**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «АиСД»**

**Тема: Ассоциативный массив**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8302 |  | Зубарева Е.А. |
| Преподаватель |  | Тутуева А.В. |

**Оглавление**

[Постановка задачи 2](#_Toc37951820)

[Описание классов и методов 2](#_Toc37951821)

[Оценка временной сложности 3](#_Toc37951822)

[Описание unit-тестов 3](#_Toc37951823)

[Пример работы 4](#_Toc37951824)

[Код 5](#_Toc37951825)

## Постановка задачи

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева. Наличие unit-тестов ко всем реализуемым методам является ***обязательным*** требованием.

Список методов:

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

## Описание классов и методов

Реализованы следующие классы:

1. List<T> - описывает двусвязный список элементов типа T. Позволяет добавлять элементы в конец списка и удалять элементы, так же реализованы очистка и деструктор. Для обхода спика реализован класс List<T>::Iterator, методы begin(), end().
2. List<T>::Iterator – описывает итератор списка List<T>, позволяет осуществлять доступ к элементу, сдвигаться по списку вперед и назад. Метод добавления элемента в список возвращает итератор, указывающий на добавленный элемент, метод удаления элемента из списка принимает итератор, чтобы удалить элемент, на который он указывает.
3. RbTree<KEY\_T, VALUE\_T> - описывает черно-красное дерево с ключами типа KEY\_T и значениями VALUE\_T, при этом тип KEY\_T должен реализовывать операторы сравнения. Реализует методы добавления, удаления, поиска узла в черно-красном дереве, а так же вывод всех пар ключ-значение, очистку и деструктор.
4. RbMap<KEY\_T, VALUE\_T> - описывает ассоциативный массив (map) с ключами типа KEY\_T и значениями типа VALUE\_T на основе черно-красного дерева. Оборачивает работу с узлами черно-красного дерева в удобный интерфейс с тербуемыми функциями, а так же хранит отдельно списки (List) ключей и значений.

## Оценка временной сложности

Добавление и удаление элементов из списков работают за О(1), а любой обход списка за О(n). Все временные сложности будут рассмотренны на примере методов RbTree, в которых реализованы все алгоритмы работы с черно-красными деревьями.

1. get\_node – поиск узла, проходит от корня бинарного дерева к листу, по скольку дерево сбалансированно, сложность этого алгоритма O(logn)
2. left/right\_rotate – левые и правые повороты, производятся за O(1)
3. insert\_fixup и remove\_fixup – перебалансировка дерева после вставки и удаления элемента соответственно, проходит от узла к корню, попутно совершая левые и правые повороты, работающие за O(1), значит весь алгоритм работает за O(logn)
4. insert, remove – в обоих случаях поиск места вставки/удаляемого узла занимает O(logn), после чего алгоритмы insert/remove\_fixup занимают такое же время. Потому сложность этих алгоритмов так же O(logn).

## Описание unit-тестов

Unit-тесты проверяют сначала только корректность работы insert, добавления и замены элементов, одновременно проверяя совпадение кол-ва узлов в дереве и длин списков ключей и значений.

После чего проверяется метод remove, сначала удаляя один элемент, потом все, проверяя длину.

Далее идет проверка работы со строками. И под конец – получение значения по ключу.

## Пример работы

1 - add element

2 - remove element

other - exit

1

enter key:a

enter value:val1

output: {a: val1, }

debug map tree:

a:val1:black

1 - add element

2 - remove element

other - exit

1

enter key:b

enter value:val2

output: {a: val1, b: val2, }

debug map tree:

a:val1:black

b:val2:red

1 - add element

2 - remove element

other - exit

1

enter key:c

enter value:val3

output: {a: val1, b: val2, c: val3, }

debug map tree:

a:val1:red

b:val2:black

c:val3:red

1 - add element

2 - remove element

other - exit

1

enter key:d

enter value:val4

output: {a: val1, b: val2, c: val3, d: val4, }

debug map tree:

