

А. В. МАЕР, О. С. ЧЕРЕПАНОВ

**ВВЕДЕНИЕ В СТРУКТУРЫ
И АЛГОРИТМЫ ОБРАБОТКИ
ДАНЫХ**

УЧЕБНОЕ ПОСОБИЕ



Курганский
государственный
университет



Библиотечно-издательский
центр

65-48-12

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное
учреждение высшего образования
«Курганский государственный университет»

А. В. Маер, О. С. Черепанов

**ВВЕДЕНИЕ В СТРУКТУРЫ И АЛГОРИТМЫ
ОБРАБОТКИ ДАННЫХ**

Учебное пособие

Курган 2021

УДК 519.688 (075.8)
ББК 32.973я73
М 13

Рецензенты:

кандидат технических наук, доцент кафедры «Автоматизация и робототехника» ФГБОУ ВО «Омский государственный технический университет» М. С. Пешко;

кандидат физико-математических наук, доцент кафедры «Опτικο-электронных систем и дистанционного зондирования» Национальный исследовательский «Томский государственный университет» Л. Г. Шаманаева.

Печатается по решению методического совета Курганского государственного университета.

Маер А. В., Черепанов О. С.

Введение в структуры и алгоритмы обработки данных : учебное пособие / А. В. Маер, О. С. Черепанов. – Курган : Изд-во Курганского гос. ун-та, 2021. – 107 с.

Учебное пособие состоит из пяти глав. Целью учебного пособия является формирование начального представления о структурах данных и алгоритмах. В первой главе дается обзор математического аппарата, используемого при анализе алгоритмов. Во второй главе вводится понятие псевдокода. Третья глава посвящена основным структурам данных. В четвертой главе проводится анализ алгоритмов и выполняется их построение с помощью псевдокода. Пятая глава содержит набор заданий для выполнения учебных проектов.

Учебное пособие предназначено для студентов направлений 09.03.03, 09.03.04, 10.03.01, 10.05.03.

ISBN 978-5-4217-0576-5

© Курганский государственный
университет, 2021

© Маер А. В., Черепанов О. С., 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 НЕКОТОРЫЕ МАТЕМАТИЧЕСКИЕ ОБОЗНАЧЕНИЯ	5
1.1 Асимптотические обозначения	5
1.2 Стандартные функции и обозначения	9
2 ПСЕВДОКОД	13
3 СТРУКТУРЫ ДАННЫХ	16
3.1 Кучи	18
3.1.1 Сохранение основного свойства кучи.....	20
3.1.2 Построение кучи	21
3.2 Стеки и очереди	23
3.2.1 Стеки.....	24
3.2.2 Очереди.....	25
3.3 Связанные списки.....	27
3.4 Корневые деревья	29
4 АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ	32
4.1 Сортировки во внутренней памяти.....	33
4.1.1 Сортировка слиянием	34
4.1.2 Сортировка с помощью кучи	36
4.1.3 Быстрая сортировка	37
4.1.4 Сортировка подсчетом.....	40
4.2 Сортировки во внешней памяти.....	42
4.2.1 Прямое слияние.....	43
4.2.2 Естественное слияние	44
4.2.3 Многопутевое слияние	46
4.3 Поиск подстрок.....	47
4.3.1 Простейший алгоритм.....	49
4.3.2 Алгоритм Рабина-Карна	50
4.3.3 Алгоритм Кнута-Морриса-Пратта	53
4.4 Базовые алгоритмы на графах	56
4.4.1 Поиск в ширину.....	56
4.4.2 Поиск в глубину.....	59
4.5 Двоичные деревья поиска.....	62
5 ЛАБОРАТОРНЫЙ ПРАКТИКУМ	70
5.1 Лабораторная работа № 1	70
5.2 Лабораторная работа № 2	73
5.3 Лабораторная работа № 3	80
5.4 Лабораторная работа № 4	86
5.5 Лабораторная работа № 5	90
5.6 Лабораторная работа № 6	95
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	106

ВВЕДЕНИЕ

Учебное пособие подходит в качестве первого знакомства со структурами данных и алгоритмами. В нем проводится анализ современных и наиболее популярных алгоритмов. При построении алгоритмов использован наиболее эффективный (по мнению авторов) инструмент – псевдокод. Описание алгоритмов выполнено максимально просто, чтобы читатель с минимальной подготовкой мог воспользоваться данным учебным пособием.

Последний раздел посвящен практическим заданиям, которые можно выполнить после знакомства с теоретической частью пособия.

Материал изложен таким образом, что он будет полезен студентам-бакалаврам технических специальностей.

1 НЕКОТОРЫЕ МАТЕМАТИЧЕСКИЕ ОБОЗНАЧЕНИЯ

Анализируя алгоритм, можно стараться найти точное число выполняемых им действий. Но в большинстве случаев игра не стоит свеч, и достаточно оценить асимптотику роста времени работы алгоритма при стремлении размера входа к бесконечности. Если у одного алгоритма скорость роста меньше, чем у другого, то в большинстве случаев он будет эффективнее для всех входов, кроме совсем коротких (хотя бывают и исключения).

1.1 Асимптотические обозначения

Хотя во многих случаях эти обозначения используются неформально, полезно начать с точных определений.

Θ -обозначение. Например, время $T(n)$ работы алгоритма на входах длины n есть $\Theta(n^2)$. Точный смысл этого утверждения такой: найдутся такие константы $c_1, c_2 > 0$ и такое число n_0 , что $c_1 n^2 \leq T(n) \leq c_2 n^2$ при всех $n \geq n_0$. Вообще, если $g(n)$ – некоторая функция, то запись $f(n) = \Theta(g(n))$ означает, что найдутся такие $c_1, c_2 > 0$ и такое n_0 , что $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ для всех $n \geq n_0$ (рисунок 1.1). (Запись $f(n) = \Theta(g(n))$ читается так: «Эф от эн есть тэта от же от эн»).

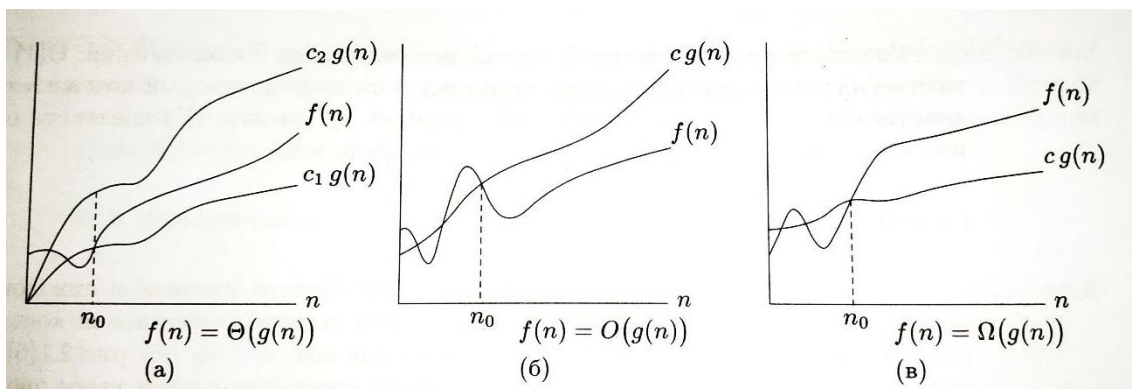


Рисунок 1.1 – Иллюстрация к определениям $f(n) = \Theta(g(n))$,
 $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$

Разумеется, это обозначение следует употреблять с осторожностью: установив, что $f_1(n) = \Theta(g(n))$ и $f_2(n) = \Theta(g(n))$, не следует заключать, что $f_1(n) = f_2(n)$!

Определение $\Theta(g(n))$ предполагает, что функции $f(n)$ и $g(n)$ асимптотически неотрицательны, т. е. неотрицательны для достаточно больших значений n . Заметим, что если функции f и g строго положительны, то можно исключить n_0 из определения (изменив константы c_1 и c_2 так, чтобы для малых n неравенство также выполнялось).

Если $f(n) = \Theta(g(n))$, то говорят, что $g(n)$ является асимптотически точной оценкой для $f(n)$. На самом деле это отношение симметрично: если $f(n) = \Theta(g(n))$, то $g(n) = \Theta(f(n))$.

Например, проверим, что $\left(\frac{1}{2}\right)n^2 - 3n = \Theta(n^2)$. Согласно определению, надо указать положительные константы c_1, c_2 и число n_0 так, чтобы неравенство $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ выполнялось для всех $n \geq n_0$. Разделим на n^2 : $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$. Видно, что для выполнения второго неравенства достаточно положить $c_2 = 1/2$. Первое будет выполнено, если (например) $n_0 = 7$ и $c_1 = 1/14$.

Другой пример использования формального определения: покажем, что $6n^3 \neq \Theta(n^2)$. В самом деле, пусть найдутся такие c_2 и n_0 , что $6n^3 \leq c_2 n^2$ для всех $n \geq n_0$. Но тогда $n \leq c_2/6$ для всех $n \geq n_0$, что явно не так.

Отыскивая асимптотически точную оценку для суммы, мы можем отбрасывать члены меньшего порядка, которые при больших n становятся малыми по сравнению с основным слагаемым. Заметим также, что коэффициент при старшем члене роли не играет (он может повлиять только на выбор констант c_1 и c_2). Например, рассмотрим квадратичную функцию $f(n) = an^2 + bn + c$, где a, b, c – некоторые константы и $a > 0$. Отбрасывая члены младших порядков и коэффициент при старшем члене, находим, что $f(n) = \Theta(n^2)$. Чтобы убедиться в этом формально, можно положить $c_1 = a/4$, $c_2 = 7a/4$ и $n_0 = 2 * \max(|b|/a, \sqrt{|c|/a})$ (проверьте, что требования действительно выполнены). Вообще, для любого полинома $p(n)$ степени d с положительным старшим коэффициентом имеем $p(n) = \Theta(n^d)$.

Упомянем важный частный случай использования Θ -обозначений: $\Theta(1)$ обозначает ограниченную функцию, отделенную от нуля

некоторой положительной константой при достаточно больших значениях аргумента (из контекста обычно ясно, что именно считается аргументом функции).

Θ - и Ω -обозначения. Запись $f(n) = \Theta(g(n))$ включает в себя две оценки: верхнюю и нижнюю. Их можно разделить. Говорят, что $f(n) = O(g(n))$, если найдется такая константа $c > 0$ и такое число n_0 , что $0 \leq f(n) \leq cg(n)$ для всех $n \geq n_0$. Говорят, что $f(n) = \Omega(g(n))$, если найдется такая константа $c > 0$ и такое число n_0 , что $0 \leq cg(n) \leq f(n)$ для всех $n \geq n_0$. Эти записи читаются так: «Эф от эн есть о большое от же от эн», «Эф от эн есть омега большая от же от эн».

По-прежнему мы предполагаем, что функции f и g неотрицательны для достаточно больших значений аргумента. Для этих функций выполняются следующие свойства.

Для любых двух функций $f(n)$ и $g(n)$ свойство $f(n) = \Theta(g(n))$ выполнено тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Для любых двух функций свойства $f(n) = O(g(n))$ и $g(n) = \Omega(f(n))$ равносильны.

Как мы видели, $an^2 + bn + c = \Theta(n^2)$ (при положительных a). Поэтому $an^2 + bn + c = O(n^2)$. Другой пример: при $a > 0$ можно написать $an + b = O(n^2)$ (положим $c = a + |b|$ и $n_0 = 1$). Заметим, что в этом случае $an + b \neq \Omega(n^2)$ и $an + b \neq \Theta(n^2)$.

Асимптотические обозначения (Θ , O и Ω) часто употребляются внутри формул. Например, рекуррентное соотношение $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ для времени работы сортировки слиянием. Здесь $\Theta(n)$ обозначает некоторую функцию, про которую нам важно знать лишь, что она не меньше c_1n и не больше c_2n для некоторых положительных c_1 и c_2 и для всех достаточно больших n .

Часто асимптотические обозначения употребляются не вполне формально, хотя их подразумеваемый смысл обычно ясен из контекста. Например, мы можем написать выражение $\sum_{i=1}^n O(i)$, имея в виду сумму $h(1) + h(2) + \dots + h(n)$, где $h(i)$ – некоторая функция, для которой $h(i) = O(i)$. Легко видеть, что сама эта сумма как функция от n есть $O(n^2)$.

Типичный пример использования асимптотических обозначений – цепочка равенств наподобие $2n^2 + 3n + 1 = 2n^2 + \theta(n) = \theta(n^2)$. Второе из этих равенств ($2n^2 + \theta(n) = \theta(n^2)$) понимается при этом так: какова бы ни была функция $h(n) = \theta(n)$ в левой части, сумма $2n^2 + h(n)$ есть $\theta(n^2)$.

o- и ω-обозначения. Запись $f(n) = O(g(n))$ означает, что с ростом n отношение $f(n)/g(n)$ остается ограниченным. Если к тому же $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, то мы пишем $f(n) = o(g(n))$, если для всякого положительного $\varepsilon > 0$ найдется такое n_0 , что $0 \leq f(n) \leq \varepsilon g(n)$ при всех $n \geq n_0$ (тем самым запись $f(n) = o(g(n))$ предполагает, что $f(n)$ и $g(n)$ неотрицательны для достаточно больших n). Например, $2n = o(n^2)$, но $2n^2 \neq o(n^2)$.

Аналогичным образом вводится ω -обозначение: говорят, что $f(n)$ есть $\omega(g(n))$ («Эф от эн есть омега малая от же от эн»), если для любого положительного c существует такое n_0 , что $0 \leq cg(n) \leq f(n)$ при всех $n \geq n_0$. Другими словами, $f(n) = \omega(g(n))$ означает, что $g(n) = o(f(n))$. Например, $\frac{n^2}{2} = \omega(n)$, но $\frac{n^2}{2} \neq \omega(n^2)$.

Сравнение функций. Введенные нами определения обладают некоторыми свойствами транзитивности, рефлексивности и симметричности.

Транзитивность:

$f(n) = \theta(g(n))$ и $g(n) = \theta(h(n))$ влечет $f(n) = \theta(h(n))$,

$f(n) = O(g(n))$ и $g(n) = O(h(n))$ влечет $f(n) = O(h(n))$,

$f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$ влечет $f(n) = \Omega(h(n))$,

$f(n) = o(g(n))$ и $g(n) = o(h(n))$ влечет $f(n) = o(h(n))$,

$f(n) = \omega(g(n))$ и $g(n) = \omega(h(n))$ влечет $f(n) = \omega(h(n))$.

Рефлексивность:

$f(n) = \theta(f(n))$, $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$.

Симметричность:

$f(n) = \theta(g(n))$ если и только если $g(n) = \theta(f(n))$.

Обращение:

$f(n) = O(g(n))$ если и только если $g(n) = \Omega(f(n))$.

$f(n) = o(g(n))$ если и только если $g(n) = \omega(f(n))$.

Можно провести такую параллель: отношения между функциями f и g подобны отношениям между числами a и b :

$$\begin{aligned}
f(n) &= O(g(n)) \approx a \leq b, \\
f(n) &= \Omega(g(n)) \approx a \geq b, \\
f(n) &= \Theta(g(n)) \approx a = b, \\
f(n) &= o(g(n)) \approx a < b, \\
f(n) &= \omega(g(n)) \approx a > b.
\end{aligned}$$

Параллель эта, впрочем, весьма условна: свойства числовых неравенств не переносятся на функции. Например, для любых двух чисел a и b всегда или $a \leq b$, или $a \geq b$, однако нельзя утверждать, что для любых двух (положительных) функций $f(n)$ и $g(n)$ или $f(n) = O(g(n))$, или $f(n) = \Omega(g(n))$. В самом деле, можно проверить, что ни одно из этих двух соотношений не выполнено для $f(n) = n$ и $g(n) = n^{1+\sin n}$ (показатель степени в выражении для $g(n)$ меняется в интервале от 0 до 2). Заметим еще, что для чисел $a \leq b$ влечет $a < b$ или $a = b$, в то время как для функций $f(n) = O(g(n))$ не влечет $f(n) = o(g(n))$ или $f(n) = \Theta(g(n))$.

1.2 Стандартные функции и обозначения

Монотонность. Говорят, что функция $f(n)$ монотонно возрастает, если $f(m) \leq f(n)$ при $m \leq n$. Говорят, что функция $f(n)$ монотонно убывает, если $f(m) \geq f(n)$ при $m \leq n$. Говорят, что функция $f(n)$ строго возрастает, если $f(m) < f(n)$ при $m < n$. Говорят, что функция $f(n)$ строго убывает, если $f(m) > f(n)$ при $m < n$.

Целые приближения снизу и сверху. Для любого вещественного числа x через $[x]$ мы обозначаем его целую часть, т. е. наибольшее целое число, не превосходящее x . Симметричным образом $\lceil x \rceil$ обозначает наименьшее целое число, не меньшее x . Очевидно, $x - 1 < [x] \leq x \leq \lceil x \rceil < x + 1$ для любого x . Кроме того, $[n/2] + \lceil n/2 \rceil = n$ для любого целого n . Наконец, для любого x и для любых целых положительных a и b имеем $\lceil [x/a]/b \rceil = \lceil x/ab \rceil$ и $\lfloor \lceil x/a \rceil / b \rfloor = \lfloor x/ab \rfloor$ (чтобы убедиться в этом, полезно заметить, что для любого z и для целого n свойства $n \leq z$ и $n \leq [z]$ равносильны). Функции $x \mapsto [x]$ и $x \mapsto \lceil x \rceil$ монотонно возрастают.

Многочлены. Многочленом (полиномом) степени d от переменной n называют функцию $p(n) = \sum_{i=0}^d a_i n^i$ (d – неотрицательное целое число). Числа a_1, a_2, \dots, a_d называют коэффициентами многочлена. Мы считаем, что старший коэффициент a_d не равен нулю (если это не так, уменьшим d – это можно сделать, если только многочлен не равен нулю тождественно).

Для больших значений n знак многочлена $p(n)$ определяется старшим коэффициентом (остальные члены малы по сравнению с ним), так что при $a_d > 0$ многочлен $p(n)$ асимптотически положителен (положителен при больших n) и можно написать $p(n) = \Theta(n^d)$.

При $a \geq 0$ функция $a \mapsto n^a$ монотонно возрастает, при $a \leq 0$ монотонно убывает. Говорят, что функция $f(n)$ полиномиально ограничена, если $f(n) = n^{O(1)}$, или, другими словами, если $f(n) = O(n^k)$ для некоторой константы k .

Экспоненты. Для любых вещественных m, n и $a \neq 0$ имеем

$$\begin{aligned} a^0 &= 1, (a^m)^n = a^{mn}, \\ a^1 &= a, (a^m)^n = (a^n)^m, \\ a^{-1} &= 1/a, a^m a^n = a^{m+n}. \end{aligned}$$

При $a \geq 1$ функция $n \mapsto a^n$ монотонно возрастает.

Мы будем иногда условно полагать $0^0 = 1$. Функция $n \mapsto a^n$ называется показательной функцией, или экспонентой. При $a > 1$ показательная функция растет быстрее любого полинома: каково бы ни было b ,

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0 \quad (1.1)$$

или, другими словами, $n^b = o(a^n)$. Если в качестве основания степени взять число $e = 2,71828 \dots$, то экспоненту можно записать в виде ряда:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{k=0}^{\infty} \frac{x^k}{k!}, \quad (1.2)$$

где $k! = 1 * 2 * 3 * \dots * k$.

Для всех вещественных x выполнено неравенство $e^x \geq 1 + x$, которое обращается в равенство лишь при $x = 0$. При $|x| \leq 1$ можно оценить e^x сверху и снизу так: $1 + x \leq e^x \leq 1 + x + x^2$. Можно сказать, что $e^x = 1 + x + \Theta(x^2)$ при $x \rightarrow 0$, имея в виду

соответствующее истолкование обозначения Θ (в котором $n \rightarrow \infty$ заменено на $x \rightarrow 0$).

При всех x выполнено равенство $\lim_{n \rightarrow \infty} (1 + \frac{x}{n})^n = e^x$.

Логарифмы. Мы будем использовать такие обозначения:

$lgn = \log_2 n$ (двоичный логарифм),

$\ln n = \log_e n$ (натуральный логарифм),

$lg^k n = (lg n)^k$,

$\lg \lg n = \lg(lgn)$ (повторный логарифм).

Мы будем считать, что в формулах знак логарифма относится лишь к непосредственно следующему за ним выражению, так что $lgn + k$ есть $\lg(n) + k$ (а не $\lg(n + k)$). При $b > 1$ функция $n \mapsto \log_b n$ (определенная при положительных n) строго возрастает.

Следующие тождества верны при всех $a > 0$, $b > 0$, $c > 0$ и при всех n (если только основания логарифмов не равны 1):

$$a = b^{\log_b a}, \log_b \left(\frac{1}{a}\right) = -\log_b a,$$

$$\log_c(ab) = \log_c a + \log_c b, \log_b a = \frac{1}{\log_a b}$$

$$\log_b a^n = n \log_b a, a^{\log_b c} = c^{\log_b a},$$

$$\log_b a = \frac{\log_c a}{\log_c b}.$$

Изменение основания у логарифма умножает его на константу, поэтому в записи типа $O(\log n)$ можно не уточнять, каково основание логарифма. Мы будем чаще всего иметь дело с двоичными логарифмами (они появляются, когда задача делится на две части) и потому оставляем за ними обозначение lg .

Для натурального логарифма есть ряд (который сходится при $|x| < 1$):

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

При $x > -1$ справедливы неравенства:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x,$$

которые обращаются в равенства лишь при $x = 0$.

Говорят, что функция $f(n)$ ограничена полилогарифмом, если $f(n) = lg^{O(1)} n$. Предел (1.1) после подстановок $n = lgt$ и $a = 2^c$ дает:

$$\lim_{m \rightarrow \infty} \frac{lg^b m}{(2^c)^{lg m}} = \lim_{m \rightarrow \infty} \frac{lg^b m}{m^c} = 0$$

и, таким образом, $lg^b n = o(n^c)$ для любой константы $c > 0$. Другими словами, любой полином растет быстрее любого полилогарифма.

Факториалы. Запись $n!$ (читается «эн факториал») обозначает произведение всех чисел от 1 до n . Полагают $0! = 1$, так что $n! = n(n-1)!$ при всех $n = 1, 2, 3, \dots$

Сразу же видно, что $n! \leq n^n$ (каждый из сомножителей не больше n). Более точная оценка дается формулой Стирлинга, которая гласит, что:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + \Theta(1/n)).$$

Из формулы Стирлинга следует, что:

$$n! = o(n^n),$$

$$n! = \omega(2^n),$$

$$lg(n!) = \Theta(n lg n).$$

Справедлива также следующая оценка:

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{1/12n}.$$

Итерации логарифма. Мы используем обозначение $lg^* n$ («логарифм со звездочкой от эн») для функции, называемой итерированным логарифмом. Эта функция определяется так. В начале рассмотрим i -ую итерацию логарифма, функцию $lg^{(i)}$, определенную так: $lg^{(0)} = n$ и $lg^{(i)}(n) = lg(lg^{(i-1)}n)$ при $i > 0$ (последнее выражение определено, если $lg^{(i-1)}n$ определено и положительно). Будьте внимательны: обозначения $lg^i n$ и $lg^{(i)} n$ внешне похожи, но означают совершенно разные функции.

Теперь $lg^* n$ определяется как минимальное число $i \geq 0$, при котором $lg^{(i)} n \leq 1$. Другими словами, $lg^* n$ – это число раз, которое нужно применить к функции lg , чтобы из n получить число, не превосходящее 1.

Функция $lg^* n$ растет исключительно медленно:

$$lg^* 2 = 1,$$

$$lg^* 4 = 2,$$

$$lg^* 16 = 3,$$

$$lg^* 65536 = 4,$$

$$\lg^* 2^{65536} = 5.$$

Поскольку число атомов в наблюдаемой части Вселенной оценивается как 10^{80} , что много меньше 2^{65536} , то значения n , для которых $\lg^* n > 5$, вряд ли могут встретиться.

Числа Фибоначчи. Последовательность чисел Фибоначчи определяется рекуррентным соотношением:

$$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2} \text{ при } i \geq 2.$$

Другими словами, в последовательности Фибоначчи 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ... каждое число равно сумме двух предыдущих. Числа Фибоначчи связаны с так называемым отношением золотого сечения φ и с сопряженным с ним числом $\hat{\varphi}$:

$$\varphi = \frac{1+\sqrt{5}}{2} = 1,61803 \dots,$$

$$\hat{\varphi} = \frac{1-\sqrt{5}}{2} = -0,61803 \dots$$

Здесь имеет место формула:

$$F_i = \frac{\varphi^i - \hat{\varphi}^i}{\sqrt{5}},$$

которую можно доказать по индукции. Поскольку $|\hat{\varphi}| < 1$, слагаемое $|\hat{\varphi}^i/\sqrt{5}|$ меньше $\frac{1}{\sqrt{5}} < 1/2$, так что F_i равно числу $\varphi^i/\sqrt{5}$, округленному до ближайшего целого.

Число F_i быстро (экспоненциально) растет с ростом i .

2 ПСЕВДОКОД

Рассматриваемые в учебном пособии алгоритмы будут формализованы с помощью псевдокода. Такое представление будет схоже с уже известными языками программирования. Конструкция псевдокода допускает некоторые упрощения:

- часть вычислений может быть описана на естественном языке для лучшего понимания;
- технологические подробности не рассматриваются (например, обработка исключений).

Опишем конструкции псевдокода с помощью алгоритма сортировки вставками. Данный алгоритм наиболее актуален для работы с короткими последовательностями. Идея алгоритма

заключается в том, чтобы брать элементы из неотсортированной части последовательности и добавлять их к отсортированной, двигаясь справа налево, пока не найдем элемент меньше вставляемого (рисунок 2.1).



Рисунок 2.1 – Пример работы алгоритма сортировки вставками для колоды карт

Ниже приведем пример записи псевдокода для процедуры СОРТИРОВКА-ВСТАВКАМИ. Входом процедуры будет массив $A[1..n]$ (длину массива будем обозначать через $length[A]$). Алгоритм работает без использования дополнительной памяти. Когда процедура СОРТИРОВКА-ВСТАВКАМИ завершает работу, мы получаем массив A , который отсортирован по возрастанию.

