Huffman Coding:

In computer science and information theory, a Huffman code is an optimal prefix code found using the algorithm developed by David A. Huffman "A Method for the Construction of Minimum-Redundancy Codes". The process of finding such a code is called Huffman coding and is a common technique in entropy encoding, including in lossless data compression. The algorithm's output can be viewed as a variable-length code table for encoding a source symbol. Huffman's algorithm derives this table based on the estimated probability or frequency of occurrence for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in linear time to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

Huffman Coding Example 1:

Construct a Huffman tree by using these nodes.

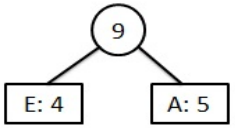
Value	A	B	C	D	E	F
Frequency	5	25	7	15	4	12

Solution:

Step 1: According to the Huffman coding we arrange all the elements (values) in ascending order of the frequencies.

Value	E	A	C	F	D	B
Frequency	4	5	7	12	15	25

Step 2: Insert first two elements which have smaller frequency.



Value	C	EA	F	D	B
Frequency	7	9	12	15	25

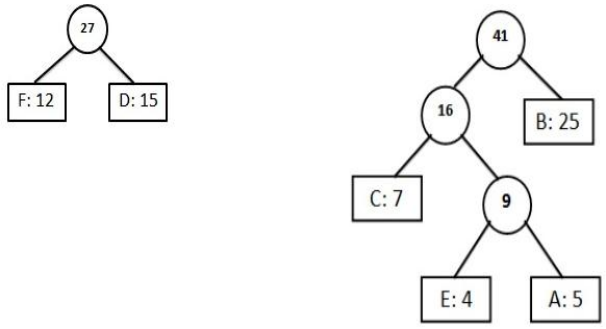
Step 3: Taking next smaller number and insert it at correct place.

Value	C	EA	F	D	B
Frequency	7	9	12	15	25

Step 4: Next elements are F and D so we construct another subtree for F and D.

Value	CEA	B	FD
Frequency	16	25	27

Step 5: Taking next value having smaller frequency then add it with CEA and insert it at correct place.



Value	FD	CEAB
Frequency	27	41

Step 6: We have only two values hence we can combined by adding them.

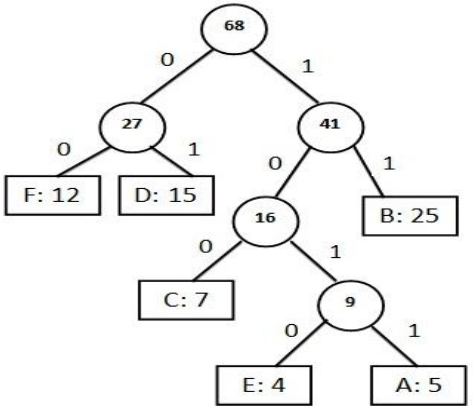


Fig. Huffman Tree:

Value	FDCEAB
Frequency	68

Now the list contains only one element i.e. FDCEAB having frequency 68 and this element (value) becomes the root of the Huffman tree.

Huffman Coding Example 2:

Suppose that we have a 100,000 character data file that we wish to store . The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

A *binary code* encodes each character as a binary string or *codeword*. We would like to find a binary code that encodes the file using as few bits as possible, ie., *compresses it* as much as possible.

In a *fixed-length code* each codeword has the same length. In a *variable-length code* codewords may have different lengths. Here are examples of fixed and variable legth codes for our problem (note that a fixed-length code must have at least 3 bits per codeword).

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
a fixed-length	000	001	010	011	100	101
a variable-length	0	101	100	111	1101	1100

The fixed length-code requires $300,000$ bits to store the file. The variable-length code uses only

$(45{\cdot}1+13{\cdot}3+12{\cdot}3+16{\cdot}3+9{\cdot}4+5{\cdot}4){\cdot}1000 = 224,000$ bits,

Encoding:

Given a code and a message it is easy to encode the message. Just replace the characters by the codewords.

Example: $\Gamma = \{a, b, c, d\}$

If the code is

$$C_1\{a = 00,\ b = 01,\ c = 10,\ d = 11\}.$$

then bad is encoded into 010011

If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

then bad is encoded into 1100111

Decoding:

$$\begin{aligned} C_1 &= \{a = 00,\ b = 01,\ c = 10,\ d = 11\}. \\ C_2 &= \{a = 0,\ b = 110,\ c = 10,\ d = 111\}. \\ C_3 &= \{a = 1,\ b = 110,\ c = 10,\ d = 111\} \end{aligned}$$

Given an encoded message, *decoding* is the process of turning it back into the original message.
A message is *uniquely decodable* (vis-a-vis a particular code) if it can only be decoded in one way.

For example relative to C_1, 010011 is uniquely decodable to bad.
Relative to C_2 1100111 is uniquely decodable to bad.
But, relative to C_3, 1101111 is not uniquely decipherable since it could have encoded either bad or acad.

