МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Інститут **ІКНІ**

Кафедра **ПЗ**

ЗВІТ

До лабораторної роботи №9

На тему: «Нелінійні структури даних: червоно-чорні дерева»

З дисципліни: «Алгоритми та структури даних»

Лектор : доцент каф.ПЗ

Коротєєва Т.О.

Виконала: ст.гр.ПЗ-23

Кохман О.В.

Прийняв: асистент каф.ПЗ

Франко А.В.

«____»_____2022 р.

∑____ _ _ _ _ _ _ _ _ .

Львів – 2022

Тема: нелінійні структури даних: червоно-чорні дерева

Мета: ознайомитися з червоно-чорними деревами та отримати навички програмування алгоритмів, що їх обробляють.

Теоретичні відомості

Дерева як засіб реалізації словників ефективні, якщо їх висота мала, але мала висота не гарантується, і в гіршому випадку дерева не більш ефективні, ніж списки. Червоно-чорні дерева – це один з типів збалансованих дерев пошуку, в яких передбачені операції балансування гарантують, що висота дерева не перевищить O(log N).

Червоно-чорне дерево (*red-black tree*) – це двійкове дерево пошуку, вершини якого розділені на червоні (*red*) і чорні (*black*). Таким чином, кожна вершина зберігає один додатковий біт – її колір.

При цьому повинні виконуватися певні вимоги, які гарантують, що глибина будь-яких двох листків дерева відрізняється не більше, ніж у два рази, тому дерево можна назвати збалансованим (*balanced*).

Кожна вершина червоно-чорного дерева має поля *color* (колір), *key* (ключ), *left* (лівий нащадок), *right* (правий нащадок) і *p* (предок). Якщо у вершини відсутній нащадок або предок, відповідне поле містить значення *nil*. Для зручності ми будемо вважати, що значення *nil*, які зберігаються в полях *left* і *right*, є посиланнями на додаткові (фіктивні) листки дерева. При такому заповненні дерева кожна вершина, що містить ключ, має двох нащадків.

Двійкове дерево пошуку називається червоно-чорним деревом, якщо воно має такі властивості (будемо називати їх RB-властивостями, *red-black properties*):

1) кожна вершина або червона, або чорна;

2) Правило червоного: Якщо елемент червоний, його батько повинен бути чорним.

3) Правило шляху: Кількість чорних елементів повинна бути однаковою для всіх шляхів від кореневого елемента до елемента, що не має дочірніх або має один дочірній елемент.

Індивідуальне завдання

1) читає з клавіатури ключі N, M (цілі, дійсні або символи залежно від варіанту завдання);

2) програма зберігає першу послідовність до червоно-чорного дерева;

3) кожного разу, коли до дерева додається новий елемент, потрібно вивести статистику (згідно з варіантом завдання);

4) після побудови дерева для кожного елемента другої послідовності M потрібно вивести результати наступних операцій над деревом:

1. Чи є елемент у дереві та його колір?

2. Нащадок (нащадки) та його (їх) колір.

3. Батько та його колір.

Використовувати готові реалізації структур даних (наприклад, STL) **заборонено**.

Варіант 9: N, M – символи; голосний елемент та його колір; нащадки та їх колір.

