**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»**

**Тема: «Списки и их реализация»**

**Вариант 18(1)**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Плюснина Е.Ю. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**Постановка задачи**

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева. (1 вариант)

# Описание реализуемого класса, оценка временной сложности

class RedBlackTree – класс ассоциативного массива.

Node\* root – указатель на «корень» дерева.

Node\* nil – указатель на нулевой элемент.

int size – размер.

Key find – ключ массива.

Value value – значение массива.

Node\* parent – «предок».

Node\* left – левый элемент.

Node\* right – правый элемент.

Color color – цвет узла.

**Функции:**

void insert(Key, Value) – добавление элемента с ключом и значением. Оценка временной сложности: О(logn)

void remove(Key) – удаление элемента дерева по ключу. Оценка временной сложности: О(logn)

Value find(Key) – поиск элемента по ключу. Оценка временной сложности: О(logn)

void clear() – очищение ассоциативного массива. Оценка временной сложности: О(n)

Key\* get\_keys() – возвращает список ключей. Оценка временной сложности: О(n)

Value\* get\_values() – возвращает список значений. Оценка временной сложности: О(n)

void print() – вывод в консоль. Оценка временной сложности: О(n)

# Описание реализованных unit-тестов

TEST\_CLASS(Empty) – класс для проверки изначально пустого массива.

insert\_empty – вставка в пустой массив.

insert1 – проверка вставки элемента.

remove\_error – проверка попытки удалить из пустого массива.

size – проверка размера.

TEST\_CLASS(NotEmpty) – класс для проверки не пустого массива.

size – проверка размера.

find\_error – ошибка при поиске.

find1 – проверка работы поиска.

get\_values – проверка возвращения списка значений

get\_keys – проверка возвращения списка ключей.

insert – проверка вставки.

remove\_end – проверка удаления из конца (листа).

remove\_error – ошибка при удалении.

remove\_middle – удаление из середины (элемента с двумя детьми).

remove\_root – удаление корневого элемента.

clear – очистка массива.

# Листинг

**Stack.h**

#pragma once

#include <iostream>

template <class T>

class Stack

{

private:

class Node

{

public:

Node(T data = 0, Node\* previous\_branch = NULL, Node\* next\_branch = NULL)

{

this->data = data;

this->previous\_branch = previous\_branch;

this->next\_branch = next\_branch;

};

~Node()

{ }

T data;

Node\* previous\_branch;

Node\* next\_branch;

private:

};

Node\* first;

Node\* last;

size\_t size;

public:

Stack(Node\* first = NULL, Node\* last = NULL) {

this->first = first;

this->last = last;

size = 0;

};

size\_t get\_size();

void push(T);

T pop();

bool IsEmpty();

void clear();

~Stack()

{

if (first != NULL) {

while (first != NULL && first->next\_branch != NULL) //while we can go next

{

first = first->next\_branch;

delete first->previous\_branch;

}

size = 0;

if (first != NULL) delete first;

}

};

};

**Stack.cpp**

#include"Iterator.h"

#include "Stack.h"

using namespace std;

template <typename T>

size\_t Stack<T>::get\_size()

{

return size;

}

template <typename T>

void Stack<T>::push(T add)

{

if (size == 0)

{

first = new Node(add);

last = first;

}

else

{

Node\* this\_branch = new Node(add);

this\_branch->previous\_branch = last;

last->next\_branch = this\_branch;

last = this\_branch;

}

size++;

}

template <typename T>

T Stack<T>::pop()

{

T this\_branch;

if (size == 1)

{

size = 0;

this\_branch = last->data;

first = NULL;

last = NULL;

}

else if (size > 1)

{

this\_branch = last->data;

last = last->previous\_branch;

delete last->next\_branch;

last->next\_branch = NULL;

size--;

}

else {

throw out\_of\_range("Error");

}

return this\_branch;

}

template <typename T>

bool Stack<T>::IsEmpty()

{

if (first == NULL) {

return true;

