CS145 – PROGRAMMING ASSIGNMENT

HUFFMAN CODING

OVERVIEW

This program focuses on nodes, priority queues, and tree manipulation, maps, string, files, and program coding in general.

INSTRUCTIONS

For this assignment you will be generating all code on your own. You will be submitting two primary files, and then another optionally another node file.

The files you will need to submit:

- `CodeToText.java`
- `TextToCode.java`
- `Node.java` - *maybe*

HUFFMAN CODE

In the general word of computer science we use the ASCII code to turn characters on the computer screen into binary for storage in memory. The ASCII code was developed in 1963 and encoded 127 "characters" into 7 bit representations. This code was then expanded upon in 1992 with the introduction of UTF-8 encoding which allowed for 1/2/3/4 byte representations (8/16/24/32 bit).

However the thing about these codes is that each character requires the same amount of space, so the most common character and the least common character require the same number of bits.

However in 1952 when memory and storage space was extremely primitive and expensive, David A. Huffman of MIT developed an encoding idea that was based on the relative frequency of each symbol. The idea being that the most common symbol would be given the smallest number of bits, and the least common symbol would be given longer bits.

In this way, storage space would be saved, and at the time, saving even a single bit was valuable.

However the major downside to this, is that each and every document would develop its own code, one that changed based upon the number of times a particular symbol came up. In common English, the letter "e" is the most common letter, so it would tend to have a small encoding, but there is a novel called Gadsby that is 50,000 words and uses the letter 'e' 4 times. So the Huffman Coding of this would give 'e' a abnormally large number of bits compared to normal writing.

IMPLEMENTATION DETAILS:

You will be writing two programs.

- Program 1
  - Titled - `TextToCode.java`
  - This program will require a customized node class. You have the option of adding the node class to main class, or creating a second file for it.
  - This program will ask the user for the name of a text file to read, and will then generate the Huffman Code for that document, and the print off two files.
    - The first file will be the code itself.
    - The second file will be the encoded file after applying the code to the text.
- Program 2
  - Titled - `CodeToText.java`
  - This program will read both the code file and the encoded file, and print the decoded file to the screen.
  - Note that this program doesn't write any files.

TEXTTOCODE.JAVA

To create a Huffman code you will be mixing up and using a Priority Queue and a type of tree. You should use the built in Java Priority Queue, but you will need to hold a custom tree in your code.

Note that due to subtle differences in execution, different versions of this program may output a slightly different code each time you run it. **That is fine**, as long as when you combine the code + huff encoding, the output should be the same.

THE NODE CLASS

Your node class will be the node of a tree class, but it will also contain two pieces of data. One piece of data will be the Character, and the other will be the frequency. It will then also have a left and right child pointers.

The node will also need to be comparable to be put into the priority queue.

I would suggest the following methods for your node:

- A constructor that sets both the `Character` and frequency
- A constructor that sets the frequency and the `Character` to null.
- A toString() method that prints both.
- A compareTo() method that compares based on the frequency and ONLY the frequency.
- An isLeaf() method to determine if the node is a leaf or not.

HOW TO CREATE A HUFFMAN CODE

The instructions to create a Huffman code are as follows:

1. Ask the user for a text file.
   a. Make sure the file name they type in ends with "`.txt`", repeat the question if it isn't.
   b. So it should be `"FILENAME.txt"` or something similar.
      ❖ NOTE: The name might not be "filename" !!!!
2. Open the file for reading character by character.
   a. This will require the use of FileInputStream instead of a normal Scanner for file reading.
      ❖ `FileInputStream x = new FileInputStream(File F);`
   b. Then to read character by character:

```
❖   while(x.available() > 0) {
❖       char c = (char) x.read();
❖   }
```

   c. You will have to adapt the code above to work with a try/catch here in a way similar to how we do a scanner.
   d. Take each character and add it to a list.

