МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»
Кафедра програмної інженерії та інформаційних технологій управління

Звіт з лабораторної роботи № 3
з дисципліни «Основи теорії алгоритмів»

Виконав:
ст. гр. КН-221в
Шулюпов Є.Р.
Перевірила:
доцент каф. ПІТУ
Солонська С.В.

Харків
2022

ТЕМА: БАЗОВІ СТРУКТУРИ ДАНИХ. ЧЕРВОНО-ЧОРНІ ДЕРЕВА

ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Розробити програму, яка читає з клавіатури цілі числа N, M (1 < N, M < 256), N пар <ключ, значення> (ключ — ціле, дійсне число або рядок в залежності від варіанту завдання; значення — рядок; усі рядки до 255 символів), жодний з яких не повторюється та ще M ключів. Всі рядки розділяються пробілом або новим рядком. Програма зберігає пар рядків до хеш-таблиці та видає на екран значення, що відповідають переліченим ключам.

Використати ключем ціле число, та застосувати тривіальне хешування.

МЕТА РОБОТИ

Ознайомлення з червоно-чорними деревами та отримати навички програмування алгоритмів, що їх обробляють. Розробити програму, яка читає з клавіатури числа N, M (1 < N, M < 256); послідовність N ключів (цілих, дійсних чисел або рядків (до 255 символів) в залежності від варіанту завдання); послідовність M ключів. Програма зберігає першу послідовність до червоно-чорного дерева.

Кожного разу, коли до дерева додається новий елемент, потрібно вивести статистику (згідно варіанту завдання).

1 ОСНОВНІ ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

Дерева у якості 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 - один з типів збалансованих дерев пошуку, в яких передбачені операції балансування, гарантують, що висота дерева не перевищить O (lg 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 глибини будь-яких двох листя відрізняються 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 дитина), right (правий дитина) і p (батько). Якщо у вершини відсутня дитина або батько, відповідне поле містить nil. Для зручності ми будемо вважати, що значення nil, що зберігаються в полях left і right, є посиланнями на додаткові (фіктивні) листя дерева. У такому поповнень дереві кожна стара вершина (що містить ключ) має двох дітей.

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 Кожен лист (nil) - чорний.

3 Якщо вершина червона, обидва її дитини чорні.

4 Всі шляхи, що йдуть вниз від кореня до листя, містять однакову кількість чорних вершин.

Операції Tree-Insert і Tree-Delete виконуються на червоно-чорному дереві за час O (lg n), але вони змінюють дерево, і результат може не володіти RB-властивостями. Щоб відновити ці властивості, треба перефарбувати деякі вершини і змінити структуру дерева.