a:val1:black

b:val2:black

c:val3:black

d:val4:red

1 - add element

2 - remove element

other - exit

1

enter key:e

enter value:val5

output: {a: val1, b: val2, c: val3, d: val4, e: val5, }

debug map tree:

a:val1:black

b:val2:black

c:val3:red

d:val4:black

e:val5:red

1 - add element

2 - remove element

other - exit

2

enter key:b

output: {a: val1, c: val3, d: val4, e: val5, }

debug map tree:

a:val1:black

c:val3:red

d:val4:black

e:val5:red

## Код

list.h

#ifndef LAB1\_LIST\_H  
#define LAB1\_LIST\_H  
  
template <typename T>  
class List {  
public:  
 class Node {  
 public:  
 T value;  
 Node\* next = nullptr;  
 Node\* prev = nullptr;  
  
 Node() {};  
 Node(T const& v) : value(v) {}  
 };  
  
 class Iterator {  
 public:  
 Node\* node = nullptr;  
  
 Iterator() {}  
 Iterator(Node\* n) : node(n) {}  
  
 Iterator operator++(int) {  
 Node\* last = node;  
 node = node->next;  
 return Iterator(last);  
 }  
  
 Iterator operator--(int) {  
 Node\* last = node;  
 node = node->prev;  
 return Iterator(last);  
 }  
  
 T& operator\*() {  
 return node->value;  
 }  
  
 bool operator==(Iterator const& it) {  
 return it.node == node;  
 }  
  
 bool operator!=(Iterator const& it) {  
 return it.node != node;  
 }  
 };  
  
private:  
 Node\* first = nullptr;  
 Node\* last = nullptr;  
 Node \_end;  
 int length = 0;  
  
public:  
 Iterator begin() {  
 return Iterator(first);  
 }  
  
 Iterator end() {  
 return Iterator(&\_end);  
 }  
  
 Iterator add(T const& value) {  
 Node\* node = new Node(value);  
 if (last != nullptr) {  
 last->next = node;  
 node->prev = last;  
 } else {  
 first = last = node;  
 }  
 \_end.prev = node;  
 node->next = &\_end;  
 last = node;  
 length++;  
 return Iterator(node);  
 }  
  
 void erase(Iterator iterator) {  
 Node\* node = iterator.node;  
 if (node->next == &\_end) {  
 last = \_end.prev = node->prev;  
 }  
 if (node->prev == nullptr) {  
 first = node->next;  
 } else {  
 node->prev->next = node->next;  
 }  
 node->next->prev = node->prev;  
 length--;  
 delete (node);  
 }  
  
 void print() {  
 std::cout << "[";  
 for (auto it = begin(); it != end(); it++) {  
 std::cout << \*it << " ";  
 }  
 std::cout << "]";  
 }  
  
 void clear() {  
 Node\* node = first;  
 while (node != nullptr && node != &\_end) {  
 Node\* next = node->next;  
 delete(node);  
 node = next;  
 }  
 \_end.next = \_end.prev = nullptr;  
 first = last = nullptr;  
 length = 0;  
 }  
  
 int get\_length() {  
 return length;  
 }  
  
 ~List() {  
 clear();  
 }  
};  
  
#endif //LAB1\_LIST\_H

tree.h

#include <iostream>  
#include "list.h"  
  
  
#ifndef LAB1\_TREE\_H  
#define LAB1\_TREE\_H  
  
template <typename KEY\_T, typename VALUE\_T>  
class RbTree {  
public:  
 enum NodeColor : int {  
 *BLACK* = 0,  
 *RED* = 1  
 };  
  
 typedef typename List<KEY\_T>::Iterator KeyIterator;  
 typedef typename List<VALUE\_T>::Iterator ValueIterator;  
  
