

Reading: GT 9.3, pp. 440–442 [up to 30 minutes]

The most important thing to understand is how Huffman code is constructed,. Can you construct such a code manually for any string of characters that I will give to you? Can you apply the Huffman code to compress this string?

Ingredients: Intro, Huffman code example, Algorithm, Complexity

Huffman coding is a popular general purpose lossless compression algorithm.

Intro

The purpose of data *compression algorithms* is to decrease the number of space a given piece of information uses (email, multimedia, executables, any file).

Historically primary application was to decrease required *storage space*. Nowadays it is also very important to decrease the communication *bandwidth* (think for example streaming audio and video).

Lossless compression is reversible, i.e. it guarantees that the result of compression and decompression is identical to the original data. Lossless compression is essential for most application (word processing documents, economic data, vector graphics, *source code* of programs, executables).

Lossy compression changes the data slightly to achieve higher compression rates. Such data after decompression is not identical to the original. Lossy compression is used in certain specific domains, when small changes are acceptable (mostly in graphics, video, and audio).

A Historical Note. The algorithm, which we are studying today, was published by David A. Huffman in 1952. It is a *lossless* algorithm; used in bzip2 (a popular Unix compressor). It is often used as the last stage of lossy compression scheme, among others in JPEG and MPEG.

DEFLATE is a derivative of Huffman coding, used in gzip (the .gz file format) and by various applications producing .zip files.

The history of the .zip format goes back to the shareware application PKZIP released in 1989 for DOS. Nowadays the most popular application using this

format is WinZip. Java archives (jar) are actually .zip archives—try to unzip to see what is inside!. So are Firefox extensions (xpi files).

Competing lossless compression algorithms are *arithmetic coding* and *Lempel-Ziv-Welch* (LZW). The latter is used in the popular GIF format. This also explains why GIF files are usually larger than JPEG files containing the same image. GIF is compressing images in a lossless way.

For this reason GIF (and PNG) are preferred for storing vector graphics in bitmap form (JPEG produces dirty noise on such images).

Huffman Coding

An example. We want to compress:

a_fast_runner_need_never_be_afraid_of_the_dark

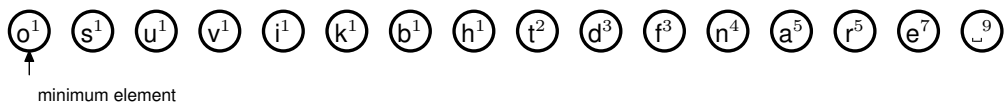
In Huffman code various symbols are mapped to different amounts of bits. The mapping is determining by path in a binary tree. So the first part of the compression is devoted to building this tree.

Below we present the complete derivation for the example of Figure 9.16 in the book¹

1. First, compute the frequency histogram for the input file:

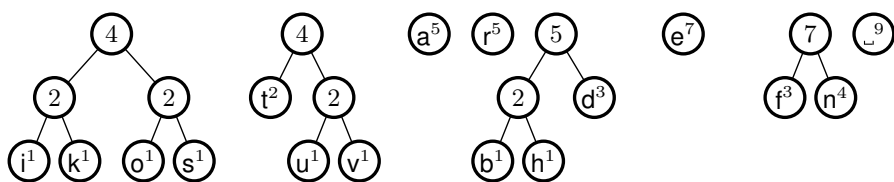
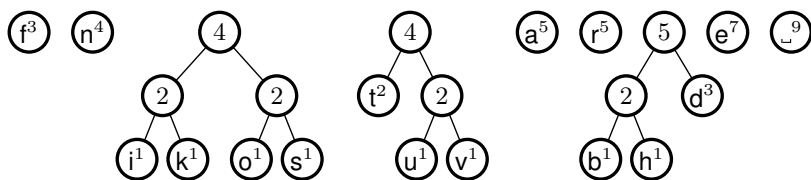
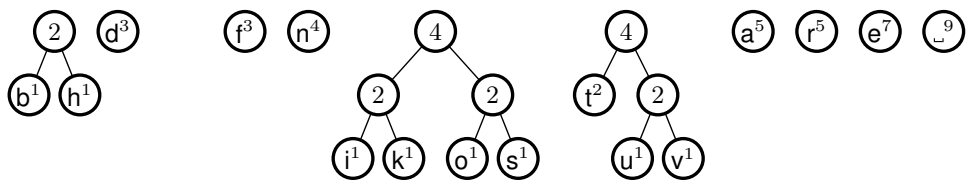
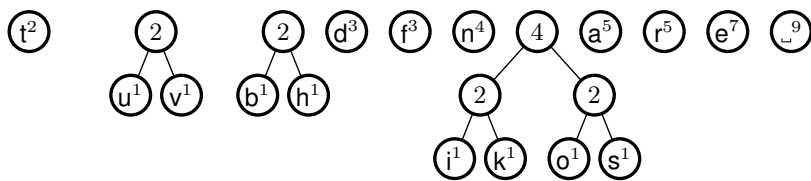
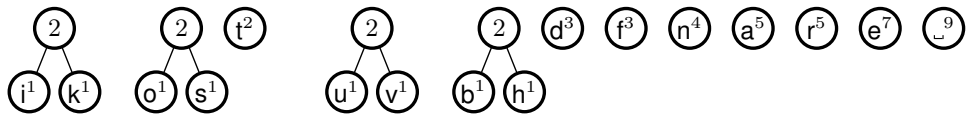
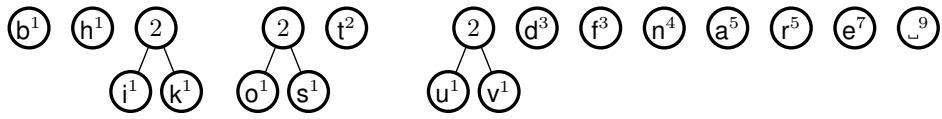
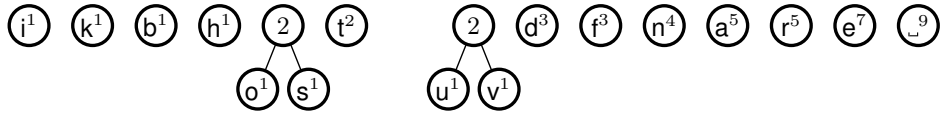
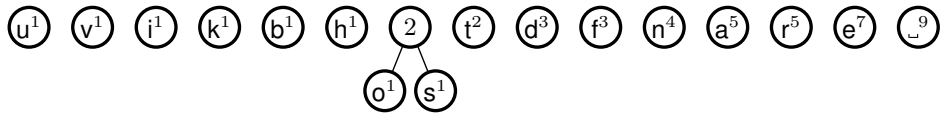
symbol	_	a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

