**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»**

Тема: «Алгоритмы кодирования»

Вариант 1

|  |  |  |
| --- | --- | --- |
| Студент гр. 8301 |  | Онвука П.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Цель работы

Реализовать кодирование и декодирование по алгоритму Хаффмана входной строки, вводимой через консоль. Посчитать объем памяти, который занимает исходная и закодированная строки. Вывести на экран таблицу частот и кодов, результат кодирования и декодирования, коэффициент сжатия.

## Описание программы

Алгоритм Хаффмана — жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью.

Этот метод кодирования состоит из двух основных этапов:

1. Построение оптимального кодового дерева.
2. Построение отображения код-символ на основе построенного дерева.

Идея алгоритма состоит в следующем: зная вероятности появления символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством [префиксности](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%84%D0%B8%D0%BA%D1%81%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана.

1. Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
2. Выбираются два свободных узла дерева с наименьшими весами.
3. Создается их родитель с весом, равным их суммарному весу.
4. Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0. Битовые значения ветвей, исходящих от корня, не зависят от весов потомков.
6. Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

В данной программе используются следующие структуры данных:

HuffmanTree: Для хранения таблицы кодирования и декодирования которые строятся на основе готового дерева; указатель на корень дерева; очередь с приоритетами.

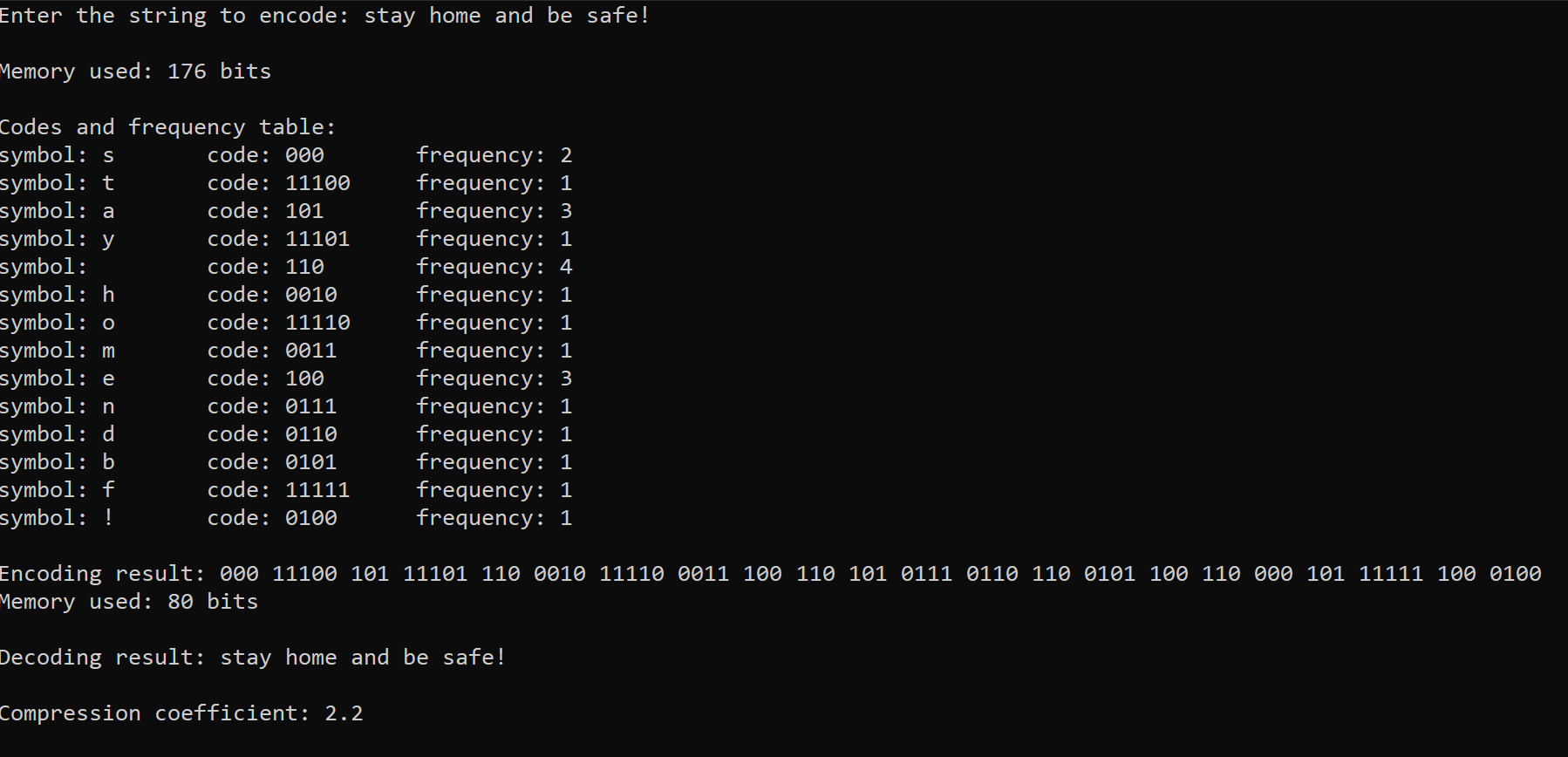
priority\_queue: Используется для хранения узлов для создания дерева Хаффмана.

