**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 |  | Горбанев А.В |
| 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 “Red\_Black\_Tree”, “List”,”Node”.

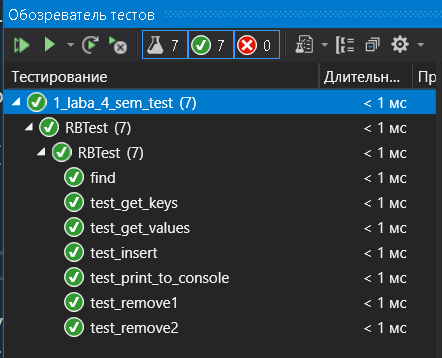
Оценка временной сложности каждого метода:

|  |  |
| --- | --- |
| Функция | Сложность |
| Insert(key,data) | O(log(n)) |
| Remove(key) | O(log(n)) |
| Find(key) | O(log(n)) |
| Clear | O(n) |
| Get\_keys | O(n) |
| Get\_values | O(n) |
| print | O(n) |

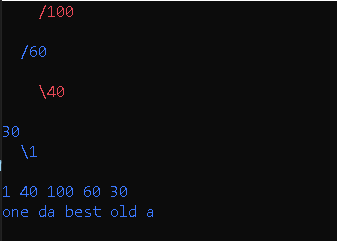
## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Проверка методов класса «Tree\_Node» | |
| Название теста | Что проверяет |
| Test\_insert | Проверка метода void insert(T1 key,T2 data) |
| Test\_remove1 | Проверка метода void remove(T1 key) |
| Test\_remove2 | Проверка метода void remove(T1 key) |
| Test\_find | Проверка метода void find(T1 key) |
| Test\_print\_to\_console | Проверка метода void print() |
| Test\_get\_keys | Проверка метода get\_keys() |
| Test\_get\_values | Проверка метода get\_values() |