Код програми

Назва файлу: main.cpp

```
#include <iostream>
using namespace std;

struct node {
    char data{};
    node* left = nullptr;
    node* right = nullptr;
    node* parent = nullptr;
    string color;
};
char vowels[] = { 'a', 'e', 'i', 'o', 'u' };
class RedBlackTree {

    node* root;

public:
    RedBlackTree() : root(nullptr) {}

    node* GetRoot() { return root; }

    void InsertNode(char stuff) {
        if (root == nullptr) {
            root = new node();
```

```
        root->data = stuff;
        root->parent = nullptr;
        root->color = "BLACK";
        cout << "Element inserted.\n";
    }
    else {
        auto linker = GetRoot();
        node* newnode = new node();
        newnode->data = stuff;

        while (linker != nullptr) {
            if (linker->data > stuff) {
                if (linker->left == nullptr) {
                    linker->left = newnode;
                    newnode->color = "RED";
                    newnode->parent = linker;
                    cout << "Element inserted.\n"; break;
                }
                else { linker = linker->left; }
            }
            else {
                if (linker->right == nullptr) {
                    linker->right = newnode;
                    newnode->color = "RED";
                    newnode->parent = linker;
                    cout << "Element inserted.\n"; break;
                }
                else { linker = linker->right; }
            }
        }
        RB_Insert_Fixup(newnode);
    }
}

void RB_Insert_Fixup(node* z) {
    while (z->parent->color == "RED") {
        auto grandparent = z->parent->parent;
        auto uncle = GetRoot();
        if (z->parent == grandparent->left) {
            if (grandparent->right) { uncle = grandparent->right; }
            if (uncle->color == "RED") {
                z->parent->color = "BLACK";
                uncle->color = "BLACK";
                grandparent->color = "RED";
                if (grandparent->data != root->data) { z = grandparent; }
                else { break; }
            }
            else if (z == grandparent->left->right) {
                LeftRotate(z->parent);
            }
            else {
                z->parent->color = "BLACK";
                grandparent->color = "RED";
                RightRotate(grandparent);
                if (grandparent->data != root->data) { z = grandparent; }
                else { break; }
            }
        }
        else {
            if (grandparent->left) { uncle = grandparent->left; }
            if (uncle->color == "RED") {
                z->parent->color = "BLACK";
                uncle->color = "BLACK";
```

```
                grandparent->color = "RED";
                if (grandparent->data != root->data) { z = grandparent; }
                else { break; }
            }
            else if (z == grandparent->right->left) {
                RightRotate(z->parent);
            }
            else {
                z->parent->color = "BLACK";
                grandparent->color = "RED";
                LeftRotate(grandparent);
                if (grandparent->data != root->data) { z = grandparent; }
                else { break; }
            }
        }
    }
    root->color = "BLACK";
}

node* TreeSearch(char stuff) {
    auto temp = GetRoot();
    if (temp == nullptr) { return nullptr; }

    while (temp) {
        if (stuff == temp->data) { return temp; }
        else if (stuff < temp->data) { temp = temp->left; }
        else { temp = temp->right; }
    }
    return nullptr;
}

void LeftRotate(node* x) {
    node* nw_node = new node();
    if (x->right->left) { nw_node->right = x->right->left; }
    nw_node->left = x->left;
    nw_node->data = x->data;
    nw_node->color = x->color;
    x->data = x->right->data;

    x->left = nw_node;
    if (nw_node->left) { nw_node->left->parent = nw_node; }
    if (nw_node->right) { nw_node->right->parent = nw_node; }
    nw_node->parent = x;

    if (x->right->right) { x->right = x->right->right; }
    else { x->right = nullptr; }

    if (x->right) { x->right->parent = x; }
}

void RightRotate(node* x) {
    node* nw_node = new node();
    if (x->left->right) { nw_node->left = x->left->right; }
    nw_node->right = x->right;
    nw_node->data = x->data;
    nw_node->color = x->color;

    x->data = x->left->data;
    x->color = x->left->color;

    x->right = nw_node;
    if (nw_node->left) { nw_node->left->parent = nw_node; }
    if (nw_node->right) { nw_node->right->parent = nw_node; }
```

```
        nw_node->parent = x;

        if (x->left->left) { x->left = x->left->left; }
        else { x->left = nullptr; }

        if (x->left) { x->left->parent = x; }
    }

    void PreorderTraversal(node* temp) {
        if (!temp) { return; }
        cout << "--> " << temp->data << "<" << temp->color << ">";
        PreorderTraversal(temp->left);
        PreorderTraversal(temp->right);
    }

    void PostorderTraversal(node* temp) {
        if (!temp) { return; }
        PostorderTraversal(temp->left);
        PostorderTraversal(temp->right);
        cout << "--> " << temp->data << "<" << temp->color << ">";
    }

    node* findVowel(node* temp) {
        if (!temp) { return nullptr; }
        for (int i = 0; i < 5; ++i) {
            if (vowels[i] == temp->data) { return temp; }
        }
        auto result = findVowel(temp->left);
        if (result) { return result; }
        result = findVowel(temp->right);
        if (result) { return result; }
    }
};

void menu() {
    cout << "\n_____________________________________";
    cout << "\n\n  *****Working with Red-Black-Tree******";
    cout << "\n_____________________________________";
    cout << "\n\n1. Enter N key element";
    cout << "\n2. Enter M key element";
    cout << "\n3. Exit.";
    cout << "\n_____________________________________";
    cout << "\nPlease, select option -- ";
}


int main() {
    RedBlackTree demo;
    int option;
    char input;
    menu();
    cin >> option;
    while (option != 3) {
        switch (option) {
        case 1: {
            cout << "\nElement to be inserted -- ";
            cin >> input; demo.InsertNode(input);
            auto result = demo.findVowel(demo.GetRoot());
            if(result) {
                cout << "Vowel character: " << result->data << ". Color: " <<
result->color;
                if (result->left) { cout << "\nLeft node: " << result->left->data
<< ". Left node color: " << result->left->color; }
```

```
                else { cout << "\nLeft node was not found"; }
                if (result->right) { cout << "\nRight node: " << result->right-
>data << ". Right node color: " << result->right->color; }
                else { cout << "\nRight node was not found"; }
            }
            else {
                cout << "Vowel character was not found";
            }
            break;
        }
        case 2: {
            cout << "\nElement to be searched -- ";
            cin >> input;
            auto result = demo.TreeSearch(input);
            if (result) {
                cout << "\nElement found: " << result->data << " Color: " <<
result->color;
                if (result->left) { cout << "\nLeft node: " << result->left->data
<< ". Left node color: " << result->left->color; }
                else { cout << "\nLeft node was not found"; }
                if (result->right) { cout << "\nRight node: " << result->right-
>data << ". Right node color: " << result->right->color; }
                else { cout << "\nRight node was not found"; }
                if (result->parent) { cout << "\nParent node: " << result-
>parent->data << ". Parent node color: " << result->parent->color; }
                else { cout << "\nParent node was not found"; }
            }
            else { cout << "\nElement was not found"; }
            break;
        }

        default: cout << "Wrong Choice.\n";
        }
        cout << "\nAnything Else?";
        cin >> option;
    }
    cout << "\nTerminating.... ";
    return 0;
}
```