}

return false;

}

template <typename T>

void Stack<T>::clear()

{

if (first != NULL)

{

while (first->next\_branch != NULL)

{

first = first->next\_branch;

delete first->previous\_branch;

}

size = 0;

first = NULL;

last = NULL;

}

}

**Iterator.h**

#pragma once

template <class T>

class Iterator {

public:

virtual T next() = 0;

virtual bool has\_next() = 0;

};

**RB-Tree.h**

#pragma once

#include"Iterator.h"

#include"Stack.h"

#include "Stack.cpp"

enum Color {Red, Black};

template <class Key, class Value>

class RedBlackTree {

private:

class Node {

public:

Key find;

Value value;

Node\* parent;

Node\* left;

Node\* right;

Color color;

Node(Key find, Value value, Node\* left = NULL, Node\* right = NULL, Node\* parent = NULL, Color color = Red) {

this->find = find;

this->value = value;

this->parent = parent;

this->left = left;

this->right = right;

this->color = color;

}

~Node() {};

};

class dft\_Iterator

{

private:

Stack<Node\*> Stack;

Node\* IterCurrent;

Node\* nil;

public:

dft\_Iterator(Node\* cur, Node\* nill) {

IterCurrent = cur;

nil = nill;

};

Node\* next() {

Node\* temp = IterCurrent;

if (IterCurrent->right != nil) {

Stack.push(IterCurrent->right);

}

if (IterCurrent->left != nil) {

IterCurrent = IterCurrent->left;

}

else {

if (!Stack.IsEmpty()) {

IterCurrent = Stack.pop();

}

else {

IterCurrent = nil;

}

}

return temp;

};

bool has\_next() {

if (IterCurrent != nil) {

return true;

}

return false;

};

~dft\_Iterator() {}

};

Node\* root;

Node\* nil;

int size;

void recovery(Node\*);

void leftRotate(Node\*);

void rightRotate(Node\*);

void recoveryRemove(Node\* x);

public:

RedBlackTree() {

nil = new Node(0, 0, NULL, NULL, NULL, Black);

root = nil;

size = 0;

}

~RedBlackTree() {

delete root;

clear();

}

void insert(Key, Value);

void remove(Key);

Value find(Key);

void clear();

Key\* get\_keys();

Value\* get\_values();

void print();

int get\_size();

bool get\_color(Key);

friend class Node;

};

**RB-Tree.cpp**

#include"Iterator.h"

#include "RB-Tree.h"

using namespace std;

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::insert(Key find, Value value) {

Node\* current = root, \* parent;

bool moveRight;

size++;

if (root == nil) {

root = new Node(find, value, nil, nil, nil, Black);

return;

}

parent = current;

while (current != nil) {

parent = current;

if (find > current->find) {

current = current->right;

moveRight = true;

}

else {

current = current->left;

moveRight = false;

}

}

if (moveRight) {

parent->right = new Node(find, value, nil, nil, parent, Red);

recovery(parent->right);

}

else {

parent->left = new Node(find, value, nil, nil, parent, Red);

recovery(parent->left);

}

}

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::recovery(Node\* New\_Node) {

Node\* parent, \* grandParent;

while (New\_Node->parent->color == Red) {

parent = New\_Node->parent;

grandParent = parent->parent;

if (grandParent->left == parent) {

if (grandParent->right->color == Red) {

grandParent->right->color = Black;

grandParent->left->color = Black;

grandParent->color = Red;

New\_Node = grandParent;

}

else {

if (parent->right == New\_Node) {

New\_Node = parent;

leftRotate(New\_Node);

}

New\_Node->parent->color = Black;

New\_Node->parent->parent->color = Red;

rightRotate(New\_Node->parent->parent);

}

}

else {

if (grandParent->left->color == Red) {

grandParent->right->color = Black;

grandParent->left->color = Black;

grandParent->color = Red;

New\_Node = grandParent;

}

else {

if (parent->left == New\_Node) {

New\_Node = parent;

rightRotate(New\_Node);

