МИНОБРНАУКИ РОССИИ

*Федеральное государственное бюджетное образовательное учреждение*

*высшего образования*

***«МИРЭА – Российский технологический университет»***

**РТУ МИРЭА**

---

Отчет по выполнению практического задания №7

**Тема:**

НЕЛИНЕЙНЫЕ СТРУКТУРЫ

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Враженко Д.О.

Группа: ИКБО-50-23

Вариант: 6

Москва – 2024

# ЦЕЛЬ РАБОТЫ

**Часть 7.1.** Освоить балансировку дерева поиска.

**Часть 7.2.** Освоить графы: создание, алгоритмы обхода, важные задачи теории графов.

# ХОД РАБОТЫ

**Часть 7.1. Балансировка дерева поиска**

**Формулировка задачи:**

Составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту.

Процедуры оформить в виде самостоятельных режимов работы созданного дерева. Выбор режимов производить с помощью пользовательского (иерархического ниспадающего) меню.

Провести полное тестирование программы на дереве размером n=10 элементов, сформированном вводом с клавиатуры. Тест-примеры определить самостоятельно. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием созданного дерева, принципов программной реализации алгоритмов работы с деревом, описанием текста исходного кода и проведенного тестирования программы.

**Индивидуальный вариант:** 6.

**Тип значения узла:** Вещественное.

**Тип дерева:** АВЛ-дерево.

**Реализовать алгоритмы:** Вставка элемента (и балансировка), Обратный обход, Симметричный обход, Найти сумму значений листьев, Найти среднее арифметическое всех узлов.

**Математическая модель решения (описание алгоритма):**

AVL-дерево – это сбалансированное двоичное дерево поиска. Так как по определению оно обязано быть сбалансированным, после каждого добавления нового элемента AVL-дерево необходимо проверять на разбалансированность и, в случае её обнаружения, балансировать.

Для определения разбалансированности вводится параметр фактора баланса, который для каждого узла равен разницы высот правого и левого поддеревьев. Дерево является сбалансированным, если для всех его элементов справедливо правило: баланс фактор по модулю не больше единицы. Если же баланс фактор становится равен 2 или -2 – дерево нуждается в балансировке.

Для балансировки используются повороты дерева – они делятся на левые и правые, большие и маленькие. Причём большой поворот является комбинацией двух маленьких.

Также после балансировки изменяются значения высот поддеревьев, поэтому важно исправить их на корректные.

После вставки, балансировки и исправления высот поддеревьев к дереву возможно применять разнообразные операции:

Прямой обход – вывод значения узла, а потом рекурсивный вывод левого и правого поддеревьев.

Обратный обход – рекурсивный вывод левого и правого поддеревьев, а потом вывод значения узла.

Симметричный обход – рекурсивный вывод левого поддерева, вывод собственного значения узла, рекурсивный вывод правого поддерева. Именно при симметричном обходе на выводе получается отсортированная последовательность элементов дерева поиска.

Обход в ширину – последовательный вывод всех элементов, находящихся на одном уровне, после чего вывод следующего уровня. Отличается отсутствием рекурсивных вызовов методов.

Нахождение суммы значений листьев – рекурсивный обход всех элементов со сложением результатов для левого и правого поддеревьев, причем собственное значение возвращают только листья, остальные оставляют его без изменений.

Нахождение среднего арифметического значений всех узлов – рекурсивный обход всех элементов со сложением результатов для левого и правого под-

деревьев, причем все узлы возвращают собственное значение. Для нахождения количества также производится рекурсивный обход, в ходе которого каждый узел увеличивает счетчик на единицу.

