Федеральное агентство морского и речного транспорта
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ МОРСКОГО И РЕЧНОГО ФЛОТА
имени адмирала С.О. МАКАРОВА»

М.Ю.Ястребов

ПРЕДСТАВЛЕНИЕ ДАННЫХ И АЛГОРИТМЫ ИХ ОБРАБОТКИ

Санкт-Петербург
2015

УДК
ББК

Рецензенты:
доктор технических наук, профессор
Н.А.Молдовян
кандидат физико-математических наук, доцент
В.О.Кузнецов

Ястребов М.Ю.
Представление данных и алгоритмы их обработки: учебное пособие. - СПб: СПГУМРФ им. адмирала С.О.Макарова, 2015. – 125 с.

Предназначено для студентов направления 090900.62 «Информационная безопасность», специальности 090303.65 «Информационная безопасность автоматизированных систем» и для углубленного изучения магистрам по направлению 090900.68 «Информационная безопасность».

Содержание соответствует рабочей программе дисциплины направления 090900.62 «Информационная безопасность».

Печатается по решению редакционно-издательского совета Государственного университета морского и речного флота им. адмирала С.О. Макарова.

УДК
ББК

ВВЕДЕНИЕ

Понятие алгоритма наряду с понятиями множества и натурального числа является исходным, интуитивно ясным понятием, не определяемым через более простые понятия. Умение разбивать какой-либо процесс на отдельные шаги, этапы — это врожденная способность человеческого интеллекта.

Под алгоритмом мы понимаем упорядоченную последовательность инструкций, которые надо выполнить для получения заданного результата. Происхождение термина связано с именем хорезмийского математика IX века аль-Хорезми (хорезмийцы — ираноязычный народ в Центральной Азии, позднее слившийся с другими народами). Аль-Хорезми описал вычислительные процедуры, связанные с решением квадратных уравнений.

Аналогичными термину «алгоритм» можно считать такие понятия, как «метод», «процедура», «программа». Важными свойствами алгоритма (или предъявляемыми к нему требованиями) являются:

1) *конечность* (процесс должен заканчиваться за конечное число шагов;

2) *однозначная определённость* при выборе каждого шага, наличие способов проверки каждого условия, используемого при формулировке инструкций;

3) *исходные данные* (ввод), которые задаются изначально, либо определяются в ходе исполнения алгоритма;

4) *выходные данные* (вывод), то есть результат исполнения алгоритма;

5) *эффективност*ь — количество времени и объём ресурсов, затрачиваемых на исполнение алгоритма не должны превышать разумные пределы.

Одним из первых в истории науки алгоритмов можно считать алгоритм Евклида, описывающий последовательность действий при нахождении наибольшего общего делителя двух натуральных чисел путём многократного деления с остатком [19]. В состав алгоритма могут входить «цельными блоками», «подпрограммами» другие алгоритмы (например, алгоритм деления столбиком при осуществлении алгоритма Евклида).

Оценка количественных характеристик эффективности алгоритмов, исполняемых компьютерами, относится к *анализу алгоритмов*.

Способы описания алгоритмов могут быть различными: словесный, с помощью блок-схемы, в виде компьютерной программы, косвенный (в виде примера его выполнения).

Глава 1. НЕОБХОДИМЫЕ МАТЕМАТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Понятия комбинаторики

Факториалы. Числом $n!$ (читается: «n-факториал») называется произведение натуральных чисел от 1 до n:

$1!=1;\ 2!=1\cdot 2=2;$ далее, при $n\geq 3$: $n!=1\cdot 2\cdot ...\cdot n$. Дополнительно полагают $0!=1$ (это позволяет сохранить единообразие многих формул с участием факториалов).

Факториалы соседних натуральных чисел связаны рекуррентным соотношением: $(n+1)!=n!\cdot(n+1)$.

Принцип умножения. Принцип умножения (произведения) задает правило для подсчёта количества различных наборов из r элементов в случае, когда последние выбираются по одному из r конечных множеств. Благодаря этому принципу подсчет количества вариантов во многих случаях приводит к большим числам.

Если для пары (a,b) первый член может быть выбран из k элементов $\{a_1,a_2,...,a_k\}$, а второй — из n элементов $\{b_1,b_2,...,b_n\}$, то общее количество таких пар (a_i,b_j) равно произведению kn.

В общем случае, если строится набор из r элементов, причем первый член может быть выбран k_1 способами, второй — k_2 способами, и т.д., наконец, последний — k_r способами, то общее количество r-членных наборов равно произведению $k_1k_2...k_r$.

Для случая пар принцип умножения иллюстрируется (по аналогии с матрицами) прямоугольной таблицей, в которой пара (a_i,b_j) стоит на пересечении i-й строки и j-го столбца.

Перестановки. Перестановкой из n элементов $a_1,a_2,...,a_n$ называется их расположение в определенном порядке:

$$(a_{i_1},a_{i_2},...,a_{i_n}).$$

Две перестановки считаются различными, если хотя бы один элемент занимает в них разные позиции.

Пример. Все перестановки из цифр 1, 2, 3:

(1,2,3); (1,3,2); (2,1,3); (2,3,1); (3,1,2); (3,2,1).

Число перестановок из n элементов принято обозначать через P_n. Для числа перестановок справедлива формула: $P_n=n!$.

Размещения. Размещением из n различных элементов по k элементов $(0\leq k\leq n)$ называется упорядоченный набор каких-либо k из этих элементов.

Два размещения из n по k считаются различными, если они разли-

чаются составом и/или порядком следования входящих в них элементов.

Пример. Все размещения из трех элементов — цифр 1, 2, 3 по два:

(1,2); (2,1); (1,3); (3,1); (2,3); (3,2).

Число размещений из n по k принято обозначать через A_n^k. Для числа размещений справедлива формула:

$$A_n^k = \frac{n!}{(n-k)!} = n(n-1)(n-2)...(n-k+1) \ .$$

Если $k = n$, то размещение из n по n является перестановкой из n элементов. Обе формулы (1) и (2) дают в этом случае одинаковый результат: $A_n^n = \frac{n!}{0!} = \frac{n!}{1} = P_n$.

Сочетания. Сочетанием из n различных элементов по k элементов $(0 \le k \le n)$ называется набор каких-либо k из этих элементов без учета порядка их следования.

Два сочетания из n по k считаются различными, если они различаются составом входящих в них элементов.

Пример. Все сочетания из пяти элементов – цифр 1, 2, 3, 4, 5 по два:

(1,2); (1,3); (1,4); (1,5); (2,3); (2,4); (2,5); (3,4); (3,5); (4,5).

Число сочетаний из n по k принято обозначать через C_n^k или через $\binom{n}{k}$. Для числа сочетаний справедлива формула:

$$C_n^k = \frac{A_n^k}{k!} = \frac{n!}{k!(n-k)!} \ .$$

Числа C_n^k называют *биномиальными коэффициентами*, поскольку они участвуют в разложении бинома (двучлена) $(x+y)^n$ по степеням x (бином Ньютона):

$$(x+y)^n = \sum_{i=0}^{n} C_n^i \cdot x^i y^{n-i} = = y^n + C_n^1 \cdot xy^{n-1} + C_n^2 \cdot x^2 y^{n-2} + ... + C_n^n \cdot x^n \ .$$

Частными случаями разложения бинома являются хорошо известные формулы для квадрата и куба суммы.

С биномиальными коэффициентами связано большое количество тождеств, в частности:

$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$; $C_n^k = C_n^{n-k}$; $\sum_{k=0}^{n} C_n^k = 2^n$; $C_n^k = \frac{n}{k} C_{n-1}^{k-1}$ и др. [7].

1.2. Подстановки и инверсии

Связь подстановок с перестановками. *Подстановкой* совокупности $x = (a_1 a_2 ... a_n)$ называётся её взаимно-однозначное отображение на себя (биекция), при котором элемент a_j переходит в некоторый элемент

a_{i_j}. Подстановка описывается таблицей из двух строк, вторая из которых является перестановкой исходных элементов $(a_{i_1}\, a_{i_2} \ldots a_{i_n})$ и показывает, в какие элементы переходят соответствующие элементы верхней строки.

Обычно в качестве элементов a_i рассматриваются сами натуральные числа i. Таким образом, результат действия подстановки

$$\tau = \begin{pmatrix} 1 & 2 & \ldots & n \\ i_1 & i_2 & \ldots & i_n \end{pmatrix} \qquad (1)$$

описывается перестановкой

$$\tau = (i_1 \;\; i_2 \;\ldots\; i_n), \qquad (2)$$

для которой используется то же обозначение. Задание подстановки таблицей (1) не зависит от порядка следования столбцов.

Пример. Подстановка $\begin{pmatrix} 1\,2\,3\,4\,5\,6 \\ 2\,1\,3\,5\,6\,4 \end{pmatrix}$ переводит:

$$1 \to 2,\; 2 \to 1,\; 3 \to 3,\; 4 \to 5,\; 5 \to 6,\; 6 \to 4.$$

Количество подстановок множества из n элементов совпадает с количеством его перестановок $n!$. Подстановку (1) часто отождествляют с задающей её перестановкой (2).

Инверсии. *Инверсией* подстановки (или задающей её перестановки) называется всякая пара натуральных чисел (i_j, i_k), для которой $1 \le j < k \le n$, и при этом $i_j > i_k$ (то есть больший номер предшествует меньшему).

Пример. Перестановка $(1\,2\,3\,4\,5)$ содержит нуль инверсий; перестановка $(1\,3\,2\,4\,5)$ содержит одну инверсию $(3,2)$; перестановка $(5\,4\,3\,2\,1)$ содержит $4+3+2+1=10$ инверсий; перестановка $(4\,2\,5\,3\,1)$ содержит $3+1+2+1=7$ инверсий.

Подстановка называется *чётной*, если она содержит чётное число инверсий. В противном случае она называется *нечётной*.

Транспозицией называется подстановка, меняющая местами два элемента и оставляющая остальные на месте.

Транспозиция двух соседних элементов меняет чётность перестановки на противоположную, поскольку число инверсий при этом изменяется (уменьшается или увеличивается) на единицу. Это же имеет место и при транспозиции любых двух элементов i_j и i_k. Действительно, если между i_j и i_k находится m элементов, то транспозицию можно получить, последовательно переставляя $2m+1$ раз соседние элементы; при этом чётность «перебросится» нечётное число раз. Например, перестановка $(2\,3\,5\,4\,1)$ является чётной, поскольку содержит 4 инверсии: $(2\,1)$, $(3\,1)$, $(4\,1)$, $(5\,1)$, $(5\,4)$; транспозиция, меняющая местами 1 и 3, приводит к перестановке $(2\,1\,5\,4\,3)$, которая содержит 3 инверсии и является поэтому нечётной.

Любую перестановку можно получить из начальной (чётной) перестановки $(1,2,...,n)$ путём последовательного выполнения транспозиций. Это можно сделать разными способами, но всех случаях чётная перестановка получается после чётного числа транспозиций, а нечётная — после нечётного.

Таблица инверсий. *Таблицей инверсий* перестановки $(i_1\ i_2\ ...\ i_n)$ называется упорядоченная совокупность n чисел $[b_1, b_2, ..., b_n]$, где b_j — число элементов слева от j, которые больше, чем j (то есть образуют с ним инверсию).

Пример. Таблица инверсий перестановки

$$(4\,2\,5\,3\,1\,7\,9\,6\,8) \qquad (3)$$

имеет вид

$$[4\,1\,2\,0\,0\,2\,0\,1\,0]. \qquad (4)$$

Действительно, слева от 1 расположены четыре элемента $4, 2, 5, 3$, образующие с 1 инверсию; слева от 2 инверсию образует единственный элемент 4; слева от 3 находятся два инверсных с ним элемента 4 и 5, и т.д.

Сумма элементов таблицы инверсий равна общему числу инверсий в перестановке. Элементы таблицы удовлетворяют условиям:

$$0 \le b_1 \le n-1;\ 0 \le b_2 \le n-2;\ ...\ 0 \le b_{n-1} \le 1;\ b_n = 0. \qquad (5)$$

Перестановка однозначно восстанавливается по своей таблице инверсий. Для этого последовательно определяется относительное расположение элементов — сначала элемента n, затем $n-1$, и так далее, вплоть до 1.

Пример. Восстановим перестановку (3) по её таблице инверсий (4). Начнём с наибольшего элемента 9. Поместим 8 справа от него, поскольку 8 может образовывать инверсию только с 9, и эта инверсия присутствует из-за $b_8 = 1$:

$$(9\,8).$$

Далее, 7 расположим левее и 8, и 9, так как $b_7 = 0$:

$$(7\,9\,8).$$

Условие $b_6 = 2$ означает, что левее 6 находятся два из бóльших его элементов, поэтому 6 расположим между 9 и 8:

$$(7\,9\,6\,8).$$

Поскольку $b_5 = 0$, левее элемента 5 нет бóльших его (и расположенных ранее) элементов:

$$(5\,7\,9\,6\,8).$$

Аналогично, 4 поместим левее всех уже расположенных элементов ввиду условия $b_4 = 0$:

$$(4\,5\,7\,9\,6\,8).$$

Левее 3 должны находиться два (так как $b_3 = 2$) из уже расположенных элементов:

$$(4\,5\,3\,7\,9\,6\,8).$$

Левее 2 должен находиться один (так как $b_2 = 1$) из уже расположенных элементов:

$$(4\,2\,5\,3\,7\,9\,6\,8).$$

Наконец, элементу 1 должны предшествовать четыре из уже расположенных элементов:

$$(4\,2\,5\,3\,1\,7\,9\,6\,8).$$

Заметим, что элементы таблицы инверсий не зависят друг от друга, подчиняясь только ограничениям (5). В отличие от этого, элементы перестановки должны быть различными.

Произведение подстановок. *Произведением подстановок* $\sigma = \begin{pmatrix} 1 & 2 & \dots & n \\ i_1 & i_2 & \dots & i_n \end{pmatrix}$ и $\tau = \begin{pmatrix} i_1 & i_2 & \dots & i_n \\ j_1 & j_2 & \dots & j_n \end{pmatrix}$ называется подстановка $\sigma\tau$, получающаяся их последовательным применением, причём первым применяется правый множитель τ, так что $(\sigma\tau)(x)=\sigma(\tau(x))$.

Пример. Произведением двух подстановок $\sigma = \begin{pmatrix} 1\,2\,3\,4\,5\,6 \\ 3\,5\,1\,4\,2\,6 \end{pmatrix}$ и $\tau = \begin{pmatrix} 1\,2\,3\,4\,5\,6 \\ 2\,6\,3\,4\,1\,5 \end{pmatrix}$ является подстановка $\sigma\tau = \begin{pmatrix} 1\,2\,3\,4\,5\,6 \\ 5\,6\,1\,4\,3\,2 \end{pmatrix}$; при этом осуществляются переходы:

$$1 \xrightarrow[\tau]{} 2 \xrightarrow[\sigma]{} 5;\quad 2 \xrightarrow[\tau]{} 6 \xrightarrow[\sigma]{} 6;\quad 3 \xrightarrow[\tau]{} 3 \xrightarrow[\sigma]{} 1;$$
$$4 \xrightarrow[\tau]{} 4 \xrightarrow[\sigma]{} 4;\quad 5 \xrightarrow[\tau]{} 1 \xrightarrow[\sigma]{} 3;\quad 6 \xrightarrow[\tau]{} 5 \xrightarrow[\sigma]{} 2.$$

Пусть

$$\sigma = \begin{pmatrix} 1\,2\,3\,4\,5\,6 \\ 3\,5\,1\,4\,2\,6 \end{pmatrix},\ \tau = \begin{pmatrix} 1\,2\,3\,4\,5\,6 \\ 2\,6\,3\,4\,1\,5 \end{pmatrix}.$$

Тогда

$$\sigma\tau = \begin{pmatrix} 1\,2\,3\,4\,5\,6 \\ 3\,1\,2\,4\,6\,5 \end{pmatrix}.$$

При этом:

$$1 \xrightarrow[\sigma]{} 3 \xrightarrow[\tau]{} 3;\quad 2 \xrightarrow[\sigma]{} 5 \xrightarrow[\tau]{} 1;\quad 3 \xrightarrow[\sigma]{} 1 \xrightarrow[\tau]{} 2;$$
$$4 \xrightarrow[\sigma]{} 4 \xrightarrow[\tau]{} 4;\quad 5 \xrightarrow[\sigma]{} 2 \xrightarrow[\tau]{} 6;\quad 6 \xrightarrow[\sigma]{} 6 \xrightarrow[\tau]{} 5.$$

Нетрудно проверить, что *произведение подстановок не обладает свойством коммутативности*: $\sigma\tau \neq \tau\sigma$. В то же время свойство ассоциативности имеет место: $(\sigma\tau)\rho = \sigma(\tau\rho)$.

Представление подстановок циклами. Пусть подстановка осуществляет переходы: $i_\alpha \to i_\beta$, $i_\beta \to i_\gamma$,.... Для некоторого i_δ в этой последовательности мы обязательно будем иметь $i_\delta \to i_\alpha$, поскольку переходящих друг в друга элементов может быть не более n, и все они различны. В результате элементы $i_\alpha, i_\beta, ..., i_\delta$ циклически переходят друг в друга.

Подстановка $(i_\alpha\, i_\beta \ldots i_\delta)$, в которой элементы $i_\alpha, i_\beta, ..., i_\delta$ циклически переходят друг в друга, а остальные остаются на своих местах, называется *циклом*.

Пример. Подстановка $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 2 & 5 & 6 & 1 & 4 \end{pmatrix}$ является произведением циклов (135) и (46): $\sigma = (135)(46)$. Элемент 2, не вошедший ни в один цикл, переходит в себя.

Таким образом, *всякая подстановка разлагается в произведение циклов*.

Обратная подстановка. Подстановка τ называется *обратной* к подстановке σ, если их произведение $\tau\sigma$ даёт тождественную подстановку $\begin{pmatrix} 1 & 2 & \ldots & n \\ 1 & 2 & \ldots & n \end{pmatrix}$.

Обозначение: $\tau = \sigma^{-1}$. Для получения обратной подстановки достаточно поменять местами её строки (и переупорядочить по возрастанию чисел верхней строки). Так, для подстановки σ из последнего примера обратная подстановка

$$\sigma^{-1} = \begin{pmatrix} 3 & 2 & 5 & 6 & 1 & 4 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 2 & 1 & 6 & 3 & 4 \end{pmatrix}.$$

1.3. Показатели роста функций

Характеристики, используемые при анализе алгоритмов, как правило, описываются возрастающими положительными функциями натурального аргумента n. Они выражают необходимые затраты машинного времени, объём используемой оперативной и/или внешней памяти, число итераций в циклических процедурах и т.п. Параметр n (или функция от него) при этом характеризует объём входных данных, обрабатываемых алгоритмом.

Например, при обработке множества всех перестановок из n элементов объём входных данных равен (или пропорционален) $n!$. При работе с квадратными матрицами n-го порядка объём входных данных задаётся числом n^2. При упорядочивания списка из n записей объём входных данных определяет само число n.

Если для последовательностей $x_n = f(n)$ и $y_n = g(n)$ при всех n выполняется неравенство

$$|x_n| \le C|y_n|, \tag{6}$$

то это выражается записью: $x_n = O(y_n)$ или $f(n) = O(g(n))$ (говорят, что «x_n есть О *большое от* y_n»)

В частности, если для бесконечно больших (стремящихся к бесконечности) последовательностей существует конечный предел отношения: $\lim(x_n / y_n) = a$, то последовательность $z_n = x_n / y_n$ является ограниченной величиной: $|x_n / y_n| \le C = const$, так что $x_n = O(y_n)$.

Значение константы C зависит от конкретных функций f и g.

Пример. Пусть $f(n) = a_0 n^k + a_1 n^{k-1} + ... + a_{k-1} n^1 + a_n$ — произвольный многочлен степени k. Примем $g(n) = n^k$. Имеем:

$$f(n) = n^k \left(a_0 + \frac{a_1}{n} + ... + \frac{a_{k-1}}{n^{k-1}} + \frac{a_k}{n^k} \right).$$

Поскольку $a_0 + \frac{a_1}{n} + ... + \frac{a_{k-1}}{n^{k-1}} + \frac{a_k}{n^k} \xrightarrow[n\to\infty]{} a_0$, то в силу ограниченности сходящейся последовательности

$$\left| a_0 + \frac{a_1}{n} + ... + \frac{a_{k-1}}{n^{k-1}} + \frac{a_k}{n^k} \right| \le C_1 \;\Rightarrow\; |f(n)| \le C_1 |n^k| \;\Rightarrow$$

$$\Rightarrow\; f(n) = O(n^k).$$

Таким образом, *характер роста значений многочлена от натурального аргумента определяется его старшей степенью.*

Формулы, содержащие символ O называются *асимптотическими*. Они описывают приближенно рост значений функции f в сопоставлении с какой-либо функцией g, принимаемой в качестве стандарта.

1.4. Числа Фибоначчи

Числа Фибоначчи — эта последовательность F_n, задаваемая рекуррентной формулой: $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$ при $n \ge 2$.

Начальный отрезок этой последовательности:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,$$
$$1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025$$

Числа Фибоначчи используются при анализе многих алгоритмов, реализующих различные не связанные между собой задачи. Они участвуют во многих тождествах. В частности:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \tag{7}$$

$$F_{n+1}F_{n-1}-(F_n)^2=(-1)^n; \quad (8)$$

$$\text{НОД}(F_n,F_{n+1})=1; \quad (9)$$

$$F_{n+m}=F_mF_{n+1}+F_{m-1}F_n; \quad (10)$$

$$n\,|\,m \Rightarrow F_n\,|\,F_m; \quad (11)$$

$$\text{НОД}(F_m,F_n)=F_{\text{НОД}(m,n)}. \quad (12)$$

1.5. Системы счисления

Позиционные и непозиционные системы счисления. *Системой счисления в широком смысле называют всякий систематический способ записи чисел с помощью ограниченного числа символов, называемых цифрами и являющихся обозначениями некоторых натуральных чисел.*

В привычном способе записи чисел с помощью десятичных цифр значение каждой цифры существенным образом зависит от её позиции в записи числа. Например, в записи 3033 первая слева цифра «3» указывает количество тысяч, третья слева – количество десятков и, наконец, крайняя справа – количество единиц. Способы записи чисел, обладающие указанным свойством, называют *позиционными системами счисления*.

Напротив, например, при записи чисел так называемыми римскими цифрами значение последних не зависит от позиции, которую они занимают. Например, цифра «V», служащая обозначением числа пять, имеет одинаковое значение в записях чисел: IV (четыре), XVI (шестнадцать), XXV (двадцать пять).

Способы записи чисел, в которых значения используемых цифр не зависят от их позиции в записи числа, называют *непозиционными системами счисления.*

Очевидными недостатками непозиционных систем счисления, в частности, римских цифр, являются быстро растущая длина записи чисел и невозможность сведения арифметических действий к простым алгоритмам преобразования цифр.

Десятичная запись рациональных чисел подразумевает представление их в виде суммы степеней числа *десять* с коэффициентами — десятичными цифрами. Так, запись

$$n=4018 \quad (13)$$

означает:

$$n=4\cdot10^3+0\cdot10^2+1\cdot10^1+8\cdot10^0.$$

Аналогично, запись $n=5.92$ означает:

$$n=5\cdot10^0+9\cdot10^{-1}+2\cdot10^{-2}.$$

Поэтому в вычислительном устройстве, использующем десятичное представление, элемент, который представляет отдельный разряд, должен иметь десять различных устойчивых состояний. Эти состояния как раз и выражают десятичные цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Например, в случае конторских счет таким элементом является проволока с откинутым влево

определенным количеством косточек. В арифмометре отдельный разряд реализован колесиком, могущим быть повернутым на десять различных углов (всякий раз при повороте в окошке видна та или другая цифра).

Для выполнения арифметических операций необходимо знать таблицы сложения и умножения десятичных цифр (их запоминают в начальной школе).

По-видимому, только привычность записей вида (13) удерживает от двух вполне естественных вопросов. Во-первых, любое ли натуральное число можно представить в виде (13)? И, во-вторых, является ли такое представление единственным?

Позиционные системы счисления с произвольным основанием. Ответ на поставленные вопросы дает следующая теорема:

При любом фиксированном натуральном числе $p>1$ *всякое натуральное число* n *представимо, и притом единственным образом, в виде*

$$n = a_k \cdot p^k + a_{k-1} \cdot p^{k-1} + ... + a_1 \cdot p^1 + a_0 \cdot p^0, \qquad (14)$$

где $a_k \neq 0$, $0 \le a_i \le p-1$ $(i = 0,...,k)$.

Представлению числа n в форме (14) отвечает его запись в p-ичной системе счисления в виде $(a_k a_{k-1}...a_0)$; здесь каждый из символов a_i является обозначением соответствующего целого числа, лежащего между 0 и $p-1$, и называется *цифрой* p*-ичного представления.*

Таким образом, записанное в троичной системе число $(1022)_3$ обозначает: $1 \cdot 3^3 + 0 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0$.

В системе счисления с основанием, большим десяти, понадобятся дополнительные цифры – символы для обозначения числа десять и следующих за ним вплоть до числа, на единицу меньшего основания системы.

Перевод чисел в другую систему счисления. Для перевода в привычную десятичную систему счисления числа n, записанного первоначально в p-ичной системе в виде $(a_k a_{k-1}...a_0)_p$, следует:

- записать его в виде суммы степеней с основанием p согласно (14);
- произвести соответствующие арифметические действия в десятичной системе.

Примеры. 1. $(4031)_5 = 4 \cdot 5^3 + 0 \cdot 5^2 + 3 \cdot 5^1 + 1 \cdot 5^0 = (616)_{10}$.

2. $(11001)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (25)_{10}$.

Для перевода числа n, первоначально записанного в десятичной системе, в систему с основанием p, нужно определить коэффициенты $a_k, a_{k-1}, ..., a_0$ в представлении (14). При этом цифра a_0 (“число единиц”) получается как остаток от деления исходного числа n на p: $n = n_1 p + a_0$, где

$$n_1 = a_k p^{k-1} + a_{k-1} p^{k-2} + ... + a_2 p + a_1.$$

Далее, цифра a_1 получается как остаток от деления первого неполного частного n_1 на p: $n_1 = n_2 p + a_1$, где

$$n_2 = a_k p^{k-2} + a_{k-1} p^{k-3} + \ldots + a_2,$$ и т.д.

Последовательные деления заканчиваются, когда очередное неполное частное окажется строго меньше p. Именно оно и является первой слева p-ичной цифрой a_k числа n.

Пример. Переведем в систему счисления с основанием 6 десятичное число $n = (231)_{10}$. Проводя последовательные деления на 6 с остатком, получаем:

$$231 = 38 \cdot 6 + 3; \quad 38 = 6 \cdot 6 + 2; \quad 6 = 1 \cdot 6 + 0;$$

Неполное частное 1 оказалось строго меньше основания 6; поэтому крайняя слева цифра шестиричного представления равна 1, а далее следуют взятые в порядке, обратном их вычислению, остатки последовательных делений: 0, 2 и 3, так что

$$(231)_{10} = (1023)_6.$$

Двоичная система счисления. Полезно помнить таблицу начальных степеней основания 2:

n	0	1	2	3	4	5	6	7	8	9	10
2^n	1	2	4	8	16	32	64	128	256	512	1024

Двоичными цифрами являются символы 0 и 1.

Примеры:

1. $(10011)_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = (19)_{10}$;

2. $(324)_{10} = 162 \cdot 2 + 0$; $162 = 81 \cdot 2 + 0$; $81 = 40 \cdot 2 + 1$; $40 = 20 \cdot 2 + 0$; $20 = 10 \cdot 2 + 0$; $10 = 5 \cdot 2 + 0$; $5 = 2 \cdot 2 + 1$; $2 = 1 \cdot 2 + 0 \quad \Rightarrow (324)_{10} = (101000100)_2$.

Сложение двоичных чисел осуществляется согласно *таблице сложения* в отдельном двоичном разряде $a + b$ для слагаемых a и b с привлечением переноса из предыдущего (младшего) разряда m и переноса в последующий (старший) разряд s:

m	a	b	$a+b$	s
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Умножение в отдельном двоичном разряде осуществляется согласно таблице умножения (переносы из разряда в разряд отсутствуют):

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

Восьмеричная система счисления. Цифрами системы с основанием восемь являются символы 0, 1, 2, 3, 4, 5, 6, 7.

Примеры.

$$(603)_8 = 6\cdot 8^2 + 0\cdot 8^1 + 3\cdot 8^0 = (387)_{10};$$

$$(95)_{10} = 1\cdot 8^2 + 3\cdot 8^1 + 7\cdot 8^0 = (137)_8.$$

Приведем таблицу двоичного представления восьмеричных цифр, отводя на каждую из них по три двоичных разряда и дописывая при необходимости нули впереди:

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Тройки цифр в правом столбце называют *двоичными триадами*. Таким образом, каждой восьмеричной цифре взаимно однозначно соответствует двоичная триада.

Установим связь между восьмеричным и двоичным представлениями числа.

Пусть $c = a_m$ - восьмеричная цифра, стоящая в восьмеричном представлении множителем при $8^m = (2^3)^m = 2^{3m}$, и ей соответствует двоичная триада $(\alpha\beta\gamma)$. Тогда соответствующее слагаемое в представлении (3) преобразуется к виду:

$$c\cdot 8^m = (\alpha\cdot 2^2 + \beta\cdot 2^1 + \gamma\cdot 2^0)\cdot 2^{3m} = \alpha\cdot 2^{3m+2} + \beta\cdot 2^{3m+1} + \gamma\cdot 2^{3m}.$$

Поэтому восьмеричной цифре c в разряде m соответствуют три цифры двоичного разложения в разрядах $3m+2$, $3m+1$, $3m$, причём именно цифры двоичной триады для c.

Итак, для перехода от восьмеричного представления числа к двоичному достаточно заменить каждую восьмеричную цифру соответствующей двоичной триадой.

Примеры.

$(370)_8 = (011\grave{}111\grave{}000)_2 = (11111000)_2$.

$(10101)_2 = (010\grave{}101)_2 = (25)_8$.

$(1)_2 = (001)_2 = (1)_8$.

Шестнадцатеричная система счисления. Цифрами этой системы наряду с $0,1,...,9$ являются также индивидуальные символы для остальных натуральных чисел, меньших основания шестнадцать. Принято использовать для этой цели начальные латинские буквы в соответствии с таблицей:

10	11	12	13	14	15
A	*B*	*C*	*D*	*E*	*F*

Примеры перевода:

$(2A01)_{16} = 2\cdot16^3 + 10\cdot16^2 + 0\cdot16^1 + 1\cdot16^0 = (10753)_{10}$;

$(199)_{10} = 12\cdot16^1 + 7\cdot16^0 = C\cdot16^1 + 7\cdot16^0 = (C7)_{16}$.

Приведем теперь таблицу двоичного представления шестнадцатеричных цифр, отводя на каждую из них по четыре двоичных разряда и дописывая при необходимости нули впереди:

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Четверки цифр в правом столбце называют *двоичными тетрадами*. Таким образом, каждой шестнадцатеричной цифре соответствует двоичная тетрада.

Связь между двоичным и шестнадцатеричным представлениями числа аналогична связи между двоичным и восьмеричным представлениями — с той разницей ($8=2^3$; $16=2^4$), что теперь каждой шестнадцатеричной цифре соответствует двоичная тетрада.

Пример.

$(C5)_{16}=(1100\grave{}0101)_2=(11000101)_2$;

$(111011)_2=(0011\grave{}1011)_2=(3B)_{16}$.

Как правило, в компьютерах адреса элементов памяти записываются в шестнадцатеричной системе.

Алгоритм перебора подмножеств. Двоичная запись чисел и двоичная арифметика дают удобный алгоритм перебора всех подмножеств данного множества из n элементов. Пусть элементы множества A перенумерованы:

$$A=\{a_1,a_2,...,a_n\}.$$

Свяжем с произвольным подмножеством набор из n нулей и единиц, записывая на i-м месте единицу, если элемент a_i входит в подмножество, и записывая нуль в противном случае. Например, для множества из четырех элементов подмножеству $\{a_2,a_4\}$ соответствует набор (0101); всему множеству A соответствует набор (1111), пустому подмножеству соответствует набор (0000).

Теперь можно последовательно получить наборы для всех подмножеств, начав с нуля в n-разрядной двоичной записи (00...0) и последовательно прибавляя по единице, причем результат записывая снова в виде n-разрядного двоичного числа.

Попутно получаем, что число всех подмножеств для множества из n элементов равно 2^n. Действительно, в соответствии с комбинаторным *принципом умножения* общее количество наборов из n элементов, в которых на каждый позиции может быть одно из двух значений, равно 2^n.

1.6. Округляющие функции

1.6.1. Функции целой и дробной части

Для произвольного вещественного числа x его *целой частью* $[x]$ называется наибольшее целое число, не превосходящее x.

Примеры.

$[11]=-11$; $[0]=0$; $[8]=8$; $[12{,}5]=12$; $[-9.33]=-10$; $[\pi]=3$; $[e]=2$.

При всех x выполняется неравенство $x-1<[x]\le x$. При этом $[x]=x\Leftrightarrow x\in\mathbb{Z}$.

Для произвольного вещественного числа x его *дробной частью* $\{x\}$ называется разность $x-[x]$.

Примеры.

$\{-11\}=0;\ \{-9.33\}=0,67;\ \{0\}=0;\ \{8\}=0;\ \{12,5\}=0,5;$

$\{\pi\}=0,14159...;\ \{e\}=0,718281828459045....$

Равенство $\{x\}=x$ означает, что $x\in[0,1)$.

Функция $y=\{x\}$ является периодической с периодом 1, непрерывной справа на всей числовой оси, а также непрерывной на промежутках $[n,n+1),\ n\in\mathbb{Z}$.

1.6.2. Функции «пол» и «потолок»

Для целой части числа x используется также более распространенное в последнее время обозначение $\lfloor x\rfloor$ и название «пол» (англ. floor — в смысле ограничения снизу). Наряду с функцией «пол» используется также функция «потолок» (англ. ceiling): $\lceil x\rceil$ — наименьшее целое, большее или равное x. Например, $\lceil -3\rceil=-3;\ \ \lceil 5\rceil=5;\ \lceil 15/7\rceil=3;\ \lceil -6/5\rceil=-1;$ $\lceil 6/5\rceil=2$.

Функцию «потолок» можно определить также следующим образом: $\lceil x\rceil=n$, если для целого числа n выполняется неравенство $n-1<x\le n$. Если x является целым числом, то $\lceil x\rceil=x=\lfloor x\rfloor$, в остальных случаях $\lfloor x\rfloor<x<\lceil x\rceil$.

Для натуральных чисел a и b справедливо неравенство

$$\lceil a/b\rceil\le(a+b-1)/b. \qquad (15)$$

Например, $\lceil 17/3\rceil=6\le(17+3-1)/3=19/3$.

Глава 2. ИНФОРМАЦИОННЫЕ СТРУКТУРЫ

2.1. Линейные списки

Линейный однонаправленный список — это совокупность данных одного типа (*элементов списка,* или *узлов списка*), связанных между собой отношением строгого порядка, определяющим следование элементов друг за другом. Требование упорядоченности обусловлено тем, что с каждым элементом линейного списка при хранении связан конкретный физический адрес, и задаётся правило перехода от очередного элемента к следующему.

Элемент списка может иметь вторичную структуру, состоящую из отдельных полей, представляющих разные типы данных (целые числа, вещественные числа, строки символов, логические значения «истина»/«ложь» $\Leftrightarrow$ «1» /«0» и т.д.). Такие списки часто называют *файлами*.

Количество элементов списка (его длина) задаётся целым неотрицательным числом n ($n=0$ означает, что список пуст). Элемент списка, имеющий номер k, будем обозначать $x[k]$.

Операции с линейными списками. С линейными списками могут выполняться следующие операции (их следует отличать от операций над значениями элементов):

1. Получение доступа к элементу $x[k]$ по известному номеру k. Время доступа напрямую зависит от типа памяти, в котором хранится список. Память компьютера разделяется на:

- память *с произвольным доступом* (RAM);
- память *с доступом только для чтения* (ROM);
- внешнюю память для хранения больших объёмов информации;
- *ассоциативную* память (CAM), в которой доступ осуществляется не по адресу, а по содержанию (значению).

