концепцию.

- **Ассоциированные типы** (англ. *associated types*) — вспомогательные типы, имеющие некоторое отношение к моделирующему концепцию типу.
- **Инварианты** (англ. *invariants*) — такие характеристики типов, которые должны быть постоянно верны во время исполнения. Обычно выражаются в виде предусловий и постусловий. Невыполнение предусловия влечёт непредсказуемость соответствующей операции и может привести к ошибкам.
- **Гарантии сложности** (англ. *complexity guarantees*) — максимальное время выполнения допустимого выражения или максимальные требования к различным ресурсам в ходе выполнения этого выражения.

В C++ ООП реализуется посредством виртуальных функций и наследования, а ОП — с помощью шаблонов классов и функций. Тем не менее, суть обеих методологий связана с конкретными технологиями реализации лишь косвенно. Говоря более формально, ООП основано на полиморфизме подтипов, а ОП — на параметрическом полиморфизме. В других языках то и другое может быть реализовано иначе. Например, мультиметоды в CLOS имеют сходную с параметрическим полиморфизмом семантику.

Массер и Степанов выделяют следующие этапы в решении задачи по методологии ОП:

1. Найти полезный и эффективный алгоритм.
2. Определить обобщённое представление (параметризовать алгоритм, минимизировав требования к обрабатываемым данным).
3. Описать набор (минимальных) требований, удовлетворяя которые всё ещё можно получить эффективные алгоритмы.
4. Создать каркас на основе классифицированных требований.

Минимизация и создание каркаса ставят целью создание такой структуры, при которой алгоритмы не зависят от конкретных типов данных. Этот подход отражён в структуре библиотеки STL.

Альтернативный подход к определению обобщённого программирования, который можно назвать **обобщённым программированием типов данных** (англ. *datatype generic programming*), был предложен Ричардом Бёрдом и Ламбертом

Меертенсом. В нём структуры типов данных являются параметрами обобщённых программ. Для этого в язык программирования вводится новый уровень абстракции, а именно параметризация по отношению к классам алгебр с переменной сигнатурой. Хотя теории обоих подходов не зависят от языка программирования, подход Массера — Степанова, делающий упор на анализ концепций, сделал C++ своей основной платформой, тогда как обобщённое программирование типов данных используют почти исключительно Haskell и его варианты.

Средства обобщённого программирования реализуются в языках программирования в виде тех или иных синтаксических средств, дающих возможность описывать данные (типы данных) и алгоритмы (процедуры, функции, методы), параметризуемые типами данных. У функции или типа данных явно описываются формальные параметры-типы. Это описание является обобщённым и в исходном виде непосредственно использовано быть не может.

В тех местах программы, где обобщённый тип или функция используется, программист должен явно указать фактический параметр-тип, конкретизирующий описание. Например, обобщённая процедура перестановки местами двух значений может иметь параметр-тип, определяющий тип значений, которые она меняет местами. Когда программисту нужно поменять местами два целых значения, он вызывает процедуру с параметром-типом «целое число» и двумя параметрами — целыми числами, когда две строки — с параметром-типом «строка» и двумя параметрами — строками. В случае с данными программист может, например, описать обобщённый тип «список» с параметром-типом, определяющим тип хранимых в списке значений. Тогда при описании реальных списков программист должен указать обобщённый тип и параметр-тип, получая, таким образом, любой желаемый список с помощью одного и того же описания.

Компилятор, встречая обращение к обобщённому типу или функции, выполняет необходимые процедуры статического контроля типов, оценивает возможность заданной конкретизации и при положительной оценке генерирует код, подставляя фактический параметр-тип на место формального параметра-типа в обобщённом описании. Естественно, что для успешного использования обобщённых описаний фактические типы-параметры должны удовлетворять определённым условиям. Если обобщённая функция сравнивает значения типа-параметра, любой

конкретный тип, использованный в ней, должен поддерживать операции сравнения, если присваивает значения типа-параметра переменным — конкретный тип должен обеспечивать корректное присваивание.

Обобщённое программирование в языках

C++

В языке C++ обобщённое программирование основывается на понятии «шаблон», обозначаемом ключевым словом **template**. Широко применяется в стандартной библиотеке C++ (см. STL), а также в сторонних библиотеках boost, Loki. Большой вклад в появление развитых средств обобщённого программирования в C++ внёс Александр Степанов.

В качестве примера приведём шаблон (обобщение) функции, возвращающей большее значение из двух.

```
// Описание шаблона функции
template <typename T> T max(T x, T y)
{
    if (x < y)
        return y;
    else
        return x;
}
...
// Применение функции, заданной шаблоном

int a = max(10,15);
...
double f = max(123.11, 123.12);
...
```

или шаблон(обобщение) класса связного списка:

```
template< class T >
 class List
 {
 /* ... */
```

```
 public:
   void Add( const T& Element );
   bool Find( const T& Element );
 /* ... */
 };
```

Java

Java предоставляет средства обобщённого программирования, синтаксически основанные на C++, начиная с версии J2SE 5.0. В этом языке имеются **generics** или «контейнеры типа T» — подмножество обобщённого программирования.

.NET

На платформе .NET средства обобщённого программирования появились в версии 2.0.

```
// Объявление обобщенного класса.
public class GenericList<T>
{
    void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        GenericList<int> list1 = new
GenericList<int>();
        GenericList<string> list2 = new
GenericList<string>();
        GenericList<ExampleClass> list3 = new
GenericList<ExampleClass>();
    }
}
```

Пример на C#

```
interface IPerson
{
  string GetFirstName();
```

```
  string GetLastName();
}

class Speaker
{
  public void SpeakTo<T>(T person) where T :
IPerson
  {
    string name = person.GetFirstName();
    this.say("Hello, " + name);
  }
}
```

D

Пример рекурсивной генерации на основе шаблонов D:

```
// http://digitalmars.com/d/2.0/template.html
template Foo(T, R...) // T - тип, R - набор типов
{
    void Foo(T t, R r)
    {
        writeln(t);
        static if (r.length)   // if more arguments
            Foo(r);            // do the rest of
the arguments
    }
}

void main()
{
    Foo(1, 'a', 6.8);
}

/+++++++++++++++
 prints:
 1
 a
 6.8
 +++++++++++++++/
```

Object Pascal

Поддержка обобщённого программирования компилятором Free Pascal появилась начиная с версии 2.2 в 2007 году. В Delphi — с октября 2008 года. Основа поддержки обобщённых классов сначала появилась в Delphi 2007 .NET в 2006 году, но она затрагивала только .NET Framework. Более полная поддержка обобщённого программирования была добавлена в Delphi 2009. Обобщённые классы также поддерживаются в Object Pascal в системе PascalABC.NET.

Пример на языке Nim.

```
import typetraits

proc getType[T](x: T): string =
  return x.type.name


echo getType(21)       # напечатает int
echo getType(21.12)    # напечатает float64
echo getType("string") # напечатает string
```

Интенсиональный полиморфизм

Интенсиональный полиморфизм (англ. *intensional polymorphism*) представляет собой технику оптимизирующей компиляции параметрического полиморфизма на основе сложного теоретико-ти́пового анализа, которая состоит в вычислениях над типами во время выполнения программы. Интенсиональный полиморфизм позволяет реализовывать бестеговую сборку мусора, необёрнутую (*unboxed*) передачу аргументов в функции и упакованные (оптимизированные по памяти) плоские структуры данных.

3.18. Мономорфизация

Мономорфизация (англ. *monomorphizing*) представляет собой технику оптимизирующей компиляции параметрического полиморфизма, которая заключается в порождении мономорфного экземпляра для *каждого* случая использования полиморфной функции или типа. Другими словами, параметрический полиморфизм на уровне исходного кода транслируется в ad hoc полиморфизм на уровне

целевой платформы. Мономорфизация повышает быстродействие (точнее, делает полиморфизм «бесплатным»), но вместе с тем может увеличивать размер выходного машинного кода.

Система типов Хиндли — Милнера

Вывод типов (англ. *type inference*) — в программировании возможность компилятора самому логически вывести тип значения у выражения. Впервые механизм вывода типов был представлен в языке ML, где компилятор всегда выводит наиболее общий полиморфный тип для всякого выражения. Это не только сокращает размер исходного кода и повышает его лаконичность, но и нередко повышает повторное использование кода[1].

Вывод типов характерен для функциональных языков программирования, хотя со временем эта возможность частично появилась и в объектно-ориентированных языках (C#, D, Visual Basic .NET, C++11, Vala, Go), где она ограничивается возможностью опустить тип идентификатора в определении с инициализацией (см. синтаксический сахар). Например:

```
var s = "Hello, world!";  // Тип переменной s (от string) выведен из инициализатора
```

Алгоритм Хиндли — Милнера

Алгоритм Хи́ндли — Ми́лнера — механизм вывода типов выражений, реализуемый в языках программирования, основанных на системе типов Хиндли — Милнера (англ.), таких как ML (первый язык этого семейства), Standard ML, OCaml, Haskell, F#, Fortress и Boo. Язык Nemerle использует этот алгоритм с рядом необходимых изменений.

Механизм вывода типов основан на возможности автоматически полностью или частично выводить тип выражения, полученного при помощи вычисления некоторого выражения. Так как этот процесс систематически производится во время трансляции программы, транслятор часто может вывести тип переменной или функции без явного указания типов этих объектов. Во многих случаях можно опускать явные декларации типов — это можно делать для достаточно простых объектов, либо для языков с простым синтаксисом. Например,

в языке Haskell реализован достаточно мощный механизм вывода типов, поэтому указание типов функций в этом языке программирования не требуется. Программист может указать тип функции явно для того, чтобы ограничить её использование только для конкретных типов данных, либо для более структурированного оформления исходного кода.

Для того, чтобы получить информацию для корректного вывода типа выражения в условиях отсутствия явной декларации типа этого выражения, транслятор либо собирает такую информацию из явных деклараций типов подвыражений (переменных, функций), входящих в изучаемое выражение, либо использует неявную информацию о типах атомарных значений. Такой алгоритм не всегда помогает определить тип выражения, особенно в случаях использования функций высших порядков и параметрического полиморфизма достаточно сложной природы. Поэтому в сложных случаях, когда есть необходимость избежать неоднозначностей, рекомендуется явно указывать тип выражений.

Сама модель типизации основана на алгоритме вывода типов выражений, который имеет своим источником механизм получения типов выражений, используемый в типизированном λ-исчислении, который был предложен в 1958 г. Х. Карри и Р. Фейсом. Далее уже́ Роджер Хиндли в 1969 г. расширил сам алгоритм и доказал, что он выводит наиболее общий тип выражения. В 1978 г. Робин Милнер независимо от Р. Хиндли доказал свойства эквивалентного алгоритма. И, наконец, в 1985 г. Луис Дамас окончательно показал, что алгоритм Милнера является законченным и может использоваться для полиморфных типов. В связи с этим алгоритм Хиндли — Милнера иногда называют также и ***алгоритмом Дамаса — Милнера***.

Система типов определяется в модели Хиндли — Милнера следующим образом:

1. Примитивные типы *v являются* типами выражений.
2. Параметрические переменные типов α являются типами выражений.
3. Если σ_1 и σ_2 — типы выражений, то тип $\sigma_1 \to \sigma_2$ является типом выражений.
4. Символ $\perp$ является типом выражений.

Выражения, типы которых вычисляются, определяются довольно стандартным образом:

1. Константы являются выражениями.
2. Переменные являются выражениями.
3. Если e_1 и e_2 — выражения, то $(e_1 e_2)$ — выражение.
4. Если v — переменная, а e — выражение, то $\lambda v,e$ — выражение.

Говорят, что тип σ_1 является экземпляром типа σ_2, когда имеется некое преобразование ρ такое, что:

$$\sigma_1 = \rho(\sigma_2)$$

При этом обычно полагается, что на преобразования типов ρ накладываются ограничения, заключающиеся в том, что:

1. $\rho(\sigma_1 \to \sigma_2) = \rho(\sigma_1) \to \rho(\sigma_2)$
2. $\rho(v) = v$

Сам алгоритм вывода типов состоит из двух шагов — генерация системы уравнений и последующее решение этих уравнений.

Построение системы уравнений

Построение системы уравнений основано на следующих правилах:

1. $f\,\Gamma v = \tau$ — в том случае, если связывание $v{:}\tau$ находится в Γ.
2. $f\,\Gamma(ef) = \tau$— в том случае, если $\tau_1 = \tau_2 \to \tau$, где $\tau_1 = f\,\Gamma e$ и $\tau_2 = f\,\Gamma f$.
3. $f\,\Gamma(\lambda v,e) = \tau \to \tau_e$ — в том случае, если $\tau_e = f\,\Gamma' e$ и Γ' является расширением Γ связыванием $v{:}\tau$.

В этих правилах под символом Γ понимается набор связываний переменных с их типами:

$$\Gamma = v_1{:}A_1,\ v_2{:}A_{2,\ldots},\ v_n{:}A_n$$

Решение системы уравнений

Решение построенной системы уравнений основано на алгоритме унификации. Это достаточно простой алгоритм. Имеется некоторая функция *u*, которая принимает на вход уравнение типов и возвращает подстановку, которая делает левую и правую части уравнения одинаковыми ("унифицирует" их). Подстановка — это просто проекция переменных типов на сами типы. Такие подстановки могут вычисляться различными способами, которые зависят от конкретной реализации алгоритма Хиндли — Милнера.

В классическом варианте система типов Хиндли — Милнера (а также просто «Хиндли — Милнер» или «X-M», англ. *HM*), положенная в основу языка ML, представляет собой подмножество Системы F, ограниченное предикативным пренексным полиморфизмом с целью обеспечения возможности автоматического выведения типов, для чего в состав Хиндли — Милнера традиционно также включался так называемый «Алгоритм W», разработанный Робином Милнером. В дальнейшем Хиндли — Милнер был существенно развит по нескольким направлениям, наиболее заметным из которых являются классы типов, дальнейшим обобщением которых стали квалифицированные типы.

Многие реализации X-M являются улучшенной версией системы, представляя собой «систему главной типизации» (англ. *principal typing scheme*), которая за один проход с почти линейной сложностью одновременно выводит наиболее общие полиморфные типы для каждого выражения и строго проверяет их согласование.

С момента своего появления система типов Хиндли — Милнера была расширена по нескольким направлениям. Одним из наиболее известных расширений является поддержка ad hoc полиморфизма посредством классов типов.

Автоматическое выведение типов было сочтено *необходимостью* при первоначальной разработке языка ML в качестве интерактивной системы доказательства теорем «Logic for Computable Functions», из-за чего и были наложены соответствующие ограничения. В дальнейшем на основе ML был разработан целый ряд эффективно компилируемых языков общего назначения, ориентированных на крупномасштабное программирование, а в этом случае необходимость поддержки

выведения типов резко снижается, так как интерфейсы модулей в промышленной практике в любом случае необходимо явно аннотировать типами. Поэтому было предложено множество вариантов расширения Хиндли — Милнера, отказывающихся от выведения типов ради расширения возможностей, вплоть до поддержки полной Системы F с импредикативным полиморфизмом, таких как язык модулей высшего порядка, изначально основанный на явном аннотировании типов модулей и имеющий множество расширений и диалектов, а также расширения языка Haskell (`Rank2Types`, `RankNTypes` и `ImpredicativeTypes`).

Компилятор MLton языка Standard ML осуществляет мономорфизацию, но за счёт других применимых к Standard ML оптимизаций результирующее увеличение выходного кода не превышает 30 %.

Си и C++

В языке Си функции не являются объектами первого класса, но возможно определение указателей на функции, что позволяет строить функции высших порядков. Также доступен небезопасный параметрический полиморфизм за счёт явной передачи необходимых свойств типа через бестиповое подмножество языка, представленное нетипизированным указателем `void*` (называемым в сообществе языка «*обобщённым указателем*» (англ. *generic pointer*). Назначение и удаление информации о типе при приведении типа к `void*` и обратно не является ad hoc полиморфизмом, так как не меняет представление указателя, однако, его записывают явно для обхода системы типов компилятора.

Например, стандартная функция `qsort` способна обрабатывать массивы элементов любого типа, для которого определена функция сравнения.

```
struct segment { int start; int end; };

int seg_cmpr( struct segment *a, struct segment *b
)
{ return abs( a->end - a->start ) - abs( b->end -
b->start ); }
```

```
int str_cmpr( char **a, char **b )
{ return strcmp( *a, *b ); }

struct segment segs[] = { {2,5}, {4,3}, {9,3},
{6,8} };
char* strs[] = { "three", "one", "two", "five",
"four" };

main()
{
    qsort( strs, sizeof(strs)/sizeof(char*),
sizeof(char*),
                (int (*)(void*,void*))str_cmpr );

    qsort( segs, sizeof(segs)/sizeof(struct
segment), sizeof(struct segment),
                (int (*)(void*,void*))seg_cmpr );
    ...
}
```

Тем не менее, в Си возможно идиоматическое воспроизведение типизированного параметрического полиморфизма без использования `void*`.

Язык C++ предоставляет подсистему шаблонов, использование которых внешне похоже на параметрический полиморфизм, но семантически реализуется сочетанием ad hoc-механизмов:

```
template <typename T> T max(T x, T y)
{
    if (x < y)
        return y;
    else
        return x;
}

int main()
{
    int a = max(10,15);
    double f = max(123.11, 123.12);
    ...
```

```
}
```

Мономорфизация при компиляции шаблонов C++ является *неизбежной*, так как в системе типов языка отсутствует поддержка полиморфизма — полиморфный язык здесь является статической надстройкой над мономорфным ядром языка. Это приводит к кратному увеличению объёма получаемого машинного кода, что получило известность как «*раздувание кода*».

3.19. Основные разновидности полиморфизма

3.19.1. Ad hoc полиморфизм

Ad hoc полиморфизм (в русской литературе чаще всего переводится как «*специальный полиморфизм*» или «*специализированный полиморфизм*», хотя оба варианта не всегда верны) поддерживается во многих языках посредством перегрузки функций и методов, а в слабо типизированных — также посредством приведения типов.

В следующем примере (язык Pascal) функции `Add` выглядят как реализующие одну и ту же функциональность над разными типами, но компилятор определяет их как две совершенно разные функции.

```
program Adhoc;

function Add( x, y : Integer ) : Integer;
begin
    Add := x + y
end;

function Add( s, t : String ) : String;
begin
    Add := Concat( s, t )
end;

begin
    Writeln(Add(1, 2));
    Writeln(Add('Hello, ', 'World!'));
end.
```

В динамически типизируемых языках ситуация может быть более сложной, так как выбор требуемой функции для вызова может быть осуществлён только во время исполнения программы.

Перегрузка представляет собой *синтаксический* механизм, позволяющий по единому идентификатору вызывать разные функции.

Приведение типов представляет собой *семантический* механизм, осуществляемый для преобразования фактического типа аргумента к ожидаемому функцией, при отсутствии которого произошла бы ошибка типизации. То есть, при вызове функции с приведением типа происходит исполнение различного кода для различных типов (предваряющего вызов самой функции).

3.19.2. Параметрический полиморфизм

Параметрический полиморфизм позволяет определять функцию или тип данных обобщённо, так что значения обрабатываются идентично вне зависимости от их типа. Параметрически полиморфная функция использует аргументы на основе поведения, а не значения, апеллируя лишь к необходимым ей свойствам аргументов, что делает её применимой в любом контексте, где тип объекта удовлетворяет заданным требованиям поведения.

Развитые системы типов (такие как Хиндли — Милнер) предоставляют механизмы для определения полиморфных типов, что делает использование полиморфных функций более удобным и обеспечивает статическую типобезопасность. Такие системы являются системами типов второго порядка, добавляющими к системам типов первого порядка (используемым в большинстве процедурных языков) параметризацию типов (посредством ти́повой переменной) и абстракцию типов (посредством экзистенциальной квантификации над ними). В системах типов второго порядка нет непосредственной необходимости в поддержке примитивных типов, так как они могут быть выражены посредством более развитых средств.[15]

Классическим примером полиморфного типа служит *список элементов произвольного типа*, для которого во многих языках, типизируемых по Хиндли — Милнеру (большинство статически типизируемых функциональных языков программирования), предоставляется синтаксический сахар. Следующий пример демонстрирует

определение нового алгебраического типа «параметрически полиморфный список» и двух функций над ним:

```
data List a = Nil | Cons a (List a)

length :: List a -> Integer
length Nil = 0
length (Cons x xs) = 1 + length xs

map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

При подстановке в ти́повую переменную `a` конкретных типов `Int`, `String` и т. д. будут построены, соответственно, конкретные типы `List Int`, `List String` и т. д. Эти конкретные типы, в свою очередь, снова могут быть подставлены в эту ти́повую переменную, порождая типы `List List String`, `List (Int, List String)` и т. д. При этом для всех объектов всех построенных типов будет использоваться *одна и та же* физическая реализация функций `length` и `map`.

Ограниченные формы параметрического полиморфизма доступны также в некоторых императивных (в частности, объектно-ориентированных) языках программирования, где для его обозначения обычно используется термин «обобщённое программирование». В частности, в языке Си нетипизированный параметрический полиморфизм традиционно обеспечивается за счёт использования нетипизированного указателя `void*`, хотя возможна и типизированная форма. Использование шаблонов C++ внешне похоже на параметрический полиморфизм, но семантически реализуется сочетанием ad hoc-механизмов; в сообществе C++ его называют «*статическим полиморфизмом*» (для противопоставления «*динамическому полиморфизму*»).

Параметричность

Формализация параметрического полиморфизма ведёт к понятию параметричности, состоящему в возможности предсказывать поведение программ, глядя только на их типы. Например, если

функция имеет тип «`forall a. a -> a`», то без каких-либо дополняющих язык внешних средств можно доказать, что она может быть *только* тождественной. Поэтому параметричность часто сопровождается лозунгом «*теоремы забесплатно*» (англ. *theorems for free*).

Важным следствием этого является также *независимость представлений* (англ. *representation independence*), означающее, что функции над абстрактным типом нечувствительны к его структуре, и различные реализации этого типа могут свободно подменять друг друга (даже в рамках одной программы), не влияя на поведение этих функций.

Наиболее развитые параметрически полиморфные системы (см. лямбда-куб, зависимые типы) доводят идею параметричности до возможности полного доказательства корректности программ: они позволяют записывать программы, которые *верны по построению*, так что прохождение проверки согласования типов само по себе даёт *гарантию*, что поведение программы соответствует ожидаемому.

3.19.3. Полиморфизм записей и вариантов

Отдельную проблему представляет распространение параметрического полиморфизма на агрегатные типы: размеченные произведения типов (традиционно называемые записями) и размеченные суммы типов (также известные как вариантные типы). Различные «исчисления записей» (англ. *record calculi*) служат формальной базой для объектно-ориентированного и модульного программирования.

```
val r = {a = 1, b = true, c = "hello"}
val {a = n, ... = r1} = r
val r2 = {d = 3.14, ... = r1}
```

Многоточие означает некоторый ***ряд*** типизированных полей, т.е. абстракцию кода от конкретных типов записей, которые могли бы здесь обрабатываться (причём «длина» этого ряда также может варьироваться). Компиляция полиморфного обращения к полям, которые могут располагаться в разном порядке в разных записях, представляет сложную проблему, как с точки зрения контроля безопасности операций на уровне языка, так и с точки зрения быстродействия на уровне машинного кода. Наивным решением может

быть динамический поиск по словарю при каждом обращении (и скриптовые языки его применяют), однако, очевидно, что это чрезвычайно неэффективно. Эта проблема активно исследуется на протяжении уже нескольких десятилетий; построено множество теоретических моделей и практичных систем на их основе, различающихся по выразительной мощности и метатеоретической сложности. Важнейшими достижениями в этой области считаются **рядный полиморфизм**, предложенный Митчелом Вандом (Mitchell Wand (англ.)), и **полиморфное исчисление записей**, построенное Ацуси Охори (англ. *Atsushi Ohori*). Более распространённой, но отстающей по многим характеристикам моделью является подтипизация на записях.

Поддержка полиморфизма записей в той или иной форме может открывать в полиморфном языке такие возможности, как сообщения высшего порядка (наиболее мощную форму объектно-ориентированного программирования), бесшовное встраивание операций над элементами баз данных (SQL) в код на языке общего назначения, и др., вплоть до *расширяемого программирования* (т.е. программирования, свободного от проблемы выражения), гарантии отсутствия необработанных исключений в программах и определённых форм метапрограммирования.

3.19.4. Полиморфизм подтипов

Полиморфизм включения (*inclusion polymorphism*) является обобщённой формализацией подтипизации и наследования. Эти понятия не следует путать: подтипы определяют отношения на уровне интерфейсов, тогда как наследование определяет отношения на уровне реализации.

Подтипизация (*subtyping*), или полиморфизм подтипов (*subtype polymorphism*), означает, что поведение параметрически полиморфной функции ограничивается множеством типов, ограниченных в иерархию «супертип — подтип». Например, если имеются типы `Number`, `Rational` и `Integer`, ограниченные отношениями `Number :> Rational` и `Number :> Integer`, то функция, определённая на типе `Number`, также сможет принять на вход аргументы типов `Integer` или `Rational`, и её поведение будет идентичным. Действительный тип объекта может быть скрыт как «чёрный ящик», и предоставляться лишь по запросу идентификации объекта. На самом

деле, если тип `Number` является абстрактным, то конкретного объекта этого типа даже не может существовать (см. абстрактный класс, но не следует путать с абстрактным типом данных). Данная иерархия типов известна (особенно в контексте языка Scheme) как числовая башня, и обычно содержит большее количество типов.

Идея подтипов мотивируется увеличением множества типов, которые могут обрабатываться уже написанными функциями, и за счёт этого повышением коэффициента повторного использования кода в условиях сильной типизации (то есть увеличением множества типизируемых программ). Это обеспечивается **правилом включения** (*subsumption rule*): «*если выражение* `e` *принадлежит к типу* `t'` *в контексте типизации* `Γ`*, и выполняется* `t'<:t`*, то* `e` *принадлежит также и к типу* `t`».

Отношения подтипизации возможны на самых разных категориях типов: примитивных типах (как `Integer <: Number`), типах-суммах, типах-произведениях, функциональных типах и др. Более того, Лука Карделли предложил концепцию степенны́х родо́в (*«power» kinds*) для описания подтипизации: *степенью* данного типа в слое родо́в (*power-kind of the type*) он назвал род всех его подтипов.

Особое место в информатике занимает подтипизация на записях.

3.19.5. Подтипизация на записях

Подтипизация на записях, также известная как *включение* или *вхождение* (***subsumption*** — см. принцип подстановки Барбары Лисков), служит наиболее распространённым теоретическим обоснованием объектно-ориентированного программирования (ООП) (но не единственным — см. #полиморфизм записей и вариантов).

Принцип подстановки Барбары Лисков (англ. *Liskov Substitution Principle*, ***LSP***) в объектно-ориентированном программировании является специфичным определением *подтипа*, предложенным Барбарой Лисков в 1987 году на конференции в основном докладе под названием *Абстракция данных и иерархия*.

В последующей статье Лисков кратко сформулировала свой принцип следующим образом:

> *Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T.*

Роберт С. Мартин определил этот принцип так:

> *Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.*

Таким образом, идея Лисков о «подтипе» даёт определение понятия **замещения** — если S является подтипом T, тогда объекты типа T в программе могут быть замещены объектами типа S без каких-либо изменений желательных свойств этой программы (например, корректность).

Этот принцип является *важнейшим* критерием для оценки качества принимаемых решений при построении иерархий наследования. Сформулировать его можно в виде простого правила: тип S будет подтипом T *тогда и только тогда*, когда каждому объекту oS типа S соответствует некий объект oT типа T таким образом, что для всех программ P, реализованных в терминах T, поведение P не будет меняться, если oT заменить на oS.

Более простыми словами можно сказать, что поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.

Саттер и Александреску в своём руководстве по использованию C++ для выражения этого принципа также используют фразу «подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс». По мнению данных авторов, публичное наследование в C++ можно употреблять только тогда, когда оно удовлетворяет принципу Лисков. Приватное наследование, по их же мнению, дозволено использовать для доступа к protected части базы и перекрытия виртуальных методов. В любом же ином случае, то есть для всего лишь повторного использования кода из базы, наследование применять нельзя.

Основания: использование публичного наследования для повторного использования кода приводит к тому, что внешний мир начинает считать класс Derived разновидностью класса Base, и возможно появление кода, явно использующего этот факт. Это сильно сужает простор для манёвра архитектора в дальнейшем поддержании и рефакторинге класса Derived.

3.19. 6. Проектирование по контракту

Принцип подстановки (замещения) Лисков имеет близкое отношение к методологии контрактного программирования, и ведёт к некоторым ограничениям на то, как контракты могут взаимодействовать с наследованием:

- Предусловия не могут быть усилены в подклассе.
- Постусловия не могут быть ослаблены в подклассе.
- Исторические ограничения ("правило истории") - подкласс не должен создавать новых мутаторов свойств базового класса. Если базовый класс не предусматривал методов для изменения определенных в нем свойств, подтип этого класса так же не должен создавать таких методов. Иными словами, неизменяемые данные базового класса не должны быть изменяемыми в подклассе. Данная концепция, представленная Лисков и Винг, являлась новаторской для теории программной архитектуры.

Также принцип LSP подразумевает, что методы подкласса не могут генерировать никаких дополнительных исключений, кроме тех, которые сами являются подклассами исключений, генерируемых методами надкласса. См. Ковариантность и контравариантность и типы данных.

Функция, использующая иерархию классов с нарушениями принципа Лисков, помимо оперирования ссылкой на базовый класс, оказывается также вынуждена знать и о подклассе. Подобная функция нарушает принцип открытости/закрытости, поскольку она требует модификации в случае появления в системе новых производных классов.

В данном контексте принцип подстановки можно переформулировать следующим образом:

> *Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, не зная об этом.*

Принцип Барбары Лисков заставляет задуматься о том, что такое «декларация типа» в терминах объектно-ориентированного языка программирования, который мы используем. Достаточно ли нам описать интерфейс объекта с помощью обычного абстрактного класса со списком методов, типами параметров и возвращаемого значения? Каким образом мы можем декларировать требования к значениям параметров метода и свойства, которыми будет обладать возвращаемое значение? Как нам описать исключения, которые может сгенерировать метод во время выполнения? Как нам описать изменение состояния объекта на разных этапах его жизненного цикла?

Задавая себе эти вопросы и находя ответы, можно спроектировать систему, которая действительно будет удовлетворять принципу подстановки Барбары Лисков.

Мартин Абади (*Martín Abadi*) и Лука Карделли формализовали подтипизацию на записях посредством ограниченной квантификации над параметрически полиморфными типами; при этом параметр типа задаётся неявно.

В подтипизации на записях выделяются две разновидности: в ширину и в глубину.

Подтипизация в ширину (*width subtyping*) подразумевает добавление в запись новых полей. Например:

```
type Object  = { age: Int }
type Vehicle = { age: Int; speed: Int }
type Bike    = { age: Int; speed: Int; gear: Int; }
type Machine = { age: Int; fuel: String }
```

С одной стороны, можно записать отношения подтипизации `Vehicle <: Object`, `Bike <: Vehicle` (а поскольку подтипизация транзитивна, то и `Bike <: Object`) и `Machine <: Object`. С другой стороны, можно говорить, что типы `Vehicle`, `Bike` и

`Machine` *включают* (наследуют) все свойства типа `Object`. (Здесь подразумевается структурная семантика системы типов.)

Поскольку интуитивно *тип* зачастую рассматривается как *множество значений*, то увеличение количества полей в *подтипе* может выглядеть контр-интуитивно с точки зрения теории множеств. В действительности, типы — это *не* множества, они предназначены для верификации поведения программ, и идея подтипизации состоит в том, что подтип обладает *по меньшей мере* свойствами своего супертипа, и таким образом, способен эмулировать его *как минимум* там, где ожидается объект супертипа. Или иначе: супертип определяет *минимальный* набор свойств для множества объектов, и тогда тип, обладающий этими свойствами и, возможно, какими-то другими, формирует подмножество этого множества, а следовательно, является его подтипом.

Отношения подтипов в терминах множеств выглядят более интуитивно в случае с вариантными типами:

```
type Day     = mon | tue | wed | thu | fri | sat |
sun
type Workday = mon | tue | wed | thu | fri
type WeekEnd = sat | sun
```

Здесь `Workday <: Day` и `WeekEnd <: Day`.

Именование полей позволяет абстрагироваться от порядка их следования в записях (в отличие от кортежей), что даёт возможность строить произвольные направленные ациклические графы наследования, формализуя множественное наследование:

```
type Car = { age: Int; speed: Int; fuel: String }
```

Теперь `Car <: Vehicle` и одновременно `Car <: Machine`. Очевидно также, что `Car <: Object` (см. ромбовидное наследование).

Мно́жественное насле́дование — свойство, поддерживаемое частью объектно-ориентированных языков программирования, когда класс может иметь более одного суперкласса (непосредственного класса-

родителя), интерфейсы поддерживают множественное наследование во многих языках программирования. Эта концепция является расширением «простого (или одиночного) наследования» (англ. *single inheritance*), при котором класс может наследоваться только от одного суперкласса.

В список языков, поддерживающих множественное наследование, входят: Io, Eiffel, C++, Dylan, Python, некоторые реализации классов JavaScript (например, dojo.declare), Perl, Curl, Common Lisp (благодаря CLOS), OCaml, Tcl (благодаря Incremental Tcl)[1], а также Object REXX (за счёт использования классов-примесей).

Множественное наследование позволяет классу перенимать функциональность у множества других классов, как например, класс `StudentMusician` может наследовать от класса `Person`, класса `Musician` и класса `Worker`, что сокращённо можно написать:

```
StudentMusician : Person, Musician, Worker.
```

Неопределённость при множественном наследовании, как в примере выше, возникает если, к примеру, класс `Musician` наследует от классов `Person` и `Worker`, а класс `Worker`, в свою очередь, наследует от `Person`; подобная ситуация называется ромбовидным наследованием. Таким образом, у нас получаются следующие правила:

```
Worker          :  Person
Musician        :  Person, Worker
StudentMusician :  Person, Musician, Worker
```

Если компилятор просматривает класс StudentMusician, то ему необходимо знать, нужно ли объединять возможности классов или они должны быть раздельными. Например, логично будет присоединить «Age» (возраст) класса Person к классу StudentMusician. Возраст человека не меняется, если вы рассматриваете его как Person (человек), Worker (рабочий) или Musician (музыкант). Однако, будет довольно логичным отделить свойство «Name» (имя) в классах Person и Musician, если они используют сценический псевдоним, отличающийся от настоящего имени. Варианты объединения и разделения вполне корректны для каждого из собственных контекстов и только

программист знает, какой вариант является правильным для проектируемого класса.

Языки обладают различными способами разрешения таких проблем вложенного наследования, а именно:

- Eiffel предоставляет программисту возможность явным образом объединить или разделить унаследованные элементы от суперклассов. Eiffel автоматически объединит элементы, если у них будет одинаковое имя и реализация. Автор класса имеет возможность переименовать наследуемые элементы для их разделения. Кроме того, Eiffel позволяет явным образом выполнять *повторное наследование* вида `A: B, B`.
- C++ требует, чтобы программист указал, элемент какого из родительских классов должен использоваться, то есть «Worker::Person.Age». C++ не поддерживает явно повторяемое наследование, так как отсутствует способ определить какой именно суперкласс следует использовать (смотри *критику*). C++, также, допускает создание единственного экземпляра множественного класса благодаря механизму виртуального наследования (например, «`Worker::Person`» и «`Musician::Person`» будут ссылаться на один и тот же объект).
- Perl использует список классов для наследования в указанном порядке. Компилятор использует первый метод, который он находит при глубинном поиске в списке суперклассов или использовании C3-линеаризации иерархии классов. Различные расширения обеспечивают альтернативные схемы композиции классов.
- Python (см. наследование и множественное наследование в Python) имеет синтаксическую поддержку для множественного наследования, а порядок базовых классов определяется алгоритмом C3-линеаризации.
- Common Lisp Object System предусматривает полный контроль методов комбинации со стороны программиста, а если этого не достаточно, то метаобъектный протокол (Metaobject Protocol) дает программисту возможность *модифицировать* наследование, динамическое управление, реализация класса и другие внутренние механизмы без опасения повлиять на стабильность системы.

- Logtalk поддерживает оба интерфейса и реализацию мультинаследования, предусматривая объявление метода *алиасов*, поддерживающего как переименование, так и доступ к методам, которые могут оказаться недоступными, благодаря механизму разрешения конфликтов.
- Curl допускает только такие классы, которые явным образом отмечены как *доступные* для повторного наследования. Доступные классы должны определять *вторичный конструктор* для каждого обычного конструктора класса. Сначала вызывается обычный конструктор, статус доступного класса инициализируется за счет конструктора подкласса, а вторичный конструктор вызывается для всех остальных подклассов.
- Ocaml выбирает последнее совпавшее определение в списке наследования классов для определения метода реализации, используемого в случае неопределенности. Для переопределения поведения по умолчанию нужно просто указать метод, вызываемый при определении предпочитаемого класса.
- Tcl допускает существование множества родительских классов — их последовательность влияет на разрешение имен членов класса.
- Delphi с версии 2007 позволяет частично реализовать множественное наследование с помощью помощников классов (Class Helpers).

Smalltalk, C#, Objective-C, Java, Nemerle и PHP не допускают множественного наследования, что позволяет избежать многих неопределенностей. Однако, они, кроме Smalltalk, позволяют классам реализовать множественные интерфейсы. Кроме того, PHP и Ruby позволяют эмулировать множественное наследование за счет использования примесей (traits в PHP и mixins в Ruby), которые, как и интерфейсы, полноценными классами не являются. Множественное наследование интерфейсов позволяет расширить ограниченные возможности.

3.19.7. Ромбовидное наследование

Ромбовидное наследование (англ. *diamond inheritance*) — ситуация в объектно-ориентированных языках программирования с поддержкой

множественного наследования, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C. При этой схеме наследования может возникнуть неоднозначность: если метод класса D вызывает метод, определенный в классе A (и этот метод не был переопределен в классе D), а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C?

Например, в области разработки графических интерфейсов класс `Button` («Кнопка») может одновременно наследовать от класса `Rectangle` («Прямоугольник», для внешнего вида) и от класса `Clickable` («Доступен для кликанья мышкой», для реализации функциональности/обработки ввода), а `Rectangle` и `Clickable` наследуют от класса `Object` («Объект»). Если вызвать метод `equals` («Равно») для объекта `Button`, и в классе `Button` не окажется такого метода, но в классе `Object` будет присутствовать метод `equals` по-своему переопределенный как в классе `Rectangle`, так и в `Clickable`, то какой из методов должен быть вызван?

Проблема ромба (англ. *diamond problem*) получила своё название благодаря очертаниям диаграммы наследования классов в этой ситуации. В данной статье, класс A обозначается в виде вершины, классы B и C по отдельности указываются ниже, а D соединяется с обоими в самом низу, образуя ромб.

Решения

Различные языки программирования решают проблему ромбовидного наследования следующими способами:

- C++ по умолчанию не создает ромбовидного наследования: компилятор обрабатывает каждый путь наследования отдельно, в результате чего объект D будет на самом деле содержать два разных подобъекта A, и при использовании членов A потребуется указать путь наследования (`B::A` или `C::A`). Чтобы сгенерировать ромбовидную структуру наследования, необходимо воспользоваться виртуальным наследованием класса A на нескольких путях наследования: если оба наследования от A к B и от A к C помечаются спецификатором `virtual` (например, `class B : virtual`

`public A`), C++ специальным образом проследит за созданием только одного подобъекта `A`, и использование членов `A` будет работать корректно. Если виртуальное и невиртуальное наследования смешиваются, то получается один виртуальный подобъект `A` и по одному невиртуальному подобъекту `A` для каждого пути невиртуального наследования к `A`. При виртуальном вызове метода виртуального базового класса используется так называемое правило доминирования: компилятор запрещает виртуальный вызов метода, который был перегружен на нескольких путях наследования.

- Common Lisp пытается реализовать и разумное поведение по умолчанию, и возможность изменить его. По умолчанию выбирается метод с наиболее специфичными классами аргументов; затем, методы выбираются по порядку, в котором родительские классы указаны при определении подкласса. Однако программист вполне может изменить это поведение путём указания специального порядка разрешения методов или указания правила для объединения методов.
- Eiffel обрабатывает подобную ситуацию при помощи директив `select` и `rename`, и методы предка, которые используются в потомках, указываются явно. Это позволяет совместно использовать методы родительского класса в потомках или предоставлять им отдельную копию родительского класса.
- Perl и Io обрабатывают наследования через поиск в глубину в том порядке, который используется в определении класса. Класс `B` и его предки будут проверены перед классом `C` и его предками, так что метод в `A` будет унаследован от `B`; список разрешения — [`D`, `B`, `A`, `C`]. При этом в Perl данное поведение может быть изменено при помощи `mro` или других модулей для применения C3-линеаризации (как в Python) или других алгоритмов.
- В Python проблема ромба остро встала в версии 2.3 после введения классов с общим предком `object`; начиная с этой версии было решено создавать список разрешения при помощи C3-линеаризации. В случае ромба это означает поиск в глубину, начиная слева (`D`, `B`, `A`, `C`, `A`), а затем удаление из списка всех, кроме последнего включения каждого класса, который в списке повторяется. Следовательно, итоговый порядок разрешения выглядит так: [`D`, `B`, `C`, `A`].
- Scala список разрешения создается аналогично Python, но через поиск в глубину начиная справа. Следовательно,

предварительный список разрешения ромба — [D, C, A, B, A], а после удаления повторений — [D, C, B, A].

- JavaFX Script, начиная с версии 1.2, позволяет множественное наследование за счет применения примесей. В случае конфликта, компилятор запрещает прямое использование неопределенных переменных или функции. К каждому наследуемому члену по-прежнему будет возможен доступ за счет приведения объекта к нужной примеси, например, `(individual as Person).printInfo();`.

Прочие примеры

Языки, допускающие лишь простое наследование (как например, Ада, Objective-C, PHP, C#, Delphi/Free Pascal и Java), предусматривают множественное наследование интерфейсов (в Objective-C называемые протоколами). Интерфейсы по сути являются абстрактными базовыми классами, все методы которых также абстрактны, и где отсутствуют поля. Таким образом, проблема не возникает, так как всегда будет только одна реализация определенного метода или свойства, не допуская возникновения неопределенности.

Проблема ромба не ограничивается лишь наследованием. Она также возникает в таких языках, как Си и C++, когда заголовочные файлы A, B, C и D, а также отдельные предкомпилированные заголовки, созданные из B и C, подключаются (при помощи инструкции `#include`) один к другому по ромбовидной схеме, указанной вверху. Если эти два предкомпилированных заголовка объединяются, объявления в A дублируются, и директива защиты подключения `#ifndef` становится неэффективной. Также проблема обнаруживается при объединении стеков подпрограммного обеспечения; например, если A — это база данных, а B и C — кэши, то D может запросить как B, так и C подтвердить (COMMIT) выполнение транзакции, приводя к дублирующим вызовам подтверждений A.

Подтипизация в глубину (*depth subtyping*) подразумевает, что типы конкретных полей записи могут подменяться на их подтипы:

```
type Voyage   = { veh: Vehicle; date: Day }
type Sports   = { veh: Bike;    date: Day }
type Vacation = { veh: Car;     date: WeekEnd }
```

Из определений выше можно вывести, что `Sports <: Voyage` и `Vacation <: Voyage`.

3.19.8. Методы в подтипах записей

Полноценная поддержка объектно-ориентированного программирования предполагает включение в число полей записей также функций, обрабатывающих значения типов записей, которым они принадлежат. Такие функции традиционно называются «**методами**». Обобщённой моделью связывания записей с методами является матрица диспетчеризации (*dispatch matrix*); на практике большинство языков раскладывают её на вектора по строкам либо по столбцам — соответственно, реализуя либо организацию на основе классов (*class-based organisation*), либо организацию на основе методов (*method-based organisation*). При этом следует отличать *переопределение методов в подтипах* (*method overriding*) от *подтипизации функций* (*functional subtyping*). Переопределение методов не связывает их отношениями подтипизации на функциях, но позволяет изменять сигнатуры их типов. При этом возможны три варианта: инвариантное, ковариантное и контравариантное переопределение, и два последних используют подтипизацию своих параметров (подробнее см. ковариантность и контравариантность). Исчисление Абади — Карделли рассматривает только инвариантные методы, что необходимо для доказательства безопасности.

Многие объектно-ориентированные языки предусматривают встроенный механизм связывания функций в методы, реализуя таким образом организацию программ на основе классов. Языки, рассматривающие функции как объекты первого класса и типизирующие их (см. функции первого класса, функциональный тип — не путать с типом возвращаемого значения функции), позволяют произвольно выстраивать организацию на основе методов, что обеспечивает возможность производить объектно-ориентированное проектирование без прямой поддержки со стороны языка. Некоторые языки (например, OCaml) поддерживают обе возможности.

Языки с системами типов, основанными на формальной теории подтипизации (OCaml, проект successor ML), рассматривают системы объектов и системы классов независимо. Это значит, что с объектом связывается прежде всего *объектный тип*, и лишь при явном указании объектный тип связывается с неким классом. При этом

диспетчеризация осуществляется на уровне объекта, а значит, в таких языках два объекта, относящиеся к одному классу, вообще говоря, могут иметь различный набор методов. Вместе с формально определённой семантикой множественного наследования это даёт непосредственную всестороннюю поддержку примесей.

CLOS реализует матрицу диспетчеризации посредством мультиметодов, то есть динамически диспетчеризуемых методов, полиморфных сразу по нескольким аргументам.

Некоторые языки используют своеобразные ad hoc-решения. Например, система типов языка C++ предусматривает автоматическое приведение типов (то есть является слабой), не полиморфная, эмулирует выделение подтипов через манифестное наследование с инвариантными сигнатурами методов и не поддерживает абстракцию типов (не путать с сокрытием полей). Наследование в C++ реализуется набором ad hoc-механизмов, однако, его использование называется в сообществе языка «полиморфизмом» (а сокрытие полей — «абстракцией»). Имеется возможность управлять графом наследования: ромбовидное наследование в C++ называется «*виртуальным наследованием*» и задаётся явным атрибутом `virtual`; по умолчанию же осуществляется дублирование унаследованных полей с доступом к ним по квалифицированному имени. Использование такого языка может быть небезопасно — нельзя гарантировать устойчивость программ (язык называется безопасным, если программы на нём, которые могут быть приняты компилятором как правильно построенные, в динамике никогда не выйдут за рамки допустимого поведения).

Подтипизация высшего порядка

«*Система* $F^{\omega}_{<:}$» (читается «*F-саб-омега*») является расширением Системы F (не представленным в лямбда-кубе), формализующим ограниченную квантификацию над ти́повыми операторами, распространяя отношения подтипизации с рода $*$ на типы высших родо́в. Существует несколько вариантов системы $F^{\omega}_{<:}$, различающихся по выразительной мощности и метатеоретической сложности, но большинство основных идей заложил Лука Карделли.

3.20. Сочетание разновидностей полиморфизма

3.20.1. Классы типов

Класс типов реализует единую независимую таблицу виртуальных методов для множества (универсально квантифицированных) типов. Этим классы типов отличаются от классов в объектно-ориентированном программировании, где всякий объект всякого (ограниченно квантифицированного) типа сопровождается указателем на таблицу виртуальных методов. Классы типов являются не типами, но категориями типов; их экземпляры представляют собой не значения, а типы.

Например:

```
class Num a where
   (+), (*) :: a -> a -> a
   negate :: a -> a
```

Это объявление читается так: «*Тип* `a` *принадлежит классу* `Num`*, если на нём определены функции* `(+)`, `(*)` *и* `negate`*, с заданными сигнатурами*».

```
instance Num Int where
   (+) = addInt
   (*) = mulInt
   negate = negInt

instance Num Float where
   (+) = addFloat
   (*) = mulFloat
   negate = negFloat
```

Первое объявление читается так: «*Существуют функции* `(+)`, `(*)` *и* `negate` *соответствующих сигнатур, которые определены над типом* `Int`». Аналогично читается второе утверждение.

Теперь можно корректно типизировать следующие функции (причём выведение типов разрешимо):

```
square :: Num a => a -> a
square x = x * x

squares3 :: Num a, Num b, Num c => (a, b, c) -> (a,
b, c)
squares3 (x, y, z) = (square x, square y, square z)
```

Поскольку операция умножения реализуется физически различным образом для целых и чисел с плавающей запятой, в отсутствии классов типов уже здесь потребовались бы две перегруженные функции `square` и восемь перегруженных функций `squares3`, а в реальных программах со сложными структурами данных дублирующегося кода оказывается намного больше. В объектно-ориентированном программировании проблемы такого рода решаются посредством динамической диспетчеризации, с соответствующими накладными расходами. Класс типов осуществляет диспетчеризацию статически, сводя параметрический и ad hoc полиморфизм в единую модель. С точки зрения параметрического полиморфизма, класс типов имеет параметр (переменную типа), пробегающий множество типов. С точки зрения ad hoc полиморфизма, это множество не только дискретно, но и задано явным образом до уровня реализации. Проще говоря, сигнатура `square :: Num a => a -> a` означает, что функция параметрически полиморфна, но *спектр типов* её параметра ограничен лишь теми типами, что принадлежат к классу типов `Num`. Благодаря этому, функция типизируется единственным образом, несмотря на обращение к перегруженной функции из её тела.

Встроенная поддержка классов типов была впервые реализована в языке Haskell, но они также могут быть введены в любой параметрически полиморфный язык путём простого препроцессинга[11], а также реализованы идиоматически (см., например, язык модулей ML#Реализация альтернативных моделей). Однако, непосредственная поддержка может упрощать автоматическое рассуждение о смысле программ.

Типы, допускающие проверку на равенство (*equality types*) в Haskell реализуются как инстансы класса типов `Eq` (обобщая переменные типа, допускающего проверку на равенство (*equality type variables*) из Standard ML):

```
(==) :: Eq a => a -> a -> Bool
```

Для снижения рутинного кодирования некоторых часто очевидно необходимых свойств пользовательских типов в Haskell дополнительно предусмотрен синтаксический сахар — конструкция `deriving`, допустимая для ограниченного набора стандартных классов типов (таких как `Eq`). (В русскоязычном сообществе её использование нередко путается с понятием «наследования» из-за особенностей перевода слова «*derive*».)

3.20.2. Обобщённые алгебраические типы данных

Политипизм

Иногда используется термин «политипизм» или «обобщённость типа данных». По сути политипизм означает встроенную в язык поддержку полиморфизма конструкторов типов, предназначенную для унификации программных интерфейсов и повышения повторного использования кода. Примером политипизма является обобщённый алгоритм сопоставления с образцом.

По определению, в политиповой функции «*хотя и возможно наличие конечного числа ad hoc-ветвей для некоторых типов, но ad hoc-комбинатор отсутствует*».

Политипизм может быть реализован посредством использования универсального типа данных или полиморфизма высших рангов. Расширение PolyP языка Haskell представляет собой синтаксическую конструкцию, упрощающую определение политиповых функций в Haskell.

Политиповая функция является в некотором смысле более обобщённой, чем полиморфная, но, с другой стороны, функция может быть политиповой и при этом не полиморфной, что видно на примере следующих сигнатур функциональных типов:

```
head   :: [a] -> a
(+)    :: Num a => a -> a -> a
length :: Regular d => d a -> Int
sum    :: Regular d => d Int -> Int
```

Первая функция (`head`) является полиморфной (параметрически полиморфной[>>>]), вторая (инфиксный оператор «+») — перегруженной (ad hoc полиморфной[>>>]), третья и четвёртая — политиповыми: переменная типа `d` в их определении варьируется над конструкторами типов. При этом, третья функция (`length`) является *политиповой* ***и*** *полиморфной* (предположительно, она вычисляет «длину» для некоторого множества полиморфных агрегатных типов — например, количество элементов в списках и в деревьях), а четвёртая (`sum`) является *политиповой,* ***но не*** *полиморфной* (мономорфной над агрегатными типами, принадлежащими классу типов `Regular`, для которых она, вероятно, вычисляет сумму целых, образующих объект конкретного агрегатного типа).

Динамически типизируемые языки единообразно представляют разновидности полиморфизма, что формирует самобытную идеологию в них и влияет на применяемые методологии декомпозиции задач. Например, в Smalltalk любой класс способен принять сообщения любого типа, и либо обработать его самостоятельно (в том числе посредством интроспекции), либо ретранслировать другому классу — таким образом, любой метод формально является параметрически полиморфным, при этом его внутренняя структура может ветвиться по условию динамического типа аргумента, реализуя специальный полиморфизм. В CLOS мультиметоды одновременно являются функциями первого класса, что позволяет рассматривать их одновременно и как ограниченно квантифицированные[en], и как обобщённые (истинно полиморфные).

Статически полиморфно типизированные языки, напротив, могут реализовать разновидности полиморфизма ортогонально (независимо друг от друга), позволяя выстраивать их хитросплетение типобезопасным образом. Например, OCaml поддерживает параметрические классы (по возможностям аналогичные шаблонам классов C++, но верифицируемые без необходимости инстанцирования), их наследование вширь и вглубь (в том числе множественное), идиоматическую реализацию классов типов (посредством сигнатур — см. соответствующий пример использования языка модулей ML), рядный полиморфизм, параметрический полиморфизм рангов выше 1-го (посредством так называемых *локально-абстрактных типов*, реализующих экзистенциальные типы) и обобщённые алгебраические типы данных.

3.21. Специальные системы типов

Ряд специальных систем типов был разработан для использования в определённых условиях с определёнными данными, а также для статического анализа программ. В большинстве своём они основываются на идеях формальной теории типов и используются лишь в составе исследовательских систем.

3.21.1. Экзистенциальные типы

Экзистенциальные типы, то есть типы, определённые посредством экзистенциального квантификатора (квантора существования), представляют собой механизм инкапсуляции на уровне типов: это композитный тип, скрывающий реализацию некоего типа в своём составе.

Понятие экзистенциального типа часто используется совместно с понятием типа записи для представления модулей и абстрактных типов данных, что обусловлено их назначением — отделением реализации от интерфейса. Например, тип `T = ∃X { a: X; f: (X → int); }` описывает интерфейс модуля (семейства модулей с одинаковой сигнатурой), имеющий в своём составе значение данных типа `X` и функцию, принимающую параметр *в точности этого же* типа `X` и возвращающую целое число. Реализация может быть различной:

- `intT = { a: int; f: (int → int); }`
- `floatT = { a: float; f: (float → int); }`

Оба типа являются подтипами более общего экзистенциального типа `T` и соответствуют конкретно реализованным типам, так что любое значение, принадлежащее любому из них, принадлежит также к типу `T`. Если `t` — значение типа `T`, то `t.f(t.a)` проходит проверку типов, вне зависимости от того, к какому абстрактному типу принадлежит `X`. Это даёт гибкость при выборе типов, подходящих для конкретной реализации, так как пользователи извне обращаются только к значениям интерфейсного (экзистенциального) типа и изолированы от этих вариаций.

В общем случае механизм проверки согласования типов не способен определить, к какому именно экзистенциальному типу принадлежит данный модуль. В примере выше `intT { a: int; f: (int → int); }` также мог бы иметь тип `∃X { a: X; f: (int →int); }`. Простейшим решением является явное указание для каждого модуля подразумеваемого в нём типа, например:

- `intT = { a: int; f: (int → int); } as ∃X { a: X; f: (X → int); }`

Хотя абстрактные типы данных и модули использовались в языках программирования довольно давно, формальная модель экзистенциальных типов была построена лишь к 1988 году[5]. Теория представляет собой типизированное лямбда-исчисление второго порядка, аналогичное Системе F, но с экзистенциальной квантификацией вместо универсальной.

Экзистенциальные типы явным образом доступны в качестве экспериментального расширения языка Haskell, где они представляют собой специальный синтаксис, позволяющий использовать переменную типа в определении алгебраического типа, не вынося её в сигнатуру конструктора типов, то есть не повышая его арность[6]. Язык Java предоставляет ограниченную форму экзистенциальных типов посредством джокера[en]. В языках, реализующих классический let-полиморфизм в стиле ML, экзистенциальные типы могут быть симулированы посредством так называемых «*значений, индексированных типами*».

3.21.2. Явное и неявное назначение типов

Многие статические системы типов, например, такие как в языках Си и Java, требуют **провозглашения типа**: программист должен явно назначать каждой переменной конкретный тип. Другие, такие как система типов Хиндли — Милнера, применяемая в языках ML и Haskell, осуществляют **выведение типов**: компилятор выстраивает заключение о типах переменных на основании того, как программист использует эти переменные.

Например, для функции `f(x,y)`, осуществляющей сложение `x` и `y`, компилятор может сделать вывод, что `x` и `y` должны быть числами —

поскольку операция сложения определена только для чисел. Следовательно, вызов где-либо в программе функции `f` для параметров, отличных от числовых, (например, для строки или списка) сигнализирует об ошибке.

Числовые и строковые константы и выражения обычно зачастую выражают тип в конкретном контексте. Например, выражение `3.14` может подразумевать вещественное число, тогда как `[1,2,3]` может быть списком целых — обычно массивом.

Вообще говоря, выведение типов возможно, если оно принципиально разрешимо в теории типов. Более того, даже если выведение неразрешимо для данной теории типов, выведение зачастую возможно для многих реальных программ. Система типов языка Haskell, являющаяся разновидностью системы типов Хиндли — Милнера, представляет собой ограничение Системы Fω для так называемых полиморфных типов первого ранга, на которых выведение разрешимо. Многие компиляторы Хаскела предоставляют полиморфизм произвольного ранга в качестве расширения, но это делает выведение типов неразрешимым, так что требуется явное провозглашение типов. Однако, проверка согласования типов остаётся разрешимой и для программ с полиморфизмом первого ранга типы по-прежнему выводимы.

3.21.3. Унифицированные системы типов

Некоторые языки, например, C#, имеют унифицированную системы типов. Это означает, что все типы языка вплоть до примитивных наследуются от единого корневого объекта (в случае с C# — от класса `Object`). В Java есть несколько примитивных типов, не являющихся объектами. Наряду с ними Java также предоставляет обёрточные объектные типы, так что разработчик может использовать примитивные или объектные типы по своему усмотрению.

Совместимость типов

Механизм проверки согласования типов в языке со статической типизацией проверяет, что всякое выражение соответствует типу, ожидаемому тем контекстом, в котором оно присутствует. Например, в операторе присваивания вида `x := e` выведенный для выражения *e*

тип должен соответствовать типу, который провозглашён или выведен для переменной x. Нотация соответствия, называемая **совместимостью**, специфична для каждого языка.

Если *e* и x имеют единый тип, и присваивание разрешено для этого типа, то это выражение является корректным. Поэтому в простейших системах типов вопрос *совместимости* двух типов упрощается до вопроса их *равенства* (*эквивалентности*). Однако разные языки имеют разные критерии для определения совместимости типов двух выражений. Эти *теории эквивалентности* варьируются между двумя крайними случаями: **структурными системами типов** (англ. *structural type system*), в которых два типа эквивалентны, если описывают одинаковую внутреннюю структуру значения; и **номинативными системами типов** (англ. *nominative type system*), в которых синтаксически различные типы никогда не эквивалентны (то есть два типа равны только в том случае, если равны их идентификаторы).

В языках с подтипами правила совместимости более сложные. Например, если A является подтипом B, то значение, принадлежащее типу A, может быть использовано в контексте, ожидающем значение типа B, даже если обратное неверно. Как и в случае эквивалентности, отношения подтипов различаются в разных языках, и здесь возможно много вариантов правил. Наличие в языке параметрического или ситуативного полиморфизма может также влиять на совместимость типов.

3.21.4. Влияние на стиль программирования

Одни программисты предпочитают статические системы типов, другие — динамические. Статически типизированные языки сигнализируют об ошибках согласования типов на этапе компиляции, могут порождать более эффективно исполняемый код, и для таких языков осуществимо более релевантное автодополнение в интегрированных средах разработки. Сторонники динамической типизации утверждают, что они лучше подходят для быстрого прототипирования, и что ошибки согласования типов составляют лишь малую часть возможных ошибок в программах. С другой стороны, в статически типизированных языках явная декларация типов обычно не требуется, если язык поддерживает вывод типов, а некоторые динамически типизированные языки производят оптимизацию на этапе

выполнения программы, зачастую посредством применения частичного вывода типов.

Определение

Тип данных характеризует одновременно:

- множество допустимых значений, которые могут принимать данные, принадлежащие к этому типу;
- набор операций, которые можно осуществлять над данными, принадлежащими к этому типу.

Первое свойство можно рассматривать как теоретико-множественное определение понятия типа; второе — как процедурное (или поведенческое) определение.

Кроме этого, в программировании используется низкоуровневое определение типа — как заданных размерных и структурных характеристик ячейки памяти, в которую можно поместить некое значение, соответствующее этим характеристикам. Такое определение является частным случаем теоретико-множественного. На практике, с ним связан ряд важных свойств (обусловленных особенностями организации памяти компьютера), требующих отдельного рассмотрения .

Теоретико-множественное определение, особенно в низкоуровневом варианте, чаще всего используется в императивном программировании. Процедурное определение в большей степени связывается с параметрическим полиморфизмом. Объектно-ориентированное программирование использует процедурное определение при описании взаимодействия компонентов программы, и теоретико-множественное — при описании реализации этих компонентов на ЭВМ, соответственно, рассматривая «*класс-как-поведение*» и «*класс-как-объект в памяти*»[*источник не указан 1156 дней*].

Операция назначения типа информационным сущностям называется **типизацией**. Назначение и проверка согласования типов может осуществляться заранее (статическая типизация), непосредственно при использовании (динамическая типизация) или совмещать оба метода.

Типы могут назначаться «раз и навсегда» (сильная типизация) или позволять себя изменять (слабая типизация).

Типы позволяют избежать парадокса Рассела, в частности, Чёрч ввёл типы в лямбда-исчисление именно с этой целью[6].

В естественном языке за типизацию отвечают вопросительные местоимения.

Единообразная обработка данных разных типов называется полиморфизмом.

Понятие типобезопасности опирается преимущественно на процедурное определение типа. Например, попытка деления числа на строку будет отвергнута большинством языков, так как для этих типов не определено соответствующее поведение. Слабо типизированные языки тяготеют к низкоуровневому определению. Например, «*число*» и «*запись*» имеют различное поведение, но значение адреса «*записи*» в памяти ЭВМ может иметь то же низкоуровневое представление, что и «число». Слабо типизированные языки предоставляют возможность нарушить систему типов, назначив этому значению поведение «*числа*» посредством операции приведения типа. Подобные трюки могут использоваться для повышения эффективности программ, но несут в себе риск крахов, и поэтому в безопасных языках не допускаются, либо жёстко обособляются.

К неполным по Тьюрингу языкам описания данных (таким как SGML) процедурное определение обычно неприменимо.

Классификация

Существуют различные классификации типов и правил их назначения.

По аналогии с математикой, типы данных делят на *скалярные* (примитивные) и *нескалярные* (агрегатные). Значение нескалярного типа (нескалярное значение) имеет множество видимых пользователю компонентов, а значение скалярного типа (скалярное значение) не имеет такового. Примерами нескалярного типа являются массивы, списки и т. д.; примеры скалярного типа — «целое», «логическое» и т. д.

Структурные (агрегатные) типы не следует отождествлять со структурами данных: одни структуры данных непосредственно воплощаются определёнными структурными типами, но другие строятся посредством их композиции, чаще всего рекурсивной. В последнем случае говорят о рекурсивных типах данных. Примером структур данных, которые почти всегда строятся посредством композиции объектов рекурсивного типа, являются бинарные деревья.

По другой классификации типы делятся на самостоятельные и зависимые. Важными разновидностями последних являются ссылочные типы, частным случаем которых, в свою очередь, являются указатели. Ссылки (в том числе и указатели) представляют собой несоставной зависимый тип, значения которого являются адресом в памяти ЭВМ другого значения. Например, в языке Си тип «*указатель на целое без знака*» записывается как «`unsigned *`», в языке ML тип «*ссылка на целое без знака*» записывается как «`word ref`».

Также типы делятся на мономорфные и полиморфные (см. переменная типа).

3.21.5. Некоторые распространённые типы данных

Логический тип

Логические, или булевы значения (по фамилии их изобретателя — Буля), могут иметь лишь одно из двух состояний — «истина» или «ложь». В разных языках обозначаются `bool`, `BOOL`, или `boolean`. «Истина» может обозначаться как `true`, `TRUE` или `#T`. «Ложь», соответственно, `false`, `FALSE` или `#F`. В языках C и C++ любое ненулевое число трактуется как «истина», а ноль — как «ложь». В Python некоторым единичным типам также назначается то или иное «логическое значение». В принципе, для реализации типа достаточно одного бита, однако из-за особенностей микропроцессоров, на практике размер булевых величин обычно равен размеру машинного слова.

Целочисленные типы

Целочисленные типы содержат в себе значения, интерпретируемые как числа (знаковые и беззнаковые).

Числа с плавающей запятой

Используются для представления вещественных (не обязательно целых) чисел. В этом случае число записывается в виде x=a*10^b. Где 0<=a<1, a b — некоторое целое число из определённого диапазона. a называют мантиссой, b — порядком. У мантиссы хранятся несколько цифр после запятой, а b — хранится полностью.

Строковые типы

Последовательность символов, которая рассматривается как единое целое в контексте переменной. В разных языках программирования накладываются разные ограничения на строковые переменные. Строки могут содержать управляющие последовательности.

Указатели

Указатель — переменная, диапазон значений которой состоит из адресов ячеек памяти или специального значения для обозначения того, что в данный момент в переменной ничего не записано.

Идентификационные типы

Идентификационные типы интерпретируются не как число, а как уникальный идентификатор объекта. Например, FourCC.

Абстрактные типы данных

Типы данных, которые рассматриваются независимо от контекста и реализации в конкретном языке программирования. Абстракция в математическом смысле означает, что алгебра данных рассматривается с точностью до изоморфизма. Абстрактные типы находят широкое применение в методологии программирования, основанной на пошаговой разработке программ. На этапе построения спецификации проектируемой программы алгебра данных моделирует объекты предметной области, в терминах решаемой задачи. В процессе пошагового уточнения данные конкретизируются путём перехода к промежуточным представлениям до тех пор, пока не будет найдена их реализация с помощью базовой алгебры данных используемого языка

программирования. Существует несколько способов определения абстрактных типов: алгебраический, модельный и аксиоматический. При модельном подходе элементы данных определяются явным образом. При алгебраическом используются методы алгебраических отношений, а при аксиоматическом подходе используется логическая формализация.

Примеры

- примитивные типы, в том числе:
  - логический тип
  - целые типы
  - вещественные типы
- ссылочные типы
- опциональные типы[en]
  - обнуляемые типы[en]
- Композитные типы, в том числе:
  - массивы
  - записи
  - кортежи
  - абстрактные типы (АТД, англ. *ADT*)
- алгебраические типы
  - вариантные типы[en]
- подтипы[en]
- унаследованные типы
- объектные типы, то есть объекты, значением которых являются типы — например, переменные типов
- частичные типы[en]
- рекурсивные типы[en]
- функциональные типы, например бинарные функции
- универсально квантифицированные типы, такие как параметрические типы
- экзистенциально квантифицированные, такие как модули
- зависимые типы — типы, зависящие от термов (значений)
- уточняющие типы[en] — типы, идентифицирующие подмножества других типов
- Предопределённые типы (являющиеся фактически структурными, но предоставляемые на правах примитивных) для удобства промышленных разработок, такие как «*дата*», «*время*», «*валюта*» и др.

3.21.6. Самоприменение

Тип может быть параметризован другим типом, в соответствии с принципами абстракции и параметричности. Например, для реализации функции сортировки последовательностей нет необходимости знать все свойства составляющих её элементов — необходимо лишь, чтобы они допускали операцию сравнения — и тогда составной тип «*последовательность*» может быть определён как параметрически полиморфный. Это означает, что его компоненты определяются с использованием не конкретных типов (таких как «*целое*» или «*массив целых*»), а параметров-типов. Такие параметры называются переменными типа (англ. *type variable*) — они используются в определении полиморфного типа так же, как параметры-значения в определении функции. Подстановка конкретных типов в качестве фактических параметров для полиморфного типа порождает мономорфный тип. Таким образом, параметрически полиморфный тип представляет собой конструктор типов, то есть оператор над типами в арифметике типов.

Определение функции сортировки как параметрически полиморфной означает, что она сортирует абстрактную последовательность, то есть последовательность из элементов некоторого (неизвестного) типа. Для функции в этом случае требуется знать о своём параметре лишь два свойства — то, что он представляет собой последовательность, и что для её элементов определена операция сравнения. Рассмотрение параметров процедурным, а не декларативным, образом (то есть их использование на основе поведения, а не значения) позволяет использовать одну функцию сортировки для любых последовательностей — для последовательностей целых чисел, для последовательностей строк, для последовательностей последовательностей булевых значений, и так далее — и существенно повышает коэффициент повторного использования кода. Ту же гибкость обеспечивает и динамическая типизация, однако, в отличие от параметрического полиморфизма, первая приводит к накладным расходам. Параметрический полиморфизм наиболее развит в языках, типизированных по Хиндли — Милнеру, то есть потомках языка ML. В объектно-ориентированном программировании параметрический полиморфизм принято называть обобщённым программированием.

Несмотря на очевидные преимущества параметрического полиморфизма, порой возникает необходимость обеспечивать

различное поведение для разных подтипов одного общего типа, либо аналогичное поведение для несовместимых типов — то есть в тех или иных формах Ad hoc полиморфизма. Однако, ему не существует математического обоснования, так что требование типобезопасности долгое время затрудняло его использование. Ad hoc полиморфизм реализовывался внутри параметрически полиморфной системы типов посредством различных трюков. Для этой цели использовались либо вариантные типы, либо параметрические модули (функторы), либо так называемые «*значения, индексированные типами*» (англ. *type-indexed values*), которые, в свою очередь, также имеют ряд реализаций[10]. Классы типов, появившиеся в языке Haskell, предоставили более изящное решение этой проблемы.

Подробнее по этой теме см. Класс типов.

Если рассматриваемой информационной сущностью является тип, то назначение ей типа приведёт к понятию «*тип типа*» («*метатип*»). В теории типов это понятие носит название «*род типов*» (англ. *kind of a type* или *type kind*). Например, род «*» включает все типы, а род «* -> *» включает все унарные конструкторы типов. Рода явным образом применяются при полнотиповом программировании — например, в виде конструкторов типов в языках семейства ML.

Подробнее по этой теме см. Род (теория типов).

Расширение безопасной полиморфной системы типов классами и родами типов сделало Haskell первым типизированным в полной мере языком. Полученная система типов оказала влияние на другие языки (например, Scala, Agda).

Ограниченная форма метатипов присутствует также в ряде объектно-ориентированных языков в форме метаклассов. В потомках языка Smalltalk (например, Python) всякая сущность в программе является объектом, имеющим тип, который сам также является объектом — таким образом, метатипы являются естественной частью языка. В языке C++ отдельно от основной системы типов языка реализована подсистема RTTI, также предоставляющая информацию о типе в виде специальной структуры.

Динамическое выяснение метатипов называется отражением (а также рефлексивностью или интроспекцией).

3.21.7. Представление на ЭВМ

Наиболее заметным отличием реального программирования от формальной теории информации является рассмотрение вопросов эффективности не только в терминах О-нотации, но и с позиций экономической целесообразности воплощения тех или иных требований в физически изготовляемой ЭВМ. И в первую очередь это сказывается на допустимой точности вычислений: **понятие «число» в ЭВМ на практике не тождественно понятию числа в арифметике.** Число в ЭВМ представляется ячейкой памяти, размер которой определяется архитектурой ЭВМ, и диапазон значений числа ограничивается размером этой ячейки. Например, процессоры архитектуры Intel x86 предоставляют ячейки, размер которых в байтах задаётся степенью двойки: *1, 2, 4, 8, 16* и т. д. Процессоры архитектуры Сетунь предоставляли ячейки, размер которых в трайтах задавался кратным тройке: *1, 3, 6, 9* и т. д.

Попытка записи в ячейку значения, превышающего максимально допустимый для неё предел (который известен) приводит к ошибке переполнения. При необходимости расчётов на более крупных числах используется специальная методика, называемая длинной арифметикой, которая в силу значительной ресурсоёмкости не может осуществляться в реальном времени. Для наиболее распространённых в настоящее время архитектур ЭВМ «родным» является размер ячеек в 32 и 64 бит (то есть 4 и 8 байт).

Кроме того, целые и вещественные числа имеют разное представление в этих ячейках: неотрицательные целые представляются непосредственно, отрицательные целые — в дополнительном коде, а вещественные кодируются особым образом. Из-за этих различий сложение чисел «`1`» и «`0.1`», которое в теории даёт значение «`1.1`», на ЭВМ непосредственно невозможно. Для его осуществления необходимо сперва выполнить преобразование типа, породив на основании значения целого типа «`1`» новое значение вещественного типа «`1.0`», и лишь затем сложить «`1.0`» и «`0.1`». В силу специфики реализации вещественных чисел на ЭВМ, такое преобразование осуществляется не абсолютно точно, а с некоторой долей приближения. По той же причине сильно типизированные языки (например, Standard ML) рассматривают вещественный тип как equality types (или identity types) (Equality type).

Для низкоуровневого представления составных типов важное значение имеет понятие о выравнивании данных. Языки высокого уровня обычно изолируют (абстрагируют) программиста от этого свойства, однако, с ним приходится считаться при связывании независимо скомпилированных модулей между собой. Однако, некоторые языки (Си, C++) предоставляют возможность контролировать низкоуровневое представление типов, в том числе и выравнивание. Такие языки временами называют языками среднего уровня.

4. Структура данных и алгоритмы

Структура данных (англ. *data structure*) — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике. Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс.

Термин «структура данных» может иметь несколько близких, но тем не менее различных значений:

- Абстрактный тип данных;
- Реализация какого-либо абстрактного типа данных;
- Экземпляр типа данных, например, конкретный список;
- В контексте функционального программирования — уникальная единица (англ. *unique identity*), сохраняющаяся при изменениях. О ней неформально говорят как об одной структуре данных, несмотря на возможное наличие различных версий.

Структуры данных формируются с помощью типов данных, ссылок и операций над ними в выбранном языке программирования.

Различные виды структур данных подходят для различных приложений; некоторые из них имеют узкую специализацию для определённых задач. Например, B-деревья обычно подходят для создания баз данных, в то время как хеш-таблицы используются повсеместно для создания различного рода словарей, например, для отображения доменных имён в интернет-адреса компьютеров.

При разработке программного обеспечения сложность реализации и качество работы программ существенно зависит от правильного выбора структур данных. Это понимание дало начало формальным методам разработки и языкам программирования, в которых именно структуры данных, а не

алгоритмы, ставятся во главу архитектуры программного средства. Большая часть таких языков обладает определённым типом модульности, позволяющим структурам данных безопасно переиспользоваться в различных приложениях. Объектно-ориентированные языки, такие как Java, C# и C++, являются примерами такого подхода.

Многие классические структуры данных представлены в стандартных библиотеках языков программирования или непосредственно встроены в языки программирования. Например, структура данных хэш-таблица встроена в языки программирования Lua, Perl, Python, Ruby, Tcl и др. Широко используется стандартная библиотека шаблонов (STL) языка C++.

Фундаментальными строительными блоками для большей части структур данных являются массивы, записи (`struct` в Си и `record` в Паскале), размеченные объединения (`union` в Си) и ссылки. Например, двусвязный список может быть построен с помощью записей и ссылок, где каждая запись (узел) будет хранить данные и ссылки на «левый» и «правый» узлы.

Сравнение структур данных в функциональном и императивном программировании

Проектировать структуры данных для функциональных языков более сложно, чем для императивных, как минимум по двум причинам:

1. Почти все структуры данных интенсивно используют присваивание, которое в чисто функциональном стиле не используется;
2. Функциональные структуры данных являются более гибкими, и поэтому там, где в императивном программировании старая версия теряется, просто заменяясь новой, в функциональном она автоматически продолжает существовать. Другими словами, в императивном программировании (если не принять особых мер, которые могут серьёзно усложнить программу) структуры данных являются ***эфемерными*** (англ. *ephemeral*), а в функциональных программах они как правило ***постоянные*** (англ. *persistent*).

4.1. Понятие структур данных и алгоритмов

Структуры данных и *алгоритмы* являются строительным материалом программного обеспечения. В основе работы вычислительной машины лежит возможность оперировать только с одним видом данных – отдельными битами. Встроенные структуры данных представлены

регистрами и словами памяти, в которых хранятся двоичные величины. Однако подлежащие решению задачи редко выражаются на языке битов.

Как правило, **данные имеют форму чисел, литер, текстов, символов и более сложных структур – последовательностей, списков, деревьев**. Еще разнообразнее алгоритмы, применяемые для решения вычислительных задач; фактически алгоритмов не меньше чем вычислительных задач. Для точного описания абстрактных структур данных и алгоритмов программ используются такие системы формальных обозначений, в которых смысл всякого предложения определяется точно и однозначно.

Среди средств, представляемых языками программирования, имеется возможность ссылаться на элемент данных, пользуясь присвоенным ему именем, или, иначе, идентификатором. Одни именованные величины являются константами, которые сохраняют постоянное значение, другие – переменными, которым с помощью оператора может быть присвоено любое новое значение.

Компилятор, транслирующий исходный текст программы в двоичный код, связывает каждый идентификатор с определенным адресом памяти. Чтобы компилятор смог это выполнить, следует сообщить ему о «типе» каждой именованной величины. Человек, решающий задачу «вручную», обладает интуитивной способностью разбираться в типах данных и операциях, которые для каждого типа справедливы. Например, нельзя извлечь квадратный корень из слова или написать число с заглавной буквы.

Одна из причин, позволяющая легко провести такое распознавание, состоит в том, что слова, числа и другие обозначения отображаются по-разному. **В машинном представлении все типы данных сводятся к последовательности битов, поэтому различие в типах следует делать явным. Типы данных, принятые в языках программирования, включают натуральные и целые числа, вещественные (действительные) числа (в виде приближенных десятичных дробей), литеры, строки.**

В некоторых языках тип каждой константы или переменной определяется компилятором по записи присваиваемого значения; например, наличие десятичной точки может служить признаком

вещественного числа. В других языках требуется явное задание типа каждой переменной**. Значение переменной может многократно меняться, однако ее тип меняться не должен никогда; это значит, что компилятор может проверить операции, выполняемые над этой переменной, и убедиться в том, что все они согласуются с описанием типа переменной**. Такая проверка может быть проведена путем анализа всего текста программы, и в этом случае она охватит все возможные действия, определяемые данной программой.

В зависимости от назначения языка защита типов, осуществляемая на этапе компиляции, может быть более или менее жесткой. Например, **язык PASCAL, изначально являющийся, прежде всего, инструментом для иллюстрирования структур данных и алгоритмов,** сохраняет от своего первоначального назначения строгую защиту типов. PASCAL-компилятор в большинстве случаев расценивает смешение в одном выражении данных разных типов или применение к типу данных несвойственных ему операций как фатальную ошибку.

Напротив, **язык С, предназначенный, прежде всего, для системного программирования**, обладает весьма слабой защитой типов. С-компиляторы в таких случаях лишь выдадут предупреждения. Отсутствие жесткой защиты типов дает дополнительные возможности, но за правильностью действий приходится следить самостоятельно.

Структура данных относится, по существу, к «пространственным» понятиям: ее можно свести к схеме организации информации в памяти машины, в то время как алгоритмы является соответствующим процедурным элементом в структуре программы.

Первые алгоритмы были разработаны для решения численных задач, однако в настоящее время в равной степени важны численные и нечисловые алгоритмы: например, поиск в тексте заданного слова, планирование событий, сортировка данных в указанном порядке и т.п. Нечисловые алгоритмы оперируют с данными, которые не обязательно являются числами; более того, не нужны глубокие математические понятия, чтобы их конструировать или понимать. Из этого, однако, не следует, что в изучении таких алгоритмов математике нет места; напротив, точные, математические методы необходимы при поиске

наилучших решений нечисловых задач при доказательстве правильности этих решений.

Структуры данных, применяемые в алгоритмах, могут быть чрезвычайно сложными. В результате выбор правильного представления данных часто служит ключом к удачной реализации и может в большей степени сказываться на эффективности решений, чем детали используемого алгоритма. В пособии представлены **базовые элементы данных и примеры собранных из них информационных структур**.

4.1.2. Информация и ее представление

Начиная изучение структур данных необходимо установить, что понимается под информацией, как информация передается и как она физически размещается в памяти вычислительной машины.

4.1.2.1. Природа информации

В теоретико-информационном смысле **информация** рассматривается как **мера уменьшения неопределенности**. Предположим, что имеется *n* возможных состояний некоторой системы, в которой каждое состояние имеет вероятность появления *p*, причем все вероятности независимы. Тогда неопределенность этой системы определяется в виде:

$$H = -\sum_{i=1}^{n} \left[p(i) \bullet \log_2 p(i) \right]$$

Для измерения неопределенности системы выбрана единица, называемая *битом*. Бит является мерой неопределенности, связанной с наличием двух возможных состояний. Бит используется для измерения как неопределенности, так и информации, что вполне объяснимо, поскольку количество полученной информации равно количеству неопределенности, устраненному в результате получения информации.

4.1.2.2. Хранение информации

В цифровых вычислительных машинах можно выделить **три основных вида запоминающих устройств:** *сверхоперативная*, *оперативная* и *внешняя память*. Обычно ***сверхоперативная память*** **строится на регистрах**. Регистры используются для временного хранения и преобразования информации. Некоторые из наиболее важных регистров содержатся в центральном процессоре. **Центральный процессор содержит регистры, в которые помещаются аргументы (операнды) арифметических операций. Сами операции выполняется с помощью логических схем**. Кроме запоминания операндов и результа операций регистры используются для временного хранения команд программы и информации о следующей выполняемой команды.

Оперативная память **предназначена для запоминания постоянной по своей природе информации**. **Важнейшим свойством оперативной памяти является *адресуемость*: каждая ячейка памяти имеет свой идентификатор, однозначно идентифицирующий ее в общем массиве ячеек. Идентификатор называется адресом.**

В большинстве вычислительных систем единицей адресации является ***байт*** **– ячейка, состоящая из 8 двоичных разрядов.** Определенная ячейка оперативной памяти или множество ячеек могут быть связаны с конкретной переменной. Однако **для выполнения арифметических вычислений, в которых участвует переменная, необходимо, чтобы до начала вычислений значение переменной было перенесено из ячейки памяти в регистр.**

Если результат вычисления должен быть присвоен переменной, то результирующая величина снова должна быть перенесена из соответствующего регистра в связанную с этой переменной ячейку оперативной памяти. Во время выполнения программы ее команды и данные в основном размещаются в ячейках оперативной памяти. Полное множество элементов оперативной памяти часто называют *основной памятью*.

Внешняя память **служит для долговременного хранения данных.** Данные на внешней памяти могут сохраняться после завершения создавшей их программы и впоследствии многократно использованы

той же программой при повторных ее запусках или другими программами. **Внешняя память используется также для хранения самих программ, когда они не выполняются**.

Поскольку стоимость внешней памяти значительно меньше оперативной, а объем значительно больше, то еще одно назначение внешней памяти – **временное хранение тех кодов и данных выполняемой программы, которые не используются на данном этапе ее выполнения**. Активные коды выполняемой программы и обрабатываемые ею на данном этапе данные должны быть размещены в оперативной памяти, так как **прямой обмен между внешней памятью и операционными устройствами (регистрами) невозможен.**

Как хранилище данных, внешняя память обладает в основном теми же свойствами, что и оперативная, в том числе и свойством адресуемости. В принципе структуры данных на внешней памяти могут быть теми же, что и в оперативной, и алгоритмы их обработки могут быть одинаковыми. Но внешняя память имеет иную физическую природу. На физическом уровне для нее применяются иные методы доступа, обладающие другими временными характеристиками. В результате структуры и алгоритмы, эффективные для оперативной памяти, не оказываются таковыми для внешней памяти.

4.1.2.3. Классификация структур данных

Независимо от содержания и сложности **любые данные в памяти вычислительной машины представляются последовательностью двоичных разрядов, или битов, а их значениями являются соответствующие двоичные числа**. Данные, рассматриваемые в виде последовательности битов, имеют простую организацию или, другими словами, слабо структурированы. Для человека описывать и работать со сложными данными в терминах последовательностей битов весьма неудобно. Более крупные и содержательные элементы данных образуются на основе понятия «структуры данного».

Под *структурой данных* в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур

данных, направления которых соответствуют различным особенностям их рассмотрения.

Прежде чем приступать к изучению конкретных структур данных, **приведем их общую классификацию по нескольким признакам.** Каждую структуру данных характеризуют *логическим* и *физическим* представлениями. Понятие «*физическая структура данных*» отражает способ физического представления данных в памяти машины и называется иначе **структурой хранения, внутренней структурой или структурой памяти**. Рассмотрение структуры данных без учета ее представления в машинной памяти называется **абстрактной или *логической структурой*.**

Физическое представление обычно не соответствует логическому, и, кроме того, может существенно различаться в разных программных системах. Степень различия зависит от самой структуры и особенностей среды, в которой она должна быть отражена. Вследствие этого различия **существуют процедуры, осуществляющие отображение логической структуры в физическую и наоборот.** Эти процедуры обеспечивают доступ к физическим структурам и выполнение над ними различных операций, причем каждая операция рассматривается применительно к логической или физической структуре.

Различают ***простые*** **(базовые, примитивные) структуры (типы) данных и *интегрированные* (структурированные, композитные, сложные).** *Простыми* называются структуры данных, которые не могут быть разделены на составные части, большие, чем биты. С точки зрения физической структуры важным является то обстоятельство, что в данной машинной архитектуре, в данной системе программирования всегда можно сказать, каков будет размер данного простого типа и какова структура его размещения в памяти. С логической точки зрения простые данные являются неделимыми единицами.

Интегрированными называются структуры данных, составными частями которых являются другие структуры данных – простые или в свою очередь интегрированные. **Интегрированные структуры данных конструируются с использованием средств интеграции данных, предоставляемых алгоритмическими языками**.

В зависимости от отсутствия или наличия явно заданных связей между элементами данных различают ***несвязные структуры*** **(векторы, массивы, строки, стеки, очереди) и** ***связные структуры*** **(связные списки).**

Важный признак структуры данных – ее *изменчивость* – изменение числа элементов и (или) связей между элементами структуры. В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости.

По признаку изменчивости различают структуры *статические, полустатические, динамические*. Классификация структур данных по признаку изменчивости приведена на рис. 1.1.

Рис. 1.1. Классификация структур данных**.**

Базовые структуры данных, статические, полустатические и динамические характерны для оперативной памяти и часто называются *оперативными структурами*. **Файловые структуры соответствуют структурам данных для внешней памяти.**

Важный признак структуры данных – характер упорядоченности элементов. По этому признаку структуры можно разделить на *линейные* и *нелинейные*. В зависимости от взаимного расположения элементов в памяти линейные структуры можно разделить на структуры с *последовательным* распределением элементов в памяти

(**векторы, строки, массивы, стеки, очереди**) и структуры с*произвольным связным* распределением элементов в памяти (**односвязные, двусвязные списки**). Пример нелинейных структур – многосвязные списки, деревья, графы.

В языках программирования понятие «структуры данных» тесно связано с понятием «типы данных». Любые данные характеризуются своими типами. Информация по каждому типу однозначно определяет:

- структуру хранения данных указанного типа, т.е. выделение памяти и представление данных в ней, с одной стороны, и интерпретирование двоичного представления, с другой;
- множество допустимых значений, которые может иметь объект описываемого типа;
- множество допустимых операций, которые применимы к объекту описываемого типа.

При описании и конструировании структур данных будет использоваться язык Паскаль. Язык был создан Н.Виртом для иллюстрации структур данных и алгоритмов и традиционно используется для этих целей.

4.1.3. Операции над структурами данных

Над любыми структурами данных могут выполняться **четыре общие операции: создание, уничтожение, выбор (доступ), обновление.**

Операция создания заключается в выделении памяти для структуры данных. Память может выделяться в процессе выполнения (динамическое размещение в памяти) или на этапе компиляции (статическое размещение). В ряде языков (например, в С) для структурированных данных операция создания включает в себя также установку начальных значений параметров создаваемой структуры.

Для структур данных, объявленных в программе, память выделяется автоматически средствами систем программирования, на этапе компиляции, либо при активизации процедурного блока, в котором объявляются соответствующие переменные. Можно самостоятельно выделять память для структур данных, используя имеющиеся в системе подпрограммы выделения/освобождения памяти. **В объектно-**

ориентированных языках программирования при разработке нового объекта для него должны быть определены процедуры создания и уничтожения, представляемые конструкторами и деструкторами.

Независимо от используемого языка, имеющиеся структуры данных не появляются «из ничего», а явно или неявно объявляются операторами создания структур. В результате всем экземплярам структур в программе выделяется память для их размещения.

Операция уничтожения структур данных противоположна по своему действию операции создания. Некоторые языки, такие как BASIC, FORTRAN не дают возможности уничтожать созданные структуры данных. В языках PL/1, C, PASCAL структуры данных, имеющиеся внутри блока, уничтожаются в процессе выполнения программы при выходе из этого блока (например, удаление локальных переменных подпрограмм). Операция уничтожения помогает эффективно использовать память.

Операция выбора используется для доступа к данным внутри структуры. Способ доступа зависит от типа структуры данных, к которой осуществляется обращение. **Реализация метода доступа – один из наиболее важных свойств структур**.

Операция обновления позволяет изменить значения данных в структуре данных. Примером операции обновления является операция присваивания, или, более сложная форма – передача параметров.

Вышеуказанные четыре операции обязательны для всех структур данных. Помимо этого для каждой структуры данных могут быть определены специфические операции.

4.1.4. Порядок алгоритма

Критерием оценки эффективности алгоритма является его порядок. ***Порядком алгоритма*** **называется функция O(n), позволяющая оценить зависимость времени выполнения алгоритма от объема обрабатываемых данных (n – количество элементов).** Говорят, алгоритм принадлежит классу O(f(n)), где f(n) – некоторая функция от n. Такое обозначение читается как «O большое от f(n)» или менее

строго «пропорционально f(n)». Например, алгоритм последовательного поиска принадлежит к классу O(n), а бинарный – к классу O(log(n)). Эффективность тем выше, чем меньше время его выполнения зависит от объема данных.

Поскольку для положительных чисел log(n) < n, можно сделать вывод, что бинарный поиск всегда быстрее последовательного. Предположим, экспериментально определено, что некоторый алгоритм принадлежит к классу $O(n^2+n)$. Следовательно, можно подобрать константу k, для которой:

Быстродействие=k(n^2+n)*

Отсюда видно, что умножение математической функции внутри скобок в О-нотации на константу не оказывает влияния на смысл нотации. Например, O(3*f(n)) эквивалентно O(f(n)), поскольку 3 можно вынести как коэффициент пропорциональности. Кроме того, если величина n достаточно велика при тестировании алгоритма, можно утверждать, что влияние члена «n» поглощается «n^2». Поэтому алгоритм $O(n^2+n)$ эквивалентен $O(n^2)$. То же справедливо и для высших степеней, например, влияние «n^2» будет поглощено «n^3». В свою очередь, влияние log(n) будет поглощаться членом n.

Предположим, некоторый алгоритм выполняет несколько различных задач. Первая задача принадлежит к классу O(n), вторая – к классу $O(n^2)$, третья – к классу O(log(n)). Требуется определить быстродействие алгоритма в целом. Ответом будет $O(n^2)$, поскольку к этому классу принадлежит доминантная часть алгоритма.

Таким образом, значения О большого являются репрезентативными только для больших значений n. Для маленьких значений О-нотация не имеет смысла, а на общий результат оказывают влияние другие члены нотации.

Предположим, проводится тестирование двух алгоритмов. Эмпирическим путем выведены следующие зависимости:

Быстродействие 1=k1(n+100000)*

*Быстродействие 2=k2*n^2*

Константы k1 и k2 сравнимы по величине. Если следовать О-нотации, предпочтительнее будет первый алгоритм, поскольку он принадлежит к классу O(n). Однако, если известно, что в реальных условиях n не будет превышать 100, более эффективным окажется второй алгоритм. Следовательно, алгоритм нужно выбирать не только основываясь на О-нотации, но и исходя из условий его применения и статистических данных о времени его выполнения.

О-нотация относится к *среднему случаю*. Определенные условия выполнения алгоритма и исходных данных позволяют получить лучший и худший случай. Например, при последовательном поиске в массиве данных элемента, расположенного в самом начале, приведет к его обнаружению на первой же итерации цикла. Такая ситуация известна как *лучший случай* и ее можно представить как O(1) (выполнение алгоритма занимает одно и то же время независимо от количества элементов).

Если бы искомый элемент располагался бы всегда в конце массива, последовательный поиск был бы очень медленным и его порядок равен O(n). Такая ситуация известна как *худший случай*. Для бинарного алгоритма поиска лучший случай соответствует расположению искомого элемента точно посередине массива. Тем не менее, быстродействие бинарного поиска в худшем случае намного выше, чем для последовательного.

В общем случае при выборе алгоритма следует учитывать значения в О-нотации для среднего и худшего случаев. Лучшие случаи, как правило, не интересны, поскольку обычно обеспокоены граничными условиями, по которым судят о быстродействии.

Большинство алгоритмов с точки зрения порядка сводятся к трем основным типам: степенным $O(n^a)$, линейным O(n) и логарифмическим $O(\log_a n)$. Эффективность степенных алгоритмов считается плохой, линейных – удовлетворительной, логарифмических – хорошей. Аналитическое определение порядка алгоритма сложно, но в большинстве случаев возможно. В реальных задачах имеются ограничения, определяемые как логикой задачи, так и свойствами конкретной вычислительной среды, которые могут помогать или мешать и существенно влиять на эффективность конкретной реализации алгоритма. Поэтому выбор того или иного алгоритма всегда остается за разработчиком.

4.1.5. Структурность данных и технологии программирования

Знание структуры данных позволяет организовать их хранение и обработку максимально эффективным образом с точки зрения минимизации затрат памяти и процессорного времени. Другим важным преимуществом, которое обеспечивается структурным подходом к данным, является возможность структурирования сложного программного изделия.

Современные промышленно выпускаемые программные пакеты – изделия чрезвычайно сложные, объем которых исчисляется тысячами и миллионами строк кода, а трудоемкость разработки – сотнями человеко-лет. Разработать такое программное изделие сразу невозможно, оно должно быть представлено в виде определенной структуры – составных частей и связей между ними. Правильное структурирование дает возможность на каждом этапе разработки сосредоточить внимание на одной обозримой части изделия или поручить реализацию разных его частей разным исполнителям.

При структурировании больших программных изделий возможно применение подхода, основанного на структуризации алгоритмов и известного, как «*нисходящее проектирование*» или «программирование сверху вниз», или подхода, основанного на структуризации данных и известного, как «*восходящее проектирование*» или «программирование снизу вверх».

В первом случае структурируют действия, которые должна выполнять программа. **Большую и сложную задачу представляют в виде нескольких подзадач меньшего объема.** Таким образом, **модуль самого верхнего уровня, отвечающий за решение всей задачи в целом, получается достаточно простым и обеспечивает только последовательность обращений к модулям, реализующим подзадачи.**

На первом этапе проектирования модули подзадач выполняются в виде «заглушек». Затем каждая подзадача в свою очередь подвергается декомпозиции по тем же правилам. Процесс дробления на подзадачи продолжается до тех пор, пока на

очередном уровне декомпозиции не получат подзадачу, реализация которой будет вполне обозримой.

В предельном случае декомпозиция может быть доведена до того, что подзадачи самого нижнего уровня могут быть решены элементарным действием, например, с помощью одного оператора языка программирования.

Другой подход к структуризации основывается на данных. У реального программного изделия всегда есть Заказчик. Заказчик имеет входные данные, и хочет, чтобы по ним были получены выходные данные, а какими средствами это обеспечивается – его не интересует. Таким образом, **задачей любого программного изделия является преобразование входных данных в выходные.**

Инструментальные средства программирования предоставляют набор базовых типов данных и операции над ними. Интегрируя базовые типы, создают более сложные структуры, и определяет новые операции над ними. **Полученные на первом шаге композиции «строительные блоки» используются в качестве базового набора для следующего шага, результатом которого будут еще более сложные конструкции данных с соответствующими операциями над ними. В идеале последний шаг композиции дает структуры, соответствующие выходным данным задачи, а операции над этими типами реализуют в полном объеме задачу проекта.**

Нередко противопоставляют нисходящее проектирование восходящему, придерживаясь одного выбранного подхода, что в корне не верно. **Реализация проекта всегда ведется встречными путями с постоянной коррекцией алгоритмов по результатам разработки структур данных и наоборот.**

Еще одним технологическим приемом, связанным со структуризацией данных является *инкапсуляция*, которая заключается в том, что сконструированный новый тип данных оформляется таким образом, что его внутренняя структура недоступна извне, т.е. пользователям данного типа. Оперировать с данными этого типа возможно только через вызовы процедур, определенных в нем. Новый тип данных представляется в виде «черного ящика», для которого известны входы и выходы, но содержимое – неизвестно и недоступно.

Инкапсуляция полезна как средство преодоления сложности, и как средство защиты от ошибок. Первая цель достигается за счет того, что сложность внутренней структуры нового типа и алгоритмов выполнения операций над ним исключается из поля зрения разработчика-пользователя. Вторая цель достигается тем, что возможности доступа пользователя ограничиваются лишь заведомо корректными входными точками, следовательно, снижается и вероятность ошибок.

Современные языки программирования блочного типа (PASCAL, C) обладают развитыми возможностями построения программ модульной структуры и управления доступом модулей к данным и процедурам. Сконструированные и полностью закрытые типы данных представляют объекты, а процедуры, работающие с их внутренней структурой – методами. Развитие данного подхода связано с объектно-ориентированной методологией, реализованной в виде объектных моделей в различных языках программирования.

4.2. Простые структуры данных

Простой тип данных определяет упорядоченное множество значений некоторого параметра. Простые типы описываются *базовыми типами*, к которым относятся: *числовые*, *битовые*, *логические*, *символьные*, *перечисляемые*, *интервальные* и *указатели*. Структура некоторых простых типов языка Паскаль приведена на рис. 2.1.

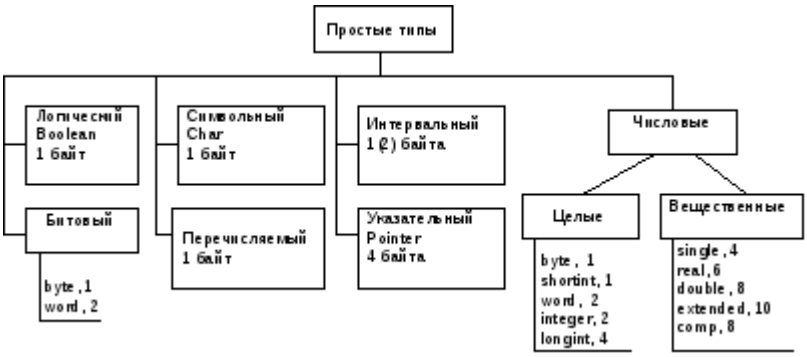


Рис. 2.1. Структура простых типов языка Паскаль.

Для каждого типа указан размер памяти в байтах, требуемый для размещения переменных соответствующего типа. В других языках набор простых типов может несколько отличаться. Все простые типы, за исключением вещественных и указателей, являются *порядковыми*.

4.2.1. Порядковые типы

Порядковые типы имеют конечное (счетное) множество значений, с каждым из которых соотносится целое число – порядок. Значения порядковых типов упорядочены (расположены) по возрастанию или убыванию. В табл. 2.1 приведены функции, применимые к любому порядковому типу. Для всех функций тип аргумента должен быть порядковым.

Для функций High и Low аргументом может быть **переменная порядкового типа, типа-массива, типа-строки.** Результат функции для величины порядкового типа – максимальное (минимальное) значение этой величины, типа-массива – максимальное (минимальное) значение индекса, типа-строки – объявленный размер строки (ноль для функции Low).

Табл. 2.1. Функции для величин порядкового типа.

Функция	Определение	Тип результата
Hi	Получение максимального значения величины.	Целый
Lo	Получение минимального значения величины.	Целый
Odd	Проверка на нечетность.	Булевый
Ord	Порядковый номер.	Целый
Pred	Предшествующее значение.	Совпадает с аргументом
Succ	Последующее значение.	Совпадает с аргументом

Функция Odd возвращает True для нечетного аргумента и False для четного.

Функция Ord преобразует любой порядковый тип в целый тип. Например, если x – переменная целого типа, то Ord(x) = x. Для символьного типа в соответствии со стандартом ASCII:

Ord(B) = 66, Pred(B) = A, Succ(B) = C

Для порядковых типов справедливы соотношения:

Ord(Pred(X)) = Ord(X)–1

Ord(Succ(X)) = Ord(X)+1

4.2.2. Целочисленный тип

С помощью *целочисленного типа* может быть представлено количество объектов, являющихся дискретными по своей природе. В языке Паскаль существуют два базовых типа для работы с целочисленными значениями: Integer и Cardinal. Подтипы базовых типов включают также ShortInt, SmallInt, LongInt, Int64, Byte, Word и LongWord. В табл. 2.2 перечислены диапазон значений и формат хранения (представление) в памяти для каждого из них.

Табл. 2.2. Целые типы данных.

Тип	**Диапазон значений**	**Представление**
Int64	$-2^{63}..2^{63}-1$	знаковый 64-битный
Integer	-2147483648..2147483647	знаковый 32-битный
LongInt	-2147483648..2147483647	знаковый 32-битный
Cardinal	0..4294967295	беззнаковый 32-битный
SmallInt	-32768..32767	знаковый 16-битный

ShortInt	-128..127	знаковый 8-битный
Byte	0..255	беззнаковый 8-битный
Word	0..65535	беззнаковый 16-битный
LongWord	0..4294967295	беззнаковый 32-битный

К целочисленным операциям относятся четыре основных арифметических действия (сложение, вычитание, умножение и деление) для которых применимы математические правила старшинства операций. Для изменения порядка вычислений используются круглые скобки. Их можно использовать для составления выражений:

a+b/c-25·(d-e)

Результат операции над целыми числами не должен выходить за диапазон допустимых значений. В некоторых компиляторах можно задать режим проверки на переполнение каждой целочисленной операции, но это может привести к большим издержкам во время выполнения, если только данная проверка не производится аппаратными средствами.

В математике в результате деления двух целых чисел a/b получается два значения: частное q и остаток r, такие что:

a=q·b+r

В языке Паскаль для получения частного используется оператор деления «/». Для целочисленного деления используется операция div, а для получения остатка – операция mod, которая может быть представлена через div:

x mod *y=x-(x* div *y)·y*

Следует помнить, что арифметические операции для типа LongInt выполняются более чем вдвое дольше, нежели для типа Integer. Причина заключается в необходимости привлечения дополнительных

команд для распространения переноса, возникающего из слова (двух байт) младших разрядов в слово старших разрядов.

4.2.3. Символьный тип

Значениями *символьного типа* являются символы некоторого предопределенного множества. В основном символьный тип данных используется как базовый для построения составного типа «строка символов». В большинстве современных вычислительных машин таким множеством является кодировка ASCII или UNICODE. Множество ASCII состоит из 256 символов, упорядоченных определенным образом, и содержит символы заглавных и строчных букв, цифр и других символов, включая специальные управляющие символы.

Кодировка ASCII не является единственной. Другой схожей кодировкой является EBCDIC (Extended Binary Coded Decimal Interchange Code – расширенный двоично-кодированный десятичный код обмена), применяемый в вычислительных машинах IBM. В EBCDIC код символа также занимает один байт, но с иной кодировкой, чем в ASCII.

Кодировки ASCII и EBCDIC включают в себя буквенные символы только латинского алфавита. Символы национальных алфавитов занимают свободные места в таблицах кодов и, таким образом, одна таблица может поддерживать только один национальный алфавит. Этот недостаток преодолен в кодировке UNICODE, которая получила большое распространение.

В кодировке UNICODE каждый символ кодируется двумя байтами, что обеспечивает 65536 возможных кодовых комбинаций и дает возможность иметь единую таблицу кодов, включающую в себя все национальные алфавиты.

В языке Паскаль используется кодировка ASCII, а стандартным символьным типом данных является тип Char. В памяти переменная типа Char занимает 1 байт. Значениями символьного типа являются множество всех символов кодировки ASCII, включая невидимые символы клавиатуры. Каждому символу присвоен код – целое число типа Byte (0..255), который возвращает функция Ord. Например, Ord(A) = 65; Ord(F) = 70. Стандартная функция Chr(X), возвращающая

символ по его коду (аргумент X должен быть байтовым), например, Chr (90) = Z.

В табл. 2.3 перечислены некоторые коды служебных символов клавиатуры. Для включения символа, не имеющего физического изображения, используется его ASCII-код с символом # перед ним.

Табл. 2.3. Специальные коды ASCII.

Символ	Клавиша	Назначение
#32	<Пробел>	Пропуск позиции
#27	<Escape>	Отмена действия
#26	<Ctrl+Z>	Конец файла
#13	<Enter>	Возврат каретки
#10	–	Конец строки

Операция сравнения является типичной над символьным типом данных. При сравнении коды символов рассматриваются как целые числа без знака. Кодовые таблицы строятся так, что результаты сравнения подчиняются *лексикографическим правилам*: символы, занимающие в алфавите места с меньшими порядковыми номерами, имеют меньшие коды, чем символы, занимающие места с большими номерами, например, <‘A’ ’Z’ (65 < 90).

4.2.4. Перечисляемый тип

Язык Паскаль позволяет создавать собственные типы, которые точнее соответствуют объектам решаемой задачи. Предположим, шкала некоторого устройства содержит следующие позиции: off (выключено), low (слабо), medium (средне), high (сильно). Для представления таких позиций можно объявить целочисленную переменную и для обозначения позиций четыре произвольных числа.

Однако в таком случае придется постоянно отслеживать по документации соответствие позиции и принятого для него номера, что

усложнит работу с программой и ее дальнейшее сопровождение. Более того, возможно ошибочное присвоение некорректного номера, выход за диапазон или любая другая непредвиденная ситуация. Использование типа-перечисления решает эти проблемы.

Перечисляемый тип – упорядоченный набор идентификаторов, заданный их перечислением. Значение данного типа представляет собой любой идентификатор из этого набора. Перечисляемые типы аналогичны целочисленным, однако набор операций, выполняемых над ними, ограничен: допустимы операции присваивания (:=), равенства (=) и неравенства (<, >, >=, <=). Операции отношений определены потому, что набор значений в объявлении интерпретируется как упорядоченная последовательность.

В языке Паскаль перечисляемый тип является стандартным и определяется набором идентификаторов, с которыми могут совпадать значения параметра:

type

< имя типа > = (< идентификатор 1, идентификатор 2,..., идентификатор n >)

Объявление перечисляемого типа для приведенного выше примера:

type

TPosition = (Off, Low, Medium, High);

Порядок перечисления идентификаторов важен, т.к. им определяется порядковые номера, которые присваиваются идентификаторам. Для переменной перечислимого типа выделяется один байт, в который записывается порядковый номер присваиваемого значения. Перечисляемый тип может быть сразу описан в разделе переменных:

var

Position: (Off,Low,Medium,High);

Перечисляемым идентификаторам ставится в соответствие последовательность целых чисел (порядок), начинающаяся с нуля. Поэтому к данным перечисляемого типа можно применять все стандартные функции и операции для порядковых типов, например:

Ord(Low) = 1

Low Off>

Succ(Medium) = High

Pred(Medium) =Low

4.2.5. Интервальный тип

Другой способ образования новых типов из уже существующих заключается в ограничении допустимого диапазона значений некоторого стандартного типа или границ перечисляемого. Ограничение определяется заданием минимального и максимального значений диапазона. При этом изменяется диапазон допустимых значений по отношению к базовому типу, но представление в памяти полностью соответствует базовому типу.

Таким способом формируется *интервальный тип*. Значения интервального типа могут принадлежать ограниченному поддиапазону некоторого базового типа. Базовым типом диапазона может быть любой порядковый тип, кроме интервального.

Для введения интервального типа необходимо указать имя типа и границы диапазона:

type

< имя типа > = < мин. значение >..< макс. значение >

Минимальное значение при определении не может быть больше максимального:

type

TTemp = -50..+50; { тип ShortInt }

TIndex = 1..100; { тип Byte }

Все операции, применимые к величинам базового типа, можно применять и к величинам соответствующего интервального типа с учетом его границ. Преимущества использования интервальных типов заключается в наглядности представления, экономном распределении памяти под переменные и дополнительном контроле значений переменных.

4.2.6. Логический тип

Логический тип является перечисляемым с двумя возможными значениями «ложь» и «истина»:

type

Boolean = (False, True);

Логические типы языка Паскаль приведены в табл. 2.4.

Табл. 2.4. Логические типы данных.

Название типа	Длина, байт
Boolean	1
ByteBool	1
WordBool	2
LongBool	4

Основным типом является Boolean, для него справедливы следующие соотношения:

Ord(False) = 0; Ord(True) = 1;

Succ(False) = True; Pred(True) = False; <False True

Остальные три типа введены для совместимости с другими языками и операционной системой Windows. Для них справедливы следующие соотношения:

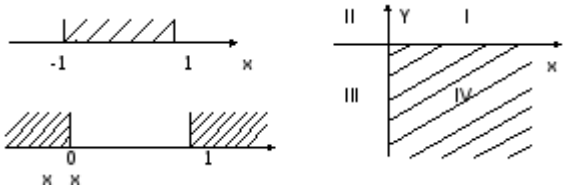
Ord(False) = 0;

0><Ord(True) (любое целое число)

Логический тип имеет большое значение поскольку:

- операции отношения являются функциями, возвращающими значение булевого типа;
- условный оператор проверяет выражение булевого типа;
- операции булевой алгебры определены для булевого типа.

Примеры применения логических функций для сведения нескольких условий в одно логическое выражение приведены на рис. 2.2.



а), б), в).

Рис. 2.2. Одномерная (а), двухсвязная одномерная (б) и двухмерная (в) области координат.

Сформированные логические выражения по этим условиям выглядят следующим образом:

а). -1 1)=<x) and (x=<(-1→1≤x≤

в). x 1)>0) or (x<(x→1>0 или x<

б). 0)<0) and (y>(x→⎡0>x⎡

⎣0<y⎣

4.2.7. Битовый тип

В ряде задач может потребоваться работа с отдельными двоичными разрядами данных. Чаще всего это возникают в системном программировании, когда, например, отдельный разряд связан с состоянием аппаратного переключателя. Данные *битового типа* представляются в виде набора битов, упакованных в байты или слова, и не связанных друг с другом. Операции над такими данными обеспечивают доступ к выбранному биту. В языке Паскаль роль битовых типов выполняют беззнаковые целые типы Byte и Word. Над этими типами помимо операций, характерных для числовых типов, допускаются побитовые логические операции и операции сдвига.

4.2.8. Вещественный тип

В отличие от рассмотренных порядковых типов, значения которых всегда сопоставляются с рядом целых чисел и, следовательно, представляются в памяти абсолютно точно, значение вещественных типов может быть определено лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа. Суммарное количество байтов, диапазоны допустимых значений чисел вещественных типов и количество значащих цифр после запятой в представлении чисел приведены в табл. 2.5.

Табл. 2.5. Вещественные типы.

Тип	Диапазон значений	Значащие цифры	Размер в байтах
Real	$2.9\cdot10^{-39}..1.7\cdot10^{38}$	11-12	6
Single	$1.4\cdot10^{-45}..3.4\cdot10^{38}$	7-8	4
Double	$4.9\cdot10^{-324}..1.8\cdot10^{308}$	15-16	8
Extended	$3.1\cdot10^{-4944}..1.2\cdot10^{4932}$	19-20	10

В языке Паскаль переменные *вещественного типа* представляются в форме, близкой к научной нотации:

const

pi: Real = 3.1415926;

eq: Real = 1.19e-31;

Для вычислений с плавающей запятой необходимо минимум 32 разряда (одинарная точность, тип Single). Однако часто и одинарной точности оказывается недостаточно, поэтому языки поддерживают объявления переменных и вычисления с двойной точностью 64 разряда (тип Double).

Для переменных вещественного типа базовыми являются арифметические операции и операции отношения. Другие математические операции реализуются в разных языках по-разному. В языке Паскаль к стандартным функциям относятся также арифметические функции, перечисленные в табл. 2.6.

Табл. 2.6. Стандартные арифметические функции.

Функция	**Назначение**	**Тип результата**
Abs(x)	Абсолютное значение аргумента.	совпадает с x
Arctan(x)	Арктангенс аргумента.	вещественный
Cos(x)	Косинус аргумента.	-//-
Exp(x)	Вычисление экспоненты.	-//-
Frac(x)	Дробная часть аргумента.	-//-
Int(x)	Целая часть числа.	-//-
Ln(x)	Натуральный логарифм.	-//-
Pi(x)	Число pi = 3.1415926535897932385.	-//-

Round(x)	Округление до ближайшего целого.	-//-
Sin(x)	Синус аргумента.	-//-
Sqr(x)	Квадрат аргумента.	совпадает с x
Sqrt(x)	Квадратный корень аргумента.	вещественный
Trunc(x)	Целая часть вещественного числа.	-//-

При разработке программного обеспечения для численных расчетов не следует допускать следующие основные ошибки: *исчезновение операнда*, *умножение ошибки* и *потерю значимости*.

Ошибка исчезновения операнда может возникнуть в операциях сложения или вычитания, если один операнд относительно мал по сравнению с другим операндом. Например, при десятичной арифметике с пятью цифрами:

$0.1234 \times 10^{3} + 0.1234 \times 10^{-4} = 0.1234 \times 10^{3}$

второй операнд будет игнорирован вследствие своей малой величины.

Умножение ошибки – это большая абсолютная ошибка, которая может появиться при использовании арифметики с плавающей точкой, даже если относительная ошибка мала. Обычно это является результатом операции умножения или деления. Рассмотрим вычисление x*x:

$0.1234 \times 10^{3} * 0.1234 \times 10^{3} = 0.1522 \times 10^{5}$

и предположим, что при вычислении x произошла ошибка на единицу младшего разряда, что соответствует абсолютной ошибке 0.1:

$0.1235 \times 10^{3} * 0.1235 \times 10^{3} = 0.1525 \times 10^{5}$

Абсолютная ошибка теперь равна 30, что на порядок превышает исходную ошибку.

Полная потеря значимости – наиболее грубая ошибка, вызванная вычитанием почти равных чисел:

f1 = 0.12342;

f2 = 0.12346;

В математике f2 – f1 = 0.00004, что, конечно, вполне представимо как четырехразрядное число с плавающей точкой: 0.4000×10^{-4}. Однако программа, выполняющая вычисление разности в четырехразрядном представлении с плавающей точкой даст ответ:

$0.1235 – 0.1234 = 0.1000 \times 10^{-3}$

что даже приблизительно не является приемлемым ответом.

Потеря значимости встречается довольно часто, поскольку проверка на равенство реализуется вычитанием и последующим сравнением с нулем. Следующее выражение для вещественных чисел f1 и f2 недопустимо:

если f1 = f2, тогда...

Правильный способ проверки равенства с плавающей точкой состоит в том, чтобы ввести малую величину epsilon:

если |f2 – f1| < epsilon, тогда...

и затем сравнивать с ней абсолютную разницу.

Ошибки в вычислениях с плавающей точкой часто можно уменьшить *изменением порядка действий*. Поскольку сложение производится слева направо. Например, для четырехразрядного десятичного вычисления:

1234.0 + 0.5678 + 0.5678 = 1234.0

лучше изменить порядок, чтобы не было исчезновения слагаемых:

0.5678 + 0.5678 + 1234.0 = 1235.0

В качестве другого примера рассмотрим арифметическое тождество:

$(x + y)*(x – y) = x^2 – y^2$

При вычислении выражения небольшая ошибка, являющаяся результатом сложения и вычитания, значительно возрастает при умножении. При вычислении выражения по формуле $x^2 – y^2$ ошибка уменьшается от исчезновения слагаемого, и результат получается более точным.

4.2.9. Указательный тип

Все рассмотренные типы данных имеют одну общую черту. Перед использованием переменная должна быть предварительно описана в разделе переменных. Переменные, определяемые в явных описаниях, называются *статическими*. Память для таких переменных выделяется при запуске программы (или при входе в структурный блок, в котором описана переменная, например, в подпрограммы) и освобождается при завершении ее работы (или при выходе из блока).

Указательный тип данных позволяет использовать *динамические переменные*, когда выделение и освобождение памяти заданного объема выполняется по явному требованию. Ячейки памяти, относящейся к динамической области, могут неоднократно выделяться и освобождаться по необходимости во время выполнения программы. Выделение и освобождение динамической памяти может быть очень хаотичным.

Указатель (ссылка) – переменная, значением которой является физический адрес некоторой ячейки памяти. Адрес занимает 4 смежных байта памяти (два слова). Поскольку переменные большинства типов данных занимают несколько смежных байтов, то указатель содержит адрес первой ячейки памяти. Необходимость использования указателей при решении прикладных задач с использованием языков высокого уровня существует очень часто.

Во-первых, указатели используются для представления одной и той же области памяти, и, следовательно, одних и тех же физических данных,

как данных разной логической структуры. В этом случае вводятся два или более указателей, которые содержат адрес одной и той же области памяти, но имеющих разный тип. Обращаясь к области памяти по тому или иному указателю, можно обработать ее содержимое как данные того или иного типа.

Во-вторых, указатели незаменимы при работе с *динамическими структурами данных*. Память под такие структуры выделяется в ходе выполнения программы. Стандартные подпрограммы выделения памяти возвращают адрес выделенной области памяти в виде указателя на нее. К содержимому динамически выделенной области памяти можно обращаться только через указатель.

Значением *указательного типа* (pointer type) является адрес другой переменной или константы. Объект, на который указывает указатель, называют *указуемым*. Указатели используются для выполнения операций над адресами ячеек (рис. 2.3).

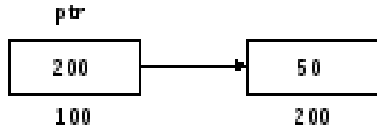


Рис. 2.3. Переменная-указатель и указуемая переменная.

В примере указатель ptr является переменной, расположенной в памяти по адресу 100. Содержимое этой ячейки, значение 200, является фактическим адресом переменной, которая содержит значение 50. К переменной можно получить доступ, просто обращаясь к ней по имени, но также можно использовать и *разыменование указателя*. При выполнении операции разыменования, получают не содержимое переменной указателя ptr, а содержимое ячейки памяти, адрес которой содержится в ptr, т.е. указуемый объект.

Операции над указателями. Указатели могут участвовать в операциях присваивания, получения адреса и выборки.

Операция присваивания – двухместная, оба операнда которой указатели. Операция присваивания копирует значение одного

указателя в другой, в результате оба указателя будут содержать один и тот же адрес памяти.

Операция получения адреса – одноместная, ее операнд может иметь любой тип, результатом является типизированный (в соответствии с типом операнда) указатель, содержащий адрес объекта-операнда.

Операция выборки – одноместная, ее операндом является обязательно типизированный указатель, а результат – данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя-операнда.

К указателю можно прибавить или вычесть целое число. Поскольку память имеет линейную структуру, прибавление к адресу числа даст адрес области памяти, смещенной на это число байт (или других единиц измерения) относительно исходного адреса.

Результат операций «указатель + целое» и «указатель – целое» имеет тип «указатель». Можно вычесть один указатель из другого (оба указателя-операнда при этом должны иметь одинаковый тип). Результат такого вычитания будет иметь тип целого числа со знаком. Его значение показывает, на сколько байт (или других единиц измерения) один адрес отстоит от другого.

Сложение указателей не имеет смысла. Поскольку программа оперирует относительными адресами и при разном исполнении может размещаться в разных областях памяти, сумма двух адресов будет давать разные результаты при разном исполнении. Смещение объектов друг относительно друга не зависит от адреса загрузки программы, поэтому результат операции вычитания указателей будет постоянным, и такая операция является допустимой.

Типизированные указатели. При объявлении типизированного указателя всегда должен быть определен тип объекта, адресуемого указателем. При определении типа-указателя используется базовый тип, перед которым ставится признак указателя «^». По соглашению, идентификаторы типа указателя и переменной-указателя начинаются с заглавной буквы P.

Описание типизированного указателя имеет вид:

type

P = ^t;

где P – идентификатор типа указателя; t – идентификатор (не определение) типа элемента данных, на который ссылается указатель.

Типизированный указатель может быть определен в разделе переменных. Например, следующие объявления:

var

ipt: ^Byte;

cpt: ^Char;

означают, что переменная ipt представляет адрес области памяти, в которой хранится целое число, а cpt – адрес области памяти, в которой хранится символ.

Физическая структура адреса не зависит от типа и значения данных, хранящихся по этому адресу. Тем не менее, указатели ipt и cpt имеют разный тип, и поэтому оператор присваивания:

cpt:=ipt;

недопустим.

Для типизированных указателей правильнее говорить не о едином типе данных «указатель», а о целом семействе типов: «указатель на целое», «указатель на символ» и т.д. Указатели могут быть определены и на более сложные, интегрированные структуры данных, и указатели на указатели.

Пусть имеется описание:

type

Tarr = **array**[1..20] **of** Real;

var

Pint: ^Integer;

Parr: ^Tarr;

Для такого описания Pint – переменная, указывающая на элемент целого типа (целое число), а Parr – указатель адреса первого элемента массива вещественных чисел. Сами указатели являются статическими переменными и располагаются в сегменте данных, а элементы, на которые ссылаются указатели, занимают место в динамической памяти (рис. 2.4).

Рассмотрим действия, которые можно выполнять с типизированными указателями. Значение переменной указательного типа можно задать процедурами по работе с динамической памятью, адресным оператором @, или операцией присваивания.

Указатели одного и того же типа можно сравнивать с помощью операций «=» и «<>». Переход от указателя к объекту ссылки производится с помощью операции разыменования, которая обозначается знаком ^, стоящим после имени указателя:

Pint^ – объект (целое число), на который указывает Pint;

Parr^ – массив вещественных чисел, на который ссылается переменная Parr.

2

Динамическая память

0-й элемент массива

. . .

2-й элемент массива

1-й элемент массива

Ц

ссылки

елое число

Статическая
память

Pint

Parr

Рис. 2.4. Размещение переменной типа Tarr в динамической памяти.

С объектами указателей можно производить действия как с обычными переменными соответствующего типа:

Pint^:=Pint^+3;

Parr^[i]:=0;

Переход от объекта к его адресу осуществляется операцией взятия адреса@ или функцией Addr. Если выполнить присваивания:

Pint:=@X;

или

Pint:=Addr(X);

то будет:

Pint^:=X

Таким образом, операция взятия адреса является обратной к операции разыменования.

Для динамического распределения памяти существуют стандартные процедуры. Процедура New(P) где P – указатель, позволяет выделить область памяти такого размера, в котором можно разместить величину базового типа. Указатель принимает значение адреса выделенной области.

Процедура Dispose(P), позволяет освободить область памяти, на которую указывает указатель P, для последующего использования. После выполнения процедуры значение указателя P становится неопределенным.

Процедура GetMem(P, Size) позволяет выделить в динамической памяти область размером Size байт, при этом адрес выделенной области присваивается переменной P.

Процедура FreeMem(P, Size) освобождает занятую область памяти с адресом, задаваемым оператором P и размером Size байт. После выполнения процедуры область памяти становится свободной, а значение указателя оказывается неопределенным.

Значение одного указателя можно присвоить другому. Указатели, ссылающиеся на объекты разных типов, сами являются разнотипными и для них недопустима операция присваивания.

Рассмотрим несколько типичных приемов работы с указателями. Пусть имеется описание:

type

PComplex= ^TComplex; { тип-указатель }

TComplex=**record**{ определение базового типа }

Re, Im: Real;

end;

AdrInt = ^Integer; { тип-указатель }

var

X: TComplex;

P1,P2,P3,P4: PComplex;

Adr:AdrInt;

тогда возможны следующие операции:

New(P1); { выделение памяти для типа TComplex }

New(Adr); { выделение памяти для типа Integer }

P2:=@X; { определение адреса переменной X }

P3:=P1; { присвоение значения другого указателя }

P4:=nil; { присвоение значения nil – «пустая» ссылка, не указывающая

ни на какой объект, «адресный ноль» }

X:=P1^; { переменной X присваивается значение

элемента, на который указывает P1 }

P3^:=X; { элементу, на который указывает P3,

присваивается значение переменной X }

X:=@Y; { X – адрес параметра Y }

После работы с указателями, следует обязательно освободить память, занимаемую теми объектами, на которые они указывают:

Dispose(P1); { освобождение памяти от данных, на которые указывает P1 }

New(Adr);

Нетипизированные указатели. В языке Паскаль существует стандартный тип-указатель Pointer, не связанный с базовым типом. Нетипизированный указатель совместим с любым другим типом-указателем. Переменная типа Pointer содержит адрес объекта неопределенного типа.

Пусть имеются следующие описания:

var

p1: ^Char;

p2: ^Boolean;

pp:Pointer;

Использование указателей типа Pointer позволяет динамически размещать с одного адреса данные различных типов. Напомним, присваивание значения одного указателя другому возможно только между указателями объектов идентичного типа. В примере символьный и булевый тип занимает одинаковый объем памяти (1 байт), однако следующая запись некорректна:

p1:=p2;

Нетипизированный указатель совместим по присваиванию с указателем любого типа, поэтому присваивание между p1 и p2 можно выполнить опосредованно:

pp:=p2;

p1:=pp;

Работа с нетипизированными указателями существенно ограничена: они могут использоваться только для сохранения адреса, а обращение по адресу, задаваемому таким указателем, невозможно (неприменима операция разыменования). Для этого необходимо привести нетипизированный указатель к конкретному типу (например, через операцию присваивания), а затем применить разыменование.

4.3. Объектные типы данных

Поскольку в основу структурного программирования положены управляющие структуры, структурный подход дает хорошие результаты при сложном управлении и простых структурах данных. В свою очередь объектный подход базируется на структурах данных и предпочтителен, когда сложность построения алгоритмов заключена в выборе организации данных. Объектные типы данных относятся к динамическим структурам, однако, ввиду их большой важности и применения в последующем изложении, они будут рассмотрены отдельно.

4.3.1. Объявление и реализация классов

Для объявления классов (*объектных типов*) используется зарезервированное слово class. Определим некоторый класс графических примитивов TFigure следующим образом:

TFigure=**class**

fColor:Byte;

fThickness: Byte;

fCanvas: TCanvas;

procedure SetColor(Value: Byte);

```
procedure SetThickness(Value: Byte);

procedure PrepareCanvas;

end;
```

По принятому соглашению имена классов начинаются с заглавной буквы «T», имена полей данных начинаются с буквы «F», и поля класса объявляются до методов. Класс объединяет данные, представленные *атрибутами* (*полями*) и *алгоритмы* (*методы*) по их обработке.

В примере к полям данных класса относятся: поле fColor, хранящее код цвета, поле fThickness, задающее толщину линий и поле fCanvas, представляющее полотно, на котором будет происходить отображение графических примитивов.

Методы класса определяют действия, выполняемые над данными. Их совокупность характеризует функциональный аспект поведения класса. Методы представляют собой процедуры и функции, принадлежащие классу. К методам класса относится метод PrepareCanvas, выполняющий подготовку полотна к работе и два метода задания значений полей данных – SetColor и SetThickness.

Таким образом, в одной информационной структуре TFigure оказались объединены как исходные параметры, так и необходимые средства по выполнению их реализации. Такое объединение (сокрытие) данных и методов в качестве собственных ресурсов класса получило название *инкапсуляции*.

Для окончательного оформления шаблона требуется поместить класс в интерфейсный раздел модуля и дать реализацию всех методов:

```
unit figures;

Interface

type

TFigure = class
```

```
fColor: Byte;
fThickness: Byte;
fCanvas: TCanvas;
procedure SetColor(Value: Byte);
procedure SetThickness(Value: Byte);
procedure PrepareCanvas;
end;
Implementation
procedure TFigure.SetColor(Value: Byte);
begin
if fColor <> Value then
fColor:=Color;
end;
procedure TFigure.SetThickness(Value: Byte);
begin
if fThickness <> Value then
fThickness:=Value;
end;
procedure TFigure.PrepareCanvas;
```

begin

{ Подготовка полотна для рисования }

end;

end.

Методы SetColor и SetThickness выполняют присвоение внутреннему полю fColor и fThickness значения в том случае, если текущее значение отличается от передаваемого. К полям класса никогда не следует обращаться напрямую, а только посредством специальных методов, обеспечивающих корректность выполнения операции присваивания.

Теперь объявим переменную f класса TFigure:

var

f:TFigure;

Переменную f называют *экземпляром класса*, *объектной ссылкой* или просто *объектом*. Через объект f возможен доступ к методам и полям класса. Однако для начала необходимо создать сам объект. Для этого необходимо вызвать специальную процедуру Create, называемую *конструктором*:

f:=TFigure.Create;

Конструктор не объявлен в классе TFigure, однако присутствует в нем от класса TObject благодаря специальному механизму наследования классов друг от друга. В результате будет выделена область памяти в размере, необходимом для хранения объекта f. Обратите внимание, конструктор вызывается с помощью ссылки на тип, а не на экземпляр типа, в отличие от методов, которые всегда вызываются с помощью ссылки на экземпляр. Связано это с тем, что объект f на момент вызова конструктора еще не создан.

После создания объекта с ним можно работать:

```
uses figures;

var

f: TCircle;

begin

f:=TCircle.Create;

f.SetColor($FF);

f.SetThickness(1);

f.PrepareCanvas;

f.Free;

end.
```

После выполнения методов объект f следует удалить, чтобы он не занимал места в памяти. Удаление выполняет метод Destroy, определенный в классе TObject (базовом классе), но лучше использовать Free, т.к. он инкапсулирует вызов Destroy: в начале определяется, существует ли объект и только затем выполняется вызов Destroy. В противном случае метод Free ничего не делает.

Класс Figure можно модифицировать, например, явно добавить к методам конструктор Create с помощью зарезервированного слова constructor и деструктор Destroy с помощью зарезервированного слова destructor:

```
TFigure = class

...

constructor Create;

destructor Destroy;
```

end;

В конструкторе присваиваются полям начальные значения и создается объект полотна:

constructor TFigure.Create;

begin

fColor:=$FF;

fThickness:=1;

fCanvas:=TCanvas.Create;

end;

В *деструкторе* обычно выполняются действия, связанные с освобождением задействованных в течение работы объекта ресурсов:

destructorTFigure.Destroy;

begin

{ Освобождение ресурсов, используемых в работе объекта }

fCanvas.Free;

end;

4.3.2. Директивы видимости

Ключевая идея объектного подхода заключается в *сокрытии информации*. В хорошо спроектированном классе никакие изменения внутри не должны отражаться на пользователях данного класса. *Директивы видимости*, используемые в объектной модели языка Паскаль, позволяют ограничить доступность элементов класса.

Существует четыре уровня сокрытия информации, которые задаются директивами Private (закрытый), Public (общедоступный), Protected (защищенный) и Published (публикуемый). Наиболее строгой директивой является Private, которая ограничивает видимость атрибутов и методов класса тем модулем, в котором расположен данный класс. Директива Public наоборот, наименее строга, и обеспечивает доступ к объявлениям класса из любого модуля, имеющего доступ к данному модулю.

Директива Protected внутри модуля, в котором объявлен класс, действует аналогично директиве Private, а извне модуля – аналогично директиве Private. Однако вне модуля доступ к защищенным полям данных и методов все же возможен внутри производного класса. При создании классов директива Private должна использоваться только для полей и методов, действительно зависящих от класса. Любые поля и методы, доступ к которым из производных классов может потребоваться, должны быть объявлены в разделе Protected.

Для директивы Published правила видимости аналогичны Public, но разница между ними заключается в том, что Published указывает компилятору добавлять *информацию о типе времени выполнения* (Run-Time Type Information, RTTI) для всех объявлений этого раздела. Информация RTTI необходима инспектору объектов при работе с компонентами на форме в среде Borland® Delphi.

Распределим методы и поля по уровням доступности:

TFigure = **class**

private

fColor: Byte;

fThickness: Byte;

fCanvas: TCanvas;

public

constructor Create;

destructor Destroy; override;

procedure SetColor(Value: Byte);

procedure SetThickness(Value: Byte);

procedure PrepareCanvas;

end;

4.3.3. Свойства классов

Другим способом сохранения целостности объекта является применение свойств. ***Свойства* в объектной модели языка Паскаль предназначены для обеспечения доступа к скрытым атрибутам класса**. Обратимся к классу TFigure и введем свойства:

TFigure = **class**

private

fColor: Byte;

fThickness: Byte;

fCanvas: TCanvas;

protected

procedure SetColor(Color: Byte);

procedure SetThickness(Thickness: Byte);

property Color: Byte **read** fColor **write** SetColor;

property Thickness: Byte **read** fThickness **write** SetThickness;

public

constructor Create;

destructor Destroy;

procedure PrepareCanvas;

end;

Для определения свойств используется зарезервированное слово Property, вслед за которым задается имя свойства, затем его тип и *спецификаторы чтения-записи*. По принятому соглашению все методы доступа начинаются с приставки Get и Set для получения и установки значений полей соответственно. Возможны три комбинации для режима чтения-записи: *запись и чтение* полей (наличие директив read и write), *только запись* (write only) и *только чтение* (read only).

В примере свойство Color предназначено для модификации и чтения поля fColor, а свойство fThickness – поля fThickness. Чтение будет производиться непосредственно путем обращения к полям, а запись – с помощью методов записи SetColor и SetThickness.

Доступ к свойствам похож на обычный доступ к полям данных, однако компилятор всегда транслирует обращения к свойствам в вызовы соответствующих методов доступа. Например, присваивание:

Color:=16;

будет транслировано в вызов:

SetColor(16);

Таким образом, свойства предоставляют пользователю интерфейс, полностью скрывающий реализацию объекта, и обеспечивают контроль доступа к нему со стороны внешних объектов.

4.3.4. Структурированная обработка ошибок

В объектной модели языка Паскаль введено понятие *исключений* – ошибок времени выполнения, которые могут возникать как при некорректных действиях пользователя, так и в случае неудачного выполнения различных действий (файловых операций, запроса ресурсов операционной системы и т.д.). Средства обработки исключений позволяют упростить задачу корректного выхода из подпрограмм, их вызвавших.

Базовым классом для всех исключений является TException. *Объект исключения* содержит информацию о типе исключения, например, деление на ноль (класс EZeroDivide), недостаточность системных ресурсов (класс EOutOfMemory), ошибка ввода-вывода (класс EInOutError). Разветвленная структура исключений позволяет фиксировать только интересующие типы. Например, ошибки переполнения или деления на ноль относятся к классу EMathError. Поэтому, чтобы не обрабатывать все особые ситуации чисел с плавающей точкой, можно локализовать только интересующие типы, входящие в данную подгруппу.

Существуют два типа *защищенных блоков*, которые позволяют изменить распространение исключений. К ним относятся конструкции вида try…except…end и try…finally…end. Структурному блоку except…end передается управление только в случае появления исключения, когда требуется немедленно отреагировать на возникшую внештатную ситуацию:

try

{ охраняемый блок операторов }

except

{ Блок реакции с обработчиками особой ситуации }

end;

Внутри блока except создаются обработчики особых ситуаций для классов исключений. Обработчик имеет следующий формат:

on<класс особой ситуации>**do**

begin

{ код обработки особой ситуации }

end;

Объект исключения содержит текстовое описание исключения в свойстве Message и адрес исключения, доступный по методу ExceptAddr.

Содержимое finally…end выполняется в любом случае для гарантированного освобождения выделенных ресурсов:

try

{ охраняемый блок операторов }

finally

{ Блок реакции с кодом завершения }

end;

Исключения реализуются через стек вызова (список вложенных подпрограмм), и имеют ассоциированную с ним область действия. *Необработанные локально* (в рамках *процедуры-источника*), исключения передается вверх по стеку вызова к следующему блоку, который сможет его обработать. Если исключение не обрабатывается пользователем, оно передается в *глобальный обработчик исключений*.

Поскольку исключение по возможности следует анализировать на максимально низком уровне, то оно в основном осуществляется в блоке try…except…end. Использовать эту конструкцию следует в

случае, когда при возникновении исключения требуется выполнение неотложных действий.

Обработчики исключений схожи с виртуальными методами, т.к. их можно использовать для переопределения или дополнения обработки. Если требуется активизировать стандартную обработку исключения после выполнения обработке в блоке except, можно повторно сгенерировать данное исключение с помощью директивы Raise.

Общий синтаксис повторного возбуждения исключения таков:

try

{ Охраняемые операторы }

except

on ESomeException **do**

begin

{ Локальная обработка особой ситуации }

raise; { повторное возбуждение исключения }

end;

end;

Схему структурированной обработки исключений можно представить в следующем виде:

try

{ получение дескриптора запрашиваемого ресурса }

try

{ выполнение потенциально опасных операций }

...

finally

{ Блок реакции с кодом завершения; освобождение ресурса }

end;

{ Блок реакции с обработчиками особой ситуации }

except

{ выделение требуемых классов исключений }

on< класс особой ситуации 1 >**do**

on< класс особой ситуации 2 >**do**

...

on< класс особой ситуации N >**do**

...

{ если требуется, повторное возбуждение исключительной ситуации }

raise;

end;

4.3.5. Применение объектов

В последующем изложении во **многих примерах будут рассматриваться особенности выполнения различных алгоритмов не над фактическими данными, а над указателями, задающими расположение этих данных. Необходимость этого состоит в отделении особенностей физического и логического представления данных от обрабатываемых их алгоритмов**. Разделяя данные и

средства манипулирования ими, мы будем больше основное внимание уделять характеристикам алгоритмов, а не заниматься ненужной разработкой или оптимизацией методов для числовых, строковых и любых другим типов данных.

Поскольку фактически будет проводиться работа с указателями, имеет смысл использовать структуры данных, которые хранят элементы в форме указателей. Таким образом, совокупность обрабатываемых данных должна быть определенным образом организована посредством своих указателей.

Класс TList предоставляет возможность организовать работу со списком разнотипных объектов посредством указателей. Класс позволяет добавлять, удалять, перегруппировывать, сортировать и получать доступ к объектам, на которые указывают элементы списка.

Для доступа к элементу списка служит свойство класса:

```
List: PPointerList;
```

которое представляет структуру:

```
PPointerList = ^TPointerList;

TPointerList = array[0..MaxListSize-1] of Pointer;
```

Перед работой с объектом списка его необходимо создать путем вызова конструктора:

```
var aList: TList;

...

aList:=TList.Create;

...

try
```

...

finally

aList.Free;

end;

Свойство Items позволяет выполнить обращение к элементу массива через методы чтения- записи для проверки на выход за диапазон. Свойство List выполняет прямое обращение без проверки и по скорости доступа выше. Например, для сохранения в объекте aList указателя p под первым порядковым номером следует выполнить обращение:

aList.Items[0]:=p;

или поскольку свойство Items является свойством по умолчанию:

aList[0]:=p;

Свойство Count содержит общее число элементов списка. Не все элементы списка содержат фактические указатели, некоторые могут содержать адресный ноль nil. Увеличение значения свойства Count приведет к добавлению в конец списка nil-элементов, а уменьшение – удаление элементов с конца списка. Метод Pack служит для удаления всех nil-элементов из списка и коррекции значения Count.

Свойство Capacity предназначено для управления динамической памятью при добавлении элементов. Чтение свойства позволит определить количество объектов, которые список может хранить без включения механизма перераспределения памяти. Добавление нового элемента приводит к автоматическому увеличению значения емкости. Для повышения быстродействия перед добавлением большого числа «тяжеловесных» объектов желательно предварительно задать свойство Capacity:

aList.Clear;

aList.Capacity:=Count;

for i:=1 **to** Count **do** List.Add(...);

Метод Add добавляет новый элемент, заданным своим указателем, в конец списка и возвращает его порядковый номер (индекс).

function Add(Item: Pointer): Integer;

Метод Clear удаляет все элементы из списка и устанавливает счетчик Count в 0.

Для удаления элемента из списка, заданного своим порядковым номером, служит метод Delete. После выполнения метода все элементы, следовавшие за удаленным, будут смещены, а значение счетчика уменьшено на единицу. Метод не освобождает память, ассоциированную с удаленным элементом.

procedureDelete(Index: Integer);

Метод Remove удаляет первый найденный элемент, заданный указателем, и возвращает порядковый номер элемента, предшествовавший удаленному. После выполнения метода все элементы, следовавшие за удаленным, будут смещены, а значение счетчика уменьшено на единицу.

function Remove(Item: Pointer): Integer;

Метод Extract удаляет элемент списка по указателю.

function Extract(Item: Pointer): Pointer;

Метод Exchange меняет местами содержимое двух элементов, заданных порядковыми номерами.

procedure Exchange(Index1, Index2: Integer);

Методы First и Last возвращают первый и последний элементы списка:

function First: Pointer;

function Last: Pointer;

Метод IndexOf возвращает порядковый номер первого вхождения элемента в список. При отсутствии элемента функция возвращает -1.

functionIndexOf(Item:Pointer):Integer;

Метод Insert помещает элемент в заданную позицию списка. Все элементы, начиная с заданного, смещаются вверх на один.

procedure Insert(Index: Integer; Item: Pointer);

Метод Move перемещает указатель с первой позиции на вторую.

procedure Move(CurIndex, NewIndex: Integer);

Метод Sort выполняет сортировку элементов списка в соответствии с функцией сравнения.

procedure Sort(Compare: TListSortCompare);

Прототип функции следующий:

TListSortCompare = **function**(Item1, Item2: Pointer): Integer;

Функция должна возвращать положительное целое, при Item1 < Item2, 0 при равенстве и отрицательное целое при Item1 > Item2.

При уничтожении объекта списка TList память, выделенная под элементы, не освобождается. В таких случаях говорят, что объект не владеет своими данными. В некотором роде это преимущество, поскольку можно быть уверенным в сохранности данных. Однако существует проблема, связанная с ручным освобождением элементов списка. Следующий код удаления некорректен:

for i:=0 **to** aList.Count-1 **do**

begin

```
TObject(aList[i]).Free;

aList.Delete(i);

end;
```

Ошибка заключается в том, что при очередном удалении элемента уменьшается общее их количество, а верхняя граница переменной цикла остается прежней. Правильнее было бы организовать обратный цикл и начать удалять с последнего элемента:

```
for i:=aList.Count-1 downto 0 do

begin

TObject(aList[i]).Free;

aList.Delete(i);

end;
```

4.4. Статические структуры данных

Статические структуры представляют структурированное множество **базовых структур**. Например, **вектор может быть представлен упорядоченным множеством чисел. Статические структуры отличаются отсутствием изменчивости, и память для них выделяется автоматически на этапе компиляции или при выполнении – в момент активизации того программного блока, в котором они определены.**

Ряд языков (PL/1, ALGOL-60) допускают размещение статических структур в памяти на этапе выполнения по явному требованию, но и в этом случае объем выделенной памяти остается неизменным до уничтожения структуры. Выделение памяти на этапе компиляции является столь удобным свойством статических структур, что в ряде задач их используют даже для представления объектов, обладающих изменчивостью. Например, когда размер массива неизвестен заранее, для него резервируется максимально возможный размер.

При физическом представлении статической структуре нередко ставится в соответствие **дескриптор**, или **заголовок**, который содержит общие сведения о структуре. Дескриптор хранится, как и сама физическая структура, состоит из полей, характер, число и размеры которых зависят от той структуры, которую он описывает и от принятых способов ее обработки.

В ряде случаев дескриптор необходим, т.к. выполнение операции доступа к структуре требует обязательного знания каких-либо ее параметров, которые хранятся в дескрипторе. Другие хранимые в дескрипторе параметры не являются необходимыми, но их использование позволяет сократить время доступа или обеспечить контроль правильности доступа к структуре.

Статические структуры в языках программирования связаны со структурированными типами. Структурированы типы теми средствами интеграции, которые позволяют строить структуры данных произвольной сложности. К таким **типам относятся: массивы, записи (в некоторых языках – структуры) и множества (реализованы не во всех языках).**

4.4.1. Векторы

Вектор **(одномерный массив) – структура данных с фиксированным числом однотипных элементов**. Каждый элемент вектора имеет уникальный в рамках заданного вектора номер. **Обращение к элементу вектора выполняется по имени вектора и номеру требуемого элемента.**

Элементы вектора размещаются в памяти в расположенных подряд (смежных) ячейках. Под элемент вектора выделяется количество байт памяти, определяемое базовым типом элемента вектора. **Необходимое число байтов памяти для хранения одного элемента вектора называется *слотом*. Размер памяти для хранения вектора определяется произведением длины слота на число элементов.** В языках программирования вектор представляется одномерным массивом:

< имя >: **array**[n..k]**of**< тип >;

где n-номер первого элемента, k-номер последнего элемента.

Представление в памяти вектора показано табл. 4.1. Здесь и далее смещение указано в байтах относительно начального адреса расположения вектора.

Табл. 4.1. Представление вектора в памяти.

Смещение	**+0**	**+SizeOf(тип)**	**…**	**+(k-n)*SizeOf(тип)**
Идентификатор	**Имя [n]**	**Имя [n+1]**	**…**	**Имя [k]**

В таблице приняты следующие обозначения:

@Имя - адрес вектора (адрес первого элемента вектора);

SizeOf(тип) - размер слота (количество байтов памяти для записи одного элемента вектора);

SizeOf(тип)·(k-n) - относительный адрес элемента с номером k (смещение элемента с номером k).

Пусть объявлен следующий вектор:

var v:**array**[-2..2]**of**real;

Представление вектора в памяти показано в табл. 4.2.

Табл. 4.2. Представление вектора v в памяти.

Смещение	**+0**	**+6**	**+12**	**+18**	**+24**
Идентификатор	**v[-2]**	**v[-1]**	**v[0]**	**v[1]**	**v[2]**

В языках, где память под массив выделяется на этапе компиляции (C, PASCAL, FORTRAN), при описании типа вектора граничные значения индексов должны быть определены. В языках, где память может

распределяться динамически (ALGOL, PL/1), значения индексов могут быть заданы во время выполнения программы.

Количество байт непрерывной области памяти, одновременно занятых вектором равно:

ByteSize = (k-n+1)·SizeOf(тип)

Обращение к i-тому элементу вектора выполняется по адресу вектора плюс смещение к данному элементу. Смещение i-ого элемента вектора равно:

ByteNumer= (i-n)·SizeOf(тип)

а его адрес:

@ByteNumber = @имя + ByteNumber

где @имя - адрес первого элемента вектора.

Например:

var v: **array**[5..10] **of** Word

Базовый тип элемента вектора Word, поэтому на каждый элемент выделяется по два байта. Смещения элементов относительно @v показано в табл. 4.3.

Табл. 4.3. Представление вектора v.

Смещение	+0	+2	+4	+6	+8	+10
Идентификатор	v[5]	v[6]	v[7]	v[8]	v[9]	v[10]

Данный вектор будет занимать в памяти: (10-5+1)·2 = 12 байт.
Смещение к элементу вектора с номером 8: (8-5) ·2 = 6. Адрес элемента с номером 8: @v + 6.

При доступе к вектору задается имя вектора и номер элемента вектора. Адрес i-го элемента может быть вычислен:

@Имя[i] = @Имя+i·SizeOf(тип)-n·SizeOf(тип) (4.1)

Такое вычисление не может быть выполнено на этапе компиляции, т.к. значение переменной i в это время еще неизвестно. Следовательно, вычисление адреса элемента должно производиться на этапе выполнения при каждом обращении к элементу вектора. Для этого, во-первых, должны быть известны параметры формулы (4.1), во-вторых, при каждом обращении должны выполняться две операции умножения и две операции сложения. Преобразовав формулу (4.1), получим:

@Имя[i] =A0+i·SizeOf(тип) (4.2)

где A0 = @Имя-n·SizeOf(тип)

Число хранимых параметров может быть сокращено до двух, а число операций – до одного умножения и одного сложения, т.к. значение A0 может быть вычислено на этапе компиляции и сохранено вместе с SizeOf(тип) в дескрипторе вектора.

Обычно в дескрипторе вектора сохраняются и граничные значения индексов. При каждом обращении к элементу вектора заданное значение сравнивается с граничными и программа аварийно завершается, если заданный индекс выходит за допустимые пределы.

Таким образом, информация, содержащаяся в дескрипторе вектора, позволяет сократить время доступа и обеспечить проверку правильности обращения. Но за эти преимущества приходиться платить быстродействием (обращения к дескриптору) и памятью (размещение самого дескриптора).

В языке C, например, дескриптор вектора отсутствует, точнее, не сохраняется на этапе выполнения. Индексация массивов в C обязательно начинается с нуля. Компилятор каждое обращение к элементу массива заменяет на последовательность команд, реализуя частный случай формулы (4.1) при n = 0.

4.4.2. Массивы

Массив – структура данных, характеризуемая:

- фиксированным набором однотипных элементов;
- каждый элемент имеет уникальный набор значений индексов.

Количество индексов определяют *мерность* массива, например, два индекса – двумерный массив, три индекса – трехмерный массив, один индекс – одномерный массив или вектор. Обращение к элементу массива выполняется по имени массива и значениям индексов для данного элемента.

Другое определение: **массив – вектор, каждый элемент которого – вектор. Синтаксис описания массива представляется в виде:**

< Имя >: **array** [n1..k1] [n2..k2]..[nn..kn]**of**< Тип >.

Для двумерного массива:

m:**array**[n1..k1] [n2..k2]**of**< Тип > или

m: **array** [n1..k1, n2..k2] **of** < Тип >

Двумерный массив можно представить в виде таблицы из (k1-n1+1) строк и (k2-n2+1) столбцов. Для двумерного массива, состоящего из (k1-n1+1) строк и (k2-n2+1) столбцов физическая структура представлена в табл. 4.4.

Табл. 4.4. Представление массива m.

Смещение	**+0**	**+SizeOf(тип)**	**…**	**+(k2-n2)*SizeOf(тип)**
Идентификатор	**m[n1,n2]**	**m[n1,n2+1]**	**…**	**m[n1,k2]**
			…	
	m[k1,n2]	**m[k1,n2+1]**	**…**	**m[n1,k2]**

+(k1-n1)*(k2-n2+1)*SizeOf(тип)	**+((k1-n1)*(k2-n2)+1)*SizeOf(тип)**	**…**	**+((k1-n1)*(k2-n2+1)+(k2-n2))*SizeOf(тип)**

Многомерные массивы хранятся в непрерывной области памяти. Размер слота определяется базовым типом элемента массива. Количество элементов массива и размер слота определяют размер памяти для хранения массива. Принцип распределения элементов массива в памяти определен языком программирования. Так в языке FORTRAN элементы распределяются по столбцам (быстрее меняется левые индексы), в PASCAL – по строкам (изменение индексов выполняется в направлении справа налево).

Количество байтов памяти, занятых двумерным массивом, определяется по формуле:

ByteSize = (k1-n1+1)·(k2-n2+1)·SizeOf(Тип) (4.3)

Адресом массива является адрес первого байта начального элемента массива.

Смещение элемента массива m[i1,i2] равно:

ByteNumber = [(i1-n1)·(k2-n2+1)+(i2-n2)]·SizeOf(Тип) (4.4)

а его адрес:

@ByteNumber = @m + ByteNumber

Например:

var m: **array**[3..5][7..8] **of** Word;

Базовый тип элемента Word требует два байта памяти, тогда таблица смещений элементов массива относительно @m будет следующей как

показано в табл. 4.5. Массив будет занимать в памяти: (5-3+1) · (8-7+1) · 2 = 12 байт

адрес элемента m[4,8] равен:

@m+((4-3) · (8-7+1)+(8-7)) · 2 = @m+6

Табл. 4.5. Представление массива m.

Смещение	**Идентификатор**
+0	m[3,7]
+2	m[3,8]
+4	m[4,7]
+6	m[4,8]
+8	m[5,7]
+10	m[5,8]

Таким образом, преимущество использования массивов заключается в быстром вычислении адресов элементов. Независимо от значения элемента, алгоритм вычисления будет одним и тем же. Другими словами, получение доступа к элементу с индексом n принадлежит к классу операций O(1) и не зависит от величины n.

Недостаток массивов связан с ресурсоемкими операциями вставки и удаления элементов. При вставке элемента с индексом i в общем случае следует все элементы с индексами с i по n переместить на одну позицию, чтобы освободить место под новый элемент. Фактически, следует выполнить следующий код:

```
{ сначала освободить место под новый элемент }

for j:=n downto i do

m[j+1]:=m[j];
```

```
{ вставить новый элемент в позицию с индексом i}

m[j]:=a;

{ увеличить значение длины массива на единицу }

Inc(n);
```

Чем больше количество элементов, тем больше времени потребуется на выполнение операции. Следовательно, операция вставки элемента в массив принадлежит к классу O(i). Та же ситуация и для удаления элемента из массива. Но в этом случае удаление элемента с индексом i означает перемещение всех элементов, начиная с индекса i+1 и до конца массива, на одну позицию к началу массива, чтобы «закрыть» образовавшуюся «дыру». Удаление так же принадлежит к классу O(i).

