EECS 233 Programming Assignment #2
100 points
Due October 19, 2018 before 11:59pm(EST)

In this assignment, you will implement the Huffman encoding of English characters, using a combined list and binary tree data structure. The method should have the following signature:

```
public static String huffmanCoder(String inputFileName, String outputFileName);
```

which will read and compress an input text file inputFileName, (based on the Huffman encoding produced on the input file) and output the compressed file in outputFileName

NOTE: that these strings represent the file names, not content – you need to perform file I/O

The method will output the outcome of its execution, e.g., "OK", "Input file error", etc. The output file can simply contain the characters "0" and "1", instead of the binary bits. It means you are not required to use sophisticated binary bit operations to actually store the encoded characters.

NOTE: Since, you will be translating each character into a sequence of zeros and ones, the sum representing the full character, your output file size will actually be significantly larger than the input file. *Therefore, Decompressing a file is not required.*

The program should also print out

(a) The Huffman encoding of characters in the form of a table of character/frequency/ character/frequency/encoding triples – one triple per line, with ":" separating the elements, e.g.:

```
a: 315: 10010
b: 855: 1010
...
```

(b) The calculated amount of space savings (see below for details).

You will need to use a Java's standard list facility but implement your own **`HuffmanNode`** class with the following fields:

- ➔ `inChar`: the character denoted by the node. T*his is meaningful only for the leaf nodes of a Huffman tree so interior nodes can have garbage here.*
  - ◆ Storing garbage does not represent good programming style. You need to distinguish between a meaningful character and "nothing". Use the **Character** wrapper class instead of the primitive "char" type to account for any and all valid character types. This way, you can use "null" to represent "nothing".
- ➔ `frequency`:The number of occurrences of characters in the subtree rooted at this node.
  - ◆ For a leaf node, the frequency is the frequency of the character in the leaf node;
  - ◆ For an interior node, the frequency is the sum of all frequency values in the leaves of the subtree.
- ➔ `left`: left child of a node in the Huffman tree
- ➔ `right`: right child of a node in the Huffman tree

➔ NOTE: the above class contains no "next" field since you are using Java's native lists and the objects of this class are just elements in the list)

(c) Basic tree properties. Specifically, you need to print out:
   a. The number of leaves
   b. The depth of your tree
   c. The balance factor of your tree

you have to write your own code to implement these functions and return correct values.

The program should also have the following helper methods

- Scan the input text file to generate the initial list of `HuffmanNodes`
  - you will need to read the file and define a local table in this method to remember the frequency
- Merge two `HuffmanNodes` and return the combined HuffmanNode
- Run the Huffman encoding algorithm to produce the Huffman tree
- Traverse the Huffman tree to output the character encoding, and
- Scan the input text file (again!), produce the encoded output file, and compute the savings.
- Miscellaneous methods to calculate the properties(depth, number of leaf nodes, balance factor) of your Huffman Tree.

You can use whatever method of computing the character frequencies. However, once you have them, you do need to generate Huffman encoding, rather than assign variable-length codes to characters in some ad-hoc manner.

Testing

As a test of your program, you need to run it on a real sizable text file (but make sure you first debug your program on small text files). Please pick a **plain-text** (UTF-8) literary piece at ***http://www.gutenberg.org/browse/scores/top*** and use that piece for (a) generating the Huffman tree and (b) encoding the text using the generated encoding. (Again, make sure you use a plain-text version of your piece.) Please compute the space reduction you would achieve on your selected text. For this, you assume that in the original encoding, each character costs 8 bits in space, and then as you encode the text with Huffman encoding, you count how many bits each character will now occupy, and so you can maintain the running total to the end.

Notice: Before your programs are graded, I will feed random text files to your programs as input, so please make sure your programs work for any input files (plain text of course) before your submission. You can test your code by feeding multiple input files and check them if they are correct.

Submission and Grading

Name your file ***P2_YourCaseID_YourLastName.zip*** and submit your zip file electronically to Canvas.

A zip file containing:

1. All source codes, including sufficient comments for the TA to understand what you have done (code without comments will be penalized). This must include the main method. I will run your program using command:

   ```
   java HuffmanCompressor inputFile outputFile
   ```

   so please name your classes accordingly.

2. The text file you used to test your program (up to 50K in size).
3. The Huffman encoding table you generated using the above file.
4. The encoded text file.
5. A text file contains the calculated amount of space savings your encoding has achieved. (Again, this will not be reflected in the actual sizes of the original and encoded files because you are representing bits as entire characters in your output file.)
6. The depth, number of leaf nodes, balance factor of your Huffman Tree.

Rubric:

Task	Points
Scanning the input file and generating character frequencies	**10**
Producing the Huffman tree (including the HuffmanNode class specification, depth, number of leaf nodes, etc.)	**35**
Using the Huffman Tree to produce the encoding table	**25**
Producing the encoded output file and computing space savings	**15**
Calculate the properties of Huffman Tree	**5**
Style and completeness of classes/ submission	**10**
	100

Tips:

1. Gutengerg uses UTF-8 encoding for plain text. This is a superset of ASCII, and occasionally you may get some weird characters. If you read your file line-by-line or character-by-character (rather than by binary bytes), Java should be able to handle most of those weird characters automatically. In rare cases it may get you into trouble (e.g., when a character is encoded with more than 16 bits) so it's best to choose text in plain English, without characters with accents for example.

2. Some UTF-8 files include special characters in the first three characters. Again, if you read your file by characters this should be handled for you, but just be aware of this if you encounter an issue.