2 ОПИСАННЯ РОЗРОБЛЕНОГО ЗАСТОСУНКУ

Програма реалізована в одному файлі source.cpp

```
#include<iostream>
using namespace std;
enum Color{ RED, BLACK };

struct Node
{
    int data;
    bool color;
    Node* left, * right, * parent;
};
class IncorrectData{ };
class RBTree
```

```
{
private:
    void LeftRotate(Node* x);
    void RightRotate(Node* x);
    void DeleteNode(Node* parent, Node* curr, int stuff);
    void RBDeleteFix(Node* z);
    void RBInFix(Node* z);
    Node* GetRoot() { return root; }
    Node* root;
public:
    RBTree() : root(nullptr) {}
    void Insert(int stuff);
    void Delete(int stuff);
    void GetMin();
    void GetColor(int x);
};

void RBTree::Insert(int stuff)
{
    if (root == nullptr) {
        root = new Node();
        root->data = stuff;
        root->parent = nullptr;
        root->color = BLACK;
        cout << "Element inserted.\n";
    }
    else {
        Node* 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; }
            }
        }
        RBInFix(newnode);
    }
}

void RBTree::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 RBTree::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 RBTree::RBInFix(Node* z)
{
    while (z->parent->color == RED)
    {
        Node* grandparent = z->parent->parent;
        Node* 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;
}

void RBTree::DeleteNode(Node* parent, Node* curr, int stuff) {
    if (curr == nullptr)
    {
        return;
    }
    if (curr->data == stuff) {
        //CASE -- 1
        if (curr->left == nullptr && curr->right == nullptr) {
            if (parent->data == curr->data) { root = nullptr; }
            else if (parent->right == curr) {
                RBDeleteFix(curr);
                parent->right = nullptr;
            }
            else {
                RBDeleteFix(curr);
```

```
                parent->left = nullptr;
            }
        }
        //CASE -- 2
        else if (curr->left != nullptr && curr->right == nullptr) {
            int swap = curr->data;
            curr->data = curr->left->data;
            curr->left->data = swap;
            DeleteNode(curr, curr->right, stuff);
        }
        else if (curr->left == nullptr && curr->right != nullptr) {
            int swap = curr->data;
            curr->data = curr->right->data;
            curr->right->data = swap;
            DeleteNode(curr, curr->right, stuff);
        }
        //CASE -- 3
        else {
            bool flag = false;
            Node* temp = curr->right;
            while (temp->left) { flag = true; parent = temp; temp = temp->left; }
            if (!flag) { parent = curr; }
            int swap = curr->data;
            curr->data = temp->data;
            temp->data = swap;
            DeleteNode(parent, temp, swap);
        }
    }
}

void RBTree::Delete (int stuff) {
    Node* temp = root;
    Node* parent = temp;
    bool flag = false;
    if (!temp)
    {
        DeleteNode(nullptr, nullptr, stuff);
    }
    while (temp) {
        if (stuff == temp->data)
        {
```

```
                flag = true;
                DeleteNode(parent, temp, stuff); break;
            }
            else if (stuff < temp->data)
            {
                parent = temp;
                temp = temp->left;
            }
            else
            {
                parent = temp;
                temp = temp->right;
            }
        }

        if (!flag) { cout << "\nElement doesn't exist in the table"; }
    }

    void RBTree::RBDeleteFix(Node* z)
    {
        while (z->data != root->data && z->color == BLACK) {
            Node* sibling = GetRoot();
            if (z->parent->left == z) {
                if (z->parent->right) { sibling = z->parent->right; }
                if (sibling) {
                    //CASE -- 1
                    if (sibling->color == RED) {
                        sibling->color = BLACK;
                        z->parent->color = RED;
                        LeftRotate(z->parent);
                        sibling = z->parent->right;
                    }
                    //CASE -- 2
                    if (sibling->left == nullptr && sibling->right == nullptr) {
                        sibling->color = RED;
                        z = z->parent;
                    }
                    else if (sibling->left->color == BLACK && sibling->right->color == BLACK)
{
                        sibling->color = RED;
                        z = z->parent;
```

```
                }
                //CASE -- 3
                else if (sibling->right->color == BLACK) {
                    sibling->left->color = BLACK;
                    sibling->color = RED;
                    RightRotate(sibling);
                    sibling = z->parent->right;
                }
                else {
                    sibling->color = z->parent->color;
                    z->parent->color = BLACK;
                    if (sibling->right) { sibling->right->color = BLACK; }
                    LeftRotate(z->parent);
                    z = root;
                }
            }
        }
        else {
            if (z->parent->right == z) {
                if (z->parent->left)
                {
                    sibling = z->parent->left;
                }
                if (sibling) {
                    //CASE -- 1
                    if (sibling->color == RED) {
                        sibling->color = BLACK;
                        z->parent->color = RED;
                        RightRotate(z->parent);
                        sibling = z->parent->left;
                    }
                    //CASE -- 2
                    if (sibling->left == nullptr && sibling->right == nullptr) {
                        sibling->color = RED;
                        z = z->parent;
                    }
                    else if (sibling->left->color == BLACK && sibling->right->color ==
BLACK) {
                        sibling->color = RED;
                        z = z->parent;
                    }
```

```
                    //CASE -- 3
                    else if (sibling->left->color == BLACK) {
                        sibling->right->color = BLACK;
                        sibling->color = RED;
                        RightRotate(sibling);
                        sibling = z->parent->left;
                    }
                    else {
                        sibling->color = z->parent->color;
                        z->parent->color = BLACK;
                        if (sibling->left) { sibling->left->color = BLACK; }
                        LeftRotate(z->parent);
                        z = root;
                    }
                }
            }

        }
    }
    z->color = BLACK;
}


void RBTree::GetMin()
{
    Node* temp = root;
    if (temp->left == nullptr)
    {
        cout << "Найменший елемент дерева:" << endl;
        cout << temp->data << endl;

    }
    else
    {
        while (temp->left != nullptr)
        {
            if (temp->left != nullptr)
            {
                temp = temp->left;
            }
        }
```

```
        cout << "Найменший елемент дерева:" << endl;
        cout << temp->data << endl;
    }
}

void RBTree::GetColor(int x)
{
    Node* temp = root;
    Node* parent = temp;
    bool flag = false;
    if (!temp)
    {
        cout << "Елемент присутній в дереві. Його колір:";
        cout << temp->color;
    }
    while (temp) {
        if (x == temp->data)
        {
            flag = true;
            cout << "Елемент присутній в дереві. Його колір:" << endl;
            if (temp->color == RED)
                cout << "RED" << endl;
            else
                cout << "BLACK" << endl;
            break;
        }
        else if (x < temp->data)
        {
            parent = temp;
            temp = temp->left;
        }
        else
        {
            parent = temp;
            temp = temp->right;
        }
    }

    if (!flag) { cout << "Елемент відсутній в дереві\n"; }
}
int main()
```

```
{
    setlocale(LC_ALL, "Ukr");
    try
    {
        RBTree a;
        int n, m;
        cout << "Введіть число n, кількість цих чисел буде записана до червоно-чорнго дерева" << endl;
        cin >> n;
        if (n > 256 || n < 0)
            throw IncorrectData();
        for (int i = 0; i < n; i++)
        {
            int x;
            cout << "Введіть елемент для введення" << endl;
            cin >> x;
            a.Insert(x);
            a.GetMin();
        }
        cout << "Введіть число m, кількість цих чисел буде перевірена на другу умову" << endl;
        cin >> m;
        if (n > 256 || n < 0)
            throw IncorrectData();
        for (int i = 0; i < m; i++)
        {
            int x;
            cout << "Введіть елемент для перевірки" << endl;
            cin >> x;
            a.GetColor(x);
        }
    }
    catch (IncorrectData)
    {
        cout << "Неправильні введені числа!"<<endl;
    }
    return 0;
}
```

