Содержание

1. Математические основы анализа алгоритмов

1.1. Лучший, средний, худший случаи

В этом и следующих разделах основной целью будет получение точной оценки времени выполнения для некоторых конкретных алгоритмов.

Рассмотрим следующую задачу: требуется найти максимальный элемент в массиве из n чисел. Для решения можно предложить следующий алгоритм (процедура 1.1.1):

```
Процедура 1.1.1. Get Max Element
01:  Get Max Element Impl(A[1, n])
02:        result = A[1]
03:        for i = 1 to n do
04:              if A[i] > result then
05:                    result = A[i]
06:        return result
```

Подсчитаем число шагов выполнения этого алгоритма как можно точнее. Первый оператор выполняется один раз. Оператор цикла – n раз. Проверка условия выполняется при каждой итерации цикла, т. е. тоже n раз. Особого внимания заслуживает оператор $\{5\}$ изменения результата. Количество выполнений этого оператора зависит от данных исходной задачи и не выражается в явной форме. В большинстве алгоритмов встречаются операторы, количество выполнений которых зависит от исходных данных. В связи с этим при точной оценке сложности алгоритма принят такой подход, когда вместо всех возможных наборов входных данных проводится оценка сложности в трех случаях – *наилучшем*, *наихудшем* и *среднем*.

Чтобы оценить количество выполнений оператора изменения результата, используем вычислительный эксперимент. Создадим метод, который генерирует все возможные перестановки заданного числового множества $\{1,2,\dots,n\}$. Затем для каждой из них будем запускать поиск максимума и считать число выполнений оператора $\{5\}$. Для множества из десяти элементов статистика по выполнению оператора присваивания имеет следующий вид:

Количество выполнений оператора	**Количество перестановок**	**Вероятность**
0	362880	0,1
1	1026576	0,282896825
2	1172700	0,323164683
3	723680	0,199426808
4	269325	0,07421875
5	63273	0,017436343
6	9450	0,002604167
7	870	0,000239749
8	45	1,24008e–05
9	1	2,75573e–07

Вероятность в третьей колонке таблицы посчитана как отношение значения второй колонки к сумме всех значений во второй колонке. Зная вероятно-

сти, можно построить график распределения случайной величины, соответствующей указанному количеству выполнений оператора {5} (рис. 1.1).

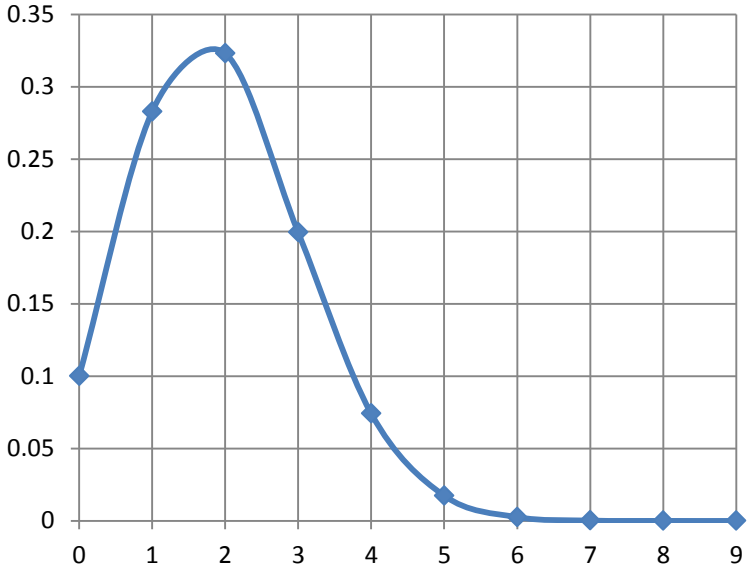

Рис. 1.1. График распределения случайной величины

Из приведенного графика можно видеть наихудший и наилучший случаи для оператора {5}. В наихудшем случае, который работает для единственной перестановки, количество выполнений оператора {5} равно 9. Лучший случай – количество выполнений оператора равно 0. Оценка для среднего случая находится по следующему правилу: для каждой строки таблицы значение из первого столбца умножается на значение из третьего столбца, после чего результат складывается. Получим приблизительно 1,93, т. е. в среднем оператор {5} выполняется два раза.

Вычислительный эксперимент обычно сложно провести для больших n. Поэтому анализ сложности алгоритма проводят аналитически. Рассмотрим все случаи для алгоритма поиска максимума. Наилучший ($B(n)$) – это случай, когда максимальный элемент является первым элементом в массиве. Тогда количество выполнений оператора {5} будет равно нулю. Наихудший случай ($W(n)$) соответствует ситуации, когда исходный массив отсортирован по возрастанию. Тогда оператор {5} будет выполняться при каждой итерации цикла, кроме последней, т. е. $n - 1$ раз.

При среднем случае ($A(n)$) для вывода оценки количества выполнений оператора {5} сделаем следующее предположение о природе элементов исходного массива: будем считать, что все они различны. Без этого предположения вывод оценки становится затруднительным. При первой итерации цикла оператор {5} не выполнится (это особенность реализации!). При второй итерации вероятность выполнения оператора равна $^1/_2$, так как есть возможность того, что добавляемый к множеству чисел (из одного элемента) новый элемент окажется среди них наибольшим. То есть можно сказать, что при второй итерации цикла оператор {5} «выполнится» $^1/_2$ раз. При третьей итерации цикла вероятность выполнения оператора {5} равна $^1/_3$: числовое множество состоит из двух эле-

ментов; добавляем третий, и вероятность того, что он наибольший равняется $^1/_3$. Таким образом, число выполнений оператора {5} выражается формулой

$$S = 0 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Замкнутое выражение для S получаем, воспользовавшись формулой для частичной суммы гармонического ряда:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + \gamma + \frac{1}{2n} + o\left(\frac{1}{n}\right),$$

где $\gamma = 0{,}57721566\ldots$ – *постоянная Эйлера*.

Тогда $S = H_n - 1 \approx \ln n + \frac{1}{2n} - 0{,}42 \approx \ln n$.

Данная оценка является весьма точной (с точностью до аддитивной константы). Таким образом, для количества выполнений оператора {5} получены следующие результаты: $B(n) = 0$, $W(n) = n - 1$, $A(n) = \ln n$.

Теперь легко можно подсчитать общее число выполнений операторов в анализируемом алгоритме. Требуется рассмотрение трех случаев. Кроме этого, иногда для операторов вводят так называемые *стоимости выполнения*, характеризующие время выполнения отдельного оператора. Пусть, например, стоимости выполнения операторов процедуры (1.1.1) равны c_2, c_3, c_4, c_5. Тогда время выполнения алгоритма следующее:

- $B(n) = c_2 + nc_3 + nc_4 + 0 \cdot c_5 = c_2 + n(c_3 + c_4)$;
- $W(n) = c_2 + n(c_3 + c_4) + (n-1)c_5$;
- $A(n) = c_2 + nc_3 + nc_4 + \ln n \cdot c_5$.

Можно видеть, что при выполнении общей задачи точного анализа алгоритмов активно используется математический аппарат, в частности аппарат теории вероятностей.

1.2. Рекуррентные соотношения

Пусть задана последовательность $\{a_n\} = a_1, a_2, \ldots, a_n \ldots$ Тогда формулу для вычисления ее элементов будем называть *рекуррентной*, если при вычислении каждого члена последовательности необходимо знать предыдущие[*].

Последовательность $\{a_n\}$ называется *рекуррентной порядка* k ($k \in \mathbb{N}$), если существует формула $a_{n+k} = f(a_{n+k-1}, a_{n+k-2}, \ldots, a_n)$, с помощью которой элемент a_{n+k} вычисляется по k предыдущим элементам $a_{n+k-1}, a_{n+k-2}, \ldots, a_n$. Данная формула называется *рекуррентным соотношением* (рекуррентным уравнением) порядка k. Обратим внимание на то, что первые k элементов последовательности $\{a_n\}$ должны быть известны. Примером рекуррентного соотношения может служить последовательность Фибоначчи:

[*] Рекуррентная (от фр. *récurrente*) – возвращающаяся к началу.

$$\begin{cases} f_0 = 0, f_1 = 1, \\ f_{n+2} = f_{n+1} + f_n, n \geq 0 \end{cases} \text{ или } \begin{cases} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2. \end{cases}$$

Более формально рекуррентное соотношение определяется как *рекурсивная функция* с целочисленными значениями.

Определение 1.1. Рекуррентное соотношение вида

$$a_{n+k} = \alpha_1 \cdot a_{n+k-1} + \alpha_2 \cdot a_{n+k-2} + \cdots + \alpha_k \cdot a_n, \tag{1.1}$$

где $n \geq k > 0$, называется *однородным линейным*.

Определение 1.2. Рекуррентное соотношение вида

$$a_{n+k} = \alpha_1 \cdot a_{n+k-1} + \alpha_2 \cdot a_{n+k-2} + \cdots + \alpha_k \cdot a_n + f(n), \tag{1.2}$$

где $n \geq k > 0$, называется *неоднородным линейным*.

Помимо вышеупомянутых уравнений (1.1) и (1.2), выделим отдельный класс уравнений, которые можно описать соотношением вида

$$\begin{cases} T(1) = c, \\ T(n) = a \cdot T(n/b) + f(n), n > 1, \end{cases} \tag{1.3}$$

где $a \geq 1$ и $b \geq 1$; c – константа; функция $f(n)$ – асимптотически положительная; n/b может рассматриваться как $\lfloor n/b \rfloor$, так и $\lceil n/b \rceil$.

В более общем виде уравнение (1.3) может быть приведено к виду

$$T(n) = \sum_{i=1}^{k} a_i \cdot T\left(\left\lfloor \frac{n}{b_i} \right\rfloor\right) + f(n), n > 1, \tag{1.4}$$

где $k \geq 1$; все коэффициенты $a_i \geq 0$ и $\sum a_i \geq 1$; все $b_i > 1$; функция $f(n)$ – ограниченная, положительная и неубывающая; для любой константы $c > 1$ существуют константы n_0 и $d > 0$, такие, что для всех $n > n_0$ справедливо соотношение $f(n/c) \geq d \cdot f(n)$.

Методы решения рекуррентных уравнений

Очень часто оказывается полезным получение замкнутого вида рекуррентного соотношения, т. е. такого вида, в котором значение a_n выражается через номер n. Получение замкнутого вида рекуррентного соотношения имеет смысл лишь в том случае, если требуется оценить скорость роста функции $\{a_n\}$ либо быстро вычислить значение a_n для достаточно больших n. Существует множество методик, которые позволяют построить такой вид. Основные из них будут рассмотрены ниже.

Метод подстановки

Метод подстановки работает в том случае, если решаемое уравнение можно путем замены переменных свести к уравнению, для которого решение уже известно. Например, пусть требуется найти решение в замкнутом виде уравнения $T(n) = 2T\left(\sqrt{n}\right) + 1$. Будем считать, что $n = 2^m$. С учетом этого рекуррентное соотношение примет вид $T(2^m) = 2T\left(2^{m/2}\right) + 1$ или, после замены $T(2^m)$ на $S(m)$,

$$S(m) = 2S(m/2) + 1. \quad (1.5)$$

Структура уравнения (1.5) совпадает со структурой уравнения (1.3), для которого существует решение в замкнутом виде. Предположив, что решение уравнения (1.5) имеет вид $F(m)$, можно записать решение исходного уравнения как $T(n) = F(\log_2 n)$.

Метод подстановки очень хорошо подходит для решения рекуррентных уравнений, описывающих декомпозиционные методы. Так как функция $T(n)$ – не убывающая, то решение можно искать подбором. Решением будет такая функция $g(n)$, для которой верно неравенство

$$g(n) \geq a \cdot g(n/b) + f(n).$$

Из всех таких функций требуется выбрать функцию наименьшего порядка. Рассмотрим идею метода на следующем примере. Требуется решить уравнение $T(n) = 2T(\sqrt{n}) + 1$. Пусть $g(n) = cn^2$, где c – некоторая положительная константа, отличная от нуля. Тогда $2c(\sqrt{n})^2 + 1 = 2c \cdot n + 1 < cn^2$, следовательно, $T(n) = O(n^2)$. Попробуем улучшить решение. Рассмотрим функцию $g(n) = c \cdot \log_2 \log_2 n$. Тогда

$$2c \log_2 \log_2 \sqrt{n} \vee c \log_2 \log_2 n \Leftrightarrow \frac{1}{2^{2c-1}} (\log_2 n)^{2c} \vee (\log_2 n)^c$$
$$\Leftrightarrow (\log_2 n)^c \vee 2^{2c-1} \Rightarrow (\log_2 n)^c > 2^{2c-1}.$$

Причем полученное неравенство выполняется, начиная с $n \geq 16$. Следовательно, $O(\log_2 \log_2 n)$ не является решением. Пусть $g(n) = c \cdot \log_2 n$. Тогда $2c \log_2 \sqrt{n} + 1 = c \log_2 n + 1 = O(\log_2 n)$. Из последнего равенства можно заключить, что трудоемкость алгоритма, описываемого рассмотренным рекуррентным соотношением, по крайней мере $T(n) = O(\log_2 n)$.

Метод итераций

Идея метода итераций заключается в том, чтобы расписать исследуемое рекуррентное соотношение через множество других и затем просуммировать полученные слагаемые. Например, требуется решить рекуррентное соотношение вида

$$\begin{cases} T(0) = 1, \\ T(n) = T(n-1) + c, n \geq 1. \end{cases}$$

Последовательно выражая $T(n-k)$, где $k \leq n$, через предыдущие члены, можем записать выражение для $T(n)$:

$$T(n) = T(n-1) + c = (T(n-2) + c) + c = \cdots = T(n-k) + k \cdot c$$

Несложно видеть, что данный процесс завершится при $k = n$. Следовательно, решением рекуррентного соотношения будет функция $T(n) = 1 + n \cdot c$.

Характеристические многочлены

Рассмотрим общую схему решения однородного линейного рекуррентного соотношения, имеющего вид

$$a_{n+k} = \alpha_1 \cdot a_{n+k-1} + \alpha_2 \cdot a_{n+k-2} + \cdots + \alpha_k \cdot a_n. \quad (1.6)$$

Так как всякая рекуррентная последовательность k-го порядка однозначно определяется заданием k ее первых членов, то можно сформировать следующий многочлен:

$$P(x) = x^k - \alpha_1 \cdot x^{k-1} - \alpha_2 \cdot x^{k-2} - \cdots - \alpha_k. \quad (1.7)$$

Определение 1.3. Многочлен $P(x)$ называется *характеристическим многочленом* однородного линейного рекуррентного уравнения (1.6).

Пусть λ – корень характеристического многочлена $P(x)$. Тогда последовательность $a_0, a_1, a_2, \ldots, a_n$, где $a_n = c \cdot \lambda^n$, c – произвольная константа, удовлетворяет рекуррентному соотношению (1.6). Действительно, после подстановки значений a_i в (1.5) получим

$$c \cdot \lambda^{n+k} - \alpha_1 \cdot c \cdot \lambda^{n+k-1} - \cdots - \alpha_k \cdot c \cdot \lambda^n = \\ = c \cdot \lambda^n(\lambda^k - \alpha_1 \cdot \lambda^{k-1} - \cdots - \alpha_k) = 0.$$

Если же λ – корень кратности r, то решением рекуррентного соотношения может быть любая из последовательностей $c_1 \cdot \lambda^n, \ldots, c_r \cdot n^{r-1}\lambda^n$, где $c_1, \ldots, c_r$ – произвольные константы. Заметим, что если существуют последовательности $\{a_n\}$ и $\{b_n\}$, удовлетворяющие уравнению (1.6), то и линейная комбинация $\{\alpha \cdot a_n + \beta \cdot b_n\}$ также является решением (1.6), где α, β – константы.

Теорема 1. Пусть $\lambda_1, \ldots, \lambda_k$ – простые корни характеристического многочлена $P(x)$. Тогда общее решение рекуррентного соотношения (1.5) будет иметь вид

$$a_n = c_1 \cdot \lambda_1^n + \cdots + c_k \cdot \lambda_k^n, \quad (1.8)$$

для любого n; $c_1, \ldots, c_k$ – константы.

Если же характеристический многочлен $P(x)$ имеет кратные корни $\lambda_1, \ldots, \lambda_s (s \le k)$ с кратностями $r_1, \ldots, r_s, \sum r_i = k$, то решение рекуррентного соотношения (1.9) можно записать в виде

$$a_n = \sum_{i=1}^{s} \lambda_i^n \cdot \sum_{j=0}^{r_i-1} c_{ij} \cdot n^j, \; n \ge 0, \quad (1.9)$$

для любого n; c_{ij} – константы.

Константы $c_1, \ldots, c_k$ в случае, если уравнение (1.6) имеет решение вида (1.8), можно определить из следующей системы уравнений:

$$\begin{cases} c_1 + c_2 + \cdots + c_k = a_0, \\ c_1 \cdot \lambda_1 + c_2 \cdot \lambda_2 + \cdots + c_k \cdot \lambda_k = a_1 \\ \cdots \\ c_1 \cdot \lambda_1^{k-1} + c_2 \cdot \lambda_2^{k-1} + \cdots + c_k \cdot \lambda_k^{k-1} = a_{k-1}. \end{cases} \quad (1.10)$$

Система (1.10) разрешима относительно $c_1, \dots, c_k$, так как ее определитель есть определитель Вандермонда, и он не равен нулю:

$$\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ \lambda_1^{k-1} & \cdots & \lambda_k^{k-1} \end{bmatrix} = \prod_{i<j} (\lambda_i - \lambda_j) \neq 0.$$

Если же уравнение (1.6) имеет решения вида (1.9), то рассуждения для нахождения свободных коэффициентов будут аналогичны.

Рассмотрим пример. Пусть требуется найти замкнутый вид рекуррентного соотношения Фибоначчи $f_{n+2} = f_{n+1} + f_n$. Характеристический многочлен для данного уравнения будет иметь вид $\lambda^2 - \lambda - 1 = 0$. Решением данного уравнения являются $\lambda_1 = (1 + \sqrt{5})/2$ и $\lambda_2 = (1 - \sqrt{5})/2$. Так как корни λ_1 и λ_2 являются простыми, то общий вид решения можно записать в виде $c_1(1 + \sqrt{5})/2 + c_2(1 - \sqrt{5})/2$, где c_1, c_2 – произвольные константы. Для определения констант c_1 и c_2 составим систему уравнений согласно (1.10)

$$\begin{cases} c_1 + c_2 = 0, \\ c_1(1 + \sqrt{5})/2 + c_2(1 - \sqrt{5})/2 = 1, \end{cases}$$

из которой $c_1 = +1/\sqrt{5}$ и $c_2 = -1/\sqrt{5}$.

С учетом полученных значений замкнутый вид для последовательности Фибоначчи примет вид

$$f_n = \frac{1}{\sqrt{5}}(\varphi^n - (1 - \varphi)^n), \varphi = \frac{1 + \sqrt{5}}{2}$$

Далее рассмотрим общую схему решения неоднородного линейного рекуррентного соотношения, имеющего вид

$$a_{n+k} = \alpha_1 \cdot a_{n+k-1} + \alpha_2 \cdot a_{n+k-2} + \cdots + \alpha_k \cdot a_n + f(n), \qquad (1.11)$$

Общее решение уравнения (1.11) представляет собой сумму частного решения (обозначим его через a^p) и общего решения соответствующего ему однородного уравнения (обозначим его через a^g), т. е. $a_n = a^p + a^g$.

Общего метода нахождения частного решения a^p не существует. Частное решение находят исходя из структуры функции $f(n)$. Так, если $f(n) = \alpha \cdot u^n$, то $a^p = \beta \cdot u^n$. Если $f(n)$ – многочлен степени k, то и частное решение – многочлен степени k. В качестве примера исследуем структуру частного решения уравнения (1.11) для случая, когда $f(n) = c$, где c – некоторая константа. Если $\sum \alpha_i \neq 1$, то существует решение $a_n^p = d$, где $d = c/(1 - \sum \alpha_i)$. Если же $\sum \alpha_i = 1$, то существует решение $a_n^p = dn$, где $d = c/\sum i\alpha_i$ и $\sum i\alpha_i \neq 0$.

Производящие функции

Для решения рекуррентных соотношений и выполнения комбинаторных подсчетов в ряде случаев удобно использовать аппарат производящих функций. Последовательности $\{a_n\}$ поставим в соответствие формальный ряд

$$G(z) = \sum_{n=0}^{\infty} a_n z^n. \qquad (1.12)$$

Определение 1.4. Ряд $G(z)$, имеющий вид (1.12), называется *производящей функцией* для данной последовательности $\{a_n\}$.

Соответствие между некоторой последовательность $\{a_n\}$ и рядом $G(z)$ обычно выражается словами «последовательность генерируется производящей функцией», «производящая функция генерирует (порождает, производит) последовательность», или математическими формулами вида:

$$(a_0, a_1, a_2, \dots) \Leftrightarrow (a_0 + a_1 z + a_2 z^2 + \cdots).$$

Сумма (1.12) включает все $n \geq 0$, но иногда оказывается полезным расширение диапазона суммирования на все целые n. Этого можно добиться, если положить $a_{-1} = a_{-2} = \cdots = 0$ для всех $n < 0$. В таких случаях можно по-прежнему говорить о последовательности $\{a_n\} = a_0, a_1, a_2, \dots$, считая, что a_n не существует для отрицательных n.

Перед тем, как рассмотреть операции над производящими функциями, отметим, что производящая функция, во-первых, есть символьная конструкция. Это значит, что вместо переменной z может быть использован произвольный объект, для которого определены операции сложения и умножения. Во-вторых, вопрос о сходимости или расходимости ряда (1.12) рассматриваться не будет, что является важным аспектом изучения числовых рядов. В-третьих, вопрос о вычислении данного ряда для конкретного значения z также опускается. Единственное, что необходимо при работе с производящими функциями (и в целом с бесконечными суммами), это понимание того, что объект z не обязательно суть число, хотя существуют ситуации, когда придавать ему свойства числа будет необходимо.

Рассмотрим основные операции над производящими функциями:

1. Сложение рядов: $G(z) = \sum_n a_n z^n$ и $H(z) = \sum_n b_n z^n$

$$G(z) + H(z) = \sum_n (a_n + b_n) z^n.$$

2. Умножение на число: $\alpha G(z) = \sum_n \alpha \cdot a_n z^n$.

3. Произведение рядов:

$$G(z) \cdot H(z) = \sum_n c_n z^n, c_n = \sum_{i=0}^{n} a_i b_{n-i}.$$

4. Производная:

$$G'(z) = \sum_n (n+1) a_{n+1} z^n \Leftrightarrow z G'(z) = \sum_n n a_n z^n.$$

В качестве примера использования операций над производящими функциями, получим производящую функцию для последовательности $1,1,1, \dots$. В этом случае $G(z) = 1 + z + z^2 + \cdots = 1 + z(1 + z + \cdots) = 1 + zG(z)$, откуда $G(z) = \frac{1}{1-z}$. В качестве следующего примера рассмотрим более общую последовательность $1, \alpha, \alpha^2, \dots$. Для нее производящая функция есть $G(z) = \frac{1}{1-\alpha z}$, так как $G(z) = 1 + \alpha z + \alpha^2 z^2 + \cdots = 1 + \alpha z(1 + \alpha z + \cdots) = 1 + \alpha z G(z)$.

Рассмотрим алгоритм, который позволяет найти решение рекуррентного соотношения путем использования производящих функций. Пусть нам задана последовательность $\{a_n\}$. Тогда необходимо:

1. Записать уравнение, выражающее a_n через другие элементы последовательности. Полученное уравнение должно оставаться справедливым для всех целых n с учетом того, что $a_{-1} = a_{-2} = \cdots = 0$.

2. Умножить обе части уравнения на z^n и просуммировать по всем n. В левой части получится сумма $\sum_n a_n z^n = G(z)$. Правую часть следует преобразовать так, чтобы она превратилась в другое выражение, включающее $G(z)$.

3. Решить полученное уравнение относительно $G(z)$, тем самым получив замкнутое выражение для $G(z)$.

4. Разложить $G(z)$ в степенной ряд и взять коэффициент при z^n, который является замкнутым видом для a_n.

В качестве примера рассмотрим последовательность Фибоначчи. Имеем

$$f_n = \begin{cases} 0, & n < 0, \\ 1, & n = 0, \\ f_{n-1} + f_{n-2}, & n > 0. \end{cases}$$

Шаг 1 требует записать заданное уравнение в виде единственного уравнения без ветвлений. Таким образом

$$f_n = f_{n-1} + f_{n-2} + [n = 0]$$

Шаг 2 предлагает преобразовать уравнение для f_n в уравнение для $G(z)$. В этом случае последовательность преобразований будет иметь вид:

$$G(z) = \sum_n f_n z^n = \sum_n f_{n-1} z^n + \sum_n f_{n-2} z^n + \sum_n [n = 0] z^n =$$
$$= \sum_n f_n z^{n+1} + \sum_n f_n z^{n+2} + z = zG(z) + z^2 G(z) + 1,$$

откуда следует, что $G(z) = \frac{1}{1-z-z^2}$.

Разложим полученную дробь на сумму двух дробей, т. е. $G(z) = \frac{A}{1-\alpha z} + \frac{B}{1-\beta z}$, где A, B, α, β – константы. С учетом этого замкнутый вид для $G(z)$записываем так:

$$G(z) = A \sum_{n \geq 0} (\alpha z)^n + B \sum_{n \geq 0} (\beta z)^n = \sum_{n \geq 0} (A \cdot \alpha^n + B \cdot \beta^n) z^n. \qquad (1.13)$$

Непосредственно разложение на сумму двух дробей можно получить, воспользовавшись методом «неопределенных коэффициентов». Опуская арифметические выкладки, получим

$$A = +1/\sqrt{5}\,, B = -1/\sqrt{5}\,,$$
$$\alpha = \left(1 + \sqrt{5}\right)/2, \beta = (1 - \sqrt{5})/2.$$

Следовательно, замкнутый вид для функции $G(z)$ можно записать как

$$G(z) = \sum_{n\geq 0}(\frac{1}{\sqrt{5}}(\varphi^n - (1-\varphi)^n)z^n, \varphi = \frac{1+\sqrt{5}}{2}.$$

Из полученного соотношения делаем вывод о том, что замкнутый вид для последовательности Фибоначчи выглядит следующим образом:

$$f_n = \frac{1}{\sqrt{5}}(\varphi^n - (1-\varphi)^n), \varphi = \frac{1+\sqrt{5}}{2}.$$

Основная теорема о рекуррентных соотношениях

Пусть задано рекуррентное соотношение следующего вида:

$$\begin{cases} T(1) = c, \\ T(n) = a \cdot T(n/b) + d(n),\ n > 1, \end{cases} \tag{1.14}$$

где a, b, c – некоторые положительные константы, $d(n)$ – заданная функция, которую принято называть *управляющей функцией*.

Отметим, что соотношение (1.14) можно применить только тогда, когда n является целой степенью числа b. Но если функция $T(n)$ достаточно гладкая, то полученное решение можно будет распространить на другие значения n.

Воспользуемся методом итераций для получения замкнутого вида рекуррентного соотношения (1.14). Для этого запишем несколько значений искомой функции $T(n)$:

$T(1) = c,$

$T(b) = aT(1) + d(b) = ac + d(b),$

$T(b^2) = aT(b) + d(b^2) = a^2c + ad(b) + d(b^2),$

…

$$T(b^k) = a^k c + a^{k-1}d(b) + a^{k-2}d(b^2) + \cdots + d(b^k) = a^k c + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right).$$

В последнем равенстве первое и второе слагаемое называются *однородным* и *частным* решениями рекуррентного соотношения (1.14).

Для определенного класса управляющих функций можно найти точное решение рекуррентного соотношения (1.14).

Определение 1.5. Функция $d(n)$ целочисленного аргумента является *мультипликативной*, если для всех положительных целых чисел m и n справедливо $d(mn) = d(m)d(n)$. В частности, мультипликативной является функция $d(n) = n^p$.

Пусть в выражении для $T(b^k)$ управляющая функция является мультипликативной. Тогда для частного решения можно записать

$$\sum_{j=0}^{k-1} a^j d(b^{k-j}) = \sum_{j=0}^{k-1} a^j \left(d(b)\right)^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j = \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}.$$

Последнее выражение имеет смысл, если $a \neq d(b)$. В случае же равенства $a = d(b)$ имеем

$$d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j = k \cdot d(b)^k = k \cdot a^k.$$

Выполнив замену $n = b^k$, получим следующую теорему.

Теорема 2. Решением рекуррентного соотношения (1.11), где $d(n)$ – мультипликативная функция, является

$$T(n) = \begin{cases} c\, n^{\log_b a} + \dfrac{n^{\log_b a} - n^{\log_b d(b)}}{\dfrac{a}{d(b)} - 1}, & a \neq d(b) \\ c\, n^{\log_b a} + \log_b n \cdot n^{\log_b a}, & a = d(b). \end{cases} \qquad (1.15)$$

Теорема 2 позволяет находить точные решения, однако формула решения выглядит достаточно громоздко. Попробуем несколько «огрубить» результаты теоремы. Заметим, что имеют место следующие три ситуации:

1. Если $a > d(b)$, то порядок частного решения будет $O(n^{\log_b a})$. В этом случае однородное и частное решения имеют одинаковый порядок роста, не зависящий от функции $d(n)$.

2. Если $a < d(b)$, то порядок частного решения будет $O(n^{\log_b d(b)})$. В этом случае частное решение превышает однородное. Когда $d(n) = n^p$, порядок частного решения будет $O(n^p)$.

3. Если $a = d(b)$, то порядок частного решения превосходит порядок однородного на $\log_b n$. Когда $d(n) = n^p$, порядок частного решения равен $O(n^p \log n)$.

Предположим, что управляющая функция в рекуррентном соотношении (1.14) задана не точно, а в виде $O(n^p)$. Так как $O((nm)^p) = O(n^p m^p)$, то при анализе рекуррентного соотношения можно использовать технику, применявшуюся при выводе теоремы 2.

Теорема 3[*]. Пусть дано рекуррентное соотношение вида

$$T(n) = aT(n/b) + O(n^p).$$

Тогда

$$T(n) = \begin{cases} O(n^p), & p > \log_b a, \\ O(n^p \log_b n), & p = \log_b a, \\ O\left(n^{\log_b a}\right), & p < \log_b a. \end{cases}$$

[*] Более строгий вариант этой теоремы был сформулирован и доказан Дж. Бентли, Д. Хакен и Дж. Саксом в 1980 году.

Приведем некоторые другие рекуррентные соотношения, возникающие при анализе рекурсивных алгоритмов. Если рекурсия вызывает аддитивное уменьшение размерности задачи, то используется соотношение

$$\begin{cases} T(0) = c, \\ T(n) = aT(n-b) + d(n),\ n \geq 1. \end{cases} \tag{1.16}$$

Иногда рекурсия ведет к «квадратичному» уменьшению сложности. В этом случае возникает соотношение

$$\begin{cases} T(2) = c, \\ T(n) = a\sqrt{n}T\left(\sqrt{n}\right) + d(n),\ n \geq 3. \end{cases} \tag{1.17}$$

Для решения приведенных соотношений (1.16, 1.17) в некоторых частных случаях могут быть применены соображения, использовавшиеся при решении рекуррентного соотношения (1.14).

1.3. Задачи для самостоятельного решения

1. Расположить следующие функции в порядке степени роста:

а) n; б) $\sqrt{n}$; в) $\log n$; г) $\log\log n$; д) $\log^2 n$; е) $\frac{n}{\log n}$; ж) $\sqrt{n}\log^2 n$; з) $\left(\frac{1}{3}\right)^n$; и) 8.

2. Требуется оценить время выполнения следующей программы:

Процедура 1.3.1. Pascal Triangle

```
01:  Pascal Triangle Impl(n)
02:        C[0, 0] = 1
03:        for i = 1 to n do
04:              C[i, 0] = 1
05:              for j = 1 to i do
06:                    C[i, j] = C[i - 1, j] + C[i - 1, j - 1]
```

3. Требуется оценить время выполнения следующей программы:

Процедура 1.3.2. GCD

```
01:  GCD Impl(a, b)
02:        while (a ≠ 0) and (b ≠ 0) do
03:              if a > b then
04:                    a = a mod b
05:              else
06:                    b = b mod a
07:        return (a + b)
```

4. Решить рекуррентные соотношения:

а) $\begin{cases} a_0 = 1, a_1 = 8, \\ a_n = a_{n-1} + 6a_{n-2}; \end{cases}$ б) $\begin{cases} a_0 = 2, a_1 = 5, \\ a_n = 7a_{n-1} - 12a_{n-2}; \end{cases}$ в) $\begin{cases} a_0 = 1, a_1 = 0, \\ a_n = 4a_{n-1} - 4a_{n-2}; \end{cases}$

г) $\begin{cases} a_0 = 1, \\ a_n = a_{n-1} + n; \end{cases}$ д) $\begin{cases} a_0 = 4, a_1 = 5, \\ a_n = 5a_{n-1} - 6a_{n-2} + 6 \cdot 3^n; \end{cases}$ е) $\begin{cases} a_0 = 2, \\ a_n = 3a_{n-1} + 2^n. \end{cases}$

5. Используя соответствующие рекуррентные уравнения, найти суммы:

а) $1^2+2^2+\ldots+n^2$; б) $1^3+2^3+\ldots+n^3$; в) $k^1+k^2+\ldots+k^n$;

г) $1+4+\ldots+(3n-2)$; д) $1\cdot 2+\ldots+n\cdot(n+1)$; е) $1^2\cdot 2+\ldots+n^2\cdot(n+1)$.

