**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Ассоциативный массив»**

**Вариант 1.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Новокрещенов Д.К. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

## Постановка задачи и описание реализуемого класса и методов.

Для решения поставленной задачи необходимо реализовать ассоциативный шаблонный массив на основе красно-черного дерева.

Для этого мне понадобились класс «Node» и класс «RedBlackTree».

**enum Color {RED, BLACK};**

class Node

|  |  |
| --- | --- |
| **Компонент** | **Описание** |
| Key\_Type Key; | Ключ элемента |
| Value\_Type Value; | Данные элемента |
| Node\* parent; | Ссылка на родительский элемент |
| Node\* left; | Ссылка на левого потомка |
| Node\* right; | Ссылка на правого потомка |
| Color COLOR; | Цвет узла |

**class RedBlackTree**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Описание** | |
| Node\* root; | Указатель на корень дерева | |
| Node\* nil; | Указатель на нулевой узел | |
| size\_t size; | Размер дерева (количество элементов в дереве) | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void recovery(Node\*); | Восстановление свойств после вставки элемента | O(1) |
| void leftRotate(Node\*); | Левый поворот | O(1) |
| void rightRotate(Node\*); | Правый поворот | O(1) |
| void recoveryRemove(Node\*); | Восстановление свойства после удаления элемента | O(1) |
| void iteration\_over(Node\* actual, BiList<Key\_Type>\* keys, BiList<Value\_Type>\* values); | Перебор всех элементов и вставка их в список ключей или значений | O(n) |
| void insert(Key\_Type, Value\_Type); | Вставка элемента | O(logn) |
| void remove(Key\_Type); | Удаление элемента по ключу | O(logn) |
| Value\_Type find(Key\_Type); | Поиск значения элемента по ключу | O(logn) |
| void clear(); | Очистка дерева | O(n) |
| BiList<Key\_Type>\* get\_keys(); | Получение списка ключей | O(n) |
| BiList<Value\_Type>\* get\_values(); | Получение списка значений | O(n) |
| void print(); | Вывод ассоциативного массива в консоль | O(n) |
| size\_t get\_size(); | Получение размера | O(n) |

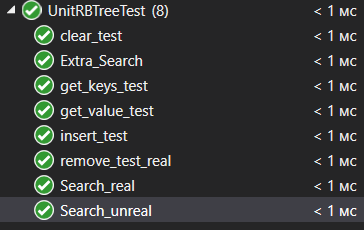
## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| **Наименование теста** | **Описание** |
| clear\_test | Тест метода clear() |
| Extra\_Search | Тест поиска для всех элементов |
| get\_keys\_test | Тест метода get\_keys() |
| get\_values\_test | Тест метода get\_values() |
| insert\_test | Тест метода insert() |
| remove\_test\_real | Тест метода remove() |
| Search\_real | Тест метода find() для существующего элемента |
| Search\_unreal | Тест метода find() для несуществующего элемента |

