Huffman Lossless Compression

Submitted By:

Aaditya Jain -	CB.EN.U4AIE19001
Anirudh Bhaskar -	CB.EN.U4AIE19007
CS Ayush Kumar -	CB.EN.U4AIE19017
Rohith Ramakrishnan -	CB.EN.U4AIE190052
Srinath Murali -	CB.EN.U4AIE19063

Table of Contents

Objective:

Pre-process a text file (allow only alphabets to remain, filter all other characters from each line), then tokenise it (to a list of words) and store it in an array of characters. Create an alphabet BST that stores a Key-Value pair of characters in this array (from the file) and its count; read the file, one character at a time and insert it to the BST along with a count value, with the character as the key (decides the ordering in the BST). The first occurrence inserts the new node for that character with count as one, successive duplicate occurrences locates this node and increments the count by one. Once the entire file is done this way, we start populating a Heap with the characters but this time with the count as the key. THIS SHOULD BE A MAX HEAP. The root of the heap also contains an index 1(binary), the left child is 10 and right child is 11. This will continue down the entire heap and the binary indexing is done based on the number of times a character (alphabet) occurs; the more times It occurs the smaller should be its index value, thus saving on file size. Encode the file by replacing the ASCII character by the binary value string. Once we have this heap, we must be able to also decode to get the original file back.

Implementation:

- A Text File is read character by character and stored in a character Array
- A Binary Search Tree is generated and the character array is fed in as the input
- As the characters repeat the BST's Nodes will increment the frequency of the character
- When the char is read, we will search the whole BST and if the char exists, we increment otherwise we create a new node for this character
- We create a Node Array to store the Frequency and the character
- Using a Maxheap we Implement the Huffman Algorithm and store the corresponding result in another BST.
- For Decoding, the compressed file is given as the input along with the BST.
- The corresponding encoded/compressed file is retrieved without any loss other than special characters.

Data-Structures used:

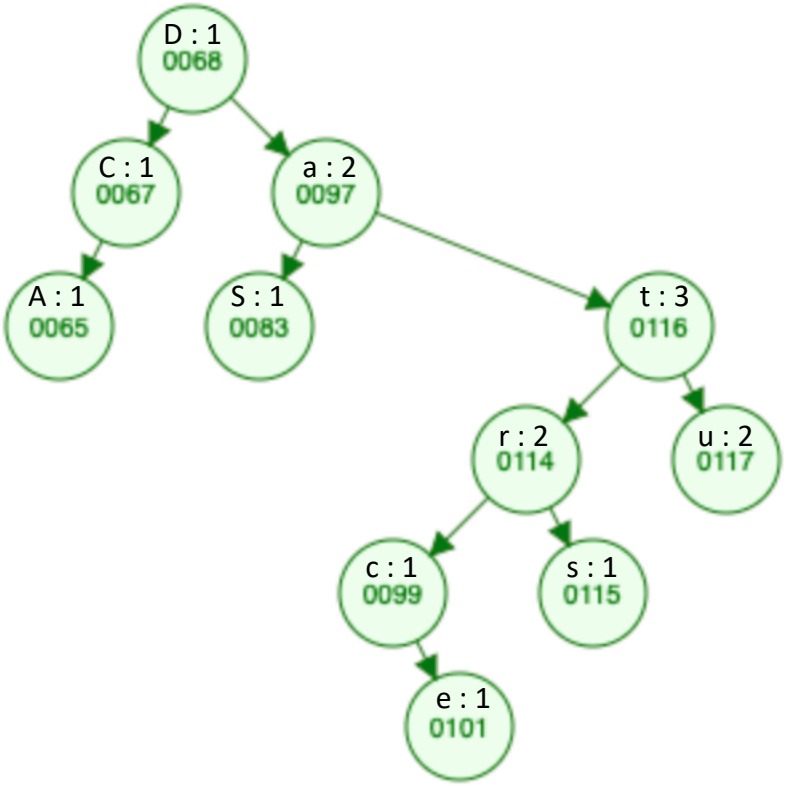
- Binary Search Tree
- Node Array
- LinkedLists
- MaxHeap
- Arrays

Huffman Algorithm Example:

Contents of Text File:
DataStructuresCA

$$Char\ Array: \\ [D \quad ,a \quad ,t \quad ,a \quad ,S \quad ,t \quad ,r \quad ,u \quad ,c \quad ,t \quad ,u \quad ,r \quad ,e \quad ,s \quad ,C \quad ,A]$$