In fact, one can show that every message encoded using C_1 and C_2 are uniquely decipherable. The unique decipherability property is needed in order for a code to be useful.

Fixed-length codes are always uniquely decipherable (why).
We saw before that these do not always give the best compression so we prefer to use variable length codes.

Prefix Code: A code is called a prefix (free) code if no codeword is a prefix of another one.

Example: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$ is a prefix code.

Important Fact: Every message encoded by a prefix free code is uniquely decipherable. Since no codeword is a prefix of any other we can always find the first codeword in a message, peel it off, and continue decoding. Example:

$$01101100 = 01101100 = abba$$

We are therefore interested in finding good (best compression) prefix-free codes.

Fixed length vs variable length codes:

Problem: Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150$$

a 25% saving.

Huffman Coding

Step 1: Pick two letters x, y from alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea)
Label the root of this subtree as z.

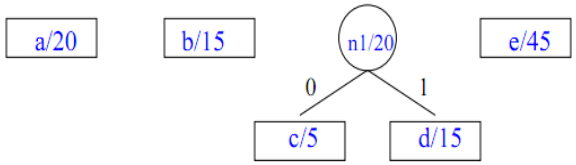
Step 2: Set frequency $f(z) = f(x) + f(y)$.
Remove x, y and add z creating new alphabet $A' = A \cup \{z\} - \{x, y\}$.
Note that $|A'| = |A| - 1$.

Repeat this procedure, called *merge*, with new alphabet A' until an alphabet with only one symbol is left.

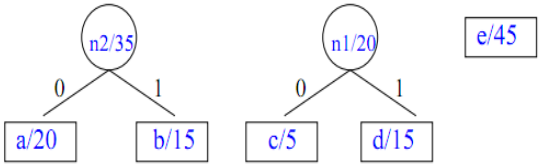
The resulting tree is the Huffman code.

Huffman coding: Example 3

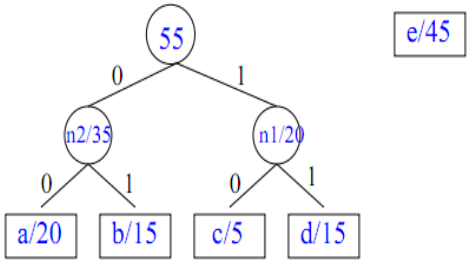
Let $A = \{a/20, b/15, c/5, d/15, e/45\}$ be the alphabet and its frequency distribution.
In the first step Huffman coding merges c and d.



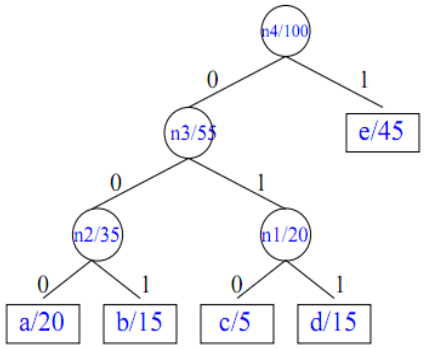
Alphabet is now $A_1 = \{a/20, b/15, n1/20, e/45\}$.
Algorithm merges a and b
(could also have merged $n1$ and b).



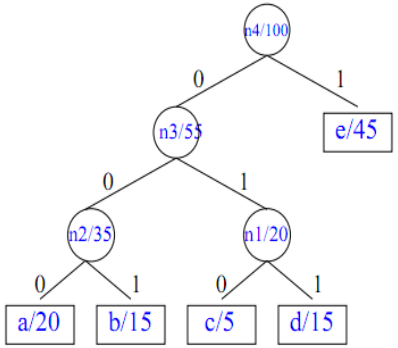
Alphabet is $A_2 = \{n2/35, n1/20, e/45\}$.
Algorithm merges $n1$ and $n2$.



Current alphabet is $A_3 = \{n3/55, e/45\}$.
Algorithm merges e and $n3$ and finishes.



Huffman code is obtained from the Huffman tree.



Huffman code is
$a = 000$, $b = 001$, $c = 010$, $d = 011$, $e = 1$.
This is the optimum (minimum-cost) prefix code for this distribution.

Huffman coding algorithm:

Given an alphabet A with frequency distribution $\{f(a) : a \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with labels (frequencies) as keys.

```
Huffman(A)
{  n = |A|;
   Q = A;                          the future leaves
   for i = 1 to n − 1              Why n − 1?
   {  z = new node;
      left[z] =Extract-Min(Q);
      right[z] =Extract-Min(Q);
      f[z] = f[left[z]] + f[right[z]];
      Insert(Q, z);
   }
   return Extract-Min(Q)  root of the tree
}
```

Running time is $O(n \log n)$, as each priority queue operation takes time $O(\log n)$.