DESIGN ANALYSIS AND ALGORITHM

PRACTICAL NO 6

027_Abhishek_Ojha

Experiment No - 6 **Date of Experiment :- 7 October 2021**

Program :- Write a program to Implement Huffman's code algorithm.

Input:

A string with different characters, say "ACCEBFFFFAAXXBLKE"

Output:

Code for different characters:

Data: K, Frequency: 1, Code: 0000

Data: L, Frequency: 1, Code: 0001

Data: E, Frequency: 2, Code: 001

Data: F, Frequency: 4, Code: 01

Data: B, Frequency: 2, Code: 100

Data: C, Frequency: 2, Code: 101

Data: X, Frequency: 2, Code: 110 Data: A,

Frequency: 3, Code: 111

Algorithm

huffmanCoding(string)

Input: A string with different characters.

Output: The codes for each individual characters.

Begin

define a node with character, frequency, left and right child of the node for Huffman tree.

create a list 'freq' to store frequency of each character, initially, all are 0

for each character c in the string do

```
            increase the frequency for character ch in freq list.
    done

    for all type of character ch do
        if the frequency of ch is non zero then
```

027_Abhishek_Ojha

```
            add ch and its frequency as a node of priority queue Q.
    done

    while Q is not empty do
            remove item from Q and assign it to left child of node
            remove item from Q and assign to the right child of node
        traverse the node to find the assigned code
    done
End
```

Fig :-

Suppose the string below is to be sent over a network.

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits are required to send this string.

Using the Huffman Coding technique, we can compress the string to a smaller size.

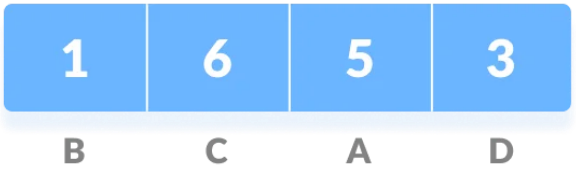
Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.

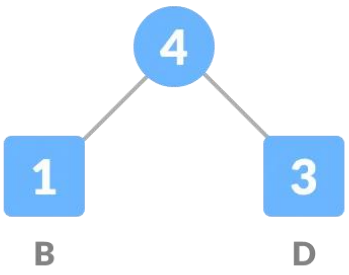
Huffman coding is done with the help of the following steps.

Calculate the frequency of each character in the string.



Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.

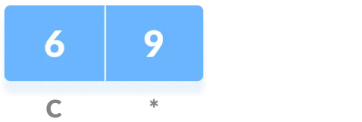
Make each unique character as a leaf node.
Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

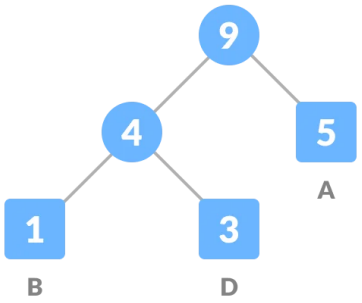


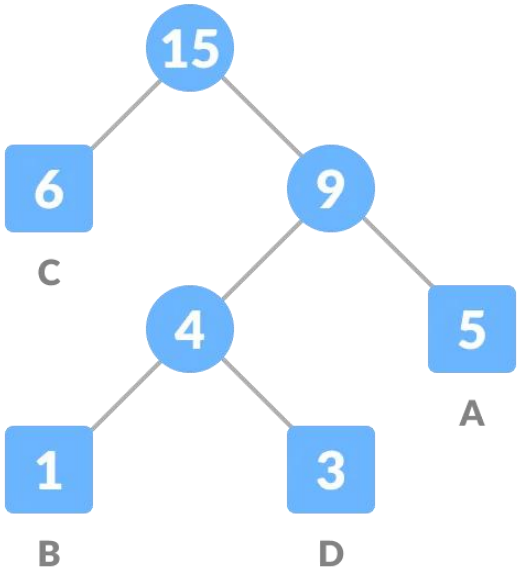
Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above).

Insert node z into the tree.

