**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «АиСД»**

**Тема: Ассоциативный массив**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8302 |  | Смирнов Е.Д. |
| 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> - двусвязный список по подобию std::list, основой особенностью которого является наличие итераторов, которые позволяют не только перебирать список, но и ссылаются на элементы даже после изменения списка. Это нужно для реализации списков ключей и значений, чтобы за констатное время добавлять и удалять в них элементы. Списки так же реализуют констуркторы и операторы копирования и присваивания с переносом.
2. rb\_map<K, V> - ассоциативный массив на основе черно-красного дерева, перегружает оператор [], реализуя одновременно методы insert и find, так же отдельно реализует методы find(key), remove(key), get\_keys(), get\_values(), length() – доступ к длине за константное время, tree\_size() – подсчитывает размер дерева за O(N), нужден для тестов, print(), show\_tree(). Ключи и значения представлены списками.
3. rb\_map<K, V>::rb\_tree и rb\_map<K, V>::rb\_tree::rb\_node – реализует черно-красное дерево и основную логику операций над ним. В отличие от rb\_map методы rb\_tree работают в основном с указателями на узлы и используются внутри более высокоуровневых методов rb\_map. Узлы дерева хранят итераторы списков ключей и значений, указывающие на элементы списков для удаления за константное время.

## Оценка временной сложности

1. Удаление и добавление элементов в list происходят за константное время из-за доступа через итераторы.
2. Поиск по ключу в черно-красном дереве происходит за логарифмическое время, т.к. проходит от корня к листу сбалансированного дерева.
3. Правые и левые повороты производятся за константное время
4. Перебаллансировка дерева – проходит по одной ветке (кол-во элементов логарифмическое), производя повороты, работающие за константу, общая сложность – логарифмическая.
5. Вставка и удаление – поиск узла производится за логарифмическое время, перебаллансировка тоже, потому общее время тоже логарифмическое (удаление и добавление в списки происходит за константное время).

## Описание unit-тестов

В первом и втором тестах проверяется базовый функционал ассоциативного массива, добавление по получение элементов, кроме того в нем и в последующих тестах проверяется совпадение размеров списков ключей, значений и черно красного дерева.

В третьем и четвертом тестах проверяется удаление по ключу, удаление лишних элементов, результат, возвращаемый методом удаления.

Потом проверяются поиск и проверка наличия элементов и очистка. После этого идет финальный тест, который сначала заполняет ассоциативный массив большим количеством случайных элементов по случайным ключам, а потом так же, по случайным ключам, производит много операций удаления, после чего сравнивает длины.

## Пример работы

map: {}

type 1 to insert value

type 2 to remove value

type 3 to get value

type 4 to debug red-black tree

type any other number to exit

1

input key:a

input value:b

map: {a: b, }

type 1 to insert value

type 2 to remove value

type 3 to get value

type 4 to debug red-black tree

type any other number to exit

1

input key:b

input value:c

map: {a: b, b: c, }

type 1 to insert value

type 2 to remove value

type 3 to get value

type 4 to debug red-black tree

type any other number to exit

1

input key:d

input value:ee

map: {a: b, b: c, d: ee, }

type 1 to insert value

type 2 to remove value

type 3 to get value

type 4 to debug red-black tree

type any other number to exit

1

input key:b

input value:bb

map: {a: b, b: bb, d: ee, }

type 1 to insert value

type 2 to remove value

type 3 to get value

type 4 to debug red-black tree

type any other number to exit

2

input key:d

value was removed

map: {a: b, b: bb, }

type 1 to insert value

type 2 to remove value

type 3 to get value

type 4 to debug red-black tree

type any other number to exit

4

rb\_map tree:

a:b[R]

b:bb[B]

## Код

main.cpp

#include <iostream>  
  
#include "rb\_map.h"  
  
  
**int** main() {  
 rb\_map<std::string, std::string> map;  
  
 **while** (**true**) {  
 std::cout << "\n\nmap: ";  
 map.print();  
 std::cout << "\n\ntype 1 to insert value\ntype 2 to remove value\ntype 3 to get value\ntype 4 to debug red-black tree\ntype any other number to exit\n";  
  
