**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

ВАРИАНТ 1

Студент гр. 9309 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Корягин Е.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2021

**Постановка задачи**

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

Список методов:

Метод insert – Метод добавляет элемент в дерево.

Метод find – Метод проверяет наличие введенного элемента в дереве.

Метод remove – Метод удаляет выбранный элемент дерева.

Метод left\_turn – осуществляет левый поворот определенного элемента дерева

Метод right\_turn – осуществляет правый поворот определенного элемента дерева

Метод tree\_repair – восстанавливает свойства красно-черного дерева после вставки нового элемента

Метод delete\_tree – восстанавливает свойства красно-черного дерева после удаления неконечного элемента

Метод delete\_tree\_nul – восстанавливает свойства красно-черного дерева после удаления конечного элемента

Метод create\_queue – осуществляет обход дерева в ширину и создает импровизированный массив в виде линейного списка всех обойдённых узлов дерева.

Метод clear – удаляет дерево и высвобождает занятую элементами память (используется в роли деструктора)

Метод get\_keys – выводит список ключей дерева

Метод get\_values – выводит список значений в виде ключ->значение дерева

Метод print – выводит все элементы дерева в виде ключ, значение, цвет. Осуществляется засчет обхода в ширину

**Описание реализуемых классов и методов**

Описание реализуемого класса:

Класс RBTree хранит в себе все необходимые указатели и данные красно-черного дерева.

**Оценка временной сложности методов**

Временная сложность метода:

|  |  |
| --- | --- |
| insert = | O(logN^2). |
| find = | O(log N). |
| remove = | O(logN^3). |
| left\_turn = | O(logN). |
| right\_turn = | O(logN). |
| tree\_repair = | O(N). |
| delete\_tree = | O(N). |
| delete\_tree\_nul = | O(N). |
| create\_queue = | O(logN). |
| clear = | O(logN). |
| get\_keys = | O(logN). |
| get\_values = | O(logN). |
| print = | O(logN). |

**Описание реализованных unit-тестов**

TEST\_METHOD(insert) – Проверка функции вставки элемента в дерево, с помощью сравнения элементов дерева с элементами массива с заведомо верной последовательностью ключей после обхода дерева в ширину.

TEST\_METHOD(remove) – Проверка функции удаления элементов из дерева, с помощью сравнения элементов дерева с элементами массива с заведомо верной последовательностью ключей после удаления этих элементов и обхода дерева в ширину.

TEST\_METHOD(find) – Проверка функции, показывающей наличие выбранного элемента в дереве.

**Текст программы**

**Source.cpp**

#include "RBTree.h"

#include <iomanip>

#include <iostream>

#include <locale>

using namespace std;

int main()

{

RBTree\* tree = new RBTree;

tree->insert(15, "one");

tree->insert(20, "two");

tree->insert(25, "three");

tree->get\_keys();

tree->get\_values();

tree->print();

tree->remove(5);

tree->remove(10);

tree->remove(15);

tree->get\_keys();

tree->get\_values();

tree->print();

tree->clear();

}

**RBTree.h**

#pragma once

#include <iomanip>

#include <iostream>

#include <locale>

#include <stdexcept>

using namespace std;

class RBTree

{

bool color; //false = black, true = red.

class RBTree\* left;

class RBTree\* right;

class RBTree\* parent;

class RBTree\* root;

string value;

public:

int key;

class RBTree\* start;

class RBTree\* next;

RBTree()

{

left = NULL;

right = NULL;

parent = NULL;

start = NULL;

next = NULL;

key = 0;

color = false;

value = "";

}

void left\_turn(RBTree\* current)

{

try

{

if (current != NULL && current->right != NULL)

{

RBTree\* newnode = current;

RBTree\* y = current->right;

if (y->left != NULL)

{

newnode->right = y->left;

y->left->parent = newnode;

y->left = NULL;

}

else

{

newnode->right = NULL;

}

if (newnode->parent == NULL)

{

root = y;

y->parent = NULL;

}

else if (newnode->parent->left == newnode)

{

newnode->parent->left = y;

y->parent = newnode->parent;

newnode->parent = NULL;

}

else

{

newnode->parent->right = y;

y->parent = newnode->parent;

newnode->parent = NULL;

}

y->left = newnode;

newnode->parent = y;

}

}

catch (exception&)

{

cout << "Something went wrong in left\_turn()" << endl;

