Никлаус Вирт

Алгоритмы и структуры данных

Новая версия для Оберона + CD

Перевод с английского под редакцией
доктора физ-мат. наук, Ткачева Ф. В.

ДМК Пресс
издательство
Москва, 2010

УДК 32.973.26-018.2
ББК 004.438
В52

Никлаус Вирт

В52 Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010. – 272 с.: ил.

ISBN 978-5-94074-584-6

В классическом учебнике тьюринговского лауреата Н.Вирта аккуратно, на тщательно подобранных примерах прорабатываются основные темы алгоритмики – сортировка и поиск, рекурсия, динамические структуры данных.

Перевод на русский язык выполнен заново, все рассуждения и программы проверены и исправлены, часть примеров по согласованию с автором переработана с целью максимального прояснения их логики (в том числе за счет использования цикла Дейкстры). Нотацией примеров теперь служит Оберон/Компонентный Паскаль – наиболее совершенный потомок старого Паскаля по прямой линии.

Все программы проверены и работают в популярном варианте Оберона – системе Блэкбокс, и доступны в исходниках на прилагаемом CD вместе с самой системой и дополнительными материалами.

Большая часть материала книги составляет необходимый минимум знаний по алгоритмике не только для программистов-профессионалов, но и любых других специалистов, активно использующих программирование в работе.

Книга может быть использована как учебное пособие при обучении будущих программистов, начиная со старшеклассников в профильном обучении, а также подходит для систематического самообразования.

Содержание компакт-диска:

Базовая конфигурация системы Блэкбокс с коллекцией модулей, реализующих программы из книги.
Базовые инструкции по работе в системе Блэкбокс.
Полный перевод документации системы Блэкбокс на русский язык.
Конфигурация системы Блэкбокс для использования во вводных курсах программирования в университетах.
Конфигурация системы Блэкбокс для использования в школах (полная русификация меню, сообщений компилятора, с возможностью использования ключевых слов на русском и других национальных языках).
Доклады участников проекта Информатика-21 по опыту использования системы Блэкбокс в обучении программированию.
Оригинальные дистрибутивы системы Блэкбокс 1.5 (основной рабочий) и 1.6rc6.
Инструкции по работе в Блэкбоксе под Linux/Wine.
Дистрибутив оптимизирующего компилятора XDS Oberon (версии Linux и MS Windows).
OberonScript – аналог JavaScript для использования в Web-приложениях.

ISBN 0-13-022005-9 (анг.)

ISBN 978-5-94074-584-6

Содержание

О новой версии классического учебника Никлауса Вирта

Новая версия учебника Н. Вирта «Алгоритмы и структуры данных» отличается от английского прототипа [1] сильнее, чем просто исправлением многочисленных опечаток и огрехов, накопившихся в процессе тридцатилетней эволюции книги. Объясняется это целями автора и переводчика при работе над книгой в контексте проекта «Информатика-21» [2], который, опираясь на обширный совокупный опыт ряда высококвалифицированных специалистов (см. списки консультантов и участников на сайте проекта [2]), ставит задачу создания единой системы вводных курсов информатики и программирования, охватывающей учащихся примерно от 5-го класса общей средней школы по 3-й курс университета. Такая система должна иметь образцом и дополнять уникальную российскую систему математического образования. Это предполагает наличие стержня общих курсов, составляющих единство без внутренних технологических барьеров (которые приводят, среди прочего, к недопустимым потерям дефицитного учебного времени) и лишь варьирующихся в зависимости от специализации, вместе с надстройкой из профессионально ориентированных курсов, опирающихся на этот стержень в отношении базовых знаний учащихся. Такая система подразумевает наличие качественных учебников (первым из которых имеет шанс стать данная книга), «говорящих» на общем образцовом языке программирования. Естественный кандидат на роль такого общего языка – Оберон/Компонентный Паскаль. Подробней об Обероне речь пойдет ниже, здесь только скажем, что Паскаль (использованный в первом издании данной книги 1975 г.), Модулу-2 (использованную во втором издании, переведенном на русский язык в 1989 г. [3]) и Оберон (использованный в данной версии) логично рассматривать соответственно как альфа-, бета- и окончательную версию одного и того же языка. Использование Оберона – самое очевидное отличие данной версии книги от предыдущего издания.

В контекст идеи о единой системе вводных курсов вписывается и узкая задача, решавшаяся новой версией учебника, – дать небольшое продуманное пособие, в котором аккуратно, но не топя читателя в болоте второстепенных деталей, прорабатывались бы традиционные темы классической алгоритмики, для полного обсуждения которых нет времени в спецкурсе, читаемом переводчиком с 2001 г. на физфаке МГУ в попытке обеспечить хотя бы минимум культуры программирования у будущих аспирантов. Здесь требуется «отлаженный» текст, пригодный для самостоятельной работы студентов. С точки зрения содержания, лучшим кандидатом на эту роль оказался прототип [1].

Что двойное переделывание программ и рассуждений в тексте (с Паскаля на Модулу-2 и затем на Оберон) не прошло безнаказанно, само по себе неудивительно. Однако затруднения, возникшие при верификации программ и текста, хотя и были преодолены, все же показались чрезмерными. Поэтому, и ввиду учебного назначения книги, встал ребром вопрос о необходимости доработки примеров. Предложения переводчика были одобрены автором на совместной рабочей сессии в апреле сего года и реализованы непосредственно в данном переводе (при первой возможности соответствующие изменения будут внесены и в прототип [1]).

Во-первых, алгоритмы поиска образца в тексте переписаны в терминах цикла Дейкстры (многоветочный while [4]). Эта фундаментальная и мощная управляющая структура поразительным образом до сих пор не представлена в распространенных языках программирования, поэтому ей посвящено новое приложение С. Раздел 1.9, в который теперь выделены эти алгоритмы, будет неплохой иллюстрацией реального применения цикла Дейкстры. Вторая группа заметно измененных программ – алгоритмы с возвратом в главе 3, в которых теперь эксплицировано применение линейного поиска и, благодаря этому, тривиализована верификация. Такое прояснение рекурсивных комбинаторных алгоритмов является довольно общим. Обсуждались – но были признаны в данный момент нецелесообразными – модификации и некоторых других программ.

Надо заметить, что программистский стиль автора вырабатывался с конца 1950-х гг., когда проблема эффективности программ висела над головами программистов дамокловым мечом, и за несколько лет до того, как Дейкстра опубликовал систематический метод построения программ [4]. В старых версиях книги заметна рефлекторная склонность к оптимизации до полного прояснения логики программ, что затрудняло эффективное применение формальной техники. Это легко объяснить: Н. Вирт осваивал только еще формирующиеся систематические методы, непосредственно участвуя в процессе создания программирования как академической дисциплины, версия за версией улучшая свои учебники.

Но и через четверть века после последней существенной переделки учебника автором аналогичная склонность к преждевременной оптимизации при не просто не вполне уверенной, а напрочь отсутствующей формальной технике – и, как следствие, запутанные циклы, – характерные черты стиля «широких программистских масс»! В профессиональных интернет-форумах до сих пор можно найти позорные дискуссии о том, нужно ли учиться писать циклы по Дейкстре, – и это в лучшем случае. Если же вообразить себе весь окружающий нас непрерывно растущий массив софта, от которого наша жизнь зависит все больше, то впору впасть в депрессию: *Quo usque tandem, Catilina*? – Сколько еще нужно *десятилетий*, чтобы система образования вышла, наконец, на уровень, давным-давно достигнутый наукой? Во всяком случае, ясно, что едва ли не главная причина проблемы – хаос, царящий в системе ИТ-образования, тормозящий создание и распространение качественных методик и поддерживаемый, среди прочего, корыстными интересами «монстров» индустрии.

Здесь уместно сказать о языке Оберон/Компонентный Паскаль, пропагандируемом в качестве общей платформы для предполагаемой единой системы курсов

программирования. Оберон – последний большой проект Никлауса Вирта, выдающегося инженера, ученого и педагога, вместе с Бэкусом, А. Ершовым, Дейкстрой, Хоором и другими пионерами компьютерной информатики превратившего программирование в систематическую дисциплину и лучше всего известного созданием серии все более совершенных языков программирования – Паскаля (1970), Модулы-2 (1980) и наконец Оберона (1988, 2007). В этих языках отражалось все более полное понимание проблематики эффективного программирования. Языки эти сохраняют идейную и стилевую преемственность, и коммерсант, озабоченный сохранением доли рынка, не назвал бы их по-разному (ср. зоопарк бейсиков). Чтобы подчеркнуть эту преемственность, самому популярному диалекту Оберона было возвращено законное фамильное имя – Компонентный Паскаль.

Оберон/Компонентный Паскаль унаследовал лучшие черты старого доброго Паскаля и добавил к ним промышленный опыт Модулы-2 (на которой программируются, например, российские спутники связи [5]), а также выверенный минимум средств объектно-ориентированного программирования. Принципальное достижение – удалось наконец добиться герметичности системы типов (теперь ее нельзя обойти средствами языка даже при работе с указателями). Это обеспечило возможность автоматического управления памятью (сбора мусора; до Оберона сбор мусора оставался прерогативой динамических языков – функциональных, скриптовых и т. п.) В результате диапазон эффективного применения Оберона, похоже, шире, чем у любого другого языка: это и вычислительные приложения, и системы управления любого масштаба (от беспилотников весом в 1 кг до грандиозных каскадов ГЭС), и, например, задачи символической алгебры с предельно динамичными структурами данных.

Особо следует остановиться на минимализме Оберона. Традиционно разработчики сосредоточиваются на том, чтобы снабдить свои языки, программы, библиотеки «богатым набором средств» – ведь так легче привлечь клиента, надеющегося побыстрее найти готовое решение для своих прикладных нужд. Погоня за «богатым набором средств» оборачивается ущербом качеству и надежности системы. Вместе с коммерческими соображениями это приводит к тому, что получается большая закрытая сложная система с вроде бы богатым набором средств, но хромающей надежностью и ограниченной расширяемостью, так что если пользователь сталкивается с нестандартной ситуацией в своих приложениях (что случается сплошь и рядом – ведь разнообразие реального мира превосходит любое воображение писателей библиотек), то он оказывается в тупике.

Н. Вирт еще со времен Паскаля, созданного в пику фантазийному Алголу-68 [6], пошел другим путем. Его гамбит заключался в том, чтобы, отказавшись от включения в язык *максимума* средств на все случаи жизни, тщательнейшим образом выделить *минимум* реально ключевых средств, – обязательно включив в этот минимум все, что нужно для безболезненной, неограниченной расширяемости программных систем, – и добиться высоконадежной реализации такого ядра. Этот замысел был с блеском реализован Н. Виртом и его соратником Ю. Гуткнехтом в проекте Оберон [7]. Минимализм и уникальная надежность Оберона

заставляют вспомнить автомат Калашникова. При этом вся мощь Оберона оказывается открытой даже программистам-непрофессионалам – физикам, инженерам, лингвистам.., занятым программированием изрядную долю своего рабочего времени.

Для преподавателя важно, что в Обероне достигнуты ортогональность и свободная комбинируемость языковых средств, смысловая прозрачность, а также беспрецедентно малый для столь мощного языка размер (см. полное описание синтаксиса в приложении В, а также обсуждение в [8]). В этом отношении Оберон побеждает за явным преимуществом традиционные промышленные языки, пресловутая избыточная сложность которых оказывается источником своего рода ренты, взимаемой с остального мира. Оберон скромно уходит в тень при рассмотрении любой языково-неспецифичной темы – от введения в алгоритмику до принципов компиляции и программной архитектуры. А после постановки базовой техники программирования на Обероне изучение промышленных языков зачастую сводится к изучению способов обходить дефекты их дизайна. Если уже старый Паскаль оказался настолько удачной платформой для обучения программированию, что принес своему автору высшую почесть в компьютерной информатике – премию им. Тьюринга, то понятно, что буквально вылизанный Оберон/Компонентный Паскаль называют уже «практически идеальной» платформой для обучения программированию.

Имея в виду исключительные педагогические достоинства Оберона, для всех примеров программ, приведенные в книге, обеспечена воспроизводимость в системе программирования для Компонентного Паскаля, известной как Блэкбокс (BlackBox Component Builder [9]). Это пулярный вариант Оберона, созданный для работы в распространенных операционных системах. Конфигурации Блэкбокса для использования в школе и университете доступны на сайте проекта «Информатика-21» [2]. Открытый, бесплатный и безупречно современный Блэкбокс оказывается естественной заменой устаревшему Турбо Паскалю – заменой тем более привлекательной, что, несмотря на минимализм и благодаря автоматическому управлению памятью, это более мощный инструмент, чем промышленные системы программирования на диалектах старого Паскаля. Краткое описание возможностей Блэкбокса с точки зрения использования в школьных курсах можно найти в статье [10].

Важное приложение к книге – полный комплект программ, представленных в тексте учебника, в виде, готовом к выполнению. Программы оформлены в отдельных модулях вместе с необходимыми вспомогательными процедурами, и все такие модули собраны в папке ADru/Mod/, которая должна лежать внутри основной папки Блэкбокса (следует иметь в виду, что файлы с расширением *.odc должны читаться из Блэкбокса). Читатель без труда разберется с компиляцией и запуском программ по комментариям в модулях, читая модули в том порядке, в каком они встречаются в тексте книги (или в лексикографическом порядке имен файлов). В тексте книги в начальных строках каждого законченного программного примера справа указано имя соответствующего модуля. Например, комментарий (*ADruS18_Поиск*) означает, что данная программа содержится в модуле

ADruS18_Поиск, который в соответствии с правилами Блэкбокса хранится в файле ADru/Mod/S18_Поиск.odc. При этом речь идет о программе из раздела 1.8, а необязательный суффикс "_Поиск" служит удобству ориентации. Вся папка ADru в составе Блэкбокса имеется на диске, если диск приложен к книге, либо может быть скачана с адреса [11].

Наконец, несколько слов о собственно переводе. Старый перевод [3] был выполнен, что называется, из общих соображений. Но совсем другое дело – иметь в виду конкретных студентов, не обязательно будущих профессиональных программистов, пытающихся за минимальное время овладеть основами программирования. Поэтому в новом переводе были предприняты особые усилия, чтобы избежать размывания смысла из-за неточностей, неизбежно вкрадывающихся при неполном понимании переводчиком оригинала (ср. примечание на с. 110 в главе о сортировках в [3], где выражена надежда, что «сам читатель разберется, что хотел сказать автор»). Например, при более-менее прямолинейной пофразовой интерпретации малейшая неточность способна развалить смысл лаконичного текста Вирта из-за того, например, что после перевода могут перестать быть однокоренными слова, благодаря которым только и обеспечивалась смысловая связь между предложениями в оригинале. Поэтому добиться полного сохранения смысла при переводе оказалось проще, выполнив его с нуля.

В отношении терминологии переводам специалистов было отдано должное. Вслед за Д. Б. Подшиваловым [3] мы используем прилагательные «массивовый», «последовательностный» и «записевый». Решающий довод в пользу таких прилагательных – они естественно вписываются в грамматическую систему русского языка, чем обеспечивается необходимая гибкость выражения.

Однако даже в отношении терминологии переводы по компьютерной тематике часто демонстрируют неполное понимание существенных деталей английской грамматики. Например, при использовании существительного в качестве определения в препозиции (что, кстати, не эквивалентно русской конструкции, выражаемой родительным падежом) множественное число может нейтрализоваться, и при переводе на русский его иногда нужно восстанавливать. Так, path length должно переводиться не как «длина пути», а как «длина путей», что, между прочим, прямо соответствует математическому определению и ощутимо помогает понимать рассуждения. Optimal search tree – «оптимальное дерево поиска», а не «дерево оптимального поиска». Advanced sort algorithms – «эффективные алгоритмы сортировки», потому что буквальное значение advanced в данном случае давно нейтрализовано. Переводить на русский язык двумя словами специфичные для стилистики английского языка синонимичные пары вроде «methods and techniques» обычно неразумно. И так далее. Масса подобных неточностей снижает удобочитаемость текста и затемняет и без того непростой смысл оригинала.

Хотя по конкретным стилистическим вопросам копья можно ломать до бесконечности, все же хочется надеяться, что предпринятые усилия в основном достигли цели – не потерять точный смысл английского «исходника» этого выдержавшего проверку временем прекрасного учебника.

Троицк, Московская обл., июль 2009 Ф. В. Ткачев

[1] Wirth N. Algorithms and Data Structures. Oberon version: 2004 //http://www.inr.ac.ru/~info21/pdf/AD.pdf
[2] Информатика-21: Международный общественный научно-образовательный проект // http://www.inr.ac.ru/~info21/
[3] Н. Вирт. Алгоритмы и структуры данных / пер. с англ. Д. Б. Подшивалова. – М.: Мир, 1989.
[4] Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978.
[5] Koltashev A. A., in: Lecture Notes in Computer Science 2789. – Springer-Verlag, 2003.
[6] Кто такой Никлаус Вирт? // http://www.inr.ac.ru/~info21/wirth/wirth.htm
[7] Wirth N. and Gutknecht J. Project Oberon. – Addison-Wesley, 1992.
[8] Свердлов С. В. Языки программирования и методы трансляции. – СПб.: Питер, 2007.
[9] http://www.oberon.ch/blackbox.html
[10] Ильин А. С. и Попков А. И. Компонентный Паскаль в школьном курсе информатики // http://inf.1september.ru/article.php?ID=200800100
[11] http://www.inr.ac.ru/~info21/ADru/

Предисловие

В последние годы признано, что умение создавать программы для вычислительных машин является залогом успеха во многих инженерных проектах и что дисциплина программирования может быть объектом научного анализа и допускает систематическое изложение. Программирование из ремесла превратилось в академическую дисциплину. Первые выдающиеся результаты на этом пути получены Дейкстрой (E. W. Dijkstra) и Хоором (C. A. R. Hoare). «Заметки по структурному программированию» Дейкстры [1] позволили взглянуть на программирование как на объект научного анализа, бросающий вызов человеческому интеллекту, а слова *структурное программирование* дали название «революции» в программировании. Работа Хоора «Аксиоматические основы программирования» [2] продемонстрировала, что программы допускают точный анализ, основанный на математических рассуждениях. И обе статьи убедительно доказывают, что многих ошибок в программах можно избежать, если программисты будут систематически применять методы и приемы, которые ранее применялись лишь интуитивно и часто неосознанно. Эти статьи сосредоточили внимание на построении и анализе программ, или, точнее говоря, на структуре алгоритмов, представленных текстом программы. При этом вполне очевидно, что систематический научный подход к построению программ уместен прежде всего в случае больших, непростых программ, работающих со сложными наборами данных. Отсюда следует, что методология программирования должна включать в себя все аспекты структурирования данных. В конце концов, программы суть конкретные формулировки абстрактных алгоритмов, основанные на конкретных представлениях и структурах данных. Выдающийся вклад в наведение порядка в огромном разнообразии терминологии и понятий, относящихся к структурам данных, сделал Хоор в статье «О структурной организации данных» [3]. В этой работе продемонстрировано, что нельзя принимать решения о структуре данных без учета того, какие алгоритмы применяются к данным, и что, обратно, структура и выбор алгоритмов часто сильно зависят от стуктуры обрабатываемых данных. Короче говоря, задачу построения программ нельзя отделять от задачи структурирования данных.

Но данная книга начинается главой о структурах данных, и для этого есть две причины. Во-первых, интуитивно ощущается, что данные предшествуют алгоритмам: нужно иметь некоторые объекты до того, как можно будет что-то с ними делать. Во-вторых, эта книга предполагает, что читатель знаком с основными понятиями программирования. Однако в соответствии с разумной традицией вводные курсы программирования концентрируют внимание на алгоритмах, работающих с относительно простыми структурами данных. Поэтому уместно посвятить вводную главу структурам данных.

На протяжении всей книги, включая главу 1, мы следуем теории и терминологии, развитой Хоором и реализованной в языке программирования *Паскаль* [4]. Сущность теории – в том, что данные являются прежде всего абстракциями реальных явлений и их предпочтительно формулировать как абстрактные струк-

туры безотносительно к их реализации в распространенных языках программирования. В процессе построения программы представление данных постепенно уточняется – в соответствии с уточнением алгоритма, – чтобы все более и более удовлетворить ограничениям, налагаемым имеющейся системой программирования [5]. Поэтому мы постулируем несколько основных структур данных, называемых фундаментальными. Очень важно, что это конструкции, которые достаточно легко реализовать на реальных компьютерах, ибо только в этом случае их можно рассматривать как истинные элементарные составляющие реального представления данных, появляющиеся как своего рода молекулы на последнем шаге уточнения описания данных. Это запись, массив (с фиксированным размером) и множество. Неудивительно, что эти базовые строительные элементы соответствуют математическим понятиям, которые также являются фундаментальными.

Центральный пункт этой теории структур данных – разграничение *фундаментальных* и *сложных* структур. Первые суть молекулы, – сами построенные из атомов, – из которых строятся вторые. Переменные, принадлежащие одному из таких фундаментальных видов структур, меняют только свое значение, но никогда не меняют ни свое строение, ни множество своих допустимых значений. Как следствие – размер занимаемой ими области памяти фиксирован. «Сложные» структуры, напротив, характеризуются изменением во время выполнения программы как своих значений, так и строения. Поэтому для их реализации нужны более изощренные методы. В этой классификации последовательность оказывается гибридом. Конечно, у нее может меняться длина; но такое изменение структуры тривиально. Поскольку последовательности играют поистине фундаментальную роль практически во всех вычислительных системах, их обсуждение включено в главу 1.

Во второй главе речь идет об алгоритмах сортировки. Там представлено несколько разных методов, решающих одну и ту же задачу. Математическое изучение некоторых из них показывает их преимущества и недостатки, а также подчеркивает важность теоретического анализа при выборе хорошего решения для конкретной задачи. Разделение на методы сортировки массивов и методы сортировки файлов (их часто называют внутренней и внешней сортировками) демонстрирует решающее влияние представления данных на выбор алгоритмов и на их сложность. Теме сортировки уделяется такое внимание потому, что она представляет собой идеальную площадку для иллюстрации очень многих принципов программирования и ситуаций, возникающих в большинстве других приложений. Похоже, что курс программирования можно было бы построить, используя только примеры из темы сортировки.

Другая тема, которую обычно не включают во вводные курсы программирования, но которая играет важную роль во многих алгоритмических решениях, – это рекурсия. Поэтому третья глава посвящена рекурсивным алгоритмам. Здесь показывается, что рекурсия есть обобщение понятия цикла (итерации) и что она является важным и мощным понятием программирования. К сожалению, во многих учебниках программирования она иллюстрируется примерами, для которых было бы достаточно простой итерации. Мы в главе 3, напротив, сосредоточим внимание на нескольких задачах, для которых рекурсия дает наиболее естественную формулировку решения, тогда как использование итерации привело бы к за-

путанным и громоздким программам. Класс алгоритмов с возвратом – отличное применение рекурсии, но самые очевидные кандидаты для применения рекурсии – это алгоритмы, работающие с данными, структура которых определена рекурсивно. Эти случаи рассматриваются в последних двух главах, для которых, таким образом, третья закладывает фундамент.

В главе 4 рассматриваются динамические структуры данных, то есть такие, строение которых меняется во время выполнения программы. Показывается, что рекурсивные структуры данных являются важным подклассом часто используемых динамических структур. Хотя рекурсивные определения возможны и даже естественны в этих случаях, на практике они обычно не используются. Вместо них используют явные ссылочные или указательные переменные. Данная книга тоже следует подобному подходу и отражает современный уровень понимания предмета: глава 4 посвящена программированию с указателями, списками, деревьями и содержит примеры с даже еще более сложно организованными данными. Здесь речь идет о том, что обычно (хотя и не совсем правильно) называют обработкой списков. Немало места уделено построению деревьев и, в частности, деревьям поиска. Глава заканчивается обсуждением так называемых хэш-таблиц, которые часто используют вместо деревьев поиска. Это дает возможность сравнить два принципиально различных подхода к решению часто возникающей задачи.

Программирование – это *конструирование*. Как вообще можно учить изобретательному конструированию? Можно было бы попытаться из анализа многих примеров выделить элементарные композиционные принципы и представить их систематическим образом. Но программирование имеет дело с задачами огромного разнообразия и часто требует серьезных интеллектуальных усилий. Ошибочно думать, что обучить ему можно, просто дав некий список рецептов. Но тогда в нашем арсенале методов обучения остаются только тщательный подбор и изложение образцовых примеров. Естественно, не следует ожидать, что изучение примеров будет равно полезным для разных людей. При таком подходе многое зависит от самого учащегося, от его прилежания и интуиции. Это особенно справедливо для относительно сложных и длинных примеров программ. Такие примеры включены в книгу не случайно. Длинные программы доминируют в практике программирования, и они гораздо больше подходят для демонстрации тех трудно определяемых, но существенных свойств, которые называют стилем и хорошей структурой. Они также должны послужить упражнениями в искусстве чтения программ, которым часто пренебрегают в пользу написания программ. Это главная причина того, почему в качестве примеров используются целиком довольно большие программы. Читатель имеет возможность проследить постепенную эволюцию программы и увидеть ее состояние на разных шагах, так что процесс разработки предстает как пошаговое уточнение деталей. Считаю, что важно показать программу в окончательном виде, уделяя достаточно внимания деталям, так как в программировании дьявол прячется в деталях. Хотя изложение общей идеи алгоритма и его анализ с математической точки зрения могут быть увлекательными для ученого, по отношению к инженеру-практику ограничиться только этим было бы нечестно. Поэтому я строго придерживался правила давать окончательные программы на таком языке, на котором они могут быть реально выполнены на компьютере.

Разумеется, здесь возникает проблема поиска нотации, которая одновременно позволяла бы выполнить программу на вычислительной машине и в то же время была бы достаточно машинно независимой, чтобы ее можно было включать в подобный текст. В этом отношении не удовлетворительны ни широко используемые языки, ни абстрактная нотация. Язык Паскаль представляет собой подходящий компромисс; он был разработан именно для этой цели и поэтому используется на протяжении всей книги. Программы будут понятны программистам, знакомым с другими языками высокого уровня, такими как Алгол 60 или PL/1: смысл нотации Паскаля объясняется в книге по ходу дела. Однако некоторая подготовка все же могла бы быть полезной. Книга «*Систематическое программирование*» [6] идеальна в этом отношении, так как она тоже основана на нотации Паскаля. Однако следует помнить, что настоящая книга не предназначена быть учебником языка Паскаль; для этой цели есть более подходящие руководства [7].

Данная книга суммирует – и при этом развивает – опыт нескольких курсов программирования, прочитанных в Федеральном политехническом институте (ETH) в Цюрихе. Многими идеями и мнениями, представленными в этой книге, я обязан дискуссиям со своими коллегами в ETH. В частности, я хотел бы поблагодарить г-на Г. Сандмайра за внимательное чтение рукописи, а г-жу Хайди Тайлер и мою жену за тщательную и терпеливую перепечатку текста. Я должен также упомянуть о стимулирующем влиянии заседаний рабочих групп 2.1 и 2.3 ИФИПа, и в особенности многих дискуссий, которые мне посчастливилось иметь с Э. Дейкстрой и Ч. Хоором. Наконец, нужно отметить щедрость ETH, обеспечившего условия и предоставившего вычислительные ресурсы, без которых подготовка этого текста была бы невозможной.

Цюрих, август 1975 Н. Вирт

[1] Dijkstra E. W., in: Dahl O-J., Dijkstra E. W., Hoare C. A. R. Structured Programming. F. Genuys, Ed., New York, Academic Press, 1972. P. 1–82 (имеется перевод: Дейкстра Э. Заметки по структурному программированию, в кн.: Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1975. С. 7–97).

[2] Hoare C. A. R. *Comm. ACM*, 12, No. 10 (1969), 576–83.

[3] Hoare C. A. R., in Structured Programming [1]. P. 83-174 (имеется перевод: Хоор К. О структурной организации данных, в кн. [1]. С. 98–197).

[4] Wirth N. The Programming Language Pascal. *Acta Informatica*, 1, No. 1 (1971), 35–63.

[5] Wirth N. Program Development by Stepwise Refinement. *Comm. ACM*, 14, No. 4 (1971), 221–27.

[6] Wirth N. Systematic Programming. Englewood Cliffs, N. J. Prentice-Hall, Inc., 1973 (имеется перевод: Вирт Н. Систематическое программирование. Введение. – М.: Мир, 1977).

[7] Jensen K. and Wirth N. PASCAL-User Manual and Report. Berlin, Heidelberg, New York; Springer-Verlag, 1974 (имеется перевод: Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка. – М.: Финансы и статистика, 1988).

Предисловие к изданию 1985 года

В этом новом издании сделано много улучшений в деталях, а также несколько более серьезных модификаций. Все они мотивированы опытом, приобретенным за десять лет после первого издания. Однако основное содержание и стиль текста не изменились. Кратко перечислим важнейшие изменения.

Главное изменение, повлиявшее на весь текст, касается языка программирования, использованного для записи алгоритмов. Паскаль был заменен на Модулу-2. Хотя это изменение не оказывает серьезного влияния на представление алгоритмов, выбор оправдан большей простотой и элегантностью синтаксиса Модулы-2, что часто приводит к большей ясности представления структуры алгоритма. Кроме того, было сочтено полезным использовать нотацию, которая приобретает популярность в довольно широком сообществе по той причине, что она хорошо подходит для разработки больших программных систем. Тем не менее тот очевидный факт, что Паскаль является предшественником Модулы, облегчает переход. Для удобства читателя синтаксис Модулы суммирован в приложении.

Как прямое следствие замены языка программирования был переписан раздел 1.11 о последовательной файловой структуре. В Модуле-2 нет встроенного файлового типа. В пересмотренном разделе 1.11 понятие последовательности как структуры данных представлено в более общем виде, и там также вводится набор программных модулей, которые явно реализуют идею последовательности конкретно в Модуле-2.

Последняя часть главы 1 является новой. Она посвящена теме поиска и, начиная с линейного и двоичного поиска, подводит к некоторым недавно изобретенным быстрым алгоритмам поиска строк. В этом разделе подчеркивается важность проверок промежуточных состояний (assertions) и инвариантов цикла для доказательства корректности представляемых алгоритмов.

Новый раздел о приоритетных деревьях поиска завершает главу, посвященную динамическим структурам данных. Эта разновидность деревьев была неизвестна во время выхода первого издания. Такие деревья допускают экономное представление и позволяют выполнять быстрый поиск по множествам точек на плоскости.

Целиком исключена вся пятая глава первого издания. Это сделано потому, что тема построения компиляторов стоит несколько в стороне от остальных глав и заслуживает более подробного обсуждения в отдельной книге.

Наконец, появление нового издания отражает прогресс, глубоко повлиявший на издательское дело в последние десять лет: применение компьютеров и изощренных алгоритмов для подготовки и автоматического форматирования документов. Эта книга была набрана и сформатирована автором с помощью компьютера Lilith и редактора документов Lara. Без этих инструментов книга не только стала бы дороже, но, несомненно, даже еще не была бы закончена.

Пало Альто, март 1985 г. Н. Вирт

Нотация

В книге используются следующие обозначения, взятые из работ Дейкстры.

В логических выражениях литера & обозначает конъюнкцию и читается как «и». Литера ~ обозначает отрицание и читается как «не». Комбинация литер or обозначает дизъюнкцию и читается как «или». Литеры **A** и **E**, набранные жирным шрифтом, обозначают кванторы общности и существования. Нижеследующие формулы определяют смысл нотации в левой части через выражение в правой. Интерпретация символа «...» в правых частях оставлена интуиции читателя.

Ai: $m \le i < n : P_i$ $\quad P_m \ \&\ P_{m+1} \ \&\ \ldots \ \&\ P_{n-1}$

Здесь P_i – некоторые предикаты, а формула утверждает, что выполняются все P_i для значений индекса i из диапазона от m до n, но не включая само n.

Ei: $m \le i < n : P_i$ $\quad P_m \text{ or } P_{m+1} \text{ or } \ldots \text{ or } P_{n-1}$

Здесь P_i – некоторые предикаты, а формула утверждает, что выполняются некоторые из P_i для каких-то значений индекса i из диапазона от m до n, но не включая само n.

Si: $m \le i < n : x_i = x_m + x_{m+1} + \ldots + x_{n-1}$

MIN i: $m \le i < n : x_i =$ минимальное среди значений $(x_m, \ldots, x_{n-1})$

MAX i: $m \le i < n : x_i =$ максимальное среди значений $(x_m, \ldots, x_{n-1})$

Глава 1

Фундаментальные структуры данных

1.1. Введение

Современные цифровые компьютеры были изобретены для выполнения сложных и длинных вычислений. Однако в большинстве приложений предоставляемая таким устройством возможность хранить и обеспечивать доступ к большим массивам информации играет основную роль и рассматривается как его главная характеристика, а возможность призводить вычисления, то есть выполнять арифметические действия, во многих случаях стала почти несущественной.

В таких приложениях большой массив обрабатываемой информации является в определенном смысле абстрактным представлением некоторой части реального мира. Информация, доступная компьютеру, представляет собой специально подобранный набор данных, относящихся к решаемой задаче, причем предполагается, что этот набор достаточен для получения нужных результатов. Данные являются абстрактным представлением реальности в том смысле, что некоторые свойства реальных объектов игнорируются, так как они несущественны для этой задачи. Поэтому абстракция – это еще и упрощение реальности.

В качестве примера можно взять файл с данными о служащих некоторой компании. Каждый служащий (абстрактно) представлен в этом файле набором данных, который нужен либо для руководства компании, либо для бухгалтерских расчетов. Такой набор может содержать некоторую идентификацию служащего, например имя и зарплату. Но в нем почти наверняка не будет несущественной информации о цвете волос, весе или росте.

Решая задачу с использованием компьютера или без него, необходимо выбрать абстрактное представление реальности, то есть определить набор данных, который будет представлять реальную ситуацию. Этот выбор можно сделать, руководствуясь решаемой задачей. Затем нужно определиться с представлением информации. Здесь выбор определяется средствами вычислительной установки. В большинстве случаев эти два шага не могут быть полностью разделены.

Выбор представления данных часто довольно сложен и не полностью определяется имеющимися вычислительными средствами. Делать такой выбор всегда нужно с учетом операций, которые нужно выполнять с данными. Хороший пример – представление чисел, которые сами суть абстракции свойств некоторых объектов. Если единственное (или основное) действие, которое нужно выполнять, – сложение, то хорошим представлением числа n может быть n черточек. Правило сложения при таком представлении – очевидное и очень простое. Римская нотация основана на этом принципе простоты, и правила сложения просты для маленьких чисел. С другой стороны, представление арабскими цифрами требует неочевидных правил сложения (для маленьких чисел), и их нужно запоминать. Однако ситуация меняется на противоположную, если нужно складывать большие числа или выполнять умножение и деление. Разбиение этих операций на более простые шаги гораздо проще в случае арабской нотации благодаря ее систематической позиционной структуре.

Хорошо известно, что компьютеры используют внутреннее представление, основанное на двоичных цифрах (битах). Это представление непригодно для использования людьми, так как здесь обычно приходится иметь дело с большим

числом цифр, но весьма удобно для электронных схем, так как два значения 0 и 1 можно легко и надежно представить посредством наличия или отсутствия электрических токов, зарядов или магнитных полей.

Из этого примера также видно, что вопрос представления часто требует рассматривать несколько уровней детализации. Например, в задаче представления положения объекта первое решение может касаться выбора пары чисел в, скажем, декартовых или полярных координатах. Второе решение может привести к представлению с плавающей точкой, где каждое вещественное число x состоит из пары целых, обозначающих дробную часть f и показатель e по некоторому основанию (например, $x = f \times 2^e$). Третье решение, основанное на знании, что данные будут храниться в компьютере, может привести к двоичному позиционному представлению целых чисел. Наконец, последнее решение может состоять в том, чтобы представлять двоичные цифры электрическими зарядами в полупроводниковом устройстве памяти. Очевидно, первое решение в этой цепочке зависит главным образом от решаемой задачи, а дальнейшие все больше зависят от используемого инструмента и применяемых в нем технологий. Вряд ли можно требовать, чтобы программист решал, какое представление чисел использовать или даже какими должны быть характеристики устройства хранения данных. Такие решения низкого уровня можно оставить проектировщикам вычислительного оборудования, у которых заведомо больше информации о существующих технологиях, чтобы сделать разумный выбор, приемлемый для всех (или почти всех) приложений, где играют роль числа.

В таком контексте выявляется важность языков программирования. Язык программирования представляет абстрактный компьютер, допускающий интерпретацию в терминах данного языка, что может подразумевать определенный уровень абстракции по сравнению с объектами, используемыми в реальном вычислительном устройстве. Тогда программист, использующий такой язык высокого уровня, будет освобожден от заботы о представлении чисел (и лишен возможности что-то сделать в этом отношении), если числа являются элементарными объектами в данном языке.

Использование языка, предоставляющего удобный набор базовых абстракций, общих для большинства задач обработки данных, влияет главным образом на надежность получающихся программ. Легче спроектировать программу, опираясь в рассуждениях на знакомые понятия чисел, множеств, последовательностей и циклов, чем иметь дело с битами, единицами хранения и переходами управления. Конечно, реальный компьютер представляет любые данные – числа, множества или последовательности – как огромную массу битов. Но программист может забыть об этом, если ему не нужно беспокоиться о деталях представления выбранных абстракций и если он может считать, что выбор представления, сделанный компьютером (или компилятором), разумен для решаемых задач.

Чем ближе абстракции к конкретному компьютеру, тем легче сделать выбор представления инженеру или автору компилятора и тем выше вероятность, что единственный выбор будет подходить для всех (или почти всех) мыслимых приложений. Это обстоятельство устанавливает определенные пределы на «высоту»

используемых абстракций по сравнению с уровнем реального «железа». Например, неразумно включать в язык общего назначения геометрические фигуры, так как из-за внутренне присущей им сложности их подходящее представление будет сильно зависеть от действий, выполняемых с ними. Однако природа и частота таких действий неизвестна проектировщику языка программирования общего назначения и соответствующего компилятора, и любой выбор проектировщика может оказаться плохим для некоторого класса приложений.

Эти соображения определили выбор нотации для описания алгоритмов и соответствующих данных в настоящей книге. Разумеется, нам хотелось бы использовать знакомые понятия математики, такие как числа, множества, последовательности и т. д., а не машинно зависимые сущности вроде строк битов. Но нам также хотелось бы использовать нотацию, для которой существуют эффективные компиляторы. Неразумно использовать язык, в сильной степени машинно зависимый, но также недостаточно и описывать программы в абстрактной нотации, в которой проблемы представления остаются нерешенными. Язык программирования Паскаль был спроектирован в попытке найти компромисс между этими двумя крайностями, а его наследники Модула-2 и Оберон учитывают опыт, накопленный за десятилетия [1.3]. Оберон сохраняет базовые понятия Паскаля с некоторыми усовершенствованиями и добавлениями; он используется на протяжении этой книги [1.5]. Оберон был успешно реализован для ряда компьютеров, при этом было продемонстрировано, что его нотация достаточно близка к реальному «железу», чтобы выбранные средства и их представления можно было объяснить с полной ясностью. Язык также близок к другим языкам, так что уроки, усвоенные здесь, могут быть с равным успехом применены и при их использовании.

1.2. Понятие типа данных

В математике переменные обычно классифицируются по некоторым важным характеристикам. Проводится четкое различие между вещественными, комплексными и логическими переменными, или между переменными, представляющими отдельные значения, множества значений, множества множеств, или между функциями, функционалами, множествами функций и т. д. Такая классификация не менее, если не более, важна в обработке данных. Мы будем придерживаться того принципа, что каждая константа, выражение или функция имеет определенный *тип*. В сущности, тип характеризует множество значений, к которому принадлежит константа, или которые может принимать переменная или выражение, или которые могут порождаться функцией.

В математических текстах тип переменной обычно можно определить просто по шрифту, без учета контекста; но это невозможно в компьютерных программах. На вычислительной установке обычно доступен только один шрифт (латинские буквы). Поэтому часто следуют правилу явно вводить соответствующий тип в *объявлении* константы, переменной или функции, причем такое объявление должно предшествовать использованию этой константы, переменной или функции. Это правило тем более разумно, что компилятор должен выбрать пред-

ставление объекта в памяти компьютера. Очевидно, что объем памяти, отведенной под переменную, должен быть выбран в соответствии с диапазоном значений, которые может принимать переменная. Если эта информация доступна компилятору, то можно избежать так называемого динамического размещения. Очень часто этот пункт оказывается ключевым для эффективной реализации алгоритма.

Сущность понятия типа, как оно используется в данном тексте и реализуется в языке программирования Оберон, выражается в следующих утверждениях [1.2]:

1. Тип данных определяет множество значений, которому принадлежит значение константы, или в котором принимает значения переменная или выражение, или которому принадлежат значения, порождаемые операцией или функцией.
2. Тип значения, обозначенного константой, переменной или выражением, может быть выведен из их объявлений и вида выражения без выполнения вычислений.
3. Каждая операция или функция требует аргументов определенных типов и дает результат некоторого, тоже определенного типа. Если операция допускает аргументы нескольких типов (например, + используется для сложения как целых, так и вещественных чисел), то тип результата может быть определен на основе особых правил языка программирования.

Компилятор может использовать такую информацию о типах для проверки законности различных конструкций. Например, ошибочное присваивание булевского (логического) значения арифметической переменной может быть обнаружено без выполнения программы. Подобная избыточность текста программы весьма полезна при ее разработке и может рассматриваться как главное преимущество хороших языков высокого уровня по сравнению с машинным кодом (или кодом символического ассемблера).

Очевидно, в конечном итоге данные будут представлены огромным количеством двоичных цифр независимо от того, была ли написана исходная программа на языке высокого уровня, использующего понятие типа, или на ассемблере, где типов нет. Для компьютера память представляется однородной массой битов без явной структуры. Но именно абстрактная структура позволяет человеку-программисту видеть смысл в монотонном пейзаже компьютерной памяти.

Теория, о которой идет речь в данной книге, и язык программирования Оберон дают некоторые способы определения типов данных. В большинстве случаев новый тип данных строится из других типов, уже определенных (назовем их *составляющими*). Значения такого типа – это обычно агрегаты значений-компонент, принадлежащих ранее определенным составляющим типам, и такие значения называются *составными*, или *структурированными*. Если используется только один составляющий тип, то есть все компоненты принадлежат одному типу, то этот тип называют *базовым*. Число различных значений типа T называют его *мощностью*. Мощность позволяет определить объем памяти для представления переменной x, имеющей тип T, что обозначается как x: T.

Поскольку составляющие типы, в свою очередь, могут быть составными, то могут выстраиваться целые иерархии структур. Впрочем, очевидно, что наимень-

шие компоненты структуры должны быть неделимыми. Поэтому нужны некие стандартные, заранее определенные типы. Обычно здесь речь идет о числах и логических значениях. Если значения типа могут быть упорядочены, то такой тип называют *упорядоченным*. В Обероне все бесструктурные типы упорядочены.

Имея такие средства, из примитивных типов можно строить агрегаты, составные типы любого уровня вложенности. На практике недостаточно иметь только один общий способ построения структуры из составляющих типов. Должным образом учитывая практические проблемы представления и использования, язык программирования общего назначения должен предлагать несколько методов структурирования данных. В математическом смысле они эквивалентны, но отличаются операциями для доступа к компонентам структур. Базовые методы структурирования, которые будут представлены ниже, суть *массив*, *запись* и *последовательность*. Более сложные структуры обычно не определяются как статические типы, а порождаются динамически во время выполнения программы, когда могут меняться их размер *и* состав. Таким структурам посвящена глава 4; сюда относятся списки, кольца, деревья и вообще конечные графы.

Переменные и типы данных вводятся в программу для использования в вычислениях. Для вычислений нужно иметь набор операций. Поэтому для каждого стандартного типа данных язык программирования предлагает некий набор примитивных, стандартных операций, а для каждого метода структурирования – специальные операции для доступа к компонентам вместе с соответствующей нотацией. Нередко считают, что суть искусства программирования – в комбинировании операций. Однако мы увидим, что хорошее структурирование данных – задача не менее фундаментальная и важная.

Важнейшие основные операции – *сравнение* и *присваивание*, то есть проверка равенства (или порядка в случае упорядоченных типов) и команда, так сказать, обеспечивающая равенство. Принципиальное различие между ними подчеркивается обозначениями, которые используются на протяжении всей книги:

Проверка равенства: `x = y` (выражение, дающее значение `TRUE` или `FALSE`)
Присваивание переменной `x`: `x := y` (оператор, делающий `x` равным `y`)

Эти основополагающие действия определены для большинства типов данных, но следует заметить, что их выполнение может требовать серьезных вычислений, если данные велики по объему и имеют сложную структуру.

Для стандартных примитивных типов данных мы постулируем не только возможность присваивания и сравнения, но еще и набор операций для создания (вычисления) новых значений. А именно: для числовых типов мы вводим стандартные арифметические операции, а для логических значений – элементарные операции логики высказываний.

1.3. Стандартные примитивные типы

Стандартные примитивные типы суть те, которые доступны в большинстве компьютеров «на уровне железа». Сюда включаются целые числа, логические значения, а также некоторый набор литер для печати. На многих компьютерах еще име-

ются дробные числа вместе со стандартными арифметическими операциями. Мы обозначаем эти типы следующими идентификаторами:

INTEGER, REAL, BOOLEAN, CHAR, SET

1.3.1. Тип INTEGER

Тип INTEGER представляет подмножество целых чисел, диапазон значений которых может меняться от одной вычислительной системы к другой. Если компьютер использует n битов для представления целых чисел в дополнительном коде, то допустимые значения x должны удовлетворять условию $-2^{n-1} \le x < 2^{n-1}$. Предполагается, что все операции с данными этого типа являются точными и соответствуют обычным законам арифметики и что вычисление будет прервано, если результат лежит за пределами указанного диапазона. Такое событие называют *переполнением*. Стандартные операции – четыре основные арифметические операции: сложение (+), вычитание (–), умножение (*) и деление (/, DIV).

Косая черта будет обозначать деление, имеющее результатом значение типа REAL, а операция DIV будет обозначать целочисленное деление, имеющее результатом значение типа INTEGER. Если мы определим частное q = m DIV n и остаток r = m MOD n, то должны выполняться следующие соотношения (предполагается, что n > 0):

$q*n + r = m$ и $0 \le r < n$

Примеры

```
 31 DIV 10 =  3
 31 MOD 10 = 1
-31 DIV 10 = -4    -31 MOD 10 = 9
```

Известно, что деление на 10^n может быть осуществлено простым сдвигом десятичных знаков на n позиций вправо с отбрасыванием избыточных цифр справа. Аналогичный метод применим, если числа представлены в двоичной, а не десятичной нотации. Если используется дополнительный код (как это имеет место практически во всех современных компьютерах), то сдвиги обеспечивают деление в соответствии с данным выше определением операции DIV. Поэтому в компиляторе нетрудно реализовать операцию вида m DIV 2^n (или m MOD 2^n) с помощью быстрой операции сдвига (или с помощью маски).

1.3.2. Тип REAL

Тип REAL представляет подмножество вещественных чисел. Если для арифметических операций с операндами типа INTEGER предполагается, что они дают точные результаты, то арифметические операции со значениями типа REAL могут быть неточными в пределах ошибок округления, вызванных тем, что вычисления производятся с конечным числом значащих цифр. Это главная причина для того, чтобы явно различать типы INTEGER и REAL, как это делается в большинстве языков программирования.

Стандартные операции – четыре основные арифметические операции: сложение (+), вычитание (–), умножение (*) и деление (/). Сущность типизации данных – в том, что разные типы становятся несовместимыми по присваиванию. Исключение делается для присваивания целочисленных значений вещественным переменным, так как семантика в этом случае однозначна. Ведь целые числа составляют подмножество вещественных. Однако присваивание в обратном направлении запрещено: присваивание вещественного значения целой переменной требует отбрасывания дробной части или округления. Стандартная функция преобразования ENTIER(x) дает целую часть величины x. Тогда округление величины x выполняется с помощью ENTIER(x + 0.5).

Многие языки программирования не содержат операций возведения в степень. Следующий алгоритм обеспечивает быстрое вычисление величины $y = x^n$, где n – неотрицательное целое:

```
y := 1.0; i := n;                                          (* ADruS13 *)
WHILE i > 0 DO   (* x0^n  =  x^i * y *)
    IF ODD(i) THEN y := y*x END;
    x := x*x;  i := i DIV 2
END
```

1.3.3. Тип BOOLEAN

Два значения стандартного типа BOOLEAN обозначаются идентификаторами TRUE и FALSE. Булевские операции – это логические конъюнкция, дизъюнкция и отрицание, которые определены в табл. 1.1. Логическая конъюнкция обозначается символом &, логическая дизъюнкция – символом OR, а отрицание – символом ~. Заметим, что операции сравнения всегда вычисляют результат типа BOOLEAN. Поэтому результат сравнения можно присвоить переменной или использовать как операнд логической операции в булевском выражении. Например, для булевских переменных p и q и целых переменных x = 5, y = 8, z = 10 два присваивания

```
p := x = y
q := (x ≤ y) & (y < z)
```

дают p = FALSE и q = TRUE.

Таблица 1.1. Булевские операции

p	q	p & q	p OR q	~p
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

В большинстве языков программирования булевские операции & (AND) и OR обладают дополнительным свойством, отличающим их от других бинарных операций. Например, сумма x+y не определена, если не определен любой из операндов x

или y, однако конъюнкция p&q определена, даже если не определено значение q, при условии что p равно FALSE. Такое соглашение оказывается важным и полезным. Поэтому точное определение операций & и OR дается следующими равенствами:

p & q = если p, то q, иначе FALSE
p OR q = если p, то TRUE, иначе q

1.3.4. Тип CHAR

Стандартный тип CHAR представляет литеры, которые можно напечатать. К сожалению, нет общепринятого стандартного множества литер для всех вычислительных систем. Поэтому прилагательное «стандартный» в этом случае может привести к путанице; его следует понимать в смысле «стандартный для вычислительной установки, на которой должна выполняться программа».

Чаще всего используется множество литер, определенное Международной организацией по стандартизации (ISO), и, в частности, его американский вариант ASCII (American Standard Code for Information Interchange). Поэтому множество ASCII приведено в приложении A. Оно содержит 95 графических (имеющих изображение) литер, а также 33 управляющие (не имеющих изображения) литеры, используемые в основном при передаче данных и для управления печатающими устройствами.

Чтобы можно было создавать алгоритмы для работы с литерами (то есть со значениями типа CHAR), которые не зависели бы от вычислительной системы, нужно иметь возможность сделать некоторые минимальные предположения о свойствах множества литер, а именно:

1. Тип CHAR содержит 26 заглавных латинских букв, 26 строчных букв, 10 десятичных цифр, а также некоторые другие графические символы, например знаки препинания.
2. Подмножества букв и цифр упорядочены и между собой не пересекаются:
   ("A" ≤ x) & (x ≤ "Z") подразумевает, что x – заглавная буква;
   ("a" ≤ x) & (x ≤ "z") подразумевает, что x – строчная буква;
   ("0" ≤ x) & (x ≤ "9") подразумевает, что x – десятичная цифра.
3. Тип CHAR содержит непечатаемые символы пробела и конца строки, которые можно использовать как разделители.

Для написания машинно независимых программ особенно важны две стандартные функции преобразования между типами CHAR и INTEGER. Назовем их ORD(ch) (дает порядковый номер литеры ch в используемом множестве литер)

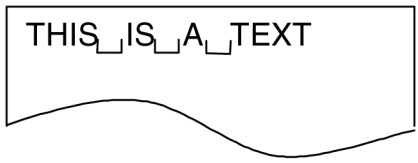


Рис. 1.1. Представление текста

и CHR(i) (дает литеру с порядковым номером i). Таким образом, CHR является обратной функцией для ORD и наоборот, то есть

ORD(CHR(i)) = i (если значение CHR(i) определено)
CHR(ORD(c)) = c

Кроме того, постулируем наличие стандартной функции CAP(ch). Ее значение – заглавная буква, соответствующая ch, если ch – буква.

если ch – строчная буква, то CAP(ch) – соответствующая заглавная буква
если ch – заглавная буква, то CAP(ch) = ch

1.3.5. Тип SET

Тип SET представляет множества, элементами которых являются целые числа из диапазона от 0 до некоторого небольшого числа, обычно 31 или 63. Например, если определены переменные

```
VAR r, s, t: SET
```

то возможны присваивания

```
r := {5}; s := {x, y .. z}; t := {}
```

Здесь переменной r присваивается множество, состоящее из единственного элемента 5; переменной t присваивается пустое множество, а переменной s – множество, состоящее из элементов x, y, y+1, ... , z–1, z.

Следующие элементарные операции определены для операндов типа SET:

*	пересечение множеств
+	объединение множеств
–	разность множеств
/	симметрическая разность множеств
IN	принадлежность множеству

Операции пересечения и объединения множеств часто называют умножением и сложением множеств соответственно; приоритеты этих операций определяются так, что приоритет пересечения выше, чем у объединения и разности, а приоритеты последних выше, чем у операции принадлежности, которая вычисляет логическое значение. Ниже даются примеры выражений с множествами и их версии с полностью расставленными скобками:

```
r * s + t   =  (r*s) + t
r – s * t   =  r – (s*t)
r – s + t   =  (r–s) + t
r + s / t   =  r + (s/t)
x IN s + t =  x IN (s+t)
```

1.4. Массивы

Вероятно, массив – наиболее широко используемая структура данных; в некоторых языках это вообще единственная структура. Массив состоит из компонент, имеющих одинаковый тип, называемый *базовым*; поэтому о массиве говорят как

об *однородной* структуре. Массив допускает *произвольный доступ*, так как можно произвольно выбрать любую компоненту, и доступ к ним осуществляется одинаково быстро. Чтобы обозначить отдельную компоненту, к имени всего массива нужно присоединить *индекс*, то есть номер компоненты. Индекс должен быть целым числом в диапазоне от 0 до n–1, где n – число элементов, или *длина* массива.

```
TYPE T = ARRAY n OF T0
```

Примеры

```
TYPE Row    = ARRAY 4 OF REAL
TYPE Card   = ARRAY 80 OF CHAR
TYPE Name   = ARRAY 32 OF CHAR
```

Конкретное значение переменной

```
VAR x: Row
```

у которой все компоненты удовлетворяют уравнению $x_i = 2^{-i}$, можно представить как на рис. 1.2.

x_0	1.0
x_1	0.5
x_2	0.25
x_3	0.125

Рис. 1.2. Массив типа Row, в котором $x_i = 2^{-i}$

Отдельная компонента массива выбирается с помощью *индекса*. Если имеется переменная-массив x, то соответствующий *селектор* (то есть конструкцию для выбора отдельной компоненты) будем изображать посредством имени массива, за которым следует индекс соответствующей компоненты i, и будем писать x_i или x[i]. Первое из этих обозначений принято в математике, поэтому компоненту массива еще называют *индексированной переменной*.

Обычный способ работы с массивами, особенно с большими, состоит в том, чтобы выборочно изменять отдельные компоненты, вместо того чтобы строить новое составное значение целиком. Для этого переменную-массив рассматривают как массив переменных-компонент и разрешают присваивания отдельным компонентам, например x[i] := 0.125. Хотя при выборочном присваивании меняется только одна компонента, с концептуальной точки зрения мы должны считать, что меняется все составное значение массива.

Тот факт, что индексы массива (фактически имена компонент массива) суть целые числа, имеет весьма важное следствие: индексы могут вычисляться. Вместо индекса-константы можно поставить общее индексное выражение; тогда подразумевается, что выражение вычисляется, а его значение используется для выбора компоненты. Такая общность не только дает весьма важное и мощное средство программирования, но и приводит к одной из самых частых ошибок в программах: вычисленное значение индекса может оказаться за пределами диапазона индексов данного массива. Будем предполагать, что «приличные» вычислительные системы выдают предупреждение, если имеет место ошибочная попытка доступа к несуществующей компоненте массива.

Мощность стуктурированного типа, то есть количество значений, принадлежащих этому типу, равна произведению мощностей его компонент. Поскольку все компоненты массивового типа T имеют одинаковый базовый тип T0, то получаем

$card(T) = card(T0)^n$

Элементы массивов сами могут быть составными. Переменная-массив, у которой все компоненты – тоже массивы, называется *матрицей*. Например,

```
M: ARRAY 10 OF Row
```

является массивом, состоящим из десяти компонент (строк), причем каждая компонента состоит из четырех компонент типа REAL. Такой массив называется матрицей 10×4 с вещественными компонентами. Селекторы могут приписываться один за другим, так что M_{ij} и M[i][j] обозначают j-ю компоненту строки M_i, которая, в свою очередь, является i-й компонентой матрицы M. Обычно используют сокращенную запись M[i,j], и аналогично для объявления

```
M: ARRAY 10 OF ARRAY 4 OF REAL
```

можно использовать сокращенную запись

```
M: ARRAY 10, 4 OF REAL
```

Если нужно выполнить некоторое действие со всеми компонентами массива или с группой идущих подряд компонент, то удобно подчеркнуть этот факт, используя оператор цикла FOR, как показано в следующих примерах, где вычисляется сумма и находится максимальный элемент массива a:

```
VAR a: ARRAY N OF INTEGER;                                  (* ADruS14 *)

sum := 0;
FOR i := 0 TO N-1 DO sum := a[i] + sum END

k := 0; max := a[0];
FOR i := 1 TO N-1 DO
    IF max < a[i] THEN k := i; max := a[k] END
END
```

В следующем примере предполагается, что дробь f представляется в десятичном виде с k-1 цифрами, то есть массивом d, таким, что

$f = \mathbf{S}i: 0 \le i < k: d_i * 10^{-i}$ или

$f = d_0 + 10*d_1 + 100*d_2 + ... + 10^{k-1}*d_{k-1}$

Предположим теперь, что мы хотим поделить f на 2. Это делается повторением уже знакомой операции деления для всех k-1 цифр d_i, начиная с i=1. Каждая цифра делится на 2 с учетом остатка деления в предыдущей позиции, а возможный остаток от деления в данной позиции, в свою очередь, запоминается для следующего шага:

```
r := 10*r +d[i];  d[i] := r DIV 2;  r := r MOD 2
```

Этот алгоритм используется для вычисления таблицы отрицательных степеней числа 2. Повторные деления пополам для вычисления величин $2^{-1}, 2^{-2}, ..., 2^{-N}$ снова удобно выразить оператором цикла FOR, так что в итоге получается пара вложенных циклов FOR.

```
PROCEDURE Power (VAR W: Texts.Writer; N: INTEGER);          (* ADruS14 *)
    (*вычислить десятичное представление отрицательных степеней числа 2*)
    VAR i, k, r: INTEGER;
        d: ARRAY N OF INTEGER;
BEGIN
    FOR k := 0 TO N-1 DO
        Texts.Write(W, "."); r := 0;
        FOR i := 0 TO k-1 DO
            r := 10*r + d[i]; d[i] := r DIV 2; r := r MOD 2;
            Texts.Write(W, CHR(d[i] + ORD("0")))
        END;
        d[k] := 5; Texts.Write(W, "5"); Texts.WriteLn(W)
    END
END Power
```

В результате для N = 10 печатается следующий текст:

```
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

1.5. Записи

Самый общий способ строить составные типы заключается в том, чтобы объединять в некий агрегат элементы произвольных типов, которые сами могут быть составными типами. Примеры из математики: комплексные числа, составленные из пары вещественных, а также координаты точки, составленные из двух или более чисел в соответствии с размерностью пространства. Пример из обработки данных: описание людей посредством нескольких свойств, таких как имя и фамилия, дата рождения.

С точки зрения математики, такой *составной тип* (compound type) является прямым (декартовым) произведением *составляющих типов* (constituent types): множество значений составного типа состоит из всевозможных комбинаций значений, каждое из которых взято из множества значений соответствующего составляющего типа. Поэтому число таких комбинаций, называемых также *n-ками* (n-tuples), равно произведению чисел элементов каждого составляющего типа; другими словами, мощность составного типа равна произведению мощностей составляющих типов.

В обработке данных сложные типы, такие как описания людей или объектов, обычно хранятся в файлах или базах данных и содержат нужные характеристики человека или объекта. Поэтому для описания составных данных такой природы

стал широко употребляться термин *запись*, и мы тоже будем его использовать вместо термина «прямое произведение». В общем случае *записевый тип* (record type) T с компонентами типов T_1, T_2, ... , T_n определяется следующим образом:

TYPE T = RECORD s_1: T_1; s_2: T_2; ... s_n: T_n END
card(T) = card(T_1) * card(T_2) * ... * card(T_n)

Примеры

```
TYPE Complex =  RECORD  re, im: REAL END
TYPE Date    =  RECORD  day, month, year:  INTEGER END
TYPE Person  =  RECORD  name, firstname: Name;
                        birthdate: Date;
                        male: BOOLEAN
                END
```

Конкретные значения записей, например, для переменных

```
z: Complex
d: Date
p: Person
```

можно изобразить так, как показано на рис. 1.3.

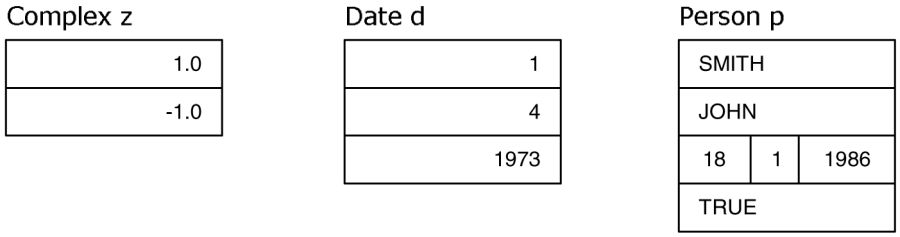


Рис. 1.3. Записи типов Complex, Date и Person

Идентификаторы s_1, s_2, ..., s_n, вводимые в определении записевого типа, суть имена, данные отдельным компонентам переменных этого типа. Компоненты записи называются *полями* (field), а их имена – *идентификаторами полей* (field identifiers). Их используют в селекторах, применяемых к переменным записевых типов. Если дана переменная x: T, ее i-е поле обозначается как $x.s_i$. Можно выполнить частичное изменение x, если использовать такой селектор в левой части оператора присваивания:

$x.s_i$:= e

где e – значение (выражение) типа T_i. Например, если имеются записевые переменные z, d, и p, объявленные выше, то следующие выражения суть селекторы их компонент:

```
z.im      (типа REAL)
d.month   (типа INTEGER)
```

```
p.name          (типа Name)
p.birthdate     (типа Date)
p.birthdate.day (типа INTEGER)
p.mail          (типа BOOLEAN)
```

Пример типа Person показывает, что компонента записевого типа сама может быть составной. Поэтому селекторы могут быть применены один за другим. Естественно, что разные способы структурирования тоже могут комбинироваться. Например, i-я компонента массива a, который сам является компонентой записевой переменной r, обозначается как r.a[i], а компонента с именем s из i-го элемента массива a, состоящего из записей, обозначается как a[i].s.

Прямое произведение по определению состоит из всевозможных комбинаций элементов своих составляющих типов. Однако нужно заметить, что в реальных приложениях не все они могут иметь смысл. Например, определенный выше тип Date допускает значения 31 апреля и 29 февраля 1985, которых в календаре нет. Поэтому приведенное определение этого типа отражает реальную ситуацию не вполне верно, но достаточно близко, а программист должен обеспечить, чтобы при выполнении программы бессмысленные значения никогда не возникали.

Следующий фрагмент программы показывает использование записевых переменных. Его назначение – подсчет числа лиц женского пола, родившихся после 2000 г., среди представленных в переменной-массиве:

```
VAR count: INTEGER;
    family: ARRAY N OF Person;

count := 0;
FOR i := 0 TO N-1 DO
   IF ~family[i].male & (family[i].birthdate.year > 2000) THEN INC(count) END
END
```

И запись, и массив обеспечивают *произвольный доступ* к своим компонентам. Запись является более общей в том смысле, что ее составляющие типы могут быть разными. Массив, в свою очередь, допускает бо́льшую гибкость в том отношении, что селекторы компонент могут вычисляться (задаваться выражениями), тогда как селекторы компонент записи суть идентификаторы полей, объявленные в определении типа записи.

1.6. Представление массивов, записей и множеств

Смысл использования абстракций в программировании – в том, чтобы обеспечить возможность спроектировать, понять и верифицировать программу, исходя только из свойств абстракций, то есть не зная о том, как абстракции реализованы и представлены на конкретной машине. Тем не менее профессиональному программисту нужно понимать широко применяемые способы представления фундаментальных структур данных. Это может помочь принять разумные реше-

ния о построении программы и данных в свете не только абстрактных свойств структур, но и их реализации на конкретной машине с учетом ее специфических особенностей и ограничений.

Проблема представления данных заключается в том, как отобразить абстрактную структуру на память компьютера. Память компьютера – это, грубо говоря, массив отдельных ячеек, которые называются *байтами*. Они интерпретируются как группы из 8 бит каждая. Индексы байтов называются *адресами*.

```
VAR store: ARRAY StoreSize OF BYTE
```

Примитивные типы представляются небольшим числом байтов, обычно 1, 2, 4 или 8. Компьютеры проектируются так, чтобы пересылать небольшие группы смежных байтов одновременно, или, как говорят, параллельно. Единицу одновременной пересылки называют *словом*.

1.6.1. Представление массивов

Представление массива – это отображение (абстрактного) массива, компоненты которого имеют тип T, на память компьютера, которая сама есть массив компонент типа BYTE. Массив должен быть отображен на память таким образом, чтобы вычисление адресов его компонент было как можно более простым (и поэтому эффективным). Адрес i для j-й компоненты массива вычисляется с помощью линейной функции

$$i = i_0 + j*s,$$

где i_0 – адрес первой компоненты, а s – количество слов, которые занимает одна компонента. Принимая, что слово является наименьшей единицей пересылки содержимого памяти, весьма желательно, чтобы s было целым числом, в простейшем случае s = 1. Если s не является целым (а это довольно обычная ситуация), то s обычно округляется вверх до ближайшего большего целого S. Тогда каждая компонента массива занимает S слов, тогда как S–s слов остаются неиспользованными (см. рис. 1.4 и 1.5). Округление необходимого числа слов вверх до ближайшего целого называется *выравниванием* (padding). Эффективность использования памяти u определяется как отношение минимального объема памяти, нужного для

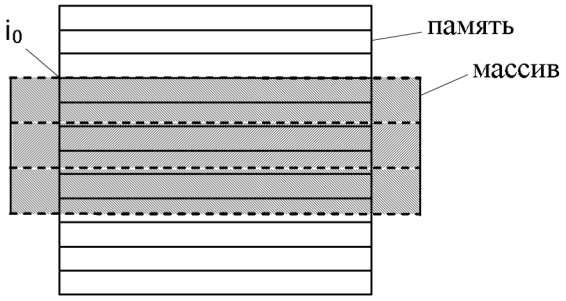


Рис. 1.4. Отображение массива на память компьютера

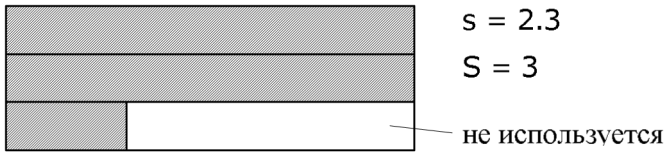


Рис. 1.5. Представление записи с выравниванием

представления структуры, к реально использованному объему:

u = s / (s, округленное вверх до ближайшего целого).

Поскольку проектировщик компилятора должен стремиться к тому, чтобы эффективность использования памяти была как можно ближе к 1, но при этом доступ к частям слова громоздок и относительно неэффективен, приходится идти на компромисс. При этом учитываются следующие соображения:

1. Выравнивание уменьшает эффективность использования памяти.
2. Отказ от выравнивания может повлечь необходимость выполнять неэффективный доступ к частям слова.
3. Доступ к частям слова может вызвать разбухание скомпилированного кода и тем самым уменьшить выигрыш, полученный из-за отказа от выравнивания.

На самом деле соображения 2 и 3 обычно настолько существенны, что компиляторы автоматически используют выравнивание. Заметим, что эффективность использования памяти всегда $u > 0.5$, если $s > 0.5$. Если же $s \le 0.5$, то эффективность использования памяти можно сильно увеличить, размещая более одной компоненты массива в каждом слове. Это называется *упаковкой* (packing). Если в одном слове упакованы n компонент, то эффективность использования памяти равна (см. рис. 1.6)

u = n*s / (n*s, округленное вверх до ближайшего целого).

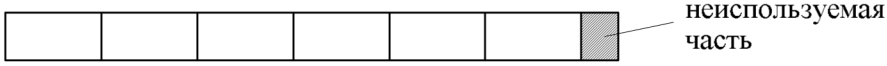


Рис. 1.6. Упаковка шести компонент в одном слове

Доступ к i-й компоненте упакованного массива подразумевает вычисление адреса слова j, в котором находится нужная компонента, а также позиции k-ой компоненты внутри слова:

```
j = i DIV n        k = i MOD n
```

В большинстве языков программирования программист не имеет возможности управлять представлением структур данных. Однако должна быть возможность указывать желательность упаковки хотя бы в тех случаях, когда в одно слово помещается больше одной компоненты, то есть когда может быть достигнут

выигрыш в экономии памяти в два раза и более. Можно предложить указывать желательность упаковки с помощью символа PACKED, который ставится перед символом ARRAY (или RECORD) в соответствующем объявлении.

1.6.2. Представление записей

Записи отображаются на память компьютера простым расположением подряд их компонент. Адрес компоненты (поля) r_i относительно адреса начала записи r называется *смещением* (offset) поля k_i. Оно вычисляется следующим образом:

$$k_i = s_1 + s_2 + \ldots + s_{i-1} \quad k_0 = 0$$

где s_j – размер (в словах) j-й компоненты. Здесь видно, что равенство типов всех компонент массива имеет приятным следствием, что $k_i = i \times s$. К несчастью, общность записевой структуры не позволяет использовать простую линейную функцию для вычисления смещения компоненты; именно поэтому требуют, чтобы компоненты записи выбирались с помощью фиксированных идентификаторов. У этого ограничения есть то преимущество, что смещения полей определяются во время компиляции. В результате доступ к полям записи более эффективен.

Упаковка может привести к выигрышу, если несколько компонент записи могут поместиться в одном слове памяти (см. рис. 1.7). Поскольку смещения могут быть вычислены компилятором, смещение поля, упакованного внутри слова, тоже может быть определено компилятором. Это означает, что на многих вычислительных установках упаковка записей в гораздо меньшей степени снижает эффективность доступа к полям, чем упаковка массивов.

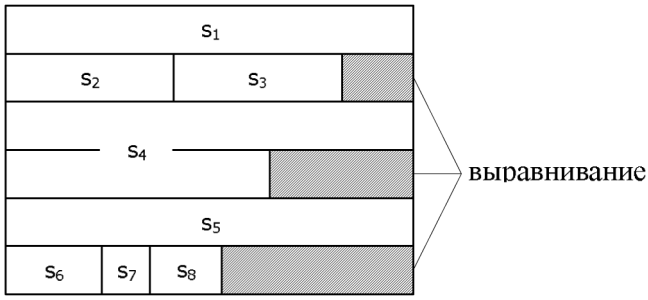


Рис. 1.7. Представление упакованной записи

1.6.3. Представление множеств

Множество s удобно представлять в памяти компьютера его характеристической функцией C(s). Это массив логических значений, чья i-я компонента означает, что i присутствует в s. Например, множество небольших чисел s = {2, 3, 5, 7, 11, 13} представляется последовательностью или цепочкой битов:

C(s) = (... 0010100010101100)

Представление множеств характеристическими функциями имеет то преимущество, что операции вычисления, объединения, пересечения и разности двух множеств могут быть реализованы как элементарные логические операции. Следующие тождества, выполняющиеся для всех элементов i множеств x и y, связывают логические операции с операциями на множествах:

```
i IN (x+y)  = (i IN x) OR (i IN y)
i IN (x*y)  = (i IN x) & (i IN y)
i IN (x-y)  = (i IN x) & ~(i IN y)
```

Такие логические операции имеются во всех цифровых компьютерах, и, более того, они выполняются одновременно для всех элементов (битов) слова. Поэтому для эффективной реализации базовых операций множества их следует представлять небольшим фиксированным количеством слов, для которых можно выполнить не только базовые логические операции, но и операции сдвига. Тогда проверка на принадлежность реализуется единственным сдвигом с последующей проверкой знака. В результате проверка вида $x \text{ IN } \{c_1, c_2, \ldots, c_n\}$ может быть реализована гораздо эффективнее, чем эквивалентное булевское выражение

$(x = c_1) \text{ OR } (x = c_2) \text{ OR } \ldots \text{ OR } (x = c_n)$

Как следствие множества должны использоваться только с небольшими числами в качестве элементов, из которых наибольшее равно длине слова компьютера (минус 1).

1.7. Файлы или последовательности

Еще один элементарный способ структурирования – *последовательность*. Обычно это однородная структура, подобная массиву. Это означает, что все ее элементы имеют одинаковый тип – *базовый тип* последовательности. Будем обозначать последовательность s из n элементов следующим образом:

$s = \langle s_0, s_1, s_2, \ldots, s_{n-1} \rangle$

Число n называется *длиной* последовательности.

Эта структура выглядит в точности как массив. Но существенная разница в том, что у массива число элементов зафиксировано в его определении, а у последовательности – нет. То есть оно может меняться во время выполнения программы. Хотя каждая последовательность в любой момент времени имеет конкретную конечную длину, мы должны считать мощность последовательностного типа бесконечной, так как нет никаких ограничений на потенциальную длину последовательностей.

Прямое следствие переменной длины последовательностей – невозможность отвести фиксированный объем памяти под переменные-последовательности. Поэтому память нужно выделять во время выполнения программы, в частности когда последовательность растет. Соответственно, когда последовательность сокра-

щается, освобождающуюся память можно утилизовать. В любом случае нужна некая динамическая схема выделения памяти. Подобными свойствами обладают все структуры переменного размера, и это обстоятельство столь важно, что мы характеризуем их как «сложные» (advanced) структуры, в отличие от фундаментальных, обсуждавшихся до сих пор.

Тогда почему мы обсуждаем последовательности в главе, посвященной фундаментальным структурам? Главная причина – в том, что стратегия управления памятью для последовательностей оказывается достаточно простой (в отличие от других «сложных» структур), если потребовать определенной дисциплины использования последовательностей. И тогда можно обеспечить довольно эффективный механизм управления памятью. Вторая причина – в том, что едва ли не в каждой задаче, решаемой с помощью компьютеров, используются последовательности. Например, последовательности обычно используют в тех случаях, когда данные пересылаются с одного устройства хранения на другое, например с диска или ленты в оперативную память или обратно.

Упомянутая дисциплина состоит в том, чтобы ограничиться только последовательным доступом. Это подразумевает, что доступ к элементам последовательности осуществляется строго в порядке их следования, а порождается последовательность присоединением новых элементов к ее концу. Немедленное следствие – невозможность *прямого доступа* к элементам, за исключением того элемента, который доступен для просмотра в данный момент. Именно такая дисциплина доступа составляет главное отличие последовательностей от массивов. Как мы увидим в главе 2, дисциплина доступа оказывает глубокое влияние на программы.

Преимущество последовательного доступа, который все-таки является серьезным ограничением, – в относительной простоте необходимого здесь способа управления памятью. Но еще важнее возможность использовать эффективные методы *буферизации* (buffering) при пересылке данных между оперативной памятью и внешними устройствами. Последовательный доступ позволяет «прокачивать» потоки данных с помощью «каналов» (pipes) между разными устройствами хранения. Буферизация подразумевает накопление данных из потока в *буфере* и последующую пересылку целиком содержимого всего буфера, как только он заполнится. Это приводит к весьма существенному повышению эффективности использования внешней памяти. Если ограничиться только последовательным доступом, то механизм буферизации довольно прост для любых последовательностей и любых внешних устройств. Поэтому его можно заранее предусмотреть в вычислительной системе и предоставить для общего пользования, освободив программиста от необходимости включать его в свою программу. Обычно здесь речь идет о *файловой системе*, где для постоянного хранения данных используют устройства последовательного доступа большого объема, в которых данные сохраняются даже после выключения компьютера. Единицу хранения данных в таких устройствах обычно называют (*последовательным*) *файлом*. Мы будем использовать термин *файл* (file) как синоним для термина *последовательность* (sequence).

Существуют устройства хранения данных, в которых последовательный доступ является единственно возможным. Очевидно, сюда относятся все виды лент. Но даже на магнитных дисках каждая дорожка представляет собой хранилище,

допускающее только последовательный доступ. Строго последовательный доступ характерен для любых устройств с механически движущимися частями, а также для некоторых других.

Отсюда следует, что полезно проводить различие между *стуктурой данных*, то есть последовательностью, с одной стороны, и *механизмом доступа к ее элементам* – с другой. Первая объявляется как структура данных, а механизм доступа обычно реализуется посредством некоторой записи, с которой ассоциированы некоторые операторы, – или, используя современную терминологию, посредством объекта доступа или «бегунка» (rider object). Различать объявление данных и механизм доступа полезно еще и потому, что для одной последовательности могут одновременно существовать несколько точек доступа, в которых осуществляется последовательный доступ к разным частям последовательности.

Суммируем главное из предшествующего обсуждения следующим образом:

1. Массивы и записи – структуры, допускающие произвольный доступ к своим элементам. Их используют, размещая в оперативной памяти.
2. Последовательности используются для работы с данными на внешних устройствах хранения, допускающих последовательный доступ, таких как диски или ленты.
3. Мы проводим различие между последовательностью как структурой данных и механизмом доступа, подразумевающим определенную позицию в ней.

1.7.1. Элементарные операции с файлами

Дисциплину последовательного доступа можно обеспечить, предоставляя набор специальных операций, помимо которых доступ к файлам невозможен. Поэтому хотя в общих рассуждениях можно использовать обозначение s_i для i-го элемента последовательности s, в программе это невозможно.

Последовательности, то есть файлы, – это обычно большие, динамические структуры данных, сохраняемые на внешних запоминающих устройствах. Такое устройство сохраняет данные, даже если программа заканчивается или отключается компьютер. Поэтому введение файловой переменной – сложная операция, подразумевающая подсоединение данных на внешнем устройстве к файловой переменной в программе. Поэтому объявим тип File в отдельном модуле, определение которого описывает тип вместе с соответствующими операциями. Назовем этот модуль Files и условимся, что последовательностная или файловая переменная должна быть явно инициализирована («открыта») посредством вызова соответствующей операции или функции:

```
VAR f: File
f := Open(name)
```

где name идентифицирует файл на внешнем устройстве хранения данных. В некоторых системах различается открытие существующего и нового файлов:

```
f := Old(name)          f := New(name)
```

Нужно еще потребовать, чтобы связь между внешней памятью и файловой переменной разрывалась, например посредством вызова Close(f).

Очевидно, набор операций должен содержать операцию для порождения (записи) последовательности и еще одну – для ее просмотра (чтения). Потребуем, чтобы эти операции применялись не напрямую к файлу, а к некоторому объекту-*бегунку*, который сам подсоединен к файлу (последовательности) и который реализует некоторый механизм доступа. Дисциплина последовательного доступа обеспечивается ограничением набора операций доступа (процедур).

Последовательность создается присоединением элементов к ее концу после подсоединения бегунка к файлу. Если есть объявление

```
VAR r: Rider
```

то бегунок r подсоединяется к файлу f оператором

```
Set(r, f, pos)
```

где pos = 0 обозначает начало файла (последовательности). Вот типичная схема порождения последовательности:

```
WHILE есть еще DO вычислить следующий элемент x; Write(r, x) END
```

Чтобы прочитать последовательность, сначала к ней подсоединяется бегунок, как показано выше, и затем производится чтение одного элемента за другим. Вот типичная схема чтения последовательности:

```
Read(r, x);
WHILE ~r.eof DO обработать элемент x; Read(r, x) END
```

Очевидно, с каждым бегунком всегда связана некоторая позиция. Обозначим ее как r.pos. Далее примем, что бегунок содержит предикат (флажок) r.eof, указывающий, был ли достигнут конец последовательности в предыдущей операции чтения (eof – сокращение от англ. «end of file», то есть «конец файла» – *прим. перев.*). Теперь мы можем постулировать и неформально описать следующий набор примитивных операций:

1a.	New(f, name)	определяет f как пустую последовательность.
1b.	Old(f, name)	определяет f как последовательность, хранящуюся на внешнем носителе с указанным именем.
2.	Set(r, f, pos)	связывает бегунок r с последовательностью f и устанавливает его в позицию pos.
3.	Write(r, x)	записывает элемент со значением x в последовательность, с которой связан бегунок r, и продвигает его вперед.
4.	Read(r, x)	присваивает переменной x значение элемента, на который указывает бегунок r, и продвигает его вперед.
5.	Close(f)	регистрирует записанный файл f на внешнем носителе (с немедленной записью содержимого буферов на диск).

Замечание. Запись элемента в последовательность – это часто достаточно сложная операция. С другой стороны, файлы обычно создаются присоединением новых элементов в конце.

Замечание переводчика. В примерах программ в книге используются еще две операции:

6. WriteInt(r, n) записывает целое число n в последовательность, с которой связан бегунок r, и продвигает его вперед.
7. ReadInt(r, n) присваивает переменной n целое число, на которое указывает бегунок r, и продвигает его вперед.

Чтобы дать более точное представление об операциях последовательного доступа, ниже приводится пример реализации. В нем показано, как можно выразить операции, если последовательности представлены массивами. Этот пример намеренно использует только понятия, введенные и обсужденные ранее, и в нем нет ни буферизации, ни последовательных устройств хранения данных, которые, как указывалось выше, делают понятие последовательности по-настоящему нужным и полезным. Тем не менее этот пример показывает все существенные свойства простейших операций последовательного доступа независимо от того, как последовательности представляются в памяти.

Операции реализуются как обычные процедуры. Такой набор объявлений типов, переменных и заголовков процедур (сигнатур) называется *определением* (definition). Будем предполагать, что элементами последовательностей являются литеры, то есть что речь идет о текстовых файлах, чьи элементы имеют тип CHAR. Объявления типов File и Rider являются хорошими примерами применения записей, так как в дополнение к полю, обозначающему массив, представляющий данные, нужны и другие поля для обозначения текущей длины файла и позиции, то есть состояния бегунка:

```
DEFINITION Files;                                    (* ADruS171_Files *)

   TYPE File;  (*последовательность литер*)
      Rider = RECORD eof: BOOLEAN END;

   PROCEDURE New(VAR name: ARRAY OF CHAR): File;
   PROCEDURE Old(VAR name: ARRAY OF CHAR): File;
   PROCEDURE Close(VAR f: File);

   PROCEDURE Set(VAR r: Rider; VAR f: File; pos: INTEGER);
   PROCEDURE Write(VAR r: Rider; ch: CHAR);
   PROCEDURE Read(VAR r: Rider; VAR ch: CHAR);

   PROCEDURE WriteInt(VAR r: Rider; n: INTEGER);
   PROCEDURE ReadInt(VAR r: Rider; VAR n: INTEGER);
END Files.
```

Определение представляет собой некоторую абстракцию. Здесь нам даны два типа данных, File и Rider, вместе с соответствующими операциями, но без каких-либо дальнейших деталей, раскрывающих их реальное представление в памяти. Что касается операций, объявленных как процедуры, то мы видим только их заголовки. Детали реализации не показаны здесь намеренно, и называется это *упрятыванием информации* (information hiding). О бегунках мы узнаем только то, что

у них есть свойство с именем eof. Этот флажок получает значение TRUE, когда операция чтения достигает конца файла. Позиция бегунка скрыта, и поэтому его инвариант не может быть разрушен прямым обращением к его полям. Инвариант выражает тот факт, что позиция бегунка всегда находится в пределах, соответствующих данной последовательности. Истинность инварианта первоначально устанавливается процедурой Set, а в дальнейшем требуется и поддерживается процедурами Read и Write (а также ReadInt и WriteInt – *прим. перев.*).

Операторы, реализующие описанные процедуры, а также все детали реализации типов данных содержатся в так называемом *модуле* (module). Представить данные и реализовать процедуры можно многими способами. В качестве простого примера приведем следующий модуль (с фиксированной максимальной длиной файла):

```
MODULE Files;                                          (* ADruS171_Files *)
   CONST MaxLength = 4096;
   TYPE
      File = POINTER TO RECORD
         len: INTEGER;
         a: ARRAY MaxLength OF CHAR
      END;
      Rider = RECORD  (* 0 <= pos <= f.len <= Max Length *)
         f: File; pos: INTEGER; eof: BOOLEAN
      END;
   PROCEDURE New (name: ARRAY OF CHAR): File;
      VAR f: File;
   BEGIN
      NEW(f); f.len := 0; f.eof := FALSE;
      (*операции с файловой директорией опущены*)
      RETURN f
   END New;
   PROCEDURE Old (name: ARRAY OF CHAR): File;
      VAR f: File;
   BEGIN
      NEW(f); f.eof := FALSE; (*поиск в директории опущен*)
      RETURN f
   END Old;
   PROCEDURE Close (VAR f: File);
   BEGIN
   END Close;
   PROCEDURE Set (VAR r: Rider; f: File; pos: INTEGER);
   BEGIN (*предполагается f # NIL*)
      r.f := f; r.eof := FALSE;
      IF pos >= 0 THEN
         IF pos <= f.len THEN r.pos := pos ELSE r.pos := f.len END
      ELSE
```

```
            r.pos := 0
        END
    END Set;

    PROCEDURE Write (VAR r: Rider; ch: CHAR);
    BEGIN
        IF (r.pos <= r.s.len) & (r.pos < MaxLength) THEN
            r.f.a[r.pos] := ch; INC(r.pos);
            IF r.pos > r.f.len THEN INC(r.f.len) END
        ELSE
            r.eof := TRUE
        END
    END Write;

    PROCEDURE Read (VAR r: Rider; VAR ch: CHAR);
    BEGIN
        IF r.pos < r.f.len THEN
            ch := r.f.a[r.pos]; INC(r.pos)
        ELSE
            r.eof := TRUE
        END
    END Read;

    PROCEDURE WriteInt (VAR r: Rider; n: INTEGER);
    BEGIN (*реализация зависит от платформы*)
    END WriteInt;

        PROCEDURE ReadInt (VAR r: Rider; VAR n: INTEGER);
    BEGIN (*реализация зависит от платформы*)
    END ReadInt;

END Files.
```

В этом примере максимальная длина, которую могут иметь файлы, задается произвольно выбранной константой. Если какая-то программа попытается создать более длинную последовательность, то это будет не ошибкой программы, а недостатком данной реализации. С другой стороны, попытка чтения за текущим концом файла будет означать ошибку программы. Здесь флаг r.eof также используется операцией записи, чтобы сообщить, что выполнить ее не удалось. Поэтому условие ~r.eof является предусловием как для Read, так и для Write (предусловие – это логическое выражение, которое должно быть истинным для корректного выполнения некоторой операции – *прим. перев.*).

1.7.2. Буферизация последовательностей

Когда данные пересылаются со внешнего устройства хранения или на него, отдельные биты передаются потоком. Обычно устройство налагает строгие временные ограничения на пересылку данных. Например, если данные записываются на ленту, лента движется с фиксированной скоростью, и нужно, чтобы данные передавались ей тоже с фиксированной скоростью. Когда источник данных исчерпан,

движение ленты прекращается, и ее скорость падает быстро, но не мгновенно. Поэтому на ленте остается промежуток между уже записанными данными и данными, которые поступят позже. Чтобы добиться высокой плотности данных, нужно, чтобы число промежутков было мало, и для этого данные передают относительно большими блоками, чтобы не прерывать движения ленты. Похожие требования имеют место при работе с магнитными дисками, где данные размещаются на дорожках с фиксированным числом блоков фиксированного размера. На самом деле диск следует рассматривать как массив блоков, причем каждый блок читается или записывается целиком и обычно содержит 2^k байтов с k = 8, 9, ... 12.

Однако в наших программах не соблюдается никаких временных ограничений. Чтобы обеспечить такую возможность, передаваемые данные буферизуются. Они накапливаются в переменной-буфере (в оперативной памяти) и пересылаются, когда накапливается достаточно данных, чтобы собрать блок нужного размера. Клиент буфера имеет к нему доступ только посредством двух процедур deposit и fetch:

```
DEFINITION Buffer;
    PROCEDURE deposit (x: CHAR);
    PROCEDURE fetch (VAR x: CHAR);
END Buffer.
```

Буферизация обладает тем дополнительным преимуществом, что она позволяет процессу, который порождает/получает данные, выполняться одновременно с устройством, которое пишет/читает данные в/из буфера. На самом деле удобно рассматривать само устройство как процесс, который просто копирует потоки данных. Назначение буфера – в какой-то степени ослабить связь между двумя процессами, которые будем называть *производителем* (producer) и *потребителем* (consumer). Например, если потребитель в какой-то момент замедляет работу, он может нагнать производителя позднее. Без такой развязки часто нельзя обеспечить полноценное использование внешних устройств, но она работает, только если скорость работы производителя и потребителя примерно равны в среднем, хотя иногда и флуктуируют. Степень развязки растет с ростом размера буфера.

Обратимся теперь к вопросу представления буфера и для простоты предположим пока, что элементы данных записываются в него (deposited) и считываются из него (fetched) индивидуально, а не поблочно. В сущности, буфер представляет собой очередь, организованную по принципу «первым пришел – первым ушел» (first-in-first-out, или fifo). Если он объявлен как массив, то две индексные переменные (скажем, in и out) отмечают те позиции, куда должны писаться и откуда должны считываться данные. В идеале такой массив должен быть бесконечным. Однако вполне до-

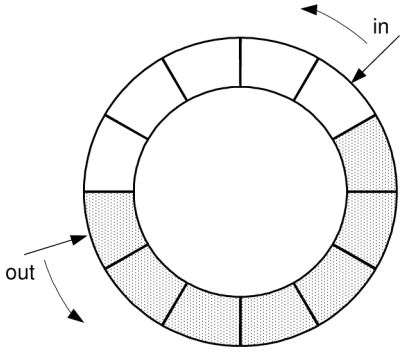


Рис. 1.8. Кольцевой буфер с индексами in и out

статочно иметь конечный массив, учитывая, что прочитанные элементы больше не нужны. Занимаемое ими место может быть использовано повторно. Это приводит к идее *кольцевого буфера*.

Операции записи и считывания элемента реализуются в следующем модуле, который экспортирует эти операции как процедуры, но скрывает буфер и его индексные переменные – и тем самым механизм буферизации – от процесса-потребителя. В таком механизме еще нужна переменная n для подсчета количества элементов в буфере в данный момент. Если N обозначает размер буфера, то очевидным инвариантом является условие $0 \le n \le N$. Поэтому операция считывания (процедура fetch) должна охраняться условием $n > 0$ (буфер не пуст), а операция записи (процедура deposit) – условием $n < N$ (буфер не полон). Невыполнение первого условия должно считаться ошибкой программирования, а нарушение второго – недостатком предложенной реализации (буфер слишком мал).