**Код программы с комментариями:**

На рис. 1 представлены коды методов для нахождения количества всех узлов в дереве и суммы их значений. Эти методы нам пригодятся для нахождения среднего арифметического всех узлов.

```
int count() // метод для подсчета количества всех узлов
{
    int count = 1;
    if (this->left)
        count += this->left->count();
    if (this->right)
        count += this->right->count();
    return count;
}

double sum() // метод для подсчета суммы значений всех узлов
{
    int sum = this->value;
    if (this->left)
        sum += this->left->sum();
    if (this->right)
        sum += this->right->sum();
    return sum;
}
```

Рисунок 1 - Методы для нахождения количества всех узлов и суммы их значений

На рис. 2 представлен код конструктора дерева.

```
avl(double value) // конструктор дерева
{
    this->value = value;
    this->left = nullptr;
    this->right = nullptr;
    this->height = 1;
}
```

Рисунок 2 - Код конструктора дерева

На рис. 3 представлен код для обратного обхода по дереву.

```
void postOrder() // обратный обход
{
    if (this->left)
        this->left->postOrder();
    if (this->right)
        this->right->postOrder();
    cout << this->value << ' ';
}
```

Рисунок 3 - Код обратного обхода

На рис. 4 представлен код для симметричного обхода.

```
void inOrder() // симметричный обход
{
    if (this->left)
        this->left->inOrder();
    cout << this->value << ' ';
    if (this->right)
        this->right->inOrder();
}
```

Рисунок 4 - Код симметричного обхода

На рис. 5 представлен для нахождения суммы значений листьев.

```
double leavesSum() // метод для подсчета суммы значений всех листьев
{
    if (!this->left && !this->right)
        return this->value;
    double sum = 0;
    if (this->left)
        sum += this->left->leavesSum();
    if (this->right)
        sum += this->right->leavesSum();
    return sum;
}
```

Рисунок 5 - Код для нахождения суммы листьев

На рис. 6 представлен код для нахождения среднего арифметического всех узлов на дереве.

```
double avg() // метод для нахождения среднего арифметического всех узлов
{ return this->sum() / this->count(); }
```

Рисунок 6 - Код для нахождения среднего арифметического

На рис. 7 представлен код функции для балансировки дерева.

```
avl* balance(avl* tree) // метод для балансировки дерева
{
    fixHeight(tree);
    if (bFactor(tree) == 2)
    {
        if (bFactor(tree->right) < 0)
            tree->right = rotateRight(tree->right);
        return rotateLeft(tree);
    }
    if (bFactor(tree) == -2)
    {
        if (bFactor(tree->left) > 0)
            tree->left = rotateLeft(tree->left);
        return rotateRight(tree);
    }
    return tree; // балансировка не нужна
}
```

Рисунок 7 - Код для балансировки дерева

На рис. 8 представлен код функций для поворота дерева налево и направо.

```
avl* rotateRight(avl* old_tree) // метод для поворота дерева направо
{
    avl* new_tree = old_tree->left;
    old_tree->left = new_tree->right;
    new_tree->right = old_tree;
    fixHeight(old_tree);
    fixHeight(new_tree);
    return new_tree;
}

avl* rotateLeft(avl* old_tree) // метод для поворота дерева налево
{
    avl* new_tree = old_tree->right;
    old_tree->right = new_tree->left;
    new_tree->left = old_tree;
    fixHeight(old_tree);
    fixHeight(new_tree);
    return new_tree;
}
```

Рисунок 8 - Коды для поворота дерева

На рис. 9 представлены коды для работы функция поворотов.

```
int height(avl* tree) // функция для нахождения высоты дерева
{ return tree ? tree->height : 0; }

int bFactor(avl* tree) // Б-фактор
{ return height(tree->right) - height(tree->left); }

void fixHeight(avl* tree) // метод для исправления глубины дерева
{
    int height_left = height(tree->left);
    int height_right = height(tree->right);
    tree->height = (height_left > height_right ? height_left : height_right) + 1;
}
```

Рисунок 9 - Коды функций для работы функций поворотов

На рис. 10 представлен код функции для ввода новых значений в дерево.

```
avl* insert(avl* tree, double new_data) // метод для ввода новых данных в дерево
{
    if (!tree)
        return new avl(new_data);
    if (new_data < tree->value)
        tree->left = insert(tree->left, new_data);
    else
        tree->right = insert(tree->right, new_data);
    return balance(tree);
}
```

Рисунок 10 - Код для ввода новых значений

**Результаты тестирования:**

На рис. 11 представлен тест для вывода дерева на экран.

```
Что вы хотите сделать?
1 - Вывод содержимого
2 - Добавление записи
3 - Обратный обход
4 - Симметричный обход
5 - Сумма значений листьев
6 - Среднее арифметическое всех узлов
0 - Выход
Номер действия: 1

AVL дерево:
7.12
    2.243
        1.1111
            0.576834
            2.2222
        6.87124
    123.433
        7.547
            123.123
        346.2
            234.1
            6754.46
```

Рисунок 11 - Тест вывода дерева на экран

На рис. 12 представлен тест добавления нового элемента в дерево.

