МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

Тема: Ассоциативный массив

Студент гр. 8302 _____ Халитов Ю.Р.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

Постановка задачи

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

Описание реализуемого класса и методов

Map<T_key, T_val> – класс словаря на основе красно–чёрного дерева (T_val – тип ключа, T_key – тип значения), имеющий следующие поля:

Таблица 1 – Поля класса Map<T_key, T_val>

Поле	Назначение
Node* root;	Указатель на корень дерева
inline static size_t size = 0;	Количество узлов дерева
inline static Node* nil = new Node(BLACK);	«лист» дерева, на который указывают все крайние элементы дерева

Node – вложенный в Map<T_key, T_val> класс узла дерева, имеющий следующие поля:

Таблица 2 – Поля класса Node

Поле	Назначение
T_key key;	Ключ
T_val value;	Значение
color col;	Цвет
Node* p;	Указатель на родителя
Node* left;	Указатель на левого потомка
Node* right;	Указатель на правого потомка

enum color – перечисление со значениями BLACK и RED, используется для окраски узлов в дереве.

Таблица 3 – Описание методов класса Map<T_key, T_val>

Метод	Назначение
void left_rotate(Node* z);	Осуществляет левый поворот дерева относительно узла z
void right_rotate(Node* z);	Осуществляет правый поворот дерева относительно узла z

void insert_fixup(Node* z);	Восстанавливает красно-черные свойства дерева после вставки узла
void insert(T_key key, T_val val);	Вставляет узел с ключем key и значением val в дерево
auto minimum(Node* x);	Возвращает указатель на узел с минимальным ключем
void transplant(Node* x, Node* y);	Заменяет поддерево x поддеровом y
void remove_fixup(Node* x);	Восстанавливает красно-черные свойства дерева после удаления узла
void remove(T_key key);	Удаляет элемент с ключем key из дерева
T_val find(T_key key, T_val nil_val);	Возвращает значение, которое соответствует ключу key или nil_val, если такого ключа не существует
void clear_subtree(Node* x);	Удаляет все узлы из поддерева x и освобождает память, выделенную под узлы
void clear();	Удаляет все узлы из дерева и освобождает память, выделенную под узлы
void get_keys_subtree(Node* x, std::vector<T_key>& v);	Добавляет в массив v все ключи поддерева x
std::vector<T_key> get_keys();	Возвращает массив ключей словаря
void get_values_subtree(Node* x, std::vector<T_val>& v);	Добавляет в массив v все значения поддерева x
std::vector<T_val> get_values();	Возвращает массив значений словаря
void print_subtree(Node* x);	Выводит все пары ключ-значение поддерева x
void print();	Выводит все пары ключ-значение словаря

Оценка временной сложности каждого метода

Таблица 4 – Оценка временной сложности методов класса Map<T_key, T_val>

Метод	Сложность
void left_rotate(Node* z);	$O(1)$
void right_rotate(Node* z);	$O(1)$
void insert_fixup(Node* z);	$O(\log(n))$
void insert(T_key key, T_val val);	$O(\log(n))$
auto minimum(Node* x);	$O(\log(n))$
void transplant(Node* x, Node* y);	$O(1)$
void remove_fixup(Node* x);	$O(\log(n))$
void remove(T_key key);	$O(\log(n))$
T_val find(T_key key, T_val nil_val);	$O(\log(n))$
void clear_subtree(Node* x);	$O(n)$
void clear();	
void get_keys_subtree(Node* x, std::vector<T_key>& v);	$O(n)$
std::vector<T_key> get_keys();	
void get_values_subtree(Node* x, std::vector<T_val>& v);	$O(n)$
std::vector<T_val> get_values();	
void print_subtree(Node* x);	$O(n)$
void print();	

n – количество элементов в дереве

Описание реализованных unit-тестов

Таблица 5 – Unit-тесты

Название	Описание
insert_one_element	Добавление одного элемента в пустой словарь
insert_few_elements	Добавление трех элементов в пустой словарь

remove_one_element	Удаление единственного элемента из словаря
remove_few_elements	Удаление двух из пяти элементов словаря
find	Поиск значения по существующему ключу
find_not_exist	Поиск значения по несуществующему ключу
get_keys	Получение вектора ключей словаря, состоящего из трех элементов
get_values	Получение вектора значений словаря, состоящего из трех элементов
clear	Очистка словаря из 5 элементов

Пример работы

В примере был создан словарь, ключами которого являются строки (std::string), а значениями – целые числа (int). В словарь было добавлено 10 элементов, затем удалено 5. В конце словарь был очищен.