Hash Code of each Character is the key and the frequencies obtained are the values.
The BST Made out of the Character array would look like:



Max-Heap implemented using the Binary Tree:

$\longrightarrow$ $[3{:}\,t \quad 2{:}\,a \quad 2{:}\,r \quad 2{:}\,u \quad 1{:}\,C \quad 1{:}\,A \quad 1{:}\,S \quad 1{:}\,s \quad 1{:}\,c \quad 1{:}\,e \quad 1{:}\,D]$

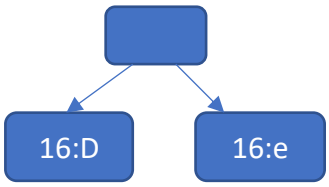
Furthermore, we use 'Inversion':
The reason we are inverting is that we need to pop the minimum values. As we are to use a Maxheap, if we invert, we are popping the false maximum values whose values are minimum originally. So, the sum count of all the frequencies is:

$$(3 + 2 + 2 + 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1) = 16$$

"Inversed" Frequency Values = (16/each frequency)

→ $[16:D \quad 16:e \quad 16:c \quad 16:s \quad 16:S \quad 16:A \quad 16:C \quad 8:u \quad 8:r \quad 8:a \quad 5.3:t]$

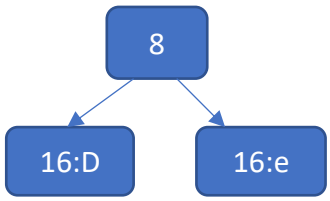
We take the first 2 maximum values i.e. 16:D, 16:e in this case and we create a node out of them



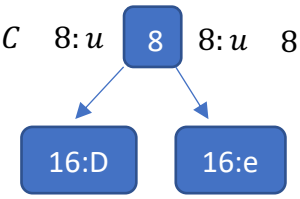
The root node should contain the sum of the frequencies of both the nodes because we applied inversion.

The node value should be = $\frac{N}{\frac{N}{a}+\frac{N}{b}}$

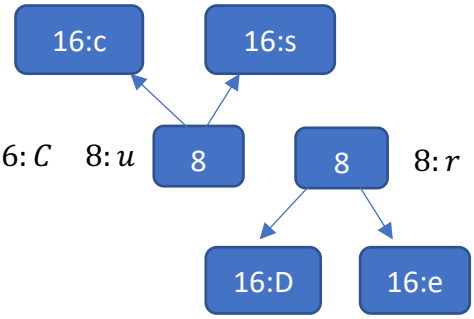
where N = Sum of frequencies (16) and a and b will be the child nodes values (16,16). So, the value of the root node will be 8.



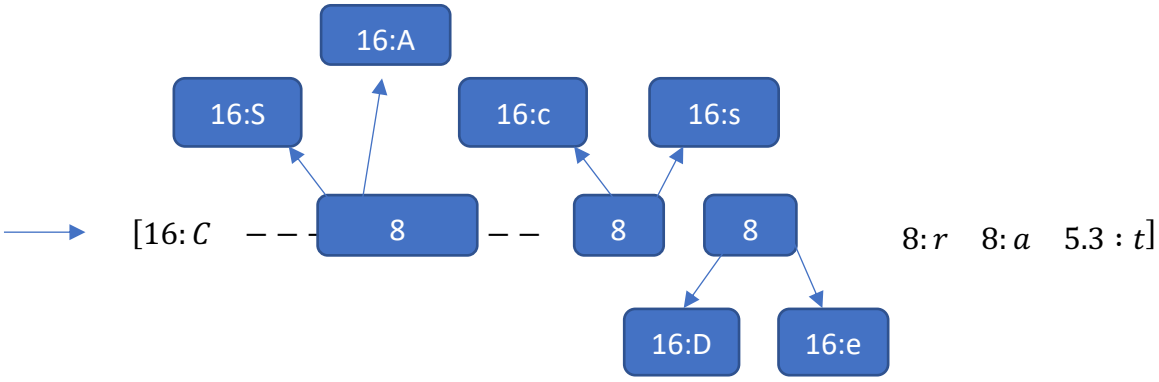
This root node will be inserted in the heap like this:

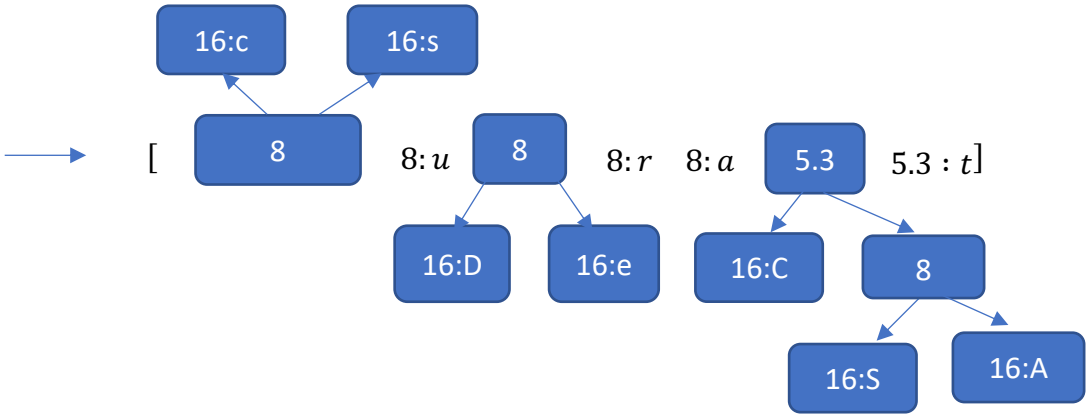
→ $[16:c \quad 16:s \quad 16:S \quad 16:A \quad 16:C \quad 8:u$  $8:u \quad 8:r \quad 8:a \quad 5.3:t]$

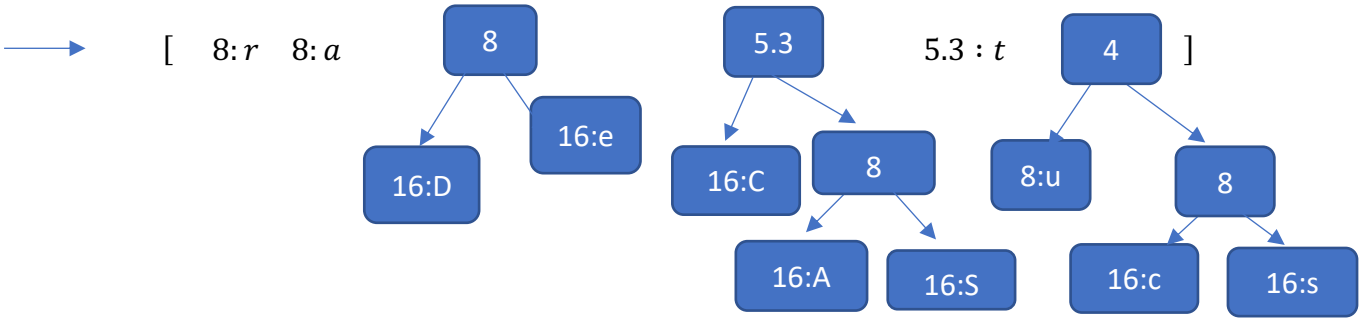
Next, we take the two maximum values again:

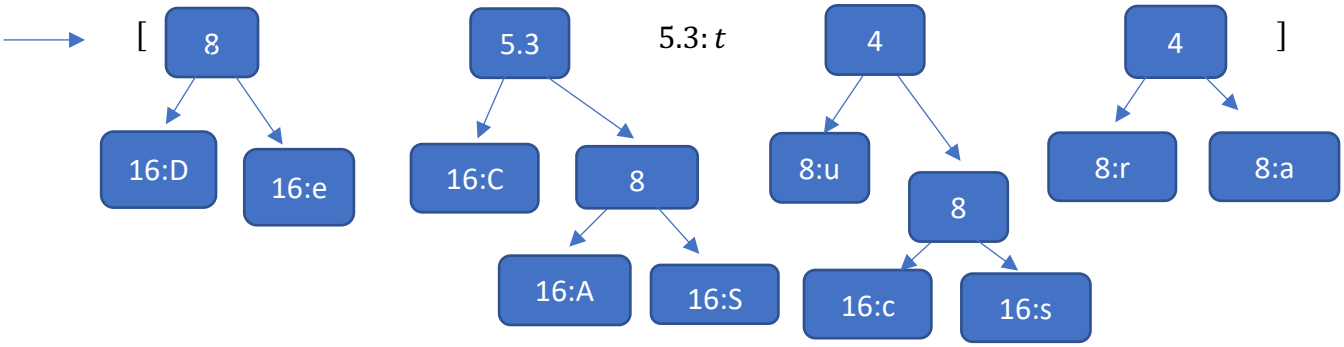


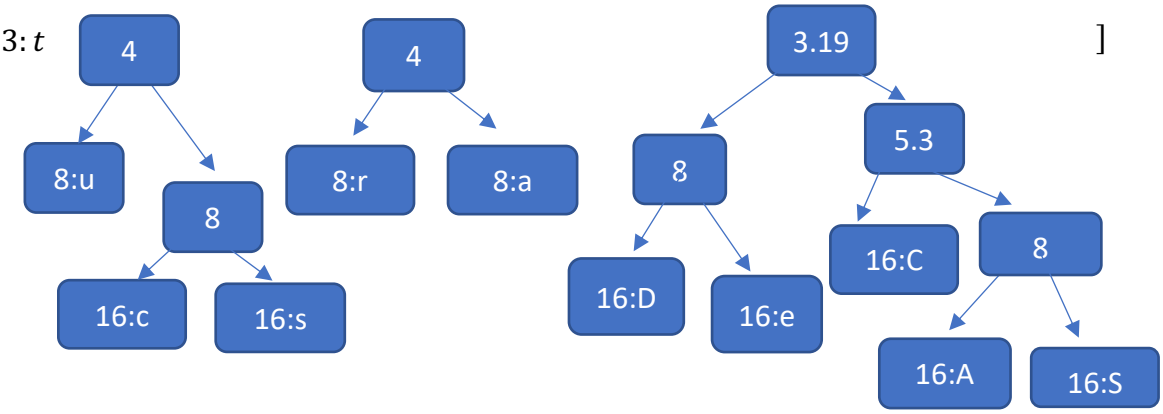
→ $[16:S \quad 16:A \quad 16:C \quad 8:u \quad 8:r \quad 8:a \quad 5.3:t]$

