Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учереждение высшего образования

**«Пермский национальный исследовательский политехнический университет»**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

Направление подготовки: Разработка информационных систем (РИС)

**Лабораторная работа “Бинарные деревья”**

Выполнил студент гр. РИС-24-3б

Караваев Артем Андреевич

Проверил:

Доц. каф. ИТАС

Ольга Андреевна Полякова

г. Пермь, 2024

Постановка задачи:

### 25. Тип информационного поля double. Найти минимальный элемент в дереве.

Требуется реализовать алгоритмы для собственного варианта бинарного дерева поиска, имеющего не менее трёх уровней .

Алгоритмы:

1. Необходимо реализовать функции для редактирования дерева:

- Вставка узла.

- Удаление узла.

- Поиск элемента по ключу.

2. Реализовать алгоритмы обхода дерева:

2.1 Прямой

2.2 Симметричный

2.3 Обратный

3. Реализовать алгоритм балансировки дерева.

4. Реализовать вертикальную и горизонтальную печать.

5. Визуализация дерева должна быть выполнена с использованием любой доступной графической библиотеки – SFML, SDL, OpenGL и подобных.

6. Пользовательский интерфейс по усмотрению разработчика - с условием кроссплатформенности (поощряется использование Qt или иных фреймворков).

6. Построить UML- диаграмму классов

7. Выполнить отчёт

Код программы:

#include <SFML/Graphics.hpp>

#include <iostream>

#include <vector>

#include <queue>

#include <sstream>

#include <algorithm>

#include <cmath>

using namespace std;

using namespace sf;

struct TreeNode {

double data;

TreeNode\* left;

TreeNode\* right;

TreeNode(double val) : data(val), left(nullptr), right(nullptr) {}

};

class BinarySearchTree {

private:

TreeNode\* root;

TreeNode\* insertHelper(TreeNode\* node, double val) {

if (node==nullptr) return new TreeNode(val);

if (val < node->data) {

node->left=insertHelper(node->left, val);

}

else if (val >= node->data) {

node->right=insertHelper(node->right, val);

}

return node;

}

TreeNode\* findMin(TreeNode\* node) {

while (node!=nullptr && node->left!=nullptr) {

node = node->left;

}

return node;

}

TreeNode\* removeHelper(TreeNode\* node, double val) {

if (node==nullptr) return nullptr;

if (val < node->data) {

node->left = removeHelper(node->left, val);

}

else if (val > node->data) {

node->right = removeHelper(node->right, val);

}

else {

if (node->left==nullptr) {

TreeNode\* temp = node->right;

delete node;

return temp;

}

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

TreeNode\* temp = node->left;

delete node;

return temp;

}

TreeNode\* temp = findMin(node->right);

node->data = temp->data;

node->right = removeHelper(node->right, temp->data);

}

return node;

}

void destroyTree(TreeNode\* node) {

if (node!=nullptr) {

destroyTree(node->left);

destroyTree(node->right);

delete node;

}

}

TreeNode\* findHelper(TreeNode\* root, double val) {

TreeNode\* result = nullptr;

TreeNode\* current = root;

while (current != nullptr) {

if (val > current->data) {

current = current->right;

}

else if (val < current->data) {

current = current->left;

}

else {

result = current;

current = current->right;

}

}

return result;

}

void preOrderHelper(TreeNode\* node, vector<double>& result) {//прямой обход

if (node!=nullptr) {

result.push\_back(node->data);

preOrderHelper(node->left, result);

preOrderHelper(node->right, result);

}

}

void inOrderHelper(TreeNode\* node, vector<double>& result) {//симметричный

if (node!=nullptr) {

inOrderHelper(node->left, result);

result.push\_back(node->data);

inOrderHelper(node->right, result);

}

}

void postOrderHelper(TreeNode\* node, vector<double>& result) {//обратный

if (node!=nullptr) {

postOrderHelper(node->left, result);

postOrderHelper(node->right, result);

result.push\_back(node->data);

}

}

int height(TreeNode\* node) {

if (node==nullptr) return 0;

return 1 + max(height(node->left), height(node->right));

}

int balanceFactor(TreeNode\* node) {

if (node==nullptr) return 0;

return height(node->left) - height(node->right);

}

TreeNode\* buildBalancedTree(vector<double>& sorted, int start, int end) {

if (start > end) return nullptr;

int mid = (end + start) / 2;

TreeNode\* node = new TreeNode(sorted[mid]);

node->left = buildBalancedTree(sorted, start, mid - 1);

node->right = buildBalancedTree(sorted, mid + 1, end);

return node;

}

public:

BinarySearchTree() : root(nullptr) {}

~BinarySearchTree() { destroyTree(root); }

void preOrderHelperNodes(TreeNode\* node, vector<TreeNode\*>& nodes) {

if (node != nullptr) {

nodes.push\_back(node);

preOrderHelperNodes(node->left, nodes);

preOrderHelperNodes(node->right, nodes);