}

New\_Node->parent->color = Black;

New\_Node->parent->parent->color = Red;

leftRotate(New\_Node->parent->parent);

}

}

}

root->color = Black;

}

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::leftRotate(Node\* current\_to\_X) {

if (current\_to\_X->right == nil) return;

Node\* current\_to\_Y = current\_to\_X->right;

if (current\_to\_Y->left != nil) {

current\_to\_X->right = current\_to\_Y->left;

current\_to\_Y->left->parent = current\_to\_X;

}

else current\_to\_X->right = nil;

if (current\_to\_Y != nil) {

current\_to\_Y->parent = current\_to\_X->parent;

}

if (current\_to\_X->parent != nil) {

if (current\_to\_X == current\_to\_X->parent->left) {

current\_to\_X->parent->left = current\_to\_Y;

}

else {

current\_to\_X->parent->right = current\_to\_Y;

}

}

else {

current\_to\_Y->parent = nil;

root = current\_to\_Y;

}

current\_to\_Y->left = current\_to\_X;

if (current\_to\_X != nil) {

current\_to\_X->parent = current\_to\_Y;

}

}

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::rightRotate(Node\* current\_to\_X) {

Node\* current\_to\_Y = current\_to\_X->left;

current\_to\_X->left = current\_to\_Y->right;

if (current\_to\_Y->right != nil) {

current\_to\_Y->right->parent = current\_to\_X;

}

if (current\_to\_Y != nil) {

current\_to\_Y->parent = current\_to\_X->parent;

}

if (current\_to\_X->parent != nil) {

if (current\_to\_X == current\_to\_X->parent->right) {

current\_to\_X->parent->right = current\_to\_Y;

}

else {

current\_to\_X->parent->left = current\_to\_Y;

}

}

else {

root = current\_to\_Y;

}

current\_to\_Y->right = current\_to\_X;

if (current\_to\_X != nil) {

current\_to\_X->parent = current\_to\_Y;

}

}

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::remove(Key find) {

if (root == nil) {

throw out\_of\_range("Tree is empty");

}

Node\* ToDelete = root;

while (ToDelete->find != find) {

if (find > ToDelete->find) {

ToDelete = ToDelete->right;

}

else {

ToDelete = ToDelete->left;

}

if (ToDelete == nil) {

throw out\_of\_range("No such element");

}

}

Node\* x, \* min;

bool originalColor = ToDelete->color;

if (ToDelete->left == nil) {

x = ToDelete->right;

if (ToDelete->parent == nil) {

root = x;

}

else

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

ToDelete->parent->left = x;

}

else {

ToDelete->parent->right = x;

}

x->parent = ToDelete->parent;

}

else if (ToDelete->right == nil) {

x = ToDelete->left;

if (ToDelete->parent == nil) {

root = x;

}

else if (ToDelete->parent->left == ToDelete) {

ToDelete->parent->left = x;

}

else {

ToDelete->parent->right = x;

}

x->parent = ToDelete->parent;

}

else {

Node\* temp = ToDelete;

min = ToDelete->right;

while (min->left != nil) min = min->left;

originalColor = min->color;

ToDelete = min;

x = ToDelete->right;

if (ToDelete->parent == temp) {

x->parent = ToDelete;

}

else {

if (ToDelete->parent == nil) {

root = x;

ToDelete->right->parent = nil;

}

else if (ToDelete->parent->right == ToDelete) {

ToDelete->parent->right = x;

}

else {

ToDelete->parent->left = x;

}

x->parent = ToDelete->parent;

ToDelete->right = temp->right;

ToDelete->right->parent = ToDelete;

}

if (temp->parent == nil) {

root = ToDelete;

}

else if (temp == temp->parent->left) {

temp->parent->left = ToDelete;

}

else {

temp->parent->right = ToDelete;

}

ToDelete->parent = temp->parent;

ToDelete->left = temp->left;

ToDelete->left->parent = ToDelete;

