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»
Вариант №1

Студент гр. 8302 _____ Никулин Л. А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2020

1.Цель работы

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

2.Описание реализуемого класса и методов

Map – основной класс для реализации функций

Tree – класс для реализации красно-чёрное дерево.

Node – класс элемента дерева.

insert (TKey, TValue)	добавление элемента с ключом и значением
remove(TKey)	удаление элемента дерева по ключу
find(TKey)	поиск элемента
clear()	очищение ассоциативного массива
get_keys()	возвращение списка ключей
get_values()	возвращение списка значений
print()	вывод дерева в консоль

3.Оценка временной сложности

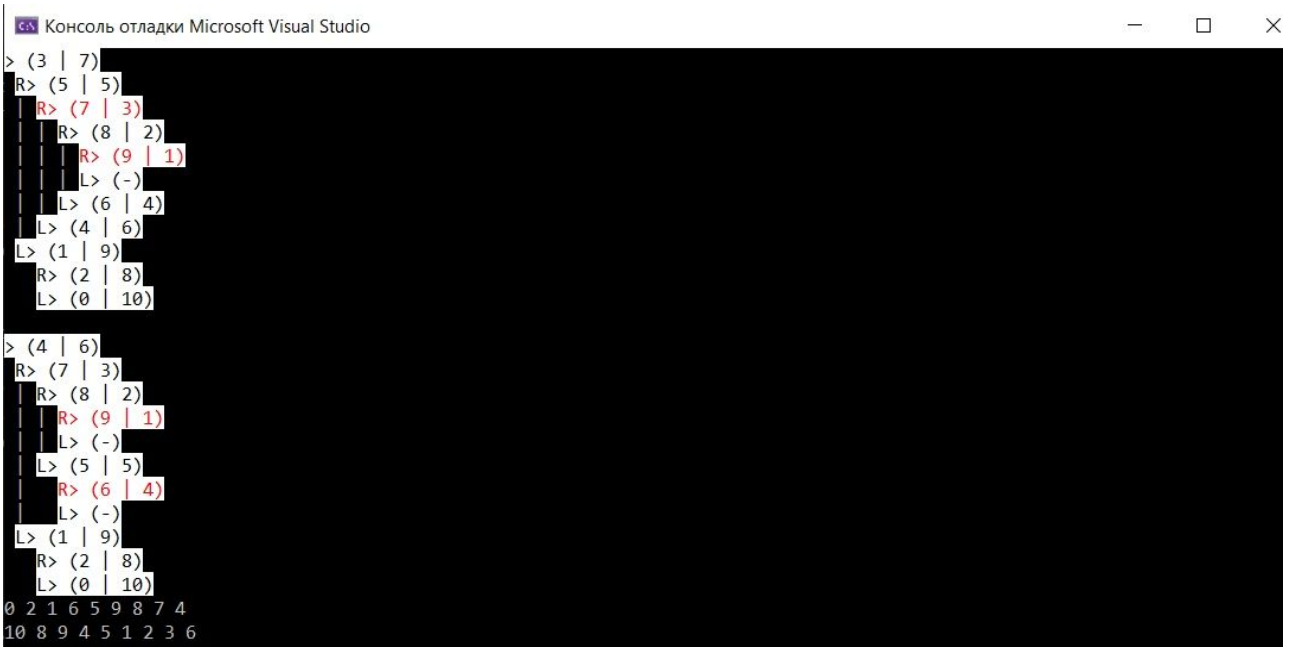
insert(TKey, TValue)	O(logN)
remove(TKey)	O(logN)
find(TKey)	O(logN)
clear()	O(N)
get_keys()	O(N)
get_values()	O(N)
print()	O(N)

4.Описание реализованных Unit-тестов.