Рис.1 – Пример работы программы

Листинг

Map.h:

```
#pragma once
#include <string>
#include <vector>
#include <iostream>

template<class T_key, class T_val>
class Map {
	enum color { RED, BLACK };
	struct Node {
		T_key key;
		T_val value;
		color col;
		Node* p;
		Node* left;
		Node* right;
		Node(color col) : col{ col } {}
		Node(T_key key, T_val value, color col = RED) : key{ key },
value{ value },
			p{ nil }, left{ nil }, right{ nil }, col{ col } {}
	};
	Node* root;
	inline static size_t size = 0;
	inline static Node* nil = new Node(BLACK);
	void left_rotate(Node* z);
	void right_rotate(Node* x);
	void insert_fixup(Node* z);
	auto minimum(Node* x);
	void transplant(Node* x, Node* y);
	void remove_fixup(Node* x);
	void clear_subtree(Node* x);
	void get_keys_subtree(Node* x, std::vector<T_key>& v);
	void get_values_subtree(Node* x, std::vector<T_val>& v);
```

```
	void print_subtree(Node* x);
public:
	Map() : root{ nil } {}
	void insert(T_key key, T_val val);
	void remove(T_key key);
	T_val find(T_key key, T_val nil_val);
	void clear();
	std::vector<T_key> get_keys();
	std::vector<T_val> get_values();
	void print();
};

template <class T_key, class T_val>
void Map<T_key, T_val>::left_rotate(Node* x)
{
	Node* y = x->right;
	x->right = y->left;
	if (y->left != nil) y->left->p = x;
	y->p = x->p;
	if (x->p == nil) root = y;
	else if (x == x->p->left) x->p->left = y;
	else x->p->right = y;
	y->left = x;
	x->p = y;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::right_rotate(Node * x)
{
	Node* y = x->left;
	x->left = y->right;
	if (y->right != nil) y->right->p = x;
	y->p = x->p;
	if (x->p == nil) root = y;
```

```
      else if (x == x->p->left) x->p->left = y;
      else x->p->right = y;
      y->right = x;
      x->p = y;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::insert_fixup(Node * z)
{
      Node* y;
      while (z->p->col == RED)
      {
            if (z->p == z->p->p->left)
            {
                  y = z->p->p->right;
                  if (y->col == RED)
                  {
                        z->p->col = BLACK;
                        y->col = BLACK;
                        z->p->p->col = RED;
                        z = z->p->p;
                  }
                  else {
                        if (z == z->p->right) {
                              z = z->p;
                              left_rotate(z);
                        }
                        z->p->col = BLACK;
                        z->p->p->col = RED;
                        right_rotate(z->p->p);
                  }
            }
            else
            {
```

```
                y = z->p->p->left;
                if (y->col == RED)
                {
                      z->p->col = BLACK;
                      y->col = BLACK;
                      z->p->p->col = RED;
                      z = z->p->p;
                }
                else
                {
                      if (z == z->p->left) {
                            z = z->p;
                            right_rotate(z);
                      }
                      z->p->col = BLACK;
                      z->p->p->col = RED;
                      left_rotate(z->p->p);
                }
          }
      }
      root->col = BLACK;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::insert(T_key key, T_val val)
{
      Node* y = nil;
      Node* x = root;
      while (x != nil && x->key != key)
      {
            y = x;
            if (key < x->key) x = x->left;
            else x = x->right;
      }
```

```
      if (x == nil)
      {
            Node* z = new Node(key, val);
            z->p = y;
            if (y == nil) root = z;
            else if (z->key < y->key) y->left = z;
            else y->right = z;
            insert_fixup(z);
            ++size;
      }
      else x->value = val;
}

template <class T_key, class T_val>
auto Map<T_key, T_val>::minimum(Node* x)
{
      while (x->left != nil) x = x->left;
      return x;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::transplant(Node* x, Node* y)
{
      if (x->p == nil) root = y;
      else if (x == x->p->left) x->p->left = y;
      else x->p->right = y;
      y->p = x->p;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::remove_fixup(Node* x)
{
      while (x != root && x->col == BLACK)
      {
```

```
if (x == x->p->left)
{
      Node* y = x->p->right;
      if (y->col == RED)