## Результат выполнения всех unit-тестов



## Пример работы программы



## Листинг

|  |
| --- |
| List.h |
| template<typename T>  class List  {  private:  class Node  {  private:  T data;  Node\* next, \* prev;  public:  Node() : next(NULL), prev(NULL) {};  Node(T data) {  this->data = data;  next = NULL;  prev = NULL;  }  ~Node() {  next = NULL;  prev = NULL;  }  void set\_data(T data) {  this->data = data;  }  T get\_data() {  return data;  }  Node\* get\_next() {  return next;  }  Node\* get\_prev() {  return prev;  }    void set\_next(Node \* pointer) {  next = pointer;  }  void set\_prev(Node\* pointer) {  prev = pointer;  }  };  Node\* head, \* tail;  Node\* get\_pointer(size\_t index)  {  if (isEmpty() || (index > get\_size() - 1))  {  throw out\_of\_range("Invalid argument");  }  else if (index == get\_size() - 1)  return tail;  else if (index == 0)  return head;  else  {  Node\* temp = head;  while ((temp) && (index--))  {  temp = temp->get\_next();  }  return temp;  }  }  public:  List() : head(NULL), tail(NULL) {}  List(const List<T>& list) {  clear();  Node\* temp = list.head;  while (temp) {  push\_back(temp->get\_data());  temp = temp->get\_next();  }  }  ~List()  {  while (head)  {  tail = head->get\_next();  delete head;  head = tail;  }  head = NULL;  }  void push\_back(T data)  {  Node\* temp = new Node;  temp->set\_data(data);  if (head)  {  temp->set\_prev(tail);  tail->set\_next(temp);  tail = temp;  }  else  {  head = temp;  tail = head;  }    }  void push\_front(T data)  {  Node\* temp = new Node;  temp->set\_data(data);  if (head)  {  temp->set\_next(head);  head->set\_prev(temp);  head = temp;  }  else  {  head = temp;  tail = head;  }  }  void push\_front(List& ls2)  {  Node\* temp = ls2.tail;  size\_t n = ls2.get\_size();  while ((temp) && (n--))  {  push\_front(temp->get\_data());  temp = temp->get\_prev();  }  }  void pop\_back()  {  if (head != tail)  {  Node\* temp = tail;  tail = tail->get\_prev();  tail->set\_next(NULL);  delete temp;  }  else if (!isEmpty())  {  Node\* temp = tail;  tail = head = NULL;  delete temp;  }  else  throw out\_of\_range("The list is empty");  }  void pop\_front()  {  if (head != tail)  {  Node\* temp = head;  head = head->get\_next();  head->set\_prev(NULL);  delete temp;  }  else if (!isEmpty())  {  Node\* temp = head;  head = tail = NULL;  delete temp;  }  else  throw out\_of\_range("The list is empty");  }    void insert(size\_t index, T data)  {  Node\* temp;  temp = get\_pointer(index);  if (temp == head)  push\_front(data);  else  {  Node\* newElem = new Node;  newElem->set\_data(data);  temp->get\_prev()->set\_next(newElem);  newElem->set\_prev(temp->get\_prev());  newElem->set\_next(temp);  temp->set\_prev(newElem);  }  }  T at(size\_t index)  {  Node\* temp;  temp = get\_pointer(index);  return temp->get\_data();  }  void remove(size\_t index)  {  Node\* temp;  temp = get\_pointer(index);  if (temp == head)  pop\_front();  else if (temp == tail)  pop\_back();  else  {  temp->get\_prev()->set\_next(temp->get\_next());  temp->get\_next()->set\_prev(temp->get\_prev());  delete temp;  }  }  void remove(T data) {  Node\* temp = head;  while (temp && temp->get\_data() != data)  temp = temp->get\_next();  if (!temp)  throw out\_of\_range("Invalid argument");  if (temp == head)  pop\_front();  else if (temp == tail)  pop\_back();  else  {  temp->get\_prev()->set\_next(temp->get\_next());  temp->get\_next()->set\_prev(temp->get\_prev());  delete temp;  }  }  size\_t get\_size()  {  Node\* temp = head;  size\_t length = 0;  while (temp)  {  length++;  temp = temp->get\_next();  }    return length;  }  void print\_to\_console()  {  Node\* temp = head;  while (temp)  {  cout << temp->get\_data() << ' ';  temp = temp->get\_next();  }  cout << endl;  }  void clear()  {  while (head)  {  tail = head->get\_next();  delete head;  head = tail;  }  }  void set(size\_t index, T data)  {  Node\* temp;  temp = get\_pointer(index);  temp->set\_data(data);  }  bool isEmpty()  {  if (!head)  return true;  else  return false;  }  }; |
| Red\_black\_tree.cpp |