Псевдокод:

СОРТИРОВКА-ВСТАВКАМИ (A)

```
1 for j ← 2 to length[A]
2   do key ← A[j]
3     ▷ добавить A[j] к отсортированной части A[1..j-1]
4     i ← j-1
5     while i > 0 and A[i] > key
6       do A[i+1] ← A[i]
7         i ← i-1
8     A[i+1] ← key
```

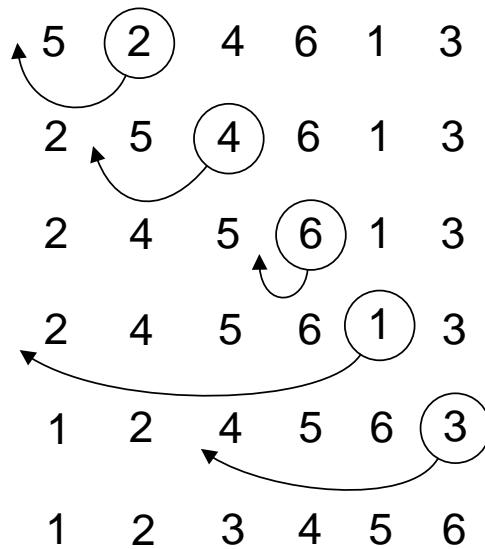


Рисунок 2.2 – Пример работы процедуры
СОРТИРОВКА-ВСТАВКАМИ

На рисунке 2.2 приведена работа алгоритма сортировки вставками для последовательности $A = \langle 6, 3, 5, 7, 2, 4 \rangle$. В основном цикле *for* процедуры определяются границы отсортированного отрезка $A[1..j - 1]$ и неотсортированного $A[j + 1..n]$. В строке 2 в дополнительную переменную *key* помещаем элемент, который собираемся добавить к упорядоченной части последовательности. В строках 5–7 анализируем отсортированную часть в поиске элемента, который меньше либо равен *key*, и перемещаем элементы, которые не соответствуют условию, вправо. К моменту выполнения строки 8 мы будем иметь позицию для вставки *key*.

Приведем основные правила написания псевдокода.

- 1 Длина отступа от левого поля показывает на степень вложенности выполняемой операции или команды. Это правило применимо для всех циклов и условных конструкций. Например, к циклу *for* (строка 1) относятся операции, выполняемые в строках со 2 по 8, строки 6–7 относятся к циклу *while* (строка 5), а строка 8 – нет. В существующих языках программирования это правило практически не используется.
- 2 Циклы *while*, *for* относятся к циклам с предусловием. Цикл *repeat* – цикл с постусловием, причем истинность условия завершает выполнение цикла. Условные конструкции *if*, *then*,

else работают точно так же, как и в существующих языках программирования.

- 3 Комментарий обозначается через символ \triangleright и завершается в конце строки.
- 4 Присваивание будем обозначать через символ \leftarrow . Одновременное присваивание конструкции вида $i \leftarrow j \leftarrow e$ говорит об одновременном присваивании переменных i и j . Сначала инициализируется переменная j , затем – i .
- 5 Переменные i , j , key , используемые в процедуре СОРТИРОВКА-ВСТАВКАМИ, будут локальными, если не определено иное.
- 6 Конструкция $A[i]$ определяет индекс i -го элемента массива. Знак «...» определяет участок массива $A[1..j]$ от 1 до j .
- 7 Для объектов, которые имеют несколько полей (атрибутов), значение поля определяется как имя_поля[имя_объекта]. Например, длину (поле объекта) массива запишем в следующем виде: $length[A]$.

Переменная, связанная с объектом, является указателем. Если x – объект, то после выполнения операции присваивания $y \leftarrow x$ для любого поля f будет выполнено равенство $f[y] = f[x]$. Теперь после изменения поля $f[x]$ будет изменяться и поле $f[y]$. Переменные x и y содержат информацию об одном и том же объекте.

Если указатель не ссылается на объект, то его значение равно *NIL*.

- 8 Входные параметры процедуры передаются по значению, фактически они становятся локальными переменными. Если в процедуру передается объект, то происходит копирование его указателя, а не полей, содержащих данные.

3 СТРУКТУРЫ ДАННЫХ

Структура данных – это способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию. Ни одна структура данных не является универсальной и не может подходить для всех целей, поэтому важно знать преимущества и ограничения, присущие некоторым из них.

В этой главе мы рассмотрим структуры данных, которые являются множествами: куча, стеки, очереди, списки и деревья. Все эти структуры данных могут изменяться во время выполнения программы, то есть являются динамическими.

Для работы конкретного алгоритма необходимо определиться с реализацией динамического множества. От этого выбора будет зависеть структура элемента множества и набор реализуемых операций.

Элементы множеств. Обычно элемент динамического множества – запись, состоящая из набора полей, которые включают одно поле – ключ, а остальные поля хранят полезную нагрузку и служебную информацию. Прежде чем выполнить операции над элементом множества, осуществляется его поиск по ключу.

В качестве служебной информации могут выступать указатели на другие элементы множества, обеспечивая тем самым его целостность.

Для ключей некоторых множеств поддерживается свойство упорядоченности (или числовой, или лексикографический порядок). Для такой реализации можем определять наименьший или наибольший элементы, следующий или предыдущий элементы и т. д.

Для каждого элемента множества будет отводиться отдельный участок общей области памяти, доступ к которому мы будем получать через указатель или индекс массива (в зависимости от языка программирования).

Операции над множествами. Операции над множествами можно разделить на две группы: изменяющие последовательность и не изменяющие последовательность. К изменяющим относятся:

- **ДОБАВЛЕНИЕ(S, x)** вставляет элемент x во множество S ;
- **УДАЛЕНИЕ(S, x)** исключает элемент x из множества S .

К не изменяющим относятся:

- **ПОИСК(S, k)** выполняет поиск элемента по ключу, соответствующему значению k . Возвращает указатель на найденный элемент или специальное значение *NIL*, если элемент отсутствует в S ;
- **МИНИМУМ(S)** возвращает указатель на элемент с минимальным ключом;

- $\text{МАКСИМУМ}(S)$ возвращает указатель на элемент с максимальным ключом;

- $\text{СЛЕДУЮЩИЙ}(S, x)$ для элемента x определяется следующий элемент (принимая во внимание порядок на ключах), и возвращается его указатель. Если x – максимальный элемент, то возвращается NIL ;

- $\text{ПРЕДЫДУЩИЙ}(S, x)$ для элемента x определяется предыдущий элемент, и возвращается его указатель. Если x – минимальный элемент, то возвращается NIL ;

Операции СЛЕДУЮЩИЙ и ПРЕДЫДУЩИЙ применимы и для множеств, допускающих дублирующиеся ключи. Операции СЛЕДУЮЩИЙ и ПРЕДЫДУЩИЙ являются симметричными. Используя МИНИМУМ и СЛЕДУЮЩИЙ можно распечатать все элементы в возрастающем порядке. Используя МАКСИМУМ и ПРЕДЫДУЩИЙ можно распечатать все элементы в убывающем порядке.

Стоимость операций над множествами определяется через количество элементов множества.

3.1 Кучи

Двоичную кучу будем представлять в виде двоичного дерева (рисунок 3.1), а для хранения элементов будем использовать массив с заданным свойством упорядоченности:

- каждой вершине соответствует индекс i массива;
- родителем вершины i будет вершина с индексом $\lfloor i/2 \rfloor$;
- детьми вершины i будут вершины с индексами $2i$ и $2i + 1$.

В дополнение к полю длина массива $\text{length}[A]$ куча имеет еще поле размер кучи $\text{heap} - \text{size}[A]$. Для размера кучи выполняется следующее неравенство $\text{heap} - \text{size}[A] \leq \text{length}[A]$. Элементы кучи будут записаны в следующем виде $A[1], \dots, A[\text{heap} - \text{size}[A]]$.

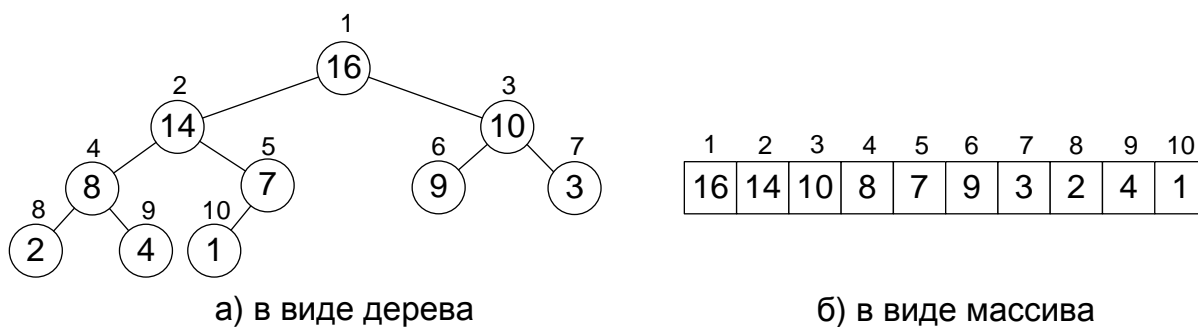


Рисунок 3.1 – Варианты представления кучи

Перемещение между вершинами дерева можно выполнять с помощью следующих процедур:

- $Parent(i) \{return \lfloor i/2 \rfloor\}$;
- $Left(i) \{return 2i\}$;
- $Right(i) \{return 2i + 1\}$.

Элемент, хранящийся по индексу 1 ($A[1]$), будет корнем дерева.

Для каждой вершины i , начиная со второй и до размера кучи, должно выполняться основное свойство кучи для сохранения упорядоченности:

$$A[Parent(i)] \geq A[i]. \quad (3.1)$$

Из неравенства (3.1) следует, что вершина-предок всегда больше своих вершин-потомков. Если основное свойство кучи сохранено для всего двоичного дерева, то наибольший элемент будет помещен в корне дерева.

Высоте дерева соответствует наибольшая длина пути от корня до листовой вершины. Для двоичного дерева, являющего кучей, все уровни заполнены полностью, кроме последнего. Поэтому высота дерева определяется как $\Theta(\lg n)$ (n – число элементов в куче). Для кучи время выполнения основных операций пропорционально высоте дерева и оценивается как $O(\lg n)$.

Приведем основные операции, реализованные для кучи:

- процедура СВОЙСТВО-КУЧА восстанавливает основное свойство кучи (3.1). Время операции определено как $O(\lg n)$;
- процедура ПОСТРОЕНИЕ-КУЧА формирует кучу из неотсортированной последовательности на базе массива. Время операции определено как $O(n)$;

- процедура СОРТИРОВКА-КУЧА выполняет сортировку массива после построения кучи без дополнительной памяти. Время операции определено как $O(n \lg n)$;
- процедура ИЗВЛЕЧЕНИЕ-МАКСИМАЛЬНЫЙ и ДОБАВЛЕНИЕ могут быть применены для создания очереди с приоритетами на основе кучи. Время обеих операции определено как $O(\lg n)$.

3.1.1 Сохранение основного свойства кучи

Процедура СВОЙСТВО-КУЧА – основная операция, восстанавливающая целостность кучи. На вход процедура получает последовательность A и индекс i элемента, для которого необходимо восстановить основное свойство кучи. Основная идея процедуры заключается в следующем:

- принимается во внимание, что поддеревья $Left(i)$ и $Right(i)$ соответствуют основному свойству кучи;
- вершина i обменивается со своими детьми для выполнения неравенства (3.1);
- сравниваются между собой дети вершины i и с самой вершиной i , больший элемент становится родителем;
- предыдущая операция повторяется для каждого изменившегося ребенка.

Для восстановления основного свойства кучи необходимо элементы $A[2]$ и $A[4]$ поменять местами (рисунок 3.2 а). После этого необходимо запустить процедуру заново для элемента $A[4]$, т. е. поменять $A[4] \leftrightarrow A[9]$ (рисунок 3.2 б). Основное свойство восстановлено для всех вершин, идея алгоритма требует запуска процедуры СВОЙСТВО-КУЧА еще раз для элемента $A[9]$ (рисунок 3.2 в).

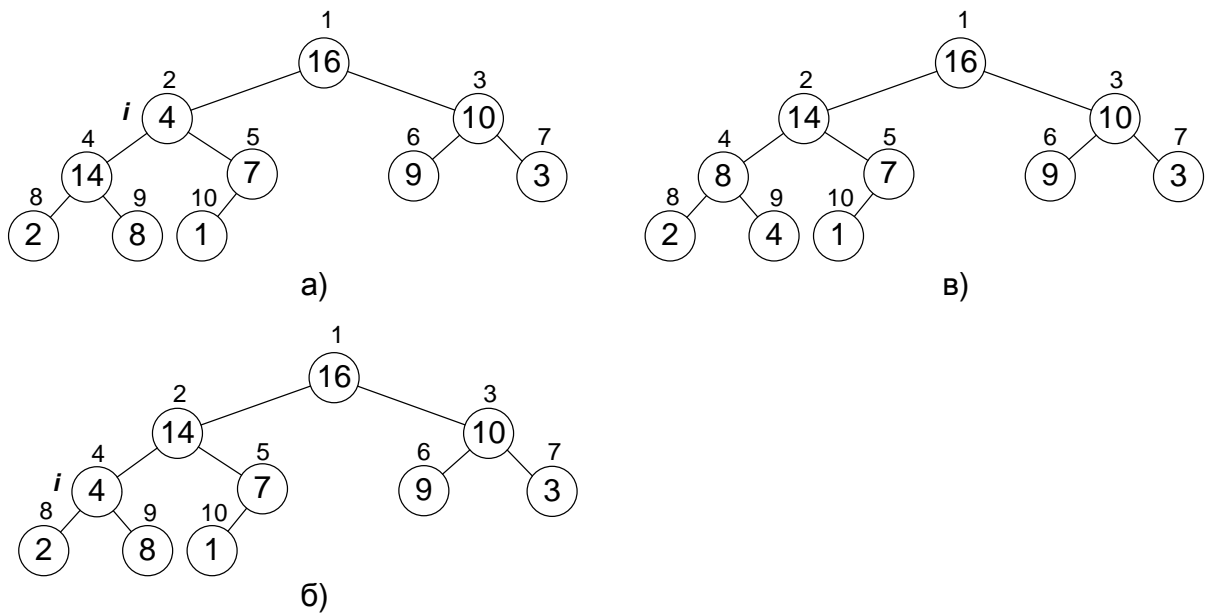


Рисунок 3.2 – Пример работы процедуры СВОЙСТВО-КУЧА (A,2) для элемента с индексом 2

СВОЙСТВО-КУЧА (A, i)

- 1 $l \leftarrow \text{Left}(i)$
- 2 $r \leftarrow \text{Right}(i)$
- 3 **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$
- 4 **then** $\text{largest} \leftarrow l$
- 5 **else** $\text{largest} \leftarrow i$
- 6 **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$
- 7 **then** $\text{largest} \leftarrow r$
- 8 **if** $\text{largest} \neq i$
- 9 **then** обменять $A[i] \leftrightarrow A[\text{largest}]$
- 10 СВОЙСТВО-КУЧА (A, largest)

Пример работы процедуры СВОЙСТВО-КУЧА изображен на рисунке 3.2. В строках 1–2 в переменных l и r сохраняются индексы детей вершины i . В строках 3–7 определяется наибольший элемент и сохраняется его индекс в переменной $largest$. Если условие в строке 8 выполняется, то необходимо восстановить основное свойство кучи и обменять элементы $A[i]$ и $A[largest]$ местами (строка 9). В строке 10 процедура вызывает сама себя для восстановления свойства кучи для элемента с индексом $largest$.

3.1.2 Построение кучи

Для превращения массива $A[1..n]$ в кучу будем применять процедуру СВОЙСТВО-КУЧА к каждой вершине, начиная с нижних

вершин, имеющих листья. Для листьев $\lfloor n/2 \rfloor + 1, \dots, n$ основное свойство кучи выполняется. Для всех остальных вершин в порядке убывания индексов будем вызывать процедуру СВОЙСТВО-КУЧА. Последовательность анализа вершин позволит сохранить основное свойство кучи для поддеревьев. Псевдокод процедуры ПОСТРОЕНИЕ-КУЧА приведен ниже.

```

ПОСТРОЕНИЕ-КУЧА (A)
1 heap-size[A] ← length[A]
2 for i ← ⌊length[A]/2⌋ downto 1
3   do СВОЙСТВО-КУЧА (A, i)

```

Пример работы процедуры ПОСТРОЕНИЕ-КУЧА можно увидеть на рисунке 3.3. В строке 1 размер кучи приравнивается к размеру массива A . В основном цикле процедуры в убывающем порядке для каждого элемента, начиная с индекса $\lfloor \text{length}[A]/2 \rfloor$ и до 1, выполняем процедуру восстановления основного свойства кучи (СВОЙСТВО-КУЧА).

Время работы процедуры ПОСТРОЕНИЕ-КУЧА оценивается как $O(n \lg n)$, которое состоит из $O(n)$ раз обращений к процедуре СВОЙСТВО-КУЧА, и время работы одного обращения будет равно $O(\lg n)$. Полученную оценку уточним ниже.

Дело в том, что время работы процедуры СВОЙСТВО-КУЧА зависит от высоты вершины, для которой она вызывается (и пропорционально этой высоте). Поскольку число вершин высоты h в куче из n элементов не превышает $\lfloor n/2^{h+1} \rfloor$, а высота всей кучи не превышает $\lfloor \lg n \rfloor$, время работы процедуры ПОСТРОЕНИЕ-КУЧА не превышает:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O \left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right). \quad (3.2)$$

Выполнив ряд преобразований в правой части формулы (3.2), получаем верхнюю оценку для суммы:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2.$$

Таким образом, время работы процедуры ПОСТРОЕНИЕ-КУЧА составляет:

$$O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).$$

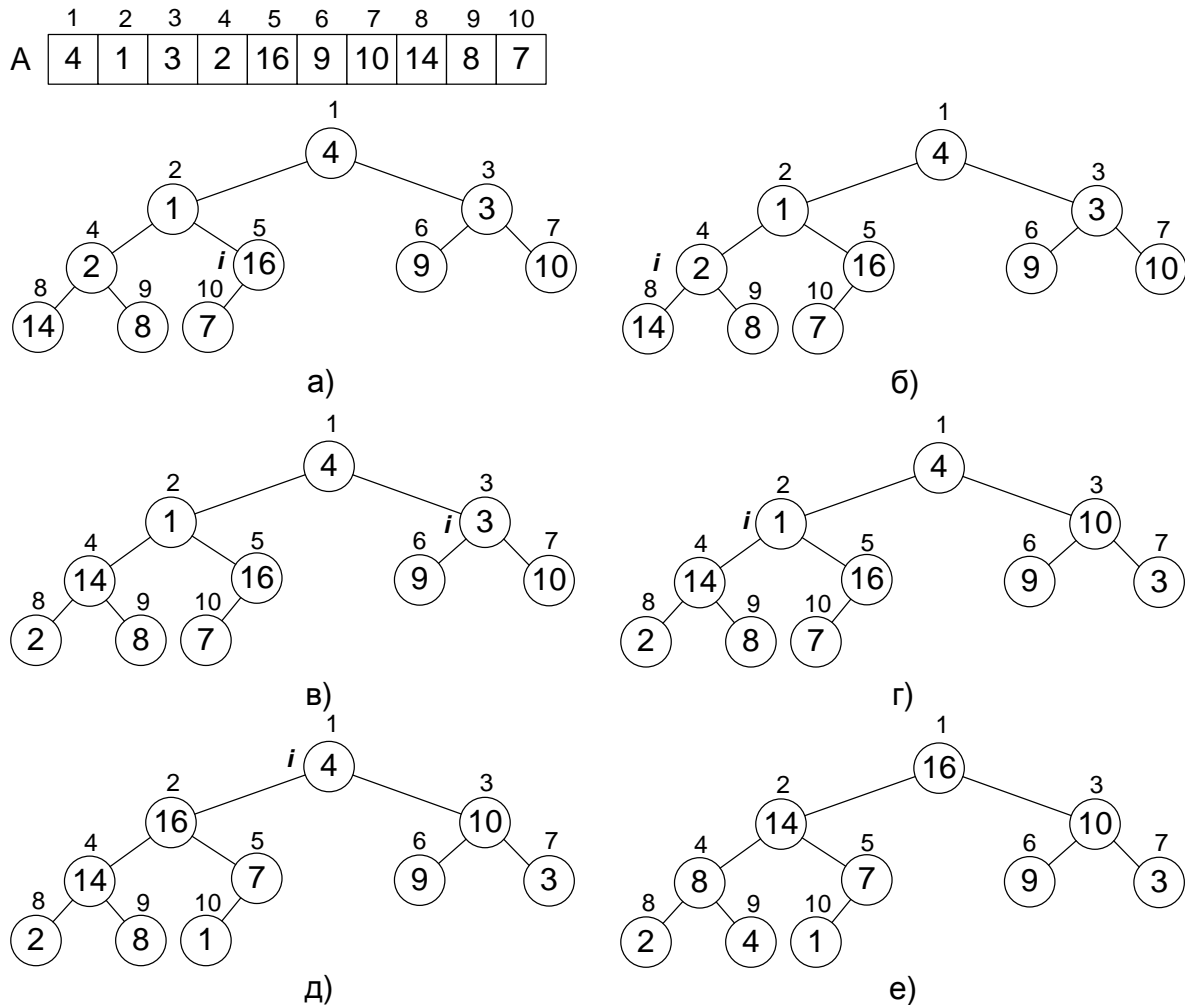


Рисунок 3.3 – Пример работы процедуры ПОСТРОЕНИЕ-КУЧА.

Состояние кучи перед каждым обращением к процедуре СВОЙСТВО-КУЧА

3.2 Стеки и очереди

Стеки и очереди относятся к динамическим множествам, для которых операция удаления всегда извлекает элемент из определенной позиции. Например, для стека можно удалить только элемент, который был добавлен последним. При этом выполняется принцип LIFO (last-in, first-out – последним пришел, первым ушел).

Из очереди удаляется элемент, который попал в очередь первым. При этом выполняется принцип FIFO (first-in, first-out – первым пришел, первым ушел). Стеки и очереди можно реализовать разными способами.

3.2.1 Стеки

Через PUSH будем обозначать операцию добавления элемента в стек, а через POP будем обозначать операцию удаления элемента, расположенного на вершине стека. Стек можно сравнить со стопкой кирпичей, из которой можно взять кирпичи только из верхнего слоя и добавить кирпичи, положив их только на верхний слой.

На рисунке 3.4 изображена работа стека, реализованного с помощью массива $A[1..n]$, состоящего из n элементов. Дополнительно вводится переменная $top[A]$, которая хранит индекс вершины стека (место последнего добавления). Таким образом, $A[top[A]]$ будет верхним элементом, а $A[1]$ – нижним элементом; всю последовательность можно записать через $A[1..top[A]]$.

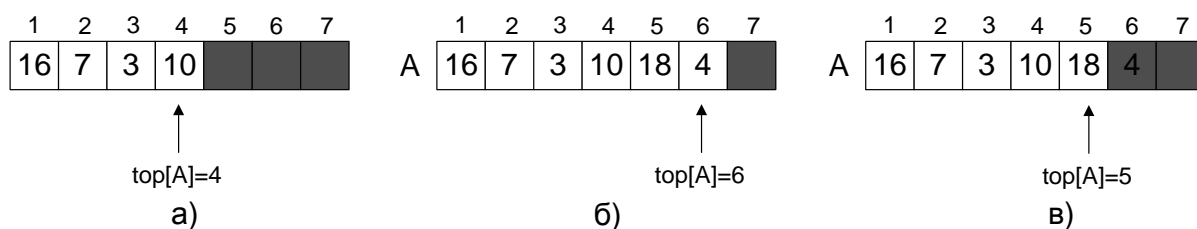


Рисунок 3.4 – Пример работы операций стека. Клетки, не входящие в стек, окрашены в темно-серый цвет

Начальное состояние стека A изображено на рисунке 3.4 а. Состояние стека A после выполнения операций PUSH(A,18) и PUSH(A,4) (рисунок 3.4 б). Состояние стека A после выполнения операции POP(A), возвращается значение 4 (рисунок 3.4 в).

Если вершина стека равна нулю $top[A] = 0$, то в стеке нет элементов. Если вершина стека совпадает с размером стека $top[A] = n$, то вставка элементов невозможна. Выполняется сигнализация о переполнении стека (overflow). Обратная ситуация, когда пытаемся удалить элемент из стека при $top[A] = 0$. Операция будет невозможна, выполняется сигнализация о недополнении стека (underflow).

Рассмотрим ниже псевдокод операций работы со стеком (проверка пустоты, добавление элемента, удаление элемента):

STACK-EMPTY (A)

```
1 if top[A] = 0
2   then return TRUE
3   else return FALSE
```

PUSH (A, x)

```
1 top[A] ← top[A]+1
2 A[top[A]] ← x
```

POP (A)

```
1 if STACK-EMPTY (A)
2   then error “underflow”
3   else top[A] ← top[A]-1
4     return A[top[A]+1]
```

Работа операций PUSH и POP изображена на рисунке 3.4. Время работы этих операций можно оценить как $O(1)$.

3.2.2 Очереди

Через ОЧЕРЕДЬ-ДОБАВИТЬ определим операцию добавления элемента к очереди, а через ОЧЕРЕДЬ-УДАЛИТЬ определим операцию удаления элемента из очереди. Позиция удаляемого элемента известна заранее, поэтому в процедуру ОЧЕРЕДЬ-УДАЛИТЬ не передается.

Для очереди вводятся два дополнительных понятия: голова и хвост. Хвост ассоциируется с последним добавленным элементом в очередь, а голова – с элементом, готовым к извлечению из очереди.

На рисунке 3.5 изображена очередь, реализованная в виде массива $A[1..n]$, свернутая в кольцо (после последнего элемента идет первый) и состоящая из $n - 1$ элемента. В переменной $head[A]$ будем сохранять индекс ячейки, в которой находится голова очереди, а в переменной $tail[A]$ – индекс следующей свободной ячейки после последнего добавленного элемента. Очередь будет включать следующие элементы с индексами $head[A], head[A] + 1, \dots, tail[A] - 1$. Очередь будет пустой, если индекс, хранящийся в голове, будет совпадать с индексом, хранящимся в хвосте, $head[A] = tail[A]$. Начальное состояние очереди соответствует $head[A] = tail[A] = 1$.

Удаление элемента из пустой очереди будет приводить к исключению *underflow*; если в очереди нет пустых ячеек $head[A] = tail[A] + 1$, то попытка вставить в нее элемент вызовет исключение *overflow*.