```
{ удалить элемент, переместив следующие

за ним элементы на одну позицию вперед }

for j:=i+1 to n do

m[j-1]:=m[j];

{ уменьшить значение длины массива на единицу }

Dec(n);
```

Открытые массивы. В языке Паскаль доступны для работы массивы открытого типа. Приведем пример объявления открытого массива, элементами которого являются вещественные числа:

```
var m: array of Real;
```

В отличие от приведенных выше способов такое объявление не резервирует память под массив, т.к. размеры его заранее не известны. Память будет выделена в процессе выполнения с помощью процедуры SetLength. В следующем примере выделяется память для хранения ста вещественных чисел, индексируемых от 0 до 99:

SetLength(m, 100);

Для массивов открытого типа не следует использовать процедуры управления динамической памятью и символ разыменования «^». Для освобождения памяти, отводимой под открытый массив, переменной следует присвоить значение nil или передать ее в качестве аргумента процедуре Finalize:

Finalize(m);

Процедура линеаризации. Основная операция физического уровня над массивом – доступ к заданному элементу. Как только реализован доступ к элементу, над ним может быть выполнена любая операция, имеющая смысл для типа данных, которому соответствует элемент. Преобразование логической структуры массива в физическую называется *процессом линеаризации*, в ходе которого многомерная логическая структура преобразуется в одномерную физическую.

В соответствии с формулами (4.3), (4.4) и по аналогии с вектором (4.1), (4.2) для двумерного массива с границами изменения индексов [B(1)…E(1)] [B(2)…E(2)], размещенного в памяти по строкам, адрес элемента с индексами [I(1),I(2)]может быть вычислен как:

$$Addr[I(1), I(2),..., I(n)] = Addr[B(1), B(2),..., B(n)] - SizeOf(Tun) \cdot \sum_{m=1}^{n} [B(m) \cdot D(m)] + SizeOf(Tun) \cdot \sum_{m=1}^{n} [I(m) \cdot D(m)] \quad (4.5)$$

Обобщая (4.5) для массива произвольной размерности *Ma*s[B(1)…E(1)] [B(2)…E(2)]… [B(*n*)…E(*n*)]:

$$Addr[I(1), I(2),..., I(n)] = Addr[B(1), B(2),..., B(n)] - SizeOf(Tun) \cdot \sum_{m=1}^{n} [B(m) \cdot D(m)] + SizeOf(Tun) \cdot \sum_{m=1}^{n} [I(m) \cdot D(m)] \quad (4.6)$$

где *D(m*) зависит от способа размещения массива.

При размещении по строкам:

$$D(m) = \lfloor E(m+1) - B(m+1) + 1 \rfloor \cdot D(m+1)$$

,

где *m=n-1,…,1* и *D(n)=1*

при размещении по столбцам:

$$D(m) = \lfloor E(m-1) - B(m-1) + 1 \rfloor \cdot D(m-1)$$

,

где *m=2,…,n* и *D(1)=1*.

При вычислении адреса элемента наиболее сложным является вычисление третьей составляющей формулы (4.6), т.к. первые две не зависят от индексов и могут быть вычислены заранее. Для ускорения вычислений множители *D(m)* могут быть вычислены заранее и сохраняться в дескрипторе массива.

Дескриптор массива содержит:

- начальный адрес массива *addr*[I(1),I(2).. I(*n*)];
- число измерений в массиве n;
- постоянную составляющую формулы линеаризации (первые две составляющие 4.6);
- для каждого из n измерений массива: значения граничных индексов *B(i), E(i)* и множитель формулы линеаризации *D(i)*.

Специальные массивы. На практике встречаются массивы, которые в силу определенных причин должны записываться в память не полностью, а частично. Это особенно актуально для массивов настолько больших размеров, что для их хранения в полном объеме памяти может быть недостаточно. К таким массивам относятся *симметричные* и *разреженные* массивы.

Двумерный массив, в котором количество строк равно количеству столбцов называется квадратной матрицей. *Симметричный массив* – квадратная матрица, у которой элементы, расположенные симметрично относительно главной диагонали, попарно равны друг другу.

Для симметричной матрицы порядка n в физической структуре достаточно отобразить не n^2, а лишь $n \cdot (n+1)/2$ элементов. В памяти необходимо представить только верхний (включая диагональ) треугольник квадратной логической структуры. Доступ к треугольному массиву организуется таким образом, чтобы можно было обращаться к любому элементу исходной логической структуры, в том числе и к элементам, значения которых хотя и не представлены в памяти, но могут быть определены на основе значений симметричных им элементов.

Для работы с симметричной матрицей должны быть определены процедуры:

- преобразования индексов матрицы в индекс вектора;
- формирования вектора и записи в него элементов верхнего треугольника элементов исходной матрицы;
- получения значения элемента матрицы из ее упакованного представления. Обращение к элементам исходной матрицы выполняется опосредованно, через указанные функции.

Разреженный массив– массив, большинство элементов которого равны между собой, так что хранить в памяти достаточно лишь небольшое число значений отличных от основного (фонового) значения остальных элементов. Различают два типа разреженных массивов:

- массивы, в которых местоположения элементов со значениями, отличными от фоновых, могут быть математически описаны;
- массивы со случайным расположением элементов.

В случае работы с разреженными массивами вопросы размещения их в памяти реализуются на логическом уровне с учетом их типа. К первому типу массивов относятся массивы, у которых местоположения элементов со значениями отличными от фонового, могут быть математически описаны, т.е. в их расположении есть какая-либо

закономерность. Элементы, значения которых являются фоновыми, называют нулевыми; элементы, значения которых отличны от фонового, – ненулевыми. Нужно помнить, что фоновое значение не всегда равно нулю.

Ненулевые значения хранятся, как правило, в одномерном массиве, а связь между местоположением в исходном, разреженном, массиве и в новом, одномерном, описывается математически с помощью формулы, преобразующей индексы массива в индексы вектора.

Для работы с разреженным массивом должны быть определены функции:

- преобразования индексов массива в индекс вектора;
- получения значения элемента массива из ее упакованного представления по двум индексам (строка, столбец);
- записи значения элемента массива в ее упакованное представление по двум индексам.

Ко второму типу массивов относятся массивы, у которых местоположения элементов со значениями, отличными от фоновых, не могут быть математически описаны, т.е. в их расположении нет закономерности.

4.4.3. Множества

Множеством **называется неупорядоченная совокупность элементов порядкового типа, элементами которого могут быть величины типов ShortInt, Byte, Char, а также интервального и перечисляемого.** Элементы множества могут принимать все значения базового типа, их количество не должно превышать 256. В табл. 4.6 перечислены операции, которые определены над множествами S, S1, S2.

Табл. 4.6. Операции над множествами.

Обозначение	Название	Результат

e in S	Принадлежность элемента e множеству S.	Логический.
S1 = S2	Равенство S1 и S2.	- « -
S1 <> S2	Неравенство S1 и S2.	- « -
S1 <= S2	Включение S1 в S2.	- « -
S1 => S2	Включение S2 в S1.	- « -
S1 + S2	Объединение S1 и S2.	Множество элементов, принадлежащих S1 или S2.
S1 - S2	Разность S1 и S2.	Множество элементов S1, не принадлежащих S2.
S1 * S2	Пересечение S1 и S2.	Множество элементов, принадлежащих S1 и S2.

В языке Паскаль для задания типа-множества есть зарезервированные слова set и of с помощью которых следует указать элементы множества в виде перечисления или диапазона:

type

< имя > = **set of** < тип >;

где

< имя > – идентификатор типа множества;

< тип > – идентификатор или определение типа элементов.

Примеры определения множеств:

type

TNumber=**setof**0..9;

TOperation = **set of** (Plus, Minus, Mult, Divide);

TSymbols = **setof**Char;

Тип множества может быть определен и при объявлении переменных:

var

Symbols: TSymbols;

Operation: TOperation;

Digits:**setof**0..9;

Значением переменной-множества может быть любое сочетание элементов, заданных ее типом. Чаще всего значение переменной-множества задается конструктором множества, т.е. перечислением его элементов в квадратных скобках:

Digits:=[1, 3, 5, 7, 9]; { нечетные цифры }

Symbols:=[‘A’..’Z’, ‘a’..’z’]; { латинский алфавит }

Символьные множества хранятся в памяти также как и числовые. Разница лишь в том, что хранятся не числа, а коды ASCII символов. Множество может быть пустым, т.е. не иметь ни одного элемента: Symbols:= [ ].

Множество в памяти хранится как массив битов, в котором каждый бит указывает, является ли элемент принадлежащим объявленному множеству или нет. Число байтов, выделяемых для данных типа множество, вычисляется по формуле:

ByteSize = (max **div** 8)-(min **div** 8)+1

где max и min – верхняя и нижняя границы базового типа данного множества.

Номер байта для элемента E вычисляется по формуле:

ByteNumber = (E **div** 8)-(min **div** 8)

а номер бита внутри байта по формуле:

BitNumber = E **mod**8

Например, переменная следующего типа:

type

TVideo= (MDA,CGA,HGC,EGA,EGAm,VGA,VGAm,SVGA,PGA,XGA);

в памяти будет занимать:

ByteSize = (9 **div**8)-(0**div**8)+1 = 2 байта

4.4.4. Записи

***Запись* – конечное упорядоченное множество полей, характеризующихся различным типом данных**. Записи являются удобным средством представления программных моделей реальных объектов предметной области, и, как правило, каждый такой **объект обладает набором свойств (*полей данных*), характеризуемых данными различных типов.**

Поля записи могут иметь простой или структурный тип.

Приведем примеры записей:

type

{ комплексное число }

Complex = **record**

Re, Im: Real;

end;

{ календарная дата }

TDate = **record**

Day: 1..31;

Month: (Jan, Feb, Mar, Apr,May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

Year: Word;

end;

и затем объявим переменные соответствующего типа:

var

Z: Complex;

Date: TDate;

Основная операция над записями – доступ к выбранному полю (операция квалификации). Практически во всех языках обозначение этой операции имеет вид:

< имя переменной-записи >.< имя поля >

Например, полям записи Z можно присвоить значения:

Z.Re:=5.2;

Z.Im:=-8.4;

При неоднократном обращении к одному и тому же полю записи или к различным полям одной записи удобно использовать оператор доступа:

with< имя переменной типа запись >**do**< оператор >

В этом случае можно не указывать имя записи, а только имена полей. С использованием оператора with можно записать:

withDate**do**

begin

Day:=18;

Month:=Jan;

Year:=2000;

end;

Большинство языков поддерживает операции, работающие с записью, как с единым целым, а не с отдельными ее полями: операция присваивания одной записи значения другой однотипной записи и сравнения двух однотипных записей на равенство/неравенство. Это позволяет обрабатывать в одной структуре данные разных типов. Например, если переменные A и B представляют собой записи одного типа, то можно использовать присваивание A:=B. При этом поля записи A получат значения соответствующих полей записи B.

В случаях, когда такие операции не поддерживаются языком явно (язык C), они могут выполняться над отдельными полями или записи могут копироваться и сравниваться как неструктурированные области памяти.

Поля записи в свою очередь могут быть записями, т.е. записи могут быть вложенными на несколько уровней. Для доступа к внутренним полям необходимо удлинять составное имя через точку (каждая точка раскрывает один уровень), либо использовать последовательно несколько операторов доступа with.

Пусть определена запись, описывающая некоторый объект:

type

{ сведения об объекте }

TItem = **record**

Name: **string**[20];

Tested: Boolean;

Date: TDate;

end;

Определим переменную типа TItem:

varItem: TItem;

Тогда дата проверки объекта задается составным именем:

Item.Date.Day:=18;

Item.Date.Month:=Jan;

Item.Date.Year:=2000;

или с использованием двух операторов with:

with Item **do**

with Date **do**

begin

Day:=18;

Month:=Jan;

Year:=2000;

end;

Упакованные записи. Доступ к отдельному полю записи эффективен, т.к. величина смещения каждого поля от начала записи постоянна и известна во время компиляции. Так как для поля иногда нужен объем памяти, не кратный размеру слова (или двойного слова), компилятор может «увеличить» запись так, чтобы каждое поле было выровнено по границе слова (или двойного слова), поскольку доступ к невыровненному полю по границе слова (или двойного слова) менее эффективен.

Например, объявление следующего типа:

TSomeStruct=**record**

c1:Char; { 1 байт, пропустить 1 байт }

w1:Word; { 2 байта }

c2:Char; { 1 байт, пропустить 1 байт }

w2: Word; { 2 байта }

end;

приведет к выделению четырех слов для каждой записи таким образом, чтобы поля были выровнены по границе слова, в то время как следующее объявление экономит память:

TSomeStruct = **record**

w1: Word; { 2 байта }

w2: Word; { 2 байта }

c1: Char; { 1 байт, пропустить 1 байт }

c2: Char; { 1 байт, пропустить 1 байт }

end;

По умолчанию для быстрого доступа элементы массива и поля записей выравниваются по границе слова или двойного слова. Для устранения такого выравнивания можно использовать зарезервированное слово packed перед словом record или array. Это приведет к снижению скорости доступа, но одновременно уменьшит объем занимаемой структурой места в памяти.

Полем записи может быть другая запись, но не такая же, что связано, прежде всего, с тем, что компилятор должен выделить для размещения записи память.

Пусть, определена запись вида:

Rec = **record**

f1: Word;

f2: Char;

f3: Rec;

end;

Для поля f1 будет выделено 2 байта, для поля f2 – 1 байт, а поле f3 – запись, которая в свою очередь состоит из f1 (2 байта), f2 (1 байт) и f3 и т.д. Компилятор, встретив подобное описание, выдает сообщение о нехватке памяти. Однако полем записи вполне может быть указатель на другую такую же запись: размер памяти, занимаемый указателем известен, и проблем с выделением памяти не возникает. Подобный прием широко используется для установления связей между однотипными записями.

PRec = ^Rec

Rec = **record**

f1: Word;

f2: Char;

f3: PRec;

end;

Записи с вариантами. В ряде прикладных задач можно столкнуться с группами объектов, чьи наборы свойств перекрываются лишь частично. Обработка таких объектов производится по одним и тем же алгоритмам, если обрабатываются общие свойства объектов, или по разным – если обрабатываются специфические свойства.

Можно описать все группы единообразно, включив в описание все наборы свойств для всех групп, но такое описание будет неэкономичным с точки зрения расходуемой памяти и неудобным с логической точки зрения. Если же описать каждую группу собственной структурой, теряется возможность обрабатывать общие свойства по единым алгоритмам.

Для задач подобного рода некоторые языки программирования (C, PASCAL) предоставляют записи с вариантами. Запись с вариантами состоит из двух частей. В первой части описываются поля, общие для всех групп объектов, моделируемых записью. Среди полей обычно бывает поле, значение которого позволяет идентифицировать группу, к которой данный объект принадлежит и, следовательно, какой из вариантов второй части записи должен быть использован при обработке.

Вторая часть записи содержит описания непересекающихся свойств – для каждого подмножества таких свойств – отдельное описание. Язык программирования может требовать, чтобы имена полей-свойств не повторялись в разных вариантах (PASCAL), или же требовать именования каждого варианта (C). В первом случае идентификация поля, находящегося в вариантной части записи при обращении к нему ничем не отличается от случая простой записи:

< имя переменной-записи >.< имя поля >

Во втором случае идентификация немного усложняется:

< имя переменной-записи >.< имя варианта >.< имя поля >

Рассмотрим использование записей с вариантами. Пусть требуется размещать на экране простые геометрические фигуры – круги, прямоугольники, треугольники. Для «базы данных», которая будет описывать состояние экрана, удобно представлять все фигуры однотипными записями.

Для любой фигуры описание должно включать координаты некоторой опорной точки (центра, правого верхнего угла, одной из вершин) и код цвета. Другие параметры построения будут разными для разных фигур: для круга – радиус; для прямоугольника – длины непараллельных сторон; для треугольника – координаты двух других вершин. Запись с вариантами для такой задачи в языке PASCAL выглядит так:

figure = **record**

ftype : Char; { тип фигуры }

x0,y0 :Word; { координаты опорной точки }

color : Byte; { цвет }

case fig_t: Char **of**

'C': (radius: Word); { радиус окружности }

'R': (len1, len2: Word); { длины сторон прямоугольника }

'T': (x1,y1,x2,y2: Word); { координаты двух вершин }

end;

Если определена переменная fig1 типа figure, в которой хранится описание окружности, то обращение к радиусу этой окружности будет

иметь вид: fig1.radius. Поле с именем fig_type введено для представления идентификатора вида фигуры, который, например, может кодироваться символами: «C» – окружность или «R» – прямоугольник, или «T» – треугольник.

Выделение памяти для записи с вариантами показано на рис. 4.1. Под запись с вариантами выделяется объем памяти, достаточный для размещения самого большого варианта. Если выделенная память используется для меньшего варианта, часть ее остается неиспользуемой. Общая для всех вариантов часть записи размещается так, чтобы смещения всех полей относительно начала записи были одинаковыми для всех вариантов.

Наиболее просто это достигается размещением общих полей в начале записи, но это не строго обязательно. Вариантная часть может и «вклиниваться» между полями общей части. Поскольку в любом случае вариантная часть имеет фиксированный (максимальный) размер, смещения полей общей части также останутся фиксированными.

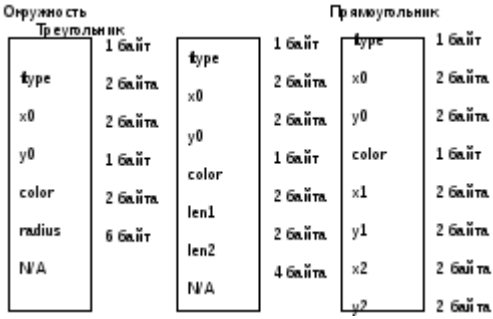


Рис. 4.1. Выделение памяти для записи с вариантами.

4.4.5. Таблицы

В предыдущем разделе было отмечено, что **полями записи могут быть интегрированные структуры – векторы, массивы, другие записи. Аналогично элементами векторов и массивов могут быть также интегрированные структуры. Одна из таких структур – *таблица*. С логической точки зрения таблица представляет**

вектор, элементами которого являются записи. Таблицы не имеют стандартного типа данных в языке Паскаль.

В таблицах доступ к элементам производится не по номеру (индексу), а по *ключу* – значению одного из свойств объекта, описываемого записью-элементом таблицы. Ключ идентифицирует данную запись во множестве однотипных. К ключу предъявляется требование уникальности в данной таблице. Ключ может включаться в состав записи и быть одним из ее полей, но может и не включаться, а вычисляться по положению записи. Таблица может иметь один или несколько ключей. Например, при интеграции в таблицу записей о студентах выборка может производиться как по личному номеру студента, так и по фамилии.

Основной операцией при работе с таблицами является операция доступа к записи по ключу. Она реализуется процедурой поиска. Поскольку поиск может быть значительно более эффективным в таблицах, упорядоченных по значениям ключей, довольно часто над таблицами необходимо выполнять операции сортировки. Такие операции рассматриваются в разделе 4.6.

Различают таблицы с фиксированной и переменной длиной записи. Очевидно, что таблицы, объединяющие записи идентичных типов, будут иметь фиксированные длины записей. Необходимость в переменной длине может возникнуть в задачах, подобных тем, которые рассматривались для записей с вариантами. Как правило, таблицы для таких задач и составляются из записей с вариантами, т.е. сводятся к фиксированной (максимальной) длине записи.

Значительно реже встречаются таблицы с действительно переменной длиной записи. Хотя в таких таблицах и экономится память, но возможности работы с такими таблицами ограничены, т.к. по номеру записи невозможно определить ее адрес. Таблицы с записями переменной длины обрабатываются только последовательно – в порядке возрастания номеров записей.

Доступ к элементу такой таблицы обычно осуществляется в два шага. На первом шаге выбирается постоянная часть записи, в которой содержится (в явном или неявном виде) длина записи. На втором шаге выбирается переменная часть записи в соответствии с ее длиной.

Прибавив к адресу текущей записи ее длину, получают адрес следующей записи.

4.4.6. Операции над статическими структурами

4.4.6.1. Алгоритмы поиска

Алгоритмы *поиска данных* и *сортировки*, выполняемые на статических структурах данных, являются типичными операциями логического уровня. Однако те же операции и алгоритмы применимы и к данным, имеющим логическую структуру таблицы, но физически размещенным в динамической памяти и на внешней памяти, а также к логическим таблицам любого физического представления, обладающим изменчивостью.

Функция сравнения. Само действие поиска элемента в наборе данных требует возможности отличать элементы друг от друга. Очевидно, сравнение числовых типов не вызывает трудности. В случае сравнения строк процедура усложняется. Можно выполнять сравнение чувствительное или нечувствительное к регистру, сравнение по локальным таблицам символов, когда оно выполняется на основе алгоритмов, специфичных для определенной страны или языка и т.д. При работе с объектами вообще не существует заранее определенной схемы сравнения за исключением сравнения указателей на эти объекты.

В таком случае лучше всего рассматривать процедуру сравнения в виде «черного ящика» – функции с четко определенным интерфейсом, которая в качестве входных параметров принимает указатели на два элемента и возвращает результат сравнения – первый элемент больше, меньше или равен второму.

В последующем изложении все описания алгоритмов даны для работы с классом TList, а функции сравнения имеют следующий тип:

```
{ Объявление прототипа функции сравнения }

TCompareFunc=function(aData1,aData2:Pointer):Integer;
```

Тип входных параметров определяет сама функция, и решает, привести ли их к определенному классу или просто к типу, который не является указателем. Пример реализации функции сравнения для целых типа LongWord приведен ниже.

```
{ Функция сравнения двух целых чисел, заданных своими указателями }

function CompareLongWord(aData1, aData2: Pointer): Integer;

begin

{ Значение первого элемента меньше значения второго }

if LongWord(aData1^) < LongWord(aData2^) then

Result:=-1else

{ Значения элементов равны }

if LongWord(aData1^) = LongWord(aData2^) then

Result:=0 else

{ Значение первого элемента больше значения второго }

Result:=1;

end;
```

Линейный поиск. Единственно возможным методом поиска элемента по значению ключа, находящегося в неупорядоченном наборе данных, является последовательный (линейный) просмотр каждого элемента, который продолжается до тех пор, пока не будет найден искомый элемент. Если просмотрен весь набор, но элемент не найден, искомый ключ отсутствует. Для последовательного поиска в среднем требуется (n+1)/2 сравнений и порядок алгоритма линейный O(n).

Программная иллюстрация линейного поиска в неупорядоченном массиве приведена ниже, где aList – исходный массив, aItem – искомый ключ; функция возвращает индекс найденного элемента или -1, если элемент отсутствует.

```
{ Линейный поиск в неотсортированном массиве }

function LineNonSortedSearch(aList: TList;

aItem: Pointer; aCompare: TCompareFunc): Integer;

var i: Integer;

begin

Result:=-1;

for i:=0 to aList.Count-1 do

if aCompare(aList.List[i],aItem) = 0 then

begin

Result:=i;

Break;

end;

end;
```

Последовательный поиск для отсортированного массива ничем не отличается от приведенного и имеет порядок алгоритма O(n). Однако алгоритм можно несколько улучшить. Если искомого элемента нет в массиве, поиск можно выполнить быстрее, т.к. итерации должны выполняться только до тех пор, пока не будет найден элемент, больший или равный искомому. Все последующие элементы будут больше искомого и поиск можно прекратить.

```
{ Линейный поиск в отсортированном массиве }

function LineSortedSearch(aList: TList;

aItem: Pointer; aCompare: TCompareFunc): Integer;

var i, CompareResult: Integer;

begin

Result:=-1;

{ Искать первый элемент, больший или равный искомому }

for i:=0 to aList.Count-1 do

begin

CompareResult:=aCompare(aList.List[i], aItem);

if CompareResult >= 0 then

begin

if CompareResult = 0 then

Result:=i else

Result:=-1;

Exit;

end;

end;

end;
```

Обратите внимание, функция сравнения вызывается только один раз при каждой итерации, т.к. временные параметры ее выполнения заранее не известны, и поэтому вызывать ее желательно следует как можно реже.

Бинарный поиск. Другим методом доступа к элементу является бинарный (двоичный или дихотомичный) поиск, который выполняется в заведомо упорядоченной последовательности элементов. Элементы массива должны располагаться в лексикографическом (символьные ключи) или численно (числовые ключи) возрастающем порядке. Для достижения упорядоченности может быть использован один из методов сортировки.

В методе сначала приближенно определяется запись в середине массива и анализируется значение ее ключа. Если оно слишком велико, то анализируется значение ключа, соответствующего записи в середине первой половины массива, и указанная процедура повторяется в этой половине до тех пор, пока не будет найдена требуемая запись. Если значение ключа слишком мало, испытывается ключ, соответствующий записи в середине второй половины массива, и процедура повторяется в этой половине.

Процесс продолжается до тех пор, пока не найден требуемый ключ или не станет пустым интервал, в котором осуществляется поиск. Чтобы найти элемент, в худшем случае требуется $\log_2(N)$ сравнений, что значительно лучше, чем при последовательном поиске. Программная иллюстрация бинарного поиска в упорядоченном массиве приведена ниже.

```
{ Двоичный поиск }

function BinarySearch(aList: TList;

aItem: Pointer; aCompare: TCompareFunc): Integer;

var L, R, M, CompareResult: Integer;

begin

{ Значения индексов первого и последнего элементов }
```

```
L:=0; R:=aList.Count-1;
while L <= R do
begin
{ Индекс среднего элемента }
M:=(L+R)div2;
{ Сравнить значение среднего элемента с искомым }
CompareResult:=aCompare(aList.List[M], aItem);
{ Если значение среднего элемента меньше искомого -
переместить левый индекс на позицию до среднего индекса }
if CompareResult < 0 then
L:=M+1 else
{ Если значение среднего элемента больше искомого -
переместить правый индекс на позицию после среднего индекса }
if CompareResult > 0 then
R:=M-1 else
begin
Result:=M;
Exit;
end;
```

end;

Result:=-1;

end;

Трассировка бинарного поиска ключа 275 в исходной последовательности: {75, 151, 203, 275, 318, 489, 524, 519, 647, 777} представлена в табл. 4.7.

Табл. 4.7. Трассировка бинарного поиска.

Итерация	**L**	**R**	**M**	**a[M]**
1	1	10	5	318
2	1	4	2	151
3	3	4	3	203
4	4	4	4	275

Алгоритм бинарного поиска можно представить несколько иначе, используя рекурсивное описание. В этом случае граничные индексы интервала L и R являются параметрами алгоритма. Рекурсивная процедура бинарного поиска представлена ниже. Для выполнения поиска необходимо при вызове функции задать значения ее формальных параметров L и R – 1 и n соответственно.

{ Рекурсивный алгоритм двоичного поиска }

functionBinaryRecurSearch(aList:TList;

aItem: Pointer; L,R: Integer; aCompare: TCompareFunc): Integer;

var i, CompareResult: Integer;

begin

```
{ Проверка ширины интервала }

if L > R then

Result:=-1 else

begin

i:=(L + R) div 2;

CompareResult:=aCompare(aList.List[i], aItem);

{ Ключ найден, возврат индекса }

if CompareResult = 0 then

Result:=i else

if CompareResult = -1 then

{ Поиск в правом подинтервале }

Result:=BinaryRecurSearch(aList,aItem,i+1,R,aCompare) else

{ Поиск в левом подинтервале }

Result:=BinaryRecurSearch(aList,aItem,L,i-1,aCompare);

end;

end;
```

4.4.6.2. Алгоритмы сортировки

Сформулируем задачу сортировки для общего случая следующим образом: имеется некоторое неупорядоченное входное множество ключей и требуется получить выходное множество тех же ключей, упорядоченных по возрастанию или убыванию. Из всех задач

программирования сортировка, возможно, имеет самый богатый выбор алгоритмов решения. Назовем некоторые факторы, влияющие на выбор алгоритма (помимо порядка алгоритма).

1. *Имеющийся ресурс памяти*: должно ли входное и выходное множество располагаться в разных областях памяти или выходное множество может быть сформировано на месте входного. В последнем случае имеющаяся область памяти должна в ходе сортировки динамически перераспределяться между входным и выходным множествами. Для одних алгоритмов это связано с большими затратами, для других – с меньшими.

2. *Исходная упорядоченность входного множества*: во входном множестве (даже если оно сгенерировано датчиком случайных чисел) могут попадаться уже упорядоченные участки. В предельном случае входное множество может оказаться уже упорядоченным. Одни алгоритмы не учитывают исходной упорядоченности и требуют одного и того же времени для сортировки любого (в том числе уже упорядоченного) множества данного объема, другие выполняются тем быстрее, чем лучше упорядоченность на входе.

3. *Временные характеристики операций*: при определении порядка алгоритма время выполнения обычно считается пропорциональным числу сравнений ключей. Ясно, что сравнение числовых ключей выполняется быстрее, чем строковых; операции пересылки, характерные для некоторых алгоритмов, выполняются тем быстрее, чем меньше объем записи, и т.п. В зависимости от обрабатываемых данных может быть выбран алгоритм, обеспечивающий минимизацию числа тех или иных операций.

4. *Сложность алгоритма*: простой алгоритм требует меньшего времени для его реализации и вероятность ошибки меньше. При промышленном изготовлении программного продукта требования соблюдения сроков разработки и надежности могут даже превалировать над требованиями эффективности функционирования.

5. *Устойчивость алгоритма:* все алгоритмы можно разделить на два типа – устойчивые и неустойчивые. К устойчивой сортировке относятся те алгоритмы, которые при наличии в исходном наборе данных нескольких равных элементов в отсортированном наборе оставляют их в том же порядке, в котором они были изначально. Например, в исходном наборе находятся три элемента с одинаковым значением 42. Пусть элемент А находится в позиции 12, элемент В – в позиции 234, С – 3456. После выполнения устойчивой сортировки они будут находиться в последовательности A,B,C. Неустойчивая сортировка не гарантирует сохранение исходного взаимного порядка, и расположение может быть B,C,A или C,A,B.

Разнообразие алгоритмов сортировки требует некоторой их классификации. Один из известных методов классификация ориентирован на логические характеристики применяемых алгоритмов. Согласно методу любой алгоритм сортировки использует одну из следующих четырех *стратегий* (или их комбинацию).

1. *Стратегия выборки*. Из входного множества выбирается следующий по критерию упорядоченности элемент и включается в выходное множество на место, следующее по номеру.

2. *Стратегия включения*. Из входного множества выбирается следующий по номеру элемент и включается в выходное множество на то место, которое он должен занимать в соответствии с критерием упорядоченности.

3. *Стратегия распределения*. Входное множество разбивается на ряд подмножеств (возможно, меньшего объема) и сортировка ведется внутри каждого такого подмножества.

4. *Стратегия слияния*. Выходное множество получается путем слияния небольших упорядоченных подмножеств.

Другим методом классификации является группирования алгоритмов по скорости выполнения. Именно такой метод будет использован при дальнейшем изложении, т.к. он в наибольшей степени отвечает практической применимости тех или иных алгоритмов. Алгоритмы рассмотрены для случая упорядочения по возрастанию ключей. В

большинстве примеров в качестве процедуры сортировки будет использоваться следующий прототип:

{ Прототип процедуры сортировки }

TSortProc=**procedure**(aList:TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

Первым параметром следует список, затем границы сортируемой области и функция сравнения.

4.4.6.2.1.Самые медленные алгоритмы сортировки

К первой группе отнесем самые медленные алгоритмы, принадлежащие к классу $O(n^2)$, хотя некоторые из них в лучших случаях могут дать высокие показатели производительности.

Сортировка простой выборкой реализует сформулированную стратегию выборки. В реализации алгоритма возникает проблема значения ключа «пусто» после выбора элемента. Часто используют в качестве такового некоторое заведомо отсутствующее во входной последовательности значение, например, максимальное из теоретически возможных. Другой, более строгий подход – создание отдельного вектора, каждый элемент которого имеет логический тип и отражает состояние соответствующего элемента входного множества («истина» – «непусто», «ложь» – «пусто»). Именно такой вариант приведен в примере.

Роль входной последовательности выполняет параметр aList, роль выходной – параметр bList, роль вектора состояний – массив c. Алгоритм несколько усложняется за счет того, что для установки начального значения при поиске минимума приходится отбрасывать уже «пустые» элементы.

{ Сортировка простой выборкой }

procedure SelectionStdSort(aList, bList: TList; aCompare: TCompareFunc);

```
var

i, j, m, N: Integer;

{ Состояние элементов входного множества}

c:arrayofBoolean;

begin

N:=aList.Count;

SetLength(c, N);

{ Сброс отметок}

for i:=0 to N-1 do c[i]:=True;

{ Поиск первого невыбранного элемента во входном множестве}

for i:=0 to N-1 do

begin

j:=0;

while not c[j] do j:=j+1;

m:=j;

{ Поиск минимального элемента }

for j:=1 to N-1 do

if c[j] and (aCompare(aList.List[j], aList.List[m]) = -1) then m:=j;

{ Запись в выходное множество}
```

```
bList.Items[i]:=aList.List[m];

{ В входное множество - "пусто" }

c[m]:=False;

end;

end;
```

Количество выполняемых сравнений для первого прохода равно n, для второго n-1 и т.д. Общее число сравнений будет равно n(n+1)/2-1 и порядок алгоритма – $O(n^2)$. Однако количество перестановок значительно меньше – при каждом выполнении цикла производится всего одна перестановка, а общее их количество n-1 и O(n).

Следовательно, если стоимость перестановки элементов (время или требуемые ресурсы) значительно больше времени сравнения, сортировка выборкой окажется достаточно эффективной. Кроме того, сортировка относится к группе устойчивых алгоритмов. Поиск наименьшего значения будет возвращать первое в списке наименьшее значение из нескольких имеющихся.

Обменная сортировка простой выборкой. Алгоритм сортировки простой выборкой, однако, редко применяется в том варианте, в каком он описан выше. Гораздо чаще применяется, обменный вариант. При обменной сортировке входное и выходное множество располагаются в одной и той же области памяти; выходное – в начале области, входное – в оставшейся ее части. В исходном состоянии входное множество занимает всю область, а выходное множество – пустое. По мере выполнения сортировки входное множество сужается, а выходное – расширяется. Обменная сортировка простой выборкой показана в следующем примере.

```
{ Обменная сортировка простой выборкой }

procedure SelectionSort(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);
```

```
var
i, j, IndexOfMin: Integer;
Temp:Pointer;
begin
{ Перебор элементов выходного множества }
for i:=aFirst to aLast-1 do
{ Входное множество - [i:N-1]; выходное - [1:i-1] }
begin
IndexOfMin:=i;
{ Поиск минимума во входном множестве }
for j:=i+1 to aLast do
{ Обмен первого элемента входного множества с минимальным }
if (aCompare(aList.List[j], aList.List[IndexOfMin]) < 0) then
IndexOfMin:=j;
Temp:=aList.List[i];
aList.List[i]:=aList.List[IndexOfMin];
aList.List[IndexOfMin]:=Temp;
end;
end;
```

Результаты трассировки представлены в табл. 4.8. Двоеточием показана граница между входным и выходным множествами. Очевидно, обменный вариант обеспечивает экономию памяти и не возникает проблемы «пустого» значения. Общее число сравнений уменьшается вдвое – $n \cdot (n-1)/2$, но порядок алгоритма остается прежним – $O(n^2)$. Количество перестановок n-1, но перестановка вдвое более времяемкая операция, чем пересылка в предыдущем алгоритме.

Табл. 4.8. Трассировка сортировки простой выборкой.

Шаг	**Содержимое массива**
Исходный	:242 447 286 708 24 11 192 860 937 561
1	11:447 286 708 24 242 192 860 937 561
2	11 24:286 708 447 242 192 860 937 561
3	11 24 192:708 447 242 286 860 937 561
4	11 24 192 242:447 708 286 860 937 561
5	11 24 192 242 286:708 447 860 937 561
6	11 24 192 242 286 447:708 860 937 561
7	11 24 192 242 286 447 561:860 937 708
8	11 24 192 242 286 447 561 708:937 860
9	11 24 192 242 286 447 561 708 860:937
Результат	11 24 192 242 286 447 561 708 860 937:

Простая модификация обменной сортировки предусматривает поиск в одном цикле просмотра входного множества сразу минимума и максимума, и обмен их с первым и с последним элементами множества соответственно. Хотя итоговое количество сравнений и пересылок в этой модификации не уменьшается, достигается экономия на количестве итераций внешнего цикла.

Приведенные алгоритмы сортировки выборкой практически нечувствительны к исходной упорядоченности. В любом случае поиск минимума требует полного просмотра входного множества. В

обменном варианте исходная упорядоченность может дать некоторую экономию на перестановках для случаев, когда минимальный элемент найден на первом месте во входном множестве.

Пузырьковая сортировка. Входное множество просматривается, при этом попарно сравниваются соседние элементы. Если порядок их следования не соответствует заданному критерию упорядоченности, то элементы меняются местами. В результате одного просмотра при сортировке по возрастанию элемент с самым большим значением ключа переместится («всплывет») на последнее место во множестве.

При следующем проходе на свое место «всплывет» второй по величине ключа элемент и т.д. Для постановки на свои места n элементов следует сделать n-1 проходов. Выходное множество формируется в конце сортируемой последовательности, при каждом следующем проходе его объем увеличивается на 1, а объем входного множества уменьшается на 1.

Достоинство пузырьковой сортировки в том, что при незначительных модификациях ее можно сделать чувствительной к исходной упорядоченности входного множества. Можно ввести некоторую логическую переменную, которая будет сбрасываться в False перед началом каждого прохода, и устанавливаться в True при любой перестановке. Если по окончании прохода значение этой переменной останется False, это означает, что менять местами больше нечего, и сортировка закончена. При такой модификации поступление на вход алгоритма уже упорядоченного множества потребует только одного просмотра.

```
{ Пузырьковая сортировка }

procedure BubbleSort(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

i,j: Integer;

Temp: Pointer;
```

```
Done: Boolean;

begin

for i:=aFirst to aLast-1 do

begin

Done:=True;

for j:=aLast downto i+1 do

{ Переставить j-й и j-1-й элементы }

if aCompare(aList.List[j],aList.List[j-1]) < 0 then

begin

Temp:=aList.List[j];

aList.List[j]:=aList.List[j-1];

aList.List[j-1]:=Temp;

Done:=False;

end;

if Done then Exit;

end;

end;
```

Пузырьковая сортировка принадлежит к алгоритмам класса $O(n^2)$. В реализации присутствуют два цикла: внешний и внутренний, при этом количество выполнений каждого цикла зависит от числа элементов. При первом выполнении внутреннего цикла будет произведено n-1

сравнений, при втором – n-2, при третьем – n-3 и т.д. Всего будет n-1 таких циклов, а общее число сравнений составит: (n-1)+(n-2)+..+1. Сумму можно упростить до n(n-1)/2 или (n^2-n)/2 и получаем O(n^2).

Количество перестановок в худшем случае (при отсортированном исходном наборе в обратном порядке) равно числу сравнений. В лучшем случае (при исходной упорядоченности всех элементов) будет выполнен всего один проход, (n-1) сравнение и ни одной перестановки, что дает линейный порядок O(n).

В сортировке всегда сравниваются и перемещаются только соседние элементы, и это делает ее удобной для обработки связных списков (см. главу 6). Перестановка в связных списках также получается более экономной. Однако при физическом перемещении элементов, если элемент с наименьшим значением оказывается в противоположном конце списка, затрачиваемое время значительно.

Пузырьковая сортировка относится к неустойчивой, поскольку из двух элементов с равными значениями первым в отсортированном наборе будет элемент, который находился в исходном наборе дальше от начала. Если изменить тип сравнения на «меньше чем» или «равен», тогда алгоритм станет устойчивым, но количество перестановок увеличится.

Результаты трассировки примера представлены в табл. 4.9.

Табл. 4.9. Трассировка пузырьковой сортировки.

Шаг	Содержимое массива
Исходный	717 473 313 160 949 764 34 467 757 800:
1	473 313 160 717 764 34 467 757 800:949
2	313 160 473 717 34 467 757:764 800 949
3	160 313 473 34 467:717 757 764 800 949
4	160 313 34 467:473 717 757 764 800 949
5	160 34: 313 467 473 717 757 764 800 949

6 34: 160 313 467 473 717 757 764 800 949

Результат :34 160 313 467 473 717 757 764 800 949

Модификация пузырьковой сортировки носит название шейкер-сортировки. Направления просмотров чередуются: за просмотром от начала к концу входного множества следует просмотр в обратном направлении. При просмотре в прямом направлении элемент с самым большим значением ставится на свое место в последовательности, при просмотре в обратном направлении – элемент с самым маленьким.

```
{ Пузырьковая двухпроходная сортировка }

procedureShakerSort(aList:TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

i: Integer;

Temp: Pointer;

begin

while aFirst < aLast do

begin

for i:=aLast downto aFirst+1 do

if aCompare(aList.List[i], aList.List[i-1]) < 0 then

begin

Temp:=aList.List[i];

aList.List[i]:=aList.List[i-1];
```

```
aList.List[i-1]:=Temp;

end;

Inc(aFirst);

for i:=aFirst+1 to aLast do

if aCompare(aList.List[i], aList.List[i-1]) < 0 then

begin

Temp:=aList.List[i];

aList.List[i]:=aList.List[i-1];

aList.List[i-1]:=Temp;

end;

Dec(aLast);

end;

end;
```

Алгоритм по-прежнему относится к классу $O(n^2)$, но время его выполнения немного меньше. Название сортировка получила ввиду того, что элементы в наборе колеблются относительно своих позиций до тех пор, пока набор не будет отсортирован. Алгоритм неустойчивый.

Описанный алгоритм весьма эффективен для задач восстановления упорядоченности, когда исходная последовательность уже была упорядочена, но подверглась незначительным изменениям. Упорядоченность в последовательности с одиночным изменением будет гарантированно восстановлена всего за два прохода.

Сортировка простыми вставками представляет дословную реализацию стратегии включения. Порядок алгоритма сортировки простыми вставками – $O(n^2)$, если учитывать только операции сравнения. Но сортировка требует еще в среднем $n^2/4$ перемещений, что делает ее менее эффективной, чем сортировка выборкой. Алгоритм сортировки простыми вставками иллюстрируется следующим примером.

```
{ Сортировка простыми вставками }

procedure InsertionStdSort(aList, bList: TList; aCompare:
TCompareFunc);

var i, j, k: Integer;

begin

{ Перебор входного массива }

for i:=0 to aList.Count-1 do

begin

j:=0;

{ Поиск места для a[i] в выходном массиве

при условии (j < i) и (b[j] <= a[i]) }

while (j < i) and (aCompare(bList.Items[j],aList.Items[i]) <= 0) do

j:=j+1;

{ Освобождение места для нового элемента }

for k:=i downto j+1 do

bList.Items[k]:=bList.Items[k-1];
```

```
{ Запись в выходной массив }

bList.Items[j]:=aList.Items[i];

end;

end;
```

Сортировка принадлежит к группе устойчивых алгоритмов. Его эффективность может быть улучшена при применении не линейного, а двоичного поиска. Однако следует иметь в виду, что такое улучшение может быть достигнуто лишь на множествах значительного по количеству элементов объема. Алгоритм требует большого числа пересылок, поэтому при значительном объеме одного элемента эффективность может определяться не количеством операций сравнения, а количеством пересылок.

Пузырьковая сортировка вставками представляет модификацию обменного варианта сортировки. В методе входное и выходное множества разделяют одну область, причем выходное – в начальной ее части. В исходном состоянии можно считать, что первый элемент последовательности уже принадлежит упорядоченному выходному множеству, остальная часть последовательности – неупорядоченное входное.

Первый элемент входного множества примыкает к концу выходного множества. На каждом шаге сортировки происходит перераспределение последовательности: выходное множество увеличивается на один элемент, а входное – уменьшается. Это происходит за счет того, что первый элемент входного множества теперь считается последним элементом выходного.

Затем выполняется просмотр выходного множества от конца к началу с перестановкой соседних элементов, которые не соответствуют критерию упорядоченности. Просмотр прекращается, когда прекращаются перестановки. Это приводит к тому, что последний элемент выходного множества «выплывает» на свое место во множестве.

Поскольку при этом перестановка приводит к сдвигу нового в выходном множестве элемента на одну позицию влево, нет смысла всякий раз производить полный обмен между соседними элементами – достаточно сдвигать старый элемент вправо, а новый элемент записать в выходное множество, когда его место будет установлено. Именно так и построен пример пузырьковой сортировки вставками.

```
{ Пузырьковая сортировка методом вставок }

procedure InsertionBublSort(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

i, j: Integer;

Temp: Pointer;

begin

{ Перебор входного массива }

for i:=aFirst+1 to aLast do

{ Входное множество - [i..N-1], выходное множество - [0..i-1] }

begin

{ Запоминается значение нового элемента }

Temp:=aList.List[i];

j:=i;

{ Поиск места для элемента в выходном множестве со сдвигом

цикл закончится при достижении начала или,
```

когда будет встречен элемент, меньший нового }

while (j > aFirst) **and** (aCompare(Temp, aList.List[j-1]) < 0) **do**

begin

{ Все элементы, большие нового сдвигаются }

aList.List[j]:=aList.List[j-1];

{ Цикл от конца к началу выходного множества }

Dec(j);

end;

{ Новый элемент ставится на свое место }

aList.List[j]:=Temp;

end;

end;

Интересная особенность приведенной реализации алгоритма состоит в следующем: значение текущего элемента сохраняется в локальной переменной, а затем при поиске нужного места его вставки (внутренний цикл) происходит перемещение каждого элемента, значение которого больше текущего, на одну позицию вправо, тем самым, перемещая «дыру» в наборе влево. В конце концов, обнаруживается нужное место, и сохраненное значение помещается в освободившееся место. Результат трассировки представлены в табл. 4.10.

Хотя обменные алгоритмы стратегии включения и позволяют сократить число сравнений при наличии некоторой исходной упорядоченности входного множества, значительное число пересылок существенно снижает эффективность алгоритмов. Поэтому алгоритмы вставками целесообразно применять к связным структурам данных,

когда операция перестановки элементов структуры требует не пересылки данных в памяти, а выполняется способом коррекции указателей.

Заметим, как и при пузырьковой сортировке, при сортировке методом вставок элементы попадают в нужные позиции только за счет смены позиций с соседними элементами. Если элемент находится далеко от требуемой позиции, его перемещение занимает значительное время. Повысить скорость можно путем перемещения элементов не через соседние элементы, а сразу в некоторый диапазон, где текущий элемент должен находиться.

Табл. 4.10. Трассировка сортировки вставками.

Шаг	Содержимое массива
Исходный	48:43 90 39 9 56 40 41 75 72
1	43 48:90 39 9 56 40 41 75 72
2	43 48 90:39 9 56 40 41 75 72
3	39 43 48 90: 9 56 40 41 75 72
4	9 39 43 48 90:56 40 41 75 72
5	9 39 43 48 56 90:40 41 75 72
6	9 39 40 43 48 56 90:41 75 72
7	9 39 40 41 43 48 56 90:75 72
8	9 39 40 41 43 48 56 75 90:72
Результат	9 39 40 41 43 48 56 72 75 90:

4.4.6.2.2. Быстрые алгоритмы сортировки

Алгоритмы второй группы работают быстрее предыдущих методов. Однако в отличие от самых быстрых алгоритмов, их математический анализ выполнить довольно сложно. Несмотря на то, что на практике эти алгоритмы выполняются достаточно быстро, используют их сравнительно редко.

Сортировка Шелла была предложена Дональдом Л.Шеллом в 1959 г. Метод пытается повысить скорость работы за счет быстрого перемещения элементов, находящихся далеко от нужных позиций. Сортировка предполагает перемещение таких элементов большими «прыжками», а окончательная установка в нужные позиции может быть выполнена одним из классических способов.

Качественный порядок сортировки остается $O(n^2)$, но среднее число сравнений, определенное эмпирическим путем – $n \cdot \log_2 n^2$. Ускорение достигается за счет того, что выявленные «не на месте» элементы при шаге сравнения большем единицы быстрее «всплывают».

Следующий пример представляет реализацию сортировку Шелла, основанную на пузырьковом алгоритме. Исходный шаг сравнения h соизмерим с половиной общего размера последовательности. Сначала выполняется пузырьковая сортировка с интервалом h. Затем величина h уменьшается вдвое и вновь выполняется пузырьковая сортировка, далее h уменьшается еще вдвое и т.д. Последняя пузырьковая сортировка выполняется при h=1.

```
{ Сортировка Шелла }

procedure ShellSort(aList: TList; aCompare: TCompareFunc);

var

h, i, t, N: Integer;

Temp: Pointer;

{ Признак перестановки }

k: Boolean;

begin

N:=aList.Count;

{ Начальное значение интервала }
```

```
h:=N div 2;
{ Цикл с уменьшением интервала до 1 }
while h> 0 do
begin
{ Пузырьковая сортировка с интервалом h}
k:=True;
{ Цикл, пока есть перестановки }
while k do
begin
k:=False;
{ Сравнение элементов на интервале h}
for i:=0 to N-h-1 do
begin
if aCompare(aList.List[i], aList.List[i+h]) = 1 then
begin
{ Перестановка }
Temp:=aList.List[i];
aList.List[i]:=aList.List[i+h];
aList.List[i+h]:=Temp;
```

```
{ Признак перестановки }
k:=True;
end;
end;
end;
{ Уменьшение интервала }
h:=h div 2;
end;
end;
```

Результаты трассировки примера представлены в табл. 4.11.

Табл. 4.11. Трассировка сортировки Шелла.

Шаг	h	Содержимое массива
Исходный		76 22 4 17 13 49 4 18 32 40 96 57 77 20 1 52
1	8	32 22 4 17 13 20 1 18 76 40 96 57 77 49 4 52
2	8	32 22 4 17 13 20 1 18 76 40 96 57 77 49 4 52
3	4	13 20 1 17 32 22 4 18 76 40 4 52 77 49 96 57
4	4	13 20 1 17 32 22 4 18 76 40 4 52 77 49 96 57
5	2	13 20 1 17 32 22 4 18 76 40 4 52 77 49 96 57
6	2	13 20 1 17 32 22 4 18 76 40 4 52 77 49 96 57
7	2	1 17 4 18 4 20 13 22 32 40 76 49 77 52 96 57
8	2	1 17 4 18 4 20 13 22 32 40 76 49 77 52 96 57

9	1	1 4 17 4 18 13 20 22 32 40 49 76 52 77 57 96
10	1	1 4 4 17 13 18 20 22 32 40 49 52 76 57 77 96
11	1	1 4 4 13 17 18 20 22 32 40 49 52 57 76 77 96
12	1	1 4 4 13 17 18 20 22 32 40 49 52 57 76 77 96
Результат		1 4 4 13 17 18 20 22 32 40 49 52 57 76 77 96

В другой реализации сортировки установка элементов в нужные позиции выполняется методом вставок. Строго говоря, здесь метод Шелла работает путем вставки отсортированных подмножеств основного набора. Каждое подмножество формируется за счет выборки каждого h-ого элемента, начиная с любой позиций в наборе. В результате будет получено h подмножеств, которые отсортированы методом вставок.

Полученная последовательность называется отсортированной по h. Затем значение h уменьшается и вновь выполняется сортировка. Уменьшение h происходит до тех пор, пока h не будет равно 1, после чего последний проход будет представлять собой стандартную сортировку методом вставок (точнее сортировку по 1).

Суть сортировки Шелла состоит в том, что сортировка по h быстро переносит элементы в область, где они должны находиться в отсортированном наборе, а уменьшение значения h позволяет постепенно уменьшать размер «прыжков» и, в конце концов, поместить элемент в требуемую позицию. Медленному перемещению предшествуют большие «скачки», сводящиеся к простой сортировке методом вставок, которая практически не передвигает элементы.

Выбор шага изменения играет важную роль. Шелл предложил в своей первой работе значения 1, 2, 4, 8, 16, 32 и т.д. Однако при таком наборе до последнего прохода элементы с четными индексами никогда не сравниваются с элементами с нечетными индексами. Следовательно, при выполнении последнего прохода все еще возможны перемещения элементов на большие расстояния (например, такая ситуация возможна, когда элементы с меньшими значениями находятся в позициях с четными индексами, а элементы с большими значениями – в позициях с нечетными индексами).

В 1969 г. Дональд Кнут предложил последовательность 1, 4, 13, 40, 121 и т.д. (каждое следующее значение на единицу больше утроенного предыдущего). Для набора средних размеров такая последовательность позволяет получить достаточно высокие показатели быстродействия при несложном методе вычисления значений последовательности. На основе эмпирических исследований Кнут для среднего случая получил порядок $O(n^{5/4})$, а для худшего случая $O(n^{3/2})$. Именно такая последовательность используется в приведенной ниже реализации алгоритма.

```
{ Сортировка Шелла с применением ряда Кнута }

procedure ShellKnuthSort(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

i, j, h, N: Integer;

Temp: Pointer;

begin

{ Начальное значение h должно быть

близко к 1/9 количества элементов }

h:=1; N:=(aLast - aFirst) div 9;

while h <= N do

h:=h*3 + 1;

{ При каждом проходе цикла значение

шага уменьшается на треть }

while h > 0 do
```

```
begin

{ Выполнить сортировку методом

вставки для каждого подмножества }

for i:=(aFirst + h) to aLast do

begin

Temp:=aList.List[i];

j:=i;

while (j >= (aFirst+h)) and (aCompare(Temp, aList.List[j-h]) < 0) do

begin

aList.List[j]:=aList.List[j-h];

Dec(j, h);

end;

aList.List[j]:=Temp;

end;

h:=h div 3;

end;

end;
```

Ряд других последовательностей позволяет получить более высокие показатели, но требуют предварительного вычисления значений последовательности, поскольку формулы вычисления более сложные. В качестве примера приведем самую быструю и известную на сегодня

последовательность, разработанную Робертом Седжвиком: 1, 5, 19, 41, 109 и т.д. Последовательность формируется путем слияния двух последовательностей – 9*4*i-9*2*i+1 для i > 0 и 4*i-3*2*i+1 для i > 1. Для этой последовательности в худшем случае порядок равен $O(n^{4/3})$ и в среднем случае $O(n^{7/6})$.

Математический анализ параметров быстродействия сортировки Шелла достаточно сложен и для оценки времени выполнения при различных значениях h ограничиваются в основном статистическими данными. При перестановке далеко отстоящих элементов возможно нарушение порядка следования элементов с равными значениями и поэтому сортировка Шелла относится к группе неустойчивых алгоритмов.

4.4.6.2.3.Самые быстрые алгоритмы сортировки

Самые быстрые алгоритмы сортировки широко используются на практике и важно понимать их особенности, чтобы оптимальным образом реализовывать и применять их в различных приложениях.

Поразрядная цифровая сортировка. Алгоритм поразрядной цифровой сортировки относится к распределительным, и требует представления ключей сортируемой последовательности в виде чисел в некоторой системе счисления P. Число проходов равно максимальному числу значащих цифр в числе – D. В каждом проходе анализируется значащая цифра в очередном разряде ключа, начиная с младшего разряда. Все ключи с одинаковым значением этой цифры объединяются в одну группу.

Ключи в группе располагаются в порядке их поступления. После того, как вся исходная последовательность распределена по группам, группы располагаются в порядке возрастания связанных с группами цифр. Процесс повторяется для второй цифры и т.д., пока не будут исчерпаны значащие цифры в ключе. Основание системы счисления P может быть любым, в частном случае 2 или 10. Для системы счисления с основанием P требуется P групп.

Порядок алгоритма качественно линейный – O(n), для сортировки требуется D·n операций анализа цифры. Однако в оценке порядка не учитывается ряд обстоятельств. Во-первых, операция выделения значащей цифры будет простой и быстрой только при P=2, для других

систем счисления эта операция может оказаться значительно более времяемкой, чем операция сравнения. Во-вторых, в оценке алгоритма не учитываются расходы времени и памяти на создание и ведение групп.

Размещение групп в статической рабочей памяти потребует памяти для P·n элементов, так как в предельном случае все элементы могут попасть в какую-то одну группу. Если же формировать группы внутри той же последовательности по принципу обменных алгоритмов, то возникает необходимость перераспределения последовательности между группами и все недостатки, присущие алгоритмам включения. Наиболее рациональным является формирование групп в виде связных списков с динамическим выделением памяти.

В приведенном примере реализации алгоритма, однако, применяется поразрядная сортировка к статической структуре данных, и формируются группы на том же месте, где расположена исходная последовательность. В качестве входного параметра процедуры сортировки служит целочисленный массив, индексируемый с единицы.

```
const

D=...; { Максимальное количество цифр в числе }

P=...; { Основание системы счисления }

{ Возвращает значение n-ой цифры в числе v }

function Digit(v, n: Integer): Integer;

begin

for n:=n downto 2 do

v:=v div P;

Digit:=v mod P;

end;
```

```
procedure Sort(var a: TA);
var
{ Индекс элемента, следующего за последним в i-ой группе }
b: array[0..P-2] of Integer;
i, j, k, m, x: Integer;
begin
{ Перебор цифр, начиная с младшей }
for m:=1 to D do
begin
{ Начальные значения индексов }
fori:=0toP-2dob[i]:=1;
{ Перебор массива }
for i:=1 to N do
begin
{ Определение m-ой цифры }
k:=Digit(a[i],m);
x:=a[i];
{ Сдвиг - освобождение места в конце k-ой группы }
for j:=i downto b[k]+1 do
```

```
a[j]:=a[j-1];

{ Запись в конец k-ой группы }

a[b[k]]:=x;

{ Модификация k-го индекса и всех больших }

for j:=k to P-2 do

b[j]:=b[j]+1;

end;

end;

end;
```

Область памяти, занимаемая массивом, перераспределяется между входным и выходным множествами. Выходное множество размещается в начале массива и разбивается на группы. Разбиение отслеживается в массиве b. Элемент массива b[i] содержит индекс в массиве a, с которого начинается i+1-я группа. Номер группы определяется значением анализируемой цифры числа, поэтому индексация в массиве b начинается с 0.

Когда очередное число выбирается из входного множества и должно быть занесено в i-ю группу выходного множества, оно будет записано в позицию, определяемую значением b[i]. Но предварительно эта позиция должна быть освобождена: участок массива от b[i] до конца выходного множества включительно сдвигается вправо. После записи числа в i-ю группу i-е и все последующие значения в массиве b корректируются – увеличиваются на 1.

Результаты трассировки при P=10 и D=4 представлены в табл. 4.12.

Табл. 4.12. Трассировка цифровой сортировки.

Цифра	**Содержимое массивов a и b**
исх.	**220 8390 9524 9510 462 2124 7970 4572 4418 1283**
1	**220 8390 9510 7970 462 4572 1283 9524 2124 4418**
	==========0========= ====2==== ==3== =====4===== ==8==
	b=(5,5,7,8,10,10,10,11,11)
2	**9510 4418 220 9524 2124 462 7970 4572 1283 8390**
	=====1===== ========2====== =6= ====7==== ==8== ==9==
	b=(1,3,6,6,6,6,7,9,10,11)
3	**2124 220 1283 8390 4418 462 9510 9524 4572 7990**
	==1== ====2==== ==3== ====4==== ========5======== ==9==
	b=(1,2,4,5,7,10,10,10,10,11)
4	**220 462 1283 2124 4418 4572 7970 8390 9510 9524**
	===0=== ==1== ==2== =====4==== ==7== ==8== =====9=====
	b=(3,4,5,5,7,7,7,8,9,11)

Быстрая сортировка Хоара относится к распределительным и обеспечивает показатели эффективности $O(n \cdot \log_2(n))$ даже при наихудшем исходном распределении. Алгоритм был разработан Хоаром в 1960 г. Алгоритм требует незначительной дополнительной памяти и относительно в реализации. Однако при реализации допускается много ошибок, которые могут остаться незамеченными и требовать дополнительного времени при выполнении, быстродействие в худшем случае может достигать показателя $O(n^2)$ и к тому же алгоритм относится к группе неустойчивых.

Тем не менее, сортировка Хоара в различных модификациях очень широко применяется. Во всех версиях Delphi методы TList.Sort и TStringList.Sort реализованы на основе быстрой сортировки. В C++ функция qsort из стандартной библиотеки времени выполнения также реализована на основе быстрой сортировки. Большое количество литературных источников по алгоритму быстрой сортировки позволяет корректно реализовать алгоритм и изучить многие особенности его работы.

Основная идея алгоритма быстрой сортировки заключается в следующем. Исходный список разделяется на два подсписка. В каждом из подсписков выбирается некоторый элемент, называемый базовым, относительно которого производится перестановка всех элементов. Элементы, значения которых меньше значения базового элемента, переносятся левее базового, а элементы, значения которых больше – правее относительно базового.

После этого можно утверждать, что базовый элемент располагается на своем месте в списке. Затем выполняется рекурсивный вызов функции сортировки для левого и правого подсписка. Рекурсивные вызовы завершаются, когда переданный функции сортировки список будет содержать всего один элемент, и, следовательно, весь список окажется отсортированным.

Таким образом, для выполнения быстрой сортировки необходимо знать два алгоритма более низкого порядка: метод выбора базового элемента и метод эффективной перестановки элементов списка.

Лучшим случаем выбора базового элемента является медиана – средний элемент отсортированного списка. Тогда количество элементов в обоих подсписков будет одинаковым. Однако процесс поиска медианы сводится к алгоритму быстрой сортировки, поэтому неприемлем.

Худшим случаем является выбор в качестве базового элемента с максимальным или минимальным значением. В этом случае один из подсписков будет пустым. Заранее невозможно определить, выбран ли такой граничный случай, однако, если при каждом рекурсивном вызове будет выбираться такой элемент в качестве базового, то для n элементов потребуется n уровней рекурсии, что может вызвать определенные проблему при достаточно большом значении n.

Таким образом, желательно выбирать в качестве базового элемент, как можно ближе расположенный к среднему и как можно дальше – от граничных элементов. В некоторых литературных источниках в качестве базового выбирается первый или последний элемент исходного списка. Такая стратегия ничем не лучше любой другой для неотсортированного исходного списка, однако для отсортированного списка она соответствует худшему случаю.

В примере приведена реализация процедуры быстрой сортировки. Базовым элементом выбирается каждый раз средний элемент неотсортированного списка. Алгоритм рекурсивный, поэтому при его вызове должны быть заданы значения границ сортируемого участка.

