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

ОТЧЕТ

по лабораторной работе № 2

по дисциплине «Алгоритмы и структуры данных»

Вариант №1

Студент гр. 8301 _____ Бобров А.Б.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2020

Цель работы

Реализовать кодирование и декодирование по алгоритму Хаффмана входной строки, вводимой через консоль.

Описание реализуемого класса и методов

Классы:

map – класс, реализованный в Л.Р. №1.

Tree – класс, реализованный в Л.Р. №1, с помощью которого реализуется красно-черное дерево.

Node – данный класс, реализованный в Л.Р. №1, представляет собой элемента дерева.

HuffmanNode – класс, используемый для реализации бин. дерева кодирования.

Методы:

void ListSorting(List<HuffmanNode*> &list) – производит сортировку по числу вхождения символа для списка узлов дерева.

void HuffmanTree(List<HuffmanNode*>& tree) – используется для построения бин. дерева кодировки

void HuffmanMap(HuffmanNode* root, map<char, bool*>& table, List<bool>& listCode) – строит map, в ключах которого находятся символы строки, а в значениях лежат их коды.

void PrintTable(map<char, bool*>::Tree::Node *root) – производит вывод таблицы кодировок.

Оценка временной сложности каждого метода

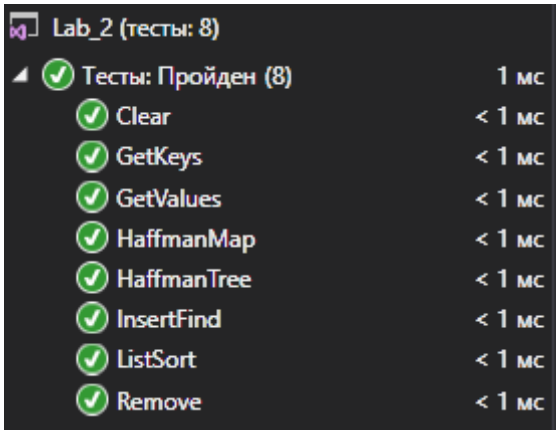
void ListSorting(List<HuffmanNode*> &list) – $\mathbf{O(n^2)}$

void HuffmanTree(List<HuffmanNode*>& tree) – $\mathbf{O(n^3)}$

void HuffmanMap(HuffmanNode* root, map<char, bool*>& table, List<bool>& listCode) – $\mathbf{O(n^2)}$

void PrintTable(map<char, bool*>::Tree::Node *root) – $\mathbf{O(n^2)}$

Описание реализованных Unit-тестов



Описание методов из Л.Р. №1:

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.

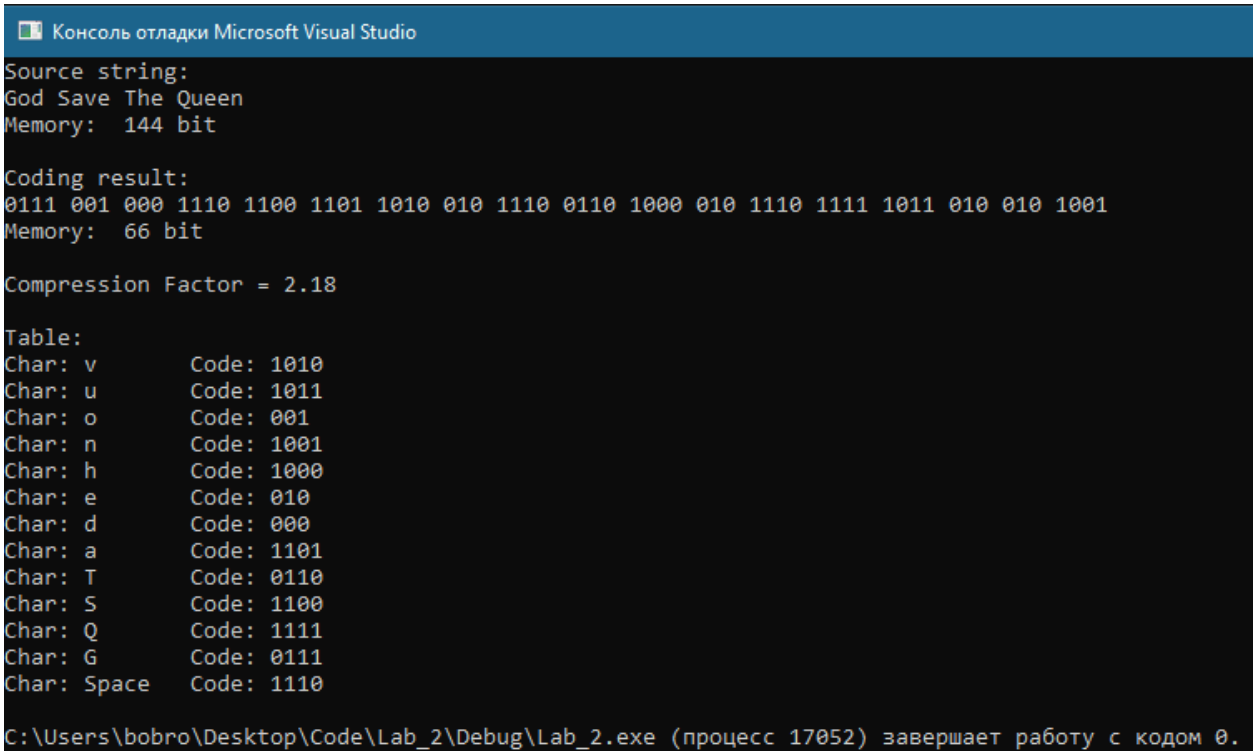
Описание новых методов:

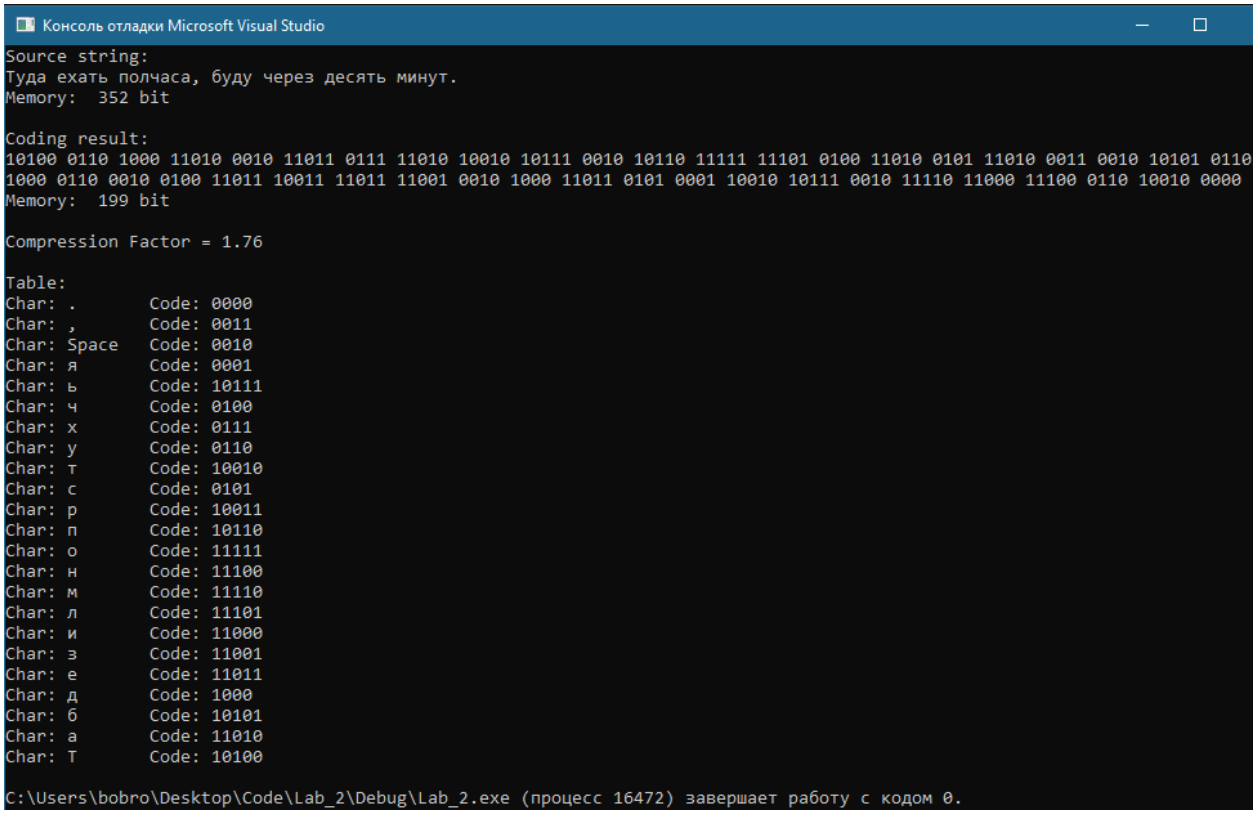
HuffmanMap – производит тест кодирования по Хаффману для каждого символа.