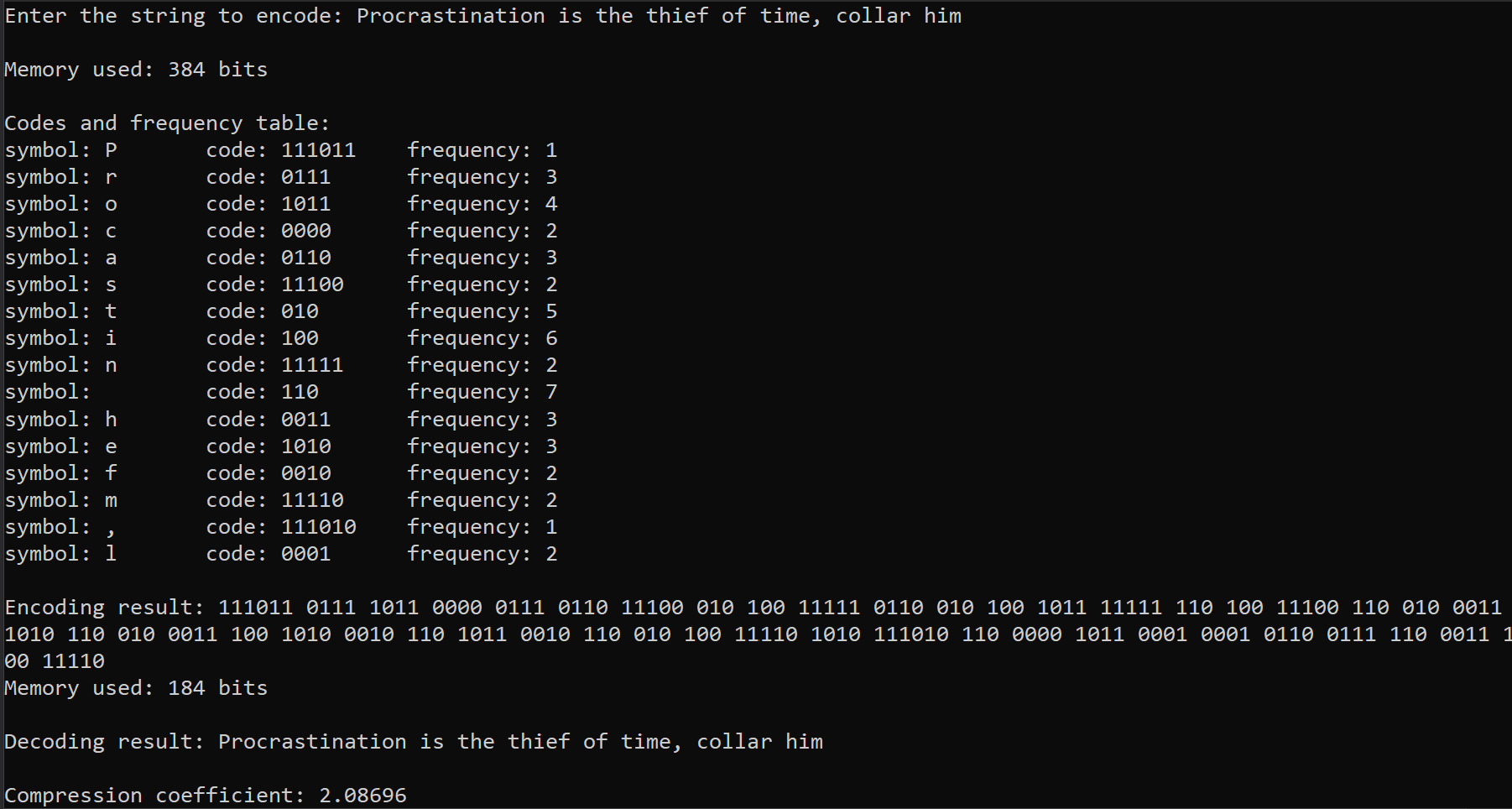
Map: Для хранения символов строки и их частот, а также для хранения таблиц кодирования и декодирования.