}

}

void right\_turn(RBTree\* current)

{

try

{

if (current != NULL && current->left != NULL)

{

RBTree\* newnode = current->left;

RBTree\* y = current;

if (newnode->right != NULL)

{

y->left = newnode->right;

newnode->right->parent = y;

newnode->right = NULL;

}

else

{

y->left = NULL;

}

if (y->parent == NULL)

{

root = newnode;

newnode->parent = NULL;

}

else if (y->parent->right == y)

{

y->parent->right = newnode;

newnode->parent = y->parent;

y->parent = NULL;

}

else

{

y->parent->left = newnode;

newnode->parent = y->parent;

y->parent = NULL;

}

newnode->right = y;

y->parent = newnode;

}

}

catch (exception&)

{

cout << "Something went wrong in right\_turn()" << endl;

}

}

void insert(int key, string value)

{

try

{

RBTree\* current;

current = root;

if (root == NULL)

{

RBTree\* elem = new RBTree;

elem->color = false;

elem->key = key;

elem->value = value;

elem->left = NULL;

elem->right = NULL;

elem->parent = NULL;

root = elem;

}

else

{

if (key == current->key)

{

cout << "Error! This key is already contained in the tree!\n";

return;

}

point:

while (current->left != NULL && current->right != NULL)

{

if (key < current->key)

{

current = current->left;

}

else if (key > current->key)

{

current = current->right;

}

else

{

cout << "Error! This key is already contained in the tree!\n";

return;

}

}

if (key < current->key)

{

if (current->left == NULL)

{

RBTree\* elem = new RBTree;

elem->color = true;

elem->key = key;

elem->value = value;

elem->left = NULL;

elem->right = NULL;

elem->parent = current;

current->left = elem;

tree\_repair(current->left);

}

else

{

current = current->left;

goto point;

}

}

else if (key > current->key)

{

if (current->right == NULL)

{

RBTree\* elem = new RBTree;

elem->color = true;

elem->key = key;

elem->value = value;

elem->left = NULL;

elem->right = NULL;

elem->parent = current;

current->right = elem;

tree\_repair(current->right);

}

else

{

current = current->right;

goto point;

}

}

else

{

if (key == current->key)

{

cout << "Error! This key is already contained in the tree!\n";

return;

}

}

}

}

catch (exception&)

{

cout << "Something went wrong in insert()" << endl;

}

}

void tree\_repair(RBTree\* newnode)

{

try

{

while (newnode != NULL && newnode->parent != NULL && newnode != root && newnode->parent->color == true) {

if (newnode->parent->parent != NULL && newnode->parent == newnode->parent->parent->left) {

RBTree\* y = newnode->parent->parent->right;

if (y != NULL && y->color == true) {

newnode->parent->color = false;

y->color = false;

newnode->parent->parent->color = true;

newnode = newnode->parent->parent;

}

else {

if (newnode == newnode->parent->right) {

newnode = newnode->parent;

left\_turn(newnode);

}

newnode->parent->color = false;

newnode->parent->parent->color = true;

right\_turn(newnode->parent->parent);

}

}

else if (newnode->parent->parent != NULL) {

RBTree\* y = newnode->parent->parent->left;

if (y != NULL && y->color == true) {

newnode->parent->color = false;

y->color = false;

newnode->parent->parent->color = true;

newnode = newnode->parent->parent;

}

else {

if (newnode == newnode->parent->left) {

newnode = newnode->parent;

right\_turn(newnode);

}

newnode->parent->color = false;

newnode->parent->parent->color = true;

left\_turn(newnode->parent->parent);

}

}

}

root->color = false;

}

catch (exception&)

{

cout << "Something went wrong in tree\_repair()" << endl;

}

}

RBTree\* find(int key)

{

try

{

RBTree\* current = root;

if (current == NULL)

{

return current;

}

if (key == current->key)

{

return current;

}

point:

while (current->left != NULL && current->right != NULL)

{

if (key < current->key)

{

current = current->left;

}

else if (key > current->key)

{

current = current->right;

}

else

{

return current;

}

}

if (key < current->key)

{

if (current->left == NULL)

{

current = current->left;

return current;

}

else

{

current = current->left;

goto point;

}

}

else if (key > current->key)

{

if (current->right == NULL)

{

current = current->right;

return current;

}

else

{

current = current->right;

goto point;