}

}

void inOrderHelperNodes(TreeNode\* node, vector<TreeNode\*>& nodes) {

if (node != nullptr) {

inOrderHelperNodes(node->left, nodes);

nodes.push\_back(node);

inOrderHelperNodes(node->right, nodes);

}

}

void postOrderHelperNodes(TreeNode\* node, vector<TreeNode\*>& nodes) {

if (node != nullptr) {

postOrderHelperNodes(node->left, nodes);

postOrderHelperNodes(node->right, nodes);

nodes.push\_back(node);

}

}

void insert(double val) {

root = insertHelper(root, val);

vector<double> obh;

inOrderHelper(root, obh);

root = buildBalancedTree(obh,0,obh.size()-1);

}

void remove(double val) {

root = removeHelper(root, val);

vector<double> obh;

inOrderHelper(root, obh);

root = buildBalancedTree(obh, 0, obh.size() - 1);

}

TreeNode\* find(double val) {

return findHelper(root, val);

}

TreeNode\* findMin() {

return findMin(root);

}

vector<double> preOrderTraversal() {

vector<double> result;

preOrderHelper(root, result);

return result;

}

vector<double> inOrderTraversal() {

vector<double> result;

inOrderHelper(root, result);

return result;

}

vector<double> postOrderTraversal() {

vector<double> result;

postOrderHelper(root, result);

return result;

}

TreeNode\* getRoot() {

return root;

}

};

class TreeVisualizer {

private:

BinarySearchTree& tree;

RenderWindow& window;

Font font;

vector<CircleShape> nodes;

vector<Text> nodeTexts;

vector<VertexArray> lines;

vector<RectangleShape> buttons;

vector<Text> buttonTexts;

RectangleShape inputBox;

Text inputText;

Text infoText;

Text traversalText;

string inputString;

bool inputActive;

int traversalType; // 0 - нет, 1 - прямой, 2 - симметричный, 3 - обратный

vector<double> traversalPath;

size\_t traversalStep;

Clock traversalClock;

TreeNode\* foundNode;

TreeNode\* minNode;

bool horizontalView;

vector<TreeNode\*> traversalNodes;

void updateVerticalVisualization() {

nodes.clear();

nodeTexts.clear();

lines.clear();

if (!tree.getRoot()) return;

queue<pair<TreeNode\*, Vector2f>> q;

q.push({ tree.getRoot(), Vector2f(window.getSize().x / 2, 200) });

while (!q.empty()) {

auto current = q.front();

q.pop();

TreeNode\* node = current.first;

Vector2f pos = current.second;

CircleShape circle(25);

circle.setPosition(pos.x - 25, pos.y - 25);

// Подсветка узлов

if (foundNode == node) {

circle.setFillColor(Color::Green);

}

else if (minNode == node) {

circle.setFillColor(Color::Blue);

}

else {

circle.setFillColor(Color::White);

}

circle.setOutlineThickness(2);

circle.setOutlineColor(Color::Black);

nodes.push\_back(circle);

Text text;

text.setFont(font);

text.setString(to\_string(node->data).substr(0, 5));

text.setCharacterSize(18);

text.setFillColor(Color::Black);

FloatRect textRect = text.getLocalBounds();

text.setOrigin(textRect.left + textRect.width / 2.0f, textRect.top + textRect.height / 2.0f);

text.setPosition(pos.x, pos.y);

nodeTexts.push\_back(text);

float childY = pos.y + 120;

float xOffset = 200 \* pow(0.7, pos.y / 120);

if (node->left) {

Vector2f leftPos(pos.x - xOffset, childY);

VertexArray line(Lines, 2);

line[0].position = Vector2f(pos.x, pos.y + 25);

line[1].position = Vector2f(leftPos.x, leftPos.y - 25);

line[0].color = Color::Black;

line[1].color = Color::Black;

lines.push\_back(line);

q.push({ node->left, leftPos });

}

if (node->right) {

Vector2f rightPos(pos.x + xOffset, childY);

VertexArray line(Lines, 2);

line[0].position = Vector2f(pos.x, pos.y + 25);

line[1].position = Vector2f(rightPos.x, rightPos.y - 25);

line[0].color = Color::Black;

line[1].color = Color::Black;

lines.push\_back(line);

q.push({ node->right, rightPos });

}

}

}

void updateHorizontalVisualization() {

nodes.clear();

nodeTexts.clear();

lines.clear();

if (!tree.getRoot()) return;

queue<pair<TreeNode\*, Vector2f>> q;

q.push({ tree.getRoot(), Vector2f(100, window.getSize().y / 2) });

while (!q.empty()) {

auto current = q.front();

q.pop();

TreeNode\* node = current.first;

Vector2f pos = current.second;

CircleShape circle(25);

circle.setPosition(pos.x - 25, pos.y - 25);

if (foundNode == node) {

circle.setFillColor(Color::Green);

}

else if (minNode == node) {

circle.setFillColor(Color::Blue);

}

else {

circle.setFillColor(Color::White);

