**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

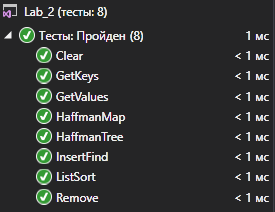
void ListSorting(List<HuffmanNode\*> &list) – **O()**

void HuffmanTree(List<HuffmanNode\*>& tree) – **O()**

void HuffmanMap(HuffmanNode\* root, map<char, bool\*>& table, List<bool>& listCode) – **O()**

void PrintTable(map<char, bool\*>::Tree::Node \*root) – **O()**

### Описание реализованных Unit-тестов



**Описание методов из Л.Р. №1:**

Сlear – 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.

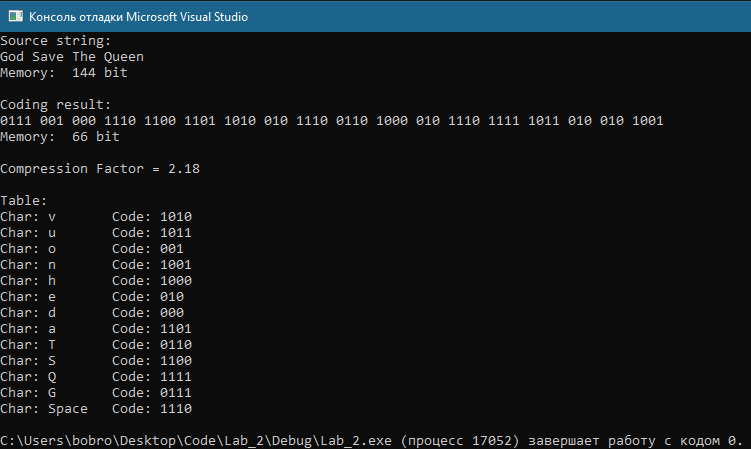
**Описание новых методов:**

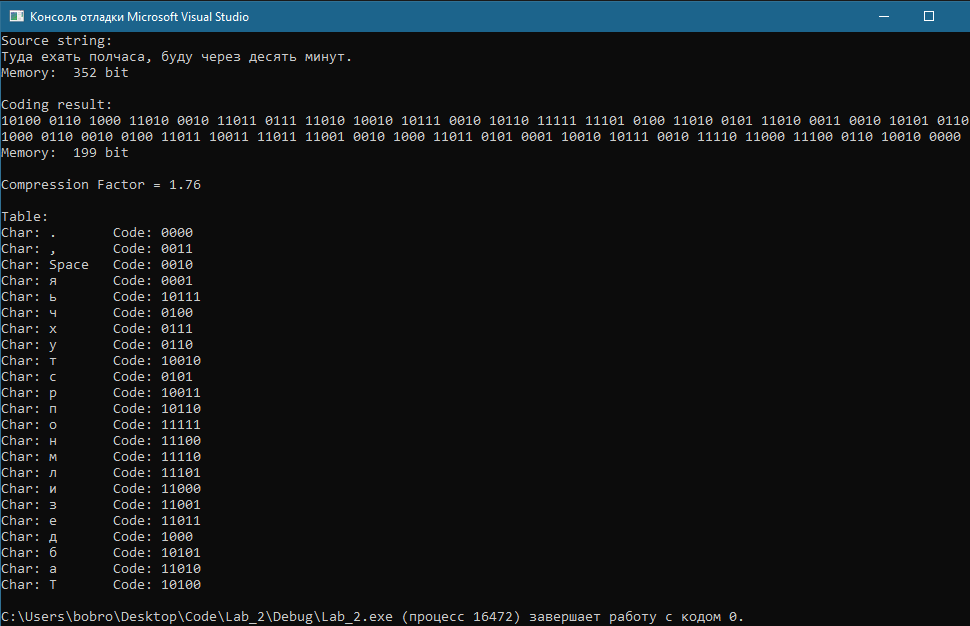
HuffmanMap – производит тест кодирования по Хаффману для каждого символа.

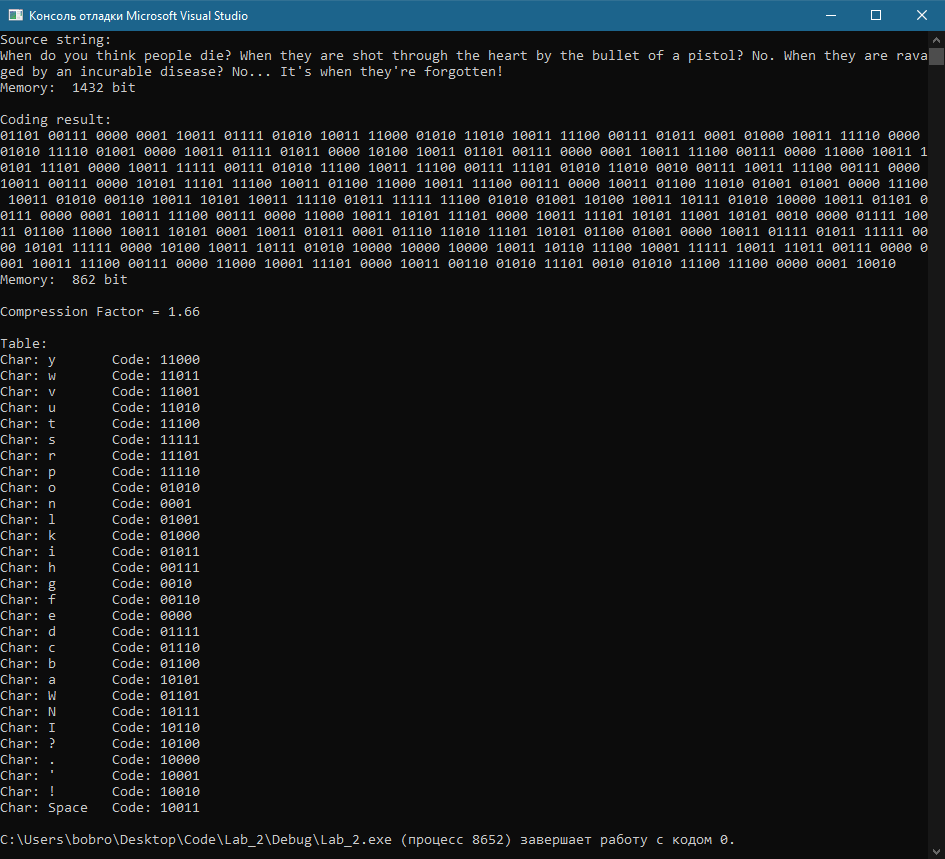
HuffmanTree –проверяет правильность построения бин. дерева поиска, проверяя каждый элемент.

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);

}

};

}