Л. А. ПАВЛОВ, Н. В. ПЕРВОВА

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Учебник

Издание второе,
исправленное и дополненное

•САНКТ-ПЕТЕРБУРГ•
•МОСКВА•
•КРАСНОДАР•
2020

УДК 004.657
ББК 32.81я73

П 12 **Павлов Л. А.** Структуры и алгоритмы обработки данных : учебник / Л. А. Павлов, Н. В. Первова. — 2-е изд., испр. и доп. — Санкт-Петербург : Лань, 2020. — 256 с. : ил. — (Учебники для вузов. Специальная литература). — Текст : непосредственный.

ISBN 978-5-8114-4881-4

Рассмотрены математические основы анализа вычислительной сложности алгоритмов, типовые структуры данных для представления множеств: массивы и динамические списковые структуры, стеки, очереди и деревья. Приведены методы решения комбинаторных задач и основные способы сокращения перебора, задачи поиска, сортировки и алгоритмы на графах.

Для студентов факультета информатики и вычислительной техники по направлению подготовки бакалавров «Информатика и вычислительная техника», а также других направлений и профилей, связанных с разработкой программного обеспечения.

УДК 004.657
ББК 32.81я73

Рецензенты:

Г. П. ОХОТКИН — доктор технических наук, профессор, декан факультета радиоэлектроники и автоматики Чувашского государственного университета им. И. Н. Ульянова;

К. А. МОРДАСОВ — кандидат технических наук, заместитель генерального директора по развитию ООО «ИТ-Консалтинг».

Обложка
П. И. ПОЛЯКОВА

ПРЕДИСЛОВИЕ

Для разработки программ недостаточно просто знать какой-либо язык программирования. Процесс проектирования программ предполагает несколько этапов, начиная с постановки задачи и заканчивая получением эффективно и правильно работающей программы. Важнейшим начальным этапом является формализация поставленной задачи. На этом этапе строится математическая модель задачи с привлечением различных математических конструкций, таких как системы линейных или дифференциальных уравнений, множества, графы, матрицы и т. п. Выбор математических конструкций зависит от характера решаемой задачи.

После построения математической модели производится поиск методов решений исходной задачи в терминах этой модели. Для вычислительных задач метод решения формально описывается в форме *алгоритма*, который представляет собой конечную последовательность инструкций, имеющих четкий смысл и выполняемых с конечными вычислительными затратами за конечное время. Другими словами, алгоритм представляет собой формально описанную процедуру решения задачи, получающую некоторые исходные данные (*вход алгоритма*) и выдающую результат вычислений. Требования, предъявляемые к исходным данным и решению, определяются формулировкой задачи.

Алгоритм оперирует данными как исходными и результирующими, так и промежуточными, формируемыми самим алгоритмом для обеспечения его работы. Поэтому важным фактором при разработке алгоритмов является также выбор соответствующих структур данных. Часто от выбора структур данных зависит эффективность выполнения алгоритма.

В большинстве случаев поставленная перед разработчиком задача может иметь разные методы решения и, соответственно, различные алгоритмы решения. В связи с этим возникают вопросы оценки качества алгоритмов, сравнения характеристик различных алгоритмов и выбора на основе такого анализа наиболее эффективного алгоритма решения поставленной задачи.

В учебнике рассматриваются структуры данных, алгоритмы и методы их анализа, которые могут быть применены для разработки программ. Содержание пособия направлено на формиро-

вание соответствующих компетенций для направления подготовки бакалавров 09.03.01 «Информатика и вычислительная техника». Пособие может оказаться полезным и для других направлений и профилей, связанных с разработкой программного обеспечения.

В первой главе приводится псевдокод для записи алгоритмов, рассматриваются асимптотические обозначения, используемые для оценки времени выполнения алгоритмов, приводятся методы решения рекуррентных соотношений, описывающих время работы рекурсивных алгоритмов.

Во второй главе содержатся сведения о типовых структурах данных для представления множеств: последовательное (массивы) и связное (динамические списковые структуры) распределения, стеки, очереди и деревья. Для динамических структур приводятся алгоритмы операций включения и исключения элементов, для деревьев – стандартные способы систематического обхода вершин (прохождения деревьев). Рассматриваются такие характеристики деревьев, как их высота, уровни вершин, длины внутренних и внешних путей, которые могут быть полезны при использовании деревьев в качестве инструмента анализа алгоритмов.

В третьей главе изложены основные методы исчерпывающего поиска: поиск с возвратом (и его разновидности: метод ветвей и границ для решения оптимизационных задач и метод альфабета отсечений для игровых задач) и методы решета, применяемые для решения комбинаторных задач. Рассматриваются основные способы сокращения перебора с целью повышения эффективности поиска решений, а также способы программирования поиска с возвратом. Обсуждаются также вопросы, связанные с разработкой эвристических алгоритмов.

Четвертая глава посвящена методам и алгоритмам поиска (последовательный, логарифмический, хеширование). Рассматривается применение сбалансированных деревьев (АВЛ-деревья, красно-черные деревья) для организации динамических множеств, а также для внешнего поиска (Б-деревья). Обсуждаются вопросы, связанные с выбором хеш-функций, и методы разрешения коллизий, возникающих при поиске с хешированием.

Пятая глава знакомит с широко распространенными методами и алгоритмами внутренней сортировки: вставками (простые

вставки и сортировка Шелла), обменная (пузырьковая и быстрая), выбором (простой выбор и пирамидальная сортировка), цифровая распределяющая, перечислением, слиянием. Изложены основные идеи внешней сортировки, связанные с формированием исходных отрезков, их распределением и слиянием. Рассмотрены задачи вычисления порядковых статистик.

Шестая глава посвящена алгоритмам на графах. Рассмотрены способы представления графов и методы и алгоритмы анализа графов: поиск в глубину, ширину, построение остовных деревьев (поиском в глубину, ширину и минимальных остовных деревьев), определение связных, двусвязных и сильно связных компонент, топологическая сортировка и транзитивное замыкание орграфов, нахождение фундаментального множества циклов, определение кратчайших путей, построение эйлеровых циклов.

Каждая глава завершается упражнениями, часть из которых предназначена для закрепления изложенного материала, а часть требует изучения дополнительной литературы (или собственных умственных усилий) для их решения. Все упражнения, связанные с разработкой алгоритмов, рекомендуется сопровождать оценками времени их выполнения (хотя бы на уровне асимптотических оценок). Упражнения повышенной сложности отмечены звездочкой.

Все алгоритмы (как приведенные в пособии, так и разработанные в качестве упражнений) рекомендуется реализовывать на каком-либо языке программирования, исследовать их для различных размеров входных данных и сравнивать полученные экспериментальные оценки с аналитическими оценками.

В пособии не затронуты вопросы, связанные с реализацией алгоритмов и структур данных на параллельной вычислительной модели. Авторы считают, что для этого актуального направления исследований должно быть отдельное учебное издание, направленное на формирование компетенций в области параллельной обработки данных и параллельного программирования в среде многоядерных процессоров и многопроцессорных вычислительных систем.

Глава 1. АЛГОРИТМЫ И ИХ СЛОЖНОСТИ

1.1. Псевдокод для записи алгоритмов

В учебном пособии для записи алгоритмов используется *псевдокод*, в котором логика алгоритма строится при помощи управляющих конструкций (операторов) языка программирования Паскаль. При этом для более удобного восприятия алгоритмов (а также для большего абстрагирования от конкретного языка программирования) синтаксис некоторых операторов псевдокода отличается от синтаксиса соответствующих операторов языка Паскаль.

Кроме операторов в псевдокоде используются также такие конструкции языков программирования, как переменные, выражения, условия, процедуры и функции. Псевдокод не имеет фиксированного набора типов данных, поэтому переменные могут представлять произвольный тип (целый массив, строку, запись и т. д.). При этом описания типов в алгоритмах не приводятся, а тип данных какой-либо переменной и ее область действия становятся ясными из контекста.

Выражение и условие, представляющее собой любое выражение, принимающее значения **true** или **false**, записываются в виде традиционной математической конструкции, а не в соответствии с синтаксисом какого-либо языка программирования. В качестве выражения или условия может выступать также произвольная фраза на естественном языке, задающая правило вычисления значения выражения, например фраза «случайный элемент из множества S» является выражением, а фраза «x – простое число» – условием.

Процедуры (**procedure**) и функции (**function**) имеют тот же смысл, что и в языке Паскаль. Отличие заключается в том, что при определении процедуры или функции не указываются типы формальных параметров (а для функций – и тип самой функции), а также нет их разделения на параметры-переменные и параметры-значения. Эта информация становится ясной из контекста. Переменные, используемые в теле процедуры или функции, считаются локальными, если глобальность какой-либо переменной не оговорена особо или не следует из контекста.

Псевдокод использует следующие операторы:

1) оператор присваивания вида

переменная ← *выражение*

вычисляет значение выражения и присваивает его переменной. В ряде случаев для сокращения записи используется одновременное присваивание, например оператор *var*1 ← *var*2 ← *expr* эквивалентен последовательности двух операторов присваивания: *var*1 ← *expr* и *var*2 ← *expr*. Время выполнения оператора присваивания определяется временем вычисления выражения и временем самого присваивания;

2) условный оператор вида

if *условие* **then** *оператор* **else** *оператор*

или

if *условие* **then** *оператор*

имеет тот же смысл, что и в Паскале. Синтаксическая двусмысленность, возникающая при использовании второго варианта условного оператора, разрешается стандартным образом: **else**-часть всегда сопоставляется ближайшей предшествующей **then**-части, которой еще не сопоставлена **else**-часть. Время выполнения условного оператора определяется как сумма времени, необходимого на вычисление значения условия и его проверки, и времени выполнения оператора в **then**-части или оператора в **else**-части в зависимости от того, какой из них выполнялся;

3) оператор варианта вида

$$\textbf{case}\begin{cases} \textit{условие} : \textit{оператор}, \\ \quad\vdots \\ \textit{условие} : \textit{оператор}, \end{cases}$$

синтаксис которого существенно изменен по сравнению с синтаксисом соответствующего оператора в языке Паскаль для более удобного восприятия логики алгоритма. В данном операторе выполняется тот оператор, для которого выполняется условие. Время выполнения оператора варианта складывается из времени вычисления значения условия и его проверки и времени выполнения соответствующего оператора;

4) операторы цикла с предусловием вида

while *условие* **do** *оператор*

и постусловием вида

repeat *оператор* **until** *условие*

имеют тот же смысл, что и в языке Паскаль. Время выполнения оператора определяется как сумма времен всех исполняемых итераций цикла, каждая из которых состоит из времени вычисления и проверки условия выхода из цикла и времени выполнения оператора тела цикла. При этом следует учитывать, что в операторе **while** число вычислений и проверок условия выхода из цикла на единицу больше, чем число итераций, а в операторе **repeat** число вычислений и проверок условия выхода из цикла равно числу итераций;

5) оператор цикла с параметром вида

for *переменная* ← *выражение*1 **to** *выражение*2
by *выражение*3 **do** *оператор*.

Здесь *переменная* является параметром цикла; *выражение*1, *выражение*2 и *выражение*3 – соответственно начальное значение, конечное значение и размер шага изменения параметра цикла. Время выполнения оператора цикла **for** определяется аналогично времени выполнения оператора цикла **while** с учетом операций, связанных с параметром цикла, которые скрыты в операторе **for**. Если параметр цикла является переменной скалярного типа и размер шага определяется функциями следования или предшествования, используются операторы вида

for *переменная* ← *выражение*1 **to** *выражение*2 **do** *оператор*

или

for *переменная* ← *выражение*1 **downto** *выражение*2 **do** *оператор*

соответственно, которые имеют тот же смысл, что и в Паскале. В случаях, когда начальное и конечное значения и размер шага параметра цикла очевидны, используется более простая форма оператора. Например, запись

for $x \in S$ **do** *оператор*

означает, что оператор тела цикла выполняется для каждого элемента множества S, при этом число итераций равно мощности множества;

6) составной оператор имеет тот же смысл, что и в Паскале. В псевдокоде вместо операторных скобок **begin** и **end** используется односторонняя фигурная скобка для обрамления последовательности операторов, образующей составной оператор. Каждый оператор последовательности записывается в отдельной строке, поэтому специальный символ (например, точка с запятой в большинстве языков программирования) для отделения операто-

ров друг от друга не указывается. Время выполнения составного оператора определяется суммой времен выполнения входящих в него операторов;

7) оператор возврата **return** используется в качестве заключительного оператора при определении процедур и функций. Наличие специального оператора возврата делает излишним применение операторных скобок **begin** и **end** для указания начала и конца определения процедуры или функции. В ряде случаев используется расширенная форма оператора возврата вида

return (*выражение*),

где значение вычисленного выражения является значением функции. Время выполнения расширенного оператора возврата определяется временем вычисления выражения. Если же выражение отсутствует, то время выполнения представляет собой некоторую фиксированную величину;

8) оператор вызова процедуры или функции вида

имя_процедуры_или_функции (*фактические_параметры*)

имеет тот же смысл, что и в Паскале. Время выполнения определяется временем вызова с учетом передачи параметров и выполнения процедуры или функции;

9) произвольный оператор представляет собой некоторую фразу на естественном языке, определяющую действие. Такие операторы используются в случаях, когда детали реализации несущественны, очевидны или рассмотрены отдельно. Время выполнения определяется временем, необходимым на реализацию указанного действия.

Для включения в алгоритм различных комментариев, облегчающих его понимание, используется символ //, который начинает комментарий, идущий до конца строки.

Используются следующие соглашения об обозначениях:

а) все логарифмы, если специально не оговорено, берутся по основанию 2 (т.е. запись $\log x$ означает $\log_2 x$), тем более что асимптотические оценки времени выполнения алгоритмов типа $O(\log n)$ не зависят от основания логарифма, поскольку при изменении основания логарифма производится умножение на константу, т. е. $\log_a n = c \log_b n$, где константа $c = \log_a b$;

б) запись вида $\lceil x \rceil$ означает наименьшее целое, большее или равное x, а $\lfloor x \rfloor$ – целая часть x, т. е. наибольшее целое, меньшее

или равное x; очевидно, что $x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1$ для любого x, а также $\lceil n/2\rceil+\lfloor n/2\rfloor=n$ для любого целого n, кроме того, для любого x и для любых целых положительных a и b справедливы равенства $\lceil\lceil x/a\rceil/b\rceil=\lceil x/(ab)\rceil$ и $\lfloor\lfloor x/a\rfloor/b\rfloor=\lfloor x/(ab)\rfloor$;

в) для более компактного представления алгоритмов операция обмена будет записываться в виде $x\leftrightarrow y$ (значения переменных x и y меняются местами), что эквивалентно последовательности из трех операторов присваивания: $t\leftarrow x$; $x\leftarrow y$; $y\leftarrow t$.

Предполагается, что для вычисления логических выражений применяется метод сокращенного вычисления. Если в выражении a **and** b (a и b – логические выражения) установлено, что a = **false**, то без вычисления b можно сказать, что все выражение будет иметь значение **false**. Аналогично для a **or** b, если a = **true**, то и значение всего выражения будет **true**.

1.2. Асимптотические обозначения

В большинстве случаев поставленная перед разработчиком задача может иметь различные методы и соответственно различные алгоритмы решения. Поэтому возникают вопросы оценки качества алгоритмов, сравнения характеристик различных алгоритмов и выбора на основе такого анализа наиболее эффективного алгоритма решения поставленной задачи.

Для оценки алгоритмов существует много критериев, часто противоречащих друг другу. В частности, такие требования, как простота алгоритма для понимания, эффективность использования ресурсов и быстрое выполнение не всегда совместимы. Существуют алгоритмы, к которым могут предъявляться и свои требования, связанные со спецификой решаемых задач. Например, в численных алгоритмах точность и устойчивость являются не менее существенными критериями оценки.

Важным критерием эффективности алгоритма является порядок роста необходимых для решения задачи времени и емкости памяти при увеличении входных данных (*вычислительная сложность*). Мера количества входных данных для каждой конкретной задачи связывается с некоторым числом, называемым *размером задачи*. Время, затрачиваемое алгоритмом на достиже-

ние результата, как функция размера задачи называется *временной сложностью* этого алгоритма. Аналогичным образом можно определить *емкостную сложность*. В учебном пособии основное внимание будет уделяться временной сложности, поскольку для многих рассматриваемых алгоритмов емкостная сложность очевидна.

Для определения временной сложности во многих случаях достаточно оценить асимптотику роста времени работы алгоритма при стремлении размера входа к бесконечности, т. е. определить скорость роста функции (*асимптотическая временная сложность*). Тогда если асимптотика одного алгоритма меньше другого, то в большинстве случаев можно сделать заключение о том, что он более эффективен с точки зрения времени работы (возможно, за исключением малых размеров входа).

Скорость роста функций описывается с помощью асимптотических обозначений. Поскольку размер входа и время работы алгоритма не могут быть отрицательными, предполагается, что при рассмотрении асимптотических соотношений все функции неотрицательны.

Говорят, что функция $f(n)$ *асимптотически равна* функции $g(n)$ (функция $f(n)$ растет с такой же скоростью, как и $g(n)$), и записывают $f(n) \sim g(n)$, если

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Говорят, что функция $g(n)$ является *асимптотически точной оценкой* функции $f(n)$, и записывают $f(n) = \Theta(g(n))$, если существуют константы c_1, $c_2 > 0$ и $n_0 \geq 0$ такие, что

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ для всех } n \geq n_0.$$

Запись $f(n) = \Theta(g(n))$ читается «$f(n)$ есть тэта от $g(n)$». Данное асимптотическое соотношение симметрично: из $f(n) = \Theta(g(n))$ следует $g(n) = \Theta(f(n))$. Пусть $f(n) = 2n^2 + 3n + 2$, $g(n) = n^2$. Согласно определению должны выполняться неравенства

$$c_1 n^2 \leq 2n^2 + 3n + 2 \leq c_2 n^2$$

для всех $n \geq n_0$. Очевидно, что первое неравенство выполняется при $c_1 = 2$, а второе – при $n_0 = 1$ (т. е. для всех $n \geq 1$) и $c_2 = 7$. Таким образом, $2n^2 + 3n + 2 = \Theta(n^2)$. Следует отметить, что правое неравенство никогда не выполняется для $n < 1$: при $n = 0$ имеет место $2 \leq c_2 0^2$, что невозможно при любой константе $c_2 > 0$.

Асимптотическое соотношение $f(n) = \Theta(g(n))$ включает в себя верхнюю и нижнюю оценки. Эти оценки можно разделить.

Говорят, что функция $g(n)$ является *верхней границей скорости роста* функции $f(n)$ (или функция $f(n)$ растет не быстрее, чем функция $g(n)$), и записывают $f(n) = O(g(n))$, если существуют константы $c > 0$ и $n_0 \geq 0$ такие, что

$$f(n) \leq c\,g(n) \text{ для всех } n \geq n_0.$$

Запись $f(n) = O(g(n))$ читается «$f(n)$ есть *о* большое от $g(n)$» или «$f(n)$ имеет порядок (степень) роста $O(g(n))$» и означает, что с ростом n отношение $f(n)/g(n)$ остается ограниченным.

Говорят, что функция $g(n)$ является *нижней границей скорости роста* функции $f(n)$ (или функция $f(n)$ растет не медленнее, чем функция $g(n)$), и записывают $f(n) = \Omega(g(n))$, если существуют константы $c > 0$ и $n_0 \geq 0$, такие что

$$c\,g(n) \leq f(n) \text{ для всех } n \geq n_0.$$

Запись $f(n) = \Omega(g(n))$ читается «$f(n)$ есть омега большое от $g(n)$».

Для любых двух функций $f(n)$ и $g(n)$ асимптотическое отношение $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. Например,

$$f(n) = 2n^2 + 3n + 2 = \Theta(n^2) = O(n^2) = \Omega(n^2).$$

В соответствии с определением можно показать, что функция $f(n) = 2n^2 + 3n + 2$ имеет порядок роста $O(n^3)$, однако это более слабое утверждение, чем то, что $f(n)$ имеет порядок роста $O(n^2)$. В этом случае $2n^2 + 3n + 2 \neq \Omega(n^3)$ и $2n^2 + 3n + 2 \neq \Theta(n^3)$. Поэтому при определении отношения $f(n) = O(g(n))$ стремятся трактовать его как верхнюю границу отношения $f(n) = \Theta(g(n))$.

Говорят, что функция $f(n)$ растет медленнее, чем функция $g(n)$, и записывают $f(n) = o(g(n))$, если

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Запись $f(n) = o(g(n))$ читается «$f(n)$ есть *о* малое от $g(n)$». Например, $2n^2 + 3n + 2 = o(n^3)$, но $2n^2 + 3n + 2 \neq O(n^3)$.

Аналогичным образом вводится ω-обозначение: говорят, что функция $f(n)$ растет быстрее, чем функция $g(n)$, и записывают $f(n) = \omega(g(n))$, если

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=0.$$

Запись $f(n)=\omega(g(n))$ читается «$f(n)$ есть омега малое от $g(n)$». Например, $2n^2+3n+2=\omega(n)$, но $2n^2+3n+2\neq\Omega(n)$.

Важным частным случаем асимптотических обозначений является случай, когда $g(n)=1$, например $O(1)$. Это говорит о том, что соответствующая функция ограничена некоторой положительной константой и не зависит от n. С точки зрения временной сложности обозначение $O(1)$ говорит о том, что время работы постоянно и не зависит от размера входа.

Следует всегда иметь в виду, что в асимптотических отношениях левые и правые части не симметричны, так как правая часть всегда содержит меньше информации, чем левая. Поэтому нельзя заменять левую часть выражения правой. Например, из двух отношений $f(n)=O(g(n))$ и $h(n)=O(g(n))$ не вытекает ложное утверждение, что $f(n)=h(n)$.

Для оценки алгоритмов в качестве основной меры эффективности выполнения алгоритма используется асимптотическая временная сложность. При этом чаще всего учитываются верхние границы роста скорости временной сложности как функции от размера входа. Например, утверждение о том, что временная сложность некоторого алгоритма есть $O(n^2)$ (можно читать как «порядка n^2»), т. е. время работы алгоритма асимптотически ограничено сверху квадратичной функцией, более полезно с точки зрения оценки алгоритма, чем утверждение о том, что временная сложность алгоритма асимптотически ограничена снизу некоторой функцией. Поэтому в литературе для определения временной сложности алгоритмов в большинстве случаев используется O-символика, т. е. запись $f(n)=O(g(n))$ трактуется как верхняя граница отношения $f(n)=\Theta(g(n))$. При этом можно сказать, что функция $f(n)$ асимптотически равна функции $g(n)$, хотя с формальной точки зрения это не совсем корректно, поскольку нижняя граница роста не является частью определения $O(g(n))$.

При анализе алгоритмов с использованием O-символики могут выполняться различные операции, в частности, сложение и умножение. Пусть $T_1(n)$ – время выполнения некоторого фраг-

мента F_1 алгоритма, $T_2(n)$ – фрагмента F_2, $T_1(n)$ имеет временную сложность $O(f(n))$ и $T_2(n) - O(g(n))$. Тогда

$$T_1(n) + T_2(n) = O(\max(f(n), g(n))),$$
$$T_1(n)T_2(n) = O(f(n)g(n)).$$

Правило сложения используется, когда необходимо определить порядок роста времени последовательного выполнения фрагментов F_1 и F_2 алгоритма. Например, $O(n^2 + n) = O(n^2)$. Из правила умножения следует, что $O(cf(n)) = O(f(n))$, если c – положительная константа, т. е. $c = O(1)$.

1.3. Определение времени работы алгоритмов

Время работы алгоритма – это число единиц времени, необходимого для обработки входа размера n. Если известно время выполнения каждой операции алгоритма, то для определения времени работы алгоритма в целом достаточно определить число выполняемых им операций.

В качестве примера рассмотрим несколько алгоритмов подсчета числа единиц в двоичном наборе $B = b_n, b_{n-1}, \ldots, b_2, b_1$ длины n (задача и алгоритмы заимствованы из [21]). *Первый способ* представлен алгоритмом 1.1. Для более простой записи алгоритма можно было бы использовать оператор цикла **for**, но в данном случае специально применен оператор цикла **while**, чтобы показать скрытые в **for** операции. Для каждой строки алгоритма указаны время t_i выполнения (некоторая фиксированная величина, не зависящая от n, т. е. $t_i = O(1)$) и число ее повторений (x – неизвестное число повторений, зависящее от входной последовательности). Ясно, что размером входа является длина n двоичного набора.

	Время t_i	Число повторений
$count \leftarrow 0$	t_1	1
$k \leftarrow 1$	t_2	1
while $k \le n$ **do**	t_3	$n + 1$
if $b_k = 1$	t_4	n
then $count \leftarrow count + 1$	t_5	x
$k \leftarrow k + 1$	t_6	n

Алгоритм 1.1. Первый способ подсчета числа единиц в двоичном наборе

Сложив время t выполнения операций с учетом числа их повторений, можно получить общее время работы алгоритма

$$T(n) = t_1 + t_2 + t_3 + n\,(t_3 + t_4 + t_6) + x\,t_5,$$

где x равно числу единиц в исходном наборе, $0 \le x \le n$.

Подобный детальный анализ алгоритмов трудно применить на практике. Величина t_i показывает только то, что время выполнения i-й операции постоянно и равно t_i, но конкретное ее значение, зависящее от многих факторов (используемый компьютер и набор его машинных команд, качество компиляции и т. п.), обычно определить невозможно. Поэтому используют менее точные оценки, ограничиваясь определением констант пропорциональности, или более общие оценки эффективности, использующие асимптотические обозначения и игнорирующие константы пропорциональности.

Если для алгоритма 1.1 принять время выполнения каждой операции равным единице, то время его работы $T(n) = 3n + x + 3$. Поскольку $0 \le x \le n$, т. е. $x = O(n)$, асимптотическая оценка $T(n) = 3n + O(n) + 3 = O(n)$.

В случаях, когда определение констант пропорциональности вызывает проблемы или не ставится задача их определения, асимптотическую оценку можно получить, используя правила сложения и умножения асимптотик. Рассмотрим такой анализ на примере алгоритма 1.1. Первые два оператора присваивания имеют некоторое постоянное время выполнения, не зависящее от размера входа, т. е. имеют время выполнения порядка $O(1)$. Время выполнения операторов в теле цикла также имеет порядок $O(1)$. Поскольку цикл выполняется n раз, время его выполнения по правилу умножения асимптотик имеет порядок $O(1n) = O(n)$. В результате временная сложность алгоритма есть $O(1) + O(n) = O(n)$.

Время работы алгоритмов во многих случаях зависит не только от размера входа n, но и от самого входа. Поэтому для таких алгоритмов рассматривают время работы *в худшем случае*, *в среднем* и *в лучшем случае*. При анализе алгоритмов наибольший интерес представляют время работы в худшем случае (определяет максимальное время работы для входов данного размера) и в среднем. Определение среднего времени работы алгоритма часто является математически сложной задачей. Кроме того, эта вели-

чина зависит от выбранного распределения вероятностей входов (обычно используется равномерное распределение), которое может отличаться от реального. Время работы в лучшем случае интересно с точки зрения определения характеристик входов, наиболее благоприятных для работы алгоритма.

В алгоритме 1.1 величина x имеет минимальное и максимальное значения, равные соответственно 0 (если входной набор содержит только нули) и n (если входной набор состоит только из единиц). Среднее значение величины x по всем возможным входным наборам при равномерном распределении их вероятностей равно $n/2$. Поэтому время работы алгоритма составляет $T_{\max}(n) = 4n + 3$ в худшем случае, $T_{\text{ave}}(n) = 3{,}5n + 3$ в среднем и $T_{\min}(n) = 3n + 3$ в лучшем случае. Таким образом, временная сложность алгоритма во всех случаях есть $O(n)$.

Второй способ подсчета числа единиц основан на том, что значение разряда в двоичном наборе содержит информацию о числе единиц в этом разряде, и представлен алгоритмом 1.2. С целью сравнения с алгоритмом 1.1 для идентичных операций использованы те же обозначения t_i времени выполнения. Время работы данного алгоритма составляет

$$T(n) = t_1 + t_2 + t_3 + n(t_3 + t_6 + t_7) = 3n + 3 = O(n)$$

и не зависит от входа (зависит только от размера входа), т. е. любые входные наборы обрабатываются за одно и то же время.

	Время t_i	Число повторений
$count \leftarrow 0$	t_1	1
$k \leftarrow 1$	t_2	1
while $k \le n$ **do**	t_3	$n + 1$
$count \leftarrow count + b_k$	t_7	n
$k \leftarrow k + 1$	t_6	n

Алгоритм 1.2. Второй способ подсчета числа единиц в двоичном наборе

Алгоритмы 1.1 и 1.2 имеют одну и ту же сложность $O(n)$. Если принять во внимание константы пропорциональности, то алгоритм 1.2 более эффективен. Константы пропорциональности были определены при условии, что время выполнения каждой операции равно единице. Для близких по времени работы алгоритмов такое упрощение может оказаться достаточно грубым. Для иллюстрации сказанного определим, при каких условиях алгоритм 1.1 в среднем эффективнее алгоритма 1.2. Пусть $a_1 = t_3 +$

$+ t_4 + t_5/2 + t_6$, $a_2 = t_3 + t_6 + t_7$, $b = t_1 + t_2 + t_3$. Тогда время работы алгоритма 1.1 в среднем $T_1(n) = a_1 n + b$, а алгоритма 1.2 – $T_2(n) = a_2 n + b$. Из неравенства $T_1(n) \le T_2(n)$ следует, что $a_1 \le a_2$, т. е. $t_4 + t_5/2 \le t_7$. Время t_5 выполнения оператора присваивания $count \leftarrow count + 1$ и время t_7 выполнения оператора присваивания $count \leftarrow count + b_k$ можно связать соотношением $t_7 = t_5 + c$, где c – некоторая константа. Тогда алгоритм 1.1 в среднем эффективнее алгоритма 1.2, если $t_4 \le t_5/2 + c$. Более того, он эффективнее даже в худшем случае, если $t_4 \le c$. Другой вопрос, существует ли такая константа. Это зависит от результатов трансляции, архитектуры процессора и т.д. Вполне может оказаться, что $c = 0$ и, следовательно, алгоритм 1.1 не будет эффективнее алгоритма 1.2.

Третий способ подсчета числа единиц в двоичном наборе представлен алгоритмом 1.3. Основой алгоритма является операция $B \wedge (B - 1)$, которая заменяет в наборе B самую правую единицу нулем (здесь B интерпретируется как двоичное представление числа при выполнении операции вычитания и как двоичный код при выполнении операции конъюнкции). Поэтому цикл повторяется до тех пор, пока B не станет равным нулю, т. е. будет состоять только из нулей. Число итераций и будет являться результатом работы алгоритма.

$$count \leftarrow 0$$
$$\textbf{while}\ \ B \neq 0\ \ \textbf{do} \begin{cases} count \leftarrow count + 1 \\ B \leftarrow B \wedge (B - 1) \end{cases}$$

Алгоритм 1.3. Третий способ подсчета числа единиц в двоичном наборе

Если предположить, что оператор $B \leftarrow B \wedge (B - 1)$ соответствует двум операторам присваивания, т. е. требует две единицы времени, а величина x есть число единиц в наборе B, то время выполнения алгоритма $T(n) = 4x + 2$. Поскольку $0 \le x \le n$, время работы алгоритма составит $T_{\max}(n) = 4n + 2$ в худшем случае, $T_{\text{ave}}(n) = 2n + 2$ в среднем и $T_{\min}(n) = 2$ в лучшем случае. Таким образом, временная сложность алгоритма в худшем случае и в среднем есть $O(n)$, в лучшем случае – $O(1)$. Очевидно, что данный алгоритм в среднем эффективнее алгоритмов 1.1 и 1.2, особенно на наборах с малым числом единиц.

Различные аспекты использования временной сложности алгоритмов можно проиллюстрировать на следующих примерах. Пусть имеются алгоритмы A_1, A_2 и A_3 с временем выполнения n, n^2 и 2^n соответственно. Для определенности за единицу времени примем одну миллисекунду. Тогда алгоритм A_1 может обработать за одну секунду вход размера 1000, A_2 – вход размера 31, а A_3 – вход размера не более 9. Пусть эти алгоритмы выполняются на компьютере с более высокой производительностью, например в 10 раз. Тогда для алгоритма A_1 десятикратное увеличение скорости увеличивает размер задачи, которую можно решить, в 10 раз, для A_2 – более чем утраивается, а для A_3 – только на три. Это показывает, насколько важен вопрос выбора наиболее эффективного алгоритма и что асимптотическая сложность является важной мерой эффективности алгоритмов.

Таким образом, для сравнения двух алгоритмов необходимо сначала сравнить их асимптотические сложности. Затем, если они окажутся одинаковыми, перейти к определению констант пропорциональности и другим тонкостям. При этом необходимо иметь в виду, что большой порядок роста времени выполнения может иметь меньшую константу пропорциональности, чем малый порядок роста. В таком случае алгоритм с быстро растущей функцией времени выполнения может оказаться предпочтительнее для задач с малым размером. Пусть, например, имеются алгоритмы A_1, A_2, A_3 и A_4 с временем выполнения соответственно $1000n$, $100n \log n$, $10n^2$ и 2^n. Тогда A_4 будет наилучшим для задач размера $2 \le n \le 9$, A_3 – для задач размера $10 \le n \le 58$, A_2 – при $59 \le n \le 1024$, а A_1 – при $n > 1024$.

1.4. Рекуррентные соотношения

Анализ рекурсивных алгоритмов обычно приводит к соотношениям, которые называются *рекуррентными*. В них время выполнения алгоритма выражается через время выполнения того же алгоритма на входах меньшего размера. В качестве примера рассмотрим рекурсивную функцию *FACT* вычисления $n!$ для целых чисел от 0 до n включительно, представленную алгоритмом 1.4. Очевидно, что размером входа является значение n.

function $FACT(n)$
  if $n \le 1$
    then $FACT \leftarrow 1$
    else $FACT \leftarrow n \times FACT(n-1)$
return

Алгоритм 1.4. Рекурсивное вычисление факториала

Проверка условия в операторе **if** и выполнение присваивания в его **then**-части требуют времени порядка $O(1)$ (обозначенного константой a_1). Время выполнения оператора в **else**-части складывается из времени работы функции $FACT(n-1)$ и постоянной составляющей (обозначенной константой a_2) порядка $O(1)$, которая включает в себя проверку условия, вызов функции, умножение и присваивание. Следовательно, время работы алгоритма описывается рекуррентным соотношением

$$T(n) = \begin{cases} a_1, & \text{если } n \le 1, \\ T(n-1) + a_2, & \text{если } n > 1. \end{cases} \qquad (1.1)$$

Рассмотрим три способа решения рекуррентных соотношений: метод подстановки, метод итераций и общий метод [9].

Метод подстановки. Идея метода заключается в нахождении такой функции $f(n)$, чтобы для всех n выполнялось неравенство $T(n) \le f(n)$. Тогда функция $f(n)$ будет верхней границей скорости роста $T(n)$, т. е. $T(n) = O(f(n))$. Другими словами, следует отгадать ответ и затем доказать его по индукции.

В качестве примера определим верхнюю оценку рекуррентного соотношения (1.1). Вероятнее всего, $T(n) = O(n)$, т. е. $T(n) \le cn$ для некоторой константы $c > 0$. При $n = 0$ это условие не выполняется, поскольку cn равно нулю, независимо от значения c. Применим функцию $f(n) = cn + d$, т. е. $T(n) \le cn + d$, где $d > 0$. Теперь при $n \le 1$ условие выполняется, если $d \ge a_1$ и $c > 0$. Пусть $n > 1$. Должно выполняться условие

$$T(n) = T(n-1) + a_2 \le cn + d.$$

Пусть эта оценка верна для $n - 1$, т. е. $T(n-1) \le c(n-1) + d$. Подставив ее в рекуррентное соотношение, получим

$$T(n) \le c(n-1) + d + a_2 = cn - c + d + a_2 \le cn + d.$$

Данное неравенство выполняется, если $c \ge a_2$. Таким образом, оценка $T(n) \le cn + d$ будет справедлива, если $d \ge a_1$ и $c \ge a_2$. Если положить $d = a_1$ и $c = a_2$, т. е. определить константы про-

порциональности функции $f(n) = cn + d$, то для всех $n \geq 0$ выполняется неравенство $T(n) \leq a_2 n + a_1$. Следовательно, $T(n) = O(n)$.

Рассмотрим более сложное рекуррентное соотношение

$$T(n) = \begin{cases} a_1, & \text{если } n = 1, \\ 2T(n/2) + a_2 n, & \text{если } n > 1, \end{cases} \qquad (1.2)$$

где a_1 и a_2 – некоторые положительные константы. Предположим, что $T(n) \leq cn \log n$. При $n = 1$ отношение не выполняется, так как в этом случае выражение $cn \log n$ всегда равно нулю. Применим функцию $T(n) \leq cn \log n + d$. Теперь при $n = 1$ эта оценка справедлива, если $d \geq a_1$. Пусть $n > 1$. Должно выполняться условие $T(n) = 2\,T(n/2) + a_2 n \leq cn \log n + d$. Пусть эта оценка верна для $n/2$, т. е. $T(n/2) \leq c\,(n/2) \log (n/2) + d$. Подставив ее в рекуррентное соотношение, получим

$$\begin{aligned} T(n) &\leq 2(c\,(n/2)\log (n/2) + d) + a_2 n = \\ &= cn \log n - cn \log 2 + 2d + a_2 n = \\ &= cn \log n - cn + 2d + a_2 n \leq cn \log n + d. \end{aligned}$$

Неравенство выполняется, если $c \geq a_2 + d$. Если положить $d = a_1$ и $c = a_1 + a_2$, то для всех $n > 0$ выполняется неравенство $T(n) \leq (a_1 + a_2)\, n \log n + a_1$. Следовательно, $T(n) = O(n \log n)$.

В ряде случаев сложные рекуррентные соотношения можно упростить, если применить известный в математике прием – замену переменных. В качестве примера рассмотрим соотношение

$$T(n) = \begin{cases} a_1, & \text{если } n = 1, \\ 2T(\sqrt{n}) + a_2 \log n, & \text{если } n > 1. \end{cases}$$

Выполним замену переменной m на $\log n$, т. е. $m = \log n$. В результате получим

$$T(2^m) = 2T(2^{m/2}) + a_2 m\,.$$

Замена $T(2^m)$ на $S(m)$ приводит к соотношению

$$S(m) = 2\,S(m/2) + a_2 m\,,$$

которое соответствует рекуррентной части соотношения (1.2) и, следовательно, имеет решение $S(m) = O(m \log m)$. Возвращаясь к обозначению $T(n)$ вместо $S(m)$, получим

$$T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)\,.$$

Метод итераций. Заключается в итерации рекуррентного соотношения, т. е. подстановки его в самого себя до тех пор, пока из правой части не исключатся рекурсивные обращения.

Рассмотрим соотношение (1.1). Подставляя его самого в себя, получим

$$T(n) = T(n-1) + a_2 = T(n-2) + 2a_2 = T(n-3) + 3a_2.$$

Продолжая этот процесс, в общем случае для некоторого $i < n$ получаем $T(n) = T(n-i) + i\,a_2$. Положив в последнем выражении $i = n-1$, окончательно получаем

$$T(n) = T(1) + a_2(n-1) = a_2(n-1) + a_1 = O(n).$$

Рассмотрим этот метод для соотношения (1.2). Подставляя его самого в себя, получим

$$\begin{aligned} T(n) &= 2T(n/2) + a_2 n = 2(2T(n/4) + a_2 n/2) + a_2 n \\ &= 4T(n/4) + 2a_2 n = 8T(n/8) + 3a_2 n. \end{aligned}$$

Продолжая этот процесс, в общем случае для некоторого $i < n$, получаем

$$T(n) = 2^i T(n/2^i) + i\,a_2 n.$$

Пусть $n = 2^k$, т. е. является степенью числа 2. Тогда при $i = k$ процесс подстановок завершается подстановкой $T(1)$

$$T(n) = 2^k T(1) + k\,a_2 n.$$

Поскольку $k = \log n$ и $T(1) = a_1$, окончательно получаем

$$T(n) = a_2 n \log n + a_1 n = O(n \log n).$$

Общий метод. Этот метод позволяет получить асимптотические оценки для рекуррентных соотношений вида

$$\begin{aligned} T(1) &= d, \\ T(n) &= a\,T(n/b) + f(n), \end{aligned} \qquad (1.3)$$

где $a \geq 1$, $b > 1$ и $d > 0$ – некоторые константы, а $f(n)$ – положительная функция. Данное соотношение обычно получается, когда алгоритм разбивает исходную задачу размера n на a подзадач размера n/b. Каждая подзадача решается рекурсивно за время $T(n/b)$ и результаты объединяются за время $f(n)$. Такой подход к проектированию эффективных алгоритмов называется *методом декомпозиции* (или методом разбиения).

Пусть n является натуральной степенью числа b, т. е. $n = b^k$. Применим метод итераций, выполняя последовательную подстановку рекуррентного соотношения самого в себя

$$\begin{aligned} T(n) &= f(n) + a\,T(n/b) = f(n) + a\,f(n/b) + a^2 T(n/b^2) = \\ &= f(n) + a\,f(n/b) + a^2 f(n/b^2) + \ldots + a^{k-1} f(n/b^{k-1}) + a^k T(1), \end{aligned}$$

где $k = \log_b n$. Поскольку $a^k = a^{\log_b n} = n^{\log_b a}$, получаем

$$T(n) = d\, n^{\log_b a} + \sum_{j=0}^{k-1} a^j f(n/b^j). \qquad (1.4)$$

Таким образом, рекуррентное соотношение можно представить в виде суммы, вычисление которой позволит определить временную сложность алгоритма. Например, в соотношении (1.2) имеем $d = a_1$, $a = b = 2$, $f(n) = a_2 n$, $k = \log n$, поэтому

$$T(n) = a_1 n + \sum_{j=0}^{k-1} 2^j a_2 n / 2^j = a_1 n + a_2 n \sum_{j=0}^{k-1} 1 =$$
$$= a_1 n + a_2 n\, k = a_2 n \log n + a_1 n = O(n \log n).$$

Часто преобразование рекуррентного соотношения в сумму приводит к довольно сложным выражениям. Если требуется определить только асимптотические оценки, общий метод решения рекуррентных соотношений позволяет получить их более простым способом для достаточно широкого класса функций $f(n)$.

Рассмотрим влияние каждого выражения в формуле (1.4) на оценку функции $T(n)$. Асимптотическая оценка первого выражения очевидна и составляет $O(n^{\log_b a})$. Рассмотрим функцию

$$g(n) = \sum_{j=0}^{k-1} a^j f(n/b^j).$$

С точки зрения анализа алгоритмов наибольший интерес представляют ряды, образуемые функцией $f(n)$ вида $c\, n^{\alpha}$, где $c > 0$ и $\alpha \geq 0$ – некоторые константы. Тогда ряд представляет собой геометрическую прогрессию

$$g(n) = cn^{\alpha} + ac(n/b)^{\alpha} + a^2 c(n/b^2)^{\alpha} + \ldots + a^{k-1} c(n/b^{k-1})^{\alpha}$$

с числом членов $k = \log_b n$ и знаменателем $q = a/b^{\alpha}$.

Возможны следующие ситуации:

1) если $a > b^{\alpha}$, т. е. $\alpha < \log_b a$ или $\alpha = \log_b a - \varepsilon$ для некоторой константы $\varepsilon > 0$, то знаменатель $q > 1$. Для такой возрастающей прогрессии сумма асимптотически равна последнему члену, т. е. $g(n) = O(a^{k-1}) = O(a^{\log_b n - 1}) = O(n^{\log_b a} / a)$. Таким образом,

$$T(n) = O(n^{\log_b a}) + O(n^{\log_b a} / a) = O(n^{\log_b a});$$

2) если $a = b^{\alpha}$, т. е. $\alpha = \log_b a$, то знаменатель $q = 1$ (все члены прогрессии равны). Число членов есть $\log_b n$, поэтому сумма равна $cn^{\log_b a} \log_b n$, т. е. $g(n) = O(n^{\log_b a} \log n)$. Следовательно,

$$T(n) = O(n^{\log_b a}) + O(n^{\log_b a} \log n) = O(n^{\log_b a} \log n);$$

3) если $a < b^{\alpha}$, т. е. $\alpha > \log_b a$ или $\alpha = \log_b a + \varepsilon$ для некоторой константы $\varepsilon > 0$, то знаменатель меньше единицы. Для такой убывающей прогрессии сумма асимптотически равна первому члену, т. е. $g(n) = O(n^{\alpha})$.

В этом случае, поскольку $\alpha > \log_b a$,

$$T(n) = O(n^{\log_b a}) + O(n^{\alpha}) = O(n^{\alpha}) = O(f(n)).$$

Таким образом, метод асимптотической оценки рекуррентных соотношений вида (1.3) для функций $f(n)$ вида cn^{α} формулируется следующим образом:

$$T(n) = \begin{cases} O(n^{\log_b a}), & \text{если } a > b^{\alpha}, \\ O(n^{\log_b a} \log n), & \text{если } a = b^{\alpha}, \\ O(f(n)), & \text{если } a < b^{\alpha}. \end{cases} \qquad (1.5)$$

Например, в соотношении (1.2) имеем $a = b = 2$, $f(n) = a_2 n$, т. е. $\alpha = 1$. Поскольку $2 = 2^1$, т. е. подходит второй случай, получаем $T(n) = O(n \log n)$.

Более общий метод решения рекуррентных соотношений вида (1.3) формулируется следующим образом [9]:

1) если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторой величины $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$;

2) если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;

3) если $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для некоторой величины $\varepsilon > 0$ и если $af(n/b) \le cf(n)$ для некоторой константы $c < 1$ и достаточно больших n, то $T(n) = \Theta(f(n))$.

Следует отметить, что в первом случае недостаточно, чтобы функция $f(n)$ была асимптотически меньше, чем $n^{\log_b a}$, а необходим некоторый зазор размера n^{ε} для некоторого $\varepsilon > 0$. Аналогично в третьем случае функция $f(n)$ должна быть асимптотиче-

ски больше $n^{\log_b a}$ с зазором n^{ε}, кроме того, должно удовлетворяться условие $af(n/b) \le cf(n)$.

В качестве примера применения общего метода рассмотрим рекуррентное соотношение $T(n) = 3\,T(n/4) + n \log n$ с начальным значением $T(1) = 1$. Имеем $a = 3$, $b = 4$, $f(n) = n \log n$; при этом $n^{\log_b a} = n^{\log_4 3} < n^{0,8}$. Функция $f(n)$ асимптотически больше, чем $n^{\log_b a}$, так как отношение $f(n)/n^{\log_b a} = (n \log n)/n^{0,8} = n^{0,2} \log n$ оценивается снизу величиной $n^{0,2}$, т. е. имеется зазор $n^{0,2}$. Проверим условие $af(n/b) \le cf(n)$: $3(n/4) \log (n/4) \le c\,n \log n$. Условие выполняется для $c = 3/4$. Таким образом, согласно третьему условию, $T(n) = \Theta(n \log n)$.

Существуют и другие типы рекуррентных соотношений, для которых не подходит рассмотренный общий метод решения, например $T(n) = 2\,T(n/2) + n \log n$ с начальным значением $T(1) = 1$. Попробуем применить общий метод. Имеем $a = b = 2$, $f(n) = n \log n$, $n^{\log_b a} = n$. Очевидно, что $f(n) = n \log n$ асимптотически больше, чем $n^{\log_b a} = n$. Однако зазор недостаточен, поскольку отношение $f(n)/n^{\log_b a} = (n \log n)/n = \log n$ не оценивается снизу величиной n^{ε} ни для какого $\varepsilon > 0$. Таким образом, применить общий метод не удается. Тем не менее это соотношение легко можно решить по формуле (1.4):

$$T(n) = n + \sum_{j=0}^{k-1} 2^j (n/2^j) \log (n/2^j) = n + n \sum_{j=0}^{k-1} (\log n - j) =$$
$$= n + n \log n (\log n + 1)/2 = O(n \log^2 n).$$

В рассмотренных выше методах решения рекуррентных соотношений предполагалось, что n является целой степенью числа b, т. е. $n = b^k$. Для произвольных значений n в рекуррентных соотношениях должны рассматриваться только целые части, поскольку функция $T(n)$ определена только для целых n. Например, рекуррентная часть соотношения (1.2) должна записываться в виде $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + a_2 n$, а в соотношении (1.3) под выражением n/b должно пониматься либо $\lceil n/b \rceil$, либо $\lfloor n/b \rfloor$. Во многих случаях оценки, полученные в предположении, что $n = b^k$, распространяются и на произвольные значения n. Это объясняется тем, что задачу размера n можно вложить в задачу

размера n', где n' – наименьшая степень числа b, большая или равная n, а затем решить рекуррентное соотношение для задачи размера n'. Однако следует быть внимательным к деталям, связанным с округлением до целого сверху или снизу, поскольку возможны случаи (пусть и достаточно редкие), когда оценки для $n = b^k$ могут отличаться от оценок для других значений n.

Упражнения

1. Используя общий метод, определить асимптотические оценки для следующих рекуррентных соотношений, предполагая, что $T(1) = 1$:

а) $T(n) = 2\,T(n/2) + \log n$;
б) $T(n) = 2\,T(n/2) + n^2$;
в) $T(n) = 2\,T(n/2) + n^3$;
г) $T(n) = 3\,T(n/2) + n$;
д) $T(n) = 3\,T(n/2) + n^2$;
е) $T(n) = 3\,T(n/2) + n^3$;
ж) $T(n) = 4\,T(n/2) + n$;
з) $T(n) = 4\,T(n/2) + n^2$;
и) $T(n) = 4\,T(n/2) + n^3$.
к) $T(n) = 7\,T(n/3) + n^2$;
л) $T(n) = 7\,T(n/4) + n^2$;
м) $T(n) = 16\,T(n/4) + n^2$.

2. Решить методом подстановки рекуррентные соотношения из упражнения 1 и сравнить результаты.

3. Решить методом итераций рекуррентные соотношения из упражнения 1 и сравнить результаты.

***4**. С помощью замены переменных решить следующие рекуррентные соотношения, считая, что $T(1) = 1$:

а) $T(n) = T(\sqrt{n}) + 1$;

б) $T(n) = 2T(\sqrt{n}) + 1$;

в) $T(n) = 3T(\sqrt{n}) + 1$.

5. Даны два алгоритма A_1 и A_2. Время работы алгоритма A_1 описывается соотношением

$$T_1(1) = 1,$$
$$T_1(n) = 7\,T_1(n/2) + n^2,$$

а время работы алгоритма A_2 – соотношением

$$T_2(1) = 1,$$
$$T_2(n) = a\,T_2(n/4) + n^2.$$

Определить наибольшее целое значение величины a, при котором алгоритм A_2 асимптотически быстрее алгоритма A_1.

***6**. Разработать рекурсивный алгоритм нахождения наибольшего и наименьшего элементов множества. Получить рекуррентное соотношение, описывающее время его работы. Определить функцию, являющуюся решением полученного рекуррентного соотношения.

7. Разработать нерекурсивный алгоритм нахождения наибольшего и наименьшего элементов множества. Определить время его работы и сравнить полученную функцию с результатом упражнения 6.

Глава 2. СТРУКТУРЫ ДАННЫХ

Процесс разработки алгоритма для решения некоторой прикладной задачи можно разбить на два этапа. На первом этапе осуществляется формализация исходной задачи, т. е. создается ее математическая модель с привлечением соответствующих математических конструкций, и строится неформальный алгоритм ее решения. На втором этапе определяются используемые абстрактные типы данных и алгоритм записывается на некотором псевдоязыке (псевдокод, блок-схема алгоритма и др.) в терминах этих абстрактных типов данных.

Абстрактный тип данных (АТД) представляет собой математическую модель данных с множеством операций, определенных для этих данных [2]. *Реализация* АТД предполагает выбор *структуры данных* для представления данных и разработку алгоритмов выполнения соответствующих операций. Концептуально АТД соответствует классу в объектно-ориентированном программировании.

При разработке алгоритмов часто приходится иметь дело с таким АТД, как множество. Примерами операций над множествами являются объединение, пересечение, определение принадлежности элемента множеству (поиск), модификация (добавление или исключение элемента, сортировка) и т. д. Трудно сформировать универсальный набор всех операций над множествами, удовлетворяющий всем возможным приложениям. Структуры данных для представления множеств во многом зависят от набора операций, которые необходимо выполнять над ними. Поэтому в этой главе рассматриваются только основные структуры данных для представления конечных множеств и некоторые операции.

Множество представляет собой объединение различных элементов. В некоторых приложениях более естественными являются понятие *мультимножества* (которое отличается от множества тем, что в нем могут содержаться одинаковые элементы) и понятие *последовательности*. Последовательность (список, кортеж, вектор) $x_1, x_2, x_3, \ldots$ формально определяется как функция, областью определения которой является множество положительных целых чисел. В такой индексированной последовательности каждый элемент занимает определенную позицию

(индекс), т. е. существует однозначное соответствие между индексом элемента и элементом последовательности. В последовательности могут быть одинаковые элементы, но занимающие разные позиции. Рассматриваемые структуры данных для представления множеств очевидным образом можно использовать и для представления мультимножеств и последовательностей.

2.1. Последовательное распределение

Простейшим представлением множества $X = \{x_1, x_2, \ldots, x_n\}$ является точный список его элементов, расположенных в последовательных ячейках памяти, т. е. множество представляется с помощью массива и требует для своего хранения непрерывную область памяти. Такое представление будем называть *последовательным распределением*.

Пусть для хранения одного элемента требуется d ячеек памяти, а элемент x_1 хранится, начиная с ячейки с адресом l_1. Тогда элемент x_2 хранится, начиная с ячейки $l_2 = l_1 + d$, элемент x_3 – начиная с ячейки $l_3 = l_1 + 2d$ и т. д. Таким образом, существует простое соотношение между номером i элемента и адресом l_i ячейки, с которой начинается хранение x_i: $l_i = l_1 + (i - 1)\,d$. Данное соотношение обеспечивает *прямой доступ* к любому элементу множества, т. е. время доступа к любому элементу множества фиксировано и не зависит от его мощности.

Элемент x_i множества может иметь сложную структуру. Например, матрицу $[a_{ij}]$ размером $m \times n$ можно представить как множество строк $s_1, s_2, \ldots, s_m$, в котором каждый элемент s_i, в свою очередь, является множеством, состоящим из n элементов. Обозначим через d число ячеек для хранения элемента a_{ij}. Тогда для хранения элемента s_i (строки матрицы) требуется nd ячеек, а сам элемент s_i хранится, начиная с ячейки $l_i = l_{11} + (i - 1)\,nd$, где l_{11} – адрес первого элемента первой строки. Таким образом, элемент a_{ij} хранится, начиная с ячейки

$$l_{ij} = l_i + (j - 1)\,d = l_{11} + ((i - 1)\,n + (j - 1))\,d.$$

В данном случае получена формула прямого доступа к элементу матрицы для ее построчного представления. Постолбцовое представление матрицы получается, если матрицу рассматривать как множество $t_1, t_2, \ldots, t_n$ столбцов, а каждый элемент

t_j, в свою очередь, является множеством из m элементов j-го столбца матрицы.

Проиллюстрируем возможность обеспечения прямого доступа к элементам матрицы с различной длиной строк. В ряде приложений (например, минимизация конечных автоматов) используются симметричные матрицы $[a_{ij}]$ размером $n \times n$, в которых $a_{ij} = a_{ji}$. Для экономии памяти можно хранить не всю матрицу, а только верхний или нижний треугольник. Построим последовательное распределение для нижнего треугольника. Матрица представляет собой множество строк $s_1, s_2, \ldots, s_n$, в котором каждый элемент s_i, в свою очередь, является множеством, состоящим из i элементов, т. е. для хранения элемента s_i необходимо id ячеек. Тогда элемент s_i хранится, начиная с ячейки

$$l_i = l_{11} + d + 2d + \ldots + (i-1)d = l_{11} + di(i-1)/2,$$

а элемент a_{ij} – с ячейки

$$l_{ij} = l_i + (j-1)d = l_{11} + (i(i-1)/2 + j - 1)d.$$

Очевидно, что при обращении к элементу матрицы всегда должно выполняться условие $i \geq j$. Поэтому, если $i < j$, необходимо обращаться к элементу a_{ji}. В результате такого представления экономится почти 50% памяти, но несколько увеличивается время вычислений из-за дополнительной проверки условия $i < j$ при каждом обращении к элементу матрицы.

Множества, которые часто модифицируются, будем называть *динамическими*, в противном случае – *статическими*. Ограничимся рассмотрением двух операций для динамических множеств: включение (добавление) элемента во множество и исключение (удаление) элемента из множества.

При последовательном распределении операция включения вставляет элемент z в заданную позицию i множества X. Сначала производится перемещение элементов $x_i, x_{i+1}, \ldots, x_n$ на одну позицию вправо, затем в освободившуюся позицию i помещается элемент z. Операция исключения удаляет элемент из заданной позиции i, перемещая элементы $x_{i+1}, \ldots, x_n$ на одну позицию влево. Очевидно, что эти операций требуют времени $O(n)$.

Основным достоинством последовательного распределения является возможность прямого доступа к любому элементу множества, поскольку существует простое соотношение между порядковым номером элемента в множестве и адресом ячейки па-

мяти, в которой он хранится. Кроме того, данное представление легко реализуемо и требует небольших расходов памяти (для статических множеств).

Существенным недостатком последовательного распределения является неэффективность реализации динамических множеств. Этот недостаток является следствием того, что массив по своей сути является статической структурой. Во-первых, размер массива приходится задавать (резервировать объем памяти) исходя из максимально возможного размера множества независимо от реального размера в конкретный момент времени, что может привести к неэффективному использованию памяти. Кроме того, не всегда можно заранее определить верхнюю границу мощности множества. Во-вторых, время выполнения операций включения и исключения зависит от размера множества.

Важной разновидностью последовательного распределения является *характеристический вектор*. Он представляет собой двоичный вектор и может быть полезен в тех случаях, когда необходимо представить подмножество некоторого основного множества. Пусть $S = \{s_1, s_2, \ldots, s_n\}$ – основное множество. Тогда подмножество $X \subseteq S$ можно представить в виде характеристического вектора V_X, состоящего из n двоичных разрядов, такого, что i-й разряд в V_X равен единице, если $s_i \in X$, и равен нулю в противном случае. Например, характеристический вектор множества X всех простых чисел между 1 и 15, т. е. основное множество $S = \{1, 2, \ldots, 15\}$, имеет вид

S:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V_X:	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

.

В качестве основных достоинств характеристических векторов можно отметить их компактность (для представления элемента используется один двоичный разряд), легкость выполнения таких операций, как определение принадлежности элемента данному множеству, включение и исключение (эти операции выполняются за фиксированное время, не зависящее от размера множества). Более того, такие операции, как объединение и пересечение, в ряде случаев можно осуществить с помощью операций дизъюнкции и конъюнкции над двоичными векторами.

Основной недостаток характеристических векторов обусловлен тем, что должно существовать простое соотношение между i и s_i. Если простого соотношения не существует, то это затрудняет (если соотношение сложное, может возрасти время обработки) или делает невозможным использование характеристических векторов. Кроме того, сильно разреженные векторы (которые содержат относительно мало единиц) неэкономичны с точки зрения памяти, поскольку требуемый объем памяти пропорционален мощности множества S, а не множества X. По этой же причине операции, основанные на обработке каждого элемента множества X (включая и операции объединения и пересечения множеств, если нет возможности использовать логические операции над двоичными векторами), требуют времени, пропорционального $|S|$.

2.2. Связное распределение

2.2.1. Связный список

При *связном распределении* для хранения множества не требуется выделения непрерывной области памяти (в отличие от последовательного распределения). Поэтому элементы множества $X = \{x_1, x_2, \ldots, x_n\}$ могут располагаться в произвольных областях памяти. Чтобы сохранить информацию о порядке следования элементов, необходимо каждому элементу x_i поставить в соответствие *указатель* (*ссылку*) на следующий элемент x_{i+1}. Значение указателя можно трактовать как адрес области памяти, где находится соответствующий элемент. Такое представление множества называется *связным списком*. Каждый элемент списка, называемый *узлом*, состоит из двух полей: в поле *info* размещается сам элемент множества, а в поле связи *next* – указатель на следующий за ним элемент. Доступ к узлу возможен только в том случае, если на него указывает хотя бы один указатель. Значение поля узла представлено в виде *указатель.имя_поля*. Например, запись *p.info* определяет значение поля *info* узла, на который ссылается указатель *p*. Для доступа ко всему списку должен существовать внешний указатель *list*, который указывает на первый элемент списка. Поле *next* последнего элемента списка, поскольку для него не существует следующего элемента,

должно содержать так называемое *пустое*, или *нулевое*, значение указателя, которое будем обозначать символом Λ (**nil** в Паскале). Список, не содержащий элементов, называется *пустым*, и для него *list* = Λ. Представление множества $X = \{x_1, x_2, \ldots, x_n\}$ в виде связного списка показано на рисунке 2.1.

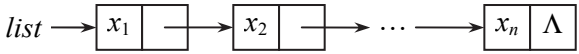


Рис. 2.1. Связный список

Связное распределение ориентировано на реализацию динамических множеств и облегчает выполнение соответствующих операций. Поэтому подобные структуры часто называют *динамическими*. Следует отметить, что, в отличие от массива для хранения списка, память заранее не резервируется. Поэтому должен существовать механизм, который по мере необходимости выделяет требуемый объем памяти для размещения узла. Предполагая, что такой механизм существует (а он существует во многих языках программирования), для запроса на выделение памяти будем использовать процедуру *new*(*p*), которая размещает новый пустой узел в некоторой области памяти и присваивает указателю *p* ссылку на этот узел. Необходим также механизм освобождения области памяти, занимаемой некоторым узлом (*сборка мусора*). Это связано с тем, что если к узлу по каким-либо причинам (например, в результате исключения) отсутствует возможность доступа, то этот узел не может быть использован и становится бесполезным, однако он занимает определенную область памяти. Для освобождения памяти, занимаемой узлом, на который ссылается указатель *p*, будем использовать процедуру *dispose*(*p*), после выполнения которой указатель *p* = Λ и любая ссылка по этому значению указателя запрещена.

Включение элемента *z* в позицию *i* множества *X* предполагает создание нового узла, запись значения элемента *z* в поле *info* этого узла и установку значений двух указателей (рис. 2.2). Новый узел должен следовать в списке за узлом с элементом x_{i-1}, поэтому с помощью указателя *p* необходимо обеспечить доступ к этому узлу для изменения значения поля *next*, чтобы оно указывало на добавленный узел. Старое значение этого поля, кото-

рое указывало на узел с элементом x_i, для сохранения целостности списка должно быть предварительно записано в поле *next* нового узла. Процедура *INSERT*(z, p), реализующая включение элемента z после узла, на который ссылается указатель p, представлена в алгоритме 2.1. Данная процедура не может включить элемент в начало списка или в пустой список, поскольку в этом особом случае необходимо изменить значение указателя *list*. Для реализации такого включения служит процедура *INS_FIRST*. При необходимости эти процедуры можно объединить в одну, включив в нее действия по распознаванию особого случая.

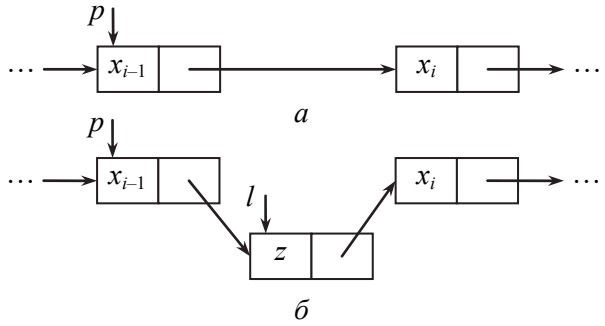


Рис. 2.2. Включение элемента в связный список:
а – до включения; *б* – после включения.

procedure *INSERT*(z, p)	**procedure** *INS _ FIRST*(z, *list*)
new(l)	*new*(l)
l.*info* ← z	l.*info* ← z
l.*next* ← p.*next*	l.*next* ← *list*
p.*next* ← l	*list* ← l
return	**return**

Алгоритм 2.1. Процедуры включения элемента в связный список

Исключение элемента x_i из множества X предполагает удаление из списка узла, содержащего элемент x_i, установку значения одного указателя и освобождение памяти, занимаемой исключаемым узлом (рис. 2.3). Исключаемый узел следует в списке за узлом с элементом x_{i-1}, поэтому с помощью указателя p необходимо обеспечить доступ к этому узлу для изменения значения поля *next*, чтобы оно указывало на узел с элементом x_{i+1}. Проце-

дура *DELETE*(*p*), реализующая исключение узла, расположенного после узла, на который ссылается указатель *p*, приведена в алгоритме 2.2. Очевидно, что, если не стоит задача сборки мусора, для исключения узла достаточно установить *p.next* ← ← *p.next.next*). Процедура *DELETE* не может исключить элемент из начала списка. Для реализации этого особого случая служит процедура *DEL_FIRST*. Эти процедуры также можно объединить в одну, включив в нее действия по распознаванию особого случая.

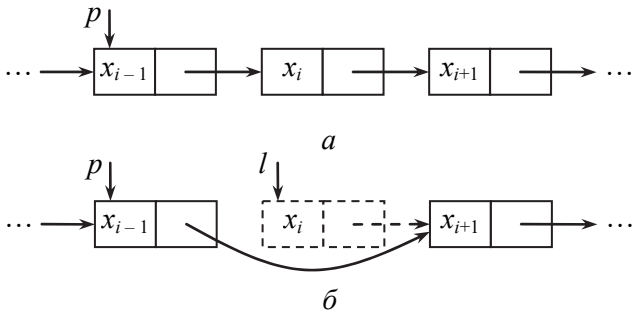


Рис. 2.3. Исключение элемента из связного списка:
а – до исключения; *б* – после исключения.

```
procedure DELETE(p)              procedure DEL_FIRST(list)
   l ← p.next                       l ← list
   p.next ← l.next                  list ← l.next
   dispose(l)                       dispose(l)
return                           return
```

Алгоритм 2.2. Процедуры исключения элемента из связного списка

Таким образом, операции включения и исключения выполняются за некоторое фиксированное время, не зависящее от размеров множества. Так же легко за фиксированное время выполняются операции конкатенации (сцепления) и разбиения списков. Для сцепления двух списков достаточно установить значение поля *next* последнего элемента первого списка (при этом должен быть обеспечен доступ к последнему элементу, например с помощью специального указателя) равным значению указателя на первый элемент второго списка. Разбиение на два

списка можно выполнить, если обеспечен доступ к узлу, непосредственно предшествующему месту разбиения.

