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: Алгоритм кодирования Шеннона-Фано

Студент гр. 0302 _____ Касаткин А.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2022

Задача: Реализовать кодирование и декодирование по алгоритму Шеннона-Фано входной строки, вводимой через консоль

Описание реализуемых алгоритмов:
Символы первичного алфавита выписывают по убыванию вероятностей (частот встречаемости). Символы полученного алфавита делят на две части, суммарные вероятности символов которых максимально близки друг другу.
В префиксном коде для первой части алфавита присваивается двоичная цифра «1», второй части — «0» (или наоборот). Полученные части рекурсивно делятся и их частям назначаются соответствующие двоичные цифры в префиксном коде.
Передавать можно кодирующее дерево или таблицу символов/кодов + закодированную последовательность. На принимающей стороне требуется побитово читать строку, оставляя путь от корня к узлу. Если передавалась кодирующая таблица, то в закодированной последовательности должны присутствовать символы-разделители.

Оценка временной сложности:
Encode O(N)
Decode O(N)
EncodeMap O(N)
Print_requency_table O(N)

Описание unit-тестов:
EncodedTextShannonFanoCoderTest
Проверка на правильность кодирования текста
DecodedTextShannonFanoCoderTest
Проверка на правильность декодирования текста

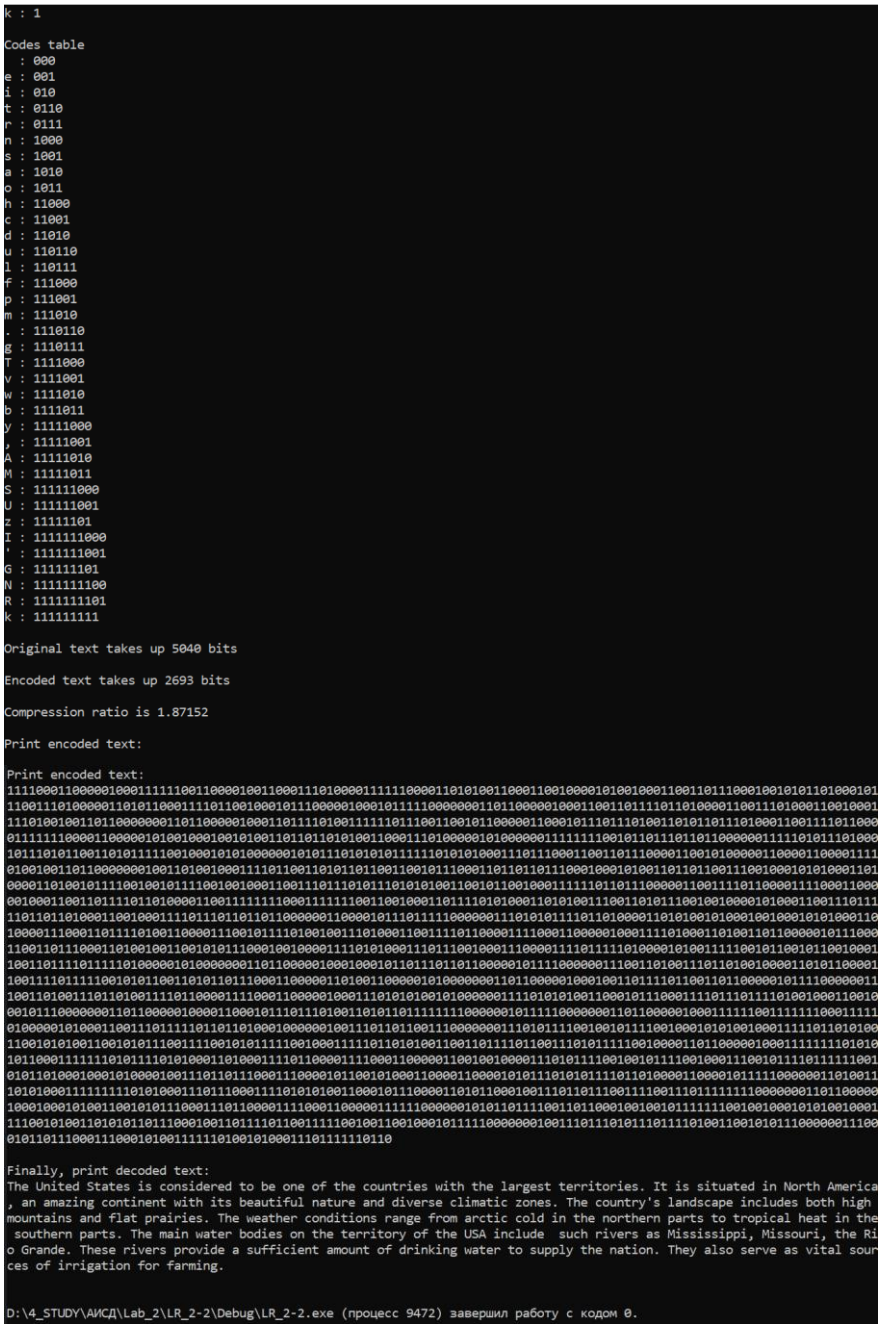
Результат работы программы:

```
Enter text to encode it by ShannonFano algorithm
Input finish after 'Enter' press

The United States is considered to be one of the countries with the largest territories. It is situated in North America
, an amazing continent with its beautiful nature and diverse climatic zones. The country’s landscape includes both high
mountains and flat prairies. The weather conditions range from arctic cold in the northern parts to tropical heat in the
 southern parts. The main water bodies on the territory of the USA include  such rivers as Mississippi, Missouri, the Ri
o Grande. These rivers provide a sufficient amount of drinking water to supply the nation. They also serve as vital sour
ces of irrigation for farming.

Text characters frequency table

  : 102
e : 55
i : 54
t : 52
r : 42
n : 39
s : 39
a : 37
o : 36
h : 25
c : 18
d : 16
u : 16
l : 12
f : 11
p : 10
m : 8
. : 7
g : 7
T : 6
v : 6
w : 5
b : 4
y : 4
, : 3
A : 2
M : 2
S : 2
U : 2
z : 2
I : 1
' : 1
G : 1
N : 1
R : 1
```



Листинг:

Main.cpp

```
#include "Class.h"
#include <iostream>

using namespace std;

int main()
{
	Map<char, int> MyMap;
	char letter;
	int different_letter = 0;
	string text = "";

	cout << "Enter text to encode it by ShannonFano algorithm\nInput finish after 'Enter' press\n\n";

	while (cin.get(letter)) {
		if (letter == '\n') break;
		text += letter;
```

```
                if (MyMap.contains(letter)) {
                        MyMap.find(letter)->value++;
                }
                else {
                        MyMap.insert(letter, 1);
                        different_letter++;
                }
        }

        ShannonFano_Coder My(MyMap, different_letter, text);

        cout << "\nText characters frequency table\n\n";
        My.print_frequency_table();

        My.ShannonFano_encoding(0, different_letter - 1);
        cout << "\nCodes table\n";
        My.print_code_table();
        cout << "\nOriginal text takes up " << My.original_text_size() << " bits\n\n";
        cout << "Encoded text takes up " << My.encoded_text_size() << " bits\n\n";
        cout << "Compression ratio is " << My.compression_ratio() << "\n\n";
        cout << "Print encoded text:\n";
        My.print_encoded_text();
        cout << "\n\nFinally, print decoded text:\n";