 **int** choice = 1;  
 std::cin >> choice;  
 **switch** (choice) {  
 **case** 1: {  
 std::string key, value;  
 std::cout << "input key: ";  
 std::cin >> key;  
 std::cout << "input value: ";  
 std::cin >> value;  
 map[key] = value;  
 **break**;  
 }  
 **case** 2: {  
 std::string key, value;  
 std::cout << "input key: ";  
 std::cin >> key;  
 **if** (map.remove(key)) {  
 std::cout << "value was removed\n";  
 } **else** {  
 std::cout << "no such key\n";  
 }  
 **break**;  
 }  
 **case** 3: {  
 std::string key, value;  
 std::cout << "input key: ";  
 std::cin >> key;  
 **if** (map.has(key)) {  
 std::cout << "value at " << key << " = " << map[key] << "\n";  
 } **else** {  
 std::cout << "no such key\n";  
 }  
 **break**;  
 }  
 **case** 4: {  
 map.show\_tree();  
 **break**;  
 }  
 **default**:  
 **return** 0;  
 }  
 }  
}

test.cpp

#include "gtest/gtest.h"  
#include "rb\_map.h"  
  
TEST (rb\_map, fill\_and\_check\_length) {  
 rb\_map<**int**, **int**> map;  
 map[0] = 1;  
 map[10] = 2;  
 map[3] = 3;  
 map[10] = -4;  
 ASSERT\_EQ(map.length(), 3);  
 ASSERT\_EQ(map.tree\_size(), 3);  
 ASSERT\_EQ(map.keys().get\_length(), 3);  
 ASSERT\_EQ(map.values().get\_length(), 3);  
}  
  
TEST (rb\_map, fill\_and\_check\_elems) {  
 rb\_map<**int**, **int**> map;  
 **for** (**int** i = 0; i < 10; i++) {  
 map[i] = i \* 2;  
 }  
 **for** (**int** i = 14; i >= 5; i--) {  
 map[i] = i \* 10;  
 }  
 ASSERT\_EQ(map.length(), 15);  
 ASSERT\_EQ(map.tree\_size(), 15);  
  
 **for** (**int** i = 0; i < 5; i++) {  
 ASSERT\_EQ(map[i], i \* 2);  
 }  
 **for** (**int** i = 14; i >= 5; i--) {  
 ASSERT\_EQ(map[i], i \* 10);  
 }  
}  
  
TEST (rb\_map, remove\_test) {  
 rb\_map<**int**, **int**> map;  
 **for** (**int** i = 0; i < 100; i++) {  
 map[i] = i;  
 }  
 ASSERT\_EQ(map.length(), 100);  
 ASSERT\_EQ(map.tree\_size(), 100);  
 **for** (**int** i = 0; i < 100; i++) {  
 map.remove(i);  
 }  
 ASSERT\_EQ(map.length(), 0);  
 ASSERT\_EQ(map.tree\_size(), 0);  
 **for** (**int** i = 0; i < 200; i++) {  
 map[i] = i;  
 }  
 ASSERT\_EQ(map.length(), 200);  
 ASSERT\_EQ(map.tree\_size(), 200);  
 **for** (**int** i = 199; i >= 0; i--) {  
 map.remove(i);  
 }  
 ASSERT\_EQ(map.length(), 0);  
 ASSERT\_EQ(map.tree\_size(), 0);  
  
}  
  
TEST (rb\_map, excess\_remove\_test) {  
 rb\_map<**int**, **int**> map;  
 **for** (**int** i = 0; i < 100; i++) {  
 map[i] = i;  
 }  
 **for** (**int** i = 50; i < 200; i++) {  
 ASSERT\_EQ(map.remove(i), i < 100);  
 }  
 ASSERT\_EQ(map.length(), 50);  
 ASSERT\_EQ(map.tree\_size(), 50);  
}  
  
TEST (rb\_map, find\_and\_has) {  
 rb\_map<**int**, **int**> map;  
 **for** (**int** i = 0; i < 100; i++) {  
 map[i] = i;  
 }  
 **for** (**int** i = 0; i < 200; i++) {  
 ASSERT\_EQ(map.has(i), i < 100);  
 ASSERT\_EQ(map.find(i) != **nullptr**, i < 100);  
 }  
}  
  
TEST (rb\_map, clear) {  
 rb\_map<**int**, **int**> map;  
 map.clear();  
 ASSERT\_EQ(map.length(), 0);  
 ASSERT\_EQ(map.tree\_size(), 0);  
 ASSERT\_EQ(map.keys().get\_length(), 0);  
  
