**3rd Milestone of Project**

**Algorithm Huffman(X)**

**Input:** String x of length n with d distinct characters

**Output:** coding tree for X

1. Compute the frequency f (c) of each character c of X.
2. (c1, f[c1] ) , (c2, f[c2] ) ,…, (cn, f[cn] )
3. Initialize a priority queue Q
4. For i=1 to n-1 Do
5. Create a single-node binary tree storing c
6. T = allocate new node.
7. Insert T into Q with key f (c).
8. While Size ( )> 1 do
9. *F1* Q.min()
10. *T1* Q.removeMin()
11. *F2* Q.min()
12. *T2* Q.removeMin()
13. Create a new binary tree T with left sub tree T1 and right sub tree T2
14. T.left=T1 T.right=T2 Insert T into Q with key f1 +f2
15. T.f = T1.f1 + T2.f2
16. return Tree Q.removeMin()

**Correctness**

**Time Complexity**

Huffman’s algorithm proceeds as shown in above. Since the alphabet contains 6 letters, the initial queue size is n , and the final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

* Line 1-2 compute the frequencies of all characters from c1 to cn in input string X.
* Line 3-7 initializes the min-priority queue Q with the characters in X. **For** loop is used to store all character’s frequency in priority queue Q.
* Lines 8–15 repeatedly extracts the two nodes T1 and T2 of lowest frequency from the queue using while loop, replacing them in the queue with a new node T, representing their merger. The frequency of T ’ is computed as the sum of the frequencies of T1 and T2 in line 15.
* The node T’ has T1 as its left child and T2 as its right child. (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.)
* After (n −1) mergers, line 16 returns the one node left in the queue, which is the root of the code tree.

.

To analyze the running time of Huffman’s algorithm, we assume that Q is implemented as a binary min-heap. For a set X of n characters, we can initialize Q using **For loop** in line 3-7 in O(n) time using the **BUILD-MIN-HEAP** procedure. The **While loop** in lines 8-15 executes exactly when only one last node left in queue (**size=1)** which takes n-1 iterations, and since each heap operation requires time O(lgn), the loop contributes O(nlgn) to the running time. Thus, the total running time of HUFFMAN on a set of n characters is **O(nlgn).**