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

Студент гр. 9382 _____ Юрьев С.Ю.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Юрьев С.Ю.

Группа 9382

Тема работы : Кодирование и декодирование

Исходные данные:

Вариант 5. Динамическое кодирование и декодирование по Хаффману – демонстрация.

Демонстрация

Содержание пояснительной записки:

«Содержание», «Введение», «Постановка задачи», «Описание алгоритмов», «Описание основных функций и структур данных», «Тестирование», «Заключение», «Список использованных источников»)

Предполагаемый объем пояснительной записки:

Не менее 25 страниц.

Дата выдачи задания: 01.09.2020

Дата сдачи реферата: 19.12.2020

Дата защиты реферата: 20.12.2020

Студент _____ Юрьев С.Ю.

Преподаватель _____ Фирсов М.А.

АННОТАЦИЯ

Курсовая работа представляет собой демонстрацию алгоритмов динамического кодирования и декодирования по Хаффману. Результатом работы является демонстрация полного хода работы алгоритма с описанием действий и визуализацией структуры данных в виде таблиц и рисунков.

SUMMARY

Coursework is a demonstration of algorithms for dynamic encoding and decoding according to Huffman. The result of the work is a demonstration of the full course of the algorithm with a description of the actions and visualization of the data structure in the form of tables and figures.

.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью работы является реализация кодирования и декодирования методами динамического кодирования и декодирования по Хаффману, визуализация алгоритмов, структур данных, действий.

Методы динамического кодирования и декодирования по Хаффману — адаптивные методы, основанные на кодировании Хаффмана. Они позволяют строить кодовые схемы в поточном режиме, не имея никаких начальных знаний из исходного распределения.

Коды символов вычисляются путем построения бинарного дерева. Встреченные символы находят свое отражение в виде листа в этом дереве. Их код равен пути в дереве до соответствующего листа.

Каждый узел содержит в себе вес, порядковый номер и хранимый символ, если это лист.

Метод динамического кодирования по Хаффману хоть и позволяет существенно сократить длину закодированного сообщения, однако по эффективности сжатия он уступает статической версии. Однако динамическое декодирование по Хаффману не требует дополнительной информации об исходном сообщении, что позволяет сжать сообщение за один проход, а также закодировать лишь часть сообщения, просто остановив метод во время его выполнения без необходимости предоставления какой-либо дополнительной информации ни кодировщику, ни декодировщику.

Так как перед началом работы метод не получает никаких дополнительных данных об обрабатываемом тексте то, чтобы было возможно декодировать полученную кодовую последовательность, в кодере и декодере должны использоваться одинаковые алгоритмы создания и обновления дерева кодов при получении символов.

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 алгоритмом Виттера .

Главная идея этого метода - заменить часто встречающиеся символы более короткими кодами, а редко встречающиеся последовательности более длинными кодами, делая это на основе бинарного дерева, обновляя его прямо при получении очередного символа. Таким образом, алгоритм основывается на кодах переменной длины.

Сам алгоритм можно разбить на несколько частей :

1) Считывание символа из входной последовательности и получение его кода исходя из положения листа, содержащего такой же символ, либо из положения особого узла NYT, в случае отсутствия подходящего листа.

2) Создание двух новых узлов из узла NYT, в случае нахождения символа встретившегося впервые, левый — новый узел NYT, правый — лист, хранящий этот символ.

3) Увеличение веса всех узлов вертикально связанных с листом, хранящим встреченный символ, и веса самого этого узла, с одновременной перестройкой дерева кодов.

4) Повторение алгоритма с шага 1) для следующего символа входной последовательности.

Таким образом, будет вычислен код каждого символа входного сообщения и получено соответствующее дерево кодов.

2.2 Динамическое декодирование по Хаффману алгоритмом Виттера.

Главная идея этого алгоритма — создание бинарного дерева кодов по переданной кодовой последовательности, с получением закодированного символа исходя из состояния бинарного дерева. Так как для кодирования одного и того же символа в разные моменты времени используются разные коды, очень важно, чтобы декодирование использовало тот же алгоритм создания и обновления дерева, что и кодирование.

Сам алгоритм можно разбить на несколько частей :

1) Получение символа из первых 8 цифр кодовой последовательности, и создание соответственного дерева кодов.

2) Циклическое получение цифр кодовой последовательности с одновременным соответствующим перемещением «вниз» по дереву кодов до невозможности дальнейшего продвижения.

3) Если остановка произошла на узле NYT - получение символа из следующих 8 цифр из кодовой последовательности и создание двух новых узлов из узла NYT, левый — новый узел NYT, правый — лист, хранящий этот символ.

4) Если остановка произошла на другом узле — получение символа из хранящегося в узле значения символа.

5) Увеличение веса всех узлов вертикально связанных с листом, хранящим встреченный символ, и веса самого этого узла, с одновременной перестройкой дерева кодов.

4) Повторение алгоритма с шага 2) для оставшейся кодовой последовательности.

Таким образом, будут вычислены все закодированные символы из кодовой последовательности.

3. ОПИСАНИЕ ОСНОВНЫХ ФУНКЦИЯ И СТРУКТУР ДАННЫХ

class Node - Структура хранит параметры узла: вес, порядковый номер, хранимый символ, флаг листа и флаг NYT, указатели на «родителя» и «детей» в случае их существования.

class PaintCell — Структура хранит параметры для вывода части рисунка в терминале: символ, который требуется нарисовать; вес, который нужно нарисовать рядом с символом; координаты строки и координаты от начала строки, чтобы нарисовать символ в нужном месте.