```
MODULE Buffer;   (*реализует кольцевые буфера*)
    CONST N = 1024;  (*размер буфера*)
    VAR n, in, out: INTEGER;
        buf: ARRAY N OF CHAR;

    PROCEDURE deposit (x: CHAR);
    BEGIN
        IF n = N THEN HALT END;
        INC(n); buf[in] := x; in := (in + 1) MOD N
    END deposit;

    PROCEDURE fetch (VAR x: CHAR);
    BEGIN
        IF n = 0 THEN HALT END;
        DEC(n); x := buf[out]; out := (out + 1) MOD N
    END fetch;

BEGIN n := 0; in := 0; out := 0
END Buffer.
```

Столь простая реализация буфера приемлема, только если процедуры deposit и fetch вызываются единственным агентом (действующим то как производитель, то как потребитель). Но если они вызываются независимыми процессами, работающими одновременно, то такая схема оказывается слишком примитивной. Ведь тогда попытку записи в полный буфер или попытку чтения из пустого буфера следует рассматривать как вполне законные. Просто выполнение таких действий должно быть отложено до того момента, когда снова будут выполнены соответствующие *охраны* (guarding conditions). В сущности, такие задержки и представляют собой необходимый механизм синхронизации между параллельными (concurrent) процессами. Можно представить эти задержки следующими операторами:

```
REPEAT UNTIL n < N
REPEAT UNTIL n > 0
```

которые нужно подставить вместо соответствующих двух условных операторов, содержащих оператор HALT.

1.7.3. Буферизация обмена между параллельными процессами

Однако представленное решение нельзя рекомендовать, даже если известно, что два процесса исполняются двумя независимыми агентами. Причина в том, что два процесса должны обращаться к одной и той же переменной n и, следовательно, к одной области оперативной памяти. Ожидающий процесс, постоянно проверяя значение n, мешает своему партнеру, так как в любой момент времени к памяти может обратиться только один процесс. Такого рода ожиданий следует избегать, и поэтому мы постулируем наличие средства, которое, в сущности, скрывает в себе механизм синхронизации. Будем называть это средство *сигналом* (signal) и примем, что оно предоставляется в служебном модуле Signals вместе с набором примитивных операций для сигналов.

Каждый сигнал s связан с охраной (условием) P_s. Если процесс нужно приостановить, пока не будет обеспечена истинность P_s (другим процессом), то он должен, прежде чем продолжить свою работу, дождаться сигнала s. Это выражается оператором Wait(s). С другой стороны, если процесс обеспечивает истинность P_s, то после этого он сигнализирует об этом оператором Send(s). Если для каждого оператора Send(s) обеспечивается истинность предусловия P_s, то P_s можно рассматривать как постусловие для Wait(s).

```
DEFINITION Signals;
    TYPE Signal;
    PROCEDURE Wait (VAR s: Signal);
    PROCEDURE Send (VAR s: Signal);
    PROCEDURE Init (VAR s: Signal);
END Signals.
```

Теперь мы можем реализовать буфер в виде следующего модуля, который должен правильно работать, когда он используется независимыми параллельными процессами:

```
MODULE Buffer;
    IMPORT Signals;
    CONST N = 1024;  (*размер буфера*)
    VAR n, in, out: INTEGER;
        nonfull: Signals.Signal;   (*n < N*)
        nonempty: Signals.Signal;  (*n > 0*)
        buf: ARRAY N OF CHAR;

    PROCEDURE deposit (x: CHAR);
    BEGIN
        IF n = N THEN Signals.Wait(nonfull) END;
        INC(n); buf[in] := x; in := (in + 1) MOD N;
        IF n = 1 THEN Signals.Send(nonempty) END
    END deposit;
```

```
    PROCEDURE fetch (VAR x: CHAR);
    BEGIN
        IF n = 0 THEN Signals.Wait(nonempty) END;
        DEC(n); x := buf[out]; out := (out + 1) MOD N;
        IF n = N-1 THEN Signals.Send(nonfull) END
    END fetch;

BEGIN n := 0; in := 0; out := 0; Signals.Init(nonfull); Signals.Init(nonempty)
END Buffer.
```

Однако нужно сделать еще одну оговорку. Данная схема разрушается, если по случайному совпадению как производитель, так и потребитель (или два производителя либо два потребителя) одновременно обращаются к переменной n, чтобы изменить ее значение. Непредсказуемым образом получится либо значение n+1, либо n-1, но не n. Так что нужно защищать процессы от опасных взаимных помех. Вообще говоря, все операции, которые изменяют значения общих (shared) переменных, представляют собой потенциальные ловушки.

Достаточным (но не всегда необходимым) условием является требование, чтобы все общие переменные объявлялись локальными в таком модуле, для процедур которого гарантируется, что они *взаимно исключают* исполнение друг друга. Такой модуль называют *монитором* (monitor) [1.7]. Условие взаимного исключения (mutual exclusion) гарантирует, что в любой момент времени только один процесс сможет активно выполнять какую-либо процедуру монитора. Если другой процесс попытается вызвать некую процедуру того же монитора, его выполнение будет автоматически задержано до того момента, когда первый процесс завершит выполнение своей процедуры.

Замечание. Слова «активно выполнять» означают, что процесс выполняет любой оператор, кроме оператора ожидания.

Наконец, вернемся к задаче, в которой производитель или потребитель (или оба) требует, чтобы данные к ним поступали блоками определенного размера. Показанный ниже модуль является вариантом предыдущего, причем предполагается, что размер блоков данных равен N_p элементов для производителя и N_c элементов для потребителя. В этом случае обычно выбирают размер буфера N так, чтобы он делился на N_p и N_c. Чтобы подчеркнуть симметрию между операциями записи и считывания данных, вместо единственного счетчика n теперь используются два счетчика, ne и nf. Они показывают соответственно число пустых и заполненных ячеек буфера. Когда потребитель находится в состоянии ожидания, nf показывает число элементов, нужных для продолжения работы потребителя; а когда производитель находится в состоянии ожидания, то ne показывает число элементов, необходимых для продолжения работы производителя. (Поэтому условие ne + nf = N выполняется не всегда.)

```
MODULE Buffer;
    IMPORT Signals;
    CONST Np = 16;  (*размер блока производителя*)
        Nc = 128;  (*размер блока потребителя*)
```

```
    N = 1024;  (*размер буфера, делится на Np и Nc*)
  VAR ne, nf: INTEGER;
    in, out: INTEGER;
    nonfull: Signals.Signal;   (*ne >= 0*)
    nonempty: Signals.Signal;  (*nf >= 0*)
    buf: ARRAY N OF CHAR;

  PROCEDURE deposit (VAR x: ARRAY OF CHAR);
  BEGIN
    ne := ne - Np;
    IF ne < 0 THEN Signals.Wait(nonfull) END;
    FOR i := 0 TO Np-1 DO buf[in] := x[i]; INC(in) END;
    IF in = N THEN in := 0 END;
    nf := nf + Np;
    IF nf >= 0 THEN Signals.Send(nonempty) END
  END deposit;

  PROCEDURE fetch (VAR x: ARRAY OF CHAR);
  BEGIN
    nf := nf - Nc;
    IF nf < 0 THEN Signals.Wait(nonempty) END;
    FOR i := 0 TO Nc-1 DO x[i] := buf[out]; INC(out) END;
    IF out = N THEN out := 0 END;
    ne := ne + Nc;
    IF ne >= 0 THEN Signals.Send(nonfull) END
  END fetch;

BEGIN
  ne := N; nf := 0; in := 0; out := 0;
  Signals.Init(nonfull); Signals.Init(nonempty)
END Buffer.
```

1.7.4. Ввод и вывод текста

Под стандартным вводом и выводом мы понимаем передачу данных в ту или иную сторону между вычислительной системой и внешними агентами, например человеком-оператором. Достаточно типично, что ввод производится с клавиатуры, а вывод – на экран дисплея. Для таких ситуаций характерно, что информация представляется в форме, понятной человеку, и обычно состоит из последовательности литер. То есть речь идет о тексте. Отсюда еще одно усложнение, характерное для реальных операций ввода и вывода. Кроме передачи данных, в них выполняется еще и преобразование представления. Например, числа, обычно рассматриваемые как неделимые сущности и представленные в двоичном виде, должны быть преобразованы в удобную для чтения десятичную форму. Структуры должны представляться так, чтобы их элементы располагались определенным образом, то есть форматироваться.

Независимо от того, что это за преобразование, задача заметно упрощается, если снова привлечь понятие последовательности. Решающим является наблюде-

ние, что если набор данных можно рассматривать как последовательность литер, то преобразование последовательности может быть реализовано как последовательность (одинаковых) преобразований элементов:

$T(<s_0, s_1, \ldots, s_{n-1}>) = <T(s_0), T(s_1), \ldots, T(s_{n-1})>$

Исследуем вкратце действия, необходимые для преобразования представлений натуральных чисел для ввода и вывода. Математическим основанием послужит тот факт, что число x, представленное последовательностью десятичных цифр $d = <d_{n-1}, \ldots, d1, d0>$, имеет значение

$$x = \mathbf{S}i: i = 0 .. n-1: d_i * 10^i$$
$$x = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \ldots + d_1 \times 10 + d_0$$
$$x = (\ldots (d_{n-1} \times 10 + d_{n-2}) \times 10 + \ldots + d_1) \times 10 + d_0$$

Пусть теперь нужно прочесть и преобразовать последовательность d, а получившееся числовое значение присвоить переменной x. Следующий простой алгоритм останавливается при считывании первой литеры, не являющейся цифрой (арифметическое переполнение не рассматривается):

```
x := 0; Read(ch);                                   (* ADruS174.ЛитерыВЧисло *)
WHILE ("0" <= ch) & (ch <= "9") DO
    x := 10*x + (ORD(ch) - ORD("0")); Read(ch)
END
```

В случае вывода преобразование усложняется тем, что разложение значения x в набор десятичных цифр дает их в обратном порядке. Младшая значащая цифра порождается первой при вычислении x MOD 10. Поэтому требуется промежуточный буфер в виде очереди типа «первым пришел – последним вышел» (то есть стека). Будем представлять ее массивом d с индексом i и получим следующую программу:

```
i := 0;                                             (* ADruS174.ЧислоВЛитеры *)
REPEAT d[i] := x MOD 10; x := x DIV 10; INC(i)
UNTIL x = 0;
REPEAT DEC(i); Write(CHR(d[i] + ORD("0")))
UNTIL i = 0
```

Замечание. Систематическая замена константы 10 в этих алгоритмах на положительное целое B даст процедуры преобразования для представления по основанию B. Часто используется случай B = 16 (шестнадцатеричное представление), тогда соответствующие умножения и деления можно реализовать простыми сдвигами двоичных цифр.

Очевидно, было бы неразумным детально описывать в каждой программе такие часто встречающиеся операции. Поэтому постулируем наличие вспомогательного модуля, который обеспечивает чаще всего встречающиеся, стандартные операции ввода и вывода для чисел и цепочек литер. Этот модуль используется в большинстве программ в этой книге, и мы назовем его Texts. В нем определен

тип Text, а также типы объектов-бегунков для чтения (Reader) и записи (Writer) в переменные типа Text, а также процедуры для чтения и записи литеры, целого числа и цепочки литер.

Прежде чем дать определение модуля Texts, подчеркнем существенную асимметрию между вводом и выводом текстов. Хотя текст порождается последовательностью вызовов процедур вывода целых и вещественных чисел, цепочек литер и т. д., ввод текста посредством вызова процедур чтения представляется сомнительной практикой. Дело здесь в том, что хотелось бы читать следующий элемент, не зная его типа, и определять его тип *после* чтения. Это приводит к понятию *сканера* (scanner), который после каждой попытки чтения позволяет проверить тип и значение прочитанного элемента. Сканер играет роль бегунка для файлов. Однако тогда нужно наложить ограничения на синтаксическую структуру считываемых текстов. Мы определим сканер для текстов, состоящих из последовательности целых и вещественных чисел, цепочек литер, имен, а также специальных литер. Синтаксис этих элементов задается следующими правилами так называемой расширенной нотации Бэкуса–Наура (EBNF, Extended Backus Naur Form; чтобы точнее отразить вклад авторов нотации в ее создание, аббревиатуру еще раскрывают как Extended Backus Normal Form, то есть «расширенная нормальная нотация Бэкуса» – *прим. перев.*):

```
item       =  integer | RealNumber | identifier | string | SpecialChar.
integer    =  ["-"] digit {digit}.
RealNumber =  ["-"] digit {digit} "." digit {digit} [("E" | "D")["+" |
              "-" digit {digit}].
identifier =  letter {letter | digit}.
string     =  '"' {any character except quote} '"'.
SpecialChar =  "!" | "?" | "@" | "#" | "$" | "%" | "^" | "&" | "+" | "-" |
              "*" | "/" | "\" | "|" | "(" | ")" | "[" | "]" | "{" | "}" |
              "<" | ">" | "." | "," | ":" | ";" | "~".
```

Элементы разделяются пробелами и/или символами конца строк.

```
DEFINITION Texts;                                    (* ADruS174_Texts *)

   CONST Int = 1; Real = 2; Name = 3; Char = 4;
   TYPE Text, Writer;
      Reader = RECORD eot: BOOLEAN END;
      Scanner = RECORD class: INTEGER;
         i: INTEGER;
         x: REAL;
         s: ARRAY 32 OF CHAR;
         ch: CHAR;
         nextCh: CHAR
      END;

   PROCEDURE OpenReader (VAR r: Reader; t: Text; pos: INTEGER);
   PROCEDURE OpenWriter (VAR w: Writer; t: Text; pos: INTEGER);
   PROCEDURE OpenScanner (VAR s: Scanner; t: Text; pos: INTEGER);
   PROCEDURE Read (VAR r: Reader; VAR ch: CHAR);
```

```
    PROCEDURE ReadInt (VAR r: Reader; VAR n: INTEGER);
    PROCEDURE Scan (VAR s: Scanner);

    PROCEDURE Write (VAR w: Writer; ch: CHAR);
    PROCEDURE WriteLn (VAR w: Writer);  (*завершить строку*)
    PROCEDURE WriteString (VAR w: Writer; s: ARRAY OF CHAR);
    PROCEDURE WriteInt (VAR w: Writer; x, n: INTEGER); (*вывести целое x с по
    крайней мере n литерами. Если n больше, чем нужно, перед числом
    добавляются пробелы*)
    PROCEDURE WriteReal (VAR w: Writer; x: REAL);
    PROCEDURE Close (VAR w: Writer);
END Texts.
```

(Выше добавлена отсутствующая в английском оригинале процедура ReadInt, используемая в примерах программ – *прим. перев.*)

Мы требуем, чтобы после вызова процедуры Scan(S) для полей записи S выполнялось следующее:

S.class = Int	означает, что прочитано целое число, его значение содержится в S.i;
S.class = Real	означает, что прочитано вещественное число, его значение содержится в S.x;
S.class = Name	означает, что прочитана цепочка литер, она содержится в S.s;
S.class = Char	означает, что прочитана специальная литера, она содержится в S.ch;
S.nextCh	содержит литеру, непосредственно следующую за прочитанным элементом, которая может быть пробелом.

1.8. Поиск

Задача поиска – одна из наиболее часто встречающихся в программировании. Она также дает прекрасные возможности показать применение рассмотренных структур данных. Есть несколько основных вариаций на тему поиска, и здесь придумано множество алгоритмов. Основное предположение, которое мы делаем в дальнейшем изложении, состоит в том, что набор данных, в котором ищется заданное значение, фиксирован. Будем предполагать, что этот набор N элементов представлен массивом, скажем

```
a: ARRAY N OF Item
```

Обычно элементы являются записями, одно из полей которых играет роль ключа. Тогда задача состоит в нахождении элемента, у которого поле ключа равно заданному значению x, которое еще называют *аргументом поиска*. Найденный индекс i, удовлетворяющий условию a[i].key = x, позволит обратиться к другим полям найденного элемента. Поскольку нам здесь интересна только задача поиска, но не данные, ради которых производится поиск, мы будем предполагать, что тип Item состоит только из ключа, то есть *сам* является ключом.

1.8.1. Линейный поиск

Если нет никакой дополнительной информации о данных, то очевидное решение – последовательно проходить по массиву, шаг за шагом увеличивая величину той части массива, где искомое значение заведомо отсутствует. Это решение известно как *линейный поиск*. Поиск прекращается при одном из двух условий:

1. Элемент найден, то есть $a_i = x$.
2. Просмотрен весь массив, но элемент не найден.

Приходим к следующему алгоритму:

```
i := 0;                                              (* ADruS18_Поиск *)
WHILE (i < N) & (a[i] # x) DO INC(i) END
```

Отметим, что порядок операндов в булевском выражении важен.

До и после каждого шага цикла выполняется следующее условие:

$(0 \le i < N) \;\&\; (\mathbf{A}k: 0 \le k < i : a_k \ne x)$

Такое условие называется инвариантом. В данном случае инвариант означает, что для всех значений k, меньших, чем i, среди a_k искомого значения x нет. Заметим, что до и после каждого шага цикла значения i – разные. Сохранение инварианта при изменении i имеет место в данном случае благодаря истинности охраны цикла (условия между ключевыми словами WHILE и DO).

Из выполнения инварианта и из того факта, что поиск прекращается, только если станет ложной охрана цикла, получается условие, выполняющееся после окончания данного фрагмента программы (так называемое постусловие для данного фрагмента):

$((i = N) \text{ OR } (a_i = x)) \;\&\; (\mathbf{A}k: 0 \le k < i : a_k \ne x)$

Это условие не только является искомым результатом, но еще и подразумевает, что если найден элемент, равный x, то это первый такой элемент. При этом i = N означает, что искомого значения не найдено.

Окончание цикла гарантировано, так как величина i на каждом шаге увеличивается и поэтому обязательно достигнет границы N после конечного числа шагов; на самом деле после N шагов, если искомого значения в массиве нет.

На каждом шаге нужно вычислять булевское выражение и увеличивать индекс. Можно ли упростить эту задачу и тем самым ускорить поиск? Единственная возможность – упростить булевское выражение, состоящее, как видим, из двух членов. Поэтому построить более простое решение удастся, только если найти условие с единственным членом, из которого будут следовать оба. Это возможно лишь при гарантии, что искомое значение всегда будет найдено, а это можно обеспечить, помещая дополнительный элемент со значением x в конец массива. Будем называть такой вспомогательный элемент *барьером* (sentinel), так как он препятствует выходу поиска за границу массива. Теперь массив a объявляется так:

```
a: ARRAY N+1 OF INTEGER,
```

и алгоритм линейного поиска с барьером выражается следующим образом:

```
a[N] := x; i := 0;                                    (* ADruS18_Поиск *)
WHILE a[i] # x DO INC(i) END
```

В итоге получается условие, выведенное из того же инварианта, что и ранее:

$(a_i = x)$ & $(Ak: 0 \le k < i : a_k \ne x)$

Очевидно, из i = N следует, что искомое значение не встретилось (не считая значения-барьера).

1.8.2. Поиск делением пополам

Понятно, что ускорить поиск невозможно, если нет дополнительной информации о данных, в которых он выполняется. Хорошо известно, что поиск может быть гораздо более эффективным, если данные упорядочены. Достаточно представить себе телефонный справочник, в котором номера даны не по алфавиту: такой справочник совершенно бесполезен. Поэтому обсудим теперь алгоритм, который использует информацию о том, что массив a упорядочен, то есть что выполняется условие

$\mathbf{A}k: 1 \le k < N : a_{k-1} \le a_k$

Ключевая идея – в том, чтобы выбрать наугад элемент, скажем a_m, и сравнить его с искомым значением x. Если он равен x, то поиск прекращается; если он меньше x, то можно заключить, что все элементы с индексами, равными или меньшими m, можно игнорировать в дальнейшем поиске; а если он больше x, то можно игнорировать все значения индекса, большие или равные m. Это приводит к следующему алгоритму, который носит название *поиск делением пополам* (binary search); в нем используются две индексные переменные L и R, отмечающие в массиве a левый и правый концы отрезка, в котором искомое значение все еще может быть найдено:

```
L := 0; R := N-1;                                     (* ADruS18_Поиск *)
m := любое значение между L и R;
WHILE (L <= R) & (a[m] # x) DO
    IF a[m] < x THEN
        L := m+1
    ELSE
        R := m-1
    END;
    m := любое значение между L и R
END
```

Подчеркнем фундаментальное структурное подобие этого алгоритма и алгоритма линейного поиска в предыдущем разделе: роль i теперь играет тройка L, m, R. Чтобы не потерять это подобие и тем самым надежней гарантировать корректность цикла, мы воздержались от соблазна мелкой оптимизации программы с целью устранения дублирования инструкции присваивания переменной m.

Следующее условие является инвариантом цикла, то есть выполняется до и после каждого шага:

$(\mathbf{A}k: 0 \le k < L : a_k < x) \;\&\; (\mathbf{A}k: R < k < N : a_k > x)$

откуда выводим для цикла такое постусловие:

$((L > R)\ \mathrm{OR}\ (a_m = x)) \;\&\; (\mathbf{A}k: 0 \le k < L : a_k < x) \;\&\; (\mathbf{A}k: R < k < N : a_k > x)$

Из него следует, что

$((L > R) \;\&\; (\mathbf{A}k: 0 \le k < N : a_k \ne x))\ \mathrm{OR}\ (a_m = x)$

Выбор m, очевидно, произволен в том смысле, что правильность алгоритма от него не зависит. Но от него зависит эффективность алгоритма. Ясно, что на каждом шаге нужно исключать из дальнейшего поиска как можно больше элементов независимо от результата сравнения. Оптимальное решение – выбрать средний элемент, так как здесь в любом случае из поиска исключается половина отрезка. Получается, что максимальное число шагов равно $\log_2 N$, округленное до ближайшего целого. Поэтому данный алгоритм представляет собой радикальное улучшение по сравнению с линейным поиском, где среднее число сравнений равно N/2.

Как уже упоминалось, можно заняться устранением дублирования инструкции присваивания m. Однако такая оптимизация на уровне кода является преждевременной в том смысле, что сначала лучше попытаться оптимизировать алгоритм на уровне логики задачи. Здесь это действительно возможно: ввиду сходства алгоритма с алгоритмом линейного поиска естественно искать решение с более простым условием окончания, то есть без второго операнда конъюнкции в охране цикла. Для этого нужно отказаться от наивного желания закончить поиск сразу после нахождения искомого значения. На первый взгляд это неразумно, но при ближайшем рассмотрении оказывается, что выигрыш в эффективности на каждом шаге больше, чем потери из-за небольшого числа дополнительных сравнений. Напомним, что число шагов не превышает log N.

Более быстрое решение основано на следующем инварианте:

$(\mathbf{A}k: 0 \le k < L : a_k < x) \;\&\; (\mathbf{A}k: R \le k < N : a_k \ge x)$

а поиск продолжается, пока два отрезка не будут покрывать весь массив:

```
L := 0; R := N;                                          (* ADruS18_Поиск *)
WHILE L < R DO
    m := (L+R) DIV 2;
    IF a[m] < x THEN L := m+1 ELSE R := m END
END
```

Можно сказать, что теперь ищется не элемент a[m] = x, а граница, отделяющая все элементы, меньшие x, от всех прочих.

Условие окончания – $L \ge R$. Есть ли гарантия его достижения? Чтобы убедиться в этом, нужно показать, что в любых обстоятельствах разность R–L уменьшается на каждом шаге. Условие $L < R$ справедливо в начале каждого шага. Тогда

среднее арифметическое удовлетворяет условию L ≤ m < R. Поэтому разность действительно уменьшается благодаря либо присваиванию значения m+1 переменной L (что увеличивает L), либо значению m переменной R (что уменьшает R), и цикл прекращается при L = R.

Однако выполнение инварианта вместе с условием L = R еще не гарантирует успеха поиска. Разумеется, если R = N, то искомого значения в массиве нет. В противном случае нужно учесть, что элемент a[R] еще не сравнивался. Поэтому нужна дополнительная проверка равенства a[R] = x. В отличие от первого решения, данный алгоритм – как и линейный поиск – находит искомое значение в позиции с наименьшим индексом.

1.8.3. Поиск в таблице

Поиск в массиве иногда называют *поиском в таблице*, особенно если ключи сами являются составными объектами, такими как массивы чисел или литер. Последний случай встречается часто; массивы литер называют *цепочками литер* (string), или *словами*. Определим тип String так:

```
String = ARRAY M OF CHAR
```

и пусть отношение порядка для цепочек определено так, что для цепочек x и y:

$$(x = y) \equiv (\mathbf{A}j: 0 \le j < M : x_j = y_j)$$
$$(x < y) \equiv \mathbf{E}i: 0 \le i < N : ((\mathbf{A}j: 0 \le j < i : x_j = y_j) \;\&\; (x_i < y_i))$$

Очевидно, равенство цепочек эквивалентно равенству всех литер. В сущности, достаточно искать пару неравных литер. Отсутствие такой пары означает равенство цепочек. Будем предполагать, что длина слов достаточно мала, скажем меньше 30, и будем использовать линейный поиск.

В большинстве приложений нужно считать, что цепочки литер имеют переменную длину. Это достигается тем, что с каждой цепочкой литер ассоциируется ее длина. Учитывая данное выше определение типа, эта длина не должна превышать максимального значения M. Такая схема оставляет достаточно гибкости для многих задач, избегая при этом сложностей динамического распределения памяти. Чаще всего используются два представления длины цепочек литер:

1. Длина задается неявно тем, что к концу цепочки литер приписывается так называемая *концевая* литера, которая в других целях не используется. Обычно для этой цели используют литеру 0X, не имеющую графического образа. (Для дальнейшего важно, что это *самая младшая* литера во всем множестве литер.)
2. Длина хранится непосредственно в первом элементе массива, то есть цепочка литер s имеет вид

   $$s = s_0, s_1, s_2, \ldots, s_{N-1}$$

   где $s_1 \ldots s_{N-1}$ суть фактические литеры цепочки, а s_0 = CHR(N). Преимущество этого способа – в том, что длина доступна непосредственно, а недоста-

ток – что максимальная длина ограничена размером множества литер, то есть в случае множества ASCII значением 256.

В дальнейшем мы придерживаемся первой схемы. Сравнение цепочек литер принимает вид

```
i := 0;
WHILE (x[i] # 0X) & (x[i] = y[i]) DO i := i+1 END
```

Здесь концевая литера играет роль барьера, а инвариант цикла имеет вид

$$\mathbf{A}j\colon 0 \le j < i : x_j = y_j \neq 0X,$$

и в результате выполняется следующее условие:

$$((x_i = 0X) \text{ OR } (x_i \neq y_i)) \;\&\; (\mathbf{A}j\colon 0 \le j < i : x_j = y_j \neq 0X).$$

Отсюда следует совпадение цепочек x и y, если $x_i = y_i$; при этом $x < y$, если $x_i < y_i$.

Теперь мы готовы вернуться к поиску в таблицах. Здесь требуется «поиск в поиске», то есть поиск среди элементов таблицы, причем для каждого элемента таблицы нужна последовательность сравнений между компонентами массивов. Например, пусть таблица T и аргумент поиска x определены так:

```
T: ARRAY N OF String;
x: String
```

Предполагая, что N может быть большим и что таблица упорядочена по алфавиту, будем использовать поиск делением пополам. Используя построенные выше алгоритмы для поиска делением пополам и для сравнения строк, получаем следующий программный фрагмент:

```
i := -1;                                                    (* ADruS183 *)
L := 0; R := N;
WHILE L < R DO
    m := (L+R) DIV 2;
    i := 0;
    WHILE (x[i] # 0X) & (T[m,i] = x[i]) DO i := i+1 END;
    IF T[m,i] < x[i] THEN L := m+1 ELSE R := m END
END;
IF R < N THEN
    i := 0;
    WHILE (x[i] # 0X) & (T[R,i] = x[i]) DO i := i+1 END
END
(* (R < N) & (T[R,i] = x[i])  устанавливают совпадение *)
```

1.9. Поиск образца в тексте (string search)

Часто встречается особый вид поиска – *поиск образца в тексте*. Он определяется следующим образом. Пусть даны массив s из N элементов и массив p из M элементов, где $0 < M < N$:

```
s: ARRAY N OF Item
p: ARRAY M OF Item
```

Требуется найти первое вхождение p в s. Обычно элементы этих массивов (Item) являются литерами; тогда s можно считать текстом, а p – образцом (pattern) или словом, и тогда нужно найти первое вхождение слова в тексте. Это основная операция в любой системе обработки текстов, и для нее важно иметь эффективный алгоритм.

Особенность этой задачи – наличие двух массивов данных и необходимость их одновременного просмотра, причем координация движения индексов-бегунков, с помощью которых осуществляются просмотры, определяется данными. Корректная реализация подобных «сплетенных» циклов упрощается, если выражать их, используя так называемый *цикл Дейкстры* – многоветочный вариант цикла WHILE. Поскольку эта фундаментальная и мощная управляющая структура представлена не во всех языках программирования, ее описанию посвящено приложение С.

Мы последовательно рассмотрим три алгоритма: простой поиск, построенный самым наивным образом; алгоритм Кнута, Морриса и Пратта (КМП-алгоритм), представляющий собой оптимизацию простого поиска; и, наконец, самый эффективный из трех алгоритм Бойера и Мура (БМ-алгоритм), основанный на пересмотре базовой идеи простого алгоритма.

1.9.1. Простой поиск образца в тексте

Прежде чем думать об эффективности, опишем простейший алгоритм поиска. Назовем его *простым поиском образца в тексте*. Удобно иметь в виду рис. 1.9, на котором схематически показан образец p длины M, сопоставляемый с текстом s длины N в позиции i. При этом индекс j нумерует элементы образца, и элементу образца p[j] сопоставляется элемент текста s[i+j].

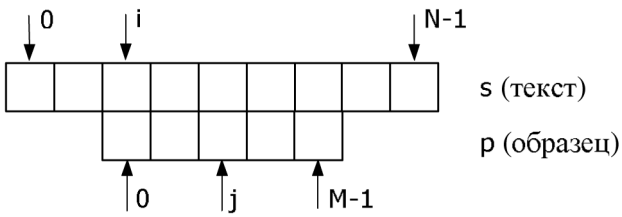


Рис. 1.9. Образец длины M, сопоставляемый с текстом s в позиции i

Предикат R(i), описывающий полное совпадение образца с литерами текста в позиции i, формулируется так:

$$R(i) = \mathbf{A}k: 0 \le j < M : p_j = s_{i+j}$$

Допустимые значения i, в которых может реализоваться совпадение, – от 0 до N–M включительно. Вычисление условия R естественным образом сводится к повторным сравнениям отдельных пар литер. Если применить теорему де Моргана к R, то окажется, что эти повторные сравнения должны представлять собой поиск на неравенство среди пар соответствующих литер текста и образца:

$$R(i) = (\mathbf{A}j: 0 \le j < M : p_j = s_{i+j}) = \sim(\mathbf{E}j: 0 \le j < M : p_j \ne s_{i+j})$$

Поэтому предикат R(i) легко реализуется в виде процедуры-функции, построенной по схеме линейного поиска:

```
PROCEDURE R (i: INTEGER): BOOLEAN;
    VAR j: INTEGER;
BEGIN
    (* 0 <= i < N *)
    j := 0;
    WHILE (j < M) & (p[j] = s[i+j]) DO INC(j) END;
    RETURN ~(j < M)
END R
```

Пусть искомым результатом будет значение индекса i, указывающее на *первое* вхождение образца в тексте s. Тогда должно удовлетворяться условие R(i). Но так как требуется найти именно первое вхождение образца, то условие R(k) должно быть ложным для всех k < i. Обозначим это новое условие как Q(i):

$$Q(i) = \mathbf{A}k: 0 \le k < i : \sim R(k)$$

Такая формулировка задачи сразу наводит на мысль построить программу по образцу линейного поиска (раздел 1.8.1):

```
i := 0;                                          (* ADruS191_Простой *)
WHILE (i <= N-M) & ~R(i) DO INC(i) END
```

Инвариантом этого цикла является предикат Q(i), который выполняется как до инструкции INC(i), так и – благодаря второму операнду охраны – после нее.

Достоинство этого алгоритма – легкость понимания логики благодаря четкой развязке двух циклов поиска за счет упрятывания одного из них внутрь процедуры-функции R. Однако это же свойство может обернуться и недостатком: во-первых, дополнительный вызов процедуры на каждом шаге потенциально длинного цикла может оказаться слишком дорогостоящим для такой базовой операции, как поиск образца. Во-вторых, более совершенные алгоритмы, рассматриваемые ниже, используют информацию, получаемую во внутреннем цикле, для увеличения i во внешнем цикле на величину, большую единицы, так что два цикла уже нельзя рассматривать как вполне независимые. Можно избавиться от процедуры вычисления R, введя дополнительную логическую переменную для ее результата и вложив цикл из процедуры в тело основного цикла по i. Однако логика взаимодействия двух вложенных циклов через логическую переменную теряет исходную прозрачность, что чревато ошибками при эволюции программы.

В подобных случаях удобно использовать так называемый *цикл Дейкстры* – цикл WHILE с несколькими ветвями, каждая из которых имеет свою охрану (см. приложение С). В данном случае две ветви будут соответствовать шагам по i и j соответственно. Вспомним рис. 1.9 и введем предикат P(i, j), означающий совпадение первых j литер образца с литерами текста, начиная с позиции i:

$P(i, j) = \mathbf{A}k: 0 \le k < j : s_{i+k} = p_k$

Тогда R(i) = P(i, M).

Рисунок 1.9 показывает, что текущее состояние процесса поиска характеризуется значениями пары переменных i и j. При этом инвариант (условие, которому удовлетворяет состояние поиска и к которому процесс должен возвращаться после каждого шага при новых i или j) можно выбрать так: в позициях до i совпадений образца нет, а в позиции i имеется частичное совпадение первых j литер образца, что формально записывается следующим образом:

Q(i) & P(i, j)

Очевидно, j = M означает, что имеет место искомое вхождение образца в текст в позиции i, а i > N – M означает, что вхождений нет вообще.

Очевидно, для поиска достаточно пытаться увеличивать j на 1, чтобы расширить совпадающий сегмент, а если это невозможно, то продвинуть образец в новую позицию, увеличив i на 1, и сбросить j в нуль, чтобы начать с начала проверку совпадения образца в новой позиции:

```
i := 0;  j := 0;
WHILE можно расширить совпадающий сегмент DO
    INC( j )
ELSIF можно продвинуть образец DO
    INC( i );  j := 0
END;
```

Остается сосредоточиться на каждом из двух шагов по отдельности и аккуратно выписать условия, при которых каждый шаг имеет смысл, то есть сохраняет инвариант. Для первой ветки это условие (i ≤ N–M) & (j < M) & ($s_{i+j} = p_j$), гарантирующее P(i, j) после увеличения j. Для второй ветки последний операнд конъюнкции должен содержать неравенство вместо равенства, что влечет ~R(i) и гарантирует Q(i) после увеличения i. Учитывая, что охраны веток вычисляются в порядке их перечисления в тексте, можно опустить последний операнд конъюнкции во второй охране и получить окончательную программу:

```
i := 0;  j := 0;                                    (* ADruS191_Простой *)
WHILE (i <= N-M) & (j < M) & (s[i+j] = p[j]) DO
    INC( j )
ELSIF (i <= N-M) & (j < M) DO
    INC( i );  j := 0
END;
```

После цикла гарантировано условие, равное конъюнкции отрицаний всех охран, то есть (i > N–M) OR (j >= M), причем из структуры шагов цикла дополнитель-

но следует, что два операнда не могут быть истинны одновременно, а j не может превысить M. Тогда i > N–M означает, что вхождений образца нет, а j = M – что справедливо Q(i) & P(i, M), то есть найдено искомое полное вхождение в позиции i.

Анализ простого поиска в тексте. Этот алгоритм довольно эффективен, если предполагать, что неравенство литер обнаруживается лишь после небольшого числа сравнений литер, то есть при небольших j. Такое вполне вероятно, если мощность типа элементов велика. Для поиска в тексте с мощностью множества литер, равной 128, разумно предположить, что неравенство имеет место уже после проверки одной или двух литер. Тем не менее поведение в худшем случае вызывает тревогу. Например, рассмотрим текст, состоящий из N–1 букв A, за которыми следует единственная буква B, и пусть образец состоит из M–1 литер A, за которыми следует одна B. Тогда нужно порядка N*M сравнений, чтобы обнаружить совпадение в конце текста. К счастью, как мы увидим в дальнейшем, существуют методы, которые в этом худшем случае ведут себя гораздо лучше.

1.9.2. Алгоритм Кнута, Морриса и Пратта

Около 1970 г. Кнут, Моррис и Пратт придумали алгоритм, который требует только порядка N сравнений литер, причем даже в худшем случае [1.8]. Алгоритм основан на том наблюдении, что, всегда сдвигая образец по тексту только на единицу, мы не используем полезную информацию, полученную в предыдущих сравнениях. Ведь после частичного совпадения начала образца с соответствующими литерами в тексте мы фактически знаем пройденную часть текста и могли бы использовать заранее подготовленную информацию (полученную из анализа образца) для более быстрого продвижения по тексту. Следующий пример, в котором ищется слово Hooligan, иллюстрирует идею алгоритма. Подчеркнуты литеры, которые уже сравнивались. Каждый раз, когда сравниваемые литеры оказываются не равны, образец сдвигается на весь уже пройденный путь, так как полное совпадение с образцом заведомо невозможно для слова Hooligan при меньшем сдвиге:

```
Hoola-Hoola girls like Hooligans.
Hooligan
   Hooligan
    Hooligan
     Hooligan
        Hooligan
         Hooligan
         ......
                  Hooligan
```

Здесь, в отличие от простого алгоритма, точка сравнения (позиция очередного элемента в тексте, сравниваемого с некоторым элементом образца) никогда не сдвигается назад. Именно эту точку сравнения (а не позицию первого элемента образца) будем теперь хранить в переменной i; переменная j, как и раньше, будет указывать на соответствующий элемент образца (см. рис. 1.10).

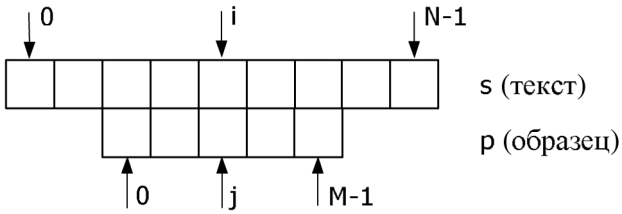


Рис. 1.10. В обозначениях алгоритма КМП позиция в тексте первой литеры образца равна i–j (а не i, как в простом алгоритме)

Центральным пунктом алгоритма является сравнение элементов s[i] и p[j], при равенстве i и j одновременно сдвигаются вправо на единицу, а в случае неравенства должен быть сдвинут образец, что выражается присваиванием j некоторого меньшего значения D. Граничный случай j = 0 показывает, что нужно предусмотреть возможность сдвига образца целиком за текущую позицию сравнения (чтобы элемент p[0] оказался выравнен с s[i+1]). Для такого случая удобно положить D = –1. Главный цикл алгоритма приобретает следующий вид:

```
i := 0;  j := 0;
WHILE (i < N) & (j < M) & ((j < 0) OR (s[i] = p[j])) DO
    INC( i );  INC( j )
ELSIF (i < N) & (j < M) DO (* ((j >= 0) & (s[i] # p[j]) *)
    j := D
END;
```

Эта формулировка не совсем полна, так как еще не определено значение сдвига D. Мы к этому вернемся чуть ниже, а пока отметим, что инвариант здесь берется как в предыдущей версии алгоритма; в новых обозначениях это Q(i–j) & P(i–j, j).

Постусловие цикла, вычисленное как конъюнкция отрицаний охран, – это выражение (j >= M) OR (i >= N), но реально могут реализоваться только равенства. Если алгоритм останавливается при j = M, то имеет место совпадение с образцом в позиции i–M (член P(i–j, j) инварианта влечет P(i–M, M) = R(i)). В противном случае остановка происходит с i = N, и так как здесь j < M, то первый член инварианта, Q(i–j), влечет отсутствие совпадений с образцом во всем тексте.

Нужно убедиться, что инвариант в алгоритме всегда сохраняется. Очевидно, инвариант выполнен в начале цикла при значениях i = j = 0. Не нарушается он и в первой ветке цикла, то есть после выполнения двух операторов, увеличивающих i и j на 1. В самом деле, Q(i–j) не нарушается, так как разность i–j не меняется. Не нарушается и P(i–j, j) при увеличении j благодаря равенству в охране ветки (ср. определение P). Что касается второй ветки, то мы просто потребуем, чтобы значение D всегда было таким, чтобы замена j на D сохраняла инвариант.

Присваивание j := D при условии D < j означает сдвиг образца вправо на j–D позиций. Хотелось бы сделать этот сдвиг как можно больше, то есть сделать D как можно меньше. Это иллюстрируется на рис. 1.11.

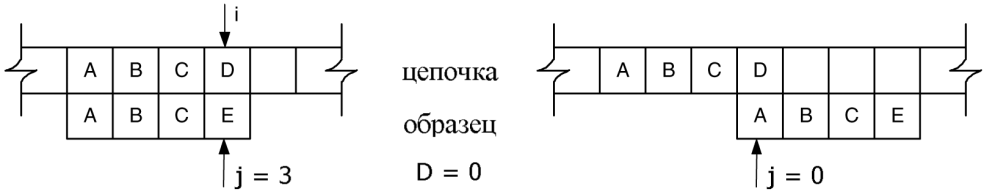


Рис. 1.11. Присваивание j := D сдвигает образец на j–D позиций

Если мы хотим, чтобы инвариант P(i–j, j) & Q(i–j) выполнялся после присваивания j := D, то до присваивания должно выполняться условие P(i–D, D) & Q(i–D). Это предусловие и будет ключом к поиску подходящего выражения для D наряду с условием P(i–j, j) & Q(i–j), которое предполагается уже выполненным перед присваиванием (к этой точке программы относятся все дальнейшие рассуждения).

Решающее наблюдение состоит в том, что истинность P(i–j, j) означает

$$p_0 \dots p_{j-1} = s_{i-j} \dots s_{i-1}$$

(ведь мы только что просмотрели первые j литер образца и установили их совпадение с соответствующими литерами текста). Поэтому условие P(i–D, D) при $D < j$, то есть

$$p_0 \dots p_{D-1} = s_{i-D} \dots s_{i-1}$$

превращается в уравнение для D:

$$p_0 \dots p_{D-1} = p_{j-D} \dots p_{j-1}$$

Что касается Q(i–D), то это условие следует из Q(i–j), если ~R(i–k) для k = D+1 ... j. При этом истинность ~R(i–k) для j = k гарантируется неравенством s[i] # p[j]. Хотя условия ~R(i–k) ≡ ~P(i–k, M) для k = D+1 ... j–1 нельзя полностью вычислить, используя только уже прочитанный фрагмент текста, но зато можно вычислить достаточные условия ~P(i–k,k). Раскрывая их с учетом уже найденных равенств между элементами s и p, получаем следующее условие:

$$p_0 \dots p_{k-1} \neq p_{j-k} \dots p_{j-1} \quad \text{для всех } k = D+1 \dots j-1$$

То есть D должно быть *максимальным* решением вышеприведенного уравнения. Рисунок 1.12 показывает, как работает этот механизм сдвигов.

Если решения для D нет, то совпадение с образцом невозможно ни в какой позиции ниже i+1. Тогда полагаем D := –1. Такая ситуация показана в верхнем примере на рис. 1.13.

Последний пример на рис. 1.12 подсказывает, что алгоритм можно еще чуть-чуть улучшить: если бы литера p_j была равна A вместо F, то мы знали бы, что соответствующая литера в тексте никак не может быть равна A, и сразу в следующей итерации цикла пришлось бы выполнить сдвиг образца с D = –1 (ср. нижнюю часть рис. 1.13). Поэтому при поиске D можно сразу наложить дополнительное ограничение $p_D \neq p_j$. Это позволяет в полной мере использовать информацию из неравенства в охране данной ветки цикла.

Примеры

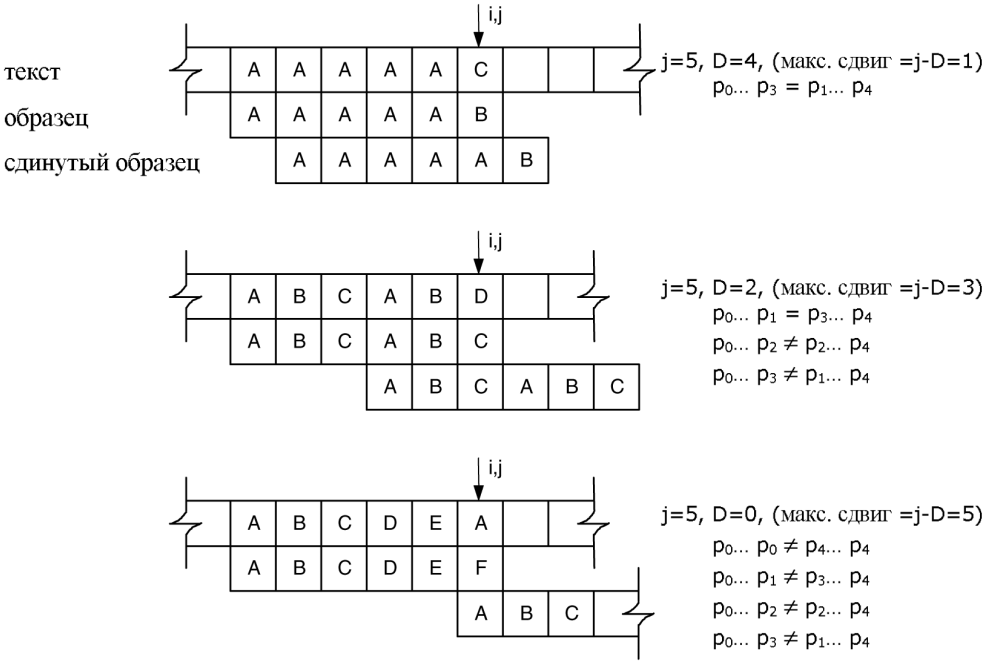


Рис. 1.12. Частичные совпадения образца и вычисление D

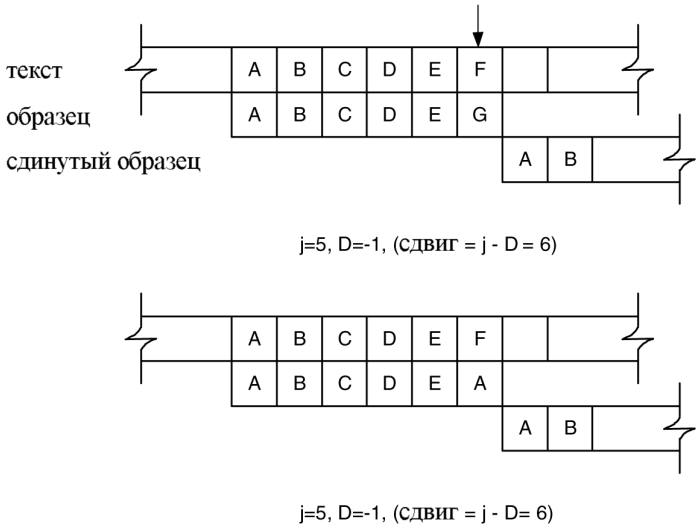


Рис. 1.13. Сдвиг образца за позицию последней литеры

Важный итог состоит в том, что значение D определяется только образцом и значением j, но не зависит от текста. Обозначим D для заданного j как d_j. Таблицу d можно построить до начала собственно поиска, что можно назвать предкомпиляцией образца. Очевидно, такие накладные расходы оправдаются, только если текст значительно длиннее образца (M << N). Если нужно найти несколько вхождений одного и того же образца, то таблицу d можно использовать повторно.

Величина $d_j < j$ является длиной максимальной последовательности, удовлетворяющей условию

$$p_0 \ldots p_{d[j]-1} = p_{j-d[j]} \ldots p_{j-1}$$

с дополнительным ограничением $p_{d[j]} \neq p_j$. Проход построенного цикла по самому образцу p вместо s последовательно находит максимальные последовательности, что позволяет вычислить d_j:

```
PROCEDURE Search (VAR p, s: ARRAY OF CHAR; M, N: INTEGER; VAR r: INTEGER);
                                                        (* ADruS192_КМП *)
    (*поиск образца p длины M в тексте s длины N; M <= Mmax*)
    (*если p найден, то r указывает его позицию в s, иначе r = -1*)
    VAR i, j, k: INTEGER;
        d: ARRAY Mmax OF INTEGER;
BEGIN
    (*вычислить d из p*)
    d[0] := -1;
    IF p[0] # p[1] THEN d[1] := 0 ELSE d[1] := -1 END;
    j := 1;  k := 0;
    WHILE (j < M-1) & (k >= 0) & (p[j] # p[k]) DO
        k := d[k]
    ELSIF j < M-1 DO (* (k < 0) OR (p[j] = p[k]) *)
        INC( j );  INC( k );
        IF p[j] # p[k] THEN d[j] := k ELSE d[j] := d[k] END; ASSERT( d[j] = D(j) );
    END;
    (*собственно поиск*)
    i := 0;  j := 0;
    WHILE (j < M) & (i < N) & (j >= 0) & (s[i] # p[j]) DO
        j := d[j];
    ELSIF (j < M) & (i < N) DO
        INC(i);  INC(j);
    END;
    IF j = M THEN r := i-M ELSE r := -1 END
END Search
```

Анализ КМП-алгоритма. Точный анализ эффективности КМП-алгоритма, как и сам алгоритм, весьма непрост. В работе [1.8] его изобретатели доказали, что число сравнений литер имеет порядок M+N, что гораздо лучше, чем M*N в простом поиске. Они также указали на то приятное обстоятельство, что указатель i всегда движется по тексту только вперед, тогда как в простом поиске просмотр текста всегда начинается с первой литеры образца после обнаружения неравенства литер, и поэтому уже просмотренные литеры могут просматриваться снова. Это может привести к трудностям, если текст считывается из внешней памяти, так как

в таких случаях обратный ход по тексту может дорого обойтись. Даже если входные данные буферизуются, образец может оказаться таким, что потребуется возврат за пределы буфера.

1.9.3. Алгоритм Бойера и Мура

Хитроумная схема КМП-алгоритма дает выигрыш, только если несовпадение обнаруживается после частичного совпадения некоторого фрагмента. Только в таком случае происходит сдвиг образца более чем на одну позицию. Увы, это скорее исключение, чем правило: совпадения встречаются реже, чем несовпадения. Поэтому выигрыш от использования КМП-стратегии оказывается не слишком существенным в большинстве случаев обычного поиска в текстах. Метод, который мы теперь обсудим, улучшает поведение не только в наихудшем случае, но и в среднем. Он был изобретен около 1975 г. Бойером и Муром [1.9], и мы будем называть его *БМ-алгоритмом*. Мы представим его упрощенную версию.

БМ-алгоритм основан на несколько неожиданной идее – начать сравнение литер не с начала, а с конца образца. Как и в КМП-алгоритме, до начала собственно поиска для образца предкомпилируется таблица d. Для каждой литеры x из всего множества литер обозначим как d_x расстояние от конца образца до ее самого правого вхождения. Теперь предположим, что обнаружено несовпадение между текстом и образцом. Тогда образец можно сразу сдвигать вправо на $d_{p[M-1]}$ позиций, и весьма вероятно, что эта величина окажется больше, чем 1. Если p_{M-1} вообще не встречается в образце, сдвиг еще больше и равен длине всего образца. Следующий пример иллюстрирует такую процедуру:

```
Hoola-Hoola girls like Hooligans.
Hooligan
    Hooligan
      Hooligan
            Hooligan
                  Hooligan
```

Так как литеры теперь сравниваются справа налево, то будет удобнее использовать следующие слегка модифицированные версии предикатов P, R и Q:

$$P(i, j) = \mathbf{A}k: j \le k < M : s_{i-M+k} = p_k$$
$$R(i) = P(i, 0)$$
$$Q(i) = \mathbf{A}k: M \le k < i : \sim R(k)$$

В этих терминах инвариант цикла будет иметь прежний вид: Q(i) & P(i, j). Мы еще введем переменную k = i-M+j. Теперь можно дать следующую формулировку БМ-алгоритма:

```
i := M; j := M; k := i;
WHILE (j > 0) & (i <= N) & (s[k-1] = p[j-1]) DO
    DEC(k); DEC(j)
ELSIF (j > 0) & (i <= N) DO
    i := i + d[ORD(s[i-1])]; j := M; k := i;
END
```

Индексы удовлетворяют ограничениям $0 \le j \le M$, $M \le i$, и $0 < k \le i$. Остановка алгоритма с $j = 0$ влечет $P(i, 0) = R(i)$, то есть совпадение в позиции $k = i-M$. Для остановки с $j > 0$ нужно, чтобы $i > N$; тогда $Q(i)$ влечет $Q(N)$ и, следовательно, отсутствие совпадений. Разумеется, нам еще нужно убедиться, что $Q(i)$ и $P(i, j)$ действительно являются инвариантами двух циклов. Они очевидным образом удовлетворены в начале цикла, поскольку $Q(M)$ и $P(x, M)$ всегда истинны.

Сначала рассмотрим первую ветку. Одновременное уменьшение k и j никак не влияет на $Q(i)$, и, следовательно, поскольку уже установлено, что $s_{k-1} = p_{j-1}$, а $P(i, j)$ выполняется до операции уменьшения j, то $P(i, j)$ выполняется и после нее.

Во второй ветке достаточно показать, что присваивание $i := i + d_{s[i-1]}$ не нарушает инвариант $Q(i)$, так как после остальных присваиваний $P(i, j)$ выполняется автоматически. $Q(i)$ выполняется после изменения i, если до него выполняется $Q(i+d_{s[i-1]})$. Поскольку мы знаем, что выполняется $Q(i)$, достаточно установить, что $\sim R(i+h)$ для $h = 1 \,..\, d_{s[i-1]}-1$. Вспомним, что величина d_x определена как расстояние от конца до крайнего правого вхождения x в образце. Формально это выражается в виде:

$$\mathbf{A}k: M-d_x \le k < M-1 : p_k \ne x$$

Для $x = s_{i-1}$ получаем

$$\begin{aligned}
&\mathbf{A}k: M-d_{s[i-1]} \,J\, k < M-1 : s_{i-1} \ne p_k \\
=\ &\mathbf{A}h: 1 \le h \le d_{s[i-1]}-1 : s_{i-1} \ne p_{M-1-h} \\
\Rightarrow\ &\mathbf{A}h: 1 \le h \le d_{s[i-1]}-1 : \sim R(i+h)
\end{aligned}$$

В следующей программе рассмотренная упрощенная стратегия Бойера–Мура оформлена подобно предыдущей программе, реализующей КМП-алгоритм:

```
PROCEDURE Search (VAR s, p: ARRAY OF CHAR; M, N: INTEGER; VAR r: INTEGER);
                                                        (* ADruS193_БМ *)
    (*поиск образца p длины M в тексте s длины N*)
    (*если p найден, то r указывает его позицию в s, иначе r = -1*)
    VAR i, j, k: INTEGER;
        d: ARRAY 128 OF INTEGER;
BEGIN
    FOR i := 0 TO 127 DO d[i] := M END;
    FOR j := 0 TO M-2 DO d[ORD(p[j])] := M-j-1 END;
    i := M; j := M; k := i;
    WHILE (j > 0) & (i <= N) & (s[k-1] = p[j-1]) DO
        DEC(k); DEC(j)
    ELSIF (j > 0) & (i <= N) DO
        i := i + d[ORD(s[i-1])]; j := M; k := i;
    END;
    IF  j <= 0  THEN  r := k  ELSE  r := -1  END
END Search
```

Анализ алгоритма Бойера и Мура. В оригинальной публикации этого алгоритма [1.9] детально изучена и его производительность. Замечательно то, что он всегда требует существенно меньше, чем N, сравнений, за исключением специ-

ально построенных примеров. В самом удачном случае, когда последняя литера образца всегда падает на неравную ей литеру текста, число сравнений равно N/M.

Авторы предложили несколько возможных путей дальнейшего улучшения алгоритма. Один состоит в том, чтобы скомбинировать описанную стратегию, которая обеспечивает большие шаги сдвига для случаев несовпадения, со стратегией Кнута, Морриса и Пратта, которая допускает большие сдвиги после (частичного) совпадения. Такой метод потребует предварительного вычисления двух таблиц – по одной для БМ-алгоритма и для КМП-алгоритма. И тогда можно брать больший из сдвигов, полученных двумя способами, так как оба гарантируют, что меньшие сдвиги не могут дать совпадения. Мы воздержимся от обсуждения деталей, так как дополнительная сложность вычисления таблиц и самого поиска, по-видимому, не приводит к существенному выигрышу в эффективности. Зато при этом увеличиваются накладные расходы, так что непонятно, является ли столь изощренное усовершенствование улучшением или ухудшением.

Замечание переводчика. Три алгоритма поиска образца в тексте – простой, КМП и БМ – иллюстрируют одну закономерность, которую важно понимать любому проектировщику. В простом алгоритме инстинктивно реализуется дебютная идея, что проверку совпадения литер образца с литерами текста надо производить в том же направлении, в котором продвигается по тексту образец. КМП-алгоритм, унаследовав эту идею, не осознавая этого факта, оптимизирует ее добавлением изощренной «заплатки» – механизма продвижения образца на более чем одну позицию с учетом уже просмотренных литер. Зато построение БМ-алгоритма начинается с критического пересмотра самой дебютной идеи простого алгоритма: ведь сравнение литер образца в порядке движения образца по тексту обусловлено, в сущности, лишь инерцией мышления. При этом и ускоряющая «заплатка» здесь оказывается несравненно проще (ср. вычисление вспомогательных таблиц d в двух программах), и итоговый алгоритм существенно быстрее, и его математический анализ сильно облегчается. Это общая закономерность: первый импульс к конкретной деятельности, который программист ощущает, заметив какой-нибудь «очевидный» способ решения (когда, что называется, «руки чешутся» начать писать код), способен помешать увидеть путь к наилучшему решению. Поэтому нужны специальные усилия, требующие дисциплины, для «пристального разглядывания» проблемы, чтобы выявить неявные, инстинктивные предположения – еще до первых попыток составить конкретное решение, когда внимание уже будет поглощено не исходной задачей, а красотой и эффективностью предполагаемого решения, ярко демонстрирующего остроумие и изобретательность программиста, а также его познания в методах оптимизации.

Упражнения

1.1. Обозначим мощности стандартных типов INTEGER, REAL и CHAR как c_{int}, c_{real} и c_{char}. Каковы мощности следующих типов данных, определенных в этой главе в качестве примеров: Complex, Date, Person, Row, Card, Name?

1.2. Какие последовательности машинных инструкций (на вашем компьютере) соответствуют следующим операциям:
   a) чтение и запись для элементов упакованных записей и массивов;
   b) операции для множеств, включая проверку принадлежности числа множеству?

1.3. Каковы могут быть аргументы в пользу определения некоторых наборов данных в виде последовательностей, а не массивов?

1.4. Пусть дано ежедневное железнодорожное расписание для поездов на нескольких линиях. Найдите такое представление этих данных в виде массивов, записей или последовательностей, которое было бы удобно для определения времени прибытия и отправления для заданных станции и направления поезда.

1.5. Пусть даны текст T, представленный последовательностью, а также списки небольшого числа слов в виде двух массивов A и B. Предположим, что слова являются короткими массивами литер небольшой фиксированной максимальной длины. Напишите программу, преобразующую текст T в текст S заменой каждого вхождения слова A_i соответствующим словом B_i.

1.6. Сравните следующие три варианта поиска делением пополам с тем, который был дан в основном тексте. Какие из трех программ правильны? Определите соответствующие инварианты. Какие варианты более эффективны? Предполагается, что определены константа N > 0 и следующие переменные:

```
VAR i, j, k, x: INTEGER;
  a: ARRAY N OF INTEGER;
```

Программа A:

```
i := 0; j := N-1;
REPEAT
  k := (i+j) DIV 2;
  IF a[k] < x THEN i := k ELSE j := k END
UNTIL (a[k] = x) OR (i > j)
```

Программа B:

```
i := 0; j := N-1;
REPEAT
  k := (i+j) DIV 2;
  IF x < a[k] THEN j := k-1 END;
  IF a[k] < x THEN i := k+1 END
UNTIL i > j
```

Программа C:

```
i := 0; j := N-1;
REPEAT
  k := (i+j) DIV 2;
  IF x < a[k] THEN j := k ELSE i := k+1 END
UNTIL i > j
```

Подсказка. Все программы должны заканчиваться с $a_k = x$, если такой элемент присутствует, или $a_k \neq x$, если элемента со значением x нет.

1.7. Некая компания проводит опрос, чтобы определить, насколько популярна ее продукция – записи эстрадных песен, а самые популярные песни должны быть объявлены в хит-параде. Опрашиваемую выборку покупателей нужно разделить на четыре группы в соответствии с полом и возрастом (скажем, не старше 20 и старше 20). Каждого покупателя просят назвать пять любимых песен. Песни нумеруются числами от 1 до N (пусть N = 30). Результаты опроса нужно подходящим образом закодировать в виде последовательности литер.

Подсказка: Используйте процедуры Read и ReadInt для чтения данных опроса.

```
TYPE hit = INTEGER;
  reponse = RECORD name, firstname: Name;
      male: BOOLEAN;
      age: INTEGER;
      choice: ARRAY 5 OF hit
  END;

VAR poll: Files.File
```

Этот файл содержит входные данные для программы, вычисляющей следующие результаты:

1. Список A песен в порядке популярности. Каждая запись списка состоит из номера песни и количества ее упоминаний в опросе. Ни разу не названные песни в список не включаются.
2. Четыре разных списка с фамилиями и именами всех респондентов, которые поставили на первое место один из трех самых популярных хитов в своей группе.

Перед каждым из пяти списков должен идти подходящий заголовок.

Литература

[1.1] Dahl O-.J., Dijkstra E. W., Hoare C. A. R. Structured Programming. F. Genuys, Ed., New York, Academic Press, 1972 (имеется перевод: Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1975).

[1.2] Hoare C. A. R., in Structured Programming [1.1]. P. 83–174 (имеется перевод: Хоор К. О структурной организации данных. В кн. [1.1]. С. 98–197.)

[1.3] Jensen K. and Wirth N. PASCAL – User Manual and Report. Springer-Verlag, 1974 (имеется перевод: Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка. – М.: Финансы и статистика, 1988).

[1.4] Wirth N. Program development by stepwise refinement. Comm. ACM, 14, No. 4 (1971), 221–27.

[1.5] Wirth N. Programming in Modula-2. Springer-Verlag, 1982 (имеется перевод: Вирт Н. Программирование на языке Модула-2. – М.: Мир, 1987).

[1.6] Wirth N. On the composition of well-structured programs. Computing Surveys, 6, No. 4, (1974) 247–259.
[1.7] Hoare C. A. R. The Monitor: An operating systems structuring concept. Comm. ACM, 17, 10 (Oct. 1974), 549–557.
[1.8] Knuth D. E., Morris J. H. and Pratt V. R. Fast pattern matching in strings. SIAM J. Comput., 6, 2, (June 1977), 323–349.
[1.9] Boyer R. S. and Moore J. S. A fast string searching algorithm. Comm. ACM, 20, 10 (Oct. 1977), 762–772.

Глава 2

Сортировка

2.1. Введение

Главная цель этой главы – дать богатый набор примеров, иллюстрирующих использование структур данных, введенных в предыдущей главе, а также показать, что выбор структуры обрабатываемых данных оказывает глубокое влияние на алгоритмы решения задачи. Тема сортировки хороша еще и тем, что показывает, как одна и та же задача может решаться с помощью многих разных алгоритмов, причем у каждого из них есть преимущества и недостатки, которые следует взвесить в условиях конкретной задачи.

Под сортировкой обычно понимают процесс перестановки объектов некоторого множества в определенном порядке. Цель сортировки – облегчить в дальнейшем поиск элементов отсортированного множества. Это очень часто выполняемая, фундаментальная операция. Объекты сортируются в телефонных справочниках, в списках налогоплательщиков, в оглавлениях книг, в библиотеках, в словарях, на складах, то есть почти всюду, где нужно искать хранимые объекты. Даже детей учат приводить вещи «в порядок», и они сталкиваются с некоторыми видами сортировки задолго до того, как начнут изучать арифметику.

Поэтому сортировка – весьма важная операция, особенно в обработке данных. Кажется, ничто не поддается сортировке лучше, чем данные! И все же при изучении сортировки наибольший интерес для нас будут представлять еще более фундаментальные приемы, используемые при построении алгоритмов. Нелегко найти приемы, которые не использовались бы так или иначе в связи с алгоритмами сортировки. При этом сортировка – идеальная тема для демонстрации широкого разнообразия алгоритмов, решающих одну и ту же задачу, причем многие из них в каком-нибудь смысле оптимальны, и почти каждый из них имеет какие-нибудь преимущества перед остальными. Поэтому здесь хорошо видно, зачем нужен анализ эффективности алгоритмов. Более того, на примере сортировок становится понятно, что можно получить весьма существенный выигрыш в эффективности, разрабатывая хитроумные алгоритмы, даже если уже имеются очевидные методы.

Влияние структуры обрабатываемых данных на выбор алгоритма – само по себе обычное явление – в случае сортировок проявляется настолько сильно, что методы сортировки обычно делят на два типа, а именно сортировки массивов и сортировки (последовательных) файлов. Эти два типа часто называют *внутренней* и *внешней сортировками*, так как массивы хранятся в быстрой, допускающей произвольный доступ внутренней оперативной памяти, а файлы уместны при работе с медленными, но вместительными внешними устройствами хранения (дисками и лентами). Важность такого деления очевидна из примера сортировки пронумерованных карточек. Представление карточек массивом соответствует раскладыванию их перед сортировщиком так, чтобы каждая карточка была непосредственно видна и доступна (см. рис. 2.1).

Однако при файловой организации карточек в каждой стопке видна только верхняя карточка (см. рис. 2.2).

Рис. 2.1. Сортировка массива

Рис. 2.2. Сортировка файла

Очевидно, что такое ограничение будет иметь серьезные последствия для метода сортировки, но оно неизбежно, если карточек так много, что они не умещаются на столе все сразу.

Сначала введем некоторые термины и обозначения, которые будут использоваться на протяжении этой главы. Если даны n элементов

$$a_0, a_1, \ldots, a_{n-1}$$

то сортировка состоит в их перестановке к виду

$$a_{k0}, a_{k1}, \ldots, a_{k[n-1]}$$

таким образом, чтобы для некоторой упорядочивающей функции f

$$f(a_{k0}) \le f(a_{k1}) \le \ldots \le f(a_{k[n-1]})$$

Часто значение такой упорядочивающей функции не вычисляется, а хранится как явная компонента (поле) в каждом элементе. Это значение называется *ключом* элемента. Следовательно, записи особенно хорошо подходят для представления элементов и могут быть объявлены, например, следующим образом:

```
TYPE Item =  RECORD  key:  INTEGER;
                (*здесь объявляются другие компоненты*)
             END
```

Другие компоненты представляют данные, нужные для иных целей, а ключ используется просто для идентификации элементов. Однако для алгоритмов сортировки важен только ключ. Поэтому в дальнейшем мы будем игнорировать любую дополнительную информацию и примем, что тип Item определен как INTEGER. Такой выбор типа ключа несколько произволен. Очевидно, с равным успехом можно использовать любой тип, для которого определено отношение полного порядка.

Метод сортировки называют *устойчивым*, если он сохраняет относительный порядок элементов с равными ключами. Устойчивость сортировки часто желательна, если элементы уже упорядочены (отсортированы) по некоторым вторичным ключам, то есть свойствам, не отраженным в основном ключе.

Данная глава не является всеобъемлющим обзором методов сортировки. Мы предпочитаем уделить больше внимания некоторым избранным алгоритмам. Интересующийся читатель найдет тщательный разбор сортировок в великолепном и полном обзоре Кнута [2.7] (см. также Лорин [2.8]).

2.2. Сортировка массивов

Главное требование при разработке алгоритмов сортировки массивов – экономное использование доступной оперативной памяти. Это означает, что перестановки, с помощью которых упорядочиваются элементы, должны выполняться *in situ* (лат.: на месте – *прим. перев.*), то есть не требуя дополнительной временной памяти, так что методы, которые требуют копирования элементов из массива a в массив-результат b, не представляют интереса. Ограничив таким образом выбор методов, попробуем классифицировать их в соответствии с эффективностью, то есть временем работы. Хорошая мера эффективности – число необходимых сравнений ключей C, а также число перестановок элементов M. Эти величины являются функциями числа сортируемых элементов n. Хотя хорошие алгоритмы сортировки требуют порядка n*log(n) сравнений, мы сначала рассмотрим несколько простых и очевидных методов, которые называются *простыми* и требуют порядка n^2 сравнений ключей. Есть три важные причины, почему нужно рассмотреть простые методы, прежде чем переходить к быстрым алгоритмам:

1. Простые методы особенно хороши для обсуждения основных принципов сортировки.
2. Соответствующие программы легко понять. К тому же они короткие: ведь нужно помнить, что программы тоже занимают место в памяти!

3. Хотя более изощренные методы требуют меньшего числа операций, но эти операции обычно сложнее устроены; поэтому простые методы оказываются быстрее для малых n, хотя их нельзя использовать для больших n.

Алгоритмы сортировки, упорядочивающие элементы *in situ*, могут быть разделены на три основные категории в соответствии с используемым приемом:

- сортировка вставками;
- сортировка выбором;
- сортировка обменом.

Изучим и сравним эти три подхода. Алгоритмы будут работать с глобальной переменной a, чьи компоненты нужно отсортировать *in situ*. Компонентами являются сами ключи. Для простоты мы игнорируем прочие данные, хранящиеся в записях типа Item. Во всех алгоритмах, которые будут разработаны в этой главе, предполагается наличие массива a и константы n, равной числу элементов массива a:

```
TYPE Item = INTEGER;
VAR a: ARRAY n OF Item
```

2.2.1. Простая сортировка вставками

Этот метод часто используется игроками в карты. Элементы (карты) мысленно разделяются на последовательность-приемник $a_0 \ldots a_{i-1}$ и последовательность-источник $a_i \ldots a_{n-1}$. На каждом шаге, начиная с i = 1 и затем увеличивая i на единицу, в последовательности-источнике берется i-й элемент и ставится в правильную позицию в последовательности-приемнике. В табл. 2.1 работа сортировки вставками показана на примере восьми взятых наугад чисел.

Алгоритм простых вставок имеет вид

```
FOR i := 1 TO n-1 DO
    x := a[i];
    вставить x в правильную позицию среди a0 ... ai-1
END
```

Таблица 2.1. Пример работы простой сортировки вставками

Начальные ключи:	44	**55**	12	42	94	18	06	67
i=1	44	55	**12**	42	94	18	06	67
i=2	12	44	55	**42**	94	18	06	67
i=3	12	42	44	55	**94**	18	06	67
i=4	12	42	44	55	94	**18**	06	67
i=5	12	18	42	44	55	94	**06**	67
i=6	06	12	18	42	44	55	94	**67**
i=7	06	12	18	42	44	55	67	94

При поиске правильной позиции для элемента удобно чередовать сравнения и пересылки, то есть позволять значению x «просеиваться» вниз, при этом сравниваем x со следующим элементом a_j, а затем либо вставляем x, либо передвигаем a_j вправо и переходим влево. Заметим, что есть два разных условия, которые могут вызвать прекращение процесса просеивания:

1. У элемента a_j ключ оказался меньше, чем ключ x.
2. Обнаружен левый конец последовательности-приемника.

