**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 \_\_\_\_\_\_\_\_\_\_\_\_ Ковтун А.С.

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

Санкт-Петербург

2021

* **Постановка задачи и описание реализуемого класса и методов**

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

Список методов:

1. insert(ключ, значение) // добавление элемента с ключом и значением
2. remove(ключ) // удаление элемента дерева по ключу
3. find(ключ) // Поиск элемента по ключу
4. clear // Очищение ассоциативного массива
5. get\_keys // Возвращает список ключей
6. get\_values // Возвращает список значений
7. print // Вывод в консоль

|  |  |  |
| --- | --- | --- |
| **Метод** | **Описание** | **Оценка временной сложности** |
| insert(ключ, значение) | Добавление элемента с ключом и значением | O(log(n)) |
| remove(ключ) | Удаление элемента дерева по ключу | O(log(n)) |
| find(ключ) | Поиск элемента по ключу | O(log(n)) |
| clear | Очищение ассоциативного массива | O(n) |
| get\_keys | Возвращает список ключей | O(n) |
| get\_values | Возвращает список значений | O(n) |
| print | Вывод в консоль | O(n) |

* **Описание реализованных unit-тестов**

|  |  |
| --- | --- |
| Имя теста | Описание |
| clear | Проверка очищения массива |
| Insert1 | Проверка добавления элемента |
| Insert2 | Проверка добавления элемента |
| Remove1 | Проверка удаления элемента дерева по ключу |
| Remove2 | Проверка удаления элемента дерева по ключу |

