Занятие - **Методы сжатия данных**

Групповое сжатие. Метод RLE

Рассмотрим изображение, содержащее текст чёрного цвета на сплошном белом фоне. При построчном чтении пикселей такого изображения будут встречаться серии белых (фон) и чёрных (буквы) пикселей. Буквой B обозначим чёрный пиксель, а буквой W — белый.

Рассмотрим некую произвольную строку изображения длиной 51 символ:
WWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWWWW
BWWWWWWWWWWWWWW

Посчитаем количество символов:
1. 4 символа «B»;
2. 47 символов «W».

Итого найдено 5 серий. Заменим серии на число повторов и сам повторяющийся символ:
9W3B24W1B14W

Получилась последовательность из 12 символов. Исходная последовательность состояла из 51 символа. Данные были сжаты в 51/12≈4.25 раза.

Возьмём строку, состоящую из большого количества неповторяющихся символов:
ABCABCABCDDDFFFFFF
После сжатия методом RLE такая строка будет выглядеть так:
1A1B1C1A1B1C1A1B1C3D6F

Исходная строка состоит из 18 символов, а сжатая — из 22. Размер данных увеличился в 22/18≈1.22 раза.

Чтобы после сжатия размер данных не увеличивался, алфавит, в котором записаны длины серий, делят на две части (обычно равные).

Например, алфавит целых чисел можно разделить на две части: положительные и отрицательные числа. Положительные числа используют для записи количества повторов одного символа, а отрицательные — для записи количества неодинаковых символов, следующих друг за другом.

Посчитаем символы с учётом вышесказанного:

- сначала друг за другом следуют 9 не одинаковых символов: «ABCABCABC»;
- затем записаны 3 символа «D»;
- наконец записаны 6 символов «F».

Сжатая строка запишется в виде:

-9ABCABCABC3D6F

Исходная строка состоит из 18 символов, а сжатая — из 15. Размер данных уменьшился в 18/15=1.2 раза.

Допустим, реализация метода RLE для записи длин серий (для подсчёта количества символов) использует переменную целочисленного типа со знаком «signed char». В такую переменную можно записать числа от -128 до 127 включительно. Как же быть, если длина серии равна 128 символам и более? В этом случае серию разделяют на части так, чтобы длина части не превышала 127 символов. Например, серия, состоящая из 256 символов «A», будет закодирована следующей строкой (256=127+127+2):

127A127A2A

Запись на некотором языке программирования алгоритма RLE с учётом этих ограничений нетривиальна.

Конечно, кодирование, которое используется для хранения изображений, оперирует с двоичными данными, а не с символами ASCII, как в рассмотренных примерах, однако принцип остаётся тем же.

1 Алгоритмы сжатия LZ

Лемпель и Зив используют следующую идею:

если в тексте сообщения появляется последовательность из двух ранее уже встречавшихся символов, то эта последовательность объявляется новым символом, для нее назначается код, который при определенных условиях может быть значительно короче исходной последовательности.

В дальнейшем в сжатом сообщении вместо исходной последовательности записывается назначенный код. При декодировании повторяются аналогичные действия и потому становятся известными последовательности символов для каждого кода.

1.1 LZ77 (скользящее окно)

Основная идея этого метода (его еще часто называют методом LZ1, см. [Ziv 77]) состоит в использовании ранее прочитанной части входного файла в качестве словаря.

Кодер создает окно для входного файла и двигает его справа налево в виде строки символов, требующих сжатие. Таким образом, метод основан на скользящем окне. Окно разбивается на две части.

Часть **слева** называется **буфером поиска**.

Она будет служить текущим словарем, и в ней всегда содержатся символы, которые недавно поступили и были закодированы.

Правая часть окна называется **упреждающим буфером**, содержащим текст, который будет сейчас закодирован.

На практике буфер поиска состоит из нескольких тысяч байт, а длина упреждающего буфера равна нескольким десяткам символов.

Вертикальная черта | между символами t и e означает текущее разделение между двумя буферами.

Итак, предположим, что текст «sir_sid_eastman_easily_t» уже сжат, а текст «eases_sea_sick_seals» нуждается в сжатии.

← выход	sir_sid_eastman_easily_t	eases_sea_sick_seals	… ← вход

Кодер просматривает буфер поиска **в обратном порядке** (справа налево) и ищет в нем первое появление символа e из упреждающего буфера.

Он обнаруживает такой символ в начале слова **easily**. Значит, e находится (по смещению) на расстоянии 8 от конца буфера поиска.

Далее кодер определяет, сколько совпадающих символов следует за этими двумя символами e. В данном случае совпадение по символам eas имеет длину 3. Кодер продолжает поиск, пытаясь найти более длинные совпадающие последовательности. В нашем (случае имеется еще одна совпадающая последовательность той же длины 3 в слове eastman со смещением 16.

Декодер выбирает самую длинную из них, а при совпадении длин - самую удаленную, и готовит метку (16, 3, «e»).

Выбор более удаленной ссылки упрощает кодер. Интересно отметить, что выбор ближайшего совпадения, несмотря на некоторое усложнение программы, также имеет определенные преимущества. При этом выбирается наименьшее смещение.

