**МИНОБРНАУКИ РОССИИ САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В. И. УЛЬЯНОВА (ЛЕНИНА)**

Кафедра САПР

**ОТЧЕТ**

**по лабораторной работе №1**

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

**Тема: «Ассоциативный массив»**

**Вариант 1**

Студент гр. 9302 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Гелета А.И.

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

Санкт-Петербург

2021

# Постановка задачи

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

# Описание реализуемого класса и методов

class RedBlackTree

Класс ассоциативного массива.

Поля:

Node\* root – корень дерева.

Node\* nil – нулевой элемент

int size – размер массива.

Методы:

insert(ключ, значение) – добавление элемента с ключом и значением.

remove(ключ) – удаление элемента дерева по ключу.

find(ключ) – поиск элемента по ключу.

clear – очищение ассоциативного массива.

get\_keys – возвращает список ключей.

get\_values – возвращает список значений.

print – вывод в консоль.

class Node

Элемент массива.

Поля:

K key – ключ ассоциативного массива (template class K).

V value – значение ассоциативного массива (template class V).

Node\* parent – «отец» элемента.

Node\* left – левый «ребенок».

Node\* right – правый «ребенок».

bool color – цвет узла.

class dft\_iterator

Класс оператора обхода в глубину.

Поля:

int\* curr – сама куча.

int size – размер кучи.

int cur\_index – текущие индекс.

DuoList right\_stack – дополнительный «стек», необходимый для осуществления обхода.

# Оценка временной сложности каждого метода

|  |  |
| --- | --- |
| Метод | Сложность |
| insert | O(logn) |
| find | O(logn) |
| remove | O(logn) |
| clear | O(n) |
| get\_keys | O(n) |
| get\_values | O(n) |
| print | O(n) |

# Описание реализованных unit-тестов

TEST\_CLASS(Empty) – класс для проверки изначально пустого массива.

insert\_empty – вставка в пустой массив.

insert1 – проверка вставки

insert2 – еще одна проверка вставки.

remove\_error – проверка попытки удалить из пустого массива.

size – проверка размера.

TEST\_CLASS(NotEmpty) – класс для проверки заранее инициализированного массива.

size – проверка размера не пустого массива.

clear – очистка.

find\_error – ошибка при поиске.

find1 – проверка работы поиска.

find2 – еще проверка поиска.

get\_values – проверка одноименного метода.

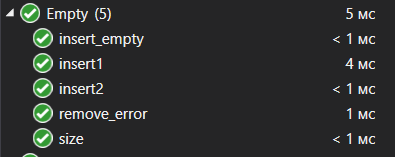
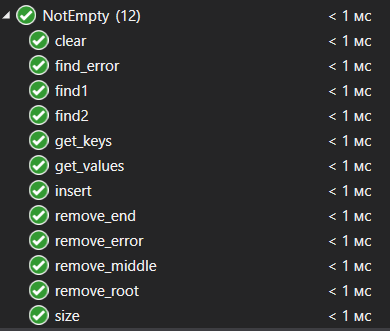
get\_keys – проверка одноименного метода.

insert – вставка.

remove\_end – проверка удаления из конца (листа).

remove\_error – ошибка при удалении.

remove\_middle – удаление из середины (элемента с двумя детьми).

remove\_root – удаление корневого элемента.

