COMP210 - Project 1

Spring 2016

For this project you'll be implementing a basic Huffman encoder/decoder program.

The Program

For your first project you are to develop a CLI program that enables basic Huffman encoding and decoding for messages that contain alphabetic characters only. The program works exclusively through files and has three modes of use. The first mode is to compute a Huffman code for a given document and write that code to another file. This option is invoked by the argument *-c*. If the program executable were named huff, then *huff -c mymessage.txt mycode.txt* would read the message from the file *mymessage.txt*, compute the Huffman code for that document, and write the code to the file *mycode.txt*. The second mode doesn't stop at computing the Huffman code but instead goes on to write the complete message and it's code to a file. The command *huff -e mymessage.txt compressedmsg.txt* would compute the Huffman code for the message found in mymessage.txt and then write that code and the compressed message to *compressedmsg.txt*. The final mode of operation decodes a previously encoded message. The command *huff -d compressedmsg.txt decodedmsg.txt* would read the code and encoded message from compressedmsg.txt and then reproduce the original message in decodedmsg.txt. These three command formats are summarized in figure 1.

```
huff -c MESSAGEFILE CODEFILE
huff -e MESSAGEFILE ENCODEDMESSAGEFILE
huff -d ENCODEDMESSAGE DECODEDMESSAGEFILE
```

Figure 1: Three Commands for the Huffman Program

File Format for the Compressed Message

We're going to skip the actual compression part and instead simply write a string of the characters 1 and 0 that would get written to memory if we compressed the file. We'll refer to these strings as *binary strings*. The end result will be a larger file since we'll swap one character in the original message for multiple 1s and 0s in the "compressed" message. The smallest possible unit that Java will let you read/write from a file is 1 Byte. If you want to attempt the actual compression, then you'd have to combine all the bits, possibly pad it to get an exact multiple of 8, and then break that message up on byte boundaries. You're more then welcome to give this a go, but it is not required. If you're the least bit interested in actual compression programs, this would be a good place to get started.

Writing the code itself should be done by writing the tree. Your goal is to write a linear representation of the tree as a single line of text so that when decoding occurs you can first reconstruct the tree through a linear scan of that line of text. Given that our messages contain only alphabetic characters, we can use comma separated values to write our tree and differentiate leaf nodes from non-leaf nodes by writing a character like * for non-leaf nodes. The end result is a single line of * and letters separated by commas. We'll discuss algorithms for doing this with our Huffman trees in class, but it ultimately boils down to choosing the right traversal pattern and working it through some recursive I/O.

All told, we have the file format for encoded/compressed messages shown in figure 2. This file consists of two lines, the first is the tree as comma separated characters and the second is the binary string that represents the compressed message.

```
COMMASEPARTEDCHARACTERS
BINARYSTRING
```

Figure 2: Encoded/Compressed Message format

Logistics

The project is **due no later than Tuesday 3/22**. Submit the source code via *handin* as assignment *proj1* and submit a UML diagram for your code. The UML will be collected in class on Wednesday 3/22. Grades are based on the completeness and correctness of the program, the documentation, the tests, and the overall quality of the design. Tests should minimally cover the main public facing interface of the program. In the event of an incomplete project, submissions that are more thoroughly tested with jUnit tests can expect to receive better grades than those with minimal testing.