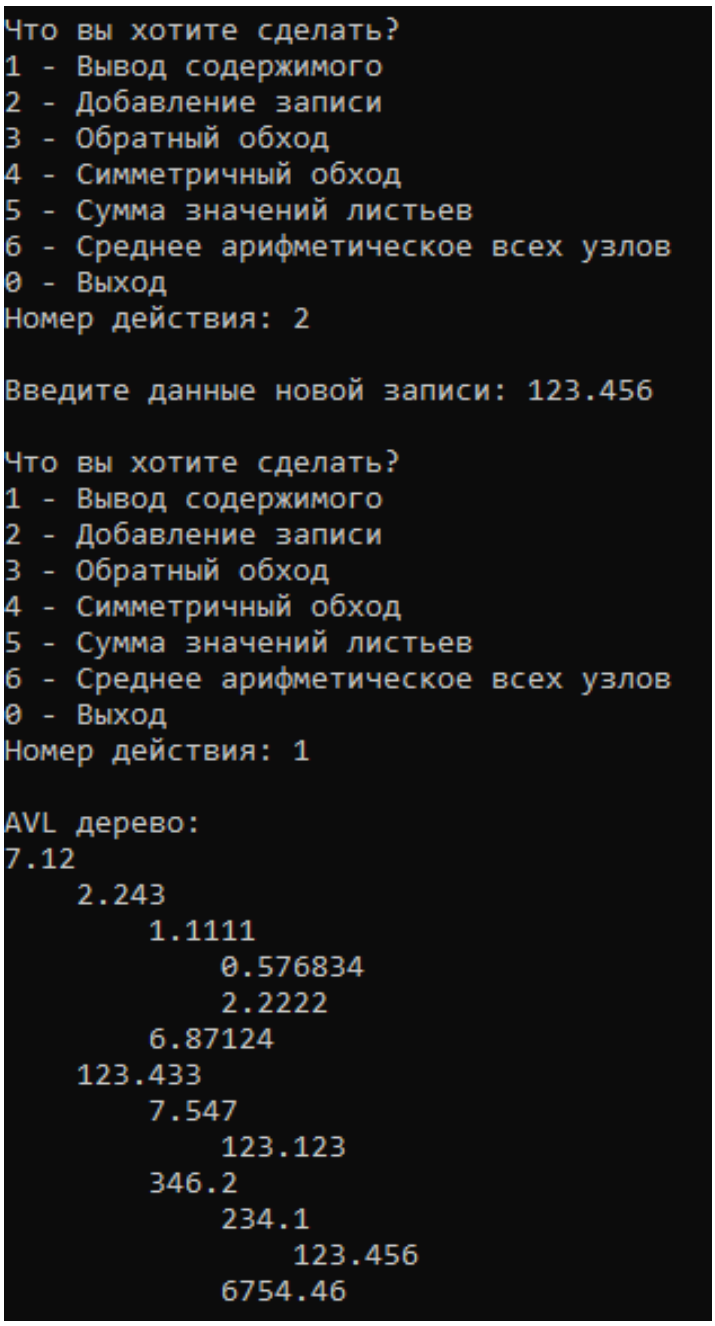


Рисунок 12 - Тест добавления нового элемента

На рис. 13 представлен тест обратного обхода.