# Текст программы

|  |  |
| --- | --- |
|  | **RBtree.h** |
|  | #pragma once  #include"Stack.h"  #include"Iterator.h"  template <class K, class V>  class RedBlackTree {  private:  class Node {  public:  K key;  V value;  Node\* parent;  Node\* left;  Node\* right;  bool color; //0 - red, 1 - black  Node(K key, V value, Node\* left = nullptr, Node\* right = nullptr, Node\* parent = nullptr, bool color = 0) {  this->key = key;  this->value = value;  this->left = left;  this->right = right;  this->parent = parent;  this->color = color;  }  ~Node() {};  //friend class RedBlackTree;  };  class dft\_Iterator // depth-first traverse  {  private:  Stack<Node\*> stack;  Node\* Icurrent;  Node\* nil;  public:  dft\_Iterator(Node\* cur, Node\* nill) {  Icurrent = cur;  nil = nill;  };  Node\* next() {  Node\* temp = Icurrent;  if (Icurrent->right != nil) stack.push(Icurrent->right);  if (Icurrent->left != nil) {  Icurrent = Icurrent->left;  }  else {  if (!stack.isEmpty()) Icurrent = stack.pop();  else Icurrent = nil;  }  return temp;  };  bool has\_next() {  if (Icurrent != nil)  return true;  return false;  };  ~dft\_Iterator() {  }  };  Node\* root;  Node\* nil;  int size;  void recovery(Node\*); //recover properties after input  void leftRotate(Node\*);  void rightRotate(Node\*);  void recoveryRemove(Node\* x); //recover properties after remove  public:  RedBlackTree() {  nil = new Node(0, 0, nullptr, nullptr, nullptr, 1);  root = nil;  size = 0;  }  ~RedBlackTree() { clear(); delete root; }  void insert(K, V); //add element with key and value  void remove(K); //remove element by key  V find(K); //find element by key  void clear(); //clear all  K\* get\_keys(); //return all keys  V\* get\_values(); //return values  void print(); //print in console  bool get\_color(K); //for tests  int get\_size() { return size; }  friend class Node;  };  template<typename K, typename V>  void RedBlackTree<K, V>::insert(K key, V value) {  Node\* cur = root, \* parent;  bool moveRight;  size++;  if (root == nil) { root = new Node(key, value, nil, nil, nil, 1); return; } //Tree is empty  parent = cur;  while (cur != nil) {  parent = cur;  if (key > cur->key) {  cur = cur->right; moveRight = true;  } //right  else {  cur = cur->left; moveRight = false;  } //left  }  if (moveRight) {  parent->right = new Node(key, value, nil, nil, parent, 0);  recovery(parent->right);  }  else {  parent->left = new Node(key, value, nil, nil, parent, 0);  recovery(parent->left);  }  }  template<typename K, typename V>  void RedBlackTree<K, V>::recovery(Node\* newNode) {  Node\* parent, \* grandParent;  while (newNode->parent->color == 0) {  parent = newNode->parent;  grandParent = parent->parent;  if (grandParent->left == parent) {  //3 cases  if (grandParent->right->color == 0) {  //I  grandParent->right->color = 1;  grandParent->left->color = 1;  grandParent->color = 0;  newNode = grandParent;  }  else {  if (parent->right == newNode) {  //II  newNode = parent;  leftRotate(newNode);  }  //III  newNode->parent->color = 1;  newNode->parent->parent->color = 0;  rightRotate(newNode->parent->parent);  }  }  else {  if (grandParent->left->color == 0) {  //I  grandParent->right->color = 1;  grandParent->left->color = 1;  grandParent->color = 0;  newNode = grandParent;  }  else {  if (parent->left == newNode) {  //II  newNode = parent;  rightRotate(newNode);  }  //III  newNode->parent->color = 1;  newNode->parent->parent->color = 0;  leftRotate(newNode->parent->parent);  }  }  }  root->color = 1;  }  template<typename K, typename V>  void RedBlackTree<K, V>::leftRotate(Node\* curx) {  if (curx->right == nil) return;  Node\* cury = curx->right;  if (cury->left != nil) {  curx->right = cury->left;  cury->left->parent = curx;  }  else curx->right = nil;  if (cury != nil) cury->parent = curx->parent;  if (curx->parent != nil) {  if (curx == curx->parent->left)  curx->parent->left = cury;  else  curx->parent->right = cury;  }  else {  cury->parent = nil;  root = cury;  }  // connect curx and cury  cury->left = curx;  if (curx != nil) curx->parent = cury;    }  template<typename K, typename V>  void RedBlackTree<K, V>::rightRotate(Node\* curx) {  Node\* cury = curx->left;  curx->left = cury->right;  if (cury->right != nil) cury->right->parent = curx;  