Может показаться, что в этом нет преимущества, поскольку в метке будет предусмотрено достаточно разрядов для хранения самого большого смещения. Однако, можно следовать LZ77 с кодированием Хаффмана или другого статистического кодирования меток, при котором более коротким смещениям присваиваются более короткие коды. Этот метод, предложенный Бернардом Хердом (Bernard Herd), называется LZH. Если получается много коротких смещений, то при LZH достигается большее сжатие.

Следующую идею очень легко усвоить. Декодер не знает, какое именно совпадение выбрал кодер, ближайшее или самое дальнее, но ему это и не надо знать! Декодер просто читает метки и использует смещение для нахождения текстовой строки в буфере поиска. Для него не важно, первое это совпадение или последнее.

В общем случае, метка из LZ77 имеет три поля:

- смещение

- длина

- следующий символ в упреждающем буфере (в нашем случае, это будет второе e в слове teases).

Эта метка записывается в выходной файл, а окно сдвигается вправо (или входной файл сдвигается влево) на четыре позиции: три позиции для совпадающей последовательности и одна позиция для следующего символа.

… sir_sid_eastman_easily_tease	s_sea_sick_seals …

Если обратный поиск не обнаружил совпадение символов, то записывается метка со смещением 0 и длиной 0. По этой же причине метка будет иметь третью компоненту. Метки с нулевыми смещением и длиной всегда стоят в начале работы, когда буфер поиска пуст или почти пуст.

Первые семь шагов кодирования нашего примера приведены ниже.

Словарь	Буфер упреждающий		Коды символов, которые надо было кодировать (они не присутствовали до этого в тексте словаря)
	sir_sid_eastman_	⇒	(0,0,«s»)
s	ir_sid_eastman_e	⇒	(0,0,«i»)
si	r_sid_eastman_ea	⇒	(0,0, «r»)
sir	_sid_eastman_eas	⇒	(0,0,«_»)
sir_	sid_eastman_easi	⇒	(4,2,«d»)
sir_sid	_eastman_easily	⇒	(4,1,«e»)
sir_sid_e	astman_easily_te	⇒	(0,0,«a»)

Пример 1. Формирование сжатия текста методом скользящего окна

LZ77 использует скользящее по сообщению окно. Не использует словарь.

Текст: cabababababm

Содержимое окна (сжимаемый текст)	Содержимое упреждающего буфера	Код
	cabababababm	<0,0,c>
c	*abababababm*	<0,0,a>
ca	*bababababm*	<0,0,b>
cab	*abababm*	<2,2,a>
cabab	*ababm*	<2,4,m>

Результат сжатия
<0,0,c><0,0,a><0,0,b><2,2,a><2,4,m>

Пример 2 Сжатие последовательности битов по алгоритму LZ

Алгоритм кодирования

Первоначально каждому символу алфавита присваивается определенный код (коды - порядковые номера, начиная с 0).

1. Выбирается первый (**один**) символ сообщения и заменяется на его код.

2. Выбираются следующие **два** символа и заменяются своими кодами.

При этом комбинации двух символов присваивается свой код. Обычно это номер, равный числу уже использованных кодов. Так, если алфавит включает 8 символов, имеющих коды от 000 до 111, то первая двух символьная комбинация получит код 1000, следующая - код 1001 и т.д.

3. Выбираются из исходного текста очередные 2, 3,...N символов до тех пор, пока не образуется еще не встречавшаяся комбинация. Тогда этой комбинации присваивается очередной код, и поскольку совокупность A из первых N-1 символов уже встречалась, то она имеет свой код, который и записывается вместо этих N-1 символов. Т.е. можно представить формирование кода в этом случае так:

xxxxxxxx код из N(=8) символов выбран из кодируемой последовательности, тогда если для первых N-1 символов уже был сформирован код, то заменяем эти N-1 символы на их код. Каждый акт введения нового кода назовем шагом кодирования.

4. Процесс продолжается до исчерпания исходного текста.

Текст (последовательность битов):

00000001111111111110000000000011011110

Пояснения к составлению кода по алгоритму

Содержимое окна (сжимаемый текст)	Содержимое упреждающего буфера	Код назначенный последовательности
	00000001111111111110000000000011011110	
0	00000011111111111110000000000011011110	0
00	00001111111111111000000000011011110	10
000	01111111111110000000000011011110	11
01	11111111111000000000011011110	100
11	1111111110000000000011011110	101
111	11111000000000011011110	110
1111	11000000000011011110	111
110	00000000011011110	1000
0000	0000011011110	1001
00000	11011110	1010
1101	1110	1011
1110		1100

После первых двух выборов код сжатой последовательности:

Код: 0.00

При выборе 000 так как код для 00 уже построен, то в результирующую строку (код строки) вставим код: заменим первые два нуля его кодом – 10 (согласно таблицы), получим 100 – это сжатие 3-х нулей

Код: 0.00.100

При выборе последовательности которой еще код не назначен – например, 01, ей присваивается код – номер следующего кода – так как это четвертый код, то в двоичной системе 100. При этом код имеет длину три, т.е. в результате получим в результирующей строке замену 01 на 001.

Код: 0.00.100.001

При выборе последовательности 11 такой под последовательности еще не было, она уникальна. Присваиваем ей код 101 и так как заменять не на что, оставляем ее такой как есть, но в код вставляем 011, расширяя код до длины 3, в соответствии с разрядностью предыдущего кода.