6. Найдите замкнутый вид для последовательности Каталана, описываемой рекуррентным соотношением вида

$$\begin{cases} c_0 = 1, \\ c_n = c_0 c_{n-1} + c_1 c_{n-2} + \cdots + c_{n-1} c_0, \; n \geq 1. \end{cases}$$

7. Возвести заданное число A в натуральную степень n за как можно меньшее количество умножений.

8. Определить количество способов, которыми можно уплатить N рублей купюрами различного достоинства $n_1, n_2, \ldots, n_m$, беря не более одной купюры каждого достоинства.

9. Подсчитать количество разбиений выпуклого n-угольника ($n \geq 3$) на треугольники диагоналями, не пересекающимися внутри многоугольника.

10. Число из $2n$ цифр называется *счастливым билетом*, если сумма первых n цифр равна сумме последних n цифр. Найти количество счастливых билетов при заданном значении n.

11. Рассматриваются цепочки, которые состоят из символов (a, b, c, d, e). Четыре пары cd, ce, ed и ee никогда не встречаются в рассматриваемых цепочках, однако любая цепочка, не содержащая этих запрещенных пар (назовем такие цепочки *хорошими*), возможна. Требуется определить число хороших цепочек длины n.

2. Структуры данных

При решении задач, как сугубо теоретических, так и практических, возникает необходимость в работе с данными. Но помимо самих данных, необходимо иметь инструментарий, который позволит их обработать в соответствие со спецификой задачи, в связи с чем возникает потребность в хранении данных внутри памяти ЭВМ. Введем ряд определений.

Определение 2.1. *Ячейкой памяти* будем называть структурную единицу ЭВМ, которая позволяет хранить некоторый фиксированный объем информации, выражающийся целым числом *байт*.

В силу того, что данные могут быть различны по своей природе, введем понятие *тип данных* (определение 2.2).

Определение 2.2. *Типом данных* будем называть некоторое конечное множество значений, каждое из которых требует для своего хранения фиксированное число ячеек памяти.

Типы данных принято разделять на *примитивные* и *составные*. К примитивным типам данных относятся такие, как булевский тип, числовые типы данных, символьные, указатели и другие. Составной тип данных представляет собой логическое объединение конечного числа как примитивных типов данных, так и составных, в том числе и самого себя. Составной тип данных обычно принято называть *структурой* или *записью*. Базируясь на введенных выше определениях, которые можно считать атомарными и присущими любой ЭВМ, дадим следующее определение *структуры данных* (определение 2.3).

Определение 2.3. *Структурой данных* будем называть программную единицу ЭВМ, описывающую способ организации ячеек памяти, хранящих данные одного и того же типа.

Таким образом, структуры данных позволяют хранить необходимые наборы данных в памяти ЭВМ. Неотъемлемой частью любой такой структуры является набор операций, который позволяет эффективно управлять данными, хранящимися в структуре. Таковыми обычно являются операции *чтения* и *записи*, предоставляющие доступ к хранимым данным. Следует отметить, что обе операции обычно реализуются на уровне ЭВМ, поэтому в дальнейшем будем предполагать, что их трудоемкость составляет $O(1)$ и не зависит от того, каким типом данных оперирует рассматриваемая структура.

2.1. Элементарные структуры данных

Под *элементарными структурами данных* будем понимать структуры данных, которые либо предоставляются непосредственно ЭВМ, либо моделируются средствами ЭВМ с использованием типов данных, доступных на уровне ЭВМ. Таким образом, элементарной структурой данных можно назвать любую базовую структуру данных ЭВМ. Ниже будут рассмотрены такие, как *массив* и *связный список*. Необходимо отметить, что их можно считать атомарными структурами данных, так как не предполагается их моделирование путем использования других элементарных структур данных. Хотя такая возможность и

существует с точки зрения ЭВМ, построенная реализация не даст ощутимый выигрыш перед системной структурой данных, поскольку она заведомо ведет к снижению эффективности некоторых из базовых операций, таких, как вставка и удаление элемента.

2.1.1. Массивы

Массив – линейная, однородная, элементарная структура данных, представляющая собой непрерывную упорядоченную последовательность ячеек памяти и предназначенная для хранения данных одного и того же типа.

Доступ к элементам массива осуществляется по *индексу*, либо по *набору индексов*, где под индексом будем понимать некоторое целочисленное значение. В общем виде массив представляет собой многомерный параллелепипед, в котором величины измерений могут быть различны. Массив A с m измерениями будем называть *m-мерным* и обозначать через $A[n_1, n_2, \dots, n_m]$, где n_i – число элементов, которое может быть записано на уровне i $(1 \le i \le m)$.

Таким образом, массив A с m измерениями позволяет хранить внутри себя $k = n_1 \cdot n_2 \cdot \dots \cdot n_m$ элементов с затратами памяти $k \cdot s = O(k)$, где s – количество ячеек памяти, требуемое для хранения одного элемента.

Как было отмечено ранее, любой массив – это непрерывный участок памяти, поэтому многомерный массив технически представляет собой одномерный массив, доступ к элементам которого осуществляется по следующей формуле:

$$(i_1, i_2, \dots, i_m) \Leftrightarrow \\ ((\dots(i_1 \cdot n_2 + \dots) \cdot n_{m-2} + i_{m-2}) \cdot n_{m-1} + i_{m-1}) \cdot n_m + i_m, \qquad (2.1)$$

где i_j – позиция элемента по i-ому измерению, $0 \le i_j < n_j, 1 \le j \le m$.

В дальнейшем изложении будем предполагать, что индексация элементов массива начинается с единицы, если не оговорено иное. Процедура вычисления соответствующего индекса согласно приведенному выражению представлена ниже (процедура 2.1.1):

Процедура 2.1.1. Get Index

```
01:  Get Index Impl(i₁, i₂, ..., iₘ)
02:        index = 0
03:        for j = 1 to m – 1 do
04:              index = (index + iⱼ) * nⱼ₊₁
05:        index = index + iₘ
06:        return index
```

Рассмотрим основные операции, которые позволяют манипулировать данными, хранящимися в массиве, при этом не ограничивая общности, будем считать, что задан одномерный массив $A[n]$, состоящий из n элементов:

1) чтение / запись элемента требуют $O(1)$ времени, так как массив является структурой данных с произвольным доступом;

2) трудоемкость поиска элементов в массиве зависит от того, упорядочен он или нет: в случае упорядоченного массива поиск необходимых данных мож-

но осуществить за $O(\log n)$ операций, иначе потребуется $O(n)$ операций в худшем случае;

3) вставка / удаление элементов в массив требуют $O(n)$ операций в худшем случае, так как изменение структуры массива требует перемещения элементов. Если же вставка и удаление элементов осуществляются только на конце массива, то трудоемкость операций составит $O(1)$. Алгоритм вставки элемента в массив приведен ниже (процедура 2.1.2).

Процедура 2.1.2. Array: Insert

```
01:  Insert Impl(A, index, element)
02:        if index > 0 then
03:              for i = n downto index do
04:                    A[i + 1] = A[i]
05:              A[index] = element
```

В заключение еще раз подчеркнем, что многомерный массив можно неявно рассматривать как массив массивов одинакового размера. В то же самое время существует понятие *рваного массива*[*], использование которого оправдано в том случае, если его элементами являются массивы произвольной длины и следует явно обозначить, что он состоит из массивов. В этом случае рваный массив будем обозначать как $A[\ \ ][\ \ ]\ldots[\ \ ]$. Пример рваного массива приведен ниже (рис. 2.1):

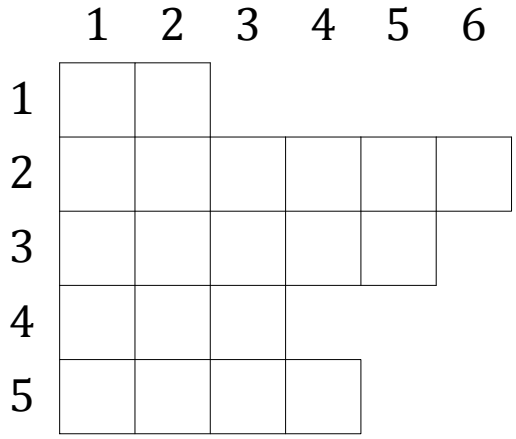

Рис. 2.1. Пример рваного массива $A[5][\ \ ]$

Следует отметить, что большинство современных императивных языков программирования имеют возможности для работы с массивами. Некоторые из языков предоставляют возможности для создания рваных массивов, как одного из типов данных.

Как было отмечено выше, операции вставки и удаления элемента в массив осуществляются в худшем случае за $O(n)$ операций. Поэтому возникает необходимость в разработке структур данных, которые позволят избавиться от данного недостатка и, как следствие, предоставят эффективные методы по включению/исключению элементов. К достоинствам же массива можно отнести экономное расходование памяти, требуемое для хранения данных, а также

[*] В англоязычной литературе используется термин *jagged array*.

быстрое время доступа к элементам массива. Следовательно, применение массивов оправдано в том случае, если обращение к данным происходит по целочисленному индексу и в ряде случаев, не предполагающих модификацию набора данных путем увеличения / уменьшения числа его элементов.

2.1.2. Связные списки

Связным списком будем называть линейную, однородную, элементарную структуру данных, предназначенную для хранения данных одного и того же типа, в которой порядок объектов определяется указателями. В отличие от массива, связный список, как следует из данного определения, хранит элементы не обязательно в последовательных ячейках памяти. Данным свойством и обосновано название «Связный список» рассматриваемой структуры данных, поскольку при помощи указателей ячейки памяти связываются в единое целое.

Рассматривая связный список, будем предполагать, что его элемент описывается составным типом данных, который хранит в себе пользовательские данные некоторого типа и указатель на следующий элемент списка. У связного списка принято различать концевые элементы: *головой* называют начальный элемент списка, *хвостом* – конечный. Таким образом, рекурсивное описание связного списка выглядит следующим образом (процедура 2.1.3):

Процедура 2.1.3. Linked List Type

```
01:  linkedListElement<T> {
02:        item: T
03:        link: linkedList
04:  }
05:  linkedList<T> {
06:        head: linkedListElement<T>
07:        tail: linkedListElement<T>
08:  }
09:  emptyLinkedList<T> = linkedList<T> {
10:        head = null
11:        tail = null
12:  }
```

Помимо данных пользователя и указателей на соседние элементы, элемент списка может хранить и некоторую служебную информацию, продиктованную спецификой задачи (например, время создания элемента списка, адрес элемента в памяти, информацию о владельце и т. д.). На практике же чаще всего используются те либо иные вариации связных списков, которые в большинстве своем обусловлены особенностями итерирования по списку. Основными из них являются односвязные, двусвязные и кольцевые списки, которые будут рассмотрены ниже вместе с базовым набором операций.

Если это не будет приводить к противоречиям, то пустой список далее будем обозначать через символ ∅.

Односвязные списки

Односвязным списком будем называть связный список, в котором каждый из элементов списка содержит указатель непосредственно на следующий элемент того же списка.

Односвязный список является простейшей разновидностью связного списка, поскольку каждый его элемент предполагает хранение минимальной информации для связи элементов списка в единую структуру. Можно видеть, что данное определение является идентичным определению связного списка, поэтому и описание их будет похожим. Детали, касающиеся односвязного списка, приведены ниже (процедура 2.1.4):

Процедура 2.1.4. Single Linked List Type

```
01:  singleLinkedListElement<T> {
02:        item: T
03:        next: singleLinkedList
04:  }
05:  singleLinkedList<T> {
06:        head: singleLinkedListElement<T>
07:        tail: singleLinkedListElement<T>
08:  }
```

Рассмотрим такие основные операции, как вставка элемента в список, удаление элемента из списка и поиск элемента в списке. Перед непосредственным рассмотрением необходимо сказать о ряде вспомогательных процедур: 1) процедура, которая проверяет, является ли список пустым; 2) процедура получения значения первого элемента списка; 3) процедура получения списка, на который указывает начальный элемент текущего списка (процедуры 2.1.5, 2.1.6):

Процедура 2.1.5. Single Linked List: Is Empty

```
01:  Is Empty Impl(L)
02:        if L.head = L.tail = null then
03:              return true
04:        return false
```

Процедура 2.1.6. Single Linked List: First, Next

```
01:  First Impl(L)
02:        if not Is Empty(L) then
03:              return L.head.item
04:        return ∅
05:
06:  Next Impl(L)
07:        if not Is Empty(L) then
08:              return L.head.next
09:        return ∅
```

Рассмотрим также процедуры разделения списка по заданному элементу (2.1.7) и процедуру соединения двух списков в единое целое (2.1.8). Процедура разделения возвращает два списка, а процедура соединения по двум спискам

формирует один. Отметим, что операция разделения будет работать только для непустых списков.

Процедура 2.1.7. Single Linked List: Split

```
01:  Split Impl(L, listElement)
02:        Left  = L
03:        Right = listElement.next
04:        listElement.next = ∅
05:        return Left, Right
```

Процедура 2.1.8. Single Linked List: Join

```
01:  Join Impl(Left, Right)
02:        if Is Empty(Left)  then return Right
03:        if Is Empty(Right) then return Left
04:        with Left do
05:              tail.next = Right
06:              tail = Right.tail
07:        return Left
```

В зависимости от того, на какую позицию помещается вставляемый элемент, процедура его вставки в список может претерпевать некоторые изменения. В связи с этим, как отдельные случаи следует рассмотреть вставку элемента в пустой список (процедура 2.1.9) и вставку элемента в начало, конец либо середину списка (процедура 2.1.10).

Процедура 2.1.9. Single Linked List: Insert Empty

```
01:  Insert Empty Impl(L, dataItem)
02:        if Is Empty(L) then
03:              temp = singleLinkedListElement<T> {
04:                    item = dataItem
05:                    next = ∅
06:              }
07:              L = singleLinkedList<T> {
08:                    head = temp
09:                    tail = temp
10:              }
11:        return L
```

Процедура 2.1.10. Single Linked List: Insert Front/Insert Back, Insert At

```
01:  Insert Front Impl(L, dataItem)
02:        return Join(Insert Empty(∅, dataItem), L)
03:
04:  Insert Back Impl(L, dataItem)
05:        return Join(L, Insert Empty(∅, dataItem))
06:
07:  Insert At Impl(L, listElement, dataItem)
08:        Left, Right = Split(L, listElement)
09:        return Join(Insert Back(Left, dataItem), Right)
```

Операция удаления элемента списка, по сравнению с операцией вставки, представляет некоторые сложности, так как для ее корректного выполнения требуется ссылка на элемент списка, расположенный непосредственно перед

удаляемым элементом. В качестве альтернативы можно воспользоваться операцией удаления элемента, который следует непосредственно после элемента, поступающего на вход процедуре. Таким образом, процедура удаления элемента списка, следующего непосредственно после заданного элемента, приведена в процедуре 2.1.11:

Процедура 2.1.11. Single Linked List: Delete

```
01:  Delete Impl(L, listElement)
02:        Left, Right = Split(L, listElement)
03:        if Is Empty(Right) then
04:              return Left
05:        return Join(Left, Next(Right))
```

Отметим, что все операции, рассмотренные выше, имеют трудоемкость $O(1)$ в худшем случае.

Операция поиска заданного значения в списке требует просмотра всех элементов списка и поэтому имеет трудоемкость $O(n)$ (процедура 2.1.12):

Процедура 2.1.12. Single Linked List: Contains

```
01:  Contains Impl(L, dataItem)
02:        if not Is Empty(L) then
03:              if (First(L) = dataItem) then
04:                    return true
05:              return Contains Impl(Next(L), dataItem)
06:        return false
```

В заключение вышеизложенного, приведем реализацию операций вставки и удаления, которые не будут изменять существующий список, а будут только создавать его копию, если это необходимо. Наличие таких операций позволяет сделать список неизменяемым (процедура 2.1.13):

Процедура 2.1.13. Single Linked List: Immutable Insert and Delete Ops

```
01:  Insert Front Impl(L, dataItem)
02:        if Is Empty(L) then
03:              return Insert Empty(L, dataItem)
04:        return singleLinkedList<T> {
05:              head = singleLinkedListElement<T> {
06:                    item = dataItem
07:                    next = L
08:              }
09:              tail = L.tail
10:        }
11:
12:  Insert Back Impl(L, dataItem)
13:        return Insert At(L, L.tail, dataItem)
14:
15:  Insert At Impl(L, listElement, dataItem)
16:        return IRec(L)
```

```
17:         IRec(L)
18:               if L.head = listElement then
19:                     return Insert Front(
20:                               Insert Front(Next(L), dataItem), First(L))
21:               return Insert Front(IRec(Next(L)), First(L))
22:
23:   Delete Impl(L, listElement)
24:         return DRec(L)
25:
26:         DRec(L)
27:               if L.head = listElement then
28:                     return Next(L)
29:               return Insert Front(DRec(Next(L)), First(L))
```

Отметим, что неизменяемые операции вставки элемента в конец списка, а также удаления последнего элемента из списка, в худшем случае имеют трудоемкость $O(n)$. Несмотря на кажущуюся неэффективность приведенных операций, они находят широкое применение при реализации неизменяемых *абстрактных типов данных*, которые более детально будут рассмотрены ниже.

Двусвязные списки

Двусвязным списком будем называть связный список, в котором каждый из элементов содержит указатель как на следующий элемент списка, так и на предыдущий.

Рекурсивное описание двусвязного списка приведено в процедуре 2.1.14:

Процедура 2.1.14. Double Linked List Type

```
01:   doubleLinkedListElement<T> {
02:         item: T
03:         next: doubleLinkedList<T>
04:         prev: doubleLinkedList<T>
05:   }
06:   doubleLinkedList {
07:         head: doubleLinkedListElement<T>
08:         tail: doubleLinkedListElement<T>
09:   }
```

На практике двусвязные списки находят широкое применение, поскольку они позволяют итерироваться по элементам списка в обоих направлениях. Что же касается операций над двусвязным списком, то они будут претерпевать незначительные изменения по сравнению с операциями, приведенными для односвязного списка. Трудоемкость операций вставки, удаления, поиска составит $O(1)$ в случае изменяемой структуры данных. Для неизменяемой либо персистентной структуры данных «Двусвязный список», выполнение любых n операций над списком будет требовать времени порядка $O(n^2)$, из расчета, что работа начинается с пустого списка. В этом случае можно говорить об амортизированном времени выполнения каждой из операций $O(n)$.

Кольцевые списки

Кольцевым списком будем называть связный список, в котором последний элемент ссылается на первый.

Обычно кольцевые списки реализуются на основе либо односвязных списков, либо двусвязных. В силу того, что список замкнут в кольцо, требуется всего лишь один указатель, который будет определять точку входа в список. Реализации базовых операций для кольцевого списка не приводятся по причине большого сходства с соответствующими операциями для односвязного списка. К основным достоинствам кольцевых списков следует отнести возможность просмотра всех элементов списка, начиная с произвольного, а также быстрый доступ к первому и последнему элементам в случае, если список реализован на базе двусвязного списка.

2.2. Абстрактные типы данных. Общая идеология

Под *абстрактным типом данных (АТД)* будем понимать математическую модель с определенным на ней набором операций и правил поведения, которые определяют ожидаемый результат каждой из операций.

Суть каждой из операций состоит в изменении внутреннего состояния АТД. Поведение же определяет инвариант, который должен сохраняться на протяжении всего жизненного цикла АТД вне зависимости от того, какие операции были выполнены в тот либо иной момент. В некоторых случаях поведение можно рассматривать как политику, которой должна придерживаться каждая из операций, определенная на АТД. В зависимости от правил, которые диктует поведение, абстрактные типы данных принято разделять на *изменяемые* и *неизменяемые*, а также на *персистентные* и *неперсистентные*[*]. Говоря об неизменяемых и персистентных АТД, будем подразумевать, что каждую из операций, определенную на АТД, можно отнести к одному из трех типов – создания, обновления, чтения. Операции создания используются для создания конкретного экземпляра АТД с последующей инициализацией его внутреннего состояния. Под операциями обновления (или изменения) будем понимать такие, выполнение которых ведет к изменению внутреннего состояния АТД. Под операциями чтения – все иные операции.

Под *неизменяемыми* АТД будем понимать такие, внутреннее состояние которых не изменяется на протяжении их жизненного цикла. Таким образом, неизменяемые АТД предполагают инициализацию внутреннего состояния операциями создания, а последующие операции изменения ведут к созданию нового экземпляра того же АТД.

Под *персистентными* АТД будем понимать такие, которые сохраняют себя после каждой операции изменения с последующей возможностью доступа к каждому из сохраненных экземпляров. Различают *частично-персистентные*

[*] В англоязычной литературе соответственно используются термины *immutable* и *persistent*.

и *полностью персистентные* АТД. В частично-персистентных операции обновления разрешены только для последней версии экземпляра АТД, в то время как полностью персистентные предполагают модификацию каждой из сохраненных версий. В большинстве случаев персистентные структуры данных являются неизменяемыми либо частично неизменяемыми.

В противоположность неизменяемым и персистентным, *изменяемые* АТД не преследуют своей целью сохранение внутреннего состояния после каждой операции изменения. Как следствие, изменяемые АТД обладают более высокой производительностью, а также характеризуются более экономным расходованием памяти, требуемой для хранения данных. В то же самое время следует отметить, что неизменяемые и персистентные структуры данных находят широкое применение на практике, в частности в функциональных языках программирования, при работе с параллельными вычислениями, а также в других прикладных отраслях, связанных с информационными технологиями.

Одним из основных понятий абстрактных типов данных является понятие *интерфейса* АТД, определяющего набор операций, доступный для манипулирования внутренним состоянием АТД. Следует понимать, что интерфейс предоставляет ограниченный набор операций для работы с экземплярами АТД, в то время как весь набор операций, определенный на АТД, может быть значительно шире. Основной целью, преследуемой интерфейсом, является предоставление возможности описания внутренних деталей АТД с использованием различных структур данных, а также других АТД. Таким образом, интерфейс определяет набор операций и поведение, в то время как конкретные реализации одного и того же АТД могут отличаться как по времени выполнения операций, так и по затратам памяти.

Рассматривая абстрактные типы данных, следует уделить внимание концепции *контейнера*. Под *контейнером* будем понимать АТД, предназначенный для хранения однотипных наборов данных и предоставляющий методы для манипулирования этими данными. Здесь следует отметить, что существует понятие *коллекции* данных, которое во многом пересекается с понятием контейнера. Чтобы избежать двусмыслицы, в дальнейшем будем придерживаться следующего соглашения: если речь идет о конкретном АТД, то будем говорить о контейнере, если же речь идет о наборе однотипных элементов, то будем говорить о коллекции данных. В этом смысле контейнер, как было отмечено выше, предоставляет возможности по сохранению коллекций данных. Таким образом, большинство АТД, которые будут рассмотрены ниже, суть контейнера, основным предназначением которых является эффективное хранение и манипулирование коллекциями данных.

Контейнеры классифицируют по следующим способам:

1) доступа к данным: то, каким образом контейнер предоставляет доступ к элементам коллекции данных, – является ли он контейнером с последовательным доступом либо с произвольным. Например, в случае обычного массива это может быть индекс, а в случае стека –вершина стека;

2) хранения данных: то, каким образом элементы коллекции данных сохранены в контейнер, – в виде массива, связного списка либо древовидной структуры данных;

3) итерирования: то, каким образом осуществляется просмотр элементов коллекции данных, – в прямом, обратном или ином порядке.

Контейнеры принято разделять на *последовательные* и *ассоциативные*. Под *последовательным* контейнером будем понимать контейнер, чьи элементы хранятся в строго линейном порядке. Под *ассоциативным* – тот, который предназначен для хранения коллекций, состоящих из пар «ключ – значение ключа», где каждый из возможных ключей встречается не более одного раза.

Неотъемлемым компонентом контейнера является *перечислитель* (или *итератор*), основное предназначение которого заключается в предоставлении набора операций, позволяющих осуществить просмотр содержимого контейнера. В зависимости от того, какая структура данных выбрана для сохранения коллекции данных, итераторы могут отличаться друг от друга для одного и того же АТД.

Следует отметить, что АТД, которые по своей природе являются контейнерами, обычно расширяют базовый набор операций контейнера путем добавления функций, специфических для данного АТД. К базовым же операциям следует отнести такие, как определение количества элементов в контейнере; проверка того, является ли контейнер пустым; преобразование контейнера в массив; создание контейнера из массива и др. Дальнейшее изложение не будет акцентировать внимание на упомянутых операциях в силу их простоты как с точки зрения понимания, так и реализации.

В общем случае все абстрактные типы данных можно условно разделить на *линейные* и *нелинейные*. Под *линейными* АТД будем понимать те, которые используют массивы либо связные списки для хранения коллекции данных. Под *нелинейными* АТД будем понимать все остальные (в частности те, которые используют иерархические структуры данных). Следует отметить, что как линейные, так и нелинейные АТД могут быть отнесены к классу либо последовательных, либо ассоциативных контейнеров, в зависимости от того, с какими наборами данных работает АТД.

Ниже будут рассмотрены АТД, получившие наиболее широкое распространение. Общей особенностью, касающейся всех рассматриваемых ниже АТД, является их возможность реализации как посредством массивов, так и посредством связных списков. Способ той либо иной организации данных выбирается исходя из специфики задачи.

2.2.1. Список

Под *списком* будем понимать линейный АТД, предназначенный для хранения упорядоченных наборов данных одного и того же типа. Альтернативным определением списка является определение через последовательный контейнер с произвольным доступом к элементам коллекции данных.

С точки зрения математики список представляет собой *конечную последовательность* однотипных элементов. Основными операциями над списком являются создание списка (как пустого, так и из массива элементов), вставка элемента в список, удаление элемента из списка и поиск элемента в списке. В зависимости от того, какое поведение накладывается на операции над списком, в качестве внутренней структуры может быть выбран массив либо связный список. Так, если требуется, чтобы экземпляр списка был неизменяемым или персистентным, то наиболее удачным будет выбор связного списка. То же самое касается и случаев, если над списком планируется проводить большое число операций изменения. Массивы же для представления списков чаще всего используются в тех случаях, когда заранее известно максимальное количество элементов, которое будет помещено в список (так называемая *емкость* списка), а также в случаях, когда элементы только вставляются в список, и не имеет принципиальной важности позиция вставки элемента.

Ниже рассмотрим реализацию абстрактного типа данных «Список» через массив (процедура 2.2.1). Реализация списка через структуру данных «Связный список» (будь то односвязный либо двусвязный) рассматриваться не будет, поскольку АТД «Список» по сути своей адаптирует методы связных списков для своих целей.

Процедура 2.2.1. ADT List

```
01:  list<T> {
02:        item: T[max items count]
03:        next: integral[max items count]
04:        itemsCount: integral
05:  }
```

К основной процедуре над рассмотренным списком относится вставка элемента в список, описание которой приведено в процедуре 2.2.2:

Процедура 2.2.2. ADT List: Insert

```
01:  Insert Impl(L, dataItem)
02:        with L do
03:              item[itemsCount + 1] = dataItem
04:              next[itemsCount + 1] = itemsCount
05:              itemsCount = itemsCount + 1
```

Процедуру проверки принадлежности заданного значения списку можно реализовать, например, так (процедура 2.2.3):

Процедура 2.2.3. ADT List: Contains

```
01:  Contains Impl(L, dataItem)
02:        with L do
03:              current = itemsCount
04:              while current ≠ 0 do
05:                    if item[current] = dataItem then
06:                          return true
07:                    current = next[current]
08:        return false
```

Еще раз отметим, что приведенная реализация списка через массив лучшим образом подходит для тех случаев, в которых не предполагается осу-

ществлять удаление элементов. Если же операция удаления необходима, то лучше прибегнуть к реализации АТД «Список» через связный список.

В качестве примера реализации списка через массив, рассмотрим следующую задачу. Пусть требуется построить список L, состоящий из n списков, – $L_1, L_2, \dots, L_n$. Будем предполагать, что списки $L_i (1 \le i \le n)$ хранят данные одного и того же типа и что суммарная длина списков L_i не превышает некоторого целого числа C. Также будем предполагать, что в распоряжении имеется $O(C)$ ячеек памяти, т. е. достаточное для того, чтобы сохранить данные списков L_i. Тогда описание списка L будет иметь следующий вид (процедура 2.2.4):

Процедура 2.2.4. ADT List of Lists

```
01:  listOfLists<T> {
02:        item: T[C]
03:        head: integral[C]
04:        next: integral[C]
05:        itemsCount: integral
06:  }
```

В сравнение со списком, описание которого приведено в процедуре 2.2.1, в описание списка L добавлен дополнительный массив, предназначенный для хранения позиций-начал каждого из списков L_i. Если же таковое требуется, то можно хранить и позиции-окончания каждого из списков L_i.

Процедура добавления элемента в список L_i будет выглядеть следующим образом (процедура 2.2.5):

Процедура 2.2.5. ADT List of Lists: Insert

```
01:  Insert Impl(L, i, dataItem)
02:        with L do
03:              item[itemsCount + 1] = dataItem
04:              next[itemsCount + 1] = head[i]
05:              head[i] = itemsCount + 1
06:              itemsCount = itemsCount + 1
```

Если же требуется обработать элементы списка L_i, то это можно сделать при помощи процедуры 2.2.6:

Процедура 2.2.6. ADT List of Lists: Process

```
01:  Process Impl(L, i)
02:        with L do
03:              current = head[i]
04:              while current ≠ 0 do
05:                    {* process current item *}
06:                    current = next[current]
```

В заключение отметим, что списки принято разделять на *динамические* и *статические*. Статические списки обычно не предполагают исключения элементов, поэтому их реализация обычно базируется на массивах. Наиболее же гибкая и эффективная реализация данного АТД базируется на основе двусвязного списка, поскольку все операции, кроме операции поиска, требуют в худшем случае $O(1)$ операций. Если же речь идет о неизменяемом списке, то в худшем случае вставка и удаление элементов будут требовать $O(n)$ операций.

2.2.2. Стек

Стек (или *магазин*) – линейный абстрактный тип данных, доступ к элементам которого организован по принципу LIFO (от англ. last-in-first-out).

В соответствие с принципом LIFO для стека должна быть определена *точка входа* (или *вершина стека*), через которую будет осуществляться вставка и удаление элементов. В частном случае стек можно рассматривать как список, работа с элементами которого происходит только на одном из концов списка.

Эффективные реализации стека базируются как на массивах, так и на связных списках. Рекомендуется использовать реализацию через массив в тех случаях, когда требуется добиться высокой производительности и экономного расходования памяти. В общем же случае наиболее предпочтительным оказывается подход, базирующийся на идее односвязного списка (можно сказать, что структура односвязного списка в этом случае используется неявно). Данный подход представляется наиболее интересным как с точки зрения расширения функционала, поддерживаемого стеком, так и с точки зрения преобразования данного АТД в неизменяемый тип данных. Детали реализации АТД «Стек» приведены в процедуре 2.2.7:

Процедура 2.2.7. ADT Stack

```
01:  stackEntry<T> {
02:        item: T
03:        next: stack
04:  }
05:  stack<T> {
06:        entry: stackEntry<T>
07:  }
```

Процедуры вставки, удаления и получения значения на вершине стека приведены ниже (процедура 2.2.8):

Процедура 2.2.8. ADT Stack: Push, Pop, Top

```
01:  Push Impl(S, dataItem)
02:        return stack<T> {
03:              entry = stackEntry<T> {
04:                    item = dataItem
05:                    next = S
06:              }
07:        }
08:
09:  Pop Impl(S)
10:        if not Is Empty(S) then
11:              return S.entry.next
12:        return ∅
13:
14:  Top Impl(S)
15:        if not Is Empty(S) then
16:              return S.entry.item
```

Можно видеть, что поведение операций, определенных на стеке данным образом, задается следующим набором инвариантов:

$$\mathrm{Top}\big(\mathrm{Push}(S,e)\big) = e,$$
$$\mathrm{Pop}\big(\mathrm{Push}(S,e)\big) = S.$$

Следует отметить, что для стека не определена операция поиска значения. Если же такая операция необходима, то реализовать ее можно с трудоемкостью $O(n)$ следующим образом (процедура 2.2.9):

Процедура 2.2.9. ADT Stack: Contains

```
01:  Contains Impl(S, dataItem)
02:        if not Is Empty(S) then
03:              if Top(S) = dataItem then
04:                    return true
05:              return Contains Impl(Pop(S), dataItem)
06:        return false
```

2.2.3. Очередь

Очередь – линейный абстрактный тип данных, доступ к элементам которого организован по принципу FIFO (от англ. first-in-first-out).

В соответствие с принципом FIFO для очереди должны быть определены две точки входа: *начало* очереди, откуда происходит удаление элементов, и *конец* очереди, куда происходит добавление элементов. В частном случае очередь можно рассматривать как список, вставка элементов в который происходит на одном конце, а удаление элементов – на другом.