find_test	проверяет нахождение элемента в контейнере
remove_test	проверяет удаление элемента из контейнера
remove_wrong_index_text	проверяет удаление несуществующего элемента в контейнере (выброс исключения)
clear_test	проверяет функцию полного очищения контейнера
clear_empty_map_test	проверяет работу функции очищения для пустого контейнера (выброс исключения)
get_keys_test	проверяет функцию возвращения списка List ключей дерева
get_values_test	проверяет функцию возвращения списка List значений дерева.
insert_test	проверяет вставку элемента в контейнер

5.Пример работы программы

```cpp
int main()
{
    map<int, int> map;

    for (int i = 0; i < 10; ++i)
    {
        map.insert(i, 10 - i);
    }

    map.print();

    map.remove(3);

    cout << endl;

    map.print();

    List<int> keys = map.get_keys();
    while (!keys.isEmpty())
    {
        cout << keys.next() << " ";
    }

    cout << endl;

    List<int> values = map.get_values();
    while (!values.isEmpty())
    {
        cout << values.next() << " ";
    }
}
```



6.Листинг

map.h

```
#include <Windows.h>
#include "list.h"
#include <exception>
using namespace std;

typedef
enum { BLACK, RED } nodeColor;   // Color of elements

#define NIL nil // NIL-ELEMENT

template <typename TKey, typename TValue>
class map {
private:
	class Tree;
	Tree* tree_of_elements; // main tree for map
public:
	map()
	{
		tree_of_elements = new Tree;
	}
	typename Tree::Node* insert(TKey, TValue);
	void remove(TKey);
	typename Tree::Node* find(TKey);
	void clear();
	List<TKey> get_keys();
	List<TValue> get_values();
	void print();
};

template <typename TKey, typename TValue>
class map<TKey, TValue>::Tree {
private:
	class Node {
	public:
		Node* right;
		Node* left;
		Node* parent = nullptr;
		pair <TKey, TValue> data;
		nodeColor color = BLACK;
	};
	void fixInsert(Node*);
	void fixDelete(Node*);
	void rotateLeft(Node*);
	void rotateRight(Node*);
```

```
        void clear(Node *);
        void get_values(typename Node*, List<TValue>&);
        void get_keys(typename Node *, List<TKey> &);
        void print(Node*, string);
        Node* nil = new Node;
public:
        friend class map<TKey, TValue>;          //      access to fields
        typename Node* insert(TKey, TValue);
        void deleteNode(Node *);
        Node* find(TKey);
        Node* root = NIL;
};

template <typename TKey, typename TValue>
void map<TKey, TValue>::remove(TKey key)
{
        auto node = find(key);
        if (node == nullptr) throw exception("There is no element in the tree");
        tree_of_elements->deleteNode(node);
}

template <typename TKey, typename TValue>
List<TValue> map<TKey, TValue>::get_values() {
        List<TValue> list;
        tree_of_elements->get_values(tree_of_elements->root, list);
        return list;
}

template <typename TKey, typename TValue>
List<TKey> map<TKey, TValue>::get_keys() {
        List<TKey> list;
        tree_of_elements->get_keys(tree_of_elements->root, list);
        return list;
}

template <typename TKey, typename TValue>
typename map<TKey, TValue>::Tree::Node* map<TKey, TValue>::insert(TKey key,
TValue value)
{
        return tree_of_elements->insert(key,value);
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::print()
{
        tree_of_elements->print(tree_of_elements->root, "");
}
```

```
template <typename TKey, typename TValue>
typename map<TKey, TValue>::Tree::Node* map<TKey, TValue>::find(TKey key)
{
	return tree_of_elements->find(key);
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::clear()
{
	tree_of_elements->clear(tree_of_elements->root);
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::fixDelete(Node* node)
{

	while (node != root && node->color == BLACK) {
		if (node == node->parent->left)
		{
			Node* brother = node->parent->right;
			if (brother->color == RED)
			{
				brother->color = BLACK;
				node->parent->color = RED;
				rotateLeft(node->parent);
				brother = node->parent->right;
			}
			if (brother->left->color == BLACK && brother->right->color ==
BLACK)
			{
				brother->color = RED;
				node = node->parent;
			}
			else
			{
				if (brother->right->color == BLACK)
				{
					brother->left->color = BLACK;
					brother->color = RED;
					rotateRight(brother);
					brother = node->parent->right;
				}
				brother->color = node->parent->color;
				node->parent->color = BLACK;
				brother->right->color = BLACK;
				rotateLeft(node->parent);
