А. М. Семахин

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. ПРИЛОЖЕНИЯ НА C++

УЧЕБНОЕ ПОСОБИЕ

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение
высшего образования
«Курганский государственный университет»

А. М. Семахин

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. ПРИЛОЖЕНИЯ НА C++

Учебное пособие

Курган 2025

УДК 519.254 (075.8)
ББК 32.973.05я73
С30

Рецензенты:

кандидат физико-математических наук, профессор кафедры программирования и автоматизации бизнес-процессов Шадринского государственного педагогического университета Владислав Юрьевич Пирогов;

кандидат физико-математических наук, доцент кафедры экономики Российской академии народного хозяйства и государственной службы при Президенте Российской Федерации (Курганский филиал) Вячеслав Михайлович Солодовников.

Печатается по решению методического совета Курганского государственного университета.

Семахин А. М.

Алгоритмы и структуры данных. Приложения на C++ : учебное пособие / А. М. Семахин. – Курган : Изд-во КГУ, 2025. – 164 с.

В учебном пособии рассматриваются линейные структуры данных, нелинейные структуры данных, сортировка данных, поиск данных, алгоритмы приложений на графах. Для закрепления теоретических знаний и приобретения практических навыков в решении задач в учебном пособии приводятся контрольные вопросы и варианты заданий для выполнения самостоятельных и курсовых работ. Учебное пособие рекомендуется для проведения занятий по дисциплине «Алгоритмы и структуры данных» со студентами, обучающимися по направлению подготовки группы 09.00.00 «Информатика и вычислительная техника», и может быть использовано специалистами, занимающимися разработкой программных приложений, реализующих алгоритмы решения задач.

Рис. – 62, библиограф. – 12.

ISBN 978-5-4217-0720-2

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Алгоритмы и структуры данных – это методы организации и хранения данных, разработки процедур (алгоритмов) для решения задач, оперирующих структурами данных.

Изучение этих методов имеет фундаментальное значение для разработки эффективных алгоритмов и оптимизации производительности программного обеспечения. Приобретение практических навыков в реализации алгоритмов решения задач на ПЭВМ студентами высших учебных заведений является актуальной задачей. Обеспечение учебного процесса печатными изданиями по изучению алгоритмов и структур данных является важным этапом в подготовке квалифицированных специалистов.

Учебное пособие имеет целью дать студентам теоретические знания и практические навыки в разработке алгоритмов решения задач по вариантам заданий и формализации на ПЭВМ с использованием языка программирования Visual C++ с применением технологии визуального проектирования и событийного программирования.

Пособие включает теоретическое обоснование линейных структур данных, нелинейных структур данных, сортировки данных, поиска данных и приложений на графе. Для закрепления теоретических знаний, полученных на лекционных занятиях по дисциплине «Алгоритмы и структуры данных» и приобретения практических навыков в разработке программных приложений приводятся контрольные вопросы и варианты заданий для выполнения лабораторных и курсовых работ.

Пособие рекомендуется для проведения занятий по дисциплине «Алгоритмы и структуры данных» и смежных дисциплин в учебном процессе студентов, обучающихся по направлению подготовки группы 09.00.00 «Информатика и вычислительная техника», и может быть использовано специалистами, занимающимися разработкой программных приложений, реализующих алгоритмы решения задач.

1 АЛГОРИТМ, ВИДЫ И СВОЙСТВА АЛГОРИТМОВ

1.1 Основные понятия и определения

Алгоритм – точное предписание, которое задаёт вычислительный процесс, начинающийся из некоторой совокупности возможных для этого алгоритма исходных данных и направленный на получение полностью определённого этими исходными данными результата.

Алгоритм – последовательность действий, приводящая к решению задачи.

Структура данных – способ организации и хранения данных в компьютере, обеспечивающий эффективный доступ к данным и их изменение. Для оценки сложности и скорости работы алгоритма используют «О-нотацию» или «О-большое».

Информация – сведения об объектах и явлениях окружающей среды, их параметрах, свойствах и состоянии, которые уменьшают имеющуюся о них степень неопределенности, неполноты знаний.

Данные в информатике – сообщения, представленные в виде, допускающем обработку программно-аппаратными средствами или интерпретацию человеком. Обработка данных выполняется в соответствии с программно- или аппаратно-реализованными алгоритмами решения задач [1; 2].

1.2 Виды алгоритмов

Алгоритмы подразделяются на:

1) алгоритм, представленный на естественном языке;

2) графический алгоритм – алгоритм, представленный в виде блок-схемы;

3) программный алгоритм – алгоритм, реализованный на языке программирования.

Например, алгоритм решения системы линейных алгебраических уравнений (СЛАУ) методом Гаусса. Пусть дана система СЛАУ с n неизвестными (1.1).

$$\begin{cases} a_{00}x_0 + a_{01}x_1 + \ldots + a_{0n-1}x_{n-1} = b_0 \\ a_{10}x_0 + a_{11}x_1 + \ldots + a_{1n-1}x_{n-1} = b_1 \\ \ldots \\ a_{n-10}x_0 + a_{n-11}x_1 + \ldots + a_{n-1n-1}x_{n-1} = b_{n-1} \end{cases} \quad (1.1)$$

Матрица коэффициентов левых частей уравнений СЛАУ (1.1) имеет вид.

$$A = \begin{pmatrix} a_{00}\ a_{01} \ldots a_{0n-1} \\ a_{10}\ a_{11} \ldots a_{1n-1} \\ \ldots \\ a_{n-10}\ a_{n-11} \ldots a_{n-1n-1} \end{pmatrix} \quad (1.2)$$

Вектор-столбец правых частей уравнений СЛАУ (1.1) имеет вид.

$$b = \begin{pmatrix} b_0 \\ b_1 \\ \ldots \\ b_{n-1} \end{pmatrix} \quad (1.3)$$

Вектор-столбец искомых переменных уравнений СЛАУ (1.1) имеет вид.

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \ldots \\ x_{n-1} \end{pmatrix} \quad (1.4)$$

Метод Гаусса – алгоритм последовательного исключения неизвестных переменных СЛАУ. При использовании метода Гаусса преобразования производят над расширенной матрицей системы, которая получается путём добавления к основной матрице A столбца свободных членов b.

I Алгоритм Гаусса, представленный на естественном языке

Этап 1. Прямой ход метода Гаусса. Приведение расширенной матрицы 1.5 к треугольному виду: все элементы матрицы 1.5, расположенные ниже главной диагонали должны быть равны нулю.

$$A' = \begin{pmatrix} a_{00}\ a_{01} \ldots a_{0n-1} & b_0 \\ a_{10}\ a_{11} \ldots a_{1n-1} & b_1 \\ \ldots & \\ a_{n-10}\ a_{n-11} \ldots a_{n-1n-1} & b_{n-1} \end{pmatrix} \quad (1.5)$$

В результате прямого хода метода Гаусса матрица 1.5 преобразуется в матрицу, имеющую вид 1.6.

$$A=\begin{pmatrix} a_{00} & a_{01} & a_{02} \dots a_{0n-1} & b_0 \\ 0 & a_{11} & a_{12} \dots a_{1n-1} & b1 \\ 0 & 0 & a_{22} \dots a_{2n-1} & b_2 \\ \dots & & & \\ 0 & 0 & 0 \dots & a_{n-1n-1}\ b_{n-1} \end{pmatrix} \quad (1.6)$$

Система линейных алгебраических уравнений 1.1 будет иметь вид

$$\begin{cases} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \dots + a_{0n-1}x_{n-1} = b_0 \\ a_{11}x_1 + a_{12}x_2 + \dots + a_{1n-1}x_{n-1} = b_1 \\ a_{22}x_2 + \dots + a_{2n-1}x_{n-1} = b_2 \\ \dots \\ a_{n-1n-1}x_{n-1} = b_{n-1} \end{cases} \quad (1.7)$$

Этап 2. Обратный ход метода Гаусса. Решение системы уравнений 1.7. Формула для вычисления i-го значения x имеет вид [3]

$$x_i = \frac{b_i - \sum_{j=i+1}^{n-1} a_{ij}x_j}{a_{ii}} \quad (1.8)$$

II Графический алгоритм метода Гаусса (блок-схема алгоритма решения СЛАУ методом Гаусса) представлен на рисунке 1.1.

III Программный алгоритм метода Гаусса, реализованный на языке программирования C++, приведён в листинге 1.1.

Листинг 1.1 – Решение СЛАУ методом Гаусса

```
/* Метод Гаусса. Возвращает 0, если решение найдено, -1 – бесконечное множество решений, -2 – система не имеет  решений. Параметры функции: n – размерность  системы, matrica_a – матрица коэффициентов СЛАУ, massiv_b – вектор правых частей, x – решение СЛАУ, matrica_a, massiv_b, x передаются как указатели */
int Slau(double** matrica_a, int n, double* massiv_b, double* x) {
  int i, j, k, r;
  double c, M, max, s;
/* Матрица a – копия матрицы коэффициентов, матрица b – копия вектора правых частей */
  double** a, * b;
```

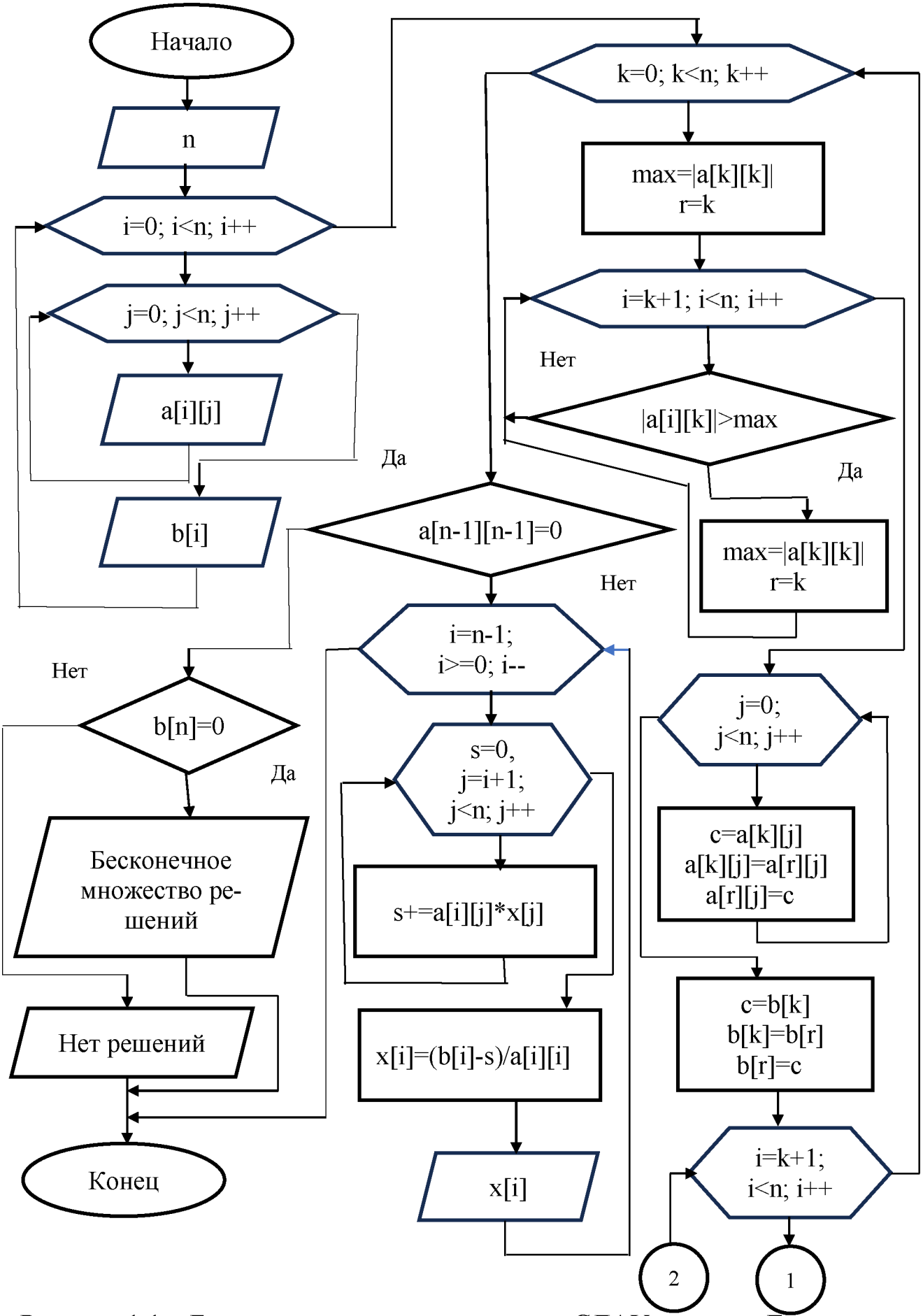
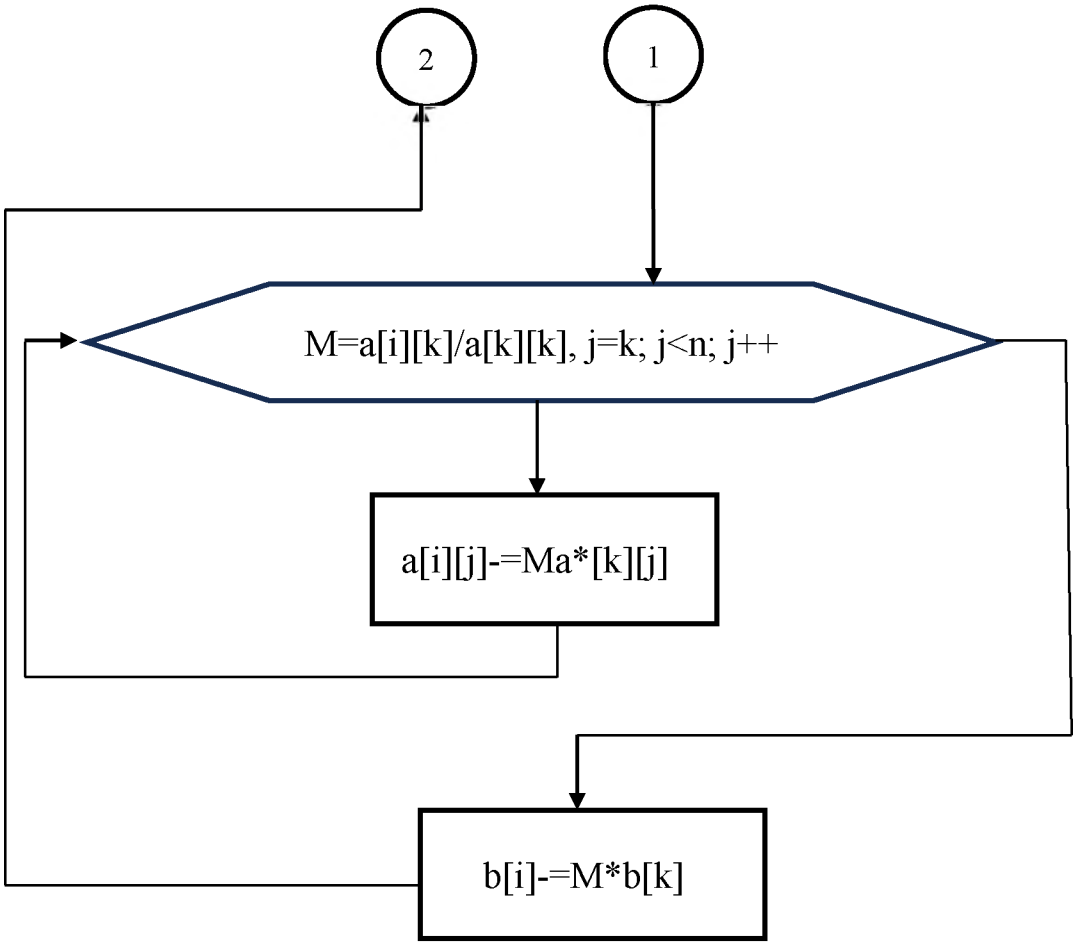


Рисунок 1.1 – Блок-схема алгоритма решения СЛАУ методом Гаусса



Продолжение рисунка 1.1

```
        //Выделение памяти для a и b
        a = new double* [n];
        for (i = 0; i < n; i++) a[i] = new double[n];
        b = new double[n];
        for (i = 0; i < n; i++)
           for (j = 0; j < n; j++) a[i][j] = matrica_a[i][j];
        for (i = 0; i < n; i++) b[i] = massiv_b[i];
//Прямой ход метода Гаусса: приведение к диагональному виду
        for (k = 0; k < n; k++) {
           //Поиск максимального по модулю элемента в k-ом столбце
           max = fabs(a[k][k]); r = k;
           for (i = k + 1; i < n; i++)
              if (fabs(a[i][k]) > max)
              { max = fabs(a[i][k]); r = i; }
```

```
/* Меняем местами k-ю и r-ю (строку, где находится максимальный
по модулю элемент) строки */
            for (j = 0; j < n; j++)
            { c = a[k][j]; a[k][j] = a[r][j]; a[r][j] = c; }
            c = b[k]; b[k] = b[r]; b[r] = c;
      //Приведение матрицы к диагональному виду
         for (i = k + 1; i < n; i++) {
            for (M = a[i][k] / a[k][k], j = k; j < n; j++) a[i][j] -= M * a[k][j];
               b[i] -= M * b[k]; } }
         //Обратный ход метода Гаусса.
         //Если последний диагональный элемент равен 0 и
         if (a[n - 1][n - 1] == 0)
       //если последний коэффициент вектора свободных членов равен 0,
            if (b[n - 1] == 0)
               //то система имеет бесконечное множество решений
               return -1;
      /* если последний коэффициент вектора свободных членов не равен 0,
      то система решений не имеет */
            else return -2;
         else
      /* Если последний диагональный элемент не равен 0, то начинается
      обратный ход метода Гаусса */
         { for (i = n - 1; i >= 0; i--) {
               for (s = 0, j = i + 1; j < n; j++)
                  s += a[i][j] * x[j]; x[i] = (b[i] - s) / a[i][i]; }
            return 0; }
      /* Освобождение памяти, выделенной под копии матрицы коэффи-
      циентов, вектора правых частей */
         for (i = 0; i < n; i++) delete[] a[i]; delete[] a; delete[] b; } [3]
```

1.3 Свойства алгоритмов

Свойства алгоритмов сформулировал русский математик А. А. Марков:

1 *Конечность* – работа алгоритма должна заканчиваться за конечное число шагов, шаги могут повторяться.

2 *Определённость (детерминированность)* – предписания алгоритма должны иметь одинаковую трактовку и понятны исполнителю алгоритма. Каждый шаг алгоритма должен быть однозначным, недвусмысленным, при одних и тех же исходных данных должны быть получены одинаковые результаты.

3 *Массовость* – решение группы задач, отличающихся исходными данными.

4 *Результативность* – алгоритм приводит к достижению цели за конечное число шагов. Алгоритм не может быть прерван из-за неопределённых ситуаций.

5 *Эффективность* – алгоритм выполняется за конечное время, а общее время работы должно лежать в допустимых для решения задачи пределах [4].

1.4 Основные этапы подготовки задачи к решению на ЭВМ

Основные этапы подготовки задачи к решению на ЭВМ:

1 *Постановка задачи* – формулирование задачи.

2 *Построение модели задачи* – подбор или разработка формальной модели задачи.

3 *Разработка алгоритма* – определение существования решения аналогичных задач, выделение абстрактных типов данных.

Абстрактные типы данных (АТД) – типы данных, определяющие организацию (структуру) и область значений конкретных данных задачи и операции над данными.

4 *Реализация алгоритма в виде программы.*

5 *Проверка правильности алгоритма.* На этапе разработки алгоритма доказывается правильность шагов алгоритма, конечность алгоритма, проверяются допустимые входные данные, полученные выходные данные. На этапе программной реализации алгоритма необходимо, чтобы в процессе преобразования правильного алгоритма получить правильную программу.

Правильность (корректность) – свойство программы, характеризующее отсутствие в ней ошибок по отношению к целям разработки. Программа определяется правилами представления (синтаксис языка программирования) и правилами интерпретации её содержательного значения (семантикой). Задача определения синтаксической правильности решается

транслятором автоматически в ходе трансляции программы. Семантическая правильность программы проверяется тестированием.

Тестирование – процесс обнаружения ошибок в программном приложении.

Спецификация задачи – формализованное описание постановки задачи.

Верификация программ – процесс определения соответствия программных приложений требованиям спецификации задачи.

6 *Анализ сложности алгоритма* – процесс определения соответствия алгоритма требованиям:

- простой для понимания, отладки и сопровождения;
- эффективный по затратам времени и памяти ЭВМ.

Если программа выполняется несколько раз, а время выполнения для пользователя несущественно, предпочтение отдаётся первому требованию, т. к. стоимость создания программы намного дороже стоимости машинного времени, затрачиваемого на решение задачи.

Если задача для своего решения требует значительных вычислительных ресурсов, выполняется многократно или время решения ограничено внешними событиями как в системах реального времени, то требование эффективности становится более важным [4].

1.5 Контрольные вопросы

1 Какое существует определение понятия «алгоритм»?

2 Какие виды имеет алгоритм?

3 Какие свойства имеет алгоритм?

4 Какое существует определение понятия «структуры данных»?

5 Какие существуют этапы подготовки задачи к решению на ЭВМ?

6 Какое существует определение понятия «абстрактные типы данных»?

7 Что понимается под спецификацией задачи?

8 Что понимается под верификацией программы?

9 Какое существует определение понятия «правильность (корректность) программы»?

10 Какое существует определение понятия «информация»?

2 АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ

Требования, предъявляемые к алгоритмам:

1) простой для понимания, перевода в программный код и отладки;

2) эффективно использовать вычислительные ресурсы и быстро выполняться.

Сложность алгоритма – количественная характеристика ресурсов, необходимых алгоритму для решения поставленной задачи.

Функция сложности О выражает относительную скорость алгоритма в зависимости от переменных. Для функции сложности сформулированы три правила:

1) $O(k*f)=O(f)$. Постоянные множители не имеют значения для определения порядка сложности, например, $O(1.5*N)=O(N)$;

2) $O(f*g)=O(f)*O(g)$, $O(f/g)=O(f)/O(g)$. Порядок сложности произведения двух функций равен произведению их сложностей, например, $O(15*N*N)=O(15*N)*O(N)=O(N)*O(N)=O(N^2)$;

3) $O(f+g)$ равна доминанте $O(f)$ и $O(g)$. Порядок сложности суммы функций определяется как порядок доминанты первого и второго слагаемых (выбирается наибольший порядок), например, $O(N^5+N^2)=O(N^5)$,

где k – константа, f и g – функции.

2.1 Виды функции сложности алгоритмов

Время выполнения алгоритма T зависит от объема входных данных N (рисунок 2.1) и записывается в виде уравнения

$$T=f(N), \qquad (2.1)$$

где T – время выполнения алгоритма, мс;

N – объем входных данных.

O-нотация позволяет учитывать в функции *f(n)* значимые элементы, отбрасывая второстепенные:

1) кубическая функция – функция *f(n)*, старший член которой содержит N^3. *O*-нотация имеет вид $O(N^3)$;

2) квадратическая функция – функция *f(n)*, старший член которой содержит N^2. О-нотация имеет вид $O(N^2)$;

3) линейная функция – функция *f(n)*, старший член которой содержит N. *O*-нотация имеет вид $O(N)$;

4) функция линерифмическая – функция, старший член которой равен N логарифмов N. O-нотация имеет вид $O(N\log N)$.

Для оценивания трудоемкости алгоритмов используется *O*-нотация.

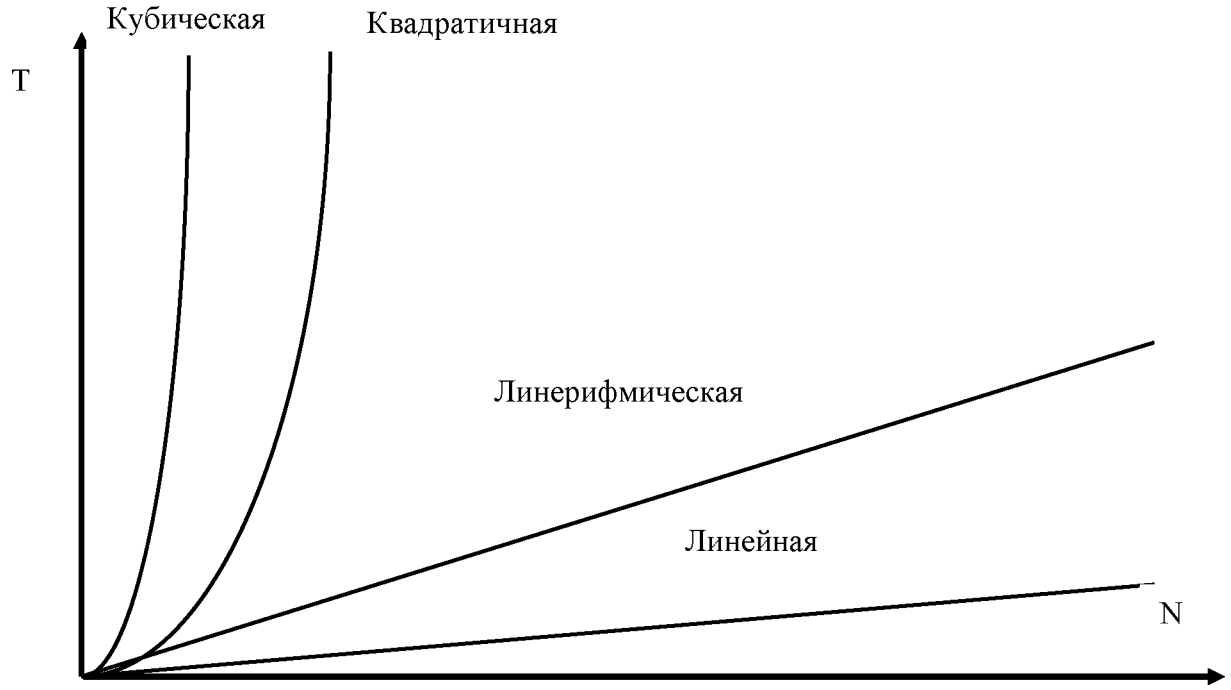


Рисунок 2.1 – Графики функций сложности алгоритмов

O-оценивание позволяет описывать характер поведения функции *f(n)* с ростом *n*: насколько быстро или медленно растет эта функция.

O-оценка разбивает функции сложности на группы в зависимости от скорости роста:

1) постоянные функции $O(1)$, которые с ростом *n* не растут;

2) функции с логарифмической скоростью роста $O(\log_2 n)$;

3) функции с линейной скоростью роста $O(n)$;

4) функции с линейно-логарифмической скоростью роста $O(n*\log_2 n)$;

5) функции с квадратичной скоростью роста $O(n^2)$;

6) функции со степенной скоростью роста $O(n^a)$ при $a>2$;

7) функции с показательной или экспоненциальной скоростью роста $O(2^n)$;

8) функции с факториальной скоростью роста $O(n!)$.

Описание *O*-нотаций приведено в таблице 2.1

Таблица 2.1 – Описание *O*-нотаций

O-нотация	**O-описание**
O(1)	Константная зависимость. Инструкции программы запускаются независимо от n. Время выполнения программы постоянно (помещение в стек). Операции в программе выполняются один или несколько раз. Алгоритм независимо от размера данных требует одно и то же время. Например, обращение к элементу массива по индексу
O(n)	Линейная зависимость. Время выполнения программы линейно и зависит от n. Входной элемент обрабатывается линейное число раз. Например, перебор элементов массива
$O(n^2)$	Квадратическая зависимость. Время выполнения программы является квадратичным. Алгоритмы используются для небольших n (цикл двойного уровня вложенности, сортировки выбором, вставками)
$O(n^3)$	Кубическая зависимость. Алгоритм программы имеет кубическое время выполнения (цикл тройного уровня вложенности). Применяется для небольших задач (умножение матриц)
O(log n)	Логарифмическая зависимость. С ростом n программа работает медленнее. Время характерно для программ, которые сводят большую задачу к набору меньших подзадач, уменьшая на каждом шаге размер подзадачи на постоянный коэффициент. Общее решение находится в одной из подзадач (бинарный поиск)
O(n*log n)	Линерифмическая зависимость. Время выполнения программы пропорционально n*logn. Возникает, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решает независимо и затем объединяет решения подзадач (комбинация, сортировки быстрая, слиянием)
$O(2^n)$	Экспоненциальная зависимость. Прямое решение задач (перебор и сравнение различных решений). Для комбинаторных задач нереализуемы. Сведение к приближенному алгоритму с приближенным значением. Например, алгоритмы перебора (brute force)
O(n!)	Факториальная зависимость. Самая высокая степень роста времени выполнения алгоритма, при которой время растёт пропорционально факториалу размера входных данных. Например, алгоритмы комбинаторики (сочетания, перестановки)

2.2 Оценка эффективности алгоритмов

Программный алгоритм имеет два параметра:

Временная сложность алгоритма – время, затрачиваемое алгоритмом на его выполнение как функция размера задачи.

Ёмкостная сложность алгоритма – число (размер задачи), выражающее меру количества (объёма) исходных данных.

При вычислении времени выполнения программ с применением функции оценки сложности алгоритма *O(n)* применяют правила сумм и произведения.

Правило сумм. Пусть время выполнения двух фрагментов программы $T_1(n)$ и $T_2(n)$ имеет порядок $O(f_1(n))$ и $O(f_2(n))$ соответственно. Время последовательного выполнения этих фрагментов вычисляется как $T_1(n) + T_2(n) = O(f_1(n)) + O(f_2(n))$ и имеет порядок $O(max(f_1(n), f_2(n))$.

Правило произведений. $T_1(n) * T_2(n) = O(f_1(n)) * O(f_2(n))$.

Время выполнения операторов базовых структур программ определяется по этим правилам.

1 Время выполнения последовательности операторов определяется по правилам сумм, т. е. равно наибольшему времени выполнения одного оператора в данной последовательности.

2 Время выполнения операторов ветвления определяется наибольшим временем выполнения одной из ветвей оператора.

3 Время выполнения цикла определяется как произведение времени выполнения тела цикла и количества повторений цикла.

2.3 Классы сложности задач

При анализе сложности алгоритмов используются понятия детерминированного и недетерминированного алгоритмов, введеные Флойдом в 1967 г. В каждый момент времени выполнения алгоритм находится в определённом состоянии. Состояние определяется совокупностью значений обрабатываемых данных и выполняемым действием.

Детерминированный алгоритм – алгоритм, который выдаёт уникальный и предопределённый результат для заданных входных данных.

Недетерминированный алгоритм – алгоритм, указывающий несколько путей обработки одних и тех же входных данных, без уточнения, какой именно вариант будет выбран.

Класс P-задачи (polinomial) – задачи, которые могут быть решены с помощью детерминированного алгоритма за полиномиальное время. *P*-задачи считаются легко полиномиальными разрешимыми задачами.

Класс E-задачи – задачи, сложность которых не менее f^n (f – константа, или полином от n). *E*-задачи считаются экспоненциальными задачами [4].

Класс NP-задачи (nondetrministically polynomial) – задачи, решаемые с помощью недетерминированного алгоритма за полиномиальное время.

2.4 Пример анализа программного алгоритма

Определите оценку функции сложности алгоритма метода сортировки BucketSort. Сделайте вывод об эффективности алгоритма.

O(P)=O(1); O(O)=O(1); O(N)=O(n)*O(O)=O(n)*O(1)=O(n);

O(L)=O(1); O(M)=O(1); O(K)=O(n)*O(M)=O(n)*O(1)=O(n);

O(J)=O(1); O(I)=O(n)*O(J)=O(n)*O(1)=O(n);

O(H)=O(1); O(Q)=O(n)*O(H)=O(n)*O(1)=O(n);

O(G)=O(n)*O(Q)=O(n)*O(n)= $O(n^2)$;

O(F)=O(n)*O(G)=O(n)* $O(n^2)$= $O(n^3)$;

O(E)=O(1); O(D)=O(1); O(C)=O(n)*O(D)=O(n)*O(1)=O(n);

O(B)=O(1); O(A)=O(P)+O(N)+O(L)+O(K)+O(I)+O(F)+O(E)+O(C)+

+O(B)=O(1)+O(n)+O(1)+O(n)+O(n)+ $O(n^3)$+O(1)+O(n)+O(1)= $O(n^3)$.

Оценка функции сложности метода BucketSort кубическая $O(n^3)$. Программный алгоритм неэффективный. При увеличении размера исходного массива время упорядочивания элементов массива возрастает значительно.

2.5 Контрольные вопросы

1 Какое существует определение понятия «сложность алгоритма»?

2 Какие существуют классы сложности алгоритмов?

3 Каким образом формулируется правило сумм?

4 Каким образом формулируется правило произведения?

5 Какие существуют классы сложности задач?

6 Что понимается под временной сложностью алгоритма?

7 Что понимается под ёмкостной сложностью алгоритма?

```
System::Collections::Generic::List<double>^ MyForm1::Bucket_Sort(System::Collections::Generic::List<double>^ List_of_Numbers, int bucket_count) {
    int N = List_of_Numbers->Count; double x_min = List_of_Numbers[0];
        double x_max = List_of_Numbers[0];
    for (int I = 1; I < N; i++) {
        if (List_of_Numbers[i] < x_min) { x_min = List_of_Numbers[i]; }
        else if (List_of_Numbers[i] > x_max) { x_max = List_of_Numbers[i]; } }
    System::Collections::Generic::List<System::Collections::Generic::List<double>^>^ Buckets = gcnew System::Collections::Generic::List<System::Collections::Generic::List<double>^>();
        for (int I = 0; I < bucket_count; i++) {
            System::Collections::Generic::List<double>^ new_bucket = gcnew System::Collections::Generic::List<double>(); Buckets->Add(new_bucket);}
        for (int I = 0; I < N; i++) {
    int     bucket_number=(bucket_count*(List_of_Numbers[i]-x_min))/(x_max-x_min);
            if (bucket_number == bucket_count) { bucket_number--; }
            Buckets[bucket_number]->Add(List_of_Numbers[i]);}
    for (int bucket_number=0; bucket_number<bucket_count; bucket_number++){
        for (int I = Buckets[bucket_number]->Count; I > 1; i--) {
        for (int j = 1; j < I; j++) {
            if (Buckets[bucket_number][j] < Buckets[bucket_number][j – 1]) {
                double aaa = (*Buckets[bucket_number])[j];
                Buckets[bucket_number][j] = Buckets[bucket_number][j – 1];
                Buckets[bucket_number][j – 1] = aaa;}}}}
    System::Collections::Generic::List<double>^ result = gcnew System::Collections::Generic::List<double>();
    for (int bucket_number=0;bucket_number<bucket_count; bucket_number++) {
        System::Collections::Generic::List<double>^bucket=Buckets[bucket_nuber];
        result->AddRange(bucket); }
    return result; }
```

P N O L K M I J A G Q H F E C D B

8 В каких случаях используется квадратическая оценка функции сложности $O(n^2)$?

9 В каких случаях используется экспоненциальная оценка функции сложности $O(2^n)$?

10 В каких случаях используется факториальная оценка функции сложности O(n!)?

2.6 Варианты заданий

Определите оценку функции сложности программного алгоритма метода сортировки MergeSort. Сделайте вывод об эффективности программного алгоритма (Листинг 2.2).

Листинг 2.2 – Программный алгоритм метода сортировки MergeSort

```
void Merging_Sort(int n, float *x) {
    int i, j, k, t, s, Fin1, Fin2;
    float* tmp = new float[n];
    k = 1;
    while (k < n) {
      t = 0;
      s = 0;
      while (t + k < n) {
        Fin1 = t + k;
        Fin2 = (t + 2 * k < n ? t + 2 * k : n);
        i = t;
        j = Fin1;
        for (; i < Fin1 && j < Fin2; s++) {
          if (x[i] < x[j]) {
            tmp[s] = x[i];
            i++;
          }
          else {
            tmp[s] = x[j];
            j++;
          }
        }
        for (; i < Fin1; i++, s++) {
```

```
            tmp[s] = x[i];
        }
        for (; j < Fin2; j++, s++) {
            tmp[s] = x[j];
        }
        t = Fin2;
    }
    k *= 2;
    for (s = 0; s < t; s++) {
        x[s] = tmp[s];
    }
  }
  delete(tmp);
}
```

3 ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ

3.1 Линейный список

Для эффективного решения задач применяют методы динамического использования памяти, т. е. описания структур данных, размер и конфигурация которых изменяются вовремя выполнения программы. В этих случаях используют динамические структуры данных. Применяя структурные типы, указатели и динамические переменные создают динамические структуры данных: списки, деревья, графы:

1) определяется структурный тип *T*, один или несколько элементов которого объявлены указателями на тот или другой структурный тип;

2) в программе объявляется переменная (например, *root*) типа *T* или переменная типа указателя на *T* в случае полностью динамического создания структуры. Имя этой переменной при выполнении программы используется как имя «корня» динамической структуры;

3) при выполнении программы по мере построения динамической структуры запрашиваются динамические переменные соответствующих типов и связываются ссылками, начиная с переменной *root*, или первой динамической переменной, указатель на которую содержится в переменной *root*.

Такой подход позволяет создать динамическую структуру с любой топологией [5].

3.1.1 Функциональное описание линейного списка

Линейный список – линейная динамическая структура данных, представляющая конечную (возможно, пустую) совокупность элементов типа *T*, над которой выполняются операции, определённые функциональным описанием структуры.

Пусть линейный список обозначен перечислением элементов $L = (t_1, t_2, \ldots, t_n)$. В функциональном описании t означает любой элемент списка. Функциональное описание линейного списка представлено в таблице 3.1.

Таблица 3.1 – Функциональное описание элементов линейного списка

Операция	Результат	Пояснение
Создание()	L	Функция без параметров, создающая пустой список
Включение(t, t', L)	L'	Это новый список с включённым элементом t перед t' или в конец списка, если элемент t' не принадлежит L
Исключение(t, L)	L'	Это новый список с исключённым элементом t из списка, если элемент t принадлежит L
Следующий(t, L)	t'	Это элемент списка, следующий за t, если он существует
Первый(L)	t	Это первый элемент списка, если он существует
СписокПуст(L)	Истина/ложь	Проверка, является ли список пустым
Поиск(t, L)	Истина/ложь	Проверка, принадлежит ли элемент t списку
Уничтожение(L)		Функция, уничтожающая список

3.1.2 Логическое описание линейного списка

Логическое описание линейного списка представляет линейный список как последовательность элементов типа T, возможно, пустую. С помощью формул Бэкуса определяется следующим образом:

типЛинСписок=(Пусто|НепустойЛинСписок)

тип НепустойЛинСписок=(начало: T; продолжение: ЛинСписок)

Операции функционального описания для любого списка имеют следующие свойства:

СписокПуст(Создание())=истина – создаётся пустой список;

СписокПуст(Включение(t, L))=ложь – если в список включается элемент, результирующий список не пуст;

Первый(L)=начало – начало непустого списка – его первый элемент.

3.1.3 Физическое представление линейного списка

По способу размещения в памяти компьютера существуют два вида линейных структур данных:

1) последовательный линейный список – элементы списка располагаются последовательно в структуре хранения. Под структуру выделяется непрерывный участок памяти;

2) связный линейный список – порядок следования элементов определяется с помощью специальных указателей, которые расположены в самих элементах и определяют ссылку на следующий элемент списка или на соседний слева и справа.

Последовательные списки реализуются в виде массивов. Элементы списка располагаются в смежных ячейках памяти. Включение и исключение элемента требуют перемещения остальных элементов списка, чтобы освободить место для нового элемента или убрать освободившуюся ячейку. Последовательные списки используются для задач со статическими данными. Для изменяющихся данных используются динамические структуры. Например, линейный связный однонаправленный список. Для него характерно произвольное число элементов, которое может быть нулевым и ограничивается объёмом доступной памяти компьютера. Другая особенность связного списка – необязательная физическая смежность элементов памяти.

Связные списки – для создания связного списка определяется структурный тип *T*, который имеет одно поле *next*, объявленное как указатель на следующий элемент типа *T*. Другие поля структуры содержат информацию, характеризующую элемент списка. При образовании первого элемента списка в поле *next* заносится пустой указатель (0, *nil, NULL*). При дальнейшем построении списка это значение будет находиться в последнем элементе, т. к. следующий элемент списка добавляется перед предыдущим. Нулевой указатель является признаком конца списка. Каждый список должен иметь особый элемент, называемый указателем на начало списка или заголовком списка (header), который обычно по формату отличен от остальных элементов [5].

3.1.4 Классификация линейных списков

Линейные списки подразделяются на:

1 *Односвязный (однонаправленный) линейный список* – линейный список, в котором элементы имеют указатели на следующие элементы (рисунок 3.1).

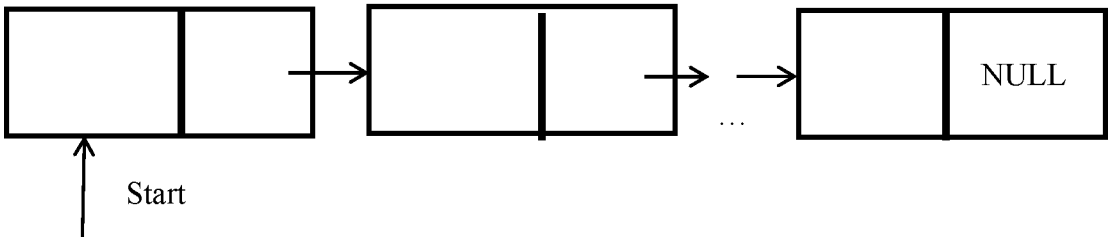


Рисунок 3.1 – Линейный (однонаправленный) односвязный список

2 *Двусвязный (двунаправленный) линейный список* – линейный список, в котором элементы имеют указатели на следующий и предыдущий элементы (рисунки 3.2 – 3.3).

