Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

Кафедра «Информационные технологии и автоматизированные системы»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

Тема: «Бинарное дерево поиска»

Семестр 2

Работу выполнила

Студентка группы РИС-23-2Б

Виноградова Юлия Дмитриевна

Проверила

Доцент кафедры ИТАС

Полякова Ольга Андреевна

Пермь 2024

**Постановка задачи**

1. Самостоятельно придумать вид Дерева и реализовать алгоритмы для этого собственного варианта бинарного дерева поиска, имеющего не менее трёх уровней.
2. Алгоритмы:
   1. Необходимо реализовать функции для редактирования дерева:
      1. Вставка узла.
      2. Удаление узла.
      3. Поиск элемента по ключу.
   2. Реализовать алгоритмы обхода дерева:
      1. Прямой
      2. Симметричный
      3. Обратный
   3. Выполнить задание своего варианта из методички: Laby\_Chast\_3.docx.
3. Реализовать алгоритм балансировки дерева.
4. Реализовать вертикальную и горизонтальную печать.
5. Визуализацию дерева выполнить с использованием любой доступной графической библиотеки – SFML, OpenGL.
6. Пользовательский интерфейс – на усмотрение разработчика - с условием кроссплатформенности (например, Windows Forms; также поощряется использование Qt или иных фреймворков).

**Реализация в консоли**

#include <iostream>

#include <string>

#include <queue>

#include <algorithm>

using namespace std;

// Определение класса узла дерева

class Node {

public:

int key; // Ключ узла

Node\* left; // Указатель на левого потомка

Node\* right; // Указатель на правого потомка

int height; // Высота узла в дереве

// Конструктор узла

Node(int key) {

this->key = key;

this->left = nullptr;

this->right = nullptr;

this->height = 1; // При создании узла, его высота равна 1

}

};

// Определение класса бинарного дерева

class BinaryTree {

public:

Node\* root; // Указатель на корень дерева

// Конструктор бинарного дерева

BinaryTree() {

root = nullptr; // При создании дерева корень равен nullptr

}

// Функция для вставки узла в дерево

Node\* insert(Node\* root, int key) {

if (root == nullptr) {

return new Node(key); // Создание нового узла, если текущий узел пустой

}

if (key < root->key) {

root->left = insert(root->left, key); // Рекурсивная вставка в левое поддерево

}

else if (key > root->key) {

root->right = insert(root->right, key); // Рекурсивная вставка в правое поддерево

}

return root;

}

// Функция для поиска минимального узла в дереве

Node\* findMin(Node\* node) {

while (node->left != nullptr) {

node = node->left; // Проход по левым узлам для нахождения минимального узла

}

return node;

}

// Функция для удаления узла из дерева

Node\* deleteNode(Node\* root, int value) {

if (root == nullptr) {

return root; // Если дерево пустое, возвращаем nullptr

}

if (value < root->key) {

root->left = deleteNode(root->left, value); // Рекурсивное удаление из левого поддерева

}

else if (value > root->key) {

root->right = deleteNode(root->right, value); // Рекурсивное удаление из правого поддерева

}

else {

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp; // Узел без левого потомка

}

else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp; // Узел без правого потомка

}

Node\* temp = findMin(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key); // Замена удаляемого узла минимальным узлом из правого поддерева

}

return root;

}

// Функция для поиска узла с заданным значением в дереве

Node\* search(Node\* root, int value) {

if (root == nullptr || root->key == value) {

return root; // Возвращаем узел, если он равен искомому значению или если дерево пустое

}

if (value < root->key) {

return search(root->left, value); // Рекурсивный поиск в левом поддереве, если значение меньше ключа текущего узла

}

else {

return search(root->right, value); // Рекурсивный поиск в правом поддереве, если значение больше ключа текущего узла

}

}

// Функция для получения высоты узла в дереве

int getHeight(Node\* node) {

if (node == nullptr) {

return 0; // Возвращаем 0, если узел пустой

}

return node->height;

}

// Функция для обновления высоты узла на основе высот его потомков

void updateHeight(Node\* node) {

if (node == nullptr) {

return; // Ничего не делаем, если узел пустой

}

node->height = 1 + max(getHeight(node->left), getHeight(node->right)); // Обновляем высоту узла

}

// Функция для преобразования дерева в AVL-дерево (балансировка)

