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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

"Ассоциативный массив"

Студент гр. 9302 _____ Кузнецов В.В.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2021

Цель работы.

Реализовать ассоциативный массив, основанный на красно-черном дереве.

Постановка задачи.

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева. Наличие unit-тестов ко всем реализуемым методам является обязательным требованием.

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

Описание реализуемого класса и методов.

Были реализованы следующие классы для выполнения лабораторной работы:

1. enum Colors – перечисление из двух цветов BLACK, RED, для идентификации меток узлов дерева;
2. class Node<K, V> - шаблонный класс для представления одного узла дерева, где K – тип ключа, V – тип вершины. Имеет следующие поля:
    a. K key – ключ;
    b. V value – значение;
    c. Colors color – цвет метки;
    d. Node<K, V>* parent, left, right – родительский узел, левый и правый потомки;

3. Class RBTree<K, V> - шаблонный класс для представления красно-черного дерева. Имеет следующие поля:
    a. Node<K, V>* root – указатель на корень дерева;
4. Class Map<K, V> - шаблонный класс для представления ассоциативного массива. Имеет следующие поля:
    a. RBTree<K, V>* tree – красно-черное дерево, которое представляет массив;

Класс Node имеет следующие методы:

1. static Node<K, V>* grandpa(Node<K, V>* n) – возвращает дедушку (родителя родителя) узла n, или нулевой указатель, если его нет.
2. static Node<K, V>* uncle(Node<K, V>* n) – возвращает дядю (ребенка дедушки, помимо родителя) узла n или нулевой указатель, если его нет.
3. static Node<K, V>* sibling(Node<K, V>* n) – возвращает брата (ребенка родителя, помимо самого узла) узла n.
4. static void rotateLeft(Node<K, V>*& n, Node<K, V>*& root) – производит левый поворот относительно узла n в дереве с корнем root.
5. static void rotateRight(Node<K, V>*& n, Node<K, V>*& root) – производит правый поворот относительно узла n в деревне с корнем root.
6. static int comp(Node<K, V>* a, Node<K, V>* b) – функция сравнения двух узлов дерева.

Класс RBTree имеет следующие методы:

1. void insert(K key, V value) – вставка в дерево узла с корнем key и значением value;
2. void fixAfterInsert(Node<K, V>*& n) – перебалансировка дерева после вставки узла n;
3. void remove(K key) – удаление узла с ключом key;
4. void fixAfterRemove(Node<K, V>*& n) – перебалансировка дерева после удаления узла и вставки на его место узла n;
5. Node<K, V>* find(K key) – поиск узла по ключу key. Возвращает найденный узел или нулевой указатель;

6. vector<K> get_keys() – возвращает вектор всех ключей узлов в дереве.
7. void addKey(Node<K, V>* node, vector<K> &keys) – добавление ключа узла node в вектор ключей keys. Рекурсивно вызывает саму себя для потомков node, если они есть, таким образом в ширину обходит все узлы дерева, если вызывать функцию с корня.
8. vector<V> get_values() – возвращает вектор всех значений узлов в дереве.
9. void addValue(Node<K, V>* node, vector<V>& values) – аналогично addKey, только для значений узла node.

Класс Map имеет следующие методы:

1. void insert(K key, V value) – вставка в массив элемента с корнем key и значением value. Вызывает insert у tree;
2. void remove(K key) – удаление элемента с ключом key. Вызывает remove у tree;
3. V find(K key) – поиск элемента по ключу key. Возвращает найденное значение или NULL;
4. Void clear() – очищает массив. Удаляет из памяти существующий tree и создает новый объект класса;
5. vector<K> get_keys() – возвращает вектор всех ключей элементов в массиве. Вызывает get_keys у tree;
6. vector<V> get_values() – возвращает вектор всех значений элементов в массиве. Вызывает get_values у tree;
7. void print() – печатает все элементы массива в формате «ключ значение», каждый с новой строки, в порядке обхода дерева tree в ширину с корня.

Оценка временной сложности методов.

Метод	Оценка
insert	O(log n)
remove	O(log n)
find	O(log n)

clear	O(1)
get_keys	O(n)
get_values	O(n)
print	O(n)

Описание реализованных unit-тестов.

В классе тестирования UnitTest1 были реализованы следующие объекты, на которых проводилось тестирование методов:

1. Map<string, int>* mapStringInt;
2. Map<int, string>* mapIntString;
3. Map<double, double>* mapDoubleDouble;

Реализованные тесты:

1. TestInsertAndFind – тестирование функций вставки и поиска элементов без исключительных ситуаций. Тестирование на всех трех объектах;
2. TestRepeatInsert – тестирование функции вставки элементов при уже существующем ключе. Тестирование проводилось на всех трех объектах.