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

std::string vitterDecoder(std::string *code) — Главная управляющая функция динамического декодирования. Получает на вход закодированную последовательность, возвращает соответствующую декодированную последовательность.

int readingFile(std::string fileName, std::string *line) — Функция считывает весь текст из файла в переданную строку. Получает на вход имя файла и указатель на строку, возвращает число.

int writingFile(std::string fileName, std::string *str) — Функция записывает в файл переданную строчку. Получает на вход имя файла и указатель на строку, возвращает число.

Node* findSymbol(char symbol, std::vector<Node*> *leafs) — Функция ищет в переданном векторе из уникальных листьев тот, что содержит указанный символ. Получает на вход символ и указатель на вектор узлов, возвращает указатель на узел.

void swapNodes(Node *a, Node *b) - Функция меняет два узла местами. Получает на вход два указателя на узлы, ничего не возвращает.

Node* findBlockLeader(std::vector<Node*> *vec, unsigned int weight) — Функция находит лидера блока. Получает на вход указатель на вектор узлов и число, возвращает указатель на узел.

void encodeSymbol(std::string* emptyStr, Node* p) — Функция записывает в переданную строку путь до указанного узла в виде последовательности цифр 1 и 0. Получает на вход указатель на строку и указатель на узел, ничего не возвращает.

void splitHafTree(Node* root, std::vector<Node*> *vec, short int level) - Функция записывает все узлы в вектор по уровням по порядку справа-налево. Получает на вход указатель на узел, указатель на вектор указателей на узлы и число, ничего не возвращает.

void remakeNumeration(Node* p) — Функция восстанавливает правильную нумерацию узлов в дереве. Получает на вход указатель на узел, ничего не возвращает.

void deleteAllNodes() - Освобождает выделенную в программе динамическую память. Ничего не принимает, ничего не возвращает.

void fillSymbolCode(char symbol, std::string *str) — Функция записывает ascii-номер символа в строку. Получает на вход символ и указатель на строку, ничего не возвращает.

char identifySymbol(std::string ascii) - Функция из ascii-кода получает соответствующий символ. Получает на вход строку, возвращает символ

Node* findNextLeaf(std::string remainCode, int *count, Node* root) - Функция проходит по дереву, ориентируясь по значению символов переданной строки, и указывает узел, где она остановилась. Получает на вход строку, указатель на число и указатель на узел, возвращает указатель на узел.

Node* slideAndIncrement(Node* p) — Функция увеличивает вес узла и делает необходимые перестройки дерева. Получает на вход указатель на узел, возвращает указатель на узел.

void setNodesCoordinates(Node* root, int x, int *y, std::vector<PaintCell> *paintArray) — Функция по порядку добавляет в вектор координаты

отображения узлов. Получает на вход указатель на узел, число, указатель на число и вектор объектов PaintCell, ничего не возвращает.

void addPaintCell(std::vector<PaintCell> *paintArray, int x, int y, char symbol) — Функция добавляет в вектор символ, который должен быть отражен на рисунке структуры данных. Получает на вход указатель на вектор объектов PaintCell, два числа и символ, ничего не возвращает.

void addAdditionalSymbols(std::vector<PaintCell> *paintArray, Node *root) - Функция добавляет на рисунок структуры данных все элементы связывающие части структуры друг с другом. Получает на вход указатель на вектор и указатель на узел, ничего не возвращает.

void paintTree(Node* root) — Главная управляющая функция для создания и вывод рисунка, показывающего структуру данных. Получает на вход указатель на узел, ничего не возвращает.

void clearScreen() - Выводит 100 пустых строк, чтобы отчистить экран терминала. Ничего не получает на вход и ничего не возвращает.

void tapToContinue() - Останавливает программу до ввода пользователем какого-либо символа. Ничего не получает на вход и ничего не возвращает.

void printHufTree(Node* root) — Функция выводит таблицу данных всех узлов дерева кодов. Получает на вход указатель на узел, ничего не возвращает.

4. ТЕСТИРОВАНИЕ

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	q	Вы хотите вводить строку из терминала или текст из файла (1 - файл, 0 - терминал)? Для выхода из программы введите 'q' q 'q' was entered. Finishing program...	Обработка команды пользователя «завершить».
2.	0 ab 0 0	Вы хотите вводить строку из терминала или текст из файла (1 - файл, 0 - терминал)? Для выхода из программы введите 'q' 0 Введите сообщение которое требуется закодировать ab Был введен следующий текст: ab Вы желаете запустить программу в пошаговом или в "упрощенном" режиме (0 - пошаговый, 1 - упрощенный)? * В упрощенном режиме программа выводит лишь минимальное количество пояснений и рисунки измененного дерева после добавления символа. * В пошаговом режиме программа выводит очень подробное описание работы алгоритма на каждом шаге. Для перехода на следующий шаг пользователь должен нажать Enter. * Для обработки больших текстов настоятельно рекомендуется	Алгоритм кодирования в пошаговом режиме

		использовать "упрощенный" режим. 0 Выбран пошаговый режим. Начало кодирования текста. Начинается кодирование символа 'a' Этот символ встречен впервые => его код = путь до NYT + код самого символа из таблицы ascii Путь до NYT = - Код символа = 01100001 Итоговый закодированный символ = 01100001 Кодирование символа закончено. Начинается базовая перестройка дерева Хаффмана. Дерево Хаффмана на данный момент: / ********** ********** / /* № 187: Par = [Null] W = 0 Type = NYT / ********** ********** /	

