CS 201/218 DATA STRUCTURES

Assignment # 04

Guidelines

1. Submit the zip folder (18X-XXXX) containing the code files (.cpp & .h including main.cpp) only.
2. Any plagiarism in assignment will lead to **zero** marks.
3. Use of STL is not allowed under any circumstances.
4. Only the submissions with executable code (i.e. code without any errors) will be accepted.
5. Submission sent to the emails will not be graded.
6. Deadline: Deadline to submit assignment is 22th April 2020 10:00 AM. No submission will be considered for grading after 22th March 2020 10:00 AM. Correct and timely submission of assignment is responsibility of every student; hence no relaxation will be given to anyone. Remember; this time no One Day Late Submission will be accepted.
7. Use the linked implementation of the **Binary Tree**. And Array implementation of **Ordered Queue**.
8. Consider only ascii characters from 0 to 127 (7 bits). And Input file should not include '*' character.

Due Date and Time:

Submission Policy

All the submissions MUST be uploaded on slate. Solutions sent to the emails will not be graded.
To avoid last minute problems (unavailability of slate, load shedding, network down etc.), you are strongly advised to start working on the assignment from day one.

Plagiarism Policy

Solutions of all the sections will be placed in a central repository and will be scanned for plagiarism. All the plagiarized solutions will be marked zero.

Overview how it Works! (Main Theme):

You've probably heard about David Huffman and his popular compression algorithm

The idea behind Huffman coding is based upon the frequency of a symbol in a sequence. The symbol that is the most frequent in that sequence gets a new code that is very small, the least frequent symbol will get a code that is very long, so that when we'll translate the input we want to encode the most frequent symbols will take less space than they used to and the least frequent symbols will take more space but because they're less frequent it won't matter that much.

For this algorithm you need to have a basic understanding of **binary tree** data structure and the **ordered queue** (**sorted on the bases of frequency**) data structure. In the source code we'll use the ordered queue.

Let's say we have the string "beep boop beer!" which in his actual form, occupies 7 bits (suppose, ascii 0 to 127). That means that in total, it occupies 15*7 = 105 bits of memory. Through encoding, the string will occupy 40 bits approximately.

To better understand this example, we'll going to apply it on an example. The string "beep boop beer!" is a very good example to illustrate this. In order to obtain the code for each element depending on its frequency we'll need to build a binary tree such that each leaf of the tree will contain a symbol (a character from the string). The tree will be built from the leafs to the root, meaning that the elements of least frequency will be farther from the root than the elements that are more frequent.

To build the tree this way we'll use a ordered queue, that the element with the least frequency is the most important. Meaning that the elements that are the least frequent will be the first ones we get from the queue. We need to do this so we can build the tree from the leaves to the root.

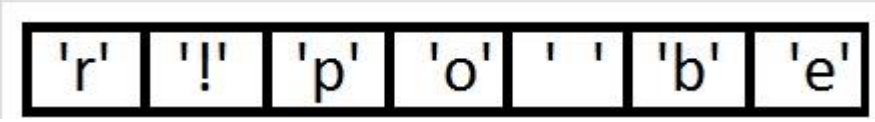
Example:

If we want to compress the string "beep boop beer!"

Firstly we calculate the frequency of each character :

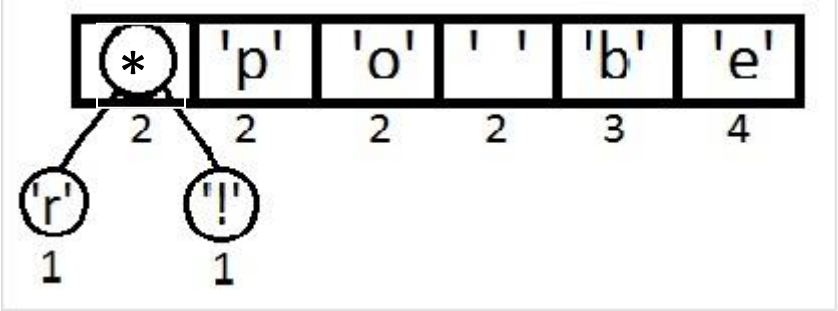
Character	Frequency
'b'	3
'e'	4
'p'	2
' '	2
'o'	2
'r'	1
'!'	1

