

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

КУРСОВАЯ РАБОТА
по дисциплине по дисциплине «Алгоритмы и структуры данных»
Тема: Кодирование и декодирование

Студентка гр. 9382

Круглова В.Д.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студентка Круглова В.Д.

Группа 9382

Тема работы : Кодирование и декодирование.

Исходные данные:

Вариант 6. Динамическое кодирование и декодирование по Хаффману –
текущий контроль.

Текущий контроль.

Содержание пояснительной записки:

«Содержание», «Введение», «Формальная постановка задачи», «Описание алгоритма», «Описание основных структур данных и функций», «Описание интерфейса пользователя», «Тестирование», «Выводы», «Приложение с кодом», «Список использованных источников»

Предполагаемый объем пояснительной записки:

Не менее 25 страниц.

Дата выдачи задания: 01.09.2020

Дата сдачи: 23.12.2020

Дата защиты: 24.12.2020

Студентка _____

Круглова В.Д.

Преподаватель _____

Фирсов М.А.

АННОТАЦИЯ

Данная курсовая работа предоставляет возможность получать задания для проведения текущего контроля по теме кодирования и декодирования по Хаффману. Также в работе осуществлена возможность получать ответы на них для последующей проверки с возможностью прослеживания промежуточных ответов.

SUMMARY

This course work provides an opportunity to receive tasks for conducting current control on the topic of Huffman coding and decoding. Also in the work, it is possible to receive answers to them for subsequent verification with the possibility of tracking intermediate answers.

.

СОДЕРЖАНИЕ

	Введение	5
1.	Постановка задачи	6
2.	Описание алгоритмов	7
2.1	Кодирование методом Фано-Шеннона	7
2.2	Кодирование методом Хаффмана	7
3.	Описание основных функций и структур	9
4.	Описание интерфейса пользователя	12
5.	Тестирование	13
	Заключение	21
	Список использованных источников	21
	Приложение А. Исходный код программы	22

ВВЕДЕНИЕ

Цель работы: создание программы для генерации заданий с ответами к ним для проведения текущего контроля среди студентов.

Хоть метод динамического кодирования по Хаффману и позволяет сократить длину закодированного сообщения, но уступает статической версии по эффективности сжатия.

Учитывая, что перед началом работы метод не получает никаких дополнительных данных об обрабатываемом тексте, для создания возможности декодировать полученную последовательность, в кодере и декодере должны использоваться одинаковые алгоритмы создания и обновления дерева кодов при получении символов.

Теоретические положения: алгоритм Виттера – метод построения дерева по коду Хаффмана, при котором код представляется в виде древовидной структуры, в которой каждый узел имеет собственный вес и уникальный номер (цифры идут сверху вниз и справа налево). Причем веса должны удовлетворять принципу братства. Где вес – это количество символов, встреченных ранее. Набор узлов с одинаковыми весами представляет собой блок.

Кодер и декодер начинают только с корневого узла, который имеет максимальный вес. Коим в начале и является узел NYT.

1. ПОСТАНОВКА ЗАДАЧИ

- 1.** Реализовать программу выполняющую кодирование и декодирование входных данных методом динамического кодирования и декодирования по Хаффману с подробными и понятными пояснениями работы алгоритма и визуализацией структуры данных на разных этапах выполнения алгоритма.
- 2.** Протестировать программу на выборке входных данных с фиксацией результата работы программы.

2. ОПИСАНИЕ АЛГОРИТМОВ

2.1 Динамическое кодирование по Хаффману.

Суть метода заключается в том, чтобы заменить часто встречающиеся символы более короткими кодами, а редко встречающиеся последовательности более длинными, пользуясь бинарным деревом, обновляя его сразу по факту получения очередного символа. Алгоритм основывается на кодах переменной длины.

Первоначально идет считывание символа и получение его кода исходя из положения листа, содержащего такой же символ, либо из положения особого узла NYT. Далее создаются два новых узла NYT, если нашелся символ, встречающийся впервые (левый потомок — NYT, а правый — лист, хранящий этот символ, его вес и номер). Потом увеличивается вес всех узлов, которые вертикально связаны с листом, хранящим встреченный символ, и веса самого этого узла, с одновременной перестройкой дерева кодов. Циклическое повторение выше описанного алгоритма для всех введенных символов.

Итогом работы алгоритма является закодированное по Хаффману сообщение.

2.2 Динамическое декодирование по Хаффману.

Суть алгоритма заключается в создании бинарного дерева кодов по имеющейся кодовой последовательности, при котором происходит получение закодированного символа исходя из его состояния.

Сначала происходит восстановление символа из первых 8 цифр кодовой последовательности, и создание соответствующего кодового дерева. Затем в цикле обрабатываются остальные цифры переданной последовательности, с дальнейшим получением данных о следующем символе и перестраивается дерево. Если код привел к NYT, обрабатываются следующие 8 цифр из кодовой последовательности и создаются два новых узла из NYT (левый потомок — NYT, правый — лист, хранящий этот символ, его вес и номер). Если же остановка произошла на другом узле, происходит получение символа

из хранящегося в узле значения. Потом увеличивается вес всех узлов, вертикально связанных с листом, хранящим встреченный символ, и веса самого этого узла, с одновременной перестройкой кодового дерева. Алгоритм повторяется для оставшейся кодовой последовательности.

Итогом работы алгоритма является декодированное по Хаффману сообщение.

3. ОПИСАНИЕ ОСНОВНЫХ ФУНКЦИЙ И СТРУКТУР

class Node

Назначение: содержит информацию об узле.

Node* m_parent – указатель на предка

Node* m_right – указатель на правого потомка

Node* m_left – указатель на левого потомка

unsigned int m_weight – вес узла

unsigned int m_number – номер узла

char m_symbol – символ, который хранится в листе

bool m_isLeaf – флаг листа (TRUE, если лист)

bool m_isNYT – флаг NYT (TRUE, если NYT)

std::string vitterCoder(char symbol, std::string *step) — управляющая функция динамического кодирования. Получает на вход символ, возвращает закодированную последовательность, соответствующую полученному символу.

Назначение: управляющая функция динамического кодирования.

Описание аргументов: кодируемый символ, указатель на шаг.

Возвращаемое значение: строка по значению.

std::string vitterDecoder(std::string *code)

Назначение: декодирует код в сообщение.

Описание аргументов: указатель на строку с кодом.

Возвращаемое значение: строка с сообщением.

void swapNodes(Node *a, Node *b)

Назначение: меняет 2 узла/листа местами с помощью замены их собственных указателей и указателей их родителей.

Описание аргументов: указатели на узлы a & b.