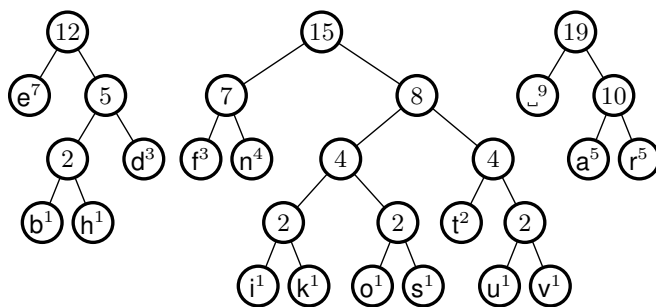
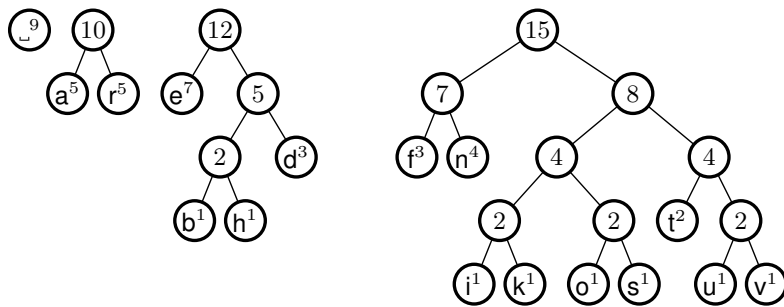
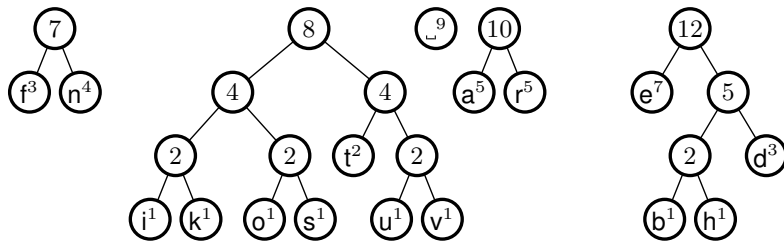
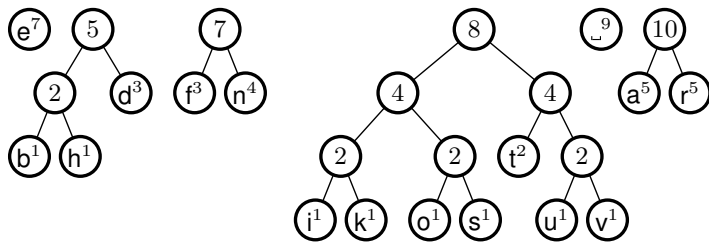
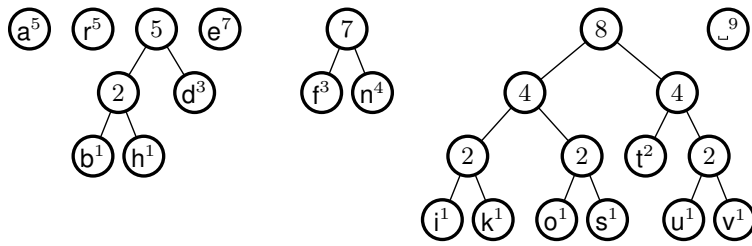
2. For each symbol create a tree with the symbol being its only node. Put the trees in a priority queue with frequencies of symbols being the keys.