   Далее тестирование проводилось только на mapStringInt, так как тип ключа и значения уже не имел роли.
3. TestRemove – тестирование функции удаления элементов и поиска несуществующих элементов;
4. TestClear – тестирование функции очищения массива;
5. TestGetKeys – тестирование функции получения вектора ключей;
6. TestGetValues – тестирование функции получения вектора значений;

   Все тесты были пройдены успешно.

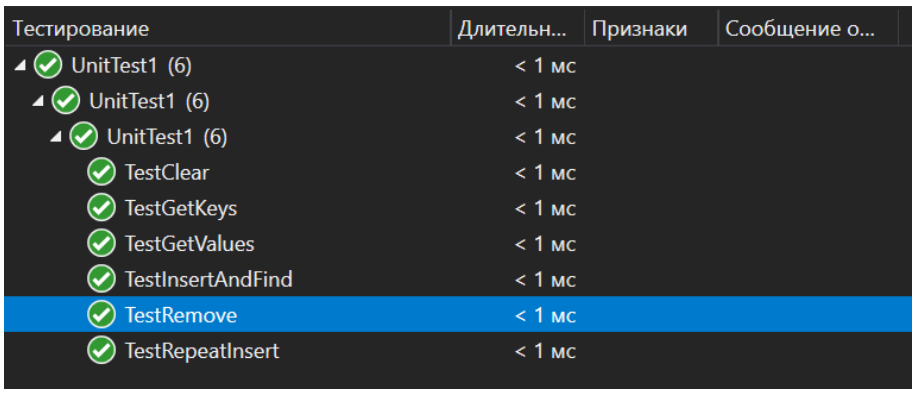

Рисунок 1 – Результаты тестирования

Пример работы программы.

Создадим ассоциативный массив фамилий и должностей, далее удалим пару элементов, выведем результат поиска по массиву и все элементы массива:

```
#include <iostream>
#include "Lab1_Kuznetsov.h"

int main()
{
    auto a = new Map<string, string>();
    a->insert("Kuznetsov", "Director");
    a->insert("Potapov", "Programmer");
    a->insert("Shishkin", "Bloger");
    a->insert("Aptekin", "Designer");
    a->remove("Aptekin");
    std::cout << "Shiskin is " << a->find("Shishkin") << std::endl;
    std::cout << "Peoples:" << std::endl;
    a->print();
    return 0;
}
```

