ECE 368 Project 2 – Ethan Glaser

After much trial and error, I have successfully created Huffman-based compression and decompression c files. The compression is done by using a linked list and binary tree to encode each character present in the file and then prints the pre-order traversal of the tree in the header the encoding to the .huff file as it reads through the input file. The decompression does the opposite, recreating the binary tree from the header and then using the tree to decode the .huff file into the .unhuff file, which matches the original text file input.

The first step that occurs in the huff.c file is to create a linked list based on the content of the input file. The text is read and added to the linked list, which consists of each character in the input as well as the frequency that the character occurs in the file. Once a linked list has been created with every character in the file, the list is converted into a binary tree. The process for this conversion involves combining the two nodes in the list with the lowest frequency, merging their binary trees and updating the frequency to be the sum of frequencies of the two nodes combined. This process continues until a single node remains. This node points to the complete binary tree for the input file, with the high frequency characters closest to the top of the tree. The next step involved creating a 2D array by iterating through the tree to determine the height and value associated with the binary path to each character present in the tree. Once this array is complete, the first modification of the compressed file begins, with the printing of the header information. The header information consists of the pre-order traversal of the binary tree, with 0s to represent non-leaf nodes and 1 followed by the character for each leaf node (ex: 01a1b). Each 1 and 0 is represented as bits by or-ing in the corresponding value to the next available bit in a temporary byte. Once the byte is filled or a character is detected, the temporary byte is output to the .huff file. Therefore, the header is relatively compressed, although the characters must be represented by their full ASCII value. The input file is then scanned again, and for each character read, the corresponding bit encoding is appended to the compressed file using the 2D array created previously. The bits are appended similarly to the process for the header encoding, although each byte is filled entirely before being written to the compressed file. An EOF character was included in the linked list to signify the end of the file, and once this value was written to the output file, the loop was exited and the compressed file is complete.

The decompression process begins by reading in the header. This occurs one byte at a time. For each bit read, a function is called that allocates the next node based on the existing tree. The value of the next node is then based on the read bit, 0 if the bit is 0, and if the bit is 1 it reads in the next byte

which contains the ASCII value of the character present in the node. The function to determine the location of the next node was one of the most difficult parts of the project. This process occurs until the function returns null, implying that every leaf node has a character associated with it and the tree is complete. Then the remainder of the compressed file is read, bit-by-bit. Starting at the head, a 0 directs the next node to the left and a 1 to the right. Once a leaf node is hit, that character is appended to the decompressed file and then the tree returns to the head and the process restarts until the compressed file is completely scanned and the EOF character is read.

In addition to the two compression and decompression executables, I created a third executable to accumulate testing data. The executable, called run.c includes many lines of bash to test the time that both huff and unhuff take to run on each of the 6 sample test files, as well as output whether the .huff.unhuff file is an exact replication of the original file and compares the size of the compressed file to the original. This made general evaluation of the entire process very easy as a single command produces output run times, compression ratios, and whether the compression and decompression were successful for each sample text file. The data for each is shown below.

File name	test0.txt	test1.txt	test2.txt	test3.txt	test4.txt	test5.txt
Huff (sec)	0.001	0.000	0.000	0.293	0.586	0.878
Unhuff (sec)	0.001	0.000	0.001	0.198	0.397	0.575
Comp. ratio	1.75	2.75	0.92258	0.73454	0.73449	0.73451

These results suggest that the compression worked effectively, as the larger files were compressed to approximately 73% of their original size, a ratio comparable to most of the students in the leaderboard in the google sheets, and the times were comparable as well, all under 1 second. The compression ratios on the smaller files were greater than 1, which is expected based on the nature of the compression and header information. I also tested the compression on song lyrics and other text files of various sizes, as well as a 125000 line file of Shakespeare's work, which had a compression ratio of 0.57 and times under 0.5 seconds for huff and unhuff.

Overall, my Huffman compression and decompression work successfully and produce files with reduced size after compression in under 1 second, even on very large text files. This project was very interesting because it gave insight on how file compression works. I have used zipping on files before but never quite understood what happens, but now I have a better understanding.