}

circle.setOutlineThickness(2);

circle.setOutlineColor(Color::Black);

nodes.push\_back(circle);

Text text;

text.setFont(font);

text.setString(to\_string(node->data).substr(0, 5));

text.setCharacterSize(18);

text.setFillColor(Color::Black);

FloatRect textRect = text.getLocalBounds();

text.setOrigin(textRect.left + textRect.width / 2.0f,

textRect.top + textRect.height / 2.0f);

text.setPosition(pos.x, pos.y);

nodeTexts.push\_back(text);

float childX = pos.x + 150;

float yOffset = 100 \* pow(0.7, pos.x / 150);

if (node->left) {

Vector2f leftPos(childX, pos.y - yOffset);

VertexArray line(Lines, 2);

line[0].position = Vector2f(pos.x + 25, pos.y);

line[1].position = Vector2f(leftPos.x - 25, leftPos.y);

line[0].color = Color::Black;

line[1].color = Color::Black;

lines.push\_back(line);

q.push({ node->left, leftPos });

}

if (node->right) {

Vector2f rightPos(childX, pos.y + yOffset);

VertexArray line(Lines, 2);

line[0].position = Vector2f(pos.x + 25, pos.y);

line[1].position = Vector2f(rightPos.x - 25, rightPos.y);

line[0].color = Color::Black;

line[1].color = Color::Black;

lines.push\_back(line);

q.push({ node->right, rightPos });

}

}

}

public:

TreeVisualizer(BinarySearchTree& t, RenderWindow& w) : tree(t), window(w), inputActive(false), traversalType(0), foundNode(nullptr), minNode(nullptr), horizontalView(false) {

if (!font.loadFromFile("C:\\Users\\Artem Loginov\\Desktop\\laboratorki\\arialmt.ttf")) {

cerr << "Нет шрифта\n";

}

// Создание кнопок

createButton(20, 20, 150, 40, "Insert", Color::Green);

createButton(190, 20, 150, 40, "Remove", Color::Red);

createButton(360, 20, 150, 40, "Pre-order", Color::Cyan);

createButton(530, 20, 150, 40, "In-order", Color::Magenta);

createButton(700, 20, 150, 40, "Post-order", Color::Yellow);

createButton(20, 80, 150, 40, "Find Min", Color::Blue);

createButton(190, 80, 150, 40, "Find", Color(255, 165, 0));

createButton(560, 80, 150, 40, "Change View", Color(500, 200, 200));

// Поле ввода

inputBox.setSize(Vector2f(200, 30));

inputBox.setPosition(360, 80);

inputBox.setFillColor(Color::White);

inputBox.setOutlineThickness(2);

inputBox.setOutlineColor(Color::Black);

inputText.setFont(font);

inputText.setCharacterSize(20);

inputText.setFillColor(Color::Black);

inputText.setPosition(365, 80);

// Информационное поле

infoText.setFont(font);

infoText.setCharacterSize(20);

infoText.setFillColor(Color::White);

infoText.setPosition(20, 550);

// Текст обхода

traversalText.setFont(font);

traversalText.setCharacterSize(20);

traversalText.setFillColor(Color::White);

traversalText.setPosition(20, 500);

updateVisualization();

}

void createButton(float x, float y, float width, float height, const string& label, Color color) {

RectangleShape button(Vector2f(width, height));

button.setPosition(x, y);

button.setFillColor(color);

buttons.push\_back(button);

Text text;

text.setFont(font);

text.setString(label);

text.setCharacterSize(20);

text.setFillColor(Color::Black);

FloatRect textRect = text.getLocalBounds();

text.setOrigin(textRect.left + textRect.width / 2.0f, textRect.top + textRect.height / 2.0f);

text.setPosition(x + width / 2.0f, y + height / 2.0f);

buttonTexts.push\_back(text);

}

void handleEvent(Event& event) {

if (event.type == Event::MouseButtonPressed) {

if (event.mouseButton.button == Mouse::Left) {

Vector2f mousePos(event.mouseButton.x, event.mouseButton.y);

// Проверка клика по кнопкам

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

if (buttons[i].getGlobalBounds().contains(mousePos)) {

handleButtonClick(i);

break;

}

}

// Проверка клика по полю ввода

if (inputBox.getGlobalBounds().contains(mousePos)) {

inputActive = true;

inputBox.setOutlineColor(Color::Blue);

}

else {

inputActive = false;

inputBox.setOutlineColor(Color::Black);

}

}

}

if (event.type == Event::TextEntered && inputActive) {

if (event.text.unicode == '\b' && !inputString.empty()) {

inputString.pop\_back();

}

else if (event.text.unicode == '\r') {

if (!inputString.empty()) {

double value = stod(inputString);

tree.insert(value);

inputString.clear();

updateVisualization();

}

}

else if (event.text.unicode >= '0' && event.text.unicode <= '9' event.text.unicode == '.' event.text.unicode == '-') {

inputString += static\_cast<char>(event.text.unicode);

