Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
Пермский национальный исследовательский политехнический университет
Электротехнический факультет
Кафедра информационных технологий и автоматизированных систем

Лабораторная работа

"Бинарные деревья"

Вариант: 12

Выполнил студент ИВТ-24-2б:
Шишкин Максим Григорьевич

_____
(дата, подпись)

Проверил доцент кафедры ИТАС:

Полякова Ольга Андреевна

_____
(дата, подпись)

Пермь 2025

Содержание

1 Постановка задачи

Целью проекта является разработка графического приложения с использованием Qt, реализующего работу с бинарным деревом. В рамках работы необходимо:

Реализовать бинарное дерево поиска с типом информационного поля char.

Предусмотреть следующие операции:

- Вставка узла
- Удаление узла
- Поиск узла
- Получение высоты дерева
- Обходы дерева: прямой, симметричный и обратный
- Балансировка дерева
- Визуализировать дерево графически.
- Обеспечить удобный пользовательский интерфейс.

2 Анализ: Платформы, языки и фреймворки

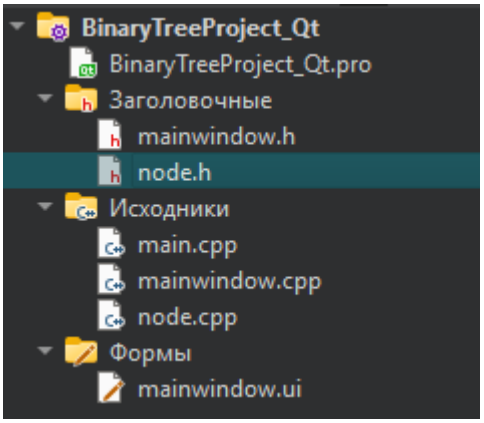
- Язык программирования: C++
- Фреймворк: Qt 6.0 (использован Qt Creator 16.0.2 Community)
- Платформа разработки: Windows 10/11
- Графическая визуализация: Использована встроенная система QGraphicsScene и QGraphicsView, без сторонних библиотек.

Qt предоставляет широкие возможности для создания кроссплатформенных GUI-приложений и позволяет эффективно реализовать как визуальную часть проекта, так и алгоритмы обработки данных.

3 Описание, реализация и структура

Файлы проекта:

1. main.cpp – запуск приложения
2. mainwindow.h / .cpp – графический интерфейс, логика кнопок, визуализация