Node\* transformToAVL(Node\* root) {

if (root == nullptr) {

return root; // Возвращаем nullptr, если дерево пустое

}

root->left = transformToAVL(root->left); // Рекурсивное преобразование левого поддерева

root->right = transformToAVL(root->right); // Рекурсивное преобразование правого поддерева

updateHeight(root); // Обновляем высоту текущего узла

int balance = getHeight(root->left) - getHeight(root->right); // Вычисляем баланс текущего узла

if (balance > 1) { // Необходимо правое вращение

if (getHeight(root->left->right) > getHeight(root->left->left)) {

root->left = leftRotate(root->left); // Производим левое вращение для левого потомка

}

root = rightRotate(root); // Правое вращение для текущего узла

}

else if (balance < -1) { // Необходимо левое вращение

if (getHeight(root->right->left) > getHeight(root->right->right)) {

root->right = rightRotate(root->right); // Производим правое вращение для правого потомка

}

root = leftRotate(root); // Левое вращение для текущего узла

}

return root; // Возвращаем корень преобразованного дерева

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y; // Поворачиваем узлы

y->left = T2; // Обновляем левое поддерево узла y

updateHeight(y); // Обновляем высоту узла y

updateHeight(x); // Обновляем высоту узла x

return x; // Возвращаем новый корень поддерева

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x; // Поворачиваем узлы

x->right = T2; // Обновляем правое поддерево узла x

updateHeight(x); // Обновляем высоту узла x

updateHeight(y); // Обновляем высоту узла y

return y; // Возвращаем новый корень поддерева

}

// Прямой обход (pre-order traversal)

void preOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

std::cout << root->key << " "; // Печатаем значение узла

preOrderTraversal(root->left); // Рекурсивно обходим левое поддерево

preOrderTraversal(root->right); // Рекурсивно обходим правое поддерево

}

// Симметричный обход (in-order traversal)

void inOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

inOrderTraversal(root->left); // Рекурсивно обходим левое поддерево

std::cout << root->key << " "; // Печатаем значение узла

inOrderTraversal(root->right); // Рекурсивно обходим правое поддерево

}

// Обратный обход (post-order traversal)

void postOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

postOrderTraversal(root->left); // Рекурсивно обходим левое поддерево

postOrderTraversal(root->right); // Рекурсивно обходим правое поддерево

std::cout << root->key << " "; // Печатаем значение узла

}

// Вертикальная печать

void printVertical(Node\* root, int level = 0, char ch = ' ') {

// Рекурсивная функция для печати дерева с использованием символов для указания отношений между узлами

if (root->right != nullptr) printVertical(root->right, level + 1, '/'); // Печать правого поддерева

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

std::cout << " "; // Отступ для наглядности уровня узла в дереве

}

std::cout << ch << "-- " << root->key << std::endl; // Вывод узла с символом и ключом

if (root->left != nullptr) printVertical(root->left, level + 1, '\\'); // Печать левого поддерева

}

// Горизонтальная печать

void printHorizontal(const Node\* root, const string& symbol = "", bool rootFlag = true, bool last = true) {

// Рекурсивная функция для горизонтальной печати дерева с использованием символов для указания отношений между узлами

cout << symbol << (rootFlag ? "" : (last ? "+--" : "|--")) << (root ? to\_string(root->key) : "") << endl; // Вывод узла с символами для связей

if (!root) { return; }

vector<Node\*> v = { root->left, root->right }; // Создание вектора дочерних узлов

for (size\_t i = 0; i < v.size(); i++) {

printHorizontal(v[i], symbol + (rootFlag ? "" : (last ? " " : "| ")), false, i + 1 >= v.size()); // Рекурсивный вызов для каждого дочернего узла

}

}

};

int main() {

system("chcp 1251 > null");

BinaryTree bst;

while (true) {

char choice;

cout << "----------------------" << endl;

cout << " Меню " << endl;

cout << "----------------------" << endl;

cout << "1. Вставить узел" << endl;

cout << "2. Удалить узел" << endl;

cout << "3. Вывести дерево" << endl;

cout << "0. Выход" << endl;

cout << "----------------------" << endl;

cout << "Выберите действие: ";

cin >> choice;

if (choice == '0') {

break;

}

else if (choice == '1') {

int key;

cout << "Введите значение для вставки: ";

cin >> key;

bst.root = bst.insert(bst.root, key);

cout << "Узел успешно вставлен!" << endl;

}

else if (choice == '2') {

int key;

cout << "Введите значение для удаления: ";

cin >> key;

bst.root = bst.deleteNode(bst.root, key);

cout << "Узел успешно удален!" << endl;