Эффективные реализации очереди базируются как на массивах, так и на связных списках. Рекомендуется использовать реализацию через массив тогда, когда требуется добиться высокой производительности и экономного расходования памяти, а также в случаях, когда заранее известно, что операции добавления и удаления элементов встречаются равновероятно (тогда достаточно завести два счетчика, каждый из которых будет перемещаться в начало массива в тот момент, когда достигнута последняя ячейка выделенного участка памяти). Как и для АТД «Стек», здесь наиболее предпочтительным оказывается подход, базирующийся на идее односвязного списка. В силу простоты реализации данный подход рассматриваться не будет. Более интересным представляется моделирование очереди через два стека, как с точки зрения реализации интерфейса одного АТД через другой АТД, так и с точки зрения возможностей, предлагаемых данным подходом. Детали описания очереди через АТД «Стек» приведены в процедуре 2.2.10:

Процедура 2.2.10. ADT Queue

```
01:  queue<T> {
02:        inStack:  stack<T>
03:        outStack: stack<T>
04:  }
```

Процедуры вставки, удаления и получения значения первого элемента очереди представлены в процедуре 2.2.11:

Процедура 2.2.11. ADT Queue: Enqueue, Dequeue, Peek

```
01:  Enqueue Impl(Q, dataItem)
02:        return queue<T> {
03:              inStack  = Push(Q.inStack, dataItem)
04:              outStack = Q.outStack
05:        }
06:
07:  Dequeue Impl(Q)
08:        if not Is Empty(Q) then
09:              Balance(Q)
10:              return queue<T> {
11:                    inStack  = Q.inStack
12:                    outStack = Pop(Q.outStack)
13:              }
14:        return ∅
15:
16:  Peek Impl(Q)
17:        if not Is Empty(Q) then
18:              Balance(Q)
19:              return Top(Q.outStack)
```

Отдельного внимания заслуживает операция балансировки стеков, суть которой заключается в том, чтобы переместить данные из входного стека в выходной (процедура 2.2.12):

Процедура 2.2.12. ADT Queue: Balance

```
01:  Balance Impl(Q)
02:        with Q do
03:              if Is Empty(outStack) then
04:                    while not Is Empty(inStack) do
05:                          outStack = Push(outStack, Top(inStack))
06:                          inStack  = Pop(inStack)
```

В силу того, что элементы, добавленные в очередь, помещаются в каждый из двух стеков ровно один раз, то можно считать, что время, затраченное на выполнения E операций добавления и D операций удаления будет пропорционально $O(E + D)$. Если же очередь реализована через массив либо связный список, то учетное время выполнения каждой из операций составит $O(1)$. Как и для стека, для АТД «Очередь» не определена операция поиска значения. При необходимости такую операцию можно реализовать с трудоемкостью $O(n)$ следующим образом (процедура 2.2.13):

Процедура 2.2.13. ADT Queue: Contains

```
01:  Contains Impl(Q, dataItem)
02:        with Q do
03:              return Contains(inStack, dataItem) or
04:                        Contains(outStack, dataItem)
```

2.2.4. Дек

Дек – линейный абстрактный тип данных, доступ к элементам которого организован по принципу двусторонней очереди[*].

Наиболее тривиальным определением дека является определение через список, в который запрещена вставка элементов в середину либо определение через очередь, где оба конца открыты для полного манипулирования данными. Дек является наиболее гибкой и, как следствие, наиболее сложной структурой данных, предоставляющей широкий доступ к формированию последовательности, в которой будут храниться необходимые наборы данных. В соответствие с определением дека, наиболее подходящей структурой данных для его реализации является двусвязный список, позволяющий производить все операции на деке (кроме операции поиска) с трудоемкостью $O(1)$. Если же требуется эффективная реализация неизменяемого дека, то структура неизменяемого двусвязного списка не подойдет по причине неэффективности операций, выполняемых над ним (напомним, что в этом случае каждая из операций вставки и удаления будет выполняться за $O(n)$). Следовательно, требуется использовать другой подход, отличный от использования связных списков.

Рассмотрим структуру данных, которая позволит эффективно реализовать неизменяемый АТД «Дек». Идея, которая лежит в основе рассматриваемого подхода заключается в том, что любой дек можно представить в виде трех последовательностей элементов: левый элемент, средняя часть дека и правый элемент (процедура 2.2.14):

Процедура 2.2.14. ADT Deque

```
01:  deque<T> {
02:        left:   T
03:        right:  T
04:        middle: deque<T>
05:  }
06:  singleDeque<T>: {
07:        item: T
08:  }
```

Отметим, что как одиночный дек, так и пустой дек подчиняются общему интерфейсу дека, т. е. над этими типами данных допустимы операции, определенные над АТД «Дек». Так как дек является двусторонней очередью, то отдельно стоит рассмотреть операции вставки в начало / конец дека и удаления из начала / конца дека (процедура 2.2.15) [**]:

Процедура 2.2.15. ADT Deque: Enqueue, Dequeue, Peek

```
01:  Empty Deque
02:        Enqueue Left Impl(D, dataItem)
03:              return singleDeque<T> {
04:                    item = dataItem
05:              }
```

[*] Термин «дек» происходит от английского *deque – double-ended queue*.

[**] Процедуры вставки, удаления и извлечения элементов приведены только для левого конца дека. Процедуры для правого конца дека будут аналогичны.

```
06:  Single Deque
07:        Enqueue Left Impl(D, dataItem)
08:              return deque<T> {
09:                    left   = dataItem
10:                    right  = Peek Left(D)
11:                    middle = ∅
12:              }
13:  Deque
14:        Enqueue Left Impl(D, dataItem)
15:              return deque<T> {
16:                    left   = dataItem
17:                    right  = Peek Right(D)
18:                    middle = Enqueue Left(middle, Peek Left(D))
19:              }
20:
21:        Dequeue Left Impl(D)
22:              with D do
23:                    if Is Empty(middle) then
24:                          return singleDeque<T> {
25:                                item = Peek Right(D)
26:                          }
27:                    return deque<T> {
28:                          left   = Peek Left(middle)
29:                          right  = Peek Right(D)
30:                          middle = Dequeue Left(middle)
31:                    }
32:
33:        Peek Left Impl(D)
34:              with D do
35:                    return left
```

Недостатком приведенной реализации дека является наличие рекурсивных вызовов в операциях вставки и удаления элементов. Таким образом, если требуется вставить n элементов в дек, то затраченное время составит $O(n^2)$. Существуют модификации данного подхода, позволяющие улучшить время работы процедур вставки и удаления до $O(\log n)$ и даже до $O(1)$. За деталями можно обратиться к источникам [1, 2, 3].

2.2.5. Дерево. Терминология и варианты реализации

Выше были рассмотрены линейные структуры данных, предназначенные для хранения последовательностей элементов. Так как последовательности задаются отношением линейного порядка, такие структуры данных, как массив и связный список, наиболее подходят для их хранения.

Зачастую возникают задачи, в которых данные связаны более сложными отношениями, поэтому есть необходимость в разработке структур данных, учитывающих особенности связей между рассматриваемыми элементами. Одним из отношений, описывающим порядок, отличный от линейного, является общее отношение частичного порядка.

Будем говорить, что набор данных M образует *иерархию*, если на нем задано отношение частичного порядка, которое не допускает циклических зависимостей (таким образом, последовательность является частным случаем иерархии). Условно иерархию можно представить в виде уровней, на каждом из которых находятся элементы, попарно между собой не принадлежащие заданному отношению и следующие непосредственно за элементами предыдущего уровня, с которыми они образуют отношение. Говоря об иерархии, обычно выделяют элемент, у которого нет предшественников. В соответствие с данным определением описание иерархии может выглядеть следующим образом (процедура 2.2.16):

Процедура 2.2.16. Hierarchy Type

```
01:  hierarchyEntry<T> {
02:        item: T
03:        next: hierarchy<T>[]
04:  }
05:  hierarchy<T> {
06:        entry: hierarchyEntry<T>
07:  }
08:  emptyHierarchy<T> = hierarchy<T>
```

Наиболее удачной математической моделью для представления иерархий является дерево. Неформально *дерево* можно определить как совокупность элементов, называемых *узлами* (или *вершинами*), и отношений, образующих иерархическую структуру узлов. Формально же дерево можно определить несколькими способами.

Определение 2.4. *Дерево* – это связный граф без циклов. Под *корневым деревом* будем понимать ориентированный граф со следующими свойствами:

1) существует в точности одна вершина, входящая степень которой равняется нулю; такую вершину принято называть *корнем дерева*;

2) входящая степень каждой из вершин дерева, отличных от корневой, равняется единице;

3) существует единственный путь от корня к каждой из вершин дерева.

Альтернативным определением является следующее рекурсивное определение дерева.

Определение 2.5. *Дерево* – это математический объект, определенный рекурсивно следующим образом:

1) один узел, называемый *корнем дерева*, является деревом;

2) пусть n – это узел, а $T_1, \dots, T_k$ – непересекающиеся между собой деревья с корнями $n_1, \dots, n_k$ соответственно. Тогда, сделав узел n родителем узлов n_i, получим новое дерево T с корнем n и поддеревьями $T_1, \dots, T_k$. В этом случае, узлы $n_1, \dots, n_k$ называются *дочерними узлами* (или *детьми*) узла n.

Будем говорить, что дерево T является *упорядоченным деревом*, если на сыновьях каждого из узлов дерева заданы отношения порядка. В противном случае дерево T является *неупорядоченным*.

Пусть T – некоторое упорядоченное дерево. Рассмотрим узел n дерева T с детьми $n_1, \dots, n_k$. *Левым сыном* узла n назовем такой узел n_i, для которого не существует узла n_j такого, что $n_j \preccurlyeq n_i$. *Правым братом* узла n_i назовем такой узел n_j, для которого не существует узла n_k такого, что $n_i \preccurlyeq n_k \preccurlyeq n_j$.

Листом назовем узел, у которого нет детей. Альтернативным может служить определение, согласно которому *лист* это узел, детьми которого являются пустые деревья. *Уровнем узла* назовем число, на единицу большее уровня родительского узла при условии, что уровень корневого узла есть 0. Тогда *высотой дерева* назовем наибольший из уровней узлов дерева. Очевидно, что высоту дерева определяют листья данного дерева.

Согласно определению 2.5 для представления иерархического набора данных в памяти ЭВМ можно выбрать следующую структуру данных, схожую со связными списками (процедура 2.2.17):

Процедура 2.2.17. Tree Data Type

```
01:  treeNode<T> {
02:        item: T
03:        children: container<tree<T>>
04:  }
05:  tree<T> {
06:        root: treeNode<T>
07:  }
```

В качестве контейнера для хранения детей данной вершины обычно выбирается последовательный контейнер (например, список) либо одна из линейных структур данных (массив, односвязный список и т. д.).

Предположив, что узлы дерева занумерованы натуральными числами от 1 до n, получим следующий способ хранения информации о дереве в памяти ЭВМ. Рассмотрим массив P, состоящий из n ячеек, изначально заполненный нулями. Тогда, записав в ячейку $P[i]$ родителя узла с номером i, получим весьма компактный способ представления дерева. Массив P принято называть *массивом предков* данного корневого дерева.

Следует отметить, что рассмотренный способ хранения дерева обладает рядом недостатков, одним из которых является отсутствие эффективного способа просмотра узлов дерева, начиная с корня. Для того чтобы избавиться от данного недостатка, дерево можно хранить в виде списков смежности узлов, где под *списком смежности* узла v будем понимать линейный список узлов, непосредственно связанных с узлом v. Тогда процедура добавления связи между узлами u и v может выглядеть следующим образом (процедура 2.2.18):

Процедура 2.2.18. Tree Data Type: Adjacency List, Insert

```
01:  Insert Impl(T, u, v)
02:        Insert Edge Impl(T, u, v)
03:        Insert Edge Impl(T, v, u)
04:  Insert Edge Impl(T, u, v)
05:        with T do
06:              Insert(adjacencyLists[u], v)
```

Исходя из этого, процедура просмотра узлов дерева из заданной вершины (необязательно корня) не представляет сложностей (процедура 2.2.19):

Процедура 2.2.19. Tree Data Type: Adjacency List, Process

```
01:  Process(T, root)
02:        Process Impl(T, root, ∅)
03:
04:  Process Impl(T, currNode, prevNode)
05:        {* process current node *}
06:        with T do
07:              foreach nextNode in adjacencyLists[currNode] do
08:                    if nextNode ≠ prevNode then
09:                          Process Impl(T, nextNode, currNode)
```

Описанный выше подход для хранения дерева обладает рядом достоинств, основным из которых является эффективный способ доступа к узлам, непосредственно связанным с заданным. Таким образом, трудоемкость процедуры (2.2.19) составляет $O(n)$, где n – число вершин в дереве. В то же самое время списки смежности не позволяют эффективно определять, связаны ли две вершины непосредственно между собой (для этого в худшем случае может потребоваться $O(n)$ операций). Для эффективного выполнения такого рода запросов можно воспользоваться *матрицами смежности* размерностью $n \times n$, где в ячейке (i, j) матрицы будет записана 1, если вершины i и j связаны между собой и 0 в противном случае. Тогда запрос о наличии связи между вершинами можно выполнять за $O(1)$. Недостатком данного подхода является большое число операций, пропорциональное $O(n^2)$, требуемое для обхода дерева.

Бинарным деревом назовем упорядоченное дерево, у которого каждый из узлов содержит не более двух детей. Таким образом, у бинарного дерева обычно принято выделять левое поддерево и правое поддерево или, с точки зрения узлов, левого сына и правого сына. Согласно данному определению, описание бинарного дерева может выглядеть следующим образом (процедура 2.2.20):

Процедура 2.2.20. Binary Tree Data Type

```
01:  binaryTreeNode<T> {
02:        item:  T
03:        left:  binaryTree<T>
04:        right: binaryTree<T>
05:  }
06:  binaryTree<T> {
07:        root: binaryTreeNode<T>
08:  }
```

Как и для общих деревьев, массивы весьма удачно подходят для хранения бинарных деревьев с фиксированным числом вершин. Пусть, как и ранее, узлы дерева занумерованы натуральными числами от 1 до n. Рассмотрим массив B, состоящий из n ячеек. Информацию о корне дерева запишем в ячейку $B[1]$. Пусть информация о некотором узле k записана в ячейку $B[i]$. Тогда информация о левом ребенке узла k будет записана в ячейку $B[2 * i]$, а информация о

правом ребенке – в ячейку $B[2 * i + 1]$. В этом случае родителем узла, хранящегося в ячейке $B[i]$, будет узел $B[\lfloor i/2 \rfloor]$. Тогда процедура обработки узлов дерева может выглядеть следующим образом (процедура 2.2.21):

Процедура 2.2.21. Binary Tree Data Type: Process

```
01:  Process Impl(B)
02:        return Process(B, 1)
03:
04:        Process Impl(B, root)
05:              {* process root *}
06:              for i = 0 to 1 do
07:                    if B[2 * root + i] ≠ ∅ then
08:                          Process(B, 2 * root + i)
```

Одним из свойств бинарных деревьев, которое позволило получить им широкое распространение, является свойство, связывающее высоту бинарного дерева h с количеством узлов дерева n:

$$\log_2(n+1) - 1 \le h \le n - 1.$$

Будем говорить, что бинарное дерево является *сбалансированным*, если его высота h есть $O(\log_2 n)$, уровни от 0 до $h - 1$ заполнены полностью, а уровень h заполнен элементами слева направо.

Определение 2.6. *M-арным* назовем дерево, у которого каждый из узлов, кроме листьев, содержит не более M детей. Если же это количество равно M, то *M-арное* дерево будем называть полным.

Специальным случаем *M*-арного дерева является бинарное дерево, у которого $M = 2$. Несложно получить формулу для количества узлов полного *M*-арного дерева высотой h:

$$n = \frac{M^{h+1} - 1}{M - 1}.$$

Обход дерева

Существует несколько способов обхода упорядоченного дерева, т. е. способов просмотра его узлов. К основным из них относятся *прямой*, *обратный* и *симметричный* обход. Для того чтобы описать правила, которых придерживается каждый из способов просмотра узлов дерева, будем считать, что задано дерево T с корнем n и поддеревьями $T_1, \dots, T_k$. Тогда:

1) при *прямом обходе* сначала посещается корень n дерева T, далее посещаются в прямом порядке поддеревья $T_1, \dots, T_k$;

2) при *обратном обходе* сначала посещаются в обратном порядке поддеревья $T_1, \dots, T_k$, последним посещается корень n дерева T;

3) при *симметричном обходе* сначала посещается в симметричном порядке поддерево T_1, затем посещается корень n дерева T, далее посещаются в симметричном порядке узлы поддеревьев $T_2, \dots, T_k$.

Процедуры, моделирующие каждый из описанных способов обхода дерева, приведены ниже (процедура 2.2.22):

Процедура 2.2.22. Tree Data Type: PreOrder, PostOrder, InOrder

```
01:  PreOrder Impl(T)
02:        if not Is Empty(T) then
03:              process(Root(T))
04:              for i = 1 to children count do
05:                    PreOrder(T_i)
06:
07:  PostOrder Impl(T)
08:        if not Is Empty(T) then
09:              for i = 1 to children count do
10:                    PostOrder(T_i)
11:              process(Root(T))
12:
13:  InOrder Impl(T)
14:        if not Is Empty(T) then
15:              InOrder(T_1)
16:              process(Root(T))
17:              for i = 2 to children count do
18:                    InOrder(T_i)
```

Следует отметить, что каждый из приведенных обходов дерева может быть реализован в виде нерекурсивной процедуры путем использования АТД «Стек». Что же касается использования этих процедур, то каждая из них получила свое конкретное применение в областях информатики (в частности, в теории игр, динамическом программировании, в теории автоматов и языков и так далее). Трудоемкость каждой из процедур есть $O(|T|)$.

Одним из обходов дерева, который отличается от описанных выше, является обход *в ширину*. В данном случае узлы дерева просматриваются по уровням. Для эффективной реализации обхода в ширину используется АТД «Очередь» (процедура 2.2.23):

Процедура 2.2.23. Tree Data Type: Breadth First Search

```
01:  BFS(T)
02:        queue q = Enqueue(∅, Root(T))
03:        while not Is Empty(q) do
04:              node = Peek(q), q = Dequeue(q)
05:              process(node)
06:              for i = 1 to children count do
07:                    q = Enqueue(q, Root(T_i))
```

Так как каждый из узлов дерева помещается в очередь один раз (в силу свойств дерева), то трудоемкость приведенной процедуры составит $O(|T|)$, где $|T|$ – число узлов в дереве T.

Помимо обходов, которые были рассмотрены выше, существует ряд других алгоритмов, которые позволяют эффективно просматривать узлы дерева, причем не только деревьев, чья структура известна заранее, а также и деревьев, которые динамически расширяются в процессе обхода. К одной из таких техник относится техника итеративного погружения, получившая широкое распространение в области искусственного интеллекта.

Преобразование произвольного дерева в бинарное дерево

Зачастую работа с произвольными деревьями затруднительна по причине того, что требуется хранить список детей для каждого из узлов дерева. В то же самое время существует широкий класс задач на бинарных деревьях, для которых разработаны эффективные алгоритмы их решения. Эти алгоритмы не всегда могут быть приспособлены для решения обобщенных версий задач на произвольных деревьях. В связи с этим возникает потребность в разработке алгоритмов, которые позволят свести задачу на произвольных деревьях к задачам на бинарных деревьях. Одним из них является алгоритм преобразования произвольного дерева в соответствующее ему бинарное.

Между произвольными упорядоченными деревьями и бинарными деревьями существует взаимно-однозначное соответствие. Для того чтобы построить такое соответствие достаточно произвольное дерево T рассмотреть с точки зрения «левый сын – правый брат».

Определение 2.7. Пусть T – произвольное дерево с узлами $\{n_1, \dots, n_k\}$, а B – некоторое бинарное дерево с узлами $\{n'_1, \dots, n'_k\}$. Тогда дерево B является взаимно-однозначным *соответствием* дерева T, если:

1) узел n_i дерева T соответствует узлу n'_i дерева B, причем метки узлов n_i и n'_i являются одинаковыми;

2) узел n_i есть корень дерева T, то узел n'_i – корень дерева B;

3) узел n_j есть самый левый сын узла n_i, то n'_j – левый сын узла n'_i; если у узла n_i нет детей, то у узла n'_i нет левого сына;

4) узел n_j есть правый брат узла n_i, то n'_j – правый сын узла n'_i.

Пример произвольного дерева и соответствующего ему бинарного дерева приведены на рис. 2.2.

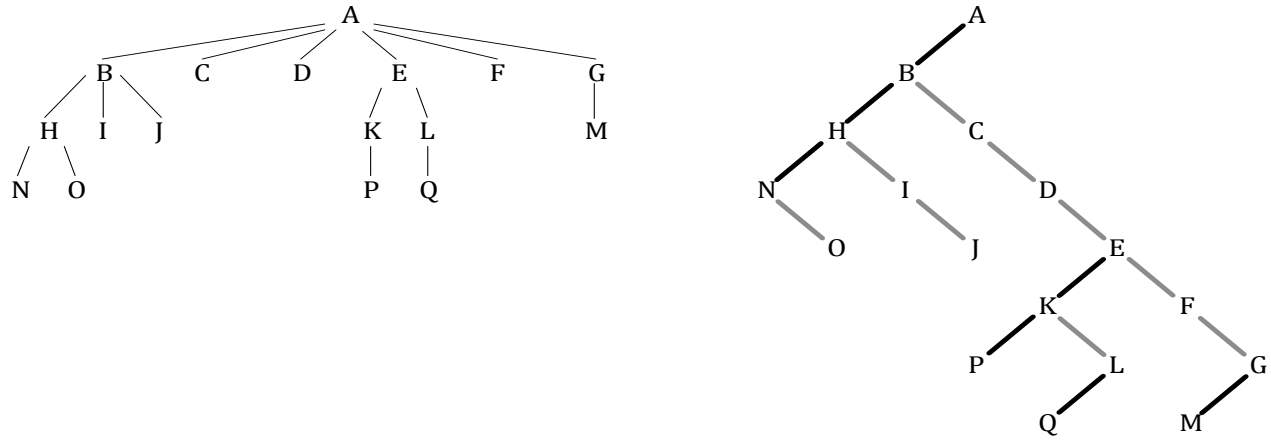

Рис. 2.2. Пример преобразования произвольного дерева в бинарное дерево

Если предположить, что некоторое произвольное дерево T описано при помощи типа данных (2.2.17), то алгоритм преобразования дерева T в бинарное дерево может выглядеть следующим образом (процедура 2.2.24):

```
Процедура 2.2.24. Tree Data Type: Build Binary Tree by Arbitrary Tree
01:  Build Binary Tree By(T)
02:         return Build Impl(T, ∅)
```

```
03:  Build Impl(T, rightTree)
04:        leftTree = ∅
05:        for i = children count downto 1 do
06:              leftTree = Build Impl(children[i], leftTree)
07:        return binaryTree {
08:              root = binaryTreeNode {
09:                    item  = Item(Root(T))
10:                    left  = leftTree
11:                    right = rightTree
12:              }
13:        }
```

Следует отметить, что существуют и другие алгоритмы преобразования произвольных деревьев в бинарные деревья, большинство из которых базируется на определение 2.7. Изменения же обычно продиктованы спецификой задачи (например, очень часто бывает полезным введение дополнительных узлов с тем, чтобы избежать связей между братьями).

Наименьший общий предок двух вершин в дереве

Одной из наиболее часто возникающих задач на деревьях является задача определения наименьшего общего предка для двух заданных вершин.

Определение 2.8. *Общим предком* узлов u и v в корневом дереве T будем называть такой узел p, который лежит на пути от корня дерева T как в узел u, так и в узел v. *Наименьшим общим предком* узлов u и v в корневом дереве T будем называть такого общего предка p узлов u и v, который наиболее удален от корня дерева T[*].

Наиболее простым алгоритмом нахождения LCA двух вершин в дереве является алгоритм с трудоемкостью $\langle O(n), O(h)\rangle$. В этом случае алгоритму предварительно требуется обойти дерево за $O(n)$ и собрать информацию о глубине узлов (т. е. об уровнях, на которых узлы размещены) и об их родителях. Непосредственно фаза запроса LCA будет иметь трудоемкость $O(h)$, где h – высота дерева (процедура 2.2.25):

Процедура 2.2.25: Tree: LCA

```
01:  LCA Query Impl(v1, v2)
02:        h1 = Depth(v1)
03:        h2 = Depth(v2)
04:        while h1 ≠ h2 do
05:              if h1 > h2 then
06:                    v1 = Parent(v1)
07:                    h1 = h1 - 1
08:              else
09:                    v2 = Parent(v2)
10:                    h2 = h2 - 1
11:        while v1 ≠ v2 do
12:              v1 = Parent(v1)
13:              v2 = Parent(v2)
14:        return v1
```

[*] В англоязычной литературе используется термин LCA от *least common ancestor*.

Внимательно проанализировав приведенный алгоритм, заметим, что подъем из вершин происходит с шагом 1. Если попытаться на каждом шаге подниматься на большее число уровней, то можно улучшить общее время выполнения алгоритма.

Рассмотрим корневое дерево T, вершины которого занумерованы числами от 1 до n. Обозначим через $P[i,k]$ $(1 \le i \le n, 0 \le k \le \lfloor \log_2 n \rfloor)$ предка вершины i, который удален от нее на 2^k уровней (в случае, если уровень вершины i меньше, чем 2^k, то соответствующее значение $P[i,k]$ будет равно корню дерева T). Тогда построение массива P можно выполнить за $O(n \log n)$ шагов, используя следующее рекуррентное соотношение:

$$\begin{cases} P[i,k] = parent(i),\ k = 0, \\ P[i,k] = P[P[i,k-1], k-1],\ k > 0. \end{cases} \tag{2.2}$$

Графическое пояснение приведенного рекуррентного соотношения приведено на рис. 2.3:

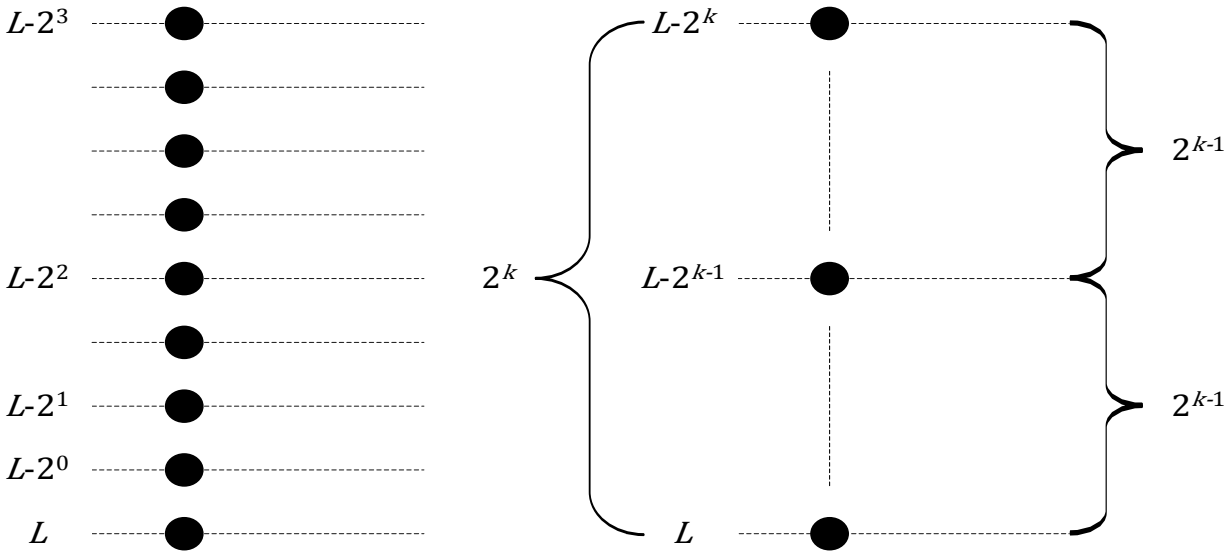

Рис. 2.3. К рекуррентному соотношению 2.1

Процедура заполнения массива P требуемыми значениями приведена ниже (2.2.26):

```
Процедура 2.2.26: Tree: LCA, PreProcess
01:  LCA PreProcess Impl(T)
02:        for i = 1 to n do
03:              P[i, 0] = Parent(i)
04:        for k = 1 to ⌊log2 n⌋ do
05:              for i = 1 to n do
06:                    P[i, k] = P[P[i, k – 1], k – 1]
```

Для того чтобы воспользоваться информацией, хранящейся в массиве P, необходимо иметь процедуру, которая будет для двух вершин проверять, является ли одна из них предком (необязательно непосредственным) другой. Существуют методы, базирующиеся на поиске в глубину, которые позволяют осуще-

ствить требуемую проверку за $O(1)$ [4]. Тогда запрос на LCA двух вершин можно реализовать за $O(\log n)$ шагов следующим образом (процедура 2.2.27):

Процедура 2.2.27: Tree: LCA

```
01:  LCA Query Impl(v1, v2)
02:        if Is Ancestor(v1, v2) then return v1
03:        if Is Ancestor(v2, v1) then return v2
04:        for k = ⌊log2 n⌋ downto 0 do
05:              if not Is Ancestor(P[v1, k], v2) then
06:                    v1 = P[v1, k]
07:        return Parent(v1)
```

Таким образом, в начале алгоритм находит наименее удаленный от корня узел, который не является предком вершины v_2, а затем возвращает родителя найденного узла в качестве LCA узлов v_1 и v_2.Трудоемкость рассмотренного выше алгоритма составляет $\langle O(n \log n), O(\log n) \rangle$. Рассмотренный выше способ нахождения LCA двух вершин носит название *техники двоичного подъема*. С другой стороны, данную технику можно рассматривать с точки зрения более общей *техники масштабирования*. Следует отметить, что существуют и другие техники нахождения LCA двух узлов в заданном корневом дереве, основные из которых рассмотрены в источнике [5].

АТД «Дерево»

Выше были рассмотрены базовые структуры данных для представления деревьев как произвольных, так и бинарных, в памяти ЭВМ. Помимо структур данных следует выделить понятие абстрактного типа данных «Дерево», на котором определен ряд операций, основными из которых являются добавление и удаление узла в дерево, поиск заданного значения в дереве, определение отношений между узлами (например, являются ли узлы братьями, является ли узел предком либо потомком другого узла и т. д.), а также операции по созданию дерева по заданной коллекции данных. Для работы с АТД «Дерево» может быть выбрана одна из структур данных, рассмотренных выше. Нередко подходы могут быть скомбинированы (процедура 2.2.28):

Процедура 2.2.28: ADT Tree

```
01:  treeDataType {
02:        parent: treeNode<T>[max nodes count]
03:        tree: tree<T>
04:  }
```

Отдельно стоит выделить АТД «Бинарное дерево» как наиболее распространенный среди всех остальных типов деревьев. Одной из областей применения АТД «Бинарное дерево» является область его использования при реализации бинарных деревьев поиска (будут рассмотрены в разд. 4.2), а также других более сложных АТД, таких, как «Множество».

Независимо от того, какой из типов деревьев используется для хранения коллекции данных, будем предполагать, что АТД «Дерево» принадлежит к классу нелинейных контейнеров. При этом хранимые элементы могут быть

упорядочены и записаны в линейный контейнер путем использования операции обхода дерева. Если же АТД «Дерево» используется для хранения пар «ключ – значение ключа», то будем предполагать, что данный АТД принадлежит к классу ассоциативных контейнеров.

2.3. Множества

Под *множеством* будем понимать некоторую совокупность *элементов*, каждый из которых является либо множеством, либо примитивным элементом, называемым *атомом*. В этом случае принято говорить, что атомы множества – это его неделимые элементы, в то время как элементы, отличные от атомарных, могут быть представлены как совокупность некоторого числа атомов (возможно, из другого множества). В дальнейшем рассмотрению будут подлежать конечные множества, т. е. те, число элементов в которых выражается натуральным числом либо нулем. Также будем предполагать, что элементы множества являются атомарными, например, множество чисел, строк, массивов, списков и т. д. (если не оговорено другое).

Будем говорить, что некоторое множество S является *линейно упорядоченным* (или просто *упорядоченным*), если на нем задано отношение линейного порядка. Основными операциями над множествами являются операции создания множества, вставки / удаления элементов, поиска, объединения и пересечения и др. Будем говорить, что некоторый АТД является «Множеством», если на нем поддерживаются операции, определенные на математической модели множества.

В зависимости от структуры элементов множества выбирается та либо иная структура данных для хранения его элементов в памяти ЭВМ. Так, например, цепочки символов (строки, последовательности бит и т. д.) обычно принято хранить в нагруженном дереве[*], в то время как числовые множества могут быть эффективно представлены как с помощью списковых АТД, так и с помощью деревьев. Следует отметить, что на выбор структуры данных, посредством которой будет реализован АТД «Множество», влияет не только природа обрабатываемых элементов, но также и то, является множеством статическим или динамическим. Ниже будут рассмотрены основные линейные способы представления множеств, а также специальный способ хранения тех, элементами которых являются множества.

2.3.1. Линейные способы представления множеств

К линейным способам представления множеств относятся массивы, линейные структуры данных, линейные АТД и битовые маски. Выбор того либо иного способа представления информации зависит от специфики задачи и от того, задано ли на множестве элементов отношение порядка. Так, массивы и линейные АТД, вместе с соответствующими структурами данных, обычно ис-

[*] Помимо термина *нагруженное дерево* используется термин *бор*. В англоязычной литературе используется термин *trie*.

пользуются для хранения упорядоченных множеств, в то время как битовые маски применяются в тех случаях, когда число элементов достаточно невелико (скажем, не превышает 20). Так как использование массивов для хранения множеств является достаточно очевидным, рассмотрим ниже способы, основанные на линейных списках и битовых масках.

Линейные списки

Будем считать, что на некотором множестве A задано отношение линейного порядка, в соответствие с которым элементы из A могут быть упорядочены. Тогда для представления подмножеств множества A можно воспользоваться как связными списками, так и линейными АТД, в которых элементы будут храниться в порядке $a_1, a_2, \dots, a_n$, причем $a_1 \preccurlyeq a_2 \preccurlyeq \cdots \preccurlyeq a_n$. Тогда операции объединения и пересечения двух множеств можно реализовать следующим образом (процедура 2.3.1):

Процедура 2.3.1: ADT Set: Union, Intersect

```
01:  Union Impl(L₁, L₂)
02:        L = ∅
03:        while not Is Empty(L₁) and not Is Empty(L₂) do
04:              if First(L₁) < First(L₂) then
05:                    L₁ = Move(L₁)
06:              else
07:                    L₂ = Move(L₂)
08:        while not Is Empty(L₁) do L₁ = Move(L₁)
09:        while not Is Empty(L₂) do L₂ = Move(L₂)
10:        return L
11:
12:        Move Impl(S)
13:              L = Insert(L, First(S))
14:              return Next(S)
15:
16:  Intersect Impl(L₁, L₂)
17:        L = ∅
18:        while not Is Empty(L₁) and not Is Empty(L₂) do
19:              if First(L₁) = First(L₂) then
20:                    L = Insert(L, First(L₁))
21:                    L₁ = Next(L₁)
22:                    L₂ = Next(L₂)
23:              if First(L₁) < First(L₂) then
24:                    L₁ = Next(L₁)
25:              if First(L₂) < First(L₁) then
26:                    L₂ = Next(L₂)
27:        return L
```

Трудоемкость приведенных выше операций есть $O(n_1 + n_2)$, где n_1 и n_2 – число элементов в множествах, которые подлежат обработке. Процедура же поиска во множестве, реализованном посредством линейного списка, будет линейна относительно числа его элементов, а вставка элемента в требуемую позицию будет иметь трудоемкость $O(1)$. Если же воспользоваться упорядочен-

ными массивами для представления множеств, то поиск элементов можно осуществлять за $O(\log n)$ операций, в то время как операция вставки будет иметь трудоемкость $O(n)$.