3. node.h / .cpp – структура узла дерева и вся логика операций



Файл node.h

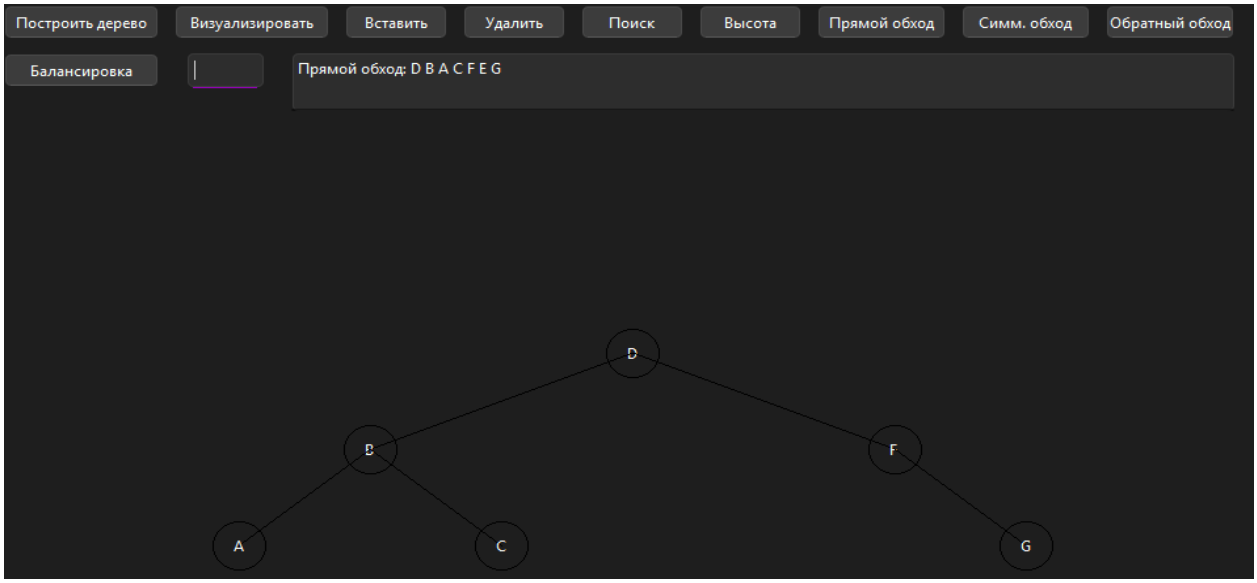
```h
#ifndef NODE_H
#define NODE_H

#include <QString>
#include <vector>

// Класс узла бинарного дерева
class Node {
public:
    char data;          // Символ, хранящийся в узле
    Node* left;         // Левый потомок
    Node* right;        // Правый потомок

    // Конструктор узла
    Node(char d);

    // Построение идеально сбалансированного дерева
    static Node* buildPerfectTree(const std::vector<char>& values, int start, int end);

    // Вставка символа в дерево поиска
    static Node* insert(Node* root, char value);

    // Удаление символа из дерева
    static Node* remove(Node* root, char value);

    // Поиск символа в дереве
    static Node* search(Node* root, char value);

    // Вычисление высоты дерева
    static int height(Node* root);

    // Прямой обход (префиксный)
    static void preorder(Node* root, QString& out);

    // Симметричный обход (инфиксный)
    static void inorder(Node* root, QString& out);

    // Обратный обход (постфиксный)
    static void postorder(Node* root, QString& out);

    // Балансировка дерева
    static void balanceTree(Node*& root);

    // Сбор значений в симметричном порядке
    static void gatherInorder(Node* root, std::vector<char>& values);

    // Построение дерева поиска из отсортированного массива
    static Node* buildBSTFromSorted(const std::vector<char>& values, int start, int end);

private:
    // Поиск минимального узла (используется при удалении)
    static Node* findMin(Node* node);
};

#endif // NODE_H
```