## Листинги программы и тестов

|  |
| --- |
| List.h |
| #pragma once  #include <cstddef>  template<class Type>  class BiList  {  public:  BiList();  ~BiList();  void push\_back(Type);  void push\_front(Type);  void pop\_back();  void pop\_front();  void insert(Type, size\_t);  Type at(size\_t);  void remove(size\_t);  size\_t get\_size();  void print\_to\_console();  void clear();  void set(size\_t, Type);  bool isEmpty();  void reverse();  private:  struct Node  {  Type data;  Node\* prev;  Node\* next;  };  Node\* head;  Node\* last;  size\_t size;  }; |
| List.cpp |
| #pragma once  #include "List.h"  #include <iostream>  template<class Type>  BiList<Type>::BiList()  {  head = NULL;  last = NULL;  size = 0;  }  template<class Type>  BiList<Type>::~BiList()  {  while (head)  {  last = head->next;  delete head;  head = last;  }  }  //The function inserts a new node at back of the list.  template<class Type>  void BiList<Type>::push\_back(Type value)  {  Node\* temp = new Node;  temp->next = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->prev = last;  last->next = temp;  last = temp;  }  else {  temp->prev = NULL;  head = last = temp;  }  size++;  }  //The function inserts a new node at front of the list.  template<class Type>  void BiList<Type>::push\_front(Type value)  {  Node\* temp = new Node;  temp->prev = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->next = head;  head->prev = temp;  head = temp;  }  else  {  temp->prev = NULL;  temp->next = NULL;  head = last = temp;  }  size++;  }  //The function deletes the last node in the list.  template<class Type>  void BiList<Type>::pop\_back()  {  if(size==0)  return;  if (size == 1)  {  delete last;  last = head = NULL;  size--;  return;  }  delete last->next;  last = last->prev;  last->next = NULL;  size--;  }  //The function deletes the first node in the list.  template<class Type>  void BiList<Type>::pop\_front()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = last = NULL;  size--;  return;  }  delete head->prev;  head->prev = NULL;  head = head->next;  size--;  }  //The function inserts a new node after the node with the number "index".  template<class Type>  void BiList<Type>::insert(Type value, size\_t index)  {  if ((size == 0 && index==0) || index >= size)  throw "Incorrect index.";  if (index == 0)  {  this->push\_front(value);  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = new Node;  temp->data = value;  temp->next = cursor;  temp->prev = cursor->prev;  cursor->prev = temp;  temp->prev->next = temp;  size++;  }  //The function gets the value from the node with the number "index".  template<class Type>  Type BiList<Type>::at(size\_t index)  {  if (index >= size || index<0)  throw "Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  return cursor->data;  }  //The function deletes the node with the "index" number.  template<class Type>  void BiList<Type>::remove(size\_t index)  {  if (index >= size || index <= 0)  throw "Incorrect index.";  if (index == 0)  {  this->pop\_front();  return;  }  if (index == size-1)  {  this->pop\_back();  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = cursor;  cursor->prev->next = cursor->next;  cursor->next->prev = cursor->prev;  delete temp;  size--;  }  //The function gets the number of items in the list  template<class Type>  size\_t BiList<Type>::get\_size()  {  size\_t size\_of\_list = 0;  Node\* cursor = head;  if (cursor == NULL)  return 0;  while (cursor)  {  size\_of\_list++;  cursor = cursor->next;  }  return size\_of\_list;  }  //The function prTypes all list in console.  template<class Type>  void BiList<Type>::print\_to\_console()  {  Node\* cursor = head;  if (head)  {  while (cursor->next)  {  std::cout << cursor->data << "<-->";  cursor = cursor->next;  }  std::cout << cursor->data;  }  else throw "List is empty.";  }  //The function deletes all node in list.  template<class Type>  void BiList<Type>::clear()  {  while (head!