**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: «Ассоциативный массив»**

**Вариант 1**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Сиротин П.О. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

Оглавление

[**1.** **Постановка задачи. Описание реализуемого класса и методов. Оценка временной сложности каждого метода** 3](#_Toc73218474)

[**2.** **Описание реализованных unit-тестов**. 4](#_Toc73218475)

[**3.** **Пример работы**. 5](#_Toc73218476)

[**4.** **Листинг**. 5](#_Toc73218477)

[**5.** **Вывод** 17](#_Toc73218478)

1. **Постановка задачи. Описание реализуемого класса и методов. Оценка временной сложности каждого метода**

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

Некоторые классы с реализацией были взяты из лабораторной работы №3 из прошлого семестра (например, класс stack с его вложенным классом st\_node). Много было переписано, много что было добавлено (как минимум, двоичная куча heap из вышеупомянутой ЛР№3 непригодна для реализации ассоциативного массива, поэтому пришлось практически полностью переписать его до вида КЧ-дерева). Были предприняты попытки использовать итераторы для упрощения написания кода (и ухода от рекурсий), но я в них так до конца и не разобрался. Для того, чтобы написать шаблонный ассоциативный массив, необходимо было углубиться в теоретическую часть – как минимум, узнать, что компиляция в С++ раздельная (каждый файл компилируется независимо), поэтому не получалось запустить код, написанный на шаблонах и разделенный в .h и .cpp («теряется» полная информация о шаблонах).

template <typename key\_t, typename value\_t>

Class Node

|  |  |
| --- | --- |
| Элемент | Описание |
| key\_t key; | Ключ узла |
| value\_t value; | Информация, хранящаяся в узле |
| Node\* left; | Указатель на левого потомка данного узла |
| Node\* right; | Указатель на правого потомка данного узла |
| Node\* parent; | Указатель на родителя данного узла |
| ColorName color | Цвет узла (ColorName – это перечисление enum, где BLACK == 0, RED == 1) |

template <typename key\_t, typename value\_t>

Class map

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| Node\* root; | Указатель на корень КЧ-дерева | | |
| Node\* nil; | Пустой лист КЧ-дерева | | |
| size\_t size; | Размер дерева (для тестирования) | | |
| Метод | | Описание | Оценка временной сложности |
| size\_t give\_size() | | Возвращает размер дерева (для тестирования) | О(1) |
| void rightRotate(Node\* curX) | | Правый поворот – вспомогательный алгоритм (для вставки нового элемента в дерево) | О(1) |
| void leftRotate(Node\* curX) | | Левый поворот – вспомогательный алгоритм (для вставки нового элемента в дерево) | О(1) |
| void insertFixup(Node\* to\_insert) | | Восстановление свойств КЧ-дерева после вставки нового элемента | О(logn) |
| void transplant(Node\* x, Node\* y) | | Перемещение поддеревьев | О(1) |
| void removeFixup(Node\* x) | | Восстановление свойств КЧ-дерева после удаления элемента | О(logn) |
| void printHelp(Node\* current) | | Вспомогательный рекурсивный метод (для вывода информации о дереве в консоль) | O(logn) |
| void recursiveGet\_Keys(Node\* current, stack<key\_t>\* keys\_list) | | Вспомогательный рекурсивный метод (для получения списка ключей дерева) | О(logn) |
| void recursiveGet\_Values(Node\* current, stack<value\_t>\* values\_list) | | Вспомогательный рекурсивный метод (для получения списка значений дерева) | O(logn) |
| void insert(key\_t key, value\_t value) | | Вставка нового узла в КЧ-дерево (по ключу и информации) | O(logn) |
| void remove(key\_t key) | | Удаления узла из КЧ-дерева по ключу | O(logn) |
| void clear() | | Очистка (удаление) КЧ-дерева | O(nlogn) |
| stack<key\_t>\* get\_keys() | | Получение списка ключей дерева | O(logn) |
| stack<value\_t>\* get\_values() | | Получение списка значений дерева | O(logn) |
| void print() | | Вывод информации о дереве (о каждом узле, начиная с корня: ключ, значение и цвет) | O(logn) |
| value\_t find(key\_t key) | | Поиск значения по ключу | O(logn) |
| key\_t give\_root\_key() | | Вспомогательный метод для удобного тестирования: возвращает ключ корня дерева | O(1) |
| value\_t give\_root\_value() | | Вспомогательный метод для удобного тестирования: возвращает информацию из корня дерева | O(1) |