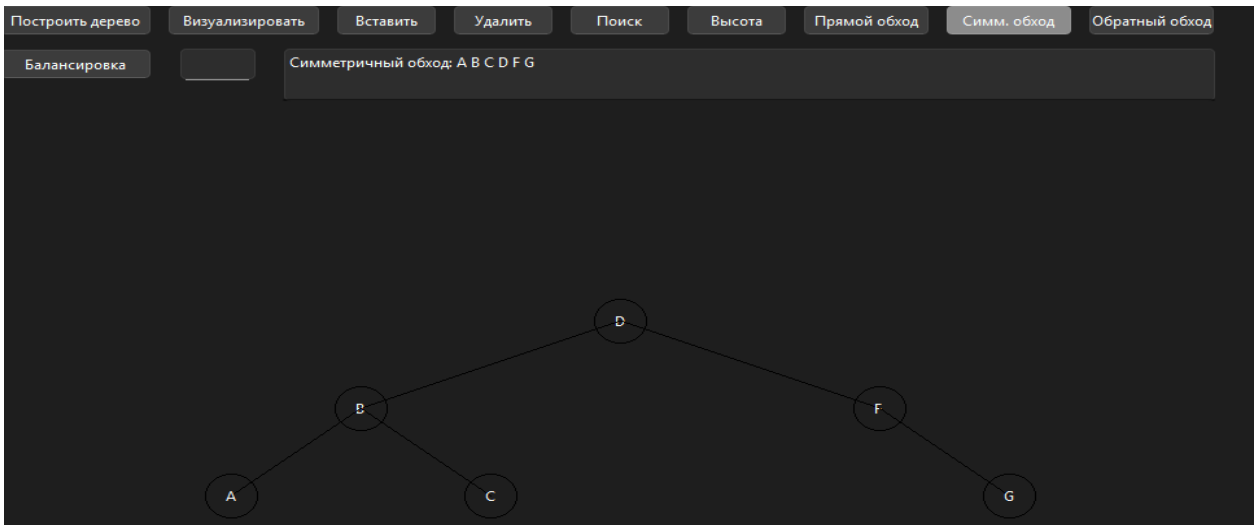
Файл node.cpp

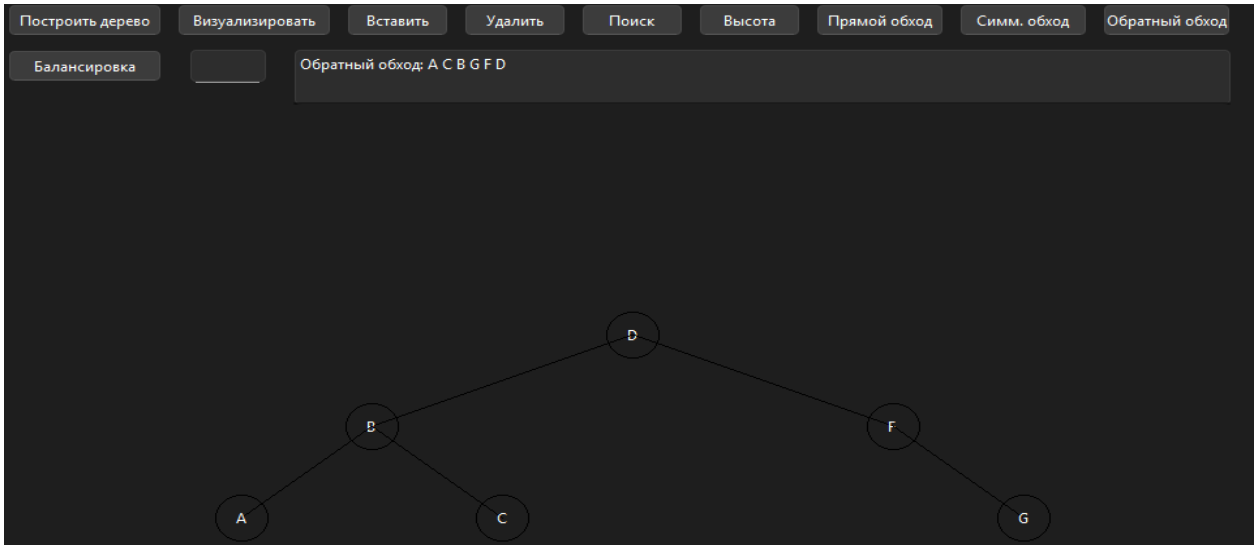
```cpp
#include "node.h"
#include <algorithm>

Node::Node(char d) : data(d), left(nullptr), right(nullptr) {}

Node* Node::buildPerfectTree(const std::vector<char>& values, int start, int end) {
    if (start > end) return nullptr;
    int mid = (start + end) / 2;
    Node* node = new Node(values[mid]);
    node->left = buildPerfectTree(values, start, mid - 1);
    node->right = buildPerfectTree(values, mid + 1, end);
    return node;
}

Node* Node::buildBSTFromSorted(const std::vector<char>& values, int start, int end) {
    return buildPerfectTree(values, start, end); // работает, если вход отсортирован
}

Node* Node::insert(Node* root, char value) {
    if (!root) return new Node(value);
    if (value == root->data) return root; // не вставляем дубликаты
    if (value < root->data)
        root->left = insert(root->left, value);
    else
        root->right = insert(root->right, value);
    return root;
}

Node* Node::findMin(Node* node) {
    while (node && node->left) node = node->left;
    return node;
}

Node* Node::remove(Node* root, char value) {
    if (!root) return nullptr;
    if (value < root->data)
        root->left = remove(root->left, value);
    else if (value > root->data)
        root->right = remove(root->right, value);
    else {
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        } else {
            Node* temp = findMin(root->right);
            root->data = temp->data;
            root->right = remove(root->right, temp->data);
        }
    }
    return root;
}

Node* Node::search(Node* root, char value) {
    if (!root || root->data == value) return root;
    if (value < root->data)
        return search(root->left, value);
    else
        return search(root->right, value);
}

int Node::height(Node* root) {
    if (!root) return 0;
    return 1 + std::max(height(root->left), height(root->right));
}

void Node::preorder(Node* root, QString& out) {
    if (!root) return;
    out += root->data;
    out += ' ';
```

