**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»**

Студент гр. 9302 \_\_\_\_\_\_\_\_\_\_\_\_ Тарабурин А.П.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2021

Оглавление

[1. Постановка задачи и описание реализуемого класса и методов 3](#_Toc55082721)

[2. Описание реализованных unit-тестов 4](#_Toc55082722)

[3. Код программы 5](#_Toc55082723)

[4. Пример работы 21](#_Toc55082724)

[5. Вывод 22](#_Toc55082725)

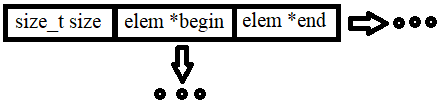
# Постановка задачи и описание реализуемого класса и методов

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

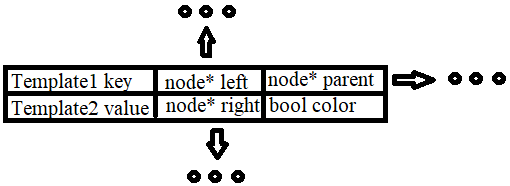
Class elem



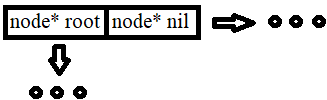
Class list



Class node



Class map



|  |  |  |
| --- | --- | --- |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void push\_back(int) | Добавление в конец списка | O(1) |
| void push\_front(int) | Добавление в начало списка | O(1) |
| void pop\_back() | Удаление последнего элемента | O(n) |
| void pop\_front() | Удаление первого элемента | O(1) |
| void insert(int, size\_t) | Добавление элемента по индексу | O(n) |
| int 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) |
| void clear() | Удаление всех элементов списка | O(n) |
| void set(size\_t, int) | Замена элемента по индексу на передаваемый элемент | O(n) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| void push\_front(list) | Вставка другого списка в начало | O(n) |
| **map** | | |
| void leftRotation(node\*x) | Левый поворот узла в выбранном дереве | O(1) |
| void rightRotation(node\*x) | Правый поворот узла в выбранном дереве | O(1) |
| void insert(T1 node\_key, T2 node\_value) | Вставка узла с выбранными ключом и значением в красно-черное дерево | O(lgn) |
| void insFixup(node<T1, T2>\* newNode) | Восстановление свойств красно-черного дерево после вставки | O(lgn) |
| void remove(T1 remove\_key) | Удаление узла с выбранным ключом из красно-черного дерева | O(lgn) |
| void transplant(node<T1, T2>\* u, node<T1, T2>\* v) | Подпрограмма удаления | O(1) |
| void delFixup(node<T1, T2>\* removeNode) | Восстановление свойств красно-черного дерево после удаления | O(lgn) |
| node<T1, T2>\* minimum(node<T1, T2>\* x) | Нахождение минимального узла в дереве | O(1) |
| node<T1, T2>\* find(T1 node\_key) | Поиск узла по ключу | O(lgn) |
| void print() | Вывод данных о дереве в консоль | O(n) |
| void clear() | Очистка дерева | O(n) |
| void get\_keys(list<T1>\* temp, node<T1, T2>\*cur) | Получение списка ключей дерева | O(n) |
| void get\_values(list<T2>\* temp, node<T1, T2>\* cur) | Получение списка значений дерева | O(n) |

