**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»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9301 |  | Русанова К.В. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2021

### Постановка задачи. Описание реализуемых алгоритмов

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

Методы:

1. insert(ключ, значение) // добавление элемента с ключом и значением

2. remove(ключ) // удаление элемента дерева по ключу

3. find(ключ) // поиск элемента по ключу

4. clear // очищение ассоциативного массива

5. get\_keys // возвращает список ключей

6. get\_values // возвращает список значений

7. print // вывод в консоль

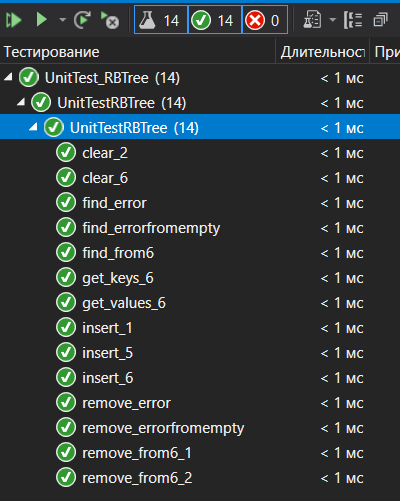
### Оценка временной сложности

Табл.

|  |  |
| --- | --- |
| **Название метода** | **Сложность** |
| insert(ключ, значение) | O(log(n)) |
| remove(ключ) | O(log(n)) |
| find(ключ) | O(log(n)) |
| clear | O(n) |
| get\_keys | O(n) |
| get\_values | O(n) |
| print | O(n) |

### Реализованные unit-тесты

1. insert\_1 — Проверка вставки одного элемента
2. insert\_5 — Проверка вставки пяти элементов
3. insert\_6 — Проверка вставки шести элементов
4. remove\_from6\_1 — Проверка удаления одного элемента после вставки шести
5. remove\_from6\_2 — Проверка удаления двух элементов после вставки шести
6. remove\_error — Проверка удаления не существующего элемента
7. remove\_errorfromempty — Проверка удаления из пустого дерева
8. find\_from6 — Проверка поиска элемента после вставки шести
9. find\_error — Проверка поиска не существующего элемента
10. find\_errorfromempty — Проверка поиска в пустом дереве
11. get\_keys\_6 — Проверка получения массива ключей
12. get\_values\_6 — Проверка получения массива значений
13. clear\_2 — Проверка вставки после очистки дерева
14. clear\_6 — Проверка вставки после очистки дерева



### Пример работы

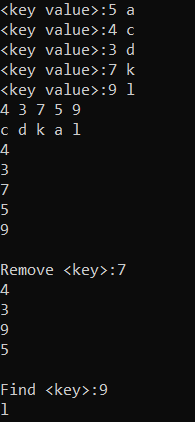


Рис. 1 — Пример работы

Лист. 1 — Программа для демонстрации примера работы

#include "RedBlackTree.h"

#include <iostream>

using namespace std;

int main()

{

RedBlackTree<int,char> tree = RedBlackTree<int, char>();

int inK;

char inV;

for (size\_t i = 0; i < 5; i++)

{

cout << "<key value>:";

cin >> inK >> inV;

tree.insert(inK, inV);

}

int\* keys = tree.get\_keys();

for (size\_t i = 0; i < tree.getSize(); i++)

cout << keys[i] << " ";

cout << endl;

char\* values = tree.get\_values();

for (size\_t i = 0; i < tree.getSize(); i++)

cout << values[i] << " ";

cout << endl;

tree.print();

cout << "Remove <key>:";

cin >> inK;

tree.remove(inK);

tree.print();

cout << "Find <key>:";

cin >> inK;

cout << tree.find(inK);

}

### Листинг

Лист. 2 — Заголовочный файл

#pragma once

#include "Stack.h"

#include <iostream>

using namespace std;

template <typename K, typename V> //type for key and value

class RedBlackTree

{

private:

class Node

{

public:

Node(K key, V value, bool color = 0, Node\* parent = nullptr, Node\* left = nullptr, Node\* right = nullptr) {

this->key = key;

this->value = value;

this->color = color;

this->left = left;

this->right = right;

this->parent = parent;

}

~Node() {};

K key;

V value;

Node\* left;

Node\* right;

Node\* parent;

bool color; // black = 0, red = 1

};

class dft\_Iterator // depth-first traverse

{

private:

Stack<Node\*> stack;

Node\* current;

Node\* nil;

public:

dft\_Iterator(Node\* root = nullptr,Node\* nil=nullptr) {

this->nil = nil;

current = root;

stack.push(current);

}

Node\* next() {

if (!has\_next()) {

throw "No more elements";

}

Node\* temp = current;

if (current->right != nil) //add right tree to stack

{

stack.push(current->right);

}

if (current->left != nil) //go left

{

current = current->left;

}else{ // if can't

current = stack.pop();

}

return temp;

}

bool has\_next() {

if (!stack.isEmpty())

return true;

else

return false;

}

~dft\_Iterator() {};

};

Node\* root;

Node\* nil;

int count;

void coloring(Node\* node) {

if (node == root)

{

node->color = 0;

return;

}

Node\* uncle;

while ((node->parent->color) && (node != root)) //red parent

{

if (node->parent == node->parent->parent->left) //if parent = left child

{

uncle = node->parent->parent->right;

if (uncle->color) //if uncle also red

{

node->parent->color = false; //repaint parent and uncle

uncle->color = false;

node->parent->parent->color = true; //repaint grand parent

node = node->parent->parent; //check grand parent

}

else { //if uncle black

if (node == node->parent->right) { // if node right child

node = node->parent;

LeftRotation(node);

}

node->parent->color = false;

node->parent->parent->color = true; //repaint grand parent

RightRotation(node->parent->parent);

}

}

else { //as in the previous if, but in the other direction

uncle = node->parent->parent->left;

if (uncle->color)

{

node->parent->color = false;

uncle->color = false;

node->parent->parent->color = true;

node = node->parent->parent;

}

else {

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

node = node->parent;

RightRotation(node);

}

node->parent->color = false;

node->parent->parent->color = true;

LeftRotation(node->parent->parent);

}

}

}

root->color = false; // restoration of root color

};

void LeftRotation(Node\* node) {

Node\* child = node->right;//fix right tree

node->right = child->left;

if (child->left != nil) // change parents

child->left->parent = node;

if (child != nil)

child->parent = node->parent;

if (node->parent != nil)

{

if (node == node->parent->left) // change place

node->parent->left = child;

else

node->parent->right = child;

}

else

root = child;

child->left = node;

if (node != nil)

node->parent = child;

};

void RightRotation(Node\* node) {

Node\* child = node->left; //fix left tree

node->left = child->right;

if (child->right != nil) // change parents

child->right->parent = node;

if (child != nil)

child->parent = node->parent;

if (node->parent != nil)

{

if (node == node->parent->right) // change place

node->parent->right = child;

else

node->parent->left = child;

}

else

root = child;

child->right = node;

if (node != nil)

node->parent = child;

};

void removeFix(Node\* node) {

Node\* brother;

while ((node != root) && (node->color == false))

{

if (node == node->parent->left) //if node = left child

{

brother = node->parent->right;

if (brother->color) //if brother red

{

brother->color = false;

node->parent->color = true; //swap parent and brother color

LeftRotation(node->parent);

brother = node->parent->right;

}

if ((brother->left->color == false) && (brother->right->color == false)) //both brother child black

{

brother->color = true;

node = node->parent;

}

else {

if (brother->right->color == false)

{

brother->left->color = false;

brother->color = true;

RightRotation(brother);

brother = node->parent->right;

}

brother->color = node->parent->color;

node->parent->color = false;

brother->right->color = false;

