**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»**

**Вариант 2**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8301 |  | Попурей Н |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

Оглавление

[1. Постановка задачи 3](#_Toc40701175)

[2. Оценка временной сложности 3](#_Toc40701176)

[3. Описание реализованных юнит-тестов 3](#_Toc40701177)

[4. Пример работы 3](#_Toc40701178)

[5. Листинг 4](#_Toc40701179)

[Lab2.cpp 4](#_Toc40701180)

[Header.h 5](#_Toc40701181)

[UnitTest1.cpp 10](#_Toc40701182)

[6. Вывод 11](#_Toc40701183)

# **Постановка задачи**

Реализовать алгоритм кодировки Шеннона-Фано для решения следующих задач:

1. Кодировка строки (fanoCoding)
2. Декодировка строки (fanoDecoding)

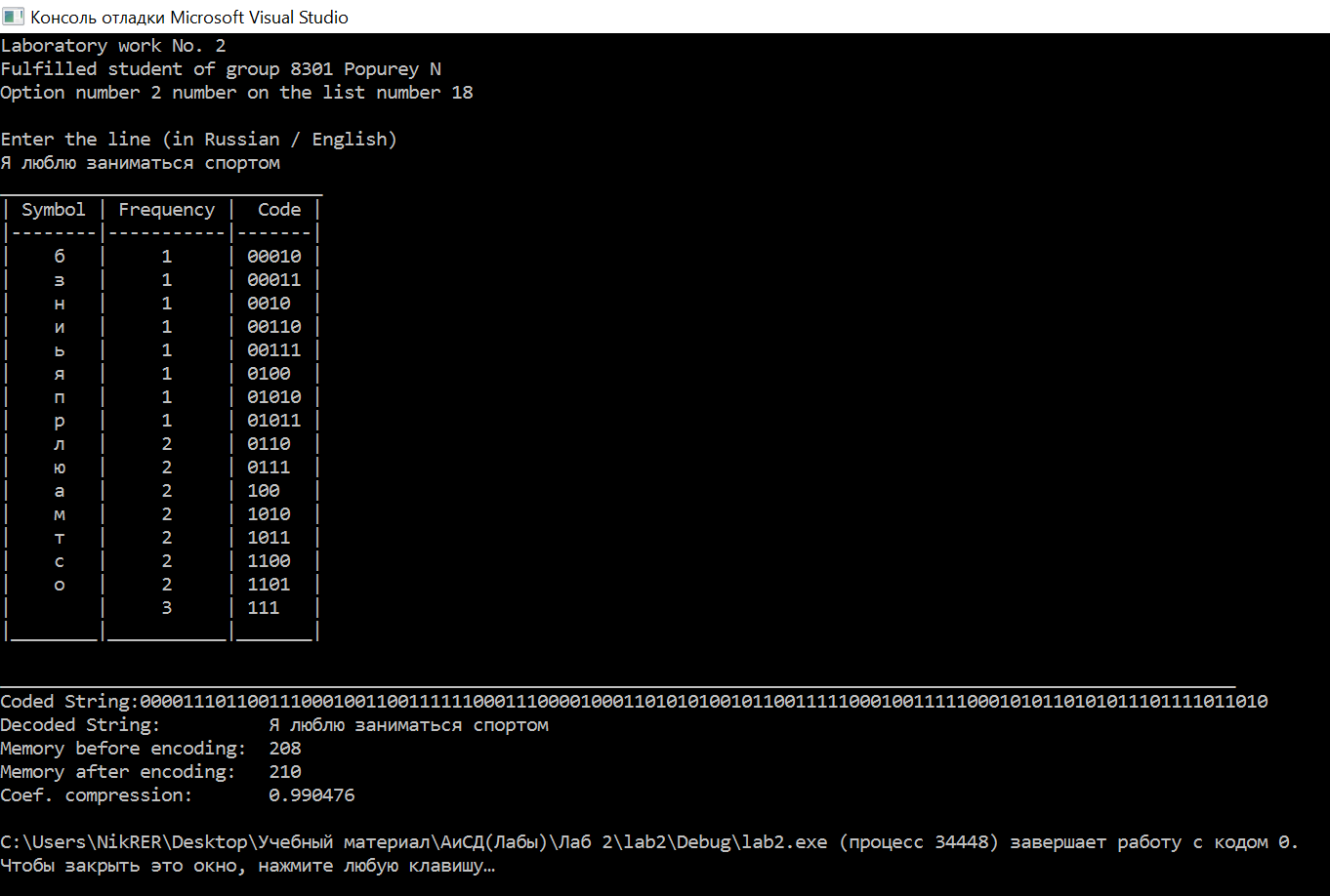
# Оценка временной сложности

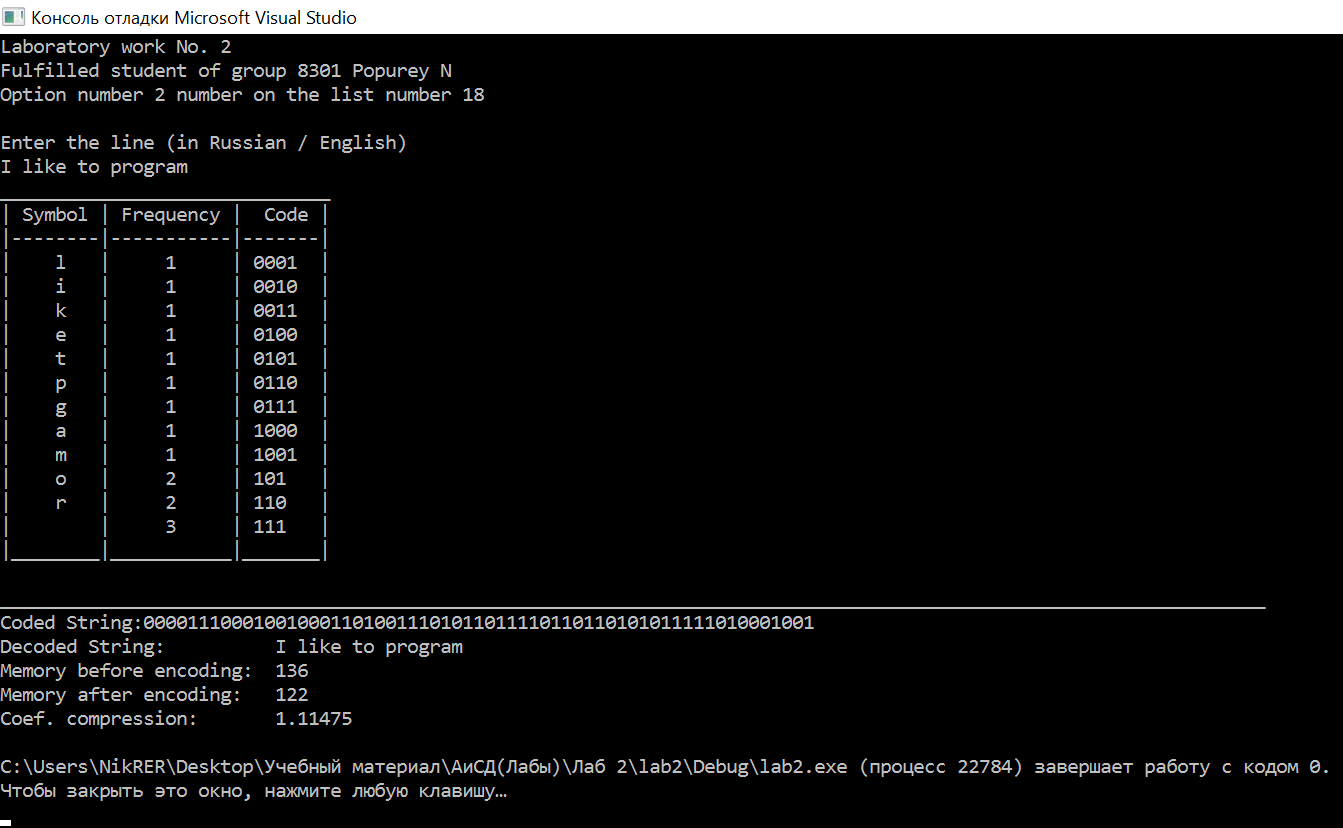
1. Кодировка строки (fanoCoding) - О(NlogN)
2. Декодировка строки (fanoDecoding) - О(NlogN)

# Описание реализованных юнит-тестов

1. Попытка кодирования без ввода строки TEST\_METHOD(fanoEmptyList)
2. Кодирование и декодирование сообщения TEST\_METHOD(fanoCodingDecoding)
3. Проверка добавления символа в список TEST\_METHOD(check)
4. Проверка подсчёта частот TEST\_METHOD(input)

# Пример работы





# Листинг

## Lab2.cpp

#include "pch.h"

#include <iostream>

#include <windows.h>

#include <string>

#include "Header.h"

using namespace std;

