Н.Вирт

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

М.: Мир, 1989, 360 стр.

Предметный указатель

ОТ ПЕРЕВОДЧИКА

Предлагаемая читателям книга — это новая работа Н. Вирта, специалиста, хорошо известного в кругу программистов. Впрочем, сразу надо сделать уточнение: эта книга представляет собой переработанную версию предыдущей книги этого же автора «Algorithms + Data Structures = Programs» (русский перевод: Алгоритмы + структуры данных = программы, М.: Мир, 1985). Эра информатики породила совершенно новую возможность: книга уже не только может переводиться на другой «человеческий» язык, ее можно переложить на другой язык программирования, так же как симфонию можно переложить для фортепиано. Все изложение в книге ведется на языке Модула-2, а не на Паскале. Но несмотря на это книга, ее сюжет, остаются прежними: как надо программировать.

Предполагается, что читатель — достаточно опытный программист, поэтому азы этого искусства здесь не затрагиваются. Скорее можно сказать, что делается попытка привить человеку некоторый стиль программирования. Хотя, конечно же, такой опытный учитель, как Н. Вирт, не делает даже и попыток определить, что же такое этот стиль. Просто его нужно впитывать «с молоком матери». В книге дается масса примеров с различного рода комментариями к ним. Чаще всего эти комментарии связаны с оценками производительности алгоритмов и программ, хотя попадаются и другие темы. Приведенные примеры достаточно небольшие, чтобы их можно было воспринять целиком, и в то же время разумно сложные, что позволяет им быть интересными и содержательными.

Теперь несколько критических замечаний. Опытные лекторы знают, как интересно читать первый раз новый курс и как скучно повторять его даже при новых слушателях, обеспечивающих каждый раз новые обратные связи. А здесь пришлось второй раз переписывать всю книгу. Поэтому она оказалась во многом неравноценной первой. Все прежние разделы как бы постарели. Однако новый раздел о поиске в массиве символов написан, как и раньше, легко и просто. В то же время, и это надо признать, новый раздел о поиске в многомерном пространстве написан весьма поверхностно даже с точки зрения просто постановки задачи и никак не иллюстрирует какие-либо принципы самого программирования.

Несколько слов о терминологии. Возможно, некоторых из читателей шокирует появление таких прилагательных, как «массивовый», «записной», «последовательностный», образованных от слов «массив», «запись», «последовательность», употребляемых как определения к слову «тип». Вообще, надо сказать, перевод слова «type» как «тип» крайне неудачен, поскольку в русском языке слово «тип» несет очень большую нагрузку как служебное слово, и отказ от его использования в работах по программированию на языках высокого уровня, наверное, обедняет язык. Попробуйте в разговоре о языке программирования употребить оборот: «объект такого типа» и вы сразу же получите от строгого слушателя или читателя вопрос: «Какого «такого» типа?». Поэтому выражения типа (!) «array type» следовало бы переводить как «тип массив» (или «тип-массив»), что в первое время и делалось. Однако из-за небрежности авторов, не заботившихся о строгости определения языков программирования, выражение «array type» стало переводиться как «тип массива», и сразу же возникли вопросы: «О каком массиве идет речь?». Поэтому при переводе было принято решение ввести в этом случае выражения «массивовый тип» и подобные ему. Вообще переводчик придерживался такого правила: определение некоторого объекта с помощью прилагательного характеризует свойство самого объекта, а с помощью прямого дополнения — принад-

лежность объекта. (Конечно, строго говоря, принадлежность — это такое же свойство, но мы его здесь толкуем более «прямолинейно».) Поэтому, например, вы не встретите в переводе выражений типа «программные ошибки» в смысле «ошибки в программе» и аналогичных выражений, которые с легкой руки авторов работ о компьютерах встречаются в литературе.

В целом, остается надеяться, что упомянутые выше «новшества» терминологии, введенные при переводе, и сделанные выше замечания никак не повлияют на общую весьма и весьма высокую оценку книги. Уверен, что эта новая книга Н. Вирта будет с интересом встречена читателями.

Д. Б. Подшивалов

ПРЕДИСЛОВИЕ

В последние годы программирование для вычислительных машин выделилось в некоторую дисциплину, владение которой стало основным и ключевым моментом, определяющим успех многих инженерных проектов, а сама она превратилась в объект научного исследования. Из ремесла программирование перешло в разряд академических наук. Первый крупный вклад в ее становление сделали Э. Дейкстра и Ч. Хоар. «Заметки по структурному программированию» Дейкстры [1] заставили взглянуть на программирование как на объект, требующий научного подхода и бросающий определенный интеллектуальный вызов; такой подход даже получил название «революции» в программировании. В статье «Аксиоматическая основа программирования для вычислительных машин» [2] Хоар продемонстрировал, что программы поддаются точному анализу, основанному на строгих математических рассуждениях. В этих работах убедительно показано, что можно избежать многих ошибок, традиционных для программистов, если последние будут осмысленно пользоваться методами и приемами, которые раньше они применяли интуитивно, часто не осознавая их как таковые. При этом основное внимание уделяется построению и анализу программ, а более точно — структуре алгоритмов, представляемых текстом программы. Причем совершенно очевидно, что систематический и научный подход прежде всего применим к большим, комплексным программам, работающим со сложными данными. Таким образом, методология программирования должна включать все аспекты строения данных. В конечном

счете *программы* представляют собой конкретные, основанные на некотором реальном представлении и *строении данных* воплощения абстрактных *алгоритмов*. Важный вклад в упорядочение терминологии и в концепции строения данных внесла работа Хоара «О структурной организации данных» [3]. Стало ясно, что решение о том, как представлять данные, невозможно принимать, не зная, какие алгоритмы будут к ним применяться, и наоборот, выбор алгоритма часто очень сильно зависит от строения данных, к которым он применяется. Короче говоря, структура программы и строение данных неразрывно связаны между собой.

Итак, у нас были две причины начать книгу главой о строении данных. Во-первых, мы интуитивно понимаем, что данные предшествуют алгоритму, ведь прежде, чем выполнять какие-либо операции, нужно иметь объекты, к которым они применяются. Во-вторых, и это более непосредственная причина, мы предполагаем, что читатель уже знаком с основными понятиями программирования. По традиции, достаточно оправданной, вводные курсы программирования посвящаются в основном алгоритмам, оперирующим данными с относительно простой структурой. Поэтому казалось вполне разумным посвятить начальную главу строению данных.

На протяжении всей книги, и в частности, в гл. 1, мы следуем теории (и терминологии), развитой в работе Хоара и воплощенной в языке программирования Паскаль [4]. Суть этой теории состоит в том, что данные прежде всего представляют собой абстракции реальных объектов и их предпочтительно рассматривать как некоторые абстрактные образования со структурами, не обязательно предусмотренными в общераспространенных языках программирования. В процессе проектирования программы представление данных постепенно, по мере уточнения самого алгоритма, все более проясняется и все более согласуется с ограничениями, накладываемыми конкретной системой программирования [5]. Поэтому мы определим несколько основных строительных конструкций — *структур данных*, которые назовем

фундаментальными структурами. Очень важно, что эти конструкции довольно легко реализуются на существующих машинах, ведь только в этом случае их можно считать действительно элементами фактического представления данных, т. е. «молекулами», обусловленными последними, заключительными этапами уточнения описания данных. Такими структурами будут: *запись*, *массив* (фиксированного размера) и *множество*. Неудивительно, что эти основные строительные блоки соответствуют математическим понятиям, отражающим фундаментальные идеи.

Краеугольным камнем всей теории структур данных является различие между фундаментальными и усложненными структурами. Фундаментальные структуры — это молекулы (в свою очередь образованные из атомов), из которых строятся составные структуры. Переменные фундаментальных структур могут изменять только свое значение, а структура их и множество допустимых значений остаются неизменными. В результате размер памяти, занимаемой такими переменными, остается постоянным. Для переменных же усложненной структуры характерна способность изменять в процессе выполнения программы и значения, и структуру. Поэтому для их реализации требуются более изощренные методы. В такой классификации последовательности занимают промежуточное место. Конечно, длина их меняется, но ведь это изменение структуры носит тривиальный характер. Поскольку последовательности играют поистине фундаментальную роль во всех практически значимых вычислительных системах, то мы их включили в гл. 1.

Во второй главе речь идет об *алгоритмах сортировки.* Здесь приводится много различных методов решения одной и той же задачи. Математический анализ некоторых из алгоритмов подчеркивает их недостатки и достоинства и помогает программисту осознать важность такого анализа при выборе способа решения поставленной задачи. Разделение на методы сортировки массивов и методы сортировки файлов (часто называемые внутренней и внешней сортировкой) демонстрирует решающее влияние пред-

ставления данных на выбор алгоритма и его сложность. Мы не уделяли бы сортировкам так много внимания, если бы нам не казалось, что они представляют собой идеальный объект для иллюстрации многих принципов программирования и ситуаций, встречающихся в других задачах. Создается впечатление, что на примерах сортировок можно построить целый курс программирования.

Есть тема, которую часто опускают во вводных курсах программирования, хотя она и играет важную концептуальную роль во многих алгоритмах, — это рекурсия. Поэтому мы посвящаем третью главу *рекурсивным решениям*. В ней показано, что рекурсия — обобщение понятия повторения (итерации), и как таковая она представляет собой важную и мощную концепцию программирования. К несчастью, во многих курсах программирования рекурсия используется в примерах, где достаточно простой итерации. Вместо этого мы в гл. 3 сосредоточиваемся на нескольких примерах задач, в которых рекурсия приводит к наиболее естественным решениям, тогда как итерации порождали бы громоздкие и трудные для понимания программы. Идеальным применением рекурсий представляется класс алгоритмов *с возвратом*, но наиболее очевидно их использование в алгоритмах обработки данных, которые сами определяются рекурсивно. Такие задачи разбираются в последних двух главах, а третья глава обеспечивает хорошую подготовку для их понимания.

В гл. 4 мы обращаемся к *данным с динамической структурой*, т. е. данным, чья структура изменяется по мере выполнения программы. Здесь показано, что данные с рекурсивной структурой образуют важный подкласс обычно используемых динамических структур. Хотя рекурсивные определения в этих случаях и естественны, и допустимы, тем не менее на практике их обычно не используют. Вместо этого пользуются в реализации механизмом более наглядным для программиста: он имеет дело с явными указателями или *ссылочными переменными*. Так что в некотором смысле наша книга отражает современную точку зрения на эту проблему. В четвертой главе мы

вводим в программирование ссылки, списки, деревья и приводим примеры, требующие данных еще более сложного строения. Здесь речь идет о том, что часто (и не совсем верно) называют *«обработкой списков»*. Значительное место отводится организации деревьев, и в частности, деревьев поиска. Заканчивается глава рассмотрением рассеянных таблиц и метода функций расстановки, который часто предпочитают деревьям поиска. Это позволяет нам сравнить между собой два принципиально разных метода, используемых для решения часто встречающейся задачи.

Программирование — это *конструирование*. Как можно научить этой конструкторской, изобретательской деятельности? Один из приемов — выделить на многих примерах элементарные принципы композиции и продемонстрировать их некоторым систематическим образом. Но ведь программирование относится к очень быстро изменяющейся и часто требующей больших интеллектуальных затрат деятельности. Поэтому кажется, что сведение дела к нескольким простым рецептам было бы ошибкой. И в нашем арсенале приемов обучения остается только одно: привести несколько тщательно подобранных, очень хороших примеров. Не следует, естественно, надеяться, что изучение примеров окажется в равной мере полезным всем. Для такого подхода характерно то, что многое зависит от учащегося, от его сообразительности и интуиции. Это особенно справедливо для относительно сложных и длинных примеров программ. Они не случайно включены в эту книгу. Длинные программы на практике встречаются чаще всего, и они лучше всего подходят для демонстрации того неуловимого, но важного свойства, которое называют стилем или мастерством. Кроме того, они служат упражнениями в искусстве чтения программ, которым часто пренебрегают, сосредоточивая внимание на их написании. Именно поэтому и включаются в качестве примеров целиком большие программы. Мы проводим читателя по всем этапам постепенного создания программ, показываем ему как бы серию «моментальных снимков» ее рождения, так мы его знакомим с *«методом последовательных уточнений»*

деталей. Я считаю важным, показав программу в окончательном виде, обращать внимание на детали, поскольку именно в них часто кроется трудность программирования. Описание принципов самого алгоритма и его математический анализ могут стимулировать «академический ум» и бросить ему вызов, но это было бы бесчестно по отношению к программисту-практику. Поэтому я строго придерживаюсь правила давать программы в окончательном виде, на том языке, на котором они действительно могут быть выполнены на вычислительной машине.

Конечно, сразу же возникает проблема представления: хотелось, чтобы программы можно было и выполнять и чтобы они были достаточно машинно-независимы, ведь их нужно включать в текст. Для этого не подходят ни общераспространенные языки, ни абстрактная нотация. Нужный компромисс обеспечивается языком Паскаль: именно с такими целями он и создавался. Поэтому в книге мы и будем им пользоваться. Программисты, знакомые с другими языками программирования высокого уровня, такими, как Алгол 60 или ПЛ/1, легко разберутся в приводимых программах, поскольку его понятия и нотация в тексте поясняются. Однако это не значит, что предварительная подготовка не нужна. Идеальную подготовку дает книга «Систематическое программирование» [6], поскольку в ее основу тоже положен Паскаль. Тем не менее она не может служить учебником языка Паскаль — для этого существуют более подходящие книги [7].

Предлагаемая книга представляет собой сжатое и переработанное изложение нескольких курсов программирования, прочитанных в Федеральном технологическом институте (ETH) в Цюрихе. Многими идеями и взглядами, изложенными в этой книге, я обязан беседам с моими коллегами в ETH. Мне хотелось бы, в частности, поблагодарить г-на Г. Сандмейера за тщательное прочтение рукописи и г-жу Хейди Тейлер за внимание и терпение при перепечатке текста. Я хотел бы также отметить большое влияние, оказанное заседаниями рабочих групп 2.1 и 2.3 ИФИПа, и особенно многочисленными беседа-

ми, которые я вел при этом с Э. Дейкстрой и Ч. Хоаром. И наконец, что не менее важно, ETH щедро предоставлял вычислительные машины и обеспечивал условия, без которых была бы невозможна подготовка этой книги.

Цюрих, август 1975 *Н. Вирт*

ЛИТЕРАТУРА

[1] Dahl O., Dijkstra E., Hoare C. A. R. Structured Programming 1972, pp. 1—82. [Имеется перевод: Дал О., Дейкстра Э., Хоар К. Структурное программирование: Пер. с англ. — М.: Мир, 1975.]
[2] Comm. ACM, 12, No. 10, 1969, pp. 576—583.
[3] См. [1].
[4] Wirth N. The Programming Language Pascal. Acta Informatica, 1, No. 1, 1971, pp. 35—63.
[5] Wirth N. Program Development by Stepwise Refinement. Comm. ACM, 14, No. 4 (1971), pp. 221—227.
[6] Wirth N. Systematic Programming, 1973. [Имеется перевод: Вирт Н. Систематическое программирование. Введение. — М.: Мир, 1977.]
[7] Jensen K., 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.]

ПРЕДИСЛОВИЕ К ИЗДАНИЮ 1986 ГОДА

В новом издании пересмотрены многие детали алгоритмов и, кроме того, сделаны некоторые изменения, носящие более принципиальный характер. Все это объясняется опытом, накопленным за те десять лет, которые прошли с момента появления первого издания. Однако большая часть содержимого книги и ее стиль остались без изменения. Остановимся кратко на главных изменениях.

Основное изменение, коснувшееся всего текста целиком, связано с языком программирования, на котором записаны все алгоритмы. Паскаль был заменен на Модулу-2. Хотя такое решение не оказывает определяющего влияния на представление алгоритмов, тем не менее оно оправдывается более простой и более элегантной синтаксической структурой языка Модула-2, часто приводящей к более прозрачному представлению структуры алгоритмов. Кроме того, кажется целесообразным употреблять такую нотацию, которая быстро приобретает признание широких слоев общества (программистов. — *Перев.*), поскольку она хорошо подходит для создания больших систем программ. Как бы то ни было, совершенно очевидно, что «родителем» Модулы был Паскаль, и это облегчало задачу перехода от одного языка к другому. Для простоты весь синтаксис языка Модула приведен в приложении.

Прямым следствием перехода на другой язык программирования была переделка разд. 1.11, посвященного структуре последовательного файла. В Модуле-2 нет встроенного файлового типа. В переписанном разд. 1.11 в качестве более общей структуры данных представлена концепция *последовательности*

и приведена, в частности, некоторая система программ-модулей, поддерживающая в Модуле-2 такие последовательности.

Новой является и последняя часть гл. 1. В ней обсуждаются проблемы поиска. Начав с линейного поиска и поиска делением пополам, мы приходим к недавно изобретенным алгоритмам быстрого поиска в строке. Здесь мы, в частности, для демонстрации правильности представленных алгоритмов используем метод доказательства, основанный на инвариантах циклов и других утверждениях о программах.

Новый раздел о *деревьях приоритетного поиска* заканчивает и главу о динамических структурах данных. В нем определяются деревья, опять же еще не открытые в то время, когда вышло первое издание. Такие деревья допускают экономное представление и быстрый поиск в множестве точек, лежащих на плоскости.

Целиком опущена пятая глава первого издания. В ней речь шла о построении трансляторов, и она выглядела несколько изолированно от предыдущих глав. Тему трансляторов следовало бы более подробно разобрать в специальной книге.

И наконец, появление нового издания связано с событиями, оказавшими за последние десять лет глубокое влияние на издательское дело: использованием вычислительных машин и сложных алгоритмов для подготовки документов и их набора. Наша книга была отредактирована автором и подготовлена для печати с помощью редактора документов *Lara* на машине *Lilith*. Без этих инструментов книга не только оказалась бы более дорогой, но и не была бы еще закончена.

Пало-Алто *Н. Вирт*

НОТАЦИЯ

В нашей книге используется следующая позаимствованная из работ Э. Дейкстры системы нотации.

В логических выражениях символ & обозначает конъюнкцию и произносится как *и*. Символ $\sim$ соответствует отрицанию и читается как *не*. Буквами **A** и **E** (полужирный шрифт) обозначаются соответственно кванторы универсальности и существования. Ниже приводятся употребляемые нами формулы (слева) и их определение в традиционной записи. Обратите внимание, что слева мы избегаем использования символа «...», апеллирующего к интуиции читателя.

Ai: $m \leqslant i < n$: $P_i \equiv P_m \,\&\, P_{m+1} \,\&\, \ldots \,\&\, P_{n-1}$

Здесь P_i — предикаты, и формула утверждает, что истинны все P_i для i, пробегающего, начиная с заданного значения m, все значения вплоть до n (само n сюда не входит).

Ei: $m \leqslant i < n$: $P_i \equiv P_m$ *или* P_{m+1} *или* ... *или* P_{n-1}

Опять, P_i — предикаты и формула утверждает, что истинен хотя бы один предикат P_i для i, пробегающего значения от m до n (исключая само n)

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

MIN i: $m \leqslant i < n$: $x_i = \text{minimum}\,(x_m, x_{m+1}, \ldots, x_{n-1})$

MAX i: $m \leqslant i < n$: $x_i = \text{maximum}\,(x_m, x_{m+1}, \ldots, x_{n-1})$

С помощью кванторов операторы min и max можно выразить так:

MIN i: $m \leqslant i < n$: $x_i = \min \equiv$

$(\mathbf{A}i\text{: } m \leqslant i < n\text{: } \min \leqslant x_i) \,\&\, (\mathbf{E}_i\text{: } m \leqslant i < n\text{: } \min = x_i)$

MAX i: $m \leqslant i < n$: $x_i = \max \equiv$

$(\mathbf{A}i\text{: } m \leqslant i < n\text{: } \max \geqslant x_i) \,\&\, (\mathbf{E}i\text{: } m \leqslant i < n\text{: } \max = x_i)$

I. ОСНОВНЫЕ ПОНЯТИЯ СТРУКТУР ДАННЫХ

1.1. ВВЕДЕНИЕ

Современная вычислительная машина была изобретена и мыслилась как некоторое устройство, облегчающее и убыстряющее проведение сложных расчетов, требующих на свое выполнение значительного времени. Теперь в большинстве случаев их доминирующим свойством и основной характеристикой считается способность хранить огромные объемы информации, к которым можно просто обратиться. Способность же проводить вычисления, т. е. выполнять арифметические действия во многих ситуациях, уже стала почти несущественной.

Во всех таких случаях значительные объемы информации, подлежащей обработке, в некотором смысле представляют *абстракцию* некоторого фрагмента реального мира. Информация, поступающая в машину, состоит из определенного множества *данных*, относящихся к какой-то проблеме, — это именно те данные, которые считаются относящимися к данной конкретной задаче и из которых, как мы надеемся, можно получить (вывести) желанный ответ. Мы говорим о данных как об абстрактном представлении реалий, поскольку некоторые свойства и характеристики объектов при этом игнорируются. Считается, что для конкретной задачи они не являются существенными, определяющими. Поэтому абстрагирование — это *упрощение* фактов.

В качестве примера можно взять, скажем, список (файл) некоторых служащих. Каждый служащий в этом списке представляется (абстрагируется) как некоторое множество данных, относящихся либо к тому, что он делает, либо к процедурам, «обрабатывающим» его деятельность. Это множество может

включать и некоторые идентифицирующие данные вроде его (или ее) фамилии или оклада. Однако маловероятно, что сюда будут включаться такие посторонние сведения, как цвет волос, вес или рост.

Решая любую задачу с помощью машины или без нее, необходимо выбрать уровень абстрагирования, т. е. определить множество данных, представляющих реальную ситуацию. При выборе следует руководствоваться той задачей, которую необходимо решить. Затем надлежит выбрать способ представления этой информации. Здесь уже необходимо ориентироваться на те средства, с помощью которых решается задача, т. е. учитывать возможности, предоставляемые вычислительной машиной. В большинстве случаев эти два этапа не бывают полностью независимыми.

Выбор представления данных часто является довольно трудной проблемой, ибо не определяется однозначно доступными средствами. Всегда нужно принимать во внимание и операции, которые выполняются над этими данными. Вот хороший пример: представление чисел. Они сами по себе уже абстракции некоторых свойств объектов, которые следует как-то охарактеризовать. Если единственной (или доминирующей) операцией будет сложение, то хорошим способом представить число n будет просто последовательность из n «черточек». При таком представлении правило сложения действительно и очевидно, и просто. В основу «римского» представления положен тот же принцип простоты, и правила сложения для небольших чисел столь же просты. С другой стороны, «арабское» представление чисел требует далеко не очевидных правил (даже для небольших чисел), правила требуют запоминания, выучивания наизусть. Ситуация, однако, изменяется, если речь заходит о сложении больших чисел или введении умножения и деления. Разложение таких операций на более простые оказывается значительно проще в случае арабского представления, что объясняется систематичностью представления, основанного на «позиционном весе» цифр числа.

Хорошо известно, что в основе внутреннего представления чисел в вычислительных машинах лежат

двоичные цифры. Для человека такое представление не подходит: уж очень велико число таких цифр. Зато оно очень подходит для электронных схем: значения 0 и 1 можно удобно и надежно кодировать наличием или отсутствием электрического тока или магнитного поля.

На этом примере можно обнаружить, что вопросы представления часто разбиваются на несколько уровней детализации. Представьте себе, что необходимо задать положение некоторого объекта. Первое решение может привести к выбору пары вещественных чисел в прямоугольной или полярной системе координат. Второе решение — к представлению с плавающей запятой, где каждое вещественное число x состоит из пары целых чисел, обозначающих дробную часть f и показатель степени e некоторого основания (так, что $x = f * 2^e$). Третье решение — оно основывается на знании того, как в машине хранятся данные, — может привести к двоичному, позиционному представлению для целых чисел. И последнее, двоичные цифры можно кодировать направлением магнитного поля в «магнитном запоминающем устройстве». Очевидно, что первое из решений в этой цепочке зависит главным образом от постановки задачи, а последующие все больше и больше — от используемого инструмента и методов работы с ним. Так, вряд ли можно требовать, чтобы программист решал, какое следует использовать представление чисел или каковыми будут характеристики запоминающего устройства. Такие решения низших уровней можно оставлять создателям самих машин, они лучше других информированы о сегодняшней технологии, влияющей на выбор, который затем будет распространен на все (или почти все) приложения, связанные с использованием чисел.

В таком контексте становится очевидным значение *языков программирования*. Любому языку программирования соответствует некоторая абстрактная машина, способная интерпретировать понятия, используемые в этом языке. Это тоже некоторый уровень абстрагирования от реальных устройств, использованных в существующих машинах. Таким образом программист, использующий такой язык высокого уровня,

освобождается (и ограждается) от вопросов представления чисел, если числа в контексте этого языка — элементарные объекты.

Значение использования любого языка, представляющего подходящее множество основных абстракций, общих для большинства задач обработки данных, заключается главным образом в том, что это приводит к получению надежных программ.

Легче строить программы, основываясь на таких знакомых понятиях, как числа, множества, последовательности и повторения, вместо разрядов, ячеек и переходов. Конечно же, в реальной машине все данные, будь то числа, множества или последовательности, выглядят как огромные массивы разрядов. Но для программиста все это не имеет значения, поскольку он (или она) не беспокоится о деталях представления выбранной абстракции, ибо уверен, что на машине (или в трансляторе) выбрано наиболее уместное представление.

Чем ближе абстракция к конкретной машине, тем легче для инженера или реализатора языка выбрать подходящее представление, тем выше вероятность, что это единственное представление подойдет для всех (или почти всех) мыслимых приложений. Этот факт устанавливает определенные пределы на степень абстрагирования от заданной машины. Например, не имеет смысла включать в основные элементы универсального языка геометрические объекты, поскольку их удобное представление из-за присущей им внутренней сложности будет в значительной мере определяться теми операциями, которые над ними совершаются. Природа и частота этих операций создателю универсального языка и его транслятора неизвестны, и какое бы представление он ни выбрал, оно может оказаться неподходящим при некоторых применениях.

Именно эти соображения определяют выбор понятий для описания алгоритмов и их данных, использованных в нашей книге. Ясно, что нам хотелось использовать привычные математические понятия чисел, множеств, последовательностей и т. п., а не присущие машинам понятия вроде строк разрядов. Столь же очевидно, что мы хотели использовать и понятия,

для которых, и это уже известно, существуют эффективные трансляторы. Причем не хотелось бы как употреблять машинно-ориентированный или машинно-зависимый язык, так и описывать программы для машины в абстрактных понятиях, оставляющих открытыми вопросы представления. В попытках найти компромисс между этими крайними соображениями уже был создан язык Паскаль, и десять лет работы с ним [1.3] привели нас к языку Модула-2. В нем сохраняются основные понятия Паскаля и добавляются некоторые улучшения и расширения. Именно этим языком мы и будем пользоваться в нашей книге [1.5]. Язык Модула-2 уже был успешно реализован на нескольких машинах, и это подтвердило достаточную близость его понятий к реальным машинам и возможность легкого объяснения как самих понятий, так и их представления. Язык достаточно близок и к другим языкам, следовательно, уроки, из него извлеченные, применимы и к ним.

1.2. КОНЦЕПЦИЯ ТИПА ДАННЫХ

В математике принято классифицировать переменные в соответствии с некоторыми важными характеристиками. Мы различаем вещественные, комплексные и логические переменные, переменные, представляющие собой отдельные значения, множества значений или множества множеств; функции мы отличаем от функционалов или множеств функций и т. д. В обработке данных понятие классификации играет такую же, если не большую роль. Мы будем придерживаться того принципа, что любая константа, переменная, выражение или функция относятся к некоторому *типу*. Фактически тип характеризует множество значений, к которым относится константа, которые может принимать некоторая переменная или выражение и которые может формировать функция.

В математическом тексте тип переменной обычно определяется по шрифту, для этого нет нужды обращаться к контексту. Такой способ в программировании не подходит, поскольку обычно на машинах имеется оборудование лишь с одним шрифтом (например, латинскими буквами). Поэтому широко ис-

пользуется правило, по которому тип явно указывается в *описании* константы, переменной или любой функции. Это правило особенно важно потому, что транслятор должен выбирать представление данного объекта в памяти машины. Ясно, что память, отводимая под значение переменной, должна выбираться в соответствии с диапазоном значений, которые может принимать переменная. Если у транслятора есть такая информация, то можно обойтись без так называемого динамического распределения памяти. Часто это очень важно для эффективной реализации алгоритма. Основные принципы концепции типа, которым мы будем следовать на протяжении всего текста и которые включены в язык программирования Модула-2 [1.2], таковы:

1. Любой тип данных определяет множество значений, к которым может относиться некоторая константа, которое может принимать переменная или выражение и которое может формироваться операцией или функцией.

2. Тип любой величины, обозначаемой константой, переменной или выражением, может быть выведен по ее виду или по ее описанию; для этого нет необходимости проводить какие-либо вычисления.

3. Каждая операция или функция требует аргументов определенного типа и дает результат также фиксированного типа. Если операция допускает аргументы нескольких типов (например, «+» используется как для сложения вещественных чисел, так и для сложения целых), то тип результата регламентируется вполне определенными правилами языка.

В результате транслятор может использовать информацию о типах для проверки допустимости различных конструкций в программе. Например, неверное присваивание логического значения арифметической переменной можно выявить без выполнения программы. Такая избыточность текста программы крайне полезна и помогает при создании программ; она считается основным преимуществом хороших языков высокого уровня по сравнению с языком машины или ассемблера. Конечно, в конце концов данные будут представлены в виде огромного количества двоичных

цифр независимо от того, была ли программа написана на языке высокого уровня с использованием концепции типа или на языке ассемблера, где всякие типы отсутствуют. Для вычислительной машины память — это однородная масса разрядов, не имеющая какой-либо структуры. И только абстрактная структура позволяет программисту разобраться в этом однообразном пейзаже памяти машины.

Теория, представленная в нашей книге, и язык программирования Модула-2 предполагают некоторые методы определения типов данных. В большинстве случаев новые типы данных определяются с помощью ранее определенных типов данных. Значения такого нового типа обычно представляют собой совокупности значений компонент, относящихся к определенным ранее составляющим типам, такие значения называются *составными*. Если имеется только один составляющий тип, т. е. все компоненты относятся к одному типу, то он называется *базовым*. Число различных значений, входящих в тип T, называется мощностью T. Мощность задает размер памяти, необходимой для размещения переменной x типа T. Принадлежность к типу обозначается $x : T$.

Поскольку составляющие типы также могут быть составными, то можно построить целую иерархию структур, но конечные компоненты любой структуры, разумеется, должны быть атомарными. Следовательно, система понятий должна допускать введение и простых, элементарных типов. Самый прямолинейный метод описания простого типа — *перечисление* всех значений, относящихся к этому типу. Например, в программе, имеющей дело с плоскими геометрическими фигурами, можно описать простой тип с именем *фигура*, значения которого обозначаются идентификаторами: *прямоугольник*, *квадрат*, *эллипс*, *круг*. Но кроме типов, задаваемых программистом, нужно иметь некоторые стандартные, предопределенные типы. Сюда обычно входят числа и логические значения. Если для значений некоторого типа существует отношение порядка, то такой тип называется упорядоченным или *скалярным*. В Модуле-2 все элементарные типы упорядочены, в случае явного перечисления

считается, что значения упорядочены в порядке перечисления.

С помощью этих правил можно определять простые типы и строить из них составные типы любой степени сложности. Однако на практике недостаточно иметь только один универсальный метод объединения составляющих типов в составной тип. С учетом практических нужд представления и использования универсальный язык должен располагать несколькими *методами объединения*. Они могут быть эквивалентными в математическом смысле и различаться операциями для выбора компонент построенных значений. Основные рассматриваемые здесь методы позволяют строить следующие объекты: *массивы*, *записи*, *множества* и *последовательности*. Более сложные объединения обычно не описываются как «статические типы», а «динамически» создаются во время выполнения программы, причем их размер и вид могут изменяться. Такие объекты рассматриваются в гл. 4 — это списки, кольца, деревья и, вообще, конечные графы.

Переменные и типы данных вводятся в программу для того, чтобы их использовать в каких-либо вычислениях. Следовательно, нужно иметь еще и множество некоторых операций. Для каждого из стандартных типов в языке предусмотрено некоторое множество примитивных, стандартных операций, а для различных методов объединения — операции селектирования компонент и соответствующая нотация. Сутью искусства программирования обычно считается умение составлять операции. Однако мы увидим, что не менее важно умение составлять данные.

Важнейшие основные операции — *сравнение* и *присваивание*, т. е. проверка отношения равенства (и порядка в случае упорядоченных типов) и действие по «установке равенства». Принципиальное различие этих двух операций выражается и четким различием их обозначений в тексте:

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

Эти основные действия определены для большинства типов данных, но следует заметить, что для

данных, имеющих большой объем и сложную структуру, выполнение этих операций может сопровождаться довольно сложными вычислениями

Для стандартных простых типов данных мы кроме присваивания и сравнения предусматриваем некоторое множество операций, создающих (вычисляющих) новые значения. Так, для числовых типов вводятся стандартные арифметические операции, а для логических значений — элементарные операции логики высказываний.

1.3. ПРОСТЕЙШИЕ ТИПЫ ДАННЫХ

Любой новый простейший тип определяется простым перечислением входящих в него различных значений. Такой тип называется *перечисляемым*. Его определение имеет следующий вид:

$$\text{TYPE} \quad T = (c_1, c_2, \ldots, c_n) \tag{1.1}$$

где T — идентификатор нового типа, а c_i — идентификаторы новых констант. Мощность множества T обозначается через card (T) = n.

ПРИМЕРЫ

```
TYPE shape = (rectangle, square, ellipse, circle)
TYPE color = (red, yellow, green)
TYPE sex = (male, female)
TYPE BOOLEAN = (FALSE, TRUE)
TYPE weekday = (Monday, Tuesday, Wednesday, Thursday, Friday,
          Saturday, Sunday)
TYPE currency = (franc, mark, pound, dollar, shilling, lira, guilder,
          krone, ruble, cruzeiro, yen)
TYPE destination = (hell, purgatory, heaven)
TYPE vehicle = (train, bus, automobile, boat, airplane)
TYPE rank = (private, corporal, sergeant, lieutenant, captain, major,
       colonel, general)
TYPE object = (constant, type, variable, procedure, module)
TYPE structure = (array, record, set, sequence)
TYPE condition = (manual, unloaded, parity, skew)
```

Определение таких типов вводит не только идентификатор нового типа, но и одновременно множество идентификаторов, обозначающих значения этого но-

вого типа. Затем везде в программе их можно уже использовать как константы, что в значительной степени способствует наглядности. Если мы введем, скажем, переменные s, d, r и b

```
VAR s: sex
VAR d: weekday
VAR r: rank
VAR b: BOOLEAN
```

то можно написать такие операторы присваивания:

```
s := male
d := Sunday
r := major
b := TRUE
```

Очевидно, они более информативны, чем эквивалентные им:

```
s := 1   d := 7   r := 6   b := 2
```

где предполагается, что s, d, r и b определяются как целые числа, а константы в порядке их перечисления отображаются в натуральные числа. Кроме этого транслятор может проверить состоятельность использования той или иной операции. Например, при таком описании s, как выше оператор $s := s + 1$ должен считаться бессмысленным.

Поскольку перечисления задают упорядоченность, то перечислимые типы будут весьма чувствительными к введению операций, порождающих для заданного аргумента предыдущие или последующие значения. Поэтому мы просто постулируем стандартные операции, присваивающие своим аргументам следующее или предыдущее значение:

```
INC(x)    DEC(x)                                    (1.2)
```

1.4. ПРОСТЕЙШИЕ СТАНДАРТНЫЕ ТИПЫ

К стандартным простейшим типам мы относим типы, имеющиеся (встроенные) на большинстве вычислительных машин. Сюда входят целые числа, логические истинностные значения и множество печа-

таемых символов. На многих машинах есть и дробные числа вместе с соответствующими стандартными арифметическими операциями. Мы будем обозначать эти типы следующими идентификаторами:

INTEGER, CARDINAL, REAL, BOOLEAN, CHAR

Тип INTEGER включает некоторое подмножество целых, размер которого варьируется от машины к машине. Если для представления целых чисел в машине используется n разрядов, причем используется дополнительный код, то допустимые числа должны удовлетворять условию $-2^{n-1} \leqslant x < 2^{n-1}$. Считается, что все операции над данными этого типа выполняются точно и соответствуют обычным правилам арифметики. Если результат выходит за пределы представимого множества, то вычисления будут прерваны. Такое событие называется *переполнением*. Четыре основные арифметические операции считаются стандартными: сложение (+), вычитание (—), умножение (*) и деление (/, DIV).

Мы будем делать отличие между эйлеровской целой арифметикой и модульной; в первой деление обозначается косой чертой, а взятие остатка от деления — REM. Пусть частное q = m/n, а остаток r = = m REM n. Всегда справедливо отношение

$$q * n + r = m \quad \text{и} \quad 0 \leqslant \mathrm{ABS}(r) < \mathrm{ABS}(n) \tag{1.3}$$

Знак остатка тот же, что и у делителя (или же остаток равен нулю). Следовательно, эйлерово целое деление несимметрично относительно нуля и для него справедливы равенства

$$(-m)/n = m/(-n) = -(m/n)$$

ПРИМЕРЫ

31 / 10 = 3	31 REM 10 = 1
-31 / 10 = -3	-31 REM 10 = -1
31 / -10 = -3	31 REM -10 = 1
-31 / -10 = 3	-31 REM -10 = -1

В модульной (конгруэнтной) арифметике значение m MOD n фактически есть некоторый класс конгруэнт-

ности, т. е. множество целых, а не единственное число. В это множество входят все числа вида m — Qn для произвольных Q. Очевидно, что такие классы можно представлять одним специфическим элементом, например самым маленьким из неотрицательных элементов. Следовательно, мы определяем R = m MOD n и в то же время Q = m DIV n, так что удовлетворяются соотношения

$$Q * n + R = m \quad \text{и} \quad 0 \leqslant R < n \tag{1.4}$$

ПРИМЕРЫ

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

Обратите внимание, что деление на 10^n можно выполнять, сдвигая десятичное число вправо на n позиций, причем «выдвигаемые» цифры просто игнорируются. Этот же прием допустим, когда число представлено не в десятичном, а в двоичном виде. Если мы имеем дело с представлением в дополнительном коде (а на большинстве современных машин именно так и обстоит дело), то сдвиг реализует деление, определенное выше как операция DIV (а не операция /). Умеренно сложные трансляторы поэтому представляют операции вида m DIV 2^n с помощью быстрой операции сдвига, однако это упрощение к выражениям типа $m/2^n$ неприменимо.

Если про некоторую переменную известно, что она не принимает отрицательных значений (скажем, это какой-то счетчик), то это свойство можно подчеркнуть, сославшись на дополнительный стандартный тип CARDINAL. Если на машине для представления целых (без знака) используется n разрядов, то переменным такого типа можно присваивать величины, удовлетворяющие условию $0 \leqslant x < 2^n$.

Тип REAL обозначает подмножество вещественных чисел. В то время как считается, что арифметика с операндами типа INTEGER и CARDINAL дает точный результат, допускается, что аналогичные действия со значениями типа REAL могут быть неточными, в пределах ошибок округлений, вызванных вычислениями с конечным числом цифр. Это принциниальное различие, и оно привело в большинстве

языков программирования к явно различным типам INTEGER и REAL.

Два значения стандартного типа BOOLEAN обозначаются идентификаторами TRUE и FALSE. Булевские операции — это логическая конъюнкция, дизъюнкция и отрицание. Их значения (результаты) определены в табл. 1.1. Логическая конъюнкция обозначается через AND (или &), логическая дизъюнкция — OR, а отрицание — через NOT (или ~). Обращаем внимание, что операции сравнения дают

Таблица 1.1. Логические операции

p	q	p AND q	p OR q	NOT p
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

результат типа BOOLEAN. Таким образом, результат некоторого сравнения можно присвоить какой-то переменной или же использовать в качестве логического операнда 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)
```

породят результаты: p = FALSE и q = TRUE.

Логические операции AND и OR в Модуле-2 (и в некоторых других языках) обладают дополнительными свойствами, отличающими их от других бинарных операций. В то время когда, например, сумма x + y не определена, если x или y неизвестны, конъюнкция p&q определена даже при неизвестном q, если известно, что p имеет значение FALSE. Такая «условность» является важным и полезным свойством. Поэтому точное определение AND и OR дается такими двумя уравнениями:

```
p AND q = если p тогда q иначе FALSE
p OR q = если p тогда TRUE иначе q
```
(1.5)

В стандартный тип CHAR входит множество печатаемых символов. К несчастью, не существует та-

кого стандартного множества символов, которое было бы принято на всех вычислительных машинах. Поэтому использование определения «стандартное» во многих случаях может приводить к неверному пониманию: его следует воспринимать как: «стандартное для той вычислительной машины, на которой следует выполнить некоторую программу».

Наиболее широко распространено множество символов, принятое в качестве стандартного Международной организацией по стандартизации (ISO), и в частности, его американская версия ASCII (American Standard Code for Information Interchange). Поэтому в приложении 1 мы и приводим это множество. Оно состоит из 95 печатаемых (графических) символов и 33 управляющих, последние используются главным образом при передаче данных и для управления печатающими устройствами.

Чтобы иметь возможность составлять алгоритмы, манипулирующие символами (т. е. значениями типа CHAR) и независящие от системы, следует все же выделить некоторые из свойств, присущих множеству символов, а именно:

1. Тип CHAR содержит 26 прописных латинских букв и 26 строчных, 10 арабских цифр и некоторое число других графических символов, например знаки пунктуации.

2. Подмножества букв и цифр упорядочены и «соприкасаются», т. е.

$$\begin{aligned}&(\text{«A»} \leqslant x) \,\&\, (x \leqslant \text{«Z»}) \to x \text{ — прописная буква}\\ &(\text{«a»} \leqslant x) \,\&\, (x \leqslant \text{«z»}) \to x \text{ — строчная буква}\\ &(\text{«0»} \leqslant x) \,\&\, (x \leqslant \text{«9»}) \to x \text{ — цифра}\end{aligned} \qquad (1.6)$$

3. Тип CHAR содержит некоторый непечатаемый символ, пробел, его можно использовать как разделитель.

Рис. 1.1. Представление текста.

Когда заходит речь о написании программ в машинно-независимой форме, то особенно важно указать, что существуют две стандартные функции преобразования типов между CHAR и CARDINAL. Мы будем называть их ORD(ch) (дает порядковый номер ch во множестве символов) и CHR(i) (дает символ с порядковым номером i). Таким образом, CHR обратна по отношению к функции ORD, и наоборот, т. е.

$$\begin{aligned} &\mathrm{ORD}(\mathrm{CHR}(i)) = i \quad (\text{если } \mathrm{CHR}(i) \text{ определена}) \\ &\mathrm{CHR}(\mathrm{ORD}(c)) = c \end{aligned} \tag{1.7}$$

Кроме того, мы вводим стандартную функцию CAP(ch). Ее значение — прописная буква, соответствующая ch (если, конечно, ch вообще буква).

$$\begin{aligned} &ch \text{ — строчная буква} \rightarrow \mathrm{CAP}(ch) = \text{соответствующая прописная} \\ &ch \text{ — прописная буква} \rightarrow \mathrm{CAP}(ch) = ch \end{aligned} \tag{1.8}$$

1.5. ОГРАНИЧЕННЫЕ ТИПЫ (ДИАПАЗОНЫ)

Часто приходится сталкиваться с положением, когда переменной присваивается значение некоторого типа, лежащее только внутри определенного интервала значений. Такое положение можно подчеркнуть, определив, что указанная переменная относится к ограниченному типу (диапазону). Такой тип задается следующим образом:

$$\mathrm{TYPE}\ T = [min..max] \tag{1.9}$$

где min и max — выражения, определяющие концы такого диапазона. Отметим, что операндами этих выражений могут быть только константы.

ПРИМЕРЫ

```
TYPE year    = [1900 .. 1999]
TYPE letter  = ["A" .. "Z"]
TYPE digit   = ["0" .. "9"]
TYPE officer = [lieutenant .. general]
TYPE index   = [0 .. 2*N-1]
```

Если есть переменные

```
VAR y: year
VAR L: letter
```

то присваивания y := 1984 и L := «L» допускаются, а y := 1291 и L := «g» — нет. Однако в легальности подобных присваиваний можно удостовериться без выполнения программы лишь в тех случаях, когда речь идет о присваивании констант. Справедливость же присваивания y := i и L := c, где i — переменная целого типа, c — символьного, транслятор не может определить, если он только просматривает текст. Системы, ведущие такие проверки в процессе выполнения самих программ, оказываются очень ценными для их разработки. Использование трансляторами избыточной информации для выделения возможных ошибок еще раз объясняет стремление применять языки высокого уровня *).

1.6. МАССИВ

Вероятно, наиболее широко известная структура данных — массив. В некоторых языках только массивы и существуют. Массив состоит из компонент, причем все они одного типа, называемого *базовым*. Поэтому структура массивов *однородна*. Кроме того, массивы относят к так называемым структурам со *случайным доступом*. Для того чтобы обозначить отдельную компоненту, к имени всего массива добавляется *индекс*, он-то и выделяет нужную компоненту. Индекс — это значение специального типа, определенного как *тип индекса* данного массива. Поэтому определение массивового типа T фиксирует как базовый тип T0, так и тип индекса TI.

```
TYPE T = ARRAY[TI] OF T0                    (1.10)
```

*) Но можно рассуждать и другим образом. В начале в язык вводятся ограничения, которые невозможно проверять статически, а затем радуются, что удается построить системы, которые будут проверять их динамически. Так и в случае диапазонов: вводились-то они для экономии памяти (главным образом), а привели к архисложной динамической поддержке. — *Прим. перев.*

ПРИМЕРЫ

```
TYPE Row  = ARRAY [1..5] OF REAL
TYPE Card = ARRAY [1..80] OF CHAR
TYPE alfa = ARRAY [0..15] OF CHAR
```

Конкретное значение переменной

```
VAR x: Row
```

с компонентами, удовлетворяющими уравнению $x_i = 2^{-i}$, можно представить следующим образом (рис. 1.2).

С помощью индекса можно выделить любую отдельную компоненту любого массива. Если есть переменная-массив x, то селектор для массива обозначается с помощью имени соответствующего массива, за которым следует необходимый индекс i требуемой компоненты — x_i или x[i]. Из-за традиционности первого, обычного обозначения компоненты массивов стали называть *переменными с индексами.*

Обычный прием работы с массивами, в особенности с большими массивами, — выборочное изменение отдельных его компонент, а не конструирование полностью нового составного значения. При этом переменная-массив рассматривается как массив составляющих переменных и возможно присваивание отдельным компонентам, например, x[i] := 0.125. Хотя выборочное изменение приводит только к коррекции одной-единственной компоненты, с концептуальной точки зрения мы должны рассматривать его как изменение всего составного значения.

Тот факт, что индексы массива, т. е. имена его компонент, должны относиться к определенному (скалярному) типу, имеет весьма важное следствие: *индексы можно вычислять.* На место индексирующей константы можно подставлять любое индексирующее выражение; оно будет вычислено и результат идентифицирует требуемую компоненту. Такая общность не только

x_1	0.5
x_2	0.25
x_3	0.125
x_4	0.0625
x_5	0.03125

Рис. 1.2. Массив типа Row.

обеспечивает важное и весьма мощное средство программирования, но и провоцирует одну из наиболее часто встречающихся в программировании ошибок. Полученный результат может оказаться за пределами интервала, выделенного для индексов данного массива. Мы будем предполагать, что «порядочная» вычислительная система в случае ошибочного обращения к несуществующей компоненте массива должна давать некоторое предупреждающее сообщение.

Мощность массивового типа, т. е. число величин, принадлежащих этому типу, есть произведение мощностей его компонент. Так как все компоненты массивового типа T относятся к одному базовому типу T0, то при типе индекса TI получаем

$$\text{card}(T) = \text{card}(T0)^{\text{card}(TI)} \tag{1.11}$$

Составляющие массивов сами могут быть составными значениями. Переменная-массив, компоненты которой опять же массивы, называется *матрицей*. Например,

```
M: ARRAY[1..10] OF Row
```

это массив, состоящий из десяти компонент (строк), каждая из которых состоит из пяти компонент типа REAL, и называется матрицей размером 10×5 с вещественными составляющими. Соответственно можно соединять и селекторы, так что M_{ij} и M[i][j] обозначает j-ю компоненту строки M_i, а это i-я компонента массива M. Обычно можно пользоваться сокращением M[i, j], и в том же духе описание

```
M: ARRAY[1..10] OF ARRAY[1..5] OF REAL
```

можно записывать более компактно:

```
M: ARRAY[1..10],[1..5] OF REAL
```

Если необходимо выполнить некоторую операцию надо всеми компонентами массива или над соседними компонентами некоторой секции массива, то для этого удобно воспользоваться оператором цикла. В приведенном примере вычисляется сумма элементов и отыскивается максимальный элемент массива, описан-

ного следующим образом:

```
VAR a: ARRAY [0 .. N-1] OF INTEGER

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 дробная часть f представляется в десятичном виде с помощью k — 1 цифры, т. е. таким массивом d, что

$$f = Si:\ 1 \leqslant i < k:\ d_i * 10^{-i}$$

Предположим, что мы хотим разделить f на 2. Это проделывается повторением знакомой операции деления для всех k — 1 цифр d_i, начиная с i = 1. Операция включает деление каждой цифры на 2 с учетом возможного «переноса» из предыдущей позиции и переноса возможного остатка r в следующую позицию

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

В программе 1.1 эта процедура используется для построения таблицы отрицательных степеней двойки. Повторяющееся деление для вычисления 2^{-1}, 2^{-2}, ...

```
MODULE Power;
  (*вычисление десятичного представления степеней 2*)
  FROM InOut IMPORT Write, WriteLn;
  CONST N = 10;
  VAR i, k, r: CARDINAL;
    d: ARRAY [1 .. N] OF CARDINAL;
BEGIN
  FOR k := 1 TO N DO
    Write("."); r := 0;
    FOR i := 1 TO k-1 DO
      r := 10*r + d[i]; d[i] := r DIV 2; r := r MOD 2;
      Write(CHR(d[i] + ORD("0")))
    END;
    d[k] := 5; Write("5"); WriteLn
  END
END Power.
```

Прогр. 1.1. Вычисление степеней двойки.

..., 2^N вновь реализуется с помощью оператора цикла со словом FOR, что приводит к появлению двух вложенных операторов цикла.

Результирующая выдача для n = 10 выглядит так:

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

1.7. ЗАПИСЬ

Наиболее общий метод получения составных типов заключается в объединении элементов произвольных типов. Причем сами эти элементы могут быть в свою очередь составными. Приведем примеры из математики: комплексные числа, состоящие из двух вещественных чисел, или координаты точки, состоящие из двух или более чисел, — это зависит от размерности пространства, накрываемого данной системой координат. А вот пример из обработки данных: человек описывается с помощью нескольких подходящих характеристик вроде имени, фамилии, даты рождения, пола и семейного положения.

В математике такие составные типы называют *декартовым произведением* составляющих его типов. Это объясняется тем, что множество значений, определенное таким составным типом, состоит из всех возможных комбинаций значений, относящихся к каждому из множеств, представляющих собой составляющие типы. Таким образом, число таких комбинаций — их иногда называют n-ками — равно произведению числа элементов в каждом из составляющих множеств. Поэтому мощность составного типа есть произведение мощностей составляющих типов.

При обработке данных составные типы, описывающие людей или объекты, обычно встречаются в файлах или банках данных; они описывают существенные характеристики некоторых персон или объектов. Поэтому к данным такой природы стало широко применяться слово *запись* (record), и именно им мы будем пользоваться вместо термина «декартово произведение». В общем случае записной тип T с компонентами T_1, T_2, ..., T_n определяется следующим образом:

```
TYPE T =      RECORD s1 : T1;
                     s2 : T2;
                     ...                                    (1.12)
                     sn : Tn
              END
```

$$card(T) = card(T_1) * card(T_2) * ... * card(T_n)$$

ПРИМЕРЫ

```
TYPE Complex = RECORD re: REAL;
                      im: REAL
               END

TYPE Date =    RECORD day: [1..31];
                      month: [1..12];
                      year: [1..2000]
               END

TYPE Person =  RECORD name, firstname: alfa;
                      birthdate: Date;
                      sex: (male, female);
                      marstatus: (single, married, widowed, divorced)
               END
```

Величины записного типа можно представлять графически; например, на рис. 1.3 приведены переменные

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

Идентификаторы s_1, s_2, ..., s_n, появляющиеся в определении записного типа, представляют собой имена, даваемые отдельным компонентам переменных данного типа. Поскольку компоненты записи называются *полями*, то эти имена называются *идентифи-*

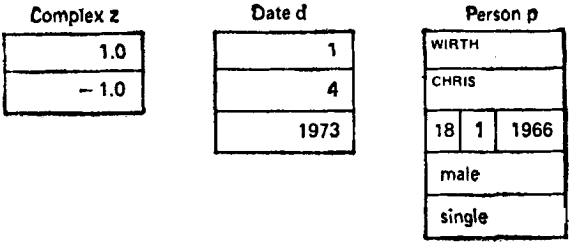


Рис. 1.3. Записи типов Complex, Date и Person.

каторами полей. При работе с записями они используются в селекторах, применяемых к переменным-записям. Если есть переменная x: T, то ее i-е поле обозначается через $x.s_i$. Выборочное изменение x осуществляется путем указания нужного селектора в левой части оператора присваивания:

x.si : = e

где e — величина (выражение) типа T_i. Возьмем, например, переменные-записи z, d и p, описанные ранее. Селекторы их составляющих выглядят так:

```
z.im              (типа REAL)
d.month           (типа [1..12])
p.name            (типа alfa)
p.birthdate       (типа Date)
p.birthdate.day   (типа [1..31])
```

Пример типа Person показывает, что составляющие записного типа сами могут быть составными. Поэтому селекторы могут соединяться один с другим. Естественно, что вкладывать один в другой можно и различные записные типы. Например, i-я компонента массива a, входящего в состав записи r, обозначается r.a[i], а компонента с селектирующим именем s в i-й компоненте-записи массива a обозначается через a[i].s.

Для декартова произведения характерно, что в него входят все комбинации элементов составляющих типов. Однако в отдельных приложениях не все они могут быть осмысленными. Скажем, тип Date, определенный выше, включает 31 апреля и 29 февраля

1985, которые, конечно же, не существуют. Так что определение такого типа не отражает совершенно точно фактическую ситуацию, хотя оно и достаточно удобно для практических целей. Поэтому ответственность за то, что бессмысленные величины никогда не встретятся в процессе исполнения программы, возлагается на программиста.

В следующем небольшом фрагменте из программы показано, как же используются записи. Нужно подсчитать число «персон» из массива *family*, относящихся к женскому полу и одиноких.

```
VAR count: CARDINAL;
  family: ARRAY [1 .. N] OF Person;
  count := 0;

FOR i := 1 TO N DO
  IF (family[i].sex = female) & (family[i].marstatus = single) THEN
    count := count + 1
  END
END
```

Можно написать и другой вариант этого оператора с конструкцией, которую называют *оператором присоединения*:

```
FOR i := 1 TO N DO
  WITH family[i] DO
    IF (sex = female) & (marstatus = single) THEN
      count := count + 1
    END
  END
END
```

Конструкция «WITH r DO s END» означает, что внутри оператора s идентификаторы полей, относящиеся к переменной r, можно использовать без префикса. Считается, что все они относятся к переменной r. Таким образом, оператор присоединения позволяет сократить сам текст программы и, кроме этого, экономит часто повторяющиеся вычисления адреса индексированной компоненты family[i].

И записи, и массивы обладают одним общим свойством — случайным доступом к компонентам. Записи более универсальны в том смысле, что для них не требуется идентичности всех составляющих типов.

В то же время массивы обеспечивают большую гибкость — селекторы компонент можно вычислять (это выражения) в отличие от селекторов для компонент записи, которые описываются в определении записного типа.

1.8. ЗАПИСИ С ВАРИАНТАМИ

На практике часто бывает удобно и естественно рассматривать два типа просто как варианты некоторого одного. Например, тип *Coordinate* из предыдущего раздела можно считать объединением его двух вариантов — прямоугольных и полярных координат, а их составляющими соответственно (а) две длины и (б) длину и угол. Для того чтобы указать, к какому варианту относится та или иная переменная, вводится третья компонента — *дискриминант типа* или *поле признака.*

```
TYPE CoordMode = (Cartesian, polar);
TYPE Coordinate =
        RECORD
          CASE kind: CoordMode OF
           Cartesian:  x, y: REAL |
           polar:      r: REAL; phi: REAL
          END
        END
```

Здесь имя поля признака — *kind*, а имена координат либо, в случае прямоугольных, x и y, либо, в случае полярных, r и phi. Множество значений, определяемое этим типом Coordinate, есть объединение двух указанных типов:

T_1 = (x, y: REAL)
T_2 = (r: REAL; phi REAL)

Его мощность — просто сумма мощностей T_1 и T_2:

$$\text{card}(T) = \text{card}(T_1) + \text{card}(T_2) \qquad (1.13)$$

Чаще, однако, варианты бывают не полностью разными типами, а скорее типами с частично совпадающими компонентами. Распространенность этой ситуации и породила понятие *вариантов* записи. Возьмем

в качестве примера тип *Person*, определенный в предыдущем разделе. Характеристики, которые нужно записывать в дело, зависят от пола конкретной персоны. Для мужчин, скажем, существенными в некотором смысле можно считать вес и наличие бороды, а для женщины определяющими — три ее размера. В следующем определении типа эти соображения учтены.

```
TYPE Person =
  RECORD name, firstname: alfa;
    birthdate: Date;
    marstatus: (single, married, widowed, divorced);
    CASE s: sex OF
     male:   weight: REAL; bearded: BOOLEAN |
     female: size: ARRAY [1 .. 3] OF INTEGER
    END
  END
```

Общий вид определенного записного типа с вариантами таков:

```
TYPE T =                                                    (1.14)
  RECORD s1: T1; s2: T2; ... ; sn-1: Tn-1;
    CASE sn: Tn OF
      v1: s11: T11; ... ; s1,n1: T1,n1 |
      v2: s21: T21; ... ; s2,n2: T2,n2 |
      ...
      vm: sm,1: Tm,1; ... ; sm,nm: Tm,nm
    END
  END
```

Идентификаторы s_i (для $i = 1 \ldots n - 1$) — это имена полей, относящихся к общей части записи, а s_{ij} — селектирующие имена полей, относящихся к *вариантной части*, идентификатор же s_n — имя дискриминантного *поля признака* типа T_n. Константы $v_1, v_2, \ldots, v_m$ обозначают значения (скалярные) поля признака типа. Каждый вариант i имеет в своей вариантной части n_i полей, Любая переменная x такого типа T состоит из компонент

$$x.s_1,\ x.s_2,\ \ldots,\ x.s_n,\ x.s_{k,1},\ \ldots,\ x.s_{k,n_k}$$

только в случае, если (и только если) текущее значение $x.s_n = v_k$. Поэтому использование компоненты с селектором $x.s_{k,h} (1 \leqslant h \leqslant n_k)$ при $x.s_n \neq v_k$ должно рассматриваться как серьезная ошибка программирования. Если говорить о типе Person, упомянутом выше, то это будет аналогично вопросу, имеет ли женщина бороду, а в случае изменения данных ей можно даже ее приделать. При использовании вариантных записей поэтому необходима крайняя внимательность, и лучше всего собирать соответствующие операции над определенными вариантами в один селектирующий оператор, так называемый *оператор варианта*. Его структура весьма похожа на структуру определения записного типа.

```
CASE x.s_n OF
    v_1 : S_1;
    v_2 : S_2;
    ...                                    (1.15)
    v_m : S_m
END
```

S_k — оператор, предназначенный для случая, когда считается, что x имеет вид варианта k. И он выбирается для исполнения только в том случае, если поле признака $x.s_n$ имеет значение v_k. Поэтому очень легко защититься от неверного использования селектирующих имен, проверив, что каждый S_k содержит только селекторы

$$x.s_1 \ldots x.s_{n-1} \text{ и } x.s_{k,1} \ldots x.s_{k,n_k}$$

Следующая простая и короткая программа иллюстрирует работу оператора варианта. Необходимо

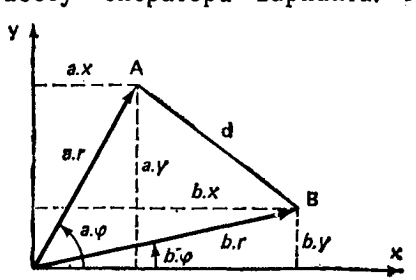


Рис. 1.4. Прямоугольные и полярные координаты.

вычислить расстояние d между двумя точками A и B, представленными переменными a и b, относящимися к типу Coordinate (запись с вариантами). В соответствии с четырьмя вариантами комбинаций прямоугольных и полярных координат есть четыре различные процедуры вычислений (см. рис. 1.4).

```
CASE a.kind OF
 Cartesian: CASE b.kind OF
             Cartesian: d := sqrt(sqr(a.x-b.x) + sqr(a.y-b.y)) |
             Polar:     d := sqrt(sqr(a.x - b.r*cos(b.phi) + sqr(a.y - b.r*sin(b.phi))
            END |
 Polar:     CASE b.kind OF
             Cartesian: d := sqrt(sqr(a.r*cos(a.phi) - b.x) + sqr(a.r*sin(a.phi) - b.y)) |
             Polar:     d := sqrt(sqr(a.r + sqr(b.r) - 2*a.r*cos(a.phi-b.phi))
            END
END
```

1.9. МНОЖЕСТВА

Третьим фундаментальным классом данных будут данные, построенные как *множества*. Соответствующий тип определяется так:

$$\text{TYPE } T = \text{SET OF } T_0 \tag{1.16}$$

Возможными значениями некоторой переменной x типа T будут множества элементов типа T_0. Множество всех подмножеств из элементов множества T_0 называется *множеством-степенью* T_0. Таким образом, тип T включает множество-степень его базового типа T_0.

ПРИМЕРЫ

```
TYPE BITSET = SET OF [0..15]
TYPE TapeStatus = SET OF exception (см. 1.3)
```

Если есть переменные

```
b: BITSET
t: ARRAY [1 .. 6] OF TapeStatus
```

то значения, имеющие вид множеств, можно «конструировать» и присваивать, например, таким обра-

зом:

```
t0 := {2, 3, 5, 7, 11, 13}
t[3] := TapeStatus{manual}
t[5] := TapeStatus{}
t[6] := TapeStatus{unloaded .. skew}
```

Здесь величина, присваиваемая t_3, называется *единичным* множеством (синглетоном), состоящим из единственного элемента *manual*; переменной t_5 присваивается пустое множество, означающее, что пятая лента переходит в рабочее состояние (с ней нет никаких неприятностей), t_6 же присваивается множество из всех трех «инцидентов».

Мощность множественного типа T равна

$$\mathrm{card}(T) = 2^{\mathrm{card}(T_0)}$$

В этом легко убедиться, если учесть, что для каждого из $\mathrm{card}(T_0)$ элементов T_0 нужно отмечать, находится он в множестве или нет, причем все элементы взаимно независимы. Совершенно очевидно, что для эффективной и экономной реализации необходима не только конечность базового типа, нужно, чтобы мощность его была осмысленно маленькой.

Для всех множественных типов определены такие элементарные операции:

*	пересечение множеств
+	объединение множеств
—	вычитание множеств
IN	проверка на вхождение

Пересечение и *объединение* двух множеств часто называются соответственно *умножением* и *сложением* множеств. Отсюда и приоритеты этих операций: пересечение старше операций объединения и вычитания, а они в свою очередь старше проверки на вхождение, которая относится к операциям отношения. Ниже приводятся примеры выражений для множеств и их эквиваленты с полностью расставленными скобками:

```
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)
```

1.10. ПРЕДСТАВЛЕНИЕ МАССИВОВ, ЗАПИСЕЙ И МНОЖЕСТВ

Смысл использования в программировании абстрактных понятий заключается в том, что программу можно построить, понять и проверить, основываясь на законах, управляющих именно этими понятиями. Причем нет необходимости «заглядывать внутрь» и знать, как реализуются и представляются в конкретной машине эти понятия. Тем не менее для профессионального программиста важно понимание общераспространенных методов представления таких основных абстрактных понятий программирования, какими являются данные. Нет ничего плохого в том, что программисту будет дозволено принимать решения, касающиеся строения данных и самой программы, исходя не только из абстрактных свойств их структур, но и ориентируясь на их реализацию в реальной машине, учитывая ее конкретные возможности и ограничения.

Проблема представления данных заключается в отображении абстрактных структур на память машины. В первом приближении эта память — массив отдельных ячеек, называемых *словами*. Индексы же этих слов называются *адресами*

```
VAR store: ARRAY ADDRESS OF WORD
```

Мощность типов ADDRESS и WORD от машины к машине изменяется. Особой проблемой остается большое разнообразие мощности типа WORD. Его логарифм называется размером слова, поскольку он определяет, из скольких разрядов состоит одно слово.

1.10.1. Представление массивов

Любое представление структуры массива заключается в отображении массива (абстрактного) с компонентами типа T на память, которая представляет собой массив с компонентами типа WORD. Массив следует отображать так, чтобы вычисление адреса его компоненты было как можно проще (и поэтому эффективно). Адрес i для j-й компоненты массива вы-

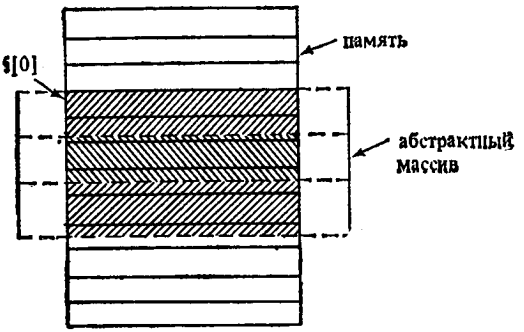


Рис. 1.5. Отображение массива на память.

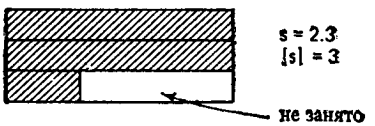


Рис. 1.6. Представление записи с выравниванием.

числяется с помощью линейной функции отображения:

$$i = i_0 + j * s \tag{1.18}$$

где i_0 — адрес первой компоненты, а s — число слов, занимаемых одной компонентой. Если предположить, что слово представляет собой самую маленькую, отдельно адресуемую единицу памяти, то станет очевидным желание, чтобы s было целым числом, а проще всего, чтобы $s = 1$. Если s не целое число (а это самая обычная ситуация), то, как правило, его округляют до ближайшего целого числа, превышающего его — $\lceil s \rceil$. Теперь каждая компонента будет занимать $\lceil s \rceil$ слов, а $\lceil s \rceil - s$ слов останутся неиспользованными (см. рис. 1.5 и 1.6). Округление необходимого числа слов до следующего целого числа называется *выравниванием* (padding). Коэффициентом использования памяти и называется частное от деления минимального размера памяти, необходимой для представления некоторых данных, на размер фактически занятой памяти:

$$u = s/\lceil s \rceil \tag{1.19}$$

Так как цель любого реализатора — как можно лучше использовать память, т. е. стремиться к $u = 1$, а доступ к части слова — неудобный и относительно неэффективный процесс, то приходится идти на компромисс. Надо учитывать следующие соображения:

1. Выравнивание уменьшает используемую память.

2. Отказ от выравнивания может привести к необходимости использования доступа к части слова.

3. Организация доступа к части слова может настолько увеличить размер оттранслированной программы, что выигрыша из-за отказа от выравнивания не будет.

Фактически соображения 2 и 3 настолько доминируют, что в трансляторах выравнивание всегда используется. Обращаем внимание, что при $s > 0.5$ коэффициент использования всегда больше 0.5. Если же, однако, $s \leqslant 0.5$, то коэффициент использования можно значительно увеличить, если в каждое слово укладывать более одной компоненты массива. Такой метод называется *упаковкой* (packing). При упаковке в слово n компонент коэффициент использования имеет вид (см. рис. 1.7)

$$u = n * s / \lceil n * s \rceil \qquad (1.20)$$

Доступ к i-й компоненте упакованного массива включает вычисление j — адреса слова, в котором размещается нужная компонента, и вычисление k — относительного положения компоненты внутри этого слова:

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

В большинстве языков программирования программист не имеет возможности управлять представлением абстрактных данных. Однако ему следовало бы предоставить возможность указывать на желательность упаковки по крайней мере в тех случаях, когда

$$u = \frac{n \cdot s}{\lceil n \cdot s \rceil}$$

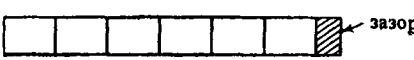


Рис. 1.7. Упаковка в одном слове шести компонент.

в одном слове можно разместить более одной компоненты, т. е. когда можно добиться коэффициента экономии памяти, равного двум или более. Мы будем считать, что в случае желательности упаковки перед словом ARRAY (или RECORD) в определении типа ставится слово PACKED.

1.10.2. Представление записей

Записи отображаются в память вычислительной машины простым объединением отображений их компонент. Адрес компоненты (поля) r_i относительно начального адреса записи r называется *смещением*. Он вычисляется так:

$$k_i = s_1 + s_2 + \ldots + s_{i-1} \qquad (1.21)$$

где s_j — размер (в словах) j-й компоненты. Теперь воспользуемся тем, что все компоненты массива — одного типа, и отсюда получим $k_i = (i-1)*s$. К несчастью, общность структуры записи не позволяет воспользоваться такой простой линейной функцией для вычисления смещения. Именно поэтому кажется весьма осмысленным требование, чтобы обращение к компонентам записи проходило по фиксированным идентификаторам. Такое ограничение дает выигрыш, поскольку относительные смещения становятся известными в момент трансляции, что приводит к улучшению эффективности доступа к полям записи. Этот факт общеизвестен.

Метод упаковки будет давать выигрыш, если в одно-единственное слово можно укладывать несколько компонент записи (см. рис. 1.8). Так как транслятор

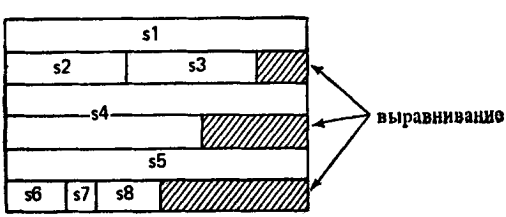


Рис. 1.8. Представление упакованной записи.

может вычислять смещения, то он может определять и смещения полей внутри слова. Это означает, что на многих машинах доступ к упакованным записям оказывается даже эффективнее доступа к упакованным массивам.

1.10.3. Представление множеств

Множество s удобно представлять в памяти машины с помощью его *характеристической функции* C(s). Это массив логических значений, причем i-я составляющая указывает, что i находится в s. Размер массива определяется мощностью множественного типа:

$$C(s_i) = (i \text{ IN } s) \tag{1.22}$$

Например, множество небольших целых s = {2, 3, 5, 7, 11, 13} представляется последовательностью логических значений F (ложь) и T (истина)

C(s) = (FFTTFTFTFFFTFTFF)

при условии, что базовый тип s — диапазон 0..15. В машине последовательность логических значений представляется последовательностью разрядов, т. е. последовательностью нулей и единиц.

Представление множеств с помощью их характеристических функций имеет то преимущество, что операции вычисления объединения, пересечения и разности двух множеств могут реализовываться элементарными логическими операциями. Ниже приводятся эквивалентные соотношения, справедливые для всех элементов i базового типа множеств x и y. Они связывают логические операции с операциями над множествами:

$$\begin{aligned} i \text{ IN } (x+y) &= (i \text{ IN } x) \text{ OR } (i \text{ IN } y) \\ i \text{ IN } (x*y) &= (i \text{ IN } x) \text{ AND } (i \text{ IN } y) \\ i \text{ IN } (x-y) &= (i \text{ IN } x) \text{ AND NOT } (i \text{ IN } y) \end{aligned} \tag{1.23}$$

Такие логические операции существуют во всех цифровых вычислительных машинах, более того, они выполняются над всеми соответствующими элементами (разрядами) некоторого слова одновременно.

Поэтому, чтобы иметь возможность эффективно реализовывать основные операции над множествами, их следует представлять небольшим и фиксированным числом слов, для которых доступны не только основные логические операции, но и операции сдвига. В этом случае проверка на присутствие элемента реализуется одним-единственным сдвигом и последующей операцией проверки знакового разряда. Отсюда следует, что проверка, записанная в виде $x \text{ IN } \{c_1, c_2, \ldots, c_n\}$, может реализовываться значительно эффективнее, чем эквивалентное ей логическое выражение:

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

Заключение же такое: множествами следует пользоваться только в случае «небольших» базовых типов. Предел мощности этих базовых типов, гарантирующих относительно эффективную реализацию, зависит от размера слова конкретной машины. И ясно, что в этом отношении предпочтение отдается машинам с большим размером слова. Если же размер слова относительно невелик, то можно основываться на представлении множеств посредством нескольких слов.

1.11. ПОСЛЕДОВАТЕЛЬНОСТИ

Четвертым элементарным классом данных является *последовательность*. Последовательностный тип можно было бы описать следующим образом:

$$\text{TYPE } T = \text{SEQUENCE OF } T_0 \qquad (1.24)$$

Уже из описания ясно, что все элементы последовательности имеют один и тот же тип. Последовательность s из n элементов мы будем обозначать $s = \langle s_0, s_1, s_2, \ldots, s_{n-1} \rangle$, причем n называется *длиной* последовательности. Структура последовательности очень похожа на структуру массива. Существенное же различие заключается в том, что у массива число элементов фиксируется в его описании, а у последовательности оно остается открытым. И предполагается, что в процессе работы программы это число может меняться. Хотя каждая последовательность в любой заданный момент времени имеет вполне определенную

конечную длину, мы должны считать мощность последовательностного типа бесконечной, поскольку на потенциальную длину переменных-последовательностей нет никаких ограничений.