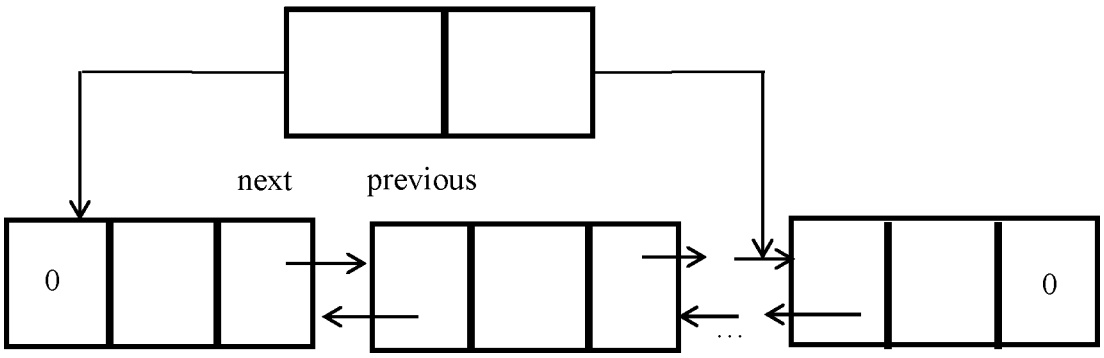


Рисунок 3.2 – Двунаправленные (двусвязные) линейные списки с полями *next* и *previous*

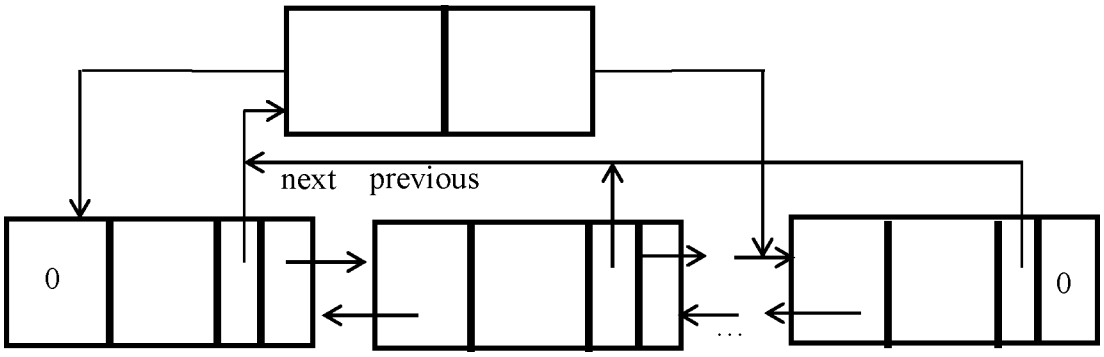


Рисунок 3.3 – Двунаправленные (двусвязные) линейные списки со ссылками на заголовок списка

В случае двусвязного линейного списка в функциональное описание структуры добавляются операции, приведённые в таблице 3.2.

Таблица 3.2 – Функциональное описание двусвязного линейного списка

Операция	Результат	Пояснение
Предыдущий(t, L)	t'	Это элемент списка, предшествующий t, если он существует
Последний(L)	t	Это последний элемент списка, если он существует

3 *Циклический линейный список* – линейный список, в котором добавлена ссылка последнего элемента на первый элемент списка (рисунок 3.4).

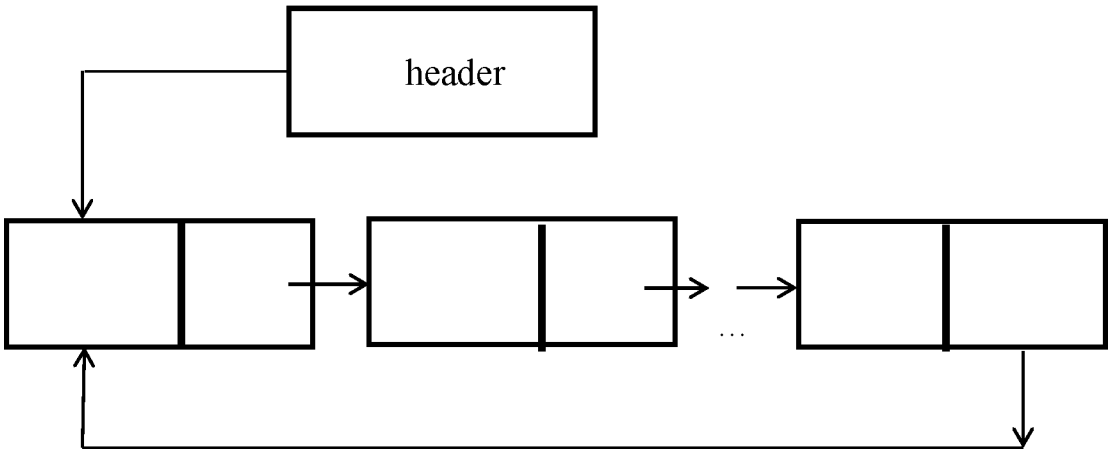


Рисунок 3.4 – Циклический линейный однонаправленный список

4 *Многосвязный линейный список* – линейный список, в котором элементы содержат по несколько ссылок, задающих различные линейные порядки (рисунки 3.5–3.7). Например, «слоёные списки» («списки с пропусками») – многосвязные списки, позволяющие пропустить (skip) ненужные элементы, перемещаясь сразу к нужному элементу линейного списка. Например, обычный односвязный список представлен на рисунке 3.5. Добавив один «уровень» ссылок, можно ускорить поиск нужного элемента (рисунок 3.6). Сначала перебираются элементы по ссылкам уровня 1, затем, когда достигнут нужный отрезок списка, – по ссылкам нулевого уровня. Добавление второго уровня ссылок (рисунок 3.7) позволяет перемещаться по списку ещё быстрее [5; 6].

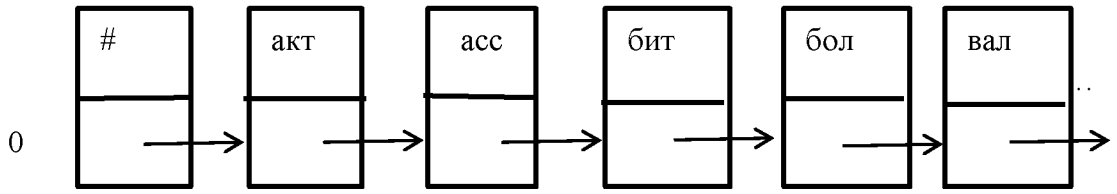


Рисунок 3.5 – «Слоёные» списки с одним уровнем ссылок

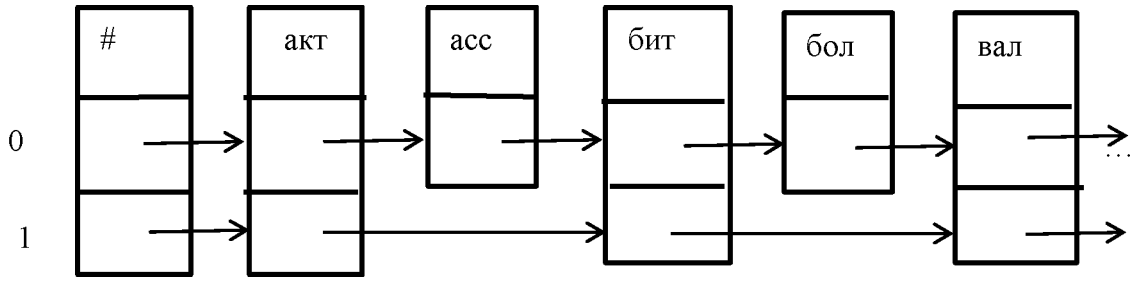


Рисунок 3.6 – «Слоёные» списки с двумя уровнями ссылок

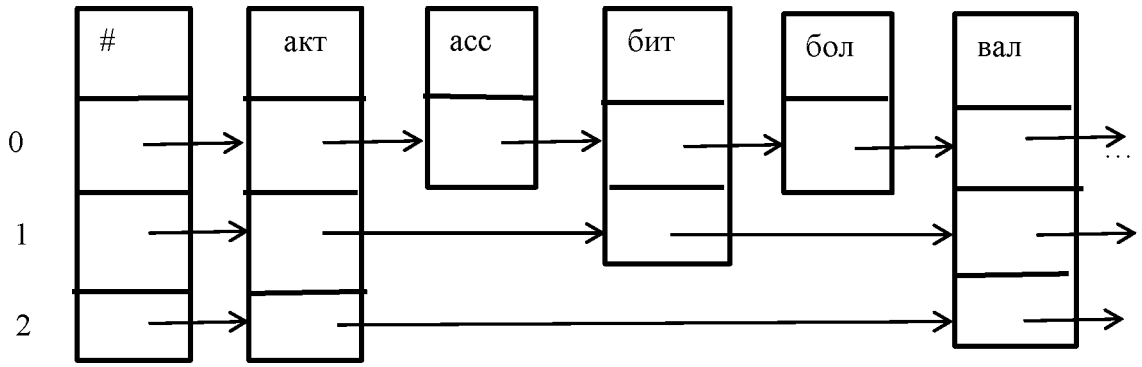


Рисунок 3.7 – «Слоёные» списки с тремя уровнями ссылок

3.1.5 Применение линейных списков

Линейные списки являются гибкой структурой:

1) легко сделать большими или маленькими;

2) легко объединить или разбить на меньшие списки, не переписывая содержимое элементов, а лишь изменяя значения указателей.

Линейные списки используются в приложениях:

- информационный поиск;
- трансляторы языков программирования;
- моделирование процессов;

•механизм управления памятью.

Для оптимизации операций над списком создают вспомогательную переменную-структуру (заголовок списка *header*), состоящую из двух полей – указателей на первый и последний элементы списка (рисунок 3.8).

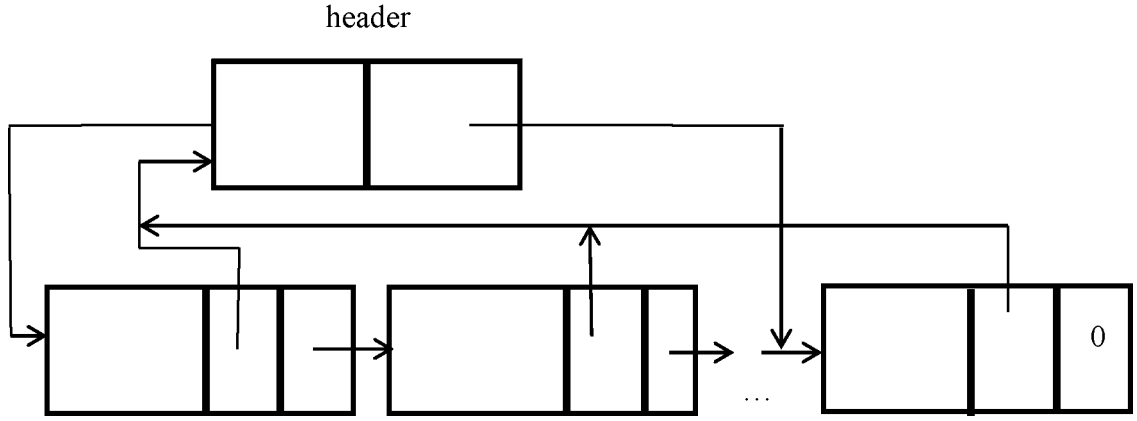


Рисунок 3.8 – Список с заголовком из двух полей-указателей

3.2 Стек

Стек – структура данных, представляющая конечную (возможно, пустую) и упорядоченную совокупность элементов типа *T*, над которой выполняются операции, определённые функциональным описанием структуры (таблица 3.3).

3.2.1 Функциональное описание стека

Таблица 3.3 – Функциональное описание стека

Операция	Результат	Пояснение
Создание()	S	Функция без параметров, создающая пустой стек
Включение(t, S)	S'	Это новый стек с включённым элементом t в вершину стека
Извлечение(S)	S'	Это новый стек с извлечённым элементом из вершины стека, если он не пуст
Последний(S)	t	Это последний (находящийся в вершине) элемент стека, если он существует
СтекПуст(S)	Истина/Ложь	Проверка, является ли стек пустым
Уничтожение()	–	Удаление стека

Пусть стек обозначается $S = (t_1, t_2, \ldots, t_n)$. В функциональном описании t означает любой элемент стека.

Стек является динамической структурой ограниченного доступа, представляющей собой контейнер для хранения данных типа *LIFO (Last Input – First Output*, последним вошел – первым вышел). Включение и извлечение элементов из стека выполняется только с одной стороны динамической структуры, называемой вершиной стека [1; 5].

3.2.2 Логическое описание стека

Логическое описание представляет стек как последовательность элементов типа *T*, возможно, пустую. С помощью формул Бэкуса стек можно определить следующим образом:

тип Стек=(Пусто|НепустойСтек)

тип НепустойСтек=(вершина: T; продолжение: Стек)

Перечисленные операции для любого стека имеют следующие свойства:

СтекПуст(Создание())=истина – создаётся пустой стек;

СтекПуст(Включение(t, S)*)=ложь* – если в стек включается элемент, результирующий стек не пуст;

Извлечение(Включение(t, S))=S – результатом включения элемента в стек и последующего извлечения элемента из вершины стека является стек, идентичный исходному;

Последний(t, S)=t – в вершине стека находится элемент, последним включённый в стек;

Включение(Последний(S),Извлечение(S))=S – результатом извлечения элемента из вершины стека и последующего его включения является стек, идентичный исходному [5].

3.2.3 Физическое представление стека

Стек реализуется в виде:

- массива (вектора);
- связного списка.

Операция включения элемента в стек называют *push*. Операцию извлечения элемента из стека – *pop*.

Векторное представление стека. Стек занимает непрерывный участок памяти, объём которого ограничен. Возможно переполнение стека. Выделение большого участка памяти ведёт к его неэффективному использованию, если стек заполняется только частично. Создание стека сопровождается инициализацией специального дескриптора стека, который может содержать поля: имя стека, адреса нижней и верхней границ стека, указатель на вершину стека.

Представление стека в виде связного списка. Время работы увеличивается в связи с необходимостью постоянных выделений и освобождения участков динамической памяти [5].

3.2.4 Применение стека

Стеки применяются для размещения локальных переменных функций, передачи параметров при вызове функций, реализации рекурсии в программировании, трансляции программ (синтаксический и аналитический анализаторы, генераторы кодов), управления динамической памятью.

3.3. Очередь

Очередь – структура данных, представляющая конечную (возможно, пустую) и упорядоченную совокупность элементов типа *T,* над которой выполняются операции, определённые функциональным описанием структуры (таблица 3.4).

3.3.1 Функциональное описание очереди

Таблица 3.4 – Функциональное описание очереди

Операция	Результат	Пояснение
Создание()	Q	Создание пустой очереди
Включение(t, Q)	Q'	Новая очередь с включённым элементом t в конец очереди
Извлечение(Q)	Q'	Новая очередь с извлечённым элементом из начала очереди, если она не пуста
Первый(Q)	t	Первый (находящийся в начале) элемент очереди, если он существует
ОчередьПуста(Q)	Истина/Ложь	Проверка, является ли очередь пустой
Уничтожение()	–	Удаление очереди

3.3.2 Логическое описание очереди

Логическое описание представляет очередь как последовательность элементов типа *T,* возможно, пустую. С помощью формул Бэкуса очередь можно определить следующим образом:

Тип Очередь=(Пусто|НепустаяОчередь)

Тип Непустая очередь=(начало: T; продолжение: Очередь)

Перечисленные операции для любой очереди имеют следующие свойства:

Очередь Пуста(Создание())=истина – создаётся пустая очередь;

ОчередьПуста(Включение(t, Q))=ложь – если в очередь включается элемент, результирующая очередь не пуста;

Первый(Включение(t, Создание()))=t – первым элементом очереди будет элемент, включённый в созданную пустую очередь;

ОчередьПуста(Извлечение(Включение(t, Создание())))=истина – результатом извлечения элемента из очереди с единственным, включённым в созданную пустую очередь элементом , будет пустая очередь [1; 5].

3.3.3 Физическое представление очереди

Для реализации очереди существуют два основных способа реализации:

1) очередь представляется в виде массива (вектора);

2) очередь представляется в виде связного списка.

Очередь в виде массива. Очередь занимает непрерывный участок памяти. Для исключения перемещения элементов очереди внутри массива используют «кольцевую» реализацию. Очередь обслуживается двумя индексами элементов – начала (header) и конца очереди (tail). Включение очередного элемента (push) осуществляется в конец, извлечение (pop) – из начала. Индекс header идентифицирует старый элемент очереди, индекс tail – первый свободный элемент после последнего включённого в очередь.

Элементы физически по очереди не перемещаются, меняются лишь значения индексов header и tail. Когда tail достигает конца массива, включение элемента в очередь приводит к перемещению индекса tail в начало массива. Если начало массива к этому моменту свободно, то очередное включение произойдёт в его нулевой элемент.

Очередь в виде связного списка. Очередь представляется в качестве линейного связного списка, включение и извлечение элементов в котором осуществляется с разных концов.

Преимущество первого способа по сравнению со вторым является экономия памяти и простота реализации. Недостаток – ограничение на максимальное количество элементов в очереди, что определяется размером массива, в котором она хранится.

Преимущества и недостатки второго способа реализации очереди аналогичны преимуществам и недостаткам при реализации стека в виде списка.

Очередь с приоритетами – структура данных, в которую помещаются элементы, разделённые на несколько групп по приоритетности обработки, и в каком бы порядке данные ни были записаны в очередь, обрабатываться первыми будут элементы с наивысшим приоритетом, а последними – с самым низким приоритетом из тех, которые на данный момент находятся в очереди.

Очередь с приоритетами позволяет хранить пары (приоритет, значение) и поддерживает операции добавления пары, поиска пары с минимальным приоритетом и извлечения пары с минимальным приоритетом.

Существуют следующие способы реализации приоритетной очереди:

1 С использованием связного списка элементов;

2 С использованием частично упорядоченных деревьев;

3 С использованием нескольких «обычных» очередей [5].

3.4 Дек

Дек – структура данных, представляющая конечную (возможно, пустую) и упорядоченную совокупность элементов типа *T,* над которой выполняются операции, определённые функциональным описанием структуры (таблица 3.5).

Дек является динамической структурой ограниченного доступа, реализуемой как очередь с двумя равноправными концами, в которые происходит включение и извлечение элементов [5].

3.4.1 Функциональное описание дека

Функциональное описание дека приведено в таблице 3.5.

Таблица 3.5 – Функциональное описание дека

Операция	Результат	Пояснение
Создание()	D	Функция без параметров, создающая пустой дек
ВключениеСлева(t, D)	D'	Новый дек с включённым элементом t в левый конец дека
ВключениеСправа(t, D)	D'	Новый дек с включённым элементом t в правый конец дека
ИзвлечениеСлева(D)	D'	Если дек не пуст, то это новый дек с извлечённым элементом из левого конца
ИзвлечениеСправа(D)	D'	Если дек не пуст, то это новый дек с извлечённым элементом из правого конца
ПервыйСлева(D)	t	Если дек не пуст, то это первый находящийся в левом конце элемент дека
ПервыйСправа(D)	t	Если дек не пуст, то это первый находящийся в правом конце элемент дека
ДекПуст(D)	Истина/Ложь	Проверка, является ли дек пустым
Уничтожение()	–	Удаление дека

3.4.2 Логическое и физическое представление дека

Логическое и физическое представления дека аналогичны обычной очереди, но говорят не о хвосте и голове, а о левом (начальном, front) и правом (конечном, end) концах дека. Реализация операций включения и исключения в дек дополняется признаками левого и правого концов.

Дек с ограниченным вводом – дек, включение элементов в котором осуществляется лишь в один конец, а извлечение – из обоих.

Дек с ограниченным выводом – дек, в котором извлечение происходит лишь из одного конца дека.

Статическая реализация дека идентична структуре кольцевой очереди.

Динамическая реализация дека осуществляется с помощью линейного двусвязного списка. На базе дека можно создать стек и очередь [2; 5].

3.5 Контрольные вопросы

1 Какое существует определение понятия «линейная динамическая структура данных»?

2 На какие виды подразделяется линейный список?

3 Какой принцип работы имеет очередь?

4 Какой принцип работы имеет стек?

5 Какое существует функциональное описание линейного списка?

6 Какое существует функциональное описание очереди?

7 Какое существует функциональное описание стека?

8 Какое существует функциональное описание дека?

9 Что понимается под деком?

10 Что понимается под стеком?

3.6 Варианты заданий

Вариант 1

Разработать программу, которая содержит информацию о наличии автобусов в автобусном парке.

Сведения о каждом автобусе содержат:

- номер автобуса;
- фамилию и инициалы водителя;
- номер маршрута.

Программа должна обеспечивать:

- начальное формирование данных в виде списка о всех автобусах;
- при выезде каждого автобуса из парка вводится номер автобуса, и программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- при въезде каждого автобуса в парк вводится номер автобуса, и программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- по запросу выдаются сведения об автобусах, находящихся в парке, или об автобусах, находящихся на маршруте.

Вариант 2

Разработать программу, которая содержит информацию о заявках на авиабилеты.

Каждая заявка содержит:

- пункт назначения;
- номер рейса;
- фамилию и инициалы пассажира;
- желаемую дату вылета.

Программа должна обеспечивать:

- хранение всех заявок в виде списка;
- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок [6; 7].

Вариант 3

Разработать программу, которая содержит информацию о книгах в библиотеке.

Сведения о книгах содержат:

- номер УДК;
- фамилию и инициалы автора;
- название;
- год издания;
- количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- начальное формирование данных о всех книгах в библиотеке в виде списка;
- при взятии каждой книги вводится номер УДК, и программа уменьшает значение количества книг на единицу или выдает сообщение о том, что требуемой книги в библиотеке нет или требуемая книга находится на руках;
- при возвращении каждой книги вводится номер УДК, и программа увеличивает значение количества книг на единицу;
- по запросу выдаются сведения о наличии книг в библиотеке.

Вариант 4

Разработать программу, которая содержит информацию о расписании движения поездов дальнего следования. Для каждого поезда указываются:

- номер поезда;
- станция назначения;
- время отправления;
- время прибытия.

Данные в программном приложении организованы в виде линейного списка.

Программное приложение выполняет действия:

- ввод в расписание данных о поезде дальнего следования:
- удаление из расписания данных о поезде дальнего следования:
- вывод всего списка поездов дальнего следования из расписания;
- поиск данных о поезде дальнего следования по номеру поезда [6; 7].

Вариант 5

Разработать программу, отыскивающую проход по лабиринту.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице.

Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода выводит найденный путь в виде координат квадратов. Для хранения пути использовать стек [6; 7].

Вариант 6

Гаражная стоянка имеет одну стояночную полосу, причем въезд и выезд находятся в одном конце полосы. Если владелец автомашины приходит забрать свой автомобиль, который не является ближайшим к выходу, то все автомашины, загораживающие проезд, удаляются, машина данного владельца выводится со стоянки, а другие машины возвращаются на стоянку в исходном порядке.

Разработать программу, которая моделирует процесс прибытия и отъезда машин. Прибытие или отъезд автомашины задается командной строкой, которая содержит признак прибытия или отъезда и номер автомашины. Программа должна выводить сообщение при прибытии или выезде любой автомашины. При выезде автомашины со стоянки сообщение должно содержать число, которое показывает, сколько раз машина удалялась со стоянки для обеспечения выезда других автомобилей.

Вариант 7

Разработать программу, моделирующую заполнение гибкого магнитного диска. Общий объем памяти на диске 360 Кбайт. Файлы имеют произвольную длину от 18 до 32 Кбайт. В процессе работы файлы либо записываются на диск, либо удаляются с него. В начале работы файлы записываются подряд друг за другом. После удаления файла на диске образуется свободный участок памяти, и вновь записываемый файл либо размещается на свободном участке, либо, если файл не вмещается в свободный участок, размещается после последнего записанного файла.

В случае, когда файл превосходит длину самого большого свободного участка, выдается аварийное сообщение. Требование на запись или удаление файла задается в командной строке, которая содержит имя файла, его длину в байтах, признак записи или удаления. Программа должна выдавать по запросу сведения о занятых и свободных участках памяти на диске. Необходимо создать список занятых участков и список свободных участков на диске [6; 7].

Вариант 8

В файловой системе каталог файлов организован как линейный список. Для каждого файла в каталоге содержатся следующие сведения:

- имя файла:
- дата создания;
- количество обращений к файлу;

Разработать программу, которая обеспечивает:

- начальное формирование каталога файлов;
- вывод каталога файлов;

- удаление файлов, дата создания которых меньше заданной;
- выборку файлов с наибольшим количеством обращений.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе [6; 7].

Вариант 9

Предметный указатель организован как линейный список.

Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от 1 до 10. Разработать программу, которая обеспечивает:

- начальное формирование предметного указателя;
- вывод предметного указателя;
- вывод номера страниц для заданного слова.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 10

Текст помощи для программы организован как линейный список.

Каждый компонент текста помощи содержит термин (слово) и текст, содержащий пояснения к этому термину. Количество строк текста, относящихся к одному термину, от одной до пяти. Разработать программу, которая обеспечивает:

- начальное формирование текста помощи;
- вывод текста помощи;
- вывод поясняющего текста для заданного термина.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе [6; 7].

4 НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ

Дерево (Tree) – нелинейная динамическая структура данных, представляющая собой конечную (возможно, пустую) и упорядоченную совокупность элементов типа *T (узлов дерева),* один из которых выделен и называется *корнем дерева,* и конечным (возможно, пустым) множеством деревьев, называемых *поддеревьями* корня.

Корень любого поддерева узла называется *потомком* (*сыном)* этого узла, а сам узел – его *предком* (*отцом).*

Узел дерева, не имеющий потомков, называется *листом (внешним узлом). Внутренним узлом* дерева называется узел, имеющий хотя бы одного потомка (сына).

Пусть узел *y* – потомок узла *x.* Тогда говорят, что узлы *x* и *y* связаны *ребром,* и дерево можно представить в виде графа. Структура данных предполагает прохождение по элементам дерева, начиная всегда с корня, граф будет ориентированным, и каждая его вершина (кроме корня) всегда будет иметь полустепень захода, равную единице, а полустепень исхода будет равна числу сыновей данной вершины. Полустепень захода корня и полустепень исхода листа всегда равны нулю.

Путь от корня до узла – последовательность ребер от корня до узла дерева.

Длина пути – количество ребер, составляющих путь.

Уровень узла x – длина пути от корня до узла.

Высота дерева – максимальное значение уровней листьев.

Степень ветвления узла – количество потомков узла дерева.

Пример дерева высоты 4 в виде графа приведен на рисунке 4.1. На рисунке чёрным цветом помечены листья дерева. Узел *x* расположен на уровне 1 и имеет четырех потомков, одним из которых является узел *y* [5].

4.1 Функциональное описание дерева

Над структурой данных «дерево» выполняются операции, определенные функциональным описанием (таблица 4.1).

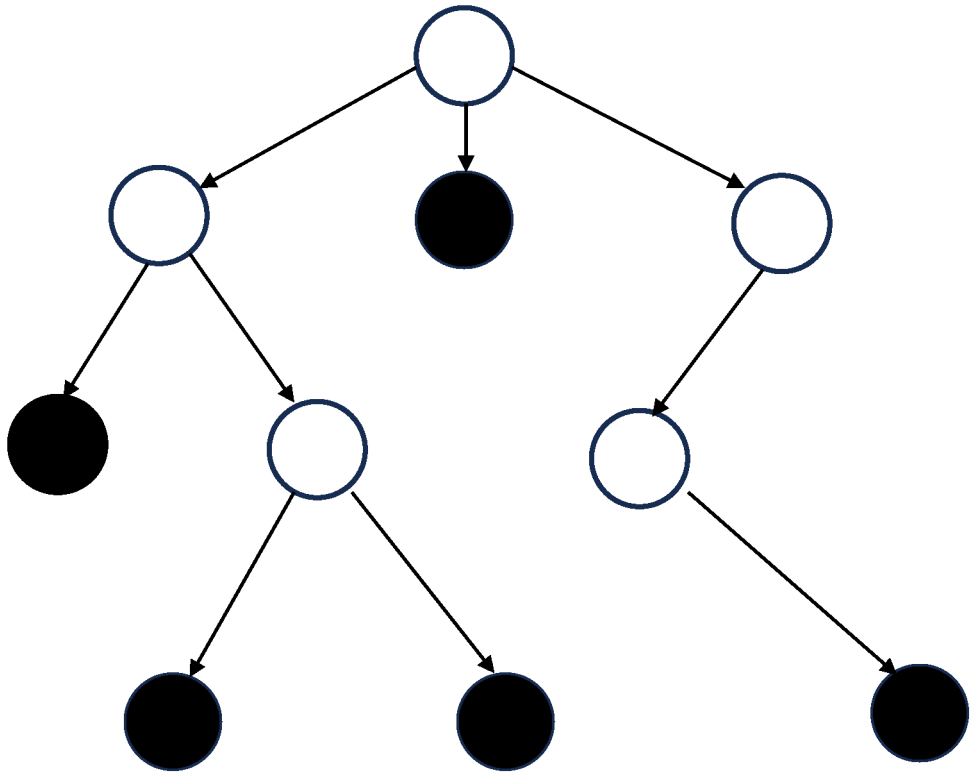


Рисунок 4.1 – Пример дерева. Чёрным цветом помечены листья дерева

Таблица 4.1 – Функциональное описание дерева

Операции	Результат	Пояснение
Создание()	Tree	Создание пустого дерева
Включение(t, Tree)	Tree’	Добавление узла в дерево
Удаление(e, Tree)	Tree’	Удаление узла из дерева
Корень(Tree)	t	Корень дерева
ДеревоПусто(Tree)	Истина/ложь	Проверка дерева на пустоту
Поиск(t, Tree)	Истина/ложь	Поиск узла в дереве
Обход(Tree)	Совокупность узлов дерева	Обход узлов дерева
Уничтожение()	–	Удаление дерева

Пусть обозначение дерева имеет вид:

$$Tree = (t_1, t_2, \ldots, t_n), \tag{4.1}$$

где t – узел дерева.

Двоичное (бинарное) дерево (BinTree) – дерево, каждый узел которого имеет не более двух потомков. Один из них называют *левым потомком*, второй – *правым*.

Пусть *TreeLeft* и *TreeRight* – левое и правое двоичные поддеревья корня двоичного дерева *Tree*. В этом случае в функциональное описание

структуры данных дерево добавляются операции, приведенные в таблице 4.2.

Таблица 4.2 – Дополнительные операции функционального описания дерева

Операции	Результат	Пояснение
ЛевоеПоддерево(Tree)	Tree’	Левое поддерево
ПравоеПоддерево(Tree)	Tree	Правое поддерево
Построение(t, TreeLeft, TreeRight)	Tree	Создание дерева с корнем, левым поддеревом и правым поддеревом

4.2 Логическое описание двоичного дерева

Логическое описание представляет двоичное дерево как последовательность элементов типа T, возможно, пустую. С помощью формул Бэкуса его можно определить следующим образом:

тип ДвоичноеДерево = (Пусто | НепустоеДвоичноеДерево)

тип НепустоеДвоичноеДерево = (корень: T; ЛевоеПоддерево, ПравоеПоддерево: Двоичное Дерево)

Операции функционального описания для любого двоичного дерева имеют следующие свойства:

ДеревоПусто(Создание() = истина – создается пустое дерево;

ДеревоПусто(Включение(t, Создание( ))) = ложь – если в пустое дерево включается элемент, результирующее дерево не пусто;

ДеревоПусто(Построение(t, Tree, Tree)) = ложь – дерево, построенное из узла t и двух поддеревьев не пусто;

Корень(Включение(t, Создание( )) = t – корень дерева с единственным элементом – этот элемент;

ЛевоеПоддерево(Построение(t, Tree, Tree'))= Tree – левое поддерево вновь созданного дерева;

ПравоеПоддерево(Построение(t, Tree, Tree)) = Tree' – правое поддерево вновь созданного дерева;

Построение(Корень(Tree), ЛевоеПоддерево(Tree), ПравоеПод-дерево(Tree)) = Tree – формирование дерева из корня, левого и правого поддеревьев.

Двоичное дерево, каждый внутренний узел которого имеет двух потомков, называют *расширенным двоичным деревом.* Расширенное двоичное дерево, у которого все листья расположены на одном уровне, называют *полным двоичным деревом.*

Высота полного двоичного дерева с n узлами равна $[log_2 n]$ [5].

4.3 Физическое представление дерева

Дерево в памяти компьютера можно представить в виде *многосвязного списка*. В этом случае каждый узел двоичного дерева имеет вид структуры, содержащей информационное поле и два поля связей, в которых содержатся указатели на левого *(left)* и правого *(right)* узлов-потомков.

При отсутствии узла-потомка соответствующее поле связи принимает значение нулевого указателя *(nil* или *NULL*) [5].

В случае представления дерева в форме многосвязного списка указатель на начало списка должен указывать на элемент списка, являющийся корнем.

Полное двоичное дерево высоты 2 приведено на рисунке 4.2.

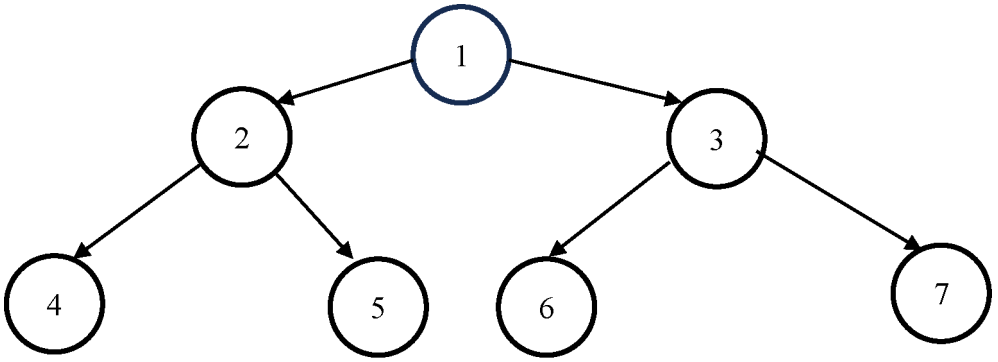


Рисунок 4.2 – Полное двоичное дерево

Возможна вырожденная в линейный список форма бинарного дерева, когда все внутренние узлы имеют ровно по одному сыну. Высота такого дерева равна n - 1, где n – число узлов в дереве. Пример вырожденного двоичного дерева приведен на рисунке 4.3.

Количество узлов в полном двоичном дереве высоты h можно определить, исходя из следующих рассуждений. На *i-м* уровне полного двоичного дерева находится 2^i узлов, поэтому количество узлов во всем дереве определяется по формуле [5]

$$n = \sum_{i=0}^{h} 2^i = 1 + \sum_{i=1}^{h} 2^i = 1 + 2\frac{2^h - 1}{2-1} = 2^{h+1} - 1. \qquad (4.2)$$

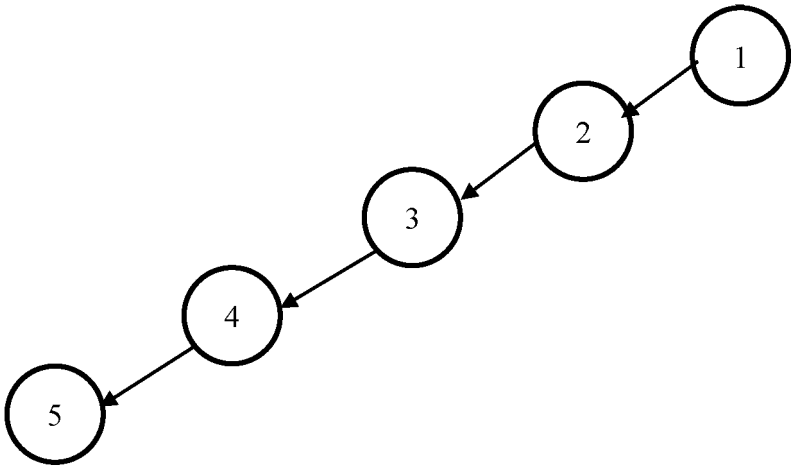


Рисунок 4.3 – Вырожденное двоичное дерево

В ряде случаев удобно представлять двоичное дерево в виде *массива*. При таком представлении если узел расположен в *i-м* элементе массива, то его левый потомок расположен в *(2i + 1)-м*, правый потомок – в *(2i + 2)-м*, а предок – в *[(i -1)/2]-м* элементах (элементы массива нумеруются с нуля).

4.4 Представление m-арного дерева бинарным деревом

Любое m-арное дерево *(m* > 2) может быть представлено бинарным деревом. Алгоритм преобразования выполняется в два этапа.

На первом этапе для каждого узла самый левый дочерний узел ставится на один уровень ниже *прямо* под узлом-отцом, остальные дочерние узлы ставятся на *этом же уровне правее первого узла*.

На втором этапе осуществляется *поворот схемы на 45 градусов по часовой стрелке*, т. е. вертикальные линии становятся левыми ветвями, а горизонтальные – правыми [4].

Схема преобразования дерева с числом связей m=4 в бинарное приводится на рисунках 4.4–4.6.

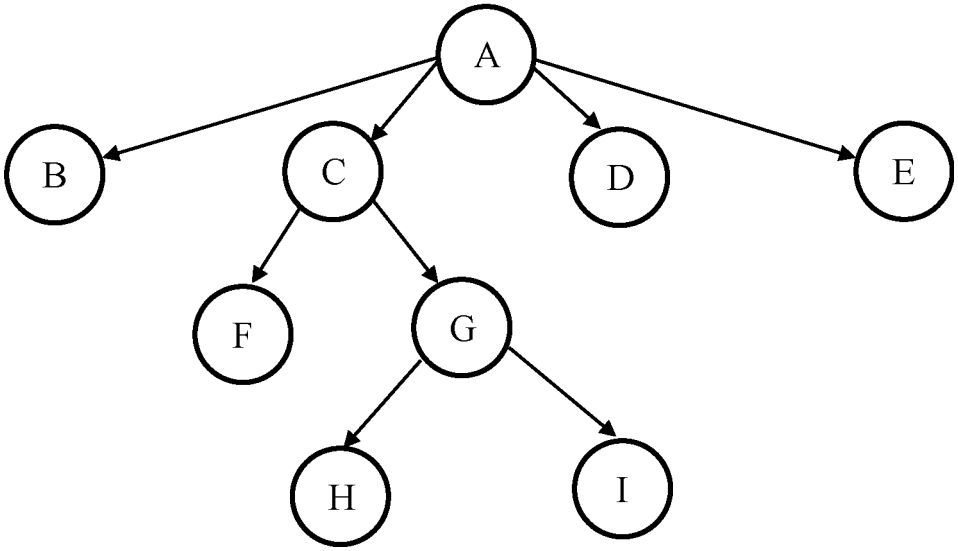


Рисунок 4.4 – Дерево с количеством связей m=4

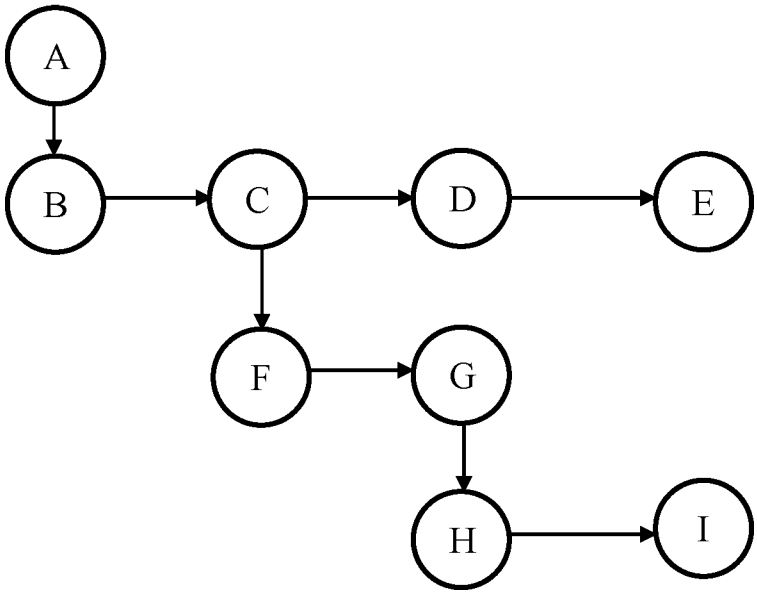


Рисунок 4.5 – Выполнение первого этапа алгоритма преобразования

4.5 Преобразование леса в бинарное дерево

Множество, состоящее из некоторого числа непересекающихся деревьев, называется *лесом.* Если удалим из дерева корень со всеми его связями с поддеревьями, получим лес, состоящий из *поддеревьев корня.* Добавив к любому лесу с однотипными элементами всего один узел и, связав его с корнями деревьев леса, получим дерево, корнем которого станет добавленный узел [4].

