МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИИТ)
Кафедра МОиСИТ

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №7.1

«Балансировка дерева поиска»
по дисциплине
«Структуры и алгоритмы обработки данных»

Выполнил студент группы ИКБО-10-24 Таганов А.А.

Москва 2025

Цель работы: составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту.

Тип двоичного дерева: Красно-чёрное дерево

Реализуемые алгоритмы: Вставка, удаление элементов и балансировка, прямой обход, симметричный обход, нахождение длины пути от корня до заданного значения, нахождение высоты дерева.

Описание решения: Красно-чёрное дерево (англ. red-black tree) — двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" (англ. red) и "чёрный" (англ. black). Для поддержания баланса после операций вставки или удаления при нарушении правил построения такого дерева, производится балансировка. При вставке может нарушиться черная высота дерева в нескольких случаях:

1. Корень красный;
2. Родитель и сын красные:
    a. Дядя красный;
    b. Дядя черный, ребенок левый;
    c. Дядя черный, ребенок правый.

Решение: в качестве звеньев деревьев будет использована структура Node, код которой представлен на листинге 1. Также на листинге представлено перечисление Color, использующееся в структуре Node для представления цвета узла.

Листинг 1 – Структура Node

```
enum Color { RED, BLACK };

struct Node {
    string city;
    Color color;
    Node *left, *right, *parent;
    Node(string city)
        :    city(city),    color(RED),    left(nullptr),    right(nullptr),
parent(nullptr) {}
};
```

Для балансировки дерева были созданы функции левого и правого поворота поддерева (листинг 2).

Листинг 2 – Функции поворота

```
void rotateLeft(Node* x) {
        Node* y = x->right;
        x->right = y->left;
        if (y->left) y->left->parent = x;

        y->parent = x->parent;
        if (!x->parent)
            root = y;
        else if (x == x->parent->left)
            x->parent->left = y;
        else
            x->parent->right = y;

        y->left = x;
        x->parent = y;
    }

    void rotateRight(Node* y) {
        Node* x = y->left;
        y->left = x->right;
        if (x->right) x->right->parent = y;

        x->parent = y->parent;
        if (!y->parent)
            root = x;
        else if (y == y->parent->left)
            y->parent->left = x;
        else
            y->parent->right = x;

        x->right = y;
        y->parent = x;
    }
```

Сама функция балансировки использует левый, правый повороты и их сочетания в зависимости от возникшего случая, нарушающего баланс дерева. Сами случаи кратко закомментированы в коде.

Листинг 3 – Функция балансировки при вставке

```
    void insertFixup(Node* z) {
        while (z->parent->color == RED) {
            if (z->parent == z->parent->parent->left) {
                Node* y = z->parent->parent->right; // y дядя
                if (y->color == RED) {                // дядя красный
                    z->parent->color = BLACK;
                    y->color = BLACK;
                    z->parent->parent->color = RED;
                    z = z->parent->parent;
                } else {
                    if (z == z->parent->right) {     // дядя черный, z внешний
                        z = z->parent;
                        rotateLeft(z);
                    }
```

```
                    // дядя черный, z внутренний
                    z->parent->color = BLACK;
                    z->parent->parent->color = RED;
                    rotateRight(z->parent->parent);
                }
            } else { // Зеркально!
                Node* y = z->parent->parent->left;   // дядя
                if (y->color == RED) {               // дядя красный
                    z->parent->color = BLACK;
                    y->color = BLACK;
                    z->parent->parent->color = RED;
                    z = z->parent->parent;
                } else {
                    if (z == z->parent->left) {      // дядя черный, ребенок
внутренний
                        z = z->parent;
                        rotateRight(z);
                    }
                    // дядя черный, ребенок внешний
                    z->parent->color = BLACK;
                    z->parent->parent->color = RED;
                    rotateLeft(z->parent->parent);
                }
            }
        }
        root->color = BLACK;
    }
```

В красно-чёрном дереве при удалении узла могут возникнуть 4 разных ситуации, требующих балансировки. Функция балансировки представлена в листинге 4, все случаи требующие балансировки закомментированы в коде.