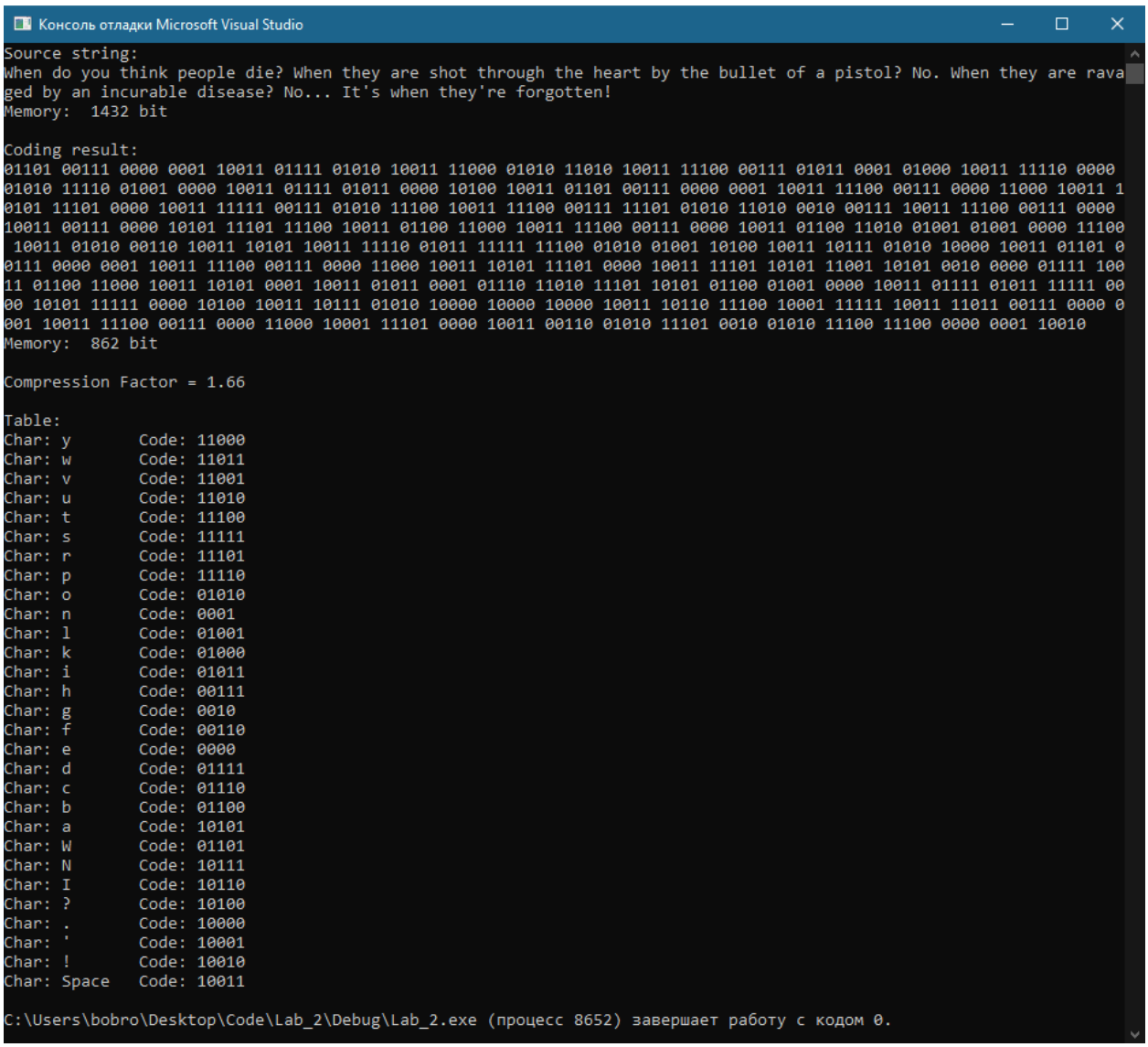
HuffmanTree –проверяет правильность построения бин. дерева поиска, проверяя каждый элемент.