Результати роботи програми наведені на рис. 3.1:

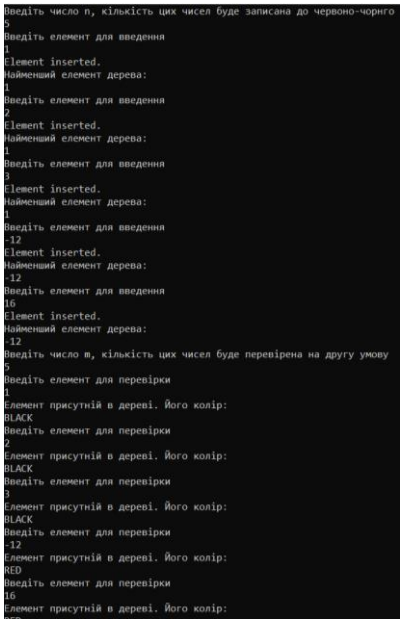

Рисунок 3.1 – Результат

ВИСНОВКИ

Під час цієї лабораторної роботи була розроблена програма, яка створює червоно-чорні дерева з введених з клавіатури символів, які заповнюють структуру даних, що визначена відповідно мого персонального варіанту, а саме червоно-чорне дерево, яким є ціле число, використавши цей вид дерев. Під час цієї лабораторної роботи була викладена інформація про різні типи динамічних структур даних та їх використання при виконанні різних програм. Ця програма може виконувати різні дії, які визначені в роботі з червоно-чорними деревами.

СПИСОК ВИКОРИСТАНИХ ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1 Методичні вказівки до виконання лабораторних робіт з курсу "Алгоритми і структури даних": для студентів, які навчаються за спец. 121 "Інженерія програмного забезпечення" [Електронний ресурс] / уклад. Н. К. Стратієнко, І. О. Бородіна ; Харківський політехнічний інститут, національний технічний університет університет – Електрон. текстові дані. – Харків, 2017. – 36 с. – Режим доступу: http://repository.kpi.kharkov.ua/handle/KhPI-Press/26426. (дата звернення: 25.02.2020)