After calculating the frequencies, we'll create binary tree nodes for each character, and we'll introduce them in the ordered queue with the frequency:

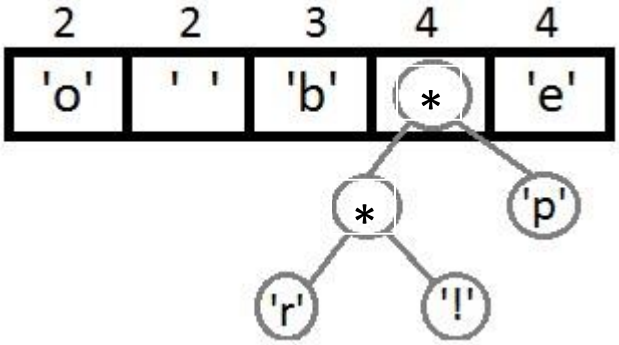


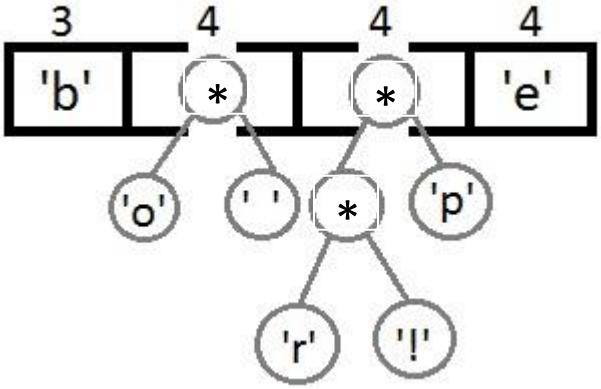
Frequencies	1	1	2	2	2	3	4

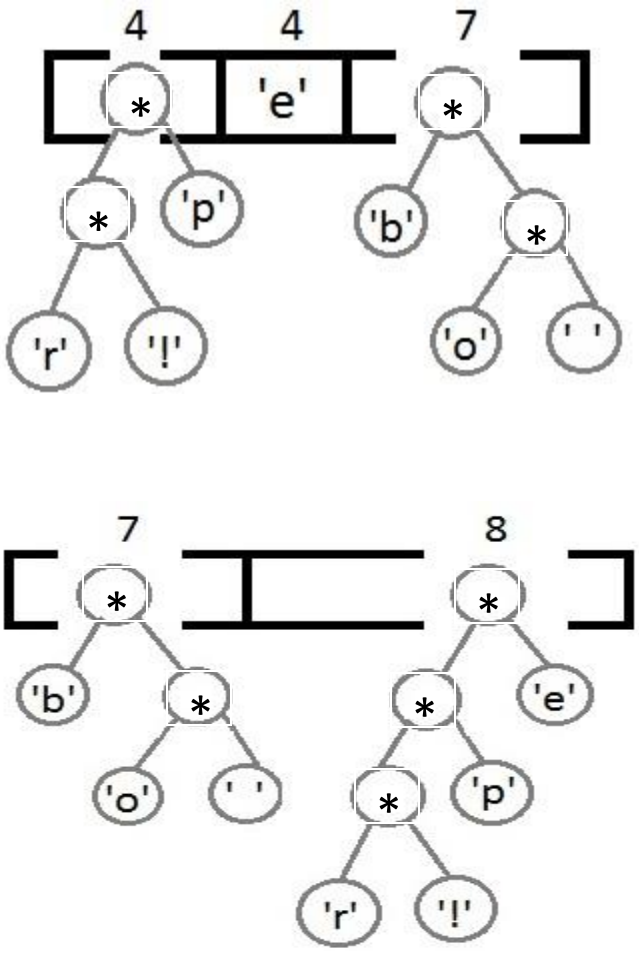
We now get the first two elements from the queue and create a link between them by creating a new binary tree node to have them both as successors (initialize root node with '*****' (character)), so that the characters are siblings and we add their frequencies. After that we add the new node we created with the sum of the frequencies of its successors as it's frequency in the queue.