Битовые маски

Рассмотрим множество A, объекты которого могут быть занумерованы целыми числами от 0 до $n-1$. Если число элементов n во множестве A сопоставимо с размером машинного слова, то можно предложить следующий способ хранения подмножеств множества A в памяти ЭВМ. Для этого достаточно зарезервировать переменную целочисленного типа, в которой бит i будет соответствовать элементу a_i из A. Отождествив элементы множества A с их номерами, можно предложить следующую реализацию операций по созданию множеств (процедура 2.3.2):

Процедура 2.3.2: ADT Set: Make Set

```
01:  Make Set Impl(a)
02:        return 1 shl a
03:
04:  Make Set Impl(A)
05:        set = 0
06:        foreach a in A do
07:              set = Include(set, Make Set(a))
08:        return set
09:
10:  Make Universe Impl(n)
11:        return (1 shl n) - 1
```

Помимо операции создания наиболее востребованными являются операции включения / исключения, подсчета числа элементов, получения всех подмножеств данного множества и др. (процедура 2.3.3):

Процедура 2.3.3: ADT Set: Contains, Cardinality, Enumerate

```
01:  Contains Impl(set, subset)
02:        if set and subset = subset then
03:              return true
04:        return false
05:
06:  Next Impl(set, subset)
07:        return (subset – 1) and set
08:
09:  Cardinality Impl(set) {
10:        if set = 0 then
11:              return 0
12:        return 1 + Cardinality(Next(set, set))
13:
14:  Enumerate Impl(set)
15:        subset = set
16:        while subset > 0 do
17:              yield return subset
18:              subset = Next(set, subset)
19:        yield return 0
```

Следует отметить, что описанные выше операции являются достаточно эффективными, поскольку большинство их них выполняется за $O(1)$, кроме операции подсчета числа элементов и перечисления подмножеств. Обе операции имеют трудоемкость $O(b)$ и $O(2^b)$ соответственно, где b – число установленных бит во множестве.

Помимо данных операций для работы с битовыми масками, т. е. с множествами, закодированными посредством бит, на практике часто возникает потребность в обработке всех множеств из заданного универсального множества вместе с их подмножествами (процедура 2.3.4):

Процедура 2.3.4: ADT Set: Process Universe

```
01:  Process Universe Impl(n)
02:        universe = Make Universe(n)
03:        for set = 0 to universe do
04:              subset = set
05:              while subset > 0 do
06:                    {* process set and subset *}
07:                    subset = Next(set, subset)
```

Для того чтобы оценить время работы процедуры 2.3.4, достаточно заметить, что подмножества из k бит могут быть выбраны C_n^k способами и у каждого из выбранных множеств будет 2^k подмножеств. Следовательно, полное число подмножеств, которое будет обработано, равно $\sum_{k=0}^{n} C_n^k \cdot 2^k = (1+2)^n$. Отсюда можно заключить, что трудоемкость приведенной процедуры есть $O(3^n)$.

Если же число элементов множества A значительно превосходит размер машинного слова, а специфика задачи требует компактно представлять подмножества A, то можно воспользоваться целочисленным массивом, каждый элемент которого будет представлять собой некоторое подмножество. Тогда, для того, чтобы обработать элемент a_i необходимо и достаточно узнать: а) сегмент – соответствующую ячейку массива, в которой элемент a_i будет храниться; б) смещение – номер соответствующего бита, который кодирует элемент с номером i (процедура 2.3.5):

Процедура 2.3.5: ADT Set: Segment, Offset

```
01:  Segment Impl(i)
02:        return i div 32
03:
04:  Offset Impl(i)
05:        return i mod 32
```

Вышеприведенные процедуры исходят из того факта, что размер машинного слова равен 32 бита. Если же размер машинного слова отличен от 32, то соответствующая константа должна быть изменена (например, 64). В заключение, авторы рекомендуют обратиться к литературному источнику [6] как к незаменимому справочнику по манипулированию с битами.

2.3.2. Системы непересекающихся множеств

Рассмотрим совокупность множеств $S = \{A_1, \dots, A_n\}$. Будем говорить, что совокупность множеств S образует *систему непересекающихся множеств (СНМ)*, если $A_i \cap A_j = \emptyset$ для всех $1 \le i, j \le n, i \ne j$.[*]

В ряде случаев удобно считать, что множества A_i разбивают множество S на n классов эквивалентности, т. е. $S = A_1 \cup \dots \cup A_n$. У каждого из классов A_i обычно выделяется представитель $a_i \in A_i$, который идентифицирует данный класс. Представителем множества A_i может быть любой элемент, так как все элементы, образующие A_i, считаются эквивалентными с точки зрения множества S. Тогда под АТД «Система непересекающихся множеств» будем понимать нелинейный АТД, предназначенный для хранения СНМ и поддерживающий следующий набор операций:

1) создание одноэлементного множества;

2) объединение множеств;

3) поиск множества, которому принадлежит заданный элемент.

В дальнейшем изложении условимся, что множество S состоит из n объектов, занумерованных от 1 до n, а каждое из множеств A_i – подмножество S. Будем считать, что операции объединения и поиска работают не с самими объектами, а с отождествленными на них номерами. При этом операция поиска возвращает номер представителя множества, которому принадлежит заданный объект.

Основными структурами данных, которые используются для хранения информации о системе непересекающихся множества, являются массивы, линейные списки и деревья. Для того чтобы оценить эффективность структуры данных, используемой для хранения СНМ, будем предполагать, что на АТД выполняется набор из m операций создания, поиска и объединения.

Реализация на основе массива

Реализация СНМ через массив заключается в том, чтобы для каждого объекта сохранить в массив номер представителя множества, которому этот объект принадлежит (процедура 2.3.6):

Процедура 2.3.6: ADT DSS: Array

```
01:  Make Set Impl(D, a)
02:        with D do
03:              leader[a] = a
04:
05:  Find Set Impl(D, a)
06:        with D do
07:              return leader[a]
08:  Union Impl(D, a, b)
09:        a = Find(D, a)
10:        b = Find(D, b)
11:        if a ≠ b then
12:              Link(D, a, b)
```

[*] В англоязычной литературе используется термин *disjoint sets*.

```
13:         Link Impl(D, a, b)
14:               with D do
15:                     for i = 1 to n do
16:                           if leader[i] = b then
18:                                 leader[i] = a
```

В полученной реализации СНМ операция поиска работает за $O(1)$, в то время как операция объединения множеств имеет трудоемкость $O(n)$. В силу того, что операцию объединения можно выполнить не более $n-1$ раз, можно заключить, что любая последовательность из m операций будет выполняться за время, не превосходящее $O(n^2+m)$. Следовательно, реализация СНМ через массив крайне неэффективна в случае достаточно большого числа объектов. Поэтому данный подход применяется при работе с множествами, число элементов в которых невелико.

Реализация на основе списка

Оценку $O(n^2+m)$ можно значительно улучшить, если вместо массива использовать линейные списки для хранения элементов каждого из подмножеств (процедура 2.3.7):

Процедура 2.3.7: ADT DSS: List

```
01:  Make Set Impl(D, a)
02:         with D do
03:               leader[a] = a
04:               Insert(La, a)
05:
06:  Find Set Impl(D, a)
07:         with D do
08:               return leader[a]
09:
10:  Union Impl(D, a, b)
11:         a = Find(D, a)
12:         b = Find(D, b)
13:         if a ≠ b then
14:               Link(D, a, b)
15:
16:         Link Impl(D, a, b)
17:               with D do
18:                     if |La| ≥ |Lb| then
19:                           foreach x in Lb do
20:                                 leader[x] = a
21:                           Insert(La, Lb)
22:                     else Link(D, b, a)
```

В полученной реализации СНМ операция поиска работает за $O(1)$, в то время как операция объединения множеств имеет трудоемкость $O(\min\{|A_i|,|A_j|\})$, где A_i и A_j – объединяемые множества. В худшем случае операция объединения имеет трудоемкость $O(n)$. Несмотря на кажущуюся неэффективность операции объединения, трудоемкость выполнения любых m

операций не будет превосходить $O(n \log n + m)$. Действительно, как можно видеть из реализации операции объединения, ключевой идеей является присоединение меньшего множества к большему. Следовательно, каждый элемент после операции объединения переходит во множество, в котором будет по крайней мере в два раза больше элементов. Отсюда можно сделать вывод о том, что каждый из элементов сделает не более $O(\log n)$ переходов. Суммарное же число переходов для n элементов составит $O(n \log n)$, что приводит к оценке $O(n \log n + m)$ для m операций.

Полученная реализация данного АТД является достаточно эффективной как с точки зрения выполнения операций поиска и объединения, так и с точки зрения возможности перечисления элементов, которые составляют каждое из подмножеств. Для того чтобы перечислить элементы подмножества A_i достаточно совершить $O(|A_i|)$ операций, что лучше чем $O(n)$.

Реализация на основе леса

Наиболее эффективная реализация СНМ получается в том случае, если хранить элементы каждого множества в виде дерева. Тогда вся система непересекающихся множеств будет представлять собой набор деревьев – *лес*. Для хранения леса будем использовать массив предков P. Представителем множества в данной реализации СНМ будем считать корень дерева, которое содержит элементы этого множества.

Для реализации операции поиска необходимо подняться от объекта до корня дерева, в котором он находится, и возвратить найденное значение. Для реализации операции объединения необходимо провести ребро от корня одного из объединяемых деревьев к корню другого (процедура 2.3.8):

Процедура 2.3.8: ADT DSS: Forest Base Impl

```
01:  Make Set Impl(D, a)
02:        with D do
03:              parent[a] = a
04:
05:  Find Set Impl(D, a)
06:        with D do
07:              if a ≠ parent[a] then
08:                    return Find Set(D, parent[a])
09:              return a
10:
11:  Union Impl(D, a, b)
12:        a = Find(D, a)
13:        b = Find(D, b)
14:        if a ≠ b then
15:              Link(D, a, b)
16:
17:        Link Impl(D, a, b)
18:              with D do
19:                    parent[b] = a
```

Если объединять деревья случайным образом, как это сделано в процедуре 2.3.6, то пользы от рассматриваемой реализации СНМ не будет, – деревья могут выродиться в цепочки и операция поиска будет работать за линейное (относительно числа элементов во множестве) время. Для получения результата следует минимизировать высоту получаемого при объединении дерева. Для этого применяется следующая эвристика, которая носит название *объединение по рангам*. Рассмотрим массив R, состоящий из n элементов. Значение в ячейке $R[i]$ будем интерпретировать как оценку сверху для высоты дерева с корнем в узле i. Тогда операцию объединения можно проводить следующими способами: 1) если ранги объединяемых деревьев не равны, то проводится ребро от дерева с меньшим рангом к дереву с большим рангом; 2) если ранги равны, то проводится ребро произвольным образом, и при этом увеличивается на единицу ранг корня дерева, к которому было проведено ребро. Таким образом, на данном этапе реализации $R[i]$ есть высота дерева с корнем в вершине i.

Объединение по рангам приводит к таким же оценкам производительности, как и рассмотренная выше реализация СНМ через списки. Для того чтобы добиться еще большего выигрыша, рассмотрим эвристику, которая получила название *сжатие путей*. Идея эвристики сжатия путей заключается в том, чтобы у всех узлов, которые были рассмотрены на пути от объекта до корня дерева, содержащего этот объект, установить родителем корень дерева. Это позволит ускорить работу последующих вызовов операции поиска для этих объектов. Иллюстрация эвристики сжатия путей приведена на рис. 2.4. Реализация операций над системой непересекающихся множеств, с учетом рассмотренных эвристик, приведена в процедуре 2.3.9:

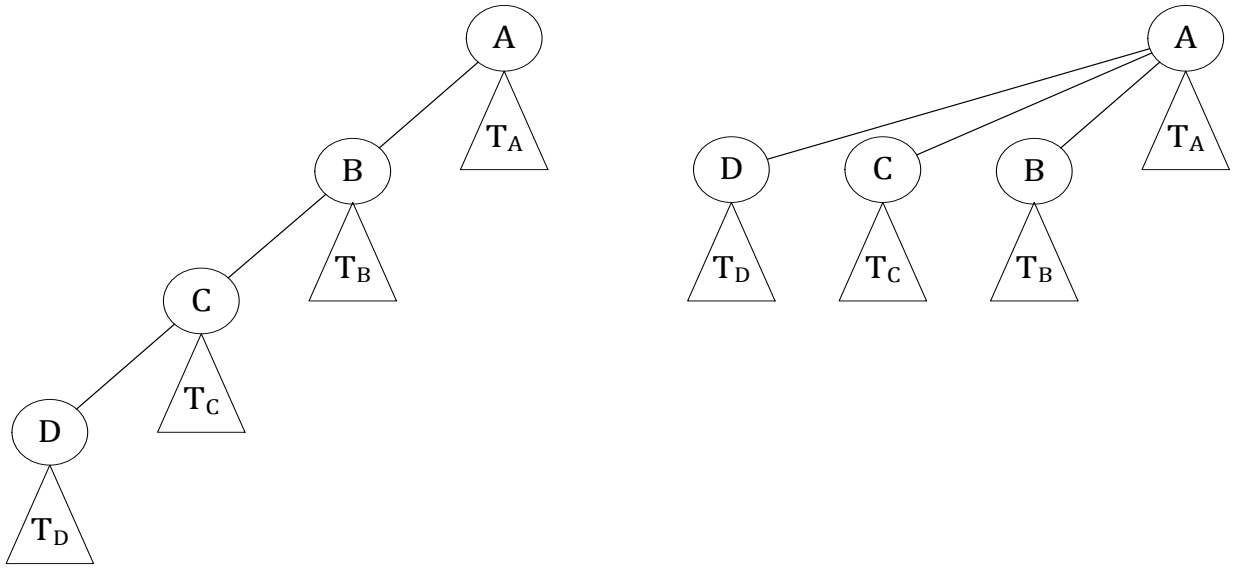

Рис. 2.4. Сжатие путей

```
Процедура 2.3.9: ADT DSS: Forest
01:  Make Set Impl(D, a)
02:        with D do
03:              parent[a] = a
04:              rank[a] = 1
```

```
05:  Find Set Impl(D, a)
06:        with D do
07:              if a ≠ parent[a] then
08:                    parent[a] = Find Set(D, parent[a])
09:              return parent[a]
10:
11:  Union Impl(D, a, b)
12:        a = Find(D, a)
13:        b = Find(D, b)
14:        if a ≠ b then
15:              Link(D, a, b)
16:
17:        Link Impl(D, a, b)
18:              with D do
19:                    if rank[a] < rank[b] then
20:                          parent[a] = b
21:                    else
22:                          parent[b] = a
23:                          if rank[a] = rank[b] then
24:                                rank[a] = rank[a] + 1
```

В полученной реализации СНМ трудоемкость операции объединения есть $O(1)$. Что касается трудоемкости операции поиска, то в худшем случае она может оказаться равной $O(n)$. В случае последовательности из m операций, суммарное время работы не будет превосходить $O(m\log^* n)$, где $\log^* n$ – это минимальное количество раз взятия двоичного логарифма от числа n для получения числа, которое меньше единицы. Следует отметить, что $\log^* n$ – очень медленно растущая величина. В частности, $\log^* n \leq 5$ для всех $n < 2^{65536}$. Так что можно считать, что операция поиска в данной реализации СНМ выполняется в среднем за время, равное константе. Оценка $O(m\log^* n)$ не является точной и может быть улучшена. Доказано, что описанная реализация СНМ является самой эффективной из всех возможных [7, 8].

В заключение отметим, что альтернативной эвристикой объединения по рангам может служить *весовая эвристика*, идея которой базируется на том, чтобы дерево с меньшим числом узлов подвешивать к корню дерева с большим числом узлов.

2.3.3. Словари

Под *словарем* будем понимать специальный тип АТД «Множество», предназначенный для хранения множеств, состоящих из пар «ключ – значение ключа». Словари принято называть *отображениями* (множество ключей отображается на множество значений), либо *ассоциативными массивами* (множество значений ассоциируется с множеством ключей). К основным операциям над словарями относятся операции создания словаря, вставки / удаления элементов, поиска значения по ключу.

Основными структурами данных, которые используются для реализации АТД «Словарь» являются бинарные деревья поиска, слоеные списки и хеш-таблицы. В соответствие с определением словаря, массивы можно отнести к словарным структурам данных, которые в качестве ключей используют целые числа. Реализация словаря посредством бинарных деревьев поиска может выглядеть следующим образом (процедура 2.3.10):

Процедура 2.3.10: ADT Dictionary

```
01:  dictionary<K, V> {
02:        container: binarySearchTree<K, V>
03:  }
04:
05:  Insert Impl(D, key, value)
06:        with D do
07:              Insert(container, key, value)
08:
09:  Delete Impl(D, key)
10:        with D do
11:              Delete(container, key)
12:
13:  Look Up Impl(D, key)
14:        with D do
15:              return Search(container, key)
```

Таким образом, трудоемкость операций вставки, удаления и поиска непосредственно зависит от трудоемкости выполнения соответствующих операций над структурой данных, которая лежит в основе словаря. Отметим, что при использовании хеш-таблиц множество ключей не обязательно линейно упорядоченное множество (т. е. такое, когда элементы можно сравнить между собой), в то время как для большинства древовидных структур данных это условие является необходимым.

Словарные структуры данных находят широкое использование на практике. В качестве примера может служить применение словарных структур данных к так называемой технике *мемоизации*[*] в контексте задач, решаемых при помощи метода динамического программирования.

Рассмотрим некоторую абстрактную вычислительную задачу P, которая может быть решена методом динамического программирования, путем разбиения исходной задачи P на более мелкие подзадачи $P_1, P_2, \dots, P_{n_P}$, где n_P – число таких подзадач. Предположим, что каждой задаче P соответствует некоторое состояние S, которое описывает задачу P. Также выделим ряд *элементарных задач*, для которых решение известно. Состояния, соответствующие элементарным задачам, назовем *начальными*. Связав все состояния в некоторый ациклический граф G, можно предложить следующий способ нахождения решения задачи P, которой соответствует состояние S (процедура 2.3.11):

[*] Термин *мемоизация* происходит от английского *memoization*.

```
Процедура 2.3.11: Memoization Technique
01:  Calculate Impl(S)
02:        if not Contains(cache, S) then
03:              result = ∅
04:              foreach Sᵢ in Adjacency List(G, S) do
05:                    accumulate(result, Calculate(Sᵢ))
06:              cache[S] = result
07:        return cache[S]
```

Трудоемкость данного алгоритма есть $O(V + E)$, где V – число состояний, а E – число переходов в графе G, при условии, что функция накопления результата работает за константное время.

Как можно видеть из приведенного алгоритма, в качестве *кеша* должна быть использована такая словарная структура данных, которая позволит эффективно осуществлять вставку и поиск элемента. Если состояния можно занумеровать натуральными числами, то в качестве кеша может быть использован массив. Перед непосредственным вызовом процедуры вычисления в словарь должны быть помещены элементарные состояния вместе с вычисленными для них значениями. Для того чтобы ответить на вопрос, почему данная техника дает ощутимый выигрыш, рассмотрим задачу определения количества путей, ведущих из вершины 1 в вершину n, на следующем графе G (рис. 2.5):

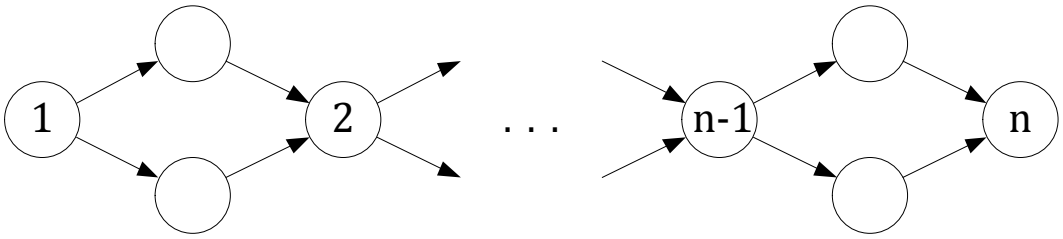

Рис. 2.5. Граф G

Если вычисление путей вести непосредственно, то количество шагов будет сравнимо с $O(2^n)$. Воспользовавшись техникой мемоизации, трудоемкость алгоритма можно значительно уменьшить. В этом случае достаточно запоминать количество путей для каждой из вершин на пути из 1 в n. Так как вычисления для каждой из вершин будут вестись ровно один раз, то трудоемкость алгоритма составит $O(n)$.

2.4. Очередь с приоритетами

Рассмотрим множества объектов A и B. Будем считать, что на множестве объектов B задано отношение частичного порядка «меньше либо равно», а элементы множества A занумерованы числами от 1 до n. *Очередь с приоритетом* для множества элементов из A и заданной функции $f: A \to B$ представляет собой АТД со следующими операциями:

1) добавление элемента;

2) удаление элемента с минимальным ключом.

Элементы множества B будем называть *приоритетами*. Соответственно, операция удаления элемента с минимальным ключом – это поиск и удаление

элемента, приоритет которого является наименьшим среди всех элементов, находящихся в очереди в данный момент.

Название «Очередь с приоритетами» обусловлено видом упорядочивания, которому подвергаются данные, хранящиеся посредством АТД. Термин «очередь» предполагает, что элементы ожидают некоторого обслуживания, а термин «приоритет» определяет последовательность обработки элементов. Необходимо отметить, что в отличие от обычных очередей, рассмотренных в разд. 2.2, АТД «Очередь с приоритетами» не придерживается принципа FIFO.

Реализация АТД «Очередь с приоритетами» может базироваться на основе массивов или списковых структур. Среди последних выбирают как связные списки, так и линейные АТД. Независимо от типа используемой линейной структуры данных, все реализации очереди с приоритетами можно условно разделить на два типа относительно того, хранятся элементы очереди в упорядоченном виде или нет. В случае упорядоченных последовательностей операцию поиска и удаления элемента с минимальным приоритетом можно выполнить за $O(1)$, а операцию вставки за $O(n)$. Если требуется удалить произвольный элемент из очереди, то на массиве потребуется $O(n)$ операций, в то время как на списковых структурах всего $O(1)$ при условии, что известна позиция, в которой находится удаляемый элемент. В случае неупорядоченных последовательностей операция вставки будет иметь трудоемкость $O(1)$, а операция поиска минимального элемента – $O(n)$. Трудоемкость выполнения операции удаления будет равна $O(n)$, если элементы хранятся в массиве, и $O(1)$, если в основе очереди лежит список.

Несложно видеть, что как линейные структуры данных, так и линейные АТД, не позволяют построить реализацию очереди с приоритетами, в которой бы все операции выполнялись быстрее, чем $O(n)$. В то же самое время такая реализация необходима, если число объектов, которые должны быть обработаны очередью, достаточно велико (скажем, $n \geq 1'000$). В таких случаях обычно применяются структуры данных, отличные от линейных, которые позволяют выполнять каждую из операций за $O(\log n)$. В большинстве своем эти структуры данных основаны на сбалансированных деревьях, из которых следует выделить бинарные кучи, биноминальные кучи, кучи Фибоначчи, деревья ван Эмде Боаса и др. Ниже будет рассмотрена реализация АТД «Очередь с приоритетами» на базе бинарной кучи, как наиболее распространенная среди остальных. Характеристика других типов деревьев выходит за рамки данного пособия [7].

Рассмотрим множество объектов A, на котором задано линейное отношение порядка, и бинарное дерево H, состоящее из $|A|$ узлов. Будем считать, что с каждым из узлов дерева H ассоциирован в точности один объект из ства A, который назовем *значением* узла. Тогда дерево H будем называть *бинарной кучей* (или *пирамидой*), если значение его узлов не превышает значений его потомков.

Бинарные кучи еще принято называть *частично-упорядоченными* деревьями, поскольку порядок задается только между родителями и детьми, в то время как порядок между братьями может быть выбран произвольно. Пример бинарной кучи для множества из 10 элементов {1,2,4,8,12,6,10,14,18,16} приведен на рис. 2.6:

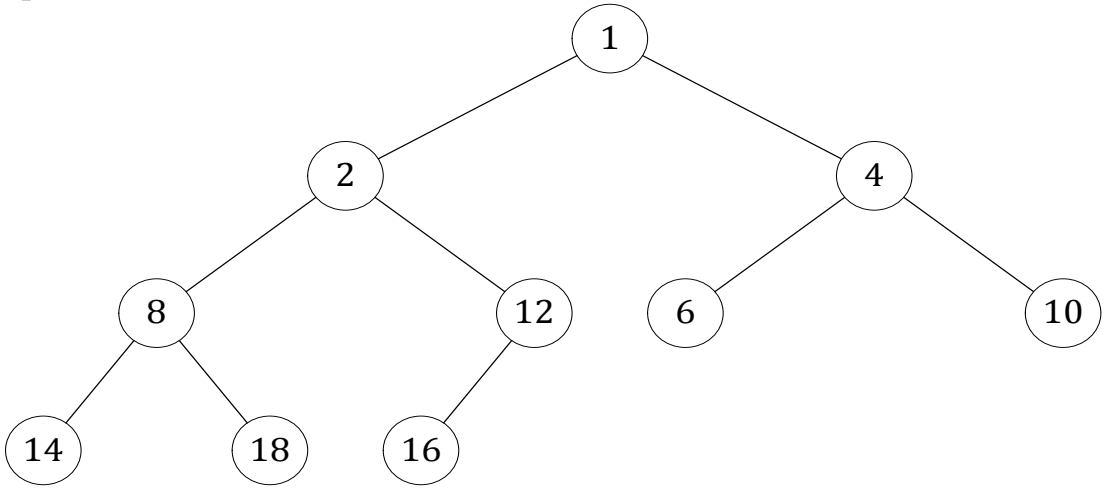

Рис. 2.6. Бинарная куча

Наиболее распространенным способом хранения пирамид в памяти ЭВМ является использование массивов. Согласно технике, описанной в разд. 2.2.5, применительно к бинарным деревьям можно предложить следующий способ хранения бинарной кучи (процедура 2.4.1):

Процедура 2.4.1: Heap

```
01:  heapEntry<T> {
02:        item: T
03:        position: integral
04:  }
05:  heap<T> {
06:        data: heapEntry<T>[max elements count]
07:        elementsCount: integral
08:  }
```

В дальнейшем изложении будем предполагать, что дерево H – это массив, для которого выполнено основное свойство бинарной кучи:

$$f(H[\lfloor i/2 \rfloor]) \leq f(H[i]),\ 1 < i \leq n, \tag{2.3}$$

где $f(H[i])$ – значение узла $H[i]$.

В частности, из условия (2.3) следует, что элемент с минимальным значением находится в корне дерева $H[1]$.

Процедуры нахождения детей и родителя данного узла могут выглядеть следующим образом (процедура 2.4.2):

Процедура 2.4.2: Heap: Parent, Left Child, Right Child

```
01:  Parent Impl(H, entry)
02:        if 1 < entry.position ≤ |H| then
03:              return H[entry.position / 2]]
04:        return ∅
```

```
05:  Left Child Impl(H, entry)
06:        if 2 * entry.position  ≤  |H| then
07:              return H[2 * entry.position]
08:        return ∅
09:  Right Child Impl(H, i)
10:        if 2 * entry.position + 1  ≤  |H| then
11:              return H[2 * entry.position + 1]
12:        return ∅
```

Основными операциями на бинарных кучах являются операция *проталкивания вверх* и операция *проталкивания вниз*. Суть каждой из операций заключается в том, чтобы поместить требуемый элемент на свое место в дереве с тем, чтобы сохранить основное свойство кучи. Таким образом, операция проталкивания вверх будет перемещать элемент до тех пор, пока не будет выполнено условие (2.3). Операция же проталкивания вниз будет перемещать элемент до тех пор, пока не будет выполнено условие

$$f(H[i]) \leq f(H[2 \cdot i]) \wedge f(H[i]) \leq f(H[2 \cdot i + 1]). \qquad (2.4)$$

В худшем случае каждой из операций может потребоваться пройти полный путь от самого глубокого листа до корня (в случае проталкивания вверх) либо наоборот (в случае проталкивания вниз), поэтому можно заключить, что каждая из них будет иметь трудоемкость $O(h)$, где h – высота дерева. Так как бинарная куча является сбалансированным деревом, то трудоемкость операций составит $O(\log n)$. Реализация операций проталкивания вверх и вниз приведена ниже (процедура 2.4.3):

Процедура 2.4.3: Heap: Swap, Push Up, Push Down

```
01:  Swap Impl(H, i, j)
02:        H[i].position = j
03:        H[j].position = i
04:        Swap(H[i], H[j])
05:
06:  Push Up Impl(H, entry)
07:        if entry.position > 1 then
08:              parent = Parent(H, entry)
09:              if f(entry) < f(parent) then
10:                    Swap(H, entry.position, parent.position)
11:                    Push Up(H, entry)
12:
13:  Push Down Impl(H, entry)
14:        leftChild  = Left Child (H, entry)
15:        rightChild = Right Child(H, entry)
16:        if leftChild ≠ ∅ then
17:              child = leftChild
18:              if rightChild ≠ ∅ then
19:                    if f(rightChild) < f(leftChild) then
20:                          child = rightChild
21:              if f(child) < f(entry) then
22:                    Swap(H, entry.position, child.position)
23:                    Push Down(H, entry)
```

На основе рассмотренных выше операций можно легко построить операции добавления и удаления элемента. Так, если требуется добавить элемент в кучу, то вначале он помещается на позицию самого нижнего правого листа, после чего происходит проталкивание вверх. В случае удаления элемента, он меняется местами с самым нижним правым листом, после чего происходит проталкивание вниз с позиции удаляемого элемента. Детали реализации приведены ниже (процедура 2.4.4):

Процедура 2.4.4: Heap: Push, Pop

```
01:  Push Impl(H, dataItem)
02:        if |H| + 1 ≤ max elements count then
03:              elementsCount = elementsCount + 1
04:              entry = heapEntry<T> {
05:                    item = dataItem
06:                    position = elementsCount
07:              }
08:              H[elementsCount] = entry
09:              Push Up(H, entry)
10:              return entry
11:        return ∅
12:
13:  Pop Impl(H, entry)
14:        if entry.position > 0 then
15:              rightist = H[elementsCount]
16:              Swap(H, entry.position, rightist.position)
17:              elementsCount = elementsCount - 1
18:              Push Up  (H, rightist)
19:              Push Down(H, rightist)
```

Помимо данного набора операций, которые можно выполнять над бинарными кучами, рассмотрим еще одну, суть которой заключается в том, чтобы преобразовать заданный массив объектов в пирамиду (процедура 2.4.5):

Процедура 2.4.5. Heap: Build

```
01:  Build Impl(A)
02:        H = Convert(A)
03:        for i = ⌊n / 2⌋ downto 1 do
04:              Push Down(H, i, n)
05:        return H
```

Для того чтобы оценить трудоемкость операции построения, заметим, что в худшем случае операции проталкивания вниз потребуется пройти весь путь от некоторого узла дерева до листа. Предположив, что высота дерева равняется h, и принимая во внимание тот факт, что на уровне i находится 2^i узлов, можно заключить, что полное число шагов S, которое совершит операция проталкивания вниз, не будет превышать следующей величины:

$$S \le \sum_{i=0}^{h} 2^i \cdot (h-i) = h + \cdots + 2^{h-1}\big(h-(h-1)\big) = 2^{h+1} - h - 2. \quad (2.5)$$

Так как бинарная куча – это сбалансированное бинарное дерево, то можно заключить, что трудоемкость операции построения не будет превышать величины $O(n)$. Следовательно, она линейна относительно числа элементов, на которых строится пирамида.

На основе бинарной кучи АТД «Очередь с приоритетами» может быть реализована следующим образом (процедура 2.4.6):

Процедура 2.4.6: ADT Priority Queue: Push, Peek, Pop

```
01:  priorityQueue<T> {
02:        H: Heap<T>(max elements count)
03:  }
04:
05:  Push Impl(Q, dataItem)
06:        with Q do
07:              Push(H, dataItem)
08:
09:  Peek Impl(Q, dataItem)
10:        with Q do
11:              if |H| > 0 then
12:                    return Item(H[1])
13:        return ∅
14:
15:  Pop Impl(Q, dataItem)
16:        with Q do
17:              if |H| > 0 then
18:                    Pop(H, H[1])
```

Помимо удовлетворительного времени работы каждой из операций (вставка и удаление – $O(\log n)$, нахождение минимума – $O(1)$) данный подход позволяет экономно расходовать память ЭВМ. Таким образом, если требуется обработать n объектов, то затраты по памяти составят $O(n)$ ячеек, что сопоставимо с реализацией очереди с приоритетами посредством массивов, других линейных структур данных и АТД.

