

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«МИРЭА – Российский технологический университет»**

**РТУ МИРЭА**

|  |
| --- |
|  |

**ОТЧЕТ**

**ПО ПРАКТИЧЕСКОЙ РАБОТЕ №5**

Основные алгоритмы работы с графами

**по дисциплине**

«Структуры и алгоритмы обработки данных»

Выполнил студент группы ИКБО-01-21 Луковников Д.Р.

Принял преподаватель Туманова М.Б.

Практическая «\_\_»\_\_\_\_\_\_\_2022 г. \_\_\_\_\_\_\_\_\_\_

работа выполнена

«Зачтено» «\_\_»\_\_\_\_\_\_\_2022 г. \_\_\_\_\_\_\_\_\_\_

Москва 2022

# СОДЕРЖАНИЕ

[ЦЕЛЬ РАБОТЫ 3](#_Toc118042674)

[ХОД РАБОТЫ 4](#_Toc118042675)

[1.1 Вставка элемента и балансировка 5](#_Toc118042676)

[1.2 Прямо обход дерева 7](#_Toc118042677)

[1.3 Симметричный обход дерева 7](#_Toc118042678)

[1.4 Сумма листьев 7](#_Toc118042679)

[1.5 Среднее арифметическое всех узлов 8](#_Toc118042680)

[1.6 Вывод дерева 8](#_Toc118042681)

[ТЕСТИРИВАНИЕ 10](#_Toc118042682)

[ВЫВОДЫ 13](#_Toc118042683)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 14](#_Toc118042684)

[ПРИЛОЖЕНИЯ 15](#_Toc118042685)

# ЦЕЛЬ РАБОТЫ

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

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.

# ХОД РАБОТЫ

**Индивидуальный вариант:** Нахождение кратчайшего пути методом Йена, граф представлен на рисунке 1.

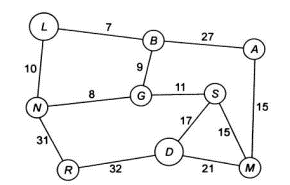


Рисунок 1 – Данный граф

**Математическая модель решения:** общая идея алгоритма представляет собой следующее, у нас будут кандидаты на кротчайшие пути. Находим первый кратчайший путь и так как ни один из последующих путей, не должен совпадать с данным, то все новые пути должны не включать в себя хотя бы одно ребро из первоначального. Поэтому исключаем по одному ребру из первого пути и находим кратчайшие пути в получаемых графах.

Листинг 1.1 – Структура узла

struct Node {

/\*

\* Структура, описывающая вершину графа

\*/

string name;

};

Граф хранится в виде матрицы смежности, можно было бы обойтись только ей, но по заданию вершины графа имеют «имена».

Листинг 1.2 – Структура графа

class Graph {

/\*

\* Класс, описывающий граф

\*/

int size; // Количество вершин графа

vector<vector<int>> matrix; // Матрица смежности

vector<Node \*> nodes; // Список вершин графа

...

## **Добавление узла**

При добавлении нового узла мы создаём новую объект узла, добавляем его в контейнер, увеличиваем количество узло и изменяем размер матрицы смежности.

Листинг 1.3 – Добавление узла

void addNode(string name) {

/\*

\* Добавление вершины в граф

\*/

Node \*node = new Node;

node->name = std::move(name);

nodes.push\_back(node);

size++;

matrix.resize(size);

for (int i = 0; i < size; i++) {

matrix[i].resize(size);

}

}

## **Добавление ребра**

При добавлении ребра или, иначе говоря, пути между двумя ребрами, нам нужны их названия и само значение дистанции, так же учитываем момент, что в нашем графе рёбра не имеют направления и двигаться по ним можно как в одну, так и в другую сторону, Листинг 1.4.