Листинг 4 – Функция балансировки при удалении

```
    void deleteFixup(Node* x) {
        while (x != root && x->color == BLACK) {
            if (x == x->parent->left) {
                Node* w = x->parent->right;          // брат
                if (w->color == RED) {               // брат красный
                    w->color = BLACK;
                    x->parent->color = RED;          // перекрашиваем узел и
брата
                    rotateLeft(x->parent);
                    w = x->parent->right;
                }
                if (w->left->color == BLACK && w->right->color == BLACK) { //
брат черный, оба ребенка черные
                    w->color = RED;
                    x = x->parent;
                } else {
                    if (w->right->color == BLACK) {  // брат черный, дальний
ребенок черный
                        w->left->color = BLACK;
                        w->color = RED;
                        rotateRight(w);
                        w = x->parent->right;
                    }
                    // брат черный, дальний ребенок красный
                    w->color = x->parent->color;
                    x->parent->color = BLACK;
                    w->right->color = BLACK;
```

```
                    rotateLeft(x->parent);
                    x = root;
                }
            } else {
                Node* w = x->parent->left;
                if (w->color == RED) {                // Case 1
                    w->color = BLACK;
                    x->parent->color = RED;
                    rotateRight(x->parent);
                    w = x->parent->left;
                }
                if (w->right->color == BLACK && w->left->color == BLACK) { //
Case 2
                    w->color = RED;
                    x = x->parent;
                } else {
                    if (w->left->color == BLACK) {    // Case 3
                        w->right->color = BLACK;
                        w->color = RED;
                        rotateLeft(w);
                        w = x->parent->left;
                    }
                    // Case 4
                    w->color = x->parent->color;
                    x->parent->color = BLACK;
                    w->left->color = BLACK;
                    rotateRight(x->parent);
                    x = root;
                }
            }
        }
        x->color = BLACK;
    }
```

После реализации функций балансировки были реализованы функции удаления и вставки узла. Функция удаления была разбита на две функции для улучшения читаемости кода.

Листинг 5 – Функция вставки

```
    void insert(const string& city) {
        Node* z = new Node(city, RED);
        z->left = z->right = z->parent = NIL;
        Node* y = NIL;
        Node* x = root;
        while (x != NIL) {
            y = x;
            if (z->city < x->city)      x = x->left;
            else if (z->city > x->city) x = x->right;
            else {
                cout << "Node with such value already exists.\n";
                delete z;
                return;
            }
        }
        z->parent = y;
        if (y == NIL)          root = z;
        else if (z->city < y->city) y->left  = z;
        else                        y->right = z;
        insertFixup(z);
    }
```

Листинг 6 – Приватная функция удаления

```
    void erase(Node* z) {
        Node* y = z;
        Color yOriginal = y->color;
        Node* x;

        if (z->left == NIL) {
            x = z->right;
            transplant(z, z->right);
        } else if (z->right == NIL) {
            x = z->left;
            transplant(z, z->left);
        } else {
            y = minimum(z->right, NIL);
            yOriginal = y->color;
            x = y->right;
            if (y->parent == z) {
                x->parent = y;
            } else {
                transplant(y, y->right);
                y->right = z->right;
                y->right->parent = y;
            }
            transplant(z, y);
            y->left = z->left;
            y->left->parent = y;
            y->color = z->color;
        }

        delete z;

        if (yOriginal == BLACK) deleteFixup(x);
    }
```

Листинг 7 – Публичная функция удаления

```
    bool remove(const string& city) {
        Node* z = findNode(city);
        if (z == NIL) return false;
        erase(z);
        return true;
    }
```

В соответствии с заданием были реализованы прямой обход, симметричный обход и функции для поиска длины от корня до заданного значения и вычисления высоты дерева.

Листинг 8– Остальные функции

```
    void preorder(Node* node) const {
        if (node == NIL) return;
        cout << node->city << " (" << (node->color == RED ? "R" : "B") << ")
";
        preorder(node->left);
        preorder(node->right);
    }

    void inorder(Node* node) const {
```

```
        if (node == NIL) return;
        inorder(node->left);
        cout << node->city << " (" << (node->color == RED ? "R" : "B") << ")
";
        inorder(node->right);
    }

    int height(Node* node) const {
        if (node == NIL) return 0;
        return 1 + max(height(node->left), height(node->right));
    }

    int pathLength(Node* node, const string& city, int depth = 0) const {
        if (node == NIL) return -1;
        if (city == node->city) return depth;
        else if (city < node->city) return pathLength(node->left, city, depth
+ 1);
        else return pathLength(node->right, city, depth + 1);
    }
```

Также было реализовано пользовательское меню для выбора режимов.