if (cury != nil) cury->parent = curx->parent;  if (curx->parent != nil) {  if (curx == curx->parent->right)  curx->parent->right = cury;  else  curx->parent->left = cury;  }  else {  root = cury;  }  cury->right = curx;  if (curx != nil) curx->parent = cury;  }  template<typename K, typename V>  void RedBlackTree<K, V>::remove(K key) {  if (root == nil) throw std::out\_of\_range("Tree is empty");  Node\* nodeToDelete = root;  while (nodeToDelete->key != key) {  if (key > nodeToDelete->key) nodeToDelete = nodeToDelete->right;  else nodeToDelete = nodeToDelete->left;  if (nodeToDelete == nil) throw std::out\_of\_range("No such element"); //error  }  Node\* x, \*min;  bool originalColor = nodeToDelete->color;  if (nodeToDelete->left == nil) {  //only one branch, deleting nodeToDelete  //putting his child instead of him  x = nodeToDelete->right;  if (nodeToDelete->parent == nil) root = x;  else if (nodeToDelete->parent->left == nodeToDelete) nodeToDelete->parent->left = x;  else nodeToDelete->parent->right = x;  x->parent = nodeToDelete->parent;  }  else if (nodeToDelete->right == nil) {  //same as before - other branch is existing  x = nodeToDelete->left;  if (nodeToDelete->parent == nil) root = x;  else if (nodeToDelete->parent->left == nodeToDelete) nodeToDelete->parent->left = x;  else nodeToDelete->parent->right = x;  x->parent = nodeToDelete->parent;  }  else {  Node\* temp = nodeToDelete;  min = nodeToDelete->right;  while (min->left != nil) min = min->left;  originalColor = min->color;  nodeToDelete = min;  x = nodeToDelete->right;  if (nodeToDelete->parent == temp) x->parent = nodeToDelete;  else {  //swap NodetoDelete and NodetoDelete->right (x)  if (nodeToDelete->parent == nil) {  root = x;  nodeToDelete->right->parent = nil;  }  else if (nodeToDelete->parent->right == nodeToDelete) nodeToDelete->parent->right = x;  else nodeToDelete->parent->left = x;  x->parent = nodeToDelete->parent;  nodeToDelete->right = temp->right;  nodeToDelete->right->parent = nodeToDelete;  }  //swap min and 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 (originalColor == 1) recoveryRemove(x);  size--;  }  template<typename K, typename V>  void RedBlackTree<K, V>::recoveryRemove(Node\* x) {  Node\* brother;  while (x != root && x->color == 1) {  if (x == x->parent->left) {  brother = x->parent->right;  if (brother->color == 0) {  brother->color = 1;  x->parent->color = 0;  leftRotate(x->parent);  brother = x->parent->right;  }  if (brother->left->color == 1 && brother->right->color == 1) {  brother->color = 0;  x = x->parent;  }  else {  if (brother->right->color == 1) {  brother->left->color = 1;  brother->color = 0;  rightRotate(brother);  brother = x->parent->right;  }  brother->color = x->parent->color;  x->parent->color = 1;  brother->right->color = 1;  leftRotate(x->parent);  x = root;  }  }  else {  brother = x->parent->left;  if (brother->color == 0) {  brother->color = 1;  x->parent->color = 0;  rightRotate(x->parent);  brother = x->parent->left;  }  if (brother->right->color == 1 && brother->left->color == 1) {  brother->color = 0;  x = x->parent;  }  else {  if (brother->left->color == 1) {  brother->right->color = 1;  brother->color = 0;  leftRotate(brother);  brother = x->parent->left;  }  brother->color = x->parent->color;  x->parent->color = 1;  brother->left->color = 1;  rightRotate(x->parent);  x = root;  }  }  }  x->color = 1;  }  template<typename K, typename V>  void RedBlackTree<K, V>::clear() {  while (root != nil) {  remove(root->key); std::cout << std::endl; print();  }  }  template<typename K, typename V>  V RedBlackTree<K, V>::find(K findKey) {  Node\* cur = root;  while (cur->key != findKey) {  if (findKey > cur->key) cur = cur->right;  else cur = cur->left;  if (cur == nil) throw std::out\_of\_range("No such element"); //error  }  return cur->value;  }  template<typename K, typename V>  K\* RedBlackTree<K, V>::get\_keys() {  K\* arr = new K[size]; int i = 0;  dft\_Iterator iter(root, nil);  while (iter.has\_next())  {  arr[i] = iter.next()->key; i++;  }  return arr;  }  template<typename K, typename V>  V\* RedBlackTree<K, V>::get\_values() {  V\* arr = new V[size]; int i = 0;  dft\_Iterator iter(root, nil);  while (iter.has\_next())  {  arr[i] = iter.next()->value; i++;  }  return arr;  }  template<typename K, typename V>  void RedBlackTree<K, V>::print() {  dft\_Iterator iter(root, nil);  Node\* cur;  while (iter.has\_next())  {  cur = iter.next();  std::cout << "\nKey: " << cur->key << " Color: " << cur->color;  std::cout << " Kids " << cur->left->key << " " << cur->right->key;  }  }  template<typename K, typename V>  bool RedBlackTree<K, V>::get\_color(K findKey) {  Node\* cur = root;  while (cur->key != findKey) {  if (findKey > cur->key) cur = cur->right;  else cur = cur->left;  if (cur == nil) throw std::out\_of\_range("No such element"); //error  }  return cur->color;  }  #include "Stack.cpp" |
