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. 8301	________________	Бобров А.Б.
Преподаватель	________________	Тутуева А.В.

Санкт-Петербург
2020

Цель работы

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

Описание реализуемого класса и методов

Классы:

map – основной реализуемый класс.

Tree – класс, с помощью которого реализуется красно-черное дерево.

Node – данный класс представляет собой элемента дерева.

Методы:

insert(ключ, значение) - добавление элемента с ключом и значением.

get_values() - возвращает список значений

remove(ключ) - удаление элемента дерева по ключу.

find(ключ) - поиск элемента.

clear() - очищение ассоциативного массива.

get_keys() - возвращает список ключей.

print() - вывод дерева.

Оценка временной сложности каждого метода

insert(ключ, значение) – O(logN)

get_values() – O(N)

remove(ключ) – O(logN)

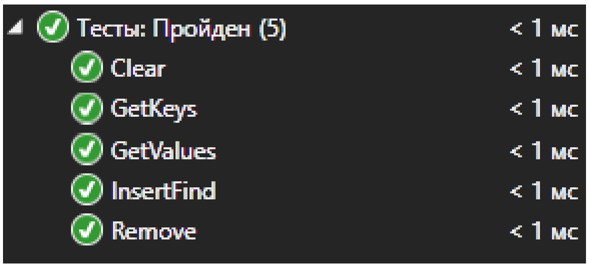
find(ключ) – O(logN)

clear() – O(N)

get_keys() – O(N)

print() – O(N)

Описание реализованных Unit-тестов



```
◢ ✓ Тесты: Пройден (5)      < 1 мс
     ✓ Clear                < 1 мс
     ✓ GetKeys              < 1 мс
     ✓ GetValues            < 1 мс
     ✓ InsertFind           < 1 мс
     ✓ Remove               < 1 мс
```

Описание методов:

Clear – производит проверку функции очищения.

GetKeys – проверяет функцию получения списка ключей.

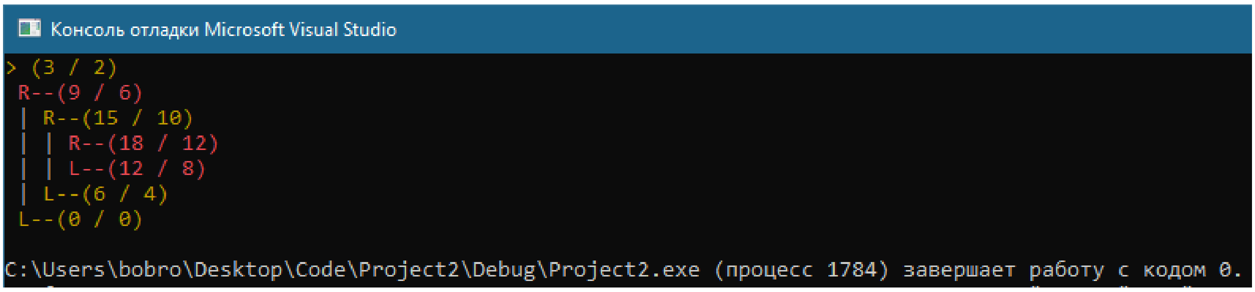
GetValues – тестирует функцию возвращающую список значений.

InsertFind – проверяет на работоспособность функции вставки и поиска.

Remove – соответственно проверяет функцию удаления элемента с помощью ключа.

Пример работы программы

После выполнения программы консоль будет иметь такой вид:



```
> (3 / 2)
 R--(9 / 6)
 | R--(15 / 10)
 | | R--(18 / 12)
 | | L--(12 / 8)
 | L--(6 / 4)
 L--(0 / 0)

C:\Users\bobro\Desktop\Code\Project2\Debug\Project2.exe (процесс 1784) завершает работу с кодом 0.
```

Листинг

main.cpp:

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

int main()
{
	map<int, int> x;
	for (int i = 0; i < 7; i++)
	{
		x.insert(i*3, i*2);
	}
	x.print();
}
```

map.h:

```
#pragma once
#include <Windows.h>
#include <exception>
#include <string>
using namespace std;
typedef enum { BLACK, RED } nodeColor;

//LIST//
template <typename T>
class List
```

```
{
	private:
		class Node
		{
			public:
				Node* next = nullptr;
				T info;
		};
		Node* end = nullptr;
		Node* current = nullptr;
		Node* start = nullptr;
	public:
		void newElement(T element)
		{
			if (!end)
			{
				end = start = current = new Node;
				end->info = element;
			}
			else
			{
				end->next = new Node;
				end = end->next;
				end->info = element;
			}
		}
		T next()
		{
			if (current)
			{
				T value = current->info;
				current = current->next;
				return value;
			}
		}
		bool isCurrent()
		{
			return current ? true : false;
		}
};

