МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Кодирование и декодирование.

Студент гр. 9381 _____ Семенов А. Н.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2020

Цель работы.

Ознакомление с алгоритмами кодирования и декодирования сообщений, позволяющими как можно больше уменьшить объем памяти закодированных сообщений. Приобретение навыков в использовании динамических структур данных, приложенных и применяемых в процессе кодирования и декодирования.

Задание.

Вариант 5

На вход подается файл с незакодированным содержимым. Требуется закодировать содержимое файла при помощи алгоритма динамического кодирования Хаффмана.

Основные теоретические положения.

Кодирование – это процесс перевода информации из исходной в удобную форму для ее передачи, обработки и хранения при помощи некоторого алгоритма.

Декодирование (обратный к кодированию процесс) – процесс перевода информации из удобной для ее передачи, обработки и хранения формы в исходную при помощи некоторого алгоритма.

Если все символы исходной информации кодируются кодами одинаковой длины, то кодирование называется равномерным. Если же используются коды разной длины, то кодирование называется неравномерным. В этом случае более часто встречающиеся символы обычно кодируют кодами меньшей длины, а для более редких – кодами большей длины. Таким образом удается значительно снизить объем памяти для хранения длинных закодированных сообщений по сравнению с сообщениями, закодированными равномерным кодом.

Для неравномерного кодирования используется условие Фано: никакой код не должен быть началом другого кода. Если выполняется это условие, то гарантируется, что закодированное сообщение можно однозначно декодировать с начала.

Динамическое кодирование по Хаффману представляет собой однопроходный метод неравномерного кодирования, который производит

кодирование каждого нового символа, встречаясь с ним в первый раз и не имея представления о последующих символах сообщения. При этом на каждом шаге такого кодирования строится и перестраивается дерево Хаффмана в зависимости от текущих частот встреч тех или иных символов, а также производится уравновешивание дерева таким образом, чтобы самые часто встреченные к определенному моменту символы имели минимальную длину кода.

Главные преимущества такого алгоритма кодирования:

- Однопроходность (перед непосредственным кодированием не нужно делать предварительный проход по сообщению в целях подсчета весов всех символов);
- Отсутствие необходимости вместе с последовательностью закодированных сообщений передавать кодовое дерево;
- Возможность кодирования прямо во время создания сообщения или дополнения сообщения с конца во время кодирования;

Недостатки алгоритма Хаффмана:

- Возможность переполнения весов узлов дерева;
- Возможность длинны кода превзойти размеры целочисленных типов;
- Плохо работает на сообщениях с минимальным количеством повторяющихся символов (в таких случаях возможны даже увеличения длин закодированных сообщений по сравнению с равномерным кодом).

Описание алгоритма.

В данном алгоритме используется структура данных – смешанное бинарное дерево – гибрид бинарного дерева и комбинации. Узлы этого дерева хранят целые числа – веса, а листья помимо весов еще и символьную информацию.

Данное дерево строится по следующему принципу: вес каждого узла, не являющегося листом равен сумме весов его сыновей; вес листа определяется текущей частотой встречи символа, ему принадлежащего. При этом после всех

преобразований шага алгоритма, дерево должно приобрести упорядоченность, то есть при обходе его по уровням от листов к корню, слева направо в рамках каждого уровня, веса узлов должны быть выстроены строго по не убыванию.

Для получения двоичного кода каждого узла дерева Хаффмана, производится построение пути от корня к узлу с добавлением к результату «1» при переходе к правому сыну, и «0» при переходе к левому.

Также в этом алгоритме используется особый лист со специальным символом ‘\0’, который обозначает любой еще не переданный кодировщику символ из алфавита. Из данного узла «растут» новые, ранее не встречавшиеся, символы. Вес такого узла – 0, так как этот мнимый символ предполагается еще ни разу не встретившимся.

Кодер и декодер изначально не знают сообщения, а знают лишь только алфавит, из которого оно состоит. Они начинают работу только с корневого узла, который имеет максимальный вес. В начале это и есть особый узел.

Также перед началом непосредственного кодирования производится генерация специального неравномерного двоичного кода для каждого символа алфавита по правилам, обеспечивающим минимальные длины этих кодов и условие ФАНО и зависящим от количества символов в алфавите. Эта специальная кодировка нужна для кодирования символов, встречающихся в сообщении впервые.

Непосредственный алгоритм кодирования:

На каждом шаге поступает очередной символ сообщения. Вначале проверяется его наличие в дереве, то есть был ли он уже встречен кодировщиком ранее.

Если символ встречен впервые, то к результату добавляется сначала двоичный код особого нулевого узла в дереве, а затем специальный код нового символа из алфавита. Затем к особому узлу добавляются два дочерних узла: левый сын – новый особый узел с нулевым весом, правый сын – узел с новым символом, вес которого по понятным причинам становится единичным.

Если же символ встречен не впервые, то есть он есть в дереве, то к результату добавляется только двоичный код узла в дереве с данным символом, а

его вес инкрементируется.

После, необходимо пересчитать веса модифицированного кодированием дерева и проверить его на предмет упорядоченности. Так как веса некоторых узлов в результате кодирования были изменены, производится пересчет всех узлов, каждый из которых принимает значение суммы весов его сыновей.

Далее создается последовательность горизонтального обхода дерева Хаффмана с помощью структуры данных – очереди. Эта последовательность обходится в обратном порядке (от листов к корню), и проверяется упорядоченность узлов данной последовательности по не убыванию их весов.

В случае нарушения упорядоченности производится перевес ветви дерева с коренным узлом, на котором обнаружилось нарушение, с веткой последнего, следующего за ним коренного узла, вес которого окажется пока еще меньше первого узла.

После этого в связи с перевешиванием ветвей, могла измениться упорядоченность весов, по причине чего производится повторный пересчет весов и проверка дерева на упорядоченность.

Таким образом, за конечное число перевешиваний обеспечивается полностью упорядоченное дерево Хаффмана.

Описание структур данных и функций.

1. class HaffNode (элемент (узел) дерева Хаффмана).

Класс узла дерева Хаффмана. Обеспечивает все необходимые для кодирования и декодирования операции над ним и веткой в целом, корневым узлом которой и является.

Имеет следующие поля:

`unsigned int weight` – вес узла дерева

`HaffNode *left` – указатель на левого сына узла дерева

`HaffNode *right` – указатель на правого сына узла дерева

`string wayCode` – строка с двоичным кодом узла дерева

`char *symbol` – указатель на символ, принадлежащий узлу дерева (в случае если узел не является листом и не имеет символьной

информации, этот указатель равен *nullptr*).

