Huffman Encoder Part 2: Encoding/Decoding

In this two part assignment, we will implement an efficient scheme for compressing a text message called Huffman coding. A simple method to encode text as a string of 0s and 1s is to represent each character as a unique string of 8 binary digits (bits). A text message is thus translated into a string of bits. By exploiting the fact that not all characters appear with the same frequency in the text, we can encode rarely used characters with long codes and frequently used ones with short codes.

In part 1, we'll use a given a set of characters and their corresponding frequencies to create an optimal coding scheme: a special binary tree where the path to each leaf represents a different character. In part 2, we'll take a text file and, using a Huffman Tree, encode/decode its message.

Converter

We are going to add a new file to the project started in Part 1. This file will be called HuffmanConverter.java. It will take an ordinary text file, record the frequencies of characters used in that file, then use Part 1's code to find the Huffman encoding. Next, we will use these to convert our text file into its equivalent Huffman encoding. Lastly, we will compare the # of bits used in the Huffman encoding versus the standard ASCII encoding. (This always encodes 7-8 bits per character, regardless of that character's frequency.)

The text file used will have several lines - `\n` (newline characters) count as characters, as well as any letters, numbers, spaces, tabs, comma, periods, colons, etc. Also, there is a difference between upper-case and lower-case letters for this assignment. Hence, the frequency of `Q` is not the same thing as the frequency of `q`. Also, the text file must END with a newline character.

We can convert integers to characters and vice versa using Java's built-in ASCII conversion. We can convert an `int i` into a `char c` by doing `c = (char)i`. Similarly, we can convert a `char c` into an `int i` by doing `i = (int)c`. This allows us to use the arrays `count[]` and `code[]` to keep track of the count and Huffman encoding for each character. Hence, for some `char c`, `count[(int)c]` tells us how often c occurs, and `code[(int)c]` tells us the Huffman encoding for character c. However, please remember that there are only 256 (or even 128) legal characters on most machines (which is also the size of the ASCII table). Hence, we can't convert the number 300 into a character.

There will be a main method in the HuffmanConverter class to do everything we need. Our HuffmanConverter will run from the command-line as:

```
java <package.>HuffmanConverter message.txt
```

where `message.txt` is a regular multi-line text file containing a message, and NOT a single-line file listing the frequencies of letters as we did in Part 1. Therefore, we will be doing everything in Part 2 from the main method of the HuffmanConverter class rather than the HuffmanTree class, although we will make use of the HuffmanTree's objects and methods for several things (so as to reuse as much code as possible). Remember that we don't know the frequencies of the characters involved in advance as they are not given as was they were in Part 1. Instead, we have to figure out those frequencies based on how often each character occurs in the file itself.

If you're using an IDE (JCreator, Eclipse, NetBeans, etc.) be sure to select the option to execute with command-line arguments, and specify the name of the input file that way.

Once the Huffman encoding for each character is found, we can encode a message by giving the Huffman code for the first character, followed by the Huffman encoding for the second character, followed by the Huffman encoding for the third character, and so on. We can (and will!) also decode the encoded message by finding the character that matches each (uniquely-prefixed) Huffman bit-string. We can then compare our decoded message to our original input (which will, if we've implemented everything correctly, be exactly the same!)

The classes for HuffmanTree, HuffmanNode and BinaryHeap will be the same as those used in Part 1. For this assignment, we will be adding only the HuffmanConverter class which should look something like this:

```
public class HuffmanConverter
{
 // The # of chars in the ASCII table dictates
 // the size of the count[] & code[] arrays.
 public static final int NUMBER_OF_CHARACTERS = 256;

 // the contents of our message...
 private String contents;

 // the tree created from the msg
 private HuffmanTree huffmanTree;

 // tracks how often each character occurs
 private int count[];

 // the huffman code for each character
 private String code[];

 // stores the # of unique chars in contents
 private int uniqueChars = 0; //(optional)
```

```
/** Constructor taking input String to be converted */
public HuffmanConverter(String input)
{
this.contents = input;
this.count = new int[NUMBER_OF_CHARACTERS];
this.code = new String[NUMBER_OF_CHARACTERS];
}

/**
* Records the frequencies that each character of our
* message occurs...
* I.e., we use 'contents' to fill up the count[] list...
*/
public void recordFrequencies()

/**
* Converts our frequency list into a Huffman Tree. We do this by
* taking our count[] list of frequencies, and creating a binary
* heap in a manner similar to how a heap was made in HuffmanTree's
* fileToHeap method. Then, we print the heap, and make a call to
* HuffmanTree.heapToTree() method to get our much desired
* HuffmanTree object, which we store as huffmanTree.
*/
public void frequenciesToTree()

/**
* Iterates over the huffmanTree to get the code for each letter.
* The code for letter i gets stored as code[i]... This method
* behaves similarly to HuffmanTree's printLegend() method...
* Warning: Don't forget to initialize each code[i] to ""
* BEFORE calling the recursive version of treeToCode...
*/
public void treeToCode()

/*
* A private method to iterate over a HuffmanNode t using s, which
* contains what we know of the HuffmanCode up to node t. This is
* called by treeToCode(), and resembles the recursive printLegend
* method in the HuffmanTree class. Note that when t is a leaf node,
* t's letter tells us which index i to access in code[], and tells
* us what to set code[i] to...
*/
private void treeToCode(HuffmanNode t, String s)
```

```
/**
* Using the message stored in contents, and the huffman conversions
* stored in code[], we create the Huffman encoding for our message
* (a String of 0's and 1's), and return it...
*/
public String encodeMessage()

/**
* Reads in the contents of the file named filename and returns
* it as a String. The main method calls this method on args[0]...
*/
public static String readContents(String filename)

/**
* Using the encoded String argument, and the huffman codings,
* re-create the original message from our
* huffman encoding and return it...
*/
public String decodeMessage(String encodedStr)

/**
* Uses args[0] as the filename, and reads in its contents. Then
* instantiates a HuffmanConverter object, using its methods to
* obtain our results and print the necessary output. Finally,
* decode the message and compare it to the input file.<p>
* NOTE: Example method provided below...
*
public static void main(String args[])
{
//call all your methods from here
}
```

The output for the program should be the initial contents of the heap before converting to the HuffmanTree, followed by the Huffman encodings for all the letters (a simple call to the huffman tree's printLegend() method is fine), followed by the encoded version of the message using the Huffman Encoding you found followed by the # of bits needed to encode the message in ASCII coding (which is contents.length() times 8), and the # of bits needed to encode the message using Huffman coding (which is the length of the String returned by the encodeMessage() method). Lastly, you should print the decoded message, which should match exactly (character-for-character) the original message).

To help you out, included are two sample test cases and the outputs you should expect to see from them. The test cases are both love poems. One is from a woman to a man, the other is from a man to a woman. Both are taken from the website http://www.lovepoemsandquotes.com. These poems are merely test cases and do not (intentionally) portray the intentions of any instructors or students in this class.

Submission

You will submit your HuffmanConverter file, as well as the files in the project from Part 1.

Grading Rubric

Criteria

Programs that do not compile successfully will get no credit for the assignment

Proper implementations of the detailed classes and methods (4 points)

Project successfully encodes and decodes a text message from a file (3 points)

Good style demonstrated in the code; sensible formatting (1 point)