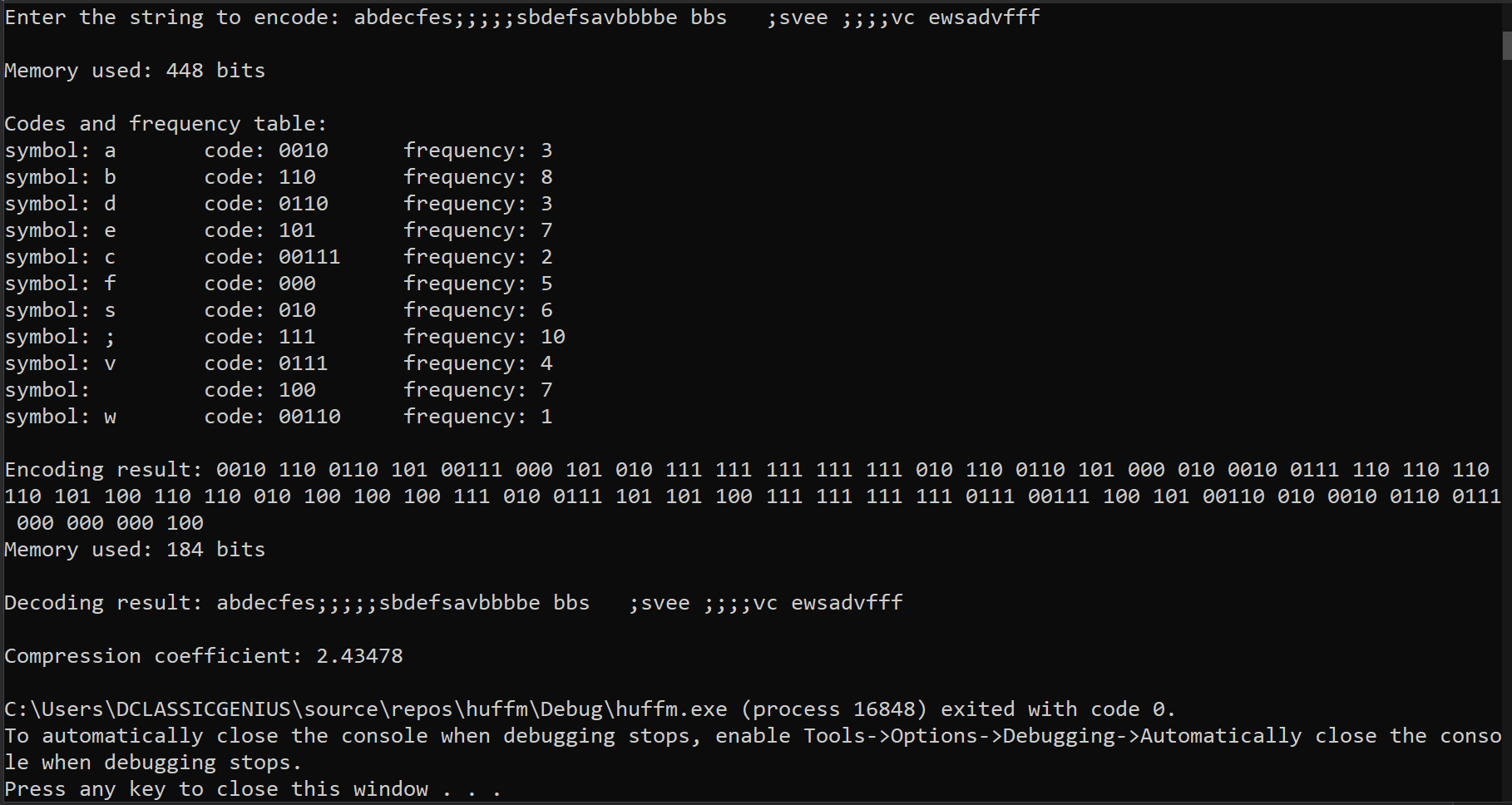
## Оценка временной сложности методов

1. **Encode** N\*log²(M) + M\*log(M)
2. **Decode** N\*log(M)
3. **trim** О(N);
4. **CompCoeff** О(1);
5. **Print** N\*log²(M) + M\*log(M)

## Примеры работы







## Листинг

#pragma once

#include <iostream>

#include <string>

#include "Map.h"

#include "priority\_queue.h"

#include "Queue.h"

using namespace std;

class HuffmanTree

{

private:

class Node

{

public:

char symbol;

int val;

Node\* right;

Node\* left;

Node(char symbol, int val, Node\* left, Node\* right)

{

this->symbol = symbol;

this->val = val;

this->right = right;

this->left = left;

}

~Node()

{

this->right = nullptr;

this->left = nullptr;

}

};

Node\* root;

priority\_queue<Node\*> nodes\_list;

void clear(Node\* node)

{

if (node != nullptr)

{

clear(node->left);

clear(node->right);

delete node;

}

}

//creates code and decode tables

void Fill(Map<char, string>& code\_table, Map<string, char>& decode\_table, Node\* direction, string& code\_str)

{

if (direction != nullptr)

{

//if node is leaf

if (direction->left == nullptr && direction->right == nullptr)

{

code\_table.insert(direction->symbol, code\_str);

decode\_table.insert(code\_str, direction->symbol);

}

code\_str.push\_back('0');

Fill(code\_table, decode\_table, direction->left, code\_str);

code\_str.push\_back('1');

Fill(code\_table, decode\_table, direction->right, code\_str);

}

if (!code\_str.empty())