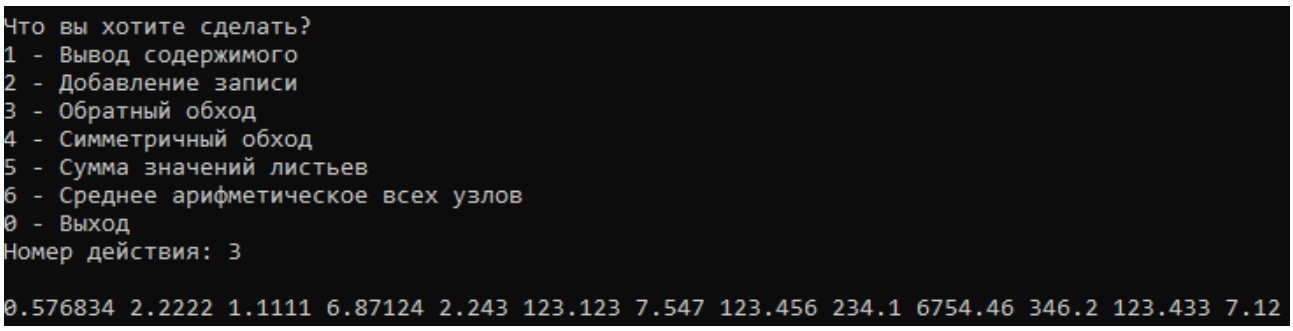


Рисунок 13 - Тест обратного обхода

На рис. 14 представлен тест симметричного обхода.

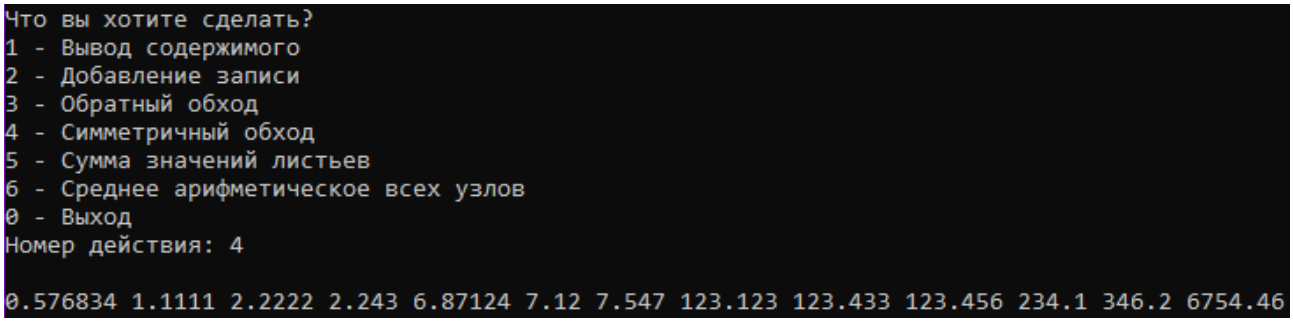


Рисунок 14 - Тест симметричного обхода

На рис. 15 представлен тест нахождения суммы значений листьев.

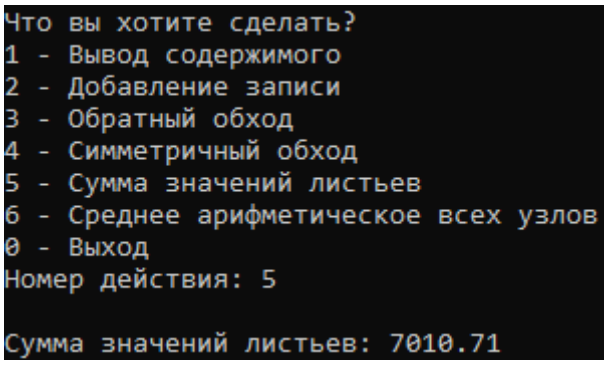


Рисунок 15 - Тест суммы листьев

На рис. 16 представлен тест нахождения среднего арифметического всех узлов на дереве.