}

inputText.setString(inputString);

}

}

void handleButtonClick(size\_t buttonIndex) {

if (buttonIndex >= buttons.size()) return;

string buttonText = buttonTexts[buttonIndex].getString();

if (buttonText == "Insert" && !inputString.empty()) {

double value = stod(inputString);

tree.insert(value);

inputString.clear();

inputText.setString(inputString);

infoText.setString("Inserted: " + to\_string(value));

}

else if (buttonText == "Remove" && !inputString.empty()) {

double value = stod(inputString);

tree.remove(value);

inputString.clear();

inputText.setString(inputString);

infoText.setString("Removed: " + to\_string(value));

}

else if (buttonText == "Find" && !inputString.empty()) {

double value = stod(inputString);

foundNode = tree.find(value);

if (foundNode) {

infoText.setString("Found: " + to\_string(value));

}

else {

infoText.setString("Not found: " + to\_string(value));

}

}

else if (buttonText == "Find Min") {

minNode = tree.findMin();

if (minNode) {

infoText.setString("Min: " + to\_string(minNode->data));

}

}

else if (buttonText == "Pre-order") {

traversalPath = tree.preOrderTraversal();

traversalType = 1;

traversalStep = 0;

traversalNodes.clear();

tree.preOrderHelperNodes(tree.getRoot(), traversalNodes);

traversalClock.restart();

infoText.setString("Pre-order traversal started");

}

else if (buttonText == "In-order") {

traversalPath = tree.inOrderTraversal();

traversalType = 2;

traversalStep = 0;

traversalNodes.clear();

tree.inOrderHelperNodes(tree.getRoot(), traversalNodes);

traversalClock.restart();

infoText.setString("In-order traversal started");

}

else if (buttonText == "Post-order") {

traversalPath = tree.postOrderTraversal();

traversalType = 3;

traversalStep = 0;

traversalNodes.clear();

tree.postOrderHelperNodes(tree.getRoot(), traversalNodes);

traversalClock.restart();

infoText.setString("Post-order traversal started");

}

else if (buttonText == "Change View") {

horizontalView = !horizontalView;

infoText.setString("View changed to " + string(horizontalView ? "horizontal" : "vertical"));

}

updateVisualization();

}

void update(float deltaTime) {

if (traversalType > 0 && traversalClock.getElapsedTime().asSeconds() > 0.5f && traversalStep < traversalPath.size()) {

ostringstream oss;

foundNode = traversalNodes[traversalStep];

if (traversalType == 1) oss << "Pre-order: ";

else if (traversalType == 2) oss << "In-order: ";

else oss << "Post-order: ";

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

oss << traversalPath[i] << " ";

}

traversalText.setString(oss.str());

traversalStep++;

traversalClock.restart();

updateVisualization();

if (traversalStep >= traversalPath.size()) {

traversalType = 0;

foundNode = nullptr;

infoText.setString("Traversal completed");

}

}

}

void updateVisualization() {

if (horizontalView) {

updateHorizontalVisualization();

}

else {

updateVerticalVisualization();

}

}

void draw() {

// Отрисовка линий

for (auto& line : lines) {

window.draw(line);

}

// Отрисовка узлов

for (auto& node : nodes) {

window.draw(node);

}

// Отрисовка текста узлов

for (auto& text : nodeTexts) {

window.draw(text);

}

// Отрисовка кнопок

for (auto& button : buttons) {

window.draw(button);

}

// Отрисовка текста кнопок

for (auto& text : buttonTexts) {

window.draw(text);

}

// Отрисовка поля ввода

window.draw(inputBox);

window.draw(inputText);

// Отрисовка информационного текста

window.draw(infoText);

// Отрисовка текста обхода

if (traversalType > 0) {

window.draw(traversalText);

}

}

};

int main() {

RenderWindow window(VideoMode(1000, 600), "Tree");

BinarySearchTree tree;

TreeVisualizer visualizer(tree, window);

Clock clock;

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

if (event.type == Event::Closed) {

window.close();

}

visualizer.handleEvent(event);

}

float deltaTime = clock.restart().asSeconds();

visualizer.update(deltaTime);

window.clear(Color(50, 50, 50));

visualizer.draw();