Листинг 1.4 – Добавление ребра

void addEdge(string name1, string name2, int distance) {

/\*

\* Добавление ребра в граф

\*/

int index1 = -1;

int index2 = -1;

for (int i = 0; i < size; i++) {

if (nodes[i]->name == name1) {

index1 = i;

}

if (nodes[i]->name == name2) {

index2 = i;

}

}

if (index1 == -1 || index2 == -1) {

cout << "Error: node not found" << endl;

return;

}

matrix[index1][index2] = distance;

matrix[index2][index1] = distance;

}

## **Поиск кратчайшего пути**

Для начала создаём 2 контейнера, в которых будет хранить расстояния от начально вершины до всех остальных, а также контейнер с уже посещёнными вершинами. И затем начинаем перебор начиная с начальной вершины, Листинг 1.5.

Листинг 1.5 – Поиск кратчайшего пути

int searchRoute(string name, string name2) {

/\*

\* Поиск кратчайшего пути между двумя вершинами

\*/

int index = getIndex(name); // Индекс начальной вершины

int index2 = getIndex(name2); // Индекс конечной вершины

if (index == -1 || index2 == -1) { // Если вершины не найдены

cout << "Error: node not found" << endl;

return -1;

}

// Массив, содержащий расстояния от начальной вершины до всех остальных

vector<int> distances(size, -1);

// Массив, содержащий индексы вершин, которые уже были просмотрены

vector<int> visited(size, 0);

distances[index] = 0;

visited[index] = 1;

int current = index; // Начинаем с начальной вершины

// Пока не просмотрены все вершины

while (current != index2) {

// Просматриваем все ребра, исходящие из текущей вершины

for (int i = 0; i < size; i++) {

// Если ребро существует и вершина еще не была просмотрена

if (matrix[current][i] != 0 && visited[i] == 0) {

// Если расстояние до вершины еще не было вычислено, либо вычисленное расстояние больше, чем расстояние через текущую вершину

if (distances[i] == -1 || distances[i] > distances[current] + matrix[current][i]) {

// Обновляем расстояние до вершины

distances[i] = distances[current] + matrix[current][i];

}

}

}

// Ищем вершину с минимальным расстоянием

int min = -1;

// Просматриваем все вершины

for (int i = 0; i < size; i++) {

// Если вершина еще не была просмотрена и расстояние до нее было вычислено

if (visited[i] == 0 && distances[i] != -1) {

Продолжение Листинга 1.5

// Если минимальное расстояние еще не было вычислено, либо вычисленное минимальное расстояние больше, чем расстояние до текущей вершины

if (min == -1 || distances[i] < distances[min]) {

min = i;

}

}

}

// Если минимальное расстояние не было вычислено, значит путь не существует

if (min == -1) {

break;

}

current = min;

// Помечаем вершину как просмотренную

visited[current] = 1;

}

return distances[index2];

}

## **Вывод графа**

Вывод графа реализуем с помощью языка graphviz, Листинг 1.6.

Листинг 1.6 – Вывод графа

void print() {

/\*

\* Вывод графа на экран

\*/

vector<string> printedTo;

vector<string> printedFrom;

cout << "digraph G {" << endl;

for (int i = 0; i < size; i++) {

for (int j = i; j < size; j++) {

if (matrix[i][j] != 0) {

string from = nodes[i]->name;

string to = nodes[j]->name;

if (find(printedTo.begin(), printedTo.end(), to) == printedTo.end() ||

find(printedFrom.begin(), printedFrom.end(), from) == printedFrom.end()) {

cout << " " << from << " -> " << to << " [label=\"" << matrix[i][j] << "\", arrowhead=none];" << endl;

printedTo.push\_back(to);

printedFrom.push\_back(from);

}

}

}

}

cout << "}" << endl;

}

# ТЕСТИРИВАНИЕ

Для избежания ошибок при вводе графа, он прописан внутри программы, а не вынесен в текстовое меню, Листинг 2.1.