Код: 0.00.100.001.011

Выбранная под последовательность 111 уже не уникальная, ее первые 11 имеют код 110, заменяем 11 на код 110 и получаем значение для кода в строку кода всего текста 1011 – уже код стал 4-х разрядный. Следующие за ним коды будут такой длины, пока не образуется код длиной 5.

Код: 0.00.100.001. 011.1011

Для последовательности 1111 заменим 111 на соответствующий ей код - 110 и получим 1101 и вставим в код

Код: 0.00.100.001.011.1011.1101

Для последовательности 110 заменим 11 на код 101, получим 1010

Код: 0.00.100.001.011.1011.1101.1010

Для последовательности 0000 заменим 000 на код последовательности 11, получим 110, но расширяем до длины 5 получаем 00110

Код: 0.00.100.001.011.1011.1101.1010. 00110

Для последовательности 00000 заменим 0000 на код последовательности 1001, получим 10010

Код: 0.00.100.001.011.1011.1101.1010.00110.10010

Для последовательности 1101 заменим110 на код последовательности 1000, получим 10001

Код: 0.00.100.001.011.1011.1101.1010.00110.10010.10001

Для последовательности 1110 заменим 111 на код последовательности 110, расширим длину до пяти разрядов получим 01100

Код: 0.00.100.001.011.1011.1101.1010.00110.10010.10011.01100

Исходный текст	0.00.000. 01. 11. 111.1111. 110. 0000.00000. 1101. 1110.
LZ-код	0.00.100.001.011.1011.1101.1010.00110.10010.10001.01100.
R (разрядность)	2 3 4 5
Вводимые коды	- 10 11 100 101 110 111 1000 1001 1010 1011 1100

1.2 Метод LZ78

В отличие от LZ77, работающего с уже полученными данными, LZ78 ориентируется на данные, которые только будут получены (LZ78 не использует скользящее окно, он хранит **словарь** из уже просмотренных фраз).

Алгоритм считывает символы сообщения до тех пор, пока накапливаемая подстрока входит целиком в одну из фраз словаря.

Как только эта строка перестанет соответствовать хотя бы одной фразе словаря, алгоритм генерирует код, состоящий из индекса строки в словаре, которая до последнего введенного символа содержала входную строку, и символа, нарушившего совпадение.

Затем в словарь добавляется введенная подстрока. Если словарь уже заполнен, то из него предварительно удаляют менее всех используемую в сравнениях фразу. Если в конце алгоритма мы не находим символ, нарушивший совпадения, то тогда мы выдаем код в виде (**индекс строки в словаре без последнего символа, последний символ).**

Пример 1. Дан текст ababaaabb. Сжать текст, используя метод LZ78
Словарь

Ссылка на символ	Символы словаря	код
1	a	<0,a>
2	b	<0,b>
3	ab	<1,b>
4	aa	<1,a>
5	abb	<3,b>

Содержимое словаря	Содержимое считанной строки	Код
a	a	<0,a>
a, b	b	<0,b>
a, b, ab	ab	<1,b>
a, b, ab ,aa	aa	<1,a>
a, b, ab ,aa, abb	abb	<3,b>

Код 0a0b1b1a3b

Пример 2. Сжать текст cababababababm. Другая форма таблицы.

Содержимое словаря		Содержимое считанной строки	Код
	1	*c*	<0,c>
c,	*2*	*a*	<0,a>
c,a	*3*	*b*	<0,b>
c, a, b	*4*	*ab*	<2,b>
c, a, b, ab	*5*	*aba*	<4,a>
c, a, b, ab, aba	*6*	*ba*	<3,a>
c, a, b, ab, aba, ba	*7*	*bab*	<6,b>
c, a, b, ab, aba, ba, abab		*m*	<0,m>

Результат сжатия: 0c0a0b2b4a3a5b0m

2 Сжатие кодами переменной длины

Префиксный код

2.1 Алгоритм метода Шеннона-Фано

— один из первых алгоритмов сжатия, который впервые сформулировали американские учёные Шеннон и Фано, и он имеет большое сходство с алгоритмом Хаффмана.

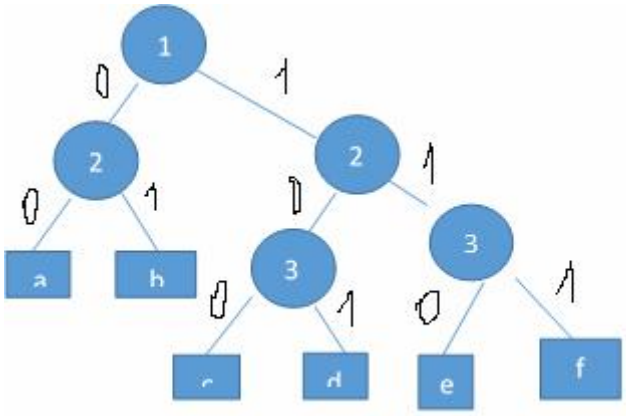
Алгоритм основан на частоте повторения символов. Так, часто встречающийся символ кодируется кодом меньшей длины, а редко встречающийся — кодом большей длины.
В свою очередь, коды, полученные при кодировании, префиксные. Это и позволяет однозначно декодировать любую последовательность кодовых слов. Но все это вступление. Для работы оба алгоритма должны иметь таблицу частот элементов алфавита.

