**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 |  | Давтян С.Д. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

## Постановка задачи и описание реализуемого класса и методов.

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

Для этого мне понадобились классы «Tree\_Node», «List» (взят из предыдущих лабораторных работ и переделан под нужную реализацию), «Node».

template<class data\_T>

class List

|  |  |  |
| --- | --- | --- |
| Поля | Описание | |
| Node \*head; | Указатель на начало списка | |
| Node \*tail; | Указатель на конец списка | |
| size\_t size; | Переменная, хранящая размер списка | |
| Методы | Описание | Оценка временной сложности |
| void push\_back(data\_T) | Добавление в конец списка | O(1) |
| void push\_front(data\_T) | Добавление в начало списка | O(1) |
| void pop\_back() | Удаление последнего элемента | O(1) |
| void pop\_front() | Удаление первого элемента | O(1) |
| void insert(size\_t, data\_T) | Добавление элемента по индексу | O(n) |
| data\_T at(const int) | Получение элемента по индексу | O(n) |
| void remove(size\_t) | Удаление элемента по индексу | O(n) |
| size\_t GetSize() | Получение размера списка | O(1) |
| void print\_to\_console() | Вывод элементов в консоль через разделитель | O(n) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void clear() | Удаление всех элементов списка | O(n) |
| void reverse | Изменение порядка элементов в списке на обратный | O(n) |
| void set(size\_t, data\_T) | Замена элемента по индексу | O(n) |
| data\_T top() | Получение первого элемента в списке | O(1) |

class Node

|  |  |
| --- | --- |
| Поля | Описание |
| Node \*next; | Указатель на следующий элемент списка |
| Node \*prev; | Указатель на предыдущий элемент списка |
| data\_T data; | Информация, хранящаяся в элементах списка |

template<class key\_type, class data\_type>

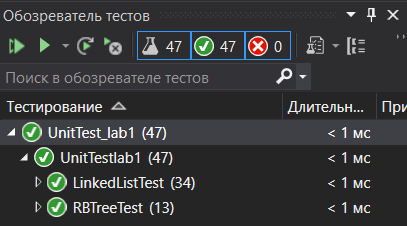
class Tree\_Node

|  |  |  |
| --- | --- | --- |
| Поля | Описание | |
| Tree\_Node\* root; | Корень дерева | |
| Tree\_Node\* parent; | Родительский элемент | |
| Tree\_Node\* right; | Правый ребенок | |
| Tree\_Node\* left; | Левый ребенок | |
| Tree\_Node\* nil; | Пустой лист дерева | |
| Color color; | Цвет узла | |
| data\_type data; | Информация, хранящаяся в узле | |
| key\_type key; | Ключ | |
| Методы | Описание | O |
| void left\_rorate(Tree\_Node\*); | Восстанавливание свойств красно-черного дерева | O(1) |
| void right\_rorate(Tree\_Node\*); | Восстанавливание свойств красно-черного дерева | O(1) |
| void Insert(key\_type, data\_type); | Добавление элемента с ключом и значением | O(log(n)) |
| void Insert\_FIXUP(Tree\_Node\*); | Восстанавливание свойств красно-черного дерева при добавлении элемента | O(log(n)) |
| void Remove(key\_type); | Удаление элемента дерева по ключу | O(log(n)) |
| void Transplant(Tree\_Node\*, Tree\_Node\*); | Перемещение поддеревьев | O(1) |
| Tree\_Node\* Tree\_Minimum(Tree\_Node\*); | Поиск минимального элемента поддерева | O(log(n)) |
| void Delete\_FIXUP(Tree\_Node\*); | Восстанавливание свойств красно-черного дерева при удалении элемента | O(log(n)) |
| data\_type Find(key\_type); | Поиск элемента по ключу | O(log(n)) |
| void Clear(); | Очищение ассоциативного массива | O(n) |
| void Get\_Keys(Tree\_Node\*, List<key\_type>\*); | Возвращение списка ключей | O(log(n)) |
| void Get\_Values(Tree\_Node\*, List<data\_type>\*); | Возвращение списка значений | O(log(n)) |
| void Print(Tree\_Node\*) | Вывод в консоль | O(log(n)) |
| Tree\_Node\* Get\_Root() | Получение корня дерева | O(1) |
| Tree\_Node\* Get\_Nil() | Получение листа дерева | O(1) |
| void Nil\_Creating() | Создание листа дерева | O(1) |