Рисунок 2 – Демонстрационный код

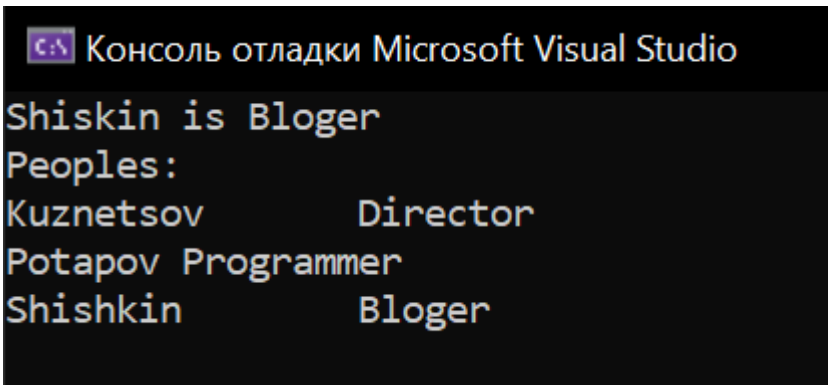

Рисунок 3 – Демонстрация работы

ЛИСТИНГ

Lab1_Kuznetsov.h

```
#pragma once
#include <vector>

using namespace std;

enum Colors {
	BLACK,
	RED
};

template <class K, class V>
class Node {
public:
	K key;
	V value;
	Colors color;
	Node<K, V>* parent;
	Node<K, V>* left;
	Node<K, V>* right;

	Node() {
		this->color = BLACK;
		this->parent = nullptr;
		this->left = nullptr;
		this->right = nullptr;
	}

	Node(K key, V value, Colors color) {
		this->key = key;
		this->value = value;
		this->color = color;
		this->parent = nullptr;
		this->left = nullptr;
		this->right = nullptr;
	}

	static Node<K, V>* grandpa(Node<K, V>* n) {
		if ((n != nullptr) && (n->parent != nullptr)) {
			return n->parent->parent;
		}
		return nullptr;
	}

	static Node<K, V>* uncle(Node<K, V>* n) {
		Node<K, V>* g = grandpa(n);
		if (g == nullptr) {
			return nullptr;
		}
		if (comp(n->parent, g->left) == 0) {
			return g->right;
		}
		return g->left;
	}

	static Node<K, V>* sibling(Node<K, V>* n) {
		if (comp(n, n->parent->left) == 0) {
			return n->parent->right;
		}
		return n->parent->left;
	}
```

```
	static void rotateLeft(Node<K, V>*& n, Node<K, V>*& root) {
		Node<K, V>* pivot = n->right;
		pivot->parent = n->parent;
		if (pivot->parent == nullptr) {
			root = pivot;
		}
		if (n->parent != nullptr) {
			if (comp(n->parent->left, n) == 0) {
				n->parent->left = pivot;
			}
			else {
				n->parent->right = pivot;
			}
		}
		n->right = pivot->left;
		if (pivot->left != nullptr) {
			pivot->left->parent = n;
		}
		n->parent = pivot;
		pivot->left = n;
	}

	static void rotateRight(Node<K, V>*& n, Node<K, V>*& root) {
		Node<K, V>* pivot = n->left;
		pivot->parent = n->parent;
		if (pivot->parent == nullptr) {
			root = pivot;
		}
		if (n->parent != nullptr) {
			if (comp(n->parent->left, n) == 0) {
				n->parent->left = pivot;
			}
			else {
				n->parent->right = pivot;
			}
		}
		n->left = pivot->right;
		if (pivot->right != nullptr) {
			pivot->left->parent = n;
		}
		n->parent = pivot;
		pivot->right = n;
	}

	static int comp(Node<K, V>* a, Node<K, V>* b) {
		if (a == nullptr || b == nullptr) {
			return -1;
		}
		if (a->key < b->key) {
			return -1;
		}
		if (a->key == b->key) {
			return 0;
		}
		return 1;
	}
};

template <class K, class V>
class RBTree {
	Node<K, V>* root;

	void fixAfterInsert(Node<K, V>*& n) {
		if (n->parent == nullptr) {
			n->color = BLACK;
```

```
		}
		else {
			if (n->parent->color == BLACK) {
				return;
			}
			Node<K, V>* u = Node<K, V>::uncle(n);
			Node<K, V>* g = nullptr;
			if ((u != nullptr) && (u->color == RED)) {
				n->parent->color = BLACK;
				u->color = BLACK;
				g = Node<K, V>::grandpa(n);
				g->color = RED;
				fixAfterInsert(g);
			}
			else {
				g = Node<K, V>::grandpa(n);
				if (g != nullptr) {
					if (Node<K, V>::comp(n, n->parent->right) == 0 &&
Node<K, V>::comp(n->parent, g->left) == 0) {
						Node<K, V>::rotateLeft(n->parent, this->root);
						n = n->left;
					}
					else if (Node<K, V>::comp(n, n->parent->left) == 0 &&
Node<K, V>::comp(n->parent, g->right) == 0) {
						Node<K, V>::rotateRight(n->parent, this->root);
						n = n->right;
					}
				}
				n->parent->color = BLACK;
				if (g != nullptr) {
					g->color = RED;
					if (Node<K, V>::comp(n, n->parent->right) == 0 &&
Node<K, V>::comp(n->parent, g->left) == 0) {
						Node<K, V>::rotateRight(g, this->root);
					}
					else if (Node<K, V>::comp(n, n->parent->left) == 0 &&
Node<K, V>::comp(n->parent, g->right) == 0) {
						Node<K, V>::rotateLeft(g, this->root);
					}
				}
			}
		}
	}

	void fixAfterRemove(Node<K, V>*& n) {
		while (Node<K, V>::comp(n, root) != 0 && n->color == BLACK) {
			if (Node<K, V>::comp(n, n->parent->left) == 0) {
				Node<K, V>* w = n->parent->right;
				if (w->color == RED) {
					w->color = BLACK;
					n->parent->color = RED;
					Node<K, V>::rotateLeft(n->parent, this->root);
					w = n->parent->right;
				}
				if (w->left->color == BLACK && w->right->color == BLACK) {
					w->color = RED;
					n = n->parent;
				}
				else {
					if (w->right->color == BLACK) {
						w->left->color = BLACK;
						w->color = RED;
						Node<K, V>::rotateRight(w, this->root);
						w = n->parent->right;
					}
```

```
                                        w->color = n->parent->color;
                                        n->parent->color = BLACK;
                                        w->right->color = BLACK;
                                        Node<K, V>::rotateLeft(n->parent, this->root);
                                        n = this->root;
                                }
                        }
                        else {
                                Node<K, V>* w = n->parent->left;
                                if (w->color == RED) {
                                        w->color = BLACK;
                                        n->parent->color = RED;
                                        Node<K, V>::rotateRight(n->parent, this->root);
                                        w = n->parent->left;
                                }
                                if (w->right->color == BLACK && w->left->color == BLACK) {
                                        w->color = RED;
                                        n = n->parent;
                                }
                                else {
                                        if (w->left->color == BLACK) {
                                                w->right->color = BLACK;
                                                w->color = RED;
                                                Node<K, V>::rotateLeft(w, this->root);
                                                w = n->parent->left;