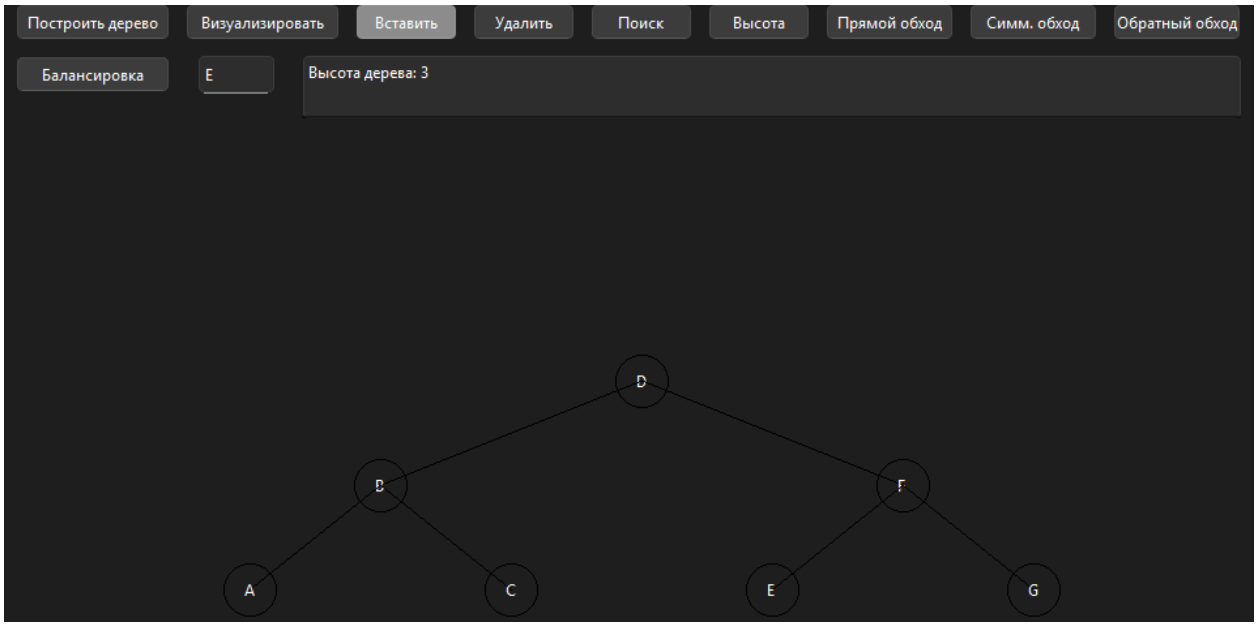
```
    preorder(root->left, out);
    preorder(root->right, out);
}

void Node::inorder(Node* root, QString& out) {
    if (!root) return;
    inorder(root->left, out);
    out += root->data;
    out += ' ';
    inorder(root->right, out);
}

void Node::postorder(Node* root, QString& out) {
    if (!root) return;
    postorder(root->left, out);
    postorder(root->right, out);
    out += root->data;
    out += ' ';
}

void Node::gatherInorder(Node* root, std::vector<char>& values) {
    if (!root) return;
    gatherInorder(root->left, values);
    values.push_back(root->data);
    gatherInorder(root->right, values);
}

void Node::balanceTree(Node*& root) {
    std::vector<char> values;
    gatherInorder(root, values);
    std::sort(values.begin(), values.end()); // сортировка обязательна
    delete root;
    root = buildBSTFromSorted(values, 0, values.size() - 1);
}
```

