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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Кодирование и декодирование.

Студент гр. 9381 _____ Семенов А.Н.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

ЗАДАНИЕ НА
КУРСОВУЮ РАБОТУ

Студент: Семенов А.Н.

Группа: 9381

Тема работы: Динамическое кодирование и декодирование по Хаффману. Демонстрация.

Исходные данные:

Файл с закодированным или незакодированным содержимым

Содержание пояснительной записки:

1. «Содержание»
2. «Введение»
3. «Описание алгоритма и программы»
4. «Тестирование»
5. «Заключение»
6. «Список использованных источников»

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: .12.2020

Дата защиты реферата: .12.2020

Студент Семенов А.Н.

Преподаватель Фирсов М. А.

АННОТАЦИЯ

Курсовая работа представляет собой программу на языке C++, обрабатывающую текстовую информацию, производя динамическое кодирование и декодирование по Хаффману. Исходная текстовая информация представляет собой файл формата .txt. Программа, реализована с помощью Command Line Interface. Данная работа реализована в стиле объектно-ориентированного программирования: все составные части проекта реализованы в отдельных классах, а каждый класс объявлен и определен в соответствующей паре .h и .cpp файлах. Также используются функции стандартной библиотеки языка C++.

SUMMARY

Course work is a program in C ++ that processes textual information, performing dynamic encoding and decoding according to Huffman. The original text information is a .txt file. The program is implemented using the Command Line Interface. This work is implemented in the style of object-oriented programming: all the components of the project are implemented in separate classes, and each class is declared and defined in the corresponding pair of .h and .cpp files. The functions of the C ++ standard library are also used.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

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.

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:

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

Недостатки алгоритма Хаффмана:

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

1. ЗАДАНИЕ

Вариант 5

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

2. 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. Таким образом, порядок неубывания восстанавливается.

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.

Непосредственный алгоритм декодирования:

На каждом шаге поступает очередной бит закодированного сообщения. Сперва каждый такой бит прокладывает путь от корня к листу в дереве Хаффмана.

Как только был встречен лист дерева Хаффмана проверяется его символ. Если символ нулевой и узел особый, то дальнейший поиск символа аналогично производится в дереве специальной кодировки.

Каждый очередной бит прокладывает путь от корня к листу в дереве специальной кодировки. Как только встречен лист, его символ и будет являться очередным раскодированным символом. При этом в дерево Хаффмана добавляется новый узел с этим впервые встретившимся символом.

Если же символ листа дерева Хаффмана не нулевой, то этот символ и будет являться очередным раскодированным символом. Вес узла с этим символом 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. Таким образом, порядок неубывания восстанавливается.

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

3. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ФУНКЦИЙ

3.1 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.

3.2 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.

`string decode(string bitMessage)` – метод, принимающий строку закодированного сообщения *bitMessage* и производящий ее декодирование. Возвращает строку с раскодированным сообщением.

Для начала строится дерево специальной кодировки по всем символам алфавита с помощью прокладывания путей методом *paveWay()*. Далее запускается главный цикл, в каждой итерации которого производится исполнение алгоритма декодирования по Хаффману. Каждый новый декодированный символ добавляется в конец результирующей строки, которую метод и возвращает.

Если исходная строка бит закончилась раньше, чем очередная итерация цикла, код считается некорректным: выбрасывается исключение с соответствующим сообщением.

`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.

3.3 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`, иначе вместо него производится печать пробела, дабы сделать символ невидимым.

3.4 Главная функция.

`int main(int argc, char* argv[])` – главная функция, выполняющая программу. Она принимает массив аргументов командной строки и их количество. Первым аргументом программы является слово, которое задает режим ее работы: encoder – кодирование, decoder – декодирование. Если больше аргументов не имеется, считывание исходного сообщения, печать промежуточных данных и результата производятся в консоли. Если имеется еще один аргумент, то он воспринимается как имя файла со входным незакодированным сообщением. Вывод промежуточных данных и запись результата в этом случае будут производится в консоли. В случае же четырех аргументов, второй из них воспринимается как название файла со входным сообщением, третий – для вывода промежуточных данных, а четвертый – для вывода результата. 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 две строки, строка входных данных *inMessage* и результата *outMessage*.