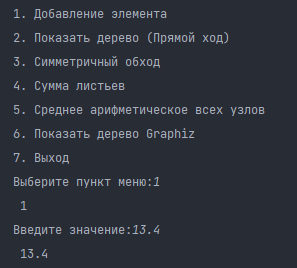


Рисунок 1 – Вставка элементов

Для удобства проверки сгенерируем граф и отрисуем его, в силу того, что это граф, а не дерево, для узлов, которые имеют 1-го nil наследника, в графе не будет отображаться левый или правый этот наследник, Рисунок 2-3.

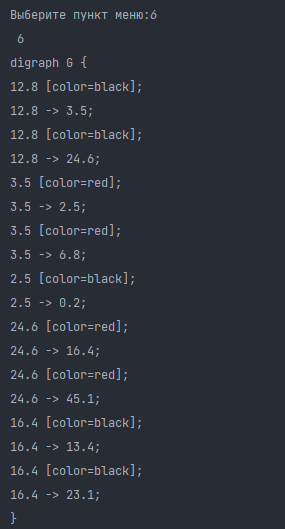


Рисунок 2 – Отрисовка графа

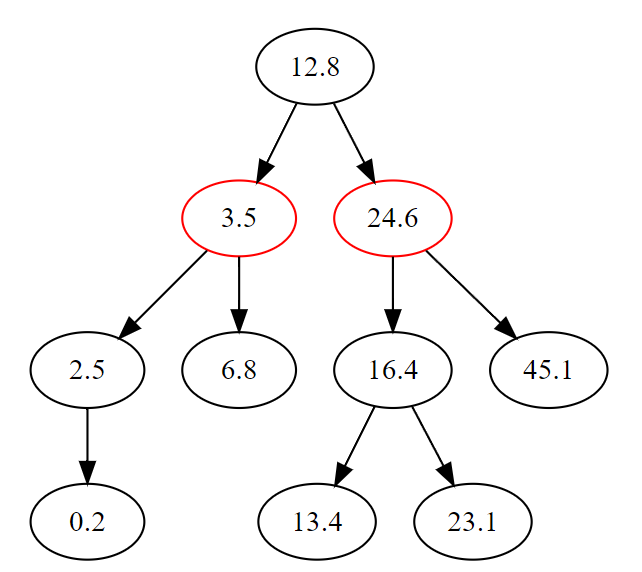


Рисунок 3 – Дерево в виде графа

Далее проверим прямой и симметричный обход, Рисунок 4-5.

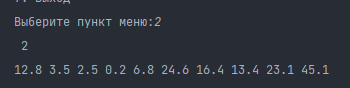


Рисунок 4 – Прямой обход

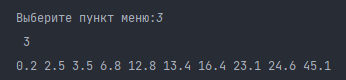


Рисунок 5 – Симметричный обход

Сравнив вывод с графом, видим, что всё работает верно. Сумма листьев и среднее арифметическое всех узлов, Рисунок 6-7.



Рисунок 6 – Сумма листьев



Рисунок 7 – Среднее арифметическое всех узлов

Все значение выведены верно, программа работает корректно.

# ВЫВОДЫ

При выполнении работы были получены навыки реализации красно-чёрного бинарного дерева поиска и создание процедур для работы с данным деревом. Программа была проверена на работоспособность и полностью протестирована.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.
2. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 01.09.2021).
3. 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).

# ПРИЛОЖЕНИЯ

Приложение А – Исходный код программы Красно-чёрного бинарного дерева

**Приложение А**

Исходный код программы Красно-чёрного бинарного дерева

Листинг 7.1 – main.cpp

/\*

\* 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.

\*

\* Вариант 23

\* Тип значения узла - Вещественное число.

\* Тип дерева - Красно-черное дерево.

\* Процедуры:

\* 1. Вставка узла в дерево и балансировка

\* 2. Прямой обход дерева

\* 3. Симметричный обход дерева

\* 4. Найти сумму значений листьев дерева

\* 5. Найти среднее арифметическое значений всех узлов

\*

\*

\* Правила:

\* 1. Корень дерева всегда черный.