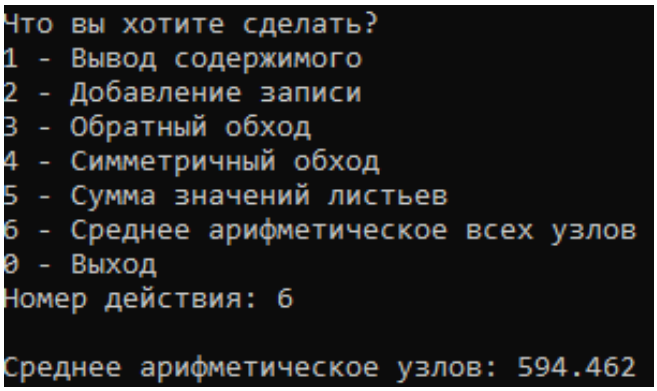


Рисунок 16 - Тест среднего арифметического

**Часть 7.2. Графы: создание, алгоритмы обхода, важные задачи теории графов**

**Формулировка задачи:**

Составить программу создания графа и реализовать процедуру для работы с графом, определенную индивидуальным вариантом задания.

Самостоятельно выбрать и реализовать способ представления графа в памяти.

В программе предусмотреть ввод с клавиатуры произвольного графа. В вариантах построения основного дерева также разработать доступный способ (форму) вывода результирующего дерева на экран монитора.

Провести тестовый прогон программы на предложенном в индивидуальном варианте задания графе. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием рассматриваемого графа, принципов программной реализации алгоритмов работы с графом, описанием текста исходного кода и проведенного тестирования программы.

**Индивидуальный вариант:** 6.

**Алгоритм:** Нахождение кратчайшего пути методом Флойда.

**Предложенный граф:** На рис. 17 представлен предложенный граф.

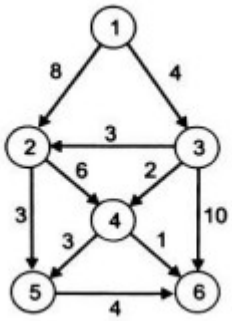


Рисунок 17 - Предложенный граф

**Математическая модель решения (описание алгоритма):**

В индивидуальном варианте №6 предоставленный граф является помеченным, взвешенным и ориентированным, но не является связным. Граф помеченный – это значит, что у каждой вершины графа есть метка (в данном случае

числовая). Граф взвешенный – это значит что ребра (дуги) графа имеют веса, обозначающие чаще всего расстояние между вершинами. Граф ориентированный – значит направление обхода и перемещения между двумя вершинами существенно. Граф не связный, так как из-за направленности ребер нельзя любые его две вершины соединить маршрутом в обоих направлениях, а некоторые нельзя соединить и в одном направлении.

В программе граф размера n представляется матрицей смежности graph[n][n], где элементу graph[i][j] соответствует вес дуги, выходящей из вершины i и заканчивающейся в вершине j. Если в классической матрице смежности элементы расположены зеркально относительно главной диагонали матрицы, так как матрица составляется для неориентированного графа, в использованном мной представлении симметрия нарушена ввиду ориентированности дуг относительно вершин.

Алгоритм Флойда-Уоршелла представляет собой на деле довольно простой алгоритм полного перебора всех возможных путей. Сначала происходит адаптация матрицы смежности, в ходе которой все нули, означающие отсутствующую дугу между вершинами, заменяют на очень большое число, теперь обозначающее бесконечное расстояние между вершинами.

Далее начинается перебор трёх индексов – i, j и k. Так сравнивается уже известное расстояние между i и j с суммарным расстоянием между i и k, а также k и j, что позволяет определить кротчайший путь, ведь мы рассматриваем все возможные промежуточные вершины между вершинами i и j.

После завершения работы алгоритма получается матрица кратчайших расстояний graph[n][n], в которой элемент graph[i][j] обозначает кратчайший из вершины i в вершину j.

**Код программы с комментариями:**

На рис. 18 представлен код метода для ввода данных в граф.

```
vector<vector<int>> enterGraph(int n) // метод для ввода данных в граф
{
    vector<vector<int>> graph(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) // перебор строк графа
    {
        cout << "Введите веса дуг, исходящих из вершины " << i << ": ";
        for (int j = 0; j < n; ++j) // перебор столбцов графа
        {
            int num;
            cin >> num;
            graph[i][j] = num;
        }
    }
    return graph;
}
```

