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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Вычислительная математика»

Тема: Алгоритмы кодирования

Студент гр. 8302 _____ Халитов Ю.Р.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

Краткое описание реализуемого алгоритма и используемых структур данных

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 равен 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.

Используемые классы:

class **Map** (из лаб. работы №1) – ассоциативный массив, используется для хранения частоты появления символа в сообщении и для хранения кодов символов после кодирования.

class **Priority_Queue** – невозрастающая приоритетная очередь, используется для хранения элементов при формирования дерева Хаффмана.

struct **Node** – элемент очереди /дерева Хаффмана

Оценки временной сложности реализуемых методов

Таблица 1 – Оценки временной сложности методов класса Priority_Queue

Метод	Сложность
int left(int);	$O(1)$
int right(int);	$O(1)$
int parent(int);	$O(1)$
void up(int);	$O(\log(n))$
void down(int);	$O(\log(n))$
Priority_queue(Map<char, int>&);	$O(n)$
Node* extract_min();	$O(\log(n))$
void insert(Node*);	$O(\log(n))$
int cur_size();	$O(1)$

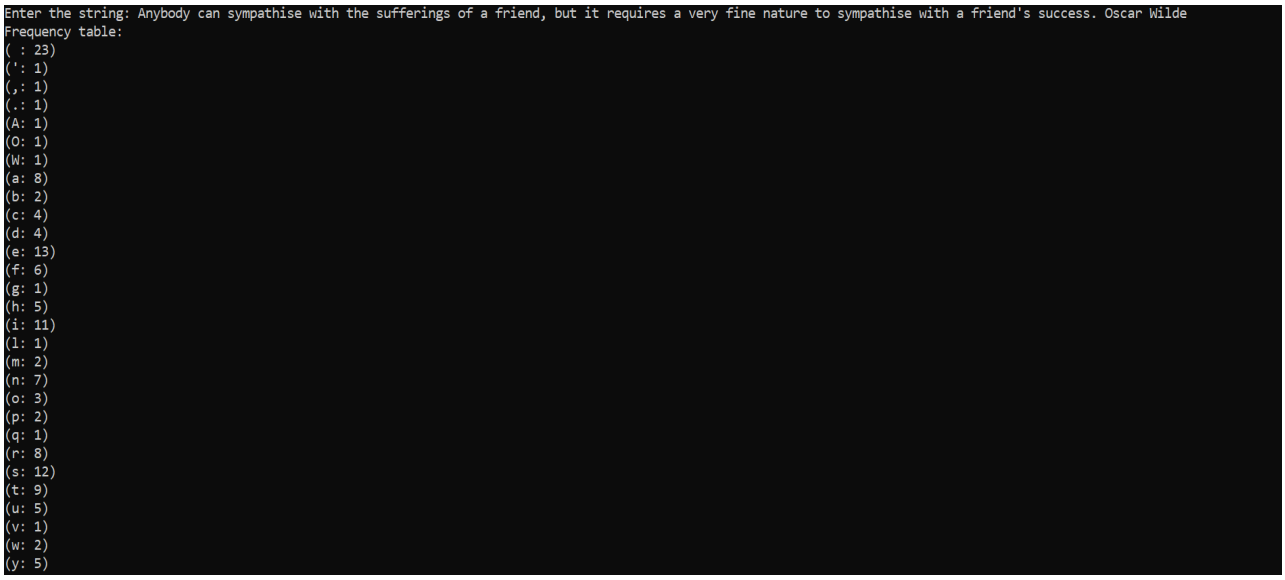
n – количество элементов в очереди

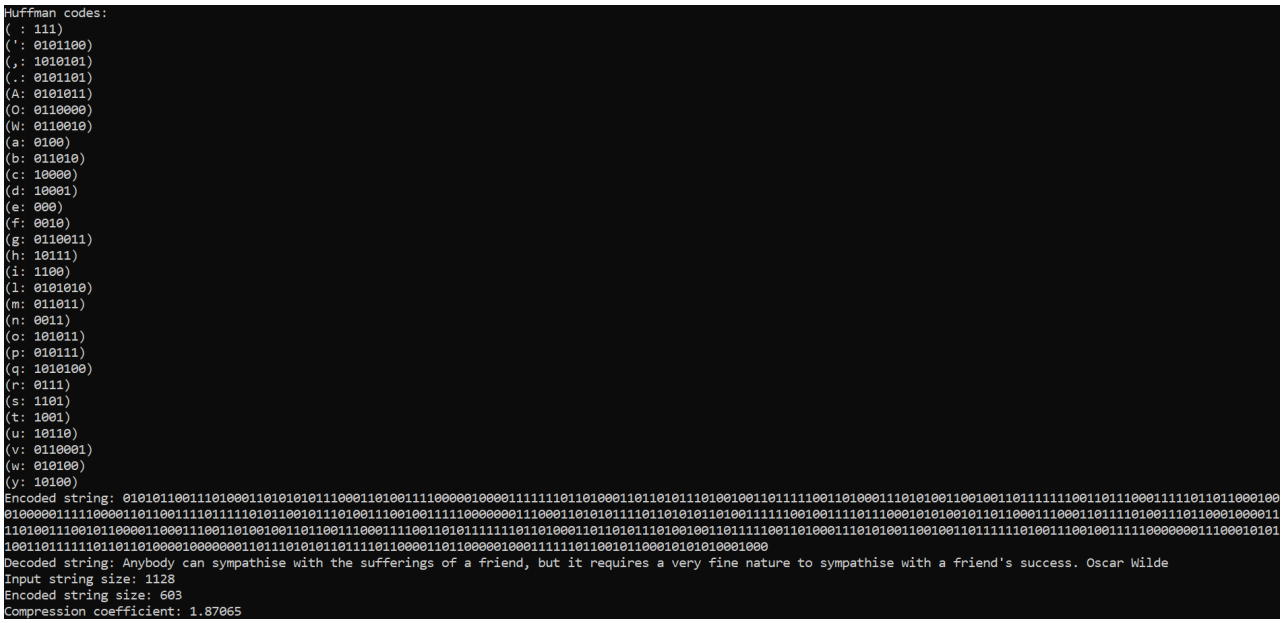
Таблица 2 – Оценки временной сложности методов алгоритма Хаффмана