Прямое следствие бесконечности мощности последовательностного типа — невозможность выделить для соответствующей переменной память заданного размера. Вместо этого мы должны выделять память в процессе выполнения программы по мере роста последовательности. Если же последовательность уменьшается, то память можно и возвращать. В любом случае следует пользоваться некой схемой динамического распределения памяти. Это свойство присуще всем типам бесконечной мощности, причем оно (свойство) настолько существенно, что мы будем говорить о типах высшего порядка по сравнению с фундаментальными типами, обсуждавшимися до этого момента.

Что же, однако, заставляет нас заняться рассмотрением последовательностей в главе, посвященной фундаментальным типам? Первая причина — стратегия управления памятью для последовательностей, если оговорить некоторую дисциплину их использования, достаточно проста (по сравнению с другими динамическими структурами). Фактически при соблюдении этих условий работа с памятью может быть достаточно быстро сведена к механизмам, гарантирующим приемлемую эффективность. Вторая же причина заключается в том, что последовательности по существу присутствуют во всех приложениях вычислительных машин, они как бы вездесущи. Данные такой структуры превалируют во всех тех случаях, когда идет работа с памятями разного вида, т. е. когда данные передаются из внешней памяти, скажем дисков или лент, в оперативную, главную память и обратно.

Упомянутая нами дисциплина требует ограничиваться только *последовательным доступом*. Под этим мы подразумеваем, что последовательность просматривается от одного элемента строго к его непосредственному преемнику, формируется же она последовательным добавлением элементов в ее конец. Отсюда немедленно следует, что прямого (непосредственного) доступа к элементам нет, исключением является

лишь элемент, с которым мы в данный момент имеем дело. Как мы убедимся (см. гл. 2), дисциплина доступа оказывает огромное влияние на саму программу.

Преимущество такой приверженности последовательному доступу, который, как бы то ни было, представляет собой серьезное ограничение, заключается в относительной простоте требуемого механизма управления памятью. Но еще более важной, если речь идет об обменах данными со вторичной памятью, выглядит возможность пользоваться эффективной техникой буферизации. Последовательный доступ позволяет нам для «перекачки» данных между памятями различного вида использовать непрерывные потоки данных. Буферизация предполагает, что части потока накапливаются в так называемых *буферах*, а затем, уже при заполнении всего буфера, передаются куда нужно. В результате, что особенно важно, более эффективно используется вторичная память. Если говорить только о последовательном доступе, то механизм буферизации достаточно хорош для любых последовательностей и любых памятей. Поэтому, как правило, этот метод включают в любую универсальную систему и программисту нет нужды заботиться о включении его в свою программу. Обычно соответствующие системы носят название файловых систем, поскольку для постоянного хранения используются очень емкие устройства последовательного доступа, и они продолжают хранить информацию даже при отключении машины. Единица данных на таких устройствах — последовательность, или *последовательный файл.*

Есть несколько типов памяти, на которых можно пользоваться только последовательным доступом. Среди них, очевидно, все типы лент. Однако даже на магнитном диске каждая дорожка с записью информации представляет собой память, допускающую только последовательный доступ. Строго последовательный доступ есть характерное свойство любого устройства, основанного на механическом движении, а также и некоторых других устройств.

Теперь суммируем все вышесказанное.

1. Массивы, записи и множества — это данные, структура которых допускает случайный доступ. Ими пользуются в тех случаях, когда их можно разместить в основной памяти со случайным доступом.

2. Для работы с вторичной памятью, для которой характерен последовательный доступ, скажем для дисков или лент, используются последовательности.

1.11.1. Элементарные операции с последовательностями

Дисциплину последовательного доступа можно стимулировать, предусмотрев исключительно для переменных-последовательностей множество соответствующих операций. Таким образом, хотя вроде и можно было бы ссылаться на i-й элемент последовательности через s_i, но в программе написать такое обращение нельзя. В множество операций, очевидно, должны входить операции для формирования и контроля последовательностей. Как уже упоминалось, любая последовательность порождается путем добавления в конец очередного элемента, а при контроле происходит продвижение к следующему элементу. Следовательно, с каждой последовательностью всегда ассоциируется некоторая *позиция*. Теперь мы постулируем и неформально опишем такие примитивные операции:

1. Open(s) определяет s как пустую последовательность, т. е. последовательность длины 0.

2. Write(s, x) добавляет элемент со значением x в последовательность s.

3. Reset(s) устанавливает текущую позицию в начало s.

4. Read(s, x) присваивает элемент из текущей позиции переменной x, позиция же перемещается к следующему элементу.

Для того чтобы более точно охарактеризовать операции над последовательностями, приведем следующий пример. Мы покажем, как его *можно было бы* выразить в терминах массивов. В начале реализация строится на введенных ранее и уже обсуждавшихся концепциях, здесь нет ни буферизации, ни последовательной памяти, которые, как мы уже упоминали, делают последовательности по-настоящему привлека-

тельным понятием. Тем не менее наш пример наглядно демонстрирует характерные особенности примитивных операций над последовательностями, если они поддерживаются системой последовательных файлов во вторичной памяти.

Первое — операции описываются нами в терминах обычных процедур. Это — собрание определений, которое называется *определяющим модулем* (definition module). Так как тип нашей последовательности фигурирует в списке формальных параметров, то этот тип также должен быть описан. Его описание — великолепный пример применения структуры класса запись, поскольку кроме поля, обозначающего массив (он моделирует последовательность), необходимы дополнительные поля, обозначающие текущую длину и позицию, т. е. описывающие *состояние* последовательности. Кроме этих полей мы введем еще одно, оно будет указывать, верно ли выполнилась операция чтения или же нет, из-за того, скажем, что не оказалось очередного элемента.

```
DEFINITION MODULE FileSystem;                                    (1.25)
 FROM SYSTEM IMPORT WORD;

 CONST MaxLength = 4096;
 TYPE Sequence = RECORD pos, length: CARDINAL;
                   eof: BOOLEAN;
                   a: ARRAY [0 .. MaxLength-1] OF WORD
                 END;

 PROCEDURE Open(VAR f: Sequence);
 PROCEDURE WriteWord(VAR f: Sequence; w: WORD);
 PROCEDURE Reset(VAR f: Sequence);
 PROCEDURE ReadWord(VAR f: Sequence; VAR w: WORD);
 PROCEDURE Close(VAR f: Sequence);
END FileSystem.
```

Обратите внимание, что в этом примере максимальная достижимая длина последовательности — произвольная константа. Если в какой-либо программе случится, что последовательность станет длиннее, то это будет рассматриваться не как ошибка в программе, а скорее как неадекватная реализация. С другой стороны, операция чтения за фактическим текущим концом последовательности действительно должна считаться ошибкой в программе. Дабы избежать

такой ситуации, следует предусматривать возможность проверки: не достигнут ли конец? Для этого есть логическое поле eof (от *end of* — конец чего-либо). Предполагается, что пользователь такой реализации абстрактного типа последовательность знает только это поле. В качестве базового типа здесь выбран тип WORD. Как правило, мы его будем использовать в любых случаях. В языке Модула-2 именно этот тип совместим по параметрам с несколькими стандартными типами, включая INTEGER.

Операторы, составляющие сами процедуры, определяются в следующем реализационном модуле:

```
IMPLEMENTATION MODULE FileSystem;
FROM SYSTEM IMPORT WORD;                                    (1.26)

PROCEDURE Open(VAR f: Sequence);
BEGIN f.length := 0; f.pos := 0; f.eof := FALSE
END Open;

PROCEDURE WriteWord(VAR f: Sequence; w: WORD);
BEGIN
  WITH f DO
    IF pos < MaxLength THEN
      a[pos] := w; pos := pos+1; length := pos
    ELSE HALT
    END
  END
END WriteWord;

PROCEDURE Reset(VAR f: Sequence);
BEGIN f.pos := 0; f.eof := FALSE
END Reset;

PROCEDURE ReadWord(VAR f: Sequence; VAR w: WORD);
BEGIN
  WITH f DO
    IF pos = length THEN f.eof := TRUE
      ELSE w := a[pos]; pos := pos+1
    END
  END
END ReadWord;

PROCEDURE Close(VAR f: Sequence);
BEGIN (*пусто*)
END Close;
END FileSystem.
```

При наших операциях над множествами можно пользоваться для записи и последующего чтения

последовательного файла такими схемами программ:

```
Open(s);
WHILE B DO P(x); WriteWord(s, x) END                    (1.27)

Reset(s); ReadWord(s, x);
WHILE ~s.eof DO Q(x); ReadWord(s, x) END
```

Здесь B есть некоторое условие, оно должно выполниться, прежде чем выполнится P и вычислит значение очередного элемента. При чтении к каждому значению, прочитанному из последовательности, применяется оператор Q. Он «охраняется» условием $\sim s.eof$, гарантирующим, что очередной элемент действительно прочитан. Дополнительный оператор чтения обусловлен тем фактом, что для прочтения n элементов нужно выполнить $n + 1$ операций чтения, причем последняя будет неудачной. Это неудачное чтение совершенно необходимо, ибо только оператор чтения изменяет значение переменной *eof*. Его можно было бы избежать, если постулировать, что оператор восстановления (reset) должен определять значение поля *eof* как $length = 0$. В языке Паскаль, например, операция reset определена именно таким разумным образом. Однако большинство распространенных файловых систем этого не делают, и именно поэтому мы придерживаемся такой стратегии при демонстрации использования последовательности.

Довольно часто в файловых системах допускаются некоторые послабления в строгой дисциплине последовательного доступа в том смысле, что серьезного наказания за этим не следует. В частности, они допускают позиционирование последовательности в любое место, а не только в начало. Такие расширения правил обычно сопровождаются введением дополнительной процедуры для работы с последовательностями $SetPos(s, k)$ при условии $0 \leqslant k < s.length$. Очевидно, что $SetPos(s,0)$ эквивалентно $Reset(s)$. Легкость реализации такого расширения и объясняется нашим определением операции записи, постулирующим, что запись всегда происходит в конец последовательности. Поэтому если операция записи происходит в момент,

когда позиция файла не соответствует его концу, то новый элемент заменит весь «хвост» этого файла, т. е. последовательность «усекается».

1.11.2. Буферизованные последовательности

При передаче данных во вторичную память или из нее отдельные разряды передаются как сплошной поток. Обычно устройства для передачи рассчитаны на строгие временные ограничения. Например, если данные записываются на ленту, то последняя движется с некоторой фиксированной скоростью и требует, чтобы данные для записи передавались так же с фиксированной скоростью. Когда источник иссякает, то лента «выключается» и скорость движения быстро уменьшается, но, конечно, не мгновенно. Поэтому между переданными данными и теми, которые последуют позже, появляется некоторый «зазор» (gap). Если мы хотим добиться высокой плотности записи данных, то необходимо стремиться к уменьшению числа зазоров. Поэтому, раз уж лента движется, данные передаются относительно большими *блоками.* Подобная ситуация остается справедливой и для магнитных дисков. Здесь данные хранятся на дорожках в фиксированном числе блоков так же фиксированного размера, его называют *размером блока* (block size). Фактически диск следует рассматривать как массив блоков; каждый блок читается или записывается целиком, обычно он содержит 2^k байтов (где $k = 8, 9$ или 10).

Однако наши программы не могут соблюдать какие-либо временные ограничения. И для того чтобы избавиться от этих сложностей, при передаче данных используется *буферизация.* Данные собираются в так называемую буферную переменную (в оперативной памяти), и когда накапливается достаточное для образования блока количество данных, этот блок передается.

Буферизация имеет еще одно преимущество: она позволяет процессу, формирующему (или получающему) данные, обрабатывать их одновременно с записью (или чтением) их устройством из буфера (или

в буфер). Фактически устройство удобно рассматривать как некоторый процесс, который сам просто копирует поток данных. Назначение же буфера — обеспечить некоторую степень независимости взаимодействия этих процессов, которые мы будем называть *производителем* (producer) и *потребителем* (consumer). Если, скажем, в некоторый момент потребитель работает медленно, то он может связаться с производителем попозже. Такая разобщенность часто очень важна для хорошего использования устройств, но это дает эффект только в тех случаях, когда скорости производителя и потребителя в среднем одни и те же, но во времени флюктуируют. Степень разобщенности увеличивается с ростом размера буфера.

Теперь перейдем к вопросу, как представлять буфер, если ориентироваться на индивидуальную укладку и извлечение отдельных элементов, а не на работу с целым блоком. Буфер фактически представляет собой очередь, построенную по принципу «первый пришел — первый ушел». Если буфер описан как массив, то две индексирующие переменные — предположим, это *in* и *out* — указывают позицию, в которую записывается элемент и из которой он извлекается. Идеально было бы, если бы у этого массива индексы не были ограничены. Однако вполне подходит и любой конечный массив, ведь надо учитывать, что однажды считанный элемент больше уже не используется. Поэтому его место можно повторно использовать. Это соображение приводит нас к идее *циклического буфера*. Операции размещения (depositing) и извлечения (fetching) элемента приводятся в следующем модуле, который экспортирует эти операции как процедуры, но закрывает сам буфер и его индексирующие переменные, т. е. фактически сам процесс буферизации, от процесса-

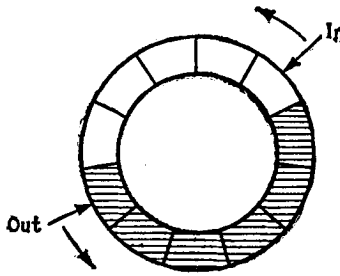


Рис. 1.9. Круговой буфер с индексами in и out.

клиента. В этот механизм входит также и переменная *n*, хранящая число элементов, находящихся в данный момент в буфере. Если через N обозначить размер буфера, то условие $0 \leqslant n \leqslant N$ будет, очевидно, инвариантом. Поэтому операцию *извлечения* (fetch) должно охранять условие $n > 0$ (т. е. буфер не пуст), а операцию *размещения* (deposit) — условие $n < N$ (буфер не полон). Неудовлетворение первого условия следует квалифицировать как ошибку программирования, а нарушение последнего — как неточность (failure) предложенной реализации (буфер слишком мал).

```
MODULE Buffer;
  EXPORT deposit, fetch;                                    (1.28)
  CONST N = 1024; (*размер буфера*)
  VAR n, in, out: CARDINAL;
    buf: ARRAY [0 .. N-1] OF WORD;
  PROCEDURE deposit(x: WORD);
  BEGIN
    IF n = N THEN HALT END;
    n := n+1; buf[in] := x; in := (in + 1) MOD N
  END deposit;
  PROCEDURE fetch(VAR x: WORD);
  BEGIN
    IF n = 0 THEN HALT END;
    n := n-1; x := buf[out]; out := (out + 1) MOD N
  END fetch;
BEGIN n := 0; in := 0; out := 0
END Buffer.
```

Такая реализация буфера пригодна лишь в том случае, если с процедурами *deposit* и *fetch* работает одна программа: она и производитель, и потребитель. Если же их активируют отдельные, одновременно работающие процессы, то такая схема выглядит слишком упрощенной. Это объясняется тем, что попытка размещения в полном буфере и попытка извлечения из пустого буфера в этом случае вполне законны. Выполнение таких действий следует просто задержать до того, как удовлетворится охраняющее условие. Эти задержки фактически требуют введения синхронизации одновременно работающих процессов. Задержки мы можем здесь представлять соответ-

ственно такими операторами:

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

Их следует подставить вместо двух операторов остановов (HALT) из (1.28). Однако мы не рекомендуем пользоваться таким решением, даже если известно, что два процесса управляются двумя отдельными реальными объектами. Причина заключается в том, что оба процессора по необходимости работают с одной переменной n, т. е. имеют доступ к одной и той же памяти. «Замерший» процесс постоянно пробует значение n, мешая своему партнеру, ибо память никогда не может отвечать на запросы более чем одного процесса. Следует избегать подобных *«мешающих ожиданий»* (busy waiting), и поэтому мы вводим некоторые механизмы, делающие детали синхронизации менее явными, т. е. фактически мы скрываем их. Этот механизм мы будем называть механизмом *сигналов*. Предполагается, что сигналы доступны из использующего модуля (будем называть его *Process*) наравне с множеством примитивных операций над такими сигналами.

Каждый сигнал s сопоставляется с охраняющим условием (P_s). Если процессу необходимо задержаться до тех пор, пока другой процесс не установит P_s, он должен, прежде чем работать, подождать прихода сигнала s. Такое действие выражается через оператор *Wait*(s). Если же другой процесс должен «открыть» P_s, то он сигнализирует об этом, выполняя оператор *Send*(s). Если P_s считать за предусловие для каждого оператора *Send*(s), то P_s можно рассматривать как постусловие для *Wait*(s).

```
DEFINITION MODULE Processes;
  TYPE Signal;                                        (1.29)
  PROCEDURE Wait(VAR s: Signal);
  PROCEDURE Send(VAR s: Signal);
  PROCEDURE Init(VAR s: Signal);
END Processes.
```

Теперь мы уже можем написать модуль буфера (1.28) в таком виде, что он будет правильно работать

и для отдельных, совместно работающих процессов.

```
MODULE Buffer;                                                    (1.30)
  IMPORT Signal, Wait, Send, Init;
  EXPORT deposit, fetch;
  CONST N = 1024; (*размер буфера*)
  VAR n, in, out: CARDINAL;
    nonfull: Signal;  (*n < N*)
    nonempty: Signal;  (*n > 0*)
    buf: ARRAY [0 .. N-1] OF WORD;
  PROCEDURE deposit(x: WORD);
  BEGIN
    IF n = N THEN Wait(nonfull) END ;
    n := n+1; buf[in] := x; in := (in + 1) MOD N;
    IF n = 1 THEN Send(nonempty) END
  END deposit;
  PROCEDURE fetch(VAR x: WORD);
  BEGIN
    IF n = 0 THEN Wait(nonempty) END ;
    n := n-1; x := buf[out]; out := (out + 1) MOD N;
    IF n = N-1 THEN Send(nonfull) END
  END fetch;
  BEGIN n := 0; in := 0; out := 0; Init(nonfull); Init(nonempty)
  END Buffer.
```

Однако необходимо сделать еще одно предостережение. Схема «развалится» при одновременном обращении производителя и потребителя к величине n для ее изменения. В результате — точно предсказать нельзя — ее значение будет или n + 1, или n — 1, но никак не n. Все, что необходимо, — это действительно защитить процессы от такого опасного взаимодействия. В общем случае все операции, *изменяющие значения общих переменных*, представляют собой потенциальную опасность.

Достаточное (но не всегда необходимое) условие — все общие переменные описывать как локальные в модуле, гарантирующем, что его процедуры одновременно выполняться не будут. Такие модули называются *мониторами* [1.7]. Подобное взаимно исключающее исполнение гарантирует, что в любой момент только один процесс активно работает с любой из процедур данного монитора. Если другой процесс обращается к некоторой процедуре того же монитора, то он будет автоматически задержан до тех пор, пока первый процесс не закончит свою процедуру.

Замечание: Активно работает — означает, что процесс выполняет любые операторы, кроме оператора ожидания (wait).

Вернемся теперь, наконец, к нашей задаче, где производитель или потребитель (или оба) требуют, чтобы данные были некоторого определенного размера. В следующем модуле (это вариант (1.30)) мы предполагаем, что у производителя размер блока данных — N_p, а у потребителя — N_c. В таком случае размер буфера N выбирается как кратное N_p и N_c. Дабы подчеркнуть симметричность операций извлечения и размещения, единственный счетчик *n* теперь будет представляться двумя счетчиками, а именно *ne* и *nf*. Они соответственно определяют число пустых и заполненных позиций в буфере. Если потребитель стоит, то *nf* указывает число элементов, направленных потребителю для обработки; если производитель ожидает, то *ne* задает число элементов, которое ему еще нужно доделать. (Поэтому соотношение ne + + nf = N не всегда справедливо.)

(1.31)

```
MODULE Buffer;
  IMPORT Signal, Wait, Send, Init;
  EXPORT deposit, fetch;
  CONST Np = 16; (*размер блока производителя*)
    Nc = 128; (*размер блока потребителя*)
    N = 1024; (*размер буфера, кратен Np и Nc*)
  VAR ne, nf: INTEGER;
    in, out: CARDINAL;
    nonfull: Signal; (*ne >= 0*)
    nonempty: Signal; (*nf >= 0*)
    buf: ARRAY [0 .. N-1] OF WORD;
  PROCEDURE deposit(VAR x: ARRAY OF WORD);
  BEGIN ne := ne - Np;
    IF ne < 0 THEN Wait(nonfull) END ;
    FOR I := 0 TO Np-1 DO buf[in] := x[i]; in := in + 1 END ;
    IF in = N THEN in := 0 END ;
    nf := nf + Np;
    IF nf >= 0 THEN Send(nonempty) END
  END deposit;
  PROCEDURE fetch(VAR x: ARRAY OF WORD);
  BEGIN nf := nf - Nc;
    IF nf < 0 THEN Wait(nonempty) END ;
    FOR i := 0 TO Nc-1 DO x[i] := buf[out]; out := out + 1 END;
    IF out = N THEN out := 0 END ;
    ne := ne + Nc;
    IF ne >= 0 THEN Send(nonfull) END
  END fetch;
BEGIN
  ne := N; nf := 0; in := 0; out := 0; Init(nonfull); Init(nonempty)
END Buffer.
```

1.11.3. Стандартные ввод и вывод

Под стандартными вводом и выводом мы понимаем передачу данных в вычислительную систему от по существу внешнего действующего объекта, скажем оператора, и наоборот. Ввод обычно может начинаться с клавишной панели, а вывод заканчивается экраном дисплея. В любом случае для этого процесса характерно, что сообщение можно прочитать, и обычно оно состоит из последовательности символов. Такая «читаемость» приводит еще к одному усложнению операций, связанных с реальным вводом и выводом. Кроме фактической передачи данных сюда еще включается и *трансформация представления.* Например, числа, обычно считающиеся неделимыми единицами в разрядном представлении, необходимо преобразовывать в удобочитаемое, десятичное представление. Структуру данных нужно выделять подходящими отступами, этот процесс называется *форматированием.*

Каким бы ни было преобразование, концепция последовательности снова приводит к значительному упрощению задачи. В основе этого утверждения лежит заключение, что если множество данных можно рассматривать как некоторую последовательность, то преобразование этой последовательности можно осуществить как последовательность (идентичных) преобразований элементов.

$$T(\langle s_0, s_1, \ldots, s_{n-1}\rangle) = \langle T(s_0),\ T(s_1), \ldots, T(s_{n-1})\rangle \qquad (1.32)$$

Мы проведем небольшое исследование операций, необходимых для преобразования натуральных чисел при вводе и выводе. В основе лежит утверждение, что любое число x, представленное последовательностью десятичных цифр $d = \langle d_{n-1}, \ldots, d_1, d_0\rangle$, имеет значение

$$x = \mathbf{S}i\colon 0 \leqslant i < n\colon d_i * 10^i$$

Представим теперь, что последовательность d нужно прочитать и преобразовать, а получившееся числовое значение присвоить переменной x. Ниже (1.33) приводится простой алгоритм, он заканчивает работу с приходом первого символа, отличного от цифры.

(Арифметическое переполнение здесь не учитывается.)

```
x := 0; Read(ch);
WHILE ("0" <= ch) & (ch <= "9") DO                    (1.33)
  x := 10*x + (ORD(ch) - ORD("0")); Read(ch)
END
```

В случае вывода преобразование усложняется, поскольку при декомпозиции x в десятичное представление цифры идут в обратном порядке. Последняя цифра формируется первой, ее вычисляют так: x MOD10. Это приводит к необходимости введения буфера — очереди, построенной по принципу «последний пришел — первый обслужился». Такой буфер мы будем представлять в виде массива d с индексом i, в результате получается следующая программа:

```
i := 0;
REPEAT d[i] := x MOD 10; x := x DIV 10; i := i+1      (1.34)
UNTIL x = 0;
REPEAT i := i-1; Write(CHR(d[i] + ORD("0"))
UNTIL i = 0
```

Замечание: Систематическая замена в (1.33) и (1.34) константы 10 на другую константу B (>0) порождает подпрограммы перевода в представлении с основанием B. Часто используются варианты для B = 8 (восьмеричное представление) и B = 16 (шестнадцатеричное представление), поскольку входящие в подпрограммы умножения и деления сводятся к простому сдвигу двоичных чисел.

Очевидно, нет никакой необходимости детально определять эти «вездесущие» операции в каждой программе. Поэтому мы считаем, что стандартный модуль (utility module), обеспечивающий большинство общих, стандартных операций ввода и вывода для чисел и строк, всегда в программе «присутствует». На такой модуль (он называется *InOut*) мы ссылаемся во многих программах нашей книги. Его процедуры обеспечивают ввод и вывод отдельных символов, целых, количественных числительных или строк. Поскольку мы его используем очень часто, то раздел определений этого модуля приводится здесь полностью.

В его процедурах чтения (read) и записи (write) нет параметра, указывающего, с какой последова-

(1.35)

```
DEFINITION MODULE InOut;
 FROM SYSTEM IMPORT WORD;
 FROM FileSystem IMPORT File;

 CONST EOL = 36C;
 VAR Done: BOOLEAN;
  termCH: CHAR; (*концевой символ в ReadInt, ReadCard*)
  in, out: File; (*только для исключительных случаев*)

 PROCEDURE OpenInput(defext: ARRAY OF CHAR);
  (*запрашивается имя файла и открывается входной файл "in".
    Done := "file was successfully opened"
    Если файл открыт, то последующий ввод идет из этого файла.
    Если имя заканчивалось символом ".", то добавляется расширение defext*)

 PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
  (*запрашивается имя файла и открывается выходной файл "out"
    Done := "file was successfully opened.
    если файл открыт, то последующая выдача идет в этот файл *)

 PROCEDURE CloseInput;
  (*входной файл закрывается, ввод возвращается к терминалу*)

 PROCEDURE CloseOutput;
  (*выходной файл закрывается, вывод возвращается к терминалу*)

 PROCEDURE Read(VAR ch: CHAR);
  (*Done := NOT in.eof*)

 PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
  (*чтение строки, т. е. последовательности символов, не содержащей
    пробелов или управляющих символов; начальные пробелы
    игнорируются. Ввод заканчивается любым символом <= " ";
    этот символ присваивается termCH.
    При вводе с терминала для возврата используется символ DEL.*)

 PROCEDURE ReadInt(VAR x: INTEGER);
  (*чтение строки и преобразование ее в целое. Синтаксис:
    целое = ["+"|"-"] цифра{цифра}.
    Начальные пробелы игнорируются.
    Done := "integer was read"*)

 PROCEDURE ReadCard(VAR x: CARDINAL);
  (*чтение строки и преобразование ее в число типа CARDINAL.
    Синтаксис: число = цифра{цифра}.
    Начальные пробелы игнорируются.
    Done := "cardinal was read"*)

 PROCEDURE ReadWrd(VAR w: WORD);
```

```
  (*Done := NOT in.eof*)
 PROCEDURE Write(ch: CHAR);
 PROCEDURE WriteLn;  (*окончание строки*)
 PROCEDURE WriteString(s: ARRAY OF CHAR);
 PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
  (*запись целого x в файл out с минимум n символами.
    Если n больше числа требуемых цифр, то
    перед числом добавляются пробелы*)
 PROCEDURE WriteCard(x, n: CARDINAL);
 PROCEDURE WriteOct(x, n: CARDINAL);
 PROCEDURE WriteHex(x, n: CARDINAL);
 PROCEDURE WriteWrd(w: WORD);
END InOut.
```

тельностью (файлом) идет работа. Подразумевается, что данные читаются со стандартного устройства ввода (клавишная панель), а записываются опять же на стандартное выводное устройство. Однако такие допущения можно отменить, обратившись к процедурам *OpenInput* и *OpenOutput*. Они требуют от оператора имя файла и подставляют заданные файлы вместо стандартных устройств, изменяя тем самым потоки данных. Процедуры *CloseInput* и *CloseOutput* возвращают систему к работе со стандартными устройствами.

1.12. ПОИСК

Одно из наиболее часто встречающихся в программировании действий — поиск. Он же представляет собой идеальную задачу, на которой можно испытывать различные структуры данных по мере их появления. Существует несколько основных «вариаций этой темы», и для них создано много различных алгоритмов. При дальнейшем рассмотрении мы исходим из такого принципиального допущения: группа данных, в которой необходимо отыскать заданный элемент, *фиксирована*. Будем считать, что множество из N элементов задано, скажем, в виде такого массива

```
a: ARRAY[0 .. N — 1] OF item
```

Обычно тип *item* описывает запись с некоторым полем, выполняющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному «аргументу поиска» x. Полученный в результате индекс i, удовлетворяющий условию $a[i].key = x$, обеспечивает доступ к другим полям обнаруженного элемента. Так как нас интересует в первую очередь сам процесс поиска, а не обнаруженные данные, то мы будем считать, что тип *item* включает только ключ, т. е. он *есть* ключ (key).

1.12.1. Линейный поиск

Если нет никакой дополнительной информации о разыскиваемых данных, то очевидный подход — простой последовательный просмотр массива с увеличением шаг за шагом той его части, где желаемого элемента не обнаружено. Такой метод называется *линейным поиском*. Условия окончания поиска таковы:

1. Элемент найден, т. е. $a_i = x$.
2. Весь массив просмотрен и совпадения не обнаружено.

Это дает нам линейный алгоритм:

```
i := 0;
WHILE (i < N) & (a[i] # x) DO i := i+1 END
```
(1.36)

Обратите внимание, что порядок элементов в логическом выражении имеет существенное значение. Инвариант цикла, т. е. условие, выполняющееся перед каждым увеличением индекса i, выглядит так:

$$(0 \le i < N) \,\&\, (Ak : 0 \le k < i : a_k \ne x) \tag{1.37}$$

Он говорит, что для всех значений k, меньших чем i, совпадения не было. Отсюда и из того факта, что поиск заканчивается только в случае ложности условия в заголовке цикла, можно вывести окончательное условие:

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

Это условие не только указывает на желаемый результат, но из него же следует, что если элемент найден, то он найден вместе с минимально возможным индексом, т. е. это первый из таких элементов. Равенство i = N свидетельствует, что совпадения не существует.

Совершенно очевидно, что окончание цикла гарантировано, поскольку на каждом шаге значение i увеличивается, и, следовательно, оно, конечно же, достигнет за конечное число шагов предела N; фактически же, если совпадения не было, это произойдет после N шагов.

Ясно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. А можно ли эту работу упростить и таким образом убыстрить поиск? Единственная возможность — попытаться упростить само логическое выражение, ведь оно состоит из двух членов. Следовательно, единственный шанс на пути к более простому решению — сформулировать простое условие, эквивалентное нашему сложному. Это можно сделать, если мы гарантируем, что совпадение всегда произойдет. Для этого достаточно в конец массива поместить дополнительный элемент со значением x. Назовем такой вспомогательный элемент «барьером», ведь он охраняет нас от перехода за пределы массива. Теперь массив будет описан так:

```
a: ARRAY[0 .. N] OF INTEGER
```

и алгоритм линейного поиска с барьером выглядит следующим образом:

```
a[N] := x; i := 0;                                        (1.38)
WHILE a[i] # x DO i := i+1 END
```

Результирующее условие, выведенное из того же инварианта, что и прежде:

$$(a_i = x) \,\&\, (\forall k : 0 \le k < i : a_k \neq x)$$

Ясно, что равенство i = N свидетельствует о том, что совпадения (если не считать совпадения с барьером) не было.

1.12.2. Поиск делением пополам (двоичный поиск)

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Вообразите себе телефонный справочник, в котором фамилии не будут расположены по порядку. Это нечто совершенно бесполезное Поэтому мы приводим алгоритм, основанный на знании того, что массив a упорядочен, т. е. удовлетворяет условию

$$Ak:\ 1 \leqslant k < N:\ a_{k-1} \leqslant a_k \tag{1.39}$$

Основная идея — выбрать случайно некоторый элемент, предположим a_m, и сравнить его с аргументом поиска x. Если он равен x, то поиск заканчивается, если он меньше x, то мы заключаем, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше x, то исключаются индексы больше и равные m. Это соображение приводит нас к следующему алгоритму (он называется *«поиском делением пополам»*). Здесь две индексные переменные L и R отмечают соответственно левый и правый конец секции массива a, где еще может быть обнаружен требуемый элемент.

```
L := 0; R := N-1; found := FALSE;                    (1.40)
WHILE (L ≤ R) & ~found DO
  m := любое значение между L и R;
  IF a[m] = x THEN found := TRUE
  ELSIF a[m] < x THEN L := m+1
  ELSE R := m-1
  END
END
```

Инвариант цикла, т. е. условие, выполняющееся перед каждым шагом, таков:

$$(L \leq R)\ \&\ (Ak: 0 \leq k < L: a_k < x)\ \&\ (Ak: R < k < N: a_k > x) \tag{1.41}$$

из чего выводится результат

$$\text{found OR } ((L > R) \,\&\, (\forall k : 0 \le k < L : a_k < x) \,\&\, (\forall k : R < k < N : a_k > x))$$

откуда следует

$$(a_m = x) \text{ OR } (\exists k : 0 \le k < N : a_k = x)$$

Выбор m совершенно произволен в том смысле, что корректность алгоритма от него не зависит. Однако на его эффективность выбор влияет. Ясно, что наша задача — исключить на каждом шагу из дальнейшего поиска, каким бы ни был результат сравнения, как можно больше элементов. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. В результате максимальное число сравнений равно $\log N$, округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое число сравнений — $N/2$.

Эффективность можно несколько улучшить, поменяв местами заголовки условных операторов. Проверку на равенство можно выполнять во вторую очередь, так как она встречается лишь единожды и приводит к окончанию работы. Но более существенно следующее соображение: нельзя ли, как и при линейном поиске, отыскать такое решение, которое опять бы упростило условие окончания. И мы действительно находим такой быстрый алгоритм, как только отказываемся от наивного желания кончить поиск при фиксации совпадения. На первый взгляд это кажется странным, однако при внимательном рассмотрении обнаруживается, что выигрыш в эффективности на каждом шаге превосходит потери от сравнения с несколькими дополнительными элементами. Напомним, что число шагов в худшем случае — $\log N$. Быстрый алгоритм основан на следующем инварианте:

$$(\forall k : 0 \le k < L : a_k < x) \,\&\, (\forall k : R \le k < N : a_k \ge x) \qquad (1.42)$$

причем поиск продолжается до тех пор, пока обе секции не «накроют» массив целиком.

```
L := 0; R := N;                                          (1.43)
WHILE L < R DO
  m := (L+R) DIV 2;
  IF a[k] < x THEN L := m+1 ELSE R := m END
END
```

Условие окончания — $L \geqslant R$, но достижимо ли оно? Для доказательства этого нам необходимо показать, что при всех обстоятельствах разность $R - L$ на каждом шаге убывает. В начале каждого шага $L < R$. Для среднего арифметического m справедливо условие $L \leqslant m < R$. Следовательно, разность действительно убывает, ведь либо L увеличивается при присваивании ему значения $m + 1$, либо R уменьшается при присваивании значения m. При $L = R$ повторение цикла заканчивается. Однако наш инвариант и условие $L = R$ еще не свидетельствуют о совпадении. Конечно, при $R = N$ никаких совпадений нет. В других же случаях мы должны учитывать, что элемент $a[R]$ в сравнениях никогда не участвует. Следовательно, необходима дополнительная проверка на равенство $a[R] = x$. В отличие от первого нашего решения (1.40) приведенный алгоритм, как и в случае линейного поиска, находит совпадающий элемент с наименьшим индексом.

1.12.3. Поиск в таблице

Поиск в массиве иногда называют *поиском в таблице*, особенно если ключ сам является составным объектом, таким, как массив чисел или символов. Часто встречается именно последний случай, когда массивы символов называют строками или словами. *Строковый* тип определяется так:

```
String = ARRAY[0 .. M - 1] OF CHAR                       (1.44)
```

соответственно определяется и отношение порядка для строк:

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

Для того чтобы установить факт совпадения, мы должны, очевидно, убедиться, что все символы сравниваемых строк соответственно равны один другому. Поэтому сравнение составных операндов сводится к поиску их несовпадающих частей, т. е. к поиску «на неравенство». Если неравных частей не существует, то можно говорить о равенстве. Предположим, что размер слов достаточно мал, скажем, меньше 30. В этом случае мы будем использовать линейный поиск и поступать таким образом.

Для большинства практических приложений желательно исходить из того, что строки имеют переменный размер. Это предполагает, что размер указывается в каждой отдельной строке. Если исходить из ранее описанного типа, то размер не должен превосходить максимального размера M. Такая схема достаточно гибка и подходит для многих случаев, в то же время она позволяет избежать сложностей динамического распределения памяти. Чаще всего используются два таких представления размера строк:

1. Размер неявно указывается путем добавления концевого символа, больше этот символ нигде не употребляется. Обычно для этой цели используется «непечатаемый» символ со значением 0C. (Для дальнейшего важно, что это *минимальный* символ из всего множества символов.)

2. Размер явно хранится в качестве первого элемента массива, т. е. строка s имеет следующий вид: $s = s_0, s_1, s_2, \ldots, s_{N-1}$. Здесь $s_1, \ldots, s_{N-1}$ — фактические символы строки, а $s_0 = CHR(N)$. Такой прием имеет то преимущество, что размер явно доступен, недостаток же в том, что этот размер ограничен размером множества символов (256).

В последующем алгоритме поиска мы отдаем предпочтение первой схеме. В этом случае сравнение строк выполняется так:

```
i := 0;                                                    (1.45)
WHILE (x[i] = y[i]) & (x[i] # 0C) DO i := i+1 END
```

Концевой символ работает теперь как барьер, а инвариант цикла таков:

$$A_j\colon\ 0 \leqslant j < i\colon\ x_j = y_j \neq 0C$$

Результирующее же условие выглядит следующим образом:

$$((x_i \neq y_i)\mathrm{OR}(x_i = 0\mathrm{C})) \,\&\, (\mathrm{Aj}:\ 0 \leqslant j < i:\ x_i = y_j \neq 0\mathrm{C})$$

При условии $x_i = y_i$ считается, что x и y совпадают, если же $x_i < y_i$, то $x < y$.

Теперь мы готовы вернуться к задаче поиска в таблице. Он требует «вложенных» поисков, а именно: поиска по строчкам таблицы, а для каждой строчки последовательных сравнений — между компонентами. Например, пусть таблица T и аргумент поиска x определяются таким образом:

```
T: ARRAY [0 .. N-1] OF String;
x: String
```

Допустим, N достаточно велико, а таблица упорядочена в алфавитном порядке. Воспользуемся поиском делением пополам. Используя соответствующие алгоритмы (1.43) и сравнения строк (1.45), о которых шла речь выше, получаем такой фрагмент программы:

```
L := 0; R := N;
WHILE L < R DO                                              (1.46)
  m := (L+R) DIV 2; i := 0;
  WHILE (T[m,i] = x[i]) & (x[i] # 0C) 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 (T[R,i] = x[i]) & (x[i] # 0C) DO i := i+1 END
END
(* (R < N) & (T[R,i] = x[i])  фиксирует  совпадение *)
```

1.12.4. Прямой поиск строки

Часто приходится сталкиваться со специфическим поиском, так называемым *поиском строки*. Его можно определить следующим образом. Пусть задан массив s из N элементов и массив p из M элементов, причем $0 < M \leqslant N$. Описаны они так:

```
s: ARRAY[0 .. N — 1] OF item
p: ARRAY[0 .. M — 1] OF item
```

Поиск строки обнаруживает первое вхождение p в s. Обычно item — это символы, т. е. s можно считать некоторым текстом, а p — образом или словом, и мы хотим найти первое вхождение этого слова в указанном тексте. Это действие типично для любых систем обработки текстов, отсюда и очевидная заинтересованность в поиске эффективного алгоритма для этой задачи. Однако, прежде чем обращать внимание на эффективность, давайте сначала разберем некий «прямолинейный» алгоритм поиска. Мы его будем называть *прямым поиском строки.*

Еще до определения алгоритма нужно более точно сформулировать, что же мы от него желаем получить. Будем считать результатом индекс i, указывающий на первое с начала строки совпадение с образом. С этой целью введем предикат P(i, j):

$$P(i, j) = Ak: 0 \leqslant k < j: s_{i+k} = p_k \qquad (1.47)$$

Наш результирующий индекс, очевидно, должен удовлетворять этому предикату P(i, M). Но этого недостаточно. Поскольку поиск должен обнаружить *первое* вхождение образа, P(k, M) должно быть ложным для всех $k < i$. Обозначим это условие через Q(i):

$$Q(i) = Ak: 0 \leqslant k < i: \sim P(k, M) \qquad (1.48)$$

Поставленная задача сразу же наводит на мысль оформить поиск как повторяюшееся сравнение, и мы предлагаем такой вариант:

```
i := -1;
REPEAT i := i+1; (* Q(i) *)
  found := P(i,M)
UNTIL found OR (i = N-M)
```

Вычисление P, естественно, вновь сводится к повторяющимся сравнениям отдельных символов. Если применить к P теорему Моргана, то получается, что итерации должны «искать» несовпадение между образом и строкой символов.

$$P(i, j) = (Ak: 0 \leqslant k < j: s_{i+k} = p_k) = (\sim Ek: 0 \leqslant k < j: s_{i+k} \neq p_k)$$

В результате этого уточнения мы приходим к повторению внутри повторения. Предикаты P и Q вклю-

чаются в соответствующие места программы как примечания. Они представляют собой инварианты циклов упомянутых итеративных процессов.

```
i := -1;                                                        (1.49)
REPEAT i := i+1; j := 0; (* Q(i) *)
  WHILE (j < M) & (s[i+j] = p[j]) DO (* P(i,j+1) *) j := j+1 END
  (* Q(i) & P(i,j) & ((j = M) OR (s[i+j] # p[j])) *)
UNTIL (j = M) OR (i = N-M)
```

В действительности член $j = M$ в условии окончания соответствует условию *found*, так как из него следует $P(i, M)$. Член $i = N - M$ имплицирует $Q(N - M)$ и тем самым свидетельствует, что нигде в строке совпадения нет. Если итерации продолжаются при $j < M$, то они должны продолжаться и при $s_{i+j} = p_j$. В соответствии с (1.47) отсюда следует $\sim P(i, j)$, затем из-за (1.48) следует $Q(i + 1)$, что и подтверждает справедливость $Q(i)$ после очередного увеличения i.

Анализ прямого поиска в строке. Этот алгоритм работает достаточно эффективно, если мы допускаем, что несовпадение пары символов происходит по крайней мере после всего лишь нескольких сравнений во внутреннем цикле. При большой мощности типа item это достаточно частый случай. Можно предполагать, что для текстов, составленных из 128 символов, несовпадение будет обнаруживаться после одной или двух проверок. Тем не менее в худшем случае производительность будет внушать опасение. Возьмем, например, строку, состоящую из $N - 1$ символов A и единственного B, а образ содержит $M - 1$ символов A и опять B. Чтобы обнаружить совпадение в конце строки, требуется произвести порядка $N*M$ сравнений. К счастью, как мы скоро увидим, есть прием, который существенно улучшает обработку этого неблагоприятного варианта.

1.12.5 Поиск в строке. Алгоритм Кнута, Мориса и Пратта

Приблизительно в 1970 г. Д. Кнут, Д. Морис и В. Пратт изобрели алгоритм, фактически требующий только N сравнений даже в самом плохом случае

[1.8]. Новый алгоритм основывается на том соображении, что, начиная каждый раз сравнение образа с самого начала, мы можем уничтожать ценную информацию. После частичного совпадения начальной части образа с соответствующими символами строки мы фактически знаем пройденную часть строки и можем «вычислить» некоторые сведения (на основе самого образа), с помощью которых потом быстро продвинемся по тексту. Приведенный пример поиска слова *Hooligan* показывает принцип работы такого алгоритма. Символы, подвергшиеся сравнению, здесь подчеркнуты. Обратите внимание: при каждом несовпадении двух символов образ сдвигается на все пройденное расстояние, поскольку меньшие сдвиги не могут привести к полному совпадению.

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

КМП-алгоритм записывается так (опять используется предикат P (1.47) и Q (1.48)):

```
i := 0; j := 0,
WHILE (j < M) & (i < N) DO                                    (1.50)
  (* Q(i-j) & P(i-j, j) *)
  WHILE (j >= 0) & (s[i] # p[j]) DO j := D END;
  i := i+1; j := j+1
END
```

Однако такая запись умышленно не совсем точная, поскольку в ней есть сдвиг на неопределенную величину D. Вскоре мы к ней вернемся, а теперь сначала отметим, что условия $Q(i-j)$ и $P(i-j, j)$ сохраняются как глобальные инварианты и к ним можно добавить отношения $0 \leqslant i < N$ и $0 \leqslant j < M$. Это предполагает, что мы должны отказаться от согла-

шения, что i всегда отмечает в тексте текущее положение первого символа образа. Точнее, выравненное положение образа теперь i — j.

Если алгоритм заканчивает работу по причине j = M, то из составляющей P(i — j, j) следует справедливость P(i — M, M), т. е., согласно (1.47), совпадение начинается с позиции i — M. Если же работа окончена из-за i = N, то поскольку j < M, из инварианта Q(i) следует, что совпадения вообще нет.

Теперь мы должны показать, что алгоритм никогда не делает инвариант ложным. Легко видеть, что в начале i = j = 0, и он истинен. Сначала исследуем эффект от двух операторов, увеличивающих i и j на единицу. Они, очевидно, не сдвигают образ вправо и не делают ложным Q(i — j), поскольку разность остается неизменной. Но, может быть, они делают ложным P(i—j, j) — вторую составляющую инварианта? Обращаем внимание, что в этой точке истинно отрицание выражения в заголовке внутреннего цикла, т. е. либо j < 0, либо $s_i = p_j$. Последнее увеличивает частичное совпадение и устанавливает P(i — j, j + 1), а первое так же постулирует истинность P(i — j, j + 1). Следовательно, увеличение на единицу i и j не может также сделать ложным тот или другой инвариант. Но в алгоритме остается только еще присваивание $j := D$. Мы просто постулируем, что значение D всегда таково, что замена j на D оставляет инвариант справедливым.

Для того чтобы найти соответствующее выражение для D, мы должны вначале понять смысл этого присваивания. При условии что D < j, присваивание соответствует *сдвигу образа вправо* на j — D позиций. Естественно, мы хотим, чтобы сдвиг был настолько больше, насколько это возможно, т. е. D должно быть как можно меньше. Этот процесс показан на рис. 1.10.

Если инвариант P(i — j, j)&Q(i — j) после присваивания j значения D истинен, то перед ним должно быть истинно P(i — D, D)&Q(i — D). Это предусловие и будет поэтому нашим ориентиром при поиске подходящего выражения для D. Основное соображение:

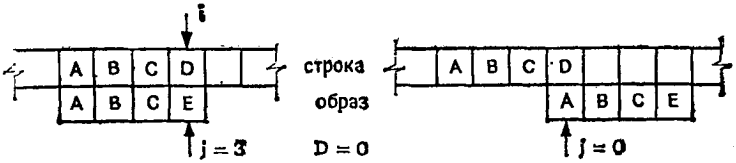


Рис. 1.10. Присваивание j := D сдвигает образ на j — D позиций.

благодаря P(i — j, j) мы знаем, что

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

(мы только что просматривали первые j символов образа и убедились в их совпадении с текстом). Поэтому условие P(i — D, D) с $D \leqslant j$, т. е.

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

превращается в

$$p_0 \cdots p_{D-1} = p_{j-D} \cdots p_{j-1} \qquad (1.51)$$

и предикат $\sim$P(i — k, M) для k = 1..j — D (чтобы убедиться в инвариантности Q(i — D)) превращается в

$$p_0 \cdots p_{k-1} \neq p_{j-k} \cdots p_{j-1} \quad \text{для всех } k = 1 \ldots j - D \qquad (1.52)$$

Важный вывод: очевидно, что значение D определяется одним лишь образом и не зависит от строки текста. Условия (1.51) и (1.52) говорят, что для определения D мы должны для каждого j найти наименьшее D, т. е. самую длинную последовательность символов образа, непосредственно предшествующих позиции j, которая совпадает полностью с началом образа. Для каждого j такое D мы будем обозначать через d_j. Так как эти значения зависят только от образа, то перед началом фактического поиска можно вычислить вспомогательную таблицу d; эти вычисления сводятся к некоторой *предтрансляции* образа. Соответствующие усилия будут оправданными, если размер текста значительно превышает размер образа ($M \ll N$). Если нужно искать многие вхождения одного и того же образа, то можно поль-

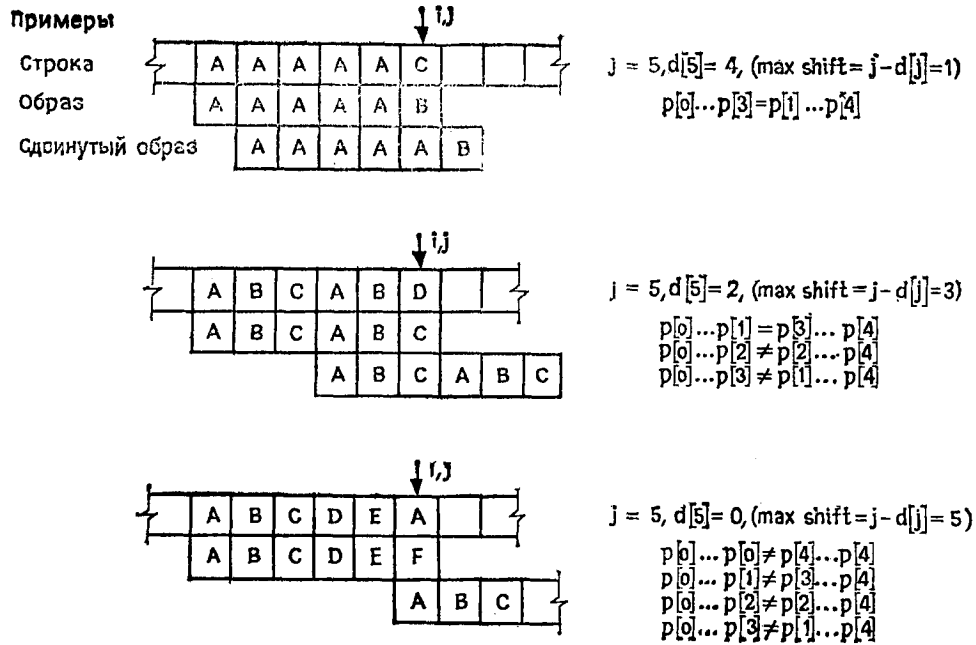


Рис. 1.11. Частичное совпадение с образом и вычисление d_j.

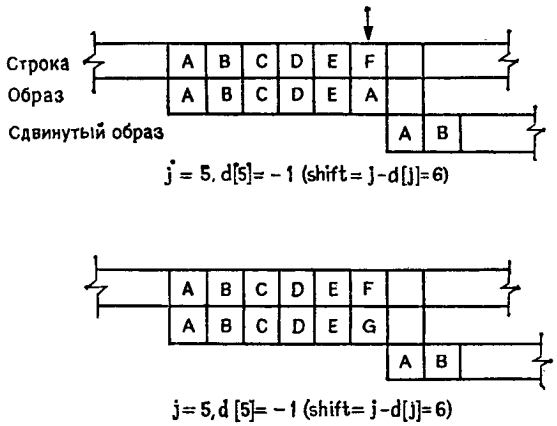


Рис. 1.12. Образ сдвигается за его последний символ.

зоваться одними и теми же d. Приведенные примеры объясняют функцию d.

Последний пример на рис. 1.11 наводит на мысль, что мы можем поступать еще лучше: так как p_j равно A вместо F, то соответствующий символ строки не может быть символом A из-за того, что условие $s_i \neq p_j$ заканчивает цикл. Следовательно, сдвиг на 5 не приведет к последующему совпадению, и поэтому мы можем увеличить размер сдвига до шести (см. рис. 1.11, верхний пример). Учитывая это, мы переопределим вычисление d_j как поиск самой длинной совпадающей последовательности

$$p_0 \cdots p_{d_j-1} = p_{j-d_j} \cdots p_{j-1}$$

с дополнительным ограничением $p_{d_j} \neq p_j$. Если никаких совпадений нет, то мы считаем $d_j = -1$, что указывает на сдвиг «на целый» образ относительно его текущей позиции (см. рис. 1.12, нижняя часть).

Ясно, что вычисление d_j само представляет собой первое приложение поиска в строке, и мы для этого можем использовать быструю версию КМП-алгоритма. Это и демонстрирует программа 1.2, состоящая из двух частей. В первой читается строка s, а затем идут итерации, начинающиеся с чтения образа, затем сле-

```
MODULE KMP;
 FROM InOut IMPORT
  OpenInput, CloseInput, Read, Write, WriteLn, Done;
 CONST Mmax = 100; Nmax = 10000; ESC = 33C;
 VAR i, j, k, k0, M, N: INTEGER;
   ch: CHAR;
   p: ARRAY [0..Mmax-1] OF CHAR;  (*образ*)
   s: ARRAY [0..Nmax-1] OF CHAR;  (*строка*)
   d: ARRAY [0..Mmax-1] OF INTEGER;
BEGIN OpenInput("TEXT"); N := 0; Read(ch);
 WHILE Done DO
  Write(ch); s[N] := ch; N := N+1; Read(ch)
 END;
 CloseInput;
 LOOP WriteLn; Write(">"); M := 0; Read(ch);
  WHILE ch > " " DO
   Write(ch); p[M] := ch; M := M+1; Read(ch)
  END;
  WriteLn;
  IF ch = ESC THEN EXIT END;
  j := 0; k := -1; d[0] := -1;
  WHILE j < M-1 DO
   WHILE (k >= 0) & (p[j] # p[k]) DO k := d[k] END;
   j := j+1; k := k+1;
   IF p[j] = p[k] THEN d[j] := d[k] ELSE d[j] := k END
  END;
  i := 0; j := 0; k := 0;
  WHILE (j < M) & (i < N) DO
   WHILE k <= i DO Write(s[k]); k := k+1 END;
   WHILE (j >= 0) & (s[i] # p[j]) DO j := d[j] END;
   i := i+1; j := j+1
  END;
  IF j = M THEN Write("!") (*найден*) END
 END
END KMP.
```

Прогр. 1.2. Поиск в строке методом КМП.

дуют его предтрансляция и, наконец, сам поиск.

Анализ КМП-поиска. Точный анализ КМП-поиска, как и сам его алгоритм, весьма сложен. В работе [1.8] его изобретатели доказывают, что требуется порядка $M + N$ сравнений символов, что значительно

лучше, чем M * N сравнений из прямого поиска. Они так же отмечают то приятное свойство, что указатель сканирования i никогда не возвращается назад, в то время как при прямом поиске после несовпадения просмотр всегда начинается с первого символа образа и поэтому может включать символы, которые ранее уже просматривались. Это может привести к неприятным затруднениям, если строка читается из вторичной памяти, ведь в этом случае возврат обходится дорого. Даже при буферизованном вводе может встретиться столь большой образ, что возврат превысит емкость буфера.

1.12.6. Поиск в строке. Алгоритм Боуера и Мура

Остроумная схема КМП-поиска дает подлинный выигрыш только тогда, когда неудаче предшествовало некоторое число совпадений. Лишь в этом случае образ сдвигается более чем на единицу. К несчастью, это скорее исключение, чем правило: совпадения встречаются значительно реже, чем несовпадения. Поэтому выигрыш от использования КМП-стратегии в большинстве случаев поиска в обычных текстах весьма незначителен. Метод же, о котором сейчас пойдет речь, в действительности не только улучшает обработку самого плохого случая, но дает выигрыш в промежуточных ситуациях. Его предложили Р. Боуер и Д. Мур приблизительно в 1975 г; мы будем называть его БМ-поиском. Вначале мы дадим упрощенную версию этого алгоритма, а затем перейдем к той, которую приводят Боуер и Мур.

БМ-поиск основывается на необычном соображении — сравнение символов начинается с конца образа, а не с начала. Как и в случае КМП-поиска, образ перед фактическим поиском трансформируется в некоторую таблицу. Пусть для каждого символа x из алфавита d_x — расстояние от самого правого в образе вхождения x до его конца. Представим себе, что обнаружено расхождение между образом и строкой. В этом случае образ сразу же можно сдвинуть вправо на $d_{p_{M-1}}$ позиций, т. е. на число позиций, ско-

рее всего большее единицы. Если p_{M-1} в образе вообще не встречается, то сдвиг становится даже больше, а именно сдвигать можно на длину всего образа. Вот пример, иллюстрирующий этот процесс:

```
Hoola-Hoola girls like Hooligans,
Hooligan
     Hooligan
       Hooligan
              Hooligan
                     Hooligan
```

Поскольку сравнение отдельных символов теперь проходит справа налево, то более удобно использовать такие слегка модифицированные версии предикатов P и Q:

$$P(i,j) = Ak: j \leq k < M: s_{i-j+k} = p_k \tag{1.53}$$
$$Q(i) = Ak: 0 \leq k < i: \sim P(i,0)$$

Эти предикаты используются при описании БМ-алгоритма, и с их помощью формулируются инвариантные условия.

```
i := M; j := M;
WHILE (j > 0) & (i < N) DO                                   (1.54)
  (* Q(i-M) *) j := M; k := i;
  WHILE (j > 0) & (s[k-1] = p[j-1]) DO
    (* P(k-j, j) & (k-j = i-M) *)
    k := k-1; j := j-1
  END;
  i := i + d[s[i-1]]
END
```

Индексы удовлетворяют условиям $0 \leqslant j \leqslant M$ и $0 \leqslant i,\ k \leqslant N$. Поэтому из окончания при $j = 0$ вместе с $P(k-j, j)$ следует $P(k, 0)$, т. е. совпадение в позиции k. При окончании с $j > 0$ небходимо, чтобы $i = N$, следовательно, из $Q(i-M)$ имплицируется $Q(N-M)$, сигнализирующее, что совпадения не существует. Конечно, мы продолжаем убеждаться, что $Q(i-M)$ и $P(k-j, j)$ действительно инварианты двух циклов. В начале повторений они, естественно, удовлетворяются, поскольку $Q(0)$ и $P(x, M)$ всегда истинны.

Давайте вначале разберемся в последствиях двух операторов, уменьшающих k и j. На $Q(i-M)$ они действуют, и так как было выяснено, что $s_{k-1} = p_{j-1}$, то справедливость $P(k-j, j-1)$ как предусловия гарантирует справедливость $P(k-j, j)$ в качестве постусловия. Если внутренний цикл заканчивается при $j > 0$, то из $s_{k-1} \neq p_{j-1}$ следует $\sim P(k-j, 0)$, так как

$$\sim P(i, 0) = Ek: 0 \leqslant k < M: s_{i+k} \neq p_k$$

Более того, из-за $k-j = M-i$ следует $Q(i-M) \& \sim \sim P(k-j, 0) = Q(i+1, M)$, фиксирующее несовпадение в позиции $i-M+1$.

Теперь нам нужно показать, что оператор $i := i + d_{s_{i-1}}$ никогда не делает инвариант ложным. Гарантировано, что перед этим оператором $Q(i + d_{s_{i-1}} - M)$ истинно. Так как мы знаем, что справедливо $Q(i+1-M)$, то достаточно установить справедливость $\sim P(i+h-M)$ для $h = 2, 3, \ldots, d_{s_{i-1}}$. Напоминаем, что d_x определяется как расстояние от самого правого в образе вхождения x до его конца. Формально это выражается так:

$$Ak: M - d_x \leqslant k < M-1: p_k \neq x$$

Подставляя s_i вместо x, получаем

$$Ah: M - d_{s_{i-1}} \leq h < M-1: s_{i-1} \neq p_h$$
$$Ah: 1 < h \leq d_{s_{i-1}}: s_{i-1} \neq p_{h-M}$$
$$Ah: 1 < h \leq d_{s_{i-1}}: \sim P(i+h-M)$$

Ниже приводится программа с упрощенной стратегией Боуера — Мура, построенная так же, как и предыдущая программа с КМП-алгоритмом. Обратите внимание на такую деталь: во внутреннем цикле используется цикл с REPEAT, где перед сравнением s и p увеличиваются значения k и j. Это позволяет исключить в индексных выражениях составляющую —1.

Анализ поиска по Боуеру и Муру. В первоначальной публикации [1.9] алгоритма приводится детальный анализ его производительности. Его замечатель-

(1.60)

```
MODULE BM;
 FROM InOut IMPORT
  OpenInput, CloseInput, Read, Write, WriteLn, Done;

 CONST Mmax = 100; Nmax = 10000;

 VAR i, j, k, i0, M, N: INTEGER;
   ch: CHAR;
   p: ARRAY [0 .. Mmax-1] OF CHAR;  (* образ *)
   s: ARRAY [0 .. Nmax-1] OF CHAR;  (*строка *)
   d: ARRAY [0C .. 177C] OF INTEGER;

BEGIN OpenInput("TEXT"); N := 0; Read(ch);
 WHILE Done DO
  Write(ch); s[N] := ch; N := N+1; Read(ch)
 END;
 CloseInput;
 LOOP WriteLn; Write(">"); M := 0; Read(ch);
  WHILE ch > " " DO
   Write(ch); p[M] := ch; M := M+1; Read(ch)
  END;
  WriteLn;
  IF ch = 33C THEN EXIT END;
  FOR ch := 0C TO 177C DO d[ch] := M END;
  FOR j := 0 TO M-2 DO d[p[j]] := M-j-1 END;

  i := M; i0 := 0;
  REPEAT
   WHILE i0 < i DO Write(s[i0]); i0 := i0+1 END;
   j := M; k := i;
   REPEAT k := k-1; j := j-1
   UNTIL (j < 0) OR (p[j] # s[i]);
   i := i + d[s[i-1]]
  UNTIL (j < 0) OR (i >= N);
  IF j < 0 THEN Write("!") END
 END
END BM.
```

Прогр. 1.3. Поиск упрощенным методом БМ.

ное свойство в том, что почти всегда, кроме специально построенных примеров, он требует значительно меньше N сравнений. В самых же благоприятных обстоятельствах, когда последний символ образа всегда попадает на несовпадающий символ текста, число сравнений равно N/M.

Авторы приводят и несколько соображений по поводу дальнейших усовершенствований алгоритма.

Одно из них — объединить приведенную только что стратегию, обеспечивающую большие сдвиги в случае *несовпадения*, со стратегией Кнута, Морриса и Пратта, допускающей «ощутимые» сдвиги при обнаружении совпадения (частичного). Такой метод требует двух таблиц, получаемых при предтрансляции: d1 — только что упомянутая таблица, а d2 — таблица, соответствующая КМП-алгоритму. Из двух сдвигов выбирается больший, причем и тот и другой «говорят», что никакой меньший сдвиг не может привести к совпадению. От дальнейшего обсуждения этого предмета мы воздержимся, поскольку дополнительное усложнение формирования таблиц и самого поиска, кажется, не оправдывает видимого выигрыша в производительности. Фактические дополнительные расходы будут высокими и неизвестно, приведут ли все эти ухищрения к выигрышу или проигрышу.

УПРАЖНЕНИЯ

1.1. Обозначим мощности стандартных типов INTEGER, REAL и CHAR через c_I, c_R и c_{CH}. Каковы мощности следующих приведенных в данной главе в качестве примеров типов: sex, BOOLEAN, weekday, Petter, digit, officer, row, alfa, complex, date, person, coordinate, tapestatus?

1.2. Как бы вы представили переменные перечисленных в упр. 1.1 типов:

а) в памяти вашей машины?

б) на Фортране?

в) на вашем любимом языке программирования?

1.3. Какими последовательностями команд на вашей машине представляются:

а) операции считывания и записи элементов упакованных записей или массивов?

б) операции над массивами, включая проверку на присутствие во множестве?

1.4. Можно ли во время выполнения контролировать корректность использования записей с вариантами? А во время трансляции можно?

1.5. Почему некоторые данные определяются как последовательности, а не как массивы?

1.6. Имеется железнодорожное расписание, содержащее список ежедневных рейсов поездов на нескольких линиях. Найдите представление этих данных с помощью массивов, записей или последовательностей, облегчающее поиск времени отправления или прибытия, если заданы станция и направление.

1.7. Пусть задан в виде последовательности некоторый текст и есть два списка из нескольких слов в виде двух массивов

A и B. Предположим слова — небольшие массивы символов, их размер невелик и фиксирован. Напишите программу, преобразующую текст T в текст S, путем замены каждого вхождения слова A_i на соответствующее слово B_i.

1.8. Сравните три следующие версии программы двоичного поиска с (1.43). Какая из трех программ верная? Определите соответствующие инварианты. Какая из версий более эффективна? В программе используются такие переменные (константа $N > 0$):

```
VAR i, j, k, x: INTEGER;
   a: ARRAY[1 .. N] OF INTEGER;
```

Программа A:

```
i := 1; j := N;
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 := 1; j := N;
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 := 1; j := N;
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 при $a_k = x$, если элемент существует, и при $a_k \neq x$, если элемента со значением x нет.

1.9. Компания с целью определения спроса на свою продукцию организует некоторый опрос. Продукция — пластинки и записи шлягеров, наиболее популярные шлягеры будут переданы по радио. Все опрашиваемые делятся на четыре группы в соответствии с полом и возрастом (скажем, моложе 20 и старше 20). Каждый опрашиваемый должен назвать пять песен. Песни идентифицируются номерами от 1 до N (пусть N = 30). Результаты опроса кодируются последовательностью символов и образуют

последовательность. Для ее чтения можно пользоваться процедурами *Read* и *ReadInt*. Данные относятся к таким типам:

```
TYPE hit = [0 .. N-1];
     sex = (male, female);
     reponse =
       RECORD name, firstname: alfa;
              s: sex;
              age: INTEGER,
              choice: ARRAY [0 .. 4] OF hit
       END;
VAR poll: Sequence
```

Файл с данными обрабатывается программой, которая должна вычислять:

1. Список песен в порядке их популярности. Каждая строка содержит номер песни и число упоминаний ее при опросе. Песни, которые ни разу не упоминались, в список не включаются.
2. Четыре списка (в соответствии с группами) с именами и фамилиями всех тех ответивших, которые поставили на первое место одну из трех наиболее популярных песен.

Все пять списков должны сопровождаться соответствующими заголовками.

ЛИТЕРАТУРА

[1.1] Dahl O., Dijkstra E., Hoare C. Structured Programming. Academic Press, 1972. [Имеется перевод: Дал О., Дейкстра Э., Хоар К. Структурное программирование. — М.: Мир, 1975.]

[1.2] Hoare C. A. R. Notes on data structuring; in Structured Programming. Dahl O., Dijkstra E., Hoare C. См. [1.1].

[1.3] Jensen K., Wirth N. Pascal User Manual and Report. Berlin: Springer-Verlag, 1974. [Имеется перевод: Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1982.]

[1.4] Wirth N. Program development by stepwise refinement. *Comm. ACM.* 14, No. 4, (1971), 221—227.

[1.5] Wirth N. Programming in Modula-2. Berlin, New York: Springer-Verlag, 1982. [Имеется перевод: Вирт Н. Программирование на языке Модула-2. — М.: Мир, 1987.]

[1.6] Wirth N. On composition of well-structured programs. *Computing Surveys*, 6, No. 4, (1974), 247—259.

[1.7] Hoare C. A. R. The Monitor: A operating systems structuring concept. *Comm. ACM.*

[1.8] Knuth D. E., Morris J. H., Pratt V. R. Fast pattern matching in strings. *SIAM J. Comput.*, 6, 2, (June 1977), 323—349.

[1.9] Boyer R. S., Moore J. S. A fast string searching algorithm. *Comm. ACM.* 20, 10, (Oct. 1977), 762—772.

2. СОРТИРОВКА

2.1. ВВЕДЕНИЕ

Основное назначение данной главы — дать побольше примеров использования структур данных, введенных в предыдущей главе, и продемонстрировать, насколько глубоко влияет выбор той или иной структуры на алгоритмы, решающие поставленную задачу. Сортировка к тому же еще и сама достаточно хороший пример задачи, которую можно решать с помощью многих различных алгоритмов. Каждый из них имеет и свои достоинства, и свои недостатки, и выбирать алгоритмы нужно, исходя из конкретной постановки задачи.

В общем сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки — облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Мы встречаемся с отсортированными объектами в телефонных книгах, в списках подоходных налогов, в оглавлениях книг, в библиотеках, в словарях, на складах — почти везде, где нужно искать хранимые объекты. Даже малышей учат держать свои вещи «в порядке», и они уже сталкиваются с некоторыми видами сортировок задолго до того, как познакомятся с азами арифметики *).

Таким образом, разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Что может легче сортироваться, чем данные? Тем не менее наш первоначальный интерес к сортировке

*) Между сортировкой и арифметикой нет никакой видимой связи, да, по-видимому, и невидимой нет. Сортировка — некий «первичный» процесс человеческой деятельности. — *Прим. перев.*

следующим образом:

```
TYPE item = RECORD key: INTEGER;
              (* здесь описаны другие компоненты *)
            END
```
(2.2)

Под «другими компонентами» подразумеваются данные, по существу относящиеся к сортируемым элементам, а ключ просто идентифицирует каждый элемент. Говоря об алгоритмах сортировки, мы будем обращать внимание лишь на компоненту-ключ, другие же компоненты можно даже и не определять. Поэтому из наших дальнейших рассмотрений вся сопутствующая информация опускается, и мы считаем, что тип элемента определен как INTEGER. Выбор INTEGER до некоторой степени произволен. Можно было взять и другой тип, на котором определяется общее отношение порядка.

Метод сортировки называется *устойчивым*, если в процессе сортировки относительное расположение элементов с равными ключами не изменяется. Устойчивость сортировки часто бывает желательной, если речь идет об элементах, уже упорядоченных (отсортированных) по некоторым вторичным ключам (т. е. свойствам), не влияющим на основной ключ.

Данную главу не следует рассматривать как исчерпывающий обзор методов сортировки. Наоборот, мы подробно рассматриваем лишь некоторые, отобранные, специфические приемы. За подробным обсуждением сортировок заинтересованный читатель может обратиться к великолепному и всеобъемлющему исследованию Д. Кнута [2.7] (см. также [2.10]).

2.2. СОРТИРОВКА МАССИВОВ

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться *на том же месте*, т. е. методы, в которых элементы из массива a передаются в результирующий массив b, представляют существенно меньший интерес. Ограни-

чив критерием экономии памяти наш выбор нужного метода среди многих возможных, мы будем сначала классифицировать методы по их экономичности, т. е. по времени их работы. Хорошей мерой эффективности может быть C — число необходимых сравнений ключей и M — число пересылок (перестановок) элементов. Эти числа суть функции от n — числа сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка $n * \log n$ сравнений, мы сначала разберем несколько простых и очевидных методов, их называют *прямыми*, где требуется порядка n^2 сравнений ключей. Начинать разбор с прямых методов, не трогая быстрых алгоритмов, нас заставляют такие причины:

1. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок.

2. Программы этих методов легко понимать, и они коротки. Напомним, что сами программы также занимают память.

3. Усложненные методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых n прямые методы оказываются быстрее, хотя при больших n их использовать, конечно, не следует.

Методы сортировки «*на том же месте*» можно разбить в соответствии с определяющими их принципами на три основные категории:

1. Сортировки с помощью включения (by insertion).

2. Сортировки с помощью выделения (by selection).

3. Сортировки с помощью обменов (by exchange).

Теперь мы исследуем эти принципы и сравним их. Все программы оперируют переменной *a*, именно здесь хранятся переставляемые на месте элементы, и ссылаются на типы *item* (2.2) и *index*, определяемый следующим образом:

```
TYPE index = INTEGER;
VAR a: ARRAY[1 .. n] OF item                    (2.3)
```

2.2.1. Сортировка с помощью прямого включения

Такой метод широко используется при игре в карты. Элементы (карты) мысленно делятся на уже «готовую» последовательность $a_1, \ldots, a_{i-1}$ и исходную последовательность. При каждом шаге, начиная с $i = 2$ и увеличивая i каждый раз на единицу, из исходной последовательности извлекается i-й элемент и перекладывается в готовую последовательность, при этом он вставляется на нужное место.

Таблица 2.1. Пример сортировки с помощью прямого включения

Начальные ключи	44	55	12	42	94	18	06	67
i=2	44	55	12	42	94	18	06	67
i=3	12	44	55	42	94	18	06	67
i=4	12	42	44	55	94	18	06	67
i=5	12	42	44	55	94	18	06	67
i=6	12	18	42	44	55	94	06	67
i=7	06	12	18	42	44	55	94	67
i=8	06	12	18	42	44	55	67	94

В табл. 2.1 показан в качестве примера процесс сортировки с помощью включения восьми случайно выбранных чисел. Алгоритм этой сортировки таков:

```
FOR i := 2 TO n DO
  x := a[i];
  включение x на соответствующее место среди a[1]... a[i]
END
```

В реальном процессе поиска подходящего места удобно, чередуя сравнения и движения по последовательности, как бы просеивать x, т. е. x сравнивается с очередным элементом a_j, а затем либо x вставляется на свободное место, либо a_j сдвигается (передается) вправо и процесс «уходит» влево. Обратите внимание, что процесс просеивания может закончиться при выполнении одного из двух следующих различных условий:

1. Найден элемент a_j с ключом, меньшим чем ключ у x.

2. Достигнут левый конец готовой последовательности.

