Encoding data : Huffman algorithm

Alphabetic characters used to be stored in *ASCII* on computers, which requires 7 bits per character.

ASCII is a **fixed-length code**, since each character requires the same number of bits to store.

Encoding data : Huffman algorithm

Notice that some characters, (e.g. **q**, **x**, **z**, **v**) are rare, and others (e.g. **e**, **s**, **t**, **a**) are common.

It might make more sense to use less bits to store the common characters, and more bits to store the rare characters.

An encoding that does this is called a variable-length code.

A code is called optimal if the space required to store data with the given distribution is a minimum.

Optimal codes are important for many applications.

Variable Length Code Example

Assume a file has the following distribution of characters.

Letter	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13

One good encoding might be the following :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

Thus, the string *treat* is encoded as *110111011*

The problem with this encoding is that the string could also be *ktve*.

Variable Length Code Example

Assume a file has the following distribution of characters.

Letter	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13

One good encoding might be the following :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

With this code, we have to somehow keep track of which letter is which.
(e.g. *11_01_1_10_11*)

To do this requires more space, and may make the code worse than a fixed-length code. Rather we use a prefix code.

Prefix Codes

A code in which no word is a prefix of another word.

To encode a string of data, concatenate the codewords together.

To decode, just read the bits until a codeword is recognized.

Since no codeword is a prefix of another, this works.

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

Notice that no codeword is the prefix of another.

Now, we can encode *treat* as *1001010100100*, and it is uniquely decodable.

Decoding prefix codes

Decode *01111011010010100100* based on the following decoding table :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

The most obvious way is to read one bit, see if it is a character, read another, see if the two are a character, etc. : *not very efficient*.

Decoding prefix codes

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

A better way is to represent the encoding of each letter as a path a binary tree :

- Each leaf node stores a character.
- A '0' means go to the left child
- A '1' means go to the right child
- The process continues until a leaf is found.

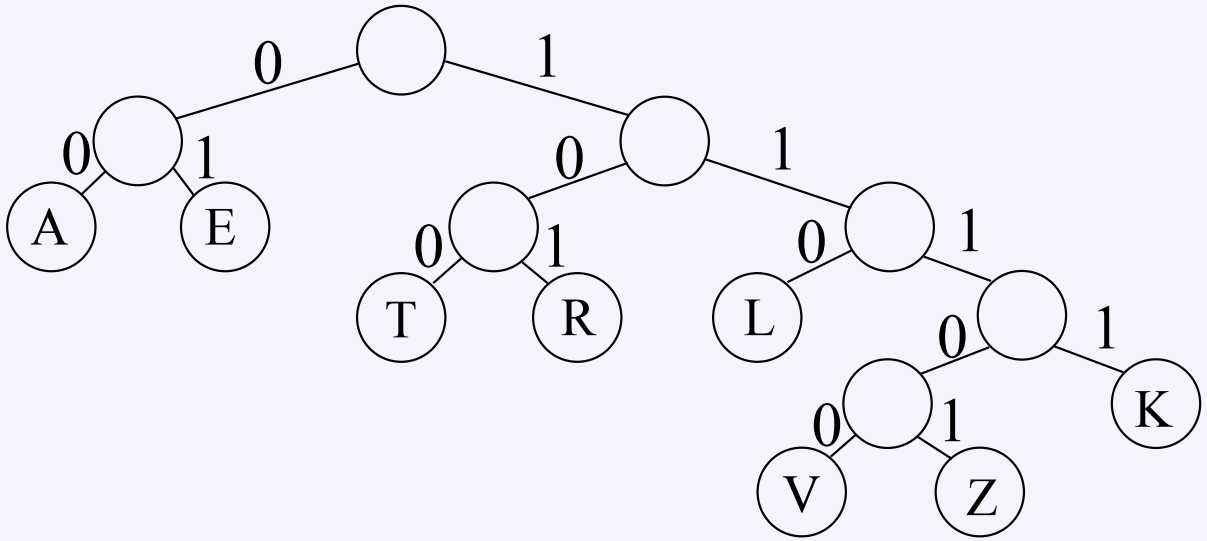
Any code represented this way is a prefix code. Why ?

Decoding prefix code

Decode *01111011010010100100*

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

We will represent the data by the following binary tree :



It is now not too hard to see that the answer is *ezrarat*.

Constructing Codes

Prefix codes make encoding and decoding data very easy.

It can be shown that optimal data compression using a character code can be obtained using a prefix code.

How can we construct such a code ?