Рисунок 18 - Метод для ввода данных

На рис. 19 представлен код метода для чтения данных из файла

```
vector<vector<int>> readGraph() // метод для чтения данных из графа
{
    ifstream file("text.txt");
    int num, n;
    file >> n;
    vector<vector<int>> graph(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) // проход по строкам графа
        for (int j = 0; j < n; ++j) // проход по столбцам графа
        {
            file >> num;
            graph[i][j] = num;
        }
    file.close();
    return graph;
}
```

Рисунок 19 - Метод для чтения данных из файла

На рис. 20 представлен код метода для создания нового графа.

```
void buildGraph(vector<vector<int>>& graph) // создание нового графа
{
    int n = graph.size();
    int inf = 999;
    for (int i = 0; i < n; ++i) // проход по строкам графа
        for (int j = 0; j < n; ++j) // проход по столбцам графа
            if (graph[i][j] == 0 && i != j)
                graph[i][j] = inf;
    for (int k = 0; k < n; ++k) // проход по индексу k
        for (int i = 0; i < n; ++i) // проход по индексу i
            for (int j = 0; j < n; ++j) // проход по индексу j
                if (graph[i][k] + graph[k][j] < graph[i][j])
                    graph[i][j] = graph[i][k] + graph[k][j];
}
```

Рисунок 20 - Метод для создания нового графа

На рис. 21 представлен код метода для вывода графа на экран.

```
void showGraph(vector<vector<int>>& graph) // метод для вывода графа на экран
{
    int n = graph.size();
    cout << "  \\ ";
    int offset = 0;
    for (int i = 0; i < n; ++i)
    {
        offset = (int)log10(i);
        offset = offset < 0 ? 0 : offset;
        cout << string(3 - offset, ' ') << i;
    }
    cout << "\n   \\" << string(n * 4 + 1, '-') << endl;
    for (int i = 0; i < n; ++i) // проход по строкам графа
    {
        offset = (int)log10(i);
        offset = offset < 0 ? 0 : offset;
        cout << string(2 - offset, ' ') << i << "|";
        for (int j = 0; j < n; ++j) // проход по столбцам графа
        {
            int num = graph[i][j];
            if (num != 999)
            {
                offset = (int)log10(num);
                offset = offset < 0 ? 0 : offset;
                cout << string(3 - offset, ' ') << num;
            }
            else
                cout << "   -";
        }
        cout << endl;
    }
}
```

Рисунок 21 - Метод для вывода графа

**Результаты тестирования:**

На рис. 22 представлен тест программы на предложенном графе, данные о котором хранятся в файле.

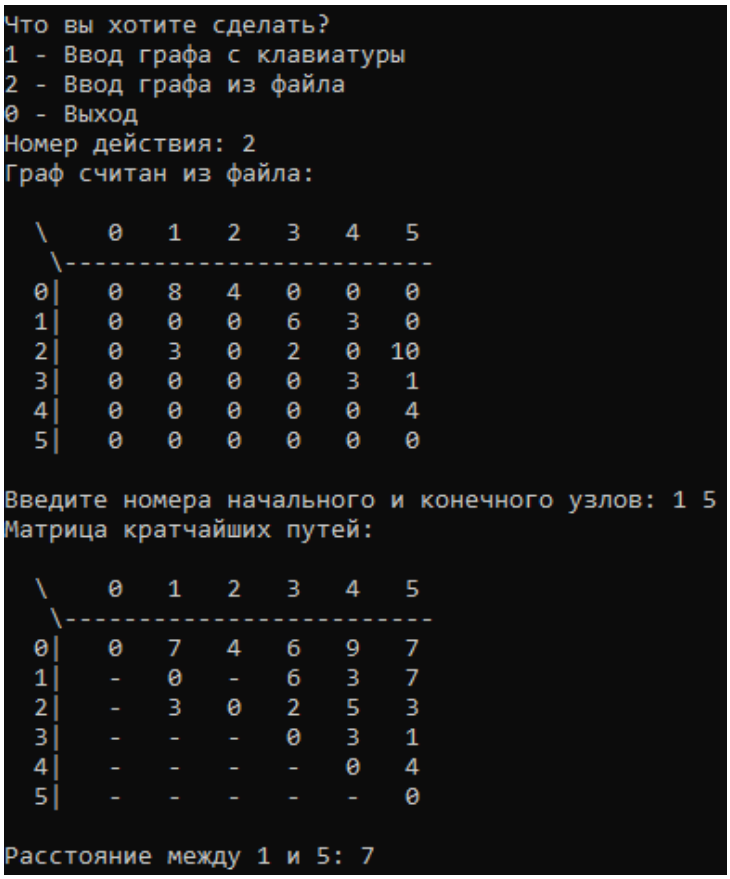


Рисунок 22 - Тест на предложенном графе

На рис. 23 представлен тест программы на придуманном графе.

