CS2040S: Data Structures and Algorithms

Problem Set 4

Due: TBA

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so by leaving a comment at the start of your .java file. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

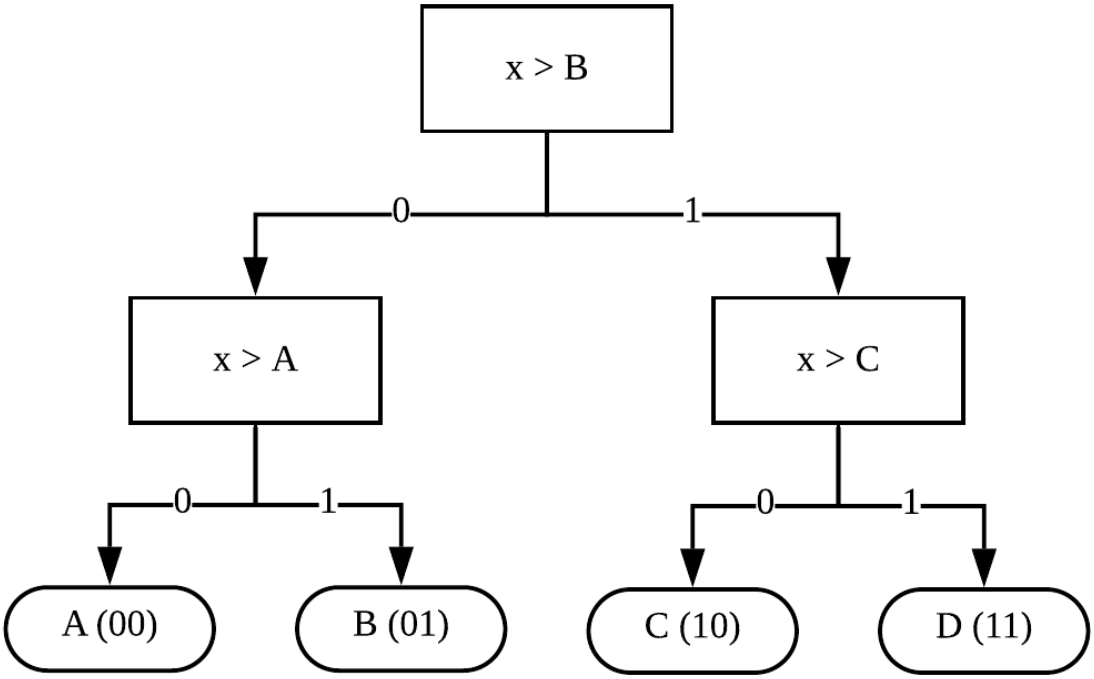
Problem 4. (Make it Smaller)

The goal of this problem is compression: take a file and make it smaller. We will take every byte of the input file and re-encode it so that the entire file is smaller.

By default, each ASCII character is stored using 8-bits. But this is quite inefficient: some characters (like 'e') appear much more frequently. Others appear much less commonly. We could save space by encoding an 'e' using only 3 bits, while using 12 bits to store every 'z' in the document. To be more precise, imagine a document has W characters in it, and character c_i appears w_i times in the document. Then the ideal thing to do is store character c_i using $\log(W/w_i)$ bits.[1]

In this problem, we will use a binary tree to generate the codewords.

Each leaf in the tree will represent a symbol in the input file. The codeword is found by looking at the path from the root to the leaf containing the symbol: start at the root, and every time you go left, add a '0' to the codeword; every time you right, add a '1' to the codeword. When you reach the leaf, return the codeword itself.



A key property of this type of code is that they are prefix-free codes: if c is the codeword for some symbol, then c is not a prefix for any other symbol (e.g. $\{00, 01\}$ is prefix-free but $\{00, 001\}$ is not prefix-free because 00 is a codeword and a prefix of 001). This is very useful because it means that

[1] The reason why that is optimal is related to the idea of entropy.

if we ever see the codeword c, we know exactly how to decode it (without worrying that it is part of some longer codeword).

In this problem, we will provide you with most of the mechanism for encoding and decoding files. The only part you have to do is the part related to tree manipulation: build the tree, and implement two query methods: one that translates symbols to codewords, and one that translates codewords to symbols.

There are two versions of this problem: the easy version and the harder version. You are required to do one of them (and a small bonus for doing the harder one).

- In the easier version, you can ignore the frequency with which each symbol appears. In this case, you only have to build a balanced tree. However, the compression performance will not be as good, since you are not getting any benefit from prioritizing repeated characters.
- In the harder version, you will weight the tree according to the frequency that each items shows up. This will get you close to an optimal encoding.

The easier version should be implemented in the `UniformTree` class, while the harder version should be implemented in the `WeightedTree` class.

Problem 4.a. Your first task is to implement the `buildTree` routine. In both cases the goal is to build a tree where all the symbols are stored at the leaves (i.e., the internal nodes do not contain any symbols, just guides for searching). The tree should be in BST-order.

It is important that the symbols are at the leaves, since that ensures that no codeword is a prefix of another codeword!