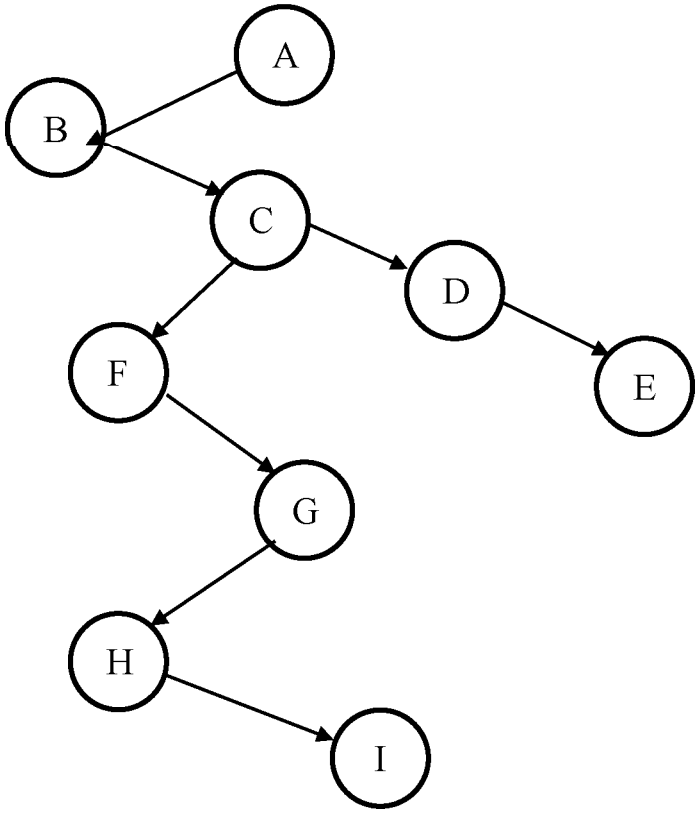


Рисунок 4.6 – Бинарное дерево

Лес может быть преобразован в бинарное дерево. Алгоритм преобразования леса в бинарное дерево схож с алгоритмом преобразования m-арного дерева:

1 Каждое дерево леса обрабатывается точно так же, как и отдельное дерево на первом этапе.

2 Корни деревьев соединяются горизонтальными линиями, и осуществляется поворот по часовой стрелке на 45 градусов относительно корня первого (левого) дерева.

Корнем леса, представленного бинарным деревом, становится корень первого дерева, второе дерево становится правым поддеревом корня первого, третье дерево – правым поддеревом корня второго дерева и т. д. Схема преобразования леса, состоящего из двух деревьев *A* и *I,* в бинарное дерево представлена на рисунках 4.7–4.9 [4].

4.6 Представление деревьев в памяти ЭВМ

Элементы древовидной структуры могут быть размещены как в последовательной (векторной) памяти, так и в динамически получаемых областях памяти. Поскольку каждый узел дерева имеет логические связи со

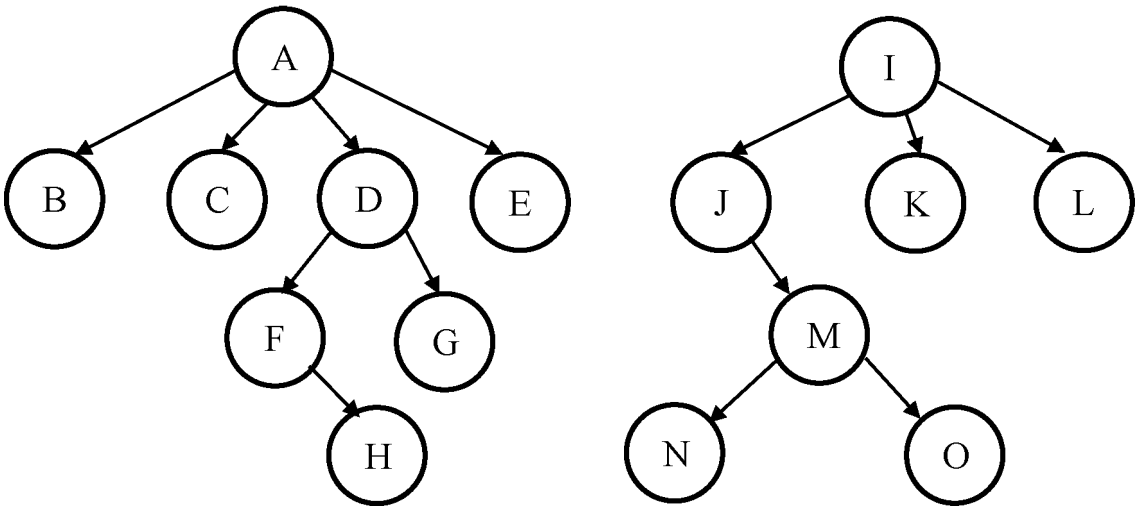


Рисунок 4.7 – Деревья A и I

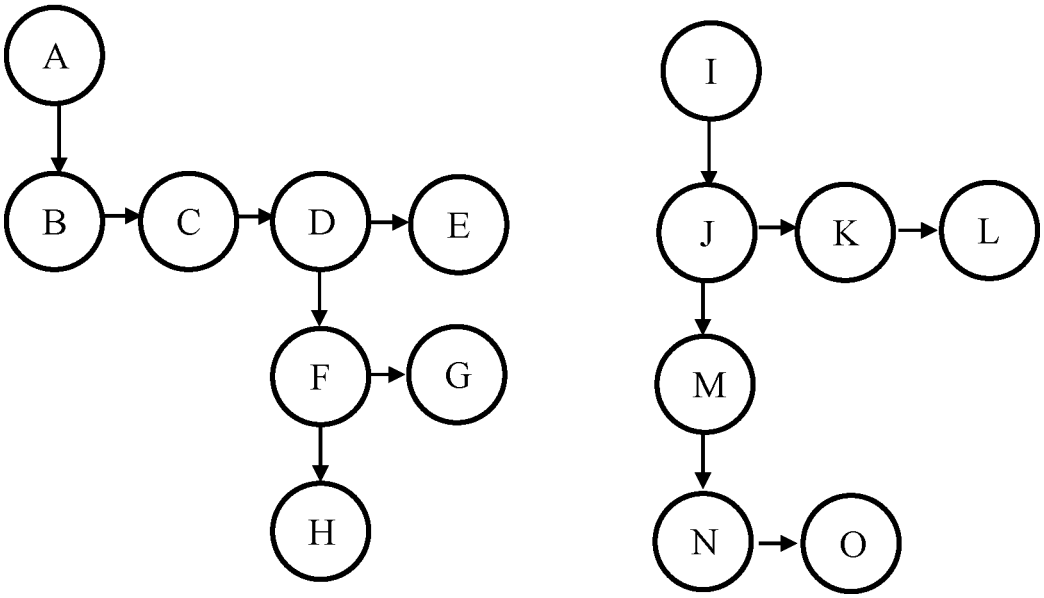


Рисунок 4.8 – Результат выполнения первого этапа алгоритма

своими дочерними узлами, эти связи должны быть отражены в физической структуре в явном или неявном виде. Рассмотрим бинарные деревья. При явном отражении связей каждый узел дерева имеет следующую структуру.

Данные – левый указатель LPTR – п*равый указатель RPTR.* Здесь поля LPTR и RPTR содержат указатели на левое и правое поддеревья данного узла. Форма указателей может быть различной.

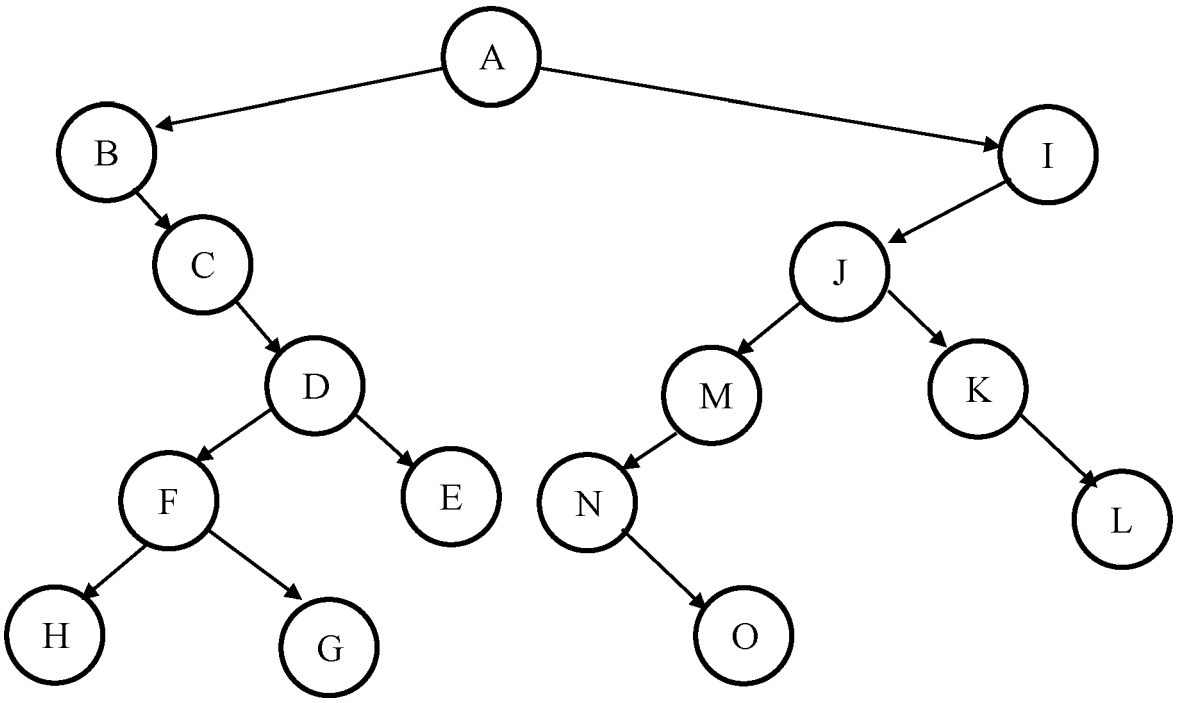


Рисунок 4.9 – Бинарное дерево

Представление дерева в векторной памяти допустимо только тогда, когда в процессе обработки объём памяти, занимаемой его элементами, не превышает фиксированного объёма векторной памяти, т. е. число элементов дерева не может превышать некоторого предельного значения. Элементы дерева занимают последовательные ячейки векторной памяти. При явном задании связей указатели LPTR и RPTR можно задавать двояко: либо в виде индексов элементов вектора, либо их адресов в памяти.

Неявное задание связей возможно только при работе с бинарными деревьями специального вида, например типа «пирамида», используемого для сортировки данных. Связи между элементами дерева-пирамиды явно не задаются, а вычисляются.

Связанное размещение элементов дерева в динамических областях памяти более удобно, обеспечивает возможность легко вставлять в дерево вершины или удалять их из него; кроме того, дерево может разрастаться до произвольного размера.

Для управления деревом необходимо наличие дескриптора или указателя на корень дерева (указателя дерева) [5].

4.7 Двоичное дерево поиска

Двоичные деревья используют для представления множества данных, среди которых происходит поиск элементов по ключу. Для решения такого рода задач предназначены двоичные деревья поиска.

Под двоичным деревом поиска (Binary Search Tree, BST) понимают двоичное дерево, для которого выполняются следующие условия:

1) для любого узла его левое и правое поддеревья являются двоичными деревьями поиска;

2) значения, размещенные во всех узлах левого поддерева произвольного узла, меньше значения, размещенного в этом узле;

3) значения, размещенные во всех узлах правого поддерева произвольного узла, не меньше значения, размещенного в этом узле.

Данные в каждом узле дерева должны быть такого типа, для которого определена операция сравнения [5].

Для двоичного дерева поиска определены все операции, присущие двоичным деревьям. Реальное время выполнения операций зависит от формы дерева. Если оно вырождено и по форме близко к линейному связному списку, поиск потребует порядка *n* операций сравнения, если по форме ближе к полному дереву, то порядка *log2n* операций. При последовательном включении в дерево случайных данных среднее время выполнения операций оценивается величиной порядка *log2n*. Если входные данные при построении дерева образуют упорядоченную последовательность, то формируется список, и все преимущества дерева будут утеряны.

4.8 Идеально сбалансированное дерево

Поскольку максимальный путь до листьев дерева определяется высотой дерева, то при заданном числе *n* узлов дерева стремятся построить дерево минимальной высоты. Этого можно достичь, если размещать максимально возможное количество узлов на всех уровнях, кроме последнего. В случае двоичного дерева это достигается таким образом, что все поступающие при построении дерева узлы распределяются поровну слева и справа от каждого узла.

Бинарное дерево *идеально сбалансировано*, если для каждого его узла количество узлов в левом и правом поддеревьях различается не более чем на 1.

Создание идеально сбалансированного дерева не вызывает затруднений. Если число узлов *n* известно и дана последовательность значений поля данных вершин *a[0], a[1],...,a[n* - 1], то можно использовать такой рекурсивный алгоритм построения идеально сбалансированного дерева.

1 Начиная с *a[0],* берем очередное значение *a[i]* в качестве значения корня дерева (поддерева).

2 Строим левое поддерево с *nl = n/2* узлами тем же способом.

3 Строим правое поддерево с *nr = n - nl* - 1 узлами.

Таким образом, значение *a[0]* окажется в корне дерева, и на него будет ссылаться указатель дерева, *a[I],.. .a[nl]*-значения попадут в левое поддерево, остальные – в правое поддерево. Следовательно, распределение значений по узлам дерева полностью определяется исходной последовательностью данных [1; 5].

4.9 Рандомизированные деревья поиска

Создание рандомизированных деревьев поиска основано на том, что «случайность» включается в алгоритм вставки элемента в дерево без каких-либо допущений относительно порядка вставки элементов. Идея заключается в том, что при вставке нового узла в дерево, состоящее из *N* узлов, вероятность появления нового узла в корне дерева должна быть равна *1/(N + 1)* [2; 5].

4.10 Оптимальные деревья поиска

Существуют достаточно редкие случаи, когда имеется информация о вероятности обращений к отдельным ключам в дереве поиска. Такие вероятности могут быть получены только на основе статистических измерений, когда ключи поиска в дереве остаются неизменными.

Пусть известны вероятности p_i обращения к *i-м* узлам с ключами k_i. Необходимо организовать бинарное дерево поиска таким образом, чтобы минимизировать математическое ожидание числа сравнений при поиске. Для этого припишем каждому узлу в определении длины пути вес, равный

вероятности обращения к этому узлу. Тогда взвешенная длина пути дерева есть сумма всех путей от корня к каждому узлу, умноженных на вероятности обращения к этому узлу:

$$P_T = \sum_{t=1}^{n} p_i * h_i \qquad (4.3)$$

Необходимо минимизировать взвешенную длину пути при данном распределении вероятностей. Узлы с большими вероятностями должны располагаться ближе к корню дерева, но тогда оптимальным. может оказаться не сбалансированное, а вырожденное дерево [5].

Такой подход учитывает только удачный поиск, т. е. поиск только существующих в дереве ключей. Если же множество ключей дерева является подмножеством ключей поиска (аргументов поиска), то наличие ключей поиска, приводящих к неудачному поиску, повлияет на структуру оптимального дерева поиска. Такая ситуация присуща, например, трансляторам, когда среди слов исходной программы отыскиваются зарезервированные слова, вероятность появления которых в программе можно установить. В таких случаях нахождение оптимальных деревьев поиска усложняется [5].

4.11 B-деревья

Для управления ростом дерева применение идеальной сбалансированности потребовало бы слишком больших затрат на балансировку, поэтому правила балансировки смягчают. Одной из распространенных структур такого типа является B-дерево. Эта структура была разработана Р. Бэйером и Э. Мак-Крейтом в 1972 г.

B-деревом порядка *n* называется сильно ветвящееся дерево степени 2n+1 (*степень узла* – это число потомков узла, а *степень дерева* – наибольшее значение степени всех узлов дерева), обладающее следующими свойствами:

1) каждый узел, за исключением корня, содержит не менее *n* и не более *2n* ключей (элементов) при заданном постоянном для дерева *n (n <= m <= 2n)*. Ключи узлов одного уровня упорядочены по возрастанию слева направо;

2) корень содержит не менее одного и не более *2n* ключей;

3) каждый узел является либо листом, т. е. не имеет потомков, либо имеет *(m* + 1) потомков, где *m* – число находящихся в узле ключей;

4) структура узла включает два массива. Первый массив из $2n$ ячеек предназначен для хранения элементов узла дерева. Второй массив из $2n + 1$ ячеек содержит $m + 1$ указателей на потомков данного узла, остальные указатели нулевые. В листовом узле массив указателей полностью свободен. Указатели деревьев на внешнем запоминающем устройстве (ВЗУ) представляют собой адреса на диске;

5) все листья дерева расположены на одном уровне.

Таким образом, в B-дереве порядка n с N элементами наихудший случай при поиске потребует $\log N$ обращений, на которые при работе с ВЗУ тратится основная часть времени. Кроме того, коэффициент использования памяти в B-дереве составляет не менее 50 %, так как страницы, содержащие узел, заполнены хотя бы наполовину [4; 5].

4.12 Контрольные вопросы

1 Какое существует определение понятия нелинейной динамической структуры данных?

2 Какое функциональное описание имеет бинарное дерево?

3 Какое логическое описание имеет бинарное дерево?

4 Какое существует определение понятия идеально сбалансированного дерева?

5 Какое существует определение понятия оптимального дерева поиска?

6 Что понимается под рандомизированным деревом поиска?

7 Что понимается под B-деревом?

8 Какие этапы включает алгоритм преобразования m-ичного дерева в двоичное дерево?

9 Какие этапы включает алгоритм преобразование леса графов в бинарное дерево?

10 Что называется вырожденным двоичным деревом?

4.13 Варианты заданий

Вариант 1

Разработать программу, которая содержит информацию о книгах в библиотеке.

Сведения о книгах содержат:

- номер УДК;
- фамилию и инициалы автора;
- название;
- год издания;
- количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- начальное формирование данных о всех книгах в библиотеке в виде двоичного дерева;
- добавление данных о книгах, вновь поступающих в библиотеку;
- удаление данных о списываемых книгах.

По запросу выдаются сведения о наличии книг в библиотеке, упорядоченные по годам издания [6; 7].

Вариант 2

Разработать программу, которая содержит информацию о заявках на авиабилеты.

Каждая заявка содержит:

- пункт назначения;
- номер рейса;
- фамилию и инициалы пассажира;
- желаемую дату вылета.

Программа должна обеспечивать:

- хранение всех заявок в виде двоичного дерева;
- добавление и удаление заявок;
- по заданному номеру рейса и дате вылета вывод заявок с их последующим удалением;
- вывод всех заявок.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе [6; 7].

Вариант 3

Англо-русский словарь построен как двоичное дерево. Каждый компонент содержит английское слово, соответствующее ему русское слово и счетчик количества обращений к данному компоненту.

Первоначально дерево формируется согласно английскому алфавиту. В процессе эксплуатации словаря при каждом обращении к компоненту в счетчик обращений добавляется единица.

Разработать программу, которая:

- обеспечивает начальный ввод словаря с конкретными значениями счетчиков обращений;
- формирует новое представление словаря в виде двоичного дерева по следующему алгоритму:

1) в старом словаре ищется компонент с наибольшим значением счетчика обращений;

2) найденный компонент заносится в новый словарь и удаляется из старого;

3) переход к этапу 1 до исчерпания исходного словаря;

- обеспечивает вывод исходного и нового словарей.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе [6; 7].

Вариант 4

На междугородной телефонной станции картотека абонентов, содержащая сведения о телефонах и их владельцах, организована как двоичное дерево. Разработать программу, которая:

- обеспечивает начальное формирование картотеки в виде двоичного дерева;
- производит вывод всей картотеки;
- вводит номер телефона и время разговора;
- выводит извещение на оплату телефонного разговора.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе [6; 7].

Вариант 5

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указываются:

- номер поезда;
- станция назначения;
- время отправления;
- время прибытия.

Данные в информационной системе организованы в виде двоичного дерева. Разработать программу, которая:

- обеспечивает ввод данных о поезде дальнего следования;
- обеспечивает удаление данных о поезде дальнего следования;
- выполняет поиск данных о поезде дальнего следования;
- производит вывод всего дерева [6; 7].

Вариант 6

Разработать программу, которая содержит информацию об автомобилях в автосалоне.

Сведения об автомобилях содержат:

- модель автомобиля;
- характеристики (тип кузова, количество дверей/мест, длина/ширина/высота, колесная база (мм), снаряженная масса автомобиля (кг), полная масса автомобиля (кг));
- характеристики двигателя (тип двигателя, рабочий объем, степень сжатия, максимальная мощность (л.с.), максимальный крутящий момент (Нм/об.мин));
- скоростные характеристики (максимальная скорость (км/ч), средний расход топлива (л/100 км), тип потребляемого бензина);
- количество автомобилей данной модели в автосалоне.

Программа должна обеспечивать:

- начальное формирование данных о всех автомобилях в автосалоне в виде двоичного дерева;

- добавление данных об автомобилях, вновь поступающих в автосалон;

- удаление данных о проданных автомобилях;

- вывод данных по всем автомобилям автосалона.

По запросу выдаются сведения о наличии автомобиля в автосалоне, упорядоченные по наименованию модели автомобилей [6; 7].

Вариант 7

Разработать программу, которая содержит информацию о заявлениях на приобретение туристических путевок.

Каждое заявление содержит:

- название страны посещения;

- время года;

- фамилию и инициалы отдыхающего;

- желаемую дату отправления;

- сроки пребывания;

- уровень обслуживания.

Программа должна обеспечивать:

- хранение всех заявлений в виде двоичного дерева;

- добавление и удаление заявлений;

- вывод заявлений по фамилии и инициалам отдыхающих с их последующим удалением;

- вывод всех заявок.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 8

Разработать программу, которая содержит информацию об абонентах кабельного телевидения.

Сведения об абонентах содержат:

- фамилию и инициалы;

- адрес;

- телефон;

- номер договора.

Программа должна обеспечивать:

- начальное формирование данных обо всех абонентах кабельного телевидения в виде двоичного дерева;

- добавление данных об абонентах, вновь поступающих в картотеку;

- удаление данных об абонентах из картотеки, разорвавших договор;

- по запросу выдавать сведения о наличии абонентов в картотеке, упорядоченные по номеру договора.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 9

Разработать программу, которая содержит информацию о безработных, зарегистрированных на бирже труда.

Данные о зарегистрированных безработных содержат:

- номер регистрации безработного;

- фамилия, имя, отчество;

- возраст;

- пол;

- образование;

- профессия;

- общий стаж работы;

- дата постановки на учет;

- желаемая заработная плата;

Программа должна обеспечивать:

- хранение всех зарегистрированных безработных в виде двоичного дерева;

- добавление данных о безработных;

- удаление данных о безработных, нашедших работу;

- вывод данных о безработных по фамилии, имени, отчеству, регистрационному номеру;

- вывод всех зарегистрированных безработных.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе [6; 7].

Вариант 10

Разработать программу, которая содержит информацию о дилерах компании.

Сведения о дилерах содержат:

- адрес;
- фамилия, имя, отчество;
- телефон;
- электронный адрес;
- объем продаж продукции за месяц.

Программа должна обеспечивать:

- начальное формирование данных обо всех дилерах фирмы в виде двоичного дерева;
- добавление данных о дилерах;
- удаление данных о дилерах;
- сведения о дилерах по фамилии, имени, отчеству.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе [6; 7].

5 СОРТИРОВКА ДАННЫХ

5.1 Основные понятия и определения

Сортировка – процесс упорядочивания данных в порядке возрастания или убывания значений.

Запись – последовательность из *n* элементов *r(l), r(2), ..., r(n).* С каждой записью связан *ключ k(i),* являющийся частью записи.

Последовательность *отсортирована по ключу,* если для всех $1 \leq i < j \leq n$ ключ *k(i)* предшествует ключу *k(j)* относительно некоторого порядка, заданного на множестве значений ключей (далее будем рассматривать лишь случай, когда этот порядок линейный).

Файл – последовательность записей как объект обработки (сортировки).

В записях файла содержатся ключи, поэтому рассматриваются действия над *k(i).*

Алгоритм сортировки – алгоритм, реализующий метод упорядочивания данных.

Алгоритм сортировки *называется устойчивым,* если порядок следования одинаковых ключей в файле не меняется после сортировки.

Для некоторых приложений обеспечение устойчивости алгоритма сортировки является необходимым. Это актуально при сортировке по новому параметру данных, упорядоченных предварительно по другим параметрам. Например, библиотечный каталог, упорядоченный по названиям произведений, необходимо упорядочить по фамилиям авторов книг. Если при этом использовать устойчивый алгоритм сортировки, то в общем каталоге список трудов каждого автора останется упорядоченным по названиям произведений [5].

5.2 Классификация методов упорядочивания данных

Классификация алгоритмов сортировки данных приведена на рисунке 5.1

Алгоритмы сортировки можно разбить на две группы: *алгоритмы внутренней сортировки* (когда сортируемые записи расположены в основной памяти компьютера) и *внешней сортировки* (когда сортируемые записи расположены во внешней памяти). Основное отличие между двумя типами сортировок заключается в том, что в условиях внутренней сортировки доступ к любому элементу не представляет трудностей, в то время как в условиях внешней сортировки предпочтителен только последовательный тип доступа или по меньшей мере доступ к блокам больших размеров.

Рисунок 5.1 – Классификация алгоритмов сортировки данных

Известно очень большое количество алгоритмов внутренней сортировки. Каждый из них можно отнести к одному из двух классов:

•алгоритмы, основанные на сравнении ключей и перестановке некоторых записей *(сортировка сравнениями);*

•алгоритмы, основанные на распределении записей по упорядоченным группам и последующем объединении содержимого этих групп *(распределяющая сортировка).*

Алгоритмы сортировки сравнениями, в свою очередь, могут быть разбиты на четыре основных подкласса:

- сортировка вставками;
- сортировка выбором;
- обменная сортировка;
- сортировка слиянием.

Основная идея *сортировки вставками* заключается в том, что на очередном этапе алгоритма соответствующая запись вставляется в надлежащее место среди отсортированных ранее записей [5].

При сортировке *выбором* на очередном этапе алгоритма из еще не отсортированных записей выбирается та, которая имеет максимальный (или минимальный) ключ, и добавляется к выбранным ранее.

Обменная сортировка заключается в многократном выполнении одной и той же процедуры: две записи, ключи которых не удовлетворяют требуемому порядку, меняются местами.

При сортировке *слиянием* файл делится на части, которые сортируются по отдельности и затем сливаются в один файл.

Наиболее важной характеристикой алгоритмов сортировки является время их выполнения. Время работы (трудоемкость или *вычислительная сложность)* алгоритма сортировки *n* элементов определяется числом выполняемых операций. При изучении и анализе алгоритмов важно знать *минимальную Tmin(n), максимальную Tmax(n)* и *среднюю Tcp(n) вычислительные сложности* работы алгоритма [5].

Вычислительная сложность алгоритмов сортировки сравнениями не зависит от значений ключей, а лишь от соотношений между ними. Например, любой алгоритм сортировки сравнениями при работе с последовательностями 5, 1, 7, 6 и 102, 2, 320, 200 произведет одинаковое число операций.

Другой важной характеристикой алгоритма сортировки является *емкостная сложность* – необходимый для работы алгоритма объем памяти. Несмотря на высокий уровень развития вычислительной техники в настоящее время (с достаточно большими объемами доступных ресурсов памяти),

емкостную сложность алгоритмов требуется учитывать при создании, анализе и сравнении алгоритмов [5].

5.3 Метод простой вставки

Метод простой вставки – метод сортировки данных, упорядочивающий последовательность постепенным включением элементов из исходного массива. Этапы алгоритма метода простой вставки:

Этап 1. Первый элемент массива считается находящимся в отсортированной последовательности.

Этап 2. Следующий элемент вставляется в правильную позицию в отсортированной части массива, сдвигая при необходимости элементы, которые больше выбранного элемента.

Этап 3. Переход на второй этап, пока не будет достигнут конец массива.

Метод простой вставки имеет квадратическую оценку сложности $O(n^2)$, является устойчивым методом, сохраняющим относительный порядок элементов с одинаковыми значениями элементов, и может быть эффективным для небольших или частично отсортированных массивов [4; 8].

5.4 Метод простого обмена

Метод простого обмена – метод сортировки данных, основанный на переборе элементов массива с проверкой условия больше (меньше) значения предшествующего элемента с последующим обменом значениями рядом стоящими элементами в случае невыполнения условия.

Алгоритм метода простого обмена включает этапы:

Этап 1. Перебор элементов массива с проверкой значений больше (меньше) рядом стоящих элементов. В конце массива будет находится элемент с максимальным (минимальным) значением элемента.

Этап 2. Последний элемент массива исключается из рассмотрения. Размер массива уменьшается на единицу. Переход на первый этап.

Итерационный процесс выполняется до тех пор, пока все элементы не отсортируются.

Оценка функции сложности метода простого обмена квадратическая $O(n^2)$ в худшем случае [6; 7].

5.5 Метод простого выбора

Метод простого выбора – метод сортировки данных, который на каждой итерации находит минимальный или максимальный элемент в неотсортированной части массива и перемещает его в начало или в конец отсортированной части. Алгоритм метода простого обмена включает следующие этапы:

Этап 1. Массив делится на две части: отсортированную и неотсортированную.

Этап 2. На каждой итерации находится минимальный или максимальный элемент в неотсортированной части и меняется местами с первым элементом неотсортированной части.

Этап 3. Размер отсортированной части увеличивается на один элемент, а размер неотсортированной части уменьшается на один элемент.

Этап 4. Этапы 2 и 3 повторяются до тех пор, пока неотсортированная часть массива полностью не исчерпается.

Метод простого выбора имеет квадратическую оценку функции сложности $O(n^2)$, является неустойчивой сортировкой, не сохраняет относительный порядок элементов с одинаковыми значениями [6].

5.6 Метод Шелла

Метод Шелла является улучшением *метода сортировки прямой вставки*. Основан на сортировке посредством вставки с уменьшающимися расстояниями.

Алгоритм сортировки включает следующие этапы:

1 Используется несколько проходов.

2 На первом проходе отдельно группируются и сортируются вставками элементы, отстоящие друг от друга на *d=size/2* позиций.

3 На втором проходе группируются и сортируются вставками элементы, отстоящие друг от друга на *d=d-1* позиций.

4 Аналогично выполняются следующие проходы. Сортировка заканчивается последним проходом при *d=1* (как при простой сортировке вставками) [6; 8].

5.7 Метод Хоара

Метод Хоара является усовершенствованием прямого метода обмена.

Алгоритм сортировки включает этапы:

1 Исходный массив разбивается на два сегмента – левый, элементы которого *arr[i].key<=median.key*, и правый сегмент с элементами *arr[j].key>=median.key*.

2 Каждый из полученных сегментов сортируется автономно, аналогично предыдущему, до получения сегментов единичной длины [6; 8].

5.8 Метод пирамиды

Пирамидальный метод сортировки является совершенствованием метода сортировки прямого выбора. Основан на повторяющихся поисках наименьшего элемента среди s элементов, затем среди оставшихся (s-1) элементов и т. д.

Разработан Вильямсом в 1964 г. Сортировка производится с помощью *дерева*. Дерево имеет структуру пирамиды. Пирамида состоит из помеченного двоичного дерева высоты h и обладает свойствами:

1 Конечная вершина имеет высоту h или $h-1$.

2 Конечная вершина высоты h находится слева от любой конечной вершины высотой $h-1$.

3 Ключ любой вершины больше ключа следующей за ней вершины (потомка). Ключ корня является наибольшим ключом дерева.

Алгоритм сортировки включает следующие этапы:

1 Исходный массив преобразуется в структуру пирамиды.

2 Упорядочивание элементов массива.

Используя свойства 1 и 2, массив из s элементов представляется в виде двоичного дерева, которое преобразуется в пирамиду.

Для любой вершины дерева a_i потомками являются вершины a_j и $a_{j+1}, j = 2 * i$. Последняя нелистовая вершина образуется из элемента a_k, $k = s \ mod \ 2$.

Пирамида строится с учетом того, что ключ вершины пирамиды больше ключей потомков (свойство 3). Начиная с последней нелистовой вершины $i = s \ mod \ 2$, проверяем, выполняется ли это свойство. Если

нарушено, то ключи вершины и потомка меняем местами. Это делаем с вершиной $i = i - 1$ и т. д., пока не достигнем корня дерева [4; 8].

5.9 Генератор случайных чисел

Случайные числа в языке программирования С++ могут быть сгенерированы функцией *rand( )* из стандартной библиотеки С++. Функция *rand()* генерирует числа в диапазоне от *0* до *RAND_MAX*.

RAND_MAX – это константа, определённая в библиотеке <*cstdlib*>. Для *MVS RAND_MAX = 32767*. При генерировании случайных чисел в программе выполняются [9]:

1) автоматизация рандомизации.

2) масштабирование диапазона случайных чисел.

3) задание вероятностного закона распределения случайных чисел.

Функция *srand()* выполняет рандомизацию. Например, *srand(time(NULL));*. Для масштабирования диапазона случайных чисел генератором случайных чисел *(ГСЧ)* используется выражение *a+rand()%b*, где *a* – начальное значение диапазона случайных чисел, *b* – конечное значение диапазона случайных чисел. Например:

```
//Генерация случайных чисел в диапазоне [0; 15]
int c = 1 + rand() % 15;.
```

Программа, реализующая получение последовательности случайных чисел, равномерно распределенных на отрезке *[0;1]*, приведена в листинге 5.1.

Листинг 5.1. Реализация равномерного распределения случайных чисел в диапазоне [0; 1].

```
// Program1.cpp : Случайные числа, равномерно распределённые
// в диапазоне [0; 1].
#include <iostream>
#include <ctime>
#include <iomanip>
using namespace std;
double get_uniform();
int main()
{ setlocale(LC_ALL, "russian");
   int i;
   const int n = 1000; //Длина случайной последовательности
   double *mas=new double[n];
```

```
    srand(time(NULL));
    for (i = 0; i < n; i++)
        mas[i] = get_uniform();
    cout.setf(ios::fixed);
    for (i = 0; i < n; i++)
    cout << "Элемент массива mas["<<i<<"]->"<<setw(5)<<
setprecision(3)<<mas[i]<<endl;
    cout.unsetf(ios::fixed); delete[] mas;
    return 0;
}
double get_uniform() {
    int number = rand();
    double result = (double)number/(RAND_MAX+1);
    return result;
}
```

Результат работы программы, генерирующей случайные числа, равномерно распределенные в диапазоне *[0; 1]*, изображен на рисунке 5.2.

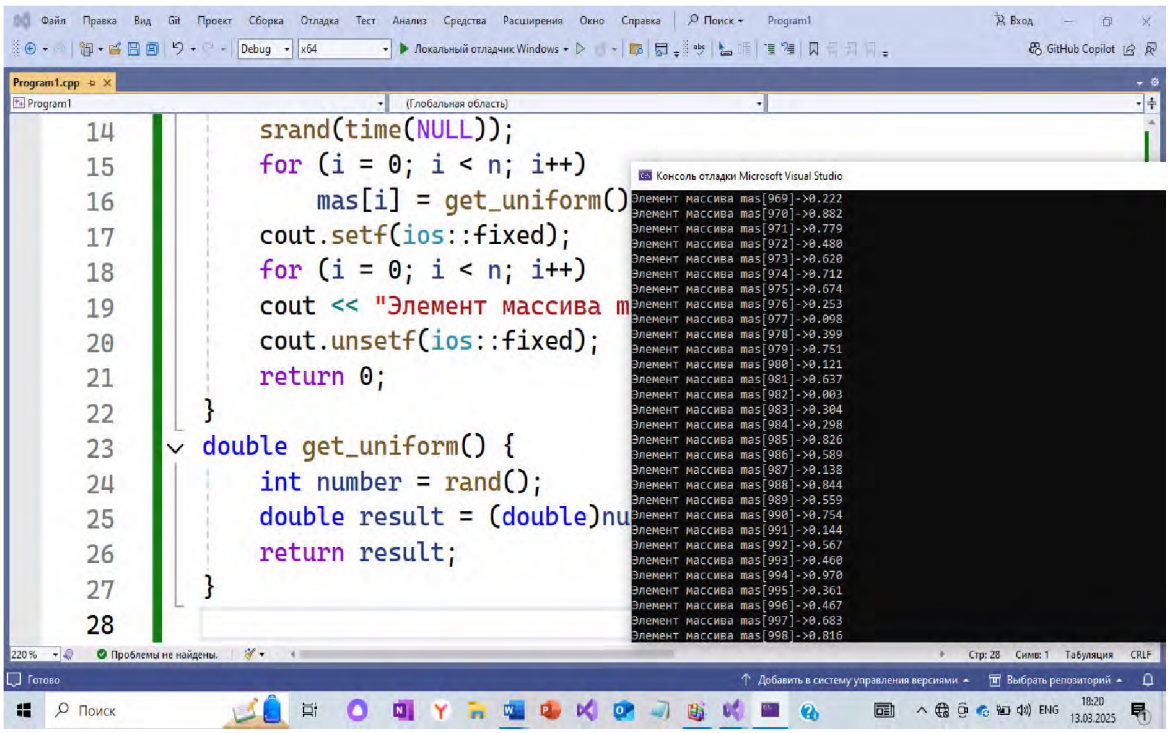


Рисунок – 5.2 Случайные числа, равномерно распределенные в диапазоне [0; 1]

Вероятностный закон распределения случайных чисел определяется обратной функцией.

В таблице 5.1 перечислены вероятностные распределения, для которых обратная функция $F^{-1}(u)$ является элементарной [6; 9].

Таблица 5.1 – Распределение вероятностей с элементарной функцией

Название	Функция *F(x)*	Функция $F^{-1}(u)$
Bradford	$\frac{ln(1+\frac{C*(x-A)}{B-A})}{ln(C+1)}$, $A<x<B, C>0$	$A+\frac{(B-A)*((C+1)^u-1)}{C}$
Burr	$(1+(\frac{x-A}{B})^{-C})^{-D-1}$, $y>A, B>0, C>0, D\leq 100$	$A+B*(u^{\frac{-1}{D+1}}-1)^{\frac{-1}{C}}$
Cauchy	$\frac{1}{2}+\frac{1}{\pi}*\frac{1}{tg(\frac{x-A}{B})}, B>0$	$A+B*arctg(\frac{2}{\pi*(2*u-1)})$
Exponential	$1-e^{\frac{A-x}{B}}, x\geq A, B>0$	$A-B*ln(1-u)$
ExtremeLB	$exp(-(\frac{x-A}{B})^{-C})$, $x>A, B>0, 0<C\leq 100$	$A+B*(-\ln u)^{\frac{-1}{C}}$
Fisk	$\frac{1}{1+(\frac{x-A}{B})^{-C}}$, $x>A, B>0, 0<C\leq 100$	$A+B*(\frac{1}{u}-1)^{\frac{-1}{C}}$
Gumbel	$exp(-exp(\frac{A-x}{B}))$, $B>0$	$A-B*ln(-\ln u)$
Laplace	$\begin{cases}\frac{1}{2}*exp(\frac{x-A}{B}), x\leq A;\\ 1-\frac{1}{2}*exp(\frac{A-x}{B}), x>A\end{cases}$	$\begin{cases}A+B*ln(2*u), u\leq 0.5;\\ A-B*ln(2*(1-u)), u>0.5\end{cases}$
Logistic	$\frac{1}{1+exp(\frac{A-x}{B})}, B>0$	$A-B*ln(\frac{1}{u}-1)$
Pareto	$1-(\frac{A}{x})^B, 0<A\leq x, B>0$	$A*(1-u)^{\frac{-1}{B}}$
Reciprocal	$ln(\frac{A}{x})/ln(\frac{A}{B})$, $0<A\leq x\leq B$	B^u*A^{1-u}
Weibull	$1-exp(-(\frac{x-A}{B})^C)$, $x>A, B>0, C>0$	$A+B*(-ln(1-u))^{\frac{1}{C}}$

Программа, реализующая получение последовательности случайных чисел, распределенных по вероятностному закону Парето, приведена в листинге 5.2.

Листинг 5.2. Программное приложение генерации случайных чисел.

```
// Program2.cpp : Получение последовательности случайных чисел,
// распределенных по вероятностному закону Парето
#include <iostream>
#include <fstream>
#include <ctime>
#include <cmath>
#include<iomanip>
using namespace std;
double Pareto(double, double); //Датчик случайных чисел
int main()
{
   int i, n;
   double A, B;
   setlocale(LC_ALL, "russian");
   ofstream f("D:\\DATA\\data.txt"); /* Описание потока
   и открытие файла для записи */
   cout << "Введите количество элементов n="; cin >> n; cout << endl;
   cout << "Введите параметр A="; cin >> A; cout << endl;
   cout << "Введите параметр B="; cin >> B; cout << endl;
   //Выделение памяти под массив из n вещественных элементов
   double* ptr = new double[n];
   cout.setf(ios::showpos | ios::fixed | ios::right);
   for (i = 0; i < n; i++) {
      ptr[i] = Pareto(A, B);
      cout << "Элемент массив X[" << i << "]=" << setw(10) << setpreci-
sion(6) << *(ptr + i) << endl;
      f << ptr[i] << '\n';
   }
   f.close(); delete[] ptr; return 0;
}
//Датчик случайных чисел, подчиняющихся вероятностному
// закону распределения Pareto
double Pareto(double A1, double B1) {
   int num;
   double root, right;
   num = rand();   //Случайное целое число
   right = ((double)num / double(RAND_MAX + 1));/* Проекция на интервал
(0, 1) */
   root = A1 / pow(1 - right, 1.0 / B1);   //Расчет обратной функции
   return root;}
```

Результат работы программы, генерирующей случайные числа, подчиняющиеся вероятностному закону Парето, изображен на рисунке 5.3.