}

else if (choice == '3') {

bst.printVertical(bst.root);

system("pause");

}

system("cls");

}

system("cls");

while (true) {

char choice;

cout << "----------------------" << std::endl;

cout << " Меню " << std::endl;

cout << "----------------------" << std::endl;

cout << "1. Найти узел по ключу" << std::endl;

cout << "2. Найти минимум" << std::endl;

cout << "0. Выход" << std::endl;

cout << "----------------------" << std::endl;

cout << "Выберите действие: ";

cin >> choice;

if (choice == '0') {

break;

}

else if (choice == '1') {

int key;

cout << "Введите ключ: ";

cin >> key;

Node\* result = bst.search(bst.root, key);

if (result != nullptr) {

cout << "Значение найдено в дереве!" << std::endl;

}

else {

cout << "Значение не найдено в дереве." << std::endl;

}

system("pause");

}

else if (choice == '2') {

Node\* result = bst.findMin(bst.root);

if (result != nullptr) {

cout << "Минимальное значение в дереве: " << result->key << std::endl;

}

else {

cout << "Дерево пустое" << std::endl;

}

system("pause");

}

system("cls");

}

system("cls");

while (true) {

char choice;

cout << "----------------------" << endl;

cout << " Меню " << endl;

cout << "----------------------" << endl;

cout << "1. Сбалансировать дерево" << endl;

cout << "2. Вывести дерево" << endl;

cout << "0. Выход" << endl;

cout << "----------------------" << endl;

cout << "Выберите действие: ";

cin >> choice;

if (choice == '0') {

break;

}

else if (choice == '1') {

bst.root = bst.transformToAVL(bst.root);

cout << "Дерево успешно сбалансировано!" << endl;

system("pause");

}

else if (choice == '2') {

bst.printVertical(bst.root);

system("pause");

}

system("cls");

}

system("cls");

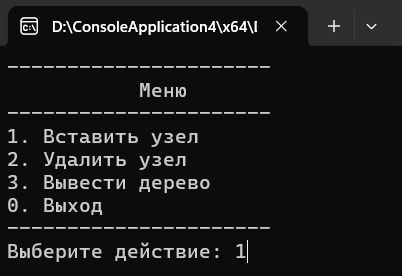
return 0;

}

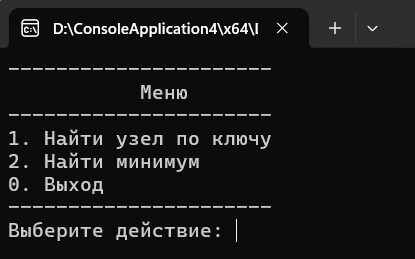
**Работа программы**

Реализовано три вида меню:

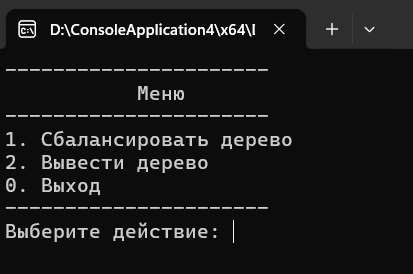
1. Создание дерева



1. Работа с деревом



1. Балансировка дерева



Все функции меню работают.

**Реализация в Qt**

#include <QApplication>

#include <QWidget>

#include <QPushButton>

#include <QHBoxLayout>

#include <QVBoxLayout>

#include <QLabel>

#include <QInputDialog>

#include <QPainter>

#include <QDebug>

#include <QMessageBox>

// Структура узла бинарного дерева

struct Node {

int key;

Node\* left;

Node\* right;

int height;

Node(int value) : key(value), left(nullptr), right(nullptr), height(1) {}

};

class BinaryTreeWidget : public QWidget {

public:

BinaryTreeWidget(QWidget\* parent = nullptr) : QWidget(parent), root(nullptr) {}

void insertNode(int key) {

root = insert(root, key);

update();

}

void deleteNode(int key) {

root = deleteNode(root, key);

update();

}

bool searchNode(int key) {

return search(root, key);

}

void preOrderTraversal() {

preOrder(root);

}

void inOrderTraversal() {

inOrder(root);

}

void postOrderTraversal() {

postOrder(root);

}

void balanceTree() {

root = transformToAVL(root);

update();

