**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 |  | Забалуев Д.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

Идея алгоритма основана на частоте появления символа в последовательности. Символ, который встречается в последовательности чаще всего, получает маленький код, а символ, который встречается реже всего, получает длинный код.

Прежде всего, высчитывается количество вхождений каждого символа в кодируемую строку. Затем эти символы становятся «листьями» бинарного дерева с весом равным количеству вхождений. Бинарное дерево строится от листьев к корню по следующему принципу: создаются узлы дерева, хранящие сам символ и его вес(приоритет) – количество вхождений символа в строку, затем строится очередь с приоритетами, где в качестве приоритета выступает вес узла; далее из очереди берутся два узла с наименьшим приоритетом и создается новый узел, для которого эти два элемента будут потомками; новый узел получает вес равный сумме весов своих потомков и добавляется обратно в очередь. Данный процесс идёт до тех пор, пока в очереди не останется один узел – корень бинарного дерева.

Чтобы получить код для каждого символа, надо просто пройтись по дереву до каждого листа, и для каждого перехода добавлять 0, если идём влево, и 1 — если направо.

Чтобы расшифровать закодированную строку, необходимо, просто идти по дереву, сворачивая в соответствующую каждому биту сторону до тех пор, пока мы не будет достигнут лист.

В данной программе используются следующие структуры данных:

* Класс HuffmanTree:

Данный класс реализует дерево Хаффмана. Дерево состоит из узлов класса Node, которые хранят символ и его частоту, а также указатели на потомков. В этом классе хранятся: таблицы кодирования и декодирования (контейнер Map), которые строятся на основе готового дерева; указатель на корень дерева; очередь с приоритетами, хранящая узлы, по которым строится дерево.

Данный класс имеет один публичный метод – createFrom(Map<char, int>& arr), который строит дерево из ассоциативного массива, хранящего символы и их частоты.

Данный класс имеет два приватных метода – putTo(…) и clear(). putTo осуществляет построение таблиц кодирования и декодирования по готовому дереву; clear вызывается в деструкторе и удаляет всё дерево.

* Класс Map:

Обычный шаблонный ассоциативный массив. В данной программе используется для возможности одновременного хранения символов строки и их частот, а также для хранения таблиц кодирования и декодирования.

* Класс List:

Обычный шаблонный двусвязный список. Используется вместе с контейнером Map для хранения всех ключей ассоциативного массива.

* Класс Queue:

Обычная шаблонная очередь. Используется при декодировании для хранения кода каждого символа строки.

* Класс priority\_queue:

Обычная шаблонная очередь с приоритетами на основе двоичной minHeap. Используется для хранения узлов для создания дерева Хаффмана

## Оценка временной сложности методов

*N – количество символов входной строки*

*M – количество символов в алфавите*

1. ***createFrom*** имеет временную сложность О(*M\*log(M)*)

*M+M\*(1+log(M))+M\*(2\*2log(M)+log(M))+2log(M)+M\*log(M) = M\*log(M)*

1. ***putTo*** имеет временную сложность О(*M\*log(M)*);

*2M\*log(M) = M\*log(M)*

1. ***EncodeToHuffmanCode*** имеет временную сложность О(*N\*log²(M) + M\*log(M)*);

*N\*log²(M) + M\*log(M) + N\*log(M) = N\*log²(M) + M\*log(M)*

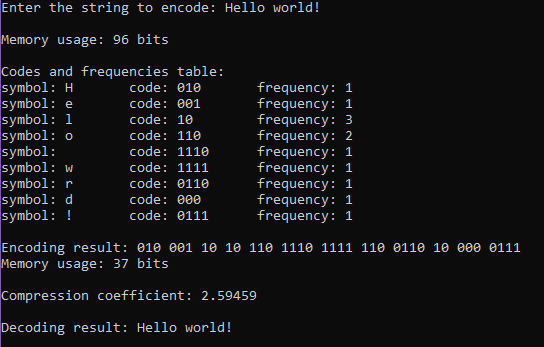
1. ***DecodeHuffmanCode*** имеет временную сложность О(*N\*log(M)*);