В заключение ознакомимся с модификацией обычной очереди, которая позволяет находить элемент с минимальным ключом за $O(1)$. Такие модификации можно рассматривать с точки зрения очередей с приоритетами, на которых не определена операция удаления минимального элемента. Для того чтобы добиться оценки $O(1)$ для операции поиска минимального элемента, достаточно реализовать очередь посредством двух стеков, как это было сделано в разд. 2.2.3. Каждый из стеков, помимо хранения самих данных, дополнительно будет хранить в каждом элементе стека s минимальный элемент среди всех присутствующих, вершиной которого является элемент s. Тогда операцию нахождения минимума на стеке можно реализовать за $O(1)$. Для этого доста-

точно прочитать значение минимума, которое записано в вершине стека. Так как очередь будет реализована через два стека, то операция нахождения минимума в очереди будет также иметь трудоемкость $O(1)$.

Реализация очереди, описанная выше, находит широкое применение при работе с так называемыми *скользящими окнами*, которые используются для анализа потоков данных. Рассмотрим последовательность событий $e_1, e_2, \dots, e_n$, упорядоченных на временной оси. Предположим, что каждое из событий характеризуется некоторой числовой характеристикой. Скользящее окно предназначено для последовательной обработки событий e_i, причем максимальное число событий, которые могут одновременно находиться в рамках окна, задается некоторым числовым параметром t. Если же число событий, находящихся в рамках окна, превышает число t, то из окна удаляется событие, которое «попало» в окно ранее других. Задача состоит в том, чтобы достаточно быстро отвечать на ряд запросов, среди которых выделяется запрос «Чему равен минимальный элемент, находящийся в текущий момент времени в рамках окна?». Для ответа на такого рода запросы, можно применить либо очередь с приоритетами, либо описанную выше модификацию FIFO-очередей. Использование последней более предпочтительно, так как все операции будут иметь трудоемкость $O(1)$, в то время как использование очередей с приоритетами приведет к трудоемкости $O(\log n)$.[*]

2.5. Задачи для самостоятельного решения

1. Разработайте алгоритм, который для заданной позиции p в многомерном массиве $A[n_1, n_2, \dots, n_m]$ определит набор индексов $(i_1, i_2, \dots, i_m)$, соответствующий заданной позиции p. Считайте, что индексация по каждому из измерений начинается с нуля, т. е. $0 \le i_j < n_j,\ 1 \le j \le m$.

2. Задан массив A из n элементов. Необходимо циклически сдвинуть массив A на k позиций влево (вправо) используя $O(1)$ дополнительной памяти.

3. Задан неотсортированный массив A, содержащий n чисел. Определите множество P, в котором n элементов, где P_i является произведением всех чисел из A, за исключением a_i. Операция деления запрещена.

4. Задана матрица A размерностью $n \times m$, состоящая из чисел. Требуется найти подматрицу матрицы A, сумма элементов которой максимальна.

5. Задан массив A, в котором n чисел. Определите множество P, содержащее все числа, полученные путем операции сложения чисел из массива A. Предложите эффективный алгоритм, позволяющий преобразовать множество P во множество P_i, содержащее все числа, которые можно получиться из множества $A \backslash \{a_i\}$.

---

[*] В англоязычной литературе используются термины *sliding window* и *tumbling window*, различие между которыми определяется политикой удаления элементов. Помимо этого выделяется политика, определяющая условия, при которых срабатывает процедура обработки элементов окна. Детальное рассмотрение таких политик выходит за рамки данного пособия.

6. В заданном массиве A из n чисел требуется каждый элемент заменить на ближайший следующий за ним элемент, который больше него.

7. Рассмотрим n-мерный параллелепипед P размерами $n_1 \times n_2 \times ... \times n_m$. Разработайте алгоритм, который построит циклический обход заданного параллелепипеда, начиная с клетки с координатами $s = (s_1, s_2, ..., s_m)$. Каждая клетка параллелепипеда P должна быть посещена ровно один раз. За один ход разрешается перейти в соседнюю клетку по одной из координат. Сформулируйте условия, при которых требуемый обход параллелепипеда P не существует.

8. Задан односвязный список L, состоящий из n элементов. Предполагая, что известен указатель на голову списка L, разработайте алгоритм, имеющий трудоемкость $O(n)$ и требующий памяти $O(1)$, который определит, имеется ли в списке L цикл. Следует отметить, что по завершении работы Вашего алгоритма, список L должен остаться неизмененным. Модифицируйте разработанный алгоритм с целью нахождения количества элементов в цикле.

9. *Графом* будем называть пару (V, E), где V – непустое конечное множество, а E – бинарное отношение на V, т. е. подмножество множества $V \times V$. Множество V будем называть множеством *вершин* графа, а E – множеством *ребер* графа. Разработайте эффективную структуру данных для хранения информации о графе в памяти ЭВМ.

10. Рассмотрим форму тела, заданного матрицей A размерностью $n \times m$. Элемент $A[i, j]$ матрицы соответствует высоте вертикального столбика относительно нижнего основания площадки, расположенного горизонтально. Сечение всех столбиков есть квадрат размерностью 1×1. Требуется определить объем невытекшей воды, который останется внутри формы после того, как ее полностью погрузили в воду, а затем подняли.

11. Рассмотрим односвязный список L, у которого, помимо указателя на следующий элемент, имеется указатель на произвольный элемент того же списка L, отличный от нулевого (процедура 2.5.1):

Процедура 2.5.1. Custom Linked List Type

```
customLinkedListElement<T> {
        item: T
        next: customLinkedList<T>
        refr: customLinkedList<T>
}
customLinkedList<T> {
        head: customLinkedListElement<T>
}
```

Разработайте алгоритм, создающий глубокую копию списка L, т. е. такую копию L', структура которой будет полностью совпадать со структурой исходного списка L.

12. Рассмотрим ЭВМ, у которой есть в распоряжении N $(1 \leq N < 2^{32})$ последовательных ячеек памяти, пронумерованных от 1 до N. Разработайте ме-

неджер памяти, который будет обрабатывать запросы на выделение и освобождение памяти.

Запрос на выделение памяти имеет один параметр K, означающий, что необходимо выделить K последовательных ячеек памяти. Если в распоряжении менеджера есть хотя бы один свободный блок из K последовательных ячеек, то он обязан в ответ на запрос выделить такой блок. При этом непосредственно перед самой первой ячейкой памяти выделяемого блока не должно располагаться свободной ячейки памяти. Если блока из K последовательных свободных ячеек памяти нет, то запрос отклоняется.

Запрос на освобождение памяти имеет один параметр T. Такой запрос означает, что менеджер должен освободить память, выделенную ранее при обработке запроса с порядковым номером T. Запросы нумеруются, начиная с единицы. Гарантируется, что запрос с номером T – запрос на выделение, причем к нему еще не применялось освобождение памяти. Если запрос с номером T был отклонен, то текущий запрос на освобождение памяти игнорируется.

13. Рассмотрим матрицу A размерностью $n \times n$, состоящую из натуральных чисел. *Путем* в матрице A будем называть последовательность из n элементов следующего вида: $p = (a_{1j_1}, a_{2j_2}, \dots, a_{nj_n})$, $1 \le j_k \le n$. *Весом* пути p назовем сумму его элементов, т. е. $w(p) = a_{1j_1} + a_{2j_2} + \cdots + a_{nj_n}$. Разработайте алгоритм, который найдет n путей в матрице A, имеющих наибольший вес.

14. Разработайте алгоритм, находящий радиус наибольшей окружности, которую можно вписать в заданный выпуклый многоугольник.

15. Рассмотрим матрицу $A[n \times m]$, состоящую из 0 и 1. В матрице A требуется найти подматрицу максимальной площади, состоящую из одних единиц.

16. Рассмотрим площадку, основание которой представляет собой прямоугольник, разделенный на $N \times M$ квадратов (т. е. прямоугольник, состоящий из N строк и M столбцов). Квадрат с координатами (i, j), $1 \le i \le N$, $1 \le j \le M$ находится на высоте H_{ij}. Разработайте алгоритм, который найдет участок площадки прямоугольного размера, удовлетворяющий следующим условиям:

- стороны участка должны быть параллельны сторонам площадки;
- площадь участка должна быть максимально возможной;
- ширина участка (то есть, количество столбцов) не должна превышать заданного числа W;
- разница между высотой самого высокого квадрата и самого низкого квадрата участка не должна превышать значения L.

17. Рассмотрим $n + 1$ пунктов, пронумерованных от 0 до n соответственно. Некоторые из пунктов соединены реками. По ним можно сплавлять лес, который вырубается в каждом из пунктов. Также известно, что в пункте 0 находится лесопилка, которая может переработать сплавленный лес, и что для каждого пункта существует единственный путь по рекам в пункт 0. Требуется построить k дополнительных лесопилок ($1 \le k \le n$) так, чтобы суммарная стои-

мость сплава леса была минимальной. Стоимость сплава одного кубометра леса на расстояние в один километр равняется одному центу.

18. Используя идею техники двоичного подъема, разработайте алгоритм, который позволит эффективно находить минимальное / максимальное число на заданном участке массива. Предложите алгоритм, трудоемкость которого составила бы $\langle O(n \log n), O(1) \rangle$.

19. На координатной прямой расположено n точек в позициях $x_1, \dots, x_n$. Для простоты будем предполагать, что $x_1 < \cdots < x_n$. В момент времени 0 каждая точка, имеющая координату x_i, начинает двигаться вправо с постоянной скоростью $v_i > 0$. В некоторые моменты времени одна из точек может «обогнать» другую. Будем считать, что никакие два «обгона» не происходят одновременно. Упорядочим все обгоны по возрастанию момента времени, в которые они происходят. Требуется вывести информацию о первых k произошедших обгонах. Предполагается, что число k не превосходит общего числа обгонов.

20. Рассмотрим n копилок, каждая из которых может быть либо разбита, либо открыта соответствующим ключом. Для каждого из n ключей известно, в какой копилке находится ключ. Требуется определить минимальное число копилок, разбив которые можно получить доступ ко всем из них.

21. Рассмотрим n городов, соединенных между собой дорогами таким образом, что для каждой пары городов существует единственный путь между ними. Требуется определить минимальное число дорог, которое должно быть разрушено, чтобы образовалась изолированная от остальных группа ровно из m городов, такая что из одного города этой группы в другой по-прежнему можно будет добраться по неразрушенным дорогам. Группа *изолирована* от остальных, если никакая неразрушенная дорога не соединяет город из этой группы с городом из другой группы.

22. Задан массив A из m различных элементов, принадлежащих множеству $\{1, .., n\}, m \leq n$. Требуется с дополнительной памятью $O(n - m)$ определить элементы, отсутствующие в массиве A.

Процедура 2.5.2. Process

```
IEnumerable<int> process(int n, int m, int[] a) {
      if (m < n) {
            a[m, n) = -1;
            for (int i = 0; i < m; ++i) {
                  while (a[i] ≥ 0 && a[a[i]] ≠ a[i]) {
                        swap(a[a[i]], a[i]);
                  }
            }
            for (int i = 0; i < n; ++i) {
                  if (a[i] ≠ i) {
                        yield return i;
                  }
            }
      }
}
```

3. Сортировка

Рассмотрим множества объектов A и B. Будем считать, что на множестве объектов из B задано отношение линейного порядка $\preccurlyeq$, а элементы множества A занумерованы числами от 1 до n, где n – число элементов в A. *Задача сортировки* заключается в том, чтобы для заданной функции $f: A \to B$ упорядочить элементы из A в соответствие с отношением $\preccurlyeq$. Применительно к числам это может означать их упорядочивание либо по возрастанию, либо по убыванию.

В общем случае, целью сортировки является поиск такой перестановки $\pi = (\pi_1, \dots, \pi_n)$ элементов из A, для которой выполняются следующие условия:

$$f(a_{\pi_1}) \preccurlyeq f(a_{\pi_2}) \preccurlyeq \cdots \preccurlyeq f(a_{\pi_n}).$$

В дальнейшем будем предполагать, что отношение $\preccurlyeq$ суть отношение линейного порядка «меньше либо равно», и все последующие алгоритмы будут преследовать своей целью построение перестановки π, для которой

$$f(a_{\pi_1}) \leq f(a_{\pi_2}) \leq \cdots \leq f(a_{\pi_n}).$$

Методы сортировки могут быть классифицированы несколькими способами. Различают алгоритмы внешней сортировки и внутренней. Возможно деление на *устойчивые* и *неустойчивые* методы сортировки. Будем говорить, что алгоритм сортировки является *устойчивым*, если в процессе сортировки относительный порядок элементов с одинаковыми ключами не изменяется. Следует отметить, что устойчивые методы сортировки находят широкое применение на практике (в частности, некоторые из строковых алгоритмов используют устойчивую сортировку данных как промежуточный этап). Третья возможная схема классификации методов сортировки – их разделение в зависимости от того, на чем они основаны: на сравнении ключей или на некоторых свойствах ключей. Четвертым способом классификации является разделение алгоритмов сортировки по времени выполнения: квадратичные, логарифмические, линейные, экспоненциальные. Существуют и другие классификации, рассмотрение которых выходит за рамки данного пособия [9].

3.1. Общие сведения. Нижние оценки сложности

Будем говорить, что некоторый алгоритм сортировки упорядочивает элементы *на месте*[*], если для его успешного завершения требуется $O(1)$ единиц памяти. Таким образом, сортировкам «на месте» достаточно памяти, которая была выделена под массив, подлежащий упорядочиванию. Данный класс алгоритмов имеет важное практическое значение, поскольку алгоритмы, ему принадлежащие, могут быть пригодны для сортировки значительно больших объемов данных, нежели алгоритмы, затраты на память у которых составляют по крайней мере $O(n)$.

Независимо от того, какие данные упорядочивает алгоритм, интерес представляет получение оценки нижнего количества операций, необходимых

[*] В англоязычной литературе используется термин *in-place*.

для того, чтобы упорядочить данные в соответствие с заданным отношением порядка. Попробуем получить оценку нижнего числа операций для алгоритмов сортировки, базирующихся на сравнении.

Пусть имеется некий алгоритм $\mathcal{A}$, выполняющий сортировку элементов множества A путем сравнения ключей элементов. Алгоритму $\mathcal{A}$ можно поставить в соответствие *дерево сортировки,* имеющее следующую структуру. Узлы дерева содержат сравнение неких ключей элементов множества A a_i и a_j. Листья дерева представляют собой перестановку исходной последовательности, соответствующую отсортированному порядку. Количество сравнений, которое должен выполнить алгоритм $\mathcal{A}$ в конкретном случае, равняется количеству ребер, ведущих от корня дерева сортировки к некоторому его листу. Лучшему случаю соответствует самый короткий путь от корня к листу, худшему – самый длинный, а для подсчета среднего случая необходимо разделить суммарную длину всех путей от корня к листьям на количество листьев.

Количество возможных листьев дерева сортировки равно $n!$, где n – число элементов сортируемой последовательности. Пусть k – определенный уровень дерева сортировки, считая от корня (уровни нумеруются с нуля). Тогда количество узлов на k-ом уровне равно 2^k. В случае, когда дерево сортировки высоты k является идеально сбалансированным, для количества листьев l справедливо следующее неравенство:

$$n! \leq l \leq 2^k.$$

Логарифмируя данное неравенство и применяя формулу Стирлинга $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, получим следующую оценку для k:

$$k > (n + 0{,}5)\log_2 n - 1{,}4427n.. \qquad (3.1)$$

Таким образом, из неравенства (3.1) можно заключить, что нижняя оценка числа сравнений для алгоритма сортировки $\mathcal{A}$, использующего сравнения, удовлетворяет величине порядка $\Omega(n \log_2 n)$. Отметим, что существует и другое доказательство оценки $\Omega(n \log_2 n)$, базирующееся на том факте, что $n!$ является произведением не менее $n/2$ сомножителей, каждое из которых не превышает $n/2$, т. е. $n! \geq (n/2)^{n/2}$.

Следует отметить, что существует класс алгоритмов, которые работают за линейное время $O(n)$, относительно числа сортируемых элементов. Особенностью таких алгоритмов является использование некоторых особенностей ключа, позволяющих избежать попарного сравнения элементов. В качестве примера рассмотрим алгоритм сортировки подсчетом.

Пусть множество A состоит из n чисел от 0 до $k - 1$. Заведем массив подсчетов C из k элементов. Индекс массива подсчетов соответствует одному из элементов множества A. Перед началом сортировки обнулим массив C. Далее просмотрим элементы множества A и подсчитаем число различных элементов в нем. В силу того, что элементы множества A ограниченны, результат можно занести в массив C. После сбора необходимой статистики, построим отсортиро-

ванную последовательность, воспользовавшись информацией из массива C. Приведенный алгоритм иллюстрирует следующая процедура (3.1.1):

Процедура 3.1.1. Counting Sort

```
01:  Counting Sort Impl(A)
02:        C[i] = 0, i∈[0, k)
03:        foreach a in A do
04:              C[a] = C[a] + 1
05:        index = 1
06:        for key = 0 to k – 1 do
07:              for count = 0 to C[key] do
08:                    A[index] = key
09:                    index = index + 1
```

Проведем анализ алгоритма (3.1.1). Обнуление массива C требует k итераций, перебор всех элементов множества A требует n итераций. Операторы вложенного цикла в сумме выполнятся ровно n раз. Таким образом, время работы сортировки подсчетом можно оценить как $O(n + k)$. Следовательно, сортировка подсчетом выполняется за линейное, относительно числа элементов исходного массива, время. Узким местом приведенного алгоритма является наличие дополнительного массива C. Если, например, сортируется массив из 100 чисел, каждое из которых принимает значение от 0 до $2^{32} - 1$, то потребуется дополнительный массив из 2^{32} элементов. В этом случае ни о каком преимуществе по скорости (а тем более по используемой памяти) перед традиционными алгоритмами сортировки говорить не приходится.

3.2. Внутренняя сортировка

Под внутренней сортировкой будем понимать те алгоритмы сортировки или их реализации, используемые для упорядочивания данных, которые полностью помещаются в оперативную память ЭВМ в виде массива. Выбор массива в качестве структуры данных для хранения сортируемых элементов является не случайным, так как это единственная «родная» структура данных большинства ЭВМ, предоставляющая произвольный способ доступа к ячейкам массива. В таком случае затратами на доступ к элементам массива можно пренебречь и считать, что они выполняются за учетное время $O(1)$.

Ниже будут рассмотрены классические методы сортировки, которые условно могут быть разделены на квадратичные, логарифмические и линейные. Данная классификация обусловлена как временем выполнения алгоритмов сортировки, так и тем, что она является наиболее распространенной при выборе методов, используемых для упорядочивания данных.

3.2.1. Квадратичная сортировка

Методы квадратичной сортировки выполняют сортировку данных за $O(n^2)$ шагов и требуют $O(1)$ дополнительной памяти. Перед непосредственным изучением методов сортировки рассмотрим алгоритм, который является интуитивно понятным и базируется на том факте, что элементы исходного множества

должны быть упорядочены по не убыванию ключей. Как следствие, для достижения цели может быть использована процедура нахождения минимального элемента в массиве (процедура 3.2.1):

Процедура 3.2.1. Basic Quadratic Sort

```
01:  Basic Quadratic Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              {* f(A[outer]) is min from f(A[outer]), …, f(A[n]) *}
04:              for inner = outer + 1 to n do
05:                    if f(A[inner]) < f(A[outer]) then
06:                          swap(inner, outer)
```

Количество итераций, совершаемое приведенным алгоритмом пропорционально n^2, поскольку $(n-1)+(n-2)+\cdots+1=\frac{(n-1)^2+(n-1)}{2}$. Далее будут рассмотрены такие классические методы сортировки, как сортировка «пузырьком», выбором, вставками, и модификация сортировки вставками, предложенная Д. Л. Шеллом.

Сортировка пузырьком

Сортировка пузырьком является одним из простейших методов устойчивой сортировки, идея которого базируется на следующем наблюдении. Предположим, что элементы сортируемого массива A расположены вертикально. Тогда элемент с наименьшим значением ключа должен всплыть вверх, за ним всплывает следующий наименьший элемент и т. д. (процедура 3.2.2):

Процедура 3.2.2. Bubble Sort

```
01:  Bubble Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              for inner = n downto outer + 1 do
04:                    if f(A[inner]) < f(A[inner - 1]) then
05:                          swap(inner, inner - 1)
```

Проанализируем время работы алгоритма. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(1)=0, \\ T(n)=T(n-1)+O(n),\ n>1. \end{cases} \quad (3.2)$$

Решением данного соотношения является функция $T(n)=O(n^2)$, поэтому сложность алгоритма пузырьковой сортировки есть $O(n^2)$, где n – число элементов в A.

Время, затрачиваемое сортировкой «пузырьком», может быть улучшено на частично упорядоченных массивах. Предположим, что после выполнения внутреннего цикла не произошло ни одного обмена. Это означает, что часть массива, которая была подвергнута анализу внутренним циклом, является отсортированной, и дальнейшую работу можно прекратить (процедура 3.2.3).

Следует обратить внимание на то, что в худшем случае приведенная оптимизация не даст выигрыша и сложность алгоритма пузырька по-прежнему составит $O(n^2)$ операций.

Процедура 3.2.3. Improved Bubble Sort

```
01:  Improved Bubble Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              swapped = false
04:              for inner = n downto outer + 1 do
05:                    if f(A[inner]) < f(A[inner - 1]) then
06:                          swap(inner, inner - 1)
07:                          swapped = true
08:              if not swapped then return
```

Сортировка выбором

Сортировка выбором, как и алгоритм пузырька, является простейшим методом упорядочивания данных и базируется на идее базового алгоритма сортировки, приведенного выше (процедура 3.2.1). Разница между алгоритмами заключается в моменте времени, в который происходит обмен элементов. Алгоритм сортировки выбором приведен в процедуре 3.2.4:

Процедура 3.2.4. Selection Sort

```
01:  Selection Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              index = outer
04:              for inner = outer + 1 downto n do
05:                    if f(A[inner]) < f(A[index]) then
06:                          index = inner
07:              if (index <> outer) then
08:                    swap(index, outer)
```

Рекуррентное соотношение для времени работы сортировки выбором не приводится по причине точного совпадения с выражением (3.2). Следовательно, временная сложность рассмотренного метода составляет $O(n^2)$ операций. Следует отметить, что сортировка выбором не принадлежит к классу устойчивых сортировок.

Сортировка вставками

Еще одним из методов квадратичной устойчивой сортировки является метод вставок, идея которого заключается в следующем. На каждой итерации алгоритм ставит текущий элемент в требуемую позицию (процедура 3.2.5):

Процедура 3.2.5. Insertion Sort

```
01:  Insertion Sort Impl(A[1, n])
02:        for outer = 2 to n do
03:              current = A[outer]
04:              if f(current) < f(A[outer – 1]) then
05:                    inner = outer
06:                    while inner > 0 do
07:                          if f(current) < f(A[inner - 1]) then
08:                                A[inner] = A[inner - 1]
09:                                inner = inner – 1
09:                          else break
10:                    A[inner] = current
```

Проведем анализ времени работы алгоритма (3.2.5). Основное внимание уделим операторам внутреннего цикла. Если исходный массив уже отсортирован по возрастанию, внутренний цикл не выполнится ни разу. Если же исходный массив изначально отсортирован по убыванию, внутренний цикл на каждой итерации главного цикла будет выполняться i раз, где i – счетчик главного цикла. Следовательно, полное число итераций, которое совершит внутренний цикл есть $\frac{(n-1)n}{2} = O(n^2)$ раз.

Проанализируем поведение алгоритма в среднем случае. Пусть текущим является i-ый элемент. В результате работы внутреннего цикла этот элемент может остаться на месте, а может занять одну из позиций в отсортированной левой части. Следовательно, i-ый элемент может занять одну из i позиций, которые будем полагать равновероятными. Если элемент остается на месте, будет выполнено ноль итераций внутреннего цикла, если сдвигается на одну позицию влево – одна итерация, и т. д. Значит, среднее число итераций для элемента на i-ой позиции равно $\frac{1}{i}(0 + 1 + 2 + .. + (i-1)) = \frac{i-1}{2}$. Просуммируем полученное выражение по всем i:

$$T(n) = \sum_{i=2}^{n} \frac{i-1}{2} = O(n^2). \qquad (3.3)$$

Итак, время работы алгоритма сортировки вставками на множестве из n элементов в лучшем случае есть $O(n)$, а в среднем и худшем случае – $O(n^2)$. В среднем случае выполняется примерно в два раза меньше операций, нежели в худшем.

Следует отметить тесную связь между числом инверсий[*] k в массиве A и числом операций, которое совершает алгоритм сортировки вставками. Так как названная сортировка меняет только соседние элементы, то количество инверсий в массиве будет уменьшаться на единицу на каждой итерации алгоритма. Можно заключить, что сложность алгоритма сортировки вставками составит $O(n + k)$ шагов. Данная оценка совпадает с полученной ранее оценкой для худшего и среднего случая, так как в массиве, упорядоченном в обратном порядке, есть $\frac{n(n-1)}{2}$ инверсий, что соответствует $O(n^2)$.

Сортировка бинарными вставками

Сортировка вставками допускает модификацию, идея которой базируется на том факте, что вставка следующего элемента осуществляется в отсортированную часть массива. Как следствие, поиск позиции, в которую будет вставляться текущий элемент, можно осуществить, используя правосторонний двоичный поиск (процедура 3.2.6)[**]:

---

[*] *Инверсией* в массиве $A[n]$ называется такая пара индексов i и j $(1 \le i < j \le n)$, для которой выполнено $f(A_i) > f(A_j)$.

[**] Двоичный поиск и его модификации будут рассмотрены в разделе 4.

Процедура 3.2.6. Binary Insertion Sort

```
01:  Binary Insertion Sort Impl(A[1, n])
02:        for outer = 2 to n do
03:              current = A[outer]
04:              if f(current) < f(A[outer – 1]) then
05:                    index = Get Index(A[1, outer – 1], f(current))
06:                    for inner = outer downto index + 1
07:                          A[inner] = A[inner - 1]
08:                    A[index] = current
```

Следует отметить, что алгоритм бинарной вставки не позволяет улучшить оценку $O(n)$ для числа перемещений элементов, несмотря на то, что число сравнений уменьшается до $O(\log n)$. Поэтому в худшем случае сортировка бинарными вставками будет по-прежнему работать за $O(n^2)$ шагов.

Модификация Шелла

Д. Л. Шелл в 1959 г. предложил следующую модификацию алгоритма сортировки вставками [10]. В отличие от алгоритма (3.2.5), Шеллом было предложено рассматривать не только элементы, расположенные в соседних позициях, но и элементы, отстоящие на некотором расстоянии h друг от друга. Далее алгоритм упорядочивает элементы, расположенные на некотором расстоянии меньшем h и т. д. Завершается сортировка Шелла упорядочиванием элементов, отстоящих друг от друга на расстоянии $h = 1$, то есть обычной сортировкой вставками. Последовательность шагов, которая была предложена Шеллом, имеет вид:

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, \frac{n}{2^k}, 1. \tag{3.4}$$

Сортировка методом Шелла, соответствующая набору шагов (3.4) приведена ниже (процедура 3.2.7):

Процедура 3.2.7. Shell Sort

```
01:  Shell Sort Impl(A[1, n])
02:        scale = 1
03:        while (n shr scale > 0) do {* shift n by scale bits to the right *}
04:              offset = n >> scale
05:              for outer = offset + 1 to n do
06:                    inner = outer
07:                    while inner > 0 do
08:                          if f(A[inner]) < f(A[inner - offset]) then
09:                                swap(inner, inner – offset)
10:                                inner = inner – offset
11:                          else break
12:              scale = scale + 1
```

Время работы процедуры (3.2.7) в худшем случае составляет $O(n^2)$ операций, что соответствует массиву данных, в котором половина элементов с меньшими значениями находится в четных позициях, а половина элементов с большими значениями – в нечетных позициях.

Помимо последовательности шагов (3.4), предложенной непосредственно Шеллом, существует множество других, которые дают лучшие оценки для худшего случая. Например, последовательность Д. Кнута [9]

$$1,4,13,40,\dots,\frac{3^k-1}{2},\frac{3^k-1}{2}\le\left\lceil\frac{n}{3}\right\rceil \tag{3.5}$$

и последовательность Р. Седжвика [11]

$$1,5,19,41,109,\dots,h_i=\begin{cases}9\cdot 2^i-9\cdot 2^{i/2}+1, & i-\text{четно},\\ 8\cdot 2^i-6\cdot 2^{\lceil i/2\rceil)}+1, & i-\text{нечетно},\end{cases}\quad 3\cdot h_i\le n \tag{3.6}$$

В случае последовательности Кнута алгоритм потребует $O(n\sqrt{n})$ операций, а для последовательности Седжвика – $O(n^{4/3})$ операций в худшем случае.

Следует отметить, что сортировка Шелла работает с любой убывающей последовательностью шагов, заканчивающейся единицей, так как после упорядочивания с шагом h_i элементы массива, упорядоченные на итерации $i-1$ с шагом h_{i-1} ($h_i < h_{i-1}$), по-прежнему остаются отсортированными [9].

3.2.2. Логарифмическая сортировка

Ниже будут рассмотрены алгоритмы сортировки, базирующиеся на сравнении ключей, время выполнения которых в среднем случае требует $O(n\log n)$ операций. Данная оценка является асимптотически наилучшей (как это было отмечено в разд. 3.1), поэтому названные алгоритмы получили широкое распространение на практике для сортировки больших объемов данных. Несмотря на оптимальную оценку для среднего случая, некоторые из логарифмических алгоритмов сортировки требуют порядка $O(n^2)$ операций в худшем случае.

Следует отметить, что как алгоритм быстрой сортировки, так и алгоритмы слияния и пирамидальной сортировки, имеет смысл применять лишь для упорядочивания множеств, число элементов в которых превосходит $16*$. Для сортировки же небольших участков массива обычно применяется либо сортировка вставками, либо ее модификация – сортировка Шелла.

Быстрая сортировка

Алгоритм *быстрой сортировки* был предложен Ч. Хоаром в 1962 г. [9]. Идея алгоритма заключается в следующем. Вначале выбирается элемент множества A, называемый *осевым*, после чего происходит переупорядочивание элементов массива относительно ключа осевого элемента так, чтобы все элементы, расположенные левее осевого, были меньше его, а элементы, расположенные правее осевого, были больше его. В левой и правой частях (относительно осевого элемента) массива элементы остаются неупорядоченными. Следовательно, алгоритм быстрой сортировки требуется применить к левой и правой частям массива, т. е. вызвать рекурсивно (процедура 3.2.8):

---

[*] Выбор числа 16 обусловлен практическими результатами. В зависимости от специфики задачи данное значение может быть уменьшено, например, до 8 либо 4.

Процедура 3.2.8. Quick Sort

```
01:  return Quick Sort Impl(A[1, n], 1, n)
02:
03:  Quick Sort Impl(A, lo, hi)
04:        if lo < hi then
05:              i = lo, j = hi
06:              pivot = f(A[lo + ⌊(hi - lo)/2⌋])
07:              do
08:                    while f(A[i]) < pivot do i = i + 1
09:                    while pivot < f(A[j]) do j = j - 1
10:                    if i ≤ j then
11:                          swap(i, j)
12:                          i = i + 1
13:                          j = j - 1
14:              while i ≤ j
15:              if i < hi then Quick Sort Impl(A, i, hi)
16:              if lo < j then Quick Sort Impl(A, lo, j)
```

В силу того, что алгоритм быстрой сортировки имеет важное практическое значение, приведем детальный анализ времени его выполнения для среднего, лучшего и худшего случаев.

Начнем со среднего случая. Пусть $A(n)$ – количество сравнений в среднем для массива длины n. Так как быстрая сортировка является рекурсивным алгоритмом, найдем рекуррентное соотношение для $A(n)$. Очевидно, что $A(0) = A(1) = 0$. В общем случае осевой элемент может быть выбран произвольно. Пусть позиция осевого элемента равняется p. Тогда рекуррентное соотношение будет иметь вид:

$$\begin{cases} A(n) = 0,\ n < 2 \\ A(n) = A(p-1) + A(n-p) + (n-1),\ n \geq 2. \end{cases}$$

Будем считать, что значение p с равной вероятностью распределено от 1 до n. Таким образом

$$A(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}\big(A(i-1) + A(n-i)\big).$$

В записанной сумме аргумент первого слагаемого пробегает значения от 0 до $n-1$, а аргумент второго слагаемого – те же значения, но в обратном порядке. Следовательно,

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} A(i).$$

Из выражения для $A(n) \cdot n - A(n-1) \cdot (n-1)$ получим, что

$$A(n) = \frac{(n+1)}{n}A(n-1) + \frac{2n-2}{n}.$$

Сделав замену $D(n) = \frac{A(n)}{n+1}$, получим следующее рекуррентное соотношение для $D(n)$:

$$D(n) = D(n-1) + 2\frac{n-1}{n(n+1)}.$$

Путем применения метода итераций к выражению для $D(n)$ можно показать, что замкнутый вид для $D(n)$ будет выражаться формулой

$$D(n) = 2\sum_{k=1}^{n}\frac{k-1}{k(k+1)} = 2\left(H_{n+1} + \frac{1}{n+1} - 2\right), \qquad (3.7)$$

Из (3.7) следует выражение для $A(n)$

$$A(n) = (n+1)D(n) \approx 2(n+1)\ln(n+1) - (4-2\gamma)n \approx$$
$$\approx 2(n+1)\ln(n+1) = O(n\log n),$$

которое говорит о том, что быстрая сортировка имеет трудоемкость $O(n\log n)$ в среднем случае.

Оценим количество сравнений для алгоритма быстрой сортировки в лучшем и худшем случаях. Очевидно, что наименьшее число сравнений будет достигаться, если осевой элемент оказывается ровно в середине сортируемого участка. Таким образом, $B(n) = (n-1) + 2B(n/2)$. Решением данного рекуррентного соотношения, является функция $B(n) = O(n\log n)$.