 // red-black tree node  
 class Node {  
 public:  
 KEY\_T key;  
 VALUE\_T value;  
 KeyIterator key\_iterator;  
 ValueIterator value\_iterator;  
 NodeColor color = *BLACK*;  
  
 Node\* left = nullptr;  
 Node\* right = nullptr;  
 Node\* parent = nullptr;  
  
 Node(KeyIterator key\_iter, ValueIterator value\_iter) {  
 this->key = \*key\_iter;  
 this->value = \*value\_iter;  
 key\_iterator = key\_iter;  
 value\_iterator = value\_iter;  
 }  
  
 void print() {  
 if (left != nullptr) {  
 left->print();  
 }  
 std::cout << key << ": " << value << ", ";  
 if (right) {  
 right->print();  
 }  
 }  
  
 // output tree with this node as root  
 void debug\_tree(int depth = 0) {  
 // output left child  
 if (left != nullptr) {  
 left->debug\_tree(depth + 1);  
 }  
 // output indentation  
 for (int i = 0; i < depth; i++) {  
 std::cout << " ";  
 }  
 // output info  
 std::cout << key << ":" << value << ":" << (color == *RED* ? "red" : "black") << "\n";  
 // output right child  
 if (right != nullptr) {  
 right->debug\_tree(depth + 1);  
 }  
 }  
  
 int get\_size() {  
 return 1 + (left != nullptr ? left->get\_size() : 0) + (right != nullptr ? right->get\_size() : 0);  
 }  
  
 ~Node() {  
 delete(right);  
 delete(left);  
 }  
 };  
  
 Node\* root = nullptr;  
  
 int get\_size() {  
 return root != nullptr ? root->get\_size() : 0;  
 }  
  
 ~RbTree() {  
 delete(root);  
 }  
  
 void clear() {  
 delete(root);  
 root = nullptr;  
 }  
  
 // output debug tree view  
 void debug\_tree() {  
 if (root != nullptr) {  
 root->debug\_tree();  
 } else {  
 std::cout << "empty tree\n";  
 }  
 }  
  
 // find node by key  
 Node\* get\_node(KEY\_T key) {  
 Node\* node = root;  
 while (node != nullptr) {  
 if (node->key == key) {  
 return node;  
 }  
 if (node->key < key) {  
 node = node->right;  
 } else {  
 node = node->left;  
 }  
 }  
 return nullptr;  
 }  
  
 // left-rotate given node  
 void left\_rotate(Node\* node) {  
 Node\* tmp = node->right;  
 node->right = tmp->left;  
 if (tmp->left != nullptr) {  
 tmp->left->parent = node;  
 }  
 tmp->parent = node->parent;  
  
 if (node->parent == nullptr) {  
 root = tmp;  
 } else {  
 if (node == node->parent->left) {  
 node->parent->left = tmp;  
 } else {  
 node->parent->right = tmp;  
 }  
 }  
 tmp->left = node;  
 node->parent = tmp;  
 }  
  
 // right-rotate given node  
 void right\_rotate(Node\* node) {  
 Node\* tmp = node->left;  
 node->left = tmp->right;  
 if (tmp->right != nullptr) {  
 tmp->right->parent = node;  
 }  
 tmp->parent = node->parent;  
  
 if (node->parent == nullptr) {  
 root = tmp;  
 } else {  
 if (node == node->parent->left) {  
 node->parent->left = tmp;  
 } else {  
 node->parent->right = tmp;  
 }  
 }  
 tmp->right = node;  
 node->parent = tmp;  
 }  
  