LeftRotation(node->parent);

node = root;

}

}

else { //as in the previous if, but in the other direction

brother = node->parent->left;

if (brother->color)

{

brother->color = false;

node->parent->color = true;

RightRotation(node->parent);

brother = node->parent->left;

}

if ((brother->right->color == false) && (brother->left->color == false))

{

brother->color = true;

node = node->parent;

}

else {

if (brother->left->color == false)

{

brother->right->color = false;

brother->color = true;

LeftRotation(brother);

brother = node->parent->left;

}

brother->color = node->parent->color;

node->parent->color = false;

brother->left->color = false;

RightRotation(node->parent);

node = root;

}

}

}

node->color = false; //root color

};

public:

void insert(K insKey, V insValue) // add element with key and value

{

count++;

if (root == nil) // add root

{

root = new Node(insKey, insValue, false, nil,nil,nil);

}

else {

Node\* newNode = new Node(insKey, insValue, true, nil, nil, nil); // new red element with key and value

Node\* parent = root;

Node\* leaf = nullptr;

while (parent != nil) //looking for a sheet

{

leaf = parent;

if (parent->key < newNode->key)

parent = parent->right;

else

parent = parent->left;

}

newNode->parent = leaf; // add new element

if (leaf->key < newNode->key)

leaf->right = newNode;

else

leaf->left = newNode;

coloring(newNode);

}

};

void remove(K remKey) // remove from key

{

count--;

Node\* node = root;

Node\* nodeA;

Node\* nodeB;

while (node->key != remKey)// find key to remove

{

if ((node == nil)||(node == nullptr))

{

throw out\_of\_range("Element not found");

}

if (node->key < remKey)

node = node->right;

else

node = node->left;

}

if ((node->left == nil) || (node->right == nil))

nodeA = node;

else { //have both child

nodeA = node->right;

while (nodeA->left != nil) nodeA = nodeA->left; //looking for node to swap

}

if (nodeA->left != nil) //fix node

nodeB = nodeA->left;

else

nodeB = nodeA->right;

nodeB->parent = nodeA->parent;

if (nodeA->parent != nil) // if not root

{

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

nodeA->parent->left = nodeB;

else

nodeA->parent->right = nodeB;

}

else {

root = nodeB;

}

if (nodeA != node)

{

node->key = nodeA->key;

node->value = nodeA->value;

}

if (nodeA->color == false) //avoiding red parent

{

removeFix(nodeB);

}

delete nodeA;

};

V find(K finKey) // item search by key

{

Node\* node = root;

while (node->key != finKey)// find key

{

if ((node == nil)||(node == nullptr))

{

throw out\_of\_range("Element not found");

}

if (node->key < finKey)

node = node->right;

else

node = node->left;

}

return node->value;

};

void clear()

{

while ((root != nil)&&(root != nullptr))

remove(root->key);

root = nil;

count = 0;

};

int getSize() {

return count;

};

K\* get\_keys() // returns a list of keys

{

dft\_Iterator i(root, nil);

K\* arr = new K[count];

int arrI = 0;

Node\* cur;

while (i.has\_next())

{

cur = i.next();

arr[arrI] = cur->key;

arrI++;

}

return arr;

};

V\* get\_values() // returns a list of values

{

dft\_Iterator i(root, nil);

V\* arr = new V[count];

int arrI = 0;

Node\* cur;

while (i.has\_next())

{

cur = i.next();

arr[arrI] = cur->value;

arrI++;

}

return arr;

};

void print()

{

dft\_Iterator i(root, nil);

Node\* cur;

while (i.has\_next())

{

cur = i.next();

cout << cur->key <<" " << endl;

}

cout << endl;

};

RedBlackTree() {

nil = new Node(' ', ' ', 0);

this->root = nil;

this->count = 0;

}

~RedBlackTree() {

clear();

delete root;

}

};

#include "Stack.cpp"