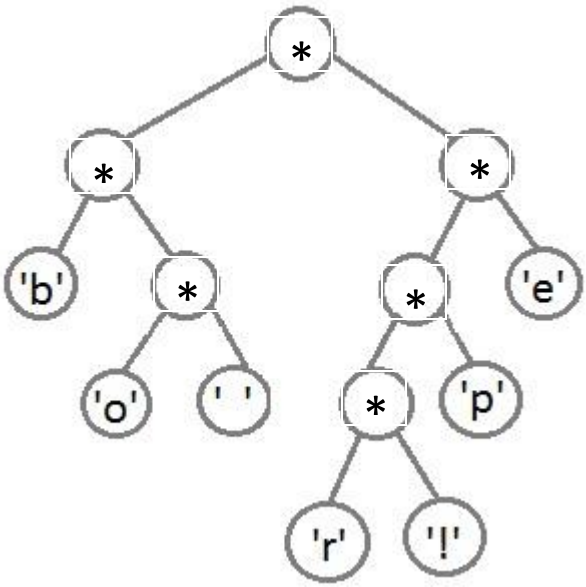
We repeat the same steps and we get the following:



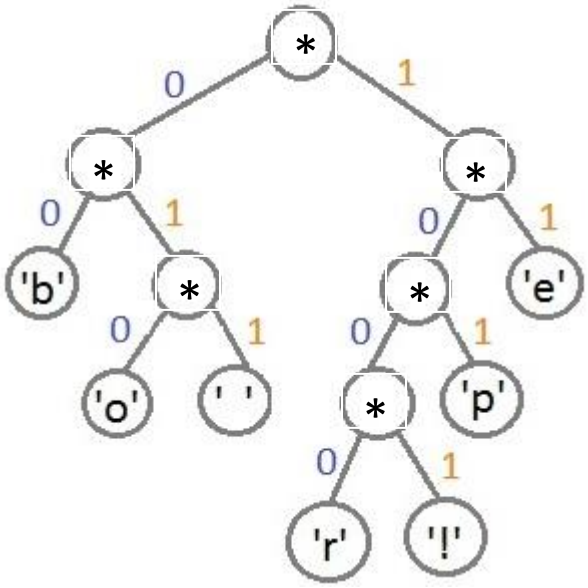




Now after we link the last two elements, we'll get the final tree:



Now, to obtain the code for each symbol we just need to traverse the trees until we get to that leaf node (symbol (character)). Now we associate 0 (zero) to left edges and 1 (one) to right edges, after each step we take to the left we concatenate a 0 to the code or 1 if we go right.



To decode a string of bits we just need to traverse the tree for each bit, if the bit is 0 we take a left step and if the bit is 1 we take a right step until we hit a leaf (which is the symbol we are looking for). For example, if we have the string “101 11 101 11” . Using above given tree, decoding it we’ll get the string “pepe”.

A practical aspect of implementing this algorithm is considering building a Huffman table as soon as we have the tree. The table is basically a structure that contains each symbol with its code because it will make encoding somewhat more efficient. It’s hard to look for a symbol by traversing a tree and at the same time calculating its code because we don’t know where exactly in the tree that symbol is located. **As a principle, we use a Huffman table for encoding and a Huffman tree for decoding.**

Character	Code
‘b’	00
‘o’	010
‘ ’	011
‘r‘	1000
‘!’	1001
‘p’	101
‘e’	11

Original String: beep boop beer!

105 bits: 110 0010 (b) 110 0101 (e) 110 0101 (e) 111 0000 (p) 010 0000 (space) 110 0010 (b) 110 1111 (o) 110 1111 (o) 111 0000 (p) 010 0000 (space) 110 0010 (b) 110 0101 (e) 110 0101 (e) 111 0010 (r) 010 0001 (!)