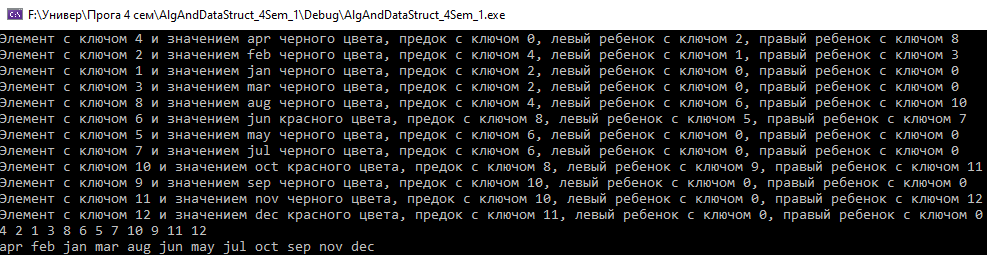
# Описание реализованных unit-тестов

|  |  |
| --- | --- |
| Имя теста | Описание |
| CreateTest | Проверка работы конструктора |
| PushBackTest1 | Проверка добавления элемента в конец пустого списка |
| PushBackTest2 | Проверка добавления элемента в конец непустого списка |
| PushFrontTest1 | Проверка добавления элемента в начало пустого списка |
| PushFrontTest2 | Проверка добавления элемента в начало непустого списка |
| PopBackTest1 | Проверка удаления элемента из конца непустого списка |
| PopBackTest2 | Проверка удаления элемента из конца списка с одним элементом |
| PopBackTest3 | Проверка удаления элемента из конца пустого списка |
| PopFrontTest1 | Проверка удаления элемента из начала непустого списка |
| PopFrontTest2 | Проверка удаления элемента из начала списка с одним элементом |
| PopFrontTest3 | Проверка удаления элемента из начала пустого списка |
| InsertTest1 | Проверка вставки элемента в пустой список на первую позицию |
| InsertTest2 | Проверка вставки элемента в пустой список на вторую позицию |
| InsertTest3 | Проверка вставки элемента в непустой список на посл. позицию |
| InsertTest4 | Проверка вставки элемента в непустой список на первую позицию |
| InsertTest5 | Проверка вставки элемента в середину непустого списка |
| AtTest1 | Проверка получения элемента по индексу |
| AtTest2 | Проверка получения элемента с первым индексом из пустого списка |
| RemoveTest1 | Проверка удаления элемента из пустого списка |
| RemoveTest2 | Проверка удаления элемента из непустого списка с посл. позиции |
| RemoveTest3 | Проверка удаления элемента из непустого списка с первой позиции |
| RemoveTest4 | Проверка удаления элемента из середины непустого списка |
| RemoveTest5 | Проверка удаления элемента с несуществующим индексом |
| GetSizeTest1 | Проверка получения размера пустого списка |
| GetSizeTest2 | Проверка получения размера непустого списка |
| SetTest1 | Проверка замены значения элемента непустого списка |
| SetTest2 | Проверка замены значения элемента с несуществующим индексом |
| SetTest3 | Проверка замены значения элемента непустого списка на такое же |
| SetTest4 | Проверка двойной замены элемента непустого списка |
| IsEmpty1 | Проверка определения пустоты списка |
| IsEmpty2 | Проверка определения пустоты списка |
| PushFrontListTest1 | Проверка вставки одного непустого в начало другого непустого |
| PushFrontListTest2 | Проверка вставки одного пустого в начало другого непустого |
| PushFrontListTest3 | Проверка вставки одного непустого в начало другого пустого |
| PushFrontListTest4 | Проверка вставки одного пустого в начало другого пустого |
| **map** | |
| ClearTest | Проверка корректной очистки текста |
| CreateTest | Проверка инициализации дерева |
| FindTest | Проверка поиска по ключу |
| InsertTest | Проверка вставки |
| LeftRotationTest | Проверка левого поворота |
| RightRotationTest | Проверка правого поворота |
| RemoveTest | Проверка удаления |