 **for** (**int** i = 0; i < 200; i++) {  
 map[i] = i;  
 }  
 ASSERT\_EQ(map.length(), 200);  
 ASSERT\_EQ(map.tree\_size(), 200);  
 ASSERT\_EQ(map.keys().get\_length(), 200);  
  
 map.clear();  
 ASSERT\_EQ(map.length(), 0);  
 ASSERT\_EQ(map.tree\_size(), 0);  
 ASSERT\_EQ(map.keys().get\_length(), 0);  
}  
  
TEST (rb\_map, massive\_random\_load) {  
 rb\_map<**int**, **int**> map;  
 **for** (**int** i = 0; i < 1000000; i++) {  
 map[rand() % 1000000] = rand();  
 }  
 ASSERT\_EQ(map.length(), map.tree\_size());  
  
 **int** remaining\_length = map.length();  
 **for** (**int** i = 0; i < 1000000; i++) {  
 **if** (map.remove(rand() % 1000000)) {  
 remaining\_length--;  
 }  
 }  
  
 ASSERT\_EQ(map.length(), map.tree\_size());  
 ASSERT\_EQ(map.length(), remaining\_length);  
}

list.h

#include <string>  
  
  
#ifndef M\_LIST\_H  
#define M\_LIST\_H  
  
**template** <**typename** T>  
**class** list {  
**public**:  
 **class** list\_node {  
 **public**:  
 T value;  
 list\_node\* next = **nullptr**;  
 list\_node\* prev = **nullptr**;  
  
 list\_node() {};  
 list\_node(T **const**& v) : value(v) {}  
 };  
  
 **class** iterator {  
 **public**:  
 list\_node\* node = **nullptr**;  
  
 iterator() {}  
 iterator(list\_node\* n) : node(n) {}  
  
 iterator **operator**++(**int**) {  
 list\_node\* last = node;  
 node = node->next;  
 **return** iterator(last);  
 }  
  
 iterator **operator**--(**int**) {  
 list\_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**:  
 list\_node\* first = **nullptr**;  
 list\_node\* last = **nullptr**;  
 list\_node list\_end;  
 **int** length = 0;  
  
**public**:  
 list() = **default**;  
  
 list(list **const**& other) {  
 **for** (**auto** it = other.begin(); it != other.**end**(); it++) {  
 add(\*it);  
 }  
 }  
  
 list(list&& other) {  
 first = other.first;  
 last = other.last;  
 length = other.length;  
 other.first = other.last = **nullptr**;  
 other.length = 0;  
 }  
  
 list& **operator**= (list **const**& other) {  
 **for** (**auto** it = other.begin(); it != other.**end**(); it++) {  
 add(\*it);  
 }  
 **return** \***this**;  
 }  
  
 list& **operator**= (list&& other) {  
 first = other.first;  
 last = other.last;  
 length = other.length;  
 other.first = other.last = **nullptr**;  
 other.length = 0;  
 **return** \***this**;  
 }  
  
 iterator begin() {  
 **return** first != **nullptr** ? iterator(first) : **end**();  
 }  
  
 iterator **end**() {  
 **return** iterator(&list\_end);  
 }  
  
 iterator add(T **const**& value) {  
 list\_node\* node = **new** list\_node(value);  
 **if** (last != **nullptr**) {  
 last->next = node;  
 node->prev = last;  
 } **else** {  
 first = last = node;  
 }  
 list\_end.prev = node;  
 node->next = &list\_end;  
 last = node;  
 length++;  
 **return** iterator(node);  
 }  
  
 **void** erase(iterator iterator) {  
 list\_node\* node = iterator.node;  
 **if** (node->next == &list\_end) {  
 last = list\_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() {  
 list\_node\* node = first;  
 **while** (node != **nullptr** && node != &list\_end) {  
 list\_node\* next = node->next;  
 **delete**(node);  
 node = next;  
 }  
 list\_end.next = list\_end.prev = **nullptr**;  
 first = last = **nullptr**;  
 length = 0;  
 }  
  
 **int** get\_length() {  
 **return** length;  
 }  
  
 ~list() {  
 clear();  
 }  
};  
  
#endif

rb\_map.h

#include <iostream>  
#include "list.h"  
  
  
#ifndef M\_MAP\_H  
#define M\_MAP\_H  
  
**template** <**typename** K, **typename** V>  
**class** rb\_map {  
**public**:  
 **class** rb\_tree {  
 **public**:  
 **enum** node\_color : **int** {  
 *BLACK* = 0,  
 *RED* = 1  
 };  
  
 // red-black tree node  
 **class** rb\_node {  
 **public**:  
 **typedef typename** list<K>::iterator key\_iter;  
 **typedef typename** list<V>::iterator value\_iter;  
  