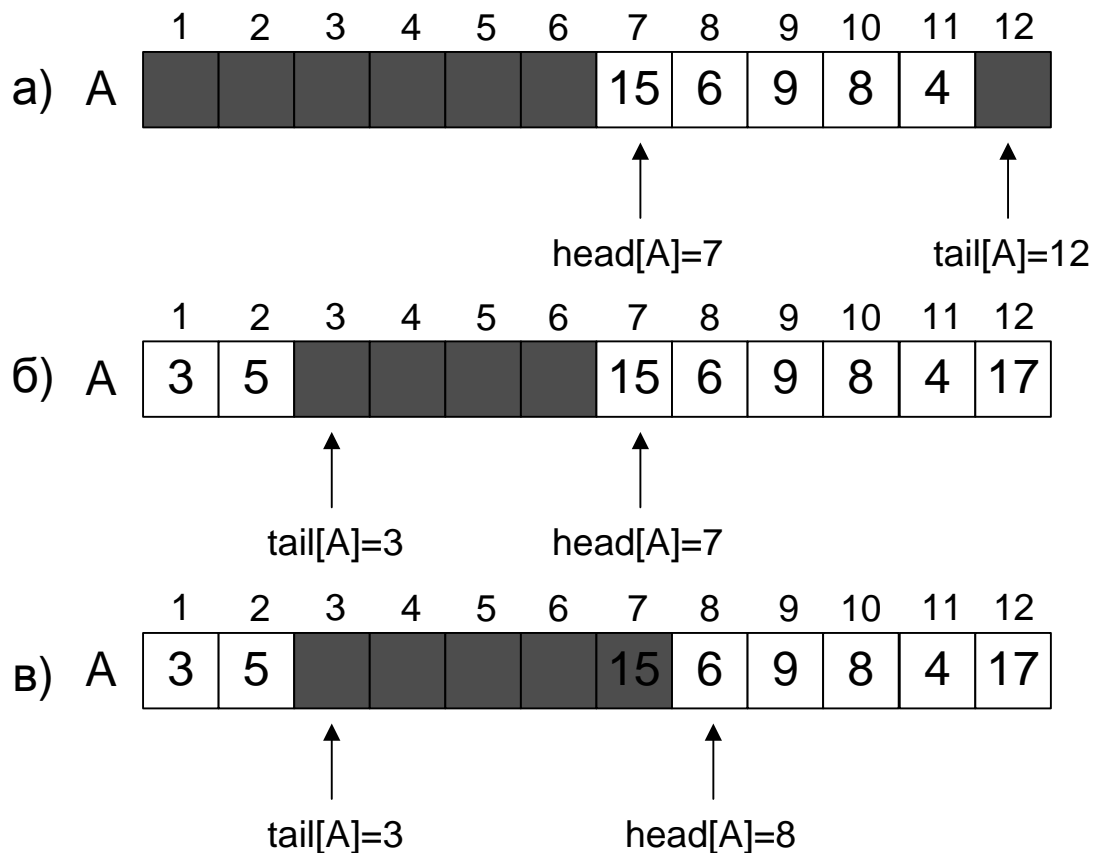


Рисунок 3.5 – Пример работы операций на очереди. Клетки, не входящие в очередь, окрашены в темно-серый цвет

Начальное состояние очереди показано на рисунке 3.5 а. В очередь добавлено три элемента: 17, 3 и 5 с помощью процедуры *ОЧЕРЕДЬ-ДОБАВИТЬ* (рисунок 3.5 б). Состояние очереди после удаления одного элемента (15) с использованием процедуры *ОЧЕРЕДЬ-УДАЛИТЬ* (рисунок 3.5 в).

Рассмотрим ниже псевдокод операций работы с очередью (добавление и удаление). В процедурах не рассматриваются случаи, приводящие к исключениям.

ОЧЕРЕДЬ-ДОБАВИТЬ (A, x)

```
1 A[tail[A]] ← x
2 if tail[A] = length[A]
3   then tail[A] ← 1
4   else tail[A] ← tail[A]+1
```

ОЧЕРЕДЬ-УДАЛИТЬ (A)

```
1 x ← A[head[A]]
2 if head[A] = length[A]
3   then head[A] ← 1
4   else head[A] ← head[A]+1
5 return x
```

Работа процедур ОЧЕРЕДЬ-ДОБАВИТЬ и ОЧЕРЕДЬ-УДАЛИТЬ продемонстрирована на рисунке 3.5. Время работы каждой процедуры можно оценить как $O(1)$.

3.3 Связанные списки

Связанный список состоит из элементов, которые включают указатели для связи с соседними узлами списка. Списки, как и массивы, представляют собой реализацию динамических множеств, для которых могут быть применимы все операции, обозначенные в начале главы 3.

Например, если человек, стоящий в очереди, запомнит, кто находится впереди него и кто находится позади него, то мы получим двухсвязный список. Для односвязного списка достаточно запомнить одного из двух рядом стоящих людей.

Схема двухсвязного списка представлена на рисунке 3.6. Каждый элемент такого списка представляет собой кортеж, включающий три служебных поля: ключ *key*, указатель на следующий элемент *next* и указатель на предыдущий элемент *prev*. Конечно же, кортеж будет обладать и дополнительными полями, в которых будет сохранена полезная нагрузка. Поле *next[x]* будет содержать информацию о следующем за *x* элементе. Поле *prev[x]* будет содержать информацию о предшествующем *x* элементу. Если поля *prev* и/или *next* не содержат информацию об элементах, то в них помещается специальное значение *NIL*. Для списков организуются два дополнительных маркера: голова списка *head* и

хвост списка *tail*. Если поле $prev[x] = NIL$, то элемент x будет головой списка. Если поле $next[x] = NIL$, то x – хвост списка.

Для обхода списка L всегда нужно начинать движение с головы списка $head[L]$. Если список не содержит ни одного элемента, то в поле $head[L]$ будет сохранено значение NIL .

Ниже приведем существующие виды списков:

- односвязный (нет поля *prev*) или двусвязный;
- упорядоченный (ключи расположены по возрастанию) или неупорядоченный;
- кольцевой (голова и хвост списка связаны друг с другом).

По умолчанию рассматриваемый список будем считать неупорядоченным двусвязным списком.

Начальное состояние двусвязного списка L приведено на рисунке 3.6 а. Состояние списка после выполнения операции СПИСОК-ДОБАВЛЕНИЕ ($L,26$) (рисунок 3.6 б). Состояние списка после выполнения операции СПИСОК-УДАЛЕНИЕ ($L,5$) (рисунок 3.6 в).

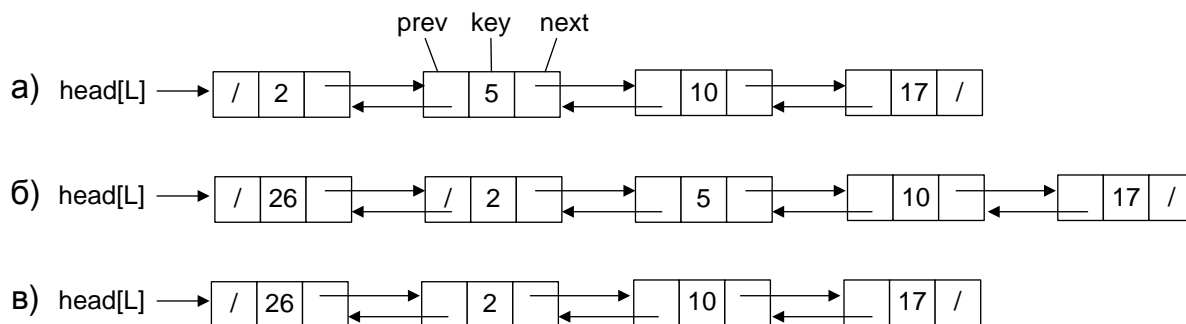


Рисунок 3.6 – Пример двусвязного списка L с элементами 2, 5, 10, 17

Поиск в списке. Процедура СПИСОК-ПОИСК (L,k) получает на вход список L и искомое значение k и выполняет поиск элемента, соответствующего ключу k . Если элемент найден, то возвращается указатель на него, иначе возвращается NIL . Рассмотрим в качестве примера список L , изображенный на рисунке 3.6 а. После выполнения процедуры СПИСОК-ПОИСК ($L,5$) будет возвращен указатель на второй элемент списка, а запуск СПИСОК-ПОИСК ($L,8$) вернет NIL . Приведем ниже псевдокод процедуры.

СПИСОК-ПОИСК (L, k)

```
1  $x \leftarrow \text{head}[L]$ 
2 while  $x \neq \text{NIL}$  and  $\text{key}[x] \neq k$ 
3   do  $x \leftarrow \text{next}[x]$ 
4 return  $x$ 
```

Время работы процедуры при необходимости просмотра всего списка будет $\Theta(n)$ операций.

Добавление элемента в список. Вставка элемента x в список L осуществляется процедурой СПИСОК-ДОБАВЛЕНИЕ в позицию головы списка (рисунок 3.6 б).

СПИСОК-ДОБАВЛЕНИЕ (L, x)

```
1  $\text{next}[x] \leftarrow \text{head}[L]$ 
2 if  $\text{head}[L] \neq \text{NIL}$ 
3   then  $\text{prev}[\text{head}[L]] \leftarrow x$ 
4  $\text{head}[L] \leftarrow x$ 
5  $\text{prev}[x] \leftarrow \text{NIL}$ 
```

Время работы процедуры можно оценить как $O(1)$.

Удаление элемента из списка. При удалении элемента из списка L с помощью процедуры СПИСОК-УДАЛЕНИЕ переписываются указатели соседних элементов. Входным параметром будет указатель на удаляемый элемент. Если удаление выполняется по ключу, то сначала необходимо запустить процедуру СПИСОК-ПОИСК, полученный указатель передать в процедуру удаления.

СПИСОК-УДАЛЕНИЕ (L, x)

```
1 if  $\text{prev}[x] \neq \text{NIL}$ 
2   then  $\text{next}[\text{prev}[x]] \leftarrow \text{next}[x]$ 
3   else  $\text{head}[L] \leftarrow \text{next}[x]$ 
4 if  $\text{next}[x] \neq \text{NIL}$ 
5   then  $\text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]$ 
```

Работа процедуры удаления показана на рисунке 3.6 в. Время работы процедуры СПИСОК-УДАЛЕНИЕ можно оценить как $O(1)$. Если удаление выполняется по ключу, то время работы будет $\Theta(n)$.

3.4 Корневые деревья

Каждый узел дерева является кортежем, содержащим несколько полей. Все поля можно разделить на ключ, дополнительную информацию и служебные данные (например, указатели на другие

узлы). Из каких конкретных полей будет состоять кортеж зависит от варианта реализации дерева.

Двоичные деревья. На рисунке 3.7 изображена схема двоичного дерева T . Служебными полями узла x будут указатели на родителя и двух детей p , $left$, $right$ соответственно. Если указатель на родителя отсутствует $p[x] = NIL$, то узел x будет корневым; если у узла отсутствует какой-нибудь ребенок, то в поле $left$ или $right$ сохраняется специальное значение NIL . Дополнительно с деревом T ассоциируется поле $root[T]$, содержащее указатель на корневую вершину. Если дерево T не содержит ни одного узла, то поле $root[T]$ будет равно NIL .

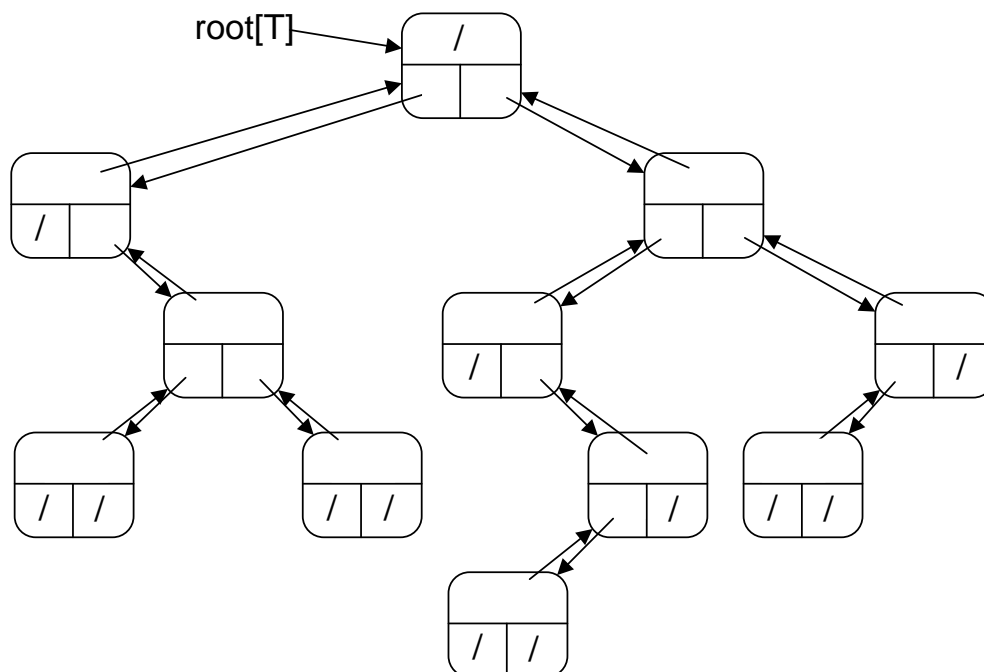


Рисунок 3.7 – Схема организации двоичного дерева T . Каждый узел содержит три служебных поля p , $left$, $right$ с визуализацией указателей на соответствующие вершины

Корневые деревья с произвольным ветвлением. Если количество детей превышает двух и равно некоторому k , то двоичное дерево обобщается до k -го дерева путем замены полей $left$ и $right$ на поля $child_1, child_2, \dots, child_k$. Если количество детей может быть любым, то такая реализация будет неэффективной.

Что же делать? Отметим, что любое произвольное дерево можно представить в виде двоичного дерева с небольшой модификацией.

При этом каждый узел сохранит два поля на детей. Левый ребенок остается без изменений, а правый ребенок трансформируется в правого соседа, т. е. станет следующим ребенком того же родителя. После проведенной модификации двоичное дерево подходит для хранения любого количества детей.

Рассмотрим более подробно схему хранения деревьев с произвольным ветвлением на основе идеи, приведенной выше:

- каждый узел x имеет поле $p[x]$, хранящее указатель на родителя;
- по-прежнему используется поле $root[T]$ для хранения корня дерева;
- каждый узел x хранит информацию о левом ребенке $left[x]$;
- каждый узел x хранит информацию о соседе справа $right - sibling[x]$.

Представленную схему называют «левый ребенок – правый сосед» (рисунок 3.8).

Если вершина x не имеет детей, тогда указатель $left[x]$ будет содержать значение NIL . Если вершина x будет последним ребенком своего родителя, то поле $right - sibling[x]$ будет равно NIL .

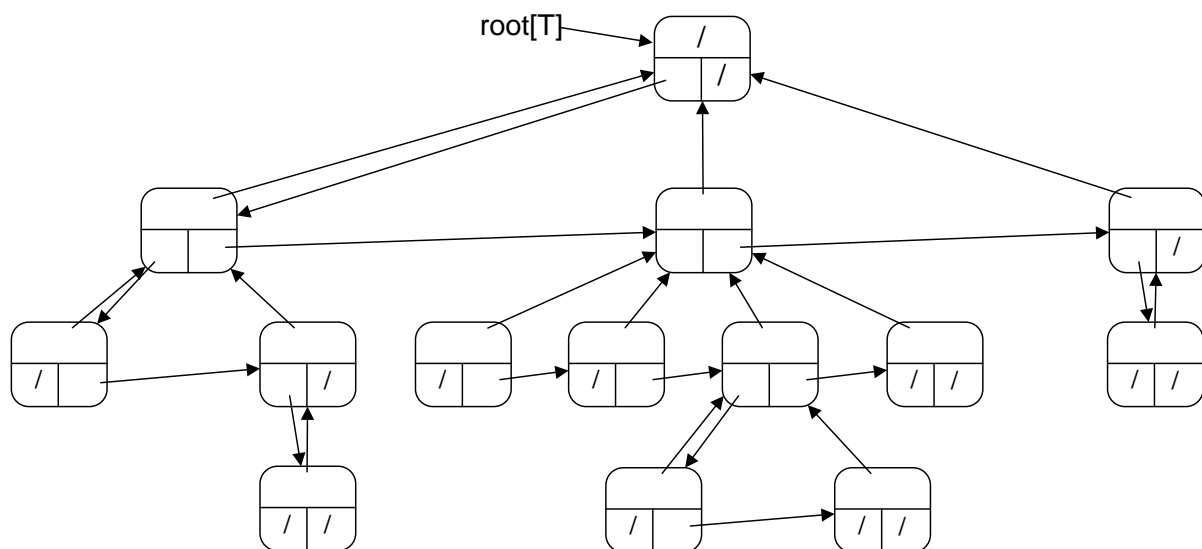


Рисунок 3.8 – Схема представления дерева T по принципу «левый ребенок – правый сосед»

4 АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Алгоритм – это формально организованная последовательность действий. Алгоритм обладает входом, через который получает исходные данные, и выходом, возвращающим результат вычислений. Внутри тела алгоритма происходит преобразование исходных данных и передача результата на выход.

Алгоритмы используются для решения различных вычислительных задач во многих областях науки и техники. Изначально формулируется задача, описывающая требования, которым должно удовлетворять решение задачи, а алгоритм, обрабатывающий эту задачу, в свою очередь формирует объект, который этим требованиям удовлетворяет.

Для более детального рассмотрения механизмов построения алгоритмов, раскрытия различных терминов и подходов обратимся к задаче сортировки, которую можно описать так:

Вход: Сформирована последовательность $\langle a_1, a_2, \dots, a_n \rangle$ из n чисел.

Выход: Выполнена перестановка $\langle a'_1, a'_2, \dots, a'_n \rangle$ исходной последовательности таким образом, что $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Рассмотрим пример, получили на вход $\langle 31, 41, 59, 26, 41, 58 \rangle$, алгоритм сортировки преобразовывает таким образом вход, что на выходе получаем $\langle 26, 31, 41, 41, 58, 59 \rangle$.

Последовательность, получаемая алгоритмом сортировки, называется входом данной задачи.

Большое количество алгоритмов применяют сортировку на начальном этапе решения комплексной задачи. Существуют различные алгоритмы сортировки. Применение того или иного алгоритма зависит от следующих условий:

- характера элементов последовательности;
- длины последовательности;
- степени упорядоченности последовательности;
- типа вычислительных ресурсов, применяемых для решения задачи (оперативная память, накопители на жестких магнитных дисках, магнитные ленты).

Алгоритмы разделяют на правильные и неправильные:

– правильные выполняют вычисления и выдают ожидаемый результат, соответствующий условиям задачи, для любого входа, принадлежащего множеству допустимых входов;

– неправильные могут не остановиться (зациклиться) или выдать неправильный результат. Если ошибки встречаются в таких алгоритмах достаточно редко, то результаты, получаемые в ходе вычислений, могут быть не бесполезными. Хотя это все же исключение, а не правило.

Четкое описание процедуры – это единственное условие формулировки алгоритма, которое может иметь вид программного модуля, блок-схемы или даже машинного кода.

4.1 Сортировки во внутренней памяти

Как уже отмечалось выше, входом для задачи сортировки должна быть последовательность чисел $\langle a_1, a_2, \dots, a_n \rangle$, а выходом – последовательность $\langle a'_1, a'_2, \dots, a'_n \rangle$, состоящая из тех же чисел, расположенных в возрастающем порядке: $a'_1 \leq a'_2 \leq \dots \leq a'_n$. Входную последовательность можно задать в виде массива, вектора, очереди, списка и т. д.

Решая практические задачи, мы не сортируем числа сами по себе. Обычно нужно сортировать кортежи, состоящие из определенного количества полей, и устанавливать порядок на этих кортежах в зависимости от одного из полей. Рассмотрим пример: в библиотеке для каждой книги хранится запись, содержащая различные поля (фамилия, имя, отчество автора, год издания, название, сколько раз ее брали читать и т. п.), и в какой-то момент может понадобиться упорядочить все записи по фамилии автора. Поле, по которому выполняется сортировка (фамилия автора в нашем примере), называется ключом, а остальные поля – вспомогательными данными (дополнительная информация). Процесс можно представить себе как упорядочивание ключей алгоритмом и перемещение (без преобразований) дополнительной информации. При большом объеме дополнительных данных перемещение лучше организовывать через указатели, не перебрасывая мегабайты информации.

Далее мы сконцентрируемся на задаче сортировки ключей. Рассматриваемые алгоритмы будут являться «каркасом» реальной

программы без обработки дополнительных данных. Реализацию обработки дополнительных полей мы оставляем на плечах практикующих программистов.

4.1.1 Сортировка слиянием

Существует немало алгоритмов, которые разработаны с применением рекурсии: решение главной задачи достигается решением более мелких задач, в которых алгоритм вызывает сам себя. Идея метода такова: задача разделяется на некоторое количество подзадач меньшего размера; затем эти подзадачи обрабатываются (с помощью рекурсивных вызовов алгоритма или напрямую, если размер достаточен); на последнем этапе результаты подзадач объединяются, и формируется решение главной задачи.

В задаче сортировки в этом случае можно выделить три этапа:

- разбиение массива на две части меньшего размера;
- сортировка каждой из последовательностей отдельно;
- объединение двух упорядоченных последовательностей половинного размера в один массив.

Первый этап сортировки выполняется до тех пор, пока размер массива не достигнет единицы (любой массив единичной длины упорядочен по определению).

Нестандартной частью задачи сортировки слиянием является третий этап. Объединение последовательностей достигается с помощью вспомогательной процедуры СЛИЯНИЕ(A , p , q , r), входными данными для которой будут массив A и числа p, q, r , обозначающие начало и конец объединяемых участков. Процедура учитывает, что $p \leq q < r$ и отрезки $A[p..q]$ и $A[q + 1..r]$ уже отсортированы, и формирует из них один отрезок $A[p..r]$.

Детальная разработка этой процедуры остается за границами нашего рассмотрения. Тем не менее мы можем предположить, что время работы процедуры СЛИЯНИЕ будет $\Theta(n)$, где n – общее количество объединяемых элементов ($n = r - p + 1$).

Давайте рассмотрим процедуру СОРТИРОВКА-СЛИЯНИЕ (A , p , r), которая упорядочивает последовательность $A[p..r]$ массива A . Если $p \geq r$, то отрезок содержит не более одного элемента, что говорит о его отсортированности. Если же $p < r$, то вычисляется

такое число q , которое делит последовательность на две примерно одинаковые части $A[p..q]$ (состоит из $\lfloor n/2 \rfloor$ элементов) и $A[q + 1..r]$ (состоит из $\lfloor n/2 \rfloor$ элементов).

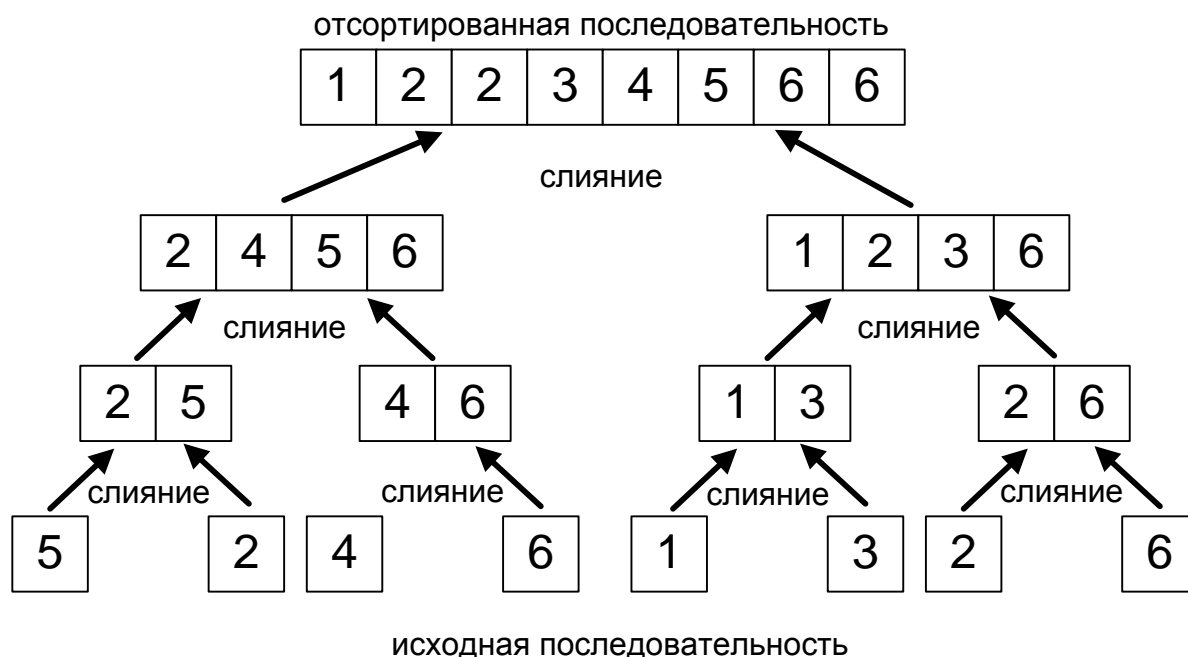


Рисунок 4.1 – Пример работы процедуры СОРТИРОВКА-СЛИЯНИЕ

СОРТИРОВКА-СЛИЯНИЕ (A, p, r)

- 1 **if** $p < r$
- 2 **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$
- 3 СОРТИРОВКА-СЛИЯНИЕ (A, p, q)
- 4 СОРТИРОВКА-СЛИЯНИЕ (A, q+1, r)
- 5 СЛИЯНИЕ (A, p, q, r)

Для сортировки всей последовательности обращаемся к процедуре СОРТИРОВКА-СЛИЯНИЕ с параметрами (A, 1, $length[A]$). В строке 1 определяется необходимость продолжения рекурсивных вызовов. В строке 2 определяется граница для разбиения последовательности. В строках 3–4 процедура вызывает сама себя с новыми границами подпоследовательностей. Наконец, в строке 5 происходит объединение отсортированных участков. Работа процедуры СОРТИРОВКА-СЛИЯНИЕ изображена на рисунке 4.1.

4.1.2 Сортировка с помощью кучи

Время работы алгоритма сортировки с помощью кучи можно оценить как $O(n \lg n)$. Размер используемой дополнительной памяти равен $O(1)$. Отметим, что алгоритм обладает преимуществами двух ранее рассмотренных алгоритмов: сортировки слиянием и сортировки вставками. Модель данных, применяемая алгоритмом, – «двоичная куча» (двоичное дерево без порядка на детях).

Алгоритм сортировки с помощью кучи состоит из двух компонентов:

– ПОСТРОЕНИЕ-КУЧА, по результатам работы которой массив трансформируется в кучу;

– максимальный элемент массива, расположенный в корне дерева ($A[1]$), обменивается с элементом в позиции $A[n]$, далее уменьшается размер кучи на единицу и запускается восстановление основного свойства кучи в корне (процедура СВОЙСТВО-КУЧА). После этого в корневой вершине вновь будет расположен максимальный из тех, что остались в двоичной куче. Работа второго компонента повторяется до тех пор, пока куча не будет состоять из одного элемента.

```
СОРТИРОВКА-КУЧА (A)
1 ПОСТРОЕНИЕ-КУЧА (A)
2 for i ← length[A] downto 2
3   do поменять A[1] ↔ A[i]
4     heap-size[A] ← heap-size[A]-1
5     СВОЙСТВО-КУЧА (A, 1)
```

Работа второго компонента алгоритма продемонстрирована на рисунке 4.2. Можно видеть структуру кучи перед каждым обращением к циклу **for** (2 строка псевдокода).