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

string text, codedText, decodedText;

list\* newList = new list;

double codedMem, decodedMem, compression;

cout << "Laboratory work No. 2" << endl;

cout << "Fulfilled student of group 8301 Popurey N" << endl;

cout << "Option number 2 number on the list number 18" << endl << endl;

cout << "Enter the line (in Russian / English)" << endl;

getline(cin, text);

codedText = newList->fanoCoding(text);

decodedText = newList->fanoDecoding(codedText);

newList->tableOutput();

decodedMem = 8 \* decodedText.length();

codedMem = 2 \* codedText.length();

compression = decodedMem / codedMem;

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "Coded String:" << codedText << endl;

cout << "Decoded String: " << decodedText << endl;

cout << "Memory before encoding: " << decodedMem << endl;

cout << "Memory after encoding: " << codedMem << endl;

cout << "Coef. compression: " << compression << endl;

delete newList;

return 0;

}

## Header.h

#pragma once

#include <iostream>

#include <string>

using namespace std;

struct listElement

{

char symbol;

int frequency;

string code;

listElement\* next, \*prev;

};

class list

{

private:

listElement\* head, \*tail;

public:

list() // constructor

{

head = nullptr;

tail = nullptr;

}

~list() // destructor

{

if (head) // if list is not empty

{

listElement\* buf;

while (head != tail) // until there is one element

{

buf = head;

head = head->next;

delete buf;

}

delete head;

head = nullptr;

tail = nullptr;

}

}

void check(char symbolInput) // filling the symbols list and counting frequency

{

bool found = false; // if symbol is already in list -> true

if (head)

{

listElement\* cur = head; // current element

if (cur->symbol == symbolInput) // if it is first symbol

{

found = true;

cur->frequency++;

}

while ((cur != tail) && (!found)) // until no more elements left or symbol found

{

cur = cur->next;

if (cur->symbol == symbolInput) // if symbol found

{

found = true;

cur->frequency++;

}

}

if (!found) this->pushBack(symbolInput); // if there is no this symbol in the list - add it

}

else this->pushBack(symbolInput); // if list is empty - add symbol

}

void input(string text) // input of data

{

int length = text.length();

for (int i = 0; i < length; i++)

{

check(text[i]);

}

}

void sortingFrequences()

{

if (head != tail) // if more than 2 elements in list

{

bool swap; // if there were swaps on this step

listElement\* buf, \*cur; // temporary element and current element

do

{

swap = false;

cur = head;

while (cur != tail)

{

if (cur->frequency > cur->next->frequency) // condition of swapping

{

buf = cur->next; // the next element after current

if (cur == head) head = buf; // if current is first element, change head

if (buf == tail) tail = cur; // if next element is last, change tail

if ((buf->next) && (cur->prev)) // if there are elements on two sides by those two elements

{

cur->next = buf->next;

buf->prev = cur->prev;

cur->prev->next = buf;

buf->next->prev = cur;

}

else if (buf->next) // if only on right side

{

cur->next = buf->next;

buf->prev = nullptr;

buf->next->prev = cur;

}

else if (cur->prev) // if only on left side

{

buf->prev = cur->prev;

cur->next = nullptr;

cur->prev->next = buf;

}

else // if not

{

buf->prev = nullptr;

cur->next = nullptr;

}

buf->next = cur;

cur->prev = buf;

cur = buf;

swap = true; // we swapped elements

}

cur = cur->next;

}

} while (swap);

}

}

void fano() // activator of recursive function

{

if (head)

{

string empty = "";

fanoRec(empty, empty, head, tail);

}

else throw logic\_error("Empty list");

}

void fanoRec(string& branch, string& fullBranch, listElement\* start, listElement\* end) // counting codes for symbols (branch - last turn to this element(1/0), fullBranch- all turns to last element)

{

listElement\* cur, \*leftEnd, \*rightStart; // current element, last element in left part, first element in right part

double fullSum = 0; // sum of all elements frequences

int sum = 0; // sum which we count to take a half

string curBranch = ""; // path to this element

curBranch = fullBranch + branch;

if (start == end) // if element is leaf

{

start->code = curBranch;

return;

}

cur = start;

fullSum += start->frequency;

while (cur != end) // counting sum of all elements

{

cur = cur->next;

fullSum += cur->frequency;

}

fullSum /= 2.; // counting a half of sum

cur = start;

while (fabs(fullSum - (sum + cur->frequency)) < fabs(fullSum - sum) && (cur != end)) // until we get half or one element element

{

sum += cur->frequency;

cur = cur->next;

}

rightStart = cur;

leftEnd = cur->prev;

string zero = "0";

string one = "1";

fanoRec(zero, curBranch, start, leftEnd); // the same for left part

fanoRec(one, curBranch, rightStart, end); // the same for right part

}

string codeSearch(char symbol) // searching the code of symbol

{

if (head)