1. Алгоритм Шеннона-Фано работает следующим образом:
   На вход приходят упорядоченные по невозрастанию частот данные.
2. Находится середина, которая делит алфавит примерно на две части. Эти части (суммы частот алфавита) примерно равны. Для левой части присваивается «1», для правой «0», таким образом мы получим листья дерева
3. Шаг 2 повторяется до тех пор, пока мы не получим единственный элемент последовательности, т.е. листок

Таким образом, видно, что алгоритм Хаффмана как бы движется от листьев к корню, а алгоритм Шеннона-Фано, используя деление, движется от корня к листям.

Пример 1. Построение Дерева Фано-Шеннона

Алфавит	Частота		Код	Кол. Байтов(ASCII)	Количество бит
a	10		00	10	2
b	8		01	8	2
c	6		100	6	3
d	5		101	5	3
e	4		110	4	3
f	3		111	3	3
				36 байтов	16 бит (2 байта)



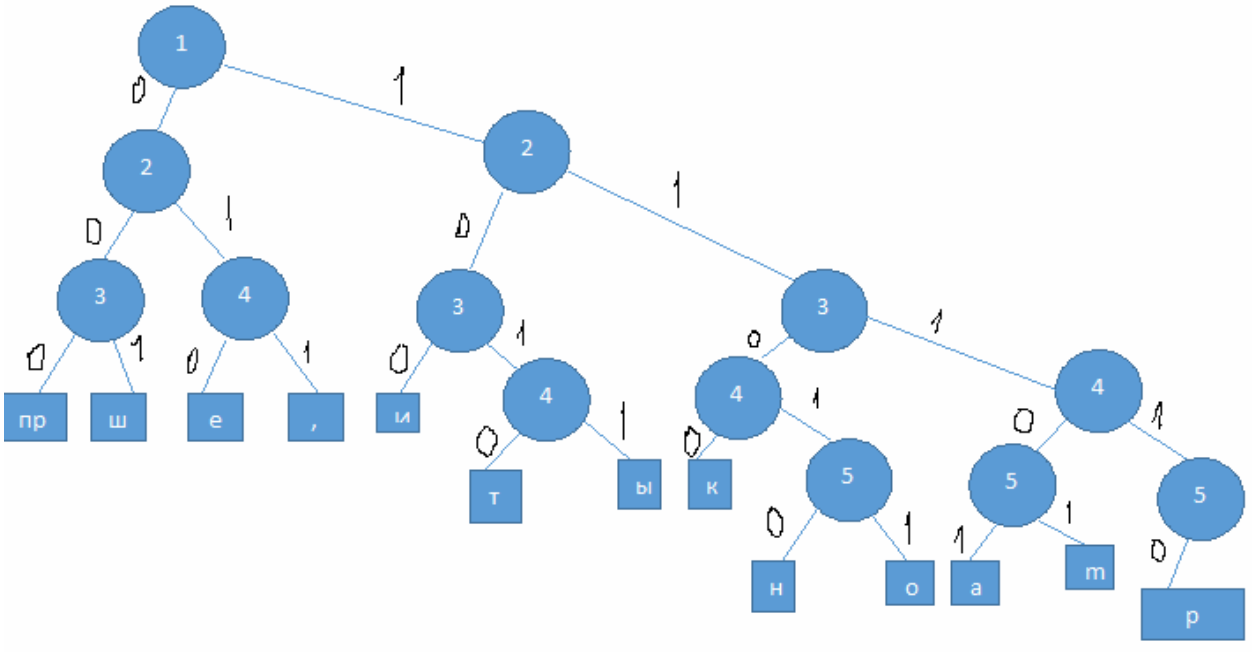
Пример 2, Закодировать фразу «Тише, мыши, тише, кот на крыше», используя метод Шеннона–Фано.

Таблица 1

Символ	Кол-во	1-я цифра	2-я цифра	3-я цифра	4-я цифра	5-я цифра	Код	Кол-во бит
пробел	5	0	0	0			000	15
ш	4	0	0	1			001	12
е	3	0	1	0			010	9
,	3	0	1	1			011	9
и	3	1	0	0			100	9
т	3	1	0	1	0		1010	12
ы	2	1	0	1	1		1011	8
к	2	1	1	1	0		1100	8
н	1	1	1	1	1		11010	4
о	1	1	1	0	0	0	11011	5
а	1	1	1	0	0	1	11100	5
м	1	1	1	0	1	0	11101	5
р	1	1	1	0	1	1	11110	5
								106

Незакодированная фраза – 30*8 бит = 240 бит.

Закодированная фраза –106 бит.



