This assignment if completed will get you a max of 20 bonus marks which may be used to cover up for some other assignment. 15 for completion and a max of 5 marks for the README.txt mentioned at the end.

Idea of Compression and Decompression:

First let me introduce you guys to the concept of data compression. Assume you have a string consisting of smaller alphabets only (a-z). Each character takes 8 bits or 1 Bytes when stored in the memory. But notice something, that there are only 26 distinct characters. So max number of bits required to distinguish each character is 5 as 2^5 = 32. In other words, if we map each character starting from 'a' to a bit sequence of length 5, we can distinguish each character. Now lets see the benefit of this.

Assume you have to a string of length 10,000. In other words, you would require **10,000*8 bits** to store this string in the memory. But if we instead store it using the concept given above, we can store this string using only **5*10,000 bits.** In other words, we saved **30,000 bits** or **3750 Bytes** of data using this simple scheme. This simple example represents how with a very simple substitution strategy, how you can save some space.

Now when you store your data in this way, you should be able to see that decompressing is not very hard either. You just have to take a group of 5 bits and map it back to its original character. For example, a sequence of 5 0's means its an 'a', a sequence represented by 00101 means an 'e'. So decompression is pretty simple in this case again.

Part 1 (Marks: 30)

Now in this part of the assignment, you have to implement a class which represents a compressor and decompressor.

Lets call our class by the name: **Transformer** and define the following two operations on it:

A **compress** method which takes a string of lowercase english alphabets as an argument and returns a compressed representation of the string.

A **decompress** method which takes a string (compressed representation of some string) as input and returns the original string back.

Part 2 (Marks: 70)

Now that we have the basics down, let's get into something complicated and more general. The above discussed simple method for compression took advantage of the fact, that there were less bits needed to identify each character uniquely than were actually used. Now get into something a little bit more complicated. But before I discuss the idea of compression, let me elaborate a little on general Binary Trees and the operation you will need to implement to make that idea work.

Take the following structure:

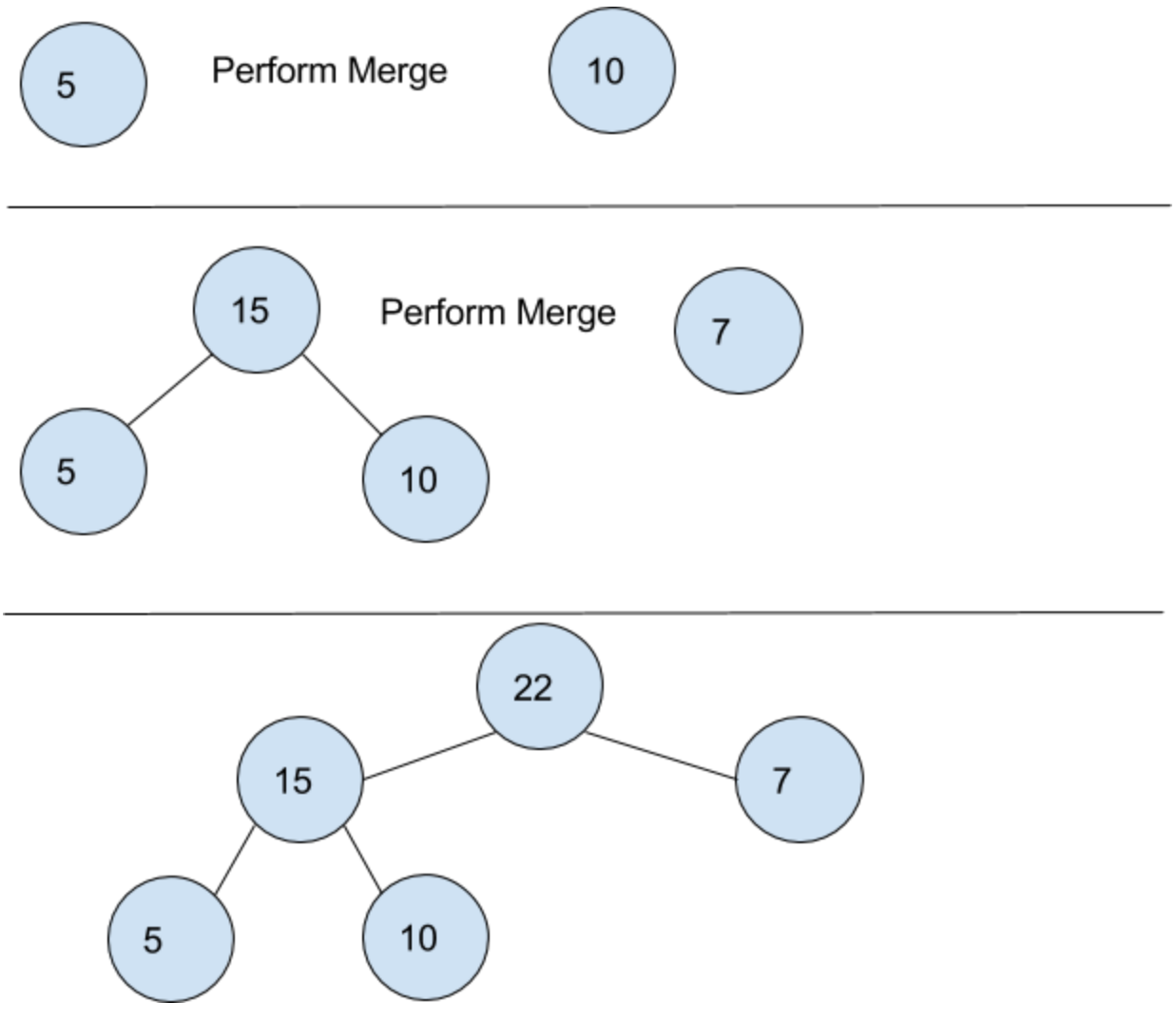
```
Struct  Node {
        Int weight;
        Char rep;
        Node* left;
        Node* right;
}
```