Описание методов структуры данных:

`HaffNode(char* symbol = nullptr, string wayCode)` – конструктор узла дерева, принимающий указатель на символ и двоичный код и заполняющий ими соответствующие поля. Адреса левого и правого сыновей обнуляются, а вес становится единичным только в случае передачи ненулевого символа. В остальных случаях вес обнуляется.

`~HaffNode()` – деструктор узла дерева, производящий зачистку памяти под правого и левого сыновей, если они у него имеются.

`HaffNode* extend(char *newSymbol)` – метод расширения дерева путем добавления нового узла с еще не встретившимся символом. Применяется исключительно к особому листу с нулевым символом. Принимает адрес нового символа в алфавите и создает правого сына, передавая в его конструктор этот адрес в качестве аргумента, обеспечивая его таким образом новым символом. Создает левого сына, передавая в его конструктор собственный адрес символа, делая его таким образом новым особым. Возвращает адрес левого сына – нового особого листа дерева.

`void recount(string wayCode = "")` – рекурсивный метод, производящий обход по всем потомкам текущего узла и пересчитывающий их веса таким образом, чтобы каждый узел, не являющийся листом имел вес, равный сумме весов его сыновей. Кроме того, данная функция, производя обход и прокладывая пути, пересчитывает также двоичный код каждого узла ветки с текущем коренным узлом, так как в процессе перевешивания ветвей дерева могли сбиться двоичные кода его узлов.

`HaffNode *findChar(char character)` – рекурсивный метод, производящий поиск заданного символа в ветви дерева Хаффмана с текущим коренным узлом, обходя ветвь КЛП-обходом. Возвращает адрес узла дерева с символом, переданным методу в качестве аргумента в случае успеха, и *nullptr* в случае отсутствия узла с данным символом в дереве.

`HaffNode& operator ++ (int)` – перегруженный постфиксный оператор

инкремента для узла дерева Хаффмана, производящий инкремент его веса.

`unsigned int getWeight() const` – константный метод геттер, возвращающий вес данного узла дерева Хаффмана.

`HaffNode *getLeft()` – метод геттер, возвращающий адрес левого сына данного узла дерева Хаффмана.

`HaffNode *getRight()` – метод геттер, возвращающий адрес правого сына данного узла дерева Хаффмана.

`string getCode()`– метод геттер, возвращающий двоичный код данного узла дерева Хаффмана.

Описание функций над элементами структуры данных, объявленных к ней дружественными:

`friend void swapNode(HaffNode *node_1, HaffNode *node_2)` – функция, производящая обмен данными двух узлов дерева, что осуществляет перевес двух его ветвей с коренными узлами, переданными функции в качестве аргументов. С помощью метода стандартной библиотеки std::swap() производится обмен значений всех полей этих узлов.

`friend HaffNode *paveWay(HaffNode *curElem, string code)` – функция, производящая прокладку маршрута по дереву, начиная от коренного узла, переданного в качестве аргумента `curElem`. Путь прокладывается в цикле по битам двоичного кода `code`, переданного в качестве аргумента. Функция возвращает адрес узла, до которого проложился путь. Если узлы закончились, а биты еще не исчерпаны, функция дополняет дерево узлами. В случае встречи в строке бит инородного символа (не являющегося 0 или 1), выбрасывается исключение с соответствующим сообщением.

`friend HaffNode *paveWay(HaffNode *curElem, char code)` – перегруженная функция, схожая по функционалу с предыдущей с единственным отличием: в качестве аргумента двоичного кода принимает лишь только 1 бит, и прокладывает всего одну дорогу.

`bool operator > (const HaffNode& obj1, const HaffNode& obj2)` –

перегруженный оператор сравнения двух узлов дерева Хаффмана. Возвращает результат сравнения весов переданных в качестве аргументов узлов.

`bool operator < (const HaffNode& obj1, const HaffNode& obj2)` – перегруженный оператор сравнения двух узлов дерева Хаффмана. Возвращает результат сравнения весов переданных в качестве аргументов узлов.

2. class HaffCoder (кодировщик по Хаффману).

Класс – кодировщик, производящий кодирование и декодирование сообщений заданного алфавита по дереву Хаффмана.

Имеет следующие поля:

`HaffNode *root` – указатель на корень дерева Хаффмана

`HaffNode *emptyElem` – указатель на особый узел дерева Хаффмана

`char nulc` – нулевой символ, хранящийся в классе

`char *alphabet` – адрес алфавита кодировщика

`string *codes` – массив специальных кодов алфавита

`Interdata& interdata` – ссылка на объект вывода промежуточных данных.

Описание методов класса:

`HaffCoder(char *alphabet, InterData& interdata)`– конструктор класса. Принимает адрес алфавита и объект вывода промежуточных данных по ссылке и заполняет ими соответствующие поля. С помощью вызова метода getCodes() генерирует массив специальных кодов алфавита. Далее в объект промежуточных данных производится вывод алфавита и специальных кодов, соответствующих каждому его символу. Создается корень дерева Хаффмана – особый узел, в конструктор которого передается адрес нулевого символа, хранящегося в объекте-кодировщике. В поля `root` и `emptyElem` записывается адрес корня.

`~HaffCoder()`– деструктор класса, производящий зачистку памяти под корень дерева Хаффмана и массив специальных кодов алфавита.

`void getCodes()` – метод, производящий генерацию специальных неравномерных двоичных кодов к каждому символу алфавита, адрес которого записан в поле класса, по правилам, обеспечивающим минимальные длины этих кодов и условие ФАНО и зависящим от количества символов в алфавите. Специальные кода записываются в массив, каждый элемент которого соответствует своему элементу массива алфавита. Адрес массива специальной кодировки записывается в поле `codes` класса.

`string encode(char character)` – метод, принимающий символ *character* и производящий его кодирование. Возвращает строку с закодированной последовательностью бит.

Для начала производится проверка наличия данного символа в алфавите. При его отсутствии в алфавите кодировщика выбрасывается исключение с соответствующим сообщением.