Код алгоритма формирования таблицы кодов Фано

```
#include <string>
#include <iostream>
using namespace std;
//string a = "abcdef";
//char ch[] = { 'a','b','c','d','e','f' };
//int frequency[] = {10,8,6,5,4,3 };
string a = " se,itukhoamp";
char ch[] = { ' ','s','e',',','i','t','u','k','h','o','a','m','p'};
int frequency[] = { 5,4,3,3,3,3,2,2,1,1,1,1,1 };
void TreeShennon(char branch, string full_branch, int start_pos,int end_pos) {
	// Среднее значение массива
	//m - номер средней буквы в последовательности, S - сумма чисел, левой ветки
	int i, m, S;
	string c_branch;		// текущая история поворотов по веткам
	//проверка если это вход нулевой то очистить историю
	if (a != "")
		c_branch = full_branch + branch;
	else c_branch = "";
	// Критерий выхода : если позиции символов совпали, то это конец
	if (start_pos == end_pos) {
		cout<<a[start_pos]<< "="<< c_branch;
		return;
	}

	//Подсчет среднего значения частоты в последовательности
	double dS = 0;
	for (int i = start_pos; i < end_pos; i++) {
		dS = dS + frequency[i];
	}
	dS= dS / 2;
	//поиск середины
	S = 0; i = start_pos; m = i;
	while (S + frequency[i] < dS && i < end_pos) {
		S = S + frequency[i];
		i++; m++;
	}
	// Рекурсия левая ветка дерева }
	TreeShennon('0', c_branch, start_pos, m);
	// Правая ветка дерева }
	TreeShennon('1', c_branch, m + 1, end_pos);

}

int main()
{
	TreeShennon(' ', "", 0, 13);
   return 0;
}
```

2.2 Алгоритм Хаффмана работает следующим образом:

Код Хаффмана - это особый тип **оптимального префиксного кода**, который обычно используется для сжатия данных без потерь. Он сжимает данные, очень эффективно экономя от 20% до 90% памяти, в зависимости от характеристик сжатых данных.

Кодовый символ – это наименьшая единица данных, подлежащая сжатию.

Кодовое слово – это последовательность кодовых символов из алфавита кода.

Префиксный код – это код, в котором никакое ***кодовое слово*** не является префиксом любого другого кодового слова.

Оптимальный префиксный код – это *префиксный код*, имеющий минимальную среднюю длину.

Кодовое дерево (дерево кодирования Хаффмана, Н-дерево) – это *бинарное дерево*, у которого: *листья* помечены символами, для которых разрабатывается *кодировка*; узлы (в том числе корень) помечены суммой вероятностей появления всех символов, соответствующих листьям *поддерева*, корнем которого является соответствующий узел.

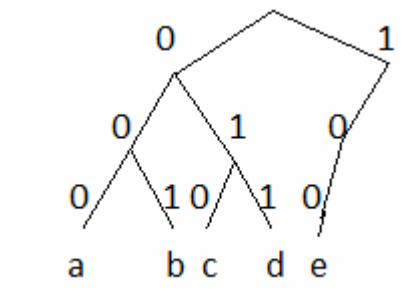
2.2.1 Пример. Создать дерево кодовое дерево Хаффмана для построения кодов символов.

На вход поступает текст, который надо сжать.

Результатом должна быть таблица кодов символов. Коды символов будут представлены последовательностью битов.

По алгоритму Хаффмана строится префиксное дерево – бинарное дерево, представляющее коды символов с префиксным свойством.

Пример такого дерева



На левых ребрах бит с кодом 0, на правых бит с кодом 1. Тогда код буквы a - 000, а код буквы c – 010. Вот примерно такое дерево надо создать и по нему сформировать коды символов.

Алгоритм построения кодового оптимального дерева.

Оптимальное – так как коды символов будут минимальными по длине. Символы, частота которых большая, должны иметь код минимального размера.

1. Надо создать частотный словарь: алфавит символов текста с указанием частоты появления символов в тексте. Предположим, что такой словарь был создан, и содержит следующие данные.

Последовательность символов и частота их появления в тексте

a	b	c	d	e
12	40	15	8	25

2. Выбираются две наименьших по частоте буквы алфавита, и создается родитель (сумма двух частот этих «листков»). Каждый символ будет листом в кодовом дереве.

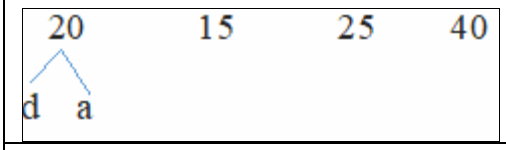
3. Потомки удаляются и вместо них записывается родитель, «ветви» родителя нумеруются: левой ветви ставится в соответствие «1», правой «0».

4. Шаг два повторяется до тех пор, пока не будет найден главный родитель — «корень».

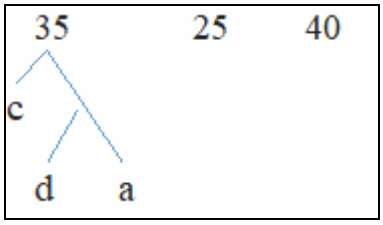
Рассмотрим построение кодового дерева Хаффмана дерева для приведенного примере.

1. Создаем массив из частот символов

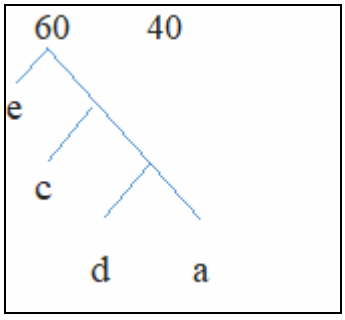
2. Выбираем два наименьших по частоте элемента, они будут листьями первого дерева. Создаем из них дерево (это символы a и d с частотами 12 и 8), суммируем их частоту и возвращаем в массив частот. В результате первого шага получаем. Символ с наименьшей частотой из двух выбранных значений образует левое поддерево, а символ с большей частотой – правое.