		Из NYT ([187]) создаются новый NYT, как левый ребенок, и новый лист с обрабатываемым символом, как правый. Старый же NYT преобразуется во внутренний узел. Новосозданный лист помечается, как "увеличиваемый последним". Дерево Хаффмана после базовой перестройки: / ********** ********** / /* № 187: Par = [Null] W = 0 Type = Internal /* № 186: Par = [187R] W = 0 Type = Leaf Symbol = a /* № 185: Par = [187L] W = 0 Type = NYT / ********** ********** / Базовая перестройка дерева закончена. Начинается цикличное увеличение веса узлов с необходимыми перестройками дерева снизу-вверх. Дерево Хаффмана на данный момент: / **********	

		********** / /* № 187: Par = [Null] W = 0 Type = Internal /* № 186: Par = [187R] W = 0 Type = Leaf Symbol = a /* № 185: Par = [187L] W = 0 Type = NYT / ********** ********** / Увеличивается вес у внутреннего узла ([187]), совершая необходимые перестроения дерева Хаффмана. Перестроения дерева не нужны. Изменяется только вес узла. Дерево Хаффмана после изменения: / ********** ********** / /* № 187: Par = [Null] W = 1 Type = Internal /* № 186: Par = [187R] W = 0 Type = Leaf Symbol = a /* № 185: Par = [187L] W = 0 Type = NYT / ********** ********** / Увеличивается вес листа, который был помечен, как "увеличиваемый последним":	

		Увеличивается вес у листа ([186]), совершая необходимые перестроения дерева Хаффмана. Перестроения дерева не нужны. Изменяется только вес самого листа. Дерево Хаффмана после изменения: / ********** ********** / /* № 187: Par = [Null] W = 1 Type = Internal /* № 186: Par = [187R] W = 1 Type = Leaf Symbol = a /* № 185: Par = [187L] W = 0 Type = NYT / ********** ********** / Циклическое увеличение веса узлов с перестройкой дерева закончено. Графическое изображение дерева Хаффмана после добавления символа: >1'a' | 1 | >0 Начинается кодирование символа 'b'	

		Этот символ встречен впервые => его код = путь до NYT + код самого символа из таблицы ascii Путь до NYT = 0 Код символа = 01100010 Итоговый закодированный символ = 001100010 Кодирование символа закончено. Начинается базовая перестройка дерева Хаффмана. Дерево Хаффмана на данный момент: / ********** ********** / /* № 187: Par = [Null] W = 1 Type = Internal /* № 186: Par = [187R] W = 1 Type = Leaf Symbol = a /* № 185: Par = [187L] W = 0 Type = NYT / ********** ********** / Из NYT ([185]) создаются новый NYT, как левый ребенок, и новый лист с обрабатываемым символом, как правый. Старый же NYT преобразуется во внутренний узел. Новосозданный лист помечается, как "увеличиваемый последним".	

		Дерево Хаффмана после базовой перестройки: / ********** ********** / /* № 187: Par = [Null] W = 1 Type = Internal /* № 186: Par = [187R] W = 1 Type = Leaf Symbol = a /* № 185: Par = [187L] W = 0 Type = Internal /* № 184: Par = [185R] W = 0 Type = Leaf Symbol = b /* № 183: Par = [185L] W = 0 Type = NYT / ********** ********** / Базовая перестройка дерева закончена. Начинается цикличное увеличение веса узлов с необходимыми перестройками дерева снизу-вверх. Дерево Хаффмана на данный момент: / ********** ********** / /* № 187: Par = [Null] W = 1 Type = Internal /* № 186: Par = [187R] W = 1	

		Type = Leaf Symbol = a /* № 185: Par = [187L] W = 0 Type = Internal /* № 184: Par = [185R] W = 0 Type = Leaf Symbol = b /* № 183: Par = [185L] W = 0 Type = NYT / ********** ********** / Увеличивается вес у внутреннего узла ([185]), совершая необходимые перестроения дерева Хаффмана. Меняются местами этот внутренний узел и лидер блока листьев веса = весу этого узла + 1. После этого пересоздается правильная нумерация узлов дерева. После перестроения вес узла увеличивается на 1. Дерево Хаффмана после изменения: / ********** ********** / /* № 187: Par = [Null] W = 1 Type = Internal /* № 186: Par = [187R] W = 1 Type = Internal /* № 185: Par = [187L] W = 1 Type = Leaf Symbol = a /* № 184: Par = [186R] W = 0 Type = Leaf Symbol = b /* № 183: Par = [186L] W = 0	

		Type = NYT / ********** ********** / Увеличивается вес у внутреннего узла ([187]), совершая необходимые перестроения дерева Хаффмана. Перестроения дерева не нужны. Изменяется только вес узла. Дерево Хаффмана после изменения: / ********** ********** / /* № 187: Par = [Null] W = 2 Type = Internal /* № 186: Par = [187R] W = 1 Type = Internal /* № 185: Par = [187L] W = 1 Type = Leaf Symbol = a /* № 184: Par = [186R] W = 0 Type = Leaf Symbol = b /* № 183: Par = [186L] W = 0 Type = NYT / ********** ********** / Увеличивается вес листа, который был помечен, как "увеличиваемый последним": Увеличивается вес у листа ([184]), совершая необходимые перестроения дерева Хаффмана.	