 // re-balance tree after insertion  
 void insert\_fixup(Node\* node) {  
 while (node->parent != nullptr && node->parent->color == *RED*) {  
 if (node->parent == node->parent->parent->left) {  
 Node\* tmp = node->parent->parent->right;  
 if (tmp != nullptr && tmp->color == *RED*) {  
 node->parent->color = *BLACK*;  
 tmp->color = *BLACK*;  
 node->parent->parent->color = *RED*;  
 node = node->parent->parent;  
 } else {  
 if (node == node->parent->right) {  
 node = node->parent;  
 left\_rotate(node);  
 }  
 node->parent->color = *BLACK*;  
 node->parent->parent->color = *RED*;  
 right\_rotate(node->parent->parent);  
 }  
 } else {  
 Node\* tmp = node->parent->parent->left;  
 if (tmp != nullptr && tmp->color == *RED*) {  
 node->parent->color = *BLACK*;  
 tmp->color = *BLACK*;  
 node->parent->parent->color = *RED*;  
 node = node->parent->parent;  
 } else {  
 if (node == node->parent->left) {  
 node = node->parent;  
 right\_rotate(node);  
 }  
 node->parent->color = *BLACK*;  
 node->parent->parent->color = *RED*;  
 left\_rotate(node->parent->parent);  
 }  
 }  
 }  
 root->color = *BLACK*;  
 }  
  
 // inserts node into tree and re-balances tree  
 // if node with same key exists, replaces its value and returns false, in this case it must be deleted  
 bool insert(Node\* node) {  
 Node\* last\_node = nullptr;  
 Node\* current\_node = root;  
 while (current\_node != nullptr) {  
 last\_node = current\_node;  
 if (node->key == current\_node->key) {  
 current\_node->value = node->value;  
 return false;  
 }  
 if (node->key < current\_node->key) {  
 current\_node = current\_node->left;  
 } else {  
 current\_node = current\_node->right;  
 }  
 }  
 node->parent = last\_node;  
 if (last\_node == nullptr) {  
 root = node;  
 } else if (node->key < last\_node->key) {  
 last\_node->left = node;  
 } else {  
 last\_node->right = node;  
 }  
 node->left = node->right = nullptr;  
 node->color = *RED*;  
 insert\_fixup(node);  
 return true;  
 }  
  
 // re-balance tree after removal  
 void remove\_fixup(Node\* node) {  
 while (node != root && (node == nullptr || node->color == *BLACK*)) {  
 if (node == node->parent->left) {  
 Node\* w = node->parent->right;  
 if (w != nullptr && w->color == *RED*) {  
 w->color = *BLACK*;  
 node->parent->color = *RED*;  
 left\_rotate(node->parent);  
 w = node->parent->right;  
 }  
 if (w == nullptr) {  
 break;  
 }  
 if ((w->left == nullptr || w->left->color == *BLACK*) &&  
 (w->right == nullptr || w->right->color == *BLACK*)) {  
 w->color = *RED*;  
 node = node->parent;  
 } else {  
 if (w->right == nullptr || w->right->color == *BLACK*) {  
 w->left->color = *BLACK*;  
 w->color = *RED*;  
 right\_rotate(w);  
 w = node->parent->right;  
 }  
 w->color = node->parent->color;  
 node->parent->color = *BLACK*;  
 w->right->color = *BLACK*;  
 left\_rotate(node->parent);  
 node = root;  
 }  
 } else {  
 Node\* w = node->parent->left;  
 if (w != nullptr && w->color == *RED*) {  
 w->color = *BLACK*;  
 node->parent->color = *RED*;  
 right\_rotate(node->parent);  
 w = node->parent->left;  
 }  
 if (w == nullptr) {  
 break;  
 }  
 if ((w->left == nullptr || w->left->color == *BLACK*) &&  
 (w->right == nullptr || w->right->color == *BLACK*)) {  
 w->color = *RED*;  
 node = node->parent;  
 } else {  
 if (w->left == nullptr || w->left->color == *BLACK*) {  
 w->right->color = *BLACK*;  
 w->color = *RED*;  
 left\_rotate(w);  
 w = node->parent->left;  
 }  
 w->color = node->parent->color;  
 node->parent->color = *BLACK*;  
 w->left->color = *BLACK*;  
 right\_rotate(node->parent);  
 node = root;  
 }  
  