2. Вставка нового элемента перед или после $x[k]$.

3. Удаление элемента $x[k]$.

4. Объединение списков с охранением порядка следования элементов (конкатенация).

5. Разбиение списка на части.

6. Дублирование списка.

7. Определение длины списка.

8. Сортировка, то есть изменение порядка следования элементов в порядке возрастания значений какого-либо поля (*ключа*), входящего в структуру элементов списка. Если ключ сортировки имеет численное значение p, то сортировка по убыванию p равносильна сортировке по возрастанию $-p$: ($s<t \Leftrightarrow -s>-t$).

Способ представления списков в памяти компьютера, как и тип памяти, выбираются с учётом тех из перечисленных выше операций, которые применяются наиболее часто.

Линейные списки с оконечным доступом. Линейные списки, в которых доступ к значению элемента, удаление и вставка элементов производятся на одном из концов, разделяются на следующие типы:

1. *Стек* — список, в котором операции доступа, вставки и удаления производятся только на одном фиксированном «верхнем» конце списка (рис. 1).

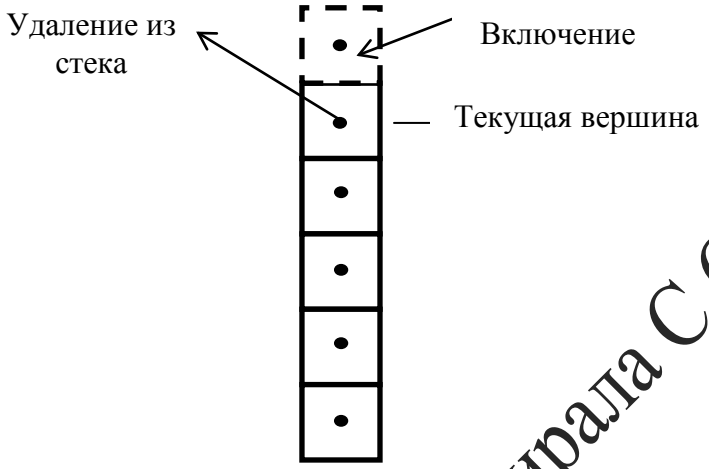


Рис. 1.

Дисциплина обслуживания стека описывается термином «последний пришёл – первый ушёл»; по-английски «last in – first out» (*LIFO*).

2. *Односторонняя очередь*, или просто *очередь*, — список, в котором вставка элементов выполняется на одном конце списка («постановка в конец очереди»), а доступ и удаление («после обслуживания») — на другом конце (рис. 2).

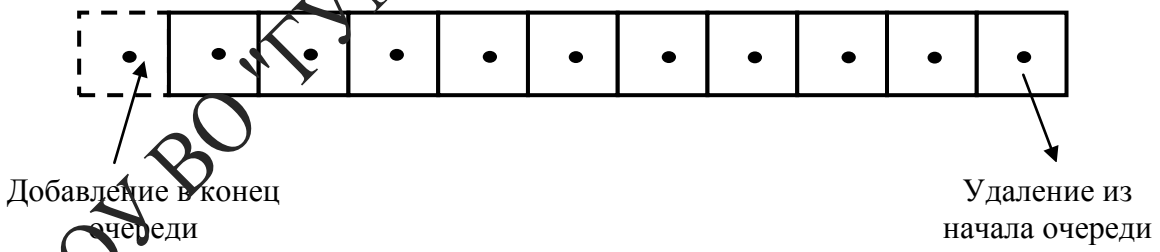


Рис. 2.

Дисциплина обслуживания односторонней очереди описывается как «первый пришёл – первый ушёл»; по-английски «first in – first out» (*FIFO*).

3. *Дек*, или *двусторонняя очередь*, (double-ended queue) — список, в котором доступ, вставка и удаление могут выполняться на обоих концах списка. Дополнительно различают *деки с ограниченным вводом* и *деки с ограниченным выводом*, в которых указанная операция (в отличие от противоположной) может выполняться только на одном из концов (рис. 3).

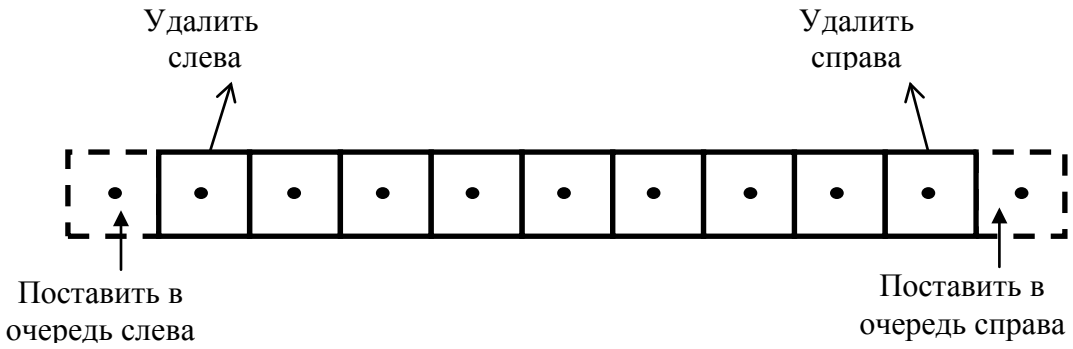


Рис. 3.

2.2. Хранение линейных списков

Последовательное распределение памяти. Естественный способ хранения линейного списка в памяти компьютера заключается в расположении элементов в непосредственно следующих друг за другом участках памяти, когда элемент $x[k+1]$ располагается непосредственно за $x[k]$.

Будем обозначать через $loc(z)$ адрес объекта z в памяти компьютера. Если элемент списка занимает c адресуемых единиц памяти (ячеек, байтов, слов), то $loc(x[k+1]) = loc(x[k]) + c$.

Начало списка задаётся *базовым адресом* l_0 таким образом, что

$$loc(x[k]) = l_0 + c(k-1). \qquad (16)$$

Последовательное распределение памяти при работе со стеком. Со стеком связывается переменная t, называемая *указателем стека*. В случае пустого стека указателю присваивается нулевое значение: $t := 0$.

Вставка нового элемента y реализуется следующими действиями:

$$t := t+1;\ \ x[t] := y.$$

Удаление элемента из непустого стека (когда $t \ge 1$) реализуется в два шага: отправка верхнего элемента на обработку в переменную y и уменьшение значения указателя на единицу и:

$$y := x[t];\ t := t-1.$$

Последовательное распределение памяти при работе с очередью. Представление очереди реализуется связыванием со списком уже двух указателей f и r — для начала и конца очереди, соответственно; при этом у пустого списка $f = r = 0$, и всегда $f \le r$,.

Вставка нового элемента y в конец очереди реализуется следующими действиями:

$$r := r+1;\ \ x[r] := y.$$

При удалении элемента из начала непустой очереди производится присваивание

$$f := f+1.$$

В результате память, занятая ранее этим элементом, учитывается операционной системой как свободная. Если перед удалением было $f = r$, то есть единственный элемент очереди был и первым, и последним, то $f := 0;\ r := 0$, и список становится пустым.

Если указанным способом производятся сначала вставка элемента в конец очереди, а затем удаление элемента из её начала, то, хотя длина очереди в результате оказывается прежней, все адреса увеличиваются на c. При большом числе повторений они могут превзойти допустимый верхний предел (переполнение памяти). Поэтому такая организация вставок/удалений приемлема, только если ожидаемо регулярное опустошение очереди, когда оказывается $f = r = 0$.

Избежать переполнения памяти при работе с очередью можно путём выделения фиксированного числа m, при котором упорядоченность элементов списка зацикливается, и считается, что $x[1]$ следует в очереди за $x[m]$. Тогда вставка элемента y в конец очереди реализуется присваиваниями:

если $r = m$, то $r := 1$, в противном случае $r := r + 1$; $\quad x[r] := y$;

если $f = m$, то $f := 1$, в противном случае $f := f + 1$; $\quad y := x[f]$.

Описанное зацикливание ограничивает длину очереди числом m.

Возможно расположение в общем сегменте памяти двух списков-очередей с переменной длиной (динамических), растущих навстречу другу из крайних ячеек (рис. 4). Это, разумеется, окажется эффективным, если имеются основания считать маловероятным быстрый одновременный рост обеих очередей.

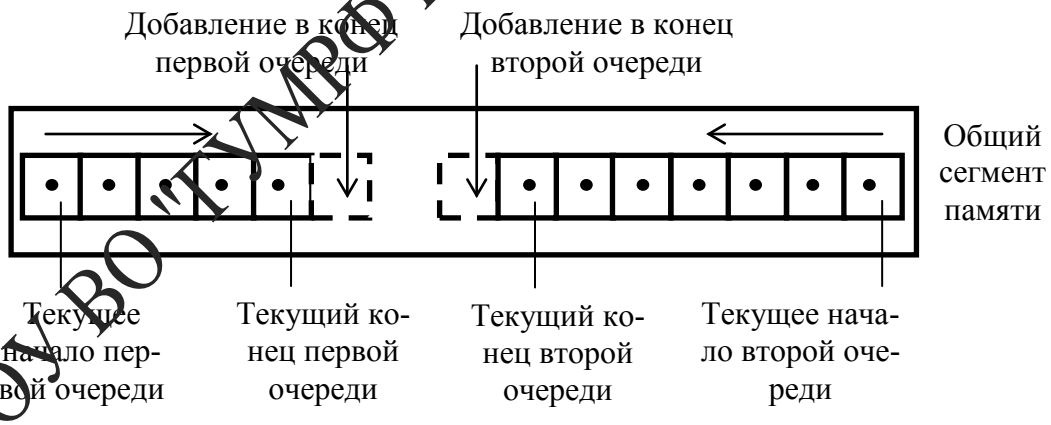


Рис. 4.

Расположение бóльшего числа списков в общем сегменте памяти предполагает отказ от фиксации адресов начальных элементов этих списков, что требует переписывания списков в новые участки при их сдвигах внутри общего сегмента. Здесь мы имеем дело с постоянно возникающей в программировании альтернативой <экономия памяти ↔ экономия времени>.

Цепное (связанное) распределение памяти при работе со списками. Последовательное распределение памяти для хранения списков требует выделения сегмента памяти, недоступного для других данных, даже если этот сегмент занят не полностью. Более гибким методом является *цепное* (иначе —*связанное*) распределение, при котором наряду с другими полями каждый элемент списка содержит дополнительное *адресное поле*, в котором хранится адрес следующего элемента, либо признак его отсутствия (рис. 5). При этом программа обслуживания списка должна содержать отдельную переменную a (*указатель списка*), значением которой является адрес начального элемента списка.

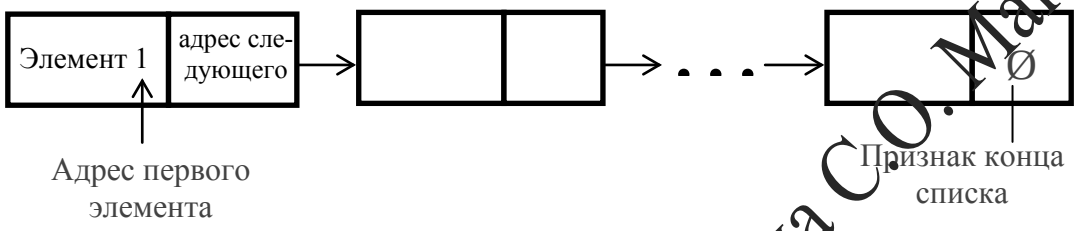


Рис. 5.

Сравнение последовательного и цепного типов распределения памяти. 1. Цепное распределение памяти для хранения списков позволяет избежать простаивания незанятой, «вакантной» части памяти, но требует дополнительных затрат памяти для обеспечения цепной адресации.

2. Удаление элемента $x[k]$ из цепного списка с произвольным доступом сводится лишь к изменению адресной части предыдущего элемента $x[k-1]$, которая теперь должна принять значение адресной части элемента, следуемого за удаляемым. В то же время при последовательном распределении удаление элемента предполагает сдвиг части списка на новые адреса.

3. Вставка нового элемента y между $x[k]$ и $x[k+1]$ в цепной список с произвольным доступом сводится к трём действиям:

1) адрес элемента y запоминается во вспомогательной переменной h;

2) в адресное поле элемента y заносится адрес элемента $x[k+1]$, до этого находившийся в адресном поле элемента $x[k]$;

3) в адресное поле элемента $x[k]$ записывается адрес расположения y, хранящийся в ячейке h.

NB: Здесь существенен порядок действий: при их перестановке адрес элемента $x[k+1]$ будет утрачен после его замены на адрес y. Вообще обмен значениями между переменными a и b должен проводиться с использованием вспомогательной переменной h: $h := a;\ a := b;\ b := h$.

4. Доступ к произвольному элементу списка быстрее реализуется при последовательном распределении памяти: доступ к элементу с номе-

ром k занимает фиксированное время, затрачиваемое на вычисление адреса $loc(x[k])$ по приведённой выше схеме (16). В то же время при цепном распределении нужно пройти по всем предшествующим элементам, чтобы, дойдя до $(k-1)$-го элемента, определить адрес k-го.

5. Объединение двух списков, как и разбиение списка на два, быстрее реализуются для цепных списков.

6. Схема цепной адресации при последовательном распределении памяти позволяет организовывать более сложные структуры, чем линейные списки. В частности, её можно применить для организации переменного количества списков переменной длины, а также для создания перекрёстных ссылок.

7. Время доступа к элементам цепного списка может резко возрастать при их попадании на разные сегменты внешнего устройства, предназначенного для хранения больших объёмов информации.

8. Цепное распределение памяти предполагает использование процедуры поиска свободной ячейки для размещения нового элемента. Для этого заводится список свободного пространства, реализуемый как стек с цепной адресацией и дисциплиной *lifo* («последний пришёл – первый ушёл»).

2.3. Циклические списки

Особой разновидностью представления в памяти цепного линейного списка является циклический список. Циклический список позволяет получить доступ к любому узлу списка, отправляясь от любого заданного элемента.

В циклически организованном цепном списке каждый элемент содержит в соответствующем поле адрес следующего элемента, причём не вырабатывается признак исчерпания списка. Такой список позволяет обслуживающей программе снова и снова просматривать его, двигаясь по кругу, пока в этом есть необходимость.

В отличие от случая незамкнутого линейного списка, элементы циклического списка являются равноправными и для выделения первого элемента необходимо иметь указатель a на начальный элемент (заголовок) списка. Однако во многих случаях нет необходимости выделять первый элемент и достаточно иметь указатель $a_{тек}$ на текущий элемент. Схема однонаправленного циклического списка приведена на рис. 6. В случае двунаправленного циклического списка два адресных поля элемента указывают на предыдущий и на последующий элементы списка. По такому списку можно передвигаться в любом направлении — как к началу, так и к концу (рис. 7).

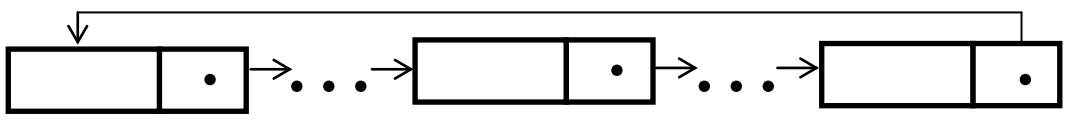


Рис. 6.

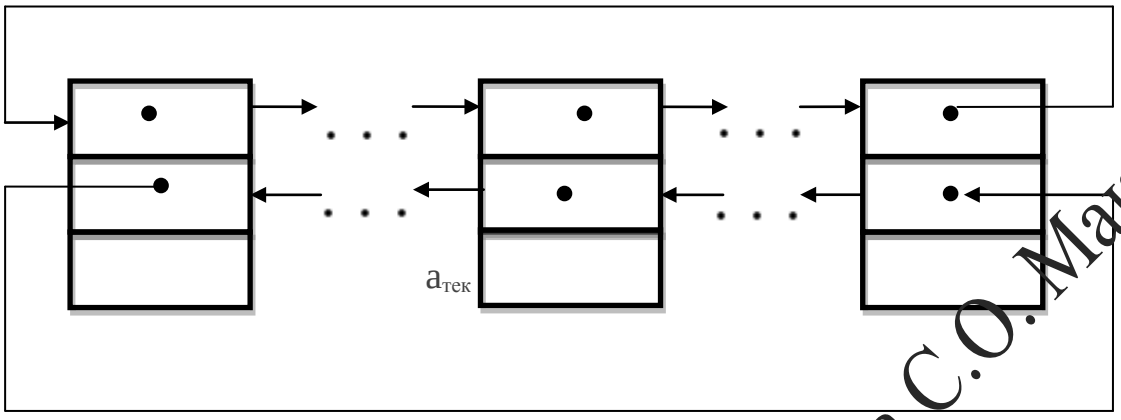


Рис. 7.

Возможность доступа ко всем элементам циклического списка с любой позиции, придаёт такому списку определённую симметрию. Обслуживающая программа может работать со всеми элементами списка одинаково.

При работе с циклическим списком следует применять специальные меры, предохраняющие от бесконечного цикла без возможности выхода из него (то есть от «зацикливания»). Возможным способом решения этой задачи является включение в список характерного элемента, который заведомо отличается от других и легко может быть идентифицирован. Например, подобный элемент может иметь значением пустое поле данных.

Другой возможностью является фиксация адреса элемента, с которого начата работа со списком. При этом должны быть предусмотрены действия на случай удаления этого элемента.

2.4. Массивы

2.4.1. Классификация и преобразование массивов

В языках программирования *массивом* называют совокупность значений переменной с индексами. Количество индексов, которыми снабжена переменная, называется *размерностью массива*. Элементы массива имеют однотипные значения и располагаются в памяти непосредственно друг за другом.

Одномерный массив $A_i\,(i=1,...,n)$ — набор n значений $\{A_1,...,A_n\}$.

Двумерный массив $A_{ij}\,(i=1,...,m;\ j=1,...,n)$ — набор $m\cdot n$ значений $\{A_{11},...,A_{1n},A_{21},...,A_{2n},...,A_{m1},...,A_{mn,}\}$. Двумерный массив представляет пря-

моугольную матрицу (A_{ij}) размера $m \times n$, состоящую из m строк и n столбцов. Элемент A_{ij} стоит на пересечении i-й строки и j-го столбца.

Трёхмерный массив $A_{ijk}\ (i = 1,...,m;\ j = 1,...,n;\ k = 1,...,l)$ — набор $m \cdot n \cdot l$ значений

$$\{A_{111},...,A_{11l},A_{121},...,A_{12l},...,A_{mn1},...,A_{mnl},\}.$$

Он имеет m слоёв, каждый из которых представляет прямоугольную матрицу размера $n \times l$. Массивы бóльших размерностей используются редко.

От цепного списка массив отличается *возможностью произвольного доступа к элементу с указанными значениями индексов*.

Данные организуются в массив, если они должны обрабатываться единообразно. Тогда обрабатывающий алгоритм делается циклическим, причём в роли параметра цикла (или параметров вложенных циклов) выступают индексы. Максимально допустимая размерность массива, типы и диапазоны значений индексов, ограничения на типы элементов определяются языком программирования и/или конкретным транслятором.

Статические массивы. Массив называется *статическим*, если его размерность и границы изменения индексов изначально фиксированы оператором объявления массива и в дальнейшем не меняются. Размер статического массива определяется на момент компиляции программы. Вся отведённая под его хранение память недоступна для размещения других объектов.

Элементы массива любой размерности считаются при их хранении линейно упорядоченными по адресам в соответствии с *лексикографическим* правилом:

- в одномерном массиве элемент A_i предшествует элементу A_j, если $i < j$;

- в двумерном массиве элемент A_{ij} предшествует элементу A_{st}, если $i < s$ либо если $i = s,\ j < t$;

- в трёхмерном массиве элемент A_{ijk} предшествует элементу A_{stp}, если $i < s$ либо если $i = s,\ j < t$, либо если $i = s,\ j = t,\ k < p$.

Динамические массивы. Массив называется *динамическим*, если границы изменения индексов (и, тем самым, число элементов) могут меняться во время исполнения программы. Такой массив обеспечивает бóльшую гибкость работы с данными, поскольку позволяет не отводить для его хранения сразу максимально возможный объём памяти, а регулировать её объём в соответствии с реально имеющимися на текущий момент данными. Размер динамического массива определяется во время исполнения программы и в дальнейшем может меняться.

Преобразование массивов. Двумерный или трёхмерный массив A может быть преобразован в одномерный массив B. Правило пересчёта индексов (откуда следует правило вычисления адресов):

- для двумерного массива размера $m \times n$:

$$A_{ij} \to B_q, \text{ где } q = (i-1)n + j; \tag{16}$$

- для трёхмерного массива размера $m \times n \times l$:

$$A_{ijk} \to B_q, \text{ где } q = (i-1)nl + (j-1)l + k. \tag{17}$$

Обратно, одномерный массив B может быть преобразован в двумерный или трёхмерный (возможно, не полностью заполненный).

Преобразование B в двумерный массив A размера $m \times n$: если при делении с остатком текущего индекса q массива B на границу n второго индекса массива A имеем $q = nd + r$, то

$$B_q \to A_{ij}, \text{ где } \begin{cases} i = d+1, j = r, & r = 0; \\ i = d+1, j = r, & r > 0. \end{cases} \tag{18}$$

Преобразование B в трёхмерный массив A размера $m \times n \times l$: если при делении с остатком имеем $q = (nl)d + r$, и $r = l\delta + \rho$ при $r > 0$, то

$$B_q \to A_{ijk}, \text{ где } \begin{cases} i = d, j = n, k = l, & r = 0; \\ i = d+1, j = \delta, k = l, & r = \rho > 0; \\ i = d+1, j = \delta + 1, k = \rho, & r = \rho > 0. \end{cases} \tag{19}$$

2.4.2. Адресация массивов

Адресация массива при последовательном распределении памяти. Когда массив хранится по последовательно предоставленным адресам, адресация его элементов задаётся формулами, аналогичными (16). Пусть начало массива задаётся *базовым адресом* l_0, причём отдельный элемент занимает объём памяти c адресуемых единиц. Тогда, в соответствии с (16) и (17):

- для элемента одномерного массива его адрес

$$loc(A_i) = l_0 + c(i-1); \tag{20}$$

- для элемента двумерного массива его адрес

$$loc(A_{ij}) = l_0 + c((i-1)m + (j-1)); \tag{21}$$

- для элемента трёхмерного массива его адрес

$$loc(A_{ijk}) = l_0 + c((i-1)mn + (j-1)n + (k-1)). \tag{22}$$

Таким образом, правило вычисления адреса элемента по заданному набору индексов обеспечивает одинаковое время доступа ко всем элементам массива.

Адресация массива при цепном распределении памяти. Цепное (цепное) распределение памяти позволяет представлять многомерные массивы в ситуациях, когда элементы одного списка могут включаться в ещё несколько списков.

В этом случае n-мерный массив реализуется двунаправленным цепным списком, в котором кроме поля для хранения значения элемента массива имеются $2n$ полей связи (по удвоенному числу индексов) с указате-

лями $u_1^{(l)}, u_1^{(r)}, .., u_n^{(l)}, u_n^{(r)}$. Указатели $u_h^{(l)}$ и $u_h^{(r)}$ ($h \in \{1,...,n\}$) содержат, соответственно, адрес предшествующего и последующего элементов в том — уже линейном — списке, который «вырезается» из общего многомерного массива, когда фиксируются значения всех индексов, кроме h-го.

Двунаправленное цепное представление массива целесообразно, когда каждый индекс соответствует какому-либо признаку. Например, поле для первого индекса связывает в общем списке людей с одинаковым цветом глаз, поле для второго индекса — людей одного пола, поле для третьего индекса — людей одного возраста и т. д. При этом поле для значения элемента может, например, выражать количество людей с соответствующим набором признаков.

2.4.3. Представление разреженных матриц

Прямоугольная матрица (A_{ij}) размера $m \times n$ называется разреженной, если большинство её элементов равно нулю. Последовательная адресация для хранения такой матрицы неэффективна, поскольку достаточно хранить только небольшое число ненулевых элементов. Разумеется, при этом усложняются алгоритмы работы с массивом.

Возможным способом хранения разреженной матрицы является организация трёх одномерных массивов одинаковой длины, равной числу ненулевых элементов, из которых первый содержит сами ненулевые значения, второй — значения их первых индексов, третий — значения их вторых индексов.

Другой возможностью является представление разреженной матрицы циклически связанными списками строк и столбцов способом, описанным в предыдущем пункте (см. также [7], с. 345 и далее). В элемент списка кроме поля *val* со значением A_{ij} включаются четыре (так как размерность массива равна 2) поля для указателей: поля *left* и *up* указывают адреса связи со следующими ненулевыми элементами при движении влево по строке или вниз по столбцу, а поля *row* и *col* задают индексы элемента, т.е. номера строки и столбца (рис. 8).

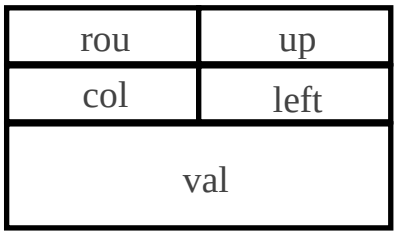


Рис. 8.

Кроме того, для каждой строки и каждого столбца задаются заголовки списков *brou*[i] и *bcol*[j]. Предполагается, что ссылка *left* в списке *brou*[i] служит адресом крайнего справа элемента i-й строки, а ссылка *up* в списке *bcol*[j] служит адресом последнего элемента j-го столбца.

Аналогичный приём можно применять, например, для представления симметричных ($A_{ij} = A_{ji}$) или антисимметричных ($A_{ij} = -A_{ji}$) матриц, то есть в ситуациях, когда значительная часть элементов массива не несёт дополнительной информации.

2.5. Деревья

2.5.1. Исходные понятия

Деревья образуют важный тип нелинейных структур данных в программировании. Древовидная структура задаёт для узлов этой структуры отношение ветвления, сходное со строением обычного дерева.

Следующее далее определение дерева формализует тип данных со связями/отношениями между узлами: один узел является исходным, из него «растут» ветки с другими узлами; если убрать ветки, идущие из корня, получатся несколько структур с меньшим количеством узлов, которые также следует считать деревьями (рис. 9, 10).

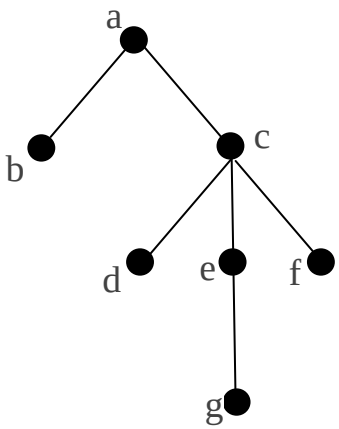


Рис. 9.

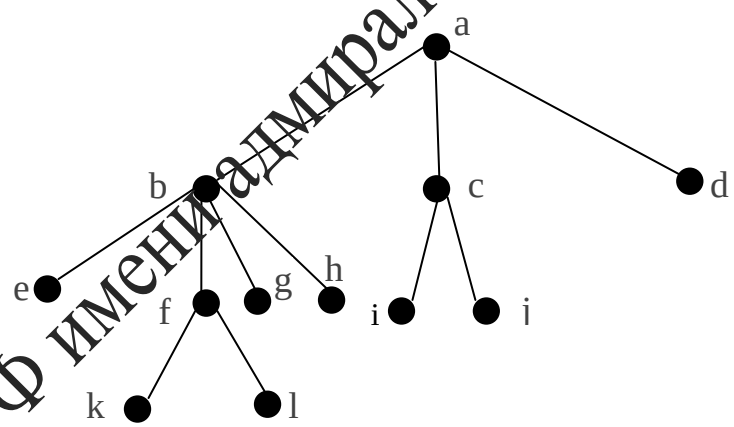


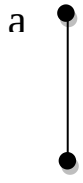
Рис. 10.

Рекурсивное определение: *Дерево* — это конечное множество T узлов, обладающее следующими свойствами:

1) выделен ровно один узел, называемый *корнем*;

2) остальные узлы распределены между $m \geq 0$ непустыми непересекающимися подмножествами $T_1,...,T_m$, каждое из которых в свою очередь является деревом; деревья $T_1,...,T_m$ называются *поддеревьями* данного корня.

Если дерево содержит всего один узел u, то u по определению является корнем. Далее, дерево с двумя узлами иметь только следующий вид (рис. 11):



b

Рис. 11.

Дерево с тремя узлами и корнем a может иметь одну из следующих структур (рис. 12):

1) $m = 1;\ T_1 = \{b, c\}$;

2) $m = 2$; тогда $T_1 = \{b\}, T_2 = \{c\}$, либо $T_1 = \{c\}, T_2 = \{b\}$.

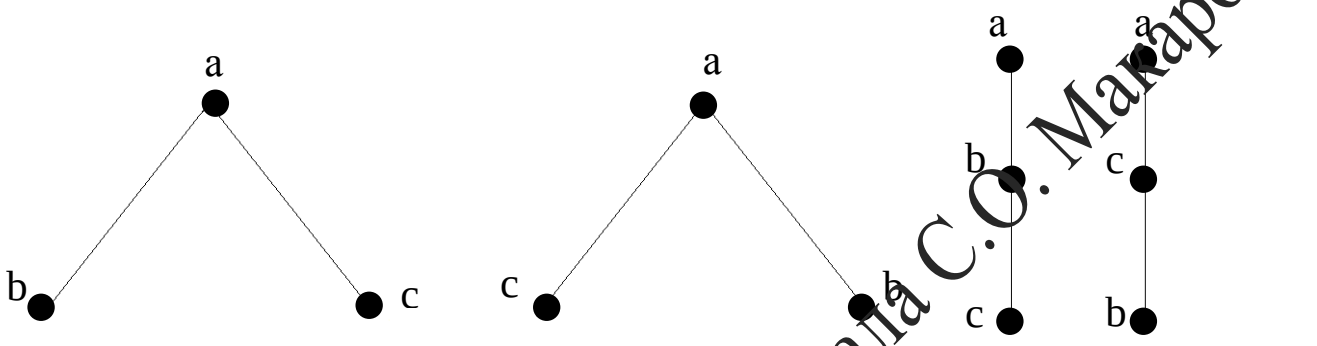


Рис. 12.

После того, как деревья с n узлами уже определены, на их основе определяются деревья с $n+1$ узлами.

Возможны эквивалентные приведённому, но уже нерекурсивные определения дерева, использующие понятия теории графов [2, 6, 10, 15]:

1. Дерево — это связный граф, не имеющий циклов.

2. Дерево — это граф, в котором любые две вершины соединены ровно одним простым путем.

3. Дерево — это связный граф, теряющий связность после удаления любого ребра.

Степень узла и уровень узла

С понятием дерева, как структуры данных, связаны следующие определения.

Каждый узел дерева либо является корнем некоторого поддерева, либо *концевым узлом* (*листом*). Неконцевой узел называется *узлом ветвления*. Количество поддеревьев узла называется его *степенью*.

Пример. На рис. 9 концевые узлы b, d, g, f имеют степень 0; узел e имеет степень 1; узел c имеет степень 3; корень a исходного дерева имеет степень 2.

Рекурсивно определяется *уровень узла по отношению к дереву* T, содержащему его: если узел является корнем, то его уровень равен нулю; уровень любого другого узла на единицу больше, чем уровень корня минимального поддерева содержащего этот узел.

Пример. Узел a на рис. 9 имеет уровень 0; узлы b и c имеют уровень 1; узлы d, e, f имеют уровень 2; узел g имеет уровень 3.

Степень узла характеризует структуру дерева ниже него, а уровень — структуру дерева выше него.

NB: Степень неконцевого узла на единицу меньше числа дуг, для которых этот узел является одним из концов (исключается «входящая» дуга, идущая от корня минимального поддерева, содержащего данный узел). *Уровень узла равен количеству рёбер, которые нужно пройти до него от корня.*

Как правило, для представления данных в виде деревьев существенен порядок перечисления поддеревьев, имеющих общий корень. Дерево с заданным порядком поддеревьев называется *упорядоченным*.

Можно доказать (индукцией по числу узлов дерева), что к *каждому узлу u дерева от корня u_0 ведёт единственный путь из узлов* $\{u_0, u_1, ..., u_{k-1}, u_k = u\}$, *где k — уровень узла u*.

Генеалогия узлов

При работе с деревьями принята «генеалогическая» терминология, согласно которой корень считается родителем корней его поддеревьев и предком других узлов этих поддеревьев
(соответственно, его потомков).

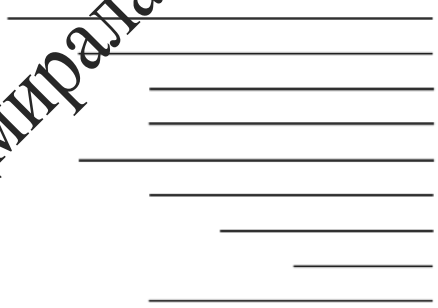


Рис. 13.

Отметим, что настоятельно рекомендуемое оформление компьютерных программ с отступами для вложенных фрагментов (рис. 13) является ещё одним способом оформления дерева.

2.5.2. Бинарные деревья

Бинарным деревом называется дерево, каждый узел которого имеет не более двух поддеревьев (равносильное условие: если из каждого узла — при общепринятой ориентации сверху от корня вниз к листьям — выходят не более двух дуг). Одно из этих поддеревьев считается левым, а другое правым (любое из них может отсутствовать).

Особая важность бинарных деревьев связана с тем, что данные со структурой дерева общего типа могут быть представлены в памяти компьютера в виде эквивалентного бинарного дерева.

Рекурсивное определение бинарного дерева: это пустое дерево, либо корень с двумя бинарными деревьями. Бинарное дерево может быть пустым

Частным случаем бинарного дерева является $(0-2)$-*дерево* (или *строго бинарное* дерево), каждый узел которого является либо корнем

точно двух поддеревьев, либо является листом (рис. 14). Количество узлов $(0-2)$-дерева нечётно.

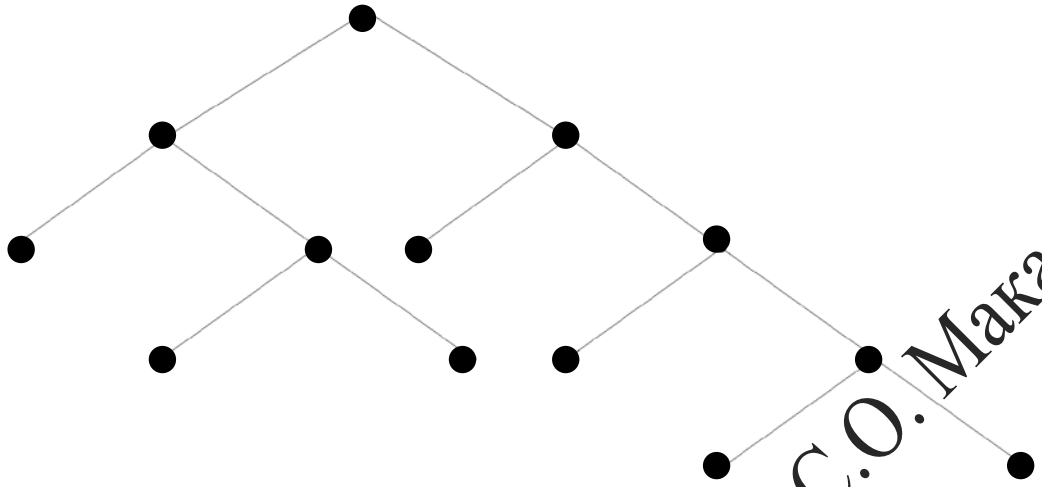


Рис. 14.

Между бинарными деревьями с n *узлами и* $(0-2)$*-деревьями с* $2n+1$ *узлами можно установить взаимно однозначное соответствие* f следующим образом. Ко всем концевым узлам бинарного дерева T присоединим $(0-2)$-дерево с тем же корнем и двумя помеченными ■ узлами; ко всем не концевым узлам с одним потомком присоединим ещё одного потомка в виде помеченного узла. В результате бинарное дерево перейдёт в $(0-2)$-дерево $T_{0-2} = f(T)$, все концевые узлы которого помечены. Обратно, удаляя все концевые узлы $(0-2)$-дерева T_{0-2}, получим бинарное дерево $T = f^{-1}(T_{0-2})$ (рис. 15).