}

}

else

{

if (key == current->key)

{

return current;

}

}

}

catch (exception&)

{

cout << "Something went wrong in find()" << endl;

}

}

void remove(int key)

{

try

{

RBTree\* current = find(key);

if (current != NULL)

{

bool originalcolor;

originalcolor = current->color;

if (current == root && current->left == NULL && current->right == NULL)

{

root = NULL;

delete current;

return;

}

if (current->left == NULL && current->right == NULL)

{

RBTree\* NIL = new RBTree;

if (current->parent->left == current)

{

if (current->color == false)

{

current->parent->left = NIL;

NIL->parent = current->parent;

}

else

current->parent->left = NULL;

}

else

{

if (current->color == false)

{

current->parent->right = NIL;

NIL->parent = current->parent;

}

else

current->parent->right = NULL;

}

if (current->color == false)

{

delete\_repair\_nul(NIL);

}

delete current;

}

else if (current->left == NULL && current->right != NULL)

{

RBTree\* del = current;

current->right->parent = current->parent;

if (current->parent != NULL && current->parent->left == current)

{

current->parent->left = current->right;

if (current->color == false)

delete\_repair(current->right);

}

else if (current->parent != NULL && current->parent->right == current)

{

current->parent->right = current->right;

if (current->color == false)

delete\_repair(current->right);

}

else

{

root = current->right;

if (current->color == false)

delete\_repair(current->right);

}

current = current->right;

delete del;

}

else if (current->left != NULL && current->right == NULL)

{

RBTree\* del = current;

current->left->parent = current->parent;

if (current->parent != NULL && current->parent->left == current)

{

current->parent->left = current->left;

if (current->color == false)

delete\_repair(current->left);

}

else if (current->parent != NULL && current->parent->right == current)

{

current->parent->right = current->left;

if (current->color == false)

delete\_repair(current->left);

}

else

{

root = current->left;

if (current->color == false)

delete\_repair(current->left);

}

current = current->left;

delete del;

}

else

{

RBTree\* point = current;

point = point->right;

if (point->left == NULL)

{

current->key = point->key;

current->value = point->value;

current->right = point->right;

if (point->right != NULL)

{

point->right->parent = current;

if (point->color == false)

delete\_repair(point->right);

}

delete point;

return;

}

RBTree\* point1;

point1 = current;

point1 = point1->right;

while (point1->left != NULL)

point1 = point1->left;

while (point->left != point1)

point = point->left;

if (point1->right == NULL)

{

RBTree\* NIL = new RBTree;

current->key = point1->key;

current->value = point1->value;

if (point1->color == false)

{

point->left = NIL;

NIL->parent = point;

}

else

point->left = NULL;

if (point1->color == false)

delete\_repair\_nul(NIL);

delete point1;

}

else

{

current->key = point1->key;

current->value = point1->value;

point->left = point1->right;

if (point1->right != NULL)

{

point1->right->parent = point;

if (point1->color == false)

delete\_repair(point1->right);

}

}

point = NULL;

point1 = NULL;

delete point, point1;

}

return;

}

else

{

cout << "Element to delete was not found." << endl;

}

}

catch (exception&)

{

cout << "Something went wrong in remove()" << endl;

}

}

void delete\_repair(RBTree\* newnode) {

try {

while (newnode != NULL && newnode->parent != NULL && newnode != root && newnode->color == false) {

if (newnode == newnode->parent->left) {

RBTree\* y = newnode->parent->right;

if (y != NULL && y->color == true) {

y->color = false;

newnode->parent->color = true;

left\_turn(newnode->parent);

y = newnode->parent->right;

}

if (y != NULL && ((y->left == NULL && y->right == NULL) || (y->right == NULL && y->left != NULL && y->left->color == false) || (y->left == NULL && y->right != NULL && y->right->color == false) || (y->left->color == false && y->right->color == false))) {

y->color = true;

newnode = newnode->parent;

}

else if (y != NULL) {

if (y->right == NULL || y->right->color == false) {

if (y->left != NULL)

{

y->left->color = false;

}

y->color = true;

right\_turn(y);

y = newnode->parent->right;

}

y->color = newnode->parent->color;

newnode->parent->color = false;

if (y->right != NULL)

{

y->right->color = false;

}

left\_turn(newnode->parent);

newnode = root;