//MAP//
template <typename TKey, typename TValue>
class map
{
	private:
		class Tree;
		Tree* tree;
	public:
		map()
		{
			tree = new Tree;
		}

		//insert element with key & value
		typename Tree::Node* insert(TKey, TValue);

		//removing element of tree using key
		void remove(TKey);

		//search of element
		typename Tree::Node* find(TKey);

		//clear associative array
		void clear();

		//return list of keys
		List<TKey> get_keys();

		//return list of values
		List<TValue> get_values();

		//print tree
		void print();
};

//insert
```

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

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

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

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

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

//remove
template <typename TKey, typename TValue>
void map<TKey, TValue>::remove(TKey key)
{
        auto node = find(key);
        if (node == nullptr) throw exception("Tree is empty");
        tree->deleteNode(node);
}

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



//TREE//
template <typename TKey, typename TValue>
class map<TKey, TValue>::Tree
{
        private:
                friend class map<TKey, TValue>;
                class Node
                {
                        public:
                                Node* right;
                                Node* left;
                                Node* parent = nullptr;
                                pair <TKey, TValue> info;
                                nodeColor color = BLACK;
                };

                void InsFix(Node*);

                void DelFix(Node*);

                void get_keys(typename Node *, List<TKey> &);
                void get_values(typename Node*, List<TValue>&);

                void Rotate_L(Node*);

                void Rotate_R(Node*);

                void print(Node*, string);

                void clear(Node *);

                Node* NN = new Node;
        public:
                typename Node* insert(TKey, TValue);
                void deleteNode(Node *);
                Node* find(TKey);
                Node* root = NN;
};


//InsFix
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::InsFix(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)
                        {
                                //uncle - red
                                node->parent->color = BLACK;
                                uncle->color = BLACK;
                                node->parent->parent->color = RED;
                                node = node->parent->parent;
                        }
                        else
                        {
                                //uncle - black
                                if (node == node->parent->right)
                                {
                                        //make node a left child
                                        node = node->parent;
                                        Rotate_L(node);
                                }
                                //change color & rotate
                                node->parent->color = BLACK;
                                node->parent->parent->color = RED;
                                Rotate_R(node->parent->parent);
                        }
                }
                else
                {
                        Node* uncle = node->parent->parent->left;
                        if (uncle->color == RED)
                        {
                                //uncle - red
                                node->parent->color = BLACK;
                                uncle->color = BLACK;
                                node->parent->parent->color = RED;
                                node = node->parent->parent;
                        }
                        else
                        {
                                //uncle - black
                                if (node == node->parent->left)
                                {
                                        node = node->parent;
                                        Rotate_R(node);
                                }
                                node->parent->color = BLACK;
                                node->parent->parent->color = RED;
                                Rotate_L(node->parent->parent);
                        }
                }
        }
```

```
	root->color = BLACK;
}

//DelFix
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::DelFix(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;
				Rotate_L(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;
					Rotate_R(brother);
					brother = node->parent->right;
				}
				brother->color = node->parent->color;
				node->parent->color = BLACK;
				brother->right->color = BLACK;
				Rotate_L(node->parent);
				node = root;
			}
		}
		else
		{
			Node* brother = node->parent->left;
			if (brother->color == RED)
			{
				brother->color = BLACK;
				node->parent->color = RED;
				Rotate_R(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;
					Rotate_L(brother);
					brother = node->parent->left;
				}
				brother->color = node->parent->color;
				node->parent->color = BLACK;
				brother->left->color = BLACK;
				Rotate_R(node->parent);
				node = root;
			}
		}
	}
	node->color = BLACK;
		}

//DelNode
template <typename TKey, typename TValue>
```

```
void map<TKey, TValue>::Tree::deleteNode(Node* node)
{
	Node *child_of_RemElement, *removable;
	if (!node || node == NN) return;
	if (node->left == NN || node->right == NN)
	{
		removable = node;
	}
	else
	{
		removable = node->right;
		while (removable->left != NN) removable = removable->left;
	}
	if (removable->left != NN)
		child_of_RemElement = removable->left;
	else
		child_of_RemElement = removable->right;
		child_of_RemElement->parent = removable->parent;
		if (removable->parent)
		if (removable == removable->parent->left)
			removable->parent->left = child_of_RemElement;
		else
			removable->parent->right = child_of_RemElement;
	else
		root = child_of_RemElement;
		if (removable != node) node->info = removable->info;
		if (removable->color == BLACK)
		DelFix(child_of_RemElement);
		delete removable;
}

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

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

//Rotate_L
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::Rotate_L(Node* node)
{
	//rotate node x to left
	Node* rightSon = node->right;
	//establish x->right link
	node->right = rightSon->left;
	if (rightSon->left != NN) rightSon->left->parent = node;
	//establish y->parent link
	if (rightSon != NN) rightSon->parent = node->parent;
	if (node->parent)
	{
		if (node == node->parent->left)
			node->parent->left = rightSon;
		else
			node->parent->right = rightSon;
	}
	else
	{
		root = rightSon;
	}
	//link x and y
	rightSon->left = node;
	if (node != NN) node->parent = rightSon;
}
```

```
//Rotate_R
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::Rotate_R(Node* node)
{
        //rotate node x to right
        Node* leftSon = node->left;
        //establish x->left link
        node->left = leftSon->right;
        if (leftSon->right != NN) leftSon->right->parent = node;
        //establish y->parent link
        if (leftSon != NN) leftSon->parent = node->parent;
        if (node->parent)
        {
                if (node == node->parent->right)
                        node->parent->right = leftSon;
                else
                        node->parent->left = leftSon;
        }
        else
        {
                root = leftSon;
        }
        // link x and y
        leftSon->right = node;
        if (node != NN) node->parent = leftSon;
}

//print
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::print(typename Tree::Node* root, string str)
{
        if (root == NN) return;
        HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
        if (root == this->root)
        {
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));
                cout << "> (" << root->info.first << " / " << root->info.second << ")" << endl;
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
                str += " ";
        }
        if (root->right != NN)
        {
                string _str = str;
                cout << _str;
                if (root->right->color == BLACK)
                        SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));
                else SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 12));
                cout << "R--(" << root->right->info.first << " / " << root->right->info.second << ")" <<
endl;
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
                _str += "| ";
                print(root->right, _str);
        }
        else if (root->left != NN)
        {
                cout << str;
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));
                cout << "R--(-)" << endl;
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
        }
        if (root->left != NN)
        {
                string _str = str;
                cout << _str;
                if (root->left->color == BLACK)
                        SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));
                else SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 12));
                cout << "L--(" << root->left->info.first << " / " << root->left->info.second << ")" <<
endl;
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
                _str += "  ";
                print(root->left, _str);
        }
        else if (root->right != NN)
        {
                cout << str;
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));
                cout << "L--(-)" << endl;
                SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));
        }
}
//clear
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::clear(typename Tree::Node* node)
{
        if (node->left) clear(node->left);
        if (node->right) clear(node->right);
        if (node == root) root = NN;
        delete node;
}

//find
template <typename TKey, typename TValue>
typename map<TKey, TValue>::Tree::Node* map<TKey, TValue>::Tree::find(TKey key)
{
        Node* current = root;
        while (current != NN)
                if (key == current->info.first)
                        return current;
                else
                {
                        current = key < current->info.first ? current->left : current->right;
                }
        return nullptr;
}

//insert
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 != NN)
        {
                if (key == current->info.first) return current;
                parent = current;
                current = key < current->info.first ? current->left : current->right;
        }
        newNode = new Node;
        newNode->info = make_pair(key, value);
        newNode->parent = parent;
        newNode->left = NN;
        newNode->right = NN;
        newNode->color = RED;
        //insert node to the tree
        if (parent)
        {
                if (key < parent->info.first)
                        parent->left = newNode;
                else
                        parent->right = newNode;
        }
        else
        {
                root = newNode;
        }
        InsFix(newNode);
        return newNode;
}
```