Далее производится проверка на наличие символа в дереве Хаффмана с помощью вызова метода *findChar()* у корня дерева. Если символ в дереве есть, то вес его узла инкрементируется, а его код добавляется к результату. В ином случае к результату добавляется сумма кода особого узла и специального кода данного символа. После чего, с помощью вызова метода *extend()* у особого узла, производится расширение дерева и добавление нового листа с текущим символом. Метод возвращает адрес нового особого узла дерева, который записывается в соответствующее поле *emtyElem* класса.

Наконец, с помощью вызова метода *checkTree()* производится пересчет весов дерева Хаффмана, его проверка на упорядоченность и перевешивание ветвей в случае ее отсутствия.

`void checkTree()`– метод, производящий пересчет весов дерева Хаффмана, его проверка на упорядоченность и перевешивание ветвей в случае ее отсутствия.

Для начала с помощью вызова метода *recount()* у корня дерева, производится пересчет его весов.

Далее создаются пустые очередь и последовательность над адресами узлов дерева. С помощью очереди производится горизонтальный обход дерева и

добавление каждого его узла в таком порядке в последовательность.

Затем запускается цикл, пробегающийся по последовательности в обратном порядке и проверяющий ее на неубывающий порядок весов узлов. Как только вес очередного узла оказался меньше веса предыдущего, адрес данного узла записывается в переменную *branch_1*, и запускается цикл, пробегающийся от следующего за элементом *branch_1*: как только встретившийся узел окажется по весу не меньше *branch_1*, в *branch_2* записывается адрес предыдущего от последнего встретившегося узла. С помощью вызова функции *swapNode()* обозначенные ветви дерева меняются местами.

После чего метод *checkTree()* вызывается рекурсивно для повторного пересчета весов дерева после перевешивания и проверки модифицированного дерева на упорядоченность.

3. class InterData (класс вывода промежуточных данных).

Класс – производящий вывод промежуточных данных в файл или на консоль, а также отрисовку дерева Хаффмана.

Имеет следующие поля:

`ofstream *fout` – указатель на объект потока вывода

Описание методов класса:

`InterData(ofstream *fout)` – конструктор класса. Принимает адрес объекта потока вывода и заполняет им соответствующее поле класса.

`friend InterData& operator<<(InterData& interdata, T obj)` – перегруженный оператор вывода в поток. Принимает объект класса InterData и шаблонный тип.

Производит вывод *obj* в поток, если его адрес *fout* не нулевой. Иначе вывод производится на консоль.

Возвращает объект *interdata*.

`void drowTree(HaffNode *root)` – метод, производящий отрисовку дерева с корнем *root* в консоли или файле.

Для начала создается пустая последовательность над адресами узлов

дерева Хаффмана. Эта последовательность заполняется адресами узлов дерева Хаффмана в порядке ЛКП-обхода с помощью вызова функции LKRdetour(). Данная функция также измеряет и возвращает высоту дерева.

Затем запускается цикл, пробегающийся по уровням дерева до последнего уровня с номером, равным высоте дерева. На каждом уровне производится обход последовательности узлов в цикле и запись в строчку каждого узла дерева в ЛКП-порядке в формате: (w[c]), где w – вес узла, а c – его символ, если имеется. Важно отметить, что запись производится с помощью функции specPrint(), в которую также передается булевская переменная, хранящая информацию о том, производить ли запись. Дело в том, что запись производится только в случае, если очередной узел последовательности принадлежит уровню, запись которого в данный момент осуществляется. В ином случае вместо каждого символа строкового представления узла записывается пробел для выравнивания дерева по ширине.

`unsigned int LKRdetour(HaffNode *curElem, vector<HaffNode*>& nodeSequence)` – рекурсивная функция, производящая ЛКП-обход по дереву с адресом коренного узла `curElem` и добавляющая адрес каждого элемента обхода в последовательность `nodeSequence`. Во время обхода функция также подсчитывает высоту дерева и ее возвращает.

`void specPrint(InterData& interdata, string str, bool isWrite)` – функция, производящая печать строки `str` в поток вывода промежуточных данных `interdata`, в случае, если `isWrite = True`, иначе измеряется длина строки, вместо которой в поток записываются подряд пробелы в количестве символов данной строки, дабы сделать ее невидимой.

`void specPrint(InterData& interdata, char c, bool isWrite)` – функция, производящая печать символа `c` в поток вывода промежуточных данных `interdata`, в случае, если `isWrite = True`, вместо него производится печать пробела, дабы сделать символ невидимым.

4. `int main(int argc, char* argv[])` – главная функция, выполняющая программу. Она принимает массив аргументов командной строки

и их количество. Если аргументов не имеется, считывание исходного незакодированного сообщения, печать промежуточных данных и результата – закодированного сообщения производятся на консоль. Если аргумент один, то он воспринимается как имя файла, в который будут выведены промежуточные данные. Считывание сообщения и запись результата в этом случае будут производится в консоли. В случае же трех аргументов, первый из них воспринимается как название файла со входным незакодированным сообщением, второй – для вывода промежуточных данных, а третий – для вывода закодированного сообщения. В случае другого количества аргументов программа завершится с соответствующей ошибкой и сообщением. Также в функции имеется проверка на корректность открытия файлов. В случае неудачного открытия файла с тем или иным названием, программа завершается с соответствующей ошибкой и сообщением.

Создается объект вывода промежуточных данных, в конструктор которого отправляется адрес потока вывода промежуточных данных.

Объявляется алфавит символов в виде строкового литерала.

Создается объект-кодировщик, в конструктор которого передаются адрес алфавита и объект вывода промежуточных данных по ссылке.

Объявляются строки, одна из которых – исходное сообщение: *message*, а другая – закодированное сообщение: *bitMessage*.

Производится считывание сообщения с потока и записи его в *message*. Далее запускается цикл, в каждой итерации которого у кодировщика вызывается метод *encode()*, производящий кодирование очередного символа сообщения и возвращающий строку закодированных битов, которая добавляется к результату *bitMessage*.

В случае исключительной ситуации, объект исключения отлавливается и его сообщение записывается на экран консоли.

Наконец, результат записывается в поток вывода.

В завершение программы файловые потоки ввода, вывода промежуточных данных и результата у соответствующих объектов закрываются.

Тестирование.