ListSort – тестирует функцию сортировки списка узлов дерева по числу вхождения символа в строку.

Пример работы программы







Листинг

main.cpp:

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

class HuffmanNode
{
	public:
		int sum;
		char symbol;
		HuffmanNode* left, * right;
		HuffmanNode()
		{
			left = right = nullptr;
		}
		HuffmanNode(HuffmanNode* L, HuffmanNode* R)
		{
			left = L;
			right = R;
			sum = L->sum + R->sum;
		}
};
```

```
void ListSorting(List<HuffmanNode*> &list)
{
	List<HuffmanNode*>::Node *left = list.start;
	List<HuffmanNode*>::Node *right = list.start->next;		//element that will be next after
head element
	List<HuffmanNode*>::Node *tempo = new List<HuffmanNode*>::Node;	//node for saving of temporary
info
	while (left->next)	//bypass except far right
	{
		while (right)	//bypass of all, relative to the left for this moment
		{
			if ((left->info->sum) >= (right->info->sum))	//is reinstall required?
			{
				swap(left->info, right->info);
			}
			right = right->next;	//to avoid looping
		}
		left = left->next;
		right = left->next;
	}
}

void HuffmanTree(List<HuffmanNode*>& tree)
{
	while (tree.get_size() != 1)
	{
		ListSorting(tree);

		HuffmanNode* SonLeft = tree.start->info;
		tree.pop_front();

		HuffmanNode* SonRight = tree.start->info;
		tree.pop_front();

		HuffmanNode* parent = new HuffmanNode(SonLeft, SonRight);
		tree.push_back(parent);
	}
}

void HuffmanMap(HuffmanNode* root, map<char, bool*>& table, List<bool>& listCode)
{
	if (root->left != nullptr)
	{
		listCode.push_back(false);
		HuffmanMap(root->left, table, listCode);
	}

	if (root->right != nullptr)
	{
		listCode.push_back(true);
		HuffmanMap(root->right, table, listCode);
	}

	if (root->left == nullptr && root->right == nullptr)
	{
		table.find(root->symbol)->info.second = new bool[listCode.get_size()];
		for (int i = 0; i < listCode.get_size(); i++)
		{
			table.find(root->symbol)->info.second[i] = listCode.get_pointer(i)->info;
		}
	}

	if (listCode.get_size() == 0)
		return;

	listCode.pop_back();
}



void PrintTable(map<char, bool*>::Tree::Node *root)
{
	int arrSize;
	if (root->info.first != 0)
	{
		PrintTable(root->right);
```

```
                if (root->info.first == ' ')
                        cout << "Char: Space" << "   Code: ";
                else
                        cout <<"Char: "<< root->info.first << "       Code: ";

                arrSize = _msize(root->info.second) / sizeof(root->info.second[0]);

                for (int i = 0; i < arrSize; i++)
                {
                        cout << root->info.second[i];
                }

                cout << endl;
                PrintTable(root->left);
        }
}

void Huffman(string str)
{
        map<char, size_t> card;
        map<char, bool*> table;
        int Memory = 0;

        system("cls");
        cout << "Source string:\n" << str << endl;
        for (size_t i = 0; i < str.length(); ++i)
        {
                if (str[i] != 0)
                {
                        card.insert(str[i], i);
                        table.insert(str[i], nullptr);
                        ++Memory;
                }
        }

        List<char> list = card.get_keys();
        List<HuffmanNode*> tree;

        while (list.get_size() != 0)
        {
                HuffmanNode* element = new HuffmanNode();
                element->sum = list.start->sum;
                element->symbol = list.start->info;
                tree.push_back(element);
                list.pop_front();
        }
        cout << "Memory:  " << Memory * 8 << " bit\n" << endl;

        float Compression = Memory * 8;
        Memory = 0;
        int arrSize;
        char ch = 'a';
        int i = 0;
        ch = str[i];

