IT251 Lab Assignment 10 – Huffman Encoding

Note:
1. Read in the input using a text file, and NOT by typing it in the console. The input file should be given as an argument while running your code. For e.g. for a file solution.cpp, compile it by 'g++ test.cpp' and run it by './a.out input.txt' , where 'input.txt' contains the input to the problem.
2. The function signatures you are asked to implement in this exercise are just guidlines and need not be strictly followed. As long as you respect the input/output constraints your code will pass the test cases.
3. We will use the term Huffman tree and encoding tree interchangebly. They are the same thing!

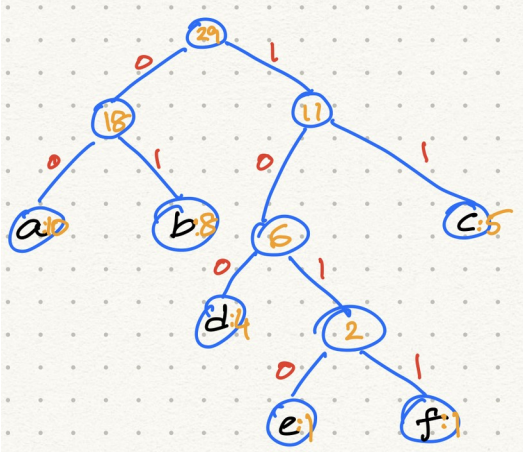
We will implement this lab exercise in three parts. The file you submit will need to solve problem 3. Do NOT submit three different files for the three problems.

Problem1 : Encoding Text.
In this problem, given a reference to (the root of) an encoding tree, and a text, you should return the encoding of the text according to the tree. It might be easier (and faster!) to first traverse the encoding tree and build a map that associates each character with its encoding. Then for each character in the text, we can simply replace it by its encoding (got from the map).

Write a function "encodeText(TreeNode* node, string text)" that implements the above operations. You will need to build a (encoding) tree to test the working of your code. A sample input / output is shown below:

Input:

Text: aaabbaedfccddbbabcabacdabbcaa, Tree:



Output:
00000001010010101001011111110010001010001110001001110000010111000 0

Problem2 : Flattening Tree.
The output of a Huffman encoding must also include the encoding tree used along with the encoded text. Only then will we be able to decode the compressed text. In this part you will write code to return the encoding tree. We will call this "flattened an encoding tree".

The flattening of a tree returns two sequences, which are got from a pre-order and an in-order traversal of the encoding tree. These two traversals will allow us to reconstruct the encoding tree unambiguously. The two sequences are as follows:
1. The first sequence describes the *shape* of the tree. It essentially does a pre-order traversal of a tree. If a node being visited is a leaf node, then return 0, else return 1. After visiting a node, the pre-order visits its left child, and then its right child.

2. The second sequence is obtained by an in-order traversal of the tree. We will only return the characters encountered in the traversal, which are exactly the leaves of the tree, since in an encoding tree, the leaf nodes correspond to the different characters of the text that are to be encoded.

For the encoding tree in problem 1, the flattened tree is as follows:
First sequence: 11001101000
Second sequence: abdefc

Write a function “flattenTree(TreeNode* node)” that implements the above functionality. You could add additional parameters to this function for the two return sequences.

Problem3 : Implementing the Huffman Encoding.

This is main code where we implement Huffman’s greedy algorithm for encoding symbols. We will use the code in the previous problems here. We take as input a text file which is a sequence of symbols. The output in this problem will also be a text file ‘output.txt’. (*Please do NOT print the output to the console, or to a differently named file. This will result in your code failing the test cases)* After building the encoding tree of the input text, your algorithm should create the file ‘output.txt’ which contains the following two things:

1. The flattened encoding tree: This will consists of two lines corresponding to the two sequences in problem 2. The first line of the file ‘output.txt’ will contain the first sequence of the flattened tree, and the second line will contain the second sequence.
2. The Encoded text: This will be the encoded text (of Problem 1). The encoded text will appear in the third line of the file ‘output.txt’.

To implement this algorithm, we will need to have a Min-Heap, which stores the nodes of the Tree, ordered by their frequencies. To create the next sub-tree, we do two extractMin operations on the heap to get two nodes n1 and n2 with the least frequencies, create a new node n, and make n1 and n2 children of n. The following snippet of pseudo code illustrates this:

```
n1 = Heap.extractMin()
n2 = Heap.extractMin()
n  = new Node()
n.leftChild = n2
n.rightChild = n1
```

Input: The input text file is a sequence of symbols. Assume that all the symbols in the text file are from the 26 characters of the english alphabet (small cases). The text file does not contain, spaces, newline characters or any other special characters.

Output: As mentioned above, the output file contains two things: the flattened encoding tree and the encoded text. The first line is the first sequence of the flattened tree, the second line is the second sequence of the flattened tree. The third line is the encoded text.

Constraints: Text size is atmost 10000 characters. Each character in the text is from the set of small case English alphabet, i.e. {a,b,c,...,x,y,z}

Sample Input/Output:
Input:
aaabbaedfccddbbabcabacdabbcaa

Output (create a file ‘output.txt’ and do NOT print on console):
11001101000
abdefc
00000001010010101001011111110010001010001110001001110000010111000 0

Explanation: The first two lines of the output file correspond to the flattened version of the tree shown below. The third line is the actual encoding of the input text using this encoding tree.

a: 00
b: 01
c: 11
d: 100
e: 1010
f: 1011