Основным достоинством связного распределения является их удобство для реализации динамических множеств, поскольку большинство операций, ориентированных на модификации множеств, выполняются за время, не зависящее от их размеров.

Существенным недостатком связного распределения является невозможность прямого доступа к элементу множества (за исключением первого), т. е. возможен только последовательный доступ. Например, если необходимо получить доступ к элементу x_i, следует последовательно, начиная с первого элемента, перемещаться по $i - 1$ узлам списка, пока не будет достигнут узел с элементом x_i. Это требует времени в среднем $O(n)$. Другой недостаток связан с тем, что необходима дополнительная память для хранения указателей.

2.2.2. Реализация связных списков

В таких языках программирования, как Паскаль, предусмотрены специальные средства для реализации объектов, имеющих несколько полей (комбинированные типы), и указателей (ссылочные типы) с соответствующими механизмами выделения и освобождения памяти. Однако это не единственный способ реализации связных списков. Иногда (либо язык не имеет соответствующих средств, либо алгоритм становится более эффективным) для реализации указателей может оказаться полезным использование массивов. В этом случае в качестве указателя используется индекс в массиве. Пустому значению Λ указателя должно соответствовать число, не являющееся индексом никакого элемента массива (например, 0). В приводимых ниже примерах доступ к элементу массива осуществляется путем указания имени массива и в квадратных скобках индекса элемента.

Пример представления с помощью массивов двух связных списков $X = \{a, b, c\}$ и $Y = \{d, e, g\}$ показан на рисунке 2.4. В массиве *info* хранятся элементы множества, а в массиве *next* – указатели, т. е. индексы позиций в массивах, где расположены последующие элементы. Поскольку при таком представлении нет встроенных механизмов выделения и освобождения памяти, они должны быть предусмотрены при реализации. Чтобы вести

учет использования позиций массивов, все свободные позиции объединены в отдельный связный *список свободных позиций* с указателем *free*, который представляет собой индекс первой свободной позиции в списке. Данный список может обеспечивать обслуживание нескольких связных списков, имеющих одинаковый тип элементов.

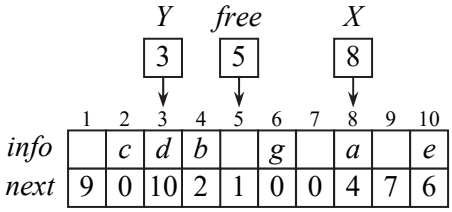


Рис. 2.4. Представление связных списков с помощью массивов

Процедура выделения памяти *new* (*p*) для обеспечения включения нового элемента в список сначала должна проверить наличие свободных позиций (если *free* = 0, свободных позиций нет). Затем, если имеются свободные позиции (*free* ≠ 0), определить индекс первой из них (установив значение указателя *p* = *free*) и исключить эту позицию из списка свободных позиций (присвоив указателю *free* значение *next* [*free*]). Например, пусть требуется включить элемент *f* в список *Y* после элемента *e*, хранящегося в *info* [10]. Тогда процедура *new* (*p*) установит указатель *p* = 5 и, установив указатель *free* = *next* [*free*] = 1, исключит позицию 5 из списка свободных позиций. Операция включения запишет элемент *f* в *info* [5] и установит значения *next* [5] = *next* [10] = 6 и *next* [10] = 5 (рис. 2.5).

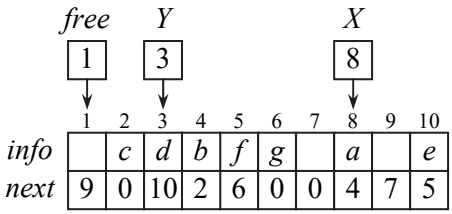


Рис. 2.5. Включение элемента *f* в список *Y*

Процедура освобождения памяти *dispose* (*p*) позицию *p*, которая стала свободной в результате исключения элемента из

списка, включает в начало списка свободных позиций, записав значение указателя *free* в позицию *p* массива *next* и установив новое значение указателя *free* = *p*. В качестве примера рассмотрим исключение элемента *e* из списка $Y = \{d, e, f, g\}$ (рис. 2.5). Исключаемый элемент *e* хранится в *info* [10], а предшествующий ему элемент *d* – в *info* [3]. Операция исключения устанавливает значение *next* [3] = *next* [10] = 5 и передает процедуре *dispose* (*p*) значение *p* = 10 для освобождения памяти, которая включает эту позицию в начало списка свободных позиций, установив значения *next* [10] = *free* = 1 и *free* = 10 (рис. 2.6). Хотя исключенный элемент *e* по-прежнему присутствует в *info* [10], в списке *Y* его уже нет, так как позиция 10 находится в списке свободных позиций.

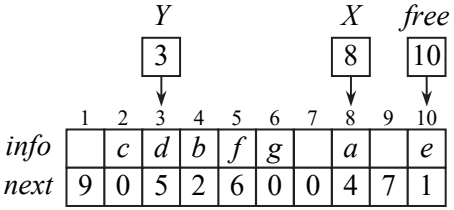


Рис. 2.6. Исключение элемента *e* из списка *Y*

Таким образом, список свободных позиций имеет одну точку доступа: новый элемент всегда добавляется в начало, исключается всегда первый элемент, т. е. список ведет себя как стек.

Вместо нескольких массивов, когда каждому полю узла связного списка соответствует свой массив, можно использовать один массив, элементами которого являются объекты комбинированного типа (записи). Каждая запись представляет собой узел списка и состоит из тех же полей.

2.2.3. Разновидности связных списков

Существуют различные виды связных списков, ориентированных на облегчение выполнения тех или иных операций.

Можно поместить в начало связного списка дополнительный фиктивный элемент (называемый *заголовком списка*), имеющий ту же структуру, что и остальные узлы списка (рис. 2.7). Поле *next* заголовка указывает на узел с первым элементом множества, поле *info* не содержит элемента множества. На практике поле

info заголовка (если позволяет его структура) можно использовать для хранения некоторой информации о множестве (например, его размер). Пустой список представляется не нулевым значением указателя *list*, а состоит из одного заголовка, поле *next* которого равно Λ, т. е. критерием пустоты списка является условие *list.next* = Λ, а указатель *list* никогда не будет иметь нулевое значение.

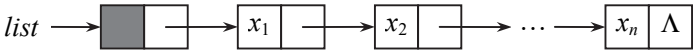


Рис. 2.7. Связный список с заголовком

Такая модификация связного списка упрощает выполнение операций включения и исключения, поскольку отпадает необходимость выявления особых случаев, связанных с включением элемента в начало списка (новый узел добавляется после заголовка) и исключением первого элемента списка (удаляется узел, непосредственно следующий за заголовком). Таким образом, для реализации этих операций достаточно использовать только процедуры *INSERT* и *DELETE*.

Другой разновидностью связного списка является *циклический* список, в котором поле *next* последнего элемента содержит указатель на первый элемент, а не пустое значение (рис. 2.8).

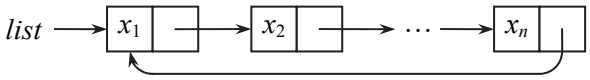


Рис. 2.8. Циклический список

Такой список дает возможность достигнуть любого элемента (хотя и не прямым доступом) из любого другого элемента списка. Операции включения и исключения реализуются так же, как и в нециклических списках. Особыми случаями являются включение в пустой список и исключение элемента из списка, если этот элемент является единственным (после его исключения список становится пустым). Другое удобство заключается в следующем. Если указатель *list* списка установить так, чтобы он указывал на последний элемент, то *list.next* будет указывать на первый элемент, т. е. одним указателем *list* обеспечивается возможность быстрого доступа не только к последнему, но и к первому элементу списка. Как и в обычный список, в циклический список при необходимости можно добавить заголовок.

В тех случаях, когда возникает необходимость в эффективном перемещении по списку как в прямом, так и обратном направлениях, удобно использовать *двусвязные списки*. В таких списках каждому элементу x_i вместо одного указателя ставятся в соответствие два: один из них указывает на следующий элемент x_{i+1}, а другой – на предшествующий элемент x_{i-1}, т. е. в структуру узла добавляется поле *prev*, указывающее на предшествующий узел (рис. 2.9). В двусвязных списках имеется возможность прямого доступа к предшествующему и последующему элементам, поэтому удобно выполнять операции включения нового элемента перед заданным элементом x_i и исключения элемента x_i без предварительной установки указателя на предшествующий узел с элементом x_{i-1}. Все эти возможности достигаются за счет дополнительных затрат памяти и усложнения основных операций со списками. При необходимости в двусвязный список можно добавить заголовок. Двусвязный список можно сделать также циклическим, если поле *next* последнего элемента будет указывать на первый элемент списка, а поле *prev* первого элемента – на последний элемент списка.

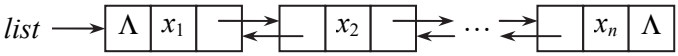


Рис. 2.9. Двусвязный список

Использование той или иной разновидности связных списков позволяет упростить выполнение одних операций со списками, но усложнить другие. Поэтому выбор представления должен осуществляться на основе анализа типов операций, которые будут выполняться над множеством. Реализация операций для различных видов связных списков предлагается в качестве упражнений для самостоятельной разработки.

2.3. Стеки

Стек представляет собой динамическую последовательность элементов с одной точкой доступа, называемой *вершиной* стека. Все элементы последовательности, кроме элемента, расположенного в вершине стека, недоступны. Новый элемент добавляется только в вершину стека, сдвигая остальные элементы последовательности. Исключить можно только элемент из вершины стека,

остальные элементы последовательности при этом сдвигаются в сторону вершины стека. Таким образом, стек работает по принципу «последним пришел – первым ушел» и часто называется структурой LIFO (Last In – First Out).

Согласно этому определению зафиксирована вершина стека, а перемещаются его элементы. Очевидно, что прямая реализация такого стека неэффективна, так как время выполнения операций включения и исключения будет пропорционально числу элементов в стеке. Поэтому при реализации стека избегают перемещения его элементов, а делают передвигаемой вершину стека. Для этого используется специальный *указатель вершины стека*, который указывает текущее положение вершины стека.

Операцию включения элемента в стек будем записывать в виде $S \Leftarrow x$ (элемент x поместить в вершину стека S), а операцию исключения из стека – в виде $x \Leftarrow S$ (исключить элемент из вершины стека S и присвоить его значение переменной x). Использование данных обозначений позволяет рассматривать логику работы алгоритма, не обращая внимания на детали реализации стека. В литературе эти операции традиционно называют *PUSH* (включение) и *POP* (исключение). При этом операцию включения элемента x в стек S обычно реализуют в виде процедуры $PUSH(x, S)$, а исключение – в виде функции $POP(S)$, которая удаляет элемент из вершины стека S и возвращает его в качестве своего значения (т. е. операции $x \Leftarrow S$ соответствует присваивание $x \leftarrow POP(S)$).

Одним из наиболее эффективных способов реализации стека является использование последовательного распределения (массива). Массив служит для хранения элементов стека. В качестве указателя вершины стека используется переменная t, являющаяся индексом последнего включенного в стек элемента. Стек состоит из элементов $S_1, S_2, \ldots, S_t$, где S_1 – нижний элемент (дно) стека, а S_t – верхний элемент (вершина) стека. Емкость (глубина) стека определяется размером массива. Пусть для реализации стека выделен массив из m компонентов, тогда всегда должно выполняться условие $t \le m$, в противном случае при включении элемента происходит переполнение, что означает ошибку. Пустой стек соответствует случаю $t = 0$ и, как правило, означает завершение работы алгоритма в целом или некоторого его фраг-

мента. Реализация операций включения и исключения представлена в алгоритме 2.3.

$S \Leftarrow x$	$x \Leftarrow S$
$t \leftarrow t+1$ **if** $t > m$ **then** // переполнение **else** $S_t \leftarrow x$	**if** $t = 0$ **then** // стек пуст **else** $\begin{cases} x \leftarrow S_t \\ t \leftarrow t-1 \end{cases}$

Алгоритм 2.3. Операции включения и исключения для стека на базе массива

На рисунке 2.10 показан пример реализации стека на базе массива (емкость стека равна 8) и процесс изменения его состояния во времени. Пусть стек находится в некотором состоянии *а*, когда он содержит 4 элемента, вершина стека – элемент 9 (рис. 2.10*а*). После выполнения операций включения $S \Leftarrow 3$ и $S \Leftarrow 8$ стек переходит в состояние *б*, вершиной стека становится элемент 8 (рис. 2.10*б*). После исключения элемента 8 из вершины стека он переходит из состояния *б* в состояние *в*, вершиной стека становится элемент 3 (рис. 2.10*в*). Хотя после исключения элемента 8 он по-прежнему присутствует в массиве, в стеке его уже нет, так как вершиной стека является элемент 3.

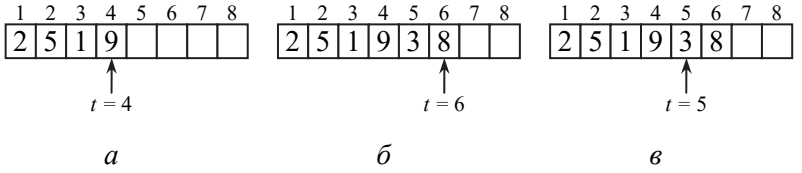


Рис. 2.10. Реализация стека на базе массива:
а – стек содержит 4 элемента, вершина стека – элемент 9;
б – стек после выполнения операций $S \Leftarrow 3$ и $S \Leftarrow 8$;
в – стек после исключения элемента 8 из вершины.

Недостатком последовательной реализации стека является фиксированная емкость стека, что может привести к его переполнению. Этот недостаток является следствием противоречия между массивом как статической структурой и стеком как динамической структурой. От указанного недостатка свободна реали-

зация стека с использованием связного распределения (динамических списковых структур).

Стек легко реализуется на базе односвязного списка, узел которого состоит из поля *info*, содержащего элемент последовательности, и поля связи *next* для ссылки на элемент, находящийся в стеке под данным элементом. Поле *next* нижнего элемента стека имеет пустое значение Λ указателя. Функции указателя t вершины стека выполняет указатель *list* связного списка. Стек пуст, если $t = \Lambda$. Очевидно, что операции $S \Leftarrow x$ соответствует процедура *INS_FIRST* включения элемента в начало списка, а операции $x \Leftarrow S$ – процедура *DEL_FIRST* исключения первого элемента списка, модифицированная так, чтобы переменной x присваивалось значение исключаемого элемента. Реализация этих операций представлена в алгоритме 2.4.

$S \Leftarrow x$	$x \Leftarrow S$
$new(l)$ $l.info \leftarrow x$ $l.next \leftarrow t$ $t \leftarrow l$	**if** $t = \Lambda$ **then** // стек пуст **else** $\begin{cases} x \leftarrow t.info \\ l \leftarrow t \\ t \leftarrow t.next \\ dispose(l) \end{cases}$

Алгоритм 2.4. Операции включения и исключения для стека на базе списка

Для многих алгоритмов проверку пустоты стека удобно выделить в отдельную операцию (будем записывать в виде $S = \varnothing$). В этом случае из операции $x \Leftarrow S$ исключается проверка пустоты стека и считается, что она определена только для непустого стека. Операцию, которая устанавливает стек S в начальное состояние (т. е. делает его пустым), будем записывать в виде $S \leftarrow \varnothing$. В ряде случаев множество операций, выполняемых над стеком, дополняется операцией определения значения элемента в вершине стека без его исключения. Иногда может быть полезна операция, которая просто исключает элемент из стека, не возвращая его значение. Эти операции являются простыми модификациями операции исключения и реализуются очевидным образом. Все рассмотренные операции со стеком выполняются за время $O(1)$.

2.4. Очереди

Очередь представляет собой динамическую последовательность элементов с двумя точками доступа: начало (голова) и конец (хвост). Новый элемент добавляется всегда в конец очереди, исключается всегда элемент, расположенный в начале очереди. Все остальные элементы очереди недоступны. Таким образом, очередь работает по принципу «первым пришел – первым ушел» и часто называется структурой FIFO (First In – First Out).

Операцию включения элемента в очередь будем записывать в виде $Q \Leftarrow x$ (элемент x поместить в конец очереди Q), а операцию исключения из очереди – в виде $x \Leftarrow Q$ (исключить элемент из начала очереди Q и присвоить его значение переменной x).

Последовательная реализация очереди требует специальных приемов. Это связано с тем, что очередь растет на одном конце и убывает на другом, т. е. двигается в сторону правой границы массива и может ее перейти, хотя в левой части массива может быть достаточно места для размещения элементов очереди. Чтобы этого избежать, необходимо массив свернуть в кольцо. Для этого удобно использовать операцию **mod** (вычисление остатка от целочисленного деления). В этом случае для очереди Q выделяется массив из m компонентов $Q_0, Q_1, Q_2, \ldots, Q_{m-1}$ и в соответствии с операцией **mod** считается, что Q_0 следует за Q_{m-1}. Если использовать переменную f в качестве указателя позиции в массиве, расположенной непосредственно перед началом очереди, а переменную r в качестве указателя ее конца, то очередь будет состоять из элементов $Q_{f+1}, Q_{f+2}, \ldots, Q_r$. Пустая очередь будет соответствовать случаю $r = f$.

Реализация операций включения и исключения представлена в алгоритме 2.5.

$Q \Leftarrow x$	$x \Leftarrow Q$
$r \leftarrow (r+1)$ **mod** m **if** $r = f$ **then** // переполнение **else** $Q_r \leftarrow x$	**if** $r = f$ **then** // очередь пуста **else** $\begin{cases} f \leftarrow (f+1) \textbf{ mod } m \\ x \leftarrow Q_f \end{cases}$

Алгоритм 2.5. Операции включения и исключения для очереди на базе массива

Переполнение означает ошибку, а пустота очереди – окончание работы алгоритма (или некоторого его фрагмента). Следует обратить внимание на то, что переполнение появляется, когда к очереди из $m - 1$ элементов добавляется m-й элемент, поэтому один из m компонентов, отведенных для хранения элементов очереди, остается пустым (емкость очереди равна $m - 1$).

На рисунке 2.11 показан пример реализации очереди на базе массива (емкость очереди равна 7) и процесс изменения ее состояния во времени. Пусть очередь находится в некотором состоянии *а*, когда она содержит 4 элемента, начало очереди – элемент 2, конец очереди – элемент 9 (рис. 2.11*а*). После выполнения операций включения $Q \Leftarrow 3$ и $Q \Leftarrow 8$ очередь переходит в состояние *б*, началом очереди остается элемент 2, концом очереди становится элемент 8 (рис. 2.11*б*). После исключения элемента 2 из начала очереди она переходит из состояния *б* в состояние *в*, началом очереди становится элемент 5, концом очереди остается элемент 8 (рис. 2.11*в*). Хотя после исключения элемента 2 он по-прежнему присутствует в массиве, в очереди его уже нет, так как началом очереди является элемент 5.

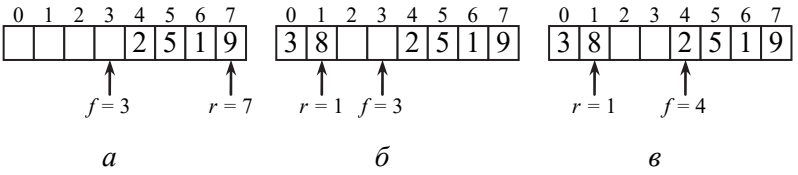


Рис. 2.11. Реализация очереди на базе массива:
а – очередь содержит 4 элемента, начало – элемент 2, конец – элемент 9;
б – очередь после выполнения операций $Q \Leftarrow 3$ и $Q \Leftarrow 8$;
в – очередь после исключения элемента 2 из начала очереди.

Недостатком последовательной реализации очереди является фиксированная емкость очереди, что может привести к переполнению. От указанного недостатка свободна реализация очереди с использованием связного распределения.

Очередь можно реализовать на базе односвязного списка, узел которого состоит из поля *info*, содержащего элемент очереди, и поля связи *next* для указания элемента последовательности, идущего после данного. Функции указателя *f* начала очереди выполняет внешний указатель *list* связного списка. Требуется также

внешний указатель r конца очереди, который ссылается на последний элемент списка. Очередь пуста, если $f = \Lambda$. Операции $Q \Leftarrow x$ соответствует процедура *INSERT* включения элемента в список, при этом поскольку включение производится всегда в конец связного списка, присваивание $l.next \leftarrow p.next$ можно заменить на $l.next \leftarrow \Lambda$. Необходимо учитывать также особый случай, когда элемент включается в пустую очередь, поскольку в этом случае требуется установка указателя f на начало очереди. Операции $x \Leftarrow Q$ соответствует процедура *DEL_FIRST* исключения первого элемента списка, модифицированная так, чтобы переменной x присваивалось значение исключаемого элемента. При этом следует учитывать особый случай, когда исключается элемент из очереди, состоящей из этого единственного элемента, поскольку в результате исключения очередь становится пустой и, следовательно, значение указателя r конца очереди должно быть нулевым.

Реализация очереди упрощается, если использовать для ее представления связный список с заголовком, поскольку в этом случае нет необходимости в выявлении особых случаев. При таком представлении очереди поле *info* заголовка не используется, поле *next* заголовка указывает на первый элемент очереди, указатель f – на заголовок списка, указатель r – на последний элемент очереди. Пустая очередь состоит из одного узла-заголовка, на который ссылаются указатели f и r, и значение его поля *next* равно Λ. В качестве критерия пустоты очереди можно использовать либо условие $f.next = \Lambda$, либо условие $f = r$. Соответствующие операции включения и исключения представлены в алгоритме 2.6.

$Q \Leftarrow x$	$x \Leftarrow Q$
$new(r.next)$ $r \leftarrow r.next$ $r.info \leftarrow x$ $r.next \leftarrow \Lambda$	**if** $f = r$ **then** // очередь пуста **else** $\begin{cases} x \leftarrow f.next.info \\ l \leftarrow f \\ f \leftarrow f.next \\ dispose(l) \end{cases}$

Алгоритм 2.6. Операции включения и исключения для очереди на базе списка

Для многих алгоритмов проверку пустоты очереди целесообразно выделить в отдельную операцию (будем записывать в виде $Q = \varnothing$). В этом случае из операции $x \Leftarrow Q$ исключается проверка пустоты очереди и считается, что она определена только для непустой очереди. Операцию, которая устанавливает очередь Q в начальное состояние (т. е. делает ее пустой), будем записывать в виде $Q \leftarrow \varnothing$. Все операции с очередью выполняются за время $O(1)$.

2.5. Деревья

2.5.1. Основные определения

Конечное *корневое дерево* T формально определяется как связный ориентированный ациклический граф, удовлетворяющий следующим условиям:

1) имеется точно одна вершина, называемая *корнем*, в которую не входит ни одно ребро;

2) в каждую вершину (за исключением корня) входит ровно одно ребро;

3) из корня к каждой вершине существует путь (последовательность ребер), причем единственный.

Ориентированный граф, состоящий из нескольких деревьев, называется *лесом*.

Обычно считается, что в дереве все ребра имеют направление от корня, поэтому при графическом изображении ориентацию ребер можно не указывать. В описании соотношений между вершинами дерева используется терминология, принятая в генеалогических деревьях. Пусть v – произвольная вершина дерева с корнем r. Все вершины, входящие в единственный путь из r в v, называются *предками* вершины v. Если вершина w является предком вершины v, то вершина v называется *потомком* вершины w. Для каждой вершины w все ее потомки образуют *поддерево*, корнем которого является вершина w. Если (v, w) – ребро дерева, то вершину v называют *отцом* вершины w, а вершину w – *сыном* вершины v. Число сыновей вершины называется ее *степенью*. Дерево, степень каждой вершины которого не превышает некоторого целого m, называется *m-арным*. Вершины, не имеющие сыновей, называются *листьями*. Вершины, имеющие сыновей (которых будем называть *братьями*), называются *внутрен-*

ними. Таким образом, в корневом дереве все вершины являются потомками его корня, и наоборот, корень есть предок всех своих потомков.

Длина пути от корня до произвольной вершины *v* называется *уровнем* (*глубиной*) вершины *v* и определяется числом ребер в этом пути. Следовательно, уровень корня дерева равен нулю, а уровень любой другой вершины имеет уровень на единицу больше уровня своего отца. *Высота* вершины есть длина самого длинного пути от этой вершины до какого-нибудь листа. *Высотой дерева* называется высота его корня. Другими словами, высота дерева равна максимальному из уровней его листьев.

Введенные понятия проиллюстрированы на рисунке 2.12. Лес состоит из двух деревьев с корнями *a* и *n*. Листьями являются вершины *c*, *d*, *g*, *h*, *i*, *l*, *m*, *p*, *q*, *r*. Вершина *a* является отцом вершин *b*, *e* и *j*, которые являются сыновьями вершины *a* и братьями по отношению друг к другу. Высоты деревьев равны соответственно 3 и 2. Уровень вершины *j* равен 1, а высота равна 2.

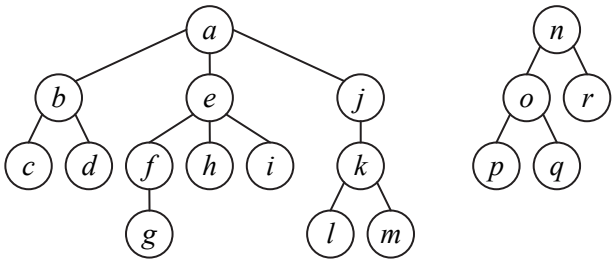
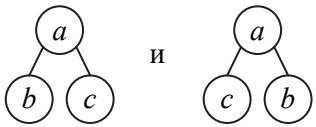


Рис. 2.12. Лес, состоящий из двух деревьев

Все рассматриваемые деревья будем считать упорядоченными, т. е. для них важен относительный порядок поддеревьев каждой вершины. Обычно предполагается, что в упорядоченном дереве множество сыновей каждой вершины упорядочено слева направо. Поэтому деревья



считаются различными.

Важной разновидностью корневых деревьев является класс *бинарных* (*двоичных*) *деревьев*. Бинарное дерево T либо пустое, либо состоит из корня и двух бинарных поддеревьев: левого T_l и правого T_r. Все вершины в поддереве T_l расположены левее всех вершин в поддереве T_r. Таким образом, в бинарных деревьях каждая вершина имеет не более двух сыновей, причем каждый сын интерпретируется либо как *левый* сын, либо как *правый* сын. Поэтому два следующих дерева

представляют собой два различных бинарных дерева. Это является существенным отличием от других деревьев. В бинарном дереве важно, каким является единственный сын вершины (левым или правым), для других деревьев такого различия нет.

2.5.2. Представления деревьев

Большинство представлений множеств с помощью деревьев как структур данных основано на связных распределениях. Каждая вершина представляет собой узел, состоящий из поля *info* и нескольких полей для хранения указателей. Такое представление называется *узловым*.

Проще всего задача представления решается для бинарных деревьев. Для этого используются узлы фиксированного размера, состоящие из полей *left* (для хранения указателя на левого сына), *info* и *right* (для хранения указателя на правого сына). Если у узла, на который ссылается указатель p, нет левого или правого сына, то $p.left = \Lambda$ или $p.right = \Lambda$. С бинарным деревом связан специальный внешний указатель *root* (*указатель дерева*), который указывает на его корень. Если $root = \Lambda$, то дерево пустое. Пример узлового представления бинарного дерева с корнем n из рисунка 2.12 показан на рисунке 2.13. При необходимости в ряде прикладных алгоритмов можно добавить указатель отца *father* для облегчения движения от потомков к предкам.

Что касается деревьев с произвольным ветвлением, то задача их представления решается сложнее.

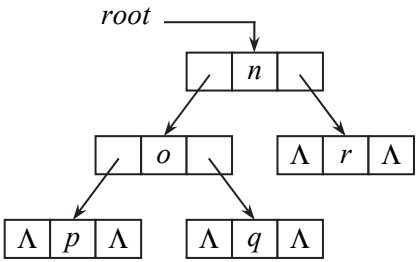


Рис. 2.13. Узловое представление бинарного дерева

Один из способов представления заключается в следующем: все узлы имеют фиксированный размер и состоят из поля *info* и поля связи *father*, указывающего на отца узла. При этом для доступа к узлам дерева необходим набор внешних указателей для каждого узла или, по крайней мере, для каждого листа. Пример такого представления дерева с корнем *a* из рисунка 2.12 показан на рисунке 2.14 (внешние указатели не показаны). Рассмотренный способ представления полезен, если необходимо подниматься по дереву от потомков к предкам. Такая операция встречается редко; чаще требуется движение от предков к потомкам.

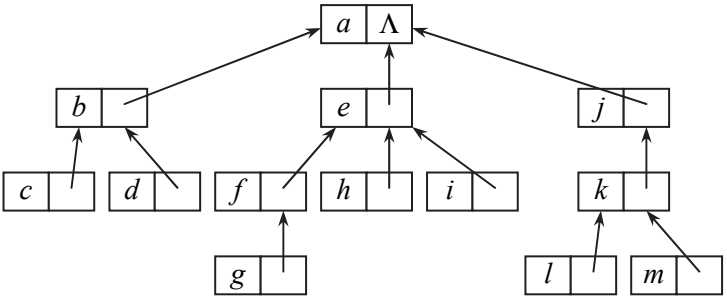


Рис. 2.14. Представление дерева с помощью узлов с полем *father*

Представление дерева с использованием указателей, ведущих от предков к потомкам, создает определенные трудности. Если известно, что число сыновей каждой вершины ограничено сверху константой *k*, то дерево можно реализовать аналогично бинарному дереву, помещая в узел *k* указателей на сыновей. Проблема заключается в том, что если у большинства вершин

число сыновей существенно меньше k, то бесполезно тратится большой объем памяти. Другая проблема заключается в том, что такая реализация невозможна, если число сыновей может быть любым и неизвестна верхняя граница k (заранее неизвестно, сколько полей для указателей необходимо выделять).

Решением указанных проблем является метод преобразования произвольного дерева (леса) в бинарное. Такое преобразование называется *естественным соответствием* между лесами и бинарными деревьями и заключается в следующем. Поле *left* бинарного дерева предназначается для указания самого левого сына данной вершины, а поле *right* – для указания следующего брата данной вершины. Другими словами, поле *left* каждого узла указывает на связный список сыновей этого узла; список связывается с помощью полей *right*. Пример такого представления леса (см. рис. 2.12) показан на рисунке 2.15 (для упрощения не показаны поля узлов дерева).

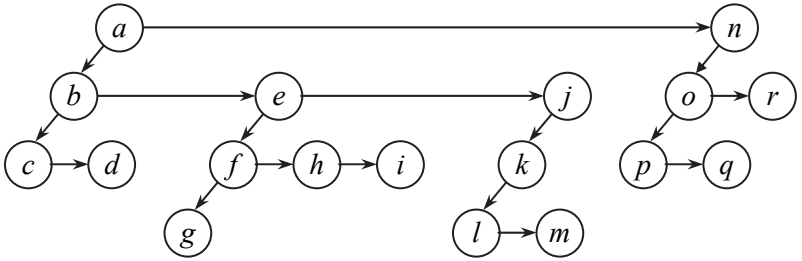


Рис. 2.15. Бинарное дерево, соответствующее лесу на рисунке 2.12

Возможны и другие представления деревьев, конкретный выбор представления определяется спецификой задачи.

2.5.3. Прохождения деревьев

Во многих приложениях необходимо пройти лес, обрабатывая вершины каким-либо способом (в простейшем случае – печать содержимого узла) в некотором систематическом порядке. Предполагается, что при посещении вершин структура леса не меняется. Рассмотрим четыре основных способа прохождения: в прямом, обратном, горизонтальном и симметричном (обычно определен только для бинарных деревьев) порядках.

При прохождении в *прямом* порядке (известном также как прохождение *в глубину*) вершины леса просматриваются в соответствии со следующей рекурсивной процедурой:

1) посетить корень первого дерева;

2) пройти в прямом порядке поддеревья первого дерева, если они есть;

3) пройти в прямом порядке оставшиеся деревья, если они есть.

Для леса на рисунке 2.12 вершины будут проходиться в следующем порядке: *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*, *n*, *o*, *p*, *q*, *r*.

Для бинарных деревьев эта процедура упрощается и выглядит следующим образом (пустое дерево проходится без выполнения каких-либо действий):

1) посетить корень;

2) пройти в прямом порядке левое поддерево;

3) пройти в прямом порядке правое поддерево.

Следует обратить внимание на то, что прохождение леса в прямом порядке в точности соответствует прямому прохождению бинарного дерева, являющегося его представлением.

При *обратном* прохождении (известном также как прохождение *снизу вверх*) вершины леса проходятся в соответствии со следующей рекурсивной процедурой:

1) пройти в обратном порядке поддеревья первого дерева, если они есть;

2) посетить корень первого дерева;

3) пройти в обратном порядке оставшиеся деревья, если они есть.

Следует обратить внимание на то, что в момент посещения произвольной вершины все ее потомки уже пройдены. Для леса, изображенного на рисунке 2.12, вершины проходятся в следующем порядке: *c*, *d*, *b*, *g*, *f*, *h*, *i*, *e*, *l*, *m*, *k*, *j*, *a*, *p*, *q*, *o*, *r*, *n*.

Для бинарных деревьев процедура имеет следующий вид:

1) пройти в обратном порядке левое поддерево;

2) пройти в обратном порядке правое поддерево;

3) посетить корень.

Симметричный порядок прохождения бинарных деревьев определяется следующим образом:

1) пройти в симметричном порядке левое поддерево;

2) посетить корень;

3) пройти в симметричном порядке правое поддерево.

Для бинарного дерева на рисунке 2.13 вершины будут проходиться в следующем порядке: *p*, *o*, *q*, *n*, *r*. Следует обратить внимание на то, что обратный порядок прохождения леса эквивалентен симметричному порядку прохождения соответствующего этому лесу бинарного дерева.

Рекурсивные процедуры прохождения бинарных деревьев очевидным образом можно записать в виде соответствующего псевдокода, например рекурсивная процедура прямого прохождения представлена алгоритмом 2.7.

procedure $PREORDER(p)$

$$\textbf{if}\ \ p \neq \Lambda\ \ \textbf{then} \begin{cases} visit(p) \\ PREORDER(p.left) \\ PREORDER(p.right) \end{cases}$$

return

Алгоритм 2.7. Рекурсивная процедура прямого прохождения бинарного дерева

Прохождение леса в *горизонтальном* порядке (известном также как прохождение *в ширину*) заключается в следующем. Вершины леса проходятся слева направо уровень за уровнем от корня вниз. Для леса, показанного на рисунке 2.12, вершины проходятся так: *a*, *n*, *b*, *e*, *j*, *o*, *r*, *c*, *d*, *f*, *h*, *i*, *k*, *p*, *q*, *g*, *l*, *m*.

Если все вершины дерева пронумеровать в порядке посещения, то рассмотренные прохождения обладают рядом интересных свойств.

При нумерации в прямом порядке все вершины поддерева с корнем *r* имеют номера, не меньшие *r*. Если D_r – множество потомков вершины *r* (включая и саму вершину *r*), то *v* будет номером некоторой вершины из D_r тогда и только тогда, когда $r \le v < r + |D_r|$. Поставив в соответствие каждой вершине *v* ее номер в прямом порядке и количество ее потомков, легко определить, является ли некоторая вершина *w* потомком для *v*, за фиксированное время, не зависящее от размера дерева.

Номера, соответствующие обратному порядку, обладают аналогичным свойством.

Номера вершин бинарного дерева, соответствующие симметричному порядку, обладают тем свойством, что номера вершин в левом поддереве для вершины v меньше v, а в правом поддереве – больше v.

При сравнении рекурсивных процедур прохождения бинарных деревьев обнаруживается значительное их сходство. Это сходство позволяет построить общий нерекурсивный алгоритм, который может быть применен к каждому из этих прохождений. Для этого используется стек S для хранения пар, состоящих из указателя на узел бинарного дерева и целого i, значение которого указывает номер применяемой операции, когда пара достигнет вершины стека. Анализ процедур прохождения позволяет выделить три типа операций: посетить корень ($visit(p)$, где p – указатель на текущий узел), перейти к левому поддереву, что соответствует операции

$$\textbf{if } p.left \neq \Lambda \textbf{ then } S \Leftarrow (p.left, 1),$$

и перейти к правому поддереву, что соответствует операции

$$\textbf{if } p.right \neq \Lambda \textbf{ then } S \Leftarrow (p.right, 1).$$

Тогда общую процедуру прохождения бинарного дерева можно представить алгоритмом 2.8 [21].

$$
\begin{array}{l}
S \leftarrow \varnothing \quad \text{// пустой стек} \\
S \Leftarrow (root, 1) \\
\textbf{while } S \neq \varnothing \textbf{ do} \left\{ \begin{array}{l}
(p, i) \Leftarrow S \\
\textbf{case} \left\{ \begin{array}{l}
i = 1: \left\{ \begin{array}{l} S \Leftarrow (p, 2) \\ \text{операция 1} \end{array} \right. \\
i = 2: \left\{ \begin{array}{l} S \Leftarrow (p, 3) \\ \text{операция 2} \end{array} \right. \\
i = 3: \text{ операция 3}
\end{array} \right.
\end{array} \right.
\end{array}
$$

Алгоритм 2.8. Общий нерекурсивный алгоритм прохождения бинарного дерева

Непосредственная конкретизация общего нерекурсивного алгоритма для прохождения бинарного дерева в прямом порядке приведет к алгоритму 2.9. Очевидно, что полученный алгоритм неэффективен. В частности, после прохождения узла p (узла, на который указывает указатель p), когда $(p, 2)$ или $(p, 3)$ попадают в вершину стека, единственное, что происходит, – это $(p.left, 1)$

или ($p.right$, 1) помещаются в стек. Эти шаги можно сделать раньше, когда в первый раз посещается узел p; тогда отпадает необходимость трехкратного включения в стек указателя на каждый узел и, следовательно, сохранения в стеке номера i выполняемой операции. Поэтому этот алгоритм можно существенно упростить (алгоритм 2.10).

$$
\begin{array}{l}
S \leftarrow \varnothing \quad \text{// пустой стек} \\
S \Leftarrow (root, 1) \\
\textbf{while}\ \ S \neq \varnothing\ \ \textbf{do}
\begin{cases}
(p, i) \Leftarrow S \\
\textbf{case}
\begin{cases}
i = 1: \begin{cases} S \Leftarrow (p, 2) \\ visit(p) \end{cases} \\
i = 2: \begin{cases} S \Leftarrow (p, 3) \\ \textbf{if}\ \ p.left \neq \Lambda\ \textbf{then}\ \ S \Leftarrow (p.left, 1) \end{cases} \\
i = 3:\ \textbf{if}\ \ p.right \neq \Lambda \\
\qquad\qquad \textbf{then}\ \ S \Leftarrow (p.right, 1)
\end{cases}
\end{cases}
\end{array}
$$

Алгоритм 2.9. Конкретизация общего алгоритма для прямого прохождения

$$
\begin{array}{l}
S \leftarrow \varnothing \quad \text{// пустой стек} \\
S \Leftarrow root \\
\textbf{while}\ \ S \neq \varnothing\ \ \textbf{do}
\begin{cases}
p \Leftarrow S \\
visit(p) \\
\textbf{if}\ \ p.right \neq \Lambda\ \ \textbf{then}\ \ S \Leftarrow p.right \\
\textbf{if}\ \ p.left \neq \Lambda\ \ \textbf{then}\ \ S \Leftarrow p.left
\end{cases}
\end{array}
$$

Алгоритм 2.10. Упрощенный вариант алгоритма 2.9

Для полученного алгоритма можно продолжить процесс улучшений. В алгоритме указатель на каждый узел помещается в стек точно один раз. Однако можно сократить число операций со стеком. Если внимательно проанализировать процедуру прямого прохождения, то можно обнаружить, что в стек достаточно помещать только указатели на правых сыновей (если они есть), а по левым сыновьям продолжать продвижение, используя для этого рабочий указатель p (алгоритм 2.11).

Конкретизировав соответствующим образом общий алгоритм и упростив его, можно получить нерекурсивный алгоритм симметричного прохождения бинарного дерева (алгоритм 2.12).

$S \Leftarrow \Lambda$ // занести в стек пустое значение указателя
$p \leftarrow root$

$$\textbf{while}\ \ p \neq \Lambda\ \ \textbf{do} \begin{cases} visit(p) \\ \textbf{if}\ \ p.left \neq \Lambda\ \ \textbf{then}\ \ S \Leftarrow p.right \\ \textbf{if}\ \ p.left \neq \Lambda\ \ \textbf{then}\ \ p \leftarrow p.left\ \ \textbf{else}\ \ p \Leftarrow S \end{cases}$$

Алгоритм 2.11. Прямое прохождение бинарного дерева

$S \leftarrow \varnothing$ // пустой стек
$p \leftarrow root$

$$\textbf{while}\ \ p \neq \Lambda\ \ \textbf{do} \begin{cases} \textbf{while}\ \ p.left \neq \Lambda\ \ \textbf{do} \begin{cases} S \Leftarrow p \\ p \leftarrow p.left \end{cases} \\ visit(p) \\ \textbf{while}\ \ p.right = \Lambda\ \textbf{and}\ S \neq \varnothing\ \ \textbf{do} \begin{cases} p \Leftarrow S \\ visit(p) \end{cases} \\ p \leftarrow p.right \end{cases}$$

Алгоритм 2.12. Симметричное прохождение бинарного дерева

Для получения нерекурсивного алгоритма обратного прохождения достаточно применить соответствующую прямую конкретизацию общего алгоритма, поскольку существенных улучшений в этом случае добиться трудно.

Для реализации горизонтального прохождения бинарного дерева необходимо использовать очередь. При посещении некоторого узла его сыновья (если они есть) помещаются в очередь: сначала левый сын, а затем – правый. Детали реализации горизонтального прохождения представлены алгоритмом 2.13.

$Q \leftarrow \varnothing$ // пустая очередь
$Q \Leftarrow root$

$$\textbf{while}\ \ Q \neq \varnothing\ \ \textbf{do} \begin{cases} p \Leftarrow Q \\ visit(p) \\ \textbf{if}\ \ p.left \neq \Lambda\ \ \textbf{then}\ \ Q \Leftarrow p.left \\ \textbf{if}\ \ p.right \neq \Lambda\ \ \textbf{then}\ \ Q \Leftarrow p.right \end{cases}$$

Алгоритм 2.13. Горизонтальное прохождение бинарного дерева

2.5.4. Прошитые бинарные деревья

С целью повышения эффективности прохождения бинарных деревьев можно использовать так называемые *прошитые* деревья. В нижней части бинарного дерева с n узлами всегда имеется $n + 1$ полей указателей со значением Λ. Можно воспользоваться этим пространством пустых значений следующим образом. Пустые значения полей *left* заменяются указателями на предшествующий узел при прохождении в симметричном порядке, а пустые значения полей *right* – на последующий узел при прохождении в симметричном порядке. Полученные таким образом связи называются *нитями*, а само бинарное дерево – *симметрично прошитым* бинарным деревом. Ясно, что в узлах должны быть предусмотрены специальные средства, чтобы отличать нити от обычных связей (например, добавить дополнительный разряд к полям *left* и *right*). Пример симметрично прошитого бинарного дерева показан на рисунке 2.16 (нити изображены пунктирными линиями).

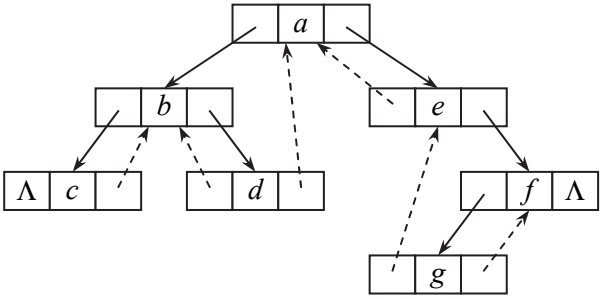


Рис. 2.16. Симметрично прошитое бинарное дерево

Аналогичным образом можно определить *прямопрошитые* бинарные деревья, в которых пустые поля *left* и *right* заменяются указателями соответственно на предшественников и преемников при прямом прохождении.

Наличие нитей в симметрично прошитом дереве позволяет повысить эффективность алгоритмов прохождения в прямом и симметричном порядках за счет исключения операций со стеками, т. е. эти прохождения можно реализовать без использования стека. Построение соответствующих алгоритмов предлагается в качестве упражнений.

2.5.5. Расширенные бинарные деревья

Любое непустое бинарное дерево можно дополнить фиктивными вершинами-листьями так, чтобы все вершины исходного дерева стали внутренними и имели точно по два сына (рис. 2.17). Построенное таким образом дерево называется *расширенным бинарным деревом*, а добавленные фиктивные листья (изображены квадратами) – *внешними вершинами* (или *внешними узлами* при узловом представлении дерева).

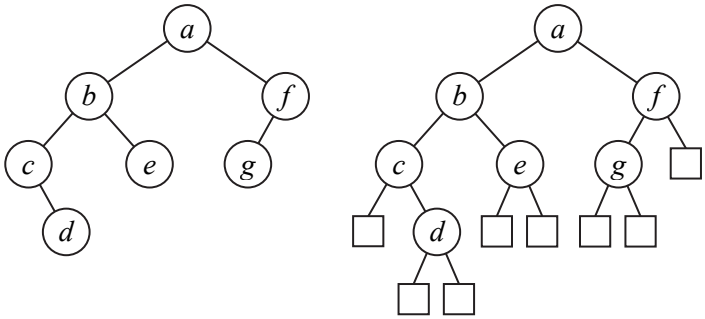


Рис. 2.17. Бинарное дерево и соответствующее ему расширенное дерево

Деревья используются не только как структуры данных для представления множеств или иерархических структур, но и для анализа определенных алгоритмов. При анализе алгоритмов их поведение можно представить в виде так называемых *деревьев решений* (дерево поиска, дерево сортировки, дерево игры и т. п.). Поэтому возникает необходимость в количественных измерениях различных характеристик деревьев.

В расширенном бинарном дереве с n внутренними вершинами всегда существует $n + 1$ внешних вершин. *Длина внешних путей* $E(T)$ расширенного бинарного дерева T с n внутренними вершинами определяется как сумма уровней всех внешних вершин. *Длина внутренних путей* $I(T)$ есть сумма уровней всех внутренних вершин. У расширенного бинарного дерева (рис. 2.17) длина внешних путей $E(T) = 25$, а длина внутренних путей $I(T) = 11$.

Среднее расстояние (т. е. средняя длина) определяется делением длины путей на соответствующее число вершин, т. е. сред-

нее расстояние до внешней вершины равно $E(T)/(n+1)$, а среднее расстояние до внутренней вершины равно $I(T)/n$.

Существует соотношение между длинами внешних и внутренних путей. Для расширенного бинарного дерева

$$E(T) = I(T) + 2n. \qquad (2.1)$$

С точки зрения анализа алгоритмов интересен диапазон значений длин путей. Максимальную длину путей имеют бинарные деревья, вырожденные до связного списка (рис. 2.18). В этом случае расширенное бинарное дерево T с n внутренними вершинами имеет длины

$$I_{\max}(T) = \sum_{i=0}^{n-1} i = \frac{1}{2} n(n-1) \quad \text{и} \quad E_{\max}(T) = \frac{1}{2} n(n+3).$$

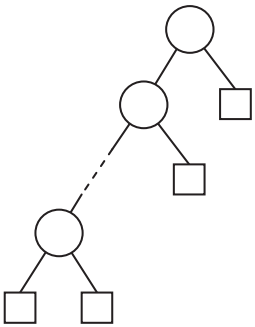


Рис. 2.18. Расширенное бинарное дерево с максимальными длинами путей

Если $l_1, l_2, \ldots, l_{n+1}$ – уровни $n+1$ внешних вершин в расширенном бинарном дереве с n внутренними вершинами, то

$$\sum_{i=1}^{n+1} \frac{1}{2^{l_i}} = 1. \qquad (2.2)$$

Это соотношение легко доказывается методом индукций.

Минимальную длину путей имеют полностью сбалансированные деревья. Бинарное дерево называется *полностью сбалансированным*, если у расширенного дерева все внешние вершины находятся на уровнях l и $l+1$, где l – некоторое целое число. Если все внешние вершины находятся на одном уровне l, то такое дерево называется *полным*.

Определим минимальную длину внешних путей для полностью сбалансированного бинарного дерева с n внутренними и

$n+1$ внешними вершинами. Пусть на уровне l находится k внешних вершин, тогда на уровне $l+1$ будет $n+1-k$ внешних вершин, причем $1 \le k \le n+1$ (при $k = n+1$ все внешние вершины находятся на уровне l).

Из соотношения (2.2) следует, что

$$\frac{k}{2^l} + \frac{n+1-k}{2^{l+1}} = 1.$$

Отсюда

$$k = 2^{l+1} - n - 1. \qquad (2.3)$$

Поскольку $k \le n+1$, то $2^l \le n+1$. Следовательно,

$$l = \lfloor \log(n+1) \rfloor. \qquad (2.4)$$

Объединение (2.3) и (2.4) дает

$$k = 2^{\lfloor \log(n+1) \rfloor + 1} - n - 1.$$

Таким образом, минимальная длина внешних путей

$$E_{\min}(T) = l\,k + (l+1)(n+1-k) =$$
$$= (n+1)\lfloor \log(n+1) \rfloor + 2(n+1) - 2^{\lfloor \log(n+1) \rfloor + 1}.$$

Полученную формулу можно представить в виде

$$E_{\min}(T) = (n+1)\log(n+1) + (n+1)(2 - \theta - 2^{1-\theta}), \qquad (2.5)$$

где $\theta = \log(n+1) - \lfloor \log(n+1) \rfloor$, $0 \le \theta < 1$.

Следует отметить, что $\theta = 0$, если все внешние вершины находятся на уровне $l = \log(n+1)$, т. е. для полных бинарных деревьев число внутренних вершин $n = 2^l - 1$, а число внешних вершин является степенью числа 2.

Очевидно, что высота полностью сбалансированного расширенного бинарного дерева $h(T) = \lceil \log(n+1) \rceil$.

Формулу для определения минимальной длины внутренних путей можно получить с помощью соотношения (2.1).

Понятие расширенного дерева легко обобщить на m-арные деревья, в которых все внутренние вершины имеют степень m. В расширенном m-арном дереве с n внутренними вершинами имеется $(m-1)n+1$ внешних вершин. Аналогом соотношения (2.1) будет соотношение

$$E(T) = (m-1)\,I(T) + m\,n.$$

Минимальная длина внешних путей равна

$$(n(m-1)+1)l - \frac{m^{l+1} - m}{m-1} + m\,n, \text{ где } l = \lfloor \log_m(n(m-1)+1) \rfloor.$$

Упражнения

1. Разработать алгоритмы включения элемента в заданную позицию и исключения элемента из заданной позиции для последовательно распределенного представления множества.

2. Разработать алгоритмы включения и исключения элемента (позиции заданы), а также выделения (процедура *new*) и освобождения (процедура *dispose*) памяти для связных списков, представленных с помощью массивов.

3. Разработать алгоритмы копирования списка, включения и исключения элементов для разновидностей связных списков:

а) циклический список (без заголовка и с заголовком);

б) двусвязный список (без заголовка и с заголовком);

в) циклический связный список (без заголовка и с заголовком).

4. Разработать алгоритм, который переставляет элементы односвязного списка в обратном порядке.

5. Разработать метод поддержания в одном линейном массиве двух стеков, при котором ни один из стеков не переполняется до тех пор, пока весь массив не будет заполнен. При этом стек никогда не перемещается внутри массива на другие позиции. Разработать алгоритмы включения и исключения элементов, манипулирующие обоими стеками, учитывая, что стеки растут навстречу друг другу.

6. Очередь можно реализовать с помощью циклического связного списка. В этом случае для обеспечения доступа к очереди достаточно задать только один указатель, который ссылается на последний элемент очереди. Тогда элемент, следующий за ним, будет первым элементом очереди. Разработать алгоритмы включения и исключения для такой реализации очереди.

7. Реализовать очередь на базе двух стеков. Оценить время выполнения операций с очередью.

8. Реализовать стек на базе двух очередей. Оценить время выполнения стековых операций.

9. Разработать алгоритм прохождения в горизонтальном порядке леса, представленного с помощью естественного соответствия бинарным деревом.

10. Разработать алгоритм копирования бинарного дерева.

11. Разработать алгоритм создания нового бинарного дерева, являющегося зеркальным отражением исходного (все левые поддеревья становятся правыми поддеревьями и наоборот).

12. Разработать алгоритм определения высоты бинарного дерева, т. е. числа ребер в самом длинном из путей от корня до листьев.

13. Разработать алгоритм определения числа вершин на заданном уровне непустого бинарного дерева.

14. Разработать алгоритм определения числа листьев в непустом бинарном дереве.

15. Разработать алгоритм преобразования бинарного дерева в симметрично прошитое бинарное дерево.

16. Разработать алгоритмы прямого и симметричного прохождений симметрично прошитого бинарного дерева, не использующие операции со стеками.

17. Разработать алгоритм преобразования бинарного дерева в прямо прошитое бинарное дерево.

18. Разработать алгоритм прямого прохождения прямо прошитого бинарного дерева, не использующего операции со стеками.

***19**. Разработать алгоритмы операций объединения, пересечения и разности множеств, представленных с помощью:

а) характеристических векторов;

б) упорядоченных связных списков;

в) бинарных деревьев.

Глава 3. ИСЧЕРПЫВАЮЩИЙ ПОИСК

Класс комбинаторных задач характеризуется тем, что в них рассматриваются задачи на существование, эффективное построение, перечисление и оптимизацию объектов, образованных из сравнительно большого числа элементов. При решении подобных задач приходится иметь дело с алгоритмами, ищущими решение методом проб и ошибок. Комбинаторные задачи обычно требуют исчерпывающего поиска множества всех возможных решений или наилучшего решения, а алгоритмы решения имеют экспоненциальную временную сложность. Общие схемы решения должны быть хорошо приспособлены к конкретной задаче так, чтобы за счет сокращения перебора полученный алгоритм стал пригодным для практического использования.

Одним из широко используемых общих методов исчерпывающего поиска является *поиск с возвратом* (backtrack), а также такие его разновидности, как *метод ветвей и границ* для поиска оптимальных решений, *метод альфа-бета-отсечений* для игровых задач. Для теоретико-числовых задач могут быть полезны *методы решета*.

3.1. Поиск с возвратом

3.1.1. Общий алгоритм поиска с возвратом

В общем случае предполагается, что решение задачи представляет собой вектор $(a_1, a_2, \ldots)$ конечной, но не определенной длины, удовлетворяющий некоторым ограничениям. Каждый элемент a_i является элементом конечного линейно упорядоченного множества A_i. Таким образом, при исчерпывающем поиске должны рассматриваться элементы множества $A_1 \times A_2 \times \ldots \times A_i$ для $i = 0, 1, 2, \ldots$ в качестве возможных решений. В качестве исходного частичного решения выбирается пустой вектор ( ) и на основе имеющихся ограничений определяется, какие элементы из множества A_1 являются кандидатами в a_1; подмножество таких кандидатов обозначим через S_1. В качестве a_1 выбирается наименьший элемент множества S_1; в результате получается частичное решение (a_1). В общем случае различные ограничения, описывающие решения, определяют, из какого подмножества S_k

множества A_k должны выбираться кандидаты для расширения частичного решения от $(a_1, a_2, \ldots, a_{k-1})$ до $(a_1, a_2, \ldots, a_{k-1}, a_k)$. Если частичное решение $(a_1, a_2, \ldots, a_{k-1})$ не предоставляет возможностей для выбора элемента a_k, т. е. $S_k = \varnothing$, то необходимо вернуться и выбрать новый элемент a_{k-1}. Если новый элемент a_{k-1} выбрать невозможно, придется вернуться еще дальше и выбрать новый элемент a_{k-2} и т. д.

Процесс поиска с возвратом удобно представить в виде дерева поиска, в котором исследуемое подмножество множества $A_1 \times A_2 \times \ldots \times A_i$ для $i = 0, 1, 2, \ldots$ представляется следующим образом. Корню дерева (нулевой уровень) ставится в соответствие пустой вектор. Его сыновья образуют множество S_1 кандидатов для выбора a_1. В общем случае вершины k-го уровня образуют множества S_k кандидатов на выбор a_k при условии, что $a_1, a_2, \ldots, a_{k-1}$ выбраны так, как указывают предки этих вершин. Пример дерева поиска представлен на рисунке 3.1, где пунктирные линии показывают порядок прохождения вершин в процессе поиска с возвратом. Вопрос о том, имеет ли задача решение $(a_1, a_2, \ldots)$, равносилен вопросу, являются ли какие-нибудь вершины дерева решениями. Для поиска всех решений необходимо получить все такие вершины.

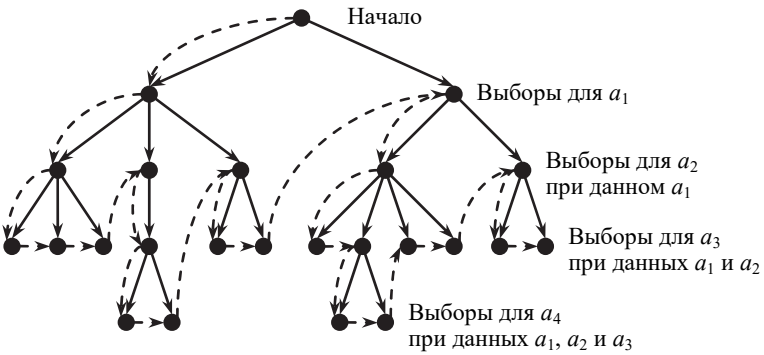


Рис. 3.1. Дерево поиска

Рассмотренный общий метод поиска с возвратом для нахождения всех решений формально описывается алгоритмом 3.1. Внутренний цикл осуществляет расширение частичного решения

(продвижение на следующий уровень в дереве поиска), если оно возможно (т. е. при $S_k \neq \varnothing$). Внешний цикл реализует возврат к предыдущему частичному решению (возврат на предыдущий уровень в дереве поиска) в случае невозможности дальнейшего продвижения (при $S_k = \varnothing$). Переменная *count* в алгоритме не имеет принципиального значения для поиска, она носит информативный характер и служит для подсчета числа исследованных вершин в дереве в процессе поиска.

определить $S_1 \subseteq A_1$
$count \leftarrow 0$
$k \leftarrow 1$
while $k > 0$ **do**
- **while** $S_k \neq \varnothing$ **do**
  - // продвижение
  - $a_k \leftarrow$ элемент из S_k
  - $S_k \leftarrow S_k - \{a_k\}$
  - $count \leftarrow count + 1$
  - **if** $(a_1, a_2, \ldots, a_k)$ – решение **then** записать его
  - $k \leftarrow k + 1$
  - определить $S_k \subseteq A_k$
- $k \leftarrow k - 1$ // возвращение

// все решения найдены

Алгоритм 3.1. Общий алгоритм поиска с возвратом

Если требуется найти только одно решение, алгоритм можно легко модифицировать так, чтобы он завершал работу после записи первого найденного решения; в этом случае останов в конце внешнего цикла **while** означает, что решений нет.

3.1.2. Применение общего алгоритма

Общий алгоритм поиска с возвратом представляет собой только общую схему решения, с которой следует подходить к конкретной задаче. Его непосредственное применение обычно приводит к алгоритмам с недопустимо большим временем работы. Поэтому общий алгоритм должен быть хорошо приспособлен к решению конкретной задачи, чтобы он был пригоден для практического использования. Часто это требует большой изобретательности при поиске способов сокращения перебора.

Для применения общего алгоритма поиска с возвратом к конкретной задаче необходимо выполнить анализ задачи с целью определения различных ограничений и специальных методов и приемов повышения эффективности процесса поиска за счет сокращения перебора. Прежде всего необходимо решить вопрос о представлении вектора решения, для каждого элемента a_k вектора определить все его возможные значения (т. е. множества A_k), детально проанализировать задачу для выявления свойственных ей ограничений, которые являются основой для вычисления подмножеств S_k кандидатов для расширения частичных решений. Таким образом, в первую очередь необходимо определиться со структурами данных для представления множеств A_k, S_k, векторов решений и их элементов a_k, а также с множеством производимых над ними операций, позволяющих добиться по возможности наибольшей эффективности вычислений. Процесс приспособления общего алгоритма завершается разработкой окончательного алгоритма решения задачи, учитывающего все ограничения и усовершенствования, направленные на сокращение перебора.

Проиллюстрируем это на примере решения одной из простейших комбинаторных задач – задачи о ферзях, которую можно сформулировать следующим образом: найти все варианты расстановки максимального числа ферзей на шахматной доске размера $n \times n$ так, чтобы ни один ферзь не атаковал другого. Поскольку ферзь атакует все поля в своей строке, своем столбце и диагоналях, очевидно, что на шахматной доске можно расставить максимум n не атакующих друг друга ферзей. Таким образом, поставленная задача сводится к определению возможности расстановки n не атакующих друг друга ферзей и, если расстановка возможна, к определению количества таких расстановок. Если же расстановка n ферзей невозможна, то решается задача расстановки $(n - 1)$ ферзей и т. д.

Сначала решим вопрос о представлении вектора решений. Очевидно, что все решения имеют одну и ту же фиксированную длину n, т. е. решение можно представить вектором $(a_1, \ldots, a_n)$. На первый взгляд, элемент a_i $(1 \le i \le n)$ этого вектора должен представлять собой координату позиции, в которой размещается i-й ферзь, т. е. упорядоченную пару чисел, определяющих соответственно номер строки и номер столбца. Однако поскольку в

каждом столбце может находиться только один ферзь, то решение можно представить более простым вектором $(a_1, \ldots, a_n)$, в котором элемент a_i $(1 \le i \le n)$ есть номер строки ферзя, расположенного в столбце с номером i, т. е. координатой позиции является пара (a_i, i). Очевидно, что множества значений элементов a_i совпадают, т. е. $A_1 = \ldots = A_n = \{1, \ldots, n\}$.

Рассмотрим свойственные задаче ограничения. Одно ограничение, связанное с тем, что в столбце может находиться только один ферзь, учтено представлением вектора решения. Другое ограничение заключается в том, что в каждой строке может быть только один ферзь, поэтому если $i \ne j$, то $a_i \ne a_j$. Наконец, поскольку ферзи могут атаковать друг друга по диагонали, мы должны иметь $|a_i - a_j| \ne |i - j|$, если $i \ne j$. Таким образом, для того чтобы определить, можно ли добавить a_k для расширения частичного решения $(a_1, a_2, \ldots, a_{k-1})$ до $(a_1, a_2, \ldots, a_{k-1}, a_k)$, достаточно сравнить элемент a_k с каждым a_i, $i < k$. Эту проверку можно реализовать в виде функции *QUEEN*, представленной алгоритмом 3.2, которая отвечает на вопрос, включить данную позицию в подмножество S_k кандидатов на выбор a_k или нет.

function $QUEEN(a_k, k)$ // тип **Boolean**
// **true**, если можно поставить ферзя в строку a_k столбца k
  $flag \leftarrow$ **true**
  $i \leftarrow 1$
  while $i < k$ **and** $flag$ **do**
    if $a_i = a_k$ **or** $|a_i - a_k| = |i - k|$
      then $flag \leftarrow$ **false**
    $i \leftarrow i + 1$
  $QUEEN \leftarrow flag$
return

Алгоритм 3.2. Функция *QUEEN*

Поскольку областью значений каждого элемента a_i вектора решения является множество целых чисел от 1 до n, нет необходимости в вычислении и явном хранении подмножеств S_k. Проще хранить наименьшее значение из S_k и следующее значение вычислять по мере необходимости. Текущее значение элемента множества S_k обозначим через s_k. Тогда проверке условия $S_k \ne \varnothing$

будет соответствовать условие $s_k \le n$. В результате процедуру нахождения всех решений задачи о не атакующих друг друга ферзях на доске размера $n \times n$ можно формально представить алгоритмом 3.3.

$s_1 \leftarrow 1$
$k \leftarrow 1$

while $k > 0$ **do** {
- **while** $s_k \le n$ **do** {
  - $a_k \leftarrow s_k$
  - $s_k \leftarrow s_k + 1$
  - **while** $s_k \le n$ **and not** $QUEEN(s_k, k)$
    - **do** $s_k \leftarrow s_k + 1$
  - **if** $k = n$
    - **then** {записать $(a_1, a_2, \ldots, a_k)$ как решение
  - $k \leftarrow k + 1$
  - $s_k \leftarrow 1$
  - **while** $s_k \le n$ **and not** $QUEEN(s_k, k)$
    - **do** $s_k \leftarrow s_k + 1$
- $k \leftarrow k - 1$

Алгоритм 3.3. Решение задачи о ферзях методом поиска с возвратом

Следует отметить, что данный алгоритм корректно обрабатывает и ситуацию, когда $k = n + 1$, поскольку вычисляемое значение s_{n+1} не меньше, чем $n + 1$, и, следовательно, множество S_{n+1} всегда пусто. Включение во внутренний цикл специальной проверки для предотвращения ситуации, когда значение k становится больше n, будет слишком дорогостоящим с точки зрения времени работы алгоритма.

3.1.3. Повышение эффективности поиска с возвратом

Практически все методы повышения эффективности поиска с возвратом связаны с сокращением перебора вариантов. Рассмотрим некоторые из них.

Поиск с ограничениями. Этот метод иногда называют отсечением ветвей, поскольку при этом удаляются поддеревья из дерева поиска. Основой метода является детальный анализ задачи для выявления ограничений, сокращающих процесс поиска. Практически этот метод нами использован в алгоритме решения задачи о ферзях, в котором учитываются ограничения, связанные

с тем, что ферзь атакует все поля в своей строке, своем столбце и диагоналях. Оценим, как влияют эти ограничения на процесс поиска. Если нет никаких ограничений, то на доске размером $n \times n$ существует $\binom{n^2}{n}$ (для $n = 8$ около $4{,}4 \cdot 10^9$) возможных способов расстановки n ферзей. Тот факт, что в каждом столбце может находиться только один ферзь, дает n^n расстановок (для $n = 8$ около $1{,}7 \cdot 10^7$). То, что в строку можно поставить только одного ферзя, говорит о том, что вектор $(a_1, \ldots, a_n)$ может быть решением только тогда, когда он является перестановкой элементов $(1, 2, ..., n)$, что дает $n!$ возможных расстановок (для $n = 8$ около $4{,}0 \cdot 10^4$). Требование, что на диагонали может находиться только один ферзь, еще больше сокращает число возможных расстановок. Для последнего ограничения аналитическое выражение, позволяющее оценить число возможных расстановок, получить трудно, поэтому необходима экспериментальная оценка размеров дерева поиска. Например, для $n = 8$ дерево поиска содержит только 2056 вершин. Таким образом, рассмотренные ограничения позволили исключить из рассмотрения большое число возможных расстановок n ферзей, а алгоритм 3.3 исследует дерево, состоящее из 2056 вершин.