Возвращаемое значение: функция ничего не возвращает.

`Node* findBlockLeader(std::vector<Node*> *vec, unsigned int weight)`

Назначение: находит лидера блока (лидер блока: тот же вес, тот же тип (лист/вн.узел), максимальный номер).

Описание аргументов: указатель на вектор, хранящий узлы, искомый вес.

Возвращаемое значение: узел – лидер блока.

`void splitHafTree(Node* root, std::vector<Node*> *vec, short int level)`

Назначение: запишет все листы и узлы в вектор по порядку КПЛ.

Описание аргументов: указатель на вектор, хранящий узлы, счетчик уровня рекурсии.

Возвращаемое значение: функция ничего не возвращает.

`void remakeNumeration(Node* p)`

Назначение: расставит номера по КПЛ.

Описание аргументов: указатель на узел p.

Возвращаемое значение: функция ничего не возвращает.

`char identifySymbol(std::string ascii)`

Назначение: опознает символ.

Описание аргументов: строка с кодом.

Возвращаемое значение: опознанный символ.

`Node* findNextLeaf(std::string remainCode, int *count, Node* root)`

Назначение: проходит по дереву, ориентируясь по значению символов строки, и возвращает узел, где он остановился.

Описание аргументов: строка с кодом, указатель на количество символов, которые нужно пройти, указатель на узел.

Возвращаемое значение: узел, в котором оказалась.

`int writingCoderAnswers(std::string fileName, std::string *str, std::vector<std::string> *arrayOfLines)`

Назначение: записывает в файл полученные ответы, запускает кодер.

Описание аргументов: строка с названием файла для записи, указатель на строку с пояснением, указатель на вектор, хранящий сообщения, требующие кодировки.

Возвращаемое значение: целое число, выявляющее ошибку определенного рода.

```
void writeTreeForTask(Node* root, std::string *tree)
```

Назначение: записывает дерево в нужном виде.

Описание аргументов: корень дерева, указатель на строку для записи ответа.

Возвращаемое значение: функция ничего не возвращает.

```
int readingFile(std::string fileName, std::vector<std::string> *arrayOfLines)
```

Назначение: создает файл с заданием.

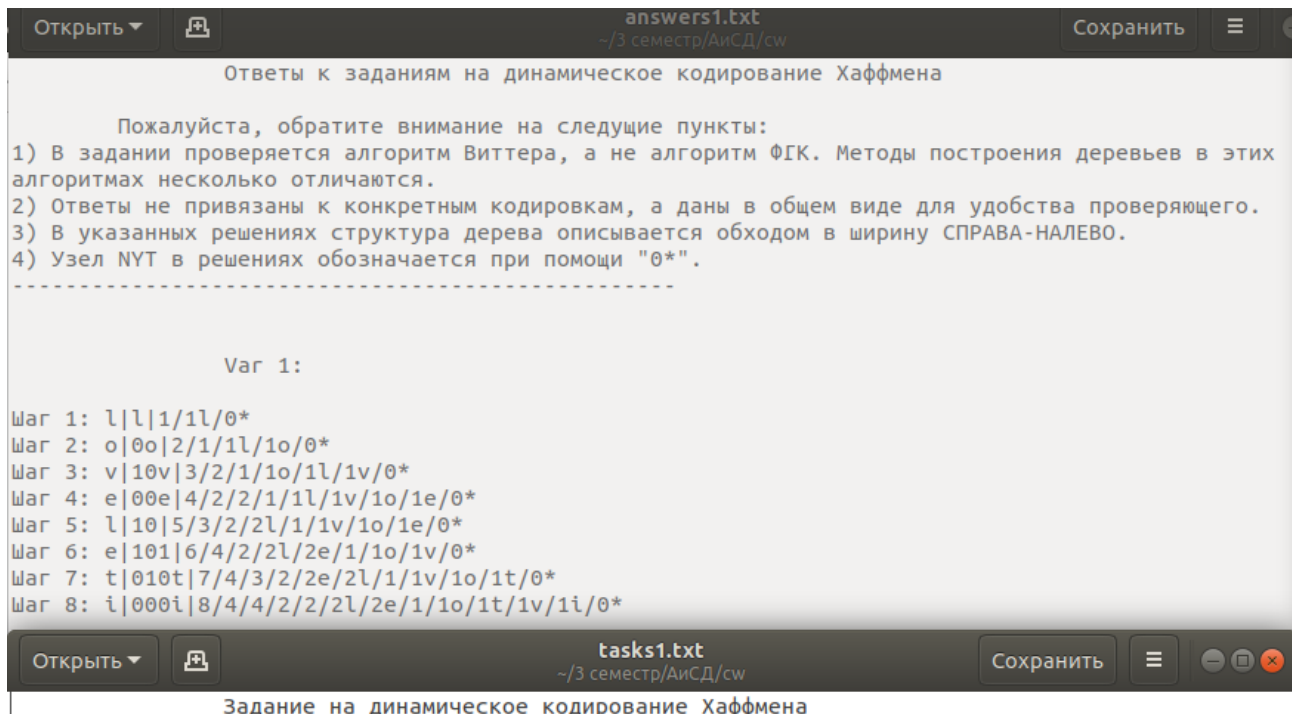
Описание аргументов: строка с названием файла для записи, указатель на строку с пояснением, указатель на вектор, хранящий сообщения, требующие кодировки.

Возвращаемое значение: целое число, выявляющее ошибку определенного рода.

4. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Все взаимодействие происходит через файлы с использованием терминала. Пользователь вводит сообщения для кодировки и их количество, после чего после запуска программы создаются два файла (один с заданием, другой с ответами), где можно будет проверить ход решения для каждого варианта, также в них всегда будет отображаться задание и пояснение к нему.

Отладочная информация выводится в консоли.