| #include <iostream>  #include "Node.h"  #include <windows.h>  #include "List.h"  void SetColor(int text, int background)  {  HANDLE hConsoleHandle = GetStdHandle(STD\_OUTPUT\_HANDLE);  SetConsoleTextAttribute(hConsoleHandle, (WORD)((background << 4) | text));  }  using namespace std;  template<typename T1, typename T2>  class Red\_Black\_Tree {  Node<T1, T2>\* root;  void leftRotate(Node<T1, T2>\* x) {  Node<T1, T2>\* nParent = x->right;  if (x == root)  root = nParent;  x->moveDown(nParent);  x->right = nParent->left;  if (nParent->left != NULL)  nParent->left->parent = x;  nParent->left = x;  }  void rightRotate(Node<T1, T2>\* x) {  Node<T1, T2>\* nParent = x->left;  if (x == root)  root = nParent;  x->moveDown(nParent);  x->left = nParent->right;  if (nParent->right != NULL)  nParent->right->parent = x;  nParent->right = x;  }  void swapColors(Node<T1, T2>\* x1, Node<T1, T2>\* x2) {  COLOR temp;  temp = x1->color;  x1->color = x2->color;  x2->color = temp;  }    void swapkeyues(Node<T1, T2> \* u, Node<T1, T2> \* v) {  int temp;  temp = u->key;  u->key = v->key;  v->key = temp;  }  void fixRedRed(Node<T1, T2>\* newElement) {  if (newElement == root) {  newElement->color = BLACK;  return;  }  Node<T1, T2>\* parent = newElement->parent, \* grandparent = parent->parent,  \* uncle = newElement->uncle();  if (parent->color != 1) {  if (uncle != NULL && uncle->color == 0) {  parent->color = BLACK;  uncle->color = BLACK;  grandparent->color = RED;  fixRedRed(grandparent);  }  else {  if (parent->isOnLeft()) {  if (newElement->isOnLeft()) {  swapColors(parent, grandparent);  }  else {  leftRotate(parent);    swapColors(newElement, grandparent);  }  rightRotate(grandparent);  }  else {  if (newElement->isOnLeft()) {  rightRotate(parent);  swapColors(newElement, grandparent);  }  else {  swapColors(parent, grandparent);  }  leftRotate(grandparent);  }  }  }  }  Node<T1, T2>\* successor(Node<T1, T2>\* x) {  Node<T1, T2>\* temp = x;  while (temp->left != NULL)  temp = temp->left;  return temp;  }  Node<T1, T2>\* BSTreplace(Node<T1, T2>\* x) {  if (x->left != NULL && x->right != NULL)  return successor(x->right);  if (x->left == NULL && x->right == NULL)    return NULL;  if (x->left != NULL)  return x->left;  else  return x->right;  }  void deleteNode(Node<T1, T2>\* v) {  Node<T1, T2>\* u = BSTreplace(v);  bool uv1 = ((u == NULL || u->color == 1) && (v->color == 1));  Node<T1, T2>\* parent = v->parent;  if (u == NULL) {  if (v == root) {  root = NULL;  }  else {  if (uv1) {  fixDoubleBlack(v);  }  else {  if (v->sibling() != NULL)  v->sibling()->color = RED;  }    if (v->isOnLeft()) {  parent->left = NULL;  }  else {  parent->right = NULL;  }  }  delete v;  return;  }  if (v->left == NULL || v->right == NULL) {  if (v == root) {  v->key = u->key;  v->left = v->right = NULL;  delete u;  }  else {  if (v->isOnLeft()) {  parent->left = u;  }  else {  parent->right = u;  }  delete v;  u->parent = parent;  if (uv1) {  fixDoubleBlack(u);  }  else {  u->color = BLACK;  }  }  return;  }  swapkeyues(u, v);  deleteNode(u);  }  void fixDoubleBlack(Node<T1, T2>\* x) {  if (x == root)  return;  Node<T1, T2>\* sibling = x->sibling(), \* parent = x->parent;  if (sibling == NULL) {  fixDoubleBlack(parent);  }  else {  if (sibling->color == 0) {  parent->color = RED;  sibling->color = BLACK;  if (sibling->isOnLeft()) {  rightRotate(parent);  }  else {  leftRotate(parent);  }  fixDoubleBlack(x);  }  else {  if (sibling->has0Child()) {  if (sibling->left != NULL && sibling->left->color == 0) {  if (sibling->isOnLeft()) {  sibling->left->color = sibling->color;  sibling->color = parent->color;  rightRotate(parent);  }  else {  sibling->left->color = parent->color;  rightRotate(sibling);  leftRotate(parent);  }  }  else {  if (sibling->isOnLeft()) {  sibling->right->color = parent->color;  leftRotate(sibling);  rightRotate(parent);  }  else {  sibling->right->color = sibling->color;  sibling->color = parent->color;    leftRotate(parent);  }  }  parent->color = BLACK;  }  else {  sibling->color = RED;  if (parent->color == 1)  fixDoubleBlack(parent);  else  parent->color = BLACK;  }  }  }  }  public:  Red\_Black\_Tree() { root = NULL; }  ~Red\_Black\_Tree() {  clear(root);  root = NULL;  };  Node<T1, T2>\* getRoot() { return root; }  Node<T1, T2>\* find(T1 n) {  Node<T1, T2>\* temp = root;  while (temp != NULL) {  if (n < temp->key) {  if (temp->left == NULL)  break;  else  temp = temp->left;  }  else if (n == temp->key) {  break;  }  else {  if (temp->right == NULL)  break;  else  temp = temp->right;  }  }  return temp;  }  void insert(T1 key, T2 data) {  Node<T1, T2>\* newNode = new Node<T1, T2>(key, data);  if (root == NULL) {  newNode->color = BLACK;  root = newNode;  }  else {  Node<T1, T2>\* temp = find(key);  newNode->parent = temp;  if (key < temp->key)  temp->left = newNode;  else  temp->right = newNode;  fixRedRed(newNode);  }  }  void remove(T1 n) {  if (root == NULL)  return;  Node<T1, T2>\* v = find(n), \* u;  if (v->key != n) {  cout << "No Node<T1, T2> found to delete with keyue:" << n << endl;  return;  }  deleteNode(v);  }  void print(Node<T1, T2>\* root, int lvl)  {  if (root != NULL)  {  print(root->right, lvl + 2);  for (int i = 0; i < lvl; i++)  {  cout << " ";  }  if (root->parent != NULL && root->parent->key >= root->key)  {  if (root->color == 0)  SetColor(12, 0);  else  SetColor(9, 0);  cout << "\\" << root->key;  cout << endl;  }  else if (root->parent != NULL && root->parent->key < root->key)  {  if (root->color == 0)  SetColor(12, 0);  else  SetColor(9, 0);  cout << "/" << root->key;  cout << endl;  }  else  {  SetColor(9, 0);  cout << root->key;  }  cout << endl;  print(root->left, lvl + 2);  }  }  List<T1>\* getListKey(Node<T1, T2>\* root, List<T1>\* A)  {  if (root)  {  getListKey(root->left, A);  getListKey(root->right, A);  A->push\_back(root->key);  }  return A;  }  List<T2>\* getListData(Node<T1, T2>\* root, List<T2>\* A)  {  if (root)  {  getListData(root->left, A);  getListData(root->right, A);  A->push\_back(root->data);  }  return A;  }  void printListKey(List<T1>\* list)  {  while (!list->isEmpty())  {  cout << list->at(0) << " ";  list->pop\_front();  }  }  void printListData(List<T2>\* list)  {  while (!list->isEmpty())  {  cout << list->at(0) << " ";  list->pop\_front();  }  }  List<Node<T1, T2>\*>\* getList(Node<T1, T2>\* root, List<Node<int, int>\*>\* A)  {  if (root)  {  getList(root->left, A);  getList(root->right, A);  A->push\_back(root);  }  return A;  }  void printList(List<Node<T1, T2>\*>\* list)  {  while (!list->isEmpty())  {  cout << list->at(0)->data;  cout << endl;  list->pop\_front();  }  }  void clear(Node<T1, T2>\* root)  {  if (root)  {  clear(root->left);  clear(root->right);  delete root;  }  }  };  int main() {  Red\_Black\_Tree<int, string> tree;  tree.insert(30, "a");  tree.insert(40, "da");  tree.insert(1, "one");  tree.insert(100, "best");  tree.insert(60, "old");  tree.print(tree.getRoot(), 0);    List<int>\* myList = new List<int>;  myList = tree.getListKey(tree.getRoot(), myList);  myList->print\_to\_console();  List<string>\* myList1 = new List<string>;  myList1 = tree.getListData(tree.getRoot(), myList1);  myList1->print\_to\_console();  } |
| 1\_laba\_4\_sem\_test.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include <string>  #include "../1\_laba\_4\_sem/red\_black\_tree.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace RBTest  {  TEST\_CLASS(RBTest)  {  private:  Red\_Black\_Tree<int, string> tree;  public:  TEST\_METHOD\_INITIALIZE(init)  {  tree.insert(30, "a");  tree.insert(40, "a");  tree.insert(15, "daada");  tree.insert(10, "daada");  tree.insert(20, "daada");  tree.insert(5, "daada");  tree.insert(3, "daada");  }  TEST\_METHOD(test\_insert)  {  tree.insert(2, "oneone");  tree.insert(21, "twotwo");  Assert::AreEqual(tree.getRoot()->left->left->left->key, 2);  Assert::AreEqual(tree.getRoot()->right->left->right->key, 21);  }  TEST\_METHOD(test\_remove1)  {  tree.remove(15);  Assert::AreEqual(tree.getRoot()->left->key, 5);  }  TEST\_METHOD(test\_remove2)  {  tree.remove(20);  Assert::AreEqual(tree.getRoot()->left->key, 5);  }  TEST\_METHOD(find)  {  Assert::AreEqual(tree.find(40)->data, (string)"a");  }  TEST\_METHOD(test\_print\_to\_console)  {  tree.print(tree.getRoot(), 0);  }  TEST\_METHOD(test\_get\_keys)  {  List<int>\* myList = new List<int>;  myList = tree.getListKey(tree.getRoot(), myList);  Assert::AreEqual(myList->at(0), 3);  Assert::AreEqual(myList->at(1), 10);  }  TEST\_METHOD(test\_get\_values)  {  List<string>\* myList1 = new List<string>;  myList1 = tree.getListData(tree.getRoot(), myList1);  Assert::AreEqual(myList1->at(0), (string)"daada");  Assert::AreEqual(myList1->at(1), (string)"daada");  }  };  } |
| Node.h |
| #include<iostream>  #pragma once  #ifndef Node\_h  using namespace std;  enum COLOR { RED, BLACK };  template<typename T1, typename T2>  class Node {  public:  T1 key;  T2 data;  COLOR color;  Node<T1, T2>\* left, \* right, \* parent;  Node(T1 key, T2 data) : key(key), data(data) {  parent = left = right = NULL;  color = RED;  }  ~Node() = default;    Node<T1, T2> \* uncle() {  if (parent == NULL || parent->parent == NULL)  return NULL;  if (parent->isOnLeft())  return parent->parent->right;  else  return parent->parent->left;  }  bool isOnLeft() { return this == parent->left; }  Node<T1, T2>\* sibling() {  if (parent == NULL)  return NULL;  if (isOnLeft())  return parent->right;  return parent->left;  }  void moveDown(Node<T1, T2>\* nParent) {  if (parent != NULL) {  if (isOnLeft()) {    parent->left = nParent;  }  else {  parent->right = nParent;  }  }  nParent->parent = parent;  parent = nParent;  }  bool has0Child() {  return (left != NULL && left->color == 0) ||  (right != NULL && right->color == 0);  }  };  #define Node\_h  #endif |

Вывод:

В ходе лабораторной работы научился работать с ассоциативным массивом на основе красно—черного дерева.