Эквивалентность решений. Метод заключается в том, что определяются критерии эквивалентности решений и процедуры перевода одного решения в другое эквивалентное решение. Если будут найдены все попарно неэквивалентные решения, то можно построить все множество решений. В задаче о ферзях два решения можно считать эквивалентными, если одно из них переводится в другое с помощью ряда вращений и/или отражений. Например, поскольку ферзь, расположенный в любом углу доски, атакует все другие угловые поля, то решений, в которых ферзи стоят более чем в одном углу, нет. Следовательно, любое решение с ферзем в позиции (1, 1) может быть переведено вращением и/или отражением в эквивалентное, в котором позиция (1, 1) пуста. Таким образом, процесс поиска можно начать с $a_1 = 2$ и получить все неэквивалентные решения. Этим сразу обрывается у дерева большое поддерево (в дереве остается 1829 вершин).

Слияние (склеивание) ветвей. Идея состоит в следующем: если два или более поддеревьев данного дерева поиска изоморфны, то достаточно исследовать только одно из них. В задаче о

ферзях можно использовать склеивание, учитывая то, что если $a_1 > \lceil n/2 \rceil$, то найденное решение можно отразить и получить решение, для которого $a_1 \le \lceil n/2 \rceil$. Следовательно, поддеревья, соответствующие, например, случаям $a_1 = 2$ и $a_1 = n - 1$, изоморфны. В результате, не исследуя поддеревья, соответствующие случаям $a_1 = \lceil n/2 \rceil + 1$, $a_1 = \lceil n/2 \rceil + 2$, …, $a_1 = n$, можно найти все неэквивалентные решения. Кроме того, поскольку использование такого склеивания не влияет на ограничение в позиции (1, 1), достаточно найти решения только для $2 \le a_1 \le \lceil n/2 \rceil$. Если величина n нечетна и $a_1 = \lceil n/2 \rceil$, то можно ограничиться случаем $a_2 \le \lceil n/2 \rceil - 2$ в соответствии с тем же самым принципом. (На самом деле, если n нечетно, лучше ограничиться случаем $2 \le a_1 \le \lfloor n/2 \rfloor$ [21].) Это позволяет существенно сократить размеры дерева. Например, для $n = 8$ дерево сводится всего к 801 вершине.

Переупорядочение поиска. Данный метод может оказаться полезным, когда существование решения вызывает сомнения или когда вместо всех решений нужно найти только одно. Есть несколько способов его использования. Во-первых, если интуиция подсказывает, что решения будут иметь некоторый определенный вид, то поиск разумно построить так, чтобы раньше других исследовались возможные решения именно этого вида. Во-вторых, если это возможно, дерево следует перестроить так, чтобы вершины меньшей степени (т. е. имеющие относительно мало сыновей) были близки к корню дерева. Например, дерево поиска на рисунке 3.2*а* предпочтительнее дерева на рисунке 3.2*б*.

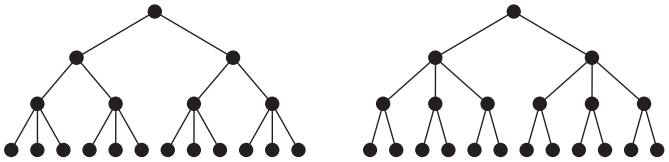


Рис. 3.2. Варианты дерева поиска:
а – предпочтительное дерево; *б* – произвольное дерево.

Перестройка дерева может оказаться полезной по следующим причинам. В общем случае, чтобы обнаружить, что путь не может вести к решению, должно быть накоплено несколько

ограничений, и обычно это случается на фиксированной глубине дерева. В результате большее число вершин будет запрещено, если большая часть ветвлений будет находиться ближе к листьям, чем к корню. Нужно помнить, что такой вид перестройки дерева может оказаться бесполезным, если должно быть исследовано все дерево. Кроме того, даже если не требуется исследовать все дерево, перестройка может переместить решения не ближе к началу поиска, а дальше от него.

Декомпозиция. Метод декомпозиции заключается в разложении задачи на k подзадач, решении этих подзадач и затем композиции решений подзадач в решение исходной задачи. В этом случае требования к объему памяти могут быть очень высокими, поскольку нужно хранить все решения подзадач, однако скорость может значительно возрасти. Например, если для решения задачи размера n требуется время $c2^n$, то использование метода декомпозиции позволит уменьшить время до $kc2^{n/k}+ T$, где T – время, требуемое для композиции решений подзадач.

3.1.4. Оценка сложности выполнения поиска с возвратом

Обычно поиск с возвратом приводит к алгоритмам, экспоненциальным по своим параметрам. Это является следствием того, что если все решения имеют длину не более n, то исследованию подлежат приблизительно $\prod_{i=1}^{n}|A_i|$ вершин дерева поиска, где $|A_i|$ – мощность множества A_i. Даже широкое использование ограничений и склеиваний позволяет в большинстве случаев добиться только того, что $|A_i|$ становится константой; при этом получаются деревья примерно с C^n вершинами для некоторой константы $C > 1$. Поскольку размеры дерева растут так быстро, можно попытаться определить возможность практического осуществления поиска (т. е. за приемлемое время) путем оценки числа вершин в дереве.

Аналитическое выражение для оценки удается получить редко, так как трудно предсказать, как взаимодействуют различные ограничения по мере появления их при продвижении вглубь дерева поиска. В подобных случаях, когда построение аналитической модели является трудной или вовсе не осуществимой за-

дачей, можно применить *метод Монте-Карло* (метод статистических испытаний). Смысл этого метода в том, что исследуемый процесс моделируется путем многократного повторения его случайных реализаций. Каждая случайная реализация называется *статистическим испытанием*.

Рассмотрим применение метода Монте-Карло для экспериментальной оценки размеров дерева поиска. Идея метода состоит в проведении нескольких испытаний, при этом каждое испытание представляет собой поиск с возвратом со случайно выбранными значениями a_i. Предположим, что имеется частичное решение $(a_1, a_2, \ldots, a_{k-1})$ и что число выборов для a_k, основанное на том, вводятся ли ограничения или осуществляется склеивание, равно $x_k = |S_k|$. Если $x_k \neq 0$, то a_k выбирается случайно из S_k и для каждого элемента вероятность быть выбранным равна $1/x_k$. Если $x_k = 0$, то испытание заканчивается. Таким образом, если $x_1 = |S_1|$, то $a_1 \in S_1$ выбирается случайно с вероятностью $1/x_1$; если $x_2 = |S_2|$, то при условии, что a_1 было выбрано из S_1, $a_2 \in S_2$ выбирается случайно с вероятностью $1/x_2$ и т. д. Математическое ожидание $x_1 + x_1x_2 + x_1x_2x_3 + x_1x_2x_3x_4 + \ldots$ равно числу вершин в дереве поиска, отличных от корня, т. е. оно равно числу случаев, которые будут исследованы алгоритмом поиска с возвратом. Существует доказательство этого утверждения [21].

Общий алгоритм поиска с возвратом легко преобразуется для реализации таких испытаний; для этого при $S_k = \varnothing$ вместо возвращения просто заканчивается испытание. Алгоритм 3.4 оценки размера дерева поиска осуществляет N испытаний для подсчета числа вершин в дереве. Операция $a_k \leftarrow rand(S_k)$ реализует случайный выбор элемента a_k из множества S_k.

Таким образом, каждое испытание представляет собой продвижение по дереву поиска от корня к листьям по случайно выбираемому на каждом уровне направлению. Поскольку в методе Монте-Карло отсутствует возврат, оценка размеров дерева выполняется за полиномиальное время.

Вычисление по методу Монте-Карло можно использовать для оценки эффективности алгоритма поиска с возвратом путем сравнения его с эталоном, полученным для задачи с меньшей размерностью. Например, при выполнении алгоритма 3.3 решения задачи о ферзях в случае доски размером 10×10 требуется поиск на дереве с 35 538 вершинами (эталон). В случае размера 11×11 после 1000 испытаний по методу Монте-Карло оценка

числа вершин была 161 668, и поэтому можно ожидать, что вычисления потребуют примерно в 4,5 раза больше времени, чем в случае с доской 10×10. Фактически оказалось, что в случае с доской 11×11 дерево имеет 166 925 вершин и время вычисления в 4 раза больше.

$count \leftarrow 0$ // суммарное число вершин в дереве
for $i \leftarrow 1$ **to** N **do**
- $sum \leftarrow 0$ // число вершин при одном испытании
- $product \leftarrow 1$ // накапливаются произведения
- определить $S_1 \subseteq A_1$
- $k \leftarrow 1$
- **while** $S_k \neq \varnothing$ **do**
  - $product \leftarrow product * |S_k|$
  - $sum \leftarrow sum + product$
  - $a_k \leftarrow rand(S_k)$
  - $k \leftarrow k + 1$
  - определить $S_k \subseteq A_k$
- $count \leftarrow count + sum$

$average \leftarrow count/N$ // среднее число вершин в дереве

Алгоритм 3.4. Метод Монте-Карло для поиска с возвратом

3.1.5. Способы программирования поиска с возвратом

Первый способ – это прямое программирование общего алгоритма 3.1 поиска с возвратом (приспособленного к конкретной задаче) в виде двух вложенных циклов: внутреннего цикла для расширения частичного решения и внешнего – для возвращения (итерационный подход).

Второй способ основан на рекурсивном подходе. Рекурсивный вариант поиска с возвратом представлен алгоритмом 3.5, где символ || означает конкатенацию векторов:

$(a_1, \ldots, a_n) \,||\, (b_1, \ldots, b_m) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$, $(\,) \,||\, (a) = (a)$,

где ( ) – пустой вектор. Очевидно, что при первом обращении к процедуре параметр *vector* представляет собой пустой вектор ( ), а параметр $i = 1$, т. е. первое обращение есть *BACKTRACK*(( ), 1).

procedure *BACKTRACK* (*vector*, *i*)
  if *vector* является решением **then** записать его
  определить S_i
  for $a \in S_i$ **do** *BACKTRACK* (*vector* || (*a*), *i* + 1)
return

Алгоритм 3.5. Рекурсивный вариант поиска с возвратом

В рекурсивном алгоритме все возвращения скрыты в механизме, реализующем рекурсию. Рекурсивный подход, как известно, требует дополнительных расходов памяти и времени по сравнению с итерационным подходом.

Третий способ использует язык ассемблера со средствами макрорасширения и основывается на предположении, что наиболее важным аспектом программы является ее быстродействие. Идея состоит в применении средств макрорасширения ассемблера для создания высокоспециализированных программ, в которых все или некоторые циклы не являются вложенными (т. е. циклы преобразуются в последовательные линейные конструкции). Такой подход устраняет определенные логические проверки и уменьшает число операций для контроля циклов. Если все решения имеют длину *n*, то это можно сделать, например, написав макрокоманду (назовем ее CODE_i) со следующим телом:

определить S_i
L_i : **if** $S_i = \varnothing$ **then goto** L_{i-1}
$a_i :=$ элемент из S_i
$S_i := S_i - \{a_i\}$

Макрокоманда CODE_i повторяется для $i = 1, 2, ..., n$, порождая программу вида

CODE_1
CODE_2
⋮
CODE_n
записать $(a_1, a_2, \ldots, a_n)$ как решение
goto L_n
L_0 : // все решения найдены

Применительно к задаче о ферзях макрокоманда CODE_i размещает на доске *i*-й ферзь.

Такой подход требует, чтобы все решения имели одну и ту же фиксированную длину. Это часто встречающийся случай, и

преимущество описанного подхода очевидно: макрокоманду CODE_i можно организовать так, что шаги будут приспособлены для S_i. Например, в задаче о ферзях нужно ограничение $2 \le a_1 \le \lceil n/2 \rceil$, и если n нечетно и $a_1 = \lceil n/2 \rceil$, то $1 \le a_2 \le \lceil n/2 \rceil - 2$. Включение этих проверок в общий алгоритм, написанный с циклами **while**, неэффективно, так как некоторые проверки нужно осуществлять каждый раз во внутреннем цикле, даже несмотря на то, что они редко что-либо дают. Используя макроподход, команды для проверок можно включить в программу только там, где это необходимо.

Существуют модификации макроподхода, не требующие, чтобы все решения имели одну и ту же длину. Кроме того, этот подход можно использовать и без макросредств, что позволяет использовать его при программировании на языках без макросредств (в том числе и на языках высокого уровня). Несмотря на некоторые неудобства, он заметно повышает быстродействие программы.

3.2. Метод ветвей и границ

Метод ветвей и границ является разновидностью поиска с возвратом и применяется для решения задач оптимизации. В таких задачах предполагается, что существует некоторый параметр (назовем его стоимостью), характеризующий качество решения. То есть каждое решение связано с определенной стоимостью, и необходимо найти оптимальное решение (решение с наименьшей стоимостью).

Пусть U – множество решений некоторой задачи. Обозначим через $cost(u)$ стоимость некоторого решения $u \in U$. Тогда оптимальным решением будет такое, которое дает минимальное значение $q_{\min} = \min\limits_{u \in U} cost(u)$ стоимости. Пусть на некотором этапе исследовано некоторое подмножество A множества U решений и определена стоимость q_A наилучшего на данный момент решения и пусть процесс поиска с возвратом достиг некоторой вершины дерева поиска. Тогда стоимость q_B соответствующего частичного решения является *нижней границей* стоимости множества B всех вытекающих из него решений (всех потомков этой вершины). При $q_B \ge q_A$ процесс продвижения по дереву прекра-

щается, поскольку уже ясно, что все решения из множества B не лучше имеющегося, т. е. исключаются из рассмотрения (отсекаются) соответствующие ветви в дереве поиска. Если $q_B < q_A$, то можно предположить, что среди потомков этой вершины может оказаться оптимальное решение. Поэтому продолжается дальнейшее исследование (продвижение по дереву поиска). Если стоимость нового полученного решения окажется меньше q_A, то это решение сохраняется в качестве наилучшего на данный момент, а его стоимость становится новым значением q_A. Таким образом, метод ветвей и границ состоит в последовательном улучшении оценки q_A стоимости, пока не будет получено оптимальное решение со стоимостью $q_{\min}$. Эффективность метода во многом зависит от стратегии исследования решений. Следует стремиться к тому, чтобы находить решения, для которых оценки стоимости близки к оптимальному решению, на ранних этапах поиска.

Для применения метода ветвей и границ стоимость должна быть четко определена для частичных решений. Кроме того, для всех частичных решений $(a_1, a_2, \ldots, a_{k-1})$ и для всех расширений $(a_1, a_2, \ldots, a_{k-1}, a_k)$ должно выполняться соотношение

$$cost\,(a_1, a_2, \ldots, a_{k-1}) \le cost\,(a_1, a_2, \ldots, a_{k-1}, a_k),$$

где $cost\,(a_1, a_2, \ldots, a_k)$ – стоимость решения $(a_1, a_2, \ldots, a_k)$. Когда стоимость обладает указанным свойством, можно отбросить частичное решение $(a_1, a_2, \ldots, a_k)$, если его стоимость больше или равна стоимости ранее вычисленных решений. Включение этого ограничения в общий алгоритм 3.1 поиска с возвратом дает алгоритм 3.6, где переменная *cost* – это стоимость текущего частичного решения, а *lowcost* – стоимость оптимального на данный момент решения (наименьшая на данный момент стоимость).

Для многих прикладных задач функция стоимости неотрицательная и часто удовлетворяет более сильному требованию

$$cost\,(a_1, a_2, \ldots, a_{k-1}, a_k) = cost\,(a_1, a_2, \ldots, a_{k-1}) + C(a_k),$$

где $C(a_k) \ge 0$ – функция стоимости, определенная для всех a_k. Это условие несколько упрощает общий алгоритм.

Рассмотрим применение метода ветвей и границ на примере типичной задачи оптимизации – *задачи коммивояжера*. В этой задаче коммивояжер должен посетить n городов, побывав в каж-

дом точно по одному разу, и возвратиться в исходный пункт. Известны стоимости перемещений между всеми парами городов (стоимость перемещения из города i в город j равна C_{ij}). Требуется определить оптимальный маршрут посещения городов, т. е. маршрут с минимальной общей стоимостью.

$lowcost \leftarrow \infty$
$cost \leftarrow 0$
определить $S_1 \subseteq A_1$
$k \leftarrow 1$
while $k > 0$ **do**
- **while** $S_k \neq \varnothing$ **and** $cost < lowcost$ **do**
  - $a_k \leftarrow$ элемент из S_k
  - $S_k \leftarrow S_k - \{a_k\}$
  - $cost \leftarrow cost(a_1, a_2, \ldots, a_k)$
  - **if** $(a_1, a_2, \ldots, a_k)$ – решение **and** $cost < lowcost$ **then**
    - хранить $(a_1, a_2, \ldots, a_k)$ как решение с наименьшей стоимостью
    - $lowcost \leftarrow cost$
  - $k \leftarrow k + 1$
  - определить $S_k \subseteq A_k$
- $k \leftarrow k - 1$
- $cost \leftarrow cost(a_1, a_2, \ldots, a_{k-1})$

// последнее сохраненное решение есть
// решение с наименьшей стоимостью

Алгоритм 3.6. Общий алгоритм метода ветвей и границ

Задача коммивояжера очевидным образом представляется в виде ориентированного взвешенного графа, вершины которого соответствуют городам, а дуги – дорогам между городами. Вес C_{ij} дуги (i, j) есть стоимость перемещения из города i в город j.

К задаче легко применить общий метод ветвей и границ, получая алгоритм, который возвращается всякий раз, когда стоимость текущего частичного решения равняется или превосходит стоимость лучшего на данный момент решения. Эта проверка устраняет просмотр некоторых частей дерева поиска, но она достаточно слабая и допускает глубокое проникновение внутрь дерева до уровня, когда ветви обрываются. Произвольный фиксированный порядок городов является причиной того, что много

времени теряется на исследование путей, которые начинаются с пары городов 1–2, если они отстоят далеко друг от друга.

Для решения подобных задач методом ветвей и границ относительно успешной является перестройка дерева поиска (переупорядочение поиска), позволяющая находить почти оптимальные решения на ранних этапах поиска. Рассмотрим *алгоритм Литтла* (Little), в котором техника перестройки дерева заключается в разбиении на каждом шаге всех оставшихся решений на два подмножества с их точными нижними границами стоимостей. Для разбиения используется некоторая дуга (i, j), выбираемая по определенным критериям так, чтобы отсечь максимальное количество ветвей дерева. Первое подмножество (левое поддерево) включает все решения, в которые эта дуга входит (обозначим его $Y(i, j)$). Второе подмножество (правое поддерево) включает все решения, в которые дуга (i, j) не входит (обозначим его $N(i, j)$).

Таким образом, дерево поиска становится бинарным, каждой вершине сопоставляется нижняя граница стоимостей всех решений, вырастающих из него. Кроме того, каждая вершина (кроме корня) определяет локальное ограничение (входит данная дуга в решение или нет) на всех ее потомков. Поэтому для обозначения вершин будем использовать те же обозначения, что и для соответствующих множеств, т. е. $Y(i, j)$ и $N(i, j)$. Чтобы получить все ограничения, относящиеся к данной вершине, достаточно пройти путь от корня дерева к этой вершине.

По количеству элементов множества $Y(i, j)$ и $N(i, j)$ не одинаковы. Общее число решений в задаче с n городами равно $(n - 1)!$. Если зафиксирована дуга (i, j), т. е. выбраны два города, то имеется $(n - 2)$ способов выбрать третий город, $(n - 3)$ способов – четвертый и т. д. Следовательно, имеется $(n - 2)!$ решений, включающих дугу (i, j), а значит

$$(n - 1)! - (n - 2)! = (n - 2)(n - 2)!$$

решений, не включающих дугу (i, j). Поэтому при $n > 2$ множество $N(i, j)$ будет содержать большее число элементов, чем множество $Y(i, j)$. Поскольку множество с меньшим числом элементов исследовать проще, то в качестве дуги (i, j) следует брать такую, при которой множество $Y(i, j)$ имеет меньшую нижнюю границу, чем множество $N(i, j)$.

Для получения нижних границ L стоимостей решений используется тот факт, что в каждое решение входит только по одному элементу из каждой строки и из каждого столбца матрицы стоимостей. Поэтому уменьшение на какое-либо число элементов любой строки или любого столбца приводит к тому, что на это же число уменьшается стоимость всех решений, но сам путь остается неизменным. На этом свойстве основана операция *приведения* матрицы стоимостей.

Операция приведения матрицы заключается в вычитании из элементов каждой строки i наименьшего элемента этой строки (обозначим h_i), затем из элементов каждого столбца j вычитается наименьший элемент этого столбца (обозначим g_j). В результате приведения стоимости всех решений уменьшатся на сумму вычтенных из каждой строки и каждого столбца чисел, а приведенная матрица будет содержать неотрицательные элементы, причем в каждой строке и каждом столбце будет по крайней мере один элемент с нулевым значением. Таким образом, нижняя граница стоимости любого решения $L = \sum_{i=1}^{n} h_i + \sum_{j=1}^{n} g_j$. Пример приведения матрицы стоимостей для $n = 5$ представлен на рисунке 3.3.

$i \setminus j$	1	2	3	4	5	h_i
1	∞	25	30	5	22	5
2	12	∞	8	16	19	8
3	17	24	∞	21	14	14
4	23	11	20	∞	9	9
5	27	13	7	26	∞	7

$\sum h_i = 43$

а

$i \setminus j$	1	2	3	4	5
1	∞	20	25	0	17
2	4	∞	0	8	11
3	3	10	∞	7	0
4	14	2	11	∞	0
5	20	6	0	19	∞
g_j	3	2	0	0	0

$\sum g_j = 5$

б

$i \setminus j$	1	2	3	4	5
1	∞	18	25	0	17
2	1	∞	0	8	11
3	0	8	∞	7	0
4	11	0	11	∞	0
5	17	4	0	19	∞

$L = \sum h_i + \sum g_j = 48$

в

Рис. 3.3. Приведение матрицы стоимостей:
а – исходная матрица стоимостей; *б* – матрица, приведенная по строкам;
в – приведенная матрица с нижней границей $L = 48$.

Чтобы получить матрицу стоимостей для множества $N(i, j)$, следует в исходной (приведенной) матрице запретить движение по дуге (i, j), положив $C_{ij} = \infty$. Для множества $Y(i, j)$ матрица стоимостей получается из исходной удалением i-й строки и j-го столбца, так как включение в маршрут дуги (i, j) исключает возмож-

ность включения других дуг, идущих из i и приходящих в j. В результате размер матрицы уменьшается на единицу. Кроме того, необходимо следить за тем, чтобы предотвратить посещение города, который уже пройден, пока по самой последней дуге не будет реализован возврат в исходный город. В простейшем случае для этого достаточно запретить дугу (j, i), положив $C_{ji} = \infty$. В общем случае если дуга (i_u, j_1) добавляется к частичному маршруту, состоящему из путей $i_1 - i_2 - ... - i_u$ и $j_1 - j_2 - ... - j_v$, то необходимо предотвратить использование дуги (j_v, i_1). Для построенных матриц вновь выполняется приведение, и на вычисленные значения увеличиваются нижние границы.

Осталось решить вопрос, какую дугу необходимо выбирать для разбиения. Если в качестве разбивающей дуги (i, j) взять дугу, у которой в приведенной матрице $C_{ij} \neq 0$, то для множества $Y(i, j)$ нижняя граница возрастет на C_{ij}, поскольку заранее известно, что эта дуга войдет во все решения. Она может еще возрасти, если матрица при удалении i-й строки и j-го столбца будет допускать дальнейшее приведение. В то же время при замене элемента C_{ij} на бесконечность матрица останется приведенной, т. е. нижняя граница для $N(i, j)$ не возрастет. Это нежелательно. Поэтому в качестве дуги (i, j) надо брать такую, у которой в приведенной матрице $C_{ij} = 0$. Поскольку таких дуг может быть несколько, надо выбрать ту, для которой увеличение нижней границы для множества $N(i, j)$ будет наибольшим, так как в этом случае получится наибольшая разница в нижних границах для множеств $Y(i, j)$ и $N(i, j)$. Другими словами, следует выбрать нуль, который при замене на бесконечность разрешает вычитать наибольшее число из его строки и столбца. Увеличение нижней границы множества $N(i, j)$ обозначим через $\Delta(i, j)$, а его значение получается при сложении наименьших чисел в i-й строке и j-м столбце (при $C_{ij} = \infty$).

Рассмотрим этот процесс для приведенной матрицы (рис. 3.3 *в*). Значения $\Delta(i, j)$ для нее следующие:

$$\Delta(1, 4) = 17 + 7 = 24;\ \Delta(2, 3) = 1;\ \Delta(3, 1) = 1;\ \Delta(3, 5) = 0;$$
$$\Delta(4, 2) = 4;\ \Delta(4, 5) = 0;\ \Delta(5, 3) = 4.$$

Наибольшее увеличение нижней границы правого поддерева дает выбор дуги (1, 4), т. е. для вершины $N(1, 4)$ нижняя граница $L = 48 + 24 = 72$, а соответствующая матрица стоимостей получа-

ется из исходной заменой значения $C_{1,4}$ на бесконечность. Для получения матрицы стоимостей для левого поддерева (вершины $Y(1, 4)$) необходимо удалить первую строку и четвертый столбец и наложить запрет на дугу (4, 1), положив $C_{4,1} = \infty$ (рис. 3.4*а*). Поскольку полученная матрица является приведенной (каждая строка и каждый столбец содержат по крайней мере один элемент с нулевым значением), нижняя граница для $Y(1, 4)$ не изменяется и остается равной 48.

$i \setminus j$	1	2	3	5
2	1	∞	0	11
3	0	8	∞	0
4	∞	0	11	0
5	17	4	0	∞

$L = 48$

а

$i \setminus j$	1	3	5
2	∞	0	11
3	0	∞	0
5	17	0	∞

$L = 48$

б

$i \setminus j$	1	5
2	∞	11
3	0	∞

$L = 48$

в

$i \setminus j$	1	5
2	∞	0
3	0	∞

$L = 48 + 11 = 59$

г

Рис. 3.4. Процесс получения маршрута 1 – 4 – 2 – 5 – 3 – 1:
а – после выбора дуги (1, 4); *б* – после выбора дуги (4, 2);
в – после выбора дуги (5, 3); *г* – приведение матрицы *в*.

Продолжим рассмотрение левого поддерева. Имеем следующие значения $\Delta(i, j)$:

$$\Delta(2, 3) = 1;\ \Delta(3, 1) = 1;\ \Delta(3, 5) = 0;$$
$$\Delta(4, 2) = 4;\ \Delta(4, 5) = 0;\ \Delta(5, 3) = 4.$$

Наибольшее увеличение нижней границы правого поддерева дают дуги (4, 2) и (5, 3). Выбираем любую из них, например (4, 2), получая цепочку 1 – 4 – 2. Тогда для $N(4, 2)$ имеем нижнюю границу $L = 48 + 4 = 52$. Матрица для $Y(4, 2)$ получается удалением четвертой строки и второго столбца. Для предотвращения посещения города 1 из города 2 (получается замкнутый контур, но еще не все города пройдены) необходимо положить $C_{2,1} = \infty$ (рис. 3.4*б*). Полученная матрица является приведенной, поэтому нижняя граница для $Y(4, 2)$ не меняется и остается равной 48.

Для полученной матрицы имеем:

$$\Delta(2, 3) = 11;\ \Delta(3, 1) = 17;\ \Delta(3, 5) = 11;\ \Delta(5, 3) = 17.$$

Наибольшее увеличение нижней границы правого поддерева дают дуги (3, 1) и (5, 3). Чтобы показать, что частичное решение может содержать произвольное число непересекающихся цепо-

чек, выберем для разбиения дугу (5, 3), получая цепочки 1 – 4 – 2 и 5 – 3. Тогда для $N(5, 3)$ имеем $L = 48 + 17 = 65$. Матрица стоимостей для $Y(5, 3)$ получается удалением пятой строки и третьего столбца и заменой $C_{3,5}$ на бесконечность (рис. 3.4*в*). Полученная матрица не является приведенной. Для ее приведения надо вычесть 11 из второй строки (рис. 3.4*г*), что увеличивает нижнюю границу стоимостей решений в левом поддереве, т. е. для $Y(5, 3)$ нижняя граница $L = 48 + 11 = 59$. Дальнейшее разбиение решений не имеет смысла, поскольку правые поддеревья будут пустыми (при наличии цепочек 1 – 4 – 2 и 5 – 3 не существует решений, в которые не входят дуги (2, 5) или (3, 1)), т. е. остается единственная возможность – добавить в решение обе оставшиеся дуги (2, 5) и (3, 1). В результате получается замкнутый путь (маршрут) 1 – 4 – 2 – 5 – 3 – 1, стоимость которого равна 59.

Чтобы выяснить, является ли полученное решение оптимальным, необходимо просмотреть значения нижних границ для всех правых поддеревьев. Если значение нижней границы какого-либо правого поддерева меньше стоимости полученного решения, то в этом поддереве может находиться лучшее решение, т. е. необходимо продолжить решение задачи, применяя метод разбиений решений для этого поддерева.

В данном примере имеются следующие значения нижних границ правых поддеревьев: $L = 65 > 59$ для $N(5, 3)$, $L = 52 < 59$ для $N(4, 2)$ и $L = 72 > 59$ для $N(1, 4)$. Поскольку оценка $L = 52$ для вершины $N(4, 2)$ меньше стоимости 59 лучшего на данный момент маршрута, более оптимальное решение возможно в поддереве с корнем $N(4, 2)$, т. е. в подмножестве решений, маршруты которых включают в себя дугу (1, 4) и не содержат дугу (4, 2). Необходимо повторить процесс разбиения решений. Матрица стоимостей для этого поддерева (рис. 3.5*а*) отличается от матрицы на рисунке 3.4*а* тем, что значение $C_{4,2} = \infty$ для запрета дуги (4, 2). Приведение этой матрицы вычитанием 4 из второго столбца (рис. 3.5*б*) и дает значение нижней границы рассматриваемого множества, равное 48 + 4 = 52. Последующее применение метода разбиений дает такой порядок выбора дуг: (4, 5), (3, 1), (2, 3) и (5, 2). Соответствующие приведенные матрицы для левых поддеревьев представлены на рис. 3.5*в* для дуги (4, 5) и рис. 3.5*г* для дуги (3, 1). В результате получается замкнутый путь

1 – 4 – 5 – 2 – 3 – 1, стоимость которого равна 52. Поскольку нет правых поддеревьев, значения нижних границ которых меньше 52, второе найденное решение является оптимальным.

$i \setminus j$	1	2	3	5
2	1	∞	0	11
3	0	8	∞	0
4	∞	∞	11	0
5	17	4	0	∞

$L = 48$

а

$i \setminus j$	1	2	3	5
2	1	∞	0	11
3	0	4	∞	0
4	∞	∞	11	0
5	17	0	0	∞

$L = 48 + 4 = 52$

б

$i \setminus j$	1	2	3
2	1	∞	0
3	0	4	∞
5	∞	0	0

$L = 52$

в

$i \setminus j$	2	3
2	∞	0
5	0	∞

$L = 52$

г

Рис. 3.5. Процесс получения маршрута 1 – 4 – 5 – 2 – 3 – 1: *а* – запрещена дуга (4, 2); *б* – приведение матрицы *а*; *в* – после выбора дуги (4, 5); *г* – после выбора дуги (3, 1).

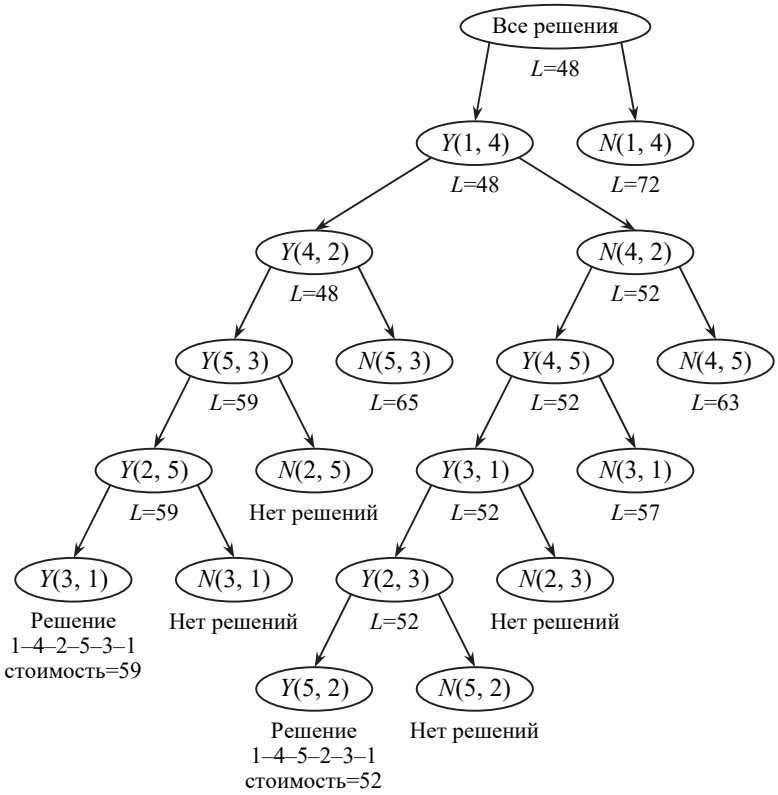


Рис. 3.6. Бинарное дерево поиска для задачи коммивояжера

Бинарное дерево поиска для рассмотренного примера решения задачи коммивояжера представлено на рисунке 3.6. В данном примере исследовано 19 вершин. Если использовать очевидную специализацию общего метода ветвей и границ, то потребуется исследовать гораздо больше вершин. На основе экспериментальных данных предполагается, что для случайной $n \times n$ матрицы стоимостей число исследуемых вершин при применении изложенного метода переупорядочения поиска равно $O(1{,}26^n)$ [21].

3.3. Альфа-бета-отсечение

Важной разновидностью метода ветвей и границ для решения игровых задач является *альфа-бета – отсечение* (α–β-отсечение). Этот метод используется при построении деревьев игр. *Дерево игры* – это дерево, которое появляется в результате исследования (способом поиска с возвратом) всех возможных последовательностей ходов. Корнем дерева игры является начальная конфигурация игры, сыновья корня – это возможные позиции после хода первого игрока, сыновья этих вершин – это возможные положения после ответного хода второго игрока и т. д. Каждый лист дерева игры представляет собой возможное окончание игры. Обычно дерево игры оценивается с позиций первого игрока. Тогда каждый лист помечается значением выигрыша первого игрока. Значения других вершин дерева определяются значениями их сыновей. Для произвольной внутренней вершины N с сыновьями $N_1, N_2, \ldots, N_k$ значение $V(N)$ определяется следующим образом:

$$V(N) = \begin{cases} \max(V(N_1), \ldots, V(N_k)), & \text{если уровень } N \text{ четный;} \\ \min(V(N_1), \ldots, V(N_k)), & \text{если уровень } N \text{ нечетный.} \end{cases}$$

Таким образом, первый игрок стремится максимизировать свой выигрыш, а второй – минимизировать свой проигрыш. Такой метод называется методом *минимакса*, так как по мере продвижения вверх по дереву попеременно используются функции максимума и минимума. Ясно, что если при условии минимакса значение корня дерева равно нулю, игра всегда завершается ничьей.

Метод α–β-отсечений использует идею метода ветвей и границ для обрывания поддеревьев дерева игры. Как только для вершины определена оценка, то становится известна некоторая информация о значении ее отца. Если отец вершины принадлежит уровню, на котором вычисляется максимум, то значение, присвоенное вершине, является нижней границей значения ее отца. Нижняя граница уровня, в котором вычисляется максимум, называется α-*значением*. Если отец находится на уровне, в котором вычисляется минимум, получается верхняя граница оценки, которая называется β-*значением*. Пусть известно α-значение некоторой вершины *N*. Если значение любого другого потомка вершины *N* будет не меньше α-значения, то можно игнорировать оставшуюся часть дерева, расположенную ниже этого потомка. Такая ситуация называется α-*отсечением*. Она возникает, если вершина, расположенная на два уровня ниже вершины с α-значением, имеет меньшее значение, чем α. Подобным образом определяется и β-*отсечение*, которое возникает, когда вершина, расположенная на два уровня ниже вершины с β-значением (вершина с β-значением находится на уровне, где вычисляется минимум), имеет большее значение, чем β.

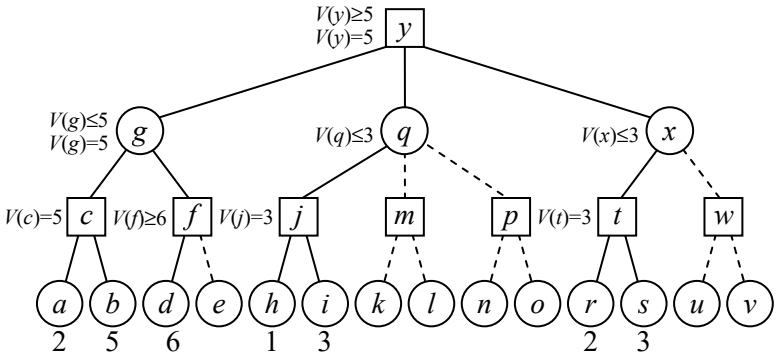


Рис. 3.7. Дерево игры

Рассмотрим процесс α–β-отсечений для дерева игры, представленного на рисунке 3.7. Вершины, расположенные на четных уровнях (где вычисляется максимум), изображены квадратами, а вершины, расположенные на нечетных уровнях (где вычисляется

минимум), – кружочками. Отсекаемые части дерева игры изображены пунктирными линиями.

После вычисления значения $V(c) = \max(V(a), V(b)) = 5$ становится известным β-значение для вершины g, равное 5, т. е. $V(g) \le 5$. Поскольку $V(d) = 6$, то $V(f) \ge 6$, что больше β-значения вершины g. Поэтому значение вершины e не влияет на значение вершины g, т. е. $V(e)$ можно игнорировать (β-отсечение). В результате получается значение $V(g) = 5$ вершины g и становится известным α-значение вершины y, равное 5, т. е. $V(y) \ge 5$. После вычисления значения $V(j) = 3$ становится известным β-значение для вершины q, равное 3, т. е. $V(q) \le 3$, что меньше α-значения вершины y. Поэтому поддерево с вершинами k, l, m, n, o, p можно игнорировать (α-отсечение). После вычисления $V(t) = 3$ и получения β-значения для вершины x, равного 3, также игнорируется поддерево с вершинами u, v, w (α-отсечение). В результате получается значение $V(y) = 5$ для вершины y и оптимальная стратегия игры, которая при условии минимакса дает наибольший выигрыш первому игроку.

Таким образом, метод α-β-отсечений позволяет проверить только часть дерева игры вместо исследования всего дерева методом поиска с возвратом.

3.4. Методы решета

Для многих теоретико-числовых вычислений широко используется другой метод исчерпывающего поиска – *метод решета*. В отличие от поиска с возвратом, в котором осуществляется поиск решений, метод решета рассматривает конечное множество элементов и исключает из него все элементы, не являющиеся решениями. В результате такого просеивания в множестве остаются только те элементы, которые являются решениями.

Работу метода решета можно показать на примере решета Эратосфена для отыскания простых чисел между N и N^2. Просеивание начинается с формирования исходного множества всех целых чисел от N до N^2. Затем поэтапно удаляются составные числа: сначала все числа, кратные двум, затем – все, кратные трем, и т. д. Процесс прекращается после просеивания для

наибольшего простого числа, меньшего или равного N. Процесс просеивания для $N = 5$ выглядит следующим образом:

Шаг 0 (исходный)

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Шаг 1 (исключаются кратные 2)

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Шаг 2 (исключаются кратные 3)

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Шаг 3 (исключаются кратные 5, кроме простого числа 5)

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Результат – простые числа: 5, 7, 11, 13, 17, 19, 23.

Основное достоинство методов решета очевидно. Если все элементы в множестве возможных решений можно пронумеровать натуральными числами, то хранить нужно только характеристический вектор. В этом векторе i-й разряд равен нулю, если i-й элемент не является решением, и равен единице в противном случае. Поэтому во множествах, состоящих из очень большого числа элементов, возможен поиск без явного порождения и исследования каждого элемента. Кроме того, в вычислительных устройствах логические операции можно выполнять параллельно над многими разрядами, что позволяет существенно увеличить эффективность вычислений.

Решето Эратосфена является специальным случаем *обобщенного модульного решета*. Пусть m_1, m_2, ..., m_t – множество из t целых чисел, называемых *модулями*. Для каждого m_i рассматривается n_i арифметических прогрессий

$$m_i\, k + a_{ij},\ k = 1, 2, ..., j = 1, 2, ..., n_i.$$

Задача состоит в отыскании всех целых чисел, заключенных в пределах некоторых целых A и B, которые для каждого m_i одновременно принадлежат одной из n_i прогрессий. В решете Эратосфена $m_1 = 2$, $m_2 = 3$, $m_3 = 5$, ..., $m_i = p$ (p – наибольшее простое число, меньшее или равное N), $n_i = m_i - 1$ и $a_{ij} = j$. Таким образом, решето Эратосфена можно интерпретировать как поиск всех чисел между N и N^2, которые одновременно являются членами одной из арифметических прогрессий в каждом из следующих множеств:

$\{2k+1\}$,
$\{3k+1, 3k+2\}$,
$\{5k+1, 5k+2, 5k+3, 5k+4\}$,
$\vdots$
$\{pk+1, ..., pk+p-1\}$.

То, что число принадлежит прогрессии $2k + 1$, означает, что оно нечетно; то, что оно принадлежит одной из прогрессий $3k + 1$ или $3k + 2$, означает, что оно не является кратным трем, и т. д.

Существует много решет, в которых модули m_1, m_2, ... заранее не известны, т. е. значение m_i будет зависеть от чисел, еще не удаленных после просеивания по модулю m_{i-1}. Поэтому многие решета строятся рекурсивным образом (*рекурсивные решета*). Обычно решето Эратосфена так и строится. После выписывания всех чисел от 2 до N вычеркиваются все числа, кратные двум, кроме самой двойки. Затем, поскольку наименьшее оставшееся число, кратные которому остались неудаленными, равно трем, удаляются все числа, кратные трем, кроме самой тройки, и т. д. Следует отметить, что на каждом шаге первое удаленное число является квадратом числа, с которого начинается просеивание. Процесс заканчивается, когда число, относительно которого производится просеивание, становится больше $\sqrt{N}$, так как никакие числа уже не могут быть удалены. Обычно размер ячеек решета Эратосфена удваивается, осуществляя предпросеивание четных чисел, т. е. просеивание начинается для исходного множества, состоящего только из нечетных чисел. Пусть X – двоичный набор; тогда рекурсивный вариант решета Эратосфена с предпросеиванием для числа 2 для нечетных простых чисел до $2n + 1$ представлен алгоритмом 3.7.

$X \leftarrow (1, 1, ..., 1)$
for $k \leftarrow 3$ **to** $\sqrt{2n+1}$ **by** 2 **do**
  // $X_{(k-1)/2}$ представляет нечетное число k
  if $X_{(k-1)/2} = 1$
    then for $i \leftarrow k^2$ **to** $2n+1$ **by** $2k$ **do** $X_{(i-1)/2} \leftarrow 0$
for $k \leftarrow 1$ **to** n **do if** $X_k = 1$ **then** вывести $2k+1$

Алгоритм 3.7. Рекурсивное решето Эратосфена

Примером более сложного решета является решето для вычисления следующей числовой последовательности U [21]. Вначале $U = (1, 2)$. Если вопрос о вхождении в U решен для всех целых чисел, меньших m, то $m \in U$ тогда и только тогда, когда m является суммой единственной пары различных элементов из U. Таким образом, $U = (1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26, \ldots)$. Эту последовательность можно вычислить с помощью двух параллельных просеиваний: целое должно быть просеяно, если является суммой, получаемой не менее чем одним способом, и не должно быть отсеяно, если является суммой, получаемой не более чем одним способом. Используются два двоичных вектора. Для $i > 2$

$X_i = 1$, если и только если i представимо в виде суммы не менее чем одним способом;

$Y_i = 1$, если и только если i представимо в виде суммы не более чем одним способом.

Двойное рекурсивное решето для вычисления элементов U до некоторого числа N представлено алгоритмом 3.8. Значение счетчика k увеличивается до тех пор, пока оно не достигнет первого целого числа, большего предыдущего элемента из U, представимого точно одним способом. Это целое принадлежит U, и поэтому разряды, соответствующие всем целым $k + i$ для $i < k$, должны быть обновлены.

$X \leftarrow (1, 1, 0, 0, \ldots, 0)$
$Y \leftarrow (1, 1, 1, 1, \ldots, 1)$
$k \leftarrow 1$

while $k < N$ **do** $\begin{cases} k \leftarrow k + 1 \\ \textbf{while}\ X_k \wedge Y_k = 0\ \textbf{and}\ k < N\ \textbf{do}\ k \leftarrow k + 1 \\ \textbf{for}\ i \leftarrow 1\ \textbf{to}\ \min(k-1, N-k)\ \textbf{do} \\ \quad \begin{cases} Y_{k+i} \leftarrow Y_{k+i} \wedge \overline{(X_{k+i} \wedge X_i \wedge Y_i)} \\ X_{k+i} \leftarrow X_{k+i} \vee (X_i \wedge Y_i) \end{cases} \end{cases}$

for $i \leftarrow 1$ **to** N **do if** $X_i \wedge Y_i = 1$ **then** вывести i

Алгоритм 3.8. Двойное рекурсивное решето

3.5. Эвристические алгоритмы

Методы исследований, основанные на неформальных, интуитивных соображениях, на общем опыте решения родственных за-

дач, называются *эвристическими*. Такие методы позволяют сократить количество исследуемых вариантов при поиске решения задачи и обычно не гарантируют наилучшего решения.

Если экспоненциальный алгоритм поиска оптимального решения, основанный на методе ветвей и границ, не применим для решения реальной задачи из-за недопустимо большого времени вычислений, делается попытка найти не оптимальное решение, а некоторое приближение к нему. Для этого определяется некоторая *эвристика*, ослабляющая требования к критерию оптимальности, что может дать полиномиальный алгоритм, результат которого будет разумно близким к оптимальному. Поэтому такие алгоритмы называются *эвристическими* (*приближенными*).

Типичным эвристическим методом в задачах оптимизации является поиск решений, которые оптимальны локально, а не глобально. Простейшую эвристику можно показать на примере решения задачи коммивояжера, в которой коммивояжер должен посетить n городов точно по одному разу и возвратиться в исходный пункт. Известны стоимости перемещений между всеми парами городов (стоимость перемещения из города i в город j равна C_{ij}). Эвристика заключается в следующем. В качестве начальной точки произвольно выбирается один из городов. Среди всех еще не посещавшихся городов в качестве следующего выбирается ближайший к последнему выбранному городу. Если все города уже посещались, возвращаются в начальный город. Таким образом, условие локальной оптимизации заключается в выборе на каждом шаге города, ближайшего к последнему пройденному. Этот алгоритм, назовем его алгоритмом *ближайшего соседа*, легко реализуется за время $O(n^2)$.

Для иллюстрации работы алгоритма воспользуемся матрицей стоимостей (см. рис. 3.3*а*). В качестве начальной точки выберем город 1. Ближайшим к нему является город 4, стоимость перемещения $C_{1,4} = 5$. Затем выбирается город 5 как ближайший к городу 4 ($C_{4,5} = 9$). Следующим будет город 3 ($C_{5,3} = 7$). Добавляем последний не посещавшийся город 2 ($C_{3,2} = 24$) и возвращаемся в исходный город 1 ($C_{2,1} = 12$). В результате получен маршрут 1 – 4 – 5 – 3 – 2 – 1 со стоимостью 57, который не является оптимальным (стоимость оптимального маршрута равна 52), но достаточно близок по стоимости к оптимальному. Очевидно, что выбор другой исходной точки может привести к совершенно

другому результату. Например, если в качестве исходного взять город 3, то алгоритм сформирует маршрут 3 – 5 – 2 – 1 – 4 – 3 со стоимостью 64.

Алгоритм ближайшего соседа относится к *жадным* алгоритмам. Такие алгоритмы на каждом шаге выбирают локально оптимальный вариант и являются основой многих эвристических алгоритмов. Во многих случаях они не позволяют получить оптимальное решение (небольшое отклонение от локальной оптимальности на некотором начальном шаге может дать существенную экономию позже). Однако для ряда задач жадные алгоритмы в действительности находят оптимальные решения.

Пусть N_n – порожденный алгоритмом ближайшего соседа маршрут со стоимостью $cost(N_n)$ и O_n – оптимальный маршрут со стоимостью $cost(O_n)$. Если матрица стоимостей симметрична ($C_{ij} = C_{ji}$ для любых i и j) и удовлетворяет неравенству треугольника ($C_{ij} \le C_{ik} + C_{kj}$ для любых i, j и k), то

$$\frac{cost(N_n)}{cost(O_n)} \le \frac{1}{2}\left(\lceil \log n \rceil + 1\right).$$

Ясно, что это не лучшая эвристика для задачи коммивояжера; существуют более сложные и тонкие эвристики, которые позволяют получать результат, более близкий к оптимальному (в среднем и худшем случаях).

Упражнения

1. Используя поиск с возвратом, решить следующие задачи:

а) найти в лабиринте все пути между двумя выделенными точками. Лабиринт может быть задан матрицей соединений, в которой для каждой пары точек указано, соединены они между собой или нет;

б) найти все расстановки пяти ферзей на шахматной доске, при которых каждое поле будет находиться под ударом одного из них;

в) получить все расстановки восьми ладей на шахматной доске, при которых ни одна ладья не угрожает другой;

г) найти все расстановки двенадцати коней на шахматной доске, при которых каждое поле будет находиться под ударом одного из них;

д) найти все расстановки восьми слонов на шахматной доске, при которых каждое поле будет находиться под ударом одного из них;

е) определить все возможные маршруты коня, начинающиеся на одном заданном поле шахматной доски и оканчивающиеся на другом. Никакое поле не должно встречаться в одном маршруте дважды.

2. Решить задачу коммивояжера с использованием общей схемы метода ветвей и границ. Определить число исследованных вершин в дереве поиска.

***3**. Решить задачу коммивояжера с использованием техники перестройки дерева поиска (алгоритм Литтла). Сравнить число исследованных вершин в дереве с соответствующим результатом, полученным в упражнении 2.

4. Решить задачу коммивояжера методом ближайшего соседа. Полученное решение сравнить с оптимальными решениями из упражнений 2 и 3.

5. Решить следующие задачи методом ветвей и границ:

а) найти в лабиринте кратчайший путь между двумя выделенными точками;

б) в данной последовательности чисел $a_1, a_2, \ldots, a_n$ выбрать возрастающую подпоследовательность наибольшей длины;

в) имеется n предметов, веса которых равны $a_1, a_2, \ldots, a_n$. Разделить эти предметы на две группы так, чтобы общие веса двух групп были максимально близки;

г) имеется n человек, которых нужно назначить на n работ. Стоимость назначения i-го человека на i-ю работу равна C_{ij}. Найти назначение, при котором каждая работа выполняется некоторым человеком и которое минимизирует общую стоимость назначения;

д) имеется m различных предметов, известны вес каждого предмета и его стоимость. Определить, какие предметы надо положить в рюкзак, чтобы общий вес не превышал заданной границы, а общая стоимость была максимальной;

е) имеется конечное множество заказов, каждый из которых требует ровно одну единицу времени для своего выполнения. Для каждого заказа известны срок выполнения и штраф за невыполнение к сроку. Требуется найти порядок выполнения заказов, при котором сумма штрафов будет наименьшей.

***6**. Используя метод α–β-отсечений, решить следующие игровые задачи:

а) имеется кучка из 100 спичек. Двое играющих поочередно берут по несколько спичек: не менее одной и не более десяти. Проигрывает тот, кто взял последнюю спичку;

б) имеется кучка из 100 спичек. Двое играющих поочередно берут по несколько спичек: не менее одной и не более чем вдвое больше, чем взял предыдущий игрок. На первом ходе можно взять одну или две спички. Выигрывает тот, кто берет последнюю спичку;

в) имеется три кучки спичек. Двое играющих по очереди делают ходы. Каждый ход заключается в том, что из какой-то одной кучки бе-

рется произвольное ненулевое число спичек. Выигрывает тот, кто взял последнюю спичку;

г) имеется 24 раскрытые карты: все карты с номерами от 1 до 6 обычной колоды, где туз считается за 1. Масти карт несущественны. Каждый игрок при своем ходе берет карту и складывает ее значение с суммой тех, которые были взяты ранее. Подсчитывается общая сумма карт, взятых игроками, а не отдельные суммы для каждого игрока. Выигрывает тот, кто берет в точности 50 очков;

д) имеется две кучки камней. Двое играющих по очереди делают ходы. Каждый ход может состоять в одном из двух: либо берется произвольное ненулевое число камней из какой-то одной кучки, либо берется одновременно по одинаковому ненулевому числу камней из обеих кучек. Выигрывает взявший последний камень.

***7**. Используя методы решета, решить следующие задачи:

а) найти счастливые числа, определяемые следующим образом. Из списка натуральных чисел 1, 2, 3, 4, 5, 6, … сначала исключается каждое второе число, в результате чего получается список 1, 3, 5, 7, 9, 11, 13, 15, 17, … . Поскольку число три является первым (не считая единицы) числом, которое не использовалось в качестве просеивающего, из оставшихся чисел исключается каждое третье число. В результате получается список 1, 3, 7, 9, 13, 15, 19, 21, 25, 27, 31, … Затем исключается каждое седьмое число и т. д.;

б) дано натуральное число N. Найти четверки меньших N простых чисел, принадлежащих одному десятку (например, 11, 13, 17, 19);

в) дано натуральное число N. Найти все меньшие N числа Мерсена. Простое число называется числом Мерсена, если оно может быть представлено в виде $2^p - 1$, где p – тоже простое число;

г) дано натуральное число N. Получить в порядке возрастания N первых натуральных чисел, которые не делятся ни на какие простые числа, кроме 2, 3 и 5;

д) найти все простые несократимые дроби, заключенные между 0 и 1, знаменатели которых не превышают 9 (дробь задается двумя натуральными числами – числителем и знаменателем);

е) найти натуральные числа, меньшие N, которые делятся с остатком, равным единице, на 3, 4, 5 и 6 и без остатка на 7.

Глава 4. МЕТОДЫ ПОИСКА

Поиск является одной из основных операций над множествами и обычно формулируется следующим образом. Имеется конечное множество $T = \{x_1, x_2, ..., x_n\}$ из n элементов, которое будем называть *таблицей*. В общем случае элементы таблицы могут иметь сложную структуру, но с каждым элементом ассоциирован некоторый *ключ* (*имя*), используемый для того, чтобы отличить один элемент от другого. Поскольку рассматривается в первую очередь процесс поиска, а обнаруженные данные с точки зрения поиска не представляют интереса, будем считать, что элементами таблицы являются имена (ключи). Поиск представляет собой запрос, который для заданной таблицы T и некоторого имени z, называемого *ключом*, или *аргументом поиска*, возвращает указатель на имя из таблицы, совпадающее с z (*успешный* поиск). Если такого имени нет, возвращает информацию (например, пустое значение указателя) о *безуспешном* поиске.

Таблица T является конечным подмножеством некоторого множества S, называемого *пространством имен*, которое может быть конечным или бесконечным. На множестве S должно быть определено отношение линейного (*естественного*) порядка, т. е. любые два элемента $x, y \in S$ сравнимы (должно выполняться в точности одно из трех условий: $x < y$, $x = y$, $x > y$). Отношение порядка должно обладать также свойством транзитивности, т. е. если $x < y$ и $y < z$, то $x < z$ для любых элементов $x, y, z \in S$. При анализе алгоритмов будем предполагать, что исход сравнений элементов $x, y \in S$ получается за время, не зависящее от мощности пространства имен и от мощности n исследуемой таблицы T.

Порядок, определенный на таблице T, будем называть *табличным*. Табличный порядок часто совпадает с естественным порядком, определенным на пространстве имен, однако такое совпадение не обязательно.

Таблицы, которые после их построения никогда не изменяются, называются *статическими*. Такие таблицы необходимо строить так, чтобы максимально минимизировать время поиска.

На большинстве таблиц, однако, производятся и другие операции. Наиболее важными из них являются операции модификации (добавление или исключение имени, сортировка), объединение нескольких таблиц в одну и т. д. В таких таблицах, называе-

мых *динамическими*, когда встает вопрос об эффективности выполнения тех или иных операций, нельзя ограничиваться оптимизацией структуры таблицы для достижения наибыстрейшего поиска. Это связано с тем, что общая эффективность работы с таблицами снижается за счет неэффективного выполнения других операций. Необходимо искать компромиссное решение, когда время поиска удлиняется из соображений более гибкой организации таблиц, которая позволяет выполнять некоторые виды операций более эффективно.

Ограничимся рассмотрением следующих четырех операций над таблицами.

Поиск z: если $z \in T$, то отметить его указателем, в противном случае указать, что $z \notin T$.

Включение z: поиск места включения; если $z \notin T$, то поместить его на соответствующее место.

Исключение z: поиск имени z в таблице; если $z \in T$, исключить его.

Распечатка: распечатка всех имен из таблицы T в их естественном порядке. Если табличный порядок не соответствует естественному, предполагается предварительная сортировка.

Детали реализации этих операций во многом зависят от структуры данных, используемой для представления таблицы.

4.1. Последовательный поиск

Последовательный (*линейный*) поиск подразумевает исследование имен в том порядке, в котором они встречаются в таблице, т. е. в худшем случае осуществляется просмотр всей таблицы. Для больших таблиц последовательный поиск нельзя отнести к методам быстрого поиска, поскольку даже в среднем он имеет тенденцию к использованию числа операций, пропорционального размеру n таблицы. Несмотря на это, последовательный поиск является единственным методом поиска, применимым к отдельным устройствам памяти и к тем таблицам, которые строятся на пространстве имен без линейного порядка. Кроме того, последовательный поиск является быстрым для достаточно малых таблиц.

Наиболее очевидный процесс последовательного поиска в таблице $T = \{x_1, x_2, \ldots, x_n\}$ первого вхождения заданного имени z

(имя z – ключ поиска) представлен алгоритмом 4.1. Алгоритм возвращает позицию i имени $x_i = z$ в случае успешного поиска и нулевое значение указателя – в случае безуспешного поиска.

```
for i ← 1 to n do
   if z = x_i then return(i) // найдено: i указывает на z
return(0) // не найдено: z не входит в T
```

Алгоритм 4.1. Последовательный поиск

Логика алгоритма показана для таблицы, представленной с помощью массива. Очевидно, что она остается той же самой и для таблицы, представленной с помощью связного списка. В этом случае i будет указывать на соответствующий узел списка при успешном поиске или иметь пустое значение Λ при безуспешном. Перемещение по списку реализуется с помощью поля *next* узла, т. е. вместо присваивания $i \leftarrow i + 1$ (скрытого в операторе **for**) следует использовать оператор $i \leftarrow i.next$.

Чтобы детализировать выполняемые внутри цикла операции, представим алгоритм в форме, близкой к машинному языку:

```
        i ← 1
цикл:   if i > n then // не найдено
        if z = x_i then // найдено
        i ← i + 1
        goto цикл
```

За каждую итерацию выполняется до четырех команд: два сравнения, одна операция увеличения i и одна операция передачи управления. Любое ускорение работы алгоритма должно быть следствием сокращения числа операций в цикле, поскольку время выполнения цикла определяет время работы всего алгоритма.

Для ускорения цикла общим приемом является добавление в таблицу специальных имен (называемых *сторожами*, или *часовыми*), которые делают необязательной явную проверку того, достиг ли указатель границы таблицы. Этот прием можно применить и к рассматриваемому алгоритму. Если перед поиском добавить искомое имя z в конец таблицы в качестве сторожа, то цикл всегда будет завершаться отысканием имени z, т. е. в цикле нет необходимости каждый раз выполнять проверку $i > n$. После выхода из цикла проверка условия $i > n$ выполняется только один раз для определения, является ли найденное имя z истинным или

специальным добавленным именем-сторожем. В результате получается улучшенный алгоритм последовательного поиска (алгоритм 4.2), в котором при каждой итерации выполняется только три операции вместо четырех.

```
x_{n+1} ← z  // добавление сторожа
i ← 1
while z ≠ x_i  do  i ← i + 1
if  i > n
   then   return (0)  // не найдено
   else   return (i)  // найдено: i указывает на z
```

Алгоритм 4.2. Улучшенный алгоритм последовательного поиска

Одной из особенностей алгоритма является то, что добавление перед поиском ключа z в конец таблицы возможно только, если таблица хранится в памяти с произвольным доступом, где имеется прямой непосредственный доступ к концу таблицы. Алгоритм неприменим, когда используется связное размещение имен таблицы (связный список с одним указателем на первый элемент) или память с последовательным доступом. В случае представления таблицы связным списком можно выйти из положения, если дополнить список указателем на последний элемент. Тогда можно реализовать добавление в конец списка узла с именем-сторожем (перед началом поиска), а после завершения поиска следует исключить этот узел.

Недостатком двух рассмотренных алгоритмов является то, что при безуспешном поиске всегда просматривается вся таблица. Если такой поиск возникает часто, имена необходимо хранить в естественном порядке. Это позволяет завершить поиск, когда при просмотре попадается первое имя, большее или равное ключу поиска. В этом случае в конец таблицы необходимо добавить имя-сторож ∞ (предполагается, что имя ∞ больше любого имени из пространства имен S), чтобы гарантировать выполнение условия завершения. В результате получается алгоритм 4.3.

Для анализа времени последовательного поиска за единицу времени примем время, необходимое для исследования одного элемента таблицы. Для нахождения в таблице i-го имени x_i требуется i единиц времени. Предполагая, что частота обращения к

именам таблицы распределена равномерно, можно вычислить среднее время успешного поиска S_n как

$$S_n = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}.$$

$x_{n+1} \leftarrow \infty$
$i \leftarrow 1$
while $z > x_i$ **do** $i \leftarrow i+1$
if $z = x_i$
  then **retutn**(i) // найдено: i указывает на z
  else **return**(0) // не найдено

Алгоритм 4.3. Последовательный поиск в упорядоченной таблице

Для безуспешного поиска среднее время равно примерно $n/2$, если имена в таблице хранятся в естественном порядке, или равно n в противном случае. Таким образом, временная сложность алгоритмов последовательного поиска составляет $O(n)$.

Операции включения и исключения в большей степени зависят от структуры данных, выбранной для представления таблицы, чем от порядка имен в таблице. Наиболее важными структурами данных для последовательного поиска являются связные списки и последовательно распределенные таблицы. Операции включения и исключения в связном списке требуют только изменения нескольких указателей, т. е. если известно местоположение, эти операции могут быть выполнены за время, не зависящее от размера таблицы. В последовательно распределенной таблице имена хранятся в последовательных ячейках. Поэтому исключение имени образует пропуск, который должен быть заполнен, а включение имени требует предварительного образования пропуска. В обоих случаях необходимо передвижение имен, что требует времени $O(n)$.

Распечатка имен заключается в последовательном просмотре имен таблицы с временем $O(n)$, если табличный порядок совпадает с естественным, в противном случае требуется предварительная сортировка с соответствующим временем.

Можно уменьшить среднее время успешного поиска, если имена в таблице хранить в порядке невозрастания частот обра-

щения. Распределение частот обращения обычно неизвестно. Если таблица является статической, то можно собрать статистику о частоте обращений, а затем реорганизовать таблицу в соответствии с полученным распределением частот. Если нет возможности собрать подобную статистику, таблицы можно сделать *самоорганизующимися*. В таких таблицах имена, к которым обращаются часто, передвигаются в направлении начала таблицы, т. е. по мере получения данных о частотах обращений происходит изменение табличного порядка. Следует отметить, что такое улучшение среднего времени успешного поиска приводит к увеличению времени безуспешного поиска, поскольку табличный порядок не совпадает с естественным.

4.2. Логарифмический поиск в статических таблицах

4.2.1. Бинарный поиск

Бинарный (*двоичный*) поиск предъявляет ряд требований к организации таблицы $T = \{x_1, x_2, \ldots, x_n\}$. Во-первых, имена в таблице должны храниться в их естественном порядке, т. е. таблица должна быть отсортирована. Во-вторых, должен обеспечиваться прямой доступ к именам таблицы, т. е. для статических таблиц наиболее удобно последовательное представление, когда имена распределены в памяти с произвольным доступом в последовательных ячейках. Другими словами, бинарный поиск требует отсортированной таблицы, организованной в виде массива.

Идея бинарного поиска заключается в том, что имя z ищется в интервале $[l, h]$, крайними точками которого являются указатели l (для нижней границы интервала) и h (для верхней границы интервала). Новый указатель m устанавливается в средней или близкой к ней точке интервала. Если $z = x_m$, искомое имя найдено. Если $z < x_m$, интервал поиска сводится к интервалу $[l, m - 1]$, если $z > x_m$, то к интервалу $[m + 1, h]$ и процесс повторяется. Если интервал становится пустым, поиск завершается безуспешно. Для получения логарифмического времени необходимо устанавливать указатель m за время, не зависящее от длины интервала, что обеспечивается возможностью прямого доступа к именам таблицы. Реализация рассмотренной идеи представлена алгоритмом 4.4.