The screenshot shows two text editor windows. The top window, titled 'answers1.txt', contains the following text:

```
Отчеты к заданиям на динамическое кодирование Хаффмена

Пожалуйста, обратите внимание на следующие пункты:
1) В задании проверяется алгоритм Виттера, а не алгоритм ФГК. Методы построения деревьев в этих алгоритмах несколько отличаются.
2) Ответы не привязаны к конкретным кодировкам, а даны в общем виде для удобства проверяющего.
3) В указанных решениях структура дерева описывается обходом в ширину СПРАВА-НАЛЕВО.
4) Узел NYT в решениях обозначается при помощи "0*".
-----

Var 1:

Шаг 1: l|l|1/1l/0*
Шаг 2: o|0o|2/1/1l/1o/0*
Шаг 3: v|10v|3/2/1/1o/1l/1v/0*
Шаг 4: e|00e|4/2/2/1/1l/1v/1o/1e/0*
Шаг 5: l|10|5/3/2/2l/1/1v/1o/1e/0*
Шаг 6: e|101|6/4/2/2l/2e/1/1o/1v/0*
Шаг 7: t|010t|7/4/3/2/2e/2l/1/1v/1o/1t/0*
Шаг 8: i|000i|8/4/4/2/2/2l/2e/1/1o/1t/1v/1i/0*
```

The bottom window, titled 'tasks1.txt', contains the following text:

```
Задание на динамическое кодирование Хаффмена

Для заданного текста (последовательности символов) схематически изобразить процесс работы кодировщика по методу динамического кодирования по Хаффмену (алгоритм Виттера).
Для этого на каждом шагу указать очередной обработанный символ входной последовательности и его код, а также полученное после обработки этого символа кодовое дерево.

Пожалуйста, обратите внимание на следующие пункты:
1) В задании проверяется алгоритм Виттера, а не алгоритм ФГК. Методы построения деревьев в этих алгоритмах несколько отличаются.
2) При указании ответов необходимо приложить кодировку символов, которой вы пользовались.
3) В указанных решениях структура дерева должна описываться обходом в ширину СПРАВА-НАЛЕВО.
4) Узел NYT в решениях должен обозначаться при помощи "0*".
-----

Var 1: loveleti
```

5. ТЕСТИРОВАНИЕ

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	q	Начинается считывание строк... Не указано количество вариантов заданий. Завершение программы...	Завершение программы по желанию пользователя
2.	6 wonderful world abcd abccad jojo awd	Начинается считывание строк... Считывание строк закончилось успешно. Создание файла с заданиями. Начинается запись решений к заданиям. Решаем 1 вар. Текст: wonderful world Шаг 1 = w w 1/1w/0* Шаг 2 = o 0o 2/1/1w/1o/0* Шаг 3 = n 10n 3/2/1/1o/1w/1n/0* Шаг 4 = d 00d 4/2/2/1/1w/1n/1o/1d/0* Шаг 5 = e 110e 5/3/2/2/1/1n/1o/1d/1w/1e/0* Шаг 6 = r 100r 6/4/2/2/2/1/1o/1d/1w/1e/1n/1r/0* Шаг 7 = f 010f 7/4/3/2/2/2/1/1d/1w/1e/1n/1r/1o/1f/0* Шаг 8 =	Граничные данные

	<p>u 000u 8/4/4/2/2/2/2/1/1w/1e/1n/1r/1o/1f/1d/1u/0*</p> <p>Шаг 9 = l 1110l 9/5/4/3/2/2/2/2/1/1e/1n/1r/1o/1f/1d/1u/1w/1l/0*</p> <p>Шаг 10 = 1100 10/6/4/4/2/2/2/2/2/1/1n/1r/1o/1f/1d/1u/1w/1l/1e/1 /0*</p> <p>Шаг 11 = w 1110 11/7/4/4/3/2/2/2/2/2w/1/1r/1o/1f/1d/1u/1n/1l/1e/1 /0*</p> <p>Шаг 12 = o 010 12/8/4/4/4/2/2/2/2/2w/2o/1/1r/1f/1d/1u/1n/1l/1e/1 /0*</p> <p>Шаг 13 = r 010 13/8/5/4/4/3/2/2/2/2w/2o/2r/1/1f/1d/1u/1n/1l/1e/1 /0*</p> <p>Шаг 14 = l 1101 14/8/6/4/4/4/2/2/2/2w/2o/2r/2l/1/1d/1u/1n/1f/1e/1 /0*</p> <p>Шаг 15 = d 000 15/8/7/4/4/4/3/2/2/2w/2o/2r/2l/2d/1/1u/1n/1f/1e/1 /0*</p> <p>Решение варианта завершено.</p> <p>Решаем 2 вар.</p> <p>Текст: abcd</p> <p>Шаг 1 = a a 1/1a/0*</p> <p>Шаг 2 = b 0b 2/1/1a/1b/0*</p> <p>Шаг 3 = c 10c 3/2/1/1b/1a/1c/0*</p> <p>Шаг 4 = d 00d 4/2/2/1/1a/1c/1b/1d/0*</p>	
--	--	--

		<p>Решение варианта завершено.</p> <p>Решаем 3 вар.</p> <p>Текст: abccad</p> <p>Шаг 1 = a a 1/1a/0*</p> <p>Шаг 2 = b 0b 2/1/1a/1b/0*</p> <p>Шаг 3 = c 10c 3/2/1/1b/1a/1c/0*</p> <p>Шаг 4 = c 01 4/2/2c/1/1a/1b/0*</p> <p>Шаг 5 = a 10 5/3/2c/2a/1/1b/0*</p> <p>Шаг 6 = d 100d 6/4/2/2a/2c/1/1b/1d/0*</p> <p>Решение варианта завершено.</p> <p>Решаем 4 вар.</p> <p>Текст: jojo</p> <p>Шаг 1 = j j 1/1j/0*</p> <p>Шаг 2 = o 0o 2/1/1j/1o/0*</p> <p>Шаг 3 = j 0 3/2j/1/1o/0*</p> <p>Шаг 4 = o 01 4/2/2j/2o/0*</p> <p>Решение варианта завершено.</p> <p>Решаем 5 вар.</p> <p>Текст: awd</p> <p>Шаг 1 = a a 1/1a/0*</p>	
--	--	---	--

		<p>Шаг 2 = w 0w 2/1/1a/1w/0*</p> <p>Шаг 3 = d 10d 3/2/1/1w/1a/1d/0*</p> <p>Решение варианта завершено.</p> <p>Решаем 6 вар.</p> <p>Текст:</p> <p>Решение варианта завершено.</p> <p>Создание файла с решениями успешно.</p>	
3.	3 wwwLeningrad abcd abccad	<p>Начинается считывание строк... Считывание строк закончилось успешно. Создание файла с заданиями. Начинается запись решений к заданиям.</p> <p>Решаем 1 вар.</p> <p>Текст: wwwLeningrad</p> <p>Шаг 1 = w w 1/1w/0*</p> <p>Шаг 2 = w 1 2/2w/0*</p> <p>Шаг 3 = w 1 3/3w/0*</p> <p>Шаг 4 = L 0L 4/3w/1/1L/0*</p> <p>Шаг 5 = e 00e 5/3w/2/1/1L/1e/0*</p> <p>Шаг 6 = n 010n 6/3/3w/2/1/1e/1L/1n/0*</p> <p>Шаг 7 = i 100i 7/4/3w/2/2/1/1L/1n/1e/1i/0*</p>	Работа программы в стандартном режиме