}

}

else {

RBTree\* y = newnode->parent->left;

if (y != NULL && y->color == true) {

y->color = false;

newnode->parent->color = true;

right\_turn(newnode->parent);

y = newnode->parent->left;

}

if (y != NULL && ((y->left == NULL && y->right == NULL) || (y->right == NULL && y->left != NULL && y->left->color == false) || (y->left == NULL && y->right != NULL && y->right->color == false) || (y->left->color == false && y->right->color == false))) {

y->color = true;

newnode = newnode->parent;

}

else if (y != NULL) {

if (y->left == NULL || y->left->color == false) {

if (y->right != NULL)

{

y->right->color = false;

}

y->color = true;

left\_turn(y);

y = newnode->parent->left;

}

y->color = newnode->parent->color;

newnode->parent->color = false;

if (y->left != NULL)

{

y->left->color = false;

}

right\_turn(newnode->parent);

newnode = root;

}

}

}

if (newnode != NULL)

{

newnode->color = false;

}

}

catch (exception&)

{

cout << "Something went wrong in delete\_repair()" << endl;

}

}

void delete\_repair\_nul(RBTree\* newnode)

{

try

{

if (newnode != NULL && newnode->parent != NULL && newnode != root && newnode->color == false) {

if (newnode == newnode->parent->left) {

RBTree\* y = newnode->parent->right;

if (y != NULL && y->color == true) {

y->color = false;

newnode->parent->color = true;

left\_turn(newnode->parent);

y = newnode->parent->right;

}

if (y != NULL && ((y->left == NULL && y->right == NULL) || (y->right == NULL && y->left != NULL && y->left->color == false) || (y->left == NULL && y->right != NULL && y->right->color == false) || (y->left->color == false && y->right->color == false))) {

y->color = true;

}

else if (y != NULL) {

if (y->right == NULL || y->right->color == false) {

if (y->left != NULL)

{

y->left->color = false;

}

y->color = true;

right\_turn(y);

y = newnode->parent->right;

}

y->color = newnode->parent->color;

newnode->parent->color = false;

if (y->right != NULL)

{

y->right->color = false;

}

left\_turn(newnode->parent);

}

}

else {

RBTree\* y = newnode->parent->left;

if (y != NULL && y->color == true) {

y->color = false;

newnode->parent->color = true;

right\_turn(newnode->parent);

y = newnode->parent->left;

}

if (y != NULL && ((y->left == NULL && y->right == NULL) || (y->right == NULL && y->left != NULL && y->left->color == false) || (y->left == NULL && y->right != NULL && y->right->color == false) || (y->left->color == false && y->right->color == false))) {

y->color = true;

}

else if (y != NULL) {

if (y->left == NULL || y->left->color == false) {

if (y->right != NULL)

{

y->right->color = false;

}

y->color = true;

left\_turn(y);

y = newnode->parent->left;

}

y->color = newnode->parent->color;

newnode->parent->color = false;

if (y->left != NULL)

{

y->left->color = false;

}

right\_turn(newnode->parent);

}

}

}

if (newnode != NULL)

{

newnode->color = false;

}

RBTree\* temp = newnode->parent;

if (newnode->parent->left == newnode)

{

newnode->parent->left = NULL;

delete newnode;

}

else

{

newnode->parent->right = NULL;

delete newnode;

}

delete\_repair(temp);

temp = nullptr;

delete temp;

}

catch (exception&)

{

cout << "Something went wrong in delete\_tree\_nul()" << endl;

}

}

void create\_queue()

{

try

{

RBTree\* marker;

RBTree\* finish;

start = root;

if (start == NULL)

{

return;

}

start->next = NULL;

finish = start;

marker = start;

if (marker->left != NULL)

{

finish->next = marker->left;

finish = finish->next;

finish->next = NULL;

}

if (marker->right != NULL)

{

finish->next = marker->right;

finish = finish->next;

finish->next = NULL;

}

marker = marker->next;

while (marker != NULL)

{

if (marker->left != NULL)

{

finish->next = marker->left;

finish = finish->next;

finish->next = NULL;

}

if (marker->right != NULL)

{

finish->next = marker->right;

finish = finish->next;

finish->next = NULL;

}

if (marker->left == NULL && marker->right == NULL && marker->next == NULL)

{

break;

}

marker = marker->next;

}

marker = NULL;

finish = NULL;

delete marker;

delete finish;

return;