 }  
 }  
 }  
  
 Node\* minimal\_node(Node\* node) {  
 while (node->left != nullptr) {  
 node = node->left;  
 }  
 return node;  
 }  
  
 Node\* tree\_successor(Node\* node) {  
 if (node->right != nullptr) {  
 return minimal\_node(node);  
 }  
 Node\* tmp = node->parent;  
 while (tmp != nullptr && node == tmp->right) {  
 node = tmp;  
 tmp = tmp->parent;  
 }  
 return tmp;  
 }  
  
 Node\* remove(Node\* node) {  
 Node\* y;  
 if (node->left == nullptr || node->right == nullptr) {  
 y = node;  
 } else {  
 y = tree\_successor(node);  
 }  
 if (y == nullptr) {  
 debug\_tree();  
 std::cout << " " << node->key << " ";  
 }  
 Node\* x;  
 if (y->left != nullptr) {  
 x = y->left;  
 } else {  
 x = y->right;  
 }  
  
 if (x != nullptr) {  
 x->parent = y->parent;  
 }  
 if (y->parent == nullptr) {  
 root = x;  
 } else {  
 if (y == y->parent->left) {  
 y->parent->left = x;  
 } else {  
 y->parent->right = x;  
 }  
 }  
 if (y != node) {  
 node->key = y->key;  
 node->value = y->value;  
 }  
 if (y->color == *BLACK* && x != nullptr) {  
 remove\_fixup(x);  
 }  
 return y;  
 }  
  
  
};  
  
#endif

rb\_map.h

#include "tree.h"  
#include "list.h"  
  
#ifndef LAB1\_RB\_MAP\_H  
#define LAB1\_RB\_MAP\_H  
  
template <typename KEY\_T, typename VALUE\_T>  
class RbMap {  
public:  
 class invalid\_key\_exception : public std::exception {  
  
 };  
  
private:  
 typedef typename RbTree<KEY\_T, VALUE\_T>::Node Node;  
 RbTree<KEY\_T, VALUE\_T> tree;  
 List<VALUE\_T> key\_list;  
 List<VALUE\_T> value\_list;  
  
public:  
 bool insert(KEY\_T key, VALUE\_T value) {  
 Node\* found = tree.get\_node(key);  
 if (found != nullptr) {  
 found->value = value;  
 \*(found->value\_iterator) = value;  
 return false;  
 } else {  
 Node\* node = new Node(key\_list.add(key), value\_list.add(value));  
 tree.insert(node);  
 return true;  
 }  
 }  
  
 bool remove(KEY\_T key) {  
 Node\* node = tree.get\_node(key);  
 if (node != nullptr) {  
 node = tree.remove(node);  
 key\_list.erase(node->key\_iterator);  
 value\_list.erase(node->value\_iterator);  
 node->right = node->left = node->parent = nullptr;  
 delete(node);  
 return true;  
 }  
 return false;  
 }  
  
 Node\* find(KEY\_T key) {  
 return tree.get\_node(key);  
 }  
  
 bool has(KEY\_T key) {  
 return find(key) != nullptr;  
 }  
  
 VALUE\_T& get(KEY\_T key) {  
 Node\* node = tree.get\_node(key);  
 if (node != nullptr) {  
 return node->value;  
 }  
 throw invalid\_key\_exception();  
 }  
  
 void print() {  
 std::cout << "{";  
 if (tree.root != nullptr) {  
 tree.root->print();  
 }  
 std::cout << "}\n";  
 }  
  
 void debug() {  
 std::cout << "debug map tree:\n";  
 tree.debug\_tree();  
 std::cout << "\n";  
 }  
  
 List<KEY\_T>& get\_keys() {  
 return key\_list;  
 }  
  
 List<VALUE\_T>& get\_values() {  
 return value\_list;  
 }  
  
 int get\_length() {  
 return tree.get\_size();  
 }  
  
 void clear() {  
 tree.clear();  
 key\_list.clear();  
 value\_list.clear();  
 }  
};  
  