Основной тест – 1:

Входные данные: *mama Anna)*

Промежуточные данные:

Алфавит и сгенерированные специальные кода каждого символа:

a - 0000000

b - 0000001

c - 0000010

d - 0000011

e - 0000100

f - 0000101

g - 0000110

h - 0000111

i - 0001000

j - 0001001

k - 0001010

l - 0001011

m - 000110

n - 000111

o - 001000

p - 001001

q - 001010

r - 001011

s - 001100

t - 001101

u - 001110

v - 001111

w - 010000

x - 010001

y - 010010

z - 010011

A - 010100

B - 010101

C - 010110

D - 010111

E - 011000

F - 011001

G - 011010

H - 011011

I - 011100

J - 011101

K - 011110

L - 011111

M - 100000

N - 100001

O - 100010

P - 100011

Q - 100100

R - 100101

S - 100110

T - 100111

U - 101000

V - 101001

W - 101010

X - 101011

Y - 101100

Z - 101101

0 - 101110

1 - 101111

2 - 110000

3 - 110001

4 - 110010

5 - 110011

6 - 110100

7 - 110101

8 - 110110

9 - 110111

. - 111000

, - 111001

! - 111010

? - 111011

: - 111100

(- 111101

) - 111110

* - 111111*

Промежуточные данные:

На вход кодировщику поступает символ 'm'

Текущее дерево кодировщика:

(0[\0])

Символ 'm' еще не встречался в кодируемом сообщении =>

1) Берется код пустого символа:

2) К нему добавляется спец-код символа 'm' из алфавита: 000110

3) К узлу с пустым символом добавляются два новых узла:

Левый сын - новый пустой узел, а правый - узел с новым добавленным символом: 'm':

```
       (0)
(0[\0])   (1[m])
```

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

```
       (1)
(0[\0])   (1[m])
```

Результат кодирования символа 'm' - 000110

На вход кодировщику поступает символ 'a'

Текущее дерево кодировщика:

```
       (1)
(0[\0])   (1[m])
```

Символ 'a' еще не встречался в кодируемом сообщении =>

1) Берется код пустого символа: 0

2) К нему добавляется спец-код символа 'a' из алфавита: 0000000

3) К узлу с пустым символом добавляются два новых узла:

Левый сын - новый пустой узел, а правый - узел с новым добавленным символом: 'a':

```
              (1)
       (0)         (1[m])
(0[\0])   (1[a])
```

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

```
              (2)
       (1)         (1[m])
```

(0[\0]) (1[a])

Результат кодирования символа 'a' - 00000000

На вход кодировщику поступает символ 'm'

Текущее дерево кодировщика:

(2)

(1) (1[m])

(0[\0]) (1[a])

Символ 'm' уже встречался в кодируемом сообщении и присутствует в дереве => вес узла с этим символом в дереве учеличивается на 1:

(2)

(1) (2[m])

(0[\0]) (1[a])

Код символа 'm', собранный по пути к узлу дерева: 1

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

(3)

(1) (2[m])

(0[\0]) (1[a])

Результат кодирования символа 'm' - 1

На вход кодировщику поступает символ 'a'

Текущее дерево кодировщика:

(3)

(1) (2[m])

(0[\0]) (1[a])

Символ 'a' уже встречался в кодируемом сообщении и присутствует в дереве => вес узла с этим символом в дереве учеличивается на 1:

(3)

(1) (2[m])

(0[\0]) (2[a])

Код символа 'a', собранный по пути к узлу дерева: 01

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

(4)

(2) (2[m])

(0[\0]) (2[a])

Результат кодирования символа 'a' - 01

На вход кодировщику поступает символ ' '

Текущее дерево кодировщика:

```
                (4)
       (2)          (2[m])
(0[\0])   (2[a])
```

Символ ' ' еще не встречался в кодируемом сообщении =>

1) Берется код пустого символа: 00

2) К нему добавляется спец-код символа ' ' из алфавита: 111111

3) К узлу с пустым символом добавляются два новых узла:

Левый сын - новый пустой узел, а правый - узел с новым добавленным символом: ' ':

```
                        (4)
                (2)          (2[m])
       (0)          (2[a])
(0[\0])   (1[ ])
```

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

```
                        (5)
                (3)          (2[m])
       (1)          (2[a])
(0[\0])   (1[ ])
```

Требуется перевесить ветки:

```
                (3)
       (1)          (2[a])
(0[\0])   (1[ ])
```

и

(2[m])

Повторный пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с перевешанными ветками и с пересчитанными весами:

```
      (5)
(2[m])                  (3)
                (1)          (2[a])
         (0[\0])   (1[ ])
```

Результат кодирования символа ' ' - 00111111

На вход кодировщику поступает символ 'A'

Текущее дерево кодировщика:

```
      (5)
(2[m])                  (3)
               (1)         (2[a])
         (0[\0])   (1[ ])
```

Символ 'A' еще не встречался в кодируемом сообщении =>

1) Берется код пустого символа: 100

2) К нему добавляется спец-код символа 'A' из алфавита: 010100

3) К узлу с пустым символом добавляются два новых узла:

Левый сын - новый пустой узел, а правый - узел с новым добавленным символом: 'A':

```
      (5)
(2[m])                           (3)
                         (1)         (2[a])
               (0)         (1[ ])
         (0[\0])   (1[A])
```

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

```
      (6)
(2[m])                           (4)
                         (2)         (2[a])
               (1)         (1[ ])
         (0[\0])   (1[A])
```

Результат кодирования символа 'A' - 100010100

На вход кодировщику поступает символ 'n'

Текущее дерево кодировщика:

```
      (6)
(2[m])                           (4)
                         (2)         (2[a])
               (1)         (1[ ])
         (0[\0])   (1[A])
```

Символ 'n' еще не встречался в кодируемом сообщении =>

1) Берется код пустого символа: 1000

2) К нему добавляется спец-код символа 'n' из алфавита: 000111

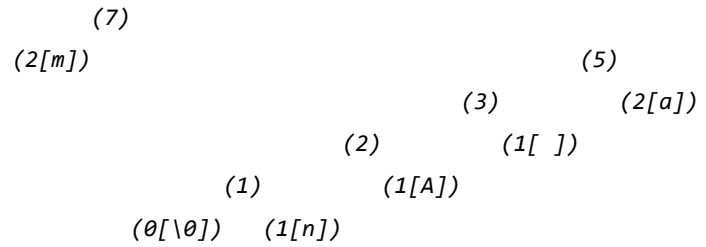
3) К узлу с пустым символом добавляются два новых узла:

Левый сын - новый пустой узел, а правый - узел с новым добавленным символом: 'n':