				node = root;
			}
```

```
                }
                else
                {
                        Node* brother = node->parent->left;
                        if (brother->color == RED)
                        {
                                brother->color = BLACK;
                                node->parent->color = RED;
                                rotateRight(node->parent);
                                brother = node->parent->left;
                        }
                        if (brother->right->color == BLACK && brother->left->color ==
BLACK)
                        {
                                brother->color = RED;
                                node = node->parent;
                        }
                        else
                        {
                                if (brother->left->color == BLACK)
                                {
                                        brother->right->color = BLACK;
                                        brother->color = RED;
                                        rotateLeft(brother);
                                        brother = node->parent->left;
                                }
                                brother->color = node->parent->color;
                                node->parent->color = BLACK;
                                brother->left->color = BLACK;
                                rotateRight(node->parent);
                                node = root;
                        }
                }
        }
        node->color = BLACK;
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::deleteNode(Node* node) {
        Node *childOfDeleteElement, *willDelete;
        if (!node || node == NIL) return;
        if (node->left == NIL || node->right == NIL) {
                willDelete = node;
        }
        else {
                willDelete = node->right;
                while (willDelete->left != NIL) willDelete = willDelete->left;
        }
```

```
	if (willDelete->left != NIL) childOfDeleteElement = willDelete->left;
	else childOfDeleteElement = willDelete->right;
	childOfDeleteElement->parent = willDelete->parent;
	if (willDelete->parent)
		if (willDelete == willDelete->parent->left)
			willDelete->parent->left = childOfDeleteElement;
		else
			willDelete->parent->right = childOfDeleteElement;
	else
		root = childOfDeleteElement;

	if (willDelete != node) node->data = willDelete->data;

	if (willDelete->color == BLACK)
		fixDelete(childOfDeleteElement);

	delete willDelete;
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::get_keys(typename Tree::Node* node, List<TKey>&
list)
{
	if (root == NIL) return;
	if (node->left != NIL) get_keys(node->left, list);
	if (node->right != NIL) get_keys(node->right, list);
	list.push_back(node->data.first);
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::get_values(typename Tree::Node* node,
List<TValue>& list)
{
	if (root == NIL) return;
	if (node->left != NIL) get_values(node->left, list);
	if (node->right != NIL) get_values(node->right, list);
	list.push_back(node->data.second);
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::clear(typename Tree::Node* node)
{
	if(root == NIL) throw exception("There is no elements in the tree");
	if (node->left != NIL) clear(node->left);
	if (node->right != NIL) clear(node->right);
	if (node == root) root = NIL;
	delete node;
}
```

```
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::print(typename Tree::Node* root, string str)
{
	if (root == NIL) return;
	HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
	if (root == this->root)
	{
		SetConsoleTextAttribute(hConsole, (WORD)((15 << 4) | 0));
		cout << "> (" << root->data.first << " | " << root->data.second << ")" <<
endl;
		SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
		str += " ";
	}
	if (root->right != NIL) {
		string _str = str;
		cout << _str;
		if (root->right->color == BLACK)
			SetConsoleTextAttribute(hConsole, (WORD)((15 << 4) | 0));
		else SetConsoleTextAttribute(hConsole, (WORD)((15 << 4) | 12));
		cout << "R> (" << root->right->data.first << " | " <<
root->right->data.second << ")" << endl;
		SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
		_str += "| ";
		print(root->right, _str);
	}
	else if (root->left != NIL) {
		cout << str;
		SetConsoleTextAttribute(hConsole, (WORD)((15 << 4) | 0));
		cout << "R> (-)" << endl;
		SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
	}
	if (root->left != NIL) {
		string _str = str;
		cout << _str;
		if (root->left->color == BLACK)
			SetConsoleTextAttribute(hConsole, (WORD)((15 << 4) | 0));
		else SetConsoleTextAttribute(hConsole, (WORD)((15 << 4) | 12));
		cout << "L> (" << root->left->data.first << " | " << root->left->data.second
<< ")" << endl;
		SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
		_str += "   ";
		print(root->left, _str);
	}
	else if (root->right != NIL) {
		cout << str;
		SetConsoleTextAttribute(hConsole, (WORD)((15 << 4) | 0));
		cout << "L> (-)" << endl;
```

```
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
        }
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::rotateLeft(Node* node)
{
        Node* rightSon = node->right;
        node->right = rightSon->left;
        if (rightSon->left != NIL) rightSon->left->parent = node;
        if (rightSon != NIL) rightSon->parent = node->parent;
        if (node->parent) {
                if (node == node->parent->left)
                        node->parent->left = rightSon;
                else
                        node->parent->right = rightSon;
        }
        else {
                root = rightSon;
        }
        rightSon->left = node;
        if (node != NIL) node->parent = rightSon;
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::rotateRight(Node* node) {
        Node* leftSon = node->left;
        node->left = leftSon->right;
        if (leftSon->right != NIL) leftSon->right->parent = node;
        if (leftSon != NIL) leftSon->parent = node->parent;
        if (node->parent) {
                if (node == node->parent->right)
                        node->parent->right = leftSon;
                else
                        node->parent->left = leftSon;
        }
        else {
                root = leftSon;
        }
        leftSon->right = node;
        if (node != NIL) node->parent = leftSon;
}

template <typename TKey, typename TValue>
typename map<TKey, TValue>::Tree::Node* map<TKey, TValue>::Tree::find(TKey key)
{
        Node* current = root;
        while (current != NIL)
```

```
		if (key == current->data.first)
			return current;
		else
		{
			current = key < current->data.first ? current->left : current->right;
		}
	return nullptr;
}

template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::fixInsert(Node* node)
{
	while (node != root && node->parent->color == RED) {
		if (node->parent == node->parent->parent->left) {
			Node* uncle = node->parent->parent->right;
			if (uncle->color == RED) {
				node->parent->color = BLACK;
				uncle->color = BLACK;
				node->parent->parent->color = RED;
				node = node->parent->parent;
			}
			else {
				if (node == node->parent->right) {
					node = node->parent;
					rotateLeft(node);
				}
				node->parent->color = BLACK;
				node->parent->parent->color = RED;
				rotateRight(node->parent->parent);
			}
		}
		else {
			Node* uncle = node->parent->parent->left;
			if (uncle->color == RED) {
				node->parent->color = BLACK;
				uncle->color = BLACK;
				node->parent->parent->color = RED;
				node = node->parent->parent;
			}
			else {
				if (node == node->parent->left) {
					node = node->parent;
					rotateRight(node);
				}
				node->parent->color = BLACK;
				node->parent->parent->color = RED;
				rotateLeft(node->parent->parent);
			}
```

```
                }
        }
        root->color = BLACK;
}

template <typename TKey, typename TValue>
typename map<TKey, TValue>::Tree::Node* map<TKey, TValue>::Tree::insert(TKey
key, TValue value)
{
        Node *current, *newNode, *parent;
        current = root;
        parent = 0;
        while (current != NIL) {
                if (key == current->data.first) return current;
                parent = current;
                current = key < current->data.first ? current->left : current->right;
        }
        newNode = new Node;
        newNode->data = make_pair(key, value);
        newNode->parent = parent;
        newNode->left = NIL;
        newNode->right = NIL;
        newNode->color = RED;
        if (parent) {
                if (key < parent->data.first)
                        parent->left = newNode;
                else
                        parent->right = newNode;
        }
        else {
                root = newNode;
        }

        fixInsert(newNode);
        return newNode;
}
```

