Data Structures Lab
Session 10

Note:

- Lab manual cover following topics
  {Huffman coding encoding, decoding, Applications }
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. It compresses data very effectively saving from 20% to 90% memory, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.
These are called fixed-length codes. If all characters were used equally often, then a fixed-length coding scheme is the most space efficient method. But such thing isn't possible in real word. If some characters are used more frequently than others, is it possible to take advantage of this fact and somehow assign them shorter codes? The price could be that other characters require longer codes, but this might be worthwhile if such characters appear rarely enough. Huffman coding variable-length codes approaches. While it is not commonly used in its simplest form for file compression, One motivation for studying Huffman coding is because it provides type of tree structure referred to as a search trie.

There are mainly two major parts in Huffman Coding

Build a Huffman Tree from input characters.
Traverse the Huffman Tree and assign codes to characters.

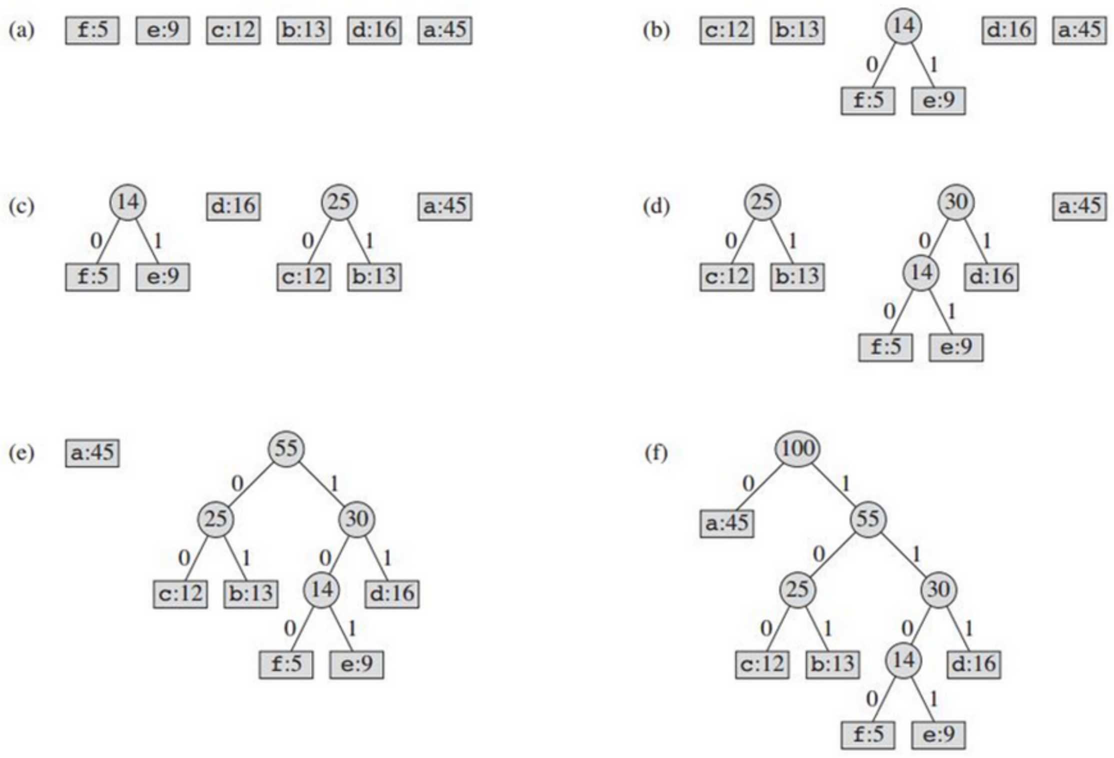
Task:1

Study given steps to build Huffman Tree
Input is an array of unique characters along with their frequency of occurrences and output the Huffman Tree.
Steps to build Huffman Tree (Compression Technique)

1. The technique works by creating a binary tree of nodes. Tree can stored in a regular array, the size of which depends on the number of symbols, n. A node can either be a leaf node or an internal node. Initially all nodes are leaf nodes, which contain the symbol itself, its frequency and optionally, a link to its child nodes. As a convention, bit '0' represents left child and bit '1' represents right child. Priority

queue is used to store the nodes, which provides the node with lowest frequency when popped. The process is described below:

a. Create a leaf node for each symbol and add it to the priority queue.
b. While there is more than one node in the queue:
c. Remove the two nodes of highest priority from the queue.
d. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequency.
e. Add the new node to the queue.
f. The remaining node is the root node and the Huffman tree is complete.



(a) f:5 e:9 c:12 b:13 d:16 a:45
(b) c:12 b:13 14 0 1 f:5 e:9 d:16 a:45
(c) 14 f:5 e:9 d:16 25 c:12 b:13 a:45
(d) 25 c:12 b:13 30 14 d:16 f:5 e:9 a:45
(e) a:45 55 25 30 c:12 b:13 14 d:16 f:5 e:9
(f) 100 a:45 55 25 30 c:12 b:13 14 d:16 f:5 e:9

TASK 2:
Implement the given decoding pseudo-code :

Decompression Technique:

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte value, usually by traversing the Huffman tree node by node as each bit is read from the input stream. Reaching a leaf node necessarily terminates the search for that particular byte value. The leaf value represents the desired character. Usually the Huffman Tree is constructed using statistically adjusted data on each compression cycle, thus the reconstruction is fairly simple. Otherwise, the information to reconstruct the tree must be sent separately.

```
Procedure HuffmanDecompression(root, S):   // root represents the root of
Huffman Tree
n := S.length                          // S refers to bit-stream to be
decompressed
for i := 1 to n
    current = root
    while current.left != NULL and current.right != NULL
        if S[i] is equal to '0'
            current := current.left
        else
            current := current.right
        endif
        i := i+1
    endwhile
    print current.symbol
endfor
```

References:

https://en.wikipedia.org/wiki/Huffman_coding#Applications
https://www.geeksforgeeks.org/priority-queue-using-binary-heap/
https://www.geeksforgeeks.org/priority-queue-set-1-introduction/
https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Huffman.html
https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Lab9: Huffman Coding		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		