```
PROCEDURE StraightInsertion;
  VAR i, j: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    x := a[i]; a[0] := x; j := i;
    WHILE x < a[j-1] DO a[j] := a[j-1]; j := j-1 END;
    a[j] := x
  END
END StraightInsertion
```

Прогр. 2.1. Сортировка с помощью прямого включения.

Такой типичный случай повторяющегося процесса с двумя условиями окончания позволяет нам воспользоваться хорошо известным приемом «барьера» (sentinel). Здесь его легко применить, поставив барьер a_0 со значением x. (Заметим, что для этого необходимо расширить диапазон индекса в описании переменной *a* до 0 ... n.) Полный алгоритм приводится в прогр. 2.1 *).

Анализ метода прямого включения. Число сравнений ключей (C_i) при i-м просеивании самое большее равно i — 1, самое меньшее — 1; если предположить, что все перестановки из n ключей равновероятны, то среднее число сравнений — i/2. Число же пересылок (присваиваний элементов) M_i равно $C_i + 2$ (включая барьер). Поэтому общее число сравнений и число пересылок таковы:

$$C_{min} = n-1 \qquad M_{min} = 3*(n-1) \qquad (2.4)$$
$$C_{ave} = (n^2 + n - 2)/4 \qquad M_{ave} = (n^2 + 9n - 10)/4$$
$$C_{max} = (n^2 + n - 4)/4 \qquad M_{max} = (n^2 + 3n - 4)/2$$

Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие же оценки — когда они первоначально расположены в обратном порядке. В некотором смысле сортировка с помощью включений демонстри-

*) Стоит отметить, что приведенная программа не совсем соответствует естественному ходу мышления при ее построении. Естественным кажется вести поиск не влево от очередного просматриваемого элемента, а с самого начала готовой последовательности. При этом и барьер специально ставить нет нужды. Попробуйте составить такую программу! — *Прим. перев.*

```
PROCEDURE BinaryInsertion;
  VAR i, j, m, L, R: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    x := a[i]; L := 1; 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
```

Прогр. 2.2. Сортировка методом деления пополам.

рует истинно естественное поведение. Ясно, что приведенный алгоритм описывает процесс устойчивой сортировки: порядок элементов с равными ключами при нем остается неизменным.

Алгоритм с прямыми включениями можно легко улучшить, если обратить внимание на то, что готовая последовательность, в которую надо вставить новый элемент, сама уже упорядочена. Естественно остановиться на двоичном поиске, при котором делается попытка сравнения с серединой готовой последовательности, а затем процесс деления пополам идет до тех пор, пока не будет найдена точка включения. Такой модифицированный алгоритм сортировки называется *методом с двоичным включением* (binary insertion), и он представлен как прогр. 2.2.

Анализ двоичного включения. Место включения обнаружено, если L = R. Таким образом, в конце поиска интервал должен быть единичной длины; значит, деление его пополам производится i log i раз. Таким образом,

$$C = \mathrm{Si}:\ 1 \leqslant i \leqslant n:\ \lceil \log i \rceil$$

Аппроксимируем эту сумму интегралом

$$\mathrm{Integral}(1:n)\ \log x\, dx = n * (\log n - c) + c \qquad (2.5)$$

где $c = \log e = 1/\ln 2 = 1.4426\ldots$. Число сравнений фактически не зависит от начального порядка элементов. Однако из-за того, что при делении, фигурирую-

щем при разбиении интервала поиска пополам, происходит отбрасывание дробной части, истинное число сравнений может оказаться на единицу больше. Все это приводит к тому, что в нижней части последовательности место включения отыскивается в среднем несколько быстрее, чем в верхней части, поэтому предпочтительна ситуация, в которой элементы чуточку упорядочены. Фактически, если в исходном состоянии элементы расположены в обратном порядке, потребуется минимальное число сравнений, а если они уже упорядочены — максимальное число. Следовательно, можно говорить о неестественном поведении алгоритма поиска.

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

К несчастью, улучшения, порожденные введением двоичного поиска, касаются лишь числа сравнения, а не числа необходимых подвижек чисел. Поскольку движение элемента, т. е. самого ключа, и связанной с ним информации занимает значительно больше времени, чем сравнение двух ключей *), то фактически улучшения не столь уж существенны, ведь важный член M так и продолжает оставаться порядка n^2. И на самом деле, сортировка уже отсортированного массива потребует больше времени, чем в случае последовательной сортировки с прямыми включениями.

Этот пример показывает, что «очевидные улучшения» часто дают не столь уж большой выигрыш, как это кажется на первый взгляд, а в некоторых случаях (случающихся на самом деле) эти «улучшения» могут фактически привести к ухудшениям. После всего сказанного сортировка с помощью включений уже не кажется столь удобным методом для цифровых машин: включение одного элемента с последующим сдвигом на одну позицию целой группы элементов не экономно. Остается впечатление, что лучший результат дадут методы, где передвижка, пусть и на

*) Это соображение верно не для всех машин. Дело в том, что сравнение связано с приостановкой «конвейера» процессора до принятия решения, а пересылка этот процесс не нарушает и идет достаточно быстро. Однако это верно для случая быстрых машин. — *Прим. перев.*

большие расстояния, будет связана лишь с одним-единственным элементом.

2.2.2. Сортировка с помощью прямого выбора

Этот прием основан на следующих принципах:

1. Выбирается элемент с наименьшим ключом.
2. Он меняется местами с первым элементом a_1.
3. Затем этот процесс повторяется с оставшимися n — 1 элементами, n — 2 элементами и т. д. до тех пор, пока не останется один, самый большой элемент.

Процесс работы этим методом с теми же восемью ключами, что и в табл. 2.1, приведен в табл. 2.2. Алгоритм формулируется так:

```
FOR i := 1 TO n-1 DO
  присвоить k индекс наименьшего из a[i]... a[n];
  поменять местами a[i] и a[k];
END
```

Такой метод — его называют *прямым выбором* — в некотором смысле противоположен прямому включению. При прямом включении на каждом шаге рассматриваются только *один* очередной элемент исходной последовательности и *все* элементы готовой последовательности, среди которых отыскивается точка включения; при прямом выборе для поиска *одного* элемента с наименьшим ключом просматриваются все элементы исходной последовательности и найденный помещается как очередной элемент в готовую последовательность. Полностью алгоритм прямого выбора приводится в прогр. 2.3.

Таблица 2.2. Пример сортировки с помощью прямого выбора

Начальные ключи	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

```
PROCEDURE StraightSelection;
  VAR i, j, k: index; x: item;
BEGIN
  FOR i := 1 TO n-1 DO
    k := i; x := a[i];
    FOR j := i+1 TO n DO
      IF a[j] < x THEN k := j; x := a[k] END
    END;
    a[k] := a[i]; a[i] := x
  END
END StraightSelection
```

Прогр. 2.3. Сортировка с помощью прямого выбора.

Анализ прямого выбора. Число сравнений ключей (C), очевидно, не зависит от начального порядка ключей. Можно сказать, что в этом смысле поведение этого метода менее естественно, чем поведение прямого включения. Для C имеем

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

Число перестановок минимально

$$M_{min} = 3 * (n - 1) \tag{2.6}$$

в случае изначально упорядоченных ключей и максимально

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

если первоначально ключи располагались в обратном порядке. Для того чтобы определить M_{avg}, мы должны рассуждать так. Алгоритм просматривает массив, сравнивая каждый элемент с только что обнаруженной минимальной величиной; если он меньше первого, то выполняется некоторое присваивание. Вероятность, что второй элемент окажется меньше первого, равна 1/2, с этой же вероятностью происходят присваивания минимуму. Вероятность, что третий элемент окажется меньше первых двух, равна 1/3, а вероятность для четвертого оказаться наименьшим — 1/4 и т. д. Поэтому полное ожидаемое число пересылок равно $H_n - 1$, где H_n — n-е гармоническое число:

$$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \tag{2.7}$$

H_n можно выразить и так:

$$H_n = \ln n + g + 1/2n - 1/12n^2 + \ldots \quad (2.8)$$

где $g = 0.577216 \ldots$ — константа Эйлера. Для достаточно больших n мы можем игнорировать дробные составляющие и поэтому аппроксимировать среднее число присваиваний на i-м просмотре выражением

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

Среднее число пересылок M_{avg} в сортировке с выбором есть сумма F_i с i от 1 до n:

$$M_{avg} = n * (g + 1) + (Si: 1 \leqslant i \leqslant n: \ln i)$$

Вновь аппроксимируя эту сумму дискретных членов интегралом

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

получаем, наконец, приблизительное значение

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

Отсюда можно сделать заключение, что, как правило, алгоритм с прямым выбором предпочтительнее строгого включения. Однако, если ключи в начале упорядочены или почти упорядочены, прямое включение будет оставаться несколько более быстрым.

2.2.3. Сортировка с помощью прямого обмена

Классификация методов сортировки редко бывает осмысленной. Оба разбиравшихся до этого метода можно тоже рассматривать как «обменные» сортировки. В данном же, однако, разделе мы опишем метод, где обмен местами двух элементов представляет собой характернейшую особенность процесса. Изложенный ниже алгоритм прямого обмена основывается на сравнении и смене мест для пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы.

Как и в упоминавшемся методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если мы будем рас-

Таблица 2.3. Пример пузырьковой сортировки

i=1	2	3	4	5	6	7	8
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

сматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу (см. табл. 2.3). Такой метод широко известен под именем *«пузырьковая сортировка»*. В своем простейшем виде он представлен в прогр. 2.4.

Улучшения этого алгоритма напрашиваются сами собой. На примере в табл. 2.3 видно, что три последних прохода не влияют на порядок элементов из-за того, что они уже отсортированы. Очевидный прием улучшения этого алгоритма — запоминать, были или не были перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то алгоритм можно заканчивать. Это улучшение однако, можно опять же улучшить, если запоминать не только сам факт, что обмен имел место, но и положение

```
PROCEDURE BubbleSort;
  VAR i, j: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    FOR j := n 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
```

Прогр. 2.4. Пузырьковая сортировка.

(индекс) последнего обмена. Ясно, что все пары соседних элементов выше этого индекса k уже находятся в желаемом порядке. Поэтому просмотры можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела для i. Внимательный программист заметит тем не менее здесь некоторую своеобразную асимметрию. Один плохо расположенный пузырек на «тяжелом конце» в массиве с обратным порядком будет перемещаться на нужное место в один проход, но плохо расположенный элемент на «легком конце» будет просачиваться на свое нужное место на один шаг при каждом проходе*). Например, массив

12 18 42 44 55 67 94 06

с помощью усовершенствованной «пузырьковой» сортировки можно упорядочить за один просмотр, а для сортировки массива

94 06 12 18 42 44 55 67

требуется семь просмотров. Такая неестественная симметрия наводит на мысль о третьем улучшении: чередовать направление последовательных просмотров. Получающийся при этом алгоритм мы соответственно назовем *«шейкерной» сортировкой* (ShakerSort)**). Таблица 2.4 иллюстрирует сортировку новым способом тех же (табл. 2.3) восьми ключей.

Анализ пузырьковой и шейкерной сортировок. Число сравнений в строго обменном алгоритме

$$C = (n^2 - n)/2 \qquad (2.10)$$

а минимальное, среднее и максимальное число перемещений элементов (присваиваний) равно соответственно

$$M_{min} = 0, \quad M_{avg} = 3 * (n^2 - n)/2, \quad M_{max} = 3 * (n^2 - n)/4 \qquad (2.11)$$

*) Проще было бы сказать так: всплывает пузырек сразу, за один проход, а тонет очень медленно, за один проход на одну позицию. — *Прим. перев.*

**) Напомним, что шейкером называется нечто вроде двух накрывающих друг друга стаканов, в которых встряхиванием вверх-вниз готовят коктейль. — *Прим. перев.*

Таблица 2.4. Пример шейкерной 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

Анализ же улучшенных методов, особенно шейкерной сортировки, довольно сложен. Минимальное число сравнений $C_{min} = n - 1$. Кнут считает, что для улучшенной пузырьковой сортировки среднее число проходов пропорционально $n - k_1 n^{1/2}$, а среднее число сравнений пропорционально $^1/_2(n^2 - n(k_2 + \ln n))$. Обратите, однако, внимание на то, что все перечисленные выше усовершенствования не влияют на число перемещений, они лишь сокращают число излишних двойных проверок. К несчастью, обмен местами двух элементов — чаще всего более дорогостоящая опера-

```
PROCEDURE ShakerSort;
  VAR j, k, L, R: index;  x: item;
BEGIN L := 2; R := n; k := n;
  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
```

Прогр. 2.5. Шейкерная сортировка.

ция, чем сравнение ключей, поэтому наши, вроде очевидные, улучшения дают не такой уж большой выигрыш, как мы интуитивно ожидали.

Такой анализ показывает, что «обменная» сортировка и ее небольшие усовершенствования представляют собой нечто среднее между сортировками с помощью включений и с помощью выбора. Фактически в пузырьковой сортировке нет ничего ценного, кроме ее привлекательного названия (Bubblesort). Шейкерная же сортировка с успехом используется в тех случаях, когда известно, что элементы почти упорядочены — на практике это бывает весьма редко.

Можно показать, что среднее расстояние, на которое должен продвигаться каждый из n элементов во время сортировки, равно n/3 «мест». Эта цифра является целью в поиске улучшений, т. е. в поиске более эффективных методов сортировки. Все строгие приемы сортировки фактически передвигают каждый элемент на всяком элементарном шаге на одну позицию. Поэтому они требуют порядка n^2 таких шагов. Следовательно, в основу любых улучшений должен быть положен принцип перемещения на каждом такте элементов на большие расстояния.

Далее мы рассматриваем три улучшенных метода: по одному для каждого из основных строгих методов сортировки — включения, выбора и обмена.

2.3. УЛУЧШЕННЫЕ МЕТОДЫ СОРТИРОВКИ

2.3.1. Сортировка с помощью включений с уменьшающимися расстояниями

В 1959 г. Д. Шеллом было предложено усовершенствование сортировки с помощью прямого включения. Сам метод объясняется и демонстрируется на нашем стандартном примере (см. табл. 2.5). Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на расстоянии 4. Такой процесс называется четверной сортировкой. В нашем примере восемь элементов и каждая группа состоит точно из двух элементов. После первого прохода элементы перегруппировываются — теперь каждый элемент группы

Таблица 2.5. Сортировка с помощью включений с уменьшающимися расстояниями

44	55	12	42	94	18	06	67

четверная сортировка дает

44	18	06	42	94	55	12	67

двойная сортировка дает

06	18	12	42	44	55	94	67

одинарная сортировка дает

06	12	18	42	44	55	67	94

отстоит от другого на две позиции — и вновь сортируются. Это называется двойной сортировкой. И наконец, на третьем проходе идет обычная или одинарная сортировка.

На первый взгляд можно засомневаться: если необходимо несколько процессов сортировки, причем в каждый включаются все элементы, то не добавят ли они больше работы, чем сэкономят? Однако на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуется сравнительно немного перестановок.

Ясно, что такой метод в результате дает упорядоченный массив, и, конечно же, сразу видно, что каждый проход от предыдущих только выигрывает (так как каждая i-сортировка объединяет две группы, уже отсортированные 2i-сортировкой). Так же очевидно, что расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу. Однако совсем не очевидно, что такой прием «уменьшающихся расстояний» может дать лучшие результаты, если расстояния не будут степенями двойки. Поэтому приводимая программа не ориентирована на некую определенную последовательность расстояний. Все t расстояний обозначаются соответственно h_1, h_2, ..., h_t, для них выполняются условия

$$h_t = 1,\ h_{i+1} < h_i \qquad (2.12)$$

Каждая h-сортировка программируется как сортировка с помощью прямого включения. Причем про-

```
PROCEDURE ShellSort;
  CONST t = 4;
  VAR i, j, k, s: index;
    x: item; m: 1 .. t;
    h: ARRAY [1 .. t] OF INTEGER;
BEGIN h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
  FOR m := 1 TO t DO
    k := h[m]; s := -k; (* место барьера *)
    FOR i := k+1 TO n DO
      x := a[i]; j := i-k;
      IF s = 0 THEN s := -k END;
      s := s+1; a[s] := x;
      WHILE x < a[j] DO a[j+k] := a[j]; j := j-k END;
      a[j+k] := x
    END
  END
END ShellSort
```

Прогр. 2.6. Сортировка Шелла.

стота условия окончания поиска места для включения обеспечивается методом барьеров. Ясно, что каждая из сортировок нуждается в постановке своего собственного барьера и программу для определения его местоположения необходимо делать насколько возможно простой. Поэтому приходится расширять массив не только на одну-единственную компоненту a_0, а на h_1 компонент. Его описание теперь выглядит так:

```
a: ARRAY[-h1 .. n] OF item
```

Сам алгоритм для $t = 4$ описывается процедурой *Shellsort* [2.11] в прогр. 2.6.

Анализ сортировки Шелла. Анализ этого алгоритма поставил несколько весьма трудных математических проблем, многие из которых так еще и не решены. В частности, не известно, какие расстояния дают наилучшие результаты. Но вот удивительный факт: они не должны быть множителями один другого. Это позволяет избежать явления уже очевидного из приведенного выше примера, когда при каждом проходе взаимодействуют две цепочки, которые до этого нигде еще не пересекались. И действительно, желательно, чтобы взаимодействие различных цепочек проходило как можно чаще. Справедлива такая теорема: если k-отсортированную последовательность

i-отсортировать, то она остается k-отсортированной. В работе [2.8] Кнут показывает, что имеет смысл использовать такую последовательность (она записана в обратном порядке): 1, 4, 13, 40, 121, ..., где $h_{k-1} = 3h_k + 1$, $h_t = 1$ и $t = \lfloor \log_3 n \rfloor - 1$. Он рекомендует и другую последовательность: 1, 3, 7, 15, 31, ... где $h_{k-1} = 2h_k + 1$, $h_t = 1$ и $t = \lfloor \log_2 n \rfloor - 1$. Математический анализ показывает, что в последнем случае для сортировки n элементов методом Шелла затраты пропорциональны $n^{1.2}$. Хотя это число значительно лучше n^2, тем не менее мы не ориентируемся в дальнейшем на этот метод, поскольку существуют еще лучшие алгоритмы.

2.3.2. Сортировка с помощью дерева

Метод сортировки с помощью прямого выбора основан на повторяющихся поисках наименьшего ключа среди n элементов, среди оставшихся $n - 1$ элементов и т. д. Обнаружение наименьшего среди n элементов требует — это очевидно — $n - 1$ сравнения, среди $n - 1$ уже нужно $n - 2$ сравнений и т. д. Сумма первых $n - 1$ целых равна $^1/_2(n^2 - n)$. Как же в таком случае можно усовершенствовать упомянутый метод сортировки? Этого можно добиться, только оставляя после каждого прохода больше информации, чем просто идентификация единственного минимального элемента. Например, сделав n/2 сравнений, можно определить в каждой паре ключей меньший. С помощью n/4 сравнений — меньший из пары уже выбранных меньших и т. д. Проделав $n - 1$ сравнений, мы можем построить дерево выбора вроде представленного на рис. 2.3 и идентифицировать его корень как нужный нам наименьший ключ [2.2].

Второй этап сортировки — спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент (дырку) в самом низу, либо на элемент из соседней ветви в промежуточных вершинах (см. рис. 2.4 и 2.5). Элемент, передвинувшийся в корень дерева, вновь будет наименьшим (теперь уже вторым) ключом, и его можно исключить. После n таких шагов дерево ста-

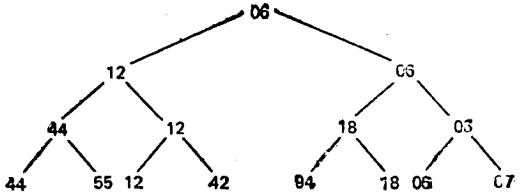


Рис. 2.3. Повторяющиеся выборы среди двух ключей.

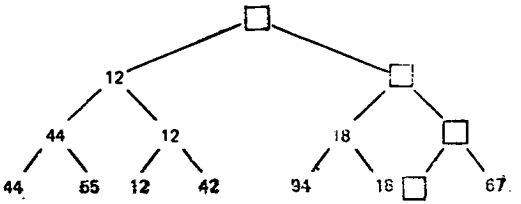


Рис. 2.4. Выбор наименьшего ключа.

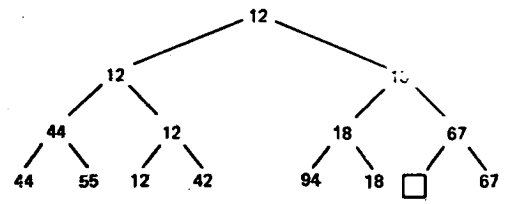


Рис. 2.5. Заполнение дырок.

нет пустым (т. е. в нем останутся только дырки), и процесс сортировки заканчивается. Обратите внимание — на каждом из n шагов выбора требуется только log n сравнений. Поэтому на весь процесс понадобится порядка $n * \log n$ элементарных операций плюс еще n шагов на построение дерева. Это весьма существенное улучшение не только прямого метода, требующего n^2 шагов, но и даже метода Шелла, где нужно $n^{1.2}$ шага. Естественно, сохранение дополнительной информации делает задачу более изощренной, поэтому в сортировке по дереву каждый отдельный шаг усложняется. Ведь в конце концов для сохранения избыточной информации, получаемой при

начальном проходе, создается некоторая древообразная структура. Наша следующая задача — найти приемы эффективной организации этой информации.

Конечно, хотелось бы, в частности, избавиться от дырок, которыми в конечном итоге будет заполнено все дерево и которые порождают много ненужных сравнений. Кроме того, надо было бы поискать такое представление дерева из n элементов, которое требует лишь n единиц памяти, а не $2n-1$, как это было раньше. Этих целей действительно удалось добиться в методе *Heapsort* *), изобретенном Д. Уилльямсом [2.14], где было получено, очевидно, существенное улучшение традиционных сортировок с помощью деревьев. Пирамида определяется как последовательность ключей $h_L, h_{L+1}, \ldots, h_R$, такая, что

$$h_i \leqslant h_{2i} \text{ и } h_i \leqslant h_{2i+1} \text{ для } i = L \ldots R/2. \qquad (2.13)$$

Если любое двоичное дерево рассматривать как массив по схеме на рис. 2.6, то можно говорить, что деревья сортировок на рис. 2.7 и 2.8 суть пирамиды, а элемент h_1, в частности, их наименьший элемент: $h_1 = \min(h_1, h_2, \ldots, h_n)$. Предположим, есть некоторая пирамида с заданными элементами $h_{L+1}, \ldots, h_R$ для некоторых значений L и R и нужно добавить новый элемент x, образуя расширенную пирамиду $h_L, \ldots, h_R$. Возьмем, например, в качестве исходной пирамиду $h_1, \ldots, h_7$, показанную на рис. 2.7, и добавим к ней слева элемент $h_1 = 44$ **). Новая пирамида получается так: сначала x ставится наверх древовидной структуры, а затем он постепенно опускается вниз каждый раз по направлению наименьшего из двух примыкающих к нему элементов, а сам этот элемент передвигается вверх. В приведенном примере значение 44 сначала меняется местами с 06, затем с 12 и

*) Здесь мы оставляем попытки перевести названия соответствующих методов на русский язык, ибо ни уже не более чем собственные имена, хотя в названиях первых упоминавшихся методов еще фигурировал некоторый элемент описания сути самого приема сортировки. — *Прим. перев.*

**) Это несколько противоречит утверждению, что $h_{L+1} \ldots h_R$ — пирамида, но надеемся, сам читатель разберется, что хотел сказать автор. — *Прим. перев.*

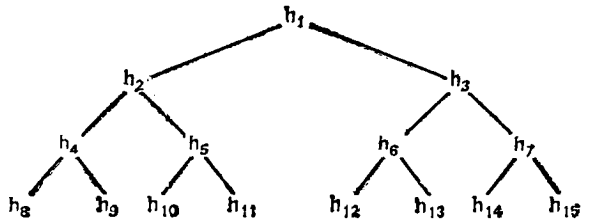


Рис. 2.6. Массив h, представленный в виде двоичного дерева.

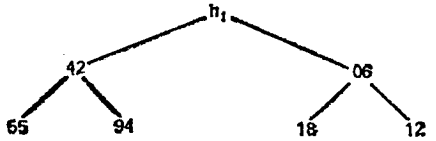


Рис. 2.7. Пирамида из семи элементов.

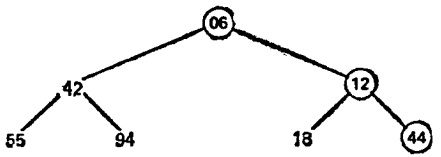


Рис. 2.8. Просеивание ключа 44 через пирамиду.

в результате образуется дерево, представленное на рис. 2.8. Теперь мы сформулируем этот сдвигающий алгоритм так: i, j — пара индексов, фиксирующих элементы, меняющиеся на каждом шаге местами. Читателю остается лишь убедиться самому, что предложенный метод сдвигов действительно сохраняет неизменным условия (2.13), определяющие пирамиду.

Р. Флойдом был предложен некий «лаконичный» способ построения пирамиды «на том же месте». Его процедура сдвига представлена как прогр. 2.7. Здесь $h_1 \ldots h_n$ — некий массив, причем $h_m \ldots h_n$ ($m = (n \text{ DIV } 2) + 1$) уже образуют пирамиду, поскольку индексов i, j, удовлетворяющих отношениям $j = 2i$ (или $j = 2i + 1$), просто не существует. Эти элементы образуют как бы нижний слой соответствующего двоичного дерева (см. рис. 2.6), для них никакой

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

Прогр. 2.7. Sift.

упорядоченности не требуется. Теперь пирамида расширяется влево; каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент. Табл. 2.6 иллюстрирует весь этот процесс, а получающаяся пирамида показана на рис. 2.6.

Следовательно, процесс формирования пирамиды из n элементов $h_1 \ldots h_n$ на том же самом месте описывается так:

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

Для того чтобы получить не только частичную, но и полную упорядоченность среди элементов, нужно проделать n сдвигающих шагов, причем после каждого шага на вершину дерева выталкивается очередной (наименьший) элемент. И вновь возникает вопрос: где хранить «всплывающие» верхние элементы и можно ли или нельзя проводить обращение на том же месте? Существует, конечно, такой выход: каждый раз брать последнюю компоненту пирамиды (скажем, это будет x), прятать верхний элемент пирамиды в освободившемся теперь месте, а x сдвигать в нужное место. В табл. 2.7 приведены необходимые в этом слу-

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
05	42	12	55	94	18	44	67

Таблица 2.7. Пример процесса сортировки с помощью 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

чае n — 1 шагов. Сам процесс описывается с помощью процедуры *sift* (прогр. 2.7) таким образом:

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

Пример из табл. 2.7 показывает, что получающийся порядок фактически является обратным. Однако это можно легко исправить, изменив направление «упорядочивающего отношения» в процедуре *sift*. В конце концов получаем процедуру Heapsort (прогр. 2.8),

```
PROCEDURE HeapSort;
  VAR L, R: index; x: item;

  PROCEDURE sift(L, R: index);
    VAR i, j: index; x: item;
  BEGIN i := L; j := 2*L; x := a[L];
    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;
      IF (j < R) & (a[j] < a[j+1]) THEN j := j+1 END
    END
  END sift;

BEGIN L := (n DIV 2) + 1; R := n;
  WHILE L > 1 DO L := L-1; sift(L, R) END;
  WHILE R > 1 DO
    x := a[1]; a[1] := a[R]; a[R] := x;
    R := R-1; sift(L, R)
  END
END HeapSort
```

Прогр. 2.8. Heapsort.

Анализ Heapsort. На первый взгляд вовсе не очевидно, что такой метод сортировки дает хорошие результаты. Ведь в конце концов большие элементы, прежде чем попадут на свое место в правой части, сначала сдвигаются влево. И действительно, процедуру не рекомендуется применять для небольшого, вроде нашего примера, числа элементов. Для больших же n Heapsort очень эффективна; чем больше n, тем лучше она работает. Она даже становится сравнимой с сортировкой Шелла.

В худшем случае нужно n/2 сдвигающих шагов, они сдвигают элементы на log(n/2), log(n/2 — 1), ..., log(n — 1) позиций (логарифм (по основанию 2) «обрубается» до следующего меньшего целого). Следовательно, фаза сортировки требует n — 1 сдвигов с самое большое log(n — 1), log(n — 2), ..., 1 перемещениями. Кроме того, нужно еще n — 1 перемещений для просачивания сдвинутого элемента на некоторое расстояние вправо. Эти соображения показывают, что даже в самом плохом из возможных случаев Heapsort потребует n * log n шагов. Великолепная производительность в таких плохих случаях — одно из привлекательных свойств Heapsort.

Совсем не ясно, когда следует ожидать наихудшей (или наилучшей) производительности. Но вообще-то кажется, что Heapsort «любит» начальные последовательности, в которых элементы более или менее отсортированы в обратном порядке. Поэтому ее поведение несколько неестественно. Если мы имеем дело с обратным порядком, то фаза порождения пирамиды не требует каких-либо перемещений. Среднее число перемещений приблизительно равно n/2 * log(n), причем отклонения от этого значения относительно невелики.

2.3.3. Сортировка с помощью разделения

Разобравшись в двух усовершенствованных методах сортировки, построенных на принципах включения и выбора, мы теперь коснемся третьего улучшенного метода, основанного на обмене. Если учесть, что пузырьковая сортировка в среднем была самой неэф-

фективной из всех трех алгоритмов прямой (строгой) сортировки, то следует ожидать относительно существенного улучшения. И все же это выглядит как некий сюрприз: улучшение метода, основанного на обмене, о котором мы будем сейчас говорить, оказывается, приводит к самому лучшему из известных в данный момент методу сортировки для массивов. Его производительность столь впечатляюща, что изобретатель Ч. Хоар [2.5] и [2.6] даже назвал метод *быстрой сортировкой* (Quicksort).

В Quicksort исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие расстояния. Предположим, у нас есть n элементов, расположенных по ключам в обратном порядке. Их можно отсортировать за n/2 обменов, сначала поменять местами самый левый с самым правым, а затем последовательно двигаться с двух сторон. Конечно, это возможно только в том случае, когда мы знаем, что порядок действительно обратный. Но из этого примера можно извлечь и нечто действительно поучительное.

Давайте, попытаемся воспользоваться таким алгоритмом: выберем наугад какой-либо элемент (назовем его x) и будем просматривать слева наш массив до тех пор, пока не обнаружим элемент $a_i > x$, затем будем просматривать массив справа, пока не встретим $a_j < x$. Теперь поменяем местами эти два элемента и продолжим наш процесс просмотра и обмена, пока оба просмотра не встретятся где-то в середине массива. В результате массив окажется разбитым на левую часть, с ключами меньше (или равными) x, и правую — с ключами больше (или равными) x. Теперь этот процесс разделения представим в виде процедуры (прогр. 2.9). Обратите внимание, что вместо отношений $>$ и $<$ используются $\geqslant$ и $\leqslant$, а в заголовке цикла с WHILE — их отрицания $<$и$>$. При таких изменениях x выступает в роли барьера для того и другого просмотра. Если взять в качестве x для сравнения средний ключ 42, то в массиве ключей

44 55 12 42 94 06 18 67

```
PROCEDURE partition;
  VAR w, x: item;
BEGIN i := 1; j := n;
  случайно выбрать 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
```

Прогр. 2.9. Сортировка с помощью разделения.

для разделения понадобятся два обмена: $18 \leftrightarrow 44$ и $6 \leftrightarrow 55$

18 06 12 42 94 55 44 67

последние значения индексов таковы: $i = 5$, а $j = 3$. Ключи $a_1 \ldots a_{i-1}$ меньше или равны ключу $x = 42$, а ключи $a_{j+1} \ldots a_n$ больше или равны x. Следовательно, существует две части, а именно

$$\begin{aligned} &Ak: 1 \leqslant k < i : a_k \leqslant x \\ &Ak: j < k \leqslant n : x \leqslant a_k \end{aligned} \qquad (2.14)$$

Описанный алгоритм очень прост и эффективен, поскольку главные сравниваемые величины i, j и x можно хранить во время просмотра в быстрых регистрах машины. Однако он может оказаться и неудачным, что, например, происходит в случае n идентичных ключей: для разделения нужно $n/2$ обменов. Этих вовсе необязательных обменов можно избежать, если операторы просмотра заменить на такие:

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

Однако в этом случае выбранный элемент x, находящийся среди компонент массива, уже не работает как барьер для двух просмотров. В результате просмотры массива со всеми идентичными ключами приведут, если только не использовать более сложные условия их окончания, к переходу через границы массива. Про-

стота условий, употребленных в прогр. 2.9, вполне оправдывает те дополнительные обмены, которые происходят в среднем относительно редко. Можно еще немного сэкономить, если изменить заголовок, управляющий самим обменом: от $i \leqslant j$ перейти к $i < j$. Однако это изменение не должно касаться двух операторов: $i := i + 1$, $j := j - 1$. Поэтому для них потребуется отдельный условный оператор. Убедиться в правильности алгоритма разделения можно, удостоверившись, что отношения (2.14) представляют собой инварианты оператора цикла с REPEAT. Вначале при $i = 1$ и $j = n$ их истинность тривиальна, а при выходе $c_i > j$ они дают как раз желаемый результат.

Теперь напомним, что наша цель — не только провести разделение на части исходного массива элементов, но и отсортировать его. Сортировку от разделения отделяет, однако, лишь небольшой шаг: нужно применить этот процесс к получившимся двум частям, затем к частям частей, и так до тех пор, пока каждая из частей не будет состоять из одного-единственного элемента. Эти действия описываются программой 2.10.

Процедура *sort* рекурсивно обращается сама к себе. Рекурсии в алгоритмах — это очень мощный механизм, его мы будем рассматривать в гл. 3.

```
PROCEDURE QuickSort;

  PROCEDURE sort(L, R: index);
    VAR i, j: index; 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;

  BEGIN sort(1, n)
  END QuickSort
```

Прогр. 2.10. Quicksort.

Во многих «старых» языках программирования от рекурсий по некоторым техническим причинам отказывались. Поэтому мы теперь покажем, как тот же алгоритм можно выразить и нерекурсивной процедурой. Решение, очевидно, заключается в том, что рекурсию нужно заменить итерацией. При этом понадобятся, конечно, некоторые дополнительные операции по сохранению нужной информации.

Суть итеративного решения заключается во введении списка требуемых разделений, т. е. разделений, которые необходимо провести. На каждом этапе возникают две задачи по разделению. И только к одной из них мы можем непосредственно сразу же приступить в очередной итерации, другая же заносится в упомянутый список. При этом, конечно, существенно, что требования из списка выполняются несколько специфическим образом, а именно в обратном порядке. Следовательно, первое из перечисленных требований выполняется последним, и наоборот. В приводимой нерекурсивной версии Quicksort каждое требование задается просто левым и правым индексами — это границы части, требующей дальнейшего разделения. Таким образом, мы вводим переменную-массив под именем *stack* и индекс *s*, указывающий на самую последнюю строку в стеке (см. прогр. 2.11). О том, как выбрать подходящий размер стека M, речь пойдет при анализе работы Quicksort.

Анализ Quicksort. Для исследования производительности Quicksort сначала необходимо разобраться, как идет процесс разделения. Выбрав некоторое граничное значение x, мы затем проходим целиком по всему массиву. Следовательно, при этом выполняется точно n сравнений. Число же обменов можно определить из следующих вероятностных соображений. При заданной границе значений x ожидаемое число операций обмена равно числу элементов в левой части разделяемой последовательности, т. е. n — 1, умноженному на вероятность того, что при обмене каждый такой элемент попадает на свое место. Обмен происходит, если этот элемент перед этим находился в правой части. Вероятность этого равна $(n-(x-1))/n$. Поэтому ожидаемое число обменов есть

```
PROCEDURE NonRecursiveQuickSort;
  CONST M = 12;
  VAR i, j, L, R: index;  x, w: item;
    s: [0 .. M];
    stack: ARRAY [1 .. M] OF RECORD L, R: index END ;
BEGIN s := 1; stack[1].L := 1; stack[s].R := n;
  REPEAT (* выбор из стека последнего запроса *)
    L := stack[s].L; R := stack[s].R; s := s-1;
    REPEAT (* разделение 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 (* запись в стек запроса из правой части *)
        s := s+1; stack[s].L := i; stack[s].R := R
      END ;
      R := j (* теперь L и R ограничивают левую часть *)
    UNTIL L >= R
  UNTIL s = 0
END NonRecursiveQuickSort
```

Прогр. 2.11. Нерекурсивная версия Quicksort.

среднее этих ожидаемых значений для всех возможных границ x.

$$M = [Sx: 1 \le x \le n : (x-1)*(n-(x-1))/n]/n$$
$$= [Su: 0 \le u \le n-1 : u*(n-u)]/n^2$$
$$= n*(n-1)/2n - (2n^2 - 3n + 1)/6n = (n - 1/n)/6 \qquad (2.15)$$

Представим себе, что мы счастливчики и нам всегда удается выбрать в качестве границы медиану, в этом случае каждый процесс разделения расщепляет массив на две половины и для сортировки требуется всего log n проходов. В результате общее число сравнений равно $n * \log n$, а общее число обменов — $n * \log(n)/6$. Нельзя, конечно, ожидать, что мы каждый раз будем выбирать медиану. Вероятность этого составляет только 1/n. Удивительный, однако, факт: средняя производительность Quicksort при случайном выборе границы отличается от упомянутого оптимального варианта лишь коэффициентом $2 * \ln(2)$.

Как бы то ни было, но Quicksort присущи и некоторые недостатки. Главный из них — недостаточно высокая производительность при небольших n, впрочем этим грешат все усовершенствованные методы. Но перед другими усовершенствованными методами этот имеет то преимущество, что для обработки небольших частей в него можно легко включить какой-либо из прямых методов сортировки. Это особенно удобно делать в случае рекурсивной версии программы.

Остается все еще вопрос о самом плохом случае. Как тогда будет работать Quicksort? К несчастью, ответ на этот вопрос неутешителен и демонстрирует одно неприятное свойство Quicksort. Разберем, скажем, тот несчастный случай, когда каждый раз для сравнения выбирается наибольшее из всех значений в указанной части. Тогда на каждом этапе сегмент из n элементов будет расщепляться на левую часть, состоящую из $n-1$ элементов, и правую, состоящую из одного-единственного элемента. В результате потребуется n (а не $\log n$) разделений и наихудшая производительность метода будет порядка n^2.

Явно видно, что главное заключается в выборе элемента для сравнения — x. В нашей редакции им становится средний элемент. Заметим, однако, что почти с тем же успехом можно выбирать первый или последний. В этих случаях хуже всего будет, если массив окажется первоначально уже упорядочен, ведь Quicksort определенно «не любит» такую тривиальную работу и предпочитает иметь дело с неупорядоченными массивами. Выбирая средний элемент, мы как бы затушевываем эту странную особенность Quicksort, поскольку в этом случае первоначально упорядоченный массив будет уже оптимальным вариантом. Таким образом, фактически средняя производительность при выборе среднего элемента чуточку улучшается. Сам Хоар предполагает, что x надо выбирать случайно, а для небольших выборок, вроде всего трех ключей, останавливаться на медиане [2.12, 2.13]. Такой разумный выбор мало влияет на среднюю производительность Quicksort, но зато значительно ее улучшает (в наихудших случаях). В некотором смысле быстрая сортировка напоминает азартную

игру: всегда следует учитывать, сколько можно проиграть в случае невезения.

Из этих соображений можно извлечь один важный урок, касающийся непосредственно программирования. Что же следует из упомянутого случая самого плохого поведения программы 2.11? Мы ее построили так, что каждое разделение заканчивается образованием правой части, состоящей лишь из единственного элемента, и требование последующей сортировки этой части заносится в стек. Следовательно, максимальное число требований, т. е. общий размер стека требований, равно n. Конечно, как правило, эта граница не достигается. (Заметим, что с рекурсивной версией программы дело обстоит не лучше, а даже хуже, поскольку системы, допускающие рекурсивные обращения к процедурам, автоматически сохраняют значения локальных переменных и параметров всех активаций таких процедур. Для этого используют некоторый неявный стек.)

Выход из положения заключается в том, что в стек надо прятать требование сортировки для более длинной части и сразу же продолжать разделение меньшей части. В этом случае размер стека M можно ограничить числом log n. Все изменения, которые нужно внести в программу 2.11, сосредоточены в разделе, устанавливающем новые требования. Теперь он выглядит так:

```
IF j-L<R-1 THEN
  IF i<R THEN (*запись в стек запроса на сортировку правой части*)
    s:=s+1; stack[s].L:=i; stack[s].R:=R
  END;
  R:=j (*продолжение сортировки левой части*)
ELSE                                                    (2.16)
  IF L<j THEN (*запись в стек запроса на сортировку левой части*)
    s:=s+1; stack[s].L:=L; srack[s].R:=j
  END;
  L:=i (*продолжение сортировки правой части*)
END
```

2.3.4. Нахождение медианы

Медианой для n элементов называется элемент, меньший (или равный) половине из n элементов и больший (или равный) другой половине из n эле-

ментов. Например, медиана для элементов

16 12 99 95 18 87 10

равна 18. Задача поиска медианы тесно связана с проблемой сортировки, поскольку очевидный метод определения медианы заключается в том, чтобы отсортировать n элементов, а затем выбрать средний элемент. Однако разделение, выполняемое программой 2.9, позволяет потенциально отыскивать медиану значительно быстрее. Этот прием можно легко обобщить и для поиска среди n элементов k-го наименьшего числа. В этом случае поиск медианы — просто частный случай $k = n/2$. Алгоритм, предложенный Ч. Хоаром [2.4], работает следующим образом. Сначала применяется операция разделения из Quicksort с $L = 1$ и $R = n$, в качестве разделяющего значения x берется a_k. В результате получаем индексы i и j, удовлетворяющие таким условиям:

$$\begin{aligned} &1.\ a_h < x \quad \text{для всех } h < i \\ &2.\ a_h > x \quad \text{для всех } h > j \\ &3.\ i > j \end{aligned} \tag{2.17}$$

При этом мы сталкиваемся с одним из таких трех случаев:

1. Разделяющее значение x было слишком мало, и граница между двумя частями лежит ниже нужной величины k. Процесс разделения повторяется для элементов $a_i \ldots a_R$ (рис. 2.9).

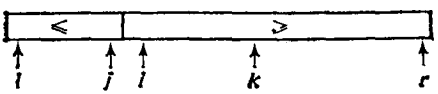


Рис. 2.9. Граница слишком мала.

2. Выбранная граница x была слишком большой. Операции разделения следует повторить для элементов $a_L \ldots a_j$ (рис. 2.10).

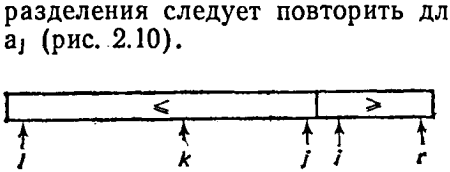


Рис. 2.10. Граница слишком велика

3. $j < k < i$: элемент a_k разделяет массив на две части в нужной пропорции, следовательно, это то, что нужно (рис. 2.11).

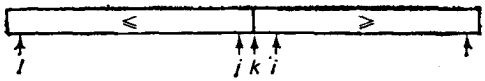


Рис. 2.11. Верная граница.

Процессы разделения повторяются до тех пор, пока не возникнет третий случай. Сами итерации описываются следующим фрагментом программы:

```
L := 1; R := n;
WHILE L < R DO                                          (2.18)
  x := a[k]; разделение (a[L]... a[R]);
  IF j < k THEN L := i END;
  IF k < i THEN R := j END
END
```

За формальным доказательством корректности этого алгоритма мы отсылаем читателя к оригинальной работе самого Хоара. Теперь легко получается и вся программа *Find*. Если предположить, что каждое разделение в среднем разбивает часть, где нахо-

```
PROCEDURE Find(k: INTEGER);
  VAR L, R, i, j: index; w, x: item;
BEGIN L := 1; R := n;
  WHILE L < R 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
```

Прогр. 2.12. Поиск k-го наибольшего элемента.

дится желаемая величина, пополам, то число требуемых сравнений равно

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

т. е. оно порядка n. Это и объясняет «мощность» программы *Find* в случаях нахождения медианы и других подобных величин и ее превосходство над прямыми методами, где сначала сортируется все множество кандидатов, а затем уже выбирается k-й элемент (в лучшем случае на это потребуется порядка $n*\log(n)$ операций). Однако в самой неблагоприятной ситуации каждый шаг разделения будет уменьшать множество кандидатов только на единицу, и в результате потребуется порядка n^2 сравнений. И вновь напоминаем: едва ли стоит пользоваться этим алгоритмом, если число элементов невелико, скажем порядка 10 *)

2.3.5. Сравнение методов сортировки массивов

Заканчивая наш обзор методов сортировки, мы попытаемся сравнить их эффективность. Как и раньше, n — число сортируемых элементов, а C и M соответственно число необходимых сравнений ключей и число обменов. Для всех прямых методов сортировки можно дать точные аналитические формулы. Они приводятся в табл. 2.8. Столбцы Min, Avg, Max определяют соответственно минимальное, усредненное и максимальное по всем n! перестановкам из n элементов значения.

Для усовершенствованных методов нет сколько-либо осмысленных, простых и точных формул. Существенно, однако, что в случае сортировки Шелла вычислительные затраты составляют $c*n^{1.2}$, а для Heapsort и Quicksort — $c*n*\log n$, где c — соответствующие коэффициенты.

Эти формулы просто вводят грубую меру для производительности как функции n и позволяют раз-

*) Автор, как правило, не учитывает в оценках коэффициенты, входящие в оценку производительности. При небольших n они могут играть большую роль, чем сами приводимые теоретические оценки. Особенно это существенно при использовании рекурсивных процедур. Даже при специальной архитектуре, ориентированной на использование процедур, итеративные методы лучше рекурсивных. — *Прим. перев.*

Таблица 2.8. Сравнение прямых методов сортировки

		Min	Avg	Max
Прямое включение	C =	n-1	$(n^2 + n - 2)/4$	$(n^2 - n)/2 - 1$
	M =	2(n-1)	$(n^2 - 9n - 10)/4$	$(n^2 - 3n - 4)/2$
Прямой выбор	C =	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	M =	3(n-1)	$n*(\ln n + 0.57)$	$n^2/4 + 3(n-1)$
Прямой обмен	C =	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	M =	0	$(n^2-n)*0.75$	$(n^2-n)*1.5$

бить алгоритмы сортировки на примитивные, прямые методы (n^2) и на усложненные или «логарифмические» методы ($n * \log(n)$). Однако для практических целей полезно иметь некоторые экспериментальные данные, способные пролить свет на те коэффициенты c, которыми один метод отличается от другого. Более того, формулы не учитывают других затрат на вычисления, кроме сравнения ключей и переносов элементов, таких, например, как управление циклами и т. д. Ясно, что эти факторы во многом зависят от отдельной вычислительной системы, но тем не менее результаты экспериментов довольно информативны. В табл. 2.9 собраны времена (в секундах) работы, обсуждавшихся выше методов сортировки, реализованных в системе Модула-2 на персональной ЭВМ Lilith. Три столбца содержат времена сортировки уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке. В начале приводятся цифры для 256 элементов, а ниже — для 2048. Четко прослеживается отличие квадратичных методов от логарифмических. Кроме того, заслуживают внимания следующие особенности.

1. Улучшение двоичного включения по сравнению с прямым включением действительно почти ничего не дает, а в случае упорядоченного массива даже получается отрицательный эффект.

2. Пузырьковая сортировка определенно наихудшая из всех сравниваемых.

Ее усовершенствованная версия, шейкерная сортировка, продолжает оставаться плохой по сравнению

Таблица 2.9. Время работы различных программ сортировки

	Упорядоченный	Случайный	В обратном порядке
	n = 256		
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
	n = 2048		
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

с прямым включением и прямым выбором (за исключением патологического случая уже упорядоченного массива).

3. Quicksort лучше в 2—3 раза, чем Heapsort. Она сортирует массив, расположенный в обратном порядке, практически с той же скоростью, что и уже упорядоченный.

2.4. СОРТИРОВКА ПОСЛЕДОВАТЕЛЬНОСТЕЙ

2.4.1. Прямое слияние

К сожалению, алгоритмы сортировки, приведенные в предыдущем разделе, невозможно применять для данных, которые из-за своего размера не помещаются в оперативной памяти машины и находятся, например, на внешних, последовательных запоминающих устройствах памяти, таких, как ленты или диски.

В таком случае мы говорим, что данные представляют собой (последовательный) файл. Для него характерно, что в каждый момент непосредственно доступна одна и только одна компонента. Это весьма сильное ограничение, если сравнивать с возможностями, предоставляемыми массивами, и поэтому приходится пользоваться другими методами сортировки. Наиболее важный из них — сортировка с помощью *слияния*. Слияние означает объединение двух (или более) последовательностей в одну-единственную упорядоченную последовательность с помощью повторяющегося выбора из доступных в данный момент элементов. Слияние намного проще сортировки, и его используют как вспомогательную операцию в более сложных процессах сортировки последовательностей. Одна из сортировок на основе слияния называется *простым слиянием*. Она выполняется следующим образом:

1. Последовательность *a* разбивается на две половины: *b* и *c*.

2. Части *b* и *c* сливаются, при этом одиночные элементы образуют упорядоченные пары.

3. Полученная последовательность под именем *a* вновь обрабатывается как указано в пунктах 1, 2; при этом упорядоченные пары переходят в такие же четверки.

4. Повторяя предыдущие шаги, сливаем четверки в восьмерки и т. д., каждый раз «удваивая» длину слитых подпоследовательностей до тех пор, пока не будет упорядочена целиком вся последовательность.

Возьмем в качестве примера такую последовательность:

44 55 12 42 94 18 06 67

После разбиения (шаг 1) получаем последовательности

44 55 12 42

94 18 06 67

Слияние одиночных компонент (т. е. упорядоченных последовательностей длины 1) в упорядоченные пары дает

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

Деля эту последовательность пополам и сливая упорядоченные пары, получаем

06 12 44 94' 18 42 55 67

Третье разделение и слияние приводят нас, наконец, к желаемому результату:

06 12 18 42 44 55 67 94

Действия по однократной обработке всего множества данных называются *фазой*. Наименьший же подпроцесс, повторение которого составляет процесс сортировки, называется *проходом* или *этапом*. В приведенном примере сортировка происходит за три прохода, каждый из которых состоит из фазы разделения и фазы слияния. Для выполнения такой сортировки нужны три ленты, поэтому она называется *трехленточным слиянием*.

Фазы разделения фактически не относятся к сортировке, ведь в них элементы не переставляются. В некотором смысле они непродуктивны, хотя и занимают половину всех операций по переписи. Если объединить разделение со слиянием, то от этих переписей можно вообще избавиться. Вместо слияния в одну последовательность результаты слияния будем сразу распределять по двум лентам, которые станут исходными для последующего прохода. В отличие от упомянутой *двухфазной* сортировки с помощью слияния будем называть такую сортировку *однофазной*. Она, очевидно, лучше, поскольку необходима только половина операций по переписи, но за это приходится «платить» четвертой лентой.

Теперь перейдем к более детальному рассмотрению программы слияния. Данные мы будем представлять как массив, обращение к элементам которого, однако, идет строго последовательно. Последующая версия сортировки слиянием будет уже основана на последовательностях, — это позволит нам сравнить две получившиеся программы и подчеркнуть строгую зависимость программы от выбранного представления для данных.

Если рассматривать массив как последовательность элементов, имеющих два конца, то его весьма

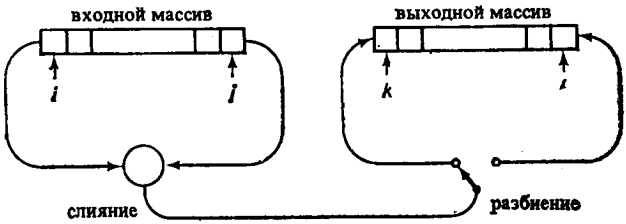


Рис. 2.12. Схема сортировки прямым слиянием для двух массивов.

просто можно использовать вместо двух последовательностей. Мы будем при слиянии брать элементы с двух концов массива, а не из двух входных файлов. Таким образом, общая схема объединенной фазы слияния-разбиения имеет вид, показанный на рис. 2.12. Направление пересылки сливаемых элементов изменяется на первом проходе после каждой упорядоченной пары, на втором — после каждой упорядоченной четверки и т. д., равномерно заполняя две выходные последовательности, представляемые двумя концами одного массива. После каждого прохода массивы «меняются ролями», выходной становится входным и наоборот.

Если объединить два концептуально различных массива в один-единственный, но двойного размера, то программа еще более упрощается. В этом случае данные представляются так:

```
a: ARRAY[1..2*n] OF item                    (2.20)
```

Индексы i и j фиксируют два входных элемента, k и L — два выходных (см. рис. 2.12). Исходными данными будут, конечно же, элементы $a_1 \ldots a_n$. Кроме этого, очевидно, нужно ввести для указания направления пересылки булевскую переменную *up*. Если *up* истинна, то в текущем проходе компоненты $a_1 \ldots a_n$ движутся на место $a_{n+1} \ldots a_{2n}$, если же истинно выражение $\sim up$, то $a_{n+1} \ldots a_{2n}$ пересылаются в $a_1 \ldots a_n$. Между последовательными проходами значение *up* изменяется на противоположное. И в заключение нам еще потребуется перемен-

ная p, задающая размер объединяемых последовательностей. Начальное значение p равно 1, и перед каждым последующим проходом она удваивается. Для простоты мы предполагаем, что всегда n равно степени двойки. Таким образом, первая версия программы сортировки с помощью простого слияния имеет такой вид:

```
PROCEDURE MergeSort;                                        (2.21)
  VAR i, j, k, L: index; up: BOOLEAN; p: INTEGER;
BEGIN up := TRUE; p := 1;
  REPEAT инициация индексов;
    IF up THEN i := 1; j := n; k := n+1; L := 2*n
    ELSE k := 1; L := n; i := n+1; j := 2*n
    END;
    слияние p-наборов из i- и j-входов в k- и L-выходы;
    up := ~up; p := 2*p
  UNTIL p = n
END MergeSort
```

Следующий этап — уточнение операторов, выделенных курсивом. Ясно, что процесс слияния n элементов сам представляет собой последовательность слияний последовательностей, т. е. p-наборов. После каждого такого частичного слияния выход переключается с нижнего на верхний конец выходного массива *) и наоборот, что гарантирует одинаковое распределение в обоих направлениях. Если сливаемые элементы направляются в левый конец выходного массива, то направление задается индексом k, и после пересылки очередного элемента он увеличивается на единицу. Если же элементы направляются в правый конец, то направление задается индексом L и он каждый раз уменьшается. Для упрощения фактического оператора слияния будем считать, что направление всегда задается индексом k, но после слияния p-набора будем менять местами значения k и L, приращение же всегда обозначается через h, имеющее значение либо 1, либо —1. Высказанные соображения приводят к такому уточнению про-

*) Точнее, с левого конца на правый. Нужно исправлять либо рисунок, либо текст. Мы выбираем последнее. — *Прим. перев.*

граммы:

```
h := 1; m := n; (*m-число сливаемых элементов *)                    (2.22)
REPEAT q := p; r := p; m := m - 2*p;
  слияние q элементов из i-входа с r элементами из j-входа;
  индекс выхода k, затем k увеличивается на h;
  h := -h; k ↔ L
UNTIL m = 0
```

Дальнейшее уточнение ведет уже к формулированию самого оператора слияния. При этом следует учитывать, что остаток подпоследовательности, оставшийся после слияния непустой подпоследовательности, добавляется простым копированием к выходной последовательности.

```
WHILE (q # 0) & (r # 0) DO
  IF a[i] < a[j] THEN                                               (2.23)
    элемент из i-входа пересылается на k-выход; i и k продвигаются;
    q := q-1
  ELSE
    элемент из j-входа посылается на k-выход, j и k продвигаются;
    r := r-1
  END
END;
копирование хвоста i-массива; копирование хвоста j-массива
```

Если теперь уточнить операции копирования остатков, то программа станет совершенно точной. Однако, прежде чем сделать это, мы хотим избавиться от ограничения на n: пока оно должно быть степенью двойки. На какие части алгоритма влияло это условие? Легко убедить себя, что наилучший способ справиться с более общей ситуацией — продолжать действовать, насколько это возможно, по-старому. В данной ситуации это означает, что мы будем продолжать слияние p-наборов до тех пор, пока не останутся последовательности размером менее p. Этот процесс затрагивает только ту часть, где определяются значения q и r — длины сливаемых последовательностей. Вместо трех операторов

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

используются четыре, эффективно, как это можно заметить, реализующие описанную выше стратегию. Через m здесь обозначается общее число элементов в двух входных последовательностях, подлежащих слиянию.

```
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, для гарантии окончания программы условие $p = n$, управляющее внешним циклом, заменяется на $p \geq n$. Проделав все эти модификации, мы можем теперь описать весь алгоритм уже как полную программу (прогр. 2.13).

Анализ сортировки с помощью слияния. Поскольку на каждом проходе p удваивается и сортировка заканчивается при $p \geq n$, то всего требуется $\lceil \log n \rceil$ проходов. На каждом проходе по определению копируются по одному разу все n элементов. Поэтому общее число пересылок

$$M = n * \lceil \log n \rceil \tag{2.24}$$

Число сравнений ключей C даже меньше M, поскольку при копировании остатков никаких сравнений не производится. Однако поскольку сортировки слиянием обычно употребляются в ситуациях, где приходится пользоваться внешними запоминающими устройствами, то затраты на операции пересылки на несколько порядков превышают затраты на сравнения. Поэтому детальный анализ числа сравнений особого практического интереса не представляет.

Алгоритм сортировки слиянием выдерживает сравнение даже с усовершенствованными методами, разбиравшимися в предыдущем разделе. Однако, хотя здесь относительно высоки затраты на работу с индексами, решающее значение играет необходимость работать с памятью размером 2n. Поэтому сортировка слиянием для массивов, т. е. для данных, размещаемых в оперативной памяти, используется редко. В последней строке табл. 2.9 приводится для срав-

```
PROCEDURE StraightMerge;
  VAR i, j, k, L, t: index;  (* диапазон индексов 1 .. 2*n*)
    h, m, p, q, r: INTEGER; up: BOOLEAN;
BEGIN up := TRUE; p := 1;
  REPEAT h := 1; m := n;
    IF up THEN i := 1; j := n; k := n+1; L := 2*n
    ELSE k := 1; L := n; i := n+1; j := 2*n
    END;
    REPEAT (*слияние серий из i-и j-входов в k-выход*)
      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 := 1 TO n DO a[i] := a[i+n] END
  END
END StraightMerge
```

Прогр. 2.13. Прямое слияние в массиве.

нения время работы алгоритма сортировки с помощью слияния (Mergesort). Оно лучше, чем у Heapsort, но хуже, чем у Quicksort.

2.4.2. Естественное слияние

В случае прямого слияния мы не получаем никакого преимущества, если данные в начале уже частично упорядочены. Размер сливаемых на k-м проходе подпоследовательностей меньше или равен 2^k и не зависит от существования более длинных уже упорядоченных подпоследовательностей, которые

можно было бы просто объединить. Фактически любые две упорядоченные подпоследовательности длиной m и n можно сразу сливать в одну последовательность из m + n элементов. Сортировка, при которой всегда сливаются две самые длинные из возможных подпоследовательностей, называется *естественным слиянием.*

Упорядоченные подпоследовательности часто называют *строками.* Однако так как слово «строка» еще чаще употребляется для названия последовательности символов, то мы для упорядоченных подпоследовательностей будем использовать термин «серия». Таким образом, подпоследовательность $a_i \ldots a_j$, удовлетворяющая условию

$$(a_{i-1} > a_i) \& (Ak : i \leqslant k < j : a_k \leqslant a_{k+1}) \& (a_j > a_{j+1}) \qquad (2.25)$$

называется *максимальной серией* или, короче, *серией.* Поэтому в сортировке естественным слиянием объединяются (максимальные) серии, а не последовательности фиксированной (заранее) длины. Если сливаются две последовательности, каждая из n серий, то результирующая содержит опять ровно n серий. Следовательно, при каждом проходе общее число серии уменьшается вдвое и общее число пересылок в самом плохом случае равно $n * \lceil \log n \rceil$, а в среднем даже меньше. Ожидаемое же число сравнений, однако, значительно больше, поскольку кроме сравнений, необходимых для отбора элементов при слиянии, нужны еще дополнительные — между последовательными элементами каждого файла, чтобы определить конец серии.

Следующим нашим упражнением будет создание алгоритма естественного слияния с помощью того же приема пошагового уточнения, который применялся при объяснении алгоритма прямого слияния. Вместо массивов здесь используются последовательности, и речь идет о несбалансированной, двухфазной сортировке слиянием с тремя лентами. Мы предполагаем, что переменная c представляет собой начальную последовательность элементов. (В реальных процессах обработки данных начальные данные, естественно, сначала копируются из исходного файла в c; это

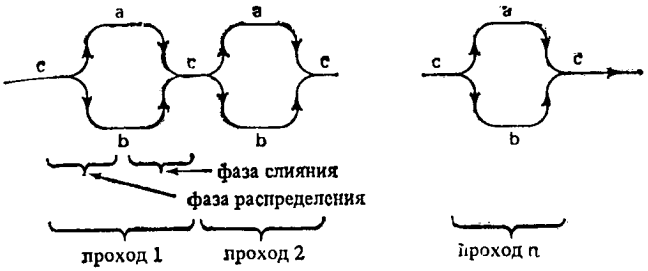


Рис. 2.13. Фазы сортировки и проходы.

делается для безопасности *). Кроме того, a и b — вспомогательные переменные-последовательности. Каждый проход состоит из фазы распределения серий из c поровну в a и b и фазы слияния, объединяющей серии из a и b вновь в c. Этот процесс иллюстрируется рис. 2.13.

В качестве примера в табл. 2.10 приведен файл с из 20 чисел в его исходном состоянии (первая строчка) и после каждого из проходов сортировки с помощью естественного слияния (строки 2—4). Обратите внимание: всего понадобилось три прохода. Процесс сортировки заканчивается, как только в c число серий станет равным единице. (Предполагается, что в начальной последовательности есть по крайней мере одна непустая серия.) Поэтому для счета числа серий, направляемых в c, мы вводим переменную L. Воспользовавшись типом последовательности (*Sequence*), введенным в разд. 2.11, можно

Таблица 2.10. Пример сортировки с помощью естественного слияния

```
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
```

*) Исходный файл может быть несколько специфическим, и его, как правило, нельзя использовать в качестве «последовательности». Например, это может быть «перфоленточный» файл. — *Прим. перев.*

написать такую программу (2.27).

```
VAR L: INTEGER; a, b, c: Sequence

REPEAT Reset(a); Reset(b); Reset(c);
  Распределение; (* с распределяется в а и b *)                    (2.27)
  Reset(a); Reset(b); Reset(c);
  L := 0; слияние (* а и b сливаются в с *)
UNTIL L = 1
```

Две фазы явно выделяются как два различных оператора. Теперь их надо уточнить, т. е. переписать с большей детализацией. Уточненное описание *распределения* (distribute) приводится в (2.28), а *слияния* (merge) — в (2.29).

```
REPEAT copyrun(c, a);
  IF ~c.eof THEN copyrun(c, b) END                                 (2.28)
UNTIL c.eof
```

```
REPEAT mergerun; L := L+1
UNTIL b.eof;                                                       (2.29)
IF ~a.eof THEN copyrun(a, c); L := L+1 END
```

Такой метод распределения приводит предположительно к следующему результату: либо в а и b будет поровну серий, либо в а будет на одну больше. Поскольку соответствующие пары серий сливаются в одну, то в а может оказаться еще одна лишняя серия, ее нужно просто скопировать. Операторы *слияние* и *распределение* формулируются в терминах уточненного оператора *слияние серий* и подчиненной процедуры *copyrun*, назначение которых совершенно понятно. Пытаясь проделать эту работу, мы сталкиваемся с серьезными трудностями: для того чтобы определить конец серии, нужно сравнивать ключи двух последовательных элементов. Однако природа последовательностей такова, что непосредственно доступен только один-единственный элемент. Поэтому невозможно избежать «заглядывания вперед», т. е. с каждой последовательностью нужно связать некий буфер. В нем содержится первый, еще не считанный элемент последовательности, т. е. буфер напоминает окно, скользящее вдоль последовательности.

Вместо того чтобы программировать явно этот механизм в нашей программе, мы предпочитаем определить еще один уровень абстракции. Это будет новый модуль с именем *Sequences*, для пользователя он заменит модуль *FileSystem*. Кроме того, мы определяем соответственно и новый тип для последовательности, но он будет ссылаться на *Sequence* из *FileSystem*. В этом новом типе будет фиксироваться не только конец последовательности, но и конец серии и, конечно же, первый элемент оставшейся части последовательности. Новый тип и связанные с ним операции задаются таким модулем определений:

```
DEFINITION MODULE Sequences;
  IMPORT FileSystem;
  TYPE item = INTEGER;

    Sequence =
      RECORD first: item;
          eor, eof: BOOLEAN;
          f: FileSystem.Sequence
      END;

  PROCEDURE OpenSeq(VAR s: Sequence);
  PROCEDURE OpenRandomSeq(VAR s: Sequence; length, seed: INTEGER);
  PROCEDURE StartRead(VAR s: Sequence);
  PROCEDURE StartWrite(VAR s: Sequence);
  PROCEDURE copy(VAR x, y: Sequence);
  PROCEDURE CloseSeq(VAR s: Sequence);
  PROCEDURE ListSeq(VAR s: Sequence);
END Sequences.
```

Необходимо сделать несколько дополнительных пояснений, касающихся выбора процедур. Как мы уже видели, алгоритмы сортировки, о которых здесь и позже пойдет речь, основаны на копировании элемента из одной последовательности в другую. Поэтому вместо отдельных операций чтения и записи мы включаем одну процедуру копирования (*copy*). После открытия файла необходимо включить механизм «заглядывания вперед», работа которого зависит от того, читается ли в данный момент файл или пишется. В первом случае первый элемент необходимо записать в «опережающий» буфер *first*. Поэтому вместо процедуры *Reset* из *FileSystem* здесь используется *StartRead* и *StartWrite*.

Еще две дополнительные процедуры включены в модуль единственно ради удобства. Процедуру *OpenRandomSeq* можно использовать вместо *OpenSeq*, если последовательность нужно инициировать какими-либо числами в случайном порядке. Процедура же *ListSeq* формирует распечатку указанной последовательности. Эти две процедуры будут употребляться при тестировании рассматриваемого алгорит-

```
IMPLEMENTATION MODULE Sequences;
  FROM FileSystem IMPORT
    File, Open, ReadWord, WriteWord, Reset, Close;
  FROM InOut IMPORT WriteInt, WriteLn;

  PROCEDURE OpenSeq(VAR s: Sequence);
  BEGIN Open(s.f)
  END OpenSeq;

  PROCEDURE OpenRandomSeq(VAR s: Sequence; length, seed: INTEGER);
    VAR i: INTEGER;
  BEGIN Open(s.f);
    FOR i := 0 TO length-1 DO
      WriteWord(s.f, seed); seed := (31*seed) MOD 997 + 5
    END
  END OpenRandomSeq;

  PROCEDURE StartRead(VAR s: Sequence);
  BEGIN Reset(s.f); ReadWord(s.f, s.first); s.eof := s.f.eof
  END StartRead;

  PROCEDURE StartWrite(VAR s: Sequence);
  BEGIN Reset(s.f)
  END StartWrite;
  PROCEDURE copy(VAR x, y: Sequence);
  BEGIN y.first := x.first;
    WriteWord(y.f, y.first); ReadWord(x.f, x.first);
    x.eof := x.f.eof; x.eor := x.eof OR (x.first < y.first)
  END copy;

  PROCEDURE CloseSeq(VAR s: Sequence);
  BEGIN Close(s.f)
  END CloseSeq;

  PROCEDURE ListSeq(VAR s: Sequence);
    VAR i, L: CARDINAL;
  BEGIN Reset(s.f); i := 0; L := s.f.length;
    WHILE i < L DO
      WriteInt(INTEGER(s.f.a[i]), 6); i := i+1;
      IF i MOD 10 = 0 THEN WriteLn END
    END;
    WriteLn
  END ListSeq;
END Sequences.
```

ма. Реализация этих концепций представлена в виде следующего модуля. Мы обращаем внимание, что в процедуре *copy* в поле *first* выходной последовательности хранится значение последнего записанного элемента. Таким образом, поля *eof* и *eor* представляют собой результаты работы *copy*, подобно тому как *eof* была определена результатом операции чтения.

Вернемся к процессу уточнения работы естественного слияния. Процедура *copyrun* и оператор *слияние* теперь удобно записываются в виде (2.30) и (2.31).

```
PROCEDURE copyrun(VAR x, y: Sequence);
BEGIN (* из x в y *)                                    (2.30)
  REPEAT copy(x, y) UNTIL x.eor
END copyrun
```

```
(* слияние из a и b в c *)                              (2.31)
REPEAT
  IF a.first < b.first THEN
    copy(a, c);
    IF a.eor THEN copyrun(b, c) END
  ELSE copy(b, c);
    IF b.eor THEN copyrun(a, c) END
  END
UNTIL a.eor OR b.eor
```

Процесс сравнения и выбора ключей при слиянии серий заканчивается, как только исчерпается одна из двух серий. После этого оставшаяся неисчерпанной серия просто передается в результирующую серию, точнее копируется ее «хвост». Это делается с помощью обращения к процедуре *copyrun*.

Вообще-то на этом создание сортировки с помощью естественного слияния следовало бы закончить. К сожалению, очень внимательный читатель может обнаружить, что программа некорректна. Ее изъян в том, что в некоторых случаях она не проводит сортировку должным образом. Возьмем, например, 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 по последовательностям a и b, получаем

a = 03' 07 13 19' 29 37 43' 57 61 71'

b = 02 05 11' 17 23 31' 41 47 59' 67

Обе эти последовательности сразу же «сливаются» в одну серию, и сортировка успешно заканчивается. Этот пример, хотя и не приводит к неправильной работе программы, демонстрирует, что распределение серии по нескольким последовательностям может дать в результате в сумме меньше серий, чем их было во входной последовательности. Это происходит из-за того, что первый элемент $i+2$-й серии может оказаться больше, чем последний элемент i-й серии, что приводит к автоматическому слиянию этих двух серий в одну-единственную серию.

Хотя мы считаем, что процедура *распределение* поровну распределяет серии в две последовательности, вследствие отмеченного явления число серий в последовательностях a и b может оказаться существенно различным. Однако наша процедура сливает только пары серий и заканчивает работу, как только будет прочитана последовательность b, при этом можно потерять остаток одной последовательности. Возьмем, например, такую входную последовательность. Она сортируется за два прохода, но часть ее теряется:

Таблица 2.11. Неверный результат работы программы 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

Эта ошибка типична для программирования. Она вызвана тем, что из виду упускается одно из возможных последствий, казалось бы, простой операции. Типична она и в том смысле, что существует несколько способов ее исправления и нужно выбрать какой-нибудь один. Вообще говоря, существуют две принципиально отличные одна от другой возможности:

1. Мы обнаруживаем, что операция распределения запрограммирована неверно и не удовлетворяет условию, чтобы число серий отличалось не более чем на единицу. Сохраняем выбранную ранее схему действий и соответствующим образом поправляем неверную процедуру.

2. Мы обнаруживаем, что исправление неверной части приводит к существенным переделкам, и пы-

```
MODULE NaturalMerge;
  FROM Sequences IMPORT item, Sequence, OpenSeq, OpenRandomSeq,
    StartRead, StartWrite, copy, CloseSeq, ListSeq;
  VAR L: INTEGER; (* число сливаемых серий *)
    a, b, c: Sequence;
    ch: CHAR;
  PROCEDURE copyrun(VAR x, y: Sequence);
  BEGIN (*из x в y*)
    REPEAT copy(x, y) UNTIL x.eor
  END copyrun;
BEGIN OpenSeq(a); OpenSeq(b); OpenRandomSeq(c, 16, 531);
  ListSeq(c);
  REPEAT StartWrite(a); StartWrite(b); StartRead(c);
    REPEAT copyrun(c, a);
      IF ~c.eof THEN copyrun(c, b) END
    UNTIL c.eof;
    StartRead(a); StartRead(b); StartWrite(c);
    L := 0;
    REPEAT
      LOOP
        IF a.first < b.first THEN
          copy(a, c);
          IF a.eor THEN copyrun(b, c); EXIT END
        ELSE copy(b, c);
          IF b.eor THEN copyrun(a, c); EXIT END
        END
      END;
      L := L+1
    UNTIL a.eof OR b.eof;
    WHILE ~a.eof DO copyrun(a, c); L := L+1 END;
    WHILE ~b.eof DO copyrun(b, c); L := L+1 END
  UNTIL L = 1;
  ListSeq(c); CloseSeq(a); CloseSeq(b); CloseSeq(c)
END NaturalMerge.
```

Прогр. 2.14. Естественное слияние.

таемся найти, какие части алгоритма можно изменить, чтобы скомпенсировать работу некорректной части программы.

В общем-то первый путь выглядит более надежным, ясным и более честным, он в достаточной мере свободен от непредусмотренных последствий и запутанных побочных эффектов. Поэтому обычно рекомендуется именно так исправлять ошибки.

Однако надо заметить, что не всегда надо отказываться и от второй возможности. Именно поэтому мы продолжим наш пример и поправим дело, моди-

фицировав процедуру слияния, а не изначально неверную процедуру распределения.

Это означает, что мы оставляем схему распределения, но отказываемся от условия, что серии распределяются поровну. В результате производительность может оказаться меньше оптимальной. Однако в наихудшем случае производительность не изменится, а существенно неравномерное распределение статистически маловероятно. Поэтому соображения, связанные с эффективностью, не могут препятствовать предложенному решению.

Если условия равного распределения серий больше не существует, то процедуру слияния следует изменить: после достижения конца одного из файлов нужно копировать не одну серию, а всю оставшуюся часть другого. Это приводит к четким и очень простым по сравнению с модификацией процедуры *распределения* изменениям. (Читатель может сам убедиться в справедливости такого утверждения.) Пересмотренная версия алгоритма слияния включена в уже полную прогр. 2.14, причем процедуры, к которым обращаются лишь один раз, прямо подставлены в соответствующее место.

2.4.3. Сбалансированное многопутевое слияние

Затраты на любую последовательную сортировку пропорциональны числу требуемых проходов, так как по определению при каждом из проходов копируются все данные. Один из способов сократить это число — распределять серии в более чем две последовательности. Слияние r серий поровну распределенных в N последовательностей даст в результате r/N серий. Второй проход уменьшит это число до r/N^2, третий — до r/N^3 и т. д., после k проходов останется r/N^k серий. Поэтому общее число проходов, необходимых для сортировки n элементов с помощью *N-путевого слияния*, равно $k = \lceil \log_N n \rceil$. Поскольку в каждом проходе выполняется n операций копирования, то в самом плохом случае понадобится $M = n * \lceil \log_N n \rceil$ таких операций.