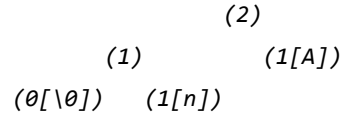
```
      (6)
(2[m])                                 (4)
                                  (2)         (2[a])
                         (1)         (1[ ])
               (0)         (1[A])
         (0[\0])   (1[n])
```

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:



Требуется перевесить ветки:

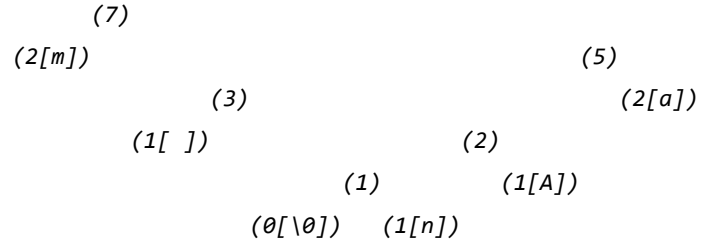


и

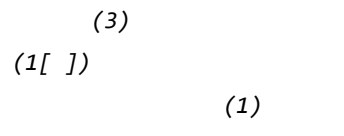
(1[ ])

Повторный пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с перевешанными ветками и с пересчитанными весами:



Требуется перевесить ветки:

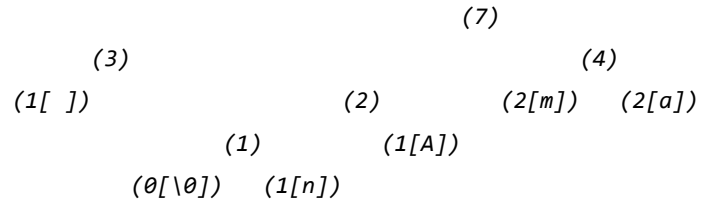


и

(2[m])

Повторный пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с перевешанными ветками и с пересчитанными весами:



Результат кодирования символа 'n' - 1000000111

```
--------------------------------
--------------------------------
На вход кодировщику поступает символ 'n'
Текущее дерево кодировщика:
                                (7)
      (3)                                (4)
(1[ ])                  (2)         (2[m])   (2[a])
                (1)         (1[A])
         (0[\0])   (1[n])

Символ 'n' уже встречался в кодируемом сообщении и присутствует в дереве => вес узла с этим символом
в дереве учеличивается на 1:
                                (7)
      (3)                                (4)
(1[ ])                  (2)         (2[m])   (2[a])
                (1)         (1[A])
         (0[\0])   (2[n])

Код символа 'n', собранный по пути к узлу дерева: 0101

Производится пересчет всех весов и перевешивание дерева в случае необходимости:
Дерево с пересчитанными весами:
                                (8)
      (4)                                (4)
(1[ ])                  (3)         (2[m])   (2[a])
                (2)         (1[A])
         (0[\0])   (2[n])

Требуется перевесить ветки:
       (2)
(0[\0])   (2[n])

и

(1[ ])

Повторный пересчет всех весов и перевешивание дерева в случае необходимости:
Дерево с перевешанными ветками и с пересчитанными весами:
                                (8)
                (4)                        (4)
       (2)              (2)         (2[m])   (2[a])
(0[\0])   (2[n])   (1[ ])   (1[A])

Требуется перевесить ветки:
(2[n])

и
```

(1[A])

Повторный пересчет всех весов и перевешивание дерева в случае необходимости:
Дерево с перевешанными ветками и с пересчитанными весами:

```
                                (8)
               (4)                          (4)
       (1)              (3)         (2[m])   (2[a])
(0[\0])   (1[A])   (1[ ])   (2[n])
```

Требуется перевесить ветки:
(2[n])

и

```
       (1)
(0[\0])   (1[A])
```

Повторный пересчет всех весов и перевешивание дерева в случае необходимости:
Дерево с перевешанными ветками и с пересчитанными весами:

```
                                (8)
      (4)                                (4)
(2[n])         (2)                  (2[m])   (2[a])
         (1[ ])          (1)
                  (0[\0])   (1[A])
```

Результат кодирования символа 'n' - 0101

На вход кодировщику поступает символ 'a'
Текущее дерево кодировщика:

```
                                (8)
      (4)                                (4)
(2[n])         (2)                  (2[m])   (2[a])
         (1[ ])          (1)
                  (0[\0])   (1[A])
```

Символ 'a' уже встречался в кодируемом сообщении и присутствует в дереве => вес узла с этим символом в дереве учеличивается на 1:

```
                                (8)
      (4)                                (4)
(2[n])         (2)                  (2[m])   (3[a])
         (1[ ])          (1)
                  (0[\0])   (1[A])
```

Код символа 'a', собранный по пути к узлу дерева: 11

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

```
                                (9)
      (4)                                (5)
(2[n])         (2)                 (2[m])   (3[a])
         (1[ ])          (1)
                  (0[\0])   (1[A])
```

Результат кодирования символа 'a' - 11

На вход кодировщику поступает символ ')'

Текущее дерево кодировщика:

```
                                (9)
      (4)                                (5)
(2[n])         (2)                 (2[m])   (3[a])
         (1[ ])          (1)
                  (0[\0])   (1[A])
```

Символ ')' еще не встречался в кодируемом сообщении =>

1) Берется код пустого символа: 0110

2) К нему добавляется спец-код символа ')' из алфавита: 111110

3) К узлу с пустым символом добавляются два новых узла:

Левый сын - новый пустой узел, а правый - узел с новым добавленным символом: ')':

```
                                        (9)
      (4)                                       (5)
(2[n])         (2)                          (2[m])   (3[a])
         (1[ ])                  (1)
                          (0)         (1[A])
                  (0[\0])   (1[])]
```

Производится пересчет всех весов и перевешивание дерева в случае необходимости:

Дерево с пересчитанными весами:

```
                                        (10)
      (5)                                        (5)
(2[n])         (3)                          (2[m])   (3[a])
         (1[ ])                  (2)
                          (1)         (1[A])
                  (0[\0])   (1[])]
```

Требуется перевесить ветки:

```
      (3)
(1[ ])                  (2)
               (1)         (1[A])
         (0[\0])   (1[])]
```

и

```
(2[m])

Повторный пересчет всех весов и перевешивание дерева в случае необходимости:
Дерево с перевешанными ветками и с пересчитанными весами:
               (10)
      (4)                                          (6)
(2[n])   (2[m])          (3)                          (3[a])
                  (1[ ])                  (2)
                                    (1)         (1[A])
                             (0[\0])   (1[)])

Результат кодирования символа ')' - 0110111110
--------------------------------
```