template <class T>

Class st\_node

|  |  |
| --- | --- |
| Элемент | Описание |
| T info | Информация внутри узла стека |
| st\_node\* prev | Указатель на предыдущий элемент стека (на элемент «снизу») |
| st\_node\* next | Указатель на следующий элемент стека (на элемент «сверху») |

template <class T>

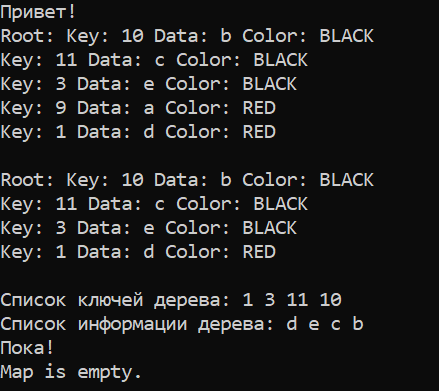
Class stack

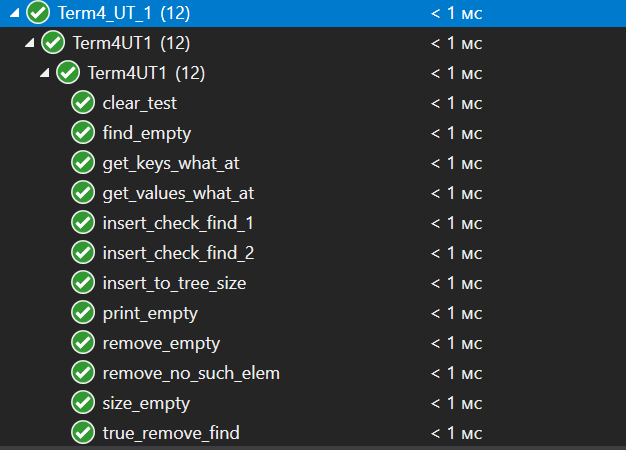
|  |  |  |  |
| --- | --- | --- | --- |
| Элемент | Описание | | |
| st\_node\* top | Указатель на первый элемент стека | | |
| Метод | | Описание | Оценка временной сложности |
| T pop() | | Удаление элемента из стека, возвращает информацию удаленного элемента | О(1) |
| void push(T to\_push) | | Вставка нового элемента в стек | О(1) |
| void print() | | Вывод информации из элементов стека в консоль | О(n) |
| T what\_at(int index) | | Возвращает данные, находящиеся в элементе, имеющим данный индекс в списке | O(n) |

# Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| find\_empty | Проверка метода find, поиск элемента в пустом дереве |
| insert\_check\_find\_1 | Проверка метода insert, поиск вставленного в дерево элемента |
| insert\_check\_find\_2 | Проверка методов insert и find: поиск элемента, которого нет в дереве (в дерево добавили ключи 1, 2 и 3, а ищется 4) |
| insert\_to\_tree\_size | Проверка метода insert, проверка на количество элементов после вставки нового элемента (в пустое дерево добавили элемент – количество [размер дерева] элементов дерева должно стать 1) |
| print\_empty | Проверка метода print, попытка вывести в консоль пустое дерево |
| remove\_empty | Проверка метода remove, попытка удалить элемент из пустого дерева |
| remove\_no\_such\_elem | Проверка метода remove, попытка удалить элемент, которого нет в дереве |
| true\_remove\_find | Проверка методов remove и find, проверка двумя элементами: правда ли удаляется сначала первый узел дерева (проверка, остался ли второй элемент в дереве), а затем второй (проверка методом find на пустое дерево) |
| get\_keys\_what\_at | Проверка метода get\_keys, проверка каждого элемента стека |
| get\_values\_what\_at | Проверка метода get\_values, проверка каждого элемента стека |
| clear\_test | Проверка метода clear, удаляются ли все узлы дерева |
| size\_empty | Проверка вспомогательного метода give\_size пустым деревом |