Время работы процедуры СОРТИРОВКА-КУЧА можно оценить как $O(n \lg n)$. Очевидно, первый компонент (построение кучи) требует времени $O(n)$, а каждое из $n - 1$ обращений ко второму компоненту алгоритма можно оценить как $O(\lg n)$.

Перед выполнением 5 строки процедуры СОРТИРОВКА-КУЧА зафиксировано состояние массива (рисунок 4.2). Элементы, отмеченные темно-серым, находятся за пределами кучи.

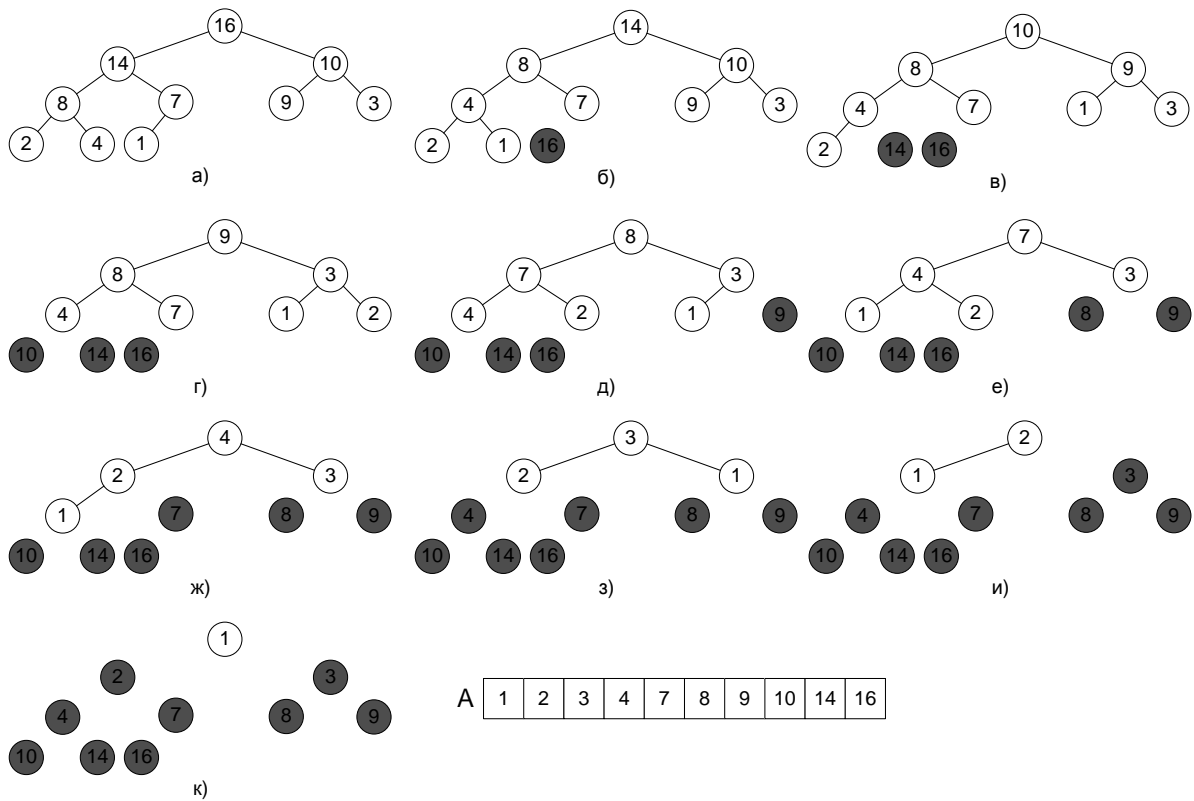


Рисунок 4.2 – Пример работы процедуры СОРТИРОВКА-КУЧА

4.1.3 Быстрая сортировка

Теперь давайте обратимся к алгоритму «быстрой сортировки», который в большинстве случаев является одним из самых эффективных. Время работы этого алгоритма для последовательности из n чисел при неблагоприятном раскладе оценивается как $\Theta(n^2)$, математическое ожидание оценивается как $\Theta(n \lg n)$, причем коэффициент при $n \lg n$ невелик. Быстрая сортировка обходится без дополнительной памяти.

Алгоритм быстрой сортировки относится к рекурсивным алгоритмам, которые главную задачу разбивают на несколько более простых и решают их по отдельности, используя так называемый принцип «разделяй и властвуй». Сортировка отрезка $A[p..r]$ выполняется следующим образом:

- элементы последовательности A перемещаются таким образом, чтобы элементы $A[p], \dots, A[q]$ не превосходили элементы $A[q + 1], \dots, A[r]$, где q – некоторое число, взятое из интервала $p \leq q < r$. Эта часть алгоритма отвечает за разбиение последовательности на две подпоследовательности;

- процедура сортировки рекурсивно вызывается для массивов $A[p..q]$ и $A[q + 1..r]$.

После этого последовательность $A[p..r]$ упорядочена.

Итак, запишем псевдокод для процедуры сортировки СОРТИРОВКА-БЫСТРАЯ:

```

СОРТИРОВКА-БЫСТРАЯ (A, p, r)
1  if p < r
2    then q ← РАЗБИЕНИЕ (A, p, r)
3      СОРТИРОВКА-БЫСТРАЯ (A, p, q)
4      СОРТИРОВКА-БЫСТРАЯ (A, q+1, r)

```

Для сортировки всей последовательности нужно запустить процедуру СОРТИРОВКА-БЫСТРАЯ (A, 1, $length[A]$).

Основная компонента алгоритма – процедура РАЗБИЕНИЕ, которая перемещает элементы последовательности $A[p..r]$ необходимым образом:

```

РАЗБИЕНИЕ (A, p, r)
1  x ← A[p]
2  i ← p-1
3  j ← r+1
4  while TRUE
5    do repeat j ← j-1
6      until A[j] ≤ x
7    repeat i ← i+1
8      until A[i] ≥ x
9    if i < j
10     then поменять A[i] ↔ A[j]
11    else return j

```

Ход работы процедуры РАЗБИЕНИЕ изображен на рисунке 4.3. Сначала в качестве пограничного элемента определяем $x = A[p]$; отрезок $A[p..q]$ будет состоять из элементов, которые не превосходят x , а отрезок $A[q + 1..r]$ – элементы, превосходящие x . Суть заключается в том, чтобы располагать элементы, не превосходящие x , в начальной части последовательности ($A[p..i]$), а элементы, превосходящие x , – в верхней части массива ($A[j..r]$). В начале обе части пусты: $i = p - 1, j = r + 1$.

Во время работы цикла while (5–8 строки псевдокода) в верхнюю и нижнюю часть массива добавляются элементы (как минимум по одному). После отработки этих строк алгоритма x удовлетворяет условиям $A[i] \geq x \geq A[j]$. Если мы обменяем значения элементов $A[i]$ и $A[j]$ между собой, то они готовы к присоединению к начальному и конечному отрезкам.

Цикл завершает свою работу, когда неравенство $i \geq j$ становится истинным. В этот момент последовательность разделена на части $A[p], \dots, A[j]$ и $A[j + 1], \dots, A[r]$; каждый представитель первой части не больше любого представителя второй части. Процедура передает на выход значение j .

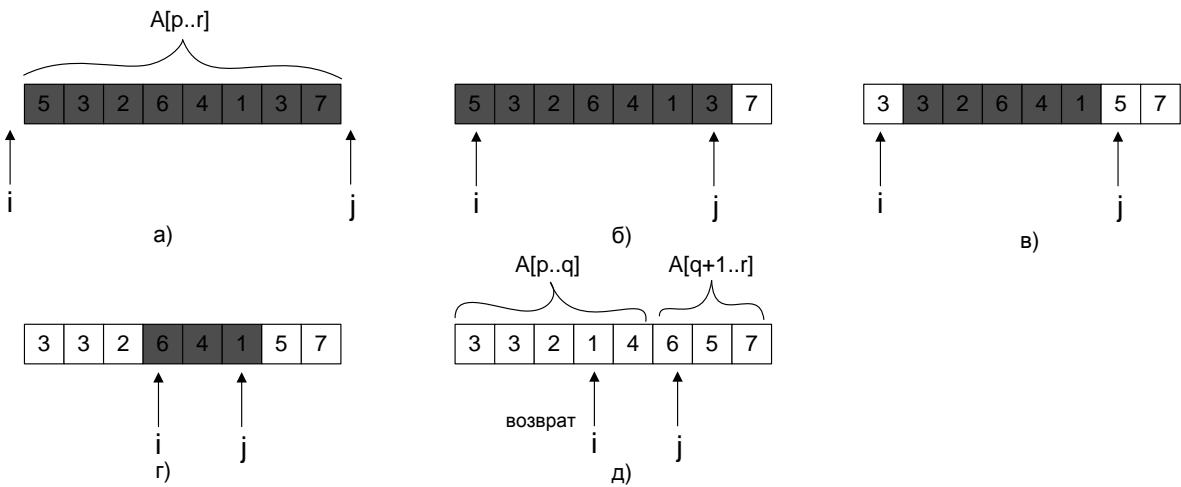


Рисунок 4.3 – Пример работы процедуры РАЗБИЕНИЕ.

Неотмеченные элементы принадлежат к обработанным отрезкам последовательности, зачерненные еще не прошли обработку

Исходное состояние последовательности, верхний и нижний накопители еще пусты, показано на рисунке 4.3 а. Элемент $x = A[p] = 5$ выбран в роли пограничного. Результат работы процедуры после выполнения первого прохода цикла в строках 4–8 псевдокода (рисунок 4.3 б). В 10 строке процедуры элементы $A[i]$ и $A[j]$ переставляются (рисунок 4.3 в). Результат работы процедуры после выполнения второго прохода цикла (рисунок 4.3 г). Результат работы процедуры после выполнения последнего прохода цикла (рисунок 4.3 д). Так как $i \geq j$, алгоритм завершает работу и отправляет на выход процедуры значение $q = j$. Элементы левее $A[j]$ (включая сам этот элемент) не превосходят значения $x = 5$, а элементы правее $A[j]$ превосходят значение $x = 5$.

В заключение стоит отметить, что вычислительная процедура доступна для понимания, но сам алгоритм имеет несколько неочевидных моментов. Во-первых, нет гарантии, что указатели i и j не покидают границ последовательности $[p..r]$ в процессе работы. Во-вторых, никак не обосновывается, что в качестве пограничного элемента выбирается самый левый $A[p]$, а не, например, правый $A[r]$. Если будет выбран самый правый элемент $A[r]$ и он окажется самым большим в последовательности, то после завершения процедуры переменные указатели будут равны $i = j = r$. В этом случае значение $q = j$, а его передавать на выход нельзя – нарушится условие $q < r$, и процедура СОРТИРОВКА-БЫСТРАЯ будет выполняться бесконечно.

Время работы процедуры РАЗБИЕНИЕ можно оценить как $\Theta(n)$, где $n = r - p + 1$.

4.1.4 Сортировка подсчетом

Следующий алгоритм, который мы рассмотрим, – сортировка подсчетом. Область применения алгоритма ограничена использованием целых положительных чисел с заранее известным максимальным k (если $k = O(n)$, то время работы алгоритма – $O(n)$).

Суть алгоритма заключается в следующем:

- определяем для каждого элемента последовательности x количество элементов, которые меньше этого элемента;
- записываем его в выходную последовательность в соответствии с этим числом (например, 25 элементов входной последовательности меньше x , то в выходной последовательности x помещается на место с индексом 26);
- если последовательность содержит повторяющиеся элементы, то вычисления немного дополняются условиями, чтобы не поместить элементы по одному и тому же индексу.

Псевдокод, который рассматривается ниже, применяет дополнительный массив $C[1..k]$ из k элементов. Исходная последовательность помещается в массиве $A[1..n]$, упорядоченная последовательность размещается в массиве $B[1..n]$.

СОРТИРОВКА-ПОДСЧЕТ (A, B, k)

```

1  for i ← 1 to k
2  do C[i] ← 0
3  for j ← 1 to length[A]
4  do C[A[j]] ← C[A[j]]+1
5  ▷ C[i] равно количеству элементов, равных i
6  for i ← 2 to k
7  do C[i] ← C[i]+C[i-1]
8  ▷ C[i] равно количеству элементов, не превосходящих i
9  for j ← length[A] downto 1
10 do B[C[A[j]]] ← A[j]
11   C[A[j]] ← C[A[j]]-1

```

Техника работы алгоритма сортировки подсчетом продемонстрирована на рисунке 4.4. В строках 1–2 выполняется начальная инициализация дополнительного массива C . В строках 3–4 выполняется подсчет одинаковых элементов последовательности, затем в строках 6–7 вычисляем расчет частичных сумм для элементов $C[2], C[3], \dots, C[k]$. В последнем цикле *for* для каждого элемента исходной последовательности A определяется нужный индекс в выходном массиве B .

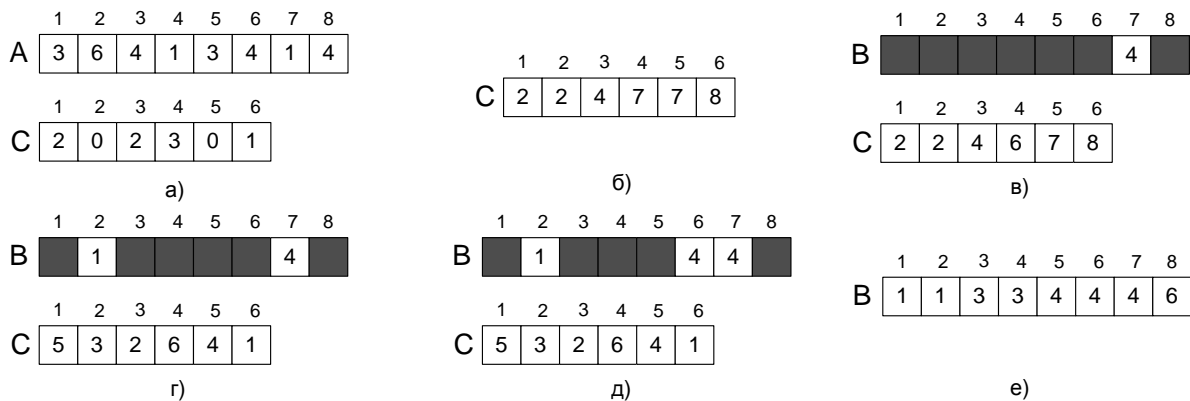


Рисунок 4.4 – Пример работы алгоритма СОРТИРОВКА-ПОДСЧЕТ применительно к массиву $A[1..8]$, состоящему из целых положительных чисел с максимальным $k = 6$

Состояние массива A и дополнительного массива C после завершения работы процедуры в 3 и 4 строках (рисунок 4.4 а). Состояние массива C после завершения работы процедуры в 6 и 7 строках (рисунок 4.4 б). Состояние массивов B и C после первой, второй и третьей итерации цикла *for* с 9 по 11 строки (рисунок 4.4 в–д). Закрашенные ячейки клетки свидетельствуют о том, что элементы

еще не заполнены для них. Состояние массива B завершения вычислений по процедуре (рисунок 4.4 е).

Если в последовательности все элементы различны, то число $A[j]$ будет находиться по индексу $C[A[j]]$, потому что такое количество чисел исходной последовательности A меньше $A[j]$; если в исходной последовательности A есть одинаковые элементы, то после каждого размещения числа в результирующем массиве B для значения в ячейке $C[A[j]]$ выполняется декремент в строке 11 процедуры, тем самым обеспечивается размещение такого же числа в ячейку с другим индексом в результирующей последовательности.

Давайте определим время работы алгоритма сортировки подсчетом: в строках 1–2 и 6–7 циклы примем за $O(k)$, в строках 3–4 и 10–11 – за $O(n)$. Таким образом, всю процедуру можно оценить как $O(k + n)$. Если k сопоставимо с n $k = O(n)$, то общее время можно оценить как $O(n)$.

Для алгоритма сортировки подсчетом характерно одно полезное свойство, называемое устойчивостью. Если мы сортируем ключи, то это свойство никак не проявляется, но, как мы уже говорили выше, сортируют обычно кортежи, состоящие из большого количества дополнительных полей. Сохранение относительного расположения этих полей относительно друг друга после выполнения сортировки и есть устойчивость. Обратимся еще раз к примеру с библиотекой. На полке стоят книги разных авторов, у Автора 1 на полке находятся десять книг, задача – отсортировать все книги на полке по фамилии автора. После завершения сортировки книги Автора 1 сохранят свое расположение относительно друг друга.

4.2 Сортировки во внешней памяти

Теперь давайте рассмотрим группу алгоритмов, выполняющих сортировку во внешней памяти. Целью таких сортировок являются последовательные файлы (чтение выполняется в последовательном режиме, а добавление данных – только после последней записи) большого размера, которые не могут быть размещены в оперативной памяти. Такой тип сортировки эффективен в системах управления базами данных и влияет на производительность СУБД.

Методы внешней сортировки не потеряли свою актуальность даже с появлением устройств, обеспечивающих «прямой» доступ к любому блоку информации. Так как единственным способом ускорения чтения/записи является более близкое расположение последовательно адресуемых блоков файла. Следует отметить, что режим работы с файлами должен оставаться последовательным.

Именно с такими файлами, решая задачи сортировки, работают современные СУБД.

Хотелось бы отметить, что время работы внешней сортировки будет сильно зависеть от объема задействованной оперативной памяти. Чем больший размер подпоследовательности вы будете загружать в оперативную память и обрабатывать ее методами внутренней сортировки, тем быстрее в целом будет выполняться алгоритм внешней сортировки.

4.2.1 Прямое слияние

Алгоритм простого слияния можно использовать как один из методов прямого слияния.

Рассмотрим пример последовательного файла A и дополнительных файлов $B1$ и $B2$. Файл A состоит из чисел a_1, a_2, \dots, a_n (n представляет собой степень числа 2). Размер файлов $B1$ и $B2$ не превосходит $n/2$.

В общем виде алгоритм прямого слияния заключается в следующем: распределение элементов из файла A в файлы $B1$ и $B2$ и последующее слияние файлов $B1$ и $B2$ в файл A .

Детально процедура прямого слияния выглядит следующим образом:

– на первой итерации последовательно анализируется файл A , элементы a_1, a_3, \dots, a_{n-1} записываются в файл $B1$, а элементы a_2, a_4, \dots, a_n – в файл $B2$. Объединение выполняется над парами элементов $(a_1, a_2), (a_3, a_4), \dots, (a_{n-1}, a_n)$, результат записывается в файл A ;

– на второй итерации снова последовательно анализируется файл A , и в файл $B1$ записываются последовательные двойки элементов с нечетными номерами, а в файл $B2$ – с четными. При

объединении формируются упорядоченные четверки элементов и записываются в файл A ;

– итерации повторяются до тех пор, пока размер подпоследовательности не превысит $n/2$;

– на последней итерации файл A будет состоять из двух упорядоченных подпоследовательностей размером $n/2$ каждая. При разбиении первая записывается в файл $B1$, а вторая – в файл $B2$. После объединения в файле A все элементы будут отсортированы.

Пример внешней сортировки простым слиянием файла $\langle 9, 22, 6, 64, 45, 32, 2, 5 \rangle$ выглядит следующим образом:

– первая итерация,

разбиение на

файл $B1$: $\langle 9, 6, 45, 2 \rangle$

файл $B2$: $\langle 22, 64, 32, 5 \rangle$

объединение файла A : $\langle 9, 22, 6, 64, 32, 45, 2, 5 \rangle$;

– вторая итерация,

разбиение на

файл $B1$: $\langle 9, 22, 32, 45 \rangle$

файл $B2$: $\langle 6, 64, 2, 5 \rangle$

объединение файла A : $\langle 6, 9, 22, 64, 2, 5, 32, 45 \rangle$;

– третья итерация,

разбиение на

файл $B1$: $\langle 6, 9, 22, 64 \rangle$

файл $B2$: $\langle 2, 5, 32, 45 \rangle$

объединение файла A : $\langle 2, 5, 6, 9, 22, 32, 45, 64 \rangle$.

Работа алгоритма прямого слияния требует всего лишь двух переменных в оперативной памяти для размещения элементов из файлов $B1$ и $B2$. Чтение и запись файлов A , $B1$ и $B2$ выполняются $O(\lg n)$ раз.

4.2.2 Естественное слияние

Алгоритм прямого слияния не учитывает то, что файл, поступающий в обработку, может быть частично упорядочен, т. е. иметь отсортированные цепочки элементов. Цепочкой называется отрезок из элементов a_i, a_{i+1}, \dots, a_j такой, что $a_k \leq a_{k+1}$ для всех $i \leq k < j$, а $a_j > a_{j+1}$. Суть алгоритма естественного слияния

сводится к нахождению цепочек при разбиении на вспомогательные файлы и последующем их объединении.

В общем виде алгоритм естественного слияния заключается в следующем: сначала выполняется распределение цепочек файла *A* по файлам *B1* и *B2*, а затем – объединение цепочек из файлов *B1* и *B2* в файл *A*.

Детально процедура естественного слияния выглядит следующим образом:

– на первой итерации определяется первая цепочка элементов и записывается в файл *B1*, вторая цепочка – в файл *B2*, третья – в файл *B1*, четвертая – в файл *B2* и т. д. При объединении первая цепочка элементов из файла *B1* упорядочивается с первой цепочкой из файла *B2*, вторая цепочка из *B1* со второй цепочкой из *B2* и т. д;

– на второй итерации операции распределения цепочек элементов по вспомогательным файлам и их объединение в файле *A* повторяются. При слиянии файлов *B1* и *B2*, если анализ одного из них завершается раньше другого, то остаток непросмотренных элементов записывается в конец файла *A*;

– вычисления завершаются, когда файл *A* состоит ровно из одной цепочки элементов.

Пример внешней сортировки естественным слиянием файла $\langle 9, 22, 6, 64, 45, 32, 2, 5 \rangle$ выглядит следующим образом:

– первая итерация,

разбиение на

файл *B1*: $\langle 9, 22, 45, 2, 5 \rangle$

файл *B2*: $\langle 6, 64, 32 \rangle$

объединение файла *A*: $\langle 6, 9, 22, 64, 32, 45, 2, 5 \rangle$;

– вторая итерация,

разбиение на

файл *B1*: $\langle 6, 9, 22, 64, 2, 5 \rangle$

файл *B2*: $\langle 32, 45 \rangle$

объединение файла *A*: $\langle 6, 9, 22, 32, 45, 64, 2, 5 \rangle$;

– третья итерация,

разбиение на

файл *B1*: $\langle 6, 9, 22, 32, 45, 64 \rangle$

файл *B2*: $\langle 2, 5 \rangle$

объединение файла A : $\langle 2, 5, 6, 9, 22, 32, 45, 64 \rangle$.

Можно утверждать, что количество операций чтения и записи файлов в худшем случае будет таким же, как в алгоритме прямого слияния, а в среднем количество будет меньше. К недостаткам алгоритма естественного слияния следует отнести: большее количество сравнений за счет распознавания концов цепочек элементов; размерность одного из вспомогательных файлов B_1 и B_2 может быть близка к размеру исходного файла A (т. к. длина цепочек элементов может быть произвольной).

4.2.3 Многопутевое слияние

В общем виде алгоритм многопутевого слияния заключается в следующем: происходит разбиение цепочек элементов исходного файла A по n вспомогательным файлам B_1, B_2, \dots, B_n и их объединение в n вспомогательных файлов C_1, C_2, \dots, C_n .

Детально процедура многопутевого слияния выглядит так:

– на первой итерации определяется первая цепочка элементов и записывается в файл B_1 , вторая цепочка – в файл B_2 , третья – в файл B_3 , четвертая – в файл B_4 и т. д. При объединении первые цепочки элементов из файлов B_1, B_2, \dots, B_n упорядочиваются между собой и размещаются в файле C_1 , вторые цепочки из файлов B_1, B_2, \dots, B_n упорядочиваются между собой и размещаются в файле C_2 и т. д.;

– на второй итерации происходит объединение по вспомогательным файлам B_1, B_2, \dots, B_n , а затем – в файлах C_1, C_2, \dots, C_n ;

– итерации объединения повторяются до тех пор, пока в файле B_1 или C_1 не будет сформирована одна единственная цепочка.

Рассмотрим работу алгоритма многопутевого слияния на примере сортировки трехпутевым слиянием файла $\langle 9, 22, 6, 64, 45, 32, 2, 5 \rangle$:

– первая итерация,

разбиение на

файл B_1 : $\langle 9, 22, 32 \rangle$

файл B_2 : $\langle 6, 64, 2, 5 \rangle$

файл B_3 : $\langle 45 \rangle$

объединение в

файл $C1$: <6, 9, 22, 45, 64>
 файл $C2$: <2, 5, 32>
 файл $C3$: <>;
 – вторая итерация,
 объединение в
 файл $B1$: <2, 5, 6, 9, 22, 32, 45, 64>
 файл $B2$: <>
 файл $B3$: <>
 объединение в
 файл $C1$: <>
 файл $C2$: <>
 файл $C3$: <>.

Из примера видно, что с увеличением длины цепочек большее количество вспомогательных файлов остается не задействовано.

Количество итераций алгоритма многопутевого слияния оценивается как $O(\log_n n)$ (n – число элементов в исходной последовательности), количество копирований элементов – как $O(\lg n)$. Число сравнений будет таким же, как и в алгоритме прямого слияния.

4.3 Поиск подстрок

Давайте детально рассмотрим задачу поиска подстрок. Мы будем оперировать понятиями: «текст» – последовательность $T[1..n]$ длины n и «образец» – последовательность $P[1..m]$ длины m . Элементами последовательностей P и T будем считать символы из алфавита Σ (алфавит может быть любым и определяется на начальном этапе задачи поиска подстрок: $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$, $\Sigma = \{a, б, в, \dots, я\}$ и т. д.). Данные последовательности принято называть строками символов или словами в алфавите Σ .

Сдвиг s (или вхождение с позиции $s + 1$) определяется как посимвольное сравнение образца P и текста T при условии, что $T[s + j] = P[j]$ и $1 \leq j \leq m$, а сдвиг изменяется в пределах $0 \leq s \leq n - m$.

Если все символы последовательности P совпали с выбранными символами текста T , то сдвиг s является допустимым, иначе –

недопустимый сдвиг. Основная цель задачи поиска подстрок – определение всех допустимых сдвигов образца P в тексте T (рисунок 4.5).