```
PROCEDURE StraightInsertion;                          (* ADruS2_Sorts *)
   VAR i, j: INTEGER; x: Item;
BEGIN
   FOR i := 1 TO n-1 DO
      x := a[i]; j := i;
      WHILE (j > 0) & (x < a[j-1]) DO
         a[j] := a[j-1]; DEC(j)
      END;
      a[j] := x
   END
END StraightInsertion
```

Анализ простой сортировки вставками. Число сравнений ключей в i-м просеивании C_i не больше i–1, как минимум равно 1 и – предполагая, что все перестановки n ключей равновероятны, – в среднем равно i/2. Число M_i пересылок (присваиваний элементов) равно $C_i + 1$. Поэтому полные числа сравнений и пересылок равны

$C_{min} = n - 1$ $\quad M_{min} = 2*(n - 1)$
$C_{ave} = (n^2 - n)/4$ $\quad M_{ave} = (n^2 + 3n - 4)/4$
$C_{max} = (n^2 - 3n + 2)/2$ $\quad M_{max} = (n^2 - n)/2$

Минимальные значения получаются, когда элементы изначально упорядочены; наихудший случай имеет место, когда элементы изначально стоят в обратном порядке. В этом смысле сортировка вставками ведет себя вполне естественно. Ясно также, что представленный алгоритм является устойчивой сортировкой: относительный порядок элементов с равными ключами не нарушается.

Этот алгоритм нетрудно усовершенствовать, заметив, что последовательность-приемник $a_0 \ldots a_{i-1}$, в которую нужно вставить новый элемент, уже упорядочена. Поэтому можно использовать более быстрый способ нахождения позиции вставки. Очевидный выбор – алгоритм деления пополам, в котором проверяется середина последовательности-приемника и затем продолжаются деления пополам, пока не будет найдена точка вставки. Такой модифицированный алгоритм называется *сортировкой двоичными вставками*:

```
PROCEDURE BinaryInsertion;                            (* ADruS2_Sorts *)
   VAR i, j, m, L, R: INTEGER; x: Item;
BEGIN
   FOR i := 1 TO n-1 DO
      x := a[i]; L := 0; R := i;
```

```
        WHILE L < R DO
            m := (L+R) DIV 2;
            IF a[m] <= x THEN L := m+1 ELSE R := m END
        END;
        FOR j := i TO R+1 BY -1 DO a[j] := a[j-1] END;
        a[R] := x
    END
END BinaryInsertion
```

Анализ сортировки двоичными вставками. Позиция вставки найдена, если L = R. Поэтому интервал поиска должен в итоге иметь длину 1; для этого нужно делить интервал длины i пополам log(i) раз. Поэтому

$$C = \sum i: 0 \le i \le n-1: \lceil \log(i) \rceil$$

Аппроксимируем эту сумму интегралом

$$\text{Integral } (0:n-1)\ \log(x)\ dx = n*(\log(n) - c) + c$$

где c = log(e) = 1/ln(2) = 1.44269... .

В сущности, число сравнений не зависит от первоначального порядка элементов. Однако из-за округления при делении интервала истинное число сравнений для i элементов может быть на 1 больше, чем ожидается. Причина этого отклонения – в том, что позиции вставки в нижней части последовательности в среднем находятся немного быстрее, чем позиции в верхней части, что благоприятствует тем случаям, когда элементы изначально сильно разупорядочены. На самом деле число сравнений минимально, когда элементы изначально стоят в обратном порядке, и максимально, когда они уже упорядочены. Здесь мы имеем пример неестественного поведения алгоритма сортировки. Тогда число сравнений примерно равно

$$C \approx n*(\log(n) - \log(e) \pm 0.5)$$

К сожалению, использование двоичного поиска уменьшает только число сравнений, но не число пересылок. На самом деле пересылки элементов, то есть ключей и прочей информации, обычно более затратны, чем сравнение двух ключей, так что получающееся здесь улучшение не принципиально: важная величина M все еще имеет порядок n^2. Причем сортировка уже упорядоченного массива требует больше времени, чем при использовании простых вставок с последовательным поиском.

Этот пример показывает, что «очевидное улучшение» вполне может оказаться гораздо менее полезным, чем ожидалось, а в некоторых случаях (которые действительно встречаются) «улучшение» может оказаться и ухудшением. Приходится заключить, что сортировка вставками не выглядит удачным методом для применения в компьютерных программах: вставка элемента, при которой весь ряд элементов сдвигается только на одну позицию, не является экономной. Лучших результатов можно ожидать от метода, в котором пересылались бы только одиночные элементы, причем на более далекие расстояния. Это наблюдение приводит к сортировке выбором.

2.2.2. Простая сортировка выбором

Данный метод основан на следующей схеме:

1. Выберем элемент с наименьшим ключом.
2. Переставим его с первым элементом a_0.
3. Повторим эти операции с остальными n–1 элементами, затем с n–2 элементами.., пока не останется только один – наибольший – элемент.

Проиллюстрируем метод на той же последовательности из восьми ключей, что и в табл. 2.1:

Таблица 2.2. Пример простой сортировки выбором

Начальные ключи:	44	55	12	42	94	18	**06**	67
i=1	06	55	**12**	42	94	18	44	67
i=2	06	12	55	42	94	**18**	44	67
i=3	06	12	18	**42**	94	55	44	67
i=4	06	12	18	42	94	55	**44**	67
i=5	06	12	18	42	44	**55**	94	67
i=6	06	12	18	42	44	55	94	**67**
i=7	06	12	18	42	44	55	67	94

Алгоритм формулируется следующим образом:

```
FOR i := 0 TO n–1 DO
    присвоить k индекс наименьшего элемента из a_i ... a_n–1;
    переставить a_i и a_k
END
```

Этот метод можно назвать *простой сортировкой выбором*. В некотором смысле он противоположен простой сортировке вставками: в последней на каждом шаге, чтобы найти позицию вставки, рассматривается один очередной элемент массива-источника и все элементы массива-приемника; в простой сортировке выбором просматривается весь массив-источник, чтобы найти один элемент с наименьшим ключом, который должен быть вставлен в массив-приемник.

```
PROCEDURE StraightSelection;                          (* ADruS2_Sorts *)
    VAR i, j, k: INTEGER; x: Item;
BEGIN
    FOR i := 0 TO n–2 DO
        k := i; x := a[i];
        FOR j := i+1 TO n–1 DO
            IF a[j] < x THEN k := j; x := a[k] END
        END;
        a[k] := a[i]; a[i] := x
    END
END StraightSelection
```

Анализ простой сортировки выбором. Очевидно, число C сравнений ключей не зависит от начального порядка ключей. В этом смысле можно сказать, что этот метод ведет себя не так естественно, как метод простых вставок. Получаем:

$$C = (n^2 - n)/2$$

Число M пересылок не меньше, чем

$$M_{min} = 3*(n - 1)$$

для изначально упорядоченных ключей, и не больше, чем

$$M_{max} = n^2/4 + 3*(n - 1),$$

если изначально ключи стояли в обратном порядке. Чтобы определить M_{avg}, будем рассуждать так: алгоритм просматривает массив, сравнивая каждый элемент с наименьшим значением, найденным до сих пор, и если элемент оказывается меньше, чем этот минимум, выполняется присваивание. Вероятность, что второй элемент меньше, чем первый, равна 1/2; такова же и вероятность присваивания минимуму нового значения. Вероятность того, что третий элемент окажется меньше, чем первые два, равна 1/3, а вероятность того, что четвертый окажется наименьшим, равна 1/4, и т. д. Поэтому полное среднее число пересылок равно H_{n-1}, где H_n – n-ое гармоническое число

$$H_n = 1 + 1/2 + 1/3 + ... + 1/n$$

H_n можно представить как

$$H_n = \ln(n) + g + 1/2n - 1/12n^2 + ...$$

где g = 0.577216... – константа Эйлера. Для достаточно больших n можно отбросить дробные члены и получить приближенное среднее число присваиваний на i-м проходе в виде

$$F_i = \ln(i) + g + 1$$

Тогда среднее число пересылок M_{avg} в сортировке выбором равно сумме величин F_i с i, пробегающим от 1 до n:

$$M_{avg} = n*(g+1) + (\mathbf{S}i: 1 \le i \le n: \ln(i))$$

Аппроксимируя дискретную сумму интегралом

$$\text{Integral } (1:n)\ \ln(x)\ dx = n * \ln(n) - n + 1$$

получаем приближенное выражение

$$M_{avg} = n * (\ln(n) + g)$$

Можно заключить, что в общем случае простая сортировка выбором предпочтительней простой сортировки вставками, хотя если ключи изначально упорядочены или почти упорядочены, простые вставки работают немного быстрее.

2.2.3. Простая сортировка обменами (пузырьковая)

Классификация методов сортировки не вполне однозначна. Оба обсуждавшихся выше метода можно также рассматривать как сортировки обменами. Однако сейчас мы представим метод, в котором обмен двух элементов играет главную роль. Получающаяся простая сортировка обменами использует сравнение и обмен пар соседних элементов до тех пор, пока не будут отсортированы все элементы.

Как и ранее в простой сортировке выбором, здесь выполняются повторные проходы по массиву, причем каждый раз наименьший элемент оставшегося множества просеивается в направлении левого конца массива. Если для разнообразия считать массив расположенным вертикально, а не горизонтально, и вообразить, что элементы – это пузырьки в сосуде с водой, причем их вес равен значению ключа, тогда проходы по массиву приводят к подъему пузырька на уровень, соответствующий его весу (см. табл. 2.3). Такой метод широко известен как *пузырьковая сортировка*.

Таблица 2.3. Пример работы пузырьковой сортировки

i = 0	1	2	3	4	5	6	7
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	**12**	55	44	42	42	42	42
94	42	**18**	55	44	44	44	44
18	94	42	**42**	55	55	55	55
06	**18**	94	67	67	67	67	67
67	67	**67**	94	94	94	94	94

```
PROCEDURE BubbleSort;                                    (* ADruS2_Sorts *)
    VAR i, j: INTEGER; x: Item;
BEGIN
    FOR i := 1 TO n-1 DO
        FOR j := n-1 TO i BY -1 DO
            IF a[j-1] > a[j] THEN
                x := a[j-1]; a[j-1] := a[j]; a[j] := x
            END
        END
    END
END BubbleSort
```

Этот алгоритм нетрудно улучшить. Пример в табл. 2.3 показывает, что последние три прохода не влияют на порядок элементов, так как они уже отсортированы. Очевидный способ улучшить алгоритм – запомнить, имел ли место хотя бы один обмен во время прохода. Тогда проход без обменов сигнализирует, что алго-

ритм может быть остановлен. Однако можно сделать еще одно улучшение, запоминая не только факт обмена, но и позицию (индекс) последнего обмена. Ясно, например, что все пары соседних элементов левее этого значения индекса (назовем его k) уже упорядочены. Поэтому последующие проходы могут останавливаться на этом значении индекса, вместо того чтобы продолжаться до i. Однако внимательный программист заметит любопытную асимметрию: если вначале только один пузырек стоит не на своем месте в «тяжелом» конце, то он доберется до своего места за один проход, а если такой пузырек стоит в «легком» конце, то он будет опускаться в свою правильную позицию только на один шаг за каждый проход. Например, массив

12 18 42 44 55 67 94 06

сортируется улучшенной пузырьковой сортировкой за один проход, а массив

94 06 12 18 42 44 55 67

требует семи проходов. Такая неестественная асимметрия наводит на мысль о третьем усовершенствовании: менять направление последовательных проходов. Получающийся алгоритм называют *шейкер-сортировкой* (shaker sort). Его работа показана в табл. 2.4, где он применяется к той же последовательности из восьми ключей, что и в табл. 2.3.

```
PROCEDURE ShakerSort;                                    (* ADruS2_Sorts *)
   VAR j, k, L, R: INTEGER;  x: Item;
BEGIN
   L := 1; R := n-1; k := R;
   REPEAT
      FOR j := R TO L BY -1 DO
         IF a[j-1] > a[j] THEN
            x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
         END
      END;
      L := k+1;
      FOR j := L TO R BY +1 DO
         IF a[j-1] > a[j] THEN
            x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
         END
      END;
      R := k-1
   UNTIL L > R
END ShakerSort
```

Анализ пузырьковой и шейкер-сортировки. Число сравнений в простой сортировке обменами равно

$$C = (n^2 - n)/2,$$

а минимальное, среднее и максимальное числа пересылок (присваиваний элементов) равны

Таблица 2.4. Пример работы шейкер-сортировки

L =	2	3	3	4	4
R =	8	8	7	7	4
dir =	↑	↓	↑	↓	↑
	44	06	06	06	06
	55	44	44	12	12
	12	**55**	**12**	**44**	18
	42	12	42	18	42
	94	42	55	42	44
	18	**94**	**18**	55	55
	06	18	67	67	67
	67	67	94	94	94

$M_{min} = 0, \quad M_{avg} = 3*(n^2 - n)/2, \quad M_{max} = 3*(n^2 - n)/4.$

Анализ улучшенных вариантов, особенно шейкер-сортировки, довольно сложен. Наименьшее число сравнений здесь равно $C_{min} = n-1$. Для улучшенной пузырьковой сортировки Кнут (см. [2.7] – *прим. перев.*) нашел, что среднее число проходов пропорционально величине $n - k_1*n^{1/2}$, а среднее число сравнений – величине $(n^2 - n*(k_2 + \ln(n)))/2$. Но нужно отметить, что ни одно из упомянутых улучшений не может повлиять на число обменов; уменьшается только число избыточных проверок. К несчастью, обмен двух элементов – обычно более затратная операция, чем сравнение ключей; поэтому все наши хитроумные улучшения имеют гораздо меньший эффект, чем интуитивно ожидается.

Этот анализ показывает, что сортировка обменами и ее небольшие улучшения хуже, чем и сортировка вставками и сортировка выбором; на самом деле пузырьковая сортировка вряд ли чем-нибудь интересна, кроме своего запоминающегося названия. Шейкер-сортировка выгодна в тех случаях, когда элементы уже стоят в почти правильном порядке, – но это редко случается на практике.

Можно показать, что среднее расстояние, которое должен пройти каждый из n элементов за время сортировки, равно $n/3$ позиций. Это число указывает, в каком направлении нужно искать более эффективные методы сортировки. В сущности, все простые методы перемещают элемент на одну позицию на каждом элементарном шаге. Поэтому они всегда требуют порядка n^2 таких шагов. Любое серьезное усовершенствование должно иметь целью увеличение расстояния, на которое перемещаются элементы в каждом прыжке.

В дальнейшем будут обсуждаться три эффективных метода, по одному на каждый из простых методов сортировки: вставками, выбором и обменами.

2.3. Эффективные методы сортировки

2.3.1. Сортировка вставками с уменьшающимися расстояниями

В 1959 г. Шелл [2.9] предложил способ ускорить сортировку простыми вставками. Проиллюстрируем метод Шелла с помощью нашего стандартного примера с восемью элементами (см. табл. 2.5). Сначала каждая группа элементов, отстоящих друг от друга на четыре позиции, сортируется отдельно. Этот процесс называется 4-сортировкой. В нашем примере с восемью элементами каждая такая группа состоит в точности из двух элементов. После первого прохода снова рассматриваются группы элементов, но уже отстоящих на две позиции, и снова группы сортируются по отдельности. Этот процесс называется 2-сортировкой. Наконец, на третьем проходе все элементы сортируются обычной сортировкой, или 1-сортировкой.

Таблица 2.5. Сортировка вставками с уменьшающимися расстояниями

	44	55	12	42	94	18	06	67
После 4-сортировки	44	18	06	42	94	55	12	67
После 2-сортировки	06	18	12	42	44	55	94	67
После 1-сортировки	06	12	18	42	44	55	67	94

Встает резонный вопрос: не превысят ли возможную экономию затраты на выполнение нескольких проходов, в каждом из которых сортируются все элементы? Однако каждая сортировка одной группы элементов либо имеет дело с относительно небольшим их числом, либо элементы уже частично отсортированы, так что нужно сделать сравнительно немного перестановок.

Очевидно, что метод приводит к отсортированному массиву, и нетрудно понять, что на каждом шаге нужно выполнить меньше работы благодаря предыдущим (так как каждая i-сортировка комбинирует две группы, отсортированные предыдущей 2i-сортировкой). Очевидно также, что допустима любая последовательность расстояний, лишь бы последнее из них было равно единице, так как в худшем случае вся работа будет выполняться на последнем проходе. Гораздо менее очевидно, что метод работает еще лучше, если уменьшающиеся расстояния выбирать отличными от степеней двойки.

Поэтому построим процедуру для произвольной последовательности расстояний. Всего будет T расстояний $h_0, h_1, \ldots, h_{T-1}$, удовлетворяющих условиям

$$h_{T-1} = 1, \quad h_{i+1} < h_i$$

Алгоритм описан в процедуре ShellSort [2.9] для T = 4:

```
PROCEDURE ShellSort;                                    (* ADruS2_Sorts *)
    CONST T = 4;
    VAR i, j, k, m, s: INTEGER;
        x: Item;
        h: ARRAY T OF INTEGER;
BEGIN
    h[0] := 9; h[1] := 5; h[2] := 3; h[3] := 1;
    FOR m := 0 TO T-1 DO
        k := h[m];
        FOR i := k TO n-1 DO
            x := a[i]; j := i-k;
            WHILE (j >= k) & (x < a[j]) DO
                a[j+k] := a[j]; j := j-k
            END;
            IF (j >= k) OR (x >= a[j]) THEN
                a[j+k] := x
            ELSE
                a[j+k] := a[j];
                a[j] := x
            END
        END
    END
END ShellSort
```

Анализ сортировки Шелла. Анализ этого алгоритма – очень сложная математическая проблема, полностью еще не решенная. В частности, неизвестно, какая последовательность расстояний дает наилучший результат. Но удивительно то, что расстояния не должны быть кратными друг другу. Тогда исчезнет явление, наблюдаемое в приведенном примере, когда каждый проход сортировки объединяет две цепочки, до того никак не «общавшиеся». На самом деле желательно, чтобы различные цепочки «общались» как можно чаще, причем выполняется следующая теорема: если последовательность после k-сортировки подвергается i-сортировке, то она остается k-отсортированной. Кнут в [2.7] (см. с. 105–115 перевода) приводит аргументы в пользу того, что неплохим выбором расстояний является последовательность (записанная здесь в обратном порядке)

1, 4, 13, 40, 121, ...

где $h_{k-1} = 3h_k+1$, $h_T = 1$, и $T = k \times \lfloor \log_3(n) \rfloor - 1$. Он также рекомендует последовательность

1, 3, 7, 15, 31, ...

где $h_{k-1} = 2h_k+1$, $h_T = 1$, и $T = k \times \lfloor \log_2(n) \rfloor - 1$. В последнем случае математическое исследование показывает, что при сортировке n элементов алгоритмом Шелла затраты пропорциональны $n^{1.2}$. Хотя это гораздо лучше, чем n^2, мы не будем углубляться в этот метод, так как есть алгоритмы еще лучше.

2.3.2. Турнирная сортировка

Простая сортировка выбором основана на повторном выборе наименьшего ключа среди n элементов, затем среди оставшихся n–1 элементов и т. д. Очевидно, для нахождения наименьшего ключа среди n элементов нужно n–1 сравнений, среди n–1 элементов – n–2 сравнений и т. д., а сумма первых n–1 целых равна $(n^2-n)/2$. Как же можно улучшить такую сортировку? Только если после каждого просмотра сохранять больше информации, чем всего лишь информацию об одном наименьшем элементе. Например, n/2 сравнений позволяют определить меньший ключ в каждой паре элементов, еще n/4 сравнений дадут меньший ключ в каждой паре из уже найденных меньших ключей, и т. д. Имея только n–1 сравнений, мы можем построить дерево выбора, показанное на рис. 2.3, причем в корне будет искомый наименьший ключ [2.2].

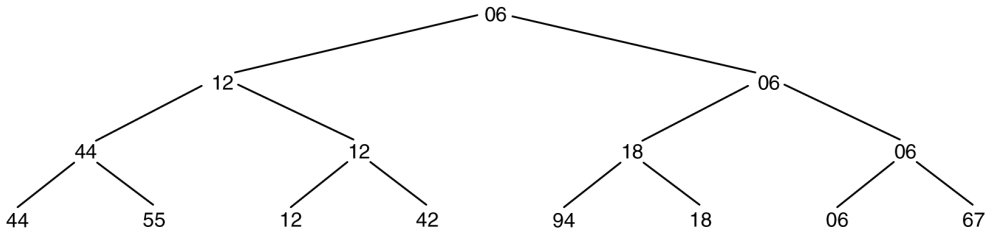


Рис. 2.3. Повторные выборы среди двух ключей («турнир ключей»)

Второй шаг состоит в спуске по пути, соответствующему наименьшему ключу, и в его удалении последовательной заменой либо на пустую позицию («дырку») в самом низу, либо на элемент из другой ветки в промежуточных узлах (см. рис. 2.4 и 2.5). Теперь элементом в корне дерева будет второй наименьший ключ, и его можно удалить. После n таких шагов выбора дерево станет пустым (то есть будет заполнено дырками), и процесс сортировки прекращается. Следует заметить, что на каждом из n шагов выбора требуется только log(n) сравнений. Поэтому вся процедура требует лишь порядка n*log(n) элементарных операций в дополнение к тем n шагам, которые нужны для построения дерева. Это очень существенное

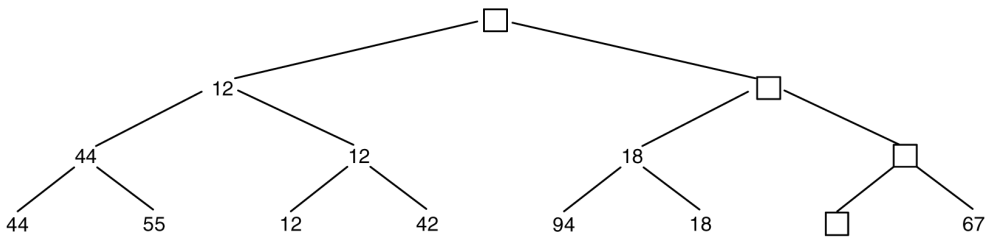


Рис. 2.4. Выбор наименьшего ключа

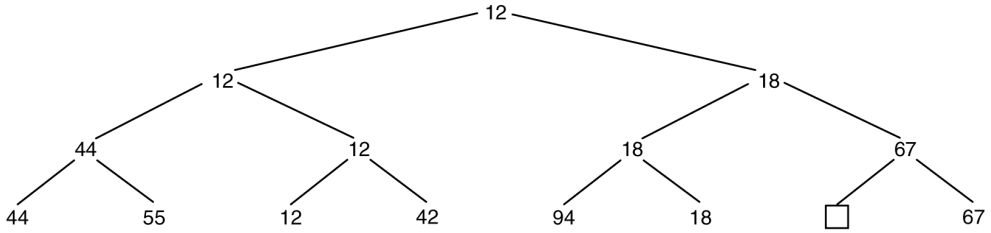


Рис. 2.5. Заполнение дырок

улучшение по сравнению с простыми методами, требующими n^2 шагов, и даже по сравнению с сортировкой Шелла, требующей $n^{1.2}$ шагов. Естественно, организовать работу такого алгоритма труднее, поэтому сложность отдельных шагов в турнирной сортировке выше: чтобы сохранить всю информацию с предыдущих шагов, нужно работать с некоторой древесной структурой. Нашей следующей задачей будет найти способ эффективно организовать всю эту информацию.

Прежде всего важно избавиться от необходимости иметь дело с дырками, которые в итоге заполнят все дерево и потребуют много лишних сравнений. Затем нужно найти способ представить дерево из n элементов в n единицах памяти вместо $2n - 1$, как это имеет место на приведенных рисунках. Обе цели были достигнуты в алгоритме Heapsort, названном так его изобретателем Вильямсом [2.12]; этот алгоритм представляет собой радикальное улучшение по сравнению с более традиционными подходами к турнирной сортировке. *Пирамида* (heap) определяется как последовательность ключей $h_L, h_{L+1}, \ldots, h_R$ ($L \geq 0$), такая, что

$$h_i < h_{2i+1} \quad \text{и} \quad h_i < h_{2i+2} \quad \text{для} \quad i = L \ldots R/2-1.$$

Если представить двоичное дерево массивом так, как показано на рис. 2.6, то деревья сортировки на рис. 2.7 и 2.8 тоже суть пирамиды, в частности элемент h_0 пирамиды является ее наименьшим элементом:

$$h_0 = \min(h_0, h_1, \ldots, h_{n-1})$$

Пусть дана пирамида с элементами $h_{L+1} \ldots h_R$ с некоторыми L и R, и пусть нужно добавить новый элемент x так, чтобы получилась расширенная пирамида $h_L \ldots h_R$. Например, возьмем начальную пирамиду $h_1 \ldots h_7$, показанную на

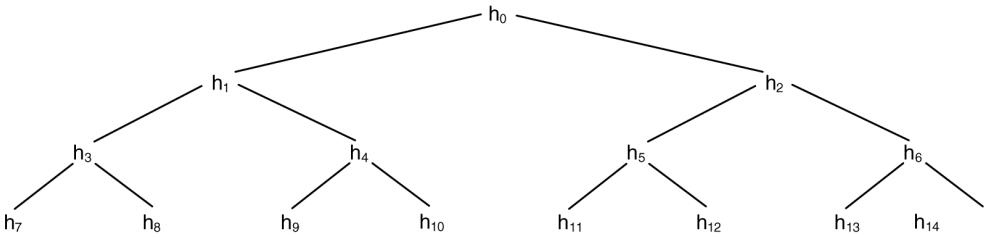


Рис. 2.6. Массив, представленный в виде двоичного дерева

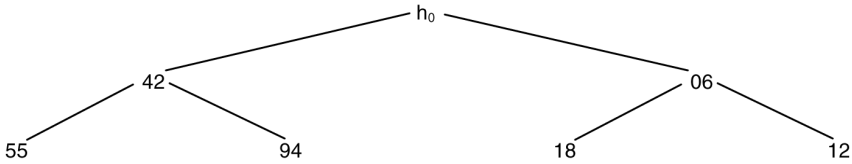


Рис. 2.7. Пирамида с семью элементами

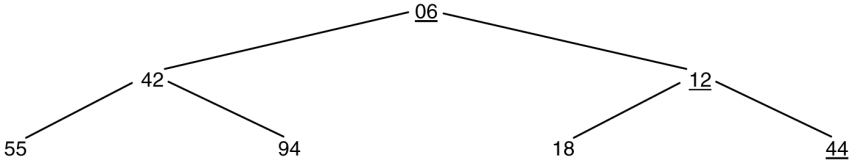


Рис. 2.8. Просеивание ключа 44 сквозь пирамиду

рис. 2.7, и расширим пирамиду влево элементом $h_0 = 44$. Новая пирамида получается так: сначала x ставится на вершину древесной структуры, а потом просеивается вниз, меняясь местами с наименьшим из двух потомков, который соответственно передвигается вверх. В нашем примере значение 44 сначала меняется местами с 06, затем с 12, так что получается дерево на рис. 2.8. Мы сформулируем алгоритм просеивания в терминах пары индексов i, j, соответствующих элементам, которые обмениваются на каждом шаге просеивания. А читателю предлагается убедиться в том, что метод действительно сохраняет инварианты, определяющие пирамиду.

Элегантный метод построения пирамиды *in situ* был предложен Флойдом [2.2]. Метод использует процедуру просеивания sift, представленную ниже. Пусть дан массив $h_0 \ldots h_{n-1}$; ясно, что элементы $h_m \ldots h_{n-1}$ (с m = n DIV 2) уже образуют пирамиду, так как среди них нет таких пар i, j, чтобы j = 2i+1 или j = 2i+2. Эти элементы образуют нижний ряд соответствующего двоичного дерева (см. рис. 2.6), где никакой упорядоченности не требуется. Затем пирамида расширяется влево, причем за один шаг в нее включается один новый элемент, который ставится на свое правильное место просеиванием. Этот процесс показан в табл. 2.6.

```
PROCEDURE sift (L, R: INTEGER);  (*просеивание*)
    VAR i, j: INTEGER;  x: Item;
BEGIN
    i := L; j := 2*i+1; x := a[i];
    IF (j < R) & (a[j+1] < a[j]) THEN j := j+1 END;
    WHILE (j <= R) & (a[j] < x) DO
        a[i] := a[j]; i := j; j := 2*j + 1;
        IF (j < R) & (a[j+1] < a[j]) THEN j := j+1 END
    END;
    a[i] := x
END sift
```

Таблица 2.6. Построение пирамиды

44	55	12	42 |	94	18	06	67
44	55	12 |	42	94	18	06	67
44	55 |	06	42	94	18	12	67
44 |	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

Следовательно, процесс порождения пирамиды из n элементов $h_0 \ldots h_{n-1}$ *in situ* описывается следующим образом:

```
L := n DIV 2;
WHILE L > 0 DO DEC(L); sift(L, n-1) END
```

Чтобы добиться не только частичного, но и полного упорядочения элементов, нужно выполнить n просеиваний, и после каждого из них с вершины пирамиды можно снять очередной (наименьший) элемент. Возникает вопрос: где хранить снимаемые с вершины элементы, и можно ли будет выполнить сортировку *in situ*. Решение существует: на каждом шаге нужно взять последний элемент пирамиды (скажем, x), записать элемент с вершины пирамиды в позицию, освободившуюся из-под x, а затем поставить x в правильную позицию просеиванием. Необходимые n-1 шагов иллюстрируются табл. 2.7. Этот процесс можно описать с помощью процедуры sift следующим образом:

```
R := n-1;
WHILE R > 0 DO
    x := a[0]; a[0] := a[R]; a[R] := x;
    DEC(R); sift(1, R)
END
```

Таблица 2.7. Пример работы сортировки HeapSort

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44 |	06
18	42	44	55	94	67 |	12	06
42	55	44	67	94 |	18	12	06
44	55	94	67 |	42	18	12	06
55	67	94 |	44	42	18	12	06
67	94 |	55	44	42	18	12	06
94 |	67	55	44	42	18	12	06

Из примера в табл. 2.7 видно, что на самом деле здесь получается обратный порядок элементов. Но это легко исправить заменой операций сравнения в процедуре sift на противоположные. Таким образом получаем следующую процедуру HeapSort. (Заметим, что в логическом смысле процедура sift является внутренней для HeapSort.)

```
PROCEDURE sift (L, R: INTEGER);                          (* ADruS2_Sorts *)
    VAR i, j: INTEGER;  x: Item;
BEGIN
    i := L; j := 2*i+1; x := a[i];
    IF (j < R) & (a[j] < a[j+1]) THEN j := j+1 END;
    WHILE (j <= R) & (x < a[j]) DO
        a[i] := a[j]; i := j; j := 2*j+1;
        IF (j < R) & (a[j] < a[j+1]) THEN j := j+1 END
    END;
    a[i] := x
END sift;

PROCEDURE HeapSort;
    VAR L, R: INTEGER;  x: Item;
BEGIN
    L := n DIV 2; R := n-1;
    WHILE L > 0 DO
        DEC(L); sift(L, R)
    END;
    WHILE R > 0 DO
        x := a[0]; a[0] := a[R]; a[R] := x;
        DEC(R); sift(L, R)
    END
END HeapSort
```

Анализ сортировки Heapsort. На первый взгляд не очевидно, что этот способ сортировки продемонстрирует хорошую эффективность. Ведь большие элементы сначала просеиваются влево, перед тем как попасть наконец в свои позиции в правом конце массива. Этот метод действительно нельзя рекомендовать для сортировки небольшого числа элементов (как в нашем примере). Однако для больших n сортировка Heapsort очень эффективна, и чем больше n, тем лучше она становится даже в сравнении с сортировкой Шелла.

В худшем случае фаза создания пирамиды требует n/2 шагов просеивания, причем на каждом шаге элементы просеиваются через log(n/2), log(n/2+1), ... , log(n–1) позиций, где логарифм (по основанию 2) округляется вниз до ближайшего целого. Затем фаза сортировки требует n–1 просеиваний, с не более чем log(n–1), log(n–2), ..., 1 пересылок. Кроме того, нужно n–1 пересылок для «складирования» элементов с вершины пирамиды в правом конце массива. Эти рассуждения показывают, что Heapsort требует порядка $n \times \log(n)$ пересылок даже в наихудшем случае. Такое отличное поведение в наихудшем случае является одним из важнейших достоинств алгоритма Heapsort.

Отнюдь не ясно, в каких случаях следует ожидать наихудшей (или наилучшей) производительности. Похоже, что обычно Heapsort «любит» начальные последовательности, в которых элементы более или менее отсортированы в обратном порядке, и в этом смысле алгоритм ведет себя неестественно. Фаза создания пирамиды не требует пересылок, если элементы изначально стоят в обратном порядке. Среднее число пересылок примерно равно $n/2 \times \log(n)$, а отклонения от этого значения сравнительно малы.

2.3.3. Быстрая сортировка

Обсудив два эффективных метода, основанных на принципах вставки и выбора, введем теперь третий, основанный на принципе обмена. Так как пузырьковая сортировка оказалась в среднем наихудшей среди трех простых алгоритмов, здесь можно ожидать сравнительно заметного улучшения. Однако удивительно, что усовершенствование обменной сортировки, которое мы собираемся обсудить, дает лучший из известных методов сортировки массивов. Производительность здесь настолько высока, что автор этого алгоритма Хоор назвал его *быстрой сортировкой* (Quicksort) [2.5], [2.6].

Построение быстрой сортировки исходит из того, что для достижения максимальной эффективности желательно выполнять обмены между максимально удаленными позициями. Пусть даны n элементов, расставленных в обратном порядке. Можно отсортировать их всего лишь за n/2 обменов, сначала взяв крайние левый и правый элементы и постепенно продвигаясь внутрь массива с обеих сторон. Естественно, такая процедура сработает, только если заранее известно, что элементы стоят в обратном порядке. Тем не менее из этого примера можно извлечь урок.

Попробуем реализовать такой алгоритм: случайно выберем любой элемент (назовем его x); будем просматривать массив слева, пока не найдем элемент $a_i > x$, а затем справа, пока не найдем элемент $a_j < x$. Затем выполним обмен двух найденных элементов и будем продолжать такой процесс просмотров и обмена, пока оба просмотра не встретятся где-то в середине массива. В результате получим массив, разделенный на левую часть с ключами, меньшими (или равными) x, и правую часть с ключами, большими (или равными) x. Теперь сформулируем этот процесс разделения (partitioning) в виде процедуры. Заметим, что отношения > и < заменены на ≥ и ≤, которые при отрицании в охране цикла WHILE превращаются в < и >. После такой замены x играет роль барьера для обоих просмотров.

```
PROCEDURE partition; (* разделение *)
    VAR i, j: INTEGER; w, x: Item;
BEGIN
    i := 0; j := n-1;
    выбрать наугад значение x;
    REPEAT
        WHILE a[i] < x DO i := i+1 END;
        WHILE x < a[j] DO j := j-1 END;
        IF i <= j THEN
            w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
        END
    UNTIL i > j
END partition
```

Например, если в качестве x выбран средний ключ 42, то для массива

```
44  55  12  42  94  06  18  67
```

потребуются два обмена $18 \leftrightarrow 44$ и $6 \leftrightarrow 55$, и разделенный массив будем иметь вид

```
18  06  12  42  94  55  44  67,
```

а последними значениями индексов будут i = 4 и j = 2. (Далее описывается инвариант цикла, то есть совокупность условий, выполняющихся в начале и в конце каждого шага цикла – *прим. перев.*) Ключи a_0 ... a_{i-1} меньше или равны ключу x = 42, а ключи a_{j+1} ... a_{n-1} больше или равны x. Следовательно, получились три части:

Ak: 1 ≤ k < i : $a_k \le x$
Ak: i ≤ k ≤ j : a_k ? x
Ak: j < k ≤ n–1 : $x \le a_k$

Смысл действий здесь в том, чтобы увеличивать i и уменьшать j, пока не исчезнет средняя часть. Этот алгоритм очень прост и эффективен, так как важные переменные i, j и x могут храниться в быстрых регистрах на протяжении просмотра. Однако он может повести себя плохо, например в случае n одинаковых ключей, когда будет сделано n/2 обменов. Такие избыточные обмены легко устранить, изменив операторы просмотра следующим образом:

```
WHILE a[i] <= x DO i := i+1 END;
WHILE x <= a[j] DO j := j-1 END
```

Но тогда выбранное значение x, которое присутствует в массиве как один из элементов, не сможет больше играть роль барьера для двух просмотров. Тогда в случае массива, в котором все ключи равны, просмотры выйдут за границы массива, если не использовать более сложные условия остановки. Но простота условий стоит того, чтобы заплатить за нее избыточными обменами, которые имеют место сравнительно редко в типичной случайной конфигурации. Небольшая экономия возможна, если заменить охрану обмена i ≤ j на i < j. Но это изменение не должно затрагивать двух операторов

```
i := i+1; j := j-1
```

для которых тогда потребовался бы отдельный условный оператор. Чтобы убедиться в правильности алгоритма разделения, можно проверить, что отношения порядка являются инвариантами оператора REPEAT. Вначале при i = 0 и j = n–1 они удовлетворяются тривиально, а после выхода по условию i > j из них следует искомый результат.

Теперь вспомним, что наша цель – не просто разделить исходный массив, но еще и отсортировать его. Однако от разделения до сортировки лишь один небольшой шаг: после разделения массива нужно применить тот же процесс к обеим получившимся частям, затем к частям тех частей и т. д., пока каждая часть не будет состоять только из одного элемента. Этот рецепт можно описать следующим образом (отметим, что в логическом смысле процедура sort является внутренней для QuickSort):

```
PROCEDURE sort (L, R: INTEGER);                    (* ADruS2_Sorts *)
    VAR i, j: INTEGER; w, x: Item;
BEGIN
    i := L; j := R;
    x := a[(L+R) DIV 2];
```

```
    REPEAT
        WHILE a[i] < x DO i := i+1 END;
        WHILE x < a[j] DO j := j-1 END;
        IF i <= j THEN
            w := a[i]; a[i] := a[j]; a[j] := w;
            i := i+1; j := j-1
        END
    UNTIL i > j;
    IF L < j THEN sort(L, j) END;
    IF i < R THEN sort(i, R) END
END sort;

PROCEDURE QuickSort;  (* быстрая сортировка *)
BEGIN
    sort(0, n-1)
END QuickSort
```

Процедура sort рекурсивно вызывает сама себя. Такое использование рекурсии в алгоритмах является очень мощным средством и будет подробно обсуждаться в главе 3. В некоторых старинных языках программирования рекурсия запрещена по техническим причинам. Поэтому покажем, как тот же самый алгоритм можно выразить в нерекурсивной форме. Очевидно, решение заключается в том, чтобы выразить рекурсию через итерацию, для чего потребуются дополнительные организационные усилия.

Ключ к итерационному решению – в том, чтобы организовать некий список запросов на разделение частей массива, которые еще только предстоит выполнить. После каждого шага возникают две задачи дальнейшего разделения. Только одну из них можно выполнять непосредственно в следующей итерации; другая сохраняется в упомянутом списке. Конечно, важно, чтобы список запросов обрабатывался в определенном порядке, а именно в обратном порядке. Это означает, что первый сохраненный запрос должен быть выполнен последним, и наоборот, то есть список обрабатывается по принципу стека. В следующей нерекурсивной версии алгоритма быстрой сортировки каждый запрос представлен просто значениями левой и правой границ сегмента, который нужно разделить. Поэтому вводятся два массива low, high, используемые как стеки с индексом s (указатель вершины стека). Выбор размера стека M будет обсуждаться при анализе алгоритма быстрой сортировки.

```
PROCEDURE NonRecursiveQuickSort;                    (* ADruS2_Sorts *)
    CONST  M = 12;
    VAR  i, j, L, R, s: INTEGER;  x, w: Item;
        low, high: ARRAY M OF INTEGER;  (*стек индексов*)
BEGIN
    s := 0; low[0] := 0; high[0] := n-1;
    REPEAT (*взять верхний запрос со стека*)
        L := low[s];  R := high[s];  DEC(s);
        REPEAT (*разделить сегмент a[L] ... a[R]*)
            i := L; j := R; x := a[(L+R) DIV 2];
```

```
            REPEAT
                WHILE a[i] < x DO i := i+1 END;
                WHILE x < a[j] DO j := j-1 END;
                IF i <= j THEN
                    w := a[i]; a[i] := a[j]; a[j] := w;
                    i := i+1; j := j-1
                END
            UNTIL i > j;
            IF i < R THEN (*сохранить в стеке запрос на сортировку правой части*)
                INC(s); low[s] := i; high[s] := R
            END;
            R := j  (*теперь L и R ограничивают левую часть*)
        UNTIL L >= R
    UNTIL s < 0
END NonRecursiveQuickSort
```

Анализ быстрой сортировки. Чтобы изучить эффективность быстрой сортировки, нужно сначала исследовать поведение процесса разделения. После выбора разделяющего значения x просматривается весь массив. Поэтому выполняется в точности n сравнений. Число обменов может быть определено с помощью следующего вероятностного рассуждения.

Если положение разделяющего значения фиксировано и соответствующее значение индекса равно u, то среднее число операций обмена равно числу элементов в левой части сегмента, а именно u, умноженному на вероятность того, что элемент попал на свое место посредством обмена. Обмен произошел, если элемент принадлежал правой части; вероятность этого равна (n–u)/n. Поэтому среднее число обменов равно среднему этих значений по всем возможным значениям u:

$$M = [\mathbf{S}u: 0 \le u \le n-1 : u*(n-u)]/n^2$$
$$= n*(n-1)/2n - (2n^2 - 3n + 1)/6n$$
$$= (n - 1/n)/6$$

Если нам сильно везет и в качестве границы всегда удается выбрать медиану, то каждый процесс разделения разбивает массив пополам, и число необходимых для сортировки проходов равно log(n). Тогда полное число сравнений равно n*log(n), а полное число обменов – n*log(n)/6.

Разумеется, нельзя ожидать, что с выбором медианы всегда будет так везти, ведь вероятность этого всего лишь 1/n. Но удивительно то, что средняя эффективность алгоритма Quicksort хуже оптимального случая только на множитель 2*ln(2), если разделяющее значение выбирается в массиве случайно.

Однако и у алгоритма Quicksort есть свои подводные камни. Прежде всего при малых n его производительность не более чем удовлетворительна, как и для всех эффективных методов. Но его преимущество над другими эффективными методами заключается в легкости подключения какого-нибудь простого метода для обработки коротких сегментов. Это особенно важно для рекурсивной версии алгоритма.

Однако еще остается проблема наихудшего случая. Как поведет себя Quicksort тогда? Увы, ответ неутешителен, и здесь выявляется главная слабость этого алго-

ритма. Например, рассмотрим неудачный случай, когда каждый раз в качестве разделяющего значения x выбирается наибольшее значение в разделяемом сегменте. Тогда каждый шаг разбивает сегмент из n элементов на левую часть из n–1 элементов и правую часть из единственного элемента. Как следствие нужно сделать n разделений вместо log(n), и поведение в худшем случае оказывается порядка n^2.

Очевидно, что ключевым шагом здесь является выбор разделяющего значения x. В приведенном варианте алгоритма на эту роль выбирается средний элемент. Но с равным успехом можно выбрать первый или последний элемент. В этих случаях наихудший вариант поведения будет иметь место для изначально упорядоченного массива; то есть алгоритм Quicksort явно «не любит» легкие задачки и предпочитает беспорядочные наборы значений. При выборе среднего элемента это странное свойство алгоритма Quicksort не так очевидно, так как изначально упорядоченный массив оказывается наилучшим случаем. На самом деле если выбирается средний элемент, то и производительность в среднем оказывается немного лучшей. Хоор предложил выбирать x случайным образом или брать медиану небольшой выборки из, скажем, трех ключей [2.10] и [2.11]. Такая предосторожность вряд ли ухудшит среднюю производительность алгоритма, но она сильно улучшает его поведение в наихудшем случае. Во всяком случае, ясно, что сортировка с помощью алгоритма Quicksort немного похожа на тотализатор, и пользователь должен четко понимать, какой проигрыш он может себе позволить, если удача от него отвернется.

Отсюда можно извлечь важный урок для программиста. Каковы последствия поведения алгоритма Quicksort в наихудшем случае, указанном выше? Мы уже знаем, что в такой ситуации каждое разделение дает правый сегмент, состоящий из единственного элемента, и запрос на сортировку этого сегмента сохраняется на стеке для выполнения в будущем. Следовательно, максимальное число таких запросов и, следовательно, необходимый размер стека равны n. Конечно, это совершенно неприемлемо. (Заметим, что дело обстоит еще хуже в рекурсивной версии, так как вычислительная система, допускающая рекурсивные вызовы процедур, должна автоматически сохранять значения локальных переменных и параметров всех активаций процедур, и для этого будет использоваться скрытый стек.) Выход здесь в том, чтобы сохранять на стеке запрос на обработку более длинной части, а к обработке короткой части приступать немедленно. Тогда размер стека M можно ограничить величиной log(n).

Соответствующее изменение локализовано в том месте программы, где на стеке сохраняются новые запросы на сортировку сегментов:

```
IF j - L < R - i THEN
    IF i < R THEN (*записать в стек запрос на сортировку правой части*)
        INC(s); low[s] := i; high[s] := R
    END;
    R := j  (*продолжать сортировкой левой части*)
ELSE
    IF L < j THEN (*записать в стек запрос на сортировку левой части*)
```

```
        INC(s); low[s] := L; high[s] := j
    END;
    L := i  (*продолжать сортировкой правой части*)
END
```

2.3.4. Поиск медианы

Медиана (median) n элементов – это элемент, который меньше (или равен) половины n элементов и который больше (или равен) элементов другой половины. Например, медиана чисел

16 12 99 95 18 87 10

равна 18. Задача нахождения медианы обычно связывается с задачей сортировки, так как очевидный метод определения медианы состоит в сортировке n элементов и выборе среднего элемента. Но использованная выше процедура разделения дает потенциальную возможность находить медиану гораздо быстрее. Метод, который мы сейчас продемонстрируем, легко обобщается на задачу нахождения k-го наименьшего из n элементов. Нахождение медианы соответствует частному случаю k = n/2.

Этот алгоритм был придуман Хоором [2.4] и работает следующим образом. Во-первых, операция разделения в алгоритме Quicksort выполняется с L = 0 и R = n–1, причем в качестве разделяющего значения x выбирается a_k. В результате получаются значения индексов i и j – такие, что

1. $a_h < x$ для всех h < i
2. $a_h > x$ для всех h > j
3. i > j

Здесь возможны три случая:

1. Разделяющее значение x оказалось слишком мало; в результате граница между двумя частями меньше нужного значения k. Тогда операцию разделения нужно повторить с элементами a_i ... a_R (см. рис. 2.9).

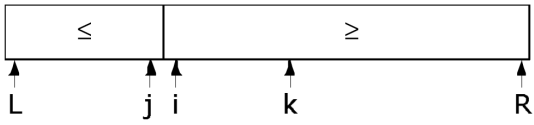


Рис. 2.9. Значение x слишком мало

2. Выбранное значение x оказалось слишком велико. Тогда операцию разделения нужно повторить с элементами a_L ... a_j (см. рис. 2.10).
3. j < k < i: элемент a_k разбивает массив на две части в нужной пропорции и поэтому является искомым значением (см. рис. 2.11).

Операцию разделения нужно повторять, пока не реализуется случай 3. Этот цикл выражается следующим программным фрагментом:

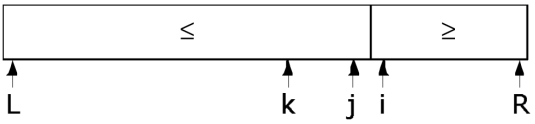


Рис. 2.10. Значение **x** слишком велико

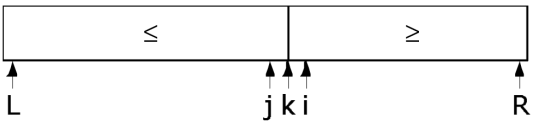


Рис. 2.11. Значение **x** оказалось правильным

```
L := 0; R := n;
WHILE L < R-1 DO
    x := a[k];
    произвести разделение сегмента (a[L] ... a[R-1]);
    IF j < k THEN L := i END;
    IF k < i THEN R := j END
END
```

За формальным доказательством корректности алгоритма отошлем читателя к оригинальной статье Хоора. Теперь нетрудно выписать процедуру Find целиком:

```
PROCEDURE Find (k: INTEGER);                                (* ADruS2_Sorts *)
    (*переставить a так, чтобы a[k] стал k-м по возрастанию*)
    VAR L, R, i, j: INTEGER; w, x: Item;
BEGIN
    L := 0; R := n-1;
    WHILE L < R-1 DO
        x := a[k]; i := L; j := R;
        REPEAT
            WHILE a[i] < x DO i := i+1 END;
            WHILE x < a[j] DO j := j-1 END;
            IF i <= j THEN
                w := a[i]; a[i] := a[j]; a[j] := w;
                i := i+1; j := j-1
            END
        UNTIL i > j;
        IF j < k THEN L := i END;
        IF k < i THEN R := j END
    END
END Find
```

Если предположить, что в среднем каждое разбиение делит пополам размер той части массива, в которой находится искомое значение, то необходимое число сравнений будет

$$n + n/2 + n/4 + \ldots + 1 \approx 2n$$

то есть величина порядка n. Это объясняет эффективность процедуры Find для нахождения медиан и других подобных величин, и этим объясняется ее превосходство над простым методом, состоящим в сортировке всего массива с последующим выбором k-го элемента (где наилучшее поведение имеет порядок n*log(n)). Однако в худшем случае каждый шаг разделения уменьшает размер множества кандидатов только на единицу, что приводит к числу сравнений порядка n^2. Как и ранее, вряд ли имеет смысл использовать этот алгоритм, когда число элементов мало, скажем меньше 10.

2.3.5. Сравнение методов сортировки массивов

Чтобы завершить парад методов сортировки, попробуем сравнить их эффективность. Пусть n обозначает число сортируемых элементов, а C и M – число сравнений ключей и пересылок элементов соответственно. Для всех трех простых методов сортировки имеются замкнутые аналитические формулы. Они даны в табл. 2.8. В колонках min, max, avg стоят соответствующие минимальные, максимальные значения, а также значения, усредненные по всем n! перестановкам n элементов.

Таблица 2.8. Сравнение простых методов сортировки

		min	avg	max
Простые вставки	C =	n-1	$(n^2 + n - 2)/4$	$(n^2 - n)/2 - 1$
	M =	2(n-1)	$(n^2 - 9n - 10)/4$	$(n^2 - 3n - 4)/2$
Простой выбор	C =	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	M =	3(n-1)	n*(ln(n) + 0.57)	$n^2/4 + 3(n-1)$
Простые обмены	C =	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	M =	0	$(n^2-n)*0.75$	$(n^2-n)*1.5$

Для эффективных методов простых точных формул не существует. Основные факты таковы: вычислительные затраты для Shellsort оцениваются величиной порядка $c*n^{1.2}$, а для сортировок Heapsort и Quicksort – величиной c*n*log(n), где c – некоторые коэффициенты.

Эти формулы дают только грубую оценку эффективности как функцию параметра n, и они позволяют классифицировать алгоритмы сортировки на примитивные, простые методы (n^2) и эффективные, или «логарифмические», методы (n*log(n)). Однако для практических целей полезно иметь эмпирические данные о величинах коэффициентов c, чтобы можно было сравнить разные методы. Более того, формулы приведенного типа не учитывают вычислительных

затрат на операции, отличные от сравнений ключей и пересылок элементов, такие как управление циклами и т. п. Понятно, что эти факторы в какой-то степени зависят от конкретной вычислительной системы, но тем не менее для ориентировки полезно иметь какие-нибудь эмпирические данные. Таблица 2.9 показывает время (в секундах), затраченное обсуждавшимися методами сортировки, при выполнении на персональном компьютере Лилит (Lilith). Три колонки содержат время, затраченное на сортировку уже упорядоченного массива, случайной перестановки и массива, упорядоченного в обратном порядке. Таблица 2.9 содержит данные для массива из 256 элементов, таблица 2.10 – для массива из 2048 элементов. Эти данные демонстрируют явное различие между квадратичными (n^2) и логарифмическими методами ($n*log(n)$). Кроме того, полезно отметить следующее:

1. Замена простых вставок (StraightInsertion) на двоичные (BinaryInsertion) дает лишь незначительное улучшение, а в случае уже упорядоченного массива приводит даже к ухудшению.
2. Пузырьковая сортировка (BubbleSort) – определенно наихудший из всех сравниваемых здесь методов. Даже его усовершенствованная версия, шейкер-сортировка (ShakerSort), все равно хуже, чем методы простых вставок (StraightInsertion) и простого выбора (StraightSelection), за исключением патологического случая сортировки уже упорядоченного массива.
3. Быстрая сортировка (QuickSort) лучше турнирной (HeapSort) на множитель от 2 до 3. Она сортирует обратно упорядоченный массив практически с такой же скоростью, как и просто упорядоченный.

Таблица 2.9. Время выполнения процедур сортировки для массивов из 256 элементов

	Упорядоченный	Случайный	Обратный
StraightInsertion	0.02	0.82	1.64
BinaryInsertion	0.12	0.70	1.30
StraightSelection	0.94	0.96	1.18
BubbleSort	1.26	2.04	2.80
ShakerSort	0.02	1.66	2.92
ShellSort	0.10	0.24	0.28
HeapSort	0.20	0.20	0.20
QuickSort	0.08	0.12	0.08
NonRecQuickSort	0.08	0.12	0.08
StraightMerge	0.18	0.18	0.18

Таблица 2.10. Время выполнения процедур сортировки для массивов из 2048 элементов

	Упорядоченный	Случайный	Обратный
StraightInsertion	0.22	50.74	103.80
BinaryInsertion	1.16	37.66	76.06
StraightSelection	58.18	58.34	73.46
BubbleSort	80.18	128.84	178.66
ShakerSort	0.16	104.44	187.36
ShellSort	0.80	7.08	12.34
HeapSort	2.32	2.22	2.12
QuickSort	0.72	1.22	0.76
NonRecQuickSort	0.72	1.32	0.80
StraightMerge	1.98	2.06	1.98

2.4. Сортировка последовательностей

2.4.1. Простые слияния

К сожалению, алгоритмы сортировки, представленные в предыдущей главе, неприменимы, когда объем сортируемых данных таков, что они не помещаются целиком в оперативную память компьютера и хранятся на внешних устройствах последовательного доступа, таких как ленты или диски. В этом случае будем считать, что данные представлены в виде (последовательного) файла, для которого характерно, что в каждый момент времени непосредственно доступен только один элемент. Это очень сильное ограничение по сравнению с теми возможностями, которые дают массивы, и здесь нужны другие методы сортировки.

Самый важный метод – сортировка слияниями (для всех вариантов сортировки слияниями Вирт употребляет родовое наименование Mergesort – *прим. перев.*). *Слиянием* (merging, collating) называют объединение двух (или более) упорядоченных последовательностей в одну, тоже упорядоченную последовательность повторным выбором из доступных в данный момент элементов. Слияние – гораздо более простая операция, чем сортировка, и эту операцию используют в качестве вспомогательной в более сложных процедурах сортировки последовательностей. Один из способов сортировки на основе слияний – *простая сортировка слияниями* (StraightMerge) – состоит в следующем:

1. Разобьем последовательность на две половины, b и c.
2. Выполним слияние частей b и c, комбинируя по одному элементу из b и c в упорядоченные пары.
3. Назовем получившуюся последовательность a, повторим шаги 1 и 2, на этот раз выполняя слияние упорядоченных пар в упорядоченные четверки.
4. Повторим предыдущие шаги, выполняя слияние четверок в восьмерки, и будем продолжать в том же духе, каждый раз удваивая длину сливаемых

подпоследовательностей, пока вся последовательность не окажется упорядоченной.

Например, рассмотрим следующую последовательность:

```
44  55  12  42  94  18  06  67
```

На шаге 1 разбиение последовательности дает две такие последовательности:

```
44  55  12  42
94  18  06  67
```

Слияние одиночных элементов (которые представляют собой упорядоченные последовательности длины 1) в упорядоченные пары дает:

```
44  94'  18  55'  06  12'  42  67
```

Снова разбивая посередине и выполняя слияние упорядоченных пар, получаем

```
06  12  44  94'  18  42  55  67
```

Наконец, третья операция разбиения и слияния дает желаемый результат:

```
06  12  18  42  44  55  67  94
```

Каждая операция, которая требует однократного прохода по всему набору данных, называется *фазой* (phase), а наименьшая процедура, из повторных вызовов которой состоит сортировка, называется *проходом* (pass). В приведенном примере сортировка состояла из трех проходов, каждый из которых состоял из фазы разбиения и фазы слияния. Чтобы выполнить сортировку, здесь нужны три ленты, поэтому процедура называется *трехленточным слиянием* (three-tape merge).

На самом деле фазы разбиения не дают вклада в сортировку в том смысле, что элементы там не переставляются; в этом отношении они непродуктивны, хотя и составляют половину всех операций копирования. Их можно вообще устранить, объединяя фазы разбиения и слияния. Вместо записи в единственную последовательность результат слияния сразу распределяется на две ленты, которые будут служить источником исходных данных для следующего прохода. В отличие от вышеописанной *двухфазной* (two-phase) сортировки слиянием, такой метод называется *однофазным* (single-phase), или методом *сбалансированных слияний* (balanced merge). Он явно более эффективен, так как нужно вдвое меньше операций копирования; плата за это – необходимость использовать четвертую ленту.

Мы детально разберем процедуру слияния, но сначала будем представлять данные с помощью массивов, только просматривать их будем строго последовательно. Затем мы заменим массивы на последовательности, что позволит сравнить две программы и показать сильную зависимость вида программы от используемого представления данных.

Вместо двух последовательностей можно использовать единственный массив, если считать, что оба его конца равноправны. Вместо того чтобы брать элементы для слияния из двух файлов, можно брать элементы с двух концов массива-источника. Тогда общий вид объединенной фазы слияния-разбиения можно проиллюстрировать рис. 2.12. После слияния элементы отправляются в массив-прием-

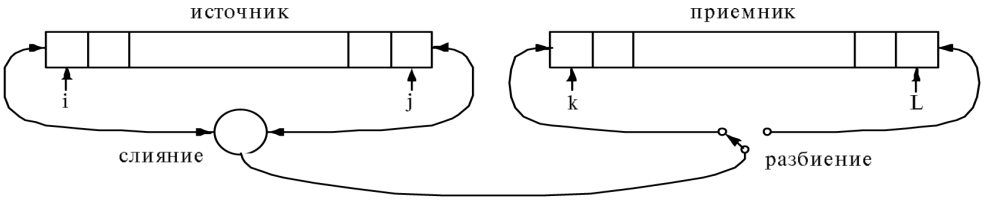


Рис. 2.12. Простая сортировка слияниями с двумя массивами

ник с одного или другого конца, причем переключение происходит после каждой упорядоченной пары, получающейся в результате слияния в первом проходе, после каждой упорядоченной четверки на втором проходе и т. д., так что будут равномерно заполняться обе последовательности, представленные двумя концами единственного массива-приемника. После каждого прохода два массива меняются ролями, источник становится приемником и наоборот.

Дальнейшее упрощение программы получается, если объединить два концептуально разных массива в единственный массив двойного размера. Тогда данные будут представлены так:

```
a: ARRAY 2*n OF элемент
```

Индексы i и j будут обозначать два элемента из массива-источника, а k и L – две позиции в массиве-приемнике (см. рис. 2.12). Исходные данные – это, конечно, элементы $a_0 \ldots a_{n-1}$. Очевидно, нужна булевская переменная up, чтобы управлять направлением потока данных; up будет означать, что в текущем проходе элементы $a_0 \ldots a_{n-1}$ пересылаются «вверх» в переменные $a_n \ldots a_{2n-1}$, тогда как ~up будет указывать, что $a_n \ldots a_{2n-1}$ пересылаются «вниз» в $a_0 \ldots a_{n-1}$. Значение up переключается перед каждым новым проходом. Наконец, для обозначения длины сливаемых подпоследовательностей вводится переменная p. Сначала ее значение равно 1, а затем оно удваивается перед каждым следующим проходом. Чтобы немного упростить дело, предположим, что n всегда является степенью 2. Тогда первый вариант простой сортировки слияниями приобретает следующий вид:

```
PROCEDURE StraightMerge;
    VAR i, j, k, L, p: INTEGER;  up: BOOLEAN;
BEGIN
    up := TRUE; p := 1;
    REPEAT
        инициировать индексные переменные;
        IF up THEN
            i := 0; j := n-1; k := n; L := 2*n-1
        ELSE
            k := 0; L := n-1; i := n; j := 2*n-1
        END;
        выполнить слияние p-наборов из i- и j-источников в k- и L-приемники;
        up := ~up; p := 2*p
    UNTIL p = n
END StraightMerge
```

На следующем шаге разработки мы должны уточнить инструкции, выделенные курсивом. Очевидно, что проход слияния, обрабатывающий n элементов, сам является серией слияний подпоследовательностей из p элементов (*p-наборов*). После каждого такого частичного слияния приемником для подпоследовательности становится попеременно то верхний, то нижний конец массива-приемника, чтобы обеспечить равномерное распределение в обе принимающие «последовательности». Если элементы после слияния направляются в нижний конец массива-приемника, то индексом-приемником является k, и k увеличивается после каждой пересылки элемента. Если они пересылаются в верхний конец массива-приемника, то индексом-приемником является L, и его значение уменьшается после каждой пересылки. Чтобы упростить получающийся программный код для слияния, k всегда будет обозначать индекс-приемник, значения переменых k иа L будут обмениваться после каждого слияния p-наборов, а переменная h, принимающая значения 1 или -1, будет всегда обозначать приращение для k. Эти проектные решения приводят к такому уточнению:

```
h := 1; m := n;  (*m = число сливаемых элементов*)
REPEAT
    q := p; r := p; m := m - 2*p;
    слить q элементов из i-источника с r элементами из j-источника;
    индекс-приемник равен k, увеличить k на h;
    h := -h;
    обменять значения k и L
UNTIL m = 0
```

На следующем шаге уточнения нужно конкретизировать операцию слияния. Здесь нужно помнить, что остаток той последовательности, которая осталась непустой после слияния, должен быть присоединен к выходной последовательности простым копированием.

```
WHILE (q > 0) & (r > 0) DO
    IF a[i] < a[j] THEN
        переслать элемент из i-источника в k-приемник;
        продвинуть i и k;  q := q-1
    ELSE
        переслать элемент из j-источника в k-приемник;
        продвинуть j и k;  r := r-1
    END
END;
скопировать остаток i-последовательности;
скопировать остаток j-последовательности
```

Уточнение операций копирования остатков даст практически полную программу. Прежде чем выписывать ее, избавимся от ограничения, что n является степенью двойки. На какие части алгоритма это повлияет? Нетрудно понять, что справиться с такой более общей ситуацией проще всего, если как можно дольше действовать старым способом. В данном примере это означает, что нужно продолжать сливать p-наборы до тех пор, пока остатки последовательностей-источников

не станут короче p. Это повлияет только на операторы, в которых устанавливаются значения длины сливаемых последовательностей q и r. Три оператора

```
q := p;  r := p;  m := m -2*p
```

нужно заменить на приведенные ниже четыре оператора, которые, как читатель может убедиться, в точности реализуют описанную стратегию; заметим, что m обозначает полное число элементов в двух последовательностях-источниках, которые еще предстоит слить:

```
IF m >= p THEN q := p ELSE q := m END;
m := m-q;
IF m >= p THEN r := p ELSE r := m END;
m := m-r
```

Кроме того, чтобы обеспечить завершение программы, нужно заменить условие p = n, которое управляет внешним циклом, на p ≥ n. После этих изменений весь алгоритм можно выразить в виде процедуры, работающей с глобальным массивом из 2n элементов:

```
PROCEDURE StraightMerge;                                (* ADruS24_MergeSorts *)
    VAR i, j, k, L, t: INTEGER;   (*диапазон индексов массива a is 0 .. 2*n-1 *)
        h, m, p, q, r: INTEGER; up: BOOLEAN;
BEGIN
    up := TRUE; p := 1;
    REPEAT
        h := 1; m := n;
        IF up THEN
            i := 0; j := n-1; k := n; L := 2*n-1
        ELSE
            k := 0; L := n-1; i := n; j := 2*n-1
        END;
        REPEAT (*слить по одной подпоследовательности
                  из i- и j-источников в k-приемник *)
            IF m >= p THEN q := p ELSE q := m END;
            m := m-q;
            IF m >= p THEN r := p ELSE r := m END;
            m := m-r;
            WHILE (q > 0) & (r > 0) DO
                IF a[i] < a[j] THEN
                    a[k] := a[i]; k := k+h; i := i+1; q := q-1
                ELSE
                    a[k] := a[j]; k := k+h; j := j-1; r := r-1
                END
            END;
            WHILE r > 0 DO
                a[k] := a[j]; k := k+h; j := j-1; r := r-1
            END;
            WHILE q > 0 DO
                a[k] := a[i]; k := k+h; i := i+1; q := q-1
            END;
```

```
            h := -h; t := k; k := L; L := t
        UNTIL m = 0;
        up := ~up; p := 2*p
    UNTIL p >= n;
    IF ~up THEN
        FOR i := 0 TO n-1 DO a[i] := a[i+n] END
    END
END StraightMerge
```

Анализ простой сортировки слияниями. Поскольку p удваивается на каждом проходе, а сортировка прекращается, как только $p > n$, то будет выполнено $\lceil \log(n) \rceil$ проходов. По определению, на каждом проходе все n элементов копируются в точности один раз. Следовательно, полное число пересылок в точности равно

$$M = n \times \lceil \log(n) \rceil$$

Число сравнений ключей C даже меньше, чем M, так как при копировании остатков никаких сравнений не требуется. Однако поскольку сортировка слияниями обычно применяется при работе с внешними устройствами хранения данных, вычислительные затраты на выполнение пересылок нередко превосходят затраты на сравнения на несколько порядков величины. Поэтому детальный анализ числа сравнений не имеет практического интереса.

Очевидно, сортировка StraightMerge выглядит неплохо даже в сравнении с эффективными методами сортировки, обсуждавшимися в предыдущей главе. Однако накладные расходы на манипуляции с индексами здесь довольно велики, а решающий недостаток – это необходимость иметь достаточно памяти для хранения 2n элементов. По этой причине сортировку слияниями редко применяют для массивов, то есть для данных, размещенных в оперативной памяти. Получить представление о реальном поведении алгоритма StraightMerge можно по числам в последней строке табл. 2.9. Видно, что StraightMerge ведет себя лучше, чем HeapSort, но хуже, чем QuickSort.

2.4.2. Естественные слияния

Если применяются простые слияния, то никакого выигрыша не получается в том случае, когда исходные данные частично упорядочены. Длина всех сливаемых подпоследовательностей на k-м проходе не превосходит 2k, даже если есть более длинные, уже упорядоченные подпоследовательности, готовые к слияниям. Ведь любые две упорядоченные подпоследовательности длины m и n можно сразу слить в одну последовательность из m+n элементов. Сортировка слияниями, в которой в любой момент времени сливаются максимально длинные последовательности, называется *сортировкой естественными слияниями*.

Упорядоченную подпоследовательность часто называют строкой (string). Но так как еще чаще это слово используют для последовательностей литер, мы вслед за Кнутом будем использовать термин *серия* (run) для обозначения упорядоченных подпоследовательностей. Подпоследовательность $a_i \ldots a_j$, такую, что

$$(a_{i-1} > a_i) \;\&\; (\mathbf{A}k: i \le k < j : a_k \le a_{k+1}) \;\&\; (a_j > a_{j+1})$$

будем называть *максимальной серией*, или, для краткости, просто серией. Итак, в сортировке естественными слияниями сливаются (максимальные) серии вместо последовательностей фиксированной предопределенной длины. Серии имеют то свойство, что если сливаются две последовательности по n серий каждая, то получается последовательность, состоящая в точности из n серий. Поэтому полное число серий уменьшается вдвое за каждый проход, и необходимое число пересылок элементов даже в худшем случае равно n*log(n), а в среднем еще меньше. Однако среднее число сравнений гораздо больше, так как, кроме сравнений при выборе элементов, нужны еще сравнения следующих друг за другом элементов каждого файла, чтобы определить конец каждой серии.

Наше очередное упражнение в программировании посвящено разработке алгоритма сортировки естественными слияниями в той же пошаговой манере, которая использовалась при объяснении простой сортировки слияниями. Вместо массива здесь используются последовательности (представленные файлами, см. раздел 1.7), а в итоге получится несбалансированная двухфазная трехленточная сортировка слияниями. Будем предполагать, что исходная последовательность элементов представлена файловой переменной c. (Естественно, в реальной ситуации исходные данные сначала из соображений безопасности копируются из некоторого источника в c.) При этом a и b – две вспомогательные файловые переменные. Каждый проход состоит из фазы распределения, когда серии из c равномерно распределяются в a и b, и фазы слияния, когда серии из a и b сливаются в c. Этот процесс показан на рис. 2.13.

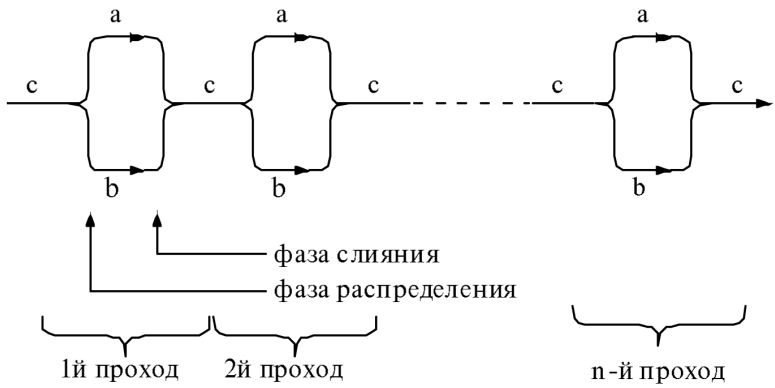


Рис. 2.13. Фазы сортировки и проходы

Пример в табл. 2.11 показывает файл c в исходном состоянии (строка 1) и после каждого прохода (строки 2–4) при сортировке этим методом двадцати чисел. Заметим, что понадобились только три прохода. Сортировка прекращается, как только в c остается одна серия. (Предполагается, что исходная последовательность содержит по крайней мере одну непустую серию.) Поэтому пусть переменная L подсчитывает число серий, записанных в c. Используя тип Rider («бегу-

нок»), определенный в разделе 1.7.1, можно сформулировать программу следующим образом:

```
VAR L: INTEGER;
    r0, r1, r2: Files.Rider;  (*см. раздел 1.7.1*)

REPEAT
    Files.Set(r0, a, 0); Files.Set(r1, b, 0); Files.Set(r2, c, 0);
    distribute(r2, r0, r1);  (*c распределяется в a и b*)

    Files.Set(r0, a, 0); Files.Set(r1, b, 0); Files.Set(r2, c, 0);
    L := 0;
    merge(r0, r1, r2)  (*a и b сливаются в c*)
UNTIL L = 1
```

Таблица 2.11. Пример сортировки естественными слияниями

17	31'	05	59'	13	41	43	67'	11	23	29	47'	03	07	71'	02	19	57'	37	61
05	17	31	59'	11	13	23	29	41	43	47	67'	02	03	07	19	57	71'	37	61
05	11	13	17	23	29	31	41	43	47	59	67'	02	03	07	19	37	57	61	71
02	03	05	07	11	13	17	19	23	29	31	37	41	43	47	57	59	61	67	71

Двум фазам в точности соответствуют два разных оператора. Их нужно теперь уточнить, то есть выразить с большей детализацией. Уточненные описания шагов distribute (распределить из бегунка r2 в бегунки r0 и r1) и merge (слить из бегунков r0 и r1 в r2) приводятся ниже:

```
REPEAT
    copyrun(r2, r0);
    IF ~r2.eof THEN copyrun(r2, r1) END
UNTIL r2.eof

REPEAT
    mergerun(r0, r1, r2); INC(L)
UNTIL r1.eof;
IF ~r0.eof THEN
    copyrun(r0, r2); INC(L)
END
```

По построению этот способ приводит либо к одинаковому числу серий в a и b, либо последовательность a будет содержать одну лишнюю серию по сравнению с файлом b. Поскольку сливаются соответствующие пары серий, эта лишняя серия может остаться только в файле a, и тогда ее нужно просто скопировать. Операции merge и distribute формулируются в терминах уточняемой ниже операции mergerun (слить серии) и вспомогательной процедуры copyrun (копировать серию), смысл которых очевиден. При попытке реализовать все это возникает серьезная трудность: чтобы определить конец серии, нужно сравнивать два последовательных ключа. Однако файлы устроены так, что каждый раз доступен только один элемент. Очевидно, здесь нужно «заглядывать вперед» на один элемент, по-

этому для каждой последовательности заводится буфер, который и должен содержать очередной элемент, стоящий в последовательности за текущим, и который представляет собой нечто вроде окошка, скользящего по файлу.

Уже можно было бы выписать все детали этого механизма в виде полной программы, но мы введем еще один уровень абстракции. Этот уровень представлен новым модулем Runs. Его можно рассматривать как расширение модуля Files из раздела 1.7, и в нем вводится новый тип Rider («бегунок»), который можно рассматривать как расширение типа Files.Rider. С этим новым типом не только можно будет выполнять все операции, предусмотренные для старого типа Rider, а также определять конец файла, но и узнавать о конце серии, а также «видеть» первый элемент в еще не прочитанной части файла. Этот новый тип вместе со своими операциями представлен в следующем определении:

```
DEFINITION Runs;                                        (* ADruS242_Runs *)
   IMPORT Files, Texts;
   TYPE Rider = RECORD (Files.Rider) first: INTEGER; eor: BOOLEAN END;

   PROCEDURE OpenRandomSeq (f: Files.File; length, seed: INTEGER);
   PROCEDURE Set (VAR r: Rider; VAR f: Files.File);
   PROCEDURE copy (VAR source, destination: Rider);
   PROCEDURE ListSeq (VAR W: Texts.Writer; f: Files.File);
END Runs.
```

Выбор процедур требует некоторых пояснений. Алгоритмы сортировки, обсуждаемые здесь и в дальнейшем, основаны на копировании элементов из одного файла в другой. Поэтому процедура copy замещает отдельные операции read и write.

Для удобства тестирования в последующих примерах мы дополнительно ввели процедуру ListSeq, которая печатает файл целых чисел в текст. Кроме того, для удобства введена еще одна процедура: OpenRandomSeq создает файл с числами в случайном порядке. Эти две процедуры будут служить для проверки обсуждаемых ниже алгоритмов. Значения полей eof и eor являются результатами операции copy аналогично тому, как ранее eof был результатом операции read.