Easier version. The `buildTree` method is called in the constructor. An array of bytes is passed in, each representing a unique symbol in the input file. When the `buildTree` routine finishes, the member variable `root` should be the root of a balanced binary tree. The tree should be in BST-order (with respect to the symbols) so that we can search the tree for a symbol efficiently. We have included a `TreeNode` class as part of the UniformNode class; you may want to look at that. (You should not need to modify it.)

You can assume that there is 2^n unique characters, making it possible to build a full binary search tree.

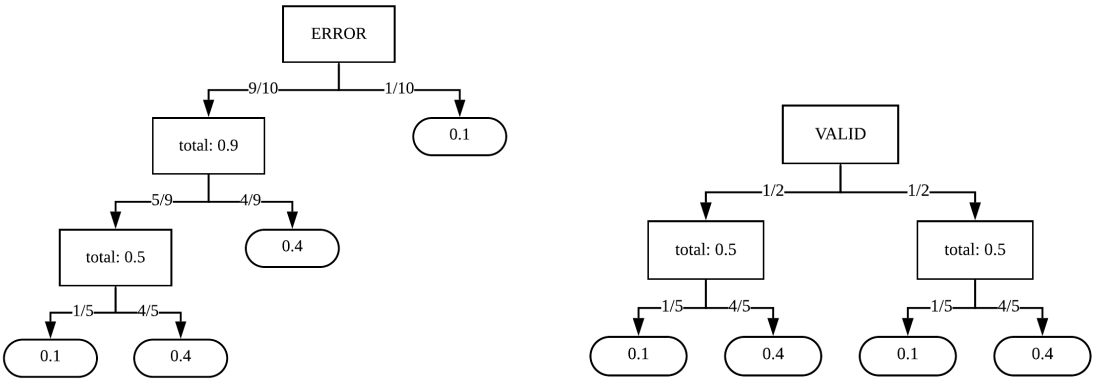
Try to make your tree construction algorithm as efficient as possible, e.g., linear time.

Harder version. In this case, the `buildTree` method takes as input an array of pairs: the symbols and their weight (i.e., the number of times it appears in the input document). Again, you want to build a tree that is in BST-order, so that it can be efficiently searched. The `Pair` type supports the Comparable interface, so you can sort it using system sort (`java.utils.Arrays.sort`) and it will sort by the symbol.

However, now the tree should be *weight-balanced* rather than simply balanced: the sum of the weight on the left side should be about equal to the sum of the weight on the right side.

Of course, exact balance cannot always be achieved because weights are discrete. Thus the goal is as follows: if W is the weight of some tree node v, ensure that neither the weight of v's left child nor that of v's right child exceeds $2W/3$.

There is one exception to this rule: if there is one item x that is of size at least $W/3$ all by itself, then the side containing that item may have more than $2W/3$ weight. However, at the very next level of the tree, x should become a leaf.



In the above example, the node on the right is a weight ratio that is more than $2W/3$, and is a intermediate node, so it is not allowed. However, in the second example, although the weight is 0.1 and 0.4, this is acceptable because the 0.4 is a single node on its own.

If you do the construction properly, an item with weight w_i should be at a leaf of depth $O(\log(W/w_i))$, (i.e., an optimal depth), where W is the total weight of the tree,

Try to make your tree construction algorithm as efficient as possible, e.g., $O(n \log n)$ time.

Problem 4.b. Your second task is to implement a query to find a codeword. The `queryCode` method takes a key as an input (in the form of a byte) and returns the codeword, in the form of a boolean array. (We will think of the codeword as a boolean array because we will write it to the file as a sequence of bits.)

You should be able to implement this by walking the tree from the root to the leaf of the appropriate symbol. If the symbol is not found at a leaf, then return null.

For the codeword returned, if `code` is the boolean array, then `code[0]` should be the first bit of the codeword (i.e., the step taken from the root), while `code[1]` should be the second bit of the codeword, etc.

Implement the query in an efficient manner.

Problem 4.c. Your third task is to implement a query to find a symbol, given a codeword. The `query` method takes a boolean array `code` and an integer `bits` as an input. The method will then output a key (in the form of a byte). The `code` array is only valid in the range from `code[0..bits-1]`. (The rest of the array is declared in advance to act as a buffer, since we do not know how big the codeword will be in advance.)

The goal, then, is to lookup the codeword `code[0..bits-1]` in the tree and if the result is a leaf, return the symbol found. Otherwise, return null.

For the codeword, if `code` is the boolean array, then `code[0]` should be the first bit of the codeword (i.e., the step taken from the root), while `code[1]` should be the second bit of the codeword, etc.

Implement the query in an efficient manner.

Once you have implemented the above three methods, the compressor and decompressor classes should work. Simply update the input and output files in the main routine of each of them and you should be able to compress and decompress accordingly. How good a compression rate do you get? (For text files with 8-bit ASCII text, I would expect you can save at least 3 bits per character.)

Problem 4.d. (Optional.) You can get better compression by being a little more clever. There are several things you might try. A natural thing to try is to build a Huffman Tree. How much better performance do you get? Another idea is that you might do better in text documents if you looked at words instead of characters: what if you map each word in the text to a leaf in the tree? Do you get better compression? What is the best compression you can get for, say, Hamlet?