        My.print_decoded_text();
        cout << "\n\n";

}
```

Class.h

```
#pragma once
#include <iostream>
#include <iomanip>

using namespace std;

template <typename first_typename>

class stack
{
private:
        class Node
        {
        public:
                first_typename value;
                Node* prev;
                Node(const int element) :value(element), prev(0) {}
        };
        Node* current;
        size_t stack_size;
public:
        stack() :current(0), stack_size(0) {}

        ~stack()
        {
                while (current)
                        pop();
        }

        void push(const first_typename element)
        {
                Node* temp = current;
                current = new Node(element);
                current->prev = temp;
                stack_size++;
        }

        void pop()
        {
```

```
		if (stack_size) {
			Node* temp = current->prev;
			delete current;
			current = temp;
			stack_size--;
		}
		else {
			throw out_of_range("Error! Stack is empty!");
		}
	}

	first_typename top() { return current->value; }

	bool empty() { return !stack_size; }

	size_t size() { return stack_size; }
};

template <typename first_typename>

class List {
	class Node {
	public:
		first_typename data;
		Node* next;
	};

	void delete_head() {
		if (head != NULL)
		{
			size_list--;
			Node* to_delete = head;
			to_delete = head;
			head = head->next;
			delete to_delete;
		}
		else
			throw out_of_range("Error! List is empty!");
	}

public:
	Node* head;
	int size_list;

	List() {
		head = NULL;
		size_list = 0;
	}

	~List() {
		size_list = 0;
	}

	void insert_node(first_typename data) {
		Node* new_node = new Node;
		new_node->data = data;
		new_node->next = NULL;
		size_list++;

		if (head == NULL) {
			head = new_node;
			return;
		}

		Node* current_last = head;
		while (current_last->next != NULL) {
			current_last = current_last->next;
		}
		current_last->next = new_node;
```

```
	}

	first_typename get_head() {
		if (size_list != 0)
			return head->data;
		else
			throw out_of_range("Error! List is empty!");
	}

	void set_next() {
		if (head != NULL)
			head = head->next;
		else
			throw out_of_range("Error! There is not next element!");
	}

	int size() {
		return size_list;
	}

};

template <typename first_typename, typename second_typename>

class Map {
private:

	class Node {
	public:
		first_typename key;
		second_typename value;
		Node* parent;
		Node* left;
		Node* right;
		int color;

		Node() {
			color = 0;
			left = nullptr;
			right = nullptr;
		}

		Node(first_typename outside_key, second_typename outside_value) {
			parent = nullptr;
			key = outside_key;
			value = outside_value;
			color = 1;
		}
	};

	Node* root;
	Node* Nil;


	void moving(Node* host_node, Node* moving_node) {
		if (host_node->parent == nullptr) {
			root = moving_node;
		}
		else if (host_node == host_node->parent->left) {
			host_node->parent->left = moving_node;
		}
		else {
			host_node->parent->right = moving_node;
		}
		moving_node->parent = host_node->parent;
	}

	void left_rotate(Node* current_node) {
		Node* right_child = current_node->right;
```

```
        current_node->right = right_child->left;
        if (right_child->left != Nil) {
                right_child->left->parent = current_node;
        }
        right_child->parent = current_node->parent;
        if (current_node->parent == nullptr) {
                root = right_child;
        }
        else if (current_node == current_node->parent->left) {
                current_node->parent->left = right_child;
        }
        else {
                current_node->parent->right = right_child;
        }
        right_child->left = current_node;