$l \leftarrow 1$
$h \leftarrow n$

while $l \le h$ **do** {
- // в этот момент имеет место
- // $1 \le l \le h \le n$, $z \notin \{x_1, ..., x_{l-1}\}$, $z \notin \{x_{h+1}, ..., x_n\}$
- $m \leftarrow \lfloor (l+h)/2 \rfloor$
- **case** {
  - $z = x_m$: **return** (m) // найдено
  - $z < x_m$: $h \leftarrow m-1$
  - $z > x_m$: $l \leftarrow m+1$

return (0) // не найдено

Алгоритм 4.4. Бинарный поиск

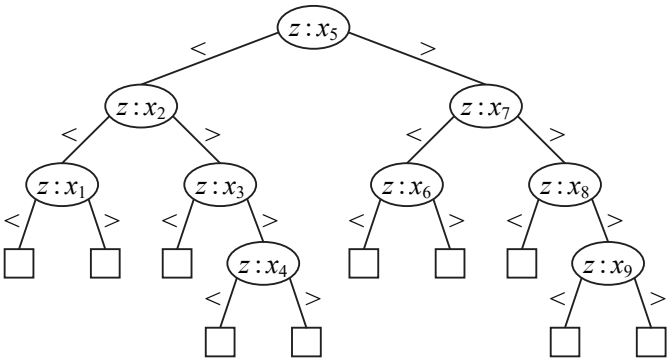


Рис. 4.1. Дерево, соответствующее бинарному поиску в таблице из девяти имён

Для анализа алгоритма последовательность сравнений ключа z с именем x_i (обозначим $z : x_i$) удобно представить расширенным бинарным деревом. На рисунке 4.1 представлено расширенное бинарное дерево, соответствующее бинарному поиску в таблице из девяти имён. Первое сравнение – это корень дерева, его левое и правое поддеревья являются бинарными деревьями, структура которых определяется последующими сравнениями. Алгоритм останавливается, когда он находит $z = x_i$ или когда достигает листа (внешней вершины) в случае безуспешного поиска. Высота дерева $T(1, n)$, соответствующего бинарному поиску в таблице с n именами, равна $\lceil \log(n+1) \rceil$. Поскольку высота дерева равна числу сравнений $z : x_i$, которые требуется сделать в худшем случае, можно сделать заключение, что для таблицы с n именами бинарный поиск требует не более $\lceil \log(n+1) \rceil$ сравнений, т. е. временная сложность алгоритма бинарного поиска есть $O(\log n)$.

4.2.2. Однородный бинарный поиск

Однородный (*равномерный*) бинарный поиск является модификацией обычного бинарного поиска. Эта модификация заключается в следующем. Вместо трех указателей – l (нижняя граница интервала), h (верхняя граница интервала) и m (середина интервала) – используется только два: текущая позиция m и величина его изменения δ. После каждого сравнения, не давшего равенства, можно установить $m \leftarrow m \pm \delta$ и $\delta \leftarrow \delta / 2$.

Практическая реализация однородного бинарного поиска требует исключительного внимания к мельчайшим деталям, например округлять результат деления к ближайшему большему или ближайшему меньшему целому, в каких случаях требуются имена-сторожа и т. п.

Один из вариантов реализации заключается в следующем. Сначала устанавливаются $m = \lceil n/2 \rceil$ и $\delta = \lfloor n/2 \rfloor$. Затем сравниваются ключ поиска z и имя x_m. Если $z = x_m$, поиск успешно завершен. В противном случае происходит либо увеличение (если $z > x_m$), либо уменьшение (если $z < x_m$) m на $\lceil \delta/2 \rceil$. В обоих случаях устанавливается новое значение $\delta = \lfloor \delta/2 \rfloor$. Снова производится сравнение z с x_m и т. д. Процесс продолжается до тех пор, пока на каком-то шаге не будет $z = x_m$ (успешный поиск) либо значение δ не станет равным нулю (безуспешный поиск). При такой реализации в случае четного n необходимо установить имя-сторож $x_0 = -\infty$. Детали реализации алгоритма предлагаются в качестве упражнения.

Если процесс поиска представить в виде расширенного бинарного дерева, то легко обнаружить, что разность между индексами имен вершин на уровне l и ее вершины-предшественника на уровне $l - 1$ представляет собой константу δ для всех вершин на уровне l. Поэтому такой поиск и называется однородным. По дереву можно также увидеть, что при безуспешном поиске перед окончанием работы алгоритма могут производиться лишние сравнения имен.

Теорию, лежащую в основе однородного бинарного поиска, можно пояснить следующим образом. Пусть на начальном этапе имеется интервал для поиска длиной $n - 1$. Сравнение со средним элементом (для четного n) или одним из двух средних эле-

ментов (для нечетного n) дает два интервала длиной $\lfloor n/2 \rfloor - 1$ и $\lceil n/2 \rceil - 1$. После повторения этого процесса k раз получается 2^k интервалов, наименьший из которых имеет длину $\lfloor n/2^k \rfloor - 1$, а наибольший – $\lceil n/2^k \rceil - 1$. Таким образом, длины двух интервалов на одном уровне отличаются не более чем на единицу. Это делает возможным выбор среднего элемента без запоминания последовательности точных значений длин интервалов.

Главным достоинством однородного бинарного поиска является то, что имеется возможность исключить из процесса поиска все операции деления, связанные с величиной δ. Для этого используется дополнительная вспомогательная таблица (конечно, если нет ограничений по объему памяти) с вычисленными заранее значениями $\delta_j = \left\lfloor \frac{n + 2^{j-1}}{2^j} \right\rfloor$ для каждого j-го шага поиска, где $1 \le j \le \lfloor \log n \rfloor + 2$. Тогда первое сравниваемое имя таблицы имеет индекс $m = \delta_1$; после каждого сравнения, не давшего равенства, устанавливаются $m = m \pm \delta_j$ и $j = j + 1$. Процесс завершается безуспешно, если достигается $\delta_j = 0$. Ясно, что для четного n необходимо имя-сторож $x_0 = -\infty$. Такая модификация позволяет осуществлять поиск быстрее обычного бинарного поиска. Однородный поиск требует времени $O(\log n)$.

4.2.3. Поиск Фибоначчи

Последовательность чисел Фибоначчи определяется рекуррентным соотношением

$$F_0 = 0,\ F_1 = 1,\ F_k = F_{k-1} + F_{k-2} \text{ при } k \ge 2.$$

Другими словами, в последовательности Фибоначчи

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

каждое число равно сумме двух предыдущих. Причем числа F_k экспоненциально растут с ростом k.

Числа Фибоначчи позволяют разработать альтернативу бинарному поиску. Технология поиска Фибоначчи заключается в следующем. Интервал поиска длины F_k разбивается на два интервала с длинами F_{k-1} и F_{k-2} соответственно, приводя к делению на две части в соотношении $\alpha = \left(\sqrt{5} - 1\right)/2 \approx 0{,}62$ (в отличие от бинарного поиска, где $\alpha \approx 0{,}5$).

Пусть имена представляют собой натуральные числа 1, 2, ..., n, т. е. $x_i = i$, $1 \le i \le n$. Это упрощает изложение без потери общности, так как в упорядоченной таблице имеет место однозначное соответствие между именем и его порядковым номером (индексом). Расширенное бинарное дерево (*дерево Фибоначчи*), соответствующее поиску Фибоначчи, показано на рисунке 4.2. Это дерево Фибоначчи порядка 6, т. е. корню дерева сопоставлено имя $i = F_6$.

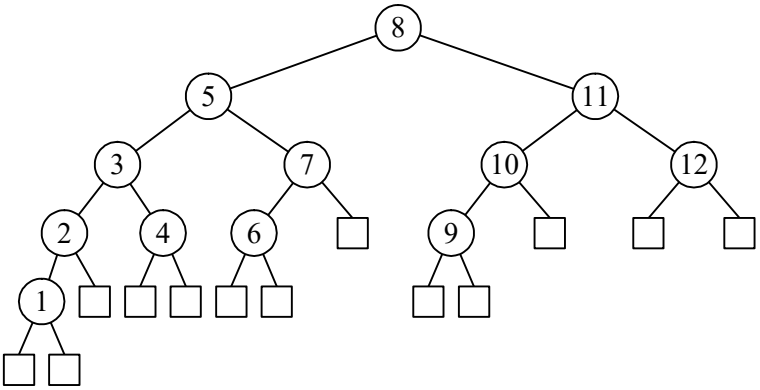


Рис. 4.2. Дерево Фибоначчи порядка 6

В общем случае дерево Фибоначчи порядка k имеет $F_{k+1} - 1$ внутренних и F_{k+1} внешних вершин (соответствуют безуспешному поиску). Строится оно следующим образом.

Если $k = 0$ или $k = 1$, дерево состоит из единственной вершины, представляющей собой внешнюю вершину.

Если $k > 1$, дерево состоит из корня, которому сопоставлено число (имя) F_k; левое поддерево представляет собой дерево Фибоначчи порядка $k - 1$; правое поддерево – дерево Фибоначчи порядка $k - 2$ с числами, увеличенными на величину F_k.

Следует отметить, что за исключением внешних вершин имена двух сыновей каждой внутренней вершины отличаются от имени отца, являющегося корнем поддерева порядка k, на одну и ту же величину – на число Фибоначчи F_{k-2} (например, для вершины 8 с сыновьями 5 и 11 имеет место $5 = 8 - F_4$ и $11 = 8 + F_4$). Если такая разница на каком-либо уровне составляет F_j, то на следующем уровне она будет равна F_{j-1} для левой ветви дерева и F_{j-2} – для правой. Например, $3 = 5 - F_3$, а $10 = 11 - F_2$.

Тогда поиск Фибоначчи ключа z в таблице $T = \{x_1, x_2, \ldots, x_n\}$ будет заключаться в следующем. Для удобства описания предполагается, что $n + 1$ представляет собой число Фибоначчи F_{k+1}. Первым исследуется имя x_i с $i = F_k$ (дерево порядка k). После каждого сравнения, не давшего равенства, если не выполнены критерии безуспешного поиска, устанавливается $i \leftarrow i \pm F_{k-2}$ и продолжается исследование поддеревьев меньшего порядка. Критериями безуспешного поиска являются: для левого поддерева – если $F_{k-2} = 0$, а для правого поддерева – если $F_{k-1} = 1$. Ясно, что нет необходимости хранить последовательность Фибоначчи, поскольку, зная два соседних числа Фибоначчи, можно однозначно восстановить всю последовательность. Поиск Фибоначчи для случая $n = F_{k+1} - 1$ представлен алгоритмом 4.5.

$i \leftarrow F_k$
$p \leftarrow F_{k-1}$
$q \leftarrow F_{k-2}$

$$\textbf{while}\ \ p \geq 1\ \textbf{and}\ q \geq 0\ \ \textbf{do case}\begin{cases} z = x_i : \textbf{return}\,(i)\ \ \text{// найдено} \\ z > x_i : \begin{cases} i \leftarrow i + q \\ p \leftarrow p - q \\ q \leftarrow q - p \end{cases} \\ z < x_i : \begin{cases} i \leftarrow i - q \\ t \leftarrow p - q \\ p \leftarrow q \\ q \leftarrow t \end{cases} \end{cases}$$

return (0) // не найдено

Алгоритм 4.5. Поиск Фибоначчи для таблиц размера $n = F_{k+1} - 1$

Чтобы распространить процедуру поиска Фибоначчи для произвольного $n \geq 1$, можно использовать следующий прием. Необходимо найти наименьшее $m \geq 0$ такое, что $n + m = F_{k+1} - 1$. Тогда первым исследуется имя x_i с $i = F_k - m$. Перед сравнением имен дополнительно необходима проверка условия $i \leq 0$ для предотвращения выхода за пределы левых поддеревьев. Чтобы избежать дополнительных затрат времени в цикле на проверку условия $i \leq 0$, можно использовать вычисленное значение m по-другому: если результат самого первого сравнения $z > x_i$, где

$i = F_k$, то устанавливается $i \leftarrow i - m$ и осуществляется переход к исследованию правого поддерева обычным поиском Фибоначчи.

Наихудшее время работы поиска Фибоначчи несколько больше обычного бинарного поиска, поскольку дерево Фибоначчи в общем случае не является полностью сбалансированным бинарным деревом. Среднее время поиска несколько меньше, чем у бинарного поиска [8]. Это связано с наличием в цикле только аддитивных операций и отсутствием мультипликативных операций. Временная сложность поиска Фибоначчи есть $O(\log n)$.

4.2.4. Интерполяционный поиск

Интерполяционный поиск предполагает равномерное распределение значений имен в некотором интервале от l до h, и исходная таблица $T = \{x_1, x_2, \ldots, x_n\}$ упорядочена по значениям имен. Поэтому, зная ключ поиска z, можно предсказать более точное положение искомого имени в таблице, чем просто в середине некоторого интервала. Положение следующего имени x_m для сравнения с z определяется делением интервала пропорционально разностям имен $x_h - x_l$ и $z - x_l$, т. е. $m = l + \left[(h-l)(z-x_l)/\right.$ $\left./(x_h - x_l)\right]$. После каждого сравнения, не давшего равенства, интервалы поиска корректируются, так же как и в алгоритме обычного бинарного поиска. Таким образом, единственным отличием интерполяционного поиска от бинарного является метод определения m.

Асимптотически интерполяционный поиск превосходит бинарный и требует в среднем $O(\log \log n)$ операций. Однако пока таблица не очень велика, данное преимущество не компенсирует требуемое для дополнительных вычислений время.

4.3. Логарифмический поиск в динамических таблицах

4.3.1. Деревья бинарного поиска

Последовательно распределенные таблицы позволяют обеспечить достаточно быстрое нахождение имен применением бинарного поиска. Однако существенным недостатком такой организации таблиц является низкая эффективность выполнения

операций включения и исключения имен относительно времени поиска, что существенно снижает общую эффективность работы с динамическими таблицами. Временная сложность операций включения и исключения составляет $O(n)$, а для бинарного поиска – $O(\log n)$.

Связная организация динамических таблиц в виде связных списков позволяет достичь высокой эффективности выполнения операций включения и исключения (время их выполнения не зависит от размера таблицы). Однако к таким таблицам нельзя применить бинарный поиск (поскольку не обеспечивается прямой доступ к именам таблицы), а возможен только последовательный поиск с временной сложностью $O(n)$.

Для динамических таблиц необходима такая структура данных, которая была бы приспособлена как к методу бинарного поиска, так и к эффективному выполнению операций включения и исключения. Такой структурой данных является дерево бинарного поиска.

Деревом бинарного поиска (ДБП) над именами $x_1, x_2, ..., x_n$ называется расширенное бинарное дерево, все внутренние вершины которого помечены именами из таблицы $T = \{x_1, x_2, ..., x_n\}$ таким образом, что симметричный порядок прохождения вершин совпадает с естественным порядком имен. Каждая из $n + 1$ внешних вершин (листьев) соответствует промежутку в таблице. Поскольку симметричный порядок прохождения вершин является естественным порядком, для каждой вершины x_i все имена в левом поддереве с корнем x_i предшествуют x_i в естественном порядке и все имена в правом поддереве с корнем x_i следуют за x_i в естественном порядке. На рисунке 4.3 представлено ДБП для таблицы $T = \{x_1, x_2, \ldots, x_9\}$, $x_1 \le x_2 \le \ldots \le x_9$.

Каждая внутренняя вершина ДБП может быть представлена узлом, состоящим из полей *left*, *info* и *right*, где *left* и *right* содержат указатели на левого и правого сыновей соответственно, а поле *info* содержит имя, хранящееся в узле (узловое представление дерева). Доступ к дереву обеспечивается с помощью внешнего указателя *root* на его корень. Очевидно, что для пустого дерева *root* = Λ. Если указатель (*left* или *right*) имеет значение Λ, это означает, что у узла нет соответствующего сына, т. е. он указывает на пустое поддерево.

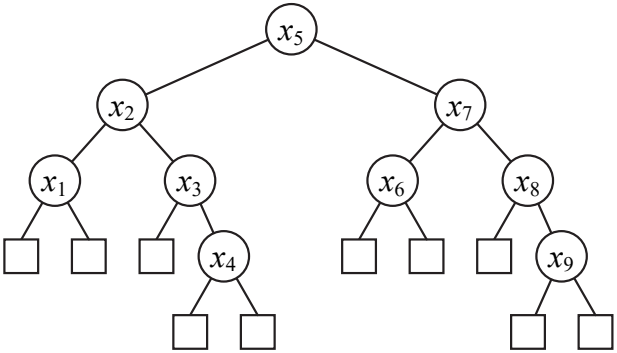


Рис. 4.3. Дерево бинарного поиска

Понятие внешних вершин носит формальный характер, поэтому соответствующие им узлы можно не создавать. При разработке некоторых алгоритмов, чтобы упростить обработку граничных условий, можно явно хранить и использовать единственный фиктивный внешний узел, соответствующий всем внешним узлам дерева, т. е. фиктивный узел будет выполнять функции сторожа. В этом случае значение указателя на этот фиктивный узел будет играть роль значения Λ.

При необходимости в структуру узла можно добавить поле *father*, которое указывает на отца данного узла, для облегчения движения от потомков к предкам. Очевидно, что для корня дерева значение этого поля равно Λ.

Поиск *z*. Поиск имени *z* в ДБП осуществляется сравнением *z* с именем, стоящим в корне. Если дерево пустое, то *z* в таблице отсутствует и поиск завершается безуспешно. Если *z* совпадает с именем в корне, поиск завершается успешно. Если *z* предшествует имени в корне, поиск продолжается рекурсивно в левом поддереве. Если *z* следует за именем в корне, поиск продолжается рекурсивно в правом поддереве корня. Очевидно, что успешный поиск завершается во внутреннем узле ДБП, а безуспешный – во внешнем узле. Процедуру поиска имени *z* в таблице, организованной в виде ДБП, можно представить алгоритмом 4.6.

Распечатка. Очевидно, что операцию распечатки, предполагающую печать имен из таблицы в естественном порядке, легко выполнить использованием симметричного прохождения бинарных деревьев.

$p \leftarrow root$
while $p \neq \Lambda$ **do**
$$\textbf{case}\begin{cases} z = p.info: \textbf{return}(p) & \text{// найдено: } p \text{ указывает на } z \\ z < p.info: p \leftarrow p.left \\ z > p.info: p \leftarrow p.right \end{cases}$$
return(Λ) // не найдено

Алгоритм 4.6. Поиск в дереве бинарного поиска

Включение *z*. Включение имени *z* в таблицу выполняется в том случае, если поиск *z* в таблице завершается безуспешно. Для таблиц, организованных в виде ДБП, включение имени *z* соответствует добавлению в дерево нового узла. Новый узел всегда добавляется как лист вместо внешнего узла, соответствующего промежутку, где могло бы находиться *z*, если бы оно входило в таблицу. При этом добавляемый узел должен быть связан с последним узлом, пройденным во время безуспешного поиска *z*.

Для реализации включения необходимо обеспечить сохранение адреса отца добавляемого узла, поскольку в алгоритме поиска указатель *p* после выхода из цикла **while** имеет значение Λ. Для этого достаточно добавить внешний указатель *q*, в котором запоминается предыдущее значение указателя *p*. Кроме того, необходимо предусмотреть возможность добавления нового узла в первоначально пустое дерево, поскольку установка связей между узлами в этом случае отличается от остальных. В результате получается процедура включения, представленная алгоритмом 4.7.

Возможна и рекурсивная реализация операции включения имени *z* в ДБП, представленная алгоритмом 4.8. Рекурсивная функция *INSERT*(*z*, *t*) выдает в качестве значения указатель на дерево, в которое добавляется *z*. Таким образом, для добавления *z* в дерево с указателем корня *root* достаточно выполнить операцию *root* ← *INSERT*(*z*, *root*). Следует отметить, что функция автоматически обрабатывает ситуацию, когда до включения нового узла ДБП было пустым. Очевидно, что рекурсивная реализация включения с точки зрения времени работы уступает нерекурсивному алгоритму.

$p \leftarrow root$
$q \leftarrow \Lambda$ // q – указатель отца добавляемого узла

while $p \neq \Lambda$ **do case** $\begin{cases} z = p.info: \text{ // } z \text{ уже в таблице} \\ z < p.info: \begin{cases} q \leftarrow p \\ p \leftarrow p.left \end{cases} \\ z > p.info: \begin{cases} q \leftarrow p \\ p \leftarrow p.right \end{cases} \end{cases}$

// z нет в таблице, добавить узел
$new(p)$
$p.info \leftarrow z$
$p.left \leftarrow p.right \leftarrow \Lambda$
if $root = \Lambda$
  then $root \leftarrow p$ // узел добавляется в пустое ДБП
  else if $z < q.info$
    then $q.left \leftarrow p$ // левый сын
    else $q.right \leftarrow p$ // правый сын

Алгоритм 4.7. Включение нового узла в дерево бинарного поиска

function $INSERT(z, t)$
  $p \leftarrow t$
  if $p = \Lambda$
    then $\begin{cases} new(p) \\ p.info \leftarrow z \\ p.left \leftarrow p.right \leftarrow \Lambda \end{cases}$
    else case $\begin{cases} z = p.info: \text{ // } z \text{ уже в таблице} \\ z < p.info: p.left \leftarrow INSERT(z, p.left) \\ z > p.info: p.right \leftarrow INSERT(z, p.right) \end{cases}$
  $INSERT \leftarrow p$
return

Алгоритм 4.8. Рекурсивная функция включения нового узла в ДБП

Исключение *z*. Исключение сложнее включения, поскольку в ДБП исключается внутренний узел, который может быть листом, иметь одного или двух сыновей. Если исключаемый узел с именем z является листом или имеет только одного сына, удаление выполняется достаточно просто – при исключении узла z его сын (если он есть) становится сыном отца узла z (рис. 4.4*а*). Если же исключаемый узел с именем z имеет двух сыновей, его прямо

удалить нельзя. В этом случае в таблице необходимо найти имя y_1, непосредственно предшествующее имени z, или имя y_2, непосредственно следующее за именем z в естественном порядке. Очевидно, что оба имени принадлежат узлам, имеющим не более одного сына. Далее имя z исключается заменой его либо именем y_1, либо именем y_2, а затем удалением узла, который содержал y_1 или y_2 соответственно (рис. 4.4*б*).

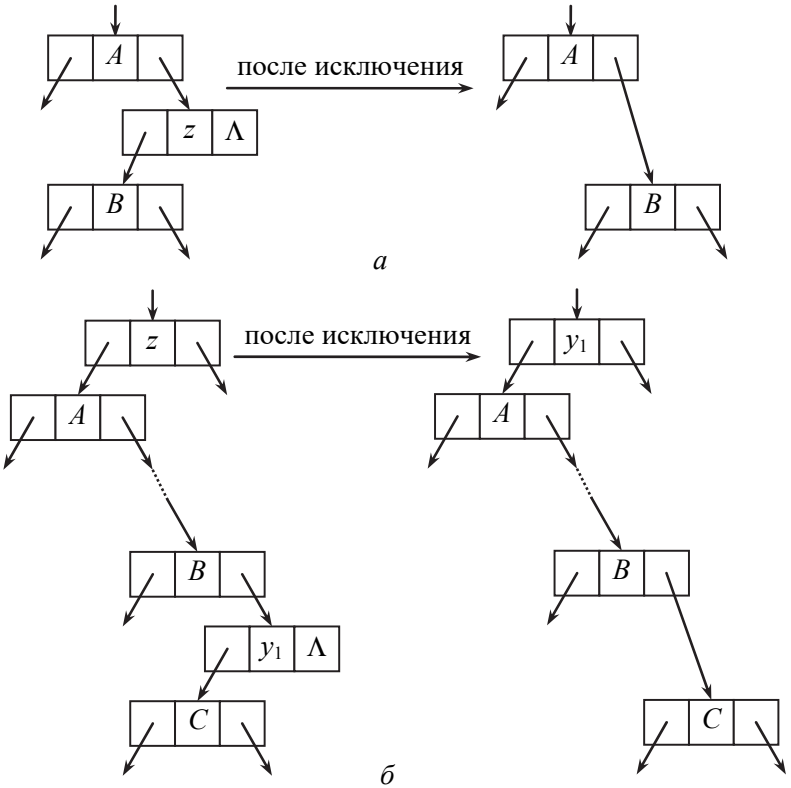


Рис. 4.4. Исключение узла из дерева бинарного поиска:
а – исключаемый узел имеет не более одного сына;
б – исключаемый узел имеет двух сыновей.

4.3.2. АВЛ-деревья

Для динамических таблиц последовательность операций включения и исключения носит случайный характер. В результа-

те порождаются случайные деревья бинарного поиска. Если алгоритмом включения последовательно включаются n имен в первоначально пустое дерево и если все перестановки входной последовательности имен равновероятны, то среднее время поиска в таких деревьях равно $(2 \ln 2) \log n + O(1)$, т. е. примерно в 1,4 раза больше, чем среднее время поиска в полностью сбалансированном дереве с n вершинами, равное $\log n + O(1)$. Для многих приложений увеличение времени поиска примерно на 40% можно считать вполне допустимым, однако ДБП без ограничений на практике не надежны. В худшем случае могут порождаться вырожденные ДБП, являющиеся, по существу, линейными списками, в которых время поиска становится линейным относительно величины таблицы, а не логарифмическим. Поэтому в ряде приложений необходимы специальные методы балансировки, гарантирующие логарифмическое время поиска даже в худшем случае без требований каких-либо допущений о последовательности операций включения и исключения.

Наиболее очевидный путь для того, чтобы динамическое дерево не стало асимметричным, – поддерживать его полностью сбалансированным во все моменты времени. Тогда если при включении в дерево новой вершины с именем C дерево становится не полностью сбалансированным, то оно должно быть перестроено (рис. 4.5). К сожалению, такое преобразование обычно меняет все дерево, и ни одно из отношений отец – сын не остается неизменным, т. е. такая операция перестройки в общем случае требует времени, пропорционального числу вершин.

Поэтому необходимы такие методы балансировки, которые после выполнения операций включения и исключения позволяют преобразовать дерево локальными изменениями вдоль одного пути от корня к листу, что требует времени $O(\log n)$. Для обеспечения такой гибкости деревья должны иметь возможность отклоняться от полностью сбалансированных деревьев, но в такой малой степени, что среднее время поиска в них было бы лишь немногим больше, чем в полностью сбалансированном бинарном дереве.

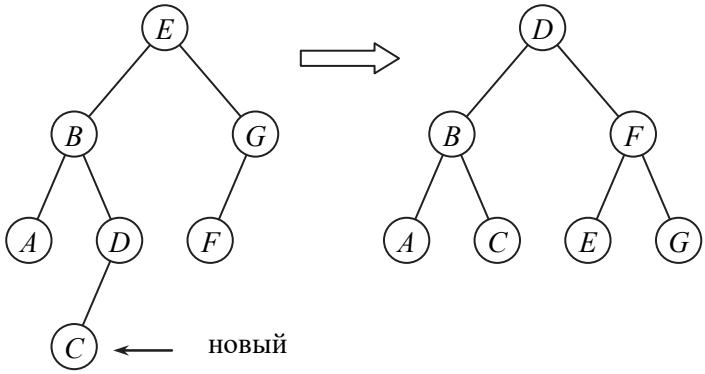


Рис. 4.5. Восстановление полной сбалансированности бинарного дерева

Пусть $h(T)$ обозначает высоту бинарного дерева T, т. е. длину самого длинного пути от корня к листу. Высота дерева с единственной вершиной равна 0, для удобства высоту пустого дерева считают равной –1.

Дерево бинарного поиска называется *АВЛ-деревом* (в честь его авторов Г. М. Адельсона-Вельского и Е. М. Ландиса), или *сбалансированным по высоте*, если два поддерева корня T_l (левое) и T_r (правое) удовлетворяют следующим условиям:

1) $|h(T_l) - h(T_r)| \le 1$;

2) T_l и T_r сбалансированы по высоте.

Таким образом, в АВЛ-дереве для каждой вершины высоты ее левого и правого поддеревьев отличаются не более чем на единицу. Пустое дерево, не имеющее ни корня, ни поддеревьев, удовлетворяет этим условиям и, следовательно, является АВЛ-деревом. Дерево с единственной вершиной также является АВЛ-деревом. Пример АВЛ-дерева показан на рисунке 4.6*а*. Дерево на рисунке 4.6*б* не является АВЛ-деревом, поскольку высота левого (пустого) поддерева вершины D равна –1, а правого – единице, т. е. высоты поддеревьев отличаются на 2.

Высота наиболее асимметричного АВЛ-дерева с n вершинами составляет примерно $1{,}44 \log n$, т. е. время поиска в худшем случае приблизительно на 44% больше, чем среднее время поиска в полностью сбалансированных бинарных деревьях. Среднее же время поиска в наиболее асимметричных АВЛ-деревьях примерно на 4% больше. Учитывая тот факт, что сильно асиммет-

ричной является только малая часть АВЛ-деревьев, среднее время поиска в АВЛ-деревьях, усредненное по всем таким деревьям, равно $\log n + O(1)$.

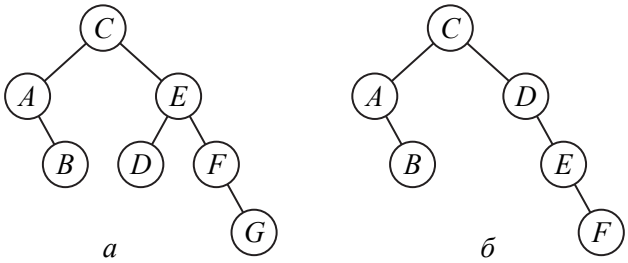


Рис. 4.6. Деревья бинарного поиска:
а – АВЛ-дерево; *б* – не АВЛ-дерево.

В результате выполнения операции включения или исключения сбалансированность дерева может быть нарушена. Основной операцией, изменяющей структуру дерева для восстановления сбалансированности, является операция *вращения*, которая представляет собой локальную операцию и сохраняет свойство упорядоченности вершин дерева (имен в таблице, представленной АВЛ-деревом) в соответствии с симметричным прохождением. На рисунке 4.7 показаны два взаимно обратных вращения: левое и правое (α, β и γ обозначают произвольные поддеревья). В результате правого вращения вершин *A* и *B* (правое вращение вокруг вершины *B*) вершина *B* становится правым сыном вершины *A*, а бывший правый сын вершины *A* (поддерево β) становится левым сыном вершины *B*. Таким образом, вершина *A* с левым поддеревом α поднимается на один уровень вверх, а вершина *B* с правым поддеревом γ опускается на один уровень вниз, поддерево β остается на том же уровне. Аналогично (в зеркальном отражении) работает левое вращение. Операции вращения выполняются за время $O(1)$ и заключаются в изменении нескольких указателей.

В ряде случаев для восстановления сбалансированности дерева возникает необходимость в последовательном выполнении двух вращений, которые для удобства можно объединить в одну операцию, называемую *двойным вращением*.

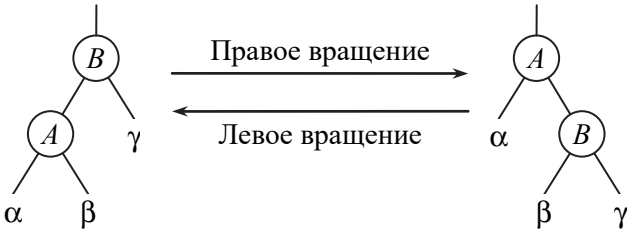


Рис. 4.7. Операции вращения

На рисунке 4.8 показано двойное вращение (α, β, γ и δ произвольные поддеревья), когда сначала выполняется правое вращение вокруг вершины *C*, а затем левое вращение вокруг вершины *A*. В результате двойного вращения вершина *A* с левым поддеревом α опускается на один уровень вниз, вершина *B* поднимается на два уровня вверх, уровень вершины *C* с правым поддеревом δ остается прежним, поддеревья β и γ поднимаются на один уровень вверх. Имеется также симметричный вариант двойного вращения.

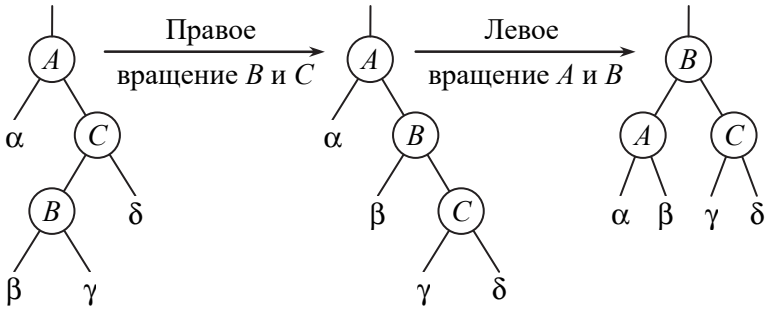


Рис. 4.8. Двойное вращение

Для контроля требований сбалансированности по высоте в процессе выполнения операций включения и исключения необходима информация о соотношении высот поддеревьев каждой вершины. Чтобы не рассматривать не затронутые локальными преобразованиями поддеревья для определения их высот, лучше всего иметь такую информацию (назовем ее *балансом*) в каждой вершине дерева. При узловом представлении дерева в структуру узла следует добавить специальное поле для хранения баланса.

Баланс некоторой вершины дерева определяется как разность высот левого и правого поддеревьев. В АВЛ-деревьях, поскольку высоты поддеревьев каждой вершины отличаются не более чем на единицу, баланс вершины может иметь только три значения: 1, 0 или –1 в зависимости от того, что высота ее левого поддерева больше, равна или меньше высоты правого поддерева. На рисунках баланс будем записывать внутри вершин.

Включение новой вершины. Для включения новой вершины *new* применяется обычный алгоритм включения в дерево бинарного поиска. После включения вершины путь из корня в лист проходится в обратном направлении. Для обеспечения возвращения необходимо модифицировать алгоритм включения в части поиска так, чтобы сохранить спускающийся путь в стеке. Другой подход предполагает добавление в структуру узла дерева специального поля *father*, указывающего на отца данной вершины. В этом случае нет необходимости в сохранении спускающегося пути в стеке. Действия, предпринимаемые при прохождении каждой вершины по восходящему пути, зависят от баланса в этой вершине и в случае, если баланс равен 1 или –1, от направления последнего шага или двух последних шагов вверх по этому пути. Возможны следующие ситуации.

1. Если баланс текущей вершины равен 0, то его значение меняется на –1, если последний шаг начинался из правого сына, и на 1, если он начинался из левого сына. Поскольку высота поддерева, корнем которого является текущая вершина, увеличивается, продолжается движение вверх по пути с информацией об увеличении высоты дерева. Процедура заканчивается, если будет достигнут корень дерева или будет обработана ситуация 2 или 3.

2. Если баланс текущей вершины равен 1 или –1 и последний шаг начинался из более короткого из двух поддеревьев текущей вершины, то значение баланса меняется на 0. Поскольку высота поддерева, корнем которого является текущая вершина, не изменяется, движение вверх прекращается и процедура заканчивается.

3. Если баланс текущей вершины равен 1 или –1 и последний шаг начинался из более высокого из двух поддеревьев текущей вершины, то:

а) если последние два шага делались в одном направлении (оба из левых сыновей или оба из правых сыновей), то выполня-

ется соответствующее вращение; на рисунке 4.9*а* показан один из симметричных вариантов такого преобразования, когда два последних шага к вершине *A* делались из правых сыновей;

б) если два последних шага делались в противоположных направлениях, то выполняется соответствующее двойное вращение; на рисунке 4.9*б* показан один из симметричных вариантов, когда два последних шага к вершине *A* делались в противоположных направлениях.

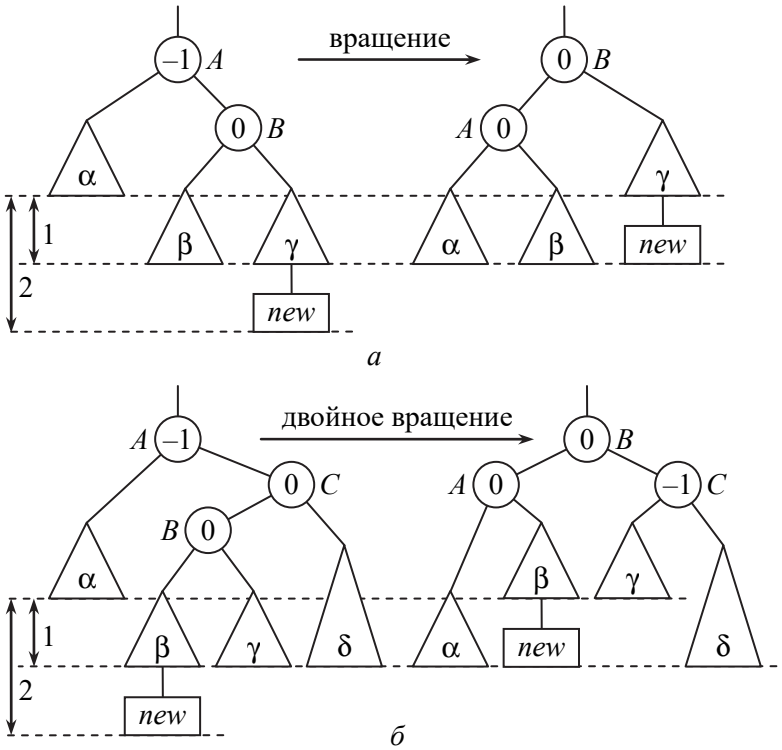


Рис. 4.9. Восстановление сбалансированности АВЛ-дерева после включения:
а – два последних шага к вершине *A* сделаны из правых сыновей;
б – два последних шага к вершине *A* сделаны сначала из левого сына (от *B* к *C*), потом из правого (от *C* к *A*).

В обоих случаях высота поддерева после балансировки остается той же, что и до включения новой вершины, т. е. эти преобразования не влияют на баланс расположенных выше вершин.

Поэтому продвижение вверх прекращается и процедура заканчивается. Таким образом, при включении новой вершины может потребоваться не более одного преобразования структуры дерева (вращения или двойного вращения).

Исключение вершины. Рассмотренный ранее алгоритм исключения гарантирует то, что удаляемая вершина имеет не менее одного пустого поддерева. Если удаляемая вершина имеет ровно одно пустое поддерево, то согласно ограничению на высоту поддеревьев другое поддерево состоит только из одной вершины. В этом случае влияние на сбалансированность дерева по высоте такое же, как если бы исключался лист. Когда лист исключается, поддерево, состоящее именно из этой вершины, теряет одну единицу высоты. Необходимо пройти путь от этого исключенного листа вверх до корня дерева, проверяя в каждой вершине, какое влияние на большее дерево оказывает уменьшение высоты поддерева. Действия, предпринимаемые в каждой вершине, расположенной на восходящем пути, зависят от баланса текущей вершины, направления последнего шага и в некоторых случаях – от баланса сына текущей вершины. Так же как и при включении, в ряде случаев преобразование восстанавливает высоту поддерева до той, которая была до исключения (процедура заканчивается); в других случаях продолжается движение вверх по пути к корню с информацией об уменьшении высоты поддерева. Возможны следующие ситуации.

1. Если баланс текущей вершины равен 0, то уменьшение высоты любого ее поддерева не влияет на высоту поддерева, имеющего корнем текущую вершину. Изменяется значение баланса в зависимости от того, высота какого поддерева уменьшилась (на 1 или –1). Процедура заканчивается.

2. Если баланс текущей вершины равен 1 или –1 и последний шаг начинался из более высокого из двух поддеревьев, то значение баланса меняется на 0 и продолжается движение вверх с информацией об уменьшении высоты поддерева.

3. Если баланс текущей вершины равен 1 или –1 и последний шаг начинался из более короткого из двух поддеревьев, то в текущей вершине нарушается соотношение высот поддеревьев. Различаются три подслучая и их симметричные варианты в соответствии с балансом другого сына текущей вершины. На рисунке 4.10 представлены варианты для ситуации, когда произошло

уменьшение высоты левого поддерева α текущей вершины *A* (показано пунктирными линиями). Следовательно, для распознавания подслучаев необходимо анализировать баланс правого сына текущей вершины.

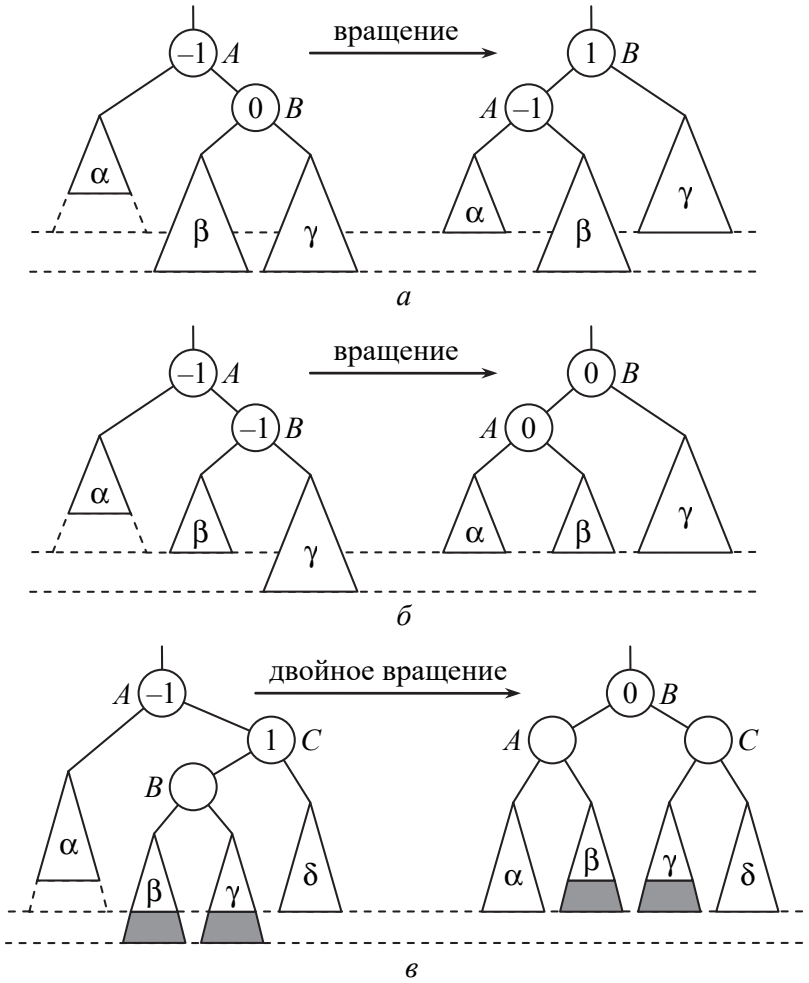


Рис. 4.10. Восстановление сбалансированности АВЛ-дерева после исключения:
а – баланс правого сына *B* текущей вершины *A* равен 0;
б – баланс правого сына *B* текущей вершины *A* равен –1;
в – баланс правого сына *B* текущей вершины *A* равен 1.

а) если баланс правого сына текущей вершины равен 0 (рис. 4.10*а*), то выполняется соответствующее вращение, которое восстанавливает соотношение высот поддеревьев в текущей вершине без изменения высоты поддерева, имеющего корнем текущую вершину. Процедура заканчивается;

б) если баланс правого сына текущей вершины равен –1 (рис. 4.10*б*), то выполняется соответствующее вращение, которое восстанавливает соотношение высот поддеревьев в текущей вершине. Продолжается движение вверх с информацией об уменьшении высоты поддерева, корнем которого является текущая вершина;

в) если баланс правого сына текущей вершины равен 1, (рис. 4.10*в*), то выполняется соответствующее двойное вращение, которое восстанавливает соотношение высот поддеревьев в текущей вершине. Продолжается движение вверх с информацией об уменьшении высоты поддерева, корнем которого является текущая вершина. На рисунке 4.10*в* баланс вершины B может иметь любое из значений 1, 0 или –1, но, по крайней мере, одно из двух поддеревьев вершины B должно иметь высоту, обозначенную закрашенной областью. Балансы вершин A и C после преобразования зависят от высоты двух закрашенных поддеревьев.

Следует обратить внимание на то, что в противоположность операции включения, требующей не более одного преобразования, исключение может требовать в худшем случае $\lfloor h/2 \rfloor$ преобразований, где h – высота всего дерева. Однако в большинстве случаев процедура исключения не потребует прослеживания восходящего пути до самого корня. Обычно она завершается после некоторого числа шагов, не зависящего от высоты дерева.

4.3.3. Красно-черные деревья

Как и АВЛ-деревья, *красно-черные деревья* (*RB-деревья*) являются одним из типов сбалансированных деревьев бинарного поиска, используемых для организации динамических таблиц. В таких деревьях специальные операции балансировки гарантируют, что высота дерева не превзойдет $O(\log n)$, а следовательно, и табличные операции (поиск, включение и исключение) будут выполняться за время, не превышающее $O(\log n)$.

RB-дерево – это дерево бинарного поиска, вершины которого разделены на красные и черные. При этом должны выполняться определенные требования, которые гарантируют, что уровни двух любых листьев отличаются не более чем в два раза, поэтому дерево называется сбалансированным.

Каждая вершина RB-дерева может быть представлена узлом, состоящим из полей *color* (цвет), *info* (имя), *left* (указатель на левого сына) и *right* (указатель на правого сына). Для удобства считается, что RB-дерево является расширенным бинарным деревом, т. е. значения Λ, хранящиеся в полях *left* и *right*, являются ссылками на фиктивные внешние узлы (листья) дерева, которые будем называть Λ-сыновьями. В таком случае каждый узел, содержащий имя, имеет двух сыновей и является внутренним узлом. При разработке алгоритмов, чтобы упростить обработку граничных условий, можно использовать единственный фиктивный узел, соответствующий всем внешним узлам бинарного дерева (Λ-сыновьям). Этот фиктивный узел имеет ту же структуру, что и обычный узел дерева. Тогда достаточно все указатели Λ заменить указателями на фиктивный узел.

Дерево бинарного поиска называется красно-черным деревом (RB-деревом), если оно обладает следующими свойствами, называемыми *RB-свойствами*:

1) каждая вершина – либо красная, либо черная;

2) каждый лист (внешняя вершина) – черный;

3) если вершина красная, оба ее сына черные;

4) все пути, идущие от корня к листьям, содержат одинаковое количество черных вершин.

Будем предполагать, что в RB-деревьях корень черный, и поддерживать это свойство при преобразовании структуры дерева. Пример RB-дерева показан на рисунке 4.11. Черные вершины изображены как темные, красные – как светлые.

Все пути, ведущие от произвольной вершины x к листьям, содержат одно и то же число черных вершин. Число черных вершин в любом из них (сама вершина x в это число не входит) называется *черной высотой* вершины x и обозначается $bh(x)$. Например, $bh(8) = 2$, $bh(6) = 2$, а $bh(14) = 3$. Черная высота дерева определяется черной высотой его корня. RB-дерево с n внутренними вершинами имеет высоту не более $2\log(n + 1)$. Поэтому

поиск выполняется за время $O(\log n)$, так как время его выполнения есть $O(h)$ для дерева высоты h.

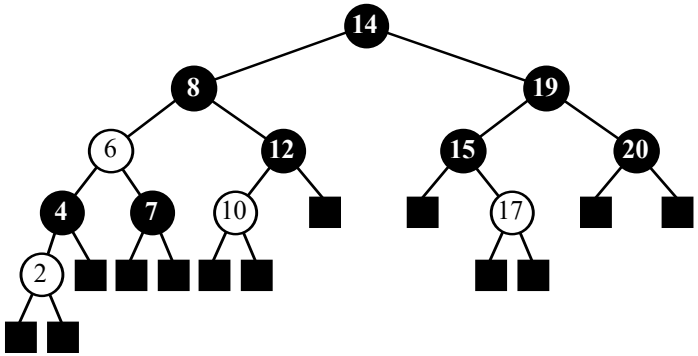


Рис. 4.11. Красно-черное дерево

Операции включения и исключения изменяют дерево, поэтому результат может не обладать RB-свойствами. Для восстановления этих свойств требуется перекрасить некоторые вершины и изменить структуру дерева. Структура дерева меняется с помощью локальных операций левого и правого вращений, аналогичных вращениям для АВЛ-деревьев.

Включение новой вершины. Для включения новой вершины применяется обычный алгоритм включения имени z в дерево бинарного поиска. Новая вершина окрашивается красным цветом. После этого необходимо восстановить RB-свойства, которые могли быть нарушены включением. Для этого путь из корня в лист проходится в обратном направлении.

После включения новой красной вершины выполняются все RB-свойства, кроме одного – новая красная вершина может иметь красного отца. При этом следует помнить, что новая красная вершина имеет двух черных Λ-сыновей. Следует отметить, что черная высота дерева при включении не меняется, так как новая красная вершина с черными Λ-сыновьями замещает внешнюю черную вершину (лист). Если новая вершина включается в пустое дерево, то она перекрашивается в черный цвет и процесс включения завершается. Таким образом, если новая красная вершина имеет черного отца, никакие RB-свойства не нарушаются и никакие действия по балансировке не производятся.

Если же отцом новой красной вершины является красная вершина, то действия, предпринимаемые при прохождении вершин по восходящему пути, зависят от направления движения (от левого или правого сына) и цвета брата отца (дяди) новой вершины. Пусть *p* указывает на исследуемую на текущем этапе вершину (на начальном этапе – это добавленная красная вершина), *f* – на вершину-отца, *u* – на вершину-дядю и *g* – на вершину-деда вершины *p*. Тогда если вершина *p* – корень дерева или имеет черного отца, то движение вверх прекращается и процесс завершается. Если же вершина *p* имеет красного отца *f* и не является корнем дерева, то возможны шесть случаев. Три из них симметричны трем другим, их различие в том, является ли отец *f* вершины *p* левым или правым сыном своего отца (вершины *g*). Рассмотрим один из симметричных вариантов, а именно: отец *f* вершины *p* является левым сыном своего отца *g*. Тогда возможны следующие случаи, проиллюстрированные на рисунке 4.12, где все поддеревья α, β, γ, δ и ε имеют черный корень и одинаковую черную высоту.

Случай 1. Если вершина *u* красная (вершина *f* тоже красная), то вершины *f* и *u* перекрашиваются в черный цвет, а вершина *g* – в красный. При этом число черных вершин на любом пути от корня к листьям останется прежним. Действия не зависят от того, является ли вершина *p* правым (рис. 4.12*а*) или левым (рис. 4.12*б*) сыном вершины *f*. Нарушение RB-свойств возможно в единственном месте нового дерева: у красной вершины *g* может быть красный отец. Поэтому процесс движения вверх продолжается, присвоив указателю *p* значение указателя *g*.

Случай 2. Если вершина *u* черная и красная вершина *p* является правым сыном красной вершины *f* (рис. 4.12*в*), то выполняется левое вращение вокруг вершины *f* и указателю *p* присваивается значение указателя *f*. В результате этих действий случай 2 сводится к случаю 3. Следует отметить, что после вращения количество черных вершин на путях от корня к листьям остается прежним.

Случай 3. Если вершина *u* черная и красная вершина *p* является левым сыном красной вершины *f* (рис. 4.12*г*), которая, в свою очередь, является левым сыном черной вершины *g* (вершина *g* черная, поскольку ее сыновья *f* и *u* разных цветов, а это возможно только для черных вершин), то выполняется правое вра-

щение вокруг вершины *g*. Затем для устранения нарушений RB-свойств перекрашиваются: вершина *f* в черный, а вершина *g* – в красный цвет. Поскольку теперь отец (вершина *f*) вершины *p* черный, движение вверх прекращается и процесс завершается.

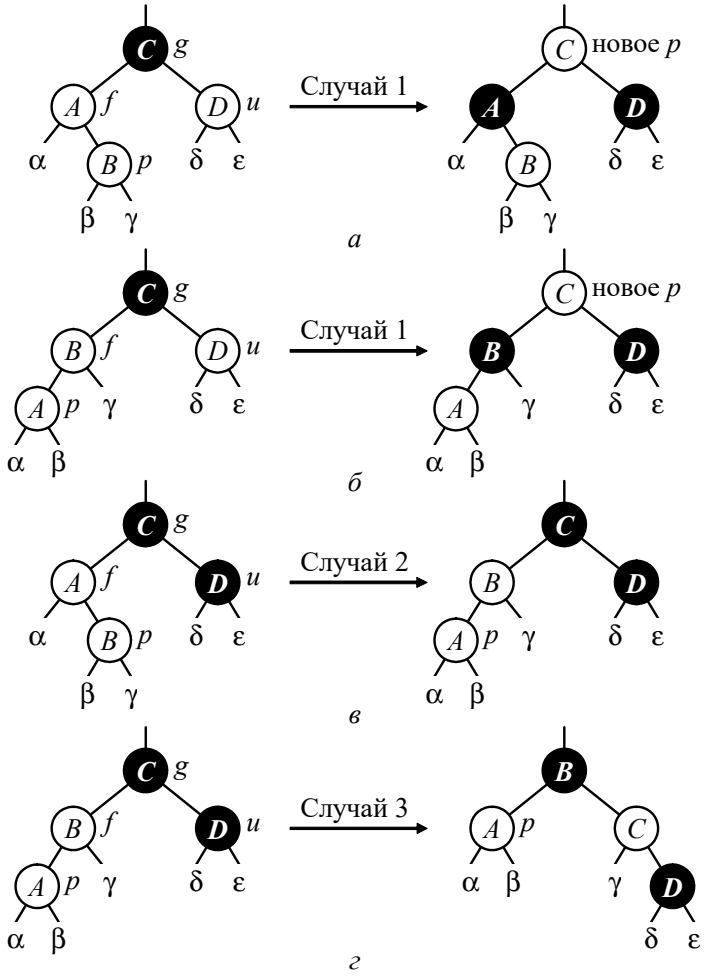


Рис. 4.12. Восстановление сбалансированности RB-дерева после включения:
а – вершина *u* красная, вершина *p* – правый сын вершины *f*;
б – вершина *u* красная, вершина *p* – левый сын вершины *f*;
в – вершина *u* черная и вершина *p* – правый сын вершины *f*;
г – вершина *u* черная и вершина *p* – левый сын вершины *f*.

Анализ рассмотренных случаев показывает, для чего корень дерева необходимо поддерживать черным: если отец новой вершины красный, он не может быть корнем, и вершина-дед всегда существует (а значит, всегда есть вершина-дядя). Для поддержания этого свойства на случай, если цвет корня поменялся в результате преобразований, процесс включения необходимо завершать операцией окраски корня дерева в черный цвет.

Поскольку высота RB-дерева есть $O(\log n)$, то операция включения требует времени $O(\log n)$. Причем движение вверх по дереву продолжается только для случая 1 (при этом перекрашиваются только некоторые вершины). В случаях 2 и 3 после выполнения вращений процесс движения прекращается, при этом выполняется не более двух вращений.

Исключение вершины. Из дерева всегда исключается вершина, имеющая не более одного непустого поддерева, т. е. вершина, имеющая одного или двух Λ-сыновей. Следует отметить, что при исключении красной вершины черные высоты не меняются и красные вершины не могут стать смежными, т. е. RB-свойства не нарушаются. Поэтому нарушение RB-свойств может вызвать только исключение черной вершины. В этом случае любой проходивший через нее путь теперь содержит на одну черную вершину меньше. Поэтому для восстановления RB-свойств необходимо пройти путь от исключенной вершины вверх до корня дерева с информацией об уменьшении черной высоты поддерева, предпринимая необходимые действия.

Пусть p указывает на исследуемую на текущем этапе вершину (на начальном этапе – это вершина, занявшая место исключенной вершины, т. е. либо единственный сын исключенной вершины, либо фиктивная вершина, если у исключенной вершины были только Λ-сыновья), f – на вершину-отца и w – на вершину-брата вершины p. Вершина p является корнем поддерева, черная высота которого уменьшилась на единицу в результате исключения вершины. Тогда если вершина p – корень дерева или окрашена в красный цвет, то для восстановления RB-свойств достаточно перекрасить ее в черный цвет. В результате движение вверх прекращается и процесс завершается. Если же вершина p черная и не является корнем дерева, то возможны восемь случаев. Четыре из них симметричны четырем другим, их разница в том, является ли черная вершина p левым или правым сыном

вершины *f*. Рассмотрим один из симметричных вариантов, а именно – вершина *p* является левым сыном вершины *f*. Тогда возможны следующие случаи, представленные на рисунке 4.13, где α, β, γ, δ, ε и ζ обозначают произвольные поддеревья (возможно, и пустые), серым цветом обозначены вершины, которые могут быть и красными, и черными.

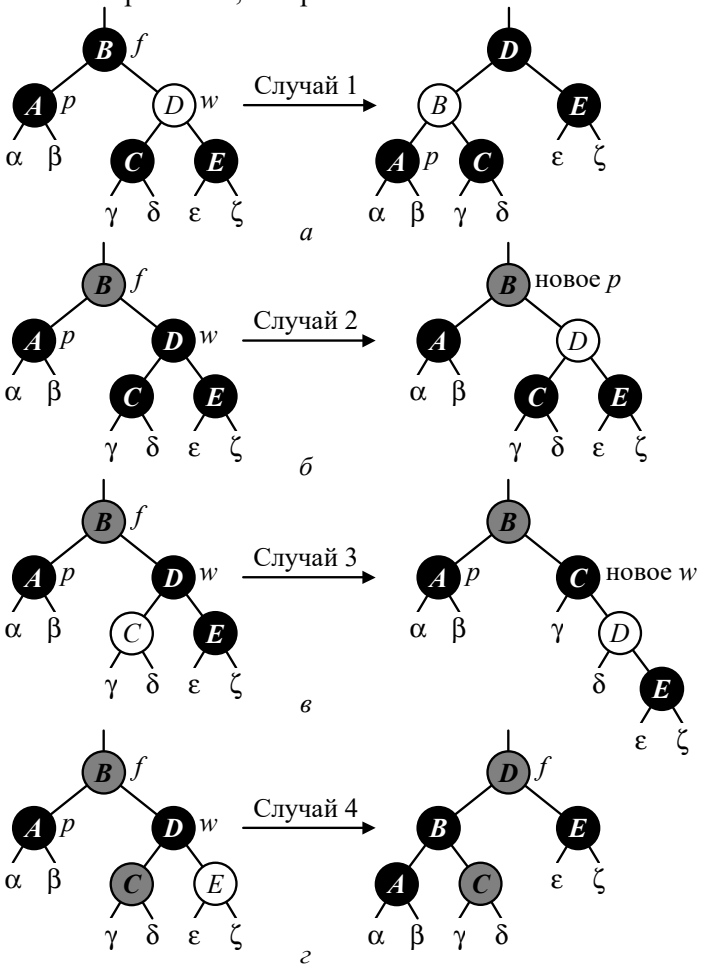


Рис. 4.13. Восстановление сбалансированности RB-дерева после исключения:
а – вершина *w* красная; *б* – вершина *w* черная и оба ее сына черные;
в – вершина *w* черная, ее левый сын красный, правый сын черный;
г – вершина *w* черная и ее правый сын красный.

Случай 1. Если вершина w красная (в этом случае их отец f окрашен в черный цвет и оба сына вершины w – черные), то вершина w окрашивается в черный цвет, а вершина f – в красный и производится левое вращение вокруг вершины f (рис. 4.13*а*). В результате этих действий случай 1 сводится к одному из случаев 2, 3 или 4.

Случай 2. Если вершина w черная и оба ее сына черные, то вершина w перекрашивается в красный цвет (рис. 4.13*б*). В результате черная высота поддерева с корнем f уменьшается на единицу, так как поддерево с корнем p уже имело уменьшенную черную высоту, а перекрашивание вершины w уменьшило черную высоту поддерева с корнем w. Поэтому продолжается движение вверх с информацией об уменьшении черной высоты поддерева присваиванием указателю p значения указателя f. Следует отметить, что если процесс попал в случай 2 из случая 1, то вершина p (новое значение p) имеет красный цвет. Поэтому перекрашивание вершины p в черный цвет восстанавливает RB-свойства (ранее уменьшенная черная высота увеличивается на единицу), движение вверх прекращается и процесс завершается.

Случай 3. Если вершина w черная, ее левый сын красный, а правый – черный, то вершина w перекрашивается в красный цвет, левый сын – в черный, а затем применяется правое вращение вокруг вершины w (рис. 4.13*в*). В результате новым братом w вершины p будет черная вершина с красным правым сыном, что сводит случай 3 к случаю 4.

Случай 4. Если вершина w черная, а ее правый сын красный, то вершина w окрашивается в цвет вершины f, затем вершина f и правый сын вершины w окрашиваются в черный цвет и производится левое вращение вокруг вершины f (рис. 4.13*г*). В результате восстанавливаются RB-свойства, движение вверх прекращается и процесс завершается.

Поскольку высота RB-дерева есть $O(\log n)$, то операция исключения требует времени $O(\log n)$. Причем как только обнаруживается случай 1, 3 или 4, движение вверх прекращается и процесс завершается. При этом выполняется $O(1)$ операций и самое большее три вращения. До этого возможно несколько повторений случая 2, но при каждом повторении указатель p перемещается вверх по дереву и никакие вращения не производятся. Таким образом, при восстановлении RB-свойств после исключения

вершины производится не более трех вращений. Для сравнения – в АВЛ-деревьях может потребоваться $\lfloor h/2 \rfloor$ вращений для дерева высоты h.

Как при включении, так и при исключении требуется пройти путь из корня в лист в обратном направлении. Для обеспечения возвращения необходимо модифицировать этап поиска так, чтобы сохранить спускающийся путь в стеке. Другой подход предполагает добавление в структуру узла специального поля *father*, указывающего на отца данной вершины. В этом случае нет необходимости в сохранении спускающегося пути в стеке.

4.3.4. Цифровой поиск

Рассмотренные ранее методы поиска были основаны на сравнении имен, т. е. на предположении о том, что на пространстве имен S определен линейный порядок и что время, требующееся для сравнения двух имен, не зависит от мощности пространства имен S. Методы *цифрового поиска* используют тот факт, что имя или его представление можно всегда интерпретировать как последовательность символов над некоторым конечным алфавитом A (в крайнем случае, над двоичным алфавитом $\{0, 1\}$). Тогда естественным порядком на S является лексикографический порядок, индуцированный линейным порядком на A. Предполагается, что исход сравнения двух символов (не имен) получается за время, не зависящее от мощности пространства имен и от мощности n исследуемой таблицы T.

В качестве примера цифрового поиска рассмотрим построение таблицы для таких последовательностей десятичных цифр: 2448, 28, 31415, 64, 672, 67430, которые можно представить в виде *дерева цифрового поиска* (рис. 4.14). В дереве цифрового поиска последовательность цифр для каждого имени получается при прохождении некоторого пути от корня дерева к листу; идентичные префиксы (начальные подпоследовательности) различных имен объединяются до тех пор, пока это возможно.

Поиск имени z с цифрами $d_1, d_2, ..., d_l$ начинается в корне дерева и продолжается сравнением самой левой еще не сравнивавшейся цифры d_j с метками всех ветвей, выходящих из текущей вершины. Если d_j совпадает с меткой некоторой ветви, то продолжается движение по этой ветви до следующей вершины. Если

ни одна метка не совпадает с d_j, то поиск завершается безуспешно. Поиск завершается успешно, если все цифры имени z найдены.

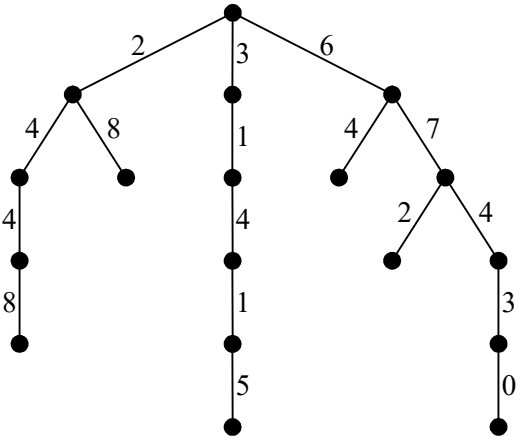


Рис. 4.14. Дерево цифрового поиска

Особенностью рассмотренного примера является то, что ни одно из имен в таблице не является префиксом другого имени. Для многих приложений это нехарактерно. Можно предложить два метода, позволяющих строить деревья цифрового поиска для имен, которые являются префиксами других имен в таблице. Первый метод заключается в том, что к каждой вершине дерева добавляется специальный разряд со значением «конец имени». Таким способом отмечается конец слова в этой вершине. Если в рассмотренном примере необходимо, чтобы последовательности 314 и 31415 были именами (314 является префиксом имени 31415), то в вершинах, к которым ведут ребра, помеченные цифрами 4 и 5, надо установить этот разряд равным 1, а во всех других вершинах вдоль пути 31415 надо установить этот разряд равным 0. Второй метод заключается в том, что имена модифицируются таким образом, чтобы ни одно имя не могло быть префиксом другого, например к концу каждого имени присоединить пробел или другой символ, не принадлежащий алфавиту. Очевидно, что второй метод может увеличить число вершин в дереве цифрового поиска по сравнению с первым методом.

Дерево цифрового поиска можно представить в памяти различными способами. Для эффективной реализации представление должно позволять одновременное сравнение символа в имени с метками всех ветвей, выходящих из вершины, поскольку поочередный просмотр всех выходящих из вершины ветвей приведет к существенному увеличению времени поиска и такой метод в большинстве случаев будет медленнее бинарного поиска. Один из способов превращения многократного просмотра ветвей в элементарную операцию, которая может быть выполнена за время, не зависящее от числа ветвей, заключается в следующем. Каждой вершине дерева сопоставляется вектор с компонентами для каждого символа алфавита независимо от того, имеется ли в таблице имя, использующее эту компоненту. Каждая цифра (символ) в подлежащем поиску имени используется как индекс или указатель соответствующей компоненты вектора, которая определяет следующий шаг поиска. Представленное таким способом дерево цифрового поиска называется *бором*. Бор, соответствующий дереву цифрового поиска (см. рис. 4.14), представлен на рисунке 4.15. Для экономии памяти последовательность вершин вдоль пути, где не происходит ветвление (например, путь 1415), сжата в один узел. Когда такой узел достигается, оставшиеся символы искомого имени должны сравниваться с символами, содержащимися в узле.