Node* create_huffman_tree(Map<char, int>&);	$O(n\log(n))$ (если не учитывать инициализацию очереди элементами словаря, сложность которой $O(n)$), n – количество элементов в словаре
std::string encode(std::string, Map<char, std::string>&)	O(n), n – количество символов в строке
std::string decode(std::string, Node*)	O(n), n – количество битов в закодированной последовательности

Примеры работы программы

1. Anybody can sympathise with the sufferings of a friend, but it requires a very fine nature to sympathise with a friend's success. Oscar Wilde





2. An enemy can partly ruin a man, but it takes a good-natured injudicious friend to complete the thing and make it perfect. Mark Twain

```
Enter the string: An enemy can partly ruin a man, but it takes a good-natured injudicious friend to complete the thing and make it perfect. Mark Twain
Frequency table:
( : 23)
(,: 1)
(-: 1)
(.: 1)
(A: 1)
(M: 1)
(T: 1)
(a: 11)
(b: 1)
(c: 4)
(d: 5)
(e: 11)
(f: 2)
(g: 2)
(h: 2)
(i: 9)
(j: 1)
(k: 3)
(l: 2)
(m: 4)
(n: 11)
(o: 5)
(p: 3)
(r: 6)
(s: 2)
(t: 11)
(u: 5)
(w: 1)
(y: 2)
```

```
Huffman codes:
( : 00)
(,: 0111100)
(-: 0110111)
(.: 0111101)
(A: 0110110)
(M: 1000110)
(T: 1000111)
(a: 1100)
(b: 0111111)
(c: 01100)
(d: 10110)
(e: 1110)
(f: 011010)
(g: 010001)
(h: 101000)
(i: 1001)
(j: 0111110)
(k: 101001)
(l: 100001)
(m: 01110)
(n: 1101)
(o: 10101)
(p: 01001)
(r: 0101)
(s: 100010)
(t: 1111)
(u: 10111)
(w: 010000)
(y: 100000)
Encoded string: 011011011010011101101111001110100000000110011001101000100111000101111110000110000000010110111100111010011000001110110011010111100000111111011111110010011111
1001111110010100111101000100011000001000110101101011011001101111101110011111011101011110101100010011101011111010111101101001011001001101011011110001000011010010110011110110
1101100011111010100011001010101110010011000011110111111100011111010001110001111101000100111010100010011001101101100001110110010100111100010011111000100111100101011010111001
100111101111010010001101100010110100100100011101000011001001101
Decoded string: An enemy can partly ruin a man, but it takes a good-natured injudicious friend to complete the thing and make it perfect. Mark Twain
Input string size: 1056
Encoded string size: 564
Compression coefficient: 1.87234
```

3. A friend is a second self. Aristotle

```
Enter the string: A friend is a second self. Aristotle
Frequency table:
( : 6)
(.: 1)
(A: 2)
(a: 1)
(c: 1)
(d: 2)
(e: 4)
(f: 2)
(i: 3)
(l: 2)
(n: 2)
(o: 2)
(r: 2)
(s: 4)
(t: 2)
Huffman codes:
( : 111)
(.: 110110)
(A: 0111)
(a: 110111)
(c: 11010)
(d: 0000)
(e: 100)
(f: 0011)
(i: 1100)
(l: 0101)
(n: 0110)
(o: 0100)
(r: 0010)
(s: 101)
(t: 0001)
Encoded string: 011111100110010110010001100000111110010111111011111110110011010010001100000111101100010100111101101110111001011001010001010000010101100
Decoded string: A friend is a second self. Aristotle
Input string size: 288
Encoded string size: 135
Compression coefficient: 2.13333
```