В качестве следующего примера на программирование мы начнем разрабатывать программу сор-

тировки, в основу которой положим многопутевое слияние. Чтобы подчеркнуть отличие новой программы от прежней естественной двухфазной процедуры слияния, мы будем формулировать многопутевое слияние как сбалансированное слияние с одной-единственной фазой. Это предполагает, что в каждом проходе участвует равное число входных и выходных файлов, в которые по очереди распределяются последовательные серии. Мы будем использовать N последовательностей (N — четное число), поэтому наш алгоритм будет базироваться на N/2-путевом слиянии. Следуя ранее изложенной стратегии, мы не будем обращать внимание на автоматическое слияние двух следующих одна за другой последовательностей в одну. Поэтому мы постараемся программу слияния не ориентировать на условие, что во входных последовательностях находится поровну серий.

В нашей программе мы впервые сталкиваемся с естественным приложением такой структуры данных, как массив файлов. И в самом деле, даже удивительно, насколько эта программа отличается от предыдущей, хотя мы всего лишь перешли от двухпутевого слияния к многопутевому. Такое отличие — результат условия, что процесс слияния по исчерпании одного из входов еще не заканчивается. Поэтому нужно хранить список входов, остающихся активными, т. е. входов, которые еще не исчерпались. Появляется и еще одно усложнение: нужно после каждого прохода переключать группы входных и выходных последовательностей.

Мы начнем с определений и к двум уже знакомым типам *item* и *sequence* добавим тип

```
seqno = [1 .. N]                                    (2.33)
```

Ясно, что номера последовательностей используются как индексы для массива последовательностей, состоящих из элементов. Будем считать, что начальная последовательность элементов задается переменной

```
f0: Sequence                                        (2.34)
```

и для процесса сортировки имеем в своем распоряжении n лент (n — четное)

```
f: ARRAY seqno OF Sequence                          (2.35)
```

Для решения задачи переключения лент *) можно рекомендовать технику карты индексов лент. Вместо прямой адресации ленты с помощью индекса она адресуется через карту t, т. е. вместо f_i мы будем писать f_{t_i}, причем сама карта (map) определяется так:

```
t: ARRAY seqno OF seqno
```

Если в начале $t_i = i$ для всех i, то переключение представляет собой обмен местами компонент $t_i \leftrightarrow t_{Nh+i}$ для всех $i = 1 \ldots Nh$, где $Nh = N/2$. В этом случае мы всегда можем считать, что $f_{t_1}, \ldots, f_{t_{Nh}}$ — входные последовательности, а $f_{t_{Nh+1}}, \ldots, f_{t_N}$ — выходные. (Далее вместо f_{t_j} мы будем просто говорить: «последовательность j»). Теперь можно дать первый «набросок» алгоритма.

```
MODULE BalancedMerge;
  VAR i, j: seqno;
    L: INTEGER; (* число распределяемых серий *)
    t: ARRAY seqno OF seqno;
BEGIN (* распределение начальных серий в t[1]...t[Nh] *)          (2.37)
  j := Nh; L := 0;
  REPEAT
    IF j < Nh THEN j := j+1 ELSE j := 1 END;
    копирование одной серии из f0 в последовательность j;
    L := L+1
  UNTIL f0.eof;
  FOR i := 1 TO N DO t[i] := i END;
  REPEAT (* слияние из t[1]...t[nh] в t[nh+1]...t[n] *)
    установка входных последовательностей;
    L := 0;
    j := Nh+1; (*j -индекс выходной последовательности*)
    REPEAT L := L+1;
      слияние a серий с входов b в t[j];
      IF j < N THEN j := j+1 ELSE j := Nh+1 END
    UNTIL все входы исчерпаны
    переключение последовательностей
  UNTIL L = 1
  (* отсортированная последовательность в t[1] *)
END BalancedMerge.
```

*) Надо отметить, что сама задача еще не описана и не ясно, какие здесь могут быть проблемы. — *Прим. перев.*

Прежде всего уточним оператор первоначального распределения серий. Используя определение последовательности (2.26) и процедуры *copy* (2.32) заменяем *копирование одной серии из f0 в последовательность j* на оператор

```
REPEAT copy (f0, f[j]) UNTIL f0.eor
```

Копирование серии прекращается либо при обнаружении первого элемента следующей серии, либо по достижении конца входного файла.

В нашем алгоритме сортировки осталось определить более детально следующие операторы:

1. Установка на начало входных последовательностей.
2. Слияние одной серии из входа на t_j.
3. Переключение последовательностей.
4. Все входы исчерпаны.

Во-первых, необходимо точно идентифицировать текущие входные последовательности. Обратите внимание: число *активных* входов может быть меньше N/2. Фактически максимальное число входов может быть равно числу серий, и сортировка заканчивается как только останется одна-единственная последовательность. Может оказаться, что в начале последнего прохода сортировки число серий будет меньше N/2. Поэтому мы вводим некоторую переменную, скажем k1, задающую фактическое число работающих входов. Инициация k1 включается в оператор *установка входных последовательностей* таким образом:

```
IF L < Nh THEN k1 := L ELSE k1 := Nh END;
FOR i := 1 TO k1 DO StartRead(f[t[i]]) END
```

Понятно, что оператор (2) по исчерпании какого-либо входа должен уменьшать k1. Следовательно, предикат (4) можно легко представить отношением k1 = 0. Уточнить же оператор (4), однако, несколько труднее: он включает повторяющийся выбор наименьшего из ключей и отсылку его на выход, т. е. в текущую выходную последовательность. К тому же этот процесс усложняется необходимостью определять ко-

нец каждой серии. Конец серии достигается либо (1), когда очередной ключ меньше текущего ключа, либо (2) по достижении конца входа. В последнем случае вход исключается из работы и k1 уменьшается, в первом же закрывается серия, элементы соответствующей последовательности уже не участвуют в выборе, но это продолжается лишь до того, как будет закончено создание текущей выходной серии. Отсюда следует, что нужна еще одна переменная, скажем k2, указывающая число входов источников, действительно используемых при выборе очередного элемента. В начале ее значение устанавливается равным k1 и уменьшается всякий раз, когда из-за условия (1) серия заканчивается.

К сожалению, недостаточно ввести переменную k2. Нам необходимо не только знать число последовательностей, но и знать, какие из них действительно используются. Очевидное решение — использовать массив булевских компонент, определяющих доступность последовательностей. Однако мы выбираем другой метод, он приведет к более эффективной процедуре выбора, а это в конечном счете наиболее часто повторяющаяся часть алгоритма. Вместо булевского массива вводим вторую карту для лент —*ta*. Эта карта используется вместо t, причем $ta_1 \ldots ta_{k2}$ — индексы доступных последовательностей. Таким образом, оператор (2) можно записать следующим образом:

```
k2 := k1;
REPEAT выбор минимального ключа
  ta[mx] — номер последовательности, где он есть;
  copy(f[ta[mx]], f[t[j]]);                                    (2.39)
  IF f[ta[mx]].eof THEN исключение ленты
  ELSIF f[ta[mx]].eor THEN закрытие серии
  END
UNTIL k2 = 0
```

Так как число последовательностей в любом практическом использовании обычно бывает достаточно мало, то в качестве алгоритма выбора, который нам нужно определить на очередном этапе уточнения, может быть использован простой линейный поиск.

Оператор *исключить ленту* предполагает уменьшение k1 и k2, а также переупорядочение индексов в карте *ta*. Оператор *закрыть серию* просто уменьшает k2 и переупорядочивает соответствующим образом *ta*. Детали показаны в прогр. 2.15 — результаты уточнений,

```
MODULE BalancedMerge;
  FROM Sequences IMPORT item, Sequence, OpenSeq, OpenRandomSeq,
    StartRead, StartWrite, copy, CloseSeq, ListSeq;

  CONST N = 4; Nh = N DIV 2;
  TYPE seqno = [1 .. N];
  VAR i, j, mx, tx: seqno;
    L, k1, k2: CARDINAL;
    min, x: item;
    t, ta: ARRAY seqno OF seqno;
    f0: Sequence;
    f: ARRAY seqno OF Sequence;

BEGIN OpenRandomSeq(f0, 100, 737); ListSeq(f0);
  FOR i := 1 TO N DO OpenSeq(f[i]) END;
  (*распределение начальных серий  t[1]...t[Nh] *)
  FOR i := 1 TO Nh DO StartWrite(f[i]) END;
  j := Nh; L := 0; StartRead(f0);
  REPEAT
    IF j < Nh THEN j := j+1 ELSE j := 1 END;
    REPEAT copy(f0, f[j]) UNTIL f0.eor;
    L := L+1
  UNTIL f0.eof;
  FOR i := 1 TO N DO t[i] := i END;
  REPEAT (*слияние из t[1]...t[nh] в  t[nh + 1]...t[n] *)
    IF L < Nh THEN k1 := L ELSE k1 := Nh END;
    FOR i := 1 TO k1 DO
      StartRead(f[t[i]]); ta[i] := t[i]
    END;
    L := 0;    (*число сливаемых серий *)
    j := Nh+1;  (*j -индекс входной последовательности *)
    REPEAT (*слияние входных серий в t[j] *)
      L := L+1; k2 := k1;
      REPEAT (* выбор минимального ключа *)
        i := 1; mx := 1; min := f[ta[1]].first;
        WHILE i < k2 DO
          i := i+1; x := f[ta[i]].first;
          IF x < min THEN min := x; mx := i END
        END;
        copy(f[ta[mx]], f[t[j]]);
```

```
      IF f[ta[mx]].eof THEN (* исключение ленты *)
        StartWrite(f[ta[mx]]); ta[mx] := ta[k2];
        ta[k2] := ta[k1]; k1 := k1-1; k2 := k2-1
      ELSIF f[ta[mx]].eor THEN (* закрытие серии *)
        tx := ta[mx]; ta[mx] := ta[k2]; ta[k2] := tx; k2 := k2-1
      END
    UNTIL k2 = 0;
    IF j < N THEN j := j+1 ELSE j := Nh+1 END
  UNTIL k1 = 0;
  FOR i := 1 TO Nh DO
    tx := t[i]; t[i] := t[i+Nh]; t[i+Nh] := tx;
  END
 UNTIL L = 1;
 ListSeq(f[t[1]])
 (* отсортированная последовательность в t[1] *)
END BalancedMerge
```

Прогр. 2.15. Сортировка с помощью сбалансированного слияния.

начинающихся с (2.37) до (2.39). Оператор *переключения последовательности* написан в соответствии с данными ранее пояснениями.

2.4.4. Многофазная сортировка

Мы уже познакомились с необходимыми приемами и в достаточной мере готовы к исследованиям и программированию, связанным с новыми, более эффективными, чем сбалансированная сортировка, алгоритмами. В сбалансированной сортировке исчезла необходимость в чистых операциях копирования, когда распределение и слияние оказались объединенными в одну фазу. Возникает вопрос: а нельзя ли еще эффективнее работать с данными последовательностями? Оказывается, можно, в основе нашего очередного усовершенствования лежит отказ от жесткого понятия прохода и переход к более изощренному использованию последовательностей. Мы больше не будем считать, что есть N/2 входов и столько же выходов и они меняются местами после каждого отдельного прохода. Более того, уже само понятие прохода делается расплывчатым. Новый метод был изобретен Р. Гилстэдом [2.3] и называется *многофазной сортировкой* (Polyphase Sort),

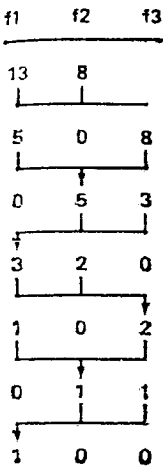


Рис. 2.14. Многофазная сортировка слиянием 21 серии на трех переменных-последовательностях.

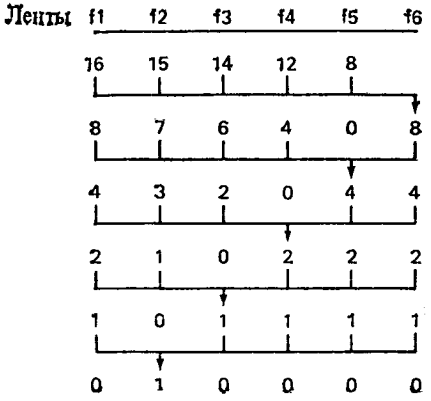


Рис. 2.15. Многофазная сортировка слиянием 65 серий на шести переменных-последовательностях.

Сначала продемонстрируем его на примере с тремя последозательностями. В каждый момент сливаются элементы из двух источников и пишутся в третью переменную-последовательность. Как только одна из входных последовательностей исчерпывается, она сразу же становится выходной для операции слияния из оставшейся, неисчерпанной входной последовательности и предыдущей выходной.

Поскольку известно, что n серий на каждом из входов трансформируются в n серий на выходе, то нужно только держать список из числа серий в каждой из последовательностей (а не определять их действительные ключи). Будем считать (рис. 2.14), что в начале две входные последовательности f_1 и f_2 содержат соответственно 13 и 8 серий. Таким образом, на первом проходе 8 серий из f_1 и f_2 сливаются в f_3, на втором — 5 серий из f_3 и f_1 сливаются в f_2 и т. д. В конце концов в f_1 оказывается отсортированная последовательность.

Таблица 2.12. Идеальное распределение серий по двум последовательностям

L	$a_1^{(L)}$	$a_2^{(L)}$	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

Во втором примере многофазная сортировка идет на 6 последовательностях (рис. 2.15). Сначала в f_1 находится 16 серий, в f_2 — 15, в f_3 — 14, в f_4 — 12 и 8 серий в f_5. На первом частичном проходе сливаются и попадают в f_6 8 серий, а в конце f_2 содержит все отсортированное множество элементов.

Многофазная сортировка более эффективна, чем сбалансированная, поскольку она имеет дело с N — 1-путевым слиянием, а не с N/2-путевым, если она начинается с N последовательностей. Ведь число необходимых проходов приблизительно равно $\log_N n$, где n — число сортируемых элементов, а N — степень операции слияния, — это и определяет значительное преимущество нового метода.

Конечно, для этих примеров мы тщательно выбирали начальное распределение серий. Выбирая распределение для хорошей работы алгоритма, мы шли «с конца», т. е. начинали с заключительного распределения (последней строки на рис. 2.15). Переписывая так таблицы для наших двух примеров и «вращая» каждую строку на одну позицию относительно предыдущей строки, получаем табл. 2.13 и 2.14, соответствующие шести проходам для трех и шести последовательностей.

Из табл. 2.12 при $L > 0$ можно вывести соотношения

$$a_2^{(L+1)} = a_1^{(L)}$$
$$a_1^{(L+1)} = a_1^{(L)} + a_2^{(L)} \qquad (2.40)$$

а $a_1^{(0)} = 1$, $a_2^{(0)} = 0$. Считая $f_{i+1} = a_1^{(i)}$, получаем для $i > 0$

$$f_{i+1} = f_i + f_{i-1},\ f_1 = 1,\ f_0 = 0 \qquad (2.41)$$

Таблица 2.13. Идеальное распределение серий по пяти последовательностям

L	$a_1(L)$	$a_2(L)$	$a_3(L)$	$a_4(L)$	$a_5(L)$	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

Это рекурсивные правила (или рекурсивные отношения), определяющие так называемые *числа Фибоначчи*:

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

Любое из чисел Фибоначчи — сумма двух предшествующих. Следовательно, для хорошей работы многофазной сортировки на трех последовательностях необходимо, чтобы числа начальных серий в двух входных последовательностях были двумя соседними числами Фибоначчи.

А что можно сказать по поводу второго примера с шестью последовательностями (табл. 2.13)? Опять легко вывести правила образования для числа серий:

$$\begin{aligned}
a_5^{(L+1)} &= a_i^{(L)} \\
a_4^{(L+1)} &= a_1^{(L)} + a_5^{(L)} = a_1^{(L)} + a_1^{(L-1)} \\
a_3^{(L+1)} &= a_1^{(L)} + a_4^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} \\
a_2^{(L+1)} &= a_1^{(L)} + a_3^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} + a_1^{(L-3)} \\
a_1^{(L+1)} &= a_1^{(L)} + a_2^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} + \\
&+ a_1^{(L-3)} + a_1^{(L-4)}
\end{aligned} \qquad (2.42)$$

Подставляя f_i вместо $a_1^{(i)}$, получаем

$$\begin{aligned}
&f_{i+1} = f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4} \text{ для } i \geqslant 4 \\
&f_4 = 1 \\
&f_i = 0 \text{ для } i < 4
\end{aligned} \qquad (2.43)$$

Такие числа называются числами Фибоначчи четвертого порядка. В общем случае *числа Фибоначчи*

порядка p определяются следующим образом:

$$f_{i-1}^{(p)} = f_i^{(p)} + f_{i-1}^{(p)} + f_{i-2}^{(p)} + f_{i-3}^{(p)} + f_{i-4}^{(p)}$$

для $i \geqslant p$

$$f_p^{(p)} = 1 \qquad (2.44)$$

$f_i^{(p)} = 0$ для $0 \leqslant i < p$

Заметим, что обычные числа Фибоначчи — числа первого порядка.

Теперь уже ясно, что начальные числа серий для совершенной многофазной сортировки с N последовательностями представляют собой суммы любых $N-1$, $N-2$, ..., 1 последовательных чисел Фибоначчи порядка $N-2$ (см. табл. 2.14). Отсюда явно следует, что наш метод применим лишь для входов, сумма серий на которых есть сумма $N-1$ таких чисел Фибоначчи. А что делать, если число начальных серий отличается от такой идеальной суммы? Ответ прост (и типичен для подобных ситуаций): будем считать, что у нас есть гипотетические пустые серии, причем столько, что сумма пустых и реальных серий равна идеальной сумме. Такие серии будем называть просто *пустыми* (dymmy runs).

Однако фактически этот выход из положения неудовлетворителен, поскольку он сразу же порождает новый и более трудный вопрос: как узнавать во время слияния такие пустые серии? Прежде чем отвечать на такой вопрос, сначала исследуем проблему

Таблица 2.14. Число серий, допускающее идеальное распределение

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

начального распределения фактических и пустых серий на N — 1 лентах *).

Однако в поисках подходящего правила распределения необходимо прежде всего знать, как будут сливаться пустые и реальные серии. Ясно, что если мы будем считать серию из последовательности i пустой, то это просто означает, что в слиянии она не участвует и оно (слияние) проходит не из N — 1 последовательностей, а из меньшего их числа. Слияние пустых серий на всех N — 1 источников означает, что никакого реального слияния не происходит, вместо него в выходную последовательность записывается результирующая пустая серия. Отсюда можно заключить, что пустые серии нужно как можно более равномерно распределять по N — 1 последовательностям, ведь мы заинтересованы в реальном слиянии из как можно большего числа источников.

Теперь на некоторое время забудем о пустых сериях и рассмотрим задачу распределения неизвестного числа серий по N — 1 последовательностям. Ясно, что в процессе распределения можно вычислять числа Фибоначчи порядка N — 2, определяющие желательное число серий в каждом из источников. Предположим, например, что N = 6, и, ссылаясь на табл. 2.14, начнем распределять серии, как указано в строке с индексом L = 1 (1, 1, 1, 1, 1); если есть еще серии, то переходим ко второй строке (2, 2, 2, 2, 1), если все еще остаются серии, то берем третью строку (4, 4, 4, 3, 2), и т. д. Будем называть индекс строки *уровнем*. Очевидно, чем больше число серий, тем выше уровень чисел Фибоначчи, который в данном случае равен количеству проходов или переключений, необходимых для последующей сортировки. Тогда в первом приближении можно сформулировать такой алгоритм распределения:

1. Пусть наша цель — получить при распределении числа Фибоначчи порядка N — 2 и уровня 1.

---

*) Размещать недостающие пустые серии в одной последовательности не имеет смысла, ибо алгоритм может просто зациклиться. Следовательно, их нужно как-то перемешивать с реальными на всех лентах. — *Прим. перев.*

2. Проводим распределение, стремясь достичь цели.

3. Если цель достигнута, вычисляем следующий уровень чисел Фибоначчи; разность между этими числами и числами на предыдущем уровне становится новой целью распределения. Возвращаемся к шагу 2. Если цели невозможно достичь из-за того, что входные данные исчерпаны, заканчиваем распределение.

Правила вычисления следующего уровня чисел Фибоначчи содержатся в их определении (2.44). Таким образом, мы можем сконцентрировать внимание на шаге 2, где, имея цель, необходимо распределять одну за другой поступающие серии по $N - 1$ выходным последовательностям. Именно в этот момент в наших рассуждениях и появляются пустые серии.

Предположим, повышая уровень, мы вводим новые цели с помощью разностей d_i для $i = 1 \ldots N - 1$, где d_i обозначает число серий, которые нужно на данном шаге направить в последовательность i. Теперь можно считать, что сразу помещаем d_i пустых серий в последовательность i, а затем рассматриваем последующее распределение как замену пустых серий на реальные, отмечая каждую замену вычитанием 1 из счетчика d_i. Таким образом, по исчерпании входного источника d_i будет указывать число пустых серий в последовательности i.

Какой алгоритм ведет к оптимальному распределению, неизвестно, но излагаемый ниже алгоритм дает весьма хорошие результаты. Он называется *горизонтальным распределением* (см. Кнут, т. 3, с. 322); этот термин становится понятным, если представить себе серии сложенными в пирамиду, как это показано на рис. 2.16 для $N = 6$ уровня 5 (см. табл. 2.14).

Для того чтобы получить равномерное распределение остающихся пустых серий наиболее быстрым способом, будем считать, что замены на реальные серии сокращают размер пирамиды: пустые серии выбираются из пирамиды слева направо. При таком способе серии распределяются по последовательностям в порядке номеров, приведенных на рис. 2.16.

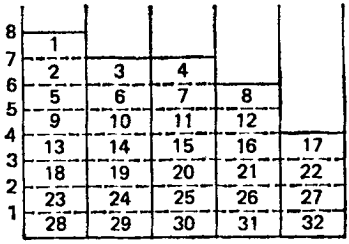


Рис. 2.16. Горизонтальное распределение серий.

Теперь мы уже в состоянии описать алгоритм в виде процедуры с именем *select*, к которой обращаются каждый раз после копирования, когда нужно выбирать, куда отправлять новую серию. Будем считать, что есть переменная j, указывающая индекс текущей выходной последовательности, а a_i и d_i — числа идеального и «пустого» распределения для последовательности i.

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

Эти переменные инициируются такими значениями:

$a_i = 1$, $d_i = 1$ для $i = 1 \ldots N - 1$

$a_N = 0$, $d_N = 0$ (пустые)

$j = 1$, level $= 1$

Отметим, что *select* должно вычислять очередную строку табл. 2.14, т. е. значения $a_i^{(L)} \ldots a_{N-1}^{(L)}$ при каждом увеличении уровня. В этот же момент вычисляются и следующие цели — разности $d_i = a_i^{(L)} - a_i^{(L-1)}$. Приведенный алгоритм основан на том, что результирующее d_i при увеличении индекса уменьшается (ведущие вниз ступени на рис. 2.16). Обратите внимание, что переход от уровня 0 к уровню 1 представляет собой исключение, поэтому алгоритм можно использовать, начиная с уровня 1. Работа *select* заканчивается уменьшением d_j на 1, эта операция соответствует замене пустой серии в последовательности j на реальную серию.

Будем считать, что у нас есть процедура копирования серии с входа f_0 в f_1, тогда мы можем сформулировать фазу начального распределения следующим образом (предполагается, что есть по крайней мере одна серия):

```
REPEAT select; copyrun
UNTIL f0.eof
```

Однако здесь следует на мгновение задержаться и вспомнить, что происходило при распределении серий в ранее разбиравшейся сортировке естественным слиянием: две последовательно поступающие на один выход серии могут превратиться в одну-единственную, что приведет к «нарушению» ожидаемого числа серий. Если мы имеем дело с алгоритмом, правильная работа которого не зависит от числа серий, то на такой побочный эффект можно было не обращать внимания. Однако в многофазной сортировке мы особенно заботимся о точном числе серий в каждой последовательности. Поэтому мы не можем отмахнуться от эффекта такого случайного слияния и приходится усложнять наш алгоритм распределения. Нужно для всех последовательностей хранить ключ последнего элемента последней серии. К счастью, наша реализация модуля Sequences это предусматривала. В случае выходных последовательностей, *f. first* представляет последний записанный элемент. Очередная попытка написать алгоритм распределения может выглядеть так:

```
REPEAT select;                                              (2.48)
  IF f[j].first <= f0.first THEN продолжать старую серию END;
  copyrun
UNTIL f0.eof
```

Очевидная ошибка — забыли, что f[i].first имеет какое-то значение лишь после копирования первого элемента. Правильно было бы распределить по одной серии в N — 1 выходных последовательностей, не обращаясь к f[j].first *). Оставшиеся же серии распре-

*) Надо еще учесть, что при таком начальном «засеивании» серии могут кончиться! — *Прим. перев.*

деляются в соответствии с (2.49).

```
WHILE ~f0.eof DO                                              (2.49)
  select;
  IF f[j].first <= f0.first THEN
    copyrun;
    IF f0.eof THEN d[j] := d[j] + 1 ELSE copyrun END
  ELSE copyrun
  END
END
```

Теперь мы наконец можем приступить к самому главному алгоритму многофазной сортировки слиянием. Его основная структура подобна главной части программы N-путевого слияния: внешний цикл сливает серии, пока не будут исчерпаны входы, внутренний сливает одиночные серии из каждого источника, а самый внутренний цикл выбирает начальный ключ и пересылает включаемый элемент в целевой файл. Принципиальные отличия же в следующем:

1. В каждом проходе только одна выходная последовательность, а не N/2.

2. Вместо переключения на каждом проходе N/2 входных и N/2 выходных последовательностей на выход подключается освободившаяся последовательность. Это делается с помощью карты индексов последовательностей.

3. Число входных последовательностей меняется от серии к серии. В начале каждой серии оно определяется по счетчикам пустых серий d_i: если для всех i $d_i > 0$, то делаем вид, что сливаем $N-1$ пустых серий и создаем пустую серию на выходе, т. е. просто увеличиваем d_N для выходной последовательности. В противном случае сливается по одной серии из всех источников, где $d_i = 0$, а d_i для других уменьшается на единицу; это означает, что из них взято по пустой серии. Число входных последовательностей, участвующих в слиянии, обозначается через k.

4. Определять окончание фазы по концу $N-1$-й последовательности уже нельзя, поскольку из этого источника может требоваться слияние пустых серий,

Вместо этого теоретически необходимое число серий определяется по коэффициентам a_i. Сами коэффициенты a_i вычисляются на этапе распределения, теперь они могут вновь перевычисляться.

Теперь в соответствии с этими правилами можно написать главную часть многофазной сортировки. Считается, что все N — 1 последовательностей с исходными сериями находятся в состоянии чтения, а компоненты карты лент $t_i = i$.

```
REPEAT (* слияние из t[1]...t[N-1] в t[N] *)                    (2.50)
  z := a[N-1]; d[N] := 0; StartWrite(f[t[N]]);
  REPEAT k := 0; (* слияние одной серии *)
    (* определение числа активных входных последовательностей *)
    FOR i := 1 TO N-1 DO
      IF d[i] > 0 THEN d[i] := d[i] - 1
      ELSE k := k+1; ta[k] := t[i]
      END
    END;
    IF k = 0 THEN d[N] := d[N] + 1
    ELSE слияние одной реальной серии из t[1]...t[k] в t[N]
    END;
    z := z-1
  UNTIL z = 0;
  StartRead(f[t[N]]);
  поворот в карте t; вычисление a[i] для следующего уровня;
  StartWrite(f[t[N]]); level := level - 1
UNTIL level = 0
(* отсортированный результат в t[1] *)
```

Фактически операция слияния почти идентична той же операции в сортировке с помощью N-путевого слияния, разница только в том, что здесь алгоритм исключения последовательности несколько проще. Поворот карт индексов последовательностей и соответствующих счетчиков d_i (также как и перевычисление коэффициентов a_i при переходе на низший уровень) очевиден, с этими действиями можно детально познакомиться в прогр. 2.16, которая и представляет собой полный алгоритм многофазной сортировки (Polyphase sort).

```
MODULE Polyphase;
  FROM Sequences IMPORT Item, Sequence, OpenSeq, OpenRandomSeq,
    StartRead, StartWrite, copy, CloseSeq, ListSeq;

  CONST N = 6;
  TYPE seqno = [1 .. N];

  VAR i, j, mx, tn: seqno;
    k, dn, z, level: CARDINAL;
    x, min: Item;
    a, d: ARRAY seqno OF CARDINAL;
    t, ta: ARRAY seqno OF seqno;
    f0: Sequence;
    f: ARRAY seqno OF Sequence;

  PROCEDURE select;
    VAR i, z: CARDINAL;
  BEGIN
    IF d[j] < d[j+1] THEN j := j+1
    ELSE
      IF d[j] = 0 THEN
        level := level + 1; z := a[1];
        FOR i := 1 TO N-1 DO
          d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
        END
      END;
      j := 1
    END;
    d[j] := d[j] - 1
  END select;
  PROCEDURE copyrun; (* из f0 в f[j] *)
  BEGIN
    REPEAT copy(f0, f[j]) UNTIL f0.eor
  END copyrun;

BEGIN OpenRandomSeq(f0, 100, 561); ListSeq(f0);
  FOR i := 1 TO N DO OpenSeq(f[i]) END;
  (* распределение начальных серий *)
  FOR i := 1 TO N-1 DO
    a[i] := 1; d[i] := 1; StartWrite(f[i])
  END;
```

```
level := 1; j := 1; a[N] := 0; d[N] := 0; StartRead(f0);
REPEAT select; copyrun
UNTIL f0.eof OR (j = N-1);
WHILE ~f0.eof DO
  select; (* f[j].first = последний, записанный в f[j] элемент *)
  IF f[j].first <= f0.first THEN
    copyrun;
    IF f0.eof THEN d[j] := d[j] + 1 ELSE copyrun END
  ELSE copyrun
  END
END;

FOR i := 1 TO N-1 DO t[i] := i; StartRead(f[i]) END;
t[N] := N;
REPEAT (* слияние из t[1]...t[N-1] в t[N] *)
  z := a[N-1]; d[N] := 0; StartWrite(f[t[N]]);
  REPEAT k := 0; (* слияние одной серии *)
    FOR i := 1 TO N-1 DO
      IF d[i] > 0 THEN d[i] := d[i] - 1
      ELSE k := k+1; ta[k] := t[i]
      END
    END;
    IF k = 0 THEN d[N] := d[N] + 1
    ELSE (* слияние одной реальной серии из t[1]...t[k] в t[N] *)
      REPEAT i := 1; mx := 1; min := f[ta[1]].first;
        WHILE i < k DO
          i := i+1; x := f[ta[i]].first;
          IF x < min THEN min := x; mx := i END
        END;
        copy(f[ta[mx]], f[t[N]]);
        IF f[ta[mx]].eor THEN (* сброс этого источника *)
          ta[mx] := ta[k]; k := k-1
        END
      UNTIL k = 0
    END;
    z := z-1
  UNTIL z = 0;
  StartRead(f[t[N]]); (* поворот последовательностей *)
  tn := t[N]; dn := d[N]; z := a[N-1];
  FOR i := N TO 2 BY -1 DO
    t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
```

```
    END;
    f[1] := tn; d[1] := dn; a[1] := z;
    StartWrite(f[t[N]]); level := level - 1
  UNTIL level = 0;
  ListSeq(f[t[1]]);
  FOR i := 1 TO N DO CloseSeq(f[i]) END
END Polyphase.
```

Прогр. 2.16. Многофазная сортировка.

2.4.5. Распределение начальных серий

Мы пришли к сложным программам последовательной сортировки, поскольку более простые методы, работающие с массивами, предполагают, что у нас есть достаточно большая память со случайным доступом, в которой можно хранить все сортируемые данные. Очень часто такой памяти в машине нет, и вместо нее приходится пользоваться достаточно большой памятью на устройствах с последовательным доступом. Мы видели, что методы последовательной сортировки не требуют практически никакой оперативной памяти, если не считать буферов для файлов и, конечно, самой программы. Однако даже на самых маленьких машинах есть оперативная память со случайным доступом по размерам, почти всегда превышающая память, требующуюся для создания нами здесь программ. Непростительно было бы не попытаться использовать ее оптимальным образом.

Решение заключается в объединении методов сортировок для массивов и для последовательностей. В частности, на этапе распределения начальных серий можно использовать какую-либо специально приспособленную сортировку массивов для того, чтобы получить серии, длина которых L была бы приблизительно равна размеру доступной оперативной памяти. Ясно, что в последующих проходах, выполняющих слияние, никакая дополнительная сортировка, ориентированная на массивы, улучшений не даст, так как размер серий постоянно растет и всегда превышает размер доступной оперативной памяти. Поэтому мы можем полностью сосредоточить внима-

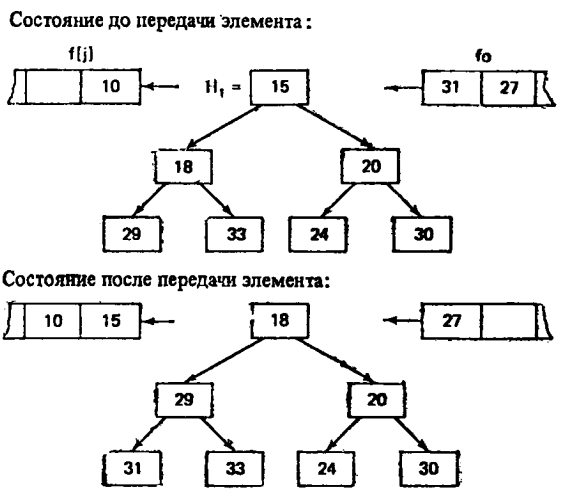


Рис. 2.17. Процесс просеивания ключа через пирамиду.

ние на улучшении алгоритма формирования начальных серий.

Естественно, мы сразу же обращаемся к логарифмическим алгоритмам сортировки массивов. Наиболее подходящими из них представляются сортировка по дереву или метод Heapsort. Пирамиду можно рассматривать как некоторого рода туннель, через который должны пройти все элементы — одни быстрее, другие медленнее. Наименьший ключ легко извлекается с вершины пирамиды, а его замещение — очень эффективный процесс. «Фильтрация» элемента из входной последовательности f0 через всю пирамиду-туннель H в выходную последовательность f[j] просто описывается такими операторами:

```
WriteWord(f[j], H[1]); ReadWord(f0, H[1]); sift(1, n)   (2.51)
```

Процесс *sift* (просеивание) уже описывался в разд. 2.2.5, он предназначен для просеивания вновь поступающего элемента H_1 вниз пирамиды, на свое место. Отметим, что H_1 — наименьший элемент из находящихся в пирамиде. На рис. 2.17 дан пример

этого процесса. Программа в конечном счете получается более сложной, поскольку:

1. В начале пирамиды H пуста и ее нужно прежде заполнить.

2. Ближе к концу работы пирамида будет заполнена лишь частично, а затем вообще опустошится.

3. Нужно сохранять информацию о началах новых серий, чтобы вовремя менять индекс выхода j.

Прежде всего формально опишем переменные, явно участвующие в нашем процессе.

```
VAR f0: Sequence;
  f: ARRAY seqno OF Sequence;
  H: ARRAY [1 .. m] OF item;                    (2.52)
  L, R: INTEGER
```

где m — размер пирамиды H. Для обозначения величины m/2 используем константу mh; L и R — индексы, ограничивающие пирамиду. Процесс фильтрации можно разделить на пять различных частей:

1. Чтение первых mh элементов с входа (f_0) и размещение их в верхней половине пирамиды, где не требуется никакой упорядоченности ключей.

2. Чтение следующих mh элементов и расстановка их в нижней половине пирамиды, причем каждый элемент просеивается на свое место (построение пирамиды).

3. Индексу L присваивается значение m, и для всех оставшихся элементов f_0 проделываются следующие шаги: H_1 пересылается в соответствующую выходную последовательность. Если его ключ меньше или равен ключу следующего элемента входной последовательности, то этот элемент принадлежит той же самой серии и его можно просеять на соответствующее место. В противном случае надо сократить размер пирамиды и поместить новый элемент во вторую, верхнюю пирамиду, которая строится для следующей серии. Границей между пирамидами служит индекс L. Таким образом, нижняя (текущая) пирамида состоит из элементов $H_1 \ldots H_L$, а верхняя (следующая) — из $H_{L+1} \ldots H_m$. Если $L = 0$, то выход переключается и m вновь присваивается L.

4. Вход исчерпан. Первое, присваивание R значе-

ния m, затем сброс нижней части, заканчивающей текущую серию и одновременное построение верхней части и постепенное ее перемещение в позиции H_{L+1} ... H_R.

5. Формирование из оставшихся элементов пирамиды последней серии.

Теперь мы можем подробно описать эти пять шагов в виде законченной программы, которая в случае конца серии, когда нужно изменить индекс выходной последовательности, обращается к процедуре *select*. В прогр. 2.17 вместо нее используется «пустая» процедура («заглушка»), просто считающая число сформированных серий. Все элементы записываются в последовательность f_1.

Если теперь мы попытаемся объединить эту программу, например, с многофазной сортировкой, то встретимся с серьезными затруднениями. Начальная часть программы сортировки включает довольно сложную процедуру переключения переменных-последовательностей и рассчитана на использование процедуры *copyrun*, которая записывает в выбранную последовательность точно одну серию. Программа же *Heapsort* в свою очередь сложная процедура, рассчитанная на обособленную процедуру *select*, просто выбирающую новое направление пересылки. Не было бы никаких проблем, если бы в одной (либо двух) программах требуемая процедура вызывалась в одном-единственном месте, но ведь она вызывается в нескольких местах обеих программ.

В такой ситуации лучше всего воспользоваться так называемыми *сопрограммами*, при сосуществовании нескольких процессов именно их и надо использовать. Наиболее типичный случай — совместная работа процесса, порождающего поток информации для различных объектов и некоторого процесса, потребляющего этот поток. Взаимоотношение производителя — потребителя легче всего выразить в терминах сопрограмм, одна из них может быть самóй главной программой. Сопрограмму можно рассматривать как процесс, имеющий одну или несколько точек разрыва. Если встречается такая точка разрыва, то управление возвращается к программе, активировавшей дан-

ную сопрограмму. Если происходит новое обращение к сопрограмме, то выполнение возобновляется с этой точки разрыва. В нашем случае Polyphase Sort можно было бы рассматривать как главную программу, обращающуюся к *copyrun*, которая оформлена как сопрограмма. Она состоит из основного тела прогр. 2.17, в котором каждое обращение к *select* воспринимается как точка разрыва. Проверку на конец файла теперь следует везде заменить на проверку достижения сопрограммой концевой точки.

Анализ и заключение. Какой же производительности можно ожидать от многофазной сортировки с начальным распределением серий с помощью Heapsort? Сначала разберем, какие улучшения можно получить от введения пирамиды.

В последовательности со случайно распределенными ключами средняя длина серий равна 2. Какова же длина серий, профильтрованных через пирамиду размером m? Казалось бы, она должна быть m, но, к счастью, вероятностный анализ дает значительно лучший результат, а именно — 2m (см. Кнут, т. 2, с. 304). Поэтому ожидаемый коэффициент улучшения равен m.

Оценку производительности многофазной сортировки можно получить из табл. 2.15, где указано максимальное число начальных серий, которые можно отсортировать за заданное число частичных проходов (уровней) с заданным числом (N) последовательностей. Например, с шестью последовательностями и пирамидой размера m = 100 можно за 10 частичных проходов отсортировать файл, содержащий до 165 680 100 начальных серий *). Это замечательная производительность.

Возвращаясь вновь к комбинации многофазной сортировки с Heapsort, нельзя не поразиться ее сложности. А ведь она решает ту же самую просто сформулированную задачу упорядочения множества элементов, что и любая из небольших программок, осно-

*) В действительности за 10 проходов будет отсортировано только 192 100 серий, а указанное число серий — за 20 проходов. — *Прим. перев.*

```
MODULE Distribute; (* с помощью Heapsort *)
  FROM InOut IMPORT WriteInt, WriteLn;
  FROM FS IMPORT File, Open, ReadWord, WriteWord, Reset, Close;
  CONST m = 16; mh = m DIV 2; (* размер пирамиды *)
  TYPE item = INTEGER;

  VAR L, R: CARDINAL;
    count: CARDINAL;
    x: item;
    H: ARRAY [1 .. m] OF item; (* пирамида *)
    f0, f1: File;

  PROCEDURE select;
  BEGIN count := count + 1 (* пусто *)
  END select;

  PROCEDURE sift(L, R: CARDINAL);
    VAR i, j: CARDINAL; x: item;
  BEGIN i := L; j := 2*L; x := H[L];
    IF (j < R) & (H[j] > H[j+1]) THEN j := j+1 END;
    WHILE (j <= R) & (x > H[j]) DO
      H[i] := H[j]; i := j; j := 2*j;
      IF (j < R) & (H[j] > H[j+1]) THEN j := j+1 END
    END;
    H[i] := x
  END sift;

  PROCEDURE OpenRandomSeq(VAR s: File; length, seed: INTEGER);
    VAR i: INTEGER;
  BEGIN Open(s);
    FOR i := 0 TO length-1 DO
      WriteWord(s, seed); seed := (31*seed) MOD 997 + 5
    END
  END OpenRandomSeq;

  PROCEDURE List(VAR s: File);
    VAR i, L: CARDINAL;
  BEGIN Reset(s); i := 0; L := s.length;
    WHILE i < L DO
      WriteInt(INTEGER(s.a[i]), 6); i := i+1;
      IF i MOD 10 = 0 THEN WriteLn END
    END;
    WriteLn
```

```
  END List;
BEGIN count := 0; OpenRandomSeq(f0, 200, 991); List(f0);
  Open(f1); Reset(f0);
  select;
(* этап 1: заполнение верхней части пирамиды *)
  L := m;
  REPEAT ReadWord(f0, H[L]); L := L-1
  UNTIL L = mh;
(* этап 2: заполнение нижней части пирамиды*)
  REPEAT ReadWord(f0, H[L]); sift(L,m); L := L-1
  UNTIL L = 0;
(* этап 3: пропуск элементов через пирамиду *)
  L := m; ReadWord(f0, x);
  WHILE ~f0.eof DO
    WriteWord(f1, H[1]);
    IF H[1] <= x THEN
      (*x принадлежит той же серии *) H[1] := x; sift(1,L)
    ELSE (* начало очередной серии *)
      H[1] := H[L]; sift(1, L-1); H[L] := x;
      IF L <= mh THEN sift(L,m) END ;
      L := L-1;
      IF L = 0 THEN
        (* пирамида полна; начало новой серии.*) L := m; select
      END
    END ;
    ReadWord(f0, x)
  END ;
(* этап 4: сброс нижней части пирамиды *)
  R := m;
  REPEAT WriteWord(f1, H[1]);
    H[1] := H[L]; sift(1, L-1); H[L] := H[R]; R := R-1;
    IF L <= mh THEN sift(L,R) END ;
    L := L-1
  UNTIL L = 0;
(* этап 5: сброс верхней части пирамиды *)
  select;
  WHILE R > 0 DO
    WriteWord(f1, H[1]); H[1] := H[R]; R := R-1; sift(1,R)
  END ;
  List(f1);
  Close(f0); Close(f1)
END Distribute.
```

Прогр. 2.17. Распределение начальных серий с помощью пирамиды.

ванных на простых принципах сортировки массива. Поэтому и вся эта глава была написана, чтобы продемонстрировать следующее:

1. Между алгоритмом и структурой данных, с которыми он работает, существует очень глубокая связь: структура данных влияет на вид программы.

2. Усложняя программу, можно улучшить ее производительность, даже если структура данных (последовательности, а не массивы) для этого не особенно подходит.

УПРАЖНЕНИЯ

2.1. Какие из алгоритмов, представленных программами 2.1—2.6, 2.8, 2.10 и 2.13, обеспечивают устойчивую сортировку?

2.2. Будет ли программа 2.2 правильно работать, если в заголовке цикла $L \leqslant R$ заменить на $L < R$? Будет ли она правильно работать, если операторы $R := m - 1$ и $L := m + 1$ упростить до $L := m$ и $R := m$? Если нет, то приведите последовательность $a_1 \ldots a_n$, на которой измененная программа будет работать неправильно.

2.3. Запрограммируйте три метода прямой сортировки и измерьте на вашей машине время их работы. Определите коэффициенты, на которые надо умножать C и M, чтобы получить оценки для реального времени работы.

2.4. Определите инварианты в трех методах прямой сортировки.

2.5. Перед вами «очевидная» версия программы 2.9. Найдите последовательность значений $a_1 \ldots a_n$, при которой программа работает неправильно.

```
i := 1; j := n; 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 программу, где алгоритмы быстрой и пузырьковой сортировки скомбинированы следующим образом. Quicksort используется для получения подпоследовательности (неотсортированной) длины m ($1 \leqslant m \leqslant n$), а заканчивается сортировка методом пузырька. Заметим, что последняя сортировка может проходить по всему массиву из n элементов, при этом минимизируется управление. Найдите значение m, минимизирующее общее время сортировки. Замечание: ясно, что оптимальное значение m будет достаточно мало, поэтому можно позволить себе точно $m - 1$ проходов по массиву, а не тратить последний проход на фиксацию того факта, что никаких обменов не было.

2.7. Проведите такие же эксперименты, как и в упр. 2.6, использовав вместо пузырьковой сортировки метод прямого выбора. Конечно, эта сортировка не может проходить по всему массиву, поэтому ожидаемое время манипуляции с индексами будет несколько больше.

2.8. Напишите рекурсивную программу Quicksort, исходя из предусловия, что меньший подмассив обрабатывается раньше большего. Первое действие выполните с помощью итерации, а для второго используйте рекурсивное обращение. (Таким образом, ваша программа сортировки будет содержать одно рекурсивное обращение в отличие от программы 2.10, содержащей два обращения, и программы 2.11, где их вообще нет.)

2.9. Найдите перестановку ключей 1, 2, ..., n, при которой быстрая сортировка ведет себя наихудшим (наилучшим) образом ($n = 5, 6, 7$).

2.10. Напишите программу естественного слияния, аналогичную программе простого слияния (прогр. 2.13) и работающую с массивом удвоенного размера, причем процесс идет с двух концов к середине. Сравните производительность вашей программы с производительностью прогр. 2.13.

2.11. Обратите внимание, что при двухпутевом естественном слиянии мы не просто слепо выбираем из имеющихся ключей наименьший, а когда встречается конец серии, то остаток второй серии копируется в выходную последовательность. Например, слияние двух последовательностей

2, 4, 5, 1, 2, ...
3, 6, 8, 9, 7, ...

дает последовательность

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

а не последовательность

2, 3, 4, 5, 1, 2, 6, 8, 9, ...

которая выглядит более упорядоченной. Почему мы остановились на такой стратегии?

2.12. Существует метод сортировки, похожий на многофазную сортировку — каскадное слияние [2.1 и 2.9]. При нем слияние идет так: если, например, речь идет о шести последовательностях, то, начиная с «идеального распределения серий» на последовательностях T1, ..., T6, сначала проводится пятипутевое слияние T1 ... T5 на T6, до тех пор пока не станет пустой последовательность T5, затем (T6 уже не трогается) проводится четырехпутевое слияние на T5, затем трехпутевое на T4, двухпутевое на T3 и копирование из T1 в T2. Следующий проход работает по аналогичной схеме: все начинается с пятипутевого слияния на T1 и т. д. Хотя такая схема выглядит хуже многофазного слияния, поскольку некоторые последовательности «простаивают», и есть просто операции копирования, тем не менее она, что удивительно, оказывается лучше многофазной сортировки при очень больших файлах и шести или более последо-

вательностях. Напишите программу такого каскадного слияния с четко выделенной структурой.

ЛИТЕРАТУРА

[2.1] Betz B. K. 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. [Имеется перевод: Кнут Д. Искусство программирования для ЭВМ, т. 3. — М.: Мир, 1978.]
[2.8] — » — » —, 108—119.
[2.9] — » — » —, 342.
[2.10] Lorin H. A Guided Bibliography to Sorting. *IBM Syst. J.*, 10, No. 3, (1971), 244—254.
[2.11] Shell D. L. A Highspeed Sorting Procedure. *Comm. ACM*, 2, No. 7, (1959), 30—32.
[2.12] Singleton R. C. An Efficient Algorithm for Sorting with Minimal Storage (Algorithm 347). *Comm. ACM*, **12**, No. 3, (1969), 185.
[2.13] Van Emden M. H. Increasing the Efficiency of Quicksort (Algorithm 402). *Comm. ACM*, **13**, No. 9, (1970), 563—566, 693.
[2.14] Williams J. W. J. Heapsort (Algorithm 232). *Comm. ACM*, 7, No. 6, (1964), 347—348.

3. РЕКУРСИВНЫЕ АЛГОРИТМЫ

3.1. ВВЕДЕНИЕ

Рекурсивным называется объект, частично состоящий или определяемый с помощью самого себя. Рекурсия встречается не только в математике, встречается она и в повседневной жизни. Кто не видел рекламной картинки, содержащей свое собственное изображение?

В частности, рекурсивные определения представляют собой мощный аппарат в математике. Вот несколько общеизвестных примеров: натуральные числа, деревья и определенные функции.

1. Натуральные числа:

(a) 0 есть натуральное число,

(b) число, следующее за натуральным, есть натуральное число.

2. Деревья:

(a) 0 есть дерево (его называют «пустым деревом»),

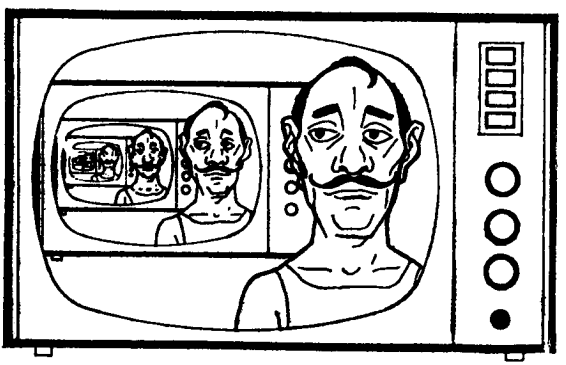


Рис. 3.1. Рекурсивное отображение.

(b) если t1 и t2 — деревья, то построение, содержащее вершину с двумя ниже расположенными деревьями, опять же дерево.

3. Функция «факториал» n! (для неотрицательных целых чисел).

(a) $0! = 1$,

(b) $n > 0$: $n! = n*(n-1)!$

Мощность рекурсивного определения заключается, очевидно, в том, что оно позволяет с помощью конечного высказывания определить бесконечное множество объектов. Аналогично с помощью конечной рекурсивной программы можно описать бесконечное вычисление, причем программа не будет содержать явных повторений. Однако рекурсивные алгоритмы лучше всего использовать, если в решаемой задаче, вычисляемой функции или структуре обрабатываемых данных рекурсия уже присутствует явно. В общем виде рекурсивную программу P можно выразить как некоторую композицию **P** из множества операторов **S** (не содержащих P) и самой P:

$$P \equiv \mathbf{P}[\mathbf{S}, P]$$

Для выражения рекурсивных программ необходимо и достаточно иметь понятие процедуры или подпрограммы, поскольку они позволяют дать любому оператору имя, с помощью которого к нему можно обращаться. Если некоторая процедура P содержит явную ссылку на саму себя, то ее называют *прямо рекурсивной*, если же P ссылается на другую процедуру Q, содержащую (прямую и косвенную) ссылку на P, то P называют *косвенно рекурсивной*. Поэтому по тексту программы рекурсивность не всегда явно определима.

Как правило, с процедурой связывают множество локальных объектов, т. е. множество переменных, констант, типов и процедур, которые определены только в этой процедуре и вне ее, не существуют или не имеют смысла. При каждой рекурсивной активации такой процедуры порождается новое множество локальных, связанных переменных. Хотя они имеют те же самые имена, что и соответствующие элементы локального множества предыдущего «поколения»

этой процедуры, их значения отличны от последних, а любые конфликты по именам разрешаются с помощью правил, определяющих область действия идентификаторов: идентификатор всегда относится к самому последнему порожденному множеству переменных. Это же правило справедливо и для параметров процедуры, по определению связанных с самой процедурой.

Подобно операторам цикла, рекурсивные процедуры могут приводить к незаканчивающимся вычислениям, и поэтому на эту проблему следует особо обратить внимание. Очевидно основное требование, чтобы рекурсивное обращение к P управлялось некоторым условием B, которое в какой-то момент становится ложным. Поэтому более точно схему рекурсивных алгоритмов можно представить в любой из следующих форм:

$$P \equiv \text{IF B THEN P[S,P] END} \tag{3.2}$$

$$P \equiv \text{P[S,IF B THEN P END]} \tag{3.3}$$

Основной способ доказательства конечности некоторого повторяющегося процесса таков:

1. Определяется функция $f(x)$ (x — множество переменных), такая, что из $f(x) \leqslant 0$ следует истинность условия окончания цикла (с предусловием или постусловием).

2. Доказывается, что при каждом прохождении цикла $f(x)$ уменьшается.

Аналогично доказывается и окончание рекурсии — показывается, что P уменьшает $f(x)$, такую, что из $f(x) \leqslant 0$ следует $\sim B$. В частности, наиболее надежный способ обеспечить окончание процедуры — ввести в нее некоторый параметр (значение), назовем его n, и при рекурсивном обращении к P в качестве параметра задавать $n - 1$. Если в этом случае в качестве B используется $n > 0$, то окончание гарантировано. Это опять же выражается двумя схемами:

$$P(n) \equiv \text{IF } n > 0 \text{ THEN P[S,P}(n-1)\text{] END} \tag{3.4}$$

$$P(n) \equiv \text{P[S,IF } n > 0 \text{ THEN P}(n-1)\text{ END]} \tag{3.5}$$

В практических приложениях важно убедиться, что максимальная глубина рекурсий не только конечна, но и достаточно *мала*. Дело в том, что каждая

рекурсивная активация процедуры P требует памяти для размещения ее переменных. Кроме этих локальных переменных нужно еще сохранять текущее «состояние вычислений», чтобы можно было вернуться в него по окончании новой активации P. С такой ситуацией мы уже встречались в гл. 2 при разработке процедуры Quicksort (быстрая сортировка). Мы обнаружили, что при прямолинейном составлении программы, разделяющей n элементов на две части, и двух рекурсивных обращениях для сортировки этих частей в худшем случае глубина рекурсии может достигать n. При разумных подходах оказалось, что можно ограничить глубину числом log n. Разница между значениями n и log n вполне достаточна, чтобы от случая, крайне не подходящего для применения рекурсивной процедуры, перейти к случаю, когда оно (применение) имеет практический смысл.

3.2. КОГДА РЕКУРСИЮ ИСПОЛЬЗОВАТЬ НЕ НУЖНО

Рекурсивные алгоритмы особенно подходят для задач, где обрабатываемые данные определяются в терминах рекурсии. Однако это не означает, что такое рекурсивное определение данных гарантирует бесспорность употребления для решения задачи рекурсивного алгоритма. Фактически объяснение концепций рекурсивных алгоритмов на неподходящих для этого примерах и вызвало широко распространенное предубеждение против использования рекурсий в программировании; их даже сделали синонимом неэффективности.

Программы, в которых следует избегать алгоритмической рекурсии, можно охарактеризовать некоторой схемой, отражающей их строение. Это схема (3.6) и эквивалентная ей схема (3.7). Специфично, что в них есть единственное обращение к P в конце (или начале) всей конструкции.

P ≡ IF B THEN S; P END (3.6)

P ≡ S; IF B THEN P END (3.7)

Такие схемы естественны в ситуациях, где вычисляемые значения определяются с помощью простых рекуррентных отношений. Возьмем хорошо известный пример вычисления факториала

$$i = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ \ldots \tag{3.8}$$
$$f_i = 1,\ 1,\ 2,\ 6,\ 24,\ 120,\ \ldots$$

Первое из чисел определяется явно — $f_0 = 1$, а последующие определяются рекурсивным образом с помощью предшествующего числа:

$$f_{i+1} = (i + 1) * f_i \tag{3.9}$$

Такое рекуррентное отношение предполагает рекурсивный алгоритм вычисления n-го факториального числа. Если мы введем две переменные I и F, обозначающие i и f_i на i-м уровне рекурсии, то обнаружим, что для перехода к следующему числу последовательности (3.8) нужно проделать такие вычисления:

$$I := I + 1;\ F := I * F \tag{3.10}$$

и, подставляя (3.10) вместо S в (3.6), получаем рекурсивную программу

```
P ≡ IF I < n THEN I := I + 1; F := I * F; P END
I := 0; F := 1; P                                    (3.11)
```

В принятых нами обозначениях первую строчку (3.11) можно переписать так:

```
PROCEDURE P;
BEGIN
  IF I < n THEN I := I + 1; F := I * F; P END
END P                                                (3.12)
```

В (3.13) приведена более часто употребляемая и полностью эквивалентная ей запись, где вместо P приведена так называемая *процедура-функция*, т. е. процедура, с которой связывается значение-результат и которую поэтому можно применять прямо как составляющую выражения. В этом случае переменная F становится излишней, а роль I явно выполняет

параметр процедуры.

```
PROCEDURE F (I : INTEGER) : INTEGER;
BEGIN
IF I > 0 THEN RETURN I*F (I — 1) ELSE RETURN 1 END
END F                                                    (3.13)
```

Теперь уже ясно, что в нашем примере рекурсия крайне просто заменяется итерацией. Это проделано в такой программе:

```
I := 0; F := 1;                                          (3.14)
WHILE I < n DO I := I + 1; F := I * F END
```

В общем-то программы, построенные по схемам (3.6) и (3.7), нужно переписывать, руководствуясь схемой

```
P ≡ [x := x0; WHILE B DO S END]                          (3.15)
```

Конечно, существуют и более сложные схемы рекурсий, которые можно и необходимо переводить в итеративную форму. Возьмем в качестве примера вычисление *чисел Фибоначчи*, определяемых рекурсивным соотношением

$$\mathrm{fib}_{n+1} = \mathrm{fib}_n + \mathrm{fib}_{n-1} \quad \text{для} \quad n > 0 \qquad (3.16)$$

и $\mathrm{fib}_1 = 1$, $\mathrm{fib}_0 = 0$. Прямое, «наивное» переписывание приводит к рекурсивной программе:

```
PROCEDURE Fib(n: INTEGER): INTEGER;                      (3.17)
BEGIN
  IF n = 0 THEN RETURN 0
  ELSIF n = 1 THEN RETURN 1
  ELSE RETURN Fib(n-1) + Fib(n-2)
  END
END Fib
```

Вычисление fib_n путем обращения к Fib(n) приводит к рекурсивным активациям этой процедуры-функции. Как часто это происходит? Каждое обращение с $n > 1$ приводит еще к двум обращениям, т. е. общее число вызовов растет экспоненциально (см. рис. 3.2). Ясно, что такая программа практического интереса не представляет.

Однако, к счастью, числа Фибоначчи можно вычислять и по итеративной схеме, где с помощью вспо-

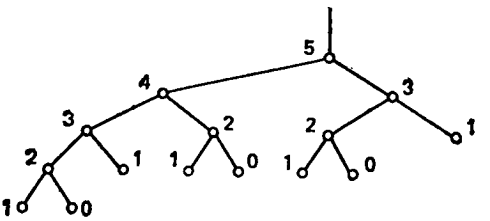


Рис. 3.2. 15 обращений к Fib(n) при n = 5.

могательных переменных $x = fib_i$ и $y = fib_{i-1}$ удается избежать повторных вычислений одной и той же величины:

```
i := 1; x := 1; y := 0;                                    (3.18)
WHILE i < n DO z := x; x := x + y; y := z; i := i + 1 END
```

Замечание: Присваивание переменным x, y, z можно записать с помощью двух операторов, не используя вообще переменной *z*: x := x + y; y := x − y.

Итак, урок таков: следует избегать рекурсий там, где есть очевидное итерационное решение. Однако это не означает, что от рекурсий следует избавляться любой ценой. Существует много хороших примеров применения рекурсии, что мы и продемонстрируем в последующих разделах и главах. То, что существуют реализации рекурсивных процедур на фактически нерекурсивных машинах, доказывает, что для практических целей любую рекурсивную программу можно трансформировать в чисто итеративную. Однако это требует явной работы со стеком для рекурсий, причем эти действия часто настолько затуманивают суть программы, что ее бывает трудно понять. Вывод: алгоритмы, рекурсивные по своей природе*), а не итеративные, нужно формулировать как рекурсивные процедуры. Для лучшего понимания читателю предлагается сравнить прогр. 2.10 и 2.11.

Оставшуюся часть главы мы посвятим разработке

*) Здесь необходимо уточнить выражение «по своей природе». Скажем, факториал рекурсивен по своей природе или итеративен? — *Прим. перев.*

некоторых рекурсивных программ для задач, где использование этих рекурсий вполне оправданно. Кроме того, в гл. 4 мы также будем интенсивно пользоваться рекурсивными методами, если, конечно, структура данных будет делать их естественными и очевидными.

3.3. ДВА ПРИМЕРА РЕКУРСИВНЫХ ПРОГРАММ

Симпатичный узор на рис. 3.5 состоит из суперпозиции пяти кривых. Эти кривые соответствуют некоторому регулярному образу, и считается, что их можно нарисовать на экране дисплея или на графопостроителе под управлением какой-либо вычислительной машины. Наша задача — найти рекурсивную схему, по которой можно было бы создать программу для такого рисования. Рассматривая рисунок, мы обнаруживаем, что три наложенные друг на друга кривые имеют форму, показанную на рис. 3.3, обозначим их через H_1, H_2 и H_3. Видно, что H_{i+1} получается соединением четырех экземпляров H_i вдвое меньшего размера, повернутых соответствующим образом и «стянутых» вместе тремя прямыми линиями. Заметим, что H_1 можно считать состоящей из четырех вхождений пустой H_0, соединенных этими же тремя линиями. Кривая H_i называется *кривой Гильберта* i-го порядка в честь ее первооткрывателя Д. Гильберта (1891 г.).

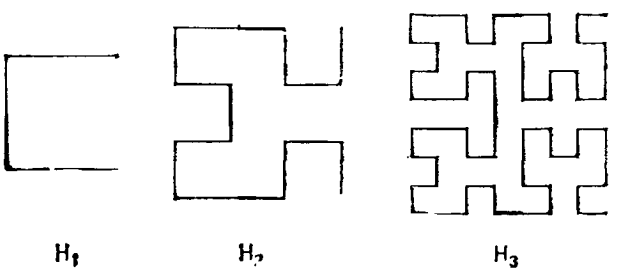


Рис. 3.3. Кривые Гильберта первого, второго и третьего порядков.

Поскольку каждая кривая H_i состоит из четырех вдвое меньших H_{i-1}, то процедура для рисования H_i будет включать четыре обращения для рисования H_{i-1}, соответствующим образом повернутых и уменьшенных вдвое. Для наглядности мы обозначим эти четыре части через A, B, C и D, а процедуры, рисующие соединительные прямые, будем обозначать стрелками, указывающими в соответствующем направлении. В этом случае появляется такая схема рекурсий (рис. 3.3).

```
A:  D← A↓ A→ B
B:  C↑ B→ B↓ A
C:  B→ C↑ C← D
D:  A↓ D← D↑ C
```
(3.19)

Представим себе, что для рисования части прямой в нашем распоряжении есть процедура *line*, передвигающая перо в заданном направлении на заданное расстояние. Для удобства будем предполагать, что направление задается целым параметром i как 45*i градусов. Если единичную длину прямой обозначить через u, то с помощью рекурсивных обращений к аналогично составленным процедурам для B и D и к самой A легко написать процедуру, соответствующую схеме A.

```
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.
```
(3.20)

Эта процедура инициируется главной программой по одному разу для каждой из кривых Гильберта, образующих приведенный рисунок. Главная программа определяет начальную точку кривой, т. е. исходные координаты пера (Px и Py) и единичное приращение u. Квадрат, где рисуется кривая, помещается в середине страницы, заданной шириной и высотой.

```
MODULE Hilbert;
 FROM Terminal IMPORT Read;
 FROM LineDrawing IMPORT width, height, Px, Py, clear, line;
 CONST SquareSize = 512;
 VAR i,x0,y0,u: CARDINAL; ch: CHAR;
 PROCEDURE A(i: CARDINAL);
 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;
 PROCEDURE B(i: CARDINAL);
 BEGIN
  IF i > 0 THEN
  C(i-1); line(2,u); B(i-1); line(0,u);
  B(i-1); line(6,u); A(i-1)
 END
END B;
PROCEDURE C(i: CARDINAL);
BEGIN
 IF i > 0 THEN
  B(i-1); line(0,u); C(i-1); line(2,u);
  C(i-1); line(4,u); D(i-1)
 END
END C;
PROCEDURE D(i: CARDINAL);
BEGIN
 IF i > 0 THEN
  A(i-1); line(6,u); D(i-1); line(4,u);
  D(i-1); line(2,u); C(i-1)
 END
END D;
BEGIN clear;
 x0 := width DIV 2; y0 := height DIV 2;
 u := SquareSize; i := 0;
 REPEAT i := i+1; u := u DIV 2;
  x0 := x0 + (u DIV 2); y0 := y0 + (u DIV 2);
  Px := x0; Py := y0; A(i); Read(ch)
 UNTIL (ch = 33C) OR (i = 6);
 clear
END Hilbert.
```

Прогр. 3.1. Кривые Гильберта.

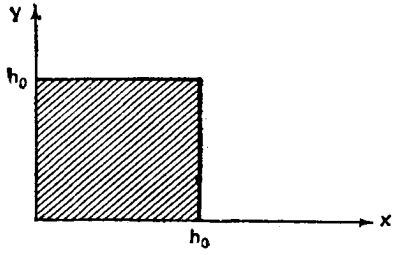


Рис. 3.4. Рамка для кривых.

Эти параметры, так же как и процедура рисования прямой, находятся в модуле с названием *LineDrawing*. Программа рисует n кривых Гильберта — H_1...H_n (см. прогр. 3.1 и рис. 3.4)

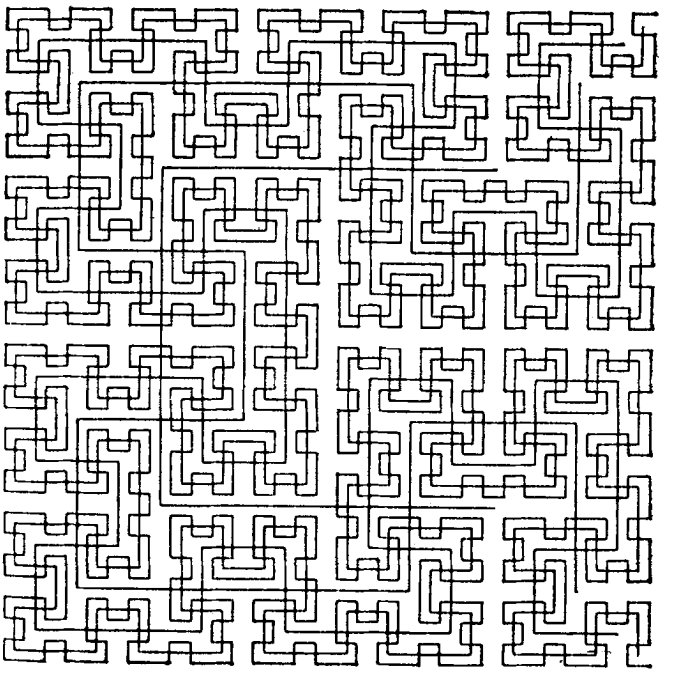


Рис. 3.5. Кривые Гильберта H_1 ... H_5.

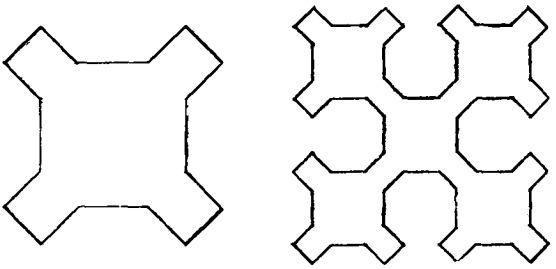


Рис. 3.6. Кривые Серпинского первого и второго порядка.

Подобный, но более сложный и эстетически утонченный пример приводится на рис. 3.7. Он также получен путем наложения друг на друга нескольких кривых. Первые две из них показаны на рис. 3.6. Кривая S_i называется *кривой Серпинского* i-го порядка. Какова же рекурсивная схема этих кривых? Попробуем в качестве основного строительного блока взять S_i, возможно, без одного ребра. Это не приведет нас к решению. Главное отличие кривой Серпинского от кривой Гильберта в том, что первая замкнута (и в ней нет пересечений). Это означает, что основная рекурсивная схема должна давать разомкнутую кривую, четыре части которой соединяются линиями, не принадлежащими самому рекурсивному образу. И действительно, эти замыкающие линии представляют собой отрезки прямых в четырех внешних углах, на рис. 3.6 они выделены жирными линиями. Можно считать, что они принадлежат к непустой начальной кривой S — квадрату, «стоящему» на одном углу. Теперь легко составить рекурсивную схему. Четыре составляющих образа вновь обозначим через A, B, C, D, а связывающие линии будем рисовать явно. Обратите внимание: четыре рекурсивных образа по существу идентичны, но лишь повертываются на 90°.

Основной образ кривых Серпинского задается схемой

S: A↘ B↙ C↖ D↗ (3.21)

а рекурсивные составляющие (горизонтальные и вертикальные отрезки — двойной длины):

```
A: A↘ B→ D↗ A
B: B↙ C↓ A↘ B
C: C↖ D← B↙ C
D: D↗ A↑ C↖ D
```
(3.22)

Если использовать те же примитивы рисования, что и в случае кривых Гильберта, то приведенные рекурсивные схемы без труда трансформируются в (прямо или косвенно) рекурсивный алгоритм.

```
PROCEDURE A(k: INTEGER);                                    (3.23)
BEGIN
 IF k > 0 THEN
  A(k-1); line(7, h); B(k-1); line(0, 2*h);
  D(k-1); line(1, h); A(k-1)
 END
END A
```

Эта процедура порождается из первой строки рекурсивной схемы (3.22). Аналогично получаются и процедуры, соответствующие образам B, C и D. Главная программа строится по образу (3.21). Ее задача — установить начальные значения для координат рисунка и задать единичную длину линий h, — это все зависит от формата бумаги (прогр. 3.2). На рис. 3.7 приведен результат работы такой программы с $n = 4$.

Ясно, что в этих примерах рекурсия используется элегантным способом. Правильность программ легко подтверждается их структурой и формируемыми образами. Кроме того, употребление в соответствии со схемой 3.5 явного параметра для уровня гарантирует окончание работы, так как глубина рекурсии не может быть больше n. По сравнению с таким рекурсивным построением эквивалентные программы, где избегали употребления рекурсии, выглядят крайне сложными и запутанными. Попытайтесь понять программы, приведенные в работе [3.3], и вы сами в этом убедитесь.

```
MODULE Sierpinski;
 FROM Terminal IMPORT Read;
 FROM LineDrawing IMPORT width, height, Px, Py, clear, line;
 CONST SquareSize = 512;
 VAR i,h,x0,y0: CARDINAL; ch: CHAR;
 PROCEDURE A(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   A(k-1); line(7, h); B(k-1); line(0, 2*h);
   D(k-1); line(1, h); A(k-1)
  END
 END A;
 PROCEDURE B(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   B(k-1); line(5, h); C(k-1); line(6, 2*h);
   A(k-1); line(7, h); B(k-1)
  END
 END B;
 PROCEDURE C(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   C(k-1); line(3, h); D(k-1); line(4, 2*h);
   B(k-1); line(5, h); C(k-1)
  END
 END C;
 PROCEDURE D(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   D(k-1); line(1, h); A(k-1); line(2, 2*h);
   C(k-1); line(3, h); D(k-1)
  END
 END D;
BEGIN clear; i := 0; h := SquareSize DIV 4;;
 x0 := CARDINAL(width) DIV 2; y0 := CARDINAL(height) DIV 2 + h;
 REPEAT i := i+1; x0 := x0-h;
  h := h DIV 2; y0 := y0+h; Px := x0; Py := y0;
  A(i); line(7,h); B(i); line(5,h);
  C(i); line(3,h); D(i); line(1,h); Read(ch)
 UNTIL (i = 6) OR (ch = 33C);
 clear
END Sierpinski.
```

Прогр. 3.2. Кривые Серпинского.

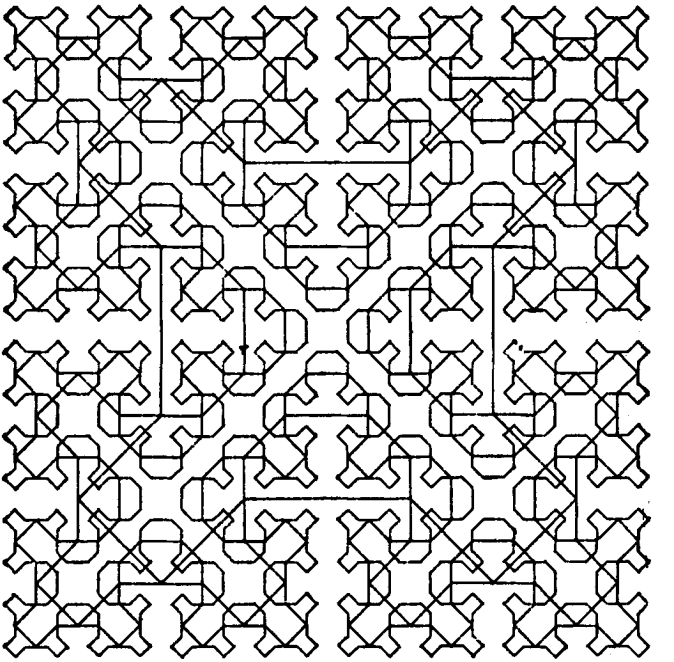


Рис. 3.7. Кривые Серпинского S_1 ... S_4.

3.4. АЛГОРИТМЫ С ВОЗВРАТОМ

Особенно интригующая область программирования — задачи так называемого «искусственного интеллекта». Здесь мы имеем дело с алгоритмами, ищущими решение не по заданным правилам вычислений, а путем проб и ошибок. Обычно процесс проб и ошибок разделяется на отдельные задачи. Часто эти задачи наиболее естественно выражаются в терминах рекурсии и требуют исследования конечного числа подзадач. В общем виде весь процесс можно мыслить как процесс поиска, строящий (и обрезающий) дерево подзадач. Во многих проблемах такое дерево поиска растет очень быстро, рост зависит от параметров задачи и часто бывает экспоненциальным. Соответственно увеличивается и стоимость поиска. Иногда, используя некоторые эвристики, дерево поиска

удается сократить и тем самым свести затраты на вычисления к разумным пределам.

Мы не ставили своей целью обсуждение в этой книге общих правил эвристики. Более того, сейчас мы исходим лишь из принципа, что задача искусственного интеллекта разбивается на подзадачи, и обсуждаем применение при этом рекурсии. Начнем с демонстрации основных методов на хорошо известном примере — задаче о *ходе коня.*

Дана доска размером $n \times n$, т. е. содержащая n^2 полей. Вначале на поле с координатами x_0, y_0 помещается конь — фигура, перемещающаяся по обычным шахматным правилам. Задача заключается в поиске последовательности ходов (если она существует), при которой конь точно один раз побывает на всех полях доски (обойдет доску), т. е. нужно вычислить $n^2 - 1$ ходов.

Очевидный прием упростить задачу обхода n^2 полей — решать более простую: либо выполнить очередной ход, либо доказать, что никакой ход не возможен. Поэтому начнем с определения алгоритма выполнения очередного хода. Первый его вариант приведен в (3.24).