Наихудший случай для числа сравнений получается тогда, когда осевой элемент разбивает массив на максимально неравные участки, т. е. когда осевой элемент оказывается либо минимальным, либо максимальным элементом сортируемого массива. В этом случае имеет место следующее рекуррентное соотношение $W(n) = (n-1) + W(n-1) + T(1)$. Решением данного рекуррентного соотношения, является функция $W(n) = O(n^2)$. Таким образом, в худшем случае алгоритм быстрой сортировки не является быстрым. Его трудоемкость сравнима с трудоемкостью квадратичных алгоритмов.

Помимо выбора среднего элемента в качестве осевого (т. е. элемента, находящегося в средней позиции), существует более эффективный подход, базирующийся на выборе «медианы из трех». Суть метода заключается в том, чтобы произвольным образом выбирать три элемента из сортируемого участка и в качестве осевого выбирать медиану среди выбранных элементов. Сам же Ч. Хоар предлагает выбирать осевой элемент случайно.

Одним из основных приложений алгоритма быстрой сортировки является нахождение k-ой порядковой статистики множества A, т. е. такого элемента из A, который после упорядочивания A будет находиться на k-ой позиции. К частным случаям рассматриваемой задачи можно отнести поиск минимального и максимального элементов, который в обоих случаях выполняется за $O(n)$ операций. В общем случае поиск k-го по величине элемента можно выполнить предварительно отсортировав массив A и затем взяв элемент, находящийся на

k-ой позиции. Тогда алгоритм затратит как в худшем, в среднем, так и в лучшем случаях $O(n \log n)$ шагов. Существует алгоритм, который в среднем случае выполняется за $O(n)$ шагов и идея которого базируется на идее алгоритма быстрой сортировки (процедура 3.2.9).

Процедура 3.2.9. Select

```
01:  Select Impl(A[1, n], k)
02:        pivot = f(A[random(1, n)])
03:        A₁, A₂ = []
04:        for i = 1 to n do
05:              if f(A[i]) < pivot then add A[i] into A₁
06:              if f(A[j]) > pivot then add A[i] into A₂
07:        if k ≤ |A₁| then
08:              return Select Impl(A₁, k)
09:        if k > n - |A₂| then
10:              return Select Impl(A₂, k – (n - |A₂|))
11:        return pivot
```

Следует отметить, что существует усовершенствованная версия рассмотренного алгоритма, которая гарантирует время поиска $O(n)$ в худшем случае. Детали алгоритма приводятся в [4, 12].

В заключение рассмотрения алгоритма быстрой сортировки скажем о том, что существует нерекурсивная реализация алгоритма (3.2.8), базирующаяся на АТД «Стек». Такая реализация алгоритма находит применение на практике, поскольку позволяет избежать накладных расходов, связанных с рекурсивными вызовами. Помимо этого, алгоритм быстрой сортировки может быть реализован на многопроцессорной машине, т. к. рекурсивные вызовы сортируют не пересекающиеся между собой части исходного массива A.

Сортировка слиянием

Алгоритм *сортировки слиянием* основан на стратегии декомпозиции и использует технику «разделяй и властвуй». Впервые алгоритм сортировки слиянием был предложен Дж. фон Нейманом в 1945 г. [9]. Идея его состоит в следующем. Первый этап заключается в разбиении массива на две равные части. На втором этапе алгоритм рекурсивно применяется к каждой из полученных частей. Последний, третий этап алгоритма, сливает отсортированные на предыдущем шаге части в один результирующий массив (отсюда название «сортировка слиянием»). Следует отметить, что процедура слияния будет линейна по времени выполнения, т. е. требовать $O(n)$ операций (процедура 3.2.10):

Процедура 3.2.10. Merge Sort

```
01:  Merge Sort Impl(A)
02:        if |A| > 1 then
03:              L = A[1, [n / 2]]
04:              R = A[[n / 2] + 1, n]
05:              Merge Sort Impl(L)
06:              Merge Sort Impl(R)
07:              Merge(A, L, R)
```

```
08:  Merge(A, L, R)
09:        a = 1, l = 1, r = 1
10:        while l ≤ |L| and r ≤ |R| do
11:              if f(L[l]) < f(R[r]) then
12:                    A[a] = L[l], l = l + 1
13:              else
14:                    A[a] = R[r], r = r + 1
15:              a = a + 1
16:        A[a, n] = L[l, |L|]
17:        A[a, n] = R[r, |R|]
```

Проанализируем время работы приведенного алгоритма. Пусть длина исходного массива равна n. Как было отмечено выше, слияние двух отсортированных частей требует времени, которое линейно зависит от n. С учетом правила разделения на подзадачи, получаем следующее рекуррентное соотношение для времени работы алгоритма:

$$\begin{cases} T(1) = 0, \\ T(n) = 2T(n/2) + O(n), \ n > 1. \end{cases} \tag{3.8}$$

Решением данного соотношения является функция $T(n) = O(n \log_2 n)$, поэтому сложность алгоритма сортировки слиянием есть $O(n \log_2 n)$, где n – число элементов в A. В худшем случае алгоритм сортировки требует также $O(n \log_2 n)$ шагов. Делаем вывод, что сортировка слиянием является асимптотически лучше алгоритма быстрой сортировки. Следует отметить, что сортировка слиянием принадлежит к классу устойчивых алгоритмов сортировки, а также может быть реализована на многопроцессорной машине, поскольку рекурсивные вызовы сортируют не пересекающиеся между собой части L и R исходного массива A.

Сортировка слиянием, рассмотренная в том виде, в каком она приведена в процедуре (3.2.10) обладает одним серьезным недостатком, который заключается в выделении дополнительной памяти под массивы L и R, в результате чего затраты по памяти составляют $O(n)$ байт. Существует модификация алгоритма сортировки слиянием, сортирующая элементы массива на месте и базирующаяся на так называемой технике *sqrt-декомпозиции*. Существуют и более сложные методы, описание которых приводится в [9]. Общей идей, объединяющей все существующие подходы, является разделение отсортированных частей массива A на некоторое количество блоков $K_1, K_2, \dots K_m$, желательно одинакового размера, после чего происходит слияние соседних блоков между собой, возможно, с некоторым предварительным числом подготовительных операций, общее число которых не превышает $O(n)$.

Пирамидальная сортировка

Пирамидальная сортировка базируется на АТД «Очередь с приоритетом», реализованной посредством структуры данных бинарная куча, рассмотренной ранее в разд. 2.4. Таким образом, алгоритм пирамидальной сортировки может выглядеть следующим образом (процедура 3.2.11):

Процедура 3.2.11. Heap Sort Base

```
01:  Heap Sort Base Impl(A)
02:        priorityQueue queue(n, f)
03:        foreach a in A do
04:              queue.push(a)
05:        for i = 1 to n do
06:              A[i] = queue.top()
07:              queue.pop()
```

Трудоемкость представленного алгоритма есть $O(n \log n)$ в худшем случае, поэтому пирамидальная сортировка, как и алгоритм сортировки слиянием, считается асимптотически лучше алгоритма быстрой сортировки. Недостатком процедуры (3.2.11) является использование АТД «Очередь с приоритетами», в связи с чем затраты на память составят $O(n)$, где n – число сортируемых элементов. Можно попытаться свести затраты памяти к $O(1)$, но для этого следует отойти от использования очереди и реализовать бинарную кучу непосредственно на массиве A (процедура 3.2.12):

Процедура 3.2.12. Heap Sort

```
01:  Heap Sort Impl(A)
02:        for i = ⌊n / 2⌋ downto 1 do
03:              Push Down(A, i, n)
04:        for i = n downto 1 do
05:              swap(A[1], A[i])
06:              Push Down(A, 1, i – 1)
```

Следует отметить, что в реализации процедуры «проталкивания вниз» следует учесть то, что наверху должен оказаться элемент с наибольшим ключом, что в дальнейшем позволяет его ставить в корректную позицию. Важным замечанием является то, что в общем случае пирамидальная сортировка не принадлежит к классу устойчивых алгоритмов. Если же такое поведение потребуется, то функция f получения ключа должна быть изменена таким образом, чтобы учитывать начальную позицию элементов в исходном массиве.

Несмотря на то, что пирамидальная сортировка является асимптотически оптимальной, на практике рассмотренный алгоритм не получил широкого распространения в силу большой константы, скрытой в асимптотике O.

3.2.3. Линейная сортировка

Линейная сортировка выполняет упорядочивание множества A за фиксированное число просмотров всех элементов из A. Особенностью методов, требующих линейное число операций, является использование некоторых внутренних характеристик ключей, которые позволяют избавиться от операции сравнения. Единственным ограничением, которое накладывается на структуру ключей, является их конечное множество. Таким образом, ниже будут рассмотрены алгоритмы, трудоемкость которых есть $O(n)$ в худшем случае, где n – число сортируемых объектов.

Блочная сортировка

Блочная сортировка (или *карманная*) – алгоритм линейной сортировки, предназначенный для упорядочивания таких наборов данных, ключи которых принадлежат множеству B с заведомо известным числом элементов. В этом случае ожидаемое время работы алгоритма есть $O(n)$, т. е. линейно зависит от числа элементов в заданном наборе.

Рассмотрим основные принципы работы блочной сортировки. Оригинальная версия алгоритма базируется на том факте, что ключи элементов сортируемого множества являются вещественными числами, равномерно распределенными на интервале $[0,..,1)$. Далее алгоритм распределяет элементы множества A по карманам, каждый из которых представляет собой АТД «Список» либо другой АТД, в который вставка элементов осуществляется за $O(1)$. Далее элементы каждого из карманов сортируются каким-либо алгоритмом сортировки (например, сортировкой вставками). После того, как карманы отсортированы, происходит их слияние в результирующий массив. Предположив, что количество карманов равно n и функция распределения по карманам выглядит как $d(a) = \lfloor n \cdot f(a) \rfloor$, можно предложить следующую реализацию карманной сортировки (процедура 3.2.13):

Процедура 3.2.13. Bucket Sort

```
01:  Bucket Sort Impl(A)
02:        bucket[i].clear(), i∈[0, n)
03:        foreach a in A do
04:              bucket[⌊n·f(a)⌋].push(a)
05:        for i = 0 to n – 1 do
06:              sort(bucket[i])
07:        A = bucket[0] ∪ bucket[1] ∪ ... ∪ bucket[n – 1]
```

Проанализируем время работы приведенного алгоритма. Соответствующее рекуррентное соотношение будет иметь вид

$$T(n) = \sum_{i=0}^{n-1} O(n_i^2) + O(n), \tag{3.9}$$

где n_i – число элементов в кармане i.

В случае равномерного распределения ключей решением полученного рекуррентного соотношения является функция $T(n) = O(n)$ [4, 7]. Если же ключи элементов распределены не равномерно, то в худшем случае трудоемкость приведенного алгоритма будет составлять $O(n^2)$.

Следует отметить, что если вставка в карман осуществляется в конец списка и используемый алгоритм сортировки карманов является устойчивым, то и вся блочная сортировка в целом также будет принадлежать к классу устойчивых алгоритмов сортировки.

Одной из модификаций блочной сортировки является *сортировка подсчетом*. Простейшая версия сортировки подсчетом была рассмотрена ранее в разд. 3.1, процедура (3.1.1). Ниже приводится более совершенный алгоритм

сортировки подсчетом, который не учитывает тот факт, что сортируемые элементы суть числа, а базируется лишь на том, что ключи сортируемых элементов принадлежат множеству $[0,\ k)$ (процедура 3.2.14):

Процедура 3.2.14. Counting Sort

```
01:  Counting Sort Impl(A)
02:        C[i] = 0, i∈[0, k)
03:        foreach a in A do
04:              C[f(a)] = C[f(a)] + 1
05:        for key = 1 to k – 1 do
06:              C[key] = C[key] + C[key – 1]
07:        for i = n downto 1 do
08:              key = f(A[i])
09:              B[C[key]] = A[i]
10:              C[key] = C[key] – 1
11:        A = B
```

Время работы процедуры (3.2.14), как и процедуры (3.1.1), составляет $O(n+k)$ шагов. Обратим внимание на то, что заключительный проход по элементам массива A осуществляется в обратном порядке, что позволяет добиться устойчивого поведения алгоритма.

Поразрядная сортировка

Будем рассматривать множество элементов A и такое множество ключей B, каждый из которых может быть представлен в виде

$$b = b_d b_{d-1} \dots b_1, b \in B. \tag{3.10}$$

Будем говорить, что ключ элемента $f(a) = b$ состоит из d разрядов, причем значение в разряде i $(1 \le i \le d)$ ключа b может быть получено как $f_i(a)$. Также будем считать, что на множестве элементов B_i, соответствующего значениям, которые могут встречаться на позиции i-го разряда, задано отношение линейного порядка «меньше либо равно» и в каждом из множеств B_i существует нулевой элемент.

Поразрядная сортировка (или *цифровая*) – алгоритм сортировки, предназначенный для упорядочивания наборов данных, ключи которых принадлежат множеству B с описанными выше свойствами (процедура 3.2.15):

Процедура 3.2.15. Radix Sort

```
01:  Radix Sort Impl(A)
02:        for i = 1 to d do
03:              sort A using f_i:A → B_i key-function
```

Время работы поразрядной сортировки зависит от времени выполнения сортировки массива A по одному из ключей. Так, если вспомогательная процедура сортировки работает за линейное время $O(n)$, то приведенный алгоритм поразрядной сортировки будет иметь трудоемкость $O(d \cdot n)$.

Поразрядную сортировку принято разделять на два класса: сортировка от младших разрядов к старшим (LSD-сортировка) и сортировка от старших разрядов к младшим (MSD-сортировка)[*].

LSD-сортировка

Основным применением алгоритма LSD-сортировки является упорядочивание данных, длина ключей которых является одинаковой. Если же длины ключей оказались различны, то их всегда можно выровнять путем добавления нулевых элементов на позиции старших разрядов.

Как было отмечено выше, LSD-сортировка проводит упорядочивание данных, начиная от младших разрядов к старшим и, таким образом, придерживается следующего линейного порядка:

$$\forall b, c \in B | b \leq c \Leftrightarrow b_i = c_i \forall i = \overline{1, d} \vee \exists i : b_j = c_j, j < i \wedge b_i < c_i.$$

Если в качестве вспомогательной процедуры использовать сортировку подсчетом, то можно предложить следующую реализацию алгоритма (процедура 3.2.16):

Процедура 3.2.16. LSD Radix Sort

```
01:  LSD Radix Sort Impl(A)
02:         for i = 1 to d do
03:               Counting Sort(A, f_i)
```

В силу того, что сортировка подсчетом в худшем случае работает за $O(n + k)$ шагов, трудоемкость алгоритма LSD-сортировки есть $O(d \cdot (n + k))$.

Следует отметить, что использование LSD-сортировки оправдано в том случае, если число разрядов является не достаточно большим. Так, если требуется отсортировать n $(n \geq 10^5)$ 32-битных чисел, то предпочтительнее будет использование быстрой сортировки либо другой, работающей за время $O(n \log n)$ и базирующейся на непосредственном сравнений ключей. В том числе, некоторые из рассмотренных ранее логарифмических алгоритмов сортировки не требуют дополнительных затрат памяти, что также является несомненным преимуществом перед поразрядной сортировкой.

MSD-сортировка

Основным применением алгоритма MSD-сортировки является упорядочивание данных, длина ключей которых может быть различной. При этом для выравнивания длин ключей нулевые элементы добавляются на позиции младших разрядов, в отличие от LSD-сортировки. Таким образом, рассматриваемый алгоритм базируется на следующем линейном отношении порядка:

$$\forall b, c \in B | b \leq c \Leftrightarrow b_i = c_i \forall i = \overline{1, d} \vee \exists i : b_j = c_j, j > i \wedge b_i < c_i.$$

В этом случае будем говорить, что элемент b *лексикографически меньше либо равен* элементу c. Соответственно порядок данных, построенный с использованием данного отношения, будем называть *лексикографическим*.

[*] Термин LSD происходит от английского *least significant digit*. Термин MSD происходит от английского *most significant digit.*

Заметим, что в процессе поразрядной MSD-сортировки элементы исходного множества будут разбиваться на блоки, каждый из которых – последовательность соседних элементов с одинаковым префиксом ключей. Поэтому при переходе к упорядочиванию по следующему разряду, блоки не будут менять свои места, а будут лишь перемещаться элементы внутри каждого из блоков. Если в качестве вспомогательной процедуры использовать сортировку подсчетом, то можно предложить следующую рекурсивную реализацию алгоритма (процедура 3.2.17):

Процедура 3.2.17. MSD Radix Sort

```
01:  return MSD Radix Sort Impl(A, 1, n, d)
02:  MSD Radix Sort Impl(A, lo, hi, d)
03:        if (lo < hi) then
04:              C[i] = 0, i∈[0, k)
05:              foreach a in A[lo, hi] do
06:                    C[f_d(a)] = C[f_d(a)] + 1
07:              for key = 1 to k – 1 do
08:                    C[key] = C[key] + C[key – 1]
09:              T = C
10:              for i = hi downto lo do
11:                    key = f_d(A[i]),
12:                    B[C[key]] = A[i]
13:                    C[key] = C[key] – 1
14:              A[lo, hi] = B
15:              for key = 0 to k – 1 do
16:                    MSD Radix Sort Impl(A, lo, lo + T[key] - 1, d - 1)
17:                    lo = lo + T[key]
```

Трудоемкость алгоритма MSD-сортировки, как и LSD-алгоритма, есть $O(d \cdot (n + k))$, поскольку сортировка подсчетом линейна относительно числа элементов и количества ключей k.

Необходимо отметить, что алгоритм MSD-сортировки принадлежит к классу устойчивых алгоритмов, а также может быть реализован на многопроцессорной машине, т. к. рекурсивные вызовы сортируют не пересекающиеся между собой блоки исходного массива A.

3.3. Внешняя сортировка

Под *внешней сортировкой* будем понимать те алгоритмы сортировки или их реализации, использующиеся для упорядочивания данных, которые не помещаются в оперативную память ЭВМ в виде массива. Таким образом, алгоритмы внешней сортировки применяются для упорядочивания данных, хранящихся в файлах достаточно большого размера.

Любой из алгоритмов внешней сортировки базируется на алгоритмах внутренней сортировки. Отдельные части массива данных сортируются в оперативной памяти и с помощью специального алгоритма сцепляются в один массив, упорядоченный по ключу.

Наиболее популярным алгоритмом внешней сортировки является сортировка слиянием и ее модификации применительно к упорядочиванию файлов.

3.4. Задачи для самостоятельного решения

1. Рассмотрим перестановку P натуральных чисел от 1 до n и некоторый массив $A[n]$, на элементах которого задано отношение линейного порядка. Разработайте алгоритм, который упорядочит элементы массива A таким образом, что после применения перестановки P к массиву A k раз, элементы массива A будут идти в неубывающем порядке.

2. Рассмотрим n отрезков на прямой, каждый из которых характеризуется двумя координатами x_1 и x_2, обозначающими, соответственно, начало и конец отрезка. Разработайте алгоритм, который позволит найти объединение и пересечение всех отрезков.

3. Рассмотрим массив A из n $(1 \le n \le 10^6)$ натуральных чисел, не превышающих $50'000$. К массиву A применяется M операций, каждая из которых состоит из двух фаз: изменение элемента, находящегося в позиции i; возврат числа инверсий в массиве A. Разработайте алгоритм, который позволит эффективно отвечать на поступающие запросы. Рассмотрите случаи, когда запросы поступают в режиме *on-line* и *off-line* соответственно.

4. Рассмотрим два текста A и B, содержащие, соответственно, n и m символов. Будем предполагать, что каждый из символов текста встречается в нем ровно один раз. Будем говорить, что текст X (возможно, пустой) называется *общим подтекстом* текстов A и B, если X может быть получен вычеркиванием некоторых символов как из A, так и из B. *Наибольшим общим подтекстом* двух текстов называется их общий подтекст, имеющий максимально возможную длину. Требуется определить длину наибольшего общего подтекста заданных текстов A и B.

5. В нулевой момент времени мастеру одновременно поступает n работ, пронумерованных от 1 до n. Для каждой работы i заранее известна следующая информация: время выполнения работы t_i, кратное суткам; штраф p_i за каждые сутки ожидания работы i до момента начала ее выполнения. Одновременно может выполняться только одна работа, и, если мастер приступает к выполнению некоторой работы, то он продолжает выполнять ее, пока не закончит. Таким образом, суммарный штраф, который надо будет уплатить, равен сумме $\sum p_i \cdot s_i$, где s_i – время начала выполнения работы i. Требуется найти такой порядок выполнения работ, при котором штраф будет минимальным.

6. Рассмотрим n программ, которые нужно протестировать. Известно, что для тестирования программы с номером i требуется t_i дней. С другой стороны, тестирование программы с номером i должно быть завершено через d_i дней, начиная с сегодняшнего дня. Например, если тестирование надо завершить послезавтра, то $d_i = 2$. Разработайте алгоритм, который отвергнет минимальное число программ, чтобы не затянуть сроки тестирования оставшихся проектов, а также составит одну из возможных последовательностей их выполнения.

7. Рассмотрим n детей, для каждого из которых известно время, требуемое для того, чтобы уложить ребенка спать, и время, которое он будет находиться во сне. Задача состоит в том, чтобы уложить всех детей спать, причем

делать это разрешается строго последовательно. Разработайте алгоритм, который определит, в каком порядке должны будут засыпать дети, при этом максимизировав время между моментом, когда заснул последний, и проснулся первый из уже заснувших.

8. Разработайте алгоритм, который отсортирует элементы заданного стека S в неубывающем порядке. Предполагается, что детали реализации стека S неизвестны и доступны лишь операции, определяемые АТД «Стек».

9. Разработайте алгоритм, который отсортирует элементы заданного стека S в неубывающем порядке, используя операцию реверса, суть которой заключается в том, чтобы взять фиксированное число элементов с вершины стека и записать их в тот же стек в обратном порядке.

10. На прямой линии определенным образом расположено n отрезков. Из них нужно выбрать наибольшее количество непересекающихся между собой. Модифицируйте полученный алгоритм для задачи, в которой каждому из отрезков задан вес и требуется выбрать набор попарно непересекающихся отрезков с максимальным суммарным весом.

11. Рассмотрим массив A из n натуральных чисел. Обозначим через S следующую сумму: $\sum|A_i - A_{i+1}|$, где $1 \leq i < n$. Требуется найти такую перестановку элементов массива A, при которой сумма S была бы минимальной. Рассмотрите случай, когда разрешается переставлять все элементы массива A, за исключением элемента, находящегося в заданной позиции p $(1 \leq p \leq n)$.

4. Поиск

Рассмотрим множества объектов A и B. *Задача поиска* заключается в том, чтобы для заданной функции $f: A \to B$ определить все элементы $a \in A$ такие, что $f(a) = b$ для заданного элемента b из B. Элементы множества B будем называть *ключами*. Процедуру поиска элементов a будем называть *алгоритмом поиска*. В ряде случаев алгоритмы поиска требуют, чтобы на множестве элементов B было задано отношение линейного порядка $\preccurlyeq$.

Результатом поиска может быть один из двух исходов: либо поиск завершился *успешно*, и искомые объекты были найдены; либо поиск оказался *неудачным*, и искомые объекты не были обнаружены.

Методы поиска могут быть классифицированы несколькими способами. Различают алгоритмы *внешнего* поиска и *внутреннего*. Возможно деление на *статические* и *динамические* методы поиска, где термин статический означает, что содержимое множества A остается неизменным. Термин динамический означает, что множество A изменяется путем вставки (либо удаления) в него элементов. Третья возможная схема классификации методов поиска – их разделение в зависимости от того, на чем они основаны: на сравнении ключей или на некоторых их свойствах. Существуют и другие классификации, рассмотрение которых выходит за рамки данного пособия [9].

Ниже будут рассмотрены методы поиска, которые являются базовыми и могут быть применены к широкому классу математических объектов. Описываемые алгоритмы претендуют на некую универсальность, т. к. они не учитывают внутренней структуры объектов, к которым алгоритм применяется. Следует акцентировать внимание на том, что приведенные алгоритмы не имеют ничего общего с алгоритмами поиска на графах либо строковыми алгоритмами, идеи которых изначально базируются на особенностях объектов, их определяющих.

4.1. Элементарные методы поиска

Будем говорить, что метод поиска является *элементарным*, если для успешного завершения алгоритма, лежащего в его основе, достаточно, чтобы элементы множества A хранились в элементарной структуре данных. В этом разделе будем считать, что элементы из множества A записаны в массив и занумерованы натуральными числами от 1 до $|A|$, где $|A|$ – число элементов в A. В дальнейшем будем предполагать, что $|A| = n, n \in \mathbb{N}$. Если же алгоритм поиска работает не с дискретными объектами, а, например, с непрерывной функцией на заданном отрезке, то это будет специально оговорено.

Ниже будут рассмотрены базовые алгоритмы поиска – линейный, двоичный, тернарный с его модификацией, базирующейся на использовании «золотого сечения».

4.1.1. Линейный поиск

Линейный поиск является простейшим методом поиска, идея которого заключается в последовательном просмотре элементов множества A. Условно алгоритм линейного поиска может быть описан при помощи процедуры 4.1.1:

```
Процедура 4.1.1. Linear Search
01:  Linear Search Impl(A[1, n], b)
02:        for index = 1 to n do
03:              if f(A[index]) = b then
04:                    return index
```

Проанализируем время работы приведенного алгоритма. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(1) = 1, \\ T(n) = T(n-1) + O(1),\ n > 1. \end{cases} \tag{4.1}$$

Решением данного соотношения является функция $T(n) = O(n)$, поэтому сложность алгоритма линейного поиска есть $O(n)$, где n – число элементов в A. Основным недостатком метода является просмотр всех элементов множества A, число которых может быть достаточно велико. Следовательно, данный алгоритм не используется в ситуациях, когда процедуру поиска требуется повторять неоднократно. Линейный поиск применим как к упорядоченным множествам, так и к множествам, на которых не задано отношение порядка, а только ключ поиска. Делаем вывод, что метод линейного поиска является наиболее универсальной процедурой по сравнению с методами, рассмотреными ниже.

4.1.2. Двоичный поиск

Двоичный поиск (или *бинарный*, *дихотомия*) – метод поиска, базирующийся на том, что на множестве B задано отношение линейного порядка $\preccurlyeq$, в соответствие с которым элементы множества могут быть упорядочены.

Идея алгоритма двоичного поиска заключается в следующем. Пусть элемент с искомым ключом находится в позиции i. Тогда:

1) для всех j таких, что $i < j$ выполняется $f(A_i) \preccurlyeq f(A_j)$;

2) для всех j таких, что $j < i$ выполняется $f(A_j) \preccurlyeq f(A_i)$.

Следовательно, выбирая заведомо хорошую позицию i, которая будет проверяться на совпадение с ключом, двоичный поиск позволяет отбросить от дальнейшего анализа значительную часть элементов множества A. Хорошей позицией в случае двоичного поиска является средний элемент.

Предположив, что на множестве ключей B задано линейное отношение порядка «меньше либо равно», алгоритм двоичного поиска можно описать следующим образом (процедура 4.1.2).

Зачастую данные, которые поступают на вход двоичному поиску, не являются упорядоченными. В этом случае непосредственное применение двоичного поиска может привести к некорректному результату. Поэтому для достижения гарантированного успеха первое, что необходимо сделать, это проверить, являются ли элементы множества A отсортированными по ключу. Такую проверку можно осуществить, используя один проход по массиву, последовательно сравнивая соседние элементы (процедура 4.1.3).

Процедура 4.1.2. Binary Search

```
01:  Binary Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > 2) do
04:              middle = lo + ⌊(hi - lo)/2⌋
05:              if f(A[middle]) = b then
06:                    return middle
07:              if f(A[middle]) < b then lo = middle + 1
08:              if f(A[middle]) > b then hi = middle - 1
09:        return Linear Search(A[lo, hi], b)
```

Процедура 4.1.3. Check Binary Search Prerequisites

```
01:  for index = 1 to n - 1 do
02:        if f(A[index]) > f(A[index - 1]) then
03:              return false
04:  return true
```

Проанализируем время работы двоичного поиска. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(1) = 1, \\ T(n) = T(n/2) + O(1),\ n > 1. \end{cases} \tag{4.2}$$

Решением данного соотношения является функция $T(n) = O(\log n)$. Поэтому сложность алгоритма двоичного поиска есть $O(\log n)$, где n – число элементов в A. Основным достоинством метода является его время выполнения в худшем случае, которое составляет $O(\log n)$ операций. Исходя из асимптотики времени выполнения алгоритма двоичного поиска, можно сделать вывод о том, что он очень хорошо подходит для задач, в которых процедуру поиска требуется повторять многократно для одного и того же множества ключей. Существуют ситуации, в которых линейный поиск будет приносить ощутимый выигрыш во времени выполнения. Одной из них является такая, при которой процедуру поиска требуется осуществлять не более чем $\lfloor \log_2 n \rfloor$ раз, при условии, что массив данных не является упорядоченным и ключи не могут быть отсортированы за линейное время. Для применения алгоритма двоичного поиска необходимо применение алгоритма сортировки, трудоемкость которого, по крайней мере, составит $O(n \log n)$.

Левосторонний / правосторонний двоичный поиск

В ряде случаев требуется определить не только вхождение элемента с заданным ключом в искомый массив данных, но и левую и правую границы, между которыми находятся все элементы из A, удовлетворяющие заданному критерию. В этом случае принято говорить о *левостороннем* и *правостороннем* двоичном поиске. Целью левостороннего является нахождение первого вхождения элемента с заданным ключом. Соответственно, целью правостороннего – нахождение последнего вхождения искомого ключа в заданный массив. Если элемент с заданным ключом не обнаружен, то левосторонний / правосторонний

поиск должен вернуть такой индекс i, для которого справедливо следующее соотношение:

$$f(A[i-1]) < b < f(A[i]). \quad (4.3)$$

Данное соотношение задает такую позицию i, в которую элемент с ключом b может быть вставлен без нарушения заданного отношения порядка.

С точки зрения практики наиболее полезными являются следующие соотношения:

для левостороннего поиска:

$$f(A[i-1]) < b \leq f(A[i]);$$

для правостороннего поиска:

$$f(A[i-1]) \leq b < f(A[i]).$$

Следует сказать, что возвращаемый индекс может варьироваться в зависимости от требований задачи. Несмотря на выбор критериев, оба метода выполняются за $O(\log n)$ шагов в худшем случае. Процедуры левостороннего и правостороннего поиска приведены ниже (процедура 4.1.4):

Процедура 4.1.4. Leftmost/Rightmost Binary Search

```
01:  Leftmost Binary Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > 2) do
04:              middle = lo + ⌊(hi - lo)/2⌋
05:              if f(A[middle]) < b then
06:                    lo = middle + 1
07:              else
08:                    hi = middle
09:        return Linear Search(A[lo, hi], b)
10:
11:  Rightmost Binary Search Impl(A[1, n], b)
12:        lo = 1, hi = n
13:        while (hi – lo > 2) do
14:              middle = lo + ⌊(hi - lo)/2⌋
15:              if f(A[middle]) > b then
16:                    hi = middle
17:              else
18:                    lo = middle + 1
19:        return Linear Search(A[lo, hi], b)
```

Вещественный двоичный поиск

Вещественный двоичный поиск используется для решения уравнений вида $f(x) = y$ на отрезке $[l, u]$ с заданной точность ε, где $f(x)$ – монотонная функция, множество A –область определения функции f, множество B – область значений функции f.

Будем предполагать, что $f(x)$ – неубывающая на заданном отрезке функция. Применив идеи для дискретного двоичного поиска, рассмотренные выше,

процедуру вещественного двоичного поиска можно описать следующим образом (процедура 4.1.5):

Процедура 4.1.5. Real Binary Search

```
01:  Real Binary Search Impl([l, u], y)
02:        lo = l, hi = u
03:        while (lo + ε < hi) do
04:              middle = (lo + hi) / 2
05:              if f(middle) + ε < y then
06:                    lo = middle
07:              else
08:                    hi = middle
09:        return (lo + hi) / 2
```

Время работы приведенной процедуры есть $O\left(\log\frac{u-l}{\varepsilon}\right)$ в худшем случае. Отметим, что применение вещественного двоичного поиска в том виде, в котором он приведен в процедуре (4.1.5) может привести к зацикливанию по причине плохого выбора точности ε. Во избежание этого используется итеративная версия процедуры (4.1.5), приведенная ниже (процедура 4.1.6):

Процедура 4.1.6. Iterative Real Binary Search

```
01:  Iterative Real Binary Search Impl([l, u], y)
02:        lo = l, hi = u
03:        for iteration = 1 to number of iterations do
04:              middle = (lo + hi) / 2
05:              if f(middle) + ε < y then
06:                    lo = middle
07:              else
08:                    hi = middle
09:        return (lo + hi) / 2
```

Выбор числа итераций зависит от специфики задачи и от точности, которой должно удовлетворять найденное значение. На практике же обычно достаточно 40–50 прогонов.

Среди задач, которые можно эффективно решать при помощи вещественного двоичного поиска следует отметить задачу нахождения вещественного корня *n*-ой степени из числа x. В этом случае нижней границей поиска будет число 1, а верхней – число x.

4.1.3. Тернарный поиск

Тернарный поиск (или *троичный*) – метод, который используется для поиска максимума / минимума выпуклой / вогнутой на заданном отрезке $[l,u]$ функции $f(x)$.

Пусть требуется найти максимум функции $f(x)$. Алгоритм базируется на том факте, что должно существовать некоторое значение x, чтобы:

1) для всех x_1, x_2, таких, что $l \le x_1 < x_2 \le x$, выполнялось $f(x_1) < f(x_2)$;
2) для всех x_1, x_2, таких, что $x \le x_1 < x_2 \le u$, выполнялось $f(x_1) > f(x_2)$.