}

// Метод отрисовки виджета

void paintEvent(QPaintEvent\* event) override {

Q\_UNUSED(event); // Подавляем предупреждение о неиспользуемом параметре события отрисовки

QPainter painter(this); // Создаем объект QPainter для отрисовки на виджете

painter.setRenderHint(QPainter::Antialiasing); // Устанавливаем сглаживание для рисования

int initialX = width() / 2; // Вычисляем начальное положение X для отрисовки корня дерева по центру виджета

int initialY = 50; // Определяем начальное положение Y для отрисовки корня дерева

drawTree(painter, initialX, initialY, root, 1); // Вызываем функцию для отрисовки всего дерева, начиная с корня

}

private:

Node\* root;

// Функция для вставки узла в дерево

Node\* insert(Node\* root, int key) {

if (root == nullptr) {

return new Node(key); // Создание нового узла, если текущий узел пустой

}

if (key < root->key) {

root->left = insert(root->left, key); // Рекурсивная вставка в левое поддерево

}

else if (key > root->key) {

root->right = insert(root->right, key); // Рекурсивная вставка в правое поддерево

}

return root;

}

// Функция для поиска минимального узла в дереве

Node\* findMin(Node\* node) {

while (node->left != nullptr) {

node = node->left; // Проход по левым узлам для нахождения минимального узла

}

return node;

}

Node\* deleteNode(Node\* node, int key) {

if (node == nullptr) {

return nullptr;

}

if (key < node->key) {

node->left = deleteNode(node->left, key);

} else if (key > node->key) {

node->right = deleteNode(node->right, key);

} else {

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

delete node;

return nullptr;

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

Node\* temp = node->right;

delete node;

return temp;

} else if (node->right == nullptr) {

Node\* temp = node->left;

delete node;

return temp;

} else {

Node\* minNode = findMin(node->right);

node->key = minNode->key;

node->right = deleteNode(node->right, minNode->key);

}

}

return node;

}

// Функция для поиска узла с заданным значением в дереве

Node\* search(Node\* root, int value) {

if (root == nullptr || root->key == value) {

return root; // Возвращаем узел, если он равен искомому значению или если дерево пустое

}

if (value < root->key) {

return search(root->left, value); // Рекурсивный поиск в левом поддереве, если значение меньше ключа текущего узла

}

else {

return search(root->right, value); // Рекурсивный поиск в правом поддереве, если значение больше ключа текущего узла

}

}

// Прямой обход (pre-order traversal)

void preOrder(Node\* root) {

if (root == nullptr) {

return;

}

qDebug() << root->key; // Печатаем значение узла

preOrder(root->left); // Рекурсивно обходим левое поддерево

preOrder(root->right); // Рекурсивно обходим правое поддерево

}

// Симметричный обход (in-order traversal)

void inOrder(Node\* root) {

if (root == nullptr) {

return;

}

inOrder(root->left); // Рекурсивно обходим левое поддерево

qDebug() << root->key; // Печатаем значение узла

inOrder(root->right); // Рекурсивно обходим правое поддерево

}

// Обратный обход (post-order traversal)

void postOrder(Node\* root) {

if (root == nullptr) {

return;

}

postOrder(root->left); // Рекурсивно обходим левое поддерево

postOrder(root->right); // Рекурсивно обходим правое поддерево

qDebug() << root->key; // Печатаем значение узла

}

// Функция для получения высоты узла в дереве

int getHeight(Node\* node) {

if (node == nullptr) {

return 0; // Возвращаем 0, если узел пустой

}

return node->height;

}

// Функция для обновления высоты узла на основе высот его потомков

void updateHeight(Node\* node) {

if (node == nullptr) {

return; // Ничего не делаем, если узел пустой

}

node->height = 1 + std::max(getHeight(node->left), getHeight(node->right)); // Обновляем высоту узла

}

// Функция для преобразования дерева в AVL-дерево (балансировка)

Node\* transformToAVL(Node\* root) {

if (root == nullptr) {

return root; // Возвращаем nullptr, если дерево пустое

}

root->left = transformToAVL(root->left); // Рекурсивное преобразование левого поддерева

root->right = transformToAVL(root->right); // Рекурсивное преобразование правого поддерева

updateHeight(root); // Обновляем высоту текущего узла

int balance = getHeight(root->left) - getHeight(root->right); // Вычисляем баланс текущего узла

if (balance > 1) { // Необходимо правое вращение

if (getHeight(root->left->right) > getHeight(root->left->left)) {

root->left = leftRotate(root->left); // Производим левое вращение для левого потомка