Ещё один частный случай бинарного дерева — *полное бинарное дерево*. В таком дереве каждый узел, не являющийся листом, имеет степень 2 (имеет два поддерева) и все листья имеют один и тот же уровень n. Число n в этом случае называется уровнем полного бинарного дерева. В полном бинарном дереве количество узлов k равно сумме геометрической прогрессии:

$$k = 2^0 + 2^1 + 2^2 + ... + 2^n = 2^{n+1} - 1.$$

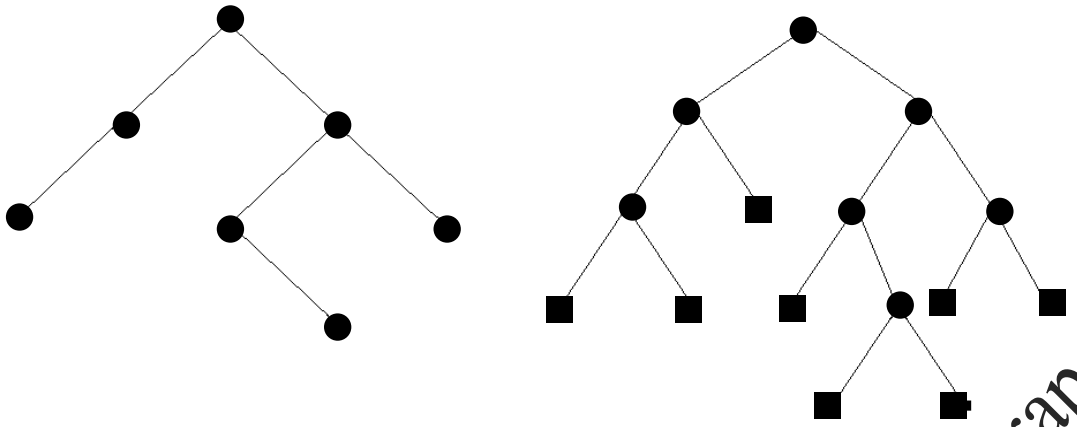


Рис. 15

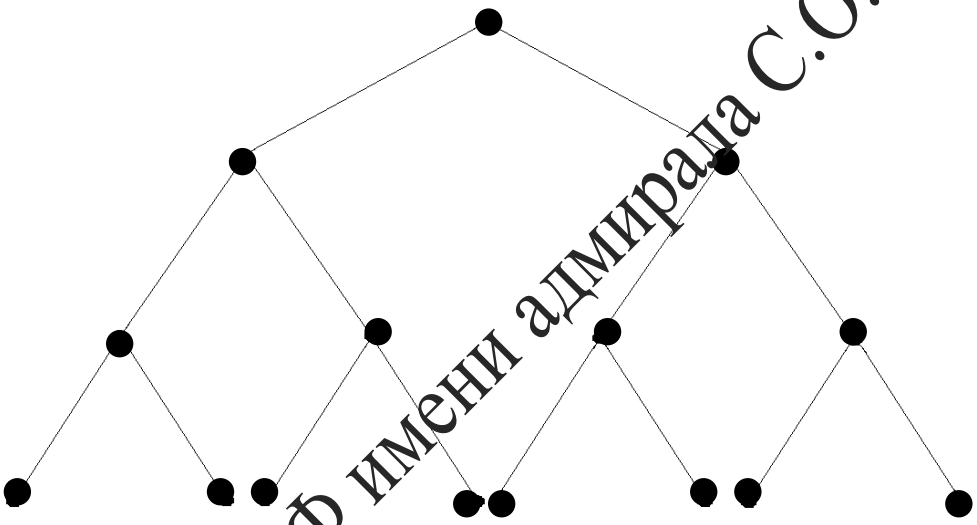


Рис. 16.

На (рис. 16) изображено полное бинарное дерево уровня 3.

С рекурсивным определением бинарных деревьев связан естественный способ их представления в памяти компьютера в виде цепных списков или в виде массивов.

Представление бинарных деревьев цепным списком. Списочное представление бинарных деревьев основано на элементах, соответствующих узлам дерева. Каждый элемент наряду с полем данных имеет два поля указателей. Один указатель используется для связывания элемента с левым потомком, а другой с правым. Листья имеют пустые указатели потомков. При таком способе представления обязательно следует сохранять указатель на узел, являющийся корнем всего дерева.

Итак, для представления бинарного дерева вводится указатель *t* на дерево (адрес корня), и в каждый узел помимо поля значения *val* включаются ещё два поля *llink* и *rlink* (рис. 17) для цепной адресации с двумя узлами следующего уровня — левым и правым, которые являются корня-

ми соответствующих поддеревьев; в случае отсутствия одного из них или обоих в эти поля помещается соответствующий признак $\varnothing$. Указатель пустого дерева $t = \varnothing$.

Рис. 17.

Пример. Дерево, изображённое на рис. 18, имеет в памяти представление, приведённое на рис. 19.

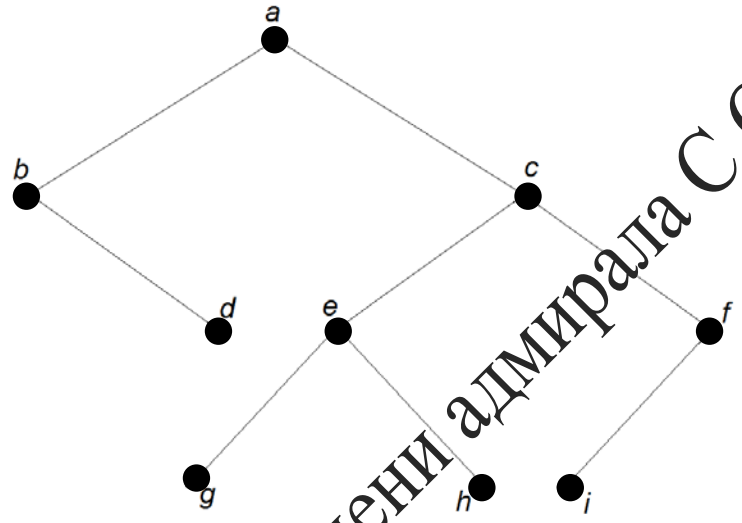


Рис. 18.

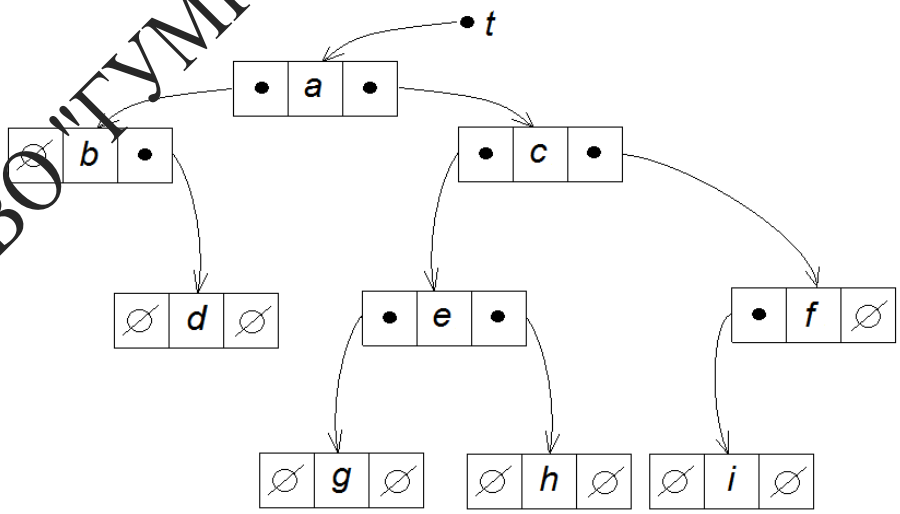


Рис. 19.

Представление бинарных деревьев массивом. В виде одномерного массива проще всего представляется полное бинарное дерево, так как оно всегда имеет строго определенное число узлов на каждом уровне. Узлы можно естественным образом упорядочить, нумеруя слева направо последовательно по уровням и использовать эти номера в качестве индексов в одномерном массиве (рис. 20).

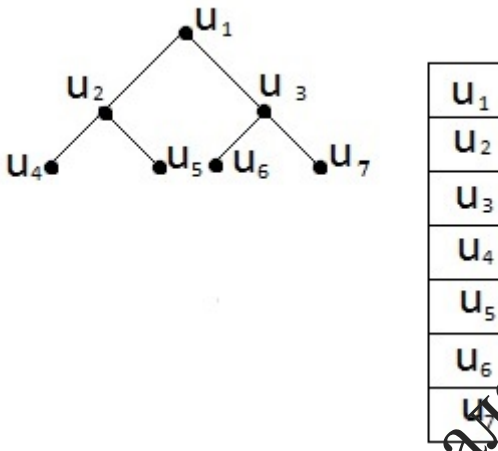


Рис. 20.

Если есть основания предполагать, что число уровней дерева в процессе работы с ним не будет значительно меняться, то данный способ представления полного бинарного дерева значительно более экономичен, чем представление списковой структурой.

Для представления неполных бинарных деревьев можно применить следующий способ:

1. Бинарное дерево дополняется до полного дерева, вершины последовательно нумеруются.

2. В массив заносятся поля данных только тех узлов, которые были в исходном неполном дереве, в соответствии с индексами этих узлов в полном дереве.

3. В незанятые элементы массива заносится характерное значение в качестве признака пропуска элемента.

Если в полном бинарном дереве узел имеет уровень k и занимает в этом уровне позицию с номером i, то его индекс в представляющем массиве $j = 2^k - 1 + i$. Индекс корня единице. Для узла с координатами в дереве $(k;i)$ и, соответственно, индексом $j = 2^k - 1 + i$ в представляющем массиве индексы j' и j'' его левого и правого непосредственных потомков вычисляются по формулам: $j' = 2j$; $j'' = 2j + 1$.

Главным недостатком рассмотренного способа представления бинарного дерева является то, что структура данных является статической. Размер массива выбирается исходя из максимально возможного количе-

ства уровней бинарного дерева. При этом, чем менее полным является дерево, тем менее рационально используется память.

2.5.3. Обход бинарных деревьев

Обходом дерева называется алгоритм перебора его узлов (с последующей обработкой — просмотром, изменением и т. п.), при котором каждый узел посещается ровно один раз, то есть перебор без пропусков и повторений. Каждый такой алгоритм приводит к линейному упорядочению узлов дерева в порядке обхода, позволяя говорить о предыдущем узле и о следующем узле.

Введём следующие обозначения: t — указатель на дерево (то есть на корень уровня 0); s — указатель на вершину стека; l — переменная связи; $\varnothing$ — признак отсутствия связи (в случае листа, то есть концевого узла).

Различают следующие основные способы обхода бинарного дерева:

Прямой (префиксный) порядок обхода. Его рекурсивное определение: (а) если бинарное дерево пусто, то обход завершён (ничего делать не требуется); (б) если дерево не пусто, то выполняются три действия — сначала переход в корень; затем обход левого поддерева; в заключение обход правого поддерева.

Содержательно это означает, что каждый узел посещается до того, как посещены его потомки. Из каждого узла, рассматриваемого как корень поддерева, производится переход в левый узел следующего уровня до тех пор, пока такой уровень находится; когда левого нижнего узла больше нет, возврат в предыдущий корень и переход в правый узел следующего за ним уровня, из которого снова в левый нижний и т. д.

Заметим, что правило поиска выхода из лабиринта соответствует описанному алгоритму, применённому для нахождения узла с заданным значением (признаком, что узел является выходом).

Пример. Для дерева, представленного на рис. 18, этот обход приводит к последовательности узлов

$$a \to b \to d \to c \to e \to g \to h \to f \to i.$$

Алгоритм прямого обхода. Заводится стек, в который (всякий раз сверху) помещается ссылка на корень очередного поддерева.

╔

Ш1. Инициализация: стек опустошается: $s := \varnothing$, и $l := t$.

Ш2. Проверка пустоты связи: $l = \varnothing$?

Если «да», то

переход к **Ш4.**

если «нет», то

обработка узла *node*(l) с адресом l.

(Под обработкой узла подразумеваются действия с полем данных узла, ради которых устраивается обход дерева).

Ш3. В стек кладётся сверху значение указателя: $s := l$. Теперь l указывает на непустое дерево, подлежащее обходу в текущий момент. Указателю l присваивается значение ссылки на левое поддерево: $l := llink(l)$. Переход к **Ш2**.

Ш4. Если стек пуст, обход всего дерева закончен. В противном случае указателю l присваивается значение верхней ссылки стека: $l := s$ и она удаляется из стека. Переход к **Ш5**.

Ш5. Указателю l присваивается значение ссылки на правое поддерево: $l := rlink(l)$. Переход к **Ш2**.

╚

***NB*:** Этот алгоритм (как и описанные ниже) предполагает, что в обработку узла не входит его удаление из дерева.

Симметричный (центрированный, инфиксный, лексикографический) порядок обхода. Его рекурсивное определение: (а) если бинарное дерево пусто, обход завершён (ничего делать не требуется); (б) если дерево не пусто, то выполняются три действия — обход левого дерева, посещение корня, обход правого дерева.

Содержательно это означает, что сначала обходится левое дерево, затем посещается его корень, после чего обходится правое дерево.

Пример. Для дерева, представленного на рис. 18, этот обход приводит к последовательности узлов

$$b \to d \to a \to g \to e \to h \to c \to i \to f .$$

Алгоритм симметричного обхода. Заводится стек для ссылки на корень очередного поддерева.

╔

Ш1. Инициализация: стек опустошается, и $l := t$.

Ш2. Проверка пустоты связи: $l = \varnothing$?

Если «да», то

переход к **Ш4**.

Ш3. В стек кладётся сверху значение указателя: $s := l$. Теперь l указывает на непустое дерево, подлежащее обходу в текущий момент. Указателю l присваивается значение ссылки на левое поддерево: $l := llink(l)$. Переход к **Ш2**.

Ш4. Если стек пуст, то

обход всего дерева закончен,

в противном случае:

указателю l присваивается значение верхней ссылки стека: $l := s$;

производится переход к следующему шагу **Ш5**.

Ш5. Обработка узла $node(l)$, на который указывает текущее значение l (действия с полем данных узла, ради которых устраивается обход дерева).

Указателю l присваивается значение ссылки на правое поддерево: $l := rlink(l)$.

Переход к **Ш2**.

⊥

Обратный (постфиксный, концевой) порядок обхода. Его рекурсивное определение: (а) если бинарное дерево пусто, то обход завершён (ничего делать не требуется); (б) если дерево не пусто, то выполняются три действия — обход левого дерева, обход правого дерева, переход в корень.

Содержательно это означает, что узлы посещаются снизу вверх слева направо. В момент посещения узла все его потомки уже пройдены.

Пример. Для дерева, представленного на рис. 18, этот обход приводит к последовательности узлов

$$d \to b \to g \to h \to e \to i \to f \to c \to a$$

Обход в ширину (поуровневый). Сначала посещается корень дерева, затем слева направо узлы первого уровня, затем узлы второго уровня и т. д.

Пример. Для дерева, представленного на рис. 18, этот обход приводит к последовательности узлов

$$a \to b \to c \to d \to e \to f \to g \to h \to i.$$

2.6. Динамическое распределение памяти

Представление динамических массивов — частный случай динамического распределения памяти, когда несколько объектов (списков, массивов, файлов) могут независимо расти или уменьшаться в области общего пула памяти. Обычно элементы (узлы) этих объектов имеют разный размер, и желательны технологии для резервирования и освобождения блоков переменного размера в большой области памяти из последовательных ячеек.

Технология резервирования памяти. После нескольких актов занятия/освобождения ячеек в общем пуле памяти, последний, будучи изначально сплошным, приобретает «пунктирный» вид, когда по мере возрастания адресов свободные сплошные — не закрашенные — участки чередуются с занятыми, закрашенными (рис. 21). Технология резервирования в таком фрагментированном пуле должна решать две задачи:

(а) фиксация фрагментации пула в памяти компьютера;

(б) построение подходящего алгоритма поиска свободного блока из n последовательных ячеек при заданном n.

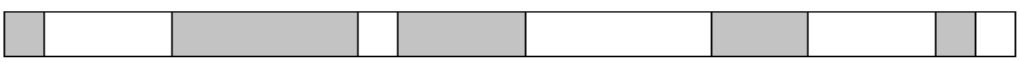


Рис. 21.

Задача (а) решается путём поддержания *списка свободной памяти* в определённом месте; чаще всего такой список содержится в том же общем пуле, подлежащем динамическому распределению. Первое слово каждой свободной области памяти содержит размер области и адрес первого слова следующей свободной области. Свободные области при адресации могут быть связаны в порядке возрастания или убывания размера, либо в порядке возрастания адресов, либо в произвольном порядке.

Алгоритм решения задачи (б): если необходима свободная область размера n, то находится область, у которой размер $m \geq n$, её размер уменьшается до $m-n$, и соответствующим образом изменяется адрес новой, уменьшенной свободной области в согласии с принятым способом упорядочивания. Выбор свободной области размера $m \geq n$ может осуществляться двумя наиболее распространёнными способами:

1) *Метод наилучшего подходящего блока* — выбирается блок с наименьшим допустимым размером m. Это предполагает просмотр всего списка свободной памяти и соответствующих затрат машинного времени. Идеология метода — сохранение бóльших свободных областей «на потом», с целью уменьшения фрагментации общего пула памяти. Недостатком метода является тенденция к увеличению количества свободных блоков малого размера.

2) *Метод первого подходящего блока* — выбирается первый встретившийся в списке блок с допустимым размером m.

Технология высвобождения памяти. Когда необходимость в выделенном блоке отпадает, он должен быть возвращён в список свободного пространства. Здесь обычно используются два подхода:

1) *Сборка мусора* — список свободного пространства не изменяется вплоть до тех пор, пока пул не окажется заполненным; после этого строится новый список.

Недостатком метода является тенденция к замедлению работы при почти заполненном пуле совместно используемой памяти. Кроме того, фактически освободившаяся, но неучтенная как свободная, область памяти, вынуждает алгоритм резервирования занимать часть свободной памяти в другом месте, увеличивая фрагментацию пула.

2) *Уплотнение* — немедленный возврат и слияние освобождённых областей. Все занятые блоки перемещаются в соседние позиции, после чего вся свободная область оказывается одним сплошным блоком.

Недостатком этой технологии является увеличение затрат времени на перемещение задействованных блоков и изменение указателей связи.

Глава 3. МАШИННАЯ АРИФМЕТИКА

3.1. Машинное представление целых чисел

Под формой машинного представления чисел понимается совокупность правил, устанавливающих соответствие между записью числа и его количественным значением.

Представление целых чисел в прямом коде. Натуральные числа n, записанные в двоичной системе счисления, являются наборами двоичных разрядов из нулей и единиц (битов):

$$n \leftrightarrow a_{k-1}a_{k-2}...a_1a_0,$$

что соответствует равенству

$$n = \sum_{i=0}^{k-1} a_i 2^i,\ a_i \in \{0,1\}.$$

Для единообразного представления как положительных, так и отрицательных целых чисел a отводится специальный *знаковый разряд* a_*, обособленный от остальных разрядов. Принято соглашение: нуль в знаковом разряде представляет «+», а единица «−». Знаковый разряд помещается либо перед остальными разрядами, либо после них. Мы будем придерживаться первого варианта.

Тогда *представление целого числа в прямом коде*, занимающее $k+1$ двоичных разрядов (k цифровых и один знаковый), имеет вид:

$$[a]_{\text{пр}} = a_*a_{k-1}a_{k-2}...a_1a_0, \qquad (23)$$

что соответствует равенству

$$a = \pm\sum_{i=0}^{k-1} a_i 2^i.$$

Цифровой разряд a_i имеет вес 2^i; в частности, старший цифровой разряд имеет вес 2^{k-1}.

Подстроку $a_{k-1}a_{k-2}...a_1a_0$ прямого кода будем называть его *модульной* (или *цифровой*) *частью*.

При фиксированном числе k цифровых двоичных разрядов в прямом коде, в соответствии с комбинаторным принципом умножения, может быть представлено (возможно, с нулевыми первыми цифрами) $2\cdot 2^k - 1 = 2^{k+1} - 1$ целых чисел (единица вычитается, поскольку две строки с нулевыми цифровыми разрядами представляют одно и то же число: $+0 = -0$).

Наименьшее представляемое число равно $-\sum_{i=0}^{k-1} 2^i = -(2^k - 1)$; наибольшее равно $\sum_{i=0}^{k-1} 2^i = 2^k - 1$ (последние равенства вытекают из формулы суммирования k членов геометрической прогрессии с начальным чле-

ном 1 и знаменателем 2). Неоднозначность представления нуля является недостатком прямого кода.

Инвертирование. Рассмотрим сначала пример. Натуральное число 6 имеет трёхразрядную двоичную запись (без знакового разряда): $[6]_{\text{пр}} = 110$. Если в этой записи нули поменять на единицы, а единицы на нули, получится строка 001, которая представляет в прямом коде число $1 = 2^3 - 6 - 1$. При этом

$$6 + 1 = 7 = 2^3 - 1; \quad [2^3 - 1]_{\text{пр}} = 111.$$

Таким же образом $[10]_{\text{пр}} = 1010$; если поменять в этой записи двоичные цифры по схеме $0 \mapsto 1,\ 1 \mapsto 0$, получим строку 0101, которая представляет в прямом коде число $5 = 2^4 - 10 - 1$; при этом

$$10 + 5 = 15 = 2^4 - 1; \quad [2^4 - 1]_{\text{пр}} = 1111.$$

Инвертированием (или *инверсией*) k-разрядной битовой строки $a = (a_{k-1}, a_{k-2}, ..., a_1, a_0)$ называется строка a', полученная преобразованием по правилу $0 \mapsto 1,\ 1 \mapsto 0$. Операцию инвертирования называют также *дополнением до единицы*.

Например, если $a = (10011)$, то $a' = (01100)$.

Если рассматривать строки a и a' как двоичные записи натуральных чисел, то их сумма по правилу двоичного сложения даёт строку из единиц, то есть двоичную запись числа $2^k - 1$:

$$a + a' = 2^k - 1. \tag{24}$$

Представление целых чисел в обратном коде.

Поскольку вычисление разности двух чисел традиционным способом «в столбик» производится существенно дольше, чем сложение, и связано с «заниманием» единицы из старшего разряда, целесообразно заменить его более быстрыми операциями инвертирования и сложения. Этой цели служит представление отрицательных чисел в виде, отличном от прямого кода.

Положительные целые числа имеют обратный код (англ. *inverted code* или *complement on one* — «дополнение до единицы») $[a]_{обр}$, *совпадающий с прямым кодом*.

Обратный код представления отрицательного числа a получается инвертированием модульной части прямого кода.

Максимально возможное положительное число в $(k+1)$-разрядной сетке (т.е. с k цифровыми разрядами), представимое в обратном коде, по-прежнему равно $2^k - 1$. Наименьшее представимое отрицательное число равно $-(2^k - 1)$.

Примеры. 1. Пусть для представления целого числа отведён байт (8 битов). Обратный код для $a=-89$ получается следующим образом:

$$-89 \underset{\text{прямой код}}{\rightarrow} 1_*1011001 \underset{\text{инверти-рование модуля}}{\rightarrow} 1_*0100110;$$

$$[-89]_{\text{обр}} = 1_*0100110.$$

Если интерпретировать полученную битовую строку как прямой код отрицательного числа с семью цифровыми разрядами, то это код числа $a' = -38 = -(2^7 - 1 - |-89|)$:

$$[-38]_{\text{пр}} = 1_*0100110 = [-89]_{\text{обр}}.$$

2. Содержимое байта $1_*0100110$, интерпретируемое как обратный код, представляет отрицательное (поскольку в старшем разряде единица) число $a=-89$, модуль которого получается инвертированием всех разрядов кроме старшего (знакового):

$$1_*0100110 \underset{\text{инверти-рование модуля}}{\rightarrow} 1_*1011001 \underset{\text{прямой код}}{\rightarrow} -89$$

Эта же битовая строка, интерпретируемая как прямой код, представляет отрицательное число $a'=-38$. При этом

$$a + a' = -89 + (-38) = -127 = -(2^7 - 1).$$

3. Обратный 8-битовый код $1_*1111110$ представляет число $a=-1$:

$$1_*1111110 \underset{\text{инверти-рование модуля}}{\rightarrow} 1_*0000001 \underset{\text{прямой код}}{\rightarrow} -1.$$

Эта же битовая строка, интерпретируемая как прямой код, представляет отрицательное число $a'=-126$. При этом $a + a' = -1 + (-126) = -127 = -(2^7 - 1)$.

Пусть a' – отрицательное число, прямой код которого совпадает с обратным кодом отрицательного числа a. В силу (24), $|a| + |a'| = 2^k - 1$. Так как $|a| = -a$, $|a'| = -a'$, то $-a - a' = 2^k - 1$, и $a' = -2^k + 1 - a = -(2^k - 1 + a)$. Таким образом, получаем утверждение:

Обратный код отрицательного числа a с одним знаковым и k цифровыми разрядами совпадает с прямым кодом отрицательного числа $a' = -(2^k - 1 + a)$:

$$[a]_{\text{обр}} = [-(2^k - 1 + a)]_{\text{пр}}.$$

Пример. $-106 \underset{\text{прямой код}}{\rightarrow} 1_*1101010 \underset{\text{инвертирование}}{\rightarrow} 1_*0010101$. Полученная битовая строка, рассматриваемая как прямой код, представляет число $-21 = -(2^7 - 1 + (-106))$.

К обратному коду можно придти также следующим образом. Изменим интерпретацию знакового разряда a_* в (23), приписывая ему по умолчанию отрицательный вес $-(2^k - 1)$. Это означает понимание записи (23) как равенства

$$a' = -a_*(2^k - 1) + \sum_{i=0}^{k-1} a_i 2^i .$$

У положительного числа знаковый разряд $a_* = 0$, так что для него новая интерпретация знакового разряда ничего не меняет. У отрицательного же числа $a_* = 1$, и запись (23) означает в новой интерпретации, что $a' = -(2^k - 1) + \sum_{i=0}^{k-1} a_i 2^i$. Поскольку при этом $2^k - 1 = \sum_{i=0}^{k-1} 2^i$, то

$$a' = -\sum_{i=0}^{k-1} 2^i + \sum_{i=0}^{k-1} a_i 2^i = -\sum_{i=0}^{k-1} (2^i - a_i 2^i) = -\sum_{i=0}^{k-1} (1 - a_i) 2^i .$$

Коэффициенты $1 - a_i$ получаются из цифр a_i прямого кода числа $|a| = \sum_{i=0}^{k-1} a_i 2^i$ поразрядным инвертированием $0 \mapsto 1,\ 1 \mapsto 0$, как это имеет место и при получении обратного кода.

Итак, *получение обратного кода отрицательного числа a с k цифровыми разрядами равносильно приписыванию знаковому разряду a_* отрицательного веса $-(2^k - 1)$ и получению прямого кода получающегося числа*.

Для изменения знака отрицательного числа a, записанного в обратном коде, следует инвертировать все его разряды, включая знаковый; в результате получится прямой код положительного числа $-a$.

Пример. $[-5]_{\text{обр}} = 1_*010$; инверсия всех разрядов, включая знаковый, даёт 0_*101 — прямой код числа 5.

Обратный код имеет тот же недостаток, что и прямой — двоякое представление нуля: отрицательный нуль $1_*11...1$ как обратный код для $-0 = 1_*00...0$, и положительный нуль $0_*00...0$.

Приведём без доказательства следующее утверждение:

Вычисление обратного кода разности чисел $a - b$ можно осуществить в виде $a - b = a + (-b)$ по схеме:

1) *представить целые числа a и $-b$ в обратном коде*;

2) *сложить битовые строки по правилам двоичной арифметики*;

3) *прибавить, к младшему разряду суммы перенос* λ, *возникающий после сложения знаковых разрядов; последний равен нулю или единице*:

$$[a-b]_{\text{обр}} = [a]_{\text{обр}} + [-b]_{\text{обр}} + \quad . \qquad (25)$$

Пример. Рассмотрим вычисление разности $12-5$. Имеем:
$[12]_{\text{обр}} = 0{*}1100$ (для положительных чисел обратный код совпадает с прямым); далее, $[-5]_{\text{обр}} = 1{*}1010$;

$$0{*}1100 + 1{*}1010 = \underset{\lambda}{1}0{*}0110;\ 0{*}0110 + \underset{\lambda}{1} = 0{*}0111 = [7] \quad .$$

Разность отрицательного числа и положительного целесообразно вычислять в машинной арифметике, приписывая знак минус сумме их модулей: $-12-5 = -(12+5)$. Такой же результат даёт и вычисление по схеме (25). Так, при вычислении разности $-12-5$ с использованием обратных кодов при пяти цифровых разрядах имеем:
$[-12]_{\text{обр}} = 1{*}10011$;
$[-5]_{\text{обр}} = 1{*}11010$;
$1{*}10011 + 1{*}11010 = \underset{\lambda}{1}1{*}01101;\ 1{*}01101 + \underset{\lambda}{1} = 1{*}01110$ — обратный код для числа -17.

Представление целых чисел в дополнительном коде.

Дополнительный код (англ. *two's complement code* или *twos-complement code* — «дополнение до двух») *представления положительного числа совпадает с его прямым кодом*.

Максимальное представимое число в $(k+1)$-разрядной сетке (k цифровых разрядов и один знаковый) по-прежнему равно $2^k - 1$.

Дополнительный код отрицательного числа получается инвертированием модульной части его прямого кода, прибавлением единицы к полученной битовой строке по правилам двоичной арифметики и сохранением единицы в качестве знакового разряда.

Примеры. 1. Пусть для представления целого числа отведён байт (8 битов). Дополнительный код для $a = -89$ получается следующим образом:

$$-89 \xrightarrow[\text{код}]{\text{прямой}} 1{*}1011001 \xrightarrow[\text{рование модуля}]{\text{инверти-}} 1{*}0100110 \xrightarrow{+1} 1{*}0100111;$$

$$[-89]_{\text{доп}} = 1{*}0100111.$$

Если интерпретировать полученную битовую строку как прямой код, то это код числа $a' = -39 = -(2^7 - |-89|)$.

2. Дополнительный код для числа $a = -90$:

$$-90 \xrightarrow[\text{код}]{\text{прямой}} 1{*}1011010 \xrightarrow[\text{рование модуля}]{\text{инверти-}} 1{*}0100101 \xrightarrow{+1} 1{*}0100110;$$

$$[-90]_{\text{доп}} = 1{*}0100110.$$

3. Содержимое байта $1_*0100110$, интерпретируемое как дополнительный код, представляет отрицательное (поскольку в знаковом разряде единица) число a, модуль которого получается вычитанием единицы и последующим инвертированием всех разрядов, кроме старшего (знакового):

$$1_*0100110 \underset{-1}{\rightarrow} 1_*0100101 \underset{\substack{\text{инверти-}\\\text{рование}\\\text{модуля}}}{\rightarrow} 1_*1011010 \underset{\substack{\text{прямой}\\\text{код}}}{\rightarrow} -90.$$

4. Дополнительный 8-битовый код $1_*1111111$ представляет число -1:

$$1_*1111111 \underset{-1}{\rightarrow} 1_*1111110 \underset{\substack{\text{инверти-}\\\text{рование}\\\text{модуля}}}{\rightarrow} 1_*0000001 \underset{\substack{\text{прямой}\\\text{код}}}{\rightarrow} -1.$$

Минимальное отрицательное число, представимое в дополнительном коде, занимающем k основных (цифровых) разрядов и один знаковый, есть -2^k. В частности, байт (восемь разрядов), интерпретируемый как дополнительный код, представляет числа от $-128 = -2^7$ до $127 = 2^7 - 1$.

Недостатком дополнительного кода является необходимость выполнения при его получении арифметических действий, при которых единица переноса может проходить по всем разрядам. Это существенно увеличивает время, необходимое для взаимных преобразований между прямым и дополнительным кодом.

Поскольку инвертирование модуля отрицательного числа a означает, в силу (24), переход к числу $2^k - 1 - |a| = 2^k - 1 + a$, то после прибавления к его прямому коду единицы получается $2^k + a$. Итоговое дописывание знаковой единицы означает получение прямого кода числа $a' = -2^k - a$. Таким образом,

$$[a]_{\text{доп}} = [a']_{\text{пр}} = [-2^k - a]_{\text{пр}} = [-2^k + |a|]_{\text{пр}}. \qquad (26)$$

Итак, *переход к дополнительному коду можно рассматривать как приписывание знаковому разряду a_* отрицательного веса -2^k.*

Поскольку $[a]_{\text{доп}} = [a']_{\text{пр}}$, то, беря дополнительный код уже для a', получим в силу (26):

$$[a']_{\text{доп}} = [-2^k - (-2^k - a)]_{\text{пр}} = [a]_{\text{пр}}.$$

Итак, *для того чтобы вернуться от дополнительного кода к прямому коду, достаточно повторно применить операцию получения дополнительного кода*.

Приведём без доказательства следующее утверждение:

Дополнительный код разности натуральных чисел $a - b = a + (-b)$, можно получить сложением дополнительных кодов чисел a и $-b$ (напомним, что для положительного a дополнительный код совпадает с прямым):

$$[a - b]_{\text{доп}} = [a]_{\text{пр}} + [-b]_{\text{доп}}. \qquad (27)$$

Примеры.

Пусть $a = 14,\ b = 5$.

$[14]_{\text{пр}} = 0{*}1110$;

$[-5]_{\text{пр}} = 1{*}0101$; $[-5]_{\text{обр}} = 1{*}1010$; $[-5]_{\text{доп}} = 1{*}1011$;

$[14]_{\text{пр}} + [-5]_{\text{доп}} = 0{*}1110 + 1{*}1011 = 0{*}01001$ — дополнительный (он же и прямой) код разности $14 - 5 = 9$.

Ещё один пример — когда разность оказывается отрицательной:

$[11]_{\text{пр}} = 0{*}1011$; $[-13]_{\text{пр}} = 1{*}1101$; $[-13]_{\text{обр}} = 1{*}0010$; $[-13]_{\text{доп}} = 1{*}0011$;

$[11]_{\text{пр}} + [-13]_{\text{доп}} = 0{*}1011 + 1{*}0011 = 1{*}1110$ — дополнительный код числа $-2 = 11 - 13$.

Представление дробных чисел в форме с естественной точкой. Естественная форма представления числа в ЭВМ — следствие повседневной практики записи чисел в виде целой и дробной части, разделённых десятичной точкой (или запятой). У отрицательных чисел перед старшей цифрой помещается знак «–». Представление чисел с естественной точкой в виде последовательности цифр предполагает дополнительное указание на положение точки.

Представление дробных чисел в форме с фиксированной точкой. Необходимость в указании положения десятичной точки отпадает, если её место в разрядной сетке по умолчанию фиксировано.

Пусть представление осуществляется в позиционной системе счисления с основанием q, и на представление отводится N цифровых разрядов: l разрядов на дробную часть и $N - l$ разрядов на целую. Тогда диапазон для модуля ненулевых представляемых чисел — от q^{-l} до $q^{-l}(q^N - 1)$. Например, при $q = 10,\ N = 6,\ l = 2$ наибольшее число (все цифры в записи — девятки) есть $9999.99 = 10^{-2}(10^6 - 1) = 10^4 - 0.01$.

3.2. Арифметика чисел с фиксированной точкой

Выполнение арифметических действий с числами в форме с фиксированной точкой осуществляется быстрее и проще, чем в случае представления с естественной точкой. По сути, алгоритмы этих операций такие же, как при операциях с целыми числами (без дробной части).

При выполнении арифметических действий возможно переполнение отведённой разрядной сетки, рассматриваемое как аварийная ситуация.