С учетом вышеприведенного, алгоритм тернарного поиска можно описать следующим образом (процедура 4.1.7):

Процедура 4.1.7. Real Ternary Search

```
01:  Real Ternary Search Impl([l, u])
02:        lo = l, hi = u
03:        while (lo + ε < hi) do
04:              loThird = lo + (hi – lo) / 3
05:              hiThird = hi – (hi – lo) / 3
06:              if f(loThird) + ε < f(hiThird) then
07:                    lo = loThird
08:              else
09:                    hi = hiThird
10:        return (lo + hi) / 2
```

Если же алгоритм тернарного поиска применяется к дискретным множествам, то соответствующая подпрограмма будет претерпевать незначительные изменения (процедура 4.1.8)

Процедура 4.1.8. Ternary Search

```
01:  Ternary Search Impl(A[1, n])
02:        lo = 1, hi = n
03:        while (hi – lo > 3) do
04:              loThird = lo + (hi – lo) / 3
05:              hiThird = hi – (hi – lo) / 3
06:              if f(A[loThird]) < f(A[hiThird]) then
07:                    lo = loThird
08:              else
09:                    hi = hiThird
10:        return Linear Search(A[lo, hi])
```

Проанализируем время работы тернарного поиска. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(n) = O(1), & n \le 3, \\ T(n) = T(2/3 \cdot n) + O(1), & n > 3. \end{cases} \tag{4.4}$$

Решением данного соотношения является функция $T(n) = O(\log_{1.5} n)$. Если алгоритм тернарного поиска применяется к функциям, работающим с непрерывными объектами, то сложность составит $O\left(\log_{1.5} \frac{u-l}{\varepsilon}\right)$.

Отметим, что недостатком тернарного поиска является сравнение значений функции в двух точках, определяющих границы каждой из третьих. Однако существуют техники, которые позволяют получить значительный прирост в скорости, путем «обсчета» каждой из рассматриваемых точек только один раз. Одна из таких техник будет рассмотрена ниже.

Поиск с помощью «золотого сечения»

Скрытую константу в оценке времени работы тернарного поиска можно значительно уменьшить, если вместо деления отрезка на три равные части, делить его в отношении «золотого сечения».

Пусть требуется найти, как и в предыдущем разделе, максимум функции $f(x)$, выпуклой на заданном отрезке $[l,u]$. Будем делить отрезок $[l,u]$ точками x_1 и x_2 $(x_1 < x_2)$ так, чтобы выполнялись следующие соотношения

$$\begin{cases} x_1 = l + \varphi \cdot (u - l), & x_1 = x_2 - \varphi \cdot (x_2 - l), \\ x_2 = u - \varphi \cdot (u - l), & x_2 = x_1 + \varphi \cdot (u - x_1). \end{cases} \quad (4.5)$$

Учитывая, что неизвестный параметр $\varphi \in (0,1)$, получим единственное решение системы уравнений (4.5) в виде $\varphi = 2 - \frac{1+\sqrt{5}}{2}$ * Соответствующая подпрограмма приведена ниже (процедура 4.1.9):

Процедура 4.1.9. Gold Section Ternary Search

```
01:  Gold Section Ternary Search Impl([l, u])
02:        φ = 2 − (1+√5)/2
03:        lo = l, hi = u
04:        if (lo + ε < hi) do
05:              loThird = lo + φ·(hi – lo)
06:              hiThird = hi - φ·(hi – lo)
07:              fLoThird = f(loThird)
08:              fHiThird = f(hiThird)
09:              do
10:                    if fLoTthird + ε < fHiThird then
11:                          lo = loThird
12:                          loThird = hiThird
13:                          hiThird = hi - φ·(hi – lo)
14:                          fLoThird = fHiThird
15:                          fHiThird = f(hiThird)
16:                    else
17:                          hi = hiThird
18:                          hiThird = loThird
19:                          loThird = lo + φ·(hi – lo)
20:                          fHiThird = fLoThird
21:                          fLoThird = f(loThird)
22:              while (lo + ε < hi)
23:        return (lo + hi) / 2
```

4.1.4. Интерполяционный поиск

Интерполяционный поиск, как и двоичный, используется для поиска данных в упорядоченных массивах. Их отличие состоит в том, что вместо деления области поиска на две примерно равные части, интерполяционный поиск производит оценку новой области поиска по расстоянию между ключом и текущими значениями граничных элементов (процедура 4.1.10):

* Отсюда название метода «золотого сечения».

Процедура 4.1.10. Interpolation Search

```
01:  Interpolation Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > threshold) do
04:              x = lo + (b – A[lo]) * (hi – lo) / (A[hi] – A[lo])
05:              if A[x] = b then
06:                    return x
07:              if A[x] < b then lo = x + 1
08:              if A[x] > b then hi = x - 1
09:        return Binary Search(A[lo, hi], b)
```

Использование интерполяционного поиска оправдано в том случае, когда данные в массиве, в котором происходит поиск, распределены достаточно равномерно. Тогда трудоемкость приведенной процедуры может составить $O(\log\log n)$, что асимптотически лучше трудоемкости двоичного поиска $O(\log n)$. Если данные распределены не достаточно равномерно, а, например, экспоненциально, то трудоемкость интерполяционного поиска может составить $O(n)$ в худшем случае. На практике интерполяционный поиск обычно используется для поиска значений в больших файлах данных, причем поиск продолжается до тех пор, пока не будет достигнут некий пороговый диапазон, в котором находится искомое значение. После этого обычно применяется двоичный либо линейный поиск.

В качестве примера использования интерполяционного поиска рассмотрим следующую задачу. Пусть есть словарь, состоящий из всех слов русского языка, упорядоченных по алфавиту. Задача состоит в том, чтобы достаточно быстро найти некоторое слово $\alpha_1\alpha_2 \ldots \alpha_n$. Очевиден тот факт, что человек не будет искать требуемое слово, используя идеи двоичного поиска, а начнет со страницы, которая содержит слова с префиксом искомого слова. В зависимости от места остановки поиска дальнейшие действия могут пропустить либо небольшое количество страниц, либо количество страниц, существенно большее половины допустимого интервала. Таким образом, основное отличие интерполяционного поиска от рассмотренных выше методов состоит в том, что в случае равномерно распределенных данных интерполяционный поиск учитывает разницу между «немного больше» и «существенно больше», тем самым позволяя сократить время поиска искомых данных до минимума.

4.2. Деревья поиска

Рассмотрим корневое бинарное дерево B, с каждым узлом которого ассоциирован некоторый объект, называющийся *ключом узла*. Предположим, что на множестве ключей задано отношение линейного порядка и все ключи являются различными. Дерево B является *бинарным деревом поиска*, если на нем определены операции интерфейса АТД «Бинарное дерево» и для каждого его узла выполнено следующее условие: ключ узла больше, чем любой из ключей его левого поддерева, и меньше, чем любой из ключей его правого поддерева.[*] Усло-

[*] В англоязычно литературе используется термин *BST – binary search tree.*

вие, накладываемое на порядок ключей в дереве, в дальнейшем будем называть *основным свойством* бинарного дерева поиска.

Для хранения BST в памяти ЭВМ воспользуемся подходом, описанным в разд. 2.2.5, процедура (2.2.20). В дальнейшем значение узла будем отождествлять с ключом, которому соответствует данное значение. Тогда процедура создания дерева может выглядеть следующим образом (процедура 4.2.1):

Процедура 4.2.1. BST: Make BST

```
01:  Make BST Impl(key, leftTree, rightTree)
02:        return binarySearchTree {
03:              root = binarySearchTreeNode {
04:                    key = key
05:                    left  = leftTree
06:                    right = rightTree
07:              }
```

Отметим, что требование к уникальности ключей не является строгим, поскольку узлы, ключи которых совпадают с ключом корня дерева, могут быть размещены либо слева, либо справа.

В соответствие с основным свойством BST процедура поиска может выглядеть следующим образом (процедура 4.2.2):

Процедура 4.2.2. BST: Contains

```
01:  Contains Impl(B, key)
02:        if not Is Empty(B) then
03:              R = Root(B)
04:              with R do
05:                    if key < Key(R) then return Contains(left , key)
06:                    if key > Key(R) then return Contains(right, key)
07:              return true
08:        return false
```

Операция вставки в дерево также не представляет сложностей. В этом случае достаточно спуститься до некоторого листа дерева и создать у него левого либо правого потомка, в зависимости от значения ключа, который вставляется в дерево (процедура 4.2.3):

Процедура 4.2.3. BST: Insert

```
01:  Insert Impl(B, key)
02:        if not Contains(B, key) then
03:              return IRec(B)
04:        return B
05:
06:        IRec Impl(B)
07:              if Is Empty(B) then
08:                    return Make BST(key, ∅, ∅)
09:              R = Root(B)
10:              with R do
11:                    if key < Key(R) then
12:                          return Make BST(Key(R), IRec(left), right)
13:                    if key > Key(R) then
14:                          return Make BST(Key(R), left, IRec(right))
```

Трудоемкость приведенного алгоритма есть $O(h)$, где h – высота дерева, так как в худшем случае вставляемое значение будет «прицеплено» к самому глубокому листу.

Перед рассмотрением процедуры удаления узла дерева с заданным ключом, введем ряд определений. *Преемником* узла n в заданном бинарном дереве поиска назовем либо пустой узел, если ключ узла n является наибольшим, либо узел, ключ которого непосредственно следует после ключа узла n в отсортированной последовательности ключей. *Предшественником* узла n в заданном бинарном дереве поиска назовем либо пустой узел, если ключ узла n является наименьшим, либо узел, ключ которого непосредственно следует перед ключом узла n в отсортированной последовательности ключей.[*] Для того чтобы найти преемника либо предшественника заданного узла, можно воспользоваться процедурой симметричного обхода, которая применительно к бинарным деревьям поиска будет иметь вид (процедура 4.2.4):

Процедура 4.2.4. BST: InOrder

```
01:  InOrder Impl(B)
02:        if not Is Empty(B) then
03:              R = Root(B)
04:              with R do
05:                    InOrder(left)
06:                    process(R)
07:                    InOrder(right)
```

Несложно видеть, что узлы дерева B в процессе симметричного обхода будут обработаны в порядке возрастания ключей. Непосредственное применение процедуры (4.2.4) к нахождению преемника и предшественника может привести к временным затратам $O(n)$, где n – число узлов дерева. Для того чтобы добиться трудоемкости $O(h)$, преемника и предшественника нужно находить непосредственно. Детали, касающиеся процедуры нахождения преемника приведены ниже (процедура 4.2.5). Процедура нахождения предшественника не приводится по причине сходства с (4.2.5).

Процедура 4.2.5. BST: Successor

```
01:  Successor Impl(R)
02:        if not Is Empty(Right(R)) then
03:              successor = Root(Right(R))
04:              while not Is Empty(Left(successor)) do
05:                    successor = Root(Left(successor))
06:              return successor
07:        else
08:              successor = Parent(R)
09:              while successor ≠ ∅ do
10:                    if Root(Left(successor)) = R then
11:                          break
12:                    R = successor, successor = Parent(R)
13:              return successor
```

[*] В англоязычной литературе соответственно используются термины *successor* и *predecessor*.

Для того чтобы удалить узел из бинарного дерева поиска, соответствующий заданному ключу, требуется найти необходимый узел и заменить его предшественником (в этом случае говорят о *левом удаление*) либо преемником (в этом случае говорят о *правом удалении*). Если у удаляемого узла только один потомок или их нет вовсе, то узел заменяется соответствующим потомком. Процедура правого удаления приведена ниже (процедура 4.2.6):

Процедура 4.2.6. BST: Delete

```
01:  Delete Impl(B, key)
02:        if Contains(B, key) then
03:              return DRec(B)
04:        return B
05:
06:        DRec Impl(B)
07:              R = Root(B)
08:              if key = Key(R) then
09:                    if Is Leaf(R) then
10:                          return ∅
11:                    with R do
12:                        if Is Empty(left)  then return right
13:                        if Is Empty(right) then return left
14:                        key = Key(Successor(R))
15:                        return Make BST(key, left, Delete(right, key))
16:              with R do
17:                  if key < Key(R) then
18:                        return Make BST(Key(R), DRec(left), right)
19:                  if key > Key(R) then
20:                        return Make BST(Key(R), left, DRec(right))
```

Несложно видеть, что процедура удаления узла дерева будет иметь трудоемкость $O(h)$, поскольку трудоемкость всех этапов (поиска узла, нахождения и удаления преемника) не превосходит $O(h)$.

Следует отметить, что для бинарных деревьев поиска, как и для списков, определяются операции *соединения* и *разделения*. Операция соединения соединяет два дерева в одно, предполагая, что ключи одного дерева строго меньше, чем ключи другого. Операция разделения разделяет заданное дерево на два поддерева по заданному ключу таким образом, чтобы ключи одного были строго меньше, чем ключи другого. Обе операции имеют трудоемкость $O(h)$.

Так же, как и для линейных списков, операции вставки и удаления в бинарное дерево поиска могут быть реализованы путем применения конечного числа операций разделения и соединения. Так как каждая из них имеет трудоемкость $O(h)$, то и трудоемкость операций, реализованных посредством соединения и разделения составит $O(h)$. Отметим, что на практике непосредственная реализация операций вставки и удаления является наиболее предпочтительной, так как скрытая константа в асимптотике O в этом случае будет значительно меньше.

Если применять бинарные деревья поиска в том виде, в каком они описаны выше, то трудоемкость всех операций может составить $O(n)$ в случае, если дерево выродиться в цепь (например, при последовательной вставке возрастающей последовательности ключей). Для того чтобы дерево оставалось сбалансированным, т. е. его высота удовлетворяла бы оценке $O(\log n)$, используются специальные виды бинарных деревьев поиска, каждое из которых придерживается фиксированного набора инвариантов, сохранение которых позволяет добиться трудоемкости $O(\log n)$ для всех операций.

Под операциями *балансировки* будем понимать такие операции над бинарным деревом поиска, которые позволяют сохранить инварианты, определенные на дереве, с целью поддержания его в сбалансированном состоянии.

К основным операциям балансировки относятся операции *вращения* (или *поворота*), среди которых выделяют *малые вращения* (рис. 4.1) и *большие вращения* (или *двойные вращения*) (рис. 4.2, 4.3). Идея их заключается в том, чтобы изменить некоторые связи «предок – потомок» в дереве, при этом сохранив основное свойство бинарных деревьев поиска. Как среди малых, так и среди больших вращений выделяют *правые* и *левые*. Процедуры малых вращений приведены ниже (4.2.7).

Процедура 4.2.7. BST: Rotate Left, Rotate Right

```
01:  Rotate Left Impl(B)
02:        R = Root(B)
03:        with R do
04:              if Is Empty(right) then
05:                    return B
06:              X = Root(right)
07:              leftTree  = Make BST(Key(R), left, Left(X))
08:              return Make BST(Key(X), leftTree, Right(X))
09:
10:  Rotate Right Impl(B)
11:        R = Root(B)
12:        with R do
13:              if Is Empty(left) then
14:                    return B
15:              X = Root(left)
16:              rightTree = Make BST(Key(R), Right(X), right)
17:              return Make BST(Key(X), Left(X), rightTree)
```

Несложно видеть, что обе операции имеют трудоемкость $O(1)$, так как они затрагивают фиксированное число узлов дерева.

Большие вращения базируются на малых вращения. Большое левое вращение вначале совершает правый поворот, а затем левый (процедура 4.2.8).

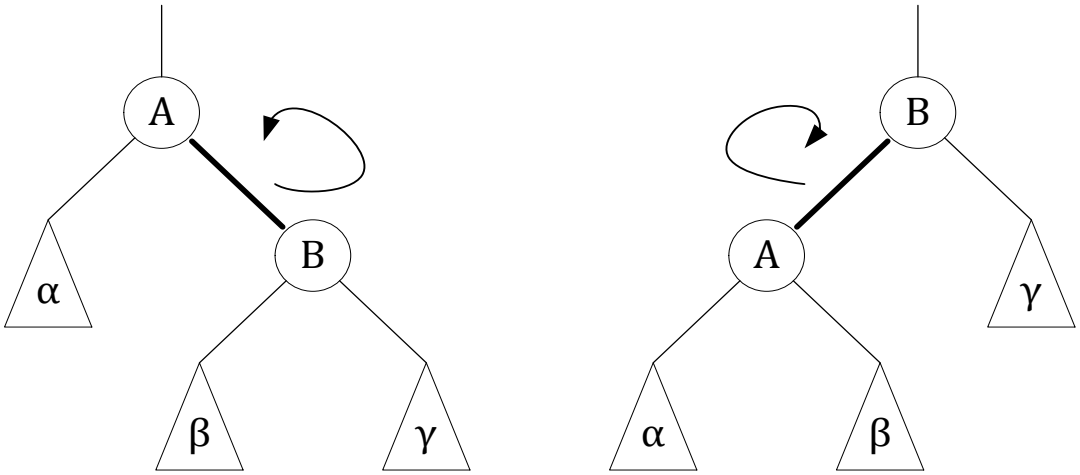

Рис. 4.1. Малые левое и правое вращения

Процедура 4.2.8. BST: Double Rotate Left

```
01:  Double Rotate Left Impl(B)
02:        R = Root(B)
03:        with R do
04:              if Is Empty(right) then
05:                    return B
06:              rotated = Make BST(Key(R), left, Rotate Right(right))
07:              return Rotate Left(rotated)
```

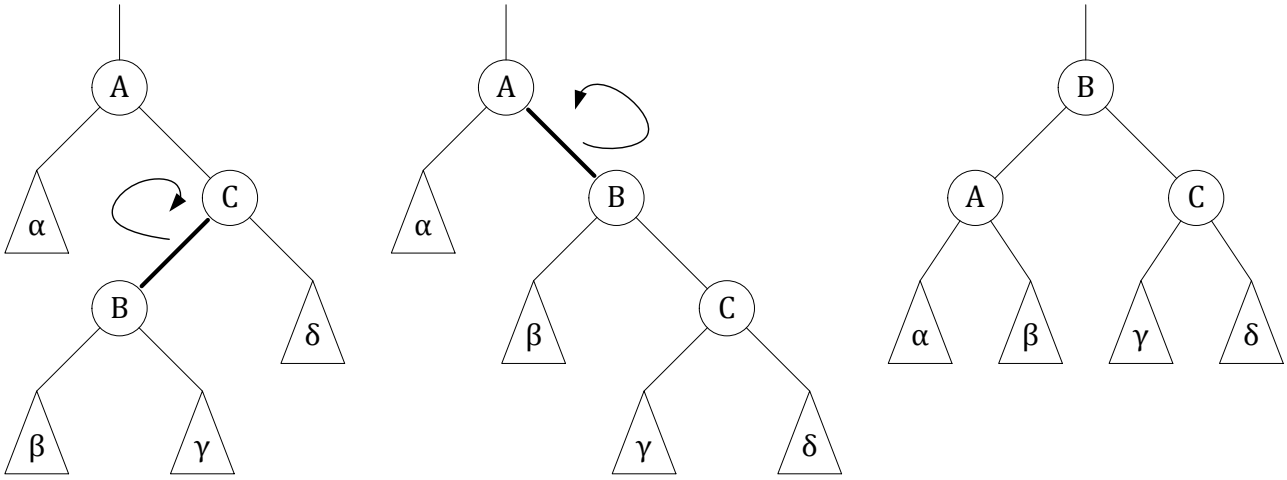

Рис. 4.2. Большое левое вращения

Большое правое вращение вначале совершает левый поворот, а затем правый (процедура 4.2.9).

Процедура 4.2.9. BST: Double Rotate Right

```
01:  Double Rotate Right Impl(B)
02:        R = Root(B)
03:        with R do
04:              if Is Empty(left) then
05:                    return B
06:              rotated = Make BST(Key(R), Rotate Left(left), right)
07:              return Rotate Right(rotated)
```

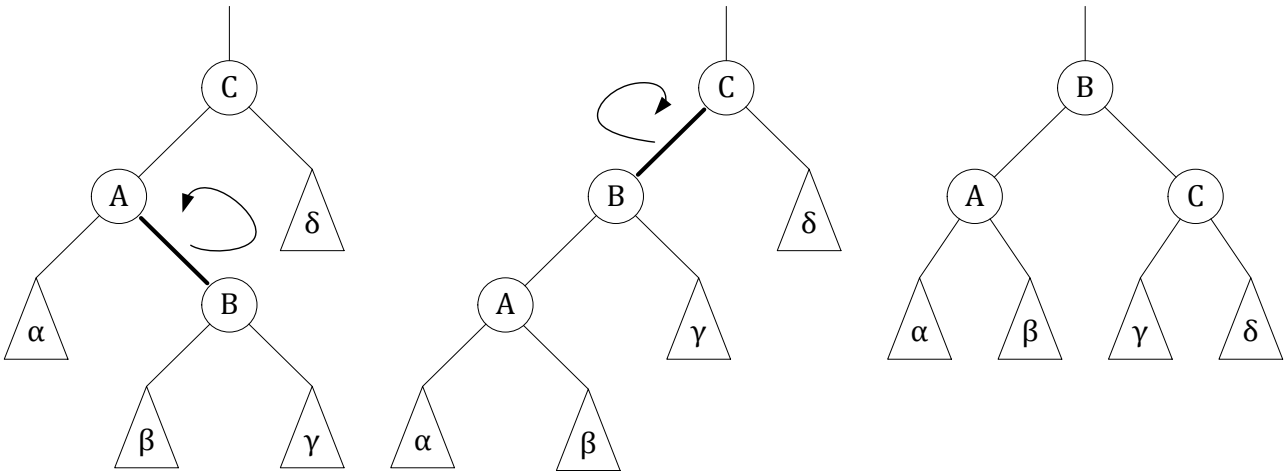

Рис. 4.3. Большое правое вращения

Так как малые вращения имеют трудоемкость $O(1)$, то и трудоемкость больших вращений составит $O(1)$.

В зависимости от свойств, которыми обладает бинарное дерево поиска, операции вращения могут претерпевать те либо иные изменения (например, вращения на AVL-дереве, на красно-черном дереве и т. д.). Общей идеей, объединяющей все типы вращений, является подъем (или вытягивание) поддеревьев с низких уровней на верхние, что позволяет достичь сравнительной сбалансированности всего дерева в целом.

Помимо операций вращения существует ряд других операций и техник, которые позволяют поддерживать дерево в сбалансированном состоянии. Отметим лишь, что большинство из них базируется на специфических свойствах бинарных деревьев, на которых эти операции определены.

4.2.1. AVL-дерево

AVL-дерево – сбалансированное бинарное дерево поиска, у которого высота поддеревьев каждого из узлов отличается не более, чем на 1.*

Несложно показать, что высота сбалансированного AVL-дерева с n узлами будет удовлетворять следующей оценке:

$$\log_2(n+1) - 1 \le h < 1{,}4404 \cdot \log_2(n+1) - 1{,}3277. \qquad (4.6)$$

Поскольку AVL-дерево является бинарным, то количество узлов не должно превышать величины $2^{h+1} - 1$. Следовательно, $h \ge \log_2(n+1) - 1$. Для доказательства левой части неравенства составим рекуррентное соотношение для числа узлов дерева высотой h:

$$\begin{cases} T(0) = 1,\ T(1) = 2, \\ T(h) = T(h-1) + T(h-2) + 1,\ n > 1. \end{cases} \qquad (4.7)$$

Для того чтобы найти замкнутый вид данного уравнения, сделаем замену $T(h) + 1 = F(h)$. Получим уравнение

* Аббревиатура *AVL* происходит от фамилий советских ученых Г. М. Адельсон-Вельский и Е. М. Ландис, которые впервые предложили данный тип деревьев в 1962 г.

$$\begin{cases} F(0) = 1,\ F(1) = 3, \\ F(h) = F(h-1) + F(h-2),\ n > 1, \end{cases} \tag{4.8}$$

решив которое найдем $F(h) = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h$.

Следовательно, для числа узлов AVL-дерева высотой h справедлива следующая оценка:

$$n \geq T(h) = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h - 1. \tag{4.9}$$

Если учесть, что $\left|\frac{1-\sqrt{5}}{2}\right| < 1$, то полученное неравенство можно переписать в виде

$$n \geq \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h - 1,$$

откуда $h < \log_{\varphi} \frac{n+1}{1+\frac{2}{\sqrt{5}}} \approx 1{,}4404 \cdot \log_2(n+1) - 1{,}3277$.

Видим, что высота AVL-дерева есть $O(\log n)$, где n – число узлов дерева. Полученная оценка утверждает тот факт, что высота AVL-дерева никогда не превысит высоту идеально сбалансированного дерева более чем на 45 %.

Используя операции вращения, достаточно просто поддерживать инвариант, определенный для AVL-дерева.

Рассмотрим некоторое дерево B. Будем говорить, что левое его поддерево *тяжелее* правого или наоборот, если разность между их высотой больше единицы. Процедуры определения тяжести поддеревьев заданного дерева приведены ниже (процедура 4.2.10):

Процедура 4.2.10. AVL-tree: Invariant Ops

```
01:  Is Le Heavy Impl(B) return Score(B) > 1
02:  Is Ri Heavy Impl(B) return Score(B) < 1
03:
04:  Is Balanced Impl(B)
05:        return |Score(B)| ≤ 1
06:
07:  Score Impl(B)
08:        R = Root(B)
09:        with R do
10:              return Height(left) - Height(right)
```

Будем предполагать, что трудоемкость процедур 4.2.10 есть $O(1)$. Этого можно добиться, если в каждом узле дерева хранить его высоту. Операция балансировки будет заключаться в том, чтобы определить, нуждается ли дерево в балансировке вообще и, если это необходимо, осуществить некоторый поворот. Левое или правое вращение выбирается исходя из того, какое из поддеревьев тяжелее. Так, в случае левого поддерева требуется применить правый поворот, в случае правого поддерева – левый поворот. В зависимости от того, как сба-

лансированы поддеревья более тяжелого дерева, выбирается большой либо малый поворот (процедура 4.2.11):

Процедура 4.2.11. AVL-tree: Make Balanced

```
01:  Make Balanced Impl(B)
02:        if not Is Balanced(B) then
03:              R = Root(B)
04:              with R do
05:                    if Is Le Heavy(B) then
06:                          if Is Ri Heavy(left) then
07:                                return Double Rotate Right(B)
08:                          return Rotate Right(B)
09:                    if Is Ri Heavy(B) then
10:                          if Is Le Heavy(right) then
11:                                return Double Rotate Left(B)
12:                          return Rotate Left(B)
13:        return B
```

Операцию балансировки требуется выполнять всякий раз, когда в дереве происходят изменения. В операциях вставки и удаления перед непосредственным возвратом из процедуры необходимо проверить, является ли дерево сбалансированным, и, если это не так, то осуществить балансировку. В качестве примера приведем процедуру, описывающую операцию вставки в AVL-дерево (процедура 4.2.12).

Процедура 4.2.12. AVL-tree: Insert

```
01:  Insert Impl(B, key)
02:        if not Contains(B, key) then
03:              return IRec(B)
04:        return B
05:
06:        IRec Impl(B)
07:              if Is Empty(B) then
08:                    return Make BST(key, ∅, ∅)
09:              R = Root(B)
10:              with R do
11:                    if key < Key(R) then
12:                          result = Make BST(Key(R), IRec(left), right)
13:                    if key > Key(R) then
14:                          result = Make BST(Key(R), left, IRec(right))
15:              return Make Balanced(result)
```

Операция удаления по сравнению с приведенной для общего бинарного дерева поиска, будет претерпевать изменения, аналогичные операции вставки. Операция поиска в AVL-дереве будет схожа с операцией поиска для BST (процедура 4.2.2).

Недостатком рассмотренных AVL-деревьев является многократное выполнение операции балансировки для операций вставки и удаления элемента, что может привести к росту скрытой константы в оценке $O(\log n)$. Таким образом, несмотря на сравнительно высокую степень сбалансированности,

AVL-дерево не получило достаточно широкого распространения на практике. Экспериментальным путем было выяснено, что в среднем одна операция балансировки приходится на каждые две операции вставки и на каждые пять операций удаления.

4.2.2. (2-4)-дерево

(2-4)-дерево – сбалансированное дерево поиска, у которого каждый из узлов может содержать не более четырех дочерних элементов и высота поддеревьев одного и того же уровня является одинаковой (рис. 4.4).

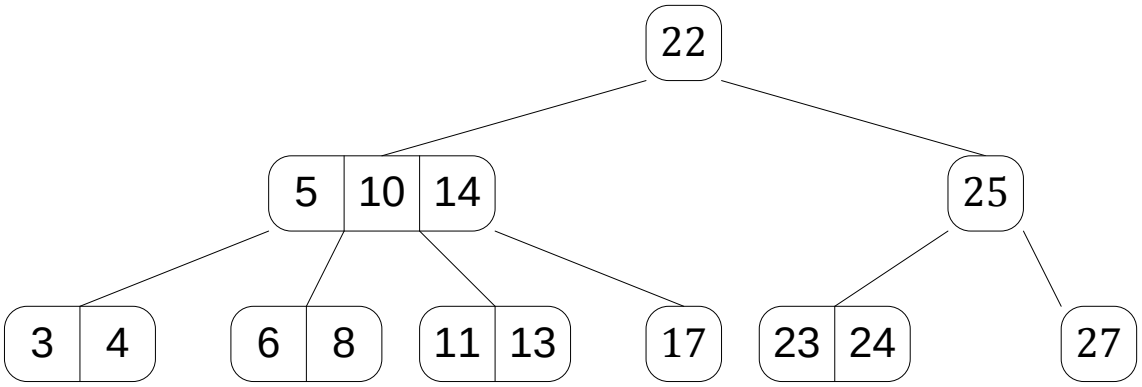

Рис. 4.4. Пример (2-4)-дерева

Согласно определению (2-4)-дерева, каждый из его узлов можно отнести к одному из трех типов: *2-узел*, *3-узел* либо *4-узел*. Отдельно выделим тип *листового узла*, который также относится к одному из трех типов, перечисленных выше, но у которого нет дочерних элементов (рис. 4.5).

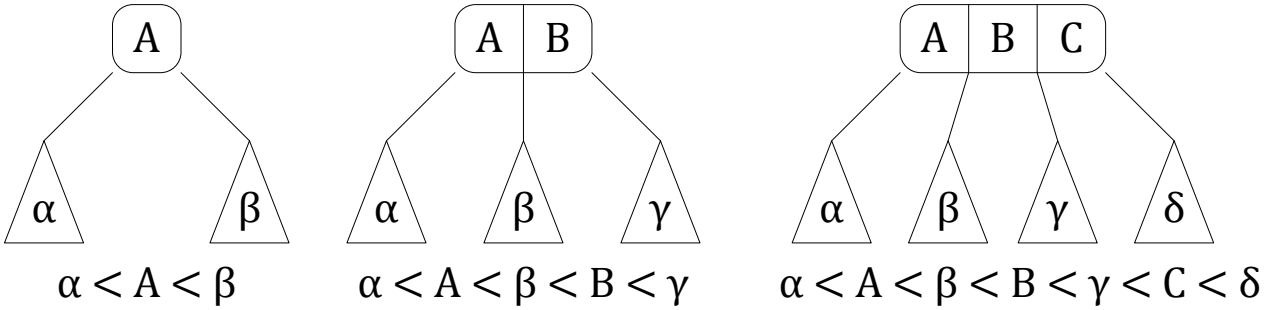

Рис. 4.5. Узлы (2-4)-дерева

Отдельно выделяются 1-узел и 5-узел, имеющие в точности один и пять потомков соответственно. Узлы данного типа могут быть использованы в целях упрощения реализации некоторых из операций, определенных над (2-4)-деревом. Наличие же 1- и 5-узлов в дереве не допускается по определению.

К основным операциям над (2-4)-деревьями относятся операции поиска элемента с заданным ключом, вставки и удаления элемента. В силу свойства сбалансированности (2-4)-дерева, все операции будут иметь трудоемкость $O(h)$, где высота дерева h удовлетворяет следующей оценке:

$$\frac{1}{2}\log_2(n+1) - 1 \le h \le \log_2(n+1) - 1 \Rightarrow h = O(\log n). \qquad (4.10)$$

Для доказательства неравенства (4.10) достаточно рассмотреть дерево, у которого все узлы являются либо 4-узлами, либо 2-узлами.

К вспомогательным операциям на (2-4)-дереве относятся операции *разделения* узла, *слияния* двух узлов и операция *перемещения*.

Операция *разделения* может применяться либо к 4-узлу (рис. 4.6), либо к 5-узлу (рис. 4.7). Если вставка элемента в (2-4)-дерево осуществляется «сверху-вниз», то обычно используют разделение 4-узлов. Для вставки элемента «снизу-вверх» используется как разделение 4-узлов, так и 5-узлов. Процедура разделения 4-узла приведена ниже (процедура 4.2.13):

Процедура 4.2.13. (2-4)-Tree: Split (4-Node Split Impl)

```
01:  Split Impl(B)
02:        R = Root(B)
03:        if Is 4-Node(R) then
04:              with R do
05:                    left  = Make 2-Node(key1, child1, child2)
06:                    right = Make 2-Node(key3, child3, child4)
07:                    return Make (2-4)-Tree (
08:                          Make 2-Node (
09:                                key2,
10:                                Make (2-4)-Tree(left),
11:                                Make (2-4)-Tree(right)))
12:        return B
```

Трудоемкость приведенной процедуры есть $O(1)$, так как она выполняет фиксированное число шагов по изменению связей «предок – потомок».