{

listElement\* cur = head;

if (cur->symbol == symbol) return cur->code;

while (cur != tail)

{

cur = cur->next;

if (cur->symbol == symbol) return cur->code;

}

}

else return "";

}

int frequencySearch(char symbol) // searching the frequency of symbol

{

if (head)

{

listElement\* cur = head;

if (cur->symbol == symbol) return cur->frequency;

while (cur != tail)

{

cur = cur->next;

if (cur->symbol == symbol) return cur->frequency;

}

}

else return 0;

}

string fanoCoding(string decodedText) // coding the text by codes

{

string codedText = "";

input(decodedText);

sortingFrequences();

fano();

int length = decodedText.length();

for (int i = 0; i < length; i++)

{

codedText += this->codeSearch(decodedText[i]);

}

return codedText;

}

string fanoDecoding(string codedText)

{

int length = codedText.length();

int iterator = 0, level = 0; // iterator for string, level of code (amount of digits in it)

string decodedText = "";

listElement\* start, \*end, \*cur; // first and last elements in search area, current element

while (iterator < length)

{

start = head;

end = tail;

level = 0;

while (start != end) // until element found

{

if (codedText[iterator + level] == '0') // checking each symbol of code by level determination

{

cur = start; // if zero - we must search from start

while (cur->code[level] != '1') // searching the first element of right side

{

cur = cur->next;

}

end = cur->prev; // determining the last element of left side

level++; // change the level to next symbol

}

else // the same, but from end to start

{

cur = end;

while (cur->code[level] != '0')

{

cur = cur->prev;

}

start = cur->next;

level++;

}

}

decodedText += start->symbol;

iterator += level; // moving to next code

}

return decodedText;

}

void tableOutput() // output the table of frequences and codes

{

if (head)

{

listElement\* cur = head;

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "| Symbol | Frequency | Сode |" << endl;

cout << "|--------|-----------|-------|" << endl;

cur = cur->next;

if (cur->code.length() == 2) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

if (cur->code.length() == 3) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

if (cur->code.length() == 4) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

if (cur->code.length() == 5) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

while (cur != tail)

{

cur = cur->next;

if (cur->code.length() == 2) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

if (cur->code.length() == 3) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

if (cur->code.length() == 4) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

if (cur->code.length() == 5) {

//cur = cur->next;

cout << "| " << cur->symbol << " | " << cur->frequency << " | " << cur->code << " |" << endl;

}

}

cout << "|\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_|" << endl;

cout << endl;

}

}

void pushBack(char symbolInput) // adding an element to the end of the list

{

listElement\* newElement = new listElement;

if (head) // if list is not empty

{

tail->next = newElement;

newElement->next = nullptr;

newElement->prev = tail;

tail = newElement;

newElement->symbol = symbolInput;

newElement->frequency = 1;

newElement->code = "";

}

else

{

head = newElement;

tail = head;

newElement->next = nullptr;

newElement->prev = nullptr;

newElement->symbol = symbolInput;

newElement->frequency = 1;

newElement->code = "";

}

}

};

## UnitTest1.cpp

#include "stdafx.h"

#include <CppUnitTest.h>

#include <C:\Users\NikRER\Desktop\Учебный материал\АиСД(Лабы)\Лаб 2\lab2\lab2\Header.h>

#include "stdafx.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

Assert assert;

namespace UnitTest1

{

TEST\_CLASS(UnitTest1)

{

public:

TEST\_METHOD(fanoEmptyList)

{

bool error = false;

list\* newList = new list;

try { newList->fano(); }

catch (logic\_error) { error = true; }

assert.IsTrue(error);

delete newList;

}

TEST\_METHOD(fanoCodingDecoding)

{

string text , codedText, decodedText;

text = "wwwcom";

list\* newList = new list;

codedText = newList->fanoCoding(text);

decodedText = newList->fanoDecoding(codedText);

assert.AreEqual(text, decodedText);

delete newList;

}

TEST\_METHOD(check)

{

char letter = 'w';

list\* newList = new list;

newList->check(letter);

assert.AreEqual(1, newList->frequencySearch(letter));

delete newList;

}

TEST\_METHOD(input)

{

string text = "wwwcom";

list\* newList = new list;

newList->input(text);

assert.AreEqual(3, newList->frequencySearch('w'));

delete newList;

}

};

}

# Вывод

Приобретены навыки кодирования и декодирования строк с помощью алгоритма Шеннона-Фано.