```
{ Быстрая сортировка Хоара с выбором

среднего элемента в качестве базового }

procedure QuickHoarStd1Sort(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

L, R: Integer;

M, Temp: Pointer;

begin

if aFirst >= aLast then Exit;

{ В качестве базового элемента выбирается средний }

M:=aList.List^[(aFirst+aLast) div 2];

{ Начальные значения индексов }

L:=aFirst-1; R:=aLast+1;
```

```
{ Приступить к разбиению списка }

whileTruedo

begin

repeat Dec(R);

until aCompare(aList.List[R], M) <= 0;

repeat Inc(L);

until aCompare(aList.List[L], M) >= 0;

if L >= R then Break;

Temp:=aList.List[L];

aList.List[L]:=aList.List[R];

aList.List[R]:=Temp;

end;

{ Выполнить быструю сортировку левого подсписка }

QuickHoarStd1Sort(aList, aFirst, R, aCompare);

{ Выполнить быструю сортировку правого подсписка }

QuickHoarStd1Sort(aList, R+1, aLast, aCompare);

end;
```

В алгоритме для разделения списка используются два индекса L и R. Первый используется для прохождения по элементам списка слева направо, второй – справа налево. Предварительно настраиваются начальные значения индексов – перед первым элементом и после

последнего элемента списка. Затем организуется бесконечный цикл. В первом внутреннем цикле уменьшается значение индекса R до тех пор, пока он не будет указывать на элемент, значение которого меньше или равно значению базового элемента. Во втором внутреннем цикле увеличивается значение индекса L до тех пор, пока он не будет указывать на элемент, значение которого больше или равно базовому элементу.

После завершения обоих внутренних циклов возможны два исхода. В первом случае индекс L меньше индекса R, т.е. два элемента, на которые указывают индексы, расположены в неверном порядке. Тогда следует поменять расположение элементов и продолжить выполнение внутренних циклов. Во втором случае индексы равны или пересеклись – значение левого индекса равно или превосходит значение правого. Следовательно, список успешно разделен, и выполнение циклов можно прервать. После выхода из бесконечного цикла рекурсивно применяется тот же алгоритм для левого и правого подсписка.

Результаты трассировки приведены в табл. 4.13. В каждой строке таблицы показаны текущие положения индексов L и R, звездочками отмечены элементы, ставшие на свои места. Для каждого прохода показаны границы подмножества, в котором ведется сортировка.

Таблица 4.13. Трассировка сортировки Хоара.

Проход	Содержимое массива
1	**==========**
	42L 79 39 65 60 29 86 95 25 37R
	37 79L 39 65 60 29 86 95 25 42R
	37 42L 39 65 60 29 86 95 25R 79
	37 25 39L 65 60 29 86 95 42R 79
	37 25 39 65L 60 29 86 95 42R 79

37 25 39 42L 60 29 86 95R 65 79

37 25 39 42L 60 29 86R 95 65 79

37 25 39 42L 60 29R 86 95 65 79

37 25 39 29 60L 42R 86 95 65 79

2 **==========**

29L 25 39 37R 42* 60 86 95 65 79

29 25L 39 37R 42 60 86 95 65 79

29 25 37L 39R 42 60 86 95 65 79

3 **=======**

25L 29R 37* 39 42 60 86 95 65 79

4 **==**

25* 29 37* 39 42 60 86 95 65 79

5 **==**

25* 29* 37* 39 42 60 86 95 65 79

6 **==========**

25* 29* 37* 39* 42* 60L 86 95 65 79R

25* 29* 37* 39* 42* 60L 86 95 65L 79

25* 29* 37* 39* 42* 60L 86 95L 65 79

25* 29* 37* 39* 42* 60L 86R 95 65 79

7 ==========

25* 29* 37* 39* 42* 60* 79L 95 65 86R

25* 29* 37* 39* 42* 60* 79L 86 65R 95

25* 29* 37* 39* 42* 60* 65 86L 79R 95

8 ==

25* 29* 37* 39* 42* 60* 65 79* 86 95

9 =======

25* 29* 37* 39* 42* 60* 65* 79* 86L 95R

10 ==

25* 29* 37* 39* 42* 60* 65* 79* 86* 95

Незначительная модификация алгоритма позволит избавиться от одного рекурсивного вызова. В следующем примере используется цикл while совместно с изменением значения переменной aFirst. В результате левые подсписки будут обрабатываться рекурсивно, а правые – итеративно.

{ Быстрая сортировка Хоара (без одной рекурсии) }

procedure QuickHoarStd2Sort(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

L, R: Integer;

M, Temp: Pointer;

```
begin
{ Повторять, по в списке
есть хотя бы два элемента }
while(aFirst<aLast)do
begin
{ В качестве базового элемента выбирается средний }
M:=aList.List^[(aFirst+aLast) div 2];
{ Начальные значения индексов }
L:=aFirst-1; R:=aLast+1;
{ Приступить к разбиению списка }
whileTruedo
begin
repeat Dec(R);
until aCompare(aList.List[R], M) <= 0;
repeat Inc(L);
until aCompare(aList.List[L], M) >= 0;
if L >= R then Break;
Temp:=aList.List[L];
aList.List[L]:=aList.List[R];
```

```
aList.List[R]:=Temp;

end;

{ Выполнить быструю сортировку левого подсписка }

if aFirst < R then

QuickHoarStd2Sort(aList, aFirst, R, aCompare);

{ Выполнить быструю сортировку правого подсписка и устранение
рекурсии }

aFirst:=R+1;

end;

end;
```

Другая незначительная модификация состоит в выборе базового элемента случайным образом: случайно выбранный элемент меняется местом со средним и становится базовым. При наличии хорошего генератора псевдослучайных чисел вероятность попадания на худший случай (граничное значение) становится пренебрежимо мало. Заметим, однако, что понижение вероятности не повышает скорость выполнения алгоритма в целом. Пример реализации алгоритма со случайным выбором базового элемента приведен ниже.

```
{ Быстрая сортировка Хоара со случайным выбором базового элемента
}

procedure QuickHoarRNDSort(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

L, R: Integer;
```

```
M, Temp: Pointer;
begin
while aFirst < aLast do
begin
{ Начало добавляемой части }
{ Выбрать случайный элемент, переставить его со
средним элементом и взять в качестве базового }
R:=aFirst + Random(aLast - aFirst + 1);
L:=(aFirst + aLast) div 2;
M:=aList.List[R];
aList.List[R]:=aList.List[L];
aList.List[L]:=M;
{ Конец добавляемой части }
L:=aFirst-1;
R:=aLast+1;
while True do
begin
repeat Dec(R);
until aCompare(aList.List[R], M) <= 0;
```

```
repeat Inc(L);

until aCompare(aList.List[L], M) >= 0;

if L >= R then Break;

Temp:=aList.List[L];

aList.List[L]:=aList.List[R];

aList.List[R]:=Temp;

end;

if (aFirst < R) then

QuickHoarRNDSort(aList, aFirst, R, aCompare);

aFirst:=R+1;

end;

end;
```

Самым эффективным методом выбора базового элемента на сегодняшний день является метод трех медиан, заключающийся в приближенном определении медианы. Согласно методу из подсписка выбираются первый, последний и средний по порядку элементы. Элемент с наименьшим значением помещается в первую позицию, средний элемент – в середину списка, с наибольшим значением – в последнюю позицию. В результате, находится не только медиана трех элементов, но и сокращается размер частей списка на два элемента, поскольку уже известно, что они находятся в правильных частях списка относительно базового элемента.

4.4.6.2.4. Сортировка слиянием

Алгоритмы сортировки слиянием привлекают простотой и наличием важных особенностей: они имеют порядок $O(n \cdot \log_2 n)$, не имеют

худших случаев, допускают сортировку содержимого файлов, размер которых слишком велик для полного размещения в памяти.

Для пояснения алгоритма сортировки поясним сначала принцип слияние. Пусть имеются два отсортированных в порядке возрастания массива p[1], p[2], ..., p[n] и q[1], q[2], ..., q[n] и имеется пустой массив r[1], r[2], ..., r[2n], который требуется заполнить значениями массивов p и q в порядке возрастания.

Для слияния выполняются следующие действия: сравниваются p[1] и q[1], и меньшее из значений записывается в r[1]. Предположим, это значение p[1]. Тогда p[2] сравнивается с q[1] и меньшее из значений заносится в r[2]. Предположим, это значение q[1]. Тогда на следующем шаге сравниваются значения p[2] и q[2] и т.д., пока не будет достигнута граница одного из массивов. Тогда остаток другого массива просто дописывается в конец результирующего массива r.

Такой алгоритм известен как алгоритм *двухпутевого слияния*. На практике элементы не удаляются из исходных массивов, а используются указатели на текущие начальные элементы, которые при копировании передвигаются на следующий элемент.

Время выполнения алгоритма двухпутевого слияния зависит от количества элементов в обоих массивах. Если в первом находится n элементов, а во втором – m, то в худшем случае потребуется (n+m) сравнений. Следовательно, алгоритм двухпутевого слияния принадлежит к классу O(n). Пример слияния двух массивов показан на рис. 4.2.

На основе алгоритма двухпутевого слияния можно прийти к рекурсивному определению сортировки слиянием: исходный массив следует разделить на две половины, применить к каждой половине алгоритм сортировки слиянием, а затем с помощью алгоритма двухпутевого слияния объединить подсписки в один отсортированный массив. Рекурсивные вызовы завершаются, когда подсписок n-ого уровня, переданный алгоритму сортировки, будет содержать всего один элемент, поскольку он, очевидно, уже отсортирован.

Сортировка слиянием обладает единственным недостатком – требуется наличие третьего списка, в котором будут храниться результаты слияния. Размер требуемой дополнительной памяти составляет до

половины размера исходно массива.Пример реализации алгоритма приведен ниже. Помимо известных параметров процедура требует передачи указателя на временный массив, который будет использоваться для выполнения процедуры слияния.

```
{ Сортировка слиянием }

procedure MSS(aList: TList; aFirst, aLast: Integer;

aCompare: TCompareFunc; aTempList: PPointerList);

var

Mid, i, j, ToInx,

FirstCount: Integer;

begin

{ Вычислить среднюю точку }

Mid:=(aFirst + aLast) div 2;

{ Рекурсивная сортировка слиянием первой и второй половин списка }

if aFirst < Mid then

MSS(aList, aFirst, Mid, aCompare, aTempList);

if (Mid+1) < aLast then

MSS(aList, Mid+1, aLast, aCompare, aTempList);

{ Скопировать первую половину списка во вспомогательный список }

FirstCount:=Mid-aFirst+1;

Move(aList.List[aFirst], aTempList[0], FirstCount*SizeOf(Pointer));
```

```
{ Установить значения индексов: i - индекс для вспомогательного
списка

(т.е. первой половины); j - индекс для второй половины списка;

ToInx - индекс в результирующем списке, куда будут копироваться

отсортированные элементы }

i:=0; j:=Mid+1; ToInx:=aFirst;

{ Выполнить слияние двух списков; повторять

пока один из списков не опустеет }

while (i < FirstCount) and (j <= aLast) do

begin

{ Определить элемент с наименьшим значением из следующих

элементов в обоих списках и скопировать его; увеличить

значение соответствующего индекса }

if aCompare(aTempList[i], aList.List[j]) <= 0 then

begin

aList.List[ToInx]:=aTempList[i];

Inc(i);

end

else

begin
```

```
aList.List[ToInx]:=aList.List[j];

Inc(j);

end;

{ В объединенных списках есть еще один элемент }

Inc(ToInx);

end;

{ Если в первом подсписке остались элементы, скопировать их }

if i < FirstCount then

Move(aTempList[i], aList.List[ToInx], (FirstCount - i)*SizeOf(Pointer));

{ Если во втором списке остались элементы, то они уже находятся в
нужных

позициях, т.е. сортировка завершена; если второй список пуст,
сортировка

также завершена }

end;

procedure MergeSortStd(aList: TList;

aFirst, aLast: Integer; aCompare: TCompareFunc);

var

TempList: PPointerList;

ItemCount: Integer;
```

```
begin

{ Есть хотя бы два элемента для сортировки }

if aFirst < aLast then

begin

{ создать временный список указателей }

ItemCount:=aLast-aFirst+1;

GetMem(TempList, ((ItemCount+1) div 2)*SizeOf(Pointer));

try

MSS(aList, aFirst, aLast, aCompare, TempList);

finally

FreeMem(TempList, ((ItemCount+1) div 2)*SizeOf(Pointer));

end;

end;

end;
```

Процедура сортировки рекурсивно вызывает саму себя для сортировки первой и второй половин переданной ей части массива. Затем она копирует первую половину во вспомогательный массив и выполняется слияние двух списков. Поскольку элементы перемещаются только при выполнении процедуры слияния, устойчивость всей сортировки будет зависеть от устойчивости самого слияния двух половин.

Обратите внимание, если в обеих половинах имеются элементы с одинаковым значением, оператор сравнения гарантирует, что первым в результирующий список попадет элемент из первой половины списка.

Следовательно, операция слияния сохраняет относительный порядок следования элементов, и сортировка слиянием является устойчивой.

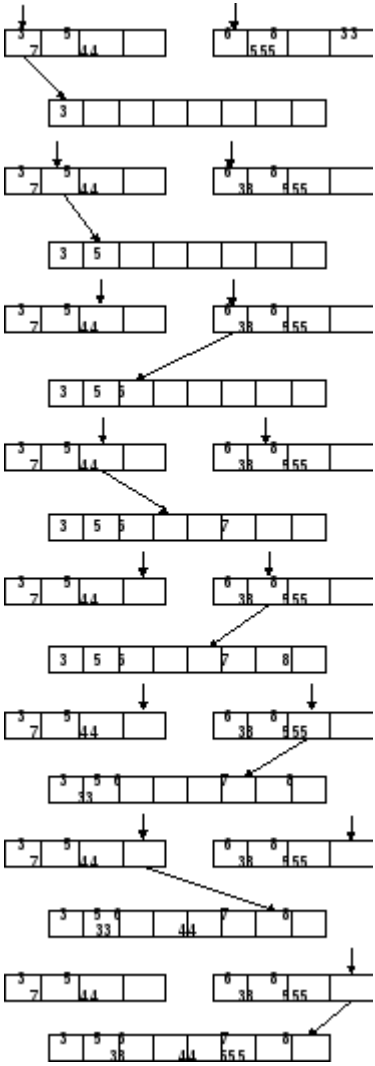


Рис. 4.2. Пример слияния двух массивов.

4.5. Полустатические структуры данных

Полустатические структуры данных характеризуются следующими признаками:

- переменная длина и простые процедуры ее изменения;
- изменение длины структуры происходит в определенных пределах, не превышая максимального (предельного) значения.

На логическом уровне полустатическая структура представляет последовательность данных, связанная отношениями линейного списка. Доступ к элементу может осуществляться по его порядковому номеру. На физическом уровне полустатическая структура данных представляет последовательность слотов, где каждый следующий элемент расположен в памяти в следующем слоте. Физическое представление также может иметь вид однонаправленного связного списка (цепочки), где каждый следующий элемент адресуется указателем, находящимся в текущем элементе. В последнем случае ограничения на длину структуры гораздо менее строги.

4.5.1. Стеки

***Стек*– последовательный список с переменной длиной, включение и исключение элементов из которого выполняются с одной стороны списка, называемого вершиной. Другие названия стека – магазин или очередь, функционирующая по принципу LIFO (Last-In-First-Out – «последним пришел - первым исключается»). Основные операции над стеком – включение нового элемента (англ. push заталкивать) и исключение элемента из стека (англ. pop – выскакивать).** Полезными могут быть также такие как определение текущего числа элементов в стеке и очистка стека.

Рассмотрим пример, демонстрирующий принцип включения элементов в стек и исключения элементов из стека. На рис. 5.1 изображены состояния стека: пустого (а), после последовательного включения в него элементов с именами 'A', 'B', 'C' (б-г), после последовательного удаления из стека элементов 'C' и 'B' (д, е), после включения в стек элемента 'D' (ж).

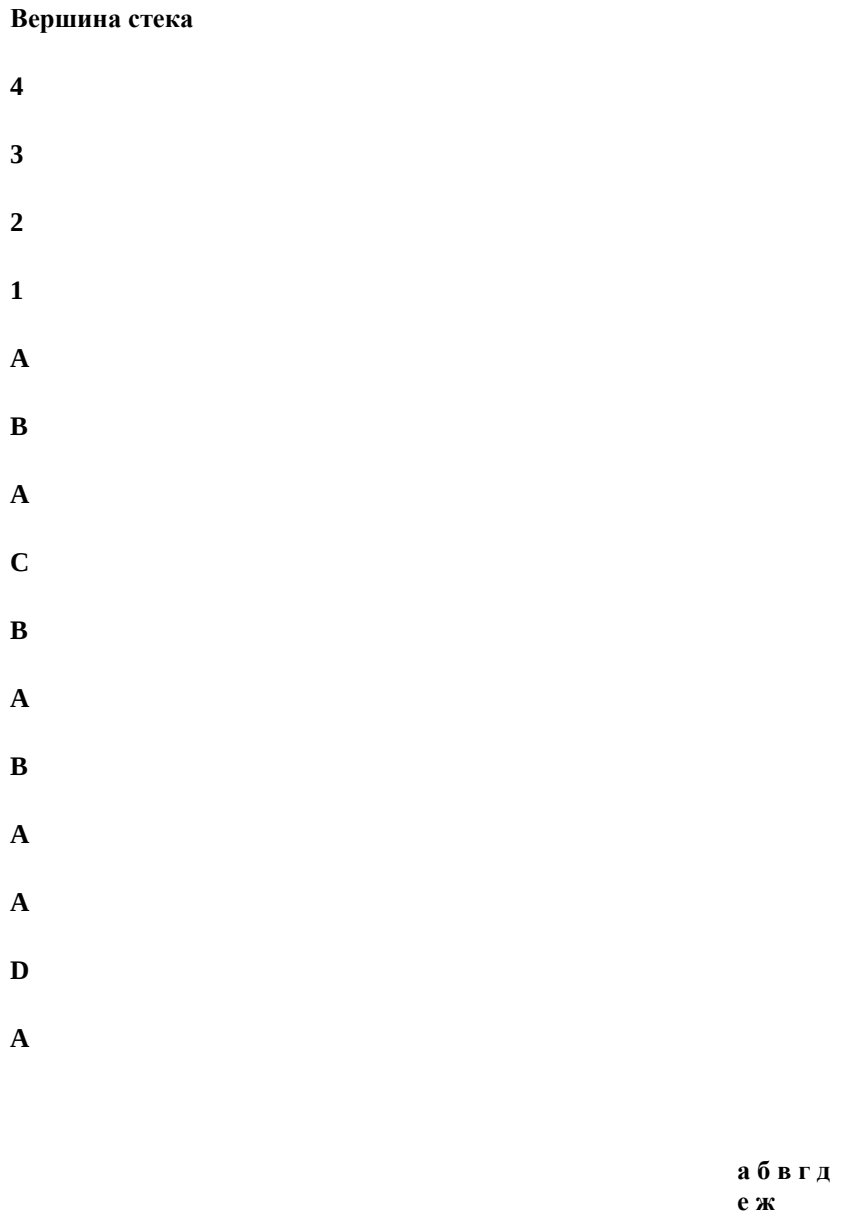


Рис 5.1. Включение и исключение элементов из стека.

При представлении стека в статической памяти для него выделяется память, как для вектора. В дескрипторе вектора кроме обычных параметров должен находиться указатель стека – адрес вершины стека. Указатель стека может указывать либо на первый свободный элемент стека, либо на последний записанный в стек элемент. Все равно, какой из этих двух вариантов выбрать, важно впоследствии строго придерживаться его при работе со стеком.

При занесении элемента в стек элемент записывается на место, определяемое указателем стека, затем указатель модифицируется таким образом, чтобы он указывал на следующий свободный элемент (если указатель указывает на последний записанный элемент, то сначала модифицируется указатель, а затем производится запись элемента). Модификация указателя состоит в прибавлении или вычитании из него значения, равного размеру элемента.

Операция исключения элемента состоит в модификации указателя стека (в направлении, обратном модификации при включении) и выборке значения, на которое указывает указатель стека. После выборки слот, в котором размещался выбранный элемент, считается свободным.

Операция очистки стека сводится к записи в указатель стека начального значения – адреса начала выделенной области памяти, а операция определения размера стека – к вычислению разности указателя стека и адреса начала области, отведенной под стек.

4.5.1.1. Стеки в вычислительных системах

Стек является удобной структурой данных для многих задач вычислительной техники. Наиболее типичной задачей является обеспечение вложенных вызовов процедур. Предположим, имеется процедура A, которая вызывает процедуру B, а та в свою очередь – процедуру C. Когда выполнение процедуры A дойдет до вызова B, процедура A приостанавливается и управление передается на входную точку процедуры B. Когда B доходит до вызова C, приостанавливается B и управление передается на процедуру C. Когда заканчивается выполнение процедуры C, управление должно быть возвращено в B, причем в точку, следующую за вызовом C. При завершении B

управление должно возвращаться в A, в точку, следующую за вызовом B.

Правильную последовательность возвратов легко обеспечить, если при каждом вызове процедуры записывать адрес возврата в стек. В результате, когда процедура A вызывает процедуру B, в стек заносится адрес возврата в A; когда B вызывает C, в стек заносится адрес возврата в B. Когда C заканчивается, адрес возврата выбирается из вершины стека – а это адрес возврата в B. Когда заканчивается B, в вершине стека находится адрес возврата в A, и возврат из B произойдет в A.

Языки программирования блочного типа (PASCAL, C и др.) используют стек для размещения локальных переменных процедур и иных программных блоков. При каждой активизации процедуры память для ее локальных переменных выделяется в стеке, а при завершении процедуры память освобождается. Поскольку при вызовах процедур всегда строго соблюдается вложенность, то в вершине стека всегда находится память, содержащая локальные переменные активной в данный момент процедуры.

Такой прием делает возможной легкую реализацию рекурсивных процедур. Когда процедура вызывает саму себя, для всех ее локальных переменных выделяется память в стеке, и вложенный вызов работает с собственным представлением локальных переменных. Когда вложенный вызов завершается, занимаемая переменными область памяти в стеке освобождается и актуальным становится представление локальных переменных предыдущего уровня. За счет этого в языках PASCAL и C любые процедуры/функции могут вызывать сами себя. Рекурсия использует стек в скрытом виде, но все рекурсивные процедуры могут быть реализованы и без рекурсии с явным использованием стека.

4.5.2. Очереди fifo

Очередью FIFO(First-In-First-Out – «первым пришел – первым исключается») является последовательный список с переменной длиной, в котором включение элементов выполняется только с одной стороны списка (конец или хвост очереди), а исключение – с другой стороны (начало или голова очереди). Например, очереди к прилавкам и кассам являются типичным бытовым примером очереди FIFO.

Основные операции над очередью – те же, что и над стеком – включение, исключение, определение размера, очистка, неразрушающее чтение.

При представлении очереди вектором в статической памяти в дескрипторе должны находиться два указателя: на начало очереди (первый элемент в очереди) и на ее конец (первый свободный элемент в очереди). При включении элемента в очередь элемент записывается по адресу, определяемому указателем на конец, после чего указатель увеличивается на значение, равное размеру элементу. При исключении элемента из очереди выбирается элемент, адресуемый указателем на начало, после чего указатель уменьшается.

Очевидно, со временем указатель на конец при очередном включении элемента достигнет верхней границы той области памяти, которая выделена для очереди. Если операции включения чередовались с операциями исключения элементов, то в начальной части отведенной под очередь памяти имеется свободное место. Чтобы места, занимаемые исключенными элементами, могли быть повторно использованы, очередь замыкается в кольцо: указатели (на начало и на конец), достигнув конца выделенной области памяти, переключаются на ее начало. Такая организация очереди в памяти называется *кольцевой очередью*.

В исходном состоянии указатели на начало и на конец указывают на начало области памяти. Равенство двух указателей (при любом их значении) является признаком пустой очереди. Если в процессе работы с кольцевой очередью число операций включения превышает число операций исключения, то может возникнуть ситуация, в которой указатель конца «догонит» указатель начала. Это ситуация заполненной очереди, но если в этой ситуации указатели сравняются, эта ситуация будет неотличима от ситуации пустой очереди. Для различения этих двух ситуаций к кольцевой очереди предъявляется требование, чтобы между указателем конца и указателем начала оставался «зазор» из свободных элементов. Когда «зазор» сокращается до одного элемента, очередь считается заполненной, и дальнейшие попытки записи в нее блокируются.

Очистка очереди сводится к записи одного и того же (не обязательно начального) значения в оба указателя. Определение размера состоит в вычислении разности указателей с учетом кольцевой природы очереди.

Возможны и другие варианты организации очереди: например, всякий раз, когда указатель конца достигнет верхней границы памяти, сдвигать все непустые элементы очереди к началу области памяти, но как этот, так и другие варианты требуют перемещения в памяти элементов очереди и менее эффективны, чем кольцевая очередь.

4.5.2.1. Очереди с приоритетами

В реальных задачах возникает необходимость в формировании списков, отличных от FIFO или LIFO. Порядок выборки элементов из таких очередей определяется приоритетами элементов. *Приоритет*в общем случае может быть представлен числовым значением, которое вычисляется на основании значений каких-либо полей элемента, либо на основании внешних факторов. Так, и FIFO, и LIFO-очереди могут трактоваться как приоритетные очереди, в которых приоритет элемента зависит от времени его включения в очередь. При выборке элемента всякий раз выбирается элемент с наибольшим приоритетом.

Очереди с приоритетами могут быть реализованы на линейных списковых структурах в смежном или связном представлении. Возможны очереди с приоритетным включением, в которых последовательность элементов очереди все время поддерживается упорядоченной, т.е. каждый новый элемент включается на то место в последовательности, которое определяется его приоритетом, а при исключении всегда выбирается элемент из начала.

Возможны очереди с приоритетным исключением – новый элемент включается всегда в конец очереди, а при исключении в очереди ищется (поиск только линейный) элемент с максимальным приоритетом и после выборки удаляется из последовательности. В обоих случаях требуется поиск, а если очередь размещается в статической памяти – еще и перемещение элементов.

4.5.2.2. Очереди в вычислительных системах

Типичным примером кольцевой очереди в вычислительной системе является буфер клавиатуры BIOS(Базовой Системы Ввода-Вывода) IBM PC. Буфер клавиатуры занимает последовательность байтов памяти по адресам от 40:1E до 40:2D включительно. По адресам 40:1A

и 40:1C располагаются указатели на начало и конец очереди соответственно. При нажатии на любую клавишу генерируется прерывание 9.

Обработчик прерывания читает код нажатой клавиши и помещает его в буфер клавиатуры – в конец очереди. Коды нажатых клавиш могут накапливаться в буфере клавиатуры, прежде чем они будут прочитаны программой. Программа при вводе данных с клавиатуры обращается к прерыванию 16h. Обработчик этого прерывания выбирает код клавиши из буфера (из начала очереди) и передает в программу.

Очередь является одним из ключевых понятий в многозадачных операционных системах (Windows NT, Unix, OS/2 и др.). Ресурсы вычислительной системы (процессор, оперативная память, внешние устройства и т.п.) используются всеми задачами, одновременно выполняемыми в среде. Поскольку многие виды ресурсов не допускают одновременного использования разными задачами, такие ресурсы предоставляются задачам поочередно.

Задачи, претендующие на использование того или иного ресурса, выстраиваются в очередь к этому ресурсу. Такие очереди обычно приоритетные, однако, довольно часто применяются и FIFO-очереди, т.к. это единственная логическая организация очереди, которая гарантированно не допускает постоянного вытеснения задачи более приоритетными. LIFO-очереди обычно используются операционными системами для учета свободных ресурсов.

4.5.3. Деки

Дек (от англ. deq – double ended queue, т.е. очередь с двумя концами) – особый вид очереди в котором включение и исключение элементов может осуществляться с любого из двух концов списка. Частный случай – дек с ограниченным входом и дек с ограниченным выходом. Логическая и физическая структуры деки аналогичны структурам кольцевой FIFO-очереди.

Операции над деком заключаются в определении размера, очистки, включение/исключение элемента справа/слева. На рис. 5.2 в качестве примера показана последовательность состояний дека при включении и удалении пяти элементов. На каждом этапе стрелка указывает, с какого конца дека (левого или правого) осуществляется включение или

исключение элемента. Элементы соответственно обозначены буквами A, B, C, D, E.

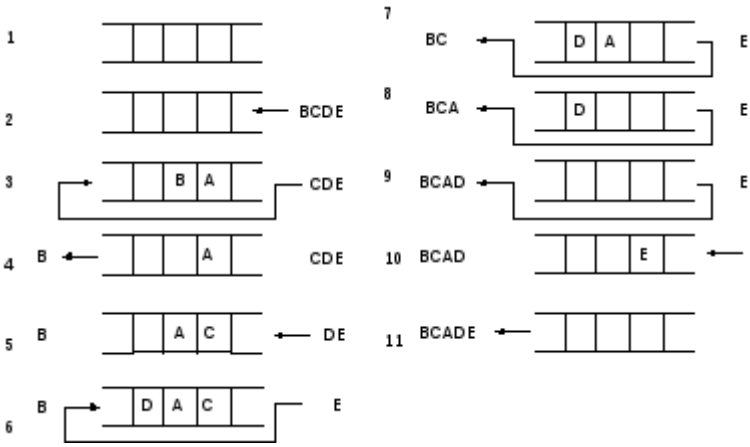


Рис. 5.2. Состояния дека в процессе изменения.

4.5.3.1. Деки в вычислительных системах

Задачи, требующие структуры дека, встречаются в вычислительной технике гораздо реже, чем задачи, реализуемые на структуре стека или очереди. Как правило, вся организация дека выполняется без каких-либо специальных средств системной поддержки. В качестве примера такой системной поддержки рассмотрим организацию буфера ввода в языке REXX. В обычном режиме буфер ввода связан с клавиатурой и работает как FIFO-очередь. В REXX имеется возможность назначить в качестве буфера ввода программный буфер и направить в него вывод программ и системных утилит.

В распоряжении пользователя имеются операции QUEUE – запись строки в конец буфера и PULL – выборка строки из начала буфера. Дополнительная операция PUSH – запись строки в начало буфера – превращает буфер в дек с ограниченным выходом. Такая структура буфера ввода позволяет программировать на REXX весьма гибкую конвейерную обработку с направлением выхода одной программы на вход другой и модификацией перенаправляемого потока.

4.5.4. Строки

Строка*– линейно упорядоченная последовательность символов, принадлежащих конечному множеству символов, называемому *алфавитом. Строки обладают следующими свойствами:

- длина, как правило, переменна, хотя алфавит фиксирован;
- обычно обращение к символам строки идет с одного конца последовательности, т.е. важна упорядоченность последовательности; в связи с этим свойством строки часто называют также цепочками;
- чаще всего целью доступа к строке является не отдельный ее элемент (хотя это тоже не исключается), а некоторая цепочка символов в строке.

Говоря о строках, обычно имеют в виду *текстовые строки*– строки, состоящие из символов, входящих в алфавит какого-либо выбранного языка, цифр, знаков препинания и других служебных символов. Однако следует иметь в виду, что символы, входящие в строку могут принадлежать любому алфавиту. Так, в языке PL/1, наряду с типом данных «символьная строка» CHAR(n) существует тип данных «битовая строка» BIT(n). Битовые строки, составляются из однобитовых символов, принадлежащих алфавиту: {0,1}. Все строковые операции применимы как к символьным, так и к битовым строкам.

Хотя строки рассматриваются в главе, посвященной полустатическим структурам данных, в тех или иных конкретных задачах изменчивость строк может варьироваться от полного ее отсутствия до практически неограниченных возможностей изменения. В большинстве языков программирования (C, PASCASL, PL/1 и др.) строки представлены полустатическими структурами.

В зависимости от назначения языка средства работы со строками занимают в языке более или менее значимое место. Язык С является языком системного программирования. Типы данных, с которыми работает язык С, максимально приближены к типам, с которыми работают машинные команды. Поскольку машинные команды не работают со строками, в языке С нет такого типа данных. Строки в С представляются в виде массивов символов. Операции над строками

могут быть выполнены как операции обработки массивов или при помощи библиотечных (не встроенных) функций строковой обработки.

В языках универсального назначения строковый тип обычно является базовым: STRING в PASCAL, CHAR(n) в PL/1. Основные операции над строками реализованы как базовые операции или встроенные функции. Специальные библиотеки обеспечивают расширенный набор строковых операций.

Язык REXX ориентирован на обработку текстовой информации. Поэтому в языке нет средств описания типов данных: все данные представляются в виде символьных строк. Операции над данными, не свойственные символьным строкам, выполняются специальными функциями, либо приводят к прозрачному преобразованию типов. Например, интерпретатор REXX, встретив оператор, содержащий арифметическое выражение, сам переводит его операнды в числовой тип, вычислит выражение и преобразует результат в символьную строку. Целый ряд строковых операций является простыми операциями языка, а встроенных функций обработки строк в REXX несколько десятков.

4.5.4.1. Операции над строками

Базовыми операциями над строками являются:

- определение длины строки;
- присваивание строк;
- конкатенация (сцепление) строк;
- выделение подстроки;
- поиск вхождения.

Операция определения длины строки есть функция, возвращаемое значение которой – целое число – текущее число символов в строке. Операция присваивания имеет тот же смысл, что и для других типов данных.

Сравнение строк производится по следующим правилам. Сравниваются первые символы двух строк в соответствии с лексикографическими правилами: если символы не равны, то строка, содержащая символ, место которого в алфавите ближе к началу,

считается меньшей. Если символы равны, сравниваются вторые, затем третьи символы и т.д. При достижении конца одной из строк строка меньшей длины считается меньшей. При равенстве длин строк и попарном равенстве всех символов в них строки считаются равными.

Результатом операции сцепления двух строк является строка, длина которой равна суммарной длине строк-операндов. Как и во всех операциях над строками, которые могут увеличивать длину строки (присваивание, сцепление, сложные операции), возможен случай, когда длина результата окажется большей, чем отведенный для него объем памяти. Проблема возникает только в тех языках, где длина строки ограничивается. Возможны три варианта решения проблемы, определяемые правилами языка или режимами компиляции:

- не контролировать превышение, возникновение такой ситуации неминуемо приведет к труднолокализуемой ошибке;
- завершать программу аварийно с локализацией и диагностикой ошибки;
- ограничивать длину результата в соответствии с объемом отведенной памяти;

Операция выделения подстроки выделяет из исходной строки последовательность символов, начиная с заданной позиции n, с заданной длиной l. В языке PASCAL соответствующая функция называется COPY. В языках PL/1, REXX соответствующая функция – SUBSTR – обладает свойством, отсутствующим в PASCAL: она может применяться в левой части оператора присваивания. Например, если исходное значение некоторой строки S – 'ABCDEFG', то выполнение оператора SUBSTR(S,3,3)='012' изменит значение строки S на – 'AB012FG'.

Операция поиска находит место первого вхождения подстроки-эталона в исходную строку. Результатом операции может быть номер позиции в исходной строке, с которой начинается вхождение эталона или указатель на начало вхождения. В случае отсутствия вхождения результатом операции должно быть некоторое специальное значение, например, нулевой номер позиции или пустой указатель.

На основе базовых операций могут быть реализованы и любые другие более сложные операции. Например, операция удаления из строки

символов с номерами от n1 включительно n2 может быть реализована как последовательность следующих шагов:

- выделение из исходной строки подстроки, начиная с позиции 1, длиной n1-1;
- выделение из исходной строки подстроки, начиная с позиции n2+1, длиной длина исходной строки – n2;
- сцепление подстрок, полученных на предыдущих шагах.

В целях повышения эффективности некоторые вторичные операции могут быть реализованы как базовые – по собственным алгоритмам, с непосредственным доступом к физической структуре строки.

4.5.4.2. Представление строк в памяти

Представление строк в памяти зависит от требований изменчивости строки и средства такого представления варьируются от абсолютно статических до динамических. Универсальные языки в основном обеспечивают работу со строками переменной длины, но максимальная длина строки должна быть указана при ее создании. Если возможностей, предоставляемых языком, недостаточно следует:

- определить новый тип данных «строка» и использовать его для представления средства динамической работы с памятью;
- выбрать язык, ориентированный на обработку текста (SNOBOL, REXX), в котором представление строк базируется на динамическом управлении памятью.

Векторное представление строк. Представление строк в виде векторов, принятое в большинстве универсальных языков, позволяет работать со строками, размещенными в статической памяти. Кроме того, векторное представление позволяет обращаться к отдельным символам строки как к элементам вектора – по индексу.

Самым простым способом является представление строки в виде вектора постоянной длины. При этом в памяти отводится фиксированное количество байт, в которые записываются символы. Если строка меньше отводимого под нее вектора, то лишние места заполняются пробелами, а если строка выходит за пределы вектора, то

лишние (обычно справа строки) символы должны быть отброшены. На рис. 5.3 приведена схема, на которой показано представление двух строк: 'ABCD' и 'PQRSTUVW' в виде вектора постоянной длины на шесть символов.

A B D

P Q R S T U

Рис. 5.3. Представление строк векторами постоянной длины.

Представление строк вектором переменной длины с признаком конца. Данный метод и все последующие учитывают переменную длину строк. Признак конца строки – особый символ, принадлежащий алфавиту (полезный алфавит оказывается меньше на один символ), и занимает то же количество разрядов, что и все остальные символы. Издержки памяти составляют 1 символ на строку. Такое представление строки показано на рис. 5.4. Специальный символ-маркер конца строки обозначен как 'eos'. В языке С, например, в качестве маркера конца строки используется символ с кодом 0.

A B D eos

P Q R S T U W eos

Рис. 5.4. Представление строк переменной длины с признаком конца.

Представление строк вектором переменной длины со счетчиком. Счетчик символов – целое число, и для него отводится достаточное количество битов, чтобы их с избытком хватило для представления длины самой длинной строки. Обычно для счетчика отводят от 8 до 16 битов. При таком представлении издержки памяти в расчете на одну строку составляют 1-2 символа.

При использовании счетчика символов возможен произвольный доступ к символам в пределах строки, поскольку можно легко проверить, что обращение не выходит за пределы строки. Счетчик размещается в таком месте, где он может быть легко доступен – в начале строки или в дескрипторе.

Максимально возможная длина строки ограничена разрядностью счетчика. В PASCAL, например, строка представляется в виде массива символов, индексация в котором начинается с 0. Однобайтный счетчик числа символов в строке является нулевым элементом этого массива. Такое представление строк показано на рис. 5.5. И счетчик символов, и признак конца в предыдущем случае могут быть доступны как элементы вектора.

3 A b d 8 p q r s t u V w

Рис. 5.5. Представление строк переменной длины со счетчиком.

В двух предыдущих вариантах обеспечивалось максимально эффективное расходование памяти (1-2 «лишних» символа на строку), но изменчивость строки была крайне невысока. Поскольку вектор – статическая структура, каждое изменение длины строки требует создания нового вектора, пересылки в него неизменяемой части строки и уничтожения старого вектора, что сводит на нет преимущества работы со статической памятью. Поэтому наиболее популярным способом представления строк в памяти являются вектора с управляемой длиной.

Вектор с управляемой длиной. Память под вектор с управляемой длиной отводится при создании строки и ее размер, и размещение остаются неизменными все время существования строки. В дескрипторе такого вектора-строки может отсутствовать начальный индекс, т.к. он может быть зафиксирован раз и навсегда установленными соглашениями, но появляется поле текущей длины строки.

Размер строки может изменяться от 0 до значения максимального индекса вектора. «Лишняя» часть отводимой памяти может быть заполнена любыми кодами – она игнорируется. Поле конечного индекса может быть использовано для контроля превышения строкой объема отведенной памяти. Представление строк в виде вектора с

управляемой длиной (при максимальной длине 10) показано на рис. 5.6.

Хотя такое представление строк не обеспечивает экономии памяти, проектировщики систем программирования, как видно, считают это приемлемой платой за возможность работать с изменчивыми строками в статической памяти.

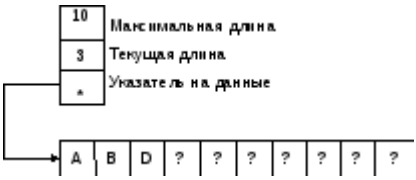


Рис. 5.6. Представление строк вектором с управляемой длиной.

Символьно-связное представление строк. Списковое представление строк в памяти обеспечивает гибкость в выполнении разнообразных операций над строками (операций включения и исключения отдельных символов и целых цепочек) и использование системных средств управления памятью при выделении необходимого объема памяти для строки. Однако при этом возникают дополнительные расходы памяти. Другим недостатком спискового представления строки является то, что логически соседние элементы строки не являются физически соседними в памяти. Это усложняет доступ к группам элементов строки по сравнению с доступом в векторном представлении строки.

Однонаправленный линейный список. Каждый символ строки представляется в виде элемента связного списка. Элемент содержит код символа и указатель на следующий элемент (рис. 5.7). Одностороннее сцепление представляет доступ только в одном направлении вдоль строки. На каждый символ строки необходим один указатель, который обычно занимает 2-4 байта.

A

B

D nil

Рис. 5.7. Представление строки однонаправленным связным списком.

Двунаправленный линейный список. В каждый элемент списка добавляется указатель на предыдущий элемент (рис. 5.8). Двустороннее сцепление допускает двустороннее движение вдоль списка, что может значительно повысить эффективность выполнения строковых операций. При этом на каждый символ строки необходимо два указателя, т.е. 4-8 байт.

nil A

B

D nil

Рис. 5.8. Представление строки двунаправленным связным списком.

Блочно-связное представление строк позволяет в большинстве операций избежать затрат, связанных с управлением динамической памятью, но в то же время обеспечивает эффективное использование памяти при работе со строками переменной длины.

Многосимвольные звенья фиксированной длины.
Многосимвольные группы (звенья) организуются в список, так что каждый элемент списка, кроме последнего, содержит группу элементов строки и указатель следующего элемента списка. Поле указателя последнего элемента списка хранит признак конца – пустой указатель.

В процессе обработки строки из любой ее позиции могут быть исключены или в любом месте вставлены элементы, в результате чего звенья могут содержать меньшее число элементов, чем было

первоначально. По этой причине необходим специальный символ, который означал бы отсутствие элемента в соответствующей позиции строки. Обозначим такой символ 'emp', он не должен входить в множество символов, из которых организуется строка. Пример многосимвольных звеньев фиксированной длинны по 4 символа в звене показан на рис. 5.9.

Представление обеспечивает более эффективное использование памяти, чем символьно-связное. Операции вставки/удаления в ряде случаев могут сводиться к вставке/удалению целых блоков. Однако при удалении одиночных символов в блоках могут накапливаться пустые символы emp, что может привести даже к худшему использованию памяти, чем в символьно-связном представлении.

A B D emp nil

P Q R S

T U V W nil

Рис. 5.9. Представление строки многосимвольными звеньями постоянной длины.

Многосимвольные звенья переменной длины. Переменная длина блока дает возможность избавиться от пустых символов и экономить память для строки. Однако появляется потребность в специальном символе – признаке указателя (на рис. 5.10 он обозначен символом 'ptr').

С увеличением длины групп символом, хранящихся в блоках, эффективность использования памяти повышается. Недостатком метода является усложнение операций по резервированию памяти для элементов списка и возврату освободившихся элементов в общий список доступной памяти.

Метод спискового представления особенно удобен в задачах редактирования текста, когда большая часть операций приходится на изменение, вставку и удаление целых текстовых блоков. Поэтому в

этих задачах целесообразно список организовать так, чтобы каждый его элемент содержал одно или несколько слов.

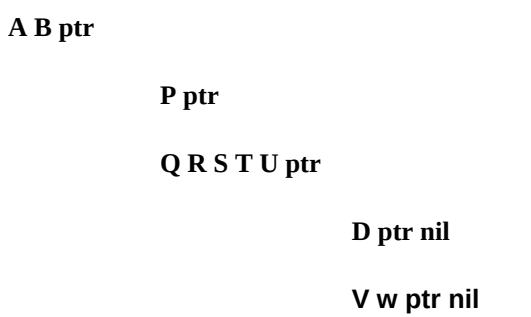


Рис. 5.10.
Представление строки многосимвольными звеньями переменной длины.

Многосимвольные звенья с управляемой длиной. Память выделяется блоками фиксированной длины. В каждом блоке помимо символов строки и указателя на следующий блок содержится номера первого и последнего символов в блоке. При обработке строки в каждом блоке обрабатываются только символы, расположенные между этими номерами. Признак пустого символа не используется: при удалении символа из строки оставшиеся в блоке символы уплотняются и корректируются граничные номера.

Вставка символа может быть выполнена за счет имеющегося в блоке свободного места, а при отсутствии – выделением нового блока. Хотя операции вставки/удаления требуют пересылки символов, диапазон пересылок ограничивается одним блоком. При каждой операции изменения может быть проанализирована степень заполнения соседних блоков, и два полупустых соседних блока могут быть переформированы в один.

Для определения конца строки может использоваться пустой указатель в последнем блоке или указатель на последний блок в дескрипторе строки. Второй способ может быть весьма полезен при выполнении некоторых операций, например, сцепления. В дескрипторе может храниться также и длина строки: считывать ее из дескриптора удобнее, чем подсчитывать перебором всех блоков строки. Пример

представления строки в виде звеньев с управляемой длиной 8 показан на рис. 5.11.

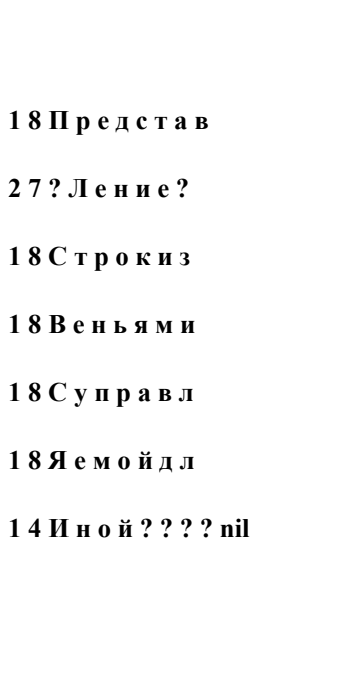


Рис. 5.11. Представление строки звеньями управляемой длины.

4.6. Связные линейные списки

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения и исключения. Список, отражающий отношения соседства между элементами, называется линейным. Списки рассматривались ранее, однако речь шла о полустатических структурах данных, и на размер списка накладывались ограничения. Если ограничения на длину списка не допускаются, то список представляется в памяти в виде

связной структуры. Линейные связные списки являются простейшими динамическими структурами данных.

Графически связи в списках удобно изображать с помощью стрелок. Если компонента не связана ни с какой другой, то в поле указателя записывают значение, не указывающее ни на какой элемент. Такая ссылка обозначается специальным именем – nil.

4.6.1. Машинное представление связных линейных списков

На рис. 6.1 приведена структура односвязного списка. В нем поле INF является информационным, поле NEXT – указатель на следующий элемент списка. Каждый список должен иметь особый элемент, называемый указателем начала списка или головой списка, который по формату обычно отличен от остальных элементов. В поле указателя последнего элемента списка находится признак nil, свидетельствующий о конце списка.

Голова списка

Inf next

Inf next

Inf nil

Рис. 6.1. Структура односвязного списка.

Обработка односвязного списка не всегда удобна, т.к. отсутствует возможность продвижения в противоположную сторону. Такую возможность обеспечивает двухсвязный список, каждый элемент которого содержит указатель на следующий и предыдущий элементы списка. Структура линейного двухсвязного списка показана на рис. 6.2. Наличие двух указателей в каждом элементе усложняет список и приводит к дополнительным затратам памяти, но в то же время обеспечивает более эффективное выполнение некоторых операций над списком.

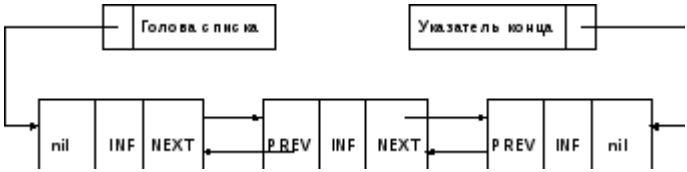


Рис. 6.2. Структура двухсвязного списка.

Разновидностью рассмотренных видов линейных списков является кольцевой список, который может быть организован на основе односвязного или двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются, как показано на рис. 6.3. При работе с такими списками несколько упрощаются некоторые процедуры, выполняемые над списком. Однако при просмотре такого списка следует принять меры предосторожности, чтобы не попасть в бесконечный цикл.

…

…

PREV INF NEXT

PREV INF NEXT

PREV INF NEXT

Голова списка

Рис. 6.3. Структура кольцевого двухсвязного списка.

В памяти список представляет совокупность дескриптора и одинаковых по размеру и формату записей, размещенных в произвольных областях памяти и связанных друг с другом в линейно

упорядоченную цепочку с помощью указателей. Запись содержит информационные поля и поля указателей на соседние элементы списка, причем некоторыми полями информационной части могут быть указатели на блоки памяти с дополнительной информацией, относящейся к элементу списка.

Дескриптор списка реализуется в виде особой записи и содержит адрес начала списка, имя списка, текущее число элементов в списке, описание элемента и т.д. Дескриптор может находиться в той же области памяти, в которой располагаются элементы списка, или для него выделяется другое место.

4.6.2. Реализация операций над связными линейными списками

В разделе рассматриваются некоторые операции над линейными списками. Выполнение операций иллюстрируется рисунками со схемами изменения связей и программными примерами. На всех рисунках сплошными линиями показаны связи, имевшиеся до выполнения и сохранившиеся после выполнения операции, а значком 'x' отмечены связи, разорванные при выполнении операции.

Во всех операциях важна последовательность коррекции указателей, которая обеспечивает корректное изменение списка, не затрагивающее другие элементы. При неправильном порядке коррекции легко потерять часть списка. Поэтому рядом с устанавливаемыми связями в скобках показаны шаги, на которых эти связи устанавливаются. Во всех примерах считаются определенными следующие типы данных:

структура информационного поля списка:

type

data = ...;

элемент односвязного списка (sll - single linked list):

sllptr = ^slltype; { указатель в односвязном списке }

slltype = **record** { элемент односвязного списка }

inf : data; { информационная часть }

next : sllptr; { указатель на следующий элемент }

end;

элемент двухсвязного списка (dll - double linked list):

dllptr = ^dlltype; { указатель в двухсвязном списке }

dlltype = **record** { элемент односвязного списка }

inf : data; { информационная часть }

next : sllptr; { указатель на следующий элемент (вперед) }

prev : sllptr; { указатель на предыдущий элемент (назад) }

end;

Перебор элементов списка.Операция перебора, возможно, чаще других выполняется над линейными списками. При ее выполнении осуществляется последовательный доступ к элементам списка – ко всем элементам до конца списка или до нахождения искомого элемента.

{ Перебор 1-связного списка }

procedure LookSll(head: sllptr);

{ head - указатель на начало списка }

var cur: sllptr; { адрес текущего элемента }

begin

cur:=head; { первый элемент списка назначается текущим }

while cur <> **nil do**

begin

< обработка c^.inf >

{ обрабатывается информационная часть

эл-та, на который указывает cur.

Обработка может состоять в:

- печати содержимого инф. части;
- модификации полей инф. части;
- сравнения полей инф. части с образцом при поиске по ключу; }

cur:=cur^.next;

{ из текущего элемента выбирается указатель

на следующий элемент и для следующей итерации

следующий элемент становится текущим; если текущий

элемент был последний, то его поле next содержит

пустой указатель и в cur запишется nil, что приведет

к выходу из цикла при проверке условия while }

end;

end;

В двухсвязном списке возможен перебор как в прямом направлении (выглядит так же, как и перебор в односвязном списке), так и в обратном. В последнем случае параметром процедуры должен быть указатель на конец списка, и переход к следующему элементу должен осуществляться по указателю назад:

cur:=cur^.prev;

В кольцевом списке окончание перебора должно происходить не по признаку последнего элемента – такой признак отсутствует, а по достижению элемента, с которого начался перебор.

Вставка элемента в список. Вставка элемента в середину односвязного списка показана на рис. 6.4.

…

prev

Inf next

Inf next

Inf next

Inf next

…

cur

(1)

Рис. 6.4. Вставка элемента в середину 1-связного списка.

Следующий пример демонстрирует реализацию этой операции.

{ Вставка элемента в середину 1-связного списка }

```
procedure InsertSll(prev: sllptr; inf: data);
{ prev - адрес предыдущего эл-та; inf - данные нового элемента }
var cur: sllptr; { адрес нового элемента }
begin
{ Создание нового элемента – выделение
памяти для него и запись инф. части }
New(cur);
cur^.inf:=inf;
{ элемент, следовавший за предыдущим
теперь будет следовать за новым }
cur^.next:=prev^.next;
{ новый элемент следует за предыдущим }
prev^.next:=cur;
end;
```

Рисунок 6.5 представляет вставку в двухсвязный список.

(2)

(1)

(3)

(4)

cur

prev

…

…

PREV INF NEXT

PREV INF NEXT

PREV INF NEXT

PREV INF NEXT

…

…

Рис. 6.5. Вставка элемента в середину 2-связного списка.

Приведенные примеры обеспечивают вставку в середину списка, но не могут быть применены для вставки в начало списка. В этом случае должен модифицироваться указатель на начало списка, как показано на рис. 6.6.

head – указатель на начало

cur

(1)

(2)

…

Inf next

Inf next

Inf next

Рис. 6.6. Вставка элемента в начало 1-связного списка.

В примере процедура выполняет вставку элемента в любое место односвязного списка.

```
{ Вставка элемента в любое место 1-связного списка }

procedure InsertSll(

var

{ указатель на начало списка, может измениться в

процедуре, если head=nil - список пустой }

head: sllptr;

{ указатель на элемент, после которого делается вставка,

если prev=nil - вставка перед первым элементом }

prev: sllptr;

inf: data { данные нового элемента }

cur: sllptr); { адрес нового элемента }

begin
```

```
{ выделение памяти для нового элемента и запись его инф. части }

New(cur);

cur^.inf:=inf;

{ если есть предыдущий элемент - вставка в середину списка }

if prev <> nil then

begin

cur^.next:=prev^.next;

prev^.next:=cur;

end else

{ вставка в начало списка }

begin

{ новый элемент указывает на бывший первый элемент списка;

если head = nil, то новый элемент будет и последним элементом списка
}

cur^.next:=head;

{ новый элемент становится первым в списке,

указатель на начало теперь указывает на него }

head:=cur;

end;

end;
```

Удаление элемента из списка. Удаление элемента из односвязного списка показано на рис. 6.7. Процедуру удаления легко выполнить, если известен адрес элемента, предшествующего удаляемому (prev на рис. 6.7. а). Однако на рис. 6.7 и в примере приводится процедура для случая, когда удаляемый элемент задается своим адресом del.

prev

…

Inf next

INF NEXT

INF NEXT

del

…

а).

del

head

INF NEXT

INF NEXT

…

б).

Рис. 6.7. Удаление элемента из 1-связного списка из середины списка (а) и из начала (б).

Процедура обеспечивает удаление из середины и из начала списка.

```
{ Удаление элемента из любого места 1-связного списка }

procedure DeleteSll(

var head: sllptr; { указатель на начало списка, может измениться в процедуре }

del: sllptr); { указатель на элемент, который удаляется }

var prev: sllptr; { адрес предыдущего элемента }

begin

{ попытка удаления из пустого списка расценивается как ошибка

(в последующих примерах этот случай учитываться не будет) }

if head = nil then

begin

Writeln('Ошибка!');

Halt;

end;

{ если удаляемый элемент - первый в списке,

то следующий за ним становится первым }

if del = head then

head:=del^.next else

{ удаление из середины списка }

begin
```

```
{ приходится искать элемент, предшествующий удаляемому; поиск
производится перебором списка с самого его начала, пока не будет
найдет элемент, поле next которого совпадает с адресом удаляемого
элемента }

prev:=head^.next;

while (prev^.next <> del) and (prev^.next <> nil) do

prev:=prev^.next;

if prev^.next = nil then

begin

{ это случай, когда перебран весь список, но элемент не найден,

он отсутствует в списке; расценивается как ошибка }

Writeln('Ошибка!');

Halt;

end;

{ предыдущий элемент теперь указывает на следующий за удаляемым
}

prev^.next:=del^.next;

end;

{ элемент исключен из списка, теперь можно освободить занимаемую
им память }

Dispose(del);

end;
```

Удаление элемента из двухсвязного списка требует коррекции большего числа указателей, как показано на рис. 6.8. Процедура удаления, чем для односвязного списка, так как в ней не нужен поиск предыдущего элемента (выбирается по указателю назад).

(1)

(2)

del

…

…

PREV INF NEXT

PREV INF NEXT

PREV INF NEXT

…

…

Рис. 6.8. Удаление элемента из 2-связного списка.

Перестановка элементов списка.Изменчивость динамических структур предполагает не только изменения размера структуры, но и изменения связей между элементами. В качестве примера приведем перестановку двух соседних элементов списка (рис. 6.9). В алгоритме предполагается известным адрес элемента, предшествующего паре, в которой производится перестановка. В алгоритме также не учитывается случай перестановки первого и второго элементов.

(3)

(2)

prev

…

…

Inf next

Inf next

Inf next

Inf next

(1)

Рис. 6.9. Перестановка соседних элементов 1-связного списка.

{ Перестановка двух соседних элементов в 1-связном списке }

procedure ExchangeSll(

prev: sllptr { указатель на элемент, предшествующий переставляемой паре });

var p1, p2: sllptr); { указатели на элементы пары }

begin

p1:=prev^.next; { указатель на 1-й элемент пары }

p2:=p1^.next; { указатель на 2-й элемент пары }

p1^.next:=p2^.next; { 1-й элемент пары указывает на следующий за парой }

p2^.next:=p1; { 1-й элемент пары теперь следует за 2-ым }

prev^.next:=p2; { 2-й элемент пары теперь становится 1-ым }

end;

В процедуре перестановки для двухсвязного списка нетрудно учесть и перестановку в начале и конце списка (рис. 6.10).

(5)

(2)

(4)

(6)

(3)

(1)

…

…

PREV INF NEXT

…

…

PREV INF NEXT

PREV INF NEXT

Рис. 6.10.
Перестановка соседних элементов 2-связного списка.

Копирование части списка.При копировании исходный список сохраняется в памяти, и создается новый список. Информационные поля элементов нового списка содержат те же данные, что и в элементах старого списка, но поля связок в новом списке совершенно другие, поскольку элементы нового списка расположены по другим адресам в памяти. Существенно, что операция копирования предполагает дублирование данных в памяти.

{ Копирование части 1-связного списка; head - указатель на начало копируемой

части; num - число элементов. Функция возвращает указатель на список-копию }

function CopySll(head: sllptr; num: integer): sllptr;

var cur, head2, cur2, prev2: sllptr;

begin

{ исходный список пуст - копия пуста }

if head = **nil then**

CopySll:=**nil else**

```
begin
cur:=head;
prev2:=nil;
{ перебор исходного списка до конца или по счетчику num }
while (num > 0) and (cur <> nil) do
begin
{ выделение памяти для элемента выходного списка и
запись в него информационной части }
New(cur2);
cur2^.inf:=cur^.inf;
{ если 1-й эл-т выходного списка - запоминается указатель на
начало, иначе - записывается указатель в предыдущий элемент }
if prev2 <> nil then
prev2^.next:=cur2 else head2:=cur2;
prev2:=cur2; { текущий элемент становится предыдущим }
cur:=cur^.next; { продвижение по исходному списку }
num:=num-1; { подсчет элементов }
end;
cur2^.next:=nil; { пустой указатель - в последний элемент выходного
списка }
```

```
CopySll:=head2; { вернуть указатель на начало выходного списка }

end;

end;
```

Слияние двух списков.Операция слияния заключается в формировании из двух списков одного и аналогична операции сцепления строк. В случае односвязного списка слияние выполняется просто. Последний элемент первого списка содержит пустой указатель на следующий элемент, этот указатель служит признаком конца списка. Вместо пустого указателя в последний элемент первого списка заносится указатель на начало второго списка. Таким образом, второй список становится продолжением первого.

```
{ Слияние двух списков. head1 и head2 - указатели на начала

списков. На результирующий список указывает head1 }

procedure UniteSll(var head1, head2: sllptr);

var cur: sllptr;

begin { если 2-й список пустой - нечего делать }

if head2 <> nil then

begin

{ если 1-й список пустой, выходным списком будет 2-й }

if head1 = nil then

head1:=head2 else

{ перебор 1-го списка до последнего его элемента }

begin
```

```
cur:=head1;

while cur^.next <> nil do

cur:=cur^.next;

{ последний элемент 1-го списка указывает на начало 2-го }

cur^.next:=head2;

end;

head2:=nil; { 2-й список аннулируется }

end;

end;
```

4.6.3. Применение линейных списков

Линейные списки находят широкое применение, когда непредсказуемы требования на размер памяти, необходимой для хранения данных, большое число сложных операций над данными, особенно включений и исключений. На базе линейных списков могут строиться стеки, очереди и деки. Представление очереди с помощью линейного списка позволяет достаточно просто обеспечить желаемые механизмы обслуживания очереди.

Линейные связные списки используются также для представления таблиц в случаях, когда размер таблицы может существенно изменяться в процессе работы с ней. Однако доступ к элементам связного линейного списка может быть только последовательным, что не позволяет применить к такой таблице эффективный двоичный поиск и это существенно ограничивает их применимость.

Поскольку упорядоченность такой таблицы не может помочь в организации поиска, задачи сортировки таблиц, представленных линейными связными списками, возникают значительно реже, чем для

таблиц в векторном представлении. Однако, в некоторых случаях для таблицы, хотя и не требуется частое выполнение поиска, но задача генерации отчетов требует расположения записей таблицы в некотором порядке.

Некоторые алгоритмы, возможно, потребуют каких-либо усложнений структуры, например, быструю сортировку Хоара целесообразно проводить только на двухсвязном списке; в цифровой сортировке удобно создавать промежуточные списки для цифровых групп и т.д.

Приведем примеры сортировки элементов односвязного линейного списка. Будем полагать, что определен следующий тип данных:

```
type

lptr = ^item; { указатель на элемент списка }

item = record { элемент списка }

key : Integer; { ключ }

inf : data; { данные }

next: lptr; { указатель на элемент списка }

end;
```

Сортировку будем вести по возрастанию ключей. Параметром функции сортировки будет указатель на начало неотсортированного списка, а возвращает функция указатель на начало отсортированного списка. Прежний, несортированный список перестает существовать.

Следующий пример демонстрирует сортировку выборкой. Указатель newh является адресом начала выходного списка, исходно пустого. Во входном списке ищется максимальный элемент. Найденный элемент исключается из входного списка и включается в начало выходного списка. Работа алгоритма заканчивается, когда входной список станет пустым.

```
{ Сортировка выборкой на 1-связном списке }

function Sort(head: lptr): lptr;

var newh, max, prev, pmax, cur: lptr;

begin

newh:=nil; { выходной список - пустой }

while head <> nil do { цикл, пока не опустеет входной список }

begin

max:=head;

prev:=head; { нач. максимум - 1-й элемент }

cur:=head^.next; { поиск максимума во входном списке }

while cur <> nil do

begin

if cur^.key > max^.key then

begin

{ запоминается адрес максимума и адрес предыдущего элемента }

max:=cur;

pmax:=prev;

end;

prev:=cur;
```

```
cur:=cur^.next; { движение по списку }

end;

{ исключение максимума из входного списка }

if max = head then

head:=head^.next else

pmax^.next:=max^.next;

{ вставка в начало выходного списка }

max^.next:=newh;

newh:=max;

end;

Result:=newh;

end;
```

Следует обратить внимание на несколько особенностей алгоритма. Во-первых, во входном списке ищется всякий раз не минимальный, а максимальный элемент. Поскольку элемент включается в начало выходного списка, элементы с большими ключами оттесняются к концу выходного списка и последний, таким образом, оказывается отсортированным по возрастанию ключей.

Во-вторых, при поиске во входном списке сохраняется не только адрес найденного элемента в списке, но и адрес предшествующего ему в списке элемента – это впоследствии облегчает исключение элемента из списка.

В-третьих, обратите внимание на то, что не возникает проблем с пропуском во входном списке тех элементов, которые уже выбраны – они просто исключены из входной структуры данных.

В следующем примере представлена реализация сортировки вставками. Из входного списка выбирается (и исключается) первый элемент и вставляется в выходной список «на свое место» в соответствии со значениями ключей. Сортировка включением на векторной структуре требовала большого числа перемещений элементов в памяти. Обратите внимание, в обоих примерах пересылок данных не происходит, все элементы остаются на своих местах в памяти, меняются только связи между ними – указатели.

