БУ ВО Ханты-Мансийского автономного округа – Югры

«Сургутский государственный университет»

Политехнический институт

Кафедра автоматики и компьютерных систем

ОТЧЁТ

по лабораторной работе №6

по дисциплине: «Алгоритмы и структуры данных»

Выполнил: студент(ка) группы №609-31,

Гаврилов Е.Е.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2025г.

**Цель работы:** изучить основные принципы построения сбалансированных и цифровых деревьев, их свойства и назначение, закрепить навыки структурного программирования.

**Задание:**

1. Реализовать один из вариантов задания в соответствии с таблицей 2 и тестовое приложение. При необходимости реализовать вспомогательные структуры данных и функции, например, функции вывода на дерева на экран, удаления одного узла и т.п.

2. Составить отчет, в котором привести листинги реализованных функций, свойства и отличительные особенности дерева, для которого были реализованы функции и выводе по работе.

**Вариант 3:** вставка ключа в красно-черное дерево

**Свойства** и **отличительные особенности** дерева, для которого были реализованы функции:

**Красно-черное дерево** – бинарное дерево поиска, каждый узел которого может быть красным или черным. Корень такого дерева обязательно черный (корни поддеревьев не обязательно черные). Если узел красный, то все его потомки черные. Все фиктивные NULL –узлы (листья) - черные. Количество черных узлов, которые встречаются на пути от корня дерева до любого из листьев одинаково для всех листьев. Поддержка сбалансированности поддерживается путем изменения цвета узлов и поворотов.

Алгоритм вставки ключа в красно-черное дерево:

По умолчанию, считается, что новый узел красного цвета.

1. Если дерево пустое, вставить узел в корень, перекрасить в черный, вставка окончена.
2. Иначе вставить узел в подходящее место.
3. Если новый родитель узла черный, все в порядке, вставка окончена.
4. Иначе, кроме «родителя», необходимо рассмотреть следующие узды: «дед» (родитель родителя) и «дядя» (второй дочерний узел деда).
5. Если дядя красный, отец и дядя перекрашиваются в черный, дед (если не является корнем дерева) перекрашивается в красный. Вставка завершена.
6. Если дядя черный (или NULL, ведь по свойствам дерево NULL-узел тоже черный), проверить, не образует ли цепочка дед-родитель-узел зигзаг. Если да, произвести поворот вокруг узла-родителя, так что бы образовалась прямая цепочка дед-родитель-узел).
7. Если в результате вставки или в результате п. 6 образовала прямая цепочка дед-родитель-узел, перекрасить деда в красный, родителя в черный, произвести поворот относительно родителя, так что бы родитель стал родителем для узла и для деда. Вставка завершена.

**Вывод:** были изучены основные принципы построения сбалансированных деревьев на основе красно-черного дерева. Разработан метод вставки нового узла в дерево.

Листинг 1. Redblacktree.h

#pragma once

#include <iostream>

using namespace std;

enum class Color

{

RED,

BLACK

};

template <typename T>

class RBTree;

template <typename T>

class RBNode

{

friend class RBTree<T>;

private:

T data;

RBNode<T> \*left;

RBNode<T> \*right;

RBNode<T> \*parent;

Color color;

RBNode(T value);

void Clear();

void Print(int level);

};

template <typename T>

class RBTree

{

private:

public:

RBNode<T> \*root;

RBTree<T>();

RBTree<T>(RBNode<T> \*node);

RBTree<T>(T value);

~RBTree<T>();

void Insert(RBNode<T> \*node);

void Insert(T data);

void Clear();

void Print();

};

#include "redblacktree.tpp"

Листинг 2. Redblacktree.tpp

#include "redblacktree.h"

template <typename T>

RBNode<T>::RBNode(T value)

{

color = Color::BLACK;

left = nullptr;

right = nullptr;

parent = nullptr;

data = value;

}

template <typename T>

void RBNode<T>::Clear()

{

if (left != nullptr)

{

left->Clear();

delete left;