\* 2. Все листья дерева (NIL) черные.

\* 3. Если узел красный, то оба его потомка черные.

\* 4. Глубина в черных узлах одинакова.

\*

\* https://dreampuf.github.io/GraphvizOnline/

\*

\*/

#include <iostream>

#include <vector>

using namespace std;

struct Node {

double value; // Node value

Node \*left, \*right, \*parent; // Left, right and parent nodes

bool color; // false - black, true - red

};

class Tree {

Node \*root;

Node \*nil;

public:

Tree() {

/\*

\* Constructor for Tree class

\*/

Продолжение Листинг 7.1

nil = new Node;

nil->color = false;

nil->left = nil;

nil->right = nil;

nil->parent = nil;

root = nil;

}

void add(double value) {

/\*

\* Add new node to tree

\*/

Node \*newNode = new Node;

newNode->value = value;

newNode->left = nil;

newNode->right = nil;

newNode->color = true; // Red

/\*

\* Find place for new node

\*/

Node \*current = root;

Node \*parent = nil;

while (current != nil) {

parent = current;

if (newNode->value < current->value) {

current = current->left;

} else {

current = current->right;

}

}

newNode->parent = parent;

if (parent == nil) {

root = newNode;

} else if (newNode->value < parent->value) {

parent->left = newNode;

} else {

parent->right = newNode;

}

if (newNode->parent == nil) {

newNode->color = false;

return;

}

if (newNode->parent->parent == nil) {

return;

}

fixTree(newNode);

if (root == nil) {

root = newNode;

root->color = false; // Black

}

}

Продолжение Листинг 7.1

void fixTree(Node \*node) {

/\*

\* Fix tree after adding new node

\*/

Node \*parent = nil;

Node \*grandparent = nil;

while ((node != root) && (node->color != false) && (node->parent->color == true)) {

parent = node->parent;

grandparent = node->parent->parent;

if (parent == grandparent->left) {

Node \*uncle = grandparent->right;

if (uncle->color == true) {

grandparent->color = true;

parent->color = false;

uncle->color = false;

node = grandparent;

} else {

if (node == parent->right) {

leftRotate(parent);

node = parent;

parent = node->parent;

}

rightRotate(grandparent);

swap(parent->color, grandparent->color);

node = parent;

}

} else {

Node \*uncle = grandparent->left;

if (uncle->color == true) {

grandparent->color = true;

parent->color = false;

uncle->color = false;

node = grandparent;

} else {

if (node == parent->left) {

rightRotate(parent);

node = parent;

parent = node->parent;

}

leftRotate(grandparent);

swap(parent->color, grandparent->color);

node = parent;

}

}

}

root->color = false;

}

void leftRotate(Node \*node) {

Node \*right = node->right;

node->right = right->left;

if (node->right != nil) {

node->right->parent = node;

Продолжение Листинг 7.1

}

right->parent = node->parent;

if (node->parent == nil) {

root = right;

} else if (node == node->parent->left) {

node->parent->left = right;

} else {

node->parent->right = right;

}

right->left = node;

node->parent = right;

}

void rightRotate(Node \*node) {

Node \*left = node->left;

node->left = left->right;

if (node->left != nil) {

node->left->parent = node;

}

left->parent = node->parent;

if (node->parent == nil) {

root = left;

} else if (node == node->parent->left) {

node->parent->left = left;

} else {

node->parent->right = left;

}

left->right = node;

node->parent = left;

}

void showGraphiz() {

cout << "digraph G {" << endl;

showTreeGraphiz();

cout << "}" << endl;

}

// Прямой обход

void showTree(Node \*node = nullptr) {

if (node == nullptr) {

node = root;

}

if (node != nil) {

cout << node->value << " ";

showTree(node->left);

showTree(node->right);

}

}

// Симметричный обход

void showTreeSymmetric(Node \*node = nullptr) {

if (node == nullptr) {

node = root;

