МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Бинарные деревья поиска

Студент гр. 8304 _____ Мешков М.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2019

Цель работы.

Получить опыт работы с бинарными деревьями поиска, изучить на практике особенности их реализации.

Постановка задачи.

Вариант 16.

По заданным различным элементам построить красно-черное дерево поиска. Для построенной структуры данных проверить, входит ли в неё элемент, и если не входит, то добавить его в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Описание алгоритма.

Красно-чёрное дерево — один из видов из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

Красно черное дерево обладает следующими свойствами:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;

2. Корень — как правило чёрный.

3. Все листья — чёрные и не содержат данных.

4. Оба потомка каждого красного узла — чёрные.

5. Любой простой путь от узла-предка до листового узла-потомка, содержит одинаковое число чёрных узлов.

При вставке элемента может встретиться 4 случая:

1. Вставляемый элемент не имеет родителя, значит он корень и должен быть перекрашен в черный.

2. Вставляемый элемент имеет черного родителя, в этом случае нужно просто вставить элемент, дерево будет корректным.

3. Вставляемый элемент имеет красного дядю, в этом случае этот дядя и родетель перекрашиваются в черный, а дедушка перекрашивается в красный. Затем начинается проверка корректности начиная с дедушки.

4. Дерево имеет черного дядю (или не имеет его вовсе) и красного родителя. В этом случае может быть осуществлен поворот вокруг родителя, направление которого зависит от того является ли родитель правым или левым ребенком дедушки. Затем родитель перекрашивается в черный, а дедушка - в красный и осуществляется поворот вокруг дедушки, направление которого зависит от того является ли добавляемый элемент левым или правым ребенком своего родителя.

Описание основных структур данных и функций.

Функция main осуществляет всё взаимодействие с пользователем, она приводит к созданию всех остальных используемых структур данных.

Специально для решения данной задачи был написан класс RedBlackTree, реализующий красно-черное дерево с использованием динамической памяти.

Этот класс имеет следующий интерфейс:

- setEventInfoOutput - позволяет установить поток, в который будет посылаться информация о том, что происходит в дереве,
- insert - позволяет добавить новое значение в дерево,
- contains - позволяет проверить наличие в дереве переданного значения,
- height - возвращает высоту дерева,
- toBracketView - возвращает скобочное предствавление дерева.

Тестирование.

Программа была успешно протестирована. Ниже приведены основные проверочные входные данные.

Ввод	Дерево после вставки
3 8 3 9 10 1 end	(((# 1R #) 3B #) 8B (# 9B (# 10R #)))
1 2 3 4 5 e	((# 1B #) 2B ((# 3R #) 4B (# 5R #)))
1 1 1 1 end	(# 1B #)
end	пустое дерево
5 8 13 5 2 g	(((# 2R #) 5B #) 8B (# 13B #))
7 6 r	((# 6R #) 7B #)
1 2 1 2 3 m	((# 1R #) 2B (# 3R #))

Выводы.

В ходе выполнения работы был получен опыт работы с красно-черным бинарным деревом поиска в контексте выполнения данной задачи, изучены методы работы с ним. Была написана реализация красно-черного бинарного дерева поиска через динамическую память.

ПРИЛОЖЕНИЕ А

Файл main.cpp

```
#include <iostream>
#include "redblacktree.h"

int main()
{
    RedBlackTree<int> t;
    t.setEventInfoOutput(&std::cout);

    std::cout << "Enter a number(s) to insert it to the tree (or
non-number to exit):" << std::endl;

    for (int n = 0; std::cin >> n; ) {
        std::cout << std::endl;
        if (!t.contains(n)) {
            std::cout << "Inserting value " << n << std::endl;
            t.insert(n);
            std::cout << "Bracket view: " <<
t.toBracketView(std::to_string) << std::endl;
            std::cout << "Tree height = " << t.height() <<
std::endl;
        }
        else
            std::cout << "Rejecting value " << n << " because it
already exists." << std::endl;
        std::cout << std::endl;
    }
    return 0;
}
```