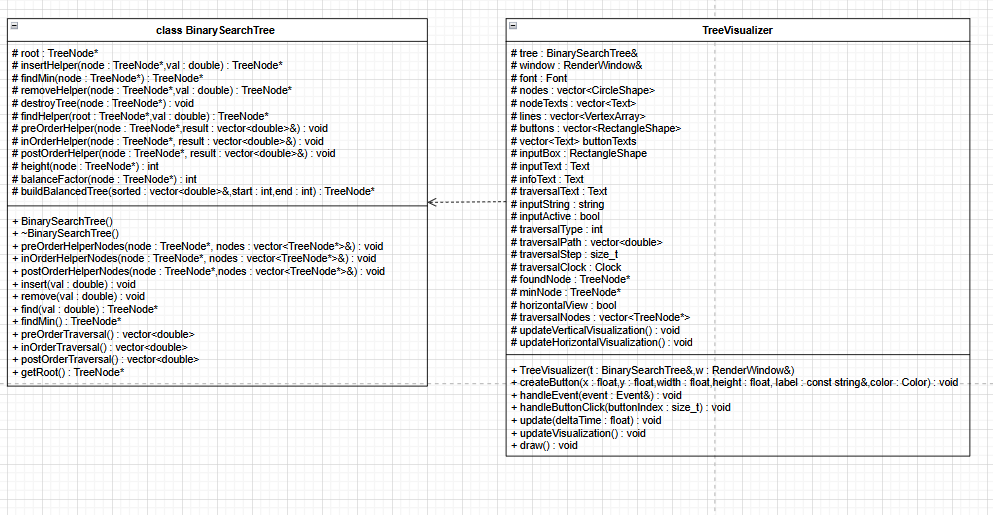
window.display();

}

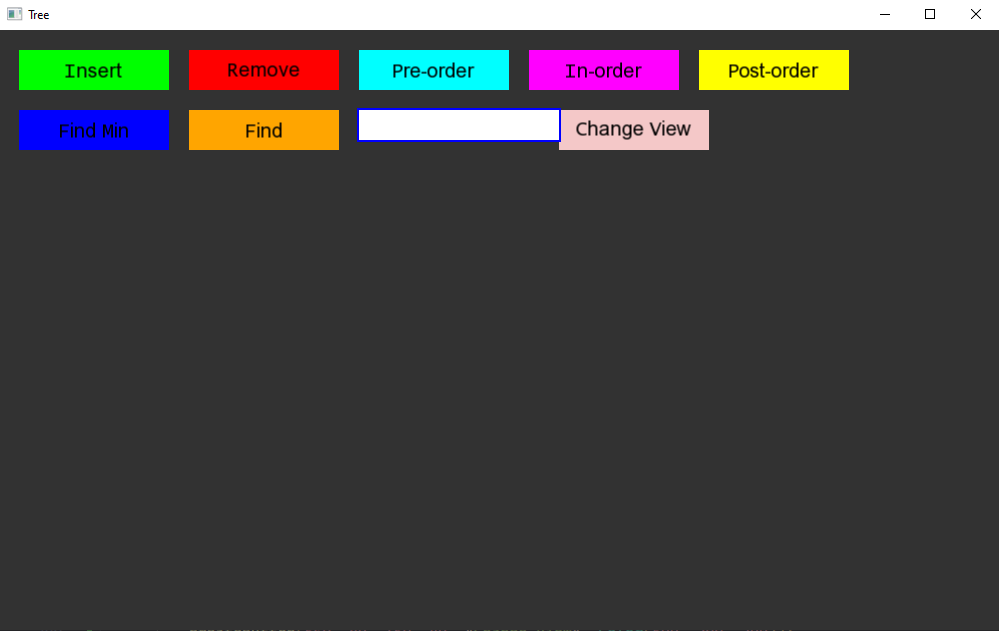
return 0;