Unit-тесты:

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Project2/map.h"
#include <stdexcept>

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest2
{
        TEST_CLASS(UnitTest2)
        {
        public:

                TEST_METHOD(InsertFind)
                {
                        map<int, int> map;
                        bool bef = map.find(4);
                        map.insert(4, 1);
                        bool aft = map.find(4);
                        Assert::AreEqual(!bef, aft);
                }

                TEST_METHOD(GetKeys)
                {
                        map<int, int> map;
                        map.insert(4, 1);
                        map.insert(5, 2);
                        List<int> list = map.get_keys();
                        int sum = 0;
                        while (list.isCurrent())
                                sum += list.next();
                        Assert::IsTrue(sum == 9);
                }

                TEST_METHOD(GetValues)
                {
                        map<int, int> map;
                        map.insert(4, 1);
                        map.insert(5, 2);
                        List<int> list = map.get_values();
                        int sum = 0;
                        while (list.isCurrent())
                                sum += list.next();
                        Assert::IsTrue(sum == 3);
                }

                TEST_METHOD(Remove)
                {
                        map<int, int> map;
                        map.insert(5, 1);
                        bool bef = map.find(5);
                        map.remove(5);
                        bool aft = map.find(5);
                        Assert::AreEqual(bef, !aft);
                }

                TEST_METHOD(Clear)
                {
                        map<int, int> map;
                        map.insert(4, 1);
                        map.insert(5, 2);
                        map.clear();
                        Assert::AreEqual(!map.find(4), !map.find(5));
                }
        };
}
```