Производится считывание сообщения с потока и записи его в *inMessage*. 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 *outMessage*.

В случае декодирования в outMessage записывается возвращаемое значения метода *decode()* – результат декодирования.

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.

4. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Запускать программу следует в терминале Windows.

Предварительно следует установить в консоли кодировку Unicode командой: chcp 65001.

Существует три варианта запуска программы:

1) Работа полностью в консоли – тогда программа запускается с единственным аргументом: *encoder* или *decoder*, в зависимости от выбора режима работы программы. Этот аргумент обязателен и всегда является первым при запуске программы;

2) Два аргумента программы – второй аргумент после режима – название файла для считывания исходного сообщения; запись промежуточных данных и результата в этом случае будут производиться в консоли;

3) Четыре аргумента программы – после режима сначала название файла с исходным сообщением, состоящий из символов алфавита в случае кодирования или из последовательности бит (0 или 1) в случае декодирования, затем название файла для записи промежуточных данных, наконец название файла для записи результата.

Название файла - путь до файла относительно директории cw.

Алфавит кодировщика программы:
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 .,!?:() ;

При просмотре промежуточной демонстрации кодирования в txt-файле предварительно следует установить в нем моноширный шрифт 'Consolas' для грамотной отрисовки деревьев.

При встречи нераспознанного символа (не присутствующего в алфавите), кодировщик остановит работу на этом символе.

При встречи нераспознанного бита (не 0 или 1), декодировщик также остановит работу на этом символе.

5. 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:

```
      (7)
(2[m])                                      (5)
                                  (3)         (2[a])
                        (2)         (1[ ])
                (1)         (1[A])
        (0[\0])   (1[n])
```

Требуется перевесить ветки:

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

и

(1[ ])

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

```
      (7)
(2[m])                                      (5)
               (3)                             (2[a])
        (1[ ])                  (2)
                        (1)         (1[A])
                  (0[\0])   (1[n])
```

Требуется перевесить ветки:

```
      (3)
```

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

и

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

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

```
                                  (7)
      (3)                                 (4)
(1[ ])                  (2)         (2[m])   (2[a])
               (1)         (1[A])
         (0[\0])   (1[n])
```

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 кодировщика (таблица 1):

Номер теста	Входные данные	Выходные данные
2	abrakadabra!	0000000000000010000101101000001010011000000011011011001000111010
3	rokoko barokko	0010110001000000001010111010100111111100000000011110000000000101101011
4	Hello, my name is Sahsa!	0110110000010000000101110111000100011001110011000111111010000011000000100100011110000011110100000000011111111110100000001000111100001100111000001001100011111010000001111101111001011001110100
5	Happy new year!!!	0110110000000000001001101110010010110011111110000001110100000010000000100000110001111000111000010110100011101000101000
6	abcdefgh	0000000000000010000000101000000011000000010011000000010110000000011001000000111
7	tgUFYuRRtysash	0011010000011000101000100011001000101100110000111010001001010101110110001001011000001100101000000000100110000000000111
8	The new 2021 year!!??	1001110000011100000010010011111110000001110111000100001110100110000111001011101101110001011110010000010010111001000000000010000010111101001110100000110100111011111001
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?	1011110111110001111111000110110000000001100000111110000001000010000011100000011101001110001000010100001011001100000011010011100010000101000010111011000001110100100000000100111111100000000100011100000111000111000001011000001011011101110011110111011000001111101100010111011111001010000000110000000001100111110110011000101000000111010011001000011101111101010000011110001110011101011000011100110111000101100110100101110000011110100100111110000001001000001101010100111011101100100001001000001101010011100011100110001100001000111111001001010100101000011011101110001010011110111110000000010110001100000

