Bryan Settles

ECE 368 Project 2

I started off this project by reviewing my previous Huffman coding project that I coded in ECE 264. That project consisted of a variety of steps, to ensure that students encoded the text file with the correct tree and list structure as intended, and was our first big introduction to some of the applications of binary trees. It also allowed us to see one use of ASCII values. At first, I tried to make changes to this code to fit our current project, and got the Huffman part working. However, it was difficult to program the unhuff part, and I decided to reorganize my code and start from the beginning in order to get a better understanding of the program and how it worked.

The first simple step in the compression process was to receive the input file and to create a function to calculate the frequency of the different characters. After that, I sought to create a binary tree, with the location of the letters based on the number of occurrences of that particular character. Once that was created, I planned to generate binary code values by traversing the tree and determining the position of each of the different characters, with a move to the left counting as a ‘0’ and a move to the right counting as a ‘1’. However, when planning and researching other methods of encoding Huffman trees, I decided on a different method.

In my program, to create the tree, I joined two lowest frequency tree nodes together, adding their frequencies to make their parent node. I repeated this process for all of the different characters and frequencies until the tree was complete. After this, my goal was to

create a binary sequence to assign to each character, giving the characters with the highest frequencies the shortest binary sequences possible. I created a number of helper functions in order to organize the process and to prevent any confusion or human error on my part. The first thing written to the '.huff' file is the header. After that, the text file is then traversed and letters are stored according to the binary sequences that they were previously assigned using the binary tree. At the end of this process, the filename is then changed to include the ".huff" that was required to come at the end of it.

At this point I felt comfortable starting to plan and write code for the decompression process, which I started in a similar fashion to the compression process. I received the input file, immediately changed the filename to include the ".unhuff" to come after the ".huff" that was added earlier, and began decoding the message by reading the header, which then created the corresponding tree nodes and list nodes. I created several helper functions to assist in this process as well. When the newline character, '\n', is encountered, the header has been fully read, and the reconstruction of the binary tree begins. While the file is read, the binary tree is referenced so the output file can be correct, as well as to make sure that the output is put in correct order. It goes until the end of the file is reached.

All in all, there are still many optimizations that need to be made in order to improve on the compression ratio in this project. No excess flags were used to compile this program. As stated in the instructions, there is only 1 argument to the program, the name of the file to be encoded and decoded. "./huff" is used to compress the file, and "./unhuff" is used to decompress the file.