Протокол роботи

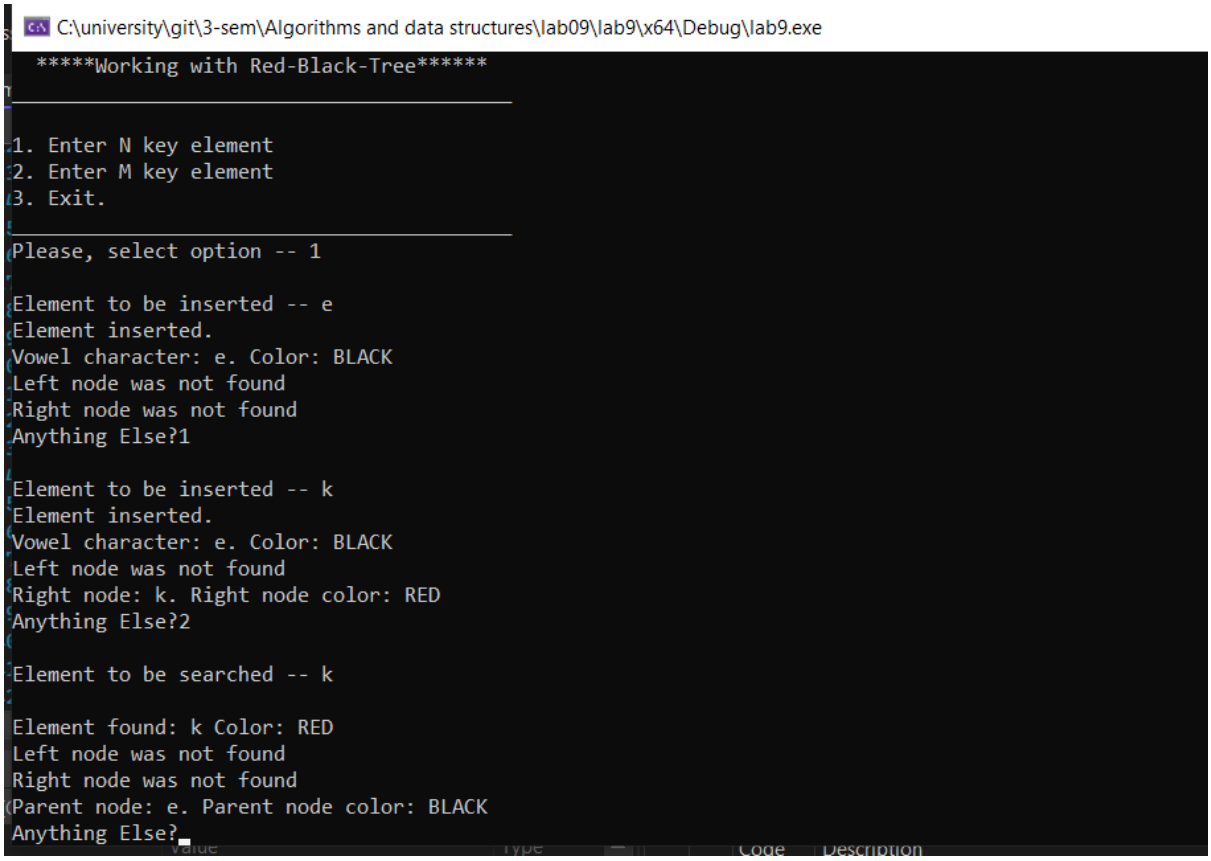

Рис. 1 Результат роботи програми.

Висновок

На цій лабораторній роботі я дізналась про нелінійні структури даних, а саме про червоно-чорні дерева, реалізувала програму за допомогою червоно-чорних дерев та визначила складність алгоритму.