}

root = rightRotate(root); // Правое вращение для текущего узла

}

else if (balance < -1) { // Необходимо левое вращение

if (getHeight(root->right->left) > getHeight(root->right->right)) {

root->right = rightRotate(root->right); // Производим правое вращение для правого потомка

}

root = leftRotate(root); // Левое вращение для текущего узла

}

return root; // Возвращаем корень преобразованного дерева

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y; // Поворачиваем узлы

y->left = T2; // Обновляем левое поддерево узла y

updateHeight(y); // Обновляем высоту узла y

updateHeight(x); // Обновляем высоту узла x

return x; // Возвращаем новый корень поддерева

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x; // Поворачиваем узлы

x->right = T2; // Обновляем правое поддерево узла x

updateHeight(x); // Обновляем высоту узла x

updateHeight(y); // Обновляем высоту узла y

return y; // Возвращаем новый корень поддерева

}

void drawTree(QPainter& painter, int x, int y, Node\* node, int level) {

if (node == nullptr) {

return; // Если узел пуст, завершаем функцию

}

int radius = 20; // Радиус узла

int spacing = 100; // Расстояние между узлами на одном уровне

int levelHeight = 80; // Высота уровня

// Рисуем круг для узла

painter.setPen(Qt::black);

painter.setBrush(Qt::white);

painter.drawEllipse(QPointF(x, y), radius, radius);

// Помещаем текст (ключ узла) в центр круга

painter.setPen(Qt::black);

painter.drawText(QRectF(x - radius, y - radius, 2 \* radius, 2 \* radius), Qt::AlignCenter, QString::number(node->key));

if (node->left != nullptr) {

// Рассчитываем координаты для левого потомка

int leftX = x - spacing;

int leftY = y + levelHeight;

// Рисуем линию от текущего узла к левому потомку

painter.drawLine(x, y + radius, leftX, leftY);

// Рекурсивно вызываем drawTree для левого потомка

drawTree(painter, leftX, leftY, node->left, level + 1);

}

if (node->right != nullptr) {

// Рассчитываем координаты для правого потомка

int rightX = x + spacing;

int rightY = y + levelHeight;

// Рисуем линию от текущего узла к правому потомку

painter.drawLine(x, y + radius, rightX, rightY);

// Рекурсивно вызываем drawTree для правого потомка

drawTree(painter, rightX, rightY, node->right, level + 1);

}

}

};

int main(int argc, char\* argv[]) {

QApplication app(argc, argv);

// Создание виджета для отображения бинарного дерева

BinaryTreeWidget binaryTreeWidget;

// Кнопка для вставки узла

QPushButton insertButton("Вставить узел");

QObject::connect(&insertButton, &QPushButton::clicked, [&binaryTreeWidget]() {

bool ok;

int key = QInputDialog::getInt(nullptr, "Вставка узла", "Ключ:", 0, INT\_MIN, INT\_MAX, 1, &ok);

if (ok) {

binaryTreeWidget.insertNode(key);

}

});

// Кнопка для удаления узла

QPushButton deleteButton("Удалить узел");

QObject::connect(&deleteButton, &QPushButton::clicked, [&binaryTreeWidget]() {

bool ok;

int key = QInputDialog::getInt(nullptr, "Удаление узла", "Ключ:", 0, INT\_MIN, INT\_MAX, 1, &ok);

if (ok) {

binaryTreeWidget.deleteNode(key);

}

});

// Кнопка для поиска по ключу

QPushButton searchButton("Поиск по ключу");

QObject::connect(&searchButton, &QPushButton::clicked, [&binaryTreeWidget]() {

bool ok;

int key = QInputDialog::getInt(nullptr, "Поиск по ключу", "Ключ:", 0, INT\_MIN, INT\_MAX, 1, &ok);

if (ok) {

bool found = binaryTreeWidget.searchNode(key);

QString message = found ? "Узел найден!" : "Узел не найден!";

QMessageBox::information(nullptr, "Результат поиска", message);

}

});

// Кнопки для обхода дерева

QPushButton preOrderButton("Прямой обход");

QObject::connect(&preOrderButton, &QPushButton::clicked, [&binaryTreeWidget]() {

binaryTreeWidget.preOrderTraversal();

});

QPushButton inOrderButton("Симметричный обход");

QObject::connect(&inOrderButton, &QPushButton::clicked, [&binaryTreeWidget]() {

binaryTreeWidget.inOrderTraversal();