Машинное умножение целых чисел. В q-ичной системе счисления умножение целого числа $a = \sum_{i=0}^{k} a_i q^i$ на целое число $b = \sum_{j=0}^{k} b_j q^j$ реализуется согласно равенству

$$ab = \sum_{i=0}^{k} (a_i b) q^i .$$

Машинное умножение целых чисел основано на этом равенстве: частичное произведение $a_i b$ реализуется последовательностью операций сложения:

$$a_i b = \underbrace{b + b + ... + b}_{\substack{a_i \text{ операций} \\ \text{сложения}}};$$

после этого умножение на степень основания системы счисления q^i осуществляется как сдвиг на i разрядов влево по разрядной сетке с дописыванием i нулей справа; затем складываются полученные числа. Итак, *машинное умножение целых чисел осуществляется с помощью аппаратно реализованных операций сложения и сдвига*.

Машинное деление целых чисел. Алгоритм деления существенно зависит не только от реализованных в компьютере аппаратных средств, но и от кода представления делимого и делителя — прямого, дополнительного или обратного.

Возможный алгоритм деления для чисел, представленных в прямом коде, основан на алгоритме деления с остатком и является формальной алгоритмизацией привычного деления «в столбик», поизводимого «на бумаге» (см. [16], с. 83 и далее).

Важным промежуточным этапом на каждой итерации деления «в столбик» является отыскание неполного частного при делении $(n+1)$-разрядного числа $(u_n u_{n-1} ... u_0)$ на n-разрядное число $(v_{n-1} u_{n-2} ... v_1 v_0)$ (в школе это учат делать «подбором»).

Например, при делении 3157 на 59 с остатком сначала $315 = 59 \cdot 5 + 20$, так что неполное частное $t = 5$, остаток $r = 20$. На следующем шаге к остатку r дописывается следующая цифра u_j делимого, что означает деление на 59 числа $r \cdot 10 + u_j$ (в общем случае при основании системы счисления q делится $r \cdot q + u_j$).

Подбор t может осуществляться циклической процедурой последовательного вычитания n-разрядного делителя из $(n+1)$-разрядного очередного остатка $r' = r \cdot q + u_j$ вплоть до получения отрицательной разности. Более прямой метод, использующий старшие разряды r' и делителя, описан у Кнута ([8], с. 310 и далее).

3.3. Арифметика чисел с плавающей точкой

3.3.1. Представление чисел в форме с плавающей точкой

Запись числа a *в форме с плавающей точкой* (или *с плавающей запятой*) имеет вид

$$a = m \cdot q^p = \pm|m| \cdot q^p, \qquad (28)$$

где q — основание системы счисления (обычно 2, реже 10), m — *мантисса* (или *дробная часть*), удовлетворяющая неравенству $|m| < 1$, p — целое число со знаком, называемое *порядком*.

Для представления мантиссы и порядка отводится *машинное слово* с фиксированным (для конкретного типа компьютеров) числом $n = s' + s''$ q-ичных разрядов: s' разрядов для мантиссы, s'' разрядов для порядка.

Представление (28) называется *нормализованным*, если старшая цифра мантиссы отлична от нуля, что равносильно условию

$$q^{-1} \le |m| < 1.$$

Пример. Записи $a = 0.00237 \cdot 10^3$ и $a = 0.0237 \cdot 10^2$ являются ненормализованными представлениями числа $a = 2.37$, а запись $a = 0.237 \cdot 10^1$ — нормализованным представлением.

Если для представления модуля мантиссы отводится s разрядов, то в случае нормализованного представления

$$q^{-1} \le |m| \le 1 - q^{-s}.$$

Замечание. Нормализованность представления может определяться и по-другому для разных типов компьютеров. При этом по умолчанию подразумеваются те или иные соглашения.

Сравнение положительных нормализованных чисел по величине сводится к сравнению их порядков, и лишь при равенстве порядков — к сравнению мантисс:

$$m_1 q^{p_1} < m_2 q^{p_2} \Leftrightarrow (p_1 < p_2) \text{ или } (p_1 = p_2 \text{ и } m_1 < m_2).$$

Дискретность множества чисел с плавающей точкой обусловлена конечностью множества представляемых чисел (*машинных чисел*) при фиксированном количестве q-ичных разрядов для мантиссы и для порядка. Отсюда вытекает существование наименьшего (ближайшего к нулю) положительного машинного числа. Это число ε (*машинное эпсилон*) представляется мантиссой, содержащей единицу в старшем разряде (требование нормализованности) и нули в остальных разрядах и минимально возможным при данной разрядности отрицательном порядке $p_{\min}$:

$$\varepsilon = 0.1 \cdot q^{p_{\min}}.$$

Сетка машинных чисел с плавающей точкой меняется с ростом порядка: числа с малыми порядками расположены гуще, числа с большими порядками — реже. Наибольшее машинное число имеет мантиссу, состоящую из старших цифр q-ичной системы (из единиц для двоичной системы, из девяток для десятичной), и максимально возможный порядок $p_{\max}$.

Свойства арифметических операций. Дискретность множества машинных чисел приводит к тому, что *арифметические операции с ними*

выполняются лишь приближённо, так что для них вводятся новые обозначения, например, с точкой: $\dot{+}, \dot{-}, \dot{\times}, \dot{/}$ вместо стандартных символов $+, -, \times, /$.

Для этих новых операций уже *могут не выполняться привычные законы арифметических действий*, прежде всего законы ассоциативности: порядок выполнения операций, то есть способ расстановки скобок, может существенно повлиять на результат.

3.3.2. Сложение чисел в форме с плавающей точкой

При обсуждении машинной реализации действий над нормализованными числами с плавающей точкой следует иметь ввиду, что «…В стандартных подпрограммах… на машинном языке в очень большой степени используются крайне специфические особенности конкретной модели компьютера. Именно поэтому так мало сходства между двумя подпрограммами, скажем, сложения чисел с плавающей точкой, написанными для разных машин» ([8], с. 249).

Результатом сложения двух нормализованных чисел в форме с плавающей запятой должно быть число с таким же видом машинного представления.

Алгоритм сложения чисел $m_1 \cdot q^{p_1} \dot{+} m_2 \cdot q^{p_2}$ основан на равенстве

$$m_1 q^{p_1} + m_2 q^{p_2} = (m_1 + m_2 q^{p_2 - p_1}) q^{p_1}. \qquad (29)$$

Таким образом, в качестве порядка для суммы можно принять порядок одного из слагаемых, а в качестве мантиссы — сумму мантиссы этого слагаемого и изменённой мантиссы другого слагаемого. Принято брать за порядок суммы больший из двух исходных порядков.

Алгоритм сложения нормализованных чисел.

╔

Ш1. *Помещение слагаемого с бóльшим порядком на первое место в сумме* (*для определённости дальнейших действий*). Если $p_1 < p_2$, то слагаемые переставляются. После этого можно считать, что $p_1 \geq p_2$.

Ш2. *Назначение порядка* p *для суммы* по бóльшему порядку: $p := p_1$.

Ш3. *Выравнивание разделяющих точек* (и, тем самым, положения разрядов в мантиссах слагаемых): $m_2 := m_2 / q^{p_1 - p_2}$. Фактически это означает, что после десятичной точки дописывается $p_1 - p_2$ нулей, но предполагается, что новая мантисса умножена уже не на q^{p_2}, а на q^{p_1}.

Ш4. *Сложение мантисс* с выравненными разрядами:

$$m := m_1 + m_2 / q^{p_1 - p_2}.$$

Ш5. *Нормализация результата*. Модуль мантиссы суммы может в данный момент содержать более чем s' полагающихся цифр, то есть может оказаться, что $|m| \geq 1$ или $|m| < q^{t_{\min}}$.

╚

Алгоритм нормализации суммы.

╔

Ш1. *Проверка ненормализованности с превышением.* Если $|m| \geq 1$ (переполнение дробной части), перейти к шагу **Ш4.**

Ш2. *Проверка ненормализованности с недостатком.* Если $|m| \geq q^{-1}$ (то есть мантисса нормализована), переход к шагу **Ш5** (к округлению).

Ш3. Масштабирование сдвигами влево. Сейчас $|m| < q^{-1}$, то есть первые разряды мантиссы — нулевые. Мантисса m сдвигается влево на один разряд относительно десятичной точки перед первым её разрядом (то есть умножается на число q: $m := mq$), но при этом для сохранения равенства порядок p уменьшается на единицу: $p := p - 1$. Возврат к шагу **Ш2** вплоть до обеспечения нормализованности.

Ш4. Масштабирование сдвигами вправо. Сейчас $|m| \geq 1$, то есть у мантиссы ненулевая целая часть. Мантисса m сдвигается вправо на один разряд относительно подрузамеваемой десятичной точки (то есть умножается на q^{-1}: $m := mq^{-1}$), но при этом для сохранения равенства порядок p увеличивается на единицу: $p := p + 1$. Возврат к шагу **Ш1** вплоть до обеспечения нормализованности.

Ш5. Округление мантиссы до p разрядов, то есть мантиссе присваивается значение ближайшего кратного q^{-p} (кратного единице младшего разряда). Возможны разные способы округления, когда таких ближайших числа оказывается два.

Ш6. Проверка допустимости порядка. Если в результате предшествующих действий окажется, что порядок p вышел за допустимый диапазон («переполнение» либо «исчезновение» порядка), то выдаётся сигнал об ошибке.

╚

3.3.3. Умножение и деление чисел в форме с плавающей точкой

Умножение. Для машинного умножения чисел, представленных в форме с плавающей точкой, достаточно перемножить мантиссы (как перемножаются целые числа и числа с фиксированной точкой) и сложить порядки:

$$\left(m_1 q^{p_1}\right) \cdot \left(m_2 q^{p_2}\right) = (m_1 m_2) q^{p_1 + p_2}.$$

После этого проводится корректировка результата, обеспечивающая нормализованность мантиссы $m_1 m_2$ и допустимый диапазон для порядка $p_1 + p_2$. Если суммарный порядок $p_1 + p_2$ меньше минимально допустимого, то произведение представляется машинным нулём. Если суммарный

порядок $p_1 + p_2$ больше максимально допустимого, то выдаётся сигнал об ошибке, если только приведение мантиссы к нормализованному виду путем соответствующего изменения порядка не вернёт последний в допустимый диапазон.

Деление. Деление $m_1 q^{p_1} : m_2 q^{p_2} = m q^{p}$ выполняется в четыре этапа.

1. Определение предварительного порядка частного: $p := p_1 - p_2$. Если порядок частного окажется выше допустимого предела, то деление завершается аварийно. Если порядок частного меньше минимально допустимого, то он запоминается, поскольку последующая нормализация мантиссы может вернуть его в требуемый диапазон.

2. Вычисление частного мантисс: $m := m_1 / m_2$ по правилу деления целых чисел (и чисел с фиксированной точкой).

3. Нормализация результата путём сдвига мантиссы и соответствующего изменения порядка (она аналогична нормализации при умножении).

4. Завершающее округление мантиссы.

3.3.4. Точность арифметических операций с числами в форме с плавающей точкой

Дискретность множества рациональных чисел, представленных в форме с плавающей запятой, означает, что арифметические операции с ними выполняется приближённо. Поэтому отклонение полученного машинного результата от истинного может оказаться значительным. Точный анализ возникающих ошибок при большом объёме вычислений затруднителен. Можно сформулировать эмпирическое (то есть основанное на практике) правило:

Сложение чисел с одинаковыми знаками и умножение в формате с плавающей точкой, как правило, не ведут к существенной потери точности результата. В то же время вычитание близких чисел одного знака может привести к значительному искажению результата.

Последнее имеет место, когда выравнивание порядков в (29) приводит к большому сдвигу разрядов мантиссы m_2 за пределы разрядной сетки. Возможным выходом в такой ситуации может служить принятие соглашения о том, что дробная часть может быть ненормализованной, если порядок p имеет минимально возможное значение.

Характеристики точности представления. Пусть вещественное число a представлено в компьютере приближённо (всегда рациональным) числом $\tilde{a}$.

Абсолютной ошибкой (*абсолютной погрешностью*) представления называется число $\Delta = |\tilde{a} - a|$.

Относительной ошибкой (*относительной погрешностью*) представления называется число $\delta = (\tilde{a} - a)/a$. При этом $\tilde{a} = a(1 \pm \delta)$.

Значащими цифрами мантиссы нормализованного представления называются все цифры за исключением нулей в конце. Это связано с тем, что практика доставляет числа (например, физические константы в системе единиц СИ) в форме с фиксированной точкой и с несколькими цифрами после точки. Требование нормализованности мантиссы при машинном представлении вынуждает сдвигать точку влево, а также дописывать несколько нулей в конце и, соответственно, изменять порядок, поскольку должна быть заполнена вся разрядная сетка. В таком случае нули в конце мантиссы не несут никакой информации. Например, значение числа Авогадро (официально рекомендованное в 2010 г.) есть $6.02214129 \cdot 10^{23}$. Если для мантиссы отведено одиннадцать разрядов, то нормализованное представление имеет вид $0.60221412900 \cdot 10^{24}$, при том, что фактически две последние цифры мантиссы неизвестны.

3.4. Вычисления с многократной точностью

Разрядность машинного слова определяет стандартную точность арифметических операций с плавающей точкой. В случаях, когда требуется проводить вычисления с большей точностью, для представления чисел в памяти отводится два или более машинных слов и арифметические операции реализуются уже программными (а не аппаратными) средствами.

Такие вычисления могут проводиться при работе с числами как с плавающей (чаще), так и с фиксированной запятой.

Для чисел с фиксированной точкой потребность в увеличении точности связана, прежде всего, с потребностью в большом числе точных цифр дробной части в исходных данных. Для чисел с плавающей точкой, бывает необходимо расширить как число разрядов мантиссы, так и диапазон значений порядка.

3.4.1. Вычисления с большими целыми числами

Пусть для представления большого целого числа в q-ичной системе счисления используется n k-разрядных машинных слов. Отдельное слово может представлять q^k различных значений (числа от 0 до $q^k - 1$), и поэтому его можно рассматривать как один разряд системы счисления с основанием $b = q^k$. Тогда n машинных слов представляют n-разрядное b-ичное число, и арифметические действия, осуществляемые поразрядно («в столбик»), сводятся к следующим элементарным операциям:

– сложение одноразрядных чисел (и вычитание как сложение с обратным знаком) с получением переноса (0 или 1);

– умножение одноразрядных чисел с получением двухразрядного результата («два пишем, три в уме»);

– деление двухразрядного числа на одноразрядное с получением одноразрядного неполного частного и одноразрядного остатка.

В терминах этих операций формулируются алгоритмы арифметических действий с многоразрядными числами. Поскольку основание b системы счисления отлично от 2, то не применимы «быстрые» двоичные операции сложения/вычитания/умножения, сводимые к сложениям и сдвигам. Приходится использовать традиционные способы выполнения операций по схеме «в столбик». Что касается операции деления, то она и в двоичной арифметике выполняется «в столбик» и является наиболее трудоёмкой.

Сложение неотрицательных чисел. Пусть $u=\sum_{i=0}^{n-1}u_i b^i$, и $v=\sum_{i=0}^{n-1}v_i b^i$, так что в b-ичной записи $u=(u_{n-1},...,u_1,u_0)_b$, $v=(v_{n-1},...,v_1,v_0)_b$.

Алгоритм сложения формирует в b-ичной записи сумму $w=u+v=(w_n,w_{n-1},...,w_1,w_0)_b$. Введём вспомогательные переменные: i — номер текущего разряда, λ — признак переноса, возникающего в текущем разряде (при сложении младших, нулевых разрядов переноса ещё не было: $\lambda=0$). При сложении в текущем разряде $u_i+v_i+\lambda\le(b-1)+(b-1)+1=2b-1$, так что перенос в следующий разряд — целая часть частного $(u_i+v_i+\lambda)/b$ — либо нуль, либо единица.

Ш.1. Начальные присваивания. $i:=0$; $\lambda:=0$.

Ш.2. Сложение разрядов с текущим номером i:

$w_i:=(u_i+v_i+\lambda)\bmod b$ — цифра в текущем разряде;

$\lambda:=\lfloor(u_i+v_i+\lambda)/b\rfloor$.

Ш.3. $i:=i+1$;

если $i<n$, то вернуться к **Ш.2**,

в противном случае $w_n:=\lambda$ — перенос при сложении старших разрядов.

Вычитание неотрицательных чисел. Пусть в b-ичной записи $u=(u_{n-1},...,u_1,u_0)_b$, $v=(v_{n-1},...,v_1,v_0)_b$. Поскольку $u-v=-(v-u)$, достаточно иметь алгоритм для случая $u\ge v$ с неотрицательной разностью.

Алгоритм вычитания формирует в b-ичной записи неотрицательную разность $w=u-v=(w_{n-1},w_{n-2},...,w_1,w_0)_b$. Введём вспомогательные переменные: i — номер текущего разряда, λ — признак необходимости заимствования из следующего, старшего разряда: λ присваивается значение -1, если заимствование необходимо, и значение 0, если такой необходимости нет. При вычитании в текущем разряде это значение уже было сформировано вычитанием в предшествующем $(i-1)$-м разряде; поэтому *новое значение* λ*, формируемое в текущем разряде, полагается равным* -1*, если при текущем значении* λ *выполняется* $u_i-v_i+\lambda<0$. Признак

$\lambda = -1$ играет роль точки, которая ставится над цифрой уменьшаемого слева от текущего разряда, если нужно «занимать единицу».

Наименьшая цифра u_i в текущем разряде уменьшаемого есть 0. Наибольшая цифра v_i в текущем разряде вычитаемого есть $b-1$. Разность между ними ещё уменьшается на единицу, если было заимствование (если над u_i стоит точка при вычитании в столбик вручную). Итак, наименьшее значение разности в текущем разряде есть $0-(b-1)+(-1)=-b$. В свою очередь, наибольшее значение разности в текущем разряде получается, когда из максимально возможной цифры $(b-1)$ уменьшаемого вычитается 0, и не было заимствования (то есть $\lambda = 0$). Поэтому при формировании разности в текущем i-м разряде выполняется неравенство

$$-b \le u_i - v_i + \lambda < b \underset{+b}{\Leftrightarrow} 0 \le u_i - v_i + \lambda + b < 2b.$$

Заимствование производится, если

$$-b \le u_i - v_i + \lambda < 0 \Leftrightarrow 0 \le u_i - v_i + \lambda + b < b \Leftrightarrow$$
$$\Leftrightarrow [(u_i - v_i) / b] + 1 = 0 \quad b = .$$

Заимствование не производится, если

$$0 \le u_i - v_i + \quad < b \Leftrightarrow b \le u_i - v_i + \quad + b < 2b \Leftrightarrow$$
$$\Leftrightarrow [(u_i - v_i) / b] + 1 = 1 \quad b = .$$

Таким образом, значение признака заимствования совпадает с числом $-1+[(u_i - v_i + \quad + b \quad b$.

Ш.1. Начальные присваивания $i := 0$; $\quad 0 =$.

Ш.2. Вычитание разрядов с текущим номером i:

$w_i := \lambda(u_i \bmod 1 + \quad b$;

$\lambda := -1 + [(u_i - v_i + \lambda) / b]$.

Ш.3. $i := i + 1$;

если $i < n$, то вернуться к **Ш.2.**

Умножение неотрицательных чисел. Пусть в b-ичной записи $u = (u_{m-1},...,u_1,u_0)_b$, $v = (v_{n-1},...,v_1,v_0)_b$.

Алгоритм умножения формирует в b-ичной записи неотрицательное произведение $w = u \cdot v = (w_{m+n-1},...,w_1,w_0)_b$ как сумму попарных произведений: $w_l = \sum_{i+j=l} u_i b^i \cdot v_j b^j = \sum_{i+j=l} u_i v_j b^{i+j}$ с учётом того, что в разряд b^{i+j} мог придти ненулевой перенос λ из предыдущего произведения $u_i b^{i-1} \cdot v_j b^j$ $(0 \le \quad < b)$ и что может оказаться $u_i v_j \ge b$.

Рассмотрим для примера в случае $b = 10$ произведение двух разрядов $(7 \cdot 10^3) \cdot (5 \cdot 10^6)$ с переносом $\lambda = 2$ из предыдущего произведения. В результате получаем

$$(7\cdot 10^3)\cdot(5\cdot 10^6)+\lambda\cdot 10^9 = 37\cdot 10^9 = 3\cdot 10^{10}+7\cdot 10^9.$$

Множитель 3 при 10^{10} (это перенос λ в следующий разряд) есть целая часть от $37/10$, то есть $\lambda=[37/10]$. Множитель 7 при 10^9 (в текущем разряде) есть остаток от деления 37 на 10, то есть $37 \bmod 10$.

Для накопления итогового коэффициента w_9 при 10^9 (поскольку $10^9 = 10^3\cdot 10^6 = 10^4\cdot 10^5$ и т. д.) его начальное значение полагается, как обычно при циклическом сложении, равным нулю.

╔

Ш.1. Начальные присваивания.

$w_{m+n-1} := 0, ..., w_0 := 0$ (для накопления сумм);

$j := 0$ — начальный номер разряда множителя v, параметр внешнего цикла.

Ш.2. $i := 0$ — параметр внутреннего цикла, номер начального разряда множителя u;

$\lambda := 0$ — обнуление переноса.

Ш.3. $t := w_{i+j} + u_i v_j +$ — произведение текущих разрядов с учетом уже состоявшегося переноса без разбивки на два разряда b^{m+i+1} и b^{i+j};

разбивка на разряды: $w_{i+j} := t \bmod b$;

$\lambda := [t/b]$ — перенос в следующий разряд.

Ш.4. $i := i+1$;

если $i < m$, то

вернуться к **Ш.3**,

в противном случае

$w_{j+m} := \lambda$ — перенос, возникающий при перемножении старших разрядов.

Ш.5. $j := j+1$;

если $j < n$, то

вернуться к **Ш.2**.

╚

3.4.2. Модулярная арифметика

При выполнении арифметических действий можно оперировать не с самими числами, а с наборами их остатков по выбранной совокупности взаимно простых делителей. Теоретическим основанием этого является китайская теорема об остатках [19]:

Пусть натуральные числа $m_1, m_2, ..., m_k$ *попарно взаимно просты. Тогда*:

1. *Для любых целых чисел* $a_1, a_2, ..., a_k$ *таких, что* $0 \le a_i < m_i$, $i = 1,2,...,k$, *найдётся число* a, *которое при делении на* m_i *даёт остаток* a_i.

2. *Если числа* a *и* a' *обладают указанным свойством, то* $a \equiv a' (\mathrm{mod}\, m_1 m_2 ... m_k)$.

Геометрически теорема означает, что откладывая на числовой оси от начальной точки a_i последовательно n_i раз отрезок длиной m_i, можно придти при всех $i = 1,2,...,k$ к одной и той же точке a числовой оси. Из теоремы следует также, что в диапазоне $1 \le a \le M$, где $M = m_1 m_2 ... m_k$, каждое натуральное число a однозначно определяется остатками по взаимно простым модулям $m_1, m_2, ..., m_k$. С другой стороны, если известен только набор остатков $(a_1, a_2, ..., a_k)$, но нет сведений о реальной величине числа a, восстановление числа a возможно только с точностью до кратного числа M: a и $a + tM$ имеют одинаковый набор остатков $(a_1, a_2, ..., a_k)$.

Напомним некоторые свойства сравнений:

1. $a \equiv b (\mathrm{mod}\, m) \Leftrightarrow m \mid (a - b)$.

2. $a \equiv b (\mathrm{mod}\, m) \Leftrightarrow a = b + mk$, где $k \in \mathbb{Z}$.

3. *Пусть* $a \equiv c (\mathrm{mod}\, m)$ *и* $b \equiv c (\mathrm{mod}\, m)$. *Тогда* $a \equiv b (\mathrm{mod}\, m)$.

4. *Сравнения можно почленно складывать:*

$a_1 \equiv b_1 (\mathrm{mod}\, m), a_2 \equiv b_2 (\mathrm{mod}\, m) \Rightarrow a_1 + a_2 \equiv (b_1 + b_2) (\mathrm{mod}\, m)$.

5. $a \equiv b (\mathrm{mod}\, m) \Rightarrow a + mt \equiv b (\mathrm{mod}\, m)$, $t \in \mathbb{Z}$.

6. *Сравнения можно почленно перемножать:*

$a_1 \equiv b_1 (\mathrm{mod}\, m), a_2 \equiv b_2 (\mathrm{mod}\, m) \Rightarrow a_1 a_2 \equiv b_1 b_2 (\mathrm{mod}\, m)$.

7. *Обе части сравнения можно возводить в степень:*

$a \equiv b (\mathrm{mod}\, m) \Rightarrow a^n \equiv b^n (\mathrm{mod}\, m)$.

8. *Обе части сравнения можно умножить на одно и то же число:*

$a \equiv b (\mathrm{mod}\, m) \Rightarrow ac \equiv bc (\mathrm{mod}\, m)$.

9. *Обе части сравнения можно сокращать на одно и то же число, взаимное простое с модулем:*

Если $(c, m) = 1$ *и* $ac \equiv bc (\mathrm{mod}\, m)$, *то* $a \equiv b (\mathrm{mod}\, m$.

10. *Если* $ak \equiv bk (\mathrm{mod}\, mk)$, *то* $a \equiv b (\mathrm{mod}\, m)$, *то есть, обе части сравнения вместе с модулем можно разделить на одно и то же число.*

11. *Пусть* M — *наименьшее общее кратное чисел* $m_1, ..., m_n$. *Если*

$$\begin{cases} a \equiv b(\mathrm{mod}\, m_1), \\ a \equiv b(\mathrm{mod}\, m_2), \\ \quad . \\ \quad . \\ \quad . \\ a \equiv b(\mathrm{mod}\, m_n), \end{cases}$$

то $a \equiv b(\mathrm{mod}\, M)$, *то есть, если сравнение имеет место по нескольким модулям, то оно имеет место по модулю их НОК.*

12. *Пусть* $a \equiv b(\mathrm{mod}\, m)$ *и* $d \mid m$. *Тогда* $a \equiv b(\mathrm{mod}\, d)$, *то есть, если сравнение имеет место по модулю* m, *то оно имеет место по модулю любого его делителя* d.

13. *Если* $a \equiv b(\mathrm{mod}\, m)$, $d \mid a$ *и* $d \mid m$, *то* $d \mid b$, *то есть, если одна часть сравнения и модуль делятся на* d, *то и другая часть сравнения делится на* d.

Из свойств сравнений следует, что для чисел a и b, представленных наборами остатков $(a_1, a_2, ..., a_k)$ и $(b_1, b_2, ..., b_k)$ можно выполнять сложение, умножение и вычитание в виде:

$$a \pm b \to (a_1, a_2, ..., a_k) \pm (b_1, b_2, ..., b_k) \to$$
$$((a_1 \pm b_1) \bmod m_1, ..., (a_k \pm b_k) \bmod m_k);$$
$$a \cdot b \to (a_1, a_2, ..., a_k) \cdot (b_1, b_2, ..., b_k) \to$$
$$((a_1 b_1) \bmod m_1, ..., (a_k b_k) \bmod m_k).$$

Недостатком модулярной арифметики является невозможность непосредственного сравнения по величине двух чисел, представленных наборами остатков. Кроме того, модулярные операции обычно сочетаются с операциями, предполагающими обычное представление (например, если необходимо выполнять деление). Модулярные операции существенно снижают затраты времени, если аппаратная реализация позволяет производить параллельно вычисления со всеми остатками.

3.4.3. Ускорение процесса умножения

При стандартном методе умножения n разрядных чисел «в столбик» число выполняемых операций пропорционально n^2, так что затрачиваемое время $T(n) = O(n^2)$. Это время можно уменьшить, сокращая количество промежуточных умножений.

Пусть нужно перемножить в q-ичной системе два числа a и b, которые имеют по $2n$ разрядов:

$$a = (a_{2n-1} ... a_1 a_0),\ b = (b_{2n-1} ... b_1 b_0).$$

Выделяя n старших разрядов, можно представить их в виде:

$$a = a' q^n + a'',\ b = b' q^n + b'',$$

где $a' = (a_{2n-1}...a_n)$, $b' = (b_{2n-1}...b_n)$ — «более значимые половины» q-ичного представления, а $a'' = (a_{n-1}...a_0)$ и $b'' = (b_{n-1}...b_0)$ — «менее значимые половины». Имеет место равенство:

$$ab = (q^n a' + a'')(q^n b' + b'') = (a'b')q^{2n} + (a'b'' + a''b')q^n + a''b'', \quad (*)$$

которое сводит умножение $2n$-битовых чисел к четырём умножениям n-битовых и к нескольким операциям сложения и сдвига (умножение на степень основания q сводится в q-ичном представлении к сдвигу цифр на соответствующее число позиций влево и дописыванию справа нулей). Количество умножений можно уменьшить до трёх, используя дополнительные более быстрые операции сложения и вычитания. Именно,

$$a'b'' + a''b' = (a' + a'')(b' + b'') - a'b' - a''b'',$$

где $a'b'$ и $a''b''$ *уже вычислены*. Подставляя в (*) правую часть последнего равенства вместо левой, получаем окончательную формулу для быстрого умножения:

$$ab = a'b'q^{2n} + \left((a' + a'')(b' + b'') - a'b' - a''b''\right)q^n + a''b''.$$

3.5. Эффективные алгоритмы вычисления степеней

3.5.1. Бинарный метод

С целью сокращения операций умножения целесообразно по возможности часто использовать в промежуточных вычислениях возведение в квадрат: вместо цепочки действий при вычислении x^n вида

$$x \to x \cdot x = x^2 \to x^2 \cdot x = x^3 \to ... \to x^{15} \cdot x = x^{16}$$

можно провести вычисления по схеме

$$x \to x^2 \to (x^2)^2 = x^4 \to (x^4)^2 = x^8 \to (x^8)^2 = x^{16}.$$

Если показатель n не является степенью двойки, помимо возведений в квадрат приходится иногда производить умножение на x. Например, при вычислении x^{23}:

$$x \to x^2 \to (x^2)^2 = x^4 \to x^4 \cdot x = x^5 \to (x^5)^2 = x^{10} \to$$
$$\to x^{10} \cdot x = x^{11} \to (x^{11})^2 = x^{22} \to x^{22} \cdot x = x^{23}.$$

Последовательность возведений в квадрат и умножений может быть и другой. Например, после вычисления x^{10} можно дальше вычислять

$$(x^{10})^2 = x^{20} \to x^{20} \cdot x = x^{21} \to x^{21} \cdot x = x^{22} \to x^{22} \cdot x = x^{23}.$$

Если не запоминать отдельно результаты промежуточных вычислений, то на каждом шаге мы имеем только исходное число x и его очередную степень. Стандартную последовательность действий можно получить исходя из двоичной записи показателя степени n следующим образом.

В двоичной записи:

$2 = (10)_2$; $x^2 = (x)^2$;

$3 = (11)_2$; $x^3 = x^{2+1} = x^2 \cdot x$;

$4=(100)_2;\ x^4=(x^2)^2;$

$5=(101)_2;\ x^5=(x^2)^2\cdot x.$

$6=(110)_2;\ x^6=(x^2\cdot x)^2.$

Порядок возведений в квадрат и умножений на первую степень x не зависит от старшей цифры двоичного представления показателя, которая всегда равна единице. Далее, при сканировании представления слева направо, очередной цифре 0 соответствует возведение уже накопленного результата в квадрат, а очередной цифре 1 соответствует возведение в квадрат и умножение на первую степень исходного основания x.

Данное правило соответствует представлению показателя n в виде

$$n=(u_k u_{k-1}...u_1u_0)_2=u_k\cdot 2^k+u_{k-1}\cdot 2^{k-1}+...+u_1\cdot 2+u_0=$$
$$((...((u_k\cdot 2+u_{k-1})\cdot 2+u_{k-2})\cdot 2+...)\cdot 2+u_1)\cdot 2+u_0,\ u_i\in\{0,1\}$$

При возведении в степень по заданному модулю m, то есть при вычислении $x^n \bmod m$ следует каждый промежуточный результат заменять остатком от деления на m. Это позволяет оперировать с числами в существенно меньшем диапазоне:

$$23^{10}\bmod 17=(23^2)^5\bmod 17=529^5\bmod 17=\dots$$

(поскольку $529\equiv 2 \bmod 17$)

$$\dots=2^5\bmod 17=32\bmod 17=15.$$

3.5.2. Метод факторизации показателя

Пусть составной показатель n разложен в произведение с участием своего простого множителя p: $n=pq$. Вычисление степени может быть проведено по схеме $x\to x^p\to\left(x^p\right)^q$.

Например, $x^{77}=x^{7\cdot 11}=\left(x^7\right)^{11}$, где сначала вычисляется x^7 по схеме, изложенной выше:

$$x\to x^2\to x^2\cdot x=x^3\to\left(x^3\right)^2=x^6\to x^6\cdot x,$$

а затем по той же схеме вычисляется $\left(x^7\right)^{11}$.

3.6. Эффективные действия с дробями

Дробь a/b может быть представлена как упорядоченная пара чисел (первое число — числитель, второе — знаменатель). Для экономного представления естественно потребовать взаимную простоту a и b.

Умножение дробей, производимое по схеме

$$(a/b)\cdot(a'/b')=(aa')/(bb')$$

может нарушить взаимную простоту числителя и знаменателя для их произведения. Поэтому сначала находятся с помощью алгоритма Евклида наибольшие общие делители $d_1 = (a,b'),\ d_2 = (b,a')$,

так что

$$a = d_1 k, b' = d_1 l, b = d_2 m,\ a' = d_2 n,$$

и

$$(a/b)\cdot(a'/b') = (d_1 k \cdot d_2 n)/(d_2 m \cdot d_1 l) = kn/ml,$$

где целые числа k, l, m, n находятся предварительно делением «в столбик» (с нулевым остатком). Другой возможностью является отыскание наибольшего общего делителя $d = (aa', bb')$ и деление на него числителя aa' и знаменателя bb' результирующей дроби. Второй случай, однако, приводит к действиям с бóльшими числами

Деление дробей соответствует схеме

$$(a/b):(a'/b') = (ab')/(a'b),$$

и также предполагает обеспечение взаимной простоты числителя и знаменателя результирующей дроби одним из двух способов аналогично случаю произведения дробей.

Сложение и вычитание дробей соответствует схеме

$$(a/b)\pm(a'/b') = (ab' \pm a'b)/(bb'),$$

после чего дробь приводится к несократимому виду после отыскания наибольшего общего делителя числителя и знаменателя результирующей дроби. Можно также предварительно найти $d_1 = (b,b')$ и выполнить сложение/вычитание дробей с меньшими числами: $b = d_1 k,\ b' = d_1 k' \Rightarrow$ $a/(d_1 k) \pm a'/(d_1 k') =$
$= (ak' \pm a'k)/(d_1 kk')$. Пусть $ak' + a'k = t$ и $d_2 = (t, d_1)$, так что $t = d_2 l,\ d_1 = d_2 m$. Тогда $(a/b)\pm(a'/b') = l/(mkk')$, где l, m, k, k' могут оказаться существенно меньше исходных a, b, a', b', и можно показать, что дробь $l/(mkk')$ несократима.

3.7. Алгоритм Евклида для больших чисел

Традиционный алгоритм. Алгоритм Евклида для отыскания наибольшего общего делителя $d = (a,b)$ двух натуральных чисел a и b является циклическим процессом последовательных делений с остатком по схеме:

1) $r_0 = a,\ r_1 = b$;

2) при $i > 1$ $r_{i-1} = r_i q_i + r_{i+1},\ 0 \le r_{i+1} < r_i$.

При этом d равно последнему ненулевому остатку r_i (такой остаток обязательно найдётся, поскольку неотрицательные числа $r_1, r_2, \ldots$ строго убывают).

Разностный алгоритм. Если $a > b$, то $(a,b) = (a-b,b)$, и процесс замены большего из аргументов их разностью можно продолжать вплоть до получения нуля: при этом $(a',0) = a'$; $(0,b') = b'$. В этом случае не производится трудоёмкая операция деления с остатком, но число вычитаний может оказаться большим.