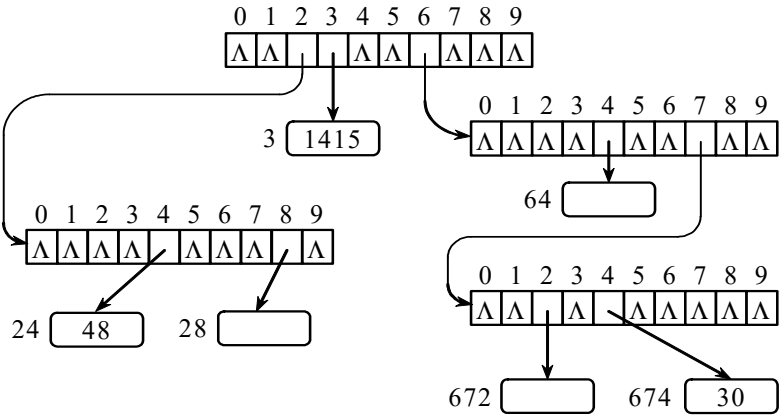


Рис. 4.15. Представление дерева цифрового поиска в виде бора

В качестве примера рассмотрим поиск по бору имени 2448. Цифра 2 используется как индекс в векторе, сопоставленном корню (остальные компоненты вектора не рассматриваются и время выполнения этой операции не зависит от размера алфавита). Найденный элемент вектора является указателем другого узла, в котором в качестве индекса используется цифра 4. Найденная компонента является указателем специального узла, содержащего суффикс 48 искомого имени. Соединение префикса 24, соответствующего пути в этот узел, и суффикса 48, который хранится явно в специальном узле, дает полное искомое имя 2448. Таким образом, имя 2448 найдено за два шага, так как исследовано минимальное число цифр, которые могут выделить это имя просмотром слева направо из всех остальных имен в таблице. При поиске имени 746 пустой указатель для седьмой компоненты вектора, сопоставленного корню, определяет поиск как безуспешный, т. е. за один шаг дается ответ, что этого имени нет в таблице.

Рассмотренный пример наглядно показывает два характерных свойства боров: они допускают быстрый поиск, особенно в случае безуспешных поисков, и имеют тенденцию к излишнему расходу памяти. Причиной неэффективного использования памяти является то, что схема размещения поля для каждого символа алфавита в каждом узле приспособлена для представления всего пространства имен, а не содержимого конкретной таблицы, т. е. бор обычно содержит место для многих имен, не принадлежащих таблице. В общем случае излишний объем памяти возникает в узлах вблизи нижней части бора, а узлы около корня имеют тенденцию к полноте. Поэтому на практике с целью экономии памяти часто используется комбинированный вариант, заключающийся в объединении методов поиска по борам для отдельных префиксов имен с другими методами поиска для суффиксов.

Поскольку методы цифрового поиска основаны на сравнении между символами в представлении имен, а на каждый символ представления требуется только по одному такому сравнению, время поиска пропорционально средней длине имен. При алфавите из c символов средняя длина n разных имен не может быть меньше, чем $\log_c n$, т. е. цифровой поиск может выполняться за время, логарифмическое относительно размера таб-

лицы, хотя и не гарантирует этого в худшем случае. Коэффициент пропорциональности зависит от многих факторов, но на практике в большинстве случаев цифровой поиск обычно выполняется быстрее бинарного, так как многопутевое разветвление от одного символа (для алфавита с c символами это обычно c-путевое разветвление) в основном не длиннее, чем двухпутевое разветвление при сравнении двух имен в бинарном поиске.

Цифровой поиск отличается от других методов также тем, что для данного множества имен существует только одно дерево цифрового поиска, т. е. не возникает проблемы построения оптимального дерева. Различные множества имен приводят к борам разного вида.

Операции включения и исключения в бору прямые, поэтому можно считать, что цифровой поиск является достаточно подходящим методом для динамических таблиц.

4.4. Хеширование

Методы *хеширования*, известные также как методы *рассеянной памяти*, методы *вычисления адреса*, основаны на том, что поиск ключа z в таблице $T = \{x_1, x_2, \ldots, x_n\}$ начинается с вычисления адреса по имени z. Суть хеширования заключается в следующем. Предполагается, что имеется память (*хеш-таблица*), состоящая из m ячеек M_j, $0 \le j \le m - 1$, при этом каждая ячейка может содержать одно имя. При заданном значении j обращение к ячейке M_j производится за постоянное время, не зависящее от размера m хеш-таблицы. Имеется функция $h : S \to A$, называемая *хеш-функцией*, которая равномерно отображает пространство имен S в пространство *хеш-адресов* $A = \{0, 1, \ldots, m - 1\}$, т. е. для любого имени $x \in S$ хеш-функция $h(x)$ принимает целочисленное значение из интервала $0, \ldots, m - 1$, которое является адресом ячейки хеш-таблицы. Число $h(x)$ называется *хеш-значением* (или *собственным адресом*) имени x. При анализе алгоритмов предполагается, что функция h может быть вычислена за время, не зависящее от мощности пространства имен, от размера m хеш-таблицы и от размера n исследуемой таблицы T.

Основным достоинством методов хеширования является то, что соответствующее им среднее время поиска не зависит от размера таблицы, если вся таблица умещается в оперативную

память, т. е. асимптотически и часто также на практике они являются самыми быстрыми методами поиска.

В качестве недостатков методов хеширования можно отметить следующие:

а) табличный порядок имен обычно не связан с их естественным порядком;

б) худший случай может оказаться хуже, чем при последовательном поиске;

в) сложность динамического расширения таблиц, поскольку расширение может приводить к потере памяти, если таблица слишком велика, или к малой производительности, если таблица слишком мала.

4.4.1. Варианты хеширования

Идеи хеширования проще рассмотреть для различных вариантов таблиц и размеров пространства имен.

Вариант 1. Особенностью данного варианта является то, что пространство имен очень мало. Рассмотрим, например, пространство имен $S = \{A, B, \ldots, Z\}$ (односимвольные имена в латинском алфавите), хеш-таблицу $M = \{M_0, M_1, \ldots, M_{25}\}$ из 26 ячеек с пространством адресов $A = \{0, 1, \ldots, 25\}$ и хеш-функцию $h : S \to A$, которая отображает символы в адреса так, чтобы сохранялся естественный порядок имен в пространстве S, т. е. $h(A) = 0,\ h(B) = 1,\ \ldots,\ h(Z) = 25$. Таблица $T = \{x_1, x_2, \ldots, x_n\} \subseteq S$ хранится в памяти следующим образом: в ячейках $M_{h(x_i)}$ размещаются имена x_i, а в ячейках, не содержащих имена (пустых ячейках), – специальный символ, не принадлежащий S (например, «–»):

$$S = \{A, B, \ldots, Z\} \qquad T = \{B, C, E, Z\}$$

A	0	1	2	3	4	$\cdots$	24	25
M	–	B	C	–	E	$\cdots$	–	Z

Четыре табличные операции выполняются простыми и быстрыми алгоритмами:

поиск z: **if** $M_{h(z)} = z$
then return $(h(z))$ // найдено: $h(z)$ указывает на z
else return (-1) // не найдено
включение z: $M_{h(z)} \leftarrow z$
исключение z: $M_{h(z)} \leftarrow$ "–"
распечатка: **for** $i \leftarrow 0$ **to** $m-1$ **do if** $M_i \neq$ "–" **then** write(M_i)

Необходимо обратить внимание на то, что в алгоритме поиска отсутствует явный цикл. Важно, чтобы любой цикл, присутствующий в вычислении хеш-функции h, не зависел от размера таблицы.

Рассмотренный вариант является идеальным случаем хеширования, когда размер хеш-таблицы оказался равным размеру пространства имен. На практике это выполняется редко, обычно память слишком мала, чтобы вместить все пространство имен.

Вариант 2. Особенностью данного варианта является то, что пространство имен велико (не умещается в память); таблица статическая. Поскольку таблица статическая, хеш-функцию можно выбрать после того, как станет известно содержимое таблицы. В этом случае можно найти легко вычислимую функцию, которая взаимно однозначно отображает множество хранящихся в таблице имен в пространство адресов (даже с сохранением естественного порядка имен в таблице). Особенно легко найти такую функцию, если хеш-таблица содержит больше ячеек, чем имен.

Таким образом, данный вариант сводится к первому варианту, алгоритмы поиска и распечатки полностью совпадают с соответствующими алгоритмами из первого варианта хеширования.

Вариант 3. Особенностью данного варианта является то, что пространство имен велико (не умещается в память); таблица динамическая. В этом случае содержимое таблицы заранее не известно. Поэтому хеш-функция должна быть построена независимо от этого на основании некоторой другой информации о таблице. Например, может быть известно, что число имен, которые необходимо хранить в таблице, не превышает некоторой границы; что некоторые имена встречаются с большей вероятностью, чем другие и т. п. На основании такой информации можно выбрать размер хеш-таблицы m и построить хеш-функцию h. Окончательным критерием выбора m и h является их эффективность на практике.

Для примера рассмотрим пространство имен S всех двоичных цепочек фиксированной длины 8, при этом предположим, что известен максимальный размер таблицы – шесть имен. Допустим, что выбрана хеш-таблица из восьми ячеек $M_0, M_1, \ldots, M_7$ и простая хеш-функция, которая отображает цепочку из трех самых правых разрядов, интерпретируемых как двоичное целое. Эта функция легко вычисляется и рассеивает пространство имен равномерно по пространству адресов.

Пусть таблица на некотором этапе (после выполнения ряда операций включения и исключения) состоит из пяти имен

$$\begin{aligned} x_1 &= 00001001 \qquad h(x_1) = (001)_2 = 1, \\ x_2 &= 01000011 \qquad h(x_2) = (011)_2 = 3, \\ x_3 &= 01100110 \qquad h(x_3) = (110)_2 = 6, \\ x_4 &= 10110000 \qquad h(x_4) = (000)_2 = 0, \\ x_5 &= 11101011 \qquad h(x_5) = (011)_2 = 3. \end{aligned}$$

Нетрудно заметить, что h отображает x_2 и x_5 на один и тот же адрес. Такая ситуация называется *коллизией*, или *конфликтом*. Какое-то из этих имен должно храниться в другой ячейке, адрес которой не совпадает с собственным адресом имени, определяемым функцией h. Выбор другой ячейки, если ячейка с собственным адресом занята, должен осуществляться на основании определенных правил, которые устанавливаются выбранным *методом разрешения коллизий*. В рассматриваемом примере для разрешения коллизии определим следующее правило: если ячейка с собственным адресом имени занята, то имя помещается в первую свободную ячейку, следующую за его собственным адресом. Таким образом, имя x_5 с собственным адресом $h(x_5) = 3$ в соответствии с выбранным методом разрешения коллизий помещается в ячейку M_4 (рис. 4.16)

j	0	1	2	3	4	5	6	7
M_j	x_4	x_1	–	x_2	x_5	–	x_3	–

Рис. 4.16. Размещение имени x_5 при коллизии с именем x_2

Поскольку возникает задача распознавания состояния ячейки (занята или свободна), свободные (пустые) ячейки должны отличаться от занятых. Это можно решить либо добавлением дополнительного разряда к каждой ячейке, либо, если это возмож-

но, использованием специального значения, не принадлежащего пространству имен, для обозначения пустой ячейки.

При возникновении коллизий поиск становится более сложным и обычно включает цикл, который может выполняться до m раз в худшем случае. При выбранном выше способе разрешения коллизий поиск требует последовательного просмотра памяти, при этом просмотр должен начинаться с собственного адреса искомого имени z и переходить от ячейки M_{m-1} к ячейке M_0 циклически. Поиск завершается безуспешно, если встречается пустая ячейка или когда просмотрена вся память. Чтобы избежать явной проверки последнего случая, необходимо в памяти всегда иметь по крайней мере одну пустую ячейку. Тогда операцию поиска ключа z в хеш-таблице можно представить алгоритмом 4.9.

$i \leftarrow h(z)$
while M_i не пуста **do**
  if $z = M_i$
    then **return**(i) // найдено: i указывает на z
    else $i \leftarrow (i+1)$ **mod** m
return(-1) // не найдено

Алгоритм 4.9. Поиск z в хеш-таблице

$i \leftarrow h(z)$
while M_i не пуста **do**
  if $z = M_i$
    then // z уже в таблице
    else $i \leftarrow (i+1)$ **mod** m
// z нет в таблице; включить его, если это возможно
if $n = m - 1$
  then // таблица заполнена; z включать нельзя
  else
    // таблица не заполнена; включить z
    $M_i \leftarrow z$
    пометить M_i "занято"
    $n \leftarrow n + 1$

Алгоритм 4.10. Включение z в хеш-таблицу

Включение имени z в хеш-таблицу представлено алгоритмом 4.10. Во время включения должна выполняться явная проверка того, что в хеш-таблице существует более одной пу-

стой ячейки. В алгоритме это осуществляется сравнением числа n уже входящих в таблицу имен с размером хеш-таблицы m. Если выбранный размер памяти m (размер хеш-таблицы) по каким-либо причинам не удовлетворяет, то расширить таблицу для методов хеширования не так просто, как в других методах поиска, поскольку хеш-функция явно зависит от размера хеш-таблицы. Расширение памяти требует нового хеширования, т. е. отыскания новой хеш-функции и перемещения всех ранее размещенных имен в соответствии с новой хеш-функцией.

Очевидный способ исключения имени из ячейки M_i, когда M_i помечается как пустая, является неправильным для большинства схем хеширования. Например, пусть исключается имя x_2 и ячейка M_3 помечается как пустая (см. рис. 4.16). Тогда если необходимо найти x_5, то последовательный поиск, начинающийся с собственного адреса имени $h(x_5) = 3$, находит, что M_3 пуста, и ошибочно заключает, что имени x_5 в таблице нет, хотя оно хранилось в ячейке M_4, из-за возникновения коллизии с x_2. Пока ячейка M_3 была занята, имя x_5 было достижимо, но когда M_3 стала пустой, x_5 стало недостижимым для алгоритма поиска. Если переместить x_5 из M_4 в его собственный адрес M_3, то это не дает гарантии, что пустая ячейка M_4 не сделает недоступным еще какое-либо имя. Просмотр всей последовательности имен (до первой пустой ячейки), анализ их собственных адресов и, если необходимо, перемещение требуют широкого поиска и проверки с существенным снижением эффективности работы. Поэтому исключение должно осуществляться по-другому.

Для реализации исключения необходимо иметь три состояния ячейки памяти: «занято», «пусто» и «исключено». Тогда исключение имени выполняется пометкой его ячейки как «исключено». Алгоритм поиска (алгоритм 4.9) для правильной его работы необходимо модифицировать так, чтобы ячейки с пометкой «исключено» игнорировались во время поиска. Алгоритм включения (алгоритм 4.10) нужно модифицировать так, чтобы новое имя включалось в пустую или «исключенную» ячейку в зависимости от того, какая из них встретится первой; в любом случае использованная ячейка помечается как занятая. Другими словами, «исключенная» ячейка ведет себя как пустая относительно включения и как заполненная относительно поиска. Данный подход при всей его привлекательности и гибкости имеет суще-

ственный недостаток. Поскольку «исключенная» ячейка ведет себя как заполненная ячейка относительно поиска, то это может привести к тому, что безуспешный поиск, который завершается, когда встречается первая пустая ячейка, становится очень неэффективным, если последовательность включений и исключений оставляет все меньше и меньше ячеек с пометкой «пусто», даже если общее число имен в таблице остается постоянным. В худшем случае при безуспешном поиске будет просматриваться вся хеш-таблица.

4.4.2. Хеш-функции

В идеальном варианте хеш-функция $h : S \to A$ должна быть легко вычислимой и отображать пространство имен в пространство адресов так, чтобы равномерно рассеять имена по памяти. Если второе требование предполагает, что каждый адрес $i \in A$ является образом примерно одинакового числа имен $x \in S$, то приемлемы простые хеш-функции, например функции, в которых подцепочки имени интерпретируются как двоичные целые. Однако на практике, поскольку вероятность появления различных имен из S в таблице различна, лучше позаботиться о том, чтобы равномерно рассеивалось по памяти содержимое таблицы, а не все пространство имен. Поэтому хеш-функция должна быть построена так, чтобы равномерно рассеять по памяти те подмножества множества S, которые могут встретиться в качестве содержимого таблицы. К сожалению, эти вероятные подмножества редко можно охарактеризовать точно. Следовательно, построение хеш-функций больше опирается на здравый смысл и интуицию, чем на какие-то формализованные методы.

Следует избегать хеш-функций, которые плохо работают на некоторых распространенных множествах имен. В частности, необходимо избегать хеш-функций, которые склонны отображать скученные имена (например, $x1$, $x2$, $x3$ или $a1$, $b1$, $c1$ и т. п.) в скученные адреса, поскольку они могут вызвать чрезмерное число коллизий.

Для обеспечения быстрого вычисления хеш-адресов большинство хеш-функций близко к примитивным операциям, допустимым для ЭВМ. Поэтому они лучше описываются на уровне операций над двоичными наборами в представлениях имен и ад-

ресов. В связи с этим будем полагать, что имена и адреса представлены двоичными наборами длины l_{name} и l_{addr} соответственно. Часто l_{name} меньше или равно длине машинного слова, и во всех представляющих интерес случаях $l_{name} > l_{addr}$, поскольку в случае $l_{name} \le l_{addr}$ каждому имени можно сопоставить свой адрес.

Простейшие хеш-функции выбирают некоторое специальное подмножество из l_{addr} разрядов из строки с l_{name} разрядами и используют их для формирования адреса. Существенным недостатком таких хеш-функций является то, что они имеют тенденцию порождать чрезмерное количество коллизий, так как все имена, представления которых совпадают по подмножеству разрядов, выбранному хеш-функцией, отображаются в один адрес.

Большинство используемых на практике хеш-функций основано на другом подходе, связанном с требованием, чтобы каждый из l_{addr} разрядов адреса зависел от всех l_{name} разрядов имени. Наиболее эффективный путь достижения этой зависимости состоит в том, чтобы производить арифметические операции, результаты которых зависят от всех разрядов имени, и затем выделить l_{addr} разрядов из этого результата. В качестве примера можно привести некоторые методы построения хеш-функций.

Метод мультипликативного хеширования. Метод свободен от недостатков, присущих методу квадрата, поскольку его основой является умножение на константу, что сохраняет равномерность распределения имен в памяти. При выборе константы следует помнить, что арифметические действия выполняются в некоторой дискретной конечной системе счисления, свойства которой (например, основание системы счисления и длину слова) необходимо учитывать, чтобы избежать вырожденных случаев, которые приводят к большому числу коллизий. Основная опасность мультипликативного хеширования относится к таблицам, которые содержат подмножества имен с представлениями, образующими арифметические прогрессии. Мультипликативные хеш-функции отображают такие прогрессии имен в арифметические прогрессии адресов, например, с разностью i. Если константа или размер памяти m выбраны произвольно, то i может оказаться делителем m и эти подмножества будут использовать только $1/i$ часть объема памяти. Следствием этого является порождение чрезмерного количества коллизий.

Метод деления с остатком. Этот метод использует хеш-функцию типа

$h(x)$ = (представление x, рассматриваемое как целое) **mod** m,

где m – размер хеш-таблицы, **mod** – операция получения остатка от деления целых чисел. Здесь решающим является выбор величины m. Например, если m четно, то $h(x)$ четно тогда и только тогда, когда представление x будет четным целым. Эта неравномерность может оказать существенное влияние на эффективность хеширования. Если m – степень основания системы счисления, применяемой в ЭВМ, то $h(x)$ будет зависеть только от самых правых символов имени x, что является еще одной опасностью неравномерности. Вообще к проблемам может приводить любой собственный делитель m. Поэтому рекомендуется, чтобы m было простым, далеко отстоящим от степеней числа 2, что гарантирует еще и то, что каждая цифра адреса зависит от всех символов имени.

Рассмотренных методов построения хеш-функций достаточно, чтобы обратить внимание на то, что любая разумная хеш-функция будет удовлетворительно работать на большинстве множеств имен, но в то же время для любой хеш-функции существуют множества имен, на которых она работает плохо. Это связано с тем, что хеш-функция делит пространство имен на группы имен с одинаковыми хеш-значениями. Если k – мощность пространства имен (в общем случае пространство имен может быть бесконечным), m – размер хеш-таблицы, то среднее число имен в таких группах будет k/m. Поэтому вполне возможна ситуация, когда большинство имен в таблице будет иметь один и тот же хеш-адрес (такая ситуация называется *первичным скучиванием*) и, как следствие, большое число коллизий.

4.4.3. Разрешение коллизий

Пока хеш-таблица не слишком заполнена, коллизии появляются редко и производительность схемы хеширования определяется прежде всего временем, требующимся для вычисления хеш-функции. По мере заполнения хеш-таблицы доступ к именам требует все большего времени из-за коллизий. Поэтому когда хеш-таблица используется в большой степени, эффективность схемы хеширования определяется схемой разрешения коллизий

и выбор схемы разрешения коллизий обычно важнее выбора хеш-функции.

Схема разрешения коллизий каждому имени x сопоставляет последовательность адресов $\alpha_0, \alpha_1, \alpha_2, \ldots$, где $\alpha_0 = h(x)$ является собственным адресом имени x. Алгоритм включения проверяет ячейки до тех пор, пока не найдет пустую. Для гарантии того, что пустая ячейка встречается, если она существует, каждый адрес i, $0 \le i \le m - 1$, должен появляться в последовательности точно один раз. По существу, эту последовательность можно представить двумя способами, соответствующими последовательному и связному распределению списков.

Открытая адресация. В рассмотренном третьем варианте хеширования использована простейшая форма открытой адресации, называемая *линейным опробованием*, которое порождает последовательность адресов $\alpha_i = (h(x) + i)$ **mod** m. Простота полученной таким путем последовательности не компенсирует существенный недостаток, который делает ее обычно неудовлетворительной. Этим недостатком является *вторичное скучивание*. Оно возникает, когда имена с различными хеш-значениями имеют одинаковые (или почти одинаковые) последовательности адресов $\alpha_1, \alpha_2, \ldots$ Когда возникает первичное скучивание, линейное опробование порождает последовательность занятых следующих одна за другой ячеек и все имена, попадающие в любые из этих ячеек в результате хеширования, порождают последовательность адресов, которая пробегает эти занятые ячейки.

Вторичного скучивания можно избежать, используя линейное опробование с приращением $\Delta(x)$, которое является функцией от x. Это приращение позволяет получить последовательность адресов $\alpha_i = (h(x) + i\,\Delta(x))$ **mod** m, которая будет опробовать каждую ячейку, если $\Delta(x)$ и m взаимно просты. Поскольку $\Delta(x)$, по существу, является другой хеш-функцией, этот метод называется *двойным хешированием*. Двойное хеширование является наилучшим методом разрешения коллизий для открытой адресации.

Метод цепочек. Метод цепочек – это способ построения последовательности указателей из собственного адреса $h(x)$ в ячейку, где x размещается окончательно. С помощью дополнительного расхода памяти для хранения указателей такой способ позволяет избежать проблемы вторичного скучивания во время поис-

ка, но не во время включения. Для иллюстрации рассмотрим пример, представленный на рис. 4.17. Пусть имена x и y образуют коллизию: x размещается по своему собственному адресу $\alpha_0 = h(x)$ и y размещается по следующему адресу α_1 в последовательности разрешения коллизий для α_0 (рис. 4.17*а*). Предположим, что имя z нельзя записать по его собственному адресу $h(z) = \beta_0$ и что, следуя по его последовательности разрешения коллизий $\beta_1, \beta_2, \ldots$, путь проходит через α_0 и α_1: например, $\beta_1 = \alpha_0$ и $\beta_2 = \alpha_1$. Для отыскания пустой ячейки во время включения просматриваются α_0 и α_1 (на рис. 4.17*б* последовательность просмотра ячеек показана штриховыми стрелками), но как только z разместится в β_3, ячейка β_3 привязывается непосредственно к $\beta_0 = h(z)$ (рис. 4.17*б*). Поэтому при поиске z производится меньше проб, чем при его включении.

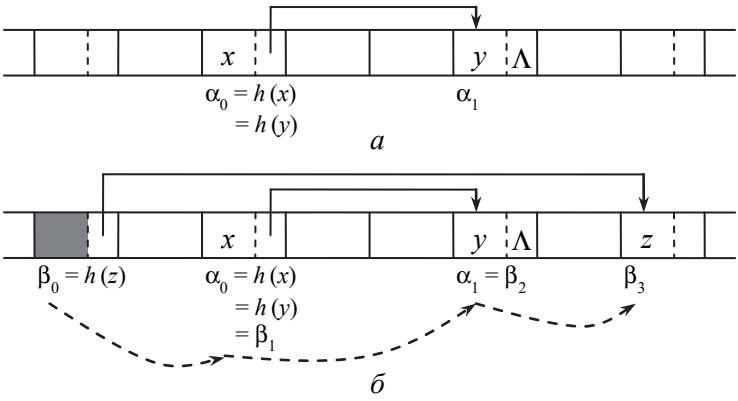


Рис. 4.17. Разрешение коллизий методом цепочек:
а – размещение имен x и y; *б* – добавление имени z.

Рассмотренные ранее методы разрешения коллизий используют ячейки только из хеш-таблицы. Метод хеширования, в котором для хранения имен используется ограниченное пространство хеш-таблицы, называется *закрытым*, или *прямым*, *хешированием*. Если такие требования не установлены, образующие коллизии имена можно помещать в отдельной дополнительной области памяти. Такое хеширование называется *открытым*, или *внешним*, и эквивалентно комбинации разных методов поиска:

хеширование используется на первом шаге, а другие методы используются для поиска среди имен с одинаковыми хеш-значениями. На рисунке 4.18 показана простейшая организация данных для открытого хеширования, когда в хеш-таблице хранятся не сами имена, а указатели на связные списки имен, образующих коллизии. Для повышения эффективности поиска множество имен с одинаковыми хеш-значениями можно упорядочить и использовать для его представления более гибкие и эффективные структуры данных.

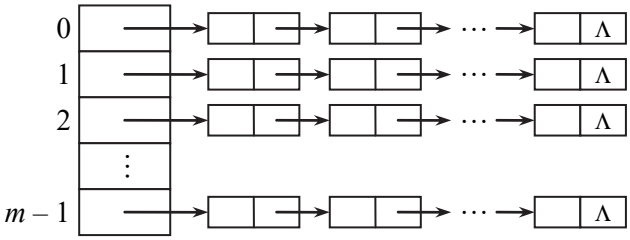


Рис. 4.18. Простейшая организация открытого хеширования

Что касается ожидаемой эффективности методов хеширования, то можно отметить следующее. Если предположить, что время доступа к любой ячейке памяти постоянно и что хеш-функция равномерно отображает пространство имен в пространство адресов, а также что содержимое таблицы есть беспристрастная выборка из пространства имен, ожидаемая эффективность метода вычисления адреса не зависит от числа имен n в таблице. Она зависит в основном от *коэффициента заполнения* хеш-таблицы $\lambda = n/m$. В экстремальном случае, когда память используется полностью ($\lambda = 1$), очевидно, что быстрого времени доступа достигнуть невозможно. Поэтому важно рассмотреть эффективность метода, когда хеш-таблица почти заполнена.

Для иллюстрации можно привести результаты приближенного анализа двойного хеширования. Математическое ожидание $U(\lambda)$ числа проб, необходимых для безуспешного поиска ключа z в хеш-таблице с коэффициентом заполнения λ, можно определить как $U(\lambda) = 1/(1 - \lambda)$, а математическое ожидание $S(\lambda)$ числа проб, необходимых для успешного поиска, как

$$S(\lambda) = \frac{1}{\lambda} \ln \frac{1}{1-\lambda}.$$

Функции $U(\lambda)$ и $S(\lambda)$ стремятся к 1 при $\lambda \to 0$, указывая, что при поиске в почти пустой таблице достаточно одной пробы. Функции $U(\lambda)$ и $S(\lambda)$ неограниченно растут при $\lambda \to 1$, что является следствием интерпретации λ как непрерывной величины, что равнозначно предположению о бесконечной памяти. Следующая таблица показывает значения функций $U(\lambda)$ и $S(\lambda)$ для различных значений коэффициента заполнения λ [21].

λ	0,5	0,75	0,9	0,95	0,99
$U(\lambda)$	2,0	4,0	10,0	20,0	100,0
$S(\lambda)$	1,39	1,85	2,56	3,15	4,65

Таким образом, если адреса, порожденные методом разрешения коллизий, независимы и равномерно распределены на пространстве адресов, как обычно бывает при двойном хешировании, то среднее число требуемых проб мало даже для такого большого коэффициента заполнения, как 90%.

4.5. Внешний поиск

Все обсуждавшиеся в предыдущих разделах методы поиска предполагают, что таблица полностью умещается в оперативной памяти. Однако для большинства реальных задач обработки данных таблицы могут быть настолько большими, что они не умещаются в оперативной памяти и доступ к ним возможен только по частям. Такие большие таблицы хранятся во внешней памяти в виде файлов и для них необходимы специальные методы *внешнего поиска*. Очевидно, что для последовательных файлов может быть использован только последовательный поиск, модифицированный так, чтобы учитывать специфику организации данных в виде последовательного файла.

Для внешней памяти с возможностью прямого доступа используют более эффективные методы хранения данных, которые позволяют существенно ускорить поиск и другие операции. Типичным представителем такой памяти является диск. Как известно, диск разбивается на дорожки, а каждая дорожка делится на сектора. Файловая система отводит место для записи файлов кластерами. Каждый кластер содержит определенное количество секторов. Поскольку время доступа (поиск сектора) может быть относительно большим, обычно записывают или считывают сектор целиком. Часто обработка данных из прочитанного сектора

занимает меньше времени, чем поиск нужного сектора. Таким образом, следует минимизировать число обращений к диску.

Одной из наиболее эффективных структур данных для представления больших таблиц во внешней памяти является сбалансированное m-арное дерево, которое является обобщением деревьев бинарного поиска.

Сбалансированное сильно ветвящееся дерево (или *Б-дерево*) порядка m есть корневое расширенное m-арное дерево, характеризующееся следующими свойствами:

1) корень либо является листом (пустое Б-дерево), либо имеет от 2 до m сыновей;

2) каждая внутренняя вершина, кроме корня, имеет от $\lceil m/2 \rceil$ до m сыновей. В общем случае нижняя граница числа сыновей определяется как $\lceil \alpha m \rceil$, $0 < \alpha < 1$. Чаще используется случай $\alpha = 1/2$, но возможны и другие значения α, которые задаются едиными для всего дерева;

3) все листья расположены на одном уровне, т. е. все пути от корня до любого листа имеют одинаковую длину, равную высоте дерева;

4) каждая внутренняя вершина с t сыновьями содержит упорядоченный набор из $t - 1$ имен и t указателей на сыновей, т. е. ее можно представить в виде $(p_1, x_1, p_2, x_2, p_3, \ldots, p_{t-1}, x_{t-1}, p_t)$, где p_i – указатель на i-го сына $(1 \le i \le t)$, x_i – имя $(1 \le i \le t - 1)$, причем $x_1 < x_2 < \ldots < x_{t-1}$. Имена x_i являются границами, разделяющими имена в сыновьях. Поэтому все имена в вершине, на которую указывает p_i, больше x_{i-1} и меньше x_i.

На рисунке 4.19 представлен пример сбалансированного максимально заполненного Б-дерева порядка 5, хранящего 24 имени (внешние вершины не показаны).

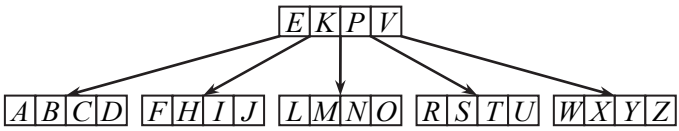


Рис. 4.19. Максимально заполненное Б-дерево порядка 5

Оценим сверху высоту h Б-дерева, хранящего n имен, т. е. на уровне h расположено $n + 1$ листьев (внешних вершин). Наибольшую высоту имеют Б-деревья с минимально заполнен-

ными вершинами, т. е. корень имеет двух сыновей, а все другие внутренние вершины – по $k = \lceil m/2 \rceil$ сыновей. Таким образом, на уровнях 1, 2, 3, …, $h-1$ имеется 2, $2k$, $2k^2$, …, $2k^{h-2}$ внутренних вершин соответственно. Поскольку в корне хранится одно имя, а во всех остальных внутренних вершинах по $k-1$ имен, получается неравенство

$$n \geq 1 + (k-1)\sum_{i=1}^{h-1} 2k^{i-1} = 1 + 2(k-1)\left(\frac{k^{h-1}-1}{k-1}\right) = 2k^{h-1} - 1,$$

из которого следует верхняя оценка высоты Б-дерева

$$h \leq 1 + \log_k \frac{n+1}{2} = 1 + \log_{\lceil m/2 \rceil} \frac{n+1}{2}.$$

Минимальную высоту имеют Б-деревья с максимально заполненными внутренними вершинами, т. е. все внутренние вершины имеют по m сыновей, причем в каждой вершине хранится по $m-1$ имен. Поэтому нижняя оценка высоты Б-дерева будет $h \geq \log_m(n+1)$.

Важным достоинством Б-деревьев является относительная простота выполнения поиска, включения и исключения.

Поиск. Поиск имени z в Б-дереве осуществляется путем сравнения z с именами $x_1, x_2, \ldots, x_{t-1}$, хранящимися в корне (поскольку эти имена упорядочены, можно применить бинарный поиск). Если z совпадает с каким-либо из этих имен, поиск завершается успешно. В противном случае, если $x_{i-1} < z < x_i$, процесс поиска продолжается рекурсивно в вершине, на которую указывает указатель p_i, расположенный между именами x_{i-1} и x_i. Безуспешный поиск завершается в листе (внешней вершине).

Включение. Включение имени z в Б-дерево выполняется в том случае, если поиск z завершается безуспешно. Безуспешный поиск завершается в листе, в котором могло бы находиться z. В отличие от деревьев бинарного поиска, Б-деревьям запрещено расти в листьях – их вынуждают расти в корне. Поэтому новое имя продвигается вверх во внутреннюю вершину. Если эта вершина не полностью заполнена, то имя z вставляется в соответствующую позицию и процесс включения завершается. Если же в вершине нет места для записи z (вершина максимально заполнена), то она расщепляется на две вершины, а среднее имя продвигается вверх в вершину-отца. Процесс продвижения вверх продолжается до тех пор, пока не встретится не полностью за-

полненная вершина. В худшем случае процесс завершится расщеплением корня, что приведет к образованию нового корня и увеличению высоты Б-дерева на единицу.

В качестве примера рассмотрим включение имени *G* в Б-дерево на рис. 4.19. Безуспешный поиск определяет лист, в котором могло бы находиться имя *G* (рис. 4.20*а*). Имя *G* продвигается в вершину с именами *F*, *H*, *I*, *J*. Поскольку она полностью заполнена, пять имен *F*, *G*, *H*, *I*, *J* разбиваются на две группы и среднее имя *H* продвигается в вершину-отца, чтобы оно служило разделителем между двумя группами (рис. 4.20*б*). Вершина-отец также полностью заполнена, поэтому она расщепляется, среднее имя *K* продвигается вверх и становится новым корнем Б-дерева (рис. 4.20*в*).

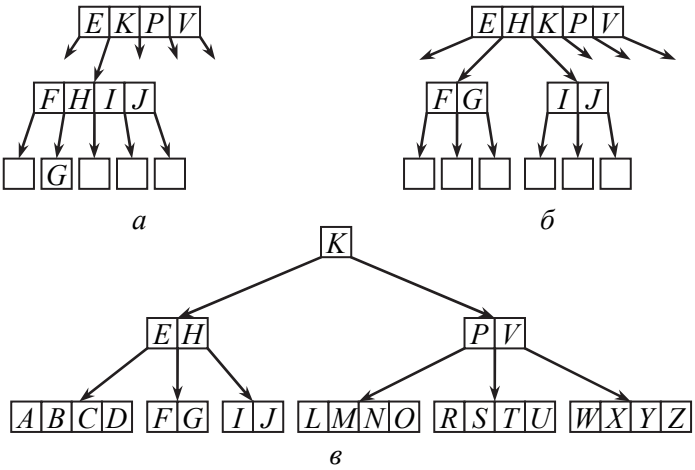


Рис. 4.20. Включение имени *G* в Б-дерево:
а – определение листа, где могло бы находиться имя *G*;
б – разбиение пяти имен *F*, *G*, *H*, *I*, *J* на две группы;
в – разбиение пяти имен *E*, *H*, *K*, *P*, *V* на две группы.

Исключение. Если исключаемое имя *z* служит разделителем между группами имен, то его прямо удалить из вершины нельзя. Его необходимо заменить именем *y*, либо непосредственно предшествующим, либо непосредственно следующим в естественном порядке. Очевидно, что эти имена хранятся во внутренних вершинах, находящихся на самом нижнем уровне Б-дерева. Если вершина, в которой хранится имя *y*, не является

минимально заполненной, то процесс завершается удалением y из этой вершины. В противном случае исключение y приводит к тому, что вершина становится недостаточно заполненной, ей не хватает одного имени. Поэтому в эту вершину перемещается разделитель из вершины-отца, а на его место – имя из одного из смежных с недозаполненной вершиной братьев. Если брат имеет достаточный запас имен, никаких проблем не возникает и процесс завершается. Если же брат является минимально заполненным, т. е. содержит ровно $\lceil m/2 \rceil - 1$ имен, то этот минимально заполненный брат, недозаполненная вершина с $\lceil m/2 \rceil - 2$ именами и их разделитель в вершине-отце объединяются в одну вершину с $m - 1$ или $m - 2$ именами. Если в результате этих действий вершина-отец стала недозаполненной, процесс повторяется на следующем более высоком уровне. В худшем случае процесс исключения завершается в корне, тогда высота Б-дерева может уменьшиться на один уровень.

В качестве примера рассмотрим исключение имени E из Б-дерева на рисунке 4.20*в*. Поскольку E является разделителем между множествами $\{A, B, C, D\}$ и $\{F, G\}$, можно на его место переместить непосредственно предшествующее имя D, которое будет разделителем между $\{A, B, C\}$ и $\{F, G\}$, и завершить процесс. Рассмотрим более сложную ситуацию – переместим на место E непосредственно следующее имя F, которое служит разделителем между $\{A, B, C, D\}$ и $\{G\}$ (рис. 4.21*а*). В вершине с единственным именем G не хватает одного имени, чтобы быть минимально заполненной. Левый брат этой вершины имеет достаточный запас имен, поэтому можно добавить разделитель F в вершину с именем G и переместить вверх D в качестве нового разделителя (рис. 4.21*б*). Если нет смежного брата с достаточным запасом имен, то должен существовать смежный брат с минимальным заполнением. Для иллюстрации рассмотрим правого брата вершины с именем G, который является минимально заполненным (рис. 4.21*а*). Выполняется объединение имен G, H, I, J в одну вершину (рис. 4.21*в*). Теперь вершина с именем F стала недозаполненной. Объединение этой вершины с минимально заполненным правым братом с именами P и V и разделителем K приводит к Б-дереву, показанному на рисунке 4.21*г*.

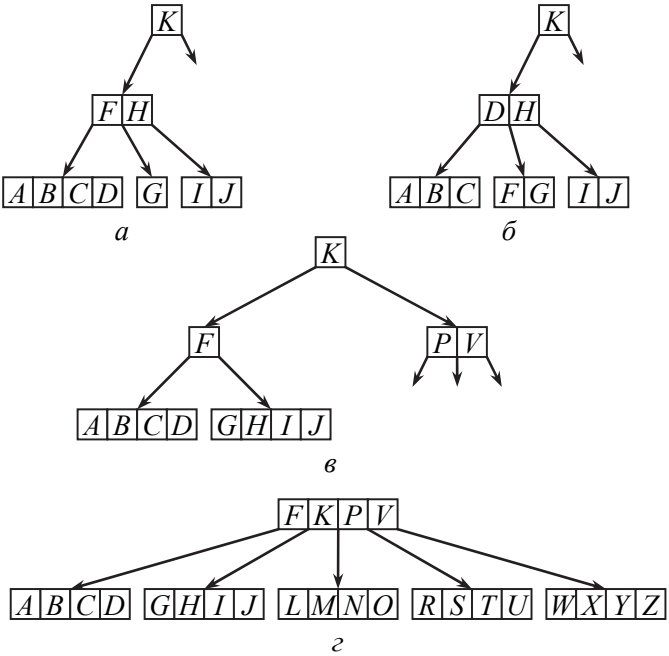


Рис. 4.21. Исключение имени *E* из Б-дерева:
а – замена исключаемого имени *E* на непосредственно следующее имя *F*;
б – перемещение *D* вверх вместо *F*, а *F* добавляется в вершину с *G*;
в – объединение имен *G*, *H*, *I*, *J* в одну вершину;
г – объединение имен *F*, *K*, *P*, *V* в одну вершину.

При реализации Б-деревьев следует учитывать ряд факторов. Прежде всего, чтобы применить бинарный поиск имени в вершине, следует хранить в каждой внутренней вершине количество содержащихся в ней имен (поскольку эта величина не фиксирована) и поддерживать эту информацию при выполнении операций включения и исключения. Кроме того, чтобы легче выполнять расщепление вершины точно пополам (при включении нового имени), лучше использовать нечетное значение m.

Как известно, Б-деревья порядка m являются эффективной структурой данных для хранения больших таблиц во внешней памяти с возможностью прямого доступа. Время работы с файлами в основном определяется количеством операций обращения к внешней памяти (чтение, запись). Поэтому стремятся к чтению или записи возможно большей информации за одно обращение к

внешней памяти. Таким образом, степень ветвления (величина m) Б-дерева обычно определяется размером сектора или некоторого блока (кластера), состоящего из определенного числа секторов. Типичная степень ветвления Б-деревьев (величина m) находится между 50 и 2000.

Очевидно, что число обращений к внешней памяти пропорционально высоте Б-дерева, т. е. $O(\log_m n)$. Увеличение степени ветвления резко сокращает высоту дерева и число обращений к внешней памяти.

Обычно файл представляет собой последовательность записей, каждая из которых может представлять собой достаточно сложную структуру, состоящую из одного и того же множества полей. Ключ, идентифицирующий запись, часто составляет небольшую часть записи. Если такие записи хранить целиком в вершине Б-дерева, то это приведет к слишком малой степени ветвления. Поэтому в вершинах дерева лучше хранить только ключи, а сами записи – в листьях (при этом в листьях не хранятся указатели на сыновей, поскольку их нет). В этом случае поиск требуемой записи предполагает прохождение пути от корня до листа, в котором содержится запись.

Частным случаем m-арных Б-деревьев, когда $m = 3$, являются так называемые *2-деревья* [2]. В 2–3-деревьях каждая внутренняя вершина может иметь двух или трех сыновей (соответственно может хранить одно или два имени). Этот вид сбалансированных деревьев является альтернативой АВЛ-деревьям и красно-черным деревьям для представления динамических таблиц.

Упражнения

1. Разработать алгоритм однородного бинарного поиска с вычислением значений δ.

2. Разработать алгоритм однородного бинарного поиска с помощью дополнительной таблицы с вычисленными заранее значениями δ.

***3**. Разработать алгоритм поиска Фибоначчи для таблиц произвольного размера $n \geq 1$.

4. Разработать алгоритм интерполяционного поиска.

5. Разработать алгоритм исключения узла из дерева бинарного поиска.

6. Разработать алгоритмы операций вращения и двойного вращения для двух вариантов узлового представления бинарных деревьев:

а) обычный вариант (узел состоит из полей *info*, *left* и *right*);

б) вариант, когда в структуру узла добавлено поле *father*, указывающее на отца данного узла.

***7**. Разработать алгоритмы включения и исключения с балансировкой для АВЛ-деревьев для двух вариантов узлового представления из упражнения 4.7, учитывая, что в структуре узла АВЛ-дерева имеется поле *balance* для хранения значения баланса данного узла.

***8**. Разработать алгоритмы включения и исключения с балансировкой для красно-черных деревьев для двух вариантов узлового представления из упражнения 7, учитывая, что в структуре узла RB-дерева имеется поле *color* для хранения цвета данного узла.

9. Построить алгоритмы включения и исключения для дерева цифрового поиска, представленного в виде бора.

10. Модифицировать алгоритмы поиска и включения в хеш-таблицу так, чтобы они правильно работали при наличии ячеек, помеченных состоянием «исключено».

***11**. Упорядоченные хеш-таблицы. В этом варианте используется цифровой или алфавитный порядок имен. Это ведет к более быстрому поиску ценой небольшой дополнительной работы во время включения. Предположим, что в хеш-таблице имеется m ячеек, n из которых заняты. Для удобства допустим, что все имена имеют строго положительное числовое значение. Таблица состоит из ячеек памяти $M_0, M_1, \ldots, M_{m-1}$. $M_j = 0$, если позиция пуста; в противном случае $M_j > 0$ – имя, хранящееся в ячейке памяти j. Пусть $h(x)$ – хеш-значение имени x и $\Delta(x)$ – приращение хеширования имени x, так что $\alpha_i = (h(x) + i\,\Delta(x))$ **mod** m, $0 \le h(x) < m$, $1 \le \Delta(x) < m$, и $\Delta(x)$ и m взаимно просты для всех x. В предположении, что в таблице не разрешается содержать более $m - 1$ имен, поиск в упорядоченной хеш-таблице осуществляется следующим образом:

$i \leftarrow h(x)$
while $M_i > x$ **do** $i \leftarrow (i + \Delta(x))$ **mod** m
if $M_i = x$
  then // найдено: i указывает на x
  else // не найдено

Разработать алгоритмы включения и исключения имен для упорядоченных хеш-таблиц. Объяснить, чем упорядоченная хеш-таблица отличается от обычной хеш-таблицы.

***12**. Разработать алгоритмы поиска, включения и исключения для Б-деревьев.

***13**. Разработать алгоритм копирования Б-дерева.

***14**. Разработать алгоритм объединения двух Б-деревьев. Для удобства в каждой вершине следует хранить высоту поддерева, корнем которого она является. Чтобы поддерживать эту информацию, соответствующим образом должны быть модифицированы алгоритмы включения и исключения.

***15**. Разработать алгоритмы поиска, включения, исключения и объединения для 2–3-деревьев.

Глава 5. СОРТИРОВКА

Сортировкой множества элементов называется расположение этих элементов по возрастанию или убыванию в соответствии с определенным отношением линейного порядка. Обычно рассматривается отношение естественного порядка, когда элементы располагаются по возрастанию. Формально задача сортировки определяется следующим образом. Дано конечное множество $T = \{x_1, x_2, \ldots, x_n\}$, которое будем называть *таблицей*. Требуется найти перестановку $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ этих n элементов, которая отобразит данную таблицу (множество T) в неубывающую последовательность $x_{\pi_1} \le x_{\pi_2} \le \ldots \le x_{\pi_n}$. Как правило, алгоритмы сортировки вырабатывают саму упорядоченную последовательность, а не упорядочивающую перестановку П.

В общем случае элементы таблицы могут иметь сложную структуру, но с каждым элементом ассоциирован некоторый *ключ* (*имя*), используемый для того, чтобы отличить один элемент от другого. Поскольку рассматривается прежде всего процесс сортировки ключей, идентифицирующих каждый элемент, а остальные компоненты не влияют на упорядочивающую функцию, будем считать, что элементами таблицы являются имена (ключи).

Алгоритм сортировки называется *устойчивым*, если он сохраняет исходный порядок равных имен. Данное свойство важно в тех случаях, когда элементы уже упорядочены по какому-то вторичному ключу и необходимо провести сортировку по первичному ключу (не зависящему от вторичного) так, чтобы внутри групп с одинаковыми первичными ключами сохранялся порядок, определяемый вторичным ключом. Если алгоритм сортировки не обладает свойством устойчивости, то эту задачу придется решать, сортируя элементы по составному ключу, являющемуся объединением первичного и вторичного ключей.

Для упрощения анализа алгоритмов сортировки будем считать, что никакие два имени не имеют одинаковых значений, т. е. любые $x_i, x_j \in T$ обладают тем свойством, что если $i \ne j$, то либо $x_i < x_j$, либо $x_i > x_j$. В этом случае сортировка должна найти перестановку П, отображающую исходную таблицу T в возрастающую последовательность $x_{\pi_1} < x_{\pi_2} < \ldots < x_{\pi_n}$. Ограничение $x_i \ne x_j$

при $i \neq j$ упрощает анализ алгоритмов сортировки без потери общности, поскольку и при наличии равных имен корректность идей и алгоритмов не нарушается.

Выделяют два вида сортировки: внутреннюю и внешнюю. *Внутренняя сортировка* решает задачу полной сортировки для случая достаточно малой таблицы, умещающейся непосредственно в оперативной памяти. *Внешняя сортировка* решает задачу полной сортировки для случая такой большой таблицы, не умещающейся в оперативной памяти, что доступ к ней организован по частям, расположенным на внешних запоминающих устройствах.

Время работы многих алгоритмов сортировки зависит от содержимого исходной таблицы, поэтому для анализа эффективности ряда алгоритмов необходим некоторый критерий, позволяющий оценить степень неотсортированности входной таблицы. В качестве такого критерия удобно использовать понятие инверсии перестановки.

Пусть $X = (x_1, x_2, \ldots, x_n)$ есть некоторая перестановка. Пара (x_i, x_j) называется *инверсией* перестановки X, если $i < j$, а $x_i > x_j$. *Вектором инверсий* перестановки X называется последовательность целых чисел $D = (d_1, d_2, \ldots, d_n)$, где d_j – число элементов x_i таких, что пара (x_i, x_j) является инверсией. Другими словами, элемент d_j – это число элементов, больших x_j и стоящих слева от него в перестановке X, т. е. $0 \le d_j < j$. Вектор инверсий однозначно определяет перестановку. Например, вектором инверсий перестановки

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 27 & 07 & 54 & 42 & 14 & 10 & 25 & 17 \end{pmatrix}$$

будет

$$\begin{array}{cccccccccc} j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ d_j & 0 & 1 & 0 & 1 & 3 & 4 & 3 & 4 \end{array}.$$

Предположим, что исходное множество имен представляется с помощью массива, хотя возможны и другие способы представления, например связный список.

Поскольку во многих алгоритмах сортировки используется операция обмена значений имен, для более компактной записи алгоритмов операцию обмена будем записывать в виде $x_i \leftrightarrow x_j$ (значения имен x_i и x_j меняются местами).

5.1. Оценки эффективности алгоритмов сортировки

Предварительно рассмотрим задачу сортировки с теоретической точки зрения, чтобы получить некоторое представление об ожидаемой эффективности. Для многих алгоритмов сортировки хорошей мерой производимой работы является число сравнений имен. Эта характеристика не всегда является определяющей для эффективности алгоритмов сортировки, поскольку существуют алгоритмы, в которых число обменов преобладает над числом сравнений имен; существуют также алгоритмы сортировки, в которых отсутствует прямое сравнение имен. Тем не менее для большинства алгоритмов сортировки указанная характеристика является определяющей. Поэтому представляет интерес минимальное число сравнений, необходимых для сортировки n имен.

Выполнение любого алгоритма сортировки, основанного на сравнении имен, можно представить в виде расширенного бинарного дерева решений, в котором каждая внутренняя вершина соответствует сравнению имен x_i и x_j (обозначим через $x_i : x_j$), а каждая внешняя вершина (лист) – исходу алгоритма. Два сына внутренней вершины показывают два возможных исхода сравнения. Бинарное дерево решений для сортировки трех имен представлено на рисунке 5.1.

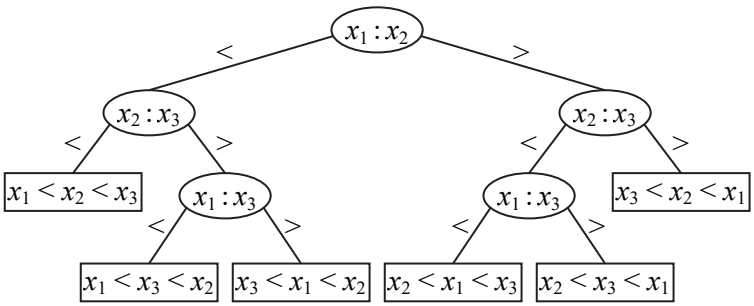


Рис. 5.1. Бинарное дерево решений для сортировки трех имен

В любом таком дереве решений каждая перестановка определяет единственный путь от корня к листу. Поскольку алгоритмы сортировки должны правильно работать на всех $n!$ перестановках имен, листья, соответствующие разным перестановкам,

должны быть разными. Следовательно, в дереве решений для сортировки n имен должно быть по крайней мере $n!$ листьев. Очевидно, что высота дерева решений равна числу сравнений, требующихся алгоритму в наихудшем случае. Обозначим через $C(n)$ минимальное число сравнений, выполняемых любым алгоритмом сортировки в худшем случае. Поскольку бинарное дерево высоты h может иметь не более 2^h листьев, должно выполняться условие $2^{C(n)} \geq n!$. Используя формулу Стирлинга (формулу приближенного вычисления факториалов), получаем

$$C(n) \geq \log n! \approx n \log n.$$

Таким образом, любой алгоритм сортировки, основанный на сравнении имен, в наихудшем случае потребует не меньше $n \log n$ сравнений.

Минимальное среднее число сравнений имен $C_{ave}(n)$, выполняемых алгоритмом, который правильно сортирует все $n!$ перестановок при условии, что все они равновероятны, также легко определяется с помощью бинарного дерева решений. Известно, что длина внешних путей дерева решений равна сумме всех расстояний от корня до листьев. Деление длины внешних путей на число листьев дает среднюю длину внешних путей, т. е. среднее число сравнений для соответствующего алгоритма. Известно, что минимальную длину внешних путей (соответственно и минимальное среднее число сравнений) имеют полностью сбалансированные деревья. Для расширенного бинарного дерева с N внешними вершинами минимальная длина внешних путей равна $N \log N + O(N)$. Положив $N = n!$ и разделив на число листьев $n!$, получим соотношение

$$C_{ave}(n) \geq \log n! \approx n \log n.$$

Таким образом, любой алгоритм сортировки, основанный на сравнении имен, потребует в среднем не меньше $n \log n$ сравнений имен.

Полученные результаты для $C(n)$ и $C_{ave}(n)$ дают некоторый эталон для сравнения эффективности многих алгоритмов сортировки, основанных на сравнении имен.

5.2. Сортировка вставками

Сортировка вставками основана на том, что имена таблицы просматриваются по одному и на каждом этапе просмотра новое имя вставляется в подходящее место среди ранее отсортированных имен. Рассмотрим два алгоритма из этого класса.

5.2.1. Простая сортировка вставками

Алгоритм 5.1 представляет собой *простую сортировку вставками*. Процесс сортировки проходит через этапы $j = 2, 3, \ldots, n$; на этапе j имя x_j вставляется на свое правильное место среди $x_1, x_2, \ldots, x_{j-1}$ уже отсортированных имен. При вставке имя x_j временно размещается в переменной t и просматриваются имена $x_{j-1}, x_{j-2}, \ldots, x_1$; они сравниваются с t и сдвигаются вправо, если они больше t. Фиктивное имя-сторож $x_0 = -\infty$ служит для остановки просмотра слева. На рисунке 5.2 показан процесс работы этого алгоритма на примере таблицы из $n = 8$ имен.

$$
\begin{array}{l}
x_0 \leftarrow -\infty \\
\textbf{for}\ \ j \leftarrow 2\ \ \textbf{to}\ \ n\ \ \textbf{do}
\begin{cases}
i \leftarrow j-1 \\
t \leftarrow x_j \\
\textbf{while}\ \ t < x_i\ \ \textbf{do}
\begin{cases}
x_{i+1} \leftarrow x_i \\
i \leftarrow i-1
\end{cases} \\
x_{i+1} \leftarrow t
\end{cases}
\end{array}
$$

Алгоритм 5.1. Простая сортировка вставками

Эффективность этого алгоритма, как и большинства алгоритмов сортировки, зависит от числа сравнений имен и числа пересылок данных, осуществляемых в трех случаях: худшем, среднем (в предположении, что все $n!$ перестановок равновероятны) и лучшем. Ключом к анализу алгоритма является число проверок условия «$t < x_i$» в цикле **while**, поскольку, если это число известно, легко выводится число пересылок имен «$t \leftarrow x_j$», «$x_{i+1} \leftarrow x_i$» и «$x_{i+1} \leftarrow t$».

Число сравнений имен $C(n)$ (число проверок условия «$t < x_i$») полностью зависит от значений элементов вектора инверсий $D = (d_1, d_2, \ldots, d_n)$ для сортируемой таблицы, где d_j – число имен, больших x_j и расположенных слева от него.

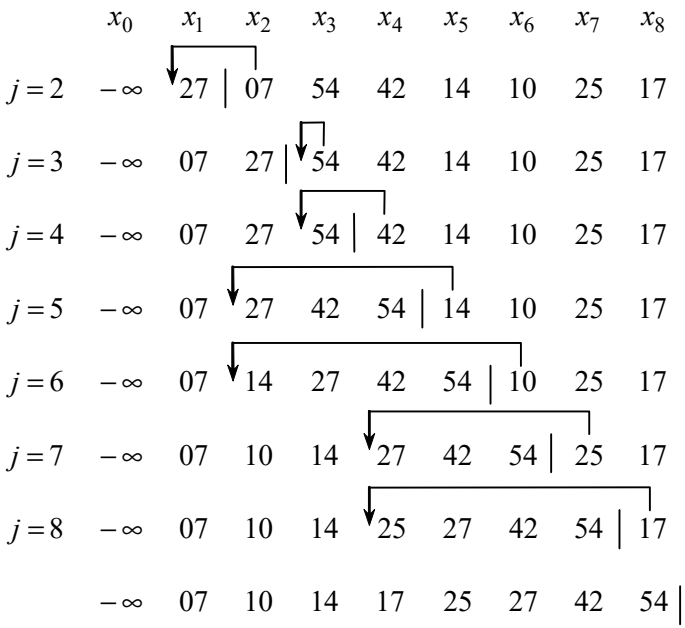


Рис. 5.2. Процесс работы алгоритма простой сортировки вставками

На j-м этапе выполняется $C_j = d_j + 1$ сравнений имен, а всего выполняется $n-1$ этапов предопределенного цикла **for** ($j = 2, \ldots, n$). Следовательно,

$$C(n) = \sum_{j=2}^{n}(d_j + 1) = n - 1 + \sum_{j=2}^{n} d_j .$$

Так как $0 \le d_j \le j-1$ и величины d_j независимы, сумма элементов вектора инверсий $\sum_{j=2}^{n} d_j$ имеет минимальное значение, равное нулю, в случае уже упорядоченной исходной таблицы максимальное значение, равное

$$0 + 1 + 2 + \ldots + n - 1 = \sum_{j=1}^{n-1} j = \frac{1}{2} n(n-1),$$

имеет в случае, когда элементы исходной таблицы первоначально расположены в обратном порядке. Среднее значение $\sum_{j=2}^{n} d_j$ по

всем перестановкам равно $\frac{1}{2}\sum_{j=1}^{n-1} j = \frac{1}{4}n(n-1)$. Таким образом, общее число сравнений имен, осуществляемых алгоритмом простой сортировки вставками, составляет:

$C_{\min}(n) = n - 1 = O(n)$ в лучшем случае;

$C_{\text{ave}}(n) = \frac{1}{4}(n-1)(n+4) = O(n^2)$ в среднем;

$C_{\max}(n) = \frac{1}{2}(n-1)(n+2) = O(n^2)$ в худшем случае.

Что касается числа пересылок $M(n)$ (присваиваний имен), то очевидно, что на j-м этапе выполняется $C_j + 1$ пересылок ($C_j - 1$ пересылок в цикле **while** и две пересылки «$t \leftarrow x_j$» и «$x_{i+1} \leftarrow t$»). Следовательно,

$$M(n) = \sum_{j=2}^{n}(C_j + 1) = \sum_{j=2}^{n}(d_j + 2) = 2(n-1) + \sum_{j=2}^{n} d_j = (n-1) + C(n).$$

Таким образом, общее число пересылок имен составляет

$M_{\min}(n) = 2(n - 1) = O(n)$ в лучшем случае,

$M_{\text{ave}}(n) = \frac{1}{4}(n-1)(n+8) = O(n^2)$ в среднем,

$M_{\max}(n) = \frac{1}{2}(n-1)(n+4) = O(n^2)$ в худшем случае.

Проведенный анализ показывает, что алгоритм простой сортировки вставками имеет асимптотическую временную сложность $O(n^2)$ в среднем и худшем случаях. При этом наилучшим для алгоритма является случай, когда исходная таблица уже упорядочена, а наихудшим – когда имена в исходной таблице первоначально расположены в обратном порядке.

Алгоритм сортировки вставками можно достаточно просто усовершенствовать. Учитывая, что вставка j-го имени осуществляется среди $x_1, x_2, \ldots, x_{j-1}$ уже отсортированных имен, можно использовать бинарный поиск для определения места вставки (*сортировка бинарными вставками*). В этом случае уменьшается число сравнений имен до $O(n \log n)$, но сама вставка требует $O(n^2)$ пересылок имен, т. е. алгоритму все равно потребуется порядка n^2 операций. Следует отметить, что сортировка бинарными вставками уже отсортированной таблицы потребует больше времени, чем использование простой сортировки вставками.

Другой путь усовершенствования заключается в использовании связного списка для представления таблицы (*сортировка вставками в связный список*). Этим достигается более эффективная вставка. Однако в связном списке невозможен бинарный поиск для определения места вставки, можно использовать только последовательный поиск. Для такого алгоритма сокращается число пересылок, но число сравнений все равно остается порядка n^2, т. е. алгоритму все равно потребуется $O(n^2)$ операций.

5.2.2. Сортировка Шелла

Основная причина неэффективности алгоритма простой сортировки вставками заключается в том, что в каждый момент времени имена сдвигаются только на одну позицию. Поэтому для улучшения необходим некоторый механизм, с помощью которого имена могли бы перемещаться на большие расстояния. Такой механизм используется в *сортировке с убывающим шагом*, названной в честь ее изобретателя *сортировкой Шелла*.

Такая сортировка заключается в следующем. Задается последовательность шагов, представляющая собой последовательность положительных целых чисел $h_1 > h_2 > \ldots > h_t = 1$. Необходимо обратить внимание на то, что последнее значение этой последовательности обязательно должно быть равно единице. Сначала отдельно группируются и сортируются (обычно при помощи метода простых вставок) имена, отстоящие друг от друга на расстоянии h_1 (h_1-сортировка). Затем имена перегруппировываются в соответствии с шагом h_2 и снова сортируются (h_2-сортировка). Процесс h-сортировок продолжается до тех пор, пока не выполнится h_t-сортировка. На последнем проходе (при h_t-сортировке) все имена таблицы образуют одну группу, поскольку $h_t = 1$. Детали сортировки Шелла показаны в алгоритме 5.2. В этом алгоритме для упрощения условия окончания поиска для определения места вставки использованы имена-сторожа. Очевидно, что каждая h-сортировка нуждается в своем собственном имени-стороже, поэтому приходится расширять исходную таблицу не на один компонент x_0 (как это делается в алгоритме простой сортировки вставками), а на $h_1 - 1$ компонентов.

for $i \leftarrow -(h_1 - 1)$ **to** 0 **do** $x_i \leftarrow -\infty$ // установка сторожей

for $l \leftarrow 1$ **to** t **do**
- // h_l – сортировка
- $k \leftarrow h_l$
- **for** $j \leftarrow k+1$ **to** n **do**
  - $i \leftarrow j - k$
  - $t \leftarrow x_j$
  - **while** $t < x_i$ **do**
    - $x_{i+k} \leftarrow x_i$
    - $i \leftarrow i - k$
  - $x_{i+k} \leftarrow t$

Алгоритм 5.2. Сортировка Шелла

Процесс работы алгоритма сортировки Шелла для последовательности шагов (4, 2, 1) представлен на рисунке 5.3 (имена-сторожа не указаны). На первом проходе отдельно сортируются четыре группы по два имени в каждой группе: (x_1, x_5), (x_2, x_6), (x_3, x_7), (x_4, x_8). На втором проходе сортируются две группы по четыре имени: (x_1, x_3, x_5, x_7), (x_2, x_4, x_6, x_8). Процесс завершается третьим проходом, во время которого сортируются все восемь имен.

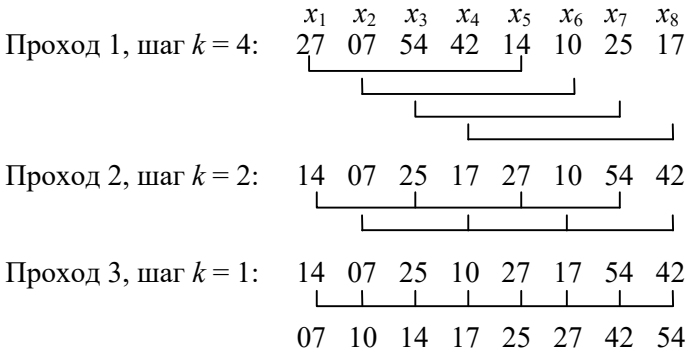


Рис. 5.3. Процесс работы алгоритма сортировки Шелла

Хорошая эффективность сортировки Шелла объясняется тем, что на первых проходах, когда шаги являются большими, сортируемые группы имен малы, поэтому сортировка вставками работает достаточно быстро. Сортировка таких подгрупп приво-

дит к тому, что вся таблица становится ближе к отсортированному виду, т. е. существенно уменьшается число инверсий. Поэтому на последующих проходах, хотя и используются шаги с меньшими значениями и, следовательно, сортируются большие группы имен, тем не менее сортировка вставками остается достаточно эффективной.

Анализ эффективности сортировки Шелла математически сложен. Приведем только некоторые результаты. Существенным фактором, влияющим на эффективность, является выбор последовательности шагов. Одно из требований состоит в том, что элементы в последовательности шагов должны быть взаимно простыми числами. Последовательность шагов рекомендуется выбирать следующим образом [8]: $h_t = 1$, $h_{l-1} = 3h_l + 1$ и $t = \lfloor \log_3 n \rfloor - 1$, т. е. последовательность (в обратном порядке) 1, 4, 13, 40, …, тогда сортировка потребует времени $O(n(\log n)^2)$, либо $h_t = 1$, $h_{l-1} = 2h_l + 1$ и $t = \lfloor \log_2 n \rfloor - 1$, т. е. последовательность (в обратном порядке) 1, 3, 7, 15, …, с временной сложностью $O(n^{1,2})$.

5.3. Обменная сортировка

Обменная сортировка некоторым систематическим образом меняет местами пары имен, не отвечающие порядку, до тех пор, пока такие пары существуют. Рассмотрим два алгоритма обменных сортировок.