```
type

data = Integer;

{ Сортировка вставками на 1-связном списке }

function Sort(head: lptr): lptr;

var newh, cur, sel: lptr;

begin

newh:=nil; { выходной список - пустой }

while head <> nil do

begin { цикл, пока не опустеет входной список }

sel:=head; { элемент, который переносится в выходной список }

head:=head^.next; { продвижение во входном списке }

{ выходной список пустой или элемент меньше 1-го - вставка в начало
}

if (newh = nil) or (sel^.key < newh^.key) then

begin

sel^.next:=newh;
```

```
newh:=sel;
end;
end else
{ вставка в середину или в конец }
begin
cur:=newh;
{ до конца выходного списка или пока ключ
следующего элемента не будет больше вставляемого }
while (cur^.next <> nil) and (cur^.next^.key < sel^.key) do
cur:=cur^.next;
{ вставка в выходной список после элемента cur }
sel^.next:=cur^.next;
cur^.next:=sel;
end;
Result:=newh;
end;
```

4.6.4. Нелинейные разветвленные списки

4.6.4.1. Основные понятия

*Нелинейным разветвленным списком*является список, элементами которого могут быть также списки. В разделе 4.6.2 были рассмотрены

двухсвязные линейные списки. Если один из указателей каждого элемента задает порядок, обратный к порядку, устанавливаемому другим указателем, то такой двусвязный список будет линейным. Однако если один из указателей задает порядок произвольного вида, не являющийся обратным по отношению к порядку, устанавливаемому другим указателем, то такой список будет нелинейным.

В обработке нелинейный список определяется как любая последовательность *атомовиподсписков*, где в качестве атома берется любой объект, который при обработке отличается от списка тем, что он структурно неделим. Если заключим списки в круглые скобки, а элементы списков разделим запятыми, то в качестве списков можно рассматривать такие последовательности:

(a,(b,c,d),e,(f,g))

( )

((a))

Первый список содержит четыре элемента: атом a, список (b,c,d), содержащий в свою очередь атомы b,c,d, атом e и список (f,g), элементами которого являются атомы f и g. Второй список не содержит элементов, тем не менее, нулевой список, в соответствии с определением является действительным списком. Третий список состоит из одного элемента: списка (a), который в свою очередь содержит атом a.

Другой способ представления, часто используемый для иллюстрации списков, – графические схемы, аналогичен способу представления, применяемому при изображении линейных списков. Каждый элемент списка обозначается прямоугольником; стрелки или указатели показывают, являются ли прямоугольники элементами одного и того же списка или элементами подсписка. Пример такого представления показан на рис. 6.11.

a

nil

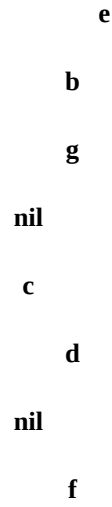


Рис. 6.11. Схематическое представление разветвленного списка.

Разветвленные списки описываются тремя характеристиками: порядком, глубиной и длиной.

Порядок. Над элементами списка задается транзитивное отношение, определяемое последовательностью, в которой элементы появляются внутри списка. В списке (x,y,z) атом x предшествует y, а y предшествует z. При этом подразумевается, что x предшествует z. Данный список не эквивалентен списку (y,z,x). При представлении списков графическими схемами порядок определяется горизонтальными стрелками. Горизонтальные стрелки истолковываются следующим образом: элемент, из которого исходит стрелка, предшествует элементу, на который она указывает.

Глубина. Максимальный уровень, приписываемый элементам внутри списка или внутри любого подсписка в списке. Уровень элемента описывается вложенностью подсписков внутри списка, т.е. числом пар круглых скобок, окаймляющих элемент.

В списке, изображенном на рис. 6.11, элементы a и e находятся на уровне 1, в то время как оставшиеся элементы – b, c, d, f и g имеют уровень 2. Глубина входного списка равна 2. При представлении списков схемами концепции глубины и уровня облегчаются для понимания, если каждому атомарному или списковому узлу приписать некоторое число l. Значение l для элемента x, обозначаемое как l(x), является числом вертикальных стрелок, которое необходимо пройти для того, чтобы достичь данный элемент из первого элемента списка. Для нашего примера l(a)=0, l(b)=1 и т.д. Глубина списка является максимальным значением уровня среди уровней всех атомов списка.

Длина – число элементов уровня 1 в списке. Например, длина списка нашего примера равна 3.

Типичный пример применения разветвленного списка – представление алгебраического выражения в виде списка. Алгебраическое выражение можно представить в виде последовательности элементарных двухместных операций вида:

< операнд 1 > < знак операции > < операнд 2 >

Выражение (a+b)*(c-(d/e))+f

будет вычисляться в следующем порядке:

a+b

d/e

c-(d/e)

(a+b)*(c-d/e)

(a+b)*(c-d/e)+f

При представлении выражения в виде разветвленного списка каждая тройка «операнд-знак-операнд» представляется в виде списка, причем, в качестве операндов могут выступать как атомы – переменные или константы, так и подсписки такого же вида (рис. 6.12). Скобочное представление выражения будет иметь вид:

(((a,+,b),*,(c,-,(d,/,e)),+,f)

Глубина списка равна 4, длина - 3.

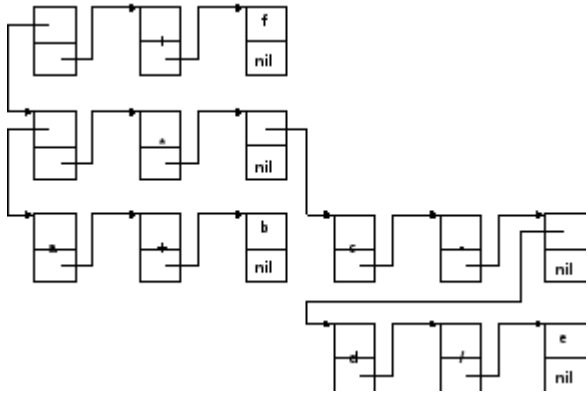


Рис. 6.12. Схема списка, представляющего алгебраическое выражение.

4.6.4.2. Представление списковых структур в памяти

В соответствии со схематичным изображением разветвленных списков типичная структура элемента такого списка в памяти должна быть такой, как показано на рис. 6.13.

data – данные атома

down – указатель на подсписок того же уровня

next – указатель на следующий элемент

Рис. 6.13. Структура элемента разветвленного списка.

Элементы списка могут быть двух видов: атомы, содержащие данные, и узлы, содержащие указатели на подсписки. В атомах не используется

поле down элемента списка, а в узлах – поле data. Поэтому логичным является совмещение этих двух полей в одно, как показано на рис. 6.14.

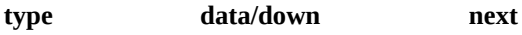


Рис. 6.14. Структура элемента разветвленного списка.

Поле type содержат признак атом/узел и может быть однобитовым. Такой формат элемента удобен для списков, атомарная информация которых занимает небольшой объем памяти. В этом случае теряется незначительный объем памяти в элементах списка, для которых не требуется поля data. В общем случае для атомарной информации необходим относительно большой объем памяти. Наиболее распространенный в данной ситуации формат структуры узла представленный на рис. 6.15.

Рис. 6.15. Структура элемента разветвленного списка.

В этом случае указатель down указывает на данные или на подсписок. Поскольку списки могут составляться из данных различных типов, целесообразно адресовать указателем down не непосредственно данные, а их дескриптор, в котором может быть описан тип данных, их длина и т.п. Описание того, является ли адресуемый указателем данных объект атомом или узлом, также может находиться в этом дескрипторе.

4.6.4.3. Операции обработки списков

Базовыми операциями при обработке списков являются следующие:

1. Операция car в качестве аргумента получает список (указатель на начало списка). Возвращаемым значением является первый элемент этого списка (указатель на первый элемент):

если X = (2,6,4,7), то car(X) = атом 2;

если X = ((1,2),6), то car(X) = (1,2);

если X – атом то операция car(X) не имеет смысла.

2. Операция cdr в качестве аргумента также получает список. Возвращаемым значением является остаток списка – указатель на список после удаления из него первого элемента:

если X = (2,6,4), то cdr(X) = (6,4);

если X = ((1,2),6,5), то cdr(X) = (6,5);

если список X содержит один элемент, то cdr(X) = nil.

3. Операция cons имеет два аргумента: указатель на элемент списка и указатель на список. Операция включает аргумент-элемент в начало аргумента-списка и возвращает указатель на получившийся список:

если X = 2 и Y = (6,4,7), то cons(X,Y) = (2,6,4,7);

если X = (1,2), Y = (6,4,7), то cons(X,Y) = ((1,2),6,4,7).

4. Операция atom выполняет проверку типа элемента списка. Она должна возвращать логическое значение true, если аргумент является атомом или false, если аргумент является подсписком.

4.6.5. Язык программирования lisp

Язык LISP является наиболее развитым и распространенным языком обработки списков. Идеология и терминология языка в значительной степени повлияла на общепринятые подходы к обработке списков и использовалась и нами в предыдущем изложении.

Все данные в языке представляются в виде списков, структура элемента списка соответствует рис. 6.15. Язык LISP обеспечивает базовые функции обработки списков car, cdr, cons, atom. Также многие

вторичные функции реализованы в языке как базовые для повышения их эффективности. **Помимо списковых операций в языке обеспечиваются операции для выполнения арифметических, логических операций, отношения, присваивания, ввода-вывода и т.д.**

Сама LISP-программа представляется как список, записанный в скобочной форме. Элементами простого программного списка является операции-функции с параметрами. **Параметрами могут быть в свою очередь обращения к функциям, которые образуют подсписки. Вся программа на LISP представляет собой единственное обращение к функции с множеством вложенных обращений – рекурсивных или к другим функциям. Поэтому программирование на языке LISP часто называют «функциональным программированием».**

Системы программирования LISP строятся и как компиляторы, и как интерпретаторы. Однако, независимо от подхода к построению системы программирования, она обязательно включает в себя «сборку мусора». Система программирования LISP автоматически следит за использованием памяти и обеспечивает ее освобождение.

4.6.6. Управление динамически выделяемой памятью

Динамические структуры характеризуется непостоянством и непредсказуемостью размера. Память под отдельные элементы таких структур выделяется в момент, когда они «начинают существовать» в процессе исполнения программы, а не во время ее трансляции. Когда в элементе структуры больше нет необходимости, занимаемая им память освобождается. В современных вычислительных средах большая часть вопросов, связанных с управлением памятью решается операционными системами или системами программирования.

Для разработчика прикладных задач динамическое управление памятью вообще прозрачно, либо осуществляется через достаточно простой интерфейс стандартных процедур/функций. Однако перед системным программистом вопросы управления памятью встают гораздо чаще.

Во-первых, эти вопросы в полном объеме должны быть решены при проектировании операционных систем и систем программирования. Во-вторых, некоторые сложные приложения могут сами распределять память в пределах выделенного им ресурса. В-третьих, знание того, как в данной вычислительной среде распределяется память, позволит построить более эффективное программное изделие даже при использовании интерфейса стандартных процедур.

В общем случае при распределении памяти должны быть решены следующие вопросы:

- способ учета свободной памяти;
- механизм выделения памяти по запросу;
- обеспечение утилизации освобожденной памяти.

Рассмотрим эти вопросы по порядку. В распоряжении программы обычно имеется адресное пространство, которое может рассматриваться как последовательность ячеек памяти с адресами, линейно возрастающими от 0 до N. Какие-то части адресного пространства обычно заняты системными программами и данными, другие – кодами и статическими данными самой программы, оставшаяся часть доступна для динамического распределения.

Обычно доступная для распределения память представляет непрерывный участок пространства с адресными границами от n1 до n2. В управлении памятью при каждом запросе на память необходимо решать, по каким адресам внутри доступного участка будет располагаться выделяемая память.

В некоторых системах программирования выделение памяти автоматизировано полностью: система не только сама определяет адрес выделяемой области памяти, но и момент, когда память должна выделяться. Например, память автоматически выделяется под элементы списков в языке LISP, под символьные строки в языках SNOBOL и REXX.

В других системах программирования – к ним относится большинство универсальных процедурных языков – моменты выделения и освобождения памяти определяются пользователем. Требуется выдать запрос на выделение/освобождение памяти при помощи стандартной подпрограммы – ALLOCATE/FREE в PL/1, malloc/free в C,

New/Dispose в PASCAL. Система сама определяет размещение выделяемого блока, и затем функция выделения памяти возвращает его адрес.

Память всегда выделяется блоками – непрерывными последовательностями смежных ячеек. Блоки могут быть фиксированной или переменной длины. Фиксированный размер блока удобнее для управления: в этом случае вся доступная для распределения память разбивается на «кадры», размер каждого из которых равен размеру блока, и любой свободный кадр годен для удовлетворения любого запроса. К сожалению, лишь ограниченный круг реальных задач может быть сведен к блокам фиксированной длины.

Одной из проблем, которые должны приниматься во внимание при управлении памятью является проблема *фрагментации* (дробления) памяти. Она заключается в возникновении «дыр» – участков памяти, которые не могут быть использованы. Различаются дыры внутренние и внешние.

Внутренняя дыра – неиспользуемая часть выделенного блока, возникает, если размер выделенного блока больше запрошенного. Внутренние дыры характерны для выделения памяти блоками фиксированной длины. *Внешняя дыра* – свободный блок, который в принципе может быть выделен, но размер его слишком мал для удовлетворения запроса. Внешние дыры характерны для выделения блоками переменной длины. Управление памятью должно быть построено таким образом, чтобы минимизировать суммарный объем дыр.

Система управления памятью должна «знать», какие ячейки имеющейся в ее распоряжении памяти свободны, а какие – заняты. **Методы учета свободной памяти основываются либо на принципе битовой карты, либо на принципе списков свободных блоков**.

В методах *битовой карты* создается «карта памяти» – массив бит, в котором каждый однобитовый элемент соответствует единице доступной памяти и отражает ее состояние: 0 – свободна, 1 – занята. Если считать единицей распределения единицу адресации – байт, то сама карта памяти будет занимать 1/8 часть всей памяти, что делает ее слишком дорогостоящей. Поэтому при применении методов

битовой карты обычно единицу распределения делают более крупной, например, 16 байт. Карта, таким образом, отражает состояние каждого 16-байтного кадра. Если карту представить как строку бит, то задача поиска участка памяти для выделения будет сведена к поиску в этой строке подстроки нулей требуемой длины.

В методах *списков свободных блоков* участки свободной памяти объединяются в связные списки. В системе имеется переменная, в которой хранится адрес первого свободного участка. В начале первого свободного участка записывается его размер и адрес следующего свободного участка. В простейшем случае список свободных блоков никак не упорядочивается. Поиск выполняется перебором списка.

Механизмы выделения памяти решают вопрос, какой из свободных участков должен быть выделен по запросу. Два основных механизма сводятся к методам «самый подходящий» и «первый подходящий».

По первому методу выделяется свободный участок, размер которого равен запрошенному или превышает его на минимальную величину. По второму методу выделяется первый же найденный свободный участок, размер которого не меньше запрошенного. При применении любого способа, если размер выбранного для выделения участка превышает запрос, выделяется запрошенный объем памяти, а остаток образует свободный блок меньшего размера.

Когда в динамической структуре данных или в отдельном ее элементе нет больше необходимости, занимаемая ею память должна быть утилизована, т.е. освобождена и сделана доступной для нового распределения. В тех системах, где память запрашивается явным образом, она и освобождена должна быть явным образом.

При представлении памяти на битовой карте достаточно просто сбросить в 0 биты, соответствующие освобожденным кадрам. При учете свободной памяти списками блоков освобожденный участок должен быть включен в список, но одного этого недостаточно. Следует еще позаботиться о том, чтобы при образовании в памяти двух смежных свободных блоков они слились в один свободный блок суммарного размера. Задача слияния смежных блоков значительно упрощается при упорядочении списка свободных блоков по адресам памяти – тогда смежные блоки обязательно будут соседними элементами этого списка.

Задача утилизации усложняется в системах, где нет явного освобождения памяти: тогда на систему ложится задача определения, какие динамические структуры или их элементы уже не нужны. Один из методов решения задачи предполагает, что система не приступает к освобождению памяти до тех пор, пока свободной памяти совсем не останется. Затем все зарезервированные блоки проверяются и освобождаются те из них, которые больше не используются. Такой метод называется «сборкой мусора».

Программа сборки мусора вызывается, когда нет возможности удовлетворить некоторый частный запрос на память, или когда размер доступной области памяти стал меньше некоторой заранее определенной границы. Алгоритм сборки мусора обычно двухэтапный. На первом этапе осуществляется маркировка всех блоков, на которые указывает хотя бы один указатель. На втором этапе все неотмеченные блоки возвращаются в свободный список, а метки стираются.

Важно, чтобы в момент включения сборщика мусора все указатели были установлены на те блоки, на которые они должны указывать. Если необходимо в некоторых алгоритмах применять методы с временным рассогласованием указателей, необходимо временно отключить сборщик мусора – пока имеется такое рассогласование. Один из серьезных недостатков метода сборки мусора состоит в том, что расходы на него увеличиваются по мере уменьшения размеров свободной области памяти.

Другой метод – освобождать любой блок, как только он перестает использоваться. Метод обычно реализуется посредством счетчика ссылок, в который записывается, сколько указателей на данный блок имеется в данный момент времени. Когда значение счетчика становится равным 0, соответствующий блок оказывается недоступным и, следовательно, не используемым. Блок возвращается в свободный список.

Такой метод предотвращает накопление мусора, не требует большого числа оперативных проверок во время обработки данных. Однако у метода есть определенные недостатки. Во-первых, следует учитывать, что требуются дополнительные затраты времени и памяти на ведение счетчиков ссылок. Во-вторых, если зарезервированные блоки образуют циклическую структуру, то счетчик ссылок каждого из них никогда не

обнулится, когда все связи, идущие извне блоков в циклическую структуру, будут уничтожены.

Существуют различные решения проблемы:

- запретить циклические и рекурсивные структуры;
- отмечать циклические структуры флажками, и обрабатывать их особым образом;
- потребовать, чтобы любая циклическая структура всегда имела головной блок, счетчик циклов которого учитывал бы только ссылки от элементов, расположенных вне цикла, и чтобы доступ ко всем блокам этой структуры осуществлялся только через него.

Практическая эффективность описанных методов зависит от многих параметров, таких как частота запросов, статистическое распределение размеров запрашиваемых блоков, способ использования системы.

4.7. Нелинейные структуры данных

4.7.1. Графы и деревья

Теория графов является важной частью вычислительной математики. Многосвязная структура граф находит применение при организации банков данных, управлении базами данных, в системах имитационного моделирования сложных комплексов, в системах искусственного интеллекта, в задачах планирования. Алгоритмы обработки нелинейных разветвленных списков, к которым могут быть отнесены и графы, были приведены ранее.

Граф– нелинейная многосвязная динамическая структура, отображающая свойства и связи сложного объекта, обладающая следующими свойствами:

- на каждый элемент (узел, вершину) может быть произвольное количество ссылок;
- каждый элемент может иметь связь с любым количеством других элементов;
- каждая связка (ребро, дуга) может иметь направление и вес.

В узлах графа содержится информация об элементах объекта. Связи между узлами задаются ребрами графа. Ребра могут иметь направленность, показываемую стрелками, тогда они называются ориентированными, ребра без стрелок – неориентированные.

С точки зрения теории графов не имеет значения, какой смысл вкладывается в вершины и ребра. Вершинами могут быть населенными пункты, а ребрами дороги, соединяющие их, или вершинами являться подпрограммы, соединение вершин ребрами означает взаимодействие подпрограмм. Часто имеет значение направления дуги в графе.

Граф, все связи которого ориентированные, называется *ориентированным* или *орграфом*. Граф со всеми неориентированными связями называется *неориентированным*, а граф со связями обоих типов –*смешанным графом*. Обозначение связей: неориентированных – (A,B), ориентированных – <A,B>. Примеры изображений графов приведены на рис. 7.1. Скобочное представление имеет вид: а). ((A,B),(B,A)) и б). (< A,B >,< B,A >).

Для ориентированного графа число ребер, входящих в узел, называется *полустепенью захода* узла, выходящих из узла –*полустепенью исхода*. Количество входящих и выходящих ребер может быть любым, в том числе и нулевым. Если ребрам графа соответствуют некоторые значения, то граф и ребра называются*взвешенными*. *Мультиграфом* называется граф, имеющий параллельные (соединяющие одни и те же вершины) ребра, в противном случае граф называется *простым*.

(B) (a) (b) (a)

а). б).

Рис. 7.1. Граф неориентированный (а) и ориентированный (б).

Путь в графе – последовательность узлов, связанных ребрами. *Элементарным* называется путь, в котором все ребра различны, *простым* называется путь, в котором все вершины различны. Путь, начинающийся и заканчивающийся в одной и той же вершине, называется циклом, а граф, содержащий такие пути – *циклическим*. Граф, в котором отсутствуют циклы, называется *ациклическим*. Два узла графа являются *смежными*, если существует путь от одного из

них до другого. Узел называется *инцидентным* к ребру, если он является его вершиной, т.е. ребро направлено к узлу.

С помощью графа можно наглядно представить разветвляющиеся связи, которые и привели к общеупотребительному термину «дерево». *Деревом* называется орграф, для которого существует узел, в которой не входит ни одной дуги – корень и в каждую вершину, кроме корня, входит одна дуга. *Степенью узла* в дереве называется количество дуг, которое из него выходит. Степень дерева равна максимальной степени узла, входящего в дерево. Существует несколько способов графического изображения деревьев (рис. 7.1). Первый способ состоит в использовании для изображения поддеревьев известного метода диаграмм Венна, второй – метода вкладывающихся друг в друга скобок, третий способ, применяемый при составлении оглавлений книг. При применении последнего способа каждой вершине приписывается числовой номер, который должен быть меньше номеров, приписанных корневым вершинам присоединенных к ней поддеревьев. Все эти представления демонстрируют одну и ту же структуру и поэтому эквивалентны.

V0 v1 v2 v5 v6 v3 v4 v7 v8 v9 v10 (v0) (v1) (v7) (v8) (v9) (v10) (v3) (v2) (v4) (v5) (v6)

а). б).

V0

V1

V2

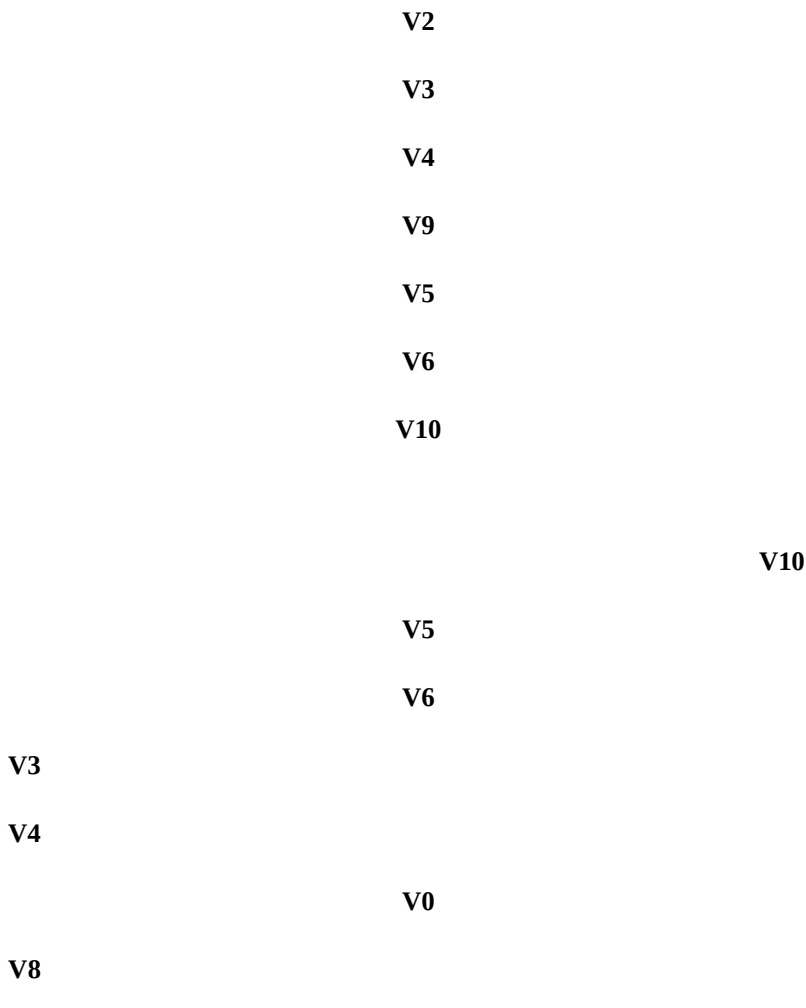
V7

V9

V8

V1

V7



в). г).

Рис. 7.2. Представление дерева: а) исходное дерево,

б) оглавление книг, в) граф, г) диаграмма Венна.

4.7.2. Машинное представление графов

Существуют два основных метода представления графов в памяти: матричный (массивами), и связными нелинейными списками. Выбор метода зависит от природы данных и операций, выполняемых над ними. Если задача требует большого числа включений и исключений узлов, то целесообразно представлять граф связными списками. В противном случае можно применить и матричное представление.

При представлении в виде списочной структуры, используется два типа списков – список вершин и список ребер. Элемент списка вершин содержит поле данных и два указателя. Один указатель связывает данный элемент с элементом другой вершины. Второй указатель связывает элемент списка вершин со списком ребер, связанных с данной вершиной.

Для ориентированного графа используют список дуг, исходящих из вершины (рис. 7.3). Элемент списка дуг состоит только из двух указателей. Первый указатель используется для того, чтобы показать в какую вершину дуга входит, а второй – для связи элементов в списке дуг вершины.

a).

(c,d)

(b,d)

(a,d)

(a,c)

(a,b)

A

B

C

D

список дуг, связанных с вершиной a

элементы списка дуг

б).

Рис. 7.3. Представление графа (а) в виде списочной структуры (б).

Распространенным способом представления графов является матричный способ (рис. 7.4). Для ненаправленных графов обычно используют *матрицы смежности*, а для ориентированных графов – *матрицы инцидентности*. Обе матрицы имеют размерность n^2, где n – число вершин в графе.

При использовании матриц смежности их элементы представляются в памяти элементами массива. Элемент матрицы имеет ноль в позиции m(i,j), если не существует ребра, связывающего вершину i с вершиной j, или единичное значение в позиции m(i,j), если такое ребро существует:

$$m(i,j) = \begin{cases} 1, \text{если узел } i \text{ смежен с } j \ (\text{есть путь} < i, j >) \\ 0 \text{ иначе} \end{cases}$$

Матрицы смежности применяются, когда в графе много связей и матрица хорошо заполнена. Для простого графа матрица состоит из нулей и единиц, для мультиграфа – из нулей и целых чисел, указывающих кратность соответствующих ребер, для взвешенного графа – из нулей и вещественных чисел, задающих вес каждого ребра.

Матрица смежности симметрична относительно главной диагонали, поэтому можно хранить и обрабатывать только часть матрицы. Для хранения одного элемента матрицы достаточно выделить один бит. Правила построения матрицы инцидентности аналогичны правилам построения матрицы смежности. Разница состоит в том, что единица в позиции m(i,j) означает выход дуги из вершины i и вход дуги в вершину j. В некоторых матричных алгоритмах обработки графов используются матрицы путей. Под путем длиной k из вершины i в вершину j будем понимать возможность попасть из вершины i в вершину j за k переходов, каждому из которых соответствует одна дуга. Одна матрица путей m_k содержит сведения о наличии всех путей одной длины k в графе. Единичное значение в позиции (i,j) означает наличие пути длины k из вершины i в вершину j:

$$m_k(i,j) = \begin{cases} 1, \textit{если } k_{i,j} \\ 0 \textit{ иначе} \end{cases}$$

Например, при k=2 имеем (<A,B>,<B,C>).

Примеры построения матриц смежности и инцидентности показаны на рис. 7.14. В первой матрице не отражены направления, поэтому a(i,j) = a(j,i), т.е. матрица симметрична относительно главной диагонали.

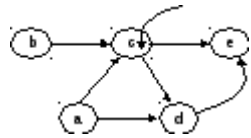


Рис. 7.4. Граф и его матричное представление.

Матрица смежности.

	a	**b**	**c**	**d**	**e**
a	0	0	1	1	0
b	0	0	1	0	0
c	1	1	0	1	1
d	1	0	1	0	1
e	0	0	1	1	0

Матрица инцидентности.

	a	**b**	**c**	**d**	**e**
a	0	0	1	1	0
b	0	0	1	0	0
c	0	0	0	1	1
d	0	0	0	0	1
e	0	0	0	1	0

Матрица m_1 полностью совпадает с матрицей инцидентности. По матрице m_1 можно построить m_2 , а по матрице m_2 можно построить m_3 и т.д. Если граф является ациклическим, то число матриц путей ограничено. В противном случае матрицы будут повторяться до бесконечности с некоторым периодом, связанным с длиной циклов. Матрицы путей не имеет смысла вычислять до бесконечности. Достаточно остановиться в случае повторения матриц.

Матрица путей m1.

	a	**b**	**c**	**d**	**e**
a	0	0	1	1	0
b	0	0	1	0	0
c	0	0	0	1	1
d	0	0	0	0	1
e	0	0	0	1	0

Матрица путей m2.

	a	**b**	**c**	**d**	**e**
a	0	0	0	1	1
b	0	0	0	1	1
c	0	0	0	1	1
d	0	0	0	1	0
e	0	0	0	0	1

Матрица путей m3.

	a	**b**	**c**	**d**	**e**
a	0	0	0	1	1
b	0	0	0	1	1
c	0	0	0	1	1
d	0	0	0	0	1

e	0	0	0	1	0

Если выполнить логическое сложение всех матриц путей, то получим транзитивное замыкание графа (рис. 7.5). В результате матрица будет содержать все возможные пути в графе.

$$M_{TR}^{3} = m_1 \ or \ m_2 \ or \ m_3$$

Рис. 7.5. Транзитивное замыкание в графе.

Матрица путей M_{TR}.

	a	**b**	**c**	**d**	**e**
a	0	0	1	1	1
b	0	0	1	1	1
c	0	0	0	1	1
d	0	0	0	1	1
e	0	0	0	1	1

Наличие циклов в графе можно определить с помощью алгоритма, который может быть реализован для матричного и для спискового способа представления графа (рис. 7.6). Принцип выделения циклов следующий. Если вершина имеет только входные или только выходные дуги, то она явно не входит ни в один цикл (обнуляемые строки на первой итерации отмечены курсивом). Можно удалить все такие вершины из графа вместе со связанными с ними дугами.

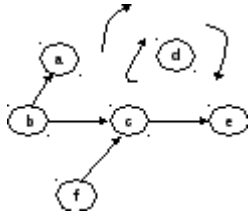


Рис. 7.6. Определение циклов в графе.

В результате появятся новые вершины, имеющие только входные или выходные дуги. Они также удаляются. Итерации повторяются до тех пор, пока граф не перестанет изменяться. Отсутствие изменений свидетельствует об отсутствии циклов, если все вершины были удалены. Все оставшиеся вершины будут обязательно принадлежать циклам.

Сформулируем алгоритм в матричном виде. Для i от 1 до n выполнить шаги 1-2:

1. если строка M(*i* ,*)* = 0, обнулить столбец *i*;
2. если столбец M(*, *i*) = 0, обнулить строку *i*;
3. если матрица изменилась, выполнить шаг 1;
4. если матрица нулевая завершить процесс, граф ациклический, иначе матрица содержит вершины, входящие в циклы.

Поиск циклов в графе.

начало	a b c d e f
итерация 1	b c d e f
итерация 2	c d e f
итерация 3	c d e
итерация 4	d e
итерация 5	d e

Матрица итераций.

	a	**b**	**c**	**d**	**e**	**f**
a	0	0	0	0	0	0
b	0→1	0	0→1	0	0	0→1
	i=1		i=2			i=2
c	0	0	0	0→1	0→1	0
				i=3	i=3	
d	0	0	0	0	1	0
e	0	0	0	1	0	0
f	0	0	0	0	0	0

Достоинством алгоритма является то, что одновременно с определением цикличности или ацикличности графа формируется список вершин, входящих в циклы. В матричной реализации после завершения алгоритма остается матрица инцидентности, соответствующая подграфу, содержащему все циклы исходного графа.

4.7.3. Бинарные деревья

С точки зрения представления в памяти различают два типа деревьев: бинарные и сильноветвящиеся. В бинарном дереве из каждой вершины выходит не более двух дуг (рис. 7.7). В сильноветвящемся дереве количество дуг может быть произвольным. Вершины в дереве, имеющие степень ноль, называются листьями.

p7

p8

p5

p4

p6

p2

p1 корень

p3

p1,p2,p6 – узлы

p3,p4,p5,p7,p8 - листья

Рис. 7.7. Бинарное дерево.

Другим признаком структурной классификации бинарных деревьев является полнота и строгость бинарного дерева. Полное бинарное дерево на всех уровнях, меньше n имеют степень узла 2, а на уровне n степень равно 0 (рис. 7.8). Неполное бинарное дерево на уровнях, меньше n может иметь степень узла меньше 2. Строго бинарное дерево состоит только из узлов, имеющих степень два или ноль (рис. 7.9). Не строго бинарное дерево содержит узлы со степенью равной одному.

а). б).

Рис. 7.8. Неполное (а) и полное (б) бинарные деревья.

а). б).

Рис. 7.9. Строго (а) и не строго (б) бинарные деревья.

4.7.3.1. Представление бинарных деревьев

Бинарные деревья могут быть представлены в виде списков или массивов. Списочное представление основано на элементах, соответствующих узлам дерева (рис. 7.10). Каждый элемент имеет поле данных и два поля указателей. Один указатель используется для связывания элемента с правым потомком, а другой – с левым. Листья имеют пустые указатели потомков.

При списковом представлении обязательно следует сохранять указатель на узел, являющийся корнем дерева. Можно заметить, что такой способ представления имеет сходство с простыми линейными списками. Сходство не случайно, т.к. дерево является разновидностью мультисписка, образованного комбинацией множества линейных списков. Каждый линейный список объединяет узлы, входящие в путь от корня дерева к одному из листьев.

В виде массива проще всего представляется полное бинарное дерево, так как оно всегда имеет строго определенное число вершин на каждом уровне (рис. 7.11). Вершины можно пронумеровать слева направо последовательно по уровням и использовать номера в качестве индексов в одномерном массиве. Если число уровней дерева в процессе обработки не будет существенно изменяться, то такой способ представления будет значительно более экономичным, чем любая списковая структура.

Однако далеко не все бинарные деревья являются полными. Для неполных бинарных деревьев применяют следующий способ представления. Бинарное дерево дополняется до полного дерева, вершины последовательно нумеруются. В массив заносятся только те вершины, которые были в исходном неполном дереве. При таком представлении элемент массива выделяется независимо от того, будет ли он содержать узел исходного дерева. Следовательно, необходимо

отметить неиспользуемые элементы массива, например, занесением специального значения в соответствующие элементы массива. В результате структура дерева переносится в одномерный массив.

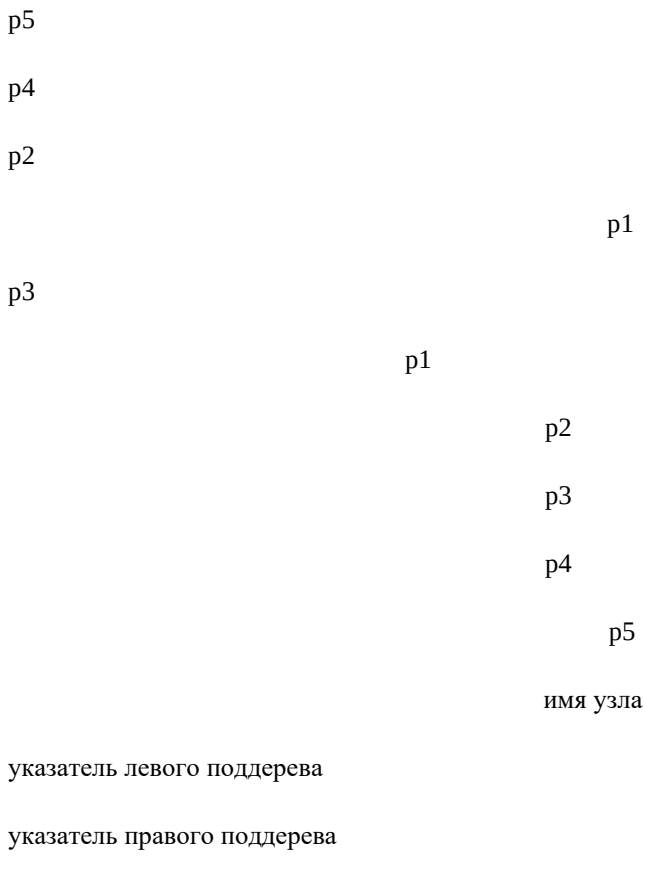


Рис. 7.10. Представление бинарного дерева в виде списковой структуры.

1

2

3

4

5

6

7

адрес последней вершины полного дерева 2^{n-1}

p5

p4

p2

p1

p3

p7

p1

p2

p3

p4

p5

p7

$A = 2^{k-1}+i-1$

Рис. 7.11. Представление бинарного дерева в виде массива.

Адрес любой вершины в массиве:

$A = 2^{k-1}+i-1$,

где k-номер уровня вершины, i - номер на уровне k в полном бинарном дереве.

Адрес корня равен единице. Для любой вершины адреса левого и правого потомков равны:

$A_L = 2^k+2^{i-1}$

$A_R = 2^k+2^{i-1}+1$

Недостатком такого способа представления бинарного дерева является то, что структура данных является статической. Размер массива выбирается исходя из максимально возможного количества уровней бинарного дерева. Причем, чем менее полным является дерево, тем менее рационально используется память.

4.7.3.2. Прохождение бинарных деревьев

В алгоритмах обработки деревьев используется операция прохождения дерева. Под прохождением дерева понимают определенный порядок обхода всех вершин дерева. Различают несколько методов прохождения. Прямой порядок прохождения бинарного дерева можно определить следующим образом (рис. 7.12):

- попасть в корень;
- пройти в прямом порядке левое поддерево;
- пройти в прямом порядке правое поддерево.

a b c d e f g h i

Рис. 7.12. Прямой порядок прохождения бинарного дерева.

Прохождение бинарного дерева в обратном порядке можно определить аналогично (рис. 7.13):

- пройти в обратном порядке левое поддерево;
- пройти в обратном порядке правое поддерево;
- попасть в корень.

Определим еще один порядок прохождения бинарного дерева, называемый симметричным:

- пройти в симметричном порядке левое поддерево;
- попасть в корень;
- пройти в симметричном порядке правое поддерево.

Порядок обхода бинарного дерева можно хранить непосредственно в структуре данных. Для этого достаточно ввести дополнительное поле указателя в элементе списковой структуры и хранить в нем указатель на вершину, следующую за данной вершиной при обходе дерева.

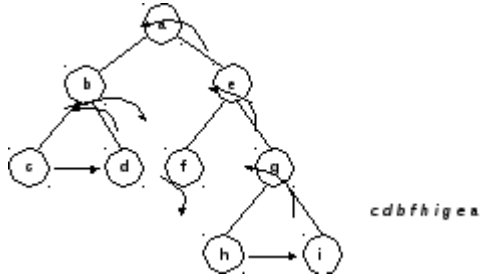
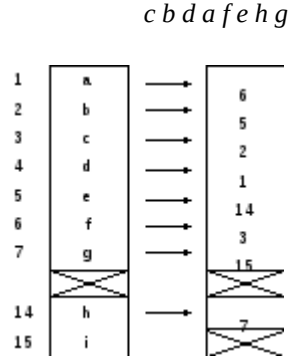


Рис. 7.13. Обратный порядок прохождения бинарного дерева.

Представление деревьев в виде массивов также допускает хранение порядка прохождения дерева. Для этого вводится дополнительный массив, в который записываются адрес вершины в основном массиве, следующей за данной вершиной. Такие структуры данных получили название прошитых бинарных деревьев. Указатели или адреса, определяющие порядок обхода называют нитями. При этом в соответствии с порядком прохождения вершин различают право прошитые, лево прошитые и симметрично прошитые бинарные деревья (рис. 7.14).



а). б).

Рис. 7.14. Представление симметрично прошитого бинарного дерева (а) в виде массивов (б).

4.7.4. Алгоритмы на деревьях

4.7.4.1. Сортировка с прохождением бинарного дерева

В качестве примера использования прохождения бинарного дерева приведем один из способов сортировки. Допустим, имеется некоторый массив и требуется упорядочить его элементы по возрастанию. Сортировка при этом распадается на две фазы: построение дерева и прохождение дерева.

Дерево строится по следующим принципам (рис. 7.15). В качестве корня создается узел, в который записывается первый элемент массива. Для каждого очередного элемента создается новый лист. Если элемент меньше значения в текущем узле, то для него выбирается левое поддерево, если больше или равен – правое. Для создания очередного узла происходят сравнения элемента со значениями существующих узлов, начиная с корня.

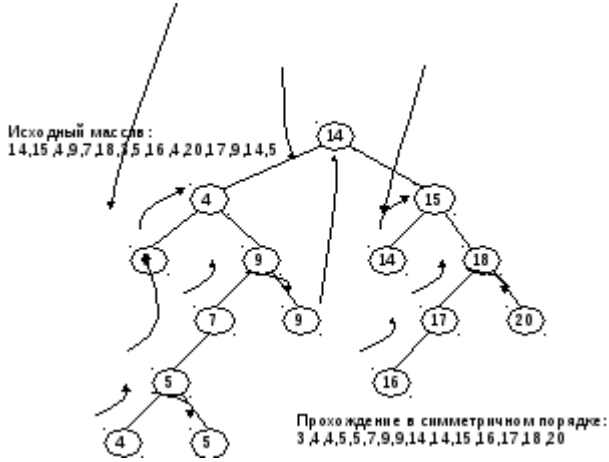


Рис. 7.15. Сортировка по возрастанию с прохождением бинарного дерева.

4.7.4.2. Сортировка методом турнира с выбыванием

Приведем другой алгоритм сортировки, основанный на использовании бинарных деревьев. Данный метод получил название турнира с выбыванием. Пусть имеется исходный массив 10, 20, 3, 1, 5, 0, 4, 8. Сортировка начинается с создания листьев дерева. В качестве листьев бинарного дерева создаются узлы, в которых записаны значения элементов исходного массива. Дерево строится от листьев к корню. Для двух соседних узлов строится общий предок, до тех пор, пока не будет создан корень. В узел-предок заносится значение, являющееся наименьшим из значений в узлах-потомках.

В результате построения такого дерева наименьший элемент попадает сразу в корень. Далее начинается извлечение элементов из дерева. Извлекается значение из корня. Данное значение является первым элементом в результирующем массиве. Извлеченное значение помещается в отсортированный массив и заменяется в дереве на специальный символ.

После этого происходит повторное занесение значений в родительские элементы от листьев к корню. При сравнениях специальный символ считается большим по отношению к любому другому значению. После повторного заполнения из корня извлекается очередной элемент и итерация повторяется. Извлечения элементов продолжаются до тех пор, пока в дереве не останутся одни специальные символы. В результате получим отсортированный массив 0, 1, 3, 4, 5, 8, 10, 20. Описанный алгоритм иллюстрируют рис. 7.16-7.19.

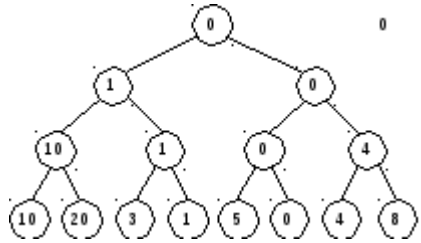


Рис. 7.16. Построение дерева сортировки.

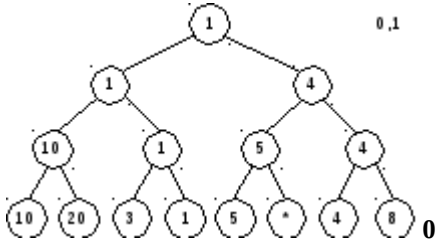


Рис. 7.17. Замена извлекаемого элемента на специальный символ.

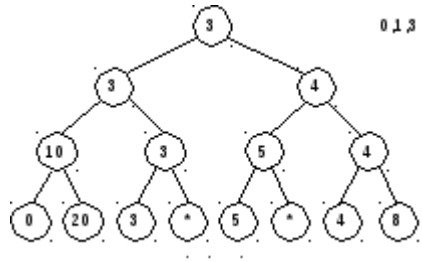
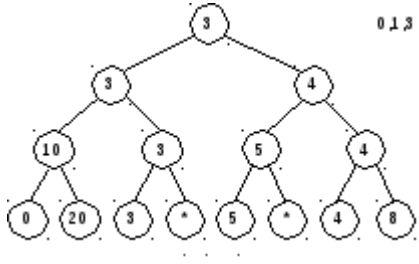


Рис. 7.18. Повторное заполнение дерева сортировки.



0,1,3,4,5,8,10,**20**

Рис. 7.19. Извлечения элементов из дерева сортировки.

4.7.4.3. Применение бинарных деревьев для сжатия информации

Бинарные деревья применяются в задачах сжатия информации. Рассмотрим пример. Имеется текстовая строка S, состоящая из 10 символов S = ABCCCDDDDD. При кодировании одного символа одним байтом для строки потребуется 10 байт. Попробуем сократить требуемую память. Рассмотрим, какие символы действительно требуется кодировать. В данной строке используются всего 4 символа. Поэтому можно использовать укороченный код.

A 00

B 01

C 10

D 11

b = 00011010101111111111 (20 бит)

В данном случае мы проанализировали текст на предмет использования символов. Можно заметить, что различные символы имеют различную частоту повторения. Существующие методы кодирования позволяют использовать этот факт для уменьшения длины кода. Одним из таких методов является кодирование Хаффмана. Он основан на использовании кодов различной длины для различных символов. Для максимально повторяющихся символов используют коды минимальной длины.

Построение кодовой таблицы происходит с использованием бинарного дерева (рис. 7.20). В корне дерева помещаются все символы и их суммарная частота повторения. Далее выбирается наиболее часто используемый символ и помещается со своей частотой повторения в левое поддерево. В правое поддерево помещаются оставшиеся символы с их суммарной частотой. Затем описанная операция проводится для всех вершин дерева, которые содержат более одного символа.

Само дерево может быть использовано в качестве кодовой таблицы для кодирования и декодирования текста. Кодирование осуществляется следующим образом. Для очередного символа в качестве кода используется путь от листа соответствующего символа к корню дерева. Причем каждому левому поддереву приписывается ноль, а каждому правому – единица.

Рис. 7.20. Построение кодовой таблицы.

Для строки S будет получен следующий код b=11011110101000000. Длина кода составляет 17 бит, что меньше по сравнению с укороченным кодом. Алгоритм распаковки можно сформулировать следующим образом:

1. i:=0, j:=0;

2. если i > n, то стоп строка распакована, иначе i:=i+1;

3. node:= root;

4. если b(i) = 0, то node:=left(node), иначе node:=right(node)

5. если left(node) = 0 и right(node) = 0, то j:=j+1, s(j):= str(node), перейти к шагу 2, иначе i:=i+1, перейти к шагу 4

В алгоритме корень дерева обозначен как root, a left(node) и right(node) обозначают левый и правый потомки узла node.

На практике такие способы упаковки используются не только для текстов, но и для произвольных двоичных данных. Любой файл можно рассматривать как последовательность байт. Тогда дерево кодирования можно построить не для символов, а для значений байт, встречающихся в кодируемом файле (рис. 7.21). Поскольку байт может принимать 256 значений, то соответствующее дерево будет иметь не более 256 листьев.

j

i

строка S

код строки b

110111

A B C

Рис. 7.21. Процесс распаковки кода.

В узлах дерева после его полного построения нет необходимости хранить несколько значений кодов и частоты повторения. Для кодирования и декодирования достаточно хранить только одно значение кода и только для листового узла. Поэтому такой способ представления кодовой таблицы является достаточно компактным. Схемы кодирования подобного типа используются в программах архивации данных и сжатия растровых изображений в форматах графических файлов.

4.7.4.4. Представление выражений с помощью деревьев

С помощью деревьев можно представлять произвольные арифметические выражения (рис. 7.22-7.23). Каждому листу в таком дереве соответствует операнд, а каждому родительскому узлу – операция. В общем случае дерево при этом может оказаться не бинарным. Однако если число операндов любой операции будет меньше или равно двум, то дерево будет бинарным. Причем если все операции будут иметь два операнда, то дерево окажется строго бинарным.

f(a,b,c,d,e))

-(A+B)((C+cos(D+E)*

Рис. 7.22. Представление арифметического выражения произвольного вида в виде дерева.

f(a+b,sin c)

Рис. 7.23. Представление арифметического выражения в виде бинарного дерева.

Бинарные деревья могут быть использованы не только для представления выражений, но и для их вычисления (рис. 7.24). В листьях записываются значения операндов. Затем от листьев к корню производится выполнение операций. В процессе выполнения в узел операции записывается результат ее выполнения. В конце вычислений в корень будет записано значение, которое и будет являться результатом вычисления выражения.

(1+10)*5

Рис. 7.24. Вычисление арифметического выражения с помощью бинарного дерева.

Помимо арифметических выражений с помощью деревьев можно представлять выражения других типов, например, логические выражения (рис. 7.25). Поскольку функции алгебры логики определены над двумя или одним операндом, то дерево для представления логического выражения будет бинарным.

((aVb)&(cVd))&(e&fVa&b)

Рис. 7.25. Представление логического выражения в виде бинарного дерева.

4.7.5. Представление сильноветвящихся деревьев

Произвольные деревья могут быть сильноветвящимися. Причем число потомков различных узлов не ограничено и заранее не известно. Тем не менее, для представления таких деревьев достаточно иметь элементы, аналогичные элементам списковой структуры бинарного дерева.

Элемент такой структуры содержит минимум три поля: значение узла, указатель на начало списка потомков узла, указатель на следующий элемент в списке потомков текущего уровня. Также как и для бинарного дерева необходимо хранить указатель на корень дерева.

При этом дерево представлено в виде структуры, связывающей списки потомков различных вершин. Такой способ представления вполне

пригоден и для бинарных деревьев. Представление деревьев с произвольной структурой в виде массивов может быть основано на матричных способах представления графов.

Рассмотрим использование сильноветвящихся деревьев для представления иерархической структуры каталогов файловой системы. Во многих файловых системах структура каталогов и файлов, как правило, представляет одно или несколько сильноветвящихся деревьев. В файловой системе корень дерева соответствует логическому диску. Листья дерева соответствуют файлам и пустым каталогам, а узлы с ненулевой степенью – непустым каталогам.

Для представления такой структуры используем расширение спискового представления сильноветвящихся деревьев (рис. 7.26). Способы представления деревьев, рассмотренные ранее, являются предельно экономичными, но не очень удобными для перемещения по дереву в разных направлениях. Именно такая задача встает при просмотре структуры каталогов. Необходимо осуществлять навигацию – перемещаться из текущего каталога в каталог верхнего или нижнего уровня, или от файла к файлу в пределах одного каталога.

Списки потомков можно сделать двунаправленными. Для этого необходимо ввести указатель на предыдущий узел last, а для упрощения перемещения по дереву от листьев к корню потребуется указатель на предок текущего узла up. Общими с традиционными способами представления являются указатели на список потомков узла down и следующий узел next.

↑a

↑f

↑c

c

↑a

nil

↑d

d

↑a

nil

↑e

e

↑a

nil

nil

f

↑b

nil

↑g

g

↑b

nil

↑h

h

↑b

nil

nil

элемент

список потомков

следующий

элемент дерева

….

Рис. 7.26. Представление сильноветвящихся деревьев в виде списков.

4.8. Методы ускорения доступа к данным

4.8.1. Хеширование данных

Ранее были рассмотрены методы доступа к данным, используя алгоритмы поиска. Для ускорения доступа к данным в таблицах можно использовать предварительное упорядочивание таблицы в

соответствии со значениями ключей (рис. 8.1). При этом могут быть использованы методы поиска в упорядоченных структурах данных, например, двоичный поиск, что существенно сокращает время поиска данных по значению ключа. Однако при добавлении новой записи требуется переупорядочить таблицу. Потери времени на повторное упорядочивание таблицы могут значительно превышать выигрыш от сокращения времени поиска.

Тем не менее, метод двоичного поиска относится к алгоритмам класса O(log(n)). Например, для установления факта наличия или отсутствия заданного элемента в наборе из 1000 элементов требуется около 10 сравнений, поскольку 2^{10}=1024. Двоичный поиск является наиболее эффективным из всех возможных методов, которые требуют использования функции сравнения. Все они используют ключ элемента для перемещения по структуре данных с применением метода, в основе которого лежит сравнение.

Другие подходы к поиску исключает саму процедуру сравнения. Каждый элемент связывается с уникальным индексом, который позволяет обнаружить элемент путем однонаправленного действия, просто извлекая элемент, расположенный в определенной позиции. Преобразование ключа элемента в значение индекса называется *хешированием* и выполняется с помощью *функции хеширования*. Массив, используемый для хранения элементов, с которыми используются значения индексов, называют хеш-таблицей.

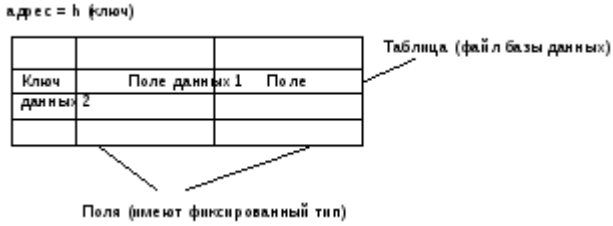
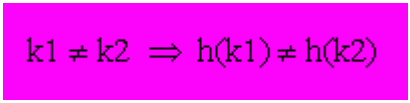


Рис. 8.1. Хеш-таблица.

Выполнение поиска с использованием хеширования требует реализации двух отдельных алгоритмов. Первый шаг состоит в хешировании, в результате которого ключ элемента преобразуется в значение индекса. Идеальной хеш-функцией является такая функция h,

которая для любых двух неодинаковых ключей дает неодинаковые адреса:



$$k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$$

Подобрать такую функцию можно, если все возможные значения ключей заранее известны. Такая организация данных называется *совершенным хешированием*. При заранее неопределенном множестве значений ключей и ограниченной длины таблицы подбор совершенной функции затруднителен. На практике используют хеш-функции, которые не гарантируют выполнение условия. Отображение двух или более ключей на один и тот же индекс называют *конфликтом* или *коллизией*. Поэтому требуется второй шаг, определяющий способ разрешения коллизий в случае их возникновения.

Хеш-таблица представляет хороший пример достижения компромисса между быстродействием и занимаемым объемом памяти. При уникальном значении ключей элемента типа word требуется создать 65536 элементов, и при этом можно гарантировать нахождение элемента с заданным значением ключа в результате одной операции. Однако при хранении не более 100 элементов такой подход окажется расточительным, т.к. 99.85% памяти массива не будет использоваться. С другой стороны, можно выделить размер памяти под массив для фактического числа элементов, выполнить сортировку и двоичный поиск. Память будет сэкономлена, но работать такой алгоритм будет значительно медленнее.

Алгоритмы хеширования предлагают компромисс, при котором хеш-таблицы будут занимать больше места, чем требуется для хранения всех элементов, но поиск будет выполняться всего за несколько операций доступа.

С хеш-таблицами выполняют следующие операции: проверка наличия элемента в таблице и удаление элемента из таблицы. Также часто необходимо расширение таблицы для помещения в нее большего числа элементов, чем предполагалось изначально. Заметим, что функционирование хеш-таблицы не предполагает извлечение записей в порядке следования, т.к. физически соседние ключи таблицы

указывают на логически далеко расположенные друг от друга элементы.

4.8.1.1. Функции хеширования

Рассмотрим примеры реализации хеш-функций. Функция должна принимать ключ элемента и преобразовывать его в значение индекса. Если в таблице предусмотрено место для n элементов, то функция хеширования должна генерировать значения индексов, лежащих в диапазоне 0..n-1. Кроме того, для различных типов ключей должны быть использованы различные функции. В идеале функция хеширования должна создавать значение индексов, которые никак не связаны с ключами. В определенном смысле хеш-функция должны быть подобна функции рандомизации, т.е. очень похожие ключи должны приводить к созданию совершенно различных хеш-значений.

Простейший случай – использование целочисленных ключей, когда элемент уникально идентифицируется целочисленным значением. Самой простой функцией будет операция деления по модулю. Если хеш-таблица содержит n элементов, хеш-значение ключа k равно:

$h(k) = k \bmod n.$

Для случае равномерного распределения значений ключей такая функция вполне подходила бы, но в общем случае множество не столь равномерно распределенное, и поэтому в качестве размера таблицы необходимо использовать простое число.

Для строковых ключей следует использовать метод преобразования строки в целочисленное значение. Один из таких способов заключается в расчете суммы всех ASCII-значений кодов символов строки. Однако ключи могут быть анаграммами друг друга, и применение такой схемы приводило бы к конфликтам. Для исключения влияния анаграмм при их использовании в качестве ключей применяют весовые коэффициенты, соответствующие позиции каждого символа в строке.

Пример реализации простой функции хеширования строковых ключей приведен ниже.

```
function SimpleHash(aKey: string; aTableSize: Integer): Integer;
```

```
var

i: Integer;

Hash: LongInt;

begin

Hash:=0;

for i:=1 to Length(aKey) do

Hash:=((Hash*17)+ord(aKey[i])) mod aTableSize;

Result:=Hash;

if Result < 0 then

Inc(Result, aTableSize);

end;
```

Функция принимает в качестве параметров значение строкового ключа и размер таблицы. Алгоритм поддерживает постоянно изменяющееся хеш-значение, изначально установленное равным нулю. Это значение изменяется для каждого символа в строке путем его умножения на небольшое простое число, добавления кода следующего символа и деления по модулю на размер таблицы. Если конечное значение окажется отрицательным (особенность операции деления по модулю в Delphi), к результату добавится значение размера таблицы.

Другая известная функция, именуемая ELF-хешем (формат исполняемых и компонуемых модуле, Executable and Linking Foramt), была предложена П.Дж.Вайнбергером. Отличие алгоритма от приведенного выше заключается в применении эффекта рандомизации, когда операция XOR вновь загружает старший полубайт действующей рабочей переменной хеша (полубайт, который должен исчезнуть в результате переполнения при выполнении следующей операции умножения), если он не равен нулю, в младшую часть переменной.

Затем старший полубайт устанавливается в ноль, в результате чего хеш-значение никогда не будет неотрицательным.

```
function SimpleHash(aKey: string; aTableSize: Integer): Integer;

var

i: Integer;

G,Hash: LongInt;

begin

Hash:=0;

for i:=1 to Length(aKey) do

Hash:=(Hash shl 4)+ord(aKey[i]);

G:=Hash and LongInt($F0000000);

if G <> 0 then

Hash:=(Hash xor (G shr 24)) xor G;

Result:=Hash mod aTableSize;

end;
```

Функция превосходит предыдущую реализацию благодаря эффекту рандомизации и выполнении для каждого символа только операций поразрядного сдвига и логических операций. В общем случае ее можно считать наилучшей.

4.8.1.2. Оценка качества хеш-функции

Правильный выбор хеш-функции важен. При ее удачном построении таблица заполняется более равномерно, уменьшается число коллизий и уменьшается время выполнения операций поиска, вставки и удаления. Для оценки качества хеш-функции проводят имитационное моделирование. Формируется целочисленный массив, длина которого совпадает с размером хеш-таблицы. Случайно генерируется достаточно большое число ключей, для каждого ключа вычисляется хеш-функция. В элементах массива просчитывается число генераций данного адреса. По результатам моделирования можно построить график распределения значений хеш-функции (рис. 8.2). Для получения корректных оценок число генерируемых ключей должно в несколько раз превышать длину таблицы.

Рис. 8.2. Распределение коллизий в адресном пространстве таблицы.

Если число элементов таблицы достаточно велико, то график строится не для отдельных адресов, а для групп адресов. Например, все адресное пространство разбивается на 100 фрагментов и подсчитывается число попаданий адреса для каждого фрагмента. Большие неравномерности свидетельствуют о высокой вероятности коллизий в отдельных местах таблицы. Такая оценка является приближенной, но позволяет предварительно оценить качество хеш-функции и избежать грубых ошибок при ее построении.

Оценка будет более точной, если генерируемые ключи будут более близки к реальным ключам, используемым при заполнении хеш-таблицы. Для символьных ключей важно добиться соответствия генерируемых кодов символов тем кодам символов, которые имеются в реальном ключе. Для этого стоит проанализировать, какие символы могут быть использованы в ключе.

Например, если ключ представляет фамилию на русском языке, то будут использованы русские буквы. Причем первый символ может быть прописным, а остальные – строчными. Если ключ представляет номерной знак автомобиля, то также несложно определить допустимые коды символов в определенных позициях ключа. Приведем пример генерации ключа из десяти латинских букв, первая из которых является прописной, а остальные строчными:

```
var

i: Integer;

s: string[10];

begin

s[1]:=chr(random(90-65)+65);

for i:=2 to 10 do

s[i]:=chr(random(122-97)+97);

end;
```

В примере допустимые коды символов располагаются последовательными непрерывными участками в кодовой таблице. Рассмотрим общий случай. Допустим, необходимо сгенерировать ключ из m символов с кодами в диапазоне от n1 до n2 (диапазон непрерывный):

```
for i:=1 to m do
```

```
str[i]:=chr(random(n2-n1)+n1);
```

На практике возможны варианты, когда символы в одних позициях ключа могут принадлежать к разным диапазонам кодов, причем между этими диапазонами может существовать разрыв. Пример генерации ключа из m символов с кодами в диапазоне от n1 до n4 с разрывом от n2 до n3 (рис. 8.3) приведен ниже.

Рис. 8.3. Диапазон кодов ключа.

```
for i:=1 to m do

begin

x:=random((n4-n3)+(n2-n1));

if x <= (n2-n1) then

str[i]:=chr(x+n1) else

str[i]:=chr(x+n1+n3-n2);