Бинарный алгоритм. Как отмечалось выше, операции умножения/деления на степень основания системы счисления (на 2 в случае двоичного представления) сводится к быстрым аппаратно реализуемым операциям сдвига всех цифр влево/вправо и дописыванию нуля на освободившееся место (вспомните школьные правила умножения на десять, а также деления на десять для чисел, оканчивающихся на нули). Бинарный алгоритм опирается на следующие свойства натуральных чисел:

1) $a = 2k, b = 2l \Rightarrow (a,b) = 2(k,l)$;

2) $a = 2k, b = 2l+1 \Rightarrow (a,b) = (k,b)$;

3) $a = 2k+1, b = 2l+1 \Rightarrow 2 \mid (a-b)$;

4) $|a-b| < \max(a,b)$.

Бинарный алгоритм на основании свойств 1) и 2) заменяет чётные аргументы их половинами, а в ситуации, когда оба нечётны, больший из них заменяется чётной разностью. При этом чётность числа устанавливается анализом младшей цифры двоичного или десятичного представления.

Теорема Ламе. Пусть a, b — целые числа, $a > b$, F_k — k-е число Фибоначчи. Тогда при $b < F_k$ алгоритм Евклида содержит не более $k-1$ итераций.

3.8. Алгоритмы разложения на простые множители

Как известно, натуральное число n единственным образом раскладывается в произведение степеней простых чисел:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k}. \qquad (30)$$

Для разложения на простые множители (называемого *факторизацией*) не известны быстро работающие при больших n алгоритмы. На этой трудности основан метод RSA шифрования с открытым ключом, когда всем известно, что большое число n (например, содержащее порядка 300 десятичных цифр), является произведением двух простых чисел, но отыскание последних для последующего взламывания шифра невозможно за приемлемое время при существующих вычислительных мощностях [11], [19].

Важную роль играет распределение простых чисел в натуральном ряду. Оно описывается функцией $\pi(n)$ — количество простых чисел, не превосходящих n. При больших n значение $\pi(n)$ приближённо равно $n / \ln n$, поскольку доказано предельное равенство:

$$\lim_{n\to\infty} \pi(n)/(n/\ln n) = 1.$$

На этом основании можно с некоторой условностью говорить, что вероятность извлечь наугад простое число из множества $\{1,2,...,n\}$ приближённо равна $1/\ln n$, а также что n-е простое число приближённо равно (лежит вблизи числа) $n \cdot \ln n$.

3.8.1.Метод пробных делений

Поскольку n содержит лишь конечное число простых множителей, разложение (30) можно получить, деля n многократно на последовательные простые числа, не превосходящие его:

$126 : 2 = 63$; 63 не делится на 2; $63 : 3 = 21$; $21 : 3 = 7$;

7 не делится на 3; 7 не делится на 5; $7 : 7 = 1$;

значит, $126 = 2 \cdot 3^2 \cdot 7$.

Наличие некоторых простых множителей (например, 2, 3, 5, 7, 11) может быть проверено с помощью признаков делимости. Для больших чисел такая проверка осуществляется быстрее, чем деление с остатком.

Этот метод крайне неэффективен, его реализация невозможна при больших n. Для его использования нужна таблица простых чисел, либо процедура проверка простоты числа, которая также является весьма трудоёмкой. Если p_k — наибольшее простое в разложении (30), то время реализации приближённо пропорционально $\max(p_{k-1}, \sqrt{p_k})$.

3.8.2. Метод Ферма

Метод Ферма основан на формуле разности квадратов: если нечётное $n = x^2 - y^2$, то $n = (x-y)(x+y) = ab$, где

$a = x - y,\ b = x + y \Leftrightarrow x = (a+b)/2,\ y = (b-a)/2$. Если множители $x-y, x+y$ нетривиальны (отличны от n и 1), то дальнейшая факторизация сводится к работе с меньшими числами. При этом $x^2 - n$ является полным квадратом: $x^2 - n = y^2$. Поиск такого x можно начинать с проверки наименьшего значения, при котором $x^2 - n \ge 0$, то есть с $x = \lceil \sqrt{n} \rceil$. Если $x^2 - n$ не является квадратом, то далее проверяются $x+1, x+2, ..., x+i, ...$ *Если при всех* $x = [\sqrt{n}]+1, ..., (n+1)/2$ *разность* $x^2 - n$ *не является квадратом, то* n *— простое* (см. [1]).

Пример. Факторизация числа $n = 11256$. Имеем $x = \lceil \sqrt{11256} \rceil = 107$;

$107^2 \ne 11256$;

$x^2 - 11256 = 193$ — не является квадратом;

$(107+1)^2 - 11256 = 408$ — не является квадратом;

$(107+2)^2 - 11256 = 625 = 25^2$.

Итак, $x = 109,\ y = 25,\ a = 84,\ b = 134,\ 11256 = 84 \cdot 134$.

Дальнейшая факторизация осуществляется просто: $84 = 2^2 \cdot 3 \cdot 7$, и $134 = 2 \cdot 67$. В итоге $11256 = 2^2 \cdot 3 \cdot 7 \cdot 67$.

3.8.3. Метод Крайчика

В этом методе нетривиальный делитель d числа n ищется как НОД числа n и числа $x - y$: $d = (n, x - y)$; при этом x и y подбираются так, чтобы заведомо выполнялось условие $1 < d < n$. Имеет место

Теорема. *Пусть выполняются три условия*:

1) x *и* y *лежат в разных классах вычетов по модулю* n;

2) x *и* $-y$, *лежат в разных классах вычетов по модулю* n;

3) $x^2 \equiv y^2 (\mathrm{mod}\, n)$.

Тогда $1 < (n, x - y) < n$.

Пусть, $u_i = [\sqrt{n}] + i$. Численные эксперименты показывают, что в разложении на множители чисел $v_i = (u_i^2 - n) \bmod n$ (остатков от деления $u_i^2 - n$ на n, так что $v_i \equiv u_i^2 (\mathrm{mod}\, n)$) довольно часто участвуют только небольшие простые множители (такие числа называют *гладкими*). Путём перемножения некоторых из сравнений

$$u_s^2 \equiv v_s (\mathrm{mod}\, n), \ldots u_t^2 \equiv v_t (\mathrm{mod}\, n),$$

в правой части удаётся получить полный квадрат: $v_s \ldots v_t = y^2$. Тогда в левой части сравнения квадрат получается автоматически: $x^2 \equiv y^2 (\mathrm{mod}\, n)$, где $x = u_s^2 \ldots u_t^2 = (u_s \ldots u_t)^2$.

Если при этом для x и y оказываются выполненными первые два условия теоремы, то наибольший общий делитель $d = (n, x - y)$ (то есть нетривиальный делитель n) находится с помощью алгоритма Евклида или каким-нибудь другим методом (см. п. 3.7).

Пример. Рассмотрим факторизацию числа $n = 2041$. Здесь $[\sqrt{2041}] = 45$. Для чисел $u_i = ([\sqrt{2041}] + i)^2$ при $i = 1, 2, \ldots$ последовательно получаем:

$$46^2 - 2041 = 75 = 3 \cdot 5^2;$$
$$47^2 - 2041 = 168 = 2^3 \cdot 3 \cdot 7;$$
$$48^2 - 2041 = 263;$$
$$49^2 - 2041 = 360 = 2^3 \cdot 3^2 \cdot 5;$$
$$50^2 - 2041 = 459 = 3^3 \cdot 17;$$
$$51^2 - 2041 = 560 = 2^4 \cdot 5 \cdot 7.$$

Полным квадратом оказывается произведение

$$75\cdot 168\cdot 360\cdot 560 = 2^{10}\cdot 3^4\cdot 5^4\cdot 7^2 = (2^5\cdot 3^2\cdot 5^2\cdot 7)^2 .$$

Теперь

$$x = 46\cdot 47\cdot 49\cdot 51 = 5402838 \equiv 311 (\mathrm{mod}\, 2041),$$

$$y = 2^5\cdot 3^2\cdot 5^2\cdot 7 = 50400 \equiv 1416 (\mathrm{mod}\, 2041),$$

и полагаем заново

$$x = 1416,\ y = 311 \Rightarrow (1416 - 311, 2041) = 13 .$$

Проверяем: $2041 = 13\cdot 157$.

Глава 4. СОРТИРОВКА В ОПЕРАТИВНОЙ ПАМЯТИ

4.1. Исходные понятия

Будем рассматривать задачу упорядочения совокупности данных $R_1,...,R_N,$, имеющих единообразную структуру вида [поле значений V_i]+[ключ K_i] $(i=1,...,N)$, в предположении, что на множестве значений ключей имеется отношение порядка "<", удовлетворяющее двум условиям:

- закон *трихотомии*: для любых значений K' и K'' выполняется ровно одно из соотношений $K'<K''$, $K'=K''$, $K''<K'$;

- закон *транзитивности*: $(K<K'$ и $K'<K'') \Rightarrow K<K''$.

Будем писать $K' \le K''$, если $K'=K''$ или $K'<K''$.

Сортировкой называется упорядочение данных в порядке возрастания или убывания значений ключевого поля (для краткости будем говорить о возрастании или убывании самого ключа). Таким образом, при решении задачи сортировки ищется такая перестановка $(p_1, p_2,..., p_N)$ исходного множества номеров $(1,2,...,N)$, при которой

$$K_{p_1} \le K_{p_2} \le ... \le K_{p_N}.$$

Сортировка называется *устойчивой*, если она сохраняет относительный порядок для одинаковых значений ключа:

$$(i<j \text{ и } K_i=K_j) \Rightarrow p_i<p_j.$$

Устойчиво сортирующая перестановка $(p_1, p_2,..., p_N)$ единственна. Требование устойчивости существенно, если данные уже упорядочены по одному или нескольким другим (вторичным) ключам, не влияющим на основной ключ, и нежелательно разрушать эту упорядоченность.

Алгоритмы сортировки находят большое практическое применение. Они всегда сопровождают процессы обработки и хранения больших объемов информации. Выбор конкретного метода сортировки зависит как от структуры обрабатываемых данных (массивы, списки, деревья и т. д.), так и от архитектуры используемого компьютера.

Алгоритмы сортировки делят на два класса:

- *внутренняя сортировка*, при которой все данные хранятся и перемещаются в оперативной (быстрой) памяти с произвольным доступом — в этом случае упорядочиваемые данные будем называть массивом;

- *внешняя сортировка*, когда из-за слишком большого объёма данные размещаются на внешних носителях — в этом случае упорядочиваемые данные будем называть файлом.

В случае внутренней сортировки важно по возможности минимизировать количество наиболее часто повторяемых действий — сравнений и обменов. Для внешней сортировки важно минимизировать число обращений к устройствам с медленным доступом (магнитным лентам, барабанам, дискам).

Рационально построенные алгоритмы сортировки N элементов предполагают в среднем $\log N$ проходов по всей совокупности данных и затрачивают время $O(N \log N)$. Отметим, что выбор основания логарифма не играет существенной роли, поскольку логарифмы с разными основаниями пропорциональны, а оценка с использованием символа O производятся с точностью до постоянного множителя:

$$\log_a N = \log_a (b^{\log_b N}) = \log_a b \log_b N = c \log_b N,$$

где $c = \log_a b$.

Разные алгоритмы сортировки различным образом воздействуют на расположение сортируемых данных в памяти. Некоторые из них фактически переставляют элементы в памяти, то есть меняют их физические адреса. Другие сохраняют исходные физические адреса, но составляют дополнительную таблицу, содержащую сортирующую перестановку. В случае громоздкой структуры данных возможно также составление дополнительной таблицы, содержащей адресные ссылки на элементы и соответствующие значения ключа сортировки; после этого сортировка происходит только внутри этой таблицы. Последнее особенно предпочтительно, когда данные располагают несколькими ключами, и может потребоваться сортировка по различным комбинациям ключей.

Ключи сортировки часто не содержатся в исходной структуре данных в качестве отдельных полей, а имеют вид функций от поля (или полей) значений. Например, можно задать сортировку по количеству букв в поле “фамилия”, по лексикографической (алфавитной) упорядоченности “имён” и/или “отчеств” и т. п.

При сортировке списка возможно введение в каждый элемент вспомогательного поля связи, содержащего адрес следующего в указанном порядке элемента.

Методы сортировки разбиваются на несколько классов в зависимости от принципов, положенных в основу сортирующего алгоритма (рис. 22).

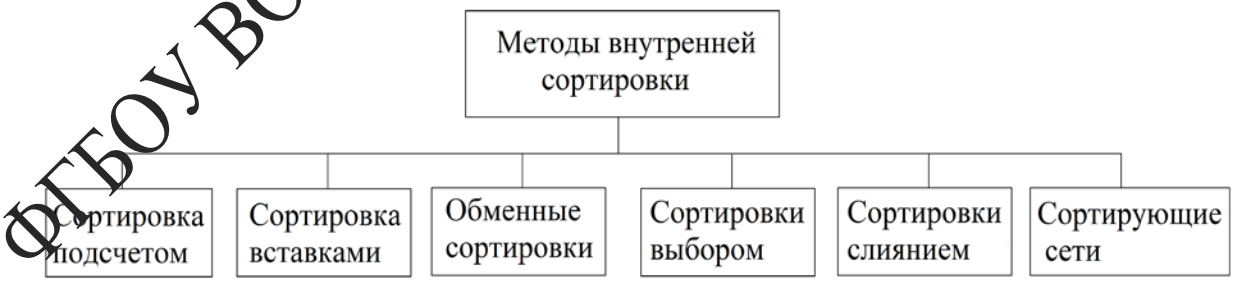


Рис. 22.

Внутри каждого класса содержатся методы, основанные на различных идеях (рис. 23).

Рис. 23.

4.2. Сортировка подсчётом

Будем сначала полагать, что все ключи исходного массива $K_1, K_2..., K_N$ различны.

Если ключ K_j должен занимать в отсортированном по возрастанию массиве место $p+1$ $(p \le 0 \le N-1)$, то это равносильно условию, что ровно для p элементов выполняется условие $K_i < K_j$. Поэтому сортировка сводится к подсчёту числа p выполненных неравенств; после этого значение $p+1$ заносится j-м элементом массива P, задающего упорядочение.

Заметим, что если K_i и K_j один раз сравнивались при исследовании ключа K_i, то сравнивать его повторно при исследовании ключа K_j нет необходимости. Кроме того не нужно сравнивать ключ с самим собой. Таким образом, достаточно обработать все пары индексов (i, j) с условием $j < i$.

Алгоритм сортировки подсчётом включает следующие этапы.

╔
- Объявление массива P с начальными нулевыми значениями $P_1, P_2,... P_N$ для подсчета ключей, меньших данного;
- внешний цикл по $i = N, N-1,..., 2$;
  - внутренний цикл по $j = i-1, i-2,..., 1$; **NB:** всегда $j < i$;
    - если $K_i < K_j$, то
      (а) P_j увеличивается на единицу,
    в противном случае, то есть при $K_i \ge K_j$,
      (б) P_i увеличивается на единицу.

╚

В процессе сортировки данные не перемещаются, но затрачивается дополнительная память для хранения массива P.

Если ключи могут иметь равные значения, то в теле внутреннего цикла будет реализовано действие (б), так что на единицу увеличится P_i, и K_i, который был изначально правее, окажется правее, чем K_j и после сортировки. Это означает устойчивость сортировки.

4.3. Сортировки вставками

В этой группе методов элементы данных просматриваются по одному, и каждый новый элемент вставляется в подходящее место среди тех, что были упорядочены ранее. Таким образом, в процессе сортировки массив разделён на две части: отсортированную и неотсортированную. На каждой итерации очередной элемент неотсортированной части переходит в отсортированную часть.

Сортировка вставками обычно применяется в ситуациях, когда уже имеется ранее отсортированный массив, к которому время от времени добавляются новые элементы.

4.3.1.Метод простых вставок

Перед рассмотрением элемента R_j элементы $R_1,...,R_{j-1}$ предполагаются уже упорядоченными, так что

$$K_1 \le K_2 \le ... \le K_{j-1}.$$

K_j сравнивается поочерёдно с $K_{j-1},K_{j-2},...,K_1$ до тех пор, пока не окажется, что R_j следует либо оставить на месте (если $K_{j-1} \le K_j$), либо вставить между R_i и R_{i+1} (если $K_i \le K_j < K_{i+1}$), либо, наконец, поставить крайним слева (если $K_j < K_1$). Поскольку после вставки все элементы $R_{i+1},R_{i+2},...$ должны переместиться вправо (иногда говорят «вверх»), удобно совмещать в теле цикла операции сравнения и перемещения.

Алгоритм сортировки методом простых вставок содержит следующие этапы.

╔

- внешний цикл по номеру j очередного вставляемого элемента R_j, начиная с $j=2$ $(2 \le j \le N)$:
  - $R := R_j;\ K := K_j$ — запоминание вставляемого элемента и его ключа;
  - внутренний цикл по i $(j-1 \ge i \ge 0)$: $i := j-1$
    - сравнение K с K_i (R сейчас занимает позицию $j-i$):
      - если $K > K_i$, то выход из цикла по i, так как R_j должен стоять на i-м месте;
      - если $K \le K_i$, то сдвиг R_i вправо на место R и уменьшение i:

$$R_{i+1} := R_i$$

$$i := i-1$$

      - если $i > 0$, то возврат к началу цикла по i;
      - если $i = 0$, то R занимает первую слева позицию:

$$R_{i+1} := R;$$

- зацикливание по j.

╚

Трудоёмкость метода: поскольку для ключа K_j равновероятно оказаться больше или меньше каждого из прежде упорядоченных ключей, то в среднем приходится выполнять $j/2$ сравнений, и общее количество итераций даётся выражением

$$(1+2+...+N)/2 = N(N+1)/2 = O(N^2).$$

4.3.2. Метод бинарных вставок

Результат проверки неравенства $K_j < K_{j-i}$ даёт наибольшую информацию, когда он позволяет отбросить максимально возможное число равновероятных вариантов (см. [17]). В силу симметрии число отбрасываемых вариантов следует по возможности уменьшать вдвое.

При сортировке бинарными вставками ключ K_j сравнивается с ключом K_t, номер которого t занимает среднее положение в ряду уже упорядоченных ключей $K_1,...,K_{j-1}$. Если $j-1$ нечётно, то $t=[j/2]+1$; если $j-1$ чётно, то можно выбрать в качестве t любой из двух номеров $\frac{j-1}{2}$ или $\frac{j-1}{2}+1$. После этого средний номер выбирается уже в (два раза более коротком) ряду $t, t+1,..., j-1$, если $K_t < K_j$, либо в ряду $1, 2,..., t$, если $K_t > K_j$, и т. д.

Этот метод экономит затраты времени на сравнение ключей, но не меняет количество перестановок элементов, которое по-прежнему есть $O(N^2)$.

4.3.3. Метод двухпутевых вставок

Этот метод ориентирован на минимизацию числа необходимых в среднем переписываний данных во время их сортировки.

Для размещения отсортированного массива $R_{p_1},...,R_{p_N}$ отводится область свободной памяти, и элемент R_1 помещается в её середину. Место для последующих элементов высвобождается путём сдвигов влево или вправо, в зависимости от того, в какую сторону число необходимых сдвигов меньше. По сравнению с методом простых вставок машинное время экономится примерно наполовину, хотя усложняется программа.

Пример. Для совокупности данных

$$504, 086, 513, 060, 907, 172, 896, 276, 653, 426, 152, 613, 700$$

метод двухпутевых вставок приводит последовательно к следующим спискам:

$$504$$
$$086,\ 504$$
$$086,\ 504, 513$$
$$060, 086, 504, 513$$
$$060, 086, 504, 513, 907$$

060, 086, 172, 504, 513, 907
060, 086, 172, 504, 513, 896, 907
060, 086, 172, 276, 504, 513, 896, 907
060, 086, 172, 276, 504, 513, 653, 896, 907
060, 086, 172, 276, 426, 504, 513, 653, 896, 907
060, 086, 152, 172, 276, 426, 504, 513, 653, 896, 907
060, 086, 152, 172, 276, 426, 504, 513, 613, 653, 896, 907
060, 086, 152, 172, 276, 426, 504, 513, 613, 653, 700, 896, 907

4.3.4. Метод Шелла (сортировка с убывающим смещением)

В этом методе экономия в среднем машинного времени достигается за счёт того, что переставляются не соседние, а значительно более удалённые друг от друга элементы. В то время как перестановка соседних элементов, нарушающих упорядоченность, уменьшает общее количество инверсий в массиве лишь на единицу, перестановка удалённых друг от друга элементов в методе Шелла может уменьшить его на большее число.

На промежуточных стадиях сортируются либо сравнительно более короткие массивы, либо уже сравнительно хорошо упорядоченные массивы с малым числом инверсий.

Если, например, исходный список содержит 16 элементов, то сначала — первый проход — упорядочиваются восемь двучленных списков со смещением на $h_1 = 8$:

$$(R_1, R_9), (R_2, R_{10}), ..., (R_8, R_{16}).$$

Затем на втором проходе упорядочиваются четыре четырёхчленных списка со смещением на $h_2 = 4$:

$$(R_1, R_5, R_9, R_{13}), ..., (R_4, R_8, R_{12}, R_{16})$$

На третьем проходе упорядочиваются два восьмичленных списка со смещением на $h_3 = 2$:

$$(R_1, R_3, R_5, R_7, R_9, R_{11}, R_{13}, R_{15})$$

и

$$(R_2, R_4, R_6, R_8, R_{10}, R_{12}, R_{14}, R_{16}).$$

Наконец, на последнем шаге сортируется весь список со смещением $h_4 = 1$.

В общем случае набор из t убывающих смещений $\{h_0, h_2, ..., h_{t-1} = 1\}$ для разбивки списка на части может выбираться разными способами и выступает в роли настраиваемого параметра. Сортировка внутри каждой из групп элементов, отстоящих на h позиций, может осуществляться, например, методом простых вставок.

Алгоритм сортировки методом Шелла содержит следующие этапы.

╔
- внешний цикл по s $(s = 0,1,...,t-1)$ — перебор смещений:
  - $h := h_s$ — работа с очередным значением смещения;
  - вложенный цикл по j $(j = h+1,...,N)$ — для сортировки элементов, отстоящих на h позиций:
    - $i := j - h$ — номер крайнего справа от R_j элемента из уже упорядоченных в группе;
    - номера остальных последовательно уменьшаются на h;
    - $K := K_j$; $R := R_j$ — данные вставляемого элемента; он будет продвигаться влево, пока не дойдёт до элемента с меньшим значением ключа;
      - вложенный цикл по i с шагом $-h$ пока $i > 0$:
        - если $K \geq K_i$ то:
          - $R_{i+h} := R$: R ставится на место R_{i+h};
          - принудительный выход из цикла по i;
        - $R_{i+h} := R_i$ — сдвиг вправо элементов с бóльшим значением ключа с целью высвобождения места вставляемому элементу.

╚

4.4. Обменные сортировки

Семейство обменных алгоритмов сортировки предполагает систематический обмен местами для пар элементов, нарушающих требуемую упорядоченность.

4.4.1. Метод пузырька

Этот метод имеет много сходства с алгоритмом поиска максимального значения в совокупности данных, в данном случае — максимального значения сортирующего ключа. Пусть требуется отсортировать элементы $R_1,...,R_N$ с ключами $K_1,...,K_N$.

Если попарно сравнивать, начиная с $i = 1$, ключи K_i и K_{i+1}, и менять местами R_i и R_{i+1} всякий раз, когда $K_i > K_{i+1}$, то после прохода по всему списку последнюю позицию R_N станет занимать элемент с максимальным значением ключа, а остальные элементы сохранят расположение относительно друг друга («самый большой пузырёк первым всплыл наверх»).

Затем такая же процедура реализуется для списка $R_1,...,R_{N-1}$, после чего уже два элемента с наибольшими значениями ключа займут позиции R_{N-1}, R_N, и т. д.

Алгоритм сортировки методом пузырька содержит следующие этапы.

╔
- внешний цикл по j $(j = 0,..., N-2)$:
  - внутренний цикл по i $(i = 2,..., N-j)$:
    - если $K_{i-1} > K_i$, то
      - $R := R_{i-1}$; $R_{i-1} := R_i$; $R_i := R$.

╚

Элементы меняются местами в пределах исходной области памяти. Число операций обмена равно количеству инверсий в исходной перестановке. Анализ показывает, что метод пузырька требует в среднем в два раза бóльших затрат времени, чем метод простых вставок ([9], с. 128–130).

Метод пузырька допускает несколько усовершенствований:

1. Если на некотором проходе не осуществлено ни одной перестановки элементов, то массив уже полностью отсортирован, и процесс можно прекращать. Для слежения за этим достаточно завести логическую переменную — 0/1-признак несостоявшегося/состоявшегося обмена.

2. Можно определять наибольший номер, для которого во время прохода выполнялись перестановки. На следующем проходе можно заканчивать сравнения на этом номере, поскольку правая часть массива после элемента с этим номером уже упорядочена.

3. Ключи с большими значениями быстро поднимаются наверх, в то время как ключи с малыми значениями опускаются в начало списка медленно. Поэтому целесообразно менять направление каждого следующего прохода («встряхивать»). Такая сортировка называется *шейкерной*.

4.4.2. Быстрая сортировка

Метод быстрой сортировки связан со стратегией, при которой для определения пары следующих сравниваемых ключей используются результаты всех предыдущих сравнений. Он основан на итерируемом (повторяемом) разбиении текущего списка (части исходного списка) на две части, левую и правую, так, что любое значение ключа левой части меньше любого значения ключа правой части.

Если в упорядоченном по возрастанию массиве элемент R_1 должен занимать позицию s, то левее s не должно в итоге оказаться элементов с бóльшими значениями ключа, а правее s — с меньшими; это достигается перекомпоновкой остальных элементов. После этого последовательность $R_1,..., R_N$ разбивается на две подпоследовательности $R_1,..., R_{s-1}$ и $R_{s+1},..., R_N$ меньшей длины, которые сортируются дальше тем же способом уже независимо друг от друга.

Введём указатели i и j с начальными значениями $i = 2$, $j = N$. Если $K_i < K_1$, то элемент R_i должен принадлежать левому подмассиву; тогда он остаётся на месте, и указатель i увеличивается на единицу — до тех пор, пока не встретится элемент R_i, принадлежащий правому подмассиву. Та-

ким же образом будем последовательно уменьшать значение указателя j, пока не встретится элемент R_j, который должен принадлежать левому подмассиву. Если окажется, что $i < j$, то по отношению к элементу R_1 они расположены неправильно и должны поменяться местами. После обмена индексы i и j продолжают сближаться, пока снова не встретится пара, подлежащая обмену местами. Встречный проход по списку с двух сторон заканчивается, когда окажется в первый раз $i \geq j$. Тогда R_1 меняется местами с R_j. Теперь $R_k \leq R_j$ при $k < j$, и $R_k \geq R_j$ при $k > j$.

Ниже приведён пример из того, как меняется исходный список вплоть до первого разделения на два подмассива заключённых в фигурные скобки.

Первоначальное расположение:

500 **080** 580 050 950 170 900 300 650 400 150 550 600 650 750 **700**;

после первого обмена:

500 080 **580** 050 950 170 900 300 650 400 **150** 550 600 650 750 700;

после второго обмена:

500 080 150 050 **950** 170 900 300 650 **400** 580 550 600 650 750 700;

после третьего обмена:

500 080 150 050 400 170 **900_300** 650 950 580 550 600 650 750 703.

переключение указателей:

500 080 150 050 400 170 **300_900** 650 950 580 550 600 650 750 703.

После разделения:

{ 080 150 050 400 170 **300**}

500

{**900** 650 950 580 550 600 650 750 703}.

Далее такая же процедура проделывается с более короткими массивами, которые расположены слева и справа от нового R_j (бывшего R_1). Один из них обрабатывается, а другой, например, более длинный, кладётся в стек. Возможно систематическое итерирование процедуры вплоть до массивов длины 2, которые сортируются очевидным образом.

Отметим, что в *стек помещаются только границы подмассива*, то есть пара номеров (l, r), а не он сам. В [9] показано, что количество элементов в стеке никогда не будет превышать $\log_2 N$. Там же приведён алгоритм описанной сортировки.

Наиболее эффективно алгоритм сортирует полностью случайные, неупорядоченные массивы с большим числом инверсий.

Элементы в процессе сортировки перераспределяются внутри исходной области памяти, занимаемой списком. Важно также, что ключи K_i и

K_j сравниваются во время прохода по данным с одним и тем же ключом K_1, который можно постоянно держать в сравнивающем регистре.

4.4.3. Обменная поразрядная сортировка

Этот метод основан на представлении значений ключа в двоичной системе. Сравнение значений ключа заменяется проверкой их отдельных двоичных разрядов: равны они 0 или 1. В остальном метод сходен с процедурой быстрой сортировки: итерируется процесс разбиения текущего списка (части исходного списка) на две части, левую и правую — так, что любое значение ключа левой части меньше любого значения ключа правой части.

Этап 1. Сначала элементы сортируются по старшему значащему двоичному разряду так, чтобы ключи, начинающиеся с 0, оказались левее ключей, начинающихся с 1. Для этого ищется крайний слева ключ K_i, начинающийся с 1 и крайний справа ключ K_j, начинающийся с 0; R_i и R_j меняются местами. После этого индексы проверяемых элементов меняются навстречу другу ($i := i+1;\ j := j-1$) пока не окажется $i > j$.

Этап 2. Обозначим через F_0 множество элементов, двоично записанный ключ которых начинается с 0, и через F_1 множество элементов, двоично записанный ключ которых начинается с 1. Теперь поразрядная сортировка применяется к F_0 и к F_1 начиная со второго значащего разряда, и т. д.

Если значения ключа распределены в диапазоне своего изменения приблизительно равномерно, то при больших N число поразрядных сравнений оказывается в среднем меньше числа сравнений ключа при быстрой сортировке. Эффективность обменной поразрядной сортировки снижается при наличии одинаковых значений ключей.

4.5. Сортировки посредством выбора

Важное семейство процедур сортировки связано с многократным выбором элементов сортируемого массива по тому или иному принципу и установкой их на требуемое место. Эти процедуры ориентированы на минимизацию числа обменов.

4.5.1. Сортировка с использованием бесконечно большого ключа

Обозначим символом ∞ значение, которое заведомо больше всех реальных значений сортирующего ключа. Процесс сортировки посредством выбора сводится к итерированию следующей процедуры:

а) Найти элемент с наименьшим значением ключа.

(б) Переслать его в область памяти, выделенную для отсортированного списка (*область вывода*), поместив на крайнее слева
ещё не занятое место.

(в) Заменить это значение ключа на ∞.

На второй итерации будет выбрано значение ключа, наименьшее из оставшихся, поскольку значение, бывшее до этого минимальным, заменилось на ∞.

В отличие от метода вставок, в распоряжении алгоритма должен иметься сразу весь сортируемый список целиком. Метод исключает перестановки элементов местами, но требует удвоения памяти и многократных проходов с $N-1$ сравнениями при поиске минимального ключа.

Разумеется, вместо поиска минимального элемента можно всякий раз искать максимальный. Тогда отсортированный список будет заполняться в порядке убывания ключа.

4.5.2. Сортировка посредством простого выбора

По аналогии с методом пузырька, можно последовательно уменьшать число сравнений во время очерёдного прохода при поиске минимального ключа без использования бесконечно большого значения ∞ (либо при поиске максимального без использования $-\infty$).

Именно, при работе, например, с максимальными значениями ключа можно выбранное в результате очередного прохода значение R_l сразу ставить в списке на соответствующую позицию (сначала на крайнюю справа, затем на вторую справа и т.д.), а находившийся на этой позиции элемент перенести на место выбранного. Тогда элементы $R_1,...,R_N$ перераспределяются в пределах исходной области памяти.

Алгоритм сортировки посредством простого выбора содержит следующие этапы.

╔

- Внешний цикл по j $(j = N, N-1,...,2)$

  - вложенный цикл по i $(i = 2,..., j)$ для поиска

    $K_l = \max(K_1,...,K_j)$

  - обмен местами R_l с R_j.

╚

По завершении j-й итерации внешнего цикла элементы $R_j,...,R_N$ занимают окончательные позиции и далее в сравнениях не участвуют.

Возможности оптимизации процедур сортировки, основанных на поиске максимумов (или минимумов), ограничены:

Теорема. *В любом алгоритме поиска максимального среди* n *элементов необходимо выполнить не менее* $n-1$ *сравнений*.

Доказательство легко проводится по индукции.

4.5.3. Сортировка методом выбора из дерева

Аналогом этого метода являются спортивные соревнования по олимпийской системе, то есть с выбыванием проигравших.

Пусть количество сортируемых элементов является степенью двойки: $N = 2^t$. Построение дерева сортировки начинается снизу, с листьев, которые содержат все элементы массива. Выбираются максимальные ключи в 2^{t-1} непересекающихся парах соседних элементов

$$(R_1, R_2), (R_3, R_4), ..., (R_{2^t-1}, R_{2^t}).$$

Обозначим их через $R_1^{(1)}, R_2^{(1)}, ..., R_{2^{t-1}}^{(1)}$. Они становятся корнями соответствующих пар. Эти «победители» снова разбиваются на пары,

$$(R_1^{(1)}, R_2^{(1)}), (R_3^{(1)}, R_4^{(1)}), ..., (R_{2^{t-1}-1}^{(1)}, R_{2^{t-1}}^{(1)}),$$

из которых выбираются элементы с максимальным ключом $R_1^{(2)}, R_2^{(2)}, ..., R_{2^{t-2}}^{(2)}$ и делаются корнями для соответствующих пар, и т. д. Первый этап завершается, когда на t-м шаге из двух оставшихся претендентов $(R_1^{(t-1)}, R_2^{(t-1)})$ выбирается элемент R_j с абсолютно максимальным значением ключа. Индекс j запоминается в качестве первого элемента упорядоченного списка (рис. 24).

После этого лист K_j заменяется на $-\infty$, так что теперь повторение всей процедуры даст элемент со вторым по величине значением ключа (рис. 25), и т. д.

Таким образом, помимо памяти, отведённой для N элементов сортируемого массива, необходима память для размещения бинарного дерева из N узлов, а также память для N элементов отсортированного массива. Число сравнений при реализации метода равно $N \log_2 N$.

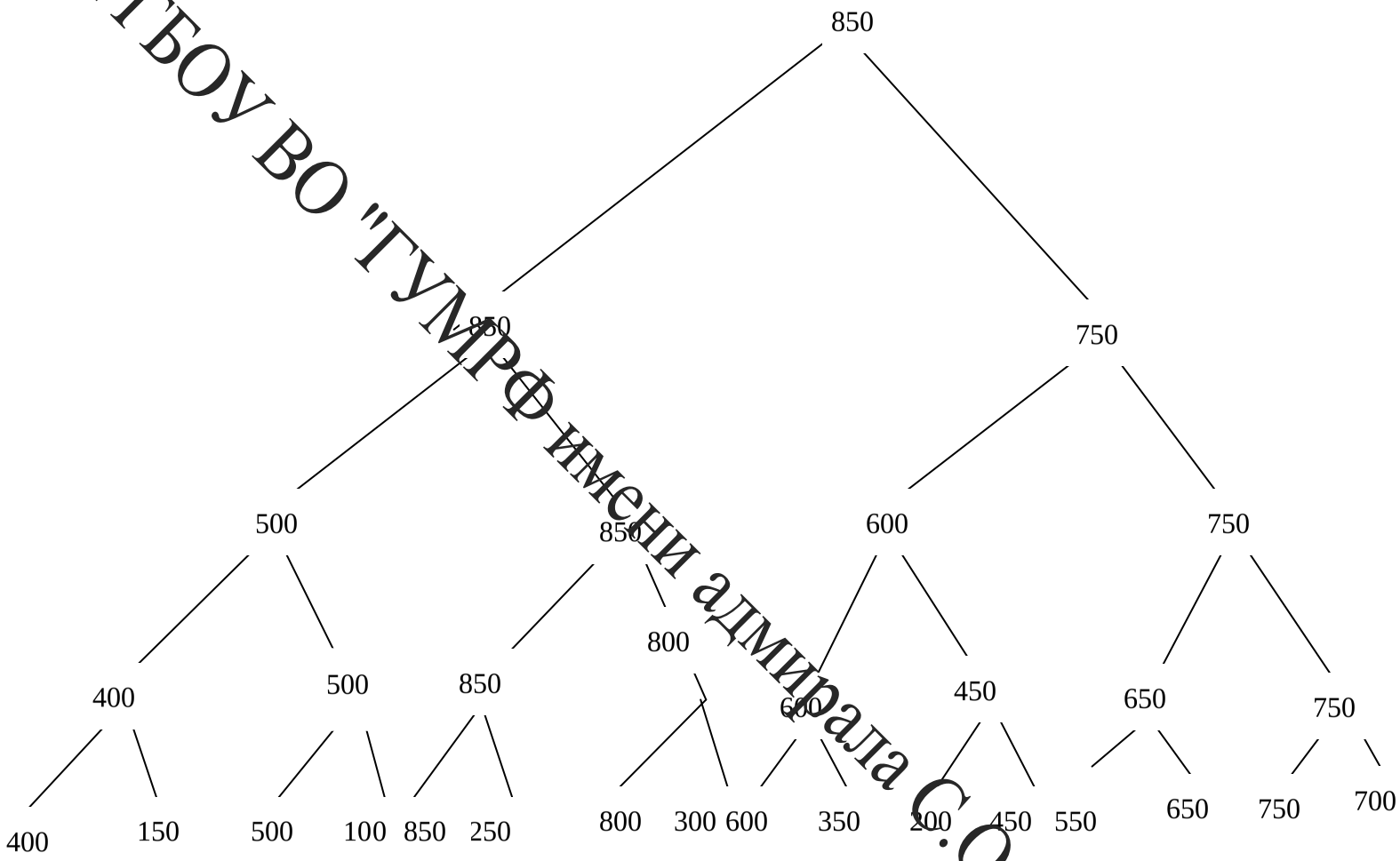


Рис. 24.