        HuffmanTree(tree);
        HuffmanNode* root = tree.start->info;
        List<bool> listCode;

        HuffmanMap(root, table, listCode);
        cout << "Coding result: " << endl;

        while (ch != 0)
        {
                i++;
                arrSize = _msize(table.find(ch)->info.second) / sizeof(table.find(ch)->info.second[0]);
                Memory = Memory + arrSize;
                for (int i = 0; i < arrSize; i++)
                        cout << table.find(ch)->info.second[i];
                cout << " ";
                ch = str[i];
        }
        cout << endl;
        cout << "Memory:  " << Memory << " bit" << endl;

        Compression = Compression / Memory;
        cout << "\nCompression Factor = " << floor(Compression * 100) / 100 << endl;
```

```
        cout << "\nTable:" << endl;
        PrintTable(table.tree->root);
}

int main()
{
        string string;
        getline(cin, string);
        Huffman(string);
}
```

map.h:

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


////////////////////////////LIST////////////////////////////
template <typename T>
class List
{
public:
        class Node
        {
        public:
                Node* next = nullptr;
                T info;
                size_t sum;
        };
        Node* end = nullptr;
        Node* current = nullptr;
        Node* start = nullptr;

        void push_back(T element, size_t sum = 0)
        {
                if (!end)
                {
                        end = start = current = new Node;
                        end->info = element;
                        end->sum = sum;
                }
                else
                {
                        end->next = new Node;
                        end = end->next;
                        end->info = element;
                        end->sum = sum;
                }
        }

        void pop_front()
        {
                if (start != end)
                {
                        Node* temp = start;
                        start = start->next;
                        delete temp;
                }
                else if (get_size() == 1)
                {
                        Node* temp = start;
                        start = end = nullptr;
                        delete temp;
                }
                else
                        throw out_of_range("The list is empty");
        }

        void pop_back()
        {
                if (start != end)
                {
                        List<T>::Node* temp = start;
```

```
                        while (temp->next != end)
                        {
                                temp = temp->next;
                        }
                        delete temp->next;
                        end = temp;
                        end->next = nullptr;
                }
                else if (get_size() == 1)
                {
                        List<T>::Node* temp = end;
                        end = start = NULL;
                        delete temp;
                }
                else
                        throw out_of_range("List is empty");
        }

        List<T>::Node* get_pointer(size_t index)
        {
                if (get_size() == 0 || (index > get_size() - 1))
                {
                        throw out_of_range("Invalid argument");
                }
                else if (index == get_size() - 1)
                        return end;
                else if (index == 0)
                        return start;
                else
                {
                        List<T>::Node *temp = start;
                        while ((temp) && (index--))
                        {
                                temp = temp->next;
                        }
                        return temp;
                }
        }

        size_t get_size()
        {
                Node* temp = start;
                size_t length = 0;
                while (temp)
                {
                        length++;
                        temp = temp->next;
                }
                return length;
        }

        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
{
public:
        class Tree;
        Tree* tree;

        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
{
        public:
                class Node
                {
                public:
                        Node* right;
                        Node* left;
                        Node* parent = nullptr;
                        pair <TKey, TValue> info;
                        nodeColor color = BLACK;
                        size_t sum = 1;
                };
                void InsFix(Node*);

                void DelFix(Node*);

                void Rotate_L(Node*);

                void Rotate_R(Node*);

                void clear(Node *);

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

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

                void print(Node*, string);

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

//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 || node == NN) return;