|  | **4sem1\_test.cpp** |
|  | #include "pch.h"  #include "CppUnitTest.h"  #include "..\4sem1\RBtree.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace My4sem1test  {  //true - black, false - red  TEST\_CLASS(Empty)  {  public:    TEST\_METHOD(insert\_empty)  {  RedBlackTree<int, char> tree;  tree.insert(3, 'c');  Assert::AreEqual(tree.get\_color(3), true);  }  TEST\_METHOD(insert1) {  RedBlackTree<char, int> tree;  tree.insert('a', 4);  tree.insert('f', 215);  Assert::AreEqual(tree.get\_color('a'), true);  Assert::AreEqual(tree.get\_color('f'), false);  }  TEST\_METHOD(insert2) {  RedBlackTree<int, char> tree;  for (int i = 1; i < 7; i++) {  tree.insert(i, (char)i + 56);  }  Assert::AreEqual(tree.get\_color(1), true);  Assert::AreEqual(tree.get\_color(2), true);  Assert::AreEqual(tree.get\_color(3), true);  Assert::AreEqual(tree.get\_color(4), false);  Assert::AreEqual(tree.get\_color(5), true);  Assert::AreEqual(tree.get\_color(6), false);  }  TEST\_METHOD(remove\_error) {  RedBlackTree<int, char> tree;  try {  tree.remove(1);  }  catch (const exception& e) {  Assert::AreEqual(e.what(), "Tree is empty");  }  }  TEST\_METHOD(size) {  RedBlackTree<int, char> tree;  Assert::AreEqual(tree.get\_size(), 0);  }  };  TEST\_CLASS(NotEmpty) {  RedBlackTree<int, char> tree;  TEST\_METHOD\_INITIALIZE(setup) {  for (int i = 1; i < 7; i++) {  tree.insert(i, (char)i + 56);  }  }  TEST\_METHOD(size) {  Assert::AreEqual(tree.get\_size(), 6);  }  TEST\_METHOD(find\_error) {  try {  tree.find(123);  }  catch (exception e) {  Assert::AreEqual(e.what(), "No such element");  }  }  TEST\_METHOD(find1) {  Assert::AreEqual(tree.find(3), (char)59);  }  TEST\_METHOD(find2) {  Assert::AreEqual(tree.find(4), (char)60);  }  TEST\_METHOD(insert) {  tree.insert(0, 'g');  Assert::AreEqual(tree.get\_size(), 7);  Assert::AreEqual(tree.get\_color(0), false);  }  TEST\_METHOD(get\_keys) {  int\* arr = new int[tree.get\_size()];  arr = tree.get\_keys();  Assert::AreEqual(arr[0], 2);  Assert::AreEqual(arr[1], 1);  Assert::AreEqual(arr[2], 4);  Assert::AreEqual(arr[3], 3);  Assert::AreEqual(arr[4], 5);  Assert::AreEqual(arr[5], 6);  }  TEST\_METHOD(get\_values) {  char\* arr = new char[tree.get\_size()];  arr = tree.get\_values();  Assert::AreEqual(arr[0], (char)58);  Assert::AreEqual(arr[1], (char)57);  Assert::AreEqual(arr[2], (char)60);  Assert::AreEqual(arr[3], (char)59);  Assert::AreEqual(arr[4], (char)61);  Assert::AreEqual(arr[5], (char)62);  }  TEST\_METHOD(remove\_error) {  try {  tree.remove(123);  }  catch (exception e) {  Assert::AreEqual(e.what(), "No such element");  }  }  TEST\_METHOD(remove\_end) {  tree.remove(6);  Assert::AreEqual(tree.get\_size(), 5);  Assert::AreEqual(tree.get\_color(1), true);  Assert::AreEqual(tree.get\_color(2), true);  Assert::AreEqual(tree.get\_color(3), true);  Assert::AreEqual(tree.get\_color(4), false);  Assert::AreEqual(tree.get\_color(5), true);  }  TEST\_METHOD(remove\_middle) {  tree.remove(4);  Assert::AreEqual(tree.get\_size(), 5);  Assert::AreEqual(tree.get\_color(1), true);  Assert::AreEqual(tree.get\_color(2), true);  Assert::AreEqual(tree.get\_color(3), true);  Assert::AreEqual(tree.get\_color(5), false);  Assert::AreEqual(tree.get\_color(6), true);  }  TEST\_METHOD(remove\_root) {  tree.remove(2);  Assert::AreEqual(tree.get\_size(), 5);  Assert::AreEqual(tree.get\_color(1), true);  Assert::AreEqual(tree.get\_color(3), true);  Assert::AreEqual(tree.get\_color(4), true);  Assert::AreEqual(tree.get\_color(5), false);  Assert::AreEqual(tree.get\_color(6), true);  }  TEST\_METHOD(clear) {  tree.clear();  Assert::AreEqual(tree.get\_size(), 0);  }  };  } |