5.3.1. Пузырьковая сортировка

Пузырьковая сортировка (алгоритм 5.3) основана на просмотре пар смежных имен последовательно слева направо и перемене мест тех имен, которые не отвечают порядку. Переменная b используется для предотвращения избыточного просмотра имен в правой части таблицы, про которые известно, что они находятся на своих окончательных позициях. В начале цикла **while** значение переменной b равно наибольшему индексу t такому, что про имя x_t еще не известно, стоит ли оно на окончательной позиции. Процесс работы алгоритма и изменения значений элементов вектора инверсий после каждого прохода для таблицы из $n = 8$ имен представлены на рисунке 5.4.

$$
\begin{array}{l}
b \leftarrow n \\
\textbf{while}\ \ b \neq 0\ \ \textbf{do}
\begin{cases}
t \leftarrow 0 \\
\textbf{for}\ \ j \leftarrow 1\ \ \textbf{to}\ \ b-1\ \ \textbf{do} \\
\quad \textbf{if}\ \ x_j > x_{j+1}\ \ \textbf{then}
\begin{cases}
x_j \leftrightarrow x_{j+1} \\
t \leftarrow j
\end{cases} \\
b \leftarrow t
\end{cases}
\end{array}
$$

Алгоритм 5.3. Пузырьковая сортировка

Проход	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
	27	07	54	42	14	10	25	17	0	1	0	1	3	4	3	4
1	07	27	42	14	10	25	17	54	0	0	0	2	3	2	3	0
2	07	27	14	10	25	17	42	54	0	0	1	2	1	2	0	0
3	07	14	10	25	17	27	42	54	0	0	1	0	1	0	0	0
4	07	10	14	17	25	27	42	54	0	0	0	0	0	0	0	0
5	07	10	14	17	25	27	42	54	0	0	0	0	0	0	0	0

Рис. 5.4. Процесс работы алгоритма пузырьковой сортировки

Эффективность рассматриваемого алгоритма зависит от трех факторов: числа проходов (числа выполнений тела цикла **while**), числа сравнений $x_j > x_{j+1}$ и числа обменов $x_j \leftrightarrow x_{j+1}$.

Очевидно, что число обменов $M(n)$ равно сумме элементов вектора инверсий. Поэтому

$M_{\min}(n) = 0$ в лучшем случае,

$M_{\text{ave}}(n) = \dfrac{1}{4} n(n-1) = O(n^2)$ в среднем,

$M_{\max}(n) = \dfrac{1}{2} n(n-1) = O(n^2)$ в худшем случае.

При сравнительном анализе различных алгоритмов необходимо иметь в виду, что для многих языков программирования операция обмена отсутствует. В таких случаях операция обмена

эквивалентна трем операциям пересылки (присваивания): $t \leftarrow x_j$, $x_j \leftarrow x_{j+1}$ и $x_{j+1} \leftarrow t$.

Анализ изменений вектора инверсий в процессе пузырьковой сортировки показывает, что каждый проход (исключая последний) уменьшает на единицу каждый ненулевой элемент вектора инверсий и сдвигает вектор на одну позицию влево. Таким образом, число проходов $A(n)$ равно наибольшему элементу вектора инверсий плюс единица, т. е.

$$A(n) = 1 + \max(d_1, d_2, \ldots, d_n).$$

Следовательно, в лучшем случае имеется всего один проход, в худшем – n проходов. Для определения среднего числа проходов необходимо найти математическое ожидание $\sum_{k=1}^{n} k P_k$, где P_k – вероятность того, что потребуется ровно k проходов, т. е. вероятность того, что наибольшим элементом вектора инверсий является $k-1$. Число векторов инверсий, содержащих только такие элементы, значения которых меньше k $(1 \le k \le n)$, равно $k^{n-k}k!$. Тогда число векторов инверсий с наибольшим элементом, равным $k-1$, будет равно $k^{n-k}k! - (k-1)^{n-k+1}(k-1)!$. Следовательно, вероятность того, что потребуется ровно k проходов, равна

$$P_k = \frac{1}{n!}(k^{n-k}k! - (k-1)^{n-k+1}(k-1)!) .$$

Теперь можно вычислить среднее значение

$$\sum_{k=1}^{n} kP_k = n + 1 - \sum_{k=1}^{n} \frac{k^{n-k}k!}{n!} = n + 1 - F(n) ,$$

где $F(n)$ – функция, асимптотическое поведение которой описывается формулой

$$F(n) = \sqrt{\pi n/2} - 2/3 - O(1/\sqrt{n}) .$$

Таким образом, общее число проходов цикла **while**, осуществляемых алгоритмом пузырьковой сортировки, составляет

$A_{\min}(n) = 1$ в лучшем случае,

$A_{\text{ave}}(n) = n - \sqrt{\pi n/2} + 5/3 + O(1/\sqrt{n}) = O(n)$ в среднем,

$A_{\max}(n) = n$ в худшем случае.

Общее число сравнений $C(n)$ исследовать несколько сложнее. Поэтому приведем только результаты анализа:

$C_{\min}(n) = n - 1 = O(n)$ в лучшем случае;

$C_{\text{ave}}(n) = \frac{1}{2}(n^2 - n\ln n) + O(n) = O(n^2)$ в среднем;

$C_{\max}(n) = \frac{1}{2}n(n-1) = O(n^2)$ в худшем случае.

Проведенный анализ показывает, что алгоритм пузырьковой сортировки имеет асимптотическую временную сложность $O(n^2)$ в среднем и худшем случаях. При этом наилучшим для алгоритма является случай, когда исходная таблица уже упорядочена, а наихудшим – когда имена в исходной таблице первоначально расположены в обратном порядке.

Усовершенствованием пузырьковой сортировки является так называемая *шейкер-сортировка*, которая предполагает попеременные проходы в противоположных направлениях. Это позволяет несколько сократить число сравнений, но не число обменов. Как пузырьковая, так и шейкер-сортировка существенно уступают по эффективности простой сортировке вставками.

Другое усовершенствование основано на идее сортировки Шелла (сортировки с убывающим шагом), когда осуществляется обмен именами, расположенными не в соседних позициях, а на больших расстояниях друг от друга. Отличием от сортировки Шелла является то, что в качестве h-сортировки используется пузырьковая сортировка. Это позволяет улучшить асимптотические характеристики, но полученный алгоритм все равно будет существенно уступать по эффективности сортировке Шелла.

5.3.2. Быстрая сортировка

Как и в простой сортировке вставками, в пузырьковой сортировке основным источником неэффективности является то, что обмены дают слишком малый эффект, так как в каждый момент времени имена сдвигаются только на одну позицию. Такие алгоритмы всегда требуют порядка n^2 операций, как в среднем, так и в худшем случаях. В быстрой сортировке используется прием, приводящий к тому, что каждый обмен совершает больше работы.

Идея метода *быстрой сортировки* заключается в том, что в таблице выбирается одно из имен, которое используется для разделения таблицы на две подтаблицы, состоящие соответственно из имен меньших и больших выбранного. Разделение можно реализовать, одновременно просматривая таблицу слева направо и справа налево, меняя местами имена в неправильных частях таблицы. Имя, используемое для расщепления таблицы (*расщепляющее*, или *базовое*, имя), затем помещается между двумя подтаблицами. Полученные подтаблицы сортируются рекурсивно.

Рекурсивный алгоритм быстрой сортировки (алгоритм 5.4) сортирует таблицу $x_f, x_{f+1}, \ldots, x_l$, где x_f является базовым именем, используемым для разбиения таблицы на подтаблицы.

procedure *QUICKSORT* (f, l)
  // отсортировать $x_f, x_{f+1}, \ldots, x_l$
  if $f > l$ **then return**
  // разделить таблицу
  $i \leftarrow f+1$
  while $x_i < x_f$ **do** $i \leftarrow i+1$
  $j \leftarrow l$
  while $x_j > x_f$ **do** $j \leftarrow j-1$
  while $i < j$ **do**
    // в этот момент $i < j, x_i \geq x_f \geq x_j$
    $x_i \leftrightarrow x_j$
    $i \leftarrow i+1$
    while $x_i < x_f$ **do** $i \leftarrow i+1$
    $j \leftarrow j-1$
    while $x_j > x_f$ **do** $j \leftarrow j-1$
  $x_f \leftrightarrow x_j$
  // отсортировать подтаблицы рекурсивно
  QUICKSORT $(f, j-1)$
  QUICKSORT $(j+1, l)$
return

Алгоритм 5.4. Рекурсивный алгоритм быстрой сортировки

Алгоритм предполагает, что имя x_{l+1} определено и больше, чем $x_f, x_{f+1}, \ldots, x_l$, т. е. перед первым обращением к процедуре *QUICKSORT*$(1, n)$ необходимо установить $x_{n+1} \leftarrow \infty$ в качестве сторожа. В начале цикла **while** $i < j$ индексы i и j указывают соответственно на первое и последнее имена, о которых известно,

что они находятся не в тех частях таблицы, в которых требуется. Когда i и j встречаются, т. е. когда $i \geq j$, все имена находятся в соответствующих частях таблицы, а имя x_f помещается между двумя частями, меняясь местами с x_j. Процесс разбиения алгоритмом 5.4 таблицы на две подтаблицы показан на рисунке 5.5.

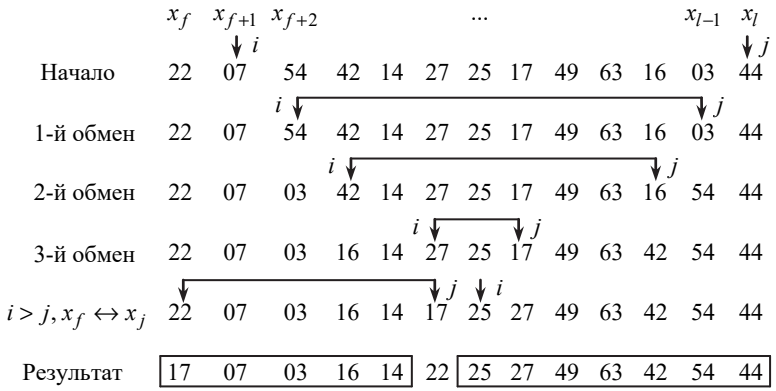


Рис. 5.5. Процесс разбиения таблицы на две подтаблицы

Определим общее число сравнений имен $x_i < x_f$ и $x_j > x_f$. В конце цикла **while** $i < j$ все имена $x_{f+1}, x_{f+2}, \ldots, x_l$ сравнивались с x_f по одному разу, исключая имена x_s и x_{s+1} (где просмотры встретились), которые сравнивались с x_f дважды. Следовательно, для таблицы из n имен ($n = l - f + 1$) к моменту разделения на подтаблицы выполняется $n + 1 = l - f + 2$ сравнений. Поскольку разбиение таблицы производится только в случае, если $f < l$, для тривиальных таблиц (вырожденных, когда $n = 0$ при $f > l$, или состоящих из одного имени, когда $n = 1$ при $f = l$) число сравнений равно нулю.

Пусть $C_{\text{ave}}(n)$ – среднее число сравнений имен для сортировки таблицы из n разных имен в предположении, что все $n!$ перестановок таблицы равновероятны. Для тривиальных таблиц $C_{\text{ave}}(0) = C_{\text{ave}}(1) = 0$. В общем случае имеем

$$C_{\text{ave}}(n) = n + 1 + \sum_{s=1}^{n} p_s (C_{\text{ave}}(s-1) + C_{\text{ave}}(n-s)), n \geq 2 .$$

Здесь p_s – вероятность того, что x_1 (расщепляющее имя) есть s-е наименьшее имя. Поскольку две подтаблицы, порожденные разбиением, случайны, т. е. все $(s-1)!$ перестановок имен в левой подтаблице равновероятны и все $(n-s)!$ перестановок имен в правой подтаблице равновероятны, $p_s = 1/n$. Таким образом,

$$C_{\text{ave}}(n) = n+1+\frac{1}{n}\sum_{s=1}^{n}(C_{\text{ave}}(s-1)+C_{\text{ave}}(n-s)),\ n \geq 2\,.$$

Поскольку эта сумма равна

$$C_{\text{ave}}(0)+C_{\text{ave}}(n-1)+C_{\text{ave}}(1)+C_{\text{ave}}(n-2)+\cdots+$$
$$+C_{\text{ave}}(n-2)+C_{\text{ave}}(1)+C_{\text{ave}}(n-1)+C_{\text{ave}}(0),$$

получаем

$$C_{\text{ave}}(n) = n+1+\frac{2}{n}\sum_{s=0}^{n-1}C_{\text{ave}}(s),\ n \geq 2.$$

Использование известных способов решения подобных рекуррентных соотношений (см. п. 1.4) дает

$$C_{\text{ave}}(n) = (\ln 4)\, n \log n + O(n) \approx 1.386\, n \log n,$$

т. е. среднее число сравнений имен равно $O(n \log n)$.

Очевидно, что худшим для алгоритма быстрой сортировки является случай, когда каждый раз для разбиения таблицы берется наибольшее или наименьшее имя. Тогда на каждом этапе одна из подтаблиц будет вырожденной ($n = 0$), а другая будет состоять из $n-1$ имен. Таким образом, если алгоритм применяется к уже отсортированным или отсортированным в обратном порядке таблицам, то производится

$$C_{\max}(n) = (n+1)+n+(n-1)+\cdots+3 = \frac{1}{2}n^2+\frac{3}{2}n-2 = O(n^2)$$

сравнений имен.

Аналогичный (но более сложный) анализ показывает, что среднее число $M_{\text{ave}}(n)$ обменов $x_i \leftrightarrow x_j$ и $x_f \leftrightarrow x_j$:

$$M_{\text{ave}}(n) = \frac{1}{6}n+\frac{2}{3}+\frac{2}{n}\sum_{s=0}^{n-1}M_{\text{ave}}(s) \approx 0.231\, n \log n\,,$$

т. е. среднее число обменов имен равно $O(n \log n)$. В худшем случае $M_{\max}(n) \leq n \log n$, т. е. $M_{\max}(n) = O(n \log n)$.

Таким образом, рекурсивный алгоритм быстрой сортировки требует $O(n \log n)$ операций в среднем, но $O(n^2)$ операций в худшем случае. Кроме того, поскольку рекурсия используется для

записи подтаблиц, рассматривающихся на более поздних этапах, в худших случаях (когда таблица уже отсортирована) глубина рекурсии может равняться n. Следовательно, для стека, реализующего рекурсию, необходима память, пропорциональная n. Для больших n такое требование становится неприемлемым.

Основным источником неэффективности быстрой сортировки является выбор расщепляющего имени. В алгоритме расщепляющим именем является первое имя таблицы (подтаблицы). Можно несколько улучшить быструю сортировку, если использовать для разделения таблицы случайно выбранное имя $x_{\mathrm{rand}(f, l)}$. Для этого достаточно в алгоритм добавить операцию $x_f \leftrightarrow x_{\mathrm{rand}(f, l)}$ непосредственно перед операцией $i \leftarrow f + 1$. Дополнительное время для выбора случайного целого числа несущественно, а случайный выбор является вполне приемлемой защитой от наихудшей ситуации. Более сильным улучшением является использование для разделения таблицы *медианы малой случайной выборки* из k (обычно k нечетное) имен или даже *медианы медиан*. Медианой является среднее по величине имя в множестве из k случайным образом выбранных из исходной таблицы имен. Например, если $k = 3$, то $C_{\mathrm{ave}}(n) \approx 1{,}188\, n \log n$.

Одним из наиболее эффективных алгоритмов внутренней сортировки является итерационный вариант быстрой сортировки (алгоритм 5.5), который свободен от некоторых недостатков рекурсивного алгоритма. В алгоритме стек S ведется явно; элементом стека является пара (f, l), показывающая, что нужно отсортировать имена $x_f, x_{f+1}, \ldots, x_l$. В стек помещается большая из двух подтаблиц и продолжается обработка меньшей подтаблицы. Это уменьшает глубину стека в худшем случае до $\lfloor \log (n + 1) / 3 \rfloor$, $n \geq 2$, т. е. для организации стека необходима память, пропорциональная $\log n$.

Учитывая то, что для небольших таблиц длины меньше некоторого m среднее число сравнений имен в быстрой сортировке больше среднего числа сравнений, например в простой сортировке вставками, для быстрой сортировки можно сделать значительное усовершенствование. Вместо того, чтобы использовать быструю сортировку для всех подтаблиц, можно применять ее только к подтаблицам длины не меньше m, а для подтаблиц длины меньше m воспользоваться простой сортировкой вставками.

```
S ⇐ (0, 0)
f ← 1
x_{n+1} ← ∞
l ← n
while f < l do
    x_f ↔ x_{rand(f, l)}
    i ← f + 1
    while x_i < x_f do i ← i + 1
    j ← l
    while x_j > x_f do j ← j − 1
    while i < j do
        // имеет место i < j, x_i ≥ x_f ≥ x_j}
        x_i ↔ x_j
        i ← i + 1
        while x_i < x_f do i ← i + 1
        j ← j − 1
        while x_j > x_f do j ← j − 1
    x_f ↔ x_j
    case
        j − 1 ≤ f and l ≤ j + 1:
            // обе подтаблицы
            // тривиальны
            (f, l) ⇐ S
        j − 1 ≤ f and l > j + 1:
            // нетривиальна только
            // правая подтаблица
            f ← j + 1
        j − 1 > f and l ≤ j + 1:
            // нетривиальна только
            // левая подтаблица
            l ← j − 1
        j − 1 > f and l > j + 1:
            // обе подтаблицы
            // нетривиальны,
            // поместить большую
            // в стек S
            if j − f > l − j
                then
                    // левая в S
                    S ⇐ (f, j − 1)
                    f ← j + 1
                else
                    // правая в S
                    S ⇐ (j + 1, l)
                    l ← j − 1
```

Алгоритм 5.5. Итерационный алгоритм быстрой сортировки

Интересным аналогом быстрой сортировки является так называемая *цифровая обменная сортировка*. В этом методе сортировки используется двоичное представление имен. Разделение таблицы осуществляется на основании самого старшего разряда

имен: если этот разряд равен единице, то имя пересылается в правую часть таблицы, если этот разряд равен нулю – в левую часть таблицы. На следующих этапах расщепления основываются на следующих разрядах имен и т. д. Таким образом, метод цифровой обменной сортировки позволяет исключить проблему выбора расщепляющего имени.

5.4. Сортировка выбором

Общая идея сортировки выбором состоит в том, что на каждом шаге $i = 1, 2, \ldots, n$ осуществляется поиск i-го наибольшего (наименьшего) имени, которое затем помещается на свою окончательную позицию. Рассмотрим два метода такой сортировки, которые отличаются способом нахождения i-го наибольшего (наименьшего) имени.

5.4.1. Простая сортировка выбором

Простая сортировка выбором представлена алгоритмом 5.6, в котором i-е наибольшее имя находится очевидным способом просмотром оставшихся $n - i + 1$ имен. Процесс работы алгоритма для таблицы из $n = 8$ имен показан на рисунке 5.6.

for $i \leftarrow n$ **downto** 2 **do**

$$\begin{cases} j \leftarrow 1 \\ \textbf{for } k \leftarrow 2 \textbf{ to } i \textbf{ do if } x_j < x_k \textbf{ then } j \leftarrow k \\ x_j \leftrightarrow x_i \end{cases}$$

Алгоритм 5.6. Простая сортировка выбором

Анализ этого алгоритма прост, поскольку имеются два предопределенных цикла. Очевидно, что время работы алгоритма не зависит от содержимого исходной таблицы. Число сравнений имен на i-м шаге равно $n - i$, а таких шагов выполняется $n - 1$, т. е. общее число сравнений имен

$$C(n) = (n-1) + (n-2) + \cdots + 1 = \frac{1}{2} n(n-1) = O(n^2).$$

Число обменов определяется внешним циклом **for** и также не зависит от входа, т. е. имеем $M(n) = n - 1 = O(n)$.

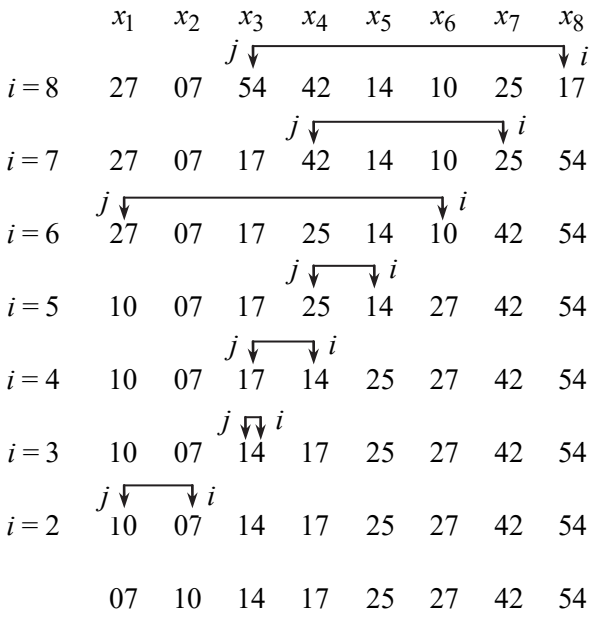


Рис. 5.6. Процесс работы алгоритма простой сортировки выбором

Таким образом, алгоритм простой сортировки выбором имеет временную сложность $O(n^2)$ независимо от содержимого исходной таблицы. Он по эффективности несколько уступает простой сортировке вставками, но имеет существенное преимущество перед пузырьковой сортировкой.

Можно довести число обменов до нуля в лучшем случае (когда исходная таблица уже отсортирована), если запретить обмен при $i = j$ (в примере на рисунке 5.6 такая ситуация возникает при $i = 3$) включением соответствующей проверки во внешний цикл **for** непосредственно перед операцией обмена. Однако для достаточно больших n это может привести к увеличению общего среднего времени работы алгоритма (если все $n!$ перестановок равновероятны), поскольку увеличивается длительность выполнения каждой итерации внешнего цикла **for** из-за дополнительной проверки условия «$i = j$».

5.4.2. Пирамидальная сортировка

Основным источником неэффективности простой сортировки выбором является поиск i-го наибольшего имени. В любом алгоритме нахождения максимума из n элементов, основанном на сравнении пар элементов, необходимо выполнить по крайней мере $n - 1$ сравнений. Это справедливо только для первого шага выбора. Более эффективный метод определения i-го наибольшего имени достигается тем, что при последующих выборах можно использовать информацию, полученную на предыдущих выборах. Такой подход к выбору i-го наибольшего имени используется в так называемом механизме турнира с выбыванием.

Механизм *турнира с выбыванием* заключается в следующем: сравниваются пары $x_1 : x_2$, $x_3 : x_4$, …, $x_{n-1} : x_n$, затем сравниваются «победители» (т. е. большие имена) этих сравнений и т. д. Пример такой процедуры для $n = 16$ показан на рисунке 5.7. Для определения наибольшего имени этот процесс потребует $n - 1$ сравнений имен. Но, определив наибольшее имя, мы будем иметь информацию о втором по величине имени: оно должно быть одним из тех, которые «потерпели поражение» от наибольшего имени. Следовательно, второе по величине имя теперь можно определить, заменяя наибольшее имя на $-\infty$ и вновь осуществляя сравнение вдоль пути от наибольшего имени к корню (с $-\infty$ сравнение не производится). Эта процедура для дерева на рис. 5.7 показана на рисунке 5.8. Выполняются сравнения 44:68, 63:68, 54:68, т. е. второе по величине имя 68 определяется за 3 сравнения (вместо 14 сравнений при простом выборе).

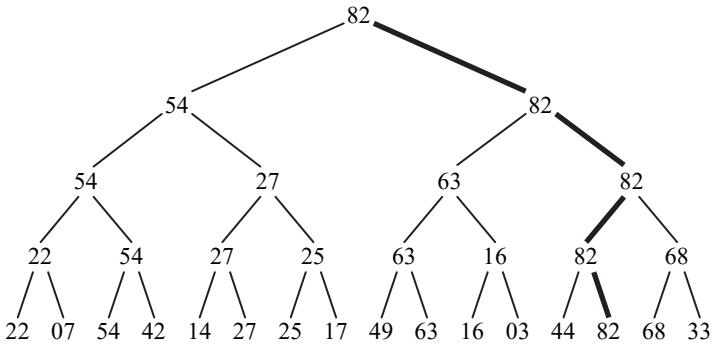


Рис. 5.7. Турнир с выбыванием для поиска наибольшего имени

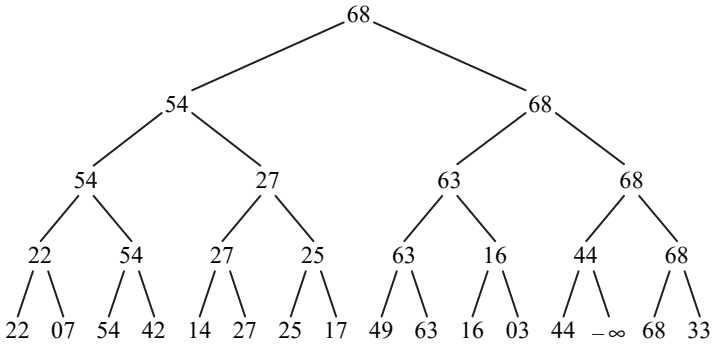


Рис. 5.8. Турнир с выбыванием для поиска второго наибольшего имени

Таким образом, поскольку дерево имеет высоту $\lceil \log n \rceil$, второе по величине имя можно найти за $\lceil \log n \rceil - 1$ сравнений вместо $n - 2$, используемых в простой сортировке выбором. Этот процесс можно продолжить. Найдя второе по величине имя, заменяем его на $-\infty$ и вновь выполняем $\lceil \log n \rceil - 1$ сравнений, чтобы найти следующее, и т. д. Очевидно, что в целом процесс использует не больше

$$n - 1 + (n-1)\left(\lceil \log n \rceil - 1\right) \approx n \lceil \log n \rceil$$

сравнений имен.

Идею турнира с выбыванием легко использовать при сортировке, если имена образуют пирамиду. *Пирамида* – это полностью сбалансированное бинарное дерево высоты h, в котором все листья находятся на расстоянии h или $h - 1$ от корня и все потомки вершины меньше его самого; кроме того, в нем все листья уровня h максимально смещены влево. Множество из 12 имен, организованных в виде пирамиды, показано на рисунке 5.9.

Удобно использовать линейное представление пирамиды, когда она хранится по уровням в одномерном массиве. Тогда сыновья имени из i-й позиции размещаются в позициях $2i$ (левый) и $2i + 1$ (правый). Таким образом, пирамида, представленная на рис. 5.9, принимает вид:

i:	1	2	3	4	5	6	7	8	9	10	11	12
x_i:	63	49	54	42	16	27	25	17	22	14	07	03

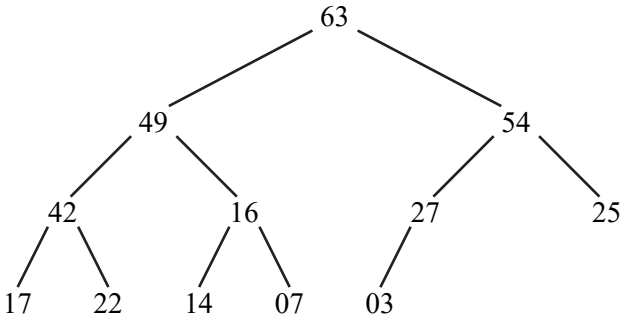


Рис. 5.9. Пирамида из 12 имен

В пирамиде наибольшее имя всегда должно находиться в корне, т. е. всегда в первой позиции массива, представляющего пирамиду. Обмен местами первого имени с n-м помещает наибольшее имя в его правильную позицию, но нарушает свойство пирамидальности в первых $n - 1$ именах. Если имеется возможность сначала построить пирамиду из исходной таблицы, а затем ее эффективно восстанавливать, то можно производить сортировку следующим образом:

Построить пирамиду из $x_1, x_2, \ldots, x_n$

for $i \leftarrow n$ **downto** 2 **do** $\begin{cases} x_1 \leftrightarrow x_i \\ \text{Восстановить пирамиду} \\ \quad \text{в } x_1, x_2, \ldots, x_{i-1} \end{cases}$

Это общее описание *пирамидальной сортировки*, состоящей из *фазы построения* пирамиды и *фазы выбора*.

Рассмотрим сначала вопросы восстановления пирамиды. В результате обмена первого имени с последним свойство пирамидальности нарушается в корне дерева, но оба поддерева этого дерева являются пирамидами. Поэтому для восстановления пирамиды необходимо сравнить корень с большим из сыновей. Если корень больше, дерево уже является пирамидой, но если корень меньше, то его надо поменять местами с большим сыном. Далее процесс рекурсивно продолжается для поддерева, корень которого заменялся. Таким образом, процедуру *RESTORE*(f, l) восстановления пирамиды из последовательности $x_f, x_{f+1}, \ldots, x_l$ в предположении, что все поддеревья являются пирамидами, можно записать следующим образом:

procedure $RESTORE(f, l)$

$$\textbf{if } x_f \neq \text{лист } \textbf{then} \begin{cases} \text{Пусть } x_m \text{ есть больший из сыновей } x_f \\ \textbf{if } x_m > x_f \textbf{ then} \begin{cases} x_m \leftrightarrow x_f \\ RESTORE(m, l) \end{cases} \end{cases}$$

return

Очевидно, что x_f является листом только тогда, когда $f > \lfloor l/2 \rfloor$. Представив этот процесс итеративным способом и дополнив деталями, получим алгоритм 5.7. Процесс восстановления пирамиды из последовательности $x_1, \ldots, x_{11}$, в которой свойство пирамидальности было нарушено в результате обмена $x_1 = 63$ с $x_{12} = 03$ (до обмена имена $x_1, \ldots, x_{12}$ образовывали пирамиду), показан на рисунке 5.10.

procedure $RESTORE(f, l)$

$j \leftarrow f$

$$\textbf{while } j \leq \lfloor l/2 \rfloor \textbf{ do} \begin{cases} \textbf{if } 2j < l \textbf{ and } x_{2j} < x_{2j+1} \\ \quad \textbf{then } m \leftarrow 2j+1 \\ \quad \textbf{else } m \leftarrow 2j \\ \textbf{if } x_m > x_j \\ \quad \textbf{then} \begin{cases} x_m \leftrightarrow x_j \\ j \leftarrow m \end{cases} \\ \quad \textbf{else } j \leftarrow l \end{cases}$$

return

Алгоритм 5.7. Процедура восстановления пирамиды

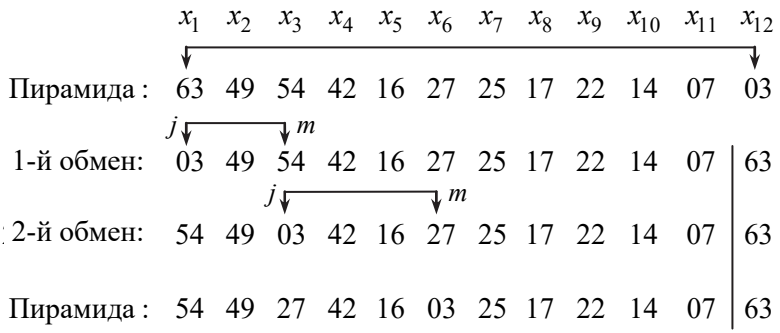


Рис. 5.10. Процесс восстановления пирамиды

Для построения пирамиды необходимо обратить внимание на то, что свойство пирамидальности уже тривиально выполняется для каждого листа, т. е. для всех x_i при $i = \lfloor n/2 \rfloor + 1, \ldots, n$. Тогда обращение к процедуре $RESTORE(i, n)$ восстановления пирамиды для $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 1$ преобразует произвольную таблицу в пирамиду на всех высших уровнях (рис. 5.11).

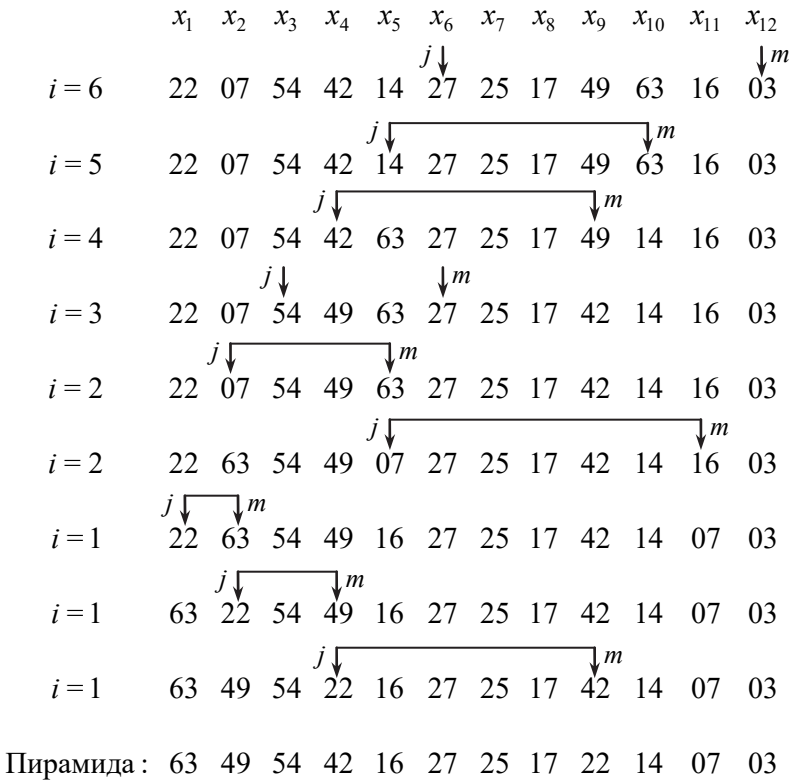


Рис. 5.11. Процесс построения пирамиды

Таким образом, полный процесс пирамидальной сортировки можно представить алгоритмом 5.8. В этом алгоритме первый цикл **for** строит пирамиду, т. е. реализует фазу построения, а второй цикл **for** реализует фазу выбора.

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do** $RESTORE(i, n)$
for $i \leftarrow n$ **downto** 2 **do** $\begin{cases} x_1 \leftrightarrow x_i \\ RESTORE(1, i-1) \end{cases}$

Алгоритм 5.8. Пирамидальная сортировка

Анализ пирамидальной сортировки достаточно сложен. Поэтому приведем здесь только результаты анализа. В фазе построения выполняется не более $O(n)$ обменов и не более $O(n)$ сравнений имен. В фазе выбора производится не более $n \log n + O(n)$ обменов и не более $2n \log n + O(n)$ сравнений имен. Таким образом, для пирамидальной сортировки любой таблицы требуется $O(n \log n)$ операций, в то время как в быстрой сортировке при некоторых таблицах совершается $O(n^2)$ операций. Несмотря на это, быстрая сортировка в среднем более эффективна, чем пирамидальная.

5.5. Распределяющая сортировка

Распределяющая сортировка основана на том, что имена распределяются по группам и содержимое групп затем объединяется таким образом, чтобы частично отсортировать таблицу; процесс повторяется до тех пор, пока таблица не будет отсортирована полностью. Такая сортировка отличается от ранее рассмотренных алгоритмов тем, что она основана не на сравнении имен, а на их представлении. Предполагается, что каждое из имен $x_1, x_2, \ldots, x_n$ имеет вид $x_i = (x_{i,p}, x_{i,p-1}, \ldots, x_{i,1})$ и их необходимо отсортировать в возрастающем лексикографическом порядке, т. е.

$$x_i = (x_{i,p}, x_{i,p-1}, \ldots, x_{i,1}) < x_j = (x_{j,p}, x_{j,p-1}, \ldots, x_{j,1})$$

тогда и только тогда, когда для некоторого $t \le p$ имеем $x_{i,l} = x_{j,l}$ для $l > t$ и $x_{i,t} < x_{j,t}$. Для простоты будем считать, что $0 \le x_{i,l} < r$, и поэтому имена можно рассматривать как целые, представленные по основанию r, т. е. каждое имя состоит из p r-ичных цифр. Для того чтобы не было имен разной длины, более короткие имена дополняются нулями. Поэтому такая сортировка часто называется *цифровой распределяющей сортировкой*.

Цифровая распределяющая сортировка основана на том, что если имена уже отсортированы по младшим разрядам l, $l-1$,

…, 1, то их можно полностью отсортировать, сортируя только по старшим разрядам $p, p-1, \ldots, l+1$ при условии, что сортировка осуществляется таким образом, чтобы не нарушить относительный порядок имен с одинаковыми цифрами в старших разрядах. Таким образом, поразрядную сортировку можно выполнить следующим образом: сначала произвести распределяющую сортировку по младшему разряду имен, переместив имена во вспомогательную область, затем произвести еще одну распределяющую сортировку по следующему разряду, переместив предварительно имена обратно в исходную область памяти, и так до тех пор, пока таблица не будет отсортирована.

Для реализации распределяющей сортировки лучше использовать связные структуры данных. В этом случае не придется перемещать имена, а достаточно скорректировать связи (указатели). Алгоритм 5.9 представляет общую процедуру цифровой распределяющей сортировки. Исходная таблица $x_1, x_2, \ldots, x_n$ организуется в очередь Q. Организуется r пустых очередей $Q_0, Q_1, \ldots, Q_{r-1}$ по одной для каждого целого числа от 0 до $r-1$.

Сформировать из $x_1, x_2, \ldots, x_n$ входную очередь Q

for $j \leftarrow 1$ **to** p **do**
- Сделать очереди $Q_0, Q_1, \ldots, Q_{r-1}$ пустыми
- **while** Q не пуста **do**
  - $X \Leftarrow Q$
  - Пусть $X = (x_p, x_{p-1}, \ldots, x_1)$
  - $Q_{x_j} \Leftarrow X$
- Сцепить очереди $Q_0, Q_1, \ldots, Q_{r-1}$ вместе для формирования новой очереди Q

Алгоритм 5.9. Цифровая распределяющая сортировка

Просматривается последовательность $x_1, x_2, \ldots, x_n$ слева направо и элемент x_i помещается в ту очередь, которая соответствует цифре младшего разряда. Как только все имена распределены по соответствующим очередям, очереди $Q_0, Q_1, \ldots, Q_{r-1}$ объединяются вместе в порядке возрастания в очередь Q. Объединение очередей подразумевает их сцепление, т. е. содержимое $(l+1)$-й очереди приписывается к концу l-й очереди (конкатенация очередей). Затем процесс повторяется для следующего разряда слева и так до тех пор, пока не будут отсортированы все

имена по всем разрядам. В результате Q будет содержать имена в порядке возрастания, начиная с головы очереди Q. Процедура такой сортировки для трехзначных десятичных чисел показана на рисунке 5.12.

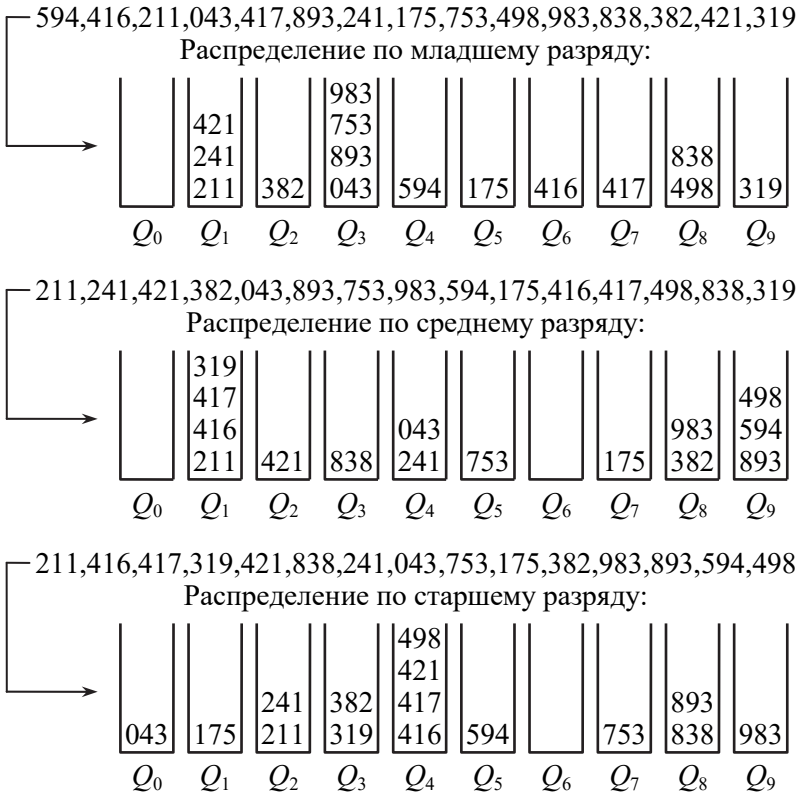


Рис. 5.12. Процесс цифровой распределяющей сортировки

Время работы алгоритма определяется общим числом операций с очередями. Всегда производится p проходов по таблице. На каждом проходе производится исключение из очереди Q каждого имени и включение его в одну из очередей Q_i. Таким образом, всего в очередях производится $2np$ операций включе-

ния/исключения. На каждом проходе выполняется также $r - 1$ сцеплений для получения очереди Q из очередей $Q_0, Q_1, \ldots, Q_{r-1}$, т. е. всего имеется $(r - 1)p$ операций сцепления. Поскольку операции включения/исключения и операции сцепления могут быть выполнены за постоянное время (не зависящее от размера очереди), требуется всего $2np + (r - 1)p$ операций с очередями. Таким образом, алгоритм требует времени $O(np + rp)$. Для случаев, когда n существенно больше r, время работы алгоритма определяется в большей степени числом имен n, чем числом разрядов, и составляет $O(n)$, т. е. цифровая распределяющая сортировка может оказаться одним из наиболее эффективных методов внутренней сортировки.

Для повышения эффективности цифровой распределяющей сортировки можно использовать компромиссный подход, который использует распределяющую сортировку только по нескольким старшим разрядам, а затем применяется простая сортировка вставками. Эффект достигается за счет того, что после распределяющей сортировки по старшим цифрам таблица становится почти упорядоченной (т. е. существенно уменьшается число инверсий), и простая сортировка вставками становится эффективной.

5.6. Сортировка подсчетом

Данная сортировка известна также как *сортировка перечислением*. Метод основан на том, что j-е имя в окончательно упорядоченной таблице превышает точно $j - 1$ остальных имен. Тогда если известно, что некоторое имя больше $j - 1$ других имен, то в отсортированной таблице оно займет j-ю позицию. Таким образом, идея сортировки заключается в попарном сравнении всех имен и подсчете, сколько из них меньше каждого отдельного имени. Очевидно, что нет необходимости сравнивать имя само с собой и после сравнения x_i с x_j не нужно сравнивать x_j с x_i.

Для реализации сортировки подсчетом необходимо каждому имени x_i исходной таблицы сопоставить элемент (счетчик) c_i, т. е. всего требуется n таких элементов. Если $x_i < x_j$, то увеличивается на единицу значение элемента c_j, в противном случае – элемента c_i. После завершения всех сравнений каждый элемент c_i будет содержать число имен, меньших имени x_i. Чтобы окончательно выполнить сортировку, достаточно поместить каждое

имя x_i в позицию $c_i + 1$ (если начальное значение $c_i = 0$) выходной таблицы. Следует отметить, что при правильной реализации сортировка подсчетом обладает свойством устойчивости.

Таким образом, сортировка подсчетом кроме исходной таблицы требует вспомогательный массив из n элементов для хранения счетчиков c_i и дополнительную выходную таблицу для формирования результатов сортировки. В целях экономии памяти сортировку подсчетом можно выполнить на месте, т. е. переразместить имена внутри исходной таблицы, используя только вспомогательный массив счетчиков. Ясно, что это приведет к некоторому увеличению времени сортировки.

Время работы сортировки подсчетом (независимо от того, используется дополнительная выходная таблица или сортировка выполняется на месте) составляет $O(n^2)$.

Разновидностью сортировки подсчетом является *сортировка распределяющим подсчетом*. Она применима в основном в тех случаях, когда исходная таблица может содержать много равных имен, причем каждое имя является целым положительным числом в диапазоне от a до b.

Сортировка выполняется следующим образом. Каждому имени i из диапазона (не из таблицы) сопоставляется элемент c_i, т. е. требуется вспомогательный массив C из $k = b - a + 1$ элементов. Сначала элементу c_i присваивается количество имен в исходной таблице, равных i. Затем находятся частичные суммы последовательности c_a, …, c_b, т. е. для всех i от $a + 1$ до b элементу c_i присваивается $c_i + c_{i-1}$. В результате значение c_i будет показывать количество имен, не превосходящих i, т. е. позицию имени i в отсортированной таблице. Для завершения сортировки имя i помещается в позицию c_i выходной таблицы. При этом необходимо учитывать следующее обстоятельство. Если все n имен в исходной таблице различны, то в отсортированной таблице имя i должно стоять в позиции c_i, так как именно столько имен в таблице не превосходит имя i. Если же встречаются равные имена, то после каждой записи имени i в выходную таблицу значение c_i должно уменьшаться на единицу, поэтому при следующей встрече с именем, равным i, оно будет записано на одну позицию левее. Чтобы сортировка была устойчивой, запись имен в выходную таблицу следует производить, просматривая исходную таблицу справа налево, начиная с имени x_n и завершая x_1.

Таким образом, сортировка распределяющим подсчетом дополнительно к исходной таблице требует вспомогательный массив из k элементов для хранения счетчиков и выходную таблицу для записи результатов сортировки. В целях экономии памяти сортировку распределяющим подсчетом можно выполнить на месте внутри исходной таблицы, что несколько усложнит алгоритм и приведет к дополнительным затратам времени.

Подсчет числа имен, равных i, и запись имен в выходную таблицу требуют времени $O(n)$, предварительная инициализация счетчиков и нахождение частичных сумм – времени $O(k)$. Таким образом, сортировка выполняется за время $O(n + k)$. Если $k = O(n)$, то время работы есть $O(n)$. Полученная оценка не противоречит нижней оценке эффективности алгоритмов сортировки, рассмотренной в п. 5.1. Это связано с тем, что нижние оценки определялись для алгоритмов сортировки, основанных на сравнении имен, а сортировка распределяющим подсчетом не сравнивает имена между собой, а использует их в качестве индексов.

5.7. Сортировка слиянием

Слияние является процессом объединения двух или более упорядоченных таблиц в одну упорядоченную таблицу. Ограничимся рассмотрением задачи слияния только двух таблиц, поскольку рассматриваемые методы достаточно легко можно обобщить для большего числа таблиц. Таким образом, необходимо решить задачу слияния двух отсортированных таблиц $x_1 \le x_2 \le \ldots \le x_n$ и $y_1 \le y_2 \le \ldots \le y_m$ в одну отсортированную таблицу $z_1 \le z_2 \le \ldots \le z_{n+m}$.

Очевидный способ слияния, называемый *прямым слиянием*, заключается в следующем: таблицы, подлежащие слиянию, просматривают параллельно, выбирая на каждом шаге меньшее из двух имен и помещая его в выходную таблицу. В алгоритме 5.10 этот процесс упрощен добавлением имен-сторожей x_{n+1} и y_{m+1}. Переменные i и j указывают соответственно на последние имена в двух входных таблицах, которые еще не были помещены в выходную таблицу.

Анализ этого алгоритма прост. Сравнение имен $x_i < y_j$ производится точно один раз для каждого имени, помещенного в выходную таблицу, т. е. выполняется $n + m$ сравнений.

$x_{n+1} \leftarrow y_{m+1} \leftarrow \infty$
$i \leftarrow j \leftarrow 1$

$$\textbf{for}\ \ k \leftarrow 1\ \ \textbf{to}\ \ n+m\ \ \textbf{do} \begin{cases} \textbf{if}\ \ x_i < y_j \\ \quad \textbf{then}\ \begin{cases} z_k \leftarrow x_i \\ i \leftarrow i+1 \end{cases} \\ \quad \textbf{else}\ \begin{cases} z_k \leftarrow y_i \\ j \leftarrow j+1 \end{cases} \end{cases}$$

Алгоритм 5.10. Прямое слияние двух отсортированных таблиц

Среднее число сравнений имен можно несколько уменьшить, если отказаться от имен-сторожей x_{n+1} и y_{m+1}, а на каждом шаге проверять, достигли соответствующие индексы правых границ исходных таблиц или нет. Если на некотором этапе одна из таблиц полностью исчерпана (все ее имена помещены в выходную таблицу), оставшиеся имена из другой таблицы просто перемещаются в выходную таблицу без сравнения имен. Очевидно, что такой алгоритм в худшем случае потребует $n + m - 1$ сравнений имен.

При $n \approx m$ прямое слияние является оптимальным. Если же одна из таблиц существенно больше другой, существует более эффективный метод слияния, называемый *бинарным слиянием*. Например, при $m = 1$ таблицы можно слить, используя бинарный поиск для отыскания места, куда нужно вставить y_1.

Пусть $n \geq m$. Идея бинарного слияния состоит в том, чтобы в самой правой части большей таблицы выделить подтаблицу, в которой будет осуществляться бинарный поиск места вставки последнего имени y_m из меньшей таблицы. Поскольку бинарный поиск наиболее эффективно работает на таблицах размера $2^k - 1$, предпочтительнее вариант, когда в выделяемой подтаблице будет $2^{\lfloor \log n/m \rfloor} - 1$ имен. Пусть имя x_l непосредственно предшествует выделенной подтаблице, т. е. $l = n + 1 - 2^{\lfloor \log n/m \rfloor}$. Выполняется сравнение самого правого имени y_m из меньшей таблицы с именем x_l. Если $y_m < x_l$, тогда x_l и вся выделенная подтаблица большей таблицы вкладываются в выходную таблицу. Если $y_m \geq x_l$, то осуществляется бинарный поиск места вставки имени y_m в выделенной подтаблице (обозначим через k наибольшее целое, такое, что $y_m > x_k$). Затем y_m и все имена $x_{k+1}, \ldots, x_n$ (т. е. имена из подтаблицы, о которых стало известно, что они больше y_m) помеща-

ются в выходную таблицу. Далее процесс повторяется для изменившихся размеров m и n таблиц.

На рисунке 5.13 показан первый шаг слияния двух таблиц: $X = \{x_1, \ldots, x_{20}\}$ и $Y = \{y_1, \ldots, y_4\}$. В большей таблице X выделяется подтаблица размера $2^{\lfloor \log 20/4 \rfloor} - 1 = 3$, состоящая из имен x_{18}, x_{19}, x_{20}. Если $y_4 < x_l = x_{17}$, то $x_{17}, x_{18}, x_{19}, x_{20}$ можно поместить в выходную таблицу, и продолжается слияние таблиц $x_1, x_2, \ldots, x_{16}$ с y_1, y_2, y_3, y_4. Если $y_4 \geq x_{17}$, то используется бинарный поиск для отыскания позиции y_4 среди x_{18}, x_{19}, x_{20} за два сравнения и помещаются y_4 и все $x_i > y_4$ в выходную таблицу. Процесс слияния продолжается для таблиц $x_1, x_2, \ldots, x_k$ и y_1, y_2, y_3, где k – наибольшее целое, такое что $y_4 > x_k$.

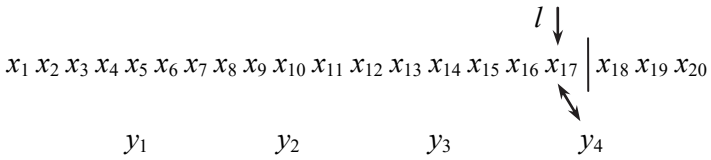


Рис. 5.13. Один шаг бинарного слияния

Общая процедура бинарного слияния представлена алгоритмом 5.11, где альтернативы **then** и **else**, относящиеся к **if** $m \leq n$, одинаковы во всем, кроме того, что роли x и y (соответственно n и m) меняются местами.

Число сравнений имен, производимых алгоритмом бинарного слияния при объединении таблиц $x_1, x_2, \ldots, x_n$ и $y_1, y_2, \ldots, y_m$ в худшем случае, составляет [8]

$$C(m, n) = m + \lfloor n / 2^t \rfloor - 1 + t\,m \quad \text{при } m \leq n, \text{ где } t = \lfloor \log n/m \rfloor.$$

При $m = n$ имеем $C(n, n) = 2n - 1$, а при $m = 1$ получаем

$$C(1, n) = 1 + \lfloor n / 2^{\lfloor \log n \rfloor} \rfloor - 1 + \lfloor \log n \rfloor = 1 + \lfloor \log n \rfloor,$$

т. е. алгоритм бинарного слияния работает как прямое слияние при $m = n$ и как бинарный поиск при $m = 1$, а также достаточно эффективно работает для промежуточных значений m.

Рассмотрим применение слияния для сортировки таблицы $T = \{x_1, x_2, \ldots, x_n\}$. Поскольку для слияния необходимо иметь по крайней мере две отсортированные последовательности, возникает вопрос о разбиении исходной таблицы на упорядоченные подтаблицы и их слияния. Такие упорядоченные сегменты данных называются *отрезками* (или *сериями*).

while $n \neq 0$ **and** $m \neq 0$ **do**

- **if** $m \leq n$
  - $t \leftarrow \lfloor \log\ n/m \rfloor$
  - **if** $y_m < x_{n+1-2^t}$
    - **then**
      - Поместить $x_{n+1-2^t}, \ldots, x_n$ в выходную таблицу
      - $n \leftarrow n - 2^t$
    - **else**
      - Используя t сравнений имен, бинарным поиском в $x_{n+2-2^t}, \ldots, x_n$ определить k – наибольшее целое, такое что $y_m > x_k$.
      - Поместить $y_m, x_{k+1}, \ldots, x_n$ в выходную таблицу
      - $m \leftarrow m - 1$
      - $n \leftarrow k$
  - $t \leftarrow \lfloor \log\ m/n \rfloor$
  - **if** $x_n < y_{m+1-2^t}$
    - **then**
      - Поместить $y_{m+1-2^t}, \ldots, y_m$ в выходную таблицу
      - $m \leftarrow m - 2^t$
    - **else**
      - Используя t сравнений имен, бинарным поиском в $y_{m+2-2^t}, \ldots, y_m$ определить k – наибольшее целое, такое что $x_n > y_k$.
      - Поместить $x_n, y_{k+1}, \ldots, y_m$ в выходную таблицу
      - $n \leftarrow n - 1$
      - $m \leftarrow k$

if $n = 0$

- **then** Поместить $y_1, y_2, \ldots, y_m$ в выходную таблицу
- **else** Поместить $x_1, x_2, \ldots, x_n$ в выходную таблицу

Алгоритм 5.11. Бинарное слияние

Одним из вариантов сортировки слиянием является *естественное двухпутевое слияние*, которое выбирает из исходной таблицы упорядоченные подпоследовательности (отрезки) и объединяет их в более длинные. Процесс сортировки проиллю-

стрирован на рисунке 5.14, где вертикальными линиями отмечены границы между отрезками, а стрелки показывают направление упорядочения внутри отрезков.

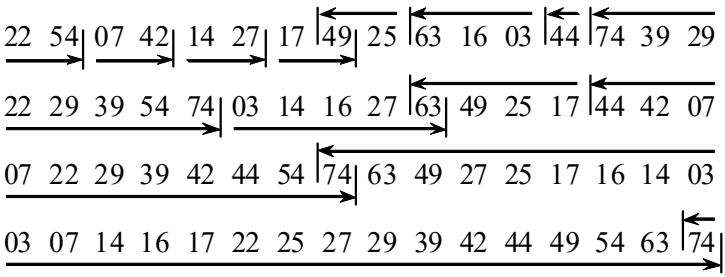


Рис. 5.14. Процесс сортировки естественным двухпутевым слиянием

Исходная таблица анализируется слева и справа, двигаясь к середине. Это необходимо для того, чтобы был доступ к двум отрезкам для слияния: один отрезок в левой части, а второй – в правой части таблицы, причем отрезок правой части должен быть упорядочен справа налево. Вторая строка формируется из первой следующим образом. Слева имеется отрезок (22, 54), а справа, если читать справа налево, отрезок (29, 39, 74). Слияние этих отрезков дает последовательность (22, 29, 39, 54, 74), которая помещается в левую часть вспомогательной таблицы (строка 2). Затем отрезок (07, 42) сливается с отрезком (44) и результат (07, 42, 44) записывается в правую часть вспомогательной таблицы. Результат (03, 14, 16, 27, 63) слияния отрезков (14, 27) и (03, 16, 63) помещается в левую часть вслед за ранее записанной последовательностью. Наконец, отрезок (17, 49) сливается с отрезком (25, 49) и полученный результат (17, 25, 49) записывается в правую часть перед ранее записанной последовательностью. Затем производится переключение таблиц (вспомогательная таблица становится исходной, а исходная – вспомогательной) и процесс анализа, слияния и переключения повторяется до тех пор, пока не будет сформирован единственный отрезок – отсортированная таблица.

В общем случае в середине таблицы возникает перекрытие отрезков, когда при движении с обоих концов считывается одно и то же имя. Такая ситуация должна обнаруживаться алгоритмом прежде, чем она может привести к осложнениям.

Таким образом, для реализации естественного двухпутевого слияния необходимы две таблицы размера n (исходная и вспомогательная). Причем результат сортировки может сформироваться в любой из них. Если определено требование, что результат должен быть в исходной таблице, то придется выполнить копирование данных из вспомогательной таблицы в исходную.

Чтобы упростить переключение таблиц, для хранения второй таблицы обычно отводят область памяти, располагающуюся вслед за исходной таблицей, т. е. $x_{n+1}, \ldots, x_{2n}$. Тогда переключение таблиц реализуется соответствующей установкой значений индексов.

Для переключения направления вывода (по возрастанию индекса – в левой части таблицы и по убыванию – в правой) достаточно использовать переменную для хранения шага приращения индекса и изменять ее знак при переключении.

Естественное двухпутевое слияние обладает тем преимуществом, что исходные таблицы с преобладанием возрастающего или убывающего расположения имен обрабатываются очень быстро. Но при этом приходится постоянно проверять, достигнут конец отрезка или нет (их длина заранее не известна и определяется случайным расположением имен), что приводит к замедлению процесса сортировки в общем случае. Чтобы избежать этого, можно использовать на каждом шаге фиксированные длины отрезков. В исходной таблице все отрезки имеют длину 1, после первого шага все отрезки (кроме, возможно, последнего) имеют длину 2, ..., после k-го шага – длину 2^k (кроме, возможно, последнего). Такой способ называется *простым двухпутевым слиянием*. В остальном общая схема слияния аналогична естественному слиянию. Пример простого двухпутевого слияния показан на рисунке 5.15.

Поскольку длина отрезков фиксирована для каждого шага, можно заранее предсказать положение отсортированной таблицы (в исходной или в дополнительной): если значение $\lceil \log n \rceil$ нечетно, то результат окажется в дополнительной таблице, если четно – в исходной.

Время работы алгоритмов слияния (как естественного, так и простого) составляет $O(n \log n)$. Однако в среднем простое слияние несколько лучше естественного.

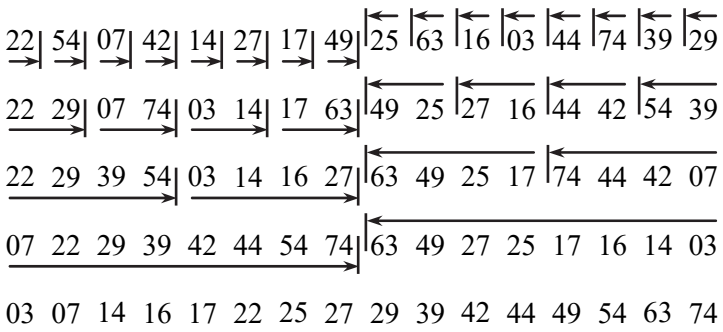


Рис. 5.15. Процесс сортировки простым двухпутевым слиянием

Рассмотренные ранее методы сортировки слиянием требуют памяти для хранения $2n$ имен. При этом в любой момент времени половина памяти не используется, а остается зарезервированной для последующего заполнения. Более логичной структурой данных является связный список. Тогда все необходимые операции перемещения имен можно заменить операциями со связями. Как правило, добавление n полей связи выгоднее добавления пространства памяти еще для n имен, так как отказавшись от перемещения имен, можно выиграть во времени. В процессе сортировки поля связи устанавливаются таким образом, что имена оказываются связанными в порядке возрастания. Такая сортировка называется *сортировкой слиянием списков*. Для реализации такой сортировки связные списки удобнее представлять с помощью массивов, используя индекс массива в качестве указателя на элемент массива, следующего за текущим элементом.

Слияние списков можно реализовать как для естественного, так и для простого слияния. Причем при использовании связного распределения естественное слияние предпочтительнее простого.

Хорошо известен и широко используется рекурсивный вариант сортировки слиянием, представленный алгоритмом 5.12. Процедура *MERGE* выполняет слияние двух отсортированных подтаблиц (необходима такая модификация алгоритма 5.10, чтобы результат слияния получался в исходной таблице). Процедура выполняет сортировку подтаблицы $x_i, x_{i+1}, \ldots, x_j$, состоящей из k имен ($k = j - i + 1$). Если $i \geq j$, подтаблица содержит не более одного имени, т. е. она отсортирована. В противном случае производится разбиение на две подтаблицы: $x_i, \ldots, x_m$ с $\lceil k/2 \rceil$ именами

и x_{m+1}, …, x_j с $\lfloor k/2 \rfloor$ именами. Для сортировки исходной таблицы из n имен первым обращением к процедуре является $MERGESORT(1, n)$. В ходе работы алгоритма производится попарное слияние одноэлементных отрезков в отрезок длиной 2, затем – двухэлементных отрезков в отрезок длиной 4 и так до тех пор, пока не будет получен один отрезок длиной n, т. е. отсортированная таблица. Таким образом, процедура реализует, по сути, простое двухпутевое слияние.

procedure $MERGESORT(i, j)$
  if $i < j$
    then $\begin{cases} m \leftarrow \lfloor (i+j)/2 \rfloor \\ MERGE(MERGESORT(i, m), MERGESORT(m+1, j)) \end{cases}$
return

Алгоритм 5.12. Рекурсивный вариант сортировки слиянием

Время работы сортировки слиянием в худшем случае определяется рекуррентным соотношением

$$T(1) = O(1),$$
$$T(n) = 2T(n/2) + n - 1,$$

решением которого является $T(n) = O(n \log n)$ (см. п. 1.4).

Исследование процесса работы данного алгоритма предлагается в качестве упражнения.

5.8. Гибридный алгоритм сортировки Timsort

Большое распространение получила сортировка *Timsort*, опубликованная в 2002 году Тимом Петерсом. В настоящее время она является стандартным алгоритмом сортировки в Python, OpenJDK 7 и реализован в Android JDK 1.5. *Timsort* представляет собой гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием. Идея алгоритма заключается в следующем. По специальному алгоритму сортируемая таблица разделяется на подтаблицы. Каждая подтаблица сортируется простой сортировкой вставками. Отсортированные подтаблицы (отрезки) собираются в единую таблицу с помощью модифицированной сортировки слиянием.

Минимальный размер подтаблиц k определяется исходя из следующих принципов: k не должно быть слишком большим (к подтаблице будет применена сортировка вставками, а она эф-

фективна только на небольших массивах) и не должно быть слишком маленьким (чем больше число сформированных отрезков, тем больше операций слияния придётся выполнить). Оптимальное значение для n / k – это степень числа 2 (или близкая к нему). Это требование обусловлено тем, что алгоритм слияния отрезков наиболее эффективно работает на отрезках примерно равного размера. Автор алгоритма ссылается на собственные эксперименты, показавшие, что наиболее эффективно использовать значения k из диапазона (32;65). Исключение – если $n < 64$, тогда $k = n$ и *Timsort* превращается в простую сортировку вставками.

Процесс формирования подтаблиц начинается с первого имени таблицы. Выполняется поиск упорядоченной подпоследовательности (отрезка). Если получившаяся подпоследовательность упорядочена по убыванию, имена переставляются так, чтобы они шли по возрастанию. Если размер текущего отрезка меньше k, подтаблица дополняется следующими за найденным отрезком именами до размера k. В результате будет получена подтаблица размером k или больше, к которой применяется сортировка вставками. Начиная со следующего за сформированным отрезком имени, аналогично формируется новый отрезок и так до тех пор, пока вся таблица не будет разбита на отрезки.

Завершающим этапом алгоритма является слияние полученных отрезков. Следует объединять отрезки примерно равного размера. Для этого создается пустой стек для хранения пар (b, s), где b – номер позиции начала отрезка, s – размер отрезка. В стек добавляется (b, s) первого отрезка. Пусть на некотором этапе в верхней части стека находятся данные о трех отрезках X, Y и Z с размерами s_X, s_Y и s_Z соответственно. Для определения, необходимо ли выполнять слияние отрезков, проверяется выполнение двух правил: $s_X > s_Y + s_Z$ и $s_Y > s_Z$. Если одно из правил нарушается, отрезок Y сливается с меньшим из отрезков X и Z. Процесс повторяется до выполнения обоих правил или полного упорядочивания данных. Если еще остались нерассмотренные отрезки, берётся следующий отрезок и его данные заносятся в стек и процесс повторяется. В противном случае процесс завершается.

Время работы алгоритма $O(n)$ в лучшем случае и $O(n \log n)$ в среднем и худшем случаях. Особенно хорошо алгоритм работает с частично отсортированными таблицами. Детали реализации алгоритма предлагаются в качестве упражнения.

5.9. Внешняя сортировка

Внешней называется сортировка, когда решается задача полной сортировки для случая такой большой таблицы, не умещающейся в оперативной памяти, что доступ к ней организован по частям, расположенным на внешних запоминающих устройствах. Внешняя сортировка файлов в корне отличается от внутренней, поскольку время доступа к файлам на внешних носителях накладывает существенные ограничения, кроме того, в каждый момент непосредственно доступна только одна запись (для последовательного файла) или блок записей (для файла с прямым доступом), но не весь файл целиком. Поэтому многие методы внутренней сортировки фактически бесполезны для внешней сортировки. Основой внешней сортировки является слияние. Сортировка слиянием применяется в основном для внешней, а не для внутренней сортировки, причем используется обычно многопутевое слияние.

При изучении основных идей внешней сортировки будем рассматривать только сортировку таблицы, организованной в виде последовательного файла как наихудшего с точки зрения доступа. Будем считать, что имеется $t + 1$ файлов, один из которых представляет собой таблицу имен $x_1, x_2, \ldots, x_n$. Остальные t файлов являются рабочими. Файл с исходной таблицей служит для ввода данных. Будем считать, что внутренняя память вместе с другими данными, программами и прочей информацией может содержать одновременно только m имен. Предполагается, что размер n входной таблицы значительно превышает m.

Общей стратегией внешней сортировки является использование внутренней памяти для сортировки имен из файла по частям, т. е. формирование множества упорядоченных подтаблиц (*исходных отрезков*) из исходной таблицы. По мере порождения эти отрезки распределяются по рабочим файлам, затем производится их слияние обратно в исходный файл так, что он будет содержать меньшее число более длинных отрезков. Далее полученные отрезки снова распределяются по рабочим файлам и снова производится их слияние и т. д. Процесс продолжается до тех пор, пока не получится единственный отрезок – отсортированная таблица. Таким образом, имеются две отдельные проблемы: как порождать исходные отрезки и как осуществлять слияние.