```
PROCEDURE TryNextMove;                                          (3.24)
BEGIN инициализация выбора хода;
  REPEAT выбор очередного кандидата из списка ходов;
    IF подходит THEN
      запись хода;
      IF доска не заполнена THEN
        TryNextMove;
        IF неудача THEN уничтожение предыдущего хода END
      END
    END
  UNTIL (был удачный ход) OR (кандидатов больше нет)
END TryNextMove
```

Если мы хотим описать этот алгоритм более детально, то нужно выбрать некоторое представление для данных. Доску, самое простое, можно представлять как матрицу, назовем ее h. Введем, кроме того, тип для индексирующих значений:

```
TYPE index = [1 .. n];                                          (3.25)
VAR h: ARRAY index, index OF INTEGER
```

Из-за того что мы хотим знать историю продвижения по доске, поля ее будем представлять целыми числами, а не булевскими значениями, позволяющими лишь определять занятость поля. Очевидно, можно остановиться на таких соглашениях:

h [x, y] = 0: поле ⟨x, y⟩ еще не посещалось (3.26)

h [x, y] = i поле ⟨x, y⟩ посещалось на i-м ходу

Теперь нужно выбрать соответствующие параметры. Они должны определять начальные условия следующего хода и результат (если ход сделан). В первом случае достаточно задавать координаты поля (x, y), откуда следует ход, и число i, указывающее номер хода (для фиксации). Для результата же требуется булевский параметр; если он истина, то ход был возможен.

Какие операторы можно уточнить на основе принятых решений? Очевидно, условие «доска не заполнена» можно переписать как $i < n^2$. Кроме того, если ввести две локальные переменные u и v для возможного хода, определяемого в соответствии с правилами «прыжка» коня, то предикат «допустим» можно представить как логическую конъюнкцию условий, что новое поле находится в пределах доски ($1 \leqslant u \leqslant n$ и $1 \leqslant v \leqslant n$) и еще не посещалось ($h_{uv} = 0$).

Фиксация допустимого хода выполняется с помощью присваивания $h_{uv} := i$, а отмена — с помощью $h_{uv} := 0$. Если ввести локальную переменную q1 и использовать ее в качестве параметра-результата при рекурсивных обращениях к этому алгоритму, то q1

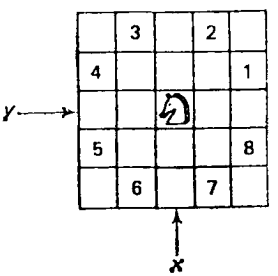


Рис. 3.8. Восемь возможных ходов коня.

можно подставить вместо «есть ход». Так мы приходим к варианту (3.27).

```
PROCEDURE Try(i: INTEGER; x, y: index; VAR q: BOOLEAN);
  VAR u, v: INTEGER; q1: BOOLEAN;
BEGIN инициация выбора хода;
  REPEAT <u,v> – координаты следующего хода,
    определяемого правилами шахмат;
    IF (1 <= u) & (u <= n) & (1 <= v) & (v <= n) & (h[u,v] = 0) THEN
      h[u,v] := i;
      IF i < n*n THEN Try(i+1, u, v, q1);
        IF ~q1 THEN h[u,v] := 0 ELSE q1 := TRUE END
      END
    END
  UNTIL q1 OR других ходов нет;
  q := q1
END Try
```

Еще один шаг детализации — и получим уже программу, полностью написанную в терминах нашего основного языка программирования. Заметим, что до этого момента программа создавалась совершенно независимо от правил, управляющих движением коня. Мы вполне умышленно откладывали рассмотрение частных особенностей задачи. Теперь самое время обратить на них внимание.

Если задана начальная пара координат x, y, то для следующего хода u, v существует восемь возможных кандидатов. На рис. 3.8 они пронумерованы от 1 до 8. Получать u, v из x, y очень просто, достаточно к последним добавлять разности между координатами, хранящиеся либо в массиве разностей, либо в двух массивах, хранящих отдельные разности. Обозначим эти массивы через dx и dy и будем считать, что они соответствующим образом инициированы. Для нумерации очередного хода-кандидата можно использовать индекс k. Подробности показаны в прогр. 3.3. Первый раз к рекурсивной процедуре обращаются с параметрами x0 и y0 — координатами поля, с которого начинается обход. Этому полю должно быть присвоено значение 1, остальные поля маркируются как свободные:

```
h [x0, y0] := 1; try (2, x0, y0, q)
```

```
MODULE KnightsTour;
  FROM InOut IMPORT
    ReadInt, Done, WriteInt, WriteString, WriteLn;
  VAR i, j, n, Nsqr: INTEGER; q: BOOLEAN;
    dx, dy: ARRAY [1 .. 8] OF INTEGER;
    h: ARRAY [1 .. 8], [1 .. 8] OF INTEGER;
  PROCEDURE Try(i, x, y: INTEGER; VAR q: BOOLEAN);
    VAR k, u, v: INTEGER; q1: BOOLEAN;
  BEGIN k := 0;
    REPEAT k := k+1; q1 := FALSE;
      u := x + dx[k]; v := y + dy[k];
      IF (1 <= u) & (u <= n) & (1 <= v) & (v <= n) & (h[u,v] = 0) THEN
        h[u,v] := i;
        IF i < Nsqr THEN Try(i+1, u, v, q1);
          IF ~q1 THEN h[u,v] := 0 END
        ELSE q1 := TRUE
        END
      END
    UNTIL q1 OR (k = 8);
    q := q1
  END Try;
BEGIN
  a[1] := 2; a[2] := 1; a[3] := -1; a[4] := -2;
  a[5] := -2; a[6] := -1; a[7] := 1; a[8] := 2;
  b[1] := 1; b[2] := 2; b[3] := 2; b[4] := 1;
  b[5] := -1; b[6] := -2; b[7] := -2; b[8] := -1;
  LOOP ReadInt(n);
    IF ~Done THEN EXIT END ;
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO h[i,j] := 0 END
    END ;
    ReadInt(i); ReadInt(j); WriteLn;
    Nsqr := n*n; h[1,1] := 1; Try(2, i, j, q);
    IF q THEN
      FOR i := 1 TO n DO
        FOR j := 1 TO n DO WriteInt(h[i,j], 5) END ;
        WriteLn
      END
    ELSE WriteString(" no path"); WriteLn
    END
  END
END KnightsTour.
```

Прогр. 3.3. Обход поля ходом коня.

Нельзя упускать еще одну деталь. Переменная h_{uv} существует только в том случае, если и u, и v лежат в диапазоне индексов 1...n. Следовательно, выражение в (3.27), подставленное вместо «допустим» из (3.24), осмысленно, только если его четыре первые составляющие истинны. Именно поэтому важно, чтобы составляющая $h_{uv} = 0$ была последней. В табл. 3.1 приводятся решения для трех исходных позиций: $\langle 3, 3\rangle$ и $\langle 2, 4\rangle$ при $n = 5$ и $\langle 1, 1\rangle$ при $n = 6$.

Какой же вывод можно сделать из этого примера? Какой схеме мы в нем следовали, типична ли она для алгоритмов поиска решения? Чему этот пример научил нас? Характерное свойство таких алгоритмов заключается в том, что в них делаются шаги в направлении общего решения, но все они фиксируются (записываются) таким образом, что позже можно вернуться вспять, отбрасывая тем самым шаги, которые ведут не к общему решению, а заводят в тупик. Такой процесс называется *возвратом* или *откатом* (backtracking). Если предположить, что число потенциальных шагов конечно, то из схемы (3.24) можно вывести универсальную схему

```
PROCEDURE Try;
BEGIN инициация выбора кандидата;
 REPEAT выбор очередного;                                    (3.28)
  IF подходит THEN
   его запись;
   IF решение неполное THEN Try;
    IF неудача THEN стирание записи END
   END
  END
 UNTIL удача OR кандидатов больше нет
END Try
```

В реальных программах могут, конечно, встречаться и другие варианты, получающиеся из схемы (3.28). Часто, например, встречается построение с явным параметром для уровня, когда указывается глубина рекурсии. Это приводит к простым условиям окончания работы. Более того, если на каждом шаге число исследуемых путей фиксировано (пусть оно равно m), то можно применять еще одну выведенную

Таблица 3.1. Три возможных обхода конем

23	10	15	4	25	23	4	9	14	25
16	5	24	9	14	10	15	24	1	8
11	22	1	18	3	5	22	3	18	13
6	17	20	13	8	16	11	20	7	2
21	12	7	2	19	21	6	17	12	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

схему — (3.29). К ней надо обращаться с помощью оператора Try(1).

```
PROCEDURE Try(i: INTEGER);
  VAR k: INTEGER;
BEGIN k := 0;
  REPEAT k := k+1; выбор k-го кандидата;
    IF подходит THEN                                    (3.29)
      его запись;
      IF i < n THEN Try(i+1);
        IF неудача THEN стирание записи END
      END
    END
  UNTIL удача OR (k = m)
END Try
```

В оставшейся части этой главы мы разберем еще три примера. В них используются различные реализации абстрактной схемы (3.29), и мы хотим еще раз продемонстрировать уместное использование рекурсии.

3.5. ЗАДАЧА О ВОСЬМИ ФЕРЗЯХ

Задача о восьми ферзях — хорошо известный пример использования методов проб и ошибок и алгоритмов с возвратами. В 1850 г. эту задачу исследовал К. Ф. Гаусс, однако полностью он ее так и не решил. Это никого не должно удивлять. Для подобных задач характерно отсутствие аналитического решения. Они требуют огромного количества изнурительной

работы, терпения и аккуратности. Поэтому такие задачи стали почти исключительно прерогативой электронных вычислительных машин, ведь им эти свойства присущи в значительно большей степени, чем человеку, пусть и гениальному.

Задача о восьми ферзях формулируется следующим образом (см. также [3.4]): восемь ферзей нужно расставить на шахматной доске так, чтобы один ферзь не угрожал другому. Воспользовавшись схемой 3.29 как шаблоном, легко получаем грубый вариант решения:

```
PROCEDURE Try(i: INTEGER);
BEGIN
    инициация выбора положения i-го ферзя;
  REPEAT выбор очередного положения;
    IF безопасное THEN поставить ферзя;
      IF i < 8 THEN Try(i+1);
        IF неудача THEN убрать ферзя END
      END
    END
  UNTIL удача OR мест больше нет
END Try
```

Чтобы идти дальше, нужно остановиться на каком-либо представлении для данных. Поскольку из шахматных правил мы знаем, что ферзь бьет все фигуры, находящиеся на той же самой вертикали, горизонтали или диагонали, то заключаем, что на каждой вертикали может находиться один и только один ферзь, поэтому при поиске места для i-го ферзя можно ограничить себя лишь i-й вертикалью. Таким образом, параметр i становится индексом вертикали, а процесс выбора возможного местоположения ограничивается восемью допустимыми значениями для индекса горизонтали j.

Остается решить вопрос: как представлять на доске эти восемь ферзей? Очевидно, доску вновь можно было бы представить в виде квадратной матрицы, но после небольших размышлений мы обнаруживаем, что это значительно усложнило бы проверку безопасности поля. Конечно, подобное решение нежелательно, поскольку такая операция выполняется очень часто. Поэтому хотелось бы остановиться на таком пред-

ставлении данных, которое, насколько это возможно, упростило бы проверку. В этой ситуации лучше всего делать непосредственно доступной именно ту информацию, которая действительно важна и чаще всего используется. В нашем случае это не поля, занятые ферзями, а сведения о том, находится ли уже ферзь на данной горизонтали или диагонали. (Мы уже знаем, что на каждой k-й вертикали ($1 \leqslant k \leqslant i$) стоит ровно один ферзь.) Эти соображения приводят к таким описаниям переменных:

```
VAR x: ARRAY [1 .. 8] OF INTEGER;
    a: ARRAY [1 .. 8] OF BOOLEAN;
    b: ARRAY [b1 .. b2] OF BOOLEAN;
    c: ARRAY [c1 .. c2] OF BOOLEAN;
```
(3.31)

где

x_i обозначает местоположение ферзя на i-й вертикали;

a_j указывает, что на j-й горизонтали ферзя нет;

b_k указывает, что на k-й /-диагонали ферзя нет;

c_k указывает, что на k-й \-диагонали ферзя нет.

Выбор границ индексов b1, b2, c1, c2 определяется, исходя из способа вычисления индексов для b и c, на /-диагонали у всех полей постоянна сумма координат i и j, а на \-диагонали постоянна их разность. Соответствующие вычисления приведены в прогр. 3.4. Если мы уже определили так данные, то оператор «Поставить ферзя» превращается в такие операторы:

```
x[i] := j, a[j] := FALSE, b[i + j] := FALSE;
c[i - j] := FALSE
```
(3.32)

а оператор «Убрать ферзя» в такие:

```
a[j] := TRUE; b[i + j] := TRUE; c[i - j] := TRUE
```
(3.33)

Условие «безопасно» выполняется, если поле с координатами $\langle i, j \rangle$ лежит на горизонтали и вертикали, которые еще не заняты. Следовательно, ему соответствует логическое выражение

```
a[j] & b[i + j] & c[i - j]
```
(3.34)

```
MODULE Queens;
  FROM InOut IMPORT WriteInt, WriteLn;
VAR i: INTEGER; q: BOOLEAN;
  a: ARRAY [1 .. 8] OF BOOLEAN;
  b: ARRAY [2 .. 16] OF BOOLEAN;
  c: ARRAY [-7 .. 7] OF BOOLEAN;
  x: ARRAY [1 .. 8] OF INTEGER;

PROCEDURE Try(i: INTEGER; VAR q: BOOLEAN);
  VAR j: INTEGER;
BEGIN j := 0;
  REPEAT j := j+1; q := FALSE;
    IF a[j] & b[i+j] & c[i-j] THEN
      x[i] := j;
      a[j] := FALSE; b[i+j] := FALSE; c[i-j] := FALSE;
      IF i < 8 THEN
        Try(i+1, q);
        IF ~q THEN
          a[j] := TRUE; b[i+j] := TRUE; c[i-j] := TRUE
        END
      ELSE q := TRUE
      END
    END
  UNTIL q OR (j = 8)
END Try;

BEGIN
  FOR i := 1 TO 8 DO a[i] := TRUE END;
  FOR i := 2 TO 16 DO b[i] := TRUE END;
  FOR i := -7 TO 7 DO c[i] := TRUE END;
  Try(1,q);
  FOR i := 1 TO 8 DO WriteInt(x[i], 4) END;
  WriteLn
END Queens.
```

Прогр. 3.4. Расстановка восьми ферзей.

На этом создание алгоритма заканчивается; полностью он представлен в прогр. 3.4. На рис. 3.9 приведено полученное решение $x = (1, 5, 8, 6, 3, 7, 2, 4)$.

Прежде чем закончить разбор задач, «посвященных» шахматной доске, мы воспользуемся задачей о восьми ферзях и представим одно важное обобщение алгоритма проб и ошибок. В общих словах, речь идет о нахождении не одного, а *всех* решений поставленной задачи.

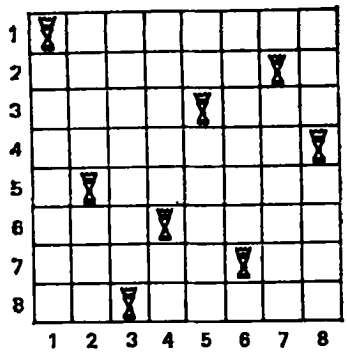


Рис. 3.9. Одно из решений задачи о восьми ферзях.

Такое обобщение получается довольно легко. Напомним, что формирование возможных кандидатов происходит регулярным образом, гарантирующим, что ни один кандидат не встретится более чем один раз. Такое свойство алгоритма обеспечивается тем, что поиск идет по дереву кандидатов так, что каждая из его вершин проходится точно один раз. Это позволяет, если найдено и должным образом зафиксировано одно решение, просто переходить к следующему кандидату, предлагаемому упомянутым процессом систематического перебора. Общую схему такого процесса (3.35) можно «вывести» из схемы (3.29).

```
PROCEDURE Try(i: INTEGER);
  VAR k: INTEGER;
BEGIN                                                          (3.35)
  FOR k := 1 TO m DO
    выбор k-го кандидата;
    IF подходит THEN его запись;
      IF i < n THEN Try(i+1) ELSE печатать решения END;
      стирание записи
    END
  END
END Try
```

Обратите внимание: из-за того, что условие окончания в процессе выбора свелось к одному отношению k = m, оператор повторения со словом REPEAT заменится на оператор цикла с FOR. Удивительно, что

```
MODULE AllQueens;
  FROM InOut IMPORT WriteInt, WriteLn;

VAR i: INTEGER;
   a: ARRAY [ 1 .. 8] OF BOOLEAN;
   b: ARRAY [ 2 .. 16] OF BOOLEAN;
   c: ARRAY [-7 .. 7] OF BOOLEAN;
   x: ARRAY [ 1 .. 8] OF INTEGER;

 PROCEDURE print;
   VAR k: INTEGER;
 BEGIN
   FOR k := 1 TO 8 DO WriteInt(x[k], 4) END ;
   WriteLn
 END print;

 PROCEDURE Try(i: INTEGER);
   VAR j: INTEGER;
 BEGIN
   FOR j := 1 TO 8 DO
    IF a[j] & b[i+j] & c[i-j] THEN
      x[i] := j;
      a[j] := FALSE; b[i+j] := FALSE; c[i-j] := FALSE;
      IF i < 8 THEN Try(i+1) ELSE print END ;
      a[j] := TRUE; b[i+j] := TRUE; c[i-j] := TRUE
    END
   END
 END Try;

BEGIN
  FOR i := 1 TO 8 DO a[i] := TRUE END ;
  FOR i := 2 TO 16 DO b[i] := TRUE END ;
  FOR i := -7 TO 7 DO c[i] := TRUE END ;
  Try(1)
END AllQueens.
```

Прогр. 3.5. Расстановка восьми ферзей (все решения).

поиск всех возможных решений выполняется более простой программой, чем в случае поиска одного-единственного решения *).

---

*) Если бы программа отыскивала лишь 12 принципиально различных решений, причем делала бы это максимально быстро, то и она сама, и данные, которыми она должна была бы манипулировать, оказались бы значительно более сложными. — *Прим. перев.*

Таблица 3.2. Двенадцать решений задачи восьми ферзей

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	n
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	072
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

Обобщенный алгоритм, приведенный в прогр. 3.5, отыскивает 82 решения задачи о восьми ферзях. Однако принципиально различных решений всего 12 — наша программа не учитывает симметрию. В табл. 3.2 приведены первые 12 решений. Приведенное справа число n указывает, сколько раз проводилась проверка на «безопасность» поля. Среднее число для всех 92 решений равно 161.

3.6. ЗАДАЧА О СТАБИЛЬНЫХ БРАКАХ

Предположим, есть два непересекающихся множества A и B одинакового размера n. Нужно найти множество из n пар $\langle a, b\rangle$, таких, что a принадлежит A, а b принадлежит B, и они удовлетворяют некоторым условиям. Для выбора таких пар существует много различных критериев; один из них называется «*правилом стабильных браков*».

Предположим, что A — множество мужчин, а B — женщин. У каждых мужчины и женщины есть различные правила предпочтения возможного партнера. Если среди n выбранных пар существуют мужчины и женщины, не состоящие между собой в браке, но предпочитающие друг друга, а не своих фактических супругов, то такое множество браков считается нестабильным. Если же таких пар нет, то множество считается стабильным. Такая ситуация характерна для многих похожих задач, где нужно проводить распределение в соответствии с некоторыми правилами

предпочтения. Так, например, идет выбор студентами — учебных заведений, призывниками — родов войск и т. п. Пример с браками выбран нами почти интуитивно, причем заметим, что правила предпочтения постоянны и в процессе выбора не изменяются. Это упрощает задачу, но сильно искажает действительность (как всякая абстракция).

Один из путей поиска решения: пробовать объединять в пары элементы двух множеств до тех пор, пока они не будут исчерпаны. Намереваясь найти все устойчивые распределения, мы можем легко набросать некоторое решение, используя в качестве образца схему программы (3.35). Пусть *Try*(*m*) — алгоритм поиска супруги для мужчины m, причем этот поиск идет в порядке списка предпочтений именно этого мужчины. Первый вариант программы, основанный на этих предположениях, таков (3.36):

```
PROCEDURE Try(m: man);
  VAR r: rank;
BEGIN                                                    (3.36)
  FOR r := 1 TO n DO
    выбор r-й претендентки для m;
    IF подходит THEN запись брака;
      IF m – не последний THEN Try(successor(m))
      ELSE записать стабильное множество
      END;
      отменить брак
    END
  END
END Try
```

Как и раньше, мы не может сдвинуться с этой точки, пока не решим, как представлять данные. Введем три скалярных типа и для простоты будем считать, что они принимают целые значения от 1 до n. Хотя формально эти три типа идентичны, однако различные их имена значительно «проясняют» суть дела. В частности, сразу понятно, что означает та или другая переменная.

```
TYPE man = [1 .. n];
     woman = [1 .. n];                                   (3.37)
     rank = [1 .. n]
```

Таблица 3.3. Пример исходных данных для wmr и mwr

r =	1	2	3	4	5	6	7	8		w =	1	2	3	4	5	6	7	8
m = 1	7	2	6	5	1	3	8	4		1	4	6	2	5	8	1	3	7
2	4	3	2	6	8	1	7	5		2	8	5	3	1	6	7	4	2
3	3	2	4	1	8	5	7	6		3	6	8	1	2	3	4	7	5
4	3	8	4	2	5	6	7	1		4	3	2	4	7	6	8	5	1
5	8	3	4	5	6	1	7	2		5	6	3	1	4	5	7	2	8
6	8	7	5	2	4	3	1	6		6	2	1	3	8	7	4	6	5
7	2	4	6	3	1	7	5	8		7	3	5	7	2	4	1	8	6
8	6	1	4	2	7	5	3	8		8	7	2	8	4	5	6	3	1

Исходные данные представляют собой две матрицы, задающие предпочтительных партнеров для мужчин и женщин.

```
VAR wmr: ARRAY man, rank OF woman
    mwr: ARRAY woman, rank OF man
```
(3.38)

Соответственно wmr_m обозначает список предпочитаемых для мужчин m, т. е. $wmr_{m,r}$ — женщина, стоящая на r-м месте в списке для мужчины m. Аналогично, mwr_w — список предпочитаемых партнеров для женщины w, а $mwr_{w,r}$ — ее кандидат. В табл. 3.3 приведен пример таких исходных данных.

Результат — массив женщин x, причем x_m соответствует партнерше для мужчины m. Для поддержания симметрии между мужчинами и женщинами вводится дополнительный список y, и y_w задает партнера для женщины w.

```
VAR x: ARRAY man OF woman
    y: ARRAY woman OF man
```
(3.39)

Фактически массив y излишен, поскольку хранящаяся в нем информация уже существует в x. Ясно, что для находящихся в браке мужчины и женщины справедливы равенства

$$x_{y_w} = w, \quad y_{x_m} = m \tag{3.40}$$

Таким образом, значение y_w можно определить с помощью простого просмотра x. Однако нет сомнения, что наличие y повышает эффективность алгоритма. Ведь информация, заключающаяся в массивах x и y, нужна для определения стабильности предлагаемого множества браков. Поскольку это множество строится

шаг за шагом и после каждого предлагаемого брака проверяется стабильность, то x и y используются еще до того, как будут определены все их компоненты. Для того чтобы знать, какие компоненты уже определены, можно ввести булевские массивы:

```
singlem: ARRAY man OF BOOLEAN
singlew: ARRAY woman OF BOOLEAN
```
(3.41)

Истинность значения $singlem_m$ означает, что x_m уже определено, а $singlew_w$ — что определено y_w. Просматривая предложенный алгоритм, мы, однако, легко выясняем, что семейное положение мужчины m определяется равенством

$$\sim \text{singlem}[k] = k < m \qquad (3.42)$$

Следовательно, массив *singlem* можно убрать, соответственно и имя *singlew* сокращается до *single*. Все эти соглашения приводят нас к уточненному алгоритму (3.43). Предикат *допустимо* можно представить в виде конъюнкции *single* и *stable*, где *stable* — функция, которую нужно еще определить.

```
PROCEDURE Try(m: man);
  VAR r: rank; w: woman;
BEGIN                                                                    (3.43)
  FOR r := 1 TO n DO
    w := wmr[m,r];
    IF single[w] & stable THEN
      x[m] := w; y[w] := m; single[w] := FALSE;
      IF m < n THEN Try(successor(m)) ELSE record set END;
      single[w] := TRUE
    END
  END
END Try
```

В этот момент все еще продолжает бросаться в глаза большое сходство с прогр. 3.5. Теперь основная задача — уточнить алгоритм определения стабильности. К сожалению, стабильность невозможно представить с помощью такого же простого выражения, каким характеризовалась безопасность поля для ферзя в прогр. 3.5. Прежде всего следует иметь в виду, что по определению стабильность связана со сравнением предпочтительности партнеров (рангов). Однако ран-

ги мужчин и женщин нигде в наших данных явно не фигурируют. Конечно, ранг женщины w с точки зрения мужчины m можно определить, но для этого нужен «трудоемкий» поиск значения w в массиве wmr_m. Поскольку вычисления стабильности — очень часто встречающаяся операция, то хотелось, чтобы эта информация была более доступна. Для этого воспользуемся двумя матрицами:

```
rmw: ARRAY man, woman OF rank;
rwm: ARRAY woman, man OF rank
```
(3.44)

причем $rmw_{m,w}$ означает ранг женщины w в списке предпочтений мужчины m, а $rwm_{w,m}$ — ранг мужчины m в списке женщины w. Ясно, что значения этих вспомогательных массивов суть константы, и их можно определить в самом начале по значениям wmr и mwr.

Процесс вычисления предиката stable теперь можно вести в строгом соответствии с его первоначальным определением. Вспомним, что мы пытаемся определить возможность брака между m и w, где $w = wmr_{m,r}$, т. е. w стоит в списке m на r-м месте. Будучи оптимистами, мы вначале будем предполагать, что стабильность сохраняется, а затем уже попытаемся найти возможные источники неприятностей. Где они могут таиться? Существуют две симметричные возможности:

1. Может существовать женщина pw, которая для m предпочтительнее w, причем для pw m предпочтительнее ее супруга.

2. Может существовать мужчина pm, который для w предпочтительнее m, причем для pm w предпочтительнее его супруги.

Исследуя первый источник неприятностей, мы сравниваем ранги $rwm_{pw,m}$ и $rwm_{pw,y_{pw}}$ для всех женщин, которых m предпочитает w, т. е. для женщин, удовлетворяющих условию $pw = wmr_{m,i}$ при $i < r$. Мы знаем, что все эти женщины уже были выданы замуж, так как если бы одна из них была еще одинокой, то m выбрал бы ее. Описанный процесс можно сформулировать как простой линейный поиск; s означает

стабильность.

```
s := TRUE; i := 1;
WHILE (i < r) & s DO
  pw := wmr[m,i]; i := i+1;
  IF ~single[pw] THEN s := rwm[pw,m] > rwm[pw, y[pw]] END
END
```
(3.45)

В поисках неприятностей второго вида мы должны проверить всех кандидатов pm, которые для w предпочтительнее «суженому», т. е. всех предпочитаемых мужчин $pm = mwr_{w,i}$ для $i < rwm_{w,m}$. Как и при исследовании неприятностей первого вида, нужно сравнивать ранги $rmw_{pm,w}$ с $rmw_{pm,x_{pm}}$. Однако здесь надо быть внимательными и не проводить сравнение с x_{pm}, если pm еще не женат. Для этого можно воспользоваться проверкой $pm < m$, так как известно, что все мужчины, предшествующие m, уже женаты.

Полный алгоритм представлен в прогр. 3.6, а в табл. 3.4 приведены девять стабильных решений, полученных на основе исходных wmr и mwr из табл. 3.3.

Наш алгоритм построен по простой схеме поиска с возвратом. Его эффективность зависит главным образом от того, на сколько удачно выбрана схема усечения дерева решений. Маквити и Уилсон [3.1, 3.2] предложили несколько более быстрый, но зато более сложный и менее «прозрачный» алгоритм; к тому же они распространили его на случай множеств различного размера.

Алгоритмы такого типа, как использованные нами в последних двух примерах, формирующие все возможные решения некоторой задачи (при определенных ограничениях), часто употребляются для выбора одного или нескольких в каком-то смысле оптимальных решений. В приведенном примере такими могут быть, скажем, решения, в среднем больше удовлетворяющие мужчин или женщин или и тех и других.

Обратите внимание, что в табл. 3.4 приведены суммы рангов всех женщин из списков предпочтения их мужей и суммы рангов всех мужей из списков предпочтения их жен. Эти величины определяются так:

$$rm = Sm : 1 \leqslant m \leqslant n : rmw_{m,x_m}$$
$$rw = Sm : 1 \leqslant m \leqslant n : rwm_{x_m,m} \qquad (3.46)$$

```
MODULE Marriage;
FROM InOut IMPORT
  ReadCard, WriteCard, WriteLn;

CONST n = 8;
TYPE man = [1 .. n];
  woman = [1 .. n];
  rank = [1 .. n];

VAR m: man; w: woman; r: rank;
  wmr: ARRAY man, rank OF woman;
  mwr: ARRAY woman, rank OF man;
  rmw: ARRAY man, woman OF rank;
  rwm: ARRAY woman, man OF rank;
  x: ARRAY man OF woman;
  y: ARRAY woman OF man;
  single: ARRAY woman OF BOOLEAN;
  h: CARDINAL;

PROCEDURE print;
  VAR m: man; rm, rw: CARDINAL;
BEGIN rm := 0; rw := 0;
  FOR m := 1 TO n DO
    WriteCard(x[m], 4);
    rm := rmw[m, x[m]] + rm; rw := rwm[x[m], m] + rw
  END;
  WriteCard(rm, 8); WriteCard(rw, 4); WriteLn
END print;

PROCEDURE stable(m: man; w: woman; r: rank): BOOLEAN;
  VAR pm: man; pw: woman;
    i, lim: rank; S: BOOLEAN;
BEGIN S := TRUE; i := 1;
  WHILE (i < r) & S DO
    pw := wmr[m,i]; i := i+1;
    IF ~single[pw] THEN S := rwm[pw,m] > rwm[pw, y[pw]] END
  END;
  i := 1; lim := rwm[w,m];
  WHILE (i < lim) & S DO
    pm := mwr[w,i]; i := i+1;
    IF pm < m THEN S := rmw[pm,w] > rmw[pm, x[pm]] END
  END;
```

```
  RETURN S
 END stable;

 PROCEDURE Try(m: man);
  VAR w: woman; r: rank;
 BEGIN
  FOR r := 1 TO n DO w := wmr[m,r];
   IF single[w] & stable(m,w,r) THEN
    x[m] := w; y[w] := m; single[w] := FALSE;
    IF m < n THEN Try(m+1) ELSE print END;
    single[w] := TRUE
   END
  END
 END Try;

BEGIN
 FOR m := 1 TO n DO
  FOR r := 1 TO n DO
   ReadCard(h); wmr[m,r] := h; rmw[m, wmr[m,r]] := r
  END
 END;
 FOR w := 1 TO n DO
  single[w] := TRUE;
  FOR r := 1 TO n DO
   ReadCard(h); mwr[w,r] := h; rwm[w, mwr[w,r]] := r
  END
 END;
 Try(1)
END Marriage.
```

Прогр. 3.6. Выбор стабильных браков.

Таблица 3.4. Решение задачи о стабильных браках

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	rm	rw	c
1	7	4	3	8	1	5	2	6	16	32	21
2	2	4	3	8	1	5	7	6	22	27	449
3	2	4	3	1	7	5	8	6	31	20	59
4	6	4	3	8	1	5	7	2	26	22	62
5	6	4	3	1	7	5	8	2	35	15	47
6	6	3	4	8	1	5	7	2	29	20	143
7	6	3	4	1	7	5	8	2	38	13	47
8	3	6	4	8	1	5	7	2	34	18	758
9	3	6	4	1	7	5	8	2	43	11	34

c — число проверок устойчивости.
Решение 1 — оптимальное для мужчин,
решение 9 — оптимальное для женщин.

Решение с минимальным значением rm называется *стабильным и оптимальным для мужчин*, а с самым маленьким значением rw — *стабильным и оптимальным для женщин*. Природа нашей стратегии выбора такова, что первыми формируются решения, больше удовлетворяющие мужчин, а решения, хорошие с точки зрения женщин, появляются только в конце. В этом смысле алгоритм ориентирован на сильный пол. Его можно быстро переориентировать, если систематически менять ролями мужчин и женщин, т. е. просто заменить mwr на wmr и наоборот.

Мы не будем уже усложнять дальше нашу программу и оставим поиск оптимального решения для следующего, теперь, наконец, последнего примера алгоритма с возвратом.

3.7. ЗАДАЧА ОПТИМАЛЬНОГО ВЫБОРА

Наш последний пример алгоритма с возвратом — логическое обобщение последних двух примеров, построенных по общей схеме (3.35). Вначале мы пользовались принципом возврата для поиска единственного решения поставленной задачи. Примерами были обход доски ходом коня и размещение восьми ферзей. Затем мы поставили себе целью поиск всех решений некоторой задачи и разобрали задачу восьми ферзей и стабильных браков. Теперь мы хотим заняться поиском *оптимального* решения. Для этого необходимо формировать все возможные решения и в процессе их получения оставлять одно, в некотором определенном смысле оптимальное. Предположим, оптимальность оценивается с помощью некоторой положительной функции f(s). В этом случае, заменяя в схеме (3.35) оператор «печать решения» на оператор

```
IF f(решение) > f(оптимум) THEN оптимум := решение END
```

(3.47)

мы получаем новый алгоритм. В переменной *optimum* хранится лучшее из полученных к этому времени решений. Разумеется, переменную нужно соответствующим образом инициировать. Значение $f(optimum)$

удобнее хранить еще в одной переменной, чтобы избежать частого его перевычисления.

В качестве примера общей проблемы поиска оптимального решения мы выбираем такую важную и часто встречающуюся задачу: нужно найти, руководствуясь некоторыми ограничениями, оптимальную выборку из заданного множества объектов. Выборки, представляющие приемлемые решения, строятся постепенно: один за одним просматриваются отдельные объекты из исходного множества. Процедура *Try* описывает процесс исследования пригодности одного отдельного объекта — она вызывается рекурсивно (для исследования очередного объекта) — до тех пор, пока все они не будут рассмотрены.

Мы видим, что при рассмотрении каждого объекта (в предыдущих примерах мы называли их кандидатами) возможны два заключения, а именно: либо включать исследуемый объект в текущую выборку, либо не включать. В такой ситуации оператор цикла уже не годится, вместо этого нужно явно описать оба варианта. Этот процесс описан в (3.48), причем здесь объекты просто нумеруются 1, 2, ..., n.

```
PROCEDURE Try(i: INTEGER);
BEGIN
  IF включение приемлемо THEN включение i-го объекта;
    IF i < n THEN Try(i+1) ELSE проверка оптимальности END;
    исключение i-го объекта                                    (3.48)
  END;
  IF приемлемо невключение THEN
    IF i < n THEN Try(i+1) ELSE проверка оптимальности END
  END
END Try
```

Из схемы видно, что существует всего 2^n возможных выборок, поэтому нужно использовать такой критерий приемлемости, который значительно бы сократил число перебираемых кандидатов. Для пояснения этого процесса возьмем конкретный пример построения выборки. Пусть каждый из n объектов $a_1, \ldots, a_n$ характеризуется весом и ценностью. Оптимальной назовем выборку с максимальной суммой ценностей и суммой весов, ограниченной некоторым пределом.

С такой проблемой сталкивается любой путешественник, упаковывающий чемоданы и стремящийся выбрать так n предметов, чтобы ценность их была оптимальной, а общий вес не превышал какого-то допустимого предела.

Теперь нам надо решить, а как же все это представить в виде определенных данных. Из вышесказанного легко следуют такие описания (3.49):

```
TYPE index = [1 .. n];
  object = RECORD weight, value: INTEGER END;

VAR obj: ARRAY index OF object;                    (3.49)
  limw, totv, maxv: INTEGER;
  s, opts: SET OF index
```

Переменные *limw* и *totv* соответствуют предельному весу и общей ценности всех n объектов. В процессе построения выборки эти две величины фактически остаются постоянными. Через s обозначена текущая выборка, здесь каждый из объектов представлен своим именем (индексом); *opts* — оптимальная выборка, полученная к данному моменту, а *maxv* — ее ценность.

Каковы же критерии приемлемости объекта для текущей выборки? Если речь идет о включении, то объект можно включать в выборку, если он подходит по весовым ограничениям. Если он не подходит, то попытки добавить еще один объект в текущую выборку можно прекратить. Если, однако, речь идет об исключении, то критерием приемлемости, т. е. возможности продолжения построения текущей выборки, будет то, что после данного исключения общая ценность будет не меньше полученного до этого момента оптимума. Ведь если сумма меньше, то продолжение поиска, хотя он и может дать некоторое решение, не приведет к оптимальному решению. Следовательно, дальнейший поиск на текущем пути бесполезен. Из этих двух условий мы определяем величины, которые нужно вычислять на каждом шаге процесса отбора:

1. Общий вес *tw* выборки, полученной к данному моменту.

2. Общая ценность *av* текущей выборки, которую можно еще достичь.

Эти две величины удобно представлять как параметры процедуры *Try*. Условие «включение приемлемо» в (3.48) можно теперь сформулировать так:

$$tw + a[i].weight \leqslant lim\ w \tag{3.50}$$

а последующая проверка на оптимальность выглядит следующим образом:

```
IF av > maxv THEN (* новый оптимум, его запись *)
  opts := s; maxv := av                                    (3.51)
END
```

Последнее присваивание основано на том соображении, что достижимое значение будет достигаться только после обработки всех объектов. Условие «исключение приемлемо» в (3.48) выражается так:

```
ReadCard(obj[i].weight); ReadCard(obj[i].value);
totv := totv + obj[i].value
END;
ReadCard(WeightInc); ReadCard(WeightLimit);
```

Поскольку полученное значение av — a[i].value вновь используется, оно, чтобы избежать перевычисления, «сохраняется» в переменной *av1*.

Из схемы (3.48) и фрагментов с (3.49) до (3.52) получаем, добавив соответствующие операторы инициации глобальных переменных, всю программу це-

Таблица 3.5. Результаты работы программы оптимального выбора

Вес	10	11	12	13	14	15	16	17	18	19	
Ценность	18	20	17	19	25	21	27	23	25	24	
10	*										18
20							*				27
30					*		*				52
40	*				*		*				70
50	*	*		*			*				84
60	*	*	*	*	*						99
70	*	*			*		*		*		115
80	*	*	*		*		*	*			130
90	*	*			*		*		*	*	139
100	*	*		*	*		*	*	*		157
110	*	*	*	*	*	*	*		*		172
120	*	*			*	*	*	*	*	*	183

```
MODULE Selection;
 (* поиск оптимального выбора объекта при ограничении *)
 FROM InOut IMPORT
  ReadCard, Write, WriteCard, WriteString, WriteLn;

 CONST n = 10;
 TYPE index = [1 .. n];
    object = RECORD value, weight: CARDINAL END ;
    ObjSet = SET OF index;

 VAR i: index;
  obj: ARRAY index OF object;
  limw, totv, maxv: CARDINAL;
  s, opts: ObjSet;
  WeightInc, WeightLimit: CARDINAL;
  tick: ARRAY [FALSE .. TRUE] OF CHAR;

 PROCEDURE Try(i: index; tw, av: CARDINAL);
  VAR av1: CARDINAL;
 BEGIN (* попытка включения *)
  IF tw + obj[i].weight <= limw THEN
   s := s + ObjSet{i};
   IF i < n THEN Try(i+1, tw + obj[i].weight, av)
   ELSIF av > maxv THEN maxv := av; opts := s
   END ;
   s := s - ObjSet{i}
  END ;
  (* попытка исключения *)
  IF av > maxv + obj[i].value THEN
   IF i < n THEN Try(i+1, tw, av - obj[i].value)
   ELSE maxv := av - obj[i].value; opts := s
   END
  END
 END Try;

BEGIN totv := 0; limw := 0;
 tick[FALSE] := " "; tick[TRUE] := "*";
 FOR i := 1 TO n DO
```

```
WriteString("Weight");
FOR i := 1 TO n DO WriteCard(obj[i].weight, 5) END;
WriteLn; WriteString("Value ");
FOR i := 1 TO n DO WriteCard(obj[i].value, 5) END;
WriteLn;
REPEAT limw := limw + WeightInc; maxv := 0;
  s := ObjSet{}; opts := ObjSet{}; Try(1, 0, totv);
  WriteCard(limw, 6);
  FOR i := 1 TO n DO
    WriteString("  "); Write(tick[i IN opts])
  END;
  WriteCard(maxv, 8); WriteLn
UNTIL limw >= WeightLimit
END Selection.
```

Прогр. 3.7. Оптимальный выбор.

ликом. Следует заметить, что для включения и исключения из множества s здесь очень удобно пользоваться операциями над множествами. В табл. 3.5 приводятся результаты работы прогр. 3.7 с допустимыми весами от 10 до 120.

Такая схема с возвратами, использующая некоторые ограничения для уменьшения роста потенциального дерева поиска, называется алгоритмом ветвей и границ.

УПРАЖНЕНИЯ

3.1. Ханойские башни. Даны три стержня и n дисков различного размера. Диски можно надевать на стержни, образуя из них башни. Пусть вначале на стержне A в убывающем порядке расположено, скажем, три диска (рис. 3.10). Задача заключается в том, чтобы перенести n дисков со стержня A на

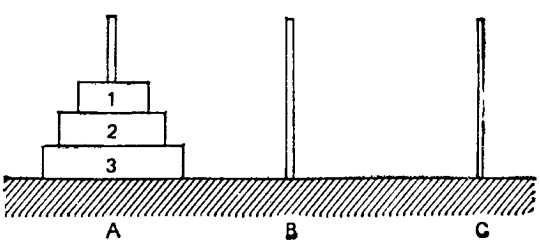


Рис. 3.10. Ханойские башни.

стержень C, сохранив их первоначальный порядок. При переносе необходимо следовать таким правилам:

1. На каждом шаге со стержня на стержень переносится точно один диск.

2. Диск нельзя помещать на диск меньшего размера.

3. Для промежуточного хранения можно использовать стержень B. Найдите алгоритм решения этой задачи. Башню удобнее рассматривать как состоящую из одного верхнего диска и башни, образованной оставшимися дисками. Опишите алгоритм как рекурсивную программу.

3.2. Напишите процедуру формирования на том же месте всех n! перестановок для n элементов $a_1, \ldots, a_n$, т. е. без дополнительного массива. После формирования очередной перестановки можно, например, обратиться к процедуре Q (с параметром), которая напечатает полученную перестановку.

Указание: Задачу формирования всех перестановок элементов $a_1, \ldots, a_m$ можно считать состоящей из m подзадач формирования всех перестановок для элементов $a_1, \ldots, a_{m-1}$, за которыми следует a_m. Причем в i-й подзадаче вначале меняются местами элементы a_i и a_m.

3.3. Определите рекурсивную схему для построения кривой на рис. 3.11. Это результат наложения четырех кривых W_1, W_2, W_3, W_4. Она похожа на кривую Серпинского (3.21) и (3.22). Разобравшись в рекурсивном ее строении, напишите рекурсивную программу для черчения такой кривой.

Рис. 3.11. W-кривые порядков 1—4.

3.4. Из 92 решений, получающихся с помощью программы 3.5 для задачи о восьми ферзях, только 12 по существу различны. Все другие решения можно получить с помощью осевой или центральной симметрии. Напишите программу для получения именно этих основных 12 решений. Обратите внимание, что поиск на первой вертикали можно ограничить позициями 1—4.

3.5. Измените так программу устойчивых браков, чтобы она находила оптимальное решение (для мужчин или женщин). Это означает, что она станет похожей на программу 3.7, работающую по методу «ветвей и границ».

3.6. Некоторая железнодорожная компания обслуживает n станций $S_1, \ldots, S_n$. Она предполагает улучшить информационное обслуживание клиентов, установив терминалы, управляемые вычислительной машиной. С терминала клиент вводит название своей станции отправления S_A и станции назначения — S_D, а ему должно выдаваться расписание поездов, обеспечивающих минимальное время поездки. Напишите программу для получения нужной информации. Расписание (это ваш банк данных) представлено данными некоторой структуры, где перечисляется время отправления (прибытия) всех поездов. Разумеется, не между всеми станциями есть прямое сообщение.

3.7. Функция Аккермана A определяется для всех неотрицательных целых аргументов m и n таким образом:

$$A(0, n) = n + 1$$
$$A(m, 0) = A(m - 1, 1) \qquad (m > 0)$$
$$A(m, n) = A(m - 1, A(m, n - 1)) \qquad (m, n > 0)$$

Напишите программу, вычисляющую A(m, n), рекурсию не используйте!

За образец возьмите программу 2.11 — нерекурсивную версию быстрой сортировки. Определите правила общего преобразования рекурсивных программ в итеративные.

ЛИТЕРАТУРА

[3.1] McVitie D. G., Wilson L. B. The Stable Marriage Problem. *Comm. ACM*, **14**, No. 7, (1971), 486—492.

[3.2] Mc Vitie D. G. 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.

4. ДАННЫЕ С ДИНАМИЧЕСКОЙ СТРУКТУРОЙ

4.1. ТИПЫ РЕКУРСИВНЫХ ДАННЫХ

Во второй главе мы ввели как основные следующие структуры для данных: массивы, записи и множества. Они названы основными, поскольку из них можно образовывать более сложные структуры и так как они чаще всего встречаются на практике. Цель описания типа данных и последующего определения некоторых переменных как относящихся к этому типу состоит в том, чтобы раз и навсегда зафиксировать диапазон значений, присваиваемых этим переменным, и соответственно размер выделяемой для них памяти. Поэтому о таких переменных говорят как о *статических* переменных. Существует, однако, много задач, которые требуют данных с более сложной структурой. Для них характерно, что в процессе вычисления изменяются не только значения переменных, но даже их структура. Поэтому такие переменные стали называться данными с *динамической структурой*. Естественно, что компоненты таких объектов на некотором уровне детализации представляют собой статические объекты, т. е. они принадлежат к одному из основных типов данных. Эта глава посвящена конструированию данных с динамической структурой, их анализу и работе с ними.

Примечательно, что существуют некоторые близкие аналогии между методами, употребляемыми при построении алгоритмов, и соответствующими методами для построения данных. Как и при любых аналогиях, между ними есть какие-то различия, однако сравнение методов построения программ и данных тем не менее проливает свет на очень многие вопросы.

Элементарным, не распадающимся на составные части оператором является оператор *присваивания* значения выражения некоторой переменной. Среди данных ему соответствуют данные скалярных типов,

т. е. данные, не имеющие структуры. И те и другие являются атомарными строительными блоками для образования составных операторов и составных типов данных. Самые простые образования, получающиеся с помощью перечисления или следования, — это составные операторы и записи. Оба состоят из конечного (обычно небольшого) количества явно перечисленных компонент, которые сами могут отличаться одна от другой. Если же компоненты идентичные, то выписывать их по отдельности нет необходимости: для того чтобы описывать повторения, число которых известно и конечно, мы пользуемся оператором цикла с параметром (FOR) и массивом. Выбор одного из двух или более вариантов выражается с помощью условного оператора или оператора варианта, и им соответствует запись с вариантами. И наконец, повторение заранее неизвестное число раз (потенциально даже бесконечное) выражается операторами повторения с предусловием (WHILE) или постусловием (REPEAT). Им соответствует такая структура данных, как последовательность (файл) — простейшая структура, допускающая построение типов бесконечной мощности.

Возникает вопрос, а существует ли структура данных, таким же образом соответствующая оператору процедуры. Естественно, наиболее интересное и новое в этом отношении свойство процедур — *рекурсивность*. По аналогии с процедурой, содержащей один или более обращений к самой себе, величины, относящиеся к такому рекурсивному типу, должны были бы состоять из одной или более компонент, относящихся к тому же типу, что и сама величина. Подобно процедурам, определения таких типов данных могли бы быть прямо или косвенно рекурсивными.

Простой пример объекта, которому больше всего подходило бы рекурсивное определение типа, — арифметическое выражение, встречающееся в языках программирования. Рекурсия в этом случае отражает возможность вложенности, т. е. использования в выражениях в качестве операндов заключенных в скобки подвыражений. Определим неформально выражения следующим образом:

Выражение состоит из *терма*, за которым следует знак операции, за которым опять идет *терм*. (Термы представляют собой операнды указанной операции.) Сам *терм* — это либо переменная, обозначенная каким-либо идентификатором, либо заключенное в скобки выражение.

Тип данных, значениями которых являются такие выражения, можно легко описать, если к уже доступным возможностям добавить рекурсию *).

```
TYPE expression = RECORD op: operator;
                    opd1, opd2: term
                  END

TYPE term =       RECORD                                    (4.1)
                    CASE t: BOOLEAN OF
                      TRUE: id: alfa |
                      FALSE: subex: expression
                    END
                  END
```

Замечание: Используя Модулу-2, мы должны употреблять не условные конструкции, которые язык запрещает здесь применять, а варианты.

Следовательно, каждая переменная типа *term* состоит из двух компонент: поля признака t и, если t — истина, поля *id*, иначе — поля *subex*. Рассмотрим в качестве примеров такие четыре выражения:

1. $x + y$
2. $x - (y * z)$ (4.2)
3. $(x + y) * (z - w)$
4. $(x/(y + z)) * w$

Эти выражения можно «нарисовать» так, как на рис. 4.1. В этом случае явно видны вложенность, рекурсивность их структуры и способ их размещения в памяти.

Второй пример рекурсивной структуры информации — генеалогическое дерево. Дерево определяется как имя человека плюс два дерева его родителей. Такое определение безусловно ведет к бесконечной структуре. Однако реальные генеалогические деревья,

*) Такое определение больше соответствует выражениям в префиксном виде, в то время как выше был определен инфиксный вид выражений. — *Прим. перев.*

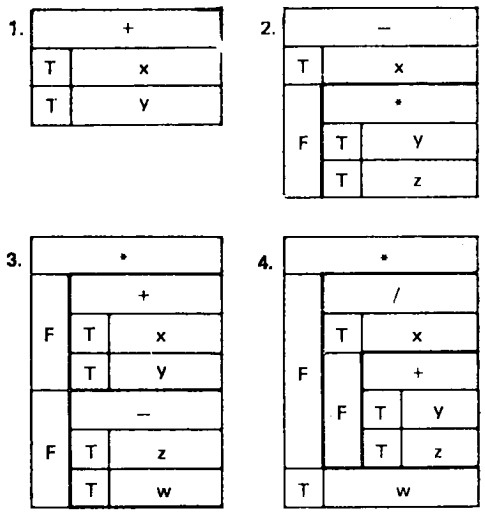


Рис. 4.1. Образ расположения в памяти записей с рекурсивной структурой.

естественно, конечны, ведь на каком-то уровне сведения о предках отсутствуют. Мы можем учесть это соображение, если вновь воспользуемся структурой с вариантами (4.3).

```
TYPE ped =  RECORD
              CASE known: BOOLEAN OF
                TRUE: name: alfa; father, mother: ped |        (4.3)
                FALSE: (*пусто*)
              END
            END
```

Обратите внимание, что каждая переменная типа *ped* содержит по крайней мере одну компоненту — поле признака под именем *known* (известен). Если это значение «истина», то есть еще три поля, в противном случае полей больше нет. Ниже приводится пример такого значения, записанного в виде вложенного выражения, а на схеме (рис. 4.2) приведено его возможное представление в памяти.

(T,Ted,(T,Fred,(T,Adam,(F),(F)),(F)),(T,Mary,(F),
(T,Fva,(F),(F))))

Здесь становится очевидной важная роль вариантов; это единственное средство, позволяющее ограничить рекурсивную структуру. Поэтому в рекурсивных определениях они всегда присутствуют. Именно здесь особенно наглядно прослеживается аналогия между структурами программ и данных. В каждую рекурсивную процедуру, чтобы ее выполнение когда-нибудь закончилось, обязательно должен включаться условный оператор (или оператор варианта). Ясно, что «конечность выполнения» соответствует конечности мощности.

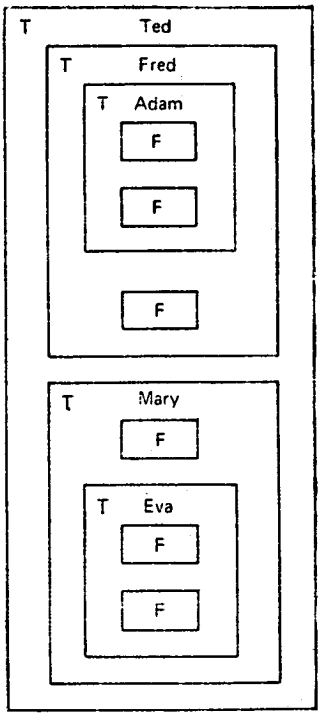


Рис. 4.2. Пример рекурсивной структуры данных.

4.2. ССЫЛКИ

Характерная особенность рекурсивных структур, четко отличающая их от основных структур вида массив, запись и множество, — способность изменять размер. Вследствие этого для рекурсивно определенных данных невозможно выделить память фиксированного размера, и поэтому транслятор не может сопоставить с компонентами таких переменных какие-либо определенные адреса. Для решения этой проблемы чаще всего используется метод, называемый *динамическим распределением* памяти, т. е. память под отдельные компоненты выделяется в момент, когда они «начинают существовать» в процессе выполнения программы, а не во время трансляции. Транслятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемой компоненты, а не самой этой компоненты. Может оказаться,

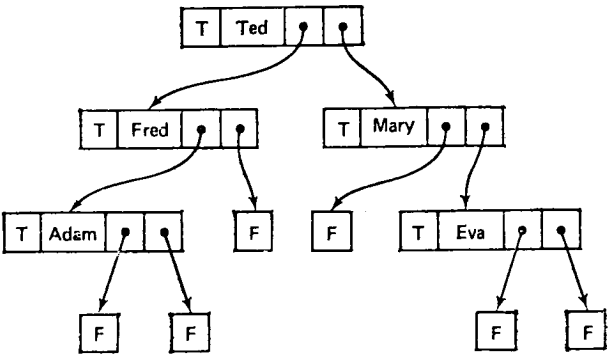


Рис. 4.3. Структура, построенная на ссылках.

что, например, генеалогическое дерево, изображенное на рис. 4.2, будет представлено в виде отдельных, не обязательно расположенных рядом записей — по одной для каждого человека. Эти записи связываются между собой с помощью адресов, находящихся в полях с именами *father* и *mother*. Графически такие связи удобнее всего изображать стрелками или ссылками (рис. 4.3).

Следует подчеркнуть, что использование ссылок при реализации рекурсивных структур — чисто технический прием. Программисту нет нужды знать об их существовании. Память может автоматически выделяться при первом упоминании новой переменной. Если, однако, явно ввести работу со ссылками (или указателями), то можно строить структуры более общего вида, чем те, которые следуют из рекурсивного определения данных. В частности, это позволяет вводить потенциально бесконечные или циклические структуры и указывать, что некоторая подструктура принадлежит нескольким разным структурам. Поэтому в современных языках программирования, как правило, есть механизмы манипулирования не только самими данными, но и ссылками на них. Это предполагает, что должна существовать четкая система, позволяющая отличать данные от ссылок на них. Поэтому необходимо вводить тип для данных, значения которых представляют собой ссылки (указатели) на

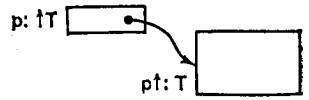


Рис. 4.4. Динамическое размещение переменной.

другие данные*). Мы используем для этого нотацию:

TYPET = POINTER TO T0 (4.4)

Описание типа (4.4) указывает**), что значения типа T являются ссылками на данные типа T0. Тот факт, что в описании T явно указывается тип данных, на которые идет ссылка, представляется нам фундаментальной особенностью. Мы в этом случае говорим, что T *связан* с T0. Понятие связанности отличает ссылки из языков высокого уровня от адресов из языков ассемблеров и представляет собой в программировании самое важное средство увеличения «безопасности» за счет избыточности написания.

Значения ссылочного типа формируются при каждом динамическом размещении каких-либо данных. Мы будем исходить из предположения, что такое размещение всегда явно фиксируется в противоположность предположению, что при первом упоминании элемента происходит его автоматическое размещение. Для этого мы вводим особую процедуру *Allocate* (разместить). Если есть ссылочная переменная p типа T, то оператор *Allocate*(*p*) действительно размещает в памяти некоторую переменную***) типа T0 и присваивает переменной p ссылку, указывающую на эту новую переменную. К самому значению ссылки теперь можно обращаться как p. А к переменной, на

*) Отметим, что это не единственный и поэтому не необходимый способ упомянутого четкого разграничения значений переменных и их адресов. В некоторых языках, скажем в Си или Bliss, для этого используются «синтаксические» механизмы. Например, a и &a или .a и a. — *Прим. перев.*

**) Именно поэтому мы переводим слово «pointer», как «ссылка», а не «указатель». — *Прим. перев.*

***) Точнее было бы сказать, что размещается неопределенное значение, ибо понятие переменной подразумевает наличие у нее некоторого имени. — *Прим. перев.*

которую указывает ссылка p, обращаются с помощью конструкции p↑.

Замечание: Если *Allocate* — процедура, импортированная из общего модуля распределения памяти, то для нее требуется явно указывать в качестве второго параметра размер переменной *).

Как уже упоминалось, для конечности мощности любого рекурсивного типа в нем должна всегда присутствовать какая-либо вариантная компонента. Наиболее типичный, часто встречающийся случай подобен генеалогическому дереву: поле признака принимает два значения (булевских), причем, если оно ложно, то все последующие компоненты отсутствуют. Этому соответствует такая схема описания:

```
TYPE T =  RECORD                                    (4.6)
            CASE terminal: BOOLEAN OF
              FALSE: S(T) |
              TRUE: (* пусто *)
            END
          END
```

Здесь S(T) — последовательность определений полей, включающая одно или несколько полей типа T, они и приводят к рекурсивности. Все структуры, описанные по схеме (4.6), представляют собой дерево (или список), подобное изображенному на рис. 4.3. Их неприятная особенность — наличие ссылок на данные, содержащие только поле признака, существенной информации тут нет. Реализации, основанные на ссылках, позволяют легко экономить память, включая информацию из поля признака в саму ссылку. Обычно принято расширять область значений любых ссылочных типов, добавляя к ним одно-единственное значение, которое не указывает ни на какой элемент. Такая ссылка обозначается специальным именем NIL, и считается, что NIL автоматически добавляется ко всем описанным ссылочным типам. Это расширение области ссылочных значений объясняет, почему можно формировать конечные структуры, не вводя явно варианты (или условия) в их (рекурсивные) описания.

*) Это означает, что при таком обращении соглашение о связанности p с определенным типом значений в результате ошибки программиста может быть нарушено. — *Прим. перев.*

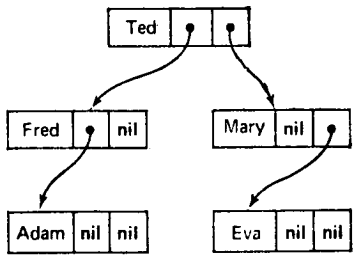


Рис. 4.5. Структура со ссылками, равными значению NIL.

В примерах (4.7) и (4.8) приводятся новые формулировки для тех же описаний типов, что и в (4.1) и (4.3), но теперь в них явно использованы ссылки. Обратите внимание, что в последнем случае (который первоначально был построен по схеме (4.6)) компонента записи с вариантами уже отсутствует, так как $\sim p.known$ теперь выражается как $p = NIL$. Изменение имени типа (от *ped* перешли к *person*, т. е. от «дерева» к «человеку») отражает изменение точки зрения, связанное с введением явных ссылочных величин. Вместо того чтобы рассматривать всю структуру как нечто целое и выделять из нее подструктуры и отдельные компоненты, мы теперь концентрируем внимание прежде всего на самих компонентах, а их взаимосвязь (представленная ссылками) вообще не фиксируется никакими описаниями.

```
TYPE termPtr =   POINTER TO term;                                   (4.7)

TYPE expPtr =    POINTER TO expression;

TYPE expression = RECORD op: operator;
                    opd1, opd2: termPtr
                  END;

TYPE term =      RECORD
                   CASE t: BOOLEAN OF
                     TRUE: id: alfa |
                     FALSE: sub: expPtr
                   END
                 END

TYPE PersonPtr = POINTER TO Person                                  (4.8)

TYPE person =    RECORD name: alfa;
                   father, mother: PersonPtr
                 END
```

На рис. 4.2 и 4.3 были приведены данные со структурой, представляющей генеалогическое дерево, а на

рис. 4.5 изображается то же дерево, но теперь ссылки на неизвестных лиц обозначены как NIL. Ясно, что в этом случае достигается значительная экономия памяти. Предположим, что Фред (Fred) и Мэри (Mary) (рис. 4.5) — брат и сестра, т. е. у них общие родители. Эта ситуация легко представляется заменой двух значений NIL в соответствующих полях обеих записей. При реализации со скрытыми ссылками или каком-либо другом методе работы с памятью программисту пришлось бы дублировать записи для их родителей — Адама (Adam) и Евы (Eve). Если речь идет о выборке данных, то не имеет значения, представлены ли оба отца (или обе матери) двумя записями или одной, но если данные нужно корректировать, то разница становится весьма существенной. Концепция ссылок как явных данных в отличие от любых «скрытых» реализаций помогает программисту точно указывать, где существует «разделение» памяти между несколькими объектами, а где — нет.

Еще одно следствие введения явных ссылок — возможность определения циклических структур и манипуляции ими. Однако такая дополнительная гибкость не только увеличивает выразительность языка, но и требует очень большой осторожности, так как работа с циклическими структурами может весьма быстро привести к бесконечным вычислениям.

То, что мощность и гибкость тесно связаны с опасностью злоупотребления, — факт в программировании хорошо известный. Здесь можно напомнить об операторе перехода (GOTO). Действительно. если продолжать аналогию между структурой программ и структурой данных, то данные с чисто рекурсивной структурой можно поставить на уровень процедур, а введение ссылок сопоставимо с использованием оператора перехода. Так же как с помощью оператора перехода можно строить любые программы (даже циклы), так и ссылки позволяют конструировать данные самой произвольной структуры (вплоть до циклической). В табл. 4.1 мы приводим некоторые параллели конструкций, относящихся к структуре программы и структуре данных.

В гл. 3 мы уже показали, что итерация — это частный случай рекурсии и обращение к рекурсив-

Таблица 4.1. Соответствие между элементами программ и структурой данных

«Образ конструкции»	Оператор программы	Тип данных
Атомарный объект	Присваивание	Скалярный тип
Перечисление	Составной оператор	Запись
Известное число повторений	Оператор цикла с параметром	Массив
Выбор	Условный оператор	Объединенный тип (запись с вариантами)
Повторение	Операторы цикла с пред- и постусловиями	Последовательность
Рекурсия	Оператор процедуры	Рекурсивный тип
Общий граф	Оператор перехода	Структура, построенная на ссылках

ной процедуре P, построенной в соответствии со схемой:

```
PROCEDURE P;
BEGIN
  IF B THEN P0; P END                                    (4.9)
END
```

где P0 — оператор, не содержащий P, — эквивалентен итеративному оператору

```
WHILE B DO P0 END
```

Аналогии, перечисленные в табл. 4.1, позволяют обнаружить такую же связь между типами рекурсивных данных и последовательностями.

```
TYPE T = RECORD
           CASE B: BOOLEAN OF                            (4.10)
             TRUE: t0: T0; t: T |
             FALSE:
           END
         END
```

Фактически рекурсивный тип, определенный в соответствии со схемой, где T0 — тип, не содержащий T, эквивалентен и может быть заменен на такой последовательностный тип:

```
SEQUENCE OF T0
```

Оставшуюся часть этой главы мы посвятим созданию структур, основанных на явных ссылках, и работе с ними. В частности, мы займемся структурами специфически простых видов, рецепты же работы с более сложными образованиями можно вывести из опыта работы с этими простыми. К таким простым видам структур мы относим списки — это самый простой случай — и деревья. Предпочтение, которое мы оказываем этим «строительным блокам» для построения сложных типов, не означает, что на практике не встречаются более сложные образования. Вот вам история, опубликованная в одной из цюрихских газет в июле 1922 г., доказывающая, что даже в таких регулярных структурах, как семейные деревья, могут встречаться иррегулярности. Это история человека, который так описывает несчастье своей жизни:

Я женился на вдове, у которой была взрослая дочь. Мой отец, довольно часто навещавший нас, влюбился в мою падчерицу и женился на ней. Следовательно, мой отец стал моим зятем, а моя падчерица — моей матерью. Спустя несколько месяцев моя жена родила сына, который стал шурином моего отца и одновременно моим дядей. У жены моего отца, т. е. моей падчерицы, тоже родился сын. Таким образом, у меня появился брат и одновременно внук. Моя жена является моей бабушкой, так как она мать моей матери. Следовательно, я муж моей жены и одновременно ее внук, другими словами, я — свой собственный дедушка.

4.3. ЛИНЕЙНЫЕ СПИСКИ

4.3.1. Основные операции

Наиболее простой способ объединить или связать некоторое множество элементов — это «вытянуть их в линию», организовать список или очередь. В этом случае с каждым элементом нужно сопоставить лишь одну-единственную ссылку, указывающую на следующий элемент.

Пусть есть два типа: *Node* и *Ptr*, определенные как (4.11). Каждая переменная типа *Node* состоит из

трех компонент, а именно идентифицирующего ключа (key), ссылки на следующий элемент (next) и, возможно, какой-либо другой информации, которую в (4.11) мы опускаем.

```
TYPE Ptr = POINTER TO Node;
TYPE Node = RECORD key: INTEGER;
              next: Ptr;                    (4.11)
              data: ...
            END;
VAR p, q: Ptr
```

На рис. 4.6 представлен список вершин, ссылка на первый элемент которого присвоена переменной p. Вероятно, самая простая операция, которую можно проделать с таким списком, — включить в его начало некоторый элемент. Сначала элемент типа *Node* размещается в памяти, и ссылка на него присваивается некоторой вспомогательной переменной q. После этого ссылкам присваиваются новые значения, и операция на этом заканчивается. Программируется это так:

```
Allocate(q, SIZE(Node)); q↑.next := p; p := q          (4.12)
```

Обратите внимание, что здесь существен порядок этих трех операторов.

Операция включения элемента в голову списка сразу же объясняет, как можно вообще формировать любой список: начать с пустого списка и последовательно добавлять в начало элементы. Целиком процесс *формирования списка* представлен фрагментом

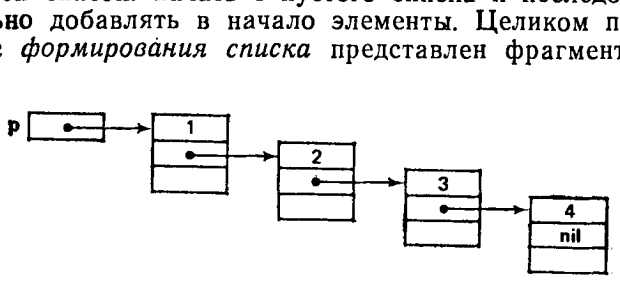


Рис. 4.6. Пример списка.

(4.13), число связываемых элементов — n.

```
p := NIL; (* в начале список пуст *)
WHILE n > 0 DO
  Allocate(q, SIZE(Node)); q↑.next := p; p := q;                    (4.13)
  q↑.key := n; n := n-1
END
```

Это самый простой способ построения списка, однако при нем порядок элементов в списке обратен порядку их включения. В некоторых случаях это нежелательно, и поэтому новые элементы необходимо добавлять не в голову, а в конец списка. Конец можно легко найти, просматривая весь список, но это слишком «наивное» решение, требующее определенных затрат, от которых просто избавиться. Достаточно ввести вторую ссылку q, указывающую на последний элемент. Такой метод применяется, например, в прогр. 4.4, где формируются перекрестные ссылки на заданный текст. Недостаток этого метода в том, что первый из включаемых элементов нужно обрабатывать иначе, чем остальные.

Явное использование ссылок намного упрощает некоторые операции, которые иначе были бы излишне запутанными. Среди элементарных операций над списками есть включение и исключение элементов (выборочное изменение списка) и, конечно же, просмотр списка. Начнем с разбора операции *включения в список.*

Предположим, после элемента списка, на который указывает ссылка p, нужно включить элемент, заданный ссылкой q (переменной). В (4.14) приведены при-

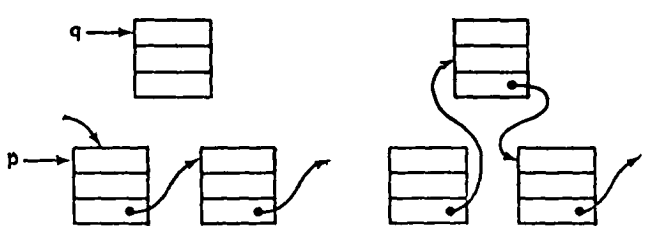


Рис. 4.7. Включение в список после элемента p↑.

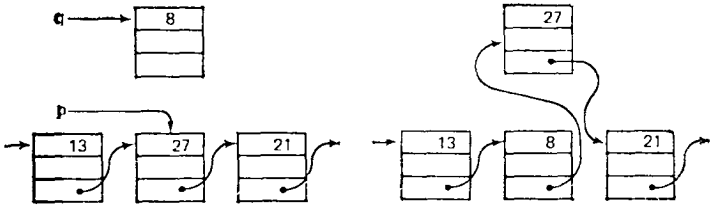


Рис. 4.8. Включение в список перед элементом p↑.

сваивания ссылкам, которые для этого нужно выполнить, а на рис. 4.7 — результат этих действий.

```
q↑.next := p↑.next; p↑.next := q                    (4.14)
```

Если требуется включение не после, а *перед* указанным p↑ элементом, то кажется, что однонаправленная цепочка связей должна затруднить работу, поскольку нет «хода» к элементам, предшествующим данному. Однако эта проблема решается довольно просто в (4.15), соответствующая схема приведена на рис. 4.8. (Здесь предполагается, что ключ нового элемента — key = 8.)

```
Allocate(q, SIZE(Node)); q↑ := p↑;
p↑.key := k; p↑.next := q                           (4.15)
```

«Трюк» заключается в том, что новая компонента на самом деле включается *после* p↑, но затем новая компонента и p↑ «меняются» значениями.

Теперь рассмотрим процесс *исключения из списка.* Исключение элемента, следующего за p↑, очевидно. В (4.16) оно приведено в комбинации с включением исключенного элемента в голову другого списка (на него указывает q). Рис. 4.9 иллюстрирует такой циклический обмен значениями трех ссылок.