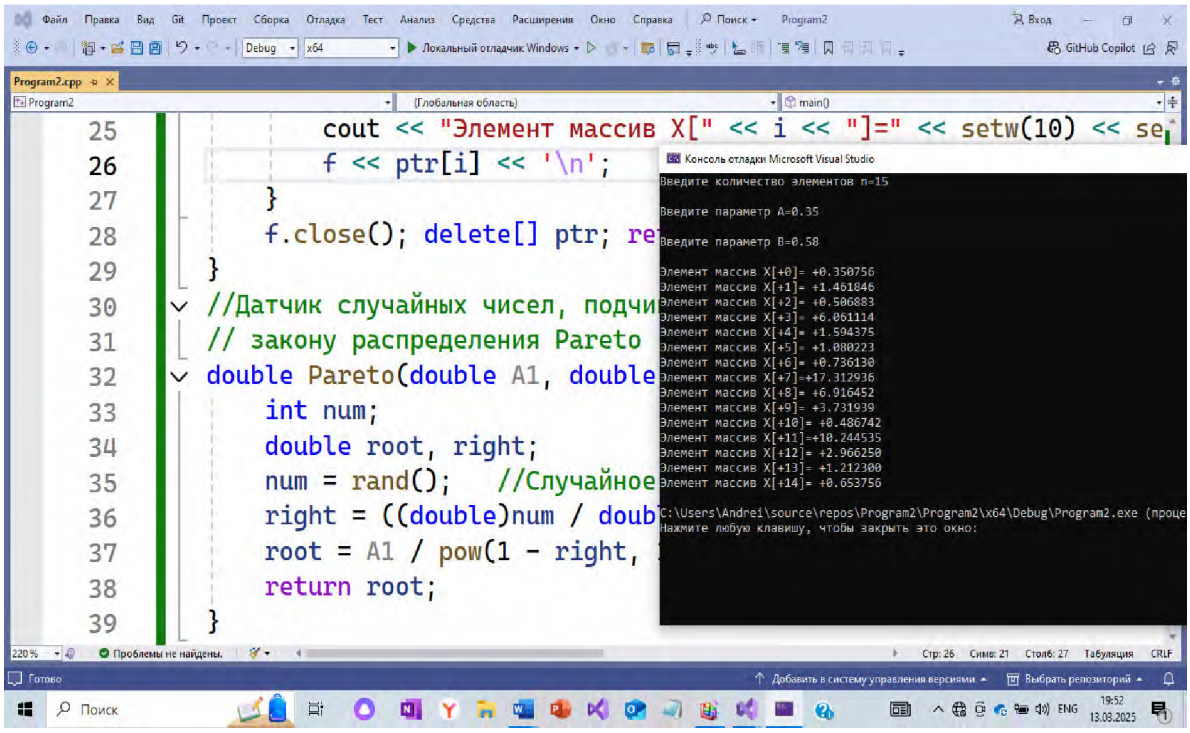


Рисунок 5.3 – Случайные числа, подчиняющиеся вероятностному закону распределения Парето

5.10 Пример выполнения варианта задания на ПЭВМ

Разработать визуальное приложение на языке Visual C++, которое выполняет:

1 Генерацию массива числовых данных размера n=5000 со случайным распределением значений элементов массива, подчиняющихся вероятностному закону распределения Weibull.

2 Сортировку по возрастанию исходного массива простыми и сложными методами упорядочивания данных (по 3 каждого вида).

3 Сравнительный анализ методов сортировки элементов массива, определив время сортировки, количество сравнений, количество перестановок. Результаты анализа представить в табличной форме записи.

Результаты выполнения анализа методов упорядочивания элементов исходного массива представить в табличной форме записи.

Разработка визуального приложения включает этапы:

1 Запуск интегрированной среды программирования Microsoft Visual Studio 2022 Community.

2 Создание проекта однооконного программного приложения.

3 Настройка свойств формы.

4 Добавление компонентов в рабочую область диалогового окна.

5 Настройка свойств компонентов.

6 Создание пустых функций обработчиков событий.

7 Определение функций обработчиков событий.

8 Отладка программного приложения.

9 Проверка правильности результатов обработки данных программным приложением.

Скриншоты выполнения программного приложения представлены на рисунках 5.4–5.11.

Эффективным методом сортировки данных является метод Хоара, имеющий наименьшее время упорядочивания элементов массива и количество перестановок.

Листинг программной реализации на языке Visual C++ приведён в приложении А.

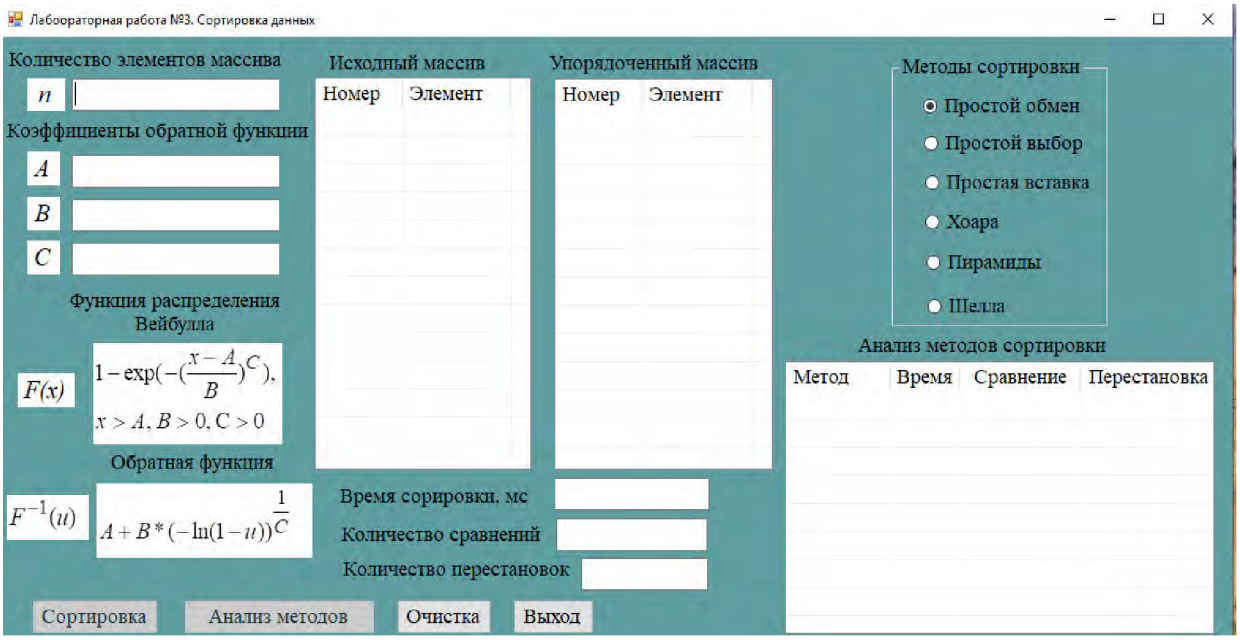


Рисунок 5.4 – Вид диалогового окна после запуска программы

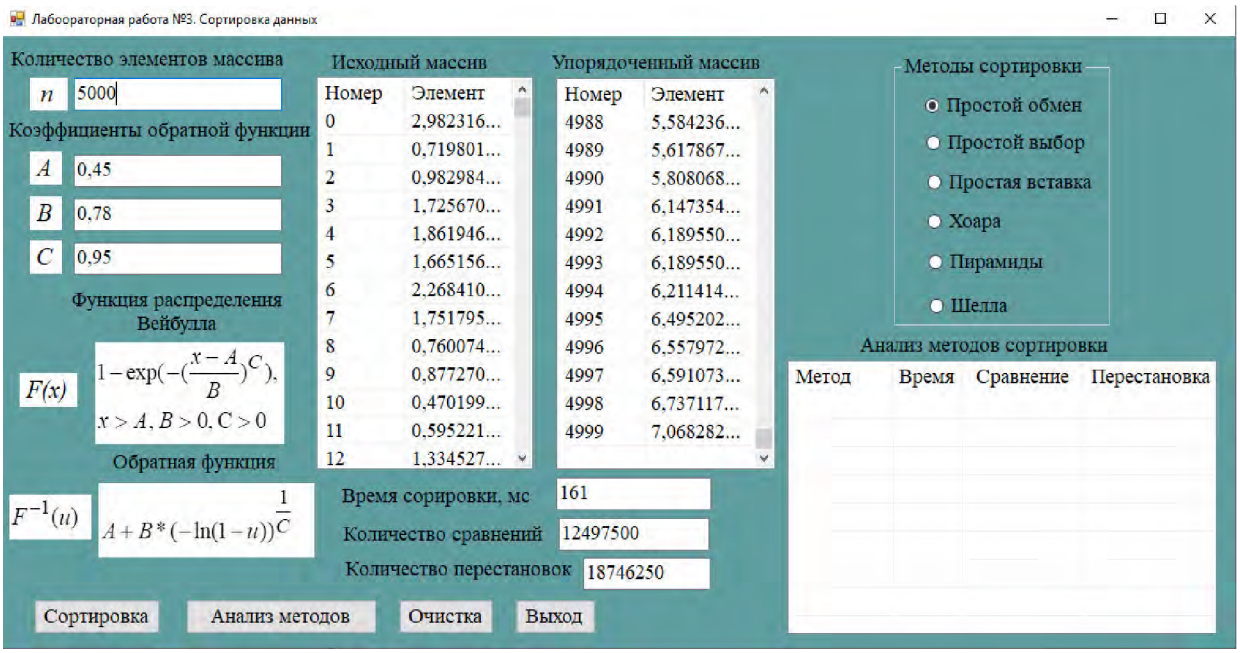


Рисунок 5.5 – Результат выполнения сортировки исходного массива методом простого обмена

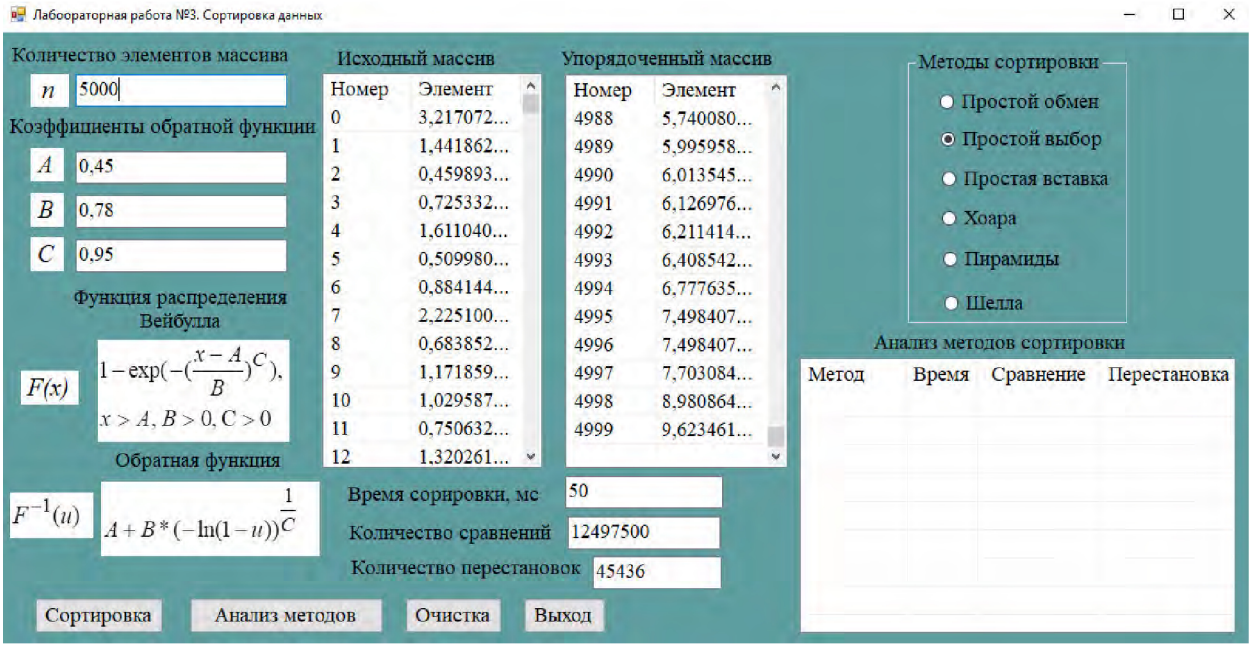


Рисунок 5.6 – Результат выполнения сортировки исходного массива методом простого выбора

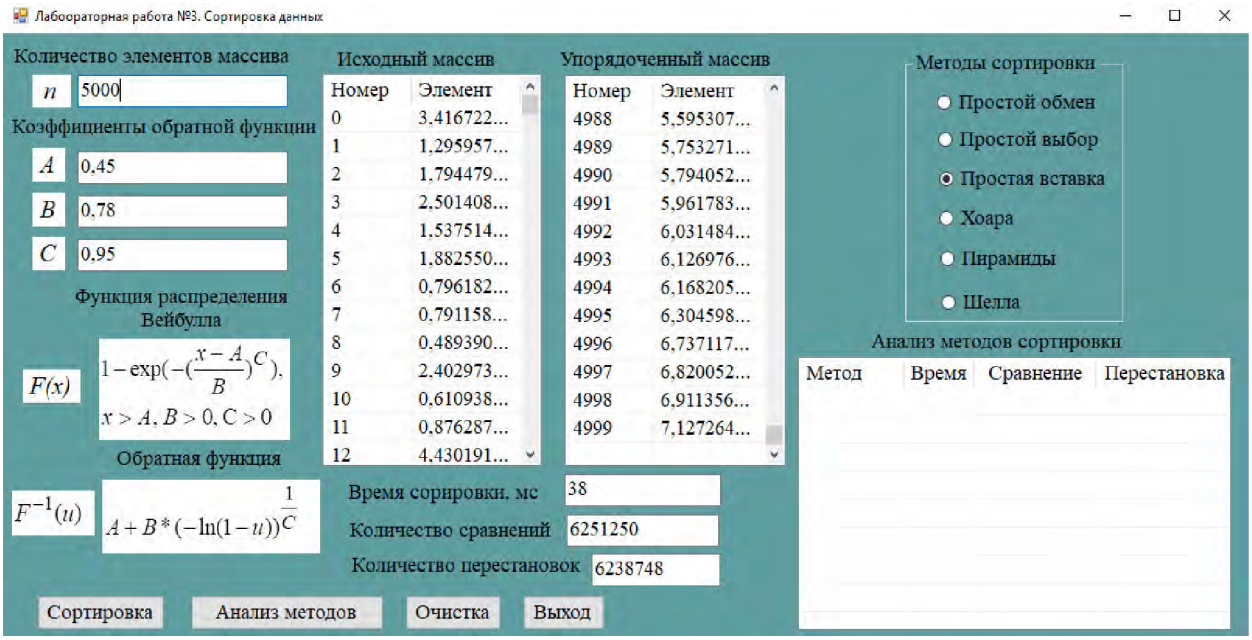


Рисунок 5.7 – Результат выполнения сортировки исходного массива методом простой вставки

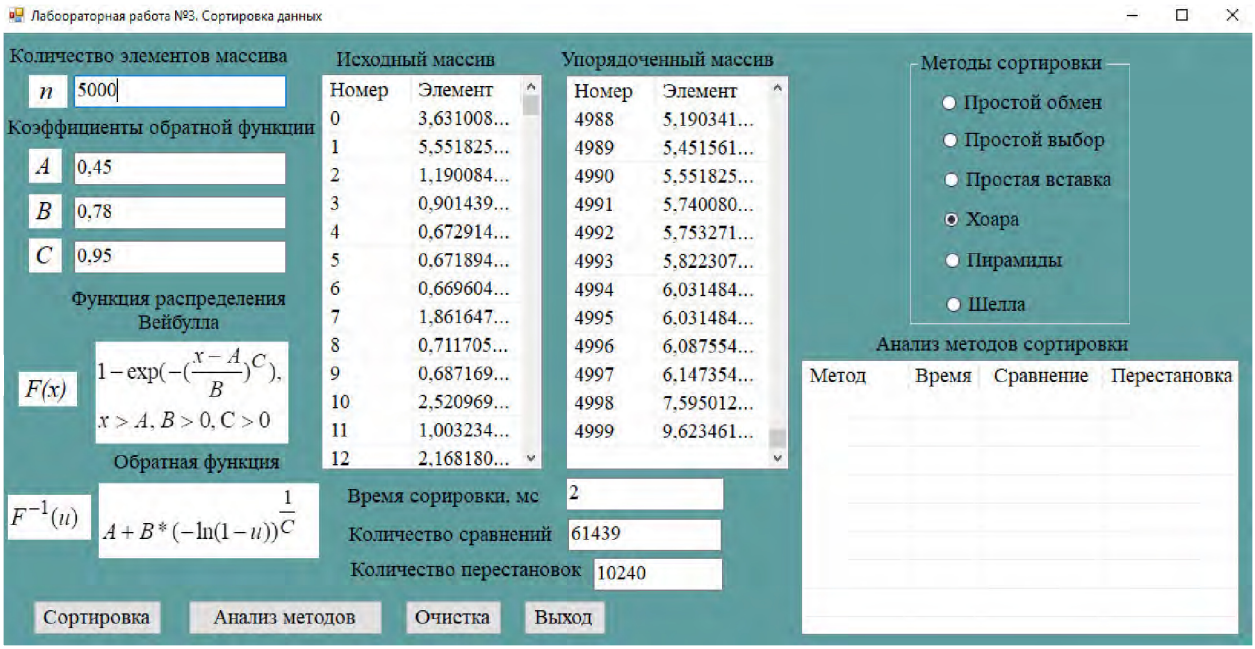


Рисунок 5.8 – Результат выполнения сортировки исходного массива методом Хоара

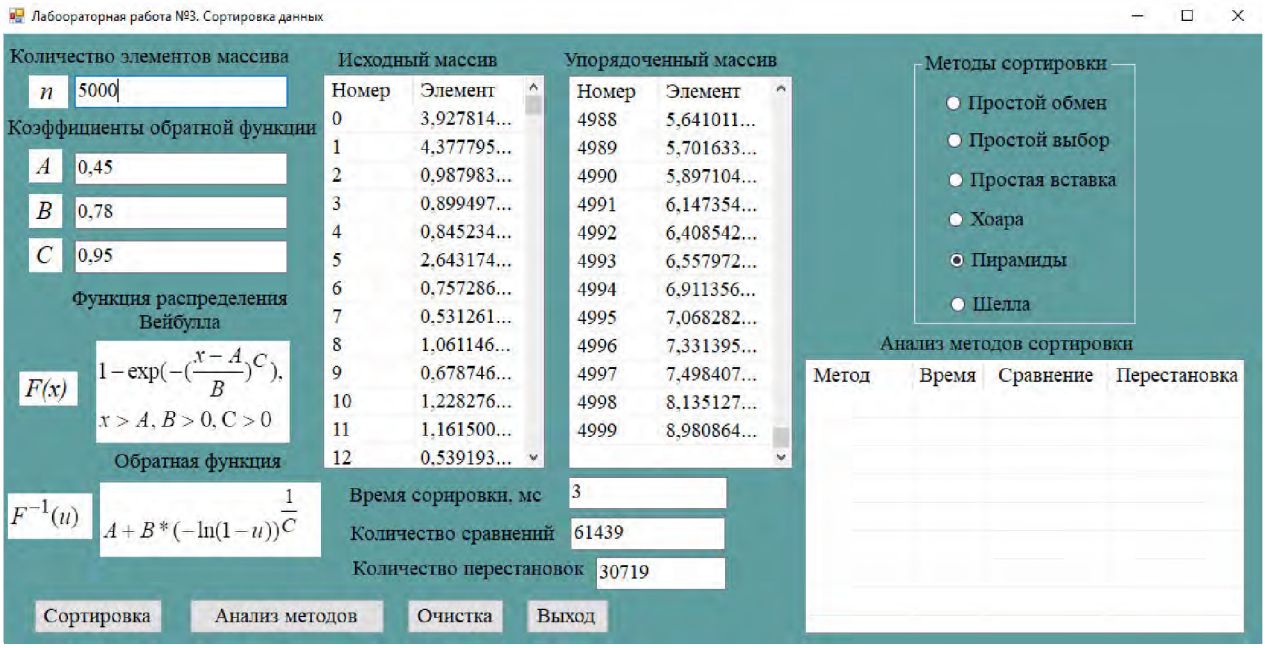


Рисунок 5.9 – Результат выполнения сортировки исходного массива методом пирамиды

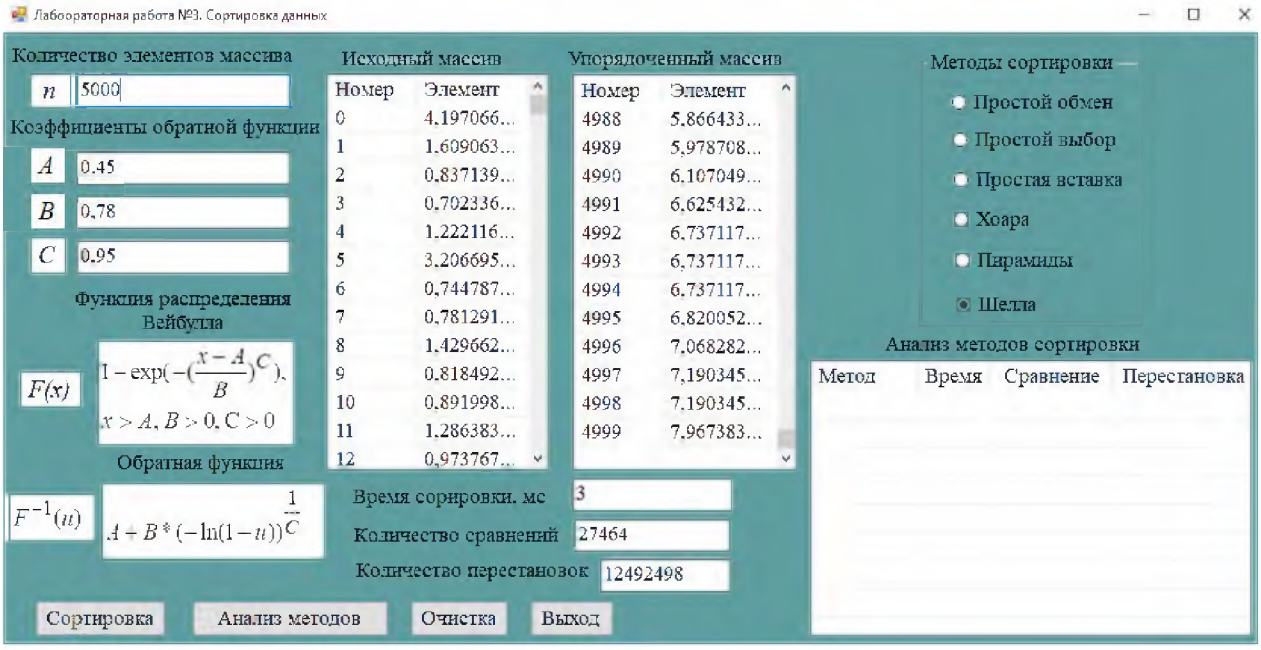


Рисунок 5.10 – Результат выполнения сортировки исходного массива методом Шелла

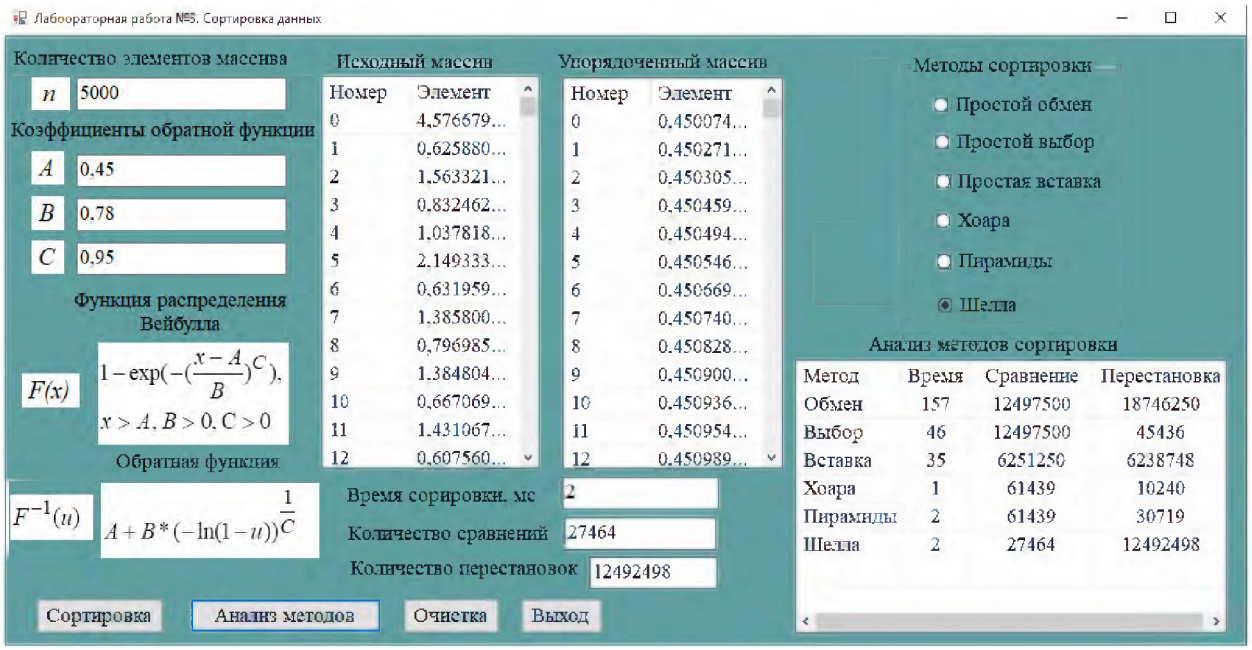


Рисунок 5.11 – Анализ методов сортировки

5.11 Контрольные вопросы

1 Какое существует определение понятия «сортировка данных»?

2 Какие этапы имеет алгоритм метода Хоара?

3 Какие этапы имеет алгоритм метода Шелла?

4 Какие этапы имеет алгоритм метода пирамиды?

5 Какие этапы имеет алгоритм метода простого обмена?

6 Какие этапы имеет алгоритм метода простого выбора?

7 Какие этапы имеет алгоритм метода простой вставки?

8 Какой вид имеет обратная функция Pareto?

9 Какие этапы включает алгоритм разработки генератора случайных чисел с использованием обратной функции?

10 Какой вид имеет обратная функция Cauchy?

5.12 Варианты заданий

Разработать визуальное приложение на языке Visual C++, которое выполняет:

1 Генерацию массива числовых данных размера N со случайным распределением значений элементов массива, подчиняющихся заданному вероятностному закону распределения.

2 Сортировку по возрастанию исходного массива простыми и сложными методами упорядочивания данных (по 3 каждого вида).

3 Сравнительный анализ методов сортировки элементов массива, определив время сортировки, количество сравнений, количество перестановок. Результаты анализа представить в табличной форме записи.

Результаты выполнения анализа методов упорядочивания элементов исходного массива представить в табличной форме записи [6; 7].

Количество элементов в массиве N и вероятностный закон распределения случайных величин приведены по вариантам в таблице 5.2.

Таблица 5.2 – Исходные данные вариантов заданий

Вариант	Количество элементов в массиве n	Вероятностный закон распределения случайной величины
1	15000	Bradford
2	20000	Burr
3	18000	Cauchy
4	25000	Exponentia
5	16000	ExtremeLB
6	23000	Fisk
7	17000	Gumbel
8	13000	Laplace
9	19000	Logistic
10	26000	Pareto
11	22000	Reciprocal
12	11000	Weibull
13	29000	ExtremeLB
14	30000	Logistic
15	16000	Fisk

6 ПОИСК ДАННЫХ

6.1 Классификация алгоритмов поиска

Поиск объекта по заданному признаку является распространенной операцией при обработке данных. Существует большое количество методов его реализации.

Алгоритм поиска – алгоритм, который в заданном файле ищет запись с заданным ключом. Результатом работы алгоритма является запись или указатель на нее. Поиск в файле по данному ключу может закончится неудачей. В случае неудачного поиска необходимо добавить новую запись с данным ключом.

Внутренний поиск данных – поиск, при котором файл с данными располагается в основной памяти.

Внешний поиск данных – поиск, при котором часть или весь файл с данными располагается во внешней памяти.

Статический поиск – поиск, при котором содержимое файла с данными не меняется.

Динамический поиск – поиск, при котором в файл с данными вставляются элементы или удаляются из него.

Алгоритмы поиска подразделяются на виды:

- внешние и внутренние;
- статические и динамические;
- основанные на сравнении ключей или на цифровых свойствах ключей;
- использующие ключи или образы ключей [5;8].

6.2 Алгоритмы, использующие сравнения ключей

6.2.1 Поиск в последовательно организованных структурах

6.2.1.1 Последовательный поиск

Последовательный поиск применяется к неупорядоченным и упорядоченным файлам, используется для данных, размещённых в массиве и в

списке. Пусть дан массив целых чисел размерности *n,* ключ поиска *key*. Требуется найти значение *i* для которого *k[i]=key*, если *i* существует.

Алгоритм метода последовательного поиска включает этапы:

Этап 1. Ввод ключа поиска *key*.

Этап 2. Перебор элементов массива с первого до последнего с проверкой условия *k[i]=key*.

Этап 3. Если условие *k[i]=key* выполняется, то переход на этап 5, иначе переход на этап 4.

Этап 4. Если достигнут конец массива, то вывод на экран сообщения «Элемент не найден» и переход на этап 6, иначе проверка следующего элемента и переход на этап 3.

Этап 5. Вывод на экран сообщения «Элемент найден», переход на этап 6.

Этап 6. Конец алгоритма [5].

Число сравнений алгоритма последовательного поиска при удачном поиске $\frac{n+1}{2}$, при неудачном поиске n. Эффективность алгоритма последовательного поиска оценивается *O(n)*.

Существуют два способа организации файла, позволяющих уменьшить вычислительную сложность поиска:

1) предложен У. Э. Смитом;

2) предложен Дж. Мак-Кейбом.

Способ, предложенный У. Э. Смитом, заключается в переупорядочивании записей файлов. Записи, которые ищутся чаще, должны располагаться ближе к началу файла. Среднее число сравнений при последовательном поиске определяется по формуле

$$\sum_{i=1}^{n} ip_i + nq, \qquad (6.1)$$

где p_i – вероятность того, что ищется запись $k(i)$, $i = \overline{1,n}$;

q – вероятность того, что ищется запись, не содержащаяся в файле;

q_n – вероятность того, что ищется запись с ключом $k(n) < key$.

Способ, предложенный Дж. Мак-Кейбом, заключается в том, чтобы просматриваемую запись переместить в позицию, более близкую к началу файла, если число операций, необходимых для продвижения элемента, невелико. Применяются следующие методы продвижения:

1) запись помещается в начало файла;

2) запись меняется местами с записью, предшествующей ей.

Если вероятность $p_i = \frac{1}{n}$, $i = \overline{1,n}$, то самоорганизующийся файл находится в случайном порядке и среднее число сравнений $\frac{n+1}{2}$. Если распределение подчиняется закону Зипфа, то среднее число сравнений $\frac{n\ln 4}{\ln n}$. Рассмотренный способ уменьшает вычислительную сложность удачного поиска, но не влияет на сложность неудачного поиска. Алгоритм способа, уменьшающего сложность неудачного поиска, включает этапы:

1) сортировка записей файла по возрастанию (убыванию);

2) перебор записей файла;

3) неудачный поиск прекратится, если встретится запись с ключом, большим (меньшим) искомого.

Средняя вычислительная сложность поиска определяется по формуле

$$\sum_{i=1}^{n} ip_i + \sum_{i=0}^{n-1}(i+1)q_i + nq_n, \qquad (6.2)$$

где q_i – вероятность того, что ищется запись с ключом $k(i) < key < k(i+1)$, $i = \overline{1,n-1}$;

q_n – вероятность того, что ищется запись с ключом $k(n) < key$.

Сложность удачного поиска остается такой же, как и для неупорядоченного. Её нельзя сократить, т. к. не удаётся использовать идею самоорганизующегося файла [5; 8].

6.2.1.2 Двоичный поиск

Эффективным методом поиска в упорядоченном файле, представленном в виде массива, является *двоичный (бинарный) поиск.* Алгоритм был предложен Дж. В. Мочли. Аргумент поиска сравнивается с ключом средней записи, если они равны, то поиск заканчивается удачно, в противном случае поиск продолжается либо в левой половине файла, либо в правой половине файла.

Двоичный поиск выполняется в отсортированном массиве и включает этапы:

Этап 1. Ввод значения ключа поиска *key*.

Этап 2. Определение области поиска элемента массива *(L, R), L=0, R=n-1*.

Этап 3. Если *R>L*, то переход на этап 4, иначе переход на этап 8.

Этап 4. Определение индекса среднего элемента *m=(L+R) div 2*.

Этап 5. Если *A[m]=key*, то переход на этап 7, иначе переход на этап 6.

Этап 6. Если *Key>A[m], то L=m+1, R=n-1*, иначе *L=0, R=m,* переход на этап 3.

Этап 7. Вывод на экран сообщения «Элемент найден», переход на этап 9.

Этап 8. Вывод на экран сообщения «Элемент не найден», переход на этап 9.

Этап 9. Конец алгоритма [5; 6].

Запись алгоритма двоичного поиска рекурсивная. Затраты на организацию рекурсии могут оказаться большими при практическом использовании.

Каждое сравнение при двоичном поиске сокращает в два раза число записей, которые надо исследовать. Максимальное число сравнений *[log2n]+1*. Среднее число сравнений при удачном поиске приближенно равно *log2(n–1)*.

Ни один алгоритм, основанный на сравнении ключей, не может иметь меньшую вычислительную сложность, чем алгоритм двоичного поиска. Любой алгоритм можно представить в виде дерева решений.

Максимальное число сравнений при удачном поиске не менее $[log_2 n]$.

Основным недостатком двоичного поиска является то, что он применим лишь к данным, представленным в виде массива. По этой причине двоичный поиск практически бесполезен в случае динамического поиска, когда много записей вставляется в связный список и удаляется из него. В случае внешнего поиска этот алгоритм можно использовать тогда, когда файл хранится в памяти с произвольным доступом [5].

6.2.1.3 Фибоначчиев поиск

Фибоначчиев поиск – модификация двоичного поиска, предложенная Д. Е. Фергюсоном. Предположим, что *n*+1 равно *j-му* числу Фибоначчи *Fj*. Тогда сравниваем ключ записи с номером F_{j-1} и аргумент поиска. Если они не равны, то поиск продолжается либо в правой, либо в левой частях файла, как и в случае двоичного поиска. Размеры этих частей, увеличенные на 1, также равны числам Фибоначчи, соответственно F_{j-2} и F_{j-1}.

В общем случае, когда *n+1* не равно какому-либо числу Фибоначчи, в качестве *Fj* надо рассмотреть ближайшее число Фибоначчи, большее *n+1*.

Любой алгоритм поиска с помощью сравнения ключей выполняет не меньше сравнений, чем алгоритм двоичного поиска. Алгоритм фибоначчиева поиска выполняет при удачном поиске в среднем около $1.042\,log_2\,n$ сравнений. Это больше числа сравнений при двоичном поиске. Алгоритм фибоначчиева поиска использует лишь операции сложения и вычитания, в то время как двоичный поиск использует операцию деления на 2. Поэтому в некоторых случаях фибоначчиев поиск может оказаться предпочтительнее двоичного [5; 8].

6.2.1.4 Интерполяционный поиск

Алгоритм *интерполяционного поиска* разработан У. У. Петерсоном.

Множество хранится в массиве. Массив отсортирован. Метод сокращает область поиска на каждой итерации. Искомый элемент определяется следующим образом. Пусть имеется область поиска (L, R). Элементы множества – целые числа, возрастающие в арифметической прогрессии. Тогда искомый элемент должен находится в массиве под индексом

$$m = L + \frac{(R-L)*(Key-A[L])}{A[R]-A[L]}, \qquad (6.3)$$

где $A[n]$ – массив целых чисел;

n – размерность массива;

Key – ключ поиска;

L – левая граница области поиска;

R – правая граница области поиска.

Метод работает для любых равномерно распределенных данных. Если данные распределены неравномерно, то интерполяционный поиск увеличивает число шагов по сравнению с дихотомическим поиском. Теоретическая сложность интерполяционного поиска – $T(log(log(n)))$.

Интерполяционный поиск асимптотически предпочтительнее двоичного. По существу, один шаг двоичного поиска уменьшает количество рассматриваемых записей с n до $\frac{1}{2}n$, а один шаг интерполяционного – с n до $\sqrt{n}$. Интерполяционный поиск требует в среднем около $log_2(\,log_2\,n)$ сравнений [1; 5].

6.2.1.5 Индексно-последовательный поиск

Пусть исходный файл *K,* содержащий *n* элементов, отсортирован по ключам. Разобьем файл на блоки, содержащие не более *m* элементов. Создадим дополнительный файл *INDEX* размером $s = \left\lfloor \frac{n-1}{m} \right\rfloor + 1$, называемый индексным, каждый элемент которого состоит из ключа *kindex* и указателя *pindex* на запись с ключом *kindex* исходного файла *K.* В файл *INDEX* помещаем максимального представителя каждого блока и указатель на него (рисунок 6.1). Элементы файла *INDEX* также являются отсортированными по ключам. Сначала находим первый элемент в *INDEX,* ключ которого не меньше элемента поиска *key*≤*INDEX.kindex[j].* Затем находим запись в файле *K* с ключом, равным аргументу поиска *key,* на участке между значениями *K[INDEX.pindex[j]* +1]<*key*<*K [INDEX.pindex[j*+1]].

Например, для поиска *key* = 44 (рисунок 6.1) находим номер блока *j*=2 (44≤*INDEX.kindex[2]*=51), последовательный поиск в котором позволяет найти искомое значение 44. При поиске *key*=45 обращение к блоку с номером 2 приводит к неудачному поиску.

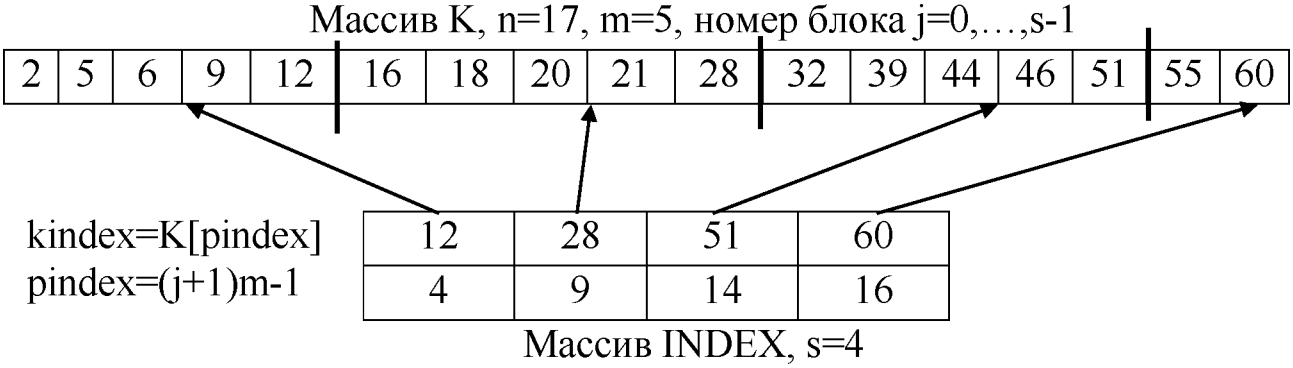


Рисунок 6.1 – Реализация индексно-последовательного поиска в массиве *K*

В случае наличия нескольких записей с одним и тем же ключом описанный алгоритм необязательно возвратит указатель на первую такую запись в файле.

В худшем случае (при неудачном поиске) потребуется произвести число операций

$$N = s + \frac{n}{s}, \tag{6.4}$$

где s – размер индексного файла;

n – размер исходного файла.

Минимальное значение этого выражения достигается при $s = \sqrt{n}$.

При использовании индексно-последовательного поиска можно основной файл и индексный файл представлять в виде списков. Это увеличит расход памяти, однако значительно облегчит вставку и удаление записей при динамическом поиске.

Если файл очень большой, то даже использование индексного файла может быть недостаточным для получения необходимой эффективности поиска. В этом случае строится индексный файл второго уровня, который обрабатывается как INDEX к индексному файлу первого уровня [5; 8].

6.3 Поиск в деревьях

6.3.1 Случайные двоичные деревья поиска

Минимальная вычислительная сложность поиска в дереве из *n* элементов имеет оценку функции сложности *0(1)*. Максимальная вычислительная сложность *O(n)*. Это неприемлемо во многих приложениях. Существует два подхода к устранению этого недостатка:

1) когда осуществляется статический поиск, можно построить оптимальное двоичное дерево поиска, время поиска в котором минимально.

2) когда производится динамический поиск, можно так вставлять и удалять записи, что дерево всегда останется сбалансированным по некоторому выделяемому признаку. Отметим, что при формировании дерева случайным образом полностью сбалансированные и полностью не сбалансированные деревья встречаются редко. Поэтому в среднем при поиске в древовидных структурах вычислительная сложность оценивается величиной *O(logn)*.

Расширенное двоичное дерево – двоичное дерево, у которого каждый внутренний узел имеет двух потомков [5].

В расширенном двоичном дереве с *n* внутренними узлами всегда существует *n+1* внешний узел (лист).

Длина внутренних путей I(T) – сумма уровней всех внутренних узлов расширенного двоичного дерева *T*.

Длина внешних путей E(T) – сумма уровней всех внешних узлов расширенного двоичного дерева *T*.

Средняя длина внутренних путей расширенного двоичного дерева T – величина, равная $\frac{I(T)}{n}$.

Средняя длина внешних путей расширенного двоичного дерева T – величина, равная $\frac{E(T)}{n+1}$.

Расширенное двоичное дерево изображено на рисунке 6.2.