=NULL)  this->pop\_front();  head = last = NULL;  }  //The function sets a new value in the node with the number "index"  template<class Type>  void BiList<Type>::set(size\_t index, Type value)  {  if (index >= size || index<0)  throw"Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  cursor->data = value;  }  //The function checks the list for emptiness.  template<class Type>  bool BiList<Type>::isEmpty()  {  if (head==NULL) return true;  return false;  }  //The function reverses a list.  template<class Type>  void BiList<Type>::reverse()  {  if (!head || !head->next)  return;  last = head;  Node\* temp = NULL;  Node\* current = head;  while (current != NULL)  {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  head = temp->prev;  } |
| RBTree.h |
| #pragma once  #include "List.h"  #include "List.cpp"  #define NULL 0  enum Color {RED, BLACK};  template<class Key\_Type, class Value\_Type>  class RedBlackTree  {  private:  class Node  {  public:  Key\_Type Key;  Value\_Type Value;  Node\* parent;  Node\* left;  Node\* right;  Color COLOR;  Node(Key\_Type Key, Value\_Type Value, Node\* parent = NULL, Node\* left = NULL, Node\* right = NULL, Color COLOR = RED);  ~Node();  };  Node\* root;  Node\* nil;  size\_t size;  void recovery(Node\*);  void leftRotate(Node\*);  void rightRotate(Node\*);  void recoveryRemove(Node\*);    void iteration\_over(Node\* actual, BiList<Key\_Type>\* keys, BiList<Value\_Type>\* values);  public:  RedBlackTree();  ~RedBlackTree();  void insert(Key\_Type, Value\_Type);  void remove(Key\_Type);  Value\_Type find(Key\_Type);  void clear();  BiList<Key\_Type>\* get\_keys();  BiList<Value\_Type>\* get\_values();  void print();  size\_t get\_size();  }; |
| RBTree.cpp |
| #pragma once  #include <iostream>  #include "RBTree.h"  #include "List.h"  using namespace std;  //конструктор для узла дерева  template<class Key\_Type, class Value\_Type>  RedBlackTree<Key\_Type, Value\_Type>::Node::Node(Key\_Type Key, Value\_Type Value, Node\* parent, Node\* left, Node\* right, Color COLOR)  {  this->Key = Key;  this->Value = Value;  this->left = left;  this->right = right;  this->parent = parent;  this->COLOR = COLOR;  }  //деструктор узла дерева  template<class Key\_Type, class Value\_Type>  RedBlackTree<Key\_Type, Value\_Type>::Node::~Node()  {  }  //восстановление после вставки  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::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) //случай 1  {  grandParent->right->COLOR = BLACK;  grandParent->left->COLOR = BLACK;  grandParent->COLOR = RED;  new\_Node = grandParent;  }  else  {  if (parent->right == new\_Node) //случай 2  {  new\_Node = parent;  leftRotate(new\_Node);  }  //случай 3  new\_Node->parent->COLOR = BLACK;  new\_Node->parent->parent->COLOR = RED;  rightRotate(new\_Node->parent->parent);  }  }  else  {  if (grandParent->left && grandParent->left->COLOR == RED) //случай 1  {  grandParent->right->COLOR = BLACK;  grandParent->left->COLOR = BLACK;  grandParent->COLOR = RED;  new\_Node = grandParent;  }  else  {  if (parent->left == new\_Node) //случай 2  {  new\_Node = parent;  rightRotate(new\_Node);  }  //случай 3  new\_Node->parent->COLOR = BLACK;  new\_Node->parent->parent->COLOR = RED;  leftRotate(new\_Node->parent->parent);  }  }  }  root->COLOR = BLACK;  }  //левый поворот  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::leftRotate(Node\* actual)  {  if (actual->right == nil) return;  Node\* temp = actual->right;  if (temp->left != nil)  {  actual->right = temp->left;  temp->left->parent = actual;  }  else  actual->right = nil;  if (temp != nil)  temp->parent = actual->parent;  if (actual->parent != nil)  {  if (actual == actual->parent->left)  actual->parent->left = temp;  else  actual->parent->right = temp;  }  else  {  temp->parent = nil;  root = temp;  }  temp->left = actual;  if (actual != nil)  actual->parent = temp;  }  //правый поворот  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::rightRotate(Node\* actual)  {  Node\* temp = actual->left;  actual->left = temp->right;  if (temp->right != nil)  temp->right->parent = actual;  if (temp != nil)  temp->parent = actual->parent;  if (actual->parent != nil)  {  if (actual == actual->parent->right)  actual->parent->right = temp;  else  actual->parent->left = temp;  }  else  {  root = temp;  }  temp->right = actual;  if (actual != nil) actual->parent = temp;  }  //восстановление после удаления элемента  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::recoveryRemove(Node\* x\_Node)  {  Node\* brother;  while (x\_Node != root && x\_Node->COLOR == BLACK)  {  if (x\_Node == x\_Node->parent->left)  {  brother = x\_Node->parent->right;  if (brother->COLOR == RED)  {  brother->COLOR = BLACK;  x\_Node->parent->COLOR = RED;  leftRotate(x\_Node->parent);  brother = x\_Node->parent->right;  }  if (brother->left->COLOR == BLACK && brother->right->COLOR == BLACK)  {  brother->COLOR = RED;  x\_Node = x\_Node->parent;  }  else  {  if (brother->right->COLOR == BLACK)  {  brother->left->COLOR = BLACK;  brother->COLOR = RED;  rightRotate(brother);  brother = x\_Node->parent->right;  }  brother->COLOR = x\_Node->parent->COLOR;  x\_Node->parent->COLOR = BLACK;  brother->right->COLOR = BLACK;  leftRotate(x\_Node->parent);  x\_Node = root;  }  }  else  {  brother = x\_Node->parent->left;  if (brother->COLOR == RED)  {  brother->COLOR = BLACK;  x\_Node->parent->COLOR = RED;  rightRotate(x\_Node->parent);  brother = x\_Node->parent->left;  }  if (brother->right->COLOR == BLACK && brother->left->COLOR == BLACK)  {  brother->COLOR = RED;  x\_Node = x\_Node->parent;  }  else  {  if (brother->left->COLOR == BLACK)  {  brother->right->COLOR = BLACK;  brother->COLOR = RED;  leftRotate(brother);  brother = x\_Node->parent->left;  }  brother->COLOR = x\_Node->parent->COLOR;  x\_Node->parent->COLOR = BLACK;  brother->left->COLOR = BLACK;  rightRotate(x\_Node->parent);  x\_Node = root;  }  }  }  x\_Node->COLOR = BLACK;  }  //конструктор для пустого дерева  template<class Key\_Type, class Value\_Type>  RedBlackTree<Key\_Type, Value\_Type>::RedBlackTree()  {  if(typeid(Value\_Type).name() == typeid(string).name())  nil = new Node(0, " ", NULL, NULL, NULL, BLACK);  else  nil = new Node(0, 0, NULL, NULL, NULL, BLACK);  root = nil;  size = 0;  }  //деструктор для дерева  template<class Key\_Type, class Value\_Type>  RedBlackTree<Key\_Type, Value\_Type>::~RedBlackTree()  {  clear();  }  //вставка нового узла  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::insert(Key\_Type Key, Value\_Type Value)  {  Node\* new\_Node = new Node(Key, Value, nil, nil, nil);  if (size == 0)  {  root = new\_Node;  root->COLOR = BLACK;  size++;  return;  }  Node\* temp = root;  new\_Node->COLOR = RED;  while (1)  {  if (Key > temp->Key)  {  if (temp->right == nil)  {  temp->right = new\_Node;  new\_Node->parent = temp;  break;  }  temp = temp->right;  }  else  {  if (temp->left == nil)  {  temp->left = new\_Node;  new\_Node->parent = temp;  break;  }  temp = temp->left;  }  }  recovery(new\_Node);  size++;  }  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::remove(Key\_Type Key)  {  if (root == nil)  throw "Tree is empty";  Node\* NodeToDelete = root;  while (NodeToDelete->Key != Key) //поиск элемента  {  if (Key > NodeToDelete->Key)  NodeToDelete = NodeToDelete->right;  else  NodeToDelete = NodeToDelete->left;  if (NodeToDelete == nil)  throw "No such element";  }  Node\* x\_Node, \*w\_Node;  Color origCOLOR = NodeToDelete->COLOR;  if (NodeToDelete->left == nil)  {  x\_Node = NodeToDelete->right;  if (NodeToDelete->parent == nil)  root = x\_Node;  else if (NodeToDelete->parent->left == NodeToDelete)  NodeToDelete->parent->left = x\_Node;  else  NodeToDelete->parent->right = x\_Node;  x\_Node->parent = NodeToDelete->parent;  }  else if (NodeToDelete->right == nil)  {  x\_Node = NodeToDelete->left;  if (NodeToDelete->parent == nil)  root = x\_Node;  else if (NodeToDelete->parent->left == NodeToDelete)  NodeToDelete->parent->left = x\_Node;  else  NodeToDelete->parent->right = x\_Node;  x\_Node->parent = NodeToDelete->parent;  }  else  {  Node\* temp = NodeToDelete;  w\_Node = NodeToDelete->right;  while (w\_Node->left != nil)  w\_Node = w\_Node->left;  origCOLOR = w\_Node->COLOR;  NodeToDelete = w\_Node;  x\_Node = NodeToDelete->right;  if (NodeToDelete->parent == temp)  x\_Node->parent = NodeToDelete;  else  {  if (NodeToDelete->parent == nil)  {  root = x\_Node;  NodeToDelete->right->parent = nil;  }  else if (NodeToDelete->parent->right == NodeToDelete)  NodeToDelete->parent->right = x\_Node;  else  NodeToDelete->parent->left = x\_Node;  x\_Node->parent = NodeToDelete->parent;  NodeToDelete->right = temp->right;  NodeToDelete->right->parent = NodeToDelete;  }  if (temp->parent == nil)  root = NodeToDelete;  else if (temp == temp->parent->left)  temp->parent->left = NodeToDelete;  else  temp->parent->right = NodeToDelete;  NodeToDelete->parent = temp->parent;  NodeToDelete->left = temp->left;  NodeToDelete->left->parent = NodeToDelete;  NodeToDelete->COLOR = temp->COLOR;  }  if (origCOLOR == BLACK)  recoveryRemove(x\_Node);  size--;  }  template<class Key\_Type, class Value\_Type>  Value\_Type RedBlackTree<Key\_Type, Value\_Type>::find(Key\_Type Key)  {  if (root == nil)  throw "Tree is empty";  Node\* nodeToFind = root;  while (nodeToFind->Key != Key) //поиск элемента  {  if (Key > nodeToFind->Key)  nodeToFind = nodeToFind->right;  else  nodeToFind = nodeToFind->left;  if (nodeToFind == nil)  throw "No such element";  }  return nodeToFind->Value;  }  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::iteration\_over(Node\* actual, BiList<Key\_Type> \*keys, BiList<Value\_Type> \*values)  {  if (keys == NULL)  {  if (actual->left!=nil)  iteration\_over(actual->left, NULL, values);  if (actual->right!=nil)  iteration\_over(actual->right, NULL, values);  values->push\_back(actual->Value);  }  else  {  if (actual->left!=nil)  iteration\_over(actual->left, keys, NULL);  if (actual->right!=nil)  iteration\_over(actual->right, keys, NULL);  keys->push\_back(actual->Key);  }  }  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::clear()  {  while (root != nil)  remove(root->Key);  }  template<class Key\_Type, class Value\_Type>  BiList<Key\_Type>\* RedBlackTree<Key\_Type, Value\_Type>::get\_keys()  {  BiList<Key\_Type>\* result = new BiList<Key\_Type>;  iteration\_over(root, result, NULL);  return result;  }  template<class Key\_Type, class Value\_Type>  BiList<Value\_Type>\* RedBlackTree<Key\_Type, Value\_Type>::get\_values()  {  BiList<Value\_Type>\* result = new BiList<Value\_Type>;  iteration\_over(root, NULL, result);  return result;  }  template<class Key\_Type, class Value\_Type>  void RedBlackTree<Key\_Type, Value\_Type>::print()  {  BiList<Key\_Type>\* keys = get\_keys();  BiList<Value\_Type>\* values = get\_values();  for (size\_t i = 0; i < size; i++)  cout << keys->at(i) << "||" << values->at(i) << endl;  cout << endl;  }  template<class Key\_Type, class Value\_Type>  size\_t RedBlackTree<Key\_Type, Value\_Type>::get\_size()  {  return size;  } |
| UnitRBTreeTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../Lab1/RBTree.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitRBTreeTest  {  TEST\_CLASS(BiListTest)  {  public:  BiList<int> Empty;  BiList<int> NotEmpty;  TEST\_METHOD\_INITIALIZE(SetUp)  {  for (size\_t i = 0; i < 3; i++)  NotEmpty.push\_back(i + 1);  }  TEST\_METHOD(IsEmpty\_for\_NotEmpty)  {  Assert::AreEqual(NotEmpty.isEmpty(), false);  }  TEST\_METHOD(IsEmpty\_for\_Empty)  {  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Get\_Size\_for\_Empty)  {  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Get\_Size\_for\_NotEmpty)  {  size\_t real\_size = 3;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(at\_correct\_index)  {  Assert::AreEqual(NotEmpty.at(2), 3);  }  TEST\_METHOD(at\_incorrect\_index)  {  try  {  Empty.at(4);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Push\_Back\_NotEmpty)  {  NotEmpty.push\_back(5);  Assert::AreEqual(NotEmpty.at(3), 5);  }  