Файл redblacktree.cpp

```
#pragma once

#include <iostream>
#include <cassert>

template<typename T>
class RedBlackTree {
public:
    RedBlackTree() = default;
    RedBlackTree(const RedBlackTree<T> &) = delete;
    ~RedBlackTree() {
        erase(m_root);
```

```
    }
    RedBlackTree &operator=(const RedBlackTree<T> &) = delete;

    void setEventInfoOutput(std::ostream *s) {
        m_eventInfoOutput = s;
    }

    void insert(const T &value) {
        Node *node = insertStupidly(value);
        insertCase1(node);
#ifndef NDEBUG
        validate();
#endif
    }

    bool contains(const T &value) const {
        Node *current = m_root;
        while (true) {
            if (current == nullptr)
                return false;

            if (value > current->value)
                current = current->right;
            else if (value < current->value)
                current = current->left;
            else
                return true;
        }
    }

    size_t height() const {
        return height(m_root);
    }

    std::string toBracketView(std::string (*valueToString)(T))
const {
        return toBracketView(m_root, valueToString);
    }

private:
    struct Node {
        T value;
        Node *parent = nullptr, *left = nullptr, *right = nullptr;
        enum class Color {
```

```
        RED, BLACK
    } color = Color::RED;
};
Node *m_root = nullptr;
std::ostream *m_eventInfoOutput = nullptr;

void erase(Node *node) {
    if (node == nullptr)
        return;
    erase(node->left);
    erase(node->right);
    delete node;
}

void rotateLeft(Node *node) {
    Node *pivot = node->right;

    pivot->parent = node->parent;
    if (node->parent != nullptr) {
        if (node->parent->left == node)
            node->parent->left = pivot;
        else
            node->parent->right = pivot;
    }
    else {
        m_root = pivot;
    }

    node->right = pivot->left;
    if (pivot->left != nullptr)
        pivot->left->parent = node;

    node->parent = pivot;
    pivot->left = node;
}

void rotateRight(Node *node) {
    Node *pivot = node->left;

    pivot->parent = node->parent;
    if (node->parent != nullptr) {
        if (node->parent->left == node)
            node->parent->left = pivot;
        else
```

```
            node->parent->right = pivot;
        }
        else {
            m_root = pivot;
        }

        node->left = pivot->right;
        if (pivot->right != nullptr)
            pivot->right->parent = node;

        node->parent = pivot;
        pivot->right = node;
    }

    Node *grandparent(Node *node) {
        if (node != nullptr && node->parent != nullptr)
            return node->parent->parent;
        else
            return nullptr;
    }

    Node *uncle(Node *node) {
        Node *g = grandparent(node);
        if (g == nullptr)
            return nullptr;
        if (node->parent == g->left)
            return g->right;
        else
            return g->left;
    }

    [[nodiscard]] Node *insertStupidly(const T &value) {
        Node *node = new Node;
        node->value = value;

        Node *prevCurrent = nullptr;
        Node **current = &m_root;
        while (true) {
            if (*current == nullptr) {
                *current = node;
                node->parent = prevCurrent;
                break;
            }
            prevCurrent = *current;
```

```
            if (value > (*current)->value)
                current = &(*current)->right;
            else
                current = &(*current)->left;
        }

        return node;
    }

    void insertCase1(Node *node) {
        if (node->parent == nullptr) {
            // Node has no parent
            node->color = Node::Color::BLACK;

            if (m_eventInfoOutput != nullptr)
                *m_eventInfoOutput << "Inserting case 1: The node
got black color." << std::endl;
        }
        else
            // Node has parent
            insertCase2(node);
    }

    void insertCase2(Node *node) {
        if (node->parent->color == Node::Color::BLACK) {
            // Node has black parent
            if (m_eventInfoOutput != nullptr)
                *m_eventInfoOutput << "Inserting case 2: Do
nothing special." << std::endl;
            return;
        }
        else
            // Node has red parent
            insertCase3(node);
    }

    void insertCase3(Node *node) {
        Node *u = uncle(node);

        if (u != nullptr && u->color == Node::Color::RED) {
            // Node has red uncle
            node->parent->color = Node::Color::BLACK;
            u->color = Node::Color::BLACK;
            Node *g = grandparent(node);
```

```
            g->color = Node::Color::RED;

            if (m_eventInfoOutput != nullptr)
                *m_eventInfoOutput << "Inserting case 3: The uncle
and the parent is recolored to black, the grandparent is recolored
to red. Start working with the grandparent:" << std::endl;

            insertCase1(g);
        }
        else {
            // Node has red parent and black or no uncle
            insertCase4Step1(node);
        }
    }

    void insertCase4Step1(Node *node) {
        Node *g = grandparent(node);

        if (node == node->parent->right && node->parent == g-
>left) {
            rotateLeft(node->parent);
            node = node->left;

            if (m_eventInfoOutput != nullptr)
                *m_eventInfoOutput << "Inserting case 4 step 1:
Rotation to left around the parent." << std::endl;
        }
        else if (node == node->parent->left && node->parent == g-
>right) {
            rotateRight(node->parent);
            node = node->right;

            if (m_eventInfoOutput != nullptr)
                *m_eventInfoOutput << "Inserting case 4 step 1:
Rotation to right around the parent." << std::endl;
        }

        insertCase4Step2(node);
    }

    void insertCase4Step2(Node *node) {
        Node *g = grandparent(node);

        node->parent->color = Node::Color::BLACK;
```

```
        g->color = Node::Color::RED;

        if (m_eventInfoOutput != nullptr)
            *m_eventInfoOutput << "Inserting case 4 step 2: The
parent recolored to black, the grandparent is recolored to red, ";

        if (node == node->parent->left) {
            rotateRight(g);

            if (m_eventInfoOutput != nullptr)
                *m_eventInfoOutput << "rotation to right around
the grandparent." << std::endl;
        }
        else {
            rotateLeft(g);

            if (m_eventInfoOutput != nullptr)
                *m_eventInfoOutput << "rotation to left around the
grandparent." << std::endl;
        }
    }

    void validate() {
        if (m_root == nullptr)
            return;
        assert(m_root->color == Node::Color::BLACK);

        validateColors(m_root);

        validatePathLengths(m_root);

        if (m_eventInfoOutput != nullptr)
            *m_eventInfoOutput << "Tree validation: ok" <<
std::endl;
    }

    void validateColors(Node *node) {
        if (node == nullptr)
            return;
        if (node->color == Node::Color::RED) {
            assert(node->left == nullptr || node->left->color ==
Node::Color::BLACK);
            assert(node->right == nullptr || node->right->color ==
Node::Color::BLACK);
```

```
        }
        validateColors(node->left);
        validateColors(node->right);
    }

    size_t validatePathLengths(Node *node) {
        if (node == nullptr)
            return 1;
        auto len = validatePathLengths(node->left);
        assert(len == validatePathLengths(node->right));
        if (node->color == Node::Color::BLACK)
            return len + 1;
        else
            return len;
    }

    size_t height(Node *node) const {
        if (node == nullptr)
            return 0;
        return std::max(height(node->left), height(node->right)) +
1;
    }

    std::string toBracketView(Node *node, std::string
(*valueToString)(T)) const {
        if (node == nullptr)
            return "#";
        return "(" + toBracketView(node->left, valueToString) +
            " " + valueToString(node->value) +
            (node->color == Node::Color::BLACK ? "B " : "R ") +
            toBracketView(node->right, valueToString) + ")";
    }
};
```