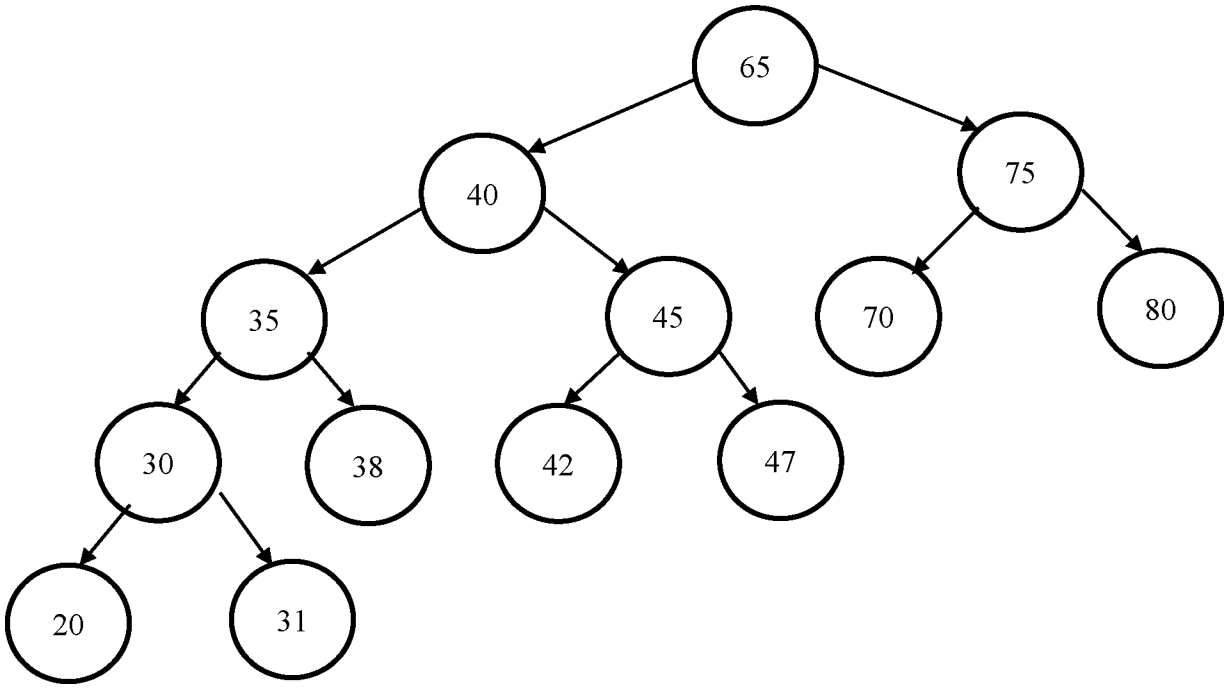


Рисунок 6.2 – Расширенное двоичное дерево поиска

Между длиной внешних и длиной внутренних путей в дереве существует связь. Длина внешнего пути любого расширенного двоичного дерева *T,* имеющего *n* внутренних узлов, на *2n* больше длины внутреннего пути и рассчитывается по формуле

$$E(T) = I(T) + 2n. \quad (6.5)$$

В расширенных двоичных деревьях поиска значения элементов (ключей) размещаются во внутренних узлах дерева. Высота дерева определяет вычислительную сложность. Длина внутренних путей характеризует вычислительную сложность удачного поиска. Длина внешних путей задает вычислительную сложность в случае неудачного поиска [1; 5].

При удачном поиске в расширенном двоичном дереве поиска, образованном n произвольными ключами, в среднем требуется $1{,}39\, log_2\, n$ сравнений.

Для проведения вставки нового элемента и при неудачном поиске в расширенном двоичном дереве, образованном n произвольными ключами, в среднем требуется около $1{,}39\,log_2\,n$ сравнений.

В худшем случае для поиска в двоичном дереве поиска с n ключами может требоваться n сравнений [2; 5].

6.3.2 Оптимальные двоичные деревья поиска

Пусть в построении дерева участвуют n различных упорядоченных записей $k<k2<...<km$ и поиск каждой $k;$ осуществляется с вероятностью $p_i;$. Если запись с ключом *key* отсутствует и ее значение лежит между k_i и k_{i+1} $(k_i<key<k_{i+1})$, вероятность этого события равна q_i. Вероятность того, что аргумент поиска меньше, чем k_1, равна q_0, а q_n – вероятность того, что аргумент поиска больше, чем k_n, при этом $\sum_{i=1}^{n} p_i + \sum_{i=1}^{n} q_i = 1$. Требуется построить двоичное дерево поиска, в котором среднее число сравнений при поиске будет минимальным при фиксированных k_i, p_i, q_i. Обозначим узлы дерева, которые не хранят значений и соответствуют неудачному поиску (листья), через y_0, y_1, …,y_n. Тогда среднее число сравнений определяется по формуле [5]

$$\sum_{i=1}^{n} p_i(уровень(k_i) + 1) + \sum_{i=0}^{n} q_i уровень(y_i). \qquad (6.6)$$

Цена дерева – среднее число сравнений.

Оптимальное дерево – дерево с минимальной ценой.

Взвешенная длина пути дерева T – величина, определяемая по формуле

$$|T| = \sum_{i=1}^{n} p_i уровень(k_i) + \sum_{i=0}^{n} q_i уровень(y_i). \qquad (6.7)$$

Цена дерева T есть $|T| + \sum_{i=1}^{n} p_i$. Поскольку $\sum_{i=1}^{n} p_i$ не зависит от структуры T, во внимание принимается $|T|$.

Метод построения оптимальных деревьев разработали Э. Н. Гильберт и Э. Ф. Мур.

Пусть ключи *A, B,* C, *D* имеют частоты обращения (или веса внутренних *узлов)* p_1=6, p_2=2, p_3=8, p_4=7 соответственно, и безуспешного поиска не бывает (веса всех листьев равны нулю). Четыре дерева поиска и их взвешенные длины путей приведены на рисунках 6.3-6.6.

Среднее время поиска в третьем дереве более чем в два раза превосходит среднее время поиска в четвертом дереве [5].

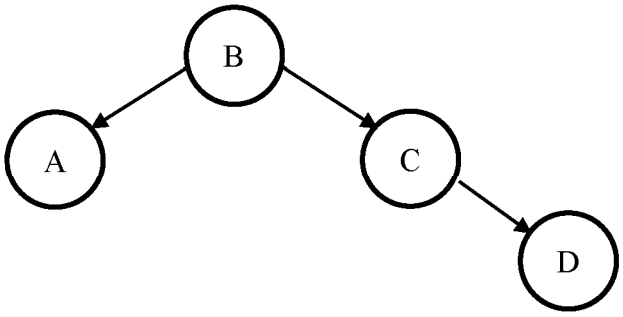


Рисунок 6.3 – Дерево поиска со взвешенной длиной $|T_{ABCD}| = 28$

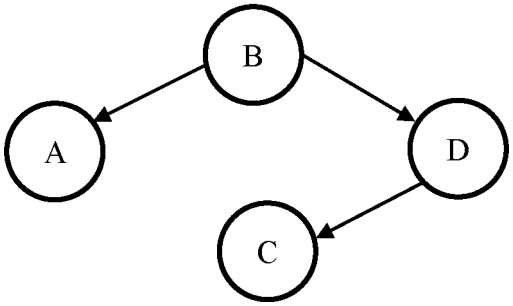


Рисунок 6.4 – Дерево поиска со взвешенной длиной $|T_{ABCD}| = 29$

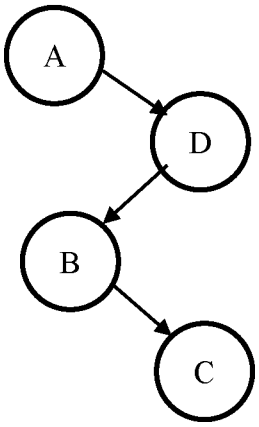


Рисунок 6.5 – Дерево поиска со взвешенной длиной $|T_{ABCD}| = 35$

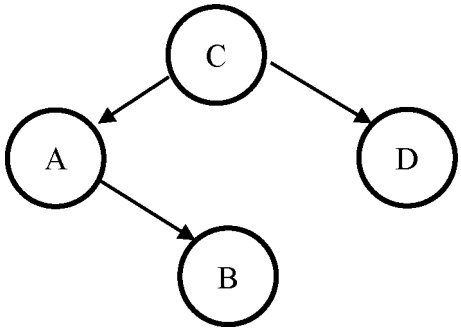


Рисунок 6.6 – Дерево поиска со взвешенной длиной $|T_{ABCD}| = 17$

Определим оптимальное двоичное дерево поиска над множеством ключей с данными частотами. При данных весах узлов $p_i \geq 0$ ($1 \leq j\rho \leq n$) и весах листьев $q_i \geq 0$, ($0 \leq i \leq n$.). Необходимо построить двоичное дерево поиска *T* над упорядоченным множеством всех ключей *k1<k2<...,<km* такое, что взвешенная длина пути дерева минимальна [5].

Поиск оптимального дерева можно перебором всех возможных деревьев из *n* элементов. Количество деревьев примерно равно $\frac{4^n}{n\sqrt{\pi n}}$. Метод динамического программирования позволяет построить полиномиальный алгоритм, используя следующие факты:

1 Оптимальное двоичное дерево поиска *T* с весами q_0, p_1, q_1,p_2,…,p_n,q_n имеет некоторый вес p_i в корне, левым поддеревом корня является оптимальное двоичное дерево поиска T_l с весами q_0, p_1, q_1, p_2,,…, p_{i-1}, q_{i-1} и правым поддеревом корня является оптимальное двоичное дерево поиска T_r,. с весами q_i, p_{i+1}, … , p_n, q_n.

2 Цена дерева *T* рассчитывается с помощью информации о двух поддеревьях T_l и T_r. В частности, $|T|$ можно вычислить по взвешенным длинам путей $|T_l|$ и $|T_r|$ и суммам весов W_l и W_r,. Соответственно по формуле

$$|T| = |T_l| + |T_r| + W_l + W_r, \qquad (6.8)$$

где $W_l = \sum_{j=1}^{i-1} p_j + \sum_{j=0}^{i-1} q_j$; $W_r = \sum_{j=j+1}^{n} p_j + \sum_{j=i}^{i-1} q_j$ – суммы весов элементов, входящих в левое и правое деревья соответственно.

Принцип оптимальности гласит, что для того, чтобы выбрать корень оптимального дерева, нужно вычислить для каждого узла k_i с весом p_i взвешенную длину пути в предположении, что k_i является корнем *T*. Это требует знания оптимальных правого и левого поддеревьев k_i. Если эти вычисления проводятся по рекурсивной схеме, алгоритм построения оптимального дерева организуют снизу вверх. Для 1<*i*< *n* строится *n*+*i*–1 оптимальное дерево на *i* последовательных узлах и смежных с ним листьях, используя построенные перед этим оптимальные деревья на *i–1* или меньшем количестве последовательных узлов. Построение осуществляется перебором всех возможных вариантов корней. Пересчитываются взвешенные длины путей и выбирается дерево с минимальной взвешенной длиной пути. Каждое оптимальное поддерево на последовательных узлах строится ровно один раз. Начало построения при *i*=1 тривиально, поскольку на одном узле существует только одно дерево. Когда процесс заканчивается при *i=n,* получено искомое оптимальное дерево [5].

Алгоритм Гарсия-Воча включает следующие этапы.

Этап 1. *Комбинация.* Рассматривается последовательность элементов y_i в заданном нами порядке с приписанными им весами q_i. Они будут расположены в листьях дерева (внешние узлы), которое создаётся.

Комбинируются два узла y_{k-1} и y_k. Пусть j и k – индексы, такие, что $j<k$ и выполняются условия:

$$\begin{aligned} &\text{а) } q_{i-1} > q_{i+1} \text{ при } \leq 1i < k; \\ &\text{б) } q_{k-1} \leq q_{k+1}; \\ &\text{в) } q_i < q_{k-1} + q_k \text{ при } j \leq i < k-1; \\ &\text{г) } q_{j-1} \geq q_{k-1} + q_k. \end{aligned} \tag{6.9}$$

В результате получается узел с весом $q_{k-1} + q_k$. Он располагается после листа с номером $j-1$. Процесс продолжается до тех пор, пока в рабочей последовательности не останется один узел.

Этап 2. *Присваивание уровней.* В последовательности остается единственный узел-корень. Номер его уровня будет нулевым. Необходимо проделать шаги комбинации в обратном порядке, определяя при этом уровни, на которых будут находиться узлы. Если узел $q_i + q_j$ имеет уровень m, узлы q_i и q_j, породившие его, имеют уровень $m+1$.

Этап 3. *Рекомбинация.* Рассмотрим изначальную последовательность листьев с полученными номерами их уровней. Начинаем заново комбинировать узлы по следующим правилам:

•узлы, содержащие q_i и q_j, должны быть соседними в рабочей последовательности;

•уровни k_i и k_jдолжны быть одинаковыми и максимальными по всем (q_i,q_j), удовлетворяющим предыдущему условию;

•индекс i должен быть минимальным среди всех (q_i,q_j), удовлетворяющим предыдущему условию.

В процессе этапа рекомбинации уже не используются значения q_i, а учитывается номер уровня узла. У каждого создаваемого узла номер уровня будет на единицу меньше, чем у его потомков. Двоичное дерево, формируемое на этом этапе, имеет минимальную взвешенную длину пути среди всех двоичных деревьев поиска, листья которых имеют веса (слева направо) $q_0, q_1, \ldots, q_n$.

При данных вероятностях $p_1, p_2, \ldots, p_n, q_1, q_2 \ldots, q_n$ существуют две эвристики для построения двоичного дерева поиска.

Первая эвристика состоит в том, чтобы поддерживать дерево «сбалансированным», насколько это возможно. Эта эвристика естественна, поскольку противоположность сбалансированного дерева (полностью асимметричное дерево) является, по существу, линейным списком и требует линейного времени поиска вместо логарифмического. Среди нескольких понятий «сбалансированности» следующие два имеют смысл в данном контексте:

1) выбрать корень (как всего дерева, так и каждого поддерева) так, чтобы у его правого и левого поддеревьев максимально близкими были либо число узлов, либо сумма весов;

2) помещать наиболее вероятные записи около корня; при последовательном применении эта эвристика приводит к деревьям, у которых веса узлов монотонно не возрастают вдоль любого пути от корня к листу (такие деревья называются *монотонными).*

Вторая эвристика заключается в том, чтобы начать с некоторого сбалансированного по некоторому закону дерева и изменять дерево локально, восстанавливая его сбалансированность [2; 5].

6.4 Поиск, использующий образы ключей (хеширование)

Метод *хеширования* (*hashing* – перемешивание) использует при поиске целочисленный образ ключа элемента. Это позволяет сократить вычислительную сложность поиска при возможном увеличении используемой для этого памяти.

Хеширование – эффективный способ представления данных, позволяющий быстро выполнять операции вставки, поиска и удаления элемента.

Данные в этом случае хранятся *хеш-таблицах* – специальным образом заполненных массивах M размера m. Размещение элементов в хеш-таблицах производится в соответствии с некоторой целочисленной функцией *H(k) – хеш-функцией,* определенной на множестве ключей *к:*

$$H: K \rightarrow \{0, 1, 2, ..., m-1\}. \qquad (6.10)$$

Алгоритмы поиска, использующие хеширование, включают два этапа:

Этап 1. Вычисление хеш-функции *H,* которая преобразует ключ поиска в адрес в хеш-таблице. Различные ключи должны отображаться на раз-

личные адреса, но часто два и более различных ключей $k \neq k^*$ могут преобразовываться в один и тот же адрес в таблице $H(k) = H(k^*)$. Эта ситуация называется *коллизией,* а ключи с одинаковыми значениями хеш-функции – *синонимами.*

Этап 2. Процесс разрешения конфликтов (коллизий), в ходе которого обрабатываются ключи-синонимы.

Для разрешения коллизий используются связные списки. Метод находит применение в динамических ситуациях, когда заранее трудно предвидеть количество синонимов. В другом методе разрешения коллизий высокая производительность поиска обеспечивается для элементов, хранящихся в массиве [2; 5].

Требования к хеш-функции:

1) для любого ключа k хеш-функция $H(k)$ с равной вероятностью принимает любое из m возможных значений.

2) значения хеш-функций должны быстро вычисляться.

Способы построения хеш-функций:

1) метод деления с остатком (модульная хеш-функция):

2) мультипликативный метод;

3) универсальное хеширование.

Метод деления с остатком (модульная хеш-функция) – метод, основанный на построении хеш-функции посредством деления с остатком, состоит в том, что ключу k ставится в соответствие остаток от деления k на m, где m – число возможных хеш-значений:

$$H(k) = k \bmod m. \qquad (6.10)$$

Мультипликативный метод. Пусть количество хеш-значений равно m. Зафиксируем константу A в интервале $0<A<1$ и положим

$H(k) = \lfloor m(kA \bmod 1) \rfloor$,

где $kA \bmod 1$ – дробная часть kA ;

$\lfloor m(kA \bmod 1) \rfloor$ – целая часть.

В отличие от метода деления с остатком, при таком ее вычислении значение хеш-функции в меньшей степени зависит от m. В качестве m выбирают степень двойки, так как умножение в этом случае реализуется простым сдвигом. Метод умножения работает при любом выборе константы A, но некоторые значения A могут быть лучше других. Оптимальный выбор

зависит от того, какого рода данные подвергаются хешированию. Д. Кнут пришел к выводу, что значение $A \approx \frac{\sqrt{5}-1}{2}$=0,6180339887 (золотое сечение).

Универсальное хеширование. Если известна хеш-функция *H(k),* то можно подобрать вводимые ключи так, что она принимает на них одно и то же значение. В результате время поиска нужного ключа будет *O(n).* Чтобы избежать такой ситуации, надо выбирать хеш-функцию из некоторого класса функций случайным образом, независящим от входных данных. Такой подход называется *универсальным хешированием* [5].

6.4.1 Разрешение коллизий методом цепочек

Открытое хешированием – способ разрешения коллизий с помощью метода цепочек. Пусть задана хеш-функциия *H(k),* определенная на множестве ключей *K:*

H: K →{0, 1, 2, ..., *m* – 1}.

Зададим массив (таблицу) списков *M* размером *m.* Элемент *k* размещаем в список *M[H(k)].* Элементы, для которых значения хеш-функции совпадают, хранятся в одном и том же списке. В *M[j]* хранится указатель на список тех элементов, для которых значение хеш-функции равно *j;* если таких элементов нет, то список *M[j]* пуст [5].

Если новый элемент размещается в начало списка, то это требует *O(1)* операций. Максимальное время поиска пропорционально длине списка синонимов. Удаление найденного элемента проводится за O*(1)* операций.

Рассмотрим пример разрешения коллизий с помощью метода цепочек для последовательности 18, 14, 9, 20, 19, 12, 5, 27, 16, 34. Результат вставки элементов в пустую хеш-таблицу в соответствии со значениями хеш-функции *H(k) = k mod 11* приведен ниже. Элемент 18 размещается в списке 7, 14 – в списке 3, 9 и 20 – в списке 9 и т. д. Очередной элемент вставляется в начало списка с целью минимизации времени вставки (таблица 6.5) [5].

6.4.2 Разрешение коллизий методом открытой адресации

Пусть требуется разместить в хеш-таблице *n* элементов. Выделяется большая непрерывная область памяти для хранения *m* элементов в таблице. Если *n* < *m,* то в хеш-таблице можно хранить все ключи при остающемся

свободном месте. Между элементами не используются никакие связи. Существует несколько методов хранения *n* элементов в таблице размером *m>n,* при которых разрешение конфликтов осуществляется за счет наличия пустых мест в таблице. Такие методы называются методами хеширования *с открытой адресацией* (*закрытое хеширование*). В этих методах так же, как и в методе цепочек, размер таблицы определяется числом различных значений хеш-функции [5].

Таблица 6.5 – Хеш-таблица

Номер списка	Элементы списка
0	Пусто
1	34→ 12
2	Пусто
3	14
4	Пусто
5	16→27→5
6	Пусто
7	18
8	19
9	20→9
10	Пусто

Линейное зондирование – метод открытой адресации. Пусть задана хеш-функция *H(k),* определенная на множестве ключей $H: K \rightarrow \{0, 1, 2, ..., m-1\}$.

Зададим массив (таблицу) *M* размером *m.* При вставке элемента *k* в таблицу вычисляется значение *j=H(k).* Далее, если *M[j]* пусто, в него записывается значение *k,* а если *M[j]=k,* то элемент *k* в таблицу не вставляется. В случае если *M[j]* занято элементом, не совпадающим по значению со вставляемым (коллизия), проверяется следующая *((j*+1) *mod m)-я* позиция в таблице. Процесс продолжается до тех пор, пока не будет найдена первая свободная позиция, в которую и осуществляется вставка элемента *k.* Если за *m* попыток не найдена свободная позиция для вставки элемента, значит, вся таблица заполнена, и для дальнейшей работы необходимо увеличение размера таблицы. При этом требуется переразмещение всех элементов в новую таблицу в соответствии с новой хеш-функцией [5].

Зондирование (проба) –проверка, определяющая, содержит ли данная позиция таблицы элемент, равный искомому, или она свободна. Шаг зондирования может быть отличен от 1.

В ходе линейного зондирования возможны три исхода:

1 Позиция таблицы содержит элемент, совпадающий с искомым. В случае поиска найден искомый элемент. При вставке, если в таблице не хранятся одинаковые элементы, он в таблицу не записывается.

2 Позиция таблицы содержит элемент, не совпадающий с искомым по значению. Зондирование таблицы продолжается (не более m раз с возвратом к началу таблицы при достижении ее конца) до тех пор, пока не будет найден искомый ключ (см. п. 1) или пустая позиция таблицы [1; 5].

3 Поиск завершился в пустой позиции таблицы. В случае поиска элемента это неудачный исход. В случае вставки это позиция для размещения элемента.

Функция, определяющая линейную последовательность зондирований (проб), записывается в виде уравнения

$$P(k,i) = (H(k) + ir) \ mod \ m, \qquad (6.12)$$

где i – номер попытки (нумерация попыток начинается с 0);

r – шаг зондирования, натуральное число.

При работе с элементом k начинают с ячейки *M[H(k)],* а затем перебирают ячейки таблицы подряд *M[H(k)* + r*], M[H(k) + 2r]* и т. д.

Рассмотрим линейное зондирование на примере вставки элементов 18, 14, 9, 20, 19, 12, 5, 27, 16, 34 в пустую хеш-таблицу (размером *m=11*) с открытой адресацией. Разрешение конфликтов (коллизий) будем производить с применением линейного зондирования с шагом *r=1*. Первоначально все элементы таблицы инициализируются значением *EMPTY*.

Элемент 18 размещается в позицию 7, затем 14 помещается в позицию 3. Элемент 9 размещается в позиции 9. При размещении 20 в таблице возникает коллизия: позиция 9 уже занята, и приходится размещать 20 в позиции 10 и т. д. При достижении конца таблицы зондирование продолжается с ее начала. Например, 16 должно было бы занять позицию 5, но возникает коллизия, и с седьмой попытки зондируется позиция нулевая (таблица 6.6).

Таблица 6.6 – Хеш-таблица

Индекс	0	1	2	3	4	5	6	7	8	9	10
Значение элементов	16	12	34	14	EMPTY	5	27	18	19	9	20

Пусть требуется найти элемент 16. Значение хеш-функции *H(6) =5*. Начинаем поиск с пятой позиции. Поскольку в ней элемента 16 нет, просматриваем последовательно все занятые позиции, переходя к началу таблицы после достижения ее конца. Поиск заканчивается успешно в нулевой позиции [2; 5].

Пусть надо найти элемент 29. Вычисляем значение хеш-функции: *H(29)* = 7. Поиск начинаем с седьмой позиции. Проходим 7, 8, 9, 10, 0, 1, 2, 3-ю позиции, ключи в которых не равны 7. При достижении четвертой пустой позиции поиск заканчивается неудачей.

6.4.3 Идеальное хеширование

Идеальное хеширование – метод, который в наихудшем случае выполняет поиск за *O(1)* обращений к памяти. В этом случае используется двухуровневая методика хеширования с универсальным хешированием на каждом уровне. Первый уровень аналогичен хешированию с исключением коллизий методом цепочек с хеш-функцией *H: K* →{0, 1, 2, ..., *m* – 1}. На втором уровне вместо списков ключей-синонимов используются небольшие вторичные хеш-таблицы S_j , каждая со своей хеш-функцией $H_j(k), j = \overline{0, m-1}$.

Все хеш-функции выбираются из универсального семейства *F*.

Рассмотрим идеальное хеширование на примере. Для построения универсального семейства хеш-функций воспользуемся теорией чисел. Выберем достаточно большое простое целое число *p>m,* чтобы все возможные ключи *k* находились в диапазоне 0, ... *p* – 1. Обозначим *Zp ={0,1,...,p–1}*, Z_p^**={1,2,...,p–1}*. Хеш-функцию зададим в виде

$$H_{a,b}(k) = ((ak + b)\ mod\ \ p)\ mod\ \ m, \qquad (6.13)$$

где $a \in Z_p^*$, $b \in Z_p$.

Множество всех таких функций образует универсальное семейство *Fp,m = {Ha,b : a* $\in Z_p^*$, *b* $\in Z_p$*}*. Каждая хеш-функция *Ha,b* отображает *Zp* на

Zm, и их количество в $F_{p,m}$ равно *p(p*–1), так как а можно выбрать (p–1) способами, а $b - p$ способами [5].

Рассмотрим размещение последовательности ключей 18, 14, 9, 20, 19, 12, 5, 27, 16, 34 в хеш-таблицу (таблица 6.7).

На первом уровне для параметров *m*=11, *p*=101, a=3, b=17 вычислим значения хеш-функции.

Таблица 6.7 – Хеш-таблица

k	18	14	9	20	19	12	5	27	16	34
$H_{3,17}(k)$	5	4	0	0	8	9	10	10	10	7

В результате получаем две цепочки ключей-синонимов (9, 20 и 5, 27, 16). На втором уровне формируем вторичные хеш-таблицы S_j, каждую размером m_j, с разрешением коллизий методом открытой адресации.

Обозначим через n_j количество ключей-синонимов для *Ha,b(k)=j.*

Для уменьшения вероятности возникновения коллизий во вторичных хеш-таблицах задают их размер $mj = n_j^2$. Пусть *Ha,b(*k_i*)=j,* тогда в случае, если ключ k_i не имеет синонимов, задаем $a_j = b_j = 0$, а в противном случае выбираем их случайным образом [5].

6.5 Контрольные вопросы

1 Какое существует определение понятия поиска данных?

2 Какие этапы имеет алгоритм метода последовательного поиска?

3 Какие этапы имеет алгоритм метода двоичного поиска?

4 Какие этапы имеет алгоритм метода интерполяционного поиска?

5 Какие этапы имеет алгоритм метода поиска, использующего образы ключей (хеширование)?

6 Что понимается под хешированием?

7 Что понимается под линейным зондированием?

8 Что понимается под открытым хешированием?

9 Что понимается под идеальным хешированием?

10 Что понимается под универсальным хешированием?

6.6 Варианты заданий

Вариант 1

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности ООО «Центр оценки и продажи недвижимости». Одним из источников прибыли этой организации является покупка и продажа квартир. Центр оценки имеет большой штат специалистов, позволяющий этой организации проводить сделки купли-продажи на высоком профессиональном уровне. Владелец квартиры, желающий ее продать, заключает договор с Центром, в котором указывается сумма, срок продажи и процент отчислений в пользу Центра оценки и продажи недвижимости в случае успешного проведения сделки. Один клиент может заключить с Центром более одного договора купли-продажи одновременно, если он владеет несколькими квартирами. Обмен квартир специалисты центра непосредственно не производят. Для этих целей используется вариант купли-продажи [6; 7].

Вариант 2

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности отдела вневедомственной охраны квартир. Этот отдел обеспечивает электронную охрану квартир граждан в одном районе города. Для установки охранной сигнализации требуется наличие квартирного телефона. Один гражданин может заключить договор на охрану нескольких квартир. Из-за ложных срабатываний сигнализации возможно несколько выездов патрульных экипажей по одной квартире. На владельца квартиры, вовремя не отключившего сигнализацию после своего прихода домой, налагается штраф, величина которого оговаривается при заключении договора охраны. Если отдел вневедомственной охраны не уберег имущество владельца квартиры, то он выплачивает пострадавшему заранее оговоренную сумму. От величины этой суммы зависит размер ежемесячной оплаты за охрану квартиры.

Вариант 3

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности отдела приватизации жилья администрации города. В нашем городе на начало 2001 г. приватизировано около 80 000 квартир граждан. Еще далеко не все проживающие в «своих» квартирах стали собственниками своего жилья. Процесс приватизации продолжается и займет еще несколько лет. Главная задача программного комплекса – не допустить приватизации одним человеком более одной квартиры. К сожалению, в отделе приватизации не используется уникальный кадастровый номер здания, поэтому вам придется использовать составной первичный ключ (адрес) для таблицы зданий, квартир и проживающих. Помните, что некоторые из проживающих в квартире могут не участвовать в приватизации [6; 7].

Вариант 4

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности предприятия «Газкомплект» по учету платы за пользование газом и газовыми приборами. Плата взымается с каждой квартиры в зависимости от количества потребленного газа или от числа проживающих, если счетчик отсутствует. Ответственный квартиросъемщик обязан каждый месяц снимать показания счетчика и производить оплату за потребленный газ через Сбербанк. Наряду с отслеживанием платы за газ предприятие производит профилактическое обслуживание газовых приборов. Правила техники безопасности предусматривают осмотр газовой плиты инспектором предприятия раз в квартал. Если обнаружены неполадки в подключении плиты или ее работе, то работник предприятия обязан немедленно устранить их за счет абонента. Оплата оказанных услуг осуществляется на месте по квитанции [6; 7].

Вариант 5

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности «Бюро технической инвентаризации» по изго-

товлению и выдаче технических паспортов на объекты недвижимости. Перед регистрацией сделки с объектом недвижимости собственник объекта должен получить на него технический паспорт в БТИ. Ежедневно в БТИ обращается до 200 человек. Назначение программного комплекса – не пропустить ни одного документа. Если технический паспорт не готов в назначенный срок, то БТИ должно выплатить неустойку. Алгоритм изготовления документа следующий. Клиент обращается к инспектору, сдает ему необходимые справки, согласовывает дату выхода техника на обмер, уплачивает аванс. Инспектор передает заявку начальнику отдела. Начальник отдела назначает исполнителя и техника. Техник выполняет обмер объекта. Исполнитель изготавливает документ и передает в отдел выдачи. В назначенный срок клиент забирает готовый документ, доплатив недостающую сумму. Один клиент (физическое или юридическое лицо) может заказать несколько технических паспортов, за изготовление которых оплата может производиться частями.

Вариант 6

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности отдела аренды ЗАО «Сириус». После удачной приватизации, когда у руководства этого предприятия оказалась большая часть акций, дела некогда мощного предприятия пошли на спад. Основная часть работников была уволена по сокращению штатов. В настоящее время основной статьей получения прибыли является сдача в аренду другим предприятиям и организациям площадей, которыми владеет «Сириус». В его собственности имеется 12-этажное здание, которое состоит примерно из 300 помещений. Почти все они сдаются в аренду. Один арендатор может арендовать несколько помещений, причем срок аренды для каждого устанавливается отдельно. Величина арендной платы и ее периодичность устанавливается арендодателем. После окончания срока аренды договор может быть продлен на прежних или новых условиях. Субаренда площадей запрещена. Закрытые договоры не удаляются из базы данных для отслеживания предыдущих арендаторов [6; 7].

Вариант 7

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности телефонной компании. Основное назначение программного комплекса – отслеживание абонентской платы за телефоны. Клиентами компании могут быть как физические лица, так и организации. Расчет с организациями ведется в безналичной форме через банк. Физические лица вносят плату через кассу компании. Клиент телефонной компании может иметь несколько телефонных номеров. Дополнительная плата за подключенный параллельно аппарат не взимается. Если телефон у абонента не работает более суток, то плата за пользование телефоном уменьшается. Междугородние и международные звонки оплачиваются отдельно по заранее установленным расценкам [6; 7].

Вариант 8

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности мелкооптового книжного магазина. Менеджер магазина, изучив спрос на книжную продукцию в городе, принимает решение о закупке партии книг в том или ином издательстве. Некоторые пользующиеся повышенным спросом книги могут быть закуплены у посредников. Часть продукции заказывается через Интернет. Покупателем в мелкооптовом магазине может быть любой человек или организация, при условии, что величина покупки превысит одну тысячу рублей. Расчет с организациями производится через банк. Расчет с физическими лицами – наличными. Покупателю выписывается счет-фактура, которая имеет уникальный номер и содержит список книг с указанием их стоимости. После уплаты указанной суммы покупатель получает товар на складе.

Вариант 9

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности ОАО «Автовокзал». Это открытое акционерное общество занимается междугородними пассажирскими перевозками по Дальневосточному региону. В его собственности находится несколько де-

сятков автобусов различной вместимости. Штат водителей превышает количество автобусов. Средний уровень сменности для машины – 2,5. Водитель не может работать более одной смены в сутки. Билеты на рейсы продаются только в здании автовокзала. Возможна предварительная продажа. Маршрут автобуса может пролегать через несколько населенных пунктов. В этом случае пассажир может купить билет до любого промежуточного пункта. Освободившимся местом после выхода пассажира распоряжается водитель. Полученную выручку он сдает в кассу предприятия после прибытия с маршрута. На линии работает контроль. Если в автобусе будет обнаружен пассажир без билета, то на водителя налагается штраф.

Вариант 10

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности агентства знакомств. Агентство занимается организацией знакомств одиноких мужчин и женщин. Возможен один из двух вариантов: человек либо регистрируется в агентстве, оставляет информацию о себе, чтобы любой мог ознакомиться с его анкетой, либо знакомится с базой уже зарегистрированных клиентов и выбирает подходящий вариант. Регистрация и подбор кандидатуры – платные. Тот, кто делает выбор по базе, платит за каждый выбранный вариант. После того, как выбор сделан, агентство согласовывает дату и время встречи с каждой из сторон, формирует и передает приглашения для знакомства обеим сторонам. Во избежание недоразумений первая встреча происходит в агентстве. Клиенты, желающие быть исключенными из базы, переносятся в архив [6; 7].

Вариант 11

Разработать программное приложение, осуществляющее поиск в таблице данных деятельности ломбарда. Человек обращается в ломбард в том случае, если ему срочно нужны деньги. Например, недостает небольшой суммы для покупки квартиры, а подходящая квартира как раз продается, и на неё уже есть и другие покупатели. Тогда человек может пойти в ломбард и заложить вещи на необходимую сумму. В ломбарде с клиентом заключается договор. В нем оговариваются следующие условия: до какого срока выкуп вещи возможен без процентов, с какого времени будет взыматься

процент, по истечении какого срока выкуп вещи невозможен, и она поступает в собственность ломбарда. Невыкупленные вещи ломбард выставляет на продажу [6; 7].

Вариант 12

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности гостиницы. В гостинице существует большое количество возможных вариантов заселения гостей: все номера различаются по категориям (суперлюкс, люкс и т. д.), по количеству комнат в номере, количеству мест в каждом номере, а также по обустройству комнат – учитывается, например, наличие телевизора, холодильника, телефона. В обязанности администратора гостиницы входит подбор наиболее подходящего для гостя варианта проживания, регистрация гостей, прием платы за проживание, оформление квитанций, выписка отъезжающих. Учитывается также возможность отъезда гостя раньше указанного при регистрации срока, при этом производится перерасчет. Существует также услуга бронирования номера.

Вариант 13

Разработать программное приложение, осуществляющее поиск в таблице данных института селекции растений. Данный институт занимается сбором, выведением и продажей различных сортов семян. В его ассортименте можно найти семена практически всех возможных видов растений: от помидоров до редких цветов. Только что выведенные сорта заносятся в отдельный список для дальнейшего тестирования. Каждый сорт семян имеет свои характеристики, такие как урожайность, морозоустойчивость, адаптация к местным условиям, сроки созревания (раннеспелый, среднеспелый, поздний) и т. п. Покупатель может выбрать сорт, отвечающий тем или иным характеристикам. Компания занимается как оптовыми, так и розничными продажами. Оптовые покупатели заносятся в базу главным образом для того, чтобы информировать их о поступлении новых или отсутствовавших в определенный момент в продаже сортов.

Вариант 14

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности приемной комиссии университета. Каждый год университет зачисляет новых абитуриентов для возможного их поступления в университет после сдачи вступительных экзаменов. На бюджетную основу могут быть зачислены: абитуриенты, получившие на школьном экзамене высокий балл ЕГЭ и успешно прошедшие собеседование; абитуриенты, набравшие необходимый для бесплатного поступления балл на университетских экзаменах, а также абитуриенты, имеющие направление от какого-либо государственного предприятия. Все остальные могут поступить в университет на платной основе, набрав необходимое установленное университетом число баллов на вступительных экзаменах [6; 7].

Вариант 15

Разработать программное приложение, осуществляющее поиск в таблице данных о деятельности кассы авиакомпании. Касса авиакомпании занимается продажей билетов на предстоящие рейсы. В билете указывается номер и название рейса, а также все остальные необходимые для пассажира данные: дата и время вылета, прибытия, номер места и класс (бизнес, экономический). Цена билета зависит от рейса, лайнера, класса, а также от времени покупки билета – иногда авиакомпании делают скидки купившим билет более чем за месяц или на горящие рейсы – все зависит от желания компании. Билеты продаются только совершеннолетним гражданам при предъявлении паспорта. У авиакомпании обычно имеется несколько касс, расположенных в разных концах города, поэтому обязательно необходимо учитывать номер кассы, в которой был продан билет, во избежание недоразумений при сдаче или обмене билета [6; 7].

7 ПРИЛОЖЕНИЯ НА ГРАФАХ

Графы как структуры данных широко применяются при решении задач. Они могут использоваться для представления данных, моделей процессов, структур программ. Многие задачи, решаемые с использованием графов, относятся к задачам выбора. Они могут быть решены с помощью переборных алгоритмов. Сложность таких алгоритмов велика, многие практические задачи не могут быть решены за приемлемое время [4].

7.1 Основные определения теории графов

Граф $G=(V,E)$ – множество V, называемое *множеством вершин (узлов)* графа, множество E, представляющее множество ребер (дуг) графа и отображение множества ребер (дуг) на множество пар элементов из множества V.

Множества V и E конечны. Каждому ребру (дуге) графа $x \in E$ можно поставить в соответствие пару вершин $(u,v) \in V$ графа, тогда ребро (дуга) x соединяет вершины u и v.

Смежные вершины – две вершины u и v, соединенные в графе ребром x, ребро x инцидентно вершинам u и v.

Полный граф – граф, у которого каждая вершина смежна со всеми остальными вершинами.

Ориентированное ребро (дуга) – ребро x, направленное от одной вершины u к другой v.

Ориентированный граф (орграф) – граф, содержащий только ориентированные ребра.

Неориентированное ребро – ребро, не имеющее определенного направления.

Неориентированный граф – граф, содержащий неориентированные рёбра.

Смешанный граф – граф, содержащий рёбра и дуги.

Если между любой парой вершин графа имеется не более одного ребра в неориентированном графе или не более одного ребра данного направления в ориентированном графе, то такой граф называется *простым*, в противном случае – *мультиграфом*.

Взвешенный граф – граф, у которого каждому ребру приписан вес.

Изолированная вершина – вершина, у которой нет ни одной смежной вершины.

Нульграф – граф, имеющий только изолированные вершины.

Полустепень исхода ориентированного графа – число ребер, исходящих из начальной вершины *v*.

Полустепень захода вершины v – число ребер, для которых вершина *v* является конечной.

Полная степень вершины – сумма полустепеней исхода и захода.

Любая последовательность ребер простого орграфа такая, что конечная вершина любого ребра этой последовательности является начальной вершиной следующего за ним ребра, задает *путь в графе*. Путь в графе обходит все вершины данной последовательности, если начало пути находится в начальной вершине первого ребра, а конец – в конечной вершине последнего ребра последовательности.

Длина пути – число ребер в пути.

Простой путь (простой относительно ребер) – путь в графе, все ребра которого различны.

Элементарный путь (простой относительно вершин) – путь, в котором все вершины различны.

Все элементарные пути – простые, но не наоборот.

Цикл (контур) – путь, начинающийся и заканчивающийся в одной и той же вершине.

Простой цикл – соответствующий ему путь простой.

Элементарный цикл проходит через любую вершину не более одного раза. В элементарном цикле начальная вершина появляется два раза, в простом – может более двух раз.

Ациклический орграф – граф, не имеющий ни одного цикла.

Связный граф – граф, в котором есть пути между любыми двумя вершинами [1; 4].

7.2 Представления графов

Реализация алгоритмов обработки графов в виде машинных процедур зависит от способа представления графов на логическом и физическом

уровнях. Способ представления графа существенно зависит от типов структур данных, допускаемых используемым языком программирования и типом ЭВМ.

Выбор наилучшего способа представления графа зависит от природы моделируемых данных (процессов) и операций, выполняемых над ними.

На выбор подходящего представления влияют следующие факторы:

- число вершин;
- максимальная полустепень исхода;
- частота изменения числа вершин и ребер;
- является ли граф ориентированным.

Выбор представления графа влияет на эффективность его обработки.

Граф G=(*V,E)* может быть полностью определен простым перечислением двух множеств *V* и *E*. Этот способ представления не позволяет создавать эффективные алгоритмы обработки графов.

Широко распространенным способом представления графов являются матричный способ и соответствующее отображение их в векторной (смежной) памяти в виде двухмерных массивов. Этот способ имеет достоинства:

- массивы легко хранить и обрабатывать в ЭВМ;
- для получения характеристик графа могут быть использованы операции матричной алгебры;
- имеются хорошо известные алгоритмы обработки графов.

Матричный способ представления имеет недостаток, связанный с тем, что массивы являются статическими по размерам структурами [2; 4].

7.2.1 Матрица смежности

Граф представляется в виде матрицы смежности *An*n,* в которой $a_{ij} = 1$, если *(*v_i*;*v_j*)* принадлежит *E,* $a_{ij} = 0$ в противном случае.