        current_node->parent = right_child;
}

void right_rotate(Node* current_node) {
        Node* left_child = current_node->left;
        current_node->left = left_child->right;
        if (left_child->right != Nil) {
                left_child->right->parent = current_node;
        }
        left_child->parent = current_node->parent;
        if (current_node->parent == nullptr) {
                root = left_child;
        }
        else if (current_node == current_node->parent->right) {
                current_node->parent->right = left_child;
        }
        else {
                current_node->parent->left = left_child;
        }
        left_child->right = current_node;
        current_node->parent = left_child;
}

void fix_after_insert(Node* x) {
        Node* uncle;
        while (x->parent->color == 1) {
                if (x->parent == x->parent->parent->right) {
                        uncle = x->parent->parent->left;
                        if (uncle->color == 1) {
                                uncle->color = 0;
                                x->parent->color = 0;
                                x->parent->parent->color = 1;
                                x = x->parent->parent;
                        }
                        else {
                                if (x == x->parent->left) {
                                        x = x->parent;
                                        right_rotate(x);
                                }
                                x->parent->color = 0;
                                x->parent->parent->color = 1;
                                left_rotate(x->parent->parent);
                        }
                }
                else {
                        uncle = x->parent->parent->right;

                        if (uncle->color == 1) {
                                uncle->color = 0;
                                x->parent->color = 0;
                                x->parent->parent->color = 1;
                                x = x->parent->parent;
                        }
                        else {
```

```
                        if (x == x->parent->right) {
                                x = x->parent;
                                left_rotate(x);
                        }

                        x->parent->color = 0;
                        x->parent->parent->color = 1;
                        right_rotate(x->parent->parent);
                }
        }
        if (x == root) {
                break;
        }
    }
    root->color = 0;
}

void fix_after_remove(Node* x) {
    Node* brother;
    while (x != root && x->color == 0) {
        if (x == x->parent->left) {
            brother = x->parent->right;
            if (brother->color == 1) {
                brother->color = 0;
                x->parent->color = 1;
                left_rotate(x->parent);
                brother = x->parent->right;
            }

            if (brother->left->color == 0 && brother->right->color == 0) {
                brother->color = 1;
                x = x->parent;
            }
            else {
                if (brother->right->color == 0) {
                    brother->left->color = 0;
                    brother->color = 1;
                    right_rotate(brother);
                    brother = x->parent->right;
                }

                brother->color = x->parent->color;
                x->parent->color = 0;
                brother->right->color = 0;
                left_rotate(x->parent);
                x = root;
            }
        }
        else {
            brother = x->parent->left;
            if (brother->color == 1) {