3. Using your list, make a map of each character and its frequency
4. Create a Java `PriorityQueue<Node>`.
5. For each character in the map, create a node with the character and the frequency and add that node to a priority queue.
6. While the PriorityQueue has a size greater than one:
    a. Remove two nodes from the Queue.
    b. Create a new node with a null character, and a combined frequency of the two nodes that were removed.
    c. Connect the two removed nodes to the left and right of the new node. *Order doesn't matter*.
        - ❖ So for example, if you remove `['r' / 45]` and `['t' / 55]`. You will create a new node `[null/100]`, connect the 'r' node to left and the 't' node to right.
    d. Add the newly created node back to the PriorityQueue.
7. Once the PriorityQueue has a single element in it, that last node is the root of a tree that contains all the letters, and all the connecting nodes.
8. Create a Character to String Map #2.
9. Then starting recursively at the root with an empty string "":
    a. If the current root is null return
    b. If the current root is a leaf, add the current root character and the current string to Map #2
    c. If the current root is not a leaf:
        - ❖ recursively go right with a "0" added to the end of the string.
        - ❖ recursively go left with a "1" added to the end of the string.
10. When you are done, you should have a map with every character and the Huffman String that goes with it.
11. For every entry in the map print two lines to a file called `"FILENAME.code"`
    a. Line one will be the character **cast** into an int. So for example don't print the character `'a'` print 65. Don't print `space`, print the number 32.
    b. Line two will be the string representation of the code you get from map.

12. Go back to your original list. Take every character one by one, and convert it to its corresponding string, and print it to a file as one long continuous line.
    a. This output should go to a file called `"FILENAME.huff"`
13. This should complete program one.

CODETOTEXT.JAVA

1. This program should start by asking the user for a file name. Make sure that the file name does NOT contain any periods. Repeat until it doesn't have a period.
2. Using the provided file name, use string techniques to create the file names that you will use for data input. (The `.huff` filename and the `.code` file name)
3. It should then check for the existence of both files.
    a. If you are missing one or both of the files, tell the user and exit.
4. If both files exist, it should read the `".code"` file and convert it into a map.
    a. As long as there is a line, read the first line, and cast it back into a character.
    b. Read the second line which is the string representation.
    c. Add the representation and the character to the map.
        i. Note that since this is decode, it is easier to go string → character.
5. Now open the `".huff"` file and read the data as a single large string.
6. Start with an empty string, and add a single character from the data string to the mini string.
    a. If the mini-string is in the map, print out the corresponding character and then reset the mini-string to empty.
    b. If the mini string is not in the map yet, add another single character to the mini-string and repeat.
7. This should decode the string/file for you.

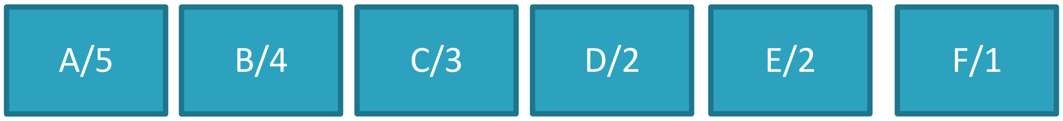
HOW THE HUFFMAN CODE WORKS

Imagine you have the following string of data:

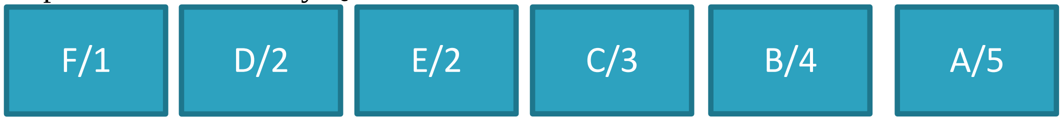
```
"aaaaabbbbcccddeef"
```

First we generate a frequency map: (a,5)(b,4)(c,3)(d,2)(e,2)(f,1).