Для простых неориентированных графов матрица смежности симметрична, $a_{ij} = a_{ji}$. В взвешенном графе $a_{ij} = w_{ij}$, где w_{ij} – вес ребра.

Матрица смежности зависит от упорядочения графа (от порядка нумерации вершин). Для различных упорядочений получаются различные матрицы (изоморфизм), однако любая матрица смежности графа *G* может

быть получена из другой матрицы смежности этого же графа путем перестановки некоторых строк и соответствующих им столбцов [1; 4].

Используя матрицу смежности, определяют все пути между вершинами v_i и v_j, их длины, пути, циклы, простые и элементарные пути (рисунок 7.1).

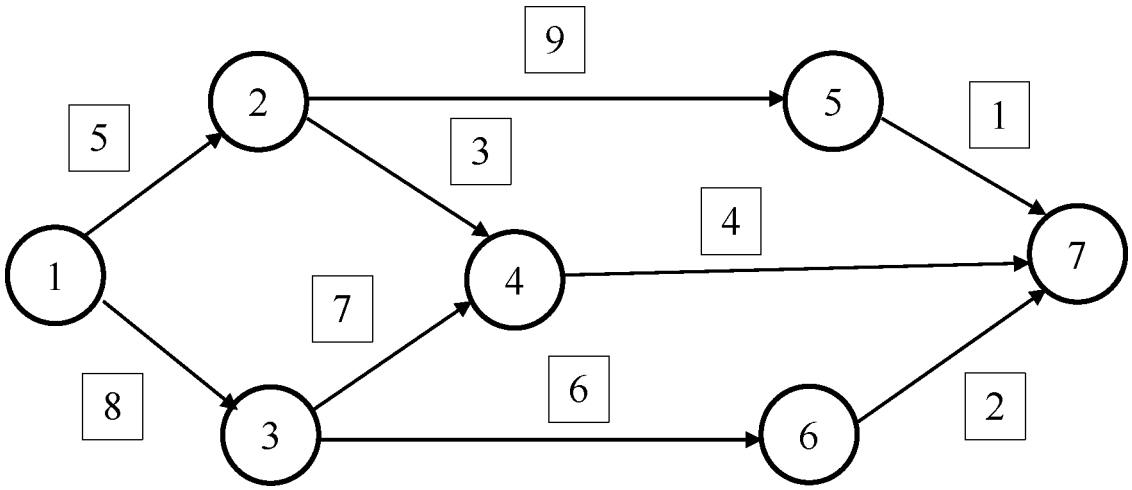


Рисунок 7.1 – Взвешенный ориентированный граф

Матрица смежности графа приведена в таблице 7.1

Таблица 7.1 – Матрица смежности

i/j	1	2	3	4	5	6	7
1	0	5	8	0	0	0	0
2	0	0	0	3	9	0	0
3	0	0	0	7	0	6	0
4	0	0	0	0	0	0	4
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	2
7	0	0	0	0	0	0	0

7.2.2 Матрица инцидентности

Граф можно задать матрицей инцидентности. Матрица инцидентности $B[b_{ij}]$ размерностью *n*m (n* – число вершин, *m* – число дуг) определяется следующим образом:

1) $b + 1$, если u_i; является начальной вершиной дуги a_j;

2) $b_{ij} = -1$, если u_i является конечной вершиной дуги *aj*;

3) $b_{ij} = 0$ в противном случае.

Если граф неориентированный, то его матрица инцидентности определяется аналогично, за исключением того, что элемент –1 поменяется на +1 [2; 4].

7.3 Алгоритмы кратчайших путей

Задачи определение путей в графах:

1) задачи определения наличия пути между двумя вершинами;
2) задачи определения всех путей между вершинами;
3) задачи определения экстремальных путей;
4) задачи определения путей от одной вершины до всех других;
5) задачи определения дуг, составляющих путь, и т. п.

Пусть дана матрица смежности. Единица в *i-й* строке и j-м столбце матрицы *A*, $a_{ij} = 1$, указывает на наличие ребра (v_i, v_j), на путь длиной 1 из v_i в v_j. Элементами матрицы A^2 будут:

$$a_{ij}^{(2)} = \sum_{k=1}^{n}(a_{ik} * a_{kj}). \quad (7.1)$$

Для каждого k условие $a_{ik} * a_{kj} = 1$ выполняется, когда оба элемента $a_{ik} = 1$ и $a_{kj} = 1$, т. е. имеются ребра (v_i, v_j) и (v_k, v_j), следовательно, существует путь из v_i в v_j длиной 2 через v_k. Тогда приведенная выше сумма равна числу различных путей длиной 2 из v_i в v_j через различные вершины v_k.

Аналогично элемент a_{ij} матрицы A^3 задает число путей длиной 3 из v_i в v_j, а по матрице A^r определяют число путей длиной *r* из v_i в v_j.

Для определения, существует ли в графе с *n* вершинами путь из v_i в v_j, необходимо проверить, имеются ли элементарные пути длиной, меньшей или равной *(n*–1). Такие пути определяются с помощью матрицы

$$B^{n-1} = A + A^2 + A^3 + \cdots + A^{n-1}. \quad (7.2)$$

Элемент b_{ij} показывает число существующих путей из v_i в v_j длиной, меньшей или равной (n–1). Если $b_{ij} > 0$, то вершина v_j достижима v_i. Матрица $B^n = A + A^2 + A^3 + \ldots + A^n$ определяет число элементарных путей циклов в графе, представленном матрицей *A*.

На практике часто требуется найти кратчайшее расстояние от одной вершины графа до другой. Эта задача моделируется с помощью связного

графа G, в котором каждому ребру приписан положительный вес, равный длине ребра. Длина пути равна сумме длин рёбер, составляющих путь [4].

Задачи о кратчайших путях относятся к фундаментальным задачам комбинаторной оптимизации, так как многие задачи можно свести к отысканию кратчайшего пути в графе. Существуют различные типы задач о кратчайшем пути между вершинами:

- двумя заданными вершинами;
- данной вершиной и всеми остальными вершинами;
- каждой парой вершин.

7.3.1 Алгоритм Дейкстры

Алгоритм Дейкстры разработан для нахождения кратчайшего пути между заданным исходным узлом и любым другим узлом сети.

В процессе выполнения алгоритма Дейкстры используется процедура пометки ребер.

Пусть i – предыдущий узел графа, j – последующий узел, S_i – кратчайшее расстояние от исходного узла 1 до узла i, l_{ij} – длина ребра графа$(i.j)$. Метка $[S_i,i]$ для узла j определяется по формуле

$$[\mathrm{S_j},i]=[\mathrm{S_i}+\mathrm{l_{ij}},\ \mathbf{i}],\ l_{ij}\geq 0. \qquad (7.3)$$

Метки в алгоритме Дейкстры могут быть двух типов: временные и постоянные.

Временная метка может быть заменена на другую временную, если будет найден более короткий путь к узлу. Статус временной метки изменяется на постоянный, если не существует более короткого пути от исходного узла к данному узлу.

Алгоритм Дейкстры включает следующие этапы:

Этап 1. Исходному узлу (узел 1) присваивается метка [0,–]. Полагаем $i=1$.

Этап 2. Вычисляются временные метки $[S_i+l_{ij},\ \mathrm{i}]$ для всех узлов j, которые можно достичь непосредственно из узла i и которые не имеют постоянных меток. Если узел j имеет метку $[\mathrm{S_j},\ \mathrm{k}]$, полученную от другого узла k, и если $S_i+l_{ij}<S_j$, тогда метка $[\mathrm{S_j},\ \mathrm{k}]$ заменяется на $[\mathrm{S_i}+\mathrm{l_{ij}},\ \mathrm{i}]$.

Этап 3. Если все узлы имеют постоянные метки, процесс вычислений заканчивается. Иначе выбирается метка $[\mathrm{S_r},\ \mathrm{p}]$ с наименьшим значением

расстояния S_r среди всех временных меток. Если таких меток несколько, то выбор произволен. Полагаем i = r и переходим к этапу 2 [4].

Пример нахождения кратчайшего пути по алгоритму Дейкстры.

Постановка задачи. Транспортная сеть состоит из пяти городов (рисунок 7.2). Расстояния между городами в километрах приведены возле соответствующих дуг сети. Найти кратчайшие расстояния от города 1 (узел 1) до остальных четырех городов [10].

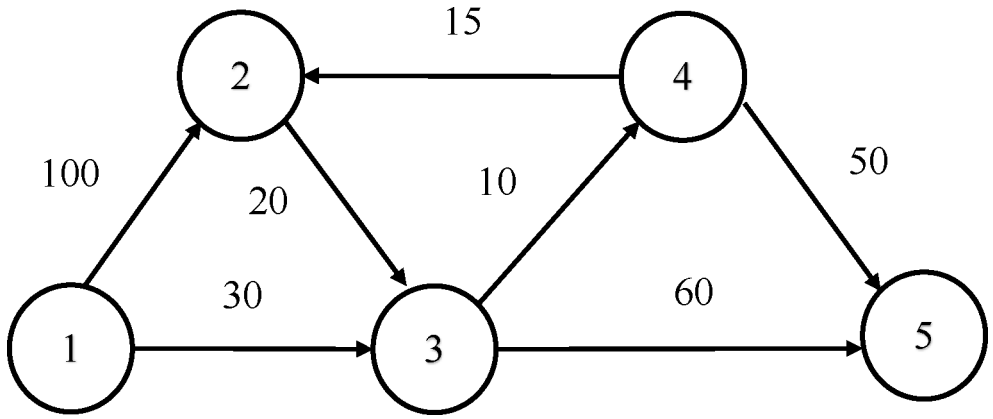


Рисунок 7.2 – Транспортная сеть, состоящая из 5 городов

Решение задачи.

1 Назначаем узлу 1 постоянную метку [0, -].

2 Из узла 1 можно достичь узлов 2 и 3. Вычисляем метки для этих узлов, в результате получается таблица меток (таблица 7.2).

Среди узлов 2 и 3 узел 3 имеет наименьшее значение расстояния (u_3=30). Поэтому статус метки этого узла изменяется на «постоянная».

Таблица 7.2 – Расчет меток для узлов 2 и 3 из узла 1

Узел	Метка	Статус
1	[0, -]	Постоянная
2	[0+100, 1]=[100, 1]	Временная
3	[0+30,1]=[30,1]	←Временная

3 Из узла 3 можно попасть в узлы 4 и 5. Получаем следующий список узлов (таблица 7.3).

Временный статус метки [40, 3] узла 4 заменяется на постоянный (u_4=40).

4 Из узла 4 можно достичь узлов 2 и 5. После вычисления меток получим следующий список (таблица 7.4).

Таблица 7.3 – Расчет меток для узлов 4 и 5 из узла 3

Узел	Метка	Статус
1	[0, -]	Постоянная
2	[100, 1]	Временная
3	[30, 1]	Постоянная
4	[30+10, 3]=[40, 3]	←Временная
5	[30+60, 3]=[90, 3]	Временная

Таблица 7.4 – Расчет меток для узлов 2 и 5 из узла 4

Узел	Метка	Статус
1	[0, -]	Постоянная
2	[40+15, 4]=[55, 4]	←Временная
3	[30, 1]	Постоянная
4	[40, 3]	Постоянная
5	[90, 3] или [40+50, 4]=[90, 3]	Временная

Временная метка [100,1], полученная узлом 2 на втором шаге, изменена на [55,4]. Это указывает на то, что найден более короткий путь к этому узлу (проходящий через узел 4).Узел 5 получает две метки с одинаковым значением расстояния S_r=90.

5 Из узла 2 можно перейти только в узел 3, но он имеет постоянную метку, которую нельзя изменить. Поэтому на данном шаге получаем список меток, как и на предыдущем шаге, но с единственным изменением: метка узла 2 получает статус постоянной. С временной меткой остается только узел 5, но так как из этого узла нельзя попасть ни в какой другой, процесс вычислений заканчивается (таблица 7.5).

Таблица 7.5 – Решение задачи определения кратчайших расстояний

Узел	Метка	Статус
1	[0, -]	Постоянная
2	[55, 4]	Постоянная
3	[30, 1]	Постоянная
4	[40, 3]	Постоянная
5	[90, 3] или [90, 4]	Постоянная

Алгоритм Дейкстры совпадает с алгоритмом Прима, определяющего остовное дерево минимального веса. Алгоритм Дейкстры относится к «жадным» алгоритмам: на каждом шаге к списку выбранных вершин добавляется та из оставшихся вершин, расстояние до которой от начальной вершины меньше, чем для других оставшихся вершин. Алгоритм Дейкстры использует метод динамического программирования, т. к. на каждом шаге выбирается оптимальное значение, которое не пересчитывается [10].

7.3.2 Алгоритм Флойда

Общая задача определения кратчайших путей (нахождение для каждой пары вершин *(u,v)* пути от вершины *v* к вершине *w*) решается последовательно с применением алгоритма Флойда нахождения кратчайших путей для каждой вершины. Временная сложность алгоритма будет равна $O(n^3)$ при использовании матрицы смежности и $O(n * e * log_2(n))$ при использовании списков смежности.

Алгоритм Флойда выполняется за *n* проходов. После *k-го* прохода a_{ij} содержит значение наименьшей длины путей из вершины *i* к *j*, которые не проходят через вершины с номером, большим *k*, т. е. между вершинами *i* и *j* могут быть вершины только с номерами, меньшими или равными *k*. При *k-м* проходе элемент a_{ij} вычисляется как $A_k[i,j] = min(A_{k-1[i.j],A_{k-1}}[i,k] + A_{k-1}[k,j])$. Алгоритм Флойда реализуется посредством трёх вложенных циклов и имеет сложность $O(n^3)$.

Алгоритм Флойда находит кратчайшие пути между любыми двумя узлами сети. Сеть представлена в виде квадратной матрицы с n строками и n столбцами. Элемент *(i,j)* равен расстоянию d_{ij} от узла *i* к узлу *j*, которое имеет конечное значение, если существует дуга *(i,j)*, иначе равен бесконечности [4].

Пусть даны три узла *i, k* и *j*. Заданы расстояния между ними d_{ij}, d_{ik}, d_{kj} (рисунок 7.3). Если выполняется неравенство $d_{ik} + d_{kj} < d_{ij}$, то путь i→j заменяется путем i→k→j. Такая замена называется треугольным оператором и выполняется в процессе выполнения алгоритма Флойда.

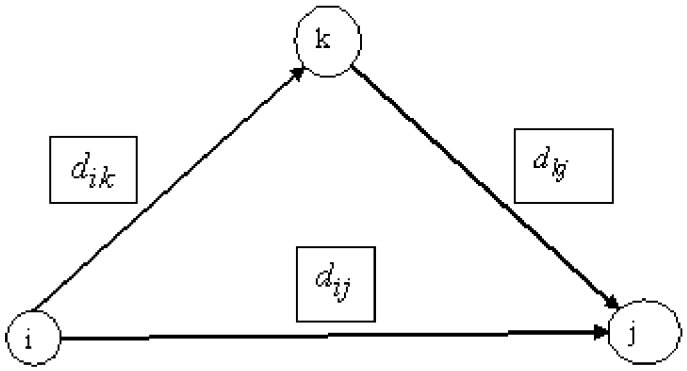


Рисунок 7.3 – Треугольный оператор Флойда

Алгоритм Флойда включает следующие этапы

Этап 1. Определяем начальную матрицу расстояний D_0 и матрицу последовательности узлов S_0. Диагональные элементы обеих матриц помечаются знаком «–», показывающим, что эти элементы в вычислениях не участвуют. Полагаем *k=1*.

Этап 2. Задаем строку *k* и столбец *k* как ведущую строку и ведущий столбец. Рассматриваем возможность применения треугольного оператора ко всем элементам d_{ij} матрицы D_{k-1}. Если выполняется неравенство $d_{ik} + d_{kj} < d_{ij}$, $(i \neq k, j \neq k, i \neq j)$ тогда выполняются следующие действия:

2.1 Создаем матрицуD_k путем замены в матрице D_{k-1} элемента d_{ij} на сумму $d_{ik} + d_{kj}$.

2.2 Создаем матрицу S_k путем замены в матрице S_{k-1} элемента s_{ij} на *k*. Полагаем *k=k+1* и повторяем шаг *k*.

Пояснение алгоритма Флойда приведено на рисунке 7.4. Строка *k* и столбец *k* являются ведущими. Строка *i* – любая строка с номером от 1 до *k–1*, а строка *p* – произвольная строка с номером от *k+1* до *n*. Столбец *j* представляет любой столбец с номером от 1 до *k–1*, а столбец *q* – произвольный столбец с номером от *k+1* до *n*. Треугольный оператор выполняется следующим образом. Если сумма элементов ведущих строки и столбца (квадраты) меньше элементов, находящихся на пересечении столбца и строки (круги), соответствующих рассматриваемым ведущим элементам, то расстояние (элемент в круге) заменяется на сумму расстояний, представленных ведущими элементами [10].

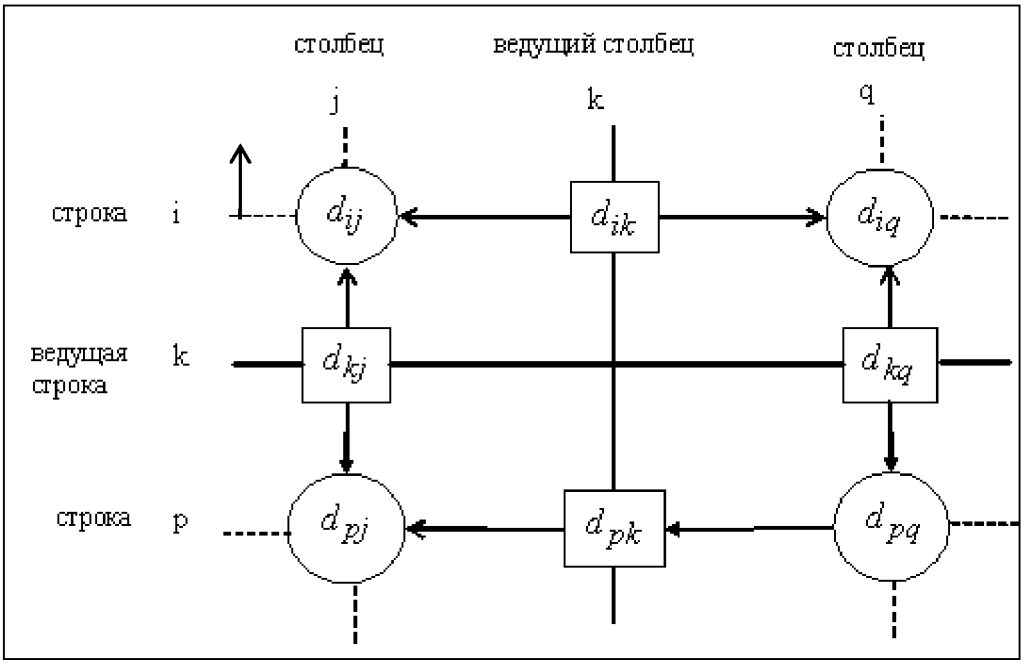


Рисунок 7.4 – Матрица D_{k-1} на k шаге алгоритма Флойда

После реализации n шагов алгоритма определение кратчайшего пути по матрицам D_n и S_n, кратчайшего пути между узлами i и j выполняется по правилам:

1 Расстояние между узлами i и j равно элементу d_{ij} в матрице Dn.

2 Промежуточные узлы пути от узла i к узлу j определяем по матрице S_n. Пусть $s_{ij}=k$, тогда имеем путь $i \rightarrow k \rightarrow j$. Если далее $s_{ik}=\boldsymbol{k}$ и $s_{kj}=\boldsymbol{j}$, тогда считаем, что весь путь определен, так как найдены все промежуточные узлы. В противном случае повторяем описанную процедуру для путей от узла i к узлу k и от k к узлу j.

Пример нахождения кратчайшего пути по алгоритму Флойда.

Постановка задачи. На рисунке 7.5 изображена сеть в виде графа. Расстояния между узлами сети в километрах проставлены возле соответствующих ребер. Ребро *(3,5)* ориентировано, поэтому не допускается движение от узла *5* к узлу *3*. Все остальные ребра допускают движение в обе стороны. Найти кратчайшие пути между любыми двумя узлами [10].

Решение задачи.

1 Строятся начальные матрицы D_0 (таблица 7.6) и S_0 (таблица 7.7). Матрица D_0 симметрична, за исключением пары элементов d_{35} и d_{53}, где $d_{53} = \infty$, поскольку невозможен переход от узла *5* узлу *3*.

2 В матрице D_0 выделены ведущие строка и столбец *(k=1)*. Элементы d_{23} и d_{32} можно улучшить с помощью треугольного оператора. Получаем матрицы D_1 (таблица 7.8) и S_1 (таблица 7.9):

- заменяем d_{23} на $d_{21} + d_{13} = 3 + 10 = 13$ и устанавливаем $s_{23} = 1$.
- заменяем d_{32} на $d_{31} + d_{12} = 10 + 3 = 13$ и устанавливаем $s_{32} = 1$.

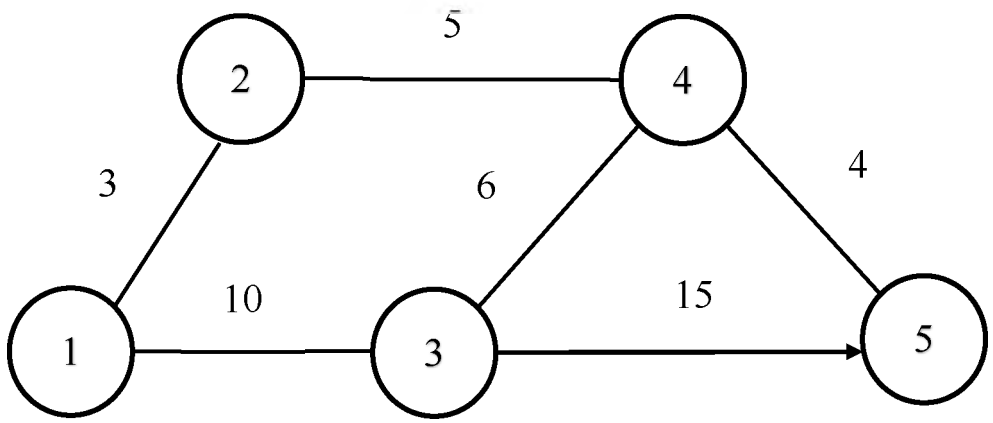


Рисунок 7.5 – Сеть, представленная в виде графа
с весовыми коэффициентами

Таблица 7.6 – Начальная матрица D_0

Строки/Столбцы	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

Таблица 7.7 – Начальная матрица S_0

Строки/Столбцы	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

Таблица 7.8 – Матрица D_1

Строки/Столбцы	1	2	3	4	5
1	-	**3**	**10**	∞	∞
2	3	-	13	5	∞
3	10	13	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

Таблица 7.9 – Матрица S_1

Строки/Столбцы	1	2	3	4	5
1	-	2	3	4	5
2	1	-	1	4	5
3	1	1	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

3 Полагаем *k=2*. В матрице D_1 выделены ведущие строка и столбец. Треугольный оператор применяется к элементам матриц D_1 и S_1, выделенных двойной рамкой. В результате получаем матрицы D_2 (таблица 7.10) и S_2 (таблица 7.11).

Таблица 7.10 – Матрица D_2

Строки/Столбцы	**1**	**2**	**3**	**4**	**5**
1	-	3	10	8	∞
2	3	-	13	5	∞
3	10	13	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

Таблица 7.11 – Матрица S_2

Строки/Столбцы	**1**	**2**	**3**	**4**	**5**
1	-	2	3	2	5
2	1	-	1	4	5
3	1	1	-	4	5
4	2	2	3		5
5	1	2	3	4	-

4 Полагаем *k=3*. В матрице D_2 выделены ведущие строка и столбец. Треугольный оператор применяется к элементам матриц D_2 и S_2, выделенных двойной рамкой. В результате получаем матрицы D_3 (таблица 7.12) и S_3 (таблица 7.13).

Таблица 7.12 – Матрица D_3

Строки/Столбцы	1	2	3	4	5
1	-	3	10	8	25
2	3	-	13	5	28
3	10	13	-	6	15
4	8	5	6	-	4
5	∞	∞	∞	4	-

Таблица 7.13 – Матрица S_3

Строки/Столбцы	1	2	3	4	5
1	-	2	3	2	3
2	1	-	1	4	3
3	1	1	-	4	5
4	2	2	3	-	5
5	1	2	3	4	-

5 Полагаем *k=4*. В матрице D_3 выделены ведущие строка и столбец. Треугольный оператор применяется к элементам матриц D_3 и S_3, выделенных двойной рамкой. В результате получаем матрицы D_4 (таблица 7.14) и S_4 (таблица 7.15).

Таблица 7.14 – Матрица D_4

Строки/Столбцы	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	11	-	6	10
4	8	5	6	-	4
5	12	9	10	4	-

Таблица 7.15 – Матрица S_4

Строки/Столбцы	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	4	-	4	4
4	2	2	3	-	5
5	4	4	4	4	-

6 Полагаем *k=5*. В матрице D_4 выделены ведущие строка и столбец. Никаких действий на шаге не выполняем. Вычисления закончены.

Конечные матрицы D_4 и S_4 содержат всю информацию, необходимую для определения кратчайших путей между любыми двумя узлами сети. Например, кратчайшее расстояние между узлами *1* и *5* равно *12* километрам (d_{15}*=12)*.

Сегмент маршрута *(i,j)* состоит из ребра *(i,j)*, если s_{ij}*=j*. Иначе узлы **i** и *j* связаны, по крайней мере, через один промежуточный узел. Например, $\boldsymbol{s_{15}}$*=4* и $s_{45} = 5$, сначала кратчайший маршрут между узлами *1* и *5* будет иметь вид *1→4→5*. Но так как $\boldsymbol{s_{14}}$*=4*, узлы *1* и *4* в определяемом пути не связаны одним ребром (но в исходной сети они могут быть связаны непосредственно). Определяем промежуточный узел (узлы) между первым и четвертым узлами: s_{14} *=2* и s_{24} *=4*, поэтому маршрут *1→4* заменяем *1→2→4*. Поскольку s_{12}*=2* и s_{24}=4, других промежуточных узлов нет. Комбинируя определенные сегменты маршрута, получаем кратчайший путь от узла *1* до узла *5*: *1→2→4→5*. Длина пути равна *12* километрам [10].

7.4 Минимальное остовное дерево

Остовное дерево связного неориентированного графа G = *(V,E)* с *n* вершинами – неориентированное дерево; содержащее все *n* вершин и *(n*–1) ребер графа. Остовное дерево связывает все вершины графа, и из каждой вершины графа можно попасть в любую другую. В полном графе *с n* вершинами имеется n^{n-2} остовных деревьев [1; 4].

Неориентированный граф изображен на рисунке 7.6.

Остовные деревья графа изображены на рисунке 7.7.

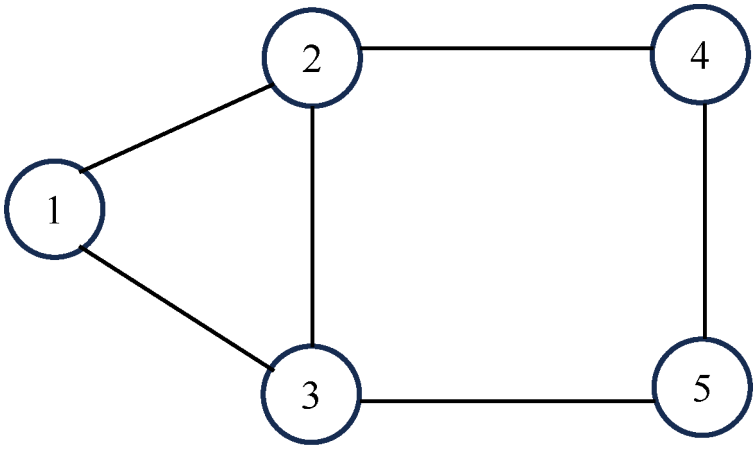


Рисунок 7.6 – Граф

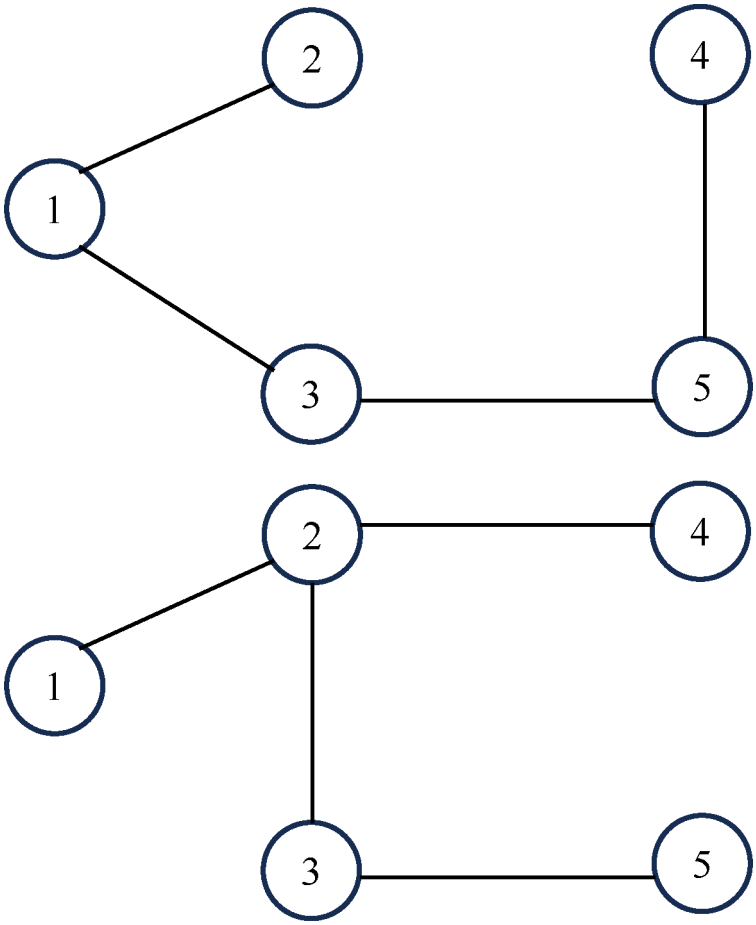


Рисунок 7.7 – Остовные деревья графа

Пусть G = (*V,E)* – связный неориентированный граф и *T = (V,F)* – остовное дерево для него. Тогда:

а) для любых двух вершин v_i и v_j путь между ними единствен;

б) если к остовному дереву *m* добавить ребро из *(E-F)*, т. е. из оставшихся, не включенных в дерево ребер графа, то возникнет ровно один цикл и *T* перестанет быть деревом.

Остовное дерево ориентированного графа – дерево, в котором одна из вершин графа связана со всеми остальными.

Задачи нахождения остовных деревьев графа:

- нахождение остовного дерева [4];
- нахождение минимального и максимального остовного дерева.

7.4.1 Алгоритмы поиска в ширину и глубину на графах

При решении задач, связанных с графами, необходим систематический обход вершин и дуг (ребер) графа. Широкое применение получили два метода обхода:

- поиск в глубину;
- поиск в ширину.

При *поиске в глубину* осуществляется регулярный обход графа по правилам:

1 Находясь в вершине *v*, нужно двигаться в любую другую w, ранее не пройденную вершину (если таковая найдется), одновременно запоминая дугу *z*, по которой мы впервые попали в вершину *w*.

2 Если из вершины *v* мы не можем попасть в ранее не пройденную вершину или таковой вообще нет, то мы возвращаемся в вершину дуги *z*, из которой впервые попали в вершину *v*, и продолжаем поиск в глубину из этой вершины.

При выполнении обхода по этим правилам мы стремимся проникнуть вглубь графа как можно дальше, затем отступаем на шаг и снова стремимся пройти вперед.

При *поиске в ширину* последовательно просматриваются, начиная с заданной вершины, все вершины дерева на уровне $k = 1$, затем на уровне k + 1 и т. д.

Сложность алгоритма поиска в ширину, как и алгоритма поиска в глубину, равна *O(e)*, *e* – число ребер графа [4].

7.4.2 Алгоритмы построения остовного дерева

Пусть G = *(V, E)* – связный взвешенный неориентированный граф, для которого задана матрица смежности, отображающая веса ребер в числа (вещественные или целые). *Стоимость (вес)* остовного дерева определяется как сумма стоимостей (весов) его ребер. Цель – найти для графа *G* остовное дерево наименьшей стоимости (минимального веса). Полный граф с *n* вершинами содержит n^{n-2} остовных деревьев. Поиск каждого остовного дерева занимает *O(e)* времени. В полном дереве *e=n*(n-l)/2*. Тогда решение задачи прямым поиском имело бы сложность $O(n^{n-2}n(n-1)) = O(n^n)$.

7.4.2.1 Алгоритм Прима

Наиболее простым алгоритмом поиска остовного дерева минимального веса является алгоритм Прима, похожий на алгоритм Дейкстры поиска кратчайшего пути. Алгоритм Прима включает этапы:

1 Вначале выбираем некоторую вершину *v,* остальные *(m*–1) вершины графа отмечаются как невыбранные.

2 Определяются веса между выбранной вершиной *v* и остальными невыбранными вершинами.

3 Выбираем вершину с наименьшим весом до нее, фиксируем выбранные ребро и вес.

4 Выбранную вершину исключаем из перечня невыбранных, число невыбранных вершин уменьшаем на 1.

5 Пункты 1-4 повторяем до тех пор, пока не будут выбраны все вершины, *(m*–1) раз [4].

7.4.2.2 Алгоритм Крускала

Пусть имеется связный взвешенный граф *G(V,E)* с *n* вершинами. Построение остовного дерева минимального веса начинаетсяс графа *T=(V,0),* имеющего только *n* вершин без ребер. Каждая вершина, таким образом, оказывается связанной только с самой собой. Построение дерева сводится к формированию набора связных компонентов, постепенным объединением которых формируется остовное дерево [4].

Упорядочим ребра множества E в порядке возрастания их веса (стоимости). Выберем ребро с наименьшим весом C_1 и включим в граф T. Теперь в графе T $(n–1)$ компонент содержит только по одной вершине, один компонент содержит две вершины и одно ребро. Выбираем следующее наименьшее ребро. Если оно связывает две вершины из разных компонент, то это ребро добавляется в граф T. Если же ребро связывает две вершины из одного компонента, то такое ребро отбрасывается, так как его добавление в связный компонент, являющийся свободным деревом, приведет к образованию цикла. Когда все вершины графа будут принадлежать одному компоненту, построение остовного дерева минимального веса T с $(n–1)$ ребрами заканчивается [4].

7.5 Контрольные вопросы

1 Какое существует определение понятия графа?

2 Какие этапы имеет алгоритм метода Дейкстры?

3 Какие этапы имеет алгоритм метода Флойда?

4 Какое существует определение понятия «остовное дерево»?

5 Какие этапы имеет алгоритм метода Крускала?

6 Какие этапы включает алгоритм метода Прима?

7 Какое существует определение связного неориентированного графа?

8 Какое существует определение ориентированного графа?

9 Какие этапы включает алгоритм обхода вершин и дуг (ребер) графа в глубину?

10 Какие этапы включает алгоритм обхода вершин и дуг (ребер) графа в ширину?

7.6 Варианты заданий 1

Разработать программное приложение на языке программирования Visual C++, реализующее алгоритм метода Дейкстры.

Вариант 1

Компания Bell Electric Company построила коммуникационную сеть между двумя приемно-передающими станциями 1 и 7. На рисунке 7.8 коммуникационная сеть представлена в виде ориентированного графа. Вес дуги – расстояние в километрах. Определить кратчайшее расстояние между узлом 1 и всеми остальными узлами коммуникационной сети [6; 7].

Вариант 2

Транспортная компания осуществляет перевозку автомобилей. На рисунке 7.9 представлена транспортная сеть в виде ориентированного графа с 9 узлами.

Вес дуги – расстояние в километрах. Определить кратчайшее расстояние между узлом 1 и всеми остальными узлами транспортной сети.

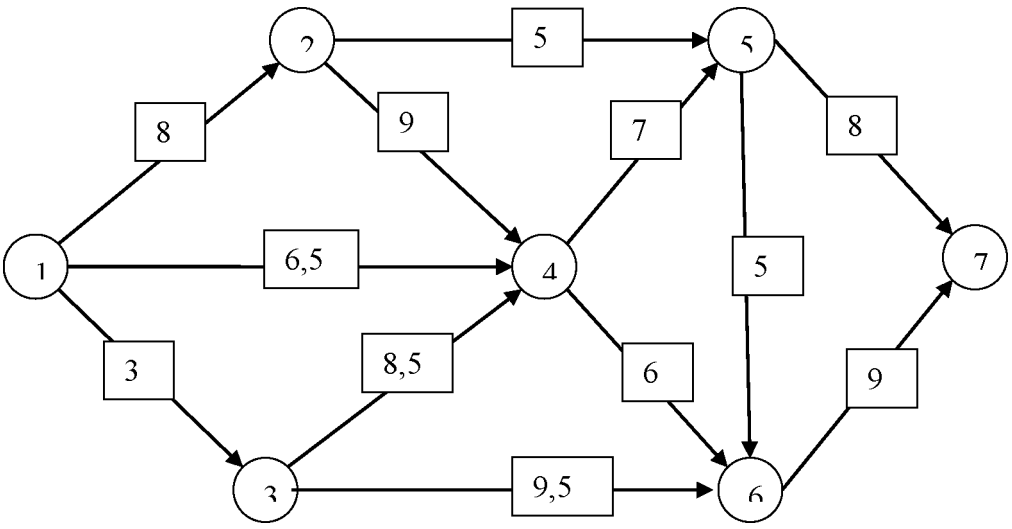


Рисунок 7.8 – Коммуникационная сеть компании Bell Electric Company

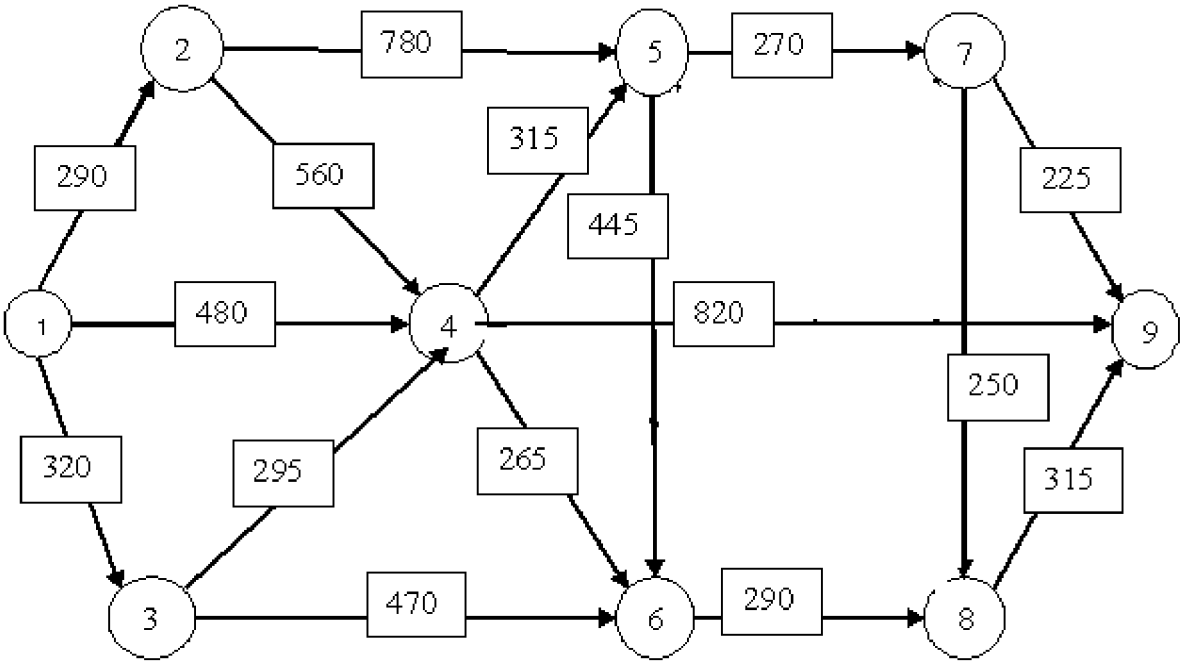


Рисунок 7.9 – Транспортная сеть перевозки автомобилей

Вариант 3

Торговая компания имеет филиалы в 8 точках города. Транспортная сеть с указанием расстояний в километрах представлена на рисунке 7.10. Определить кратчайшие пути между узлом 1 и всеми остальными узлами транспортной сети [6; 7].

