Huffman Coding

Introduction

Huffman Coding is one approach followed for **Text Compression.** Text compression means reducing the space requirement for saving a particular text.

Huffman Coding is a lossless data compression algorithm, ie. it is a way of compressing data without the data losing any information in the process. It is useful in cases where there is a series of frequently occurring characters.

Working of Huffman Algorithm:

Suppose, the given string is:

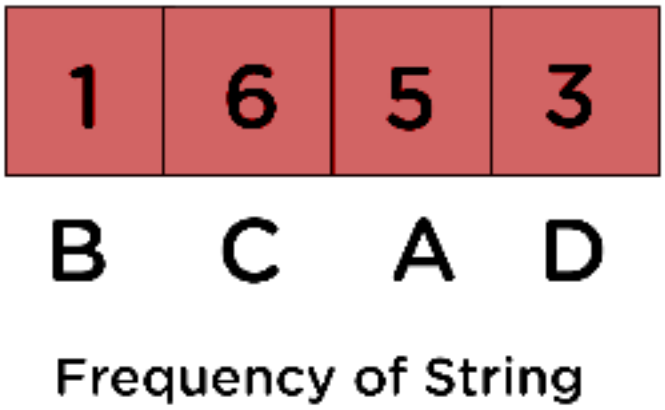
Intial String

Here, each of the characters of the string takes 8 bits of memory. Since there are a total of 15 characters in the string so the total memory consumption will be 15*8 = 120 bits. Let's try to compress its size using the Huffman Algorithm.

First-of-all, Huffman Coding creates a tree by calculating the frequencies of each character of the string and then assigns them some unique code so that we can retrieve the data back using these codes.

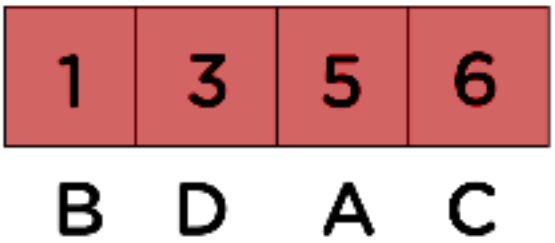
Follow the steps below:

1. Begin with calculating the frequency of each character value in the given string.



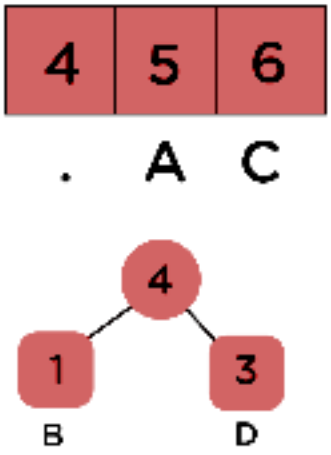
Frequency of String

2. Sort the characters in ascending order concerning their frequency and store them in a priority queue, say **Q.**
3. Each character should be considered as a different leaf node.



Characters sorted according to the frequency

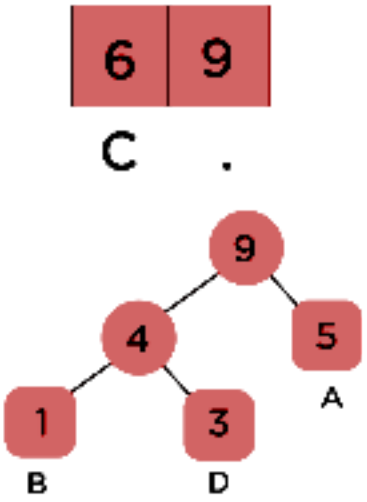
4. Make an empty node, say **z**. The left child of z is marked as the minimum frequency and the right child, the second minimum frequency. The value of z is calculated by summing up the first two frequencies.



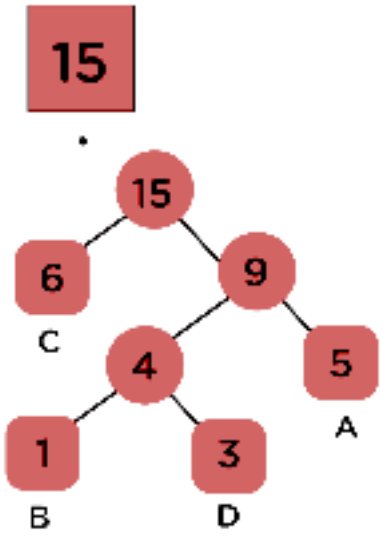
Getting the sum of the least numbers

Here, "." denote the internal nodes.

5. Now, remove the two characters with the lowest frequencies from the priority queue **Q** and append their sum to the same.
6. Simply insert the above node **z** to the tree.
7. For every character in the string, repeat steps 3 to 5.

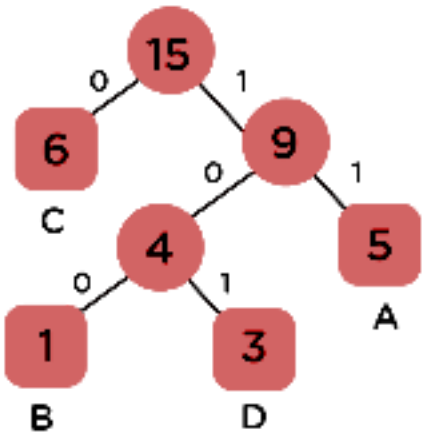


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. Assign 0 to the left side and 1 to the right side except for the leaf nodes.



Assign 0 to the left edge and 1 to the right edge

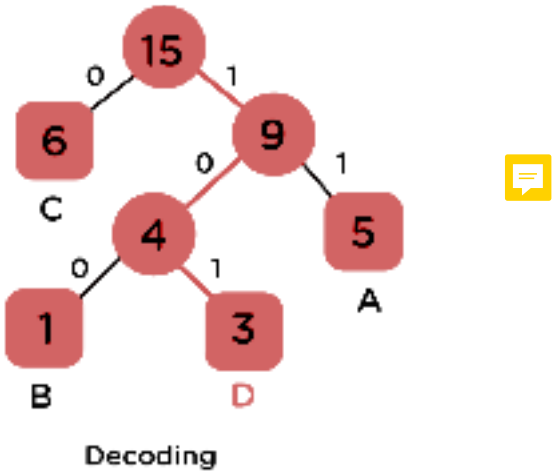
The size table is given below:

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

Size before encoding: 120 bits

Size after encoding: 32 + 15 + 28 = 75 bits

To decode the code, simply traverse through the tree (starting from the root) to find the character. Suppose we want to decode 101, then:



Decoding

Time complexity:

In the case of encoding, inserting each character into the priority queue takes **O(log n**) time. Therefore, for the complete array, the time complexity becomes **O(nlog(n))**.

Similarly, extraction of the element from the priority queue takes **O(log n)** time. Hence, for the complete array, the achieved time complexity is **O(nlog n)**.

Python Code:

Go through the given Python code, for deeper understanding:

```
# Huffman Coding in python
string = 'BCAADDDCCACACAC' #String similar to the above-taken example


# Creating tree nodes
class NodeTree(object):

    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self): #Return children of a node
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)


# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d
```

```
# Calculating frequency
freq = {}
for c in string:
    if c in freq:
        freq[c] += 1
    else:
        freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
nodes = freq

while len(nodes) > 1:
    (key1, c1) = nodes[-1]
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))

    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])

print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
    print(' %-4r |%12s' % (char, huffmanCode[char]))
```

Applications of Huffman Coding:

- They are used for transmitting fax and text.
- They are used by conventional compression formats like PKZIP, GZIP, etc.