5.9.1. Порождение исходных отрезков

Самый простой метод порождения исходных отрезков заключается в том, что из исходной таблицы считываются m имен, производится их сортировка с использованием любого из методов внутренней сортировки и полученная отсортированная последовательность имен записывается в файл в виде отрезка; далее считываются следующие m имен, из которых формируется следующий отрезок, и так до тех пор, пока не будут исчерпаны все имена исходной таблицы. Все полученные таким образом отрезки содержат m имен, за исключением, возможно, последнего отрезка, в котором может оказаться менее m имен (когда n не кратно m).

Очевидно, что число исходных отрезков в конечном счёте определяет время слияния. Поэтому для повышения эффективности внешней сортировки необходимо стремиться к образованию меньшего количества более длинных исходных отрезков. Это можно сделать, если использовать технику *выбора с замещением*. Выбор с замещением основан на идее пирамидальной сортировки. При этом подходе m имен, которые умещаются в памяти, хранятся в виде такой пирамиды, что имена в сыновьях вершины больше имени в самой вершине (вместо того чтобы быть меньше, как это имеет место в пирамидальной сортировке).

Выбор с замещением порождает исходные отрезки следующим образом. Из входного файла считываются первые m имен, затем из них формируется пирамида (в соответствии с указанным выше принципом организации), в которой корнем является наименьшее имя. Наименьшее имя выводится как первое в первом отрезке и заменяется в пирамиде следующим именем из входного файла. Поскольку в результате такой замены может нарушиться свойство пирамидальности, далее производится восстановление пирамиды в соответствии с алгоритмом 5.7 (процедура *RESTORE*), модифицированным так, чтобы для восстановления пирамиды следить за наименьшим, а не за наибольшим именем. Процесс продолжается таким образом, что к текущему отрезку всегда добавляется наименьшее в пирамиде имя, большее или равное имени, которое последним добавлено к отрезку. Добавленное имя заменяется на следующее имя из входного файла, и восстанавливается пирамида. Когда в пирамиде нет

имен, больших, чем последнее имя в текущем отрезке, отрезок обрывается и начинается новый отрезок. Этот процесс продолжается до тех пор, пока все имена не сформируются в отрезки.

При реализации процедуры выбора с замещением каждое имя x удобно рассматривать как пару (r, x), где r есть номер отрезка, в котором находится x. Таким образом, можно считать, что пирамида состоит из пар $(r_1, x_1), (r_2, x_2), \ldots, (r_m, x_m)$, причем сравнения между парами осуществляются лексикографически. Тогда, если на некотором шаге считывается имя, меньшее последнего имени в текущем отрезке, оно должно быть в следующем отрезке, и благодаря наличию номера отрезка это имя будет ниже всех имен пирамиды, которые входят в текущий отрезок, т. е. $(r_1, x_1) < (r_2, x_2)$, если $r_1 < r_2$ или $r_1 = r_2$ и $x_1 < x_2$.

Очевидно, что выбор с замещением формирует исходные отрезки не хуже, чем простой метод, так как все отрезки (кроме, возможно, последнего) содержат не меньше m имен. В лучшем случае может быть сформирован единственный исходный отрезок, т. е. можно сразу получить отсортированную таблицу.

5.9.2. Распределение и слияние отрезков

После формирования исходных отрезков возникает задача распределения их по рабочим файлам и слияния их до тех пор, пока не получится отсортированная таблица.

Простейший метод заключается в равномерном распределении отрезков по файлам $1, 2, \ldots, t$ с последующим их слиянием в файл $t + 1$. Полученные в результате более длинные отрезки снова равномерно распределяются по файлам $1, 2, \ldots, t$ и производится их слияние (образуются более длинные отрезки) в файл $t + 1$. Процесс продолжается до тех пор, пока в файле $t + 1$ не останется только один отрезок (отсортированная таблица). Очевидно, что каждый проход сокращает число отрезков в t раз (из t отрезков в результате слияния получается один отрезок). Следовательно, если имеется r исходных отрезков, то потребуется $\lceil \log_t r \rceil$ проходов, где каждый проход состоит из *фазы переписывания* (распределения отрезков по рабочим файлам), за которой следует *фаза слияния*. Таким образом, имеется всего $2\lceil \log_t r \rceil \approx (2 / \log t) \lceil \log r \rceil$ проходов по именам, половина которых не уменьшает числа отрезков.

Более эффективные методы распределения и слияния отрезков основаны на исключении фазы переписывания, которая не сокращает числа отрезков. Одним из таких методов является *многофазное слияние*. Идея многофазного слияния состоит в том, чтобы организовать исходные отрезки так, что после каждого слияния, кроме последнего, остается ровно один пустой файл, который будет принимающим при следующем слиянии. В последнем слиянии должно участвовать только по одному отрезку из каждого t непустых файлов. Распределение исходных отрезков с такими свойствами называется *совершенным распределением*. Пример многофазного слияния для 57 исходных отрезков, распределенных в соответствии с совершенным распределением по четырем файлам, показан на рисунке 5.16. На этом рисунке запись вида $n * i$ означает n отрезков порядка i (отрезок порядка i есть результат слияния i исходных отрезков).

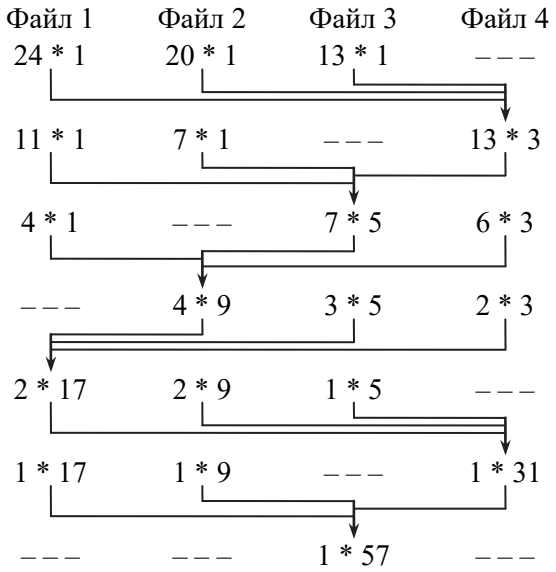


Рис. 5.16. Многофазное слияние 57 исходных отрезков

Можно легко определить формулу совершенного распределения исходных отрезков для общего случая, двигаясь в обратном направлении. Пусть $a_{k,j}$ – число отрезков в j-м файле, когда

остается осуществить k фаз слияния. Будем считать, что файл $t+1$ всегда является принимающим и что файл 1 содержит не меньше отрезков, чем файл 2, который содержит не меньше отрезков, чем файл 3, и т. д. Это можно сделать, используя логические номера файлов и переключая их соответствующим образом. Тогда в самом конце, когда больше проходов со слиянием не остается, имеем распределение

$$\begin{array}{rccccccc} j: & 1 & 2 & 3 & \ldots & t-1 & t & t+1 \\ a_{0,j}: & 1 & 0 & 0 & \ldots & 0 & 0 & 0 \end{array}$$

Когда остается k слияний, имеем распределение

$$\begin{array}{rccccccc} j: & 1 & 2 & 3 & \ldots & t-1 & t & t+1 \\ a_{k,j}: & a_{k,1} & a_{k,2} & a_{k,3} & \ldots & a_{k,t-1} & a_{k,t} & 0 \end{array}$$

Тогда следующее слияние приведет к распределению

$$\begin{array}{rccccccc} j: & 1 & 2 & 3 & \ldots & t-1 & t & t+1 \\ a_{k-1,j}: & a_{k,1}-a_{k,t} & a_{k,2}-a_{k,t} & a_{k,3}-a_{k,t} & \ldots & a_{k,t-1}-a_{k,t} & 0 & a_{k,t} \end{array}$$

Очевидно, что для совершенного распределения должно выполняться условие $2a_{k,t} \geq a_{k,1}$ при $k \geq 1$. Тогда переключение логических номеров файлов, при котором числа отрезков расположены в убывающем порядке, дает распределение

$$\begin{array}{rccccccc} j: & 1 & 2 & 3 & \ldots & t-1 & t & t+1 \\ a_{k-1,j}: & a_{k,t} & a_{k,1}-a_{k,t} & a_{k,2}-a_{k,t} & \ldots & a_{k,t-2}-a_{k,t} & a_{k,t-1}-a_{k,t} & 0 \end{array}$$

Поэтому

$$\begin{aligned} a_{k-1,1} &= a_{k,t}, \\ a_{k-1,j} &= a_{k,j-1} - a_{k,t}, 2 \leq j \leq t, \end{aligned}$$

или

$$\begin{aligned} a_{k+1,t} &= a_{k,1}, \\ a_{k+1,j} &= a_{k,j+1} + a_{k,1}, 1 \leq j < t. \end{aligned}$$

Следовательно, совершенные распределения для $t+1$ файлов можно представить следующим образом:

Файлы:	1	2	3	$\cdots$	$t-2$	$t-1$	t	$t+1$
$k=0$	1	0	0	$\cdots$	0	0	0	0
$k=1$	1	1	1	$\cdots$	1	1	1	0
$k=2$	2	2	2	$\cdots$	2	2	1	0
$k=3$	4	4	4	$\cdots$	4	3	2	0
$\vdots$				$\cdots$				
k	$a_{k,1}$	$a_{k,2}$	$a_{k,3}$	$\cdots$	$a_{k,t-2}$	$a_{k,t-1}$	$a_{k,t}$	0
$k+1$	$a_{k,2}+a_{k,1}$	$a_{k,3}+a_{k,1}$	$a_{k,4}+a_{k,1}$	$\cdots$	$a_{k,t-1}+a_{k,1}$	$a_{k,t}+a_{k,1}$	$a_{k,1}$	0

Очевидно, что число исходных отрезков не всегда удовлетворяет совершенному распределению. Выходом из такой ситуации является добавление фиктивных отрезков для получения требуемого распределения. Фиктивные отрезки прослеживаются алгоритмом многофазного слияния с помощью специальных счетчиков (обычно для каждого файла организуется свой счетчик), но не записываются физически в файлы. При этом фиктивные отрезки рекомендуется распределять как можно более равномерно по t файлам.

5.10. Порядковые статистики

Задачей вычисления *порядковых статистик* является *задача выбора k-го наименьшего имени* в таблице $T = \{x_1, x_2, \ldots, x_n\}$, т. е. имени, которое будет находиться в k-й позиции, если расположить имена в таблице в возрастающем порядке. Такое имя называется *k-й порядковой статистикой*. Для удобства будем считать, что таблица состоит из различных имен. Можно выделить специальные случаи: $k = 1$ (нахождение минимума), $k = n$ (нахождение максимума) и, если n нечетно, $k = (n + 1) / 2$ (нахождение *медианы*).

Задача, очевидно, симметрична: отыскание $(n – k + 1)$-го наименьшего (k-го наибольшего) имени можно осуществить, используя алгоритм отыскания k-го наименьшего имени, но меняя местами действия, предпринимаемые при результатах сравнений имен (больше и меньше). Таким образом, отыскание наименьшего имени ($k = 1$) эквивалентно отысканию наибольшего имени ($k = n$); отыскание второго наименьшего имени ($k = 2$) эквивалентно отысканию второго наибольшего ($k = n – 1$) и т. д. Поэтому можно считать, что $k \le \lceil n/2 \rceil$.

Для решения задачи выбора можно использовать любой из методов полной сортировки имен с последующим тривиальным обращением к k-му наименьшему имени. Такой подход требует порядка n^2 или $n \log n$ сравнений имен независимо от значения k. При таком подходе выполняется гораздо больше работы, чем требуется, поэтому необходимо искать более эффективные методы решения задачи.

Другой подход к решению задачи выбора основан на использовании одного из алгоритмов сортировки, основанных на

выборе, т. е. либо простая сортировка выбором, либо пирамидальная сортировка. Очевидно, что для этого алгоритм простой сортировки выбором (алгоритм 5.6) необходимо модифицировать так, чтобы осуществлялся выбор не i-го наибольшего, а i-го наименьшего имени. Для пирамидальной сортировки (алгоритм 5.8) имена должны храниться в виде такой пирамиды, что имена в сыновьях вершины больше имени в самой вершине (корнем должно быть наименьшее имя). В каждом случае процесс сортировки можно остановить после выполнения первых k шагов. Для простой сортировки выбором это означает использование

$$(n-1)+(n-2)+\cdots+(n-k)=kn-\frac{k(k+1)}{2}$$

сравнений имен, а для пирамидальной сортировки – использование $n + k \log n$ сравнений имен. Здесь также выполняется избыточная работа, так как полностью определяется порядок k наименьших имен, но объем этой работы при данном подходе зависит от k.

Более эффективный подход к решению задачи выбора k-го наименьшего имени основан на идее быстрой сортировки. Таблица разбивается на две подтаблицы (меньше x_1 и больше x_1), а затем подходящая подтаблица исследуется рекурсивно. Предположим, что x_1 попадает в позицию j, т. е. x_1 является j-м наименьшим именем. Если $k = j$, то процедура заканчивается; если $k < j$, то поиск k-го наименьшего имени продолжается среди $x_1, x_2, \ldots, x_{j-1}$ (в левой подтаблице); если $k > j$, продолжается поиск $(k - j)$-го наименьшего имени среди $x_{j+1}, \ldots, x_n$ (в правой подтаблице). Анализ эффективности такого алгоритма подобен анализу алгоритма быстрой сортировки. Пусть $C_{\text{ave}}(n, k)$ – среднее число сравнений, требующееся для отыскания k-го наименьшего имени. Тогда

$$C_{\text{ave}}(n,k)=n+1+\frac{1}{n}\sum_{j=1}^{k-1}C_{\text{ave}}(n-j,k-j)+\frac{1}{n}\sum_{j=k+1}^{n}C_{\text{ave}}(j-1,k)\,.$$

Тривиальная индукция по k дает $C_{\text{ave}}(n, k) = O(n)$, т. е. данный метод требует в среднем только $O(n)$ сравнений имен независимо от значения k. К сожалению, число сравнений имен в худшем случае пропорционально n^2, поэтому в ряде случаев он может быть чрезвычайно неэффективным. Как и для алгоритма

быстрой сортировки, причиной неэффективности является то, что расщепляющее имя может оказаться слишком близким к одному из двух краев таблицы (вместо того, чтобы быть ближе к середине). Для улучшения алгоритма необходимо применять способы эффективного нахождения расщепляющего имени, которое близко к середине таблицы. Вполне приемлемые результаты дает использование медианы малой случайной выборки, но наиболее близка к середине таблицы медиана медиан, что делает алгоритм пригодным даже для малых значений n.

Упражнения

1. Разработать и проанализировать алгоритм сортировки бинарными вставками.

2. Разработать и проанализировать алгоритм сортировки вставками в связный список.

3. Разработать алгоритм шейкер-сортировки.

4. Модифицировать алгоритм сортировки Шелла (алгоритм 5.2) так, чтобы отказаться от имен-сторожей.

5. Разработать алгоритм обменной сортировки, подобный сортировке Шелла, но использующий в качестве h-сортировки пузырьковую сортировку.

6. Модифицировать итерационный алгоритм быстрой сортировки так, чтобы для сортировки подтаблиц длины меньше m использовалась простая сортировка вставками.

***7**. Разработать алгоритм цифровой обменной сортировки. Проанализировать лучший и худший случаи:

а) для числа обменов;

б) числа просмотров разрядов;

в) максимальной глубины стека,

считая, что имена, которые необходимо отсортировать, есть числа 0, 1, 2, …, $2^t - 1$, расположенные в случайном порядке.

8. Разработать алгоритм цифровой распределяющей сортировки, дополнив алгоритм 5.9 необходимыми операциями с очередями.

***9**. Разработать алгоритм, использующий распределяющую сортировку только по нескольким старшим разрядам, а затем простую сортировку вставками.

10. Разработать алгоритм сортировки подсчетом, обладающий свойством устойчивости.

***11**. Разработать алгоритм сортировки подсчетом, чтобы сортировка выполнялась на месте, т. е. переразмещение имен выполнялось

внутри исходной таблицы, используя только вспомогательный массив счетчиков.

12. Разработать алгоритм сортировки распределяющим подсчетом, обладающий свойством устойчивости.

13. Модифицировать алгоритм прямого слияния (алгоритм 5.10) так, чтобы отказаться от имен-сторожей x_{n+1} и y_{m+1} и чтобы алгоритм использовал не более $n + m - 1$ сравнений имен для слияния двух отсортированных таблиц.

14. Разработать алгоритм бинарного слияния, дополнив алгоритм 5.11 необходимыми операциями для реализации бинарного поиска места вставки и перемещения имен в выходную таблицу.

15. Разработать алгоритм внутренней сортировки, основанный на естественном двухпутевом слиянии.

16. Разработать алгоритм внутренней сортировки, основанный на простом двухпутевом слиянии.

17. Разработать алгоритм сортировки слиянием списков для естественного двухпутевого слияния.

18. Разработать алгоритм сортировки слиянием списков для простого двухпутевого слияния.

19. Исследовать процесс работы рекурсивной процедуры сортировки слиянием (алгоритм 5.12), построив древовидную структуру, отражающую процесс слияния отрезков.

***20**. Исследовать детали реализации и разработать гибридный алгоритм сортировки *Timsort* (см. п. 5.8).

21. Разработать алгоритм выбора с замещением, модифицировав алгоритм восстановления пирамиды (алгоритм 5.7) так, чтобы сравниваемые элементы были парами (r, x), а наименьшее имя хранилось в вершине пирамиды.

***22**. Рассмотренные методы внешней сортировки требуют по крайней мере три файла. Реализовать внешнюю сортировку с двумя файлами, основанную на комбинации идей пузырьковой сортировки и выбора с замещением.

23. Реализовать подход быстрой сортировки к решению задачи выбора k-го наименьшего имени.

Глава 6. АЛГОРИТМЫ НА ГРАФАХ

Множество разнообразных задач теоретического и прикладного характера естественно формулируется в терминах неориентированных или ориентированных графов. Обычно решение задачи включает анализ графа или проверку его на наличие определенных свойств. Графы, соответствующие реальным задачам, часто громоздки и сложны. Поэтому большое практическое значение имеет разработка эффективных алгоритмов на графах и знание основных способов машинного представления графов.

Граф будем обозначать $G = (V, E)$, где V – множество вершин графа, E – множество ребер графа. Будем использовать символы $|V|$ и $|E|$ для обозначения соответственно числа вершин и числа ребер в графе.

6.1. Представления графов

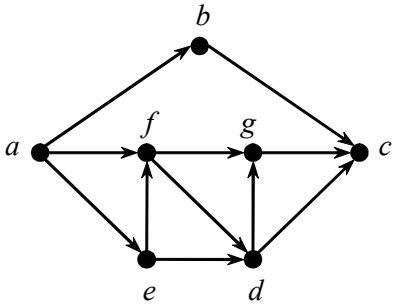
Выбор соответствующей структуры данных для представления графов оказывает существенное влияние на эффективность алгоритмов. Рассмотрим наиболее распространенные способы представления графов и их основные достоинства и недостатки.

Матрица смежности. Матрица смежности графа $G = (V, E)$ есть матрица $A = [a_{ij}]$ размера $|V| \times |V|$, в которой $a_{ij} = 1$, если $(v_i, v_j) \in E$, т. е. в G существует ребро, соединяющее вершины v_i и v_j, и $a_{ij} = 0$ в противном случае. Матрица смежности для ориентированного графа (рис. 6.1*а*) представлена на рисунке 6.1*б*.

Необходимо отметить, что в неориентированном графе ребро (v_i, v_j) идет как от v_i к v_j, так и от v_j к v_i. Поэтому матрица смежности такого графа всегда является симметричной.

Основным достоинством матрицы смежности является то, что время, необходимое для определения наличия некоторого ребра, фиксировано и не зависит от $|V|$ и $|E|$. Поэтому такое представление удобно для тех алгоритмов, в которых часто нужно знать, есть ли в графе данное ребро или нет.

Недостаток заключается в том, что независимо от числа ребер матрица занимает память объема $|V|^2$. На практике это неудобство можно иногда уменьшить, храня целую строку (столбец) матрицы в одном машинном слове.



а

	a	*b*	*c*	*d*	*e*	*f*	*g*
a	0	1	0	0	1	1	0
b	0	0	1	0	0	0	0
c	0	0	0	0	0	0	0
d	0	0	1	0	0	0	1
e	0	0	0	1	0	1	0
f	0	0	0	1	0	0	1
g	0	0	1	0	0	0	0

б

	(a,b)	(a,e)	(a,f)	(b,c)	(d,c)	(d,g)	(e,d)	(e,f)	(f,d)	(f,g)	(g,c)
a	1	1	1	0	0	0	0	0	0	0	0
b	−1	0	0	1	0	0	0	0	0	0	0
c	0	0	0	−1	−1	0	0	0	0	0	−1
d	0	0	0	0	1	1	−1	0	−1	0	0
e	0	−1	0	0	0	0	1	1	0	0	0
f	0	0	−1	0	0	0	0	−1	1	1	0
g	0	0	0	0	0	−1	0	0	0	−1	1

в

$$g = (a, a, a, b, d, d, e, e, f, f, g)$$
$$h = (b, e, f, c, c, g, d, f, d, g, c)$$

г

v	Adj(*v*)
a	*b*,*e*,*f*
b	*c*
c	–
d	*c*,*g*
e	*d*,*f*
f	*d*,*g*
g	*c*

д

Рис. 6.1. Ориентированный граф и его представления:
а – орграф; *б* – матрица смежности; *в* – матрица инциденций;
г – список ребер; *д* – структура смежности.

Если машинное слово имеет длину l двоичных разрядов, то каждая строка матрицы требует $\lceil |V|/l \rceil$ слов. Если каждая строка начинается с нового слова, то для хранения матрицы требуется $|V| \cdot \lceil |V|/l \rceil$ слов. Поскольку у неориентированного графа матрица

смежности симметрична, то для ее представления достаточно хранить только верхний или нижний треугольник. В результате экономится почти 50% памяти, однако время вычислений может при этом увеличиться, так как каждое обращение к a_{ij} должно быть заменено (для верхнего треугольника) следующим:

if $i > j$ **then** a_{ji} **else** a_{ij}.

Большинство алгоритмов, использующих представление графа его матрицей смежности, требуют времени $O(|V|^2)$. Даже начальное заполнение матрицы требует времени $O(|V|^2)$.

Матрица инциденций. Матрица инциденций графа $G = (V, E)$ есть матрица $B = [b_{ij}]$ размера $|V| \times |E|$ (строки соответствуют вершинам графа, а столбцы – ребрам). Для ориентированного графа $b_{ij} = 1$, если дуга e_j инцидентна вершине v_i и исходит из нее; $b_{ij} = -1$, если дуга e_j инцидентна вершине v_i и заходит в нее; $b_{ij} = 0$, если дуга e_j не инцидентна вершине v_i. Если имеется петля, т. е. дуга e_j вида (v_i, v_i), то для обозначения b_{ij} используется какой-нибудь дополнительный символ (например, 2). В случае неориентированного графа $b_{ij} = 1$, если ребро e_j инцидентно вершине v_i; $b_{ij} = 0$ в противном случае. Матрица инциденций ориентированного графа (рис. 6.1*а*) представлена на рис. 6.1*в*. Очевидно, что всякий столбец матрицы содержит точно два ненулевых элемента.

С алгоритмической точки зрения матрица инциденций является одним из худших способов представления графа. Это связано с тем, что требуется $|V| \cdot |E|$ ячеек памяти, причем большинство этих ячеек занято нулями. Неудобен также доступ к информации, например для определения, существует ли дуга (v_i, v_j), требуется в худшем случае перебор всех столбцов матрицы, т. е. $|E|$ шагов.

Матрица весов. Матрица весов используется для представления взвешенного графа, т. е. графа, в котором ребру (i, j) сопоставлено число w_{ij}, называемое весом ребра. Матрица весов есть матрица $W = [w_{ij}]$, где w_{ij} – вес ребра, соединяющего вершины i и j. Веса несуществующих ребер обычно полагают равными ∞ или 0 в зависимости от приложений. Когда вес несуществующего ребра равен 0, матрица весов является простым обобщением матрицы смежности.

Список ребер. Для разреженных графов (когда $|E|$ меньше $|V|^2$) более экономичным в отношении памяти может оказаться

метод представления графа списком ребер, где каждое ребро представляется парой вершин. Список ребер можно реализовать двумя массивами: $g = (g_1, g_2, \ldots, g_{|E|})$ и $h = (h_1, h_2, \ldots, h_{|E|})$. Каждый элемент в массивах есть метка вершины, а i-е ребро графа выходит из вершины g_i и входит в вершину h_i. Список ребер ориентированного графа (рис. 6.1*а*) представлен на рисунке 6.1*г*. Ясно, что объем памяти в этом случае составляет порядка $2|E|$.

Неудобством такого представления является большое число шагов (порядка $|E|$ в худшем случае), необходимое для получения множества вершин, к которым ведут ребра из данной вершины. Ситуацию можно значительно улучшить, упорядочив множество пар лексикографически и применяя бинарный поиск.

Структура смежности. При представлении графа структурой смежности каждой вершине $v \in V$ сопоставляется Adj(v) – список всех вершин, смежных с вершиной v (список смежности). В большинстве алгоритмов относительный порядок вершин в Adj(v) не важен, поэтому Adj(v) удобно считать мультимножеством (множеством, если граф простой) вершин, смежных с v. Структура смежности ориентированного графа на рисунке 6.1*а* изображена на рисунке 6.1*д*.

Если для хранения метки вершины использовать одно машинное слово, то структура смежности ориентированного графа требует порядка $|V| + |E|$ слов. Если граф неориентированный, нужно порядка $|V| + 2|E|$ слов, так как каждое ребро встречается дважды. Многие алгоритмы, использующие представление графа структурой смежности, требуют время вычислений $O(|V| + |E|)$.

В простейшем случае структура смежности может быть удобно реализована массивом из $|V|$ односвязных списков, где каждый список содержит смежные вершины. Поле данных содержит метку одной из смежных вершин, а поле указателя указывает следующую смежную вершину.

Во многих задачах на графах выбор представления является решающим для эффективности алгоритмов. Переход от одного представления к другому относительно прост и может быть выполнен за $O(|V|^2)$ операций. Поэтому если решение задачи на графе обязательно требует числа операций, по крайней мере пропорционального $|V|^2$, то время ее решения не зависит от представления графа, так как оно может быть изменено за $O(|V|^2)$ операций.

6.2. Поиск в глубину

Поиск в глубину (*англ*. Depth First Search) является одним из методов систематического исследования всех вершин и ребер графа. Он исходит из процедуры прохождения графа методом поиска с возвратом и является основой многих алгоритмов на графах, как ориентированных, так и неориентированных.

Общая идея метода состоит в следующем. Поиск начинается с посещения некоторой вершины $v_0 \in V$. Пусть v – последняя посещенная вершина. Выбирается произвольное ребро (v, w), инцидентное v. Если вершина w уже пройдена (посещалась ранее), осуществляется возврат в вершину v и выбирается другое ребро. Если вершина w еще не пройдена, то она посещается и процесс применяется рекурсивно к вершине w. Если все ребра, инцидентные вершине v, уже исследованы, осуществляется возврат назад по ребру (u, v), по которому пришли в вершину v, и продолжается исследование ребер, инцидентных u. Процесс заканчивается, когда делается попытка вернуться назад из вершины v_0, с которой начиналось исследование.

Рассмотренная процедура представлена алгоритмом 6.1. Данный алгоритм осуществляет поиск в глубину в произвольном, необязательно связном графе, заданном структурой смежности. Каждой вершине $v \in V$ графа поставлен в соответствие элемент $new(v)$, чтобы отличить уже пройденные вершины графа ($new(v)$ = **false**) от еще не пройденных ($new(v)$ = **true**).

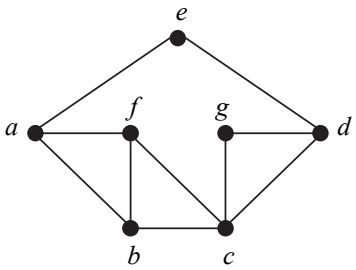
```
for x∈V do new(x) ← true   // инициализация
for x∈V do if new(x) then DFS(x)

procedure DFS(v)
   visit(v)   // посещение вершины v
   new(v) ← false
   for w∈Adj(v) do if new(w) then DFS(w)
return
```

Алгоритм 6.1. Поиск в глубину

Для неориентированного графа, представленного на рисунке 6.2, заданного структурой смежности, поиск в глубину дает следующий порядок посещения вершин: a, b, c, d, e, g, f. Очевид-

но, что порядок прохождения вершин графа не единственен, так как ребра, инцидентные вершине, могут выбираться для рассмотрения в произвольном порядке.



v	Adj(v)
a	b, e, f
b	a, c, f
c	b, d, f, g
d	c, e, g
e	a, d
f	a, b, c
g	c, d

Рис. 6.2. Неориентированный граф и его структура смежности

Поскольку для каждой вершины, которая проходится впервые, производится обращение к *DFS* ровно один раз, всего требуется $|V|$ обращений к *DFS*. При каждом обращении количество производимых действий пропорционально числу ребер, инцидентных рассматриваемой вершине. Поэтому время поиска в глубину в произвольном графе равно $O(|V| + |E|)$.

Методика поиска в глубину очевидным образом переносится на ориентированные графы и не требует никаких модификаций. Необходимо только учитывать направленность ребер (дуг).

6.3. Поиск в ширину

Поиск в ширину (*англ.* Breadth First Search) является другим методом систематического исследования всех вершин и ребер графа. В отличие от поиска в глубину, где для реализации процесса используется стек (неявно в рекурсивном алгоритме и явно в нерекурсивном), при поиске в ширину используется очередь. Процесс поиска начинается в произвольно выбранной вершине v графа, проходятся все ребра, инцидентные v. Пусть эти ребра будут (v, a_1), (v, a_2), …, (v, a_k). Затем исследуются ребра, инцидентные $a_1, a_2, \ldots, a_k$. Пусть эти ребра будут $(a_1, a_{1,1})$, $(a_1, a_{1,2})$, …, $(a_1, a_{1,m})$, $(a_2, a_{2,1})$, $(a_2, a_{2,2})$, …, $(a_2, a_{2,n})$, …, $(a_k, a_{k,t})$. Затем исследуются ребра, инцидентные $a_{1,1}, a_{1,2}, \ldots, a_{k,t}$, и т. д. Этот процесс продолжается до тех пор, пока не будут исследованы все ребра. Рассмотренная процедура поиска в ширину представлена алгоритмом 6.2.

for $x \in V$ **do** $new(x) \leftarrow$ **true** // инициализация
for $x \in V$ **do if** $new(x)$ **then** $BFS(x)$

procedure $BFS(v)$
$Q \leftarrow \varnothing$ // очередь Q пуста
$Q \Leftarrow v$
$new(v) \leftarrow$ **false**
while $Q \neq \varnothing$ **do** {
$v \Leftarrow Q$
visit(v) // посещение вершины v
for $w \in$ Adj(v) **do**
if $new(w)$ **then** {
$Q \Leftarrow w$
$new(w) \leftarrow$ **false**
}
}
return

Алгоритм 6.2. Поиск в ширину

Данный алгоритм осуществляет поиск в ширину в произвольном, необязательно связном графе, заданном структурой смежности. Каждой вершине $v \in V$ графа поставлен в соответствие элемент $new(v)$, чтобы отличить уже пройденные вершины от еще не пройденных. Для неориентированного графа (рис. 6.2) поиск в ширину дает следующий порядок посещения вершин: *a*, *b*, *e*, *f*, *c*, *d*, *g*. Как и при поиске в глубину, порядок посещения вершин не единственен, так как инцидентные ребра могут выбираться в произвольном порядке.

Вычислительная сложность алгоритма равна $O(|V| + |E|)$, так как каждая вершина помещается в очередь и удаляется из очереди в точности один раз, а число итераций в цикле **for** имеет порядок числа ребер графа.

Как и в случае поиска в глубину, процедуру поиска в ширину без всяких модификаций можно использовать и для ориентированных графов, учитывая направленность ребер (дуг).

6.4. Остовные деревья

Остовное дерево (остов, каркас, стягивающее дерево) произвольного неориентированного связного графа $G = (V, E)$ есть суграф графа G, представляющий собой дерево $T = (V, E_T)$, где $E_T \subseteq E$, т. е. подграф, содержащий все вершины графа G. Если граф G несвязный, то множество, состоящее из остовных деревь-

ев каждой связной компоненты, называется остовным лесом. Каждое дерево с n вершинами имеет в точности $n - 1$ ребер. Поэтому деревья можно считать минимально связными графами. Удаление любого ребра преобразует дерево в несвязный граф.

Для построения остовного дерева (леса) неориентированного графа G последовательно просматриваются ребра графа, оставляя те, которые не образуют циклов с уже выбранными. Очевидно, что для графа G можно построить множество остовных деревьев. Причем с увеличением числа вершин число остовных деревьев растет экспоненциально (полный граф с n вершинами имеет n^{n-2} остовных деревьев). Простейшим способом построения остовных деревьев является использование процедур поиска в глубину (*DFS*-дерево) и ширину (*BFS*-дерево).

6.4.1. DFS-дерево

DFS-дерево – это остовное дерево $T = (V, E_T)$, которое получается в результате поиска в глубину по неориентированному графу $G = (V, E)$, $E_T \subseteq E$. При этом ребра графа разбиваются на два множества: E_T – множество *ребер дерева* и E_B – множество ребер, не вошедших в остовное дерево, называемых *обратными ребрами*, так как они ведут назад в пройденные ранее вершины. Поиск в глубину вводит ориентацию на ребра графа G в соответствии с направлением прохождения, т. е. получается ориентированное остовное дерево. Если в *DFS*-дереве имеется путь из вершины v в вершину w, то v – предок w, а w – потомок v. Поскольку инцидентные вершине ребра графа могут выбираться при поиске в глубину в произвольном порядке, *DFS*-дерево для заданного графа не единственное. Построение остовного дерева способом поиска в глубину представлено алгоритмом 6.3.

В алгоритме каждой вершине $v \in V$ сопоставлен элемент $num(v)$. Эти элементы служат для постепенной нумерации вершин числами от 1 до $|V|$ по мере их прохождения. Начальное значение $num(v) = 0$ для всех $v \in V$ показывает, что ни одна вершина не пройдена. Когда вершина v посещается в первый раз, $num(v)$ присваивается ненулевое значение. Рекурсивная процедура $DFST(v, u)$ реализует поиск в глубину в графе $G = (V, E)$, содержащем v, и строит *DFS*-дерево $T = (V, E_T)$, $E_T \subseteq E$; вершина u является отцом вершины v в дереве. Граф представляется

структурой смежности. Элементы множества E_T – ребра дерева, а элементы множества E_B – обратные ребра. Если граф G несвязный, то T будет остовным лесом. Вершина, с которой начинается поиск, считается корнем соответствующего дерева.

for $x \in V$ **do** $num(x) \leftarrow 0$ // инициализация
$i \leftarrow 0$
$E_T \leftarrow \varnothing$ // E_T – множество ребер
$E_B \leftarrow \varnothing$ // E_B – множество обратных ребер
for $x \in V$ **do if** $num(x) = 0$ **then** $DFST(x, 0)$

procedure $DFST(v, u)$
  $i \leftarrow i + 1$
  $num(v) \leftarrow i$
  for $w \in$ Adj(v) **do**
    if $num(w) = 0$
      then { // (v, w) – ребро дерева; $E_T \leftarrow E_T \cup \{(v, w)\}$; $DFST(w, v)$
      else if $num(w) < num(v)$ **and** $w \neq u$
        then { // (v, w) – обратное ребро; $E_B \leftarrow E_B \cup \{(v, w)\}$
return

Алгоритм 6.3. Построение остовного дерева поиском в глубину

DFS-дерево для неориентированного графа (см. рис. 6.2) представлено на рисунке 6.3*а*. Ребра дерева изображены сплошными линиями, обратные ребра – пунктирными. Числа около ребер указывают порядок включения ребер в множества E_T и E_B.

Необходимо обратить внимание, что если $num(w) = 0$, то (v, w) – ребро дерева. Если же $num(w) \neq 0$, то условием того, что (v, w) будет обратным ребром, является соотношение $num(w) < num(v)$ и $w \neq u$. Таким образом, нумерация вершин важна для выделения обратных ребер. В случае, если не требуется явного формирования множества E_B обратных ребер, нумерация вершин не требуется, достаточно только отличать уже пройденные вершины от еще не пройденных. Поэтому вместо $num(v)$ можно использовать элементы типа $new(v)$, как в алгоритме поиска в глубину (см. алгоритм 6.1). Нет необходимости также в хранении в элементе u отца вершины v. Таким образом, более

простая версия алгоритма будет формировать только множество E_T ребер дерева.

Временная сложность алгоритма определяется сложностью поиска в глубину, т. е. $O(|V| + |E|)$.

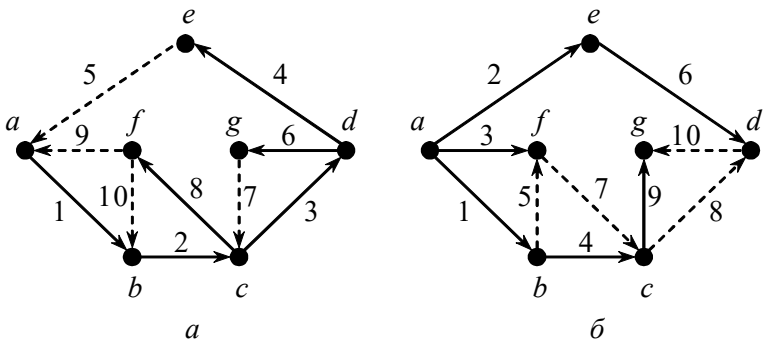


Рис. 6.3. Остовные деревья для графа на рисунке 6.2:
а – DFS-дерево; *б – BFS*-дерево.

6.4.2. BFS-дерево

BFS-дерево – это остовное дерево $T = (V, E_T)$, которое получается в результате поиска в ширину по неориентированному графу $G = (V, E)$, $E_T \subseteq E$. При этом ребра графа разбиваются на два множества: E_T – множество *ребер дерева* и E_C – множество ребер, не вошедших в остовное дерево, которые можно назвать *поперечными ребрами*, так как они соединяют вершины, не являющиеся в дереве ни предками, ни потомками друг друга. Поиск в ширину вводит ориентацию на ребра графа G в соответствии с направлением прохождения, т. е. получается ориентированное остовное дерево. Если граф $G = (V, E)$ несвязный, то $T = (V, E_T)$ будет остовным лесом. Построение *BFS*-дерева способом поиска в ширину представлено алгоритмом 6.4.

Критерием распознавания поперечных ребер является соотношение $num(w) > num(v)$. Если $num(w) < num(v)$, это означает, что все инцидентные вершине w ребра уже исследованы и классифицированы.

BFS-дерево для неориентированного графа (см. рис. 6.2) представлено на рисунке 6.3*б*.

```
for x ∈ V do num(x) ← 0   // инициализация
i ← 0
E_T ← ∅   // E_T – множество ребер дерева
E_C ← ∅   // E_C – множество поперечных ребер
for x ∈ V do if num(x) = 0 then BFST(x)

procedure BFST(u)
   Q ← ∅   // очередь Q пуста
   Q ⇐ u
   i ← i + 1
   num(u) ← i
   while Q ≠ ∅ do
      v ⇐ Q
      for w ∈ Adj(v) do
         if num(w) = 0
            then // (v, w) – ребро дерева
                 Q ⇐ w
                 i ← i + 1
                 num(w) ← i
                 E_T ← E_T ∪ {(v, w)}
         else if num(w) > num(v)
                 then // (v, w) – поперечное ребро
                      E_C ← E_C ∪ {(v, w)}
return
```

Алгоритм 6.4. Построение остовного дерева поиском в ширину

Ребра дерева изображены сплошными линиями, поперечные ребра – пунктирными. Числа около ребер дерева указывают порядок включения ребер в множества E_T и E_C.

Временная сложность алгоритма определяется сложностью поиска в ширину, т. е. $O(|V| + |E|)$.

6.4.3. Минимальное остовное дерево

Для взвешенного графа часто требуется определить остовное дерево (лес) с минимальным общим весом ребер. Остовное дерево (лес), у которого сумма весов всех его ребер минимальна, называется *минимальным остовным деревом*. Рассмотрим два достаточно популярных и эффективных алгоритма построения минимального остовного дерева.

Алгоритм Крускала

Алгоритм Крускала (Kruskal) известен как жадный алгоритм. Идея алгоритма проста и заключается в следующем. На каждом шаге выбирается новое ребро с наименьшим весом, не образующее циклов с ранее выбранными ребрами. Процесс продолжается до тех пор, пока не будет выбрано $|V| - 1$ ребер, образующих остовное дерево.

Вариант реализации алгоритма Крускала, который строит остовное дерево $T = (V, E_T)$ с минимальным общим весом W_T для взвешенного неориентированного графа $G = (V, E)$, $E_T \subseteq E$, представлен алгоритмом 6.5.

$W_T \leftarrow 0$ // сумма весов ребер остовного дерева
$E_T \leftarrow \varnothing$ // множество ребер остовного дерева
$VS \leftarrow \{\{v\} : v \in V\}$
Упорядочить список ребер E по неубыванию весов
while $|VS| > 1$
do
- Выбрать из E ребро (v, u) с наименьшим весом
- Удалить (v, u) из E
- **if** $v \in A, u \in B, A, B \in VS, A \neq B$
- **then**
  - // v и u принадлежат различным подмно-
  - // жествам A и B из VS. Заменить в VS
  - // подмножества A и B на $A \cup B$
  - $VS \leftarrow (VS - \{A, B\}) \cup \{A \cup B\}$
  - $E_T \leftarrow E_T \cup \{(v, u)\}$
  - $W_T \leftarrow W_T + w_{vu}$ // w_{vu} – вес ребра (v, u)

Алгоритм 6.5. Алгоритм Крускала

Основой работы алгоритма является множество VS, которое представляет собой разбиение множества вершин V на непересекающиеся подмножества, т. е. на связные компоненты формируемого остовного дерева, определяемые текущим содержимым множества E_T ребер дерева. Очевидно, что вначале, когда еще ни одно ребро не включено в E_T, VS содержит $|V|$ одноэлементных подмножеств, каждое из которых представляет собой вершину $v \in V$. Ребра выбираются из E в порядке возрастания стоимости и

рассматриваются по очереди. Ребро (v, u), анализируемое в условном операторе **if**, добавляется в E_T тогда и только тогда, когда v и u принадлежат разным подмножествам A и B из VS, в противном случае его добавление вызвало бы появление цикла в соответствующей связной компоненте. Добавление ребра в E_T означает, что оно соединяет две связные компоненты в одну. Поэтому в VS необходимо заменить подмножества A и B на их объединение $A \cup B$. Алгоритм завершает работу, когда VS будет содержать точно одно множество, содержащее все вершины из V. Это означает, что все вершины входят в одну связную компоненту, не имеющую цикла, т. е. завершено построение остовного дерева. Следует обратить внимание на то, что к моменту завершения построения остовного дерева могут остаться неисследованные ребра графа.

Для иллюстрации работы алгоритма рассмотрим построение минимального остовного дерева для неориентированного взвешенного графа (рис. 6.4*а*). Числа рядом с ребрами указывают их веса.

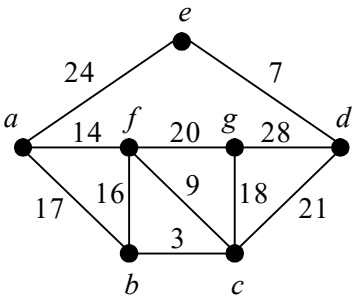
Перечислим ребра в порядке возрастания их весов.

Ребро:	(b,c)	(d,e)	(c,f)	(a,f)	(b,f)	(a,b)	(c,g)	(f,g)	(c,d)	(a,e)	(d,g)
Вес:	3	7	9	14	16	17	18	20	21	24	28

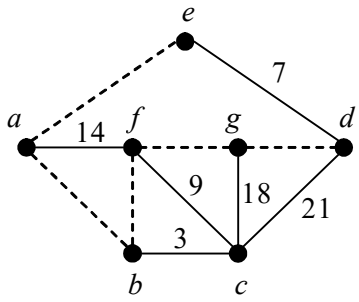
Полученное минимальное остовное дерево с минимальным общим весом ребер $W_T = 72$ представлено на рисунке 6.4*б*; последовательность шагов построения остовного дерева – на рисунке 6.4*в*.

Как видно из примера, ребра (a, e) и (d, g) алгоритмом не исследованы, так как включением в E_T дуги (c, d) завершилось построение остовного дерева. Данный факт позволяет сделать некоторые рекомендации по выбору методов сортировки ребер.

Во-первых, поскольку сортировка существенно влияет на эффективность алгоритма в целом, необходимо применять наиболее эффективные методы сортировки. Во-вторых, поскольку в общем случае не требуется полная сортировка ребер, лучше использовать сортировку, основанную на выборе, которая позволяет остановить сортировку после выполнения k шагов, расположив первые k элементов в их окончательные позиции.



а



б

Ребро	Вес	Действие	Множества в *VS*
			{*a*}, {*b*}, {*c*}, {*d*}, {*e*}, {*f*}, {*g*}
(*b*, *c*)	3	Добавить	{*a*}, {*b*, *c*}, {*d*}, {*e*}, {*f*}, {*g*}
(*d*, *e*)	7	Добавить	{*a*}, {*b*, *c*}, {*d*, *e*}, {*f*}, {*g*}
(*c*, *f*)	9	Добавить	{*a*}, {*b*, *c*, *f*}, {*d*, *e*}, {*g*}
(*a*, *f*)	14	Добавить	{*a*, *b*, *c*, *f*}, {*d*, *e*}, {*g*}
(*b*, *f*)	16	Отвергнуть	
(*a*, *b*)	17	Отвергнуть	
(*c*, *g*)	18	Добавить	{*a*, *b*, *c*, *f*, *g*}, {*d*, *e*}
(*f*, *g*)	20	Отвергнуть	
(*c*, *d*)	21	Добавить	{*a*, *b*, *c*, *d*, *e*, *f*, *g*}

в

Рис. 6.4. Построение остовного дерева алгоритмом Крускала:
а – исходный неориентированный граф; *б* – минимальное остовное дерево;
в – последовательность шагов построения минимального остовного дерева.

С рассмотренных позиций наиболее подходящей является пирамидальная сортировка. При этом необходимо учесть, что реально ребра не сортируются, а хранятся в виде пирамиды, корнем которой является ребро с наименьшим весом. Другими словами, список ребер графа представляется как очередь с приоритетами (приоритетом является вес ребра), одним из идеальных способов реализации которой является пирамида. Выбор ребра с наименьшей стоимостью (корня пирамиды) и его исключение из очереди предполагает восстановление пирамиды. Возможно применение и других структур данных для эффективной организации очередей с приоритетами, например 2–3-деревья.

Другим существенным фактором, влияющим на сложность алгоритма, является способ представления множества *VS* и спо-

соб слияния его подмножеств. Можно воспользоваться следующим методом. Каждое подмножество множества *VS* представляется ориентированным деревом, вершинам которого сопоставлены элементы подмножества. Из каждой вершины существует путь до корня (имеется указатель отца). Корень идентифицирует подмножество, т. е. элемент, сопоставленный корню, есть имя подмножества. Каждая вершина содержит информацию о размере поддерева (число вершин в поддереве), корнем которого она является. Таким образом, разбиение *VS* представляется в виде леса, состоящего из деревьев, соответствующих подмножествам из *VS*. Например, лес для разбиения $VS = \{\{a\},\{b, c, d, e\},\{f, g\}\}$ показан на рисунке 6.5*а*; именами подмножеств являются соответственно элементы *a*, *c* и *f*.

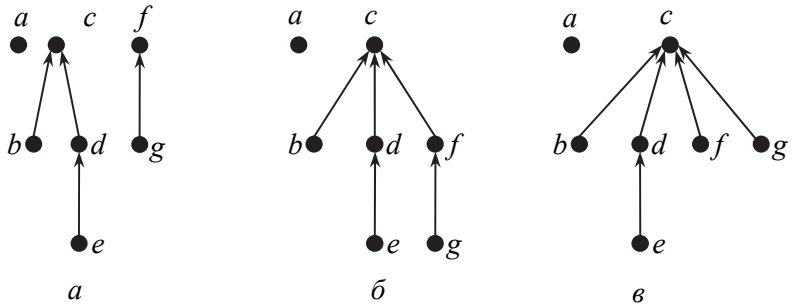


Рис. 6.5. Представление множества *VS*:
а – $VS = \{\{a\},\{b, c, d, e\},\{f, g\}\}$; *б* – $VS = \{\{a\},\{b, c, d, e, f, g\}\}$;
в – $VS = \{\{a\},\{b, c, d, e, f, g\}\}$ после сжатия пути.

Объединение подмножеств *A* и *B* происходит добавлением дуги от корня дерева, представляющего одно подмножество, к корню дерева, представляющего другое подмножество. Для большей эффективности лучше использовать *объединение с балансировкой*. Балансировка заключается в назначении корня большего по размеру дерева отцом корня меньшего дерева, т. е. имя большего подмножества становится именем подмножества, получившегося в результате объединения. Тогда высота каждого дерева будет не больше логарифма от числа его вершин (а следовательно, и от числа дуг). Результат объединения множеств $\{b, c, d, e\}$ и $\{f, g\}$ представлен на рисунке 6.5*б*; именем получившегося подмножества будет элемент *c*.

Операция поиска элемента x состоит в продвижении по указателям отцов от x до корня (до имени подмножества). Для повышения эффективности поиска можно использовать операцию *сжатия пути*, которая заключается в следующем. После завершения выполнения поиска элемента x все вершины на пути между x и корнем (включая и вершину x) становятся сыновьями корня. Например, если выполнить операцию поиска элемента g на лесе, показанном на рисунке 6.5*б*, то в качестве имени подмножества, содержащего g, определится элемент c. Сжатие пути приведет к изменению леса, который будет иметь вид, как на рисунке 6.5*в*. Сжатие пути незначительно увеличивает сложность поиска, но дает существенный выигрыш во времени при достаточно большом числе операций поиска.

Пусть каждый элемент x множества представляется узлом, состоящим из поля *info*, в котором хранится элемент x множества, поля связи *father*, указывающего на отца узла, и поля *size*, в котором содержится число узлов в поддереве с корнем x. Кроме того, для обеспечения прямого доступа к элементам множества (т. е. к узлам леса) имеется набор внешних указателей l_x, по одному для каждого элемента x множества. Тогда функцию поиска со сжатием пути $FIND(x)$, которая выдает имя подмножества, содержащего x, можно реализовать алгоритмом 6.6, а процедуру объединения $UNION(x, y)$, образующую новое подмножество, содержащее все элементы подмножеств с именами x и y, – алгоритмом 6.7.

function $FIND(x)$
  // x – узел леса, записанный в ячейку l_x
  $S \leftarrow \varnothing$ // стек S пуст
  $t \leftarrow l_x$
  while $father(t) \neq \Lambda$ **do** $\begin{cases} S \Leftarrow t \\ t \leftarrow father(t) \end{cases}$
  while $S \neq \varnothing$ **do** $\begin{cases} v \Leftarrow S \\ father(v) \leftarrow t \\ size(v) \leftarrow 1 \end{cases}$
  $FIND \leftarrow t$
return

Алгоритм 6.6. Функция поиска со сжатием пути

procedure $UNION(x, y)$
// x и y – корни деревьев, записанные в ячейках l_x и l_y
if $size(l_x) < size(l_y)$

then $\begin{cases} father(l_x) \leftarrow l_y \\ size(l_y) \leftarrow size(l_x) + size(l_y) \end{cases}$

else $\begin{cases} father(l_y) \leftarrow l_x \\ size(l_x) \leftarrow size(l_y) + size(l_x) \end{cases}$

return

Алгоритм 6.7. Процедура объединения подмножеств с балансировкой

Таким образом, в алгоритме Крускала после выбора ребра (v, u) необходимо выполнить следующую последовательность операций:

$x \leftarrow FIND(v)$
$y \leftarrow FIND(u)$
if $x \neq y$ **then** $UNION(x, y)$.

Эффективность алгоритма Крускала зависит от организации выбора ребра с наименьшим весом, способа представления разбиения VS и способа объединения подмножеств. Построение очереди с приоритетами на основе пирамиды требует $O(|E|)$ операций, восстановление пирамиды после выбора и исключения ребра с наименьшей стоимостью – $O(\log|E|)$ операций. Поскольку число операций выбора и исключения ребра из очереди пропорционально $|E|$, то общее число операций составит $O(|E| \log|E|)$. Нахождение подмножеств A и B требует, как уже отмечалось, $O(\log|E|)$ операций. Объединение подмножеств занимает постоянное время, не зависящее от их размеров. Общее число нахождения подмножеств и их объединений пропорционально числу ребер, т. е. требуется $O(|E| \log|E|)$ операций. Первоначальное разбиение VS на одноэлементные подмножества занимает время $O(|V|)$. Таким образом, временная сложность алгоритма Крускала составляет $O(|E| \log|E|)$.

Алгоритм Дейкстры – Прима

Алгоритм Дейкстры – Прима (Dijkstra, Prim) известен как алгоритм ближайшего соседа. Этот алгоритм, в отличие от алгоритма Крускала, не требует решать задачи ни сортировки ребер,

ни проверки на цикличность на каждом шаге. Алгоритм начинает построение минимального остовного дерева с некоторой произвольной вершины v графа $G = (V, E)$. Выбирается ребро (v, w), инцидентное вершине v, с наименьшим весом и включается в дерево. Затем среди инцидентных либо v, либо w выбирается ребро с наименьшим весом и включается в частично построенное дерево. В результате в дерево добавляется новая вершина, например u. Ищется наименьшее ребро, соединяющее v, w или u с некоторой другой вершиной графа. Процесс продолжается до тех пор, пока все вершины из G не будут включены в дерево, т. е. пока дерево не станет остовным.

Очевидно, что основным фактором, влияющим на эффективность алгоритма, является поиск ребер с наименьшим весом, инцидентных всем вершинам, включенным в частично построенное дерево. Худшим является случай, когда G – полный граф, т. е. каждая пара вершин графа соединена ребром. В этом случае для поиска ближайшего соседа на каждом шаге необходимо сделать максимальное число сравнений. Для выбора первого ребра нужно сравнить веса всех $|V| - 1$ ребер, инцидентных вершине v, т. е. требуется $|V| - 2$ сравнений. Для выбора второго ребра ищется наименьшее из возможных $2(|V| - 2)$ ребер, инцидентных v или w, что требует $2(|V| - 2) - 1$ сравнений. Таким образом, для выбора i-го ребра нужно $i(|V| - i) - 1$ сравнений, а общее число сравнений для построения остовного дерева будет составлять

$$\sum_{i=1}^{|V|-1}[i(|V|-1)-1] = O(|V|^3).$$

Для уменьшения числа сравнений можно каждой вершине v, еще не принадлежащей дереву, сопоставить указатель вершины дерева, ближайшей к v. Имея такую информацию, i-ю добавляемую вершину можно найти за $|V| - i$ сравнений при $i \geq 2$. Первая вершина выбирается произвольно и не требует сравнений. После добавления к дереву i-й вершины информацию о ближайших вершинах можно скорректировать за $|V| - i$ операций, сравнивая для каждой вершины v (из $|V| - i$ вершин), не принадлежащей дереву, расстояния от нее до только что добавленной и до той, которая перед этим была в дереве ближайшей к ней. Поэтому для $i \geq 2$ добавление к дереву i-й вершины требует $2(|V| - i)$ сравнений, а общее число сравнений

$$\sum_{i=1}^{|V|} 2(|V|-i) = (|V|-1)(|V|-2) = O(|V|^2).$$

Процедура построения минимального остовного дерева $T = (V, E_T)$ с общим весом W_T для графа $G = (V, E)$, заданного матрицей весов $W = [w_{ij}]$, использующей рассмотренную технику, представлена алгоритмом 6.8 (в матрице W веса всех несуществующих ребер равны ∞).

Выбрать произвольно v

for $u \in V$ **do** $\begin{cases} dist(u) \leftarrow w_{uv} \\ near(u) \leftarrow v \end{cases}$

$V_T \leftarrow \{v\}$ // множество вершин остовного дерева

$E_T \leftarrow \varnothing$ // множество ребер остовного дерева

$W_T \leftarrow 0$ // сумма весов ребер остовного дерева

while $V_T \neq V$ **do** $\begin{cases} \text{Выбрать из } V - V_T \text{ вершину } v \\ \quad \text{с наименьшим значением } dist(v) \\ V_T \leftarrow V_T \cup \{v\} \\ E_T \leftarrow E_T \cup \{(v, near(v))\} \\ W_T \leftarrow W_T + dist(v) \\ \textbf{for } x \in V - V_T \textbf{ do} \\ \quad \textbf{if } dist(x) > w_{vx} \textbf{ then } \begin{cases} dist(x) \leftarrow w_{vx} \\ near(x) \leftarrow v \end{cases} \end{cases}$

Алгоритм 6.8. Алгоритм Дейкстры – Прима

В алгоритме массивы $dist(u)$ для расстояний (весов) и $near(u)$ для названий вершин используются для записи текущей ближайшей вершины в дереве для каждой из вершин, еще не попавших в него. На i-м шаге цикла **while** множество $V - V_T$ содержит $|V| - i$ элементов. Поэтому поиск вершины v с наименьшим значением $dist(v)$ требует $|V| - i - 1$ сравнений. Операторы присваивания для V_T, E_T и W_T требуют фиксированного числа операций. Цикл **for** выполняет $|V| - i - 1$ сравнений $dist(x) > w_{vx}$. Поэтому общее число сравнений равно $O(|V|^2)$.

6.4.4. Остовные деревья ориентированных графов

Очевидно, остовное дерево (лес) орграфа является ориентированным. Рассмотренные выше алгоритмы построения остовных деревьев неориентированных графов могут находить ориентированные остовные деревья и для орграфов. Главное отличие заключается только в том, что ребра (дуги) орграфа проходятся в соответствии с их ориентацией.

Задача усложняется, если необходимо не только построить остовное дерево, но и разбить множество ребер орграфа на классы. Если поиск в глубину в неориентированном графе разбивает его ребра на два класса (ребра остовного дерева и обратные ребра), то в орграфе ребра (дуги) разбиваются на четыре класса. Непросмотренное ребро (v, w), встречающееся, когда процесс поиска в глубину находится в вершине v, можно классифицировать следующим образом.

1. Вершина w еще не пройдена. В этом случае (v, w) – *ребро дерева*, идущее к новой вершине.

2. Вершина w уже была пройдена:

а) если w потомок v в *DFS*-дереве (а не в орграфе), то ребро (v, w) называется *прямым ребром*, т. е. прямые ребра идут от предков к потомкам, но не являются ребрами дерева;

б) если w предок v в *DFS*-дереве, то ребро (v, w) называется *обратным ребром*, т. е. обратные ребра идут от потомков к предкам;

в) если v и w не являются ни предками, ни потомками друг друга (несоотносимы) в *DFS*-дереве, то ребро (v, w) называется *поперечным ребром*.

Как и в случае построения *DFS*-дерева для неориентированного графа, чтобы отличать разные классы ребер, необходимо использовать нумерацию исследуемых вершин с помощью элементов $num(v)$. Ребро (v, w) при $num(w) > num(v)$ является либо ребром дерева, либо прямым ребром. В процессе поиска в глубину различить эти ребра просто – ребро дерева всегда ведет к новой вершине. Ребро (v, w) при $num(w) < num(v)$ является либо обратным, либо поперечным ребром. Критерий их отличия основан на следующем свойстве поиска в глубину: если к моменту рассмотрения ребра (v, w) $num(w) < num(v)$, но еще не все исходящие из w ребра исследованы (просканированы), то это означа-

ет, что w является корнем поддерева, которое содержит вершину v, т. е. v – потомок w. Таким образом, ребро (v, w) при $num(w) < num(v)$ является обратным ребром, если вершина w еще не полностью просканирована к моменту анализа ребра (v, w), в противном случае – поперечным ребром.

Необходимо отметить еще одну особенность *DFS*-дерева для ориентированных графов. Она заключается в том, что в общем случае, даже если орграф связен, его *DFS*-дерево может быть несвязным, т. е. *DFS*-лесом.

Орграф, заданный структурой смежности, и построенный для него *DFS*-лес представлены на рисунке 6.6. Ребра дерева изображены сплошными линиями, а прочие ребра – штриховыми. Числа около ребер показывают порядок исследования ребер по заданной структуре смежности. Как видно из рисунка, для связного орграфа полученный *DFS*-лес несвязный и состоит из двух деревьев: дерева с вершинами a, b, c, d, e (корень a) и дерева с вершинами f и g (корень f). Ребро (b, a) обратное, ребро (a, e) прямое, ребра (d, c), (f, b), (f, c), (g, d) и (g, e) – поперечные.

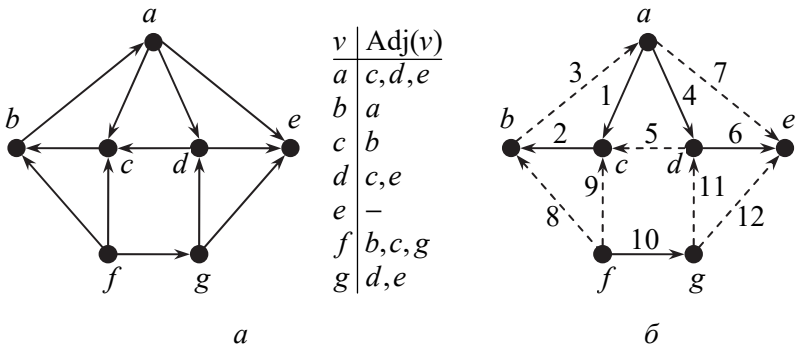


Рис. 6.6. Остовный лес орграфа:
а – орграф и его структура смежности; *б* – *DFS*-лес орграфа.

Процесс построения *DFS*-дерева (леса) $T = (V, E_T)$ для ориентированного графа $G = (V, E)$ с разбиением ребер графа на четыре класса представлен алгоритмом 6.9. В отличие от алгоритма 6.3 построения *DFS*-дерева для неориентированного графа, в него добавлены элементы $scan(v)$ для всех $v \in V$, чтобы отмечать, полностью просканирована вершина v или нет. Элементами

множеств E_F и E_C являются соответственно прямые и поперечные ребра. Очевидно, что временная сложность алгоритма равна $O(|V| + |E|)$.

```
for x ∈ V do { num(x) ← 0
             { scan(x) ← true
i ← 0
E_T ← E_B ← E_F ← E_C ← ∅
for x ∈ V do if num(x) = 0 then DFSTO(x)

procedure DFSTO(v)
   i ← i + 1
   num(v) ← i
   for w ∈ Adj(v) do
      if num(w) = 0
            then { // (v, w) – ребро дерева
                 { E_T ← E_T ∪ {(v, w)}
                 { DFSTO(w)
         else if num(w) > num(v)
                 then { // (v, w) – прямое ребро
                      { E_F ← E_F ∪ {(v, w)}
                 else if num(w) < num(v) and scan(w)
                         then { // (v, w) – обратное ребро
                              { E_B ← E_B ∪ {(v, w)}
                         else { // (v, w) – поперечное ребро
                              { E_C ← E_C ∪ {(v, w)}
   scan(v) ← false
return
```

Алгоритм 6.9. Построение *DFS*-дерева для орграфа

6.5. Связность графов

6.5.1. Связные компоненты неориентированного графа

Неориентированный граф $G = (V, E)$ называется *связным*, если все его вершины связаны между собой, т. е. существует хотя бы один путь в G между каждой парой вершин. Отношение связности определяет разбиение множества V вершин графа на непересекающиеся подмножества $V_i \subseteq V$. Вершины одного и того же множества V_i связаны друг с другом, а вершины различных мно-

жеств V_i и V_j не связаны между собой, т. е. в графе G нет ребер, связывающих вершины из разных множеств V_i и V_j. Максимальные связные подграфы $G_i = (V_i, E_i)$ графа G называются *связными компонентами* графа. Связный граф представляет собой единственную связную компоненту. Несвязный граф состоит из двух или более связных компонент.

Для отыскания связных компонент неориентированного графа легко применить технику поиска в глубину. Соответствующая модификация алгоритма поиска в глубину представлена алгоритмом 6.10. Каждой вершине v графа $G = (V, E)$ сопоставлен элемент $compnum(v)$ для присваивания общего номера связной компоненты, в которую попадает вершина v. Таким образом, алгоритм осуществляет разбиение множества V вершин графа на непересекающиеся подмножества, вершины каждого такого подмножества имеют одинаковый номер в элементах $compnum(v)$, соответствующий номеру связной компоненты. Очевидно, что этот алгоритм требует $O(|V| + |E|)$ операций.

for $x \in V$ **do** $compnum(x) \leftarrow 0$
$c \leftarrow 0$ // счетчик компонент
for $x \in V$ **do if** $compnum(x) = 0$ **then** $\begin{cases} c \leftarrow c + 1 \\ COMP(x) \end{cases}$

procedure $COMP(v)$
  $compnum(v) \leftarrow c$
  for $w \in \mathrm{Adj}(v)$ **do if** $compnum(w) = 0$ **then** $COMP(w)$
return

Алгоритм 6.10. Определение связных компонент

6.5.2. Двусвязные компоненты

Вершина v неориентированного графа $G = (V, E)$ называется *точкой сочленения*, если удаление этой вершины и всех инцидентных ей ребер ведет к разъединению оставшихся вершин, т. е. увеличивает число связных компонент графа. Граф, содержащий точку сочленения, называется *разделимым*. Связный граф без точек сочленения называется *двусвязным*. Максимальный двусвязный подграф графа называется *двусвязной компонентой,* или *блоком* этого графа. Неориентированный связный разделимый

граф и его двусвязные компоненты приведены на рисунке 6.7. Точками сочленения являются вершины *f* и *g*.