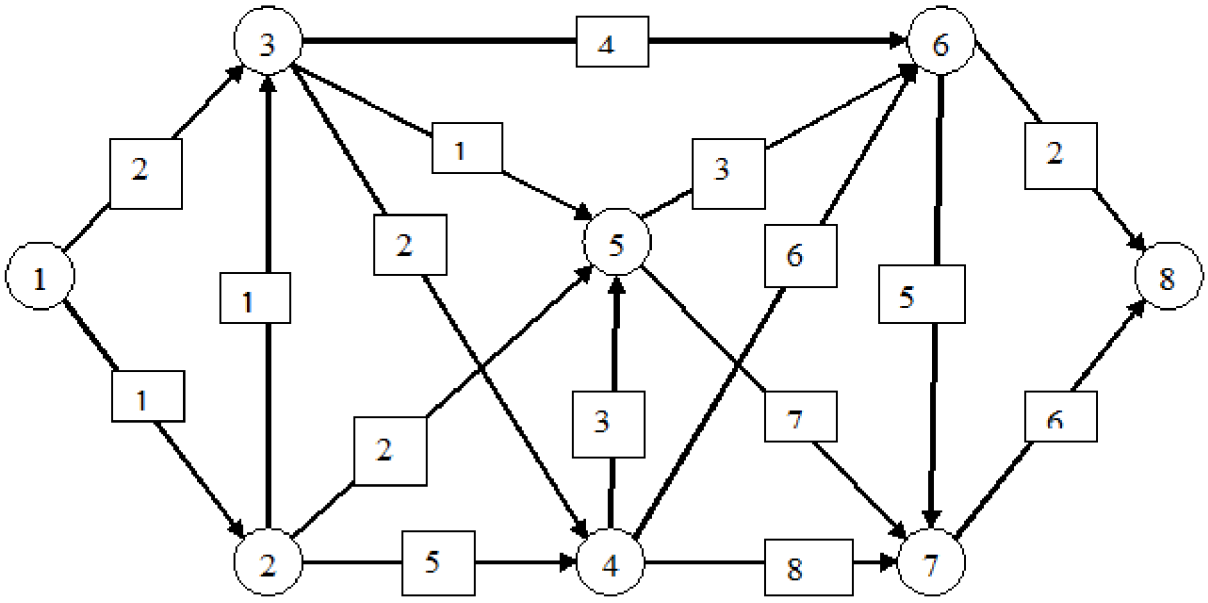


Рисунок 7.10 – Транспортная сеть торговой компании

Вариант 4

Торговая компания имеет филиалы в 9 населённых пунктах. Сеть филиалов представлена в виде ориентированного графа (рисунок 7.11). Вес дуги ориентированного графа – расстояние в километрах. Определить кратчайшие пути между узлом 1 и всеми остальными узлами графа.

Вариант 5

Логистическая компания проектирует (нефтепровод) газопровод между 10 населенными пунктами. Транспортная сеть показана на рисунке 7.12. Расстояния указаны в километрах. Определить кратчайший путь между узлами 1 и всеми остальными узлами [6; 7].

7.7 Варианты заданий 2

Разработать программное приложение на языке программирования Visual C++, реализующее алгоритм метода Флойда.

Вариант 1

Телефонная компания обслуживает 6 удаленных друг от друга районов, которые связаны сетью, показанной на рисунке 7.13. Расстояние на схеме сети указано в милях. Компании необходимо определить наиболее эффективные маршруты пересылки сообщений между любыми двумя районами [6; 7].

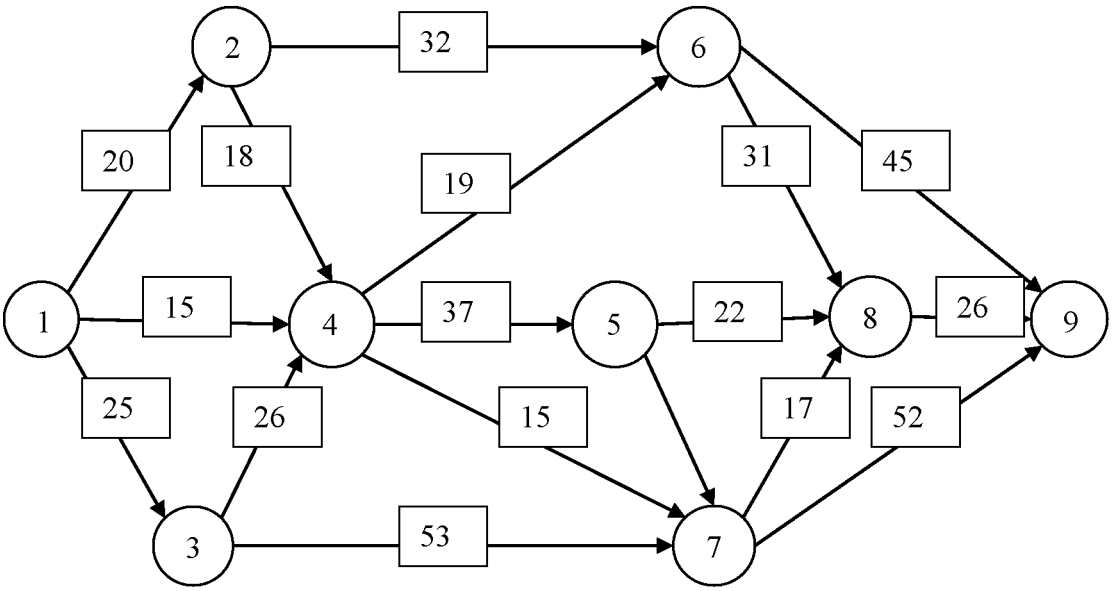


Рисунок 7.11 – Транспортная сеть филиалов торговой фирмы

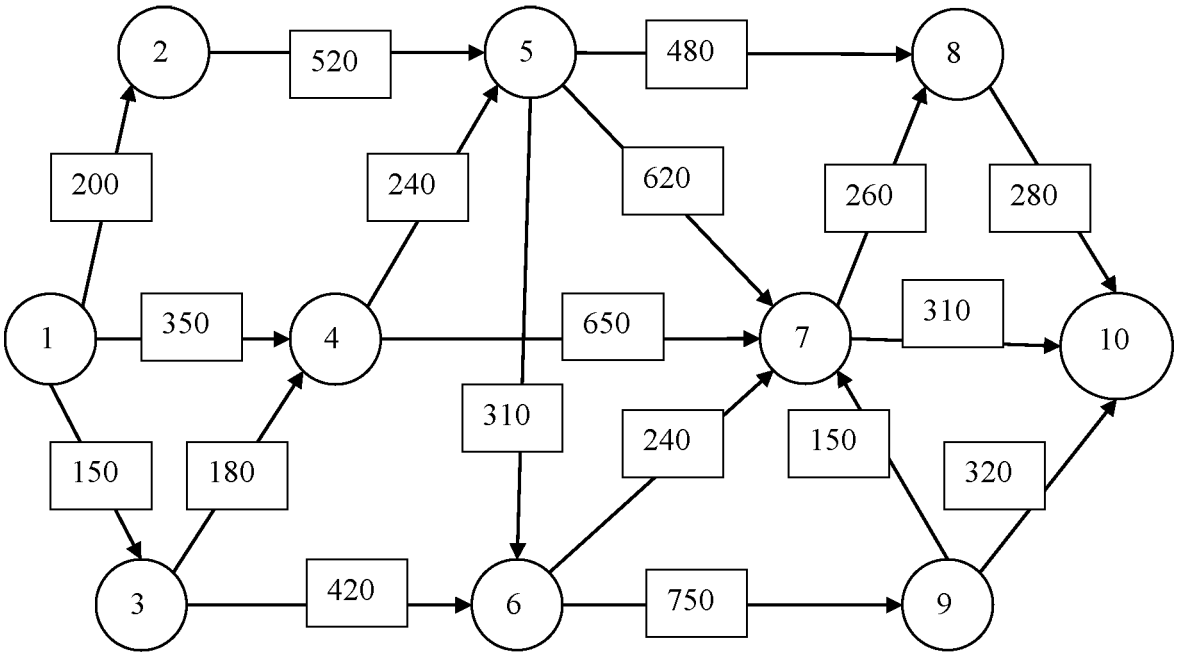


Рисунок 7.12 – Транспортная сеть газопровода, спроектированная логистической компанией

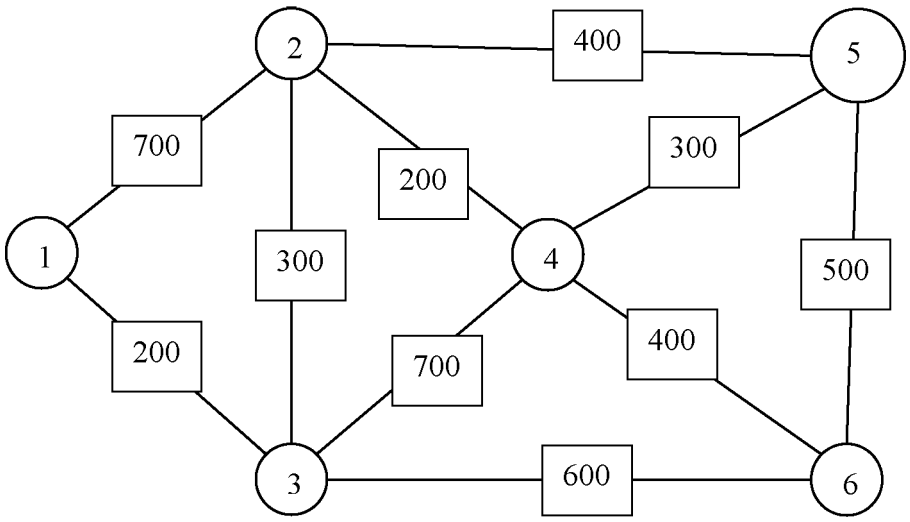


Рисунок 7.13 – Сеть телефонной компании

Вариант 2

Частное охранное предприятие осуществляет перевозку денег от ювелирных магазинов, представленных графом с 8 узлами. Расстояние между магазинами указано в километрах. Определить кратчайшие маршруты между ювелирными магазинами (рисунок 7.14).

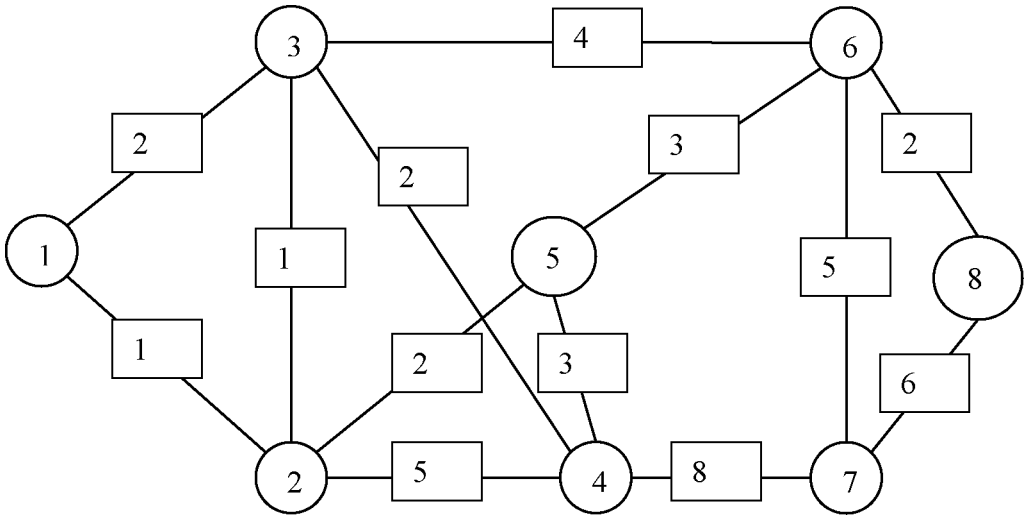


Рисунок 7.14 – Транспортная сеть перевозок денежных средств

Вариант 3

Торговая фирма производит продажу бытовой техники в 7 городах страны. Сеть филиалов представлена в виде графа на рисунке 7.15. Расстояние между городами указано в тысячах километрах. Определить кратчайшие маршруты между городами, в которых расположены филиалы торговой фирмы [6; 7].

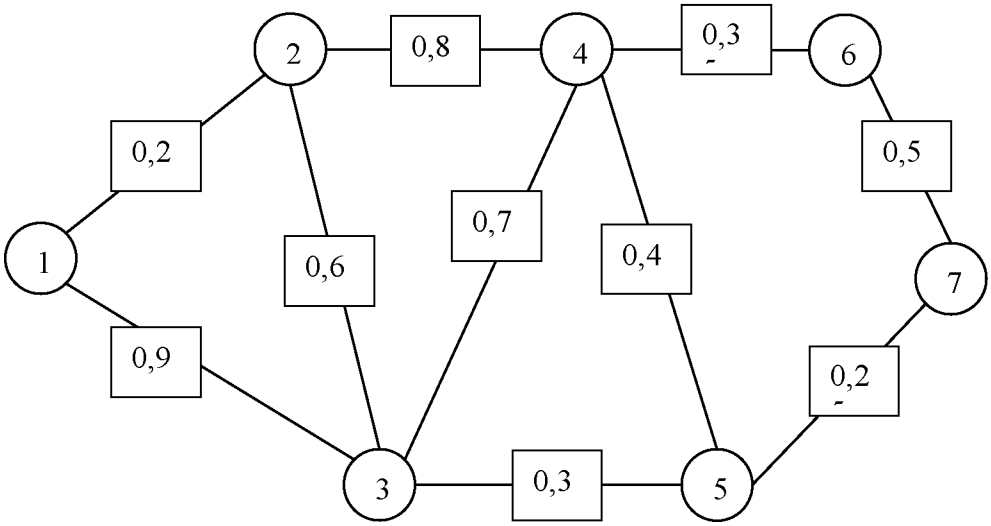


Рисунок 7.15 – Транспортная сеть филиалов торговой фирмы

Вариант 4

Компания грузоперевозок осуществляет доставку товаров в 7 городов страны. Сеть доставки показана на рисунке 7.16.

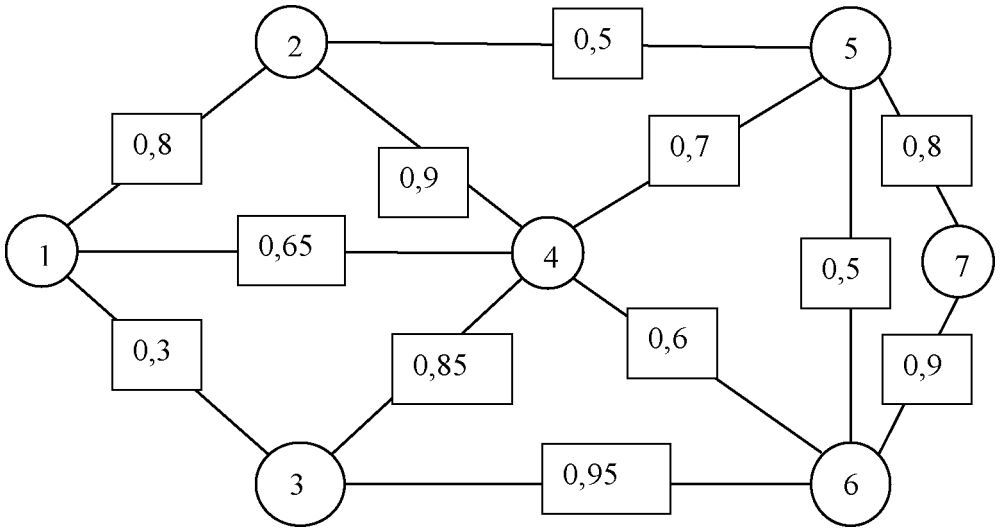


Рисунок 7.16 – Транспортная сеть доставки грузов компании

Расстояние между узлами графа выражено в тысячах километров. Определить кратчайшие пути между узлами графа [6; 7].

8 КУРСОВАЯ РАБОТА

8.1 Назначение, цели и задачи курсовой работы

Проектируемое визуальное приложение курсовой работы формализует разработанный студентом алгоритм задачи, проанализированный посредством функции оценки сложности.

Основная учебная цель выполнения курсовой работы – повышение теоретических знаний и практических навыков в разработке, анализе и программной реализации алгоритмов решения задач и выборе структур данных.

Основные задачи, решаемые студентом в процессе выполнения курсовой работы:

- разработка алгоритма и обоснование выбора структур данных;
- анализ алгоритма решения задачи;
- программная реализация алгоритма;
- проведение экспериментального исследования алгоритма и анализ его результатов;
- документирование курсовой работы в соответствии с установленными требованиями.

8.2 Требования к курсовой работе

8.2.1 Требования к функциональным характеристикам

Проектируемая система должна обеспечивать выполнение следующих основных функций:

- ввод исходных данных задачи;
- расчет параметров;
- оценка сложности реализации алгоритма по временным и объемным параметрам;
- хранение исходных данных и результатов расчетов с возможностью их загрузки для повторной обработки;
- вывод данных.

8.2.2 Требования к эксплуатационным характеристикам

- модульность;
- расширяемость.

8.2.3 Требования к программному обеспечению

-интегрированная среда разработки Microsoft Visual Studio 2022 Community;

- язык программирования Visual C++.

8.2.4 Требования к содержанию курсовой работы

К защите курсовой работы должны быть представлены визуальное приложение и альбом, включающий проектные, программные и эксплуатационные документы:

- опись альбома,
- пояснительная записка,
- спецификация,
- описание программы,
- руководство пользователя,
- руководство программиста,
- текст программы (на машинном носителе).

Пояснительная записка включает разделы и подразделы:

1 Аналитический обзор.
2 Описание алгоритма решения задачи.
3 Анализ алгоритма решения задачи.
3.1 Характеристики алгоритма.
3.2 Функции сложности алгоритмов.
3.3 Виды функций сложности алгоритмов.
3.4 Сравнение асимптотического поведения функции.
3.5 Базовое правило использования О-большого.
3.6 Анализ функции сложности.
4 Описание структуры программного комплекса.
5 Разработка диаграммы классов.
6 Описание структур данных.

7 Описание методики проведения экспериментального исследования.

7.1 Математические методы оценивания времени выполнения алгоритмов.

7.2 Классическая регрессионная модель парной (множественной) корреляции.

8 Описание и анализ результатов проведенного исследования.

9 Вывод по результатам проведенного исследования.

В начало и в конец пояснительной записки включаются введение и библиографический список соответственно [11].

Требования к структуре документов определены соответствующими стандартами ЕСПД.

Пояснительная записка оформляется в соответствии с требованиями к оформлению документации курсовых и дипломных проектов [12].

8.3 Пример выполнения курсовой работы на ПЭВМ

Скриншоты курсовой работы по дисциплине «Алгоритмы и структуры данных» приведены на рисунках 8.1 – 8.10.

Рисунок 8.1 – Информационное окно курсовой работы по дисциплине «Алгоритмы и структуры данных»

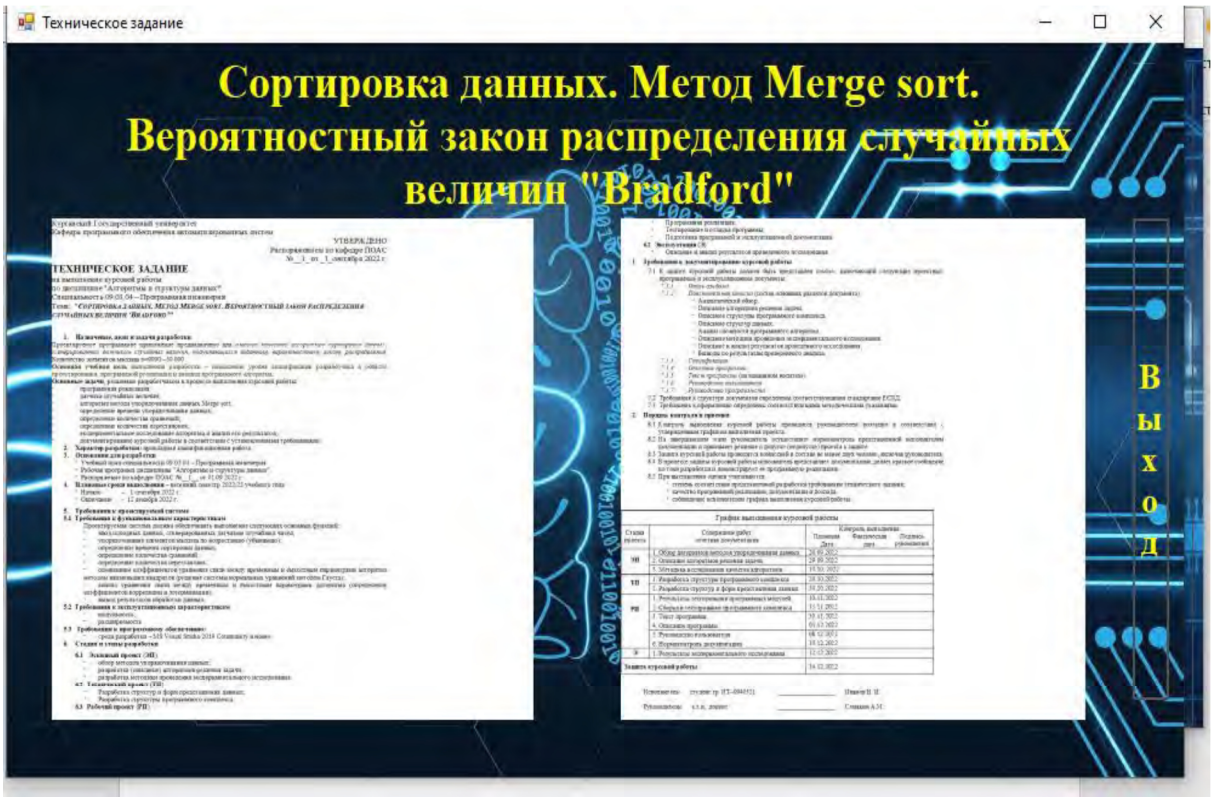


Рисунок 8.2 – Окно с техническим заданием на курсовую работу

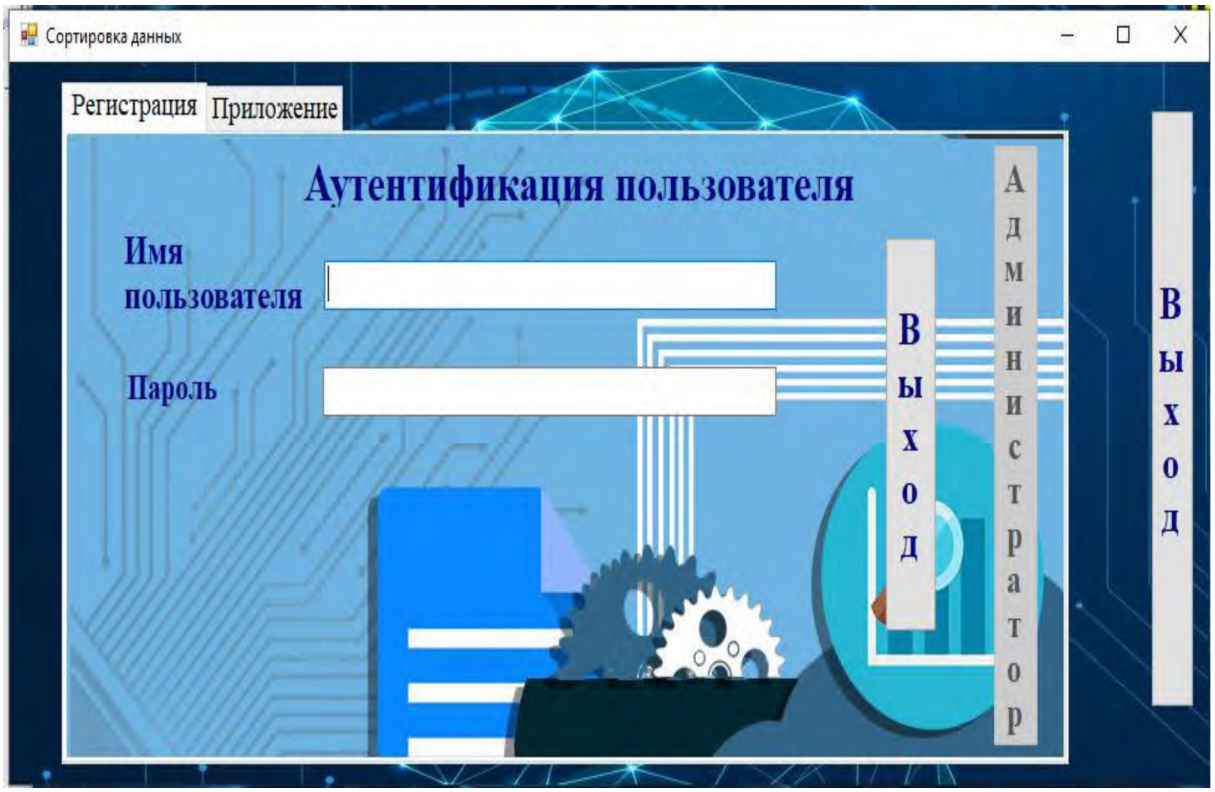


Рисунок 8.3 – Окно аутентификации пользователя
с неактивной кнопкой «Администратор»

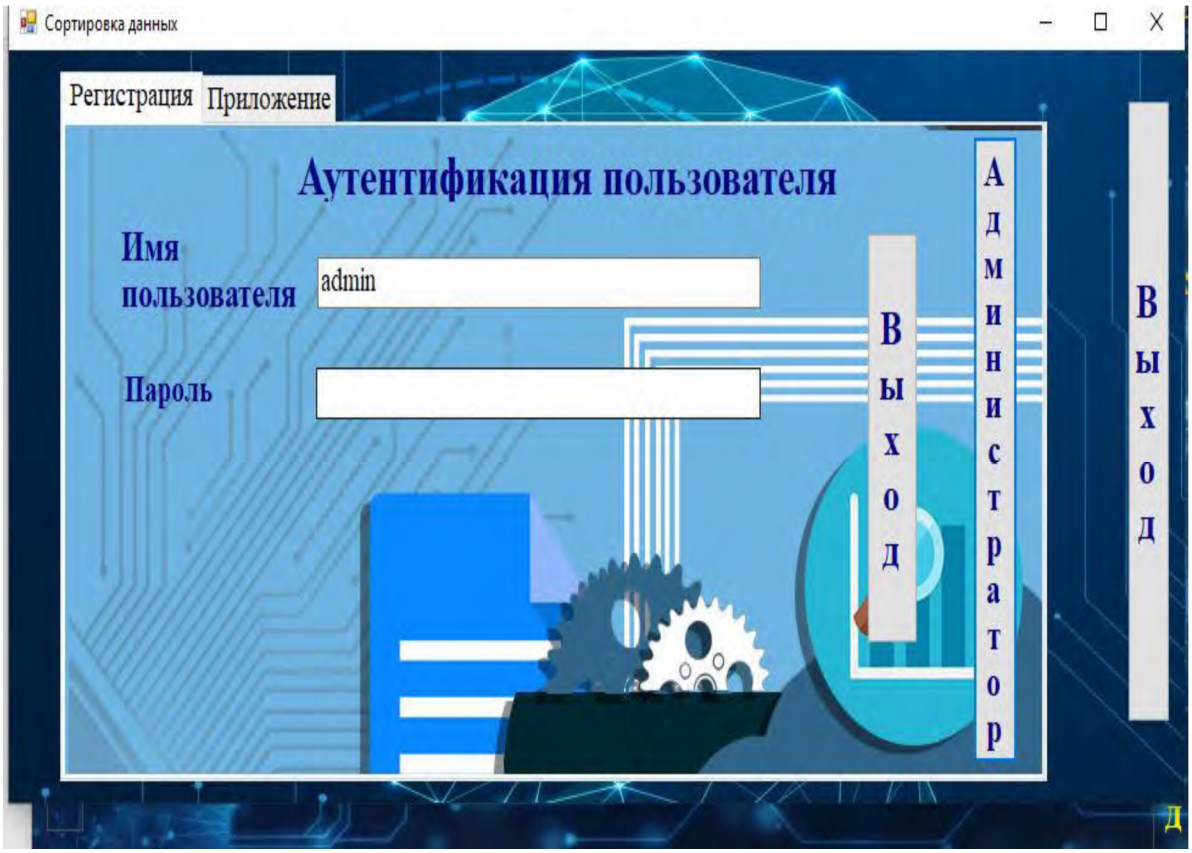


Рисунок 8.4 – Окно аутентификации пользователя
с активной кнопкой «Администратор»

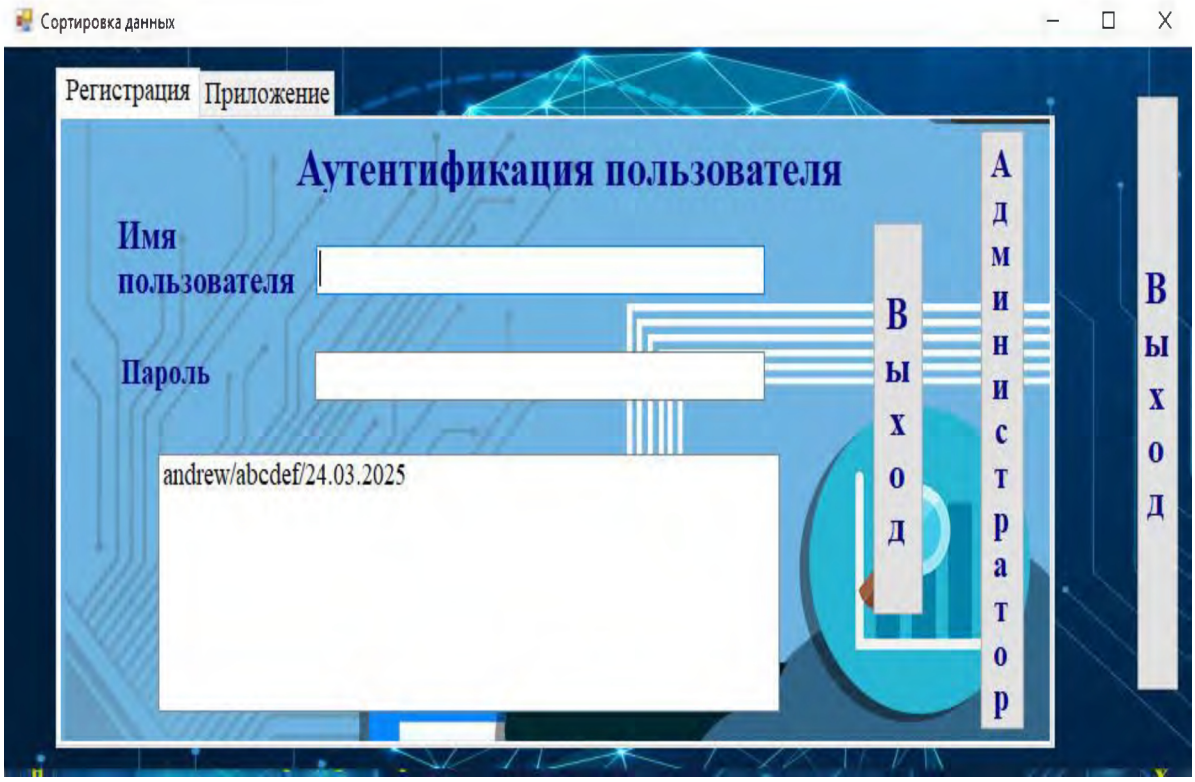


Рисунок 8.5 – Окно аутентификации пользователя
с данными пользователя

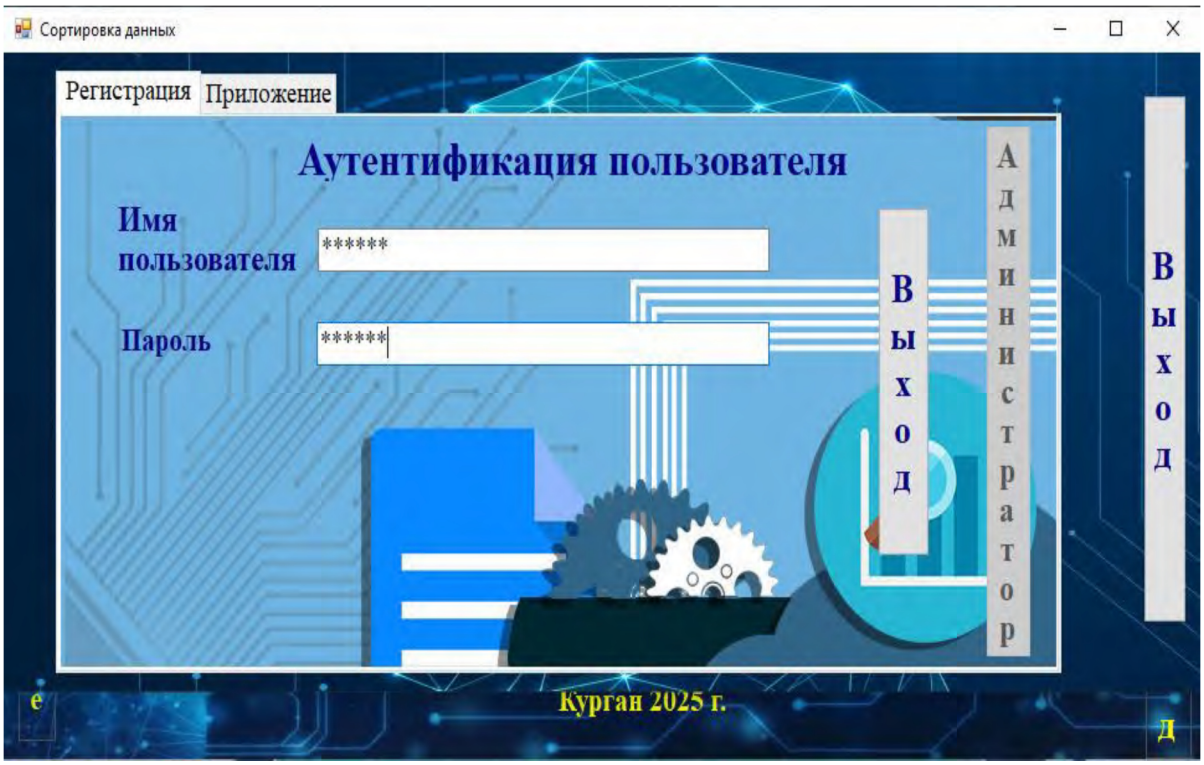


Рисунок 8.6 – Окно аутентификации пользователя
с введенными данными пользователя

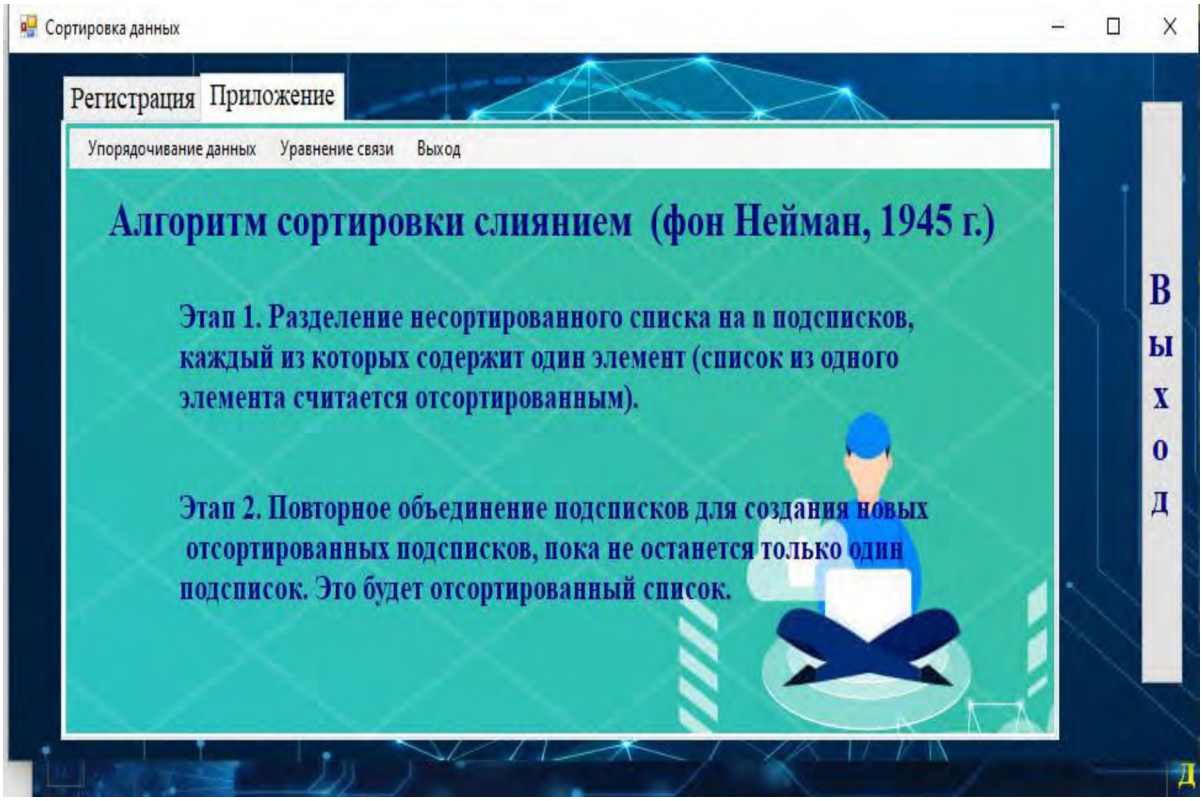


Рисунок 8.7 – Алгоритм метода сортировки слиянием

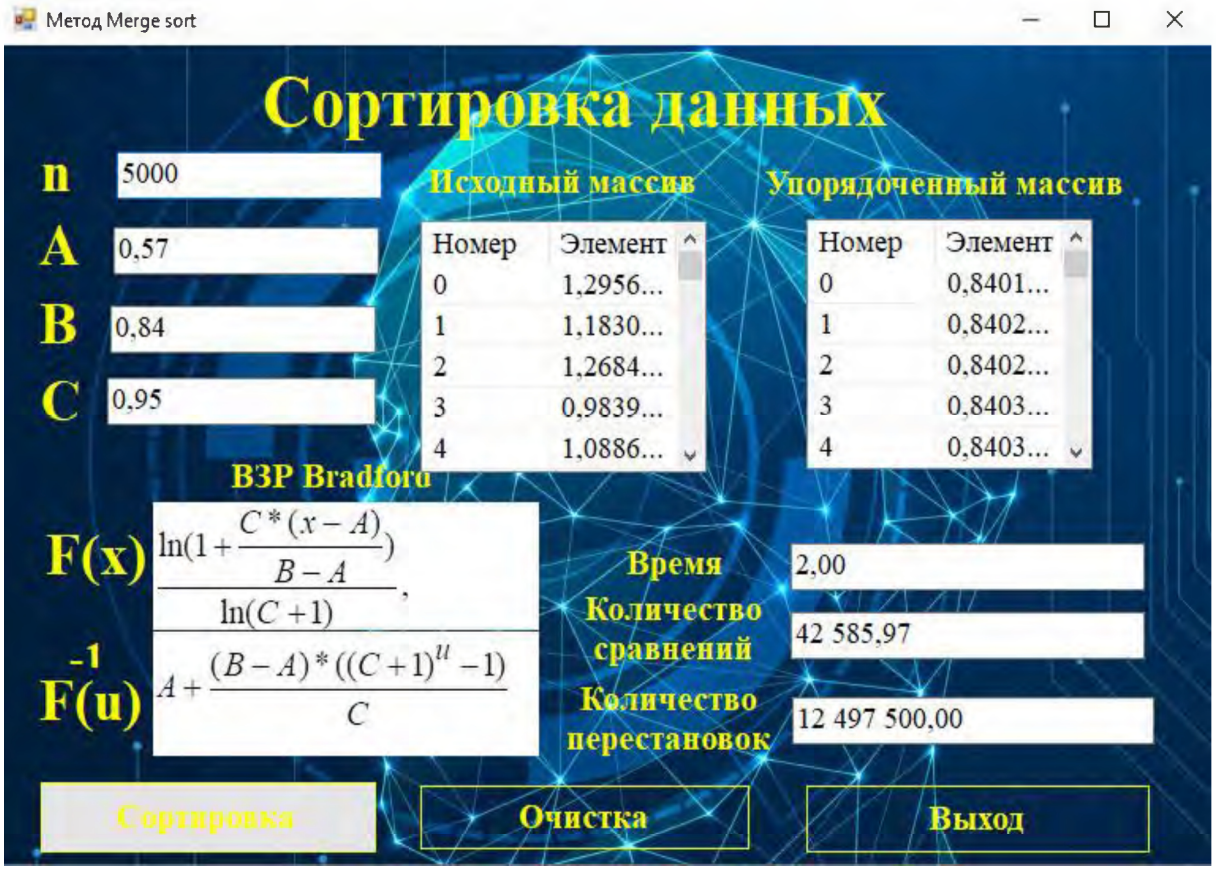


Рисунок 8.8 – Сортировка данных

Рисунок 8.9 – Оценивание коэффициентов уравнения связи между временным и ёмкостным параметрами программного алгоритма

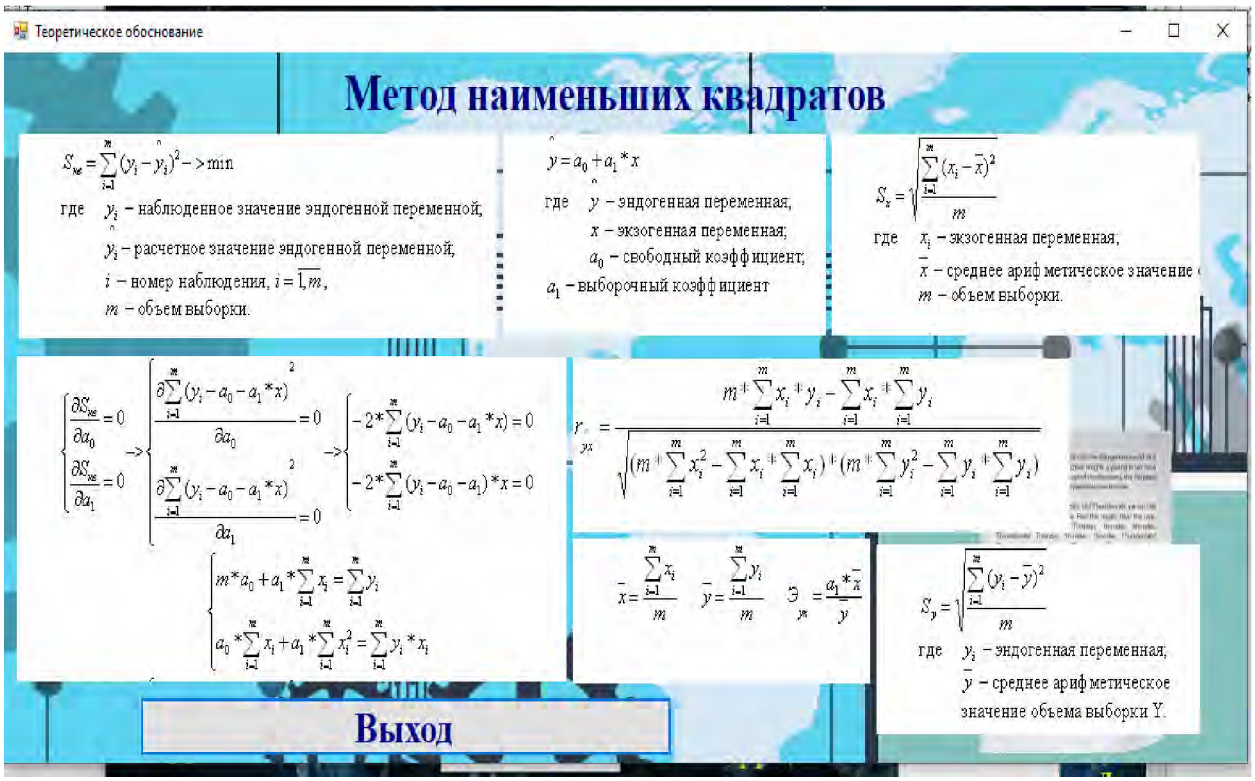


Рисунок 8.10 – Математическое обоснование программного алгоритма

8.4 Варианты заданий 1

1 Сортировка данных. Метод Timsort. Вероятностный закон распределения случайных величин ‘Bradford’.