Рис. 25.

4.5.4. Пирамидальная сортировка

Полное бинарное дерево обычно хранится в последовательных ячейках памяти; при этом узлы уровня s в порядке слева направо располагаются за узлами уровня $s-1$ (рис. 26). Родитель узла с номером i имеет номер $[i/2]$ (целая часть числа $i/2$), а его непосредственные потомки — номера $2i$ и $2i+1$.

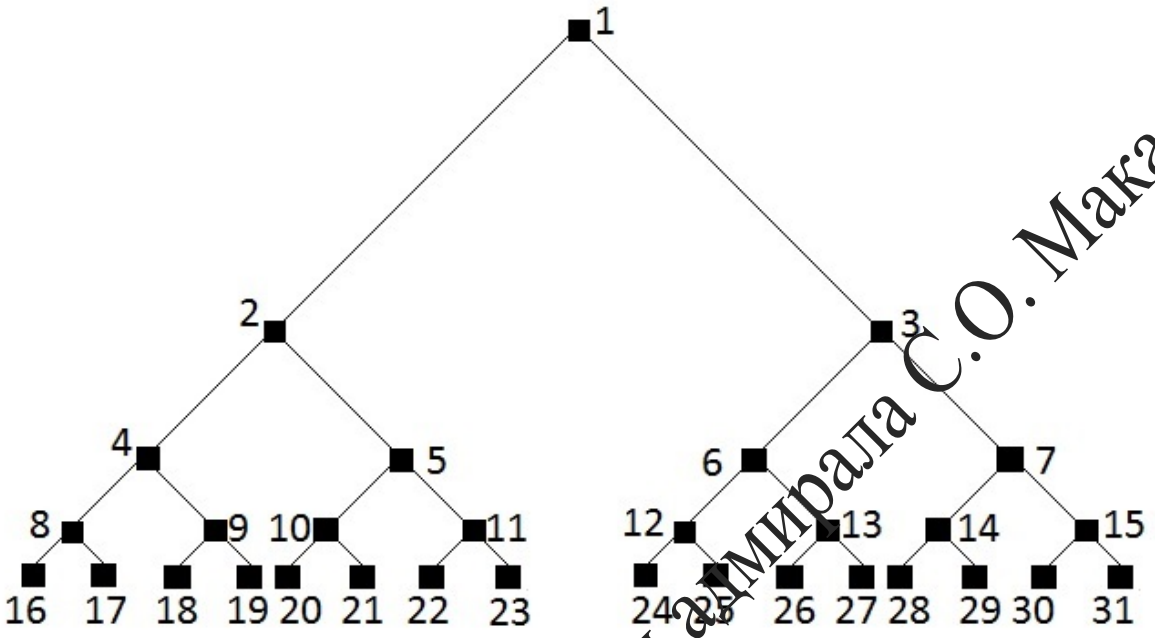


Рис. 26.

Определение. Массив ключей $K_1, K_2, ..., K_N$, представленный бинарным деревом, называется *пирамидой*, если

$$K_i \geq K_{2i}, \quad K_i \geq K_{2i+1}, \qquad (*)$$

то есть ключ в каждом узле не меньше обоих своих непосредственных потомков. Заметим, что K_i соотносится с K_{2i} и K_{2i+1} так же, как связаны родством в полном бинарном дереве соответствующие узлы (K_{2i} и K_{2i+1} — левый и правый потомки узла K_i). Поскольку число элементов массива может не совпадать с числом узлов полного бинарного дерева, то пирамиду удобно представлять как бинарное дерево со следующими свойствами:

1) все листы имеют уровень k или $k-1$;
2) если удалить листы уровня k получится полное бинарное дерево;
3) листы уровня k заполняют левый сплошной отрезок (рис. 27).

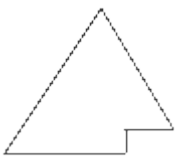


Рис. 27.

Поскольку неравенства (*) распространяются дальше по потомкам, то в пирамиде ключ в каждом узле не меньше всех своих потомков (хотя, возможно, меньше соседних ключей того же уровня и их потомков).

В частности, для пирамиды имеем:

$K_1 \ge K_2$ и $K_1 \ge K_3$;

$K_2 \ge K_4$ и $K_2 \ge K_5$; $K_3 \ge K_6$ и $K_3 \ge K_7$;

$K_4 \ge K_8$ и $K_4 \ge K_9$; $K_5 \ge K_{10}$ и $K_5 \ge K_{11}$, и т. д.

Из определения пирамиды следует, что на её вершине находится максимальный ключ: $K_1 = \max(K_1, K_2, ..., K_N)$.

Пирамидальная сортировка массива проводится в три этапа:

1. Массив $K_1, K_2, ..., K_N$ преобразуется в пирамиду путём перестановки некоторых элементов.

2. Вершина пирамиды перемещается в правый край массива, меняясь местами с находившимся там элементом.

3. Осуществляется описываемая ниже процедура «протаскивания» (или «просеивания») нового значения K_1 через дерево.

В результате протаскивания в вершине пирамиды (то есть новым ключом K_1) опять оказывается элемент, максимальный из ещё не отсортированных. Он удаляется из пирамиды, занимая очередное (справа налево) место в конце массива путём обмена местами на i-й итерации с K_{N-i+1}. При этом не отсортированными (и, значит, ещё подлежащими протаскиванию) остаются элементы $K_1, ..., K_{N-i}$. Этапы 2) и 3) циклически повторяются до исчерпания неотсортированной части массива.

Построение пирамиды. 1) Первый элемент исходного массива K_1 ставится в корень дерева; элементы K_2 и K_3 — его дети. Для того чтобы подмассив K_1, K_2, K_3 стал пирамидой, сравниваем K_1 с $K = \max(K_2, K_3)$. Если оказывается $K_1 < K$, то K_1 меняется местами с соответствующим из двух детей. Теперь K_1, K_2, K_3 — пирамида.

2) Теперь K_2 рассматривается как вершина дерева с детьми K_4, K_5, и оно также превращается в пирамиду сравнением K_2 с $K = \max(K_4, K_5)$ и, возможно, обменом местами K_2 с соответствующим из двух детей. Далее K_2 обменивается местами с K_1, если $K_1 < K_2$. Теперь пирамиду образуют K_1, K_2, K_4, K_5.

3) Далее в пирамиду превращается дерево с вершиной K_3 и детьми K_6, K_7, после чего K_3, возможно, обменивается местами с K_1. Теперь пирамиду образуют $K_1, K_2, K_3, K_4, K_5, K_6, K_7$; и т. д.

Наконец, в пирамиду превращается дерево с корнем — последним нелистовым узлом и потомками K_{N-1}, K_N либо одним потомком K_N, после чего, как и для других узлов, «идут наверх» сравнения и обмены местами с родителем вплоть до K_1.

Теперь весь массив $K_1, K_2, ..., K_N$ образует пирамиду. На её вершине находится максимальный элемент K_1. Правая граница неотсортированной части массива $r = N$. В ходе процесса перестройки массива в пирамиду он оставался в исходной области памяти.

Перемещение вершины. После того, как пирамида построена, вершина K_1 (максимальный из ещё неотсортированных элементов, в первый раз — максимальный из всех) удаляется из неё, меняясь местами с крайним справа элементом ещё неотсортированной части массива K_r (в первый раз — с K_N). Номер правой границы неотсортированной части уменьшается на единицу (в первый раз — r становится равным $N-1$).

Протаскивание элементов через пирамиду. После перемещения вершины в отсортированную часть дерево перестаёт быть пирамидой, поскольку ключ в её вершине перестаёт быть максимальным (таковым теперь является бóльший из двух элементов первого уровня). Для восстановления пирамиды вершина $K = K_1$ «протаскивается» через дерево. Это означает, что, двигаясь по дереву вниз, K меняется местами с бóльшим из двух своих детей до тех пор, пока оба они не окажутся меньше K. В результате в вершине снова оказывается максимальный элемент неотсортированной части массива.

Зацикливание этапов перемещения вершины и протаскивания вплоть до исчерпания пирамиды даёт **алгоритм пирамидальной сортировки**:

Ш.1. Начальная установка: $l := [N/2]+1$; $r := N$.

Ш.2. Преобразование массива в пирамиду при $l > 1$:

если $l > 1$, то

$l := l + 1$;

$R := R_l$;

$K := K_l$;

иначе ($l = 1$, ключи $K_1, ..., K_r$ выстроены в пирамиду)

$R := R_r$;

$K := K_r$;

$r := r - 1$;

если $r = 1$, то

$R_1 := R$;

завершить алгоритм.

Ш.3. Начало протаскивания (в данный момент $K_{[k/2]} \geq K$ при $l < [k/2] < k < r$; R_k при $r < k < N$ помещена на нужное место):

$j := l$.

Ш.4. Продвижение вниз:

$i := j$;

$j := 2j$;
если $j < r$, то
перейти к **Ш.5**;
если $j = r$, то
перейти к **Ш.6**;
если $j > r$, то
перейти к **Ш.8**.

Ш.5. Отыскание большего потомка:
Если $K < K_j$, то
$j := j + 1$.

Ш.6. Если $K \geq K_j$ перейти к **Ш.8**.

Ш.7. Подъём большего потомка наверх:
$R_i := R_j$;
перейти к **Ш.4**.

Ш.8. Занесение элемента R на нужное место в пирамиде (завершение её протаскивания):
$R_i := R$;
перейти к **Ш.2**.

Ш

В процессе сортировки элементы $R_1, R_2, ..., R_N$ меняются местами в пределах исходной области памяти.

Пример. 1. Перестройка массива в пирамиду. Пусть исходный массив имеет вид

K_1	K_2	K_3	K_4	K_5	K_6
21	82	16	15	30	90

Это соответствует бинарному дереву представленному на рис. 28.

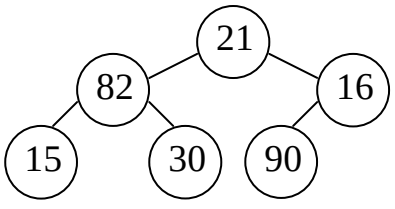


Рис. 28

Последовательные состояния дерева в процессе перестройки в пирамиду приведены на рис. 29.

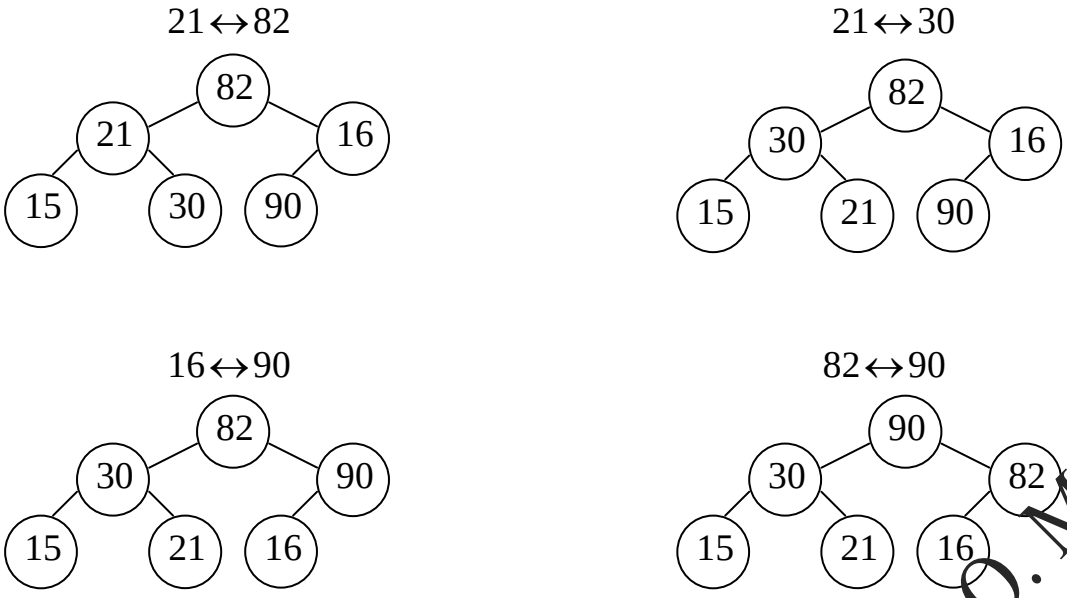
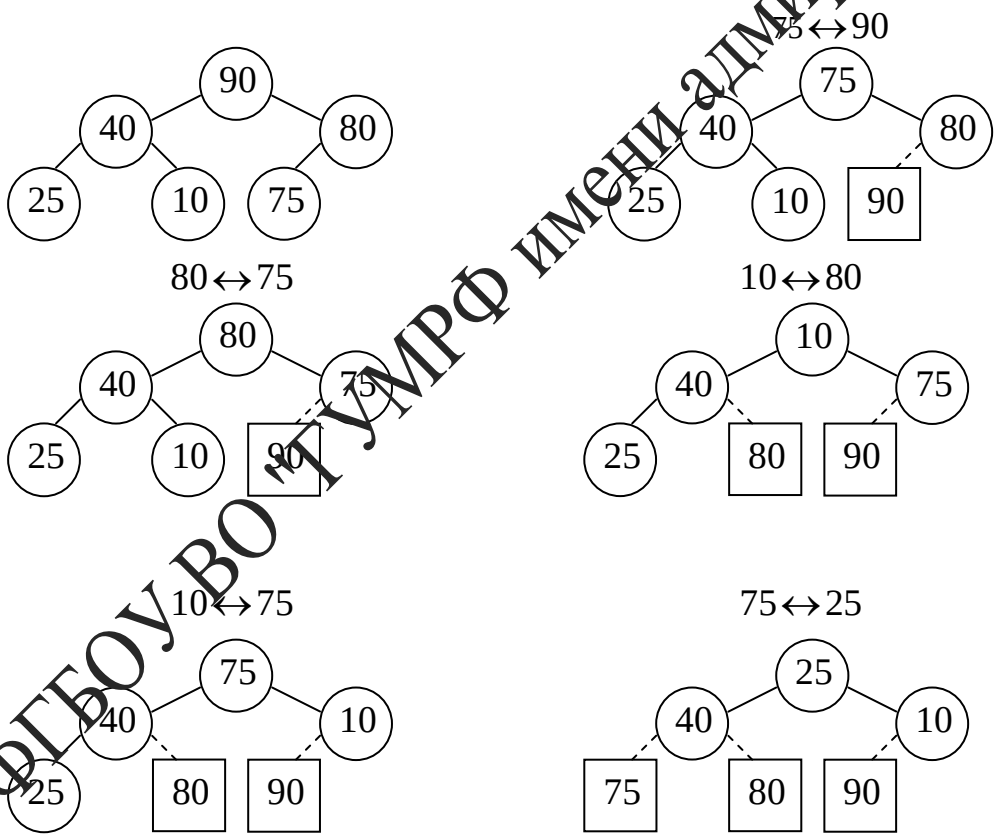


Рис. 29

2. Сортировка массива, построенного в пирамиду. Последовательные состояния дерева:



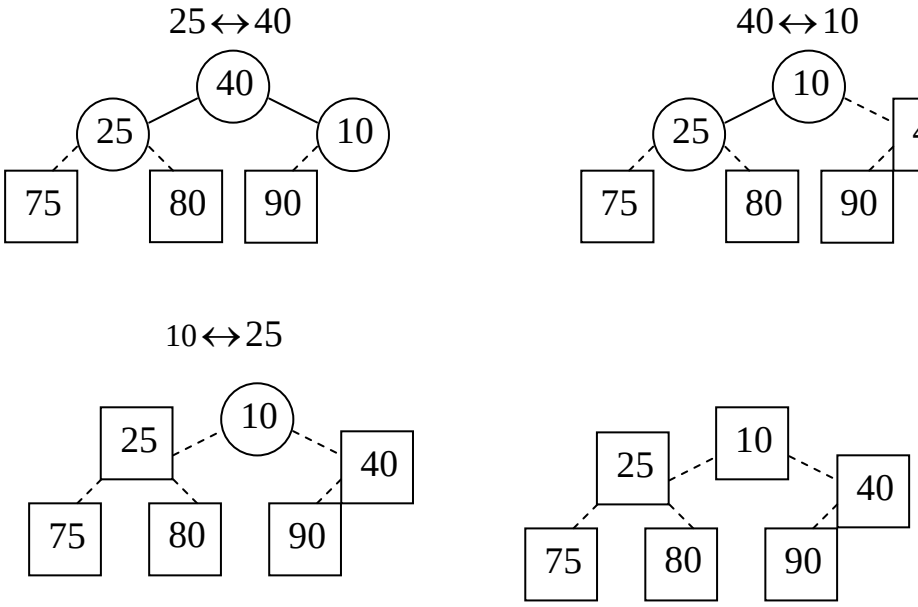


Рис. 30.

Эффективность сортировки по времени определяется величиной $O(N\log_2 N)$. Эту процедуру не следует применять при небольших значениях размера массива N, но она эффективна при больших N и становится сравнима с сортировкой Шелла. Недостатком метода следует считать то обстоятельство, что время работы алгоритма не зависит от числа инверсий в исходном массиве: изначально «более или менее упорядоченные», сортируются так же долго, как и «хаотические».

4.5.5. Сортировка методом слияния

Метод предназначен для формирования упорядоченного массива при объединении (*слиянии*) двух уже упорядоченных массивов. Подобные задачи возникают, например, в ситуациях, когда с внешних (медленных) носителей в оперативную память скачиваются части больших массивов, подлежащих объединению и сортировке.

Пусть слиянию в массив $z_1,,...,z_{m+n}$ подлежат массивы $x_1,,...,x_m$ и $y_1,,...,y_n$. Это делается с помощью следующей процедуры. Сравниваются первые (и, значит, наименьшие) элементы, и меньший из них направляется в массив z. Так продолжается до исчерпания одного из массивов. После этого оставшаяся часть другого массива отправляется в конец z.

Алгоритм сортировки методом слияния содержит следующие этапы.

╔

Ш.1. $i:=1;\ j:=1,\ k:=1$ — начальные установки индексов i, j, k для массивов x, y, z соответственно.

Ш.2. цикл по k $(k=1,...,m+n)$ — получение очередного z_k;
если $x_i < y_j$, то

$z_k := x_i$;

$i := i+1$;

если $i > m$, то выйти из цикла;

если $x_i \geq y_j$, то

$z_k := y_j$;

$j := j+1$;

если $j > n$, то выйти из цикла.

Ш.3. если исчерпан x, то дописать в z оставшиеся $y_j, ..., y_n$;

если исчерпан y, то дописать в z оставшиеся $x_i, ..., x_m$.

Ц

Сортировка многопутевым слиянием следующим образом обобщает описанную процедуру. В оперативной памяти выделяются p областей, в которых размещаются p подлежащих слиянию уже упорядоченных массивов:

$$\{K_1^{(1)}, K_2^{(1)}, ..., K_N^{(1)}\}, \ldots, \{K_1^{(p)}, K_2^{(p)}, ..., K_N^{(p)}\}.$$

Сначала выбирается минимальный из первых элементов: $K_1 = K_1^{(j)} = \min(K_1^{(1)}, K_1^{(2)}, ..., K_1^{(p)})$, записывается первым элементом упорядоченного массива и удаляется из j-го массива; j-й массив теперь начинается с $K_2^{(j)}$. Затем снова выбирается минимальный из p начальных элементов сливаемых массивов и т. д. Программная реализация предполагает использование p указателей на текущие начальные элементы соответствующих массивов.

4.5.6. Сортировка методом распределения

Этот метод эффективен при сортировке массива ключей $K_1, K_2, ..., K_N$ с лексикографическим упорядочением и для больших значений N. Например, колода из 36 карт может быть упорядочена по мастям: П<Т<Б<Ч, а внутри одной масти — по силе карт: 6<7<8<9<10<в<д<к<т.

Сначала опишем «сортировку в лоб», требующую много дополнительной памяти для промежуточных результатов сортировки, а затем укажем способ её оптимизации.

Пусть лексикографический ключ имеет k позиций (разрядов), и в i-м разряде может находиться s_i различных символов.

Начнём с примера сортировки карточной колоды из тридцати шести листов. Здесь $k = 2$ (масть карты и её значение), s_1=4 (четыре масти); s_2=9. Сначала карты раскладываются на 9 стопок вверх картинкой: в первой стопке шестёрки, во второй семёрки и т.д. Затем стопки кладутся одна на другую: снизу шестёрки, сразу на них семёрки и т. д. Далее карты пооче-

рёдно, начиная с нижней, распределяются по четырём стопкам в зависимости от масти. Наконец стопки складываются в колоду в порядке П-Т-Б-Ч.

Если при последнем распределении две карты оказались в разных стопках, то у них разные масти, и сбор их в колоду сохраняет порядок П-Т-Б-Ч, а значит, и общий порядок. Если же у них одинаковая масть, то поскольку они уже были упорядочены по картинке, то попали в свою стопку в нужном порядке: первой в стопку определённой масти попала шестёрка, затем семёрка и т. д.

Далее рассмотрим случай, когда число элементов $N \le M^k$, и ключом служит k-разрядное число, записанное в системе с основанием M. В этом случае $s_i = M = |\{0,1,...,M-1\}| = const$.

Сначала выполняется распределяющая сортировка по последнему, младшему разряду: массив $K_1, K_2, ..., K_N$ распределяется в новой области памяти по M группам $S_0, S_1, ..., S_{M-1}$, содержащим последовательно возрастающие M-ичные цифры младшего, k-го разряда: в группу S_i попадают все ключи, у которых в младшем разряде стоит i.

В худшем случае может оказаться, что в каком-либо разряде все цифры одинаковы, и в группу полностью войдёт весь массив. Таким образом, под каждую группу нужно отвести такую же память, что и для исходного массива. После этого на место исходного массива записываются последовательно сначала элементы из S_0, затем из S_1 и т. д.; последние места займут элементы с наибольшим значением $m-1$ младшего разряда.

Затем элементы раскладываются на M групп с сохранением их относительного порядка по возрастанию цифр уже предпоследнего разряда, и т. д. Распределения по группам осуществляются вплоть до начального, старшего разряда. После соединения групп в массив он оказывается полностью отсортированным.

«Сортировка в лоб» требует дополнительную память для хранения kN записей.

Радикального уменьшения необходимой дополнительной памяти можно добиться путём однократного предварительного прохода по массиву перед каждым распределением, во время которого вычисляются значения M счётчиков $\lambda_0, ..., \lambda_{M-1}$: λ_i равно количеству элементов, у которых в i-м разряде стоит i. Понятно, что

$$\lambda_0 + ... + \lambda_{M-1} = N .$$

Далее резервируется массив для N элементов. При следующем проходе ключи со значением 0 занимают места с номерами от 1 до λ_0, ключи со значением 1— места с $\lambda_0 + 1$ по $\lambda_0 + \lambda_1$, и т. д.; ключи со значением $M-1$ займут последние λ_{M-1} мест — с $(\lambda_0 + ..., +\lambda_{M-2} + 1)$-го по N-е.

На следующей итерации из двух проходов, подсчитывающего и распределительного, в качестве резервной используется область памяти с исходным массивом, который больше не нужен, и т. д.

В итоге удваивается число проходов, но объём дополнительной памяти составляет N ячеек, и ещё требуется M ячеек для счётчиков.

Сортировка связного списка. Избежать дополнительных просмотров и организации счётчиков можно при замене последовательного размещения массива ключей на хранение как исходного массива, так и промежуточного частично отсортированного массива в виде цепных списков. Для этого дополнительно заводятся M изначально пустых списков, которые заполняются по мере продвижения по списку. Платой за отказ от поразрядных счётчиков является дополнительное поле однонаправленной ссылки на следующий элемент списка. После прохода по массиву эти списки с сохранением порядка сцепляются в общий список.

Такой подход представляется особенно эффективным, когда число возможных символов в разных разрядах лексикографического ключа различно (как это имеет место, например, для колоды карт — четыре масти, девять разных значений силы карты). Список для следующего символа просматриваемого разряда организуется при попадании на символ, отличный от предыдущих.

Эффективность сортировки снижается при большом количестве разрядов ключа, в которых мало одинаковых значений.

4.6. Аппаратная сортировка (сортирующие сети)

К *сортирующим сетям* (или *сетям сортировки*) относят класс методов упорядочения, реализация которых опирается на специальные технические устройства и не зависит от предыстории сравнений: план сравнений изначально фиксирован и не меняется в процессе сортировки. Если произведено сравнение ключей K_i и K_j, то дальнейшая процедура упорядочения стандартна для $K' = \min(K_i, K_j)$ и $K'' = \max(K_i, K_j)$.

Сортирующая сеть составляется из *компараторов* («сравнивателей») — устройств, имеющих два входа для пары сравниваемых чисел и два выхода. Левый выход предназначен для большего значения K', а правый для меньшего K'' (рис. 31). Вместо термина *компаратор* употребляют также термин *модуль компаратора*, когда обсуждается не его техническое исполнение, а логическая схема алгоритма сортировки. Сети сортировки допускают эффективную аппаратную реализацию.

Рассмотрим принцип действия сортирующей сети на примере сортировки четырёх ключей K_1, K_2, K_3, K_4 с помощью сети из пяти компараторов, соединённых в соответствии с рис. 32. Компаратор (а) сравнивает K_1 и K_2. На его верхнем выходе оказывается ключ $K_1' = \min(K_1, K_2) = 10$, а на правом — ключ $K_2' = \max(K_1, K_2) = 48$.

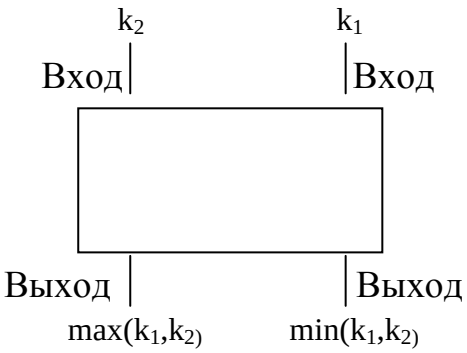


Рис. 31.

Следующий компаратор (б) аналогичным образом имеет на входе ключи K_3 и K_4, а на его выходах в таком же порядке оказываются $K_3' = \min(K_3, K_4) = 22$ и $K_4' = \max(K_3, K_4) = 31$.

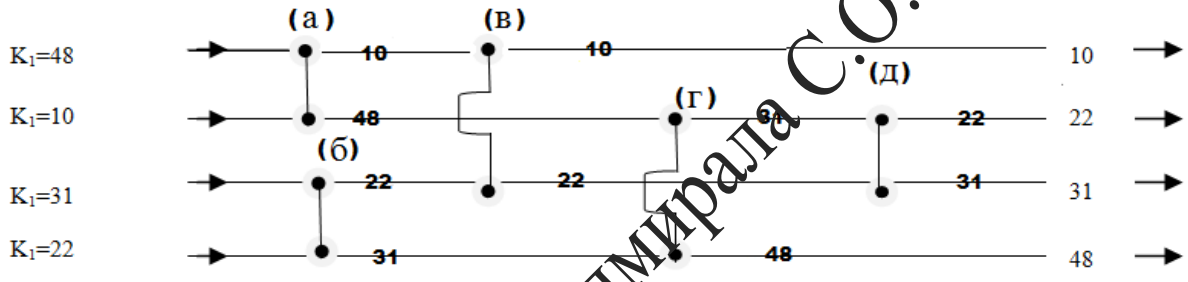


Рис. 32.

Далее на входы компаратора (в) подаются K_1' и K_3', а на выходах оказываются: на верхнем $K_1'' = \min(K_1', K_3')$, который является наименьшим из всех четырёх, а на правом — $K_2'' = \max(K_1', K_3')$. Аналогично, на входы компаратора (г) подаются K_2' и K_4', а на выходах оказываются: на верхнем — $K_3'' = \min(K_2', K_4')$, а на нижнем — $K_4'' = \max(K_2', K_4')$, который является наибольшим из всех четырёх.

Теперь остаётся неопределённым положение двух средних элементов относительно друг друга. Поэтому на вход компаратора (д) подаются K_2'' и K_3''; на его верхнем выходе оказывается второй по величине ключ, а на нижнем — третий. Выходы компараторов (в), (г), и (д) содержат полностью упорядоченный набор исходных ключей.

Количество компараторов называется *размером* сортирующей сети, а максимальное число компараторов, проходимое одним входным элементом — её *глубиной*.

Скорость сортировки повышается, если одновременно выполняются операции сравнения неперекрывающихся пар на одной и той же глубине. Так при сортировке четырёх элементов параллельно могут выполняться первое и второе сравнения, а также третье и четвёртое.

Для проверки того, что построенная сеть будет сортировать все возможные входные последовательности достаточно проверить правильность сортировки на всех возможных $N!$ перестановках множества $\{1, 2, ..., N\}$,

поскольку для сортировки важны не конкретные значения ключей, а соотношения больше/меньше между ними. Фактически же можно обойтись существенно меньшим числом из 2^N проверок ($2^N < N!$ при $N \geq 4$, и $N!$ растёт значительно быстрее, чем 2^N: $\lim_{N\to\infty} 2^N / N! = 0$):

Теорема (**двоичный принцип**[1]). *Если сеть с N входами правильно сортирует все последовательности из нулей и единиц, то она сортирует в таком же порядке любую последовательность из N ключей* ([9], с. 249]).

Минимально возможный размер сети для сортировки N элементов априори не известен и является предметом специального исследования для каждого конкретного значения N.

[1] Его называют также «принцип нулей и единиц»

Глава 5. ВНЕШНЯЯ СОРТИРОВКА

5.1. Многопутевое слияние

Напомним, что под *слиянием* уже отсортированных совокупностей данных (файлов) понимается сортировка их объединения.

Быстрая оперативная память компьютера имеет значительно меньший объём по сравнению с ёмкостью медленных периферийных запоминающих устройств — дисков, магнитных барабанов, магнитных лент. Скорость работы последних определяется скоростью не электронных, а значительно более медленных механических процессов. Поэтому рассмотренные выше методы внутренней сортировки данных, предполагающие большое число сравнений и обменов, оказываются непригодными.

Основной принцип сортировки информации, расположенной на внешних носителях — *сортировка отдельных частей файла в оперативной памяти, их возвращение на внешние носители и последующее слияние*.

Слияние оперирует с данными, имеющими структуру линейных списков и обрабатываемыми последовательно по схемам стека и очереди. Подлежащие слиянию отсортированные части файла будем называть *сериями*, а внешние устройства — *лентами*.

5.1.1. Двухпутевое слияние

В процессе слияния используются четыре вспомогательных *рабочих ленты*.

На первой фазе *производится распределение исходного файла* на S частей длины l: части файла длины l последовательно записываются в оперативную память, и там сортируются, после чего полученные *начальные серии* размещаются поочерёдно на ленте 1 и ленте 2 вплоть до исчерпания всего файла. Значение l зависит от объёма доступной оперативной памяти, а значение S от длины файла и от l.

На второй фазе ленты перематываются к началу, и производится попарное слияние находящихся на них серий. Полученные серии длины $2l$ размещаются поочерёдно на ленте 3 и ленте 4. Затем все ленты снова перематываются к началу, и содержимое лент 3 и 4 сливается в серии (снова удвоенной) длины $4l$ на ленты 1 и 2 (рис. 33).

Зацикливание второй фазы последовательно уменьшает количество сливаемых серий и удваивает их длину. Процесс заканчивается, когда останется одна серия, то есть полностью отсортированный файл.

Если исходное количество серий n нечетно, то можно формально считать, что на ленте 2 находится ещё одна серия нулевой длины. *При слиянии серий необязательно, чтобы они имели одинаковую длину*.

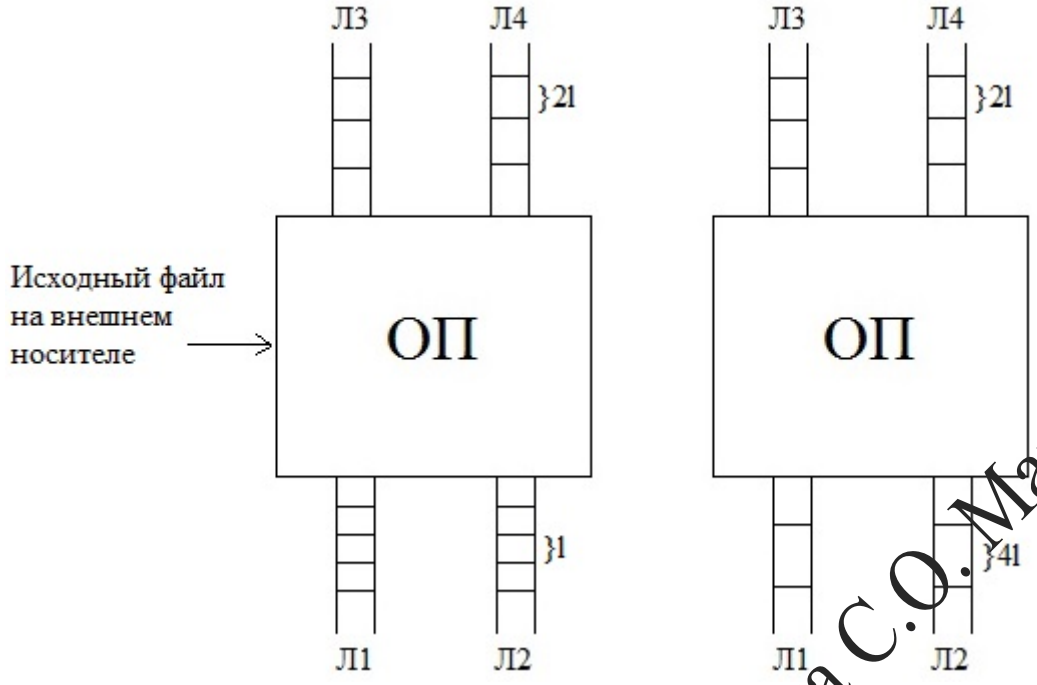


Рис. 33.

Если доступный при внутренней сортировке объём оперативной памяти вынуждает на этапе распределения разбить исходный файл на S серий, где $2^{k-1} < S \le 2^k$, то алгоритм двухпутевого слияния осуществит k проходов по всем данным (при этом $k = \lceil \log_2 S \rceil$).

Ввиду равномерного распределения серий как между лентами 1 и 2, так и между лентами 3 и 4, слияние называют *сбалансированным*. При сбалансированном слиянии после фазы распределения количество серий на разных лентах отличается друг от друга не более чем на единицу.

5.1.2. P/Q -путевое слияние

Алгоритм двухпутевого слияния, использующий четыре рабочих ленты (назовём его 2/2-путевым) следующим образом обобщается на более общий случай использования $T = P + Q$ рабочих лент при $T \ge 3$.