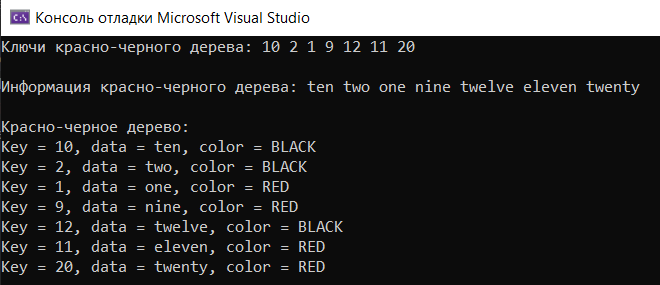
## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Проверка методов класса «Tree\_Node» | |
| Название теста | Что проверяет |
| Left\_Rotate\_Test | Проверка метода left\_rorate |
| Right\_Rotate\_Test | Проверка метода right\_rorate |
| Insert\_Test\_1 | Проверка метода Insert со вставкой элемента с большим ключом |
| Insert\_Test\_2 | Проверка метода Insert со вставкой элемента с маленьким ключом |
| Find\_Test\_1 | Проверка метода Find в непустом дереве |
| Find\_Test\_2 | Проверка метода Find в пустом дереве |
| Find\_Test\_3 | Проверка метода Find в дереве, которое не содержит искомый ключ |
| Remove\_Test\_1 | Проверка метода Remove в непустом дереве |
| Remove\_Test\_2 | Проверка метода Remove в пустом дереве |
| Remove\_Test\_3 | Проверка метода Remove в дереве, которое не содержит искомый ключ |
| Clear\_Test | Проверка метода Clear |
| Get\_Keys\_Test | Проверка метода Get\_Keys |
| Get\_Values\_Test | Проверка метода Get\_Values |
| Проверка функций для класса «List» (взято из предыдущих лабораторных работ) | |