```
r := p↑.next; p↑next := r↑. next;  r↑.next := q; q := r
                                                    (4.16)
```

Труднее исключить сам указанный элемент (а не следующий за ним), поскольку мы сталкиваемся с той же проблемой, что и при включении. Ведь проход к элементу, предшествующему указанному, невозможен. Однако теперь уже довольно очевидно простое реше-

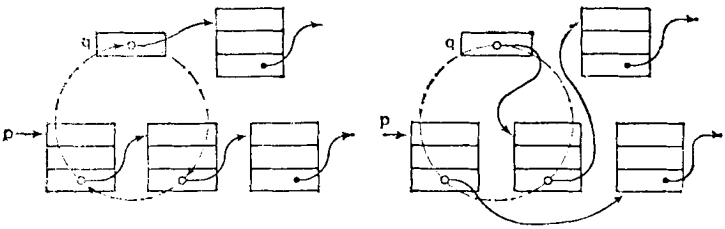


Рис. 4.9. Исключение из списка с последующим включением в другой список.

ние: исключается последующий элемент, а перед этим его значение «передвигается» вперед. Так можно поступать, когда у p↑ есть последующий, т. е. если это не последний элемент.

Разберем теперь основную операцию — *проход по списку*. Предположим, что с каждым из элементов списка нужно выполнить операцию P(x); первый элемент списка — p↑. Эту задачу можно выполнить следующим образом:

```
WHILE список, обозначенный через p, непуст DO
  выполнение операции P;
  переход к следующему
END
```

После уточнения эта операция выглядит уже так:

```
WHILE p # NIL DO
  P(p↑); p := p↑.next                                    (4.17)
END
```

Из определения оператора цикла с предусловием и списковой структуры следует, что P будет выполнено для всех элементов списка, и ни для каких других.

Одна из наиболее часто употребляемых операций — *поиск в списке* элемента с заданным ключом x. В отличие от массивов поиск в этом случае должен проходить строго последовательно. Он заканчивается либо при обнаружении элемента, либо при достижении конца списка. Такие условия приводят к тому, что конъюнкция включает два отношения. Как и ра-

нее, будем считать, что начало списка задается ссылкой p:

```
WHILE (p # NIL) & (p↑.key # x) DO p := p↑.next END
```

(4.18)

Отношение p = NIL предполагает, что p↑ не существует и, следовательно, выражение p↑.key#x само не определено. Поэтому порядок отношений весьма существен.

4.3.2. Упорядоченные списки и перестройка списков

Алгоритм (4.18) очень напоминает программу поиска при просмотре массива или последовательности. Ведь фактически последовательность — это точно такой же список, но в нем методы связи с последующим элементом остаются либо неопределенными, либо неявными. Поскольку основные операции над последовательностями не предусматривают включения новых элементов (кроме как в конец) или их исключения (кроме уничтожения всех элементов), то перед реализатором лежит широкое поле для выбора нужного метода и он может размещать элементы последовательно один за другим в «непрерывной» области памяти. Линейные списки с явными ссылками обеспечивают большую гибкость, и поэтому их нужно использовать всякий раз, когда такая гибкость действительно необходима.

Рассмотрим в качестве примера задачу, к которой будем постоянно обращаться на протяжении этой главы, демонстрируя возможные методы и приемы. Задача состоит в чтении текста, выборке из него всех слов и подсчете частоты их появления, т. е. нужно построить *алфавитный частотный словарь*.

Очевидное решение: строится список всех слов, найденных в тексте. Этот список просматривается с каждым словом. Если слово обнаруживается, то его счетчик частоты увеличивается на единицу, если же его в списке нет, то оно туда включается. Мы будем называть такой процесс просто *поиском*, хотя он, конечно же, содержит и включение. Чтобы сосредоточить внимание на основной работе со списками, мы

предположим, что слова уже выделены из исследуемого текста, закодированы целыми числами и находятся во входной последовательности.

Вид процедуры под названием *search* (поиск) непосредственно следует из алгоритма (4.18). Переменная *root* указывает на голову списка, в который добавляются в соответствии с (4.12) новые слова. Полностью весь алгоритм приведен в прогр. 4.1; здесь есть и подпрограмма печати списка полученных результатов. Печать такой таблицы служит примером действия, выполняемого по одному разу с каждым из элементов списка. Схема этого процесса уже приводилась (4.17).

Алгоритм линейного поиска из прогр. 4.1 напоминает процедуру поиска для массивов, где для упрощения условия окончания цикла используется распространенный прием — постановка барьера. Барьер можно использовать и при поиске в списке, его можно представить как пустой элемент в конце списка. В новой процедуре (4.21), заменяющей процедуру поиска в прогр. 4.1, предполагается, что добавлена глобальная переменная *sentinel*, а инициация *root* заменена на операторы, создающие элемент, выступающий в роли барьера.

```
Allocate(sentinel, SIZE(Node)); root := sentinel

PROCEDURE search(x: INTEGER; VAR root: Ptr);                    (4.21)
  VAR w: Ptr;
BEGIN w := root; sentinel↑.key := x;
  WHILE w↑.key # x DO w := w↑.next END;
  IF w = sentinel THEN (* новый элемент *)
    w := root; Allocate(root, SIZE(Node));
    WITH root↑ DO
      key := x; count := 1; next := w
    END
  ELSE w↑.count := w↑.count + 1
  END
END search
```

Ясно, что гибкость и мощность связанного списка в этом примере практически не используются, и поиск с помощью линейного просмотра можно допускать лишь в тех случаях, когда число его элементов огра-

```
MODULE List; (* прямое включение в список *
  FROM InOut IMPORT ReadInt, Done, WriteInt, WriteLn;
  FROM Storage IMPORT Allocate;

  TYPE Ptr = POINTER TO Word;
    Word =
      RECORD key: INTEGER;
        count: CARDINAL;
        next: Ptr
      END;
  VAR k: INTEGER; root: Ptr;

  PROCEDURE search(x: INTEGER; VAR root: Ptr);
    VAR w: Ptr;
  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; Allocate(root, SIZE(Word));
      WITH root↑ DO
        key := x; count := 1; next := w
      END
    ELSE w↑.count := w↑.count + 1
    END
  END search;

  PROCEDURE PrintList(w: Ptr);
  BEGIN
    WHILE w # NIL DO
      WriteInt(w↑.key, 8); WriteInt(w↑.count, 8); WriteLn;
      w := w↑.next
    END
  END PrintList;

BEGIN root := NIL; ReadInt(k);
  WHILE Done DO
    search(k, root); ReadInt(k)
  END;
  PrintList(root)
END List.
```

Прогр. 4.1. Простое включение в список.

ничено. Однако усовершенствованный метод лежит почти на поверхности — это *поиск в упорядоченном списке*. Если список упорядочен (предположим, в порядке увеличения ключей), то поиск можно заканчивать, как только будет обнаружен первый ключ со

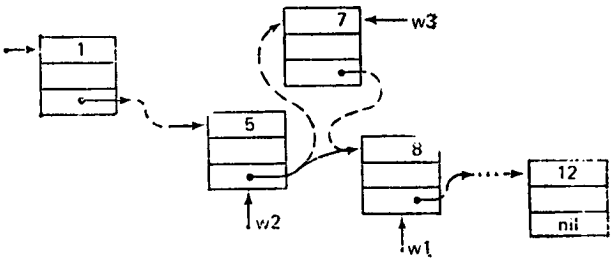


Рис. 4.10. Включение в упорядоченный список.

значением больше нового. Упорядоченность списка достигается включением нового элемента в подходящее для этого место, а не в голову списка. Поэтому упорядоченность получается фактически без дополнительных затрат, и здесь полностью используется гибкость употребляемой структуры. Ни массивы, ни последовательности такой возможности не дают. (Заметим, однако, что даже упорядоченные списки не позволяют организовать ничего похожего на двоичный поиск в массивах.)

Поиск в упорядоченных списках представляет собой типичный пример ситуации (4.15), где элемент нужно было включать перед заданным. И здесь его нужно включать перед первым элементом с ключом, большим чем заданный. Но это делается не так, как делали в (4.15). Мы не копируем значения, а в процессе прохода по списку всегда имеем дело с *двумя* ссылками: w2 отстает на один шаг от w1. Это позволяет точно определить место включения, когда по w1 будет обнаружен больший ключ. Общая схема такого включения приведена на рис. 4.10. Если только список не пуст, то ссылка на новый элемент (w3) присваивается *w2*↑*.next*. Стремясь к простоте и эффективности, мы не будем различать эту ситуацию с помощью условного оператора. Добавим просто в голову списка пустой элемент (фиктивный). Соответственно инициирующий оператор *root* := *NIL* из прогр. 4.1 заменим на такие операторы:

```
Allocate(root, SIZE(Node)); root↑.next := NIL
```

Из рис. 4.10 ясно, что условие, определяющее продолжение просмотра, т. е. переход к следующему элементу, содержит два члена, а именно:

(w1 # NIL) & (w1^.key < x)

В результате получаем такую процедуру поиска (4.23):

```
PROCEDURE search(x: integer; VAR root: Ptr);
  VAR w1, w2: Ptr;                                          (4.26)
BEGIN w1 := root; sentinel↑.key := x;
  IF w1 = sentinel THEN (* первый элемент *)
    Allocate(root, SIZE(Node));
    WITH root↑ DO
      key := x; count := 1; next := sentinel
    END
  ELSIF w1↑.key = x THEN w1↑.count := w1↑count + 1
  ELSE (* поиск *)
    REPEAT w2 := w1; w1 := w2↑.next
    UNTIL w1↑.key = x;
    IF w1 = sentinel THEN (* новый элемент *)
      w2 := root; Allocate(root, SIZE(Node));
      WITH root↑ DO
        key := x; count := 1; next := w2
      END
    ELSE (* найден, теперь переупорядочение *)
      w1↑.count := w1↑.count + 1;
      w2↑.next := w1↑.next; w1↑.next := root; root := w1
    END
  END
END search
```

Для убыстрения поиска условие продолжения из заголовка оператора цикла можно опять значительно упростить, если воспользоваться барьером. Так что в начале процесса в голове списка должен стоять пустой элемент, а в хвосте — барьер. Теперь самое время спросить: какого выигрыша можно ожидать от поиска в упорядоченном списке? Если вспомнить, что дополнительные усложнения были невелики, то не следует ожидать и потрясающих улучшений.

Допустим, что все слова встречаются в тексте с одинаковой частотой. В этом случае выигрыш от лексикографической упорядоченности, как только все слова окажутся в списке, станет нулевым. Ведь ме-

стоположение слова в списке уже не имеет значения, поскольку важны общие затраты и все слова встречаются с одинаковой частотой. Однако при включении любого нового слова некоторый выигрыш получается. Вместо всего списка в среднем просматривается лишь около половины. Следовательно, включение в упорядоченный список стоит использовать лишь для построения словаря с большим по сравнению с частотой их повторения количеством различных слов. Поэтому приведенные нами процедуры надо рассматривать скорее как примеры программирования, а не как практические инструменты.

Мы рекомендуем представлять данные в виде связанных списков в тех случаях, когда число элементов меняется относительно мало (<50), и, кроме того, нет никаких сведений о частоте их использования. Типичный пример — таблица имен в трансляторах для языков программирования. Каждое описание приводит к добавлению нового имени, а при выходе из области действия его описания это имя из списка вычеркивается. Простые связанные списки особенно подходят для относительно коротких программ. Даже в этом случае, однако, можно добиться значительного улучшения доступа, если воспользоваться очень простым приемом. И мы вновь к нему возвращаемся, поскольку это великолепная демонстрация гибкости структуры связанных списков.

Для текстов программ характерно появление «скоплений идентификаторов», т. е. за одним появлением некоторого идентификатора часто следуют другие. Это наблюдение наводит на мысль, что список надо после каждого обращения реорганизовывать: передвигать найденный элемент в голову списка и тем самым минимизировать поиск в случае повторного обращения за этим же элементом. Такой метод доступа называется «поиском с переупорядочением списка» или, несколько претенциозно, поиском в самоорганизующемся списке*). Описывая соответствующий алгоритм в виде процедуры, которую можно подставить

*) Дальше мы будем употреблять более короткое название «поиск с самоорганизацией». — *Прим. перев*

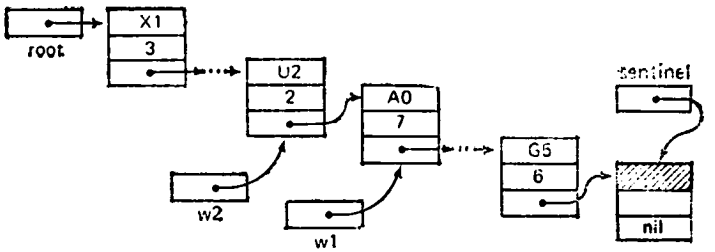


Рис. 4.11. Список до переупорядочения.

в прогр. 4.1, мы воспользуемся предыдущим опытом и с самого начала введем барьер. Действительно, барьер не только ускоряет поиск, но и упрощает саму программу. Список в самом начале работы уже должен содержать барьер. Это делается такими инициирующими операторами:

```
Allocate(sentinel, SIZE(Node)); root := sentinel;
```

Заметим, что основное различие между новым алгоритмом и прямым поиском по списку (4.21) заключается в реорганизации списка при каждом обнаружении элемента. Он вычеркивается или удаляется со своего старого места и передвигается в голову. Удаление элемента вновь требует введения двух ссылок: w1 задает местоположение идентифицированного элемента, а w2 — предшествующего ему. Это в свою очередь приводит к выделению особого случая — пустого списка. Чтобы наглядно представить процесс изме-

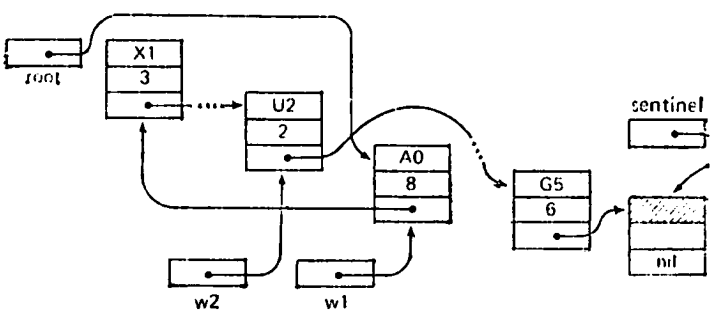


Рис. 4.12. Список после переупорядочения.

нения связей, обратимся к рис. 4.11. Здесь изображены две ссылки в момент, когда w1 указывает на нужный элемент. На рис. 4.12 представлена ситуация после соответствующего переупорядочения. Ниже (4.26) приводится целиком новая процедура поиска.

```
PROCEDURE search(x: INTEGER); VAR root: Ptr);                    (4.25)
  VAR w1, w2, w3: Ptr;
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 *)
    Allocate(w3, SIZE(Node)); w2↑.next := w3;
    WITH w3↑ DO
      key := x; count := 1; next := w1
    END
  ELSE w1↑.count := w1↑.count + 1
  END
END search
```

Выигрыш при таком методе поиска сильно зависит от того, насколько «четки» скопления во входном тексте. При заданном коэффициенте скопления улучшение более ощутимо для больших списков. Для того чтобы получить представление о примерной величине ожидаемого выигрыша, были проделаны опытные измерения. Мы применили нашу программу составления частотного словаря к одному короткому и одному относительно большому тексту и сравнили метод поиска в упорядоченном списке (4.21) с методом поиска с самоорганизацией (4.26). Полученные результаты приводятся в табл. 4.2. К сожалению, наибольший вы-

Таблица 4.2. Сравнение методов поиска в списках

	Тест 1	Тест 2
Число различных ключей	53	582
Всего ключей	315	14 311
Время поиска с упорядочением	6 207	3 200 622
Время поиска с самоорганизацией	4 529	681 581
Коэффициент улучшения	1.37	4.70

игрыш достигается в случае, когда вообще требуется другая организация данных. К этому примеру мы еще вернемся в разд. 4.4.

4.3.3. Приложение: топологическая сортировка

Хороший пример использования данных с гибкой, динамической структурой — процесс *топологической сортировки*. Имеется в виду сортировка элементов, для которых определено отношение частичного порядка, т. е. порядок задан не для всех, а лишь для некоторых пар. Приведем несколько примеров такой упорядоченности:

1. В толковом словаре слова определяются с помощью других слов. Если слово v определяется с помощью слова w, то обозначим это как $v \prec w$. Топологическая сортировка означает такое распределение слов в словаре, при котором ссылки вперед отсутствуют.

2. Любая задача (например, технический проект) распадается на ряд подзадач. Выполнение некоторых подзадач обычно начинается следом за окончанием других. Если подзадача v должна предшествовать подзадаче w, то мы будем писать $v \prec w$. Топологическая сортировка означает такое распределение работ, при котором каждая из подзадач не начинается до тех пор, пока не закончатся все подзадачи, предшествующие данной.

3. В университетских программах некоторые курсы должны читаться раньше других, так как последние основываются на изложенном в них материале. Если для курса w требуется знакомство с курсом v, то пишем $v \prec w$. Топологическая сортировка означает, что ни один курс не читается ранее тех, которые его поддерживают.

4. В программе некоторые процедуры могут содержать обращения к другим процедурам. Если процедура v вызывается из процедуры w, то это обозначается как $v \prec w$. Топологическая сортировка предполагает такое распределение описаний процедур, при котором нет ссылок вперед.

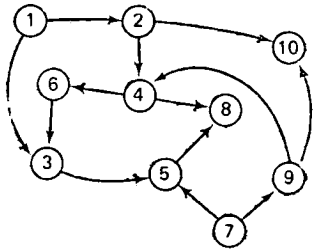


Рис. 4.13. Частично упорядоченное множество.

В общем, частичный порядок на множестве S — это отношение между элементами S. Отношение порядка обозначается символом $\prec$, читается «предшествует» и удовлетворяет следующим трем свойствам (аксиомам): для любых трех элементов x, y и z из S:

(1) если $x \prec y$ и $y \prec z$, то $x \prec z$ (транзитивность)

(2) если $x \prec y$, то не выполняется $y \prec z$ (асимметричность)

(3) не выполняется $z \prec z$ (иррефлексивность) (4.27)

По вполне понятным соображениям мы будем предполагать, что множество S, подвергаемое топологической сортировке, должно быть конечным. Следовательно, частичную упорядоченность можно представить в виде диаграммы или графа, вершины которого — элементы S, а направленные дуги — отношения порядка. На рис. 4.13 приведен пример такого графа.

Цель топологической сортировки — преобразовать частичный порядок в линейный. Графически это означает, что вершины графа нужно расположить на одной прямой так, чтобы все стрелки указывали вправо (см. рис. 4.14). Свойства (1) и (2) частичного порядка гарантируют отсутствие циклов. Это как раз то условие, при котором возможно преобразование к линейному порядку.

Что же нужно проделать, чтобы найти один из возможных линейных порядков? Рецепт крайне прост. Начнем с элемента, у которого нет ни одного предшествующего (по крайней мере один такой должен существовать, иначе в графе будут циклы). Этот эле-

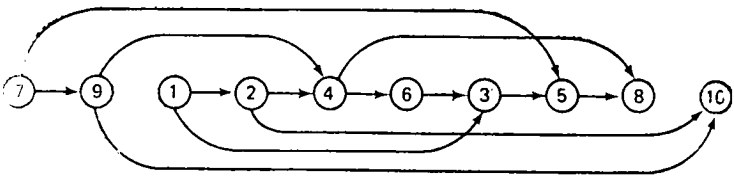


Рис. 4.14. Линейное расположение частично упорядоченного множества с рис. 4.13.

мент помещается в голову результирующего списка и вычеркивается из множества S. Остающееся множество продолжает оставаться частично упорядоченным, так что к нему можно вновь и вновь применять тот же алгоритм, пока оно не станет пустым.

Для того чтобы более строго описать наш алгоритм, нужно выбрать структуру представления S и его отношения порядка. Выбор представления диктуется операциями, которые необходимо выполнять, и в частности, операцией выбора элемента, не имеющего предшественников. Поэтому каждый элемент надо было бы представлять с помощью трех характеристик: идентифицирующим его ключом, множеством следующих за ним элементов-«последователей» и счетчиком предшествующих ему элементов. Поскольку *заранее* число элементов в S не задано, то множество удобнее организовывать в виде связанного списка. Следовательно, в дескрипторе каждого элемента есть еще дополнительная строчка, содержащая ссылку на следующий элемент списка. Мы будем считать, что ключи — это целые числа (не обязательно последовательные и из диапазона 1...n). Аналогично и множество последователей каждого из элементов удобно представлять в виде связанного списка. Элемент списка последователей каким-то образом идентифицирован и связан со следующим элементом из того же списка. Если мы назовем дескрипторы главного списка, где каждый из элементов S содержится ровно один раз, *ведущими* (*leaders*), а дескрипторы списка последователей — *ведомыми* (*trailers*), то получим такие

описания соответствующих типов:

```
TYPE LPtr = POINTER TO leader;
     TPtr = POINTER TO trailer;

     leader = RECORD key, count: INTEGER;
                trail: TPtr; next: LPtr
              END;                                    (4.28)

     trailer = RECORD id: LPtr; next: TPtr
               END
```

Предположим, что множество S и отношения порядка на нем в начале представлены парами ключей из входного файла. Исходные данные для рис. 4.13 приведены в (4.29), где мы еще для ясности добавляем символы $\prec$.

$$\begin{array}{llllllll} 1\prec 2 & 2\prec 4 & 4\prec 6 & 2\prec 10 & 4\prec 8 & 6\prec 3 & 1\prec 3 & \\ 3\prec 5 & 5\prec 8 & 7\prec 5 & 7\prec 9 & 9\prec 4 & 9\prec 10 & & \end{array} \qquad (4.29)$$

В первой части программы топологической сортировки необходимо ввести исходные данные и построить соответствующий список. Для этого последовательно читаются пары ключей x и y ($x\prec y$). Обозначим через p и q ссылки на их представление в связанном списке лидеров. Появляющиеся записи *) ищутся в списке и, если они там отсутствуют, то добавляются к нему. Эту работу выполняет процедура-функция *find*. Затем в список ведомых для x добавляется элемент, идентифицирующий y, и счетчик предшественников y этого y увеличивается на единицу. Соответствующий алгоритм будем называть *фазой ввода* (4.30). На рис. 4.15 представлена структура данных, сформированных в процессе обработки входных данных (4.29) с помощью программы (4.30). Функция *find*(*w*) поставляет ссылку на компоненту списка с ключом w (см. также прогр. 4.2).

```
(* фаза ввода *)
Allocate(head, SIZE(leader)); tail := head; z := 0; ReadInt(x);        (4.30)
WHILE Done DO
  ReadInt(y); p := find(x); q := find(y);
  Allocate(t, SIZE(trailer)); t↑.id := q; t↑.next := p↑.trail;
  p↑.trail := t; q↑.count := q↑.count + 1; ReadInt(x)
END
```

*) Дескрипторы — это объекты типа запись (record). — *Прим. перев.*

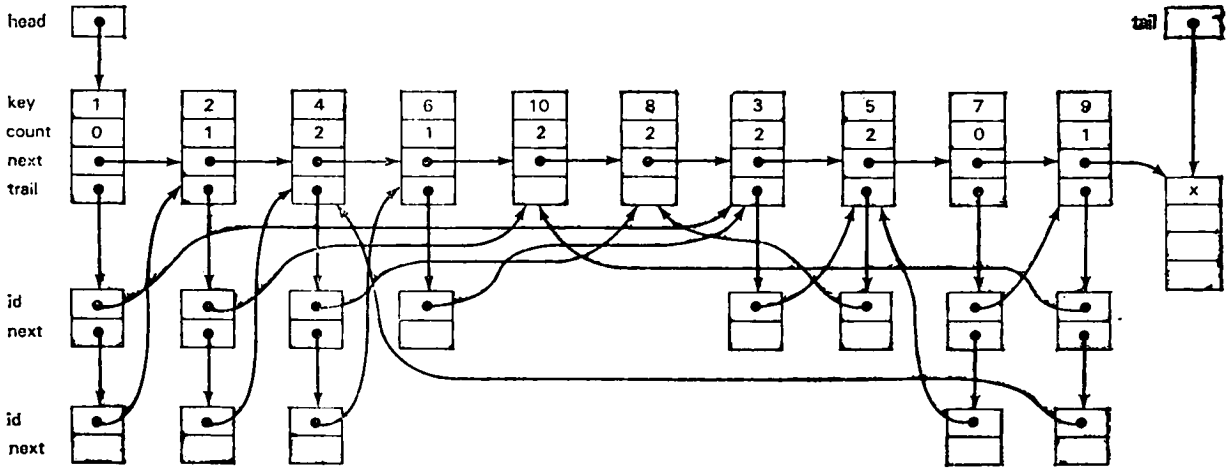


Рис. 4.15. Структура списка, сформированного программой.

После того как на фазе ввода построены данные со структурой, приведенной на рис. 4.15, можно проводить описанную выше сортировку. Однако поскольку она состоит из повторяющихся выборов элемента с нулевым числом предшественников, то разумно вначале собрать все такие элементы в одну связанную цепочку. Так как ясно, что исходная цепочка ведущих больше нам уже не нужна, то то же самое поле под именем next можно повторно использовать для образования цепочки ведущих, не имеющих предшественников. Такая операция замены одной цепочки на другую в работе со списками встречается часто. Детально она описывается в (4.31), причем новую цепочку удобнее строить в обратном порядке. Если обратиться к рис. 4.15, то увидим, что цепочка ведущих *next* заменяется на цепочку, представленную на рис. 4.16. Связи, не показанные на этом рисунке, остаются прежними.

```
(* поиск ведущего без предшественников *)
p := head; head := NIL;
WHILE p # tail DO
  q := p; p := q↑.next;                                    (4.31)
  IF q↑.count = 0 THEN (* включение в новую цепочку *)
    q↑.next := head; head := q
  END
END
```

После всех этих подготовительных действий мы получаем удобное представление для частично упорядоченного множества S и можем, наконец, приступить к собственно топологической сортировке, т. е. формированию выходной последовательности.

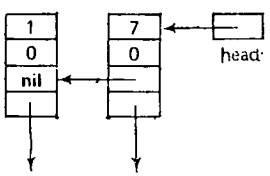


Рис. 4.16. Список ведущих с нулевыми счетчиками.

В первом приближении этот процесс можно описать так:

```
q := head;
WHILE q # NIL DO (* вывод элемента и исключение *)
  WriteInt(qt.key, 8); n := n - 1;
  t := qt.trail; q := qt.next;                                    (4.32)
  уменьшить счетчик предшественников у всех его
  последователей из списка ведомых; если счетчик = 0,
  перевести в список ведущих q
END
```

Оператор, который в (4.32) еще необходимо уточнить, состоит из одного просмотра списка (см. схему (4.17)). На каждом шаге вспомогательная переменная p указывает на ведущий элемент, у которого нужно уменьшить и проверить счетчик.

```
WHILE t # NIL DO
  p := tt.id; pt.count := pt.count - 1;                           (4.33)
  IF pt.count = 0 THEN (* включение p↑ в список ведущих *)
    pt.next := q; q := p
  END;
  t := tt.next
END
```

На этом программа топологической сортировки заканчивается. Обратите внимание, что для подсчета ведущих элементов, образованных на этапе ввода, был введен счетчик n. На фазе вывода каждый раз, когда выводится ведущий элемент, этот счетчик уменьшается. Поэтому в конце работы программы он должен стать нулевым. Если же он не нулевой, то это указывает, что остались еще элементы, причем у всех есть предшественники. Это, очевидно, случай, когда множество S не было частично упорядоченным. Приведенная выше программа фазы вывода служит примером работы с «пульсирующим» списком, т. е. списком, в который элементы включаются или из которого исключаются в непредсказуемом порядке. Следовательно, это пример процесса, полностью использующего все достоинства списков с явными связями.

```
MODULE TopSort;
  FROM InOut IMPORT OpenInput, CloseInput,
    ReadInt, Done, WriteInt, WriteString, WriteLn;
  FROM Storage IMPORT ALLOCATE;

  TYPE LPtr = POINTER TO leader;
    TPtr =    POINTER TO trailer;

    leader =  RECORD key, count: INTEGER;
                trail: TPtr; next: LPtr
              END;

    trailer = RECORD id: LPtr; next: TPtr
              END;

  VAR p, q, head, tail: LPtr;
    t: TPtr;
    x, y, n: INTEGER;

  PROCEDURE find(w: INTEGER): LPtr;
    VAR h: LPtr;
  BEGIN h := head; tail^.key := w; (* барьер *)
    WHILE h^.key # w DO h := h^.next END;
    IF h = tail THEN
      ALLOCATE(tail, SIZE(leader)); n := n+1;
      h^.count := 0; h^.trail := NIL; h^.next := tail
    END;
    RETURN h
  END find;

BEGIN
  (* инициация списка ведущих *)
  ALLOCATE(head, SIZE(leader)); tail := head; n := 0;

  OpenInput("TEXT"); ReadInt(x);
  WHILE Done DO
    WriteInt(x, 8); ReadInt(y); WriteInt(y, 8); WriteLn;
    p := find(x); q := find(y);
    ALLOCATE(t, SIZE(trailer)); t^.id := q; t^.next := p^.trail;
    p^.trail := t; q^.count := q^.count + 1; ReadInt(x)
  END;
  CloseInput;
```

```
  (* поиск ведущих без предшественников *)
  p := head; head := NIL;
  WHILE p # tail DO
    q := p; p := q↑.next;
    IF q↑.count = 0 THEN (* включение q↑ в новую цепочку *)
      q↑.next := head; head := q
    END
  END;

  (* фаза вывода *) q := head;
  WHILE q # NIL DO
    WriteLn; WriteInt(q↑.key, 8); n := n-1;
    t := q↑.trail; q := q↑.next;
    WHILE t # NIL DO
      p := t↑.id; p↑.count := p↑.count - 1;
      IF p↑.count = 0 THEN (* включение p↑ в список ведущих *)
        p↑.next := q; q := p
      END;
      t := t↑.next
    END
  END;
  IF n # 0 THEN WriteString("This set is not partially ordered") END;
  WriteLn
END TopSort.
```

Прогр. 4.2. Топологическая сортировка.

4.4. ДЕРЕВЬЯ

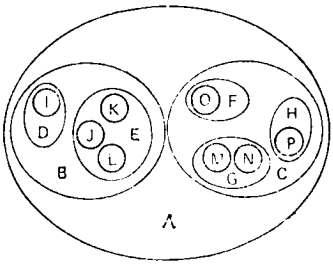
4.4.1. Основные понятия и определения

Как мы уже убедились, последовательности и списки удобно определять таким образом: последовательность (список) с базовым типом T — это либо:

1) пустая последовательность (список); либо

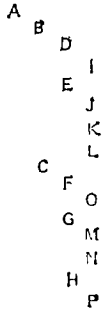
2) конкатенация (соединение) элемента типа T и некоторой последовательности с базовым типом T.

Для определения принципов построения, а именно следования или итерации, здесь используется рекурсия. Следование и итерация встречаются настолько часто, что обычно их считают фундаментальными образами строения данных и поведения программ. Однако следует всегда помнить, что их можно определять только с помощью рекурсии (обратное неверно), в то время как рекурсии можно эффективно и элегантно употреблять для определения значительно бо-

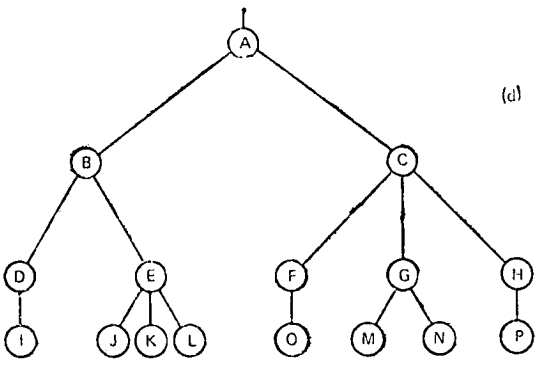


(a)

(A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P)))) (b)



(c)



(d)

Рис. 4.17. Представление древовидной структуры: a) вложенные множества, b) вложенные скобки, c) отступы, d) граф.

лее сложных структур. Хорошо известным примером служат деревья. Определим дерево таким образом: *дерево* с базовым типом T — это либо:

1) пустое дерево; либо

2) некоторая вершина типа T с конечным числом связанных с ней отдельных деревьев с базовым типом T, называемых *поддеревьями*.

Из сходства рекурсивных определений последовательностей и деревьев ясно, что последовательность (список) есть дерево, в котором каждая вершина имеет не более одного поддерева. Поэтому последовательность (список) называют иногда и *вырожденным* деревом.

Существует несколько способов изображения структуры дерева. На рис. 4.17 приведено несколько примеров таких структур, где базовый тип T — множество букв. Эти схемы относятся к одной и той же структуре и, следовательно, эквивалентны. Структура, представленная в виде графа и явно отражающая разветвления, по понятным причинам привела к появлению общеупотребительного термина «дерево». Однако довольно странно, что деревья принято рисовать перевернутыми или, можно считать и так, изображать только его корни. Как бы то ни было, и последнее толкование вводит в заблуждение, так как верхнюю вершину (A) обычно называют корнем.

Упорядоченное дерево — это дерево, у которого ребра (ветви), исходящие из каждой вершины, упорядочены. Поэтому два упорядоченных дерева на рис. 4.18 — это разные, отличные друг от друга объекты. Вершина y, находящаяся непосредственно ниже вершины x, называется непосредственным *потомком* x; если x находится на уровне i, то говорят, что y лежит на уровне i + 1. И наоборот, вершину x назы-

Рис. 4.18. Два различных двоичных дерева.

вают (непосредственным) *предком* *) у. Считается, что корень дерева находится на уровне 0. Максимальный уровень какой-либо из вершин дерева называется его *глубиной* или *высотой*.

Если элемент не имеет потомков, то его называют *терминальной* вершиной или *листом*, а нетерминальную вершину называют *внутренней*. Число непосредственных потомков внутренней вершины называют ее *степенью*. Максимальная степень всех вершин есть степень дерева. Число ветвей или ребер, которые нужно пройти от корня к вершине x, называется *длиной пути* к x. Корень имеет путь 0, его прямые потомки имеют путь длиной 1 и т. д. Вообще, вершина на уровне i имеет длину пути i. Длина пути всего дерева определяется как сумма длин путей для всех его компонент. Ее также называют *длиной внутреннего пути*. Например, длина внутреннего пути дерева, изображенного на рис. 4.17, равна 36. Очевидно, что средняя длина пути

$$P_I = (Si : 1 \leqslant i \leqslant n : n_i * i)/n \qquad (4.34)$$

где n_i — число вершин на уровне i. Для того чтобы определить, что называется длиной внешнего пути, дополним дерево специальными вершинами в тех местах, где в исходном дереве отсутствуют поддеревья. Причем будем предполагать, что все вершины должны быть одной и той же степени, а именно степени дерева. Следовательно, такое расширение дерева порождает вместо пустых ребер массу специальных вершин, которые, конечно, уже не имеют потомков. На рис. 4.19 показано такое расширенное специальными вершинами дерево с рис. 4.17, специальные вершины отмечены квадратиками. Длина внешнего пути теперь может быть определена как сумма длин путей всех специальных вершин. Если число специальных вершин на уровне i равно m_i, то средняя длина внешнего пути равна

$$P_E = (Si : 1 \leqslant i \leqslant m : m_i * i)/m \qquad (4.35)$$

*) Иногда употребляют термин «родитель» и очень часто говорят об «отце» и «сыне», поскольку это сразу объясняет термины «брат» и даже «дядя»! — *Прим. перев.*

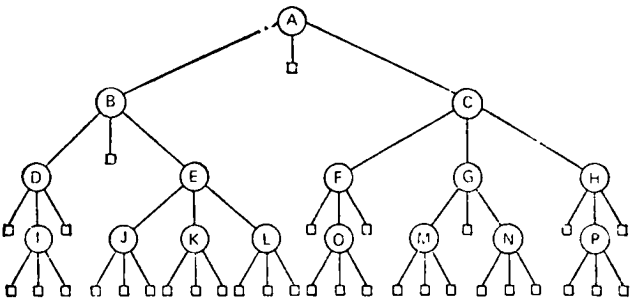


Рис. 4.19. Тернарное дерево со специальными вершинами.

Длина внешнего пути дерева с рис. 4.19 равна 120. Число специальных вершин m, которые добавляются к дереву степени d, прямо зависит от числа его исходных вершин — n. Обратите внимание: к каждой вершине ведет точно одно ребро. Таким образом, в расширенном дереве — всего m + n ребер. С другой стороны, из каждой исходной вершины выходят d ребер, а из специальных — ни одного. Поэтому всего имеется d*n + 1 ребро (1 дает ребро, ведущее к корню). Из этих двух формул мы получаем уравнение, связывающее число специальных вершин — m с числом исходных вершин — n:

$$m = (d - 1) * n + 1 \qquad (4.36)$$

Максимальное число вершин в дереве высотой h достигается в том случае, если из каждой вершины, за исключением находящихся на уровне h, исходят d поддеревьев. В этом случае для деревьев степени d на уровне 0 находится одна вершина (корень), на уровне 1 — d ее потомков, на уровне 2 — d^2 потомков d вершин уровня 1 и т. д. Отсюда получаем

$$N_d(h) = \mathrm{Si} : 0 \leqslant i < h : d^i \qquad (4.37)$$

— максимальное число вершин в дереве высотой h и степени d. При d = 2 имеем

$$N_2(h) = 2^h - 1 \qquad (4.38)$$

Особенно важную роль играют упорядоченные деревья второй степени. Их называют *двоичными* (или

бинарными) *деревьями*. Мы определим упорядоченное двоичное дерево как конечное множество элементов (вершин), которое либо пусто, либо состоит из корня (вершины) с двумя отдельными двоичными деревьями, которые называются *левым* и *правым поддеревом* этого корня. Везде далее в этом разделе мы будем иметь дело только с двоичными деревьями, поэтому если мы говорим «дерево», то это означает «упорядоченное двоичное дерево». Деревья степени больше двух называются *сильно ветвящимися деревьями* (multiway trees), о них речь пойдет в разд. 5 данной главы.

Знакомые для нас примеры двоичного дерева: генеалогическое (семейное) дерево, где у каждого человека есть потомки (!) в лице отца и матери; схема теннисного турнира, где каждая игра — это вершина, обозначенная ее победителем, а предки — две предыдущие игры соперников; арифметическое выражение с бинарными операциями, где каждому оператору соответствует вершина, а операнды — поддеревья (см. рис. 4.20).

Вернемся теперь к проблеме представления деревьев. Ясно, что попытка выражения таких рекурсивных структур в терминах разветвлений немедленно приводит к использованию ссылок. Столь же очевидно, что не следует описывать переменные с фиксированной структурой дерева, вместо этого мы будем описывать как переменные с фиксированной структурой сами вершины, т. е. их тип будет зафиксирован и степень дерева будет определять число

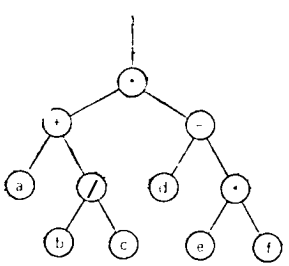


Рис. 4.20. Представление выражения (a + b/c)*(d — e*f) в виде дерева.

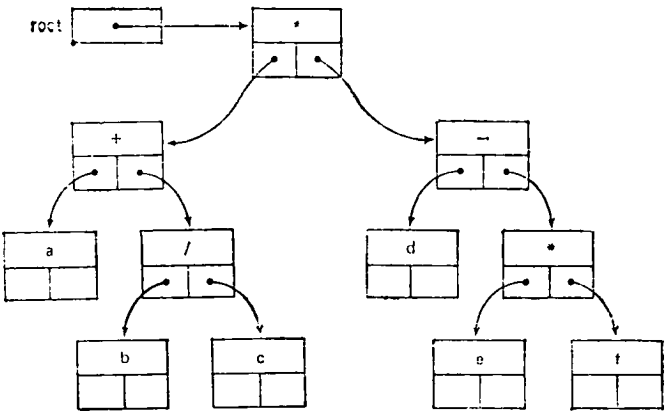


Рис. 4.21. Дерево, представленное как структура данных.

ссылочных компонент, указывающих на вершины поддеревьев. Ссылки на пустые деревья, конечно, будут обозначаться значением NIL. Таким образом, дерево на рис. 4.20 можно представить так, как на рис. 4.21, а его компоненты имеют такой тип:

```
TYPE Ptr =  POINTER TO Node;
TYPE Node = RECORD op: CHAR;
              left, right: Ptr
            END
```
(4.39)

Прежде чем обсуждать, как можно было бы воспользоваться преимуществами деревьев и как выполнять операции над ними, приведем пример, как программа может строить сами деревья. Предположим, что дерево, которое требуется сформировать, содержит вершины, относящиеся к типу (4.39), а значения этих вершин — числа, поступающие из входного файла. Чтобы сделать задачу более интересной, договоримся, что надо строить дерево минимальной глубины и состоящее из n вершин. Очевидно, минимальная высота при заданном числе вершин достигается, если на всех уровнях, кроме последнего, помещается опять же максимально возможное число вершин. Этого просто добиться, размещая приходящие вершины поровну слева и справа от каждой вершины. В результате мы

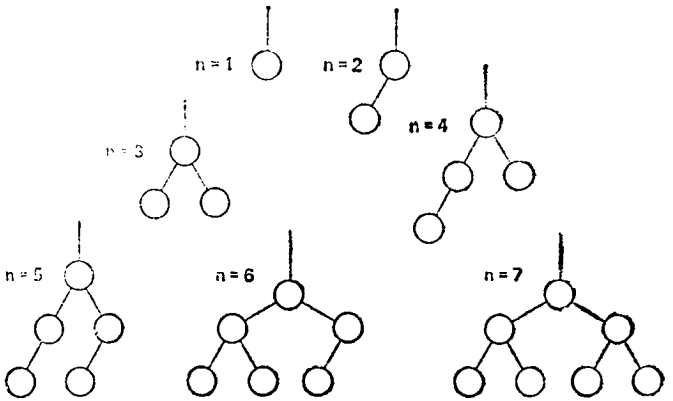


Рис. 4.22. Идеально сбалансированные деревья.

будем получать деревья со структурой, приведенной на рис. 4.22 для n = 1, ..., 7.

Правило равномерного распределения для известного числа вершин n лучше всего сформулировать, используя рекурсию:

1. Взять одну вершину в качестве корня.

2. Построить тем же способом левое поддерево с nl = n DIV 2 вершинами.

3. Построить тем же способом правое поддерево с nr = n — nl — 1 вершиной.

Этому правилу соответствует рекурсивная процедура, включенная в прогр. 4.3. Программа читает входной файл и строит идеально сбалансированное дерево. Причем мы исходим из следующего определения: дерево называется *идеально сбалансированным*, если число вершин в его левых и правых поддеревьях отличается не более чем на 1.

Предположим, например, что на вход поступают такие данные для дерева с 21 вершиной:

21 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18

В этом случае прогр. 4.3 построит идеально сбалансированное дерево, показанное на рис. 4.23. Обратите внимание на простоту и ясность нашей программы, полученные за счет использования рекурсивных про-

```
MODULE BuildTree;
  FROM InOut IMPORT OpenInput, CloseInput,
    ReadInt, WriteInt, WriteString, WriteLn;
  FROM Storage IMPORT ALLOCATE;

  TYPE Ptr = POINTER TO Node;

  Node = RECORD key: INTEGER;
         left, right: Ptr
    END;

  VAR n: INTEGER; root: Ptr;

  PROCEDURE tree(n: INTEGER): Ptr;
    VAR newnode: Ptr;
      x, nl, nr: INTEGER;
  BEGIN (* построение идеально сбалансированного дерева с n вершинами *)
    IF n = 0 THEN newnode := NIL
    ELSE nl := n DIV 2; nr := n-nl-1;
      ReadInt(x); ALLOCATE(newnode, SIZE(Node));
      WITH newnode^ DO
        key := x; left := tree(nl); right := tree(nr)
      END
    END;
    RETURN newnode
  END tree;
  PROCEDURE PrintTree(t: Ptr; h: INTEGER);
    VAR i: INTEGER;
  BEGIN (* печать дерева t с отступом h *)
    IF t # NIL THEN
      WITH t^ DO
        PrintTree(left, h+1);
        FOR i := 1 TO h DO WriteString("   ") END;
        WriteInt(key, 6); WriteLn;
        PrintTree(right, h+1)
      END
    END
  END PrintTree;

BEGIN (* первое целое – число вершин *)
  OpenInput("TEXT"); ReadInt(n);
  root := tree(n);
  PrintTree(root, 0); CloseInput
END BuildTree.
```

Прогр. 4.3. Построение идеально сбалансированного дерева.

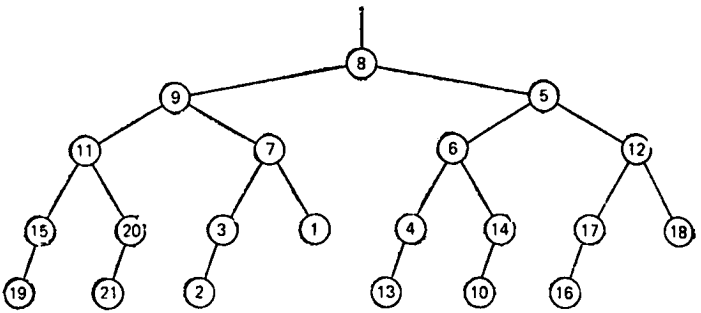


Рис. 4.23. Дерево, сформированное с помощью прогр. 4.3.

цедур. Очевидно, что рекурсивные алгоритмы особенно уместны в тех случаях, когда речь идет об обработке данных с рекурсивно определенной структурой. Это еще раз подтверждается на примере процедуры, печатающей полученное дерево. Пустое дерево не печатается, у поддерева уровня L для каждой вершины вначале печатается ее левое поддерево, затем сама вершина, выделенная отступом в L пробелов, и наконец, печатается ее правое поддерево.

4.4.2. Основные операции с двоичными деревьями

Существует много работ, которые можно выполнять с деревьями; распространенная задача — выполнение некоторой определенной операции P над каждым элементом дерева. Операцию P можно мыслить как параметр более общей задачи — «посещения» всех вершин или, как ее обычно называют, «обхода» дерева. Если эту задачу рассматривать как некий единый процесс, то отдельные вершины проходят в каком-то определенном порядке, и поэтому можно считать, что они линейно упорядочены. И действительно, описание многих алгоритмов значительно упрощается, если, ориентируясь на некоторый порядок, мы можем говорить об обработке следующего элемента дерева. Существуют три принципа упорядочения, естественно вытекающих из структуры деревьев. Как и структуру дерева, эти принципы удобнее всего выразить в тер-

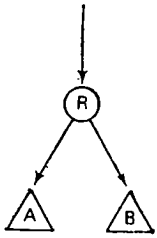


Рис. 4.24. Двоичное дерево.

шинах рекурсии. Если обратиться к двоичному дереву на рис. 4.24, где R обозначает корень, а A и B — левое и правое поддеревья, то можно говорить о таких трех порядках:

1. Сверху вниз: R, A, B (корень посещается ранее, чем поддеревья).
2. Слева направо: A, R, B.
3. Снизу вверх: A, B, R (корень посещается после поддеревьев). Обходя дерево с рис. 4.20 и выписывая символы, находящиеся в вершинах, мы получим такие три упорядоченные последовательности:

1. Сверху вниз: * + a/bc — d*ef.
2. Слева направо: a + b/c*d — e*f.
3. Снизу вверх: abc/ + def* — *.

Мы узнаем три формы записи выражений: обход *сверху вниз* дает *префиксную* запись, обход *снизу вверх — постфиксную* запись, а обход *слева направо* приводит к привычной *инфиксной* записи, хотя и без скобок, так необходимых для уточнения порядка выполнения операций.

Сформулируем теперь эти три метода обхода в виде трех конкретных программ с явным параметром t, соответствующим дереву, над каждой вершиной которого нужно выполнить операцию P — пусть это будет неявный параметр. Предположим, что есть такие определения

```
TYPE Ptr =  POINTER TO Node;
TYPE Node = RECORD ...                    (4.42)
              left, right: Ptr
            END
```

Теперь все три метода легко представить как рекурсивные процедуры; они вновь подтверждают то соображение, что работу с рекурсивно определенными данными удобнее всего определять с помощью рекурсий.

```
PROCEDURE preorder(t: Ptr)
BEGIN                                                    (4.43)
  IF t # NIL THEN
    P(t); preorder(t↑.left); preorder(t↑.right)
  END
END preorder

PROCEDURE inorder(t: Ptr);
BEGIN                                                    (4.44)
  IF t # NIL THEN
    inorder(t↑.left); P(t); inorder(t↑.right)
  END
END inorder

PROCEDURE postorder(t: Ptr);
BEGIN                                                    (4.45)
  IF t # NIL THEN
    postorder(t↑.left); postorder(t↑.right); P(t)
  END
END postorder
```

Обратите внимание: ссылка t передается как параметр-значение. Этим выражается тот факт, что речь идет о ссылке как величине, указывающей на нужное поддерево, а не о переменной, значение которой есть ссылка и которое могло бы меняться, если бы t была параметром-переменной.

Пример программы, в которой идет обход дерева, — подпрограмма печати дерева, выделяющая каждый уровень с помощью соответствующего отступа (прогр. 4.3).

Двоичные деревья часто употребляются для представления множества данных, среди которого идет поиск элементов по уникальному ключу. Если дерево организовано так, что для каждой вершины t_i справедливо утверждение, что все ключи левого поддерева t_l меньше ключа t_i, а все ключи правого поддерева t_r больше его, то такое дерево будем называть *деревом поиска*. В таком дереве можно обнаружить

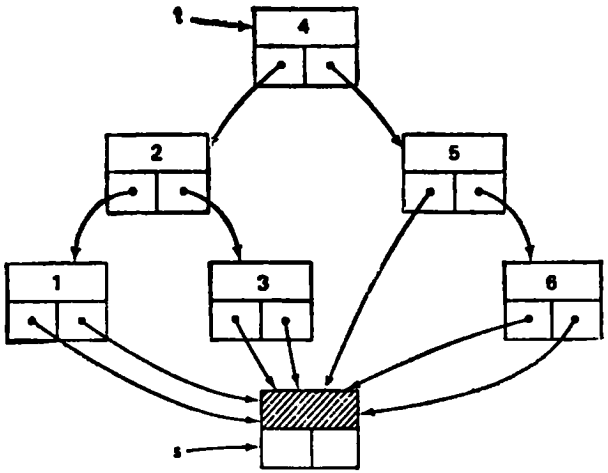


Рис. 4.25. Дерево поиска с барьером.

произвольный ключ — достаточно, начав с корня, двигаться к левому или правому поддереву на основании лишь одного сравнения с ключом текущей вершины. Мы уже знаем, что из n элементов можно организовать двоичное дерево с высотой не более log n. Поэтому, если дерево идеально сбалансировано, поиск среди его n элементов выполняется максимум за log n сравнений. Ясно, что подобные деревья значительно лучше подходят для организации таких множеств данных, чем линейные списки, о которых мы говорили в предыдущем разделе. Так как поиск идет по одному-единственному пути от корня к желанной вершине, то его можно просто программировать с помощью итерации (4.46).

```
PROCEDURE locate(x: INTEGER; t: Ptr): Ptr;
BEGIN                                                    (4.46)
  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
```

Если в дереве с корнем t не было обнаружено ключа со значением x, то функция *locate*(*x*, *t*) дает значение NIL. Как и в случае поиска в списке, сложность условия окончания наводит на мысль, что может существовать и лучшее решение, а именно использование барьера. Ведь такой прием можно применять и при поиске по дереву. Употребление ссылок позволяет закончить все ветви одним и тем же барьером. Теперь получается уже не дерево, а нечто напоминающее дерево, все листья которого «прицеплены» снизу к одному якорю (см. рис. 4.25) *). Барьер можно рассматривать как общее представление всех внешних вершин, которые добавляются к первоначальному дереву (см. рис. 4.19). В результате это позволяет получить более простую процедуру поиска (4.47):

```
PROCEDURE locate(x: INTEGER; t: Ptr): Ptr;
BEGIN s.key := x; (* барьер *)
  WHILE t.key # x DO                                    (4.47)
    IF t.key < x THEN t := t.right ELSE t := t.left END
  END;
  RETURN t
END locate
```

Обратите внимание, что если в дереве с корнем t ключ со значением x не будет обнаружен, то функция *locate*(*x*, *t*) получит не значение NIL, а s, т. е. будет указывать на барьер.

4.4.3. Поиск и включение для деревьев

Возможности динамического размещения переменных с доступом к ним через ссылки вряд ли полностью проявляются в таких примерах, где данные уже полностью сформированы и остаются неизменными. Более наглядными будут примеры, где меняется сама структура дерева, т. е. дерево растет или сокращается в ходе выполнения программы. Причем это должны быть задачи, где не подходят другие представления

*) Иногда такие структуры называют «гамаком». — *Прим. перев.*

данных, скажем массивы, а нужны именно деревья из элементов, связанных между собой ссылками.

Вначале рассмотрим случай, когда дерево постоянно только растет, но не убывает. Типичный пример — построение частотного словаря; о нем мы уже говорили, когда речь шла о связанных списках. Вернемся к нему снова. Задача состоит в том, что в заранее заданной последовательности надо определить частоту вхождения каждого из слов. Это означает, что любое слово надо искать в дереве, причем вначале дерево пустое. Если слово найдено, то счетчик его вхождений увеличивается, если нет — это новое слово включается в дерево с начальным, единичным значением счетчика. Мы будем называть эту задачу *поиском по дереву с включением.* Предположим, типы описаны таким образом:

```
TYPE WPtr = POINTER TO Word;
  Word =    RECORD                                    (4.48)
              key: INTEGER;
              count: CARDINAL;
              left, right: WPtr
            END
```

Считая, что у нас есть некоторый файл ключей и переменная, обозначающая корень дерева поиска, мы можем так сформулировать программу:

```
ReadInt(x);
WHILE Done DO search(x, root); ReadInt(x) END          (4.49)
```

Определение пути поиска здесь очевидно. Однако если он приводит в тупик (т. е. к пустому поддереву, обозначенному ссылкой со значением NIL), то данное слово необходимо включить в дерево на место пустого поддерева. Возьмем, например, двоичное дерево, приведенное на рис. 4.26, и включим в него имя *Paul.* Результат показан на том же рисунке пунктирными линиями.

Вся целиком операция приведена в прогр. 4.4. Процесс поиска оформлен как рекурсивная процедура. Ее параметр p передается как переменная, а не как значение. Это важно, поскольку в случае включения этой переменной, которая раньше хранила значение NIL,

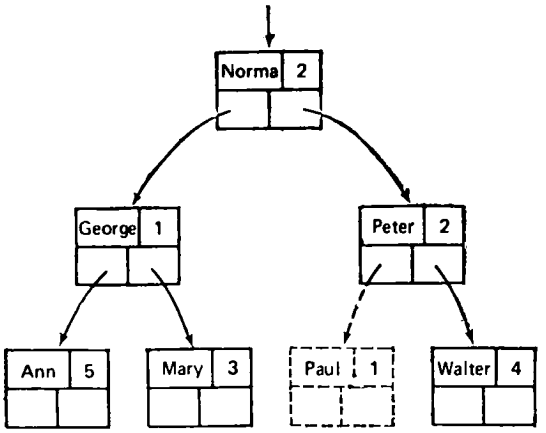


Рис. 4.26. Включение в упорядоченное двоичное дерево.

должно быть присвоено новое ссылочное значение. Если для заданной входной последовательности из 21 числа программа 4.3 строила дерево, приведенное на рис. 4.23, то новая программа (4.4) построит двоичное дерево поиска, приведенное на рис. 4.27.

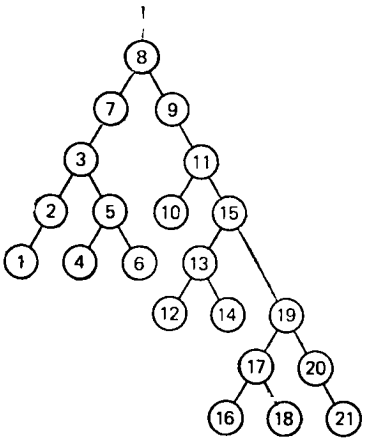


Рис. 4.27. Дерево поиска, сформированное с помощью прогр. 4.4.

```
MODULE TreeSearch;
  FROM InOut IMPORT OpenInput, CloseInput,
    ReadInt, Done, WriteInt, WriteString, WriteLn;
  FROM Storage IMPORT ALLOCATE;

  TYPE WPtr = POINTER TO Word;
    Word = RECORD key: INTEGER;
         count: INTEGER;
         left, right: WPtr
       END;

  VAR root: WPtr; n, key: INTEGER;
  PROCEDURE PrintTree(t: WPtr; h: INTEGER);
    VAR i: INTEGER;
  BEGIN (* печать дерева t с отступом h*)
    IF t # NIL THEN
      WITH t^ DO
        PrintTree(left, h+1);
        FOR i := 1 TO h DO WriteString("   ") END;
        WriteInt(key, 6); WriteLn;
        PrintTree(right, h+1)
      END
    END
  END PrintTree;

  PROCEDURE search(x: INTEGER; VAR p: WPtr);
  BEGIN
    IF p = NIL THEN (* слова в дереве нет, включение *)
      ALLOCATE(p, SIZE(Word));
      WITH p^ DO
        key := x; count := 1; left := NIL; right := NIL
      END
    ELSIF x < p^.key THEN search(x, p^.left)
    ELSIF x > p^.key THEN search(x, p^.right)
    ELSE p^.count := p^.count + 1
    END
  END search;

BEGIN root := NIL;
  (* первое целое – число вершин *)
  OpenInput("TEXT"); ReadInt(n);
  WHILE n > 0 DO
    ReadInt(key); search(key, root); n := n-1
  END;
  CloseInput; PrintTree(root, 0)
END TreeSearch.
```

Прогр. 4.4. Поиск по дереву с включением.

Использование барьера вновь несколько упрощает задачу, см. (4.50). Ясно, что перед началом работы переменная *root* должна быть инициирована ссылкой на барьер, а не значением NIL, и перед каждым поиском искомое значение x должно присваиваться полю ключа в барьере.

```
PROCEDURE search(x: INTEGER; VAR p: WPtr);
BEGIN
  IF x < p↑.key THEN search(x, p↑.left)
  ELSIF x > p↑.key THEN search(x, p↑.right)
  ELSIF p # s THEN p↑.count := p↑.count + 1
  ELSE (* включение *) Allocate(p, SIZE(Word));
    WITH p↑ DO                                          (4.50)
      key := x; left := s; right := s; count := 1
    END
  END
END
```

Хотя назначение этого алгоритма — поиск, его можно применять и для сортировки. Фактически он очень напоминает метод сортировки простым включением, но поскольку вместо массива используется дерево, то пропадает необходимость передвигать компоненты, находящиеся выше места включения. Сортировку с помощью дерева можно запрограммировать почти столь же эффективно, как и наилучшие из известных методов сортировки массивов. Но необходимо принять некоторые методы предосторожности. В случае совпадения так же нужно включать новый элемент. Если случай $x = p\uparrow.key$ обрабатывается, как и случай $x > p\uparrow.key$, то алгоритм представляет собой стабильный метод сортировки, т. е. элементы с идентичными ключами при обходе дерева выбираются в том же порядке, в котором они туда включались.

Вообще говоря, существуют и лучшие методы сортировки, но если речь идет о задачах, где требуется и поиск, и сортировка, то мы рекомендуем использовать метод поиска и включения по дереву. И он действительно очень часто применяется в трансляторах и банках данных для организации объектов, которые нужно и хранить в памяти, и искать. Подходящий

```
MODULE CrossRef;
  FROM InOut IMPORT OpenInput, OpenOutput, CloseInput,
    CloseOutput, Read, Done, EOL, Write, WriteCard, WriteLn;
  FROM Storage IMPORT ALLOCATE;

  CONST BufLeng = 10000; WordLeng = 16;

  TYPE WordPtr = POINTER TO Word;
    ItemPtr = POINTER TO Item;

    Word = RECORD key: CARDINAL;
          first, last: ItemPtr;
          left, right: WordPtr
        END;

    Item = RECORD lno: CARDINAL;
          next: ItemPtr
        END;

  VAR root: WordPtr;
    k0, k1, line: CARDINAL;
    ch: CHAR;
    buffer: ARRAY [0 .. BufLeng-1] OF CHAR;

  PROCEDURE PrintWord(k: CARDINAL);
    VAR lim: CARDINAL;
  BEGIN lim := k + WordLeng;
    WHILE buffer[k] > 0C DO Write(buffer[k]); k := k+1 END;
    WHILE k < lim DO Write(" "); k := k+1 END
  END PrintWord;

  PROCEDURE PrintTree(t: WordPtr);
    VAR i, m: INTEGER; item: ItemPtr;
  BEGIN
    IF t # NIL THEN
      WITH t↑ DO
        PrintTree(left);
        PrintWord(key); item := first; m := 0;
        REPEAT
          IF m = 8 THEN
            WriteLn; m := 0;
            FOR i := 1 TO WordLeng DO Write(" ") END
          END;
          m := m+1; WriteCard(item↑.lno, 6); item := item↑.next
```

```
    UNTIL item = NIL;
    WriteLn;
    PrintTree(right)
   END
  END
END PrintTree;

PROCEDURE Diff(i, j: CARDINAL): INTEGER;
BEGIN
  LOOP
   IF buffer[i] # buffer[j] THEN
    RETURN INTEGER(ORD(buffer[i])) - INTEGER(ORD(buffer[j]))
   ELSIF buffer[i] = 0C THEN RETURN 0
   END;
   i := i+1; j := j+1
  END
END Diff;

PROCEDURE search(VAR p: WordPtr);
  VAR item: ItemPtr; d: INTEGER;
BEGIN
  IF p = NIL THEN (* слова в дереве нет, включение *)
   ALLOCATE(p, SIZE(Word)); ALLOCATE(item, SIZE(Item));
   WITH p↑ DO
    key := k0; first := item; last := item;
    left := NIL; right := NIL
   END;
   item↑.lno := line; item↑.next := NIL; k0 := k1
  ELSE d := Diff(k0, p↑.key);
   IF d < 0 THEN search(p↑.left)
   ELSIF d > 0 THEN search(p↑.right)
   ELSE ALLOCATE(item, SIZE(Item));
    item↑.lno := line; item↑.next := NIL;
    p↑.last↑.next := item; p↑.last := item
   END
  END
END search;

PROCEDURE GetWord;
BEGIN k1 := k0;
  REPEAT Write(ch); buffer[k1] := ch; k1 := k1 + 1; Read(ch)
  UNTIL (ch < "0") OR (ch > "9") & (CAP(ch) < "A")
```

```
    OR (CAP(ch) > "Z");
  buffer[k1] := 0C; k1 := k1 + 1; (* ограничитель *)
  search(root)
 END GetWord;

BEGIN root := NIL; k0 := 0; line := 0;
 OpenInput("TEXT"); OpenOutput("XREF");
 WriteCard(0, 6); Write(" "); Read(ch);
 WHILE Done DO
  CASE ch OF
   0C .. 35C:  Read(ch) |
   36C .. 37C: WriteLn; Read(ch); line := line + 1;
               WriteCard(line, 6); Write(" ") |
   " " .. "@": Write(ch); Read(ch) |
   "A" .. "Z": GetWord |
   "[" .. "`": Write(ch); Read(ch) |
   "a" .. "z": GetWord |
   "{" .. "~": Write(ch); Read(ch)
  END
 END;
 WriteLn; WriteLn; CloseInput;
 PrintTree(root); CloseOutput
END CrossRef.
```

Прогр. 4.5. Построение таблицы перекрестных ссылок.

пример — построение таблицы перекрестных ссылок для данного примера. Этот пример мы уже использовали как иллюстрацию при формировании списков.

Наша задача — написать программу, которая (читая текст и печатая его с добавлением последовательных номеров строк) собирает все слова этого текста, запоминая номера строк, в которых встречалось данное слово. После того как просмотр будет окончен, формируется таблица, в которой все слова будут расположены в алфавитном порядке и со списками их местонахождений.

Очевидно, для хранения слов, встречающихся в тексте, лучше всего подходит дерево поиска (называемое также лексикографическим деревом). Каждая вершина не только содержит в качестве ключа само слово, но и является началом списка номеров строк. Каждую запись о появлении слова мы будем называть *item* (отметка). Таким образом, в этом примере

встречаются и деревья, и линейные списки. Программа состоит из двух основных частей (см. прогр. 4.5), а именно: фазы просмотра текста и фазы печати таблицы. Последняя представляет собой просто программу обхода дерева, причем при попадании в каждую вершину печатается ключевое значение (слово) и просматривается связанный с ней список номеров строк (отметки). Кроме того, полезно привести еще некоторые пояснения, относящиеся к прогр. 4.5.

Таблица 4.3. Результаты работы программы 4.5

```
 0 PROCEDURE search(x: INTEGER; VAR p: WPtr);
 1 BEGIN
 2    IF x < pt.key THEN search(x, pt.left)
 3    ELSIF x > pt.key THEN search(x, pt.right)
 4    ELSIF p # s THEN pt.count := pt.count + 1
 5    ELSE Allocate(p, SIZE(Word));
 6         WITH pt DO
 7            key := x; left := s; right := s; count := 1
 8         END
 9    END
10 END
```

Allocate	5						
BEGIN	1						
DO	6						
ELSE	5						
ELSIF	3	4					
END	8	9	10				
IF	2						
INTEGER	0						
PROCEDURE	0						
SIZE	5						
THEN	2	3	4				
VAR	0						
WITH	6						
WPtr	0						
Word	5						
count	4	4	7				
key	2	3	7				
left	2	7					
p	0	2	2	3	3	4	4
4							
	5	6					
right	3	7					
s	4	7	7				
search	0	2	3				
x	0	2	2	3	3	7	

(В табл. 4.3 приведены результаты обработки программой 4.5 текста (4.50).)

1. Словом считается любая последовательность букв и цифр, начинающаяся с буквы.

2. Так как размер слов может быть разным, сами фактические символы хранятся в массиве *buffer*, а вершина дерева содержит индекс первого символа ключа.

3. Желательно, чтобы номера строк в таблице перекрестных ссылок печатались в возрастающем порядке. Поэтому список отметок должен формироваться в том же порядке, в котором он будет просматриваться при печати. Это требование приводит к использованию в каждой вершине, соответствующей слову, двух ссылок: одна указывает на первый, а вторая — на последний элемент списка отметок.

4.4.4. Исключение из деревьев

Перейдем теперь к задаче, обратной включению, к *исключению* из деревьев. Наша цель — определить алгоритм исключения — изъятия из упорядоченного дерева вершины с ключом x. К сожалению, исключение элемента обычно проходит не так просто, как включение. Простым оно оказывается лишь в случае, если исключаемый элемент — терминальная вершина или вершина с одним потомком. Трудность возникает, если нужно удалить элемент с двумя потомками; ведь мы не можем указать с помощью одной ссылки сразу два направления. В этом случае удаляемый элемент нужно заменить либо на самый правый элемент его левого поддерева, либо на самый левый элемент его правого поддерева, причем они должны иметь самое большее одного потомка. Все детали приводятся в самой рекурсивной процедуре под названием *delete* (4.52). В ней различаются три случая:

1. Компоненты с ключом, равным x, нет.

2. Компонента с ключом x имеет не более одного потомка.

3. Компонента с ключом x имеет двух потомков.