}

Продолжение Листинг 7.1

if (node != nil) {

showTreeSymmetric(node->left);

cout << node->value << " ";

showTreeSymmetric(node->right);

}

}

// Сумма листьев

double sumLeaves(Node \*node = nullptr) {

if (node == nullptr) {

node = root;

}

if (node != nil) {

if (node->left == nil && node->right == nil) {

return node->value;

}

return sumLeaves(node->left) + sumLeaves(node->right);

}

return 0;

}

// Среднее арифметическое всех узлов

double averageNodes(Node \*node = nullptr) {

if (node == nullptr) {

node = root;

}

if (node != nil) {

return sumNodes(node) / countNodes(node);

}

return 0;

}

private:

// Количество узлов

int countNodes(Node \*node = nullptr) {

if (node == nullptr) {

node = root;

}

if (node != nil) {

return 1 + countNodes(node->left) + countNodes(node->right);

}

return 0;

}

// Сумма всех узлов

double sumNodes(Node \*node = nullptr) {

if (node == nullptr) {

node = root;

}

if (node != nil) {

return node->value + sumNodes(node->left) + sumNodes(node->right);

}

return 0;

}

// Вывод дерева в формате Graphiz

void showTreeGraphiz(Node \*node = nullptr) {

/\*

\* Show tree Graphiz

\*/

Продолжение Листинг 7.1

if (node == nullptr) {

node = root;

}

if (node != nil) {

if (node->left != nil) {

cout << node->value << " [color=" << (node->color ? "red" : "black") << "];" << endl;

cout << node->value << " -> " << node->left->value << ";" << endl;

}

if (node->right != nil) {

cout << node->value << " [color=" << (node->color ? "red" : "black") << "];" << endl;

cout << node->value << " -> " << node->right->value << ";" << endl;

}

showTreeGraphiz(node->left);

showTreeGraphiz(node->right);

}

}

};

int main() {

system("chcp 65001");

Tree tree;

vector<double> values = {

1.3, 2.3, 3.3, 4.3, 5.3, 6.3, 7.3, 8.3, 9.3, 10.3,

// 11.3, 12.3, 13.3, 14.3, 15.3, 16.3, 17.3, 18.3, 19.3, 20.3,

// 21.3, 22.3, 23.3, 24.3, 25.3, 26.3, 27.3, 28.3, 29.3, 30.3,

// 31.3, 32.3, 33.3, 34.3, 35.3, 36.3, 37.3, 38.3, 39.3, 40.3,

};

// // Перемешать значения

// for (int i = 0; i < values.size(); i++) {

// int index = rand() % values.size();

// double temp = values[i];

// values[i] = values[index];

// values[index] = temp;

// }

// for (double value: values) {

// tree.add(value);

// }

// Текстовое меню

int choice = 0;

while (choice != 7) {

cout << "1. Добавление элемента" << endl;

cout << "2. Показать дерево (Прямой ход)" << endl;

cout << "3. Симметричный обход" << endl;

cout << "4. Сумма листьев" << endl;

cout << "5. Среднее арифметическое всех узлов" << endl;

cout << "6. Показать дерево Graphiz" << endl;

cout << "7. Выход" << endl;

cout << "Выберите пункт меню: ";

cin >> choice;

switch (choice) {

case 1: {

double value;

cout << "Введите значение: ";

Продолжение Листинг 7.1

cin >> value;

tree.add(value);

break;

}

case 2: {

tree.showTree();

cout << endl;

break;

}

case 3: {

tree.showTreeSymmetric();

cout << endl;

break;

}

case 4: {

cout << "Сумма листьев: " << tree.sumLeaves() << endl;

break;

}

case 5: {

cout << "Среднее арифметическое всех узлов: " << tree.averageNodes() << endl;

break;

}

case 6: {

tree.showGraphiz();

break;

}

case 7: {

break;

}

default: {

cout << "Неверный пункт меню" << endl;

break;

}

}

}

return 0;

}