CSE 143, Winter 2014
Programming Assignment #8: Huffman Coding Bonus (4 points)
Due Friday, March 13, 2014, 11:30 PM

There is an extra credit option for this assignment that is worth a measly 4 points. In other words, this isn't intended as an opportunity for you to increase your grade. It is intended as an extra coding exercise for those who are interested in exploring how to make their Huffman program behave better.

If you do the extra credit option, you are still required to complete the standard `HuffmanTree` and to submit it along with your `HuffmanNode`. So if you work on this, do so only after you have completed the standard assignment. To keep things clear, for this part of the assignment you should create a class called `HuffmanTree2`. You can copy your `HuffmanTree` class and modify it appropriately to get the initial version of this class.

The main goal of this variation is to eliminate the code file. When you use a utility like zip, you don't expect it to produce two output files (a code file and a binary file). You expect it to produce one file. That's what we'll do in this variation. To do so, we'll have to be able to include information in the binary file about the tree and its structure.

In the original version we had three programs: `MakeCode`, `Encode` and `Decode`. For this version there are two main programs: `Encode2` and `Decode2`.

In all, you will have to include the following three new methods in your class along with the other methods we had in `HuffmanTree`:

Method	Description
`HuffmanTree2(BitInputStream input)`	Constructs a Huffman tree from the given input stream. Assumes that the standard bit representation has been used for the tree.
`void assign(String[] codes)`	Assigns codes for each character of the tree. Assumes the array has null values before the method is called. Fills in a String for each character in the tree indicating its code.
`void writeHeader(BitOutputStream output)`	Writes the current tree to the output stream using the standard bit representation.

In the original `HuffmanTree` we had a method called `write` that would write the codes to an output file. Here the `Encode2` program does the actual encoding. It first reads the file and computes the frequencies. Then it calls your constructor to create an appropriate `HuffmanTree`. It has to have some way to find out what codes your tree has come up with so that it can encode the characters of the file. It does so by calling the assign method in your class passing it an array of Strings that are all `null`. Your method will replace the `null`'s with codes for the characters included in the tree.

The `Encode2` program also calls the method `writeHeader` in your class. The idea is to write to the bit stream a representation of the tree that can be used to reconstruct it later. As we did with the `QuestionTree` in assignment 7, we can print the tree using a preorder traversal. For a branch node, we write a 0 indicating that it is a branch. We don't need to write anything more, because the branch nodes contain no data. For a leaf node, we will write a 1. Then we need to write the ASCII value of the character stored at this leaf. There are many ways to do this. We basically need to write some bits that can be read later to reconstruct the character value. The value will require up to 9 bits to write (it would be 8 if it weren't for our pseudo-eof character).

We need to decide on a convention for writing an integer in 9 bits that we can reverse later when we read it back in. Below are the two methods you should use to accomplish this. They are inverses of each other in that `read9` will recreate what `write9` writes to the stream:

```
// pre : 0 <= n < 512
// post: writes a 9-bit representation of n to the given output stream
private void write9(BitOutputStream output, int n) {
    for (int i = 0; i < 9; i++) {
        output.writeBit(n % 2);
        n /= 2;
    }
}

// pre : an integer n has been encoded using write9 or its equivalent
// post: reads 9 bits to reconstruct the original integer
private int read9(BitInputStream input) {
    int multiplier = 1;
    int sum = 0;
    for (int i = 0; i < 9; i++) {
        sum += multiplier * input.readBit();
        multiplier *= 2;
    }
    return sum;
}
```

You should use `read9/write9` for input and output of the character (ASCII) values. Obviously when you are handling the 0 or 1 to indicate branch versus leaf, you can use `readBit/writeBit`.

`Encode2` produces a binary file that first has a header with information about the tree and then has the individual codes for the characters of the file. The `Decode2` program has to use this information to reconstruct the original file. It begins by calling the constructor listed in the table above, asking your class to read the header information and reconstruct the tree. Once the tree has been reconstructed, the program calls your decode method from the original assignment to reproduce the original file.

A collection of files necessary for this part of the assignment will be available called `ass8-bonus.zip`. It will include `Encode2.java`, `Decode2.java`, a starter version of `HuffmanTree2.java` that includes `read9` and `write9`, and examples of encoded input files called `short.bonus` and `hamlet.bonus`. `Encode2` should produce exactly the same output when used with your version of `HuffmanTree2`. There will be a separate turn-in for the bonus in which you submit `HuffmanNode.java` and `HuffmanTree2.java` (you shouldn't need to make any changes to your node class for the bonus, but it will be easier for us to grade if you submit both when you turn it in).