## Результат выполнения всех unit-тестов



## Пример работы программы



## Листинг

|  |
| --- |
| List.h |
| #pragma once  #include <iostream>  #include <string>  using namespace std;  template<class data\_T>  class List  {  public:  List();  ~List();  void push\_back(data\_T data);  void push\_front(data\_T data);  void pop\_back();  void pop\_front();  void clear();  size\_t GetSize();  data\_T at(const int index);  bool isEmpty();  void remove(size\_t index);  void set(size\_t index, data\_T data);  void insert(size\_t index, data\_T data);  void reverse();  void print\_to\_console();  data\_T top();  private:  class Node  {  public:  Node\* next;  Node\* prev;  data\_T data;  Node(data\_T data = "\0", Node\* next = nullptr, Node\* prev = nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  };  Node\* head;  Node\* tail;  size\_t size;  };  #include "List.inl" |
| List.inl |
| #pragma once  #include "List.h"  // constructor  template<class data\_T>  List<data\_T>::List()  {  size = 0;  head = nullptr;  tail = nullptr;  }  // destructor  template<class data\_T>  List<data\_T>::~List()  {  clear();  }  // inserting an element at the end of the list  template<class data\_T>  void List<data\_T>::push\_back(data\_T data)  {  if (head == nullptr)  {  this->head = this->tail = new Node(data);  }  else  {  Node\* temp = new Node(data, nullptr, tail);  tail->next = temp;  tail = temp;  }  size++;  }  // inserting an element at the beginning of the list  template<class data\_T>  void List<data\_T>::push\_front(data\_T data)  {  if (!(isEmpty()))  {  Node\* lasthead = head;  head = new Node(data, head);  lasthead->prev = head;  }  else  {  head = new Node(data, head);  }  size++;  }  //remove the last element  template<class data\_T>  void List<data\_T>::pop\_back()  {  if (isEmpty())  throw "Error! Linked list is Empty.";  else  {  Node\* todelete = tail;  tail = tail->prev;  delete todelete;  size--;  }  }  // remove the first element  template<class data\_T>  void List<data\_T>::pop\_front()  {  if (head != nullptr) {  Node\* temp = this->head;  head = head->next;  delete temp;  size--;  }  else throw "Error! Linked list is Empty.";  }  // clear list  template<class data\_T>  void List<data\_T>::clear()  {  while (size) {  pop\_front();  }  }  // getting list size  template<class data\_T>  size\_t List<data\_T>::GetSize()  {  return size;  }  // output the list to the console  template<class data\_T>  void List<data\_T>::print\_to\_console()  {  Node\* cursor = head;  if (head)  {  while (cursor->next)  {  std::cout << cursor->data << " ";  cursor = cursor->next;  }  std::cout << cursor->data;  std::cout << std::endl;  }  else throw "Error! List is empty.";  }  // getting an element by index  template<class data\_T>  data\_T List<data\_T>::at(const int index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  int counter = 0;  Node\* current = head;  while (current != nullptr)  {  if (counter == index)  return current->data;  current = current->next;  counter++;  }  }  // checking the list for emptiness  template<class data\_T>  bool List<data\_T>::isEmpty()  {  if (head == nullptr)  return true;  else return false;  }  // replacing the element by index with the passed element  template<class data\_T>  void List<data\_T>::set(size\_t index, data\_T data)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  cursor->data = data;  }  // insert into an arbitrary place in the list by index  template<class data\_T>  void List<data\_T>::insert(size\_t index, data\_T data)  {  if (!isEmpty())  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  }  else  {  if (index != size)  throw "Error! Incorrect input.";  }  if (index == 0)  {  push\_front(data);  }  else  {  if (index == size)  push\_back(data);  else {  if (index < (size - 1) / 2)  {  Node\* previous = this->head;  for (size\_t i = 0; i < index - 1; i++)  {  previous = previous->next;  }  previous->next = new Node(data, previous->next, previous);  }  else  {  Node\* previous = this->tail;  for (size\_t i = size - 1; i > index - 1; i--)  {  previous = previous->prev;  }  previous->next = new Node(data, previous->next, previous);  }  size++;  }  }  }  // кeverse the order of items in the list  template<class data\_T>  void List<data\_T>::reverse()  {  if (head == nullptr)  throw "Error! Linked list is empty.";  if (!head || !head->next)  return;  tail = head;  Node\* temp = nullptr;  Node\* current = head;  while (current != nullptr)  {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  head = temp->prev;  }  // deleting an element by index  template<class data\_T>  void List<data\_T>::remove(size\_t index)  {  if (index >= size || index < 0)  throw "Error! Incorrect input.";  if (index == 0)  this->pop\_front();  else if (index == size - 1)  this->pop\_back();  else {  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--;  }  }  // getting the first item in the list  template<class data\_T>  data\_T List<data\_T>::top()  {  if (head)  return head->data;  else throw "Error! Stack is empty.";  } |
| RB-Tree.h |
| #pragma once  #include "List.h"  #include <iostream>  enum Color { BLACK, RED };  template<class key\_type, class data\_type>  class Tree\_Node  {  public:  ~Tree\_Node()  {  Clear();  }  void left\_rorate(Tree\_Node\* x);  void right\_rorate(Tree\_Node\* y);  // INSERT  void Insert(key\_type key, data\_type data);  void Insert\_FIXUP(Tree\_Node\* node);  // REMOVE  void Remove(key\_type key);  void Transplant(Tree\_Node\* u, Tree\_Node\* v);  Tree\_Node\* Tree\_Minimum(Tree\_Node\* node);  void Delete\_FIXUP(Tree\_Node\* node);  // FIND  data\_type Find(key\_type key);  // CLEAR  void Clear();  // GET KEYS  void Get\_Keys(Tree\_Node\* node, List<key\_type>\* keys);  // GET VALUES  void Get\_Values(Tree\_Node\* node, List<data\_type>\* values);  // PRINT  void Print(Tree\_Node\* node);  Tree\_Node\* Get\_Root() { return this->root; }  Tree\_Node\* Get\_Nil() { return this->nil; }  void Nil\_Creating()  {  this->nil = new Tree\_Node;  this->nil->color = BLACK;  this->root = nil;  }    private:  Tree\_Node\* root;  Tree\_Node\* parent;  Tree\_Node\* right;  Tree\_Node\* left;  Tree\_Node\* nil;  Color color;  data\_type data;  key\_type key;  };  #include "RB-Tree.inl" |
| RB-Tree.inl |
| #pragma once  #include "RB-Tree.h"  using namespace std;  //Restoring the properties of red-black tree  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::left\_rorate(Tree\_Node\* x)  {  Tree\_Node\* y;  y = x->right;  x->right = y->left;  if (y->left != this->nil)  y->left->parent = x;  if (y != this->nil)  y->parent = x->parent;  if (x->parent == this->nil)  this->root = y;  else if (x == x->parent->left)  x->parent->left = y;  else  x->parent->right = y;  y->left = x;  if (x != this->nil)  x->parent = y;  }  //Restoring the properties of red-black tree  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::right\_rorate(Tree\_Node\* x)  {  Tree\_Node\* y;  y = x->left;  x->left = y->right;  if (y->right != this->nil)  y->right->parent = x;  if (y != this->nil)  y->parent = x->parent;  if (x->parent == this->nil)  this->root = y;  else if (x == x->parent->left)  x->parent->left = y;  else  x->parent->right = y;  y->right = x;  if (x != this->nil)  x->parent = y;  }  //Adding an element with a key and value  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Insert(key\_type key, data\_type data)  {  Tree\_Node\* x, \* y;  y = this->nil;  x = this->root;  Tree\_Node\* New\_Node = new Tree\_Node;  New\_Node->key = key;  New\_Node->data = data;  while (x != this->nil)  {  y = x;  if (New\_Node->key < x->key)  x = x->left;  else  x = x->right;  }  New\_Node->parent = y;  if (y == this->nil)  this->root = New\_Node;  else if (New\_Node->key < y->key)  y->left = New\_Node;  else  y->right = New\_Node;  New\_Node->left = this->nil;  New\_Node->right = this->nil;  New\_Node->color = RED;  Insert\_FIXUP(New\_Node);  }  //Restoring the properties of the red-black tree when adding an element  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Insert\_FIXUP(Tree\_Node\* node)  {  while (node->parent->color == RED && node != this->root)  {  if (node->parent == node->parent->parent->left)  {  Tree\_Node\* y = node->parent->parent->right;  if (y->color == RED)  {  node->parent->color = BLACK;  y->color = BLACK;  node->parent->parent->color = RED;  node = node->parent->parent;  }  else  {  if (node == node->parent->right)  {  node = node->parent;  left\_rorate(node);  }  node->parent->color = BLACK;  node->parent->parent->color = RED;  right\_rorate(node->parent->parent);  }  }  else  {  Tree\_Node\* y = node->parent->parent->left;  if (y->color == RED)  {  node->parent->color = BLACK;  y->color = BLACK;  node->parent->parent->color = RED;  node = node->parent->parent;  }  else  {  if (node == node->parent->left)  {  node = node->parent;  right\_rorate(node);  }  node->parent->color = BLACK;  node->parent->parent->color = RED;  left\_rorate(node->parent->parent);  }  }  }  root->color = BLACK;  }  //Moving subtrees  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Transplant(Tree\_Node\* u, Tree\_Node\* v)  {  if (u->parent == this->nil)  this->root = v;  else if (u == u->parent->left)  u->parent->left = v;  else  u->parent->right = v;  v->parent = u->parent;  }  //Search for an element by key  template<class key\_type, class data\_type>  data\_type Tree\_Node<key\_type, data\_type>::Find(key\_type key)  {  if (this->root == this->nil)  throw "RB Tree is empty.";  else  {  Tree\_Node\* temp = this->root;  while (temp->key != key)  {  if (key > temp->key)  temp = temp->right;  else  temp = temp->left;  if (temp == this->nil)  throw "Node not found.";  }  return temp->data;  }  }  //Clearing a map  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Clear()  {  while (this->root != this->nil)  Remove(root->key);  }  //Returning a list of keys  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Get\_Keys(Tree\_Node\* node, List<key\_type>\* keys)  {  if (this->root == this->nil)  throw "RB Tree is empty.";  if (node == this->nil)  return;  else  {  keys->push\_back(node->key);  if (node->left != this->nil)  Get\_Keys(node->left, keys);  }  if (node->right != this->nil)  Get\_Keys(node->right, keys);  }  //Returning a list of values  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Get\_Values(Tree\_Node\* node, List<data\_type>\* values)  {  if (this->root == this->nil)  throw "RB Tree is empty.";  if (node == this->nil)  return;  else  {  values->push\_back(node->data);  if (node->left != this->nil)  Get\_Values(node->left, values);  }  if (node->right != this->nil)  Get\_Values(node->right, values);  }  //Print to console  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Print(Tree\_Node\* node)  {  if (this->root == this->nil)  throw "RB Tree is empty.";  if (node == this->nil)  return;  else  {  cout << "Key = " << node->key << ", data = " << node->data;  if (node->color == 0)  cout<< ", color = BLACK" << endl;  if (node->color == 1)  cout << ", color = RED" << endl;  if (node->left != this->nil)  Print(node->left);  }  if (node->right != this->nil)  Print(node->right);  }  //Deleting a tree element by key  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Remove(key\_type key)  {  if (this->root == this->nil)  throw "RB Tree is empty.";  Tree\_Node\* y, \* x;  Tree\_Node\* z = this->root;  while (z->key != key)  {  if (key > z->key)  z = z->right;  else  z = z->left;  if (z == this->nil)  throw "Node not found.";  }  y = z;  Color y\_original\_color = y->color;  if (z->left == this->nil)  {  x = z->right;  Transplant(z, z->right);  }  else if (z->right == this->nil)  {  x = z->left;  Transplant(z, z->left);  }  else  {  y = Tree\_Minimum(z->right);  y\_original\_color = y->color;  x = y->right;  if (y->parent == z)  x->parent = y;  else  {  Transplant(y, y->right);  y->right = z->right;  y->right->parent = y;  }  Transplant(z, y);  y->left = z->left;  y->left->parent = y;  y->color = z->color;  }  if (y\_original\_color == BLACK)  Delete\_FIXUP(x);  }  //Search for the minimum element of a subtree  template<class key\_type, class data\_type>  Tree\_Node<key\_type, data\_type>\* Tree\_Node<key\_type, data\_type>::Tree\_Minimum(Tree\_Node\* node)  {  if (this->root == this->nil)  throw "RB Tree is empty.";  else  {  while (node->left != this->nil)  node = node->left;  return node;  }  }  //Restoring the properties of the red-black tree when deleting an element  template<class key\_type, class data\_type>  void Tree\_Node<key\_type, data\_type>::Delete\_FIXUP(Tree\_Node\* x)  {  while (x != this->root && x->color == BLACK)  {  if (x == x->parent->left)  {  Tree\_Node\* w;  w = x->parent->right;  if (w->color == RED)  {  w->color = BLACK;  x->parent->color = RED;  left\_rorate(x->parent);  w = x->parent->right;  }  if (w->left->color == BLACK && w->right->color == BLACK)  {  w->color = RED;  x = x->parent;  }  else  {  if (w->right->color == BLACK)  {  w->left->color = BLACK;  w->color = RED;  right\_rorate(w);  w = x->parent->right;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->right->color = BLACK;  left\_rorate(x->parent);  x = this->root;  }  }  else  {  Tree\_Node\* w = x->parent->left;  if (w->color == RED)  {  w->color = BLACK;  x->parent->color = RED;  right\_rorate(x->parent);  w = x->parent->left;  }  if (w->right->color == BLACK && w->left->color == BLACK)  {  w->color = RED;  x = x->parent;  }  else  {  if (w->left->color == BLACK)  {  w->right->color = BLACK;  w->color = RED;  left\_rorate(w);  w = x->parent->left;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->left->color = BLACK;  right\_rorate(x->parent);  x = this->root;  }  }  }  x->color = BLACK;  } |
| UnitTest\_lab1.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include <string>  #include "..\Program1\List.inl"  #include "..\Program1\RB-Tree.inl"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTestlab1  {  TEST\_CLASS(RBTreeTest)  {  public:  Tree\_Node<int, string> tree;  TEST\_METHOD\_INITIALIZE(SetUp)  {  tree.Nil\_Creating();  int keys[3] = { 9,10,11 };  string data[3] = { "nine","ten" ,"eleven" };  for (size\_t i = 0; i < 3; i++)  tree.Insert(keys[i], data[i]);  }  TEST\_METHOD(Left\_Rotate\_Test)  {  List<int> temp;  tree.left\_rorate(tree.Get\_Root());  tree.Get\_Keys(tree.Get\_Root(), &temp);  Assert::IsTrue(temp.at(0) == 11 && temp.at(1) == 10 && temp.at(2) == 9);  }  TEST\_METHOD(Right\_Rotate\_Test)  {  List<int> temp;  tree.right\_rorate(tree.Get\_Root());  tree.Get\_Keys(tree.Get\_Root(), &temp);  Assert::IsTrue(temp.at(0) == 9 && temp.at(1) == 10 && temp.at(2) == 11);  }  TEST\_METHOD(Insert\_Test\_1)  {  List<int> temp;  tree.Insert(25, "hello");  tree.Get\_Keys(tree.Get\_Root(), &temp);  Assert::IsTrue(temp.at(0) == 10 && temp.at(1) == 9 && temp.at(2) == 11 && temp.at(3) == 25);  }  TEST\_METHOD(Insert\_Test\_2)  {  List<int> temp;  tree.Insert(-1, "hello");  tree.Get\_Keys(tree.Get\_Root(), &temp);  Assert::IsTrue(temp.at(0) == 10 && temp.at(1) == 9 && temp.at(2) == -1 && temp.at(3) == 11);  }  TEST\_METHOD(Find\_Test\_1)  {  string correct = "nine";  Assert::AreEqual(tree.Find(9), correct);  }  TEST\_METHOD(Find\_Test\_2)  {  try  {  Tree\_Node<int, int> temp;  temp.Find(5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "RB Tree is empty.");  }  }  TEST\_METHOD(Find\_Test\_3)  {  try  {  tree.Find(5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Node not found.");  }  }  TEST\_METHOD(Remove\_Test\_1)  {  List<int> temp;  tree.Insert(12, "twelve");  tree.Insert(13, "thirteen");  tree.Remove(10);  tree.Get\_Keys(tree.Get\_Root(), &temp);  Assert::IsTrue(temp.at(0) == 11 && temp.at(1) == 9 && temp.at(2) == 12 && temp.at(3) == 