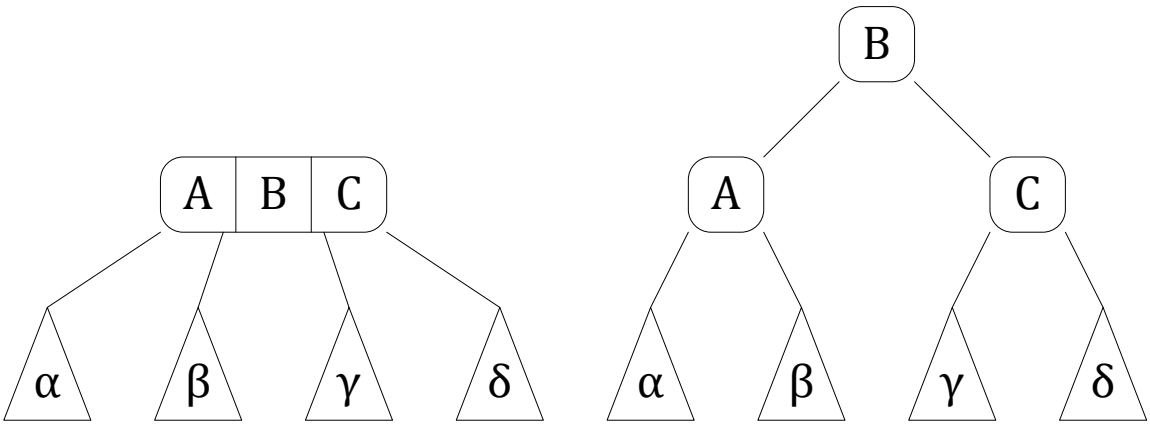

Рис. 4.6. Разделение 4-узла

Разработка процедуры разделения 5-узла не приводится по причине сходства с процедурой разделения 4-узла.

Операция *слияния*[*] применяется к двум узлам одного и того же родительского узла при условии, что один из узлов является 1-узлом (рис. 4.8). Таким образом, операция слияния может быть применена в том случае, если удаление элемента происходит из 2-листа и у удаляемого элемента брат также является

[*] В англоязычной литературе применительно к (2-4)-деревьям используется термин *fusion*.

2-листом. Следует отметить, что если родителем сливаемых узлов является 2-узел, то операцию слияния требуется *каскадировать*[*] наверх и применить к родительскому узлу (процедура 4.2.14).

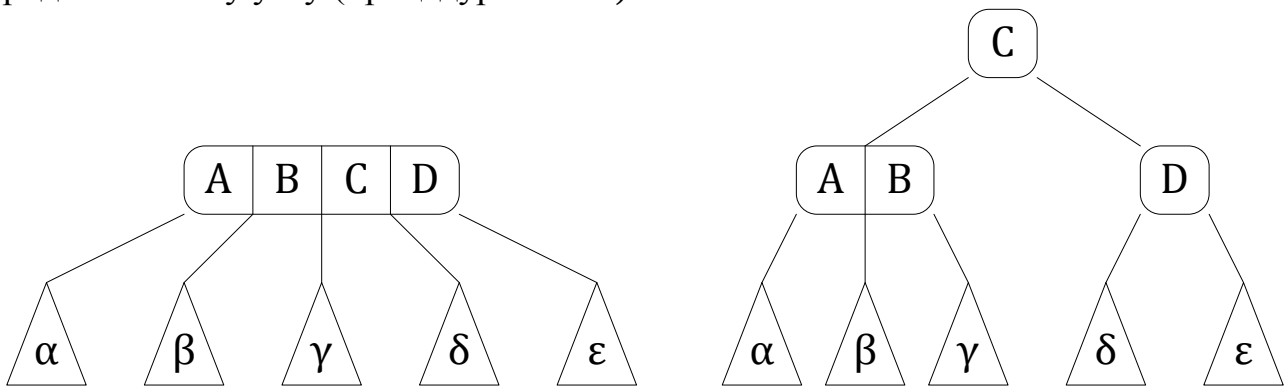

Рис. 4.7. Разделение 5-узла

Процедура 4.2.14. (2-4)-Tree: Fuse

```
01:  Fuse Impl(B)
02:        R = Root(B)
03:        if Is 1-Node(R) then
04:              if Exist Sibling(R) then
05:                    if Is 2-Node(S) then
06:                        return Complete Fuse(R, Sibling(R), Parent(R))
07:                    return Transfer(B)
08:              return Child(R)
09:        return B
10:
11:        Complete Fuse Impl(R, S, P)
12:            key1 = Get Fusing Key(S) {* Min or Max Key *}
13:            key2 = Get Fusing Key(P) {* Min or Max Key *}
14:            F = Make (2-4)-Tree(
15:                  Make 3-Node(key1, key2, Children(S) ∪ Children(R)))
16:            if Is 2-Node(P) then
17:                  return Fuse(Make (2-4)-Tree(Make 1-Node(∅, F))
18:            return Make (2-4)-Tree(
19:                       Make Node(
20:                           Keys(P) \ {key2},
21:                               Children(P) \ {T(R), T(S)} ∪ {F})
```

Трудоемкость приведенной процедуры есть $O(1)$, так как она выполняет фиксированное число шагов по изменению связей «предок – потомок».

Отметим, что в результате выполнения операции слияния высота дерева может уменьшиться на единицу.

[*] В англоязычной литературе используется термин *cascading*.

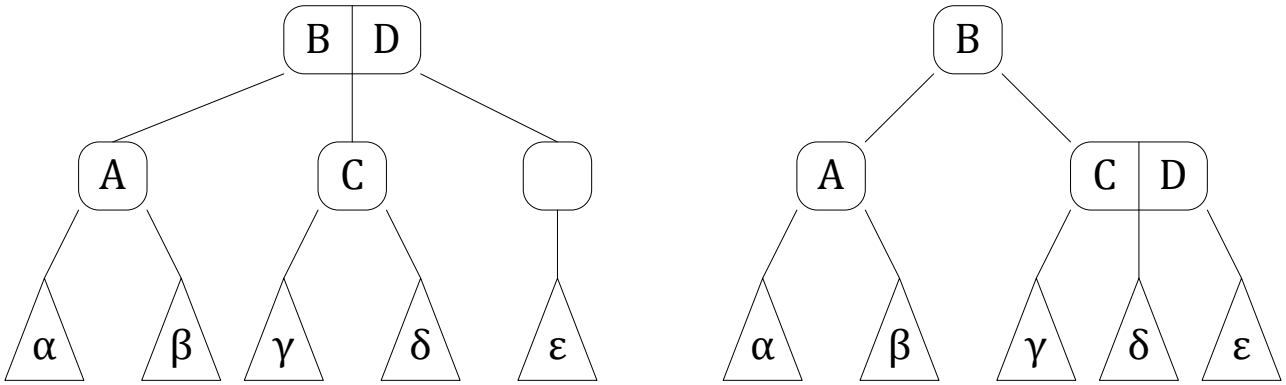

Рис. 4.8. Слияние двух узлов

Операция *перемещения*[*] (рис. 4.9), как и слияния, используется для того, чтобы удалить 1-узлы в (2-4)-дереве. Но в отличие от слияния перемещение необходимо тогда, когда братом 1-узла является 3-узел либо 4-узел. Если братом узла является 2-узел, то вызывается соответственно операция слияния узлов. Описание операции перемещения приведено в процедуре 4.2.15:

Процедура 4.2.15. (2-4)-Tree: Transfer

```
01:  Transfer Impl(B)
02:        R = Root(B)
03:        if Is 1-Node(R) then
04:              if Exist Sibling(R) then
05:                    if Is 2-Node(S) then
06:                          return Fuse(B)
07:                    return Complete Transfer(R, Sibling(R), Parent(R))
08:              return Child(R)
09:        return B
10:
11:      Complete Transfer Impl(R, S, P)
12:            key1 = Get Transferring Key(S) {* Min or Max Key *}
13:            key2 = Get Transferring Key(P) {* Min or Max Key *}
14:            FR = Make (2-4)-Tree(
15:                  Make Node(Keys(R) ∪ {key2},
16:                      Children(R) ∪ Get Appropriate Child(S, key1)))
17:            FS = Make (2-4)-Tree(
18:                  Make Node(Keys(S) \ {key1},
19:                      Children(S) \ Get Appropriate Child(S, key1)))
20:            return Make (2-4)-Tree(
21:                       Make Node(
22:                           Keys(P) \ {key2},
23:                               Children(P) \ {T(R), T(S)} ∪ {FR, FS})
```

Трудоемкость приведенной процедуры есть $O(1)$, так как она выполняет фиксированное число шагов по изменению связей «предок – потомок».

[*] В англоязычной литературе применительно к (2-4)-деревьям используется термин *transfer*.

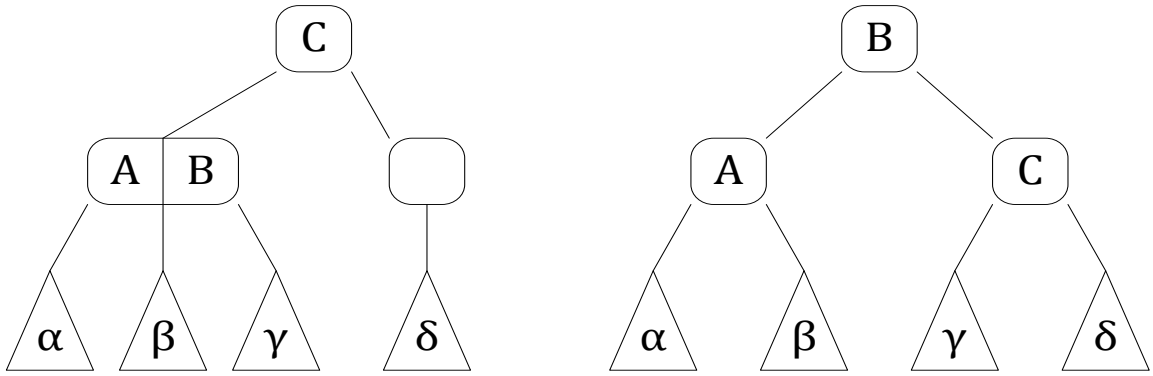

Рис. 4.9. Операция перемещения

Вставка элемента в (2-4)-дерево осуществляется только в листовой узел и может быть реализована по принципу «сверху-вниз» либо по принципу «снизу-вверх» [*]. В случае если операция вставки производится «сверху-вниз», требуется разделить каждый 4-узел, который встретится на пути от корня дерева до листа. Если операция вставки производится «снизу-вверх», необходимо осуществить «проталкивание» элементов наверх (если произошла вставка в 4-лист), причем «проталкивание» может потребоваться вплоть до корня дерева. Здесь можно воспользоваться как разделением 4-узла, так и 5-узла. Процедура вставки в (2-4)-дерево, реализованная по принципу «сверху-вниз» может выглядеть следующим образом (процедура 4.2.16):

Процедура 4.2.16. (2-4)-Tree: Insert

```
01:  Insert Impl(B, key)
02:        if not Contains(B, key) then
03:              return IRec(B)
04:        return B
05:
06:        IRec Impl(B)
07:              R = Root(B)
08:              if Is 4-Node(R) then
09:                    return IRec(Split(B))
10:              if Is Leaf(R) then
11:                    return Make (2-4)-Tree(
12:                                Make Node(Keys(R) ∪ {key}, ∅))
13:              return IRec(Get Child(R, key), B)
```

Трудоемкость приведенной процедуры есть $O(\log n)$. Реализация операции вставки «снизу-вверх» также будет иметь трудоемкость $O(\log n)$. Детали реализации оставляются читателю в качестве упражнения.

Для того чтобы удалить элемент с заданным ключом из (2-4)-дерева, требуется: найти необходимый узел, в котором храниться искомый элемент; заменить элемент его преемником; удалить преемника. Таким образом, операция удаления произвольного элемента из дерева сводится к удалению элемента, расположенного в одном из листьев того же дерева. Если удаление элемента

[*] В англоязычной литературе используются соответственно термины *top-down* и *bottom-up*.

приводит к образованию 1-узла, то требуется выполнить операцию перемещения либо слияния, чтобы сохранить инварианты, определенные на (2-4)-дереве. Детали реализации процедуры удаления приведены ниже (процедура 4.2.17):

Процедура 4.2.17. (2-4)-Tree: Delete

```
01:  Delete Impl(B, key)
02:        if not Contains(B, key) then
03:              return DRec Impl(B)
04:        return B
05:
06:        DRec Impl(B)
07:              R = Root(B)
08:              if Contains Key(R, key) then
09:                    if Is Leaf(R) then
10:                          return Delete From Leaf(B)
11:                    Replace Key(R, key, Key1(Successor(R, key)))
12:                    return DRec(Get GT Child(R, key), B)
13:              return DRec(Get Child(R, key))
14:
15:        Delete From Leaf Impl(L)
16:              R = Root(L)
17:              if Contains Key(R, key) then
18:                    D = Remove Key(R, Key)
19:                    if Is 1-Node(D) then
20:                          return Fuse(T(D))
21:                    return D
22:              return L
```

Трудоемкость приведенной процедуры есть $O(\log n)$. Так как процедура удаления может привести к образованию 1-узла, то после удаления искомого элемента высота дерева может уменьшиться на единицу.

В заключение отметим, что существуют так называемые (2-3)-деревья [4] и d-деревья, у которых максимум может быть d детей. Особенностью (2-3)-деревьев, как и (2-4)-деревьев, является гарантированная высота дерева $O(\log n)$, и, следовательно, гарантированная трудоемкость операций вставки и удаления $O(\log n)$. Рассмотрим полное d-дерево D, у каждого узла которого ровно d детей. Тогда высота дерева D есть $O(\frac{\log n}{\log d})$. Операция поиска элемента с заданным ключом в d-дереве в худшем случае будет иметь трудоемкость $O(\log n)$, так как помимо перемещения с более высоких уровней дерева на более низкие требуется осуществлять проверку наличия элемента внутри узла с использованием двоичного поиска, трудоемкость которого составит для d-узла $O(\log d)$. Таким образом, использование d-деревьев не позволяет улучшить асимптотику выполнения операции поиска в сравнение с (2-4)-деревьями. В то же самое время, реализация (2-4)- и (2-3)-деревьев представляется значительно проще реализации сбалансированных d-деревьев.

Также следует отметить, что, несмотря на асимптотику выполнения $O(\log n)$ всех операций, определенных на (2-4)-дереве, их использование на

практике вызвано рядом трудностей. К основным из них относится поддержка различных типов узлов, а также анализ ряда частных случаев с целью поддержания дерева в сбалансированном состоянии. По этой причине (2-4)-деревья обычно заменяют их бинарными эквивалентами (например, красно-черным деревом), трудоемкость выполнения операций на которых также есть $O(\log n)$.

4.2.3. Красно-черное дерево

Красно-черное дерево – сбалансированное бинарное дерево поиска, на котором определен следующий набор инвариантов:

1) каждый узел дерева раскрашен в красный либо в черный цвет;

2) корень и листья дерева окрашены в черный цвет;

3) у красного узла оба дочерних узла окрашены в черный цвет;

4) все пути от узла до листьев, родителем которых является этот узел, содержат одинаковое число черных узлов.

Красно-черное дерево было предложено Р. Байером в 1972 г. Название же «красно-черное» дерево было введено Л. Гибасом и Р. Седжвиком в 1978 г.

Между красно-черными деревьями и (2-4)-деревьями существует тесная связь, которая заключается в том, что одно дерево может быть получено из другого путем применения ряда несложных операций. Для того чтобы преобразовать красно-черное дерево в эквивалентное ему (2-4)-дерево, достаточно поднять красные узлы на уровень их предков – черных узлов (рис. 4.10). Для преобразования (2-4)-дерева в красно-черное дерево можно воспользоваться набором правил, приведенным на рис. 4.11. Видим, что одному и тому же (2-4)-дереву может соответствовать несколько красно-черных деревьев, поскольку 3-узел можно представить двумя способами в красно-черном дереве.

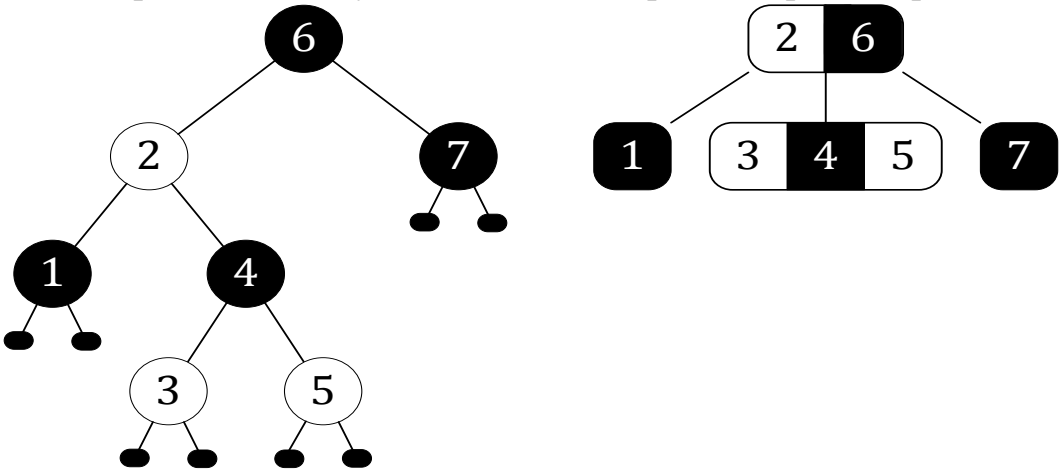

Рис. 4.10. Красно-черное дерево и эквивалентное ему (2-4)-дерево

Для того чтобы показать, что красно-черное дерево есть бинарное дерево поиска высоты $O(\log n)$, введем понятие *черной высоты* дерева. Под *черной высотой* будем понимать количество черных узлов на пути от корня дерева до любого из его листьев. Тогда число узлов, которое может содержать красно-черное дерево с черной высотой b, должно быть не менее $2^b - 1$. Так как высота дерева h не может превышать величины $2b$, то

$$h \le 2b \le 2\log_2(n+1) \Rightarrow h = O(\log n). \qquad (4.11)$$

Из соотношения (4.11) следует, что красно-черное дерево есть сбалансированное бинарное дерево поиска.

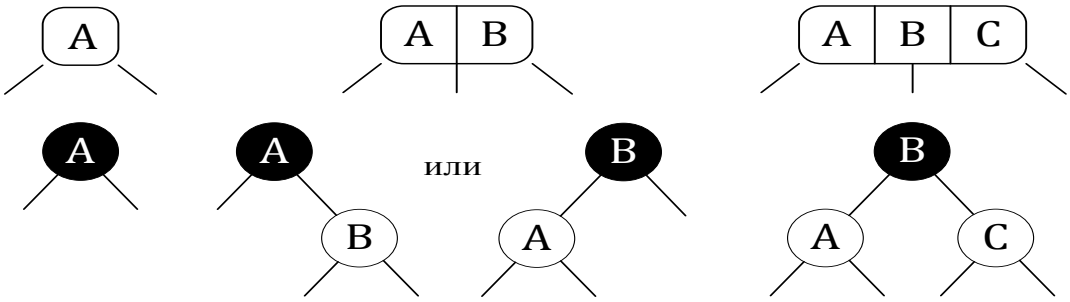

Рис. 4.11. Преобразование узлов (2-4)-дерева в узлы красно-черного дерева

Операции вставки и удаления в красно-черное дерево осуществляются, как и в обычное бинарное дерево поиска с тем лишь отличием, что после добавления либо удаления узла требуется совершить операцию балансировки. Особенностью красно-черных деревьев является то, что операция балансировки для операций добавления и удаления будет различной. За подробностями операции балансировки отправляем читателя к таким источникам как [7, 11]. Отметим лишь, что балансировка красно-черных деревьев, как и AVL-деревьев, базируется на операциях поворота.

В заключение следует отметить, что красно-черные деревья – наиболее распространенный тип BST, который используется для реализации словарных структур данных. Несмотря на то, что высота красно-черного дерева может быть выше высоты AVL-дерева, построенного для тех же элементов, использование красно-черного дерева является более предпочтительным, поскольку число поворотов, которое требуется сделать после вставки либо удаления элемента, есть $O(1)$, в то время как для AVL-дерева число поворотов может достигать величины $O(\log n)$.*

4.2.4. B-деревья

B-дерево порядка t – сбалансированное дерево поиска, на котором определен следующий набор инвариантов [13]:

1) каждый узел, кроме корня, содержит не менее $t-1$ ключей, и каждый внутренний узел имеет, по меньшей мере, t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ;

2) каждый узел, кроме корня, содержит не более $2t-1$ ключей;

3) корень содержит от 1 до $2t-1$ ключей, если дерево не пусто;

4) каждый из узлов дерева хранит ключи в отсортированном порядке;

5) все поддеревья одного и того же уровня имеют одинаковую высоту.

* Можно показать, что высота красно-черного дерева будет превышать высоту AVL-дерева не более чем на 39 %.

Из определения B-дерева следует, что рассмотренное выше (2-4)-дерево есть B-дерево с показателем 2. Несложно показать, что высота B-дерева с показателем большим единицы будет удовлетворять следующей оценке:

$$\log_{2t}(n+1) - 1 \le h \le \log_t \frac{(n+1)}{2} \Rightarrow h = O(\log n). \qquad (4.12)$$

При $t = 2$ соотношение (4.12) принимает вид (4.10).

Как и для (2-4)-деревьев, для B-деревьев определены операции поиска, вставки и удаления элемента, причем операции вставки и удаления базируются на операциях разделения, слияния и перемещения. Так, если требуется разделить полный узел B-дерева, то достаточно взять его средний элемент и переместить наверх. В случае операций слияния и перемещения требуется проверить, является ли братом узел, который содержит $t - 1$ ключей (или более), и в зависимости от результата проверки осуществить слияние узлов либо перемещение. Как и для (2-4)-деревьев, операция слияния узлов B-дерева может быть каскадирована наверх, если родитель сливаемых узлов «потерял» ключ.

Трудоемкость операции поиска в B-дереве в худшем случае есть $O(\log n)$ и она сопоставима со временем поиска в (2-4)-дереве либо в красно-черном дереве, поскольку проверка принадлежности искомого элемента заданному узлу в худшем случае имеет трудоемкость $O(\log_2 t)$. Несмотря на то, что использование B-деревьев не позволяет улучшить время поиска элемента с заданным ключом, они получили широкое распространение на практике для хранения больших объемов данных. Здесь следует отметить, что под большим объемом данных подразумевается такой, который не может быть размещен в оперативной памяти ЭВМ.

Наиболее широкое распространение B-деревья получили при проектировании систем управления базами данных (СУБД) и файловых систем. Так, в контексте СУБД B-деревья и его модификации используются для хранения индексов, позволяющих получить быстрый доступ к элементам с искомым ключом. Основной же целью использования B-деревьев при работе с внешней памятью является минимизация числа операций чтения / записи на диск. Таким образом, высота B-дерева как раз и определяет, сколько обращений придется сделать к диску в худшем случае для той, либо иной операции.

Следует отметить, что у B-дерева существуют модификации, которые получили названия *B*-дерева* и *B+-дерева*. В B-дереве вместе с ключом элемента может храниться только указатель на некий блок памяти, содержащий сопутствующую информацию для данного ключа. В *B+-дереве* вся информация хранится в листьях, а во внутренних узлах хранятся только ключи и указатели на дочерние узлы. Таким образом можно добиться максимально возможной степени ветвления во внутренних узлах. *B*-дерево* – модификация B-дерева, в которой каждый внутренний узел должен быть заполнен как минимум на две трети, а не наполовину, как в стандартном B-дереве. В отличие от B+-деревьев, узел в B*-дереве не разбивается на два узла, если полностью заполнен. Вместо этого

необходимо найти место в уже существующем соседнем узле, и только после заполнения обоих узлов они разделяются на три узла.

4.3. Задачи для самостоятельного решения

1. На плоскости находится выпуклый многоугольник P и набор точек S. Разработайте алгоритм, который для каждой точки p из S определит, принадлежит точка p многоугольнику P или нет. Точка p принадлежит многоугольнику P, если она лежит либо на границе, либо внутри многоугольника P.

2. Рассмотрим множество точек S на плоскости. Будем говорить, что они образуют шеренгу, если расположены на горизонтальной либо вертикальной прямой и расстояние между соседними точками равняется 1. Требуется заданное множество точек преобразовать в шеренгу, используя минимальное количество шагов. За один шаг разрешается передвинуть одну из точек на единицу по вертикали либо по горизонтали, но не в позиции, которые уже заняты.

3. Рассмотрим m отрезков. Длина отрезка i есть L_i. Требуется из заданных m отрезков сформировать n отрезков одинаковой длины, причем длина полученных отрезков должна быть максимально возможной. Отрезки допускается разрезать. Операция соединения отрезков запрещена.

4. Рассмотрим n объектов, занумерованных натуральными числами от 1 до n. Известно, что масса первого объекта есть m_1, масса n-го объекта – m_n, а масса i-го объекта $(1 < i < n) - m_i = d + (m_{i-1} + m_{i+1})/2$, где $d -$ некоторое заданное число. Разработайте алгоритм, который позволит эффективно определять массу j-го объекта для заданного j.

5. При наборе текста довольно часто возникают опечатки из-за неправильного нажатия на клавиши, например, замена букв, лишние буквы в словах либо их нехватка. В большинстве случаев эти опечатки можно исправить автоматически. В частности, существует метод проверки орфографии, основанный на поиске в словаре слов, *похожих* на проверяемые. Два слова называются *похожими*, если можно удалить из каждого слова не более одной буквы так, чтобы они стали одинаковыми, возможно пустыми. Требуется разработать алгоритм, который для каждого слова проверяемого текста определит количество похожих на него слов из заданного словаря.

6. Рассмотрим внешней файл записей, каждая из которых представляет ребро некоторого графа G и стоимость этого ребра. Разработайте алгоритм, который построит минимальное остовное дерево графа G, полагая, что объем основной памяти достаточен для хранения всех вершин графа, но не достаточен для хранения всех его ребер.

7. Хранилище данных на некотором сервере организовано как таблица, у которой две строки и n столбцов. В каждой клетке таблицы хранится информация о некотором объекте. Каждый объект, хранимый в таблице, характеризуется своим уникальным идентификатором, который является целым числом. Пронумеруем верхнюю строку таблицы числом 0, а нижнюю – числом 1. Также пронумеруем столбцы таблицы слева направо от 0 до $n - 1$ и обозначим через

$id(i,j)$ идентификатор объекта, хранимого в клетке на пересечении строки i столбца j. Известно, что данные хранятся в таблице в определенном порядке, для которого выполнены следующие условия:

1) $id(0,j) < id(1,j)$ для всех j от 0 до $n-1$ включительно;

2) $id(0,j) < id(0,j+1)$ для всех j от 0 до $n-2$ включительно;

3) $id(1,j) < id(1,j+1)$ для всех j от 0 до $n-2$ включительно.

Для того чтобы найти некоторый объект в хранилище, требуется отправить на сервер запрос, содержащий числа i,j и x. В ответ сервер посылает число -1, если $id(i,j) < x$; число 0, если $id(i,j) = x$; число 1, если $id(i,j) > x$.

Требуется разработать алгоритм, который по числу n определит количество запросов, затрачиваемых в худшем случае на поиск одного объекта при оптимальной организации процесса поиска.

8. Рассмотрим забор, которым огорожено некоторое квадратное поле размером $n \times n$. Один угол забора расположен в точке $(0,0)$, противоположный ему угол находится в точке (n,n). Стороны забора параллельны осям X и Y. Столбы, к которым крепится забор, стоят не только по углам, но и через метр вдоль каждой стороны поля, всего в заборе $4 \times n$ столбов. Столбы стоят вертикально и не имеют толщины (радиус равен нулю). Задача состоит в определении количества столбов, которые можно увидеть, находясь в определенной точке поля. Проблема заключается в том, что на поле расположено R огромных камней, и из-за них не видны некоторые из столбов. Основание каждого камня представляет собой выпуклый многоугольник ненулевой площади, вершины которого имеют целочисленные координаты. Камни стоят на поле вертикально и у них нет общих точек между собой, а также с забором. Точка, где стоит наблюдатель, лежит внутри, но не на границе поля, а также не на границе и не внутри камней.

По размеру поля, положению и форме камней на нем и месту, где стоит наблюдатель, требуется вычислить количество столбов, которые может видеть наблюдатель. Если вершина основания камня находится на одной линии с местом расположения наблюдателя и некоторым столбом, то наблюдатель не видит этот столб.

9. В городе H планируется создать сеть автобусных маршрутов с n автобусными остановками. Каждая остановка находится на некотором перекрестке. Поскольку H – современный город, на карте он представлен улицами с квадратными кварталами одинакового размера. Две из n остановок выбираются пересадочными станциями h_1 и h_2, которые будут соединены друг с другом прямым автобусным маршрутом, а каждая из оставшихся $n-2$ остановок будет непосредственно соединена маршрутом с одной из пересадочных станций h_1 или h_2 (но не с обеими), и не соединена ни с одной из оставшихся остановок.

Расстояние между любыми двумя остановками определяется как длина кратчайшего пути по улицам города. Это означает, что если остановка представлена парой координат (x,y), то расстояние между двумя остановками $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ будет равно $d(p_1,p_2) = |x_1 - x_2| + |y_1 - y_2|$. Если

остановки A и B соединены с одной и той же пересадочной станцией, например h_1, то длина пути из A в B есть $d(A,h_1)+d(B,h_1)$. Если они соединены с разными пересадочными станциями, например A с h_1 и B с h_2, то длина пути из A в B есть $d(A,h_1)+d(h_1,h_2)+d(B,h_2)$.

Проектировщики города H хотят быть уверены, что любой его житель сможет добраться до какой-либо точки города достаточно быстро. Поэтому проектировщики хотят сделать пересадочными станциями такие две остановки, чтобы в полученной сети автобусных маршрутов максимальная длина пути между любыми двумя остановками была минимальной.

Вариант P выбора пересадочных станций и соединения остановок с ними будет лучше варианта Q, если максимальная длина пути между любыми двумя остановками в варианте P будет меньше, чем в варианте Q. Требуется разработать алгоритм вычисления максимальной длины пути между любыми двумя остановками для наилучшего варианта P выбора пересадочных станций и соединения остановок с ними.

10. Рассмотрим битовую последовательность S достаточно большой длины, записанную в файл. Требуется найти битовые подпоследовательности последовательности S длиной от A до B включительно $(1 \le A, B \le 15)$, которые встречаются в S наиболее часто. Подпоследовательности могут перекрываться между собой. Предполагается, что последовательность S настолько длинна, что она не может поместиться полностью в оперативной памяти ЭВМ.

11. Рассмотрим прямоугольную металлическую платформу P размерами $n \times m$. Платформа P разделена на k прямоугольных секций. Стороны каждой из секций параллельны сторонам платформы P. Разработайте алгоритм, который позволит найти наибольший прямоугольник, полученный путем объединений не более чем s соседних секций платформы P $(1 \le s \le k)$.

12. Рассмотрим турнир по троеборью, в котором участвует n спортсменов. Сила каждого из спортсменов характеризуется тройкой чисел (a_i, b_i, c_i), где a_i – уровень силы спортсмена i в спорте a, b_i – в спорте b, c_i – в спорте c. Известно, что уровни силы всех спортсменов в рамках одного и того же спорта различны. Будем говорить, что спортсмен i *прямо побеждает* спортсмена j, если у спортсмена i по крайней мере два уровня силы больше, чем у спортсмена j. Разработайте алгоритм, который для каждого из n спортсменов определит, сможет он выиграть турнир или нет.

13. На плоскости построен план слаломной трассы так, что точка старта имеет координаты (0,0), точка финиша – координаты $(x, y), y > 0$. На трассе находятся n ворот, через которые обязательно должен проехать слаломист (пусть даже проезжая по одной из их границ). Порядок прохождения ворот соответствует порядку их описания.

Каждые ворота расположены на плане параллельно оси абсцисс и описываются точками (a_i, y_i) и (b_i, y_i), где $0 < y_1 < \cdots < y_n < y$, $a_i < b_i$. Требуется определить, какое минимальное расстояние должен преодолеть слаломист от старта до финиша, чтобы не нарушить правила прохождения ворот.

Литература

1. Okasaki, C. Purely Functional Data Structures / C. Okasaki – Cambridge University Press, New York, NY, USA, 1999. – 230 p.
2. Kaplan, H. Purely Functional, Real–Time Deques with Catenation / H. Kaplan, R. E. Tarjan – JACM, Vol. 46, Issue 5, Sept. 1999. – P. 577 – 603.
3. Driscoll, J. R. Making Data Structures Persistent / J. R. Driscoll [and others] – STOC '86, ACM New York, NY, USA, 1986. – P. 109 – 121.
4. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Дж. Ульман, Дж. Хопкрофт; пер. с англ. – М. : Вильямс, 2003. – 384 с.
5. Bender, M. A. The LCA Problem Revisited / M. A. Bender, M. Farach-Colton – LATIN'00, Springer-Verlag London, UK, 2000. – P. 88 – 94.
6. Уоррен, Г. С., мл. Алгоритмические трюки для программистов / Г. С. Уоррен, мл.; пер. с англ. – М. : Вильямс, 2007. – 288 с.
7. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен [и др.]; пер. с англ. – М. : Вильямс, 2007. – 1296 с.
8. Tarjan, R. E. Worst-case Analysis of Set Union Algorithms / R. E. Tarjan, J. van Leeuwen – JACM, Vol. 31, Issue 2, April. 1984. – P. 245 – 281.
9. Кнут, Д. Э. Искусство программирования. Том 3. Сортировка и поиск / Д. Э. Кнут; пер. с англ. – М. : Вильямс, 2011. – 824 с.
10. Shell, D. L. A High-Speed Sorting Procedure / D. L. Shell – CACM, Vol. 2, Issue 7, July 1959. – P. 30 – 32.
11. Седжвик, Р. Алгоритмы на C++ / Р. Седжвик; пер. с англ. – М. : Вильямс, 2011. – 1056 с.
12. Blum, M. Time bounds for selection / M. Blum [and others]. – JCSS Vol. 7, 1973. – P. 448 – 461.
13. Bayer, R. Organization and Maintenance of Large Ordered Indexes / R. Bayer, E. McCreight – Acta Informatica, Vol. 1, Fasc. 3, 1972. – P. 173 – 189.