* **Код программы**

|  |
| --- |
| Stack.h  #pragma once  #include <iostream>  template <class T>  class Stack  {  private:  class Node  {  public:  Node(T content = 0, Node\* before = NULL, Node\* next = NULL)  {  this->content = content;  this->before = before;  this->next = next;  };  ~Node()  { }  T content;  Node\* next;  Node\* before;  private:  };  Node\* head; //head list  Node\* tail; //tail list  size\_t size; //list size  public:  Stack(Node\* head = NULL, Node\* tail = NULL) {  this->head = head;  this->tail = tail;  size = 0;  };  void push(T); // add item to end  size\_t get\_size(); // get list size  T pop(); // delete last item  void clear(); // clear list  bool isEmpty(); // test for emptiness  ~Stack()  {  if (head != NULL)  {  while (head->next != NULL) //while we can go next  {  head = head->next;  delete head->before;  }  size = 0;  delete head;  }  };  }; |
| 1LAB.cpp |
| #include <iostream>  #include "RBTree.h"  int main()  {  RBTree<int> \*RB = new RBTree<int>();  RB->insert(1, 1);  RB->insert(2, 2);  RB->print();  RB->remove(1);  RB->print();  RB->insert(3, 3);  RB->print();  } |
| Stack.cpp |
| #include "Stack.h"  using namespace std;  template <class T>  void Stack<T>::push(T add)  {  if (size == 0) //if list is empty  {  head = new Node(add); //create head  tail = head;  }  else  {  Node\* curr = new Node(add);  curr->before = tail;  tail->next = curr;  tail = curr; //set new tail  }  size++;  }  template <class T>  size\_t Stack<T>::get\_size()  {  return size;  }  template <class T>  T Stack<T>::pop()  {  T cont;  if (size == 1)  {  size = 0; //setup zero position  cont = tail->content;  head = NULL;  tail = NULL;  }  else if (size > 1) //if list is not empty  {  cont = tail->content;  tail = tail->before; //set new tail  delete tail->next;  tail->next = NULL;  size--;  }  else //else return error  throw out\_of\_range("Segmentation fault");  return cont;  }  template <class T>  void Stack<T>::clear()  {  if (head != NULL)  {  while (head->next != NULL) //delete all item  {  head = head->next;  delete head->before;  }  size = 0; //setup zero position  head = NULL;  tail = NULL;  }  }  template <class T>  bool Stack<T>::isEmpty()  {  if (head == NULL)  return true;  return false;  } |
| RBTree.h |
| #pragma once  #include "Stack.h"  #include "Stack.cpp"  #include <iostream>  using namespace std;  template <typename T>  class RBTree  {  private:  class RBNode  {  public:  RBNode(T data, T key, bool color, RBNode\* parent, RBNode\* left, RBNode\* right) {  this->data = data;  this->key = key;  this->parent = parent;  this->left = left;  this->right = right;  this->color = color; // red  };  ~RBNode() {};  T data;  T key;  RBNode\* parent;  RBNode\* left;  RBNode\* right;  bool color; // true - red, false - black  };  class dft\_Iterator // depth-first traverse  {  private:  Stack<RBNode\*> stack;  RBNode\* current;  RBNode\* nil;  public:  dft\_Iterator(RBNode\* root = nullptr, RBNode\* nil = nullptr) {  this->nil = nil;  current = root;  stack.push(current);  }  RBNode\* next() {  if (!has\_next()) {  throw "No more elements";  }  RBNode\* temp = current;  if (current->right != nil) //add right tree to stack  {  stack.push(current->right);  }  if (current->left != nil) //go left  {  current = current->left;  }  else { // if can't  current = stack.pop();  }  return temp;  }  bool has\_next() {  if (!stack.isEmpty())  return true;  else  return false;  }  ~dft\_Iterator() {};  };  void set\_root(T key, T data) {  RBNode\* tmp = new RBNode(data, key, false, nil, nil, nil);  root = tmp;  root->parent = nil;  root->color = false; // black root's color  nil->color = false; // black nill's color  };  RBNode\* root;  RBNode\* nil = new RBNode(0, 0, false, nil, nil, nil);  size\_t size;  public:  RBTree() {  root = nil;  size = 0;  };  ~RBTree() {  clear();  delete root;  size = 0;  };  void left(RBNode\* newNode) {  if (newNode->right == nullptr)  return;  else  {  RBNode\* y = newNode->right;  if (y->left != nullptr)  {  newNode->right = y->left;  y->left->parent = newNode;  }  else  newNode->right = nullptr;  if (newNode->parent != nullptr)  y->parent = newNode->parent;  if (newNode->parent == nullptr)  root = y;  else  {  if (newNode == newNode->parent->left)  newNode->parent->left = y;  else  newNode->parent->right = y;  }  y->left = newNode;  newNode->parent = y;  }  };  void right(RBNode\* newNode) {  if (newNode->left == nullptr)  return;  else  {  RBNode\* y = newNode->left;  if (y->right != nullptr)  {  newNode->left = y->right;  y->right->parent = newNode;  }  else  newNode->left = nullptr;  if (newNode->parent != nullptr)  y->parent = newNode->parent;  if (newNode->parent == nullptr)  root = y;  else  {  if (newNode == newNode->parent->left)  newNode->parent->left = y;  else  newNode->parent->right = y;  }  y->right = newNode;  newNode->parent = y;  }  };  void coloring(RBNode\* node) {  if (node == root)  {  node->color = false;  return;  }  RBNode\* uncle;  while ((node->parent->color) && (node != root)) //red parent  {  if (node->parent == node->parent->parent->left) //if parent is a left child  {  uncle = node->parent->parent->right;  if (uncle->color) //if uncle is red too  {  node->parent->color = false; //repaint parent and uncle  uncle->color = false;  node->parent->parent->color = true; //repaint grand parent  node = node->parent->parent; //check grand parent  }  else { //if uncle is not red  if (node == node->parent->right) { // if node is a right child  node = node->parent;  left(node);  }  node->parent->color = false;  node->parent->parent->color = true; //repaint grand parent  right(node->parent->parent);  }  }  