13);  }  TEST\_METHOD(Remove\_Test\_2)  {  try  {  Tree\_Node<int, int> temp;  temp.Remove(5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "RB Tree is empty.");  }  }  TEST\_METHOD(Remove\_Test\_3)  {  try  {  tree.Remove(5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Node not found.");  }  }  TEST\_METHOD(Clear\_Test)  {  tree.Clear();  Assert::IsTrue(tree.Get\_Root() == tree.Get\_Nil());  }  TEST\_METHOD(Get\_Keys\_Test)  {  List<int> keys;  tree.Get\_Keys(tree.Get\_Root(), &keys);  Assert::IsTrue(keys.at(0) == 10 && keys.at(1) == 9 && keys.at(2) == 11);  }  TEST\_METHOD(Get\_Values\_Test)  {  List<string> values;  tree.Get\_Values(tree.Get\_Root(), &values);  Assert::IsTrue(values.at(0) == "ten" && values.at(1) == "nine" && values.at(2) == "eleven" );  }  };  TEST\_CLASS(LinkedListTest)  {  public:  TEST\_METHOD(ConstructorTest)  {  List<int> lst;  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(IsEmptyTest1)  {  List<int> lst;  Assert::IsTrue(lst.isEmpty());  }  TEST\_METHOD(IsEmptyTest2)  {  List<int> lst;  lst.push\_back(3);  Assert::IsTrue(lst.isEmpty() == false);  }  TEST\_METHOD(AtTest1)  {  List<int> lst;  lst.push\_back(3);  Assert::IsTrue(lst.at(0) == 3);  }  TEST\_METHOD(AtTest2)  {  List<int> lst;  lst.push\_back(3);  lst.push\_front(2);  Assert::IsTrue(lst.at(0) == 2, L"Assert 1");  Assert::IsTrue(lst.at(1) == 3, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 2, L"Assert 3");  }  TEST\_METHOD(AtTest3)  {  List<int> lst;  lst.push\_back(3);  try  {  lst.at(-5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  TEST\_METHOD(GetSizeTest1)  {  List<int> lst;  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(GetSizeTest2)  {  List<int> lst;  lst.push\_back(3);  lst.push\_front(2);  Assert::IsTrue(lst.GetSize() == 2);  }  TEST\_METHOD(GetSizeTest3)  {  List<int> lst;  lst.push\_back(3);  lst.push\_front(2);  lst.pop\_back();  lst.pop\_front();  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(PushBackTest1)  {  List<int> lst;  lst.push\_back(3);  Assert::IsTrue(lst.GetSize() == 1, L"Assert 1");  Assert::IsTrue(lst.at(0) == 3, L"Assert 2");  }  TEST\_METHOD(PushBackTest2)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  Assert::IsTrue(lst.GetSize() == 4, L"Assert 1");  Assert::IsTrue(lst.at(3) == 3, L"Assert 2");  }  TEST\_METHOD(PushFrontTest1)  {  List<int> lst;  lst.push\_front(3);  Assert::IsTrue(lst.GetSize() == 1, L"Assert 1");  Assert::IsTrue(lst.at(0) == 3, L"Assert 2");  }  TEST\_METHOD(PushFrontTest2)  {  List<int> lst;  lst.push\_front(0);  lst.push\_front(1);  lst.push\_front(2);  lst.push\_front(3);  Assert::IsTrue(lst.GetSize() == 4, L"Assert 1");  Assert::IsTrue(lst.at(3) == 0, L"Assert 2");  }  TEST\_METHOD(PopBackTest1)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.pop\_back();  Assert::IsTrue(lst.GetSize() == 3);  }  TEST\_METHOD(PopBackTest2)  {  List<int> lst;  lst.push\_back(0);  lst.pop\_back();  Assert::IsTrue(lst.GetSize() == 0);  }  TEST\_METHOD(PopBackTest3)  {  List<int> lst;  try  {  lst.pop\_back();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Linked list is Empty.");  }  }  TEST\_METHOD(PopFrontTest1)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.pop\_front();  Assert::IsTrue(lst.at(0) == 1, L"Assert 1");  Assert::IsTrue(lst.at(1) == 2, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 3, L"Assert 3");  }  TEST\_METHOD(PopFrontTest2)  {  List<int> lst;  lst.push\_back(0);  lst.pop\_front();  Assert::IsTrue(lst.isEmpty(), L"Assert 1");  }  TEST\_METHOD(PopFrontTest3)  {  List<int> lst;  try  {  lst.pop\_front();  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Linked list is Empty.");  }  }  TEST\_METHOD(InsertTest1)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.insert(0, 21);  Assert::IsTrue(lst.at(0) == 21, L"Assert 1");  Assert::IsTrue(lst.at(1) == 0, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 4, L"Assert 3");  }  