**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**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Кнауб К.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

# Постановка задачи

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

# Описание реализуемого класса и методов

Был реализован шаблонный класс RB\_TREE\_Node, который используется для создания ассоциативного массива на основе красно-черного дерева.

**Методы необходимые для реализации и дополнительные:**

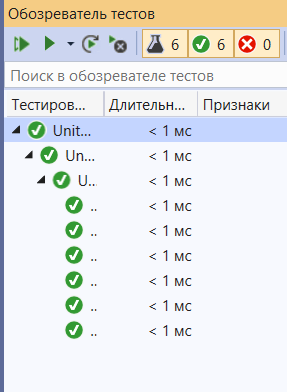
|  |  |
| --- | --- |
| **Название** | **Назначение** |
| void insert | добавление элемента с ключом и значением |
| void remove | удаление элемента дерева по ключу |
| Tree\_Node\* find | поиск элемента по ключу |
| void clear | очищение ассоциативного массива |
| void get\_keys | возвращает список ключей |
| void get\_values | возвращает список значений |
| void print | вывод в консоль |
| void Insert\_fixup | восстанавливает свойства дерева после вставки нового узла в дерево |
| void Transplant | перемещает поддеревья в дереве |
| void Remove\_fixup | восстанавливает свойства дерева после удаления узла из дерева |
| void NIL\_Root | создает новый объект NIL |
| void Left\_turn | метод левого поворота |
| void Right\_turn | метод правого поворота |
| Tree\_Node\* get\_Root | передаёт методам необходимых указателей |
| Tree\_Node\* get\_Parent | передает методам необходимых указателей |
| Tree\_Node\* Tree\_min | возвращает указатель на минимальный элемент |

# Оценка временной сложности каждого метода

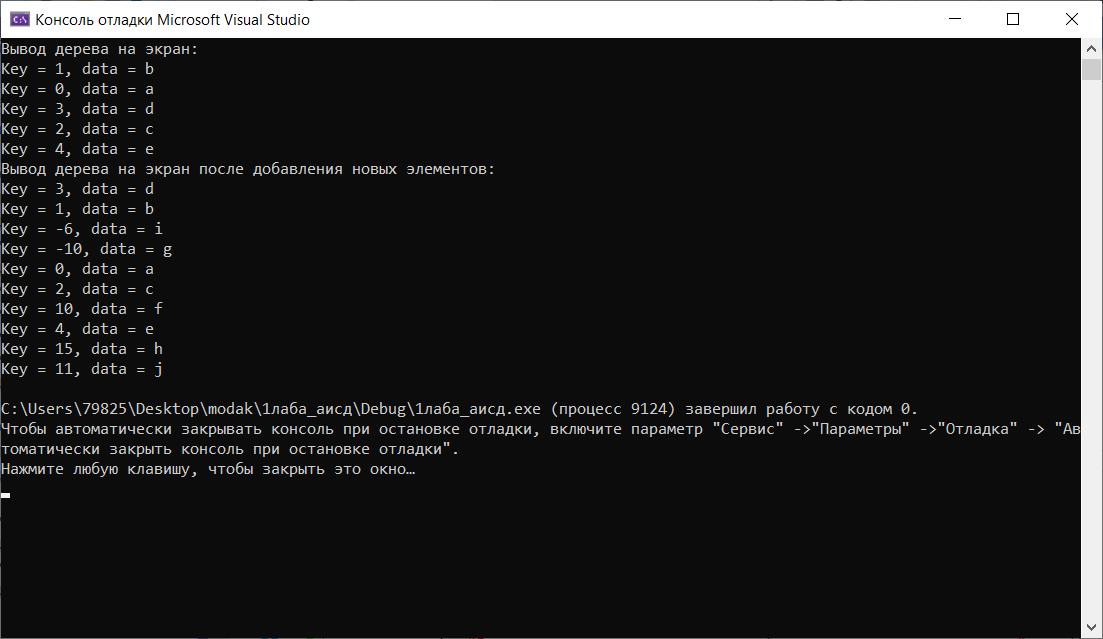
|  |  |
| --- | --- |
| void Insert | O(log n) |
| void Remove | O(log n) |
| Tree\_Node\* Find | O(log n) |
| void Clear | O(log n) |
| void get\_Keys | O(log n) |
| void get\_Values | O(log n) |
| void Print | O(log n) |
| void NIL\_Root | О(1) |
| void Left\_turn | O(1) |
| void Right\_turn | O(1) |
| void Insert\_fixup | O(log n) |
| void Transplant | O(1) |
| void Remove\_fixup | O(log n) |
| Tree\_Node\* get\_Root | O(1) |
| Tree\_Node\* get\_Parent | O(1) |
| Tree\_Node\* Tree\_min | O(log n) |

