Huffman Coding

Message Compression / Text compression reduces the size of the string significantly. It is useful in situation like communication over low bandwidth channels.

Standards encoding schemes such as, ASCII or UNICODE uses fixed length binary strings to encode character (8 bits for ASCII & 16 bits for UNICODE). where as Huffman coding uses variable length encoding optimize for the particular String 'X'. The optimization is based on the character frequencies. The frequency represents the number a character appears in the string.

Message: BCC A BB DD A ECC BB A E DD CC

We will be using this message for the rest of this lecture.

length of the message = 20.

If we store this message / send this message using ASCII code then,-

Char	no.	8-bit representation
A -	65	01000001
B -	66	01000010
C -	67	01000011
D -	68	01000100
E -	69	01000101

Therefore for the above message we need - 8 × 20 bits = 160 bits.

Now, do we need 8 bits of code to represent 5 alphabets. The answer is no.

We can use our own code to do so, i.e. we may use 3-bit representation of the character & store it. This is known as Fixed-sized codes or Fixed-length codes.

Message: B CC A BB D D A E CC BB A E DD CC.

Character	frequency/count		Code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100

To represent 5 alphabet we need 3 bits.
→ 1 bit have 0, 1
- 2 bit ⇒ 00 - 0, 01 - 1, 11 - 2, 12 - 3 } 4
$2^2 = 4$
3 bit ⇒ $2^3 = 8$ number can be represented.

Now, total cost for the message (with 20 characters) we need. 20 × 3 bits
= 60 bits.

But, If we send this message to some one then we also have to send the lookup table which will tell the receiver which code represents what.

Now, we represents the character we use ASCII codes

∴ total cost = 5 character × 8 bits
= 40 bits.

& the code will take = 5 characters × 3 bits
= 15 bits

∴ the table will take (40 + 15) bits
= 55 bits.

ASCII	our code
A 01000001	001
B 01000010	010
C 01000011	011
D 01000100	100
E 01000101	101

(lookup table).

Thus if we send the message ~~with~~ using the fixed-length code then we need –

(i) message (encoded) + (ii) lookup table
= (60 + 55) bits
= 115 bits.

This is still less then the original message used in ASCII.

So using fixed-length code we are able to reduce 35-40 percent cost.

But we can reduce it further using huffman coding which uses variable length coding.

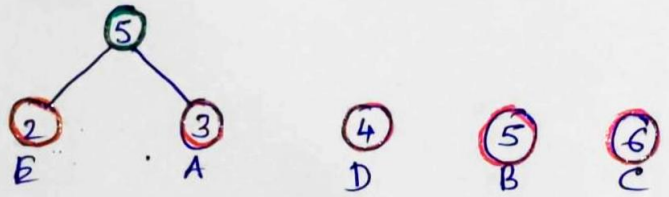
Huffman coding: The idea is to use small size code for the frequently appearing characters.

Message: B C C A B B D D A E C C B B A E D D C C

Char.	A	B	C	D	E
Count	3	5	6	4	2

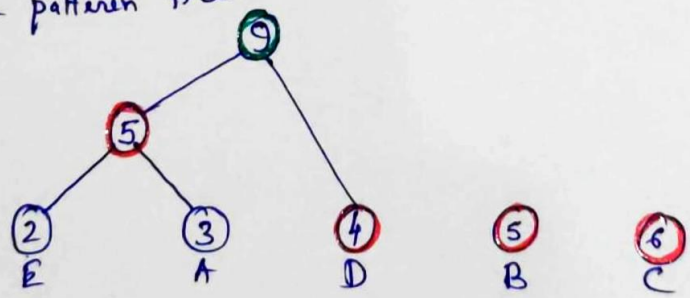
⇒ arrange the nodes in increasing order of count. & Select the smallest pair to create a new node.