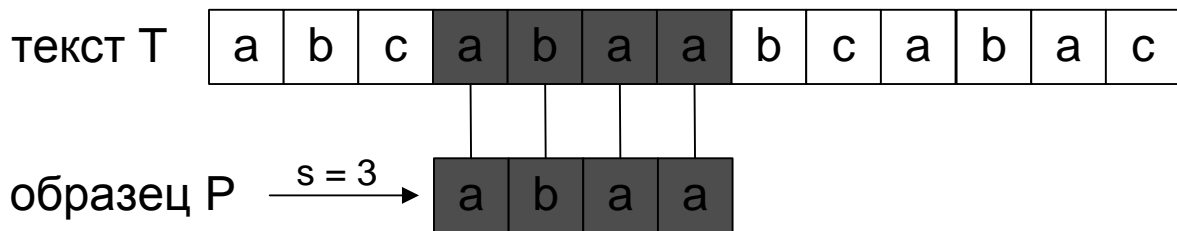


Рисунок 4.5 – Поиск подстрок. Необходимо определить все сдвиги образца P в тексте T . Единственный допустимый сдвиг равен трем

Множество всех возможных строк из алфавита определим через Σ^* . Через ε будем определять пустую строку (ее длина равна нулю). Через $||$ будем обозначать длину строки. Объединение (конкатенация) строк осуществляется следующим образом: сначала записывается первая строка x , затем ей в конец записывается вторая строка y . Эта операция обозначается как xy ; длина конкатенированной строки равна $|xy| = |x| + |y|$.

ω является префиксом строки x , если $x = \omega u$ для отдельно взятого $u \in \Sigma^*$. ω определяется как суффикс строки x , если $x = u\omega$ для отдельно взятого $u \in \Sigma^*$. Определим префикс строки x как $\omega [x$, суффикс строки x – как $\omega]x$. Например, $aba[ababaca$ и $aca]ababaca$.

ε считается префиксом и суффиксом любой строки; если ω является одновременно и префиксом, и суффиксом строки x , то следующее неравенство имеет место быть: $|\omega| \leq |x|$. Равносильными соотношениями будут $x] y$ и $xa] ya$, где a – произвольный символ; операции определения префикса $[$ и суффикса $]$ транзитивны.

Если $S[1..r]$ – строка длины r , то ее префикс длины $k \leq r$ будет обозначаться $S_k = S[1..k]$ (в частности, $S_0 = \varepsilon$ и $S_r = S$). С использованием обозначений, сформулированных выше, поиск подстроки P длины m в строке T длины n сводится к определению всех сдвигов s из интервала $0 \leq s \leq n - m$, что $P] S_{s+m}$.

При рассмотрении псевдокода для поиска подстрок равенство двух строк определяется как атомарная операция. На самом деле

время, затраченное на сравнение символов строк, пропорционально длине строк. Стоимость сравнения строк x и y при обходе слева направо есть $\Theta(t + 1)$, где t – длина наибольшего общего префикса строк x и y . Плюс одно сравнение тратится для первых отличающихся символов.

4.3.1 Простейший алгоритм

Наиболее простой алгоритм поиска образца P в тексте T на каждом шаге проверяет равенство $P[1..m] = T[s + 1..s + m]$ для всех $n - m + 1$ потенциально допустимых сдвигов s :

ПОИСК-ПОДСТРОК-ПРОСТЕЙШИЙ (T, P)

1 $n \leftarrow \text{length}[T]$

2 $m \leftarrow \text{length}[P]$

3 **for** $s \leftarrow 0$ **to** $n - m$

4 **do if** $P[1..m] = T[s + 1..s + m]$

5 **then** print «Подстрока входит со сдвигом» s

Визуально процедуру поиска можно представить следующим образом:

- последовательность P соотносится с началом текста T , и выполняется сравнение всех символов. Сдвиг s равен нулю;
- далее сдвиг увеличивается на единицу, и проверки повторяются;
- проверки повторяются для всех потенциальных сдвигов (рисунок 4.6).

Обращаем внимание, что строка 4 предполагает еще один цикл.

В худшем случае время работы процедуры ПОИСК-ПОДСТРОК-ПРОСТЕЙШИЙ можно оценить как $\Theta((n - m + 1)m)$. Рассмотрим пример, пусть текст $T = a^n$, а образец $P = a^m$. Тогда для всех потенциальных сдвигов $(n - m + 1)$ будет выполнено m сравнений символов в строке 4 процедуры, всего $(n - m + 1)m$, что есть $\Theta(n^2)$ (при $m = \lfloor n/2 \rfloor$).

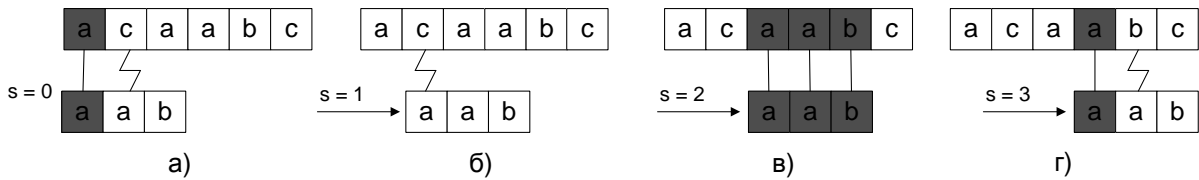


Рисунок 4.6 – Пример работы процедуры для подстроки $P = aab$ и строки $T = acaabc$.

Четыре последовательных сдвига изображены на рисунке 4.6. Совпавшие символы соединены прямыми линиями и закрашены серым. Несовпавшие символы соединены ломаными линиями. Допустимый сдвиг равен двум (рисунок 4.6 а–г).

Простейший алгоритм не является оптимальным. Неэффективность процедуры простейшего поиска связана с тем, что информация, получаемая при сравнении символов, не используется при проверке последующих сдвигов. Тем не менее такая информация может быть полезна. Например, подстрока равна $P = ababaca$, и проверки подтвердили, что сдвиг $s = 0$ допустим. Тогда недопустимыми сдвигами будут 1, 2, 3, 4, 5, а 6 сдвиг отбросить заранее нельзя.

4.3.2 Алгоритм Рабина-Карпа

Алгоритм Рабина и Карпа, заключающийся, как правило, в быстром поиске подстрок, в самом плохом случае работает за время $\Theta((n - m + 1)m)$.

Для простоты ограничим алфавит следующим образом: $\Sigma = \{0,1,2, \dots, 9\}$ (в общем случае любой символ алфавита Σ будет d -ичной цифрой, а d можно вычислить как $d = |\Sigma|$). В нашем случае последовательность из k символов будем рассматривать в виде десятичной записи числа, символы же будут преобразованы в цифры.

Через p будем записывать число, десятичной записью которого является образец $P[1..m]$. Точно так же для любого сдвига $s = 0,1, \dots, n - m$ через t_s определим десятичную запись участка текста $T[s + 1..s + m]$. Нетрудно увидеть, что если $t_s = p$, то сдвиг s будет допустимым. Предположим, что p можно рассчитать за время $O(m)$, а все t_s – за время $O(n)$, тогда, сравнивая p с каждой

десятичной записью t_s , мы сможем определить все допустимые сдвиги p за время $O(n)$.

По схеме Горнера мы можем произвести расчет десятичной записи образца p за время $O(m)$: $p = P[m] + 10(P[m-1] + 10(P[m-2] + \dots + 10(P[2] + 10P[1]) \dots))$. Точно так же можно произвести расчет t_0 .

При известном t_0 можно рассчитать t_1, t_2, \dots, t_{n-m} за время $O(n-m)$ следующим образом: $t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$. Для получения строки $T[s+2..s+m+1]$ из $T[s+1..s+m]$ необходимо убрать символ $T[s+1]$ (вычесть $10^{m-1}T[s+1]$) и добавить справа символ $T[s+m+1]$ (умножить полученную разность на 10 и прибавить к ней $T[s+m+1]$).

Если рассчитать выражение 10^{m-1} предварительно, то десятичные записи p и t_0, t_1, \dots, t_{n-m} можно определить за время $O(n+m)$, и, следовательно, допустимые сдвиги могут быть найдены за это же время $O(n+m)$.

Если десятичные числа p и t_s велики, то вычислить их за время $O(n+m)$ не представляется возможным. Обойти это ограничение можно, если все расчеты в схеме Горнера проводить по модулю фиксированного числа q . Чаще всего в качестве q подбирают такое простое число, для которого выражение $10q$ не превышает размера машинного слова. Для алфавита $\{0,1,2,\dots,d\}$ q определяется из выражения dq , которое также не должно превышать размера машинного слова. Схема Горнера будет выглядеть так:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q, \quad (4.1)$$

где $h \equiv d^{m-1} \pmod{q}$.

Единственным недостатком таких вычислений является то, что из равенства $t_s \equiv p \pmod{q}$ еще не следует равенство $t_s = p$. Сдвиг s будет заведомо ложным, если t_s и p не сравнимы по модулю. Иначе требуется дополнительная проверка образца $P[1..m]$ и текста $T[s+1..s+m]$. Если все символы совпали, то найден допустимый сдвиг, в противном случае произошло холостое срабатывание. При увеличении q количество холостых срабатываний будет уменьшаться.

Приведем псевдокод процедуры ПОИСК-ПОДСТРОК-РАБИН-КАРП. Входными параметрами будут: текст T , образец P , «основание системы счисления» d и простое число q .

ПОИСК-ПОДСТРОК-РАБИН-КАРП (T, P, d, q)

```

1  n ← length[T]
2  m ← length[P]
3  h ←  $d^{m-1} \bmod q$ 
4  p ← 0
5   $t_0 \leftarrow 0$ 
6  for i ← 1 to m
7    do p ← (d*p+P[i]) mod q
8     $t_0 \leftarrow (d*t_0 + T[i]) \bmod q$ 
9  for s ← 0 to n-m
10 do if p =  $t_s$ 
11   then if P[1..m] = T[s+1..s+m]
12     then print «Образец входит со сдвигом» s
13   if s < n-m
14     then  $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m-1]) \bmod q$ 
    
```

Рассмотрим детально работу процедуры. Все символы образца и текста преобразуются в d -ичные цифры. Строки с 1 по 5 отвечают за начальную инициализацию переменных.

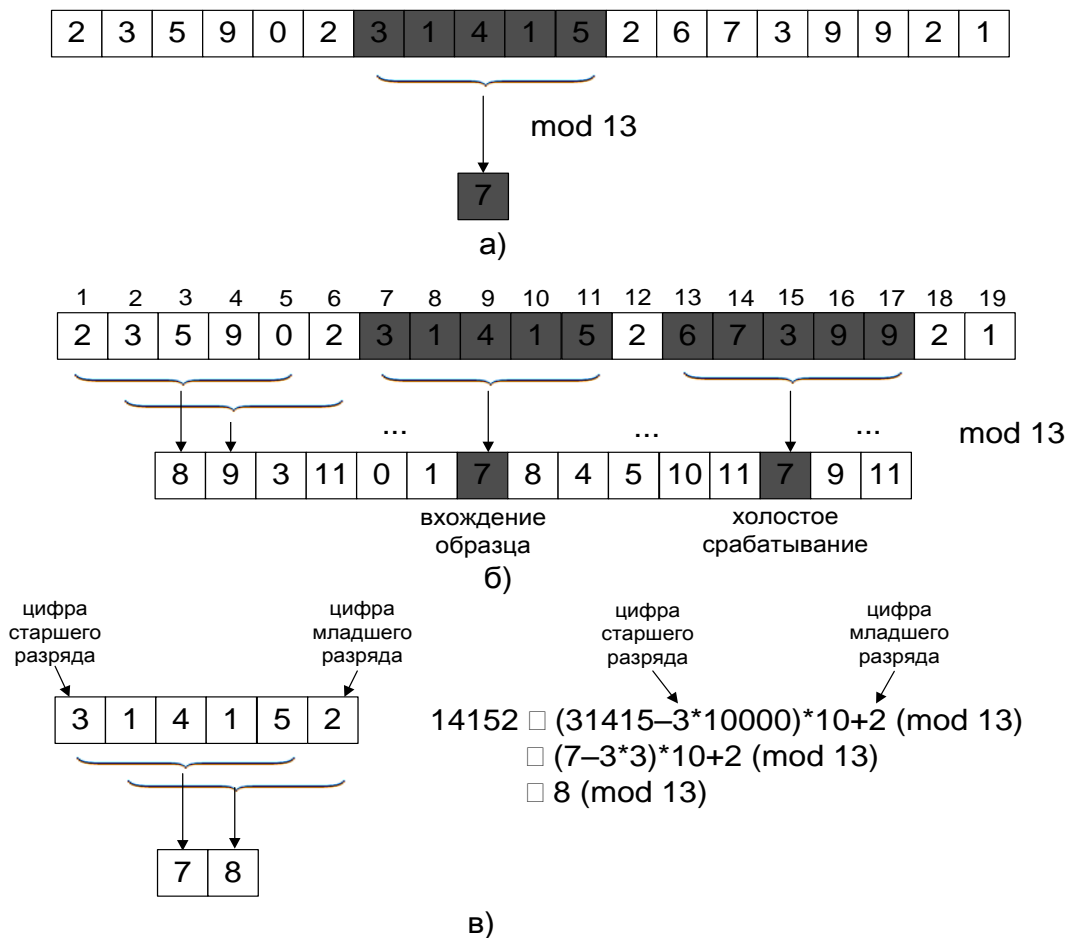


Рисунок 4.7 – Пример работы алгоритма Рабина-Карпа для алфавита $\Sigma = \{0,1,2, \dots,9\}$ и простого числа $q = 13$

На рисунке 4.7 а – строка T (текст). Серым закрашен отрезок текста T (образец) длины 5, десятичная запись которого равна 7 по модулю. На рисунке 4.7 б указаны десятичные записи отрезков длины 5 для всех сдвигов. Закрашенному образцу соответствуют две десятичные записи по модулю: первая – вхождение, вторая – холостое срабатывание. На рисунке 4.7 в – изменение десятичной записи по модулю при определении нового сдвига.

Цикл *for* в строках с 6 по 8 отвечает за первоначальный расчет значений p и t_0 по схеме Горнера. Основной цикл процедуры в строках 9–14 проверяет все сдвиги s ; условие в строке 10 определяет, сравнимы ли по модулю $t_s \equiv p$, если да, то происходит сравнение всех символов для строк $T[s + 1..s + m]$ и $P[1..m]$. Если все символы совпали, то сообщаем о найденном вхождении (строки 11–12). Для проверки следующего сдвига в строках 13–14 выполняется расчет нового значения t_s по формуле (4.1).

При использовании алгоритма на практике предполагается, что допустимых сдвигов будет немного, поэтому время работы можно оценить как $O(n + m)$ плюс время на анализ холостых срабатываний.

4.3.3 Алгоритм Кнута-Морриса-Пратта

Рассмотрим алгоритм поиска подстрок, время работы которого оценивается как $\Theta(n + m)$. Линейное время работы алгоритма обеспечивается за счет создания префикс-функции $\pi[1..m]$ и ее предварительного вычисления за время $O(m)$. Знание префикс-функции позволит проверить все потенциальные сдвиги s за время $O(1)$.

Префикс-функция строится по образцу P и содержит информацию о различных префиксах этой строки, взятой с длиной от 1 до m . Наличие таких данных позволяет не обрабатывать заведомо недопустимые сдвиги. Рассмотрим пример, в котором выполним поиск подстроки $P = ababaca$ в тексте T . Для некоторого произвольного сдвига s произошло совпадение первых пяти символов q , следующий символ не совпал с образом (рисунок 4.8 а), следовательно, нам известен участок текста $T[s + 1..s + q]$. Анализ этой информации позволит определить недопустимые сдвиги. Из рисунка 4.8 видно, что при сдвиге $s + 1$ первый символ подстроки

(буква *a*) совпадет со вторым символом текста (буква *b*). Таким образом, данный сдвиг будет недопустимым.

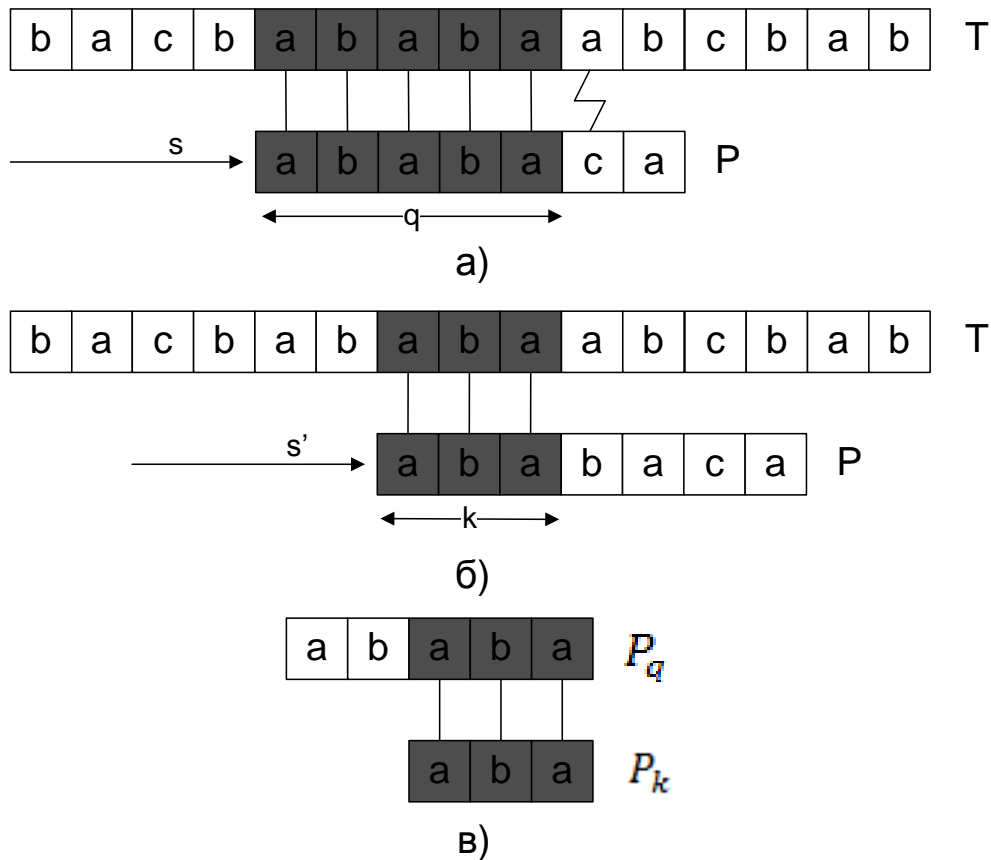


Рисунок 4.8 – Применение префикс-функции

На рисунке 4.8 а – состояние поиска, при котором совпало 5 символов текста и образца. На рисунке 4.8 б – новый сдвиг s' , который нельзя отбросить с учетом текущего знания текста. На рисунке 4.8 в – пример вычисления префикса для строки P_q .

А вот следующий сдвиг $(s + 2)$ заранее отбросить нельзя, потому что последние известные символы строки будут совпадать с первыми тремя символами подстроки. Обобщая, мы приходим к тому, что хотим знать ответ на следующий вопрос. Если $P[1..q] = T[s + 1..s + q]$, то каким будет минимальное значение нового сдвига s' при условии $s' > s$, для которого:

$$P[1..k] = T[s' + 1..s + k], \quad (4.2)$$

где $s' + k = s + q$? Принимая во внимание равенство первых q символов образца и текста, сдвиг s' будет наименьшим, который мы не можем отбросить без проведения анализа. Максимальный эффект будет достигаться, если $s' = s + q$. Тогда все сдвиги $s + 1, s +$

2, ... $s + q - 1$ можно отбросить. По крайней мере, мы можем не проверять первые k символов для нового сдвига s' (они заведомо совпадают с образцом).

Для определения сдвига s' достаточно знать подстроку P и число q . Строка $T[s' + 1..s + k]$ – это не что иное, как суффикс строки P_q . Поэтому число k , задействованное в формуле (4.2), – наибольшее число при условии, что P_k будет суффиксом P_q . На практике именно это число сохраняется в префикс-функции. Таким образом, значение сдвига s' можно определить как $s' = s + q - k$.

Сформулируем правило определения префикс-функции π для образца $P[1..m]$:

$$\pi[q] = \max\{k: k < q \text{ и } P_k]P_q\}.$$

Другими словами, $\pi[q]$ будет хранить длину максимального префикса P , который будет собственным суффиксом P_q . На рисунке 4.9 изображена префикс-функция для подстроки *ababaca*.

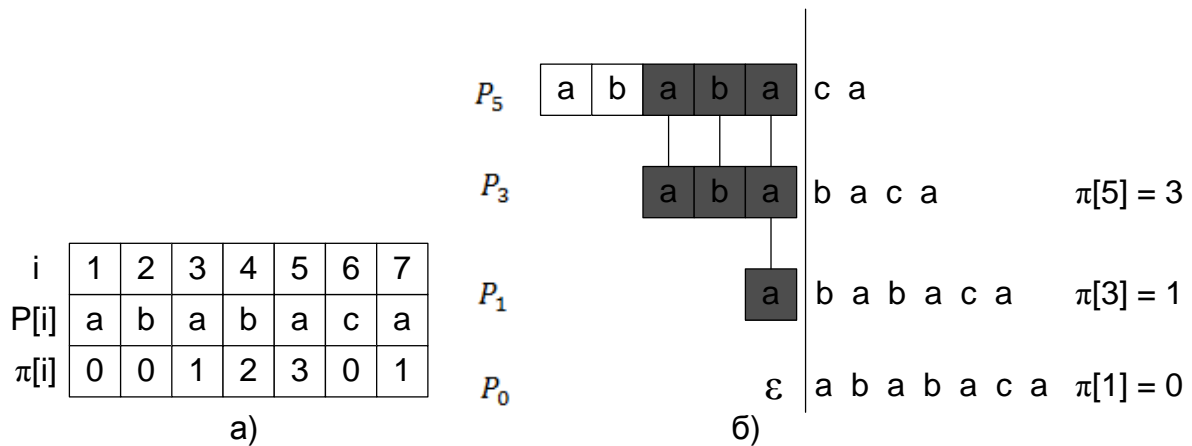


Рисунок 4.9 – Вычисление префикс-функции для образца *ababaca*

Запишем псевдокод для алгоритма Кнута-Морриса-Пратта в виде процедуры ПОИСК-ПОДСТРОК-КМП:

ПОИСК-ПОДСТРОК-КМП (T, P)

```
1  n ← length[T]
2  m ← length[P]
3  π ← ПРЕФИКС-ФУНКЦИЯ (P)
4  q ← 0
5  for i ← 1 to n
6    do while q > 0 and P[q+1] ≠ T[i]
7      do q ← π[q]
8      if P[q+1] = T[i]
9        then q ← q+1
10     if q = m
11       then print «Образец входит со сдвигом» i-m
12     q ← π[q]
```

Процедура ПРЕФИКС-ФУНКЦИЯ вызывается из ПОИСК-ПОДСТРОК-КМП и выполняет расчет префиксов для образца. Псевдокод будет выглядеть следующим образом:

ПРЕФИКС-ФУНКЦИЯ (P)

```
1  m ← length[P]
2  π[1] ← 0
3  k ← 0
4  for q ← 2 to m
5    do while k > 0 and P[k+1] ≠ P[q]
6      do k ← π[k]
7      if P[k+1] = P[q]
8        then k ← k+1
9      π[q] ← k
10 return π
```

4.4 Базовые алгоритмы на графах

4.4.1 Поиск в ширину

Рассмотрим один из базовых алгоритмов – поиск в ширину, который является основой, например, для алгоритмов Прима и Дейкстры. Название алгоритма связано с техникой его работы: сначала анализируются все соседи и только потом соседи соседей и т. д.

Для работы алгоритма на графе $G = (V, E)$ определяется начальная вершина s . В процессе работы будут найдены все

вершины, достижимые из s , в порядке увеличения расстояния от начальной вершины. Расстоянием будет не что иное, как длина кратчайшего пути из s . В результате формируется дерево поиска в ширину с корнем s . В дерево попадают только те вершины, которые достижимы из начальной вершины. Полученные пути для всех узлов дерева будут одними из кратчайших. Граф, передаваемый поиску в ширину, может быть как ориентированным, так и неориентированным.

В процессе работы вершины маркируются одним из трех цветов: белый, серый и черный. Сначала все вершины белые, в процессе обнаружения вершины становятся серыми, а когда анализ узла завершен, то вершина становится черной.

Детально процедура поиска в ширину выглядит так:

- перед началом поиска формируемое дерево состоит только из одной вершины s ;
- при обнаружении новой белой вершины v , смежной с вершиной u , ребро (u, v) и вершина v добавляются к дереву поиска;
- вершина v становится ребенком u , а u будет предком v ;
- для вершины u просматриваются все смежные вершины, после этого ее обработка завершается.

Приведем псевдокод процедуры поиска в ширину (ПВШ):

```

ПВШ ( $G, s$ )
1  for (для) каждой вершины  $u \in V[G] - \{s\}$ 
2    do  $color[u] \leftarrow$  БЕЛЫЙ
3     $d[u] \leftarrow \square$ 
4     $\pi[u] \leftarrow$  NIL
5   $color[s] \leftarrow$  СЕРЫЙ
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow$  NIL
8   $Q \leftarrow \{s\}$ 
9  while  $Q \neq \square$ 
10     do  $u \leftarrow head[Q]$ 
11     for (для) всех  $v \in Adj[u]$ 
12       do if  $color[v] =$  БЕЛЫЙ
13         then  $color[v] \leftarrow$  СЕРЫЙ
14            $d[v] \leftarrow d[u] + 1$ 
15            $\pi[v] \leftarrow u$ 
16           ОЧЕРЕДЬ-ДОБАВИТЬ ( $Q, v$ )
17     ОЧЕРЕДЬ-УДАЛИТЬ ( $Q$ )
18      $color[u] \leftarrow$  ЧЕРНЫЙ

```

На рисунке 4.10 показан пример работы процедуры ПВШ. Для каждой вершины u организованы дополнительные поля, сохраняющие цвет $color[u]$, информацию о предшественнике $\pi[u]$ и расстояние $d[u]$ до начальной вершины s . Для хранения серых вершин задействована очередь (раздел 3.2.2). В строках 1–4 выполняется начальная инициализация всех вершин за исключением s (цвет – белый, расстояние – бесконечность, предок – NIL). В строках 5–7 выполнена инициализация вершины s (цвет – серый, расстояние – ноль, предок – NIL). Далее начальная вершина помещается в очередь (строка 8). В основном цикле процедуры (строки 9–18) выполняется формирование дерева поиска в глубину. Пока очередь не пуста, мы извлекаем вершину u из головы очереди (строка 10) и начинаем просматривать все смежные с ней вершины v (строка 11). Если смежная вершина v белая, то делаем ее серой, расстояние увеличиваем на единицу, в качестве предка отмечаем вершину u (строки 13–15). Затем помещаем вершину v в очередь (строка 16). После того как просмотр всех смежных вершин для u завершен, узел u удаляется из очереди (строка 17) и перекрашивается в черный цвет в строке 18 (обработка вершины завершена). Процедура ПВШ использует представление графа $G = (V, E)$ в виде списка смежных вершин.

Время работы процедуры будет равно $O(V + E)$.