		0000000001111110110000000101011010001010101001011100110010011011010000010010111010010111101110100111110001110101101110000010011001111101011101000101011010000000010101001001100100011101101000110001000011111000000000101010001011110000010000010111001011111011100110101011111010010011000100111010000111100100101011000110001100100111011
11	13578280992175819564719395768001	1011110110001001100111001101010001101101100110000101100010111000001101111110100111100000101001001100101001011010010010011001001111111000101101010111100101011001111111110

Таблица 1

<u>Дополнительное тестирование декодировщика (таблица 2):</u>

Номер теста	Входные данные	Выходные данные
1	0000000000000001000010110100000101001100000001101101100100011101 0	abrakadabra!
2	00101100010000000010101110101001111111000000000111100000000001011010111	rokoko barokko
3	011011000001000000010111011100010001100111001100011111010000011000000100100011110000011110100000000001111111110100000000100011110000110011100000100110001111010000001111101111001011001110 10	Hello, my name is Sahsa!
4	0110110000000000000100110111001001011001111111000000111010000001000000001000000110001111000111000010110100011101000101000	Happy new year!!!
5	000000000000000100000000101000000011000000010011000000010110000000110010000001 11	abcdefgh
6	0011010000011000101000100011001000101100110000111010001001010101110110001001011000001100101000000000100110000000111	tgUFYuRRtysash
7	1001110000011100000001001001 11111	The new 2021 year!!??

	00000011101110001000011101001100 00111001011101101110001011110010 00001001011100100000000001000001 01111010011101000001101001110111 11001	
8	11000111101	Ошибка кодировщика: Закодированное сообщение имеет неверный формат: не хватает бит для полного декодирования
9	10111101100010011001110011010100 01101101100110000101100010111000 00110111111010011110000010100100 11001010010110100100100110010011 11111000101101010111100101011001 11111110	135782809921758195647193957680001

Таблица 2

ЗАКЛЮЧЕНИЕ

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.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Википедия: Адаптивный алгоритм Хаффмана. // wikipedia.org. URL: https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0%D0%BF%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD%D0%B0 (Дата обращения: 25.12.2020).

ПРИЛОЖЕНИЕ А
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; // Конструктор узла
        else weight = 1;
    } else weight = 0;
}

HaffNode::~HaffNode(){ // Деструктор узла и ветви в целом
    if (this->left) delete this->left; // Очистка памяти под левое поддерево
    if (this->right) delete this->right; // Очистка памяти под правое поддерево
}

HaffNode* HaffNode::extend(char *newSymbol){ // Росто дерева путем добавления новых узлов
    if (!this->symbol || *this->symbol != '\0') return nullptr; // Проверка соблюдения инварианта: роста дерева из особого узла
    this->left = new HaffNode(this->symbol, this->wayCode + "0"); // Создание правого сына - нового особого узла
    this->right = new HaffNode(newSymbol, this->wayCode + "1"); // Создание левого сына - узла с новым добавленным символом
    this->symbol = nullptr; // Обнуление символьного значение текущего узла (так как больше не лист)
    return this->left; // Возврат адреса нового особого узла
}

void HaffNode::recount(string wayCode){ // Пересчет весов и двоичных кодов всех потомков узла
    this->wayCode = wayCode; // Пересчет двоичного кода
    if (this->symbol) return;
    this->weight = 0; // Обнуление веса

    if (this->left){
        this->left->recount(this->wayCode + "0"); // Пересчет левого сына
```

```
        this->weight += this->left->weight; // Добавления к весу текущего узла нового веса левого узла
    }

    if (this->right){
        this->right->recount(this->wayCode + "1"); // Пересчет правого сына
        this->weight += this->right->weight; // Добавления к весу текущего узла нового веса правого узла
    }
}