Листинг 9 – Пользовательское меню

```
void menu() {
    RedBlackTree tree;
    int choice;
    string city;

    do {
        cout << "\nRed-Black Tree Menu (with NIL sentinel)\n";
        cout << "1. Insert city\n";
        cout << "2. Preorder traversal\n";
        cout << "3. Inorder traversal\n";
        cout << "4. Find path length to city\n";
        cout << "5. Find height of the tree\n";
        cout << "6. Delete city\n";
        cout << "0. Exit\n";
        cout << "Enter your choice: ";
        if (!(cin >> choice)) return;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');

        switch (choice) {
            case 1:
                cout << "Enter city name: ";
                getline(cin, city);
                tree.insert(city);
                break;
            case 2:
                tree.showPreorder();
                break;
            case 3:
                tree.showInorder();
                break;
            case 4:
                cout << "Enter city name: ";
                getline(cin, city);
                tree.showPathLength(city);
                break;
            case 5:
                tree.showHeight();
                break;
            case 6:
```

```
                cout << "Enter city name: ";
                getline(cin, city);
                if (tree.remove(city)) cout << "Deleted.\n";
                else cout << "There's no such city.\n";
                break;
            case 0:
                cout << "Exiting…\n";
                break;
            default:
                cout << "Invalid choice.\n";
        }
    } while (choice != 0);
}
```

Было проведено тестирование программы на дереве из 10 элементов. Элементы добавлялись в следующем порядке A, B, C, D, E, F, G, H, I, J. Результаты тестирования представлены на рисунках 1-5.

```
Red-Black Tree Menu (with NIL sentinel)
1. Insert city
2. Preorder traversal
3. Inorder traversal
4. Find path length to city
5. Find height of the tree
6. Delete city
0. Exit
Enter your choice: 2
Preorder: D (B) B (B) A (B) C (B) F (B) E (B) H (R) G (B) I
 (B) J (R)
```

Рис.1 – Прямой обход

```
Red-Black Tree Menu (with NIL sentinel)
1. Insert city
2. Preorder traversal
3. Inorder traversal
4. Find path length to city
5. Find height of the tree
6. Delete city
0. Exit
Enter your choice: 3
Inorder: A (B) B (B) C (B) D (B) E (B) F (B) G (B) H (R) I
(B) J (R)
```