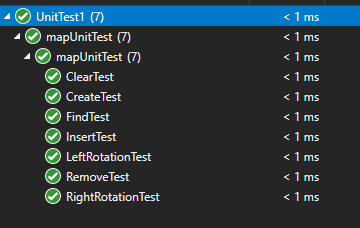
# Код программы

|  |
| --- |
| List.h |
| #pragma once  #include <iostream>  #ifndef LIST\_H  #define LIST\_H  class elem {  private:  int inf;  elem\* next;  public:  elem(int elem\_inf) { inf = elem\_inf; next = nullptr;}  ~elem() {};  elem\* getNext() { return next; }  void setNext(elem\* elem\_next) { next = elem\_next; }  int getInf() { return inf; }  void setInf(int elem\_inf) { inf = elem\_inf; }  };  class list {  private:  elem\* begin, \* end;  size\_t size;  public:  list() { begin = nullptr; end = nullptr; size = 0;}  ~list() { clear(); }  //Adding new element to end of list  void push\_back(int temp) {  elem\* newElem = new elem(temp);  if (isEmpty()) begin = newElem;  else end->setNext(newElem);  end = newElem;  size++;  }  //Adding new element to begin of list  void push\_front(int temp) {  elem\* newElem = new elem(temp);  if (isEmpty()) end = newElem;  else newElem->setNext(begin);  begin = newElem;  size++;  }  //Deleting last element from list  void pop\_back(){  if (!isEmpty()) {  if (size == 1) {  end = nullptr;  begin = nullptr;  size = 0;  }  else {  elem\* newEnd = begin;  while (newEnd->getNext() != end) newEnd = newEnd->getNext();  newEnd->setNext(nullptr);  elem\* deleted = end;  delete deleted;  end = newEnd;  size--;  }  }  else throw "List is Empty";  }  //Deleting first element from list  void pop\_front() {  if (!isEmpty()) {  if (size == 1) {  begin = nullptr;  end = nullptr;  size = 0;  }  else {  elem\* newBeg = begin->getNext();  elem\* deleted = begin;  delete deleted;  begin = newBeg;  size--;  }  }  else throw "List is Empty";  }  //Adding element to any position in list  void insert(int data, size\_t pos) {  if (pos <= size) {  if (isEmpty() || pos == size) push\_back(data);  else {  if (pos == 0) push\_front(data);  else {  elem\* newElem = new elem(data);  elem\* iter = begin;  while (pos-- > 1)  iter = iter->getNext();  newElem->setNext(iter->getNext());  iter->setNext(newElem);  size++;  }  }  }  else throw "Wrong index";  }  //Getting element from list by index  int at(size\_t pos) {  if (pos < size) {  elem\* iter = begin;  while (pos-- != 0) iter = iter->getNext();  return iter->getInf();  }  else throw "Wrong index";  }  //Deleting element from list by index  void remove(size\_t pos) {  if (pos < size) {  if (pos == 0) pop\_front();  else {  if (pos == size - 1) pop\_back();  else {  elem\* iter = begin;  while (pos-- > 1) iter = iter->getNext();  elem\* nextElem = iter->getNext();  iter->setNext(nextElem->getNext());  size--;  }  }  }  else throw "Wrong index";  }  //Getting size of list  size\_t getSize() { return size; }  //Output elements from list to console  void print\_to\_console() {  elem\* iter = begin;  for (size\_t i = 0; i < size; i++) {  std::cout << iter->getInf() << " ";  iter = iter->getNext();  }  }  //Deleting elements of list  void clear() {  while (size) pop\_back();  }  //Replacing element by index with new one  void set(size\_t pos, int data) {  if (pos < size) {  elem\* iter = begin;  while (pos-- != 0) iter = iter->getNext();  iter->setInf(data);  }  else throw "Wrong index";  }  //Checking list for filling  bool isEmpty() {  if (size == 0) return true; // 1 - Empty  else return false; // 0 - Filled  }  //Adding another list to front of this one  void push\_front(list \*lst) {  for (size\_t i = 0; i < lst->getSize(); i++)  insert(lst->at(i), i);  }  };  #endif |
| Main.cpp |
