CSE 143, Winter 2010
Assignment #8-B: Huffman Bonus (up to +3 extra credit points)

Due Sunday, March 14, 2010, 11:30 PM (no late submissions accepted)

This "secret" extra program is an extra credit extension to your HW8 Huffman program. Turn in files named `HuffmanTree.java` and `HuffmanTree2.java` from the Homework section of the web site. You should also re-download `HuffMain.java`, `Bit(In/Out)putStream.java` and download `HuffMain2.java` from the web site. This problem is meant to be tricky. The TAs will provide less help on extra credit than they would on ordinary features.

NOTE: You must still turn in regular HW8 the normal way. We won't grade your HW8-B as though it were HW8.

Description:

Real compression programs do not require two separate files the way our Huffman program does. Storing character counts in a separate `.counts` file is clunky and is not a behavior that the client would appreciate. If you choose to complete this optional extension to Homework 8, you will merge your Huffman-compressed binary file and its associated `.counts` file into a single compressed archive. You will also write code so that you are able to decompress such a file.

In this assignment you will turn in a new class `HuffmanTree2` that is an extension of `HuffmanTree` by inheritance. `HuffmanTree2` extends and overrides the behavior of its superclass to cause it to compress files without the need for a separate `.counts` file. Specifically, your `HuffmanTree2` class must have the following new constructor and methods:

`public HuffmanTree2(Map<Character, Integer> counts)`

In this constructor you should initialize your Huffman tree, as was done in the superclass. If the tree is going to be used to compress a file, you will be passed a map of character counts as normal. However, if you are decompressing a file, the main program won't know its counts of characters because it will not have a `.counts` file; so in this case your tree will be passed a map with only a single entry: EOF. So when a Huffman tree is being constructed to decompress a file, the tree is essentially empty to start. You must wait until decompressing a file to restore your overall tree and encodings.

Note that subclasses' constructors must always call a constructor from their superclass in order to compile.

`public void compress(InputStream input, BitOutputStream output) throws IOException`

As before, in this method you should read the text data from the given input file stream and use your encodings to write a Huffman-compressed version of the data to the given bit output file stream. The difference in the new version is that you should now also write your encodings into the compressed file.

Exactly how you write your encodings into the file is up to you, but one way is to write some "header" lines of text at the start of your file where each line contains a character's ASCII value followed by a space followed by its binary Huffman encoding. For example, the following might be the complete contents of `example.huf` (the compressed version of `example.txt`) when output in "byte mode" (debugging mode, with `HuffMain`'s `DEBUG` constant turned on):

```
32 00
97 11
98 10
99 010
256 011

1110001110000101110011 00
```

Another option would be to write the characters and their counts as your header, rather than the encodings. Regardless of the exact format you choose, the key is that all relevant information you'll need for decompressing later must be written into one single file represented by the given bit output stream.

One way to write such lines at the start of a file is to use a `PrintStream` object. Construct a `PrintStream` by passing another `OutputStream` to its constructor. Then `println` text into the start of the file. After that, you can write the binary compressed contents of the file as normal. (The lines of encodings are written as normal ASCII text, not as binary data.) Notice that in the above example we put a blank line after the encodings are done so that later we can tell where the encodings end and the binary compressed contents begin. Don't call `close` on your `PrintStream` when you're done with it, because that would close the underlying `BitOutputStream` and prevent you from being able to write the binary contents. Call `close` only on the `BitOutputStream`, and only after you're done writing your binary compressed data.

`public void decompress(BitInputStream input, OutputStream output) throws IOException`

As before, in this method you should read the compressed binary data from the given bit input file stream and write a decompressed text version of this data to the given output file stream. The difference in the new version is that now your `HuffmanTree2` will have been constructed with an empty map of character counts, so you will have to be able to extract the character counts and/or encodings from the input stream in order to decompress the file.

You may assume that the input stream refers to a valid file that was compressed by your program.

Decompressing files is tricky, since the file contains a mixture of ASCII header information at the start and binary Huffman-compressed data afterward. Since the start of your compressed file presumably begins with some ASCII text data, it may be useful to read lines of that data one at a time. Normally you could use a `Scanner` to do this, but you should not open a `Scanner` on your bit input stream because it will "buffer" input and this will consume bits that you don't want it to. Instead, we have added a method to the `BitInputStream` class that you can call when decompressing:

`BitInputStream` **Method**	**Description**
`public String` **`readLine`**`()`	reads until `\n` character or EOF is seen; returns line without `\n` (or empty string if no input remains)

Instead of reading lines with `Scanner`, you can read them by calling `readLine` on the `BitInputStream`. Once you have grabbed a string for a line, it is perfectly safe to scan the contents of that string by creating a `Scanner` and passing that string as the parameter to the `Scanner`'s constructor. This would make it easy to tokenize the contents of the line and read those tokens as integers, strings, or whatever type you like. The `BitOutputStream` class has a `DEBUG` flag you can set to `true` if you want to see a printout whenever bits are read/written, which can help you debug your binary I/O.

You don't want to call `readLine` when reading your binary compressed data, only when reading your new header information. So you need to put some sort of clear marker at the end of your header so that your code will know when to switch to reading one bit at a time. In our example we use a blank line to indicate this.

There is one more subtlety about decompressing that you will have to handle. In the normal `HuffMain` client program, the special EOF character is handled for us and can generally be ignored. Part of the reason this works is because `HuffMain` tells the bit in/output streams what the binary encoding for EOF will be, so they can look for this pattern and stop reading/writing when they see it. But in this new version, `HuffMain2` doesn't know the encoding for EOF when decompressing; the encoding for EOF is only discovered by your `HuffmanTree2` as you are decompressing the file. Therefore, before you write your binary data to the output stream, depending on your implementation, you may need to call the `setEOFEncoding` method on the output stream and pass it the `String` storing the binary encoded version of the EOF character. If your map of encodings is stored in a variable named `encodings`, you would write the following in your `decompress` method just after you have finished reading the file's header and rebuilt your tree/encodings:

```
String eofEncoding = encodings.get(BitOutputStream.EOF);
((BitOutputStream) output).setEOFEncoding(eofEncoding);   // inform stream about EOF char
```

Style Guidelines and Grading:

Once again, this extension is for extra credit and is completely optional. The score you receive is out of 0 possible points, so any points you earn increase your overall homework grade and cannot hurt your performance in the course.

+2 points come from external correctness. One of these points is for successfully compressing files with counts/encodings attached as a header; the other is for successfully decompressing them. So if you want partial credit, you can submit a program that just compresses the files with a suitable header, and you'll be eligible for +1 point.

A third +1 point will be based on internal correctness and style: Can you implement this feature in a clean way? This includes basics like simple commenting and indentation, but also redundancy is very important. In particular you should still make use of existing functionality from the superclass as much as possible rather than re-implementing it again. You should not need to modify your `HuffmanTree` or `HuffmanNode` very much in order to make the assignment work, though you may make minor modifications to the superclass if it helps you avoid redundancy. In particular, you will probably need to change fields and/or helper methods from `private` to `protected` so your subclass can access them.

For reference, our `HuffManTree2` class contains 45 "substantive" lines.