		<p>Шаг 8 = n 101 8/5/3/3w/2/2n/1/1L/1e/1i/0*</p> <p>Шаг 9 = g 000g 9/5/4/3w/2/2/2n/1/1e/1i/1L/1g/0*</p> <p>Шаг 10 = r 1010r 10/6/4/3/3w/2/2/2n/1/1i/1L/1g/1e/1r/0*</p> <p>Шаг 11 = a 1100a 11/7/4/4/3w/2/2/2/2n/1/1L/1g/1e/1r/1i/1a/0*</p> <p>Шаг 12 = d 0110d 12/7/5/4/3/3w/2/2/2/2n/1/1g/1e/1r/1i/1a/1L/1d/0*</p> <p>Решение варианта завершено.</p> <p>Решаем 2 вар.</p> <p>Текст: abcd</p> <p>Шаг 1 = a a 1/1a/0*</p> <p>Шаг 2 = b 0b 2/1/1a/1b/0*</p> <p>Шаг 3 = c 10c 3/2/1/1b/1a/1c/0*</p> <p>Шаг 4 = d 00d 4/2/2/1/1a/1c/1b/1d/0*</p> <p>Решение варианта завершено.</p> <p>Решаем 3 вар.</p> <p>Текст: abccad</p> <p>Шаг 1 = a a 1/1a/0*</p> <p>Шаг 2 = b 0b 2/1/1a/1b/0*</p>	
--	--	--	--

		<p>Шаг 3 = c 10c 3/2/1/1b/1a/1c/0*</p> <p>Шаг 4 = c 01 4/2/2c/1/1a/1b/0*</p> <p>Шаг 5 = a 10 5/3/2c/2a/1/1b/0*</p> <p>Шаг 6 = d 100d 6/4/2/2a/2c/1/1b/1d/0*</p> <p>Решение варианта завершено.</p> <p>Создание файла с решениями успешно.</p>	
--	--	---	--

4.	-1 qq qqq q	<p>Начинается считывание строк... Считывание строк закончилось успешно. Создание файла с заданиями. Начинается запись решений к заданиям.</p> <p>Создание файла с решениями успешно.</p>	Некорректный ввод
5.		<p>Начинается считывание строк... Проблема с открытием файла. Завершение программы...</p>	Если нет файла с текстом
6.	3 qqqqq qqqqqqq qq https://repl.it/ @kruglova9382/ Kruglovacw#headers.h	<p>Начинается считывание строк... Считывание строк закончилось успешно. Создание файла с заданиями. Начинается запись решений к заданиям.</p> <p>Решаем 1 вар.</p> <p>Текст: qqqqq</p>	Граничные данные

		<p>Шаг 1 = $q q 1/1q/0^*$</p> <p>Шаг 2 = $q 1 2/2q/0^*$</p> <p>Шаг 3 = $q 1 3/3q/0^*$</p> <p>Шаг 4 = $q 1 4/4q/0^*$</p> <p>Шаг 5 = $q 1 5/5q/0^*$</p> <p>Решение варианта завершено.</p> <p>Решаем 2 вар.</p> <p>Текст: qqqqqqq</p> <p>Шаг 1 = $q q 1/1q/0^*$</p> <p>Шаг 2 = $q 1 2/2q/0^*$</p> <p>Шаг 3 = $q 1 3/3q/0^*$</p> <p>Шаг 4 = $q 1 4/4q/0^*$</p> <p>Шаг 5 = $q 1 5/5q/0^*$</p> <p>Шаг 6 = $q 1 6/6q/0^*$</p> <p>Шаг 7 = $q 1 7/7q/0^*$</p> <p>Решение варианта завершено.</p> <p>Решаем 3 вар.</p> <p>Текст: qq</p> <p>Шаг 1 = $q q 1/1q/0^*$</p> <p>Шаг 2 = $q 1 2/2q/0^*$</p>	
--	--	--	--

		<p>Решение варианта завершено.</p> <p>Создание файла с решениями успешно.</p>	
7.	3	<p>Начинается считывание строк... Считывание строк закончилось успешно. Создание файла с заданиями. Начинается запись решений к заданиям.</p> <p>Создание файла с решениями успешно.</p>	Некорректный ввод

ЗАКЛЮЧЕНИЕ

Изучен метод динамического кодирования и декодирования по Хаффману. Написана работающая программа на языке C++ для проведения текущего контроля по этой теме. Ответы выводятся в развернутом виде, чтобы облегчить проверку.

Результаты, полученные при тестировании программы, совпадают с теоретическими ожиданиями.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

Алгоритм Виттера – 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

Реализация алгоритма на C++ –

<https://www.cyberforum.ru/cpp-beginners/thread269166.html>

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp:

```
#include "headers.h"

unsigned int count = 187; // ( = 2*кол-во символов алфавита - 1) нужно для нумерования
элементов дерева (94 печатных символа первой половины ascii)
std::vector<Node*> leafs; // здесь хранятся все адреса на узлы-листья + NYT (на нулевом
месте)
std::vector<Node*> internalNodes; // здесь хранятся все адреса на внутренние узлы
(корень на нулевом месте)

void printHufTree(Node* root) // визуализирует структуру дерева Хаффмана
{
    using namespace std;

    vector<Node*> vec;
    splitHafTree(root, &vec, 1); // запишет в вектор все узлы дерева по уровням (по порядку)
    ПОМНИ, ЧТО УРОВНИ РАЗДЕЛЕНЫ nullptr

    cout << '/' << std::string( 60, '*' ) << '/' << endl;

    auto iter = vec.begin();
    while(iter != vec.end())
    {
        if((*iter) == nullptr) // если встретили разделитель уровня
        {
            iter++;
            continue;
        }

        cout << "/* № " << (*iter)->m_number << ": Par = [";

        if ( ((*iter)->m_parent) != nullptr)
        {
            cout << (*iter)->m_parent->m_number;
            if( ((*iter)->m_parent->m_left) == (*iter) )
                cout << 'L';
            else
                cout << 'R';
        }
        else
        {
            cout << "Null";
        }
        cout << "]\tW = " << (*iter)->m_weight;

        if((*iter)->m_isLeaf) // если это лист
        {
            cout << "\tType = Leaf\tSymbol = " << (*iter)->m_symbol << endl;
        }
        else if((*iter)->m_isNYT) // если это NYT
        {
            cout << "\tType = NYT" << endl;
        }
        else // если это внутренний узел
        {
            cout << "\tType = Internal" << endl;
        }
    }
}
```

```

    }

    iter ++;
}
cout << '/' << std::string( 60, '*' ) << '/' << '\n' << endl;
}

void pushFirst(Node* a)
{
    leafs.push_back(a);
};

Node* findSymbol(char symbol, std::vector<Node*> *leafs) // ищет в переданном векторе из
уникальных листьев тот, что содержит нужный символ
{
    if (symbol == '\0') // \0 - обозначение для внутренних узлов, поэтому нет уникального
элемента с таким символом
        return nullptr;

    int lastInd = leafs->size();
    for (int i = 1; i < lastInd; i++) // идем по списку листьев и проверяем значения символа
внутри
    {
        if(((leafs->at(i))->m_symbol) == symbol) // если переданный символ встречался
раньше
        {
            return leafs->at(i);
        }
    }
    return leafs->at(0); // если же это новый символ, то вернем NYT
};

void swapNodes(Node *a, Node *b) // меняет 2 узла/листа местами с помощью замены их
собственных указателей и указателей их родителей
{
    if (a->m_parent == nullptr || b->m_parent == nullptr) // если одно из значений - корень
    {
        std::cout << "You can't swap root." << std::endl;
        return;
    }

    if (a == b)
    {
        std::cout << "You can't swap one element with itself." << std::endl;
        return;
    }

    if ((a->m_parent) == (b->m_parent)) // если у элементов один родитель
    {
        Node* parent = a->m_parent;
        if(a->m_parent->m_left == a) // если a - левый ребенок, a b - правый
        {
            a->m_parent->m_left = b;
            a->m_parent->m_right = a;
        }
        else // если же a - правый ребенок, a b - левый
        {
            a->m_parent->m_left = a;
            a->m_parent->m_right = b;
        }
    }
}