| #include "map.h"  using namespace std;  int main() {  setlocale(LC\_ALL, "rus");  map<int, string> test;  list<int>\* keys = new list<int>();  list<string>\* values = new list<string>();  node<int, string>\* temp = new node<int, string>;  test.get\_keys(keys);  test.insert(1, "jan");  test.insert(2, "feb");  test.insert(3, "mar");  temp = test.find(3);  test.rightRotation(test.getRoot());  test.insert(4, "apr");  test.insert(5, "may");  test.insert(6, "jun");  test.insert(7, "jul");  test.insert(8, "aug");  test.insert(9, "sep");  test.insert(10, "oct");  test.insert(11, "nov");  test.insert(12, "dec");  test.print();  test.get\_keys(keys);  test.get\_values(values);  keys->print\_to\_console();  cout << endl;  values->print\_to\_console();  test.clear();  return 0;  } |
| UnitTest1.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "..\Project1\Project1\List.h"  #include <iostream>  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace ListTests  {  TEST\_CLASS(ListTests)  {  public:    TEST\_METHOD(CreateTest)  {  list \*list1 = new list();  Assert::IsTrue(list1->getSize() == 0);  }  TEST\_METHOD(PushBackTest1) {  list \*list1 = new list();  list1->push\_back(1);  Assert::IsTrue(list1->at(0) == 1 && list1->getSize() == 1);  }  TEST\_METHOD(PushBackTest2) {  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->getSize() == 2);  }  TEST\_METHOD(PushFrontTest1) {  list \*list1 = new list();  list1->push\_front(1);  Assert::IsTrue(list1->at(0) == 1 && list1->getSize() == 1);  }  TEST\_METHOD(PushFrontTest2) {  list\* list1 = new list();  list1->push\_front(2);  list1->push\_front(1);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->getSize() == 2);  }  TEST\_METHOD(PopBackTest1) { // 2 elements  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->pop\_back();  Assert::IsTrue(list1->at(0) == 1 && list1->getSize() == 1);  }  TEST\_METHOD(PopBackTest2) { // 1 element  list\* list1 = new list();  list1->push\_back(1);  list1->pop\_back();  Assert::IsTrue(list1->getSize() == 0);  }  TEST\_METHOD(PopBackTest3) { // empty list  list\* list1 = new list();  try {  list1->pop\_back();  }  catch(const char\* warning){  Assert::AreEqual(warning, "List is Empty");  }  Assert::IsTrue(list1->getSize() == 0);  }  TEST\_METHOD(PopFrontTest1) { // 2 elements  list\* list1 = new list();  list1->push\_back(2);  list1->push\_back(1);  list1->pop\_front();  Assert::IsTrue(list1->at(0) == 1 && list1->getSize() == 1);  }  TEST\_METHOD(PopFrontTest2) { // 1 element  list\* list1 = new list();  list1->push\_back(1);  list1->pop\_front();  Assert::IsTrue(list1->getSize() == 0);  }  TEST\_METHOD(PopFrontTest3) { // empty list  list\* list1 = new list();  try {  list1->pop\_front();  }  catch (const char\* warning) {  Assert::AreEqual(warning, "List is Empty");  }  Assert::IsTrue(list1->getSize() == 0);  }  TEST\_METHOD(InsertTest1) { // empty list insert at first place  list\* list1 = new list();  list1->insert(1, 0);  Assert::IsTrue(list1->at(0) == 1 && list1->getSize() == 1);  }  TEST\_METHOD(InsertTest2) { // empty list insert at second place - out of range  list\* list1 = new list();  try {  list1->insert(1,1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong index");  }  Assert::IsTrue(list1->getSize() == 0);  }  TEST\_METHOD(InsertTest3) { // same as push\_back  list\* list1 = new list();  list1->push\_back(1);  list1->insert(2, 1);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->getSize() == 2 );  }  TEST\_METHOD(InsertTest4) { // same as push\_front  list\* list1 = new list();  list1->push\_back(2);  list1->insert(1, 0);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->getSize() == 2);  }  TEST\_METHOD(InsertTest5) { // insert to middle  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(3);  list1->insert(2, 1);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(AtTest1) {  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3);  }  TEST\_METHOD(AtTest2) {  list\* list1 = new list();  list1->push\_back(1);  try {  list1->at(1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong index");  }  }  TEST\_METHOD(RemoveTest1) { // empty