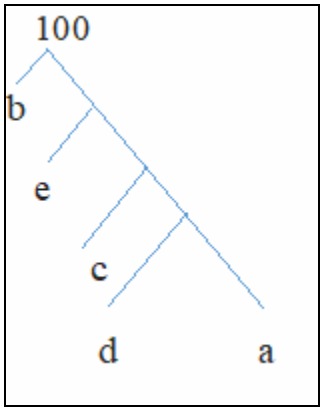
3. Снова выбираем два наименьших по частоте элемента из массива частот, они будут листьями второго дерева. Создаем из них дерево (это символ **c** и корень нового дерева с частотой 20), суммируем их частоту и возвращаем в массив частот. В результате второго шага получаем



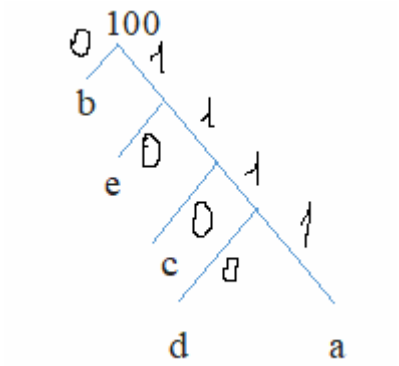
4. Снова выбираем два наименьших по частоте элемента из массива частот, они будут листьями третьего дерева. Создаем из них дерево (это символ **e** и корень нового дерева с частотой 35), суммируем их частоту и возвращаем в массив частот. В результате третьего шага получаем



5. Снова выбираем два наименьших по частоте элемента из массива частот, они будут листьями третьего дерева. Создаем из них дерево (это символ **b** и корень нового дерева с частотой 60), суммируем их частоту и возвращаем в массив частот. В результате четвертого шага получаем



Можно теперь нанести на него коды битов: на левую ветвь 0 на правую ветвь 1. Можем сформировать код каждого символа, обходя дерево сверху вниз. Видно что код символа b, который имеет самую большую частоту, имеет длину 1 бит и равен 0.



2.3 Реализация алгоритма построения кодового дерева

Для реализации дерева воспользуемся его представлением на основе таблицы и для связи элементов будем использовать курсор (индексы таблицы). Для хранения алфавита так же создадим таблицу и там будем хранить символ, частоту его появления в тексте, ссылку на узел – лист в кодовом дереве, так как все символы – это листы этого дерева.

Структуры данных для реализации

1) Кодовое дерево Tree– таблица вида. Столбцы хранят ссылки(индексы) на узлы дерева (т.е. на строки этой таблицы): leftT, rightT – индексы сыновей родителя parent. Первые 5 строк для данных о листьях – их родителях в дереве.

	leftT	rightT	parent
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

2) Алфавит с частотой и ссылкой на лист в дереве, меткой которого он является

	Символ	Частота	Ссылка На лист
1	a	12	1
2	b	40	2
3	c	15	3
4	d	8	4
5	e	25	5

3) Лес – таблица для представления всех деревьев. root – корень дерева – листа – индекс строки в дереве.

	root	Частота
1	1	12
2	2	40
3	3	15
4	4	8
5	5	25

2.3.1 Выполнение алгоритма на таблицах

1) Упорядочим таблицу Лес по возрастанию частот и будем его использовать в качестве массива частот. После упорядочения два наименьших элемента теперь две первых строки таблицы. Создаем из них дерево

	root	Частота
1	4	8
2	1	12
3	3	15
4	5	25
5	2	40

2) Создадим дерево из наименьших значений – первые два. Добавим узел в кодовое дерево. Получим

	leftT	rightT	parent
1	0	0	6
2	0	0	0
3	0	0	0
4	0	0	6
5	0	0	0
6	4	1	0

3) Изменим лес – включим дерево с частотой равной сумме двух частот первых двух деревьев и его корень в дереве – это номер новой строки, в нашей таблице это 6. В дереве вставим в поле родителя узлов с номерами 1 и 4 ссылку на их дерево

	root	Частота
1	6	20
2	3	15
3	5	25
4	2	40

	leftT	rightT	parent
1	0	0	6

2	0	0	0
3	0	0	0
4	0	0	6
5	0	0	0
6	4	1	0
7	3	6	0

4) Снова строим дерево для двух символов с наименьшими частотами. Это так же первые два символа: левое с частотой 15, правое с частотой 20.

Добавляем новое дерево в кодовое дерево и меняем родителя предыдущего поддерева

	leftT	rightT	parent
1	0	0	6
2	0	0	0
3	0	0	7
4	0	0	6
5	0	0	0
6	4	1	7
7	3	6	0

Изменяем лес

	root	Частота
1	5	25
2	7	35
3	2	40

5) Снова строим дерево для двух символов с наименьшими частотами. Это так же первые два символа: левое с частотой 25, правое с частотой 35.

Добавляем новое дерево в кодовое дерево и меняем родителя предыдущего поддерева

	leftT	rightT	parent
1	0	0	6
2	0	0	0
3	0	0	7
4	0	0	6
5	0	0	8
6	4	1	7
7	3	6	8
8	5	7	0

Изменяем лес

	root	Частота
1	8	60
3	2	40

6) Снова строим дерево для двух символов с наименьшими частотами. Это так же первые два символа: левое с частотой 25, правое с частотой 35.