code\_str.pop\_back();

}

public:

Map<char, string> code\_table;

Map<string, char> decode\_table;

HuffmanTree()

{

root = nullptr;

}

~HuffmanTree()

{

clear(root);

root = nullptr;

}

//create Huffman tree from map

void create(Map<char, int>& arr)

{

if (arr.getSize() == 0)

throw logic\_error("empty map");

List<char> symbols;

arr.get\_keys(symbols);

//filling priority queue with nodes

while (!symbols.isEmpty())

{

Node\* node = new Node(symbols.at(0), arr.find(symbols.at(0)), nullptr, nullptr);

symbols.pop\_front();

nodes\_list.insert(node, node->val);

}

//connects nodes

while (nodes\_list.getSize() > 1)

{

Node\* left = nodes\_list.extract\_min();

Node\* right = nodes\_list.extract\_min();

Node\* parent = new Node(0, left->val + right->val, left, right);

nodes\_list.insert(parent, parent->val);

}

root = nodes\_list.extract\_min();

string code\_str;

Fill(code\_table, decode\_table, root, code\_str);

}

};

//get Huffman code for the string

inline string Encode(const string& enc, HuffmanTree& encod)

{

Map<char, int> frequency;

//counts the occurrence of symbols

for (auto symbol : enc)

{

if (frequency.contains(symbol))

frequency.update(symbol, frequency.find(symbol) + 1);

else

frequency.insert(symbol, 1);

}

encod.create(frequency);

//encoding string

string encode;

for (auto symbol : enc)

{

encode += encod.code\_table.find(symbol);

encode += " ";

}

encode.pop\_back();

return encode;