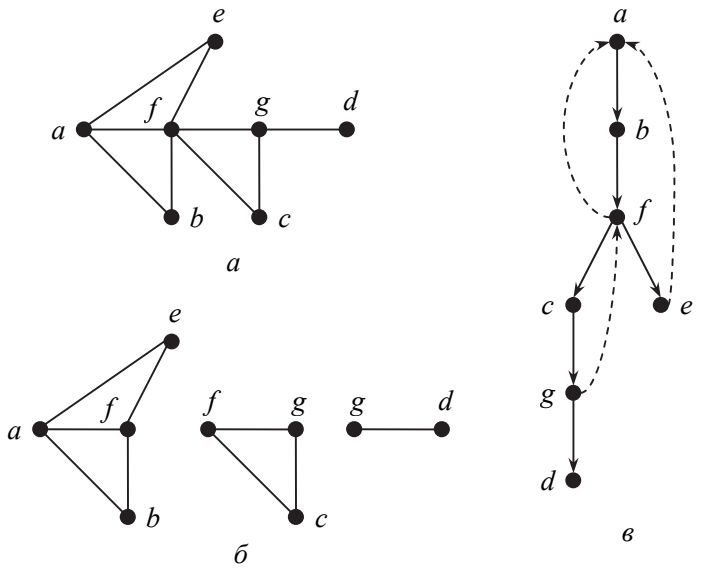


Рис. 6.7. Двусвязные компоненты неориентированного графа:
а – неориентированный граф; *б* – двусвязные компоненты графа;
в – *DFS*-дерево графа в традиционном древовидном представлении.

Следует обратить внимание на то, что если $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ – два разных блока графа G, то $V_1 \cap V_2 = \varnothing$ или $V_1 \cap V_2 = \{v\}$, где v – точка сочленения графа G. Вершина v неориентированного связного графа является точкой сочленения тогда и только тогда, когда существуют две другие вершины x и y такие, что любой путь между x и y проходит через вершину v; в этом случае удаление v и всех инцидентных ей ребер из графа G разрывает все пути между x и y, т. е. делает граф G несвязным. Нахождение точек сочленения и блоков графа является классической задачей, которая эффективно решается использованием техники поиска в глубину. Прежде всего необходимо определить критерий для распознавания точек сочленения.

Пусть $T = (V, E_T)$ – *DFS*-дерево (остовное дерево, построенное поиском в глубину) связного неориентированного графа $G = (V, E)$. Тот факт, что удаление точки сочленения v расщепля-

ет граф G по крайней мере на две части, говорит о следующем. Одна из этих частей состоит из сына вершины v и всех его потомков в *DFS*-дереве. Следовательно, в *DFS*-дереве вершина v должна иметь сына w, потомки которого (включая и w) не соединены обратными ребрами с предками вершины v. Очевидно, что корень *DFS*-дерева является точкой сочленения только тогда, когда он имеет не менее двух сыновей. Для неориентированного графа (рис. 6.7*а*) *DFS*-дерево в традиционном древовидном представлении показано на рисунке 6.7*в*. Точками сочленения являются вершины f и g. Вершина f имеет сына c, и ни из какого потомка вершины c не выходит обратное ребро к предкам вершины f. Аналогично вершина g имеет сына d, потомки которого не связаны обратными ребрами с предками вершины g.

Таким образом, можно сформулировать следующий критерий для распознавания точек сочленения: пусть $T = (V, E_T)$ – *DFS*-дерево с корнем r связного неориентированного графа $G = (V, E)$. Вершина $v \in V$ является точкой сочленения графа G тогда и только тогда, когда выполнено одно из условий:

а) $v = r$ и r имеет более одного сына;

б) $v \neq r$ и существует сын w вершины v такой, что ни w, ни какой-либо его потомок не связаны обратным ребром ни с одним предком вершины v.

Чтобы включить в процедуру поиска в глубину сформулированный критерий, для каждой вершины $v \in V$ необходимо определить два параметра: $num(v)$ и $low(v)$. Первый параметр – это номер вершины v в порядке, в котором вершины посещаются при поиске в глубину. Если E_T и E_B – множества соответственно ребер *DFS*-дерева и обратных ребер, то

$$low(v) = \min(\{num(v)\} \cup \{num(w) \mid \text{существует такое обратное ребро } (x, w) \in E_B, \text{ что } x \text{ – потомок } v, \text{ а } w \text{ – предок } v \text{ в } DFS\text{-дереве}\}),$$

т. е. $low(v)$ есть наименьшее значение $num(x)$, где x – вершина графа, в которую можно попасть из v, проходя последовательность из нуля или более ребер дерева, за которой следует не более чем одно обратное ребро. Нумерация вершин в порядке прохождения в глубину обладает тем свойством, что если x – потомок вершины v, а (x, w) – обратное ребро, причем $num(w) < num(v)$, то w – предок вершины v. Следовательно, ес-

ли v не корень, то v является точкой сочленения тогда и только тогда, когда имеет сына u, для которого $low(u) \geq num(v)$.

Переопределим $low(v)$ так, чтобы выразить через вершины, смежные с v, используя обратные ребра и значения $num(w)$ на сыновьях вершины v:

$$low(v)=\min(\{num(v)\}\cup\{low(w)\,|\,(v,w)\in E_T\}\cup\{num(w)\,|\,(v,w)\in E_B\}).$$

Тогда вычисление значения $low(v)$ предполагает следующие шаги.

1. Когда v проходится в первый раз, то $low(v)$ присваивается значение $num(v)$.

2. Когда рассматривается обратное ребро (v, w), инцидентное v, то $low(v)$ присваивается наименьшее из текущего значения $low(v)$ и $num(w)$.

3. Когда поиск в глубину возвращается к v после полного сканирования сына w этой вершины, то $low(v)$ присваивается наименьшее из текущего значения $low(v)$ и значения $low(w)$.

Необходимо обратить внимание на то, что для любой вершины v вычисление $low(v)$ заканчивается при завершении ее сканирования.

Процедура определения двусвязных компонент E_j ($j \geq 1$) неориентированного графа $G = (V, E)$ представлена алгоритмом 6.11. Для выделения ребер, принадлежащих двусвязным компонентам, используется стек S. Вначале стек пуст. По мере просмотра ребер они добавляются в стек. Пусть поиск в глубину возвращается в вершину v после полного сканирования сына w этой вершины. В этот момент завершается вычисление $low(w)$. Предположим, что $low(w) \geq num(v)$. Тогда v – точка сочленения. Ребро (v, w) вместе с ребрами, инцидентными w и его потомкам, образуют двусвязную компоненту. Эти ребра являются в точности теми ребрами, которые находятся в верхней части стека S, включая ребро (v, w). Исключение этих ребер из стека приводит к тому, что алгоритм продолжает работать с графом G', который получается из графа G удалением ребер уже выделенной двусвязной компоненты.

Для неориентированного графа (см. рис. 6.7*а*) алгоритм выделяет двусвязные компоненты в таком порядке: $E_1 = \{(g, d)\}$, $E_2 = \{(f, c),\ (c, g),\ (g, f)\}$ и $E_3 = \{(a, b),\ (b, f),\ (f, a),\ (f, e),\ (e, a)\}$. При этом получаются следующие значения элементов $low(v)$:

v	a	b	c	d	e	f	g
$low(v)$	1	1	3	6	1	1	3

.

Определим вычислительную сложность алгоритма. Поскольку он осуществляет поиск в глубину с конечным объемом дополнительной работы при прохождении каждого ребра, требуемое время выполнения равно $O(|V| + |E|)$.

```
for x ∈ V do num(x) ← 0
i ← j ← 0   // j – номер двусвязной компоненты E_j
S ← ∅   // стек S пуст
for x ∈ V do if num(x) = 0 then BICOMP(x, 0)

procedure BICOMP(v, u)
  i ← i + 1
  num(v) ← low(v) ← i
  for w ∈ Adj(v) do
    if num(w) = 0
      then { // (v, w) – ребро дерева
             S ⇐ (v, w)
             BICOMP(w, v)
             low(v) ← min(low(v), low(w))
             if low(w) ≥ num(v)
               then { // v – корень или точка сочленения.
                      // Верхняя часть стека S до (v, w)
                      // включительно содержит
                      // двусвязную компоненту
                      then { (x, y) ⇐ S
                             j ← j + 1
                             E_j ← {(x, y)}
                             // выделить из стека компоненту E_j
                             while (x, y) ≠ (v, w) do { (x, y) ⇐ S
                                                        E_j ← E_j ∪ {(x, y)} }
      else if num(w) < num(v) and w ≠ u
             then { // (v, w) – обратное ребро
                    S ⇐ (v, w)
                    low(v) ← min(low(v), num(w)) }
return
```

Алгоритм 6.11. Определение двусвязных компонент графа

v	a	b	c	d	e	f	g
$low(v)$	1	1	3	6	1	1	3

.

Определим вычислительную сложность алгоритма. Поскольку он осуществляет поиск в глубину с конечным объемом дополнительной работы при прохождении каждого ребра, требуемое время выполнения равно $O(|V| + |E|)$.

```
for x ∈ V do num(x) ← 0
i ← j ← 0   // j – номер двусвязной компоненты E_j
S ← ∅   // стек S пуст
for x ∈ V do if num(x) = 0 then BICOMP(x, 0)

procedure BICOMP(v, u)
  i ← i + 1
  num(v) ← low(v) ← i
  for w ∈ Adj(v) do
    if num(w) = 0
      then { // (v, w) – ребро дерева
             S ⇐ (v, w)
             BICOMP(w, v)
             low(v) ← min(low(v), low(w))
             if low(w) ≥ num(v)
               then { // v – корень или точка сочленения.
                      // Верхняя часть стека S до (v, w)
                      // включительно содержит
                      // двусвязную компоненту
                      then { (x, y) ⇐ S
                             j ← j + 1
                             E_j ← {(x, y)}
                             // выделить из стека компоненту E_j
                             while (x, y) ≠ (v, w) do { (x, y) ⇐ S
                                                        E_j ← E_j ∪ {(x, y)} }
      else if num(w) < num(v) and w ≠ u
             then { // (v, w) – обратное ребро
                    S ⇐ (v, w)
                    low(v) ← min(low(v), num(w)) }
return
```

Алгоритм 6.11. Определение двусвязных компонент графа

6.5.3. Сильно связные компоненты

Понятие сильной связности определено для ориентированных графов. Орграф $G = (V, E)$ называется *сильно связным*, если для любых двух его вершин v и w существуют ориентированные пути из v к w и из w к v. Даже если орграф G не сильно связный, он может содержать сильно связные подграфы. Максимальные сильно связные подграфы орграфа G называются *сильно связными компонентами* этого графа, т. е. сильно связный орграф имеет только одну сильно связную компоненту.

Поиск в глубину позволяет находить сильно связные компоненты орграфа. Напомним, что процесс поиска в глубину разбивает ребра орграфа $G = (V, E)$ на четыре множества: E_T – множество ребер *DFS*-дерева (леса), E_B – множество обратных ребер, E_F – множество прямых ребер и E_C – множество поперечных ребер. При этом номера вершин пройденного орграфа представляют собой значения элементов $num(v)$ для всех $v \in V$. Разбиение ребер можно использовать для определения сильно связных компонент. Очевидно, что прямые ребра можно не рассматривать, так как они не влияют на сильную связность. Исходящие из вершины v обратные и поперечные ребра могут идти только в такие вершины w, для которых $num(v) > num(w)$.

Пусть $G_i = (V_i, E_i)$ – сильно связная компонента орграфа $G = (V, E)$, а $T = (V, E_T)$ – *DFS*-лес для G. Пусть $v, w \in V_i$, т. е. принадлежат одной и той же сильно связной компоненте G_i. Без потери общности будем считать, что $num(v) < num(w)$. По определению в G_i существует путь P из вершины v в вершину w. Пусть x – вершина на P с наименьшим номером (возможно, это сама вершина v). Путь P, дойдя до какого-нибудь потомка вершины x, уже не сможет выйти за пределы поддерева потомков вершины x, поскольку за пределы этого поддерева могут выходить лишь поперечные и обратные ребра, идущие в вершины с номерами, меньшими x. Следовательно, вершина w – потомок вершины x. Все вершины, номера которых заключены между $num(x)$ и $num(w)$, также являются потомками вершины x. Так как $num(x) \le num(v) < num(w)$, то вершина v – потомок вершины x. Таким образом, любые две вершины в G_i имеют общего предка в G_i. Следовательно, если G_i – сильно связная компонента орграфа G, то вершины G_i определяют дерево, которое является

подграфом *DFS*-леса. Обратное утверждение неверно: не каждое поддерево *DFS*-леса соответствует сильно связной компоненте.

Сильно связные компоненты орграфа можно найти, определив корни поддеревьев, соответствующих этим компонентам. Для этого определим функцию

$lowlink(v) = \min(\{num(v)\} \cup \{num(w) \mid$ существует поперечное или обратное ребро из потомка вершины v в вершину w и w находится в той же самой сильно связной компоненте, что и $v\}$).

Другими словами, $lowlink(v)$ есть наименьший номер среди номеров вершин в той сильно связной компоненте, в которой находится вершина v; эти вершины можно достичь по пути из нуля или большего числа ребер, за которыми следует не больше одного поперечного или обратного ребра. В этом случае вершина v будет корнем сильно связной компоненты тогда и только тогда, когда $lowlink(v) = num(v)$.

Вычисление значения $lowlink(v)$ в процессе поиска в глубину предполагает следующие действия.

1. Когда v проходится в первый раз, то $lowlink(v)$ присваивается значение $num(v)$.

2. Когда рассматривается обратное ребро (v, w), то $lowlink(v)$ присваивается наименьшее из его текущего значения $lowlink(v)$ и $num(w)$.

3. Когда рассматривается поперечное ребро (v, w), для которого v и w принадлежат одной и той же сильно связной компоненте, то $lowlink(v)$ присваивается наименьшее из его текущего значения $lowlink(v)$ и $num(w)$.

4. Когда осуществляется возврат в вершину v после полного сканирования сына w вершины v, то $lowlink(v)$ есть минимум из его текущего значения $lowlink(v)$ и значения $lowlink(w)$.

Процедура определения сильно связных компонент V_j ($j \geq 1$) орграфа $G = (V, E)$ представлена алгоритмом 6.12. Стек S используется для хранения вершин в порядке прохождения в глубину. Вершина w находится в стеке S тогда и только тогда, когда она принадлежит той же сильно связной компоненте, что и предок вершины v. Поэтому в момент, когда $lowlink(v) = num(v)$, вершины сверху стека S до вершины v включительно образуют сильно связную компоненту и исключаются из стека. В операторе **if** $w \in S$ определяется наличие вершины w в стеке S. Для реа-

лизации такой проверки достаточно использовать булев массив, элементы которого сопоставлены вершинам графа G. В случае если $w \in S$, то w находится в той же сильно связной компоненте, что и v, поскольку $w \in \mathrm{Adj}(v)$ и $w \in S$, т. е. существует путь из вершины w в вершину v.

for $x \in V$ **do** $num(x) \leftarrow 0$
$i \leftarrow j \leftarrow 0$ // j – номер сильно связной компоненты V_j
$S \leftarrow \varnothing$ // стек S пуст
for $x \in V$ **do if** $num(x) = 0$ **then** $STRONG(x)$

procedure $STRONG(v)$
  $i \leftarrow i + 1$
  $num(v) \leftarrow lowlink(v) \leftarrow i$
  $S \Leftarrow v$
  for $w \in \mathrm{Adj}(v)$ **do**
    if $num(w) = 0$
      then { // (v, w) – ребро дерева; $STRONG(w)$; $lowlink(v) \leftarrow \min(lowlink(v), lowlink(w))$ }
      else if $num(w) < num(v)$
        then { // (v, w) – обратное/ оперечное ребро; **if** $w \in S$ **then** $lowlink(v) \leftarrow \min(lowlink(v), num(w))$ }
  if $lowlink(v) = num(v)$
    then { // v – корень сильно связной компоненты; $j \leftarrow j + 1$; $V_j \leftarrow \varnothing$; // выделить из стека компоненту V_j; // $top(S)$ – верхний элемент стека S; **while** $num(top(S)) \geq num(v)$ **do** { $x \Leftarrow S$; $V_j \leftarrow V_j \cup \{x\}$ } }
return

Алгоритм 6.12. Определение сильно связных компонент орграфа

Для орграфа, изображенного на рисунке 6.6*а*, алгоритм выделяет сильно связные компоненты в таком порядке: $V_1 = \{e\}$, $V_2 = \{a, b, c, d\}$, $V_3 = \{g\}$, $V_4 = \{f\}$. При этом получаются следующие значения элементов $num(v)$ и $lowlink(v)$:

v	a	b	c	d	e	f	g
$num\,(v)$	1	3	2	4	5	6	7
$lowlink\,(v)$	1	1	1	2	5	6	7

.

Алгоритм осуществляет поиск в глубину, в ходе реализации которого требуются некоторые дополнительные операции при прохождении каждого ребра графа. Каждая вершина включается в стек и исключается из него в точности один раз. Поэтому временная сложность алгоритма равна $O(|V| + |E|)$.

6.6. Топологическая сортировка

Топологической сортировкой называется способ нумерации вершин ациклического орграфа $G = (V, E)$ целыми числами $1, 2, \ldots, |V|$ так, что если из вершины с номером i в вершину с номером j идет ориентированное ребро (дуга), то $i < j$. Ясно, что если орграф содержит цикл, то его топологическая сортировка невозможна. Особенностью ациклического орграфа является то, что он обязательно имеет по крайней мере одну вершину, в которую не входят дуги, и одну вершину, из которой не выходят дуги.

Одно из решений этой задачи заключается в следующем. Выбирается произвольная вершина графа, из которой не выходят дуги. Выбранной вершине присваивается наибольший номер $|V|$. Затем эта вершина удаляется из графа вместе с входящими в нее дугами. Поскольку оставшийся орграф также ациклический, процесс повторяется и присваивается следующий наибольший номер $|V| - 1$ вершине, из которой не выходят дуги, и т. д. Чтобы сделать топологическую сортировку более эффективной, нужно при поиске вершины, из которой не выходят дуги, избежать просмотра каждого модифицированного орграфа. В этом может помочь поиск в глубину. Причем производится единственный поиск в глубину на данном ациклическом орграфе G. Дополнительно требуется массив *label* размера $|V|$ для записи номеров топологически отсортированных вершин. Если в орграфе G имеется дуга (v, w), то $label\,(v) < label\,(w)$.

Процедура топологической сортировки ациклического орграфа $G = (V, E)$ приведена в алгоритме 6.13. Ациклический ориентированный граф, заданный структурой смежности, и результат его топологической сортировки приведены на рисунке 6.8.

for $x \in V$ **do** { // инициализация; $new(x) \leftarrow$ **true**; $label(x) \leftarrow 0$ }
$i \leftarrow |V| + 1$
for $x \in V$ **do**
  if $new(x)$ **then** { // x не имеет пронумерованного предка; $TOPSORT(x)$ }

procedure $TOPSORT(v)$
  $new(v) \leftarrow$ **false**
  for $w \in \text{Adj}(v)$ **do** { // обработать всех потомков w
    if $new(w)$
      then $TOPSORT(w)$
      else if $label(w) = 0$ **then** // G имеет цикл }
  // v пронумеровано числом, меньшим числа,
  // приписанного любому потомку
  $i \leftarrow i - 1$
  $label(v) \leftarrow i$
return

Алгоритм 6.13. Топологическая сортировка орграфа

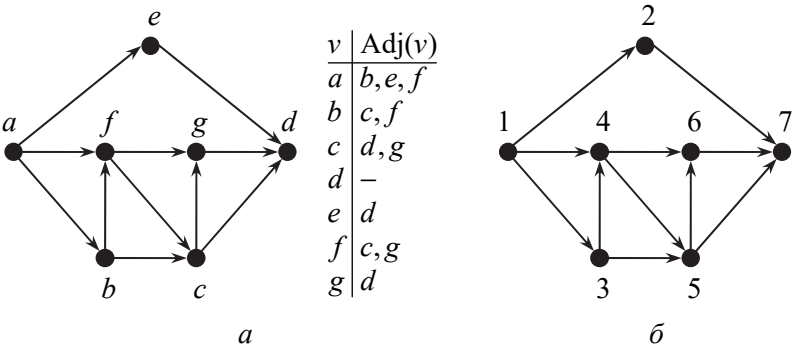


Рис. 6.8. Топологическая сортировка ациклического орграфа:
а – орграф и его структура смежности;
б – результат топологической сортировки орграфа.

Время работы алгоритма топологической сортировки равно $O(|V| + |E|)$, так как каждая дуга проходится один раз и для каждой вершины один раз вызывается *TOPSORT*.

6.7. Транзитивное замыкание

В прикладных интерпретациях теории графов часто возникают задачи, требующие ответа на вопрос, какие вершины графа связаны между собой. Если граф неориентированный, то задача сводится к определению его связных компонент (см. алгоритм 6.10). Если же граф ориентированный, то задача формулируется как определение наличия ориентированных путей между вершинами орграфа. В этом случае решение задачи становится более сложным и сводится к построению *транзитивного замыкания* орграфа.

Орграф является одним из способов представления бинарного отношения. Бинарное отношение R на некотором множестве M есть набор упорядоченных пар элементов этого множества. Если xRy, $x, y \in M$, то отношение R представляется в графе дугой (x, y). Отношение R является транзитивным, если для любых элементов $x, y, z \in M$ выполняется условие: если xRy и yRz, то xRz. Транзитивное замыкание бинарного отношения R – это отношение R^*, определяемое следующим образом: xR^*y, если существует последовательность из k элементов $x_1 = x, x_2, x_3, \ldots, x_k = y$, в которой между соседними элементами выполнено отношение R: $x_1Rx_2, x_2Rx_3, \ldots, x_{k-1}Rx_k$. Очевидно, что если xRy, то xR^*y, т. е. $R \subseteq R^*$.

Пусть $G = (V, E)$ – орграф, представляющий отношение E. Орграф $G^* = (V, E^*)$, представляющий транзитивное замыкание E^* отношения E, называется транзитивным замыканием графа G. Дуга (x, y), $x \neq y$, принадлежит E^* тогда и только тогда, когда в графе G существует ориентированный путь из вершины x в вершину y. Очевидно, что петля $(x, x) \in E^*$ тогда и только тогда, когда в графе G существует ориентированный цикл, включающий вершину x.

Пусть орграф $G = (V, E)$ представлен матрицей смежности $A = [a_{ij}]$ размера $|V| \times |V|$, где $a_{ij} = 1$, если имеется дуга $(i, j) \in E$, и $a_{ij} = 0$ в противном случае, $i, j \in V$. Пусть $G^* = (V, E^*)$ транзитивное замыкание графа G. Тогда матрица смежности $A^* = [a^*_{ij}]$ графа G^* представляет собой матрицу связности (достижимости) графа G, в которой $a^*_{ij} = 1$, если в графе G имеется ориентированный путь из i в j, и $a^*_{ij} = 0$, если такого пути нет. Пусть вер-

шины графа обозначаются числами $1, 2, \ldots, |V|$. Построение транзитивного замыкания G^* графа G сводится к вычислению матрицы A^* по матрице A. Для этого определяется последовательность матриц $A^0 = [a^0_{ij}]$, $A^1 = [a^1_{ij}]$, …, $A^{|V|} = [a^{|V|}_{ij}] = A^*$ следующим образом:

$$a^0_{ij} = a_{ij},$$
$$a^l_{ij} = a^{l-1}_{ij} \vee (a^{l-1}_{il} \wedge a^{l-1}_{lj}),$$

т. е. матрица A^l ($l \geq 1$) получается из матрицы A^{l-1} после обработки вершины l в графе, соответствующей матрице A^{l-1}. Очевидно, что $a^l_{ij} = a^{l-1}_{ij} \vee (a^{l-1}_{il} \wedge a^{l-1}_{lj})$ равно единице тогда и только тогда, когда либо $a^{l-1}_{ij} = 1$ (существует путь из i в j, использующий вершины только из множества $\{1, 2, \ldots, l-1\}$), либо a^{l-1}_{il} и a^{l-1}_{lj} одновременно равны единице (имеются пути из i в l и из l в j, использующие вершины только из множества $\{1, 2, \ldots, l-1\}$).

Рассмотренное соотношение полезно для понимания способа вычисления A^*, но оно неудобно для практических вычислений, так как желательно преобразовать A в A^* на месте, используя фиксированный объем дополнительной памяти. Пусть дана матрица A^{l-1}. Ее можно преобразовать в матрицу A^l следующим образом. Если $a^{l-1}_{il} = 0$, то $a^l_{ij} = a^{l-1}_{ij}$. Если $a^{l-1}_{il} = 1$, то $a^l_{ij} = a^{l-1}_{ij} \vee a^{l-1}_{lj}$. Обозначив i-ю строку матрицы A через $a_{i,*}$, получим

$$a^l_{i,*} = \begin{cases} a^{l-1}_{i,*}, \text{ если } a^{l-1}_{il} = 0, \\ a^{l-1}_{i,*} \vee a^{l-1}_{l,*}, \text{ если } a^{l-1}_{il} = 1. \end{cases}$$

Для $i \neq l$ значение $a^{l-1}_{i,*}$ используется только при вычислении $a^l_{i,*}$, поэтому его значения могут меняться, не влияя на вычисления $a^l_{k,*}$ для $k \neq i$. Элемент $a^{l-1}_{l,*}$, хотя и используется для вычисления других строк, но $a^l_{l,*} = a^{l-1}_{l,*}$, поэтому никакие ложные дуги не добавляются.

Рассмотренный способ вычисления транзитивного замыкания ориентированного графа $G = (V, E)$ реализован алгоритмом 6.14, известным как алгоритм Уоршелла (Warshall). Очевидно, что данный алгоритм требует $O(|V|^3)$ операций.

for $l \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    if $a_{il} = 1$ **then for** $k \leftarrow 1$ **to** $|V|$ **do** $a_{ik} \leftarrow a_{ik} \vee a_{lk}$

Алгоритм 6.14. Алгоритм Уоршелла вычисления транзитивного замыкания

Если строка матрицы A помещается в машинное слово, то самый внутренний цикл **for** с параметром k можно выполнить за одну операцию логического сложения двух строк матрицы. Экономия получается, даже если строка требует нескольких слов. Соответствующая модификация алгоритма представлена алгоритмом 6.15 и заключается в замене цикла **for** с параметром k на одну операцию присваивания $a_{i,*} \leftarrow a_{i,*} \vee a_{l,*}$. Данный алгоритм является в общем случае наиболее эффективным алгоритмом вычисления транзитивного замыкания.

for $l \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do if** $a_{il} = 1$ **then** $a_{i,*} \leftarrow a_{i,*} \vee a_{l,*}$

Алгоритм 6.15. Модификация алгоритма Уоршелла

Алгоритм Уоршелла обрабатывает все дуги, заходящие в вершину, до начала обработки следующей вершины, т. е. обрабатывает матрицу A по столбцам (алгоритм, ориентированный по столбцам). Модификацией алгоритма Уоршелла для обработки матрицы A по строкам является алгоритм Уоррена (Warren) (алгоритм 6.16). Этот алгоритм является двухпроходным строкоориентированным алгоритмом. В нем при обработке вершины, например i, в первом проходе обрабатываются только дуги, связанные с вершинами, меньшими i, во втором проходе – только дуги, связанные с вершинами, большими i. Иными словами, алгоритм преобразует матрицу смежности A графа G в матрицу смежности A^* графа G^*, обрабатывая в первом проходе только элементы матрицы, расположенные ниже ее главной диагонали, а во втором проходе – только элементы матрицы, расположенные выше ее главной диагонали. Таким образом, при каждом проходе обрабатывается не более $|V|(|V|-1)/2$ дуг.

Очевидно, что алгоритм Уоррена требует $O(|V|^3)$ операций, т. е. имеет такую же сложность, что и алгоритм Уоршелла.

// 1-й проход
for $i := 2$ **to** $|V|$ **do**
  for $j := 1$ **to** $i - 1$ **do**
    if $a_{ij} = 1$ **then for** $k := 1$ **to** $|V|$ **do** $a_{ik} := a_{ik} \vee a_{jk}$
// 2-й проход
for $i := 1$ **to** $|V| - 1$ **do**
  for $j := i + 1$ **to** $|V|$ **do**
    if $a_{ij} = 1$ **then for** $k := 1$ **to** $|V|$ **do** $a_{ik} := a_{ik} \vee a_{jk}$

Алгоритм 6.16. Алгоритм Уоррена вычисления транзитивного замыкания

Однако алгоритм Уоррена будет выполняться быстрее алгоритма Уоршелла для больших разреженных орграфов, особенно при страничной организации памяти. Ясно, что для существенного повышения эффективности алгоритм Уоррена можно модифицировать, используя тот же способ, что и для алгоритма Уоршелла при получении алгоритма 6.15, т. е. заменив самые внутренние циклы **for** операциями логического сложения двух строк матрицы.

6.8. Фундаментальное множество циклов

Пусть $T = (V, E_T)$ – остовное дерево неориентированного графа $G = (V, E)$. Если к остовному дереву T добавить произвольное ребро $e \in E - E_T$, то полученный подграф $(V, E_T \cup \{e\})$ содержит точно один цикл. Такой цикл является элементом *фундаментального множества* циклов графа G относительно дерева T. Известно, что каждое остовное дерево графа G включает $|V| - 1$ ребер. Следовательно, в фундаментальном множестве циклов относительно любого остовного дерева графа G имеется $|E| - |V| + 1$ циклов. Важной особенностью любого цикла из фундаментального множества является то, что он содержит только одно ребро из множества $E - E_T$ и больше это ребро не присутствует ни в одном другом цикле фундаментального множества.

Полезность отыскания фундаментального множества циклов обусловлена тем, что это множество полностью определяет циклическую структуру графа, т. е. любой цикл в графе может быть представлен комбинацией циклов из фундаментального множе-

ства. Для этого используется операция *симметрической разности*, которая для произвольных множеств A и B определяется следующим образом: $A \oplus B = (A \cup B) - (A \cap B)$.

Пусть $F = \{C_1, C_2, \ldots, C_{|E|-|V|+1}\}$ – фундаментальное множество циклов, где каждый цикл C_i является подмножеством ребер графа G, т. е. $C_i \subseteq E$. Тогда произвольный цикл C графа G можно однозначно представить как симметрическую разность некоторого числа циклов фундаментального множества, т. е. $C = C_{i_1} \oplus C_{i_2} \oplus \ldots \oplus C_{i_t}$. Следует обратить внимание на то, что цикл фундаментального множества нельзя выразить через симметрическую разность других циклов множества. На рисунке 6.9 показаны граф (*а*), его остовное дерево (*б*) и фундаментальное множество циклов относительно этого дерева (*в*).

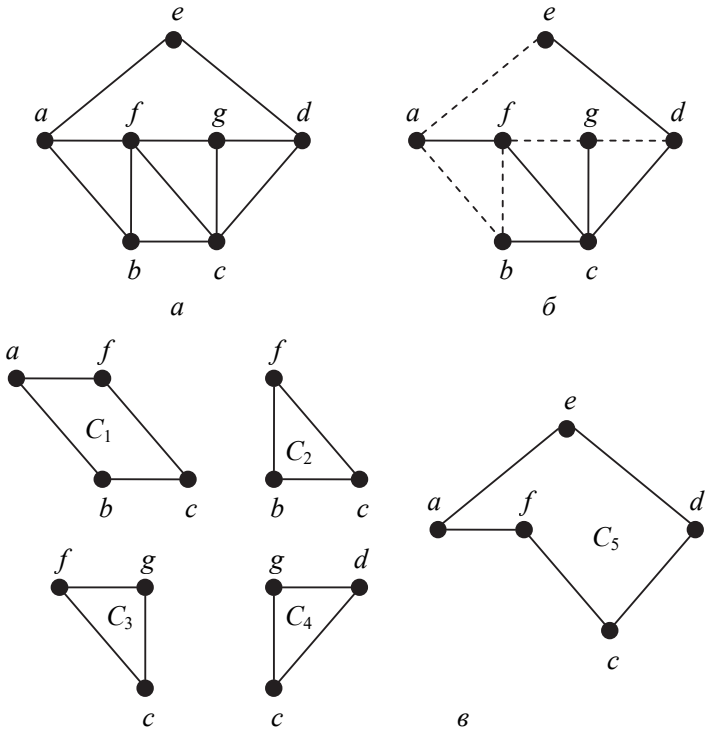


Рис. 6.9. Фундаментальное множество циклов графа:
а – неориентированный граф; *б* – остовное дерево графа;
в – фундаментальные циклы графа.

Например, цикл (a, f, b, a) есть $C_1 \oplus C_2$, а цикл (b, c, d, g, f, b) есть $C_2 \oplus C_3 \oplus C_4$. Следует отметить, что не каждая такая симметрическая разность является циклом. Например, $C_2 \oplus C_4$ состоит из двух не связанных между собой циклов.

Очевидным подходом к нахождению фундаментального множества циклов является использование поиска в глубину, который строит остовное дерево (*DFS*-дерево), и каждое обратное ребро порождает цикл относительно этого дерева. Когда поиск в глубину достигает обратного ребра (v, w), цикл состоит из ребер дерева, идущих от вершины w к вершине v, и обратного ребра (v, w). Ясно, что для хранения пути от w к v необходим стек, т. е. стек всегда содержит последовательность вершин из пути, идущего от корня к исследуемой в данный момент вершине. Поэтому если анализируемое ребро (v, w) является обратным ребром остовного дерева, то цикл будет состоять из ребра (v, w) и ребер, соединяющих вершины из верхней группы элементов стека, начиная с вершины v. При этом верхняя группа элементов не должна удаляться из стека, поскольку одно и то же ребро может входить во многие циклы.

Рассмотренный метод построения фундаментального множества циклов $F = \{C_1, C_2, \ldots, C_{|E|-|V|+1}\}$ неориентированного графа $G = (V, E)$ представлен алгоритмом 6.17.

Операции со стеком S используют индексацию элементов стека для реализации доступа к верхней группе элементов без удаления их из стека, т. е. операции детализированы до уровня работы с указателем t вершины стека.

Вычислительная сложность алгоритма, не считая операции сохранения циклов C_j, как и во всех алгоритмах, основанных на поиске в глубину, равна $O(|V| + |E|)$. Дополнительно необходимо учесть следующее. В цикле **for** каждое ребро (v, w) просматривается дважды, но одно и то же ребро входит во многие циклы C_j, т. е. на самом деле просматривается много раз. Таким образом, число операций равно $O(|V| + |E| + l)$, где l – суммарная длина всех порожденных циклов. Величина l зависит от графа и от порядка вершин в списках смежностей. Поскольку каждый цикл имеет длину не более чем $|V| - 1$, то очевидно, что $l \le (|V| - 1)(|E| - |V| + 1)$, т. е. $l = O(|V||E|)$. Общая сложность алгоритма равна $O(|V||E| + |V| + |E|)$ или $O(|V||E|)$.

$i \leftarrow j \leftarrow t \leftarrow 0$
// j – номер цикла C_j, t – указатель вершины стека S
$S \leftarrow \varnothing$ // стек S пуст
for $x \in V$ **do** $num(x) \leftarrow 0$ // инициализация
for $x \in V$ **do if** $num(x) = 0$ **then** $CYCLE(x, 0)$

procedure $CYCLE(v, u)$
  $i \leftarrow i + 1$
  $num(v) \leftarrow i$
  $t \leftarrow t + 1$
  $S_t \leftarrow v$ // поместить v в вершину стека S
  for $w \in$ Adj(v) **do**
    if $num(w) = 0$
      then $\begin{cases} CYCLE(w, v) \\ t \leftarrow t - 1 \quad \text{// исключить элемент из стека } S \end{cases}$
      else if $num(w) < num(v)$ **and** $w \neq u$
        then $\begin{cases} \text{// } (v, w) - \text{обратное ребро} \\ j \leftarrow j + 1 \\ \text{// сохранить полученный цикл в } C_j \\ C_j \leftarrow (w, S_t, S_{t-1}, \ldots, S_k) \\ \text{// здесь } S_k = w, \text{ а } S_t = v \end{cases}$
return

Алгоритм 6.17. Нахождение фундаментального множества циклов графа

6.9. Кратчайшие пути

Пусть $G = (V, E)$ – связный взвешенный простой ориентированный граф с матрицей весов дуг $W = [w_{ij}]$, где w_{ij} – длина дуги (i, j). Если в графе отсутствует дуга (i, j), то $w_{ij} = \infty$. Длина ориентированного пути есть сумма длин дуг, входящих в путь. Путь $P(s, f)$ между вершинами s и f ($s, f \in V$), имеющий минимальную длину, называется *кратчайшим путем*. Длина $dist(s, f)$ кратчайшего пути называется *расстоянием* от s до f. Очевидно, что $dist(i, i) = 0$ для любого $i \in V$.

Многие графы можно свести к рассмотренному типу орграфа. Если орграф не является простым, то можно отбросить все петли и заменить каждое множество параллельных дуг дугой с наименьшей длиной из этого множества. Если граф неориентированный, то можно рассматривать граф, который получается заменой каждого неориентированного ребра (i, j) парой дуг (i, j)

и (j, i) с весом, равным весу исходного неориентированного ребра. Если граф не взвешенный, то можно считать, что все дуги имеют один и тот же вес.

6.9.1. Кратчайшие пути от фиксированной вершины

Большинство алгоритмов нахождения кратчайших путей от фиксированной вершины s основаны на следующем. Вычисляются некоторые верхние ограничения $D(v)$ на расстояния от s до всех вершин $v \in V$. На каждом этапе, когда устанавливается, что для некоторой вершины $u \in V$ $D(u) + w_{u,v} < D(v)$, оценка $D(v)$ улучшается и принимает новое значение $D(u) + w_{u,v}$. Процесс завершается, когда дальнейшее улучшение ни одного из ограничений невозможно. Тогда значение каждого элемента $D(v) = dist(s, v)$, т. е. расстоянию от s до v. При поиске кратчайшего пути $P(s, f)$ между двумя фиксированными вершинами s и f процесс завершается, если дальнейшее улучшение элемента $D(f)$ невозможно. Тогда расстояние от s до f есть $dist(s, f) = D(f)$.

Для случая, когда веса всех дуг неотрицательны, эффективным алгоритмом нахождения кратчайших путей от фиксированной вершины s до всех остальных вершин в графе является *алгоритм Дейкстры*. Он основан на просмотре графа в ширину, в процессе которого производится специальная расстановка меток вершин. Метки вершин $v \in V$ есть значения длин путей от фиксированной вершины s до v. Сначала вершине v присваивается временная метка, значение которой есть длина пути от s до v. В этот момент еще не все возможные пути от s до v рассмотрены. В процессе исследования орграфа значение метки модифицируется до тех пор, пока не будут исследованы все возможные пути от s до v. После этого метка вершины v не изменяется и принимает окончательное значение, равное длине кратчайшего пути от s до v. Процесс завершается, когда все вершины будут иметь окончательные метки. Если требуется определить кратчайший путь только до некоторой фиксированной вершины f, то процесс можно завершить после получения окончательной метки для f.

Алгоритм Дейкстры поиска кратчайших путей от фиксированной вершины s до всех остальных вершин орграфа $G = (V, E)$ с матрицей весов $W = [w_{ij}]$ представлен алгоритмом 6.18. Для записи значений меток каждой вершине $v \in V$ сопоставлен элемент

$label(v)$. Множество T есть множество вершин графа, еще не имеющих окончательных меток. Значением элемента *last* является вершина, которая последней на данном шаге получила окончательную метку.

for $v \in V$ **do** $label(v) \leftarrow \infty$
$label(s) \leftarrow 0$
$last \leftarrow s$
$T \leftarrow V - \{s\}$
while $T \neq \varnothing$ **do**
  for $v \in T$ **do**
    if $label(v) > label(last) + w_{last,v}$
      then $label(v) \leftarrow label(last) + w_{last,v}$; $pred(v) \leftarrow last$
  $u \leftarrow$ любая вершина с $label(u) = \min\{label(k) : k \in T\}$
  $T \leftarrow T - \{u\}$
  $last \leftarrow u$
// определить кратчайшие пути $P(s, v)$ для всех $v \in V - \{s\}$
for $v \in V - \{s\}$ **do**
  if $label(v) \neq \infty$
    then // существует путь от s до v; $P(s, v) \leftarrow (s, \ldots, pred(pred(v)), pred(v), v)$
    else // не существует пути от s до v

Алгоритм 6.18. Алгоритм Дейкстры поиска кратчайших путей

Вначале вершине s присваивается окончательная метка $label(s) = 0$ (нулевое расстояние до самой себя), а остальным $|V| - 1$ вершинам – временная метка ∞. На каждом шаге метки модифицируются следующим образом. Каждой вершине $v \in T$ (не имеющей окончательной метки) присваивается новая временная метка

$$label(v) = \min(label(v), label(last) + w_{last,v}),$$

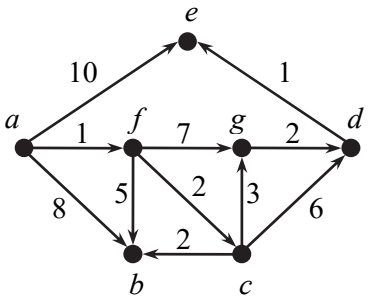
где *last* – вершина, которая получила окончательную метку на предыдущем шаге. Затем находится вершина u с наименьшей из всех временных меток, которая становится окончательной для этой вершины. Таким образом, на каждом шаге одной вершине с временной меткой присваивается окончательная метка. Процесс продолжается до тех пор, пока все вершины графа не получат окончательные метки. Тогда окончательная метка каждой вершины $v \in V$ равна длине кратчайшего пути (расстоянию) от s

до v, т. е. $label(v) = dist(s, v)$. Если $label(v) = \infty$, то это означает, что не существует ориентированного пути от s до v.

Для определения самого пути $P(s, v)$ от s до v каждой вершине $v \in V$ сопоставлен указатель $pred(v)$ – вершина, предшествующая вершине v в кратчайшем пути. Тогда последовательность вершин кратчайшего пути от s до v есть вершины

$P(s, v) = (s, \ldots, pred(pred(pred(v))), pred(pred(v)), (pred(v), v)$.

Работа алгоритма Дейкстры проиллюстрирована на рисунке 6.10 (ищутся кратчайшие пути от вершины a).



а

Шаг	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	[0]	∞	∞	∞	∞	∞	∞
1	[0]	8	∞	∞	10	[1]	∞
2	[0]	6	[3]	∞	10	[1]	8
3	[0]	[5]	[3]	9	10	[1]	6
4	[0]	[5]	[3]	9	10	[1]	[6]
5	[0]	[5]	[3]	[8]	10	[1]	[6]
6	[0]	[5]	[3]	[8]	[9]	[1]	[6]

б

Шаг	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	–	–	–	–	–	–	–
1	–	*a*	–	–	*a*	*a*	–
2	–	*f*	*f*	–	*a*	*a*	–
3	–	*c*	*f*	*c*	*a*	*a*	*c*
4	–	*c*	*f*	*c*	*a*	*a*	*c*
5	–	*c*	*f*	*g*	*a*	*a*	*c*
6	–	*c*	*f*	*g*	*d*	*a*	*c*

в

Путь	*dist* (*v*)	Вершины пути
$P(a, b)$	5	*a*, *f*, *c*, *b*
$P(a, c)$	3	*a*, *f*, *c*
$P(a, d)$	8	*a*, *f*, *c*, *g*, *d*
$P(a, e)$	9	*a*, *f*, *c*, *g*, *d*, *e*
$P(a, f)$	1	*a*, *f*
$P(a, g)$	6	*a*, *f*, *c*, *g*

г

Рис. 6.10. Процесс работы алгоритма Дейкстры:
а – взвешенный орграф; *б* – процесс присвоения меток $label(v)$ (окончательные метки заключены в квадратные скобки);
в – процесс формирования указателей $pred(v)$;
г – кратчайшие пути от вершины *a* до всех остальных вершин орграфа.

Если необходимо найти кратчайший путь от s только до некоторой фиксированной вершины f, то в алгоритме достаточно

заменить условие $T \neq \varnothing$ цикла **while** на **while** $f \in T$ и определить единственный путь $P(s, f)$.

Определим вычислительную сложность алгоритма. Цикл **while** выполняется $|V| - 1$ раз (если ищется путь между фиксированной парой вершин s и f – не более $|V| - 1$). Число операций при каждом выполнении цикла, очевидно, равно $O(|V|)$. Поэтому алгоритму требуется $O(|V|^2)$ операций.

Следует обратить внимание на то, что алгоритм Дейкстры не работает, если некоторые из весов дуг отрицательные. Это связано с тем, что после присвоения вершине окончательной метки алгоритм не может ее изменить, а наличие отрицательных весов может этого потребовать.

6.9.2. Кратчайшие пути между всеми парами вершин

Для нахождения кратчайших путей между всеми $|V|(|V| - 1)$ упорядоченными парами вершин в орграфе с $|V|$ вершинами можно использовать $|V|$-кратное применение алгоритма Дейкстры, т. е. применить его к каждой вершине графа. Такой подход, очевидно, потребует $O(|V|^3)$ операций для графов с неотрицательными весами дуг. Однако существуют более эффективные алгоритмы, которые применимы даже тогда, когда в графе присутствуют отрицательные веса дуг (но не существует ориентированных циклов отрицательной длины – в этом случае задача становится бессмысленной). Одним из таких алгоритмов является *алгоритм Флойда* (Floyd).

Процесс вычисления кратчайших путей алгоритмом Флойда основан на алгоритме Уоршелла для вычисления транзитивного замыкания (см. алгоритм 6.14). Пусть $W = [w_{ij}]$ – матрица весов орграфа $G = (V, E)$ и пусть вершины графа обозначаются числами 1, 2, …, $|V|$. Необходимо вычислить матрицу $W^* = [w^*_{ij}]$, в которой w^*_{ij} – длина кратчайшего пути от $i \in V$ до $j \in V$. Для этого определяется последовательность матриц $W^0 = [w^0_{ij}]$, $W^1 = [w^1_{ij}]$, …, $W^{|V|} = [w^{|V|}_{ij}] = W^*$ следующим образом:

$$w^0_{ij} = w_{ij},$$
$$w^l_{ij} = \min(w^{l-1}_{ij}, w^{l-1}_{il} + w^{l-1}_{lj}),$$

т. е. матрица $W^l = [w^l_{ij}]$ $(l \geq 1)$ получается из матрицы W^{l-1} после обработки вершины l в графе, соответствующем матрице W^{l-1}.

Очевидно, что значение w^l_{ij} есть длина кратчайшего пути от $i \in V$ до $j \in V$ с промежуточными вершинами, принадлежащими только множеству $\{1, 2, \ldots, l\} \subseteq V$.

Для порождения самих $|V|(|V|-1)$ кратчайших путей можно построить матрицу $P = [p_{ij}]$, в которой p_{ij} – вершина, непосредственно следующая за вершиной i в кратчайшем пути от $i \in V$ до $j \in V$. Тогда кратчайший путь от i до j представляет собой последовательность вершин $v_1, v_2, \ldots, v_t$, где $v_1 = i$, $v_t = j$, $v_k = p_{k-1,j}$ для $1 < k \le t$. Матрица P легко строится по мере вычисления матриц $W^0, W^1, \ldots, W^{|V|}$. Сначала

$$p_{ij} = \begin{cases} j, \text{ если } w_{ij} \neq \infty, \\ 0, \text{ если } w_{ij} = \infty. \end{cases}$$

Если на шаге с номером l выполняется условие $w_{il} + w_{lj} < w_{ij}$, то $p_{ij} = p_{il}$, в противном случае – сохраняется предыдущее значение. Очевидно, что в конце работы алгоритма $p_{ij} = 0$ в том случае, когда $w^*_{ij} = \infty$, т. е. когда не существует пути от i до j.

Реализация метода Флойда представлена алгоритмом 6.19. Использование матрицы P для выписывания самих кратчайших путей как последовательности вершин в алгоритме не отражено. Очевидно, что сложность этого алгоритма есть $O(|V|^3)$.

for $i \leftarrow 1$ **to** $|V|$ **do** $w_{ii} \leftarrow 0$
for $i \leftarrow 1$ **to** $|V|$ **do**
  for $j \leftarrow 1$ **to** $|V|$ **do if** $w_{ij} \neq \infty$
    then $p_{ij} \leftarrow j$
    else $p_{ij} \leftarrow 0$
for $l \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    if $w_{il} \neq \infty$
      then for $j \leftarrow 1$ **to** $|V|$ **do**
        if $w_{il} + w_{lj} < w_{ij}$
          then $\begin{cases} p_{ij} \leftarrow p_{il} \\ w_{ij} \leftarrow w_{il} + w_{lj} \end{cases}$

Алгоритм 6.19. Алгоритм Флойда поиска кратчайших путей

Следует обратить внимание на то, что рассмотренный алгоритм является наиболее эффективным из всех методов вычисления единственного кратчайшего пути между фиксированной парой вершин орграфа с отрицательными весами (при отсутствии циклов отрицательной длины). Другими словами, для орграфов с отрицательными весами дуг без циклов отрицательной длины не известен ни один алгоритм нахождения расстояния между одной фиксированной парой вершин, который был бы значительно эффективнее алгоритма нахождения расстояний между всеми парами вершин.

6.10. Эйлеровы пути

Произвольный путь, проходящий по каждому ребру графа в точности по одному разу (в одной и той же вершине графа можно бывать многократно), называется *эйлеровым путем*. Эйлеров путь, который начинается и завершается в одной и той же вершине, называется *эйлеровым циклом*. Графы, содержащие эйлеровы циклы, называются *эйлеровыми графами*.

Неориентированный граф $G = (V, E)$ имеет эйлеров цикл тогда и только тогда, когда он связный и степень каждой вершины четная (степень вершины есть число инцидентных ей ребер).

Необходимость этих условий очевидна. В несвязном графе каждый цикл принадлежит какой-либо его связной компоненте, т. е. не проходит через все ребра графа. Исключением является случай, когда все связные компоненты, кроме одной, являются изолированными вершинами. Рассмотрим необходимость четности степеней всех вершин. Пусть некоторая вершина v появляется в эйлеровом цикле k раз. Это означает, что цикл приходит в эту вершину по k инцидентным ей ребрам и выходит из нее по другим k инцидентным ребрам, т. е. степень этой вершины составляет $2k$. Поскольку цикл эйлеров, нет никаких других инцидентных вершине v ребер, по которым цикл не проходит.

Достаточность условий существования эйлерова цикла является следствием анализа процедуры его построения. Построение эйлерова цикла начинается с произвольной вершины v графа. От этой вершины строится путь по ребрам графа до тех пор, пока это возможно. Инцидентные некоторой вершине ребра, еще не вошедшие в путь, назовем свободными ребрами данной верши-

ны. Пусть в некоторый момент после добавления нового ребра к строящемуся пути мы приходим в вершину u. Число свободных ребер в вершине u уменьшается на единицу, т. е. становится нечетным. Поскольку степень вершины u четная, то у данной вершины существует хотя бы одно свободное ребро, по которому можно продолжить построение пути. После ухода из вершины u число ее свободных ребер уменьшается еще на единицу и вновь становится четным. Что касается исходной вершины v, то после начала процесса число ее свободных ребер нечетно и остается нечетным до тех пор, пока мы не вернемся в эту вершину. Таким образом, процесс построения пути может закончиться только в той вершине, из которой он начинался, т. е. получится цикл.

Если полученный цикл C_1 не проходит через все рёбра графа, то обязательно существует (поскольку граф связный) хотя бы одна принадлежащая полученному циклу вершина w с четным числом свободных ребер. Начиная с этой вершины, можно построить цикл C_2, начинающийся и завершающийся в вершине w. Для объединения циклов C_1 и C_2 в один цикл необходимо разделить цикл C_1 на два участка относительно вершины w: первый участок с началом в вершине v и концом в вершине w и второй участок с началом в вершине w и концом в вершине v. В полученный разрыв включается цикл C_2. В результате получается цикл, начинающийся и заканчивающийся в вершине v, но включающий в себя большее число ребер. Если и этот цикл не проходит через все ребра графа, то такой процесс расширения цикла повторяется до получения эйлерова цикла.

Построение эйлерова цикла для связного неориентированного графа $G = (V, E)$ без вершин нечетной степени представлено алгоритмом 6.20. Предполагается, что граф задан структурой смежности. Пусть в качестве произвольной начальной вершины выбрана вершина $v = v_0$. Цикл **while** строит путь с началом в вершине v_0, помещая вершины этого пути в стек SC и удаляя из графа включенные в путь ребра. В результате удалений ребер структура смежности графа будет содержать информацию только о свободных ребрах, еще не включенных в цикл. Процесс построения пути продолжается до тех пор, пока есть возможность удлинения пути, т. е. пока $\mathrm{Adj}(v) \neq \varnothing$. Когда достигается вершина v, для которой $\mathrm{Adj}(v) = \varnothing$, это означает, что у вершины v нет свободных ребер для продолжения построения пути.

$SC \leftarrow \varnothing$ // SC – стек для хранения циклов
$SE \leftarrow \varnothing$ // SE – стек для хранения эйлерова цикла
$v \leftarrow$ произвольная вершина графа
$SC \Leftarrow v$

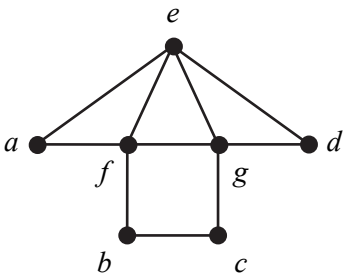
while $SC \neq \varnothing$ **do**
- $v \leftarrow top(SC)$
- // $top(SC)$ – верхний элемент стека SC
- **if** $\mathrm{Adj}(v) \neq \varnothing$
  - **then**
    - $u \leftarrow \mathrm{Adj}(v)$
    - $SC \Leftarrow u$
    - // удаление ребра (v, u) из графа
    - $\mathrm{Adj}(v) \leftarrow \mathrm{Adj}(v) - \{u\}$
    - $\mathrm{Adj}(u) \leftarrow \mathrm{Adj}(u) - \{v\}$
    - $v \leftarrow u$
  - **else**
    - $v \Leftarrow SC$
    - $SE \Leftarrow v$

Алгоритм 6.20. Построение эйлерова цикла

В этом случае $v = v_0$, т. е. построен цикл, вершины которого находятся в стеке SC, а его ребра удалены из графа. Вершина $v = v_0$ переносится в стек SE. Затем среди вершин полученного цикла необходимо найти вершину, у которой есть свободные ребра. Для этого очередной исследуемой вершиной v становится верхний элемент стека SC. Если у этой вершины нет свободных ребер, т. е. $\mathrm{Adj}(v) = \varnothing$, она переносится из стека SC в стек SE. Если же для этой вершины $\mathrm{Adj}(v) \neq \varnothing$, то процесс построения пути повторяется и в стек SC помещается полученный цикл с началом и концом в этой вершине. Процесс продолжается до тех пор, пока стек SC не станет пустым. Это означает, что в графе не осталось непройденных ребер и получен эйлеров цикл, который содержится в стеке SE.

Эйлеров граф, заданный структурой смежности, и найденный алгоритмом эйлеров цикл представлены на рисунке 6.11. Построение цикла начинается с вершины a. Первым найденным в графе циклом является цикл a, e, d, g, c, b, f, a, вершины которого хранятся в стеке SC. Затем вершина a переносится из стека SC в стек SE и ищется вершина из этого цикла, имеющая свободные ребра. Такой вершиной является вершина f, начиная с которой строится следующий цикл f, e, g, f, который добавляется в стек SC. В результате получается более длинный цикл a, e, d, g,

c, *b*, *f*, *e*, *g*, *f*, *a* (конечная вершина *a* находится в стеке *SE*). Вершина *f* переносится из стека *SC* в стек *SE* и в полученном цикле ищется вершина со свободными ребрами. В процессе такого поиска все вершины, не имеющие свободных ребер, переносятся из стека *SC* в стек *SE*. Поскольку в рассматриваемом графе больше нет вершин со свободными ребрами, процесс завершается, а полученный цикл в стеке *SE* является эйлеровым.



v	Adj(*v*)
a	*e*, *f*
b	*c*, *f*
c	*b*, *g*
d	*e*, *g*
e	*a*, *d*, *f*, *g*
f	*a*, *b*, *e*, *g*
g	*c*, *d*, *e*, *f*

Эйлеров цикл: *a*, *e*, *d*, *g*, *c*, *b*, *f*, *e*, *g*, *f*, *a*

Рис. 6.11. Эйлеров цикл в графе, заданном структурой смежности

Вычислительная сложность алгоритма определяется числом итераций цикла **while**. Каждая итерация либо помещает вершину в стек *SC* и удаляет ребро из графа, либо переносит вершину из стека *SC* в стек *SE*. Если структуру смежности реализовать таким образом, что списки смежности представляют собой двусвязные списки, причем узел с вершиной *w* из списка Adj(*v*) содержит указатель на узел с вершиной *v* в списке Adj(*w*), тогда удалить ребро можно за фиксированное время. Таким образом, сложность алгоритма есть $O(|V| + |E|)$.

Следует отметить, что рассмотренный алгоритм может обрабатывать только эйлеровы графы, поскольку не проверяет необходимые и достаточные условия существования эйлерова цикла.

Эйлеров путь, не являющийся циклом, существует в графе тогда и только тогда, когда граф связный и содержит не более чем две вершины нечетной степени. Вершины нечетной степени являются началом и концом эйлерова пути, поскольку из начальной вершины путь лишний раз выходит, а в конечную вершину лишний раз приходит.

В ориентированном графе существует эйлеров цикл тогда и только тогда, когда граф связный и входящая степень каждой вершины равна ее исходящей степени. Доказательство необходимости и достаточности условий практически такое же, что и для неориентированных графов, только вместо четности степеней вершин рассматриваются входящие и исходящие степени.

Упражнения

1. Разработать алгоритмы преобразования одного представления графа в другое для всех пар следующих представлений графа:

а) матрица смежности;

б) матрица инциденций;

в) список ребер;

г) структура смежности.

Оценить вычислительную сложность разработанных алгоритмов.

2. *Стоком* простого ориентированного графа называется вершина, в которую ведут ребра из всех других вершин графа и из которой не выходит ни одно ребро. Разработать алгоритм, который по заданной матрице смежности орграфа за время $O(|V|)$ определяет, содержит орграф сток или нет.

3. *Квадратом* ориентированного графа $G = (V, E)$ называется граф $G^2 = (V, E^2)$, построенный следующим образом: ребро $(u, w) \in E^2$, если существует вершина $v \in V$, для которой $(u, v) \in E$ и $(v, w) \in E$, т. е. две вершины соединяются ребром, если в исходном графе между ними существует путь из двух ребер. Разработать алгоритмы преобразования графа G в граф G^2, если исходный граф G представлен:

а) структурой смежности;

б) матрицей смежности.

Оценить время работы полученных алгоритмов.

4. Разработать нерекурсивный алгоритм поиска в глубину.

5. Разработать алгоритм, основанный на поиске в глубину, который находит в неориентированном связном графе путь, проходящий каждое ребро точно один раз в каждом направлении.

***6**. Разработать алгоритм, основанный на поиске в глубину, который, если возможно, так ориентирует ребра связного неориентированного графа, чтобы получить сильно связный ориентированный граф. Обосновать, что это возможно тогда и только тогда, когда удаление из неориентированного графа любого ребра оставляет граф связным.

***7**. *Мостом* в связном неориентированном графе называется ребро, удаление которого делает граф несвязным. Разработать алгоритм, который за время $O(|V| + |E|)$ находит все мосты графа. За основу ал-

горитма можно взять алгоритм 6.11 определения двусвязных компонент графа. Тогда ребро (v, w) будет мостом только в том случае, если оно является ребром *DFS*-дерева и $low(w) \geq num(w)$ (доказать данное утверждение).

***8**. Граф $G = (V, E)$ называется *двудольным*, если множество вершин V можно разбить на два непересекающихся подмножества A и B ($V = A \cup B$, $A \cap B = \varnothing$) так, что каждое ребро $(v, w) \in E$ соединяет вершину из A с вершиной из B, т. е. $v \in A$ и $w \in B$. Разработать два алгоритма, основанных соответственно на поиске в глубину и поиске в ширину, которые определяют, является заданный граф двудольным или нет, за время $O(|V| + |E|)$.

9. Модифицировать алгоритм 6.20 построения эйлерова цикла так, чтобы он мог обрабатывать любые неориентированные графы, т. е. алгоритм должен строить эйлеров цикл или эйлеров путь, если они существуют (выполняются соответствующие необходимые и достаточные условия), или сообщать о невозможности их построения. Сложность алгоритма должна остаться прежней, т. е. $O(|V| + |E|)$.

10. Модифицировать алгоритм из упражнения 9 так, чтобы он строил эйлеров цикл или эйлеров путь для ориентированных графов (при выполнении соответствующих условий их существования).

11. *Гамильтоновым циклом* в связном неориентированном графе называется цикл, содержащий все вершины графа в точности по одному разу. Разработать алгоритм, основанный на поиске с возвратом (см. п. 3.1), который находит все гамильтоновы циклы в заданном графе (если они есть).

ЗАКЛЮЧЕНИЕ

В данном учебнике была сделана попытка обобщить сведения об основных структурах данных, алгоритмах их создания и обработки, методах теоретического и экспериментального анализа алгоритмов для определения их сложности, реализации этих структур и алгоритмов в прикладных программах.

Авторы не ставили перед собой цель рассмотреть программную реализацию структур данных и алгоритмов в какой-либо среде программирования (эти задачи должны решаться в процессе выполнения соответствующих упражнений и проведения лабораторных занятий), а решили ограничиться изложением теоретических основ наиболее распространенных абстрактных структур данных и типовых алгоритмов, ставших классическими.

Изучение изложенного в учебнике теоретического материала, практическая реализация и экспериментальные исследования приведенных структур данных и алгоритмов в какой-либо среде программирования должны способствовать формированию соответствующих компетенций, связанных с решением следующих основных задач:

а) развитие навыков применения основных структур данных и типовых алгоритмов, их создания и обработки; определения теоретической и экспериментальной оценок вычислительной сложности алгоритмов; выбора структур данных при проектировании алгоритмов с целью повышения их эффективности;

б) выработка представления о возможностях конкретной системы программирования в плане реализации различных структур данных и об эффекте, достигаемом при применении структур и алгоритмов в программировании;

в) формирование умения правильно выбирать структуры данных при проектировании алгоритмов с целью повышения эффективности алгоритмов, реализовать их в конкретной системе программирования;

г) обеспечение получения практического опыта определения теоретической и экспериментальной оценок вычислительной сложности алгоритмов, уяснение связи сложности алгоритма со свойствами структур данных.

Список рекомендуемой литературы

1. *Ахо*, *А*. Построение и анализ вычислительных алгоритмов / А. Ахо, Д. Хопкрофт, Д. Ульман. – М. : Мир, 1979. – 536 с.

2. *Ахо*, *А*. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман. – М. : Вильямс, 2016. – 400 с.

3. *Бакнелл*, *Дж.* Фундаментальные алгоритмы и структуры данных в Delphi. – СПб. : Питер, 2006. – 560 с.

4. *Вирт*, *Н*. Алгоритмы и структуры данных. – М. : ДМК Пресс, 2010. – 272 с.

5. *Грин*, *Д*. Математические методы анализа алгоритмов / Д. Грин, Д. Кнут. – М. : Мир, 1987. – 120 с.

6. *Гудман*, *С*. Введение в разработку и анализ алгоритмов / С. Гудман, С. Хидетниеми. – М. : Мир, 1981. – 366 с.

7. *Кнут*, *Д*. Искусство программирования. Т. 1 : Основные алгоритмы. – 3-е изд. – М. : Вильямс, 2015. – 720 с.

8. *Кнут*, *Д*. Искусство программирования. Т. 3 : Сортировка и поиск. – 2-е изд. – М. : Вильямс, 2013. – 824 с.

9. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – 3-е изд. – М. : Вильямс, 2014. – 1328 с.

10. *Кормен*, *Т*. Алгоритмы. Вводный курс. – М. : Вильямс, 2014. – 208 с.

11. *Костюкова*, *Н. И*. Графы и их применение. Комбинаторные алгоритмы для программистов. – М. : Интуит, 2007. – 311 с.

12. *Красиков*, *И. В*. Алгоритмы. Просто как дважды два / И. В. Красиков, И. Е. Красикова. – 2-е изд. – М. : Эксмо, 2007. – 256 с.

13. *Кубенский*, *А. А*. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на С++. – СПб. : БХВ-Петербург, 2004. – 464 с.

14. *Левитин*, *А. В*. Алгоритмы. Введение в разработку и анализ. – М. : Вильямс, 2006. – 576 с.

15. *Липский*, *В*. Комбинаторика для программистов. – М. : Мир, 1988. – 213 с.

16. *Лэнгсам*, *Й*. Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Огенстайн, А. Тененбаум. – М. : Мир, 1989. – 567 с.

17. *Макконнелл*, *Дж*. Анализ алгоритмов. Активный обучающий подход. – М. : Техносфера, 2009. – 416 с.

18. *Макконелл*, *Дж*. Основы современных алгоритмов. – 2-е изд. – М. : Техносфера, 2004. – 368 с.

19. *Окулов*, *С. М*. Программирование в алгоритмах. – 3-е изд. – М. : Бином. Лаборатория знаний, 2007. – 383 с.

20. *Павлов*, *Л. А*. Структуры и алгоритмы обработки данных : учеб. пособие. – Чебоксары : Изд-во Чуваш. ун-та, 2008. – 252 с.

21. *Рейнгольд*, *Э*. Комбинаторные алгоритмы. Теория и практика / Э. Рейнгольд, Ю. Нивергельт, Н. Део. – М. : Мир, 1980. – 476 с.

22. *Свами*, *М*. Графы, сети и алгоритмы / М. Свами, К. Тхуласираман. – М. : Мир, 1984. – 455 с.

23. *Седжвик*, *Р*. Алгоритмы на Java / Р. Седжвик, К. Уэйн. – М. : Вильямс, 2013. – 848 с.

24. *Седжвик*, *Р*. Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных. – М. : Вильямс, 2013. – 1056 с.

25. *Скиена*, *С*. Алгоритмы. Руководство по разработке. – 2-е изд. – СПб. : БХВ-Петербург, 2011. – 720 с.

26. *Хусаинов*, *Б. С*. Структуры и алгоритмы обработки данных. Примеры на языке Си. – М. : Финансы и статистика, 2004. – 464 с.

ОГЛАВЛЕНИЕ

Леонид Александрович ПАВЛОВ,
Наталия Викторовна ПЕРВОВА

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Учебник

Издание второе,
исправленное и дополненное

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*
Ответственный редактор *Т. С. Спирина*
Корректор *Т. А. Кошелева*
Выпускающий *В. А. Иутин*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com;
196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
Тел.: (812) 412-92-72, 336-25-09.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 25.03.20.
Бумага офсетная. Гарнитура Школьная. Формат 60×90 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 16,00. Тираж 30 экз.

Заказ № 261-20.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.