 K key;  
 key\_iter key\_iterator;  
 value\_iter value\_iterator;  
 node\_color color = *BLACK*;  
  
 rb\_node\* left = **nullptr**;  
 rb\_node\* right = **nullptr**;  
 rb\_node\* parent = **nullptr**;  
  
 rb\_node(key\_iter key\_iter, value\_iter value\_iter) {  
 **this**->key = \*key\_iter;  
 key\_iterator = key\_iter;  
 value\_iterator = value\_iter;  
 }  
  
 V& **operator**\*() {  
 **return** \*value\_iterator;  
 }  
  
 ~rb\_node() {  
 **delete**(right);  
 **delete**(left);  
 }  
  
 **int** get\_size() {  
 **return** 1 + (left != **nullptr** ? left->get\_size() : 0) + (right != **nullptr** ? right->get\_size() : 0);  
 }  
  
 **void** show\_tree(**int** depth = 0) {  
 **if** (left != **nullptr**) {  
 left->show\_tree(depth + 1);  
 }  
 **for** (**int** i = 0; i < depth; i++) {  
 std::cout << " ";  
 }  
 std::cout << key << ":" << \*value\_iterator << (color == *RED* ? "[R]" : "[B]") << "\n";  
 **if** (right != **nullptr**) {  
 right->show\_tree(depth + 1);  
 }  
 }  
  
 **void** print() {  
 **if** (left != **nullptr**) {  
 left->print();  
 }  
 std::cout << key << ": " << \*value\_iterator << ", ";  
 **if** (right) {  
 right->print();  
 }  
 }  
 };  
  
 rb\_node\* root = **nullptr**;  
  
 **int** get\_size() {  
 **return** root != **nullptr** ? root->get\_size() : 0;  
 }  
  
 ~rb\_tree() {  
 **delete**(root);  
 }  
  
 **void** clear() {  
 **delete**(root);  
 root = **nullptr**;  
 }  
  
 **void** show\_tree() {  
 **if** (root != **nullptr**) {  
 root->show\_tree();  
 } **else** {  
 std::cout << "empty tree\n";  
 }  
 }  
  
 rb\_node\* get\_node(K key) {  
 rb\_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**;  
 }  
  
 **void** left\_rotate(rb\_node\* node) {  
 rb\_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;  
 }  
  
 **void** right\_rotate(rb\_node\* node) {  
 rb\_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;  
 }  
  
 **void** insert\_fixup(rb\_node\* x) {  
 **while** (x->parent != **nullptr** && x->parent->color == *RED*) {  
 **if** (x->parent == x->parent->parent->left) {  
 rb\_node\* y = x->parent->parent->right;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 x->parent->color = *BLACK*;  
 y->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 x = x->parent->parent;  
 } **else** {  
 **if** (x == x->parent->right) {  
 x = x->parent;  
 left\_rotate(x);  
 }  
 x->parent->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 right\_rotate(x->parent->parent);  
 }  
 } **else** {  
 rb\_node\* y = x->parent->parent->left;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 x->parent->color = *BLACK*;  
 y->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 x = x->parent->parent;  
 } **else** {  
 **if** (x == x->parent->left) {  
 x = x->parent;  
 right\_rotate(x);  
 }  
 x->parent->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 left\_rotate(x->parent->parent);  
 }  
 }  
 }  
 root->color = *BLACK*;  
 }  
  
 **bool** insert(rb\_node\* node) {  
 rb\_node\* last\_node = **nullptr**;  
 rb\_node\* current\_node = root;  
 **while** (current\_node != **nullptr**) {  
 last\_node = current\_node;  
 **if** (node->key == current\_node->key) {  
 \*current\_node->value\_iterator = \*node->value\_iterator;  
 **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**;  
 }  
  