		Перестроения дерева не нужны. Изменяется только вес самого листа. Дерево Хаффмана после изменения: / ********** ********** / /* № 187: Par = [Null] W = 2 Type = Internal /* № 186: Par = [187R] W = 1 Type = Internal /* № 185: Par = [187L] W = 1 Type = Leaf Symbol = a /* № 184: Par = [186R] W = 1 Type = Leaf Symbol = b /* № 183: Par = [186L] W = 0 Type = NYT / ********** ********** / Циклическое увеличение веса узлов с перестройкой дерева закончено. Графическое изображение дерева Хаффмана после добавления символа: >1'b' | >1 | | | >0 | 2 | >1'a'	

		Кодирование текста завершено. Итоговый закодированный текст: 01100001001100010 Желаете ли вы запустить процесс декодирования для полученного кода (1 - запустить, 0 - конец программы)? 0 Завершение программы...	
3.	0 ab 1 1	Начало декодирования текста. Первые 8 цифр в последовательности - ascii-код первого встреченного символа. Поэтому сразу распознаем первый символ и перестраиваем дерево Хаффмана. Раскодированный первый символ = 'a'. Декодирование символа закончено.	Алгоритм декодирования в «упрощенном» режиме

		Графическое изображение дерева Хаффмана после добавления первого символа: >1'a' | 1 | >0 Начинается определение следующего символа. Часть кодовой последовательности "0' указывает путь по дереву до узла NYT. Следовательно следующий символ встречается впервые, и следующие 8 символов переданного кода - ascii-код этого символа. Раскодированный символ = 'b'. Декодирование символа закончено. Графическое изображение дерева Хаффмана после добавления символа: >1'b' | >1	

		|| |>0 | 2 | >1'a' Декодирование текста завершено. Итоговый декодированный текст: ab	

4.	a q	Вы хотите вводить строку из терминала или текст из файла (1 - файл, 0 - терминал)? Для выхода из программы введите 'q' a Ввод некорректного символа. Попробуйте снова. q 'q' was entered. Finishing program...	Ввод некорректного пользовательского ввода
5.	1	Вы хотите вводить строку из терминала или текст из файла (1 - файл, 0 - терминал)? Для выхода из программы введите 'q' 1 проблема с открытием файла	Нет файла с текстом
6.	1	Вы хотите вводить строку из терминала или текст из файла (1 - файл, 0 - терминал)? Для выхода из программы введите 'q' 1 В файле расположен следующий текст: Введен пустой текст.	Файл с текстом не содержит ни одного символа

		Завершение программы...	
7.	q	Вы хотите вводить строку из терминала или текст из файла (1 - файл, 0 - терминал)? Для выхода из программы введите 'q' 0 Введите сообщение которое требуется закодировать Был введен следующий текст: Введен пустой текст. Завершение программы...	В терминале введен пустой текст

5. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

В начале выполнения программа спрашивает у пользователя хочет ли он ввести текст клавиатуры, из файла или же он хочет завершить программу и ожидает его ответа. В случае, если пользователь вводит некорректный ответ, то программа сообщает об этом пользователю и еще раз просит его ответить.

Если пользователь продолжил работу с программой, она спрашивают его о том, в каком режиме запустить алгоритмы и дает справку о каждом из режимов. При некорректном ответе пользователя программа укажет на это и запросит ввод еще раз.

При завершении демонстрации одного алгоритма, программа спросит у пользователя разрешение на демонстрацию обратного алгоритма от только что полученного результата.

При выборе пошагового режима работы для перехода на следующий шаг пользователь должен ввести любой символ.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы были изучены методы динамического кодирования и декодирования по Хаффману. Написана программа для демонстрации работы данных алгоритмов на конкретных примерах. Для программы был создан пользовательский интерфейс, позволяющий пользователю выбирать формат ввод данных, режимы выполнения, определять момент завершения программы и вводить некорректные данные.

Результаты полученные при тестировании программы совпадают с теоретическими ожиданиями.

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)
bool skipFlag = true; // если равен 0, то пошаговый режим отменяется, а также сильно сокращается промежуточный вывод (нужно для быстрой проверки рабоспособности и быстрой проверки решения большого текста)
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)


// 3 функции для пошагового режима
void clearScreen() // рисует 100 пустых строк, чтобы очистить экран
{
   if(!(::skipFlag))
      std::cout << std::string( 100, '\n' );
   else
      std::cout << std::string( 6, '\n' );
}

void tapToContinue() // организует пошаговость выполнения функции
{
   if(!(::skipFlag)) // если установлен режим пропуска пошаговой работы
   {
      char ch = std::cin.peek();
      std::cin.ignore(32767,'\n'); // cin очищается
   }
};

void printHufTree(Node* root) // визуализирует структуру дерева Хаффмана
{
   using namespace std;
   if(!(::skipFlag)) // если программа будет рисовать таблицу каждый шаг, то это очень сильно замедлит ее быстрое решение
   {
      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;
      }
   }


// функции используемые в основных функциях

void pushFirst(Node* a) // просто функция-мнемоник
{
   ::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 собвственных 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, а b - правый
      {
         a->m_parent->m_left = b;
         a->m_parent->m_right = a;
      }
      else // если же a - правый ребенок, а 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)
{
   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/NYT
{
   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() // вычищает всю, выделенную динамически, память
{
    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');
};

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;
};

Node* slideAndIncrement(Node* p) // главная дополнительная функция (делает увеличение веса узла на 1 и необходимые смещения в дереве)
{
    if(!(p->m_isLeaf)) // если это внутренний узел
    {
        bool flag = false;
        Node* previousP = p->m_parent;
        if(!(::skipFlag))
            std::cout << "Увеличивается вес у внутреннего узла ([" << p->m_number << "]), совершая необходимые перестроения дерева Хаффмана."<< std::endl;
        tapToContinue();

        // 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 искуственно, начиная с корня
            flag = true;

            if(!(::skipFlag))
```

```
            std::cout << "Меняются местами этот внутренний узел и лидер блока листьев
веса = весу этого узла + 1.\nПосле этого пересоздается правильная нумерация узлов
дерева." << std::endl;
         tapToContinue();

