Huffman hints

1. Use small files when initially writing / debugging your program! Trying to debug your program while attempting to encode "War and Peace" will… not be fun.

2. Use the BitInputStream and BitOutputStream classes to read/write *bits* (not `char`s/`int`s/Strings) to/from a file. Characters are stored using 8 bits; writing an integer literal 0 or 1 to file would not use one bit, it would use, e.g., 00000001 for char literal '1'.

3. We'll assume files will only contain standard ASCII characters, decimal values 0 – 255 (an "alphabet" size of 256).

4. You can use a `for` loop to quickly go through the first `N` characters in the ASCII table:
   ```
   for (char c = 0; c < N; c++) //whatever
   ```

5. Use a priority queue when building the compression tree to ensure the highest frequency characters use the least number of bits (rather than scanning through and finding the next node yourself).

6. Your tree nodes must be Comparable, based on the frequency count of the character they store. The Comparable interface supports generics, so you should supply the type parameter when declaring that your class implements Comparable (e.g. `class Node implements Comparable<Node>`).

7. "Interior" tree nodes (non-leaf nodes that lead to character nodes) can logically store anything but valid input text characters (along with the frequencies). However, it is suggested that you store the value `'\0'` in these nodes. This is the default value for `char`s, sometimes called the null character (not the same as `null`, an unset object reference) - useful for representing nothing (the absence of a character).

8. After you've built the tree, it might be helpful to put the characters' Huffman codes into a convenient data structure, e.g. a map or array, for quickly looking up the encoding for a particular character (rather than traversing the tree for each character).

9. Write the "code file" (the "<filename>.code" file that contains the Huffman encoding strings for individual characters) in a human-readable format (i.e., use a PrintWriter in the normal way).

   Putting this information at the beginning of a compressed file is much fancier, but significantly complicates things. Keeping the codes file human-readable is totally fine at this point.

10. The methodology used for Huffman coding results in a "prefix-free" code. A prefix-free code is one in which the bit coding sequence representing some character is never a prefix of the bit coding sequence representing any other character. For example, here is a possible bit sequence for a Huffman code on an alphabet with four characters where 'D' is the most frequent and 'A' is the least frequent:

    ```
    A 110
    D 0
    C 10
    B 111
    ```

    This is contrary to something like Morse code, which is *not* prefix-free:

```
E .
I ..
S ...
```

11. Here is an excellent, free, online Huffman code generator (also another option if the first link ever goes dead) for debugging purposes.

12. To view the Huffman encoded (compressed) text's binaries, you'll need to use a hex editor. There are a couple free ones on the web. Alternatively, you could just log your program's output to the console (with print statements) to see what's going on; probably a lot easier than messing with a hex editor.

13. Don't forget to `close()` your output streams! While the bit I/O classes are coded to close themselves, most output stream classes (e.g. PrintWriter) do not; you wouldn't see any output if they're not explicitly closed.

14. Help! My computer has *malware*. The output of my Huffman compression looks something like this:

    ```
    UÊĐ§?Hh, UŠŠq9¬N”%(˜Ê¥      ÜžàZÒÒ¯q•Tš¦s@®Đ
    ```

    You may or may not have malware, *that's your business*, but that's not related to this issue. The text editor is attempting to decode the binary stored in the generated file using **ASCII* codes**. However, the file isn't storing ASCII coded text, it's storing Huffman coded binaries (which are specific to the input text you used). As mentioned previously, if you want to view the binary values in the file, you'll have to use a hex editor. It would probably be easier to log things to the console when debugging.

    It can be challenging to debug the decoding part of the assignment because the encoded files are not readable in a normal text editor and because the character boundaries are not obvious. You'll have to find a way to overcome this. Follow your *heart.*

    **Actually probably ANSI or Unicode/UTF-8 or another popular encoding scheme, but the idea is the same*