HaffNode* HaffNode::findChar(char character){ // Поиск узла с заданным символом
    if (symbol && character == *symbol) return this; // Возврат адреса текущего узла в случае нахождения

    if (this->left){
        HaffNode *resLeft = this->left->findChar(character); // Осуществление поиска в левом поддереве
        if (resLeft) return resLeft; // Если найден, возврат
    }

    if (this->right) return this->right->findChar(character); // Осуществление поиска в правом поддереве и возврат результата
    return nullptr;
}

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

bool operator > (const HaffNode& obj1, const HaffNode& obj2){
    return obj1.getWeight() > obj2.getWeight(); // Возврат сравнения весов узлов
}

bool operator < (const HaffNode& obj1, const HaffNode& obj2){
    return obj1.getWeight() < obj2.getWeight(); // Возврат сравнения весов узлов
}

unsigned int HaffNode::getWeight() const{
    return this->weight; // Возврат веса узла
}
```

```
HaffNode* HaffNode::getLeft(){
    return this->left; // Возврат адреса левого сына узла
}

HaffNode* HaffNode::getRight(){
    return this->right; // Возврат адреса правого сына узла
}

string HaffNode::getCode(){
    return this->wayCode; // Вовзрат двоичного кода узла
}

void swapNode(HaffNode *node_1, HaffNode *node_2){ // обмен данными всех полей узлов
    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){ // Проложение пути по двоичному коду и возврат найденного по двоичному коду узла
    if (!curElem) curElem = new HaffNode; // Создание нового узла в случае отсутствия дерева
    int len = code.length();
    for (int i = 0; i < len; i++){ // Цикл по каждому биту кода
        switch (code[i]) {
            case '0':
                if (!curElem->left) curElem->left = new HaffNode(nullptr, curElem->wayCode + "0"); // Создание левого сына в случае его отсутствия
                curElem = curElem->left; // Переход к левому сыну
                break;

            case '1':
                if (!curElem->right) curElem->right = new HaffNode(nullptr, curElem->wayCode + "1"); // Создание правого сына в случае его отсутствия
                curElem = curElem->right; // Переход к правому сыну
                break;
            default:
```

```
                throw invalid_argument(string("Закодированное сообщение имеет неверный формат: встретилось: '") + code[i] + string("'\n"));
        }
    }
    return curElem; // Возврат адреса найденного узла
}

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"); // Создание левого сына в случае его отсутствия
            curElem = curElem->left; // Переход к левому сыну
            break;

        case '1':
            if (!curElem->right) curElem->right = new HaffNode(nullptr, curElem->wayCode + "1"); // Создание правого сына в случае его отсутствия
            curElem = curElem->right; // Переход к правому сыну
            break;
        default:
            throw invalid_argument(string("Ошибка кодировщика: Закодированное сообщение имеет неверный формат: встретилось: '") + code + string("'\n"));
    }