```
MODULE Runs;                                            (* ADruS242_Runs *)
   IMPORT Files, Texts;
   TYPE Rider* = RECORD (Files.Rider) first: INTEGER; eor: BOOLEAN END;

   PROCEDURE OpenRandomSeq* (f: Files.File; length, seed: INTEGER);
      VAR i: INTEGER; w: Files.Rider;
   BEGIN
      Files.Set(w, f, 0);
      FOR i := 0 TO length-1 DO
         Files.WriteInt(w, seed); seed := (31*seed) MOD 997 + 5
      END;
      Files.Close(f)
   END OpenRandomSeq;

   PROCEDURE Set* (VAR r: Rider; f: Files.File);
   BEGIN
```

```
    Files.Set(r, f, 0); Files.ReadInt (r, r.first); r.eor := r.eof
END Set;

PROCEDURE copy* (VAR src, dest: Rider);
BEGIN
    dest.first := src.first;
    Files.WriteInt(dest, dest.first); Files.ReadInt(src, src.first);
    src.eor := src.eof OR (src.first < dest.first)
END copy;

PROCEDURE ListSeq* (VAR W: Texts.Writer; f: Files.File;);
    VAR x, y, k, n: INTEGER; r: Files.Rider;
BEGIN
    k := 0; n := 0;
    Files.Set(r, f, 0); Files.ReadInt(r, x);
    WHILE ~r.eof DO
        Texts.WriteInt(W, x, 6); INC(k); Files.ReadInt(r, y);
        IF y < x THEN (*конец серии*) Texts.Write(W, "|"); INC(n) END;
        x := y
    END;
    Texts.Write(W, "$"); Texts.WriteInt(W, k, 5); Texts.WriteInt(W, n, 5);
    Texts.WriteLn(W)
END ListSeq;

END Runs.
```

Вернемся теперь к процессу постепенного уточнения алгоритма сортировки естественными слияниями. Процедуры copyrun и merge уже можно выразить явно, как показано ниже. Отметим, что мы обращаемся к последовательностям (файлам) опосредованно, с помощью присоединенных к ним бегунков. Отметим кстати, что у бегунка поле first содержит *следующий* ключ в читаемой последовательности и *последний* ключ в записываемой последовательности.

```
PROCEDURE copyrun (VAR x, y: Runs.Rider);  (* копировать серию *)
BEGIN (*копировать из x в y*)
    REPEAT Runs.copy(x, y) UNTIL x.eor
END copyrun

(*merge: слить r0 и r1 в r2*)
REPEAT
    IF r0.first < r1.first THEN
        Runs.copy(r0, r2);
        IF r0.eor THEN copyrun(r1, r2) END
    ELSE Runs.copy(r1, r2);
        IF r1.eor THEN copyrun(r0, r2) END
    END
UNTIL r0.eor OR r1.eor
```

Процесс сравнения и выбора ключей при слиянии пары серий прекращается, как только одна из серий исчерпывается. После этого остаток серии (которая еще не исчерпана) должен быть просто скопирован в серию-результат. Это делается посредством вызова процедуры copyrun.

По идее, здесь процедура разработки должна завершиться. Увы, внимательный читатель заметит, что получившаяся программа не верна. Программа некорректна в том смысле, что в некоторых случаях она сортирует неправильно. Например, рассмотрим следующую последовательность входных данных:

```
03 02 05 11 07 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67
```

Распределяя последовательные серии попеременно в a и b, получим

```
a =  03 ' 07 13 19 ' 29 37 43 ' 57 61 71'
b =  02 05 11 ' 17 23 31 ' 41 47 59 ' 67
```

Эти последовательности легко сливаются в единственную серию, после чего сортировка успешно завершается. Хотя этот пример не приводит к ошибке, он показывает, что простое распределение серий в несколько файлов может приводить к меньшему числу серий на выходе, чем было серий на входе. Это происходит потому, что первый элемент серии номер i+2 может быть больше, чем последний элемент серии номер i, и тогда две серии автоматически «слипаются» в одну серию.

Хотя предполагается, что процедура distribute запитывает серии в два файла в равном числе, важное следствие состоит в том, что реальное число серий, записанных в a и b, может сильно различаться. Но наша процедура слияния сливает только пары серий и прекращает работу, как только прочитан файл b, так что остаток одной из последовательностей теряется. Рассмотрим следующие входные данные, которые сортируются (и обрываются) за два последовательных прохода:

Таблица 2.12. Неправильный результат алгоритма MergeSort

17	19	13	57	23	29	11	59	31	37	07	61	41	43	05	67	47	71	02	03
13	17	19	23	29	31	37	41	43	47	57	71	11	59						
11	13	17	19	23	29	31	37	41	43	47	57	59	71						

Такая ошибка достаточно типична в программировании. Она вызвана тем, что осталась незамеченной одна из ситуаций, которые могут возникнуть после выполнения простой, казалось бы, операции. Ошибка типична также в том отношении, что ее можно исправить несколькими способами и нужно выбрать один. Обычно есть две возможности, которые отличаются в одном принципиальном отношении:

1. Мы признаем, что операция распределения запрограммирована неправильно и не удовлетворяет требованию, чтобы число серий отличалось не больше, чем на единицу. При этом мы сохраняем первоначальную схему программы и исправляем неправильную процедуру.
2. Мы обнаруживаем, что исправление неправильной процедуры будет иметь далеко идущие последствия, и тогда пытаемся так изменить другие части программы, чтобы они правильно работали с данным вариантом процедуры.

В общем случае первый путь кажется более безопасным, ясным и честным, обеспечивая определенную устойчивость к последствиям незамеченных, тонких побочных эффектов. Поэтому обычно рекомендуется именно этот способ решения.

Однако не всегда следует игнорировать и второй путь. Именно поэтому мы хотим показать решение, основанное на изменении процедуры слияния, а не процедуры распределения, из-за которой, в сущности, и возникла проблема. Подразумевается, что мы не будем трогать схему распределения, но откажемся от условия, что серии должны распределяться равномерно. Это может понизить эффективность. Но поведение в худшем случае не изменится, а случай сильно неравномерного распределения статистически очень маловероятен. Поэтому соображения эффективности не являются серьезным аргументом против такого решения.

Если мы отказались от условия равномерного распределения серий, то процедура слияния должна измениться так, чтобы по достижении конца одного из файлов копировался весь остаток другого файла, а не только одна серия. Такое изменение оказывается очень простым по сравнению с любыми исправлениями в схеме распределения. (Читателю предлагается убедиться в справедливости этого утверждения.) Новая версия алгоритма слияний дана ниже в виде процедуры-функции:

```
PROCEDURE copyrun (VAR x, y: Runs.Rider);              (* ADruS24_MergeSorts *)
   (*копировать серию*)
BEGIN (*из x в y*)
   REPEAT Runs.copy(x, y) UNTIL x.eor
END copyrun;

PROCEDURE NaturalMerge (src: Files.File): Files.File;  (*естественные слияния*)
   VAR L: INTEGER;  (*число серий после слияния*)
      f0, f1, f2: Files.File;
      r0, r1, r2: Runs.Rider;
BEGIN
   Runs.Set(r2, src);
   REPEAT
      f0 := Files.New("test0"); Files.Set(r0, f0, 0);
      f1 := Files.New("test1"); Files.Set (r1, f1, 0);
      (*распределить из r2 в r0 и r1*)
      REPEAT
         copyrun(r2, r0);
         IF ~r2.eof THEN copyrun(r2, r1) END
      UNTIL r2.eof;
      Runs.Set(r0, f0); Runs.Set(r1, f1);
      f2 := Files.New(""); Files.Set(r2, f2, 0);

      (*merge: слить из r0 и r1 в r2*)
      L := 0;
      REPEAT
         REPEAT
            IF r0.first < r1.first THEN
               Runs.copy(r0, r2);
               IF r0.eor THEN copyrun(r1, r2) END
            ELSE
               Runs.copy(r1, r2);
```

```
                IF r1.eor THEN copyrun(r0, r2) END
              END
            UNTIL r0.eor & r1.eor;
            INC(L)
        UNTIL r0.eof OR r1.eof;
        WHILE ~r0.eof DO copyrun(r0, r2); INC(L) END;
        WHILE ~r1.eof DO copyrun(r1, r2); INC(L) END;
        Runs.Set(r2, f2)
    UNTIL L = 1;
    RETURN f2
END NaturalMerge;
```

2.4.3. Сбалансированные многопутевые слияния

Затраты на последовательную сортировку пропорциональны необходимому числу проходов, так как на каждом проходе по определению копируется весь набор данных. Один из способов уменьшить это число состоит в том, чтобы использовать больше двух файлов для распределения серий. Если сливать r серий, которые равномерно распределены по N файлам, то получится последовательность r/N серий. После второго прохода их число уменьшится до r/N^2, после третьего – до r/N^3, а после k проходов останется r/N^k серий. Поэтому полное число проходов, необходимых для сортировки n элементов с помощью N-путевого слияния, равно $k = \log_N(n)$. Поскольку каждый проход требует n операций копирования, полное число операций копирования в худшем случае равно $M = n \times \log_N(n)$.

В качестве следующего упражнения в программировании мы разработаем программу сортировки, основанную на многопутевых слияниях. Чтобы подчеркнуть отличие этой программы от приведенной выше программы естественных двухфазных слияний, мы сформулируем многопутевое слияние в виде однофазного сбалансированного слияния. Это подразумевает, что на каждом проходе есть равное число файлов-источников и файлов-приемников, в которые серии распределяются по очереди. Если используется 2N файлов, то говорят, что алгоритм основан на N-путевом слиянии. Следуя принятой ранее стратегии, мы не будем беспокоиться об отслеживании слияния двух последовательных серий, попавших в один файл. Поэтому нам нужно спроектировать программу слияния, не делая предположения о строго равном числе серий в файлах-источниках.

Здесь мы впервые встречаем ситуацию, когда естественно возникает структура данных, представляющая собой массив файлов. На самом деле удивительно, насколько сильно наша следующая программа отличается от предыдущей из-за перехода от двухпутевых к многопутевым слияниям. Главная причина этого – в том, что процесс слияния теперь не может просто остановиться после исчерпания одной из серий-источников. Вместо этого нужно сохранить список еще активных, то есть до конца не исчерпанных, файлов-источников. Другое усложнение возникает из-за необходимости менять роли файлов-источников и файлов-приемников. Здесь становится видно удобство принятого способа косвенного доступа к файлам с помощью бегунков. На каждом проходе данные можно копировать

с одной и той же группы бегунков r на одну и ту же группу бегунков w. А в конце каждого прохода нужно просто переключить бегунки r и w на другие группы файлов.

Очевидно, для индексирования массива файлов используются номера файлов. Предположим, что исходный файл представлен параметром src и что для процесса сортировки в наличии имеются 2N файлов:

```
f, g: ARRAY N OF Files.File;
r, w: ARRAY N OF Runs.Rider
```

Тогда можно написать следующий эскизный вариант алгоритма:

```
PROCEDURE BalancedMerge (src: Files.File): Files.File;
   (*сбалансированные слияния*)
   VAR i, j: INTEGER;
      L: INTEGER; (*число распределенных серий*)
      R: Runs.Rider;
BEGIN
   Runs.Set(R, src);  (*распределить начальные серии из R в w[0] ... w[N-1]*)
   j := 0; L := 0;
   подключить бегунки w к файлам g;
   REPEAT
      копировать одну серию из R в w[j];
      INC(j); INC(L);
      IF j = N THEN j := 0 END
   UNTIL R.eof;

   REPEAT (*слияние из r в w*)
      переключить бегунки r на файлы g;
      L := 0; j := 0;  (*j = индекс файла-приемника*)
      REPEAT
         INC(L);
         слить по одной серии из каждого источника в w[j];
         IF j < N THEN INC(j) ELSE j := 0 END
      UNTIL все источники исчерпаны;
   UNTIL L = 1
   (*отсортированный файл подсоединен к w[0]*)
END BalancedMerge.
```

Связав бегунок R с исходным файлом, займемся уточнением операции первичного распределения серий. Используя определение процедуры copy, заменим фразу *копировать одну серию из R в w[j]* на следующий оператор:

```
REPEAT Runs.copy(R, w[j]) UNTIL R.eor
```

Копирование серии прекращается, когда либо встретится первый элемент следующей серии, либо будет достигнут конец входного файла.

В реальном алгоритме сортировки нужно уточнить следующие операции:

(1) *подключить бегунки w к файлам g*;
(2) *слить по одной серии из каждого источника в* w_j;

(3) *переключить бегунки r на файлы g*;
(4) *все источники исчерпаны.*

Во-первых, нужно аккуратно определить текущие последовательности-источники. В частности, число *активных* источников может быть меньше N. Очевидно, источников не может быть больше, чем серий; сортировка прекращается, как только останется единственная последовательность. При этом остается возможность, что в начале последнего прохода сортировки число серий меньше N. Поэтому введем переменную, скажем k1, для обозначения реального числа источников. Инициализацию переменной k1 включим в операцию *переключить бегунки* следующим образом:

```
IF L < N THEN k1 := L ELSE k1 := N END;
FOR i := 0 TO k1-1 DO Runs.Set(r[i], g[i]) END
```

Естественно, в операции (2) нужно уменьшить k1 при исчерпании какого-либо источника. Тогда предикат (4) легко выразить в виде сравнения k1 = 0. Однако операцию (2) уточнить труднее; она состоит из повторного выбора наименьшего ключа среди имеющихся источников и затем его пересылки по назначению, то есть в текущую последовательность-приемник. Эта операция усложняется необходимостью определять конец каждой серии. Конец серии определяется, когда (a) следующий ключ меньше текущего или (b) досигнут конец последовательности-источника. В последнем случае источник удаляется уменьшением k1; в первом случае серия закрывается исключением последовательности из дальнейшего процесса выбора элементов, но только до завершения формирования текущей серии-приемника. Из этого видно, что нужна вторая переменная, скажем k2, для обозначения числа источников, реально доступных для выбора следующего элемента. Это число сначала устанавливается равным k1 и уменьшается каждый раз, когда серия прерывается по условию (a).

К сожалению, недостаточно ввести только k2. Нам нужно знать не только количество еще используемых файлов, но и какие именно это файлы. Очевидное решение – ввести массив из булевских элементов, чтобы отмечать такие файлы. Однако мы выберем другой способ, который приведет к более эффективной процедуре выбора, – ведь эта часть во всем алгоритме повторяется чаще всего. Вместо булевского массива введем косвенную индексацию файлов с помощью отображения (map) индексов посредством массива, скажем t. Отображение используется таким образом, что t_0 ... t_{k2-1} являются индексами доступных последовательностей. Теперь операция (2) может быть сформулирована следующим образом:

```
k2 := k1;
REPEAT
    выбрать минимальный ключ,
    пусть t[m] - номер последовательности, в которой он нашелся;
    Runs.copy(r[t[m]], w[j]);
    IF r[t[m]].eof THEN
        исключить последовательность
    ELSIF r[t[m]].eor THEN
```

```
        закрыть серию
    END
UNTIL k2 = 0
```

Поскольку число последовательностей на практике довольно мало, для алгоритма выбора, который требуется уточнить на следующем шаге, можно применить простой линейный поиск. Операция *исключить последовательность* подразумевает уменьшение k1 и k2, а операция *закрыть серию* – уменьшение только k2, причем обе операции включают в себя соответствующие перестановки элементов массива t. Детали показаны в следующей процедуре, которая и является результатом последнего уточнения. При этом операция *переключить бегунки* была раскрыта в соответствии с ранее данными объяснениями:

```
PROCEDURE BalancedMerge (src: Files.File): Files.File; (* ADruS24_MergeSorts *)
    (*сбалансированные слияния*)
    VAR i, j, m, tx: INTEGER;
        L, k1, k2, K1: INTEGER;
        min, x: INTEGER;
        t: ARRAY N OF INTEGER;  (*отображение индексов*)
        R: Runs.Rider;  (*входные данные*)
        f, g: ARRAY N OF Files.File;
        r, w: ARRAY N OF Runs.Rider;
BEGIN
    Runs.Set(R, src);
    FOR i := 0 TO N-1 DO
        g[i] := Files.New(""); Files.Set(w[i], g[i], 0)
    END;
    (*распределить начальные серии из src по файлам g[0] ... g[N-1]*)
    j := 0; L := 0;
    REPEAT
        REPEAT Runs.copy(R, w[j]) UNTIL R.eor;
        INC(L); INC(j);
        IF j = N THEN j := 0 END
    UNTIL R.eof;
    REPEAT
        IF L < N THEN k1 := L ELSE k1 := N END;
        K1 := k1;
        FOR i := 0 TO k1-1 DO (*установить бегунки-источники*)
            Runs.Set(r[i], g[i])
        END;
        FOR i := 0 TO k1-1 DO (*установить бегунки-приемники*)
            g[i] := Files.New(""); Files.Set(w[i], g[i], 0)
        END;

        (*слить из r[0] ... r[k1-1] в w[0] ... w[K1-1]*)
        FOR i := 0 TO k1-1 DO t[i] := i END;
        L := 0;  (*число серий на выходе слияния*)
        j := 0;
        REPEAT (*слить по одной серии из источников в w[j]*)
```

```
            INC(L); k2 := k1;
            REPEAT (*выбрать наименьший ключ*)
               m := 0; min := r[t[0]].first; i := 1;
               WHILE i < k2 DO
                  x := r[t[i]].first;
                  IF x < min THEN min := x; m := i END;
                  INC(i)
               END;
               Runs.copy(r[t[m]], w[j]);
               IF r[t[m]].eof THEN (*исключить последовательность*)
                  DEC(k1); DEC(k2);
                  t[m] := t[k2]; t[k2] := t[k1]
               ELSIF r[t[m]].eor THEN (*закрыть серию*)
                  DEC(k2);
                  tx := t[m]; t[m] := t[k2]; t[k2] := tx
               END
            UNTIL k2 = 0;
            INC(j);
            IF j = K1 THEN j := 0 END
         UNTIL k1 = 0
      UNTIL L = 1;
      RETURN g[0]
   END BalancedMerge
```

2.4.4. Многофазная сортировка

Мы обсудили необходимые приемы и приобрели достаточно опыта, чтобы исследовать и запрограммировать еще один алгоритм сортировки, который по производительности превосходит сбалансированную сортировку. Мы видели, что сбалансированные слияния устраняют операции простого копирования, которые нужны, когда операции распределения и слияния объединены в одной фазе. Возникает вопрос: можно ли еще эффективней обработать последовательности. Оказывается, можно. Ключом к очередному усовершенствованию является отказ от жесткого понятия проходов, то есть более изощренная работа с последовательностями, нежели использование проходов с N источниками и с таким же числом приемников, которые в конце каждого прохода меняются местами. При этом понятие прохода размывается. Этот метод был изобретен Гильстадом [2.3] и называется *многофазной сортировкой*.

Проиллюстрируем ее сначала примером с тремя последовательностями. В любой момент времени элементы сливаются из двух источников в третью последовательность. Как только исчерпывается одна из последовательностей-источников, она немедленно становится приемником для операций слияния данных из еще не исчерпанного источника и последовательности, которая только что была принимающей.

Так как при наличии n серий в каждом источнике получается n серий в приемнике, достаточно указать только число серий в каждой последовательности (вме-

сто того чтобы указывать конкретные ключи). На рис. 2.14 предполагается, что сначала есть 13 и 8 серий в последовательностях-источниках f0 и f1 соответственно. Поэтому на первом проходе 8 серий сливается из f0 и f1 в f2, на втором проходе остальные 5 серий сливаются из f2 и f0 в f1 и т. д. В конце концов, f0 содержит отсортированную последовательность.

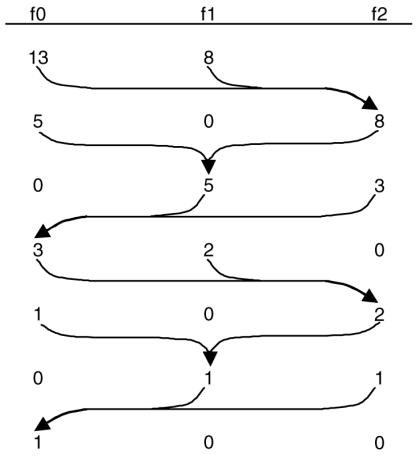


Рис. 2.14. Многофазная сортировка
с тремя последовательностями, содержащими 21 серию

Второй пример показывает многофазный метод с 6 последовательностями. Пусть вначале имеются 16 серий в последовательности f0, 15 в f1, 14 в f2, 12 в f3 и 8 в f4. В первом частичном проходе 8 серий сливаются на f5. В конце концов, f1 содержит отсортированный набор элементов (см. рис. 2.15).

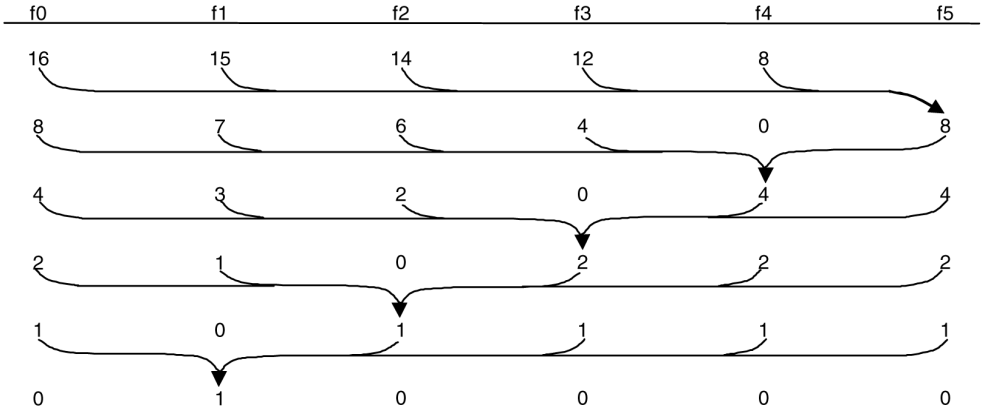


Рис. 2.15. Многофазная сортировка слиянием 65 серий
с использованием 6 последовательностей

Многофазная сортировка более эффективна, чем сбалансированная, так как при наличии N последовательностей она всегда реализует N–1-путевое слияние вместо N/2-путевого. Поскольку число требуемых проходов примерно равно $\log_N n$, где n – число сортируемых элементов, а N – количество сливаемых серий в одной операции слияния, то идея многофазной сортировки обещает существенное улучшение по сравнению со сбалансированной.

Разумеется, в приведенных примерах начальное распределение серий было тщательно подобрано. Чтобы понять, как серии должны быть распределены в начале сортировки для ее правильного функционирования, будем рассуждать в обратном порядке, начиная с окончательного распределения (последняя строка на рис. 2.15). Ненулевые числа серий в каждой строке рисунка (2 и 5 таких чисел на рис. 2.14 и 2.15 соответственно) запишем в виде строки таблицы, располагая по убыванию (порядок чисел в строке таблицы не важен). Для рис. 2.14 и 2.15 получатся табл. 2.13 и 2.14 соответственно. Количество строк таблицы соответствует числу проходов.

Таблица 2.13. Идеальные распределения серий в двух последовательностях

L	$a_0(L)$	$a_1(L)$	Sum $a_i(L)$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

Таблица 2.14. Идеальные распределения серий в пяти последовательностях

L	$a_0(L)$	$a_1(L)$	$a_2(L)$	$a_3(L)$	$a_4(L)$	Sum $a_i(L)$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Для табл. 2.13 получаем следующие соотношения для $L > 0$:

$$a_1(L+1) = a_0(L)$$
$$a_0(L+1) = a_0(L) + a_1(L)$$

вместе с $a_0(0) = 1$, $a_1(0) = 0$. Определяя $f_{i+1} = a_0(i)$, получаем для $i > 0$:

$$f_{i+1} = f_i + f_{i-1},\ f_1 = 1,\ f_0 = 0$$

Это в точности рекуррентное определение *чисел Фибоначчи*:

$$f = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Каждое число Фибоначчи равно сумме двух своих предшественников. Следовательно, начальные числа серий в двух последовательностях-источниках должны быть двумя последовательными числами Фибоначчи, чтобы многофазная сортировка правильно работала с тремя последовательностями.

А как насчет второго примера (табл. 2.14) с шестью последовательностями? Нетрудно вывести определяющие правила в следующем виде:

$$a_4(L+1) = a_0(L)$$
$$a_3(L+1) = a_0(L) + a_4(L) = a_0(L) + a_0(L-1)$$
$$a_2(L+1) = a_0(L) + a_3(L) = a_0(L) + a_0(L-1) + a_0(L-2)$$
$$a_1(L+1) = a_0(L) + a_2(L) = a_0(L) + a_0(L-1) + a_0(L-2) + a_0(L-3)$$
$$a_0(L+1) = a_0(L) + a_1(L) = a_0(L) + a_0(L-1) + a_0(L-2) + a_0(L-3) + a_0(L-4)$$

Подставляя f_i вместо $a_0(i)$, получаем

$$f_{i+1} = f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4} \quad \text{для } i > 4$$
$$f_4 = 1$$
$$f_i = 0 \quad \text{для } i < 4$$

Это *числа Фибоначчи порядка 4*. В общем случае числа Фибоначчи порядка p определяются так:

$$f_{i+1}(p) = f_i(p) + f_{i-1}(p) + \ldots + f_{i-p}(p) \quad \text{для } i > p$$
$$f_p(p) = 1$$
$$f_i(p) = 0 \quad \text{для } 0 \le i < p$$

Заметим, что обычные числа Фибоначчи получаются для $p = 1$.

Видим, что идеальные начальные числа серий для многофазной сортировки с N последовательностями суть суммы любого числа – $N-1$, $N-2$, ... , 1 – последовательных чисел Фибоначчи порядка $N-2$ (см. табл. 2.15).

Таблица 2.15. Числа серий, допускающие идеальное распределение

L \ N:	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001

Казалось бы, такой метод применим только ко входным данным, в которых число серий равно сумме N–1 таких чисел Фибоначчи. Поэтому возникает важный вопрос: что же делать, когда число серий вначале не равно такой идеальной сумме? Ответ прост (и типичен для подобных ситуаций): будем имитировать существование воображаемых пустых серий, так чтобы сумма реальных и воображаемых серий была равна идеальной сумме. Такие серии будем называть *фиктивными* (dummy).

Но этого на самом деле недостаточно, так как немедленно встает другой, более сложный вопрос: как распознавать фиктивные серии во время слияния? Прежде чем отвечать на него, нужно исследовать возникающую еще раньше проблему распределения начальных серий и решить, каким правилом руководствоваться при распределении реальных и фиктивных серий на N–1 лентах.

Чтобы найти хорошее правило распределения, нужно знать, как выполнять слияние реальных и фиктивных серий. Очевидно, что выбор фиктивной серии из i-й последовательности в точности означает, что i-я последовательность игнорируется в данном слиянии, так что речь идет о слиянии менее N–1 источников. Слияние фиктивных серий из всех N–1 источников означает отсутствие реальной операции слияния, но при этом в приемник нужно записать фиктивную серию. Отсюда мы заключаем, что фиктивные серии должны быть распределены по n–1 последовательностям как можно равномернее, так как хотелось бы иметь активные слияния с участием максимально большого числа источников.

На минуту забудем про фиктивные серии и рассмотрим проблему распределения некоторого неизвестного числа серий по N–1 последовательностям. Ясно, что числа Фибоначчи порядка N–2, указывающие желательное число серий в каждом источнике, могут быть сгенерированы в процессе распределения. Например, предположим, что N = 6, и, обращаясь к табл. 2.14, начнем с распределения серий, показанного в строке с индексом L = 1 (1, 1, 1, 1, 1); если еще остаются серии, переходим ко второму ряду (2, 2, 2, 2, 1); если источник все еще не исчерпан, распределение продолжается в соответствии с третьей строкой (4, 4, 4, 3, 2) и т. д. Индекс строки будем называть *уровнем*. Очевидно, что чем больше число серий, тем выше уровень чисел Фибоначчи, который, кстати говоря, равен числу проходов слияния или переключений приемника, которые нужно будет сделать в сортировке. Теперь первая версия алгоритма распределения может быть сформулирована следующим образом:

1. В качестве цели распределения (то есть желаемых чисел серий) взять числа Фибоначчи порядка N–2, уровня 1.
2. Распределять серии, стремясь достичь цели.
3. Если цель достигнута, вычислить числа Фибоначчи следующего уровня; разности между ними и числами на предыдущем уровне становятся новой целью распределения. Вернуться на шаг 2. Если же цель не достигнута, хотя источник исчерпан, процесс распределения завершается.

Правило вычисления чисел Фибоначчи очередного уровня содержится в их определении. Поэтому можно сосредоточить внимание на шаге 2, где при задан-

ной цели еще не распределенные серии должны быть распределены по одной в N–1 последовательностей-приемников. Именно здесь вновь появляются фиктивные серии.

Пусть после повышении уровня очередная цель представлена разностями d_i, i = 0 ... N–2, где d_i обозначает число серий, которые должны быть распределены в i-ю последовательность на очередном шаге. Здесь можно представить себе, что мы сразу помещаем d_i фиктивных серий в i-ю последовательность и рассматриваем последующее распределение как замену фиктивных серий реальными, при каждой замене вычитая единицу из d_i. Тогда d_i будет равно числу фиктивных серий в i-й последовательности на момент исчерпания источника.

Неизвестно, какой алгоритм дает оптимальное распределение, но очень хорошо зарекомендовало себя так называемое *горизонтальное распределение* (см. [2.7], с. 297). Это название можно понять, если представить себе, что серии сложены в стопки, как показано на рис. 2.16 для N = 6, уровень 5 (ср. табл. 2.14). Чтобы как можно быстрее достичь равномерного распределения остаточных фиктивных серий, последние заменяются реальными послойно слева направо, начиная с верхнего слоя, как показано на рис. 2.16.

8					
7	1				
6	2	3	4		
5	5	6	7	8	
4	9	10	11	12	
3	13	14	15	16	17
2	18	19	20	21	22
1	23	24	25	26	27
	28	29	30	31	32

Рис. 2.16. Горизонтальное распределение серий

Теперь можно описать соответствующий алгоритм в виде процедуры select, которая вызывается каждый раз, когда завершено копирование серии и выбирается новый приемник для очередной серии. Для обозначения текущей принимающей последовательности используется переменная j. a_i и d_i обозначают соответственно идеальное число серий и число фиктивных серий для i-й последовательности.

```
j, level:   INTEGER;
a, d:   ARRAY N OF INTEGER;
```

Эти переменные инициализируются следующими значениями:

$a_i = 1$, $d_i = 1$ для i = 0 ... N–2
$a_{N-1} = 0$, $d_{N-1} = 0$ (принимающая последовательность)
j = 0, level = 0

Заметим, что процедура select должна вычислить следующую строчку табл. 2.14, то есть значения $a_0(L)$ ... $a_{N-2}(L)$, при увеличении уровня. Одновременно вычисляется и очередная цель, то есть разности $d_i = a_i(L) - a_i(L-1)$. Приводимый алгоритм использует тот факт, что получающиеся d_i убывают с возрастанием индекса («нисходящая лестница» на рис. 2.16). Заметим, что переход с уровня 0 на уровень 1 является исключением; поэтому данный алгоритм должен стартовать

с уровня 1. Процедура select заканчивается уменьшением d_j на 1; эта операция соответствует замене фиктивной серии в j-й последовательности на реальную.

```
PROCEDURE select;  (*выбор приемника*)
    VAR i, z: INTEGER;
BEGIN
    IF d[j] < d[j+1] THEN
        INC(j)
    ELSE
        IF d[j] = 0 THEN
            INC(level);
            z := a[0];
            FOR i := 0 TO N-2 DO
                d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
            END
        END;
        j := 0
    END;
    DEC(d[j])
END select
```

Предполагая наличие процедуры для копирования серии из последовательности-источника src с бегунком R в последовательность f_j с бегунком r_j, мы можем следующим образом сформулировать фазу начального распределения (предполагается, что источник содержит хотя бы одну серию):

```
REPEAT select; copyrun
UNTIL R.eof
```

Однако здесь следует остановиться и вспомнить о явлении, имевшем место при распределении серий в ранее обсуждавшейся сортировке естественными слияниями, а именно: две серии, последовательно попадающие в один приемник, могут «слипнуться» в одну, так что подсчет серий даст неверный результат. Этим можно пренебречь, если алгоритм сортировки таков, что его правильность не зависит от числа серий. Но в многофазной сортировке необходимо тщательно отслеживать точное число серий в каждом файле. Следовательно, здесь нельзя пренебрегать случайным «слипанием» серий. Поэтому невозможно избежать дополнительного усложнения алгоритма распределения. Теперь необходимо удерживать ключ последнего элемента последней серии в каждой последовательности. К счастью, именно это делает наша реализация процедуры Runs. Для принимающих последовательностей поле бегунка r.first хранит последний записанный элемент. Поэтому следующая попытка написать алгоритм распределения может быть такой:

```
REPEAT select;
    IF r[j].first <= R.first THEN продолжать старую версию END;
    copyrun
UNTIL R.eof
```

Очевидная ошибка здесь в том, что мы забыли, что r[j].first получает значение только после копирования первой серии. Поэтому корректное решение требует сначала распределить по одной серии в каждую из N–1 принимающих последовательностей без обращения к first. Оставшиеся серии распределяются следующим образом:

```
WHILE ~R.eof DO
    select;
    IF r[j].first <= R.first THEN
        copyrun;
        IF R.eof THEN INC(d[j]) ELSE copyrun END
    ELSE
        copyrun
    END
END
```

Наконец, можно заняться главным алгоритмом многофазной сортировки. Его принципиальная структура подобна основной части программы N-путевого слияния: внешний цикл, в теле которого сливаются серии, пока не исчерпаются источники, внутренний цикл, в теле которого сливается по одной серии из каждого источника, а также самый внутренний цикл, в теле которого выбирается начальный ключ и соответствующий элемент пересылается в выходной файл. Главные отличия от сбалансированного слияния в следующем:

1. Теперь на каждом проходе вместо N приемников есть только один.
2. Вместо переключения N источников и N приемников после каждого прохода происходит ротация последовательностей. Для этого используется косвенная индексация последовательностей при посредстве массива t.
3. Число последовательностей-источников меняется от серии к серии; в начале каждой серии оно определяется по счетчикам фиктивных последовательностей d_i. Если $d_i > 0$ для всех i, то имитируем слияние N–1 фиктивных последовательностей в одну простым увеличением счетчика d_{N-1} для последовательности-приемника. В противном случае сливается по одной серии из всех источников с $d_i = 0$, а для всех остальных последовательностей d_i уменьшается, показывая, что число фиктивных серий в них уменьшилось. Число последовательностей-источников, участвующих в слиянии, обозначим как k.
4. Невозможно определить окончание фазы по обнаружению конца последовательности с номером N–1, так как могут понадобиться дальнейшие слияния с участием фиктивных серий из этого источника. Вместо этого теоретически необходимое число серий определяется по коэффициентам a_i. Эти коэффициенты были вычислены в фазе распределения; теперь они могут быть восстановлены обратным вычислением.

Теперь в соответствии с этими правилами можно сформулировать основную часть многофазной сортировки, предполагая, что все N–1 последовательности с начальными сериями подготовлены для чтения и что массив отображения индексов инициализирован как $t_i = i$.

```
REPEAT (*слить из t[0] ... t[N-2] в t[N-1]*)
    z := a[N-2]; d[N-1] := 0;
    REPEAT  (*слить одну серию*)
        k := 0;
        (*определить число активных источников*)
        FOR i := 0 TO N-2 DO
            IF d[i] > 0 THEN
                DEC(d[i])
            ELSE
                ta[k] := t[i]; INC(k)
            END
        END;
        IF k = 0 THEN
            INC(d[N-1])
        ELSE
            слить одну реальную серию из t[0] ... t[k-1] в t[N-1]
        END;
        DEC(z)
    UNTIL z = 0;
    Runs.Set(r[t[N-1]], f[t[N-1]]);
    выполнить ротацию последовательностей в отображении t;
    вычислить a[i] для следующего уровня;
    DEC(level)
UNTIL level = 0
(*результат сортировки находится в f[t[0]]*)
```

Реальная операция слияния почти такая же, как в сортировке N-путевыми слияниями, единственное отличие в том, что слегка упрощается алгоритм исключения последовательности. Ротация в отображении индексов последовательностей и соответствующих счетчиков d_i (а также перевычисление a_i при переходе на уровень вниз) не требует каких-либо ухищрений; детали можно найти в следующей программе, целиком реализующей алгоритм многофазной сортировки:

```
PROCEDURE Polyphase (src: Files.File): Files.File;        (* ADruS24_MergeSorts *)

    (*многофазная сортировка*)
    VAR i, j, mx, tn: INTEGER;
        k, dn, z, level: INTEGER;
        x, min: INTEGER;
        a, d:  ARRAY N OF INTEGER;
        t, ta:  ARRAY N OF INTEGER;  (*отображения индексов*)
        R: Runs.Rider;  (*исходные данные*)
        f: ARRAY N OF Files.File;
        r: ARRAY N OF Runs.Rider;

    PROCEDURE select;  (*выбор*)
        VAR i, z: INTEGER;
    BEGIN
        IF d[j] < d[j+1] THEN
            INC(j)
```

```
        ELSE
            IF d[j] = 0 THEN
                INC(level);
                z := a[0];
                FOR i := 0 TO N-2 DO
                    d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
                END
            END;
            j := 0
        END;
        DEC(d[j])
    END select;

    PROCEDURE copyrun;  (*из src в f[j]*)
    BEGIN
        REPEAT Runs.copy(R, r[j]) UNTIL R.eor
    END copyrun;

BEGIN
    Runs.Set(R, src);
    FOR i := 0 TO N-2 DO
        a[i] := 1; d[i] := 1;
        f[i] := Files.New(""); Files.Set(r[i], f[i], 0)
    END;
    (*распределить начальные серии*)
    level := 1; j := 0; a[N-1] := 0; d[N-1] := 0;
    REPEAT
        select; copyrun
    UNTIL R.eof OR (j = N-2);
    WHILE ~R.eof DO
        select;  (*r[j].first = последний элемент, записанный в f[j]*)
        IF r[j].first <= R.first THEN
            copyrun;
            IF R.eof THEN INC(d[j]) ELSE copyrun END
        ELSE
            copyrun
        END
    END;

    FOR i := 0 TO N-2 DO
        t[i] := i; Runs.Set(r[i], f[i])
    END;
    t[N-1] := N-1;
    REPEAT (*слить из t[0] ... t[N-2] в t[N-1]*)
        z := a[N-2]; d[N-1] := 0;
        f[t[N-1]] := Files.New(""); Files.Set(r[t[N-1]], f[t[N-1]], 0);
        REPEAT (*слить одну серию*)
            k := 0;
            FOR i := 0 TO N-2 DO
                IF d[i] > 0 THEN
                    DEC(d[i])
```

```
            ELSE
               ta[k] := t[i]; INC(k)
            END
         END;
         IF k = 0 THEN
            INC(d[N-1])
         ELSE (*слить одну реальную серию из t[0] ... t[k-1] в t[N-1]*)
            REPEAT
               mx := 0; min := r[ta[0]].first; i := 1;
               WHILE i < k DO
                  x := r[ta[i]].first;
                  IF x < min THEN min := x; mx := i END;
                  INC(i)
               END;
               Runs.copy(r[ta[mx]], r[t[N-1]]);
               IF r[ta[mx]].eor THEN
                  ta[mx] := ta[k-1]; DEC(k)
               END
            UNTIL k = 0
         END;
         DEC(z)
      UNTIL z = 0;

      Runs.Set(r[t[N-1]], f[t[N-1]]);   (*ротация последовательностей*)
      tn := t[N-1]; dn := d[N-1]; z := a[N-2];
      FOR i := N-1 TO 1 BY -1 DO
         t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
      END;
      t[0] := tn; d[0] := dn; a[0] := z;
      DEC(level)
   UNTIL level = 0 ;
   RETURN f[t[0]]
END Polyphase
```

2.4.5. Распределение начальных серий

Необходимость использовать сложные программы последовательной сортировки возникает из-за того, что более простые методы, работающие с массивами, можно применять только при наличии достаточно большого объема оперативной памяти для хранения всего сортируемого набора данных. Часто оперативной памяти не хватает; вместо нее нужно использовать достаточно вместительные устройства хранения данных с последовательным доступом, такие как ленты или диски. Мы знаем, что развитые выше методы сортировки последовательностей практически не нуждаются в оперативной памяти, не считая, конечно, буферов для файлов и, разумеется, самой программы. Однако даже в небольших компьютерах размер оперативной памяти, допускающей произвольный доступ, почти всегда больше, чем нужно для разработанных здесь программ. Не суметь ее использовать оптимальным образом непростительно.

Решение состоит в том, чтобы скомбинировать методы сортировки массивов и последовательностей. В частности, в фазе начального распределения серий можно использовать вариант сортировки массивов, чтобы серии сразу имели длину L, соответствующую размеру доступной оперативной памяти. Понятно, что в последующих проходах слияния нельзя повысить эффективность с помощью сортировок массивов, так как длина серий только растет, и они в дальнейшем остаются больше, чем доступная оперативная память. Так что можно спокойно ограничиться усовершенствованием алгоритма, порождающего начальные серии.

Естественно, мы сразу ограничим наш выбор логарифмическими методами сортировки массивов. Самый подходящий здесь метод – турнирная сортировка, или HeapSort (см. раздел 2.3.2). Используемую там пирамиду можно считать фильтром, сквозь который должны пройти все элементы – одни быстрее, другие медленнее. Наименьший ключ берется непосредственно с вершины пирамиды, а его замещение является очень эффективной процедурой. Фильтрация элемента из последовательности-источника src (бегунок r0) сквозь всю пирамиду H в принимающую последовательность (бегунок r1) допускает следующее простое описание:

```
Write(r1, H[0]); Read(r0, H[0]); sift(0, n-1)
```

Процедура sift описана в разделе 2.3.2, с ее помощью вновь вставленный элемент H_0 просеивается вниз на свое правильное место. Заметим, что H_0 является наименьшим элементом в пирамиде. Пример показан на рис. 2.17. В итоге программа существенно усложняется по следующим причинам:

1. Пирамида H вначале пуста и должна быть заполнена.
2. Ближе к концу пирамида заполнена лишь частично, и в итоге она становится пустой.
3. Нужно отслеживать начало новых серий, чтобы в правильный момент изменить индекс принимающей последовательности j.

Прежде чем продолжить, формально объявим переменные, которые заведомо нужны в процедуре:

```
VAR  L, R, x: INTEGER;
    src, dest: Files.File;
    r, w: Files.Rider;
    H: ARRAY M OF INTEGER;  (*пирамида*)
```

Константа M – размер пирамиды H. Константа mh будет использоваться для обозначения M/2; L и R суть индексы, ограничивающие пирамиду. Тогда процесс фильтрации разбивается на пять частей:

1. Прочесть первые mh ключей из src (r) и записать их в верхнюю половину пирамиды, где упорядоченность ключей не требуется.
2. Прочесть другую порцию mh ключей и записать их в нижнюю половину пирамиды, просеивая каждый из них в правильную позицию (построить пирамиду).

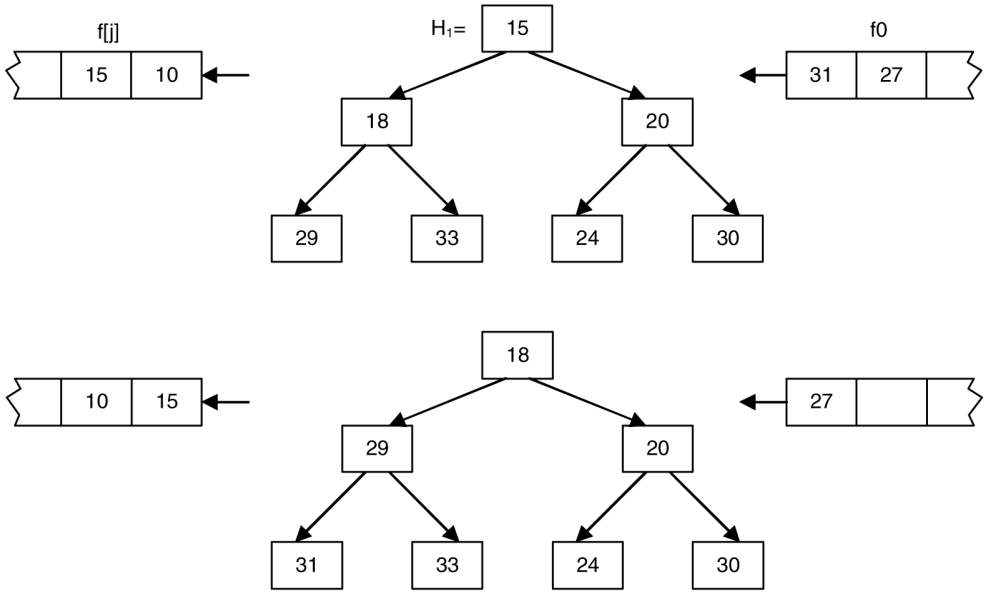


Рис. 2.17. Просеивание ключа сквозь пирамиду

3. Установить L равным M и повторять следующий шаг для всех остальных элементов в src: переслать элемент H_0 в последовательность-приемник. Если его ключ меньше или равен ключу следующего элемента в исходной последовательности, то этот следующий элемент принадлежит той же серии и может быть просеян в надлежащую позицию. В противном случае нужно уменьшить размер пирамиды и поместить новый элемент во вторую, верхнюю пирамиду, которая строится для следующей серии. Границу между двумя пирамидами указывает индекс L, так что нижняя (текущая) пирамида состоит из элементов H_0 ... H_{L-1}, а верхняя (следующая) – из H_L ... H_{M-1}. Если L = 0, то нужно переключить приемник и снова установить L равным M.
4. Когда исходная последовательность исчерпана, нужно сначала установить R равным M; затем «сбросить» нижнюю часть, чтобы закончить текущую серию, и одновременно строить верхнюю часть, постепенно перемещая ее в позиции H_L ... H_{R-1}.
5. Последняя серия генерируется из элементов, оставшихся в пирамиде.

Теперь можно в деталях выписать все пять частей в виде полной программы, вызывающей процедуру switch каждый раз, когда обнаружен конец серии и требуется некое действие для изменения индекса выходной последовательности. Вместо этого в приведенной ниже программе используется процедура-«затычка», а все серии направляются в последовательность dest.

Если теперь попытаться объединить эту программу, например, с многофазной сортировкой, то возникает серьезная трудность: программа сортировки содержит в начальной части довольно сложную процедуру переключения между последовательностями и использует процедуру copyrun, которая пересылает в точности одну серию в выбранный приемник. С другой стороны, программа HeapSort сложна и использует независимую процедуру select, которая просто выбирает новый приемник. Проблемы не было бы, если бы в одной (или обеих) программе нужная процедура вызывалась только в одном месте; но она вызывается в нескольких местах в обеих программах.

В таких случаях – то есть при совместном существовании нескольких процессов – лучше всего использовать *сопрограммы*. Наиболее типичной является комбинация процесса, производящего поток информации, состоящий из отдельных порций, и процесса, потребляющего этот поток. Эта связь типа производитель-потребитель может быть выражена с помощью двух сопрограмм; одной из них может даже быть сама главная программа. Сопрограмму можно рассматривать как процесс, который содержит одну или более точек прерывания (breakpoint). Когда встречается такая точка, управление возвращается в процедуру, вызвавшую сопрограмму. Когда сопрограмма вызывается снова, выполнение продолжается с той точки, где оно было прервано. В нашем примере мы можем рассматривать многофазную сортировку как основную программу, вызывающую copyrun, которая оформлена как сопрограмма. Она состоит из главного тела приводимой ниже программы, в которой каждый вызов процедуры switch теперь должен считаться точкой прерывания. Тогда проверку конца файла нужно всюду заменить проверкой того, достигла ли сопрограмма своего конца.

```
PROCEDURE Distribute (src: Files.File): Files.File;        (* ADruS24_MergeSorts *)
  CONST M = 16; mh = M DIV 2;  (*размер пирамиды*)
    VAR L, R: INTEGER;
        x: INTEGER;
        dest: Files.File;
        r, w: Files.Rider;
        H: ARRAY M OF INTEGER;  (*пирамида*)

    PROCEDURE sift (L, R: INTEGER);  (*просеить*)
        VAR i, j, x: INTEGER;
    BEGIN
        i := L; j := 2*L+1; x := H[i];
        IF (j < R) & (H[j] > H[j+1]) THEN INC(j) END;
        WHILE (j <= R) & (x > H[j]) DO
            H[i] := H[j]; i := j; j := 2*j+1;
            IF (j < R) & (H[j] > H[j+1]) THEN INC(j) END
        END;
        H[i] := x
    END sift;

BEGIN
    Files.Set(r, src, 0);
```

```
    dest := Files.New(""); Files.Set(w, dest, 0);
(*шаг 1: заполнить верхнюю половину пирамиды*)
    L := M;
    REPEAT DEC(L); Files.ReadInt(r, H[L]) UNTIL L = mh;
(*шаг 2: заполнить нижнюю половину пирамиды*)
    REPEAT DEC(L); Files.ReadInt(r, H[L]); sift(L, M-1) UNTIL L = 0;
(*шаг 3: пропустить элементы сквозь пирамиду*)
    L := M;
    Files.ReadInt(r, x);
    WHILE ~r.eof DO
        Files.WriteInt(w, H[0]);
        IF H[0] <= x THEN
            (*x принадлежит той же серии*) H[0] := x; sift(0, L-1)
        ELSE (*начать новую серию*)
            DEC(L); H[0] := H[L]; sift(0, L-1); H[L] := x;
            IF L < mh THEN sift(L, M-1) END;
            IF L = 0 THEN (*пирамида полна; начать новую серию*) L := M END
        END;
        Files.ReadInt(r, x)
    END;
(*шаг 4: сброс нижней половины пирамиды*)
    R := M;
    REPEAT
        DEC(L); Files.WriteInt(w, H[0]);
        H[0] := H[L]; sift(0, L-1); DEC(R); H[L] := H[R];
        IF L < mh THEN sift(L, R-1) END
    UNTIL L = 0;
(*шаг 5: сброс верхней половины пирамиды, начать новую серию*)
    WHILE R > 0 DO
        Files.WriteInt(w, H[0]); H[0] := H[R]; DEC(R); sift(0, R)
    END;
    RETURN dest
END Distribute
```

Анализ и выводы. Какой производительности можно ожидать от многофазной сортировки, если распределение начальных серий выполняется с помощью алгоритма HeapSort? Обсудим сначала, какого улучшения можно ожидать от введения пирамиды.

В последовательности со случайно распределенными ключами средняя длина серий равна 2. Чему равна эта длина после того, как последовательность профильтрована через пирамиду размера m? Интуиция подсказывает ответ m, но, к счастью, результат вероятностного анализа гораздо лучше, а именно 2m (см. [2.7], раздел 5.4.1). Поэтому ожидается улучшение на фактор m.

Производительность многофазной сортировки можно оценить из табл. 2.15, где указано максимальное число начальных серий, которые можно отсортировать за заданное число частичных проходов (уровней) с заданным числом последовательностей N. Например, с шестью последовательностями и пирамидой размера

m = 100 файл, содержащий до 165'680'100 начальных серий, может быть отсортирован за 10 частичных проходов. Это замечательная производительность.

Рассматривая комбинацию сортировок Polyphase и HeapSort, нельзя не удивляться сложности этой программы. Ведь она решает ту же легко формулируемую задачу перестановки элементов, которую решает и любой из простых алгоритмов сортировки массива.

Мораль всей главы можно сформулировать так:

1. Существует теснейшая связь между алгоритмом и стуктурой обрабатываемых данных, и эта структура влияет на алгоритм.
2. Удается находить изощренные способы для повышения производительности программы, даже если данные приходится организовывать в структуру, которая плохо подходит для решения задачи (последовательность вместо массива).

Упражнения

2.1. Какие из рассмотренных алгоритмов являются устойчивыми методами сортировки?

2.2. Будет ли алгоритм для двоичных вставок работать корректно, если в операторе WHILE условие L < R заменить на L ≤ R? Останется ли он корректным, если оператор L := m+1 упростить до L := m? Если нет, то найти набор значений $a_0 \ldots a_{n-1}$, на котором измененная программа сработает неправильно.

2.3. Запрограммируйте и измерьте время выполнения на вашем компьютере трех простых методов сортировки и найдите коэффициенты, на которые нужно умножать факторы C и M, чтобы получались реальные оценки времени.

2.4. Укажите инварианты циклов для трех простых алгоритмов сортировки.

2.5. Рассмотрите следующую «очевидную» версию процедуры Partition и найдите набор значений $a_0 \ldots a_{n-1}$, на котором эта версия не сработает:

```
i := 0; j := n-1; x := a[n DIV 2];
REPEAT
    WHILE a[i] < x DO i := i+1 END;
    WHILE x < a[j] DO j := j-1 END;
    w := a[i]; a[i] := a[j]; a[j] := w
UNTIL i > j
```

2.6. Напишите процедуру, которая следующим образом комбинирует алгоритмы QuickSort и BubbleSort: сначала QuickSort используется для получения (неотсортированных) сегментов длины m (1 < m < n); затем для завершения сортировки используется BubbleSort. Заметим, что BubbleSort может проходить сразу по всему массиву из n элементов, чем минимизируются организационные расходы. Найдите значение m, при котором полное время сортировки минимизируется.

Замечание. Ясно, что оптимальное значение m будет довольно мало. Поэтому может быть выгодно в алгоритме BubbleSort сделать в точности m-1 про-

ходов по массиву без использования последнего прохода, в котором проверяется, что обмены больше не нужны.

2.7. Выполнить эксперимент из упражнения 2.6, используя сортировку простым выбором вместо BubbleSort. Естественно, сортировка выбором не может работать сразу со всем массивом, поэтому здесь работа с индексами потребует больше усилий.

2.8. Напишите рекурсивный алгоритм быстрой сортировки так, чтобы сортировка более короткого сегмента производилась до сортировки длинного. Первую из двух подзадач решайте с помощью итерации, для второй используйте рекурсивный вызов. (Поэтому ваша процедура сортировки будет содержать только один рекурсивный вызов вместо двух.)

2.9. Найдите перестановку ключей 1, 2, ... , n, для которой алгоритм QuickSort демонстрирует наихудшее (наилучшее) поведение (n = 5, 6, 8).

2.10. Постройте программу естественных слияний, которая подобно процедуре простых слияний работает с массивом двойной длины с обоих концов внутрь; сравните ее производительность с процедурой в тексте.

2.11. Заметим, что в (двухпутевом) естественном слиянии, вместо того чтобы всегда слепо выбирать наименьший из доступных для просмотра ключей, мы поступаем по-другому: когда обнаруживается конец одной из двух серий, хвост другой просто копируется в принимающую последовательность. Например, слияние последовательностей

2, 4, 5, 1, 2, ...
3, 6, 8, 9, 7, ...

дает

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

вместо последовательности

2, 3, 4, 5, 1, 2, 6, 8, 9, ...

которая кажется упорядоченной лучше. В чем причина выбора такой стратегии?

2.12. Так называемое каскадное слияние (см. [2.1] и [2.7], раздел 5.4.3) – это метод сортировки, похожий на многофазную сортировку. В нем используется другая схема слияний. Например, если даны шесть последовательностей T1 ... T6, то каскадное слияние, тоже начинаясь с некоторого идеального распределения серий на T1 ... T5, выполняет 5-путевое слияние из T1 ... T5 на T6, пока не будет исчерпана T5, затем (не трогая T6), 4-путевое слияние на T5, затем 3-путевое на T4, 2-путевое – на T3, и, наконец, копирование из T1 на T2. Следующий проход работает аналогично, начиная с 5-путевого слияния на T1, и т. д. Хотя кажется, что такая схема будет хуже многофазной сортировки из-за того, что в ней некоторые последовательности иногда бездействуют, а также из-за использования операций простого копирования, она удивительным образом превосходит многофазную сортировку для

(очень) больших файлов и в случае шести и более последовательностей. Напишите хорошо структурированную программу на основе идеи каскадных слияний.

Литература

[2.1] Betz B. K. and Carter. Proc. ACM National Conf. 14, (1959), Paper 14.
[2.2] Floyd R. W. Treesort (Algorithms 113 and 243). Comm. ACM, 5, No. 8, (1962), 434, and Comm. ACM, 7, No. 12 (1964), 701.
[2.3] Gilstad R. L. Polyphase Merge Sorting – An Advanced Technique. Proc. AFIPS Eastern Jt. Comp. Conf., 18, (1960), 143–148.
[2.4] Hoare C. A. R. Proof of a Program: FIND. Comm. ACM, 13, No. 1, (1970), 39–45.
[2.5] Hoare C. A. R. Proof of a Recursive Program: Quicksort. Comp. J., 14, No. 4 (1971), 391–395.
[2.6] Hoare C. A. R. Quicksort. Comp. J., 5. No. 1 (1962), 10–15.
[2.7] Knuth D. E. The Art of Computer Programming. Vol. 3. Reading, Mass.: Addison-Wesley, 1973 (имеется перевод: Кнут Д. Э. Искусство программирования. 2-е изд. Т. 3. – М.: Вильямс, 2000).
[2.8] Lorin H. A Guided Bibliography to Sorting. IBM Syst. J., 10, No. 3 (1971), 244–254 (см. также Лорин Г. Сортировка и системы сортировки. – М.: Наука, 1983).
[2.9] Shell D. L. A Highspeed Sorting Procedure. Comm. ACM, 2, No. 7 (1959), 30–32.
[2.10] Singleton R. C. An Efficient Algorithm for Sorting with Minimal Storage (Algorithm 347). Comm. ACM, 12, No. 3 (1969), 185.
[2.11] Van Emden M. H. Increasing the Efficiency of Quicksort (Algorithm 402). Comm. ACM, 13, No. 9 (1970), 563–566, 693.
[2.12] Williams J. W. J. Heapsort (Algorithm 232) Comm. ACM, 7, No. 6 (1964), 347–348.

Глава 3

Рекурсивные алгоритмы

3.1. Введение

Объект называется рекурсивным, если его части определены через него самого. Рекурсия встречается не только в математике, но и в обычной жизни. Кто не видел рекламной картинки, которая содержит саму себя?

Рис. 3.1. Рекурсивное изображение

Рекурсия особенно хорошо являет свою мощь в математических определениях. Знакомые примеры – натуральные числа, древесные структуры и некоторые функции:

1. Натуральные числа:
   (a) 0 является натуральным числом.
   (b) Число, следующее за натуральным, является натуральным.
2. Древесные структуры:
   (a) ∅ является деревом (и называется «пустым деревом»).
   (b) Если t_1 и t_2 – деревья, то конструкция, состоящая из узла с двумя потомками t_1 и t_2, тоже является деревом (двоичным или бинарным).
3. Факториальная функция f(n):
   $f(0) = 1$
   $f(n) = n \times f(n - 1)$ для $n > 0$

Очевидно, мощь рекурсии заключается в возможности определить бесконечное множество объектов с помощью конечного утверждения. Подобным же образом бесконечное число расчетов может быть описано конечной рекурсивной программой, даже если программа не содержит явных циклов. Однако рекурсивные алгоритмы уместны прежде всего тогда, когда решаемая проблема, вычисляемая функция или обрабатываемая структура данных заданы рекурсивным образом. В общем случае рекурсивная программа P может быть выражена как композиция **P** последовательности инструкций S (не содержащей P) и самой P:

$P \equiv \mathbf{P}[S, P]$

Необходимое и достаточное средство для рекурсивной формулировки программ – процедура, так как она позволяет дать набору инструкций имя, с помощью которого эти инструкции могут быть вызваны. Если процедура P содержит явную ссылку на саму себя, то говорят, что она *явно рекурсивна*; если P содержит ссылку на другую процедуру Q, которая содержит (прямую или косвенную) ссылку на P, то говорят, что P *косвенно рекурсивна*. Последнее означает, что наличие рекурсии может быть не очевидно из текста программы.

С процедурой обычно ассоциируется набор локальных переменных, констант, типов и процедур, которые определены как локальные в данной процедуре и не существуют и не имеют смысла вне ее. При каждой рекурсивной активации процедуры создается новый набор локальных переменных. Хотя у них те же имена, что и у переменных в предыдущей активации процедуры, их значения другие, и любая возможность конфликта устраняется правилами видимости идентификаторов: идентификаторы всегда ссылаются на набор переменных, созданный последним. Такое же правило действует для параметров процедуры, которые по определению связаны с ней.

Как и в случае операторов цикла, рекурсивные процедуры открывают возможность бесконечных вычислений. Следовательно, необходимо рассматривать проблему остановки. Очевидное фундаментальное требование состоит в том, чтобы рекурсивные вызовы процедуры P имели место лишь при выполнении условия B, которое в какой-то момент перестает выполняться. Поэтому схема рекурсивных алгоритмов точнее выражается одной из следующих форм:

P ≡ IF B THEN **P**[S, P] END
P ≡ **P**[S, IF B THEN P END]

Основной метод доказательства остановки повторяющихся процессов состоит из следующих шагов:

1) определяется целочисленная функция f(x) (где x – набор переменных) – такая, что из f(x) < 0 следует условие остановки (фигурирующее в операторе while или repeat);
2) доказывается, что f(x) уменьшается на каждом шаге процесса.

Аналогично доказывают прекращение рекурсии: достаточно показать, что каждая активация P уменьшает некоторую целочисленную функцию f(x) и что f(x) < 0 влечет ~B. Особенно ясный способ гарантировать остановку состоит в том, чтобы ассоциировать передаваемый по значению параметр (назовем его n) с процедурой P, и рекурсивно вызывать P с n–1 в качестве значения этого параметра. Тогда, подставляя n > 0 вместо B, получаем гарантию прекращения. Это можно выразить следующими схемами:

P(n) ≡ IF n > 0 THEN **P**[S, P(n–1)] END
P(n) ≡ **P**[S, IF n > 0 THEN P(n–1) END]

В практических приложениях нужно доказывать не только конечность глубины рекурсии, но и что эта глубина достаточно мала. Причина в том, что при каждой рекурсивной активации процедуры P используется некоторый объем опера-

тивной памяти для размещения ее локальных переменных. Кроме того, нужно запомнить текущее состояние вычислительного процесса, чтобы после окончания новой активации P могла быть возобновлена предыдущая. Мы уже встречали такую ситуацию в процедуре QuickSort в главе 2. Там было обнаружено, что при наивном построении программы из операции, которая разбивает n элементов на две части, и двух рекурсивных вызовов сортировки для двух частей глубина рекурсии может в худшем случае приближаться к n. Внимательный анализ позволил ограничить глубину величиной порядка log(n). Разница между n и log(n) достаточно существенна, чтобы превратить ситуацию, в которой рекурсия в высшей степени неуместна, в такую, где рекурсия становится вполне практичной.

3.2. Когда не следует использовать рекурсию

Рекурсивные алгоритмы особенно хорошо подходят для тех ситуаций, когда решаемая задача или обрабатываемые данные определены рекурсивно. Однако наличие рекурсивного определения еще не означает, что рекурсивный алгоритм даст наилучшее решение. Именно попытки объяснять понятие рекурсивного алгоритма с помощью неподходящих примеров стали главной причиной широко распространенного предубеждения против использования рекурсии в программировании, а также мнения о неэффективности рекурсии.

Программы, в которых следует избегать использования алгоритмической рекурсии, характеризуются определенной структурой. Для них характерно наличие единственного вызова P в конце (или в начале) композиции (так называемая *концевая рекурсия*):

```
P ≡ IF B THEN S; P END
P ≡ S; IF B THEN P END
```

Такие схемы естественно возникают в тех случаях, когда вычисляемые значения определяются простыми рекуррентными соотношениями. Возьмем известный пример факториала $f_i = i!$:

```
i   = 0, 1, 2, 3, 4,   5,    ...
f_i = 1, 1, 2, 6, 24, 120, ...
```

Первое значение определено явно: $f_0 = 1$, а последующие – рекурсивно через предшествующие:

$$f_{i+1} = (i+1) * f_i$$

Это рекуррентное соотношение наводит на мысль использовать рекурсивный алгоритм для вычисления n-го факториала. Если ввести две переменные I и F для обозначения значений i и f_i на i-м уровне рекурсии, то переход к следующим членам пары последовательностей для i и f_i требует такого вычисления:

```
I := I + 1;  F := I * F
```

Подставляя эту пару инструкций вместо S, получаем рекурсивную программу

```
P ≡ IF I < n THEN I := I + 1; F := I * F; P END
I := 0; F := 1; P
```

В принятой нами нотации первая строка выражается следующим образом:

```
PROCEDURE P;
BEGIN
    IF I < n THEN I := I + 1; F := I*F; P END
END P
```

Чаще используется эквивалентная форма, данная ниже. P заменяется процедурой-функцией F, то есть процедурой, с которой явно ассоциируется вычисляемое значение и которая может поэтому быть использована как непосредственная составная часть выражений. Тогда переменная F становится лишней, а роль I берет на себя явно задаваемый параметр процедуры:

```
PROCEDURE F(I: INTEGER): INTEGER;
BEGIN
    IF I > 0 THEN RETURN I * F(I - 1) ELSE RETURN 1 END
END F
```

Ясно, что в этом примере рекурсия может быть довольно легко заменена итерацией. Это выражается следующей программой:

```
I := 0;  F := 1;
WHILE I < n DO I := I + 1; F := I*F END
```

В общем случае программы, построенные по обсуждаемым частным рекурсивным схемам, следует переписывать в соответствии со следующим образцом:

```
P ≡ [x := x0; WHILE B DO S END]
```

Существуют и более сложные рекурсивные композиционные схемы, которые могут и должны приводиться к итеративному виду. Пример – вычисление чисел Фибоначчи, определенных рекуррентным соотношением

$fib_{n+1} = fib_n + fib_{n-1}$ для $n > 0$

и соотношениями $fib_1 = 1$, $fib_0 = 0$. Непосредственный наивный перевод на язык программирования дает следующую рекурсивную программу:

```
PROCEDURE Fib (n: INTEGER): INTEGER;
    VAR res: INTEGER;
BEGIN
    IF n = 0 THEN res := 0
    ELSIF n = 1 THEN res := 1
    ELSE res := Fib(n-1) + Fib(n-2)
    END;
    RETURN res
END Fib
```

Вычисление fib_n с помощью вызова Fib(n) вызывает рекурсивные активации этой процедуры-функции. Сколько происходит таких активаций? Очевидно, каждый вызов с n > 1 приводит к двум дальнейшим вызовам, то есть полное число вызовов растет экспоненциально (см. рис. 3.2). Такая программа явно непрактична.

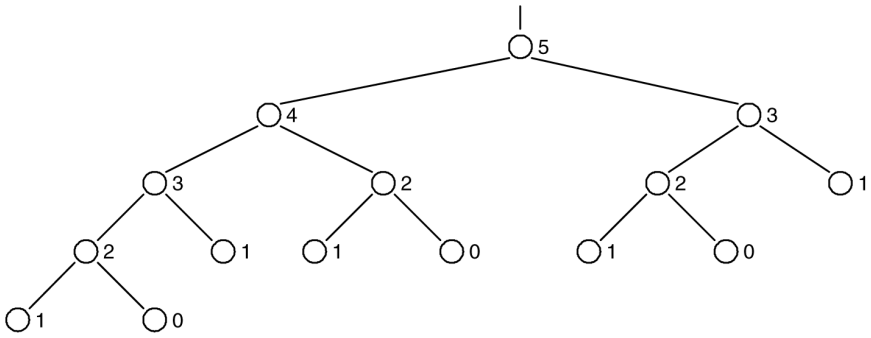


Рис. 3.2. Пятнадцать активаций при вызове Fib(5)

К счастью, числа Фибоначчи можно вычислять по итерационной схеме без многократного вычисления одних и тех же значений благодаря использованию вспомогательных переменных – таких, что $x = fib_i$ и $y = fib_{i-1}$.

```
i := 1; x := 1; y := 0;
WHILE i < n DO  z := x;  x := x + y;  y := z;  i := i + 1  END
```

Отметим, что три присваивания переменным x, y, z можно заменить всего лишь двумя присваиваниями без привлечения вспомогательной переменной z: x := x + y; y := x – y.

Отсюда мораль: следует избегать рекурсии, когда есть очевидное решение, использующее итерацию. Но это не значит, что от рекурсии нужно избавляться любой ценой. Как будет показано в последующих разделах и главах, существует много хороших применений рекурсии. Тот факт, что имеются реализации рекурсивных процедур на принципиально нерекурсивных машинах, доказывает, что любая рекурсивная программа действительно может быть преобразована в чисто итерационную. Но тогда требуется явно управлять стеком рекурсии, и это часто затемняет сущность программы до такой степени, что понять ее становится весьма трудно. Отсюда вывод: алгоритмы, которые по своей природе являются рекурсивными, а не итерационными, должны программироваться в виде рекурсивных процедур. Чтобы оценить это обстоятельство, полезно сравнить два варианта алгоритма быстрой сортировки в разделе 2.3.3: рекурсивный (QuickSort) и нерекурсивный (NonRecursiveQuickSort).

Оставшаяся часть главы посвящена разработке некоторых рекурсивных программ в ситуациях, когда применение рекурсии оправдано. Кроме того, в главе 4 рекурсия широко используется в тех случаях, когда соответствующие структуры данных делают выбор рекурсивных решений очевидным и естественным.

3.3. Два примера рекурсивных программ

Симпатичный узор на рис. 3.4 представляет собой суперпозицию пяти кривых. Эти кривые являют регулярность структуры, так что их, вероятно, можно изобразить на дисплее или графопостроителе под управлением компьютера. Наша цель – выявить рекурсивную схему, с помощью которой можно написать программу для рисования этих кривых. Можно видеть, что три из пяти кривых имеют вид, показанный на рис. 3.3; обозначим их как H_1, H_2 и H_3. Кривая H_i называется *гильбертовой кривой* порядка i в честь математика Гильберта (D. Hilbert, 1891).

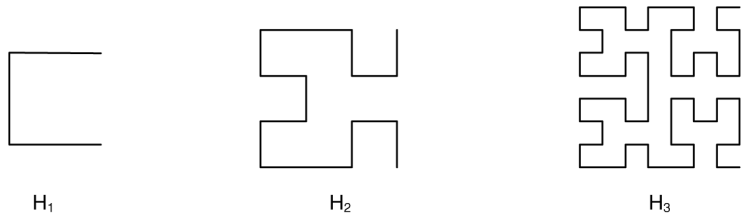


Рис. 3.3. Гильбертовы кривые порядков 1, 2 и 3

Каждая кривая H_i состоит из четырех копий кривой H_{i-1} половинного размера, поэтому мы выразим процедуру рисования H_i в виде композиции четырех вызовов для рисования H_{i-1} половинного размера и с соответствующими поворотами. Для целей иллюстрации обозначим четыре по-разному повернутых варианта базовой кривой как A, B, C и D, а шаги рисования соединительных линий обозначим стрелками, направленными соответственно. Тогда возникает следующая рекурсивная схема (ср. рис. 3.3):

A: D ← A ↓ A → B
B: C ↑ B → B ↓ A
C: B → C ↑ C ← D
D: A ↓ D ← D ↑ C

Предположим, что для рисования отрезков прямых в нашем распоряжении есть процедура line, которая передвигает чертящее перо в заданном направлении на заданное расстояние. Для удобства примем, что направление указывается целочисленным параметром i, так что в градусах оно равно 45 × i. Если длину отрезков, из которых составляется кривая, обозначить как u, то процедуру, соответствующую схеме A, можно сразу выразить через рекурсивные вызовы аналогичных процедур B и D и ее самой:

```
PROCEDURE A (i: INTEGER);
BEGIN
   IF i > 0 THEN
      D(i-1); line(4, u);
      A(i-1); line(6, u);
```

```
        A(i-1); line(0, u);
        B(i-1)
    END
END A
```

Эта процедура вызывается в главной программе один раз для каждой гильбертовой кривой, добавляемой в рисунок. Главная программа определяет начальную точку кривой, то есть начальные координаты пера, обозначенные как x0 и y0, а также длину базового отрезка u. Квадрат, в котором рисуются кривые, помещается в середине страницы с заданными шириной и высотой. Эти параметры, так же как и рисующая процедура line, берутся из модуля Draw. Отметим, что этот модуль помнит текущее положение пера.

```
DEFINITION Draw;                                        (* ADruS33_Draw *)
    CONST width = 1024; height = 800;
    PROCEDURE Clear;  (*очистить рисунок*)
    PROCEDURE SetPen(x, y: INTEGER);  (*установить перо в точку x, y*)
    PROCEDURE line(dir, len: INTEGER);
        (*начертить линию длины len в направлении dir*45 градусов;
          перо передвигается соответственно*)
END Draw.
```

Процедура Hilbert рисует гильбертовы кривые H_1 ... H_n. Она рекурсивно использует четыре процедуры A, B, C и D:

```
VAR u: INTEGER;                                         (* ADruS33_Hilbert *)

PROCEDURE A (i: INTEGER);
BEGIN
    IF i > 0 THEN
        D(i-1); Draw.line(4, u); A(i-1); Draw.line(6, u); A(i-1); Draw.line(0, u); B(i-1)
    END
END A;

PROCEDURE B (i: INTEGER);
BEGIN
    IF i > 0 THEN
        C(i-1); Draw.line(2, u); B(i-1); Draw.line(0, u); B(i-1); Draw.line(6, u); A(i-1)
    END
END B;

PROCEDURE C (i: INTEGER);
BEGIN
    IF i > 0 THEN
        B(i-1); Draw.line(0, u); C(i-1); Draw.line(2, u); C(i-1); Draw.line(4, u); D(i-1)
    END
END C;

PROCEDURE D (i: INTEGER);
BEGIN
    IF i > 0 THEN
        A(i-1); Draw.line(6, u); D(i-1); Draw.line(4, u); D(i-1); Draw.line(2, u); C(i-1)
    END
END D;
```

```
PROCEDURE Hilbert (n: INTEGER);
    CONST SquareSize = 512;
    VAR i, x0, y0: INTEGER;
BEGIN
    Draw.Clear;
    x0 := Draw.width DIV 2; y0 := Draw.height DIV 2;
    u := SquareSize; i := 0;
    REPEAT
        INC(i); u := u DIV 2;
        x0 := x0 + (u DIV 2); y0 := y0 + (u DIV 2);
        Draw.Set(x0, y0);
        A(i)
    UNTIL i = n
END Hilbert.
```

Похожий, но чуть более сложный и эстетически изощренный пример показан на рис. 3.6. Этот узор тоже получается наложением нескольких кривых, две из которых показаны на рис. 3.5. S_i называется кривой Серпиньского порядка i. Какова ее рекурсивная структура? Есть соблазн в качестве основного строительного блока взять фигуру S_1, возможно, без одного ребра. Но так решение не получится. Главное отличие кривых Серпиньского от кривых Гильберта – в том, что первые замкнуты (и не имеют самопересечений). Это означает, что базовой рекурсивной схемой должна быть разомкнутая кривая и что четыре части соединяются связками, не принадлежащими самому рекурсивному узору. В самом деле, эти связки состоят из четырех отрезков прямых в четырех самых внешних углах, показанных жирными линиями на рис. 3.5. Их можно считать принадлежащими непустой начальной кривой S_0, представляющей собой квадрат, стоящий на одном из углов. Теперь легко сформулировать рекурсивную схему. Четыре узора, из которых составляется кривая, снова обозначим как A, B, C и D, а линии-связки будем рисовать явно. Заметим, что четыре рекурсивных узора действительно идентичны, отличаясь поворотами на 90 градусов.

Вот базовая схема кривых Серпиньского:

S: A ↘ B ↙ C ↖ D ↗

А вот схема рекурсий (горизонтальные и вертикальные стрелки обозначают линии двойной длины):

A: A ↘ B → D ↗ A
B: B ↙ C ↓ A ↘ B
C: C ↖ D ← B ↙ C
D: D ↗ A ↑ C ↖ D

Если использовать те же примитивы рисования, что и в примере с кривыми Гильберта, то эта схема рекурсии легко превращается в рекурсивный алгоритм (с прямой и косвенной рекурсиями).

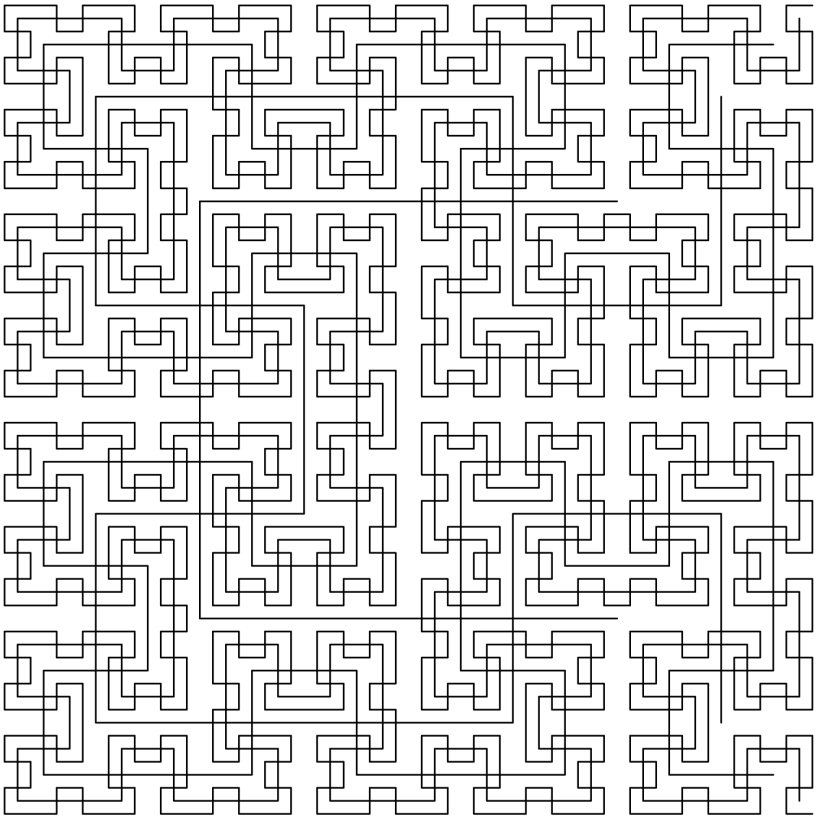


Рис. 3.4. Гильбертовы кривые H_1 ... H_5

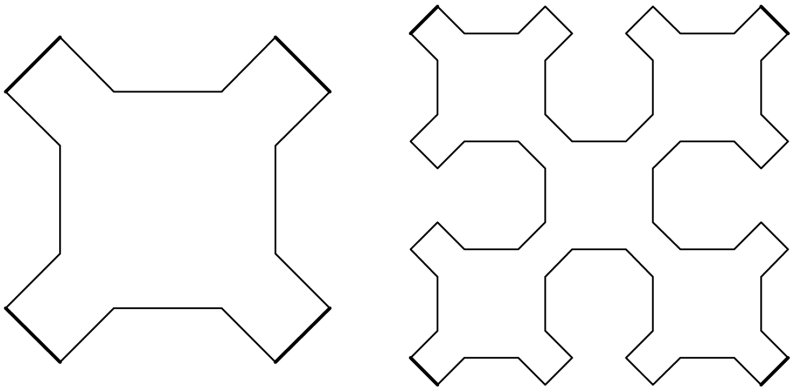


Рис. 3.5. Кривые Серпиньского S_1 и S_2

```
PROCEDURE A (k: INTEGER);
BEGIN
    IF k > 0 THEN
        A(k-1); Draw.line(7, h); B(k-1); Draw.line(0, 2*h);
        D(k-1); Draw.line(1, h); A(k-1)
    END
END A
```

Эта процедура реализует первую строку схемы рекурсий. Процедуры для узоров B, C и D получаются аналогично. Главная программа составляется по базовой схеме. Ее назначение – установить начальное положение пера и определить длину единичной линии h в соответствии с размером рисунка. Результат выполнения этой программы для n = 4 показан на рис. 3.6.