In a binary search tree, you inserted the element based on the relative weight of the item. But a general tree is not necessarily ordered like a search tree. In other words, an element can be anywhere in the other tree. Implementation of such structures differs only in the fact, that when you search for the element being inserted you do not eliminate some of your search space but search both the left and right subtrees for the element instead.

Now each Node of a tree is a tree in itself. Now lets call the operation we are going to discuss **'merge'**.

This operation will takes pointers to two arbitrary Nodes and returns the pointer to a new node whose weight will be equal to the sum of weights of these two nodes, and this new node will contain both nodes as its children.

Let's take an example to clarify:


5
Perform Merge
10
15
Perform Merge
7
5
10
22
15
7
5
10

So now extend the above struct Node in the following way:

```
Struct Node {
        Int weight;
        Char rep;
        Node*  left;
        Node* right;
        Node();
        Node *merge(Node *);
}
```

The merge operation in the above node merges the current and an external node and returns the pointer to a new node generated according to the above given rule.

Make sure your code works till here before going further. The weight for this part is about (20 Marks).

Now that's enough with the trees. The compression part begins now. The idea we are going to use for our compression and decompression is called **Huffman Encoding** which depends on the character frequency in your document. It is an algorithm for lossless data compression, which assigns a bit sequence to each character such that no bit sequence is a prefix of bit sequence of some other character (Write some cases on a paper and think why this identifies each character in a unique way). Now you are going to implement a class: let's call it **Transhuffer**, however, unlike the previous one this class will take an argument which is the name of the file you are going to compress.

Again it is going to have two public operations just like the basic compression class, **compress** and **decompress**.

compress will require the following steps (list can refer to a linked list or an array but it's better if you choose linked lists for your implementation):

1. Write a function which calculates the frequency of each distinct character in the file and returns a list of Nodes, where each Node represents a distinct character (this is where you can use the rep variable in the Node struct) and its weight is same as the frequency of that character in the document. The list should contain the nodes in **sorted order** based on node weights. (**For ease use a priority**

queue, you don't need to implement one yourself. You can use priority_queue class of c++ standard library)

2. Now take the two smallest nodes from the list (remove them from list too) or priority_queue, merge them into one node as described above and reinsert the new node into your list (again in sorted order) or priority_queue.
3. Keep repeating step 2 until there is only one node left. This node represents what we call a **huffman tree.**
4. Now this is the part where we assign each character a unique bit sequence. To do that, start traversing the tree. Label each left turn you make in the tree as '0' and each right turn as '1'. If you keep following these turns, you will get to the leaves while obtaining a bit sequence of multiple 0's and 1's. Each bit sequence will be able to identify a character uniquely and will not be a prefix of any other character. (**Think about how you would store these mappings. You can use this tree to get to these character mappings or store it in some other way. Look up map class of c++ and how to use it. That may help**).
5. Unlike the simple compression class, make a new file named as **"compressed.(extension_of_original_file)".** Ignore the parentheses around the extension part in the file name. The file should be a compressed version of the original file.

You can have private members of the class which may represent each operation described above. It is not necessary though. Implement it however you want.

decompress is not much different from before. You just have to traverse the huffman tree, and for any sequence where you reach a leaf you have to replace that bit sequence with the character in the leaf node. Output your result of this function to a file **"decompressed.(extension_of_original_file)".** This file should be exactly same as the original one.

One thing you should note is that you can only call decompress after you have called compress for the same run of the program. If you call decompress directly, there can be no output since there is no huffman tree in the memory yet. So we don't know the mappings

Write a simple main function, to show that your code is working. Submit your working code files along with all needed ones (if any).

Note: Think about why we performed each step of the compress function and why we used different classes like maps, priority_queues etc. and put it in a **README.txt** file and submit it along with your code to get a max of **5 bonus marks.**

Some Things which you might not have studied in class but may help:

1. **map class c++.**
2. **Priority queues**
3. **Bitset class. You can use vector of booleans too.**

We will discuss some of these in the tutorial to clear the confusion.