                brother->color = 0;
                x->parent->color = 1;
                right_rotate(x->parent);
                brother = x->parent->left;
            }

            if (brother->right->color == 0 && brother->left->color == 0) {
                brother->color = 1;
                x = x->parent;
            }
            else {
                if (brother->left->color == 0) {
                    brother->right->color = 0;
                    brother->color = 1;
                    left_rotate(brother);
                    brother = x->parent->left;
                }
```

```
                        brother->color = x->parent->color;
                        x->parent->color = 0;
                        brother->left->color = 0;
                        right_rotate(x->parent);
                        x = root;
                    }
                }
        }
        x->color = 0;
}

Node* find_algorithm(Node* current_node, first_typename key) {
        if (key == current_node->key) {
                return current_node;
        }

        if (current_node == Nil) {
                throw out_of_range("Error! Couldn't find key in the tree!");
        }

        if (key < current_node->key) {
                return find_algorithm(current_node->left, key);
        }
        return find_algorithm(current_node->right, key);
}

bool contains_algorithm(Node* current_node, first_typename key) {
        if (key == current_node->key) {
                return true;
        }

        if (current_node == Nil) {
                return false;
        }

        if (key < current_node->key) {
                return contains_algorithm(current_node->left, key);
        }
        return contains_algorithm(current_node->right, key);
}

void RBTree_to_stack(Node* node, stack<first_typename>& MyStack) {
        if (node != Nil) {
                MyStack.push(node->key);
                RBTree_to_stack(node->left, MyStack);
                RBTree_to_stack(node->right, MyStack);
        }
}

void get_keys_algorithm(Node* node, List<first_typename>& My_List) {
        if (node != Nil) {
                My_List.insert_node(node->key);
                get_keys_algorithm(node->left, My_List);
                get_keys_algorithm(node->right, My_List);
        }
}

void get_values_algorithm(Node* node, List<second_typename>& My_List) {
        if (node != Nil) {
                My_List.insert_node(node->value);
                get_values_algorithm(node->left, My_List);
                get_values_algorithm(node->right, My_List);
        }
}

void print_with_indent(Node* node, int level = 0)
{
        if (node != Nil)
        {
```

```
                print_with_indent(node->left, level + 1);
                for (int i = 0; i < level; i++) cout << "   ";
                cout << setw(3) << node->key;
                if (node->color == 0) { cout << " (Black)" << '\n'; }
                else { cout << " (Red)" << '\n'; }
                print_with_indent(node->right, level + 1);
        }
    }

public:
    Map() {
        Nil = new Node;
        root = Nil;
    }

    ~Map() {
        clear();
    }

    void insert(first_typename key, second_typename value) {

        Node* inserting_node = new Node(key, value);
        inserting_node->left = Nil;
        inserting_node->right = Nil;

        Node* y = nullptr;
        Node* x = root;

        while (x != Nil) {
            y = x;
            if (inserting_node->key < x->key) {
                x = x->left;
            }
            else {
                x = x->right;
            }
        }

        inserting_node->parent = y;
        if (y == nullptr) {
            root = inserting_node;