```
PROCEDURE delete(x: INTEGER; VAR p: Ptr);
  VAR q: Ptr;                                          (4.52)

  PROCEDURE del (VAR r: Ptr);
  BEGIN
    IF r↑.right # NIL THEN del(r↑.right)
    ELSE q↑.key := r↑.key; q↑.count := r↑.count;
        q := r; r := r↑.left
    END
  END del;

BEGIN (* исключение *)
  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 (*исключение  p↑*) q := p;
    IF q↑.right = NIL THEN p := q↑.left
    ELSIF q↑.left = NIL THEN p := q↑.right
    ELSE del(q↑.left)
    END;
    (*Deallocate(q)*)
  END
END delete
```

Вспомогательная рекурсивная процедура *del* начинает работать только в случае 3. Она «спускается» вдоль правой ветви левого поддерева элемента q↑, который нужно исключить, и заменяет существенную информацию (ключ и счетчик) в q↑ на соответствующие значения из самой правой компоненты r↑ левого поддерева, после чего от t↑ можно освободиться. Неописанную процедуру *Deallocate* можно рассматривать как обратную по отношению к процедуре *Allocate*. Если последняя выделяет память для новой компоненты, то первую можно применять для указания вычислительной системе, что память, занятая элементом q↑, вновь свободна и доступна для любых других целей.

Для иллюстрации работы процедуры (4.52) мы отсылаем читателя к рис. 4.28. Здесь приведено исходное дерево (a), из которого последовательно удаляются вершины с ключами 13, 15, 5 и 10. Полученные при этом деревья показаны там же: (b)—(e).

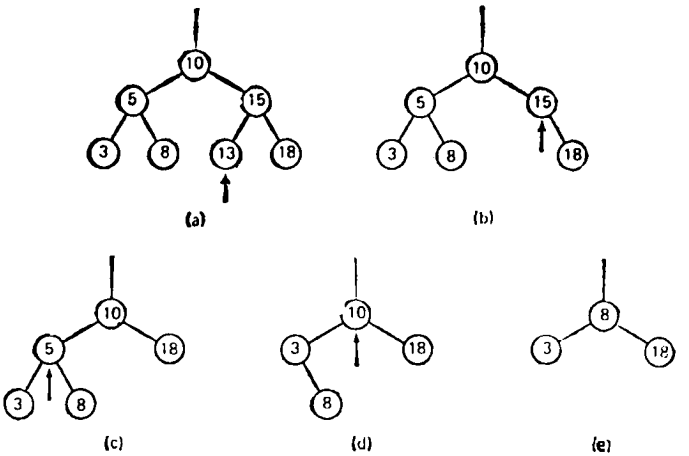


Рис. 4.28. Исключение из дерева.

4.4.5. Анализ поиска по дереву с включениями

Довольно естественно, что к алгоритму поиска с включением для деревьев испытываешь некоторое недоверие. Во всяком случае, сомнения будут оставаться до тех пор, пока мы более детально не познакомимся с его поведением. Ведь многих программистов в первую очередь беспокоит тот факт, что обычно неизвестно, как будет расти дерево и какую форму оно примет. Мы можем только предполагать, что скорее всего дерево не будет идеально сбалансированным. Поскольку среднее число сравнений, необходимое для обнаружения ключа, в идеально сбалансированном дереве с n вершинами приблизительно равно log n, то число сравнений в дереве, сформированном нашим алгоритмом, будет больше. Но насколько?

Прежде всего просто находится наихудший случай. Предположим, все поступающие ключи идут в строго возрастающем (или убывающем) порядке. Тогда каждый ключ присоединяется непосредственно справа (или слева) от его предшественника, и в результате мы получаем полностью вырожденное дерево, вытянутое в линейный список. В этом случае средние

затраты на поиск равны n/2 сравнениям. Ясно, что такой наихудший вариант приводит к очень плохой производительности, и кажется, что наш скептицизм полностью оправдывается. Правда, остается вопрос, насколько вероятен такой случай. Точнее, хотелось бы знать длину пути поиска a_n, усредненную для всех n ключей и для всех n! деревьев, полученных в результате n! перестановок из этих n исходных ключей. Такая задача анализа алгоритма оказывается достаточно простой и приводится здесь не только как типичный пример такого анализа, но и из-за практической важности получающегося результата.

Пусть даны n различных ключей со значениями 1, 2, ..., n, причем поступают они в случайном порядке. Вероятность того, что первый ключ, который становится корневым узлом, имеет значение i, равна 1/n. Его левое поддерево в конце концов будет состоять из i — 1 вершин, а правое — из n — i вершин (см. рис. 4.29). Обозначим среднюю длину пути в левом поддереве через a_{i-1}, а в правом поддереве — через a_{n-i}. Будем вновь предполагать, что все возможные перестановки оставшихся n — 1 ключей равновероятны. Средняя длина пути в дереве с n вершинами равна сумме произведений уровня каждой вершины на вероятность обращения к ней. Если все вершины ищутся с одинаковой вероятностью, то получаем

$$a_n = (Si : 1 \leqslant i \leqslant n : p_i)/n, \tag{4.53}$$

где p_i — длина пути к вершине i.

Вершины на рис. 4.29 можно разделить на три класса:

1. i — 1 вершин в левом поддереве имеют среднюю длину пути a_{i-1};

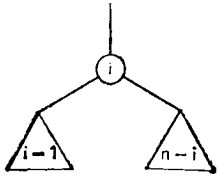


Рис. 4.29. Распределение весов по ветвям.

2. корень имеет длину пути 0;

3. n — i вершин в правом поддереве имеют среднюю длину пути a_{n-i}.

Следовательно, выражение (4.53) можно представить как сумму двух произведений

$$a_n^{(i)} = ((i-1) * a_{i-1} + (n-i) * a_{n-i})/n. \qquad (4.54)$$

Искомое значение a_n — среднее значение $a_n^{(i)}$ для всех i = 1...n, т. е. для всех деревьев с ключами в корне, равными 1, 2, ..., n:

$$\begin{aligned} a_n &= (Si: 1 \leqslant i \leqslant n: (i-1) * a_{i-1} + (n-i) * a_{n-i})/n^2 \qquad (4.55)\\ &= 2 * (Si: 1 \leqslant i \leqslant n: (i-1) * a_{i-1})/n^2\\ &= 2 * (Si: 1 \leqslant i < n: i * a_i)/n^2 \end{aligned}$$

Уравнение (4.55) представляет собой рекуррентное соотношение вида $a_n = f_1(a_1, a_2, \ldots, a_n)$. Из него мы получаем более простое рекуррентное соотношение вида $a_n = f_2(a_{n-1})$. Вначале получаем (1), раскладывая последнее произведение, и (2), подставляя n — 1 вместо n:

(1) $a_n = 2 * (n-1) * a_{n-1}/n^2 + 2 * (Si: 1 \leqslant i < n: i * a_i)/n^2$

(2) $a_{n-1} = 2 * (Si: 1 \leqslant i < n-1: i * a_i)/(n-1)^2$

Умножая (2) на $(n-1)^3/n^2$, получаем

(3) $2 * (Si: 1 \leqslant i < n-1: i * a_i)/n^2 = a_{n-1} * (n-1)^2/n^2$

и, подставив правую часть (3) в (1), найдем

$$\begin{aligned} a_n &= 2 * (n-1) * a_{n-1}/n^2 + a_{n-1} * (n-1)^2/n^2 \qquad (4.56)\\ &= a_{n-1} * (n-1)^2/n^2 \end{aligned}$$

Оказывается, a_n можно представить в нерекурсивном замкнутом виде с помощью гармонической функции

$$\begin{aligned} H_n &= 1 + 1/2 + 1/3 + \ldots + 1/n\\ a_n &= 2 * (H_n * (n+1)/n - 1) \end{aligned} \qquad (4.57)$$

Из формулы Эйлера (используя константу g = = 0,577...)

$$H_n = g + \ln n + 1/12n^2 + \ldots$$

мы получаем для больших n соотношение $a_n \doteq 2*(\ln n + g - 1)$. Так как средняя длина пути в идеально сбалансированном дереве приблизительно равна

$$a'_n \doteq \log n - 1 \qquad (4.58)$$

то, опуская слагаемые, которые при больших n становятся достаточно малыми, мы получаем

$$\lim \left(a_n/a'_n\right) = 2 * \ln (n)/\log (n) = 2 * \ln (2) \doteq 1.386\ldots \qquad (4.59)$$

Что же означает этот результат анализа (4.59)? Из него следует, что, стараясь всегда строить идеально сбалансированное дерево вместо случайного, получаемого с помощью прогр. 4.4, мы можем ожидать средний выигрыш в длине пути поиска не более 30 %. При этом мы по-прежнему предполагаем, что все ключи распределены равномерно. Подчеркиваем, что речь идет о средней длине пути, ибо в наиболее неблагоприятных случаях, когда дерево вырождается в список, выигрыш будет значительно больше. Однако такое случается крайне редко. Следует заметить, что ожидаемая средняя длина пути в случайном дереве с ростом числа вершин растет тоже строго логарифмически, хотя в худшем случае и наблюдается линейный рост.

Цифра 39 % накладывает ограничение на объем дополнительных ухищрений для перестройки структуры дерева по мере поступления ключей. Разумеется, при этом имеет существенное значение и отношение частоты обращений (поисков) к вершинам к частоте включений (коррекций дерева). Чем больше это отношение, тем больше выигрыш от процедуры перестройки. Однако цифра 39 % достаточно низка, и в большинстве приложений улучшение простого алгоритма поиска с включением не оправдывается, если, конечно, речь не идет о большем числе вершин и большем отношении доступ/включение.

4.5. СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

Из предыдущих рассуждений ясно, что процедура включения, восстанавливающая идеально сбалансированное дерево, вряд ли будет всегда выгодна, поскольку восстановление дерева после случайного

включения — довольно сложная операция. Возможно, выходом из положения будет введение менее строгого определения сбалансированности. Такое не совсем совершенное определение могло бы привести к более простой процедуре переупорядочения за счет лишь незначительного усложнения поиска. Одно из таких определений сбалансированного дерева было предложено Г. М. Адельсон-Вельским и Е. М. Ландисом [4.1]. Их критерий сбалансированности сформулирован так:

Дерево называется *сбалансированным* тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на единицу.

Деревья, удовлетворяющие такому условию, часто называют АВЛ-деревьями (по имени их открывателей). Мы их будем просто называть *сбалансированными деревьями*, поскольку упомянутый критерий сбалансированности выглядит наиболее подходящим. (Заметим, что все идеально сбалансированные деревья являются также и АВЛ-деревьями.)

Введенное определение не только само очень простое, но и приводит к простой процедуре повторной балансировки, причем средняя длина пути поиска практически совпадает с длиной в идеально сбалансированном дереве.

В сбалансированных деревьях за время, пропорциональное $O(\log n)$ даже в худшем случае, можно выполнить такие операции:

1. Найти вершину с данным ключом.
2. Включить новую вершину с заданным ключом.
3. Исключить вершину с указанным ключом.

Такие утверждения — следствие доказанной Адельсон-Вельским и Ландисом теоремы, гарантирующей, что сбалансированное дерево никогда не будет по высоте превышать идеально сбалансированное более чем на 45 % независимо от количества вершин. Если обозначить высоту сбалансированного дерева с n вершинами через $h_b(n)$, то

$$\log(n+1) \leqslant h_b(n) < 1.4404 * \log(n+2) - 0.328 \qquad (4.60)$$

Оптимум достигается, очевидно, если дерево идеально сбалансировано, при $n = 2^k - 1$. Однако какова

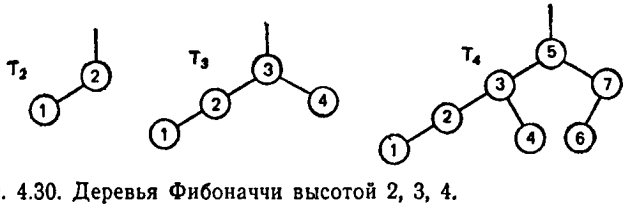


Рис. 4.30. Деревья Фибоначчи высотой 2, 3, 4.

структура *самого плохого* сбалансированного АВЛ-дерева? Чтобы найти максимальную высоту h для всех сбалансированных деревьев с n вершинами, поступим следующим образом: возьмем фиксированную высоту h и попытаемся построить сбалансированное дерево с минимальным числом вершин. Мы следуем такой стратегии, поскольку, как и в случае минимальной высоты, это значение может быть достигнуто только при некоторых вполне определенных значениях. Обозначим такое дерево высоты h через T_h. Ясно, что T_0 — пустое дерево, а T_1 — дерево с одной-единственной вершиной. Для построения T_h для $h > 1$ мы будем брать корень и два поддерева опять же с минимальным числом вершин. Следовательно, эти деревья также относятся к классу T-деревьев. Одно поддерево, очевидно, должно быть высотой $h-1$, а другому позволяется иметь высоту на единицу меньше, т. е. $h-2$. На рис. 4.30 представлены деревья высотой 2, 3 и 4. Поскольку принцип их построения очень напоминает построение чисел Фибоначчи, то мы будем называть такие деревья *деревьями Фибоначчи*. Они определяются следующим образом:

1. Пустое дерево есть дерево Фибоначчи высоты 0.
2. Единственная вершина есть дерево Фибоначчи высоты 1.
3. Если T_{h-1} и T_{h-2} — деревья Фибоначчи высотой $h-1$ и $h-2$, то $T_h = \langle T_{h-1}, x, T_{h-2}\rangle$ также дерево Фибоначчи высотой h.
4. Других деревьев Фибоначчи не существует.

Число вершин в T_h определяется из такого простого рекуррентного отношения:

$$N_0 = 0,\ N_1 = 1$$
$$N_h = N_{h-1} + 1 + N_{h-2} \qquad (4.60)$$

Числа N_l как раз и будут числом вершин для самых худших случаев (верхний предел h), их называют числами Леонарда.

4.5.1. Включение в сбалансированное дерево

Рассмотрим теперь, что может произойти при включении в сбалансированное дерево новой вершины. Если у нас есть корень r и левое (L) и правое (R) поддеревья, то необходимо различать три возможных случая. Предположим, включение в L новой вершины приведет к увеличению на 1 его высоты; тогда возможны 3 случая:

1. $h_L = h_R$: L и R станут разной высоты, но критерий сбалансированности не будет нарушен.

2. $h_L < h_R$: L и R станут равной высоты, т. е. сбалансированность даже улучшится.

3. $h_L > h_R$: критерий сбалансированности нарушится и дерево необходимо перестраивать.

Возьмем дерево, представленное на рис. 4.31. Вершины с ключами 9 и 11 можно включить, не нарушая сбалансированности дерева; дерево с корнем 10 становится односторонним (случай 1), а с корнем 8 — лишь лучше сбалансированным (случай 2). Однако включение ключей 1, 3, 5 или 7 требует последующей балансировки.

При внимательном изучении этой ситуации обнаруживается, что существуют лишь две по существу различные возможности, требующие индивидуального подхода. Оставшиеся могут быть выведены из этих двух на основе симметрии. Первый случай возникает при включении в дерево на рис. 4.31 ключей 1 или 3,

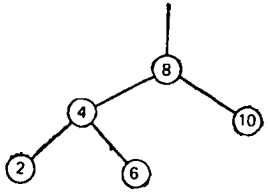


Рис. 4.31. Сбалансированное дерево.

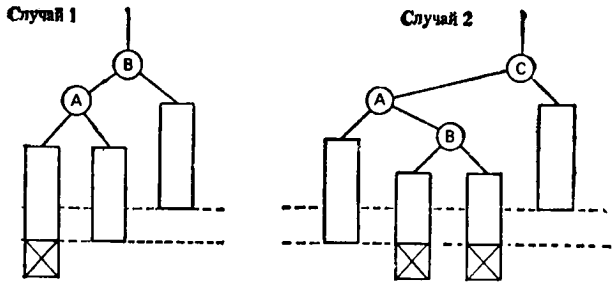


Рис. 4.32. Несбалансированность, возникшая из-за включения.

ситуация, характерная для второго случая, возникает при включении ключей 5 или 7.

Схематично эти случаи представлены на рис. 4.32, где прямоугольниками обозначены поддеревья, причем «добавленная» при включении высота отмечена перечеркиванием. Простые преобразования сразу же восстанавливают желанную сбалансированность. Их результат приведен на рис. 4.33. Обратите внимание, что допускаются лишь перемещения в вертикальном направлении, в то время как относительное горизонтальное расположение показанных вершин и поддеревьев должно оставаться без изменения.

Алгоритм включения и балансировки существенно зависит от того, каким образом хранится информация о сбалансированности дерева. Крайнее решение — хранить эту информацию полностью неявно, в структуре

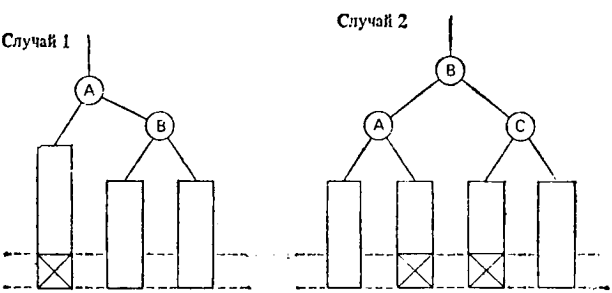


Рис. 4.33. Восстановление сбалансированности.

самого дерева. Однако в этом случае сбалансированность узла нужно определять всякий раз, когда включение затрагивает этот узел. Это ведет к очень большим затратам. Другая крайность — хранить в каждой вершине показатель сбалансированности. В этом случае определение типа Node расширяется до такого:

```
TYPE Ptr =     POINTER TO Node;
TYPE Balance = [-1 .. +1];
TYPE Node =    RECORD key: INTEGER;                    (4.62)
                  count: INTEGER;
                  left, right: Ptr
                  bal: Balance
               END
```

Показатель сбалансированности вершины мы в дальнейшем будем интерпретировать как разность между высотой правого поддерева и левого, а сам алгоритм будет основан на описании типа (4.62). Процесс включения вершины фактически состоит из следующих трех последовательно выполняемых частей:

1. Прохода по пути поиска, пока не убедимся, что ключа в дереве нет.

2. Включения новой вершины и определения результирующего показателя сбалансированности.

3. «Отступления» по пути поиска и проверки в каждой вершине показателя сбалансированности. Если необходимо — балансировка.

Хотя при таком методе и требуются некоторые избыточные проверки (если сбалансированность уже зафиксирована, то нет необходимости проверять ее в вышестоящих вершинах), мы будем вначале придерживаться этой, очевидно корректной схемы, так как ее можно реализовать простым расширением уже готовой процедуры поиска с включением из прогр. 4.4. В этой процедуре описываются действия, связанные с поиском в каждой вершине, и в силу рекурсивной природы самой процедуры ее легко приспособить для выполнения дополнительных действий при возвращении вдоль пути поиска. Информация, которую нужно передавать на каждом шаге, указывает, увеличилась или нет высота поддерева, где

произошло включение. Поэтому мы расширим список параметров процедуры и добавим булевское значение h, означающее — «высота дерева увеличилась». Поскольку через него передается результат, то это должен быть параметр-переменная.

Предположим теперь, что процесс из левой ветви возвращается к вершине p↑ (см. рис. 4.32), причем указывается, что ее высота увеличилась. В зависимости от высоты поддеревьев перед включением мы должны различать три возможные ситуации:

1. $h_L < h_R$, p↑.bal = +1, предыдущая несбалансированность в p уравновешивается.

2. $h_L = h_R$, p↑.bal = 0, левое поддерево стало «перевешивать».

3. $h_L > h_R$, p↑.bal = −1, необходима балансировка.

В третьей ситуации по показателю сбалансированности корня левого поддерева (т. е. по p1↑.bal) можно определить, имеем ли мы дело со случаем 1 или 2 (рис. 4.32). Если левое поддерево этой вершины также выше правого, то мы имеем дело со случаем 1, иначе — со случаем 2. (Убедитесь сами, что в этом случае не может существовать левого поддерева с показателем сбалансированности в его корне, равным нулю.) Операция по балансировке состоит только из последовательных переприсваиваний ссылок. Фактически ссылки циклически меняются местами, что приводит к одно- или двукратному повороту двух или трех участвующих в процессе вершин. Кроме вращения ссылок нужно должным образом поправить и показатели сбалансированности соответствующих вершин. Детальное описание процесса приводится в самой процедуре поиска, включения и балансировки (4.63).

Принцип работы алгоритма иллюстрирует рис. 4.34. Рассмотрим двоичное дерево, содержащее всего две вершины (а). Включение ключа 7 вначале приводит к несбалансированному дереву (просто к линейному списку). Его балансировка включает единственный RR-поворот и приводит к идеально сбалансированному дереву (в). Последующие включения вершин 2 и 1 приводят к несбалансированному поддереву

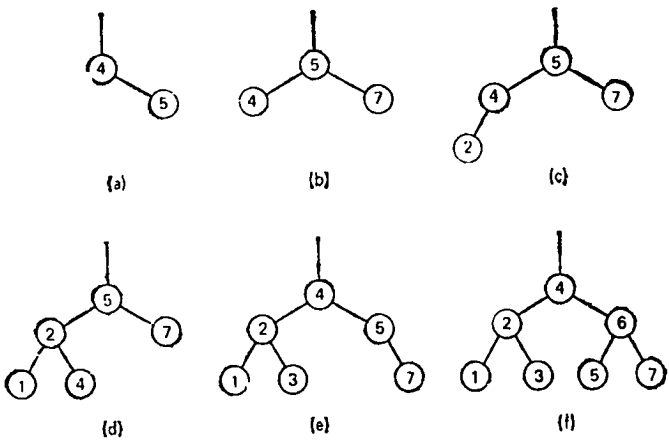


Рис. 4.34. Включение в сбалансированное дерево.

с корнем 4. С помощью одного LL-поворота оно балансируется (d). Последующее включение ключа 3 сразу же нарушает критерий сбалансированности в корневой вершине 5. После чего балансировка достигается уже с помощью более сложного двойного LR-поворота; результатом является дерево (e). При любом следующем включении баланс может быть нарушен лишь в вершине 5. И действительно, включение вершины с ключом 6 приводит к четвертому типу балансировки, выделенному в (4.63), — двойному RL-повороту. Заключительное дерево показано на рис. 4.34 (f).

В связи с производительностью алгоритма включения в сбалансированные деревья возникают, в частности, два таких интересных вопроса:

1. Если равновероятно появление любой из n! перестановок из n ключей, то какова ожидаемая высота формируемого дерева?

2. Какова вероятность того, что включение повлечет за собой балансировку?

Математический анализ нашего сложного алгоритма пока остается открытой проблемой. Экспериментальные проверки показывают, что ожидаемая высота сбалансированного дерева, построенного с

```
PROCEDURE search(x: INTEGER; VAR p: Ptr; VAR h: BOOLEAN);
 VAR p1, p2: Ptr; (*~h*)
BEGIN                                                          (4.63)
 IF p = NIL THEN (* включение *)
  ALLOCATE(p, SIZE(Node)); h := TRUE;
  WITH p↑ DO
   key := x; count := 1; left := NIL; right := NIL; bal := 0
  END
 ELSIF p↑.key > x THEN
  search(x, p↑.left, h);
  IF h THEN (*выросла левая ветвь *)
    CASE p↑.bal OF
     1: p↑.bal := 0; h := FALSE |
     0: p↑.bal := -1 |
    -1: (* балансировка *) p1 := p↑.left;
      IF p1↑.bal = -1 THEN (* однократный LL-поворот *)
       p↑.left := p1↑.right; p1↑.right := p;
       p↑.bal := 0; p := p1
      ELSE (* двойной LR-поворот *) 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 *)
   CASE p↑.bal OF
    -1: p↑.bal := 0; h := FALSE |
     0: p↑.bal := 1 |
     1: (* балансировка *) p1 := p↑.right;
      IF p1↑.bal = 1 THEN (* однократный RR-поворот *)
       p↑.right := p1↑.left; p1↑.left := p;
       p↑.bal := 0; p := p1
      ELSE (* двойной RL-поворот *) 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 p↑.count := p↑.count + 1
 END
END search
```

помощью алгоритма (4.63), такова: $h = \log(n) + c$, где c — небольшая константа ($c = 0{,}25$). Практически это означает, что сбалансированные АВЛ-деревья ведут себя так же, как и идеально сбалансированные, хотя поддерживать их намного проще. Те же экспериментальные данные подтверждают, что в среднем приблизительно на два включения приходится одна балансировка. При этом одинаково вероятны и однократные, и двукратные повороты. Конечно, ясно, что пример на рис. 4.34 подбирался специально, чтобы на минимальном числе включений продемонстрировать как можно больше поворотов.

Сложность операции балансировки предполагает, что сбалансированные деревья следует использовать только тогда, когда поиск информации происходит значительно чаще, чем ее включение. Это утверждение особенно верно, поскольку вершины дерева поиска для экономии памяти обычно хранятся в виде плотно упакованных записей. Поэтому решающим фактором, определяющим эффективность операции балансировки, часто бывают доступность показателя балансировки и простота его изменения, ведь для его представления нужно всего два разряда. Опыт показывает, что сбалансированные деревья сразу же теряют почти всю свою привлекательность, как только речь заходит о плотной упаковке для записей. И действительно, простой и очевидный алгоритм включения в дерево превзойти трудно!

4.5.2. Исключение из сбалансированного дерева

Наш опыт по удалению вершин из дерева позволяет предположить, что и исключение из сбаланси-

рованного дерева окажется более сложным, чем включение. И это действительно так, хотя алгоритм операции балансировки остается фактически тем же самым, что и при включении. В частности, балансировка опять основывается на одно- или двукратных поворотах узлов.

Принципиальная схема алгоритма исключения из сбалансированного дерева аналогична (4.52). Простые случаи — удаление терминальных вершин или вершин только с одним потомком. Если же от исключаемой вершины «отходят» два поддерева, то, как и раньше, она заменяется на самую правую вершину ее левого поддерева. Как и в случае включения (4.63), вводится булевский параметр-переменная h, указывающий, *уменьшилась ли высота поддерева*. Балансировка идет, только если h — истина. Это значение присваивается переменной h при обнаружении и исключении какой-либо из вершин или уменьшении высоты какого-либо поддерева в процессе самой балансировки. В программе (4.64) в виде процедур мы вводим две (симметричные) операции балансировки, поскольку обращение к ним встречается более чем в одной точке алгоритма исключения. Отметим, что *balanceL* используется при уменьшении высоты левого поддерева, а *balanceR* — правого.

Рис. 4.35 иллюстрирует работу нашей процедуры. Из исходного сбалансированного дерева (a) последовательно исключаются вершины с ключами 4,8,6,5,2,1 и 7 и получаются деревья (b)...(h). Исключение ключа 4 выглядит просто, поскольку он представляет собой терминальную вершину. Однако в результате получается несбалансированная вершина 3. Соответствующая операция балансировки требует единственного LL-поворота. При исключении вершины 6 вновь требуется балансировка. В этот момент одиночным RR-поворотом балансируется правое поддерево вершины 7. Исключение вершины 2, хотя само по себе оно простое, поскольку вершина имеет лишь одного потомка, приводит к усложненному двукратному RL-повороту. И четвертый случай: двукратный LR-поворот проводится при изъятии вершины 7, вместо кото-

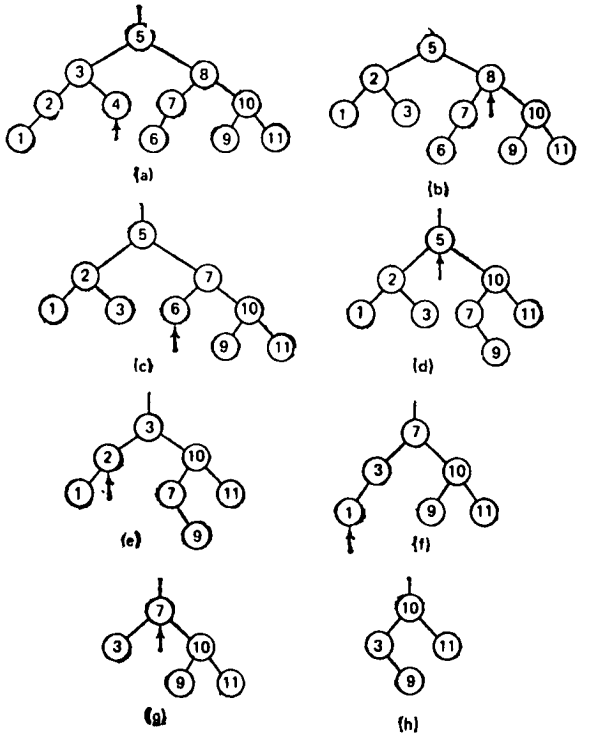


Рис. 4.35. Исключение из сбалансированного дерева.

рой вначале помещался самый правый элемент ее левого поддерева, т. е. вершина с ключом 3. (См. с. 284, 285.)

К счастью, исключение любого элемента из любого сбалансированного дерева можно в самом худшем случае провести за $O(\log n)$ операций. Однако не следует упускать из виду существенное различие в поведении процедур включения и исключения. В то время как включение одного ключа может привести самое большое к одному повороту (двух или трех вершин), исключение может потребовать поворотов во всех вершинах вдоль пути поиска. Рассмотрим, например, исключение из дерева Фибоначчи самой

(4.64)

```
PROCEDURE balanceL(VAR p: Ptr; VAR h: BOOLEAN);
 VAR p1, p2: Ptr; b1, b2: Balance;
BEGIN (*h; левая ветвь стала короче *)
 CASE p↑.bal OF
  -1: p↑.bal := 0 |
   0: p↑.bal := 1; h := FALSE |
   1: (* балансировка *) p1 := p↑.right; b1 := p1↑.bal;
     IF b1 >= 0 THEN (* однократный RR-поворот *)
       p↑.right := p1↑.left; p1↑.left := p;
       IF b1 = 0 THEN p↑.bal := 1; p1↑.bal := -1; h := FALSE
        ELSE p↑.bal := 0; p1↑.bal := 0
       END;
       p := p1
     ELSE (* двойной RL-поворот *)
       p2 := p1↑.left; b2 := p2↑.bal;
       p1↑.left := p2↑.right; p2↑.right := p1;
       p↑.right := p2↑.left; p2↑.left := p;
       IF b2 = +1 THEN p↑.bal := -1 ELSE p↑.bal := 0 END;
       IF b2 = -1 THEN p1↑.bal := 1 ELSE p1↑.bal := 0 END;
       p := p2; p2↑.bal := 0
     END
 END
END balanceL;

PROCEDURE balanceR(VAR p: Ptr; VAR h: BOOLEAN);
 VAR p1, p2: Ptr; b1, b2: Balance;
BEGIN (*h; правая ветвь стала короче *)
 CASE p↑.bal OF
  1: p↑.bal := 0 |
  0: p↑.bal := -1; h := FALSE |
 -1: (* балансировка *) p1 := p↑.left; b1 := p1↑.bal;
     IF b1 <= 0 THEN (* однократный LL-поворот *)
       p↑.left := p1↑.right; p1↑.right := p;
       IF b1 = 0 THEN p↑.bal := -1; p1↑.bal := 1; h := FALSE
        ELSE p↑.bal := 0; p1↑.bal := 0
       END;
       p := p1
     ELSE (* двойной LR-поворот *)
       p2 := p1↑.right; b2 := p2↑.bal;
       p1↑.right := p2↑.left; p2↑.left := p1;
       p↑.left := p2↑.right; p2↑.right := p;
```

```
    IF b2 = -1 THEN p↑.bal := 1 ELSE p↑.bal := 0 END;
    IF b2 = +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: Ptr; VAR h: BOOLEAN);
 VAR q: Ptr;
 PROCEDURE del(VAR r: Ptr; 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 в дереве нет *)
 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 (* исключение 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;
  (* Deallocate(q) *)
 END
END delete
```

правой вершины. В этом случае исключение любой одной вершины приводит к уменьшению высоты дерева, и кроме того, исключение самой правой вершины требует максимального числа поворотов. Таким образом, мы имеем дело с самой неудачной вершиной и самым неудачным случаем сбалансированного дерева — достаточно редкая комбинация. А вообще, какова вероятность поворотов? Приблизительно на каждые два включения встречается один поворот,

а при исключении поворот происходит даже в одном случае из пяти. Поэтому исключение из сбалансированного дерева почти столь же просто или столь же сложно, как и включение.

4.6. ДЕРЕВЬЯ ОПТИМАЛЬНОГО ПОИСКА

Во всех наших рассуждениях по поводу организации деревьев поиска мы до сих пор исходили из предположения, что частота обращения ко всем вершинам одинакова, т. е. все ключи с равной вероятностью фигурируют как аргументы поиска. Если нет никаких других соображений по поводу распределения, то такое предположение о равномерном распределении, пожалуй, самое разумное. Однако встречаются ситуации (они, скорее всего, исключение из правил), когда есть информация о вероятностях обращения к отдельным ключам. Обычно для таких ситуаций характерно «постоянство» ключей, т. е. в дерево поиска не включаются новые ключи и не исключаются старые, структура дерева остается неизменной. Вот типичный пример — сканер транслятора, определяющий, не относится ли каждое слово (идентификатор) к классу ключевых (зарезервированных) слов. Статистические измерения на сотнях транслируемых программ могут в этом случае дать точную информацию об относительных частотах появления отдельных ключей (т. е. обращения к ним).

Предположим, что в дереве поиска вероятность обращения к вершине i такова:

$$\Pr\{x = k_i\} = p_i, \quad (Si : 1 \leqslant i \leqslant n : p_i) = 1 \qquad (4.65)$$

Теперь мы хотим так организовать дерево поиска, чтобы общее число шагов поиска, подсчитанное для достаточно большого числа обращений, было минимальным. С этой целью (1) припишем в определении длины пути (4.34) каждой вершине некоторый вес (2) и условимся, что корень находится на уровне 1 (а не 0), поскольку он первым встречается на пути поиска. Тогда (внутренняя) *взвешенная длина пути* представляет собой сумму всех путей от корня к

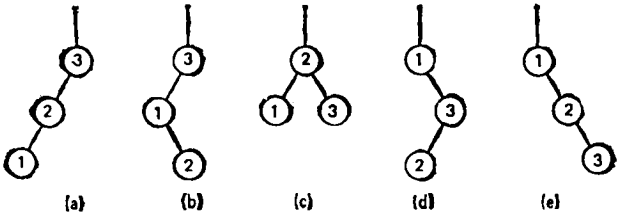


Рис. 4.36. Дерево поиска с тремя вершинами.

каждой из вершин, умноженных на вероятность обращения к этой вершине:

$$p = Si : 1 \leqslant i \leqslant n : p_i * h_i, \tag{4.66}$$

где h_i — уровень вершины i. Наша цель — минимизировать при заданном распределении вероятностей взвешенную длину пути. В качестве примера рассмотрим множество из трех ключей 1,2,3 со следующими вероятностями обращений к ним: $p_1 = 1/7$, $p_2 = 2/7$ и $p_3 = 4/7$. Эти три ключа можно расставить в дереве поиска пятью различными способами (см. рис. 4.36). Взвешенные длины путей деревьев (a)—(e), вычисленные по формуле (4.66), таковы:

$$P(a) = 11/7, \quad P(b) = 12/7, \quad P(c) = 12/7, \quad P(d) = 15/7,$$
$$P(e) = 17/7.$$

Таким образом, оптимальным оказывается не идеально сбалансированное дерево (c), а вырожденное — (a).

Упомянутый сканер транслятора сразу наводит на мысль, что задачу следует ставить при несколько более общих условиях: ведь слова, встречающиеся в исходном тексте, не всегда бывают ключевыми словами, на самом деле эта ситуация скорее исключение, чем правило. Обнаружение, что данный ключ не есть ключ дерева поиска, можно считать обращением к некоторой гипотетической «специальной вершине», включенной между следующей меньшей и следующей большей вершинами (см. рис. 4.19) с соответствующей длиной внешнего пути. Если известна вероятность q_i того, что аргумент поиска x находится между двумя ключами k_i и k_{i+1}, то эта информация

может существенно трансформировать структуру дерева оптимального поиска. Поэтому мы обобщим задачу и будем учитывать возможность неудачного поиска. Общая средняя взвешенная длина пути в этом случае имеет вид

$$P = (Si : 1 \leqslant i \leqslant n : p_i * h_i) + (Sj : 0 \leqslant j \leqslant m : q_j * h'_j), \quad (4.67)$$

где

$$(Si : 1 \leqslant i \leqslant n : p_i) + (Sj : 0 \leqslant j \leqslant m : q_j) = 1,$$

причем h_i — уровень внутренней вершины i, h'_j — уровень внешней вершины j. Среднюю взвешенную длину пути можно назвать *ценой* дерева поиска, поскольку она представляет собой некоторую меру для ожидаемых затрат на поиск. Дерево поиска, имеющее минимальную для всех деревьев с заданным множеством ключей k_i и вероятностями p_i и q_i цену, называется *оптимальным* деревом.

При построении оптимального дерева мы не будем требовать, чтобы сумма всех p и q равнялась 1. Ведь обычно эти вероятности определяются экспериментально, подсчетом обращений к вершинам. Поэтому вместо вероятностей p_i и q_i мы далее будем пользоваться просто «счетчиками» частоты обращений. Обозначим их следующим образом:

a_i = число поисков с аргументом x, равным k_i,

b_j = число поисков с аргументом x, лежащим между k_j и k_{j+1}.

Условимся, что b_0 — число поисков с аргументом x, меньшим k_1, а b_n — число поисков с x, большим чем k_n (см. рис. 4.37). Далее, вместо средней длины пути мы будем обозначать через P *общую взвешенную длину пути*:

$$P = (Si : 1 \leqslant i \leqslant n : a_i * h_i) + (Sj : 0 \leqslant j \leqslant m : b_j * h'_j). \quad (4.68)$$

Таким образом, мало того, что мы избегаем вычисления вероятностей по измеренным частотам, мы еще и выигрываем от использования в наших построениях оптимального дерева целых чисел.

Учитывая, что число возможных конфигураций из n вершин растет экспоненциально с ростом n, задача построения оптимального дерева при больших n ка-

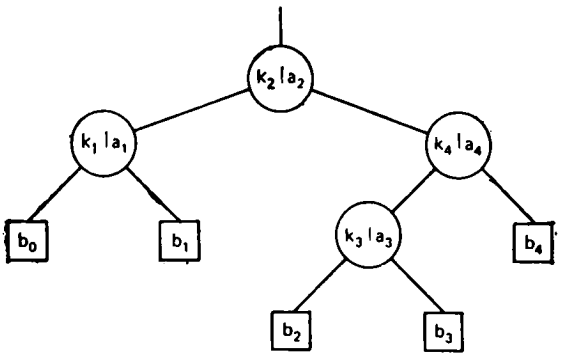


Рис. 4.37. Дерево поиска с указанием частот обращения.

жется совершенно безнадежной. Однако оптимальные деревья обладают одним важным свойством, которое помогает их обнаруживать: все их поддеревья тоже оптимальны. Если, например, дерево на рис. 4.37 оптимально, то поддерево с ключами k_3 и k_4, как показано, также оптимально. Такая особенность предполагает алгоритм, который систематически находит все бóльшие и бóльшие деревья, начиная с отдельных вершин как наименьших возможных поддеревьев. Таким образом, дерево растет «от листьев к корню», «снизу вверх», если учесть, что мы рисуем деревья сверху вниз [4.6].

В основе нашего уравнения лежит уравнение (4.69). Пусть P — взвешенная длина пути всего дерева, а P_L и P_R — соответствующие длины для левого и правого поддеревьев его корня. Ясно, что P — сумма P_L и P_R и числа случаев W, когда поиск проходит по единственному пути к корню, т. е. W — просто общее число поисков. Назовем W *весом* дерева. Тогда средняя длина пути будет P/W.

$$P = P_L + W + P_R \tag{4.69}$$

$$W = (Si : 1 \leqslant i \leqslant n : a_i) + (Sj : 0 \leqslant j \leqslant m : b_j) \tag{4.70}$$

Из этих рассуждений видно, что необходимо ввести обозначения для веса и длины пути любого из поддеревьев, включающего «соседние» ключи. Пусть

T_{ij} — оптимальное поддерево, состоящее из вершин с ключами k_{i+1}, k_{i+2}, ..., k_j. Тогда обозначим его вес через w_{ij}, а длину пути — через p_{ij}. Ясно, что $P = p_{0,n}$, а $W = w_{0,n}$. Эти величины определяются рекуррентными соотношениями (4.71) и (4.72).

$$w_{ii} = b_i \qquad (0 \leqslant i \leqslant n) \tag{4.71}$$

$$w_{ij} = w_{i,j-1} + a_j + b_j \qquad (0 \leqslant i < j \leqslant n)$$

$$p_{ii} = w_{ii} \qquad (0 \leqslant i \leqslant n) \tag{4.72}$$

$$p_{ij} = w_{ij} + \mathbf{MIN}\, k : i < k \leqslant j : (p_{i,k-1} + p_{kj}) \qquad (0 \leqslant i < j \leqslant n)$$

Последнее равенство следует непосредственно из (4.69) и определения оптимальности. Поскольку существует около $n^2/2$ значений p_{ij}, а (4.72) требует выбора одного из $0 < j - i \leqslant n$ значений, то весь процесс минимизации будет занимать приблизительно $n^3/6$ операций. Кнут отмечает, что можно избавиться от одного множителя n и тем самым сохранить практическую ценность данного алгоритма. Делается это так.

Пусть r_{ij} — значение k, при котором в (4.72) достигается минимум. Поиск r_{ij} можно ограничить значительно меньшим интервалом, т. е. сократить число вычислений до $j - i$. Это можно сделать, поскольку если мы нашли корень r_{ij} оптимального поддерева T_{ij}, то ни расширение при добавлении к дереву справа новой вершины, ни сжатие при отбрасывании самого левого узла не могут сдвинуть вправо этот оптимальный корень. Такое свойство выражается отношением

$$r_{i,j-1} \leqslant r_{ij} \leqslant r_{i+1,j} \tag{4.73}$$

которое и ограничивает поиск возможных решений для r_{ij} диапазоном $r_{i,j-1}$ ... $r_{i+1,j}$. Это в конце концов и ограничивает число элементарных шагов — около n^2.

Теперь мы уже готовы составить детальный алгоритм оптимизации. Введем следующие определения, исходя из того, что T_{ij} — оптимальные деревья, содержащие узлы с ключами k_{i+1} ... k_j:

1. a_i — частота поиска k_i.
2. b_j — частота поиска аргумента x, лежащего между k_j и k_{j+1}.
3. w_{ij} — вес T_{ij}.

4. p_{ij} — взвешенная длина пути T_{ij}.
5. r_{ij} — индекс корня T_{ij}.

После описания TYPE index = [0 .. n] вводим следующие массивы:

```
a: ARRAY [1..n] OF CARDINAL;
b: ARRAY index OF CARDINAL;
p,w:ARRAY index, index OF CARDINAL,                    (4.74)
r: ARRAY index, index OF index
```

Предположим, что вес w_{ij} уже вычислен непосредственно по a и b (см. (4.71)). Будем считать w аргументом процедуры *OptTree*, которую нужно создать. Значение же r будет ее результатом, ведь этот массив полностью описывает структуру дерева. Причем p можно рассматривать как некоторый промежуточный результат. Начав с наименьших возможных поддеревьев, т. е. деревьев, не содержащих никаких вершин, мы переходим все к большим и большим деревьям. Обозначим ширину j — i поддерева T_{ij} через h. После этого мы можем просто определить по (4.72) значения p_{ii} для всех деревьев с h = 0.

```
FOR i := 0 TO n DO p[i,i] := b[i] END                  (4.75)
```

В случае h = 1 мы имеем дело с деревьями, содержащими одну-единственную вершину, которая к тому же будет и корнем (см. рис. 4.38).

```
FOR I := 0 TO n-1 DO
  i := i+1; p[i,j] := w[i,j] + p[i,i] + p[j,j]; r[i,j] := j   (4.76)
END
```

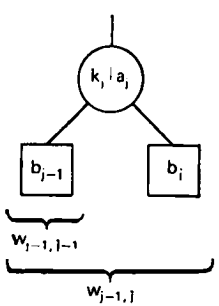


Рис. 4.38. Оптимальное дерево с одной вершиной.

Заметим, что в рассматриваемом дереве T_{ij} через i обозначена левая граница индекса, а через j — правая. Для вариантов $h > 1$ мы используем оператор цикла с параметром h, пробегающим значения от 2 до n, причем при $h = n$ перекрывается все дерево $T_{0,n}$. В каждом случае минимальная длина пути p_{ij} и соответствующий индекс корня r_{ij} определяются с помощью простого оператора цикла с индексом k, изменяющимся в интервале, заданном в (4.73).

```
FOR h := 2 TO n DO
  FOR i := 0 TO n-h DO                                    (4.77)
    j := i+h;
    найти k и min = MIN k: i < k ≤ j : (p[i,k-1] + p[k,j])
    такие, что r[i,j-1] ≤ k ≤ r[i+1,j];
    p[i,j] := min + w[i,j]; r[i,j] := k
  END
END
```

В программе 4.6 приводится детальное описание операторов, выделенных выше курсивом. Средняя длина пути в $T_{0,n}$ здесь задается частным $p_{0,n}/w_{0,n}$, а его корень — вершина с индексом $r_{0,n}$.

Теперь опишем структуру прогр. 4.6. Она состоит из двух основных компонент: процедуры для построения при заданном распределении весов w дерева оптимального поиска и процедуры печати дерева, определенного индексами r. Вначале из входного потока читаются значения a и b и ключи. В вычислении структуры дерева эти ключи не употребляются, они нужны лишь для последующей печати полученного дерева. После того как будет напечатана информация о частотах (статистика), программа начинает вычислять длину пути идеально сбалансированного дерева, определяя по пути корни его поддеревьев. После всего печатается средняя взвешенная длина пути и изображение дерева.

В третьей части вызывается процедура *OptTree*, она вычисляет дерево оптимального поиска, затем печатается изображение полученного дерева. И наконец, те же самые процедуры используются для определения и печати оптимального дерева, учитывающего лишь частоты употребления служебных слов, другие слова игнорируются.

```
MODULE OptTree;
 FROM InOut IMPORT
  OpenInput, OpenOutput, Read, ReadCard, ReadString, WriteCard,
  WriteString, Write, WriteLn, Done, CloseInput, CloseOutput;
 FROM Storage IMPORT ALLOCATE;

 CONST N = 100; (*max число слов*)
    WL = 16; (*max длина слова *)

 TYPE Word = ARRAY [0 .. WL-1] OF CHAR;
   index = [0 .. N];

 VAR ch: CHAR;
  i, j, n: CARDINAL;
  key: ARRAY index OF Word;
  a:  ARRAY index OF CARDINAL;
  b:  ARRAY index OF CARDINAL;
  p,w: ARRAY index, index OF CARDINAL;
  r:  ARRAY index, index OF CARDINAL;

 PROCEDURE BalTree(i,j: CARDINAL): CARDINAL;
  VAR k: CARDINAL;
 BEGIN k := (i+j+1) DIV 2; r[i,j] := k;
  IF i >= j THEN RETURN 0
    ELSE RETURN BalTree(i,k-1) + BalTree(k,j) + w[i,j]
  END
 END BalTree;

 PROCEDURE OptTree;
  VAR x, min: CARDINAL;
   i, j, k, h, m: CARDINAL;
 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
     x := p[i,k-1] + p[k,j];
     IF x < min THEN
      m := k; min := x
```

```
      END
     END;
     p[i,j] := min + w[i,j]; r[i,j] := m
    END
   END
 END OptTree;

 PROCEDURE PrintTree(i, j, level: CARDINAL);
  VAR k: CARDINAL;
 BEGIN
  IF i < j THEN
   PrintTree(i, r[i,j]-1, level+1);
   FOR k := 1 TO level DO WriteString("    ") END;
   WriteString(key[r[i,j]]); WriteLn;
   PrintTree(r[i,j], j, level+1)
  END
 END PrintTree;

BEGIN (* главная программа *)
 n := 0; OpenInput("TEXT");
 LOOP ReadCard(b[n]);
  IF NOT Done THEN HALT END;
  ReadCard(j);
  IF NOT Done THEN EXIT END;
  n := n+1; a[n] := j;
  ReadString(key[n])
 END;

 OpenOutput("TREE");
 (* вычисление w по 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;
 WriteString("Total weight = "); WriteCard(w[0,n], 6); WriteLn;

 WriteString("Pathlength of balanced tree = ");
 WriteCard(BalTree(0, n), 6); WriteLn;
 PrintTree(0, n, 0); WriteLn;

 Read(ch);
```

```
OptTree;
WriteString("Pathlength of optimal tree = ");
WriteCard(p[0,n], 6); WriteLn;
PrintTree(0, n, 0); WriteLn;

Read(ch);
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;
OptTree;
WriteString("optimal tree not considering b"); WriteLn;
PrintTree(0, n, 0); WriteLn;
CloseInput; CloseOutput
END OptTree
```

Прогр. 4.6. Построение дерева оптимального поиска.

В таблице 4.4 и на рис. 4.40—4.42 приведены результаты применения прогр. 4.6 к самой себе, т. е. к ее собственному тексту. Различия в этих трех рисунках наглядно демонстрируют, что сбалансированное дерево нельзя считать даже приближением к оптимальному дереву, а учет частот «других» (не служебных) слов сильно влияет на выбор оптимальной структуры.

Из алгоритма (4.77) ясно, что затраты на определение оптимальной структуры пропорциональны n^2, размер необходимой памяти также порядка n^2. Если n слишком велико, то такой подход уже неприемлем. Поэтому желательно иметь дело с более эффектив-

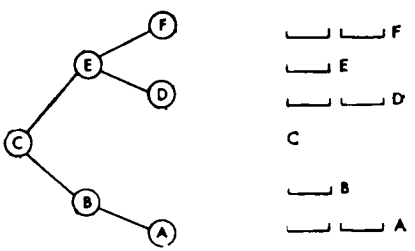


Рис. 4.39. Изображение дерева с помощью надлежащих отступов

Таблица 4.4. Служебные слова и частоты их появления

b[i-1]	a[i]	k[i]
169	3	AND
25	37	ARRAY
355	125	BEGIN
87	1	BY
264	14	CASE
247	28	CODE
90	9	CONST
118	3	DEFINITION
10	16	DIV
0	55	DO
124	299	ELSE
4	198	ELSIF
10	689	END
281	25	EXIT
35	3	EXPORT
442	19	FROM
0	0	FOR
646	464	IF
5	3	IMPLEMENTATION
13	20	IMPORT
15	2	IN
654	24	LOOP
159	15	MOD
130	16	MODULE
166	79	NIL
16	10	NOT
218	34	OF
31	95	OR
276	11	POINTER
82	171	PROCEDURE
418	1	QUALIFIED
124	6	RECORD
49	9	REPEAT
30	2	RETURN
174	22	SET
505	662	THEN
9	6	TO
385	13	TYPE
37	9	UNTIL
347	203	VAR
84	35	WHILE
0	14	WITH
981		

ными алгоритмами. Один из таких алгоритмов — алгоритм, созданный Ху и Таккером [4.5], он требует памяти порядка $O(n)$, а затраты на вычисления — порядка $O(n*\log(n))$. Однако он рассчитан лишь на случай нулевых частот для ключевых слов, т. е. в нем фиксируются только неудачные попытки отыскать слово. Еще один алгоритм был описан Уолкером и Готлибом [4.7], для него нужна память порядка $O(n)$, а затраты на вычисления — порядка $O(n*\log(n))$. В этом алгоритме ищется не оптимальное дерево, а дерево, близкое к оптимальному, и он построен на эвристических принципах. Основная идея следующая.

Будем считать, что вершины (подлинные и специальные) распределены по линейной шкале с весами, соответствующими частоте (или вероятности) обращения к ним. Найдем вершину, расположенную ближе всего к «центру тяжести». Такую вершину будем называть *центроидом*. Его индекс равен округленному до ближайшего целого значения выражению

$$((Si: 1 \leqslant i \leqslant n: i * a_i) + (Sj: 0 \leqslant j \leqslant m: j * b_j))/W \qquad (4.78)$$

Total weight = 11265
Pathlength of balanced tree = 60312

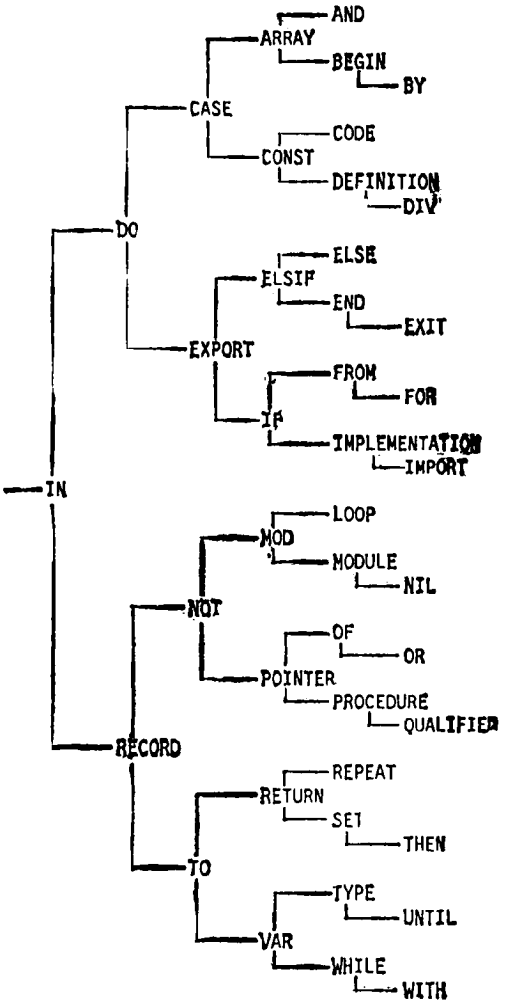


Рис. 4.40. Идеально сбалансированное дерево.

Pathlength of optimal tree = 50371

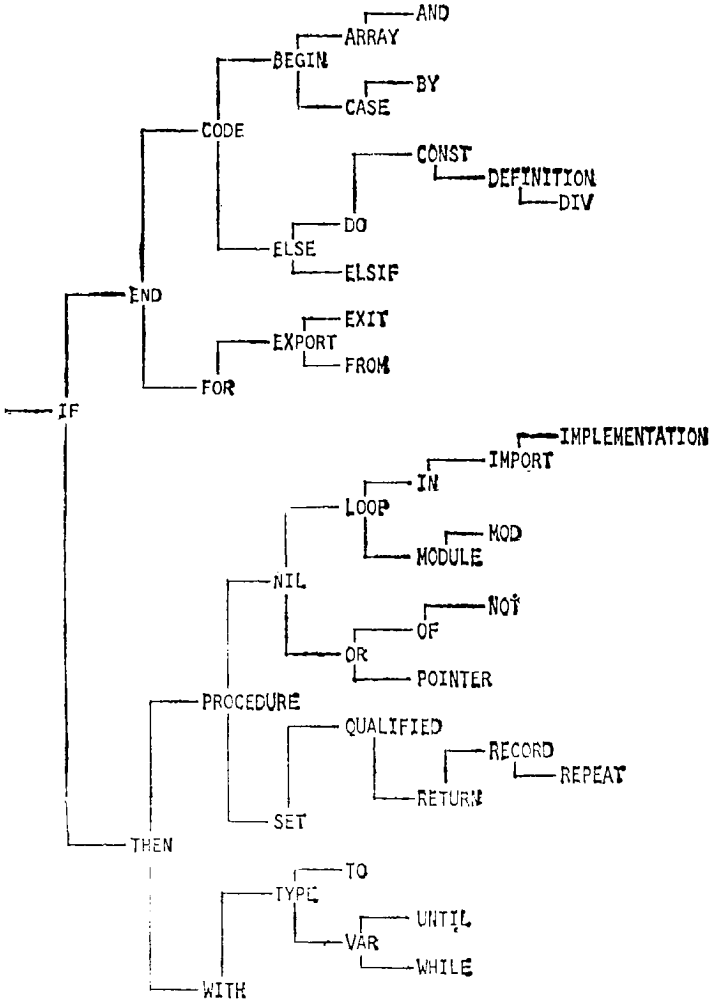


Рис. 4.11. Оптимальное дерево поиска.

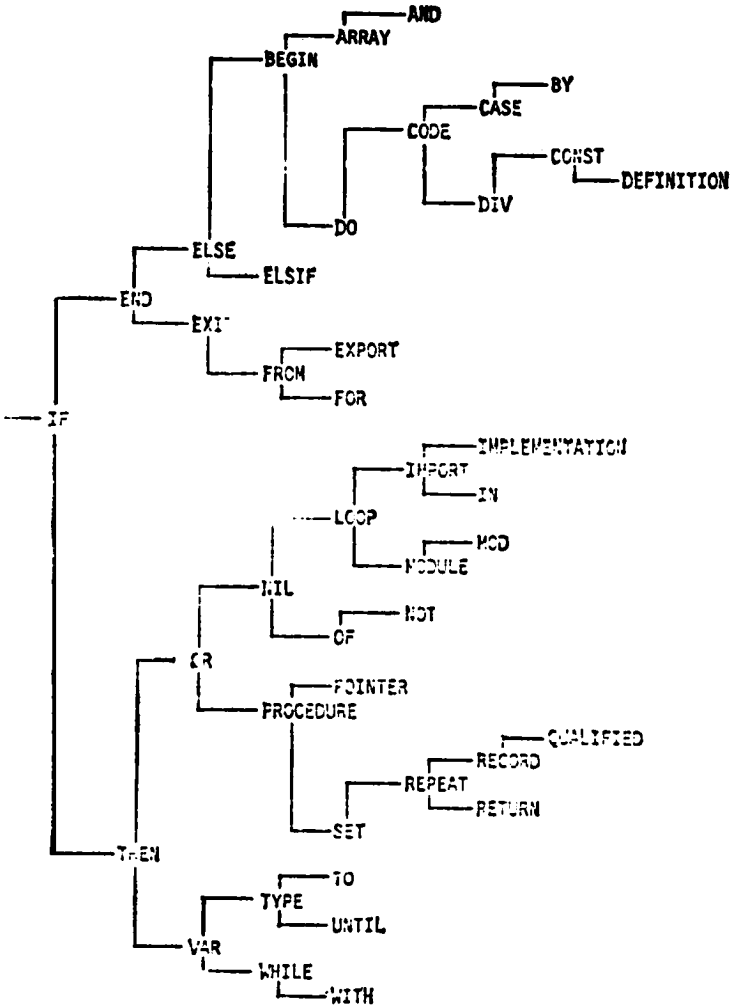


Рис. 4.42. Оптимальное дерево, учитывающее только служебные слова.

Ясно, что если вершины имеют равный вес, то корень искомого оптимального дерева совпадает с центроидом, и вообще, в большинстве случаев он будет располагаться в окрестности центроида. Затем на ограниченном интервале отыскивается локальный оптимум, а потом эта процедура вновь применяется к двум полученным поддеревьям. С увеличением размера дерева n увеличивается и вероятность того, что корень дерева находится очень близко от центроида. А как только размер дерева станет «обозримым», его оптимальное строение можно уже определить с помощью приведенного точного алгоритма.

4.7. Б-ДЕРЕВЬЯ

До сих пор мы в наших обсуждениях ограничивали себя деревьями, у каждой вершины которых было самое большее два потомка, т. е. двоичными (или бинарными) деревьями. Этого было вполне достаточно, например, для представления родственных отношений, если исходить из «родословных» представлений, когда каждый человек ассоциируется со своими родителями. В конце концов у каждого из нас не более двух родителей. А если нужно «потомственное» или «наследственное» представление? Ведь в этом случае приходится учитывать, что некоторые могут иметь более двух детей, и, следовательно, соответствующие деревья — вершины с многими ветвями. За неимением лучшего термина мы будем называть такие деревья *сильно ветвящимися*.

Конечно, в таких структурах нет ничего особенного, и мы уже встречали и в программировании, и в описании данных механизмы, позволяющие справиться с подобными ситуациями. Если, скажем, задана абсолютная верхняя граница числа детей (несколько футорологическое предположение), то детей в записи, представляющей любого человека, можно мыслить как компоненты некоторого массива. Если же число детей у разных людей сильно варьируется, то такое представление может привести к плохому использованию доступной памяти. В этом случае значительно лучше представлять потомство в виде линей-

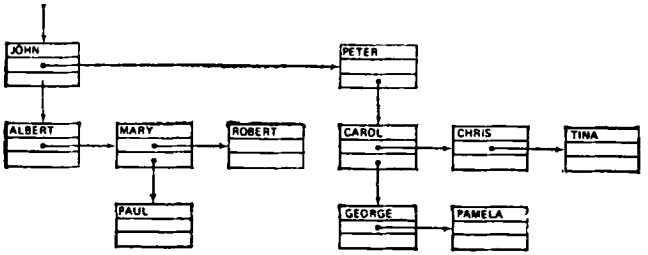


Рис. 4.43. Сильно ветвящееся дерево, представленное в виде двоичного.

ного списка со ссылками от родителя на самого младшего (или старшего) из детей. В такой ситуации можно воспользоваться такими описаниями типов:

```
TYPE Ptr = POINTER TO Person;
TYPE Person = RECORD name: alfa;                    (4.80)
                sibling, offspring: Ptr
              END
```

На рис. 4.43 приведен пример данных соответствующей структуры. Можно заметить, что если повернуть на этом рисунке ссылки на 45°, то получим нечто вроде совершенного двоичного дерева. Однако считать его таковым было бы ошибкой, поскольку функционально эти две ссылки имеют совершенно другой смысл. Обычно нельзя обращаться с братом, как с сыном *), и поэтому так не следует поступать даже при описании данных. Этот пример можно легко продолжить и, введя в запись, описывающую человека, еще другие компоненты, получить данные более сложной структуры, отражающие другие родственные отношения. Например, из отношений «сын» и «брат» нельзя вывести отношения класса «муж» и «жена» и даже обратного отношения «отец» и «мать». Подобные структуры быстро разрастаются в сложный реляционный банк данных, в котором можно выделить

*) В оригинале речь идет об отношениях sibling (брат или сестра) и offspring (сын или дочь). К сожалению, в русском языке такие отношения не выражаются одним словом. — *Прим. перев.*

несколько деревьев. Алгоритмы, работающие с такими структурами, существенно зависят от их описаний, поэтому не имеет смысла определять для них какие-либо общие правила или приемы работы.

Существует, однако, одна весьма практическая область применения сильно ветвящихся деревьев, представляющая общий интерес. Это — формирование и поддержание крупномасштабных деревьев поиска, в которых необходимы и включение новых элементов, и удаление старых, но для которых либо не хватает оперативной памяти, либо она слишком дорога, чтобы использовать ее для долговременного хранения.

В этом случае будем считать, что вершины дерева должны храниться во вторичной памяти, скажем на диске. Вводимые в данной главе динамические структуры данных особенно подходят именно для такой вторичной памяти. Принципиальное новшество — ссылки представляют собой адреса на диске, а не в оперативной памяти. Если использовать для множества данных, включающих, например, миллион элементов, двоичное дерево, то потребуется в среднем приблизительно порядка $\log(10^6)$, т. е. 20 шагов поиска. Поскольку теперь каждый шаг включает обращение к диску (с его собственным латентным временем), то крайне желательна организация памяти, требующая меньшего числа обращений. Сильно ветвящиеся деревья представляют собой идеальное решение этой проблемы. Если происходит обращение к одному элементу, расположенному во вторичной памяти, то без больших дополнительных затрат можно обратиться и к целой группе элементов. Это предполагает, что дерево разбито на поддеревья и эти поддеревья как раз те группы, которые одновременно доступны. Мы будем называть такие поддеревья страницами. На рис. 4.44 приведено разбитое на страницы двоичное дерево; каждая из страниц содержит 7 вершин.

Поскольку каждое обращение к странице теперь требует лишь одного обращения к диску, то экономия на числе таких обращений может быть существенной. Представим себе, что каждая страница содержит 100 вершин (это вполне разумное число), в этом слу-

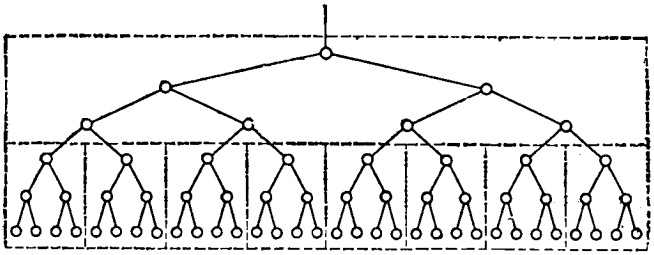


Рис. 4.44. Двоичное дерево, разделенное на страницы.

чае поиск в дереве с миллионом элементов будет в среднем требовать $\log_{100}(10^6)$ обращений к страницам (т. е. всего около 3), а не 20. Конечно, если дерево растет случайным образом, то в худшем случае может потребоваться даже и 10^4 обращений. Поэтому ясно, что для сильно ветвящихся деревьев почти обязательна некоторая схема управления их ростом.

4.7.1. Сильно ветвящиеся Б-деревья

Если речь заходит о критерии управления ростом дерева, то сразу же можно отказаться от идеальной сбалансированности, поскольку она требует слишком больших затрат на балансировку. Очевидно, правила надо как-то смягчить. Очень разумный критерий был сформулирован в 1970 г. Р. Бэйером и Е. Маккрейтом [4.2]: каждая страница (кроме одной) должна содержать при заданном постоянном n от n до 2n вершин. Таким образом, для дерева с N элементами и максимальным размером страницы в 2n вершин в самом худшем случае потребуется $\log_n N$ обращений к страницам, а ведь эти обращения составляют основную часть затрат на поиск. Кроме того, коэффициент использования памяти, а это достаточно важно, равен 50 %, поскольку страницы всегда заполнены минимум наполовину. При всех этих достоинствах наша схема ориентирована и на сравнительно простые алгоритмы поиска, включения и исключения. В дальнейшем мы их изучим более детально.

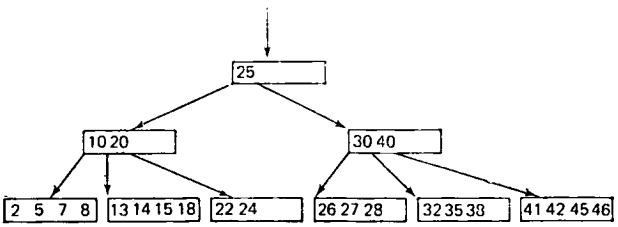


Рис. 4.45. Б-дерево порядка 2.

Структуры данных, о которых идет речь, называются *Б-деревьями*, а n — *порядком* Б-дерева. Они обладают следующими свойствами:

1. Каждая страница содержит не более 2n элементов (ключей).

2. Каждая страница кроме корневой содержит не менее n элементов.

3. Каждая страница либо представляет собой лист, т. е. не имеет потомков, либо имеет $m + 1$ потомков, где m — число ключей на этой странице.

4. Все страницы-листья находятся на одном уровне.

На рис. 4.45 приведено Б-дерево второго порядка с тремя уровнями. Все страницы содержат 2, 3 или 4 элемента, исключение представляет лишь корневая страница: в ней может находиться лишь один-единственный элемент. Все листья находятся на третьем уровне. Если спроецировать Б-дерево на один-единственный уровень, включая потомков между ключами их родительской страницы, то ключи идут в возрастающем порядке слева направо. Такое размещение представляет собой естественное развитие принципа двоичных деревьев и определяет метод поиска элемента с заданным ключом. Пусть страница имеет вид, приведенный на рис. 4.46, и задан некоторый аргумент поиска x. Предположим, страница уже считана

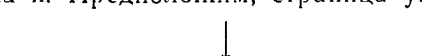


Рис. 4.46. Страница Б-дерева с m ключами.

в оперативную память и можно воспользоваться обычными методами поиска среди ключей $k_1 \ldots k_m$. Если m достаточно большое, то это может быть двоичный поиск, если же оно мало, то можно воспользоваться простым последовательным поиском. (Заметим, что время, затрачиваемое на поиск в оперативной памяти, скорее всего, пренебрежимо мало по сравнению с временем пересылки страницы из вторичной памяти в оперативную.) Если поиск неудачен, то мы попадаем в одну из следующих ситуаций:

1. $k_i < x < k_{i+1}$ для $1 \leqslant i < m$. Поиск продолжается на странице $p_i\uparrow$.
2. $k_m < x$. Поиск продолжается на странице $p_m\uparrow$.
3. $x < k_1$. Поиск продолжается на странице $p_0\uparrow$.

Если указанная ссылка имеет значение NIL, т. е. страницы-потомка не существует, то в дереве вообще не существует элемента с ключом x и поиск заканчивается.

К удивлению, включение в Б-дерево проводится сравнительно просто. Если элемент нужно поместить на страницу с $m < 2n$ элементами, то процесс включения затрагивает лишь эту страницу. Лишь включение в уже полную страницу затрагивает структуру дерева и может привести к появлению новых страниц. Для того чтобы понять, что происходит в этом случае, обратимся к рис. 4.47, иллюстрирующему процесс включения в Б-дерево порядка 2 ключа, равного 22. Включение проходит за три шага:

1. Обнаруживается, что ключ 22 отсутствует. Включение в страницу C невозможно, поскольку она уже заполнена.

2. Страница C *разделяется* на две страницы (т. е. вводится новая страница D).

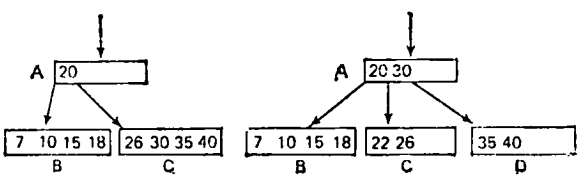


Рис. 4.47. Включение в Б-дерево ключа со значением 22.

3. Ключи — их всего 2n + 1 — поровну распределяются в C и D, а средний ключ переносится на один уровень вверх на «родительскую» страницу.

Такая весьма элегантная схема сохраняет все характерные свойства Б-деревьев. В частности, получившиеся две новые страницы содержат ровно по n элементов. Конечно, включение элемента в родительскую страницу может вновь привести к переполнению страницы, т. е. разделение будет распространяться. В крайнем случае оно может «подняться» до самого корня. Фактически только в этом случае может увеличиться высота Б-дерева. Так что Б-деревья растут несколько странно: от листьев к корню.

Теперь, основываясь на таком эскизном описании, начнем разработку полной программы. Конечно, уже ясно, что в данном случае лучше всего воспользоваться рекурсивной конструкцией, поскольку процесс разделения страниц распространяется по пути поиска в обратном направлении. Поэтому общее строение программы аналогично включению в сбалансированное дерево, хотя и есть отличие в деталях. Прежде всего нужно сформулировать определение структуры страницы. Будем считать, что элементы представляются в виде массива.

```
TYPE PPtr = POINTER TO Page;
TYPE index =[0 .. 2*n];

TYPE item = RECORD key: INTEGER;                    (4.81)
              p: PPtr;
              count: CARDINAL
            END;

TYPE page = RECORD m: index;                        (4.82)
              p0: PPtr;
              e: ARRAY [1 .. 2*n] OF item
            END
```

Как и раньше, компонента элемента с именем count предназначена для разной информации, которая относится к «смыслу» самих элементов и в реальном процессе поиска не играет никакой роли. Обратите внимание: в каждой странице достаточно места для 2n элементов. В поле m указывается фактическое число элементов на странице. Поскольку везде, кроме

корневой страницы, $m \geqslant n$, то это гарантирует по крайней мере 50 % использования памяти.

Алгоритм поиска и включения для Б-деревьев входит в прогр. 4.7 как составная часть, это процедура с именем *search*. Структура ее проста и похожа на структуру процедуры поиска для сбалансированных двоичных деревьев, только ветвление не «двоичное». Вместо этого *поиск внутри страницы* оформлен как двоичный поиск в массиве e.

Алгоритм включения выделен в отдельную процедуру лишь для ясности. Сама процедура вызывается, если поиск сигнализирует, что некоторый элемент нужно передать вверх по дереву (по направлению к корню). Об этом сообщает булевский параметр-результат h, он выполняет ту же роль, что и в алгоритме включения в сбалансированное дерево: там он указывает на рост дерева. Если h истинно, то второй параметр-результат представляет собой передаваемый элемент. Заметим, что включение начинается с мнимой страницы, так называемой «специальной» вершины (см. рис. 4.19), и новый элемент сразу же передается через параметр n для реального включения уже в «листовую» страницу. Схема этого процесса приведена в (4.83).

```
PROCEDURE search(x: INTEGER; a: PPtr; VAR h: BOOLEAN; VAR u: Item);
BEGIN
  IF a = NIL THEN (*x в дереве нет, включение *)                    (4.83)
    Элементу u присвоить x, h := TRUE, указывающее, что
    u в дереве передается вверх
  ELSE
    WITH a↑ DO
      двоичный поиск x в массиве e;
      IF найдено THEN обработка
      ELSE search(x, потомок, h, u);
        IF h THEN (* элемент *)
          IF число элементов на странице a↑ < 2n THEN
            включение u на страницу a↑ и установка h равным FALSE
          ELSE расщепление страницы и передача вверх среднего элемента
          END
        END
      END
    END
  END
END search
```

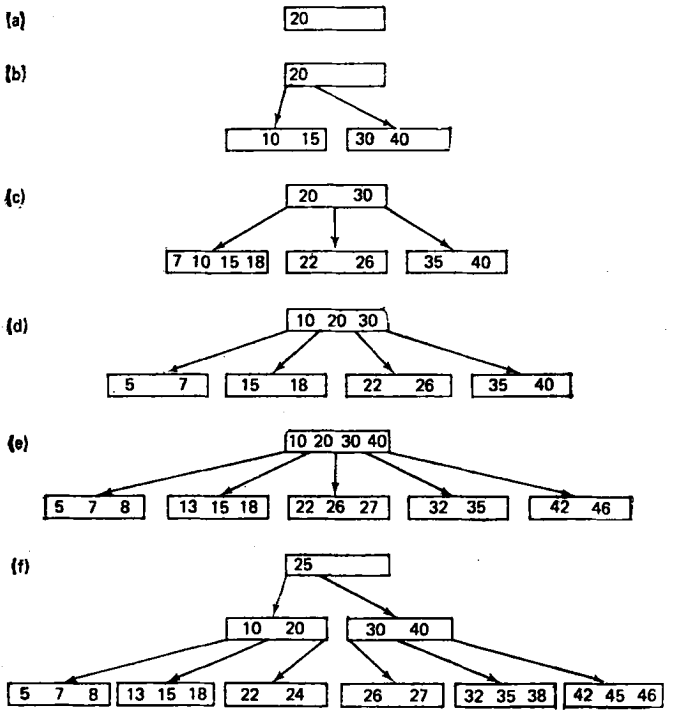


Рис. 4.48. Рост Б-дерева второго порядка.

Если после обращения к search в основной программе h имеет значение «истина», то, значит, требуется разделение корневой страницы. Поскольку эта страница особая, то процесс ее разделения приходится программировать отдельно. Он включает размещение новой корневой страницы и включение в нее единственного, переданного через параметр u элемента. Поэтому новая корневая страница будет состоять только из одного элемента. С деталями этого процесса можно познакомиться в прогр. 4.7, а на рис. 4.48 приведен результат построения с помощью этой программы Б-дерева; причем включаемые ключи идут в таком порядке:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32 38 24 45 25;

Здесь точкой с запятой отмечаются моменты появления новых страниц. Включение последнего ключа приводит к двум разделениям и появлению трех новых страниц.

Оператор присоединения (WITH) играет в этой программе особую роль. Во-первых, он указывает, что внутри тела этого оператора идентификаторы компонент страницы автоматически относятся к странице a↑. Если страницы и на самом деле расположены во вторичной памяти, что, разумеется, необходимо в больших базах данных, то оператор присоединения, кроме того, можно интерпретировать как запрос на пересылку указанной страницы в оперативную память. Каждое обращение к search предполагает пересылку только одной страницы в оперативную память, а всего для дерева из N элементов потребуется максимум $k = \log_n N$ рекурсивных обращений. Следовательно, мы должны иметь возможность накапливать в основной памяти до k страниц. Это единственное соображение, ограничивающее размер страницы 2n. На самом же деле нам надо хранить даже более чем k страниц, поскольку включение может приводить к их разделению. Кроме того, ясно, что корневую страницу лучше всего постоянно хранить в оперативной памяти, так как с нее всякий раз начинается любой поиск.

Еще одно положительное качество Б-деревьев — их экономичность и приспособленность к чисто последовательной коррекции сразу всей базы данных. Каждая страница будет вызываться в оперативную память только один раз.

Исключение элемента из Б-дерева — процесс довольно прямолинейный, хотя и усложненный в деталях. Здесь можно выделить два случая:

1. Исключаемый элемент находится на листовой странице: алгоритм удаления ясен и прост.

2. Элемент не находится на листовой странице: его нужно заменить одним из двух лексикографически смежных элементов. Если смежный элемент находится на листовой странице, то его исключить просто.

В случае 2 поиск смежного ключа похож на поиск ключа при исключении из двоичного дерева. Мы спускаемся вниз вдоль самых правых ссылок до листовой

страницы P, заменяем исключаемый элемент на правый элемент из P и уменьшаем размер P на единицу. В любом случае после уменьшения размера необходимо проверить число элементов, оставшихся на этой странице, ибо если $m < n$, то будет нарушено основное условие Б-дерева. В этом случае нужно проделать некоторые дополнительные действия. Сам факт такого *недостатка* фиксируется с помощью булевского параметра — переменной h.

Единственный выход — позаимствовать элемент с одной из соседних страниц, скажем с Q. Поскольку вызов страницы в оперативную память — операция дорогостоящая, лучше воспользоваться этой неприятной ситуацией и сделать нечто большее; например, взять не один, а несколько элементов. Обычно элементы P и Q поровну распределяются на обе страницы. Этот процесс называется *балансировкой страниц*.

Конечно, может случиться, что в Q нечего занимать, поскольку ее размер уже равен минимальному — n. В этом случае общее число элементов на P и Q равно 2n — 1, и мы можем *слить* обе страницы в одну, добавив сюда средний элемент из родительской (для P и Q) страницы, а затем целиком уничтожить страницу Q. Эти действия в точности обратны процессу разделения страниц, и за ними можно проследить на примере удаления ключа 22 (рис. 4.47). Удаление среднего ключа на родительской странице вновь может привести к тому, что ее размер станет меньше критического и потребуется проведение специальных действий (балансировки или слияния) уже на следующем уровне. В экстремальном случае слияние страниц может распространиться вверх до самой корневой страницы. Если корень сократится до нулевого размера, то он удаляется, и тем самым высота Б-дерева уменьшается. Фактически это единственный случай, когда высота дерева может сократиться. На рис. 4.49 показана постепенная деградация Б-дерева с рис. 4.48 в случае удаления ключей в такой последовательности:

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;

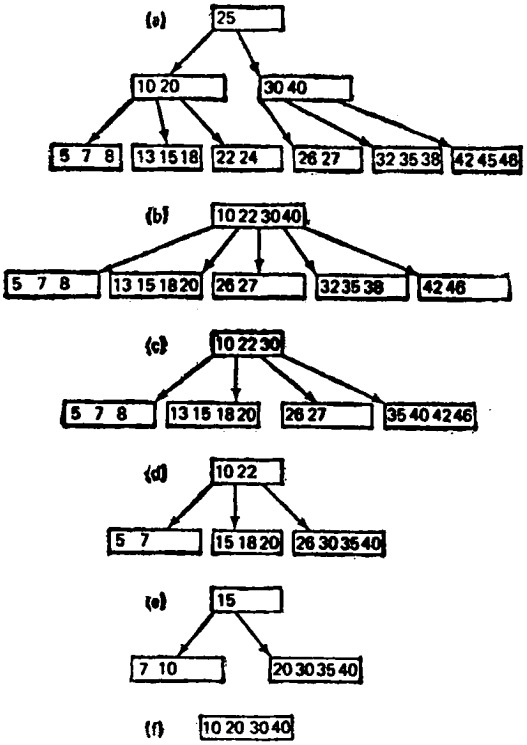


Рис. 4.49. Деградация Б-дерева второго порядка.

Как и раньше, точки с запятой фиксируют моменты, в которые делаются «снимки дерева», т. е. моменты уничтожения страниц. Алгоритм исключения в прогр. 4.7 оформлен в виде процедуры. Обратите внимание на его сходство с исключением из сбалансированного дерева.

В уже упоминавшейся работе Бэйера и Маккрейта проводится всесторонний анализ «производительности» алгоритмов, основанных на Б-деревьях. В частности, исследуется вопрос выбора оптимального размера страницы, зависящего от характеристик вычислительной системы и ее памяти.

В книге Кнута, т. 3, с. 567—570, обсуждаются различные варианты схем на Б-деревьях. Вот наиболее

```
MODULE BTree;
 FROM InOut IMPORT OpenInput, OpenOutput, CloseInput, CloseOutput,
    ReadInt, Done, Write, WriteInt, WriteString, WriteLn;
 FROM Storage IMPORT ALLOCATE;

 CONST n = 2;

 TYPE PPtr = POINTER TO Page;

  Item = RECORD key: INTEGER;
       p: PPtr;
       count: CARDINAL
      END;

  Page = RECORD m: [0 .. 2*n]; (* число элементов на странице *)
       p0: PPtr;
       e: ARRAY [1 .. 2*n] OF Item
      END;

 VAR root, q: PPtr;
   x: INTEGER;
   h: BOOLEAN;
   u: Item;
 PROCEDURE search(x: INTEGER; a: PPtr; VAR h: BOOLEAN; VAR v: Item);
  (* поиск ключа x в Б-дереве с корнем a; если он находится, то счетчик
  увеличивается. Если ключа нет, то включается новый элемент с ключом x.
  Если элемент передается вверх, то его присваивают v; h означает "дерево
  стало выше" *)
  VAR i, L, R: CARDINAL; b: PPtr; u: Item;
 BEGIN (*~h*)
  IF a = NIL THEN h := TRUE; (* в дереве нет *)
   WITH v DO
    key := x; count := 1; p := NIL
   END
  ELSE
   WITH a^ DO
    L := 1; R := m+1; (* двоичный поиск *)
    WHILE L < R DO
     i := (L+R) DIV 2;
     IF e[i].key <= x THEN L := i+1 ELSE R := i END
    END;
    R := R-1;
    IF (R > 0) & (e[R].key = x) THEN INC(e[R].count)
```

```
    ELSE (*на этой странице элемента нет*)
     IF R = 0 THEN search(x, p0, h, u)
      ELSE search(x, e[R].p, h, u)
     END;
     IF h THEN (*включение u вправо от e[R]*)
      IF m < 2*n THEN
       h := FALSE; m := m+1;
       FOR i := m TO R+2 BY -1 DO e[i] := e[i-1] END;
       e[R+1] := u
      ELSE ALLOCATE(b, SIZE(Page)); (*переполнение*)
       (*расщепление a на a и b; средний элемент в b*)
       IF R <= n THEN
        IF R = n THEN v := u
        ELSE v := e[n];
         FOR i := n TO R+2 BY -1 DO e[i] := e[i-1] END;
         e[R+1] := u
        END;
        FOR i := 1 TO n DO b↑.e[i] := a↑.e[i+n] END
       ELSE (*включение в правую страницу*)
        R := R-n; v := e[n+1];
        FOR i := 1 TO R-1 DO b↑.e[i] := a↑.e[i+n+1] END;
        b↑.e[R] := u;
        FOR i := R+1 TO n DO b↑.e[i] := a↑.e[i+n] END
       END;
       m := n; b↑.m := n; b↑.p0 := v.p; v.p := b
      END
     END
    END
   END
  END
END search;

PROCEDURE underflow(c, a: PPtr; s: CARDINAL; VAR h: BOOLEAN);
 (*a = переполнившаяся страница, c = страница "предок"
   s = индекс исключаемого из c элемента, h := *)
 VAR b: PPtr; VAR i, k, mb, mc: CARDINAL;
BEGIN mc := c↑.m; (*h, a↑.m = n-1*)
 IF s < mc THEN
  (*b := страница справа от a*) s := s+1;
  b := c↑.e[s].p; mb := b↑.m; k := (mb-n+1) DIV 2;
  (*k = число элементов, доступных на странице b*)
```

```
  at.e[n] := ct.e[s]; at.e[n].p := bt.p0;
  IF k > 0 THEN
   (* передача k элементов из b в a *)
   FOR i := 1 TO k-1 DO at.e[i+n] := bt.e[i] END;
   ct.e[s] := bt.e[k]; ct.e[s].p := b;
   bt.p0 := bt.e[k].p; mb := mb - k;
   FOR i := 1 TO mb DO bt.e[i] := bt.e[i+k] END;
   bt.m := mb; at.m := n-1+k; h := FALSE
  ELSE (* слияние страниц a и b *)
   FOR i := 1 TO n DO at.e[i+n] := bt.e[i] END;
   FOR i := s TO mc-1 DO ct.e[i] := ct.e[i+1] END;
   at.m := 2*n; ct.m := mc-1; h := mc <= n;
   (*Deallocate(b)*)
  END
 ELSE (*b := страница слева от a *)
  IF s = 1 THEN b := ct.p0 ELSE b := ct.e[s-1].p END;
  mb := bt.m + 1; k := (mb-n) DIV 2;
  IF k > 0 THEN
   (* передача k элементов из b в a *)
   FOR i := n-1 TO 1 BY -1 DO at.e[i+k] := at.e[i] END;
   at.e[k] := ct.e[s]; at.e[k].p := at.p0; mb := mb-k;
   FOR i := k-1 TO 1 BY -1 DO at.e[i] := bt.e[i+mb] END;
   at.p0 := bt.e[mb].p; ct.e[s] := bt.e[mb]; ct.e[s].p := a;
   bt.m := mb-1; at.m := n-1+k; h := FALSE
  ELSE (* слияние a и b *)
   bt.e[mb] := ct.e[s]; bt.e[mb].p := at.p0;
   FOR i := 1 TO n-1 DO bt.e[i+mb] := at.e[i] END;
   bt.m := 2*n; ct.m := mc-1; h := mc <= n;
   (*Deallocate(a)*)
  END
 END
END underflow;

PROCEDURE delete(x: INTEGER; a: PPtr; VAR h: BOOLEAN);
 (* поиск и исключение ключа x из Б-дерева; если страница становится мала,
  то балансировка или слияние с соседней; h := "страница a мала" *)
 VAR i, L, R: CARDINAL; q: PPtr;

 PROCEDURE del(P: PPtr; VAR h: BOOLEAN);
  VAR q: PPtr; (* глобальные a, R *)
 BEGIN
```

```
    WITH P↑ DO
     q := e[m].p;
     IF q # NIL THEN del(q,h);
      IF h THEN underflow(P, q, m, h) END
     ELSE
      P↑.e[m].p := a↑.e[R].p; a↑.e[R] := P↑.e[m];
      m := m - 1; h := m < n
     END
    END
  END del;

BEGIN
  IF a = NIL THEN (* в дереве x нет *)  h := FALSE
  ELSE
    WITH a↑ DO
     L := 1; R := m+1;  (* двоичный поиск *)
     WHILE L < R DO
      i := (L+R) DIV 2;
      IF e[i].key < x THEN L := i+1 ELSE R := i END
     END ;
     IF R = 1 THEN q := p0 ELSE q := e[R-1].p END ;
     IF (R <= m) & (e[R].key = x) THEN
      (* найден, исключение *)
      IF q = NIL THEN (* a - терминальная страница *)
       m := m-1; h := m < n;
       FOR i := R TO m DO e[i] := e[i+1] END
      ELSE del(q,h);
       IF h THEN underflow(a, q, R-1, h) END
      END
     ELSE delete(x, q, h);
      IF h THEN underflow(a, q, R-1, h) END
     END
    END
  END
END delete;

PROCEDURE PrintTree(p: PPtr; level: CARDINAL);
  VAR i: CARDINAL;
BEGIN
  IF p # NIL THEN
    FOR i := 1 TO level DO WriteString("      ") END ;
```

```
    FOR i := 1 TO p↑.m DO WriteInt(p↑.e[i].key, 4) END ;
    WriteLn;
    PrintTree(p↑.p0, level+1);
    FOR i := 1 TO p↑.m DO PrintTree(p↑.e[i].p, level+1) END
  END
END PrintTree;

BEGIN (* главная программа *)
  OpenInput("TEXT"); OpenOutput("TREE");
  root := NIL; Write(">"); ReadInt(x);
  WHILE Done DO
    WriteInt(x, 5); WriteLn;
    IF x >= 0 THEN
      search(x, root, h, u);
      IF h THEN (* включение новой корневой страницы *)
        q := root; ALLOCATE(root, SIZE(Page));
        WITH root↑ DO
          m := 1; p0 := q; e[1] := u
        END
      END
    ELSE
      delete(-x, root, h);
      IF h THEN (* размер корневой страницы уменьшился *)
        IF root↑.m = 0 THEN
          q := root; root := q↑.p0; (*Deallocate(q)*)
        END
      END
    END ;
    PrintTree(root, 0); WriteLn;
    Write(">"); ReadInt(x)
  END ;
  CloseInput; CloseOutput
END BTree.
```

Прогр. 4.7. Поиск, включение и исключение из Б-деревьев.

существенное замечание — следует тормозить разделение страниц тем же способом, которым тормозится их слияние, — балансировкой страниц. Остальные улучшения, по-видимому, не дают существенного выигрыша. С исчерпывающим же обзором Б-деревьев можно познакомиться в работе [4.8].

4.7.2. Двоичные Б-деревья

На первый взгляд кажется, что наименьший интерес представляют Б-деревья первого порядка ($n = 1$). Однако иногда стоит обращать внимание и на

такие исключительные варианты. Тем не менее сразу же ясно, что Б-деревья первого порядка не имеет смысла использовать для представления больших, упорядоченных, индексированных множеств данных, требующих вторичной памяти: ведь приблизительно 50 % всех страниц будут содержать только один-единственный элемент. Поэтому забудем о вторичной памяти и займемся вновь задачей построения деревьев поиска, находящихся лишь в одноуровневой оперативной памяти.

Двоичное Б-дерево (ДБ-дерево) состоит из вершин (страниц) с одним или двумя элементами. Следовательно, страница содержит две или три ссылки на потомков (отсюда еще один термин — *2-3-дерево*). По определению в Б-дереве все листовые страницы расположены на одном уровне, а все нелистовые страницы (включая и корневую) имеют двух или трех потомков. Поскольку теперь мы имеем дело только с оперативной памятью, то следует заботиться об экономном ее использовании. Поэтому представление элементов внутри страницы в виде массива уже не подходит. Выход из положения — динамическое размещение, на основе списочной структуры, — т. е. «внутри» вершины существует список из 1 или 2 элементов. Так как каждая вершина может иметь не более трех потомков и, следовательно, должна содержать самое большее три ссылки, то мы попытаемся объединять ссылки на потомков со ссылками в списке элементов (см. рис. 4.50). Таким образом, вершины Б-дерева теряют свою целостность и элементы начинают играть роль вершин в регулярном двоичном де-

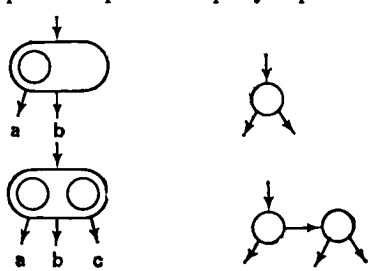


Рис. 4.50. Представление вершин ДБ-дерева.

реве. Однако остается необходимость делать различие между ссылками на потомков (по вертикали) и ссылками на «братьев» в одной странице (по горизонтали). Так как по горизонтали могут быть только ссылки вправо, то для фиксации направления достаточно одного-единственного разряда. Поэтому мы вводим булевское поле h, говорящее, что ссылка идет по горизонтали. Определение вершины дерева, соответствующее такому направлению, приведено ниже (4.84). «Схема» вершины была предложена и исследована Р. Бэйером в 1971 г. [4.3]. Такая организация дерева поиска гарантирует, что максимальная длина пути $p = 2 * \lceil \log N \rceil$.