```

```

    // менять указатели на родителей переданным элементам не нужно
}
else // если у элементов разные родители
{
    // меняем указатели на детей в их родителях

    Node *tempVal; // временное место хранения значения

    tempVal = a->m_parent;
    if( (tempVal)->m_right == a) // если a - правый ребенок, то меняем указатель
родителя на правого ребенка
        (tempVal)->m_right = b;
    else // иначе меняем указатель на левого ребенка
        (tempVal)->m_left = b;

    tempVal = b->m_parent; // ошибка!!! У родителя b сейчас оба ребенка = b ПОПРАВЬ!
    if( (tempVal)->m_right == b) // аналогично поступаем с родителем второго узла
        (tempVal)->m_right = a;
    else
        (tempVal)->m_left = a;

    // меняем указатели на родителей в детях

    b->m_parent = a->m_parent;
    a->m_parent = tempVal;
};

if(b->m_isLeaf && a->m_isLeaf) // если это 2 листа, то можно спокойно поменять у них
номера на друг друга
{
    unsigned int k = a->m_number;
    a->m_number = b->m_number;
    b->m_number = k;
}
};

Node* findBlockLeader(std::vector<Node*> *vec, unsigned int weight) // находит лидера
блока (лидер блока: тот же вес, тот же тип (лист/вн.узел), максимальный номер)
{ // после вызова сделай проверку на то, не является ли тот же элемент и лидером (для
лист-лист) и на nullptr (надо ли вообще делать swap)
    Node* p = nullptr;
    unsigned int number = 0;

    for (int i = 0; i < vec->size(); i++) // проходимся по всем элементам
    {
        if((vec->at(i))->m_weight == weight) // если вес элемента == нужному весу
        {
            if(p == nullptr) // первый подходящий элемент
            {
                p = vec->at(i);
                number = p->m_number;
            }
            else if( (vec->at(i))->m_number > number) // если у элемента номер больше
            {
                p = vec->at(i);
                number = p->m_number;
            }
        }
    }
}

```



```

    return p;
};

void encodeSymbol(std::string* emptyStr, Node* p) // записывает в переданную строку путь
до листа/НУТ
{
    Node* prevNode = p; // от этого узла будем подниматься наверх

    while(prevNode->m_parent != nullptr) // пока не находимся в корне
    {
        if(prevNode->m_parent->m_left == prevNode) // если это левый ребенок
        {
            emptyStr->insert(0,"0"); // вставляем в начало 0
            prevNode = prevNode->m_parent;
        }
        else if(prevNode->m_parent->m_right == prevNode) // если это правый ребенок
        {
            emptyStr->insert(0,"1"); // вставляем в начало 1
            prevNode = prevNode->m_parent;
        }
        else
        {
            std::cerr << "There is strange error in encodeSymbol()." << std::endl;
            return;
            // если это вывелось, значит где-то ошибка в создании дерева (указатель на
ребенка пуст)
        }
    }
};

void splitHafTree(Node* root, std::vector<Node*> *vec, short int level) // запишет все листы и
узлы в вектор по порядку КПЛ
{
    if (level == 1) // если это самый первый проход
    {
        vec->push_back(root); // то просто создаем первый блок
        vec->push_back(nullptr); // nullptr служат для разделения блоков (уровней дерева)
    }
    else
    {
        short int countOfNulls = 0; // для подсчета уровней
        auto k = vec->begin();
        while (k != vec->end()) // проходимся по всему вектору
        {
            if ((*k) == nullptr) // если встретили разделитель
            {
                countOfNulls += 1;
                if (countOfNulls == level) // если количество разделителей = уровню функции
                {
                    vec->insert(k, root); // то записываем значение в конец нужного блока
                    break; // и останавливаем итерацию
                }
            }
            k++;
        }
        if(countOfNulls < level) // если в векторе меньше блоков, чем уровень у функции, то
добавим еще один блок
        {
            vec->push_back(root);
            vec->push_back(nullptr);
        }
    }
}

```

```

    }
}

if ((root->m_left) != nullptr) // если узел внутренний, то делаем рекурсивный вызов
функции для детей узла
{
    splitHafTree(root->m_right, vec, level + 1);
    splitHafTree(root->m_left, vec, level + 1);
}
return;
};

void remakeNumeration(Node* p) // передаем этой функции указатель на корень и она
расставит номера по КПЛ
{
    std::vector<Node*> vec;
    splitHafTree(p, &vec, 1); // теперь в vec упорядоченно лежат все узлы и листья дерева
    unsigned int count = 187;

    auto iter = vec.begin();
    while(iter != vec.end()) // проходимся по всем узлам и выставляем там
соответствующие номера
    {
        if (*iter != nullptr) // пропускаем все разделители
        {
            (*iter)->m_number = count;
            count -= 1;
        }
        iter++;
    }
};

void deleteAllNodes()
{
    std::vector<Node*> *allLeafs = &leafs;
    std::vector<Node*> *allIntNodes = &internalNodes;

    auto l = allLeafs->begin();
    auto iN = allIntNodes->begin();

    while(l != allLeafs->end())
    {
        delete *l;
        l++;
    }
    while(iN != allIntNodes->end())
    {
        delete *iN;
        iN++;
    }
};

void fillSymbolCode(char symbol, std::string *str) // записывает ascii-номер символа в строку
(длина = 8)
{
    int c = (int)symbol;
    str->push_back('0');
    for(int j = 64; j > 1; j = j/2)
    {
        if(c >= j)