        }
        else if (inserting_node->key < y->key) {
            y->left = inserting_node;
        }
        else {
            y->right = inserting_node;
        }

        if (inserting_node->parent == nullptr) {
            inserting_node->color = 0;
            return;
        }

        if (inserting_node->parent->parent == nullptr) {
            return;
        }

        fix_after_insert(inserting_node);
    }

    void remove(first_typename key) {
        Node* node_to_be_deleted = Nil;
        Node* x;
        Node* y;

        node_to_be_deleted = find(key);

        y = node_to_be_deleted;
```

```
            int y_original_color = y->color;
            if (node_to_be_deleted->left == Nil) {
                x = node_to_be_deleted->right;
                moving(node_to_be_deleted, node_to_be_deleted->right);
            }
            else if (node_to_be_deleted->right == Nil) {
                x = node_to_be_deleted->left;
                moving(node_to_be_deleted, node_to_be_deleted->left);
            }
            else {
                y = node_to_be_deleted->right;
                while (y->left != Nil) {
                    y = y->left;
                }
                y_original_color = y->color;
                x = y->right;
                if (y->parent == node_to_be_deleted) {
                    x->parent = y;
                }
                else {
                    moving(y, y->right);
                    y->right = node_to_be_deleted->right;
                    y->right->parent = y;
                }

                moving(node_to_be_deleted, y);
                y->left = node_to_be_deleted->left;
                y->left->parent = y;
                y->color = node_to_be_deleted->color;
            }
            delete node_to_be_deleted;

            if (y_original_color == 0) { fix_after_remove(x); }
        }

        Node* find(first_typename key) { return find_algorithm(root, key); }

        void clear() {
            stack<first_typename> My_Stack;
            RBTree_to_stack(root, My_Stack);
            while (My_Stack.size()) {
                remove(My_Stack.top());
                My_Stack.pop();
            }
        }

        List<first_typename> get_keys() {
            List<first_typename> My_List;
            get_keys_algorithm(root, My_List);
            return My_List;
        }

        List<second_typename> get_values() {
            List<second_typename> My_List;
            get_values_algorithm(root, My_List);
            return My_List;
        }

        void print() { return print_with_indent(root); }

        bool contains(first_typename key) { return contains_algorithm(root, key); }
};

class ShannonFano_Coder {
private:
        char* char_array;
        int* frequency;
        string* code;
        string original_text;
```

```
	int alphabet_size;

	void Bubble_Sort() {
		for (int i = 0; i < alphabet_size; i++) {
			for (int j = 0; j < alphabet_size - 1; j++) {
				if (frequency[j] < frequency[j + 1]) {
					int int_temporary = frequency[j];
					char char_temporary = char_array[j];
					frequency[j] = frequency[j + 1];
					char_array[j] = char_array[j + 1];
					frequency[j + 1] = int_temporary;
					char_array[j + 1] = char_temporary;
				}
			}
		}
	}

	int get_index(char letter) {
		for (int i = 0; i < alphabet_size; i++) {
			if (char_array[i] == letter) return i;
		}
	}

	int get_index(string substring) {
		for (int i = 0; i < alphabet_size; i++) {
			if (code[i] == substring) return i;
		}
	}

	int min(int first, int second) {
		return !(second < first) ? first : second;
	}

	bool contains(string substring) {
		for (int i = 0; i < alphabet_size; i++) {