    return curElem; // Возврат адреса найденного узла
}
```

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){ // Конструктор
    getCodes(); // Генерация специальных кодов алфавита
    interdata << "Алфавит и сгенерированные специальные кода каждого символа:\n";
    for (int i = 0; i < strlen(alphabet); i++) interdata << alphabet[i] << " - "
<< codes[i] << "\n"; // Вывод алфавита
    interdata << "\n";
    root = new HaffNode(&nulc); // Генерация корня дерева Хаффмана
    emptyElem = root; // Сохранение адреса особого узла
}

HaffCoder::~HaffCoder(){ // Деструктор кодировщика
    delete root; // Очистка памяти под дерево Хаффмана
    delete codes; // Зачистка памяти под массив специальных кодов
}

void HaffCoder::getCodes(){ // Генерация специальных кодов алфавита
```

```
      size_t count = strlen(alphabet);
    int k = count;
    int p = 0; // Примерная длинна кодовых сообщений
    int accum = 1;
    while (k != 1){
        k = k / 2;
        p++;
        accum *= 2;
    }
    int q = count - accum; // Разница между истинным количеством символов и ближайшей степенью двойки

    codes = new string[count]; // Создание массива специальных кодов
      char *bitstr = new char[p+2]; // Выделение памяти под код одного символа
      for (int i = 1; i <= count; i++){ // Цикл по всем символам алфавита
            if (i >= 1 && i <= 2*q){
            itoa(i - 1, bitstr, 2); // Преобразование номера символа в его двоичный код
            codes[i - 1] = string(bitstr);
            while (codes[i - 1].length() != p + 1) codes[i - 1] = "0" + codes[i - 1]; // Дополнение двоичного кода до нужной длинны нулями
        } else {
            itoa(i - q - 1, bitstr, 2); // Преобразование номера символа в его двоичный код
            codes[i - 1] = string(bitstr);
            while (codes[i - 1].length() != p) codes[i - 1] = "0" + codes[i - 1]; // Дополнение двоичного кода до нужной длинны нулями
        }
      }
    delete bitstr;
}

void HaffCoder::checkTree(){ // Проверка дерева Хаффмана на упорядоченность
    root->recount(); // Пересчет весов и двоичных кодов дерева
    interdata << "с пересчитанными весами:\n";
    interdata.drowTree(root);
    vector<HaffNode*> nodeSequence; // Последовательность горизонтального обхода
    queue<HaffNode*> nodeQueue; // Очередь для горизонтального обхода
    nodeQueue.push(root);
    while (!nodeQueue.empty()) { // Горизонтальный обход
        nodeSequence.push_back(nodeQueue.front()); // Образование последовательности горизонтального обхода
```

```
        nodeQueue.pop();
        if                                          (nodeSequence.back()->getRight())
nodeQueue.push(nodeSequence.back()->getRight());
        if                                           (nodeSequence.back()->getLeft())
nodeQueue.push(nodeSequence.back()->getLeft());
    }

    HaffNode* branch_1; // Адреса ветвей, которые требуется перевесить
    HaffNode* branch_2;

    for (int i = nodeSequence.size() - 1; i > 0; i--){ // Проверка на неубывающий
порядок обхода последовательности
        if  (*(nodeSequence[i])  >  *(nodeSequence[i  -  1])){  //  Обнаружение
конфликтного узла
            branch_1 = nodeSequence[i]; // Приготовление его к перевесу
            int j = i - 1;
            while(*(nodeSequence[j]) < *branch_1) j--; // Поиск подходящего места
для первого узла и подходящего второго узла на его места
            branch_2 = nodeSequence[j + 1]; // Приготовление последнего к перевесу

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

            swapNode(branch_1, branch_2); // Перевес узлов
            nodeSequence.clear(); // Очистка последовательности

            interdata << "\nПовторный пересчет всех весов и перевешивание дерева
в случае необходимости:\n";
            interdata << "Дерево с перевешанными ветками и ";
            checkTree(); // Повторная проверка дерева на упорядоченность после
модификации
            break;
        }
    }
}

string HaffCoder::encode(char character){ // Функция кодирования символа
    interdata << "---------------------------------\n";
    interdata << "На вход кодировщику поступает символ '" << character << "'\n";
```

```
    char *charPtr = strchr(alphabet, character); // Проверка наличия символа в алфавите
    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; // Для последовательности бит - результата кодирования