}

if (right != nullptr)

{

right->Clear();

delete right;

}

}

template <typename T>

void RBNode<T>::Print(int level)

{

if (left != nullptr)

{

left->Print(level+1);

}

cout << data << " " << (color == Color::RED ? "R" : "B") << ":" <<level<< endl;

if(right != nullptr){

right->Print(level+1);

}

}

////////////////////////////////////////////

template <typename T>

RBTree<T>::RBTree()

{

root = nullptr;

}

template <typename T>

RBTree<T>::RBTree(RBNode<T> \*node)

{

root = node;

node->color = Color::BLACK;

node->left = nullptr;

node->right = nullptr;

node->parent = nullptr;

}

template <typename T>

RBTree<T>::RBTree(T value)

{

root = new RBNode(value);

}

template <typename T>

RBTree<T>::~RBTree()

{

Clear();

}

template <typename T>

void RBTree<T>::Clear()

{

if (nullptr != root)

{

root->Clear();

delete root;

root = nullptr;

}

}

template <typename T>

void RBTree<T>::Print()

{

if (root != nullptr)

{

root->Print(1);

}

}

template <typename T>

void RBTree<T>::Insert(T data)

{

Insert(new RBNode(data));

}

template <typename T>

void RBTree<T>::Insert(RBNode<T> \*node)

{

if (nullptr == node)

return;

node->left = nullptr;

node->right = nullptr;

node->parent = nullptr;

// root is NULL

if (nullptr == root)

{

root = node;

node->color = Color::BLACK;

return;

}

// finding right place

// get p - parent

RBNode<T> \*p = root;

node->color = Color::RED;

while (nullptr == node->parent)

{

if (node->data < p->data)

if (nullptr == p->left)

{

p->left = node;

node->parent = p;

}

else

p = p->left;

else if (nullptr == p->right)

{

p->right = node;

node->parent = p;

}

else

p = p->right;

}

if (p->color == Color::BLACK)

return;

//get g - grabdparen and u - uncle

RBNode<T> \*g = p->parent;

RBNode<T> \*u = g->left == p ? g->right : g->left;

// red uncle

if (nullptr != u && u->color == Color::RED)

{

p->color = Color::BLACK;

u->color = Color::BLACK;

if (g != root)

g->color = Color::RED;

return;

}

// zig-zag

if (p->data < g->data && node->data >= p->data)

{

g->left = node;

node->parent = g;

node->left = p;

p->parent = node;

p->right = nullptr;

RBNode<T> \*temp = node;

node = p;

p = temp;

}

else if (g->data <= p->data && p->data < node->data)

{

g->right = node;

node->parent = g;

node->right = p;

p->parent = node;

p->left = nullptr;

RBNode<T> \*temp = node;

node = p;

p = temp;

}

// straight line

if (node->data < p->data && p->data < g->data)

{

g->color = Color::RED;

p->color = Color::BLACK;

RBNode<T>\* gg = g->parent;

if (nullptr == gg)

root = p;

else{

if( g->data < gg->data )

gg->left = p;

else

gg->right = p;

}

p->parent = g->parent;

p->right = g;

g->parent = p;

g->left = nullptr;

}

else if (g->data <= p->data && p->data <= node->data)

{

g->color = Color::RED;

p->color = Color::BLACK;

if (g == root)

root = p;

RBNode<T>\* gg = g->parent;

if (nullptr == gg)

root = p;

else{

if( g->data < gg->data )

gg->left = p;

else

gg->right = p;

}

p->parent = g->parent;

p->left = g;

g->right = nullptr;

g->parent = p;

}

}

Листинг 3. Lab06.cpp

#include "redblacktree.h"

#include <iostream>

using namespace std;

int main(){

RBTree<int>\*a = new RBTree<int>();

a->Insert(6);

a->Insert(7);

a->Insert(5);

a->Insert(3);

a->Insert(1);

a->Insert(4);

a->Insert(2);

a->Print();

delete a;

}