Выходные данные:

0001100000000010100111111100010100100000011101011101101111110

Дополнительное тестирование:

Номер теста	Входные данные	Выходные данные
2	abrakadabra!	0000000000000010000101101000001010011000000011011011001000111010
3	rokoko barokko	0010110001000000001010111010100111111100000000011111000000000001011010111
4	Hello, my name is Sahsa!	0110110000010000000101110111000100011001110011000111111010000011000000100100011111000000111101000000001111111111010000000100011110000011001110001001100011110100000011111101111001011100111010
5	Happy new year!!!	0110110000000000001001101110010010110011111110000001110100000010000000100000011000111100011100001011010001110100010100
6	abcdefgh	0000000000000001000000010100000001100000000100110000001011000000011001000000111
7	tgUFYuRRtysash	0011010000011000101000100011001000101100110000111010001001010101110110001001011000001100101000000000100110000000000111
8	The new 2021 year!!??	1001110000011100000010010011111100000011101110001000011101001100001110010111011011100010111100100000100101110010000000000100000101111010011101000001101001110111111001
9	#^%!)(*&^	Ошибка кодировщика: Символ: '#' не распознан кодировщиком (не принадлежит его алфавиту)

10	1) Having written out all the words (words), I started to learn them. 2) Having: bought food, they left supermarket. 3) Barking dog doesnt bite?	1011110111110001111111000110110000000000011000 0111110000001000010000011100000000110100111 00 0100001010000101100110000001101010010000000 01 0011111100000000100011100000111000111000001 01 1000001011011101110011110110000011111011000 10 1110111101100101000000011000000001100111110 11 0011110100110010000111011111010100000111100 01 1100111010110000111001101110001011001101001 01 1100000111101001001111100000100011101010100 11 1011101100100001001000001101101010011100011 10 0110001100001000111111001001010100101000011 01 1101110001010011110011101111110000000010110 001 1000000000000001111110110000000101011010001 01 0101001011100110010011011010000010010111010 01 0111101110100111110001110101101110000010011 00 1111101011101000101011010000000010101001001 10 0100011101101000110001000011111000000000101 010 0010111100000010000001011100101111101110011 0101 0111110100100110001001110100001111001001010 11 000110001100100111011
11	1357828099217581956471939576800 1	1011110110001001100111001101010001101101100 11 0000101100010111000001101111110100111100000 10 1001001100101001011010010010011001001111111 00 010110101011110010101100111111110

Вывод.

В ходе лабораторной работы было проведено ознакомление с алгоритмами кодирования и декодирования сообщений, позволяющими как можно больше уменьшить объем памяти закодированных сообщений; был реализован алгоритм динамического кодирования по Хаффману.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

HaffNode.h:

```
#ifndef HNODE
#define HNODE

#include "IntermediaData.h"

#include <string>
using namespace std;

class HaffNode{
    unsigned int weight;
    HaffNode *left;
    HaffNode *right;
    string wayCode;
public:
    char *symbol;
    HaffNode(char* symbol = nullptr, string wayCode = "");
    ~HaffNode();
    HaffNode *extend(char *newSymbol);
    void recount(string wayCode = "");
    HaffNode *findChar(char character);
    HaffNode& operator ++ (int);
    unsigned int getWeight() const;
    HaffNode *getLeft();
    HaffNode *getRight();
    string getCode();
    friend void swapNode(HaffNode *node_1, HaffNode *node_2);
    friend HaffNode *paveWay(HaffNode *curElem, string code);
    friend HaffNode *paveWay(HaffNode *curElem, char code);
};

bool operator > (const HaffNode& obj1, const HaffNode& obj2);
bool operator < (const HaffNode& obj1, const HaffNode& obj2);

void swapNode(HaffNode *node_1, HaffNode *node_2);
```

```
HaffNode *paveWay(HaffNode *curElem, string code);
HaffNode *paveWay(HaffNode *curElem, char code);

#endif
```

HaffNode.cpp:

```
#include "HaffNode.h"

#include <iostream>

HaffNode::HaffNode(char* symbol, string wayCode): left(nullptr), right(nullptr),
symbol(symbol), wayCode(wayCode){
    if (symbol){
        if (*symbol == '\0') weight = 0;
        else weight = 1;
    } else weight = 0;
}

HaffNode::~HaffNode(){
    if (this->left) delete this->left;
    if (this->right) delete this->right;
}

HaffNode* HaffNode::extend(char *newSymbol){
    if (!this->symbol || *this->symbol != '\0') return nullptr;
    this->left = new HaffNode(this->symbol, this->wayCode + "0");
    this->right = new HaffNode(newSymbol, this->wayCode + "1");
    this->symbol = nullptr;
    return this->left;
}

void HaffNode::recount(string wayCode){
    this->wayCode = wayCode;
    if (this->symbol) return;
    this->weight = 0;

    if (this->left){
        this->left->recount(this->wayCode + "0");
        this->weight += this->left->weight;
    }

    if (this->right){
```

```
        this->right->recount(this->wayCode + "1");
        this->weight += this->right->weight;
    }
}

HaffNode* HaffNode::findChar(char character){
    if (symbol && character == *symbol) return this;

    if (this->left){
        HaffNode *resLeft = this->left->findChar(character);
        if (resLeft) return resLeft;
    }

    if (this->right) return this->right->findChar(character);
    return nullptr;
}

HaffNode& HaffNode::operator ++ (int){
    this->weight += 1;
    return *this;
}

bool operator > (const HaffNode& obj1, const HaffNode& obj2){
    return obj1.getWeight() > obj2.getWeight();
}

bool operator < (const HaffNode& obj1, const HaffNode& obj2){
    return obj1.getWeight() < obj2.getWeight();
}

unsigned int HaffNode::getWeight() const{
    return this->weight;
}

HaffNode* HaffNode::getLeft(){
    return this->left;
}

HaffNode* HaffNode::getRight(){
    return this->right;
}

string HaffNode::getCode(){
    return this->wayCode;
```

```
}

void swapNode(HaffNode *node_1, HaffNode *node_2){
    std::swap(node_1->weight, node_2->weight);
    std::swap(node_1->left, node_2->left);
    std::swap(node_1->right, node_2->right);
    std::swap(node_1->symbol, node_2->symbol);
    std::swap(node_1->wayCode, node_2->wayCode);
}

HaffNode *paveWay(HaffNode *curElem, string code){
    if (!curElem) curElem = new HaffNode;
    int len = code.length();
    for (int i = 0; i < len; i++){
        switch (code[i]) {
            case '0':
                if (!curElem->left) curElem->left = new HaffNode(nullptr,
curElem->wayCode + "0");
                curElem = curElem->left;
                break;

            case '1':
                if (!curElem->right) curElem->right = new HaffNode(nullptr,
curElem->wayCode + "1");
                curElem = curElem->right;
                break;
            default:
                throw invalid_argument(string("Закодированное сообщение имеет
неверный формат: встретилось: '") + code[i] + string("'\n"));
        }
    }
    return curElem;
}

HaffNode *paveWay(HaffNode *curElem, char code){
    if (!curElem) curElem = new HaffNode;

    switch (code) {
        case '0':
            if (!curElem->left) curElem->left = new HaffNode(nullptr,
curElem->wayCode + "0");
            curElem = curElem->left;
            break;
```

```
        case '1':
            if (!curElem->right) curElem->right = new HaffNode(nullptr,
curElem->wayCode + "1");
            curElem = curElem->right;
            break;
        default:
            throw invalid_argument(string("Ошибка кодировщика: Закодированное
сообщение имеет неверный формат: встретилось: '") + code + string("'\n"));
    }

    return curElem;
}
```