Добавляем новое дерево в кодовое дерево и меняем родителя предыдущего поддерева

	leftT	rightT	parent
1	0	0	6
2	0	0	9
3	0	0	7
4	0	0	6
5	0	0	8
6	4	1	7
7	3	6	8
8	5	7	9
9	2	8	0

Изменяем лес

	root	Частота
1	2	40
3	8	60

Изменяем лес

	root	Частота
1	9	100

7) Кодовое дерево построено, когда все узлы из леса включены в дерево. В Лесе осталось одно значение, но информацию мы по нему уже включили в дерево.

8) Как по дереву создать код символа?

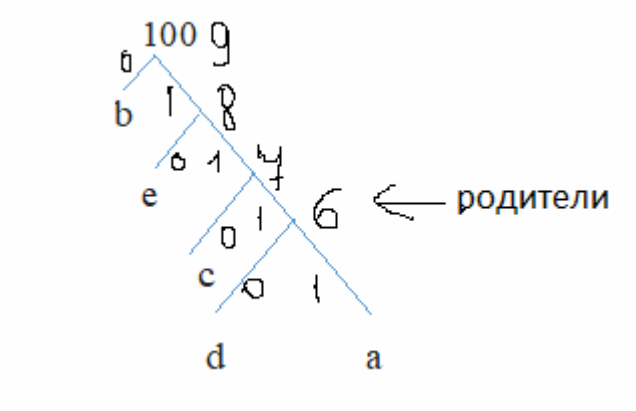
В таблице алфавита хранится символ и ссылка на его лист в дереве, мы такую информацию сразу заложили в таблицу алфавита. Выбрав эту ссылку, можно найти корень (родителя) дерева этого символа (листа) в дереве (строка с индексом равным ссылке). После этого по ссылке родителя отправляемся в узел родителя и строим код поднимаясь по ветвям к корню дерева.

Полученная последовательность битов будет собрана в обратном порядке по отношению к самому коду, т.е. ее надо перевернуть наоборот. Но подобрав для реализации соответствующую структуру, можно сразу код собрать в прямом порядке.

Найдем код символа a:

- В таблице алфавита находим символ и находим его лист в кодовом дереве, выбираем родителя – это строка 6.
- В строке 6 ищем его корень – т.е. 1 и в код заносим 1-так как ссылка правая.
- Далее в следующей строке (7 – спускаемся) ищем ссылку на родителя узла с корнем 1, т.е 6 – тоже выписываем код – 1 – так как 6 в поле правой ссылки, т.е. уже 11.
- В следующей строке ищем родителя 7, тоже 1, код 111.
- В следующей строке ищем код родителя 8 – тоже 1, код 1111.

Посмотрим на оптимальное кодовое дерево и увидим, как это мы выполняли.