T рабочих лент разбиваются на два *банка* — первый («левый») банк из P лент, аналогичных лентам лент 1 и 2, и второй («правый») банк из $Q = T - P$ лент ($1 \le P < T$). На фазе распределения исходные серии размещаются поочерёдно на ленты левого банка, и после P-путевого слияния в оперативной памяти разносятся поочерёдно по Q лентам правого банка. После этого в обратном направлении сливаются Q серий из левого банка и разносятся по P лентам левого банка. Процесс продолжается циклически вплоть до получения единственной серии — всего отсортированного файла.

Значение P обычно выбирается равным $\lceil \log_2 T / 2 \rceil$.

Дальнейшие усовершенствования метода связаны с различными способами поиска минимального из начальных элементов серий (таких эле-

ментов, соответственно, P либо Q) и с соответствующими способами представления данных (например, в виде дерева).

Выгоды многопутевого слияния по сравнению с двухпутевым связаны с уменьшением числа проходов по совокупности данных: *небольшое увеличение времени работы центрального процессора с лихвой покрывается экономией времени, затрачиваемого на механические* (*существенно более медленные*) *процессы — перемотку лент и ленточные операции чтения/записи*.

Актуальность внешней сортировки снижается по мере удешевления быстрых внешних дисковых накопителей и радикального увеличения как их ёмкости, так и размеров оперативной памяти. Далее цитата из классика: «Однако большинство подобных схем сортировки настолько изящны, а соответствующие алгоритмы так отражают глубокие исследования, выполненные в ранние годы компьютеризации, что делать их достоянием только истории науки было бы слишком большим расточительством» ([9], с. 277).

Описанная сортировка называется *двухфазной*, поскольку в ней реализуются две фазы: распределение и слияние.

5.2. Многофазное слияние

Фаза рапределения файла на серии и последующего копирования отсортированных серий на ленты не связана с перестановкой элементов, то есть с увеличением степени упорядоченности. *Целью многофазного слияния является уменьшение либо даже полное устранение копирования*.

Многофазное слияние предполагает способы распределения начальных серий между лентами, отличные от сбалансированного. При многофазном слиянии уже отсортированные серии располагаются $T-1$ лентах, а слияние (без распределения) производится на единственную свободную ленту.

5.2.1. Трёхленточное слияние

Рассмотрим ситуацию использования трёх рабочих лент Л1, Л2 и Л3.

Описанный выше многопутевой P/Q-алгоритм в случае 2/1-слияния состоит из следующих шагов.

Ш1. Распределение начальных серий на Л1 и Л2.

Ш2. Слияние серий с Л1 и Л2 на Л3. Если число серий на Л3 содержит только одну серию, то закончить сортировку — на Л3 полностью отсортированный файл.

Ш3. Копирование серий с Л3 попеременно на Л1 и Л2, после чего возврат к **Ш2.**

Если количество начальных серий равно S, то после первого слияния на ленте 3 файл окажется разбитым на $\lceil S/2 \rceil$ серий, после второго слияния $\lceil S/4 \rceil$ серий и т. д. Общее число проходов при распределении на начальные серии, слияниях и копированиях равно $2\log_2 S$.

Число копирований сокращается вдвое при использовании следующего *двухфазного* алгоритма.

Ш1. Распределение начальных серий на Л1 и Л2.
Ш2. Слияние серий с Л1 и Л2 на Л3. Если Л3 содержит только одну серию, то закончить сортировку.
Ш3. Копирование половины серий, находящихся на Л3, на Л1.
Ш4. Слияние серий с Л1 и Л3 на ленту Л2. Если Л2 содержит только одну серию, то закончить сортировку.
Ш5. Копирование половины серий, находящихся на Л2, на Л1, после чего возврат к шагу **Ш2**.

Здесь **Ш2** и **Ш3** образуют первую фазу цикла, а **Ш4** и **Ш5** вторую. Уменьшение числа проходов по всем данным уменьшается при этом за счёт снижения общего объёма копирований.

5.2.2. Фибоначчиево 2/1 слияние

Рассмотрим, при каких условиях на распределение начальных серий по лентам 1 и 2 возможно полное устранение копирований.

Пусть распределение серий по лентам на какой-либо итерации имело вид

Л1	Л2	Л3
a	b	0

(предполагаем $a > b$). Тогда следующее слияние приведёт к распределению

Л1	Л2	Л3
$a-b$	0	b

Проследим последовательность слияний от конца к началу. Процесс должен завершиться, когда на n-й итерации на двух лентах (например, на Л2 и Л3) больше не окажется серий, подлежащих слиянию, а на третьей окажется единственная серия, представляющая полностью отсортированный файл:

Л1	Л2	Л3
1	0	0

Тогда на предпоследней $(n-1)$-й итерации должно быть распределение

Л1	Л2	Л3
0	1	1

На $(n-2)$-й итерации должно быть распределение

Л1	Л2	Л3
1	2	0

На $(n-3)$-й итерации должно быть распределение

Л1	Л2	Л3
3	0	2

На $(n-4)$-й итерации должно быть распределение

Л1	Л2	Л3
0	3	5

На $(n-5)$-й итерации должно быть распределение

Л1	Л2	Л3
5	8	0

На $(n-6)$-й итерации должно быть распределение

Л1	Л2	Л3
0	13	8

Всякий раз большее число серий предыдущей итерации должно быть суммой числа серий следующей итерации, а меньшее равно большему слагаемому:

$$\{a, b\} \leftarrow \{a + b, b\}. \qquad (31)$$

Обозначим число серий на лентах после n-й итерации через $F_0 = 1 - 1 = 0$ для исчерпанной ленты (с меньшим числом серий после предпоследней итерации), а через $F_1 = 1$ число серий на ленте с отсортированным итоговым файлом:

$$F_0 = 0,\ F_1 = 1.$$

В силу (31) на предпоследней $(n-1)$-й итерации на непустых лентах было:

$$F_1,\ F_2 = F_0 + F_1.$$

Таким же образом на $(n-2)$-й итерации на непустых лентах было:

$$F_2,\ F_3 = F_1 + F_2, \text{ и т. д.}$$

На первой итерации на непустых лентах было:

$$F_{n+1},\ F_{n+2} = F_n + F_{n+1}.$$

Таким образом, *количества начальных серий на лентах при* 2/1 *слиянии без промежуточных копирований должны выражаться числами Фибоначчи* F_{n+1}, F_{n+2}.

Для наглядности рассмотрим 2/1- слияние без промежуточных копирований для других, «нефибоначчиваемых» количеств начальных серий на лентах, — например, когда их не 13 и 8, а 10 и 6. Соответствующая таблица имеет вид:

Л1	Л2	Л3
10	6	0
4	0	6
0	4	2
2	2	0
0	0	2

После четвёртой итерации две серии собрались на одной ленте. Каждая серия упорядочена внутри себя, но файл в целом остался неотсортирован. Теперь для продолжения опять понадобится распределение путём копирования одной из серий, — например, на ленту 2.

Использование пустых серий. Если количество начальных серий или количество серий перед очередной итерацией не являются числами Фибоначчи, то на каждую ленту вводятся фиктивные, пустые серии. Когда i-я серия на одной из двух лент пуста, то это практически означает, что она в процессе слияния пропускается.

Пусть на двух лентах имеется S_1 и S_2 начальных серий ($S_1 > S_2$), а до ближайших превосходящих их чисел Фибоначчи F_1 и F_2 следует добавить соответственно d_1 и d_2 пустых серий. С этой целью заводятся счётчики с начальными значениями d_1 и d_2, которые уменьшаются в ситуациях, когда кроме фактических серий на слияние направляются несколько пустых. Пусть, например, число начальных серий на лентах равно 16 и 9 соответственно. Ближайшее к большему из них число Фибоначчи есть 21, а тогда предшествующее число есть 13. Поэтому начальные значения счётчиков $d_1 = 21–16=5$, $d_2 = 13–9=4$ — см. верхнюю строку таблицы; значения чисел Фибоначчи выделены жирным курсивом.

Л1	S_1	d_1	**Л2**	S_2	d_2	**Л3**	S_3	d_3
21	16	5	***13***	9	4	**0**	0	0
8	5	3	**0**	0	0	***13***	11	2
0	0	0	***8***	5	0	***5***	3	2
5	5	0	***3***	3	0	***0***	0	0

В первый раз с ленты Л1 уходят на слияние 11 фактических серий и 2 пустых; на ней остаются 5 фактических и 3 пустых, всего 8 серий. С ленты Л2 уходят 9 фактических и 4 пустых серий, и она опустошается. На Л3 оказываются 11 фактических серий; для получения числа Фибоначчи 13 добавляем 2 пустых серии. Результат отражён во второй строке.

На следующей итерации должны сливаться по 8 серий с каждой непустой ленты. Поэтому с Л1 уходят на слияние 5 фактических серий и 3 пустых, так что она опустошается. С Л3 сливаются 8 фактических серий. Результат отражён в третьей строке.

Затем должно сливаться по 5 серий с каждой ленты. С Л2 уходят на слияние в Л1 5 фактических серий, а с Л3 — 3 фактических и последние 2 фиктивные серии. Результат отражён в четвёртой строке.

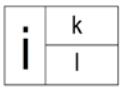
Далее сливаются только фактические серии.

5.3. Каскадное слияние

При каскадном n-ленточном слиянии на $n-1$ лентах первоначально находится в общем случае разное количество $k_1,...,k_{n-1}$ упорядоченных серий с длинами $l_1,...,l_{n-1}$.

Слияние происходит за несколько этапов. На каждом этапе последовательными слияниями (по одной серии с каждой ленты) опустошается очередная лента (с минимальным числом серий). Лента с новыми, уже более длинными сериями не участвует в дальнейших слияниях вплоть до завершения прохода по всем данным. *Длина новой серии равна сумме длин серий на участвующих лентах*. Поскольку число участвующих лент последовательно уменьшается, убывает и длина новой, «слитой» серии.[1]

Пример. Пусть в слиянии участвует 5 лент. Значком



будем обозначать, что в текущий момент на ленте i находится k серий длины l; пустую ленту обозначает значок

Ленты, не участвующие в дальнейших слияниях вплоть до завершения прохода по всем данным, помечены «*».

Исходное расположение серий на лентах:

[1] В убывании числа участвующих лент и заключается «каскадность» слияния.

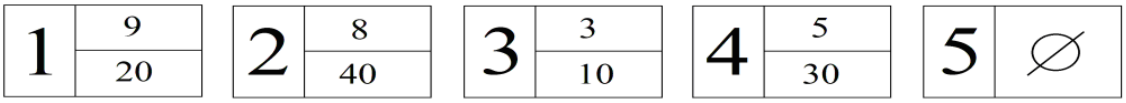


Рис. 34

Общее количество элементов на лентах, таким образом, выражается числом $9\cdot20+8\cdot40+3\cdot10+5\cdot30=680$.

Шаг 1. *Опустошение ленты* 3 *с минимальным числом серий*. После того, как ленту 5 будет слито по одной серии, на ней окажется одна серия диной $20+40+10+30=100$, а на первых четырёх лентах число серий уменьшится на единицу:

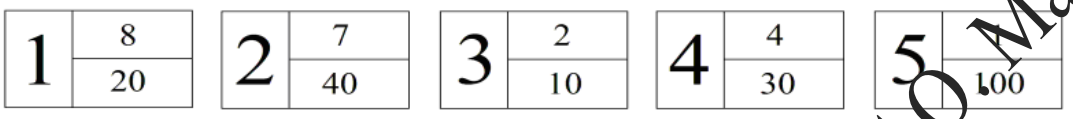


Рис. 35.

После слияния ещё одной четвёрки серий на ленту 5 получится следующее распределение серий:

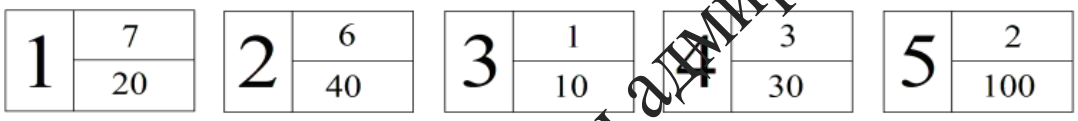


Рис. 36.

Закончится первый шаг после ещё одного слияния распределением:

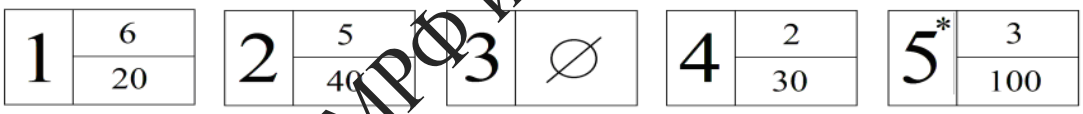


Рис. 37.

Шаг 2. *Опустошение ленты* 4. Слияние теперь производится на ленту 3. Длина возникающих на ней серий равна $20+40+30=90$. Серии ленты 5 в дальнейших слияниях не участвуют. Аналогично первому шагу придём к распределению (промежуточные слияния не отображаем):

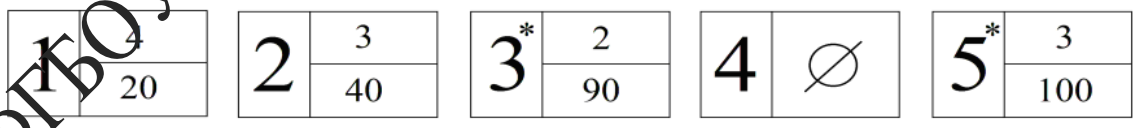


Рис. 38.

Шаг 3. *Опустошение ленты* 2. Слияние теперь производится на ленту 4. Длина возникающих на ней серий равна $20+40=60$. Серии ленты 3 и ленты 5 в дальнейших слияниях не участвуют. Аналогично предыдущему придём к распределению:

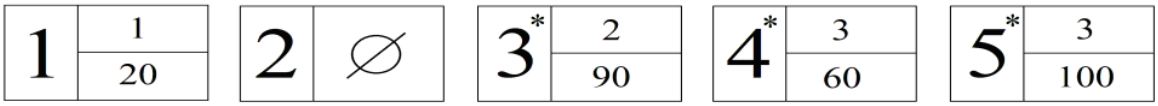


Рис. 39.

Первый этап окончен — завершён проход по всем данным. Свободна лента 2. Остальные ленты готовы ко второму этапу.

Второй этап сопровождается следующими распределениями серий (показаны ситуации опустошения очередной ленты):

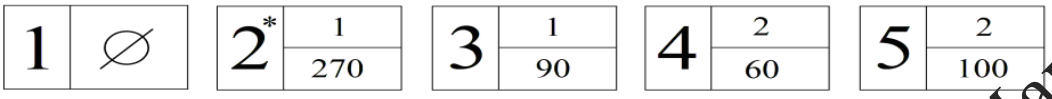


Рис. 40.

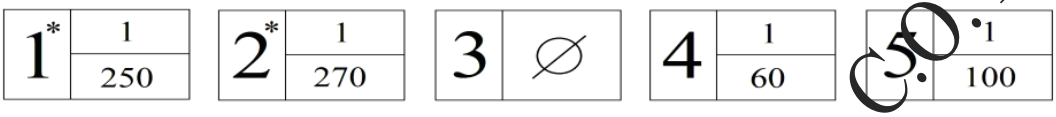


Рис. 41.

В результате на ленте 3 оказывается полностью отсортированный массив из 680 элементов.

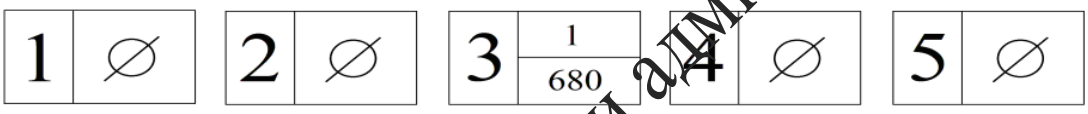


Рис. 42.

Если ставится цель, чтобы в последнем слиянии на пустую ленту участвовали все остальные ленты, по одной серии на каждой (так что они опустошаются разом), то приходим к так называемому *точному начальному распределению серий*. Отправляясь от желаемого конечного распределения (длины серий не важны) получаем перед очередным проходом по всем данным (пустая лента не показана, номера лент с указанным числом серий могут не совпадать с номером столбца в таблице — важно распределение серий):

1	**0**	**0**	**0**
1	**1**	**1**	**1**
4=1+1+1+1	**3**=1+1+1	**2**=1+1	**1**
10=4+3+2+1	**9**=4+3+2	**7**=4+3	**4**
30=10+9+7+4	**26**=10+9+7	**19**=10+9	**10**
…	…	…	…
a	*b*	*c*	*d* (*a*>*b*>*c*>*d*)
a+*b*+*c*+*d*	*a*+*b*+*c*	*a*+*b*	*a*

Последняя строка таблицы означает, что если на каком-либо «ретроспективном» шаге имеем распределение серий по лентам

$$a+b+c+d;\ a+b+c;\ a+b;\ a,$$

то после очередного прохода по всем данным придём к распределению $a,\ b,\ c,\ d$.

Глава 6. АЛГОРИТМЫ ПОИСКА

Задача поиска формулируется следующим образом. Имеется набор (массив, файл, таблица, список) элементов (записей), содержащих среди других полей и ключевое поле; требуется *найти элемент или группу элементов с заданным значением ключа*. Например, часто в качестве ключевого поля в файлах, хранящих сведения о работниках предприятия, используется табельный номер. Ключевыми полями для поиска в соответствующих файлах являются номер паспорта, автомобильный регистрационный номер, номер ИНН. Для больших файлов и их совокупностей распространён термин *базы данных*.

Как правило, записи файла имеют несколько ключей для поиска (например, ФИО, год рождения, стаж работы, и т. п.). Задача поиска может ставиться для набора значений нескольких ключевых полей — так обычно и происходит при работе с базами данных.

Как и в случае сортировки, целью поиска обычно является не сам ключ, а содержащая его (*ассоциированная с ним*) запись. Мы будем в дальнейшем для краткости говорить о поиске ключа.

По аналогии с методами сортировки, методы поиска делятся на *внутренние* (по данным, находящимся в быстрой оперативной памяти) и *внешние* (по данным, хранящимся на значительно более медленных внешних носителях). Другой аспект классификации делит методы поиска на *статические* (данные не меняются) и *динамические* (данные обновляются путём удаления и вставки новых записей).

Принципиальными для алгоритмов поиска являются следующие факторы: *упорядоченность/неупорядоченность* данных; их *структура* (способ представления); наконец *объём* области поиска.

Отметим отдельно задачу *поиска по искажённому ключу*, когда тем или иным способом формализуется количественно понятие сходства ключей и отбирается группа ключей, схожих с данным.

6.1. Последовательный поиск

Пусть имеется массив элементов $Z_1, Z_2, ..., Z_N$ с ключами $K_1, K_2, ..., K_N$. Простейшая процедура последовательного поиска перебирает ключи в соответствии с их номерами и останавливается при совпадении очередного ключа с заданным значением K. Если после просмотра всего массива ключ не найден, то выдаётся соответствующее сообщение.

Если искомый ключ в массиве отсутствует или расположен в её конце, необходимо сделать N просмотров. Среднее число сравнений ключа равно $(1+2+...+N)/N=(N+1)/2$.

Алгоритм прямого перебора:

╔

Ш1. $f := 0$ признак $1/0$ — ключ найден/не найден.

Ш2. Цикл по i $(1 \le i \le N)$:

если $K_i = K$, то

запоминание i;

$f := 1$;

выход из цикла.

Ш3. Если $f = 1$ обработка записи Z_i,

иначе сообщение об отсутствии таковой.

╚

Быстрый последовательный поиск

Оптимизация циклической программы связана, прежде всего, с уменьшением количества действий в теле цикла, в данном случае — количества сравнений.

В теле цикла вышеприведённой программы производятся два сравнения: 1) текущее значение параметра цикла i сравнивается с верхней границей N его допустимого изменения, и 2) ключ K_i сравнивается с K. Первого сравнения можно избежать: дополним массив *барьером* — значением $K_{N+1} = K$, и, увеличивая i, уже не будем сравнивать его с N. После обнаружения ключа K (теперь это гарантированно произойдёт) осуществляется выход из цикла. Далее производится анализ текущего значения i: если $i = N + 1$, то ключ не найден.

Последовательный поиск в упорядоченном массиве. Если заведомо известно, что поиск в большом массиве будет производиться многократно, то целесообразно «не поскупиться» и предварительно отсортировать её.

Пусть $K_1 < K_2 < \ldots < K_N$. Теперь при последовательном поиске от начала массива превышение очерёдным ключом K_i искомого значения K позволяет ещё до завершения полного прохода по всему массиву сделать вывод об отсутствии нужной записи.

Как и раньше, обозначим символом ∞ значение, заведомо большее возможных значений ключевого поля.

Алгоритм последовательного поиска состоит из следующих этапов:

╔

Ш1. Начальные присваивания: $i = 1$; $K_{N+1} := \infty$.

Ш2. Если $K \le K_i$, то перейти к шагу **Ш4**.

Ш3. Увеличить i;

вернуться к шагу **Ш2**.

Ш4. Если $K_i = K$, то ключ найден, в противном случае

ключ в массиве отсутствует.

Переход к шагу **Ш4** произойдёт, когда в первый раз встретится ключ, не меньший K. Тогда либо он — искомый ($K_i = K$), либо это уже лишний добавленный ключ, и нужная запись не найдена. Значение $K_{N+1} = \infty$ в добавленной фиктивной записи гарантирует выполнение условия $K \le K_i$ и выход из цикла, по крайней мере, для $i = N+1$.

Ускорение поиска возможно при наличии априорной информации о вероятностях p_i различных значений ключа поиска: $p_i = P(K_i = K)$. Тогда сортировка ключей в порядке убывания вероятностей:

$$p_1 \ge p_2 \ge \ldots \ge p_N \quad (p_1 + p_2 + \ldots + p_N = 1),$$

расположит наиболее вероятные значения в начале массива, и время поиска в среднем уменьшится.

На практике часто применяют эвристический принцип «80–20»: как правило, 80% запросов на поиск относятся к 20% содержимого. Накопленная статистика позволит поместить эти 20% записей в начало и тем уменьшить среднее время поиска. Для получения указанной статистики можно на начальном этапе обращений к массиву связать с каждой записью счётчик числа запросов на её поиск.

6.2. Бинарный поиск в упорядоченном массиве

Стратегия бинарного поиска основана на получении максимальной информации от каждого сравнения. Последнее имеет место, если любой вариант ответа на правильно поставленный при поиске вопрос вдвое уменьшает число дальнейших сравнений при наихудшем раскладе ключей. Степень 2^k быстро растёт с увеличением k, а соответствующая область поиска, оцениваемая числом $N / 2^k$, так же быстро уменьшается (см. [16]). При делении области поиска пополам количество сравнений является величиной порядка $\log_2 N$ (точнее, оно равно этому логарифму, округлённому до ближайшего целого, то есть это либо $[\log_2 N]$, либо $[\log_2 N]+1$).

Правильный вопрос, ответ на который уменьшает (упорядоченную!) область поиска вдвое, звучит так: *слева или справа от середины текущей области поиска лежит искомый ключ*?

Алгоритм бинарного поиска содержит следующие этапы.

Ш1. Начальные присваивания:

$l = 1$ — указатель левой границы области поиска;

$r = N$ — указатель правой границы области поиска.

Ш2. $s := [(l + r) / 2]$ — номер, делящий область поиска приблизительно пополам.

Ш.3. Сравнение ключей K и K_s; при этом:

если $K < K_s$, то

$r = s - 1$ — новая правая граница, так как искомый ключ лежит слева от K_s;
левая граница l сохраняется;

переход к **Ш.2**;

если $K > K_s$, то

$l = s + 1$ — новая левая граница, так как искомый ключ лежит справа от K_s;
правая граница r сохраняется;

переход к **Ш.2**;

если $K = K_s$, то поиск завершён.

NB: Целесообразен именно такой порядок сравнений K и K_s, поскольку равенство может иметь место только один раз, а в остальных случаях сразу сработает переход к суженной области поиска.

Когда текущая область поиска $\{l, r\}$ станет содержать не более, например, четырёх ключей, производится последовательный поиск.

В данном алгоритме много сходства с поиском корня функции методом деления отрезка пополам.

Если число записей N удовлетворяет неравенству $2^{k-1} \le N < 2^k$, то число необходимых сравнений в худшем случае равно k.

6.3. Поиск Фибоначчи в упорядоченном массиве

При поиске Фибоначчи с индексами массива производятся только сложения и вычитания, без применения более медленной операции деления пополам и взятия целой части.

Пусть сначала число ключей N на единицу меньше некоторого числа Фибоначчи: $N + 1 = F_{k+1}$. С ключом поиска в первый раз сравнивается ключ, имеющий номер $i = F_k$. Если от него шагом $q = F_{k-2}$ сделан переход в правую область бóльших значений, то следующий шаг перехода $q = F_{k-4}$ (пропускается шаг F_{k-3}); После перехода в левую область меньших значений следующий шаг $q = F_{k-3}$. Дальнейшие шаги определяются аналогично. Поскольку когда-нибудь шаг перехода окажется равным F_1 или F_2, то есть единице, останется проверить соседний элемент.

Ввиду рекурсивной структуры последовательности чисел Фибоначчи, всякий раз достаточно помнить только текущую тройку чисел i, q, p, где q и p — пара соседних чисел Фибоначчи. Следующая тройка, со сдвигом номеров на единицу или на два, получается из предыдущей.

Номера ключей, выбираемых для сравнения, образуют *дерево Фибоначчи порядка* k, или, для краткости, Ф-дерево, которое определяется рекурсивно следующим образом:

- если $k = 0$ или $k = 1$, то дерево является листом с номером 0;
- если $k \geq 2$, то корнем дерева является элемент с номером F_k; левое поддерево является Ф-деревом порядка $k-1$ с номером корня F_{k-1}; правое поддерево является Ф-деревом порядка $k-2$ с номером корня $F_{k-2} + F_k$.

Связь Ф-дерева с поиском: значение узла — номер очередного элемента, сравниваемого с ключом поиска K. Левый и правый непосредственные потомки его — возможный номер следующего сравниваемого элемента в зависимости от результата текущего сравнения. Значения $z_л$ и $z_п$ левого и правого непосредственных потомков внутреннего узла z отличаются от него на величину q, являющуюся числом Фибоначчи: $z_л = z - q;\ z_п = z + q$. Например, при $z = 8$:

$$z_л = z - F_4 = 8 - 3 = 5 = F_5;\ z_п = z + F_4 = 8 + 3 = 11.$$

Если разница значений узла и его непосредственных потомков на каком-либо уровне составляла F_t, то на следующем уровне для потомков на левой ветви она составит F_{t-1}, а для потомков правой ветви — F_{t-2}.

Когда шаг перехода окажется равным единице, значениями листьев, то есть номерами, к которым может привести поиск, окажутся номера всех элементов массива, а также значение 0, как признак отсутствия ключа с требуемым значением.

На рис. 43 приведено Ф-дерево порядка 6 с корнем $F_6 = 8$ для поиска в массиве из 12 элементов ($12 = F_7 - 1$).

Алгоритм поиска Фибоначчи.

Ш1. Начальные установки: $i := F_k;\ p := F_{k-1};\ q := F_{k-2}$.

Ш2. Если $K < K_i$, то
перейти к шагу **Ш3**;
если $K > K_i$, то
перейти к шагу **Ш4**;
если $K = K_i$, то ключ найден, алгоритм завершается.

Ш3. Переход влево, то есть уменьшение i:
если $q = 0$, то
ключ не найден, и алгоритм завершается;
$i := i - q$;
$a := p;\ p := q;\ q := a - q$ — изменение шага путём
сдвига чисел Фибоначчи на один номер.

.

(Три последних присваивания можно выразить присваиванием значения паре: $(p,q) := (q, p-q)$, где справа от знака := стоят прежние значения, а слева — новые).

Возврат к **Ш2**.

Ш4. Переход вправо, то есть увеличение i:

если $p = 1$, то ключ не найден, и алгоритм завершается;

$i := i + q$;

$p := p - q$; $q := q - p$ — изменение шага путём сдвига чисел Фибоначчи на два номера (в изменении q участвует уже новое значение p).

Возврат к **Ш2**.

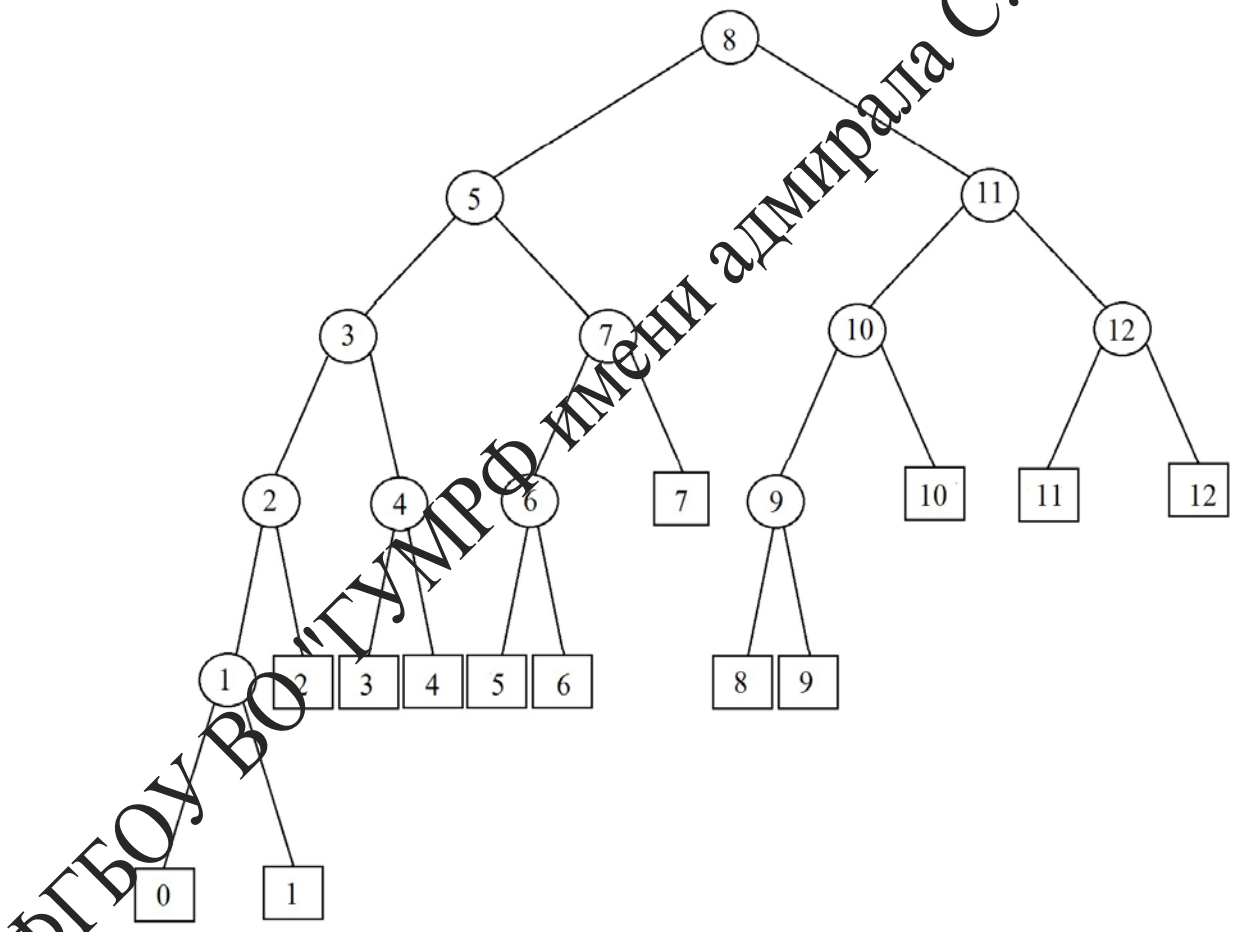


Рис. 43.

Пример. Рассмотрим, как происходит поиск в массиве

6	8	14	17	20	22	28	31	32	40	42	43	48	55	62	67
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

68	80	82	87
17	18	19	20

Здесь $N=20,\ N+1=21=F_8,\ k+1=8,\ k=7,\ F_7=13$.

Пусть ключ поиска равен 28.

1) $i=13,\ p=8,\ q=5;\ \ 28<K_{13}\Rightarrow$ к **Ш3**;

2) $i=13-5=8,\ p=5,\ q=3;\ \ 28<K_8\Rightarrow$ к **Ш3**;

3) $i=8-3=5,\ p=3;\ q=2;\ \ 28>K_5\Rightarrow$ к **Ш4**;

4) $i=5+2=7,\ p=1,\ q=2-1=1;\ \ 28=K_7\Rightarrow$ ключ найден.

Пусть ключ поиска равен 39 (в массиве отсутствует).

1) $i=13,\ p=8,\ q=5;\ \ 39<K_{13}\Rightarrow$ к **Ш3**;

2) $i=13-5=8,\ p=5,\ q=3;\ \ 39>K_8\Rightarrow$ к **Ш4**;

3) $i=8+3=11,\ p=5-3=2,\ q=3-2=1;\ 39<K_{11}\Rightarrow$ к **Ш3**;

4) $i=11-1=10,\ p=1,\ q=2-1=1;\ \ 39<K_{10}\Rightarrow$ к **Ш3**;

5) $i=10-1=9,\ p=1,\ q=1-1=0;\ \ 39>K_9\Rightarrow$ к **Ш4**.

6) Поскольку $p=1$, ключ не найден.

Если $N+1$ не является числом Фибоначчи, начальные установки модифицируются следующим образом. Находится минимальное $M\geq 0$ такое, что $N+M+1$ оказывается некоторым числом Фибоначчи: $N+M+1=F_{k+1}$. На предварительном шаге **Ш1** устанавливается $i:=F_k-M$. В начало шага **Ш2** вставляется условие "если $i\leq 0$, то перейти к шагу **Ш4**." Алгоритм циклического получения новых индексов i сдвигает поиск влево на M, и на завершающем этапе, как и раньше, приводит к концевым листьям.

Пример.

6	8	14	17	20	22	28	31	32	40	42	43	48	55	62	67
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

68	80	82	87	90	91	95	97	99	102	105	110	111	130
17	18	19	20	21	22	23	24	25	26	27	28	29	30

Здесь $N=30,\ N+1=31,\ F_{k+1}=34=F_{10},\ F_k=21,\ M=3$.

Пусть ключ поиска равен 28.

1) $i:=F_k-M=21-3=18;\ \ p=13,\ q=8;\ \ 28<K_{18}\Rightarrow$ к **Ш3**;

2) $i=18-8=10,\ p=5,\ q=3;\ \ 28<K_{10}\Rightarrow$ к **Ш3**;

3) $i = 10 - 5 = 5,\ p = 3,\ q = 2;\quad 28 > K_5 \Rightarrow$ к **Ш4**;

4) $i := 7,\ p = 1,\ q := 2 - 1 = 1;\quad 28 = K_7 \Rightarrow$ ключ найден.

6.4. Поиск по бинарному дереву

6.4.1. Бинарное дерево поиска

Если массив ключей динамически изменяется за счёт удалений и вставок, то нецелесообразно тратить время на постоянное завершение этих операций новыми сортировками. Представление совокупности ключей бинарным деревом не только упрощает удаления и вставки элементов (путём изменения цепных ссылок), но и позволяет эффективно проводить поиск и сортировку.

Напомним, что узлы бинарного дерева — записи, которые помимо поля значения *val*, указателя *llink* на левое поддерево и указателя *rlink* на правое поддерево содержат также ключевое поле *key*; если какое-нибудь из поддеревьев отсутствует, соответствующий указатель имеет значение $\varnothing$ (оба поддерева заведомо отсутствуют у листьев дерева, то есть у концевых узлов). Указатель на искомую запись бинарного дерева обозначим через p; указатель на корень дерева обозначим через t (см. п. 2.5).