list remove  list\* list1 = new list();  try {  list1->remove(1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong index");  }  Assert::IsTrue(list1->getSize() == 0);  }    TEST\_METHOD(RemoveTest2) { // same as pop\_back  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  list1->remove(2);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->getSize() == 2);  }  TEST\_METHOD(RemoveTest3) { // same as pop\_front  list\* list1 = new list();  list1->push\_back(3);  list1->push\_back(1);  list1->push\_back(2);  list1->remove(0);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->getSize() == 2);  }  TEST\_METHOD(RemoveTest4) { // remove from middle  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(3);  list1->push\_back(2);  list1->remove(1);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->getSize() == 2);  }  TEST\_METHOD(RemoveTest5) { // out of range check  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  try {  list1->remove(3);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong index");  }  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(GetSizeTest1) {  list\* list1 = new list();  Assert::IsTrue(list1->getSize() == 0);  }  TEST\_METHOD(GetSizeTest2) {  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  Assert::IsTrue(list1->getSize() == 3);  }  TEST\_METHOD(SetTest1) {  list\* list1 = new list();  list1->push\_back(0);  list1->push\_back(2);  list1->push\_back(3);  list1->set(0, 1);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(SetTest2) { //out of range  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  try {  list1->set(3, 4);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Wrong index");  }  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(SetTest3) { //set same  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  list1->set(0, 1);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(SetTest4) { //set twice  list\* list1 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  list1->set(0, 2);  list1->set(0, 1);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(IsEmptyTest1) {  list \*list1 = new list();  Assert::IsTrue(list1->isEmpty());  }  TEST\_METHOD(IsEmptyTest2) {  list\* list1 = new list();  list1->push\_front(1);  Assert::IsTrue(!list1->isEmpty());  }  TEST\_METHOD(PushFrontListTest1) {  list\* list1 = new list();  list\* list2 = new list();  list2->push\_back(1);  list2->push\_back(2);  list2->push\_back(3);  list1->push\_back(4);  list1->push\_back(5);  list1->push\_back(6);  list1->push\_front(list2);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->at(3) == 4 && list1->at(4) == 5 && list1->at(5) == 6 && list1->getSize() == 6);  }  TEST\_METHOD(PushFrontListTest2) { //pushed list is empty  list\* list1 = new list();  list\* list2 = new list();  list1->push\_back(1);  list1->push\_back(2);  list1->push\_back(3);  list1->push\_front(list2);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(PushFrontListTest3) { //first list is empty  list\* list1 = new list();  list\* list2 = new list();  list2->push\_back(1);  list2->push\_back(2);  list2->push\_back(3);  list1->push\_front(list2);  Assert::IsTrue(list1->at(0) == 1 && list1->at(1) == 2 && list1->at(2) == 3 && list1->getSize() == 3);  }  TEST\_METHOD(PushFrontListTest4) { //both lists are empty  list\* list1 = new list();  list\* list2 = new list();  list1->push\_front(list2);  Assert::IsTrue(list1->getSize() == 0);  }  };  }  #include "pch.h"  #include "CppUnitTest.h"  #include "..//map.