# Описание реализованных unit-тестов

|  |
| --- |
| Название тестов |
| Insert |
| Remove |
| Find |
| Get\_Keys |
| Get\_Values |
| Print |



# Пример работы



# Листинг

|  |
| --- |
| LinkedList.h |
| #pragma once  #pragma once  template<class NODE\_TYPE>  class LinkedList  {  public:  LinkedList();  ~LinkedList();  void push\_back(NODE\_TYPE newEl);  void push\_front(NODE\_TYPE newEl);  void pop\_back();  void pop\_front();  void insert(NODE\_TYPE newEl, size\_t index);  int at(size\_t index) const;  void remove(size\_t index);  size\_t get\_size();  void print\_to\_console();  void clear();  void set(size\_t index, NODE\_TYPE newData);  bool isEmpty();  size\_t find\_first(LinkedList\* list);  bool isPrinted();  private:  class Node  {  public:  Node(NODE\_TYPE data, Node\* next = nullptr, Node\* prev = nullptr)  {  this->data = data;  this->next = next;  this->prev = prev;  }  NODE\_TYPE data;  Node\* next;  Node\* prev;  };  Node\* head;  Node\* tail;  size\_t size;  };  #include "LinkedList.inl" |
| Tree.h |
| #pragma once  #pragma once  #include "LinkedList.h"  #include <iostream>  enum NodeColor { BLACK, RED };  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  class Tree\_Node  {  public:  ~Tree\_Node();  void Insert(NODE\_TYPE\_1 key, NODE\_TYPE\_2 data);  void Remove(NODE\_TYPE\_1 key);  Tree\_Node\* Find(NODE\_TYPE\_1 key, Tree\_Node\* node\_x);  void Clear(Tree\_Node\* tree);  void get\_Keys(Tree\_Node\* node\_x, LinkedList<NODE\_TYPE\_1>\* List\_of\_Keys);  void get\_Values(Tree\_Node\* node\_x, LinkedList<NODE\_TYPE\_2>\* List\_of\_Values);  void Print(Tree\_Node\* node\_x);    void NIL\_Root();  void Left\_turn(Tree\_Node\* node\_x);  void Right\_turn(Tree\_Node\* node\_y);  void Insert\_fixup(Tree\_Node\* node\_z);  void Transplant(Tree\_Node\* u, Tree\_Node\* v);  void Remove\_fixup(Tree\_Node\* node\_x);  Tree\_Node\* get\_Root() { return root; }  Tree\_Node\* get\_Parent() { return parent; }  Tree\_Node\* Tree\_min(Tree\_Node\* node\_x);  private:  Tree\_Node\* left, \* right, \* parent;  Tree\_Node\* root;  Tree\_Node\* NIL;  NodeColor color;  NODE\_TYPE\_1 key;  NODE\_TYPE\_2 data;  };  #include "Tree.inl" |
| Tree.cpp |
| #include "Tree.h"  #include "LinkedList.h"  #include <iostream>  using namespace std;  int main()  {  setlocale(LC\_ALL, "Rus");  Tree\_Node<int, char> Tree;  Tree.NIL\_Root();  char data[5] = { 'a', 'b','c','d','e' };  for (int i = 0; i < 5; i++)  Tree.Insert(i, data[i]);    cout << "Вывод дерева на экран: " << endl;  Tree.Print(Tree.get\_Root());  Tree.Insert(10, 'f');  Tree.Insert(-10, 'g');  Tree.Insert(15, 'h');  Tree.Insert(-6, 'i');  Tree.Insert(11, 'j');  cout<<"Вывод дерева на экран после добавления новых элементов: "<<endl;  Tree.Print(Tree.get\_Root());  return 0;  } |
| Tree.inl |
| #include "LinkedList.h"  #include "Tree.h"  #include <iostream>  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::NIL\_Root()  {  NIL = new Tree\_Node;  NIL->color = BLACK;  root = NIL;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::~Tree\_Node()  {  Clear(root);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>\* Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Tree\_min(Tree\_Node\* node\_x)  {  if (node\_x->get\_Parent() == NULL)  throw "Tree is empty";  while (node\_x->left != NIL)  node\_x = node\_x->left;  return node\_x;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Right\_turn(Tree\_Node\* node\_x)  {  Tree\_Node\* node\_y = node\_x->left;  node\_x->left = node\_y->right;  if (node\_y->right != NIL)  node\_y->right->parent = node\_x;  if (node\_y != NIL)  node\_y->parent = node\_x->parent;  if (node\_x->parent == NIL)  root = node\_y;  else if (node\_x == node\_x->parent->left)  node\_x->parent->left = node\_y;  else  node\_x->parent->right = node\_y;  node\_y->right = node\_x;  if (node\_x != NIL)  node\_x->parent = node\_y;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Left\_turn(Tree\_Node\* node\_x)  {  Tree\_Node\* node\_y = node\_x->right;  node\_x->right = node\_y->left;  if (node\_y->left != NIL)  node\_y->left->parent = node\_x;  if (node\_y != NIL)  node\_y->parent = node\_x->parent;  if (node\_x->parent == NIL)  root = node\_y;  else if (node\_x == node\_x->parent->left)  node\_x->parent->left = node\_y;  else  node\_x->parent->right = node\_y;  node\_y->left = node\_x;  if (node\_x != NIL)  node\_x->parent = node\_y;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Insert(NODE\_TYPE\_1 key, NODE\_TYPE\_2 data)  {  Tree\_Node\* node\_z = new Tree\_Node;  node\_z->key = key;  node\_z->data = data;  Tree\_Node\* node\_x = root;  Tree\_Node\* node\_y = NIL;  while (node\_x != NIL)  {  node\_y = node\_x;  if (node\_z->key < node\_x->key)  node\_x = node\_x->left;  else  node\_x = node\_x->right;  }  node\_z->parent = node\_y;  if (node\_y == NIL)  root = node\_z;  else if (node\_z->key < node\_y->key)  node\_y->left = node\_z;  else  node\_y->right = node\_z;  node\_z->left = NIL;  node\_z->right = NIL;  node\_z->color = RED;  Insert\_fixup(node\_z);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Insert\_fixup(Tree\_Node\* node\_z)  {  while (node\_z->parent->color == RED && node\_z != root)  {  if (node\_z->parent == node\_z->parent->parent->left)  {  Tree\_Node\* node\_y = node\_z->parent->parent->right;  if (node\_y->color == RED)  {  node\_z->parent->color = BLACK;  node\_y->color = BLACK;  node\_z->parent->parent->color = RED;  node\_z = node\_z->parent->parent;  }  else  {  if (node\_z == node\_z->parent->right)  {  node\_z = node\_z->parent;  Left\_turn(node\_z);  }  node\_z->parent->color = BLACK;  node\_z->parent->parent->color = RED;  Right\_turn(node\_z->parent->parent);  }  }  else  {  Tree\_Node\* node\_y = node\_z->parent->parent->left;  if (node\_y->color == RED)  {  node\_z->parent->color = BLACK;  node\_y->color = BLACK;  node\_z->parent->parent->color = RED;  node\_z = node\_z->parent->parent;  }  else  {  if (node\_z == node\_z->parent->left)  {  node\_z = node\_z->parent;  Right\_turn(node\_z);  }  node\_z->parent->color = BLACK;  node\_z->parent->parent->color = RED;  Left\_turn(node\_z->parent->parent);  }  }  }  root->color = BLACK;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Transplant(Tree\_Node\* node\_u, Tree\_Node\* node\_v)  {  if (node\_u->parent == NIL)  root = node\_v;  else if (node\_u == node\_u->parent->left)  node\_u->parent->left = node\_v;  else  node\_u->parent->right = node\_v;  node\_v->parent = node\_u->parent;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Remove(NODE\_TYPE\_1 key)  {  Tree\_Node\* node\_z = Find(key, root);  Tree\_Node\* node\_y = node\_z, \* node\_x;  NodeColor node\_y\_original\_color = node\_y->color;  if (node\_z->left == NIL)  {  node\_x = node\_z->right;  Transplant(node\_z, node\_z->right);  }  else if (node\_z->right == NIL)  {  node\_x = node\_z->left;  Transplant(node\_z, node\_z->left);  }  else  {  node\_y = Tree\_min(node\_z->right);  node\_y\_original\_color = node\_y->color;  node\_x = node\_y->right;  if (node\_y->parent == node\_z)  node\_x->parent = node\_y;  else  {  Transplant(node\_y, node\_y->right);  node\_y->right = node\_z->right;  node\_y->right->parent = node\_y;  }  Transplant(node\_z, node\_y);  node\_y->left = node\_z->left;  node\_y->left->parent = node\_y;  node\_y->color = node\_z->color;  }  if (node\_y\_original\_color == BLACK)  Remove\_fixup(node\_x);  }  template <class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Remove\_fixup(Tree\_Node\* node\_x)  {  while (node\_x != root && node\_x->color == BLACK)  {  if (node\_x == node\_x->parent->left)  {  Tree\_Node\* node\_w = node\_x->parent->right;  if (node\_w->color == RED)  {  node\_w->color = BLACK;  node\_x->parent->color = RED;  Left\_turn(node\_x->parent);  node\_w = node\_x->parent->right;  }  if (node\_w->left->color == BLACK && node\_w->right->color == BLACK)  {  node\_w->color = RED;  node\_x = node\_x->parent;  }  else  {  if (node\_w->right->color == BLACK)  {  node\_w->left->color = BLACK;  node\_w->color = RED;  Right\_turn(node\_w);  node\_w = node\_x->parent->right;  }  node\_w->color = node\_x->parent->color;  node\_x->parent->color = BLACK;  node\_w->right->color = BLACK;  Left\_turn(node\_x->parent);  node\_x = root;  }  }  else  {  Tree\_Node\* node\_w = node\_x->parent->left;  if (node\_w->color == RED)  {  node\_w->color = BLACK;  node\_x->parent->color = RED;  Right\_turn(node\_x->parent);  node\_w = node\_x->parent->left;  }  if (node\_w->right->color == BLACK && node\_w->left->color == BLACK)  {  node\_w->color = RED;  node\_x = node\_x->parent;  }  else  {  if (node\_w->left->color == BLACK)  {  node\_w->right->color = BLACK;  node\_w->color = RED;  Left\_turn(node\_w);  node\_w = node\_x->parent->left;  }  node\_w->color = node\_x->parent->color;  node\_x->parent->color = BLACK;  node\_w->left->color = BLACK;  Right\_turn(node\_x->parent);  node\_x = root;  }  }  }  node\_x->color = BLACK;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>\* Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Find(NODE\_TYPE\_1 key, Tree\_Node\* node\_x)  {  if (node\_x->get\_Parent() == NULL)  throw "Element not found";  if (node\_x == NIL || node\_x->key == key)  return node\_x;  if (key < node\_x->key)  return Find(key, node\_x->left);  else  return Find(key, node\_x->right);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Clear(Tree\_Node\* tree)  {  if (tree == NIL)  return;  else  {  if (tree->left)  Clear(tree->left);  if (tree->right)  Clear(tree->right);  }  delete tree;  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::get\_Keys(Tree\_Node\* node\_x, LinkedList<NODE\_TYPE\_1>\* List\_of\_Keys)  {  if (node\_x->get\_Parent() == NULL)  throw "Tree is empty";  if (node\_x == NIL)  return;  else  {  List\_of\_Keys->push\_back(node\_x->key);  if (node\_x->left != NIL)  get\_Keys(node\_x->left, List\_of\_Keys);  }  if (node\_x->right != NIL)  get\_Keys(node\_x->right, List\_of\_Keys);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::get\_Values(Tree\_Node\* node\_x, LinkedList<NODE\_TYPE\_2>\* List\_of\_Values)  {  if (node\_x->get\_Parent() == NULL)  throw "Tree is empty";  if (node\_x == NIL)  return;  else  {  List\_of\_Values->push\_back(node\_x->data);  if (node\_x->left != NIL)  get\_Values(node\_x->left, List\_of\_Values);  }  if (node\_x->right != NIL)  get\_Values(node\_x->right, List\_of\_Values);  }  template<class NODE\_TYPE\_1, class NODE\_TYPE\_2>  void Tree\_Node<NODE\_TYPE\_1, NODE\_TYPE\_2>::Print(Tree\_Node\* node\_x)  {  if (node\_x->get\_Parent() == NULL)  throw "Tree is empty";  if (node\_x == NIL)  return;  else  {  std::cout << "Key = " << node\_x->key << ", data = " << node\_x->data << std::endl;  if (node\_x->left != NIL)  Print(node\_x->left);  }  if (node\_x->right != NIL)  Print(node\_x->right);  } |
| LinkedList.inl |