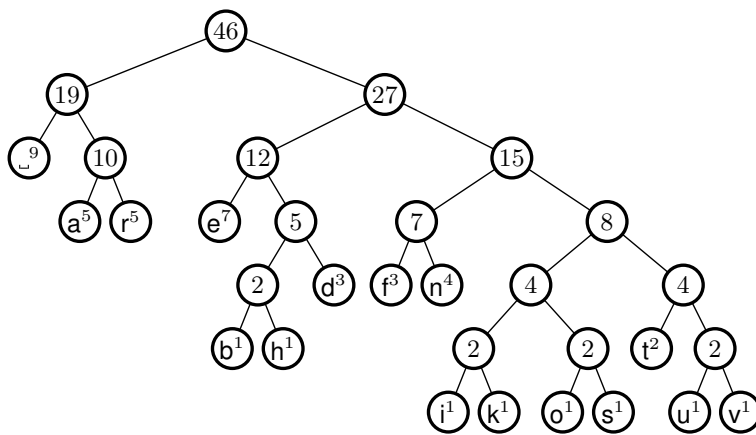
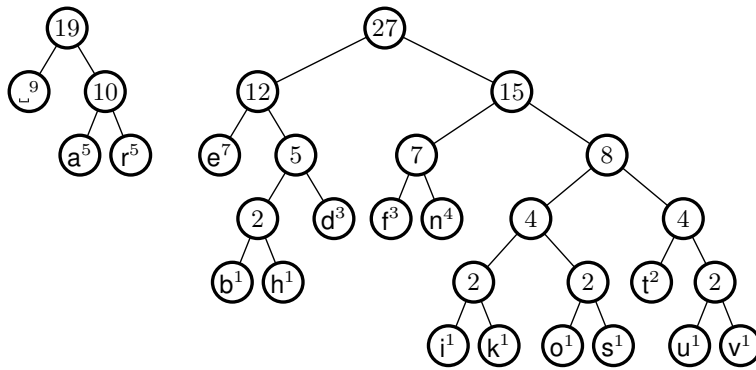


3. If the priority queue contains more than one tree, then remove two minimum key trees, merge them into a single tree (adding a new parent) with key equal to the sum of the original keys.

¹Note that the book Figure has at least one inconsistency with algorithm in p. 442. *d* cannot be the left subtree of *b*, *h*, since it has higher frequency. This has been fixed in this note.







4. Once the tree is constructed we code the input by following the path from the root for every symbol. We generate 0 whenever we follow the left branch, and 1 to denote the right branch.

Our input is compressed to:

010 00 1100 010 111011 11110 00 011 111110 1101 1101

a l f a s t l r u n n

100 001 00 1101 100 100 1011 00 1101 100 111111 100 111110

e r l n e e d l n e v e r

00 10100 100 00 010 1100 001 010 111000 1011 00 111010

l b e l a f r a i d l o

1100 00 11110 10101 100 00 1011 010 011 111001

f l t h e l d a r k

5. Divide the bit stream into 8-bit chunks obtaining:

```
01000110 00101110 11111100 00111111 10110111
01100001 00110110 01001011 00110110 01111111
00111110 00101001 00000101 10000101 01110001
01100111 01011000 01111010 10110000 10110100
11111001
```

The input line: 46 bytes. Compressed: 21 bytes.

Complexity

Let d be the number of symbols, n be the length of the input.

Theorem 1. *Huffman's algorithm runs in $O(n + d \log d)$ time.*

The Algorithm

HUFFMAN(X)

- 1 Compute frequency $f(c)$ for each character c in X .
- 2 Let Q be an empty priority queue
- 3 Insert every character c into Q as singleton trees with key $f(c)$
- 4 **while** Q .SIZE() > 1
- 5 **do** $f_1 \leftarrow Q$.MIN-KEY()
- 6 $T_1 \leftarrow Q$.REMOVE-MIN()
- 7 $f_2 \leftarrow Q$.MIN-KEY()
- 8 $T_2 \leftarrow Q$.REMOVE-MIN()
- 9 Let T be a new tree with left subtree T_1 and right subtree T_2
- 10 Q .INSERT($T, f_1 + f_2$)
- 11 **return** Q .REMOVE-MIN()