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace mapUnitTest  {  TEST\_CLASS(mapUnitTest)  {  public:  map<int, string> test;  list<int> \*keys = new list<int>;  TEST\_METHOD(CreateTest)  {  test.get\_keys(keys);  Assert::IsTrue(keys->getSize() == 0);  }  TEST\_METHOD(InsertTest)  {  test.insert(1, "jan");  test.insert(2, "feb");  test.insert(3, "mar");  test.get\_keys(keys);  Assert::IsTrue(keys->at(0) == 2 && keys->at(1) == 1 && keys->at(2) == 3);  }  TEST\_METHOD(LeftRotationTest)  {  test.insert(1, "jan");  test.insert(2, "feb");  test.insert(3, "mar");  test.leftRotation(test.getRoot());  test.get\_keys(keys);  Assert::IsTrue(keys->at(0) == 3 && keys->at(1) == 2 && keys->at(2) == 1);  }  TEST\_METHOD(RightRotationTest)  {  test.insert(1, "jan");  test.insert(2, "feb");  test.insert(3, "mar");  test.rightRotation(test.getRoot());  test.get\_keys(keys);  Assert::IsTrue(keys->at(0) == 1 && keys->at(1) == 2 && keys->at(2) == 3);  }  TEST\_METHOD(RemoveTest)  {  test.insert(1, "jan");  test.insert(2, "feb");  test.insert(3, "mar");  test.remove(2);  test.get\_keys(keys);  Assert::IsTrue(keys->at(0) == 1 && keys->at(1) == 3);  }  TEST\_METHOD(FindTest)  {  node<int, string>\* temp = new node<int, string>;  test.insert(1, "jan");  test.insert(2, "feb");  test.insert(3, "mar");  temp = test.find(3);  Assert::IsTrue(temp->getValue() == "mar");  }  TEST\_METHOD(ClearTest)  {  test.insert(1, "jan");  test.insert(2, "feb");  test.insert(3, "mar");  test.clear();  test.get\_keys(keys);  Assert::IsTrue(keys->getSize() == 0);  }  };  } |
| node.h |
| #pragma once  #include <string>  #include <iostream>  using namespace std;  #ifndef NODE\_H  #define NODE\_H  #define black false  #define red true  template<class T1, class T2>  class node {  private:  T1 key;  T2 value;  node<T1, T2>\* parent;  bool color; // false - black, true - red  node<T1, T2>\* left, \* right;  public:  node() {  parent = nullptr;  color = red;  left = nullptr;  right = nullptr;  }  node(T1 node\_key, T2 node\_value, node<T1, T2>\* node\_parent = nullptr, node<T1, T2>\* node\_leftChild = nullptr, node<T1, T2>\* node\_rightChild = nullptr, bool node\_color = red) {  key = node\_key;  value = node\_value;  parent = node\_parent;  color = node\_color;  left = node\_leftChild;  right = node\_rightChild;  }  node<T1, T2>& operator= (node<T1, T2> equated\_node) {  key = equated\_node.key;  value = equated\_node.value;  parent = equated\_node.parent;  color = equated\_node.color;  left = equated\_node.left;  right = equated\_node.right;  }  friend bool operator> (const node<T1, T2>& firstCompared, const node<T1, T2>& secondCompared) {  return firstCompared.key > secondCompared.key;  }  friend bool operator< (const node<T1, T2>& firstCompared, const node<T1, T2>& secondCompared) {  return firstCompared.key < secondCompared.key;  }  friend bool operator== (const node<T1, T2>& firstCompared, const node<T1, T2>& secondCompared) {  return firstCompared.key == secondCompared.key;  }  friend bool operator!= (const node<T1, T2>& firstCompared, const node<T1, T2>& secondCompared) {  return firstCompared.key != secondCompared.key;  }  ~node() {    }  T1 getKey() {  return key;  }  T2 getValue() {  return value;  }  void setParent(node<T1, T2>\* node\_parent) {  parent = node\_parent;  }  node<T1, T2>\* getParent() {  return parent;  }  void setLeftChild(node<T1, T2>\* leftChild) {  left = leftChild;  }  node<T1, T2>\* getLeftChild() {  return left;  }  void setRightChild(node<T1, T2>\* rightChild) {  right = rightChild;  }  node<T1, T2>\* getRightChild() {  return right;  }  bool isRed() {  return color;  }  void setColor(bool node\_color) {  color = node\_color;  }  void changeColor() {  color = !color;  }  void showInfo() {  cout << "Элемент с ключом " << key << " и значением " << value << (isRed() ? " красного " : " черного ") << "цвета, предок с ключом " << parent->key << ", левый ребенок с ключом " << left->key << ", правый ребенок с ключом " << right->key << endl;  }    void print(node<T1, T2>\* nil) {  showInfo();  if (left != nil) left->print(nil);  if (right != nil) right->print(nil);  }  };  #endif |