}

catch (exception&)

{

cout << "Something went wrong in create\_queue()" << endl;

}

}

void clear()

{

try

{

create\_queue();

cout << endl;

while (start != NULL)

{

RBTree\* deleter;

deleter = start;

deleter->left = NULL;

deleter->right = NULL;

start = start->next;

delete deleter;

}

cout << "The destructor has finished deleting the specified elements." << endl;

}

catch (exception&)

{

cout << "Something went wrong in clear()" << endl;

}

}

void get\_keys()

{

try

{

create\_queue();

cout << endl;

cout << "Tree keys:" << endl;

while (start != NULL)

{

cout << start->key << ", ";

start = start->next;

}

cout << "NULL" << endl;

}

catch (exception&)

{

cout << "Something went wrong in get\_keys()" << endl;

}

}

void get\_values()

{

try

{

create\_queue();

cout << endl;

cout << "Tree values:" << endl;

while (start != NULL)

{

cout << start->key << "->" << start->value << ", ";;

start = start->next;

}

cout << "NULL" << endl;

}

catch (exception&)

{

cout << "Something went wrong in get\_values()" << endl;

}

}

void print()

{

try

{

create\_queue();

cout << endl;

cout << "BFT\_iteration:" << endl;

while (start != NULL)

{

cout << start->key << " " << start->value << " ";

if (start->color == false)

{

cout << "black" << endl;

}

else

{

cout << "red" << endl;

}

start = start->next;

}

cout << "NULL" << endl;

}

catch (exception&)

{

cout << "Something went wrong in print()" << endl;

}

}

};

**UnitTest1.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include <stdexcept>

#include "../Lab1\_4sem/RBTree.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1

{

TEST\_CLASS(UnitTest1)

{

public:

TEST\_METHOD(insert)

{

RBTree\* tree = new RBTree;

tree->insert(40, "one");

tree->insert(50, "two");

tree->insert(70, "three");

tree->insert(20, "four");

tree->insert(30, "five");

tree->insert(10, "six");

tree->insert(25, "seven");

tree->insert(5, "eight");

int a[8] = {30, 20, 50, 10, 25, 40, 70, 5};

tree->create\_queue();

for (int i = 0; i < 8; i++)

{

Assert::AreEqual(tree->start->key, a[i]);

tree->start = tree->start->next;

}

tree->clear();

}

TEST\_METHOD(remove)

{

RBTree\* tree = new RBTree;

tree->insert(40, "one");

tree->insert(50, "two");

tree->insert(70, "three");

tree->insert(20, "four");

tree->insert(30, "five");

tree->insert(10, "six");

tree->insert(25, "seven");

tree->insert(5, "eight");

tree->remove(5);

tree->insert(5, "nine");

tree->remove(10);

tree->remove(5);

tree->remove(25);

tree->remove(20);

tree->remove(40);

tree->remove(30);

int a[2] = {50, 70};

tree->create\_queue();

for (int i = 0; i < 2; i++)

{

Assert::AreEqual(tree->start->key, a[i]);

tree->start = tree->start->next;

}

tree->clear();

}

TEST\_METHOD(find)

{

RBTree\* tree = new RBTree;

tree->insert(40, "one");

tree->insert(50, "two");

tree->insert(70, "three");

tree->create\_queue();

Assert::AreEqual(tree->find(50)->key, 50);

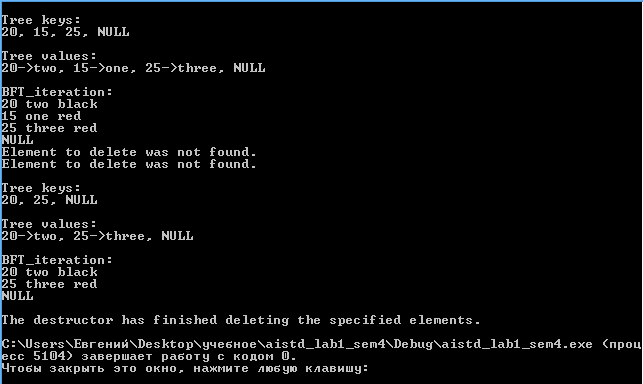
tree->clear();

}

};

}

**Пример работы программы**

****

**Вывод**

Во время выполнения работы мы ознакомились с алгоритмами балансировки двоичных деревьев.