});

QPushButton postOrderButton("Обратный обход");

QObject::connect(&postOrderButton, &QPushButton::clicked, [&binaryTreeWidget]() {

binaryTreeWidget.postOrderTraversal();

});

// Кнопка для балансировки дерева

QPushButton balanceButton("Сбалансировать дерево");

QObject::connect(&balanceButton, &QPushButton::clicked, [&binaryTreeWidget]() {

binaryTreeWidget.balanceTree();

});

// Создание основного Layout

QVBoxLayout layout;

layout.addWidget(&binaryTreeWidget);

// Создание Layout для кнопок

QHBoxLayout buttonLayout;

buttonLayout.addWidget(&insertButton);

buttonLayout.addWidget(&deleteButton);

buttonLayout.addWidget(&searchButton);

buttonLayout.addWidget(&preOrderButton);

buttonLayout.addWidget(&inOrderButton);

buttonLayout.addWidget(&postOrderButton);

buttonLayout.addWidget(&balanceButton);

// Создание основного виджета и установка Layout кнопок

QWidget mainWidget;

mainWidget.setLayout(&buttonLayout);

// Добавление кнопочного Layout на основной Layout

layout.addWidget(&mainWidget);

// Создание окна и его отображение

QWidget window;

window.setLayout(&layout);

window.setWindowTitle("Бинарное дерево поиска");

window.resize(800, 600);

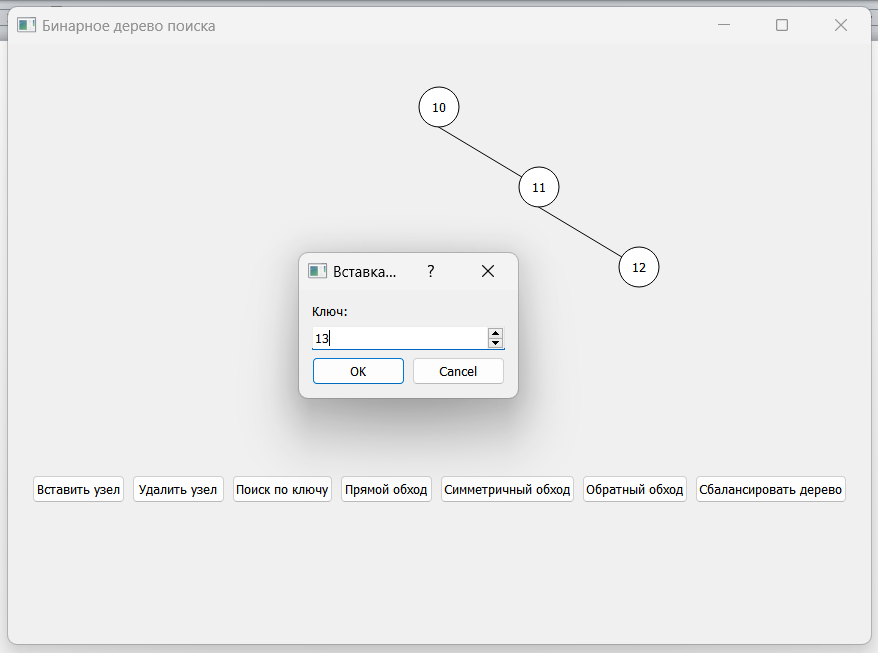
window.show();

return app.exec();

}

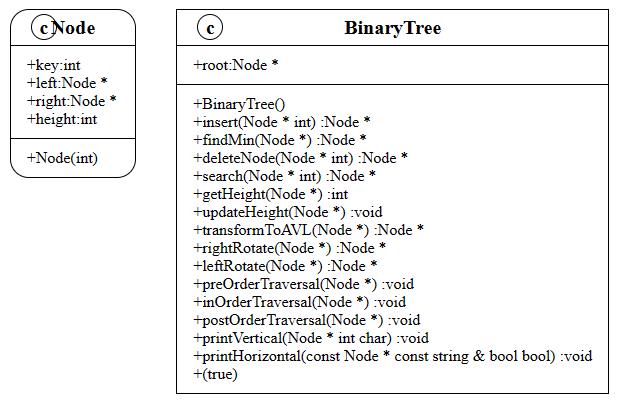
**Работа программы**

Интерфейс приложения:



Все обходы выводятся в Debug Qt.

**UML Диаграмма классов**

****