# Пример работы.





# Листинг.

|  |
| --- |
| Stack.h |
| #pragma once  #include <stdexcept>  #include <iostream>  using namespace std;  template <class T>  class stack  {  private:  class st\_node  {  public:  st\_node(T info, st\_node\* prev = nullptr, st\_node\* next = nullptr)  {  this->info = info;  this->prev = prev;  this->next = next;  }  ~st\_node() {};  T info;  st\_node\* prev;  st\_node\* next;  };  public:  st\_node\* top;  stack()  {  top = nullptr;  }  void push(T to\_push)  {  if (top)  {  st\_node\* buf = new st\_node(to\_push);  buf->prev = top;  top->next = buf;  top = buf;  }  else top = new st\_node(to\_push);  }  T pop()  {  if (!top)  throw out\_of\_range("An error occured.");  if (top->prev == nullptr)  {  T tmp = top->info;  top = nullptr;  delete top;  return tmp;  }  else  {  st\_node node\_temp = top;  T tmp = node\_temp->info;  top = top->prev;  top->next = nullptr;  node\_temp->prev = nullptr;  delete node\_temp;  return tmp;  }  }  void print()  {  st\_node\* current = top;  if (top)  {  while (current->prev)  {  cout << current->info << " ";  current = current->prev;  }  cout << current->info;  cout << endl;  }  else throw out\_of\_range("An error occured: stack is empty.");  }  T what\_at(int index)  {  st\_node\* current = top;  int i = 0;  while (i != index)  {  if (current->prev)  current = current->prev;  else throw "No such element.";  i++;  }  return current->info;  }  ~stack()  {  T tmp;  while (top)  {  tmp = pop();  }  }  }; |
| Map.h |
| #pragma once  //#include "Iterator.h"  #include "Stack.h"  #include <iostream>  #include <stdexcept>  using namespace std;  enum ColorName  {  BLACK,  RED  };  template <typename key\_t, typename value\_t>  class map  {  private:  class Node  {  public:  key\_t key;  value\_t value;  ColorName color;  Node\* left;  Node\* right;  Node\* parent;  Node(key\_t key, value\_t value, ColorName color, Node\* left, Node\* right, Node\* parent)  {  this->key = key;  this->value = value;  this->color = color;  this->left = left;  this->right = right;  this->parent = parent;  }  /\*Node()  {  this->key = nullptr;  this->value = nullptr;  this->color = BLACK;  this->left = nullptr;  this->right = nullptr;  this->parent = nullptr;  }\*/  ~Node() {}  };  private:  Node\* root;  Node\* nil;  size\_t size; // it is necessary to remember size (number of nodes) of tree to make "get\_keys()" and "get\_values()" with iterator correctly  // some helper methods  void rightRotate(Node\* curX)  {  Node\* curY = curX->left; // Y is a left son for X!  curX->left = curY->right;  if (curY->right != nil) // if right son of Y exists  curY->right->parent = curX;  if (curY != nil)  curY->parent = curX->parent; // parent of x is a parent of y  if (curX->parent == nil) // no parent for X  root = curY; // Y is root of tree  else  {  if (curX == curX->parent->left)  curX->parent->left = curY;  else  curX->parent->right = curY;  }  curY->right = curX; // now x is right son of y  if (curX != nil)  curX->parent = curY;  }  void leftRotate(Node\* curX)  {  Node\* curY = curX->right; // Y is a right son for X!  curX->right = curY->left;  if (curY->left != nil) // if left son of Y exists  curY->left->parent = curX;  if (curY != nil)  curY->parent = curX->parent; // parent of x is a parent of y  if (curX->parent == nil) // no parent for X  root = curY; // Y is root of tree  else  {  if (curX == curX->parent->left)  curX->parent->left = curY;  else  curX->parent->right = curY;  }  curY->left = curX; // now x is left son of y  if (curX != nil)  curX->parent = curY;  }  void insertFixup(Node\* to\_insert) // fix the tree after the "insert" operation  {  while ((to\_insert->parent->color == RED)&&(to\_insert != root))  {  if (to\_insert->parent == to\_insert->parent->parent->left)  {  Node\* uncle = to\_insert->parent->parent->right;  if (uncle->color == RED)  {  // CASE I  to\_insert->parent->color = BLACK;  uncle->color = BLACK;  to\_insert->parent->parent->color = RED;  to\_insert = to\_insert->parent->parent;  }  else  {  if (to\_insert == to\_insert->parent->right)  {  // CASE