HaffCoder.h:

```
#ifndef HCODER
#define HCODER

#include "HaffNode.h"

using namespace std;

class HaffCoder {
    HaffNode *root;
    HaffNode *emptyElem;
    char nulc = '\0';
    char *alphabet;
    string *codes;
    InterData& interdata;
    void getCodes();
    void checkTree();
public:
    HaffCoder(char *alphabet, InterData& interdata);
    ~HaffCoder();
    string encode(char character);
    string decode(string bitMessage);
};

#endif
```

HaffCoder.cpp:

```
#include "HaffCoder.h"
```

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <vector>
#include <queue>

HaffCoder::HaffCoder(char *alphabet, InterData& interdata): alphabet(alphabet),
interdata(interdata){
    getCodes();
    interdata << "Алфавит и сгенерированные специальные кода каждого символа:\n";
    for (int i = 0; i < strlen(alphabet); i++) interdata << alphabet[i] << " - "
<< codes[i] << "\n\n";
    root = new HaffNode(&nulc);
    emptyElem = root;
}

HaffCoder::~HaffCoder(){
    delete root;
    delete codes;
}

void HaffCoder::getCodes(){
      size_t count = strlen(alphabet);
    int k = count;
    int p = 0;
    int accum = 1;
    while (k != 1){
        k = k / 2;
        p++;
        accum *= 2;
    }
    int q = count - accum;

    codes = new string[count];
      char *bitstr = new char[p+2];
      for (int i = 1; i <= count; i++){
            if (i >= 1 && i <= 2*q){
            itoa(i - 1, bitstr, 2);
            codes[i - 1] = string(bitstr);
            while (codes[i - 1].length() != p + 1) codes[i - 1] = "0" + codes[i -
1];
        } else {
            itoa(i - q - 1, bitstr, 2);
```

```
            codes[i - 1] = string(bitstr);
            while (codes[i - 1].length() != p) codes[i - 1] = "0" + codes[i - 1];
        }
      }
    delete bitstr;
}

void HaffCoder::checkTree(){
    root->recount();
    interdata << "с пересчитанными весами:\n";
    interdata.drowTree(root);
    vector<HaffNode*> nodeSequence;
    queue<HaffNode*> nodeQueue;
    nodeQueue.push(root);
    while (!nodeQueue.empty()) {
        nodeSequence.push_back(nodeQueue.front());
        nodeQueue.pop();
        if                                    (nodeSequence.back()->getRight())
nodeQueue.push(nodeSequence.back()->getRight());
        if                                     (nodeSequence.back()->getLeft())
nodeQueue.push(nodeSequence.back()->getLeft());
    }

    HaffNode* branch_1;
    HaffNode* branch_2;

    for (int i = nodeSequence.size() - 1; i > 0; i--){
        if (*(nodeSequence[i]) > *(nodeSequence[i - 1])){
            branch_1 = nodeSequence[i];
            int j = i - 1;
            while(*(nodeSequence[j]) < *branch_1) j--;
            branch_2 = nodeSequence[j + 1];

            interdata << "\nТребуется перевесить ветки:\n";
            interdata.drowTree(branch_1);
            interdata << "\nи\n\n";
            interdata.drowTree(branch_2);

            swapNode(branch_1, branch_2);
            nodeSequence.clear();

            interdata << "\nПовторный пересчет всех весов и перевешивание дерева
в случае необходимости:\n";
            interdata << "Дерево с перевешанными ветками и ";
```

```
            checkTree();
            break;
        }
    }
}

string HaffCoder::encode(char character){
    interdata << "---------------------------------\n";
    interdata << "На вход кодировщику поступает символ '" << character << "'\n";
    char *charPtr = strchr(alphabet, character);
    if (!charPtr) throw invalid_argument(string("Ошибка кодировщика: Символ: '")
+ character + string("' не распознан кодировщиком (не принадлежит его
алфавиту)\n"));

    interdata << "Текущее дерево кодировщика:\n";
    interdata.drowTree(root);
    string result;
    HaffNode *curElem = root->findChar(character);

    if (curElem){
        result = ((*curElem)++).getCode();
        interdata << "\nСимвол '" << character << "' уже встречался в кодируемом
сообщении и присутствует в дереве => ";
        interdata << "вес узла с этим символом в дереве учеличивается на 1:\n";
        interdata.drowTree(root);
        interdata << "\nКод символа '" << character << "', собранный по пути к
узлу дерева: " << result << "\n";
    } else {
        interdata << "\nСимвол '" << character << "' еще не встречался в кодируемом
сообщении =>\n";
        interdata << "1) Берется код пустого символа: " << emptyElem->getCode()
<< "\n";
        interdata << "2) К нему добавляется спец-код символа '" << character <<
"' из алфавита: " << codes[charPtr - alphabet] << "\n";
        result = emptyElem->getCode() + codes[charPtr - alphabet];
        emptyElem = emptyElem->extend(charPtr);
        interdata << "3) К узлу с пустым символом добавляются два новых узла:\n";
        interdata << "Левый сын - новый пустой узел, а правый - узел с новым
добавленным символом: '" << character << "':\n";
        interdata.drowTree(root);
    }
    interdata << "\nПроизводится пересчет всех весов и перевешивание дерева в
случае необходимости:\n";
    interdata << "Дерево ";
```