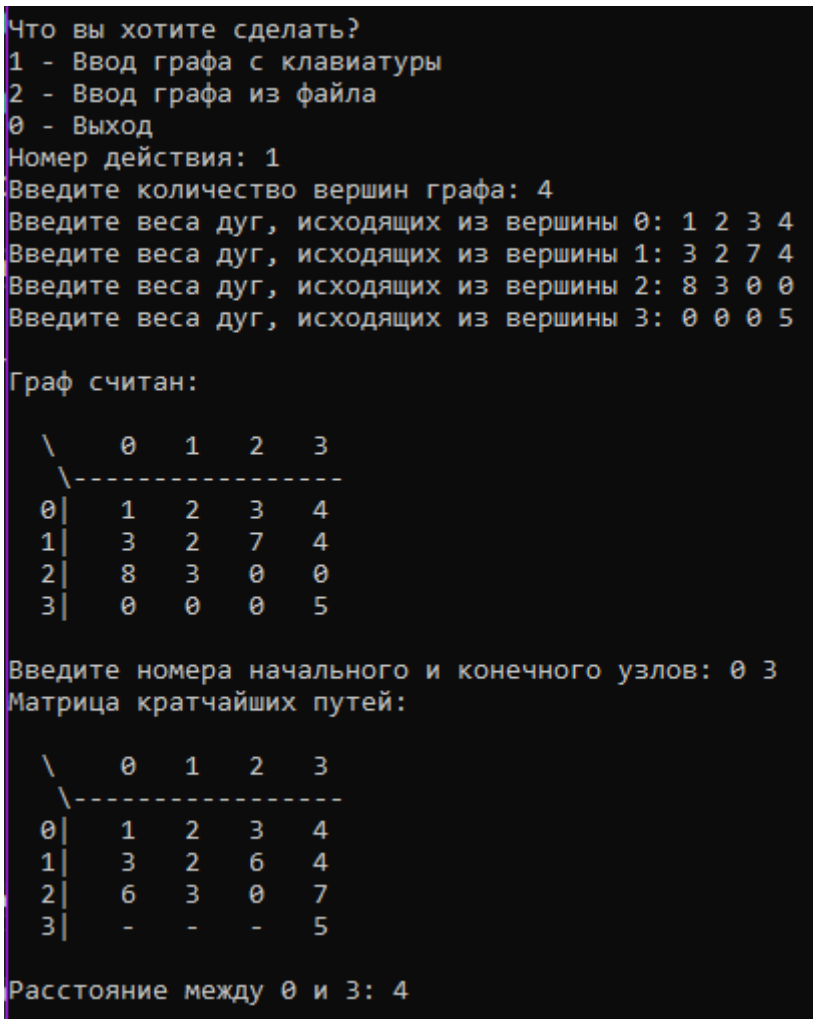


Рисунок 23 - Тест на придуманном графе

# ВЫВОД

Бинарные деревья поиска – удобные структуры данных, совмещающие удобство восприятия человеком, а также позволяют для разных целей совершать разные обходы.

Графы – очень многофункциональный инструмент (структура данных), позволяющий выражать сложные нелинейные отношения между элементами предметной области. Очень многие задачи являются нелинейными, а следовательно, нерешаемыми (или непредставляемыми) в виде линейных структур или деревьев, благодаря чему графы являются неотъемлимой частью решения таких задач.

# СПИСОК ЛИТЕРАТУРЫ


1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих, 2017. – С. 100-126.
2. Кормен Т.Х. и др. Алгоритмы. Построение и анализ, 2013. – С. 285-318.
3. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.
4. Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 01.09.2021).
5. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2021).