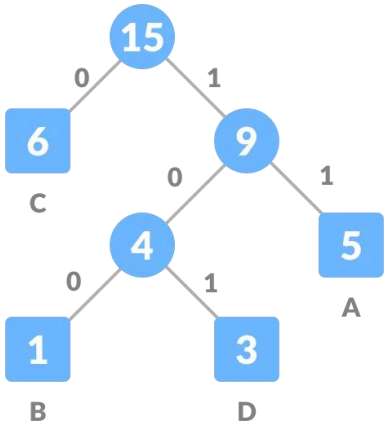
Repeat steps 3 to 5 for all the characters.







For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

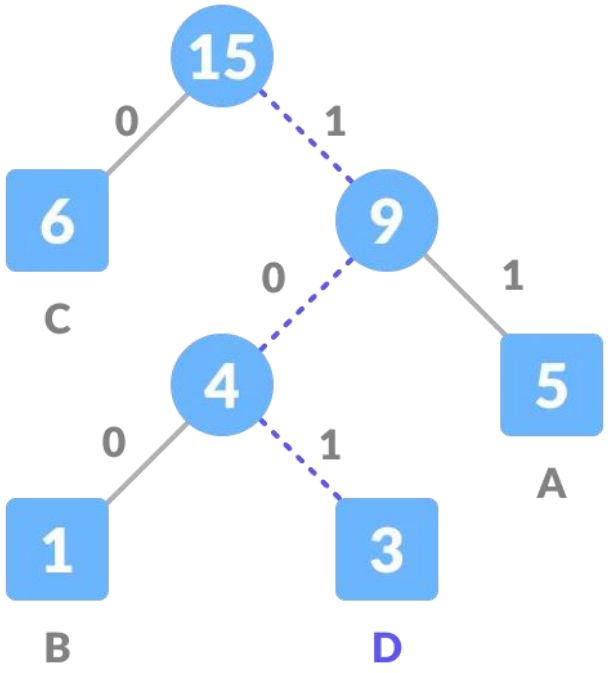
Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75.

Decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



Practical Implementation of Huffman's code algorithm

```
# A Huffman Tree Node class
node:
        def __init__(self, freq, symbol, left=None, right=None):
            # frequency of symbol
            self.freq = freq

            # symbol name (character) self.symbol =
            symbol

            # node left of current node
            self.left = left

            # node right of current node
            self.right = right

            # tree direction (0/1)
            self.huff = ''

# utility function to print huffman
# codes for all symbols in the newly #
created Huffman tree


def printNodes(node, val=''): # huffman
        code for current node newVal =
        val + str(node.huff)

        # if node is not an edge node # then
        traverse inside it if(node.left):
        printNodes(node.left, newVal)
        if(node.right):
            printNodes(node.right, newVal)

            # if node is edge node then
```

```
            # display its huffman code
        if(not node.left and not node.right):
        print(f"{node.symbol} -> {newVal}")


# characters for huffman tree
chars = ['a', 'b', 'c', 'd', 'e', 'f']

# frequency of characters
freq = [ 5, 9, 12, 13, 16, 45]

# list containing unused nodes nodes =
[]

# converting characters and frequencies
# into huffman tree nodes for x in
range(len(chars)): nodes.append(node(freq[x],
chars[x]))

while len(nodes) > 1:
        # sort all the nodes in ascending order
        # based on theri frequency nodes =
        sorted(nodes, key=lambda x: x.freq)

        # pick 2 smallest nodes
        left = nodes[0] right =
        nodes[1]

        # assign directional value to these nodes
        left.huff = 0 right.huff = 1

        # combine the 2 smallest nodes to create
        # new node as their parent newNode = node(left.freq+right.freq,
        left.symbol+right.symbol, left,
right)
```

```
        # remove the 2 nodes and add their #
        parent as new node among others
        nodes.remove(left)
        nodes.remove(right)
        nodes.append(newNode)

# Huffman Tree is ready! printNodes(nodes[0])
```

Output :

```
f:  0  c:
100    d:
101    a:
1100  b:
1101  e:
111
```

The time complexity for encoding each unique character based on its frequency is O(nlog n).

Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(nlog n).