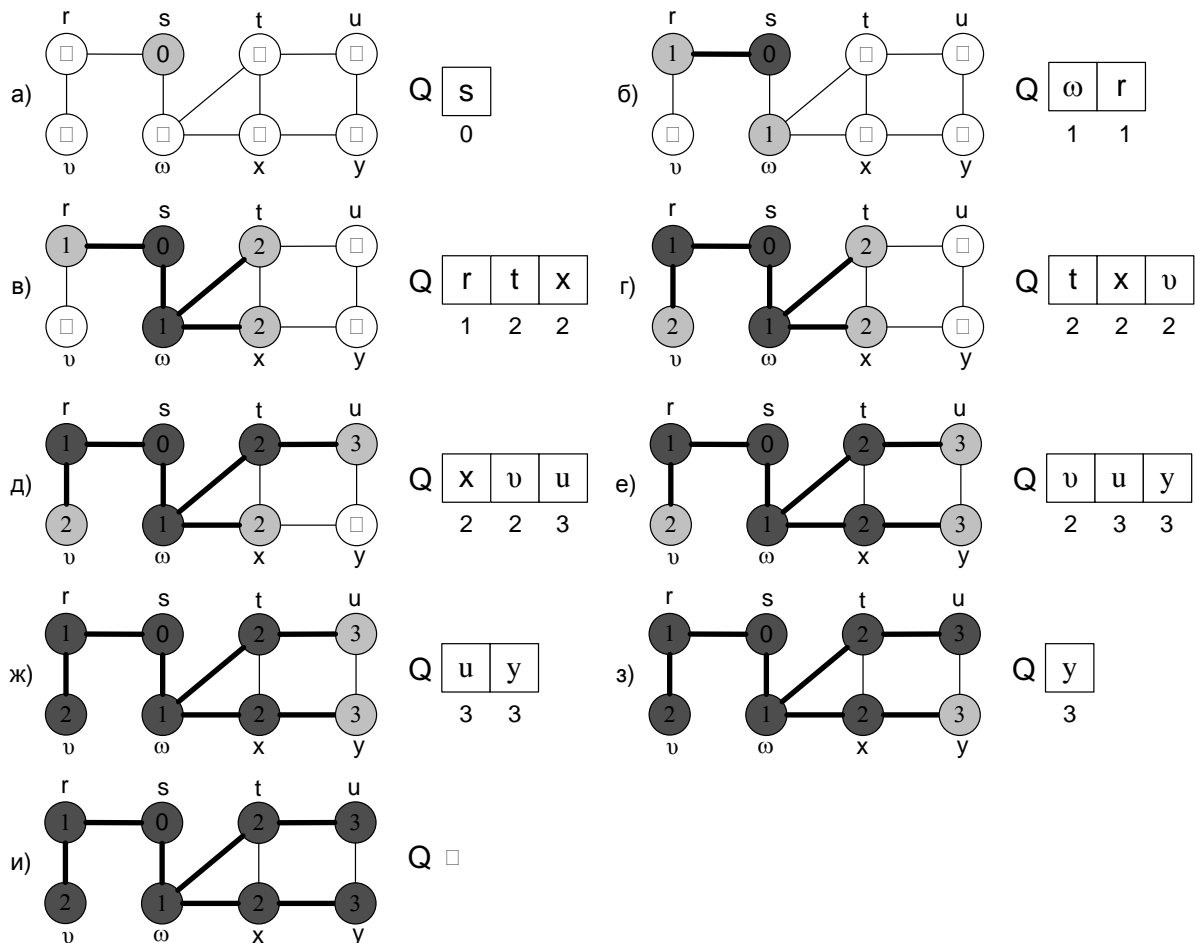


Рисунок 4.10 – Пример работы процедуры ПВШ для неориентированного графа. Показано состояние очереди перед каждой итерацией цикла *while*

4.4.2 Поиск в глубину

Идея поиска в глубину заключается в следующем:

- пока есть ребро, указывающее на еще необнаруженную вершину, то перемещаться по этому ребру. Повторять это действие, пока возможно;
- если больше нет ребер, рассмотренных выше, то возвращаться по последнему ребру назад и искать другие ребра, указывающие на непросмотренные вершины;
- рассмотренное выше повторяется до тех пор, пока не будут найдены все вершины из исходной;
- если после выполнения трех действий останутся необнаруженные вершины, то выбирается одна из таких вершин и первые три операции повторяются для этой вершины;

– процедура завершает свою работу, когда все вершины будут обработаны.

Если смежная с u вершина v впервые найдена, то так же, как и для поиска в ширину, в поле $\pi[v]$ фиксируется вершина предок u . В результате формируется дерево или несколько деревьев. Таким образом, мы приходим к понятию подграфа предшествования $G_\pi = (V, E_\pi)$, где $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq NIL\}$. Подграф предшествования – лес поиска в глубину, сформированный из деревьев поиска в глубину.

Алгоритм поиска в глубину применяет ту же цветовую схему для вершин, как и поиск в ширину: белая вершина – еще не обнаружена, серая вершина – находится в обработке и черная – обработка завершена. Каждая вершина будет принадлежать только одному дереву поиска, поэтому формируемые деревья не будут пересекаться.

Дополнительно алгоритм поиска в глубину использует два поля для хранения времени: $d[v]$ – время начала обработки и $f[v]$ – время завершения обработки. Эта служебная информация используется другими алгоритмами на графах и применяется при рассмотрении свойств поиска в глубину.

Для меток времени $d[v]$ и $f[v]$, которые являются целыми числами, будет выполняться следующее неравенство: $d[v] < f[v]$. До начала времени $d[v]$ – вершина белая, после $f[v]$ – вершина черная, между метками $d[v]$ и $f[v]$ – серая. Граф, поступающий на обработку, может быть как ориентированным, так и неориентированным.

Проанализируем псевдокод процедуры поиска в глубину (ПВГ):

ПВГ (G)

```
1 for (для) всех вершин  $u \in V[G]$ 
2   do color[u] ← БЕЛЫЙ
3      $\pi[u] \leftarrow NIL$ 
4 time ← 0
5 for (для) всех вершин  $u \in V[G]$ 
6   do if color[u] = БЕЛЫЙ
7     then ПВГ-ПРОСМОТР ( $u$ )
```

ПВГ-ПРОСМОТР (u)

```
1 color[u] ← СЕРЫЙ    ▷ Вершина  $u$  была белой
2 d[u] ← time ← time+1
3 for (для) всех  $v \in Adj[u]$  ▷ Обработать ребро ( $u, v$ )
4   do if color[v] = БЕЛЫЙ
5     then  $\pi[v] \leftarrow u$ 
6         ПВГ-ПРОСМОТР ( $v$ )
7 color[u] ← ЧЕРНЫЙ    ▷ Вершина обработана, делаем ее черной
8 f[u] ← time ← time+1
```

В строках 1–3 выполняется начальная инициализация вершин графа (цвет – белый, предок – NIL). Метка времени $time$ устанавливается в ноль (строка 4). В цикле *for* перебираем все вершины u и если вершина белая, то вызываем процедуру ПВГ-ПРОСМОТР для нее (строки 5–7).

Во время работы процедуры ПВГ-ПРОСМОТР цвет вершины u становится серым, метка времени $d[u]$ сохраняет значение $time$, увеличенное на единицу (строки 1–2). В строках 3–6 в цикле *for* перебираются все смежные u с вершины v . Если вершина v белая, то у нее фиксируется предок (строка 5) и запускается процедура ПВГ-ПРОСМОТР (строка 6). После завершения цикла *for* вершина u становится черной и в метке $f[u]$ запоминается значение $time$, увеличенное на единицу.

Время работы процедуры будет равно $\theta(V + E)$.

На рисунке 4.11 приведен пример работы процедуры ПВГ.

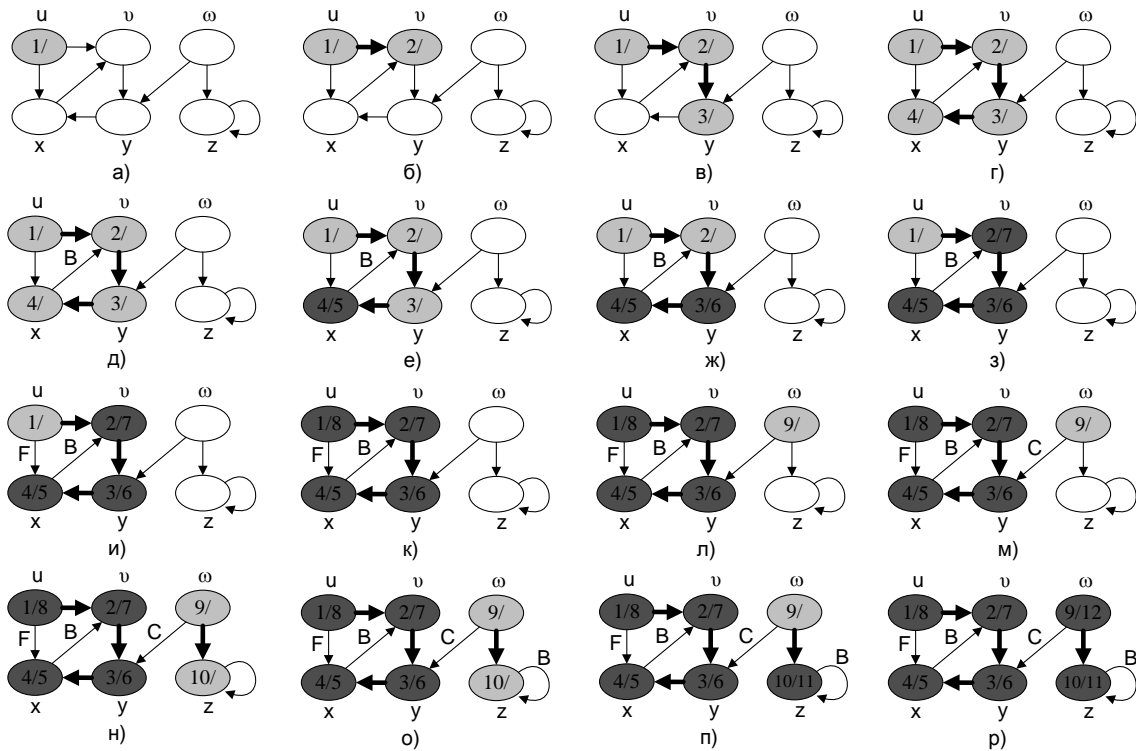


Рисунок 4.11 – Пример работы процедуры ПВГ для ориентированного графа. Выполнена классификация ребер (ребра дерева – серые, В – обратные, С – перекрестные, F – прямые)

4.5 Двоичные деревья поиска

Деревья поиска могут быть использованы для реализации динамических множеств с использованием следующих процедур: ПОИСК, МИНИМУМ, МАКСИМУМ, ПРЕДЫДУЩИЙ, СЛЕДУЮЩИЙ, ДОБАВЛЕНИЕ и УДАЛЕНИЕ.

Время работы основных процедур зависит от высоты дерева. Если все его уровни максимально заполнены вершинами, то его высота и время работы процедур пропорциональны логарифму количества узлов дерева. Напротив, если дерево организовано в виде последовательной цепочки из n узлов, это время увеличивается до $\Theta(n)$. Предположим, что высота случайного дерева поиска равна $O(\lg n)$, тогда время работы основных процедур будет $\Theta(\lg n)$.

Двоичные деревья поиска, возникающие при решении практических задач, редко когда будут соответствовать случайным. Этот недостаток можно преодолеть за счет дополнительной меры – балансировка, гарантируя тем самым, что высота дерева из n узлов будет $O(\lg n)$.

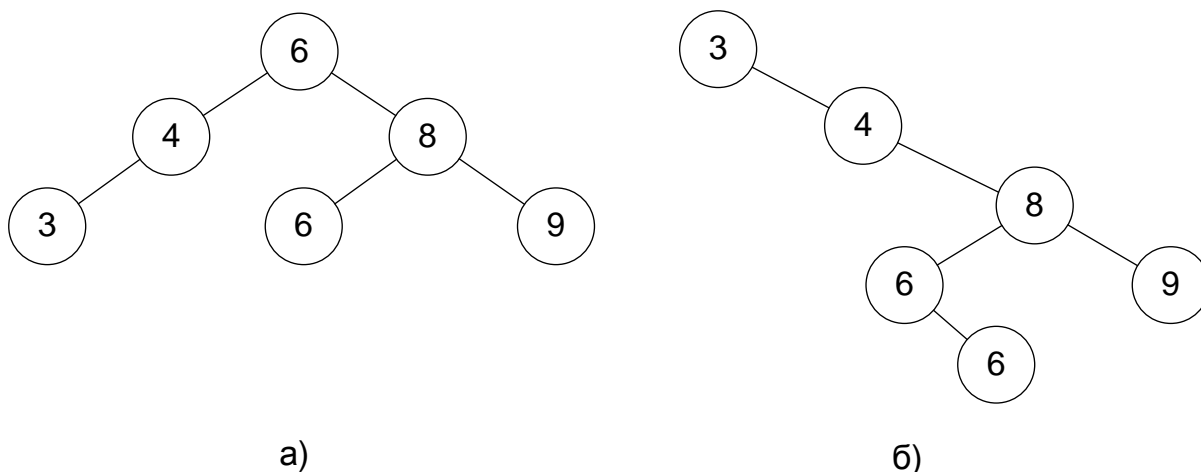


Рисунок 4.12 – Варианты представления двоичных деревьев поиска

На рисунке 4.12 а – пример сбалансированного дерева высоты 2 с 6 узлами. На рисунке 4.12 б – дерево, состоящее из тех же узлов, для высоты 4 (менее эффективная структура).

У каждой вершины в двоичном дереве поиска есть родитель (за исключением корня), левый и правый ребенок (рисунок 4.12). Каждая вершина дерева может содержать следующую информацию:

- ключ (*key*);
- указатель на левого ребенка (*left*);
- указатель на правого ребенка (*right*);
- указатель на родителя (*p*);
- дополнительные данные.

Если ребенок или родитель отсутствуют, то в соответствующее поле помещаем значение *NIL*.

Хранение ключей в двоичном дереве организовано упорядоченно: для произвольной вершины x , если выполняется условие $key[y] < key[x]$, вершина y помещается в левое поддерево узла x . Если выполняется условие $key[y] \geq key[x]$, y помещается в правое поддерево.

На рисунке 4.12 показаны примеры двоичных деревьев поиска, вершины которых расположены с учетом свойства упорядоченности.

Структура двоичного дерева поиска позволяет вывести все ключи в неубывающем порядке. Процедура ДЕРЕВО-ОБХОД выводит все ключи левого поддерева, затем печатается ключ корня дерева и наконец – ключи правого поддерева.

Запишем процедуру ДЕРЕВО-ОБХОД($root[T]$), которая будет выводить ключи, входящие в дерево T , в указанном выше порядке.

```
ДЕРЕВО-ОБХОД (x)
1  if x ≠ NIL
2  then ДЕРЕВО-ОБХОД (left[x])
3     напечатать key[x]
4     ДЕРЕВО-ОБХОД (right[x])
```

Например, для деревьев, показанных на рисунке 4.12, ключи будут выведены в следующем порядке: 3, 4, 6, 6, 8, 9. Правильность процедуры гарантируется за счет свойства упорядоченности. Время работы процедуры можно оценить как $\Theta(n)$, где n – количество узлов дерева: обработка каждого узла занимает фиксированное время (помимо рекурсивных вызовов) и происходит единожды.

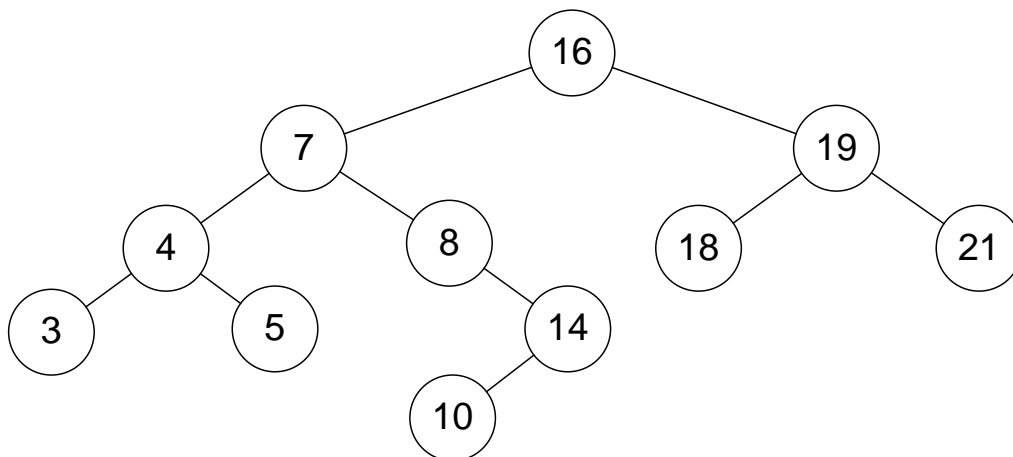


Рисунок 4.13 – Демонстрация двоичного дерева для типовых процедур

На примере рисунка 4.13 покажем работу следующих процедур: ДЕРЕВО-ПОИСК: ищем ключ 14, для этого двигаемся от корня по следующей цепочке $16 \rightarrow 7 \rightarrow 8 \rightarrow 14$; ДЕРЕВО-МИНИМУМ: для определения узла с минимальным ключом 3 будем двигаться всегда влево; ДЕРЕВО-МАКСИМУМ: для определения максимального узла с ключом 21 – направо; ДЕРЕВО-СЛЕДУЮЩИЙ: для узла с ключом 16 следующим узлом будет 18 (это минимальный элемент в правом поддереве узла 16); ДЕРЕВО-ПРЕДЫДУЩИЙ: для узла с ключом 16 предыдущим узлом будет 14.

Поиск. Приведем псевдокод процедуры ДЕРЕВО-ПОИСК. Параметрами процедуры будут ключ k , для которого выполняется поиск, и указатель x на точку входа в поддерево, в котором предполагается осуществить поиск. Если ключ k найден, то возвращается указатель на узел с этим ключом, иначе возвращается NIL .

```
ДЕРЕВО-ПОИСК (x, k)
1  if x = NIL or k = key[x]
2    then return x
3  if k < key[x]
4    then return ДЕРЕВО-ПОИСК (left[x], k)
5  else return ДЕРЕВО-ПОИСК (right[x], k)
```

В первой строке, если вершина не существует или искомый ключ соответствует ключу в текущем узле, то поиск завершается и возвращается указатель на эту вершину. В третьей строке проверяется следующее: если искомый ключ меньше ключа в текущем узле $k < key[x]$, то поиск продолжается в левом поддереве текущего узла, если ключ больше ключа в текущем узле $k > key[x]$, то поиск продолжается в правом поддереве. Время поиска можно оценить как $O(h)$ (где h – высота дерева), потому что длина пройденного пути будет не больше h .

В качестве альтернативы рекурсивного алгоритма можно привести итеративную версию, которая, в большинстве случаев, более эффективна:

```
ДЕРЕВО-ПОИСК-ИТЕРАЦИОННЫЙ (x, k)
1  while x ≠ NIL and k ≠ key[x]
2    do if k < key[x]
3      then x ← left[x]
4      else x ← right[x]
5  return x
```

Минимум и максимум. Приведем псевдокод процедуры ДЕРЕВО-МИНИМУМ. Если мы будем все время двигаться по указателям $left$, пока значение $left$ не станет равно NIL , то найдем минимальный ключ в дереве поиска (рисунок 4.13). Алгоритм на выходе возвращает указатель на узел с минимальным ключом.

ДЕРЕВО-МИНИМУМ (x)

```
1 while left[x]  $\neq$  NIL
2     do  $x \leftarrow$  left[x]
3 return  $x$ 
```

Правильность алгоритма ДЕРЕВО-МИНИМУМ гарантируется за счет свойства упорядоченности. Если у узла x отсутствует левый ребенок, то минимальным элементом в поддереве с корнем x будет сам x (любой ключ правого поддерева больше x), иначе минимальный элемент следует искать в левом поддереве с корнем x .

Процедура ДЕРЕВО-МАКСИМУМ симметрична:

ДЕРЕВО-МАКСИМУМ (x)

```
1 while right[x]  $\neq$  NIL
2     do  $x \leftarrow$  right[x]
3 return  $x$ 
```

Время работы обоих алгоритмов можно оценить как $O(h)$ (движение по дереву выполняется только вниз).

Следующий и предыдущий элементы. Рассмотрим псевдокод процедуры ДЕРЕВО-СЛЕДУЮЩИЙ, которая определяет следующий элемент по отношению к текущему. Алгоритм возвращает указатель на следующий элемент за x или значение NIL , если x – последний узел.

ДЕРЕВО-СЛЕДУЮЩИЙ (x)

```
1 if right[x]  $\neq$  NIL
2     then return ДЕРЕВО-МИНИМУМ (right[x])
3  $y \leftarrow$  p[x]
4 while  $y \neq$  NIL and  $x =$  right[y]
5     do  $x \leftarrow$   $y$ 
6      $y \leftarrow$  p[y]
7 return  $y$ 
```

В первой строке проверяется существование правого поддерева для узла x , если оно не пусто, то следующим элементом будет минимальный элемент в этом поддереве (строка 2). Пример этого случая приведен на рисунке 4.13. Следующим для ключа 16 будет узел 18. Строки 3–6 отвечают за поиск следующего элемента, когда правое поддерево x пустое. В третьей строке во вспомогательной переменной y фиксируем родителя x . В цикле *while* выполняем поиск такого родителя y , для которого x будет левым ребенком (строки 4–6).

Как только это условие выполнится, то вершина u будет следующим элементом и процедура возвращает указатель на этот узел (строка 7).

Время работы процедуры ДЕРЕВО-СЛЕДУЮЩИЙ можно оценить как $O(h)$ (обход дерева осуществляется или только вверх, или только вниз). Процедуру ДЕРЕВО-ПРЕДЫДУЩИЙ мы оставляем для самостоятельного рассмотрения, потому что она является симметричной для ДЕРЕВО-СЛЕДУЮЩИЙ.

Добавление элемента. Обратимся к псевдокоду процедуры ДЕРЕВО-ДОБАВЛЕНИЕ. Она выполняет вставку элемента в нужное место в дереве T с учетом свойства упорядоченности. Входом процедуры будет указатель z на новый узел. У вершины будут заполнены такие поля, как ключ $key[z]$, указатели на левого $left[z] = NIL$ и правого $right[z] = NIL$ ребенка. Во время выполнения процедуры новый узел z добавляется в нужное место в дереве T , при этом меняются некоторые поля узла z и его будущего родителя.

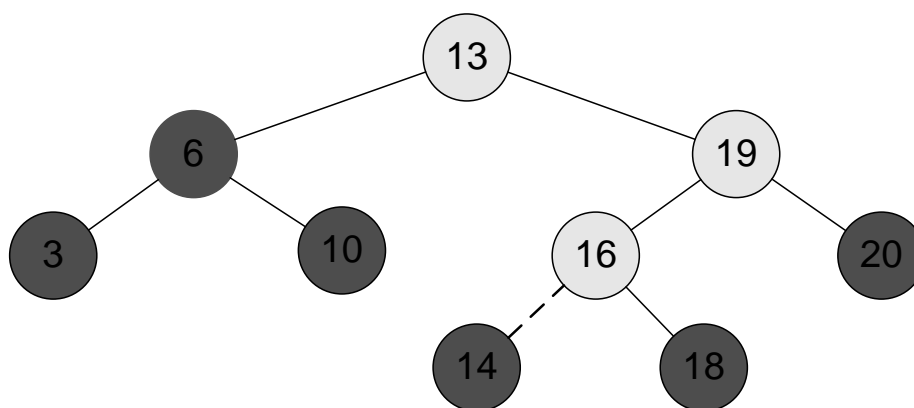


Рисунок 4.14 – Пример работы процедуры ДЕРЕВО-ДОБАВЛЕНИЕ для ключа 14. Светлые узлы показывают траекторию обхода дерева для определения места для нового узла. Пунктирной линией визуализирован процесс связывания нового узла с деревом

ДЕРЕВО-ДОБАВЛЕНИЕ (T, z)

```
1   $y \leftarrow NIL$ 
2   $x \leftarrow root[T]$ 
3  while  $x \neq NIL$ 
4      do  $y \leftarrow x$ 
5          if  $key[z] < key[x]$ 
6              then  $x \leftarrow left[x]$ 
7              else  $x \leftarrow right[x]$ 
8   $p[z] \leftarrow y$ 
9  if  $y = NIL$ 
10 then  $root[T] \leftarrow z$ 
11 else if  $key[z] < key[y]$ 
12     then  $left[y] \leftarrow z$ 
13     else  $right[y] \leftarrow z$ 
```

Выполнение процедуры ДЕРЕВО-ДОБАВЛЕНИЕ продемонстрировано на рисунке 4.14. В строках 1 и 2 выполняется начальная инициализация переменных y и x . В цикле *while* определяется вершина y , которая будет родителем для добавляемого узла z (строки 3–7). В строке 8 для вершины z записывается родитель. Если значение вершины y равно NIL , то z будет корнем дерева T (строка 10), иначе если ключ узла z меньше ключа y , то z становится левым ребенком y ; ключ узла z больше либо равен ключу y , то z будет правым ребенком y (строки 11–13).

Как и рассмотренные выше процедуры, время работы ДЕРЕВО-ДОБАВЛЕНИЕ можно оценить как $O(h)$.

Удаление элемента. Определение удаляемой вершины с помощью процедуры ДЕРЕВО-УДАЛЕНИЕ вычисляется несколько сложнее, чем добавление элемента. В процессе удаления рассматриваются три частных случая (рисунок 4.15):

– у вершины z нет детей, тогда правому или левому ребенку родителя z (в зависимости от того, каким ребенком является z) необходимо присвоить значение NIL ;

– у вершины z есть один ребенок, тогда родителя z связываем с ребенком z ;

– у вершины z – двое детей, тогда находим следующую за z вершину y , затем копируем данные из y в z , а вершину y удаляем.