```
VAR h: INTEGER;                                    (* ADruS33_Sierpinski *)

PROCEDURE A (k: INTEGER);
BEGIN
    IF k > 0 THEN
        A(k-1); Draw.line(7, h); B(k-1); Draw.line(0, 2*h);
        D(k-1); Draw.line(1, h); A(k-1)
    END
END A;

PROCEDURE B (k: INTEGER);
BEGIN
    IF k > 0 THEN
        B(k-1); Draw.line(5, h); C(k-1); Draw.line(6, 2*h);
        A(k-1); Draw.line(7, h); B(k-1)
    END
END B;

PROCEDURE C (k: INTEGER);
BEGIN
    IF k > 0 THEN
        C(k-1); Draw.line(3, h); D(k-1); Draw.line(4, 2*h);
        B(k-1); Draw.line(5, h); C(k-1)
    END
END C;

PROCEDURE D (k: INTEGER);
BEGIN
    IF k > 0 THEN
        D(k-1); Draw.line(1, h); A(k-1); Draw.line(2, 2*h);
        C(k-1); Draw.line(3, h); D(k-1)
    END
END D;

PROCEDURE Sierpinski* (n: INTEGER);
    CONST SquareSize = 512;
    VAR i, x0, y0: INTEGER;
BEGIN
```

```
    Draw.Clear;
    h := SquareSize DIV 4;
    x0 := Draw.width DIV 2; y0 := Draw.height DIV 2 + h;
    i := 0;
    REPEAT
        INC(i); x0 := x0-h;
        h := h DIV 2; y0 := y0+h; Draw.Set(x0, y0);
        A(i); Draw.line(7,h); B(i); Draw.line(5,h);
        C(i); Draw.line(3,h); D(i); Draw.line(1,h)
    UNTIL i = n
END Sierpinski.
```

Элегантность приведенных примеров убеждает в полезности рекурсии. Правильность получившихся программ легко установить по их структуре и по схемам композиции. Более того, использование явного (и уменьшающегося) параметра уровня гарантирует остановку, так как глубина рекурсии не может превысить n. Напротив, эквивалентные программы, не использующие рекурсию явно, оказываются весьма громоздкими, и понять их нелегко. Читатель легко убедится в этом, если попытается разобраться в программах, приведенных в [3.3].

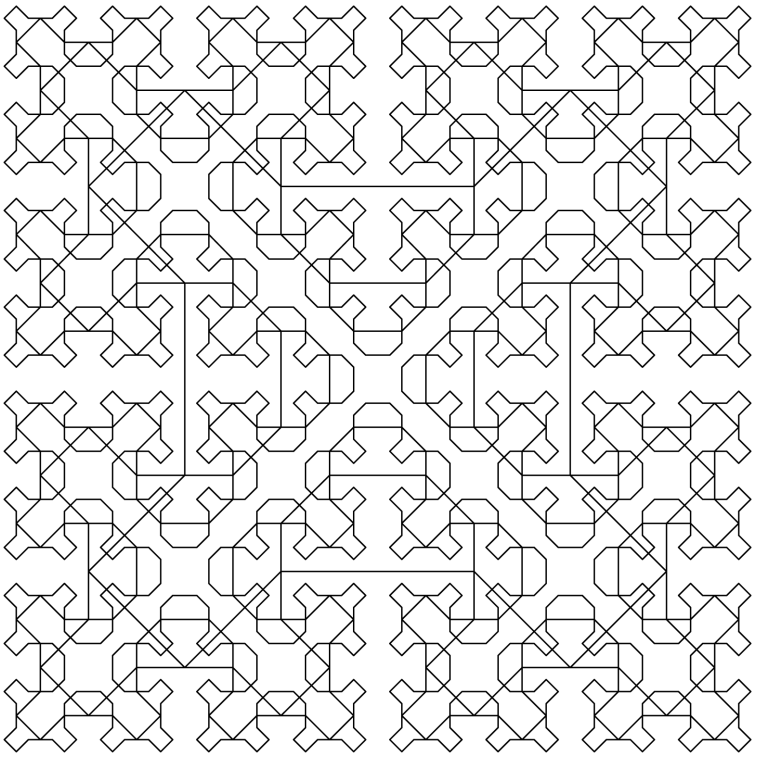


Рис. 3.6. Кривые Серпиньского S_1 ... S_4

3.4. Алгоритмы с возвратом

Весьма интригующее направление в программировании – поиск общих методов решения сложных зачач. Цель здесь в том, чтобы научиться искать решения конкретных задач, не следуя какому-то фиксированному правилу вычислений, а методом проб и ошибок. Общая схема заключается в том, чтобы свести процесс проб и ошибок к нескольким частным задачам. Эти задачи часто допускают очень естественное рекурсивное описание и сводятся к исследованию конечного числа подзадач. Процесс в целом можно представлять себе как поиск-исследование, в котором постепенно строится и просматривается (с обрезанием каких-то ветвей) некое дерево подзадач. Во многих задачах такое дерево поиска растет очень быстро, часто экспоненциально, как функция некоторого параметра. Трудоемкость поиска растет соответственно. Часто только использование эвристик позволяет обрезать дерево поиска до такой степени, чтобы сделать вычисление сколь-нибудь реалистичным.

Обсуждение общих эвристических правил не входит в наши цели. Мы сосредоточимся в этой главе на общем принципе разбиения задач на подзадачи с применением рекурсии. Начнем с демонстрации соответствующей техники в простом примере, а именно в хорошо известной *задаче о путешествии шахматного коня*.

Пусть дана доска $n \times n$ с n^2 полями. Конь, который передвигается по шахматным правилам, ставится на доске в поле <x0, y0>. Задача – обойти всю доску, если это возможно, то есть вычислить такой маршрут из n^2-1 ходов, чтобы в каждое поле доски конь попал ровно один раз.

Очевидный способ упростить задачу обхода n^2 полей – рассмотреть подзадачу, которая состоит в том, чтобы либо выполнить какой-либо очередной ход, либо обнаружить, что дальнейшие ходы невозможны. Эту идею можно выразить так:

```
PROCEDURE TryNextMove; (*попытаться сделать очередной ход*)
BEGIN
    IF доска не заполнена THEN
        инициализировать перебор допустимых ходов и выбрать первый;
        WHILE ~(ходов больше нет) & ~(обход можно завершить с этого хода)
        DO
            выбрать следующий допустимый ход
        END
    END
END TryNextMove;
```

Предикат *обход можно завершить с этого хода* удобно выразить в виде процедуры-функции с логическим значением, в которой – раз уж мы собираемся записывать порождаемую последовательность ходов – подходящее место как для записи очередного хода, так и для ее отмены в случае неудачи, так как именно в этой процедуре выясняется успех завершения обхода.

```
PROCEDURE CanBeDone ( ход ): BOOLEAN;
BEGIN
    записать ход;
```

```
    TryNextMove;
    IF не удалось достроить обход THEN
        отменить запись хода
    END;
    RETURN  удалось достроить обход
END CanBeDone
```

Здесь уже видна схема рекурсии.

Чтобы уточнить этот алгоритм, необходимо принять некоторые решения о представлении данных. Во-первых, мы хотели бы записать полную историю ходов. Поэтому каждый ход будем характеризовать тремя числами: его номером i и двумя координатами <x, y>. Эту связь можно было бы выразить, введя специальный тип записей с тремя полями, но данная задача слишком проста, чтобы оправдать соответствующие накладные расходы; будет достаточно отслеживать соответствующие тройки переменных.

Это сразу позволяет выбрать подходящие параметры для процедуры TryNextMove. Они должны позволять определить начальные условия для очередного хода, а также сообщать о его успешности. Для достижения первой цели достаточно указывать параметры предыдущего хода, то есть координаты поля x, y и его номер i. Для достижения второй цели нужен булевский параметр-результат со значением *обход завершен успешно*. Получается следующая сигнатура:

```
PROCEDURE TryNextMove (x, y, i: INTEGER; VAR done: BOOLEAN)
```

Далее, очередной допустимый ход должен иметь номер i+1. Для его координат введем пару переменных u, v. Это позволяет выразить предикат *обход можно завершить с этого хода*, используемый в цикле линейного поиска, в виде вызова процедуры-функции со следующей сигнатурой:

```
PROCEDURE CanBeDone (u, v, i1: INTEGER): BOOLEAN
```

Условие *доска не заполнена* может быть выражено как $i < n^2$. А для условия *ходов больше нет* введем логическую переменную eos. Тогда логика алгоритма проясняется следующим образом:

```
PROCEDURE TryNextMove (x, y, i: INTEGER; VAR done: BOOLEAN);
    VAR eos: BOOLEAN; u, v: INTEGER;
BEGIN
    IF i < n^2 THEN
        инициализировать перебор допустимых ходов и выбрать первый
<u, v>;
        WHILE ~eos & ~CanBeDone(u, v, i+1) DO
            выбрать следующий допустимый ход <u, v>
        END;
        done := ~eos
    ELSE
        done := TRUE
    END
END TryNextMove;
```

```
PROCEDURE CanBeDone (u, v, i1: INTEGER): BOOLEAN;
   VAR  done: BOOLEAN;
BEGIN
   записать ход;
   TryNextMove(u, v, i1, done);
   IF ~done THEN
      отменить запись хода
   END;
   RETURN done
END CanBeDone
```

Заметим, что процедура TryNextMove сформулирована так, чтобы корректно обрабатывать и вырожденный случай, когда после хода x, y, i выясняется, что доска заполнена. Это сделано по той же, в сущности, причине, по которой арифметические операции определяются так, чтобы корректно обрабатывать нулевые значения операндов: удобство и надежность. Если (как нередко делают из соображений оптимизации) вынести такую проверку из процедуры, то каждый вызов процедуры придется сопровождать такой охраной – или доказывать, что охрана в конкретной точке программы не нужна. К подобным оптимизациям следует прибегать, только если их необходимость доказана – после построения корректного алгоритма.

Следующее очевидное решение – представить доску матрицей, скажем h:

```
VAR h: ARRAY n, n OF INTEGER
```

Решение сопоставить каждому полю доски целое, а не булевское значение, которое бы просто отмечало, занято поле или нет, объясняется желанием сохранить полную историю ходов простейшим способом:

h[x, y] = 0: поле <x, y> еще не пройдено
h[x, y] = i: поле <x, y> пройдено на i-м ходу ($0 < i \le n^2$)

Очевидно, запись допустимого хода теперь выражается присваиванием $h_{xy} := i$, а отмена – $h_{xy} := 0$, чем завершается построение процедуры CanBeDone.

Осталось организовать перебор допустимых ходов u, v из заданной позиции x, y в цикле поиска процедуры TryNextMove. На бесконечной во все стороны доске для каждой позиции x, y есть несколько ходов-кандидатов u, v, которые пока конкретизировать нет нужды (см., однако, рис. 3.7). Предикат для выбора допустимых ходов среди ходов-кандидатов выражается как логическая конъюнкция условий, описывающих, что новое поле лежит в пределах доски, то есть $0 \le u < n$ и $0 \le v < n$, и что конь по нему еще не проходил, то есть $h_{uv} = 0$. Деталь, которую нельзя упустить: переменная h_{uv} существует, только если оба значения u и v лежат в диапазоне 0 ... n–1. Поэтому важно, чтобы член $h_{uv} = 0$ стоял последним. В итоге выбор следующего допустимого хода тогда представляется уже знакомой схемой линейного поиска (только выраженной через цикл repeat вместо while, что в данном случае возможно и удобно). При этом для сообщения об исчерпании множества ходов-кандидатов можно использовать переменную eos. Оформим эту операцию в виде процедуры Next, явно указав в качестве параметров значимые переменные:

```
PROCEDURE Next (VAR eos: BOOLEAN; VAR u, v: INTEGER);
BEGIN
   (*~eos*)
   REPEAT взять очередной ход-кандидат u, v
   UNTIL (кандидатов больше нет) OR
            ((0 <= u) & (u < n) & (0 <= v) & (v < n) & (h[u, v] = 0));
   eos := кандидатов больше нет
END Next;
```

Инициализация перебора ходов-кандидатов выполняется внутри аналогичной процедуры First, порождающей первый допустимый ход; см. детали в окончательной программе, приводимой ниже.

Остался только один шаг уточнения, и мы получим программу, полностью выраженную в нашей основной нотации. Заметим, что до сих пор программа разрабатывалась совершенно независимо от правил, описывающих допустимые ходы коня. Мы сознательно откладывали рассмотрение таких деталей задачи. Но теперь пора их учесть.

Для начальной пары координат x, y на бесконечной свободной доске есть восемь позиций-кандидатов u, v, куда может прыгнуть конь. На рис. 3.7 они пронумерованы от 1 до 8.

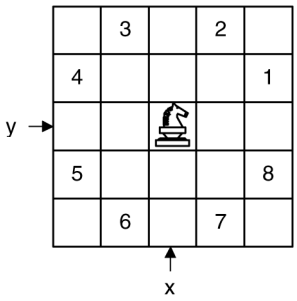


Рис. 3.7. Восемь возможных ходов коня

Простой способ получить u, v из x, y состоит в прибавлении разностей координат, хранящихся либо в массиве пар разностей, либо в двух массивах одиночных разностей. Пусть эти массивы обозначены как dx и dy и правильно инициализированы:

```
dx  =  (2, 1, -1, -2, -2, -1, 1, 2)
dy  =  (1, 2, 2, 1, -1, -2, -2, -1)
```

Тогда можно использовать индекс k для нумерации очередного хода-кандидата. Детали показаны в программе, приводимой ниже.

Мы предполагаем наличие глобальной матрицы h размера n × n, представляющей результат, константы n (и nsqr = n^2), а также массивов dx и dy, представляющих возможные ходы коня без ограничений (см. рис. 3.7). Рекурсивная процедура стартует с параметрами x0, y0 – координатами того поля, с которого должно начаться путешествие коня. В это поле должен быть записан номер 1; все прочие поля следует пометить как свободные.

```
VAR h: ARRAY n, n OF INTEGER;                    (* ADruS34_KnightsTour *)
   dx, dy: ARRAY 8 OF INTEGER;

PROCEDURE CanBeDone (u, v, i: INTEGER): BOOLEAN;
   VAR  done: BOOLEAN;
BEGIN
   h[u, v] := i;
   TryNextMove(u, v, i, done);
   IF ~done THEN h[u, v] := 0 END;
```

```
    RETURN  done
END CanBeDone;

PROCEDURE TryNextMove (x, y, i: INTEGER; VAR done: BOOLEAN);
    VAR  eos: BOOLEAN;  u, v: INTEGER;  k: INTEGER;

    PROCEDURE Next (VAR eos: BOOLEAN;  VAR u, v: INTEGER);
    BEGIN
        REPEAT
            INC(k);
            IF k < 8 THEN  u := x + dx[k]; v := y + dy[k]  END;
        UNTIL (k = 8) OR ((0 <= u) & (u < n) & (0 <= v) & (v < n) & (h[u, v] = 0));
        eos := (k = 8)
    END Next;

    PROCEDURE First (VAR eos: BOOLEAN;  VAR u, v: INTEGER);
    BEGIN
        eos := FALSE;  k := -1;  Next(eos, u, v)
    END First;

BEGIN
    IF i < nsqr THEN
        First(eos, u, v);
        WHILE ~eos & ~CanBeDone(u, v, i+1) DO
            Next(eos, u, v)
        END;
        done := ~eos
    ELSE
        done := TRUE
    END;
END TryNextMove;

PROCEDURE Clear;
    VAR i, j: INTEGER;
BEGIN
    FOR i := 0 TO n-1 DO
        FOR j := 0 TO n-1 DO h[i,j] := 0 END
    END
END Clear;

PROCEDURE KnightsTour (x0, y0: INTEGER; VAR done: BOOLEAN);
BEGIN
    Clear; h[x0,y0] := 1; TryNextMove(x0, y0, 1, done);
END KnightsTour;
```

Таблица 3.1 показывает решения, полученные для начальных позиций <2,2>, <1,3> для n = 5 и <0,0> для n = 6.

Какие общие уроки можно извлечь из этого примера? Видна ли в нем какая-либо схема, типичная для алгоритмов, решающих подобные задачи? Чему он нас учит? Характерной чертой здесь является то, что каждый шаг, выполняемый в попытке приблизиться к полному решению, запоминается таким образом, чтобы

Таблица 3.1. Три возможных обхода конем

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

23	4	9	14	25
10	15	24	1	8
5	22	3	18	13
16	11	20	7	2
21	6	17	12	19

1	16	7	26	11	14
34	25	12	15	6	
17	2	33	8	13	10
32	35	24	21	28	
23	18	3	30	9	20
36		22		4	

от него можно было позднее отказаться, если выяснится, что он не может привести к полному решению и заводит в тупик. Такое действие называется *возвратом* (backtracking). Общая схема, приводимая ниже, абстрагирована из процедуры TryNextMove в предположении, что число потенциальных кандидатов на каждом шаге конечно:

```
PROCEDURE Try; (*попытаться достроить решение*)
BEGIN
    IF решение не полно THEN
        инициализировать перебор новых шагов и выбрать первый;
        WHILE ~(шагов больше нет) & ~CanBeDone(новый шаг) DO
            выбрать следующий новый шаг
        END
    END
END Try;

PROCEDURE CanBeDone ( шаг ): BOOLEAN;
(*решение можно достроить, продолжая его шагом*)
BEGIN
    записать шаг;
    Try;
    IF не удалось достроить решение THEN
      отменить запись шага
    END;
    RETURN удалось достроить решение
END CanBeDone
```

Разумеется, в реальных программах эта схема может варьироваться. В частности, в зависимости от специфики задачи может варьироваться способ передачи информации в процедуру Try при каждом очередном ее вызове. Ведь в обсуждаемой схеме предполагается, что эта процедура имеет доступ к глобальным переменным, в которых записывается выстраиваемое решение и, следовательно, содержится, в принципе, полная информация о текущем шаге построения. Напри-

мер, в рассмотренной задаче о путешествии коня в процедуре TryNextMove нужно знать последнюю позицию коня на доске. Ее можно было бы найти поиском в массиве h. Однако эта информация явно наличествует в момент вызова процедуры, и гораздо проще ее туда передать через параметры. В дальнейших примерах мы увидим вариации на эту тему.

Отметим, что условие поиска в цикле оформлено в виде процедуры-функции CanBeDone для максимального прояснения логики алгоритма без потери обозримости программы. Разумеется, можно оптимизировать программу в других отношениях, проведя эквивалентные преобразования. Например, можно избавиться от двух процедур First и Next, слив два легко верифицируемых цикла в один. Этот единственный цикл будет, вообще говоря, более сложным, однако в том случае, когда требуется сгенерировать *все* решения, может получиться довольно прозрачный результат (см. последнюю программу в следующем разделе).

Остаток этой главы посвящен разбору еще трех примеров, в которых уместна рекурсия. В них демонстрируются разные реализации описанной общей схемы.

3.5. Задача о восьми ферзях

Задача о восьми ферзях – хорошо известный пример использования метода проб и ошибок и алгоритмов с возвратом. Ее исследовал Гаусс в 1850 г., но он не нашел полного решения. Это и неудивительно, ведь для таких задач характерно отсутствие аналитических решений. Вместо этого приходится полагаться на огромный труд, терпение и точность. Поэтому подобные алгоритмы стали применяться почти исключительно благодаря появлению автоматического компьютера, который обладает этими качествами в гораздо большей степени, чем люди и даже чем гении.

В этой задаче (см. также [3.4]) требуется расположить на шахматной доске восемь ферзей так, чтобы ни один из них не угрожал другому. Будем следовать общей схеме, представленной в конце раздела 3.4. По правилам шахмат ферзь угрожает всем фигурам, находящимся на одной с ним вертикали, горизонтали или диагонали доски, поэтому мы заключаем, что на каждой вертикали может находиться один и только один ферзь. Поэтому можно пронумеровать ферзей по занимаемым ими вертикалям, так что i-й ферзь стоит на i-й вертикали. Очередным шагом построения в общей рекурсивной схеме будем считать размещение очередного ферзя в порядке их номеров. В отличие от задачи о путешествии коня, здесь нужно будет знать положение всех уже размещенных ферзей. Поэтому в качестве параметра в процедуру Try достаточно передавать номер размещаемого на этом шаге ферзя i, который, таким образом, является номером столбца. Тогда определить положение ферзя – значит выбрать одно из восьми значений номера ряда j.

```
PROCEDURE Try (i: INTEGER);
BEGIN
    IF i < 8 THEN
        инициализировать перебор безопасных позиций j и выбрать первую;
```

```
        WHILE ~(больше нет) & ~CanBeDone(i, j) DO
            выбрать следующую безопасную позицию j
        END
    END
END Try;

PROCEDURE CanBeDone (i, j: INTEGER): BOOLEAN;
    (*решение можно достроить, поставив i-го ферзя в j-ю строку*)
BEGIN
    установить ферзя;
    Try(i+1);
    IF не удалось достроить решение THEN
        убрать ферзя
    END;
    RETURN удалось достроить решение
END CanBeDone
```

Чтобы двигаться дальше, нужно решить, как представлять данные. Напрашивается представление доски с помощью квадратной матрицы, но небольшое размышление показывает, что тогда действия по проверке безопасности позиций получатся довольно громоздкими. Это крайне нежелательно, так как это самая часто выполняемая операция. Поэтому мы должны представить данные так, чтобы эта проверка была как можно проще. Лучший путь к этой цели – как можно более непосредственно представить именно ту информацию, которая конкретно нужна и чаще всего используется. В нашем случае это не положение ферзей, а информация о том, был ли уже поставлен ферзь на каждый из рядов и на каждую из диагоналей. (Мы уже знаем, что в каждом столбце k для $0 \le k < i$ стоит в точности один ферзь.) Это приводит к такому выбору переменных:

```
VAR x: ARRAY 8 OF INTEGER;
    a: ARRAY 8 OF BOOLEAN;
    b, c: ARRAY 15 OF BOOLEAN
```

где

x_i означает положение ферзя в i-м столбце;
a_j означает, что «в j-м ряду ферзя еще нет»;
b_k означает, что «на k-й /-диагонали нет ферзя»;
c_k означает, что «на k-й \-диагонали нет ферзя».

Заметим, что все поля на /-диагонали имеют одинаковую сумму своих координат i и j, а на \-диагонали – одинаковую разность координат i-j. Соответствующая нумерация диагоналей использована в приведенной ниже программе Queens. С такими определениями операция *установить ферзя* раскрывается следующим образом:

```
x[i] := j; a[j] := FALSE; b[i+j] := FALSE; c[i-j+7] := FALSE
```

операция *убрать ферзя* уточняется в

```
a[j] := TRUE; b[i+j] := TRUE; c[i-j+7] := TRUE
```

Поле <i, j> безопасно, если оно находится в строке и на диагоналях, которые еще свободны. Поэтому ему соответствует логическое выражение

a[j] & b[i+j] & c[i-j+7]

Это позволяет построить процедуры перечисления безопасных значений j для i-го ферзя по аналогии с предыдущим примером.

Этим, в сущности, завершается разработка алгоритма, представленного целиком ниже в виде программы Queens. Она вычисляет решение x = (0, 4, 7, 5, 2, 6, 1, 3), показанное на рис. 3.8.

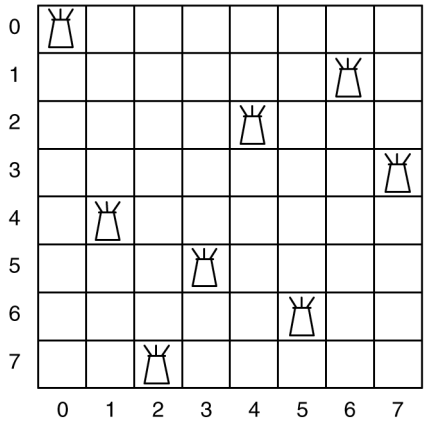


Рис. 3.8. Одно из решений задачи о восьми ферзях

```
PROCEDURE Try (i: INTEGER; VAR done: BOOLEAN);          (* ADruS35_Queens *)
   VAR  eos: BOOLEAN;  j: INTEGER;

   PROCEDURE Next;
   BEGIN
      REPEAT  INC(j);
      UNTIL (j = 8) OR (a[j] & b[i+j] & c[i-j+7]);
      eos := (j = 8)
   END Next;

   PROCEDURE First;
   BEGIN
      eos := FALSE;  j := -1;  Next
   END First;

BEGIN
   IF i < 8 THEN
      First;
      WHILE ~eos & ~CanBeDone(i, j) DO
         Next
```

```
        END;
        done := ~eos
    ELSE
        done := TRUE
    END
END Try;

PROCEDURE CanBeDone (i, j: INTEGER): BOOLEAN;
    (*решение можно достроить, поставив i-го ферзя в j-ю строку*)
    VAR done: BOOLEAN;
BEGIN
    x[i] := j; a[j] := FALSE; b[i+j] := FALSE; c[i-j+7] := FALSE;
    Try(i+1, done);
    IF ~done THEN
        x[i] := -1; a[j] := TRUE; b[i+j] := TRUE; c[i-j+7] := TRUE
    END;
    RETURN done
END CanBeDone;

PROCEDURE Queens*;
    VAR  done: BOOLEAN;  i, j: INTEGER;  (*печать в глобальный W*)
BEGIN
    FOR i := 0 TO   7 DO  a[i] := TRUE;  x[i] := -1   END;
    FOR i := 0 TO 14 DO  b[i] := TRUE;  c[i] := TRUE  END;

    Try(0, done);
    IF  done  THEN
        FOR i := 0 TO 7 DO Texts.WriteInt(W, x[ i ], 4) END;
        Texts.WriteLn(W)
    END
END Queens.
```

Прежде чем закрыть шахматную тему, покажем на примере задачи о восьми ферзях важную модификацию такого поиска методом проб и ошибок. Цель модификации – в том, чтобы найти не одно, а все решения задачи.

Выполнить такую модификацию легко. Нужно вспомнить, что кандидаты должны порождаться систематическим образом, так чтобы ни один кандидат не порождался больше одного раза. Это соответствует систематическому поиску по дереву кандидатов, при котором каждый узел проходится в точности один раз. При такой организации после нахождения и печати решения можно просто перейти к следующему кандидату, доставляемому систематическим процессом порождения. Формально модификация осуществляется переносом процедуры-функции CanBeDone из охраны цикла в его тело и подстановкой тела процедуры вместо ее вызова. При этом нужно учесть, что возвращать логические значения больше не нужно. Получается такая общая рекурсивная схема:

```
PROCEDURE Try;
BEGIN
    IF решение не полно THEN
        инициализировать перебор новых шагов и выбрать первый;
```

```
        WHILE ~(шагов больше нет) DO
            записать новый шаг;
            Try;
            отменить запись нового шага
            выбрать следующий новый шаг
        END
    ELSE
        печатать решение
    END
END Try
```

Интересно, что поиск всех возможных решений реализуется более простой программой, чем поиск единственного решения.

В задаче о восьми ферзях возможно еще более заметное упрощение. В самом деле, несколько громоздкий механизм перечисления допустимых шагов, состоящий из двух процедур First и Next, был нужен для взаимной изоляции цикла линейного поиска очередного безопасного поля (цикл по j внутри Next) и цикла линейного поиска первого j, дающего полное решение. Теперь, благодаря упрощению охраны последнего цикла, нужда в этом отпала и его можно заменить простейшим циклом по j, просто отбирая безопасные j с помощью условного оператора IF, непосредственно вложенного в цикл, без использования дополнительных процедур.

Так модифицированный алгоритм определения всех 92 решений задачи о восьми ферзях показан ниже. На самом деле есть только 12 существенно различных решений, но наша программа не распознает симметричные решения. Первые 12 порождаемых здесь решений выписаны в табл. 3.2. Колонка n справа показывает число выполнений проверки безопасности позиций. Среднее значение частоты по всем 92 решениям равно 161.

```
PROCEDURE write;                                        (* ADruS35_Queens *)
    VAR k: INTEGER;
BEGIN
    FOR k := 0 TO 7 DO Texts.WriteInt(W, x[k], 4) END;
    Texts.WriteLn(W)
END write;

PROCEDURE Try (i: INTEGER);
    VAR j: INTEGER;
BEGIN
    IF i < 8 THEN
        FOR j := 0 TO 7 DO
            IF a[j] & b[i+j] & c[i-j+7] THEN
                x[i] := j; a[j] := FALSE; b[i+j] := FALSE; c[i-j+7] := FALSE;
                Try(i + 1);
                x[i] := -1; a[j] := TRUE; b[i+j] := TRUE; c[i-j+7] := TRUE
            END
        END
    ELSE
```

```
        write;
        m := m+1 (*подсчет решений*)
    END
END Try;

PROCEDURE AllQueens*;
    VAR  i, j: INTEGER;
BEGIN
    FOR i := 0 TO 7   DO a[i] := TRUE; x[i] := -1  END;
    FOR i := 0 TO 14 DO b[i] := TRUE; c[i] := TRUE  END;
    m := 0;
    Try(0);
    Log.String('всего решений: '); Log.Int(m); Log.Ln
END AllQueens.
```

Таблица 3.2. Двенадцать решений задачи о восьми ферзях

x0	x1	x2	x3	x4	x5	x6	x7	n
0	4	7	5	2	6	1	3	876
0	5	7	2	6	3	1	4	264
0	6	3	5	7	1	4	2	200
0	6	4	7	1	3	5	2	136
1	3	5	7	2	0	6	4	504
1	4	6	0	2	7	5	3	400
1	4	6	3	0	7	5	2	072
1	5	0	6	3	7	2	4	280
1	5	7	2	0	3	6	4	240
1	6	2	5	7	4	0	3	264
1	6	4	7	0	3	5	2	160
1	7	5	0	2	4	6	3	336

3.6. Задача о стабильных браках

Предположим, что даны два непересекающихся множества A и B равного размера n. Требуется найти набор n пар <a, b> – таких, что a из A и b из B удовлетворяют некоторым ограничениям. Может быть много разных критериев для таких пар; один из них называется *правилом стабильных браков*.

Примем, что A – это множество мужчин, а B – множество женщин. Каждый мужчина и каждая женщина указали предпочтительных для себя партнеров. Если n пар выбраны так, что существуют мужчина и женщина, которые не являются мужем и женой, но которые предпочли бы друг друга своим фактическим супругам, то такое распределение по парам называется нестабильным. Если таких пар нет, то распределение *стабильно*. Подобная ситуация характерна для многих похожих задач, в которых нужно сделать распределение с учетом предпочтений, на-

пример выбор университета студентами, выбор новобранцев различными родами войск и т. п. Пример с браками особенно интуитивен; однако следует заметить, что список предпочтений остается неизменным и после того, как сделано распределение по парам. Такое предположение упрощает задачу, но представляет собой опасное искажение реальности (это называют абстракцией).

Возможное направление поиска решения – пытаться распределить по парам членов двух множеств одного за другим, пока не будут исчерпаны оба множества. Имея целью найти все стабильные распределения, мы можем сразу сделать набросок решения, взяв за образец схему программы AllQueens. Пусть Try(m) означает алгоритм поиска жены для мужчины m, и пусть этот поиск происходит в соответствии с порядком списка предпочтений, заявленных этим мужчиной. Первая версия, основанная на этих предположениях, такова:

```
PROCEDURE Try (m: man);
    VAR r: rank;
BEGIN
    IF m < n THEN
        FOR r := 0 TO n-1 DO
            взять r-ю кандидатуру из списка мужчины m;
            IF допустима THEN
                записать супружескую пару;
                Try(следующий за m);
                отменить этот брак
            END
        END
    ELSE
        записать стабильное решение
    END
END Try
```

Исходные данные представлены двумя матрицами, указывающими предпочтения мужчин и женщин:

```
VAR  wmr: ARRAY n, n OF woman;
     mwr: ARRAY n, n OF man
```

Соответственно, wmr_m обозначает список предпочтений мужчины m, то есть $wmr_{m,r}$ – это женщина, находящаяся в этом списке на r-м месте. Аналогично, mwr_w – список предпочтений женщины w, а $mwr_{w,r}$ – мужчина на r-м месте в этом списке.

Пример набора данных показан в табл. 3.3.

Результат представим массивом женщин x, так что x_m обозначает супругу мужчины m. Чтобы сохранить симметрию между мужчинами и женщинами, вводится дополнительный массив y, так что y_w обозначает супруга женщины w:

```
VAR x, y: ARRAY n OF INTEGER
```

На самом деле массив y избыточен, так как в нем представлена информация, уже содержащаяся в x. Действительно, соотношения

```
x[y[w]] = w,  y[x[m]] = m
```

Таблица 3.3. Пример входных данных для wmr и mwr

	r = 0	1	2	3	4	5	6	7
m = 0	6	1	5	4	0	2	7	3
1	3	2	1	5	7	0	6	4
2	2	1	3	0	7	4	6	5
3	2	7	3	1	4	5	6	0
4	7	2	3	4	5	0	6	1
5	7	6	4	1	3	2	0	5
6	1	3	5	2	0	6	4	7
7	5	0	3	1	6	4	2	7

	r = 0	1	2	3	4	5	6	7
w = 0	3	5	1	4	7	0	2	6
1	7	4	2	0	5	6	3	1
2	5	7	0	1	2	3	6	4
3	2	1	3	6	5	7	4	0
4	5	2	0	3	4	6	1	7
5	1	0	2	7	6	3	5	4
6	2	4	6	1	3	0	7	5
7	6	1	7	3	4	5	2	0

выполняются для всех m и w, которые состоят в браке. Поэтому значение y_w можно было бы определить простым поиском в x. Однако ясно, что использование массива y повысит эффективность алгоритма. Информация, содержащаяся в массивах x и y, нужна для определения стабильности предполагаемого множества браков. Поскольку это множество строится шаг за шагом посредством соединения индивидов в пары и проверки стабильности после каждого преполагаемого брака, массивы x и y нужны даже еще до того, как будут определены все их компоненты. Чтобы отслеживать, какие компоненты уже определены, можно ввести булевские массивы

```
singlem, singlew: ARRAY n OF BOOLEAN
```

со следующими значениями: истинность $singlem_m$ означает, что значение x_m еще не определено, а $singlew_w$ – что не определено y_w. Однако, присмотревшись к обсуждаемому алгоритму, мы легко обнаружим, что семейное положение мужчины k определяется значением m с помощью отношения

```
~singlem[k]  =  k < m
```

Это наводит на мысль, что можно отказаться от массива singlem; соответственно, имя singlew упростим до single. Эти соглашения приводят к уточнению, показанному в следующей процедуре Try. Предикат *допустима* можно уточнить в конъюнкцию операндов single и *брак стабилен*, где предикат *брак стабилен* еще предстоит определить:

```
PROCEDURE Try (m: man);
    VAR r: rank; w: woman;
BEGIN
    IF m < n THEN
        FOR r := 0 TO n-1 DO
            w := wmr[m,r];
            IF single[w] & брак стабилен THEN
                x[m] := w; y[w] := m; single[w] := FALSE;
                Try(m+1);
```

```
                single[w] := TRUE
            END
        END
    ELSE
        записать решение
    END
END Try
```

У этого решения все еще заметно сильное сходство с процедурой AllQueens. Ключевая задача теперь – уточнить алгоритм определения стабильности. К несчастью, свойство стабильности невозможно выразить так же просто, как при проверке безопасности позиции ферзя. Первая особенность, о которой нужно помнить, состоит в том, что, по определению, стабильность следует из сравнений рангов (то есть позиций в списках предпочтений). Однако нигде в нашей коллекции данных, определенных до сих пор, нет непосредственно доступных рангов мужчин или женщин. Разумеется, ранг женщины w во мнении мужчины m вычислить можно, но только с помощью дорогостоящего поиска значения w в wmr_m. Поскольку вычисление стабильности – очень частая операция, полезно обеспечить более прямой доступ к этой информации. С этой целью введем две матрицы:

```
rmw: ARRAY man, woman OF rank;
rwm: ARRAY woman, man OF rank
```

При этом $rmw_{m,w}$ обозначает ранг женщины w в списке предпочтений мужчины m, а $rwm_{w,m}$ – ранг мужчины m в аналогичном списке женщины w. Значения этих вспомогательных массивов не меняются и могут быть определены в самом начале по значениям массивов wmr и mwr.

Теперь можно вычислить предикат *брак стабилен*, точно следуя его исходному определению. Напомним, что мы проверяем возможность соединить браком m и w, где $w = wmr_{m,r}$, то есть w является кандидатурой ранга r для мужчины m. Проявляя оптимизм, мы сначала предположим, что стабильность имеет место, а потом попытаемся обнаружить возможные помехи. Где они могут быть скрыты? Есть две симметричные возможности:

1) может найтись женщина pw с рангом, более высоким, чем у w, по мнению m, и которая сама предпочитает m своему мужу;
2) может найтись мужчина pm с рангом, более высоким, чем у m, по мнению w, и который сам предпочитает w своей жене.

Чтобы обнаружить помеху первого рода, сравним ранги $rwm_{pw,m}$ и $rwm_{pw,y[pw]}$ для всех женщин, которых m предпочитает w, то есть для всех $pw = wmr_{m,i}$ таких, что $i < r$. На самом деле все эти женщины pw уже замужем, так как, будь любая из них еще не замужем, m выбрал бы ее еще раньше. Описанный процесс можно сформулировать в виде линейного поиска; имя переменной S является сокращением для Stability (стабильность).

```
i := -1; S := TRUE;
REPEAT
    INC(i);
```

```
    IF i < r THEN
        pw := wmr[m,i];
        IF ~single[pw] THEN S := rwm[pw,m] > rwm[pw, y[pw]] END
    END
UNTIL (i = r) OR ~S
```

Чтобы обнаружить помеху второго рода, нужно проверить всех кандидатов pm, которых w предпочитает своей текущей паре m, то есть всех мужчин $pm = mwr_{w,i}$ с $i < rwm_{w,m}$. По аналогии с первым случаем нужно сравнить ранги $rmwp_{m,w}$ и $rmw_{pm,x[pm]}$. Однако нужно не забыть пропустить сравнения с теми x_{pm}, где pm еще не женат. Это обеспечивается проверкой pm < m, так как мы знаем, что все мужчины до m уже женаты.

Полная программа показана ниже. Таблица 3.4 показывает девять стабильных решений, найденных для входных данных wmr и mwr, представленных в табл. 3.3.

```
PROCEDURE write;                                    (* ADruS36_Marriages *)
    (*для вывода используется глобальный объект записи W*)
    VAR m: man; rm, rw: INTEGER;
BEGIN
    rm := 0; rw := 0;
    FOR m := 0 TO n-1 DO
        Texts.WriteInt(W, x[m], 4);
        rm := rmw[m, x[m]] + rm; rw := rwm[x[m], m] + rw
    END;
    Texts.WriteInt(W, rm, 8); Texts.WriteInt(W, rw, 4); Texts.WriteLn(W)
END write;

PROCEDURE stable (m, w, r: INTEGER): BOOLEAN; (*брак стабилен*)
    VAR pm, pw, rank, i, lim: INTEGER;
        S: BOOLEAN;
BEGIN
    i := -1; S := TRUE;
    REPEAT
        INC(i);
        IF i < r THEN
            pw := wmr[m,i];
            IF ~single[pw] THEN S := rwm[pw,m] > rwm[pw, y[pw]] END
        END
    UNTIL (i = r) OR ~S;

    i := -1;  lim := rwm[w,m];
    REPEAT
        INC(i);
        IF i < lim THEN
            pm := mwr[w,i];
            IF pm < m THEN S := rmw[pm,w] > rmw[pm, x[pm]] END
        END
    UNTIL (i = lim) OR ~S;

    RETURN S
END stable;
```

```
PROCEDURE Try (m: INTEGER);
    VAR w, r: INTEGER;
BEGIN
    IF m < n THEN
        FOR r := 0 TO n-1 DO
            w := wmr[m,r];
            IF single[w] & stable(m,w,r) THEN
                x[m] := w; y[w] := m; single[w] := FALSE;
                Try(m+1);
                single[w] := TRUE
            END
        END
    ELSE
        write
    END
END Try;

PROCEDURE FindStableMarriages (VAR S: Texts.Scanner);
    VAR m, w, r: INTEGER;
BEGIN
    FOR m := 0 TO n-1 DO
        FOR r := 0 TO n-1 DO
            Texts.Scan(S); wmr[m,r] := S.i; rmw[m, wmr[m,r]] := r
        END
    END;

    FOR w := 0 TO n-1 DO
        single[w] := TRUE;
        FOR r := 0 TO n-1 DO
            Texts.Scan(S); mwr[w,r] := S.i; rwm[w, mwr[w,r]] := r
        END
    END;

    Try(0)
END FindStableMarriages
```

Этот алгоритм прямолинейно реализует обход с возвратом. Его эффективность зависит главным образом от изощренности схемы усечения дерева решений. Несколько более быстрый, но более сложный и менее прозрачный алгоритм дали Маквити и Уилсон [3.1] и [3.2], и они также распространили его на случай множеств (мужчин и женщин) разного размера.

Алгоритмы, подобные последним двум примерам, которые порождают все возможные решения задачи (при определенных ограничениях), часто используют для выбора одного или нескольких решений, которые в каком-то смысле оптимальны. Например, в данном примере можно было бы искать решение, которое в среднем лучше удовлетворяет мужчин или женщин или вообще всех.

Заметим, что в табл. 3.4 указаны суммы рангов всех женщин в списках предпочтений их мужей, а также суммы рангов всех мужчин в списках предпочтений их жен. Это величины

rm = **S**m: 0 ≤ m < n: $rmw_{m,x[m]}$
rw = **S**m: 0 ≤ m < n: $rwm_{x[m],m}$

Таблица 3.4. Решение задачи о стабильных браках

	x0	x1	x2	x3	x4	x5	x6	x7	rm	rw	c
0	6	3	2	7	0	4	1	5	8	24	21
1	1	3	2	7	0	4	6	5	14	19	449
2	1	3	2	0	6	4	7	5	23	12	59
3	5	3	2	7	0	4	6	1	18	14	62
4	5	3	2	0	6	4	7	1	27	7	47
5	5	2	3	7	0	4	6	1	21	12	143
6	5	2	3	0	6	4	7	1	30	5	47
7	2	5	3	7	0	4	6	1	26	10	758
8	2	5	3	0	6	4	7	1	35	3	34

c = сколько раз вычислялся предикат *брак стабилен* (процедуры stable).
Решение 0 оптимально для мужчин; решение 8 – для женщин.

Решение с наименьшим значением rm назовем стабильным решением, оптимальным для мужчин; решение с наименьшим rw – оптимальным для женщин. Характер принятой стратегии поиска таков, что сначала генерируются решения, хорошие с точки зрения мужчин, а решения, хорошие с точки зрения женщин, – в конце. В этом смысле алгоритм выгоден мужчинам. Это легко исправить путем систематической перестановки ролей мужчин и женщин, то есть просто меняя местами mwr и wmr, а также rmw и rwm.

Мы не будем дальше развивать эту программу, а задачу включения в программу поиска оптимального решения оставим для следующего и последнего примера применения алгоритма обхода с возвратом.

3.7. Задача оптимального выбора

Наш последний пример алгоритма поиска с возвратом является логическим развитием предыдущих двух в рамках общей схемы. Сначала мы применили принцип возврата, чтобы находить *одно* решение задачи. Примером послужили задачи о путешествии шахматного коня и о восьми ферзях. Затем мы разобрались с поиском *всех* решений; примерами послужили задачи о восьми ферзях и о стабильных браках. Теперь мы хотим искать *оптимальное* решение.

Для этого нужно генерировать все возможные решения, но выбрать лишь то, которое оптимально в каком-то конкретном смысле. Предполагая, что оптимальность определена с помощью функции f(s), принимающей положительные значения, получаем нужный алгоритм из общей схемы Try заменой операции *печатать решение* инструкцией

```
IF f(solution) > f(optimum) THEN optimum := solution END
```

Переменная optimum запоминает лучшее решение из до сих пор найденных. Естественно, ее нужно правильно инициализировать; кроме того, обычно значение f(optimum) хранят еще в одной переменной, чтобы избежать повторных вычислений.

Вот частный пример общей проблемы нахождения оптимального решения в некоторой задаче. Рассмотрим важную и часто встречающуюся проблему выбора оптимального набора (подмножества) из заданного множества объектов при наличии некоторых ограничений. Наборы, являющиеся допустимыми решениями, собираются постепенно посредством исследования отдельных объектов исходного множества. Процедура Try описывает процесс исследования одного объекта, и она вызывается рекурсивно (чтобы исследовать очередной объект) до тех пор, пока не будут исследованы все объекты.

Замечаем, что рассмотрение каждого объекта (такие объекты назывались кандидатами в предыдущих примерах) имеет два возможных исхода, а именно: либо исследуемый объект включается в собираемый набор, либо исключается из него. Поэтому использовать циклы repeat или for здесь неудобно, и вместо них можно просто явно описать два случая. Предполагая, что объекты пронумерованы 0, 1, ... , n–1, это можно выразить следующим образом:

```
PROCEDURE Try (i: INTEGER);
BEGIN
    IF i < n THEN
        IF включение допустимо THEN
            включить i-й объект;
            Try(i+1);
            исключить i-й объект
        END;
        IF исключение допустимо THEN
            Try(i+1)
        END
    ELSE
        проверить оптимальность
    END
END Try
```

Уже из этой схемы очевидно, что есть 2^n возможных подмножеств; ясно, что нужны подходящие критерии отбора, чтобы радикально уменьшить число исследуемых кандидатов. Чтобы прояснить этот процесс, возьмем конкретный пример задачи выбора: пусть каждый из n объектов $a_0, \ldots, a_{n-1}$ характеризуется своим весом и ценностью. Пусть оптимальным считается тот набор, у которого суммарная ценность компонент является наибольшей, а ограничением пусть будет некоторый предел на их суммарный вес. Эта задача хорошо известна всем путешественникам, которые пакуют чемоданы, делая выбор из n предметов таким образом, чтобы их суммарная ценность была наибольшей, а суммарный вес не превышал некоторого предела.

Теперь можно принять решения о представлении описанных сведений в глобальных переменных. На основе приведенных соображений сделать выбор легко:

```
TYPE Object = RECORD weight, value: INTEGER END;
VAR a: ARRAY n OF Object;
    limw, totv, maxv: INTEGER;
    s, opts: SET
```

Переменные limw и totv обозначают предел для веса и суммарную ценность всех n объектов. Эти два значения постоянны на протяжении всего процесса выбора. Переменная s представляет текущее состояние собираемого набора объектов, в котором каждый объект представлен своим именем (индексом). Переменная opts – оптимальный набор среди исследованных к данному моменту, а maxv – его ценность.

Каковы критерии допустимости включения объекта в собираемый набор? Если речь о том, имеет ли смысл *включать* объект в набор, то критерий здесь – не будет ли при таком включении превышен лимит по весу. Если будет, то можно не добавлять новые объекты к текущему набору. Однако если речь об *исключении*, то допустимость дальнейшего исследования наборов, не содержащих этого элемента, определяется тем, может ли ценность таких наборов превысить значение для оптимума, найденного к данному моменту. И если не может, то продолжение поиска, хотя и может дать еще какое-нибудь решение, не приведет к улучшению уже найденного оптимума. Поэтому дальнейший поиск на этом пути бесполезен. Из этих двух условий можно определить величины, которые нужно вычислять на каждом шаге процесса выбора:

1. Полный вес tw набора s, собранного на данный момент.
2. Еще достижимая с набором s ценность av.

Эти два значения удобно представить параметрами процедуры Try. Теперь условие *включение допустимо* можно сформулирловать так:

```
tw + a[i].weight < limw
```

а последующую проверку оптимальности записать так:

```
IF av > maxv THEN (*новый оптимум, записать его*)
    opts := s; maxv := av
END
```

Последнее присваивание основано на том соображении, что когда все n объектов рассмотрены, достижимое значение совпадает с достигнутым. Условие *исключение допустимо* выражается так:

```
av - a[i].value > maxv
```

Для значения av - a[i].value, которое используется неоднократно, вводится имя av1, чтобы избежать его повторного вычисления.

Теперь вся процедура составляется из уже рассмотренных частей с добавлением подходящих операторов инициализации для глобальных переменных. Обратим внимание на легкость включения и исключения из множества s с помощью операций для типа SET. Результаты работы программы показаны в табл. 3.5.

```
TYPE Object = RECORD value, weight: INTEGER END; (* ADruS37_OptSelection *)
VAR a: ARRAY n OF Object;
    limw, totv, maxv: INTEGER;
    s, opts: SET;

PROCEDURE Try (i, tw, av: INTEGER);
    VAR tw1, av1: INTEGER;
BEGIN
    IF i < n THEN
        (*проверка включения*)
        tw1 := tw + a[i].weight;
        IF tw1 <= limw THEN
            s := s + {i};
            Try(i+1, tw1, av);
            s := s - {i}
        END;
        (*проверка исключения*)
        av1 := av - a[i].value;
        IF av1 > maxv THEN
            Try(i+1, tw, av1)
        END
    ELSIF av > maxv THEN
        maxv := av; opts := s
    END
END Try;
```

Таблица 3.5. Пример результатов работы программы Selection при выборе из 10 объектов (вверху). Звездочки отмечают объекты из отпимальных наборов opts для ограничений на суммарный вес от 10 до 120

вес:	10	11	12	13	14	15	16	17	18	19	
ценность:	18	20	17	19	25	21	27	23	25	24	
limw ↓											maxv
10	*										18
20							*				27
30					*		*				52
40	*				*		*				70
50	*	*		*			*				84
60	*	*	*	*	*						99
70	*	*			*		*		*		115
80	*	*	*		*		*	*			130
90	*	*			*		*		*	*	139
100	*	*		*	*		*	*	*		157
110	*	*	*	*	*	*	*		*		172
120	*	*			*	*	*	*	*	*	183

```
PROCEDURE Selection (WeightInc, WeightLimit: INTEGER);
BEGIN
    limw := 0;
    REPEAT
        limw := limw + WeightInc; maxv := 0;
        s := {}; opts := {}; Try(0, 0, totv);
    UNTIL limw >= WeightLimit
END Selection.
```

Такая схема поиска с возвратом, в которой используются ограничения для предотвращения избыточных блужданий по дереву поиска, называется *методом ветвей и границ* (branch and bound algorithm).

Упражнения

3.1. (Ханойские башни.) Даны три стержня и n дисков разных размеров. Диски могут быть нанизаны на стержни, образуя башни. Пусть n дисков первоначально находятся на стержне A в порядке убывания размера, как показано на рис. 3.9 для n = 3. Задание в том, чтобы переместить n дисков со стержня A на стержень C, причем так, чтобы они оказались нанизаны в том же порядке. Этого нужно добиться при следующих ограничениях:

1. На каждом шаге со стержня на стержень перемещается только один диск.
2. Диск нельзя нанизывать поверх диска меньшего размера.
3. Стержень B можно использовать в качестве вспомогательного хранилища.

Требуется найти алгоритм выполнения этого задания. Заметим, что башню удобно рассматривать как состоящую из одного диска на вершине и башни, составленной из остальных дисков. Опишите алгоритм в виде рекурсивной программы.

3.2. Напишите процедуру порождения всех n! перестановок n элементов $a_0, \ldots, a_{n-1}$ *in situ*, то есть без использования другого массива. После порождения очередной перестановки должна вызываться передаваемая в качестве параметра процедура Q, которая может, например, печатать порожденную перестановку.

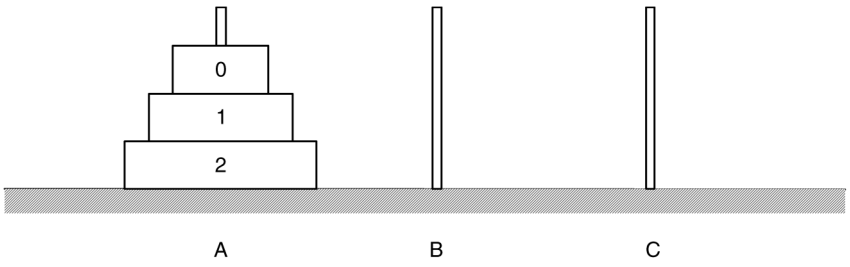


Рис. 3.9. Ханойские башни

Подсказка. Считайте, что задача порождения всех перестановок элементов $a_0, \ldots, a_{m-1}$ состоит из m подзадач порождения всех перестановок элементов $a_0, \ldots, a_{m-2}$, после которых стоит a_{m-1}, где в i-й подзадаче предварительно были переставлены два элемента a_i и a_{m-1}.

3.3. Найдите рекурсивную схему для рис. 3.10, который представляет собой суперпозицию четырех кривых W_1, W_2, W_3, W_4. Эта структура подобна кривым Серпиньского (рис. 3.6). Из рекурсивной схемы получите рекурсивную программу для рисования этих кривых.

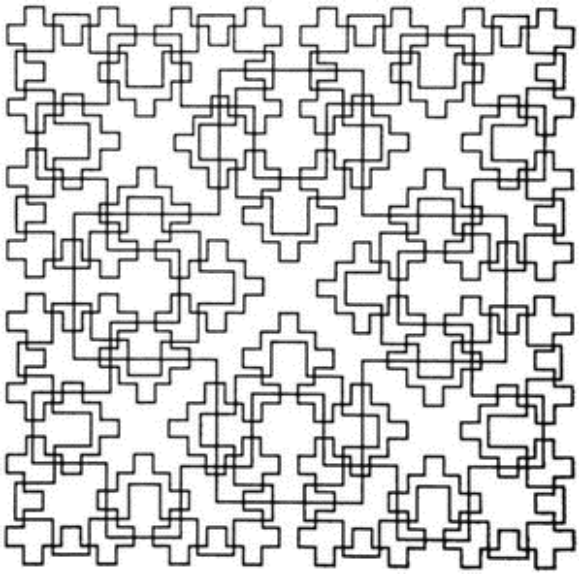


Рис. 3.10. Кривые $W_1 - W_4$

3.4. Из 92 решений, вычисляемых программой AllQueens в задаче о восьми ферзях, только 12 являются существенно различными. Остальные получаются отражениями относительно осей или центральной точки. Придумайте программу, которая определяет 12 основных решений. Например, обратите внимание, что поиск в столбце 1 можно ограничить позициями 1–4.

3.5. Измените программу для задачи о стабильных браках так, чтобы она находила оптимальное решение (для мужчин или женщин). Получится пример применения метода ветвей и границ, уже реализованного в задаче об оптимальном выборе (программа Selection).

3.6. Железнодорожная компания обслуживает n станций $S_0, \ldots, S_{n-1}$. В ее планах – улучшить обслуживание пассажиров с помощью компьютеризованных информационных терминалов. Предполагается, что пассажир указывает свои станции отправления SA и назначения SD и (немедленно) получает расписа-

ние маршрута с пересадками и с минимальным полным временем поездки. Напишите программу для вычисления такой информации. Предположите, что график движения поездов (банк данных для этой задачи) задан в подходящей структуре данных, содержащей времена отправления (= прибытия) всех поездов. Естественно, не все станции соединены друг с другом прямыми маршрутами (см. также упр. 1.6).

3.7. Функция Аккермана A определяется для всех неотрицательных целых аргументов m и n следующим образом:

```
A(0, n)  = n + 1
A(m, 0) = A(m-1, 1)  (m > 0)
A(m, n) = A(m-1, A(m, n-1))   (m, n > 0)
```

Напишите программу для вычисления A(m,n), не используя рекурсию. В качестве образца используйте нерекурсивную версию быстрой сортировки (программа NonRecursiveQuickSort). Сформулируйте общие правила для преобразования рекурсивных программ в итеративные.

Литература

[3.1] McVitie D. G. and Wilson L. B. The Stable Marriage Problem. Comm. ACM, 14, No. 7 (1971), 486–492.

[3.2] McVitie D. G. and Wilson L. B. Stable Marriage Assignment for Unequal Sets. Bit, 10, (1970), 295–309.

[3.3] Space Filling Curves, or How to Waste Time on a Plotter. Software – Practice and Experience, 1, No. 4 (1971), 403–440.

[3.4] Wirth N. Program Development by Stepwise Refinement. Comm. ACM, 14, No. 4 (1971), 221–227.

Глава 4

Динамические структуры данных

4.1. Рекурсивные типы данных

В главе 1 массивы, записи и множества были введены в качестве фундаментальных структур данных. Мы назвали их фундаментальными, так как они являются строительными блоками, из которых формируются более сложные структуры, а также потому, что на практике они встречаются чаще всего. Смысл определения типа данных, а затем определения переменных, имеющих этот тип, состоит в том, чтобы раз и навсегда фиксировать диапазон значений этих переменных, а значит, и способ их размещения в памяти. Поэтому такие переменные называют *статическими*. Однако есть много задач, где нужны более сложные структуры данных. Для таких задач характерно, что не только значения, но и структура переменных меняется во время вычисления. Поэтому их называют *динамическими структурами*. Естественно, компоненты таких структур – на определенном уровне разрешения – являются статическими, то есть принадлежат одному из фундаментальных типов данных. Эта глава посвящена построению, анализу и работе с динамическими структурами данных.

Надо заметить, что существуют близкие аналогии между методами структурирования алгоритмов и данных. Эта аналогия, как и любая другая, не является полной, тем не менее сравнение методов структурирования программ и данных поучительно.

Элементарный неделимый оператор – присваивание значения некоторой переменной. Соответствующий член семейства структур данных – скалярный, неструктурированный тип. Эта пара представляет собой неделимые строительные блоки для составных операторов и для типов данных. Простейшие структуры, получаемые посредством перечисления, суть последовательность операторов и запись. И та, и другая состоят из конечного (обычно небольшого) числа явно перечисленных компонент, которые все могут быть различными. Если все компоненты идентичны, то их не обязательно выписывать по отдельности: в этом случае используют оператор for и массив, чтобы указать известное, конечное число повторений. Выбор между двумя или более элементами выражается условным оператором и расширением записевых типов соответственно. И наконец, повторение с заранее неизвестным (и потенциально бесконечным) числом шагов выражается операторами while и repeat. Соответствующая структура данных – последовательность (файл) – это простейшее средство для построения типов с бесконечной мощностью.

Возникает вопрос: существует ли структура данных, которая аналогичным образом соответствовала бы оператору процедуры? Естественно, в этом отношении самым интересным и новым свойством процедур является рекурсия. Значения такого рекурсивного типа данных должны содержать одну или более компонент, принадлежащих этому же типу, подобно тому как процедура может содержать один или более вызовов самой себя. Как и процедуры, определения типов данных могли бы быть явно или косвенно рекурсивными.

Простой пример объекта, который весьма уместно представлять рекурсивно определенным типом, – арифметическое выражение, имеющееся в языках про-

граммирования. Рекурсия используется, чтобы отразить возможность вложений, то есть использования подвыражений в скобках в качестве операндов выражений. Поэтому дадим следующее неформальное определение выражения:

Выражение состоит из терма, за которым следует знак операции, за которым следует терм. (Два этих терма – операнды операции.) Терм – это либо переменная, представленная идентификатором, либо выражение, заключенное в скобки.

Тип данных, значениями которого представляются такие выражения, может быть легко описан, если использовать уже имеющиеся средства, добавив к ним рекурсию:

```
TYPE expression = RECORD op: INTEGER;
                    opd1, opd2: term
                  END
TYPE term =       RECORD
                    IF t: BOOLEAN THEN id: Name ELSE subex: expression END
                  END
```

Поэтому каждая переменная типа term состоит из двух компонент, а именно поля признака t, а также, если t истинно, поля id, или в противном случае поля subex. Например, рассмотрим следующие четыре выражения:

1. x + y
2. x – (y * z)
3. (x + y) * (z – w)
4. (x/(y + z)) * w

Эти выражения схематически показаны на рис. 4.1, где видна их «матрешечная», рекурсивная структура, а также показано размещение этих выражений в памяти.

Второй пример рекурсивной структуры данных – семейная родословная. Пусть родословная определена именем индивида и двумя родословными его родителей. Это определение неизбежно приводит к бесконечной структуре. Реальные родословные ограничены, так как о достаточно далеких предках информация отсутствует. Снова предположим, что это можно учесть с помощью некоторой условной структуры (ped от pedigree – родословная):

```
TYPE ped = RECORD
             IF known: BOOLEAN THEN name: Name; father, mother: ped  END
           END
```

Заметим, что каждая переменная типа ped имеет по крайней мере одну компоненту, а именно поле признака known (известен). Если его значение равно TRUE, то есть еще три поля; в противном случае эти поля отсутствуют. Пример конкретного значения показан ниже в виде выражения с вложениями, а также с помощью диаграммы, показывающей возможное размещение в памяти (см. рис. 4.2).

```
(T, Ted, (T, Fred, (T, Adam, (F), (F)), (F)), (T, Mary, (F), (T, Eva, (F), (F)))
```

Понятно, почему важны условия в таких определениях: это единственное средство ограничить рекурсивную структуру данных, поэтому они обязательно

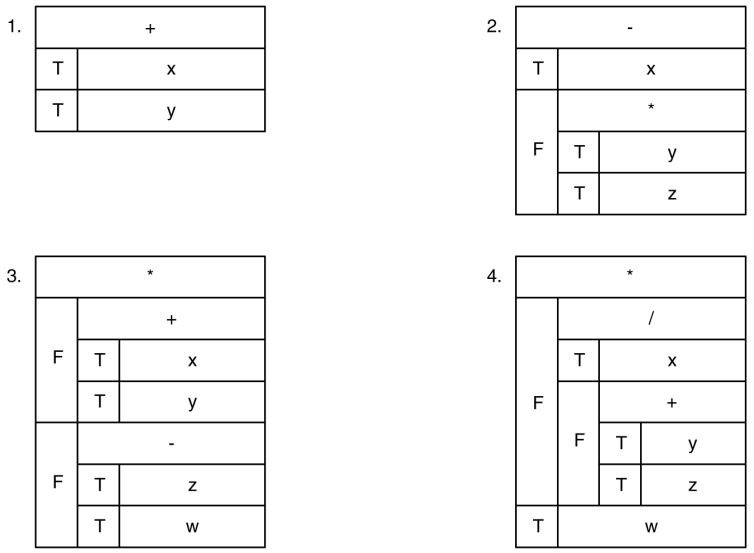


Рис. 4.1. Схемы расположения в памяти рекурсивных записевых структур

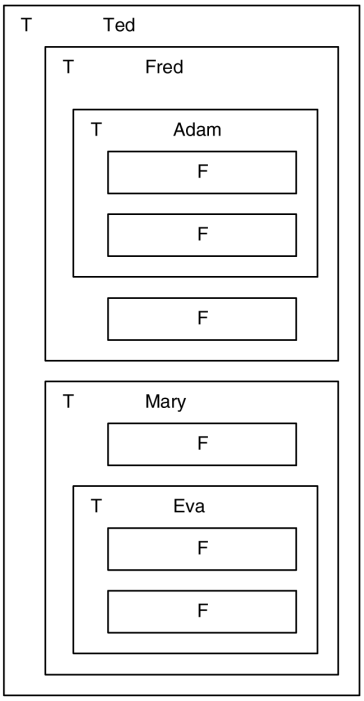


Рис. 4.2. Пример рекурсивной структуры данных

сопровождают каждое рекурсивное определение. Здесь особенно четко видна аналогия между структурированием программ и данных. Условный оператор (или оператор выбора) обязательно должен быть частью каждой рекурсивной процедуры, чтобы обеспечить завершение ее выполнения. На практике динамические структуры используют ссылки или указатели на свои элементы, а идея альтернативы (для завершения рекурсии) реализуется в понятии указателя, как объясняется в следующем разделе.

4.2. Указатели

Характерное свойство рекурсивных структур, четко отличающее их от фундаментальных структур (массивов, записей, множеств), – это их способность менять свой размер. Поэтому невозможно выделить фиксированный участок памяти для размещения рекурсивно определенной структуры, и, как следствие, компилятор не может связать конкретные адреса с компонентами таких переменных. Метод, чаще всего применяемый для решения этой проблемы, состоит в *динами-*

ческом распределении памяти (dynamic allocation of storage), то есть распределении памяти отдельным компонентам в тот момент, когда они возникают при выполнения программы, а не во время трансляции. При этом компилятор отводит фиксированный объем памяти для хранения адреса динамически размещаемой компоненты вместо самой компоненты. Например, родословная, показанная на рис. 4.2, будет представлена отдельными – вполне возможно, несмежными – записями, по одной на каждого индивида. Эти записи для отдельных людей связаны с помощью адресов, записанных в соответствующие поля father (отец) и mother (мать). Графически это лучше всего выразить с помощью стрелок или указателей (рис. 4.3).

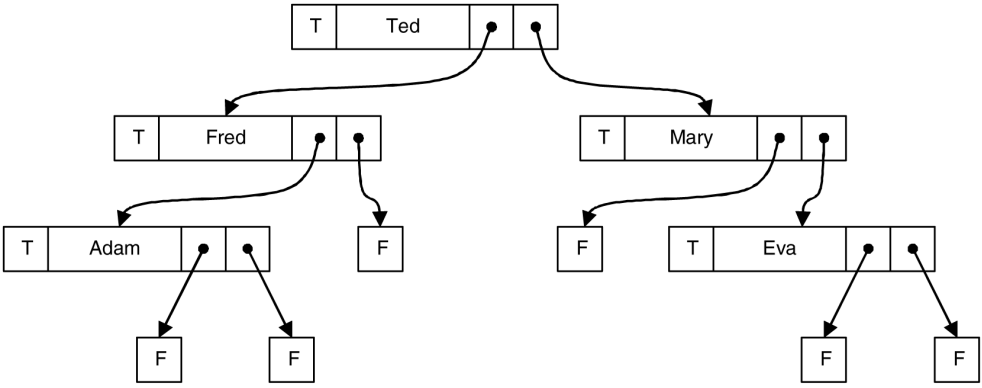


Рис. 4.3. Структура данных, связанная указателями

Важно подчеркнуть, что использование указателей для реализации рекурсивных структур – это всего лишь технический прием. Программисту не обязательно знать об их существовании. Память может распределяться автоматически в тот момент, когда в первый раз используется ссылка на новую компоненту. Но если явно разрешается использование указателей, то можно построить и более общие структуры данных, чем те, которые можно описать с помощью рекурсивных определений. В частности, тогда можно определять потенциально бесконечные или циклические структуры (графы) и указывать, что некоторые структуры используются совместно. Поэтому в развитых языках программирования принято разрешать явные манипуляции не только с данными, но и со ссылками на них. Это требует проведения четкого различия на уровне обозначений между данными и ссылками на данные, а также необходимость иметь типы данных, значениями которых являются указатели (ссылки) на другие данные. Мы будем использовать следующую нотацию для этой цели:

```
TYPE T = POINTER TO T0
```

Такое определение типа означает, что значения типа T – это указатели на данные типа T0. Принципиально важно, что тип элементов, на которые ссылается

указатель, очевиден из определения T. Мы говорим, что T *связан с* T0. Эта связь отличает указатели в языках высокого уровня от адресов в машинном языке и является весьма важным средством повышения безопасности в программировании посредством отражения семантики программы синтаксическими средствами.

Значения указательных типов порождаются при каждом динамическом размещении элемента данных. Мы будет придерживаться правила, что такое событие всегда должно описываться явно, в противоположность механизму автоматического размещения элемента данных при первой ссылке на него. С этой целью введем процедуру NEW. Если дана указательная переменная p типа T, то оператор NEW(p) размещает где-то в памяти переменную типа T0, а указатель на эту новую переменную записывает в переменную p (см. рис. 4.4). Сослаться в программе на само указательное значение теперь можно с помощью p (то есть это значение указательной переменной p). При этом переменная, на которую ссылается p, обозначается как p^. Обычно используют ссылки на записи. Если у записи, на которую ссылается указатель p, есть, например, поле x, то оно обозначается как p^.x. Поскольку ясно, что полями обладает не указатель, а только запись p^, то мы допускаем сокращенную нотацию p.x вместо p^.x.

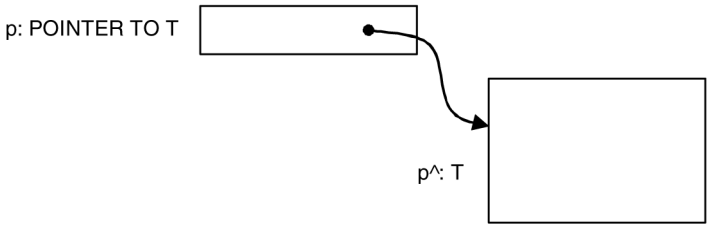


Рис. 4.4. Динамическое размещение переменной p^

Выше указывалось, что в каждом рекурсивном типе необходима компонента, позволяющая различать возможные варианты, чтобы можно было обеспечить конечность рекурсивных структур. Пример семейной родословной показывает весьма часто встречающуюся ситуацию, когда в одном из двух случаев другие компоненты отсутствуют. Это выражается следующим схематическим определением:

```
TYPE T = RECORD
            IF nonterminal: BOOLEAN THEN S(T) END
         END
```

S(T) обозначает последовательность определений полей, среди которых есть одно или более полей типа T, чем и обеспечивается рекурсивность. Все структуры типа, определенного по этой схеме, имеют древесное (или списковое) строение, подобное показанному на рис. 4.3. Его особенность – наличие указателей на компоненты данных, состоящие только из поля признака, то есть не несущие другой полезной информации. Метод реализации с явными указателями подсказывает простой способ сэкономить память, разрешив включать информацию о поле при-

знака в само указательное значение. Обычно для этого расширяют диапазон значений всех указательных типов единственным значением, которое вообще не является ссылкой ни на какой элемент. Обозначим это значение специальным символом NIL и постулируем, что все переменные указательных типов могут принимать значение NIL. Вследствие такого расширения диапазона указательных значений конечные структуры могут порождаться при отсутствии вариантов (условий) в их (рекурсивных) определениях.

Ниже даются новые формулировки объявленных ранее явно рекурсивных типов данных с использованием указателей. Заметим, что здесь уже нет поля known, так как ~p.known теперь выражается посредством p = NIL. Переименование типа ped в Person (индивид) отражает изменение точки зрения, произошедшее благодаря введению явных указательных значений. Теперь вместо того, чтобы сначала рассматривать данную структуру целиком и уже потом исследовать ее подструктуры и компоненты, внимание сосредоточивается прежде всего на компонентах, а их взаимная связь (представленная указателями) не фиксирована никаким явным определением.

```
TYPE term =           POINTER TO TermDescriptor;
TYPE exp =            POINTER TO ExpDescriptor;
TYPE ExpDescriptor =  RECORD op: INTEGER;  opd1, opd2: term END;
TYPE TermDescriptor = RECORD id: ARRAY 32 OF CHAR END
TYPE Person =         POINTER TO RECORD
                          name: ARRAY 32 OF CHAR;
                          father, mother:  Person
                      END
```

Замечание. Тип Person соответствует указателям на записи безымянного типа (PersonDescriptor).

Структура данных, представляющая родословную и показанная на рис. 4.2 и 4.3, снова показана на рис. 4.5, где указатели на неизвестных лиц обозначены константой NIL. Получающаяся экономия памяти очевидна.

В контексте рис. 4.5 предположим, что Fred и Mary – брат и сестра, то есть у них общие отец и мать. Эту ситуацию легко выразить заменой двух значений NIL в соответствующих полях двух записей. Реализация, которая скрывает указатели

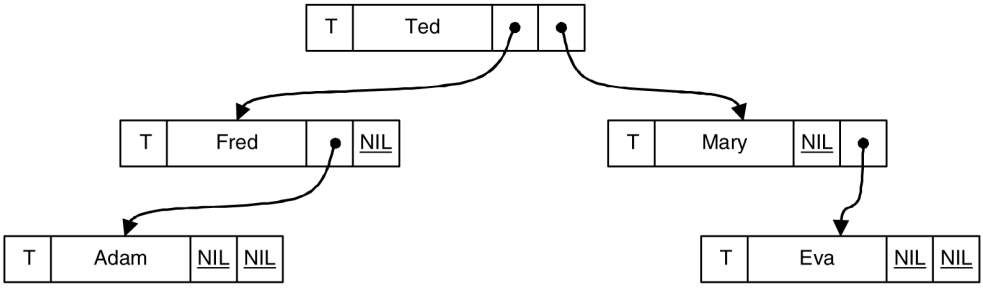


Рис. 4.5. Структура данных с указателями, имеющими значение NIL

или использует другие приемы работы с памятью, заставила бы программиста представить записи для родителей, то есть Adam и Eva, дважды. Хотя для чтения данных не важно, одной или двумя записями представлены два отца (или две матери), разница становится существенной, когда разрешено частичное изменение данных. Трактовка указателей как явных элементов данных, а не как скрытых средств реализации, позволяет программисту четко указать, где нужно совместить используемые блоки памяти, а где – нет.

Другое следствие явных указателей – возможность определять и манипулировать циклическими структурами данных. Разумеется, такая дополнительная гибкость не только предоставляет дополнительные возможности, но и требует от программиста повышенного внимания, поскольку работа с циклическими структурами данных легко может привести к бесконечным процессам.

Эта тесная связь мощи и гибкости средств с опасностью их неправильного использования хорошо известна в программировании и заставляет вспомнить оператор GOTO. В самом деле, если продолжить аналогию между структурами программ и данных, то чисто рекурсивные структуры данных можно сопоставить с процедурами, а введение указателей сравнимо с операторами GOTO. Ибо как оператор GOTO позволяет строить любые программные схемы (включая циклы), так и указатели позволяют строить любые структуры данных (включая кольцевые). [Однако в отличие от операторов GOTO, типизированные указатели не нарушают структурированности соответствующих записей – *прим. перев.*] Параллели между структурами управления и структурами данных суммированы в табл. 4.1.

Таблица 4.1. Соответствия структур управления и структур данных

Схема построения	Оператор программы	Тип данных
Неделимый элемент	Присваивание	Скалярный тип
Перечисление	Операторная последовательность	Запись
Повторение (число повторений известно)	Оператор **for**	Массив
Выбор	Условный оператор (запись с вариантами)	Объединение типов
Повторение	Оператор **while** или	Последовательностный тип **repeat**
Рекурсия	Процедура	Рекурсивный тип данных
Общий граф	Оператор перехода указателями	Структура, связанная

В главе 3 мы видели, что итерация является частным случаем рекурсии и что вызов рекурсивной процедуры P, определенной в соответствии со следующей схемой,

```
PROCEDURE P;
BEGIN
    IF B THEN P0; P END
END
```

где оператор P0 не включает в себя P и может быть заменен на эквивалентный оператор цикла

```
WHILE B DO P0 END
```

Аналогии, представленные в табл. 4.1, подсказывают, что похожая связь должна иметь место между рекурсивными типами данных и последовательностью. В самом деле, рекурсивный тип, определенный в соответствии со схемой

```
TYPE T =  RECORD
        IF b: BOOLEAN THEN t0: T0; t: T END
    END
```

где тип T0 не имеет отношения к T, может быть заменен на эквивалентную последовательность элементов типа T0.

Остальная часть этой главы посвящена созданию и работе со структурами данных, компоненты которых связаны с помощью явных указателей. Особое внимание уделяется конкретным простым схемам; из них можно понять, как работать с более сложными структурами. Такими простыми схемами являются линейный список (простейший случай) и деревья. Внимание, которое мы уделяем этим средствам структурирования данных, не означает, что на практике не встречаются более сложные структуры. Следующий рассказ, опубликованный в цюрихской газете в июле 1922 г., доказывает, что странности могут встречаться даже в тех случаях, которые обычно служат образцами регулярных структур, таких как (генеалогические) деревья. Мужчина жалуется на свою жизнь следующим образом:

Я женился на вдове, у которой была взрослая дочь. Мой отец, который часто нас навещал, влюбился в мою приемную дочь и женился на ней. Таким образом, мой отец стал моим зятем, а моя приемная дочь стала моей мачехой. Через несколько месяцев моя жена родила сына, который стал сводным братом моему отцу и моим дядей. Жена моего отца, то есть моя приемная дочь, тоже родила сына, который стал мне братом и одновременно внуком. Моя жена стала мне бабушкой, так как она мать моей мачехи. Следовательно, я муж моей жены и в то же время ее приемный внук; другими словами, я сам себе дедушка.

4.3. Линейные списки

4.3.1. Основные операции

Простейший способ связать набор элементов – выстроить их в простой список (list) или очередь. Ибо в этом случае каждому элементу нужен единственный указатель на элемент, следующий за ним.

Пусть типы Node (узел) и NodeDesc (desc от descriptor) определены, как показано ниже. Каждая переменная типа NodeDesc содержит три компоненты, а именно идентифицирующий ключ key, указатель на следующий элемент next и, возможно, другую информацию. Для дальнейшего нам понадобятся только key и next:

```
TYPE Node =        POINTER TO NodeDesc;
TYPE NodeDesc =  RECORD key: INTEGER; next: Node; data: ... END;
VAR p, q: Node   (*указательные переменные*)
```

На рис. 4.6 показан список узлов, причем указатель на его первый элемент хранится в переменной p. Вероятно, простейшая операция со списком, показанным на рис. 4.6, – это вставка элемента в голову списка. Сначала размещается элемент типа NodeDesc, при этом ссылка (указатель) на него записывается во вспомогательную переменную, скажем q. Затем простые присваивания указателей завершают операцию. Отметим, что порядок этих трех операторов менять нельзя.

```
NEW(q); q.next := p; p := q
```

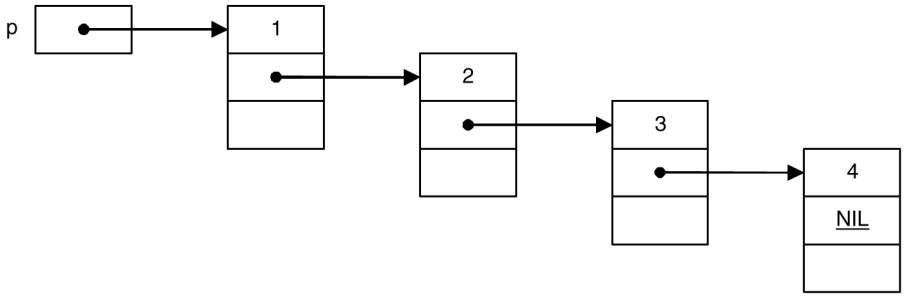


Рис. 4.6. Пример связного списка

Операция вставки элемента в голову списка сразу подсказывает, как такой список можно создать: начиная с пустого списка, нужно повторно добавлять в голову новые элементы. Процесс *создания списка* показан в следующем программном фрагменте; здесь n – число элементов, которые нужно связать в список:

```
p := NIL;  (*начинаем с пустого списка*)
WHILE n > 0 DO
    NEW(q); q.next := p; p := q;
    q.key := n; DEC(n)
END
```

Это простейший способ создания списка. Однако здесь получается, что элементы стоят в обратном порядке по сравнению с порядком их добавления в список. В некоторых задачах это нежелательно, и поэтому новые элементы должны присоединяться в конце, а не в начале списка. Хотя конец списка легко найти простым просмотром, такой наивный подход приводит к вычислительным затратам, которых можно избежать, используя второй указатель, скажем q, который всегда указывает на последний элемент. Этот метод применяется, например, в программе CrossRef (раздел 4.4.3), где создаются перекрестные ссылки для некоторого текста. Его недостаток – в том, что первый элемент должен вставляться по-другому, чем все остальные.

Явное наличие указателей сильно упрощает некоторые операции, которые в противном случае оказываются громоздкими. Среди элементарных операций со списками – вставка и удаление элементов (частичное изменение списка), а также, разумеется, проход по списку. Мы сначала рассмотрим *вставку* (insertion) в список.

Предположим, что элемент, на который ссылается указатель q, нужно вставить в список *после* элемента, на который ссылается указатель p. Нужные присваивания указателей выражаются следующим образом, а их эффект показан на рис. 4.7.