2.3.2 Код построения дерева и формирование кода входного текста в битовом формате

```
// Haffman.cpp: определяет точку входа для консольного приложения.
//

#include "stdafx.h"
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>

using namespace std;
struct recordTree {
	int left;
	int right;
	int parent;
};
struct recordTableAlphabet {
	char symbol;
	int frequency;//частота (вероятность) появления символа в тексте
	int leaf;//курсор - ссылка на элемент в др.таблице
};
//запись списка деревьев
struct recordForest {
	int frequency;
	int root;
};
vector<recordTableAlphabet> AL;
list<recordForest> forest;
vector<recordTree>  tree;
//сравнение частот, используется алгоритме sort из STL, при выборе двух
//наименьших значений из Forest
bool f(recordForest &x, recordForest &y) {
	return x.frequency < y.frequency;
}
//массив для определения символов текста ASCII и их частот
//используется в нижеследующей функции
recordTableAlphabet A[256];
//Создание алфавита текста подлежащего сжатию
//на вход поступает текст, и возвращает таблицу –алфавит и параллельно
//количество элементов в таблице
vector<recordTableAlphabet> createTableAlphabet(string s, int &k) {
```

```
	int n = 256;
	vector<recordTableAlphabet> Alphabet;
	for (int i = 0; i<s.length(); i++) {
		A[(int)s[i]].symbol = s[i];
		A[(int)s[i]].frequency++;
	}
	k = 0;
	for (int i = 0; i<n; i++) {
		if (A[i].frequency) {
			A[i].leaf = k;
			Alphabet.push_back(A[i]);
			k++;
		}
	}
	//sort(Alphabet.begin(), Alphabet.end());
	return Alphabet;
}
//создание леса на основании алфавита
//возвращает лес – как список записей
//список выбран в связи с удобством удаления после выбора //двух первых
элементов, создании нового и включении его в  //список
list<recordForest> createForest(vector<recordTableAlphabet> AL) {
	list<recordForest> F;
	recordForest f;
	for (int i = 0; i < AL.size(); i++) {
		f.frequency= AL[i].frequency;
		f.root = AL[i].leaf;
		F.push_back(f);
	}
	return F;
}
//создание дерева – таблицы на основе вектора из записей //дерева
vector<recordTree> createTree(int n) {
	vector<recordTree> T(n);
	for (int i = 0; i < T.size(); i++) {
		T[i].left=0; T[i].right = 0; T[i].parent = 0;
	}
	return T;
}
void printTableAlphabet(vector<recordTableAlphabet> AL, int n);
void printTree(vector<recordTree> T);
void printForest(list<recordForest> F);
//построение кодового дерева по рассмотренному алгоритму
//используются: forest, tree – глобальные переменные
```

```
void createHafman() {
	list<recordForest>::iterator itr;
	while (forest.size()>1) {
		forest.sort(f);
		recordTree rt;
		itr = forest.begin();
		recordForest f1 = *itr;
		recordForest f2 = *(++itr);
		//добавление элемента в дерево
		rt.left = f1.root;
		rt.right = f2.root;
		rt.parent = 0;
		tree.push_back(rt);

		f1.frequency = f1.frequency + f2.frequency;
		//корректируем Tree ссылки на символы
		tree[f1.root].parent = tree.size()-1;
		tree[f2.root].parent = tree.size()-1;
		f1.root = tree.size()-1;
		//удаляем первые два обработанных, вставляем новый суммарный
		forest.pop_front(); forest.pop_front();
		forest.push_front(f1);
		printForest(forest);
	}
	printTree(tree);

}
//создание таблицы кодов символов
//на входе таблица с алфавитом AL и дерево T
//результат: таблица, реализованная на основе вектора, с //записями типа
symbol_kod
//для представления кода как последовательность битов //используется
вектор с типом элемента bool vector<bool>
struct symbol_kod {
	char symbol;
	vector<bool> kod;
};
list<symbol_kod> Kods(vector<recordTableAlphabet> AL, vector<recordTree> T)
{
	string steck;
	list<symbol_kod> lst;
	symbol_kod sk;
	int root, root1;
	for (int i = 0; i < AL.size(); i++) {
		sk.kod.clear();
```

```
                root = T[AL[i].leaf].parent;
                if (T[root].left == i)
                        sk.kod.push_back(0);
                if (T[root].right == i)
                        sk.kod.push_back(1);
                root1 = T[root].parent;
                while(root1!=0){

                        if (T[root1].left == root)
                                sk.kod.push_back(0);
                        if (T[root1].right == root)
                                sk.kod.push_back(1);
                        root = root1;
                        root1 = T[root].parent;
                }
                sk.symbol = AL[i].symbol;


                lst.push_back(sk);
        }
        return lst;
}
//функция формирования тестового текста по рассмотренному //примеру
string createText();
void printKods(list<symbol_kod> F);

//сжатие текста с применением созданных кодов
//на вход поступает текст, таблица с кодами
//коды в битовом формате
vector<vector<bool>> kompression(string t, list<symbol_kod> lst);

void resultkompterrion(vector<vector<bool>> V);

int main()
{
        string text = createText();
        int n;

        AL=createTableAlphabet(text,n);
        printTableAlphabet(AL, n);
        forest= createForest(AL);
        printForest(forest);
        tree = createTree(AL.size());
        printTree(tree);
        createHafman();
```

```
        list<symbol_kod> kods =Kods(AL, tree);
        printKods(kods);
        text = "aaccddee";
        vector<vector<bool>> V = kompression(text, kods);
        resultkompterrion(V);

    return 0;
}
string createText() {
string s;
for (int i = 1; i <= 12; i++) {
        s += 'a';
}
for (int i = 1; i <= 40; i++) {
        s += 'b';
}
for (int i = 1; i <= 15; i++) {
        s += 'c';
}
for (int i = 1; i <= 8; i++) {
        s += 'd';
}
for (int i = 1; i <= 25; i++) {
        s += 'e';
}
return s;
}
//поиск в таблице с кодами кода символа
//на входе символ и таблица, результат код символа как //последовательность
битов
vector<bool> findeKod(char ch, list<symbol_kod> lst) {
        list<symbol_kod>::iterator itr;
        for (itr = lst.begin(); itr != lst.end(); ++itr) {
                if( (*itr).symbol ==ch)
                        return (*itr).kod;
        }
}
//сжимает переданный текст и возвращает это текст в виде //вектора,
содержащего последовательность битов //закодированного текста
vector<vector<bool>> kompression(string t, list<symbol_kod> lst) {
        vector<vector<bool>> v;
        vector<bool> vv;
        int k = 0;
        for (int i=0; i<t.length(); i++){
                vv = findeKod(t[i], lst);
```

```
                v.push_back(vv); k++;
        }
        return v;
}
void printTableAlphabet(vector<recordTableAlphabet> AL, int n) {
        for (int i = 0; i < AL.size(); i++) {
                std::cout << AL[i].symbol << " " << AL[i].frequency << " " <<
AL[i].leaf << endl;
        }
}
void printTree(vector<recordTree> T) {
        for (int i = 0; i < T.size(); i++) {
                std::cout << T[i].left << " " << T[i].right << " " << T[i].parent <<
endl;
        }
}
void printForest(list<recordForest> F) {
        list<recordForest>::iterator itr;
        cout << endl;
                for (itr = forest.begin(); itr!=forest.end(); ++itr) {
                        cout << (*itr).frequency << " " << (*itr).root<<endl;
        }
}
void printKods(list<symbol_kod> F) {
        list<symbol_kod>::iterator itr;
        cout << endl;
        for (itr = F.begin(); itr != F.end(); ++itr) {
                cout << (*itr).symbol << " ";
                for (int i = 0; i < (*itr).kod.size(); i++)
                        cout << (*itr).kod.at(i);
                cout << endl;
        }
}
void resultkompterrion(vector<vector<bool>> V) {
        int k = 0,m=0;
        string s = "";
        for (int i = 0; i < V.size(); i++) {
                for (int k = 0; k < V[i].size(); k++)
                std::cout << V[i].at(k);
                k++;
        }

}
```