    HaffNode *curElem = root->findChar(character); // Проверка наличия узла с текущим символом в дереве

    if (curElem){ // При наличии
        result = ((*curElem)++).getCode(); // Добавления к пустому результату код найденного узла и инкремент его веса
        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 { // При отсутствии
        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]; // Добавление к пустому результату код пустого узла а затем спец код символа
        emptyElem = emptyElem->extend(charPtr); // Расширение дерева путем добавления новых узлов (с новым символом) к пустому узлу, сохранение нового пустого узла
        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(); // Проверка дерева на упорядоченность
```

```
    interdata << "\nРезультат кодирования символа '" << character << "' - " << result << "\n";
    interdata << "---------------------------------\n";
    return result; // Возврат результата
}

string HaffCoder::decode(string bitMessage){ // Функция декодирования сообщения
    string result; // Создание пустого результата

    interdata << "Начало работы декодировщика: на вход декодеру подается последовательность бит:\n";
    interdata << bitMessage << "\n";

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

    len = bitMessage.length();
    HaffNode *curElem;
    int i = 0;
    while (i < len) { // Главный цикл декодирования
        interdata << "---------------------------------\n";
        interdata << "Текущее дерево кодировщика:\n";
        interdata.drowTree(root);
        interdata << "\nОставшееся нераскодированное сообщение: " << bitMessage.substr(i) << "\n";
        interdata << "\nПо первым битам нераскодированной части сообщения строится путь в дереве Хаффмана до тех пор, пока не встретится лист\n";
        curElem = root; // Начало движения от корня дерева
        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]); // Прокладка маршрута до листа в дереве Хаффмана
            i++;
        }
        interdata << "Следующий набор бит: " << curElem->getCode() << " привел декодера к узлу с символом '";
        if (*curElem->symbol == '\0'){ // Если встретился особый узел
```

```
            interdata << "\\0' - пустым символом =>\n=> по следующим битам нераскодированной части сообщения определяется символ из специальной кодировки\n";
            interdata << "\nОставшееся нераскодированное сообщение: " << bitMessage.substr(i) << "\n";
            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]); // Прокладка маршрута до листа в дереве специальной кодировки
                i++;
            }
            interdata << "Следующий набор бит: " << curElem->getCode() << " привел декодера к узлу с символом '" << *curElem->symbol << "' в дереве специальной кодировки, который и будет очередным результатом декодирования\n";
            emptyElem = emptyElem->extend(curElem->symbol); // Расширение дерева путем добавления новых узлов (с новым символом) к пустому узлу, сохранение нового пустого узла
            interdata << "\nК узлу с пустым символом добавляются два новых узла:\n";
            interdata << "Левый сын - новый пустой узел, а правый - узел с новым добавленным символом: '" << *curElem->symbol << "':\n";
            interdata.drowTree(root);
        } else { // Если встретился узел с уже имеющимся в сообщении символом
            interdata << *curElem->symbol << "', который и будет очередным результатом декодирования\n";
            (*curElem)++; // Инкремент веса встретившегося узла
            interdata << "Вес узла с этим символом в дереве учеличивается на 1:\n";
            interdata.drowTree(root);
        }

        result.push_back(*curElem->symbol); // Добавление символа узла, на котором декодировщик остановился, к результату

        interdata << "\nПроизводится пересчет всех весов и перевешивание дерева в случае необходимости:\n";
        interdata << "Дерево ";
        checkTree(); // Проверка дерева на упорядоченность

        interdata << "\nТекущий результат декодирования сообщения: '" << result << "'\n";
```

```
        interdata << "--------------------------------\n";
    }
    interdata << "Конец работы декодировщика.\n";
    delete specTree; // Зачистка памяти под дерево специальной кодировки
    return result; // Возврат результата
}
```