```

```

    {
        c -= j;
        str->push_back('1');
    }
    else
    {
        str->push_back('0');
    }
}
if(c == 1)
    str->push_back('1');
else
    str->push_back('0');
};

Node* slideAndIncrement(Node* p)
{
    if(!(p->m_isLeaf)) // если это внутренний узел
    {
        Node* previousP = p->m_parent;

        // сдвигаем p в дереве выше, чем узлы-листья с весом w+1
        Node* a = findBlockLeader(&leafs, (p->m_weight) + 1);
        if(a != nullptr) // если такие листья есть
        {
            swapNodes(a, p); // т.к. здесь swap листа с внутренним узлом, то swap не поменяет
            // автоматически внутреннюю нумерацию
            Node* p = internalNodes.at(0); // на первом месте в списке внутренних узлов стоит
            // корень
            remakeNumeration(p); // пересоздаем нумерацию искусственно, начиная с корня
        }
        p->m_weight += 1;
        return previousP;
    }
    else // если же это лист
    {
        // сдвигаем p в дереве выше, чем внутренние узлы с весом w
        Node* b = findBlockLeader(&internalNodes, p->m_weight);
        if (b != nullptr)
        {
            swapNodes(p, b);
            remakeNumeration(internalNodes.at(0));
        }
        p->m_weight += 1;
        return p->m_parent;
    }
};

```

void writeTreeForTask(Node* root, std::string *tree) // в переданную строку записывает дерево в настоящий момент в виде: 3/2/1w/1o...

```

{
    std::vector<Node*> vec;
    splitHafTree(root, &vec, 1); // теперь в vec упорядоченно лежат все узлы и листья
    // дерева (они лежат по обходу в ширину СПРАВА-НАЛЕВО)

    auto iter = vec.begin();
    while(iter != vec.end()) // проходимся по всем узлам
    {
        if ((*iter) != nullptr) // пропускаем все разделители уровней

```

```

{
    tree->append(std::to_string((*iter)->m_weight)); // записываем вес узла
    if((*iter)->m_isNYT)
        tree->push_back('*');
    else if((*iter)->m_isLeaf) // и если этот узел - лист
    {
        tree->push_back((*iter)->m_symbol); // то записываем и его символ
        tree->push_back('/'); // символ разделения узлов
    }
    else
        tree->push_back('/');
}
iter++;
}
}

```

std::string vitterCoder(char symbol, std::string *step) // step - строка, которая показывает результат шага (нужна для проверки заданий)

```

{
    std::string code;
    Node* nodeForIncrease = nullptr;
    Node* p = findSymbol(symbol, &leafs); // смотрим не встречался ли нам такой символ раньше

```

```

    { // записали в начало строки шага обрабатываемый символ
    step->push_back(symbol);
    }

```

```

    if (p->m_isNYT) // если этот символ встретился впервые
    {

```

```

        // Кодирование символа:
        encodeSymbol(&code, p); // записывает в строку путь до NYT

```

```

        { // добавляет в строку шага часть с |...|
        step->push_back('|');
        step->append(code);
        step->push_back(symbol);
        step->push_back('|');
        }

```

```

        { // кавычки здесь, чтобы не держать в памяти symbolCode (переменная будет уничтожена при выходе из них)

```

```

            std::string symbolCode = "";
            fillSymbolCode(symbol, &symbolCode); // получаем ascii-код символа
            code.append(symbolCode); // добавляем его в код
        }

```

```

        // Перестройка дерева:

```

```

        p->m_left = new Node(p); // создаем новый NYT
        p->m_right = new Node(p, symbol); // и создаем новый лист
        p->m_isNYT = false; // в узле, где мы находимся, указываем, что он больше не NYT
        nodeForIncrease = p->m_right; // указываем, что лист был только что создан, и его вес нужно увеличить

```

```

        leafs.push_back(p->m_right); // записываем новосозданный лист в конец вектора листьев

```

```

        leafs.at(0) = p->m_left; // меняем указатель на новый NYT в векторе

```

```

        internalNodes.push_back(p); // так как старый NYT стал внутренним узлом сохраним
        его в соответствующий вектор
    }
    else // если же символ встречался ранее
    {
        // Кодирование символа:

        encodeSymbol(&code, p); // записывает в строку путь до листа (что и является кодом
        символа)

        { // добавляет в строку шага часть с [...]
            step->push_back('|');
            step->append(code);
            step->push_back('|');
        }

        // Перестройка дерева:

        // swap лист с тем же элементов с листом-лидером блока (блок - один тип, один вес)
        Node* k = findBlockLeader(&leafs, p->m_weight);
        if (k != p) // на nullptr проверка здесь не нужна т.к. гарантированно есть хотя бы 1
        элемент такого веса (сам p)
            swapNodes(p, k);
        if ( p->m_parent->m_left->m_isNYT ) // если p после перемещения все равно брат NYT
        (это нужно во избежания пары проблем с slideAndIncrement() )
        {
            nodeForIncrease = p; // лист будет увеличен в конце отдельно
            p = p->m_parent;    // а цикличное увеличение начнется с его родителя
        }
    }

    while (p != nullptr) // увеличиваем вес и делаем необходимые сдвиги в дереве от p до
    корня
    {
        p = slideAndIncrement(p);
    };

    if (nodeForIncrease != nullptr) // делаем еще одно увеличение и сдвиг в дереве, если это
    необходимо
    {
        slideAndIncrement(nodeForIncrease);
    };

    writeTreeForTask(internalNodes.at(0), step); // записываем в строку-шаг полученное в
    результате кодирования дерево (СПРАВА-НАЛЕВО)

    return code; // возвращаем строку по значению
};

```

```

char identifySymbol(std::string ascii) // принимает на вход строку с ascii-кодом символа и
возвращает опознанный символ
{
    int num = 0;
    char symbol = 0;
    for (int i = 0; i < ascii.size(); i++) // идем по всем символам строки
    {
        if (ascii[i] == '0')
        {
            continue;

```

```

    }
    else if(ascii[i] == '1')
    {
        num = 1;
        for (int k = 7 - i; k > 0; k--)
        {
            num *= 2;
        }
        symbol += num;
    }
}
return symbol;
};