 **void** remove\_fixup(rb\_node\* x) {  
 **while** (x != root && (x == **nullptr** || x->color == *BLACK*)) {  
 **if** (x == x->parent->left) {  
 rb\_node\* y = x->parent->right;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 y->color = *BLACK*;  
 x->parent->color = *RED*;  
 left\_rotate(x->parent);  
 y = x->parent->right;  
 }  
 **if** (y == **nullptr**) {  
 **break**;  
 }  
 **if** ((y->left == **nullptr** || y->left->color == *BLACK*) &&  
 (y->right == **nullptr** || y->right->color == *BLACK*)) {  
 y->color = *RED*;  
 x = x->parent;  
 } **else** {  
 **if** (y->right == **nullptr** || y->right->color == *BLACK*) {  
 y->left->color = *BLACK*;  
 y->color = *RED*;  
 right\_rotate(y);  
 y = x->parent->right;  
 }  
 y->color = x->parent->color;  
 x->parent->color = *BLACK*;  
 y->right->color = *BLACK*;  
 left\_rotate(x->parent);  
 x = root;  
 }  
 } **else** {  
 rb\_node\* y = x->parent->left;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 y->color = *BLACK*;  
 x->parent->color = *RED*;  
 right\_rotate(x->parent);  
 y = x->parent->left;  
 }  
 **if** (y == **nullptr**) {  
 **break**;  
 }  
 **if** ((y->left == **nullptr** || y->left->color == *BLACK*) &&  
 (y->right == **nullptr** || y->right->color == *BLACK*)) {  
 y->color = *RED*;  
 x = x->parent;  
 } **else** {  
 **if** (y->left == **nullptr** || y->left->color == *BLACK*) {  
 y->right->color = *BLACK*;  
 y->color = *RED*;  
 left\_rotate(y);  
 y = x->parent->left;  
 }  
 y->color = x->parent->color;  
 x->parent->color = *BLACK*;  
 y->left->color = *BLACK*;  
 right\_rotate(x->parent);  
 x = root;  
 }  
  
 }  
 }  
 }  
  
 rb\_node\* tree\_successor(rb\_node\* node) {  
 **if** (node->right != **nullptr**) {  
 **while** (node->left != **nullptr**) {  
 node = node->left;  
 }  
 **return** node;  
 }  
 rb\_node\* tmp = node->parent;  
 **while** (tmp != **nullptr** && node == tmp->right) {  
 node = tmp;  
 tmp = tmp->parent;  
 }  
 **return** tmp;  
 }  
  
 rb\_node\* remove(rb\_node\* node) {  
 rb\_node\* y;  
 **if** (node->left == **nullptr** || node->right == **nullptr**) {  
 y = node;  
 } **else** {  
 y = tree\_successor(node);  
 }  
 **if** (y == **nullptr**) {  
 show\_tree();  
 std::cout << " " << node->key << " ";  
 }  
 rb\_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\_iterator = \*y->value\_iterator;  
 }  
 **if** (y->color == *BLACK* && x != **nullptr**) {  
 remove\_fixup(x);  
 }  
 **return** y;  
 }  
 };  
  
**public**:  
 **class** invalid\_key\_exception : **public** std::exception {  
  
 };  
  
**private**:  
 rb\_tree tree;  
 list<K> key\_list;  
 list<V> value\_list;  
  
**public**:  
 **typedef typename** rb\_map<K, V>::rb\_tree::rb\_node node\_t;  
  
 V& **operator**[] (K **const**& key) { // insert  
 node\_t\* found = tree.get\_node(key);  
 **if** (found != **nullptr**) {  
 **return** \*(found->value\_iterator);  
 } **else** {  
 node\_t\* node = **new** node\_t(key\_list.add(key), value\_list.add(V()));  
 tree.insert(node);  
 **return** \*(node->value\_iterator);  
 }  
 }  
  
 V **const**& **operator**[] (K **const**& key) **const** { // access  
 node\_t\* node = tree.get\_node(key);  
 **if** (node != **nullptr**) {  
 **return** \*(node->value\_iterator);  
 }  
 **throw** invalid\_key\_exception();  
 }  
  
 **bool** remove(K key) {  
 node\_t\* 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\_t\* find(K key) {  
 **return** tree.get\_node(key);  
 }  
  
 **bool** has(K key) {  
 **return** find(key) != **nullptr**;  
 }  
  
 **void** print() {  
 std::cout << "{";  
 **if** (tree.root != **nullptr**) {  
 tree.root->print();  
 }  
 std::cout << "}\n";  
 }  
  
 **void** show\_tree() {  
 std::cout << "rb\_map tree:\n";  
 tree.show\_tree();  
 std::cout << "\n";  
 }  
  
 list<K>& keys() {  
 **return** key\_list;  
 }  
  
 list<V>& values() {  
 **return** value\_list;  
 }  
  
 **int** length() {  
 **return** value\_list.get\_length();  
 }  
  
 **int** tree\_size() {  
 **return** tree.get\_size();  
 }  
  
 **void** clear() {  
 tree.clear();  
 key\_list.clear();  
 value\_list.clear();  
 }  
};  
  
#endif