```
q.next := p.next;  p.next := q
```

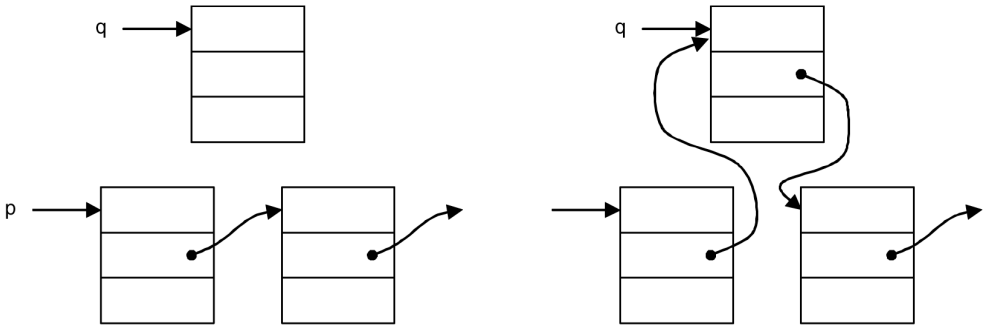


Рис. 4.7. Вставка после p^

Если нужна вставка *перед* указанным элементом p^, а не после него, то, казалось бы, возникает затруднение, так как в однонаправленной цепочке ссылок нет никакого пути от элемента к его предшественникам. Однако здесь может выручить простой прием. Он проиллюстрирован на рис. 4.8. Предполагается, что ключ нового элемента равен 8.

```
NEW(q); q^ := p^; p.key := k; p.next := q
```

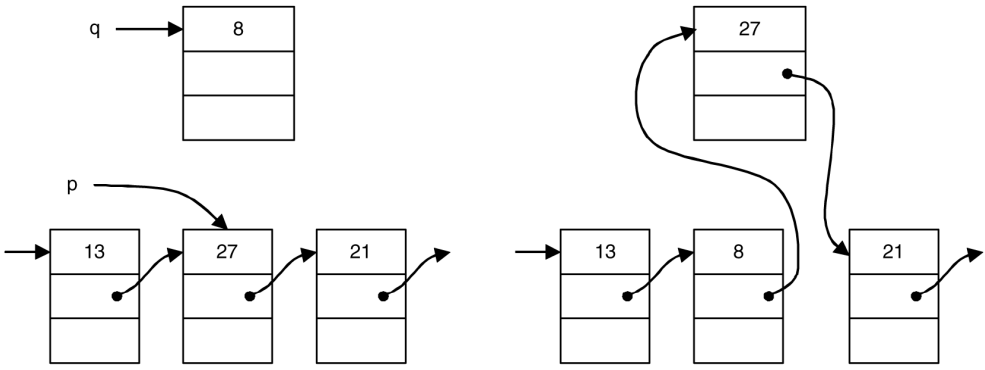


Рис. 4.8. Вставка перед p^

Очевидно, трюк состоит в том, чтобы вставить новую компоненту после p^, а потом произвести обмен значениями между новым элементом и p^.

Теперь рассмотрим *удаление из списка* (list deletion). Нетрудно удалить элемент, стоящий сразу за p^. Эта операция показана здесь вместе с повторной вставкой удаленного элемента в начало другого списка (обозначенного переменной q). Ситуация показана на рис. 4.9, из которого видно, что имеет место циклический обмен значений трех указателей.

```
r := p.next;  p.next := r.next;  r.next := q;  q := r
```

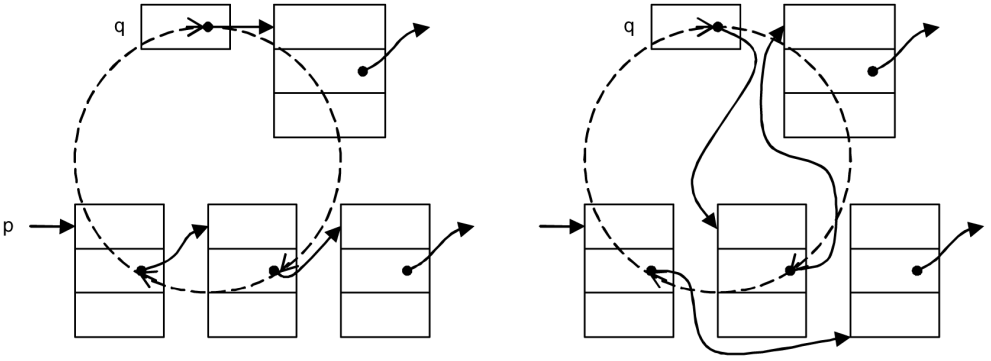


Рис. 4.9. Удаление и повторная вставка

Удаление самого элемента, на который указывает ссылка (а не следующего), труднее, так как здесь возникает та же проблема, что и со вставкой: невозможно просто так перейти назад от элемента к предшествующему. Но удаление следующего элемента после копирования его значения вперед – довольно очевидное и простое решение. Его можно применить, когда за p^ стоит некоторый элемент, то есть p^ не является последним элементом в списке. Однако необходимо гарантировать, что нет других переменных, ссылающихся на удаляемый элемент.

Обратимся теперь к фундаментальной операции *прохода по списку*. Предположим, что для каждого элемента списка, у которого первый элемент p^, нужно выполнить некоторую операцию P(x). Эту задачу можно выполнить так:

```
WHILE список, на который ссылается p, не пуст DO
    выполнить операцию P;
    перейти к следующему элементу

END
```

Детали этой операции выглядят следующим образом:

```
WHILE p # NIL DO
    P(p); p := p.next
END
```

Из определения оператора while и структуры связей следует, что P применяется ко всем элементам списка и ни к каким другим.

Очень часто используемая операция со списками – *поиск* в списке элемента с заданным ключом x. В отличие от массивов, поиск здесь должен быть чисто последовательным. Поиск прекращается, либо когда элемент найден, либо когда достигнут конец списка. Это выражается логической конъюнкцией двух термов. Снова предположим, что сначала p указывает на голову списка.

```
WHILE (p # NIL) & (p.key # x) DO p := p.next END
```

Равенство p = NIL означает, что записи p^ не существует, так что выражение p.key # x не определено. Поэтому порядок двух термов важен.

4.3.2. Упорядоченные списки и перестройка списков

Представленный алгоритм поиска в линейном списке очень похож на поиск в массиве или последовательности. На самом деле последовательность – это в точности линейный список, для которого способ связи с последующим элементом скрыт. Поскольку основные операции для последовательностей не допускают вставку новых элементов (разве что в конце) или удаления (кроме удаления всех элементов), выбор представления отдается полностью на откуп проектировщику, и он может выбрать последовательное размещение в памяти, располагая последовательные компоненты в непрерывных областях памяти. Линейные списки с явными указателями дают больше гибкости, и поэтому их следует использовать, когда в такой гибкости есть необходимость.

Чтобы показать различные приемы, рассмотрим в качестве примера задачу, которая будет возникать на протяжении этой главы. Задача состоит в том, чтобы прочесть текст, определить все слова в нем и подсчитать частоту их появления в тексте. То есть построить *алфавитный частотный словарь*.

Очевидное решение – строить список слов, обнаруживаемых в тексте. Список просматривается для каждого слова. Если слово в нем найдено, соответствующий счетчик увеличивается; в противном случае слово добавляется в список. Для простоты будем называть этот процесс просто *поиском,* хотя на самом деле здесь может иметь место и вставка. Чтобы сосредоточиться на существенных для нас здесь аспектах работы со списками, предположим, что слова из обрабатываемого текста уже извлечены и закодированы целыми числами, из которых и состоит входная последовательность.

Ниже дана самая прямолинейная формулировка процедуры поиска (процедура search). Переменная root ссылается на голову списка, в который новые слова вставляются по мере необходимости. Полная программа представлена ниже; в нее включена процедура печати построенного списка в виде таблицы. Печать таблицы – пример операции, в которой некоторая операция выполняется ровно один раз для каждого элемента списка.

```
TYPE Word = POINTER TO                                    (* ADruS432_List *)
    RECORD key, count: INTEGER; next: Word END;

PROCEDURE search (x: INTEGER; VAR root: Word);
    VAR w: Word;
BEGIN
    w := root;
    WHILE (w # NIL) & (w.key # x) DO w := w.next END;
    (* (w = NIL) OR (w.key = x) *)
    IF w = NIL THEN  (*новый элемент*)
        w := root;
        NEW(root); root.key := x; root.count := 1; root.next := w
    ELSE
        INC(w.count)
    END
END search;

PROCEDURE PrintList (w: Word);
    (*для вывода используется глобальный объект печати W*)
BEGIN
    WHILE w # NIL DO
        Texts.WriteInt(W, w.key, 8); Texts.WriteInt(W, w.count, 8);
        Texts.WriteLn(W);
        w := w.next
    END
END PrintList;
```

Алгоритм линейного поиска здесь похож на процедуру поиска для массива, так что можно вспомнить простой прием для упрощения условия завершения цикла – использование барьера. Этот прием можно применить и при поиске по списку; тогда барьер представляется вспомогательным элементом в конце списка. Новая процедура приведена ниже. Мы должны предположить, что добавлена глобальная переменная sentinel и что инициализация root := NIL заменена операторами

```
NEW(sentinel);  root := sentinel
```

порождающими элемент, который будет использоваться в качестве барьера.

```
PROCEDURE search (x: INTEGER; VAR root: Word);            (* ADruS432_List2 *)
    VAR w: Word;
BEGIN
    w := root; sentinel.key := x;
    WHILE w.key # x DO w := w.next END;
    IF w = sentinel THEN  (*новый элемент*)
        w := root;
        NEW(root); root.key := x; root.count := 1; root.next := w
    ELSE
        INC(w.count)
    END
END search
```

Очевидно, что мощь и гибкость связного списка в данном примере используются плохо и что последовательный просмотр всего списка приемлем, только если число элементов невелико. Однако легко сделать простое улучшение: *поиск в упорядоченном списке*. Если список упорядочен (скажем, по возрастанию ключей), то поиск может быть прекращен, как только встретился первый ключ, больший нового. Упорядочение списка достигается вставкой новых элементов в надлежащем месте, а не в начале списка. При этом упорядоченность получается практически бесплатно. Это достигается благодаря легкости вставки элемента в связный список, то есть благодаря полному использованию его гибкости. Такой возможности нет ни при работе с массивами, ни с последовательностями. (Однако заметим, что даже упорядоченные списки не дают возможности реализовать аналог двоичного поиска для массивов.)

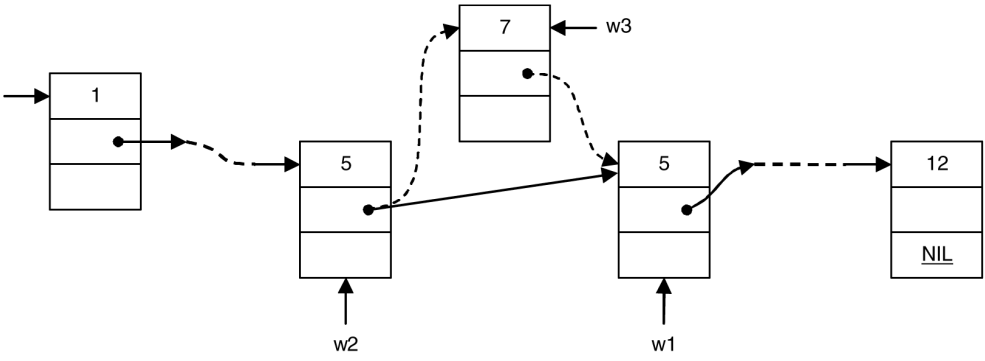


Рис. 4.10. Вставка в упорядоченный список

Поиск в упорядоченном списке дает типичный пример ситуации, когда новый элемент нужно вставить *перед* заданным, в данном случае перед первым элементом, чей ключ оказался слишком большим. Однако мы применим здесь новый прием, который отличается от показанного ранее. Вместо копирования значений вдоль списка проходят два указателя; w2 отстает на шаг от w1 и поэтому указывает на правильное место вставки, когда w1 обнаруживает слишком большой ключ. Общий шаг вставки показан на рис. 4.10. Указатель на новый элемент (w3) должен быть присвоен полю w2.next, за исключением того случая, когда список еще пуст. Из соображений простоты и эффективности нежелательно отслеживать этот особый случай с помощью условного оператора. Единственный способ избежать этого – ввести фиктивный элемент в начале списка. Соответственно, оператор инициализации root := NIL заменяется на

```
NEW(root); root.next := NIL
```

Рассматривая рис. 4.10, можно записать условие перехода к следующему элементу; оно состоит из двух термов, а именно

```
(w1 # NIL) & (w1.key < x)
```

Получается такая процедура поиска:

```
PROCEDURE search (x: INTEGER; VAR root: Word);          (* ADruS432_List3 *)
    VAR w1, w2, w3: Word;
BEGIN
    (*w2 # NIL*)
    w2 := root; w1 := w2.next;
    WHILE (w1 # NIL) & (w1.key < x) DO
        w2 := w1; w1 := w2.next
    END;
    (* (w1 = NIL) OR (w1.key >= x) *)
    IF (w1 = NIL) OR (w1.key > x) THEN  (*новый элемент*)
        NEW(w3); w2.next := w3;
        w3.key := x; w3.count := 1; w3.next := w1
    ELSE
        INC(w1.count)
    END
END search
```

Чтобы ускорить поиск, охрану оператора while опять можно упростить, используя барьер. Тогда с самого начала нужен как фиктивный элемент в начале списка, так и барьер в конце.

Теперь пора спросить, какой выигрыш нам даст поиск в упорядоченном списке. Помня о том, что дополнительные усложнения оказались малы, не стоит ожидать чуда.

Предположим, что все слова в тексте встречаются с одинаковой частотой. В этом случае выигрыш от лексикографического упорядочения – после того как все слова включены в список – тоже нулевой, так как положение слова не играет роли, если только полное число всех операций поиска велико и если все слова встречаются с одинаковой частотой. Однако выигрыш имеет место при вставке нового слова. Ведь тогда вместо прохода по всему списку в среднем просматривается только половина списка. Поэтому вставка в упорядоченный список оправдывает себя, только если в обрабатываемом тексте число различных слов велико по сравнению с частотами их появления. И поэтому предыдущие примеры программ хороши только как упражнения в программировании, но не для практического использования.

Организацию данных в связный список можно рекомендовать, когда число элементов относительно мало (< 50), меняется и, более того, когда нет информации о частоте обращения к ним. Типичный пример – таблица имен в компиляторах языков программирования. Каждое объявление добавляет новое имя, которое удаляется из списка после выхода из его области видимости. Использование простых связных списков уместно в приложениях с относительно короткими программами. Но даже в этом случае можно добиться значительного ускорения доступа к данным с помощью очень простого приема, который упоминается здесь прежде всего потому, что он представляет собой удачную иллюстрацию гибкости связных списков.

Типичное свойство программ состоит в том, что появления одного и того же идентификатора очень часто группируются таким образом, что за одним его появлением то же слово часто появляется еще один или несколько раз. Это обстоятельство наводит на мысль реорганизовывать список после каждой операции поиска перемещением найденного слова в голову списка, чтобы уменьшить длину пути поиска, когда это слово нужно будет снова найти. Такой способ доступа называют *поиском с переупорядочением списка*, или, несколько помпезно, поиском в *самоорганизующемся списке*. Представляя соответствующий алгоритм в виде процедуры, воспользуемся уже приобретенным опытом и с самого начала введем барьер. На самом деле наличие барьера не только ускоряет поиск, но в данном случае еще и упрощает программу. В исходном состоянии список не должен быть пуст, но уже должен содержать элемент-барьер. Операторы инициализации таковы:

```
NEW(sentinel);  root := sentinel
```

Заметим, что главное отличие нового алгоритма от простого поиска в списке – это действия по переупорядочиванию при обнаружении элемента. В этом случае он удаляется со старой позиции и вставляется в голову списка. Такое удаление снова требует пары следующих друг за другом указателей, чтобы помнить положение предшественника w2 найденного элемента w1. Это, в свою очередь, требует особой обработки первого элемента (то есть пустого списка). Чтобы стало понятней, какие здесь нужны манипуляции с указателями, на рис. 4.11 показаны два указателя в тот момент, когда w1 был идентифицирован как искомый элемент. Конфигурация после корректного переупорядочения показана на рис. 4.12

```
PROCEDURE search (x: INTEGER; VAR root: Word);          (* ADruS432_List4 *)
    VAR w1, w2: Word;
BEGIN
    w1 := root; sentinel.key := x;
    IF w1 = sentinel THEN (*первый элемент*)
        NEW(root); root.key := x; root.count := 1; root.next := sentinel
    ELSIF
        w1.key = x THEN INC(w1.count)
    ELSE (*поиск*)
        REPEAT w2 := w1; w1 := w2.next
        UNTIL w1.key = x;
        IF w1 = sentinel THEN (*новый элемент*)
            w2 := root;
            NEW(root); root.key := x; root.count := 1; root.next := w2
        ELSE (*нашли, переупорядочиваем*)
            INC(w1.count);
            w2.next := w1.next; w1.next := root; root := w1
        END
    END
END search
```

Выигрыш в этом методе поиска сильно зависит от степени группировки слов во входных данных. При фиксированной степени группировки выигрыш будет

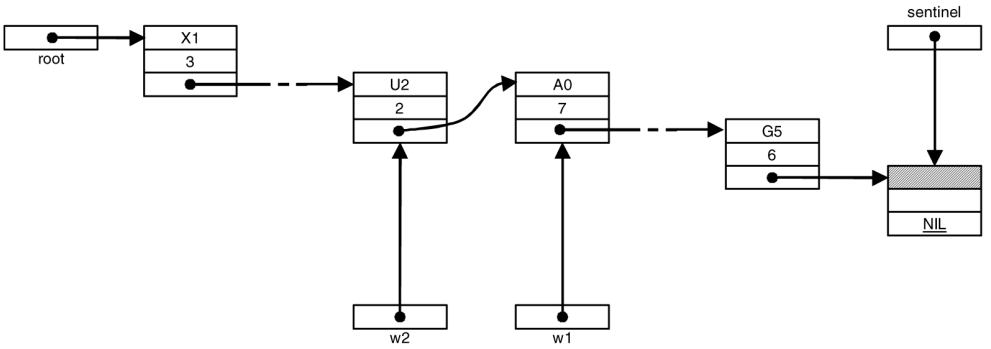


Рис. 4.11. Список до переупорядочения

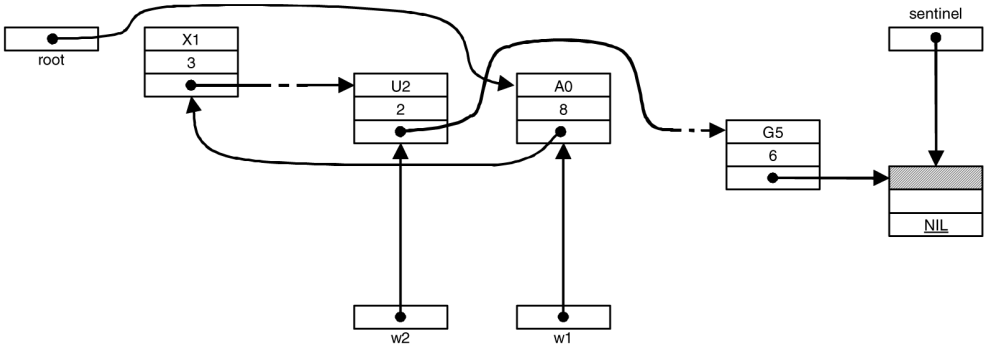


Рис. 4.12. Список после переупорядочения

тем больше, чем длинней список. Чтобы получить представление об ожидаемом выигрыше, были выполнены эмпирические измерения с использованием вышеприведенной программы для одного короткого и одного относительно длинного текста, и затем сравнивались методы поиска в упорядоченном списке и поиск с переупорядочиванием. Данные измерений собраны в табл. 4.2. К сожалению, наибольший выигрыш достигается в том случае, когда данные все равно нужно организовывать по-другому. Мы вернемся к этому примеру в разделе 4.4.

Таблица 4.2. Сравнение методов поиска в списке

	Тест 1	Тест 2
Число разных ключей	53	582
Число появлений ключей	315	14341
Время поиска с упорядочением	6207	3200622
Время поиска с перегруппировкой	4529	681584
Фактор ускорения	1.37	4.70

4.3.3. Применение: топологическая сортировка

Хороший пример использования гибкой динамической структуры данных – *топологическая сортировка*. Это сортировка элементов, для которых определен *частичный порядок*, то есть отношение порядка задано для некоторых, но не для всех пар элементов. Вот примеры частичного упорядочения:

1. В словаре одни слова могут быть определены через другие. Если слово v используется в определении слова w, то будем писать v ⟨ w. Топологическая сортировка слов означает такую их расстановку, чтобы каждое слово стояло после всех тех слов, которые нужны для его определения.
2. Задача (например, инженерный проект) разбита на подзадачи. Обычно одни подзадачи должны быть выполнены до выполнения других. Если подзадача v должна быть выполнена до подзадачи w, то мы пишем v ⟨ w. Топологическая сортировка означает такую их расстановку, чтобы при начале выполнения каждой подзадачи все нужные подзадачи были бы уже выполнены.
3. В программе университетского обучения некоторые курсы предполагают владение материалом других курсов, которые поэтому должны изучаться раньше первых. Если курс v должен быть прослушан до курса w, то мы пишем v ⟨ w. Топологическая сортировка означает расстановку курсов в таком порядке, чтобы все курсы, которые нужно прослушать до некоторого курса, всегда стояли в списке до него.
4. В программе некоторые процедуры могут содержать вызовы других процедур. Если процедура v вызывается в процедуре w, то пишем v ⟨ w. Топологическая сортировка подразумевает такую расстановку объявлений процедур, чтобы никакая процедура не вызывала еще не объявленные процедуры.

В общем случае частичный порядок на множестве S – это некоторое отношение между элементами S. Будем обозначать его символом ⟨ (читается «предшествует») и требовать выполнения следующих трех свойств (аксиом) для любых разных элементов x, y, z из S:

1) если x ⟨ y и y ⟨ z, то x ⟨ z (транзитивность);
2) если x ⟨ y, то неверно, что y ⟨ x (антисимметрия);
3) неверно, что z ⟨ z (нерефлексивность).

По очевидным причинам будем предполагать, что множества S, к которым будет применяться топологическая сортировка, конечны. Поэтому частичная сортировка может быть проиллюстрирована диаграммой или графом, в котором вершины обозначают элементы S, а направленные ребра представляют отношения упорядоченности. Пример показан на рис. 4.13.

Задача топологической сортировки – погрузить отношение частичного порядка в линейный порядок. Графически это подразумевает расположение вершин графа в линию так, чтобы все стрелки указывали вправо, как показано на рис. 4.14. Свойства (1) и (2) частичного упорядочения гарантируют, что граф не содержит петель. Это в точности то условие, при котором погружение в линейный порядок возможно.

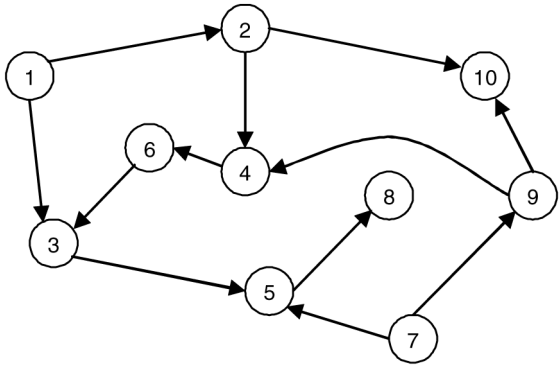


Рис. 4.13. Частично упорядоченное множество

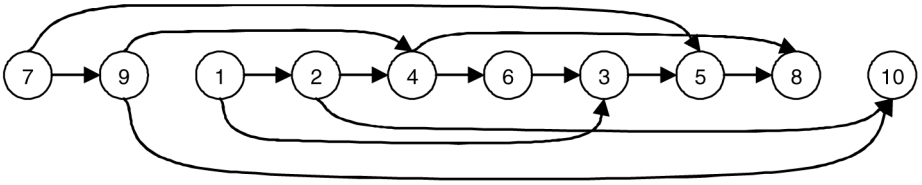


Рис. 4.14. Расположение в линию частично упорядоченного множества из рис. 4.13

Как приступить к поиску одного из возможных линейных упорядочений? Рецепт довольно прост. Начнем с выбора любого элемента, у которого нет предшественников (должен существовать хотя бы один такой элемент, иначе в графе будет петля). Этот элемент помещается в начало будущего списка и удаляется из множества S. Остаток множества по-прежнему частично упорядочен, и тот же алгоритм может применяться снова – до тех пор, пока множество не станет пустым.

Чтобы описать алгоритм более строго, нужно выбрать структуру данных и представление для S и для его частичного порядка. Выбор такого представления определяется операциями, которые нужно с ним производить, в частности речь идет об операции выбора элементов, не имеющих предшественников. Поэтому каждый элемент должен представляться тремя характеристиками: своим идентифицирующим ключом, множеством элементов-«последователей» и числом элементов-«предшественников». Так как число n элементов в S не фиксировано заранее, множество удобно организовать как связный список. Следовательно, дополнительной компонентой описания каждого элемента будет указатель на следующий элемент списка. Будем считать, что ключи являются целыми (но не обязательно идущими подряд от 1 до n). Аналогично, множество последователей каждого элемента удобно представить связным списком. Каждый элемент списка последователей описывается своим ключом и указателем на следующий элемент в этом списке. Если дескрипторы главного списка, в котором каждый элемент множества S встречается в точности один раз, назвать *ведущими* (leader), а дескрипторы эле-

ментов в списках последователей – *ведомыми* (trailer), то придем к следующим объявлениям типов данных:

```
TYPE Leader =      POINTER TO LeaderDesc;
     Trailer =     POINTER TO TrailerDesc;
     LeaderDesc =  RECORD key, count: INTEGER;
                      trail: Trailer;  next: Leader
                   END;
     TrailerDesc = RECORD id: Leader;  next: Trailer
                   END
```

Предположим, что множество S и отношения порядка между его элементами изначально представлены как последовательность пар ключей во входном файле. Такие входные данные для примера на рис. 4.13 представлены ниже, где для ясности явно указаны символы б , обозначающие отношение частичного порядка:

```
1 ⟨ 2     2 ⟨ 4     4 ⟨ 6     2 ⟨ 10    4 ⟨ 8     6 ⟨ 3     1 ⟨ 3
3 ⟨ 5     5 ⟨ 8     7 ⟨ 5     7 ⟨ 9     9 ⟨ 4     9 ⟨ 10
```

Первая часть программы топологической сортировки должна прочесть входные данные и преобразовать их в некую списковую структуру. Это делается последовательным чтением пар ключей x и y (x ⟨ y). Обозначим указатели на их представления в связном списке ведущих как p и q. Эти записи нужно найти в списке, а если их там еще нет, вставить в этот список. Данная задача решается процедурой-функцией find. Затем к списку ведомых для x нужно добавить новый дескриптор, содержащий ссылку на q; счетчик предшественников для y увеличивается на 1. Этот алгоритм называется *фазой ввода*. Рисунок 4.15 показывает списочную структуру, порождаемую при обработке вышеприведенных входных данных. Функция find(w) выдает указатель на элемент списка с ключом w.

В следующем фрагменте программы мы используем средства *сканирования текстов* системы Оберон. Здесь текст рассматривается как последовательность не литер, а *лексем*, которые могут быть идентификаторами, числами, цепочками литер, а также специальными литерами (такими как +, *, < и т. д.). Процедура Texts.Scan(S) просматривает текст, читая очередную лексему. Сканер S является разновидностью бегунка для текстов.

```
(*фаза ввода*)
NEW(head); tail := head; Texts.Scan(S);
WHILE S.class = Texts.Int DO
   x := S.i; Texts.Scan(S); y := S.i; p := find(x); q := find(y);
   NEW(t); t.id := q; t.next := p.trail;
   p.trail := t; INC(q.count); Texts.Scan(S)
END
```

После того как в фазе ввода построена структура данных, показанная на рис. 4.15, можно начинать выполнение собственно топологической сортировки, описанной ранее. Но так как она состоит из повторяющегося выбора элемента с нулевым числом предшественников, разумно сначала собрать все такие элемен-

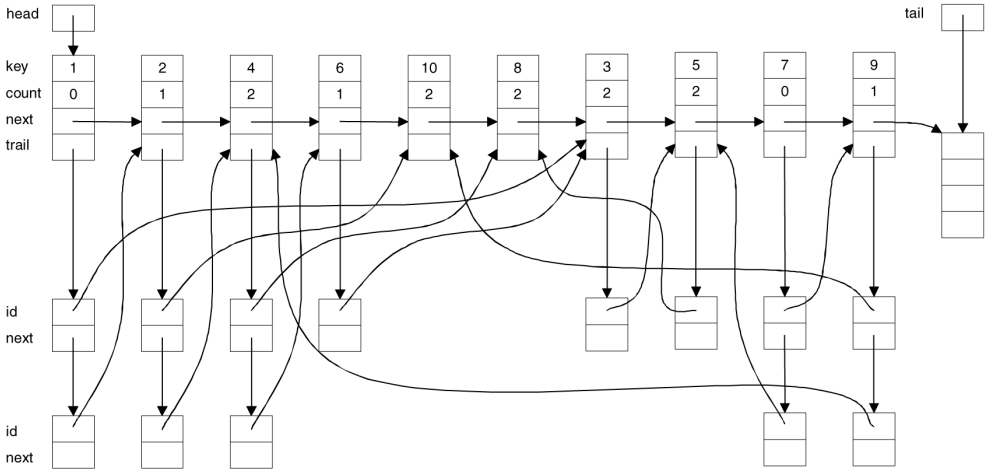


Рис. 4.15. Списковая структура, порожденная программой TopSort

ты в связный список. Заметив, что исходный список ведущих впоследствии не понадобится, можно повторно использовать то же самое поле next для связывания ведущих с нулевым числом предшественников. Такая операция замены одного списка другим часто встречается в обработке списков. Ниже она выписана в деталях. Для простоты новый список создается в обратном порядке:

```
(*поиск ведущих без предшественников*)
p := head; head := NIL;
WHILE p # tail DO
    q := p; p := q.next;
    IF q.count = 0 THEN (*вставить q^ в новый список*)
        q.next := head; head := q
    END
END
```

Обращаясь к рис. 4.15, видим, что список ведущих, образованный цепочкой ссылок next, заменяется показанным на рис. 4.16, где неизображенные указатели остаются без изменений.

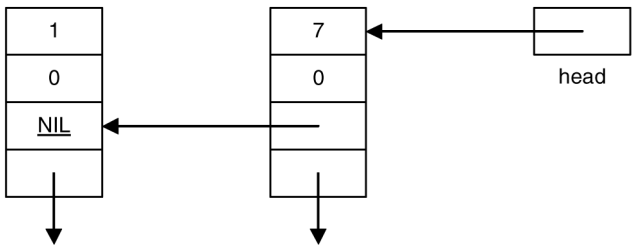


Рис. 4.16. Список ведущих с нулевым числом предшественников

После всей этой подготовки, имея удобное представление частично упорядоченного множества S, мы можем наконец приступить собственно к задаче топологической сортировки, то есть порождения выходной последовательности. Первая черновая версия может быть такой:

```
q := head;
WHILE q # NIL DO (*распечатать этот элемент, затем удалить его*)
    Texts.WriteInt(W, q.key, 8); DEC(n);
    t := q.trail; q := q.next;
    уменьшить счетчики предшественников
    для всех его последователей в списке t;
    если счетчик обнуляется, вставить этот элемент
    в список ведущих q
END
```

Операция, которую здесь нужно уточнить, представляет собой еще один просмотр списка. На каждом шаге вспомогательная переменная p обозначает ведущего, чей счетчик нужно уменьшить и проверить.

```
WHILE t # NIL DO
    p := t.id; DEC(p.count);
    IF p.count = 0 THEN (*вставить p^ в список ведущих*)
        p.next := q; q := p
    END;
    t := t.next
END
```

Этим завершается построение программы топологической сортировки. Заметим, что в программе введен счетчик n для подсчета ведущих, порожденных в фазе ввода. Этот счетчик уменьшается каждый раз, когда ведущий выводится в фазе вывода. Поэтому его значение должно снова стать нулем после завершения программы. Если этого не происходит, значит, каждый из оставшихся элементов имеет предшественников. Очевидно, множество S в этом случае не является частично упорядоченным. Фаза вывода в представленном виде – пример процесса с «пульсирующим» списком, где элементы вставляются и удаляются в непредсказуемом порядке. Другими словами, этот пример в полной мере демонстрирует гибкость списочной структуры с явными связями.

```
VAR head, tail: Leader; n: INTEGER;                    (* ADruS433_TopSort *)

PROCEDURE find (w: INTEGER): Leader;
    VAR h: Leader;
BEGIN
    h := head; tail.key := w; (*барьер*)
    WHILE h.key # w DO h := h.next END;
    IF h = tail THEN
        NEW(tail); INC(n);
        h.count := 0; h.trail := NIL; h.next := tail
    END;
```

```
    RETURN h
END find;

PROCEDURE TopSort (VAR R: Texts.Reader);
    VAR p, q: Leader; t: Trailer;  (*для вывода используется глобальный W*)
        x, y: INTEGER;
BEGIN
    (*инициализировать список ведущих с фиктивным элементом-барьером*)
    NEW(head); tail := head; n := 0;

    (*фаза ввода*)
    Texts.Scan(S);
    WHILE S.class = Texts.Int DO
        x := S.i; Texts.Scan(S); y := S.i; p := find(x); q := find(y);
        NEW(t); t.id := q; t.next := p.trail;
        p.trail := t; INC(q.count); Texts.Scan(S)
    END;

    (*поиск ведущих без предшественников*)
    p := head; head := NIL;
    WHILE p # tail DO
        q := p; p := q.next;
        IF q.count = 0 THEN (*вставить q в новый список*)
            q.next := head; head := q
        END
    END;

    (*фаза вывода*)
    q := head;
    WHILE q # NIL DO
        Texts.WriteLn(W); Texts.WriteInt(W, q.key, 8); DEC(n);
        t := q.trail; q := q.next;
        WHILE t # NIL DO
            p := t.id; DEC(p.count);
            IF p.count = 0 THEN (*вставить p в список ведущих*)
                p.next := q; q := p
            END;
            t := t.next
        END
    END;

    IF n # 0 THEN
        Texts.WriteString(W, "Набор не является частично упорядоченным")
    END;
    Texts.WriteLn(W)
END TopSort.
```

4.4. Деревья

4.4.1. Основные понятия и определения

Мы видели, что последовательности и списки удобно определять следующим образом. Последовательность (список) с базовым типом T – это:

1) пустая последовательность (список);
2) конкатенация (сцепление) некоторого элемента типа T и последовательности с базовым типом T.

Тем самым рекурсия используется как средство определения метода структурирования, а именно последовательности или итерации. Последовательности и итерации встречаются настолько часто, что их обычно рассматривают в качестве фундаментальных схем структурирования и поведения. Но нужно помнить, что их определить с помощью рекурсии можно, однако обратное неверно, тогда как рекурсию можно элегантно и эффективно использовать для определения гораздо более изощренных структур. Хорошо известный пример – деревья. Определим понятие дерева следующим образом. *Дерево* с базовым типом T – это либо:

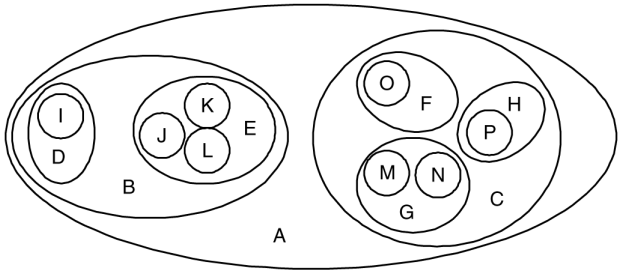
1) пустое дерево, либо
2) узел типа T с конечным числом *поддеревьев*, то есть не соединяющихся между собой деревьев с базовым типом T.

Сходство рекурсивных определений последовательностей и деревьев показывает, что последовательность (список) – это дерево, в котором у каждого узла не больше одного поддерева. Поэтому список можно считать вырожденным деревом.

Есть несколько способов представить дерево. Например, рис. 4.17 показывает несколько таких представлений для дерева с базовым типом T = CHAR. Все эти представления изображают одну и ту же структуру и потому эквивалентны. Но именно представление в виде графа объясняет, почему здесь используют термин «дерево». Довольно странно, что деревья чаще изображают растущими вниз или – если использовать другую метафору – показывают корни деревьев. Однако последнее описание вносит путаницу, так как корнем (root) обычно называют верхний узел (A).

Упорядоченное дерево – это такое дерево, в котором для ветвей каждого узла фиксирован определенный порядок. Поэтому два упорядоченных дерева на рис. 4.18 различны. Узел y, который находится непосредственно под узлом x, называется (непосредственным) *потомком* узла x; если x находится на уровне i, то говорят, что y находится на *уровне* i+1. Обратно, узел x называется (непосредственным) *предком* узла y. Уровень корня по определению равен нулю. Максимальный уровень элементов в дереве называется его *глубиной*, или *высотой*.

a)



b)

```
(A(B(D(I),E(J,K,L)),C(F(O),G(M,N),H(P))))
```

c)

```
A
    B
        D
            I
        E
            J
            K
            L
    C
        F
            O
        G
            M
            N
        H
            P
```

d)

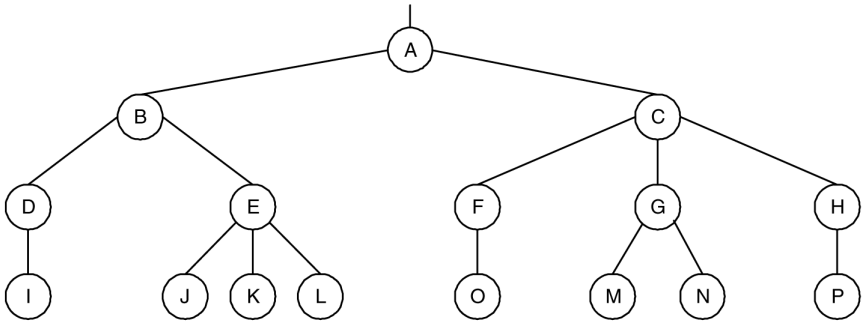


Рис. 4.17. Представление дерева посредством (a) вложенных подмножеств, (b) скобочного выражения, (c) текста с отступами, (d) графа

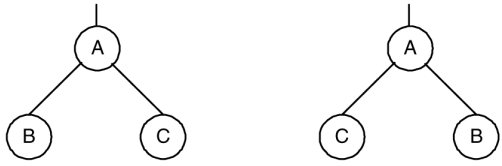


Рис. 4.18. Два разных упорядоченных дерева

Если узел не имеет потомков, то он называется *концевым* (*терминальным*), или *листом*; узел, не являющийся концевым, называется *внутренним*. Количество (непосредственных) потомков внутреннего узла – это его *степень*. Максимальная степень всех узлов называется степенью дерева. Число ветвей или ребер, по которым нужно пройти, чтобы попасть из корня в узел x, называется его *длиной пути*. Длина пути корня равна нулю, его непосредственных потомков – 1 и т. д. Вообще, длина пути узла на уровне i равна i. Длина путей дерева определяется как сумма длин путей всех его компонент. Ее еще называют *длиной внутренних путей*. Например, у дерева на рис. 4.17 длина внутренних путей равна 36. Очевидно, средняя длина пути равна

$$P_{int} = (\mathbf{S}i: 1 \le i \le n: n_i \times i) / n$$

где n_i – число узлов на уровне i. Чтобы определить *длину внешних путей*, расширим дерево особыми дополнительными узлами всюду, где в исходном дереве отсутствовало поддерево. Здесь имеется в виду, что все узлы должны иметь одинаковую степень, а именно степень дерева. Такое расширение дерева равнозначно заполнению пустых ветвей, причем у добавляемых дополнительных узлов, конечно, потомков нет. Дерево с рис. 4.17, расширенное дополнительными узлами, показано на рис. 4.19, где дополнительные узлы показаны квадратиками. Длина внешних путей теперь определяется как сумма длин путей всех дополнительных узлов. Если число дополнительных узлов на уровне i равно m_i, то средняя длина внешнего пути равна

$$P_{ext} = (\mathbf{S}i: 1 \le i \le m_i \times i) / m$$

Длина внешних путей дерева на рис. 4.19 равна 120.

Число m дополнительных узлов, которые нужно добавить к дереву степени d, определяется числом n исходных узлов. Заметим, что к каждому узлу ведет в точности одно ребро. Таким образом, в расширенном дереве m + n ребер. С другой

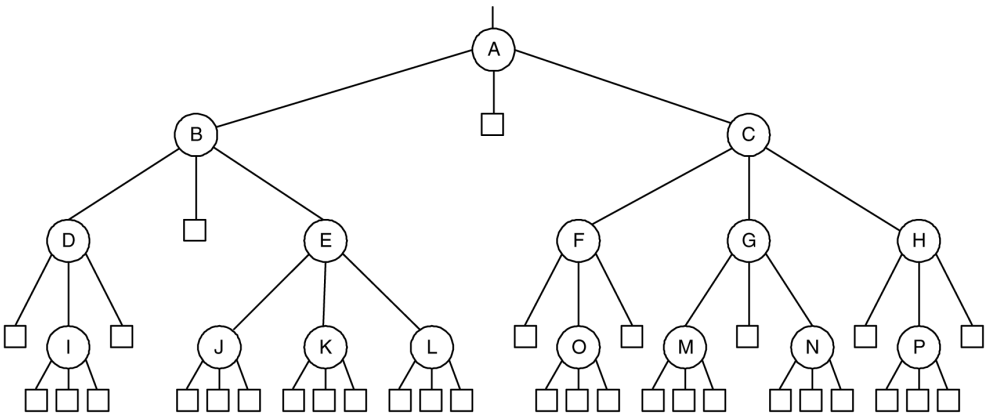


Рис. 4.19. Троичное дерево, расширенное дополнительными узлами

стороны, из каждого исходного узла выходит d ребер, из дополнительных узлов – ни одного. Поэтому всего имеется d*n + 1 ребер, где 1 соответствует ребру, ведущему к корню. Две формулы дают следующее уравнение, связывающее число m дополнительных узлов и число n исходных узлов: $d \times n + 1 = m + n$, откуда

$$m = (d-1) \times n + 1$$

Дерево заданной высоты h будет иметь максимальное число узлов, если у всех узлов есть d поддеревьев, кроме узлов на уровне h, у которых поддеревьев нет. Тогда у дерева степени d на уровне 0 есть 1 узел (а именно корень), на уровне 1 – d его потомков, на уровне 2 – d^2 потомков d узлов уровня 1 и т. д. Отсюда получаем выражение

$$N_d(h) = \mathbf{S}i: 0 \le i < h: d^i$$

для максимального числа узлов дерева высоты h и степени d. В частности, для d = 2 получаем

$$N_2(h) = 2^h - 1$$

Особенно важны упорядоченные деревья степени 2. Их называют *двоичными* (*бинарными*) деревьями. Определим упорядоченное двоичное дерево как конечное множество элементов (узлов), которое либо пусто, либо состоит из корня (корневого узла) с двумя отдельными двоичными деревьями, которые называют *левым* и *правым поддеревом* корня. В дальнейшем мы будем в основном заниматься двоичными деревьями, поэтому под деревом всегда будем подразумевать *упорядоченное двоичное дерево*. Деревья степени больше 2 называются *сильно ветвящимися деревьями*, им посвящен раздел 4.7.1.

Знакомые примеры двоичных деревьев – семейная родословная, где отец и мать индивида представлены узлами-потомками (!); таблица результатов теннисного турнира, где каждому поединку соответствует узел, в котором записан победитель, а две предыдущие игры соперников являются потомками; арифметическое выражение с двухместными операциями, где каждому оператору соответствует узел, а операндам – поддеревья (см. рис. 4.20).

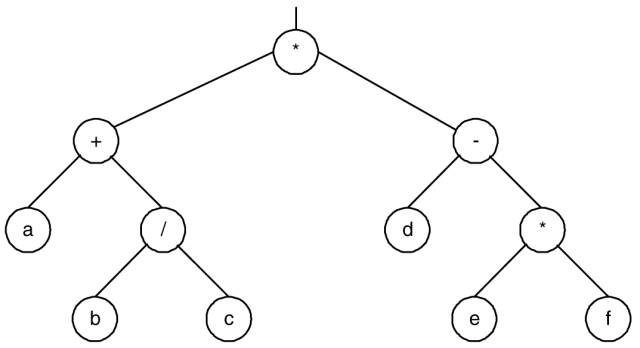


Рис. 4.20. Представление в виде дерева для выражения (a + b/c) * (d – e*f)

Обратимся теперь к проблеме представления деревьев. Изображение таких рекурсивных конструкций в виде ветвящихся структур подсказывает, что здесь можно использовать наш аппарат указателей. Очевидно, нет смысла объявлять переменные с фиксированной древесной структурой; вместо этого фиксированную структуру, то есть фиксированный тип, будут иметь узлы, для которых степень дерева определяет число указательных компонент, ссылающихся на поддеревья узла. Очевидно, ссылка на пустое дерево обозначается с помощью NIL. Следовательно, дерево на рис. 4.20 состоит из компонент определенного ниже типа и может тогда быть построено, как показано на рис. 4.21.

```
TYPE Node =     POINTER TO NodeDesc;
TYPE NodeDesc = RECORD  op: CHAR; left, right: Node  END
```

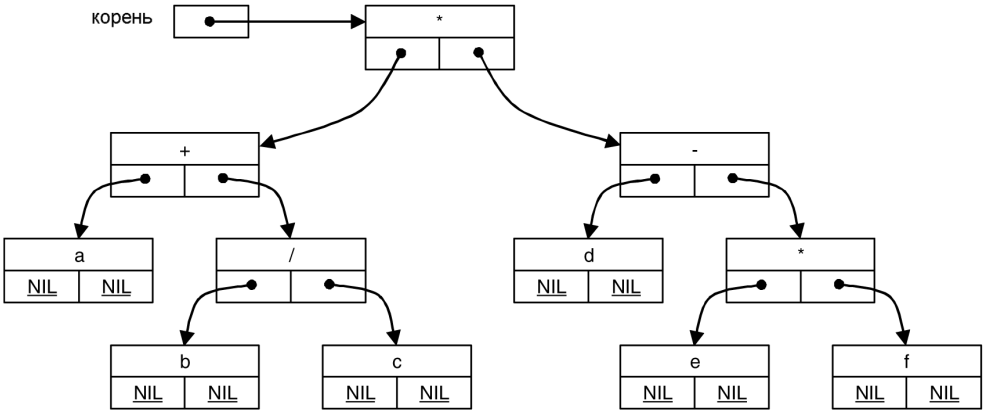


Рис. 4.21. Дерево рис. 4.20, представленное как связная структура данных

Прежде чем исследовать, какую пользу можно извлечь, применяя деревья, и как выполнять операции над ними, дадим пример программы, которая строит дерево. Предположим, что нужно построить дерево, значениями в узлах которого являются n чисел, считываемых из входного файла. Чтобы сделать задачу интересней, будем строить дерево с n узлами, имеющее минимальную высоту. Чтобы получить минимальную высоту при заданном числе узлов, нужно размещать максимальное возможное число узлов на всех уровнях, кроме самого нижнего. Очевидно, этого можно достичь, распределяя новые узлы поровну слева и справа от каждого узла. Это означает, что мы строим дерево для заданного n так, как показано на рис. 4.22 для n = 1, ... , 7.

Правило равномерного распределения при известном числе узлов n лучше всего сформулировать рекурсивно:

1. Использовать один узел в качестве корня.
2. Построить таким образом левое поддерево с числом узлов nl = n DIV 2.
3. Построить таким образом правое поддерево с числом узлов nr = n – nl – 1.

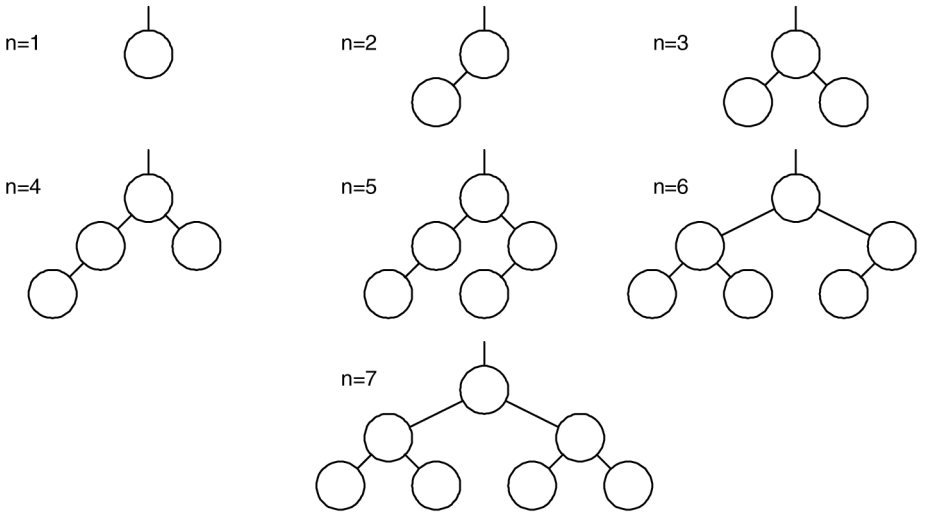


Рис. 4.22. Идеально сбалансированные деревья

Это правило реализуется рекурсивной процедурой, которая читает входной файл и строит идеально сбалансированное дерево. Вот определение: дерево является *идеально сбалансированным*, если для каждого узла число узлов в левом и правом поддеревьях отличается не больше чем на 1.

```
TYPE  Node = POINTER TO RECORD                    (* ADruS441_BalancedTree *)
        key: INTEGER;  left, right: Node
    END;

VAR R: Texts.Reader; W: Texts.Writer; root: Node;

PROCEDURE tree (n: INTEGER): Node;
    (*построить идеально сбалансированное дерево с n узлами*)
    VAR new: Node;
        x, nl, nr: INTEGER;
BEGIN
    IF n = 0 THEN new := NIL
    ELSE nl := n DIV 2; nr := n-nl-1;
        NEW(new); Texts.ReadInt(R, new.key);
        new.key := x; new.left := tree(nl); new.right := tree(nr)
    END;
    RETURN new
END tree;

PROCEDURE PrintTree (t: Node; h: INTEGER);
    (*распечатать дерево t с h отступами*)
    VAR i: INTEGER;
BEGIN
    IF t # NIL THEN
```

```
        PrintTree(t.left, h+1);
        FOR i := 1 TO h DO Texts.Write(W, TAB) END;
        Texts.WriteInt(W, t.key, 6); Texts.WriteLn(W);
        PrintTree(t.right, h+1)
    END
END PrintTree;
```

Например, предположим, что имеются входные данные для дерева с 21 узлами:

```
8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18
```

Вызов root := tree(21) читает входные данные и строит идеально сбалансированное дерево, показанное на рис. 4.23. Отметим простоту и прозрачность этой программы, построенной с использованием рекурсивных процедур. Очевидно, что рекурсивные алгоритмы особенно удобны там, где нужно обрабатывать информацию, структура которой сама определена рекурсивно. Этот факт снова проявляется в процедуре, которая печатает получившееся дерево: если дерево пусто, то ничего не печатается, для поддерева на уровне L сначала печатается его левое поддерево, затем узел с отступом в L символов табуляции и, наконец, его правое поддерево.

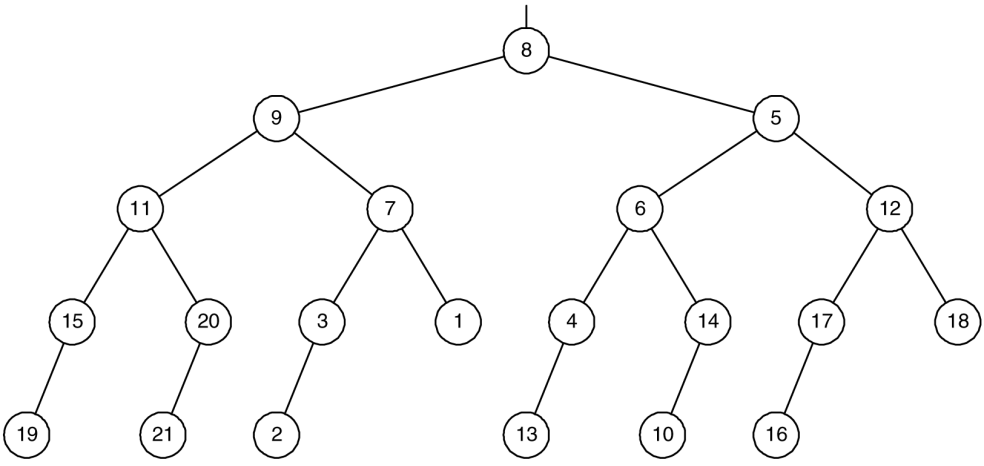


Рис. 4.23. Дерево, порожденное программой tree

4.4.2. Основные операции с двоичными деревьями

Есть много операций, которые может понадобиться выполнить с древесной структурой; например, часто нужно выполнить заданную процедуру P в каждом узле дерева. Тогда P следует считать параметром более общей задачи обхода всех узлов, или, как обычно говорят, *обхода дерева*. Если мы рассмотрим такой обход как

единый последовательный процесс, то получится, что отдельные узлы посещаются в некотором конкретном порядке и как бы расставляются в линию. Описание многих алгоритмов сильно упрощается, если обсуждать последовательность обработки элементов в терминах какого-либо отношения порядка. Из структуры деревьев естественно определяются три основных отношения порядка. Их, как и саму древесную структуру, удобно выражать на языке рекурсии. Имея в виду двоичное дерево на рис. 4.24, где R обозначает корень, а A и B – левое и правое поддеревья, три упомянутых отношения порядка таковы:

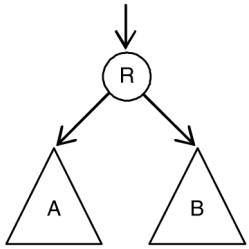


Рис. 4.24. Двоичное дерево

1. Прямой порядок (preorder): R, A, B (корень до поддеревьев)
2. Центрированный порядок (inorder): A, R, B
3. Обратный порядок (postorder): A, B, R (корень после поддеревьев)

Обходя дерево на рис. 4.20 и записывая соответствующие литеры по мере посещения узлов, получаем следующие упорядоченные последовательности:

1. Прямой порядок: * + a / b c – d * e f
2. Центрированный порядок: a + b / c * d – e * f
3. Обратный порядок: a b c / + d e f * – *

Здесь можно узнать три формы записи выражений: *прямой* обход дерева выражения дает *префиксную* нотацию, *обратный* – *постфиксную*, а *центрированный* – обычную *инфиксную*, правда, без скобок, которые нужны для уточнения приоритетов операций.

Сформулируем теперь эти три метода обхода в виде трех конкретных программ с явным параметром t, обозначающим дерево, которое нужно обработать, и с неявным параметром P, обозначающим операцию, применяемую к каждому узлу. Подразумевается следующее определение:

```
TYPE Node = POINTER TO RECORD ... left, right: Node END
```

Теперь три метода легко формулируются в виде рекурсивных процедур; этим снова подчеркивается тот факт, что операции с рекурсивно определенными структурами данных удобней всего определять в виде рекурсивных алгоритмов.

```
PROCEDURE preorder (t: Node);
BEGIN
    IF t # NIL THEN
        P(t); preorder(t.left); preorder(t.right)
    END
END preorder

PROCEDURE inorder (t: Node);
BEGIN
    IF t # NIL THEN
        inorder(t.left); P(t); inorder(t.right)
    END
END inorder
```

```
PROCEDURE postorder (t: Node);
BEGIN
    IF t # NIL THEN
        postorder(t.left); postorder(t.right); P(t)
    END
END postorder
```

Заметим, что указатель t передается по значению. Этим выражается тот факт, что передается ссылка на рассматриваемое поддерево, а не переменная, чьим значением является эта ссылка и чье значение могло бы быть изменено, если бы t передавался как параметр-переменная.

Пример обхода дерева – печать с правильным числом отступов, соответствующим уровню каждого узла.

Двоичные деревья часто используют для представления набора данных, к элементам которого нужно обращаться по уникальному ключу. Если дерево организовано таким образом, что для каждого узла t_i все ключи в его левом поддереве меньше, чем ключ узла t_i, а ключи в правом поддереве больше, чем ключ t_i, то такое дерево называют *деревом поиска*. В дереве поиска можно найти любой ключ, стартуя с корня и спускаясь в левое или правое поддерево в зависимости только от значения ключа в текущем узле. Мы видели, что n элементов можно организовать в двоичное дерево высоты всего лишь log(n). Поэтому поиск среди n элементов можно выполнить всего лишь за log(n) сравнений, если дерево идеально сбалансировано. Очевидно, дерево гораздо лучше подходит для организации такого набора данных, чем линейный список из предыдущего раздела. Так как поиск здесь проходит по единственному пути от корня к искомому узлу, его легко запрограммировать с помощью цикла:

```
PROCEDURE locate (x: INTEGER; t: Node): Node;
BEGIN
    WHILE (t # NIL) & (t.key # x) DO
        IF t.key < x THEN t := t.right ELSE t := t.left END
    END;
    RETURN t
END locate
```

Функция locate(x, t) возвращает значение NIL, если в дереве, начинающемся с корня t, не найдено узла со значением ключа x. Как и в случае поиска в списке, сложность условия завершения подсказывает лучшее решение, а именно использование барьера. Этот прием применим и при работе с деревом. Аппарат указателей позволяет использовать единственный, общий для всех ветвей узел-барьер. Получится дерево, у которого все листья привязаны к общему «якорю» (рис. 4.25). Барьер можно считать общим представителем всех внешних узлов, которыми было расширено исходное дерево (см. рис. 4.19):

```
PROCEDURE locate (x: INTEGER; t: Node): Node;
BEGIN
    s.key := x;  (*барьер*)
    WHILE t.key # x DO
```

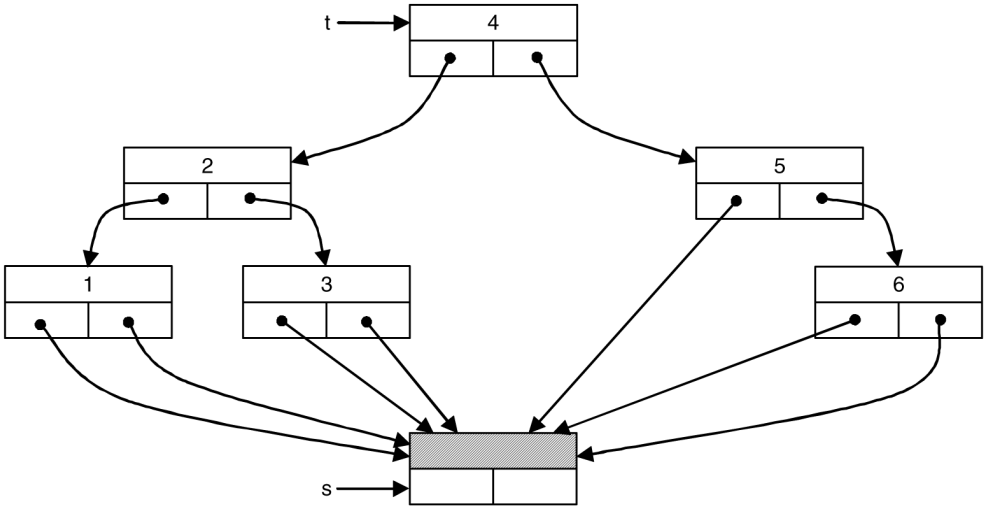


Рис. 4.25. Дерево поиска с барьером (узел s)

```
        IF t.key < x THEN t := t.right ELSE t := t.left END
    END;
    RETURN t
END locate
```

Заметим, что в этом случае locate(x, t) возвращает значение s вместо NIL, то есть указатель на барьер, если в дереве с корнем t не найдено ключа со значением x.

4.4.3. Поиск и вставка в деревьях

Вряд ли всю мощь метода динамического размещения с использованием указателей можно вполне оценить по примерам, в которых набор данных сначала строится, а в дальнейшем не меняется. Интересней примеры, в которых дерево меняется, то есть растет и/или сокращается, во время выполнения программы. Это как раз тот случай, когда оказываются непригодными другие представления данных, например массив, и когда лучшее решение получается при использовании дерева с элементами, связанными посредством указателей.

Мы сначала рассмотрим случай только растущего, но никогда не сокращающегося дерева. Типичный пример – задача составления частотного словаря, которую мы уже рассматривали в связи со связными списками. Сейчас мы рассмотрим ее снова. В этой задаче дана последовательность слов, и нужно определить число вхождений каждого слова. Это означает, что каждое слово ищется в дереве (которое вначале пусто). Если слово найдено, то счетчик вхождений увеличивается; в противном случае оно включается в дерево со счетчиком, выставленным в 1. Мы будем называть эту задачу *поиск по дереву со вставкой*. Предполагается, что определены следующие типы данных:

```
TYPE Node =  POINTER TO RECORD
                 key, count: INTEGER;
                 left, right: Node
             END
```

Как и раньше, не составляет труда найти путь поиска. Однако если он завершается тупиком (то есть приводит к пустому поддереву, обозначенному указателем со значением NIL), то данное слово нужно вставить в дерево в той точке, где было пустое поддерево. Например, рассмотрим двоичное дерево на рис. 4.26 и вставку имени Paul. Результат показан пунктирными линиями на том же рисунке.

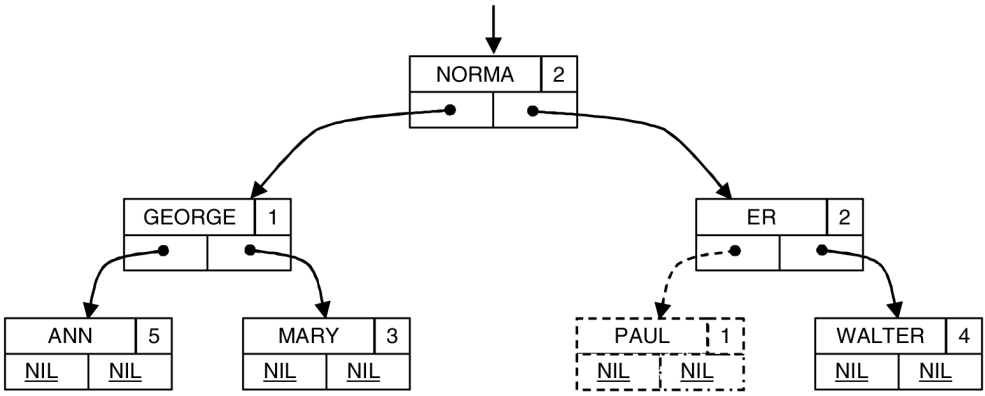


Рис. 4.26. Вставка в упорядоченное двоичное дерево

Процесс поиска формулируется в виде рекурсивной процедуры. Заметим, что ее параметр p передается по ссылке, а не по значению. Это важно, так как в случае вставки переменной, содержавшей значение NIL, нужно присвоить новое указательное значение. Для входной последовательности из 21 числа, которую мы уже использовали для построения дерева на рис. 4.23, процедура поиска и вставок дает двоичное дерево, показанное на рис. 4.27; для каждого ключа k выполняется вызов search(k, root), где root – переменная типа Node.

```
PROCEDURE PrintTree (t: Node; h: INTEGER);                (* ADruS443_Tree *)
   (*распечатать дерево t с h отступами*)
   VAR i: INTEGER;
BEGIN
   IF t # NIL THEN
      PrintTree(t.left, h+1);
      FOR i := 1 TO h DO Texts.Write(W, TAB) END;
      Texts.WriteInt(W, t.key, 6); Texts.WriteLn(W);
      PrintTree(t.right, h+1)
   END
END PrintTree;
```

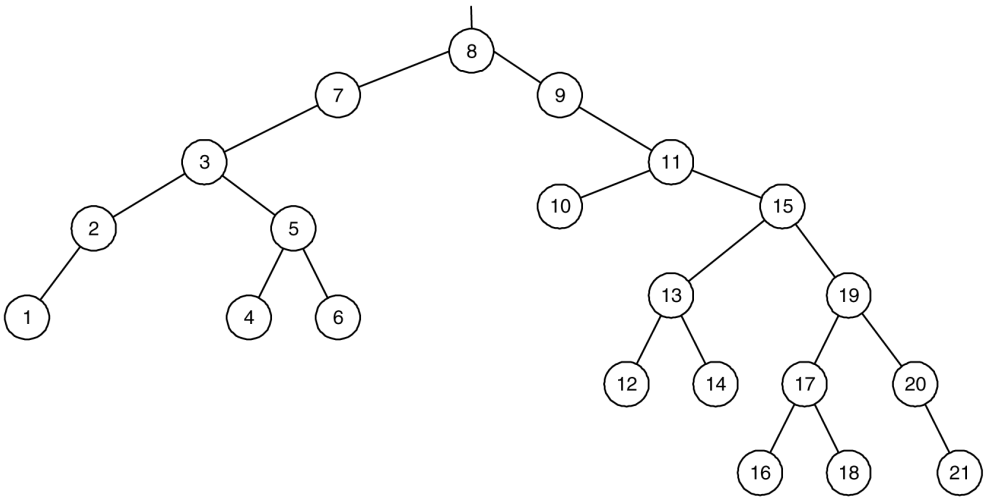


Рис. 4.27. Дерево поиска, порожденное процедурой поиска и вставки

```
PROCEDURE search (x: INTEGER; VAR p: Node);
BEGIN
    IF p = NIL THEN (*x в дереве нет; вставить*)
        NEW(p); p.key := x; p.count := 1; p.left := NIL; p.right := NIL
    ELSIF x < p.key THEN search(x, p.left)
    ELSIF x > p.key THEN search(x, p.right)
    ELSE INC(p.count)
    END
END search
```

И снова использование барьера немного упрощает программу. Ясно, что в начале программы переменная root должна быть инициализирована указателем на барьер s вместо значения NIL, а перед каждым поиском искомое значение x должно быть присвоено полю ключа барьера:

```
PROCEDURE search (x: INTEGER; VAR p: Node);
BEGIN
    IF x < p.key THEN search(x, p.left)
    ELSIF x > p.key THEN search(x, p.right)
    ELSIF p # s THEN INC(p.count)
    ELSE (*вставить*)
        NEW(p); p.key := x; p.left := s; p.right := s; p.count := 1
    END
END search
```

Хотя целью этого алгоритма был поиск, его можно использовать и для сортировки. На самом деле он очень похож на сортировку вставками, а благодаря использованию дерева вместо массива исчезает необходимость перемещать компоненты, стоящие выше точки вставки. Древесную сортировку можно запрограм-

мировать так, что она будет почти так же эффективна, как и лучшие известные методы сортировки массивов. Однако нужны некоторые предосторожности. Если обнаружено совпадение, новый элемент тоже нужно вставить. Если случай `x = p.key` обрабатывать так же, как и случай `x > p.key`, то алгоритм сортировки будет устойчивым, то есть элементы с равными ключами будут прочитаны в том же относительном порядке при просмотре дерева в нормальном порядке, в каком они вставлялись.

Вообще говоря, есть и более эффективные методы сортировки, но для приложений, где нужны как поиск, так и сортировка, можно уверенно рекомендовать алгоритм поиска и вставки. Он действительно очень часто применяется в компиляторах и банках данных для организации объектов, которые нужно хранить и искать. Хорошим примером здесь является построение указателя перекрестных ссылок для заданного текста; мы уже обращались к этому примеру для иллюстрации построения списков.

Итак, наша задача – написать программу, которая читает текст и печатает его, последовательно нумеруя строки, и одновременно собирает все слова этого текста вместе с номерами строк, в которых встретилось каждое слово. По завершении просмотра должна быть построена таблица, содержащая все собранные слова в алфавитном порядке вместе с соответствующими списками номеров строк.

Очевидно, *дерево поиска* (иначе *лексикографическое дерево*) будет прекрасным кандидатом для представления информации о словах, встречающихся в тексте. Теперь каждый узел будет не только содержать слово в качестве значения ключа, но и являться началом списка номеров строк. Каждую запись о вхождении слова в текст будем называть «элементом» (item; нехватка синонимов для перевода компенсируется кавычками – *прим. перев.*). Таким образом, в данном примере фигурируют как деревья, так и линейные списки. Программа состоит из двух главных частей, а именно из фазы просмотра и фазы печати таблицы. Последняя – простая адаптация процедуры обхода дерева; здесь при посещении каждого узла выполняется печать значения ключа (слова) вместе с соответствующим списком номеров строк. Ниже даются дополнительные пояснения о программе, приводимой далее. Таблица 4.3 показывает результаты обработки текста процедуры `search`.

1. Словом считается любая последовательность букв и цифр, начинающаяся с буквы.
2. Так как длина слов может сильно меняться, их литеры хранятся в особом массиве-буфере, а узлы дерева содержат индекс первой буквы ключа.
3. Желательно, чтобы в указателе перекрестных ссылок номера строк печатались в возрастающем порядке.

Поэтому список «элементов» должен сохранять порядок соответствующих вхождений слова в тексте. Из этого требования вытекает, что нужны два указателя в каждом узле, причем один ссылается на первый, а другой – на последний «элемент» списка. Мы предполагаем наличие глобального объекта печати `W`, а также переменной, представляющей собой текущий номер строки в тексте.

```
CONST WordLen = 32;                                    (* ADruS443_CrossRef *)

TYPE Word = ARRAY WordLen OF CHAR;
   Item =  POINTER TO RECORD  (*"элемент"*)
              lno: INTEGER; next: Item
           END;
   Node = POINTER TO RECORD
              key: Word;
              first, last: Item;  (*список*)
              left, right: Node  (*дерево*)
           END;

VAR line: INTEGER;

PROCEDURE search (VAR w: Node; VAR a: Word);
   VAR q: Item;
BEGIN
   IF w = NIL THEN (*слова в дереве нет; вставить новый узел*)
      NEW(w); NEW(q); q.lno := line;
      COPY(a, w.key); w.first := q; w.last := q; w.left := NIL; w.right := NIL
   ELSIF w.key < a THEN search(w.right, a)
   ELSIF w.key > a THEN search(w.left, a)
   ELSE (*слово в дереве*)
      NEW(q); q.lno := line; w.last.next := q; w.last := q
   END
END search;

PROCEDURE Tabulate (w: Node);
   VAR m: INTEGER; item: Item;
BEGIN
   IF w # NIL THEN
      Tabulate(w.left);
      Texts.WriteString(W, w.key); Texts.Write(W, TAB); item := w.first; m := 0;
      REPEAT
         IF m = 10 THEN
            Texts.WriteLn(W); Texts.Write(W, TAB); m := 0
         END;
         INC(m); Texts.WriteInt(W, item.lno, 6); item := item.next
      UNTIL item = NIL;
      Texts.WriteLn(W); Tabulate(w.right)
   END
END Tabulate;

PROCEDURE CrossRef (VAR R: Texts.Reader);
   VAR root: Node;
      i: INTEGER; ch: CHAR; w: Word;
   (*для вывода используется глобальный объект печати W*)
BEGIN
   root := NIL; line := 0;
   Texts.WriteInt(W, 0, 6); Texts.Write(W, TAB);
   Texts.Read(R, ch);
   WHILE ~R.eot DO
      IF ch = 0DX THEN (*конец строки*)
         Texts.WriteLn(W);
```

```
            INC(line);
            Texts.WriteInt(W, line, 6); Texts.Write(W, 9X);
            Texts.Read(R, ch)
        ELSIF ("A" <= ch) & (ch <= "Z") OR ("a" <= ch) & (ch <= "z") THEN
            i := 0;
            REPEAT
                IF i < WordLen-1 THEN w[i] := ch; INC(i) END;
                Texts.Write(W, ch); Texts.Read(R, ch)
            UNTIL (i = WordLen-1) OR ~(("A" <= ch) & (ch <= "Z")) &
                ~(("a" <= ch) & (ch <= "z")) & ~(("0" <= ch) & (ch <= "9"));
            w[i] := 0X; (*конец цепочки литер*)
            search(root, w)
        ELSE
            Texts.Write(W, ch);
            Texts.Read(R, ch)
        END;
    END;
    Texts.WriteLn(W); Texts.WriteLn(W); Tabulate(root)
END CrossRef;
```

Таблица 4.3. Пример выдачи генератора перекрестных ссылок

```
0  PROCEDURE search (x: INTEGER; VAR p: Node);
1  BEGIN
2     IF x < p.key THEN search(x, p.left)
3     ELSIF x > p^key THEN search(x, p.right)
4     ELSIF p # s THEN INC(p.count)
5     ELSE (*insert*) NEW(p);
6        p.key := x; p.left := s; p.right := s; p.count := 1
7     END
8  END
BEGIN       1
ELSE        5
ELSIF       3   4
END         7   8
IF          2
INC         4
INTEGER     0
NEW         5
Node        0
PROCEDURE   0
THEN        2   3   4
VAR         0
count       4   6
insert      5
key         2   3   6
left        2   6
p           0   2   2   3   3   4   4   5   6   6   6   6
right       3   6
s           4   6   6
search      0   2   3
x           0   2   2   3   3   6
```

4.4.4. Удаление из дерева

Обратимся теперь к операции, противоположной вставке, то есть удалению. Наша цель – построить алгоритм для удаления узла с ключом x из дерева с упорядоченными ключами. К сожалению, удаление элемента в общем случае сложнее, чем вставка. Его несложно выполнить, если удаляемый узел является концевым или имеет единственного потомка. Трудность – в удалении элемента с двумя потомками, так как нельзя заставить один указатель указывать в двух направлениях. В этой ситуации удаляемый элемент должен быть заменен либо на самый правый элемент его левого поддерева, либо на самый левый элемент его правого поддерева, причем оба этих элемента не должны иметь более одного потомка. Детали показаны ниже в рекурсивной процедуре delete. В ней различаются три случая:

1. Отсутствует компонента с ключом, равным x.
2. У компоненты с ключом x не более одного потомка.
3. У компоненты с ключом x два потомка.