TEST\_METHOD(Push\_Front\_NotEmpty)  {  NotEmpty.push\_front(0);  Assert::AreEqual(NotEmpty.at(0), 0);  }  TEST\_METHOD(remove\_correct\_index)  {  int temp = NotEmpty.at(2);  NotEmpty.remove(1);  Assert::AreEqual(NotEmpty.at(1), temp);  }  TEST\_METHOD(Pop\_Back\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_back();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Pop\_Back\_for\_Empty)  {  Empty.pop\_back();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Pop\_Front\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_front();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(insert\_correct\_index)  {  NotEmpty.insert(4, 1);  Assert::AreEqual(NotEmpty.at(1), 4);  }  TEST\_METHOD(insert\_incorrect\_index)  {  try  {  Empty.insert(2, 0);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(remove\_incorrect\_index)  {  try  {  Empty.remove(2);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(reverse\_for\_NotEmpty)  {  int last\_inf = NotEmpty.at(NotEmpty.get\_size() - 1);  NotEmpty.reverse();  Assert::AreEqual(NotEmpty.at(0), last\_inf);  }  TEST\_METHOD(Pop\_Front\_for\_Empty)  {  Empty.pop\_front();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(set\_correct\_index)  {  NotEmpty.set(2, 3);  Assert::AreEqual(NotEmpty.at(2), 3);  }  TEST\_METHOD(set\_incorrect\_index)  {  try  {  NotEmpty.set(10000000, 2);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Test\_Clear\_NotEmpty)  {  NotEmpty.clear();  size\_t real\_size = 0;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Test\_Clear\_Empty)  {  Empty.clear();  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Push\_Back\_Empty)  {  Empty.push\_back(5);  Assert::AreEqual(Empty.at(0), 5);  Empty.pop\_back();  }  TEST\_METHOD(Push\_Front\_Empty)  {  Empty.push\_front(0);  Assert::AreEqual(Empty.at(0), 0);  }  };  TEST\_CLASS(UnitRBTreeTest)  {  public:  RedBlackTree<int, string> TestTree;  TEST\_METHOD\_INITIALIZE(SetUp)  {  int keys[10] = { 12,-11,45,76,34,-9,15,-87,97,111 };  string values[10] = { "cat","dog" ,"pig" ,"Kolpakov" ,"tree" ,"house" ,"beat" ,"meat" ,"SOS" ,"stackoverflow" };  for (size\_t i = 0; i < 10; i++)  TestTree.insert(keys[i], values[i]);  }  TEST\_METHOD(Search\_real)  {  string answer = "Kolpakov";  Assert::AreEqual(TestTree.find(76), answer);  }  TEST\_METHOD(Extra\_Search)  {  int keys[10] = { 12,-11,45,76,34,-9,15,-87,97,111 };  string values[10] = { "cat","dog" ,"pig" ,"Kolpakov" ,"tree" ,"house" ,"beat" ,"meat" ,"SOS" ,"stackoverflow" };  for (size\_t i = 0; i < 10; i++)  Assert::AreEqual(TestTree.find(keys[i]), values[i]);  }  TEST\_METHOD(Search\_unreal)  {  try {  TestTree.find(2020);  }  catch (const char\* error)  {  Assert::AreEqual(error, "No such element");  }  }  TEST\_METHOD(insert\_test)  {  string answer = "SPb";  TestTree.insert(1, answer);  Assert::AreEqual(TestTree.find(1), answer);  }  TEST\_METHOD(remove\_test\_real)  {  try {  string answer = "SPb";  TestTree.remove(1);  TestTree.find(1);  }  catch (const char\* error)  {  Assert::AreEqual(error, "No such element");  }  }  TEST\_METHOD(get\_keys\_test)  {  BiList<int>\* keys = TestTree.get\_keys();  try {  for (size\_t i = 0; i < 10; i++)  {  TestTree.remove(keys->at(i));  }  }  catch (const char\* error)  {  Assert::AreEqual(1, 0);  }  Assert::AreEqual(TestTree.get\_size(), (size\_t)0);  }  TEST\_METHOD(get\_value\_test)  {  size\_t check = 0;  string value[10] = { "cat","dog" ,"pig" ,"Kolpakov" ,"tree" ,"house" ,"beat" ,"meat" ,"SOS" ,"stackoverflow" };  BiList<string>\* values = TestTree.get\_values();  for (size\_t i = 0; i < 10; i++)  {  string temp = values->at(i);  for (size\_t j = 0; j < 10; j++)  {  if (temp == value[j])  check++;  }  }  Assert::AreEqual(check, (size\_t)10);  }  TEST\_METHOD(clear\_test)  {  TestTree.clear();  Assert::AreEqual(TestTree.get\_size(), (size\_t)0);  }  };  } |

## Результат выполнения всех unit-тестов



## Вывод

В ходе выполнения работы я научился реализовывать ассоциативный массив на основе красно-черного дерева