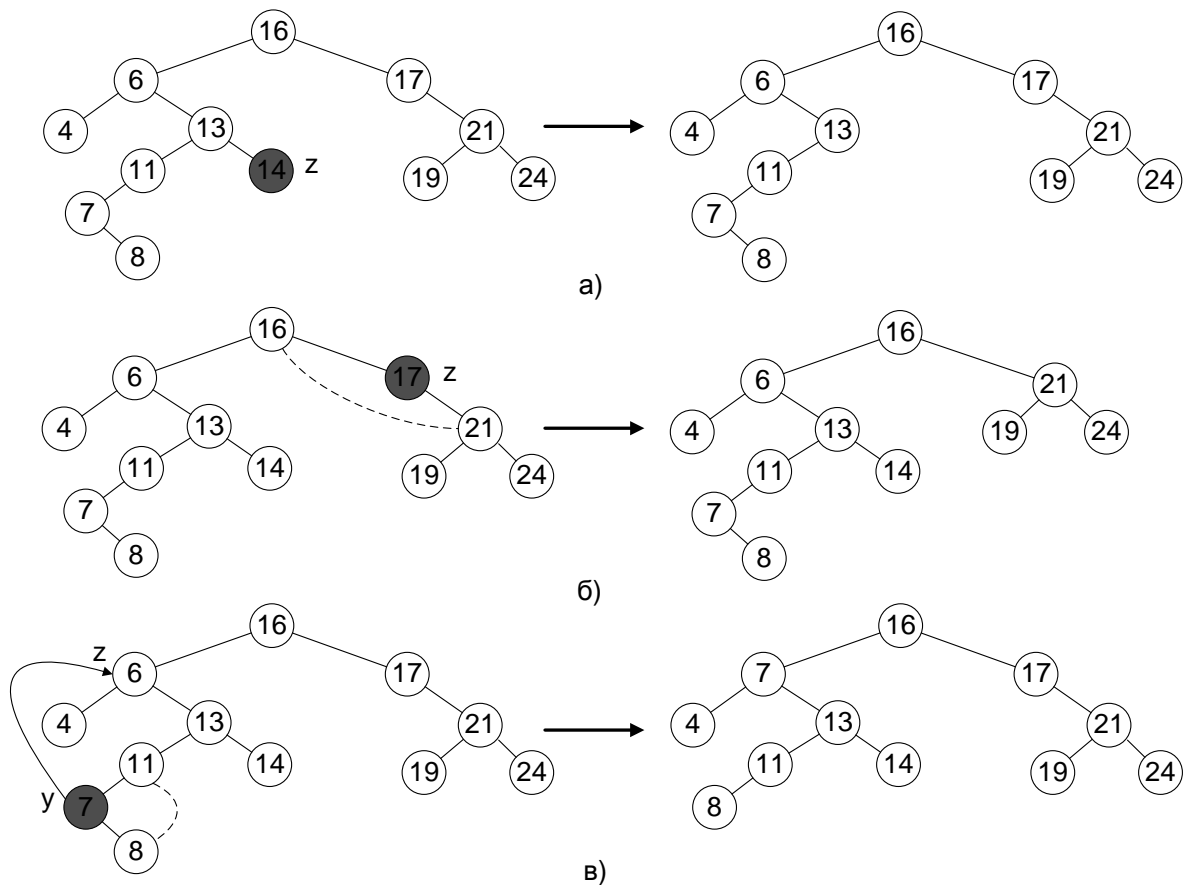


Рисунок 4.15 – Три случая удаления вершины z с помощью процедуры ДЕРЕВО-УДАЛЕНИЕ

Псевдокод процедуры ДЕРЕВО-УДАЛЕНИЕ описывает все эти случаи, но в несколько другой последовательности.

```

ДЕРЕВО-УДАЛЕНИЕ (T, z)
1  if left[z] = NIL or right[z] = NIL
2  then y ← z
3  else y ← ДЕРЕВО-СЛЕДУЮЩИЙ (z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y = left[p[y]]
12     then left[p[y]] ← x
13     else right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
16     ▷ копируем доп. данные, связанные с y
17 return y

```

Строки 1–3 отвечают за вычисление вершины u , которая будет удалена из дерева (или сама z , или следующая за z). Затем в переменной x сохраняем информацию о существующем ребенке u или присваивается значение NIL (строки 4–6). В строках 7–8 переписывается родитель вершины x на родителя вершины u . Если родитель u отсутствует, то узел x становится корнем дерева (строки 9–10). В строках с 11 по 13 родитель вершины u связывается с узлом x . Если вершина u не совпадает с вершиной z , то копируем данные из u в z (строки 14–16). В строке 17 возвращаем указатель на удаляемую вершину.

Время работы процедуры оценивается как $O(h)$.

5 ЛАБОРАТОРНЫЙ ПРАКТИКУМ

5.1 Лабораторная работа № 1

Требования к лабораторной работе:

- 1) создание графического пользовательского интерфейса;
- 2) реализация поиска чисел методом дихотомии (метод деления отрезка пополам);
- 3) сортировку последовательности организовать с помощью обобщенного алгоритма стандартной библиотеки шаблонов;
- 4) предусмотреть возможность загрузки файлов с исходной неотсортированной последовательностью в интерфейс программы;
- 5) модифицировать алгоритм дихотомии таким образом, чтобы получить возможность нахождения всех повторяющихся чисел.

Пример исходного кода для лабораторной работы:

```
using System;
using System.Data;
using System.IO;
using System.Linq;
using System.Windows.Forms;

namespace Lab_1
{
    public partial class Form1 : Form
    {
        double[] numb;
        int count;
        public Form1()
        {
```

```

        InitializeComponent();
    }

    private void TextBox2_TextChanged(object sender, EventArgs e)
    {
        button2.Enabled = true ? textBox2.Text.Length != 0 && listBox1.Items.Count > 0 :
button2.Enabled = false;
    }

    private void TextBox2_KeyPress(object sender, KeyPressEventArgs e)
    {
        char numbersSearch = e.KeyChar;
        if (!Char.IsDigit(numbersSearch) && numbersSearch != 8
            && numbersSearch != 44 && numbersSearch != 45) e.Handled = true;
    }

    private void listBoxSelect(int first, int endNumber)
    {
        listBox1.SelectionMode = SelectionMode.MultiSimple;
        for (int i = first + 1; i < endNumber; i++) listBox1.SetSelected(i, true);
    }

    private void SearchBinary(double key, double[] numb, int leftBoard, int rightBoard)
    {
        while (leftBoard <= rightBoard)
        {
            int middle = (rightBoard + leftBoard) / 2;
            if (key < numb[middle]) rightBoard = middle - 1;
            else if (key > numb[middle]) leftBoard = middle + 1;
            if (key == numb[middle])
            {
                count++;
                int l = 1;
                while (middle != 0 && key == numb[middle - l])
                {
                    count++;
                    l++;
                }
                int r = 1;
                while (middle != numb.Length - 1 && key == numb[middle + r])
                {
                    count++;
                    r++;
                }
                listBoxSelect(middle - l, middle + r);
                break;
            }
        }
    }

    private void Button2_Click(object sender, EventArgs e)
    { var key = Convert.ToDouble(textBox2.Text);

```



```

int rightBoard = numb.Length - 1;
int leftBoard = 0;
count = 0;
listBox1.ClearSelected();
SearchBinary(key, numb, leftBoard, rightBoard);
if (count != 0) label1.Text = "Всего найдено:" + Convert.ToString(count);
else label1.Text = "По запросу " + Convert.ToString(key) + " ничего не найдено.";
}

private void WriteTextInListBox(string textFile)
{ try
  {
    numb = textFile.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries).
      Select(double.Parse).ToArray();
  }
  catch (System.FormatException)
  {
    MessageBox.Show("В файле есть посторонние символы. Чтение файла не
доступно.");
    return;
  }
  Array.Sort(numb);
  for (int i = 0; i < numb.Length; i++)
  {
    listBox1.Items.Add(numb[i].ToString() + "\r\n");
  }
}

private void Button1_Click(object sender, EventArgs e)
{ if (openFileDialog1.ShowDialog() == DialogResult.OK)
  { string lineOfFile, str = " ";
    string textFile = "";
    listBox1.Items.Clear();
    textBox2.Clear();
    label1.Text = " ";
    var sr = new StreamReader(openFileDialog1.FileName);
    if (sr.Peek() > -1)
    {
      while ((lineOfFile = sr.ReadLine()) != null)
      {
        textFile += str + lineOfFile;
      }
      WriteTextInListBox(textFile);
    }
    else MessageBox.Show("Файл пуст!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Warning);
  }
}
}
}

```

5.2 Лабораторная работа № 2

Требования к лабораторной работе:

- 1) создание графического пользовательского интерфейса;
- 2) реализовать алгоритмы быстрой сортировки и сортировки подсчетом;
- 3) предусмотреть возможность загрузки файлов с исходной неотсортированной последовательностью в интерфейс программы;
- 4) подготовить файлы с различным количеством элементов: 100, 1000, 10000, 100000 и 1000000;
- 5) провести анализ алгоритмов по следующим критериям: время, количество перестановок и количество сравнений.

Пример исходного кода для лабораторной работы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Media;
using System.Threading;
using System.Threading.Tasks;
using System.IO;
using System.Windows.Forms;

namespace MLab2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            SetStatus("");
        }

        private BackgroundWorker loadWorker;
        private BackgroundWorker inputWorker;
        private BackgroundWorker quickSortingWorker;
        private BackgroundWorker countingSortWorker;
        private BackgroundWorker outputWorker;
        private Stream fileStream;
        private int totalLineCount = 0;
        int localLineCount = 0;

        private List<string> rawData = new List<string>();
```

```

private double[] data;

bool doubleDetected = false;
bool doubleRemoved = true;

// Quick Sort
double[] quickSortData;
int quickSortCompareCount = 0;
int quickSortSwapCount = 0;
DateTime qTime1;
DateTime qTime2;
TimeSpan quickSortTime;

// Counting Sort
int[] countingSortData;
int countingSortCompareCount = 0;
int countingSortSwapCount = 0;
DateTime cTime1;
DateTime cTime2;
TimeSpan countingSortTime;

private void selectFileButton_Click(object sender, EventArgs e)
{
    using (OpenFileDialog openFileDialog = new OpenFileDialog())
    {
        openFileDialog.InitialDirectory = "c:\\";

        if (openFileDialog.ShowDialog() == DialogResult.OK)
        {
            filePathTextBox.Text = openFileDialog.FileName;
            totalLineCount = File.ReadLines(openFileDialog.FileName).Count();
            data = new double[totalLineCount];
            fileStream = openFileDialog.OpenFile();

            loadWorker = CreateBackgorundWorker(LoadFileBackgroundWork,
LoadFileWorkerProgressChanged, LoadFileRunWorkerCompleted);

            inputBox.Items.Clear();
            loadWorker.RunWorkerAsync();
        }
    }
}

private void sortButton_Click(object sender, EventArgs e)
{
    quickSortingWorker = CreateBackgorundWorker(QuickSortBackgroundWork,
QuickSortWorkerProgressChanged, QuickSortRunWorkerCompleted);
    quickSortingWorker.RunWorkerAsync();
    if (doubleRemoved)
    {
        countingSortWorker = CreateBackgorundWorker(CountingSortBackgroundWork,
CountingSortWorkerProgressChanged, CountingSortRunWorkerCompleted);

```

```

        countingSortWorker.RunWorkerAsync();
    }
}

private BackgroundWorker CreateBackgorundWorker(DoWorkEventHandler
doWorkEventHandler,
    ProgressChangedEventHandler progressChangedEventHandler,
    RunWorkerCompletedEventHandler runWorkerCompletedEventHandler)
{
    BackgroundWorker _worker = new BackgroundWorker();

    _worker.WorkerSupportsCancellation = false;
    _worker.WorkerReportsProgress = true;
    _worker.DoWork += doWorkEventHandler;
    _worker.ProgressChanged += progressChangedEventHandler;
    _worker.RunWorkerCompleted += runWorkerCompletedEventHandler;

    return _worker;
}

private void LoadFileRunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs args)
{
    inputWorker = CreateBackgorundWorker(FillInInputBoxBackgroundWork,
FillInInputBoxWorkerProgressChanged, FillInInputBoxRunWorkerCompleted);
    inputWorker.RunWorkerAsync();
}

private void LoadFileWorkerProgressChanged(object sender,
ProgressChangedEventArgs args)
{
    SetProgressBarStatus((int)(((float)localLineCount / (float)totalLineCount) * 100));
}

private void LoadFileBackgroundWork(object sender, DoWorkEventArgs e)
{
    SetStatus("Чтение файла...");

    try
    {
        using (StreamReader sr = new StreamReader(fileStream))
        {
            localLineCount = 0;
            rawData.Clear();

            while (!sr.EndOfStream)
            {
                string s = sr.ReadLine();
                rawData.Add(s);
                localLineCount++;
                loadWorker.ReportProgress(100);
            }
        }
    }
}

```

```

        SetStatus("Завершение чтения файла...");
    }
}
catch (Exception ex)
{
    SetStatus("Ошибка чтения файла! " + ex.Message);
    SystemSounds.Exclamation.Play();
}
}

private void FillInInputBoxRunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs args)
{
    if (!disableOutput.Checked)
    {
        inputBox.BeginUpdate();
        inputBox.Items.AddRange(rawData.ToArray());
        inputBox.EndUpdate();
    }
    sortButton.Enabled = true;
    SetStatus("Файл загружен!");
}

private void FillInInputBoxWorkerProgressChanged(object sender,
ProgressChangedEventArgs args)
{
    SetProgressBarStatus(args.ProgressPercentage);
}

private void FillInInputBoxBackgroundWork(object sender, DoWorkEventArgs e)
{
    doubleDetected = false;

    for (int i = 0; i < rawData.Count; i++)
    {
        double d = Convert.ToDouble(rawData[i]);
        data[i] = d;
        if (!doubleDetected)
            doubleDetected = (Math.Abs(d % 1) <= (Double.Epsilon * 100)) ? false : true;
    }

    if (doubleDetected)
    {
        string message = "В файле обнаружены вещественные числа! " +
            "Сортировка подсчётом используется только для натуральных чисел. " +
            "Удалить все вещественные числа?";
        string caption = "Предупреждение";

        if (MessageBox.Show(message, caption, MessageBoxButtons.YesNo,
MessageBoxIcon.Exclamation) == DialogResult.Yes)
        {

```

```

List<double> dataCopy = new List<double>();

for (int i = 0; i < data.Length; i++)
{
    if (Math.Abs(data[i] % 1) <= Double.Epsilon * 100)
    {
        dataCopy.Add(data[i]);
    }
}

data = dataCopy.ToArray();
doubleRemoved = true;
}
else
{
    doubleRemoved = false;
}
}
else
{
    doubleRemoved = true;
}

for (int i = 0; i < data.Length; i++)
{
    inputWorker.ReportProgress((i / rawData.Count) * 100);
}
}

private void QuickSortRunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs args)
{
    qTime2 = DateTime.Now;
    quickSortTime = qTime2 - qTime1;
    quicksortTimeLabel.Text = "Время: " + string.Format("{0} мс",
quickSortTime.TotalMilliseconds);
    quicksortComparesLabel.Text = "Кол-во сравнений: " +
quickSortCompareCount.ToString();
    quicksortShiftsLabel.Text = "Кол-во перестановок: " +
quickSortSwapCount.ToString();
    if (!disableOutput.Checked)
    {
        quicksortBox.BeginUpdate();
        quicksortBox.Items.AddRange(Array.ConvertAll<double, string>(quickSortData, x
=> x.ToString()));
        quicksortBox.EndUpdate();
    }
}

private void QuickSortWorkerProgressChanged(object sender,
ProgressChangedEventArgs args)
{

```

```

}

private void QuickSortBackgroundWork(object sender, DoWorkEventArgs e)
{
    SetStatus("Сортировка...");
    quickSortData = new double[data.Length];
    quickSortCompareCount = 0;
    quickSortSwapCount = 0;
    Array.Copy(data, quickSortData, data.Length);
    qTime1 = DateTime.Now;
    QuickSort(quickSortData, 0, quickSortData.Length - 1);
}

private void CountingSortRunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs args)
{
    cTime2 = DateTime.Now;
    countingSortTime = cTime2 - cTime1;
    countingTimeLabel.Text = "Время: " + string.Format("{0} мс",
countingSortTime.TotalMilliseconds);
    countingComparesLabel.Text = "Кол-во сравнений: " +
countingSortCompareCount.ToString();
    countingShiftsLabel.Text = "Кол-во перестановок: " +
countingSortSwapCount.ToString();
    if (!disableOutput.Checked)
    {
        countingBox.BeginUpdate();
        countingBox.Items.AddRange(Array.ConvertAll<int, string>(countingSortData, x
=> x.ToString()));
        countingBox.EndUpdate();
    }

    SetStatus("Сортировка завершена!");
}

private void CountingSortWorkerProgressChanged(object sender,
ProgressChangedEventArgs args)
{
}

private void CountingSortBackgroundWork(object sender, DoWorkEventArgs e)
{
    countingSortData = new int[data.Length];
    countingSortData = Array.ConvertAll<double, int>(data, x => (int)x);
    countingSortCompareCount = 0;
    countingSortSwapCount = 0;
    cTime1 = DateTime.Now;
    CountingSort(countingSortData);
}

```

```

private int Partition(double[] a, int p, int r)
{
    int i = p;
    for (int j = p; j <= r; j++)
    {
        if (a[j].CompareTo(a[r]) <= 0)
        {
            double temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            quickSortSwapCount++;
        }
        quickSortCompareCount++;
    }
    return i - 1;
}

```

```

private void QuickSort(double[] array, int p, int r)
{
    int q;
    if (p < r)
    {
        q = Partition(array, p, r);
        QuickSort(array, p, q - 1);
        QuickSort(array, q + 1, r);
    }
}

```

```

private void CountingSort(int[] array)
{
    int[] sorted = new int[array.Length];

    int min = 0;
    int max = 0;

    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] < min)
        {
            min = array[i];
        }
        else if (array[i] > max)
        {
            max = array[i];
        }
        countingSortCompareCount++;
    }

    int[] frequencies = new int[max - min + 1];

    for (int i = 0; i < array.Length; i++)

```



```

    {
        frequencies[array[i] - min]++;
    }

    frequencies[0]--;
    for (int i = 1; i < frequencies.Length; i++)
    {
        frequencies[i] = frequencies[i] + frequencies[i - 1];
    }

    for (int i = array.Length - 1; i >= 0; i--)
    {
        sorted[frequencies[array[i] - min]--] = array[i];
    }

    Array.Copy(sorted, array, data.Length);
}

private int previousValue = 0;
private void SetProgressBarStatus(int value)
{
    if (value != previousValue)
    {
        toolStripProgressBar.Value = value;
    }
}

private void SetStatus(string status)
{
    toolStripStatusLabel.Text = status;
}
}
}

```

5.3 Лабораторная работа № 3

Требования к лабораторной работе:

- 1) создание графического пользовательского интерфейса;
- 2) реализация одного из трех алгоритмов внешней сортировки файлов: прямое слияние, естественное слияние или многопутевое слияние;
- 3) оперативную память использовать только для сравнения минимального количества элементов;
- 4) подготовить несколько файлов с исходной неотсортированной последовательностью разной длины для демонстрации работы алгоритма.

Пример исходного кода для лабораторной работы:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace MLab3
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void selectFileButton_Click(object sender, EventArgs e)
        {
            using (OpenFileDialog openFileDialog = new OpenFileDialog())
            {
                openFileDialog.InitialDirectory = "c:\\users\\bortn\\desktop\\IO Files\\";

                if (openFileDialog.ShowDialog() == DialogResult.OK)
                {
                    fileNameTextBox.Text = openFileDialog.FileName;
                    outputBox.Text = File.ReadAllText(openFileDialog.FileName).Replace(" ",
"\n");
                    sortButton.Enabled = true;
                }
            }
        }

        private void sortButton_Click(object sender, EventArgs e)
        {
            SimpleMergeSort(fileNameTextBox.Text);
            outputBox.Text = File.ReadAllText(fileNameTextBox.Text).Replace(" ", "\n");
        }

        private void SimpleMergeSort(string fileName)
        {
            DateTime time1;
            DateTime time2;
            TimeSpan elapsedTime;

            double a1 = 0;
            double a2 = 0;
            int k = 0;
            int i = 0;

```

```

int j = 0;
int count = 0;

time1 = DateTime.Now;

StreamReader sr = new StreamReader(fileName);

while (!sr.EndOfStream)
{
    a1 = ReadDouble(sr);
    count++;
}

sr.Close();

k = 1;
while (k < count)
{
    sr = new StreamReader(fileName);
    StreamWriter sw1 = new StreamWriter("f1.txt");
    StreamWriter sw2 = new StreamWriter("f2.txt");

    //if (!sr.EndOfStream) a1 = ReadDouble(sr);
    while (!sr.EndOfStream)
    {
        for (i = 0; i < k && !sr.EndOfStream; i++)
        {
            a1 = ReadDouble(sr);
            WriteDouble(sw1, a1);
        }

        for (j = 0; j < k && !sr.EndOfStream; j++)
        {
            a1 = ReadDouble(sr);
            WriteDouble(sw2, a1);
        }
    }

    sr.Close();
    sw1.Close();
    sw2.Close();

    StreamWriter sw = new StreamWriter(fileName);
    StreamReader sr1 = new StreamReader("f1.txt");
    StreamReader sr2 = new StreamReader("f2.txt");

    //

    bool a1NeedUpdate = true;
    bool a2NeedUpdate = true;

    while (!sr1.EndOfStream && !sr2.EndOfStream)

```

```

{
    i = 0;
    j = 0;

    bool flag = false;

    while (i < k && j < k && !flag/*&& !sr1.EndOfStream &&
!sr2.EndOfStream*/)
    {
        if (a1NeedUpdate && !sr1.EndOfStream)
        {
            a1NeedUpdate = false;
            a1 = ReadDouble(sr1);
        }

        if (a2NeedUpdate && !sr2.EndOfStream)
        {
            a2NeedUpdate = false;
            a2 = ReadDouble(sr2);
        }

        if (a1 < a2)
        {
            WriteDouble(sw, a1);
            a1 = double.NaN;
            a1NeedUpdate = true;
            i++;
        }
        else
        {
            WriteDouble(sw, a2);
            a2 = double.NaN;
            a2NeedUpdate = true;
            j++;
        }

        if (double.IsNaN(a1) && sr1.EndOfStream)
            flag = true;

        if (double.IsNaN(a2) && sr2.EndOfStream)
            flag = true;
    }

    while (i < k)
    {
        if (double.IsNaN(a1))
        {
            if (sr1.EndOfStream)
                break;

            a1 = ReadDouble(sr1);
        }
    }

```

```

else
{
    WriteDouble(sw, a1);
    a1 = double.NaN;
    a1NeedUpdate = true;
    i++;
}
}

//while (i < k && !sr1.EndOfStream)
//{{
// if (a1NeedUpdate)
// {
//     a1NeedUpdate = false;
//     a1 = ReadDouble(sr1);
// }
// WriteDouble(sw, a1);
// a1 = double.NaN;
// a1NeedUpdate = true;
// i++;
//}}

while (j < k)
{
    if (double.IsNaN(a2))
    {
        if (sr2.EndOfStream)
            break;

        a2 = ReadDouble(sr2);
    }
    else
    {
        WriteDouble(sw, a2);
        a2 = double.NaN;
        a2NeedUpdate = true;
        j++;
    }
}

//while (j < k && !sr2.EndOfStream)
//{{
// if (a2NeedUpdate)
// {
//     a2NeedUpdate = false;
//     a2 = ReadDouble(sr2);
// }
// WriteDouble(sw, a2);
// a2 = double.NaN;
// a2NeedUpdate = true;
// j++;
}}
```

```

    //}
}

//

while (!sr1.EndOfStream)
{
    if (a1NeedUpdate)
    {
        a1NeedUpdate = false;
        a1 = ReadDouble(sr1);
    }
    WriteDouble(sw, a1);
    a1 = double.NaN;
    a1NeedUpdate = true;
}

while (!sr2.EndOfStream)
{
    if (a2NeedUpdate)
    {
        a2NeedUpdate = false;
        a2 = ReadDouble(sr2);
    }
    WriteDouble(sw, a2);
    a2 = double.NaN;
    a2NeedUpdate = true;
}

if (!double.IsNaN(a1))
{
    WriteDouble(sw, a1);
    a1 = double.NaN;
}

if (!double.IsNaN(a2))
{
    WriteDouble(sw, a2);
    a2 = double.NaN;
}

sw.Close();
sr1.Close();
sr2.Close();

k *= 2;
}

time2 = DateTime.Now;
elapsedTime = time2 - time1;

```

```

        sortTimeLabel.Text = string.Format("Время сортировки: {0} ms",
elapsedTime.TotalMilliseconds);

        File.Delete("f1.txt");
        File.Delete("f2.txt");
    }

private double ReadDouble(StreamReader reader)
{
    string input = "";
    char ch = '\0';

    while ((ch = (char)reader.Read()) != ' ' && !reader.EndOfStream)
    {
        input += ch;
    }

    return Convert.ToDouble(input);
}

private void WriteDouble(StreamWriter writer, double d)
{
    writer.Write(d + " ");
}
}
}

```

5.4 Лабораторная работа № 4

Требования к лабораторной работе:

- 1) создание графического пользовательского интерфейса;
- 2) реализация поиска подстрок двумя алгоритмами: простейший поиск и алгоритм Кнута-Морриса-Пратта;
- 3) организовать вывод всех найденных сдвигов;
- 4) во время работы алгоритмов выполнять подсчет количества сравниваемых элементов;
- 5) для алгоритма Кнута-Морриса-Пратта предусмотреть вывод префикс-функции.