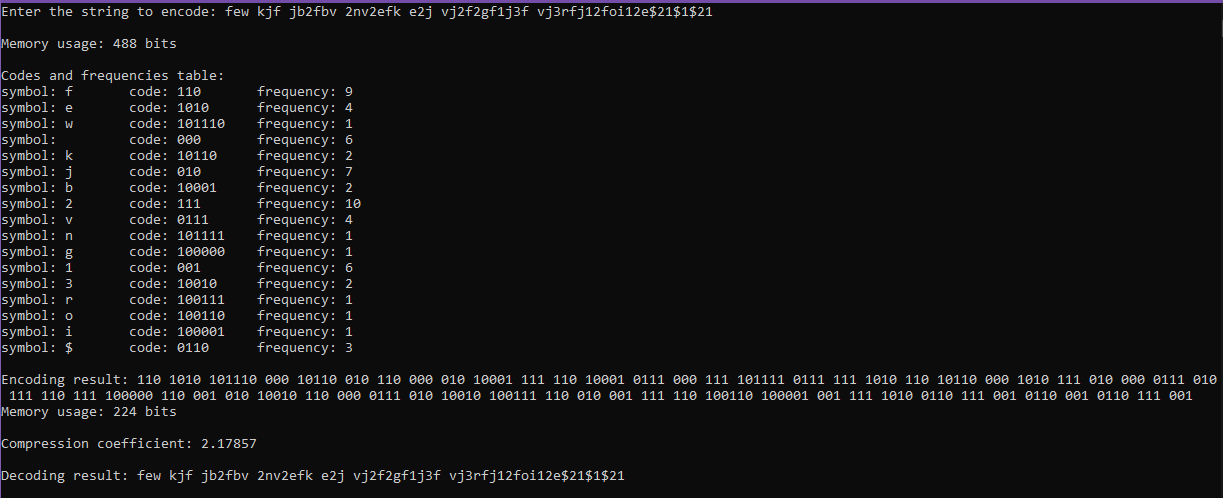
*N+N\*(1+log(M) + log(M)) = N\*log(M)*

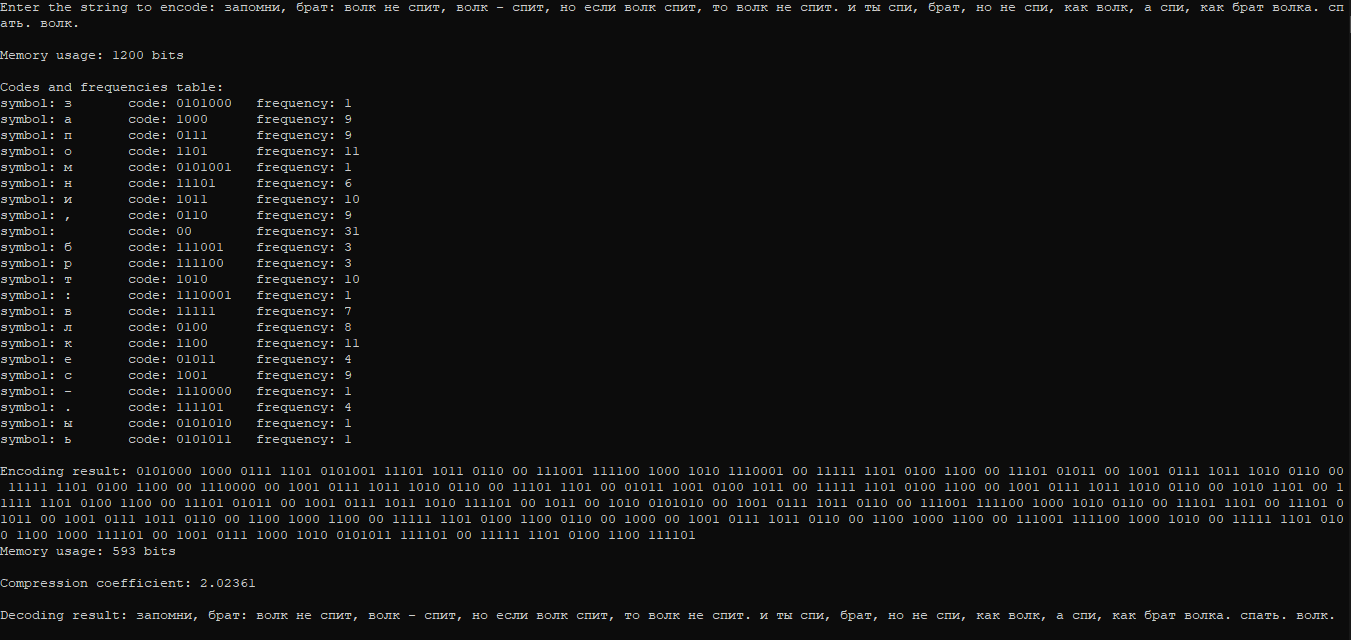
1. ***trim*** имеет временную сложность О(*N*);
2. ***CompressionCoefficient*** имеет временную сложность О(*1*);
3. ***PrintFrequencyAndCodeTable*** имеет временную сложность О(*N\*log²(M) + M\*log(M)*);