TEST\_METHOD(InsertTest2)  {  List<int> lst;  lst.insert(0, 21);  Assert::IsTrue(lst.at(0) == 21, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 1, L"Assert 2");  }  TEST\_METHOD(InsertTest3)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.insert(2, 21);  Assert::IsTrue(lst.at(2) == 21, L"Assert 1");  Assert::IsTrue(lst.at(3) == 2, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 4, L"Assert 3");  }  TEST\_METHOD(InsertTest4)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.insert(2, 21);  Assert::IsTrue(lst.at(2) == 21, L"Assert 1");  Assert::IsTrue(lst.at(3) == 2, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 6, L"Assert 3");  }  TEST\_METHOD(InsertTest5)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  try  {  lst.insert(-3, 21);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  TEST\_METHOD(SetTest1)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.set(0, 21);  Assert::IsTrue(lst.at(0) == 21, L"Assert 1");  Assert::IsTrue(lst.at(1) == 1, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 3, L"Assert 3");  }  TEST\_METHOD(SetTest2)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.set(2, 21);  Assert::IsTrue(lst.at(2) == 21, L"Assert 1");  Assert::IsTrue(lst.at(3) == 3, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 9, L"Assert 3");  }  TEST\_METHOD(SetTest3)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.set(7, 21);  Assert::IsTrue(lst.at(7) == 21, L"Assert 1");  Assert::IsTrue(lst.at(8) == 8, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 9, L"Assert 3");  }  TEST\_METHOD(SetTest4)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.set(8, 21);  Assert::IsTrue(lst.at(8) == 21, L"Assert 1");  Assert::IsTrue(lst.at(7) == 7, L"Assert 2");  Assert::IsTrue(lst.GetSize() == 9, L"Assert 3");  }  TEST\_METHOD(SetTest5)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  try  {  lst.set(-3, 21);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  TEST\_METHOD(RemoveTest1)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.remove(0);  Assert::IsTrue(lst.at(0) == 1, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 1, L"Assert 2");  }  TEST\_METHOD(RemoveTest2)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.remove(2);  Assert::IsTrue(lst.at(2) == 3, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 8, L"Assert 2");  }  TEST\_METHOD(RemoveTest3)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.push\_back(6);  lst.push\_back(7);  lst.push\_back(8);  lst.remove(6);  Assert::IsTrue(lst.at(6) == 7, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 8, L"Assert 2");  }  TEST\_METHOD(RemoveTest4)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  lst.push\_back(3);  lst.push\_back(4);  lst.push\_back(5);  lst.remove(5);  Assert::IsTrue(lst.at(4) == 4, L"Assert 1");  Assert::IsTrue(lst.GetSize() == 5, L"Assert 2");  }  TEST\_METHOD(RemoveTest5)  {  List<int> lst;  lst.push\_back(0);  lst.push\_back(1);  lst.push\_back(2);  try  {  lst.remove(-2);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect input.");  }  }  };  } |
| main.cpp |
| #include <iostream>  #include "RB-Tree.h"  #include "List.h"  using namespace std;  int main()  {  setlocale(LC\_ALL, "Rus");  Tree\_Node<int, string> tree;  List<int> keys\_list;  List<string> values\_list;  tree.Nil\_Creating();  int keys[3] = { 9,10,11 };  string values[3] = { "nine","ten" ,"eleven" };  for (size\_t i = 0; i < 3; i++)  tree.Insert(keys[i], values[i]);  tree.Insert(12, "twelve");  tree.Insert(20, "twenty");  tree.Insert(1, "one");  tree.Insert(2, "two");  //tree.Remove(10);  cout << "Ключи красно-черного дерева: ";  tree.Get\_Keys(tree.Get\_Root(), &keys\_list);  keys\_list.print\_to\_console();  cout << endl;  cout << "Информация красно-черного дерева: ";  tree.Get\_Values(tree.Get\_Root(), &values\_list);  values\_list.print\_to\_console();  cout << endl;  cout << "Красно-черное дерево: " << endl;  tree.Print(tree.Get\_Root());  } |