II  to\_insert = to\_insert->parent;  leftRotate(to\_insert);  }  // CASE III  to\_insert->parent->color = BLACK;  to\_insert->parent->parent->color = RED;  rightRotate(to\_insert->parent->parent);  }  }  else  {  Node\* uncle = to\_insert->parent->parent->left;  if (uncle->color == RED)  {  // CASE I  to\_insert->parent->color = BLACK;  uncle->color = BLACK;  to\_insert->parent->parent->color = RED;  to\_insert = to\_insert->parent->parent;  }  else  {  if (to\_insert == to\_insert->parent->left)  {  // CASE II  to\_insert = to\_insert->parent;  rightRotate(to\_insert);  }  // CASE III  to\_insert->parent->color = BLACK;  to\_insert->parent->parent->color = RED;  leftRotate(to\_insert->parent->parent);  }  }  }  root->color = BLACK;  }  // helpers for "remove" method  void transplant(Node\* x, Node\* y) // swap x and y  {  if (x->parent == nil)  root = y;  else  if (x == x->parent->left)  x->parent->left = y;  else x->parent->right = y;  y->parent = x->parent;  }  void removeFixup(Node\* x) // fix the tree after the "remove" operation  {  Node\* w;  while ((x != root) && (x->color == BLACK))  {  if (x == x->parent->left)  {  w = x->parent->right;  if (w->color == RED) // brother of x is RED  {  // CASE I  w->color = BLACK;  x->parent->color = RED;  leftRotate(x->parent);  w = x->parent->right;  }  if ((w->left->color == BLACK) && (w->right->color == BLACK)) // w is BLACK, his sons are BLACK too  {  // CASE II  w->color = RED;  x = x->parent;  }  else  {  // CASE III  if (w->right->color == BLACK) // w, brother of x, is BLACK, left son of w is RED, right - BLACK  {  w->left->color = BLACK;  w->color = RED;  rightRotate(w);  w = x->parent->right;  }  // CASE IV  w->color = x->parent->color; // w, brother of x, is BLACK, his right son is RED  x->parent->color = BLACK;  w->right->color = BLACK;  leftRotate(x->parent);  x = root;  }  }  else  {  w = x->parent->left;  if (w->color == RED)  {  // CASE I  w->color = BLACK;  x->parent->color = RED;  rightRotate(x->parent);  w = x->parent->left;  }  if ((w->right->color == BLACK) && (w->left->color == BLACK))  {  // CASE II  w->color = RED;  x = x->parent;  }  else  {  // CASE III  if (w->left->color == BLACK)  {  w->right->color = BLACK;  w->color = RED;  leftRotate(w);  w = x->parent->left;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->left->color = BLACK;  rightRotate(x->parent);  x = root;  }  }  }  x->color = BLACK;  }  void printHelp(Node\* current) // recursive helper method  { // can be done with iterators or with recursion  //dftIterator iter(root, nil); // iterator starts from root!  //Node\* current;  string color;  /\*while (iter.has\_next())  {  current = iter.next();  if (current->color)  color = "RED";  else color = "BLACK";  cout << "Key: " << current->key << " Data: " << current->value << " Color: " << color << endl;  }\*/  if (current->color)  color = "RED";  else color = "BLACK";  if (current != nil)  {  cout << "Key: " << current->key << " Data: " << current->value << " Color: " << color << endl;  if (current->right != nil)  printHelp(current->right);  if (current->left != nil)  printHelp(current->left);  }  else return; // there's no node  }  void recursiveGet\_Keys(Node\* current, stack<key\_t>\* keys\_list) // recursive helper method (get\_keys)  {  if (current == nil)  return; // end of branch  else  {  keys\_list->push(current->key);  if (current->right != nil)  recursiveGet\_Keys(current->right, keys\_list);  }  if (current->left != nil)  recursiveGet\_Keys(current->left, keys\_list);  }  void recursiveGet\_Values(Node\* current, stack<value\_t>\* values\_list) // recursive helper method (get\_values)  {  if (current == nil)  return; // end of branch  else  {  values\_list->push(current->value);  if (current->right != nil)  recursiveGet\_Values(current->right, values\_list);  }  if (current->left != nil)  recursiveGet\_Values(current->left, values\_list);  }  // iterator will help us to print information from tree and make keys and values lists of a tree  //class dftIterator : public Iterator<Node\*> // what a nice throwback to the Lab3 from previous term!  //{  //public:  // dftIterator(Node\* start, Node\* null\_nil)  // {  // cur\_map = start;  // //current.push(start);  // it\_nil = null\_nil;  // }  // Node\* next()  // {  // Node\* tmp = cur\_map;  // if (tmp->right != it\_nil)  // current.push(cur\_map->right);  // if (tmp->left != it\_nil)  // cur\_map = cur\_map->left;  // else  // if (current.top != nullptr)  // cur\_map = current.pop();  // else cur\_map = it\_nil;  // return tmp;  // }  // bool has\_next()  // {  // return (cur\_map != it\_nil);  // }  // ~dftIterator() {}  //private:  // stack<Node\*> current; // stack is built with a pointer on Node from map class as a data  // Node\* cur\_map;  // Node\* it\_nil;  //};  public:  map()  {  this->nil = new Node(0, 0, BLACK, nullptr, nullptr, nullptr);  this->root = nil;  this->size = 0;  }  ~map()  {  clear();  }  void insert(key\_t key, value\_t value) // inserts a node to the map  {  // new node is ALWAYS RED  // in some cases it will become BLACK  Node\* y = nil;  Node\* x = root;  while (x != nil)  {  y = x;  if (key < x->key)  x = x->left;  else x = x->right;  }  Node\* to\_insert = new Node(key, value, RED, nil, nil, y);  if (y == nil)  {  root = to\_insert;  }  else  if (to\_insert->key < y->key)  y->left = to\_insert;  else  y->right = to\_insert;  size++;  insertFixup(to\_insert);  /\*Node\* x\_node = root;  Node\* x\_parent = x\_node;  if (x\_node == nil) // RB-tree is empty  {  root = new Node(key, value, BLACK, nil, nil, nil); // creating a root  return;  }  while (x\_node != nil) // moving down to the leafs  {  x\_parent = x\_node; // it is necessary to save the pointer on parent to end insertion algorythm  if (key > x\_node->key)  {  x\_node = x\_node->right; // right son is "bigger" than his parent  }  else  {  x\_node = x\_node->left; // left son is "smaller" than his parent  }  if (key > x\_parent)  {  x\_parent->right = new Node(key, value, RED, nil, nil, x\_parent); // right son is "bigger" than his parent  recover(x\_parent->right);  }  else  {  x\_parent->left = new Node(key, value, RED, nil, nil, x\_parent); // left son is "smaller" than his parent  recover(x\_parent->left);  }  }\*/  }  void remove(key\_t key) // removes a node from map by key  {  if (root == nil)  throw "Nothing to remove: tree is empty.";  Node\* toBeDeleted = root;  Node\* x, \* min;  while (toBeDeleted->key != key)  {  if (key > toBeDeleted->key)  toBeDeleted = toBeDeleted->right;  else  toBeDeleted = toBeDeleted->left;  if (toBeDeleted == nil)  throw "There's no node in RB-Tree with that key.";  }  ColorName originalColor = toBeDeleted->color;  if (toBeDeleted->left == nil)  {  x = toBeDeleted->right;  transplant(toBeDeleted, toBeDeleted->right);  }  else  if (toBeDeleted->right == nil)  {  x = toBeDeleted->left;  transplant(toBeDeleted, toBeDeleted->left);  }  else  {  min = toBeDeleted->right;  while (min->left != nil)  min = min->left;  originalColor = min->color;  x = min->right;  if (min->parent == toBeDeleted)  x->parent = min;  else  {  transplant(min, min->right);  min->right = toBeDeleted->right;  min->right->parent = min;  }  transplant(toBeDeleted, min);  min->left = toBeDeleted->left;  min->left->parent = min;  min->color = toBeDeleted->color;  }  if (originalColor == BLACK)  removeFixup(x);  size--;  }  void clear() // clears map  {  while (root != nil)  remove(root->key);  cout << "Map is empty." << endl;  }  stack<key\_t>\* get\_keys() // returns array of keys (need to use size and iterator)  {  if (root == nil)  throw "Nothing to get: map is empty.";  //dftIterator iter(root, nil); // iterator starts from root!  //Node\* current;  //key\_t\* out\_list = new key\_t[size];  //int i = 0;  /\*while (iter.has\_next())  {  current = iter.next();  out\_list[i] = current->key;  i++;  }\*/  stack<key\_t>\* out\_list = new stack<key\_t>();  recursiveGet\_Keys(root, out\_list);  return out\_list;  }  stack<value\_t>\* get\_values() // returns array of values (need to use size and iterator)  {  // i could do that with iterators but i suck on the topic of iteratos :c  if (root == nil)  throw "Nothing to get: map is empty.";  //dftIterator iter(root, nil); // iterator starts from root!  //value\_t\* out\_list = new value\_t[size];  //int i = 0;  /\*while (iter.has\_next())  {  current = iter.next();  out\_list[i] = current->value;  i++;  }\*/  stack<value\_t>\* out\_list = new stack<value\_t>();  recursiveGet\_Values(root, out\_list);  return out\_list;  }  void print()  {  if (root == nil)  throw "Nothing to print: map is empty.";  else  {  cout << "Root: ";  printHelp(root);  cout << endl;  }  }  value\_t find(key\_t key) // finds an element by key and returns value from this element  {  if (root == this->nil)  throw "Nothing to find: tree is empty.";  else  {  Node\* toFind = root;  while (toFind->key != key)  {  if (key > toFind->key)  toFind = toFind->right;  else  toFind = toFind->left;  if (toFind == this->nil)  throw "There's no node in RB-Tree with that key.";  }  return toFind->value;  }  }  size\_t give\_size() // incapsulation: method just gives us size (needs for get\_keys and get\_values made with iterator)  {  return size;  }  }; |
| UnitTest1.cpp |
| #include "CppUnitTest.h"  #include <stdexcept>  #include "../Term4\_LabWork1/Stack.h"  #include "../Term4\_LabWork1/Map.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace Term4UT1  {  TEST\_CLASS(Term4UT1)  {  public:  map<int, char>\* RBtree;  TEST\_METHOD\_INITIALIZE(SetUp)  {  RBtree = new map<int, char>();  }  TEST\_METHOD\_CLEANUP(cleanUp)  {  delete RBtree;  }  TEST\_METHOD(insert\_check\_find\_1)  {  RBtree->insert(1, 'a');  RBtree->insert(2, 'b');  RBtree->insert(3, 'c');  Assert::AreEqual(RBtree->find(3), 'c');  }  TEST\_METHOD(insert\_check\_find\_2)  {  try  {  RBtree->insert(1, 'a');  RBtree->insert(2, 'b');  RBtree->insert(3, 'c');  char symbol = RBtree->find(4);  }  catch (const char\* error)  {  Assert::AreEqual(error, "There's no node in RB-Tree with that key.");  }  }  TEST\_METHOD(insert\_to\_tree\_size)  {  RBtree->insert(1, 'a');  Assert::AreEqual((size\_t)1, RBtree->give\_size());  }  TEST\_METHOD(find\_empty)  {  try  {  RBtree->insert(5, 'c');  }  catch (const char\* error)  {  Assert::AreEqual(error, "Nothing to find: tree is empty.");  }  }  TEST\_METHOD(print\_empty)  {  try  {  RBtree->print();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Nothing to print: map is empty.");  }  }  TEST\_METHOD(remove\_empty)  {  try  {  RBtree->remove(1);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Nothing to remove: tree is empty.");  }  }  TEST\_METHOD(remove\_no\_such\_elem)  {  try  {  RBtree->insert(2, 'c');  RBtree->remove(1);  }  catch (const char\* error)  {  Assert::AreEqual(error, "There's no node in RB-Tree with that key.");  }  }  TEST\_METHOD(true\_remove\_find)  {  try  {  RBtree->insert(2, 'c');  RBtree->insert(1, 'b');  RBtree->remove(1);  char c = RBtree->find(2);  Assert::AreEqual(c, 'c');  RBtree->remove(2);  c = RBtree->find(1);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Nothing to find: tree is empty.");  }  }  TEST\_METHOD(get\_keys\_what\_at)  {  stack<int>\* keys\_list;  RBtree->insert(1, 'a');  RBtree->insert(2, 'b');  RBtree->insert(3, 'c');  keys\_list = RBtree->get\_keys();  Assert::IsTrue((keys\_list->what\_at(0) == 1) && (keys\_list->what\_at(1) == 3) && (keys\_list->what\_at(2) == 2));  }  TEST\_METHOD(get\_values\_what\_at)  {  stack<char>\* values\_list;  RBtree->insert(1, 'a');  RBtree->insert(2, 'b');  RBtree->insert(3, 'c');  values\_list = RBtree->get\_values();  Assert::IsTrue((values\_list->what\_at(0) == 'a') && (values\_list->what\_at(1) == 'c') && (values\_list->what\_at(2) == 'b'));  }  TEST\_METHOD(clear\_test)  {  RBtree->insert(1, 'a');  RBtree->insert(2, 'b');  RBtree->insert(3, 'c');  RBtree->clear();  Assert::AreEqual((size\_t)0, RBtree->give\_size());  }  TEST\_METHOD(size\_empty)  {  Assert::AreEqual((size\_t)0, RBtree->give\_size());  }  };  } |

# Вывод

Я изучил и научился реализовывать ассоциативный массив map с помощью красно-чёрного дерева и работать с ним. Для успешного выполнения работы я изучил материал по использованию шаблонов в С++.