```
PROCEDURE delete (x: INTEGER; VAR p: Node);          (* ADruS444_Deletion *)
   (*удалить*)
   PROCEDURE del (VAR r: Node);
   BEGIN
      IF r.right # NIL THEN
         del(r.right)
      ELSE
         p.key := r.key; p.count := r.count;
         r := r.left
      END
   END del;

BEGIN
   IF p = NIL THEN (*слова в дереве нет*)
   ELSIF x < p.key THEN delete(x, p.left)
   ELSIF x > p.key THEN delete(x, p.right)
   ELSE
      (*удалить p^:*)
      IF p.right = NIL THEN p := p.left
      ELSIF p.left  = NIL THEN p := p.right
      ELSE del(p.left)
      END
   END
END delete
```

Вспомогательная рекурсивная процедура del активируется только в случае 3. Она спускается по крайней правой ветви левого поддерева элемента q^, который должен быть удален, и затем заменяет содержимое (ключ и счетчик) записи q^ на соответствующие значения крайней правой компоненты r^ этого левого поддерева, после чего запись r^ более не нужна.

Заметим, что мы не упоминаем процедуру, которая была бы обратной для NEW и указывала бы, что память больше не нужна и ее можно использовать для других

целей. Вообще предполагается, что вычислительная система распознает ненужную более переменную по тому признаку, что никакие другие переменные больше на нее не указывают, и что поэтому к ней больше невозможно обратиться. Такой механизм называется *сборкой мусора*. Это средство не языка программирования, а, скорее, его реализации.

Рисунок 4.28 иллюстрирует работу процедуры delete. Рассмотрим дерево (a); затем последовательно удалим узлы с ключами 13, 15, 5, 10. Получающиеся деревья показаны на рис. 4.28 (b–e).

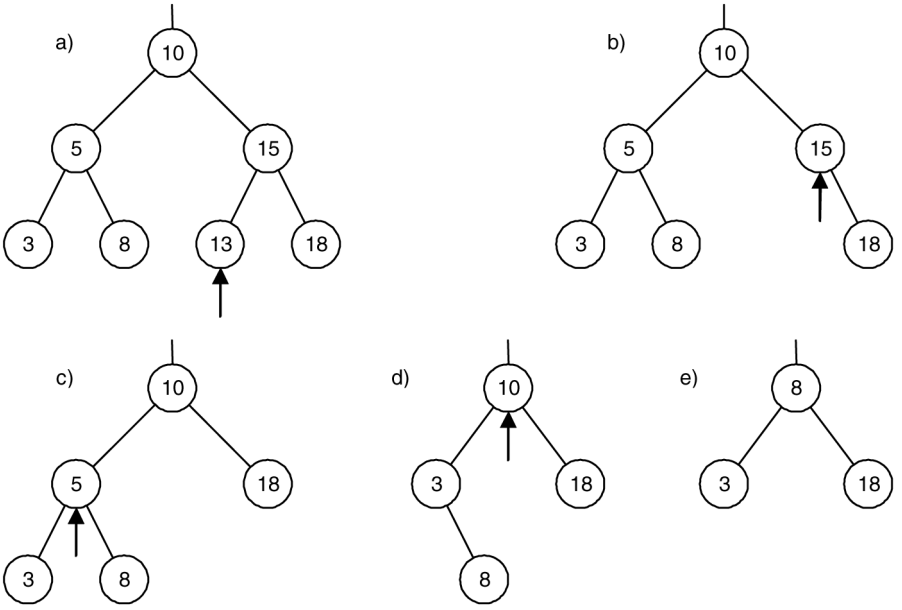


Рис. 4.28. Удаление из дерева

4.4.5. Анализ поиска по дереву со вставками

Вполне естественно испытывать подозрения в отношении алгоритма поиска по дереву со вставкой. По крайней мере, до получения дополнительных сведений о его поведении должна оставаться толика скептицизма. Многих программистов сначала беспокоит тот факт, что в общем случае мы не знаем, как будет расти дерево и какую форму оно примет. Можно только догадываться, что скорее всего оно не получится идеально сбалансированным. Поскольку среднее число сравнений, нужное для отыскания ключа в идеально сбалансированном дереве с n узлами, примерно равно log(n), число сравнений в дереве, порожденном этим алгоритмом, будет больше. Но насколько больше?

Прежде всего легко определить наихудший случай. Предположим, что все ключи поступают в уже строго возрастающем (или убывающем) порядке. Тогда

каждый ключ присоединяется непосредственно справа (слева) к своему предку, и получается полностью вырожденное дерево, то есть по сути линейный списк. Средние затраты на поиск тогда составят n/2 сравнений. В этом наихудшем случае эффективность алгоритма поиска, очевидно, очень низка, что вроде бы подтверждает наш скептицизм. Разумеется, остается вопрос, насколько вероятен этот случай. Точнее, хотелось бы знать длину a_n пути поиска, усредненную по всем n ключам и по всем n! деревьям, которые порождаются из n! перестановок n различных исходных ключей. Эта задача оказывается довольно простой и обсуждается здесь как типичный пример анализа алгоритма, а также ввиду практической важности результата.

Пусть даны n различных ключей со значениями 1, 2, ... , n. Предположим, что они поступают в случайном порядке. Вероятность для первого ключа – который становится корневым узлом – иметь значение i равна 1/n. В конечном счете его левое поддерево будет содержать i-1 узлов, а его правое поддерево – n-i узлов (см. рис. 4.29). Среднюю длину пути в левом поддереве обозначим как a_{i-1}, а в правом как a_{n-i}, снова предполагая, что все возможные перестановки остальных n-1 ключей равновероятны. Средняя длина пути в дереве с n узлами равна сумме произведений уровня каждого узла, умноженного на вероятность обращения к нему. Если предположить, что все узлы ищутся с равной вероятностью, то

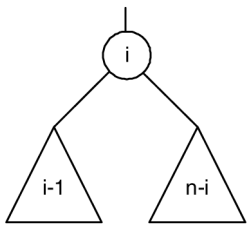


Рис. 4.29. Распределение весов по ветвям

$$a_n = (\mathbf{S}i: 1 \le i \le n: p_i) / n$$

где p_i – длина пути узла i.

В дереве на рис. 4.29 узлы разделены на три класса:

1) i-1 узлов в левом поддереве имеют среднюю длину пути a_{i-1};
2) у корня длина пути равна 0;
3) n-i узлов в правом поддереве имеют среднюю длину пути a_{n-i}.

Получаем, что вышеприведенная формула выражается как сумма членов 1 и 3:

$$a_n^{(i)} = ((i-1) * a_{i-1} + (n-i) * a_{n-i}) / n$$

Искомая величина a_n есть среднее величин $a_n^{(i)}$ по всем i = 1 ... n, то есть по всем деревьям с ключами 1, 2, ... , n в корне:

$$\begin{aligned} a_n &= (\mathbf{S}i: 1 \le i \le n: (i-1)\, a_{i-1} + (n-i)\, a_{n-i}) / n^2 \\ &= 2 * (\mathbf{S}i: 1 \le i \le n: (i-1)\, a_{i-1}) / n^2 \\ &= 2 * (\mathbf{S}i: 1 \le i < n: i * a_i) / n^2 \end{aligned}$$

Это уравнение является рекуррентным соотношением вида $a_n = f_1(a_1, a_2, \ldots, a_{n-1})$. Из него получим более простое рекуррентное соотношение вида $a_n = f_2(a_{n-1})$. Следующее выражение (1) получается выделением последнего члена, а (2) – подстановкой n-1 вместо n:

(1) $a_n = 2*(n-1) * a_{n-1} / n^2 + 2 * (\mathbf{S}i: 1 \le i < n-1: i * a_i) / n^2$
(2) $a_{n-1} = 2 * (\mathbf{S}i: 1 \le i < n-1: i * a_i) / (n-1)^2$

Умножая (2) на $(n-1)^2/n^2$, получаем:

(3) $2 * (\mathbf{S}i: 1 \le i < n-1: i * a_i) / n^2 = a_{n-1} * (n-1)^2/n^2$

Подставляя правую часть уравнения (3) в (1), находим:

$a_n = 2 * (n-1) * a_{n-1} / n^2 + a_{n-1} * (n-1)^2 / n^2 = a_{n-1} * (n-1)^2 / n^2$

В итоге получается, что a_n допускает нерекурсивное замкнутое выражение через гармоническую сумму:

$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$
$a_n = 2 * (H_n * (n+1)/n - 1)$

Из формулы Эйлера (используя константу Эйлера $g = 0.577...$)

$H_n = g + \ln n + 1/12n^2 + \ldots$

выводим приближенное значение для больших n:

$a_n = 2 * (\ln n + g - 1)$

Так как средняя длина пути в идеально сбалансированном дереве примерно равна

$a_n' = \log n - 1$

то, пренебрегая константами, не существенными при больших n, получаем:

$\lim (a_n/a_n') = 2 * \ln(n) / \log(n) = 2 \ln(2) = 1.386...$

Чему учит нас этот результат? Он говорит нам, что усилия, направленные на то, чтобы всегда строить идеально сбалансированное дерево вместо случайного, позволяют нам ожидать – всегда предполагая, что все ключи ищутся с равной вероятностью, – среднее сокращение длины пути поиска самое большее на 39%. Нужно подчеркнуть слово среднее, поскольку сокращение может, конечно, быть намного больше в том неудачном случае, когда создаваемое дерево полностью выродилось в список, вероятность чего, однако, очень низка. В этой связи стоит отметить, что средняя длина пути случайного дерева тоже растет строго логарифмически с ростом числа его узлов, несмотря на то что длина пути для наихудшего случая растет линейно.

Число 39% накладывает ограничение на объем дополнительных усилий, которые можно с пользой потратить на какую-либо реорганизацию дерева после вставки ключей. Естественно, полезная отдача от таких усилий сильно зависит от отношения числа обращений к узлам (извлечение информации) к числу вставок (обновлений). Чем выше это отношение, тем больше выигрыш от процедуры реорганизации. Число 39% достаточно мало, чтобы в большинстве приложений улучшение простого алгоритма вставки в дерево не оправдывало себя, за исключением случаев, когда велики число узлов и отношение числа обращений к числу вставок.

4.5. Сбалансированные деревья

Из предыдущего обсуждения ясно, что процедура вставки, всегда восстанавливающая идеальную сбалансированность дерева, вряд ли может быть полезной, так как восстановление идеального баланса после случайной вставки – операция довольно нетривиальная. Возможный путь повышения эффективности – попытаться найти менее жесткое определение баланса. Такой неидеальный критерий должен приводить к более простым процедурам реорганизации дерева за счет незначительного ухудшения средней эффективности поиска. Одно такое определение сбалансированности было дано Адельсоном-Вельским и Ландисом [4.1]. Вот оно:

> Дерево называется *сбалансированным* в том и только в том случае, когда для каждого узла высота двух его поддеревьев отличается не более чем на 1.

Деревья, удовлетворяющие этому условию, часто называют по именам их изобретателей *АВЛ-деревьями*. Мы будем называть их просто сбалансированными, так как этот критерий оказался в высшей степени удачным. (Заметим, что все идеально сбалансированные деревья также являются АВЛ-деревьями.)

Это определение не только само простое, но и приводит к не слишком сложной процедуре балансировки, а средняя длина поиска здесь практически не отличается от случая идеально сбалансированных деревьев. Следующие операции могут выполняться для сбалансированных деревьев за время порядка O(log n) даже в наихудшем случае:

1) поиск узла с заданным ключом;
2) вставка узла с заданным ключом;
3) удаление узла с заданным ключом.

Эти утверждения суть прямые следствия теоремы, доказанной Адельсоном-Вельским и Ландисом, которая гарантирует, что сбалансированное дерево не более чем на 45% превосходит по высоте его идеально сбалансированный вариант, независимо от числа узлов. Если обозначить высоту сбалансированного дерева с n узлами как $h_b(n)$, то

$$\log(n+1) < h_b(n) < 1.4404*\log(n+2) - 0.328$$

Ясно, что оптимум достигается для идеально сбалансированного дерева с $n = 2k-1$. Но какова структура наихудшего АВЛ-дерева? Чтобы найти максимальную высоту h всех сбалансированных деревьев с n узлами, фиксируем высоту h и попытаемся построить сбалансированное дерево с минимальным числом узлов. Эта стратегия предпочтительна потому, что, как и в случае с минимальной высотой, это значение высоты может быть получено только для некоторых конкретных значений n. Обозначим такое дерево высоты h как T_h. Очевидно, что T_0 – пустое дерево, а T_1 – дерево с одним узлом. Чтобы построить дерево T_h для $h > 1$, присоединим к корню два поддерева, у которых число узлов снова минимально. Поэтому поддеревья принадлежат этому же классу T. Очевидно, одно поддерево должно иметь высоту h-1, а другое тогда может иметь высоту на единицу меньше, то есть h-2. Рисунок 4.30 показывает деревья с высотой 2, 3 и 4. Поскольку прин-

цип их построения сильно напоминает определение чисел Фибоначчи, их называют *деревьями Фибоначчи* (см. рис. 4.30). Определяются они так:

1. Пустое дерево есть дерево Фибоначчи высоты 0.
2. Дерево с одним узлом есть дерево Фибоначчи высоты 1.
3. Если T_{h-1} и T_{h-2} – деревья Фибоначчи высоты $h-1$ и $h-2$ соответственно, то $T_h = <T_{h-1}, x, T_{h-2}>$ – дерево Фибоначчи.
4. Других деревьев Фибоначчи нет.

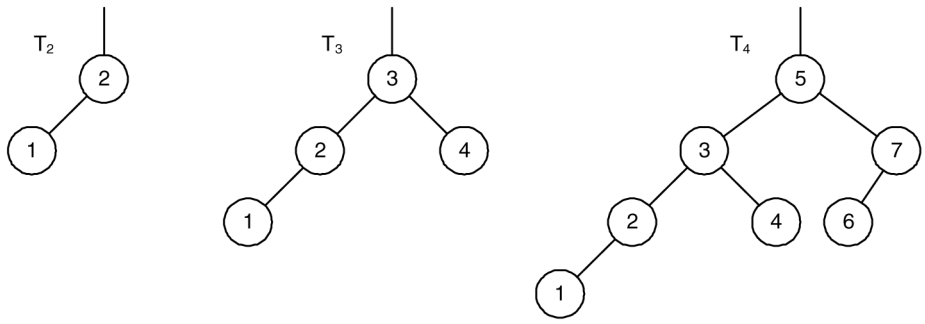


Рис. 4.30. Деревья Фибоначчи высоты 2, 3 и 4

Число узлов в T_h определяется следующим простым рекуррентным соотношением:

$N_0 = 0, \quad N_1 = 1$
$N_h = N_{h-1} + 1 + N_{h-2}$

N_h – это число узлов, для которого может реализоваться наихудший случай (верхний предел для h); такие числа называются *числами Леонардо*.

4.5.1. Вставка в сбалансированное дерево

Теперь посмотрим, что происходит, когда в сбалансированное дерево вставляется новый узел. Если r – корень с левым и правым поддеревьями L и R, то могут иметь место три случая. Пусть новый узел вставляется в L, увеличивая его высоту на 1:

1. $h_L = h_R$: высота L и R становится неравной, но критерий сбалансированности не нарушается.
2. $h_L < h_R$: высота L и R становится равной, то есть баланс только улучшается.
3. $h_L > h_R$: критерий сбалансированности нарушается, и дерево нужно перестраивать.

Рассмотрим дерево на рис. 4.31. Узлы с ключами 9 или 11 можно вставить без перестройки: поддерево с корнем 10 станет асимметричным (случай 1), а у дерева с корнем 8 баланс улучшится (случай 2). Однако вставка узлов 1, 3, 5 или 7 потребует перестройки для восстановления баланса.

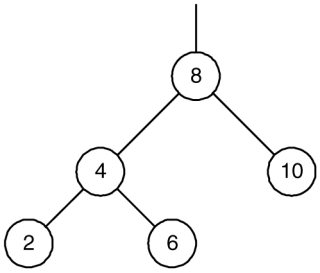


Рис. 4.31. Сбалансированное дерево

Пристально рассматривая ситуацию, обнаруживаем, что есть только две существенно разные конфигурации, требующие раздельного анализа. Остальные сводятся к этим двум по соображениям симметрии. Случай 1 соответствует вставке ключей 1 или 3 в дерево на рис. 4.31, случай 2 – вставке ключей 5 или 7.

Эти два случая изображены в более общем виде на рис. 4.32, где прямоугольники обозначают поддеревья, а увеличение высоты за счет вставки указано крестиками. Желаемый баланс восстанавливается простыми преобразованиями. Их результат показан на рис. 4.33; заметим, что разрешены только перемещения по вертикали, а относительное горизонтальное положение показанных узлов и поддеревьев не должно меняться.

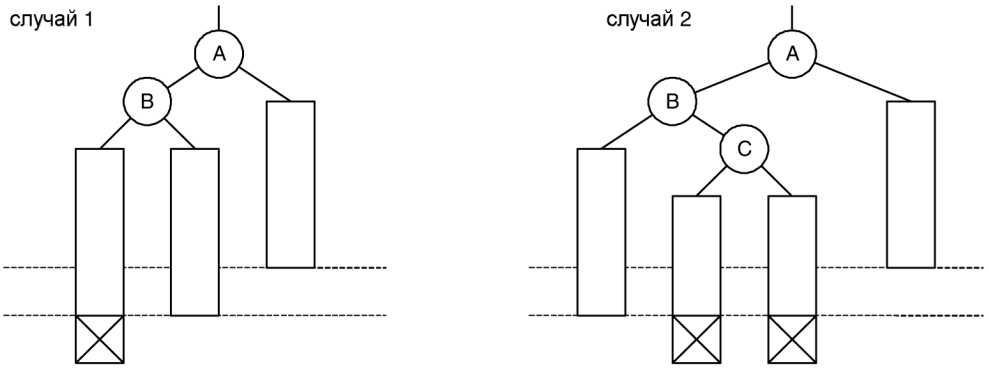


Рис. 4.32. Нарушение баланса в результате вставки

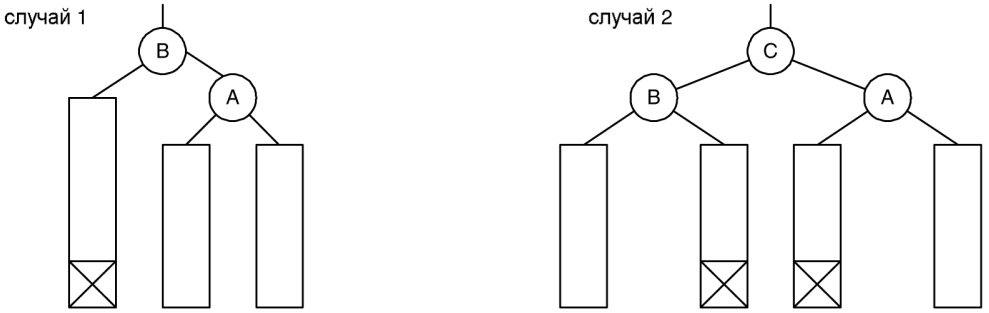


Рис. 4.33. Восстановление баланса

Алгоритм вставки и балансировки критически зависит от способа хранения информации о сбалансированности дерева. Одна крайность – вообще не хранить эту информацию в явном виде. Но тогда баланс узла должен быть заново вычислен каждый раз, когда он может измениться при вставке, что приводит к чрезмерным накладным расходам. Другая крайность – в явном виде хранить соответствующий баланс в каждом узле. Тогда определение типа Node дополняется следующим образом:

```
TYPE Node = POINTER TO RECORD
                key, count, bal: INTEGER;  (*bal = -1, 0, +1*)
                left, right: Node
            END
```

В дальнейшем балансом узла будем называть разность высоты его правого и левого поддеревьев и положим приведенное определение типа узла в основу построения алгоритма. Процесс вставки узла состоит из трех последовательных шагов:

1) пройти по пути поиска, пока не выяснится, что данного ключа в дереве еще нет;
2) вставить новый узел и определить получающийся баланс;
3) вернуться по пути поиска и проверить баланс в каждом узле. Если нужно, выполнять балансировку.

Хотя этот способ требует избыточных проверок (когда баланс установлен, его не нужно проверять для предков данного узла), мы будем сначала придерживаться этой очевидно корректной схемы, так как ее можно реализовать простым расширением уже построенной процедуры поиска и вставки. Эта процедура описывает нужную операцию поиска для каждого отдельного узла, и благодаря ее рекурсивной формулировке в нее легко добавить дополнительные действия, выполняемые при возвращении по пути поиска. На каждом шаге должна передаваться информация о том, увеличилась или нет высота поддерева, в котором сделана вставка. Поэтому мы расширим список параметров процедуры булевским параметром h со значением *высота поддерева увеличилась*. Ясно, что это должен быть параметр-переменная, так как через него передается некий результат.

Теперь предположим, что процесс возвращается в некий узел p^ (A на рис. 4.32 – *прим. перев.*) из левой ветви с указанием, что ее высота увеличилась. Тогда нужно различать три случая соотношения высот поддеревьев до вставки:

1) $h_L < h_R$, p.bal = +1; дисбаланс в p исправляется вставкой;
2) $h_L = h_R$, p.bal = 0; после вставки левое поддерево перевешивает;
3) $h_L > h_R$, p.bal = -1; нужна балансировка.

В третьем случае изучение баланса корня (B на рис. 4.32 – *прим. перев.*) левого поддерева (скажем, p1.bal) покажет, какой из случаев 1 или 2 на рис. 4.32 имеет место. Если у того узла тоже левое поддерево выше, чем правое, то мы имеем дело со случаем 1, иначе со случаем 2. (Убедитесь, что в этом случае не может встретиться левое поддерево с нулевым балансом в корне.) Необходимые операции ба-

лансировки полностью выражаются в виде нескольких присваиваний указателей. На самом деле имеет место циклическая перестановка указателей, приводящая к одиночной или двойной «ротации» двух или трех участвующих узлов. Кроме ротации указателей, нужно обновить и показатели баланса соответствующих узлов. Детали показаны в процедурах поиска, вставки и балансировки.

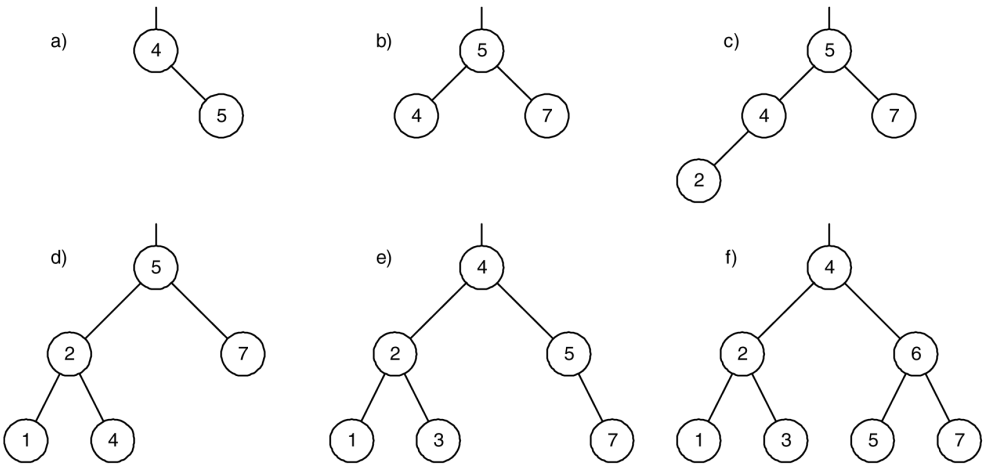


Рис. 4.34. Вставки в сбалансированное дерево

Работа алгоритма показана на рис. 4.34. Рассмотрим двоичное дерево (a), состоящее только из двух узлов. Вставка ключа 7 дает сначала несбалансированное дерево (то есть линейный список). Его балансировка требует одиночной RR-ротации, приводящей к идеально сбалансированному дереву (b). Дальнейшая вставка узлов 2 и 1 приводит к разбалансировке поддерева с корнем 4. Это поддерево балансируется одиночной LL-ротацией (d). Следующая вставка ключа 3 тут же нарушает баланс в корневом узле 5. После чего баланс восстанавливается более сложной двойной LR-ротацией; результат – дерево (e). После следующей вставки баланс может нарушиться только в узле 5. В самом деле, вставка узла 6 должна приводить к четвертому случаю балансировки из описанных ниже, то есть двойной RL-ротации. Окончательный вид дерева показан на рис. 4.34 (f).

```
PROCEDURE search (x: INTEGER; VAR p: Node; VAR h: BOOLEAN);
   VAR p1, p2: Node;                                    (* ADruS45_AVL *)

BEGIN
   (*~h*)
   IF p = NIL THEN (*вставка*)
      NEW(p); p.key := x; p.count := 1; p.left := NIL; p.right := NIL; p.bal := 0;
      h := TRUE;
   ELSIF p.key > x THEN
      search(x, p.left, h);
```

```
        IF h THEN  (*выросла левая ветвь*)
            IF p.bal = 1 THEN p.bal := 0; h := FALSE
            ELSIF p.bal = 0 THEN p.bal := -1
            ELSE (*bal = -1, восстановить баланс*) p1 := p.left;
                IF p1.bal = -1 THEN  (*одиночная LL-ротация*)
                    p.left := p1.right; p1.right := p;
                    p.bal := 0; p := p1
                ELSE (*двойная LR-ротация*) p2 := p1.right;
                    p1.right := p2.left; p2.left := p1;
                    p.left := p2.right; p2.right := p;
                    IF p2.bal = -1 THEN p.bal := 1 ELSE p.bal := 0 END;
                    IF p2.bal = +1 THEN p1.bal := -1 ELSE p1.bal := 0 END;
                    p := p2
                END;
                p.bal := 0; h := FALSE
            END
        END
    ELSIF p.key < x THEN
        search(x, p.right, h);
        IF h THEN  (*выросла правая ветвь*)
            IF p.bal = -1 THEN p.bal := 0; h := FALSE
            ELSIF p.bal = 0 THEN p.bal := 1
            ELSE (*bal = +1, восстановить баланс*) p1 := p.right;
                IF p1.bal = 1 THEN  (*одиночная RR-ротация*)
                    p.right := p1.left; p1.left := p;
                    p.bal := 0; p := p1
                ELSE (*двойная RL-ротация*) p2 := p1.left;
                    p1.left := p2.right; p2.right := p1;
                    p.right := p2.left; p2.left := p;
                    IF p2.bal = +1 THEN p.bal := -1 ELSE p.bal := 0 END;
                    IF p2.bal = -1 THEN p1.bal := 1 ELSE p1.bal := 0 END;
                    p := p2
                END;
                p.bal := 0; h := FALSE
            END
        END
    ELSE INC(p.count)
    END
END search
```

Возникает два особенно интересных вопроса касательно производительности алгоритма вставки в сбалансированные деревья:

1. Если все n! перестановок n ключей встречаются с одинаковой вероятностью, какова будет средняя высота получающегося сбалансированного дерева?
2. С какой вероятностью вставка приводит к необходимости балансировки?

Задача математического исследования этого сложного алгоритма остается нерешенной. Эмпирические тесты подтверждают предположение, что средняя высота сбалансированного дерева, построенного таким способом, равна h = log(n)+c, где c –

небольшая константа ($c \approx 0.25$). Это означает, что на практике АВЛ-деревья так же хороши, как и идеально сбалансированные деревья, хотя работать с ними гораздо проще. Эмпирические данные также показывают, что в среднем одна балансировка нужна примерно на каждые две вставки. Здесь одиночные и двойные ротации равновероятны. Разумеется, пример на рис. 4.34 был тщательно подобран, чтобы показать наибольшее число ротаций при минимуме вставок.

Сложность действий по балансировке говорит о том, что использовать сбалансированные деревья следует только тогда, когда поиск информации производится значительно чаще, чем вставки. Тем более что с целью экономии памяти узлы таких деревьев поиска обычно реализуются как плотно упакованные записи. Поэтому скорость изменения показателей баланса, требующих только двух битов каждый, часто решающим образом влияет на эффективность балансировки. Эмпирические оценки показывают, что привлекательность сбалансированных деревьев сильно падает, если записи должны быть плотно упакованы. Так что превзойти простейший алгоритм вставки в дерево оказывается нелегко.

4.5.2. Удаление из сбалансированного дерева

Наш опыт с удалением из деревьев подсказывает, что и в случае сбалансированных деревьев удаление будет более сложной операцией, чем вставка. Это на самом деле так, хотя операция балансировки остается в сущности такой же, как и для вставки. В частности, здесь балансировка тоже состоит из одиночных или двойных ротаций узлов.

Процедура удаления из сбалансированного дерева строится на основе алгоритма удаления из обычного дерева. Простые случаи – концевые узлы и узлы с единственным потомком. Если удаляемый узел имеет два поддерева, мы снова будет заменять его самым правым узлом его левого поддерева. Как и в случае вставки, добавляется булевский параметр-переменная h со значением *высота поддерева уменьшилась*. Необходимость в балансировке может возникнуть, только если h истинно. Это случается при обнаружении и удалении узла, или если сама балансировка уменьшает высоту поддерева. Здесь мы введем две процедуры для (симметричных) операций балансировки, так как их нужно вызывать из более чем одной точки алгоритма удаления. Заметим, что процедуры balanceL или balanceR вызываются после того, как уменьшилась высота левой или правой ветви соответственно.

Работа процедуры проиллюстрирована на рис. 4.35. Если дано сбалансированное дерево (a), то последовательное удаление узлов с ключами 4, 8, 6, 5, 2, 1, и 7 приводит к деревьям (b)...(h). Удаление ключа 4 само по себе просто, так как соответствующий узел концевой. Но это приводит к несбалансированному узлу 3. Его балансировка требует одиночной LL-ротации. Балансировка опять нужна после удаления узла 6. На этот раз правое поддерево для корня (7) балансируется одиночной RR-ротацией. Удаление узла 2, само по себе бесхитростное, так как узел имеет лишь одного потомка, требует применения сложной двойной RL-ротации. Наконец, четвертый случай, двойная LR-ротация, вызывается после удаления

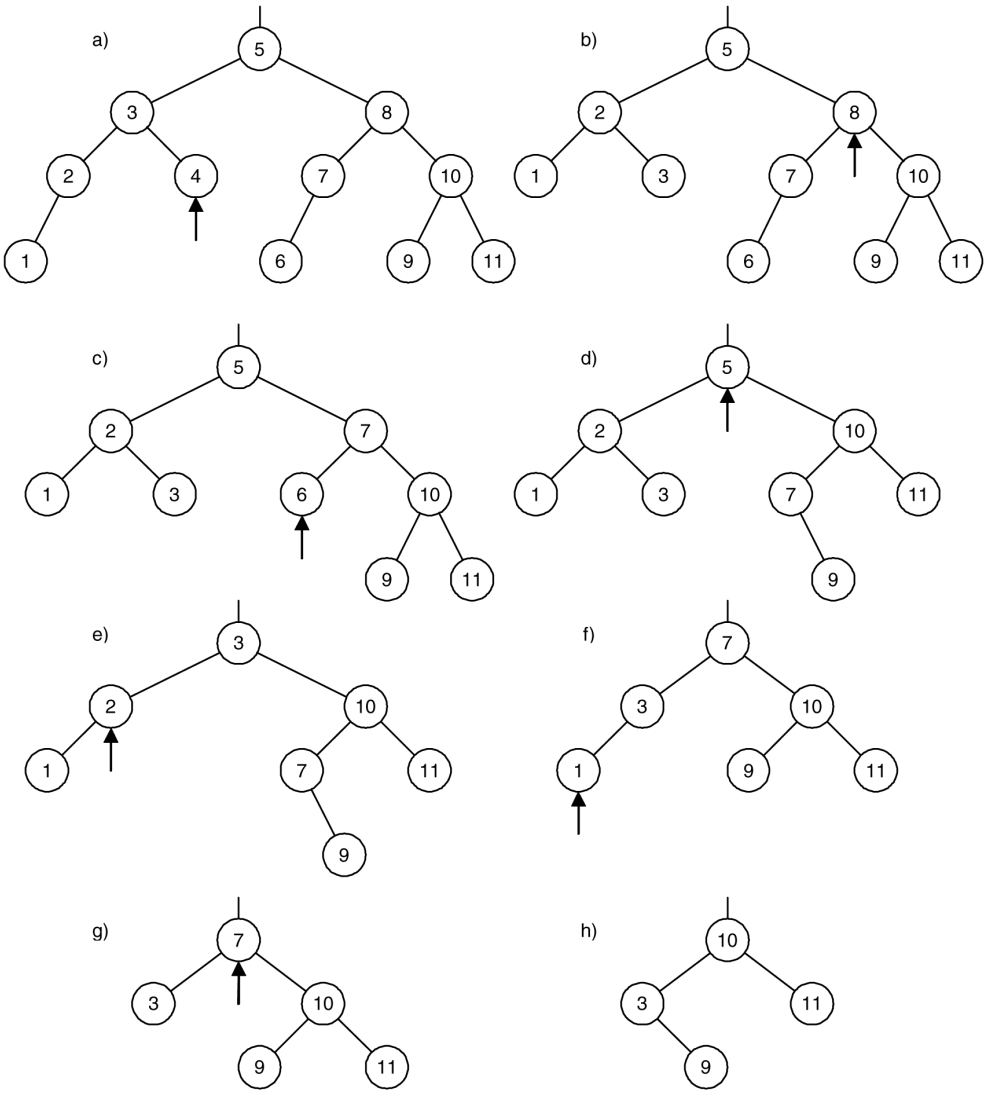


Рис. 4.35. Удаления из сбалансированного дерева

узла 7, который сначала замещается крайним правым элементом своего левого поддерева, то есть узлом 3.

```
PROCEDURE balanceL (VAR p: Node; VAR h: BOOLEAN);          (*ADruS45_AVL*)
   VAR p1, p2: Node;
BEGIN
   (*h; уменьшилась левая ветвь*)
   IF p.bal = -1 THEN p.bal := 0
```

```
    ELSIF p.bal = 0 THEN p.bal := 1; h := FALSE
    ELSE (*bal = 1, восстановить баланс*) p1 := p.right;
      IF p1.bal >= 0 THEN  (*одиночная RR-ротация*)
        p.right := p1.left; p1.left := p;
        IF p1.bal = 0 THEN p.bal := 1; p1.bal := -1; h := FALSE
        ELSE p.bal := 0; p1.bal := 0
        END;
        p := p1
      ELSE (*двойная RL-ротация*)
        p2 := p1.left;
        p1.left := p2.right; p2.right := p1;
        p.right := p2.left; p2.left := p;
        IF p2.bal = +1 THEN p.bal := -1 ELSE p.bal := 0 END;
        IF p2.bal = -1 THEN p1.bal := 1 ELSE p1.bal := 0 END;
        p := p2; p2.bal := 0
      END
    END
END balanceL;

PROCEDURE balanceR (VAR p: Node; VAR h: BOOLEAN);
    VAR p1, p2: Node;
BEGIN
    (*h; уменьшилась правая ветвь*)
    IF p.bal = 1 THEN p.bal := 0
    ELSIF p.bal = 0 THEN p.bal := -1; h := FALSE
    ELSE (*bal = -1, rebalance*) p1 := p.left;
      IF p1.bal <= 0 THEN (*одиночная LL-ротация*)
        p.left := p1.right; p1.right := p;
        IF p1.bal = 0 THEN p.bal := -1; p1.bal := 1; h := FALSE
        ELSE p.bal := 0; p1.bal := 0
        END;
        p := p1
      ELSE (*двойная LR-ротация*)
        p2 := p1.right;
        p1.right := p2.left; p2.left := p1;
        p.left := p2.right; p2.right := p;
        IF p2.bal = -1 THEN p.bal := 1 ELSE p.bal := 0 END;
        IF p2.bal = +1 THEN p1.bal := -1 ELSE p1.bal := 0 END;
        p := p2; p2.bal := 0
      END
    END
END balanceR;

PROCEDURE delete (x: INTEGER; VAR p: Node; VAR h: BOOLEAN);
    VAR q: Node;

    PROCEDURE del (VAR r: Node; VAR h: BOOLEAN);
    BEGIN
```

```
        (*~h*)
        IF r.right # NIL THEN
            del(r.right, h);
            IF h THEN balanceR(r, h) END
        ELSE
            q.key := r.key; q.count := r.count;
            q := r; r := r.left; h := TRUE
        END
    END del;

BEGIN
    (*~h*)
    IF p = NIL THEN (*ключа нет в дереве*)
    ELSIF p.key > x THEN
        delete(x, p.left, h);
        IF h THEN balanceL(p, h) END
    ELSIF p.key < x THEN
        delete(x, p.right, h);
        IF h THEN balanceR(p, h) END
    ELSE (*удалить p^*)
        q := p;
        IF q.right = NIL THEN p := q.left; h := TRUE
        ELSIF q.left = NIL THEN p := q.right; h := TRUE
        ELSE
            del(q.left, h);
            IF h THEN balanceL(p, h) END
        END
    END
END delete
```

К счастью, удаление элемента из сбалансированного дерева тоже может быть выполнено – в худшем случае – за O(log n) операций. Однако нельзя игнорировать существенную разницу в поведении процедур вставки и удаления. В то время как вставка единственного ключа может потребовать самое большее одной ротации (двух или трех узлов), удаление может потребовать ротации в каждом узле пути поиска. Например, рассмотрим удаление крайнего правого узла дерева Фибоначчи. В этом случае удаление любого одного узла приводит к уменьшению высоты дерева; кроме того, удаление крайнего правого узла требует максимального числа ротаций. Здесь имеет место весьма неудачная комбинация – наихудший выбор узла в наихудшей конфигурации сбалансированного дерева. А насколько вероятны ротации в общем случае?

Удивительный результат эмпирических тестов состоит в том, что хотя одна ротация требуется примерно на каждые две вставки, лишь одна ротация потребуется на пять удалений. Поэтому в сбалансированных деревьях удаление элемента является столь же легкой (или столь же трудоемкой) операцией, как и вставка.

4.6. Оптимальные деревья поиска

До сих пор наш анализ организации деревьев поиска опирался на предположение, что частота обращений ко всем узлам одинакова, то есть что все ключи с равной вероятностью могут встретиться в качестве аргумента поиска. Вероятно, это лучшая гипотеза, если о распределении вероятностей обращений к ключам ничего не известно. Однако бывают случаи (хотя они, скорее, исключения, чем правило), когда такая информация есть. Для подобных случаев характерно, что ключи всегда одни и те же, то есть структура дерева поиска не меняется, то есть не выполняются ни вставки, ни удаления. Типичный пример – лексический анализатор (сканер) компилятора, который для каждого слова (идентификатора) определяет, является ли оно ключевым (зарезервированным) словом. В этом случае статистическое исследование сотен компилируемых программ может дать точную информацию об относительных частотах появления таких слов и, следовательно, обращений к ним в дереве поиска.

Предположим, что в дереве поиска вероятность обращения к ключу i равна

$$Pr\{x = k_i\} = p_i, \quad (\mathbf{S}i: 1 \le i \le n : p_i) = 1$$

Мы сейчас хотим организовать дерево поиска так, чтобы полное число шагов поиска – для достаточно большого числа попыток – было минимальным. Для этого изменим определение длины пути, (1) приписывая каждому узлу некоторый вес и (2) считая, что корень находится на уровне 1 (а не 0), поскольку с ним связано первое сравнение на пути поиска. Узлы, к которым обращений много, становятся тяжелыми, а те, которые посещаются редко, – легкими. Тогда *взвешенная длина путей* (внутренних) равна сумме всех путей из корня до каждого узла, взвешенных с вероятностью обращения к этому узлу:

$$P = \mathbf{S}i: 1 \le i \le n : p_i * h_i$$

h_i – уровень узла i. Теперь наша цель – минимизировать взвешенную длину путей для заданного распределения вероятностей. Например, рассмотрим набор ключей 1, 2, 3 с вероятностями обращения $p1 = 1/7$, $p2 = 2/7$ и $p3 = 4/7$. Из этих трех ключей дерево поиска можно составить пятью разными способами (см. рис. 4.36).

Из определения можно вычислить взвешенные длины путей деревьев (a)–(e):

$$P(a) = 11/7, \quad P(b) = 12/7, \quad P(c) = 12/7, \quad P(d) = 15/7, \quad P(e) = 17/7$$

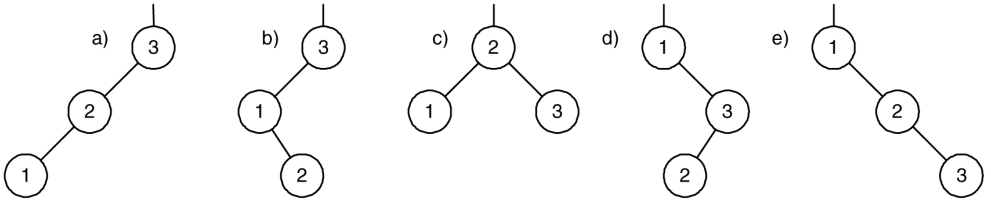


Рис. 4.36. Деревья поиска с 3 узлами

Таким образом, в этом примере оптимальным оказывается не идеально сбалансированное дерево (c), а вырожденное дерево (a).

Пример сканера компилятора сразу наводит на мысль, что задачу нужно немного обобщить: слова, встречающиеся в исходном тексте, не всегда являются ключевыми; на самом деле ключевые слова – скорее исключения. Обнаружение того факта, что некое слово k не является ключом в дереве поиска, можно рассматривать как обращение к так называемому дополнительному узлу, вставленному между ближайшими меньшим и большим ключами (см. рис. 4.19), для которого определена длина внешнего пути. Если вероятность q_i того, что аргумент поиска x лежит между этими двумя ключами k_i и k_{i+1}, тоже известна, то эта информация может существенно изменить структуру оптимального дерева поиска. Поэтому, обобщая задачу, будем учитывать также и безуспешные попытки поиска. Теперь общая взвешенная длина путей равна

$$P = (\mathbf{S}i: 1 \le i \le n : p_i * h_i) + (\mathbf{S}i: 1 \le i \le m : q_i * h'_i)$$

где

$$(\mathbf{S}i: 1 \le i \le n : p_i) + (\mathbf{S}i: 1 \le i \le m : q_i) = 1$$

и где h_i – уровень (внутреннего) узла i, а h'_j – уровень внешнего узла j. Тогда среднюю взвешенную длину пути можно назвать *ценой* дерева поиска, поскольку она является мерой ожидаемых затрат на поиск. Дерево поиска с минимальной ценой среди всех деревьев с заданным набором ключей k_i и вероятностей p_i и q_i называется *оптимальным деревом*.

Чтобы найти оптимальное дерево, не нужно требовать, чтобы сумма всех чисел p или q равнялась 1. На самом деле эти вероятности обычно определяются из экспериментов, где подсчитывается число обращений к каждому узлу. В дальнейшем

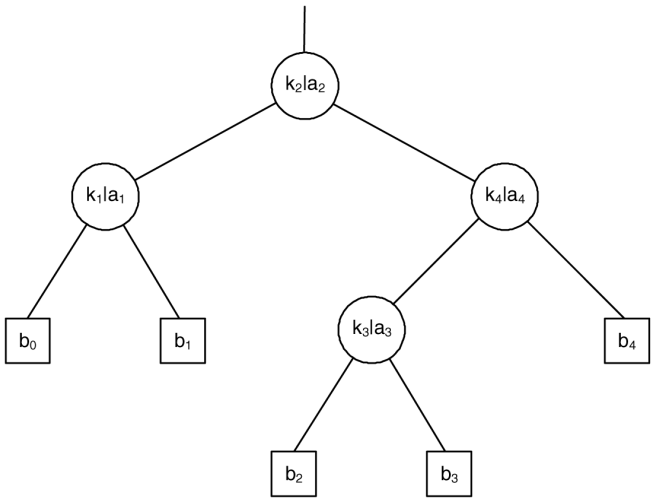


Рис. 4.37. Дерево поиска с частотами обращений

вместо вероятностей p_i и q_j мы будем использовать такие счетчики и обозначать их следующим образом:

a_i = сколько раз агрумент поиска x был равен k_i
b_j = сколько раз агрумент поиска x оказался между k_j и k_{j+1}

По определению, b_0 – число случаев, когда x оказался меньше k_1, а b_n – когда x был больше k_n (см. рис. 4.37). В дальнейшем мы будем использовать P для обозначения полной взвешенной длины путей вместо средней длины пути:

$$P = (\mathbf{S}i: 1 \le i \le n : a_i * h_i) + (\mathbf{S}i: 1 \le i \le m : b_i * h'_i)$$

Таким образом, в дополнение к тому, что уже не нужно вычислять вероятности по измеренным частотам, получаем дополнительный бонус: при поиске оптимального дерева можно работать с целыми числами вместо дробных.

Учитывая, что число возможных конфигураций n узлов растет экспоненциально с n, задача нахождения оптимума для больших n кажется безнадежной. Однако оптимальные деревья обладают одним важным свойством, которое помогает в их отыскании: все их поддеревья также оптимальны. Например, если дерево на рис. 4.37 оптимально, то поддерево с ключами k_3 и k_4 также оптимально. Это свойство подсказывает алгоритм, который систематически строит все большие деревья, начиная с отдельных узлов в качестве наименьших возможных поддеревьев. При этом дерево растет от листьев к корню, то есть, учитывая, что мы рисуем деревья вверх ногами, в направлении снизу вверх [4.6].

Уравнение, представляющее собой ключ к этому алгоритму, выводится так. Пусть P будет взвешенной длиной путей некоторого дерева, и пусть P_L и P_R – соответствующие длины для левого и правого поддеревьев его корня. Ясно, что P равна сумме P_L, P_R, а также количества проходов поиска по ребру, ведущему к корню, что просто равно полному числу попыток поиска W. Будем называть W весом дерева. Тогда его средняя длина путей равна P/W:

$$P = P_L + W + P_R$$
$$W = (\mathbf{S}i: 1 \le i \le n : a_i) + (\mathbf{S}i: 1 \le i \le m : b_i)$$

Эти соображения показывают, что нужно как-то обозначить веса и длины путей поддеревьев, состоящих из некоторого числа смежных ключей. Пусть T_{ij} – оптимальное поддерево, состоящее из узлов с ключами $k_{i+1}, k_{i+2}, \ldots, k_j$. Тогда пусть w_{ij} обозначает вес, а p_{ij} – длину путей поддерева T_{ij}. Ясно, что $P = p_{0,n}$ и $W = w_{0,n}$. Все эти величины определяются следующими рекуррентными соотношениями:

$$w_{ii} = b_i \qquad (0 \le i \le n)$$
$$w_{ij} = w_{i,j-1} + a_j + b_j \qquad (0 \le i < j \le n)$$
$$p_{ii} = w_{ii} \qquad (0 \le i \le n)$$
$$p_{ij} = w_{ij} + \mathrm{MIN}\ k: i < k \le j : (p_{i,k-1} + p_{kj}) \qquad (0 \le i < j \le n)$$

Последнее уравнение прямо следует из определений величины P и оптимальности. Так как имеется примерно $n^2/2$ значений p_{ij}, а операция минимизации в правой части требует выбора из $0 < j-i \le n$ значений, то полная минимизация потребует примерно $n^3/6$ операций. Кнут указал способ сэкономить один фактор

n (см. ниже), и только благодаря этой экономии данный алгоритм становится интересным для практических целей.

Пусть r_{ij} – то значение величины k, в котором достигается минимум p_{ij}. Оказывается, поиск r_{ij} можно ограничить гораздо меньшим интервалом, то есть уменьшить число j–i шагов вычисления. Ключевым здесь является наблюдение, что если мы нашли корень r_{ij} оптимального поддерева T_{ij}, то ни расширение дерева добавлением какого-нибудь узла справа, ни уменьшение дерева удалением крайнего левого узла не могут сдвинуть оптимальный корень влево. Это выражается соотношением

$$r_{i,j-1} \le r_{ij} \le r_{i+1,j},$$

которое ограничивает поиск решения для r_{ij} интервалом $r_{i,j-1}$ ... $r_{i+1,j}$. Это приводит к полному числу элементарных шагов порядка n^2.

Мы готовы теперь построить алгоритм оптимизации в деталях. Вспомним следующие определения для оптимальных деревьев T_{ij}, состоящих из узлов с ключами k_{i+1} ... k_j:

1. a_i: частота поиска ключа k_i.
2. b_j: частота случаев, когда аргумент поиска x оказывается между k_j и k_{j+1}.
3. w_{ij}: вес T_{ij}.
4. p_{ij}: взвешенная длина путей поддерева T_{ij}.
5. r_{ij}: индекс корня поддерева T_{ij}.

Объявим следующие массивы:

```
a:      ARRAY n+1 OF INTEGER;  (*a[0] не используется*)
b:      ARRAY n+1 OF INTEGER;
p,w,r:  ARRAY n+1, n+1 OF INTEGER;
```

Предположим, что веса w_{ij} вычислены из a и b простейшим способом. Будем считать w аргументом процедуры OptTree, которую нужно разработать, а r – ее результатом, так как r полностью описывает структуру дерева. Массив p можно считать промежуточным результатом. Начиная с наименьших возможных деревьев, то есть вообще не имеющих узлов, мы переходим ко все большим деревьям. Обозначим ширину j–i поддерева T_{ij} как h. Тогда можно легко определить значения p_{ii} для всех деревьев с h = 0 в соответствии с определением величин p_{ij}:

```
FOR i := 0 TO n DO p[i,i] := b[i] END
```

В случае h = 1 речь идет о деревьях с одним узлом, который, очевидно, и является корнем (см. рис. 4.38):

```
FOR i := 0 TO n-1 DO
    j := i+1; p[i,j] := w[i,j] + p[i,i] + p[j,j]; r[i,j] := j
END
```

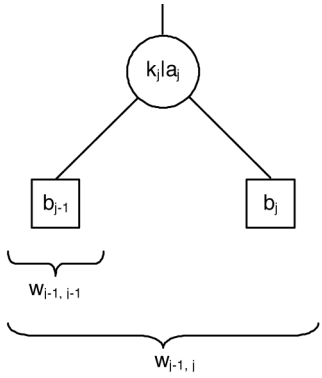


Рис. 4.38. Оптимальное дерево поиска с единственным узлом

Заметим, что i и j обозначают левый и правый пределы значений индекса в рассматриваемом дереве T_{ij}. Для случаев h > 1 используем цикл с h от 2 до n, причем случай h = n соответствует всему дереву $T_{0,n}$. В каждом случае минимальная длина путей p_{ij} и соответствующий индекс корня r_{ij} определяются простым циклом, в котором индекс k пробегает интервал для r_{ij}:

```
FOR h := 2 TO n DO
    FOR i := 0 TO n-h DO
        j := i+h;
        найти k и min = MIN k: i < k < j : (p[i,k-1] + p[k,j]), чтобы r[i,j-1] < k < r[i+1,j];
        p[i,j] := min + w[i,j]; r[i,j] := k
    END
END
```

Детали того, как уточняется предложение, набранное курсивом, можно найти в программе, приводимой ниже. Теперь средняя длина пути дерева $T_{0,n}$ дается отношением $p_{0,n}/w_{0,n}$, а его корнем является узел с индексом $r_{0,n}$.

Опишем теперь структуру проектируемой программы. Ее двумя главными компонентами являются процедура для нахождения оптимального дерева поиска при заданном распределении весов w, а также прцедура для печати дерева при заданных индексах r. Сначала вводятся частоты a и b, а также ключи. Ключи на самом деле не нужны для вычисления структуры дерева; они просто используются при распечатке дерева. После печати статистики частот программа вычисляет длину путей идеально сбалансированного дерева, заодно определяя и корни его поддеревьев. Затем печатается средняя взвешенная длина пути и распечатывается само дерево.

В третьей части вызывается процедура OptTree, чтобы вычислить оптимальное дерево поиска; затем это дерево распечатывается. Наконец, те же процедуры используются для вычисления и печати оптимального дерева с учетом частот только ключей, игнорируя частоты значений, не являющихся ключами. Все это можно суммировать так, как показано ниже, начиная с объявлений глобальных констант и переменных:

```
CONST N = 100; (*макс. число ключевых слов*)          (* ADruS46_OptTree *)
    WordLen = 16; (*макс. длина ключа*)

VAR key: ARRAY N+1, WordLen OF CHAR;
    a, b: ARRAY N+1 OF INTEGER;
    p, w, r: ARRAY N+1, N+1 OF INTEGER;

PROCEDURE BalTree (i, j: INTEGER): INTEGER;
    VAR k, res: INTEGER;
BEGIN
    k := (i+j+1) DIV 2; r[i, j] := k;
    IF i >= j THEN res := 0
    ELSE res := BalTree(i, k-1) + BalTree(k, j) + w[i, j]
    END;
    RETURN  res
END BalTree;
```

```
PROCEDURE ComputeOptTree (n: INTEGER);
    VAR x, min, tmp: INTEGER;
        i, j, k, h, m: INTEGER;
BEGIN
    (*аргумент: w, результаты: p, r*)
    FOR i := 0 TO n DO p[i, i] := 0 END;
    FOR i := 0 TO n-1 DO
        j := i+1; p[i, j] := w[i, j]; r[i, j] := j
    END;
    FOR h := 2 TO n DO
        FOR i := 0 TO n-h DO
            j := i+h; m := r[i, j-1]; min := p[i, m-1] + p[m, j];
            FOR k := m+1 TO r[i+1, j] DO
                tmp := p[i, k-1]; x := p[k, j] + tmp;
                IF x < min THEN m := k; min := x END
            END;
            p[i, j] := min + w[i, j]; r[i, j] := m
        END
    END
END ComputeOptTree;

PROCEDURE WriteTree (i, j, level: INTEGER);
    VAR k: INTEGER;
    (*для вывода используется глобальный объект печати W*)
BEGIN
    IF i < j THEN
        WriteTree(i, r[i, j]-1, level+1);
        FOR k := 1 TO level DO Texts.Write(W, TAB) END;
        Texts.WriteString(W, key[r[i, j]]); Texts.WriteLn(W);
        WriteTree(r[i, j], j, level+1)
    END
END WriteTree;

PROCEDURE Find (VAR S: Texts.Scanner);
    VAR i, j, n: INTEGER;
    (*для вывода используется глобальный объект печати W*)
BEGIN
    Texts.Scan(S); b[0] := SHORT(S.i);
    n := 0; Texts.Scan(S); (*ввод: a, ключ, b*)
    WHILE S.class = Texts.Int DO
        INC(n); a[n] := SHORT(S.i); Texts.Scan(S); COPY(S.s, key[n]);
        Texts.Scan(S); b[n] := SHORT(S.i); Texts.Scan(S)
    END;

    (*вычислить w из a и b*)
    FOR i := 0 TO n DO
        w[i, i] := b[i];
        FOR j := i+1 TO n DO
            w[i, j] := w[i, j-1] + a[j] + b[j]
        END
    END;
```

```
    Texts.WriteString(W, "Полный вес = ");
    Texts.WriteInt(W, w[0, n], 6); Texts.WriteLn(W);
    Texts.WriteString(W, "Длина путей сбалансированного дерева = ");
    Texts.WriteInt(W, BalTree(0, n), 6); Texts.WriteLn(W);
    WriteTree(0, n, 0); Texts.WriteLn(W);

    ComputeOptTree(n);

    Texts.WriteString(W, "Длина путей оптимального дерева = ");
    Texts.WriteInt(W, p[0, n], 6); Texts.WriteLn(W);
    WriteTree(0, n, 0); Texts.WriteLn(W);

    FOR i := 0 TO n DO
        w[i, i] := 0;
        FOR j := i+1 TO n DO  w[i, j] := w[i, j-1] + a[j] END
    END;

    ComputeOptTree(n);

    Texts.WriteString(W, "оптимальное дерево без учета  b"); Texts.WriteLn(W);
    WriteTree(0, n, 0); Texts.WriteLn(W)
END Find;
```

В качестве примера рассмотрим следующие входные данные для дерева с тремя ключами:

```
20   1 Albert  10  2 Ernst  1  5 Peter  1
b0 = 20
a1 = 1     key1 = Albert       b1 = 10
a2 = 2     key2 = Ernst        b2 = 1
a3 = 4     key3 = Peter        b3 = 1
```

Результаты работы процедуры Find показаны на рис. 4.39; видно, что структуры, полученные в трех случаях, могут сильно отличаться. Полный вес равен 40, длина путей сбалансированного дерева равна 78, а для оптимального дерева – 66.

Рис. 4.39. Три дерева, построенные процедурой ComputeOptTree

Из этого алгоритма очевидно, что затраты на определение оптимальной структуры имеют порядок n^2; к тому же и объем требуемой памяти имеет порядок n^2. Это неприемлемо для очень больших n. Поэтому весьма желательно найти более эффективные алгоритмы. Один из них был разработан Ху и Такером [4.5]; их алгоритм требует памяти порядка O(n) и вычислительных затрат порядка O(n*log(n)). Однако в нем рассматривается только случай, когда частоты ключей равны нулю, то есть учитываются только безуспешные попытки поиска ключей. Другой алго-

ритм, также требующий O(n) элементов памяти и O(n*log(n)) вычислительных операций, описан Уокером и Готлибом [4.7]. Вместо поиска оптимума этот алгоритм пытается найти почти оптимальное дерево. Поэтому его можно построить на использовании эвристик. Основная идея такова.

Представим себе, что узлы (реальные и дополнительные) распределены на линейной шкале и что каждому узлу приписан вес, равный соответствующей частоте (или вероятности) обращений. Затем найдем узел, ближайший к их центру тяжести. Этот узел называется *центроидом*, а его индекс равен величине

$$(\mathbf{S}i: 1 \le i \le n : i*a_i) + (\mathbf{S}i: 1 \le i \le m : i*b_i) / W,$$

округленной до ближайшего целого. Если вес всех узлов одинаков, то корень искомого оптимального дерева, очевидно, совпадает с центроидом. В противном случае – рассуждают авторы алгоритма – он будет в большинстве случаев находиться вблизи центроида. Тогда выполняется ограниченный поиск для нахождения локального оптимума, после чего та же процедура применяется к двум получающимся поддеревьям. Вероятность того, что корень лежит очень близко к центроиду, растет вместе с объемом дерева n. Как только поддеревья становятся достаточно малыми, оптимум для них может быть найден посредством точного алгоритма, приведенного выше.

4.7. Б-деревья (B-trees)

До сих пор наше обсуждение ограничивалось двоичными деревьями, то есть такими, где каждый узел имеет не более двух потомков. Этого вполне достаточно, если, например, мы хотим представить семейные связи, подчеркивая происхождение, то есть указывая для каждого человека его двух родителей. Ведь ни у кого не бывает больше двух родителей. Но что, если возникнет необходимость указывать потомков каждого человека? Тогда придется учесть, что у некоторых людей больше двух детей, и тогда деревья будут содержать узлы со многими ветвями. Будем называть такие деревья *сильно ветвящимися*.

Разумеется, в подобных структурах нет ничего особенного, и мы уже знакомы со всеми средствами программирования и определения данных, чтобы справиться с такими ситуациями. Например, если задан абсолютный верхний предел на число детей (такое предположение, конечно, немного футуристично), то можно представлять детей полем-массивом в записи, представляющей человека. Но если число детей сильно зависит от конкретного индивида, то это может привести к плохому использованию имеющейся памяти. В этом случае гораздо разумнее организовать потомство с помощью линейного списка, причем указатель на младшего (или старшего) потомка хранится в записи родителя. Возможное определение типа для этого случая приводится ниже, а возможная структура данных показана на рис. 4.40.

```
TYPE Person = POINTER TO RECORD
                  name: alfa;
                  sibling, offspring: Person
              END
```

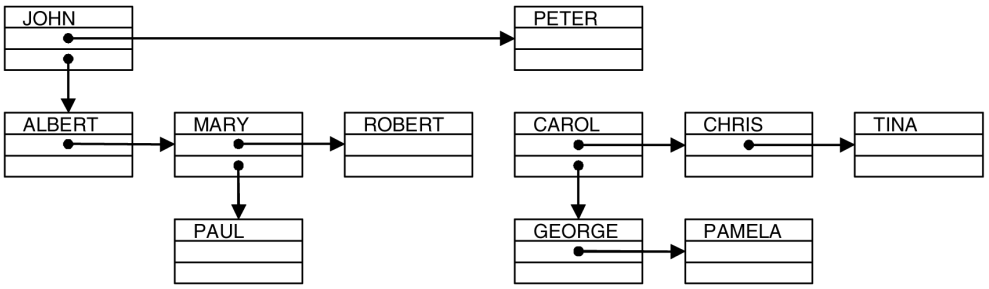


Рис. 4.40. Сильно ветвящееся дерево, представленное как двоичное дерево

При наклоне картинки на 45 градусов она будет выглядеть как обычное двоичное дерево. Но такой вид вводит в заблуждение, так как функциональный смысл двух структур совершенно разный. Обычно нельзя безнаказанно обращаться с братом как с сыном и не следует так поступать даже при определении структур данных. Структуру данных в этом примере можно усложнить еще больше, вводя в запись для каждого индивида дополнительные компоненты, представляющие другие семейные связи, например супружеские отношения между мужем и женой или даже информацию о родителях, ведь подобную информацию не всегда можно получить из ссылок от родителей к детям и к братьям-сестрам. Подобная структура быстро вырастает в сложную реляционную базу данных, на которую можно отобразить несколько деревьев. Алгоритмы, работающие с такими структурами, тесно связаны с соответствующими определениями данных, так что не имеет смысла указывать ни какие-либо общие правила, ни общеприменимые приемы.

Однако есть очень полезная область применения сильно ветвящихся деревьев, представляющая общий интерес. Это построение и поддержка больших деревьев поиска, в которых нужно выполнять вставки и удаления элементов, но оперативная память компьютера либо недостаточна по размеру, либо слишком дорога, чтобы использовать ее для длительного хранения данных.

Итак, предположим, что узлы дерева должны храниться на внешних устройствах хранения данных, таких как диски. Динамические структуры данных, введенные в этой главе, особенно удобны для того, чтобы использовать внешние носители. Главная новация будет просто в том, что роль указателей теперь будут играть дисковые адреса, а не адреса в оперативной памяти. При использовании двоичного дерева для набора данных из, скажем, миллиона элементов потребуется в среднем примерно $\ln 10^6$ (то есть около 20) шагов поиска. Поскольку здесь каждый шаг требует обращения к диску (с неизбежной задержкой), то крайне желательно организовать хранение так, чтобы уменьшить число таких обращений. Сильно ветвящееся дерево – идеальное решение проблемы. Если имеет место обращение к элементу на внешнем носителе, то без особых усилий становится доступной целая группа элементов. Отсюда идея разделить дерево на поддеревья таким образом, чтобы эти поддеревья представлялись блоками, которые доступ-

ны сразу целиком. Назовем такие поддеревья *страницами*. На рис. 4.41 показано двоичное дерево, разделенное на страницы по 7 узлов на каждой.

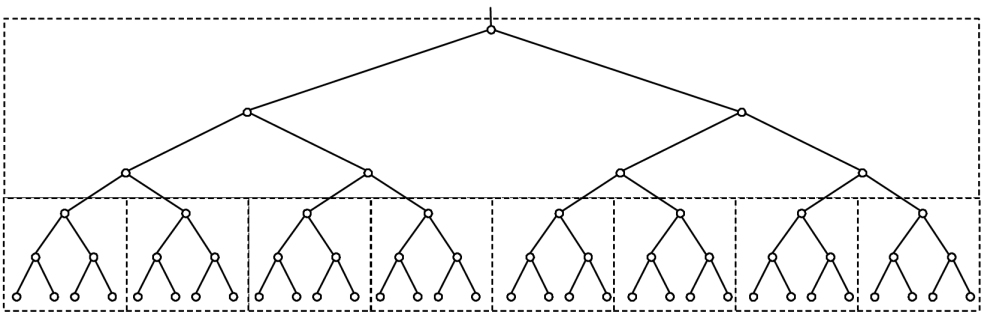


Рис. 4.41. Двоичное дерево, разделенное на страницы

Экономия на числе обращений к диску – теперь обращение к диску соответствует обращению к странице – может быть значительной. Предположим, что на каждой странице решено размещать по 100 узлов (это разумное число); тогда дерево поиска из миллиона элементов потребует в среднем только $\log_{100}(10^6)$ (то есть около 3) обращений к страницам вместо 20. Правда, при случайном росте дерева это число все равно может достигать 10^4 в наихудшем случае. Ясно, что ростом сильно ветвящихся деревьев почти обязательно нужно как-то управлять.

4.7.1. Сильно ветвящиеся Б-деревья

Если искать способ управления ростом дерева, то следует сразу исключить требование идеальной сбалансированности, так как здесь понадобятся слишком большие накладные расходы на балансировку. Ясно, что нужно несколько ослабить требования. Весьма разумный критерий был предложен Бэйером и Маккрейтом в 1970 г. [4.2]: каждая страница (кроме одной) содержит от n до $2n$ узлов, где n – некоторая заданная константа. Поэтому для дерева из N элементов и с максимальным размером страницы в $2n$ узлов потребуется в худшем случае $\log_n N$ обращений к страницам; а именно на обращения к страницам приходятся основные усилия в подобном поиске. Более того, важный коэффициент использования памяти будет не меньше 50%, так как страницы всегда будут заполнены по крайней мере наполовину. Причем при всех этих достоинствах описываемая схема требует сравнительно простых алгоритмов для поиска, вставки и удаления элементов. В дальнейшем мы подробно изучим их.

Такую структуру данных называют *Б-деревом* (также B-дерево; B-tree), число n – его *порядком*, и при этом предполагаются следующие свойства:

1. Каждая страница содержит не более $2n$ элементов (ключей).
2. Каждая страница, кроме корневой, содержит не менее n элементов.

3. Каждая страница либо является концевой, то есть не имеет потомков, либо имеет m+1 потомков, где m – число ключей на этой странице.
4. Все концевые страницы находятся на одном уровне.

Рисунок 4.42 показывает Б-дерево порядка 2, имеющее 3 уровня. На всех страницах – 2, 3 или 4 элемента; исключением является корень, где разрешено иметь только один элемент. Все концевые страницы – на уровне 3. Ключи будут стоять в порядке возрастания слева направо, если Б-дерево «сплющить» в один слой так, чтобы потомки вставали между ключами соответствующих страниц-предков. Такая организация является естественным развитием идеи двоичных деревьев поиска и предопределяет способ поиска элемента с заданным ключом. Рассмотрим страницу, показанную на рис. 4.43, и пусть задан аргумент поиска x. Предполагая, что страница уже считана в оперативную память, можно использовать обычные методы поиска среди ключей $k_0 \ldots k_{m-1}$. Если m велико, то можно применить поиск делением пополам; если оно мало, то будет достаточно обычного последовательного поиска. (Заметим, что время поиска в оперативной памяти будет, вероятно, пренебрежимо мало по сравнению со временем считывания страницы в оперативную память из внешней.) Если поиск завершился неудачей, то возникает одна из следующих ситуаций:

1. $k_i < x < k_{i+1}$ для $0 < i < m-1$. Поиск продолжается на странице p_i^
2. $k_{m-1} < x$. Поиск продолжается на странице p_{m-1}^.
3. $x < k_0$. Поиск продолжается на странице p_{-1}^.

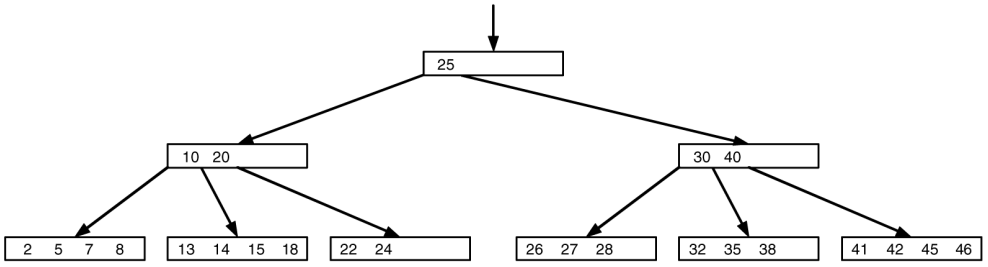


Рис. 4.42. Б-дерево порядка 2

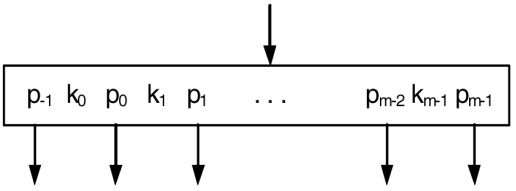


Рис. 4.43. Б-дерево с m ключами

Если в каком-то случае указатель оказался равен NIL, то есть страница-потомок отсутствует, то это значит, что элемента с ключом x во всем дереве нет, и поиск заканчивается.

Удивительно, но вставка в Б-дерево тоже сравнительно проста. Если элемент нужно вставить на странице с m < 2n элементами, то вставка затрагивает только содержимое этой страницы. И только вставка в уже заполненную страницу затронет структуру дерева и может привести к размещению новых страниц. Чтобы понять, что случится в этом случае, рассмотрим рис. 4.44, который показывает вставку ключа 22 в Б-дерево порядка 2. Здесь выполняются следующие шаги:

1. Обнаруживается, что ключа 22 в дереве нет; вставка на странице C невозможна, так как C уже полна.
2. Страница C разбивается на две (то есть размещается новая страница D).
3. 2n+1 ключей поровну распределяются между страницами C и D, а средний ключ перемещается на один уровень вверх на страницу-предок A.

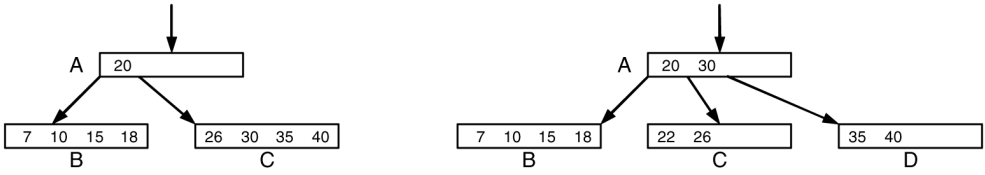


Рис. 4.44. Вставка ключа 22 в Б-дерево

Эта очень изящная схема сохраняет все характеристические свойства Б-деревьев. В частности, разбиваемые страницы содержат в точности n элементов. Разумеется, вставка элемента на странице-предке тоже может вызвать переполнение, так что снова нужно выполнять разбиение, и т. д. В крайнем случае, процесс разбиений достигнет корня. И это единственный способ увеличить высоту Б-дерева. Странная манера расти у Б-деревьев: вверх от листьев к корню.

Разработаем теперь подробную программу на основе этих набросков. Уже ясно, что самой удобной будет рекурсивная формулировка, так как процесс разбиения может распространяться в обратном направлении по пути поиска. Поэтому по общей структуре программа получится похожей на вставку в сбалансированное дерево, хотя в деталях будут отличия. Прежде всего нужно определить структуру страниц. Выберем представление элементов с помощью массива:

```
TYPE Page = POINTER TO PageDescriptor;
    Item = RECORD key: INTEGER;
               p: Page;
               count: INTEGER  (*данные*)
           END;
    PageDescriptor = RECORD m: INTEGER;   (* 0 .. 2n *)
                         p0: Page;
                         e: ARRAY 2*n OF Item
                     END
```

Поле count представляет любую информацию, которая может быть связана с элементом, но оно никак не участвует собственно в поиске. Заметим, что каждая страница предоставляет место для 2n элементов. Поле m указывает реальное число элементов на данной странице. Так как m ≥ n (кроме корневой страницы), то гарантируется, что память будет использована по меньшей мере на 50%.

Алгоритм поиска и вставки в Б-дерево сформулирован ниже в виде процедуры search. Его основная структура бесхитростна и подобна поиску в сбалансированном двоичном дереве, за исключением того факта, что выбор ветви не ограничен двумя вариантами. Вместо этого поиск внутри страницы реализован как поиск делением пополам в массиве e.

Алгоритм вставки для ясности сформулирован как отдельная процедура. Она вызывается, когда поиск показал, что нужно передать элемент вверх по дереву (в направлении корня). Это обстоятельство указывается булевским параметром-переменной h; его использование аналогично случаю алгоритма вставки в сбалансированное дерево, где h сообщал, что поддерево выросло. Если h истинно, то второй параметр-переменная u содержит элемент, передаваемый наверх. Заметим, что вставки начинаются в виртуальных страницах, а именно в так называемых дополнительных узлах (см. рис. 4.19); при этом новый элемент сразу передается через параметр u вверх на концевую страницу для реальной вставки. Вот набросок этой схемы:

```
PROCEDURE search (x: INTEGER; a: Page; VAR h: BOOLEAN; VAR u: Item);
BEGIN
    IF a = NIL THEN (*x в дереве нет, вставить*)
        присвоить x элементу u, установить h в TRUE, чтобы сообщить,
        что элемент u передается вверх по дереву
    ELSE
        поиск делением пополам значения x в массиве a.e;
        IF найден THEN
            обработать данные
        ELSE
            search(x, descendant, h, u);
            IF h THEN (*наверх был передан элемент*)
                IF число элементов на странице a^ < 2n THEN
                    вставить u на странице a^ и установить h в FALSE
                ELSE
                    разбить страницу и передать средний элемент наверх
                END
            END
        END
    END
END search
```

Если h равен TRUE после вызова процедуры search в главной программе, значит, требуется разбить корневую страницу. Так как корневая страница играет особую роль, то это действие следует запрограммировать отдельно. Оно состоит просто в размещении новой корневой страницы и вставке единственного элемента,

задаваемого параметром u. В результате новая корневая страница содержит единственный элемент. Полный текст программы приведен ниже, а рис. 4.45 показывает, как она строит Б-дерево при вставке ключей из следующей последовательности:

```
20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;
```

Точки с запятой отмечают моменты, когда сделаны «снимки», – после каждого размещения страниц. Вставка последнего ключа вызывает два разбиения и размещение трех новых страниц.

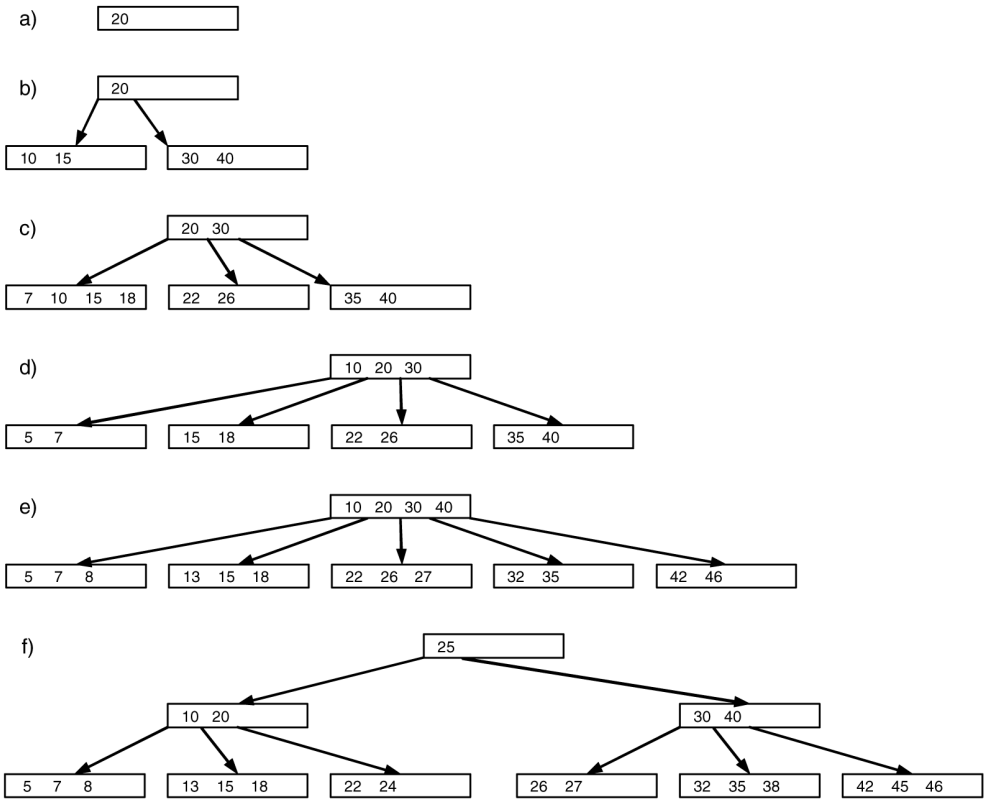


Рис. 4.45. Рост Б-дерева порядка 2

Поскольку каждый вызов процедуры поиска подразумевает одно копирование страницы в оперативную память, то понадобится не более $k = \log_n(N)$ рекурсивных вызовов для дерева из N элементов. Поэтому нужно иметь возможность разместить k страниц в оперативной памяти. Это первое ограничение на размер страницы 2n. На самом деле нужно размещать больше, чем k страниц, так как вставка может вызвать разбиение. Отсюда следует, что корневую страницу лучше посто-

янно держать в оперативной памяти, так как каждый запрос обязательно проходит через корневую страницу.

Другое достоинство организации данных с помощью Б-деревьев – удобство и экономичность чисто последовательного обновления всей базы данных. Каждая страница загружается в оперативную память в точности один раз.

Операция удаления элементов из Б-дерева по идее довольно проста, но в деталях возникают усложнения. Следует различать две разные ситуации:

1. Удаляемый элемент находится на концевой странице; здесь алгоритм удаления прост и понятен.
2. Элемент находится на странице, не являющейся концевой; его нужно заменить одним из двух соседних в лексикографическом смысле элементов, которые оказываются на концевых страницах и могут быть легко удалены.

В случае 2 нахождение соседнего ключа аналогично соответствующему алгоритму при удалении из двоичного дерева. Мы спускаемся по крайним правым указателям вниз до концевой страницы P, заменяем удаляемый элемент крайним правым элементом на P и затем уменьшаем размер страницы P на 1. В любом случае за уменьшением размера должна следовать проверка числа элементов m на уменьшившейся странице, так как $m < n$ нарушает характеристическое свойство Б-деревьев. Здесь нужно предпринять дополнительные действия; это условие *недостаточной заполненности* (underflow) указывается булевским параметром h.

Единственный выход – позаимствовать элемент с одной из соседних страниц, скажем Q. Поскольку это требует считывания страницы Q в оперативную память, – что относительно дорого, – есть соблазн извлечь максимальную пользу из этой нежелательной ситуации и позаимствовать за один раз больше одного элемента. Обычная стратегия – распределить элементы на страницах P и Q поровну на обеих страницах. Это называется *балансировкой страниц* (page balancing).

Разумеется, может случиться, что элементов для заимствования не осталось, поскольку страница Q достигла минимального размера n. В этом случае общее число элементов на страницах P и Q равно $2n-1$, и можно объединить обе страницы в одну, добавив средний элемент со страницы-предка для P и Q, а затем полностью избавиться от страницы Q. Эта операция является в точности обращением разбиения страницы. Целиком процесс проиллюстрирован удалением ключа 22 на рис. 4.44. И здесь снова удаление среднего ключа со страницы-предка может уменьшить размер последней ниже разрешенного предела n, тем самым требуя выполнения специальных действий (балансировки или слияния) на следующем уровне. В крайнем случае, слияние страниц может распространяться вверх до самого корня. Если корень уменьшается до размера 0, следует удалить сам корень, что вызывает уменьшение высоты Б-дерева. На самом деле это единственная ситуация, когда может уменьшиться высота Б-дерева. Рисунок 4.46 показывает постепенное уменьшение Б-дерева с рис. 4.45 при последовательном удалении ключей

```
25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;
```

Здесь, как и раньше, точки с запятой отмечают места, где делаются «снимки», то есть там, где удаляются страницы. Следует особо подчеркнуть сходство структуры алгоритма удаления с соответствующей процедурой для сбалансированного дерева.

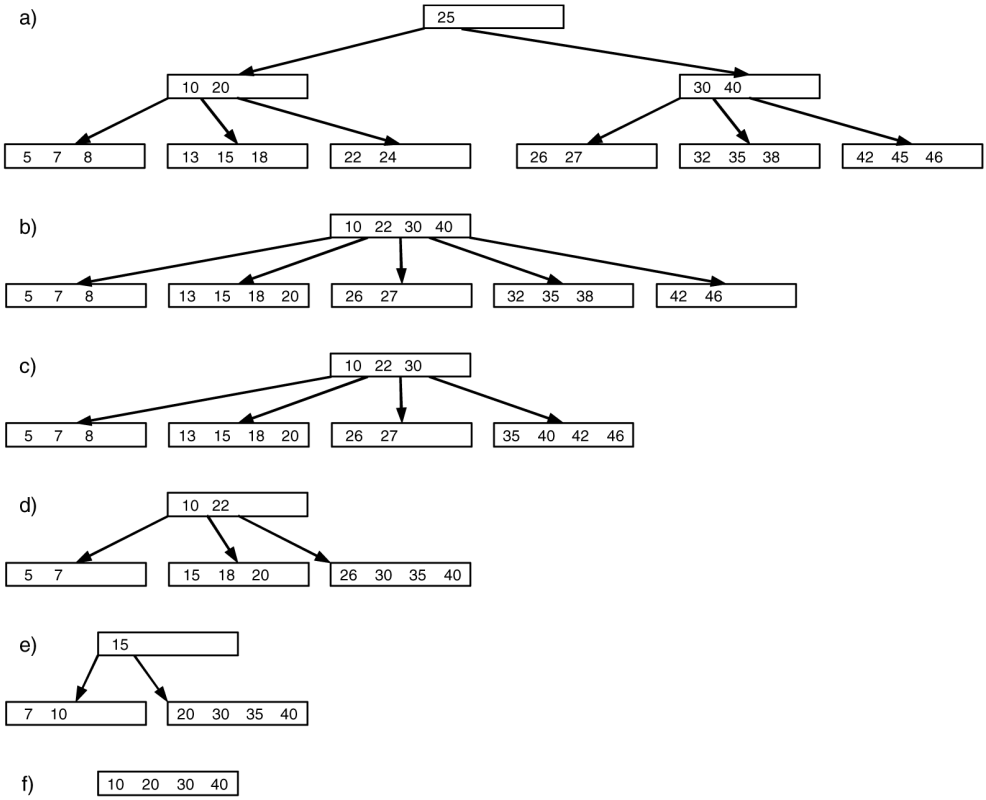


Рис. 4.46. Удаление элементов из Б-дерева порядка 2

```
TYPE Page = POINTER TO PageRec;                    (* ADruS471_Btrees *)
    Entry = RECORD
        key: INTEGER; p: Page
    END;

    PageRec = RECORD
        m: INTEGER;  (*число элементов на странице*)
        p0: Page;
        e: ARRAY 2*N OF Entry
    END;

    VAR root: Page; W: Texts.Writer;
```

```
PROCEDURE search (x: INTEGER; VAR p: Page; VAR k: INTEGER);
    VAR i, L, R: INTEGER; found: BOOLEAN; a: Page;
BEGIN
    a := root; found := FALSE;
    WHILE (a # NIL) & ~found DO
        L := 0; R := a.m; (*поиск делением пополам*)
        WHILE L < R DO
            i := (L+R) DIV 2;
            IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
        END;
        IF (R < a.m) & (a.e[R].key = x) THEN found := TRUE
        ELSIF R = 0 THEN a := a.p0
        ELSE a := a.e[R-1].p
        END
    END;
    p := a; k := R
END search;

PROCEDURE insert (x: INTEGER; a: Page; VAR h: BOOLEAN; VAR v: Entry);
    (*a # NIL. Искать ключ x в Б-дереве с корнем a;
      вставить новый элемент с ключом x.
      Если наверх нужно передать элемент, присвоить его v.
      h := "высота дерева увеличилась"*)

    VAR i, L, R: INTEGER;
        b: Page; u: Entry;
BEGIN
    (* ~h *)
    IF a = NIL THEN
        v.key := x; v.p := NIL; h := TRUE
    ELSE
        L := 0; R := a.m; (*поиск делением пополам*)
        WHILE L < R DO
            i := (L+R) DIV 2;
            IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
        END;
        IF (R < a.m) & (a.e[R].key = x) THEN (*нашли, делать нечего*)
        ELSE (*на этой странице нет*)
            IF R = 0 THEN b := a.p0 ELSE b := a.e[R-1].p END;
            insert(x, b, h, u);
            IF h THEN (*вставить u слева от a.e[R]*)
                IF a.m < 2*N THEN
                    h := FALSE;
                    FOR i := a.m TO R+1 BY -1 DO a.e[i] := a.e[i-1] END;
                    a.e[R] := u; INC(a.m)
                ELSE (*переполнение; разбить a на a, b
                            и присвоить средний элемент переменной v*)
                    NEW(b);
                    IF R < N THEN (*вставить в левую страницу a*)
```

```
                    v := a.e[N-1];
                    FOR i := N-1 TO R+1 BY -1 DO a.e[i] := a.e[i-1] END;
                    a.e[R] := u;
                    FOR i := 0 TO N-1 DO b.e[i] := a.e[i+N] END
                ELSE (*вставить в правую страницу b*)
                    DEC(R, N);
                    IF R = 0 THEN
                        v := u
                    ELSE
                        v := a.e[N];
                        FOR i := 0 TO R-2 DO b.e[i] := a.e[i+N+1] END;
                        b.e[R-1] := u
                    END;
                    FOR i := R TO N-1 DO b.e[i] := a.e[i+N] END
                END;
                a.m := N; b.m := N; b.p0 := v.p; v.p := b
            END
        END
    END
END insert;

PROCEDURE underflow (c, a: Page; s: INTEGER; VAR h: BOOLEAN);
    (*a = страница, где слишком мало элементов (underflow),
      c = страница-предок,
      s = индекс стертого значения в c*)
    VAR b: Page; i, k: INTEGER;
BEGIN
    (*h & (a.m = N-1) & (c.e[s-1].p = a) *)
    IF s < c.m THEN (*b := страница справа от a*)
        b := c.e[s].p;
        k := (b.m-N+1) DIV 2; (*k = число свободных элементов на странице b*)
        a.e[N-1] := c.e[s]; a.e[N-1].p := b.p0;
        IF k > 0 THEN (*сбалансировать перемещением k-1 элементов с b на a*)
            FOR i := 0 TO k-2 DO a.e[i+N] := b.e[i] END;
            c.e[s] := b.e[k-1]; b.p0 := c.e[s].p;
            c.e[s].p := b; DEC(b.m, k);
            FOR i := 0 TO b.m-1 DO b.e[i] := b.e[i+k] END;
            a.m := N-1+k; h := FALSE
        ELSE (*слить страницы a и b, b больше не нужна*)
            FOR i := 0 TO N-1 DO a.e[i+N] := b.e[i] END;
            DEC(c.m);
            FOR i := s TO c.m-1 DO c.e[i] := c.e[i+1] END;
            a.m := 2*N; h := c.m < N
        END
    ELSE (*b := страница слева от a*)
        DEC(s);
        IF s = 0 THEN b := c.p0 ELSE b := c.e[s-1].p END;
        k := (b.m-N+1) DIV 2; (*k = число свободных элементов на странице b*)
```

```
        IF k > 0 THEN
            FOR i := N-2 TO 0 BY -1 DO a.e[i+k] := a.e[i] END;
            a.e[k-1] := c.e[s]; a.e[k-1].p := a.p0;
            (*переместить k-1 элементов с b на a, один на c*)  DEC(b.m, k);
            FOR i := k-2 TO 0 BY -1 DO a.e[i] := b.e[i+b.m+1] END;
            c.e[s] := b.e[b.m]; a.p0 := c.e[s].p;
            c.e[s].p := a; a.m := N-1+k; h := FALSE
        ELSE (*слить страницы a и b, a больше не нужна*)
            c.e[s].p := a.p0; b.e[N] := c.e[s];
            FOR i := 0 TO N-2 DO b.e[i+N+1] := a.e[i] END;
            b.m := 2*N; DEC(c.m); h := c.m < N
        END
    END
END underflow;

PROCEDURE delete (x: INTEGER; a: Page; VAR h: BOOLEAN);
    (*найти и удалить ключ x в Б-дереве a;
      если на странице стало слишком мало элементов,
      сбалансировать с соседней страницей или слить;
      h := "на странице слишком мало элементов"*)

    VAR i, L, R: INTEGER; q: Page;

    PROCEDURE del (p: Page; VAR h: BOOLEAN);
        VAR k: INTEGER; q: Page;  (*глобальные a, R*)
    BEGIN k := p.m-1; q := p.e[k].p;
        IF q # NIL THEN del(q, h);
            IF h THEN underflow(p, q, p.m, h) END
        ELSE p.e[k].p := a.e[R].p; a.e[R] := p.e[k];
            DEC(p.m); h := p.m < N
        END
    END del;

BEGIN
    IF a # NIL THEN
        L := 0; R := a.m; (*поиск делением пополам*)
        WHILE L < R DO
            i := (L+R) DIV 2;
            IF x <= a.e[i].key THEN R := i ELSE L := i+1 END
        END;
        IF R = 0 THEN q := a.p0 ELSE q := a.e[R-1].p END;
        IF (R < a.m) & (a.e[R].key = x) THEN (*нашли*)
            IF q = NIL THEN (*a — концевая страница*)
                DEC(a.m); h := a.m < N;
                FOR i := R TO a.m-1 DO a.e[i] := a.e[i+1] END
            ELSE
                del(q, h);
                IF h THEN underflow(a, q, R, h) END
            END
        ELSE
```

```
            delete(x, q, h);
            IF h THEN underflow(a, q, R, h) END
        END
    END
END delete;

PROCEDURE ShowTree (VAR W: Texts.Writer; p: Page; level: INTEGER);
    VAR i: INTEGER;
BEGIN
    IF p # NIL THEN
        FOR i := 1 TO level DO Texts.Write(W, 9X) END;
        FOR i := 0 TO p.m-1 DO Texts.WriteInt(W, p.e[i].key, 4) END;
        Texts.WriteLn(W);
        IF p.m > 0 THEN ShowTree(W, p.p0, level+1) END;
        FOR i := 0 TO p.m-1 DO ShowTree(W, p.e[i].p, level+1) END
    END
END ShowTree;
```

Эффективность Б-деревьев изучалась весьма подробно, результаты можно найти в упомянутой статье Бэйера и Маккрейта. В частности, там обсуждается вопрос оптимального размера страницы, который сильно зависит от характеристик используемой вычислительной системы и памяти.

Вариации на тему Б-деревьев обсуждаются у Кнута ([2.7], с. 521–525 перевода). Заслуживает внимания наблюдение, что разбиение страницы следует откладывать, как следует откладывать и слияние страниц, и сначала пытаться сбалансировать соседние страницы. В остальном похоже, что выгода от предлагавшихся улучшений незначительна. Весьма полный обзор Б-деревьев можно найти в [4.8].

4.7.2. Двоичные Б-деревья

На первый взгляд, Б-деревья первого порядка (n = 1) – наименее интересная их разновидность. Но иногда стоит присмотреться к особому случаю. Ясно, однако, что Б-деревья первого порядка не могут быть полезны для представления больших упорядоченных индексированных наборов данных с использованием внешних устройств хранения. Поэтому мы теперь забудем про внешнюю память и снова рассмотрим деревья поиска, предполагая работу в оперативной памяти.

Двоичное Б-дерево (ДБ-дерево; BB-tree) состоит из узлов (страниц) с одним или двумя элементами. Поэтому страница содержит два или три указателя на потомков; этим объясняется термин *2–3 дерево*. В соответствии с определением Б-деревьев все концевые страницы находятся на одном уровне, а все неконцевые страницы ДБ-деревьев имеют двух или трех потомков (включая корень). Так как мы теперь имеем дело только с оперативной памятью, то нужно подумать об оптимальном использовании памяти, так что представление элементов в узле с помощью массива не кажется подходящим. Альтернативой будет динамическое, связное размещение; то есть в каждом узле есть связный список элементов длины 1 или 2. Так как у каждого узла не больше трех потомков и поэтому в нем нужно хранить не более трех указателей, соблазнительно объединить указатели на по-

томков с указателями в списке элементов, как показано на рис. 4.47. Тогда узел Б-дерева теряет свою специфику и начинает играть роль узла обычного двоичного дерева. Однако все-таки нужно отличать указатели на потомков (вертикальные стрелки) от указателей на «братьев» на той же странице (горизонтальные стрелки). Поскольку горизонтальными могут быть только указатели, направленные вправо, достаточно одного бита, чтобы отмечать эту разницу. Поэтому введем булевское поле h со значением «горизонтальный». Определение узла дерева для такого представления приводится ниже. Его предложил и исследовал в 1971 г. Бэйер (R. Bayer [4.3]); такая организация дерева поиска гарантирует максимальную длину пути p = 2*log(N).