      {
            y->col = BLACK;
            x->p->col = RED;
            left_rotate(x->p);
            y = x->p->right;
      }
      if (y->left->col == BLACK && y->right->col == BLACK)
      {
            y->col = RED;
            x = x->p;
      }
      else
      {
            if (y->right->col == BLACK)
            {
                  y->left->col = BLACK;
                  y->col = RED;
                  right_rotate(y);
                  y = x->p->right;
            }
            y->col = x->p->col;
            x->p->col = BLACK;
            y->right->col = BLACK;
            left_rotate(x->p);
            x = root;
      }
}
else
{
      Node* y = x->p->left;
```

```
                if (y->col == RED)
                {
                      y->col = BLACK;
                      x->p->col = RED;
                      right_rotate(x->p);
                      y = x->p->left;
                }
                if (y->right->col == BLACK && y->left->col == BLACK)
                {
                      y->col = RED;
                      x = x->p;
                }
                else
                {
                      if (y->left->col == BLACK)
                      {
                            y->right->col = BLACK;
                            y->col = RED;
                            left_rotate(y);
                            y = x->p->left;
                      }
                      y->col = x->p->col;
                      x->p->col = BLACK;
                      y->left->col = BLACK;
                      right_rotate(x->p);
                      x = root;
                }
            }
      }
      x->col = BLACK;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::remove(T_key key)
```

```
{
	Node* z = root;
	while (z != nil && z->key != key)
	{
		if (key < z->key) z = z->left;
		else z = z->right;
	}
	if (z == nil) throw std::out_of_range{ "remove element out of
range" };
	Node * y = z;
	Node * x;
	color y_orig_color = y->col;
	if (z->left == nil)
	{
		x = z->right;
		transplant(z, z->right);
	}
	else if (z->right == nil)
	{
		x = z->left;
		transplant(z, z->left);
	}
	else
	{
		y = minimum(z->right);
		y_orig_color = y->col;
		x = y->right;
		if (y->p == z)
		{
			x->p = y;
		}
		else
		{
			transplant(y, y->right);
```

```
                y->right = z->right;
                y->right->p = y;
            }
            transplant(z, y);
            y->left = z->left;
            y->left->p = y;
            y->col = z->col;
      }
      if (y_orig_color == BLACK) remove_fixup(x);
}

template <class T_key, class T_val>
T_val Map<T_key, T_val>::find(T_key key, T_val nil_value)
{
      Node* x = root;
      while (x != nil)
      {
            if (key == x->key) return x->value;
            else if (key < x->key) x = x->left;
            else x = x->right;
      }
      return nil_value;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::clear_subtree(Node* x)
{
      if (x != nil)
      {
            clear_subtree(x->left);
            clear_subtree(x->right);
            delete x;
      }
}
```

```
template <class T_key, class T_val>
void Map<T_key, T_val>::clear()
{
	clear_subtree(root);
	root = nil;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::get_keys_subtree(Node* x, std::vector<T_key>& v)
{
	if (x != nil)
	{
		get_keys_subtree(x->left, v);
		v.push_back(x->key);
		get_keys_subtree(x->right, v);
	}
}

template <class T_key, class T_val>
std::vector<T_key> Map<T_key, T_val>::get_keys()
{
	std::vector<T_key> v;
	get_keys_subtree(root, v);
	return v;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::get_values_subtree(Node* x, std::vector<T_val>&
v)
{
	if (x != nil)
	{
		get_values_subtree(x->left, v);
```

```
            v.push_back(x->value);
            get_values_subtree(x->right, v);
      }
}

template <class T_key, class T_val>
std::vector<T_val> Map<T_key, T_val>::get_values()
{
      std::vector<T_val> v;
      get_values_subtree(root, v);
      return v;
}

template <class T_key, class T_val>
void Map<T_key, T_val>::print_subtree(Node * x)
{
      if (x != nil)
      {
            print_subtree(x->left);
            std::cout << "(key: " << x->key << ", value: " << x->value <<
")" << std::endl;
            print_subtree(x->right);
      }
}

template <class T_key, class T_val>
void Map<T_key, T_val>::print()
{
      print_subtree(root);
      }
```