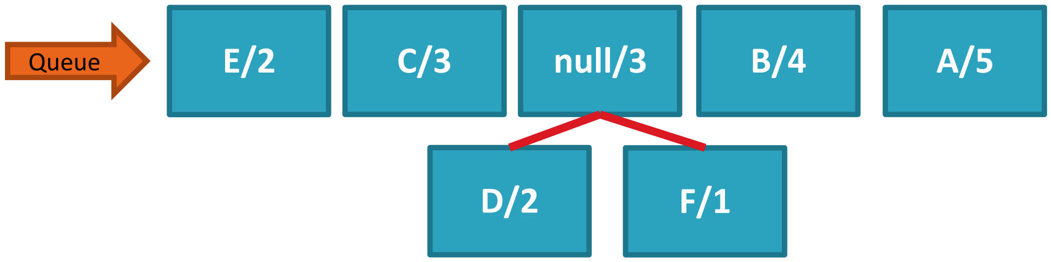
Then we create nodes of these.



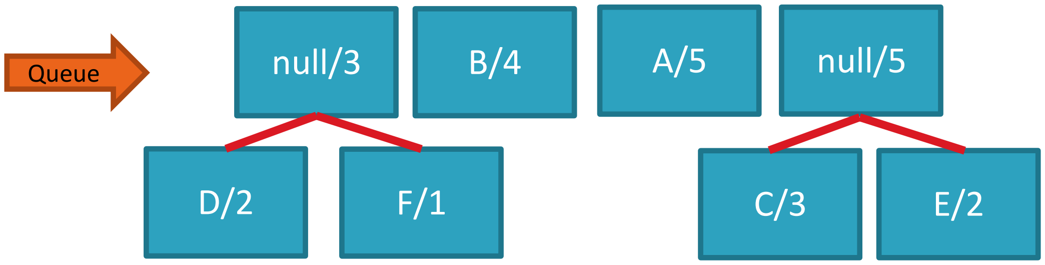
Then put them in a Priority Queue



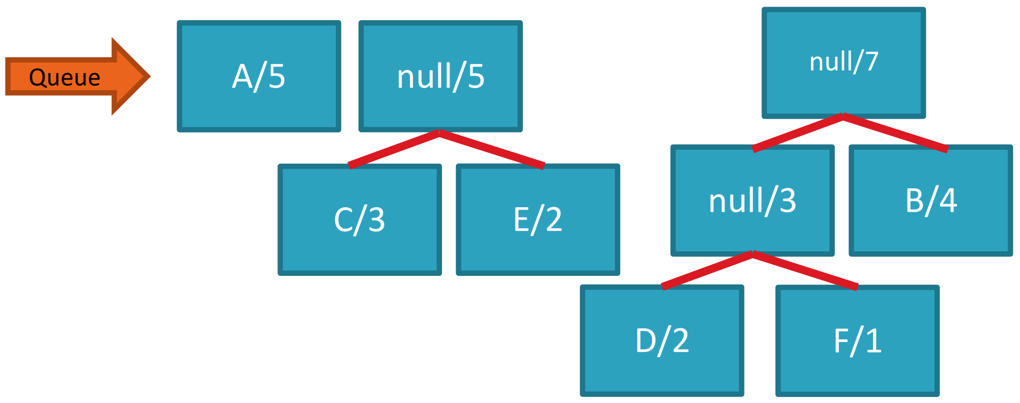
Take off 2, combine them into a new node, and reinsert.



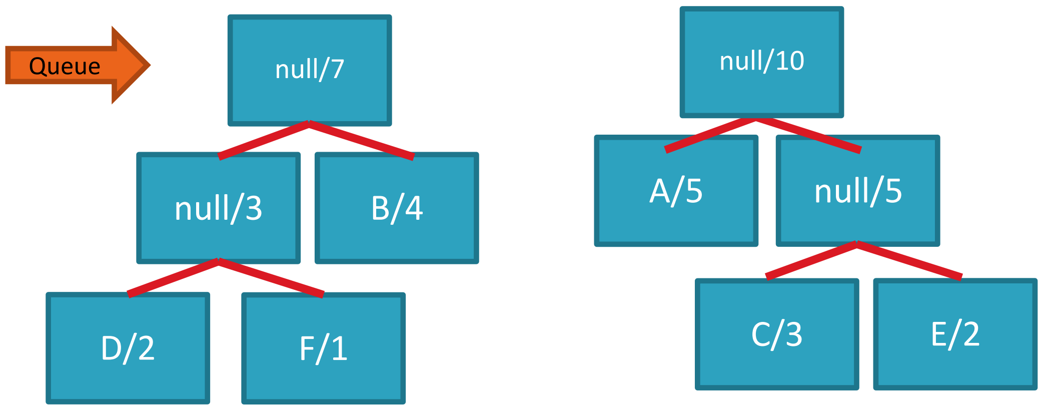
Take off 2, combine them into a new node, and reinsert.