list.h

```
template <typename TValue>
class List
{
private:
        class Node
        {
        public:
                Node* next = nullptr;
                TValue data;
        };
```

```
	Node* tail = nullptr;
	Node* head = nullptr;
public:
	void push_back(TValue element)
	{
		if (!tail)
		{
			tail = head = new Node;
			tail->data = element;
		}
		else
		{
			tail->next = new Node;
			tail = tail->next;
			tail->data = element;
		}
	}
	TValue next()
	{
		TValue value = head->data;
		head = head->next;
		return value;
	}
	bool isEmpty()
	{
		if (!head) return true;
		else return false;
	}
};
```

lab1.cpp

```
#include <iostream>
#include "map.h"
using namespace std;

int main()
{
	map<int, int> map;

	for (int i = 0; i < 10; ++i)
	{
		map.insert(i, 10 - i);
	}

	map.print();

	map.remove(3);
```

```
	cout << endl;

	map.print();

	List<int> keys = map.get_keys();
	while (!keys.isEmpty())
	{
		cout << keys.next() << " ";
	}

	cout << endl;

	List<int> values = map.get_values();
	while (!values.isEmpty())
	{
		cout << values.next() << " ";
	}
}
```

Lab1Tests.cpp

```
#include "CppUnitTest.h"
#include "../lab1/map.h"
#include <stdexcept>

using namespace std;
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Lab1Tests
{
	TEST_CLASS(Lab1Tests)
	{
	private:
		map<int, int> card;
		List<int> list;
	public:
		TEST_METHOD(insert_test)
		{
			bool before = card.find(5);
			card.insert(5, 1);
			bool after = card.find(5);
			Assert::AreEqual(!before, after);
		}
		TEST_METHOD(remove_test)
		{
			card.insert(5, 1);
			bool before = card.find(5);
			card.remove(5);
			bool after = card.find(5);
```

```
		Assert::AreEqual(before, !after);
	}
	TEST_METHOD(remove_wrong_index_text)
	{
		card.insert(5, 1);
		card.insert(6, 2);
		card.insert(7, 3);
		try
		{
			card.remove(8);
		}
		catch (const std::exception &ex)
		{
			Assert::AreEqual(ex.what(), "There is no element in the
tree");
		}
	}
	TEST_METHOD(clear_test)
	{
		card.insert(5, 1);
		card.insert(6, 2);
		card.clear();
		Assert::AreEqual(!card.find(5), !card.find(6));
	}
	TEST_METHOD(clear_empty_map_test)
	{
		try
		{
			card.clear();
		}
		catch (const std::exception& ex)
		{
			Assert::AreEqual(ex.what(), "There is no elements in the
tree");
		}
	}
	TEST_METHOD(find_test)
	{
		bool before = card.find(5);
		card.insert(5, 1);
		bool after = card.find(5);
		Assert::AreEqual(!before, after);
	}
	TEST_METHOD(get_keys_test)
	{
		card.insert(5, 1);
		card.insert(6, 2);
		card.insert(7, 3);
```

```
			list = card.get_keys();
			int sum_of_keys = 0;
			while (!list.isEmpty())
				sum_of_keys += list.next();
			Assert::IsTrue(sum_of_keys == 18);
		}
		TEST_METHOD(get_values_test)
		{
			card.insert(5, 1);
			card.insert(6, 2);
			card.insert(7, 3);
			list = card.get_values();
			int sum_of_values = 0;
			while (!list.isEmpty())
				sum_of_values += list.next();
			Assert::IsTrue(sum_of_values == 6);
		}
	};
}
```