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){} // Конструктор

void InterData::drowTree(HaffNode *root){ // Функция отрисовки дерева
    vector<HaffNode*> nodeSequence; // Последовательность ЛКП-обхода
    int height = LKRdetour(root, nodeSequence); // ЛКП-обход и измерение высоты

    for (int i = 1; i <= height; i++) { // Цикл по уровням отрисовки
        for (int j = 0; j < nodeSequence.size(); j++){
            bool isWrite = (i == nodeSequence[j]->getCode().length() -
root->getCode().length() + 1); // Определение принадлежности узла к текущему
уровню
            // Печать узла в формате веса и символа, если имеется, видимым, в
случае принадлежности текущему уровню и невидимым, в обратном случае
            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){ //ЛКП-обход и измерение высоты дерева
    if (!curElem) return 0;
    int H1 = LKRdetour(curElem->getLeft(), nodeSequence); // Измерение высоты
левого поддерева
```

```
    nodeSequence.push_back(curElem);
    int H2 = LKRdetour(curElem->getRight(), nodeSequence); // Измерение высоты правого поддерева
    return (H1 > H2 ? H1 : H2) + 1; // Возврат большей из высот с добавлением едицицы - высоты текущего уровня
}

void specPrint(InterData& interdata, string str, bool isWrite){ // Печасть строки видимой или невидимой (пробелы вместо символов) в зависимости от аргумента isWrite
    if (isWrite) interdata << str;
    else {
        int len = str.length();
        for (int i = 0; i < len; i++) interdata << ' '; // Печать пробелов в количестве длинны строки
    }
}

void specPrint(InterData& interdata, char c, bool isWrite){ // Печасть символа видимой или невидимой (пробел вместо символа) в зависимости от аргумента 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; // Поток входных данных
    ofstream fout; // Поток выходных данных
    ofstream finterdata; // Поток промежуточных данных
    ofstream *p_finterdata = nullptr; // Адрес потока промежуточных данных
```

```
    bool decoder; // Использовать ли декодировщик?


    if (argc > 1 && argc < 6){
        if (argv[1] == string("encoder")) decoder =
false; // Определение режима работы
        else if (argv[1] == string("decoder"))
decoder = true;
        else {
            cout << "Нераспознанный режим работы
программы: '" << argv[1] << "'\n";
            return 1;
        }


        if (argc > 2){
            fin.open(argv[2]); // Открытие файла
для чтения исходных данных
            if (!fin){
                cout << "Ошибка открытия файла: "
<< argv[2] << endl;
                return 1;
            }


            if (argc > 3) {
                finterdata.open(argv[3]); //
Открытие файла для записи промежуточных данных
                if (!finterdata){
                    cout << "Ошибка открытия файла:
" << argv[3] << endl;
                    return 1;
                }
                p_finterdata = &finterdata;


                if (argc == 5) {
                    fout.open(argv[4]); // Открытие
файла для записи результата
                    if (!fout){
                        cout << "Ошибка открытия
файла: " << argv[4] << endl;
                        return 1;
                    }
                }
            }
        }
```

```
    } else {
        cout << "Некорректное число аргументов" <<
endl;
        return 1;
    }

    InterData interdata(p_finterdata); // Создание
объекта вывода промежуточных данных

    char *alphabet =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ0123456789.,!?:() "; // Алфавит

    HaffCoder coder(alphabet, interdata); //
Создание объекта-кодировщика
    string inMessage; // Входная строка
    string outMessage; // Выходная результирующая
строка
    if (argc > 2) getline(fin, inMessage); //
Чтение входных данных из потока
    else {
        if (decoder) cout << "Введите
закодированную последовательность бит: ";
        else cout << "Введите сообщение с символами
из заданного алфавита: ";
        getline(cin, inMessage); // Чтение входных
данных с консоли
    }
    size_t len = inMessage.length();
    interdata << "\nПромежуточные данные:\n";
    try {
        if (decoder) outMessage =
coder.decode(inMessage); // Декодирование входной
последовательности бит
        else for (int i = 0; i < len; i++)
outMessage += coder.encode(inMessage[i]); //
Кодирование входного сообщения
    } catch (invalid_argument ex){ cout <<
ex.what(); return 1;} // Обработка исключения
кодировщика

    if (argc == 5) fout << outMessage << "\n"; //
Вывод результата в поток
    else {
        if (decoder) cout << "\nРезультат
декодирования:\n";
```

```
        else cout << "\nРезультат кодирования:\n";
        cout << outMessage << "\n"; // Вывод
результата на консоль
    }
    // Закрытие потоков
    fin.close();
    finterdata.close();
    fout.close();

    return 0;
}
```