end;
```

4.8.1.3. Методы разрешения коллизий

Подобрать совершенную хеш-функцию для заранее неизвестной последовательности крайне трудно, поэтому приходиться мириться с возможными конфликтами. Для разрешения конфликтов используют различные методы, которые разделяют на группу методов с *закрытой адресацией* и с *открытой адресацией*. К первой группе относят методы разрешения конфликтов посредством связывания, ко второй – методы с линейным, квадратичным, псевдослучайным зондированием и двойное хеширование.

Разрешение коллизий посредством открытой адресации. В методах данной группы применяются алгоритмы, обеспечивающие перебор элементов таблицы в поисках свободного места для новой записи.

Линейное зондирование. Метод состоит в последовательном переборе элементов таблицы с некоторым фиксированным шагом (рис. 8.4):

$a=h(key)+c*i$

где i – номер попытки разрешить коллизию. Константа задает длину шага. При единичном шаге происходит последовательный перебор всех элементов после текущего.

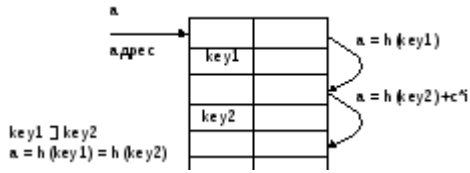


Рис. 8.4. Линейное зондирование.

С линейным зондированием связана проблема кластеризации. Элементы имеют тенденцию к образованию непрерывных групп, или *кластеров,* – занятых ячеек. Добавление новых элементов приводит к увеличению размера кластеров. Кластеры влияют на среднее число зондирований, требуемых для обнаружения существующего элемента (*попадания*) и выяснения отсутствия элемента (*промаха*). Д.Кнут показал, что среднее число зондирования для обнаружения попадания равно:

$\frac{1}{2}(1+1/(1-F))$

Где F– коэффициент загрузки, т.е. количество элементов в таблице, деленное на размер таблицы.

Среднее число зондирований для обнаружения промаха:

$\frac{1}{2}(1+1/(1-F)^2)$

Например, если таблица заполнена примерно наполовину, то для обнаружения попадания потребуется приблизительно 1.5 зондирования, а для обнаружения промаха – 2.5 зондирования. При загрузке 90% эти значения будут уже 5.5 и 55.5 зондирований соответственно. Следовательно, чтобы эффективность оставалась приемлемой для линейного зондирования, загрузка таблицы должна быть на уровне 60%.

Рассмотрим алгоритмы вставки, поиска и удаления элемента при линейном зондировании.

Вставка элемента:

1. i=0

2. $a=(h(key)+c*i) \bmod i$

3. Если $t(a)=свободно$, то $e(a)=key$, записать элемент, элемент добавлен иначе $i=i+1$, перейти к шагу 2.

Поиск элемента:

1. i=0

2. $a=(h(key)+c*i) \bmod i$

3. Если $t(a)=key$, то элемент найден. Если $t(a)=свободно$, то элемент не найден $i=i+1$, перейти к шагу 2.

Процедура удаления несколько сложнее. Каждый элемент таблицы может находиться в двух состояниях: свободно и занято. Если удалить элемент, переведя его в состояние свободно, то после такого удаления алгоритм поиска будет работать некорректно. Предположим, ключ удаляемого элемента имеет в таблице ключи синонимы. В том случае, если за удаляемым элементом в результате разрешения коллизий были размещены элементы с другими ключами, то поиск этих элементов после удаления всегда будет давать отрицательный результат, так как алгоритм поиска останавливается на первом элементе, находящемся в состоянии свободно.

Скорректировать ситуацию можно различными способами. Самый простой из них заключается в том, чтобы производить поиск элемента не до первого свободного места, а до конца таблицы. Однако такая модификация алгоритма сведет на нет весь выигрыш в ускорении доступа к данным, который достигается в результате хеширования.

Другой способ сводится к тому, чтобы проследить адреса всех ключей-синонимов для ключа удаляемого элемента и при необходимости переразместить соответствующие записи в таблице. Скорость поиска после такой операции не уменьшится, но затраты времени на само переразмещение элементов могут оказаться значительными.

Существует подход, который свободен от перечисленных недостатков. Он состоит в том, что для элементов хеш-таблицы добавляется состояние удалено. Данное состояние в процессе поиска интерпретируется, как занято, а в процессе записи как свободно. Опишем алгоритмы вставки и удаления элемента для таблицы, имеющей три состояния элементов.

Вставка элемента:

1. i=0

2. $a=(h(key)+c*i) \bmod i$

3. Если *t(a)=свободно,* или *t(a)=удалено*, то *t(a)=key*, записать элемент, элемент добавлен, иначе $i=i+1$, перейти к шагу 2.

Удаление элемента:

1. i=0

2. $a=(h(key)+c*i) \bmod i$

3. Если *t(a)=key,* то *t(a)=удалено*, элемент удален. Если *t(a)=свободно*, то элемент не найден $i=i+1$, перейти к шагу 2.

Как видно, алгоритм поиска для таблицы, имеющей три состояния, почти не отличается от алгоритма поиска без учета удалений. Разница состоит в том, что при организации самой таблицы необходимо

отмечать свободные и удаленные элементы, например, зарезервировав два значения для ключевого поля. Другой вариант реализации может предусматривать введение дополнительного поля, в котором фиксируется состояние элемента. Длина такого поля может составлять всего два бита, что вполне достаточно для фиксации одного из трех состояний.

Слишком частое удаление элементов приведет к появлению большого числа ячеек, отмеченных как удаленные. В свою очередь, это увеличит число зондирований, требуемое для обнаружения попадания или промаха. Если число удаленных ячеек станет слишком большим, желательно выделить новую таблицу и скопировать в нее все элементы.

Однако возможен еще один алгоритм удаления. Удалим элемент и отметим ячейку как свободную. Затем перераспределим все элементы, расположенные в кластере удаленного элемента. Для этого поэлементно будем удалять и вновь вставлять элементы.

Удаление элемента:

1. i=0

2. $a=(h(key)+c*i) \bmod i$

3. Если *t(a)=key*, то *t(a)=свободно*, элемент удален. Если *t(a)=свободно*, то элемент не найден *i=i+1*, перейти к шагу 2.

4. *i=i+1*

5. Если *t(a)=key*, то *t(a)=свободно* вставить элемент *t(a)*, перейти к шагу 5.

Квадратичное зондирование. Метод отличается от линейного тем, что шаг перебора элементов нелинейно зависит от номера попытки найти свободный элемент (рис. 8.5):

$a=(h(key2)+c*i+d*i^2) \bmod n$

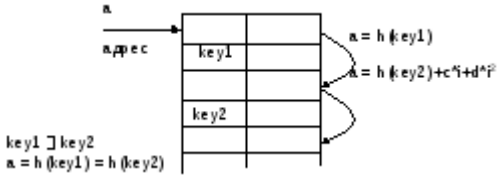


Рис. 8.5. Квадратичное зондирование.

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов. Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки. Кроме того, в отличие от локализованных кластеров при линейном зондировании, здесь возможно появление кластеров, распределенных по всей таблице. Другой серьезный недостаток состоит в том, что квадратичное зондирование не гарантирует посещение всех ячеек.

Двойное хеширование. Метод основан на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций (рис. 8.6):

$$a=(h1(key)+i*h2(key)) \bmod n$$

Если первая функция для двух ключей генерирует одинаковый индекс, маловероятно, что вторая функция сгенерирует тот же индекс. Таким образом, два ключа, которые первоначально хешируются в одну ячейку, затем не будут соответствовать одной и той же последовательности зондирования и будет ликвидирована неизбежная кластеризация, сопряженная с линейным зондированием.

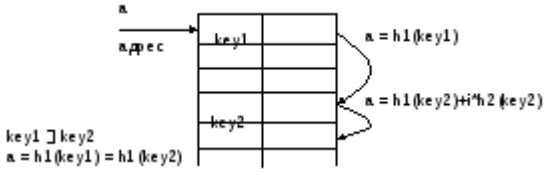


Рис. 8.6. Двойное хеширование.

Если размер таблицы равен простому числу, последовательность зондирования обеспечит посещение всех ячеек, прежде чем начнется сначала, что позволит избежать проблем, связанных с квадратичным и псевдослучайным зондированием. При использовании метода вторая функция не должна возвращать 0. Для этого в самой функции хеширования следует выполнить деление по модулю на aTableSize-1 (получить значение в диапазоне 0..aTableSize-2), а затем добавить 1.

Псевдослучайное зондирование. Метод требует использования генератора случайных чисел и заключается в следующем:

1. выполнить хеширование ключа для получения индекса, но не выполнять деление по модулю на размер таблицы;
2. установить начальное значение генератора, равным полученному значению;
3. сгенерировать первое случайное число с плавающей точкой (0..1) и умножить его на aTableSize для получения целочисленного значения в диапазоне (0..aTableSize-1).

В результате будет первая точка зондирования. Если ячейка занята, сгенерировать следующее случайное число, умножить на размер таблицы и вновь выполнить зондирование, пока не обнаружится свободная ячейка.

Метод предотвращает появления кластеров, характерных для линейного метода, однако не гарантирует посещение каждой ячейки таблицы. Вероятность пропуска пустой ячейки достаточно не велика, но вполне возможна, когда таблица заполнена в значительной степени.

Разрешение коллизий посредством закрытой адресации. В методе связывания используются дополнительные ячейки, помимо тех, что требуются хеш-таблице. Сначала с помощью хеш-функции выполняется вычисление индекса, а затем элемент сохраняется в односвязном списке, расположенном в данной ячейке. В случае возникновения коллизии при заполнении таблицы в список для требуемого адреса хеш-таблицы добавляется новый элемент (рис. 8.7).

Для поиска в хеш-таблице вычисляется адрес по значению ключа. Затем выполняется поиск в списке, связанном с вычисленным адресом. Процедура удаления из таблицы сводится к поиску элемента и его удалению из списка.

Существует несколько вариантов вставки элемента в список: поместить элемент в начало списка, в конец списка или упорядочить элементы списка и поместить в соответствующей позиции сортировки. Каждый способ имеют свои преимущества.

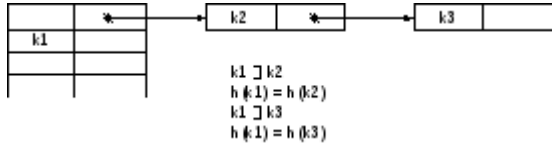


Рис. 8.7. Разрешение коллизий методом связных списков.

В первом случае имеет место эффект стека, т.к. последние помещенные элементы будут найдены первыми в случае их поиска. Такой вариант подходит, когда поиск новых элементов будет выполняться чаще поиска старых. Второй способ реализует эффект очереди, когда чаще должны быть востребованы более старые элементы. В последнем случае нет предпочтений, но любой элемент важно найти максимально быстро. Такой способ имеет преимущество только при большом числе элементов. Однако в этом случае желательно расширить таблицу, чем допускать появление большого числа элементов.

При использовании связывания никогда не возникнет ситуации нехватки места. Возможно добавление любого числа элементов в хеш-таблицу и при этом будет происходить только увеличение связных списков. Однако это преимущество имеет и обратную сторону. Проблема заключается в том, длина списков будет расти все больше и больше, и время поиска будет также увеличиваться. Поскольку любая имеющая *смысл* операция, которую можно выполнить с хеш-таблицами, предполагает поиск, большая часть времени будет тратиться на поиск в связных списках, который реализуется оператором сравнения.

Напомним, для минимизации числа зондирований таблицу следует расширять каждый раз, когда коэффициент загрузки превышает 60%. Каждое сравнение в списке можно рассматривать с позиций зондирования. Идея хеширования и заключается в сведении до минимума числа сравнений.

Для алгоритма связывания коэффициент загрузки по-прежнему может быть вычислен как число элементов, деленное на число ячеек, и может иметь значение большее 1.0 (можно представить средней длиной связных списков, присоединенных к ячейкам таблицы). Для несортированного списка среднее число зондирований для успешного поиска равно F/2, для безрезультатного F. Для отсортированного списка оба значения следует разделить на $\log_2(F)$.

Следовательно, несмотря на первую привлекательность применения связных списков, выгода от их использования может быть сведена на нет. В любом случае следует использовать расширение таблицы и эффект стека или очереди при помещении элементов.

4.8.1.4. Переполнение таблицы и рехеширование

По мере заполнения хеш-таблицы будут происходить коллизии и в результате их разрешения методами открытой адресации очередной адрес может выйти за пределы адресного пространства таблицы (рис. 8.8). Чтобы такое явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией. С одной стороны это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой к нерациональному расходованию адресного пространства.

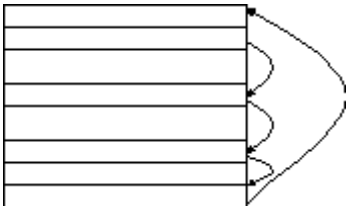


Рис. 8.8. Циклический переход к началу таблицы.

Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных элементов. Поэтому во всех примерах использовался циклический переход к

началу таблицы, путем взятия остатка от целочисленного деления адреса на длину таблицы.

Однако здесь не учитывается возможность многократного превышения адресного пространства. Более корректным будет алгоритм, использующий сдвиг адреса на 1 элемент в случае каждого повторного превышения адресного пространства. Это повышает вероятность найти свободные элементы в случае повторных циклических переходов к началу таблицы. Формула вычисления адреса будет иметь следующий вид:

$a=((h(key)+c*i) div\ n+(h(key)+c*i) \bmod n) \bmod n$

Степень загрузки таблицы и выбор хеш-функции также влияют на возможность выхода за пределы адресного пространства таблицы. При большой загрузки возникают частые коллизии и циклические переходы в начало таблицы. При неудачном выборе функции происходят аналогичные явления. В худшем случае при полном заполнении таблицы алгоритмы циклического поиска свободного места приведут к зацикливанию. Поэтому необходимо избегать плотного заполнения таблиц.

Не всегда при организации хеширования можно правильно оценить требуемую длину таблицы, поэтому в случае большой загрузки может понадобиться рехеширование. В этом случае увеличивают длину таблицы, изменяют хеш-функцию и переупорядочивают данные.

Производить отдельную оценку плотности заполнения таблицы после каждой операции вставки нецелесообразно, поэтому можно производить оценку косвенным образом по числу коллизий во время одной вставки. Достаточно определить некоторый порог числа коллизий m, при превышении которого следует произвести рехеширование.

Алгоритм вставки, реализующий такой подход:

1. $i=0$

2. $a=((h(key)+c*i) div\ n+(h(key)+c*i) \bmod n) \bmod n$

3. Если *t(a)=свободно*, или *t(a)=удалено*, то *t(a)=key*, записать элемент, элемент добавлен. Если *i>m*, требуется рехеширование *i=i+1*, перейти к шагу 2.

4.8.2. Организация данных для поиска по вторичным ключам

До сих пор рассматривались способы поиска в таблице по ключам, позволяющим однозначно идентифицировать запись. Такие ключи называют *первичными*. Возможен вариант организации таблицы, при котором отдельный ключ не позволяет однозначно идентифицировать запись. Такая ситуация часто встречается в базах данных. Идентификация записи осуществляется по некоторой совокупности ключей. Ключи, не позволяющие однозначно идентифицировать запись в таблице, называются *вторичными ключами*.

Однако даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные. Например, поисковые системы сети Internet часто организованы как наборы записей, соответствующих Web-страницам. В качестве вторичных ключей для поиска выступают ключевые слова, а задача поиска сводится к выборке из таблицы некоторого множества записей, содержащих требуемые вторичные ключи.

4.8.2.1. Инвертированные индексы

Рассмотрим метод организации таблицы с инвертированными индексами. Для таблицы строится отдельный набор данных, содержащий инвертированные индексы. Вспомогательный набор содержит для каждого значения вторичного ключа отсортированный список адресов записей таблицы, которые содержат данный ключ (рис. 8.9).

Поиск осуществляется по вспомогательной структуре достаточно быстро, так как фактически отсутствует необходимость обращения к основной структуре данных. Область памяти, используемая для индексов, является относительно небольшой по сравнению с другими методами организации таблиц. Недостатками такой организации являются большие затраты времени на составление вспомогательной структуры данных и ее обновление. Причем эти затраты возрастают с

увеличение объема базы данных. Система инвертированных индексов является чрезвычайно удобной и эффективной при организации поиска в больших таблицах.

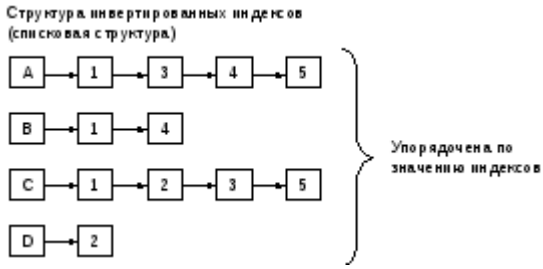


Рис. 8.9. Метод организации таблицы с инвертированными индексами.

4.8.2.2. Битовые карты

Для таблиц небольшого объема используют организацию вспомогательной структуры данных в виде битовых карт (рис. 8.10). Для каждого значения вторичного ключа записей основного набора данных записывается последовательность битов. Длина последовательности битов равна числу записей. Каждый бит в битовой карте соответствует одному значению вторичного ключа и одной записи. Единица означает наличие ключа в записи, а ноль – отсутствие.

Основным преимуществом такой организации является простая и эффективная организация обработки сложных запросов, которые могут объединять значения ключей различными логическими предикатами. В этом случае поиск сводится к выполнению логических операций запроса непосредственно над битовыми строками и интерпретации результирующей битовой строки. Другим преимуществом является простота обновления карты при добавлении записей.

К недостаткам битовых карт следует отнести увеличение длины строки пропорционально длине файла. При этом заполненность карты единицами уменьшается с увеличением длины файла. При большой длине таблицы и редко встречающихся ключах битовая карта превращается в большую разреженную матрицу, состоящую в основном из одних нулей.

Рис. 8.10. Организация вспомогательной структуры данных в виде битовых карт.

5. Модели данных

5.1. СОСТАВНЫЕ ЕДИНИЦЫ ИНФОРМАЦИИ

Каждый из наблюдаемых объектов, процессов характеризуется рядом присущих ему свойств. Но точно так же, как взятое в отдельности любое свойство еще не представляет сущность (объект, процесс) в целом, так и изолированно взятый тот или иной реквизит, характеризующий своим значением одно из свойств сущности, не может представлять законченного сообщения о наблюдаемом объекте (процессе). Требуется некоторая взаимосвязанная совокупность реквизитов для того, чтобы воспроизвести некоторое сообщение о сущности, определенную информацию о явлении.
Каждое *j*-е свойство в сообщении C_i представлено значением определенного приписанного этому свойству реквизита R_j:

$$C_i = (R_1, R_2, \ldots, R_j, \ldots, R_m),$$

где реквизиты R_j могут быть и признаками, и числовыми переменными-основаниями.
Реквизитом-признаком называется такой реквизит, значение которого определяет некоторое обстоятельство действия (место действия, действующих лиц, предметы и продукты труда, время и др.).
Реквизит-основание — это такой реквизит, значение которого определяет некоторую меру действия (количество или стоимость предметов и продуктов труда, норму выработки или времени и др.). Чаще реквизит-основание является реквизитом числового типа (иногда его называют количественным).
Каждый реквизит в сообщении имеет лишь одно значение (строку или число). Однако поскольку одна и та же сущность (допустим, факт отпуска изделий покупателям) фиксируется многократно с возникновением каждый раз нового сообщения, значения любого реквизита R_j меняются в зависимости от обстоятельств.
Каждое из сообщений, отображающих какой-либо один хозяйственный факт, глубоко индивидуально, поскольку варьируются значения по составным свойствам сущностей. Так, при отпуске готовых изделий покупателям сообщения могут фиксироваться по каждому из складов, по каждому из наименований продукции, для каждого из покупателей, каждый день и т. д. В связи с меняющимися значениями свойств этой сущности все сообщения будут отличаться друг от друга.
Так как каждый из m реквизитов сообщения C_i может принимать одно из K_j значений, где K_j—длина номенклатуры для реквизита-признака и диапазон значений для реквизита числового типа, то потенциально множество значений сообщения данного вида равно произведению $\prod_{i=1}^{m} K_j$.

В действительности, однако, из-за наличия определенной логической взаимосвязи реквизитов, различной вероятности появления отдельных значений реквизита и сочетаний значений разных реквизитов множество значений меньше теоретически возможного, но тем не менее, как правило, велико.
Каждое сообщение в множестве сообщений данного вида отличается от другого значением хотя бы одного из входящих в сообщение реквизитов. Все множество этих сообщений объединяется в один вид благодаря одинаковому составу свойств, отображаемых реквизитами, или структурой сообщения.
Структурой сообщения объединяется некоторая совокупность разных реквизитов, т. е. в данном случае некоторое более сложное по

структуре информационное образование, состоящее из **элементарных единиц информации — реквизитов**.

Единицу информации, состоящую из совокупности других единиц информации, ассоциативно связанных между собой некоторыми отношениями, назовем ***составной единицей информации*** (СЕИ), или просто составной. Единицу информации, входящую в СЕИ, назовем составляющей единицей информации, или просто составляющей. В рассмотренном выше примере в качестве составляющих использовались реквизиты $R_1, R_2, \ldots, R_m$.

Составляющая единица информации может быть по положению в структуре, в свою очередь, составной единицей информации, но более низкого уровня, чем СЕИ, в состав которой входит эта составляющая. Наоборот, СЕИ может быть составляющей, если она находится в структуре не на первом (для СЕИ) уровне, а в составе другой, более укрупненной СЕИ.

Для каждой СЕИ будем различать ее наименование, структуру, значение и некоторые специальные свойства.

Наименование СЕИ (или имя) служит для обращения к ней и обычно представляется словом или группой слов, например «движение материалов за месяц». Чаще используются **сокращенные названия СЕИ — идентификаторы**. **Для однозначной трактовки возможных употреблений синонимов СЕИ применяется тезаурус СЕИ.**

Структурой СЕИ называется ее реквизитный состав с учетом иерархического вхождения СЕИ более низкого уровня в состав рассматриваемой СЕИ. **Рекурсивность определения структуры СЕИ обеспечивает возможность построения весьма сложных информационных конструкций, вплоть до интегрированных баз данных.**

Под значением СЕИ понимается некоторая конструкция, в которой каждому реквизиту, входящему в структуру СЕИ, присвоено значение или некоторое множество значений.

Для СЕИ могут быть определены арифметические, логические и текстовые операции, а также операции отношения. При арифметических операциях каждый реквизит, входящий в структуру СЕИ, участвует в арифметических операциях над реквизитами, ему может быть присвоено некоторое значение или множество значений. При логических операциях СЕИ рассматривается как некоторая переменная булевского типа, которой может быть присвоено значение этого же типа. При операциях отношения СЕИ рассматривается как множество значений, над которыми определены операции реляционной алгебры. Естественно, что при выполнении операций СЕИ выступают в качестве операндов соответствующих выражений.

Примером составной единицы информации может быть некоторое множество документированной информации (также относимой к структурированной). Такая информация может быть представлена на любом носителе данных. Именно анализ такой информации позволяет в определенной мере изучить состав, внутреннее строение и свойства обрабатываемой информации и информации, получаемой в результате обработки.

Моделью данных называется формализованное описание структуры единиц информации и операций над ними в информационной системе. Любая модель данных должна обеспечивать представление основных категорий восприятия реального мира — объектов, их свойств, связей и взаимодействий объектов
Наиболее распространены реляционная, сетевая и иерархическая модели данных
Основным понятием для этих моделей является отношение, причем реляционная модель данных оперирует отношениями общего вида, а остальные модели — отдельными частными случаями отношений

5.2. РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ

Реляционная модель данных **является совокупностью отношений, из которых образуются новые производные отношения в результате выполнения запросов пользователей информационной системы. Множества, образующие область определения отношений в реляционной модели данных, могут содержать только значения реквизитов.**
Математический аппарат, который позволяет записывать структуру отношений, а также производить преобразования отношений, называется **реляционной алгеброй**.
Структура отношения (СЕИ, фрагмента СЕИ) определяется указанием имени отношения *R* и перечислением в скобках имен реквизитов, связанных с каждым множеством из области определения, например *R (A, B, . ., F),* где *A, B, ., F* — имена реквизитов. Иногда **структуру отношения называют схемой отношения.**
Рассмотрим операции, которые могут быть произведены над отношениями. Как правило, в перечень включаются следующие операции: **объединение, пересечение, вычитание, проекция, произведение, ограничение, соединение, деление и выборка**.
Объединение, пересечение и вычитание — бинарные операции, они выполняются над двумя отношениями-синонимами. Синонимами

называются отношения R_1 и R_2 с одинаковым порядком n, если для реквизита A_i (i = 1,2,.., n) из отношения R_1 найдется реквизит B_j (j = 1, 2,.., n) из R_2, такой, что значения A_i и B_j принадлежат общей области определения. Реквизиты A_i и B_j, обладающие этим свойством, называются р о л е в ы м и.

Результат объединения отношений R_1 и R_2 является отношением того же порядка, строки которого принадлежат или R_1 или R_2, или им обоим. Пересечение отношений R_1 и R_2 состоит из строк, принадлежащих R_1 и R_2 одновременно. Если из отношения R_1 вычитается отношение R_2, то результат содержит строки из R_1, отсутствующие в R_2.

Рассмотрим пример с таблицами значений отношений R_1 и R_2 следующего вида:

R_1

G_1	P	U
q_1	2	u_1
q_2	6	u_2
q_3	3	u_1
q_2	1	u_2
q_4	5	u_2
q_3	2	u_4

R_2

G_2	E	U
q_2	6	u_2
q_1	2	u_4
q_3	7	u_1
q_3	3	u_1

В заголовках столбцов показаны имена реквизитов, каждая строка таблицы представляет собой кортеж, R_1 и R_2 являются синонимами, пары ролевых реквизитов — G_1 и G_2, P и E.

Отношения $R_A = R_1 \cup R_2$, $R_B = R_1 \cap R_2$, $R_c = R_1 \setminus R_2$ имеют следующие значения. Реквизиты в R_A, R_B, R_c получили новые имена.

R_A

G	F	U
q_1	2	u_1
q_2	6	u_2
q_3	3	u_1
q_2	1	u_2
q_4	5	u_2
q_3	2	u_4
q_1	2	u_4
q_3	7	u_1

R_B

G	F	U
q_2	6	u_2
q_3	3	u_1

R_C

G	F	U
q_1	2	u_1
q_2	1	u_2
q_4	5	u_2
q_3	2	u_4

Для определения операции проекции рассмотрим отношение R порядка n. Реквизиты отношения нумеруются целыми числами от 1 до n. Список из нескольких целых чисел (обозначим его L) определяет номера реквизитов и их взаимный порядок в отношении после выполнения проекции. Запись операции проекции означает, что реквизиты исходного отношения выбраны и переставлены согласно списку L. Из отношения R [L] исключаются некоторые строки, чтобы соблюдалось условие о несовпадении значений строк. Реквизиты в R [L] могут получить новые имена, не совпадающие с их наименованиями в R. Кроме того, целые числа в L могут повторяться тогда один и тот же столбец из R несколько раз входит в R [L] под разными именами. Вместо номеров реквизитов могут указываться их идентификаторы, например операции R_1 [2] и $R_1[P]$ эквивалентны.
В качестве примера построим некоторые проекции отношения R_1. Так, $R_3 = R_1$ [3] имеет значения $\{ u_1, u_2, u_4\}$. Проекция $R_4 = R_1$ [1,3,3] представлена следующей таблицей:

R_4		
G	H	H_1
q_1	u_1	u_1
q_2	u_2	u_2
q_3	u_1	u_1
q_4	u_2	u_2
q_5	u_4	u_4

Операция проекции позволяет получить «вертикальное подмножество» отношения (подмножество столбцов). Она применяется для того, чтобы исключить из обработки реквизиты отношений, значения которых не требуются при формировании ответов на запросы. Благодаря операции проекции уменьшается объем промежуточных данных и соответственно время вычисления результатов.
С помощью проекции можно получить отношения-синонимы из несовпадающих по структуре исходных отношений Значения этих проекций, вообще говоря, не совпадают, поэтому выбор исходного отношения влияет на правильность результата. Пусть нам необходим список отделов учреждения. В базе данных поддерживаются отношения СЛУЖАЩИЙ (ФАМИЛИЯ, ОТДЕЛ, ..). ТЕХНОЛОГ (ФАМИЛИЯ, ОТДЕЛ, ...) и РУКОВОДИТЕЛЬ ОТДЕЛА (ФАМИЛИЯ, ОТДЕЛ, ...) Дадим им сокращенные обозначения R_c, R_T и R_p соответственно. Проекции R_c [2] и R_p [2] приводят к формированию полного списка отделов учреждения. В то же время проекция R_T [2]

может содержать меньше отделов, чем в действительности, так как в некоторых отделах не работают технологи.

Произведение двух отношений *R* и *S* в реляционной алгебре определяется следующим образом. Строки отношения $T = R \otimes S$ ($\otimes$ — знак произведения) образуются путем сцепления во всех возможных сочетаниях строк первого отношения r_i со строками второго s_j. Сцепление означает, что строка r_i из n элементов и строка s_j из m элементов образуют новую строку длиной $n+m$, в которой сначала располагаются элементы из r_i а затем — из s_j.

Количество строк в $R \otimes S$ равно произведению числа строк в исходных отношениях *R* и S . Структура отношения $T = R \otimes S$ получается сцеплением списка реквизитов отношения *R* со списком реквизитов отношения *S*.

Рассмотрим таблицы значений отношений *R и S*.

R

A_1	A_2	A_3
a	2	*f*
b	3	*f*
c	1	*f*
d	2	*f*

S

B_1	B_2
d	2
d	3
c	4

Произведение $T = R \otimes S$ имеет таблицу:

T

A_1	A_2	A_3	B_1	B_2
a	2	*f*	*d*	2
a	2	*f*	*d*	3
a	2	*f*	*c*	4
b	3	*f*	*d*	2
b	3	*f*	*d*	3
b	3	*f*	*c*	4
c	1	*f*	*d*	2
c	1	*f*	*d*	3
c	1	*f*	*c*	4
d	2	*f*	*d*	2
d	2	*f*	*d*	3
d	2	*f*	*c*	4

Операция ограничения определена на одном отношении *R*. В этом отношении должны быть выделены два непересекающихся списка реквизитов одинаковой длины $M_1 = \{A_1, A_2, \bullet\bullet\bullet, A_k\}$ и $M_2 = \{B_1, B_2, \ldots, B_k\}$. M_1 и M_2 должны быть синонимами. Рассмотрим две таблицы r_1 и r_2, получаемые из отношения *R*. Таблица r_1 содержит все столбцы из

таблицы значений R, имена которых указаны в M_1. Одинаковые по значению строки в r_1 не подавляются. Таблица r_2 определена аналогично на списке реквизитов M_2. Между кортежами этих таблиц устанавливается бинарное отношение θ одного из следующих видов: $=$, $\neq$, $>$, $<$, $\geqslant$, $\leqslant$. Если i-я строка из r_1 и i-я строка из r_2 удовлетворяют отношению θ, то i-я строка отношения r остается в нем после выполнения операции ограничения. Это условие проверяется для всех строк.

Обозначим операцию ограничения следующим образом: $V = R\,[M_1 \theta M_2]$, где V — имя отношения, получаемого в результате ограничения.

Если списки M_1 и M_2 — однореквизитные, то применение любого из отношений θ не приводит к двусмысленности. Если M_1 и M_2 состоят из нескольких реквизитов, сравнение строк на равенство означает попарное совпадение символов в этих строках, а сравнения $<$, $>$, $\leq$, $\geq$ в этом случае неприменимы.

Рассмотрим таблицу значений отношения R_5:

R_5		
A_1	A_2	A_3
a_1	2	3
a_2	4	2
a_3	5	5
a_4	7	6
a_5	9	4
a_6	3	6

Ограничение $V = R_5[A_2 < A_3]$ дает в результате

V		
A_1	A_2	A_3
a_1	2	3
a_6	3	6

С помощью операции ограничения выделяется некоторое подмножество строк исходного отношения, для значений элементов которых выполняется отношение θ.

Операции произведения и ограничения часто встречаются в связке $W = (R \otimes S)\,[M_1 \theta M_2]$, причем M_1 — список реквизитов из R, а M_2— синонимичный список реквизитов из S. Поэтому целесообразно ввести отдельную операцию — соединение отношений R и S, обозначаемую $R\,[M_1 \theta M_2]\,S$, и определить ее с помощью тождества

$$R\,[M_1\theta M_2]\,S = (R \otimes S)\,[M_1\theta M_2].$$

Для указанных выше таблиц отношений *R* и S результат операции $W_1 = R\,[A_2 = B_2]\,S$ характеризуется таблицей

W_1				
A_1	A_2	A_3	B_1	B_2
a	2	*f*	*d*	2
b	3	*f*	*d*	3
d	2	*f*	*d*	2

Соединение $W_2 = R\,[A_2 < B_2]\,S$ порождает таблицу

W_2				
A_1	A_2	A_3	B_1	B_2
a	2	*f*	*d*	3
a	2	*f*	*c*	4
b	3	*f*	*c*	4
c	1	*f*	*d*	2
c	1	*f*	*d*	3
c	1	*f*	*c*	4
d	2	*f*	*d*	3
d	2	*f*	*c*	4

Операция соединения позволяет совместить информацию из двух отношений в одном отношении.

Важный частный случай операции соединения называется натуральным соединением (или эквисоединением). Для эквисоединения в качестве отношения θ берется только равенство, множества имен реквизитов M_1 и M_2 должны содержать пары ролевых реквизитов, принадлежащие *R u S* соответственно, в результате эквисоединения столбцы с именами из M_2 подавляются. Операция эквисоединения обозначается $Z = R \triangleright\!\triangleleft S$. Отношение W_1 полученное выше, не является эквисоединением *R* и S, поскольку имеются две пары ролевых реквизитов A_2 и B_2, A_1 и B_1. Поэтому $W_3 = R \triangleright\!\triangleleft S$ описывается таблицей

W_3		
A_1	A_2	A_3
d	2	*f*

Если отношения *R и S* являются синонимами, то эквисоединение определяется как $R \triangleright\triangleleft S = R \cap S$. Если *R* и S не содержат общих реквизитов, то по определению $R \triangleright\triangleleft S = R \otimes S$.

Таким образом, эквисоединение может быть выполнено над любыми двумя отношениями.

Операция деления рассчитана на получение «горизонтального подмножества» отношения. Она рассматривает произвольное отношение *R* (делитель) как бинарное. Для этого все реквизиты разделяются на два непересекающихся подмножества *M* и *M'* и рассматриваются две таблицы *r (M)* и *r (M'),* состоящие из соответствующих столбцов таблицы значения *R.* Для каждой строки r_i таблицы *r (M)* вводится операция «образ», результатом которой являются строки $r_j \in r\ (M')$, такие, что пара (r_i, r_j) принадлежит отношению *r*. Операция взятия образа обозначается как $im\ (r_i) = \{r'_{j1},\ r'_{j2},\ \ldots,\ r'_{jk}\}$.

Очевидно, что существуют и образы строк *r*, которые являются строками из *r (M).* Отношение-делитель S должно быть синонимом для *r (M).* Операция деления сводится к вычислению образов всех строк делителя S и формированию пересечения этих образов. Результат деления является подмножеством строк таблицы *r (M).* Общее обозначение операции деления $r\ [M \div N]\ S$, где *N* — список реквизитов отношения S.

Рассмотрим, например, процесс деления отношения R_6 на S_1 с таблицами значений:

R_6			
X	*Y*	*Z*	*U*
x_1	y_1	z_1	u_1
x_1	y_2	z_2	u_2
x_2	y_3	z_1	u_1
x_3	y_1	z_1	u_1
x_2	y_3	z_2	u_2
x_3	y_1	z_2	u_2

S_1	
Z'	*U'*
z_1	u_1
z_2	u_2

Множество *M = {Z, U)* и множество *N= {Z', U'}* являются синонимами, поскольку *Z, Z'* и *U, U'* — пары ролевых реквизитов. Вычисляем образы строк из S_1:

$im\ (z_1u_1) = \{(x_1,\ y_1),\ (x_2,\ y_3),\ (x_3,\ y_1)\}$;

$im\ (z_2u_2) = \{(x_1,\ y_2),\ (x_2,\ y_3),\ (x_3,\ y_1)\}$.

Пересечение образов дает множество $\{(x_2, y_3), (x_3, y_1)\}$, т. е. результат деления (обозначим его T_2) сведем в таблицу

T_2	
X	Y
x_2	y_3
x_3	y_1

Отметим, что объединение образов строк из S_1 в нашем примере вычисляется с помощью эквисоединения и проекции

$$T_3 = (R_6 \rhd\lhd S_1)\ [1,\ 2].$$

Врезультате получаем отношение T_3, сведенное в таблицу

T_3	
X	Y
x_1	y_1
x_1	y_2
x_2	y_3
x_3	y_1

Операция выборки является обобщением операции ограничения. Для значений реквизитов отношения R формулируются условия вида <имя реквизита> θ <значение реквизита>, где $\theta = \{=, \neq, >, <, \geqslant, \leqslant\}$. Эти условия могут связываться вместе с помощью логических операций $\wedge$ (и) $\vee$ (или) в общее условие выборки. Операция выборки обозначается следующим образом:

$$Q = R\ [\text{условие выборки}],$$

где Q — имя результата выборки. Отношение Q содержит такие строки из R, которые удовлетворяют условию выборки. Надо отметить, что все введенные ранее операции использовали в качестве операндов исключительно отношения, а в условии выборки надо указывать отдельные значения реквизитов.

В качестве примера рассмотрим для таблицы T оператор

$$Q = T\ [(A_1 = b) \wedge (B_2 < 4)].$$

Таблица значений Q содержит строки:

Q				
A_1	A_2	A_3	B_1	B_2
b	3	f	d	2
b	3	f	d	3

Корректировка отношения, т. е. добавление в таблицу новых строк (включение) либо изъятие из нее некоторых строк (исключение), производится операторами объединения и вычитания. Например, включим в отношение T_3 две строки $\{x_1, y_2\}$ и $\{x_3, y_2\}$. Они должны образовать отдельное отношение $T_в$ с таблицей:

$T_в$	
X	Y
x_1	y_2
x_3	y_2

Включение реализуется оператором объединения

$$T_4 = T_3 \cup T_в.$$

Отношение T_4 имеет таблицу:

T_4	
X	Y
x_1	y_1
x_1	y_2
x_2	y_3
x_3	y_1
x_3	y_2

Включение $\{x_1, y_2\}$ не произошло, поскольку такая строка уже имелась в T_3.

Допустим, что из T_4 необходимо исключить строки $\{x_3, y_1\}$ и $\{x_1, y_3\}$, составляющие таблицу отношения $T_и$:

$T_и$	
X	Y
x_3	y_1
x_1	y_3

Исключение выполняется оператором вычитания

$$T_5 = T_4 \setminus T_и.$$

Получаем таблицу соответствия T_5:

T_4	
X	Y
x_1	y_1
x_1	y_2
x_2	y_3
x_3	y_2

Строка $\{x_1, y_3\}$ из $T_и$ не произвела никакого действия, поскольку равной ей по значению строки в T_4 нет.

Реализацию конкретных запросов пользователей с помощью операторов реляционной алгебры рассмотрим на примере двух отношений по СЕИ из учета движения материальных ценностей в производстве.

1. Приходный ордер
   ОРДЕР. (СК, ПОСТ, ННОМ, КОЛ, Ц).
2. Лимитная карта
   ЛИМК. (СК, ЦЕХ, ННОМ, ЛИМ, КОЛ, ДАТА, ОСТ).

Список реквизитов:
СК — склад;
ПОСТ — поставщик;
ННОМ — номенклатурный номер материала;
КОЛ — количество материала;
Ц — цена;
СУМ — сумма;
ЛИМ — лимит запаса материала;
ОСТ — остаток запаса материала.

Значения отношений для СЕИ ОРДЕР и ЛИМК. показаны в табл. 1 и 2.

Таблица 1

ОРДЕР				
СК	ПОСТ	ННОМ	КОЛ	Ц
01	ПОСТ1	427211	1600	17,6
02	ПОСТ2	741653	940	12,0
04	ПОСТ2	301621	720	19,4
03	ПОСТ3	219432	800	15,2
01	ПОСТ4	427211	1260	17,6
02	ПОСТ4	741653	400	12,0
04	ПОСТ4	301621	670	19,4
02	ПОСТ5	177216	900	10,5

Таблица 2

ЛИМК						
СК	ЦЕХ	ННОМ	ЛИМ	КОЛ	ДАТА	ОСТ
01	01	427211	6000	18	140281	9200
04	02	301621	5400	26	060481	7000
02	02	177216	7600	114	260181	5200
01	02	427211	4200	82	120481	6000
02	02	741653	6500	40	170881	8500
01	03	427211	7000	15	021081	9000
04	03	301621	3800	90	090681	4000
03	03	219432	9100	74	050581	8000
02	03	741653	8300	105	100281	9600
01	04	427211	7900	56	210981	16000
04	04	301621	4800	14	071281	5200

Сформулируем несколько запросов к этим отношениям.

1. В какие цехи поступают материалы с номенклатурными номерами 301621, 427211 и 741653 одновременно?

Значения реквизитов не могут непосредственно использоваться при формулировке операций реляционной алгебры, поэтому надо создать содержащее их отношение МТ:

МТ
НМ
301621
427211
741653

Упомянутые в запросе реквизиты ЦЕХ и НИОМ принадлежат одному отношению ЛИМК. Поэтому целесообразно выполнить проекцию отношения ЛИМК, чтобы оставить в нем только реквизиты ЦЕХ и НИОМ:

ЛИМК1 = ЛИМК [2,3]

Номера цехов, в которые поступает материал с данным номенклатурным номером, выделяются операцией взятия образа номенклатурного номера в ЛИМК1. Поскольку в запросе речь идет о цехах, получающих материалы 301621, 427211, 741653 одновременно, требуется выполнить пересечение образов, т. е. деление:

ЦМ = ЛИМК1 [НИОМ ÷ НМ] МТ

Запрос может быть записан в одну строку

$$ЦМ = ЛИМК\ [2,3]\ (ННОМ \div НМ)\ МТ$$

Выполнение указанных действий приводит к следующей таблице отношения ЦМ:

ЦМ
ЦЕХ
02
03

Если бы запрашивались номера цехов, получающие хотя бы один материал из списка, зафиксированного в соответствии МТ, образы номенклатурных номеров в ЛИМК1 пришлось бы объединить, т. е. выполнить эквисоединение и проекцию

$$ЦМ1 = (ЛИМК1 \triangleright\triangleleft МТ)\ [1]$$

Таблица ЦМ1 имеет значения

ЦМ1
ЦЕХ
01
02
03
04

2. Какие поставщики поставляют те же материалы, что и поставщик ПОСТ2?

Снова необходимо ввести отношение

СП
П
ПОСТ2

Задача должна быть разделена на две части. Сначала получим список материалов, поставляемых ПОСТ2. Реквизиты ПОСТ и ННОМ принадлежат отношению ОРДЕР. Требуемые материалы являются образами ПОСТ2. Поэтому список материалов СМ вычисляется с помощью операций проекции и деления

$$СМ = ОРДЕР\ [2,3]\ [ПОСТ \div П]\ СП$$

Таблица значений СМ содержит строки

СМ
ННОМ
741653
301621

Поставщики, которые удовлетворяют условию запроса, должны поставлять все материалы, содержащиеся в отношении СМ, и, возможно, какие-то материалы сверх этого перечня. Пересечение образов строк из СМ, взятых из отношения ОРДЕР [2,3], решает поставленную задачу, так как поставщики обязательно поставляют и материал 741653, и материал 301621. Поэтому список поставщиков СПИСОК, поставляющих те же материалы, что и ПОСТ2, вычисляется таким образом:

СПИСОК = ОРДЕР [2,3] [ННОМ ÷ ННОМ]СМ

Запрос может быть переписан в одну строку

СПИСОК = ОРДЕР [2,3] [ННОМ ÷ ННОМ] (ОРДЕР [2,3] [ПОСТ ÷ ÷ П] СП)

Результирующая таблица отношения СПИСОК

СПИСОК
ПОСТ
ПОСТ2
ПОСТ4

Естественно, что сам ПОСТ2 удовлетворяет требованиям запроса.

3. Материалы каких поставщиков требуются во всех цехах предприятия?

Запрос также состоит из двух частей. Очевидно, что список всех цехов предприятия на основе имеющихся отношений получить невозможно, поэтому следует ограничился цехами, получающими материалы со стороны.

Нужный список СЦ получается операцией проектирования:

СЦ = ЛИМК [2]

Поскольку запрос охватывает все цехи из СЦ, далее можно пользоваться только операцией деления (пересечение образов). Список материалов, требуемый во всех цехах из СЦ, это

СМ = ЛИМК [2,3] [ЦЕХ ÷ ЦЕХ] (ЛИМК [2])

Поставщики, поставляющие весь набор материалов из СМ, определяются путем

СПИСОК = ОРДЕР [2,3] [ИНОМ ÷НОМ] СМ

Запрос также можег быть записан в одну строку

СПИСОК = ОРДЕР [2,3] [ННОМ ÷ ННОМ] (ЛИМК [2,3] [ЦЕХ = = ЦЕХ] (ЛИМК [2]))

Результирующая таблица имеет вид

СПИСОК
ПОСТ
ПОСТ1
ПОСТ4

4. Какие материалы поступают либо в цех 02, либо в цех 03?

Реквизиты, которые упоминаются в формулировке запроса (ЦЕХ и ННОМ), находятся в одном отношении ЛИМК. Для выполнения запроса необходимо отношение СЦ с таблицей

СЦ
ЦЕХ
02
03

Материалы, поступающие в цех 02, являются образами значения ЦЕХ = 02 в отношении ЛИМК [2,3]. То же самое справедливо для материалов, поступающих в цех 03. Поскольку требуются номенклатурные номера материалов, поступающих либо в цех 02, либо в цех 03, эти образы необходимо объединить, что равносильно выполнению эквисое-динения. Окончательно получаем выражение

СМАТ = (ЛИМК [2,3] ▷◁ СЦ) [2]

Отношение СМАТ представляет собой список номенклатурных номеров материалов, поступающих либо в цех 02, либо в цех 03, и имеет значение

СМАТ
ННОМ
301621
177216
427211
741653
219432

5. По каким номенклатурным номерам имеется достаточный запас материалов?

Требуется определить все материалы, по которым лимит не превышает наличного остатка. Так как реквизиты (ЛИМ и ОСТ) находятся в одном отношении, то достаточно применить операцию ограничения

СПМ = ЛИМК [ЛИМ ⩽ ОСТ] [3]

Тогда таблица СПМ содержит строки

СПМ
ННОМ
427211
301621
741653

6. Для каждого поставляемого материала выбрать его номенклатурный номер и список поставщиков.

Требуемая информация получателя в результате выполнения проекции

ПОСТМАТ = ОРДЕР [3,2]

Отношение ПОСТМАТ для каждого номенклатурного номера материала содержит соответствующие имена поставщиков. Однако для удобства дальнейшего использования отношение ПОСТМАТ необходимо отсортировать, поэтому окончательный вид запроса следующий:

ПОСТМАТ = SORT UP [1, ОРДЕР [3,2]]

Значением ПОСТМАТ является таблица

ПОСТМАТ	
ННОМ	ПОСТ
177211	ПОСТ 5
219432	ПОСТ 3
301621	ПОСТ 2
301621	ПОСТ 4
427211	ПОСТ 1
427211	ПОСТ 4
741653	ПОСТ 2
741653	ПОСТ 4

Операторы реляционной алгебры могут быть использованы для выражения достаточно сложных запросов. В тех случаях, когда средств реляционной алгебры оказывается недостаточно, необходимо изменить структуру отношений в базе данных.

5.3. Нормализация отношений

Задача группировки имен реквизитов в отношения, набор которых заранее не фиксирован, допускает множество различных вариантов решений. Рациональные варианты группировки должны учитывать следующие требования:

корректировка отношений не должна приводить к двусмысленности или потере информации;

перестройка набора отношений при введении новых типов данных должна быть минимальной.

Нормализация представляет собой один из наиболее изученных способов преобразования отношений, позволяющих улучшить характеристики СЕИ и базы данных по перечисленным критериям. Среди зависимостей между реквизитами отношения специально выделяются **функциональные зависимости**. Реквизит *B* отношения *r* функционально зависит от реквизита *A,* если в каждый момент времени каждому значению *A* соответствует не более чем одно значение *B.* В свою очередь *A* функционально определяет *B.* Иллюстрирует это понятие рис. 1; каждое ребро в графе определяет элемент отношения *r (A, B).*

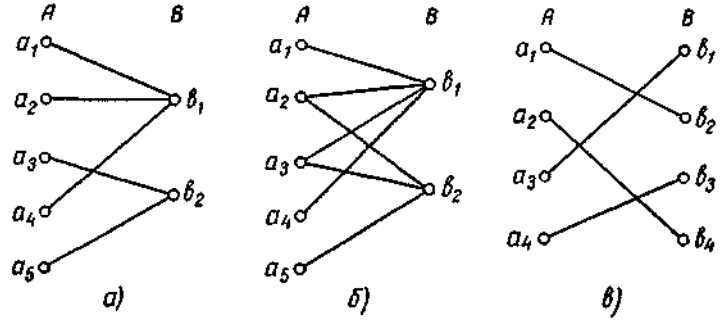


Рис. 1. Зависимости между реквизитами:
а — функциональная A→B; *б* — нефункциональная (кортежи $<a_3, b_1>$, $<a_2, b_2>$ нарушают свойство функциональности),
в — взаимнооднозначная *A*↔B

Функциональная зависимость *B* от *A* соответствует понятию однозначной функции *B = f (A).* Данное определение легко распространяется на случай зависимости между группами реквизитов. Группу *M* составим из реквизитов $A_1, A_2, ..., A_m$, группу *N* — из реквизитов $B_1, B_2, ..., B_n$. Кортежи отношений *r [M]* и *r [N]* будем

считать значениями групп реквизитов *M* и *N. С* учетом этого к *M* и *N* применимо общее определение функциональной зависимости.
Наличие функциональной зависимости реквизита *B* от реквизита *A* обозначается *r . A → r . B,* ее отсутствие — *r . A ↛ r. B.* Если ясно, о каком отношении идет речь, то пишут *A →B* или $A \not\to B$. Случай *A → B, B → A* обозначается *A ↔B* и называется взаимно-однозначным соответствием (рис. 1, б). Очевидно, что в случае *A →B* число элементов в реквизите *A* больше, чем в реквизите *B,* а при взаимно-однозначном соответствии число элементов одинаково.
При обработке отношений очень важно различать кортежи по содержащимся в них значениям реквизитов, а не по взаимному расположению кортежей. С этой целью вводятся понятия **вероятного и первичного ключей.**
Вероятный ключ *K* отношения *r* является комбинацией из *n* реквизитов (возможно *n*= 1), обладающей следующими двумя свойствами: уникальностью (значения *K* (понимаемые как строки проекции *r[K])* и кортежи отношения *r* находятся во взаимнооднозначном соответствии); неизбыточностью. Набор реквизитов в *K* нельзя сократить без нарушения первого свойства. Вероятный ключ всегда существует, так как все реквизиты отношения *r* заведомо обладают первым свойством.
Если в отношении существует несколько вероятных ключей, то для идентификации кортежей используется один из них, называемый **первичным.** Реквизиты, не входящие ни в какой ключ, называются неосновными. В ряде случаев удобно ввести искусственный первичный ключ в виде нумерации кортежей отношения. Номер кортежа при этом должен стать одним из реквизитов этого отношения.
Найдем, например, ключи в отношении *Q* (ПРЕДПРИЯТИЕ, АДРЕС, ПРОДУКЦИЯ, ВЫПУСК).
Различных предприятий, естественно, меньше, чем различных кортежей в Q, так как предприятие может выпускать несколько видов продукции. Аналогично данная продукция может выпускаться несколькими предприятиями. **Поэтому ни ПРЕДПРИЯТИЕ, ни ПРОДУКЦИЯ по отдельности вероятного ключа не образуют**. **Вместе они составляют вероятный ключ**, поскольку ВЫПУСК и АДРЕС ими однозначно определяются. Если учесть зависимость ПРЕДПРИЯТИЕ ↔ АДРЕС, то появляется еще один вероятный ключ АДРЕС, ПРОДУКЦИЯ.
Легко доказать, что каждый неключевой реквизит из отношения *r* функционально зависит от каждого вероятного ключа из *r.* При предположении обратного возникает противоречие с понятием

вероятного ключа. По той же причине два вероятных ключа всегда находятся во взаимно-однозначном соответствии.

Для отношения *Q* можно теперь вывести зависимости:

ПРЕДПРИЯТИЕ, ПРОДУКЦИЯ → АДРЕС
ПРЕДПРИЯТИЕ, ПРОДУКЦИЯ → ВЫПУСК
АДРЕС, ПРОДУКЦИЯ → ПРЕДПРИЯТИЕ
АДРЕС, ПРОДУКЦИЯ → ВЫПУСК
ПРЕДПРИЯТИЕ, ПРОДУКЦИЯ ↔ АДРЕС, ПРОДУКЦИЯ

Важное свойство составного (многореквизитного) вероятного ключа состоит в том, что никакие два входящих в него реквизита не могут быть связаны функциональной зависимостью. Рассмотрим отношение *R* с составным ключом *ABC* и зависимостью $B \to C$. Докажем, что в отношении *R'*, полученном из *R* отбрасыванием реквизита С, столько же кортежей, что и в самом *R*. Допустим обратное. Тогда в *R* найдется строка ab $(a \in A, b \in B)$, которой в *R* соответствуют по крайней мере две строки abc_1 и abc_2 $(c_1, c_2 \in C)$, а это противоречит функциональной зависимости $B \to C$. Отсюда следует, что ключ *ABC* является избыточным.

Следовательно, в отношении *Q*

ПРЕДПРИЯТИЕ $\not\to$ ПРОДУКЦИЯ
ПРОДУКЦИЯ $\not\to$ ПРЕДПРИЯТИЕ
АДРЕС $\not\to$ ПРОДУКЦИЯ
ПРОДУКЦИЯ $\not\to$ АДРЕС

Следующие три свойства функциональных зависимостей справедливы как для отдельных реквизитов, так и для групп реквизитов. Поэтому условимся, что реквизиты A_i $(i = \overline{1, m})$, B_j $(j = \overline{1, r})$ и C_k $(k = \overline{1, p})$ принадлежат одному отношению *R*.

Могут быть сформулированы следующие теоремы, принадлежащие Армстронгу:

1) $A_1A_2 \dots A_m \to A_i$ для $i = \overline{1, m}$.

2) $A_1A_2 \dots A_m \to B_1B_2 \dots B_r$ тогда и только тогда, когда $A_1A_2 \dots A_m \to B_j$ для любого $j = \overline{1, r}$.

3) Если $A_1A_2 \dots A_m \to B_1B_2 \dots B_r$ и $B_1B_2 \dots B_r \to C_1C_2 \dots C_p$, то $A_1A_2 \dots A_m \to C_1C_2 \dots C_p$.

В отношении Q можно выделить следующие зависимости, вытекающие из теорем Армстронга:

ПРЕДПРИЯТИЕ, ПРОДУКЦИЯ → ПРЕДПРИЯТИЕ
ПРЕДПРИЯТИЕ, ПРОДУКЦИЯ → ПРОДУКЦИЯ
АДРЕС, ПРОДУКЦИЯ → АДРЕС
АДРЕС, ПРОДУКЦИЯ → ПРОДУКЦИЯ
ПРЕДПРИЯТИЕ, ПРОДУКЦИЯ → АДРЕС, ВЫПУСК
АДРЕС, ПРОДУКЦИЯ → ПРЕДПРИЯТИЕ, ВЫПУСК

Приведенные теоремы о функциональных зависимостях (список их может быть расширен) позволяют формализовать и автоматизировать процесс установления таких зависимостей.
Совокупность функциональных зависимостей может быть изображена графически в виде диаграммы. Вершинами диаграммы функциональных зависимостей являются различные группы имен реквизитов. Дуга от вершины к вершине означает наличие функциональной зависимости между соответствующими группами реквизитов. Зависимости, которые следуют из теорем Армстронга, на диаграмме, как правило, не показываются. Диаграмма функциональных зависимостей отношения *Q* приведена на рис. 2.

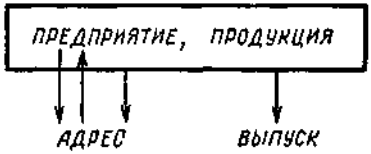


Рис. 2. Диаграмма функциональных зависимостей отношения *Q*

Вероятные ключи на любой диаграмме функциональных зависимостей соответствуют вершинам, в которые не заходит ни одна дуга. Многореквизитный ключ обводится рамкой.
Функциональные зависимости и вероятные ключи отношений должны учитываться при выполнении операций реляционной алгебры. В качестве примера рассмотрим реализацию эквисоединения. Пусть отношение *R (A, B, C)* с помощью операции проекции разделено на R_1 *(A, B)* и R_2*(B, C).* Таблицы значений отношений *R* и $T = R_1 \triangleright\triangleleft R_2$ совпадают только тогда, когда реквизит *B* является вероятным ключом в R_1 или в R_2. *A, B* и C можно рассматривать и как группы реквизитов.
Наличие в отношении достаточно большого числа функциональных зависимостей обычно приводит к усложнению процессов корректировки. Например, исключение или замена одного значения реквизита, который функционально зависит от многих других реквизитов, требует исключения или замены большого количества строк отношения. Поэтому целесообразно все отношения трансформировать в отношения более простой структуры, у которых функциональные зависимости взаимосвязаны простейшим способом. Такой процесс преобразования отношений называется **нормализацией**, а отношения, для которых не допускаются те или иные варианты функциональных зависимостей, называются **нормальными формами**.

Первая нормальная форма отношения (сокращенно 1НФ) определяется условием — среди значений реквизитов не должно быть кортежей. Ненормализованные СЕИ не соответствуют условиям 1НФ, а нормализованные СЕИ находятся в 1НФ. Если предполагается для обработки извлекать только часть реквизита (например, из даты только год), то отношение, содержащее такой реквизит, не находится в 1НФ. Условия, определяющие 1НФ отношения, не содержат никаких ограничений на возможные варианты функциональных зависимостей. Поэтому создание и корректировка значений в отношении, находящемся в 1НФ, часто бывают затруднены. В рассмотренное ранее отношение *Q,* в котором соблюдаются условия 1НФ, невозможно включить сведения о предприятии и его адресе, пока предприятие не выпустит какую-то продукцию. Если предприятие, производящее единственный вид продукции, прекращает его выпуск, то необходимое при этом удаление кортежа разрушает сведения об адресе предприятия. Далее, одинаковое значение адреса встречается во многих кортежах, если предприятие выпускает большой ассортимент продукции. Любая замена адреса предприятия становится при этом очень трудоемкой операцией. **Указанные факты объясняются функциональной зависимостью адреса от предприятия**. Подобные функциональные зависимости называются **неполными.**

Выделим в отношении *R* два различных подмножества реквизитов *F* и *E,* связанных условием $F \rightarrow E$. Если *E* функционально не зависит от любого подмножества *F,* то функциональная зависимость $F \rightarrow E$, называется полной. Если существует группа реквизитов G, такая, что $F \supset G$ и $G \rightarrow E$, то функциональная зависимость называется неполной.

Отношение *R* находится во второй нормальной форме (2НФ), если оно находится в 1НФ и каждый неосновной реквизит функционально полно зависит от каждого вероятного ключа. Если вероятный ключ одноreквизитный, то условия 2НФ соблюдаются автоматически. Отсюда, например, следует, что бинарное отношение всегда находится в 2НФ. Для отношения в 2НФ не характерны аномалии процессов формирования и корректировки, которые были рассмотрены выше на примере отношения *Q.* Поэтому отношение, не находящееся в 2НФ, целесообразно разделить на части, каждая из которых находится в 2НФ.

Выделим в отношении *R* вероятный ключ *F* и неключевые реквизиты *A, B, ..., E.* Допустим, что $F \supset G$ и $G \rightarrow E$, следовательно, *R* находится в 1НФ, но не в 2НФ. С помощью операции проекции создадим два отношения R_1 *(G, E)* и R_2 *(F, A, B, ...*), каждое из которых находится в 2НФ. R_2 содержит вероятный ключ и те неключевые реквизиты,

которые не участвуют в неполной зависимости. R_1 содержит неключевой реквизит, участвующий в неполной зависимости, и подмножество вероятного ключа, которое функционально определяет этот реквизит. В рассматриваемом примере АДРЕС функционально зависит от части первичного ключа — реквизита ПРЕДПРИЯТИЕ. Поэтому отношение Q не удовлетворяет условиям 2НФ. Надо разделить его на две проекции Q_1 (ПРЕДПРИЯТИЕ, АДРЕС) и Q_2 (ПРЕДПРИЯТИЕ, ПРОДУКЦИЯ, ВЫПУСК), каждая из которых находится в 2НФ. При необходимости можно создать исходное отношение Q с помощью эквисоединения

$$Q = Q_1 \triangleright\triangleleft Q_2.$$

Правильность эквисоединения подтверждается тем, что ПРЕДПРИЯТИЕ — вероятный ключ в Q_1.

В отношении, удовлетворяющем 2НФ, остаются транзитивные функциональные зависимости между реквизитами. Реквизит $A \in U$ транзитивно зависит от множества реквизитов $X \subset U$, если существует группа реквизитов $Y \subset U$, такая, что $X \to Y$, $Y \not\to X$, $Y \to A$.

Наличие транзитивных зависимостей может создавать нежелательные эффекты при корректировке. Например, для исключения какого-то значения реквизита A требуется удаление значительного числа строк.

Назовем некоторый реквизит (возможно, набор реквизитов), от которого какой-то другой реквизит функционально (полностью функционально) зависит, детерминантой. Тогда можно определить третью нормальную форму отношения (3НФ) условием —каждая детерминанта должна являться вероятным ключом. Если вероятный ключ единственный, то 3НФ определяется как отношение в 2НФ, в котором каждый неключевой реквизит нетранзитивно зависит от первичного ключа. Сформулированное определение не запрещает транзитивной зависимости реквизитов, входящих в первичный ключ, от других вероятных ключей, что недопустимо согласно первому определению. Поэтому первое определение содержит более сильные условия и соответствующая форма отношения часто называется усиленной 3НФ. Надо отметить, что бинарное отношение обязательно находится в 3НФ, поскольку для наличия транзитивной функциональной зависимости в отношении должны быть минимально три реквизита.

Отношение в 3НФ обладает простейшим набором функциональных зависимостей — первичный ключ функционально определяет каждый неключевой реквизит и других функциональных зависимостей в отношении нет. Отношение в 3НФ не создает аномалий при корректировке, о которых упоминалось выше. Поэтому отношение, находящееся в 2НФ, но не в 3НФ, целесообразно разделить (с

помощью операции проекции) на части, не содержащие транзитивных зависимостей.

Рассмотрим отношение СПИСОК (НС, ФИО, Г, КС, ВК), где НС — номер зачетной книжки студента, ФИО — его фамилия, имя, отчество, Г — номер группы, КС — код специальности, ВК — выпускающая кафедра. Функциональные зависимости этого отношения приводятся на рис. 3.

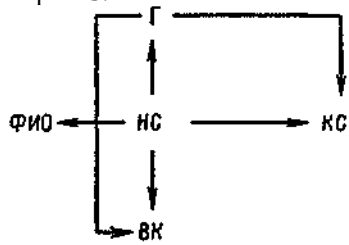


Рис 3. Функциональные зависимости отношения СПИСОК (НС, ФИО, Г, КС, ВК)

Обоснование зависимостей очень простое: каждый студент имеет одну зачетную книжку, учится в одной группе, по одной специальности, на одной выпускающей кафедре; студенты каждой группы обучаются по одной специальности, на одной выпускающей кафедре.

Соответствие СПИСОК находится в 2НФ и содержит две транзитивные зависимости НС → Г →КС и НС→ Г→ ВК. Разделить СПИСОК на проекции, удовлетворяющие ЗНФ, можно несколькими способами. Некоторые из них показаны в табл. 3. Имеются две возможности при ликвидации цепочки зависимостей $A \rightarrow B \rightarrow C$. Либо связь $A \rightarrow B$ принадлежит первой проекции, а связь $B \rightarrow C$ — второй проекции, либо А → В относится к первой проекции, а $A \rightarrow$ С — ко второй. В последнем случае очень трудно следить за соблюдением соотношения $B \rightarrow$С, так как реквизиты B и C принадлежат разным отношениям. Поэтому практически рекомендуется первый способ. Нежелательным с этой точки зрения является вариант № 4 (табл. 3).

Таблица 3

Номер варианта	Набор проекций
1	С1 (НС, ФИО, Г) С2 (НС, ВК) С3 (КС, ВК) С4 (Г, ВК) С5 (НС, КС)
2	С1 (НС, ФИО) С2 (НС, Г) С3 (Г, КС, ВК)
3	С1 (НС, ФИО, Г) С2 (Г, КС, ВК)
4	С1 (НС, ФИО, КС) С2 (Г, КС, ВК)

Если известна диаграмма функциональных зависимостей отношения, то существует алгоритм получения производных отношений в ЗНФ. На диаграмме выделим вероятные ключи (вершины, в которые не заходит ни одна дуга). Каждый вероятный ключ вместе с реквизитами, которые непосредственно функционально зависят от него, образуют отношение в ЗНФ. Затем из диаграммы функциональных зависимостей исключаются реквизиты, вошедшие в указанные отношения при условии, что от них не зависят функционально оставшиеся реквизиты. В полученной диаграмме снова найдутся вершины, в которые не заходит ни одна дуга, и процесс выделения отношений продолжается до исчерпания всех реквизитов.

Рассмотрим пример выполнения алгоритма. На рис. 4 приведена диаграмма функциональных зависимостей отношения с реквизитами СЛУЖ — служащий, ДОЛЖ — должность; ОКЛАД, ДАТА изменения должности и/или оклада; ТЕМА, по которой служащий работает в отделе НИИ; ОТДЕЛ, НИИ, ФИН — объем финансирования работ по теме.

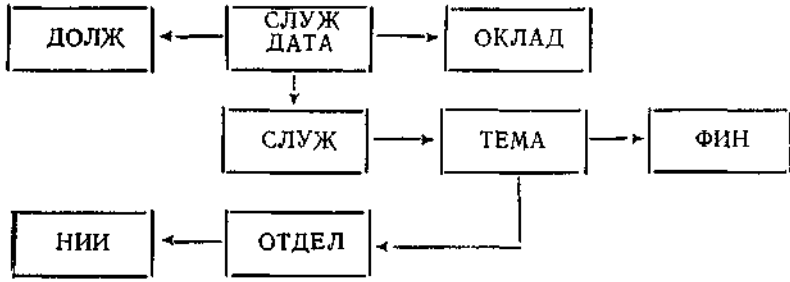


Рис. 4. Диаграмма функциональных зависимостей отношения *RT*

Зависимость (СЛУЖ, ДАТА) → СЛУЖ и все транзитивные зависимости следуют из теорем Армстронга. Транзитивные зависимости из диаграммы удалены. В начальном положении вероятным ключом является (СЛУЖ, ДАТА), что позволяет выделить отношение в ЗНФ

R1 (СЛУЖ, ДАТА, ДОЛЖ, ОКЛАД)

Затем вероятным ключом становится СЛУЖ, что соответствует отношению

R2 (СЛУЖ, ТЕМА)

Следующий вероятный ключ — ТЕМА и следующее отношение в ЗНФ

R3 (ТЕМА, ФИН, ОТДЕЛ)

И наконец,

R4 (ОТДЕЛ, НИИ)

Отношение, удовлетворяющее ЗНФ, может содержать еще одно, нежелательное с точки зрения корректировки, свойство, а именно многозначную зависимость реквизитов. Введем определение такой зависимости.

Выделим в отношении *R* три реквизита *X, Y, Z.* В *R* имеется многозначная зависимость между *X* и *Y* (обозначается $X \rightarrow\rightarrow Y$), если при наличии в *R* двух строк со значениями (x_1, y_1, z_1) и (x_2, y_2, z_2) обязательно содержатся строкисо значениями (x_1, y_1, z_2) и (x_1, y_2, z_1). Здесь $x_1 \in X, y_1, y_2 \in Y, z_1, z_2 \in Z$. При корректировке отношения, содержащего многозначно зависимые реквизиты, изменение значения одного из таких реквизитов приводит к корректировке большого числа кортежей. Поэтому целесообразно иметь дело с отношениями, не содержащими многозначно зависимых реквизитов. Это условие фиксируется в определении четвертой нормальной формы (4НФ) отношения.

Отношение находится в 4НФ, если при наличии многозначной зависимости $X \rightarrow\rightarrow Y$ либо XY содержит все реквизиты R либо вероятный ключ R содержится в X. Приведение отношения R (X, Y, Z) к 4НФ (если оно не обладает этим свойством) сводится к его разделению на две проекции $R1(X, Y)$ и $R2(X, Z)$. На рис. 5 и табл. 4 представлено отношение R с реквизитами ЦЕХ, ПРОДУКЦИЯ, СЫРЬЕ, не удовлетворяющее 4НФ из-за многозначной зависимости реквизитов ЦЕХ и ПРОДУКЦИЯ

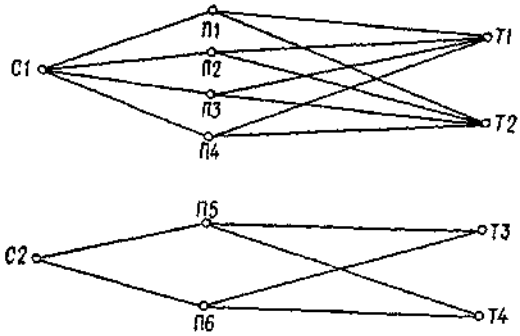


Рис. 5. Графическая интерпретация отношения R

.

Таблица 4

R		
ЦЕХ	ПРОДУКЦИЯ	СЫРЬЕ
С1	П1	Т1
С1	П2	Т1
С1	П3	Т1
С1	П4	Т1
С1	П1	Т2
С1	П2	Т2
С1	П3	Т2
С1	П4	Т2
С2	П5	Т3
С2	П6	Т3
С2	П5	Т4
С2	П6	Т4

Проекции $R1$ и $R2$ отношения R со свойствами 4НФ содержат следующие значения:

R1	
ЦЕХ	ПРОДУКЦИЯ
С1	П1
С1	П2
С1	П3
С1	П4
С2	П5
С2	П6

R2	
ЦЕХ	СЫРЬЕ
С1	Т1
С1	Т2
С2	Т3
С2	Т4

Обратное преобразование сводится к выполнению эквисоединения по атрибуту ЦЕХ.

При нормализации отношений происходит снижение их порядка. Этот процесс можно искусственно довести до логического конца и получить набор отношений с минимально возможным порядком — два.

Соответствующая модель данных называется бинарной реляционной моделью. Если исходное отношение в реляционной модели данных имеет однореквизитный первичный ключ (допустим, *A)* и структуру

$R\ (A,\ B,\ C,\ D..\ ),$

то его бинарные проекции получаются по следующему простому правилу:

$R1\ (A,\ B)$

$R2\ (A,\ C)$

$R3\ (A,\ D)$

Если первичный ключ исходного отношения состоит из нескольких реквизитов, в отношение вводится дополнительный реквизит НОМЕР КОРТЕЖА, значения которого являются порядковыми номерами кортежей (строк) отношения. Значения реквизита НОМЕР КОРТЕЖА находятся во взаимнооднозначном соответствии со строками отношения, поэтому он может стать первичным ключом отношения. Таким образом, произвольное отношение приводится к частному случаю с одно-реквизитным ключом, преобразование которого в набор бинарных отношений уже известно Возможны и другие варианты получения бинарных отношений из *k*-арного, однако всегда должна обеспечиваться возможность восстановления исходного отношения по его бинарным проекциям с помощью операции эквисоединения. Имена бинарных отношений обычно формируются из пары имен реквизитов, входящих в отношение.

Рассмотрим пример преобразования произвольного отношения в набор бинарных отношений. Отношение РЕЙСЫ описывает морские порты и

заходящие в них суда. Диаграмма функциональных зависимостей отношения РЕЙСЫ показана на рис. 6.

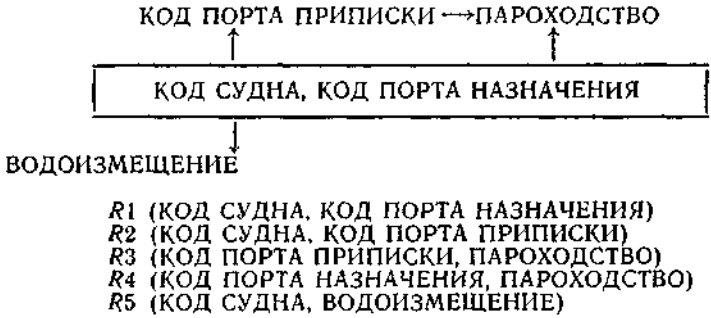


Рис. 6. Переход от диаграммы функциональных зависимостей к бинарным отношениям

Значения в бинарной реляционной модели данных группируются в **триплеты**. **Триплетом называется набор из имени неключевого реквизита,значения первичного ключа из какого-то кортежа и значения неключевого реквизита из того же кортежа.** Например,

ВОДОИЗМЕЩЕНИЕ, Лисичанск, 35000

ПАРОХОДСТВО, Баку, Каспийское

Обозначим элементы триплета следующим образом: *A* — имя неключевого реквизита, *V* — одно из его значений, *E* — соответствующее значение первичного ключа. Сам триплет получит выражение A *(E)θV*. Через θ обозначено одно из отношений $=, \neq, <, >, \leqslant, \geqslant$. Один или два компонента триплета в запросах пользователей могут объявляться неизвестными (обозначаются знаком «?»), и поэтому возможны шесть типов запросов (табл. 5).

Таблица 5

Формальная запись	Действие при обработке запроса	Пример
$A(E)=?$	Найти значение реквизита	Каково водоизмещение судна «Лисичанск»
$A(?)\Theta V$	Найти объекты с заданным значением реквизита	Какие суда имеют водоизмещение выше 10 000 тонн
$?(E)\Theta V$	Найти все имена реквизитов, имеющие данное значение для конкретного объекта	—
$?(E)=?$	Найти всю информацию о данном объекте	Какие сведения известны о судне «Лисичанск»
$A(?)=?$	Перечислить значения данного реквизита для каждого объекта	Вывести данные о водоизмещении всех судов
$?(?)\Theta V$	Перечислить все реквизиты различных объектов, имеющие данное значение	—

Более сложные запросы могут быть образованы за счет логических связок между запросами типов 1, 2, 3.

Для обработки бинарных отношений используются все операторы реляционной алгебры.

К преимуществам бинарной реляционной модели следует отнести хорошую обозримость модели из-за простоты бинарных отношений и соответствие всех отношений ЗНФ и даже 4НФ. Недостаток этой модели заключается в дублировании значений первичных ключей, следовательно, бинарная модель данных занимает больший объем памяти, чем обычная реляционная модель.

5.4. СЕТЕВАЯ И ИЕРАРХИЧЕСКАЯ МОДЕЛИ ДАННЫХ

Сетевая модель данных устанавливает два типа взаимосвязей: между реквизитами, входящими в СЕИ, и между СЕИ различной структуры. В последнем случае реквизиты нормализованных СЕИ

$S.(1 : n_S)$. $(A, B, \ldots, E)$, $T.(1 : n_1)$, $(F, G, \ldots, I)$, $\ldots$, $Q.(1 : n_2)$,

(L, M, ..., P) служат для создания отношения

$R.(A, B, \ldots, E, F, G, \ldots, I, \ldots, L, M, \ldots, P, X, Y, \ldots)$.

В отношении *R* значения реквизитов *X, Y, ...* определяются всей совокупностью СЕИ *S, T, ..., Q* и не имеют смысла для этих СЕИ, рассматриваемых по отдельности. Определенные таким образом реквизиты называются данными пересечения. Поскольку СЕИ *S, T, ..., Q* уже хранятся в базе данных и удовлетворяют требованиям 1НФ, список реквизитов в *R* можно сократить, оставив в *R* первичные ключи каждой СЕИ $(K_s, K_т, , K_Q)$ и данные пересечения. В специальной литературе по сетевым моделям данных СЕИ называются основными типами данных, а отношения, определенные на нескольких именах СЕИ, —зависимыми типами данных.

В качестве примера рассмотрим СЕИ с данными о поставщиках ПОСТ $(1 : n_1)$. (КП, АП), с данными о покупателях ПОК.$(1 : n_2)$. (КПК, АПК) и о товарах ТОВ.$(1 : n_3)$.(КТ, ТУ). В примере используются следующие сокращения: ПОСТ — поставщик, КП — код поставщика, АП — адрес поставщика, ПОК — покупатель, КПК — код покупателя, АПК — адрес покупателя, ТОВ — товар, КТ — код товара, ТУ — технические условия или ГОСТ, регламентирующие качество товара, n_1, n_2, n_3— размерности соответствующих СЕИ.

Реквизиты КП, КПК, КТ являются первичными ключами в СЕИ ПОСТ, ПОК и ТОВ, поэтому зависимый тип данных — отношение ППТ — в нашем примере может быть определен в виде

ППТ (КП, КПК, КТ, КОЛ),

где КОЛ (данные пересечения) — количество товара, поставляемого данным поставщиком какому-то покупателю.

Основные и зависимые типы данных называются взаимосвязанными, если в их структуре содержатся одинаковые имена реквизитов, а соответствующие множества значений реквизитов пересекаются. В сетевой модели данных формируется ряд требований к допустимым в модели взаимосвязанным типам данных:

каждая взаимосвязь охватывает два типа данных, причем один из них — основной, а другой — зависимый;

один и тот же тип данных не может быть одновременно и основным, и зависимым;

основной тип данных может не иметь связей с зависимыми типами; зависимый тип данных должен иметь связь хотя бы с одним основным типом данных.

Графическое изображение взаимосвязи между типами данных в сетевой модели называется диаграммой взаимосвязей. На диаграмме основные типы данных показываются прямоугольниками, зависимые типы—кругами и взаимосвязи — ребрами. Пример диаграммы

взаимосвязей, включающей ранее введенные типы данных ПОСТ, ПОК, ТОВ и ППТ, приводится на рис. 7.

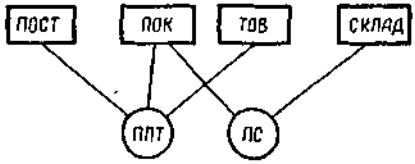


Рис. 7. Диаграмма взаимосвязей основных и зависимых типов данных

На диаграмме дополнительно указаны СЕИ СКЛАД и отношение ПС, определенное на СЕИ ПОК и СКЛАД.

Описанная выше общая схема формирования отношений между СЕИ в сетевой модели данных допускает ряд частных случаев. Отношение между $N > 2$ различными СЕИ можно трансформировать в отношении между двумя СЕИ, после чего все отношения в сетевой модели данных становятся бинарными. Алгоритм преобразования рассмотрим на примере отношения ППТ, определенного на трех СЕИ: ПОСТ, ПОК, ТОВ. Введем дополнительную СЕИ S со структурой S.(ПОСТ, ТОВ) и нормализуем ее. Каждое значение S представляет собой строку, в которой указан некоторый поставщик и какой-то поставляемый им товар. Теперь ППТ описывает связь между двумя СЕИ: S и ПОК. Указанный переход к бинарным отношениям не дает возможности, например, извлекать информацию о товарах, не зная соответствующих поставщиков. Кроме того, структура дополнительной СЕИ определена неоднозначно (допустимы варианты S.(ПОСТ, ПОК) и S.(ПОК, ТОВ)), и выбор варианта пока не формализован.

Наличие в сетевой модели данных только бинарных отношений позволяет хранить и обрабатывать отношения одними и теми же методами. По имени бинарного отношения и имени одной из СЕИ однозначно устанавливается имя второй СЕИ, что упрощает алгоритмы обработки поисковых запросов. Следует также отметить, что бинарные отношения между СЕИ встречаются наиболее часто, а отношения между $N > 4$ различными СЕИ практически не встречаются.

Рассмотрим важный частный случай бинарного отношения между СЕИ, называемый веерным отношением, или набором. Бинарное отношение между СЕИ S и СЕИ *R* называется веерным, если каждому значению СЕИ *R* ставится в соответствие единственное значение СЕИ S.СЕН S в веерном отношении называется владельцем, а СЕИ *R* — членом отношения (набора).

Бинарное отношение очень редко обладает свойством веерного отношения. В качестве примера рассмотрим разбиение множества объектов на непересекающиеся классы. Информацию об объектах содержит СЕИ — член набора, информацию о классах объектов — СЕИ — владелец набора. Иногда при формировании базы данных вносятся упрощающие предположения, равносильные определению веерного соответствия, которые справедливы для узкого круга решаемых задач, а в общем случае не верны. Например, если в учреждении запрещена работа по совместительству, то СЕИ *R*.(ФИО, ДОЛЖНОСТЬ) и S.(КОД ОТДЕЛА, ФИО РУКОВОДИТЕЛЯ) связаны веерным отношением *W,* в котором S — владелец, а *R* — член отношения.

Структура веерного отношения описывается в следующем виде: <имя отношениям (<имя СЕИ-владельца> — <имя СЕИ-члена>). Отношение из предыдущего примера имеет описание структуры *W (S — R).*

Компактное представление веерного отношения осуществляется путем формирования новых информационных элементов, называемых веерами. Введем множество значений СЕИ-владельца $A = \{a_1, a_2, \ldots a_n\}$ и множество значений СЕИ-члена

$$B = \{b_{11}, b_{12}, \ldots, b_{1i}, b_{21}, \ldots, b_{2s}, \ldots, b_{n1}, \ldots, b_{nr}\}.$$

Значения a_i, b_{ij} могут оыть заменены их сокращенными кодами (например, значениями ключей).

Веер представляет собой множество вида $\{a_i, b_{i1}, b_{i2}, \ldots, b_{in}\}$, где $a_i \in A$, $b_{ij} \in B$ и СЕИ a_i вступает в отношение со всеми b_{ij}, указанными вслед за нею. Последовательность вееров, относящихся к одной и той же паре СЕИ, образует веерное отношение.

Веерное отношение может уточняться путем введения свойств «обязательный», «необязательный», «автоматический», «неавтоматический». Свойство «обязательный» означает, что, после того как значение СЕИ включено в отношение, оно становится его постоянным членом. Его можно обновлять, но нельзя удалять из отношения. Свойство «необязательный» означает, что любое значение СЕИ из отношения можно удалять.

Свойство «автоматический» означает, что при появлении нового значения СЕИ-владельца, оно сразу же ставится в пару с некоторым значением СЕИ-члена и образует новый элемент веерного отношения. Несоблюдение этого правила характерно для свойства «неавтоматический».

Диаграмма сетевой модели данных, содержащей только бинарные отношения между СЕИ, значительно отличается от ранее

рассмотренной диаграммы. Данные пересечения на ней специально не показываются. Факт установления отношения между двумя СЕИ изображается стрелкой, которая соединяет прямоугольники с именами СЕИ. В общем случае стрелка является двунаправленной, а если отношение—веерное, то стрелка ориентирована от СЕИ-владельца к СЕИ-члену.

Если введенную диаграмму рассматривать как граф, то он будет обладать свойствами сети. Поэтому соответствующая модель данных называется сетевой.

Пример диаграммы сетевой модели данных приводится на рис. 8.

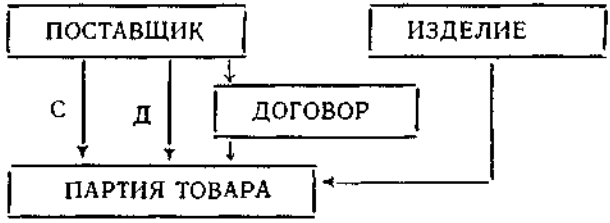


Рис. 8. Диаграмма взаимосвязей для бинарных отношений между СЕИ

Бинарные отношения определены на СЕИ ПОСТАВЩИК, ИЗДЕЛИЕ, ДОГОВОР, ПАРТИЯ ТОВАРА. При создании сетевой модели сделаны допущения — один договор регламентирует действия одного поставщика и одна партия товара формируется на основе одного договора (в общем случае эти допущения могут нарушаться). Поэтому отношения между СЕИ ПОСТАВЩИК и ДОГОВОР, между СЕИ ДОГОВОР и ПАРТИЯ ТОВАРА в нашем частном случае являются веерными. Между СЕИ ПОСТАВЩИК и ПАРТИЯ ТОВАРА поддерживаются два отношения С — планируемые поставки и *Д* — фактические поставки. Отношения С и *Д* — веерные в силу свойства транзитивности веерных отношений.
Существенным утверждением является возможность преобразования произвольного бинарного отношения на СЕИ *R* и *S* в два веерных отношения. Один элемент отношения ставит в соответствие друг другу одно значение СЕИ *R* и одно значение СЕИ *S*. Представим бинарное отношение в виде нормализованной СЕИ *RS* со структурой *RS (R,S).* Тогда каждое значение *RS* связано с единственным значением *R* и с единственным значением *S*. Первое веерное отношение устанавлива-

ется между *R* и *RS*, а второе — между *S* и *RS*. *RS* является членом в обоих веерных отношениях. Такое преобразование иллюстрирует рис.9. Каждое ребро изображает отдельный элемент отношения, образующие его значения СЕИ показаны на концах ребра. Соответствующие веерные множества имеют вид:

$$\{(a_1, a_1b_2),\ (a_2, a_2b_1, a_2b_2),\ (a_3, a_3b_1, a_3b_3, a_3b_4),\ (a_4, a_4b_2, a_4b_3),\ (a_5, a_5b_4)\};$$

$$\{(b_1, a_2b_1, a_3b_1),\ (b_2, a_1b_2, a_2b_2, a_4b_2),\ (b_3, a_3b_3, a_4b_3),\ (b_4, a_3b_4, a_5b_4)\}.$$

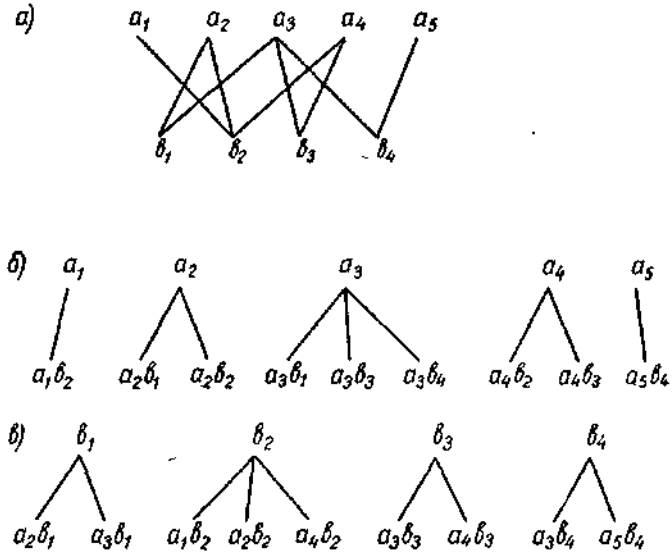


Рис.9. Преобразование бинарного отношения в два веерных

Рассмотрим отношение ПТИ, связывающее СЕИ ПАРТИЯ ТОВАРА и ИЗДЕЛИЕ на сетевой модели (см. рис. 8). Два веерных отношения, тождественных исходному отношению ПТИ, имеют структуру

W_1 (ПАРТИЯ ТОВАРА — ПТИ),
W_2 (ИЗДЕЛИЕ — ПТИ).

Введенное преобразование дает возможность все отношения в сетевой модели данных выразить в виде веерных отношений. Отрицательным следствием в этом случае является возникновение дублирующих значений. Преимущества перехода от произвольных отношений к

веерным состоят в следующем. Элементы одного веера не совпадают ни с одним элементом в других веерах, принадлежащих тому же веерному отношению, и это позволяет при обработке запросов извлекать из базы данных минимальное число вееров. Данные пересечения в веерном отношении можно просто включить в структуру СЕИ-члена, а не создавать из них самостоятельные информационные единицы. И наконец, значения СЕИ-членов в пределах одного веера можно упорядочить по отдельным реквизитам и это позволяет ускорить выборку значений СЕИ-членов из веера.
Основными операциями над веерными отношениями (наборами) в сетевой базе данных являются IM (образ) и INIM (пересечение образов). Для обработки СЕИ служат SECT (сечение), IN (пересечение), UN (объединение), SUB (вычитание).
Формат оператора IM:

M_1 = IM <имя набора> (<список значений реквизитов>, ...)

Указанные в IM значения реквизитов должны принадлежать одной и той же СЕИ и однозначно определять одно значение этой СЕИ. Знак ... в формате оператора IM, означает, что таким же способом может быть определено произвольное число значений СЕИ. Оператор IM вычисляет образ каждого значения СЕИ, т. е. множество значений СЕИ, поставленных в соответствие данному значению СЕИ, и производит объединение полученных образов. Результат получает имя M_1.
Формат оператора INIM совпадает с ранее определенным форматом оператора IM. Оператор INIM вычисляет образы значений СЕИ, указанных в описании оператора, и производит пересечение полученных образов. Результату вычислений дается имя M_1.
Если данная СЕИ образована сочетанием нескольких СЕИ (как, например, ПТИ в наборах W_1 и W_2), то оператор SECT позволяет извлечь какой-то один компонент. Формат оператора SECT:

M_2 = SECT <имя СЕИ> (<имя СЕИ>)

Первая из указанных СЕИ содержит в своей структуре имя СЕИ, приведенное в скобках, значения последней извлекаются и получают имя M_2.
Рассмотрим процесс определения партий товара, в которые входят изделия И4 и И9 (неважно, в какую именно). Для этого необходимо выполнить операторы

M_1 = IM W_2 (И4, И9)
M_2 = SECT M_1 (ПАРТИЯ ТОВАРА)

В M_1 получаются значения СЕИ ПТИ.(ПАРТИЯ ТОВАРА, ИЗДЕЛИЕ), в которых код изделия либо И4, либо И9. Оператор SECT извлекает из M_1 сведения только по партиям товара.
Действие операторов IN, UN, SUB рассмотрено в 5.2. Форматы этих операторов следующие:

M_3 = IN <имя СЕИ>, <имя СЕИ>
M_3 = UN <имя СЕИ>, <имя СЕИ>
M_3 = SUB < имя СЕИ>, <имя СЕИ>

СЕИ, входящие в описание одного и того же оператора, должны совпадать по структуре (за исключением размерности). Результат выполнения операторов IN, UN, SUB обозначен через M_3.
Если на наборах определена упорядоченность СЕИ-членов, то над СЕИ дополнительно необходимы такие операции:

MAX, MIN — выделяют все СЕИ заданного веера с ключом больше (меньше), чем у заданной СЕИ;

FIRST, LAST — определяют для данного веера СЕИ с наибольшим (наименьшим) ключом.

Формат оператора MAX:

M_4 = MAX <имя набора> (<значение СЕИ-владельца>, <значение СЕИ-члена>)

В M_4 после выполнения оператора MAX появятся значения СЕИ-членов из веера, определяемого значением СЕИ-владельца, с ключом, большим, чем у СЕИ-члена, указанного в описании оператора.
Форматы операторов MIN и MAX совпадают.
Формат оператора FIRST (или LAST):

M_5 = FIRST < имя набора> (<значение СЕИ-владельца>)

В M_5 после выполнения оператора FIRST появится значение СЕИ-члена с наибольшим ключом в веере, определяемом значением СЕИ-владельца.
Список операторов для сетевой модели данных может быть расширен введением операторов, обрабатывающих отдельные реквизиты, операторов корректировки и библиотечных операторов.
Веерное отношение, представленное в виде совокупности вееров, может быть легко преобразовано в отношение со свойствами 1НФ. Для этого из каждого веера образуем пары $<a_i, b_{i1}>, <a_i, b_{i2}>, \ldots, <a_i, b_{ij}>, \ldots, <a_i, b_{in}>$. Объединение таких пар, полученных из всех вееров веерного отношения, позволяет создать отношение, удовлетворяющее 1НФ, в структуру которого включены все реквизиты СЕИ-владельца и все реквизиты СЕИ-члена.
Сетевая модель данных является универсальной. Сетевая модель предлагает самый естественный способ отображения информационных

представлений пользователей. Также надо отметить, что системы управления сетевой базой данных пока являются более эффективными, поскольку обеспечивают меньшее время на обработку запросов, чем соответствующие системы для реляционной базы данных.

Принципиальное достоинство сетевой модели данных — наличие аппарата для определения и обработки отношений между различными СЕИ. Такая возможность может быть реализована также и в рамках реляционной модели данных, но с менее эффективными методами представления и обработки информации.

Сетевая модель данных допускает достаточно эффективный частный случай — *иерархическую модель.*

Иерархический подход во многих случаях обеспечивает естественный способ моделирования предметной области. Он особенно эффективен, если структура предметной области хорошо соответствует условиям задачи классификации.

Элементарными компонентами иерархической модели данных являются СЕИ, которые в специальной литературе (применительно к иерархической модели) часто называются **сегментами**. Составные единицы информации могут связываться между собой с помощью веерных отношений. СЕИ-владелец в этом случае чаще называется исходной СЕИ, а СЕИ-член — порожденной СЕИ. Если для двух СЕИ S_0 n S_n найдутся такие СЕИ S_1, S_2..... S_{n-1} и веерные отношения W_0, W_1 .... W_{n-1} со структурой W_0 (S_0— S_1), W_1 (S_1 — S_2),..., W_{n-1}(S_{n-1}—S_n), то говорят, что существует путь от S_0 к S_n. В иерархической модели данных требуется, чтобы путь между любыми двумя СЕИ, если он существует, был единственным. Если по аналогии с диаграммой сетевой модели данных интерпретировать имена СЕИ как вершины графа, а веерные отношения — как дуги графа, то иерархической модели данных будет соответствовать граф типа дерева. СЕИ, соответствующая корню такого графа, называется корневой СЕИ. Одно значение корневой СЕИ вместе со всеми значениями других СЕИ, которые достигаются от него по иерархической цепочке согласно веерным отношениям, образуют запись иерархической базы данных. Число различных записей иерархической базы данных равно числу различных значений корневой СЕИ. Набор записей иерархической базы данных, порожденных одной корневой СЕИ, образует одну иерархическую базу данных.

Допускается существование в ЭИС многих иерархических баз данных, в том числе пересекающихся. Две базы данных $БД_1$ и $БД_2$ называются пересекающимися, если в $БД_1$ имеется СЕИ *S,* а в $БД_2$ — СЕИ *T,* совпадающие по структуре и пересекающиеся по значениям.

Пересекающиеся значения СЕИ хранятся в одной из баз данных (например, в $БД_1$), а в $БД_2$ их место занимают адреса хранения соответствующих значений в $БД_1$.

Рассмотрим пример. Данные о движении грузов сгруппированы в две пересекающиеся базы данных. Первая состоит из СЕИ СТАНЦИЯ НАЗНАЧЕНИЯ, СОСТАВ и ГРУЗ, СЕИ второй базы данных — СОСТАВ и СТАНЦИЯ ОТПРАВЛЕНИЯ. Пересекающиеся значения СЕИ СОСТАВ хранятся в $БД_1$. Диаграмма взаимосвязей $БД_1$ и $БД_2$ приводится на рис. 10.

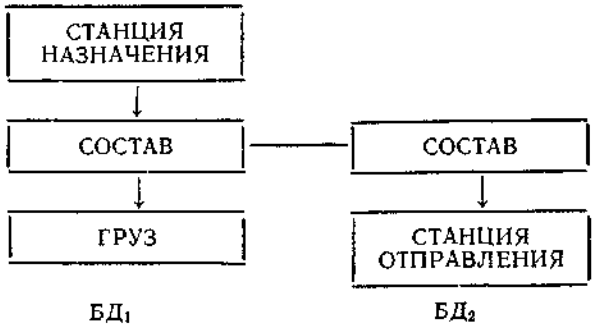


Рис. 10 Диаграмма взаимосвязей для иерархических баз данных $БД_1$ и $БД_2$

Правила построения диаграммы те же, что и для сетевой модели данных. Соединение СЕИ СОСТАВ из разных баз данных показывает, что они совпадают по структуре и пересекаются по значениям. Пример одного значения записи $БД_1$ и связанных с ним значений записей $БД_2$ иллюстрирует рис. 11.

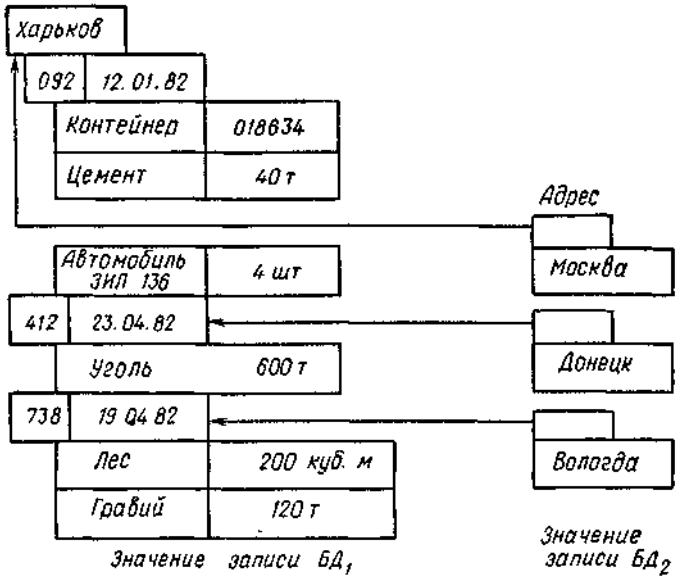


Рис. 11. Взаимосвязь значений записей в базах данных $БД_1$ и $БД_2$

СЕИ СТАНЦИЯ НАЗНАЧЕНИЯ и СТАНЦИЯ ОТПРАВЛЕНИЯ содержат только название станции, СЕИ СОСТАВ — реквизиты КОД СОСТАВА и ДАТА ПРИБЫТИЯ, СЕИ ГРУЗ—реквизиты НАЗВАНИЕ ГРУЗА и СПЕЦИФИКАЦИЯ ГРУЗА.

Рассмотрим трансформацию иерархической базы данных в множество отношений. На диаграмме иерархической базы данных выделим все пути от корневой СЕИ до таких порожденных СЕИ, которые не являются исходными ни в одном веерном отношении. Для *i*-го пути зафиксируем кортеж Γ_i, содержащий все СЕИ, входящие в этот путь. Если два кортежа Γ_i и Γ_j имеют общие СЕИ, то любой из них сокращается со стороны корневой СЕИ так, чтобы Γ_i и Γ_j имели ровно одну общую СЕИ. Каждый кортеж Γ_i порождает отношение, в структуре которого перечисляются все реквизиты из всех СЕИ, входящих в Γ_i.

На диаграмме $БД_1$, например, всего один путь, поэтому получаем одно отношение *R* со структурой

R (A1, A2, A3, A4, A5),

где используются следующие имена реквизитов: *A1* — название станции назначения, *A2* — код состава, *A3* — дата прибытия, *A4* — название груза, *A5*—спецификация груза.

Приведем фрагмент таблицы значений отношения *R*, получаемый из значения записи $БД_1$ (см. рис. 11):

R				
A1	*A2*	*A3*	*A4*	*A5*
Харьков	092	21.01.82	Контейнер	018634
Харьков	092	12.01.82	Цемент	40 т
...	.	.	...	
Харьков	092	12.01.82	Автомобиль ЗИЛ/36	4 шт
Харьков	412	23.09.82	Уголь	600 т
Харьков	738	19.04.82	Лес	200 куб. м
Харьков	738	19.04.82	Гравий	120 т

В иерархической базе данных операциями выборки являются следующие: ПОЛУЧИТЬ УНИКАЛЬНЫЙ, ПОЛУЧИТЬ СЛЕДУЮЩИЙ, ПОЛУЧИТЬ СЛЕДУЮЩИЙ ВНУТРИ.

Оператор ПОЛУЧИТЬ УНИКАЛЬНЫЙ выделяет первое из значений некоторой СЕИ S_k, удовлетворяющее сформулированным в описании оператора условиям. Каждое условие относится к одной из СЕИ, лежащих на пути между корневой СЕИ и S_k. В качестве примера рассмотрим путь $S_1, S_2, ..., ..., S_k$, где S_1 — корневая СЕИ. Условие имеет вид (<имя реквизита СЕИ> θ<значение реквизита СЕИ>), где θ — один из знаков отношения: =, ≠, >, ≥, <, ≤. Формат оператора ПОЛУЧИТЬ УНИКАЛЬНЫЙ:

ПОЛУЧИТЬ УНИКАЛЬНЫЙ <имя СЕИ S_1 > < условие>
<имя СЕИ S_2> <условие>
. . .
<имя СЕИ $S_к$ > <условие>

Запрос к рассмотренной выше базе данных о перевозках грузов «Получить первое значение СЕИ СОСТАВ для состава с номером 092» представляется в следующем виде:

ПОЛУЧИТЬ УНИКАЛЬНЫЙ СТАНЦИЯ НАЗНАЧЕНИЯ
СОСТАВ (КОД СОСТАВА='092')

Поскольку конкретное название станции назначения не оговаривается, оператор будет последовательно извлекать все записи иерархической базы данных, начиная с первой. В каждой записи будут проверяться значения реквизита КОД СОСТАВА и СЕИ СОСТАВ и при первом проявлении значения 092 в качестве ответа выдается значение всей СЕИ. 092 ЧЧ.ММ.ГГ, где ЧЧ.ММ.ГГ— значение даты прибытия.

Оператор ПОЛУЧИТЬ СЛЕДУЮЩИЙ предназначен для выборки значения СЕИ S_k,которое непосредственно следует в записи иерархической базы данных за значением СЕИ S_k, сформированным ранее оператором ПОЛУЧИТЬ УНИКАЛЬНЫЙ. Формат оператора ПОЛУЧИТЬ СЛЕДУЮЩИЙ имеет вид:

```
ПОЛУЧИТЬ СЛЕДУЮЩИЙ <имя СЕИ Sк> <условие>
```

Оператор ПОЛУЧИТЬ СЛЕДУЮЩИЙ может быть использован в циклических процессах для выборки последовательности значений S_k. В этом случае оператор сопровождается меткой и используется выражение

```
ЕСЛИ КОНЕЦ СЕИ = 'НЕТ' ТО НА <метка>
```

Для удовлетворения запроса «Получить сведения о всех действиях состава 092» необходимо выполнить следующую последовательность операторов:

```
     ПОЛУЧИТЬ УНИКАЛЬНЫЙ СТАНЦИЯ НАЗНАЧЕНИЯ
                          СОСТАВ (КОД СОСТАВА='092')
М: ПОЛУЧИТЬ СЛЕДУЮЩИЙ СОСТАВ (КОД СОСТАВА =
   = '092')
     ЕСЛИ КОНЕЦ СЕИ = 'НЕТ' ТО НА М
```

Список всех станций назначения может быть получен группой операторов:

```
     ПОЛУЧИТЬ УНИКАЛЬНЫЙ СТАНЦИЯ НАЗНАЧЕНИЯ
М:   ПОЛУЧИТЬ СЛЕДУЮЩИЙ
     ЕСЛИ КОНЕЦ СЕИ = 'НЕТ' ТО НА М
```

Оператор ПОЛУЧИТЬ СЛЕДУЮЩИЙ. ВНУТРИ выводит последовательность значений порожденной СЕИ для одного из значений СЕИ S_k, которое уже выработано операторами ПОЛУЧИТЬ УНИКАЛЬНЫЙ и ПОЛУЧИТЬ СЛЕДУЮЩИЙ. Он обычно используется в циклических процессах. Формат оператора:

```
ПОЛУЧИТЬ СЛЕДУЮЩИЙ ВНУТРИ имя порожденной СЕИ
```

Чтобы выбрать сведения о грузах, доставленных составом 092 в Харьков, требуются следующие действия:

```
     ПОЛУЧИТЬ УНИКАЛЬНЫЙ СТАНЦИЯ НАЗНАЧЕНИЯ
     СТАНЦИЯ = 'ХАРЬКОВ' СОСТАВ (КОД СОСТАВА = '092')
М: ПОЛУЧИТЬ СЛЕДУЮЩИЙ ВНУТРИ ГРУЗ
   ЕСЛИ КОНЕЦ СЕИ = 'НЕТ' ТО НА М
```

Преимущество иерархических баз данных заключается в высоком быстродействии алгоритмов выборки данных для рассмотренных выше операторов выборки,которое обеспечивается расположением

необходимых значений СЕИ в одной и той же записи иерархической базы данных.

К недостаткам следует отнести большую разницу во времени реализации симметричных запросов. Для двух СЕИ S_i и S_j, связанных веерным отношением, пара симметричных запросов формулируется в виде «Получить все значения СЕИ S_i , соответствующие одному значению S_j» и «Получить все значения СЕИ S_j, соответствующие одному значению S_i». В рассматриваемом примере это пара симметричных запросов — «Получить номера всех составов, разгружающихся в Харькове» и «Получить все станции, на которых разгружается состав 092». Второй запрос обрабатывается гораздо медленнее, поскольку СЕИ СОСТАВ является порожденной по отношению к СЕИ СТАНЦИЯ НАЗНАЧЕНИЯ. Проблема, очевидно, усугубляется по мере роста числа уровней в иерархической базе данных.

Нежелательные свойства иерархической модели данных проявляются и при корректировке. Так, при включении в базу данных кода состава, который пока еще не сделал ни одного рейса, придется включать еще и фиктивную станцию назначения как значение корневой СЕИ для этого состава. При замене каких-то значений внутри некорневых СЕИ необходимо произвести поиск во всех записях базы данных, так как нужные для обновления данные могут встретиться несколько раз.

На сетевом принципе моделирования основана *модель семантических сетей,* которая предназначена для отображения структуры понятий, сущности событий и действий. Ее отличительная особенность — наличие фиксированного списка имен для всех связей между элементами данных.

Основой для определения того или иного понятия является множество его отношений с другими понятиями. Обязательными отношениями являются класс, к которому принадлежит данное понятие, свойства, выделяющие понятие из всех понятий этого класса, примеры данного понятия. Поскольку термины, использованные в определении понятия, сами служат понятиями, то их определение организуется по той же схеме. В итоге **связи понятий образуют структуру, в общем случае сетевую.** Перечислим обязательные связи в семантической сети при установлении структуры понятий:

связь «естьнек» (от слов «есть некоторый»). Направлена от частного понятия к более общему, и показывает принад-лежность элемента к классу;

связь «имеет». Описывает случай, когда свойством является наличие некоторого предмета или владение им;

связь «есть». Относится к таким свойствам, которые имеют характер качества;

связь «может». Она связывает понятие и действия, которые могут выполняться всеми объектами, образующими понятие.

Фрагмент семантической сети для понятия ПРЕДПРИЯТИЕ приведен на рис. 12.

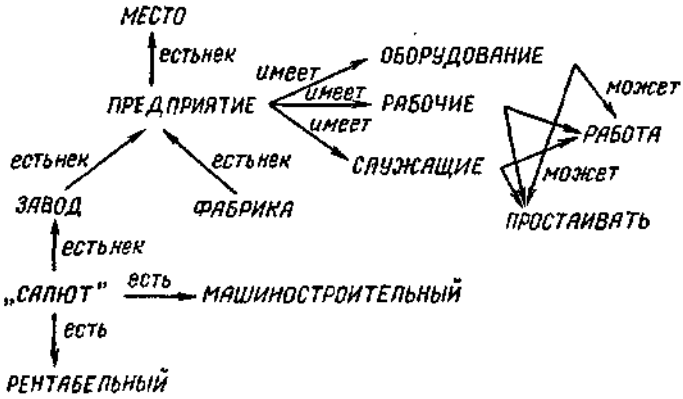


Рис. 12. Семантическая сеть для понятия ПРЕДПРИЯТИЕ

Для предприятия устанавливаются более **общие понятия** (в данном случае — ОБЪЕКТ) и **производные понятия** (ЗАВОД, ФАБРИКА), а также **конкретный завод** — «САЛЮТ». Среди ресурсов, которыми обладает предприятие, указаны ОБОРУДОВАНИЕ, РАБОЧИЕ и СЛУЖАЩИЕ и два действия, характерные для этих ресурсов.

Рассмотрим теперь представление событий и действий в семантической сети. Предварительно выделяются простые отношения, которые характеризуют основные компоненты события. В первую очередь из события выделяется действие, обычно описываемое глаголом. Далее необходимо найти объекты, которые действуют, объекты, над которыми это действие производится, и т. д. Все эти отношения предметов, событий и качеств с глаголом называются падежами. Обычно рассматриваются следующие падежи:

агент — действующее лицо, вызывающее действие;

условие — логическая зависимость, существующая между двумя событиями;

инструмент — предмет или устройство, вызывающее действие или являющееся орудием его осуществления;

место — указание на то, где происходит событие;
объект — предмет, подвергающийся действию;
цель — указание на цель действия;
качество — указание свойства понятия;
адресат — лицо, пользующееся результатом действия или испытывающее этот результат;
время — указание на то, когда происходит событие.

В качестве примера рассмотрим предложение: «Директор завода «Салют» остановил 25 марта 1982 г. цех № 4, чтобы заменить оборудование». Структура этого предложения, выраженная средствами семантической сети, показана на рис. 13.

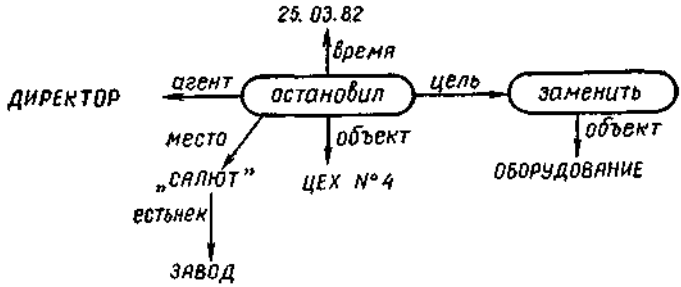


Рис. 13. Семантическая сеть для предложения «Директор завода «Салют» остановил 25 марта 1982 г. цех № 4, чтобы заменить оборудование» (пример условный)

Запросы к семантической сети определяются зафиксированными в ней бинарными отношениями. Можно запросить образ объекта, действия, предмета, события; можно запросить образ элементов, которые получаются в результате обработки другого запроса, и т. п. Уточнение требуемых отношений производится путем указания в запросе имени связи. Например, «С какой целью остановлен цех № 4», «Какие действия могут выполнять объекты, имеющиеся у предприятия». Если обозначить образ объекта *A* в отношении *r* через *im (A, r),* то эти запросы выглядят как

im (остановил, цель),
im (*im* (предприятие, имеет), может).

Аппарат семантических сетей удачно дополняет аппарат СЕИ СЕИ эффективно описывают структуру документированной информации, а семантические сети рассчитаны на анализ структуры произвольных текстов.

5.5. Вопросы точности координатных и атрибутивных данных

Использование любой информации допустимо, если она удовлетворяет определенным критериям и стандартам. Одним из критериев применимости пространственно-временных данных в геоинформационных системах (ГИС) является ***точность*** - близость результатов, расчетов или оценок к истинным значениям (или значениям, принятым за истинные). Например, точность горизонтали в цифровой базе данных, полученной на основе дигитализации по карте, можно оценить сравнением ее с горизонталью на исходной карте.

Рассмотрим несколько показателей точности в ГИС: **точность вычисления, точность измерения, точность представления.**

Точность вычисления определяется количеством значимых цифр после запятой, *точность измерений* - количеством значимых цифр при измерениях, *точность представления* - количеством разрядов, описывающих координатные данные.

Точность вычислений и измерений не адекватна точности представления. Большое количество значимых цифр не всегда гарантирует точность вычислений или измерений.

Точность вычисления в ГИС велика, обычно она намного выше, чем точность самих данных. Более того, **набор специальных методов и алгоритмов в ряде случаев позволяет повысить точность первичных измерений.**

Точность входит в комплекс данных, определяющий важный показатель - ***качество данных.***

В США разработаны национальные стандарты для цифровых картографических данных, которые применяются при оценке точности цифровых данных. Стандарт выделяет несколько компонентов качества данных:

- позиционную точность;
- точность атрибутов;

- логическую непротиворечивость;
- полноту;
- происхождение.

Позиционная точность выражается степенью отклонения данных ГИС о местоположении от истинного положения объекта на местности. Обычно точность карт приблизительно определяется толщиной линии, или 0,4 мм. Это соответствует 10 м в масштабе 1 : 25 000.

Для проверки позиционной точности используют независимые более точные источники, например карту более крупного масштаба, систему глобального позиционирования (GPS) и др.

Можно на основе известного в статистике правила "переноса ошибок" оценить точность, зная погрешности, вносимые различными источниками. Например, при создании цифровой модели имели место следующие погрешности: 1 мм в исходном материале, 0,4 мм на карте, предназначенной для цифрования, 0,1 мм при цифровании.

При независимой комбинации источников ошибок общую погрешность A можно оценить, суммируя квадраты отдельных погрешностей и извлекая квадратный корень из суммы:

Точность атрибутов определяется близостью значений атрибута к его истинной величине. Атрибуты могут со временем меняться: довольно часто по сравнению с координатными данными.

В зависимости от типов данных точность атрибутов может быть измерена разными способами. Для непрерывных атрибутов (поверхностей), например в полигонах Тиссена, точность выражается как погрешность измерений. Для атрибутов категорий объектов, например классифицированных полигонов, точность зависит от того, являются ли категории подходящими, достаточно подробными и определенными, и от того, какова вероятность наличия в данных грубых ошибок.

Точность атрибута может быть различной в разных частях карты, поэтому полезнее рассчитывать пространственную вариацию

вероятности ошибки в классификации, чем пользоваться обобщенными статистическими показателями.

Понятие *логической непротиворечивости* связано с непротиворечивостью данных в базах данных.

В среде ГИС это понятие распространяется на внутреннюю непротиворечивость структур данных и внутреннюю топологическую непротиворечивость векторных данных. В частности, это определяет такие требования, как замкнутость полигонов, уникальность идентификатора полигона, наличие или отсутствие узлов на пересечениях дуг.

Понятие *полноты* (достаточности) данных связано со степенью охвата данными множества соответствующих объектов. В зависимости от правил отбора, генерализации и масштаба определяют число соответствующих объектов для полного описания ситуации, картографической композиции, явления и т.п. .

Несколько специфический показатель *происхождение* включает сведения об источниках данных и операциях по созданию базы данных, о методах кодирования данных, времени сбора данных, методе обработки данных, точности результатов вычислений и т.п.

5.5.1.Топологическая модель

Основные понятия. Большое количество графических данных в ГИС со специфическими взаимными связями требует топологического описания объектов и групп объектов, которое зависит от "связанности" (простой или сложной). Оно определяет совокупность топологических моделей.

Напомним, что **топологические свойства фигур не изменяются при любых деформациях, производимых без разрывов или соединений.** На рис. 14 представлены топологически родственные фигуры: прямоугольный четырехугольник, замкнутый контур произвольной формы, окружность, треугольник. **Эти объекты (фигуры) имеют одинаковую топологию - одинаковые топологические свойства.** Другим примером топологически родственных фигур могут служить арифметические знаки сложения "+" и умножения "х".

Рис. 14. Топологически родственные фигуры

В геоинформационных системах применение термина *топологический* не такое строгое, как в топологии. В ГИС топологическая модель определяется наличием и хранением совокупностей взаимосвязей, таких, как соединенность дуг на пересечениях, упорядоченный набор звеньев (цепей), образующих границу каждого полигона, взаимосвязи смежности между ареалами и т.п.

В общем смысле слово *топологический* означает, что в модели объекта хранятся взаимосвязи, которые расширяют использование данных ГИС для различных видов пространственного анализа.

Топологическими характеристиками графические модели ГИС существенно отличаются от моделей САПР. Соответственно это различие просматривается в **программно-технологическом обеспечении** этих систем.

Например, вплоть до настоящего времени много разработок ГИС выполняется с использованием средств Автокада, версий от 10 до 13. **Однако в нем не предусмотрены ни работа с покрытиями, ни оверлейные процедуры, ни обработка топологических данных**. Принципиально такие операции в системах CAD (Computer-Aided Desing) возможны, но путем доработки программного обеспечения, что требует достаточно высокой квалификации пользователя и, естественно, ограничивает их круг.

В системах ГИС названные выше процедуры являются встроенными и делают доступным анализ картографической информации широкому кругу пользователей без всякой доработки.

Элементы топологии, входящие в описание моделей данных ГИС, в простейшем случае определяются связями между элементами основных типов координатных данных. Например, в логическую

структуру описания данных могут входить указания о том, какие линии входят в район, в каких точках эти линии пересекаются.

Топологические модели позволяют представлять элементы карты и всю карту в целом в виде графов. Площади, линии и точки описываются границами и узлами (дуговая/узловая структура). Каждая граница идет от начального к конечному узлу, и известно, какие площади находятся слева и справа.

Теоретической основой моделей служат **алгебраическая топология** и **теория графов**. **В соответствии с алгебраической топологией координатные типы данных: площади, линии и точки называются 2-ячейками, 1-ячейками и 0-ячейками соответственно. Карта рассматривается как ориентированный двухмерный ячеечный комплекс.**

Двойственность между теорией графов и алгебраической топологией позволяет применять теоретические положения графов, а также **топологический подход**.

Топологическое векторное представление данных отличается от нетопологического наличием возможности получения исчерпывающего списка взаимоотношений между связанными **геометрическими примитивами** без изменения хранимых координат пространственных объектов.

Необходимая процедура при работе с топологической моделью - подготовка геометрических данных для построения топологии. Этот процесс не может быть полностью автоматизирован уже на данных средней сложности и реализуется только при дополнительных затратах труда (обычно значительных). Таким образом, данные, хранимые в системе, не предусматривающей поддержки топологии, не могут быть надежно преобразованы в топологические данные другой системы чисто автоматическим алгоритмом.

Топологические характеристики должны вычисляться в ходе количественных преобразований моделей объектов ГИС, а затем храниться в базе данных совместно с координатными данными.

Основные топологические характеристики моделей ГИС.

Топологические модели в ГИС задаются совокупностью следующих характеристик:

• связанность векторов - контуры, дороги и прочие векторы должны храниться не как независимые наборы точек, а как взаимосвязанные друг с другом объекты;

• связанность и примыкание районов - информация о взаимном расположении районов и об узлах пересечения районов (рис.15, в);

• пересечение - информация о типах пересечений позволяет воспроизводить мосты и дорожные пересечения (рис.15, *а).* Так Т-образное пересечение (3 линии) является трехвалентным, а Х-образное (4 линии сходятся в точке пересечения) называют четырехвалентным;

• близость - показатель пространственной близости линейных или ареальных объектов (рис.15, б), оценивается числовым параметром, в данном случае символом S.

Топологические характеристики линейных объектов могут быть представлены визуально с помощью связанных графов. Граф сохраняет структуру модели со всеми узлами и пересечениями. Он напоминает карту с искаженным масштабом. Примером такого графа может служить схема метрополитена. Разница между картой метро и схемой метро показывает разницу между картой и графом.

Узлы графа, описывающего картографическую модель, соответствуют пересечениям дорог, местам смыкания дорог с мостами и т.п. Ребра такого графа описывают участки дорог и соединяющие их объекты. В отличие от классической сетевой модели в данной модели длина ребер может не нести информативной нагрузки.

а *б*

в

Рис.15. Основные топологические свойства моделей ГИС: а –

пересечение; б – близость; в - связанность

Топологические характеристики ареальных объектов могут быть представлены с помощью графов покрытия и смежности. Граф покрытия топологически гомоморфен контурной карте соответствующих районов. Ребра такого графа описывают границы между районами, а его узлы (вершины) представляют точки смыкания районов. Степень вершины такого графа - это число районов, которые в ней смыкаются. **Граф смежности это как бы вывернутый наизнанку граф покрытия**. **В нем районы изображаются узлами (вершинами), а пара смыкающихся районов - ребрами**. **На основе такого графа ГИС может выдать ответ на вопрос, является ли проходимой рассматриваемая территория, разделенная на проходимые или непроходимые участки.**

Топологические характеристики сопровождаются **позиционной и описательной информацией**. Вершина графа покрытия может быть дополнена координатными точками, в которых смыкаются соответствующие районы, а ребрам приписывают левосторонние и правосторонние идентификаторы.

После введения точечных объектов при построении линейных и площадных объектов необходимо "создать" топологию. Эти процессы включают вычисление и кодирование связей между точками, линиями и ареалами.

Пересечения и связи имеют векторное представление. Топологические характеристики заносятся при кодировании данных в виде дополнительных атрибутов. Этот процесс осуществляется автоматически во многих ГИС в ходе дигитализации (картографических или фотограмметрических) данных,

Объекты связаны множеством отношений между собой. Это определяет эффективность применения реляционных моделей и баз данных, в основе которых используется понятие *отношения.* В свою очередь, отношения задают множества связей. Простейшие примеры таких связей : "ближайший к ...", "пересекает", "соединен с ...".

Каждому объекту можно присвоить признак, который представляет собой идентификатор ближайшего к нему объекта того же класса; таким образом кодируются связи между парами объектов.

В ГИС часто кодируются два особых типа связей: связи в сетях и связи между полигонами.

Топологические сети состоят из объектов двух типов: линий (звенья, грани, ребра, дуги) и узлов (вершины, пересечения, соединения).

Простейший способ кодирования связей между звеньями и узлами заключается в присвоении каждому звену двух дополнительных атрибутов -идентификаторов узлов на каждом конце (входной узел и выходной узел).

В этом случае при кодировании геометрических данных будут иметь место два типа записей:

1) координаты дуг: *(x1,y1), (x2,y2), ... , (xn,yn);*

2) атрибуты дуг; входной узел, выходной узел, длина, описательные характеристики.

Такая структура позволяет, перемещаясь от звена к звену, определять те из них, у которых перекрываются номера узлов.

Более сложная, но и более совершенная структура имеет список всех звеньев для каждого узла. Это может быть выполнено добавлением к первым двум записи третьего типа;

3) узел: (x, *y),* смежные дуги (со знаком "+" для входного угла и со знаком "-" - для выходного).

Чтобы избежать неудобств, связанных с хранением неодинакового количества идентификаторов дуг, используют два отдельных файла:

1) простой упорядоченный список, в котором файл узлов сжат до ряда идентификаторов дуг;

2) таблицу, в которой для каждого узла хранится информация о положении первой дуги списка.

Используемое в настоящее время математическое обеспечение ГИС почти исключительно основано на топологических моделях, дающих хорошее формализованное представление о пространственных соотношениях между основными объектами карты. Однако, если требуется установить более сложные соотношения, например включение или порядок, нужны дополнительные средства.

5.5.2. Растровые модели

Основы построения. Напомним, что **модель данных представляет собой отображение непрерывных последовательностей реального мира в набор дискретных объектов.**

В растровых моделях дискретизация осуществляется наиболее простым способом - весь объект (исследуемая территория) отображается в пространственные ячейки, образующие регулярную сеть. При этом каждой ячейке растровой модели соответствует одинаковый по размерам, но разный по характеристикам (цвет, плотность) участок поверхности объекта. В ячейке модели содержится одно значение, усредняющее характеристику участка поверхности

объекта. В теории обработки изображений эта процедура известна под названием *пикселимция*.

Если векторная модель дает информацию о том, *где* расположен тот или иной объект, то растровая - информацию о том, *что* расположено в той или иной точке территории. Это определяет основное назначение растровых моделей - непрерывное отображение поверхности.

В растровых моделях в качестве атомарной модели используют двухмерный элемент пространства - пиксель (ячейка). Упорядоченная совокупность атомарных моделей образует растр, который, в свою очередь, является моделью карты или геообъекта.

Векторные модели относятся к бинарным или квазибинарным. Растровые позволяют отображать полутона.

Как правило, каждый элемент растра или каждая ячейка должны иметь лишь одно значение плотности или цвета. Это применимо не для всех случаев. Например, когда граница двух типов покрытий может проходить через центр элемента растра, элементу дается значение, характе-ризующее большую часть ячейки или ее центральную точку. Ряд систем позволяет иметь несколько значений для одного элемента растра.

Характеристики растровых моделей. Для растровых моделей существует ряд характеристик: **разрешение, значение, ориентация, зоны, пожение.**

Разрешение - минимальный линейный размер наименьшего участка пространства (поверхности), отображаемый одним пикселем.

Пиксели обычно представляют собой прямоугольники или квадраты, реже используются треугольники и шестиугольники. Более высоким разрешением обладает растр с меньшим размером ячеек. Высокое разрешение подразумевает обилие деталей, множество ячеек, минимальный размер ячеек.

Значение - элемент информации, хранящийся в элементе растра (пикселе). Поскольку при обработке применяют типизированные дан-

ные, то есть необходимость определить типы значений растровой модели.

Тип значений в ячейках растра определяется как реальным явлением так и особенностями ГИС. В частности, в разных системах можно использовать разные классы значений: целые числа, действительные (десятичные) значения, буквенные значения.

Целые числа могут служить характеристиками оптической плотности или кодами, указывающими на позицию в прилагаемой таблице или легенде. Например, возможна следующая легенда, указывающая наименование класса почв: 0 - пустой класс, 1 - суглинистые, 2 - песчаные, 3 - щебнистые и т.п.

Ориентация - угол между направлением на север и положением колонок растра.

Зона растровой модели включает соседствующие друг с другом ячейки, имеющие одинаковое значение. Зоной могут быть отдельные объекты, природные явления, ареалы типов почв, элементы гидрографии и т.п.

Для указания всех зон с одним и тем же значением используют понятие *класс зон.* Естественно, что не во всех слоях изображения могут присутствовать зоны. Основные характеристики зоны - ее значение и положение.

Буферная зона - зона, границы которой удалены на известное расстояние от любого объекта на карте. Буферные зоны различной ширины могут быть созданы вокруг выбранных объектов на базе таблиц сопряженных характеристик.

Положение обычно задается упорядоченной парой координат (номер строки и номер столбца), которые однозначно определяют положение каждого элемента отображаемого пространства в растре.

Проводя сравнение векторных и растровых моделей, отметим удобство векторных для организации и работы со взаимосвязями объектов. Тем не менее, используя простые приемы, например включая взаимосвязи в

таблицы атрибутов, можно организовать взаимосвязи и в растровых системах.

Необходимо остановиться на вопросах точности отображения в растровых моделях. В растровых форматах в большинстве случаев неясно, относятся координаты к центральной точке пикселя или к одному из его углов. Поэтому точность привязки элемента растра определяют как 1/2 ширины и высоты ячейки.

Растровые модели имеют следующие достоинства:

• растр не требует предварительного знакомства с явлениями, данные собираются с равномерно расположенной сети точек, что позволяет в дальнейшем на основе статистических методов обработки получать объективные характеристики исследуемых объектов. Благодаря этому растровые модели могут использоваться для изучения новых явлений, о которых не накоплен материал. В силу простоты этот способ получил наибольшее распространение;

• растровые данные проще для обработки по параллельным алгоритмам и этим обеспечивают более высокое быстродействие по сравнению с векторными;

• некоторые задачи, например создание буферной зоны, много проще решать в растровом виде;

• многие растровые модели позволяют вводить векторные данные, в то время как обратная процедура весьма затруднительна для векторных моделей;

• процессы растеризации много проще алгоритмически, чем процессы векторизации, которые зачастую требуют экспертных решений.

Наиболее часто растровые модели применяют при обработке аэрокосмических снимков для получения данных дистанционных исследований Земли.

Метод группового кодирования. Самый простой способ ввода растровых моделей - прямой ввод одной ячейки за другой. Недостатками данного подхода являются требования большого объема памяти в

компьютере и значительного времени для организации процедур ввода-вывода. Например, снимок искусственного спутника Земли (ИСЗ) Landsat имеет 74 000 000 элементов растра и это требует огромных ресурсов для хранения данных.

При растровом вводе информации в ГИС возникает проблема ее сжатия, так как наряду с полезной может попадать и избыточная (в том числе и бесполезная) информация. **Для сжатия информации, полученной со снимка или карты, применяется кодирование участков развертки или метод группового кодирования**, учитывающий, что довольно часто в нескольких ячейках значения повторяются.

Суть метода группового кодирования состоит в том, что данные вводятся парой чисел, первое обозначает длину группы, второе - значение. Изображение просматривается построчно, и как только определенный тип элемента или ячейки встречается впервые, он помечается **признаком начала**. Если за данной ячейкой следует цепочка ячеек того же типа, то их число подсчитывается, а последняя ячейка помечается **признаком конца**. В этом случае в памяти хранятся только позиции помеченных ячеек и значения соответствующих счетчиков.

Применение такого метода значительно упрощает хранение и воспроизведение изображений (карт), когда однородные участки (как правило) превосходят размеры одной ячейки.

Обычно ввод осуществляют слева направо, сверху вниз. Рассмотрим, например, бинарный массив матрицы (5 х 6) :

000111 001110 001110 011111 011111.

При использовании метода группового кодирования он будет вводиться как:

30312031303120511051.

Вместо 30 необходимо только 20 элементов данных. В рассмотренном примере экономия составляет 30 %, однако на практике при работе с большими массивами бинарных данных она бывает гораздо больше.

Метод группового кодирования имеет ограничения и может использоваться далеко не во всех ГИС.

Элементы бинарной матрицы, т.е. растровой модели, могут принимать только два значения: "1" или "0". Эта матрица соответствует черно-белому изображению. На практике возможно полутоновое или цветное изображение. В этих случаях значения в ячейках растровой модели могут различаться по типам. **Тип значений в ячейках растра определяется как исходными данными, так и особенностями программных средств ГИС**. В качестве значений растровых данных могут быть применены целые числа, действительные (десятичные) значения, буквенные значения.

В одних системах используются только целые числа, в других - различные типы данных. При этом ставится условие единства значений для отдельных растровых слоев. Целые числа часто служат кодами, указывающими на позицию в прилагаемой таблице или легенде.

Структурно определенные растровые модели. Растровые модели делятся на регулярные, нерегулярные и вложенные (рекурсивные или иерархические) мозаики (рис.16).

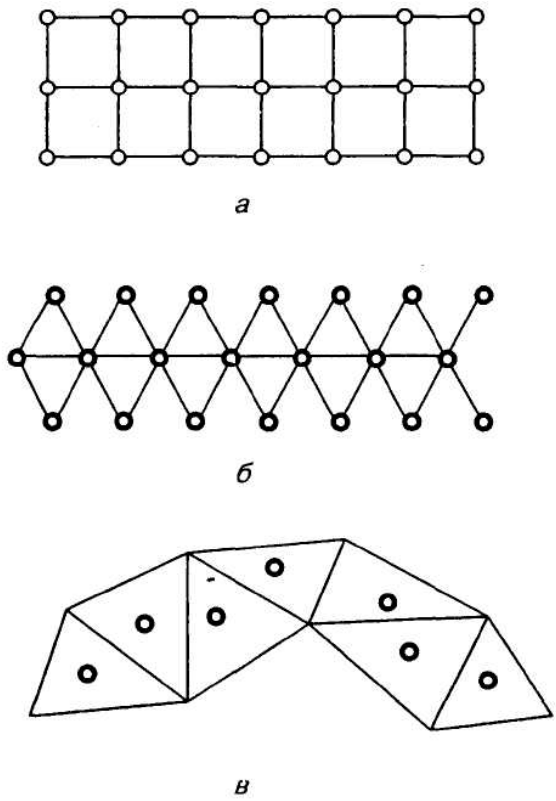


Рис.16. Растровые модели: а – регулярная прямоугольная решетка; б – регулярная треугольная решетка; в – полигоны Тиссена

Плоские регулярные мозаики бывают трех типов: квадрат (рис.16, а), треугольник и шестиугольник (рис.16, б). Квадрат - самая удобная модель, так как позволяет относительно просто проводить обработку больших массивов данных. Треугольные мозаики служат хорошей основой для создания выпуклых (сферических) покрытий.

Среди нерегулярных мозаик чаще всего используют треугольные сети неправильной формы (Triangulated Irregular Network - TIN) и полигоны Тиссена (рис.16, в). Сети TIN удобны для создания цифровых моделей отметок местности по заданному набору точек. Они применяются как в растровых, так и в векторных моделях.

Модель треугольной нерегулярной сети (TIN) в значительной мере альтернативна цифровой модели рельефа, построенной на регулярной сети. TIN-модель была разработана в начале 70-х гг. как простой способ построения поверхностей на основе набора неравномерно расположенных точек. В 70-е гг. было создано несколько вариантов данной системы, коммерческие системы на базе TIN стали появляться в 80-х гг. как пакеты программ для построения горизонталей.

Модель TIN используется для цифрового моделирования рельефа. При этом узлам и ребрам треугольной сети соотносятся исходные и производные атрибуты цифровой модели.

Полигоны Тиссена (или диаграммы Вороного) представляют собой геометрические конструкции, образуемые относительно множества точек таким образом, что границы полигонов являются отрезками перпендикуляров, восстанавливаемых к линиям, соединяющим две ближайшие точки. Полигоны Тиссена позволяют проводить анализ на соседство, близость и достижимость.

Нерегулярная выборка лучше, чем регулярная, отражает характер реальной поверхности и это является достоинством полигонов Тиссена.

При построении TIN-модели дискретно расположенные точки соединяются линиями, образующими треугольники. В пределах каждого треугольника поверхность обычно представляется плоскостью. Поскольку поверхность каждого треугольника задается высотами трех его вершин, применение треугольников обеспечивает каждому участку мозаичной поверхности точное прилегание к смежным участкам. Это

обеспечивает непрерывность поверхности при нерегулярном расположении точек.

Данная модель позволяет использовать в качестве элементов мозаики более сложные многоугольники, но их всегда можно разбить на треугольники.

В векторных ГИС модель TIN можно рассматривать как полигоны с атрибутами угла наклона, экспозиции и площади, с тремя вершинами, имеющими атрибуты высоты, и с тремя сторонами, характеризующимися углом наклона и направлением.

Для выбора точек модели используют три основных алгоритма: алгоритм Фоулера и Литла, алгоритм ключевых точек, эвристическое удаление точек.

С аналитической точки зрения основу таких вложенных, или иерархических, мозаик составляют (рекурсивно) раскладываемые модели. Рекурсивная декомпозиция треугольников приводит к образованию треугольных **квадродеревьев**, причем декомпозиция шестиугольников невозможна. Единицы с более высоким уровнем разрешающей способности можно объединять, формируя шестиугольники, что приводит к образованию **семиразрядного дерева**. Схема адресации для вложенных шестиугольных мозаик была разработана Л. Гибсоном и Д. Лукасом. Они назвали ее **генерализованной сбалансированной троичной мозаикой**.

Квадратомическое дерево - одна из наиболее широко известных структур данных, использующихся применительно к **площадям, линиям и точкам**.

Бесструктурные гиперграфовые и решетчатые модели. Они обрабатывают координатные данные в виде простых строк координат без какой-либо структуры. В случае обработки площадей общие границы всегда вводятся в ЭВМ дважды. Пример практического применения этих моделей - хранимые в памяти ЭВМ полные полигоны и векторные цепные коды.

Гиперграфовые модели основаны на теории множеств и гиперграфов и используют шесть абстрактных типов данных: **класс, атрибут класса, связь класса, объект, атрибут объекта, связь объекта.**

Класс соответствует границе гиперграфа, причем объекты являются узлами этого графа. Каждый класс содержит объекты с атрибутами объекта и различаемый узел, содержащий атрибут класса. Используя подклассы, вводят иерархию классов и объектов.

Связи классов и связи объектов устанавливают соотношения между теми классами, которые не связаны иерархически. Связи классов представляют потенциальные соотношения между классами, а связи объектов - действительные соотношения между объектами. Для образования мультисвязи можно объединить несколько связей объектов. Несколько классов объектов образуют гиперклассы, которые связаны гиперсвязями.

Гиперграфовые модели применимы как к координатным, так и к атрибутивным данным. Как правило, они отличаются высокой степенью сложности.

Решетчатые модели базируются на математической теории решеток, оперирующей с частично упорядоченными наборами данных. Они полезны в тех случаях, когда отсутствует четкая иерархия объектов.

Элементы алгебраической теории автоматных моделей синтеза типовых конструктивных моделей упрощают процесс получения сложных графических изображений. Однако такой подход, находящий широкое применение в САПР, пока не используется в технологиях ГИС.

5.5.3. Оверлейные структуры

Цифровая карта может быть организована как множество слоев (покрытий или карт-подложек). Концепция послойного представления графической информации заимствована из систем CAD, однако в ГИС она получила качественно новое развитие.

Принципиальное отличие состоит в том, что слои в ГИС могут быть как векторными, так и растровыми, причем векторные слои обязатель-

но должны иметь одну из трех характеристик векторных данных, т.е. **векторный слой должен быть определен как точечный, линейный или полигональный** дополнительно к его тематической направленности.

Другое важное отличие послойного представления геоинформационных векторных данных заключается в том, что они являются **объектными, т.е. несут информацию об объектах, а не об отдельных элементах объекта, как в САПР.**

Слои в ГИС являются типом цифровых картографических моделей, которые построены на основе объединения (типизации) пространственных объектов (или набора данных), имеющих общие свойства или функциональные признаки. Такими свойствами могут быть: принадлежность к одному типу координатных объектов (точечные, линейные полигональные); принадлежность к одному типу пространственных объектов (жилые здания, подземные коммуникации, административные границы и т.д.); отображение на карте одним цветом.

В качестве отдельных слоев можно объединять данные, полученные в результате сбора первичной информации.

Совокупность слоев образует интегрированную основу графической части ГИС (рис.17).

Рис. 17. Пример слоев интегрированной ГИС

Принадлежность объекта или части объекта к слою позволяет использовать и добавлять групповые свойства объектам данного слоя. А как известно из теории обработки данных, именно их групповая обработка является основой

повышения производительности автоматизированных систем. Слои могут иметь как векторные, так и растровые форматы. Однако многие ГИС допускают возможность работы со слоями только векторного типа, а растр используется в качестве подложки. В связи с этим следует отметить возможности системы ER Mapper трансформировать растровое изображение снимка в заданную картографическую проекцию.

Данные, размещенные на слоях, могут обрабатываться как в интерактивном, так и в автоматическом режиме. С помощью системы фильтров или заданных параметров объекты, принадлежащие слою, могут быть одновременно масштабированы, перемещены, скопированы, записаны в базу данных. В других случаях (при установке других режимов) можно наложить запрет на редактирование объектов слоя, запретить их просмотр или сделать невидимыми.

Многослойная организация электронной карты при наличии гибкого механизма управления слоями позволяет объединить и отобразить не только большее количество информации, чем на обычной карте, но существенно упростить анализ картографических данных с помощью селекции данных, необходимых для визуализации и механизма "прозрачности" цифровой карты.

Таким образом, разбиение на слои позволяет решать задачи типизации и разбиения данных на типы, повышать эффективность интерактивной обработки и групповой автоматизированной обработки, упрощать процесс хранения информации в базах данных, включать автоматизированные методы пространственного анализа на стадии сбора данных и при моделировании, упрощать решение экспертных задач.

5.5.4. Трехмерные модели

Большинство ГИС хранят информацию о точках местности в виде трехмерных координат. Однако для многих приложений ГИС, таких, как построение карт, трехмерные координаты преобразуют в двухмерное представление, т.е. строят двухмерные (2D) модели.

Со второй половины 90-х гг. заметна тенденция к построению трехмерных (3D) моделей. С одной стороны, это продиктовано решением практических задач, с другой - увеличением мощности вычислительных ресурсов, что необходимо для трехмерного

моделирования. Такая модель должна соответствовать отображению трехмерной реальности, по возможности близкой к той, что видит человеческий глаз на местности.

В настоящее время существуют два основных способа представления трехмерных моделей в ГИС.

Первый способ, назовем его *псевдотрехмерным,* основан на том, что создается структура данных, в которых значение третьей координаты Z (обычно высота) каждой точки *(X, Y)* записывается в качестве атрибута. При этом значение Z может быть использовано в перспективных построениях для создания изображений трехмерных представлений. Поскольку это не истинное трехмерное представление, его часто именуют 2,5-мерным (два-с-половиной-мерным).

Такие 2,5-мерные модели дают возможность эффективного решения ряда задач:

• представление рельефа и других непрерывных поверхностей на базе ЦМР или TIN;

• расчет перспективной модели для любой задаваемой точки обзора;

• "натяжение" дополнительных слоев на поверхность с использованием цвета и световых эффектов;

• визуальное преобразование одних классов данных в другие (например, объемный слой промышленных выбросов преобразовать в изображение экологической карты и результирующей карты действия на окружающую растительность);

• создание динамической модели "полета" над территорией. Второй способ - создание *истинных трехмерных представлений* - структур данных, в которых местоположение фиксируется в трех измерениях *(X, У, Z). В* этом случае Z- не атрибут, а элемент местоположения точки. Такой подход позволяет регистрировать данные в нескольких точках с одинаковыми координатами *X* и *У,* например, при зондировании атмосферы или при определении объемов горных выработок.

Истинные трехмерные представления позволяют:

• наглядно изображать (визуализировать) объемы;

• решать задачи, связанные с моделированием объемов;

• решать новый класс задач - разработка трехмерных ГИС;

• производить синтез трехмерных структур. Оба способа трехмерных представлений пространственной информации имеют несколько важных приложений:

• проектирование инженерных и промышленных сооружений (шахты, карьеры, плотины, водохранилища);

• моделирование геологических процессов;

• моделирование трехмерных потоков в газообразных и жидкостных средах.

В ГИС наряду с цифровыми моделями местности, которые, как правило, отражают статические свойства, широко используются динамические модели, например модель явления.

Трехмерные явления характеризуются несколькими свойствами: распределение, геометрическая сложность, топологическая сложность, точность измерения, точность представления.

Распределение может быть непрерывное (например, поле поверхности) и дискретное (например, рудные тела).

Топологическая сложность обусловливается связями внутри объекта. Например, составной объект состоит из таких же, но более мелких объектов одного класса. Смешанный объект включает несколько классов и состоит из более мелких неоднородных объектов.

Геометрическая сложность зависит от типов кривых и геометрических конструкций.

Точность представления определяет допуски при проектировании, изысканиях, научных исследованиях.

Точность измерения выражается допусками и погрешностью средств измерения.

Применение трехмерных моделей позволяет строить новые модели и расширяет возможности ГИС как системы принятия решений. С использованием методов трехмерной графики можно по-новому решать задачи проектирования жилой застройки, размещения объектов бытового и хозяйственного назначения в муниципальных округах, создавать новые типы трехмерных условных знаков и т.д.

Примером подобной разработки может служить ГИС Star Informatic для решения задач городского планирования и задач урбанизации, разработанная специалистами из Бельгии и Великобритании (фирма Star).

Выводы

Данные в ГИС обладают своей спецификой и не имеют прямых аналогов в других автоматизированных системах. Они имеют множество форматов (практически каждая ГИС - свой) и разные формы представления.

Информационная основа ГИС содержит типизированные и нетипизированные записи, а также графические данные с двумя основными формами представления ***- векторной и растровой.*** *Растровые и векторные модели имеют свои преимущества при решении разных задач и дополняют друг друга в системе комплексной обработки данных ГИС.* ***Векторные данные разделяются на три основных типа: точечные, линейные и полигональные****. Каждый тип характеризуется своими методами обработки.*

Остается нерешенной проблема автоматизированного преобразования растровых моделей в векторные.

Интеграция данных в ГИС позволяет решать задачи проекционных преобразований и объемного представления трехмерных объектов, включая их динамическую визуализацию.

6. Методы организации данных

6.1. Линейная организация данных

Метод организации значений данных в памяти ЭВМ представляет собой аппарат для фиксации однородного бинарного отношения $R \subset D \times D$, где D представляет собой множество информационных единиц — **записей**. *Записью* называется одно значение СЕИ, выбранное из некоторого множества значений, объединенных общим именем. Множество D обычно содержит все значения СЕИ с заданным именем, которые хранятся в данный момент времени в памяти.
Отношение *R* описывает связь данной записи с последующей записью либо с последующей и предыдущей. Отношение R обладает антирефлексивностью, поскольку практически никогда не требуется поддерживать связь каждой записи с самой собой. В литературе организация значений данных часто называется структурой данных.
Организация значений данных (далее называемая просто организацией данных) может быть линейной и нелинейной. При *линейной организации данных* каждая запись, кроме первой и последней, связана с одной предыдущей и одной последующей записью. У записей, соответствующих нелинейной организации данных, количество предыдущих и последующих им записей не ограничено.
Линейные методы организации данных различаются только способами указания предыдущей и последующей записей по отношению к данной записи. Но это приводит к тому, что алгоритмы, эффективные для одних методов организации данных, становятся неприемлемыми для других методов. Поэтому обоснованный выбор организации является нетривиальной задачей
Среди линейных методов выделяются последовательная и строчная организация данных. При *последовательной организации данных* записи располагаются в памяти строго одна за другой согласно заданному логическому порядку. Логический порядок записей определяется последовательностью вызова их на обработку.
Последовательная организация данных соответствует, как правило, понятию **«массив» («файл»).**
Записи массива могут быть упорядоченными или неупорядоченными по значениям ключевого реквизита (ключа), имя которого одинаково во всех записях. **Ключевой реквизит обычно является реквизитом-признаком**. Часто требуется поддерживать упорядоченность записей по нескольким именам ключевых признаков. В этом случае среди **признаков устанавливается старшинство** и упорядоченность

записей определяется как лексикографическая. Понятие «ключ» в данном контексте отличается от ранее введенного понятия «вероятный ключ» тем, что не требуется однозначная идентификация записей массива (по значению ключа получаем доступ к подмножеству записей).

Записи, составляющие массив с точки зрения способа указания их длины, делятся на записи **фиксированной, переменной и неопределенной длины.** Записи фиксированной (постоянной) длины имеют одинаковую, заранее известную длину. Если длины записей неодинаковы, они указываются в самой записи. Такие записи называют записями переменной длины. Вместо явного указания длины записи можно отмечать окончание записи специальным символом — **разделителем**, который не должен встречаться среди информационных символов значения записи. Записи, заканчивающиеся разделителем, называются записями неопределенной длины.

Адреса промежуточных записей фиксированной длины в массиве задаются формулой

$A_i = A_1 + (i — 1)\, l,$

где A_i — начальный адрес первой записи;

A_i — начальный адрес i-й записи;

l — длина одной записи.

Для массива записей переменной и неопределенной длины подобной простой формулы не существует. Массив записей нефиксированной длины, как правило, занимает меньший объем памяти. Их обработка, однако, ведется с меньшей скоростью, поскольку затруднено обнаружение начала следующей в логическом порядке записи.

Основные операции над значениями данных: формирование, поиск и корректировка.

Данные обычно возникают в неупорядоченной форме. Перед обработкой, как правило, целесообразно отсортировать их значения по ключевым реквизитам, что составляет одну из основных работ по *формированию* (подготовке) данных.

Упорядоченные данные эффективны для организации быстрого поиска информации. Выходные документы, выводимые на печать, полученные на основе отсортированных данных, удобны для дальнейшего использования человеком. Многие алгоритмы задач управления вообще рассчитаны на использование только упорядоченных данных. Отсортированные данные позволяют организовать быструю обработку нескольких массивов. Далее будем считать все массивы упорядоченными по возрастанию значений одного реквизита, когда для ключа i-й записи p_i справедливо условие $p_i \le p_{i+1}$.

Поиском называется процедура выделения из некоторого множества записей определенного подмножества, записи которого удовлетворяют некоторому заранее поставленному условию. Условия поиска часто называют запросом на поиск, или **поисковым признаком**.
Простейшим поисковым признаком может служить значение ключевого реквизита (**поиск по совпадению**).
Алгоритмы всех разновидностей поиска можно получить из алгоритмов поиска по совпадению, которые рассматриваются в дальнейшем.
Базовым методом доступа к массиву является **ступенчатый поиск**. Этот метод предполагает упорядоченность обрабатываемых данных. Рассмотрим сначала двухступенчатый поиск в массиве, состоящем из M записей. Для определенности будем считать, что массив отсортирован по возрастанию значений ключевого реквизита p_i, где $i = 1,2,\ldots, M$. Для заданного M выбирается константа $d_1 < M$, называемая **шагом поиска**. Если необходимо отыскать запись, ключевой реквизит которой равен величине q, производятся следующие действия. Значение q последовательно сравнивается с рядом ключей $p_1, p_1+d_1, p_1+2d_1, \ldots, p_1+kd_1, \ldots$ до тех пор, пока будет впервые достигнуто неравенство $p_1+md_1 \geqslant q$. Здесь заканчивается первый этап (первая ступень) поиска. На втором этапе сравнения производятся между q и ключевыми признаками

$$p_1+(m-1)d_1, \quad p_2+(m-1)d_1, \quad \ldots, \quad p_l+(m-1)d_1, \ldots$$

до тех пор, пока в процессе сравнений не будут извлеченывсе требуемые записи с ключом либо будет достигнут ключ, больший, чем q. В последнем случае поиск заканчивается безрезультатно.
Эффективность поиска измеряется количеством произведенных сравнений. Для двухступенчатого поиска среднее число сравнений примерно составит

$$C = \frac{M}{2d_1} + \frac{d_1}{2}.$$

Параметр d_1 — выбираемый, и естественно выбрать d_1 так, чтобы оно минимизировало С. Заменим d_1непрерывной переменной x и вычислим производную

$$C' = -\frac{M}{2x^2} + \frac{1}{2}.$$

Из условия $C = 0$ получаем $d_1 \approx \sqrt{M}$. Вторая производная C'' в точке $x=\sqrt{M}$ положительна, следовательно, достигается минимальное значение C.
При n-ступенчатом поиске заранее выбираются константы n и S. На первом этапе ключевые реквизиты для сравнения с искомым ключом q

выбираются из массива по закону арифметической прогрессии, начиная с p_1 с шагом $d_1 = \lfloor M/S \rfloor$, где скобки $\lfloor \rfloor$ означают округление числа до целого в меньшую сторону. Когда будет достигнут ключ $p_1 + kd_1 \geqslant q$, выбирается шаг $d_2 = \lfloor d_1/S \rfloor$ и организуются сравнения с этим шагом, начиная с ключа $p_1 + (k-1)d_1$. Описанные действия повторяются n раз, причем для $t < n$ $d_t = \lfloor d_{t-1}/S \rfloor$ и $d_n = 1$.

Подобно тому как в двухступенчатом поиске лучшее значение $S = M/d_1 = \sqrt{M}$, в общем случае лучшее значение $S = M^{1/n}$ и, кроме того, существует оптимальное n. Среднее число сравнений составляет

$$C = 0{,}5 \cdot n \cdot M^{1/n}.$$

Приравняв к нулю производную C по n, получаем $n \approx \ln M$. Ступенчатый поиск имеет два важных частных варианта — последовательный поиск, когда $n = 1$ и $d_1 = 1$, и бинарный (дихотомический) поиск, когда $S = 2$. Число сравнений при последовательном поиске в упорядоченном массиве и равновероятном местонахождении искомой записи определяется выражением

$$C = \sum_{i=1}^{M} r_i \cdot i = \frac{1}{M} \sum_{i=1}^{M} i = \frac{M+1}{2}.$$

Здесь $r_i = 1/M$ — вероятность извлечения i-й записи при поиске.

Для бинарного поиска вводится левая граница интервала A и правая граница интервала B. Первоначально интервал охватывает весь массив, т. е. $A = 0$, $B = M + 1$. Вычисляется серелина интервала i по формуле $i = (A+B)/2$, с округлением в меньшую сторону. Ключ i-й записи p_i сравнивается с искомым значением q. Если $p_i = q$, то ключи соседних записей проверяются на равенство их ключа с q, все такие записи выводятся, и поиск заканчивается. В случае $p_i > q$ записи с номерами $i+1, i + 2, \ldots, M$ заведомо не содержат значение q и надо сократить интервал поиска, приняв $B = i$. Аналогично при $p_i < q$ надо взять $A = i$. Далее середина интервала вычисляется заново, и все действия повторяются. Если будет достигнут нулевой интервал, то требуемой записи в массиве нет. Среднее число сравнений при бинарном поиске

$$C = \log_2 M - 1.$$

Все рассмотренные выше методы поиска основывались на проведении сравнений пар ключевых признаков. Принципиально иную возможность поиска представляет собой вычисление номера записи с искомым ключом q. Соответствующий метод поиска в упорядоченном массиве называется **интерполяционным.** Зависимость ключевого

признака p_i от номера записи i в упорядоченном массиве является монотонно возрастающей функцией (рис. 1).

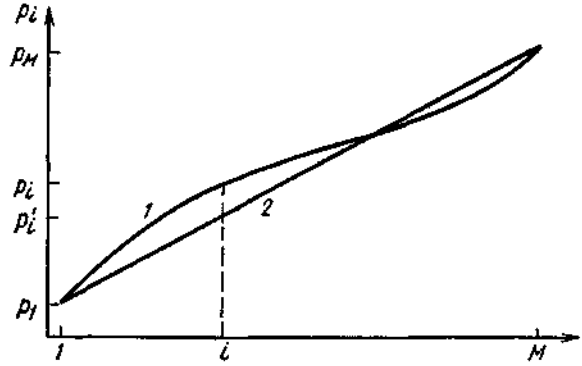


Рис. 1. График зависимости значений p_i от номера записи i в упорядоченном массиве (1) и его линейная интерполяция (2)

Значения p_i для $1 < i < M$ будем приближенно интерполировать с помощью линейной функции, которую построим по двум точкам $(1, p_i)$ и $(M, p_м)$. Для произвольной точки p'_i, которая является приближенным значением для p_i, в соответствии с линейной интерполяцией справедливо выражение

$$\frac{p_M - p_1}{p'_i - p_1} = \frac{M-1}{i-1}.$$

При вычислении номера записи с ключевым признаком q значение q подставляется вместо p'_i и вычисляется соответствующий номер записи

$$i = 1 + \frac{q - p_1}{p_M - p_1}(M-1),$$

который округляется до ближайшего целого значения. Затем из массива извлекается i-я запись с ключом p_i. Поскольку используется приближенная формула, возможны следующие соотношения $p_i = q$, $p_i > q$, $p_i < q$. Поэтому после интерполяционного поиска требуется провести последовательный поиск значения q, начиная с i-й записи.

Корректировка значений массива может касаться одной его записи или группы записей. В последнем случае она назьвается групповой. Также рассматривается включение новой записи, исключение ненужной записи, замена значений реквизитов в записи. В любом случае перед

непосредственно корректировкой выполняется поиск местонахождения корректируемой записи.
Массив записей, подвергающихся изменениям, называется основным. Любое отдельное изменение данных основного массива касается одной его записи.
Изменения, которые необходимо внести в основной массив, накапливаются в специальном упорядоченном массиве изменений, рассчитанном на $1 \leq^{\wedge} M_0 \leq M$ записей. Обычно M_0 составляет несколько процентов от M. При необходимости обработки основного массива он объединяется с массивом изменений.
При объединении основного массива с массивом изменений выполняются следующие операции:

ключ очередной записи основного массива сравнивается с ключом очередной записи массива изменений, и запись с меньшим значением ключа переписывается на другой участок памяти, в котором формируется новое состояние основного массива (результат объединения);

когда все записи одного из массивов будут перезаписаны, оставшиеся записи другого массива перезаписываются без сравнений и объединение заканчивается;

если ключи записей при сравнении совпадут и запись массива изменений помечена как исключаемая, то ни одна из записей не записывается в результат объединения. При равенстве значений ключей происходит и замена значений реквизитов в записи основного массива.

Один из методов, применяемых для ускорения доступа к данным, первоначально хранящимся в последовательной форме, состоит в образовании на его основе вспомогательного инвертированного массива данных.
Инвертированный массив позволяет по известному значению ключевого признака определить адреса всех записей, которые его содержат.
Элементы, составляющие инвертированный массив, называются группами. Число групп равно числу различных ключевых реквизитов в записях исходного массива. Каждая группа содержит ключевой реквизит, имеющийся в массиве, в качестве своего собственного ключа и набор адресов записей, содержащих в своем составе этот ключ. Адреса записей в группе должны быть упорядочены. Массив групп, упорядоченный по значениям ключа, и есть инвертированный массив. Группы имеют неопределенную длину. Структуры основного массива и инвертированного массива показаны на рис. 2.

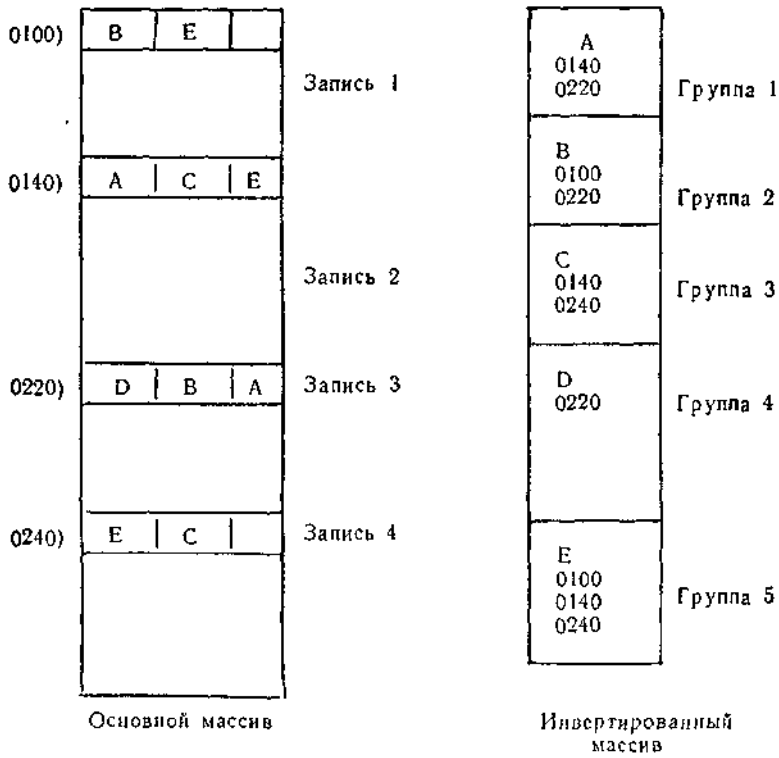


Рис. 2. Основной массив записей и его инвертированный массив

Прописными латинскими буквами обозначены значения ключа. Структура варианта инвертированного массива показана также в табл. 1.
Таблица 1

Основной массив

Адрес записи	Имена ключей		Неключевые реквизиты
	A	B	
	значения ключей		
400	a_1	e_1	$x_1\ y_1$
500	a_2	e_1	$x_2\ y_2$
600	a_2	e_2	$x_3\ y_3$
700	a_3	e_1	$x_4\ y_4$
800	a_2	e_3	$x_5\ y_5$
900	a_1	e_3	$x_6\ y_6$

Инвертированный массив для ключа 1

a_1 400 900	Группа 1
a_2 500 600 800	Группа 2
a_3 700	Группа 3

Инвертированный массив для ключа *B*

e_1 400 500 700	Группа 1
e_2 600	Группа 2
e_3 800 900	Группа 3

Основной эффект инвертированного массива проявляется при поиске данных по нескольким условиям. Пусть дан запрос «найти все записи, содержащие ключи A и C». Система обратится к инвертированному массиву и найдет группы ключей *A* и *C*. Совпадающие значения адресов в *A* и *C* укажут в нашем примере на искомую запись с адресом 0140. Этот пример поясняет и общее правило. Таким же способом, например, может быть установлено отсутствие записей, удовлетворяющих запросу «найти все записи, содержащие ключи *B* и C».

Логические связки в запросах могут быть любыми, и с математической точки зрения требуемые поисковые операции есть операции пересечения, объединения, вычитания и т. п. под множеством адресов, которые хранятся в группах, названных в запросе.

Так, при обработке запроса «найти все записи, содержащие ключи *E* или C, кроме A», которому в терминах теории множеств соответствует запись $(E \cup C) \backslash A$, производится последовательное вычисление

$$(\{0100,\ 0140,\ 0240\} \cup \{0140,\ 0240\}) \setminus \{0140,\ 0220\} = \{0100,\ 0240\}.$$

Результат означает, что запросу удовлетворяют записи с адресами 0100 и 0240.

Следует отметить, что поиск по инвертированному массиву обнаруживает только адреса записей и плохо приспособлен для указания всех ключей, связанных с найденной записью. Между тем эта информация часто запрашивается. В одном из наших примеров запись с адресом 0140 была найдена по значениям ключей *A* и *C* очень быстро, но определить, есть ли в этой записи третий ключ *E,* используя только инвертированный массив, невозможно.

Корректировка основного массива вызывает очевидные изменения в инвертированном массиве. Алгоритмы внесения изменений в обоих случаях одинаковы. Как правило, одно изменение в основном массиве (включение, исключение) приводит к корректировке нескольких групп в инвертированном массиве.

Отсюда можно сделать вывод, что корректировка данных в инвертированном формации в строке массиве является достаточно трудоемким процессом.

К достоинствам инвертированного массива следует отнести широкий диапазон запросов, которые могут быть обработаны по единому алгоритму, а также небольшой его объем, что позволяет быстро извлекать данные при поиске.

Решение целого ряда задач обработки данных требует применения таких методов организации данных, которые позволили бы связать физически разнесенные в памяти данные в логическую последовательность. Простейшим методом, применяемым для этих целей, является **строчная организация данных**.

Строчная организация — это линейная организация данных, в которой логическая последовательность записей обеспечивается с помощью специальных указателей (адресов связи). В адресе связи указывается адрес хранения записи, следующей за данной записью в логической последовательности. Для строчной организации данных используются понятия-синонимы «строка» и «цепь». В строке выделяются собственная информация, т. е. записи, используемые в прикладных расчетах, и ассоциативная информация — все адреса связи.

Возможны два способа хранения строки — совместное размещение собственной и ассоциативной информации, когда запись и ее адрес связи располагаются в памяти вплотную (рис. 3, *а),* и раздельное (рис. 3, *б).*

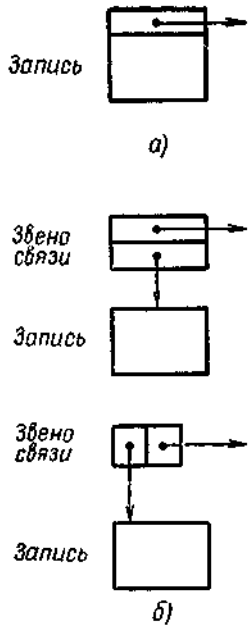


Рис .3. Совместное (а) и раздельное (б) хранение ассоциативной и собственной информации в строке .

В последнем случае ассоциативная информация, относящаяся к каждой записи, образует звено связи. На графических иллюстрациях адрес СЕЯЗИ изображается прямоугольником со стрелкой; стрелка указывает на элемент данных, адрес хранения которого содержится в адресе связи.

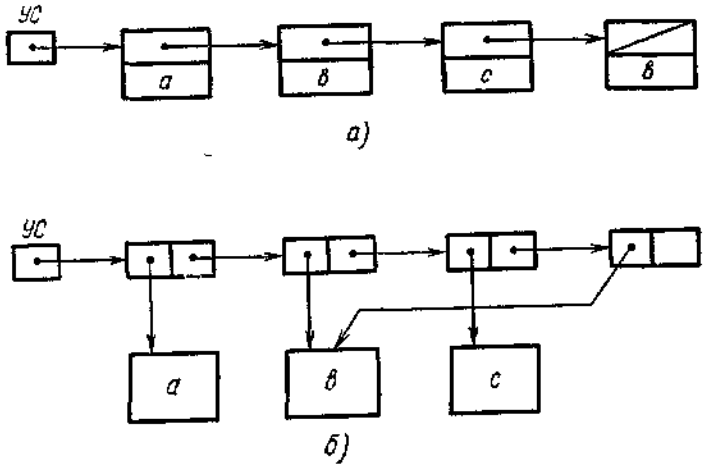


Рис. 4. Однонаправленная строка с совместным (а) и раздельным *(б)* хранением ассоциативной и собственной информации

Раздельное размещение собственной и ассоциативной информации оправдано, если имеется возможность разместить звенья связи на запоминающем устройстве с более высоким быстродействием.

Основные возможности строчной организации данных обеспечивает так называемая однонаправленная строка (или прямая), в которой реализуется взаимосвязь между записями типа «следующий». Каждая запись однонаправленной строки должна иметь адрес связи, содержащий адрес расположения следующей записи. Если следующая запись отсутствует, то вместо адреса связи в указатель помещается специальный знак, который называется признаком конца списка, или терминатором, и условно обозначается КС или $\varnothing$. На графических иллюстрациях адрес связи, соответствующий концу строки, перечеркивается.

Строка начинается с указателя строки (УС), в котором для однонаправленной строки адресуется первая ее запись. Место расположения в памяти первой, согласно логической последовательности, записи строки может быть задано и другими средствами. Однонаправленная строка с последовательностью записей *а, в, с, в* показана на рис. 4.4.
Строчная организация данных удобна, если по некоторым причинам нецелесообразно перезаписывать поступающие данные. Для обработки

данных в упорядоченной последовательности все записи снабжаются адресами связи и с их помощью определяется цепочка записей, в которой значения ключевых реквизитов возрастают.
В качестве примера рассмотрим последовательность записей с ключами 12, 46, 31, 14, 52, 24, 27, 40. Структуру записей, соединенных в цепь, иллюстрирует рис. 5, *а.*

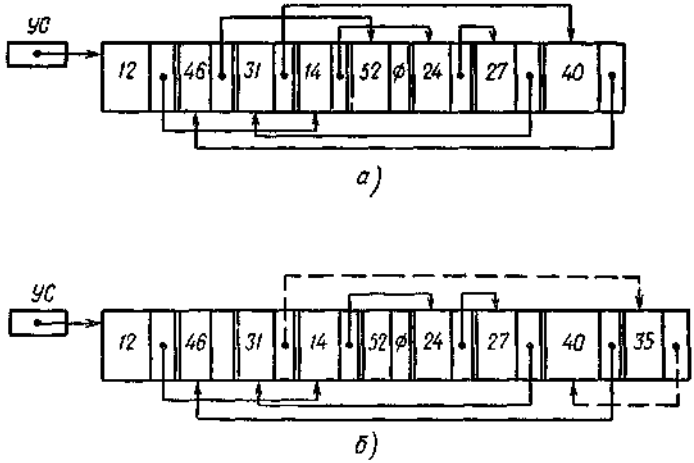


Рис. 5 Формирование упорядоченной строки из неупорядоченных записей *(а)* и корректировка этой строки *(б)*

На обработку эти записи поступят в порядке, заданном адресами связи 12, 14, 24, 27, 31, 40, 46, 52. При поступлении новой записи с ключом 35 находится ее место в цепочке и производится замена адресов связи с целью включения ее в цепочку. Измененные адреса показаны на рис. 5, *б* пунктиром. Недостатком такого преобразования данных является замедление поиска в строке по сравнению с массивом.
Для поиска данных в однонаправленной строке используется единственный метод— последовательный поиск. Ключевой реквизит первой записи (ее адрес извлекается из УС) сравнивается с искомым значением *q,* затем такое же сравнение выполняется для ключа второй записи, которая извлекается по адресу связи первой записи, и т. д. В процессе сравнении из всех записей, у которых ключ совпал со значением *q,* выбирается информация, необходимая для ответа на запрос.
Простейшее усовершенствование однонаправленной строки состоит в организации второго входа в строку через дополнительный указатель. Этот указатель может адресовать фиксированную запись строки либо адрес связи в указателе будет переменной величиной. В последнем

случае указатель адресует запись, к которой произошло последнее обращение, и называется указателем текущего значения (УТЗ).
Оценим число сравнений при поиске с использованием УТЗ . Если УТЗ адресует *i*-ю запись, а поиск заканчивается на *j*-й записи, то среднее число сравнений

$$C(i) = 1 + \frac{1}{M}\sum_{j=1}^{i} j + \frac{1}{M}\sum_{j=1}^{M-i} j \approx \frac{i^2}{M} + \frac{M}{2} - i,$$

где M — число записей в строке.
Одно сравнение при поиске тратится на выбор указателя: УС —если $q<p_i$, и УТЗ, если $p_i<q$. Первая сумма в $C(i)$ оценивает возможную продолжительность поиска, начиная с УС, вторая — ту же величину при поиске через УТЗ. Полученные $C(i)$ следует еще раз усреднить. Тогда

$$C_{ср} = \frac{1}{M}\sum_{i=1}^{M} C(i) \approx \frac{M}{3}$$

Следовательно, использование строки с дополнительным УТЗ сокращает время поиска на одну треть (для последовательного поиска в строке $C_{ср} \approx M/2$).
Рассмотрим строку с фиксированным указателем. Если обращение к каждой записи равновероятно, то в указателе необходимо адресовать среднюю запись с номером $M/2$ При поиске первое сравнение q и $p_{M/2}$ расходуется на выбор указателя для входа в строку, как и в предыдущем случае с УТЗ. После этого получаются два равноценных последовательных поиска над 0.5M записями каждый. Поэтому среднее число сравнений составит

$$C_{ср} = \frac{0.5M + 1}{2} + 1 \approx \frac{M}{4},$$

что доказывает преимущество фиксированного указателя над УТЗ.

Дальнейшая модернизация однонаправленной строки состоит в выделении $t > 1$ фиксированных указателей. Доказано, что наилучшее распределение фиксированных указателей при равновероятном местонахождении искомой записи — равномерное с обязательной адресацией первой записи. Если каждый указатель будет содержать и адрес, и ключевой реквизит соответствующей записи, что позволяет не тратить время на извлечение записи, то получаем так называемые К-индексы.

Индексом называется набор адресов и ключей записей, которые выбираются из линейно-организованных данных (массива или строки) по определенному закону. Отдельный элемент набора индексов также называется индексом.

Имеются **три практически важные разновидности индексов**:

каждый элемент индекса описывает одну запись массива или строки (сплошная индексация). Набор индексов в этом случае совпадает в основном с введенным ранее инвертированным массивом;

номера записей, информация о которых выносится в индекс, образуют арифметическую прогрессию с шагом *d,* причем первый индекс адресует первую запись. Индексы такого типа называются К-ндексами;

ключи записей, информация о которых выносится в индекс, приближенно образуют арифметическую прогрессию, причем первый индекс адресует первую запись. Индексы такого типа называются А-ндексами.

Точное описание А-индексов состоит в следующем. А-индекс с номером *i* хранит адрес записи, ключевой реквизит которой равен или непосредственно больше значения $p_1 + z\,(i—1)$, где z — константа, p_1 — ключ первой записи.

В качестве примера в табл. 2 приведены *К*- и А-индексы для массива и строки. Шаг арифметической прогрессии для К-индексов равен 5, для А-индексов $z = 15$. В А-индексе можно не хранить значение ключа адресуемой записи. У записей массива и строки приводятся только значения ключевых признаков. Строка в данном примере занимает сплошной участок памяти, но порядок записей не совпадает с последовательностью их обработки, которая задается с помощью адресов связи. Такой способ хранения строки называется цепным каталогом.

Если строка или массив снабжена К-индексом, то поиск ведется в две стадии: в массиве индексов, который отсортирован в силу упорядоченности исходной строки (массива); среди записей, расположенных между двумя соседними индексами, найденными на первой стадии.

Таблица 2

Адрес	Значение ключа
0100	6
0101	10
0102	16
0103	17
0104	22
0105	25
0106	31
0107	36
0108	40
0109	43
0110	46
0111	50
0112	52
0113	53
0114	61
0115	65
0116	71
0117	77
0118	85
0119	89

K-индексы

6	0100
25	0105
46	0110
65	0115

A-индексы

0100
0104
0107
0112
0116
0118

а) Массив

Адрес	Значение ключа	Адрес связи
		0107
0100	25	0116
0101	50	0119
0102	71	0115
0103	16	0109
0104	36	0117
0105	46	0101
0106	61	0108
0107	6	0112
0108	65	0102
0109	17	0118
0110	85	0114
0111	43	0105
0112	10	0103
0113	53	0106
0114	89	КС
0115	77	0110
0116	31	0104
0117	40	0111
0118	22	0100
0119	52	0113

УС

K-индексы

6	0107
25	0100
46	0105
65	0108

A-индексы

0107
0118
0104
0119
0102
0110

б) Строка

При поиске записи с ключом q сначала находится индекс K_i такой, что $K_i \leq q < K_{i+1}$ (i — номер индекса в массиве индексов). Далее поиск продолжается, начиная с адреса, определенного р индексе K_i. Требуемое число сравнений для строки составляет

$$C_{ср} = \log_2 W_K + \frac{M}{2W_K},$$

где M — число записей в строке (массиве);

W_K — число К-индексов.

Первое слагаемое описывает бинарный поиск среди W_K индексов, а второе — последовательный поиск на выбранном участке строки. Если К-индексы созданы для массива, то

$$C_{\text{ср}} = \log_2 W_K + \log_2 \frac{M}{W_K} = \log_2 M.$$

Последняя формула показывает, что использование К-индексов для массива не приводит к ускорению поиска.

Рассмотрим поиске использованием А-индексов. На первом этапе номер требуемого далее А -индекса определяется по формуле

$$i = \left\lfloor \frac{q - p_1}{z} \right\rfloor + 1,$$

где q — искомое значение;

z — разность значений ключей для А-индексов;

$\lfloor X \rfloor$ — функция округления числа X до целого в меньшую сторону;

p_1 — ключ первой записи.

Интервал записей на втором этапе поиска определяется номерами (адресами) записей, указанными в i-м и $(i + 1)$-м A-индексе. Поиск в этом интервале ведется бинарным методом, если записи организованы в массив, и последовательным методом, если — в строку. На вычисление величины i сравнения не тратятся, поэтому эффективность поиска с применением А-индексов в случае массива оценивается

$$C_{\text{ср}} = \log_2 (M/W_A)$$

и в случае строки (W_A — количество А-индексов) —

$$C_{\text{ср}} = \frac{M}{2W_A}.$$

Сравнение формул показывает, что поиск в массиве с А-индексами осуществляется быстрее, чем в строке. Время поиска быстро уменьшается с ростом W_A, что соответствует уменьшению z. Однако при выборе малых значений z возможны случаи, когда соседние А-индексы относятся к одной и той же записи (разность ключей соседних записей намного больше z) и увеличение числа индексов W_A не сопровождается дальнейшим сокращением времени поиска.

Выбор шага d для К-индексов в строке основан на соответствии применяемого в этом случае алгоритма поиска и двухступенчатого поиска, рассмотренного ранее. Поэтому оптимальное $d \approx \sqrt{M}$.

Индексы обоих типов используют дополнительную память сверх объема, занятого собственно данными, причем А-индексы несколько меньше.

При корректировке записей индексы также должны изменяться: всегда изменяются К-индексы и значительно реже — А-индексы.

При включении новой записи с ключом q определяется К-индекс K_i, такой, что $K_{i-1} \leq q < K_i$, где i — номер К-индекса. Затем все К-индексы с номером i и больше принимают значения ключей и адресов тех записей, которые непосредственно предшествуют ранее зафиксированным в этих индексах записям.
Аналогично при удалении записи с ключом q все К-индексы с номером i и больше принимают значения ключей и адресов тех записей, которые непосредственно следуют за ранее указанными в этих индексах записями.
Рассмотрим корректировку строки, снабженной А-индексами. Пусть в строке $(a_1, a_2, \ldots, a_m)$ в индексе A_i была адресована запись a_k с ключом p_k. Тогда при вставке в строку записи b с ключевым реквизитом q индекс A_i будет адресовать запись b лишь тогда, когда $p_1 + z\,(i - 1) \leq q < p_k$, и не изменится в остальных случаях. Только при исключении записи a_k A-индекс с номером i изменится, он будет адресовать a_{k+1}. Однако при корректировке изменяется только одно значение А-индекса, а не несколько, как это характерно для К-индексов.
Таким образом можно сделать вывод, что А-индексы целесообразнее К-индексов — они характеризуются меньшим объемом памяти, необходимым для их размещения, значительно меньшей трудоемкостью корректировки, а также более быстрым поиском при достаточно большом M.
Возможность определения места хранения данных, реализованная в А-индексах, используется также при размещении данных согласно адресной функции. Упорядоченность записей по значениям ключа в этом случае, вообще говоря, не соблюдается.
Адресной функцией называется зависимость
$i = f(p)$.
где i — номер (адрес) записи;
p — значение ключевого реквизита в записи.
Адресная функция может вырабатывать одинаковое значение i для значений ключей, принадлежащих разным записям, которые в этом случае называются синонимами.

К функции f предъявляются следующие требования:

она должна быть задана аналитически и вычислиться достаточно быстро;

ключевые реквизиты, подчиняющиеся произвольному распределению, функция должна переработать в равномерно распределенные номера записей. Это условие обычно соблюдается приближенно;

число записей-синонимов должно составлять 10—20% общего числа записей.

Известно достаточно много адресных функций, хорошо соответствующих этим требованиям. Простейшая адресная функция имеет вид

$i = p — a,$

где a — консганта.

Пусть известны минимальное значение ключа $p_{мин}$ и максимальное $p_{макс}$. Тогда $a = p_{мин} — 1$. Необходимый участок памяти для данных оценивается в $p_{макс} — p_{мин} + 1$ запись. Записи-синонимы связываются в цепочки с помощью адресов связи, они занимают дополнительную (резервную) память.

Пример размещения записей с ключами 13, 11, 14, 11, 15, 18, 14, 16 согласно адресной функции $i = p — a$ показан на рис.6.

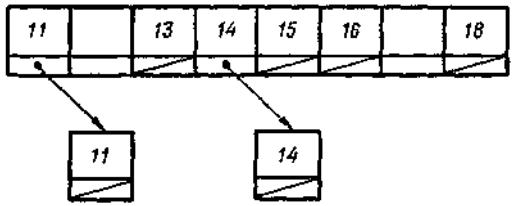


Рис. 6. Размещение записей в соответствии с адресной функцией $i=p—a$

При доступе к записи с ключом q вычисляется $i = f(q)$ и производится обращение к i-й записи. При необходимости с помощью адресов связи извлекаются все синонимы.

Недостаток адресной функции вида $i = p — a$ — большой объем неиспользуемой памяти, если $p_{макс} — p_{мин}$ много больше, чем количество записей M.

Указанного недостатка лишена функция вида

$$i = ОСТ \ (p/m).$$

Здесь m — целое число;

ОСТ — остаток от деления p на m.

Значение m принимается равным простому числу, которое непосредственно больше либо меньше числа записей M.

Выделяются две зоны памяти — основная и резервная. Основная зона содержит m записей. Резервная зона предназначена для размещения записей-синонимов.

При формировании данных согласно адресной функции сначала производится заполнение основной зоны. Если при этом позиция основной зоны, полученная при вычислении, уже занята, то текущая запись помещается в резервную зону и адресуется из позиции основной зоны. В дальнейшем при такой ситуации наращивается цепочка записей в резервной зоне. Например, для массива ключей со значениями 15, 10,20, 16, 25, 8, 13, 6, 12, 9, 28, 10, 17, 6, 23, 16, 18, 8 необходимо выбрать m = 17, поскольку М = 18 (возможно также m = 19). Содержимое основной и резервной зон иллюстрирует рис. 7.

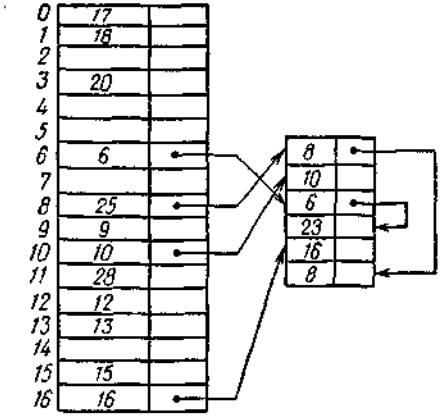


Рис. 7. Размещение записей в соответствии с адресной функцией i= ОСТ *(p/m).* Остатки от деления p= на 17 показаны слева

Резервная зона заполняется последовательно. При поиске значения, например q = 8, вычисляется i =ОСТ (8/17) = 8 и далее последовательно сравниваются 25 и q, 8 и q, 8 и q. Для обнаружения искомого ключа потребовались одно деление и три сравнения, поскольку всегда необходимо просматривать цепочку в резервной зоне до конца. В рассматриваемом примере число записей синонимов составляет 6/18, или 33%.

При корректировке записей в строке вновь поступающая запись с помещается в любом незанятом участке памяти. Пусть ее ключевой реквизит равен q. Затем среди записей строки $\{a_k; k=1,2,..,$ М$\}$ отыскиваются такие записи a_i и a_{i+1}, что $p_i \le q < p_{i+1}$. Адрес связи в c заменяется на адрес связи a_i , адрес связи в a_i заменяется на начальный адрес записи c.

При исключении записи a_n из строки ее адрес связи присваивается адресу связи в a_{n-1}. После этого запись a_n становится недоступной. Для быстрого выполнения корректировки необходимо вести учет свободных участков памяти, фактически они организуются в строку. Обработка этой строки свободной памяти может выполняться двумя путями. Либо позиция каждой исключенной записи сцепляется со строкой свободной памяти, либо исключенная запись помечается специальным образом; после исчерпания строки свободных участков памяти помеченные ранее участки образуют новую строку.

Рассмотрим включение новой записи в строку, организованную в виде цепного каталога. Новая запись размещается в первой из свободных позиций каталога, адрес которой хранится в указателе свободной памяти УСП. После этого в УСП переносится адрес связи, хранящийся в этой свободной позиции. Строка свободных позиций сокращается на одну. Действия по включению в каталог нового звена связи совпадают с операциями, описанными выше для строк со свободной укладкой. При этом изменяются адреса связи в новом звене и звене записи a_i.

Исключение записи из цепного каталога также аналогично действиям, описанным для строк со свободной укладкой. Освободившееся звено связи может стать первым или последним в списке свободных строк. Обычно оно становится первым, что позволяет унифицировать алгоритмы вставки и исключения звеньев связи. Теперь при любой корректировке изменяются значения трех адресов связи в каталоге: УСП, включаемого (удаляемого) звена, и звена, предшествующего ему в цепи. Вставку записи *d*61 и удаление записи *d30* в цепном каталоге иллюстрирует рис. 8.

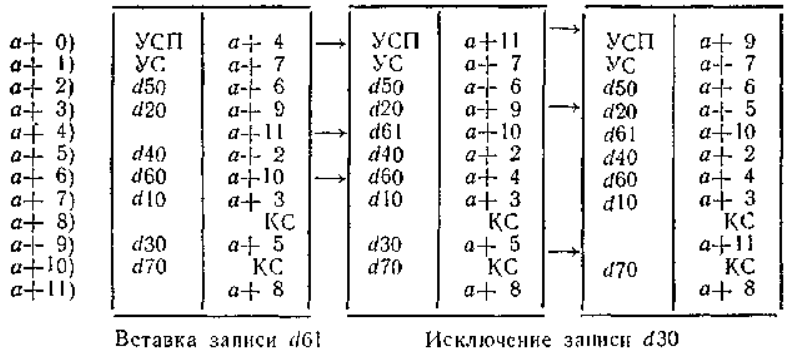


Рис. 8. Процесс корректировки записей в цепном каталоге

Запись *d61* помещается между записями *d* 60 и *d* 70. Цифры обозначают значения ключа. Стрелками отмечены звенья, адреса связи которых изменяются. Для обмена адресов необходимо специальное поле обмена.

Дополнительными возможностями по сравнению с однонаправленными строчными структурами обладают **двунаправленная и кольцевая строки.**

Двунаправленные строки рассчитаны на обработку записей в двух направлениях — прямом и обратном. Для этого в звеньях связи вводится новый адрес связи, который реализует связь с предыдущей записью. Для обратного направления в строке требуется второй адрес в указателе строки. Двунаправленный эквивалент строки, показанной на рис. 4, представлен на рис. 9.

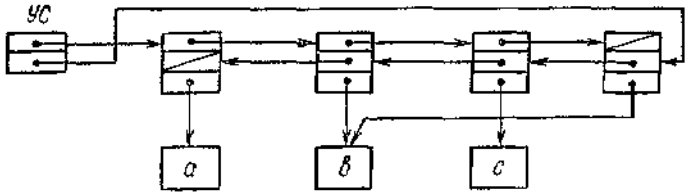


Рис. 9. Двунаправленная строка *(а, в, с, в*)

Кольцевая строка получается из однонаправленной заменой признака конца строки на адрес связи с первой записью строки (рис. 10).

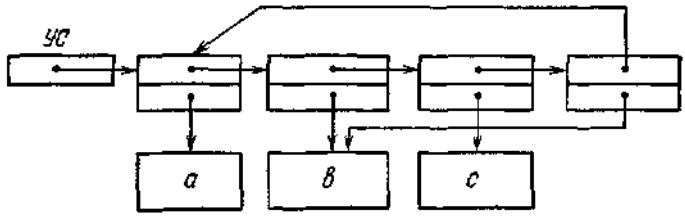


Рис. 10. Кольцевая строка *(а, в, с, в)*

Естественно, можно преобразовать в кольцо и двунаправленную строку.

С помощью кольцевой строки можно, зная одну из записей строки, быстро извлечь все ее записи. Двунаправленная строка дает возможность обрабатывать записи как в порядке возрастания ключей, так и в порядке их убывания.

Набор строк, организованных на общем множестве записей, называется мультисписком. Мультисписок предназначен для обработки записей по нескольким ключевым реквизитам. Рассмотрим построение мультисписка, предназначенного для быстрого поиска записей по заданному набору ключей. Если система рассчитана на обработку *d* ключей, то создается *d* строк. Каждый из них определяет цепочку записей, упорядоченных по одному из *d* ключей. Длина каждой строки в мультисписке равна числу исходных записей. Результат формирования мультисписка для записей из табл. 3 — следующие четыре строки:

(c, a, f, d, b, h, e)

(b, h, a, f, e, d, c)

(f, a, c, d, h, b, e)

(h, c, e, a, f, d, b)

Таблица 3

Признаки				Идентификатор записи
1	2	3	4	
10	30	16	15	*a*
36	2	60	60	*e*
4	88	32	11	*c*
30	70	42	58	*d*
75	62	87	13	*e*
11	43	2	25	*f*
42	22	59	7	*h*

Простейший запрос, который обрабатывается с помощью мультисписка, имеет вид

$\langle U = u \rangle \wedge \langle V = v \rangle \wedge \ldots \wedge \langle Z = z \rangle$, где *U, V*..... Z—некоторые имена признаков; *u, v, ..., z*—некоторые значения признаков.

Для начала поиска можно выбрать любую из строк мультисписка, признак которой встречается в запросе, например *U*. В этой строке отыскиваются все записи, удовлетворяющие условию $U = u$. Они затем

проверяются на наличие значений *v*,..., *z*, и окончательно отобранные записи составляют результат поиска. Естественно выбрать для начала поиска наиболее короткую строку из набора *U, V*,..., *Z*. Этот способ может быть использован и в инвертированном массиве.

6.2. Нелинейная организация данных

Нелинейная организация данных — множество записей, каждая из которых связана с произвольным количеством предшествующих и последующих записей. Важнейшими вариантами нелинейной организации данных являются **деревья, списки и решетки**.
Древовидной организацией данных (деревом) называется множество записей, расположенных по уровням следующим образом: на 1-м уровне расположена только одна запись (корень дерева), к любой записи *i*-го уровня ведет адрес связи только от одной записи *(i* — 1)-го уровня.
В данном определении понятия «дерево» и «уровень» вводятся одновременно. Если записи получат номера уровней, соответствующие определению, то они получат и древовидную организацию.
Количество уровней в дереве называется рангом. Записи дерева, которые адресуются от общей записи *(i* — 1)-го уровня, образуют группу. Максимальное число элементов в группе называется порядком дерева. В дереве (рис. 11) группами являются множества записей *{B, C, D}*, *{E, F}*, {G}; порядок этого дерева равен 3, ранг 3.

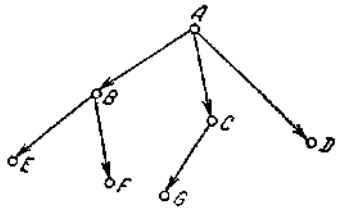


Рис. 11. Древовидная организацпя данных

Деревья обычно формируются двунаправленными, адрес от записи *i* + 1-го уровня к записи *i*-го уровня называется обратным.
При размещении древовидной структуры в памяти каждая запись может занимать произвольное место. Назовем звеном связи набор адресов связи, принадлежащих одной записи. Если порядок дерева равен *p,* то псе звенья связи состоят из *p*+1 адреса (один адрес — обратный). Корень дерева адресуется из специального указателя

дерева. Незанятые адреса связи содержат признак конца списка КС. В качестве примера размещения дерева в памяти ЭВМ на рис. 12 показан один из возможных вариантов представления дерева (см. рис. 11).

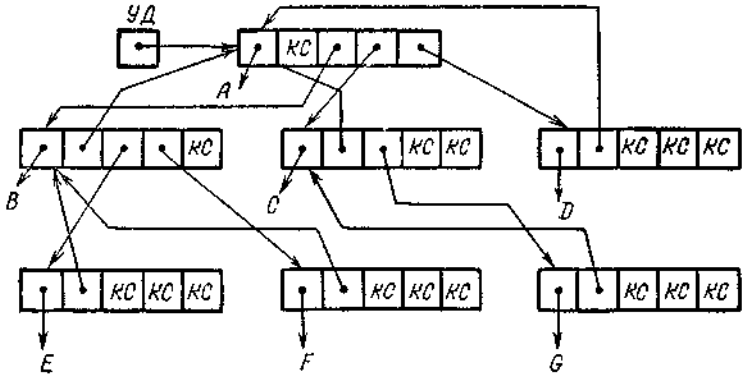


Рис. 12. Размещение в памяти дерева (см. рис. 11)

Рассмотрим деревья порядка 2 (бинарные). Они интересны тем, что составляющие их записи могут быть упорядочены. Для этого один из признаков записи должен быть объявлен ключевым.

Чтобы определить понятие упорядоченности бинарных деревьев, требуется ввести ряд вводных понятий. В качестве примера рассмотрим упорядоченное бинарное дерево (рис. 13, в скобках указано значение ключа). Запись *a* — корень дерева. Записи, у которых заполнены два адреса связи, называются полными, записи с одним заполненным адресом— неполными, записи с двумя незаполненными адресами — концевыми. На рис. 13 записи *a, в, e, f* — полные, *c* — неполная, *d, h, i, j, k*— концевые. Адреса связи делятся на левые и правые. Так, адрес от *e* к *h* — левый, от *e* к *i* — правый. Каждая запись имеет левую и правую ветви. Правую (левую) ветвь записи образует поддерево, адресованное из этой записи через правый (левый) адрес связи. У записи *c* (см. рис. 13) правая ветвь состоит из записей *f, i, k,* левая ветвь — пустая.

В упорядоченном бинарном дереве значение ключа каждой записи должно быть больше, чем значение ключа у любой записи на его левой ветви, и не меньше, чем ключ любой записи на его правой ветви. Упорядоченное бинарное дерево формируется из неупорядоченного массива записей по следующему алгоритму. Первая запись массива с

ключом p_1 становится корнем дерева. Значение ключа второй записи p_2 сравниваем с p_1, находящимся в корне дерева. Если $p_2 < p_1$ то помещаем вторую запись на левой от корня ветви, в противном случае — на правой ветви. Выбор места i-й записи массива в дереве производится следующим образом. Ключ p_i сравнивается с корневым значением и выполняется переход по левому адресу, если $p_1 > p_i$, а при $p_1 \leq p_i$ — по правому адресу. Ключ, записанный по этому адресу, также сравнивается с p_i и снова организуется переход по левому или правому адресу и т. д. Если требуемый адрес не заполнен, то он должен адресовать теперь запись с ключом p_i. Итак, алгоритм строит упорядоченное бинарное дерево из одной записи, затем из первых двух, первых трех и т.д. Например, дерево на рис. 13 получается из массива с ключевыми признаками 23, 10, 18, 27, 15, 32, 8, 30, 32, 21. При формировании упорядоченного бинарного дерева в среднем производится

$$C = 1{,}39\ M \log_2 M$$

сравнений пар признаков, где M — число записей, для которых строится дерево.

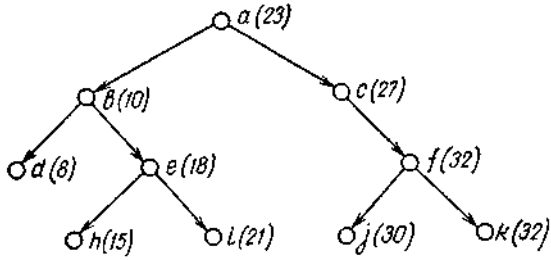


Рис. 13. Упорядоченное бинарное дерево

В процессе поиска данных по совпадению в упорядоченном бинарном дереве просматривается некоторый путь по дереву, начинающийся всегда в его корне. Искомое значение q сравнивается со значением корня p_1. Если они равны, то запись, соответствующая корню, помещается в поле результата и производится сравнение q и значения ключа вершины, лежащей справа от корня. Это необходимо для того, чтобы обнаружить в бинарном дереве другие вершины с ключом, равным q. Если $p_1 > q$, то просмотр дерева продолжается по левой дуге корня, если $p_1 < q$, то — по правой. Для произвольной вершины дерева p_i ; результаты сравнения означают:
$p_i = q$ — запись, соответствующая i-й вершине, помещается в поле результата, и путь продолжается по правой дуге p_i ;

$p_i > q$ — производится сравнение q с вершиной, лежащей на левой дуге p_i;

$p_i < q$ — производится сравнение q с вершиной, лежащей на правой дуге p_i.

Поиск заканчивается, когда в какой-либо вершине p_i отсутствует дуга, необходимая для дальнейшего просмотра дерева.
Для алгоритма поиска по совпадению в произвольном бинарном дереве среднее число сравнений равно:

$$C_{ср} = 1{,}39 \log_2 M.$$

Включение новой записи при корректировке бинарного дерева, по существу, означает отработку одного этапа алгоритма формирования дерева с включаемой записью на входе.
Сложность исключения зависит от того, какая запись исключается — концевая, неполная или полная. Первые два случая аналогичны корректировке строки. Адрес связи на исключаемую концевую запись заменяется на символ КС (конец списка), адрес связи на исключаемую неполную запись заменяется на ее собственный адрес связи.
При исключении полной записи решается задача о подстановке на ее место другой записи, такой, что ее ключ не нарушает общей упорядоченности бинарного дерева — такие записи называются соседними. Способ нахождения соседней записи очень простой. Если исключаемая запись имеет ключ q, то от нее производится переход по левой или правой ветви (что безразлично) и поиск q. Запись, на которой остановится поиск, будет соседней. Она пересылается на место исключаемой записи, а сама соседняя запись исключается. Это уже простая задача, так как соседняя запись не может быть полной.
Ранг упорядоченного бинарного дерева можно сократить путем специальных преобразований; преобразованное дерево называется подравненным. Назовем длиной ветви дерева разность между максимальным уровнем записей этой ветви и уровнем записи, определяющей ветвь. Так, на рис. 13 у записи *c* длина левой ветви — 0, длина правой ветви —2. Бинарное дерево является подравненным, если для каждой его записи длины ее ветвей различаются не более чем на 1. Поэтому дерево на рис. 13 не является подравненным. Чтобы перейти к подравненному дереву, надо найти запись, соседнюю справа к *c,* поместить ее на место *c,* а запись *c* заново включить в дерево. После таких преобразований разность длин ветвей сократится. Эти действия проводятся со всеми записями, нарушающими подравненность.

Лучшая последовательность преобразования ветвей — от корня в соответствии с возрастанием номеров уровней.
Среднее число сравнений при поиске в подравненном дереве сокращается до $C_{ср} = 1{,}04 \log_2 M$ по сравнению с $1{,}39 \log_2 M$ в произвольном бинарном упорядоченном дереве.
Таблица 4

Критерий сравнения	Массив	Строка	Бинарное дерево	Примечание
Время формирования	Сортировка $t_{11} M \log_2 M$ максимально	Сортировка $t_{21} M \log_2 M$ максимально	Формирование $t_{31} M \log_2 M$ в среднем	$t_{21} < t_{31} < t_{11}$
Время поиска	В среднем $t_{12} (\log_2 M - 1)$	В среднем $t_{22} M/2$	В среднем $t_{32} \log_2 M$	$t_{32} < t_{12}$
	Максимально $t_{12} \log_2 M$	Максимально $t_{22} M$	Максимально $t_{32} M$	
Время корректировки	Сдвиг $0{,}5M$ записей	Замена 3—4 адресов	Замена 3—4 адресов	Предварительно выполняется поиск
Объем дополнительной памяти	0	$2M$ адресов связи	$3M$ адресов связи	

Максимальное число сравнений ограничивается $C_{макс} = 1{,}44 \log_2 M$.
Однако для подравненного дерева характерны нарушения условий подравненности после включения-исключения некоторых записей.
Последовательная, строчная и бинарная древовидная организации данных предназначены для решения общей задачи — обработки записей с одним ключевым признаком. Поскольку они взаимозаменяемы, имеет смысл задача выбора лучшей структуры данных. Обозначим через *П* — последовательную, С — строчную и *Д* — бинарную древовидную организацию данных, а через > — знак «лучше». Сравним методы организации данных по следующим критериям:
время формирования *С* >П> *Д;*
время поиска *П* > *Д* > С;
время корректировки *Д* > *С* >П;
объем дополнительной памяти П > *С* > *Д.*
Обоснование этих неравенств содержится в табл. 4. Через t_{ij} обозначены константы с размерностью времени.
Абсолютно лучшего метода организации данных не существует. Однако массив и строка допускают использование *А*-индексов, в этом

случае при поиске и корректировке строка предпочтительнее бинарного дерева и может быть признана лучшей по всей совокупности критериев.

Ключевой признак в задачах обработки данных часто удобно представлять разделенным на отдельные символы. Древовидная организация данных, ориентированная на посимвольное хранение ключевых признаков, называется позиционным деревом Каждый символ ключа соответствует вершине дерева: первый — вершине первого уровня, второй — вершине второго уровня и т. д. Каждый путь на дереве соответствует цепочке символов, образующих какой-то ключ. Если у некоторых ключей несколько начальных символов совпадает, то они имеют общие вершины в дереве. Вершина, относящаяся к последнему символу ключа, содержит специальный разделитель (например, «*») и адрес связи на запись с соответствующим ключом. В памяти строится столько деревьев, сколько различных первых символов есть в значениях ключевых признаков. На рис. 14 показаны два позиционных дерева для доступа к данным, приведенным в табл. 5.

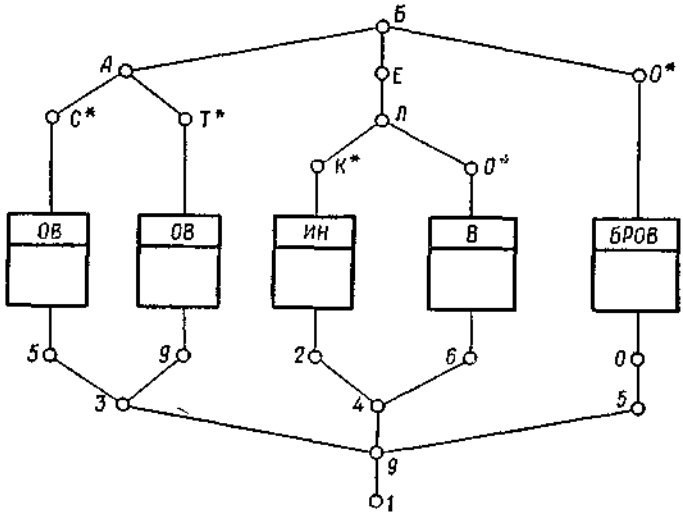


Рис. 14. Два позиционных дерева для признаков ФАМИЛИЯ и ГОД РОЖДЕНИЯ

Для сокращения объема дерева и времени поиска в нем не учитываются те символы ключей, которые не влияют на однозначное отличие записей. Так, в дереве для ключей-фамилий у значения

БОБРОВ (см. рис. 14) первые две буквы определяют ветвь, ведущую только к одной записи, поэтому остальные символы фамилии хранятся вне дерева. Разумеется, корректировка (например, включение фамилии БОКОВ) потребует переноса части этих символов в дерево.

Таблица 5

ФАМИЛИЯ	ГОД РОЖДЕНИЯ
Белов	1946
Батов	1939
Бобров	1950
Басов	1935
Белкин	1942

В древовидной организации данных связь какой-то записи с N записями, составляющими ее группу, реализуется с помощью N адресов связи. Возможно, однако, связать все записи группы в цепочку и адресовать с предшествующего уровня первую запись группы. Таким образом получается новая нелинейная организация данных — **список**. ***Список* является множеством элементов, каждый из которых может быть либо записью, либо списком**. Структуру списка выражают формулой, в которой записи помечаются буквами, а списки заключаются в круглые скобки. Список, включенный в другой список, называется подсписком. Перечисление всех списков из записей $A_1, A_2, \ldots, A_n$, образующих множество L^*_K, сводится к следующему. Обозначим через Lfc множество всех кортежей с элементами из L_k. Введем последовательность множеств

$$L_1 = L_0^* \cup L_0;$$
$$L_2 = L_1^* \cup L_1;$$
$$L_3 = L_2^* \cup L_2;$$
$$\ldots$$

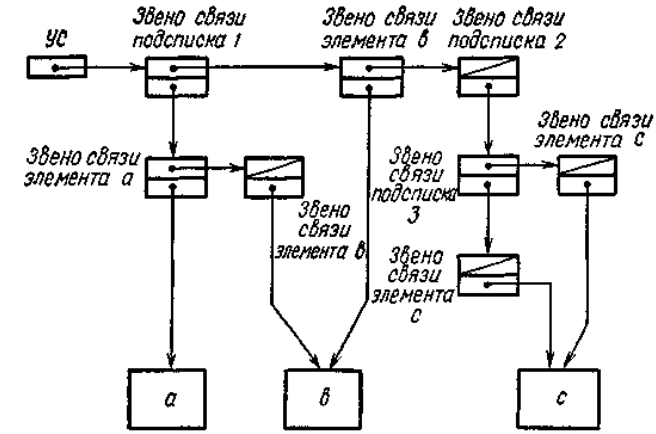
Всесписки содержатся в множестве $L = \bigcup_{i=0}^{\infty} L_i$. Пусть

$$L_0 = \{A, B\};$$
$$L_1 = \{(A, B), (A), (B), A, B\};$$
$$L_2 = \{((A, B), (A), (B), A, B), ((A, B), (A), (B), A, \ldots)\}$$

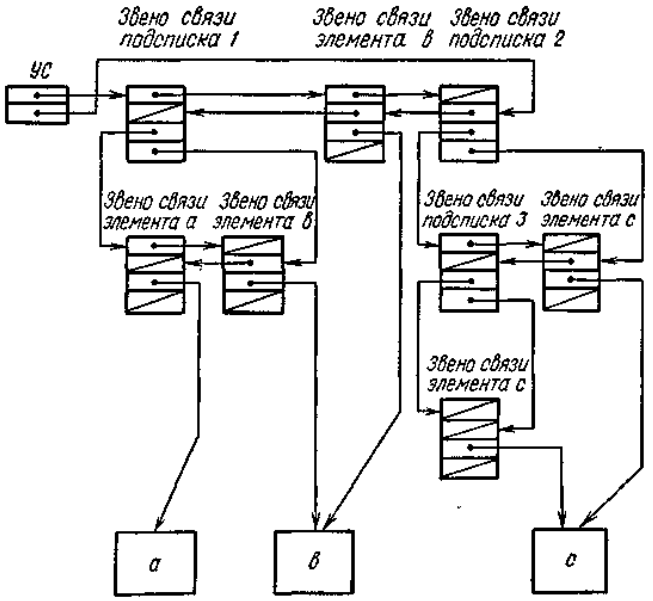
и т. д.

Введенный аппарат списков намного шире, чем требуется для представления деревьев**. Он используется самостоятельно также в задачах искусственного интеллекта и анализа текстовой информации**.

Записи, входящие в список, могут занимать произвольные участки в памяти ЭВМ. Адреса связи, принадлежащие каждой записи, образуют звено связи. Список, как и строка, может быть однонаправленным, двунаправленным и кольцевым. Последний вариант на практике встречается редко. В звене связи однонаправленного списка — два адреса: первый указывает на следующий элемент списка, второй — на подсписок или запись. В звене связи двунаправленного списка — четыре адреса: два из них обеспечивают прямое и обратное направление в списке, третий и четвертый адресуют начало и конец подсписка. Однонаправленный и двунаправленный варианты списка *((а, в), в, ((с), с))* показаны на рис. 15.



а)



б)

Рис. 15. Однонаправленный *(а)* и двунаправленный (б) список *((а, в), в ((с), с))*

Обозначение конца списка совпадает с обозначением конца строки. Дерево тоже имеет формулу, так как существует эквивалентный ему список. Формула дерева, приведенного на рис. 11: *(A, (B, (E, F), C,* (G), *D)).*

Решетки предназначены для представления записей с несколькими ключевыми признаками. Рассмотрим случай трех ключевых признаков *A, B, C* с возможными значениями $A_1 < A_2$, $B_1 < B_2 < B_3$, $C_1 < C_2$. 12 записей содержат все комбинации этих значений. Запись *X* с набором значений ключей A_i, B_i, C_i имеет в решетке адрес связи на запись *Y* с ключами A_j, B_j, C_j, если у *X* и *Y* любые две пары ключей равны, а для одной пары ключ из *У* непосредственно больше ключа из *X*. Пример решетки показан на рис. 16.

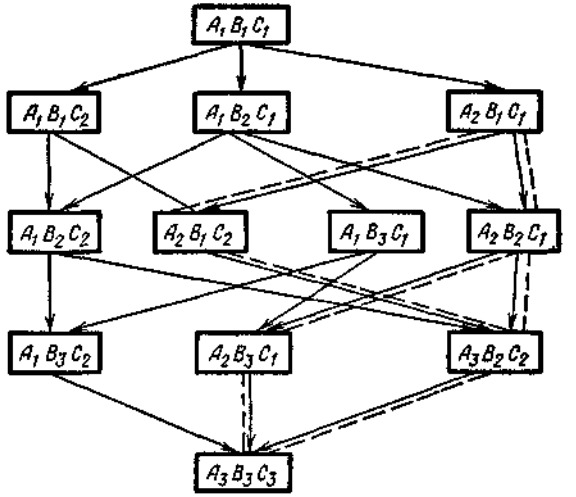


Рис. 16. Пример решетки для трех ключевых признаков

Входом в решетку служит запись с ключами < A_1, B_1, C_1 >.

В общем случае записи, содержащие N ключевых признаков *A*1 (со значениями $a_{11} < a_{12} < \ldots < a_{1p}$), $A2$ ($a_{21} < a_{22} < \ldots a_{2g}$), ..., AN ($a_{N1}\, a_{N2} \ldots a_{Nl}$), образуют структуру решетки, если существует путь из адресов связи от записи *X* со значениями ключей $< a_{1i}, a_{2j}, \ldots, a_{NK} >$ к записи $Y < a_{1l}, a_{2m}, \ldots, a_{Nn} >$, когда для некоторого $t \leqslant N$ $a_{ts} <$

$< a_{ts}\ (a_{ts} \in X,\quad a_{tr} \in Y)$, а остальные пары значений ключей из X и Y совпадают. Решетки удобно создавать в тех случаях, когда в записях содержатся почти все возможные комбинации значений ключа. При уменьшении числа записей возможно появление нескольких входов в решетку или разделение ее на несвязные компоненты. Преимуществом решеток является небольшое число сравнений при поиске (в рассматриваемом примере пять сравнений максимально при обработке запроса $<A = a_h> \wedge <B = b_n> \wedge <C = c_m>$). Та же задача для мультисписка требует максимально шесть сравнений. Характеристики решетки можно улучшить почти вдвое, сделав ее двунаправленной.

Еще одно положительное свойство решеток состоит в том, что записи с постоянным значением какого-то ключа образуют подрешетку. Например, подрешетка получается для $A = a_2$ (на рис. 16 она выделена пунктирными стрелками). Имеются подрешетки и для записей с постоянными значениями двух признаков.

Решетка, мультисписок и N инвертированных массивов (которые в этом случае превращаются в N массивов из M индексов каждый) позволяют решать одну и ту же задачу — обработку M записей по N ключевым признакам. Мультисписок имеет преимущество перед системой инвертированных массивов по объему занимаемой памяти. Мультисписок и решетка занимают примерно равную память, но в решетке обеспечивается более быстрый поиск, так как из-за ветвления адресов связи в подрешетках не надо просматривать их полностью. В мультисписке каждой подрешетке соответствует отдельный список, просматриваемый последовательно, до конца.

6.3. Страничная организация данных

Объемы данных, хранимых в информационных системах, как правило, очень велики и часто достигают 10^7—10^8 символов. Обследование ряда ЭИС показало, что одна составная единица информации содержит от 2500 до 150000 значений (в среднем, 22000 значений) и число символов в значении колеблется от 12 до 120 (в среднем 40 символов). Для хранения таких объемов информации необходимо использовать запоминающие устройства большой емкости.

Память запоминающего устройства подразделяется на страницы. *Страницей* называется такой участок памяти, информация из которого может быть считана или записана с помощью одного оператора, а при извлечении необходимых для обработки данных требуется считать всю страницу, на которой они располагаются. Страницы памяти

группируются в *тома памяти,* связанные, как правило, с одним механизмом доступа.

Когда данные располагаются в страничной памяти, к известным уже критериям выбора метода организации данных (время формирования, поиска и корректировки, объем дополнительной памяти) добавляются новые критерии и появляются новые факторы, влияющие на эти критерии.

Рассмотрим особенности представления массива на страницах **внешнего запоминающего устройства**. Записи массива должны быть упорядочены по значениям некоторого ключевого признака. На каждой странице располагается часть записей массива (блок записей) при условии, что записи одного блока упорядочены и упорядоченность не нарушается при переходе от страницы к странице. Страница обычно не заполняется полностью, т. е. в ней оставляется резервная память (обычно 10— 15% размера страницы). Если этого не делать, то включение новых записей потребует создания для них новых страниц при каждой корректировке. Эти страницы будут содержать достаточно мало записей, отчего резко возрастет объем дополнительной памяти, необходимый для массива. Ключ последней записи в каждой странице выносится в массив К-индексов. Пример размещения отсортированного массива на страницах внешнего запоминающего устройства приводится на рис. 17.

-

К-индексы страниц	Номер страницы	Записи массива (представлены своими ключами)						
24	1	09	11	15	18	20	24	
37	2	26	29	30	32	35	37	
51	3	38	41	44	45	48	51	
61	4	52	54	55	58	60	61	
73	5	63	66	67	69	72	73	
84	6	75	76	78	81	83	84	
								Резервная память

Рис. 17. Размещение записей массива в страничной памяти

При обработке массива значительное время (до 95%) расходуется на извлечение страниц из внешней памяти. Поэтому при расчете времени

формирования, поиска и корректировки данных в массиве обычно учитывается только время выборки страниц, на которое оказывают влияние два специфических фактора — вероятности обращения к каждой странице, называемые активностями страниц, и взаимное расположение страниц на внешнем запоминающем устройстве. Возможны два режима извлечения страниц из отдельного тома памяти: доступ от начальной точки и блуждающий доступ. Доступ от начальной точки предполагает, что после обработки *i-й* страницы механизм доступа отводится в стандартное положение (начальную точку) и извлечение *(i* + 1)-й страницы происходит от начальной точки, затем механизм доступа снова возвращается в нее. Обработка следующей страницы при блуждающем доступе начинается от места чтения-записи предыдущей страницы.

Рассмотрим оптимизацию введенных режимов доступа — нахождение оптимальной начальной точки для первого случая и нахождение оптимального взаимного расположения страниц для второго. Критерием оптимальности служит минимум среднего времени доступа. Известны вероятности обращения к страницам (активности) r_i $i = 1, 2,\ldots,N$ и время t_{ij} доступа от i-й страницы к j-й, которое обычно описывается линейной функцией с константами *a, b:*

$$t_{ij} = a + b\,|i - j|.$$

В случае когда $i = j$, время доступа всегда меньше, чем $t_{ij} = a$, однако эта разница при анализе обычно не учитывается. Коэффициент b всегда положителен. Чем ниже величина b, тем выше быстродействие запоминающего устройства. У реальных запоминающих устройств указанная формула для t_{ij} соблюдается обычно приближенно. Для запоминающих устройств с параметром $b = 0$ (память равного времени доступа) различное взаимное расположение страниц не оказывает влияния на время доступа к данным.

Время доступа от «средней» точки x к произвольной странице i составляет

$$T(x) = \sum_{i=1}^{N} r_i t_{xi} = \sum_{i=1}^{x} r_i (a + b_x - b_i) +$$

$$+ \sum_{i=x+1}^{N} r_i (a - b_x + b_i).$$

Будем считать i изменяющимся непрерывно и введем две функции $F(x) = \int_0^x r(y)\,dy$ и $\Phi(x) = \int_0^x yr(y)\,dy$. Тогда получим

$$T(x) = a - 2b\Phi(x) + 2bxF(x) - bx + b\Phi(N).$$

Точка x, доставляющая минимум времени доступа $T(x)$, должна удовлетворять уравнению $T'(x) = 0$, т. е.

$$-2b\Phi'(x) + 2bF(x) + 2bxF'(x) - b = 0.$$

Еслиучесть очевидное равенство $\Phi'(x) = xF'(x)$, то решением этого уравнения является $F(x) = 0{,}5$, откуда может быть определено x. Возвращаясь к дискретному распределению r_i, можно установить, что для оптимального x величина

$$R_x = \sum_{i=1}^{x} r_i$$

должна как можно меньше отличаться от 0,5.

В качестве примера рассмотрим том запоминающего устройства состоящий из $N = 7$ страниц с активностями страниц $r_1 = 0{,}27$; $r_2 = 0{,}16$; $r_3 = 0{,}10$; $r_4 = 0{,}24$; $r_5 = 0{,}08$; $r_6 = 0{,}09$; $r_7 = 0{,}06$. Вычисляем $R_1 = 0{,}27$; $R_2 = 0{,}43$; $R_3 = 0{,}53$; $R_4 = 0{,}77$. Поэтому средняя точка в нашем примере соответствует третьей странице.

Целевая функция в задаче оптимального блуждающего доступа соответствует минимуму времени доступа T от одной произвольной страницы к другой произвольной странице:

$$T = \sum_{i=1}^{N}\sum_{j=1}^{N} t_{ij} r_i r_j = a + b\sum_{i=1}^{N}\sum_{j=1}^{N} |i-j| r_i r_j.$$

Укажем без доказательства, что минимум T достигается, если соблюдается одно из двух неравенств

$$r_1 \leqslant r_N \leqslant r_2 \leqslant r_{N-1} \leqslant \ldots$$

$$r_N \leqslant r_1 \leqslant r_{N-1} \leqslant r_2 \leqslant \ldots$$

Оптимальная перестановка страниц получается в результате следующей процедуры. Отсортируем страницы по убыванию r_i и полученную последовательность обозначим переменной j. После этого j-я страница должна получить в своем томе номер $R(j)$, определяемый по формуле

$$R(j) = \lfloor N/2 \rfloor + 1 - (-1)^j \lfloor j/2 \rfloor,$$

где $\lfloor z \rfloor$ — целая часть z.

В условиях предыдущего примера отсортированная последовательность r_i имеет вид 0,27; 0,24; 0,16; 0,10; 0,09; 0,08; 0,06.

Оптимальное размещение страниц показано на рис. 18. График приблизительно симметричен относительно *R.*(*j*) == [_N/2_] + 1. Применение блуждающего доступа ограничено, поскольку требуемая упорядоченность по активности страниц часто разрушает важную упорядоченность страниц по ключу.

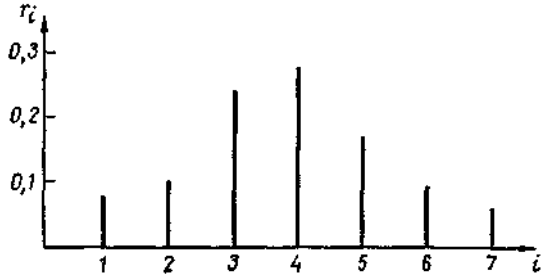


Рис. 18. Активности для оптимального размещения страниц при блуждающем доступе к ним

При включении или исключении записи с ключом *q* корректировка производится только в пределах *i*-й страницы, где *i* определяется по условию $K_{i-1} < q \leq K_i$ (K_{i-1}, K_i — К-индексы для массива). Внутри страницы алгоритм корректировки производит сдвиги хранящихся там записей (см.6.1). Включение и исключение записей в отдельных страницах могут происходить неравномерно.

Когда резервная память страницы будет исчерпана и в нее потребуется включить новую запись, наступает переполнение страницы. Частота переполнений описывается формулой

$$K = \frac{V+1}{2r-1},$$

где *K* —ожидаемое число корректирующих обращений к произвольной странице до наступления ее переполнения;

V — объем свободной памяти в странице, выраженной в количестве записей;

$r > 0{,}5$ — вероятность того, что корректирующее обращение является включением.

Величина 1 —*r* описывает вероятность исключения записи, замены значений реквизитов не учитываются. В случае $r \leq 0{,}5$ страница, если ее рассматривать в среднем, никогда не переполняется.

Когда *i*-я страница будет переполнена, необходимо создать новую страницу, перезаписать в нее часть записей из *i*-й страницы, добавить один К-индекс для новой страницы и скорректировать значения К-индексов.

Рассмотрим задачу оптимального распределения массивов между томами запоминающих устройств. Введем матрицы $B = \|b_{ij}\|$ и $X = \|x_{jk}\|$, где $i = 1,2,\ldots,T$— номер запроса пользователя, $j = 1,2,\ldots, N$ — номер массива, $k = 1,2,\ldots, p$ — номер тома запоминающего устройства:

$$b_{ij} = \begin{cases} 1, \text{ если обработка } i\text{-го запроса включает обращения к } j\text{-му массиву;} \\ 0 — \text{в противном случае.} \end{cases}$$

$$x_{jk} = \begin{cases} 1, \text{ если } j\text{-й массив размещен на } k\text{-м томе;} \\ 0 — \text{в противном случае.} \end{cases}$$

Предполагается, что при $b_{ij_1} = b_{ij_2} = \ldots = b_{ij_s} = 1$ массивы $j_1, j_2,\ldots, j_s$ обрабатываются совместно. Введем выражение

$$E_{ik} = \sum_j b_{ij} x_{jk}.$$

E_{ik} равно числу массивов, хранящихся в k-*м* томе памяти и необходимых при обработке j-го запроса. Если значение $E_{ik} \geq 2$ для некоторых j, k, то время обработки i-го запроса будет достаточно большим из-за перемещений механизма доступа от одного массива к другому (эти перемещения отсутствуют при $E_{ik} = 0$ и $E_{ik} = 1$). Поэтому целевой функцией в задаче распределения массивов, которую необходимо минимизировать, является

$$F = \sum_{ik} E_{ik} = \sum_{i,k,j} b_{ij} x_{jk}.$$

Среди ограничений задачи выделим условия

$$\sum_j V_j x_{jk} \leqslant W_k, \quad k = \overline{1, p},$$

где V_j — число символов в j-м массиве;
W_k — число символов в k-*м* томе.
Приведенные условия характеризуют ограниченную емкость томов памяти.
Условия $\sum_k x_{jk} = 1$ разрешают хранить каждый массив только в одном томе.

Таблица 6

ФАМИЛИЯ	ПРОФЕССИЯ			
	слесарь	токарь	штамповщик	электрик
Бардин		A (1)	A (2)	
Басов	A (3)		A (4)	A(5)
Батов	A (6)	A (7)		
Белов			A (8)	A (9)
Иванов		A (10)		A (11)
Исаев	A (12)	A (13)	A (14)	

Задача с целевой функцией F является известной задачей линейного программирования с булевскими переменными x_{jk}.

Размещение записей упорядоченной строки на страницах запоминающего устройства в основном производится так же, как и для массива. Однако каждая страница представляет собой цепной каталог, в котором размещается часть записей строки, связанная в упорядоченную цепь. Для строк также справедливы выводы об оптимальном доступе и переполнении страниц, сформулированные выше.

Рассмотрим особенности страничной организации мультисписков, которые предназначены для обработки записей по нескольким ключевым признакам. В качестве примера в табл.6. показаны 14 записей с ключевыми признаками ФАМИЛИЯ и ПРОФЕССИЯ (остальные реквизиты в данном случае несущественны). На пересечении строки с некоторой фамилией и столбца с некоторой профессией указан номер записи, которая содержит именно эти значения в качестве ключей.

В простейшем случае мультисписок будет содержать две строки — с указателем ФАМИЛИЯ — (*A* (1), *A* (2), *A* (3),..., *A* (13), *A* (14)) и с указателем ПРОФЕССИЯ — (*A* (3), *A* (6), *A* (12), *A* (1), *A* (7), *A* (10), *A* (13), *A* (2), *A* (4), *A* (8), *A* (14), *A* (5), *A* (9), *A* (11)).

При размещении мультисписка в страничной памяти необходимо размещать каждую строку в небольшом числе рядом расположенных страниц, что позволит уменьшить время доступа к записям.

Эффективная организация мультисписка предполагает выполнение следующих условий: число записей в каждой строке должно быть небольшим; адреса хранения записей строки должны монотонно возрастать либо монотонно убывать; строка должна размещаться по возможности в одной странице.

Для сокращения длины строк в мультисписке необходимо детализировать содержимое указателей. Например, указатель

ФАМИЛИЯ = 'Ба' определяет строку (*A* (1), *A* (2), *A* (3), *A* (4), *A* (5)), указатель ФАМИЛИЯ ='Бе' — строку (*A* (6), *A* (7), *A* (8), *A* (9)), указатель ФАМИЛИЯ = 'И' — строку (*A* (10), *A* (11), *A* (12), *A* (13), *A* (14)).

Поскольку реквизит ПРОФЕССИЯ содержит четыре значения, возможно существование следующих четырех строк: (*A* (3), *A* (6), *A* (12)), (*A* (1), *A* (7), *A* (10), *A* (13)), (*A* (2), *A* (4), *A* (8), *A* (14)), (*A* (5), *A* (9), (11)).

При поиске в «сокращенных» строках необходимо сначала проанализировать все указатели, чтобы выбрать одну строку, заведомо содержащую требуемую информацию. Рассмотрим, например, запрос

НАЙТИ ФАМИЛИЯ = 'Иванов' ∧ ПРОФЕССИЯ = 'Электрик'

Мультисписок содержит три строки для реквизита ФАМИЛИЯ и четыре — для реквизита ПРОФЕССИЯ, приведенные выше. Следовательно, потребуются три обращения к памяти для выбора строки (*A* (10), *A* (11), *A* (12), *A* (13), *A* (14)) и четыре обращения для выбора строки (*A* (5), *A* (9), *A* (11)).

Последняя строка короче, поэтому она просматривается полностью до извлечения нужной записи *A* (11), т. е. необходимы три обращения к памяти. Общее число обращений в рассматриваемом примере—10. Однако указатели всех строк занимают, как правило, одну страницу, как и сами строки, которые разделены на достаточно короткие участки. Поиск также ускоряется оттого, что в каждом указателе хранится информация о числе записей в строке.

При размещении дерева в страницах внешней памяти необходимо, чтобы связи записей в каждой странице имели свойства дерева, в том числе единственную корневую запись. Доступ к записям страницы может производиться только через корневую запись. Пример распределения записей дерева по страницам внешней памяти показан на рис. 19.

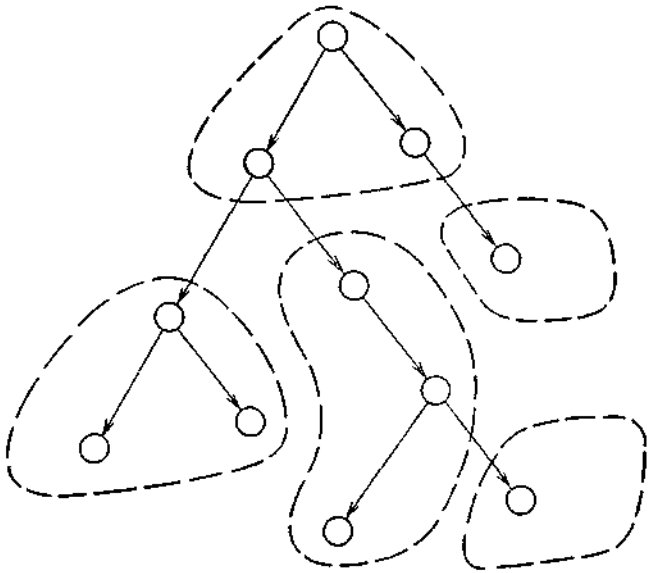


Рис. 19. Размещение записей дерева и страничной памяти.
Пунктиром обведены записи, образующие страницу

Если объем данных очень большой и индексы к ним занимают несколько страниц, то приходится создавать индексы для этих индексов. Получаемая структура называется B-деревом (рис. 20).

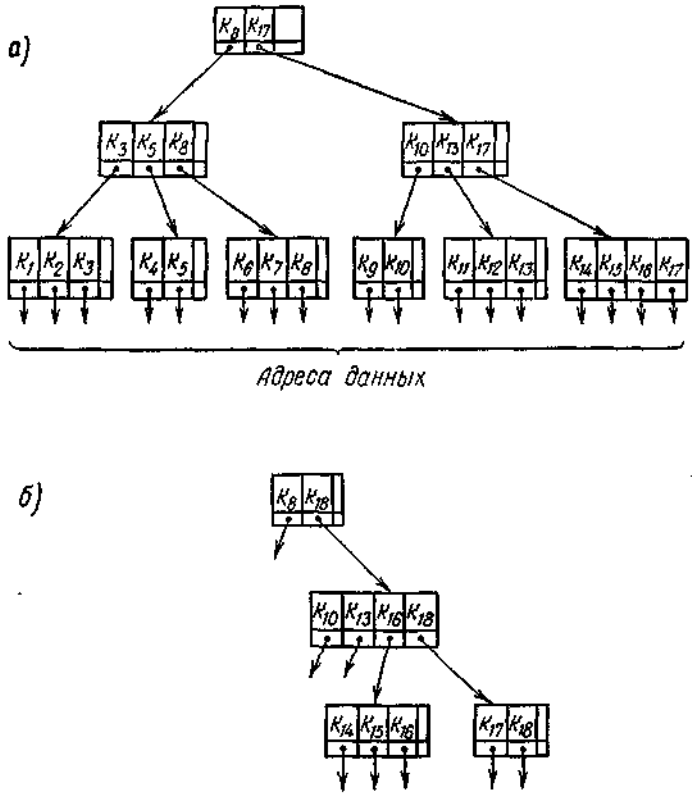


Рис. 20. Формирование (а) и корректировка (б) В-дерева

Нижний уровень В-дерева образуют страницы, заполненные К-индексами. Обычно в каждой странице оставляется свободная память в расчете на расширение числа индексов. Последний индекс каждой страницы поступает на страницу предпоследнего уровня В-дерева. Когда она будет почти заполнена индексами, последний из них поступит в страницу более высокого уровня и т. д. Поиск данных (например, значения *q)* начинается с корня В-дерева, который обычно имеет два разветвления. Предположим, что $K_5 < q < K_{17}$.
В этом случае выполняется переход по правой ветви на второй уровень и поиск в соответствующей странице. Будем считать, что $K_{10} < q < K_{12}$. Следовательно, надо спуститься на третий уровень по адресу, записанному в K_{13}. Третий уровень последний, и поиск в нем

происходит как в обычном К-индексе. Достоинство В-дерева состоит в его простом расширении.
При переполнении какой-либо страницы половина ее содержимого переходит в новую страницу, а на вышестоящем уровне появляется новый индекс. В рассматриваемом примере это произойдет при добавлении $K_{18} > K_{17}$. Состояние изменившихся страниц показано на рис.20, *б.*
Чтобы эффективно использовать высокоразвитый аппарат представления и обработки данных, реализованный в **реляционной, сетевой, иерархической и бинарной моделях данных,** необходимо выбрать метод организации данных для представления *n*-арных отношений, отношений между *N* СЕИ, веерных отношений и записей иерархической базы данных. Предполагается, что данные должны располагаться **на внешних запоминающих устройствах со страничной организацией памяти**. В качестве критериев выбора используются среднее время поиска при обработке запроса и среднее время реализации корректировки при ограничении на общий объем памяти, требуемой для представления данных. В большинстве случаев заранее неизвестно взаимное расположение страниц, на которых располагается необходимая информация, и вместо расчета среднего времени приходится ограничиваться расчетом числа страниц, считываемых при поиске и корректировке одного значения.
В качестве вариантов представления *n*-арных отношений необходимо анализировать те структуры данных и их модификации, которые предназначены для обработки данных по нескольким ключевым признакам. Среди них

основной массив с инвертированным массивом к нему (см. табл. 1);

мультисписок (см. табл. 3);

последовательная структура данных с *n* общими деревьями для доступа по *n* ключам (см. рис. 14 для $n = 2$).

Одна запись массива или мультисписка соответствует строке отношения.
Эффективность указанных структур данных оценивается на примере реализации оператора выборки с формальной записью

$$R\,[(A_1 = a_1) \wedge (A_2 = a_2) \wedge \ldots \wedge (A_m = a_m)],$$

где *R* — имя отношения,
A_i — имя реквизита, входящего в структуру отношения *R;*
a_i — значение реквизита A_i;
$m \leq n$-— число реквизитов, для которых сформулированы условия выборки.

Если отношение реализовано с использованием инвертированного массива, выполнение оператора выборки потребует *m* обращений к инвертированному массиву, поскольку в условии оператора указаны имена *m* реквизитов. После обработки адресов, извлеченных из инвертированного массива, становятся известны адреса *d* записей из основного массива, удовлетворяющих условиям оператора выборки. Некоторые из этих записей могут располагаться на одних и тех же страницах, поэтому в среднем потребуется $C = D\,(1 - \mathbf{e}^{-d/D})$ обращений к основному массиву, где $\mathbf{e} \approx 2{,}73$; *D* - число страниц в основном массиве. Полученные *d* записей трансформируются в *d* строк отношения, которое является результатом выборки. Суммарное число обращений к внешней памяти составит

$$C_1 = m + C = m + D\,(1 - \mathrm{e}^{-d/D}).$$

Если для представления отношения используется мультисписок, то сначала по известным длинам списков определяется самый короткий из списков, необходимых для реализации выборки (что требует чтения одной страницы). Пусть самый короткий список содержит *f* записей, тогда по аналогии с предыдущим выражением суммарное число обращений к внешней памяти равно:

$$C_2 = 1 + D\,(1 - \mathrm{e}^{-f/D}).$$

В третьем варианте представления отношения общие деревья требуют для своего размещения больше страниц, чем инвертированный массив, так как для организации разветвлений п дереве необходимы дополнительные адреса связи. Поэтому для обращения к *m* общим деревьям расходуется $m_0 > m$ чтений страниц. В остальном алгоритмы третьего и первого вариантов аналогичны, и общее число чтений составит

$$C_3 = m_0 + D\,(1 - \mathrm{e}^{-d/D}).$$

Таблица 7

ПРЕП	ДИСЦ	ГРУП
$П_1$	$Д_1$	$Г_2$
$П_1$	$Д_2$	$Г_1$
$П_2$	$Д_1$	$Г_1$
$П_2$	$Д_3$	$Г_2$
$П_3$	$Д_2$	$Г_3$
$П_3$	$Д_2$	$Г_4$

Сравнение значений C_1 и C_3 показывает, что $C_1 \leq C_3$ и первый вариант всегда не хуже третьего. Сравнить так же просто C_1 и C_2 не удается, поскольку зависимость между *f* и *d* обычно неизвестна.

Однако если условие оператора выборки определяет значения небольшого числа реквизитов *(m* невелико), то первый вариант представления отношения лучше, чем второй, по критерию минимального числа обращений к внешней памяти.

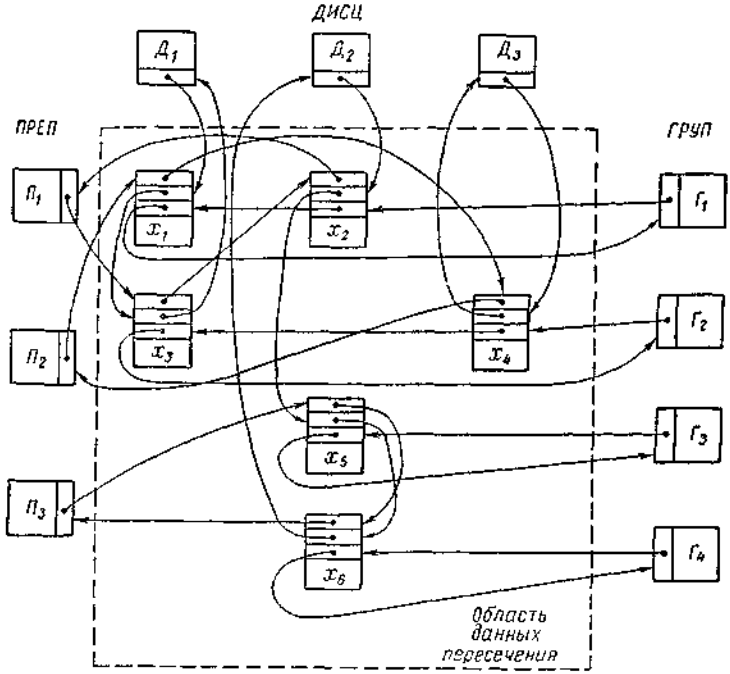


Рис. 21. Организация данных для представления в памяти отношения между СЕИ ПРЕП, ДИСЦ, ГРУП

Выбор наилучшего представления отношения для реализации других операторов реляционной алгебры изучен пока недостаточно.
При выборе метода организации данных для отношения между N СЕИ и веерного отношения предпочтение обычно отдается различным модификациям строки главным образом потому, что число взаимосвязей данного значения СЕИ с остальными сильно меняется от значения к значению и заранее неизвестно.Организация данных для отношения между тремя СЕИ, связи которых зафиксированы в табл. 7 вместе с данными пересечения x_i, определяющими фрагмент расписания занятий в вузе, приведена на рис. 21. Значения СЕИ ПРЕП содержат информацию об отдельных преподавателях, значения СЕИ ДИСЦ— об учебных дисциплинах и значения СЕИ ГРУП — о студенческих группах. Данные пересечения (обозначены x_i) уточняют

время и место проведения занятий. Для каждого значения каждой СЕИ организуется кольцевая строка. В кольцо связываются те элементы данных пересечения, которые имеют одинаковое значение одной из трех СЕИ. Число кольцевых строк равно суммарному числу значений во всех составных единицах информации.

При определении структуры данных для веерного отношения остановимся сначала на размещении в памяти одного веера, например $\{a_1, b_{11}, b_{12}, b_{13}\}$. Указанные в веере значения связываются в двунаправленную кольцевую строку, и дополнительно значения b_{11}, b_{12}, b_{13} содержат адрес связи на a_1 (рис. 22).

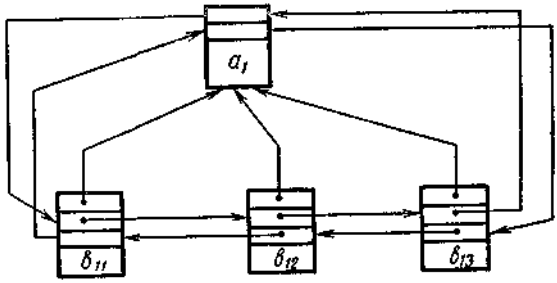


Рис. 22. Организация данных для представления в памяти веерного отношения

При такой организации адресов связи каждому оператору, манипулирующему с веером, соответствует цепочка адресов связи, специально предназначенная для его быстрой реализации. Вееры, образующие веерное отношение, могут объединяться в массивы с записями переменной длины (одна запись хранит информацию из одного веера) либо эти записи можно разместить в памяти с помощью адресной функции. Если при вычислениях из веерного отношения извлекается до 5% вееров, то второй способ предпочтительнее, поскольку обеспечивает минимальное время поиска вееров. Этот последний вывод справедлив и для организации записей иерархической базы данных в памяти ЭВМ.

При реализации систем машинной обработки информации чаще используются простые методы организации данных, например линейные методы для обрабатываемых данных выбираются чаще, чем нелинейные. Надо также отметить, что разработка любого нового класса запоминающих устройств вносит значительные коррективы в систему критериев методов организации данных, в алгоритмы формирования и обработки данных.

Литература

1. Ахо А., Холкрофт Д., Ульман Д. Структуры данных и алгоритмы. – Вильямс 2000.
2. Бен-Ари М. Языки программирования. Практический сравнительный анализ. – М.: Мир 2000.
3. Вирт Н. Алгоритмы и структуры данных. – Невский Диалект 2001.
4. Зубов В.С. Структуры и методы обработки данных. Практикум в среде Delphi – Филинъ 2004.
5. Кнут Д. Искусство программирования для ЭВМ. Тома 1-3. – Вильямс 2000.
6. Кубенский А. Создание и обработка структур данных в примерах на Java. – BHV-СПб 2003.
7. Роберт Седжвик. Фундаментальные алгоритмы на С. Части 1-5. – Диасофт 2003.
8. Таланов В.А., Алексеев В.Е. Графы и алгоритмы. Структуры данных. Модели вычислений. – Бином 2006.

- *Robin Milner* A Theory of Type Polymorphism in Programming (англ.). — Jcss, 1978. — Vol. 17. — P. 348–375.
- *Stavros Macrakis* Safety and power (англ.) // ACM SIGSOFT Software Engineering Notes. — 1982. — Vol. 7, iss. 2, no. April. — P. 25–26. — DOI:10.1145/1005937.1005941.
- *Luca Cardelli, Peter Wegner* On Understanding Types, Data Abstraction, and Polymorphism (англ.). — ACM

Computing Surveys, 1985. — P. 471-523. — ISSN 0360-0300.
- *Альфред Ахо, Рави Сети, Джеффри Ульман.* Компиляторы: принципы, технологии и инструменты. — Addison-Wesley Publishing Company, Издательский дом «Вильямс», 1985, 2001, 2003. — 768 с. — ISBN 5-8459-0189-8 (рус.), 0-201-10088-6 (ориг.).
- *Лука Карделли*[en] Typeful programming (англ.) // IFIP State-of-the-Art Reports. — Springer-Verlag, 1991. — Iss. Formal Description of Programming Concepts. — P. 431–507.
- *Andrew W. Appel* A Critique of Standard ML (англ.). — Princeton University, revised version of CS-TR-364-92, 1992.
- *Andrew K. Wright, Matthias Felleisen*[en] A Syntactic Approach to Type Soundness (англ.) // Information and Computation. — 1992. — Vol. 115, iss. 1. — P. 38–94. — DOI:10.1006/inco.1994.1093.
- *Lawrence C. Paulson.* ML for the Working Programmer. — 2nd. — Cambridge University Press, 1996. — 492 с. — ISBN 0-521-57050-6 (твёрдый переплёт), 0-521-56543-X (мягкий переплёт).
- *Pierce, Benjamin C.* Types and Programming Languages. — MIT Press, 2002. — ISBN 0-262-16209-1.
    - Перевод на русский язык: *Пирс Б.* Типы в языках программирования. — Добросвет, 2012. — 680 с. — ISBN 978-5-7913-0082-9.
- *John C. Mitchell.* Concepts in Programming Languages. — Cambridge University Press, 2004. — ISBN 0-511-04091-1 (eBook in netLibrary); 0-521-78098-5 (hardback).
- *Harper.* Practical Foundations for Programming Languages. — version 1.37 (revised 01.11.2014). —

licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License., 2012. — 544 с.
- *Vijay Saraswat*[en]. Java is not type-safe.

объектно-ориентированного и модульного программирования.