EECS 233 Programming Assignment #2

Due March 17, 2016 (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 in a compact way. In fact, because you will be translating each character into a sequence of zeros and ones, each being the full character itself, your output file size will actually be significantly larger than the input file. 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/encoding triples – one triple per line, with ":" separating the elements, e.g. (the encoding is imaginary, just an example):

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. This is meaningful only for the leaf nodes of a Huffman tree so interior nodes can have garbage here. But storing garbage rarely represents a good programming style. So you need to distinguish between a meaningful character and "nothing". To do so even when any value may potentially represent some valid character code you should use the **Character** wrapper class instead of the primitive "char" type. Then you can have "null" to represent "nothing".
- frequency: the frequency of occurrences of all characters in the subtree rooted at this node. For a leaf node, this 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

(Food for thought: why does the above class contain no "next" field as described in the lecture?)

Explain your choice of the Java List implementation (ArrayList vs LinkedList) in the comments. (Do not look for a deep reason why one of the choices will be unsuitable – both are actually workable; but because you have to pick one implementation, share your thoughts why you picked one and not the other.)

You will need to implement several helper methods, specifically:

- A method to scan the input text file to generate the initial list of HuffmanNodes (you will need to access a file and define a local table in this method to remember the frequency),
- A method to run the Huffman encoding algorithm to produce the Huffman tree,
- A method to traverse the Huffman tree to output the character encoding, and
- A method to scan the input text file (again!), produce the encoded output file, and compute the savings.

For all file I/O operations use Java's BufferedReader/Writer facilities.

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.

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** 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**.** **Important: Please either find a text up to 50K or truncate the text and use only the first 50K of it for this step! (Otherwise people running the blackboard will kill me…)**

Deliverables:

A zip file containing:

1. All source codes, including sufficient comments for the TAs to understand what you have done (code without comments will be penalized). This must include the main method. We 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. The calculated amount of space savings your encoding has achieved on your text file. (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.)

Grading rubrics:

Scanning the input file and generating character frequencies: 20 pts
Producing the Huffman tree (including the HuffmanNode class specification): 25
Using the Huffman tree to produce the encoding table: 25
Producing the encoded output file, and computing the space savings: 20
Style and completeness of submission: 10

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.