main.cpp:

```
#include <iostream>
#include "Map.h"
```

```
int main()
{
	Map<std::string, int> map;
	std::cout << "** add 10 elements **" << std::endl;
	for (auto i = 0; i < 10; ++i) {
		std::string key = "hey" + std::to_string(i);
		map.insert(key, i);
	}
	map.print();
	std::cout << "** remove 5 elements **" << std::endl;
	for (auto i = 0; i < 10; i += 2) {
		std::string key = "hey" + std::to_string(i);
		map.remove(key);
	}
	map.print();
	std::cout << "** clear map **" << std::endl;
	map.clear();
	map.print();
}
```

tests.cpp:

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../lab1sem4/Map.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace lab1sem4tests
{
	TEST_CLASS(lab1sem4tests)
	{
	public:
```

```
		TEST_METHOD(insert_one_element)
		{
			std::vector<std::string> keys{ "key0" };
			std::vector<int> values{ 0 };
			Map<std::string, int> map;
			map.insert("key0", 0);
			Assert::IsTrue(keys == map.get_keys() && values ==
map.get_values());
		}
		TEST_METHOD(insert_few_elements)
		{
			std::vector<std::string> keys{ "key0", "key1", "key2"};
			std::vector<int> values{ 0, 1, 2 };
			Map<std::string, int> map;
			map.insert("key0", 0);
			map.insert("key1", 1);
			map.insert("key2", 2);
			Assert::IsTrue(keys == map.get_keys() && values ==
map.get_values());
		}
		TEST_METHOD(remove_few_element)
		{

			std::vector<std::string> keys{ "key0", "key2", "key3" };
			std::vector<int> values{ 0, 2, 3 };
			Map<std::string, int> map;
			map.insert("key0", 0);
			map.insert("key1", 1);
			map.insert("key2", 2);
			map.insert("key3", 3);
			map.insert("key4", 4);
			map.remove("key1");
			map.remove("key4");
```

```
            Assert::IsTrue(keys == map.get_keys() && values ==
map.get_values());
        }
        TEST_METHOD(remove_one_element)
        {

            std::vector<std::string> keys{ };
            std::vector<int> values{ };
            Map<std::string, int> map;
            map.insert("key0", 0);
            map.remove("key0");
            Assert::IsTrue(keys == map.get_keys() && values ==
map.get_values());
        }
        TEST_METHOD(find)
        {
            Map<std::string, int> map;
            map.insert("key0", 0);
            map.insert("key1", 1);
            map.insert("key2", 2);
            Assert::IsTrue(map.find("key0", -1) == 0 &&
map.find("key1", -1) == 1 && map.find("key2", -1) == 2);
        }
        TEST_METHOD(find_not_exist)
        {
            Map<std::string, int> map;
            map.insert("key0", 0);
            map.insert("key1", 1);
            map.insert("key2", 2);
            Assert::IsTrue(map.find("key3", -1) == -1);
        }
        TEST_METHOD(clear)
        {
```

```
            std::vector<std::string> keys{ };
            std::vector<int> values{ };
            Map<std::string, int> map;
            map.insert("key0", 0);
            map.insert("key1", 1);
            map.insert("key2", 2);
            map.insert("key3", 3);
            map.insert("key4", 4);
            map.clear();
            Assert::IsTrue(keys == map.get_keys() && values ==
map.get_values());
        }
        TEST_METHOD(get_keys)
        {
            std::vector<std::string> keys{ "key0", "key1", "key2" };
            Map<std::string, int> map;
            map.insert("key0", 0);
            map.insert("key1", 1);
            map.insert("key2", 2);
            Assert::IsTrue(keys == map.get_keys());
        }
        TEST_METHOD(get_values)
        {
            std::vector<int> values{ 0, 1, 2 };
            Map<std::string, int> map;
            map.insert("key0", 0);
            map.insert("key1", 1);
            map.insert("key2", 2);
            Assert::IsTrue(values == map.get_values());
        }
    };
}
```