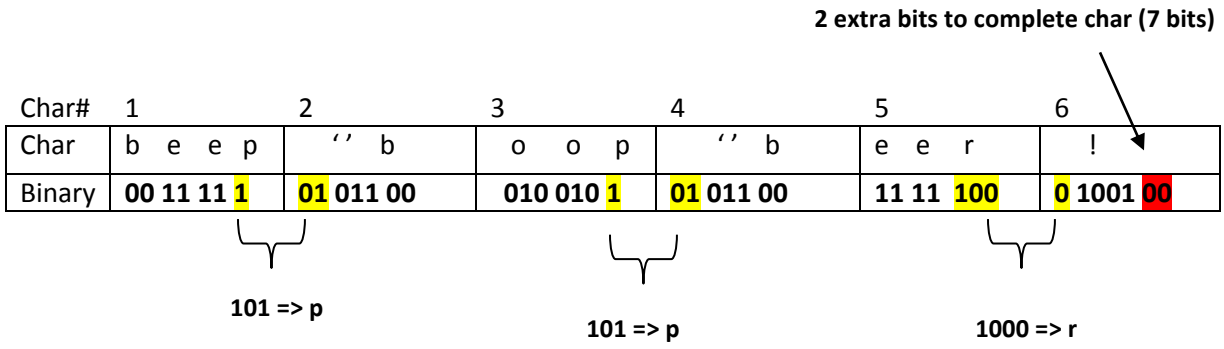
Encoded 40 bits: 00 (b) 11 (e) 11 (e) 101 (p) 011 (space) 00 (b) 010 (o) 010 (o) 101 (p) 011 (space) 00 (b) 11 (e) 11 (e) 1000 (r) 1001 (!)

Build Compressed file:

Use Original String & Huffman Table to construct **encoded.txt**

Create 7 bits character using Huffman Table. When 7 bits are completed, and code is incomplete, remaining bits will move to next character.

Since the character encodings have different lengths, often the length of a Huffman-encoded file does not come out to an exact multiple of 7 bits. Files are stored as sequences of character (7 bits), so in cases like this the remaining digits of the last bit are filled with 0s.



character Number	Binary	ASCII value / char
1st	**00 11 11 1**	62 / ‘ > ’
2nd	**01 011 00**	44 / ‘ ’ ’
3rd	**010 010 1**	37 / ‘ % ’
4th	**01 011 00**	44 / ‘ ’ ’
5th	**11 11 100**	124 / ‘ | ’
6th	**0 1001 00**	36 / ‘ $ ’

Write these characters (>’%’|$) to encoded file.

Uncompressing the file:

Read the **character** from encoded.txt and convert in 7 bits binary and search original character from **Huffman tree**, write that character on to reconstructed.txt

What to do in the Assignment?

Input: You will be provided a text file namely original.txt.

Tasks:

- Read the data from "original.txt" file
- Calculate the original size of the file in bits.
- Find out how many unique characters are there in the file.
- Find the frequency of each character. (print characters and their frequencies) **Marks: 5**
- Build Huffman tree using Ordered Queues (Note: tree should be link list Structure & Queue should be Array based Structure). Print the tree inorderly (including '*'). **Marks:15**
- Construct a Huffman Table which will contain a character and its corresponding encoding. Print this table. **Marks: 30**
- Store encoding of each character in a separate file called **encoded.txt** and print the size of encoded.txt file in bits. **Marks: 30**
- Next using the "encoded.txt" reconstructs the original file and call it "**reconstructed.txt**". **Marks:20**
- Check (using programming) whether "**original.txt**" and "**reconstructed.txt**" are exactly same or not. Print complete report which includes original.txt size in bits, encoded.txt size in bits, reconstructed.txt size in bits, and print those characters whose are different including line & column numbers. **Bonus Marks:20** (which will be added only in assignments)

Note: Provide Menu for all the above tasks.