*N\*log²(M) + M\*(log(M) + log(M)) = N\*log²(M) + M\*log(M)*

## Примеры работы







## Листинг

main.cpp

#include "HaffmanAlgoritm.h"

using std::cin;

using std::cout;

using std::endl;

int main()

{

std::string sample;

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

std::getline(cin, sample);

HuffmanTree encoding;

auto code = EncodeToHuffmanCode(sample, encoding);

auto decode = DecodeHuffmanCode(code, encoding);

cout << "\nMemory usage: " << sample.length() \* 8 << " bits" << endl;

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

PrintFrequencyAndCodeTable(encoding, sample);

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

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

cout << "\nCompression coefficient: " << CompressionCoefficient(sample, code) << endl;

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

return 0;

}

HuffmanAlgorithm.h

#pragma once

#include <iostream>

#include <string>

#include "Map.h"

#include "priority\_queue.h"

#include "Queue.h"

class HuffmanTree

{

private:

class Node

{

public:

char symbol;

int amount;

Node\* right;

Node\* left;

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

{

this->symbol = symbol;

this->amount = amount;

this->right = right;

this->left = left;

}

~Node()

{

this->right = nullptr;

this->left = nullptr;

}

};

Node\* root;

priority\_queue<Node\*> nodes\_list; //stores huffman's tree nodes while constructing it

void clear(Node\* node)

{

if (node != nullptr)

{

clear(node->left);

clear(node->right);

delete node;

}

}

//creates code and decode tables from the tree

void putTo(Map<char, std::string>& code\_table, Map<std::string, char>& decode\_table, Node\* direction, std::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);

}

//goes left - adds 1 to the code

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

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

//goes right - adds 0 to the code

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

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

}

if(!code\_str.empty())

code\_str.pop\_back();

}

public:

Map<char, std::string> code\_table;

Map<std::string, char> decode\_table;

HuffmanTree()

{

root = nullptr;

}

~HuffmanTree()

{

clear(root);

root = nullptr;

}

//creates Huffman tree basing on Map of symbol:frequency pairs

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

{

if (arr.getSize() == 0)

throw std::logic\_error("Map to create huffman code from was empty");

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->amount);

}

//connecting nodes

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

{

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

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

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

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

}

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

std::string code\_str;

putTo(code\_table, decode\_table, root, code\_str); //create code and decode tables

}

};

//get Huffman code for the string

inline std::string EncodeToHuffmanCode(const std::string& encodable, HuffmanTree& encoding)

{

Map<char, int> frequencies;

//counts symbols repeats amount

for (auto symbol : encodable)

{

if (frequencies.contains(symbol))

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

else

frequencies.insert(symbol, 1);

}

encoding.createFrom(frequencies); //creates Huffman tree

//encoding string with created code table

std::string encoded;

for (auto symbol : encodable)

{

encoded += encoding.code\_table.find(symbol);

encoded += " ";

}

encoded.pop\_back();

return encoded;

}

//decodes the code string using Huffman's tree for this string

inline std::string DecodeHuffmanCode(const std::string& encoded\_str, HuffmanTree& h\_tree)

{

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

throw std::logic\_error("Decode table is empty");

//filling queue with each code separated by space

Queue<std::string> symbols\_codes;

std::string code;

for (auto symbol : encoded\_str)

{

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

code += symbol;

else if (symbol == ' ')

{

symbols\_codes.enqueue(code);

code.clear();

}

else

throw std::invalid\_argument("Wrong code string format!");

}

symbols\_codes.enqueue(code);

code.clear();

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

std::string decode;

while(symbols\_codes.getSize() > 0)

{

code = symbols\_codes.dequeue();

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

throw std::invalid\_argument("Attached huffman tree is wrong!");

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

}

return decode;

}

//deletes all spaces in the string

inline std::string trim(std::string str)

{

std::string trimmed;

for (auto element : str)

{

if (element != ' ')

trimmed += element;

}

return trimmed;

}

//calculates Compression Coefficient

inline float CompressionCoefficient(std::string& original, std::string& coded)