                                        }
                                        w->color = n->parent->color;
                                        n->parent->color = BLACK;
                                        w->left->color = BLACK;
                                        Node<K, V>::rotateRight(n->parent, this->root);
                                        n = root;
                                }
                        }
                }
                n->color = BLACK;
        }

        void addKey(Node<K, V>* node, vector<K>& keys) {
                keys.push_back(node->key);
                if (node->left != nullptr) {
                        addKey(node->left, keys);
                }
                if (node->right != nullptr) {
                        addKey(node->right, keys);
                }
        }

        void addValue(Node<K, V>* node, vector<V>& values) {
                values.push_back(node->value);
                if (node->left != nullptr) {
                        addValue(node->left, values);
                }
                if (node->right != nullptr) {
                        addValue(node->right, values);
                }
        }

public:
        RBTree() {
                this->root = nullptr;
        }

        void insert(K key, V value) {
                Node<K, V>* n = new Node<K, V>(key, value, RED);
                Node<K, V>* current = this->root;
```

```
        Node<K, V>* parent = nullptr;
        while (current != nullptr) {
                if (current->key == key) {
                        current->value = value;
                        return;
                }
                parent = current;
                current = key < current->key ?
                        current->left : current->right;
        }
        n->parent = parent;
        if (parent != nullptr) {
                if (key < parent->key) {
                        parent->left = n;
                }
                else {
                        parent->right = n;
                }
        }
        else {
                this->root = n;
        }
        this->fixAfterInsert(n);
}

void remove(K key) {
        Node<K, V>* n = find(key);
        if (n == nullptr) return;
        Node<K, V>* x = nullptr;
        Node<K, V>* y = nullptr;
        if (n->left == nullptr && n->right == nullptr) {
                if (n->parent == nullptr) {
                        this->root == nullptr;
                }
                else {
                        if (Node<K, V>::comp(n, n->parent->left) == 0) {
                                n->parent->left = nullptr;
                        }
                        else {
                                n->parent->right = nullptr;
                        }
                }
                return;
        }
        if (n->left == nullptr || n->right == nullptr) {
                y = n;
        }
        else {
                y = n->right;
                while (y->left != nullptr) {
                        y = y->left;
                }
        }
        if (y->left != nullptr) {
                x = y->left;
        }
        else {
                x = y->right;
        }
        x->parent = y->parent;
        if (y->parent != nullptr) {
                if (Node<K, V>::comp(y, y->parent->left) == 0) {
                        y->parent->left = x;
                }
                else {
```

```
                        y->parent->right = x;
                    }
                }
                else {
                    this->root = x;
                }
                if (Node<K, V>::comp(y, n) != 0) {
                    n->value = y->value;
                    n->key = y->key;
                }
                if (y->color == BLACK) {
                    fixAfterRemove(x);
                }
        }

        Node<K, V>* find(K key) {
                Node<K, V>* curr = this->root;
                while (curr != nullptr && curr->key != key) {
                    if (curr->key < key) {
                        curr = curr->right;
                    }
                    else {
                        curr = curr->left;
                    }
                }
                return curr;
        }

        vector<K> get_keys() {
                vector<K> keys;
                if (this->root != nullptr) {
                    addKey(this->root, keys);
                }
                return keys;
        }

        vector<V> get_values() {
                vector<V> values;
                if (this->root != nullptr) {
                    addValue(this->root, values);
                }
                return values;
        }
};

template <class K, class V>
class Map {
        RBTree<K, V>* tree;

public:
        Map() {
                tree = new RBTree<K, V>();
        }

        void insert(K key, V value) {
                tree->insert(key, value);
        }

        void remove(K key) {
                tree->remove(key);
        }

        V find(K key) {
                Node<K, V>* node = tree->find(key);
                if (node == nullptr) {
```

```
                throw 0;
            }
            else {
                return node->value;
            }
        }

        void clear() {
            delete tree;
            tree = new RBTree<K, V>();
        }

        vector<K> get_keys() {
            return tree->get_keys();
        }

        vector<V> get_values() {
            return tree->get_values();
        }

        void print() {
            vector<K> k = tree->get_keys();
            vector<V> v = tree->get_values();
            for (int i = 0; i < k.size(); i++) {
                cout << k[i] << '\t' << v[i] << endl;
            }
        }
};
```