Huffman code is an algorithm to construct prefix codes

Huffman Code

A Huffman code can be constructed by building the encoding tree from the bottom-up in a greedy fashion.

Since the less frequent nodes should end up near the bottom of the tree, it makes sense that we should consider these first.

We'll see an example, then the algorithm.

Huffman Code Example

Suppose I want to store this very long word in an electronic dictionary : *floccinaucinihilipilification*.

I want to store it using as few bits as possible.

The frequency of letters, sorted in decreasing order, is as follows :

i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

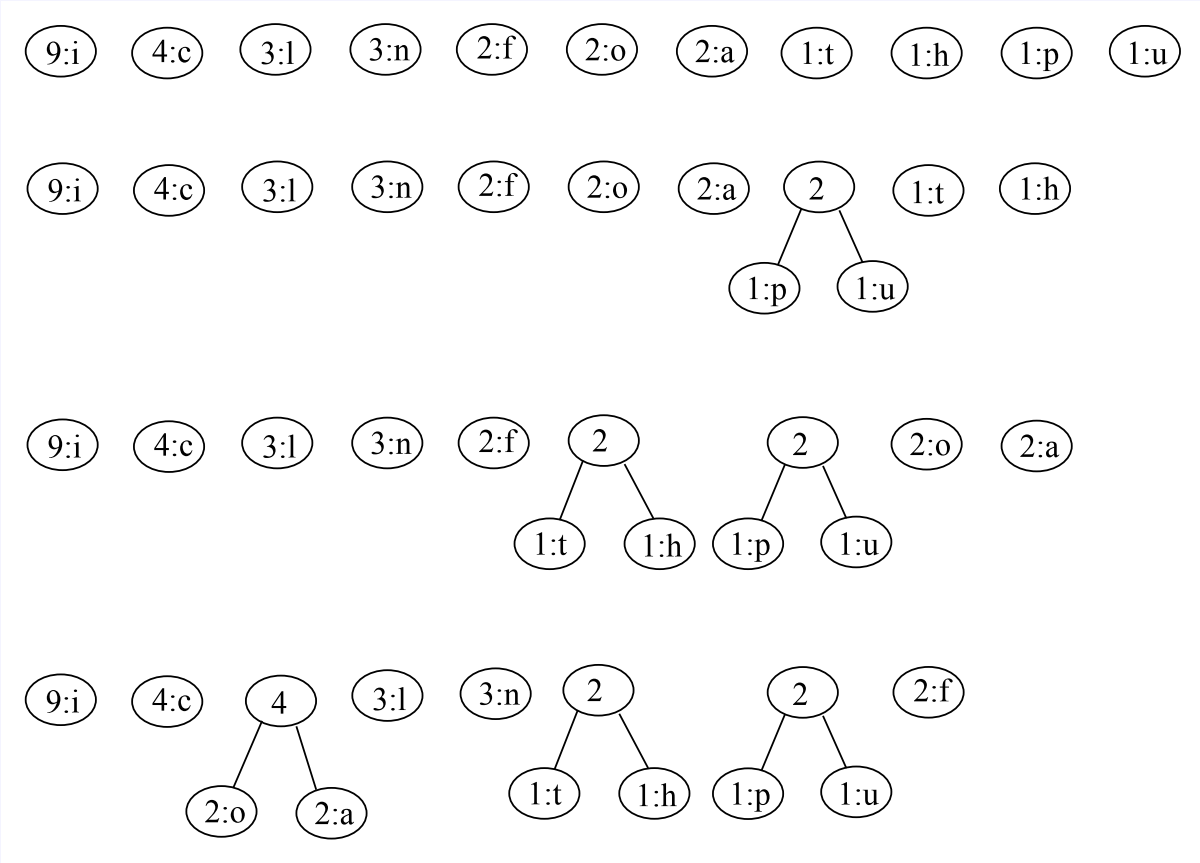
Huffman Code Example

i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

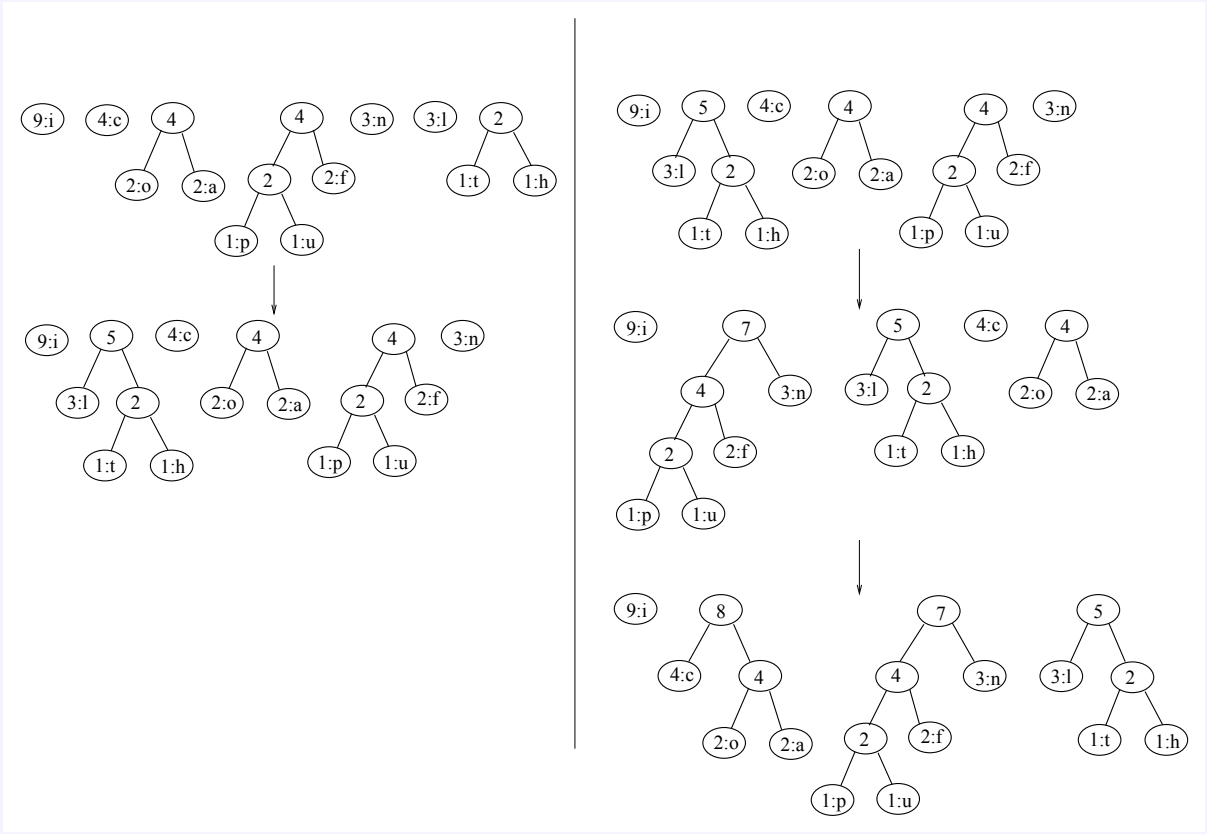
The algorithm works sort of like this :

- Consider each letter as a node in a not-yet-constructed tree.
- Label each node with its letter and frequency.
- Pick two nodes x and y of least frequency.
- Insert a new node, and let x and y be its children. Let its frequency be the combined frequency of x and y.
- Take x and y off the list.
- Continue until only 1 node is left on the list.