```
    checkTree();

    interdata << "\nРезультат кодирования символа '" << character << "' - " <<
result << "\n";
    interdata << "---------------------------------\n";
    return result;
}

string HaffCoder::decode(string bitMessage){
    string result;

    HaffNode *specTree = new HaffNode;
    int len = strlen(alphabet);
    for (int i = 0; i < len; i++) paveWay(specTree, codes[i])->symbol = alphabet
+ i;

    len = bitMessage.length();
    HaffNode *curElem;
    int i = 0;
    while (i < len) {
        curElem = root;
        while (!curElem->symbol){
            if (i >= len) throw invalid_argument("Ошибка кодировщика:
Закодированное сообщение имеет неверный формат: не хватает бит для полного
декодирования\n");
            curElem = paveWay(curElem, bitMessage[i]);
            i++;
        }

        if (*curElem->symbol == '\0'){
            curElem = specTree;
            while (!curElem->symbol){
                if (i >= len) throw invalid_argument("Ошибка кодировщика:
Закодированное сообщение имеет неверный формат: не хватает бит для полного
декодирования\n");
                curElem = paveWay(curElem, bitMessage[i]);
                i++;
            }
            emptyElem = emptyElem->extend(curElem->symbol);
        } else (*curElem)++;

        result.push_back(*curElem->symbol);
        checkTree();
    }
```

```
    return result;
}
```

IntermediaData.h:

```
#ifndef INTERDATA
#define INTERDATA

#include <iostream>
#include <fstream>
#include <vector>

class HaffCoder;
class HaffNode;

using namespace std;

class InterData{
    ofstream *fout;
public:
    InterData(ofstream *fout);
    template <class T>
    friend InterData& operator<<(InterData& interdata, T obj){
        if (interdata.fout) *interdata.fout << obj;
        else cout << obj;
        return interdata;
    }
    void drowTree(HaffNode *root);
};

unsigned int LKRdetour(HaffNode *curElem, vector<HaffNode*>& nodeSequence);

void specPrint(InterData& interdata, string str, bool isWrite);
void specPrint(InterData& interdata, char c, bool isWrite);

#endif
```

IntermediaData.cpp:

```
#include "IntermediaData.h"

#include "HaffCoder.h"
```

```
#include <string>
#include <Windows.h>

InterData::InterData(ofstream *fout): fout(fout){}

void InterData::drowTree(HaffNode *root){
    vector<HaffNode*> nodeSequence;
    int height = LKRdetour(root, nodeSequence);

    for (int i = 1; i <= height; i++) {
        for (int j = 0; j < nodeSequence.size(); j++){
            bool isWrite = (i == nodeSequence[j]->getCode().length() -
root->getCode().length() + 1); //SetConsoleTextAttribute(hStdOut, (WORD)((0 << 4)
| 0));
            specPrint(*this, '(', isWrite);
            specPrint(*this, to_string(nodeSequence[j]->getWeight()), isWrite);
            if (nodeSequence[j]->symbol) {
                specPrint(*this, '[', isWrite);
                if (*nodeSequence[j]->symbol == '\0') specPrint(*this, "\\0",
isWrite);
                else specPrint(*this, *nodeSequence[j]->symbol, isWrite);
                specPrint(*this, ']', isWrite);
            }
            specPrint(*this, ')', isWrite);
        }
        *this << "\n";
    }
}

unsigned int LKRdetour(HaffNode *curElem, vector<HaffNode*>& nodeSequence){
    if (!curElem) return 0;
    int H1 = LKRdetour(curElem->getLeft(), nodeSequence);
    nodeSequence.push_back(curElem);
    int H2 = LKRdetour(curElem->getRight(), nodeSequence);
    return (H1 > H2 ? H1 : H2) + 1;
}

void specPrint(InterData& interdata, string str, bool isWrite){
    if (isWrite) interdata << str;
    else {
        int len = str.length();
        for (int i = 0; i < len; i++) interdata << ' ';
    }
```

```
}

void specPrint(InterData& interdata, char c, bool isWrite){
    if (isWrite) interdata << c;
    else interdata << ' ';
}
```

main.cpp:

```
#include <iostream>
#include <fstream>
#include <string>

#include "HaffCoder.h"

using namespace std;

int main(int argc, char* argv[]){
    ifstream fin;
    ofstream fout;
    ofstream finterdata;
    ofstream* p_finterdata = nullptr;

    switch (argc) {
        case 1:
            break;
        case 2:
            finterdata.open(argv[1]);
            if (!finterdata){
                cout << "Ошибка открытия файла: " << argv[1] << endl;
                return 1;
            }
            p_finterdata = &finterdata;
            break;
        case 4:
            fin.open(argv[1]);
            if (!fin){
                cout << "Ошибка открытия файла: " << argv[1] << endl;
                return 1;
            }

            finterdata.open(argv[2]);
            if (!finterdata){
```

```
                cout << "Ошибка открытия файла: " << argv[2] << endl;
                return 1;
            }
            p_finterdata = &finterdata;

            fout.open(argv[3]);
            if (!fout){
                cout << "Ошибка открытия файла: " << argv[3] << endl;
                return 1;
            }
            break;
        default:
            cout << "Некорректное число аргументов" << endl;
            return 1;
    }

    InterData interdata(p_finterdata);

    char                                *alphabet                                   =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.,!?:() ";

    HaffCoder coder(alphabet, interdata);
    string message;
    string bitMessage;
    if (argc == 4) getline(fin, message);
    else {
        cout << "Введите сообщение с символами из заданного алфавита: ";
        getline(cin, message);
    }
    size_t len = message.length();
    interdata << "\nПромежуточные данные:\n";
    try { for (int i = 0; i < len; i++) bitMessage += coder.encode(message[i]); }
    catch (invalid_argument ex){ cout << ex.what(); }

    if (argc == 4) fout << bitMessage << "\n";
    else {
        cout << "\nРезультат кодирования:\n";
        cout << bitMessage << "\n";
    }

    fin.close();
    finterdata.close();
    fout.close();
```

```
    return 0;
}
```