```

Node* findNextLeaf(std::string remainCode, int *count, Node* root) // проходит по дереву, ориентируясь по значению символов строки, и возвращает узел, где он остановился

```

{
    Node* currentNode = root;
    int k = 0;

    while( !(currentNode->m_isLeaf) && !(currentNode->m_isNYT))
    {
        if(remainCode[k] == '0')
        {
            currentNode = currentNode->m_left;
        }
        else if(remainCode[k] == '1')
        {
            currentNode = currentNode->m_right;
        }
        k += 1;
    }
    *count = k; // сообщает наружу, сколько символов занимает путь до листа

    return currentNode;
};

```

std::string vitterDecoder(std::string *code) // декодирует код в сообщение

```

{
    setlocale(LC_ALL, "rus");
    Node* root = new Node; // использован конструктор для создания первого NYT
    std::string startCode = *code; // переданный код, который будем раскодировывать
    std::string message = ""; // сюда будут добавляться друг за другом

```

раскодированные символы

```

    std::string temporaryStr = ""; // строка для временного хранения значений
    int count = 0;
    char symbol;

```

```

    ::leafs.push_back(root); // сохраняем корень в список листьев, т.к. он пока NYT

```

// Первыми всегда идут 8 цифр кода символа

```

    temporaryStr.assign(startCode, 0, 8); // сохраняем 8 цифр кода во временной
    переменной

```

```

    startCode.erase(0,8); // удаляем код символа из общей строки кода

```

```

    symbol = identifySymbol(temporaryStr); // распознаем этот символ

```

```

    message.push_back(symbol); // записываем этот символ в конец

```

декодированного сообщения

// начальная перестройка дерева

```

    root->m_left = new Node(root); // создаем новый NYT

```

```

    root->m_right = new Node(root, symbol); // и создаем новый лист

```

```

    root->m_isNYT = false;          // в узле, где мы находимся(бывший NYT), указываем,
    что он больше не NYT

    ::leafs.at(0) = root->m_left;    // обновили NYT в списке
    ::leafs.push_back(root->m_right); // записали новый лист в список листьев
    ::internalNodes.push_back(root); // записали корень в список внутренних узлов (он
    занял 0 позицию и с нее не уйдет)

    root->m_weight = 1;              // это простейший случай, поэтому прописали веса
    вручную
    root->m_right->m_weight = 1;

    while(startCode != "")          // пока не проанализируем строку до конца
    {
        Node* nextLeaf = findNextLeaf(startCode, &count, root); // находит к какому листу
        ведет последовательность (в count записывается количество символов, кодирующих
        путь до листа)
        Node* nodeToIncrease = nullptr; // лист для увеличения (нужен для
        отделения узлов, вес которых нужно увеличивать в самом конце)

        std::string temp;
        temp.assign(startCode, 0, count); // код, который ведет до листа

        if(nextLeaf->m_isNYT) // если последовательность привела к NYT -> встретился
        новый символ
        {
            startCode.erase(0, count); // удалили путь до NYT из общей строки кода
            temporaryStr.assign(startCode, 0, 8); // сохранили во временное хранилище 8
            символов (код нововстреченного символа)
            startCode.erase(0, 8); // удалили код нововстреченного символа из
            общей строки кода
            symbol = identifySymbol(temporaryStr); // идентифицируем символ по его коду
            message.push_back(symbol); // записываем идентифицированный символ в
            строку раскодированного сообщения

            // перестройка дерева
            nextLeaf->m_left = new Node(nextLeaf); // создаем новый NYT
            nextLeaf->m_right = new Node(nextLeaf, symbol); // и создаем новый лист
            nextLeaf->m_isNYT = false; // в узле, где мы находимся, указываем,
            что он больше не NYT

            ::leafs.at(0) = nextLeaf->m_left; // обновили NYT
            ::leafs.push_back(nextLeaf->m_right); // указали новосозданный лист в списке
            листьев
            ::internalNodes.push_back(nextLeaf); // старый NYT записали в список внутренних
            узлов

            nodeToIncrease = nextLeaf->m_right;
        }
        else // если же последовательность привела к конкретному листу -> символ из этого
        листа встречается повторно
        {
            // раскодирование символа
            startCode.erase(0, count); // удалили путь до листа из общей строки кода
            symbol = nextLeaf->m_symbol; // узнали, что за символ был встречен из
            значения листа
            message.push_back(symbol); // записали этот символ в конец раскодированного
            сообщения

            Node* k = findBlockLeader(&::leafs, nextLeaf->m_weight); // находим лидера блока

```

```

        if (k != nextLeaf) // если лидер блока != листу, содержащему анализируемый
символ
        {
            swapNodes(nextLeaf, k); // то меняем местами эти 2 листа (swapNodes уже имеет
внутри изменение нумерации дерева для лист swap лист)
        }

        if ( nextLeaf->m_parent->m_left->m_isNYT ) // если p после перемещения все равно
брат NYT (это нужно во избежания пары проблем с slideAndIncrement() )
        {
            nodeToIncrease = nextLeaf;        // лист будет увеличен в конце отдельно
            nextLeaf = nextLeaf->m_parent;    // а цикличное увеличение начнется с его
родителя
        }
    }

    while(nextLeaf != nullptr) // цикличное увеличение веса узлов снизу вверх
    {
        nextLeaf = slideAndIncrement(nextLeaf);
    };

    if(nodeToIncrease != nullptr) // увеличение веса узла, специально вынесенного
последним на увеличение
    {
        slideAndIncrement(nodeToIncrease);
    };
}

return message;
};

```

```

int readingFile(std::string fileName, std::vector<std::string> *arrayOfLines) // считывает из
файла массив строк
{
    using namespace std;
    char ch;
    int count;
    std::string a;
    ifstream file(fileName, ios::in | ios::binary); // открываем файл для чтения
    if (!file)
    {
        cout << "Проблема с открытием файла.\nЗавершение программы..." << endl;
        return -1;
    }

    file >> count; // считали количество вариантов заданий
    if(file.fail()) // если количество вариантов заданий не было введено
    {
        cout << "Не указано количество вариантов заданий.\nЗавершение программы..." <<
endl;
        return -2;
    }
    getline(file, a, '\n'); // считываем остаток строки, где находилось число вариантов, для
перехода к данным

    for(int k = 0; k < count; k++) // циклично считываем строки из файла (если указано
большее количество вариантов, чем их есть на деле, то эта разница будет заполнена
копиями последней строки)