{

if (trim(coded).empty())

throw std::logic\_error("Coded string can not be empty");

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

}

inline void PrintFrequencyAndCodeTable(HuffmanTree& h\_tree, const std::string& encodable)

{

Map<char, int> frequencies;

std::string symbols;

//calculates symbols repeats amount

for (auto symbol : encodable)

{

if (frequencies.contains(symbol))

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

else

{

symbols += symbol;

frequencies.insert(symbol, 1);

}

}

for (auto element : symbols)

{

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

}

}

huffmantest.cpp

#include "pch.h"

#include "CppUnitTest.h"

#include "../LABA 2/HaffmanAlgoritm.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace huffmantest

{

TEST\_CLASS(huffmantest)

{

public:

TEST\_METHOD(createFrom\_exception\_test)

{

HuffmanTree encoding;

Map<char, int> dict;

try

{

encoding.createFrom(dict);

}

catch (std::logic\_error& err)

{

Assert::AreEqual("Map to create huffman code from was empty", err.what());

}

}

TEST\_METHOD(createFrom\_putTo\_test)

{

Map<char, int> dictionary;

std::string str = "beep bear";

for (auto symbol : str)

{

if (dictionary.contains(symbol))

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

else

dictionary.insert(symbol, 1);

}

HuffmanTree encoding;

encoding.createFrom(dictionary);

const std::string expected\_for\_B = "00";

const std::string expected\_for\_P = "010";

const std::string expected\_for\_R = "011";

Assert::AreEqual(expected\_for\_B, encoding.code\_table.find('b'));

Assert::AreEqual(expected\_for\_P, encoding.code\_table.find('p'));

Assert::AreEqual(expected\_for\_R, encoding.code\_table.find('r'));

}

TEST\_METHOD(encodeToHuffmanCode\_test)

{

HuffmanTree h\_tree;

std::string str = "it is test string";

std::string expected = "111 10 110 111 00 110 10 0100 00 10 110 00 10 0111 111 0101 0110";

Assert::AreEqual(expected, EncodeToHuffmanCode(str, h\_tree));

}

TEST\_METHOD(decodeHuffmanCode\_test)

{

HuffmanTree h\_tree;

std::string str = "it is test string";

std::string code = EncodeToHuffmanCode(str, h\_tree);

Assert::AreEqual(str, DecodeHuffmanCode(code, h\_tree));

}

TEST\_METHOD(decodeHuffmanCode\_exception\_test1)

{

HuffmanTree h\_tree;

std::string code = "11 00 101";

try

{

std::string decode = DecodeHuffmanCode(code, h\_tree);

}

catch (std::logic\_error& err)

{

Assert::AreEqual("Decode table is empty", err.what());

}

}

TEST\_METHOD(decodeHuffmanCode\_exception\_test2)

{

HuffmanTree h\_tree;

std::string str = "it is test string";

std::string code = EncodeToHuffmanCode(str, h\_tree);

try

{

std::string decode = DecodeHuffmanCode(str, h\_tree);

}

catch (std::invalid\_argument & err)

{

Assert::AreEqual("Wrong code string format!", err.what());

}

}

TEST\_METHOD(decodeHuffmanCode\_exception\_test3)

{

HuffmanTree h\_tree;

std::string str = "it is test string";

std::string code\_1 = EncodeToHuffmanCode(str, h\_tree);

std::string code\_2 = "11 00 100 110";

try

{

std::string decode = DecodeHuffmanCode(code\_2, h\_tree);

}

catch (std::invalid\_argument & err)

{

Assert::AreEqual("Attached huffman tree is wrong!", err.what());

}

}

TEST\_METHOD(trim\_test)

{

std::string str = "h e l l o w o r l d !";

std::string expected = "helloworld!";

Assert::AreEqual(expected, trim(str));

}

TEST\_METHOD(compressionCoefficient\_test)

{

std::string not\_code = "Have you ever heard the sounds of ants ?";

std::string code = "11 00000 101";

Assert::AreEqual((float)32, CompressionCoefficient(not\_code, code));

}

TEST\_METHOD(compressionCoefficient\_exception\_test)

{

std::string not\_code = "Have you ever heard the sounds of ants ?";

std::string code;

try

{

float a = CompressionCoefficient(not\_code, code);

}

catch (std::logic\_error& err)

{

Assert::AreEqual("Coded string can not be empty", err.what());

}

}

};

}