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

1. 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.

2. 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

3. Оценка временной сложности методов

N – количество символов входной строки

M – количество символов в алфавите

1) ***createFrom*** имеет временную сложность O(*M*log(M)*)
   M+M(1+log(M))+M*(2*2log(M)+log(M))+2log(M)+M*log(M) = M*log(M)*
2) ***putTo*** имеет временную сложность O(*M*log(M)*);
   *2M*log(M) = M*log(M)*
3) ***EncodeToHuffmanCode*** имеет временную сложность O(*N*log²(M) + M*log(M)*);
   *N*log²(M) + M*log(M) + N*log(M) = N*log²(M) + M*log(M)*
4) ***DecodeHuffmanCode*** имеет временную сложность O(*N*log(M)*);
   N+N(1+log(M) + log(M)) = N*log(M)*
5) ***trim*** имеет временную сложность O(*N*);
6) ***CompressionCoefficient*** имеет временную сложность O(*1*);
7) ***PrintFrequencyAndCodeTable*** имеет временную сложность O(*N*log²(M) + M*log(M)*);
   *N*log²(M) + M*(log(M) + log(M)) = N*log²(M) + M*log(M)*

4. Примеры работы

```
Enter the string to encode: Hello world!

Memory usage: 96 bits

Codes and frequencies table:
symbol: H       code: 010       frequency: 1
symbol: e       code: 001       frequency: 1
symbol: l       code: 10        frequency: 3
symbol: o       code: 110       frequency: 2
symbol:         code: 1110      frequency: 1
symbol: w       code: 1111      frequency: 1
symbol: r       code: 0110      frequency: 1
symbol: d       code: 000       frequency: 1
symbol: !       code: 0111      frequency: 1

Encoding result: 010 001 10 10 110 1110 1111 110 0110 10 000 0111
Memory usage: 37 bits

Compression coefficient: 2.59459

Decoding result: Hello world!
```

```
Enter the string to encode: few kjf jb2fbv 2nv2efk e2j vj2f2gf1j3f vj3rfj12foi12e$21$1$21

Memory usage: 488 bits

Codes and frequencies table:
symbol: f       code: 110       frequency: 9
symbol: e       code: 1010      frequency: 4
symbol: w       code: 101110    frequency: 1
symbol:         code: 000       frequency: 6
symbol: k       code: 10110     frequency: 2
symbol: j       code: 010       frequency: 7
symbol: b       code: 10001     frequency: 2
symbol: 2       code: 111       frequency: 10
symbol: v       code: 0111      frequency: 4
symbol: n       code: 101111    frequency: 1
symbol: g       code: 100000    frequency: 1
symbol: 1       code: 001       frequency: 6
symbol: 3       code: 10010     frequency: 2
symbol: r       code: 100111    frequency: 1
symbol: o       code: 100110    frequency: 1
symbol: i       code: 100001    frequency: 1
symbol: $       code: 0110      frequency: 3

Encoding result: 110 1010 101110 000 10110 010 110 000 010 10001 111 110 10001 0111 000 111 101111 0111 111 1010 110 10110 000 1010 111 010 000 0111 010
 111 110 111 100000 110 001 010 10010 110 000 0111 010 10010 100111 110 010 001 111 110 100110 100001 001 111 1010 0110 111 001 0110 001 0110 111 001
Memory usage: 224 bits

Compression coefficient: 2.17857

Decoding result: few kjf jb2fbv 2nv2efk e2j vj2f2gf1j3f vj3rfj12foi12e$21$1$21
```

```
Enter the string to encode: запомни, брат: волк не спит, волк - спит, но если волк спит, то волк не спит. и ты спи, брат, но не спи, как волк, а спи, как брат волка. сп
ать. волк.

Memory usage: 1200 bits

Codes and frequencies table:
symbol: з       code: 0101000   frequency: 1
symbol: а       code: 1000      frequency: 9
symbol: п       code: 0111      frequency: 9
symbol: о       code: 1101      frequency: 11
symbol: м       code: 0101001   frequency: 1
symbol: н       code: 11101     frequency: 6
symbol: и       code: 1011      frequency: 10
symbol: ,       code: 0110      frequency: 9
symbol:         code: 00        frequency: 31
symbol: б       code: 111001    frequency: 3
symbol: р       code: 111100    frequency: 3
symbol: т       code: 1010      frequency: 10
symbol: :       code: 1110001   frequency: 1
symbol: в       code: 11111     frequency: 7
symbol: л       code: 0100      frequency: 8
symbol: к       code: 1100      frequency: 11
symbol: е       code: 01011     frequency: 4
symbol: с       code: 1001      frequency: 9
symbol: -       code: 1110000   frequency: 1
symbol: .       code: 111101    frequency: 4
symbol: ы       code: 0101010   frequency: 1
symbol: ь       code: 0101011   frequency: 1

Encoding result: 0101000 1000 0111 1101 0101001 11101 1011 0110 00 111001 111100 1000 1010 1110001 00 11111 1101 0100 1100 00 11101 01011 00 1001 0111 1011 1010 0110 00
 11111 1101 0100 1100 00 1110000 00 1001 0111 1011 1010 0110 00 11101 1101 00 01011 1001 0100 1011 00 11111 1101 0100 1100 00 1001 0111 1011 1010 0110 00 1010 1101 00 1
1111 1101 0100 1100 00 11101 01011 00 1001 0111 1011 1010 111101 00 1011 00 1010 0101010 00 1001 0111 1011 0110 00 111001 111100 1000 1010 0110 00 11101 1101 00 11101 0
1011 00 1001 0111 1011 0110 00 1100 1000 1100 00 11111 1101 0100 1100 0110 00 1000 00 1001 0111 1011 0110 00 1100 1000 1100 00 111001 111100 1000 1010 00 11111 1101 010
0 1100 1000 111101 00 1001 0111 1000 1010 0101011 111101 00 11111 1101 0100 1100 111101
Memory usage: 593 bits

Compression coefficient: 2.02361

Decoding result: запомни, брат: волк не спит, волк - спит, но если волк спит, то волк не спит. и ты спи, брат, но не спи, как волк, а спи, как брат волка. спать. волк.
```

5. 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());
			}
		}
	};
}
```