#endif //LAB1\_RB\_MAP\_H

main.cpp

#include <map>  
#include <iostream>  
#include "gtest/gtest.h"  
  
#include "rb\_map.h"  
#include "list.h"  
  
int main() {  
 RbMap<std::string, std::string> map;  
  
 int command = 0;  
 std::string key;  
 std::string value;  
 while (true) {  
 std::cout << "\n\n1 - add element\n2 - remove element\nother - exit\n";  
 std::cin >> command;  
 switch(command) {  
 case 1:  
 std::cout << "enter key:";  
 std::cin >> key;  
 std::cout << "enter value:";  
 std::cin >> value;  
 map.insert(key, value);  
 break;  
 case 2:  
 std::cout << "enter key:";  
 std::cin >> key;  
 map.remove(key);  
 break;  
 default:  
 return 0;  
 }  
 std::cout << "output: ";  
 map.print();  
 map.debug();  
 }  
  
 return 0;  
}

tests.cpp

#include "gtest/gtest.h"  
#include "../rb\_map.h"  
  
TEST (Map, InsertionTest) {  
 RbMap<int, int> map;  
 map.insert(1, 2);  
 map.insert(1, 3);  
 map.insert(2, 4);  
 map.insert(3, 10);  
 ASSERT\_EQ(map.get\_length(), 3);  
 ASSERT\_EQ(map.get\_length(), map.get\_keys().get\_length());  
 ASSERT\_EQ(map.get\_length(), map.get\_values().get\_length());  
}  
  
TEST (Map, InsertAndRemoveTest) {  
 RbMap<int, int> map;  
 map.insert(1, 2);  
 map.insert(1, 3);  
 map.insert(2, 4);  
 map.insert(3, 10);  
 map.remove(1);  
 ASSERT\_EQ(map.get\_length(), 2);  
 ASSERT\_EQ(map.get\_length(), map.get\_keys().get\_length());  
 ASSERT\_EQ(map.get\_length(), map.get\_values().get\_length());  
}  
  
TEST(Map, InsertAndRemoveAllTest) {  
 RbMap<int, int> map;  
 map.insert(1, 2);  
 map.insert(1, 3);  
 map.insert(2, 4);  
 map.insert(3, 10);  
 map.remove(1);  
 map.remove(3);  
 map.remove(2);  
 ASSERT\_EQ(map.get\_length(), 0);  
 ASSERT\_EQ(map.get\_length(), map.get\_keys().get\_length());  
 ASSERT\_EQ(map.get\_length(), map.get\_values().get\_length());  
}  
  
TEST(Map, InsertAndClearTest) {  
 RbMap<int, int> map;  
 map.insert(1, 2);  
 map.insert(1, 3);  
 map.insert(2, 4);  
 map.insert(3, 10);  
 map.clear();  
 ASSERT\_EQ(map.get\_length(), 0);  
 ASSERT\_EQ(map.get\_length(), map.get\_keys().get\_length());  
 ASSERT\_EQ(map.get\_length(), map.get\_values().get\_length());  
}  
  
TEST(Map, StringMapTest) {  
 RbMap<std::string, std::string> map;  
 map.insert("a", "value\_b");  
 map.insert("b", "value\_b");  
 map.insert("a", "value\_c");  
 map.insert("c", "value\_a");  
 ASSERT\_EQ(map.get\_length(), 3);  
 ASSERT\_EQ(map.get\_length(), map.get\_keys().get\_length());  
 ASSERT\_EQ(map.get\_length(), map.get\_values().get\_length());  
}  
  
TEST(Map, FindTest) {  
 RbMap<int, int> map;  
 map.insert(1, 2);  
 map.insert(1, 3);  
 map.insert(2, 4);  
 map.insert(3, 10);  
 ASSERT\_EQ(map.get(3), 10);  
}