Рис.2 – Симметричный обход

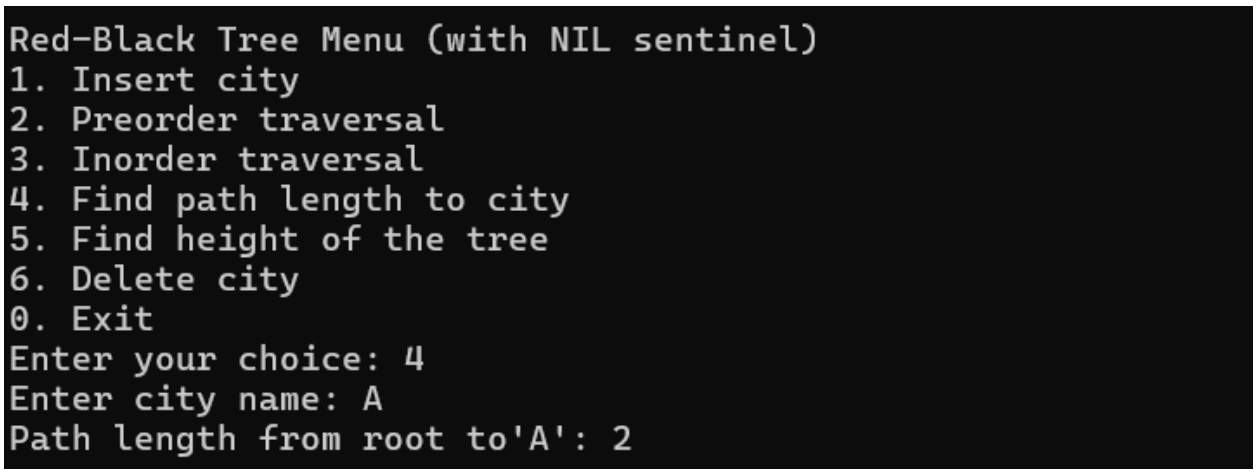

Рис.3 – Расстояние до узла

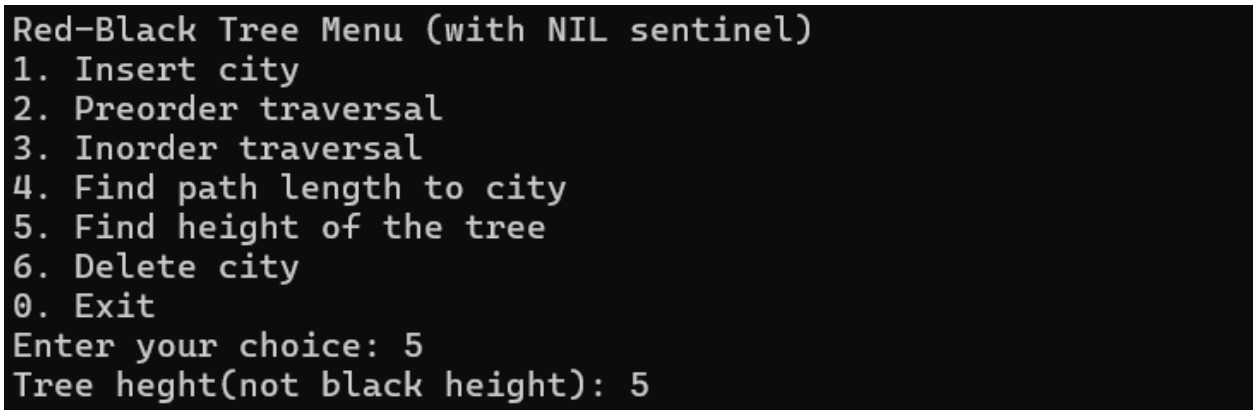

Рис.4 – Высота

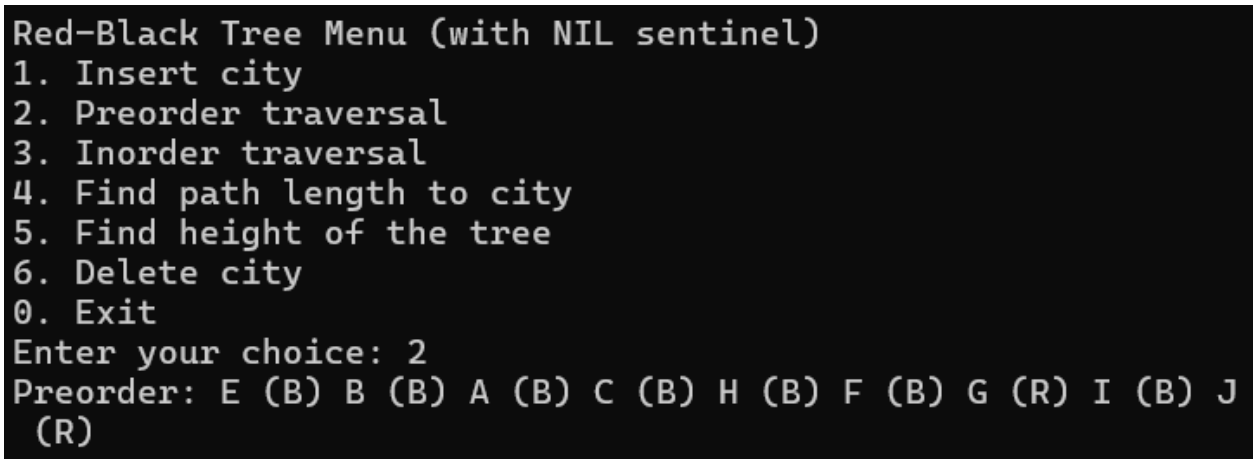

Рис.5 – Прямой обход после удаления текущего корня

Вывод

Поставленные задачи выполнены: изучены и реализованы алгоритмы по работе с красно-чёрным деревом, алгоритмы симметричного и прямого обхода.

Литература.

1. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

2. Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 03.11.2025).

3. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс].URL:https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 03.11.2025).