```



```

{
    getline(file, a, '\n'); // пустые строки будут считаться отдельными строками
    arrayOfLines->push_back(a);
    //cout << "string nom " << k + 1 << ": " << a << endl; // просто для проверки
}

file.close(); // закрываем файл
return 0;
}

int writingTasks(std::string fileName, std::string *str, std::vector<std::string> *arrayOfLines) //
создает файл с заданием
{
    using namespace std;
    ofstream out(fileName); // открываем файл для записи
    if (!out) // если не удалось его открыть
    {
        cout << "Не получается открыть файл для создания заданий.\n";
        return -1;
    }

    out << *str << "\n"; // в str содержится текст задания
    for(int i = 0; i < 50; i++) // отделяем задание от вариантов заданий с помощью "-----"
        out << '-';
    out << "\n\n";

    for (int i = 0; i < arrayOfLines->size(); i++)
    {
        out << "Var " << i + 1 << ": " << arrayOfLines->at(i) << "\n";
    }

    out.close();
    return 0;
}

int writingCoderAnswers(std::string fileName, std::string *str, std::vector<std::string>
*arrayOfLines)
{
    using namespace std;
    ofstream out(fileName); // открываем файл для записи
    if (!out) // если не удалось его открыть
    {
        cout << "Не получается открыть файл для создания ответов.\n";
        return -1;
    }
    string newLine = "", step = "";
    int counter = 0;

    out << *str << "\n"; // в str содержится уточнение, к какому заданию эти ответы
    for(int i = 0; i < 50; i++) // отделяем уточнение от непосредственно решений с
помощью "-----"
        out << '-';
    out << "\n\n\n";
    cout << "\n\n" << endl;

    auto iter = arrayOfLines->begin();
    while(iter != arrayOfLines->end()) // проходимся по всем введенным строкам
    {
        counter += 1;
        out << "\t\tVar " << counter << ":\n\n";
        cout << "\tРешаем " << counter << " вар.\n" << endl;
    }
}

```

```

cout << " Текст: " << (*iter) << "\n" << endl;

::leafs.clear();
::internalNodes.clear();
Node *root = new Node; // был вызван конструктор для корневого NYT
pushFirst(root);      // вставили начальный NYT в список листьев

for (int i = 0; i < (*iter).size(); i++) // для каждого символа строки вызывается кодер
{
    string a = vitterCoder((*iter).at(i), &step); // в a - код символа, в step - строка-шаг,
    которую должен изобразить ученик
    newLine.append(a); // добавили его в общий код строки
    out << "Шаг " << i + 1 << ": " << step << "\n";
    cout << "Шаг " << i + 1 << " = " << step << "\n" << endl;
    step = "";
}
out << "\n\n\n";
cout << "\tРешение варианта завершено.\n\n" << endl;
deleteAllNodes();
iter++;
}
out.close();
return 0;
}

int main()
{
    using namespace std;
    setlocale(LC_ALL, "rus");

    string inputFile = "./text.txt";
    string tasks1 = "./tasks1.txt", answers1 = "./answers1.txt"; // задания и ответы по
кодирования
    string task1Statement = "", task2Statement = ""; // здесь будут храниться
формулировки заданий
    string ans1Statement = "", ans2Statement = ""; // здесь будут храниться
    std::vector<std::string> arrayOfLines, arrayOfCodes; // здесь будут храниться
переданные строки, и их полученные коды
    { // формулировка заданий и уточнения к файлу ответов
        task1Statement = "\t\tЗадание на динамическое кодирование Хаффмена\n\nДля
заданного текста (последовательности символов) схематически изобразить процесс
работы кодировщика по методу динамического кодирования по Хаффмену (алгоритм
Виттера).\nДля этого на каждом шагу указать очередной обработанный символ входной
последовательности и его код, а также полученное после обработки этого символа
кодированное дерево.\n\n\tПожалуйста, обратите внимание на следующие пункты: \n1) В
задании проверяется алгоритм Виттера, а не алгоритм ФГК. Методы построения
деревьев в этих алгоритмах несколько отличаются.\n2) При указании ответов
необходимо приложить кодировку символов, которой вы пользовались.\n3) В указанных
решениях структура дерева должна описываться обходом в ширину СПРАВА-НАЛЕВО.\n4)
Узел NYT в решениях должен обозначаться при помощи \"0*\".";
        ans1Statement = "\t\tОтветы к заданиям на динамическое кодирование Хаффмена\n\n
\tПожалуйста, обратите внимание на следующие пункты: \n1) В задании проверяется
алгоритм Виттера, а не алгоритм ФГК. Методы построения деревьев в этих алгоритмах
несколько отличаются.\n2) Ответы не привязаны к конкретным кодировкам, а даны в
общем виде для удобства проверяющего.\n3) В указанных решениях структура дерева
описывается обходом в ширину СПРАВА-НАЛЕВО.\n4) Узел NYT в решениях обозначается
при помощи \"0*\".";
    }

    cout << "Начинается считывание строк..." << endl;

```

```

    if(readingFile(inputFile, &arrayOfLines) < 0) // считали строки из файла в вектор (если
что-то не так пошло с файлом - закругляемся)
        return 0;
    cout << "Считывание строк закончилось успешно." << endl;

    writingTasks(tasks1, &task1Statement, &arrayOfLines); // создали файл с заданиями на
кодирование
    cout << "Создание файла с заданиями." << endl;

    cout << "Начинается запись решений к заданиям." << endl;
    writingCoderAnswers(answers1, &ans1Statement, &arrayOfLines); // ,кодировщик
вызывается изнутри этой функции
    cout << "Создание файла с решениями успешно." << endl;

    return 0;
}

```

headers.h:

```

#ifndef HEADERS_H
#define HEADERS_H

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

// описывает узел дерева
class Node
{
public:
    // обеспечивают построение дерева Хаффмана с другими узлами
    Node* m_parent = nullptr;
    Node* m_right = nullptr;
    Node* m_left = nullptr;

    // характеристики узла
    int m_weight = 0; // вес узла
    int m_number = 187;
    char m_symbol = '\0'; // символ который хранится в листе
    bool m_isLeaf = false; // флаг листа (TRUE, если лист)
    bool m_isNYT = true; // является ли узел NYT

    Node() = default; // базовый конструктор (исп. для корня)
    Node(Node* parent): m_parent{parent}, m_number{(parent->m_number) - 2} // конструктор
для NYT с родителем
    {};
    Node(Node* parent, char symbol): m_parent{parent}, m_symbol{symbol}, m_isNYT{false},
m_isLeaf{true}, m_number{(parent->m_number) - 1} // конструктор для листьев
    {};
};

void pushFirst(Node* a);
void deleteAllNodes();

```

```
void splitHafTree(Node* root, std::vector<Node*> *vec, short int level);  
  
std::string vitterCoder(char symbol);  
  
#endif
```