| #include "LinkedList.h"  #include <iostream>  #include <stdexcept>  #include <string>  using namespace std;  template<class NODE\_TYPE>  LinkedList<NODE\_TYPE>::LinkedList()  {  head = nullptr;  tail = nullptr;  size = 0;  }  template<class NODE\_TYPE>  LinkedList<NODE\_TYPE>::~LinkedList()  {  clear();  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::push\_back(NODE\_TYPE newEl)  {  Node\* temp;  if (size == 0)  {  head = new Node(newEl);  tail = head;  }  else  {  temp = tail;  tail->next = new Node(newEl);  tail = tail->next;  tail->prev = temp;  }  size++;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::push\_front(NODE\_TYPE newEl)  {  Node\* temp;  if (size == 0)  {  head = new Node(newEl);  tail = head;  }  else  {  temp = head;  head = new Node(newEl, head);  temp->prev = head;  }  size++;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::pop\_back()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = nullptr;  tail = nullptr;  }  else  {  Node\* cur = tail;  cur = cur->prev;  cur->next = nullptr;  delete tail;  tail = cur;  cur->prev = tail->prev;  }  size--;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::pop\_front()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = nullptr;  tail = nullptr;  }  else  {  Node\* cur = head;  cur = cur->next;  cur->prev = nullptr;  delete head;  head = cur;  }  size--;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::insert(NODE\_TYPE newEl, size\_t index)  {  Node\* temp\_pointer = nullptr, \* cur;  Node\* temp = new Node(newEl);  cur = head;  int i = 0;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  while (cur != NULL && i<int(index))  {  cur = cur->next;  i++;  }  if (cur == NULL)  {  if (head == NULL)  {  cout << "List is empty\n";  return;  }  return;  }  if (cur != head)  {  temp\_pointer = cur->prev;  cur->prev = temp;  temp->next = cur;  temp->prev = temp\_pointer;  }  else  push\_front(newEl);  if (temp\_pointer != NULL)  temp\_pointer->next = temp;  size++;  }  template<class NODE\_TYPE>  int LinkedList<NODE\_TYPE>::at(size\_t index) const  {  if (size == 0)  cout << "List is empty\n";  if (index >= size || index < 0) {  throw out\_of\_range("Index is greater than list size");  }  if ((index < size) && (size != 0))  {  size\_t counter = 0;  Node\* cur = head;  while (counter != index)  {  cur = cur->next;  counter++;  }  return cur->data;  }  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::remove(size\_t index)  {  size\_t counter = 0;  Node\* cur = head;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  while (counter != index)  {  cur = cur->next;  counter++;  }  if (cur == head)  {  pop\_front();  return;  }  if (cur == tail)  {  pop\_back();  return;  }  Node\* prev, \* next;  prev = cur->prev;  next = cur->next;  if (prev != NULL)  prev->next = cur->next;  if (next != NULL)  next->prev = cur->prev;  delete cur;  size--;  }  template<class NODE\_TYPE>  size\_t LinkedList<NODE\_TYPE>::get\_size()  {  return size;  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::print\_to\_console()  {  Node\* cur = head;  cout << "List elements:\t";  while (cur != nullptr)  {  cout << cur->data << ' ';  cur = cur->next;  }  cout << '\n';  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::clear()  {  while (head)  {  tail = head->next;  delete head;  head = tail;  size--;  }  }  template<class NODE\_TYPE>  void LinkedList<NODE\_TYPE>::set(size\_t index, NODE\_TYPE newData)  {  Node\* cur = head;  if (index >= size || index < 0)  throw out\_of\_range("Index is greater than list size");  size\_t counter = 1;  while (counter <= index)  {  counter++;  cur = cur->next;  }  cout << "Element " << cur->data;  cur->data = newData;  cout << " replaced by " << cur->data << '\n';  }  template<class NODE\_TYPE>  bool LinkedList<NODE\_TYPE>::isEmpty()  {  if (head != NULL)  return true;  else  return false;  }  template<class NODE\_TYPE>  bool LinkedList<NODE\_TYPE>::isPrinted()  {  Node\* cur = head;  while (cur != nullptr)  {  cout << cur->data << ' ';  cur = cur->next;  }  cout << '\n';  if (cur == nullptr)  return true;  else  return false;  }  template<class NODE\_TYPE>  size\_t LinkedList<NODE\_TYPE>::find\_first(LinkedList\* list)  {  int counter = 0;  Node\* curL1, \* curL2;  curL1 = (\*list).head;  curL2 = head;  if (curL1 == nullptr || curL2 == nullptr)  return 0;  if ((\*list).get\_size() > size)  {  while (curL2->data != curL1->data && curL1->next != NULL)  {  curL1 = curL1->next;  counter++;  }  while (curL2->data == curL1->data && curL2->next != NULL && curL1->next != NULL)  {  curL2 = curL2->next;  curL1 = curL1->next;  }  if (curL2->next == NULL)  {  return counter;  }  else  return 0;  }  else  {  if ((\*list).get\_size() == size)  if (curL1->data == curL2->data)  goto analysis;  else  return 0;  while (curL1->data != curL2->data && curL2->next != NULL)  {  curL2 = curL2->next;  counter++;  }  analysis:  while (curL1->data == curL2->data && curL1->next != NULL && curL2->next != NULL)  {  curL1 = curL1->next;  curL2 = curL2->next;  }  if (curL1->next == NULL)  {  return counter;  }  else  return 0;  }  } |
| UnitTest1 |
| #include "CppUnitTest.h"  #include "C:\Users\79825\Desktop\modak\1лаба\_аисд\LinkedList.h"  #include "C:\Users\79825\Desktop\modak\1лаба\_аисд\Tree.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest1  {  TEST\_CLASS(UnitTest1)  {  public:  TEST\_METHOD(Insert)  {  Tree\_Node<int, int> tree;  LinkedList<int> check, test;  tree.NIL\_Root();  check.push\_back(1);  check.push\_back(0);  check.push\_back(2);  tree.Insert(1, 0);  tree.Insert(2, 0);  tree.Insert(0, 0);  tree.get\_Keys(tree.get\_Root(), &test);  Assert::AreEqual(test.at(0), check.at(0));  Assert::AreEqual(test.at(1), check.at(1));  Assert::AreEqual(test.at(2), check.at(2));  }  TEST\_METHOD(Remove)  {  Tree\_Node<int, int> tree;  LinkedList<int> check, test;  tree.NIL\_Root();  check.push\_back(2);  check.push\_back(0);  tree.Insert(1, 0);  tree.Insert(2, 0);  tree.Insert(0, 0);  tree.Remove(1);  tree.get\_Keys(tree.get\_Root(), &test);  Assert::AreEqual(test.at(0), check.at(0));  Assert::AreEqual(test.at(1), check.at(1));  }  TEST\_METHOD(Find)  {  Tree\_Node<int, int> tree;  Tree\_Node<int, int>\* test\_tree;  LinkedList<int> check, test;  tree.NIL\_Root();  check.push\_back(2);  try  {  tree.Insert(1, 0);  tree.Insert(2, 0);  tree.Insert(0, 0);  test\_tree = tree.Find(2, tree.get\_Root());  test\_tree->get\_Keys(test\_tree, &test);  Assert::AreEqual(test.at(0), check.at(0));  }  catch (const char\* exception)  {  Assert::AreEqual(exception, "RB Tree is empty");  }  }  TEST\_METHOD(get\_Keys)  {  Tree\_Node<int, int> tree;  LinkedList<int> check, test;  tree.NIL\_Root();  check.push\_back(1);  check.push\_back(0);  check.push\_back(2);  tree.Insert(1, 0);  tree.Insert(2, 0);  tree.Insert(0, 0);  tree.get\_Keys(tree.get\_Root(), &test);  Assert::AreEqual(test.at(0), check.at(0));  Assert::AreEqual(test.at(1), check.at(1));  Assert::AreEqual(test.at(2), check.at(2));  }  TEST\_METHOD(get\_Values)  {  Tree\_Node<int, int> tree;  LinkedList<int> check, test;  tree.NIL\_Root();  check.push\_back(89);  check.push\_back(47);  check.push\_back(63);  tree.Insert(1, 89);  tree.Insert(2, 63);  tree.Insert(0, 47);  tree.get\_Values(tree.get\_Root(), &test);  Assert::AreEqual(test.at(0), check.at(0));  Assert::AreEqual(test.at(1), check.at(1));  Assert::AreEqual(test.at(2), check.at(2));  }    TEST\_METHOD(Print)  {  Tree\_Node<int, int> tree;  tree.NIL\_Root();  try  {  tree.Print(tree.get\_Root());  }  catch (const char\* exception)  {  Assert::AreEqual(exception, "RB Tree is empty");  }  }    };  } |