Бинарное дерево является *деревом поиска*, если для каждого узла q его левый потомок имеет меньшее значение ключа, а правый — не меньшее:

$$key(llink(q)) < key(q) \le key(rlink(q)),$$

причём такое же неравенство распространяется от непосредственных потомков узла q на все узлы q' и q'' его левого и правого поддерева соответственно:

$$key(q') < key(q) \le key(q'').$$

На рис. 44 приведено бинарное дерево поиска.

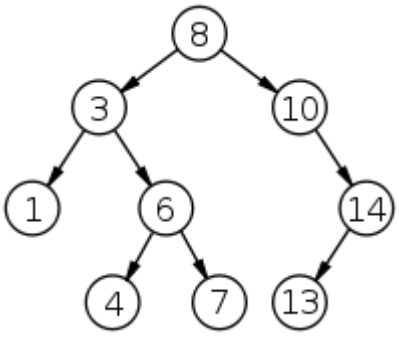


Рис. 44.

Бинарное дерево поиска обладает следующими важными свойствами:

- максимальное значение ключа находится в крайнем правом узле; переход к этому узлу из корня производится по ссылкам на правого потомка до тех пор, пока такие ссылки не пусты;

- минимальное значение ключа находится в крайнем левом узле; переход к этому узлу из корня производится по ссылкам на левого потомка до тех пор, пока такие ссылки не пусты;

- для перехода к узлу с непосредственно следующим за $key(q)$ значением ключа нужно сначала перейти к правому потомку $rlink(q)$, а далее смещаться по ссылкам на левого потомка до тех пор, пока эти ссылки не пусты;

- для перехода к узлу с непосредственно предшествующим $key(q)$ значением ключа нужно сначала перейти к левому потомку $llink(q)$, а далее смещаться по ссылкам на правого потомка до тех пор, пока эти ссылки не пусты.

6.4.2. Поиск и вставка в дереве поиска.

Рассмотрим поиск по дереву поиска указателя p на узел с заданным значением ключа $key(p) = K$ со вставкой новой записи, если ключ не найден.

Алгоритм поиска и вставки в дереве поиска может быть реализован рекурсивно: сравнить ключ поиска со значением ключа в корневом узле; если ключи совпадают, выдать указатель на найденный ключ; в противном случае искать ключ рекурсивно в левом или правом поддереве — в зависимости от выявленного неравенства.

Циклическая реализация алгоритма содержит следующие этапы:

Ш1. Начальные установки: $p := t$.

Ш2. Сравнение:

если $K < key(p)$, то

переход к шагу **Ш3**;

если $K > key(p)$, то

переход к шагу **Ш4**;

если $K = key(p)$, то

поиск завершается.

Ш3. Переход к левому поддереву:

если $llink(p) \neq \varnothing$, то

$p := llink(p)$;

переход к **Ш2**;

иначе

переход к **Ш5**.

Ш4. Переход к правому поддереву:

если $rlink(p) \neq \varnothing$, то
$p := rlink(p)$;
переход к **Ш2**;
иначе
переход к **Ш5**.

Ш5. Вставка ключа в дерево при неудачном поиске:

1) из списка свободной памяти выбирается адрес $fadr$ и заменяется там на следующий;
2) указателю q вставляемой записи с ключом K присваивается выбранный адрес, и вставляемая запись делается концевым узлом:
$q := fadr$;
$key(q) := K$;
$llink(q) := \varnothing$,
$rlink(q) := \varnothing$;
3) устанавливается связь с узлом, до этого бывшим концевым:
если было $K < key(p)$, то
$llink(p) := q$,
иначе
$rlink(p) := q$.

Ц

Недостатком данного алгоритма является теоретическая возможность превращения дерева фактически в линейный список, когда все поля связи $llink$ либо все поля связи $rlink$ оказываются пустыми.

Среднее число сравнений при поиске в дереве с N узлами имеет порядок $2\ln N$ (логарифм натуральный). Алгоритм может быть использован для сортировки ключей, и особенно эффективен при совмещении поиска и сортировки.

NB: Алгоритм вставки элемента в бинарное дерево поиска, начиная с корня для исходно пустого дерева, обеспечивает сохранение следующего условия: *Все ключи, которые больше расположенного в данном узле, попадают в ходе вставок в его правое поддерево, а ключи, которые меньше — в левое.*

Префиксная стратегия обхода дерева (см. п. 2.5) позволяет выводить ключи в порядке возрастания. Это даёт ещё один механизм сортировки — *сортировку обходом дерева поиска*.

6.4.3. Удаление из дерева поиска

Пусть при поиске узла, подлежащего удалению, определено значение q указателя на него с помощью приведённого выше алгоритма поиска и вставки. Следует различать три случая:

1) узел q не имеет ни одного потомка;

2) узел q имеет только одного потомка q';

3) узел q имеет обоих потомков.

В первом случае обнуляется ссылка на узел q его родителя, и область памяти, занятая удаляемым узлом, возвращается в пул свободной памяти. Во втором случае ссылка на q в родительском узле заменяется ссылкой на q'. При наличии у q обоих потомков следует найти непосредственно следующий за ним по порядку ключей узел s — самый левый в правом поддереве, заменить содержимое узла q (ключевое поле и поля данных) на содержимое узла s, после чего удалить узел s по схеме первого или второго случая, поскольку $llink(s) = \varnothing$.

Алгоритм удаления узла из дерева поиска содержит следующие этапы.

╔

Ш1. Проверка пустоты ссылки $rlink$ удаляемого элемента:

$t := q$;

если $rlink(t) = \varnothing$, то

$q := llink(t)$;

перейти к **Ш4**.

Ш2. Поиск преемника удаляемого элемента:

$r := rlink(t)$;

если $llink(r) = \varnothing$, то

$llink(r) := llink(t)$;

$q := r$;

перейти к **Ш4**.

Ш3. Поиск пустой ссылки $llink$:

1) $s := llink(r)$;

2) если $llink(s) \neq \varnothing$, то

$r := s$;

возврат к 1).

3) если $llink(s) = \varnothing$, то

$llink(s) := llink(t)$;

$llink(r) := rlink(s)$;

$rlink(s) := rlink(t)$;

$q := s$.

Ш4. Область, занятая удаляемым узлом t, переходит в пул свободной памяти.

╚

6.4.4. Оптимальное бинарное дерево поиска

Если накоплена статистика о частоте обращения к поиску для всех ключей, то возможна оптимизация структуры дерева, направленная на более быстрое нахождение более часто заказываемых ключей.

Пусть узлы дерева представляют имеющиеся в файле значения $K_1 \le K_2 \le ... \le K_N$. Заданы $2N+1$ вероятностей, связанных с возможными положениями ключа поиска K относительно узлов:

$$p_1,..., p_N, q_0, q_1,..., q_N,$$

где

$p_i = P(K = K_i)$ — вероятность совпадения ключа поиска с K_i;

$q_0 = P(K < K_1)$;

$q_N = P(K > K_N)$;

$q_t = P(K_t < K < K_{t+1}) \quad 1 \le t \le N-1,$

$p_1 + ... + p_N + q_0 + q_1 + ... + q_N = 1.$

Таким образом, вероятности p_i описывают ситуации удачного поиска, а q_t — неудачного. В ряде задач целью поиска может служить как раз установление факта отсутствия ключа в списке, так что следует учитывать при оптимизации равноправно оба возможных исхода.

Если внутренний узел K_j имеет уровень u_j, то для его отыскания при удачном поиске потребуется $u_j + 1$ сравнений (начиная с корня дерева, имеющего нулевой уровень, и заканчивая сравнением с K_j). Если v_j — уровень $(j+1)$- го внешнего узла, то при неудачном поиске потребуется v_j сравнений

Тогда математическое ожидание количества сравнений при поиске выражается числом

$$c = \sum_{j=1}^{N} p_j (u_j + 1) + \sum_{t=0}^{N} q_t v_t$$

и называется *ценой дерева*. Требуется найти структуру дерева, при которой цена минимальна:

$$c = c(u_1,...,u_N, v_0,...,v_N) \to \min$$

В этой задаче математического программирования управляемыми переменными, которые определяют структуру дерева, являются числа u_j, v_k.

Дерево с минимальной ценой при заданных вероятностях p_i, q_j называется *оптимальным*.

Оптимальные деревья обладают важным свойством: *все поддеревья оптимального дерева также оптимальны*. В [9] описан ряд основанных на динамическом программировании алгоритмов построения оптимальных деревьев.

6.5. Префиксный символьный поиск

6.5.1. Префиксное дерево

При префиксном символьном поиске сравнение ключей связано не с проверкой числового соотношения $K < K_i$, а с последовательным учётом символов, входящих в состав ключа поиска (как это делается при поиске перевода данного слова с помощью словаря). В качестве ключа может использоваться любая строка из символов данного алфавита (включая символ конца слова). Например, ключ может задаваться как последовательность цифр и/или букв. На практике обычно, символы алфавита кодируются числами.

Введём необходимые определения.

Алфавитом будем называть конечный набор различных символов $S = \{s_1, s_2, ..., s_M\}$ (включая символ конца слова Θ). Словом в этом алфавите называется любой набор его символов, заканчивающийся символом Θ.

Решается задача поиска слов, которые, возможно, хранятся в базе данных («пробить по базе»).

База данных B реализована в виде M *-арного дерева,* то есть дерева, у которого каждый узел является корнем не более чем M поддеревьев (иными словами, каждый узел является родителем не более чем для M узлов).

Префиксом узла уровня l называется последовательность l символов данного алфавита $[s_{i_1}, s_{i_2}, ..., s_{i_l}]$ (среди которых могут быть и совпадающие). Префиксом корня является пустое слово (состоящее только из символа конца слова).

Узел уровня l — это M -мерный вектор

$$u^{(l)} = (u_1^{(l)}, u_2^{(l)}, ..., u_M^{(l)});$$

каждая его компонента $u_i^{(l)}$ соответствует определённому символу s_i алфавита и может иметь в качестве значения:

- либо слово, начинающееся с префикса узла — в том случае, когда в базе B имеется единственное слово, начинающееся с таким префиксом; именно оно является целью поиска;
- либо ссылку на узел уровня $l+1$, если префикс не является законченным словом, но в базе есть слова с таким началом; в $(l+1)$-м узле в этом случае следует смотреть компоненту, соответствующую символу префикса s_{l+1}
- либо пустую ссылку $\varnothing$, если слова с таким префиксом в базе нет.

Реализованная таким образом база данных называется *префиксным деревом* (или *лучом*, или *нагруженным деревом*).

Пример. База данных состоит из слов

{лес, лето, лото, лотос, лось, мал, мала, мол, мел, мех, меха, месть, смета, сом, соха, тол, холл}

в алфавите {м, л, х, с, т, а, о, ь, е, Θ}.

Символ конца слова Θ подразумевается присутствующим в конце каждого слова (на рис. 45, изображающем соответствующее префиксное дерево, символ Θ заменён жирным подчёркиванием клетки с полным словом). Префиксное дерево содержит пять уровней, включая корень t.

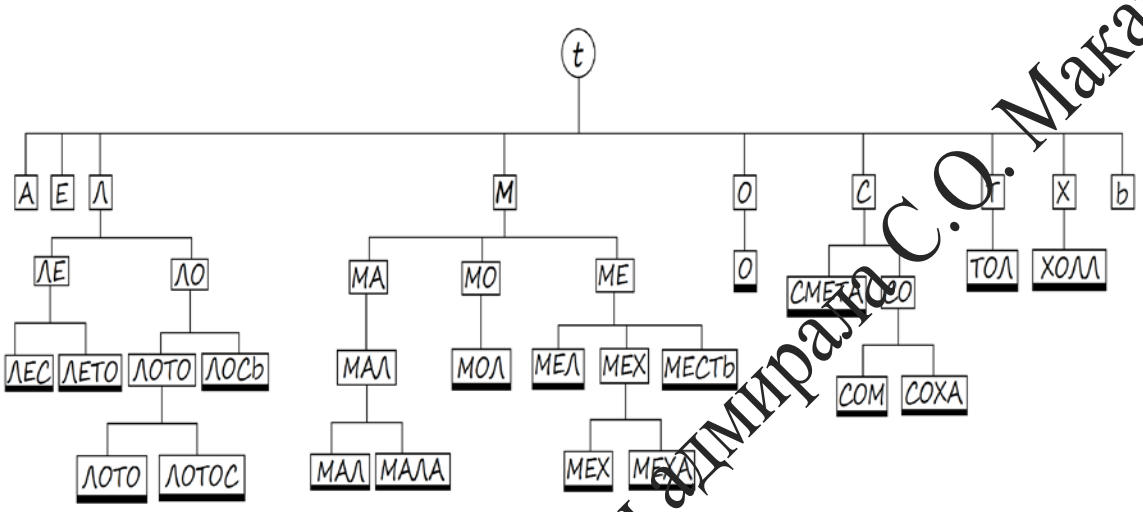


Рис. 45.

6.5.2. Алгоритм префиксного поиска

Префиксный (или *лучевой*) *поиск* — это поиск по префиксам в базе, представленной префиксным деревом. Символы ключа обычно представляются числами в диапазоне от 0 до $M-1$, где символу конца слова соответствует нуль, так что поиск называют также *цифровым*.

Алгоритм. Пусть t — указатель на дерево (то есть адрес корня); p — ссылка на узел; $u(p)$ — вектор в текущем узле p; ключ $K = s_1 s_2 ... s_n$, где каждое s_k — символ алфавита; k — текущий номер символа в ключе.

Ш1. Начальная установка.
$p := t$ — ссылка указывает вначале на корень луча;
$k := 0$.

Ш2. Переход к узлу следующего уровня:
$k := k + 1$;
$x := u_k$; здесь u_k — компонента узлового вектора, закреплённая за символом s_k;
если x является ссылкой, то
перейти к **Ш3**;

если x является ключом, то
перейти к **Ш4** (x — это единственный претендент на совпадение с ключом поиска).

Ш3. Продвижение по лучу:
если $x \neq \varnothing$, то
$p := x$;
вернуться к **Ш2**;
если $x \neq \varnothing$, то
ключ в базе данных отсутствует.

Ш4. Сравнение ключа со словом в соответствующей компоненте вектора u:
если $x = K$, то
ключ найден, p указывает на искомую запись;
иначе
ключ отсутствует.

⊥

NB: При неудачном поиске он завершается со значением x, наиболее близким к ключу K.

Префиксный поиск со вставкой. Алгоритм поиска легко дополняется до *алгоритма поиска со вставкой* отсутствующего ключа в базу: пустая ссылка в компоненте вектора на шаге **Ш4** заменяется ключом поиска. Таким образом, подавая в корень очередные слова для поиска со вставкой, можно сформировать базу данных в виде префиксного дерева.

Бинарный префиксный поиск. Это частный случай префиксного поиска по M-арному дереву при $M = 2$. Каждый символ исходного алфавита кодируется двоичным числом фиксированной длины. Теперь очередной символ ключа поиска — бит с возможными значениями 0 или 1. Эти значения определяют выбор ветки в двоичном дереве при переходе к очередному узлу.

6.6. Поиск по вторичным ключам

Возможна ситуация, когда несколько полей $K_1,...,K_m$ отдельной записи файла, вместе или по отдельности, могут участвовать в задании поиска. Особенно часто такие ситуации возникают при работе с *базами данных*. Последние являются файлами разнообразной структуры; для работы с ними разработаны разнообразные языки программирования высокого уровня.

Инвертированные файлы

Для ускорения поиска с использованием нескольких ключей используются *инвертированные файлы*, связанные с исходным основным файлом. Отдельная запись инвертированного файла состоит из значения ключа поиска и адресной части. Адресная часть может содержать либо набор физических адресов записей основного файла с данным значением ключа, либо указатель на цепной список из таких адресов.

Изменения в основном файле — вставка, удаление записей, изменение значений полей, участвующих в построении ключа поиска — должны отражаться в инвертированных файлах.

Геометрические данные. Рассмотрим ситуацию, когда ключевые поля файла задают координаты точек. Например, это могут быть широта и долгота географических точек (населённых пунктов, пересечений дорог и т.п.) на поверхности Земли; другой пример — декартовы координаты точек пространства или плоскости; третий пример — номера строки и столбца точки (пикселя) экрана. Для ускорения поиска всех объектов, отвечающих заданным диапазонам изменения координат, может использоваться инвертированный файл, отвечающий дискретной сетке координат с некоторым шагом. Тогда ключевым полем отдельной записи инвертированного файла является набор округлённых значений координат, а в ссылочной части отражается список физических адресов записей основного файла, в которых значения координат тяготеют к данному узлу координатной сетки.

Комбинаторное хеширование. *Хешированием* (более подробно см. гл. 7) называют способ отображения множества с потенциально большим числом элементов в множество с малым числом элементов $h: A \to B$, при котором каждый образ $b \in B$ имеет примерно одинаковое максимально возможное число прообразов $a \in A$, таких что $h(a) = b$.

Для ключа поиска типа “битовая строка” возможны запросы на поиск «по маске», в которых задаётся только часть двоичных цифр. В тех позициях, в которых соответствующая цифра может быть произвольной, в запросе могут стоять, например, звёздочки «*». Так, если ключ является 8-битовой строкой, то поиск всех записей, у которых в разрядах с нечётными номерами стоит 1, а в последнем восьмом разряде — 0, делается с помощью запроса по маске вида <1*1*1*10>.

Для быстрого поиска по «запросам со звёздочкой» применяется технология инвертированных файлов. Каждая возможная маска является ключом, а адресная часть содержит, например, указатель на связанный список из физических адресов подходящих под маску записей основного файла.

Глава 7. ХЕШИРОВАНИЕ

7.1. Исходные понятия

Исходной ситуацией для процедуры, называемой хешированием, является наличие динамического массива x элементов, содержащих ключ поиска K, причём имеются основания считать, что количество фактически имеющихся элементов значительно меньше максимально возможного количества разных ключей. В этом случае нецелесообразно резервировать последовательную область памяти под весь потенциально возможный набор значений ключа, поскольку массив окажется редко заполненным. Другой предпосылкой является цель затруднить восстановление секретного значения $x(K)$, например, пароля, по ключу K (например, по фамилии) в случае несанкционированного доступа.

Задача поиска элемента $x(K)$ по ключу K, в конечном счёте, сводится к определению адреса $adr(K)$, по которому этот элемент хранится в массиве a_x существенно меньшего размера.

Определение этого адреса может осуществляться с помощью промежуточного ключа (*хеша, хеш-кода*) $h(K)$, которому уже однозначно соответствует адрес $adr(K)$:

$$K \xrightarrow{h} h(K) \xrightarrow{f} adr(h(K)),$$

так что

$$adr(K) = f(h(K)).$$

Обозначим множество возможных значений ключа поиска через K, а множество возможных значений хеш-кода через H. Отображение $\mathrm{K} \xrightarrow{h} \mathrm{H}$ называется *хеш-функцией*.

Для обеспечения цели экономии памяти количество $|\mathrm{H}|$ возможных значений ключа хеш-кода должно быть существенно меньше, чем количество ключей $|\mathrm{K}|$. Например, если ключом является набор из десяти латинских букв, то $|\mathrm{K}| = 26^{10}$. На практике, однако, редко оказываются задействованными все возможные значения ключей, что является предпосылкой для отказа от взаимной однозначности отображения $K \to h(K)$. Допускается возможность совпадения хеш-кодов, называемого в этом случае *коллизией*:

$$h(K) = h(K') \text{ при } K \neq K'.$$

Конкретный вид хеш-функций зависит от типа данных, которым реализован ключ. Наиболее частыми являются ситуации, когда ключ является двоичным или десятичным числом, либо строкой символов. Разумеется, в конечном счёте, все типы ключа оказываются представленными в компьютере битовыми строками (наборами из нулей и единиц). Хеш-код при этом

имеет тот же тип данных, что и ключ, но реализуется в существенно меньшем диапазоне числовых значений либо существенно более короткой строкой.

Хеширование является промежуточным этапом многих криптографичеких процедур. Выбор конкретной хеш-функции определяется спецификой решаемой задачи. Важными характеристиками хеш-функций служат разрядность аргумента и функции, вычислительная сложность и криптостойкость. С криптографическим аспектом хеширования связано требование случайного перемешивания ключей $h(K)$ — случайного в том смысле, что хотя функция h задаётся детерменированным алгоритмом, её значения внешне выглядят непредсказуемыми.

7.2. Методы хеширования

Пусть значения ключа являются неотрицательными целыми числами (так всегда можно интерпретировать представляющую ключ K битовую строку).

7.2.1. Хеширование методом деления

В методе деления хеш-функция задаётся как вычет ключа K по модулю некоторого числа m (то есть как остаток от деления нацело K на m):

$$h(K) = K \bmod m.$$

В этом случае хеш-коды ключей образуют множество $\mathrm{H} = \{0,1,...,m-1\}$, и их количество $|\mathrm{H}| = m$.

Для того чтобы в формировании $h(K)$ участвовали все разряды двоичного представления K, рекомендуется выбирать в качестве модуля m простое число, далёкое от степени двойки.

Максимально возможное число коллизий $l_{\max}$ на единицу больше целой части дроби $|\mathrm{K}|/m$:

$$l_{\max} = [|\mathrm{K}|/m] + 1.$$

Например, если $\mathrm{K} = \{1,2,...,19\}$, то

$$|\mathrm{K}| = 19,\ m = 3,\ [|\mathrm{K}|/3] + 1 = 6 + 1 = 7,$$

и семь чисел $1, 4, 7, 10, 13, 16, 19$ имеют одинаковый остаток 1 по модулю 3.

7.2.2. Хеширование методом умножения

Пусть желаемое количество различных хеш-кодов равно m. Выберем некоторое число $a \in (0,1)$. В методе умножения хеш-функция задаётся формулой:

$$h(K) = [m \cdot \{Ka\}],$$

то есть $h(K)$ — целая часть произведения m на дробную часть числа Ka. Например, если $K=1208,\ a=0.017,\ m=100$, то $Ka=20.536$, $\{Ka\}=0.536$, $m\cdot 0.536=53.6$, $h(K)=[53.6]=53$.

Возможна следующая модификация метода умножения. Пусть битовая строка для представления ключа K имеет l разрядов. В качестве m можно выбрать степень двойки: $m=2^j$, где $j<l$. Выберем целое $s\in(0,2^l)$ и положим $a=s/2^l$. Тогда вычисление Ka можно провести, минуя деление:

$$Ka=K\cdot(s/2^l)=(Ks)/2^l.$$

Произведение целых чисел Ks занимает $2l$ битов, и переход от Ks к $[Ks/2^l]$ приводит к l младшим разрядам произведения Ks. Теперь в качестве кода $h(K)$ можно взять j старших из этих l разрядов.

7.2.3. Универсальное хеширование

Универсальное хеширование основано на принципе случайного выбора хеш-функции $\mathrm{K}\xrightarrow{h_i}\mathrm{H}$ из некоторого заранее сформированного множества $\Phi=\{h_1,h_2,...,h_N\}$ при каждом запуске алгоритма хеширования. Благодаря этому хеширование не будет работать постоянно плохо (то есть не будет давать неприемлемо большое число коллизий) даже для таких совокупностей ключей, на которых плохо работает некоторая фиксированная хеш-функция.

Пусть ключ K — натуральное число, $m=|\mathrm{H}|$ — желаемое число различных хеш-кодов.

Множество хеш-функций Φ называется *универсальным*, если для каждой пары ключей $K,K'\in\mathrm{K}$ при случайном выборе $h_j\in\Phi$ вероятность коллизии не превосходит $1/m$:

$$P\big(h_j(K)=h_j(K')\big)\le 1/m.$$

Универсальное множество Φ можно построить следующим образом. Пусть $\mathbb{Z}_p=\{0,1,...,p-1\}$ — полная система вычетов по *простому* модулю $p>m$, $\mathbb{Z}_p^*=\{1,...,p-1\}$ — система ненулевых вычетов. Если $a\in\mathbb{Z}_p^*,\ b\in\mathbb{Z}_p$, то определяем хеш-функцию $h_{a,b}\in\Phi$ формулой $h_{a,b}(K)=g_{a,b}(K)\bmod m$, где $g_{a,b}(K)=(aK+b)\bmod p\in\mathbb{Z}_p$ — остаток целочисленного деления $aK+b$ на p.

Убедимся, что множество Φ универсально.

Из теории чисел известно ([19], с. 41), что если K пробегает полную систему вычетов по модулю p, то соответствующие значения $(aK+b)\bmod p$, в силу взаимной простоты a и p, также пробегают полную систему вычетов. Функции $g_{a,b}(K)\bmod p$, рассматриваемые как отображе-

ния $\mathbb{Z}_p \to \mathbb{Z}_p$, различны для разных пар (a,b), то есть, по крайней мере, для одного вычета $r \in \mathbb{Z}_p$ выполняется $g_{a,b}(r) \neq g_{a',b'}(r)$ при $(a,b) \neq (a',b')$.

Действительно, если $b \neq b'$, то $g_{a,b}(0) = b,\ g_{a',b'}(0) = b'$. Если же $b = b',\ a \neq a'$, то $g_{a,b}(1) = a + b,\ g_{a',b'}(1) = a' + b$, и

$$((a+b) - (a'+b)) \bmod p = (a - a') \bmod p \neq 0 \bmod p.$$

Таким образом, количество N функций $g_{a,b}$ равно $(p-1)p$.

Для любой функции $h_{a,b}$ вероятность коллизии при $K, K' \in \mathbb{Z}_p, K \neq K'$ — это с вероятность попадания ключей в один класс вычетов по модулю m:

$$P\left(h_{a,b}(K) = h_{a,b}(K')\right) = P\left((g_{a,b}(K) \equiv g_{a,b}(K')\right) \bmod m.$$

Для ключа K в множестве $\{0,1,..., p-1\}$ имеется не более чем $\lceil p/m \rceil - 1$ значений K', сравнимых с K по модулю m и отличных от K. (Например, при $p = 17,\ m = 3,\ K = 7$ среди чисел $\{0,1,...,16\}$ имеется 6 значений, сравнимых с 7 по модулю 3 — это числа 1,4,7,10,13,16; исключая само $K = 7$, получаем $5 = \lceil 17/3 \rceil - 1$ значений). Применяя к выражению $\lceil p/m \rceil - 1$ неравенство (15) (см. гл. 1, с. 17), получаем, что количество подходящих для коллизии ключей K' не превосходит числа

$$((p+m-1)/m) - 1 = (p-1)/m.$$

Поскольку общее количество ключей в $\mathbb{Z}_p$, отличных от K, есть $p-1$, то

$$P\left(h_{a,b}(K) = h_{a,b}(K')\right) \leq ((p-1)/m)/(p-1) = 1/m.$$

7.3. Методы разрешения коллизий

7.3.1. Метод цепочек

Метод цепочек предписывает объединение элементов с совпадающими хеш-кодами в цепной список, который просматривается последовательно вплоть до обнаружения требуемого ключа (либо до исчерпания списка при его отсутствии).

Значением хеш-кода $h(K)$ является номер элемента массива T_x (*хеш-таблицы*), в котором хранится указатель на первый элемент списка $L(K)$, содержащего все элементы $x(K')$ с $h(K') = h(K)$. При отсутствии таких элементов значением элемента массива T_x является специальный признак $\varnothing$ (или *NIL*).

Операции вставки нового элемента $x(K)$ или удаления элемента из хеш-таблицы T_x сводятся к соответствующим операциям над списком $L(K)$.

7.3.2. Метод открытой адресации

Метод открытой адресации предполагает размещение в хеш-таблице не указателя на список элементов с данным значением хеш-кода, а самих элементов. По мере заполнения хеш-таблицы значения *NIL* меняются на $x(K)$. Этот метод не позволяет хранить более чем $|\mathrm{H}|$ элементов, где H — множество значений хеш-кода.

Устранение возникающей при вставке нового элемента коллизии возможно при не полностью заполненной хеш-таблице. Для этого применяется *повторное хеширование* — вычисление по определённому алгоритму тех адресов (номеров элементов таблицы), которые следует проверить при поиске или вставке элемента для обнаружения возможного присутствия $x(K)$ в T_x. Повторное хеширование позволяет вычислить последовательно все адреса, по которым могут располагаться элементы с хеш-кодом $h(K)$).

При открытой адресации не тратится память на указатели — вместо указателя на список $L(K)$ хранится тот из элементов с одинаковым хеш-кодом, который был записан в таблицу первым.

Классификация методов открытой адресации ориентирована на алгоритм повторного хеширования.

Пусть хеш-таблица содержит m ячеек с номерами $0,1,...,m-1$. Последовательность номеров исследуемых ячеек зависит при открытой адресации от ключа K вставляемого элемента и не совпадает с естественным порядком от 0 до $m-1$. Исходную хеш-функцию h вместе с правилом просмотра хеш-таблицы T_x можно считать новой процедурой хеширования

$$\bar{h}: \mathrm{K} \times \{0,1,...,m-1\} \to \{0,1,...,m-1\},$$

когда номер очередной исследуемой ячейки i зависит от ключа K и от предшествующего проверенного номера. При этом требуется, чтобы при каждом фиксированном значении K последовательность проверяемых номеров $\bar{h}(K,0), \bar{h}(K,1),..., \bar{h}(K,m-1)$ давала перестановку множества $\{0,1,...,m-1\}$. Тогда будет обеспечен просмотр всей таблицы без пропусков и повторений.

Метод линейных проб. В этом методе перестановка задаётся циклическим сдвигом на m, начиная с первого проверяемого номера $h(K)$:

$$\bar{h}(K,i) = \big(h(K)+i\big) \bmod m.$$

Недостатком метода линейных проб является тенденция к увеличению длины цепочек последовательно занятых ячеек (*кластеров*), что увеличивает среднее время поиска незанятой ячейки. Действительно, если в хеш-таблице из m элементов имеется кластер из q элементов с номерами $h+1,...,h+q$, то следующий за этим кластером её элемент с номером

$h+q+1$ окажется подходящим, если хеш-номер попадет в один из номеров кластера (тогда нужно будет двигаться по цепочке до первого свободного, то есть до $h+q+1$), либо если хеш-номер сразу окажется равным $h+q+1$. Вероятность этого, подсчитанная по схеме равновозможных исходов, равна $(q+1)/m$ — растёт с увеличением q.

Метод квадратичных проб. В этом методе перестановка задаётся формулой:

$$\bar{h}(K,i) = \left(h(K) + \alpha i + \beta i^2\right) \bmod m.$$

Первой проверяется ячейка $h(K)$, которая сдвигается затем циклически на $\alpha i + \beta i^2$. Для того, чтобы были проверены все ячейки без повторений, значения сдвига при $i = 0,1,...,m-1$ должны давать полную систему вычетов по модулю m. Рассмотрим два случая.

1. Пусть m — простое число, $\bar{h}(K,i) = \left(h(K) + i^2\right) \bmod m$ (здесь α=0, β=1). Для $0 \le i < j < m$ совпадение адресов $\bar{h}(K,i) = \bar{h}(K,j)$ означает, что имеет место сравнение

$$j^2 \equiv i^2 (\bmod m) \Leftrightarrow (j-i)(j+i) \equiv 0 (\bmod m).$$

Ввиду условий на i и j, отсюда следует, что $(j-i,m)=1$, и тогда $j \equiv i (\bmod m)$. Следовательно, $j > m/2$, так что *совпадение адресов случится самое раннее после $m/2$ попыток, то есть когда уже занято не менее половины ячеек.*

2. Пусть $m = 2^k$, и $\bar{h}(K,i) = \left(h(K) + \frac{1}{2}i + \frac{1}{2}i^2\right) \bmod 2^k$. Для $0 \le i < j < 2^k$ совпадение адресов означало бы, что имеет место сравнение

$$\frac{1}{2}(j^2 - i^2) + \frac{1}{2}(j-i) \equiv 0 (\bmod 2^k) \Leftrightarrow$$

$$\Leftrightarrow \frac{1}{2}(j-i)(j+i+1) \equiv 0 (\bmod 2^k). \qquad (*)$$

Если $j-i$ чётно, то $j+i+1 = (j-i)+2i+1$ нечётно, и, следовательно, взаимно просто с 2^k. Тогда

$$(*) \Leftrightarrow \frac{1}{2}(j-i) \equiv 0 (\bmod 2^k) \Leftrightarrow (j-i) \equiv 0 (\bmod 2^{k+1}),$$

что невозможно, ввиду условий на i и j.

Если $j-i$ нечётно, то $j+i+1$ чётно. Выделим в нём максимальную степень двойки: $j+i+1 = 2^l(2t+1)$. Тогда

$$(*) \Leftrightarrow \frac{1}{2} \cdot 2^l(2t+1) \equiv 0 (\bmod 2^k) \Leftrightarrow$$

$$\Leftrightarrow 2t+1 \equiv 0 (\bmod 2^{k-l+1}) \Leftrightarrow l \ge k+1,$$

что также невозможно ввиду условий на i и j.

Таким образом, в случае

$$\overline{h}(K,i)=\left(h(K)+\frac{1}{2}i+\frac{1}{2}i^2\right)\bmod 2^k$$

будет просмотрена вся таблица без пропусков и повторений.

Метод двойного хеширования. В этом методе перестановка задаётся формулой

$$\overline{h}(K,i)=(h_1(K)+ih_2(K))(\bmod m),$$

где h_1 и h_2 — хеш-функции. Двойное хеширование является одним из наилучших методов получения перестановки, которая в этом случае обладает многими характеристиками случайной перестановки.

Для того чтобы последовательность проб гарантировала просмотр всей таблицы, достаточно выбрать в качестве m степень двойки ($m=2^t$) и построить функцию h_2 так, чтобы все её значения были нечётными. Действительно, в этом случае разность $jh_2(K)-ih_2(K)=(j-i)h_2(K)$ может делиться на m только в случае $i\equiv j(\bmod m)$, так нечетное число $h_2(K)$ взаимно просто с m.

ЛИТЕРАТУРА

1. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. М., СПб., Киев: Вильямс, 2001.

2. Берж К. Теория графов и её применения. М.: ИЛ, 1962. — 320 с.

3. Вирт Н. Алгоритмы и структуры данных. СПб.: Невский диалект, 2008. — 352 с. 6.

4. Воробьёв Н.Н. Числа Фибоначчи. М.: Наука, 1978. — 144 с.

5. Дал О., Дейкстра Э., Хоар К. Структурное программирование. М.: Мир, 1975. — 247 с.

6. Зыков А.А. Основы теории графов. М.: Вузовская книга, 2004. — 664 с.

7. Кнут. Д. Искусство программирования. Том I. Основные алгоритмы. М.: «Вильямс». 2004. — 720 с.

8. Кнут. Д. Искусство программирования. Том I. Получисленные алгоритмы. М.: «Вильямс». 2003. — 832 с.

9. Кнут. Д. Искусство программирования. Том III. Сортировка и поиск. М.: «Вильямс». 2004. — 832 с.

10. Кормен Т. и др. Алгоритмы: построение и анализ. М.: «Вильямс». 2005. — 1296 с.

11. Коутинхо С. Введение в теорию чисел и алгоритм RSA. — М.: Постмаркет, 2001. — 328 с.

12. Криницкий Н.А., Миронов Г.А., Фролов Г.Д. Программирование и алгоритмические языки. М.: Наука, 1979. — 512 с.

13. Лорин Г. Сортировка и системы сортировки. М.: Наука, 1983. — 384 с.

14. Оре О. Теория графов. М.: Либроком, 2009. — 354 с.

15. Седжвик Р. Фундаментальные алгоритмы на С. Анализ/Структуры данных/Сортировка/.: СПб.: ДиаСофтЮП, 2003. — 672 с.

16. Шауман А.М. Основы машинной арифметики. Л.: ЛГУ, 1979. — 312 с.

17. Яглом А.М., Яглом И.М. Вероятность и информация. М.: «КомКнига». 2007. — 512 с.

18. Ястребов М.Ю. Математика. Теория вероятностей. Часть I. Вероятности случайных событий. СПб.: СПГУВК, 2004. — 43 с.

19. Ястребов М.Ю., Ныркова А.П. Основы теории чисел. — СПб.: СПГУМРФ. 2013. — 55 с.

СОДЕРЖАНИЕ

Михаил Юрьевич Ястребов

ПРЕДСТАВЛЕНИЕ ДАННЫХ И АЛГОРИТМЫ ИХ ОБРАБОТКИ

Учебное пособие