}

//decodes the string

inline string Decode(const string& encoded\_str, HuffmanTree& h\_tree)

{

if (h\_tree.decode\_table.getSize() == 0)

throw logic\_error("empty table to decode");

//filling queue with the codes

Queue<string> codes;

string code;

for (auto symbol : encoded\_str)

{

if (symbol == '1' || symbol == '0')

code += symbol;

else if (symbol == ' ')

{

codes.enqueue(code);

code.clear();

}

else

throw invalid\_argument("invalid code");

}

codes.enqueue(code);

code.clear();

//decodes code string with decode table created from Huffman's tree

string decode;

while (codes.getSize() > 0)

{

code = codes.dequeue();

if (!h\_tree.decode\_table.contains(code))

throw invalid\_argument("wrong attached huff. tree");

decode += h\_tree.decode\_table.find(code);

}

return decode;

}

//deletes all spaces in the string

inline string trim(string str)

{

string trimmed;

for (auto element : str)

{

if (element != ' ')

trimmed += element;

}

return trimmed;

}

//calculating Compression Coefficient

inline float CompCoeff(string& original, string& coded)

{

if (trim(coded).empty())

throw logic\_error("Coded string cannot be empty");

return (float)(original.length() \* 8) / trim(coded).length();

}

inline void Print(HuffmanTree& h\_tree, const string& encod)

{

Map<char, int> frequency;

string symbols;

//calculates symbols occurrence

for (auto symbol : encod)

{

if (frequency.contains(symbol))

frequency.update(symbol, frequency.find(symbol) + 1);

else

{

symbols += symbol;

frequency.insert(symbol, 1);

}

}

for (auto element : symbols)

{

cout << "symbol: " << element << "\tcode: " << h\_tree.code\_table.find(element) << "\tfrequency: " << frequency.find(element) << endl;

}

}

Main.cpp

#include "huffman.h"

using namespace std;

int main()

{

string str;

cout << "Enter the string to encode: ";

getline(cin, str);

HuffmanTree encoding;

auto code = Encode(str, encoding);

auto decode = Decode(code, encoding);

cout << "\nMemory used: " << str.length() \* 8 << " bits" << endl;

cout << "\nCodes and frequency table:" << endl;

Print(encoding, str);

cout << "\nEncoding result: " << code << endl;

cout << "Memory used: " << trim(code).length() << " bits" << endl;

cout << "\nDecoding result: " << decode << endl;

cout << "\nCompression coefficient: " << CompCoeff(str, code) << endl;

return 0;

}

TEST.cpp

#include "pch.h"

#include "CppUnitTest.h"

#include "../huffm/huffman.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace huffmantest

{

TEST\_CLASS(huffmantest)

{

public:

Assert assert;

TEST\_METHOD(create\_empty)

{

HuffmanTree enc;

Map<char, int> name;

try

{

enc.create(name);

}

catch (logic\_error& error)

{

assert.AreEqual("empty map", error.what());

}

}

TEST\_METHOD(encode)

{

HuffmanTree h\_tree;

string str = "mississippi river";

string result = "1011 11 00 00 11 00 00 11 010 010 11 1000 011 11 1001 1010 011";

assert.AreEqual(result, Encode(str, h\_tree));

}

TEST\_METHOD(decode)

{

HuffmanTree h\_tree;

string str = "what a day!";

string code = Encode(str, h\_tree);

assert.AreEqual(str, Decode(code, h\_tree));

}

TEST\_METHOD(decode\_empty)

{

HuffmanTree h\_tree;

string code = "10 100 111";

try

{

string decode = Decode(code, h\_tree);

}

catch (logic\_error& error)

{

assert.AreEqual("empty table to decode", error.what());

}

}

TEST\_METHOD(trim\_)

{

string str = "ha p p y days";

string result = "happydays";

assert.AreEqual(result, trim(str));

}

TEST\_METHOD(compCoeff\_empty)

{

string str = "rainy days";

string code;

try

{

float a = CompCoeff(str, code);

}

catch (logic\_error& error)

{

assert.AreEqual("Coded string cannot be empty", error.what());

}

}

};

}