2 Сортировка данных. Метод Bitonic sort. Вероятностный закон распределения случайных величин ‘Burr’.

3 Сортировка данных. Метод MSD. Вероятностный закон распределения случайных величин ‘Cauchy’.

4 Сортировка данных. Метод LSD. Вероятностный закон распределения случайных величин ‘Exponential’.

5 Сортировка данных. Метод Bucket sort. Вероятностный закон распределения случайных величин ‘ExtremeLB’.

6 Сортировка данных. Метод Merge sort. Вероятностный закон распределения случайных величин ‘Fisk’.

7 Сортировка данных. Метод Bogosort. Вероятностный закон распределения случайных величин ‘Gumbel’.

8 Сортировка данных. Метод Introsort. Вероятностный закон распределения случайных величин ‘Laplace’.

9 Сортировка данных. Метод Cycle sort. Вероятностный закон распределения случайных величин ‘Logistic’.

10 Сортировка данных. Метод Partience sorting. Вероятностный закон распределения случайных величин ‘Pareto’.

11 Сортировка данных. Метод Smoothsort. Вероятностный закон распределения случайных величин ‘Reciprocal’.

12 Сортировка данных. Метод Strand sort. Вероятностный закон распределения случайных величин ‘Weibull’.

13 Сортировка данных. Метод Coctail sort. Вероятностный закон распределения случайных величин ‘Bradford’.

14 Сортировка данных. Метод Comb sort. Вероятностный закон распределения случайных величин ‘Burr’.

15 Сортировка данных. Метод Counting sort. Вероятностный закон распределения случайных величин ‘Cauchy’.

16 Сортировка данных. Метод Tournament sort. Вероятностный закон распределения случайных величин ‘Exponential’.

17 Сортировка данных. Метод Stooge sort. Вероятностный закон распределения случайных величин ‘ExtremeLB’.

18 Сортировка данных. Метод Timsort. Вероятностный закон распределения случайных величин ‘Cauchy’.

19 Сортировка данных. Метод Bitonic sort. Вероятностный закон распределения случайных величин ‘Exponential’.

20 Сортировка данных. Метод MSD. Вероятностный закон распределения случайных величин ‘Weibull’.

21 Сортировка данных. Метод LSD. Вероятностный закон распределения случайных величин ‘Bradford’.

22 Сортировка данных. Метод Bucket sort. Вероятностный закон распределения случайных величин ‘Burr’.

23Сортировка данных. Метод Merge sort. Вероятностный закон распределения случайных величин ‘Laplace’.

24 Сортировка данных. Метод Bogosort. Вероятностный закон распределения случайных величин ‘Pareto’.

25Сортировка данных. Метод Introsort. Вероятностный закон распределения случайных величин ‘Reciprocal’.

26 Сортировка данных. Метод Merge sort. Вероятностный закон распределения случайных величин ‘Bradford’.

27 Сортировка данных. Метод Introsort. Вероятностный закон распределения случайных величин ‘Cauchy’.

28 Сортировка данных. Метод Timsort. Вероятностный закон распределения случайных величин ‘Pareto’.

29 Сортировка данных. Метод Cycle sort. Вероятностный закон распределения случайных величин ‘Cauchy’.

30 Сортировка данных. Метод Coctail sort. Вероятностный закон распределения случайных величин ‘Pareto’.

8.5 Варианты заданий 2

1 Транспортная задача. Метод потенциалов.

2 Эвристические методы поиска решения на графах. Метод минимальной стоимости.

3 Динамическое программирование. Метод обратной прогонки.

4 Сортировка файлов простым слиянием.

5 Сортировка файлов естественным слиянием.

6 Транспортная задача. Распределительный метод.

7 Остовное дерево наименьшей стоимости. Алгоритм Прима.

8 Остовное дерево наименьшей стоимости. Алгоритм Борувки.

9 Алгоритм определения максимального потока сети.

10 Алгоритм минимизации стоимости потока в сети с ограниченной пропускной способностью.

11 Алгоритм определения критического пути.

12 Методы поиска решения на графах. Методы поиска в ширину.

13 Методы поиска решения на графах. Методы поиска в глубину.

14 Эвристические методы поиска решения на графах. Метод экстремума.

15 Алгоритм поиска оптимального кода. Алгоритм Хаффмана.

16 Динамическое программирование. Метод прямой прогонки.

17 Алгоритм поиска оптимального кода. Алгоритм Шеннона-Фено.

18 Алгоритм сжатия данных. Алгоритм «арифметическое кодирование».

19 Структуры данных и алгоритмы для внешней памяти. Внешние деревья поиска. B дерево (B tree).

20 Алгоритм метода ветвей и границ. Задача расшифровки криптограмм.

21 Остовное дерево наименьшей стоимости. Алгоритм Крускала.

22 Алгоритмы внешней сортировки. Многофазная сортировка.

23 Алгоритмы внешней сортировки. Каскадная сортировка.

24 Алгоритм внешней сортировки. Сбалансированное многопутевое слияние.

25 Алгоритмы на графах. Задача поиска кратчайшего отрицательно взвешенного пути при одном источнике (алгоритм Беллмана-Форда).

26 Алгоритм Йена.

27 Алгоритм Форда.

28 Оптимальное двоичное дерево поиска. Алгоритм Гарсия-Воча.

29 Поиск в деревьях. Красно-черные деревья (RB tree).

30 Структуры данных и алгоритмы для внешней памяти. Внешние деревья поиска. B+ дерево (B+ tree).

31 Алгоритм Джонсона нахождения всех пар кратчайших путей.

32 Алгоритм с возвратом. Задача Гамильтона.

33 Алгоритм метода ветвей и границ. Задача коммивояжера.

34 Алгоритм Литтла. Задача коммивояжера [11].

ЗАКЛЮЧЕНИЕ

Алгоритмы и структуры данных применяются для создания эффективных программных решений. Алгоритмы обеспечивают системный подход к решению задач, разбивая их на этапы. Структуры данных обеспечивают хранение и доступ к данным, с которыми работают алгоритмы.

Для закрепления теоретических знаний и приобретения практических навыков в реализации алгоритмов решения задач с использованием языков программирования в учебном пособии приводятся контрольные вопросы и варианты заданий для выполнения самостоятельных и курсовых работ.

Для реализации алгоритмов решения задач используется язык программирования Visual C++, технология визуального проектирования и событийного программирования, интегрированная среда программирования Microsoft Visual Studio 2022 Community.

Пособие рекомендуется для проведения занятий по дисциплине «Алгоритмы и структуры данных» и смежных дисциплин в учебном процессе студентов, обучающихся по направлению подготовки 09.00.00 «Информатика и вычислительная техника», и может быть использовано специалистами, занимающимися разработкой программных приложений, реализующих алгоритмы решения задач.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 Кормен Т. Алгоритмы. Анализ и построение / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – Москва : БИНОМ, 2000. – 960 с.

2 Ахо Альфред В. Структуры данных и алгоритмы : учебное пособие / Ахо Альфред В., Джон Хопкрофт, Ульман Джеффри Д. ; пер. с англ. – Москва : Издательский дом «Вильямс», 2000. – 432 с.

3 Алексеев Е. Р. Программирование на Microsoft Visual C++ и Turbo C++ Explorer. / Алексеев Е. Р. – Москва : НТ Пресс, 2007. – 359 с.

4 Хусаинов Б. С. Структуры и алгоритмы обработки данных. Примеры на языке Си : учебное пособие / Б. С. Хусаинов. – Москва : Финансы и статистика, 2004. – 464 с.

5 Анашкина Н. В. Технологии и методы программирования / Н. В. Анашкина, Н. Н. Петухова, В. Ю. Смольянинов. – Москва : Академия, 2012. – 384 с.

6 Алгоритмы и структуры данных : методические указания к выполнению лабораторных и курсовых работ для студентов направления (специальности) 231000.62 – «Программная инженерия» / Министерство образования и науки Российской Федерации, Курганский государственный университет, Кафедра программного обеспечения автоматизированных систем ; сост. А. М. Семахин. – Курган : Изд-во Курганского гос. ун-та, 2012. – 63 с.

7 Алгоритмы и структуры данных : методические указания к выполнению практических и контрольных работ для студентов направления подготовки 09.03.04 «Программная инженерия» / Министерство науки и высшего образования Российской Федерации, Курганский государственный университет, Кафедра программного обеспечения автоматизированных систем ; сост. А. М. Семахин. – Курган : Изд-во Курганского гос. ун-та, 2018. – 57 с.

8 Сэджвик Р. Фундаментальные алгоритмы на C++ / Р. Сэджвик. – Москва : DiaSoft, 2001. – 688 с. – Ч. 1–5.

9 Труб И. И. Объектно-ориентированное моделирование на C++: учебный курс / И. И. Труб. – Санкт-Петербург : Питер, 2006. – 411 с.

10 Таха Х. А. Введение в исследование операций / Х. А. Таха. – Москва : Вильямс, 2006. – 912 с.

11 Алгоритмы и структуры данных : методические указания к выполнению курсовых работ для студентов направления 231000.62 / Министерство образования и науки Российской Федерации, Курганский государственный университет, Кафедра программного обеспечения автоматизированных систем ; сост. А. М. Семахин. – Курган : Изд-во Курганского гос. ун-та, 2014. – 73 с.

12 Дик Д. И. Дипломное проектирование : учеб. пособие / Д. И. Дик. – Курган : Изд-во Курганского гос. ун-та, 2018. – 148 с.

ПРИЛОЖЕНИЕ А
Листинг программной реализации варианта задания раздела «5 Сортировка данных»

```
#pragma once
#include <iostream>
#include <cmath>

namespace TI1 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::Windows::Forms::DataVisualization::Charting;

    /// <summary>
    /// Сводка для MyForm
/// </summary>
    public ref class MyForm : public System::Windows::Forms::Form
    {
    public:
        MyForm(void)
        {
            InitializeComponent();
            //Настройка listView1 – увеличить ширину компоненты
            //на ширину вертикальной полосы прокрутки
            int w = 0;
            for (int i = 0; i < listView1->Columns->Count; i++)
                w += listView1->Columns[i]->Width;
            if (listView1->BorderStyle == BorderStyle::Fixed3D)
                w += 4;
            listView1->Width = w + 17;
            listView1->FullRowSelect = true;
```

```
    }
  protected:
    /// <summary>
    /// Освободить все используемые ресурсы.
    /// </summary>
    ~MyForm()
    {
      if (components)
      {
        delete components;
      }
    }
  private: System::Windows::Forms::Label^ label1;
  protected:
  private: System::Windows::Forms::Label^ label2;
  private: System::Windows::Forms::RadioButton^ radioButton1;
  private: System::Windows::Forms::RadioButton^ radioButton2;
  private: System::Windows::Forms::RadioButton^ radioButton3;
  private: System::Windows::Forms::RadioButton^ radioButton4;
  private: System::Windows::Forms::RadioButton^ radioButton5;
  private: System::Windows::Forms::RadioButton^ radioButton6;
  private: System::Windows::Forms::Button^ button1;
  private: System::Windows::Forms::TextBox^ textBox1;
  private: System::Windows::Forms::Label^ label3;
  private: System::Windows::Forms::Label^ label4;
  private: System::Windows::Forms::PictureBox^ pictureBox1;
  private: System::Windows::Forms::DataVisualization::Charting::Chart^
chart1;
  private: System::Windows::Forms::ListView^ listView1;
  private: System::Windows::Forms::Label^ label5;
  private: System::Windows::Forms::Label^ label6;
  private: System::Windows::Forms::Label^ label7;
  private: System::Windows::Forms::PictureBox^ pictureBox2;
  private: System::Windows::Forms::PictureBox^ pictureBox3;
  private: System::Windows::Forms::Label^ label8;
  private: System::Windows::Forms::Label^ label9;
```

```
    private: System::Windows::Forms::Label^ label10;
    private: System::Windows::Forms::Label^ label11;
    private: System::Windows::Forms::Label^ label12;
    private: System::Windows::Forms::TextBox^ textBox2;
    private: System::Windows::Forms::TextBox^ textBox3;
    private: System::Windows::Forms::Button^ button2;
    private: System::Windows::Forms::Button^ button3;
    private: System::Windows::Forms::PictureBox^ pictureBox4;
    private: System::Windows::Forms::TextBox^ textBox4;
    private: System::Windows::Forms::ColumnHeader^ columnHeader1;
    private: System::Windows::Forms::ColumnHeader^ columnHeader2;
    private: System::Windows::Forms::ColumnHeader^ columnHeader3;
    private: System::Windows::Forms::ColumnHeader^ columnHeader4;
    private: System::Windows::Forms::ColumnHeader^ columnHeader5;
    private: System::Windows::Forms::Button^ button4;
    private: System::Windows::Forms::Button^ button5;
    private: System::Windows::Forms::Label^ label13;
    private:
        /// <summary>
        /// Обязательная переменная конструктора.
        /// </summary>
        System::ComponentModel::Container ^components;
#pragma region Windows Form Designer generated code
        /// <summary>
        /// Требуемый метод для поддержки конструктора — не изменяйте
        /// содержимое этого метода с помощью редактора кода.
        /// </summary>
        void InitializeComponent(void)
        {
            System::ComponentModel::ComponentResourceManager^ resources =
(gcnew System::ComponentModel::ComponentRe-
sourceManager(MyForm::typeid));
            System::Windows::Forms::DataVisualization::Charting::ChartArea^
chartArea3 = (gcnew System::Windows::Forms::DataVisualization::Chart-
ing::ChartArea());
```

```
System::Windows::Forms::DataVisualization::Charting::Legend^ leg-
end3 = (gcnew System::Windows::Forms::DataVisualization::Charting::Leg-
end());
System::Windows::Forms::DataVisualization::Charting::Series^ series3
= (gcnew System::Windows::Forms::DataVisualization::Charting::Series());
this->label1 = (gcnew System::Windows::Forms::Label());
this->label2 = (gcnew System::Windows::Forms::Label());
this->radioButton1 = (gcnew System::Windows::Forms::Radio-
Button());
this->radioButton2 = (gcnew System::Windows::Forms::Radio-
Button());
this->radioButton3 = (gcnew System::Windows::Forms::Radio-
Button());
this->radioButton4 = (gcnew System::Windows::Forms::Radio-
Button());
this->radioButton5 = (gcnew System::Windows::Forms::Radio-
Button());
this->radioButton6 = (gcnew System::Windows::Forms::Radio-
Button());
this->button1 = (gcnew System::Windows::Forms::Button());
this->textBox1 = (gcnew System::Windows::Forms::TextBox());
this->label3 = (gcnew System::Windows::Forms::Label());
this->label4 = (gcnew System::Windows::Forms::Label());
this->pictureBox1 = (gcnew System::Windows::Forms::PictureBox());
this->chart1 = (gcnew System::Windows::Forms::DataVisualiza-
tion::Charting::Chart());
this->listView1 = (gcnew System::Windows::Forms::ListView());
this->columnHeader1 = (gcnew System::Windows::Forms::Column-
Header());
this->columnHeader2 = (gcnew System::Windows::Forms::Column-
Header());
this->columnHeader3 = (gcnew System::Windows::Forms::Column-
Header());
this->columnHeader4 = (gcnew System::Windows::Forms::Column-
Header());
```

```
        this->columnHeader5 = (gcnew System::Windows::Forms::Column-
Header());
        this->label5 = (gcnew System::Windows::Forms::Label());
        this->label6 = (gcnew System::Windows::Forms::Label());
        this->label7 = (gcnew System::Windows::Forms::Label());
        this->pictureBox2 = (gcnew System::Windows::Forms::PictureBox());
        this->pictureBox3 = (gcnew System::Windows::Forms::PictureBox());
        this->label8 = (gcnew System::Windows::Forms::Label());
        this->label9 = (gcnew System::Windows::Forms::Label());
        this->label10 = (gcnew System::Windows::Forms::Label());
        this->label11 = (gcnew System::Windows::Forms::Label());
        this->label12 = (gcnew System::Windows::Forms::Label());
        this->textBox2 = (gcnew System::Windows::Forms::TextBox());
        this->textBox3 = (gcnew System::Windows::Forms::TextBox());
        this->button2 = (gcnew System::Windows::Forms::Button());
        this->button3 = (gcnew System::Windows::Forms::Button());
        this->pictureBox4 = (gcnew System::Windows::Forms::PictureBox());
        this->textBox4 = (gcnew System::Windows::Forms::TextBox());
        this->button4 = (gcnew System::Windows::Forms::Button());
        this->button5 = (gcnew System::Windows::Forms::Button());
        this->label13 = (gcnew System::Windows::Forms::Label());
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox1))->BeginInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>chart1))->BeginInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox2))->BeginInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox3))->BeginInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox4))->BeginInit();
        this->SuspendLayout();
        //
        // label1
        //
        this->label1->Location = System::Drawing::Point(67, 6);
```

```
this->label1->Name = L"label1";
this->label1->Size = System::Drawing::Size(244, 27);
this->label1->TabIndex = 0;
this->label1->Text = L"1 Комбинаторные методы";
//
// label2
//
this->label2->Location = System::Drawing::Point(30, 38);
this->label2->Name = L"label2";
this->label2->Size = System::Drawing::Size(155, 31);
this->label2->TabIndex = 1;
this->label2->Text = L"Номер варианта";
//
// radioButton1
//
this->radioButton1->Location = System::Drawing::Point(42, 71);
this->radioButton1->Name = L"radioButton1";
this->radioButton1->Size = System::Drawing::Size(272, 25);
this->radioButton1->TabIndex = 2;
this->radioButton1->TabStop = true;
this->radioButton1->Text = L"Сочетания из h элементов по l";
this->radioButton1->UseVisualStyleBackColor = true;
//
// radioButton2
//
this->radioButton2->Location = System::Drawing::Point(45, 99);
this->radioButton2->Name = L"radioButton2";
this->radioButton2->Size = System::Drawing::Size(249, 25);
this->radioButton2->TabIndex = 3;
this->radioButton2->TabStop = true;
this->radioButton2->Text = L"Сочетания с повторениями";
this->radioButton2->UseVisualStyleBackColor = true;
//
// radioButton3
//
this->radioButton3->Location = System::Drawing::Point(44, 132);
```

```
this->radioButton3->Name = L"radioButton3";
this->radioButton3->Size = System::Drawing::Size(284, 25);
this->radioButton3->TabIndex = 4;
this->radioButton3->TabStop = true;
this->radioButton3->Text = L"Размещения из h элементов по l";
this->radioButton3->UseVisualStyleBackColor = true;
//
// radioButton4
//
this->radioButton4->Location = System::Drawing::Point(45, 167);
this->radioButton4->Name = L"radioButton4";
this->radioButton4->Size = System::Drawing::Size(250, 25);
this->radioButton4->TabIndex = 5;
this->radioButton4->TabStop = true;
this->radioButton4->Text = L"Размещения с повторением";
this->radioButton4->UseVisualStyleBackColor = true;
//
// radioButton5
//
this->radioButton5->Location = System::Drawing::Point(44, 201);
this->radioButton5->Name = L"radioButton5";
this->radioButton5->Size = System::Drawing::Size(241, 25);
this->radioButton5->TabIndex = 6;
this->radioButton5->TabStop = true;
this->radioButton5->Text = L"Перестановки h элементов";
this->radioButton5->UseVisualStyleBackColor = true;
//
// radioButton6
//
this->radioButton6->Location = System::Drawing::Point(45, 234);
this->radioButton6->Name = L"radioButton6";
this->radioButton6->Size = System::Drawing::Size(267, 25);
this->radioButton6->TabIndex = 7;
this->radioButton6->TabStop = true;
this->radioButton6->Text = L"Перестановки с поворениями";
this->radioButton6->UseVisualStyleBackColor = true;
```

```
            //
            // button1
            //
            this->button1->Font = (gcnew System::Drawing::Font(L"Times New
Roman", 24, System::Drawing::FontStyle::Regular, System::Draw-
ing::GraphicsUnit::Point,
                static_cast<System::Byte>(204)));
            this->button1->Location = System::Drawing::Point(501, 358);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(223, 45);
            this->button1->TabIndex = 8;
            this->button1->Text = L"Расчёт 1";
            this->button1->UseVisualStyleBackColor = true;
            this->button1->Click += gcnew System::EventHandler(this,
&MyForm::button1_Click);
            //
            // textBox1
            //
            this->textBox1->Location = System::Drawing::Point(191, 35);
            this->textBox1->Name = L"textBox1";
            this->textBox1->Size = System::Drawing::Size(100, 29);
            this->textBox1->TabIndex = 9;
            //
            // label3
            //
            this->label3->Location = System::Drawing::Point(38, 264);
            this->label3->Name = L"label3";
            this->label3->Size = System::Drawing::Size(97, 35);
            this->label3->TabIndex = 10;
            this->label3->Text = L"Результат";
            this->label3->TextAlign = System::Drawing::ContentAlignment::Mid-
dleCenter;
            //
            // label4
            //
            this->label4->AutoSize = true;
```

```
this->label4->Location = System::Drawing::Point(418, 10);
this->label4->Name = L"label4";
this->label4->Size = System::Drawing::Size(154, 21);
this->label4->TabIndex = 11;
this->label4->Text = L"2 Метод Шеннона";
//
// pictureBox1
//
this->pictureBox1->BackColor = System::Drawing::Color::Transparent;
this->pictureBox1->BackgroundImage = (cli::safe_cast<System::Drawing::Image^>(resources->GetObject(L"pictureBox1.BackgroundImage")));
this->pictureBox1->BackgroundImageLayout = System::Windows::Forms::ImageLayout::Center;
this->pictureBox1->Location = System::Drawing::Point(334, 35);
this->pictureBox1->Name = L"pictureBox1";
this->pictureBox1->Size = System::Drawing::Size(377, 45);
this->pictureBox1->TabIndex = 12;
this->pictureBox1->TabStop = false;
//
// chart1
//
chartArea3->Name = L"ChartArea1";
this->chart1->ChartAreas->Add(chartArea3);
legend3->Name = L"Legend1";
this->chart1->Legends->Add(legend3);
this->chart1->Location = System::Drawing::Point(334, 127);
this->chart1->Name = L"chart1";
series3->ChartArea = L"ChartArea1";
series3->Legend = L"Legend1";
series3->Name = L"Series1";
this->chart1->Series->Add(series3);
this->chart1->Size = System::Drawing::Size(377, 204);
this->chart1->TabIndex = 13;
this->chart1->Text = L"chart1";
```

```
        //
        // listView1
        //
        this->listView1->Columns->AddRange(gcnew cli::array< System::Windows::Forms::ColumnHeader^ >(5) {
            this->columnHeader1, this->columnHeader2,
                this->columnHeader3, this->columnHeader4, this->columnHeader5
        });
        this->listView1->GridLines = true;
        this->listView1->HideSelection = false;
        this->listView1->Location = System::Drawing::Point(730, 39);
        this->listView1->Name = L"listView1";
        this->listView1->Size = System::Drawing::Size(409, 292);
        this->listView1->TabIndex = 14;
        this->listView1->UseCompatibleStateImageBehavior = false;
        this->listView1->View = System::Windows::Forms::View::Details;
        //
        // columnHeader1
        //
        this->columnHeader1->Text = L"Итерация";
        this->columnHeader1->Width = 90;
        //
        // columnHeader2
        //
        this->columnHeader2->Text = L"P1";
        this->columnHeader2->TextAlign = System::Windows::Forms::HorizontalAlignment::Center;
        this->columnHeader2->Width = 89;
        //
        // columnHeader3
        //
        this->columnHeader3->Text = L"P2";
        this->columnHeader3->TextAlign = System::Windows::Forms::HorizontalAlignment::Center;
        this->columnHeader3->Width = 85;
```

```
            //
            // columnHeader4
            //
            this->columnHeader4->Text = L"P3";
            this->columnHeader4->TextAlign = System::Windows::Forms::Hori-
zontalAlignment::Center;
            this->columnHeader4->Width = 81;
            //
            // columnHeader5
            //
            this->columnHeader5->Text = L"H";
            this->columnHeader5->TextAlign = System::Windows::Forms::Hori-
zontalAlignment::Center;
            //
            // label5
            //
            this->label5->Location = System::Drawing::Point(115, 310);
            this->label5->Name = L"label5";
            this->label5->Size = System::Drawing::Size(176, 21);
            this->label5->TabIndex = 15;
            this->label5->Text = L"3 Исходные данные";
            this->label5->TextAlign = System::Drawing::ContentAlignment::Mid-
dleCenter;
            //
            // label6
            //
            this->label6->AutoSize = true;
            this->label6->Location = System::Drawing::Point(718, 409);
            this->label6->Name = L"label6";
            this->label6->Size = System::Drawing::Size(156, 21);
            this->label6->TabIndex = 16;
            this->label6->Text = L"3 Мера Харкевича";
            //
            // label7
            //
            this->label7->Location = System::Drawing::Point(840, 6);
```

```
            this->label7->Name = L"label7";
            this->label7->Size = System::Drawing::Size(186, 30);
            this->label7->TabIndex = 17;
            this->label7->Text = L"2 Расчётные данные";
            //
            // pictureBox2
            //
            this->pictureBox2->BackgroundImage = (cli::safe_cast<Sys-
tem::Drawing::Image^>(resources->GetObject(L"pictureBox2.Back-
groundImage")));
            this->pictureBox2->BackgroundImageLayout = System::Win-
dows::Forms::ImageLayout::Center;
            this->pictureBox2->Location = System::Drawing::Point(15, 340);
            this->pictureBox2->Name = L"pictureBox2";
            this->pictureBox2->Size = System::Drawing::Size(471, 332);
            this->pictureBox2->TabIndex = 18;
            this->pictureBox2->TabStop = false;
            //
            // pictureBox3
            //
            this->pictureBox3->BackgroundImage = (cli::safe_cast<Sys-
tem::Drawing::Image^>(resources->GetObject(L"pictureBox3.Back-
groundImage")));
            this->pictureBox3->Location = System::Drawing::Point(501, 441);
            this->pictureBox3->Name = L"pictureBox3";
            this->pictureBox3->Size = System::Drawing::Size(250, 54);
            this->pictureBox3->TabIndex = 19;
            this->pictureBox3->TabStop = false;
            //
            // label8
            //
            this->label8->Location = System::Drawing::Point(495, 498);
            this->label8->Name = L"label8";
            this->label8->Size = System::Drawing::Size(282, 35);
            this->label8->TabIndex = 20;
            this->label8->Text = L"Iцелес.>0 –информация полезная";
```

```
        this->label8->TextAlign = System::Drawing::ContentAlignment::Mid-
dleLeft;
        //
        // label9
        //
        this->label9->Location = System::Drawing::Point(497, 533);
        this->label9->Name = L"label9";
        this->label9->Size = System::Drawing::Size(257, 45);
        this->label9->TabIndex = 21;
        this->label9->Text = L"Iцелес.=0–информация пустая";
        this->label9->TextAlign = System::Drawing::ContentAlignment::Mid-
dleLeft;
        //
        // label10
        //
        this->label10->Location = System::Drawing::Point(499, 576);
        this->label10->Name = L"label10";
        this->label10->Size = System::Drawing::Size(216, 33);
        this->label10->TabIndex = 22;
        this->label10->Text = L"Iцелес.<0–дезинформаця";
        this->label10->TextAlign = System::Drawing::ContentAlignment::Mid-
dleLeft;
        //
        // label11
        //
        this->label11->Location = System::Drawing::Point(784, 528);
        this->label11->Name = L"label11";
        this->label11->Size = System::Drawing::Size(79, 21);
        this->label11->TabIndex = 23;
        this->label11->Text = L"Iцелес. 1";
        //
        // label12
        //
        this->label12->Location = System::Drawing::Point(781, 583);
        this->label12->Name = L"label12";
        this->label12->Size = System::Drawing::Size(79, 21);
```

```
        this->label12->TabIndex = 24;
        this->label12->Text = L"Iцелес. 2";
        //
        // textBox2
        //
        this->textBox2->Location = System::Drawing::Point(142, 270);
        this->textBox2->Name = L"textBox2";
        this->textBox2->Size = System::Drawing::Size(184, 29);
        this->textBox2->TabIndex = 25;
        //
        // textBox3
        //
        this->textBox3->Location = System::Drawing::Point(866, 522);
        this->textBox3->Name = L"textBox3";
        this->textBox3->Size = System::Drawing::Size(273, 29);
        this->textBox3->TabIndex = 26;
        //
        // button2
        //
        this->button2->Font = (gcnew System::Drawing::Font(L"Times New
Roman", 24, System::Drawing::FontStyle::Regular, System::Draw-
ing::GraphicsUnit::Point,
            static_cast<System::Byte>(204)));
        this->button2->Location = System::Drawing::Point(730, 356);
        this->button2->Name = L"button2";
        this->button2->Size = System::Drawing::Size(408, 45);
        this->button2->TabIndex = 27;
        this->button2->Text = L"Расчёт 2";
        this->button2->UseVisualStyleBackColor = true;
        this->button2->Click += gcnew System::EventHandler(this,
&MyForm::button2_Click);
        //
        // button3
        //
```

```
        this->button3->Font = (gcnew System::Drawing::Font(L"Times New
Roman", 24, System::Drawing::FontStyle::Regular, System::Draw-
ing::GraphicsUnit::Point,
            static_cast<System::Byte>(204)));
        this->button3->Location = System::Drawing::Point(525, 627);
        this->button3->Name = L"button3";
        this->button3->Size = System::Drawing::Size(190, 45);
        this->button3->TabIndex = 28;
        this->button3->Text = L"Расчёт 3";
        this->button3->UseVisualStyleBackColor = true;
        this->button3->Click += gcnew System::EventHandler(this,
&MyForm::button3_Click);
        //
        // pictureBox4
        //
        this->pictureBox4->BackgroundImage = (cli::safe_cast<Sys-
tem::Drawing::Image^>(resources->GetObject(L"pictureBox4.Back-
groundImage")));
        this->pictureBox4->Location = System::Drawing::Point(783, 435);
        this->pictureBox4->Name = L"pictureBox4";
        this->pictureBox4->Size = System::Drawing::Size(351, 75);
        this->pictureBox4->TabIndex = 31;
        this->pictureBox4->TabStop = false;
        //
        // textBox4
        //
        this->textBox4->Location = System::Drawing::Point(866, 579);
        this->textBox4->Name = L"textBox4";
        this->textBox4->Size = System::Drawing::Size(273, 29);
        this->textBox4->TabIndex = 33;
        //
        // button4
        //
        this->button4->Font = (gcnew System::Drawing::Font(L"Times New
Roman", 24, System::Drawing::FontStyle::Regular, System::Draw-
ing::GraphicsUnit::Point,
```

```
            static_cast<System::Byte>(204)));
        this->button4->Location = System::Drawing::Point(743, 627);
        this->button4->Name = L"button4";
        this->button4->Size = System::Drawing::Size(190, 45);
        this->button4->TabIndex = 34;
        this->button4->Text = L"Очистка";
        this->button4->UseVisualStyleBackColor = true;
        this->button4->Click += gcnew System::EventHandler(this,
&MyForm::button4_Click);
        //
        // button5
        //
        this->button5->Font = (gcnew System::Drawing::Font(L"Times New
Roman", 24, System::Drawing::FontStyle::Regular, System::Draw-
ing::GraphicsUnit::Point,
            static_cast<System::Byte>(204)));
        this->button5->Location = System::Drawing::Point(947, 625);
        this->button5->Name = L"button5";
        this->button5->Size = System::Drawing::Size(190, 45);
        this->button5->TabIndex = 35;
        this->button5->Text = L"Выход";
        this->button5->UseVisualStyleBackColor = true;
        this->button5->Click += gcnew System::EventHandler(this,
&MyForm::button5_Click);
        //
        // label13
        //
        this->label13->Location = System::Drawing::Point(396, 91);
        this->label13->Name = L"label13";
        this->label13->Size = System::Drawing::Size(248, 30);
        this->label13->TabIndex = 36;
        this->label13->Text = L"График зависимости H=f(P2) ";
        //
        // MyForm
        //
        this->AutoScaleDimensions = System::Drawing::SizeF(10, 21);
```

```
        this->AutoScaleMode = System::Windows::Forms::AutoScale-
Mode::Font;
        this->AutoSize = true;
        this->BackColor = System::Drawing::Color::Fro-
mArgb(static_cast<System::Int32>(static_cast<System::Byte>(128)),
static_cast<System::Int32>(static_cast<System::Byte>(128)),
            static_cast<System::Int32>(static_cast<System::Byte>(255)));
        this->BackgroundImageLayout = System::Windows::Forms::ImageLay-
out::Stretch;
        this->ClientSize = System::Drawing::Size(1164, 680);
        this->Controls->Add(this->label13);
        this->Controls->Add(this->button5);
        this->Controls->Add(this->button4);
        this->Controls->Add(this->textBox4);
        this->Controls->Add(this->pictureBox4);
        this->Controls->Add(this->button3);
        this->Controls->Add(this->button2);
        this->Controls->Add(this->textBox3);
        this->Controls->Add(this->textBox2);
        this->Controls->Add(this->label12);
        this->Controls->Add(this->label11);
        this->Controls->Add(this->label10);
        this->Controls->Add(this->label9);
        this->Controls->Add(this->label8);
        this->Controls->Add(this->pictureBox3);
        this->Controls->Add(this->pictureBox2);
        this->Controls->Add(this->label7);
        this->Controls->Add(this->label6);
        this->Controls->Add(this->label5);
        this->Controls->Add(this->listView1);
        this->Controls->Add(this->chart1);
        this->Controls->Add(this->pictureBox1);
        this->Controls->Add(this->label4);
        this->Controls->Add(this->label3);
        this->Controls->Add(this->textBox1);
        this->Controls->Add(this->button1);
```

```
        this->Controls->Add(this->radioButton6);
        this->Controls->Add(this->radioButton5);
        this->Controls->Add(this->radioButton4);
        this->Controls->Add(this->radioButton3);
        this->Controls->Add(this->radioButton2);
        this->Controls->Add(this->radioButton1);
        this->Controls->Add(this->label2);
        this->Controls->Add(this->label1);
        this->Font = (gcnew System::Drawing::Font(L"Times New Roman",
14.25F, System::Drawing::FontStyle::Regular, System::Drawing::GraphicsU-
nit::Point,
            static_cast<System::Byte>(204)));
        this->FormBorderStyle = System::Windows::Forms::FormBorder-
Style::FixedSingle;
        this->Margin = System::Windows::Forms::Padding(5);
        this->Name = L"MyForm";
        this->StartPosition = System::Windows::Forms::FormStartPosi-
tion::CenterScreen;
        this->Text = L"Измерение информации";
        this->Load += gcnew System::EventHandler(this,
&MyForm::MyForm_Load);
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox1))->EndInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>chart1))->EndInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox2))->EndInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox3))->EndInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>pictureBox4))->EndInit();
        this->ResumeLayout(false);
        this->PerformLayout();
    }
#pragma endregion
    int N;
```

```
   //Вычисление факториала
      int factorial(int k) {
         if (k <= 1) return k;
         else return k * factorial(k-1);
      }
   private: System::Void button1_Click(System::Object^ sender, System::Even-
tArgs^ e) {
      listView1->Items->Clear();
//Расчёт количества информации с использованием комбинаторных мер
      int n, h, l, alfa, beta, gamma;
      double q;
        try {
         n = System::Convert::ToInt32(textBox1->Text);
         }
      catch (...) {
         MessageBox::Show("Введите номер варианта!", "Исходные дан-
ные",
            MessageBoxButtons::OK,
            MessageBoxIcon::Error);
            textBox1->Clear(); textBox1->TabIndex = 0;
      }
      h = n + 2; if (n % 2 == 0) l = 3; else l = 2;
      alfa = 2; beta = 2; gamma = n; N = n;
      if (radioButton1->Checked) { //Сочетания из h элементов по l
         q = (double)factorial(h)/ (double)(factorial(l)*factorial(h-l));
         textBox2->Text = q.ToString("n");
      }
      if (radioButton2->Checked) { //Сочетания с повторениями
         q = (double)factorial(h+l-1) / (double)(factorial(l) * factorial(h - l));
         textBox2->Text = q.ToString("n");
      }
      if (radioButton3->Checked) { /* Размещения из h элементов по l эле-
ментов */
         q = (double)factorial(h) / (double)factorial(h - l);
         textBox2->Text = q.ToString("n");
      }
```

```
        if (radioButton4->Checked) { /* Размещения с повторениями по l из h элементов */
            q = pow((double)h,l);
            textBox2->Text = q.ToString("n");
        }
        if (radioButton5->Checked) { //Перестановки h элементов
            q = (double)factorial(h);
            textBox2->Text = q.ToString("n");
        }
        if (radioButton6->Checked) { /* Перестановки с повторениями элементов */
            q = (double)(alfa + beta + gamma)/(double)(factorial(alfa)* factorial(beta)* factorial(gamma));
            textBox2->Text = q.ToString("n");
        }
        if (textBox1->Text->Length == 0) button2->Enabled = false;
        else button2->Enabled = true;
        if (textBox1->Text->Length == 0) button3->Enabled = false;
        else button3->Enabled = true;
}
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e) {
//Расчёт количества информации с использованием метода К.Э. Шеннона
    listView1->Items->Clear();
    chart1->Series->Clear();
    Series^ series1 = gcnew Series(L"Линейный график");
    series1->Color = Color::Red;
    series1->IsVisibleInLegend = false;
    series1->IsXValueIndexed = false;
    series1->ChartType = SeriesChartType::Point;
    chart1->Series->Add(series1);
    double p1, p2, p3, H;
        p3 = 1.0 / (1.0 + N);
    for (int i = 0; p2+p3 <= 1; i++) {
        p1 = 1.0 - p2 - p3; p2 += 0.05;
        H = -(p1 * log2(p1) + p2 * log2(p2) + p3 * log2(p3));
```

```
        //Добавить в listView1 элемент – строку (данные в первый столбец)
        listView1->Items->Add(i.ToString());
        /* Добавить в добавленную строку подэлементы – данные во второй
столбец */
        listView1->Items[i]->SubItems->Add(p1.ToString("n"));
        /* Добавить в добавленную строку подэлементы – данные в третий
столбец */
        listView1->Items[i]->SubItems->Add(p2.ToString("n"));
        /* Добавить в добавленную строку подэлементы – данные в чет-
вртый столбец */
        listView1->Items[i]->SubItems->Add(p3.ToString("n"));
        /* Добавить в добавленную строку подэлементы – данные в пятый
столбец */
        listView1->Items[i]->SubItems->Add(H.ToString("n"));
        series1->Points->AddXY(p2, H);
    }
}
private: System::Void button3_Click(System::Object^ sender, System::Even-
tArgs^ e) {
    double I1, I2;
    I1 = log2((3.0/N)*(2.0/N)) - log2(1.0/3.0);
    I2 = log2(2.0 / N) - log2(1.0/3.0);
    textBox3->Text = I1.ToString("r");
    textBox4->Text = I2.ToString("r");
}
private: System::Void button4_Click(System::Object^ sender, System::Even-
tArgs^ e) {
    chart1->Series->Clear(); listView1->Items->Clear(); textBox1->Clear();
    textBox2->Clear(); textBox3->Clear(); textBox4->Clear();
    textBox1->TabIndex = 0;
}
private: System::Void button5_Click(System::Object^ sender, System::Even-
tArgs^ e) {
    this->Close();
    }
```

```
private: System::Void MyForm_Load(System::Object^ sender, System::Even-
tArgs^ e) {
   chart1->Series->Clear(); listView1->Items->Clear(); textBox1->Clear();
   textBox2->Clear(); textBox3->Clear(); textBox4->Clear();
   textBox1->TabIndex = 0;
   button2->Enabled = false;
   button3->Enabled = false;
      } };}
```

Учебное издание

Семахин Андрей Михайлович

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ.
ПРИЛОЖЕНИЯ НА C++

Учебное пособие

Редактор Н. М. Быкова

Подписано в печать 15.04.25	Формат 60x84 1/16	Бумага 80 г/см^2
Печать цифровая	Усл. печ. л. 10,25	Уч.-изд. л. 10,25
Заказ 19	Тираж 100	

БИЦ Курганского государственного университета.
640002, г. Курган, ул. Советская, 63/4.
Курганский государственный университет.

ISBN 978-5-4217-0720-2

Курганский
государственный
университет

Библиотечно-издательский
центр

65-48-12