      }

      p->m_weight += 1;

      if(flag)
      {
         if(!(::skipFlag))
            std::cout << "После перестроения вес узла увеличивается на 1." << std::endl;
         tapToContinue();
      }
      else
      {
         if(!(::skipFlag))
            std::cout << "Перестроения дерева не нужны. Изменяется только вес узла." <<
std::endl;
         tapToContinue();
      }
      return previousP;
   }
   else // если же это лист
   {
      bool flag = false;

      if(!(::skipFlag))
         std::cout << "Увеличивается вес у листа ([" << p->m_number << "]), совершая
необходимые перестроения дерева Хаффмана."<< std::endl;
      tapToContinue();

      // 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));
         flag = true;

         if(!(::skipFlag))
            std::cout << "Меняются местами этот лист и лидер блока внутренних узлов
веса = весу этого листа.\nПосле этого пересоздается правильная нумерация узлов
дерева." << std::endl;
         tapToContinue();
      }
      p->m_weight += 1;

      if(flag)
      {
         if(!(::skipFlag))
            std::cout << "После этого перестроения вес листа увеличивается на 1." <<
std::endl;
         tapToContinue();
      }
      else
      {
         if(!(::skipFlag))
```

```
            std::cout << "Перестроения дерева не нужны. Изменяется только вес самого
листа." << std::endl;
            tapToContinue();
        }
        return p->m_parent;
    }
};


// основные функции (кодировщик и декодировщик)

std::string vitterCoder(char symbol)
{
    std::cout << "\t\tНачинается кодирование символа \'" << symbol << "\'" << std::endl;
    tapToContinue();

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

    if (p->m_isNYT) // если этот символ встретился впервые
    {
        std::cout << "Этот символ встречен впервые => его код = путь до NYT + код самого
символа из таблицы ascii" << std::endl;
        tapToContinue();
        // Кодирование символа:

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

        std::cout << "Путь до NYT = " << code;
        if (code == "")
        {
            std::cout << '-';
        }
        std::cout << std::endl;

        // записывает в строку соответствующий код символа
        std::string symbolCode = "";
        fillSymbolCode(symbol, &symbolCode);
        std::cout << "Код символа = " << symbolCode << std::endl;
        tapToContinue();
        code.append(symbolCode);
        std::cout << "Итоговый закодированный символ = " << code << std::endl;
        tapToContinue();

        std::cout << "\t\tКодирование символа закончено." << std::endl;
        tapToContinue();
        clearScreen();



        // Перестройка дерева:
        if(!(::skipFlag))
            std::cout << "\t\tНачинается базовая перестройка дерева Хаффмана." << std::endl;
        tapToContinue();

        if(::internalNodes.size() == 0) // если пока нет ни одного внутреннего узла (в том
числе и корня)
        {
```

```
            if(!(::skipFlag))
                std::cout << "Дерево Хаффмана на данный момент:\n" << std::endl;
            printHufTree(::leafs.at(0));
            tapToContinue();
        }
        else
        {
            if(!(::skipFlag))
                std::cout << "Дерево Хаффмана на данный момент:\n" << std::endl;
            printHufTree(::internalNodes.at(0));
            tapToContinue();
        }



        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

        if(!(::skipFlag))
        {
            std::cout << "Из NYT ([" << p->m_number << "]) создаются новый NYT, как левый
ребенок, и новый лист с обрабатываемым символом, как правый.\nСтарый же NYT
преобразуется во внутренний узел." << std::endl;
            std::cout << "Новосозданный лист помечается, как \"увеличиваемый последним\".\
n" << std::endl;
            tapToContinue();

            std::cout << "Дерево Хаффмана после базовой перестройки:\n" << std::endl;
            printHufTree(::internalNodes.at(0));
            tapToContinue();



            std::cout << "Базовая перестройка дерева закончена." << std::endl;
            tapToContinue();
            clearScreen();
        }
    }
    else // если же символ встречался ранее
    {
        std::cout << "Символ \'" << symbol << "\' уже встречался ранее => его код = путь
до листа, в котором он хранится";
        if(!(::skipFlag))
            std::cout << "([" << p->m_number << "])." << std::endl;
        if(::skipFlag)
            std::cout << '.' << std::endl;
        tapToContinue();
        // Кодирование символа:

        encodeSymbol(&code, p); // записывает в строку путь до листа
        std::cout << "Путь до этого листа и, следовательно, код символа = " << code <<
std::endl;
```

```
        tapToContinue();

        std::cout << "\t\tКодирование символа закончено." << std::endl;
        tapToContinue();
        clearScreen();
        // Перестройка дерева:

        if(!(::skipFlag))
            std::cout << "\t\tНачинается базовая перестройка дерева Хаффмана." << std::endl;
        tapToContinue();

        if(!(::skipFlag))
            std::cout << "Дерево Хаффмана на данный момент:\n" << std::endl;
        printHufTree(::internalNodes.at(0));
        tapToContinue();