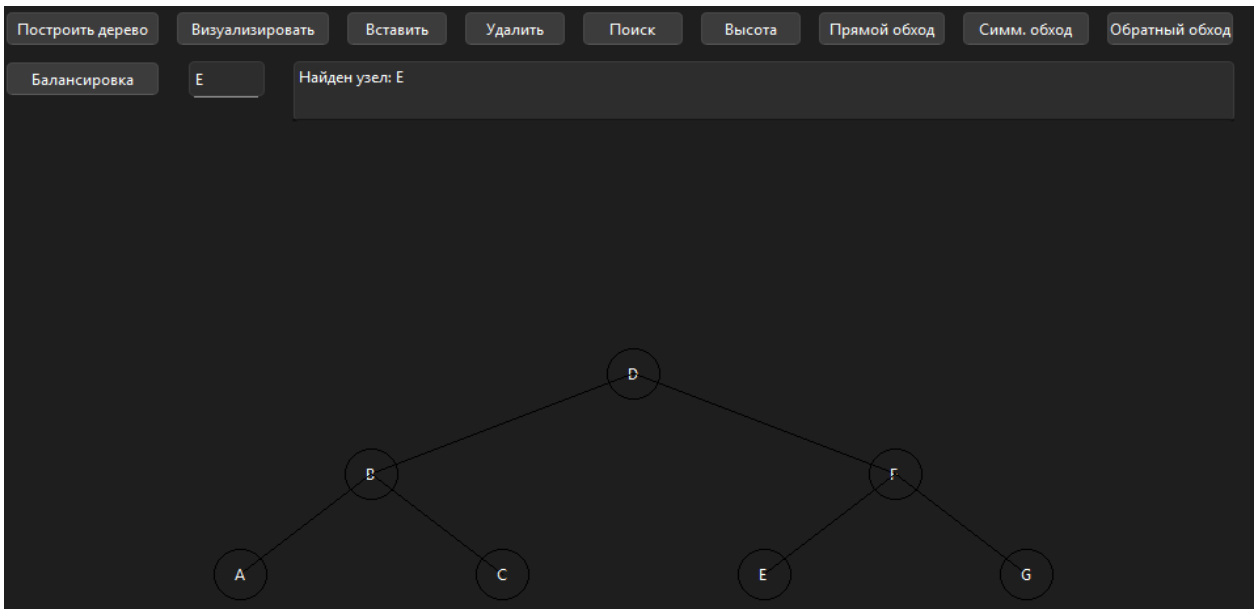
4 Результаты работы



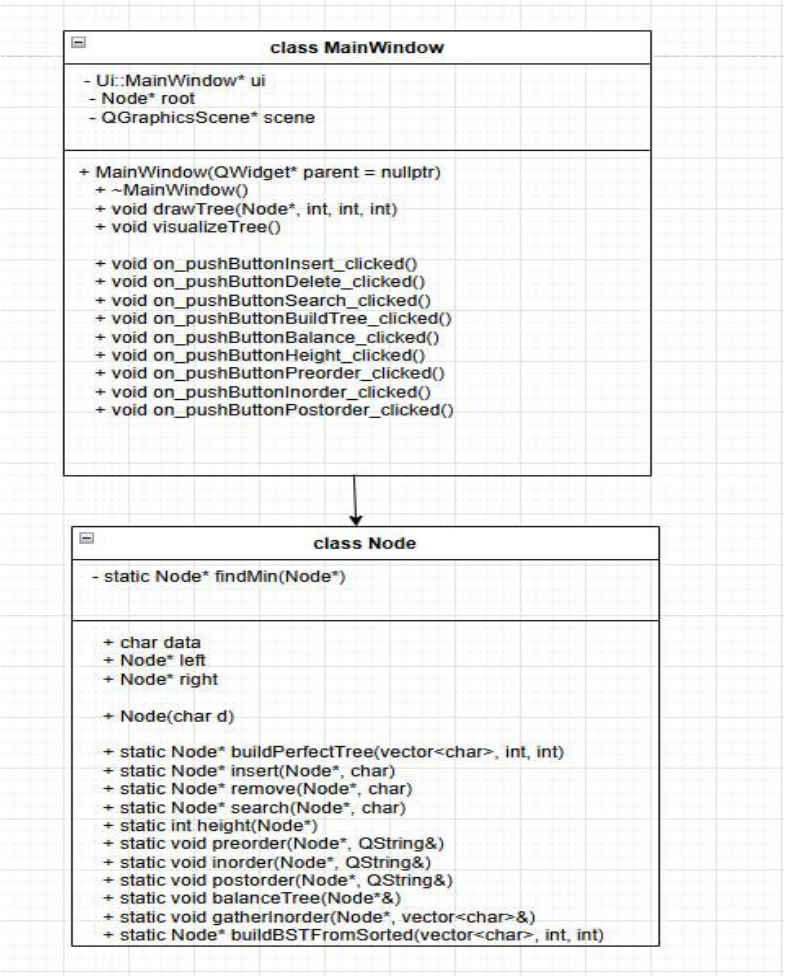

Построить дерево
Визуализировать
Вставить
Удалить
Поиск
Высота
Прямой обход
Симм. обход
Обратный обход
Балансировка
Симметричный обход: A B C D F G
D
B
F
A
C
G


Построить дерево
Визуализировать
Вставить
Удалить
Поиск
Высота
Прямой обход
Симм. обход
Обратный обход
Балансировка
Обратный обход: A C B G F D
D
B
F
A
C
G


Построить дерево
Визуализировать
Вставить
Удалить
Поиск
Высота
Прямой обход
Симм. обход
Обратный обход
Балансировка
E
Высота дерева: 3
D
B
F
A
C
E
G



5 UML-диаграмма классов



6 Ссылка на github

ссылка на github - https://github.com/MAKSPOWERO/mas1