			if (code[i] == substring) return true;
		}
		return false;
	}

public:
	ShannonFano_Coder(Map<char, int>& MyCustomMap, const int different_letter, string
text_to_encode) {
		char_array = new char[different_letter];
		frequency = new int[different_letter];
		code = new string[different_letter];
		alphabet_size = different_letter;
		original_text = text_to_encode;

		List<char> map_keys = MyCustomMap.get_keys();
		List<int> map_values = MyCustomMap.get_values();

		for (int i = 0; i < alphabet_size; i++)
		{
			char_array[i] = map_keys.get_head();
			frequency[i] = map_values.get_head();
			code[i] = "";
			map_keys.set_next();
			map_values.set_next();
		}

		Bubble_Sort();

	}

	~ShannonFano_Coder() {
		delete[] char_array;
		delete[] frequency;
		delete[] code;
```

```
}

void print_frequency_table() {
	for (int i = 0; i < alphabet_size; i++)
	{
		cout << char_array[i] << " : " << frequency[i] << '\n';
	}
}

void ShannonFano_encoding(int begin, int end) {
	if (begin == end)
		return;
	int left = begin;
	int right = end;
	int sum_left = 0;
	int sum_right = 0;
	while (left <= right) {
		if (sum_left <= sum_right) {
			sum_left += frequency[left];
			left++;
		}
		else {
			sum_right += frequency[right];
			right--;
		}
	}
	for (int i = begin; i < left; i++) {
		code[i] += "0";
	}
	for (int i = left; i <= end; i++) {
		code[i] += "1";
	}
	ShannonFano_encoding(begin, left - 1);
	ShannonFano_encoding(left, end);
}

void print_code_table() {
	for (int i = 0; i < alphabet_size; i++)
	{
		cout << char_array[i] << " : " << code[i] << '\n';
	}
}

int original_text_size() {
	int sum_size = 0;
	for (int i = 0; i < alphabet_size; i++)
	{
		sum_size += 8 * frequency[i];
	}
	return sum_size;
}

int encoded_text_size() {
	int sum_size = 0;
	for (int i = 0; i < alphabet_size; i++)
	{
		sum_size += code[i].size() * frequency[i];
	}
	return sum_size;
}

float compression_ratio() {
	int original_size = original_text_size();
	float divided = original_size / 1.0f;
	int encoded_size = encoded_text_size();
	float divisor = encoded_size / 1.0f;
	float quotient = divided / divisor;
	return quotient;
}
```

```
	string encoded_text() {
		string encoded_text = "";
		for (int i = 0; i < original_text.size(); i++) {
			encoded_text += code[get_index(original_text.at(i))];
		}
		return encoded_text;
	}

	string decoded_text() {
		string decoded_text = "";
		string text_to_decode = encoded_text();
		int max_code_lenght = 0;
		for (int i = 0; i < alphabet_size; i++) {
			if (code[i].size() > max_code_lenght) max_code_lenght = code[i].size();
		}
		while (text_to_decode.size() > 0) {
			for (int i = 0; i <= min(max_code_lenght, text_to_decode.size()); i++) {
				string substring = text_to_decode.substr(0, i);
				if (contains(substring)) {
					decoded_text += char_array[get_index(substring)];
					text_to_decode.erase(0, i);
					break;
				}
			}
		}
		return decoded_text;
	}

	void print_encoded_text() { cout << encoded_text(); }

	void print_decoded_text() { cout << decoded_text(); }
};
```

Text.txt

```
The United States is considered to be one of the countries with the largest territories. It is situated in North America, an amazing continent with its beautiful nature and diverse climatic zones. The country’s landscape includes both high mountains and flat prairies. The weather conditions range from arctic cold in the northern parts to tropical heat in the southern parts. The main water bodies on the territory of the USA include  such rivers as Mississippi, Missouri, the Rio Grande. These rivers provide a sufficient amount of drinking water to supply the nation. They also serve as vital sources of irrigation for farming.
```