        // 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(!(::skipFlag))
            std::cout << "Лист меняется местами с лидером своего блока." << std::endl;

        if ( p->m_parent->m_left->m_isNYT ) // если p после перемещения все равно брат NYT
(это нужно во избежания пары проблем с slideAndIncrement() )
        {
            if(!(::skipFlag))
                std::cout << "Лист после обмена мест с лидером блока остался братом для NYT
=> этот лист помечается, как \"увеличиваемый последним\"." << std::endl;
            tapToContinue();
            nodeForIncrease = p; // лист будет увеличен в конце отдельно
            p = p->m_parent;    // а цикличное увеличение начнется с его родителя
        }

        if(!(::skipFlag))
        {
            std::cout << "Дерево Хаффмана после базовой перестройки:\n" << std::endl;
            printHufTree(::internalNodes.at(0));
            tapToContinue();

            std::cout << "Базовая перестройка дерева закончена." << std::endl;
            tapToContinue();
            clearScreen();
        }
    }

    if(!(::skipFlag))
        std::cout << "\t\tНачинается цикличное увеличение веса узлов с необходимыми
перестройками дерева снизу-вверх." << std::endl;
    tapToContinue();

    if(!(::skipFlag))
        std::cout << "Дерево Хаффмана на данный момент:\n" << std::endl;
    printHufTree(::internalNodes.at(0));
    tapToContinue();
```

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

        if(!(::skipFlag))
            std::cout << "Дерево Хаффмана после изменения:\n" << std::endl;
        printHufTree(::internalNodes.at(0));
        tapToContinue();
        if(!(::skipFlag))
            std::cout << '\n' << std::endl;
    };

    if (nodeForIncrease != nullptr) // делаем еще одно увеличение и сдвиг в дереве, если это
необходимо
    {
        if(!(::skipFlag))
            std::cout << "\tУвеличивается вес листа, который был помечен,
как \"увеличиваемый последним\":" << std::endl;
        tapToContinue();

        slideAndIncrement(nodeForIncrease);

        if(!(::skipFlag))
            std::cout << "Дерево Хаффмана после изменения:\n" << std::endl;

        printHufTree(::internalNodes.at(0));
        tapToContinue();
    };

    if(!(::skipFlag))
        std::cout << "Циклическое увеличение веса узлов с перестройкой дерева
закончено." << std::endl;

    std::cout << "Графическое изображение дерева Хаффмана после добавления
символа:\n" << std::endl;
    paintTree(::internalNodes.at(0));
    tapToContinue();
    clearScreen();

    return code; // возвращаем строку по значению
};

std::string vitterDecoder(std::string *code) // декодирует СТРОКУ кода в сообщение
{
    Node* root = new Node; // использован конструктор для создания первого NYT
    std::string startCode = *code;
    std::string message = "";
    std::string temporaryStr = "";
    int count = 0;
    char symbol;

    ::leafs.clear(); // очищаем от дерева из кодировки, чтобы показать, что функция
самодостаточна
    ::internalNodes.clear();

    ::leafs.push_back(root);
```

```
    std::cout << "Первые 8 цифр в последовательности - ascii-код первого встреченного символа.\nПоэтому сразу распознаем первый символ и перестраиваем дерево Хаффмана." << std::endl;
    tapToContinue();

    // Первыми всегда идут 8 цифр кода символа
    temporaryStr.assign(startCode, 0, 8);
    startCode.erase(0,8);
    symbol = identifySymbol(temporaryStr);

    std::cout << "Раскодированный первый символ = \'" << symbol << "\'." << std::endl;
    tapToContinue();

    std::cout << "\t\tДекодирование символа закончено." << std::endl;
    tapToContinue();
    clearScreen();

    message.push_back(symbol); // первый символ выделен

    if(!(::skipFlag))
        std::cout << "Дерево Хаффмана на данный момент:\n" << std::endl;
    printHufTree(root);
    tapToContinue();

    // 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, указываем, что он больше не NYT

    ::leafs.at(0) = root->m_left; // обновили NYT
    ::leafs.push_back(root->m_right);
    ::internalNodes.push_back(root);

    root->m_weight = 1;
    root->m_right->m_weight = 1;

    if(!(::skipFlag))
        std::cout << "Дерево Хаффмана после изменения:\n" << std::endl;
    printHufTree(root);
    tapToContinue();
    clearScreen();

    std::cout << "Графическое изображение дерева Хаффмана после добавления первого символа:\n" << std::endl;
    paintTree(::internalNodes.at(0));
    tapToContinue();

    while(startCode != "")
    {
        clearScreen();
        std::cout << "\t\tНачинается определение следующего символа.\n" << std::endl;
        tapToContinue();

        std::string temp;
        Node* nextLeaf = findNextLeaf(startCode, &count, root); // находит к какому листу ведет последовательность
        Node* nodeToIncrease = nullptr;
```

```
        temp.assign(startCode, 0 , count);
        std::cout << "Часть кодовой последовательности \"" << temp << "\' указывает путь
по дереву до ";

        if(nextLeaf->m_isNYT) // если последовательность привела к NYT -> встретился новый
символ
        {
            std::cout << "узла NYT.\n Следовательно следующий символ встречается впервые,
и следующие 8 символов переданного кода - ascii-код этого символа." << std::endl;
            startCode.erase(0, count); // удалили путь до NYT
            temporaryStr.assign(startCode, 0, 8);
            startCode.erase(0,8);
            symbol = identifySymbol(temporaryStr);
            message.push_back(symbol);

            std::cout << "Раскодированный символ = \'" << symbol << "\'." << std::endl;
            tapToContinue();

            std::cout << "\t\tДекодирование символа закончено." << std::endl;
            tapToContinue();
            clearScreen();

            // перестройка дерева
            if(!(::skipFlag))
                std::cout << "\t\tНачинается базовая перестройка дерева Хаффмана." <<
std::endl;
            tapToContinue();

            if(!(::skipFlag))
                std::cout << "Дерево Хаффмана на данный момент:\n" << std::endl;
            printHufTree(root);
            tapToContinue();

            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);
            ::internalNodes.push_back(nextLeaf);

            nodeToIncrease = nextLeaf->m_right;
        }
        else // если же последовательность привела к конкретному листу -> символ из этого
листа встретился повторно
        {
            std::cout << "листа, содержащего символ \'" << nextLeaf->m_symbol << "\'.\n" <<
std::endl;

            std::cout << "\t\tДекодирование символа закончено." << std::endl;
            tapToContinue();
            clearScreen();

            // декодирование символа
            startCode.erase(0, count); // удалили путь до листа
            symbol = nextLeaf->m_symbol;
            message.push_back(symbol);
```

```
            Node* k = findBlockLeader(&::leafs, nextLeaf->m_weight);
            if (k != nextLeaf) // на nullptr проверка здесь не нужна т.к. гарантированно есть
хотя бы 1 элемент такого веса (сам p)
                swapNodes(nextLeaf, k);


            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)
        {
            nextLeaf = slideAndIncrement(nextLeaf);
        }
        if(nodeToIncrease != nullptr)
        {
            slideAndIncrement(nodeToIncrease);
        }

        std::cout << "Графическое изображение дерева Хаффмана после добавления
символа:\n" << std::endl;
        paintTree(::internalNodes.at(0));
        tapToContinue();
    }
    return message;
};


// работа с файлами

int readingFile(std::string fileName, std::string *line) // считывает все, что есть в текстовом
файле, в том числе и \n
{
    using namespace std;
    ifstream file(fileName, ios::in | ios::binary); // окрываем файл для чтения
    char ch;

    if (!file)
    {
        cout << "проблема с открытием файла" << endl;
        return 1;
    }

    while(file.get(ch))
    {
        line->push_back(ch);
    }

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