UnitTest1.cpp

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Lab1_Kuznetsov/Lab1_Kuznetsov.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
    TEST_CLASS(UnitTest1)
    {
    public:
        Map<string, int>* mapStringInt = new Map<string, int>();
        Map<int, string>* mapIntString = new Map<int, string>();
        Map<double, double>* mapDoubleDouble = new Map<double, double>();

        TEST_METHOD(TestInsertAndFind)
        {
            mapStringInt->insert("abc", 1);
            mapStringInt->insert("ade", 2);
            mapStringInt->insert("bbc", 3);

            mapIntString->insert(1, "a");
            mapIntString->insert(2, "abc");
            mapIntString->insert(3, "def");

            mapDoubleDouble->insert(3.52, 2.64);
            mapDoubleDouble->insert(3.53, 2.10934);
            mapDoubleDouble->insert(3.5333, 2.69402);

            Assert::AreEqual(1, mapStringInt->find("abc"));
            Assert::AreEqual((string)"def", mapIntString->find(3));
            Assert::AreEqual(2.69402, mapDoubleDouble->find(3.5333));
        }
```

```
TEST_METHOD(TestRepeatInsert) {
	mapStringInt->insert("abc", 1);
	mapStringInt->insert("abc", 4);

	mapIntString->insert(1, "a");
	mapIntString->insert(1, "abc");

	mapDoubleDouble->insert(3, 2.66434);
	mapDoubleDouble->insert(3, 2.10934);

	Assert::AreEqual(4, mapStringInt->find("abc"));
	Assert::AreEqual((string)"abc", mapIntString->find(1));
	Assert::AreEqual(2.10934, mapDoubleDouble->find(3));
}

TEST_METHOD(TestRemove) {
	bool notFindMarker = false;

	mapStringInt->insert("abc", 1);
	mapStringInt->insert("ade", 2);
	mapStringInt->insert("bbc", 3);

	mapStringInt->remove("bbc");
	mapStringInt->remove("ade");

	try {
		mapStringInt->find("bbc");
	}
	catch (int) {
		notFindMarker = true;
	}

	Assert::IsTrue(notFindMarker);
	Assert::AreEqual(1, mapStringInt->find("abc"));

	mapStringInt->remove("abc");
	mapStringInt->insert("bse", 1);

	Assert::AreEqual(1, mapStringInt->find("bse"));
}

TEST_METHOD(TestClear) {
	bool notFindMarker = false;

	mapStringInt->insert("abc", 1);
	mapStringInt->insert("ade", 2);
	mapStringInt->insert("bbc", 3);

	mapStringInt->clear();

	try {
		mapStringInt->find("bbc");
	}
	catch (int) {
		notFindMarker = true;
	}

	Assert::IsTrue(notFindMarker);
	mapStringInt->insert("bse", 1);
	Assert::AreEqual(1, mapStringInt->find("bse"));
}

TEST_METHOD(TestGetKeys) {
	mapStringInt->insert("abc", 1);
```

```
			mapStringInt->insert("ade", 2);
			mapStringInt->insert("bbc", 3);
			mapStringInt->insert("bec", 4);
			mapStringInt->insert("bec", 5);

			auto k = mapStringInt->get_keys();
			Assert::AreEqual(4, (int)k.size());
			Assert::AreEqual((string)"abc", k[0]);
			Assert::AreEqual((string)"ade", k[1]);
			Assert::AreEqual((string)"bbc", k[2]);
			Assert::AreEqual((string)"bec", k[3]);
		}

		TEST_METHOD(TestGetValues) {
			mapStringInt->insert("abc", 1);
			mapStringInt->insert("ade", 2);
			mapStringInt->insert("bbc", 3);
			mapStringInt->insert("bec", 4);
			mapStringInt->insert("bec", 5);

			auto v = mapStringInt->get_values();
			Assert::AreEqual(4, (int)v.size());
			Assert::AreEqual(1, v[0]);
			Assert::AreEqual(2, v[1]);
			Assert::AreEqual(3, v[2]);
			Assert::AreEqual(5, v[3]);
		}
	};
}
```