Листинг

```
#include <iostream>
#include <string>
#include "Map.h"

struct Node
{
	char c;
	int freq;
	Node* left;
	Node* right;
	Node(char c, int freq) : c{ c }, freq{ freq }, left{
nullptr }, right{ nullptr } {}
	Node(char c, int freq, Node* left, Node* right) : c{ c },
freq { freq }, left{ left }, right{ right } {}
};

class Priority_queue
{
	Node** data;
	const int MAX_SIZE;
	int size;
	int left(int);
	int right(int);
	int parent(int);
	void up(int);
	void down(int);
public:
	Priority_queue(Map<char, int>&);
	Node* extract_min();
	void insert(Node*);
	int cur_size();
};


int Priority_queue::left(int i)
{
	return 2 * i + 1;
}

int Priority_queue::right(int i)
{
	return 2 * i + 2;
}

int Priority_queue::parent(int i)
{
	return (i - 1) / 2;
}

void Priority_queue::up(int i)
{
	while (i != 0 && data[i]->freq < data[parent(i)]->freq)
	{
		std::swap(data[i], data[parent(i)]);
		i = parent(i);
	}
```

```
}

void Priority_queue::down(int i)
{
     int l = left(i);
     int r = right(i);
     int least = i;
     if (l < size && data[l]->freq < data[i]->freq) least = l;
     if (r < size && data[r]->freq < data[least]->freq) least = r;
     if (i != least)
     {
          std::swap(data[i], data[least]);
          down(least);
     }
}

Priority_queue::Priority_queue(Map<char, int>& map) : MAX_SIZE{
map.node_count() }, size{ MAX_SIZE }
{
     std::vector<char> char_vec = map.get_keys();
     std::vector<int> freq_vec = map.get_values();
     data = new Node * [size];
     for (int i = 0; i < size; ++i) data[i] = new
Node(char_vec[i], freq_vec[i]);
     for (int i = size / 2 - 1; i >= 0; --i) down(i);
}

Node* Priority_queue::extract_min()
{
     if (size <= 0) throw "nothing to extract";
     Node * min = data[0];
     data[0] = data[size - 1];
     --size;
     down(0);
     return min;
}

void Priority_queue::insert(Node* node)
{
     if (size + 1 > MAX_SIZE) throw "the queue is full";
     data[size++] = node;
     up(size - 1);
}

int Priority_queue::cur_size()
{
     return size;
}


Node* create_huffman_tree(Map<char, int>& freq)
{
     Priority_queue queue(freq);
     int queue_size = queue.cur_size();
     for (int i = 0; i < queue_size - 1; ++i)
     {
          Node* left_node = queue.extract_min();
          Node* right_node = queue.extract_min();
```

```
          Node* new_node = new Node('\0', left_node->freq +
right_node->freq, left_node, right_node);
          queue.insert(new_node);
     }
     return queue.extract_min();
}


std::string encode(std::string input_string, Map<char,
std::string>& codes)
{
     std::string encode_string = "";
     for (char c : input_string) encode_string += codes.find(c,
"err");
     return encode_string;
}

std::string decode(std::string encoded_string, Node* huffman_tree)
{
     Node* cur_node = huffman_tree;
     std::string decoded_string = "";
     for (char c : encoded_string)
     {
          if (c == '0') cur_node = cur_node->left;
          else if (c == '1') cur_node = cur_node->right;
          if (cur_node->left == nullptr && cur_node->right ==
nullptr)
          {
               decoded_string += cur_node->c;
               cur_node = huffman_tree;
          }
     }
     return decoded_string;
}

void create_huffman_codes_map(Map<char, std::string>& codes, Node*
root, std::string str)
{
     if (root == nullptr) return;
     if (root->c != '\0') codes.insert(root->c, str);
     create_huffman_codes_map(codes, root->left, str + "0");
     create_huffman_codes_map(codes, root->right, str + "1");
}


using namespace std;

int main()
{
     string input_string;
    cout << "Enter the string: ";
     getline(cin, input_string);

     Map<char, int> freq;
     for (char c : input_string) freq.insert(c, freq.find(c, 0) +
1);
     cout << "Frequency table: " << endl;
     freq.print();
```

```
    auto huffman_tree = create_huffman_tree(freq);
    Map<char, string> codes;
    create_huffman_codes_map(codes, huffman_tree, "");
    cout << "Huffman codes: " << endl;
    codes.print();

    string encoded_string = encode(input_string, codes);
    cout << "Encoded string: " << encoded_string << endl;
    string decoded_string = decode(encoded_string, huffman_tree);
    cout << "Decoded string: " << decoded_string << endl;
    int encoded_size = encoded_string.size();
    int decoded_size = decoded_string.size() * 8;
    double coef = (double)decoded_size / encoded_size;
    cout << "Input string size: " << decoded_size << endl;
    cout << "Encoded string size: " << encoded_size << endl;
    cout << "Compression coefficient: " << coef << endl;
}
```