```
int writingFile(std::string fileName, std::string *str) // записывает в файл кодированный текст в виде одной СТРОКИ
{
    using namespace std;
    ofstream out(fileName); // открываем файл для записи
    if (!out)
    {
        cout << "Не получается открыть файл для вывода.\n";
        return 1;
    }
    out << *str;

    out.close();
    return 0;
}



// 4 функции для рисования картинки дерева в терминале (paintTree - главная, ее и надо вызывать)
void setNodesCoordinates(Node* root, int x, int *y, std::vector<PaintCell> *paintArray) // записывает в переданный вектор, где и какой символ рисовать (NYT будет последним в списке)
{
    if(root == nullptr)
        return;



    root->m_x = x;
    x += 2; // различие в отступах от начала строки

    setNodesCoordinates(root->m_right, x, y, paintArray);

    root->m_y = *y;
    (*y) += 2; // разные строки

    paintArray->push_back(PaintCell(root->m_symbol, root->m_x, root->m_y, root->m_weight));

    setNodesCoordinates(root->m_left, x, y, paintArray);

    return;
}

void addPaintCell(std::vector<PaintCell> *paintArray, int x, int y, char symbol) // добавит в вектор в нужное место переданный символ
{
    auto iter = paintArray->begin();
    while(iter != paintArray->end())
    {
        if((*iter).m_y < y)
        {
            iter++;
            continue;
        }
        if((*iter).m_y > y)
        {
            paintArray->emplace(iter, PaintCell(symbol, x, y, -2));
            return;
        }
```

```
        // этот блок начинается, если (*iter).m_y == y
        while((*iter).m_x < x)
        {
            if((*iter).m_y == y)
                iter++;
            else
                break;
        }

        paintArray->emplace(iter, PaintCell(symbol, x, y, -2));
        return;
    }

    std::cout << "Strange thing in add..." << std::endl;
    return;
}

void addAdditionalSymbols(std::vector<PaintCell> *paintArray, Node *root) // добавляет в
вектор для рисования символы ">" и "|"
{
    if(root == nullptr)
        return;

    if((root->m_right) != nullptr)
    {
        for(int i = (root->m_y) - 1; i > root->m_right->m_y; i--)
        {
            addPaintCell(paintArray, (root->m_x) + 1, i, '|');
        }
        addPaintCell(paintArray, (root->m_x) + 1, root->m_right->m_y, '>');
    }
    if((root->m_left) != nullptr)
    {
        for(int i = (root->m_y) + 1; i < root->m_left->m_y; i++)
        {
            addPaintCell(paintArray, (root->m_x) + 1, i, '|');
        }
        addPaintCell(paintArray, (root->m_x) + 1, root->m_left->m_y, '>');
    }

    addAdditionalSymbols(paintArray, root->m_right);
    addAdditionalSymbols(paintArray, root->m_left);
}

void paintTree(Node* root)
{
    std::vector<PaintCell> array;
    int a = 0;
    setNodesCoordinates(root, 0, &a, &array);
    addAdditionalSymbols(&array, root);


    int h = 0, flag = 0; // h - номер строки, flag - откуда отсчитывать пробелы
    auto iter = array.begin();
    while(iter != array.end()) // идем по всем элементам упорядоченного вектора и
печатаем их значения
    {
        if( ((*iter).m_y) > h ) // если следующий элемент на следующей строчке
        {
```

```
            std::cout << std::endl;
            h += 1;
            flag = 0; // показывает, что надо отсчитывать пробелы от начала строчки
            continue;
        }

        for(int k = flag; k < (*iter).m_x; k++)
        {
            std::cout << ' ';
        }

        if((*iter).m_w == -2) // если это символ | или >
        {
            std::cout << (*iter).m_symbol;
            flag = (*iter).m_x + 1;
        }

        else if((*iter).m_symbol == '\n') // чтобы на картинке было видно, что это символ
перехода строки
        {
            std::cout << (*iter).m_w << "\'\\n\'";
            flag = (*iter).m_x + 1;
        }

        else if((*iter).m_symbol == '\0') // чтобы при сохранении результатов работы в файл
не было видно странных символов
        {
            std::cout << (*iter).m_w;
            flag = (*iter).m_x + 1;
        }

        else
        {
            std::cout << (*iter).m_w << '\'' << (*iter).m_symbol << '\'';
            flag = (*iter).m_x + 1;
        }
        iter++;
    }
}