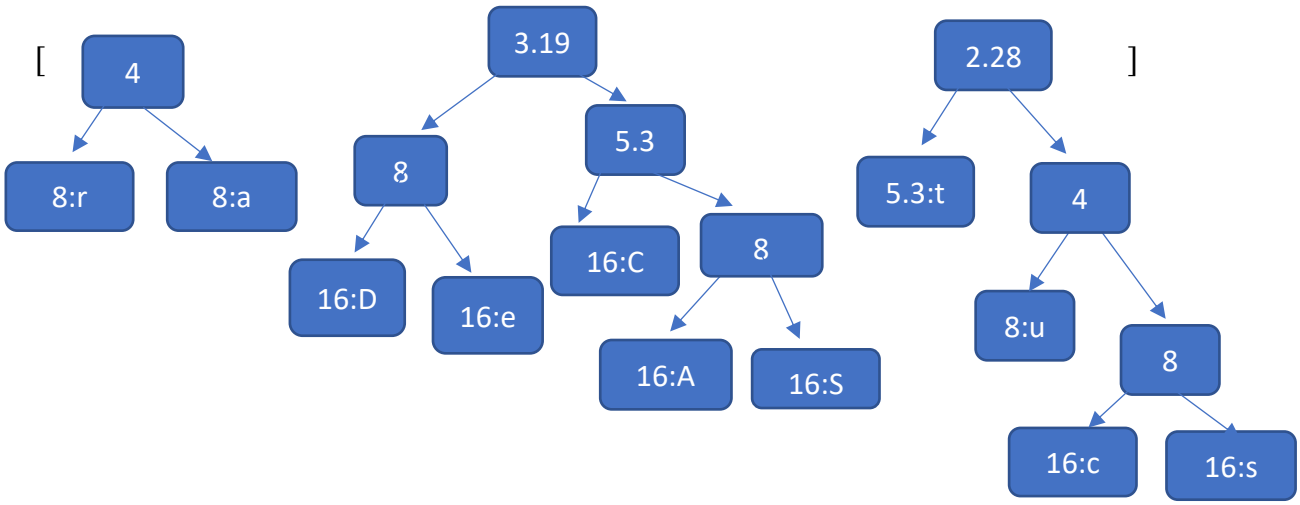

16:A
16:S
16:c
16:s
[16: C
8
8
8
8: r
8: a
5.3 : t]
16:D
16:e

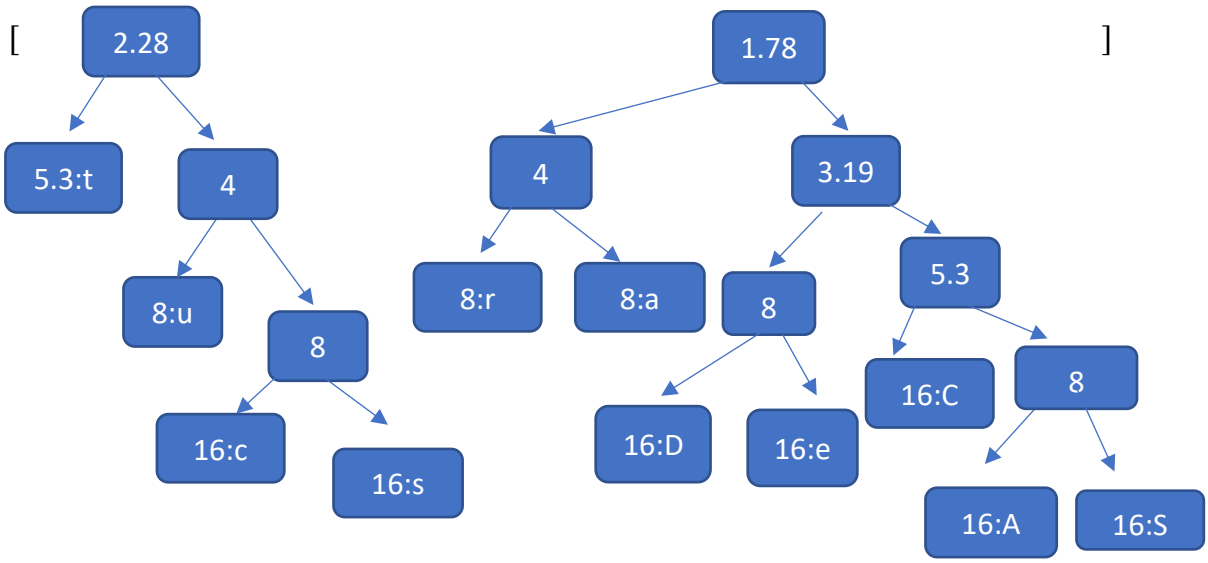

16:c
16:s
8
[
8: u
8
8: r
8: a
5.3
5.3 : t]
16:D
16:e
16:C
8
16:S
16:A


[
8: r
8: a
8
16:e
16:D
5.3
5.3 : t
4
]
16:C
8
8:u
8
16:A
16:S
16:c
16:s

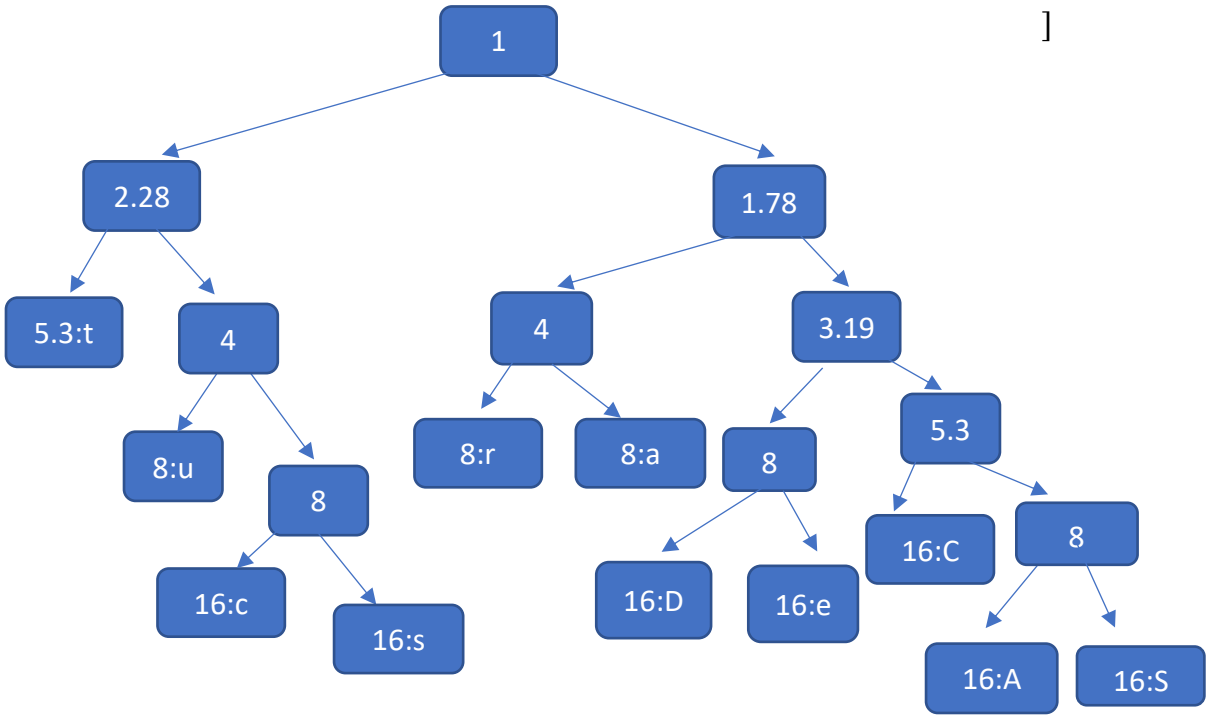

[
8
5.3
5.3: t
4
4
]
16:D
16:e
16:C
8
8:u
8:r
8:a
8
16:A
16:S
16:c
16:s

→ [5.3: *t* 
4
4
3.19
8:u
8:r
8:a
8
8
5.3
16:c
16:s
16:D
16:e
16:C
8
16:A
16:S
]

→ [
4
3.19
2.28
8:r
8:a
8
5.3
5.3:t
4
16:D
16:e
16:C
8
8:u
8
16:A
16:S
16:c
16:s
]

→ [
2.28
1.78
5.3:t
4
4
3.19
8:u
8
8:r
8:a
8
5.3
16:c
16:s
16:D
16:e
16:C
8
16:A
16:S
]

→ [ ]



In the final step the MaxHeap will only contain one root node.

Encoding:

To encode the string, we require a BST. We first read the given text file and covert it to a character array. The characters are loaded to a BST with their frequency, which then is broken to a node array and converted to a Maxheap. Then we use the Huffman Algorithm and form the tree.

To find the corresponding bits for encoding, we follow the following steps:

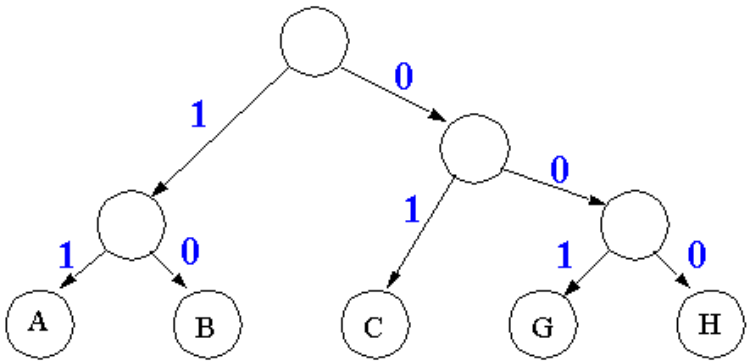
- The given string is converted to a BST
- Each node of the BST contains the character and their respective frequency
- Two nodes of the minimum frequency are retrieved form the heap
- New internal nodes are created with frequency as the sum of the retrieved nodes
- The process is repeated until the heap has only one node left
- The tree is traversed from the root
- Moving to the left child, 0 is written to the array and 1 while moving to the right child

Considering the text “DataStructeresCA”, the encoded file would be:

“11110011000111000000101101111100110010111010011011101 0”

And bits corresponding to each character are:

t:00
r:010
a:011
S:1000
s:1001
A:1010
C:1011
u:110
e:1110
D:11110
c:11111



Decoding:

To decode the encoded data, we require the Huffman tree. We iterate through the binary encoded data. To find character corresponding to current bits, we use following simple steps:

- We start from root and do following until a leaf is found.
- If current bit is 0, we move to left node of the tree and if the bit is 1, we move to right node of the tree.
- If during traversal, we encounter a leaf node, we obtain the character of that particular leaf node and then again continue the iteration of the encoded data starting from root.

Result and Innovation:

Considering the text “DataStucturesCA” again and comparing the sizes of the original and encoded file, we get:
Size of the Original file is: 0.052734375 Kilobytes for Ascii
Size of the Compressed/Encoded file is: 0.001953125 Kilobytes for Huffman Compression

Furthermore, we encoded an image using Huffman Algorithm,

- The Image was loaded and encoded to a text file using Base64
- The text file was further compressed using the Huffman algorithm.
- This compressed text file can be decompressed/decoded to get the image back without any loss.

The size of the image is 909.04 Kilobytes and the size of the encoded text file is 1213.16 Kilobytes. After using the Huffman algorithm, the size of the text file is 900.27734375 Kilobytes. We obtain the original image of the same size after decoding.

This Double-Encrypted file can be useful for Confidential Image Transfer , where the whole image is sent in a Binary Format.