| map.h |
| #pragma once  #include "node.h"  #include "list.h"  using namespace std;  #ifndef MAP\_H  #define MAP\_H  template < class T1, class T2 >  class map {  private:  node<T1, T2>\* nil = new node<T1, T2>();  node<T1, T2>\* root;  public:  map() {  nil->setColor(black);  nil->setLeftChild(nil);  nil->setRightChild(nil);  nil->setParent(nil);  root = nil;  }  map(node<T1, T2>\* map\_root\_node) {  root = map\_root\_node;  }  ~map() {  clear();  }  node<T1, T2>\* getRoot() {  return root;  }  void leftRotation(node<T1, T2>\* x) {  node<T1, T2>\* y = x->getRightChild();  x->setRightChild(y->getLeftChild());  if (y->getLeftChild() != nil) y->getLeftChild()->setParent(x);  if (y != nil) y->setParent(x->getParent());  if (x->getParent() == nil && y != nil) root = y;  else if (x == x->getParent()->getLeftChild()) x->getParent()->setLeftChild(y);  else x->getParent()->setRightChild(y);  y->setLeftChild(x);  if (x != nil) x->setParent(y);  }  void rightRotation(node<T1, T2>\* x) {  node<T1, T2>\* y = x->getLeftChild();  x->setLeftChild(y->getRightChild());  if (y->getRightChild() != nil) y->getRightChild()->setParent(x);  if (y != nil) y->setParent(x->getParent());  if (x->getParent() == nil && y != nil) root = y;  else if (x == x->getParent()->getLeftChild()) x->getParent()->setLeftChild(y);  else x->getParent()->setRightChild(y);  y->setRightChild(x);  if (x != nil) x->setParent(y);  }  void insert(T1 node\_key, T2 node\_value) {  node<T1, T2>\* newNode = new node<T1, T2>(node\_key, node\_value, root, nil, nil);  node<T1, T2>\* y = nil;  node<T1, T2>\* x = root;  while (x != nil) {  y = x;  if (newNode < x) x = x->getLeftChild();  else x = x->getRightChild();  }  newNode->setParent(y);  if (y == nil) root = newNode;  else if (&newNode < &y) y->setLeftChild(newNode);  else y->setRightChild(newNode);  insFixup(newNode);  }  void insFixup(node<T1, T2>\* newNode) {  node<T1, T2>\* y = nullptr, \* x = nullptr;  while (newNode->getParent()->isRed()) {  if (newNode->getParent() == newNode->getParent()->getParent()->getLeftChild()) {  y = newNode->getParent()->getParent()->getRightChild();  if (y->isRed()) {  newNode->getParent()->setColor(black);  y->setColor(black);  newNode->getParent()->getParent()->setColor(red);  newNode = newNode->getParent()->getParent();  }  else {  if (newNode == newNode->getParent()->getRightChild()) {  newNode = newNode->getParent();  leftRotation(newNode);  }  newNode->getParent()->setColor(black);  newNode->getParent()->getParent()->setColor(red);  rightRotation(newNode->getParent()->getParent());  }  }  else {  if (newNode->getParent() == newNode->getParent()->getParent()->getRightChild()) {  y = newNode->getParent()->getParent()->getLeftChild();  if (y->isRed()) {  newNode->getParent()->setColor(black);  y->setColor(black);  newNode->getParent()->getParent()->setColor(red);  newNode = newNode->getParent()->getParent();  }  else {  if (newNode == newNode->getParent()->getLeftChild()) {  newNode = newNode->getParent();  rightRotation(newNode);  }  newNode->getParent()->setColor(black);  newNode->getParent()->getParent()->setColor(red);  leftRotation(newNode->getParent()->getParent());  }  }  }  }  root->setColor(black);  }  void delFixup(node<T1, T2>\* removeNode) {  node<T1, T2>\*w = removeNode;  while (removeNode != root && !removeNode->isRed()) {  if (removeNode != removeNode->getParent()->getLeftChild()) {  w = removeNode->getParent()->getRightChild();  if (w->isRed()) {  w->setColor(black);  removeNode->getParent()->setColor(red);  leftRotation(removeNode->getParent());  w = removeNode->getParent()->getRightChild();  }  if (!w->getLeftChild()->isRed() && !w->getRightChild()->isRed()) {  w->setColor(red);  removeNode = removeNode->getParent();  }  else {  if (!w->getRightChild()->isRed()) {  w->getLeftChild()->setColor(black);  w->setColor(red);  rightRotation(w);  w = removeNode->getParent()->getRightChild();  }  w->setColor(removeNode->getParent()->isRed());  removeNode->getParent()->setColor(black);  w->getRightChild()->setColor(black);  leftRotation(removeNode->getParent());  removeNode = root;  }  }  else {  w = removeNode->getParent()->getLeftChild();  if (w->isRed()) {  w->setColor(black);  removeNode->getParent()->setColor(red);  rightRotation(removeNode->getParent());  w = removeNode->getParent()->getLeftChild();  }  if (!w->getRightChild()->isRed() && !w->getLeftChild()->isRed()) {  w->setColor(red);  removeNode = removeNode->getParent();  }  else {  if (!w->getLeftChild()->isRed()) {  w->getRightChild()->setColor(black);  w->setColor(red);  leftRotation(w);  w = removeNode->getParent()->getLeftChild();  }  w->setColor(removeNode->getParent()->isRed());  removeNode->getParent()->setColor(black);  w->getLeftChild()->setColor(black);  rightRotation(removeNode->getParent());  removeNode = root;  }  }  }  removeNode->setColor(black);  }  node<T1, T2>\* minimum(node<T1, T2>\* x) {  while (x->getLeftChild() != nil) {  x = x->getLeftChild();  }  return x;  }  void transplant(node<T1, T2>\* u, node<T1, T2>\* v) {  if (u->getParent() == nil) root = v;  else  if (u = u->getParent()->getLeftChild()) u->getParent()->setLeftChild(v);  else u->getParent()->setRightChild(v);  v->setParent(u->getParent());  }  void remove(T1 remove\_key) {  node<T1, T2>\* removeNode = find(remove\_key);  if (removeNode != nil) {  node<T1, T2>\* y = removeNode;  node<T1, T2>\* x = nil;  bool yOrigColor = y->isRed();  if (removeNode->getLeftChild() == nil) {  x = removeNode->getRightChild();  transplant(removeNode, removeNode->getRightChild());  }  else {  if (removeNode->getRightChild() == nil) {  x = removeNode->getLeftChild();  transplant(removeNode, removeNode->getLeftChild());  }  else {  y = minimum(removeNode->getRightChild());  yOrigColor = y->isRed();  x = y->getRightChild();  if (y->getParent() == removeNode) x->setParent(y);  else {  transplant(y, y->getRightChild());  y->setRightChild(removeNode->getRightChild());  y->getRightChild()->setParent(y);  }  transplant(removeNode, y);  y->setLeftChild(removeNode->getLeftChild());  y->getLeftChild()->setParent(y);  y->setColor(removeNode->isRed());  }  }  if (yOrigColor == black) delFixup(removeNode);  }  }  node<T1, T2>\* find(T1 node\_key) {  node<T1, T2>\* findNode = root;  while (findNode->getKey() != node\_key && findNode != nil) {  if (findNode->getKey() > node\_key) findNode = findNode->getLeftChild();  else findNode = findNode->getRightChild();  }  return findNode;  }  void print(){  root->print(nil);  }  void clear() {  while (root != nil) remove(root->getKey());  }  void get\_keys(list<T1>\* temp) {  temp->clear();  if(root != nil)get\_keys(temp, root);  }  void get\_keys(list<T1>\* temp, node<T1, T2>\*cur) {  temp->push\_back(cur->getKey());  if (cur->getLeftChild() != nil) get\_keys(temp, cur->getLeftChild());  if (cur->getRightChild() != nil) get\_keys(temp, cur->getRightChild());  }  void get\_values(list<T2>\* temp) {  temp->clear();  if (root != nil)get\_values(temp, root);  }  void get\_values(list<T2>\* temp, node<T1, T2>\* cur) {  temp->push\_back(cur->getValue());  if (cur->getLeftChild() != nil) get\_values(temp, cur->getLeftChild());  if (cur->getRightChild() != nil) get\_values(temp, cur->getRightChild());  }  };  #endif |

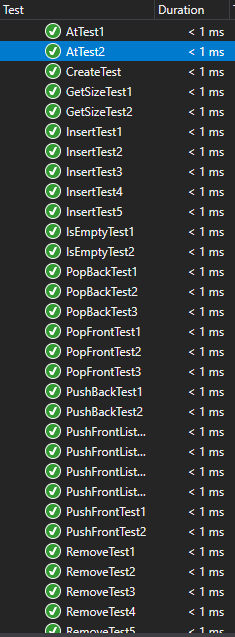
# Пример работы



mapUnitTest



listUnitTest



# Вывод

При выполнении данной лабораторной работы я познакомился с представлением красно-черного дерева в компьютере, а также изучил алгоритмы для работы с ним.