```
TYPE Node = POINTER TO RECORD
              key: INTEGER;
              ..........
              left, right: Node;
              h: BOOLEAN  (*правый указатель горизонтальный*)
            END
```

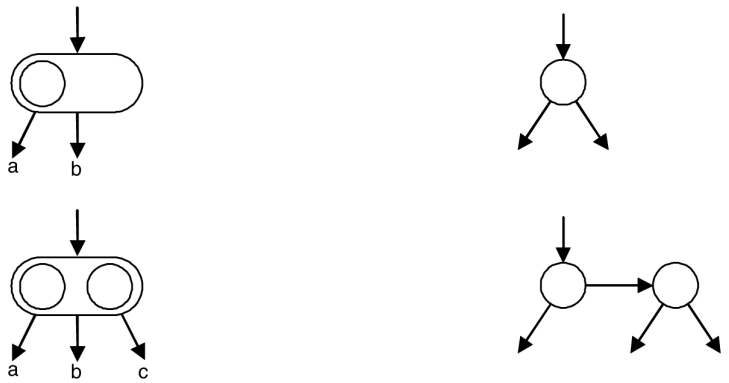


Рис. 4.47. Представление узлов ДБ-деревьев

Рассматривая вставку ключа, следует различать четыре возможных случая, возникающие при росте левых или правых поддеревьев. Эти четыре случая показаны на рис. 4.48. Напомним, что для Б-деревьев характерен рост снизу вверх к корню и что для них нужно обеспечивать, чтобы все концевые страницы были на одном уровне. Простейший случай (1) имеет место, когда растет правое поддерево узла A, причем A является единственным ключом на своей (виртуальной) странице. Тогда потомок B просто становится «братом» для A, то есть вертикальный указатель становится горизонтальным. Такое простое «поднимание руки» невозможно, если у A уже есть «брат». Тогда получается страница с 3 узлами, и ее нужно разбивать (случай 2). Ее средний узел B нужно передать на следующий уровень вверх.

Теперь предположим, что увеличилась высота левого поддерева узла B. Если снова узел B на странице один (случай 3), то есть его правый указатель ссылается

на потомка, то левому поддереву (A) разрешается стать «братом» для B. (Так как левый указатель не может быть горизонтальным, то здесь нужна простая ротация указателей.) Но если у B уже есть «брат», то поднятие A создает страницу с тремя указателями, требующую разбиения. Это разбиение выполняется самым простым способом: C становится потомком B, который, в свою очередь, поднимается на следующий уровень вверх (случай 4).

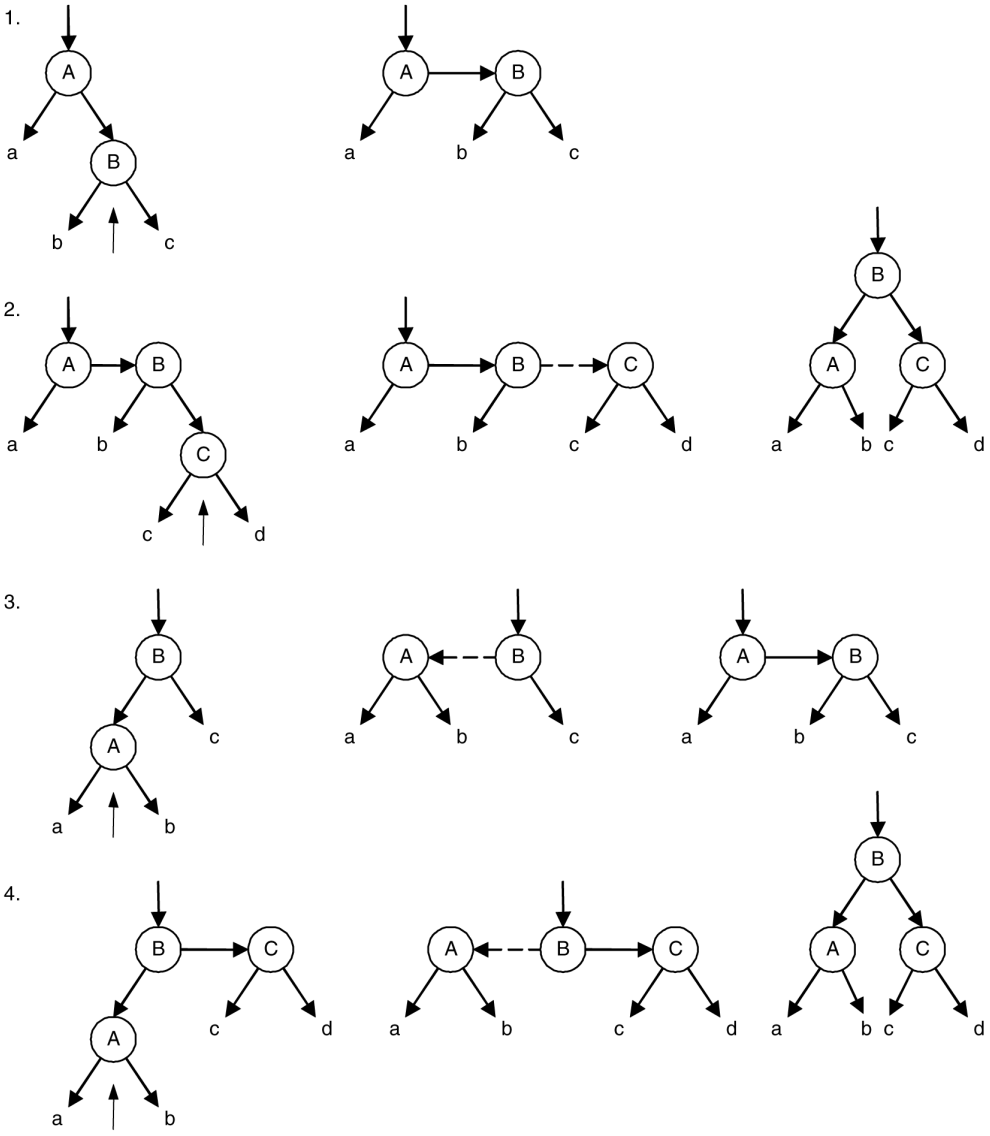


Рис. 4.48. Вставка узлов в ДБ-дереве

Нужно заметить, что при поиске ключей в сущности безразлично, идем ли мы по горизонтальному или по вертикальному указателю. Поэтому кажется неестественным, что нужно беспокоиться о том, что левый указатель в случае 3 становится горизонтальным, хотя его страница по-прежнему содержит не более двух членов. В самом деле, алгоритм вставки являет странную асимметрию для случаев роста левого или правого поддеревьев, так что ДБ-деревья кажутся довольно искусственной конструкцией. Не существует доказательства того, что это неправильно; но здоровая интуиция подсказывает, что здесь что-то неладно и что эту асимметрию следует устранить. Это приводит к понятию *симметричного двоичного Б-дерева* (СДБ-дерево; SBB-tree), которое тоже было исследовано Бэйером в 1972 г. [4.4]. В среднем оно приводит к чуть более эффективному поиску, но при этом слегка усложняются алгоритмы вставки и удаления. Кроме того, в каждом узле теперь нужны два бита (булевские переменные lh и rh), чтобы указать природу двух его указателей.

Мы ограничимся детальным рассмотрением задачи о вставках, и здесь нам снова нужно различать четыре случая роста поддеревьев. Они показаны на рис. 4.49, где очевидна симметрия, к которой мы стремились. Заметим, что каждый раз, когда растет поддерево узла A, не имеющего «брата», корень этого поддерева становится «братом» узла A. Этот случай можно в дальнейшем не рассматривать.

Во всех четырех случаях, рассмотренных на рис. 4.49, возникает переполнение страницы с последующим ее разбиением. Случаи обозначены в соответствии с направлениями горизонтальных указателей, связывающих трех «братьев» в средних рисунках. Исходное положение показано в левом столбце; средний столбец иллюстрирует тот факт, что был поднят узел с более низкого уровня, когда выросло его поддерево; рисунки в правом столбце показывают окончательный результат перегруппировки узлов.

Больше нет смысла сохранять понятие страниц, из которого эволюционировал этот способ организации дерева, поскольку мы только хотели бы ограничить максимальную длину пути величиной 2*log(N). Для этого достаточно обеспечить, чтобы два горизонтальных указателя никогда не могли встречаться подряд ни на каком пути поиска. Однако нет резона запрещать узлы с горизонтальными указателями, направленными влево и вправо, то есть по-разному трактовать левое и правое. Поэтому определим симметричное двоичное Б-дерево следующими свойствами:

1. Каждый узел содержит один ключ и не более двух поддеревьев (точнее, указателей на них).
2. Каждый указатель является либо горизонтальным, либо вертикальным. Ни на каком пути поиска не могут идти подряд два горизонтальных указателя.
3. Все концевые узлы (узлы без потомков) находятся на одном уровне.

Из этого определения следует, что самый длинный путь поиска не может более чем вдвое превышать по длине высоту дерева. Поскольку никакое СДБ-дерево с N узлами не может иметь высоту больше, чем log(N), то немедленно получаем, что 2*log(N) – верхний предел на длину пути поиска. Наглядное представление о том, как растут такие деревья, дает рис. 4.50. Ниже приведены последователь-

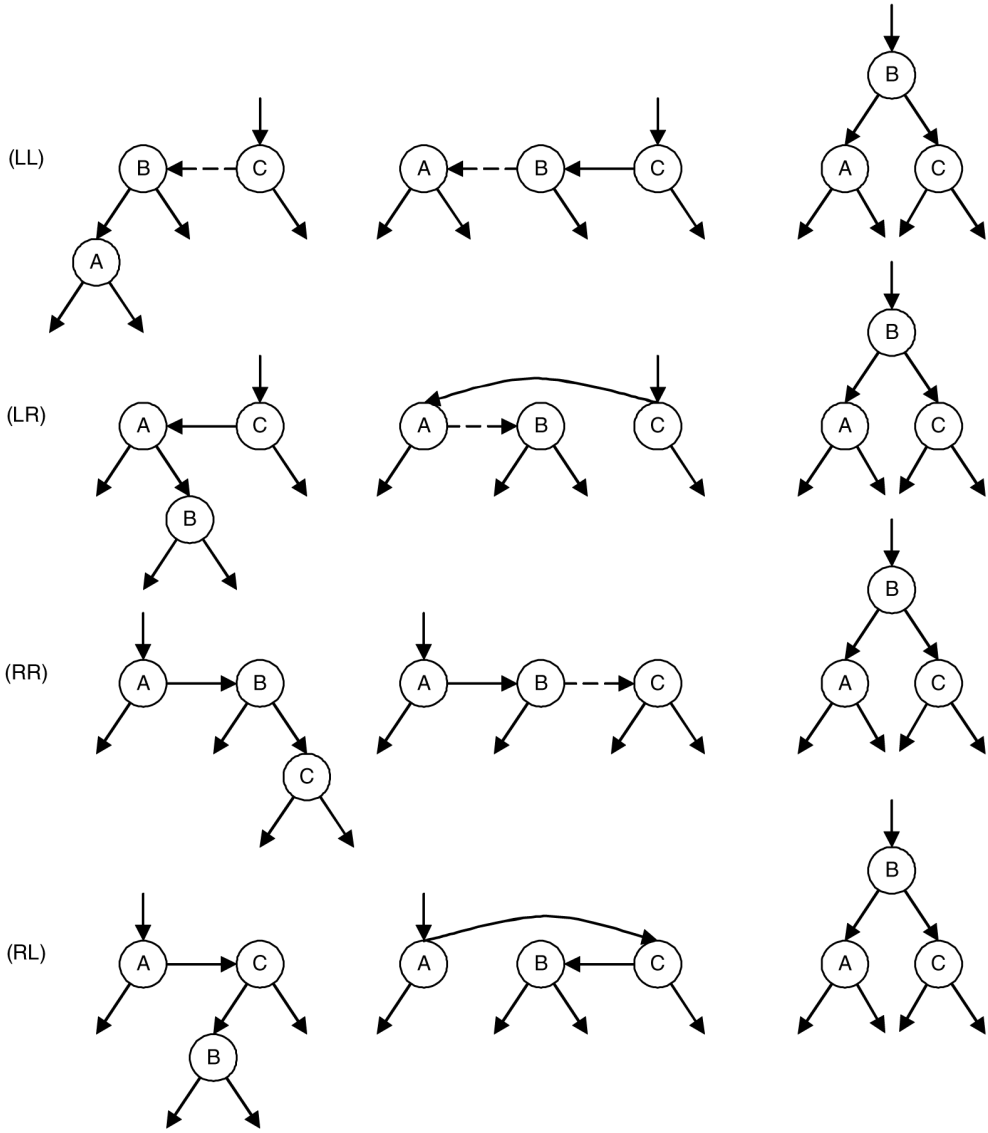


Рис. 4.49. Вставка в СДБ-деревьях

ности вставляемых ключей, причем каждой точке с запятой соответствует картинка, показывающая состояние дерева в данный момент.

(1) 1 2; 3; 4 5 6; 7;
(2) 5 4; 3; 1 2 7 6;
(3) 6 2; 4; 1 7 3 5;
(4) 4 2 6; 1 7; 3 5;

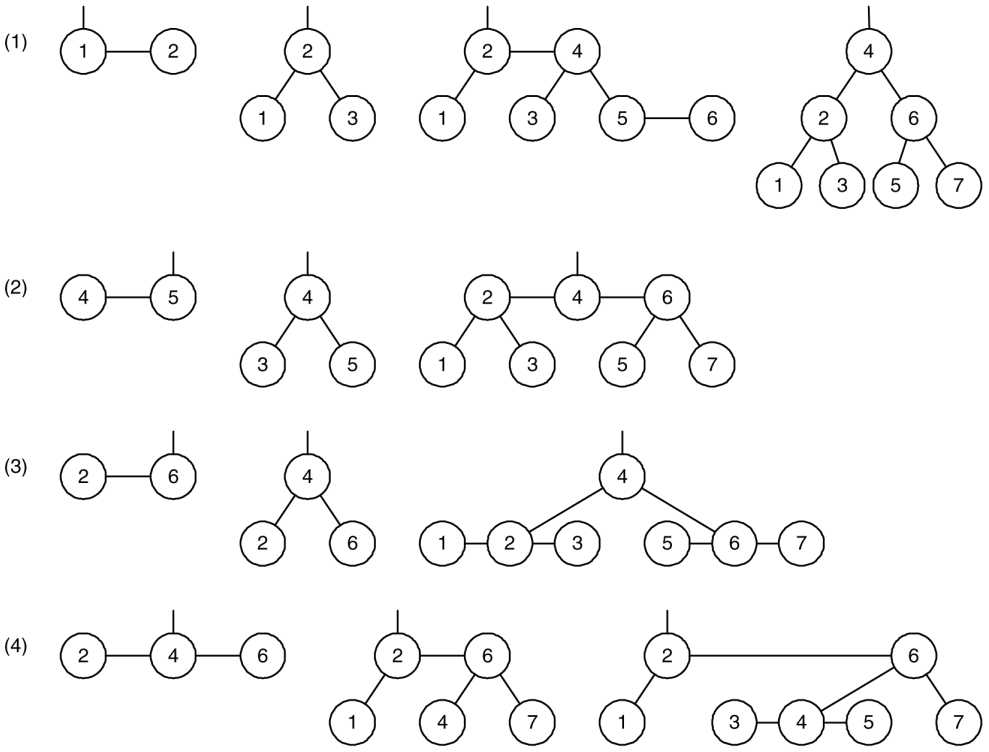


Рис. 4.50. Вставки ключей с 1 по 7

Эти картинки делают особенно очевидным третье свойство Б-деревьев: все концевые узлы находятся на одном уровне. Напрашивается сравнение со свежеподстриженными садовыми оградами из кустарника.

Алгоритм построения СДБ-деревьев показан ниже. Он основан на определении типа Node с двумя компонентами lh и rh, указывающими соответственно, является ли горизонтальным левый и правый указатель.

```
TYPE Node = RECORD
                key, count: INTEGER;
                left, right: POINTER TO Node;
                lh, rh: BOOLEAN
            END
```

Схема рекурсивной процедуры search снова повторяет основной алгоритм вставки в двоичное дерево. В нее введен третий параметр, указывающий, изменилось ли поддерево с корнем p, и соответствующий параметру h программы поиска для Б-дерева. Однако нужно отметить одно следствие представления страниц связными списками: страница обходится за один или два вызова процедуры поиска. Поэтому следует различать случай выросшего поддерева (на который указы-

вает вертикальная ссылка) от «братского» узла (на который указывает горизонтальная ссылка), который получил нового «брата», так что требуется разбиение страницы. Проблема легко решается введением трехзначного параметра h со следующими значениями:

1. h = 0: поддерево p не требует изменений структуры дерева.
2. h = 1: узел p получил «брата».
3. h = 2: увеличилась высота поддерева p.

```
PROCEDURE search (VAR p: Node; x: INTEGER; VAR h: INTEGER);
    VAR q, r: Node;                                          (* ADruS472_BBtrees *)
BEGIN
    (*h=0*)
    IF p = NIL THEN (*вставить новый узел*)
        NEW(p); p.key := x; p.L := NIL; p.R := NIL; p.lh := FALSE; p.rh := FALSE;
        h := 2
    ELSIF x < p.key THEN
        search(p.L, x, h);
        IF h > 0 THEN  (*левая ветвь выросла или получила "брата"*)
            q := p.L;
            IF p.lh THEN
                h := 2; p.lh := FALSE;
                IF q.lh THEN (*LL*)
                    p.L := q.R; q.lh := FALSE; q.R := p; p := q
                ELSE (*q.rh, LR*)
                    r := q.R; q.R := r.L; q.rh := FALSE; r.L := p.L; p.L := r.R; r.R := p; p := r
                END
            ELSE
                DEC(h);
                IF h = 1 THEN p.lh := TRUE END
            END
        END
    ELSIF x > p.key THEN
        search(p.R, x, h);
        IF h > 0 THEN  (*левая ветвь выросла или получила "брата"*)
            q := p.R;
            IF p.rh THEN
                h := 2; p.rh := FALSE;
                IF q.rh THEN (*RR*)
                    p.R := q.L; q.rh := FALSE; q.L := p; p := q
                ELSE (*q.lh, RL*)
                    r := q.L; q.L := r.R; q.lh := FALSE; r.R := p.R; p.R := r.L; r.L := p; p := r
                END
            ELSE
                DEC(h);
                IF h = 1 THEN p.rh := TRUE END
            END
        END
    END
END search;
```

Заметим, что действия, выполняемые здесь для реорганизации узлов, очень похожи на то, что делается в алгоритме поиска для сбалансированных АВЛ-деревьев. Очевидно, что все четыре случая можно реализовать простыми ротациями указателей: простые ротации в случаях LL и RR, двойные ротации в случаях LR и RL. На самом деле процедура search оказывается здесь чуть более простой, чем для АВЛ-деревьев. Ясно, что схему СДБ-деревьев можно рассматривать как альтернативу идее АВЛ-балансировки. Поэтому разумно и полезно сравнить производительность в двух случаях.

Мы воздержимся от сложного математического анализа и сосредоточимся на некоторых основных отличиях. Можно доказать, что АВЛ-сбалансированные деревья составляют подмножество СДБ-деревьев. Так что множество последних шире. Следовательно, длина путей у них в среднем больше, чем у АВЛ-деревьев. В этой связи отметим реализующее наихудший случай дерево (4) на рис. 4.50. С другой стороны, реорганизовывать ключи здесь нужно реже. Поэтому сбалансированные деревья следует предпочесть в тех задачах, где поиск ключей выполняется гораздо чаще, чем вставки (или удаления); если эти частоты сравнимы, то предпочтительными могут стать СДБ-деревья. Очень трудно сказать, где проходит граница. Здесь все сильно зависит не только от отношения частот поиска и реорганизации, но еще и от особенностей реализации. Это прежде всего касается тех случаев, когда для записи в узле используют плотно упакованное представление, так что доступ к полям требует извлечения частей слова.

СДБ-деревья позднее возродились под названием *красно-черных деревьев* (red-black tree). Разница в том, что у симметричного двоичного Б-дерева каждый узел содержит два h-поля, которые сообщают, являются ли хранящиеся в узле указатели горизонтальными, а у красно-черного дерева каждый узел содержит единственное h-поле, сообщающее, является ли горизонтальным указатель, ссылающийся на этот узел. Название отражает идею считать узел черным или красным в зависимости от того, является ли указатель, ссылающийся на него, вертикальным или горизонтальным. Два красных узла никогда не могут идти подряд ни на каком пути. Поэтому, как и в случаях ДБ- и СДБ-деревьев, длина любого пути поиска превосходит высоту дерева не более чем вдвое. Существует каноническое отображение двоичных Б-деревьев на красно-черные.

4.8. Приоритетные деревья поиска

Деревья, особенно двоичные, представляют собой очень эффективные способы организации данных, допускающих линейное упорядочение. В предыдущих главах были показаны чаще всего используемые изобретательные схемы для эффективного поиска и обслуживания таких деревьев (вставки, удаления). Однако не похоже, чтобы деревья были полезны в задачах, где данные располагаются не в одномерном, а в многомерном пространстве. На самом деле эффективный поиск в многомерных пространствах все еще остается одной из наиболее трудных проблем информатики, причем для многих приложений особенно важен случай двух измерений.

Присмотревшись к проблеме, можно обнаружить, что деревья все еще могут быть полезны, по крайней мере в двумерном случае. В конце концов, мы изображаем деревья на бумаге в двумерном пространстве. Поэтому вспомним вкратце главные свойства двух основных видов деревьев, рассмотренных до сих пор.

1. Дерево поиска подчиняется следующим инвариантам:

   p.left ≠ NIL подразумевает p.left.x < p.x,
   p.right ≠ NIL подразумевает p.x < p.right.x,

   которые выполняются для всех узлов p с ключом x. Очевидно, что эти инварианты ограничивают только горизонтальное положение узлов и что их вертикальные позиции могут быть выбраны произвольно, чтобы минимизировать время доступа при поиске (то есть длины путей).
2. Куча (heap), или *приоритетное дерево* (priority tree), подчиняется следующим инвариантам:

   p.left ≠ NIL подразумевает p.y ≤ p.left.y,
   p.right ≠ NIL подразумевает p.y ≤ p.right.y,

   и оба свойства выполняются для всех узлов p с ключом y. Очевидно, эти инварианты ограничивают только положение узлов по вертикали.

Можно объединить две эти пары условий в одном определении способа древесной организации в двумерном пространстве, подразумевая, что каждый узел имеет два ключа x и y, которые можно считать координатами узла. Такое дерево представляет собой набор точек в плоскости, то есть в двумерном декартовом пространстве; поэтому его называют *декартовым деревом* [4.9]. Мы предпочитаем название *приоритетное дерево поиска*, так как оно показывает, что эта структура возникла из комбинации приоритетного дерева и дерева поиска. Определяющими здесь являются следующие инварианты, выполняющиеся для каждого узла p:

p.left ≠ NIL подразумевает (p.left.x < p.x) & (p.y ≤ p.left.y)
p.right ≠ NIL подразумевает (p.x < p.right.x) & (p.y ≤ p.right.y)

Однако не должно сильно удивлять, что эффективность поиска в таких деревьях не особо впечатляющая. Ведь здесь гораздо меньше свободы для выбора расположения узлов, которое могло бы обеспечить короткие пути поиска. Поэтому не удается гарантировать логарифмические ограничения на вычислительные затраты при поиске, вставке или удалении элементов. Хотя в отличие от обычных несбалансированных деревьев поиска, где ситуация была такой же, здесь шансы на хорошее поведение в среднем невелики. Хуже того, операции вставки и удаления могут быть весьма громоздкими. Например, рассмотрим дерево на рис. 4.51a. Если координаты нового узла C таковы, что он должен находиться над и между узлами A и B, то потребуются серьезные усилия для преобразования (a) в (b).

МакКрейт нашел схему, похожую на балансировку, которая гарантирует логарифмические временные ограничения для операций вставки и удаления за счет их усложнения. Он назвал свою схему приоритетным деревом поиска [4.10]; одна-

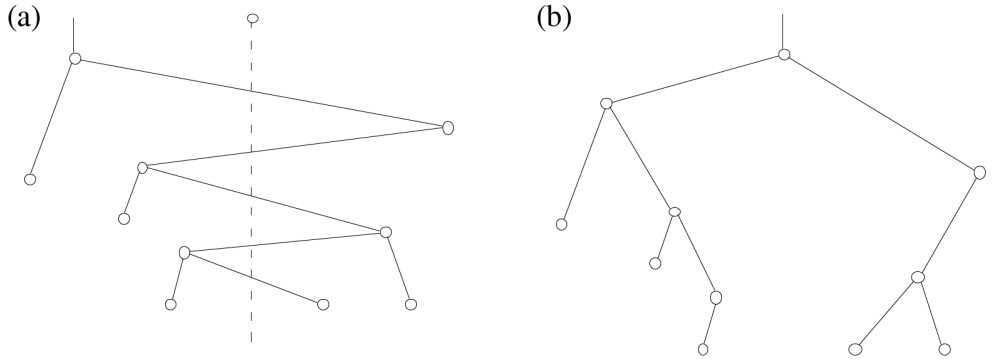


Рис. 4.51. Вставка в приоритетное дерево поиска

ко в нашей классификации ее нужно называть сбалансированным приоритетным деревом поиска. Мы воздержимся от обсуждения этой схемы, так как она очень сложна и вряд ли используется на практике. Решая несколько более ограниченную, но не менее полезную для приложений задачу, МакКрейт нашел еще одну древесную структуру, которую мы рассмотрим полностью. Вместо бесконечного пространства поиска он рассмотрел пространство данных, ограниченное с двух сторон. Обозначим граничные значения для координаты x как x_{min} и x_{max}.

В вышеописанной схеме (несбалансированных) приоритетных деревьев поиска каждый узел p делит плоскость на две части линией x = p.x. Все узлы левого подддерева лежат слева от p, а все узлы правого – справа. Для эффективного поиска такой выбор, вероятно, плох. К счастью, можно выбрать разделительную линию по-другому. Свяжем с каждым узлом p интервал [p.L .. p.R), состоящий из всех значений x от (и включая) p.L и до (но исключая) p.R. В этом интервале разрешено находиться значению координаты x узла p. Затем потребуем, чтобы левый потомок (если он есть) лежал в левой половине, а правый – в правой половине этого интервала. Поэтому разделительная линия – это не p.x, а (p.L+p.R)/2. Для каждого потомка интервал делится пополам, так что высота дерева ограничивается величиной $\log(x_{max}-x_{min})$. Этот результат выполняется, только если нет ни одной пары узлов с одинаковым значением x; впрочем, это условие гарантировано инвариантом. Если координаты – целые числа, то этот предел не может превышать длину компьютерного слова. В сущности, поиск протекает как в алгоритме деления пополам или в поиске по остаткам (radix search), и поэтому такие деревья называют приоритетными деревьями поиска по остаткам (radix priority search trees, [4.10]). Для них имеют место логарифмические ограничения на число операций, необходимых для поиска, вставки и удаления элемента, и для них также справедливы следующие инварианты, выполняющиеся для каждого узла p:

p.left ≠ NIL влечет (p.L ≤ p.left.x < p.M) & (p.y ≤ p.left.y)
p.right ≠ NIL влечет (p.M ≤ p.right.x < p.R) & (p.y ≤ p.right.y)

где

```
p.M       = (p.L + p.R) DIV 2
p.left.L  = p.L
p.left.R  = p.M
p.right.L = p.M
p.right.R = p.R
```

для всех узлов p, причем root.L = x_{min}, root.R = x_{max}.

Решающее достоинство такой схемы – в том, что операции обслуживания при вставке и удалении (сохраняющие инварианты) действуют в пределах одного «отростка» дерева между двумя разделительными линиями, так как положение последних по координате x фиксировано независимо от значений x у вставляемых узлов.

Типичные операции с приоритетными деревьями поиска – вставка, удаление, поиск элемента с наименьшим (наибольшим) значением x (или y), большим (меньшим) заданного предела, а также перечисление точек, лежащих в заданном прямоугольнике. Ниже приведены процедуры для вставки и перечисления. Они основаны на следующих объявлениях типов:

```
TYPE Node = POINTER TO RECORD
    x, y: INTEGER;
    left, right: Node
END
```

Заметим, что необязательно хранить атрибуты x_L и x_R в самих узлах. Их можно вычислять каждый раз в процессе поиска. Однако для этого в рекурсивной процедуре insert нужны два дополнительных параметра. В первом вызове (с p = root) их значения равны x_{min} и x_{max} соответственно. В остальном поиск протекает как в обычном дереве поиска. Если встречается пустой узел, то элемент вставляется. Если у вставляемого узла значение y меньше, чем у проверяемого в данный момент, новый узел меняется местами с проверяемым. Наконец, узел вставляется в левое поддерево, если его значение x меньше, чем среднее значение интервала, и в правое поддерево в противном случае.

```
PROCEDURE insert (VAR p: Node; X, Y, xL, xR: INTEGER);
    VAR xm, t: INTEGER;                    (* ADruS48_PrioritySearchTrees *)
BEGIN
    IF p = NIL THEN (*нет в дереве, вставить*)
        NEW(p); p.x := X; p.y := Y; p.left := NIL; p.right := NIL
    ELSIF p.x = X THEN (*найден; не вставлять*)
    ELSE
        IF p.y > Y THEN
            t := p.x; p.x := X; X := t;
            t := p.y; p.y := Y; Y := t
        END;
        xm := (xL + xR) DIV 2;
        IF X < xm THEN insert(p.left, X, Y, xL, xm)
```

```
        ELSE insert(p.right, X, Y, xm, xR)
        END
    END
END insert
```

Задача перечисления всех точек x, y, лежащих в заданном прямоугольнике, то есть удовлетворяющих условиям $x0 \le x < x1$ и $y \le y1$, решается с помощью приводимой ниже процедуры enumerate. Она вызывает процедуру report(x,y) для каждой найденной точки. Заметим, что одна сторона прямоугольника лежит на оси, то есть нижняя граница для y равна 0. Этим гарантируется, что перечисление требует не более O(log(N) + s) операций, где N – мощность пространства поиска по x, а s – число перечисляемых узлов.

```
PROCEDURE enumerate (p: Ptr; x0, x1, y1, xL, xR: INTEGER);
    VAR xm: INTEGER;                          (* ADruS48_PrioritySearchTrees *)
BEGIN
    IF p # NIL THEN
        IF (p.y <= y1) & (x0 <= p.x) & (p.x < x1) THEN
            report(p.x, p.y)
        END;
        xm := (xL + xR) DIV 2;
        IF x0 < xm THEN enumerate(p.left, x0, x1, y1, xL, xm) END;
        IF xm < x1 THEN enumerate(p.right, x0, x1, y1, xm, xR) END
    END
END enumerate
```

Упражнения

4.1. Введем понятие рекурсивного типа, объявляемого следующим образом:

```
RECTYPE T = T0
```

и обозначающего множество значений, определенное типом T0 и расширенное единственным значением NONE. Например, определение типа person можно тогда упростить до

```
RECTYPE person = RECORD name: Name;
                        father, mother: person
                 END
```

Какова схема организации памяти для рекурсивной структуры, соответствующей рис. 4.2? Надо полагать, реализация такого средства должна основываться на динамической схеме распределения памяти, и поля father и mother в приведенном примере должны содержать указатели, порождаемые автоматически, но недоступные программисту. Какие трудности возникнут при реализации такой схемы?

4.2. Определите структуру данных, описанную в последнем абзаце раздела 4.2, в терминах записей и указателей. Можно ли представить такую семейную группу с помощью рекурсивных типов, предложенных в предыдущем упражнении?

4.3. Предположим, что очередь Q, устроенная по принципу «первый вошел – первый вышел» (first-in-first-out = fifo), с элементами типа T0 реализована как связный список. Напишите модуль с подходящей структурой данных, процедурами для вставки и извлечения элемента из Q, а также функцию проверки, пуста очередь или нет. Процедуры должны иметь собственный механизм, обеспечивающий экономное использование памяти.

4.4. Предположим, что записи в связном списке имеют поле ключа типа INTEGER. Напишите программу сортировки списка по возрастанию ключа. Затем постройте процедуру обращения списка.

4.5. Для организации циклических списков (см. рис. 4.52) обычно используют так называемый заголовок списка. Зачем вводят такой заголовок? Напишите процедуры вставки, удаления и поиска элемента по заданному ключу. Сделайте это один раз, предполагая наличие заголовка, и другой раз при его отсутствии.

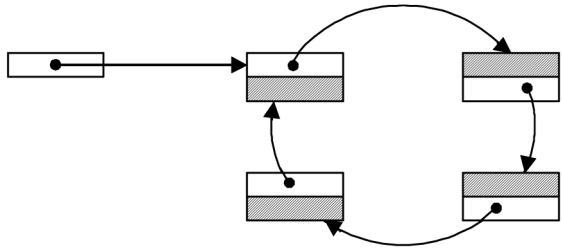


Рис. 4.52. Циклический список

4.6. Двунаправленный список – это список элементов, связанных в обоих направлениях (см. рис. 4.53). Обе цепочки ссылок начинаются в заголовке. Как и в предыдущем упражнении, постройте модуль с процедурами для поиска, вставки и удаления элементов.

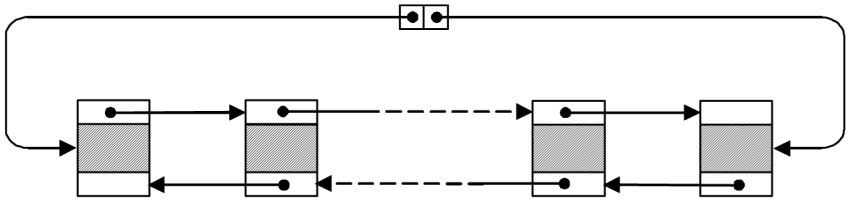


Рис. 4.53. Двунаправленный список

4.7. Будет ли правильно работать представленная программа топологической сортировки, если некоторая пара <x,y> встречается во входных данных более одного раза?

4.8. В программе топологической сортировки сообщение "Набор не является частично упорядоченным" во многих случаях не слишком полезно. Усовершенствуйте программу так, чтобы она выдавала последовательность тех элементов, которые образуют цикл, если такая существует.

4.9. Напишите программу, которая читает программный текст, находит все определения и вызовы процедур и пытается топологически отсортировать процедуры. Пусть P ⟨ Q означает, что процедура P вызывается процедурой Q.

4.10. Нарисуйте дерево, построенное приведенной программой построения идеально сбалансированного дерева, если входные данные – натуральные числа 1, 2, 3, ... , n.

4.11. Какие последовательности узлов получатся при обходе дерева на рис. 4.23 в прямом, центрированном и обратном порядке?

4.12. Найдите правило построения таких последовательностей n чисел, при подаче которых на вход простейшей программе поиска и вставки (раздел 4.4.3) получались бы идеально сбалансированные деревья.

4.13. Рассмотрим следующие два порядка обхода двоичных деревьев:

a1. Обойти правое поддерево.
a2. Посетить корень.
a3. Обойти левое поддерево.

b1. Посетить корень.
b2. Обойти правое поддерево.
b3. Обойти левое поддерево.

Есть ли какие-нибудь простые связи между последовательностями узлов, получающимися в этих двух случаях, и последовательностями, порождаемыми тремя отношениями порядка, определенными в тексте?

4.14. Определите структуру данных для представления n-арных деревьев. Затем напишите процедуру, которая обходит n-арное дерево и порождает двоичное дерево, содержащее те же элементы. Предположите, что каждый ключ, хранимый в элементе, занимает k слов, а каждый указатель – одно слово оперативной памяти. Каков выигрыш по требуемой памяти при использовании двоичного дерева вместо n-арного?

4.15. Пусть дерево строится в соответствии с приводимым ниже определением рекурсивной структуры данных (см. упр. 4.1). Напишите процедуру поиска элемента с заданным ключом x и применения к этому элементу операции P:

```
RECTYPE Tree = RECORD x: INTEGER;
                      left, right: Tree
               END
```

4.16. В некоторой файловой системе директория всех файлов организована как упорядоченное двоичное дерево. Каждый узел соответствует файлу и хранит имя этого файла и, среди прочего, дату последнего обращения к нему, закодированную как целое. Напишите программу, которая обходит это дерево и стирает все файлы, последнее обращение к которым произошло до некоторой даты.

4.17. В некоторой древесной структуре для каждого элемента эмпирически измеряется частота обращений, которая хранится в виде счетчика в соответствующем узле. В определенные моменты времени производится реорганизация дерева: выполняется обход дерева и строится новое дерево с помощью процедуры простого поиска и вставки, причем ключи вставляются в порядке уменьшающихся значений числа обращений. Напишите программу для выполнения такой реорганизации. Будет ли средняя длина пути в этом дереве равна, хуже или намного хуже, чем для оптимального дерева?

4.18. Описанный в разделе 4.5 метод анализа алгоритма вставки в дерево можно также использовать для вычисления среднего числа сравнений C_n и среднего числа пересылок (обменов) M_n, которые делает алгоритм быстрой сортировки при сортировке n элементов массива, предполагая равную вероятность всех $n!$ перестановок n ключей 1, 2, ... n. Постройте соответствующее рассуждение и определите C_n и M_n.

4.19. Нарисуйте сбалансированное дерево с 12 узлами, высота которого минимальна среди всех сбалансированных деревьев с 12 узлами. В какой последовательности должны вставляться узлы, чтобы процедура АВЛ-вставок порождала это дерево?

4.20. Найдите такую последовательность n ключей для вставки, чтобы процедура вставки в АВЛ-дерево выполнила каждый из четырех действий балансировки (LL, LR, RR, RL) хотя бы один раз. Какова минимальная длина n такой последовательности?

4.21. Найдите сбалансированное дерево с ключами 1 ... n и перестановку этих ключей – такую, чтобы при подаче ее на вход процедуре удаления для АВЛ-деревьев каждая из четырех операций балансировки выполнилась хотя бы один раз. Найти такую последовательность минимальной длины.

4.22. Какова средняя длина пути дерева Фибоначчи T_n?

4.23. Напишите программу, которая порождает почти оптимальное дерево в соответствии с алгоритмом, основанном на выборе центроида в качестве корня.

4.24. Пусть ключи 1, 2, 3, ... вставляются в пустое Б-дерево порядка 2. Какие ключи будут вызывать разбиения страниц? Какие ключи вызывают увеличение высоты дерева? Если ключи стираются в том же порядке, какие ключи вызывают слияние страниц (и освобождение памяти) и какие ключи вызывают уменьшение высоты? Ответьте на вопрос (a) для схемы удаления, использующей балансировку, и (b) для схемы без балансировки (когда страница становится пустой, с соседней страницы забирается единственный элемент).

4.25. Напишите программу для поиска, вставки и удаления ключей из двоичного Б-дерева. Используйте тип узла и схему вставки для двоичного Б-дерева, показанную выше.

4.26. Найдите последовательность вставки ключей в пустое симметричное двоичное Б-дерево, которая заставляет представленную процедуру вставки выполнить все четыре операции балансировки (LL, LR, RR, RL) хотя бы по одному разу. Найдите кратчайшую такую последовательность.

4.27. Напишите процедуру удаления элементов для симметричного двоичного Б-дерева. Затем найдите дерево и короткую последовательность удалений – такие, чтобы все четыре ситуации, требующие балансировки, возникали хотя бы по разу.

4.28. Определите структуру данных и напишите процедуры вставки и удаления элемента для приоритетного дерева поиска. Процедуры должны обеспечивать сохранение указанных инвариантов. Сравните их производительность с приоритетными деревьями поиска по остаткам.

4.29. Спроектируйте модуль со следующими процедурами для работы с приоритетными деревьями поиска по остаткам:
– Вставить точку с координатами `x`, `y`.
– Перечислить все точки в заданном прямоугольнике.
– Найти точку с наименьшей координатой `x` в заданном прямоугольнике.
– Найти точку с наибольшей координатой `y` внутри заданного прямоугольника.
– Перечислить все точки, лежащие внутри двух (пересекающихся) прямоугольников.

Литература

[4.1] Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации // Доклады АН СССР, 146 (1962), 263–266.

[4.2] Bayer R. and McCreight E. M. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1, No. 3 (1972), 173–189.

[4.3] Bayer R. and McCreight E. M. Binary B-trees for Virtual memory. Proc. 1971 ACM SIGFIDET Workshop, San Diego, Nov. 1971. P. 219–235.

[4.4] Bayer R. and McCreight E. M. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. Acta Informatica, 1, No. 4 (1972), 290–306.

[4.5] Hu T. C. and Tucker A. C. SIAM J. Applied Math, 21, No. 4 (1971), 514–532.

[4.6] Knuth D. E. Optimum Binary Search Trees. Acta Informatica, 1, No. 1 (1971), 14–25.

[4.7] Walker W. A. and Gotlieb C. C. A Top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees, in: Graph Theory and Computing (New York: Academic Press, 1972), P. 303–323.

[4.8] Comer D. The ubiquitous B-Tree. ACM Comp. Surveys, 11, 2 (June 1979), 121–137.

[4.9] Vuillemin J. A unifying look at data structures. Comm. *ACM*, 23, 4 (April 1980), 229–239.

[4.10] McCreight E. M. Priority search trees. SIAM J. of Comp. (May 1985).

Глава 5

Хэширование

5.1. Введение

В главе 4 подробно обсуждалась следующая основная проблема: если задан набор элементов, характеризующихся ключом (который определяет отношение порядка), то как организовать этот набор, чтобы извлечение элемента с заданным ключом требовало наименьших усилий? Ясно, что в конечном счете доступ к каждому элементу в памяти компьютера осуществляется указанием его адреса в памяти. Поэтому вышеуказанная проблема по сути сводится к нахождению подходящего отображения H ключей (K) в адреса (A):

$H: K \rightarrow A$

В главе 4 это отображение реализовывалось с помощью различных алгоритмов поиска в списках и деревьях на основе разных способов организации данных. Здесь мы опишем еще один подход, простой по сути и во многих случаях очень эффективный. Затем мы обсудим и некоторые его недостатки.

В этом методе данные организуются с помощью массива. Поэтому H является отображением, преобразующим ключи в индексы массива, откуда и происходит название *преобразование ключей*, нередко используемое для этого метода. Заметим, что здесь нам не понадобятся процедуры динамического размещения; массив является одной из фундаментальных, статических структур. Метод преобразования ключей часто используют в тех задачах, где с примерно равным успехом можно применить и деревья.

Фундаментальная трудность при использовании преобразования ключей заключается в том, что множество возможных значений ключей гораздо больше, чем множество доступных адресов в памяти (индексов массива). К примеру, возьмем имена длиной до 16 букв в качестве ключей, идентифицирующих отдельных людей во множестве из тысячи человек. Здесь есть 26^{16} возможных значений ключей, которые нужно отобразить на 10^3 возможных индексов. Очевидно, что функция H отображает несколько значений аргументов в одно значение индекса. Если задан ключ k, то первый шаг операции поиска состоит в вычислении соответствующего индекса h = H(k), а второй – очевидно, обязательный – шаг состоит в проверке того, действительно ли элемент с ключом k соответствует элементу массива (таблицы) T с индексом h, то есть выполняется ли равенство T[H(k)].key = k. Мы сразу сталкиваемся с двумя вопросами:

1. Какую функцию H надо взять?
2. Что делать, если H не смогла вычислить адрес искомого элемента?

Ответ на второй вопрос состоит в том, чтобы использовать метод, который даст альтернативную позицию, скажем индекс h', и если там по-прежнему нет искомого элемента, то третий индекс h", и т. д. (Такие попытки обозначаются ниже как *пробы* (probe) – *прим. перев.*) Ситуацию, когда в вычисленной позиции находится элемент, отличный от искомого, называют *коллизией*; задача порождения альтернативных индексов называется *разрешением коллизий*. Далее мы обсудим выбор функции преобразования ключей и методы разрешения коллизий.

5.2. Выбор хэш-функции

Хорошая функция преобразования ключей должна обеспечивать как можно более равномерное распределение ключей по всему диапазону значений индекса. Других ограничений на распределение нет, но на самом деле желательно, чтобы оно казалось совершенно случайным. Это свойство дало методу несколько ненаучное название *хэширование* (hashing от англ. «превращать в фарш» и «мешанина» – *прим. перев.*). H называется *хэш-функцией*. Очевидно, эта функция должна допускать эффективное вычисление, то есть состоять из очень небольшого числа основных арифметических операций.

Предположим, что имеется функция преобразования ORD(k), которая вычисляет порядковый номер ключа k во множестве всех возможных ключей. Кроме того, предположим, что индекс массива i принимает значения в диапазоне целых чисел 0 .. N-1, где N – размер массива. Тогда есть очевидный вариант:

```
H(k)  =  ORD(k) MOD N
```

Такой выбор обеспечивает равномерное распределение ключей по диапазону индексов и поэтому является основой большинства хэш-функций. Это выражение очень быстро вычисляется, если N есть степень 2, но именно этого случая следует избегать, если ключи являются последовательностями букв. Предположение, что все ключи равно вероятны, в этом случае неверно, и на самом деле слова, отличающиеся лишь немногими буквами, будут с большой вероятностью отображаться на одно и то же значение индекса, так что получится весьма неоднородное распределение. Поэтому особенно рекомендуется в качестве значения N выбирать простое число [5.2]. Как следствие придется использовать полную операцию деления, которую нельзя заменить простым отбрасыванием двоичных цифр, но это не является проблемой на большинстве современных компьютеров, имеющих встроенную инструкцию деления.

Часто используют хэш-функции, состоящие в применении логических операций, таких как исключающее «или», к некоторым частям ключа, представленного как последовательность двоичных цифр. На некоторых компьютерах эти операции могут выполняться быстрее, чем деление, но иногда они приводят к удивительно неоднородному распределению ключей по диапазону индексов. Поэтому мы воздержимся от дальнейшего обсуждения таких методов.

5.3. Разрешение коллизий

Если оказывается, что элемент таблицы, соответствующий данному ключу, не является искомым элементом, то имеет место коллизия, то есть у двух элементов ключи отображаются на одно значение индекса. Тогда нужна вторая проба с некоторым значением индекса, полученным из данного ключа детерминированным способом. Есть несколько способов порождения вторичных индексов. Очевидный способ – связать все элементы с одинаковым первичным индексом H(k) в связный список. Это называют *прямым связыванием* (direct chaining). Элементы этого списка могут находиться в первичной таблице или вне ее; во втором случае об-

ласть памяти, где они размещаются, называется *областью переполнения* (overflow area). Недостатки этого метода – необходимость поддерживать вторичные списки, а также что каждый элемент таблицы должен содержать указатель (или индекс) на список конфликтующих элементов.

Альтернативный способ разрешения коллизий состоит в том, чтобы вообще отказаться от списков и просто перебирать другие элементы в той же таблице, пока не будет найден искомый элемент либо пустая позиция, что означает отсутствие указанного ключа в таблице. Такой метод называется *открытой адресацией* (open addressing [5.3]). Естественно, последовательность индексов во вторичных попытках должна быть всегда одной и той же для заданного ключа. Тогда алгоритм поиска в таблице может быть кратко описан следующим образом:

```
h := H(k); i := 0;
REPEAT
    IF T[h].key = k THEN элемент найден
    ELSIF T[h].key = free THEN элемента в таблице нет
    ELSE (*коллизия*)
        i := i+1; h := H(k) + G(i)
    END
UNTIL найден или нет в таблице (или таблица полна)
```

В литературе предлагались разные функции для разрешения коллизий. Обзор темы, сделанный Моррисом в 1968 г. [4.8], вызвал значительную активность в этой области. Простейший метод – проверить соседнюю позицию (считая таблицу циклической), пока не будет найден либо элемент с указанным ключом, либо пустая позиция. Таким образом, $G(i) = i$; в этом случае индексы h_i, используемые для поиска, даются выражениями

$$h_0 = H(k)$$
$$h_i = (h_{i-1} + i) \text{ MOD } N, i = 1 \ldots N-1$$

Этот способ называется *методом линейных проб* (linear probing). Его недостаток – тенденция элементов к скучиванию вблизи первичных ключей (то есть ключей, не испытавших коллизии при вставке). Конечно, в идеале функция G должна тоже распределять ключи равномерно по множеству свободных позиций. Однако на практике это довольно сложно обеспечить, и здесь предпочитают компромиссные методы, которые не требуют сложных вычислений, но все же работают лучше, чем линейная функция. Один из них состоит в использовании квадратичной функции, так что индексы для последовательных проб задаются формулами

$$h_0 = H(k)$$
$$h_i = (h_0 + i^2) \text{ MOD } N, \; i > 0$$

Заметим, что при вычислении очередного индекса можно обойтись без возведения в квадрат, если воспользоваться следующими рекуррентными соотношениями для $h_i = i^2$ и $d_i = 2i + 1$:

$$h_{i+1} = h_i + d_i$$
$$d_{i+1} = d_i + 2, \; i > 0$$

причем $h_0 = 0$ и $d_0 = 1$. Этот способ называется *методом квадратичных проб* (quadratic probing), и он, в общем, обходит упомянутую проблему скучивания,

практически не требуя дополнительных вычислений. Незначительный недостаток здесь в том, что при последовательных пробах проверяются не все элементы таблицы, то есть при вставке можно не обнаружить свободной позиции, хотя в таблице они еще есть. На самом деле в методе квадратичных проб проверяется по крайней мере половина таблицы, если ее размер N является простым числом. Это утверждение можно доказать следующим образом. Тот факт, что i-я и j-я пробы попадают в один элемент таблицы, выражается уравнением

$$i^2 \text{ MOD } N = j^2 \text{ MOD } N$$
$$(i^2 - j^2) \equiv 0 \quad (\text{modulo } N)$$

Применяя формулу для разности квадратов, получаем

$$(i + j)(i - j) \equiv 0 \quad (\text{modulo } N)$$

и так как $i \neq j$, то заключаем, что хотя бы одно из чисел i или j должно быть не меньше N/2, чтобы получить $i+j = c*N$ с целым c. На практике этот недостаток не важен, так как необходимость выполнять N/2 вторичных проб при разрешении коллизий случается крайне редко, и только если таблица уже почти полна.

В качестве применения описанной техники перепишем процедуру порождения перекрестных ссылок из раздела 4.4.3. Главные отличия – в процедуре search и в замене указательного типа Node глобальной хэш-таблицей слов T. Хэш-функция H вычисляется как остаток от деления на размер таблицы; для разрешения коллизий применяются квардатичные пробы. Подчеркнем, что для хорошей производительности важно, чтобы размер таблицы был простым числом.

Хотя метод хэширования весьма эффективен в этом случае, – даже более эффективен, чем методы, использующие деревья, – у него есть и недостаток. Просмотрев текст и собрав слова, мы, вероятно, захотим создать из них алфавитный список. Это несложно, если данные организованы в виде дерева, потому что принцип упорядоченности – основа этого способа организации. Однако простота теряется, если используется хэширование. Здесь и проявляется смысл слова «хэширование». Для печати таблицы придется не только выполнить сортировку (которая здесь не показана), но оказывается даже предпочтительным отслеживать вставляемые ключи, явным образом связывая их в список. Поэтому высокая производительность метода хэширования при поиске частично компенсируется дополнительными операциями, необходимыми для завершения полной задачи порождения упорядоченного указателя перекрестных ссылок.

```
CONST N = 997;  (*простое, размер таблицы*)          (*ADruS53_CrossRef*)
    WordLen = 32;   (*максимальная длина ключей*)
    Noc = 16;   (*макс. число элементов в слове*)

TYPE
    Word = ARRAY WordLen OF CHAR;
    Table = POINTER TO ARRAY N OF
        RECORD key: Word; n: INTEGER;
            lno: ARRAY Noc OF INTEGER
        END;

VAR line: INTEGER;
```

```
PROCEDURE search (T: Table; VAR a: Word);
    VAR i, d: INTEGER; h: LONGINT; found: BOOLEAN;
    (*используется глобальная переменная line*)
BEGIN
(*вычислить хеш-индекс h для a*)
    i := 0; h := 0;
    WHILE a[i] > 0X DO h := (256*h + ORD(a[i])) MOD N; INC(i) END;
    d := 1; found := FALSE;
    REPEAT
        IF T[h].key = a THEN (*совпадение*)
            found := TRUE; T[h].lno[T[h].n] := line;
            IF T[h].n < Noc THEN INC(T[h].n) END
        ELSIF T[h].key[0] = " " THEN (*новый элемент таблицы*)
            found := TRUE; COPY(a, T[h].key); T[h].lno[0] := line; T[h].n := 1
        ELSE (*коллизия*) h := h+d; d := d+2;
            IF h >= N THEN h := h-N END;
            IF d = N THEN Texts.WriteString(W," Переполнение таблицы"); HALT(88)
            END
        END
    UNTIL found
END search;

PROCEDURE Tabulate (T: Table);
    VAR i, k: INTEGER;
    (*для вывода используется глобальный объект печати W*)
BEGIN
    FOR k := 0 TO N-1 DO
        IF T[k].key[0] # " " THEN
            Texts.WriteString(W, T[k].key); Texts.Write(W, TAB);
            FOR i := 0 TO T[k].n -1 DO Texts.WriteInt(W, T[k].lno[i], 4) END;
            Texts.WriteLn(W)
        END
    END
END Tabulate;

PROCEDURE CrossRef (VAR R: Texts.Reader);
    VAR i: INTEGER; ch: CHAR; w: Word;
        H: Table;
BEGIN
    NEW(H); (*разместить и очистить хеш-таблицу*)
    FOR i := 0 TO N-1 DO H[i].key[0] := " " END;
    line := 0;
    Texts.WriteInt(W, 0, 6); Texts.Write(W, TAB); Texts.Read(R, ch);
    WHILE ~R.eot DO
        IF ch = 0DX THEN (*конец строки*) Texts.WriteLn(W);
            INC(line); Texts.WriteInt(W, line, 6); Texts.Write(W, 9X); Texts.Read(R, ch)
        ELSIF ("A" <= ch) & (ch <= "Z") OR ("a" <= ch) & (ch <= "z") THEN
            i := 0;
            REPEAT
                IF i < WordLen-1 THEN w[i] := ch; INC(i) END;
                Texts.Write(W, ch); Texts.Read(R, ch)
            UNTIL (i = WordLen-1) OR ~(("A" <= ch) & (ch <= "Z")) &
```

```
            ~(("a" <= ch) & (ch <= "z")) & ~(("0" <= ch) & (ch <= "9"));
          w[i] := 0X; (*конец цепочки литер*)
          search(H, w)
        ELSE Texts.Write(W, ch); Texts.Read(R, ch)
        END;
        Texts.WriteLn(W); Texts.WriteLn(W); Tabulate(H)
    END
END CrossRef
```

5.4. Анализ хэширования

Производительность вставки и поиска в методе хэширования для худшего случая, очевидно, ужасная. Ведь нельзя исключать, что аргумент поиска таков, что все пробы пройдут в точности по занятым позициям, ни разу не попав в нужные (или свободные). Нужно иметь большое доверие законам теории вероятности, чтобы применять технику хэширования. Здесь нужна уверенность в том, что в среднем число проб мало. Приводимые ниже вероятностные аргументы показывают, что это число не просто мало, а очень мало.

Снова предположим, что все возможные значения ключей равновероятны и что хэш-функция H распределяет их равномерно по диапазону индексов таблицы. Еще предположим, что некоторый ключ вставляется в таблицу размера N, уже содержащую k элементов. Тогда вероятность попадания в свободную позицию с первого раза равна (N–k)/N. Этой же величине равна вероятность p_1 того, что будет достаточно одного сравнения. Вероятность того, что понадобится в точности еще одна проба, равна вероятности коллизии на первой попытке, умноженной на вероятность попасть в свободную позицию на второй. В общем случае получаем вероятность p_i вставки, требующей в точности i проб:

$$
\begin{aligned}
p_1 &= (N-k)/N \\
p_2 &= (k/N) \times (N-k)/(N-1) \\
p_3 &= (k/N) \times (k-1)/(N-1) \times (N-k)/(N-2) \\
&\cdots\cdots\cdots \\
p_i &= (k/N) \times (k-1)/(N-1) \times (k-2)/(N-2) \times \ldots \times (N-k)/(N-(i-1))
\end{aligned}
$$

Поэтому среднее число E проб, необходимых для вставки k+1-го ключа, равно

$$
\begin{aligned}
E_{k+1} &= \mathbf{S}i\colon 1 \le i \le k+1 \;:\; i \times p_i \\
&= 1 \times (N-k)/N \;+\; 2 \times (k/N) \times (N-k)/(N-1) \;+\; \ldots \\
&\quad + (k+1) * (k/N) \times (k-1)/(N-1) \times (k-2)/(N-2) \times \ldots \times 1/(N-(k-1)) \\
&= (N+1) \,/\, (N-(k-1))
\end{aligned}
$$

Поскольку число проб для вставки элемента совпадает с числом проб для его поиска, этот результат можно использовать для вычисления среднего числа E проб, необходимых для доступа к случайному ключу в таблице. Пусть снова размер таблицы обозначен как N, и пусть m – число ключей уже в таблице. Тогда

$$
\begin{aligned}
E &= (\mathbf{S}k\colon 1 \le k \le m \;:\; E_k) \,/\, m \\
&= (N+1) \times (\mathbf{S}k\colon 1 \le k \le m \;:\; 1/(N-k+2))/m \\
&= (N+1) \times (H_{N+1} - H_{N-m+1})
\end{aligned}
$$

где H – гармоническая функция. H можно аппроксимировать как $H_N = \ln(N) + g$, где g – постоянная Эйлера. Далее, если ввести обозначение a для отношения m/(N+1), то получаем

$$E = (\ln(N+1) - \ln(N-m+1))/a = \ln((N+1)/(N-m+1))/a = -\ln(1-a)/a$$

Величина a примерно равна отношению занятых и свободных позиций; это отношение называется *коэффициентом заполнения* (load factor); a = 0 соответствует пустой таблице, a = N/(N+1) ≈ 1 – полной. Среднее число E проб для поиска или вставки случайного ключа дано в табл. 5.1 как функция коэффициента заполнения.

Таблица 5.1. Среднее число проб E как функция коэффициента заполнения a

a	E
0.1	1.05
0.25	1.15
0.5	1.39
0.75	1.85
0.9	2.56
0.95	3.15
0.99	4.66

Числа получаются удивительные, и они объясняют исключительно высокую производительность метода преобразования ключей. Даже если таблица заполнена на 90%, в среднем нужно только 2,56 пробы, чтобы найти искомый ключ или свободную позицию. Особо подчеркнем, что это число не зависит от абсолютного числа ключей, а только от коэффициента заполнения.

Приведенный анализ предполагает, что применяемый метод разрешения коллизий равномерно рассеивает ключи по оставшимся позициям. Методы, используемые на практике, дают несколько худшую производительность. Детальный анализ метода линейных проб дает следующий результат для среднего числа проб:

$$E = (1 - a/2) / (1 - a)$$

Некоторые численные значения E(a) приведены в табл. 5.2 [5.4].

Результаты даже для простейшего способа разрешения коллизий настолько хороши, что есть соблазн рассматривать хэширование как панацею на все случаи жизни. Тем более что его производительность превышает даже самые изощренные из обсуждавшихся методов с использованием деревьев, по крайней мере с точки зрения числа сравнений, необходимых для поиска и вставки. Но именно поэтому важно явно указать некоторые недостатки хэширования, даже если они очевидны при непредвзятом анализе.

Разумеется, серьезным недостатком по сравнению с методами с динамическим размещением являются фиксированный размер таблицы и невозможность изменять его в соответствии с текущей необходимостью. Поэтому обязательно нужна достаточно хорошая априорная оценка числа обрабатываемых элементов данных, если неприемлемы плохое использование памяти или низкая производительность (или переполнение таблицы). Даже если число элементов известно точно, – что бывает крайне редко, – стремление к хорошей производительности заставляет выбирать таблицу немного большего размера (скажем, на 10%).

Таблица 5.2. Среднее число проб для метода линейных проб

a	E
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Второй серьезный недостаток методов «рассеянного хранения» становится очевидным, если ключи нуж-

но не только вставлять и искать, но и удалять. Удаление элементов в хэш-таблице – чрезвычайно громоздкая операция, если только не использовать прямое связывание в отдельной области переполнения. Поэтому разумно заключить, что древесные способы организации по-прежнему привлекательны и даже предпочтительны, если объем данных плохо предсказуем, сильно меняется и даже может уменьшаться.

Упражнения

5.1. Если объем информации, связанной с каждым ключом, относительно велик (в сравнении с самим ключом), то ее не следует хранить в хэш-таблице. Объясните, почему, и предложите схему представления такого набора данных.

5.2. Рассмотрите решение проблемы скучивания, состоящее в построении деревьев вместо списков переполнения, то есть в организации ключей, претерпевших коллизию, в виде древесной структуры. Тогда каждый элемент хэш-таблицы может рассматриваться как корень (возможно, пустого) дерева. Сравните среднюю производительность такого метода хэширования с деревьями с производительностью метода открытой адресации.

5.3. Придумайте схему, в которой для разрешения коллизий при вставках и удалениях из хэш-таблицы используются квадратичные приращения. Проведите экспериментальное сравнение такой схемы с простым двоичным деревом, используя случайную последовательность ключей для вставки и удаления.

5.4. Главный недостаток методов, использующих хэш-таблицы, – в том, что размер таблицы фиксируется, когда окончательное число элементов еще не известно. Предположим, что в вашей вычислительной системе есть механизм динамического распределения памяти, позволяющий получить память в любой момент. Тогда если хэш-таблица H заполнена (или почти заполнена), создается большая таблица H', и все ключи переносятся из H в H', после чего область памяти из-под H возвращается системе управления памятью. Это так называемое *повторное хэширование*. Напишите программу, которая выполняет повторное хэширование таблицы H размера N.

5.5. Очень часто ключи – не целые числа, а последовательности литер. Такие слова могут сильно варьироваться по длине, и поэтому хранить их в полях фиксированного размера неудобно и расточительно. Напишите программу, которая работает с хэш-таблицей и с ключами переменной длины.

Литература

[5.1] Maurer W. D. An Improved Hash Code for Scatter Storage. Comm. *ACM*, 11, No. 1 (1968), 35–38.

[5.2] Morris R. Scatter Storage Techniques. Comm. ACM, 11, No. 1 (1968), 38–43.

[5.3] Peterson W. W. Addressing for Random-access Storage. IBM J. Res. & Dev., 1 (1957), 130–146.

[5.4] Schay G. and Spruth W. Analysis of a File Addressing Method. Comm. ACM, 5, No. 8 (1962) 459–462.

Приложение А
Множество символов ASCII

	0	10	20	30	40	50	60	70
0	nul	dle		0	@	P	'	p
1	soh	dc1	!	1	A	Q	a	q
2	stc	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
A	lf	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[	k	{
C	ff	fs	,	<	L	\	l	|
D	cr	gs	-	=	M	]	m	}
E	so	rs	.	>	N	^	n	~
F	si	us	/	?	O	_	o	del

Приложение B
Синтаксис Оберона

1. Синтаксис Оберона

```
ident  =  letter {letter | digit}.
number  =  integer | real.
integer  =  digit {digit} | digit {hexDigit} "H" .
real  =  digit {digit} "." {digit} [ScaleFactor].
ScaleFactor  =  ("E" | "D") ["+" | "-"] digit {digit}.
hexDigit  =  digit | "A" | "B" | "C" | "D" | "E" | "F".
digit  =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant  =  """ character """ | digit {hexDigit} "X".
string  =  """ {character} """ .
qualident = [ident "."] ident.
identdef = ident ["*"].

TypeDeclaration  =  identdef "=" type.
type  =  qualident | ArrayType | RecordType | PointerType | ProcedureType.
ArrayType  =  ARRAY length {"," length} OF type.
length  =  ConstExpression.
RecordType  =  RECORD ["(" BaseType ")"] FieldListSequence END.
BaseType  =  qualident.
FieldListSequence  =  FieldList {";" FieldList}.
FieldList  =  [IdentList ":" type].
IdentList  =  identdef {"," identdef}.
PointerType  =  POINTER TO type.
ProcedureType = PROCEDURE [FormalParameters].

VariableDeclaration  =  IdentList ":" type.

designator  =  qualident {"." ident | "[" ExpList "]" | "(" qualident ")" | "^" }.
ExpList  =  expression {"," expression}.
expression  =  SimpleExpression [relation SimpleExpression].
relation  =  "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression  =  ["+"|"-"] term {AddOperator term}.
AddOperator  =  "+" | "-" | OR .
term  =  factor {MulOperator factor}.
MulOperator  =  "*" | "/" | DIV | MOD | "&" .
factor  =  number | CharConstant | string | NIL | set |
           designator [ActualParameters] | "(" expression ")" | "~" factor.
set  =  "{" [element {"," element}] "}".
element  =  expression [".." expression].
ActualParameters  =  "(" [ExpList] ")" .
```

```
statement = [assignment | ProcedureCall |
    IfStatement | CaseStatement | WhileStatement | RepeatStatement |
    LoopStatement | WithStatement | EXIT | RETURN [expression] ].
    assignment = designator ":=" expression.
ProcedureCall = designator [ActualParameters].
IfStatement = IF expression THEN StatementSequence
    {ELSIF expression THEN StatementSequence}
    [ELSE StatementSequence]
    END.
CaseStatement = CASE expression OF case {"|" case} [ELSE StatementSequence] END.
Case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
WhileStatement = WHILE expression DO StatementSequence END.
LoopStatement = LOOP StatementSequence END.
WithStatement = WITH qualident ":" qualident DO StatementSequence END .

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading = PROCEDURE identdef [FormalParameters].
ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
ForwardDeclaration = PROCEDURE "^" identdef [FormalParameters].
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection = [VAR] ident {"," ident} ":" FormalType.
FormalType = {ARRAY OF} qualident.

DeclarationSequence = {CONST {ConstantDeclaration ";"} |
    TYPE {TypeDeclaration ";"} | VAR {VariableDeclaration ";"}}
    {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
Module = MODULE ident ";" [ImportList] DeclarationSequence
    [BEGIN StatementSequence] END ident "." .
ImportList = IMPORT import {"," import} ";" .
Import = ident [":=" ident].
```

2. Символы и ключевые слова

+	:=	ARRAY	IS	TO
-	^	BEGIN	LOOP	TYPE
*	=	CASE	MOD	UNTIL
/	#	CONST	MODULE	VAR
~	<	DIV	NIL	WHILE
&	>	DO	OF	WITH
.	<=	ELSE	OR	
,	>=	ELSIF	POINTER	
;	..	END	PROCEDURE	
|	:	EXIT	RECORD	
(	)	IF	REPEAT	
[	]	IMPORT	RETURN	
{	}	IN	THEN	

3. Стандартные типы данных

CHAR, BOOLEAN, SHORTINT	(8 бит)
INTEGER	(16 or 32 bits)
LONGINT, REAL, SET	(32 bits)
LONGREAL	(64 bits)

4. Стандартные функции и процедуры

Имя	Тип аргумента	Тип результата	
ABS(x)	числовой	тип x	абсолютное значение
ODD(x)	целый тип	BOOLEAN	x MOD 2 = 1
CAP(x)	CHAR	CHAR	соответствующая заглавная буква
ASH(x, n)	целые типы	LONGINT	$x \times 2^n$ (арифметический сдвиг)
LEN(v, n)	v: массивовый тип	LONGINT	длина v в n-м измерении
LEN(v)	v: массивовый тип	LONGINT	длина v в 0-м измерении
ORD(x)	CHAR	INTEGER	числовой код литеры x
CHR(x)	INTEGER	CHAR	литера с числовым кодом x
SHORT(x)	LONGINT	INTEGER	тождество (возможна потеря цифр!)
	INTEGER	SHORTINT	
	LONGREAL	REAL	
LONG(x)	SHORTINT	INTEGER	тождество
	INTEGER	LONGINT	
	REAL	LONGREAL	
ENTIER(x)	вещественный тип	LONGINT	наибольшее целое, не превывающее x
INC(v, n)	целые типы		v := v + n
INC(v)	целые типы		v := v + 1
DEC(v, n)	целые типы		v := v – n
DEC(v)	целые типы		v := v –1
INCL(v, n)	v: SET; n: целый тип		v := v + {n}
EXCL(v, n)	v: SET; n: целый тип		v := v – {n}
COPY(x, v)	x: массив литер, цепочка литер		v := x
NEW(v)	указательный тип		разместить v^
HALT(x)	целая константа		остановить вычисление

Приложение C
Цикл Дейкстры

Дейкстра [C.1] (см. также [C.2]) ввел многоветочный вариант цикла WHILE, обосновав его в своей теории систематического вывода императивных программ. Этот цикл оказывается удобным для выражения и, главное, верификации алгоритмов, обычно записываемых через вложенные циклы, позволяя резко снизить усилия по отладке.

В версии языка Оберон, известной как Оберон-07 и представленной в [C.3], синтаксис этого цикла определяется следующим правилом:

```
WhileStatement
= WHILE логическое выражение DO
      последовательность операторов
  {ELSIF логическое выражение DO
      последовательность операторов}
  END.
```

Если вычисление любого из логических выражений (охран) дает TRUE, то выполняется соответствующая последовательность операторов. Вычисление охран и выполнение соответствующих последовательностей повторяются до тех пор, пока все охраны не будут давать FALSE. Таким образом, постусловием для этого цикла является конъюнкция отрицаний всех охран.

Пример:

```
WHILE m > n DO m := m - n
ELSIF n > m DO n := n - m
END
```

Постусловие: ~(m > n) & ~(n > m), что эквивалентно n = m.

Инвариант цикла должен выполняться в начале и в конце каждой ветви.

Грубо говоря, обычно n-веточный цикл Дейкстры соответствует конструкциям из n обычных циклов, каким-то образом вложенных друг в друга.

Удобство цикла Дейкстры обусловлено тем, что вся логика сколь угодно сложного цикла выводится на один уровень и радикально проясняется, причем алгоритмы с разными случаями запутанных циклов трактуются совершенно единообразно.

Эффективный способ построения такого цикла состоит в том, чтобы перечислять все возможные ситуации, которые могут возникнуть, описывая их соответствующими охранами, и для каждой из них независимо от остальных строить операции, обеспечивающие продвижение по данным, вместе с операциями, обеспечивающими восстановление инварианта. Перечисление ситуаций заканчивается, когда дизъ-

юнкция (или) всех охран покроет предполагаемое предусловие цикла. При этом задача корректного построения цикла облегчается, если отложить беспокойство о порядке вычисления охран и выполнения ветвей, экономии операций при вычислении охран и т. п. до тех пор, пока цикл не будет корректно построен. Такие мелкие оптимизации редко по-настоящему важны, а их реализация сильно облегчается, когда корректность сложного цикла уже обеспечена.

Хотя в теории Дейкстры последовательность выбора ветвей цикла и вычисления соответствующих охран не определена, в этой книжке принято, что охраны вычисляются в текстуальном порядке.

Во многих языках программирования цикл Дейкстры приходится моделировать. В старых версиях Оберона, включая Компонентный Паскаль, достаточно использовать цикл LOOP, тело которого состоит из многоветочного условного оператора с оператором выхода в ветке ELSE:

```
LOOP IF логическое выражение THEN
    последовательность операторов
{ELSIF логическое выражение THEN
    последовательность операторов}
ELSE EXIT END END.
```

В других языках конструкция может быть более громоздкой, но это с лихвой окупается упрощением построения и отладки сложных циклов.

Литература

[C.1] Dijkstra E. W. A Discipline of Programming. Prentice-Hall, 1976 (имеется перевод: Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978).

[C.2] Gries D. The Science of Programming. Springer-Verlag, 1981 (имеется перевод: Грис Д. Наука программирования. – М.: Мир, 1984).

[C.3] Wirth N. The Programming Language Oberon. Revision 1.9.2007.

Предметный указатель

Никлаус Вирт

Алгоритмы и структуры данных
Новая версия для Оберона + CD

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод *Ткачев Ф. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Харевская И. В.*

Подписано в печать 05.10.2009. Формат 70×100 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 36. Тираж 1000 экз.
№

Web-сайт издательства: www.dmk-press.ru
Internet-магазин: www.alians-kniga.ru