Пример исходного кода для лабораторной работы:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

```

```

namespace MLab4
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        int simpleEntryCount = 0;
        int simpleComparisonCount = 0;

        int kmpEntryCount = 0;
        int kmpComparisonCount = 0;

        List<int> entries = new List<int>();

        void KMPSearch(string text, string pattern)
        {
            int M = pattern.Length;
            int N = text.Length;

            int[] lps = new int[M];
            int j = 0;

            ComputePrefixFunction(pattern, M, lps);

            kmpPrefixFunctionLabel.Text += "[ ";
            foreach (int prefixLength in lps)
            {
                kmpPrefixFunctionLabel.Text += prefixLength.ToString() + ", ";
            }
            kmpPrefixFunctionLabel.Text =
kmpPrefixFunctionLabel.Text.Remove(kmpPrefixFunctionLabel.Text.Length - 2, 2);
            kmpPrefixFunctionLabel.Text += " ]";

            int i = 0;
            while (i < N)
            {
                if (pattern[j] == text[i])
                {
                    j++;
                    i++;
                }
                kmpComparisonCount++;

                if (j == M)
                {
                    kmpEntryCount++;
                    // обрабатываем найденное вхождение
                }
            }
        }
    }
}

```



```

        //Console.WriteLine("Found pattern "
        //+ "at index " + (i - j));

        j = lps[j - 1];
    }
    else if (i < N && pattern[j] != text[i])
    {
        if (j != 0)
            j = lps[j - 1];
        else
            i = i + 1;
    }
}
}

void ComputePrefixFunction(string pattern, int M, int[] lps)
{
    int len = 0;
    int i = 1;
    lps[0] = 0;

    while (i < M)
    {
        if (pattern[i] == pattern[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else
            {
                lps[i] = len;
                i++;
            }
        }
        kmpComparisonCount++;
    }
}

void SimpleSearch(string text, string pattern)
{
    int M = pattern.Length;
    int N = text.Length;
    int counter = 0;

    for (int i = 0; i < N - M + 1; i++)

```

```

    {
        counter = 0;
        for (int j = 0; j < M; j++)
        {
            simpleComparisonCount++;
            if (text[i + j] == pattern[j])
            {
                counter++;
            }
            else
            {
                break;
            }

            if (counter == M)
            {
                entries.Add(i);
                simpleEntryCount++;
            }
        }
    }
}

private void searchButton_Click(object sender, EventArgs e)
{
    entries.Clear();
    inputTextBox.SelectionStart = 0;
    inputTextBox.SelectAll();
    inputTextBox.SelectionBackColor = Color.White;

    kmpPrefixFunctionLabel.Text = "КМП Префикс функция:";

    simpleEntryCount = 0;
    simpleComparisonCount = 0;
    SimpleSearch(inputTextBox.Text, patternTextBox.Text);
    simpleEntryCountLabel.Text = "Кол-во вхождений: " + simpleEntryCount;
    simpleComparisonCountLabel.Text = "Кол-во сравнений: " +
simpleComparisonCount;

    kmpEntryCount = 0;
    kmpComparisonCount = 0;
    KMPSearch(inputTextBox.Text, patternTextBox.Text);
    kmpEntryCountLabel.Text = "Кол-во вхождений: " + kmpEntryCount;
    kmpComparisonCountLabel.Text = "Кол-во сравнений: " + kmpComparisonCount;

    foreach (int entry in entries)
    {
        inputTextBox.SelectionStart = entry;
        inputTextBox.SelectionLength = patternTextBox.Text.Length;
        inputTextBox.SelectionBackColor = Color.Yellow;
    }
}
}
}

```

5.5 Лабораторная работа № 5

Требования к лабораторной работе:

- 1) создание графического пользовательского интерфейса;
- 2) реализация одного из двух алгоритмов на графах: поиск в ширину или поиск в глубину;
- 3) после завершения работы алгоритма вывести все найденные пути из исходной вершины;
- 4) реализовать блок построения графа, включающий возможность добавления/удаления вершин и ребер. Предусмотреть возможность построения ориентированных ребер.

Пример исходного кода для лабораторной работы:

Поиск в ширину:

```
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
int f;
char ch,ch1;
int color[9],d[9],pred[9];
typedef struct zveno * svqz;
struct zveno
{int Key;//вершина графа
svqz Sled;//указатель на следующую смежную вершину
};

class spisok
{public:
int data;
spisok *next;}*curs=NULL, *head=NULL, *end=NULL;

int count=0,i,poi,poi1=0,mov;

void add_end(int x)
{if(!head)
{end=new spisok;
head=end; end->next=end;
end->data=x;count++;}
else{curs=end;end=new spisok;end->next=head;
curs->next=end; end->data=x;count++;}
}

void del_nach()
{if (!end){printf("Нечего удалять"); return;}
curs=head->next;delete head;
```

```

head=curs;
end->next=head;count--;
if (count==0){head=NULL;end=NULL;}
}

```

```

void view() //просмотр
{ setfillstyle(1,0);bar(50,30,320,40);
  if (!head){return;}
  curs=head;char s1[5]; setcolor(7); outtextxy(55,31,"Очередь серых: ");
  for(i=0;i<count;i++){sprintf(s1,"%d ",curs->data);
                                outtextxy(175+i*10,31,s1);curs=curs->next;}

  getch();
}

```

```

class Spisok
{svqz beg[9];//Описание типа списков смежности
svqz res;//указатель на найденное звено
void Poisk(svqz,int);
public:
Spisok(int N);
svqz GetPoisk(){return res;}
void MakeGraph(int N);
void AddGraph(int x,int y);
void PrintGraph(int N);
void Printgraph2(int N,int z);
void bfs(int N,int s);
};

```

```

void Spisok::AddGraph(int x,int y)
{svqz ukzv,uzel,temp; //упорядочить граф
if(beg[x])
{Poisk(beg[x],y);
if(!GetPoisk()){uzel=new(zveno);uzel->Key=y; uzel->Sled=NULL;
                ukzv=beg[x];
                if (ukzv->Key>y){temp=beg[x];beg[x]=uzel;uzel->Sled=temp;}
                else{ while (ukzv->Sled&&ukzv->Sled->Key<y) ukzv=ukzv->Sled;
                        temp=ukzv->Sled; ukzv->Sled=uzel; uzel->Sled=temp;
                }
            }
}
else{beg[x]=new(zveno);(beg[x])->Key=y;(beg[x])->Sled=NULL;}
}

```

```

void Spisok::Poisk(svqz uksp,int ment)
{svqz q;res=NULL; q=uksp;
while(q&&!res){if(q->Key==ment)res=q;q=q->Sled;}
}

```

```

void Spisok::MakeGraph(int N)/////////
{int x,y;
textcolor(3); gotoxy(5,7);cprintf("Нач. вершина ");clreol();
textcolor(2);gotoxy(5,8);cprintf("Конеч. вершина ");clreol();
}

```

```

do{ textcolor(15);
gotoxy(20,7);do{x=getch()-48;}while(x<0||x>N);cprintf("%d",x);
gotoxy(20,8);do{y=getch()-48;}while(y<0||y>N);cprintf("%d",y);
if(x){AddGraph(x,y);if(ch=='1')AddGraph(y,x);
    PrintGraph(N);
    }
}while(x);
}

```

```

void Spisok::PrintGraph(int N)
{ gotoxy(5,15); textcolor(12);
if(ch=='1'){cprintf("Неориентированный граф");printf("\n");}
else {cprintf("Ориентированный граф");printf("\n");}
printf("\n"); svqz ukzv; textcolor(7);
for(int i=1;i<=N;i++)
{cprintf("...%d ",i); ukzv=beg[i];
if(!ukzv){cprintf(" Список пуст");printf("\n");}
else{clrhol();
    while(ukzv){cprintf("%d ",ukzv->Key);ukzv=ukzv->Sled;}
    printf("\n");}
}
}

```

```

void fullV(int k){setfillstyle(1,color[k]);floodfill(70*k,193,5);
}

```

```

void Spisok::Printgraph2(int N,int z)
{ svqz ukzv;setcolor(5);char *s;
for(int k=1;k<=N;k++)circle(70*k,200,20);
for(int x=1;x<=N;x++){setcolor(5);sprintf(s,"%d",x);outtextxy(70*x-4,195,s);}
for(int i=1;i<=N;i++)
{ukzv=beg[i];
while(ukzv)
{if(i==ukzv->Key){setcolor(13);ellipse(70*i,160,0,360,15,19);}
else
if(z=='1'){setcolor(1); ellipse(35*(ukzv->Key+i),180-2,0,180,35*(ukzv->Key-
i),20+abs(i-ukzv->Key)*15);}
else
if(i>ukzv->Key)
{setcolor(11); ellipse(35*(i+ukzv->Key),220+2,180,360,35*(i-ukzv-
>Key),20+(i-ukzv->Key)*15);}
else{setcolor(9); ellipse(35*(ukzv->Key+i),180-2,0,180,35*(ukzv->Key-i),20-(i-
ukzv->Key)*15);}
ukzv=ukzv->Sled;
}
}
setcolor(9);outtextxy(0,400,"Синий цвет – направление слева направо");
setcolor(11);outtextxy(0,410,"Красный цвет – направление справа налево");
setcolor(15);outtextxy(0,430,"Нажмите любую кнопку для выхода.");
}

```

```

Spisok::Spisok(int N)

```

```

{for(int i=0;i<=N;i++)beg[i]=NULL;}

void Spisok::bfs(int N,int s)//поиск в ширину
{svqz ukzv,uzel;
for(int u=1;u<=N;u++){color[u]=15;d[u]=-1;pred[u]=-1;fullV(u);}
color[s]=7; d[7]=0; pred[s]=0; fullV(s);add_end(s);view();
while(count)
{int i=head->data; ukzv=beg[i];
while(ukzv)
{if(color[ukzv->Key]==15)
{color[ukzv->Key]=7;fullV(ukzv->Key); d[ukzv->Key]=d[i]+1;
pred[ukzv->Key]=i; add_end(ukzv->Key);view();}
ukzv=ukzv->Sled;
}
del_nach(); color[i]=0;fullV(i);view();
}
char s0[100],s1[100],s2[10]; int x=350,y=5; setcolor(14);
for(i=1;i<=N;i++)
{s0[0]='\0',s1[0]='\0',s2[0]='\0';
u=pred[i]; sprintf(s0," От %d до %d : ",s,i);
if(u<0) sprintf(s1,"Нет пути ");
else
if(!u) sprintf(s1,"Это начальная вершина ");
else
{s1[0]='\0'; sprintf(s2," %d ",i);strcat(s1,s2);
while(u!=s){sprintf(s2," %d ",u);u=pred[u];strcat(s1,s2);}
sprintf(s2," %d ",s);strcat(s1,s2);
}
strcat(s0,s1);
outtextxy(x,y+i*10,s0);
}
}

```

```

void Spisok::dfs_visit(int u)
{svqz ukzv;
color[u]=7;D[u]=++t;fullV(u);
ukzv=beg[u];
while(ukzv)
{if(color[ukzv->Key]==15)
{pred[ukzv->Key]=u;dfs_visit(ukzv->Key);}
ukzv=ukzv->Sled;
}
color[u]=0;F[u]=++t;fullV(u);
}

```

```

void Spisok::dfs(int N) // поиск в глубину

```

```

{for(int u=1;u<=N;u++)
{color[u]=15;fullV(u);pred[u]=-1;}
t=0;
for(int i=1;i<=N;i++)
{if(color[i]==15)dfs_visit(i);}
}

```

```
//-----
char s1[100],s2[100];
for(int y=1;y<=N;y++)
{s1[0]='\0';s2[0]='\0';
u=pred[y];sprintf(s2," Ⓜ® %d: ",y);strcat(s1,s2);
if(u<0)strcat(s1,"КГВ ЁГВЁ");
else{ while(u>0){ sprintf(s2," %d ",u);u=pred[u];strcat(s1,s2);}
}
outtextxy(400,5+y*10,s1);
}
}

void main()// для поиска в ширину
{int x,y,s;textcolor(11);clrscr();
printf("Программа Поиск в ширину. Выберите тип графа :\n"
"1:неориентированный граф   Иначе: ориентированный \n" );
ch=getch();
textcolor(3); sprintf("Введите число вершин (<=8) ");
gotoxy(30 ,3);do{ f=(getch()-48);}while(f<1||f>8);cprintf("%d",f);printf("\n");
textcolor(11);
printf("Введите пути между вершинами ");
printf("\ндля завершения введите 2 раза ноль '0'\n");
Spisok A(f); A.MakeGraph(f);printf("\n");
gotoxy(40,23);textcolor(15);
printf("нажмите любую клавишу..."); gotoxy(10,24);textcolor(12);
printf("Введите вершину от которой нужно найти путь ");scanf("%d",&s);
int GrDr,GrMd;GrDr=9; GrMd=2; //инициализируем графический режим
initgraph(&GrDr,&GrMd,"d:\\bc\\bgi");setbkcolor(0); setcolor(12);
if(ch=='1')outtextxy(170,10,"Неориентированный граф ");
else outtextxy(170,10,"Ориентированный граф");
int temp1=f;
A.Printgraph2(f,ch); getch();
f=temp1;
A.bfs(f,s);
setcolor(15);outtextxy(500,150,"КОНЕЦ");getch();
closegraph();
}

void main() //для поиска в глубину
{int x,y,s;textcolor(11);clrscr();
printf("Программа Поиска в глубину. Выберите тип графа :\n"
"1:неориентированный граф   Иначе: ориентированный \n" );
ch=getch();
textcolor(3); sprintf("Введите число вершин (<=8) ");
gotoxy(30 ,3);do{ f=(getch()-48);}while(f<1||f>8);cprintf("%d",f);printf("\n");
textcolor(11);
printf("Введите пути между вершинами ");
printf("\ндля завершения введите 2 раза ноль '0'\n");
Spisok A(f); A.MakeGraph(f);printf("\n");
gotoxy(40,23);textcolor(15);
printf("нажмите любую клавишу..."); gotoxy(10,24);textcolor(12);
printf("Введите вершину от которой нужно найти путь ");scanf("%d",&s);
```

```

int GrDr,GrMd;GrDr=9; GrMd=2; //инициализируем графический режим
initgraph(&GrDr,&GrMd,"d:\\bc\\bgi");setbkcolor(0); setcolor(12);
if(ch=='1')outtextxy(170,10,"Неориентированный граф ");
else outtextxy(170,10,"Ориентированный граф");
int temp1=f;
A.Printgraph2(f,ch); getch();
f=temp1;
A.dfs(f);
setcolor(15);outtextxy(500,150,"КОНЕЦ");getch();
closegraph();
}

```

5.6 Лабораторная работа № 6

Требования к лабораторной работе:

- 1) создание графического пользовательского интерфейса;
- 2) организовать построение двоичного дерева поиска с помощью процедур ДЕРЕВО-ДОБАВЛЕНИЕ и ДЕРЕВО-УДАЛЕНИЕ;
- 3) реализовать процедуры ДЕРЕВО-ПОИСК, ДЕРЕВО-МИНИМУМ, ДЕРЕВО-МАКСИМУМ, ДЕРЕВО-СЛЕДУЮЩИЙ, ДЕРЕВО-ПРЕДЫДУЩИЙ и ДЕРЕВО-ОБХОД;
- 4) предусмотреть возможность добавления ключей-дубликатов.

Пример исходного кода для лабораторной работы:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace LabWork6_BinaryTree
{
    public partial class Form1 : Form
    {
        public string key;
        public List<int> NodeList = new List<int>();

        public static double l = 50.0;
        public static List<string> UsesVertex = new List<string>();
        public Form1()
    }
}

```



```

    {
        InitializeComponent();
    }
private int[] ParseInput()
{
    string[] input = textBox1.Text.Split(' ');
    int[] res = null;
    if (input.Length > 0)
    {
        res = new int[input.Length];
        for (int i = 0; i < input.Length; i++)
        {
            if (int.TryParse(input[i], out res[i]) == false)
            {
                res = null;
                break;
            }
        }
    }
    return res;
}
public void Draw(int[] input)
{
    Tree tree;
    tree = Factory.CreateOrganizedTree(input);
    TreeDraw td = new TreeDraw(2.0);
    workspace.Image = td.DrawTree(tree);
}
private void drawEdgeButton_Click(object sender, EventArgs e)
{
    NodeList = textBox1.Text.ToString().Split().Select(int.Parse).ToList();
    int[] input = ParseInput();
    Draw(input);
}

private void textBox1_TextChanged(object sender, EventArgs e)
{
    if (textBox1.Text.Length == 0) drawEdgeButton.Enabled = false;
    else drawEdgeButton.Enabled = true;
}

private void getData_Click(object sender, EventArgs e)
{
    workspace.Image = null;
    NodeList.Clear();
    textBox1.Clear();
}

```

```

private void label5_Click(object sender, EventArgs e)
{
}

private void button2_Click(object sender, EventArgs e)
{
    int keyDelete = Convert.ToInt32(textBox4.Text);
    if (NodeList.Contains(keyDelete))
    {
        NodeList.Remove(keyDelete);

        textBox1.Text = string.Join(" ", NodeList.ToArray());
        if (NodeList.Count != 0 )
        {
            int[] input = ParseInput();
            Draw(input);
        }
        else workSpace.Image = null;
    }
}

public Node Find(int key)
{
    return Find(key, Tree.root);
}

public Node Find(int key, Node parent)
{
    if (parent != null)
    {
        if (key == parent.KeyValue)
        {
            return parent;
        }
        if (key < parent.KeyValue)
        {
            return Find(key, parent.LeftNode);
        }
        else
        {
            return Find(key, parent.RightNode);
        }
    }
    return null;
}

private void button1_Click(object sender, EventArgs e)
{
    label10.Text = "";
    label11.Text = "";
    label4.Text = "";
    label6.Text = "";
    Node node = Find(Convert.ToInt32(textBox2.Text));
}

```

```

if (node != null)
{
    workspace.Image = DrawKey(node, node.Parrent);
    //label11.Text = node.Parrent.KeyValue.ToString();
    label10.Text = "";
    label11.Text += PredSuccessor(node) + " ";
    label10.Text += Successor(node) + " ";
    label4.Text = Max(Tree.root).ToString();
    label6.Text = Min(Tree.root).ToString();
}

}

public int Successor(Node node)
{
    Node x = node;
    if (node.RightNode != null)
        return Min(node.RightNode);
    Node y = node.Parrent;
    while (y != null && x == node.RightNode)
    {
        x = y;
        y = node.Parrent;
    }
    return y.KeyValue;
}

public int PredSuccessor(Node node)
{
    Node x = node;
    if (node.LeftNode != null)
        return Max(node.LeftNode);
    Node y = node.Parrent;
    while (y != null && x == node.LeftNode)
    {
        x = y;
        y = y.Parrent;
    }
    if(y == null)
    {
        return -1;
    }
    return y.KeyValue;
}

}

public int Max(Node node)
{
    int max = node.KeyValue;
    while (node.RightNode != null)
    {
        max = node.RightNode.KeyValue;
        node = node.RightNode;
    }
}

```

```

        return max;
    }
    public int Min(Node node)
    {
        int min = node.KeyValue;
        while (node.LeftNode != null)
        {
            min = node.LeftNode.KeyValue;
            node = node.LeftNode;
        }
        return min;
    }
    public Bitmap DrawKey(Node key, Node parrent)
    {
        Tree tree;
        int[] input = ParseInput();
        tree = Factory.CreateOrganizedTree(input);
        TreeDraw td = new TreeDraw(2.0);
        var im = td.DrawTree(tree);
        im = td.KeyNode(im, key, Tree.root, tree);
        return im;
    }

    private void groupBox4_Enter(object sender, EventArgs e)
    {
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LabWork6_BinaryTree
{
    class Factory
    {
        public static Tree CreateOrganizedTree(params int[] keys)
        {
            if (keys == null)
            {
                throw new ArgumentNullException("keys");
            }

            Tree tree = new Tree();
            foreach (int key in keys)
            {
                tree.Add(key);
            }
        }
    }
}

```

```

        return tree;
    }
    private static Node CreateOrganizedNode(int[] a, int left, int right, Node parrent, Tree
tree)
    {
        if (left <= right)
        {
            int middle = (left + right) / 2;
            Node node = tree.Add(a[middle]);
            CreateOrganizedNode(a, left, middle - 1, node, tree);
            CreateOrganizedNode(a, middle + 1, right, node, tree);
            return node;
        }
        else
        {
            return null;
        }
    }
}
}

```

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace LabWork6_BinaryTree
{
    public class Node
    {
        private Node parrent;
        int X;
        int Y;
        private Node leftNode;
        private Node rightNode;
        private int key;
        private readonly int depth;

        public Node(Node parrent, int keyValue)
        {
            this.parrent = parrent;
            this.key = keyValue;

            this.LeftNode = null;
            this.RightNode = null;

            if (this.Parrent == null)
            {
                this.depth = 0;
            }
        }
    }
}

```

```

        else
        {
            this.depth = this.Parrent.Depth + 1;
        }
    }

    public Node Parrent => parrent;
    public int KeyValue => key;
    public int Depth => depth;
    public Node LeftNode { get => leftNode; set => leftNode = value; }
    public Node RightNode { get => rightNode; set => rightNode = value; }

    public bool IsRoot()
    {
        return this.Parrent == null;
    }

}

}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LabWork6_BinaryTree
{
    public class Tree
    {
        public static Node root;
        public int depth;

        public Tree()
        {
            root = null;
            depth = 0;
        }

        public Node Root { get => root; set => root = value; }
        public int Depth
        {
            get => depth;

            set
            {
                if (Depth < value)
                {
                    this.depth = value;
                }
            }
        }
    }
}

```

```

    }
}

private Node SearchNode(int keyValue)
{
    Node currentNode = this.Root;

    while (currentNode != null && currentNode.KeyValue != keyValue)
    {
        int key = currentNode.KeyValue;
        if (keyValue > key)
        {
            currentNode = currentNode.RightNode;
        }
        else
        {
            currentNode = currentNode.LeftNode;
        }
    }

    return currentNode;
}

private Node CreateNode(int keyValue)
{
    Node currentNode = root;
    Node oldNode = null;

    while (currentNode != null)
    {
        oldNode = currentNode;
        if (keyValue < currentNode.KeyValue)
        {
            currentNode = currentNode.LeftNode;
        }
        else
        {
            currentNode = currentNode.RightNode;
        }
    }
    return AddNode(keyValue, oldNode);
}

private Node AddNode(int keyValue, Node oldNode)
{
    Node newNode = new Node(oldNode, keyValue);
    if (oldNode == null)
    {
        Root = newNode;
    }
    else if (keyValue > oldNode.KeyValue)
    {

```

```

        oldNode.RightNode = newNode;
    }
    else
    {
        oldNode.LeftNode = newNode;
    }
    Depth = newNode.Depth;
    return newNode;
}

private bool Exists(int keyValue, out Node foundNode)
{
    foundNode = SearchNode(keyValue);
    return foundNode != null;
}

public bool Exists(int keyValue)
{
    return Exists(keyValue, out Node FoundNode);
}

public Node Add(int keyValue)
{
    Node newNode;
    if (Exists(keyValue, out newNode) == false)
    {
        newNode = CreateNode(keyValue);
    }
    return newNode;
}

}
}

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LabWork6_BinaryTree
{
    public class TreeDraw
    {
        private double factor;
        private int size;

        private int marginTop = 50;
        private int marginLeft = 10;
        Font fo;
    }
}

```



```

public TreeDraw(double scale)
{
    factor = scale;
    size = (int)(10 * factor);
    fo = new Font("Aerie", (int)(3 * factor));
}
public Bitmap DrawTree(Tree tree)
{
    Bitmap img = new Bitmap(ImageWidth(tree), ImageHeight(tree));
    Graphics gr = Graphics.FromImage(img);
    gr.Clear(Color.White);
    gr.CompositingQuality =
System.Drawing.Drawing2D.CompositingQuality.HighQuality;
    gr.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality;
    Node root = tree.Root;
    int startPos = ((int)Math.Pow(2, tree.Depth) * size) - size;
    DrawNode(root, startPos, gr, tree.Depth);
    gr.Dispose();
    return img;
}
public Bitmap KeyNode(Bitmap im, Node key, Node parent, Tree tree)
{
    int startPos = ((int)Math.Pow(2, tree.Depth) * size) - size;
    Graphics gr = Graphics.FromImage(im);
    gr.CompositingQuality =
System.Drawing.Drawing2D.CompositingQuality.HighQuality;
    gr.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality;
    KeyNodeDraw(parent, startPos, tree.Depth, key, gr);

    return im;
}
public void KeyNodeDraw(Node parent, int x, int maxDepth, Node key, Graphics gr)
{ if (parent != null)
    {
        int y = parent.Depth * size + 1 + marginTop;
        int margin = ((int)Math.Pow(2, (maxDepth - parent.Depth)) * (size / 2));
        if (key.KeyValue == parent.KeyValue)
        {
            drawSelect(parent, gr, x, y);
        }
        if (key.KeyValue < parent.KeyValue )
        {
            KeyNodeDraw(parent.LeftNode, x - margin, maxDepth, key, gr);
        }
        else
        {
            KeyNodeDraw(parent.RightNode, x + margin, maxDepth, key, gr);
        }
    }
}
}

```

```

private int ImageHeight(Tree tree)
{
    return (tree.Depth + 1) * size + (marginTop * 2);
}

private int ImageWidth(Tree tree)
{
    return (int)(Math.Pow(2, tree.Depth) * 2 * size) + marginLeft;
}

private void DrawNode(Node node, int x, Graphics gr, int maxDepth)
{
    if (node != null)
    {
        int y = node.Depth * size + 1 + marginTop;
        int margin = ((int)Math.Pow(2, (maxDepth - node.Depth)) * (size / 2));

        if (node.RightNode != null) gr.DrawLine(new Pen(Color.Black), x + 20, y + 10,
x + 25 + margin, y + 25);
        if (node.LeftNode != null) gr.DrawLine(new Pen(Color.Black), x + 20, y + 10, x +
25 - margin, y + 25);
        CreateNode(node, gr, x, y);
        DrawNode(node.LeftNode, x - margin, gr, maxDepth);

        DrawNode(node.RightNode, x + margin, gr, maxDepth);
    }
}

private void CreateNode(Node node, Graphics gr, int x, int y)
{
    gr.FillEllipse(Brushes.White, x + marginLeft, y, size, size);
    gr.DrawEllipse(new Pen(Color.Black, (float)(1 * factor)), x + marginLeft, y, size,
size);
    Rectangle rectangle = new Rectangle(x + marginLeft, y, size, size);
    StringFormat sf = new StringFormat(StringFormatFlags.DirectionRightToLeft);
    sf.Alignment = StringAlignment.Center;
    sf.LineAlignment = StringAlignment.Center;
    gr.DrawString(node.KeyValue.ToString(), fo, Brushes.Black, rectangle, sf);
}

private void drawSelect(Node node, Graphics gr, int x, int y)
{
    gr.FillEllipse(Brushes.White, x + marginLeft, y, size, size);
    gr.DrawEllipse(new Pen(Color.Red, (float)(1 * factor)), x + marginLeft, y, size, size);
    Rectangle rectangle = new Rectangle(x + marginLeft, y, size, size);
    StringFormat sf = new StringFormat(StringFormatFlags.DirectionRightToLeft);
    sf.Alignment = StringAlignment.Center;
    sf.LineAlignment = StringAlignment.Center;
    gr.DrawString(node.KeyValue.ToString(), fo, Brushes.Black, rectangle, sf);
}
}
}

```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

- 1 Кормен Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – Москва : МЦНМО, 2002. – 960 с.
- 2 Седжвик Р. Алгоритмы на Java / Р. Седжвик, К. Уэйн. – 4-е изд. – Москва : ООО «И. Д. Вильямс», 2013. – 848 с.
- 3 Кормен Т. Алгоритмы: вводный курс / Т. Кормен. – Москва : ООО «И. Д. Вильямс», 2014. – 208 с.
- 4 Дасгупта С. Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани. – Москва : МЦНМО, 2014. – 320 с.
- 5 Макконнелл Д. Основы современных алгоритмов / Д. Макконнелл. – 2-е доп. изд. – Москва : Техносфера, 2004. – 368 с.
- 6 Уоррен Г. Алгоритмические трюки для программистов / Г. Уоррен. – 2-е изд. – Москва : ООО «И. Д. Вильямс», 2014. – 512 с.
- 7 Ахо А. Построение и анализ вычислительных алгоритмов / А. Ахо, Д. Хопкрофт, Д. Ульман. – Москва : Мир, 1979. – 536 с.
- 8 Вирт Н. Алгоритмы и структуры данных / Н. Вирт. – Москва : Мир, 1989. – 360 с.

Учебное издание

Маер Алексей Владимирович
Черепанов Олег Сергеевич

ВВЕДЕНИЕ В СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Учебное пособие

Редактор А. С. Темирова

Подписано в печать 29.03.21	Формат 60x84 1/16	Бумага 80 г/м ²
Печать цифровая	Усл. печ. л. 6,7	Уч.-изд. л. 6,7
Заказ 49	Тираж 100	

БИЦ Курганского государственного университета.
640020, г. Курган, ул. Советская, 63/4.
Курганский государственный университет.