        if (node->left) get_keys(node->left, list);
        if (node->right) get_keys(node->right, list);
```

```
        list.push_back(node->info.first, node->sum);
}

//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.push_back(node->info.second, node->sum);
}

//clear
template <typename TKey, typename TValue>
void map<TKey, TValue>::Tree::clear(typename Tree::Node* node)
{
        if (node->left != NN) clear(node->left);
        if (node->right != NN) clear(node->right);
        if (node == root) root = NN;
        delete node;
}

//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));
        }
}
```

```
//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;
}

//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;
}

//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;
}

//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 <stdexcept>
#include "../Lab_2/map.h"
#include "../Lab_2/main.cpp"
using namespace Microsoft::VisualStudio::CppUnitTestFramework;
using namespace std;

namespace Lab2UnitTest
{
        TEST_CLASS(Lab2UnitTest)
        {
        public:
                TEST_METHOD(InsertFind)
                {
                        map<int, int> card;
                        bool before = card.find(5);
                        card.insert(5, 1);
                        bool after = card.find(5);
                        Assert::AreEqual(!before, after);
                }
                TEST_METHOD(Remove)
                {
                        map<int, int> card;
                        card.insert(5, 1);
                        bool before = card.find(5);
                        card.remove(5);
                        bool after = card.find(5);
                        Assert::AreEqual(before, !after);
                }
                TEST_METHOD(Clear)
                {
                        map<int, int> card;
                        card.insert(5, 1);
                        card.insert(6, 2);
                        card.clear();
                        bool findTwoElements;
                        if (card.find(5) == nullptr && card.find(6) == nullptr) findTwoElements = false;
                        Assert::IsFalse(findTwoElements);
                }
                TEST_METHOD(GetKeys)
                {
                        map<int, int> card;
                        card.insert(5, 1);
                        card.insert(6, 2);
                        card.insert(7, 3);
                        List<int> list = card.get_keys();
                        int sum_of_keys = 0;
                        while (list.isCurrent())
                                sum_of_keys += list.next();
                        Assert::IsTrue(sum_of_keys == 18);
                }
                TEST_METHOD(GetValues)
                {
                        map<int, int> card;
                        card.insert(5, 1);
                        card.insert(6, 2);
                        card.insert(7, 3);
                        List<int> list = card.get_values();
                        int sum_of_values = 0;
                        while (list.isCurrent())
                                sum_of_values += list.next();
                        Assert::IsTrue(sum_of_values == 6);
```

```
		}
		TEST_METHOD(ListSort)
		{
			List<HuffmanNode*> list;
			HuffmanNode* sum = new HuffmanNode;
			sum->symbol = 'a';
			sum->sum = 19;
			HuffmanNode* b = new HuffmanNode;
			b->symbol = 'b';
			b->sum = 7;
			list.push_back(b);
			list.push_back(sum);
			ListSorting(list);
			Assert::AreEqual(list.get_pointer(0)->info->symbol, 'b');
			Assert::AreEqual(list.get_pointer(1)->info->symbol, 'a');
		}
		TEST_METHOD(HaffmanTree)
		{
			List<HuffmanNode*> list;
			HuffmanNode* sum = new HuffmanNode;
			sum->symbol = 'a';
			sum->sum = 19;
			HuffmanNode* b = new HuffmanNode;
			b->symbol = 'b';
			b->sum = 7;
			HuffmanNode* symbol = new HuffmanNode;
			symbol->symbol = 'c';
			symbol->sum = 24;
			list.push_back(symbol);
			list.push_back(b);
			list.push_back(sum);
			HuffmanTree(list);
			Assert::AreEqual(list.start->info->left->symbol, 'c');
			Assert::AreEqual(list.start->info->right->left->symbol, 'b');
			Assert::AreEqual(list.start->info->right->right->symbol, 'a');
		}
		TEST_METHOD(HaffmanMap)
		{
			List<HuffmanNode*> list;
			HuffmanNode* sum = new HuffmanNode;
			sum->symbol = 'a';
			sum->sum = 19;
			HuffmanNode* b = new HuffmanNode;
			b->symbol = 'b';
			b->sum = 7;
			HuffmanNode* symbol = new HuffmanNode;
			symbol->symbol = 'c';
			symbol->sum = 24;
			list.push_back(symbol);
			list.push_back(b);
			list.push_back(sum);
			HuffmanTree(list);
			HuffmanNode* root = list.start->info;
			List<bool> listCode;
			map<char, bool*> table;
			table.insert('a', nullptr);
			table.insert('b', nullptr);
			table.insert('c', nullptr);
			HuffmanMap(root, table, listCode);
			Assert::AreEqual(table.find('c')->info.second[0], false);
			Assert::AreEqual(table.find('b')->info.second[0], true);
			Assert::AreEqual(table.find('b')->info.second[1], false);
			Assert::AreEqual(table.find('a')->info.second[0], true);
			Assert::AreEqual(table.find('a')->info.second[1], true);
		}
	};
}
```