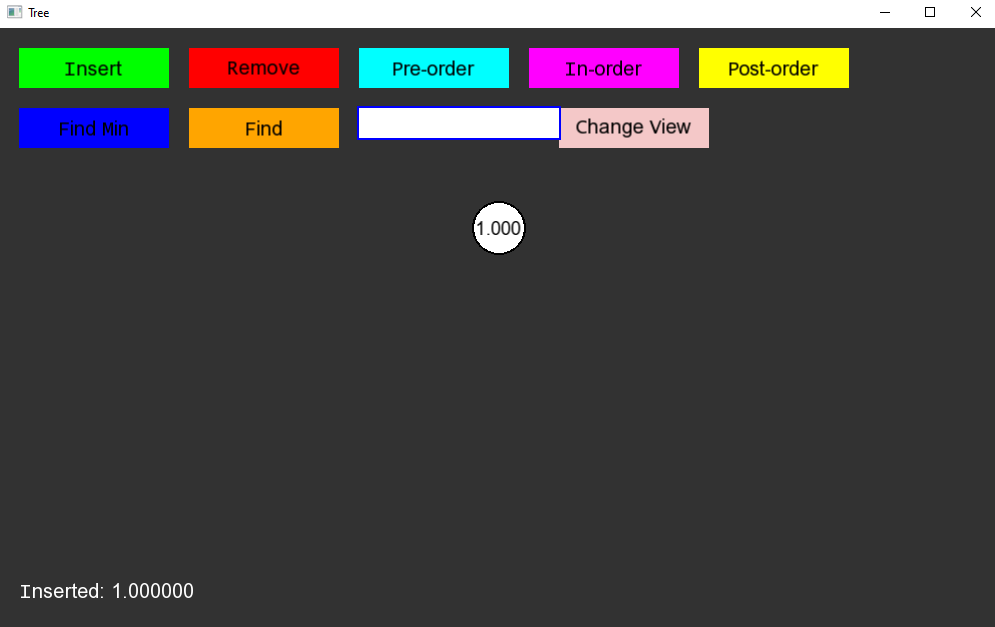
}

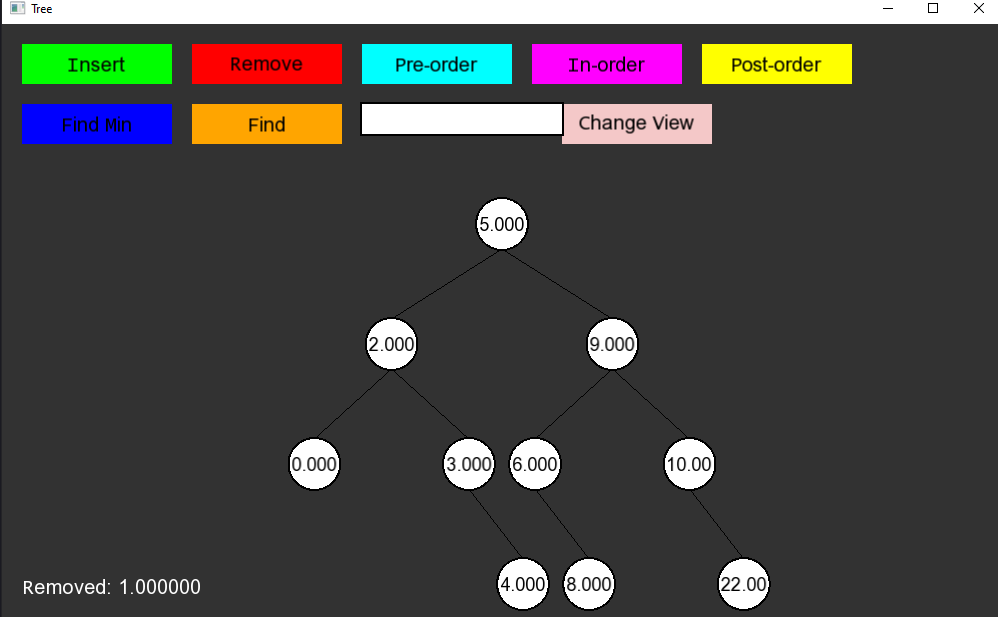
UML диаграмма:

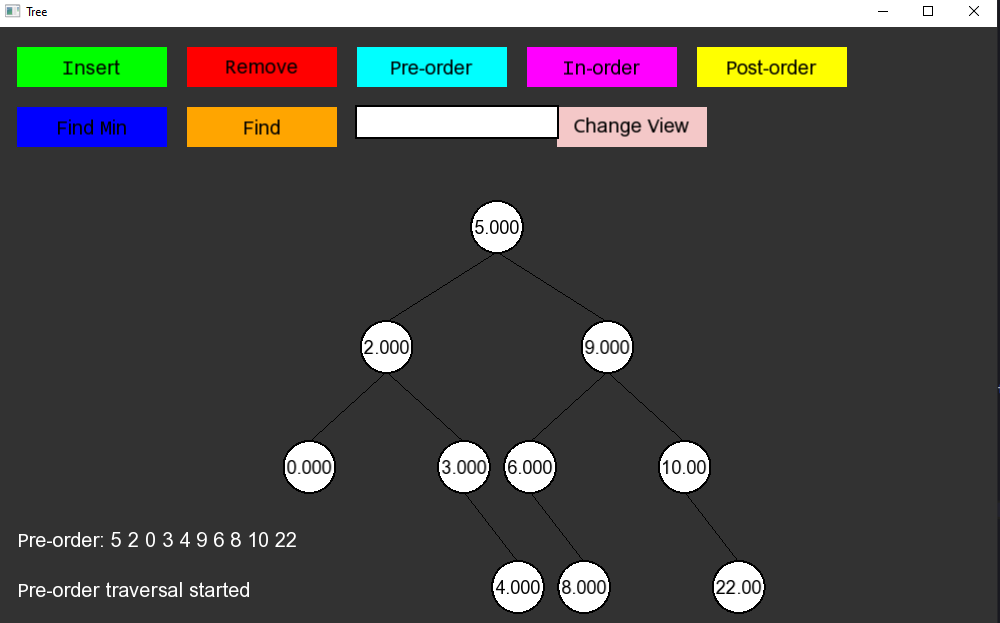


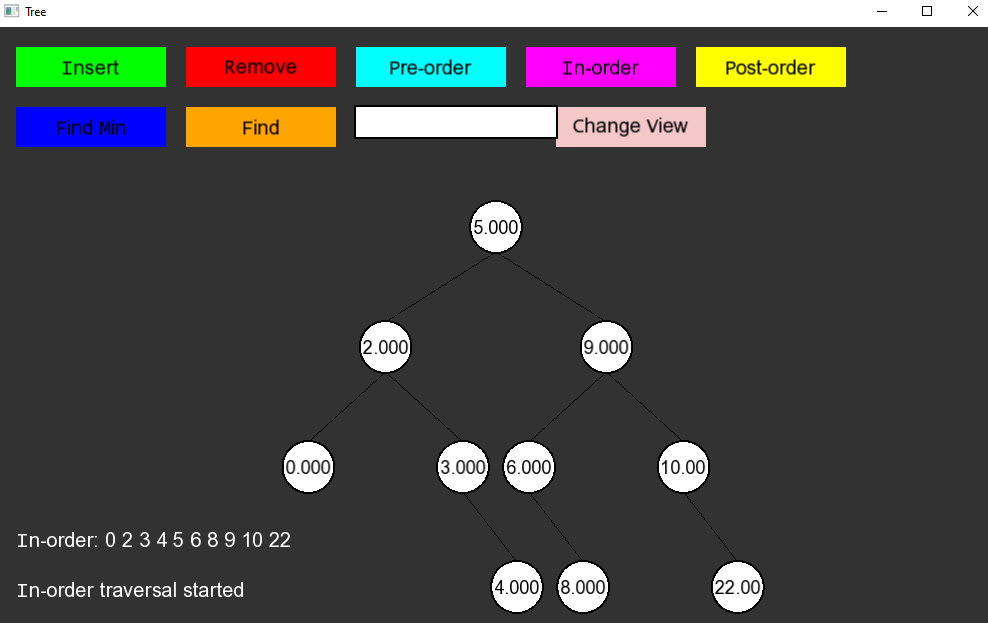
Работа программы:

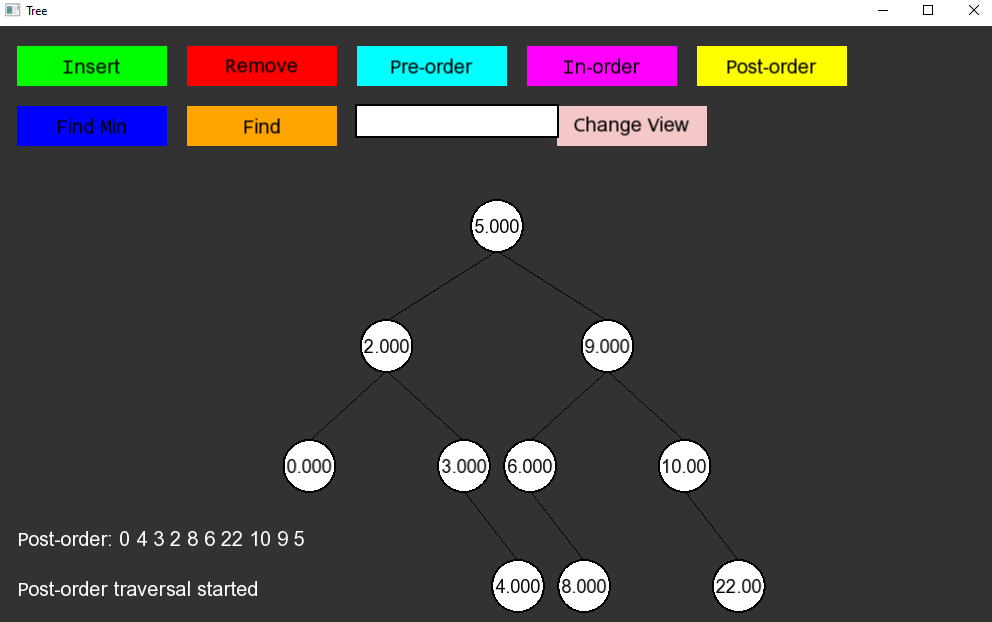


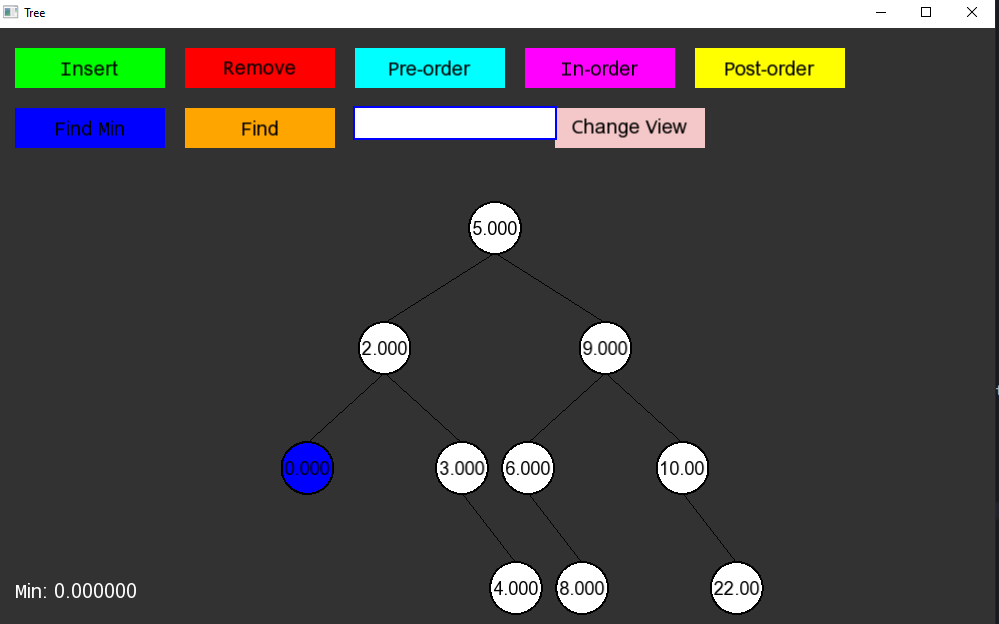


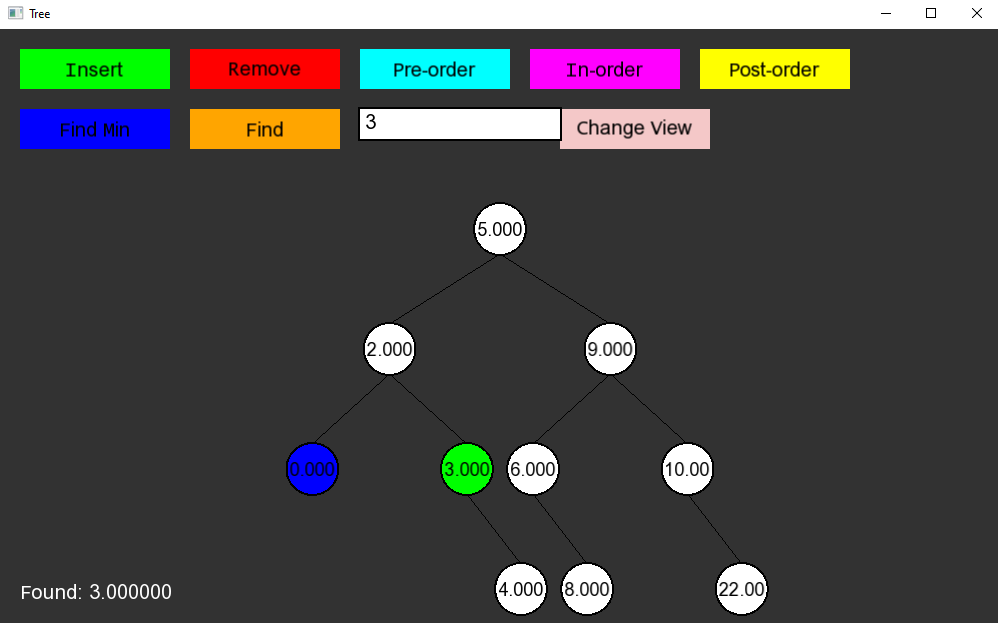


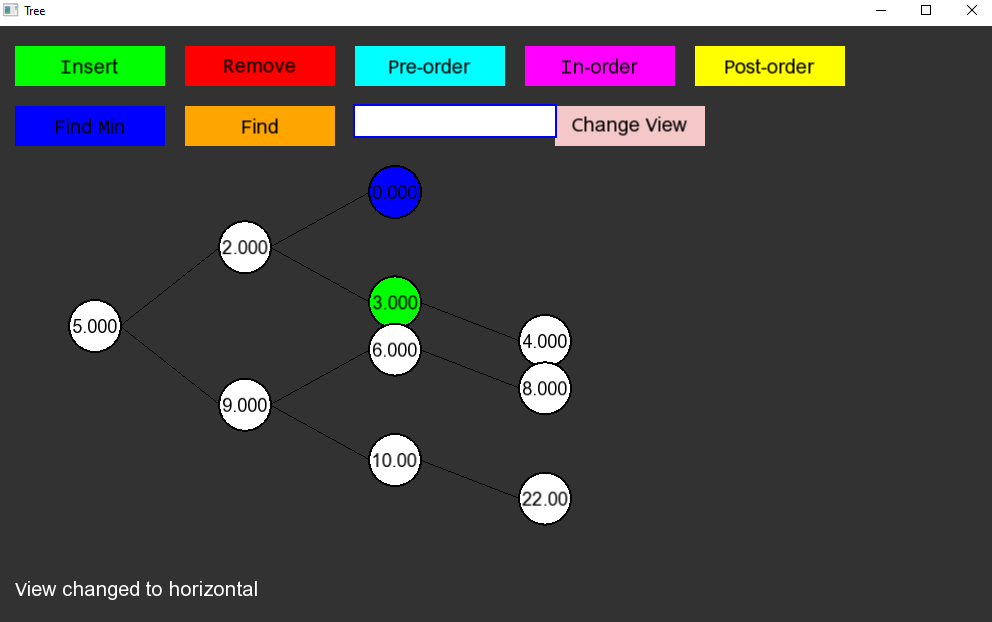












https://github.com/Prefix008/lab