```
TYPE Ptr = POINTER TO Node;
TYPE Node = RECORD key: INTEGER;                    (4.84)
              .....
              left, right: Ptr;
              h: BOOLEAN (* горизонтальная правая ветвь *)
            END
```

Рассматривая задачу включения ключей, следует различать четыре возможных ситуации, возникающих при росте левых или правых поддеревьев. Все эти четыре случая показаны на рис. 4.51. Напомним, что для Б-деревьев характерен рост от основания вверх к корню и необходимо следить, чтобы все листья были на одном уровне. Самый простой случай (1) — рост правого поддерева вершины А, причем А — единственный элемент на странице (мнимой). В этой ситуации предок В просто становится братом А, т. е. «вертикальная» ссылка превращается в «горизонтальную». Если вершина А уже имеет брата, то такой простой «поворот» правой ветви невозможен. Мы получаем страницу с тремя вершинами, и ее необходимо разделить (2). Средняя вершина страницы (В) передается на следующий вышележащий уровень.

Представим теперь, что растет в высоту левое поддерево вершины В. Если В снова одна на странице (случай 3), т. е. ее правая ссылка указывает лишь на предка, то левое поддерево (А) может стать братом В. (Так как левая ссылка не может быть горизонтальной, то требуется некоторое простое «враще-

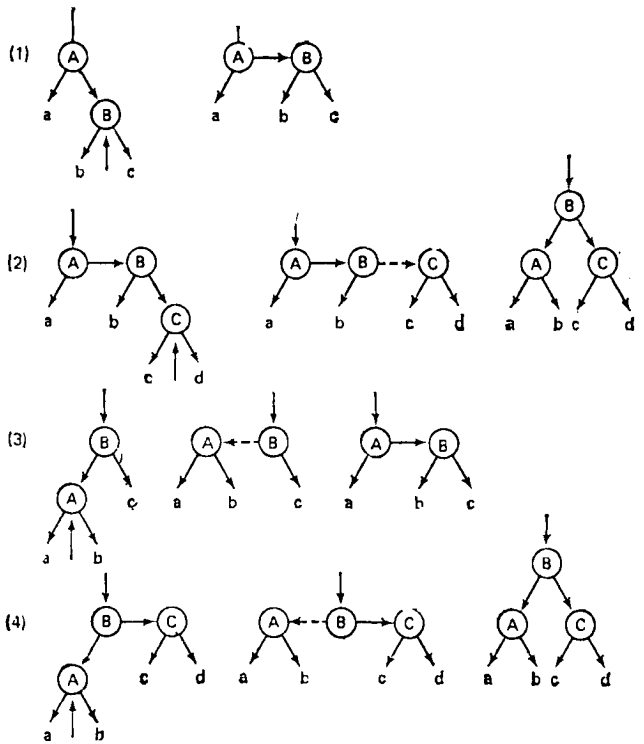


Рис. 4.51. Включение вершин в ДБ-дерево.

ние» ссылок*).) Если же B уже имеет брата, то «подъем» A приводит к странице с тремя элементами, требующей разделения. Разделение происходит очень просто: вершина C становится потомком B, а последняя поднимается уровнем выше (случай 4).

Следует заметить, что при поиске некоторого ключа фактически нет разницы, двигаемся мы вдоль горизонтальной или вдоль вертикальной ссылки. Поэтому забота о горизонтальности левой ссылки в третьем случае, когда страница содержит не более двух элементов, выглядит несколько искусственной. И действительно, алгоритм включения странно асимметричен

*) Конечно, можно назвать этот процесс «вращением» (rotation), но лучше посмотреть на рис. 4.51 (3). — *Прим. перев.*

в том, что касается роста левого или правого поддерева, отсюда и вся организация ДБ-деревьев кажется какой-то надуманной. Доказательства этой надуманности нет, но интуиция говорит, что здесь «что-то не то» и нужно было бы от асимметрии избавиться. Так появляется понятие *симметричного двоичного Б-дерева* (СДБ-дерево), предложенное в 1972 г. Бэйером [4.4]. Новая организация ведет к несколько лучшему среднему поиску, однако алгоритмы включения и исключения становятся опять же несколько более сложными. Кроме того, в каждой вершине нужны теперь два разряда (булевские переменные lh и rh), указывающие природу двух ссылок.

Поскольку мы ограничимся лишь детальным разбором алгоритма включения, нам нужно вновь разобраться в четырех случаях роста поддеревьев. Они приведены на рис. 4.52: ясно видно, что мы добились симметрии. Обратите внимание: при росте любого поддерева вершины A, не имеющей братьев, корень

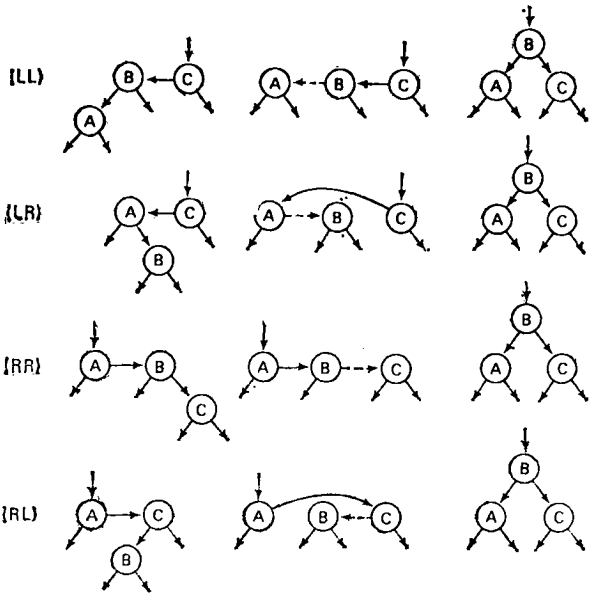


Рис. 4.52. Включение в СДБ-дерево.

этого поддерева становится для A братом. Этот случай не нуждается в дальнейшем обсуждении.

Все четыре случая, приведенные на рис. 4.52, отражают ситуацию переполнения страницы и ее последующего разделения. Они помечены первыми буквами направлений ссылок (L и R) у трех братьев дерева в середине рисунка. Начальное положение приведено в левой части, в середине — то, что получается при подъеме нижней вершины, вызванном ростом ее поддерева, а в правой части — то, что получается после перестановки вершин.

Желательно в дальнейшем уже не возвращаться к понятию страницы, с которого начиналась новая организация дерева: ведь мы заинтересованы лишь в том, чтобы максимальная длина пути была ограничена величиной $2 * \log N$. Для этого же нужно быть уверенными, что нигде на любом из путей не встречаются две подряд горизонтальные ссылки. Однако нет никаких причин запрещать вершины со ссылками вправо и влево. Поэтому мы определим СДБ-деревья как деревья, обладающие следующими свойствами:

1. Каждая вершина содержит один ключ и не более двух ссылок на поддеревья.

2. Ссылки бывают горизонтальные и вертикальные. Нет ни одного пути поиска с двумя подряд горизонтальными ссылками.

3. Все терминальные вершины (вершины, не имеющие потомков) находятся на одном (терминальном) уровне.

Из этого следует, что длина самого протяженного пути поиска не более чем вдвое превышает высоту дерева. Так как высота никакого СДБ-дерева с N вершинами не может быть выше $\log N$, то отсюда следует, что верхняя граница пути поиска — $2 * \lceil \log N \rceil$. На рис. 4.53 показано, как происходит рост четырех таких деревьев, когда в них последовательно включаются следующие ключи:

(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; (4.85)

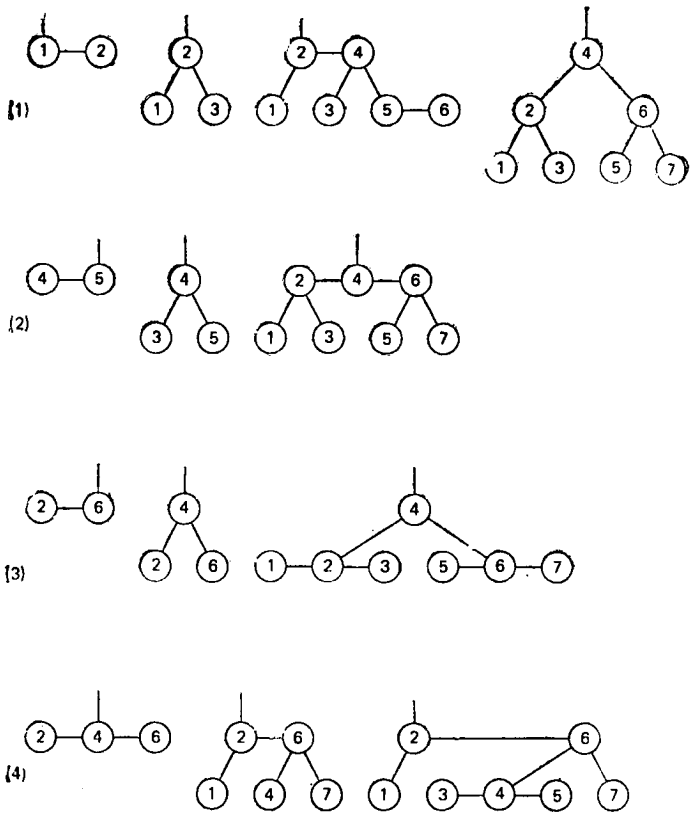


Рис. 4.53. Рост СДБ-дерева при включении вершин из последовательности (4.85).

В каждой «строке» рисунка приведены последовательные состояния соответствующего дерева в моменты, как и раньше отмеченные в последовательности ключей точкой с запятой. Рисунки особенно наглядно демонстрируют третье свойство СДБ-деревьев — все терминальные вершины лежат на одном уровне. Поэтому напрашивается сравнение этих структур с только что подстриженным садовым кустарником.

Алгоритм построения СДБ-деревьев приведен в (4.87). Он основан на определении типа Node (4.86)

с двумя компонентами lh и rh, указывающими на горизонтальность левой и правой ссылки.

```
TYPE Node = RECORD key: Integer;
              count: CARDINAL;
              left, right: Ptr;                    (4.86)
              lh, rh: BOOLEAN
            END
```

Рекурсивная процедура *search* (4.87), как и раньше, следует схеме основного алгоритма включения в двоичное дерево. Добавляется третий параметр h, он указывает, изменяется или нет поддерево с корнем p, и полностью соответствует параметру h в программе поиска в Б-дереве. Однако нужно указать и на последствия представления страниц в виде связанных списков: любая страница проходит с помощью одного или двух обращений к процедуре поиска. Необходимо различать такие случаи: выросло поддерево (указанное вертикальной ссылкой) или «братская» вершина (указанная горизонтальной ссылкой) получила еще одного брата и, следовательно, требуется разделение страницы. Проблему легко решить, если считать, что h принимает одно из таких трех значений:

1. h = 0: поддерево p не приводит к каким-либо изменениям в структуре дерева.
2. h = 1: вершина p получила брата.
3. h = 2: поддерево p увеличилось в высоту.

Заметим, что действия по перестановке вершин очень похожи на аналогичные действия в алгоритме поиска в сбалансированных деревьях (4.63). Из (4.87) ясно, что все четыре случая можно реализовать с помощью простого «поворота» ссылок; в случаях LL и RR это однократный поворот, а в случаях LR и RL — двукратный. Фактически процедура (4.87) оказывается даже несколько проще, чем (4.63). Ясно, что схема с СДБ-деревьями представляет собой альтернативу критерию АВЛ-сбалансированности. Поэтому хорошо было бы сравнить их «производительности» *).

*) Возможно, кому-либо не нравится термин «производительность» в контексте разговора о программах. Но ничего лучшего нет. — *Прим. перев.*

```
PROCEDURE search(x: INTEGER; VAR p: Ptr; VAR h: CARDINAL);
 VAR p1, p2: Ptr;  (*h = 0*)
BEGIN                                                          (4.87)
 IF p = NIL THEN (*включение*)
  ALLOCATE(p, SIZE(Node)); h := 2;
  WITH p↑ DO
   key := x; count := 1;
   left := NIL; right := NIL; lh := FALSE; rh := FALSE
  END
 ELSIF p↑.key > x THEN
  search(x, p↑.left, h);
  IF h > 0 THEN (*выросла левая ветвь*)
   IF p↑.lh THEN
    p1 := p↑.left; h := 2; p↑.lh := FALSE;
    IF p1↑.lh THEN (*LL*)
     p↑.left := p1↑.right; p1↑.right := p; p := p1;
     p↑.lh := FALSE
    ELSIF p1↑.rh THEN (*LR*)
     p2 := p1↑.right; p1↑.right := p2↑.left; p2↑.left := p1;
     p↑.left := p2↑.right; p2↑.right := p; p := p2;
     p1↑.rh := FALSE
    END
   ELSE h := h-1;
    IF h > 0 THEN p↑.lh := TRUE END
   END
  END
 ELSIF p↑.key < x THEN
  search(x, p↑.right, h);
  IF h > 0 THEN (*выросла правая ветвь*)
   IF p↑.rh THEN
    p1 := p↑.right; h := 2; p↑.rh := FALSE;
    IF p1↑.rh THEN (*RR*)
     p↑.right := p1↑.left; p1↑.left := p; p := p1;
     p1↑.rh := FALSE
    ELSIF p1↑.lh THEN (*RL*)
     p2 := p1↑.left; p1↑.left := p2↑.right; p2↑.right := p1;
     p↑.right := p2↑.left; p2↑.left := p; p := p2;
     p1↑.lh := FALSE
    END
   ELSE h := h-1;
    IF h > 0 THEN p↑.rh := TRUE END
   END
  END
 ELSE (*найден*) p↑.count := p↑.count + 1
 END
END search
```

Мы не будем проводить математический анализ, а сконцентрируем внимание лишь на некоторых основных отличиях. Можно доказать, что АВЛ-сбалансированные деревья представляют собой подмножество всех СДБ-деревьев. Следовательно, класс последних шире. Отсюда следует, что их длина пути в среднем больше, чем у АВЛ-деревьев. Заметим, что наихудшее в этом отношении дерево — дерево (4) на рис. 4.53. С другой стороны, здесь реже приходится переставлять вершины. Поэтому сбалансированные деревья предпочтительнее в тех случаях, когда поиск ключей происходит значительно чаще, чем включение (или исключение). Если это не так, то можно отдать предпочтение и схеме на СДБ-деревьях. Очень трудно сказать, где проходит граница. Строго говоря, она зависит не только от отношения между частотами поисков и изменений структуры, но и от особенностей реализации. Например, записи, соответствующие вершинам, могут быть плотно упакованными, и поэтому обращение к их полям требует выделения частей слова.

4.8. ДЕРЕВЬЯ ПРИОРИТЕТНОГО ПОИСКА

Деревья, и в частности, двоичные деревья, представляют собой очень эффективную организацию данных, для которых существует отношение линейной упорядоченности. В предыдущих разделах мы уже представили наиболее часто встречающиеся изобретательные схемы, обеспечивающие эффективный поиск и сопровождение (включение, исключение), представление таких данных. Однако деревья, как это кажется, бесполезны в задачах, где данные располагаются не в одномерном, а в многомерных пространствах. Хотя случай двумерного пространства и является во многих практических приложениях весьма важным, тем не менее фактически организация эффективного поиска в многомерном пространстве остается одной из наиболее «неуловимых» проблем информатики.

Однако при внимательном знакомстве с сутью дела оказывается, что из деревьев все еще можно извлечь

пользу, по крайней мере в случае двумерных пространств. В конце концов, мы ведь рисуем на бумаге деревья в двумерном пространстве. Поэтому давайте кратко перечислим характеристики двух главных видов деревьев, с которыми мы до сих пор встречались.

1. Дерево поиска управляется инвариантами:

$$\begin{aligned} &p.left \neq NIL \rightarrow p.left.x < p.x \\ &p.right \neq NIL \rightarrow p.x < p.right.x \end{aligned} \qquad (4.88)$$

справедливыми для всех вершин p с ключом k. Ясно, что инвариантом ограничиваются лишь *горизонтальные* положения вершин, а вертикальные положения вершин можно выбирать так, чтобы минимизировать время доступа при поиске (т. е. длину пути).

2. Пирамида, называемая также *деревом с приоритетом*, управляется инвариантами

$$\begin{aligned} &p.left \neq NIL \rightarrow p.y \leqslant p.left.y \\ &p.right \neq NIL \rightarrow p.y \leqslant p.right.y \end{aligned} \qquad (4.89)$$

справедливыми для всех вершин p с ключом k. В этом случае ясно, что инвариантами ограничиваются только *вертикальные* положения.

Кажется вполне естественным объединить эти два условия для определения некоторой древовидной организации в двумерном пространстве, где каждая вершина имеет два ключа: x и y, которые можно рассматривать как координаты данной вершины. Такое дерево представляет собой множество точек на плоскости, т. е. в двумерном декартовом пространстве, поэтому такие деревья называются *декартовыми деревьями* [4.9]. Мы же предпочитаем употреблять термин *деревья приоритетного поиска*, поскольку из него следует, что эта структура произошла из объединения деревьев с приоритетами и деревьев поиска. Для них характерны инварианты, справедливые для каждой вершины *p*:

$$\begin{aligned} &p.left \neq NIL \rightarrow (p.left.x < p.x) \,\&\, (p.y \leqslant p.left.y) \\ &p.right \neq NIL \rightarrow (p.x < p.right.x) \,\&\, (p.y \leqslant p.right.y) \end{aligned} \qquad (4.90)$$

Однако нет ничего удивительного в том, что «поисковые возможности» таких деревьев практически крайне

бедны. В конце концов, большая степень свободы в выборе положения вершины становится недостатком, и мы больше не имеем возможности выбирать размещение, ведущее к короткому пути. И действительно, уже нельзя гарантировать для усилий, затрачиваемых на поиск, включение или исключение элемента, каких-либо «логарифмических» границ. И хотя с этим мы уже встречались в случае простого, несбалансированного дерева поиска, тем не менее вероятность хорошего в среднем поведения схемы крайне мала. Даже еще хуже — действия по сопровождению могут стать почти неуправляемыми. Рассмотрим, например, дерево с рис. 4.54(a). Включение новой вершины C с координатами, требующими разместить ее выше и между A и B, приводит к значительным затратам по перестройке (a) в (b).

Маккрейт предложил схему, похожую на балансировку, в которой за счет более усложненных операций включения и исключения удастся гарантировать логарифмические оценки для этих операций. Он назвал соответствующую структуру деревом поиска с приоритетом [4.10], однако в соответствии с нашей классификацией ее следовало бы назвать *сбалансированным деревом поиска с приоритетом*. От рассмотрения этой структуры мы воздержимся из-за ее сложности и практической малопригодности. Занимаясь более ограниченной, но практически не менее «злободневной» проблемой, Маккрейт предложил еще

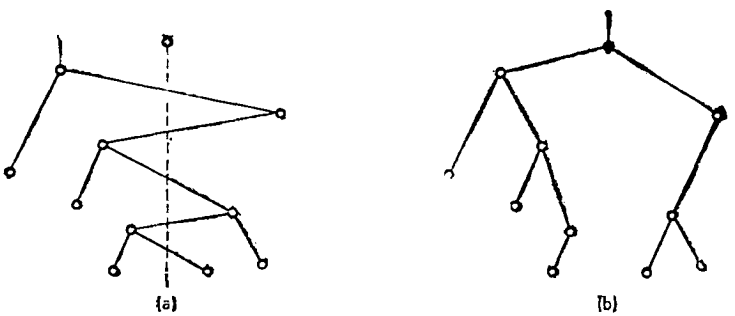


Рис. 4.54. Включение в дерево приоритетного поиска.

одну структуру дерева, которую мы здесь детально обсудим. Вместо неограниченного пространства поиска мы будем рассматривать данные в пространстве, ограниченном прямоугольником с двумя открытыми сторонами. Граничные значения для координаты x мы будем обозначать через xmin и xmax.

В схеме с (несбалансированным) деревом поиска с приоритетом, упомянутым выше, каждая вершина p делит плоскость прямой $x = p.x$ на две части. Все вершины левого поддерева находятся слева от нее, а вершины правого поддерева — справа. Для эффективного поиска такой выбор может оказаться неподходящим. К счастью, мы можем выбирать разделяющую прямую различными способами. Свяжем с каждой из вершин p некоторый интервал [p.L ... p.R), ранжирующий все величины x, начиная с x.L до x.R (исключая последнее). Это будет интервал, в котором может находиться x-значение вершины. Затем постулируем, что левый потомок (если он есть) должен находиться внутри левой половины этого интервала, а правый потомок — внутри правой половины. Следовательно, разделяющая прямая будет не p.x, а $(p.L + p.R)/2$. Для каждого потомка интервал уменьшается вдвое, и, таким образом, высота дерева ограничена значением $\log(x_{max} - x_{min})$. Это заключение верно только в том случае, если никакие две вершины не имеют одинаковых x-значений. Однако выполнение этого условия гарантируется инвариантом (4.90). Если мы имеем дело с целочисленными координатами, то предел, самое большее, равен длине слова используемой машины. Фактически поиск идет аналогично поиску делением пополам или корневому поиску (radix search), и поэтому такие деревья называются *корневыми деревьями поиска с приоритетом* (radix priority search trees) [4.10]. В них число операций, затрачиваемых на поиск, включение и исключение, ограничено логарифмической оценкой и подчиняется в каждой вершине p таким инвариантам:

$$p.left \neq NIL \rightarrow (p.L \leqslant p.left.x < p.M) \,\&\, (p.y \leqslant p.left.y) \quad (4.91)$$
$$p.right \neq NIL \rightarrow (p.M \leqslant p.right.x < p.R) \,\&\, (p.y \leqslant p.right.y)$$

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 вершин p, а root.L = x_{min}, root.R = x_{max}.

Решающее достоинство корневой схемы заключается в том, что операции по поддержанию (т. е. сохранению инвариантов при включении или исключении) ограничиваются одним-единственным «разрезом» дерева, поскольку разделяющая линия имеет фиксированное x вне зависимости от x-значения включаемых вершин.

Типичными для деревьев поиска с приоритетом операциями являются включение, исключение, нахождение элемента с минимальным (максимальным) значением x (или y), большим (меньшим) заданного предела, и нумерация точек, лежащих внутри заданного прямого угла. Ниже проводятся процедуры для включения и нумерации. Они базируются на таких описаниях типов:

```
TYPE Ptr = POINTER TO Node;
  Node = RECORD                                          (4.92)
           x: [xmin .. xmax]; y: CARDINAL;
           left, right: Ptr
         END
```

Обратите внимание, что нет нужды записывать в самую вершину атрибуты xL и xR. Более того, они вычисляются в процессе поиска. Однако это требует введения в рекурсивную процедуру insert двух дополнительных параметров. Их значения при первом обращении (с p = root) равны соответственно x_{min} и x_{max}. Если не считать этого, то процесс поиска аналогичен поиску в любом, регулярном дереве поиска. Если встречается пустая вершина, то элемент включается в дерево. Если вершина, которую нужно вставить, имеет y-значение, меньшее чем такое же значение у только что пройденной вершины, то новая вершина и эта пройденная «меняются местами». И наконец, вершина включается в левое поддерево, если ее x-значение менее среднего значения интер-

вала, в противном случае она включается в правое поддерево.

```
PROCEDURE insert(VAR p: Ptr; X, Y, xL, xR: CARDINAL);
 VAR xm, t: CARDINAL;
BEGIN
 IF p = NIL THEN (* нет в дереве, включение *)
  ALLOCATE(p, SIZE(Node));
  WITH p↑ DO
   x := X; y := Y; left := NIL; right := NIL
  END
 ELSIF p↑.x = X THEN (*найдено; не включать *)
 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
```

Приведенная ниже процедура enumerate нумерует все точки x, y, лежащие внутри заданного прямого угла, т. е. удовлетворяющие соотношениям $x0 \leqslant x < x1$ и $y \leqslant y1$. При обнаружении каждой такой точки следует обращение к некоторой процедуре report(x, y). Заметьте, что одна сторона прямого угла совпадает с осью x, т. е. нижняя граница для y равна 0. Это гарантирует, что процесс нумерации потребует, самое большее, $O(\log N + s)$ операции, где N — мощность пространства поиска x, а s — число пронумерованных вершин.

```
PROCEDURE enumerate(p: Ptr; x0, x1, y, xL, xR: CARDINAL);
 VAR xm: CARDINAL;
BEGIN
 IF p # NIL THEN
  IF (p↑.y <= y) & (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, y, xL, xm) END;
  IF xm < x1 THEN enumerate(p↑.right, x0, x1, y, xm, xR) END
 END
END enumerate
```

УПРАЖНЕНИЯ

4.1. Введем понятие рекурсивного типа и будем его описывать таким образом:

```
RECTYPE T = T0
```

Рекурсивный тип определяет множество значений, определенных типом T0, к которому добавлено единственное значение NONE. В этом случае определение типа ped (см. (4.3)) можно упростить до такого:

```
RECTYPE ped = RECORD name: alfa;
                     father, mother: ped
                     END
```

Как будет располагаться в памяти рекурсивная структура, соответствующая рис. 4.2? Очевидно, реализация такого типа будет основываться на схеме динамического распределения памяти, причем поля father и mother из нашего примера должны автоматически дополняться ссылками, скрытыми от программиста. Какие трудности могут встретиться при реализации такой схемы?

4.2. Определите структуру данных, описанных в последнем подразделе разд. 4.2, с помощью записей и ссылок. Можно ли представить данные такого класса в терминах рекурсивных типов, определенных в предыдущем упражнении?

4.3. Предположим, что есть очередь Q с элементами типа T0, построенная на принципе «первый пришел — первый вышел». Определите модуль с соответствующими описанием данных, процедурами для включения и исключения элементов из Q и функцией, определяющей, не пуста ли очередь. В процедурах должны предусматриваться собственные механизмы экономного повторного использования памяти.

4.4. Предположим, записи в связном списке содержат ключевое поле, относящееся к типу INTEGER. Напишите программу, сортирующую список в порядке возрастания ключей. А затем постройте процедуру, «оборачивающую» такой список.

4.5. В круговых списках обычно предусматривается так называемый *заголовок списка* (см. рис. 4.55). Зачем необходимы такие заголовки? Напишите процедуры для включения, исключения и поиска элемента, идентифицированного заданным клю-

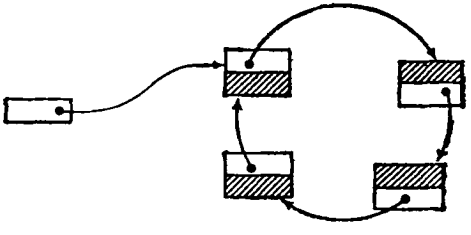


Рис. 4.55. Круговой список.

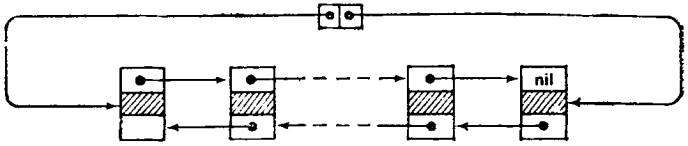


Рис. 4.56. Двунаправленный список.

чом. Проделайте это для списка с заголовком, а потом — без заголовка.

4.6. Двунаправленный список — это список с элементами, у которых связи указывают в обе стороны (см. рис. 4.56). Обе связи начинаются из заголовка. Как и в предыдущем упражнении, напишите модуль с процедурами для поиска, включения и исключения элементов.

4.7. Будет ли правильно работать программа 4.2, если во входном потоке некоторая пара ⟨x, y⟩ встретится более одного раза?

4.8. В программе 4.2 сообщение «This set is not partially ordered» («Это множество не является частично упорядоченным») во многих случаях бесполезно. Дополните программу так, чтобы она, если цикл среди элементов существует, выдавала его явно.

4.9. Напишите программу, которая будет читать текст любой программы, выделять из него определения процедур и обращения к ним, а затем пытаться определить, есть ли среди процедур топологическая упорядоченность. Будем считать, что $P \prec Q$, если Q обращается к P.

4.10. Нарисуйте дерево, которое построит прогр. 4.7, если на вход подаются числа 1, 2, 3, ..., n.

4.11. В какой последовательности будут посещаться вершины дерева на рис. 4.23, если оно обходится сверху вниз, слева направо и снизу вверх?

4.12. Найдите правило построения последовательности из n чисел, для которой программа 4.4 построит идеально сбалансированное дерево.

4.13. Рассмотрим два порядка обхода двоичных деревьев:

a1. Обходится правое поддерево.
a2. Посещается корень.
a3. Обходится левое поддерево.
b1. Посещается корень.
b2. Обходится правое поддерево.
b3. Обходится левое поддерево.

Имеются ли какие-либо простые соотношения между последовательностями вершин при обходе по таким правилам и по правилам, приведенным в тексте?

4.14. Определите структуру данных для представления n-арных деревьев. Напишите процедуру для обхода таких деревьев и построения двоичных деревьев с теми же элементами. Предположим, что ключи элементов занимают k слов, а каждая ссылка — одно слово. Каков выигрыш в памяти при использовании вместо n-арных деревьев двоичных?

4.15. Предположим, при построении деревьев используется определение типа, аналогичное приведенному в упр. 4.1:

```
RECTYPE Tree = RECORD x: INTEGER;
                      left, right: Tree
               END.
```

Напишите процедуру для обнаружения элемента с заданным ключом и выполнения над ним операции P.

4.16. В некоторой файловой системе справочник файлов организован в виде упорядоченного двоичного дерева. Каждой вершине соответствует некоторый файл, здесь содержится имя файла и, кроме всего прочего, дата последнего обращения к нему, закодированная целым числом. Напишите программу, которая обходит дерево и удаляет все файлы, последнее обращение к которым происходило до некоторой определенной даты.

4.17. В некоторой древовидной структуре опытным путем измеряется частота обращения к каждому из элементов. Для этого с любым из элементов связан счетчик обращений. По прошествии определенного времени дерево реорганизуется, для этого оно просматривается и с помощью программы 4.4 строится новое дерево, в которое ключи включаются в порядке убывания счетчиков частот обращения. Напишите программы, выполняющие такую реорганизацию. Будет ли средняя длина пути в новом дереве равна, больше или даже значительно больше длины пути в оптимальном дереве?

4.18. Описанный в разд. 4.5 метод анализа алгоритма включения в дерево можно использовать и для вычисления средних значений для числа сравнений C_n и числа пересылок M_n (обменов), выполняемых при быстрой сортировке (прогр. 2.10) массива из n элементов, считая, что все n! перестановок для n ключей 1, 2, ..., n равновероятны. Найдите аналогию и определите C_n и M_n.

4.19. Нарисуйте сбалансированное дерево с 12 вершинами, имеющее максимальную высоту для всех (сбалансированных) деревьев с 12 вершинами. В какой последовательности надо включать вершины, чтобы процедура (4.63) построила такое дерево?

4.20. Найдите такую последовательность из n элементов (включаемых), чтобы процедура (4.63) выполнила каждое из четырех действий по балансировке (LL,LR,RR,RL) по крайней мере один раз. Какова минимальная длина такой последовательности?

4.21. Найдите такое сбалансированное дерево с ключами 1 ... n и такую последовательность этих ключей, чтобы процедура исключения (4.64) ключей этой последовательности по крайней мере по одному разу выполнила каждую из четырех подпрограмм балансировки. Какова минимальная длина такой последовательности?

4.22. Какова средняя длина пути в дереве Фибоначчи?

4.23. Напишите программу формирования почти оптимального дерева, положив в ее основу алгоритм выбора в качестве корня центроида (4.78).

4.24. Предположим, в пустое Б-дерево включаются ключи 1, 2, 3, ... (прог. 4.7). Какие ключи приведут к расщеплению страниц? Какие ключи приведут к увеличению высоты дерева? Если в том же порядке исключать ключи, то какие вызовут слияние страниц, а какие уменьшение высоты? Ответьте на эти вопросы, считая, что а) используется схема с балансировкой (прогр. 4.7) и b) используется схема без балансировки (при нехватке берется один элемент с соседней страницы).

4.25. Напишите программу поиска, включения и исключения ключей из двоичных Б-деревьев. Используйте определение типа вершины (4.84). Схема включения приведена на рис. 4.51.

4.26. Найдите такую последовательность включения ключей, при которой, если начать с пустого симметричного Б-дерева, процедура (4.87) выполнит все четыре действия по балансировке (LL,LR,RR,RL) по крайней мере по одному разу. Какова самая маленькая такая последовательность?

4.27. Напишите процедуру для исключения элементов из симметричного Б-дерева. Найдите дерево и некоторую короткую последовательность исключений, приводящие по крайней мере к однократному появлению каждой из четырех ситуаций повторной балансировки.

4.28. Напишите определение структуры данных и процедуры для включения и исключения элемента из дерева поиска с приоритетом. Процедуры должны основываться на инвариантах (4.90). Сравните их производительность с производительностью для деревьев поиска с «корневым» приоритетом.

4.29. Постройте модуль, содержащий процедуры, работающие с деревьями поиска с «корневым» приоритетом:
- включение точки с координатами x, y;
- нумерации всех точек внутри заданного квадранта;
- поиска точки с минимальной x-координатой в заданном квадранте;
- поиска точки с максимальной y-координатой в заданном квадранте;
- поиска всех точек, лежащих внутри (пересекающихся) квадрантов.

ЛИТЕРАТУРА

[4.1] Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации. — Доклады АН СССР, 146, 1962, с. 263—266.

[4.2] Bayer R., McCreight E. M. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1, No. 3, (1972), 173—189.

[4.3] Bayer R., McCreight E. M. Binary B-trees for Virtual Memory. Proc. 1971 ACM SIGFIDET Workshop, San Diego, Nov. 1971, 219—235.

[4.4] Bayer R., 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., 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., Gotlieb C. C. A Top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees. In *Graph Theory and Computing*. (New York: Academic Press, 1972), 303—323.
[4.8] Comer D. The Ubiquitous B-Tree. *ACM Comp. Surveys*, **11**, No. 2, (June 1979), 121—137.
[4.9] Vuillemin J. A Unifying look at data structures. *Comm. ACM*, **23**, No. 4, (April 1980), 229—239.
[4.10] McCreight E. M. Priority search trees. *SIAM J. of Comp.* (May 1985).

5. ПРЕОБРАЗОВАНИЯ КЛЮЧЕЙ (РАССТАНОВКА)

5.1. ВВЕДЕНИЕ

Главный вопрос, так долго обсуждавшийся в гл. 4, формулировался так: задано множество элементов, характеризуемых некоторым ключом (в множестве ключей задано отношение порядка); как надо организовать упомянутое множество, чтобы поиск элемента с заданным ключом потребовал как можно меньше затрат? Ясно, что доступ к каждому элементу, расположенному в памяти машины, в конце концов идет через определенный адрес памяти. Следовательно, сформулированная задача фактически сводится к задаче определения подходящего отображения (H) ключей (K) в адреса (A):

$$H : K \to A$$

В гл. 4 такое отображение строилось в виде всяких алгоритмов поиска по спискам и деревьям, основанных на различных способах организации самих данных. Сейчас мы опишем еще один подход; он и проще, и во многих случаях гораздо эффективнее. Конечно, ему присущи и некоторые недостатки, но об этом мы поговорим позже.

В нашем методе данные организованы как обычный массив. Поэтому H — отображение, преобразующее ключи в индексы массива, отсюда и появилось название *«преобразование ключей»*, обычно даваемое этому методу. Следует заметить, что у нас нет никакой нужды обращаться к каким-либо процедурам динамического размещения, ведь массив — это один из основных, статических объектов. Метод преобразования ключей часто используется для таких задач, где можно с успехом воспользоваться и деревьями.

Основная трудность, связанная с преобразованием ключей, заключается в том, что множество возможных значений значительно шире множества допустимых адресов памяти (индексов массива). Возьмем в качестве примера имена, включающие до 16 букв и представляющие собой ключи, идентифицирующие отдельные индивиды из множества в тысячу персон. Следовательно, мы имеем дело с 26^{16} возможными ключами, которые нужно отобразить в 10^3 возможных индексов. Поэтому функция H будет функцией класса «много значений в одно значение». Если дан некоторый ключ k, то первый шаг операции поиска — вычисление связанного с ним индекса $h = H(k)$, а второй (совершенно необходимый) — проверка, действительно ли h идентифицирует в массиве (таблице) T элемент с ключом k, т. е. верно ли отношение $T[H(k)].key = k$. Отсюда мы сразу же сталкиваемся с такими двумя вопросами:

1. Какого вида функцию H нужно использовать?
2. Что делать в ситуации, когда H не указывает на местоположение желаемого элемента?

Ответ на второй вопрос: нужно использовать какой-нибудь метод, указывающий альтернативное местоположение, скажем индекс h′, а если и там нет нужного элемента, — третий индекс h″ и т. д. Ситуация, когда в указанном месте нет элемента с заданным ключом, называется *конфликтом*, а задача формирования альтернативного индекса — *разрешением конфликта*. В последующих разделах мы рассмотрим, как выбирать функцию преобразования и возможные методы разрешения конфликтов.

5.2. ВЫБОР ФУНКЦИИ ПРЕОБРАЗОВАНИЯ

Само собой разумеется, что любая хорошая функция преобразования должна как можно равномернее распределять ключи по всему диапазону значений индекса. Если это требование удовлетворяется, то других ограничений уже нет, и даже хорошо, если преобразование будет выглядеть как совершенно случайное. Такая особенность объясняет несколько ненаучное название этого метода — «перемалывание»

(хэширование*)), т. е. дробление аргумента, превращение его в какое-то «месиво». Функция же H будет называться «функцией расстановки». Ясно, что H должно вычисляться достаточно эффективно, т. е. состоять из очень небольшого числа основных арифметических операций.

Представим себе, что есть функция преобразования ORD(k), обозначающая порядковый номер ключа k в множестве всех возможных ключей. Кроме того, будем считать, что индекс массива i лежит в диапазоне 0 ... N — 1, где N — размер массива. В этом случае ясно, что нужно выбрать следующую функцию:

$$H(k) = ORD(k) \text{ MOD } N \qquad (5.1)$$

Для нее характерно равномерное отображение значений ключа на весь диапазон изменения индексов, поэтому ее кладут в основу большинства преобразований ключей. Кроме того, при N, равном степени двух, эта функция эффективно вычисляется. Однако если ключ представляет собой последовательность букв, то именно от такой функции и следует отказаться. Дело в том, что в этом случае допущение о равновероятности всех ключей ошибочно. В результате слова, отличающиеся только несколькими символами, с большой вероятностью будут отображаться в один и тот же индекс, что приведет к очень неравномерному распределению. Поэтому на практике рекомендуется в качестве N брать простое число [4.7]. Следствием такого выбора будет необходимость использования полной операции деления, которую уже нельзя заменить выделением нескольких двоичных цифр. Однако на большинстве современных машин, имеющих встроенную операцию деления, это не будет серьезным препятствием.

Часто употребляется функция расстановки, «состоящая» из логических операций, скажем «исключающего или», примененных к некоторой части клю-

*) В русской литературе, к сожалению, очень часто употребляется это слово, хотя еще в 1956 г. будущий академик А. П. Ершов начал использовать русский термин «расстановка», которым мы и будем пользоваться. — *Прим. перев.*

ча, рассматриваемого как последовательность двоичных цифр. На некоторых машинах такие операции могут *) выполняться быстрее, чем деление, но иногда они распределяют ключи по диапазону изменения далеко не равномерно. Поэтому мы не будем детально в них разбираться.

5.3. РАЗРЕШЕНИЕ КОНФЛИКТОВ

Если обнаруживается, что строка таблицы, соответствующая заданному ключу, не содержит желаемого элемента, то, значит, произошел конфликт, т. е. два элемента имеют такие ключи, которые отображаются в один и тот же индекс. В этой ситуации нужна вторая попытка с индексом, вполне определенным образом получаемым из того же заданного ключа. Существует несколько методов формирования вторичного индекса. Очевидный прием — связывать вместе все строки с идентичным первичным индексом H(k). превращая их в связанный список. Такой прием называется *прямым связыванием* (direct chaining). Элементы получающегося списка могут либо помещаться в основную таблицу, либо нет; в этом случае память, где они размещаются обычно называется *областью переполнения.* Недостаток такого приема в том, что нужно следить за такими вторичными списками и в каждой строке отводить место для ссылки (или индекса) на соответствующий список конфликтующих элементов.

Другой прием разрешения конфликтов состоит в том, что мы совсем отказываемся от ссылок и вместо этого просматриваем другие строки той же таблицы — до тех пор, пока не обнаружим желаемый элемент или же пустую строку. В последнем случае мы считаем, что указанного ключа в таблице нет. Такой прием называется *открытой адресацией* [5.1]. Естественно, что во второй попытке последовательность индексов должна быть всегда одной и той же

*) Это не совсем так: логические операции *всегда* будут выполняться быстрее деления (если это не деление на степень двойки). — *Прим. перев.*

для любого заданного ключа. В этом случае алгоритм просмотра строится по такой схеме:

```
h := H(k); i := 0;
REPEAT                                                    (5.2)
  IF T[h].key = k THEN элемент найден
  ELSIF T[h].key = free THEN элемента в таблице нет
  ELSE (* конфликт *)
    i := i+1; h := H(k) + G(i)
  END
UNTIL либо найден, либо нет в таблице (либо она полна)
```

В литературе предлагались самые разные функции G(i) для разрешения конфликтов. Приведенный в работе Морриса [5.2] (1968) обзор стимулировал активную деятельность в этом направлении. Самый простой прием — посмотреть следующую строку таблицы (будем считать ее круговой), и так до тех пор, пока либо будет найден элемент с указанным ключом, либо встретится пустая строка. Следовательно, в этом случае G(i) = i, а индексы h_i, употребляемые при последующих попытках, таковы:

$$h_0 = H(k)$$
$$h_i = (h_0 + i) \text{ MOD } N, \qquad i = 1 \ldots N-1 \tag{5.3}$$

Такой прием называется *линейными пробами*, его недостаток заключается в том, что строки имеют тенденцию группироваться вокруг первичных ключей (т. е. ключей, для которых при включении конфликта не возникало). Конечно, хотелось бы выбрать такую функцию G, которая вновь равномерно рассеивала бы ключи по оставшимся строкам. Однако на практике это приводит к слишком большим затратам, потому предпочтительнее некоторые компромиссные методы; будучи достаточно простыми с точки зрения вычислений, они все же лучше линейной функции. Один из них — использование квадратичной функции, в этом случае последовательность пробуемых индексов такова:

$$h_0 = H(k)$$
$$h_i = (h_0 + i^2) \text{ MOD } N, \quad i > 0 \tag{5.4}$$

Обратите внимание, что если воспользоваться рекуррентными соотношениями (5.5) для $h_i = i^2$ и $d_i = 2i + 1$ при $h_0 = 0$ и $d_0 = 1$, то при вычислении очередного индекса можно обойтись без операции возведения в квадрат.

$$\begin{aligned} h_{i+1} &= h_i + d_i \\ d_{i+1} &= d_i + 2 \quad (i>0) \end{aligned} \tag{5.5}$$

Такой прием называется *квадратичными пробами*; существенно, что он позволяет избежать группирования, присущего линейным пробам, не приводя практически к дополнительным вычислениям. Небольшой же его недостаток заключается в том, что при поиске пробуются не все строки таблицы, т. е. при включении элемента может не найтись свободного места, хотя на самом деле оно есть. Если размер N — простое число, то при квадратичных пробах просматривается по крайней мере половина таблицы. Такое утверждение можно вывести из следующих рассуждений. Если i-я и j-я пробы приводят к одной и той же строке таблицы, то справедливо равенство

$$i^2 \text{ MOD } N = j^2 \text{ MOD } N$$

или

$$(i^2 - j^2) \equiv 0 \pmod N$$

Разлагая разность на два множителя, мы получаем

$$(i + j)(i - j) \equiv 0 \pmod N$$

Поскольку $i \neq j$, то либо i, либо j должны быть больше N/2, чтобы было справедливо равенство $i + j = cN$, где c — некоторое целое число. На практике упомянутый недостаток не столь существен, так как N/2 вторичных попыток при разрешении конфликтов встречаются очень редко, главным образом в тех случаях, когда таблица почти заполнена.

В качестве примера использования метода рассеянных таблиц приведем переписанную программу (прогр. 4.5) генератора перекрестных ссылок — прогр. 5.1. Принципиальное различие заключается в процедуре search и замене ссылочного типа WPtr

```
MODULE XRef;
FROM InOut IMPORT OpenInput, OpenOutput, CloseInput, CloseOutput,
   Read, Done, EOL, Write, WriteCard, WriteString, WriteLn;
FROM Storage IMPORT ALLOCATE;

CONST P = 997; (* простое число, размер таблицы *)
   BufLeng = 10000; WordLeng = 16;
   free = 0;

TYPE WordInx = [0 .. P-1];
   ItemPtr = POINTER TO Item;

   Word = RECORD key: CARDINAL;
        first, last: ItemPtr;
      END;

   Item = RECORD lno: CARDINAL;
        next: ItemPtr
      END;

VAR k0, k1, line: CARDINAL;
   ch: CHAR;
   T: ARRAY [0 .. P-1] OF Word; (* массив расстановки *)
   buffer: ARRAY [0 .. BufLeng-1] OF CHAR;

PROCEDURE PrintWord(k: CARDINAL);
  VAR lim: CARDINAL;
BEGIN lim := k + WordLeng;
  WHILE buffer[k] > 0C DO Write(buffer[k]); k := k+1 END;
  WHILE k < lim DO Write(" "); k := k+1 END
END PrintWord;

PROCEDURE PrintTable;
  VAR i, k, m: CARDINAL; item: ItemPtr;
BEGIN
  FOR k := 0 TO P-1 DO
    IF T[k].key # free THEN
      PrintWord(T[k].key); item := T[k].first; m := 0;
      REPEAT
        IF m = 8 THEN
          WriteLn; m := 0;
          FOR i := 1 TO WordLeng DO Write(" ") END
        END;
        m := m+1; WriteCard(item^.lno, 6); item := item^.next
```

```
    UNTIL item = NIL;
    WriteLn;
  END
 END
END PrintTable;

PROCEDURE Diff(i, j: CARDINAL): INTEGER;
BEGIN
 LOOP
  IF buffer[i] # buffer[j] THEN
   RETURN INTEGER(ORD(buffer[i])) - INTEGER(ORD(buffer[j]))
  ELSIF buffer[i] = 0C THEN RETURN 0
  END;
  i := i+1; j := j+1
 END
END Diff;

PROCEDURE search;
 VAR i, h, d: CARDINAL; found: BOOLEAN;
  ch: CHAR; x: ItemPtr;
 (* глобальные переменные: T, buffer, k0, k1 *)
BEGIN (* вычисление индекса h слова, начинающегося с buffer [k0] *)
 i := k0; h := 0; ch := buffer[i];
 WHILE ch > 0C DO
  h := (256*h + ORD(ch)) MOD P; i := i+1; ch := buffer[i]
 END;
 ALLOCATE(x, SIZE(Item)); x↑.lno := line; x↑.next := NIL;
 d := 1; found := FALSE;
 REPEAT
  IF Diff(T[h].key, k0) = 0 THEN (* совпадение *)
   found := TRUE; T[h].last↑.next := x; T[h].last := x
  ELSIF T[h].key = free THEN (* новый элемент *)
   WITH T[h] DO
    key := k0; first := x; last := x
   END;
   found := TRUE; k0 := k1
  ELSE (* конфликт *) h := h+d; d := d+2;
   IF h >= P THEN h := h-P END;
   IF d = P THEN WriteString(" Table overflow"); HALT END
  END
 UNTIL found
END search;
```

```
PROCEDURE GetWord;
BEGIN k1 := k0;
  REPEAT Write(ch); buffer[k1] := ch; k1 := k1 + 1; Read(ch)
  UNTIL (ch < "0") OR (ch > "9") & (CAP(ch) < "A")
    OR (CAP(ch) > "Z");
  buffer[k1] := 0C; k1 := k1 + 1; (* ограничитель *)
  search
END GetWord;
BEGIN k0 := 1; line := 0,
  FOR k1 := 0 TO P-1 DO T[k1].key := free END ;
  OpenInput("TEXT"); OpenOutput("XREF");
  WriteCard(0, 6); Write(" "); Read(ch);
  WHILE Done DO
    CASE ch OF
      0C .. 35C:  Read(ch) |
      36C .. 37C: WriteLn; Read(ch); line := line + 1;
                  WriteCard(line, 6); Write(" ") |
      " " .. "@": Write(ch); Read(ch) |
      "A" .. "Z": GetWord |
      "[" .. "`": Write(ch); Read(ch) |
      "a" .. "z": GetWord |
      "{" .. "~": Write(ch); Read(ch)
    END
  END ;
  WriteLn; WriteLn; CloseInput;
  PrintTable; CloseOutput
END XRef.
```

Прогр. 5.1. Построение таблицы перекрестных ссылок с использованием функции расстановки.

на таблицу слов — T. Функция расстановки H — это значение по модулю размера таблицы; для разрешения конфликтов были выбраны квадратичные пробы. Заметим, что для эффективной работы существенно, чтобы размер таблицы был простым числом.

Хотя преобразование ключей в данном случае крайне эффективно, фактически оно работает даже лучше, чем методы, ориентированные на поиск по деревьям, тем не менее и ему присущи некоторые недостатки. После просмотра всего текста и накопления слов, по-видимому, у нас возникает желание напечатать эти слова в алфавитном порядке. Если мы пользовались деревьями, то эта задача решается естественным образом: ведь в основу было положено

упорядоченное дерево поиска *). Однако при преобразовании ключей это сделать не так просто, именно здесь полностью и проявляется буквальное значение слова «перемалывание». Оказывается, что перед печатью не только нужно проводить какую-либо сортировку (в прогр. 5.1 мы этот процесс опустили), но желательно сохранять последовательность включения ключей, связывая их явно в список. Поэтому очень высокая в случаях простого поиска производительность метода расстановок будет снижена за счет дополнительных операций, необходимых для выполнения всех условий задачи формирования упорядоченного указателя перекрестных ссылок.

5.4. АНАЛИЗ МЕТОДА ПРЕОБРАЗОВАНИЯ КЛЮЧЕЙ

Ясно, что в наиболее неблагоприятных случаях производительность метода преобразования ключей при включении элементов и поиске будет удручающе плохой. Ведь вполне возможно, что аргумент поиска окажется таким, что все вторичные пробы будут приходиться на занятые места, а нужные (свободные) будут всякий раз пропускаться. Фактически, если человек использует функции расстановки, он должен быть полностью уверен в корректности применения законов теории вероятностей **). Мы же лишь желаем убедиться, что в среднем число проб будет небольшим. Следующие вероятностные рассуждения показывают, что оно будет даже очень небольшим.

Как и раньше, будем предполагать, что все допустимые ключи равновероятны, и функция расстановки равномерно отображает их на весь диапазон индексов таблицы. Представим себе, что есть некоторый ключ и его нужно вставить в таблицу размером n, где уже содержится k элементов. Вероятность

*) Следует иметь в виду, что в общепринятых кодировках ISO буквы русского языка упорядочены не по алфавиту. — *Прим. перев.*

**) В частности, это означает, что он должен быть уверенным в равномерном распределении исходных ключей. Обычно именно это условие и нарушается. — *Прим. перев.*

сразу же попасть в свободное место равна $(n-k)/n$. Это же и вероятность (p_1) того, что нужно всего одно сравнение. Вероятность того, что понадобится точно одна дополнительная попытка, равна вероятности конфликта при первой попытке, умноженной на вероятность попадания в свободное место при второй. Таким образом, мы получаем общую схему вычисления p_i — вероятности того, что требуется точно i попыток:

$$\begin{aligned} p_1 &= (n-k)/n \\ p_2 &= (k/n)*(n-k)/(n-1) \\ p_3 &= (k/n)*(k-1)/(n-1)*(n-k)/(n-2) \\ &\\ p_i &= (k/n)*(k-1)/(n-1)*(k-2)/(n-2)*\ldots*(n-k)/(n-i+1) \end{aligned} \tag{5.6}$$

Поэтому среднее число попыток E при включении $k+1$-го элемента равно:

$$\begin{aligned} E_{k+1} &= Si: 1 \le i \le k+1 : i*p_i \\ &= 1*(n-k)/n + 2*(k/n)*(n-k)/(n-1) + \ldots + \\ &\quad (k+1)*(k/n)*(k-1)/(n-1)*(k-2)/(n-2)*\ldots*1/(n-k+1) \\ &= (n+1)/(n-k+1) \end{aligned} \tag{5.7}$$

Поскольку число проб, необходимых для размещения некоторого ключа, совпадает с числом проб, затрачиваемых на его поиск, то результат (5.3) можно использовать и для вычисления среднего числа (E) попыток, необходимых для доступа в таблице к некоторому случайному ключу. Обозначим размер таблицы через n, а через m — число ключей, фактически помещенных в таблицу. Тогда:

$$\begin{aligned} E &= (Sk: 1 \le k \le m : E_k)/m \\ &= (n+1)*(Sk: 1 \le k \le m : 1/(n-k+2))/m \\ &= (n+1)*(H_{n+1} - H_{n-m+1})/m \end{aligned} \tag{5.8}$$

где H — гармоническая функция. Ее можно аппроксимировать таким выражением: $H_n = \ln(n) + g$, где g — эйлерова константа. Если, кроме того, еще подставить $a = m/(n+1)$, то

$$\begin{aligned} E &= (\ln(n+1) - \ln(n-m+1))/a = \ln((n+1)/(n-m+1))/a \\ &= -\ln(1-a)/a \end{aligned} \tag{5.9}$$

Таблица 5.1. Среднее число попыток в зависимости от коэффициента заполнения

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

Значение a, приблизительно соответствующее отношению занятой памяти ко всей имеющейся, называется *коэффициентом заполнения*; $a = 0$ соответствует пустой таблице, а значение $a = n/(n+1)$ — полностью заполненной. Среднее число (E) попыток при поиске или включении случайно выбранного ключа в зависимости от коэффициента заполнения приводится в табл. 5.1. Приведенные числа действительно вызывают удивление и объясняют исключительно высокую производительность метода преобразования ключей. Ведь даже при 90 % заполнения в среднем необходимо лишь 2,56 попыток, чтобы обнаружить некоторый ключ или найти свободное место. Заметим, в частности, что это число зависит лишь от коэффициента заполнения, а не от абсолютного числа присутствующих ключей.

Наш анализ основывался на предположении, что при разрешении конфликтов ключи равномерно отображаются на оставшуюся свободную память. Реальные же методы разрешения дают несколько худшую производительность. Детальный анализ метода линейных проб дает такое значение для среднего числа попыток $E = (1 - a/2)/(1 - a)$.

Некоторые значения E(a) перечислены в табл. 5.2. Результаты, полученные даже для такого «убогого» приема разрешения конфликтов, настолько хороши, что соблазнительно рассматривать метод преобразования ключей (расстановку) как панацею от всех бед. Ведь производительность его выше производительности методов, основанных на использовании де-

Таблица 5.2. Среднее число попыток при использовании 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

ревьев с самой изощренной организацией, о которых мы говорили (если сравнивать количество шагов, необходимых для поиска или включения). Поэтому важно здесь явно указать его недостатки, хотя они и так очевидны при беспристрастном рассмотрении.

Конечно, основной недостаток по сравнению с другими методами, основанными на динамическом размещении,— это фиксированный размер таблиц и невозможность настройки на реальные требования *). Поэтому, если мы хотим избежать неэкономного использования памяти или низкой производительности (или даже переполнения таблицы), нужно научиться достаточно хорошо заранее оценивать число классифицируемых элементов. Если же число элементов известно точно, а это крайне редкий случай, то желание получить хорошую производительность приводит к тому, что размер таблицы приходится слегка увеличивать (скажем, на 10 %).

Второй основной недостаток методов, основанных на расстановках, становится сразу же очевидным, как только заходит речь о том, что ключи нужно не только вставлять в таблицу и отыскивать, но и исключать из таблицы. Изъятие строки из расстановочной таблицы — вещь крайне трудная, если только не используются явные списки в отдельной области переполнения. Таким образом, надо быть справедли-

*) Ссылки на экономичность динамических методов распределения памяти обманчивы: эта память не появляется из ничего, она берется из свободной памяти фиксированного размера. Такая память может и исчерпываться, и, самое главное, для ее организации требуется много времени, иногда настолько много, что все оценки самих методов, в которых идет динамическое размещение, теряют всякий смысл. — *Прим. перев.*

выми и сказать, что организация памяти в виде деревьев продолжает оставаться привлекательной и даже желательной, когда объемы данных неизвестны, сильно варьируются и иногда даже уменьшаются.

УПРАЖНЕНИЯ

5.1. Если связанная с каждым из ключей информация относительно велика (по сравнению с самим ключом), то ее не следует хранить в таблице вместе с ключами. Объясните почему и предложите схему для хранения такой информации.

5.2. С проблемой скопления можно бороться так. Вместо списков переполнения можно использовать деревья переполнения, т. е. помещать в некоторое дерево те ключи, которые вызывают конфликт. Следовательно, каждый элемент в таблице можно считать корнем дерева (возможно, пустого). Сравните производительность (ожидаемую) с производительностью метода открытой адресации.

5.3. Предложите схему, допускающую включения и исключения из таблицы расстановки и основанную на квадратичных смещениях при разрешении конфликтов. Сравните экспериментально такую схему со схемой, использующей простую организацию информации в виде двоичного дерева. При сравнении используйте случайные последовательности включаемых и исключаемых ключей.

5.4. Основной недостаток метода расстановок заключается в том, что при неизвестном числе элементов размер таблицы должен быть фиксированным. Предположим, что в вашей системе существует механизм динамического распределения памяти, обеспечивающей вас всегда нужной памятью. В этом случае, если таблица H уже заполнена, формируется большая таблица H′ и все ключи из H трансформируются для H′, а память, занимавшаяся таблицей H, возвращается администратору системы. Такой процесс называется *повторной расстановкой*. Напишите программу повторной расстановки для H размером n.

5.5. Часто ключами являются не целые числа, а последовательности букв. Такие слова сильно различаются по длине, и их невозможно экономно и удобно располагать в полях фиксированного размера. Напишите программу для метода расстановок, работающую с ключами переменного размера.

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. and Dev.*, 1, (1957), 130—146.

[5.4] Schay G., Spruth W. Analysis of a File Addressing Method. *Comm. ACM*, 5, No. 8, (1962), 459—462.

ПРИЛОЖЕНИЯ

1. 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	stx	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

Символы-ограничители

bs возврат
ht горизонтальная табуляция
lf переход на новую строку
vt вертикальная табуляция
ff переход на новую страницу
cr возврат каретки

Символы-разделители

fs разделитель файлов
gs разделитель групп
rs разделитель записей
us разделитель единиц

2. СИНТАКСИС МОДУЛЫ-2

1. Идент = буква {буква | Цифра}.
2. Число = Целое | Вещественное.
3. Целое = Цифра {Цифра} |
   Цифра8 {Цифра8} ("B" | "C") |
   Цифра {Цифра16} "H".
4. Вещественное = Цифра {Цифра} "." {Цифра}
   [Порядок].
5. Порядок = "E" ["+" | "—"] Цифра {Цифра}.
6. Цифра 16 = Цифра | "A" | "B" | "C" | "D" | "E" | "F".
7. Цифра = Цифра8 | "8" | "9".
8. Цифра8 = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
9. Строка = "'" {символ} "'" | '"' {символ} '"'.
10. Полный идент = Идент {"." Идент}.
11. Описание константы = Идент "=" Константное
    выражение.
12. Константное выражение = Выражение.
13. Описание типа = Идент "=" Тип.
14. Тип = Простой тип |
    Массивовый тип |
    Записной тип |
    Множественный тип |
    Ссылочный тип |
    Процедурный тип.
15. Простой тип = Полный идент |
    Перечисление |
    Диапазон.
16. Перечисление = "(" Список идент ")".
17. Список идент = Идент {"," Идент}.
18. Диапазон = [Полный идент] "[" Константное выражение ".." Константное выражение "]".
19. Массивовый тип = ARRAY Простой тип {"," Простой тип} OF Тип.
20. Записной тип = RECORD Несколько списков полей END.
21. Несколько списков полей = Список полей {";" Список полей}.
22. Список полей = [Список идент ":" Тип | CASE
    [Идент] ":" Полный идент OF
    Вариант записи {"|"
    Вариант записи} [ELSE

Несколько списков полей]END]
23. Вариант записи = [Список меток варианта ":" Несколько списков полей].
24. Список меток варианта = Метки варианта {"," Метки варианта}.
25. Метки варианта = Константное выражение ["..." Константное выражение].
26. Множественный тип = SET OF Простой тип.
27. Ссылочный тип = POINTER TO Тип.
28. Процедурный тип = PROCEDURE [Список формальных типов].
29. Список формальных типов = "(" [[VAR] Формальный тип {"," [VAR] Формальный тип}] ")" [":" Полный идент].
30. Описание переменной = Список идент ":" Тип.
31. Переменная = Полный идент {"." Идент | "["Список выражений"]" | "↑" }
32. Список выражений = Выражение {"," Выражение}.
33. Выражение = Простое выражение [Отношение Простое выражение].
34. Отношение = "=" | "#" | "<" | "<=" | ">" | ">=" | IN
35. Простое выражение = ["+" | "−"] Терм {Операция сложения Терм}.
36. Операция сложения = "+" | "−" | OR.
37. Терм = Фактор {Операция умножения Фактор }.
38. Операция умножения = "*" | "/" | DIV | REM | MOD | AND
39. Фактор = Число | Строка | Множество | Переменная [Фактические параметры] | "("Выражение")" | NOT Фактор.
40. Множество = [Полный идент] "{" [Элемент {"," Элемент}] "}".
41. Элемент = Константное выражение ["..." Константное выражение].
42. Фактические параметры = "(" [Список выражений] ")".
43. Оператор = Присваивание | Обращение к процедуре | Условный оператор |

Оператор варианта|
Цикл с предусловием|
Цикл с постусловием|
Цикл с шагом|
Бесконечный цикл|
Оператор присоединения|
EXIT
RETURN Выражение.

44. Присваивание = Переменная ":=" Выражение.
45. Обращение к процедуре = Переменная [Фактические параметры].
46. Несколько операторов = Оператор {";" Оператор}.
47. Условный оператор = IF Выражение THEN Несколько операторов {ELSIF Выражение THEN Несколько операторов} [ELSE. Несколько операторов] END.
48. Оператор варианта = CASE Выражение OF Вариант {"|" Вариант} [ELSE Несколько операторов] END.
49. Вариант = [Список меток варианта ":" Несколько операторов].
50. Цикл с предусловием = WHILE Выражение DO Несколько операторов END.
51. Цикл с постусловием = REPEAT Несколько операторов UNTIL Выражение.
52. Цикл с шагом = FOR Идент ":=" Выражение TO Выражение (BY Константное выражение] DO Несколько операторов END.
53. Бесконечный цикл = LOOP Несколько операторов END.
54. Оператор присоединения = WITH Переменная DO Несколько операторов END.
55. Описание процедуры = заголовок процедуры ";" (Блок Идент| FORWARD).
56. Заголовок процедуры = PROCEDURE Идент [Формальные параметры].
57. Блок = {Описание} (BEGIN

Несколько операторов] END.
58. Описание = CONST {Описание константы ";"} |
TYPE {Описание типа ";"} |
VAR {Описание переменной ";"} |
Описание процедуры ";" |
Описание модуля ";".
59. Формальные параметры = "(" [Спецификация параметров {";" Спецификация параметров}] ")" [":" Полный идент].
60. Спецификация параметров = [VAR] Список идент ":" Формальный тип.
61. Формальный тип = [ARRAY OF] Полный идент.
62. Описание модуля = MODULE Идент [Приоритет] ";" {Импорт} [Экспорт] Блок Идент.
63. Приоритет = "[" Константное выражение "]".
64. Экспорт = EXPORT [QUALIFIED] Список идент Список идент ";".
65. Импорт = [FROM Идент] IMPORT Список идент Список идент ";".
66. Модуль определений = DEFINITION MODULE Идент ";" {Импорт} {Определение} END Идент".".
67. Определение = CONST {Описание константы ";"} |
TYPE {Идент ["=" Тип] ";"} |
VAR {Описание переменной ";"} |
Заголовок процедуры ";".
68. Модуль = MODULE Идент [Приоритет] ";" {Импорт} Блок Идент ".".
69. Единица трансляции = Модуль определений | [IMPLEMENTATION] Модуль.

ОГЛАВЛЕНИЕ