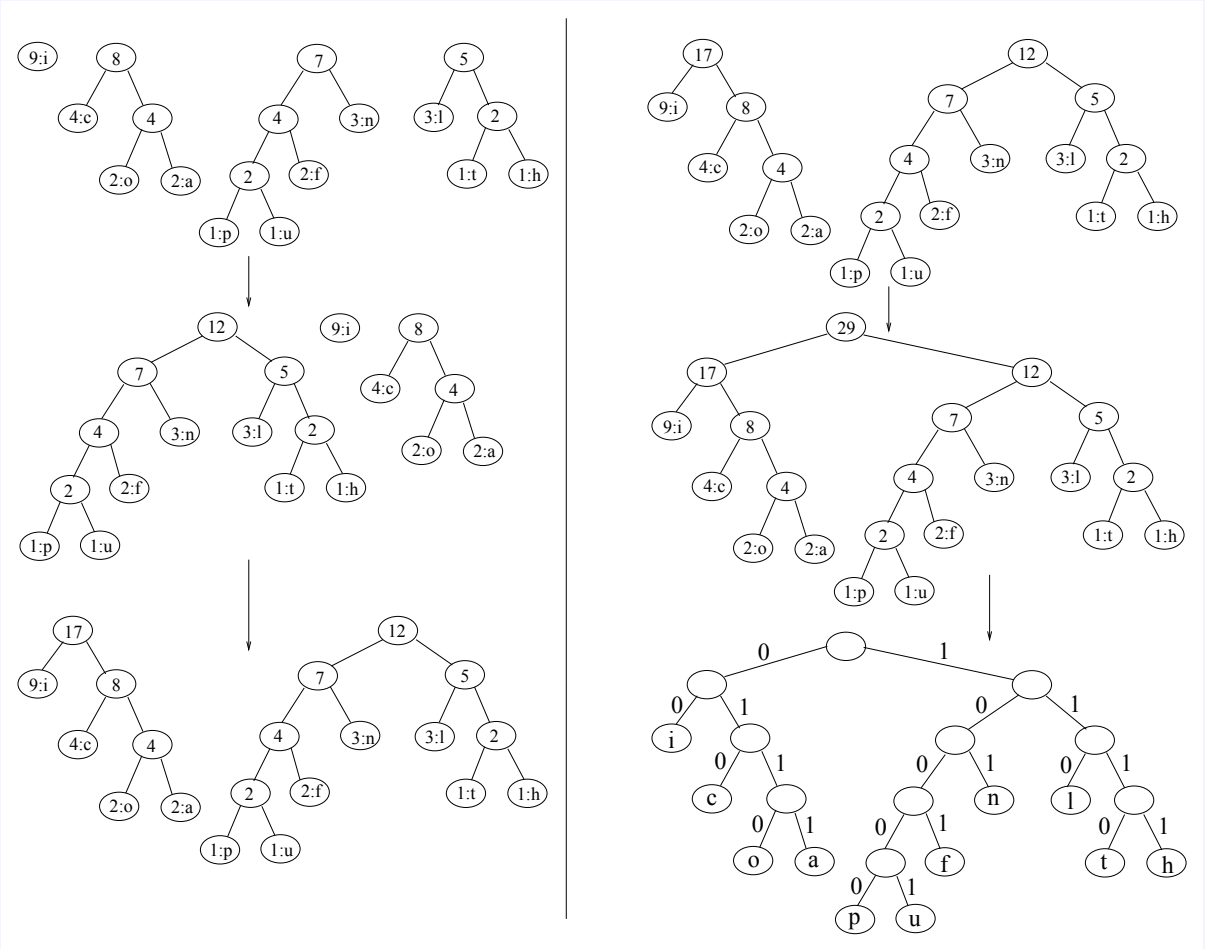
Huffman Example



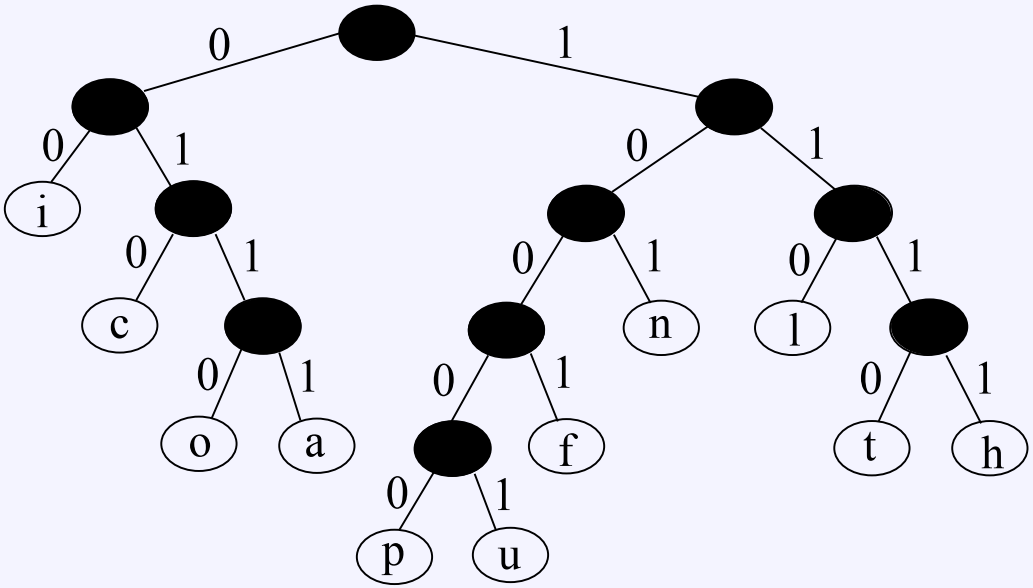
Huffman's encoding



Huffman's encoding



We can now list the code :



i (9)	00	f (2)	1001
c (4)	010	t (1)	1110
n (3)	101	h (1)	1111
l (3)	110	p (1)	10000
o (2)	0110	u (1)	10001
a (2)	0111		