ToDelete->color = temp->color;

}

if (originalColor == Black) recoveryRemove(x);

size--;

}

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::recoveryRemove(Node\* x) {

Node\* brother;

while (x != root && x->color == Black) {

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

brother = x->parent->right;

if (brother->color == Red) {

brother->color = Black;

x->parent->color = Red;

leftRotate(x->parent);

brother = x->parent->right;

}

if (brother->left->color == Black && brother->right->color == Black) {

brother->color = Red;

x = x->parent;

}

else {

if (brother->right->color == Black) {

brother->left->color = Black;

brother->color = Red;

rightRotate(brother);

brother = x->parent->right;

}

brother->color = x->parent->color;

x->parent->color = Black;

brother->right->color = Black;

leftRotate(x->parent);

x = root;

}

}

else {

brother = x->parent->left;

if (brother->color == Red) {

brother->color = Black;

x->parent->color = Red;

rightRotate(x->parent);

brother = x->parent->left;

}

if (brother->right->color == Black && brother->left->color == Black) {

brother->color = Red;

x = x->parent;

}

else {

if (brother->left->color == Black) {

brother->right->color = Black;

brother->color = Red;

leftRotate(brother);

brother = x->parent->left;

}

brother->color = x->parent->color;

x->parent->color = Black;

brother->left->color = Black;

rightRotate(x->parent);

x = root;

}

}

}

x->color = Black;

}

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::clear() {

while (root != nil) {

remove(root->find);

cout << endl;

print();

}

}

template<typename Key, typename Value>

Value RedBlackTree<Key, Value>::find(Key findKey) {

Node\* cur = root;

while (cur->find != findKey) {

if (findKey > cur->find) {

cur = cur->right;

}

else {

cur = cur->left;

}

if (cur == nil) {

throw out\_of\_range("No such element");

}

}

return cur->value;

}

template<typename Key, typename Value>

Key\* RedBlackTree<Key, Value>::get\_keys() {

Key\* arr = new Key[size]; int i = 0;

dft\_Iterator Iterator(root, nil);

while (Iterator.has\_next()) {

arr[i] = Iterator.next()->find; i++;

}

return arr;

}

template<typename Key, typename Value>

Value\* RedBlackTree<Key, Value>::get\_values() {

Value\* arr = new Value[size]; int i = 0;

dft\_Iterator Iterator(root, nil);

while (Iterator.has\_next()) {

arr[i] = Iterator.next()->value; i++;

}

return arr;

}

template<typename Key, typename Value>

void RedBlackTree<Key, Value>::print() {

dft\_Iterator Iterator(root, nil);

Node\* current;

while (Iterator.has\_next())

{

current = Iterator.next();

cout << "\nKey: " << current->find << " Color: " << current->color << endl;;

cout << " Kids " << current->left->find << " " << current->right->find << endl;

}

}

template<typename Key, typename Value>

bool RedBlackTree<Key, Value>::get\_color(Key findKey) {

Node\* current = root;

while (current->find != findKey) {

if (findKey > current->find) {

current = current->right;

}

else {

current = current->left;

}

if (current == nil) {

throw out\_of\_range("No such element");

}

}

return current->color;

}

template<typename Key, typename Value>

int RedBlackTree<Key, Value>::get\_size() {

return size;

}

**UnitTest\_4sem\_1lab.cpp**

#include "CppUnitTest.h"

#include "..\AlgStrD\_4sem\_lab1\RB-Tree.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Lab1sem4test

{

TEST\_CLASS(Empty)

{

public:

TEST\_METHOD(insert\_empty)

{

RedBlackTree<int, char> tree;

tree.insert(3, 'c');

Assert::AreEqual(tree.get\_color(3), true);

}

TEST\_METHOD(insert\_1) {

RedBlackTree<int, char> tree;

for (int i = 1; i < 7; i++) {

tree.insert(i, (char)i + 56);

}

Assert::AreEqual(tree.get\_color(1), true);