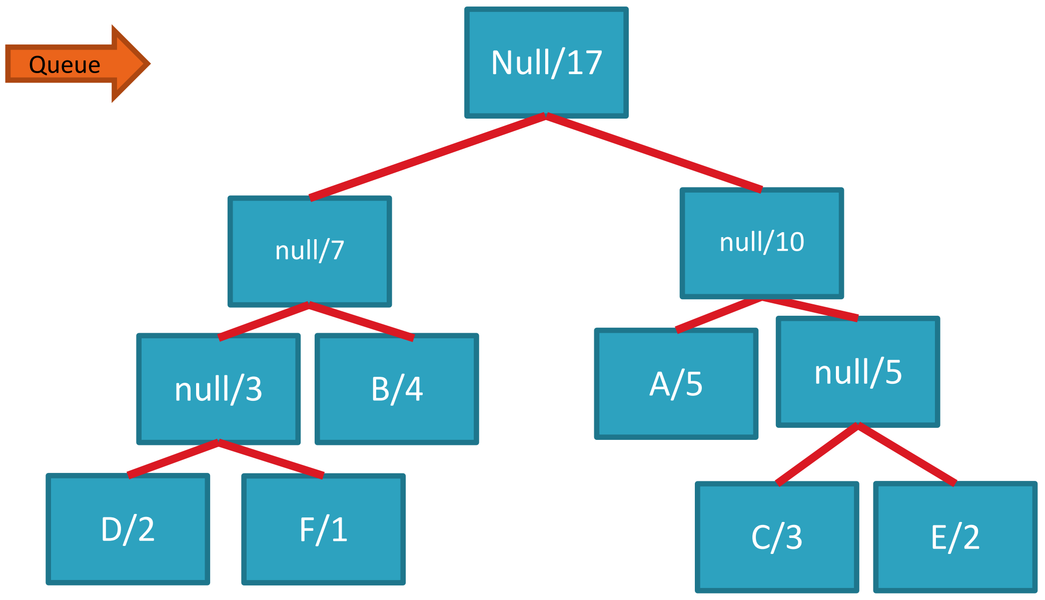
Repeat (2 off, make new node)



Repeat (2 off, make new node)



One Last time



Now:

If we let left = 1 and right = 0

D = 111 F = 110 B = 10 A = 01 C = 001 E = 000

and `"aaaaabbbbcccddeef"` becomes 01010101011010101000100100111000000110 or something similar.

FINAL NOTES

Enclosed in this program are some sample files for you to use to test your program.

Note, every program might implement things a little bit differently, so your `TextToCode` output and my `TextToCode` output, or your `TextToCode` output and another students `TextToCode` output might not match 100%, that is acceptable.

But your code should be able to decode data from anyone output, if provided the 2 files.

`PandPchapter1.txt` is the first chapter of Jane Austin's Pride and Prejudice, which is a good size data to check your work.

`Hamlet.txt` is the entire play of Hamlet as written by W. Shakespeare. Test this at the end to make sure your program can handle it. It might take up to a full minute to run, but more than 60 seconds is too long.

`Short.txt` is a very short text file to test, but it also has an accompanying `short.code` and `short.huff` that you can use to check your decoder.

Make sure to look at `short.code` and `short.huff` in a text editor to make sure your output matches the samples or is similar, remember, it might not be 100% the same.