else {  uncle = node->parent->parent->left;  if (uncle->color)  {  node->parent->color = false;  uncle->color = false;  node->parent->parent->color = true;  node = node->parent->parent;  }  else {  if (node == node->parent->left) {  node = node->parent;  right(node);  }  node->parent->color = false;  node->parent->parent->color = true;  left(node->parent->parent);  }  }  }  root->color = false;  }  bool is\_empty() {  return (root == nullptr);  };  void insert(T key, T data) {  if (root == nil) // add root  {  root = new RBNode(key, data, false, nil, nil, nil);  }  else {  RBNode\* newNode = new RBNode(key, data, true, nil, nil, nil);  RBNode\* parent = root;  RBNode\* leaf = nullptr;  while (parent != nil)  {  leaf = parent;  if (parent->key < newNode->key)  parent = parent->right;  else  parent = parent->left;  }  newNode->parent = leaf;  if (leaf != nullptr) {  if (leaf->key < newNode->key)  leaf->right = newNode;  else  leaf->left = newNode;  coloring(newNode);  }  }  size++;  };  RBNode\* find(T key) {  RBNode\* x = root;  while (x->key != key)// find key to remove  {  if ((x == nil) || (x == nullptr))  {  throw out\_of\_range("Element not found");  }  if (x->key < key)  x = x->right;  else  x = x->left;  }  return x;  };  void fix(RBNode\* node) {  if (node) {  RBNode\* brother;  while ((node != root) && (node->color == false))  {  if (node == node->parent->left) //if node is a left child  {  brother = node->parent->right;  if (brother->color) //if brother is red  {  brother->color = false;  node->parent->color = true; //swap parent's and brother's colors  left(node->parent);  brother = node->parent->right;  }  if ((brother->left->color == false) && (brother->right->color == false)) //both brother child black  {  brother->color = true;  node = node->parent;  }  else {  if (brother->right->color == false)  {  brother->left->color = false;  brother->color = true;  right(brother);  brother = node->parent->right;  }  brother->color = node->parent->color;  node->parent->color = false;  brother->right->color = false;  left(node->parent);  node = root;  }  }  else { //as in the previous if, but in the other direction  brother = node->parent->left;  if (brother->color)  {  brother->color = false;  node->parent->color = true;  right(node->parent);  brother = node->parent->left;  }  if ((brother->right->color == false) && (brother->left->color == false))  {  brother->color = true;  node = node->parent;  }  else {  if (brother->left->color == false)  {  brother->right->color = false;  brother->color = true;  right(brother);  brother = node->parent->left;  }  brother->color = node->parent->color;  node->parent->color = false;  brother->left->color = false;  right(node->parent);  node = root;  }  }  }  node->color = false; //root color  }  };  void remove(T remKey) {  RBNode\* node = find(remKey);  if (node == nullptr) return; // if there is no such node we cant delete it  // if we had node, we can delete it  size--;  RBNode\* nodeA;  RBNode\* nodeB;  if ((node->left == nil) || (node->right == nil)) // if node has <2 children  nodeA = node;  else {  nodeA = node->right;  while (nodeA->left != nil) {  nodeA = nodeA->left; //looking for node to swap  }  }  if (nodeA->left != nil) { //fix node  nodeB = nodeA->left;  }  else {  nodeB = nodeA->right;  }  nodeB->parent = nodeA->parent;  if (nodeA->parent != nil) // if nodeA is not root  {  if (nodeA == nodeA->parent->left) {  nodeA->parent->left = nodeB;  }  else {  nodeA->parent->right = nodeB;  }  }  else { // if nodeA is root  root = nodeB;  }  if (nodeA != node)  {  node->key = nodeA->key;  node->data = nodeA->data;  }  if (nodeA->color == false) //avoiding red parent  {  fix(nodeB);  }  delete nodeA;  };  void clear() {  while ((root != nil) && (root != nullptr))  remove(root->key);  root = nil;  size = 0;  };  size\_t get\_size() {  return size;  };  T\* get\_keys() {  dft\_Iterator i(root, nil);  T\* arr = new T[size];  int arrI = 0;  RBNode\* cur;  while (i.has\_next())  {  cur = i.next();  arr[arrI] = cur->key;  arrI++;  }  return arr;  };  T\* get\_data() {  dft\_Iterator i(root, nil);  T\* arr = new T[size];  int arrI = 0;  RBNode\* cur;  while (i.has\_next())  {  cur = i.next();  arr[arrI] = cur->data;  arrI++;  }  return arr;  };  void print() {  dft\_Iterator i(root, nil);  RBNode\* cur;  while (i.has\_next())  {  cur = i.next();  cout << cur->key << " " << endl;  }  cout << endl;  };  }; |
| UnitTest1.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include"../1LAB/RBTree.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace lab1test  {  TEST\_CLASS(lab1test)  {  public:  RBTree <int> RB;  int\* keys;  int\* Ktree;  int\* values;  int\* Vtree;  TEST\_METHOD(insert\_1)  {  RB.insert(1, 1);  keys = new int[1]{ 1 };  Ktree = RB.get\_keys();  Assert::AreEqual(keys[0], Ktree[0]);  }  TEST\_METHOD(remove\_1)  {  RB.insert(1, 1);  RB.insert(2, 2);  RB.remove(2);  Assert::AreEqual((int)RB.get\_size(), 1);  }  TEST\_METHOD(clear)  {  RB.insert(1, 1);  RB.insert(2, 2);  RB.clear();  Assert::AreEqual((int)RB.get\_size(), 0);  }  TEST\_METHOD(insert\_2)  {  RB.insert(1, 1);  RB.insert(2, 3);  keys = new int[2]{ 1, 3};  Ktree = RB.get\_keys();  Assert::AreEqual((int)RB.get\_size(), 2);  }  TEST\_METHOD(remove\_2)  {  RB.insert(1, 1);  RB.insert(2, 2);  RB.remove(1);  Assert::AreEqual((int)RB.get\_size(), 1);  }  };  } |

* **Пример работы**

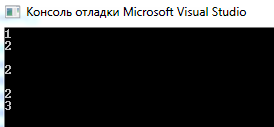


Рисунок 1. Пример работы программы

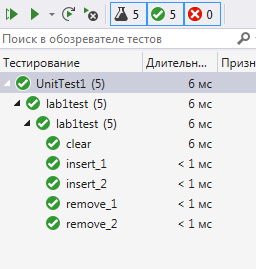


Рисунок 2. Реализованные unit-тесты

* **Вывод**

Выполняя лабораторную работу, я научился реализовывать ассоциативный ряд на основе красно-черного дерева.