Assert::AreEqual(tree.get\_color(2), true);

Assert::AreEqual(tree.get\_color(3), true);

Assert::AreEqual(tree.get\_color(4), false);

Assert::AreEqual(tree.get\_color(5), true);

Assert::AreEqual(tree.get\_color(6), false);

}

TEST\_METHOD(remove\_error) {

RedBlackTree<int, char> tree;

try {

tree.remove(1);

}

catch (const exception& e) {

Assert::AreEqual(e.what(), "Is empty");

}

}

TEST\_METHOD(size) {

RedBlackTree<int, char> tree;

Assert::AreEqual(tree.get\_size(), 0);

}

};

TEST\_CLASS(NotEmpty) {

RedBlackTree<int, char> tree;

TEST\_METHOD\_INITIALIZE(setup) {

for (int i = 1; i < 7; i++) {

tree.insert(i, (char)i + 56);

}

}

TEST\_METHOD(size) {

Assert::AreEqual(tree.get\_size(), 6);

}

TEST\_METHOD(find\_error) {

try {

tree.find(123);

}

catch (exception e) {

Assert::AreEqual(e.what(), "No element");

}

}

TEST\_METHOD(find1) {

Assert::AreEqual(tree.find(3), (char)59);

}

TEST\_METHOD(insert) {

tree.insert(0, 'g');

Assert::AreEqual(tree.get\_size(), 7);

Assert::AreEqual(tree.get\_color(0), false);

}

TEST\_METHOD(get\_keys) {

int\* arr = new int[tree.get\_size()];

arr = tree.get\_keys();

Assert::AreEqual(arr[0], 3);

Assert::AreEqual(arr[1], 2);

Assert::AreEqual(arr[2], 4);

Assert::AreEqual(arr[3], 1);

Assert::AreEqual(arr[4], 5);

Assert::AreEqual(arr[5], 6);

}

TEST\_METHOD(get\_values) {

char\* arr = new char[tree.get\_size()];

arr = tree.get\_values();

Assert::AreEqual(arr[0], (char)47);

Assert::AreEqual(arr[1], (char)50);

Assert::AreEqual(arr[2], (char)49);

Assert::AreEqual(arr[3], (char)48);

Assert::AreEqual(arr[4], (char)51);

Assert::AreEqual(arr[5], (char)52);

}

TEST\_METHOD(remove\_error) {

try {

tree.remove(123);

}

catch (exception e) {

Assert::AreEqual(e.what(), "No element");

}

}

TEST\_METHOD(remove\_end) {

tree.remove(6);

Assert::AreEqual(tree.get\_size(), 5);

Assert::AreEqual(tree.get\_color(1), true);

Assert::AreEqual(tree.get\_color(2), true);

Assert::AreEqual(tree.get\_color(3), true);

Assert::AreEqual(tree.get\_color(4), false);

Assert::AreEqual(tree.get\_color(5), true);

}

TEST\_METHOD(remove\_middle) {

tree.remove(4);

Assert::AreEqual(tree.get\_size(), 5);

Assert::AreEqual(tree.get\_color(1), true);

Assert::AreEqual(tree.get\_color(2), true);

Assert::AreEqual(tree.get\_color(3), true);

Assert::AreEqual(tree.get\_color(5), false);

Assert::AreEqual(tree.get\_color(6), true);

}

TEST\_METHOD(remove\_root) {

tree.remove(2);

Assert::AreEqual(tree.get\_size(), 5);

Assert::AreEqual(tree.get\_color(1), true);

Assert::AreEqual(tree.get\_color(3), true);

Assert::AreEqual(tree.get\_color(4), true);

Assert::AreEqual(tree.get\_color(5), false);

Assert::AreEqual(tree.get\_color(6), true);

}

TEST\_METHOD(clear) {

tree.clear();

Assert::AreEqual(tree.get\_size(), 0);

}

};

}

# Вывод

Я научилась реализовывать шаблонный ассоциативный массив (map) на основе красно-черного дерева.