(1)



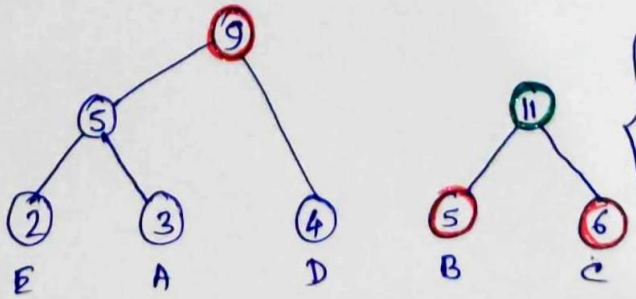
Selected smallest among red circled nodes and add them to create a new node marked as green

(2) Repeat the process until we are forming a binary tree/optimal merge pattern tree.



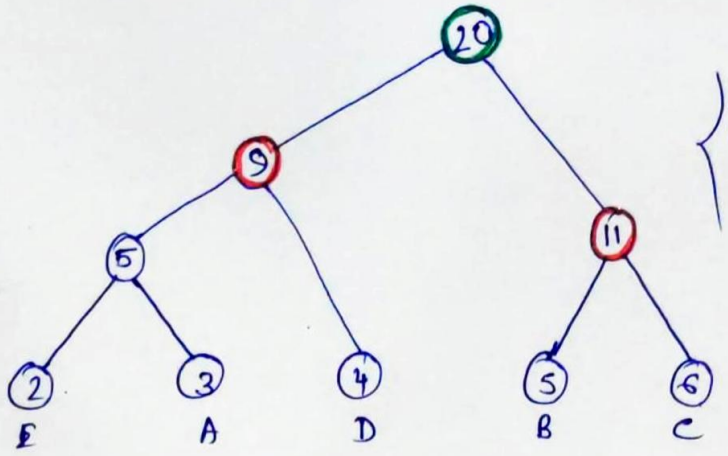
Selected smallest two nodes among red circled nodes to create a new node marked as green

(3)



Selected smallest two nodes among the red circled nodes to create a new node marked as green

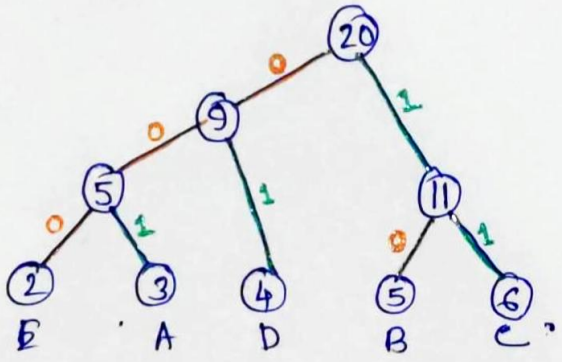
(4)



(3)

Selected two smallest nodes among the red circled nodes to create a new node marked as green

(5) Tree formation is complete now mark the left hand side edges '0' & the right hand side edges to '1'.



Char	Count	code	size.
A	3	001	3×3 = 9
B	5	10	5×2 = 10
C	6	11	6×2 = 12
D	4	01	4×2 = 8
E	2	000	2×3 = 6

total = 45 bits.

Size $\Rightarrow \sum(d_i * f_i) = [(3\times3)+(3\times3)+(2\times4)+(2\times5)+(2\times6)] = 45.$

Message: B C C A B B D D A E C C B B A E D D C C

10 11 11 001 10 10 ....

Size of the message = 45 bits.

But to decode we also have to send the lookup table / tree.

to represent the characters — 5 character × 8 bits
= 40 bits.

& the code needs — (3 + 5 + 6 + 4 + 2) bits
= 12 bits.

∴ the lookup table will need (40 + 12) = 52 bits.

∴ the message needs — (i) size of the encoded message + (ii) lookup table
= (45 + 52) bits
= 97 bits.

Algorithm: Huffman (X):

Input: String X of length n with d distinct characters.

Output: coding tree for X.

Compute the frequency f(c) of each character c of X.

Initialize a priority queue Q.

for each character c in X do.

{ Create a single node binary tree T storing c.

Insert T into Q with key f(c)

}

while Q.size() > 1 do

{

$f_1 \leftarrow$ Q.minKey()

$T_1 \leftarrow$ Q.removeMin()

$f_2 \leftarrow$ Q.minKey()

$T_2 \leftarrow$ Q.removeMin()

Create a new binary tree T with left subtree T_1 & right subtree T_2.

Insert T into Q with key $f_1 + f_2$

}

return tree Q.removeMin().