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

    string inputFile = "./text.txt";
    string codedFile = "./codedText.txt";
    string line = "", newLine = "";
    char ch;
    bool fileFlag = false;

    Node *a = new Node;
    pushFirst(a); // вставили начальный NYT

    { // получение ввода и команд от пользователя
        cout << "Вы хотите вводить строку из терминала или текст из файла (1 - файл, 0 -
терминал)?" << endl;
        cout << "Для выхода из программы введите 'q'" << endl;
        while(true)
```

```
        {
            ch = cin.peek();
            if(ch == 'q') // выход из программы
            {
                cin.ignore(32767,'\n');
                cout << "\'q\' was entered. Finishing program..." << endl;
                return 0;
            }
            else if(ch == '1') // ввод из файла
            {
                cin.ignore(32767,'\n');
                fileFlag = true;
                break;
            }
            else if(ch == '0') // ввод из терминала
            {
                cin.ignore(32767,'\n');
                cout << "Введите сообщение которое требуется закодировать" << endl;
                break;
            }
            else // введен какой-то другой символ
            {
                cout << "Ввод некорректного символа. Попробуйте снова.\n" << endl;
                cin.ignore(32767,'\n');
            }
        }

        if(fileFlag) // если ввод идет через файл
        {
            if(readingFile(inputFile, &line)) // считывает все символы в переданную строку
            {
                return 0;
            }
            cout << "В файле расположен следующий текст:\n" << line << '\n' << endl;
        }
        else // если ввод идет через терминал
        {
            cin >> noskipws >> line >> skipws; // считывается строка со всеми разделителями
            cin.ignore(32767,'\n');
            cout << "Был введен следующий текст:\n" << line << '\n' << endl;
        }

        if(line.size() == 0)
        {
            cout << "Введен пустой текст.\nЗавершение программы..." << endl;
            return 0;
        }
        cout << "Вы желаете запустить программу в пошаговом или в \"упрощенном\"
режиме (0 - пошаговый, 1 - упрощенный)?\n" << endl;
        cout << " * В упрощенном режиме программа выводит лишь минимальное
количество пояснений и рисунки\n   измененного дерева после добавления символа." <<
endl;
        cout << " * В пошаговом режиме программа выводит очень подробное описание
работы алгоритма на каждом шаге.\n   Для перехода на следующий шаг пользователь
должен нажать Enter." << endl;
        cout << " * Для обработки больших текстов настоятельно рекомендуется
использовать \"упрощенный\" режим.\n" << endl;
        while(true)
        {
            ch = cin.peek();
            cin.ignore(32767,'\n');
```

```
        if(ch == '0')
        {
            cout << "Выбран пошаговый режим.\n" << endl;
            ::skipFlag = false;
            break;
        }
        if(ch == '1')
        {
            cout << "Выбран \"упрощенный\" режим.\n" << endl;
            ::skipFlag = true;
            break;
        }
        else
            cout << "Ввод некорректного символа. Попробуйте снова.\n" << endl;
    }
}



cout << "\t\tНачало кодирования текста." << endl;
tapToContinue();
clearScreen();

for (int i = 0; i < line.size(); i++) // для каждого символа вызывается кодер
{
    cout << '\n' << endl;
    string a = vitterCoder(line[i]);
    newLine.append(a);
}

cout << "\n\t\tКодирование текста завершено.\n" << endl;

cout << "\t\tИтоговый закодированный текст:\n\n" << newLine << endl; // выводится
итоговая закодированная строка
writingFile(codedFile, &newLine); // сохраняем результат в файл
clearScreen();

{
    cout << "Желаете ли вы запустить процесс декодирования для полученного кода (1
- запустить, 0 - конец программы)?" << endl;
    while(true)
    {
        ch = cin.peek();
        cin.ignore(32767,'\n');

        if(ch == '0')
        {
            cout << "Завершение программы..." << endl;
            deleteAllNodes(); // освобождает память выделенную под дерево хаффмана
            return 0;
            break;
        }
        if(ch == '1')
            break;
        else
            cout << "Ввод некорректного символа. Попробуйте снова.\n" << endl;
    }
}

cout << "\t\tНачало декодирования текста." << endl;
```

```
    tapToContinue();
    clearScreen();

    std::string decodedMessage = vitterDecoder(&newLine); // декодер же просто получает всю строку целиком как результат работу кодировщика

    cout << "\n\t\tДекодирование текста завершено.\n" << endl;

    cout << "\t\tИтоговый декодированный текст:\n\n" << decodedMessage << endl;

    deleteAllNodes(); // освобождает память выделенную под дерево хаффмана
    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
    unsigned int m_weight = 0; // вес узла
    unsigned 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
    int m_x = -1, m_y = -1;

    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
    {};
};

class PaintCell
{
public:
    char m_symbol = '#';
    int m_x = -1;
    int m_y = -1;
    int m_w = -1;
    PaintCell(char s, int x, int y, int w): m_symbol{s}, m_x{x}, m_y{y}, m_w{w}
```

```
   {}
};

void pushFirst(Node* a);
void deleteAllNodes();
void splitHafTree(Node* root, std::vector<Node*> *vec, short int level);
void paintTree(Node* root);
char identifySymbol(std::string ascii);
Node* findNextLeaf(std::string remainCode, int *count, Node* root);

std::string vitterCoder(char symbol);

#endif
```