Matthew Wen

In this 368 project, I am looking into the compression of huffman. The idea behind huffman is to compress by using a binary tree that would store the direction of a character inside a binary tree; if a leaf is it, it would start right back to the top of the tree to get the next character. The process of setting the location of the tree is determined by the number of occurrences of the character inside the text file, therefore, the program needs to find an easy way to read the complex file, count the number of occurrences for each, and then determine its location for each before going through the text file again to determine how each character in the text file should be stored. On the other hand, the decompressed program must be efficient enough to take the compressed huffman file, understand the header by taking in the table information, and then decompress each character to get the output.

Based on my understanding of the topic, I will underline the basic structure of my huffman code by talking about the algorithm used to create the binary tree. The code starts off with a complex data type which stores the size of the header, the size of the compressed file, the size of the uncompressed file, and then a 256 array. When the advanced data type is initialized, it will only store the size of the uncompressed file, because the size of the header and size of the compressed file is unknown. After the advanced data type is malloc into memory, it will start reading the uncompressed file by bytes. Because of the assumption that the file being read is a .txt file, it is safe to assume the data type would be in ascii format with character values ranging from 0 to 255. Now, let's say for example that the huffman code recognizes the character 'a'. it would increment the index at its ascii equivalent, or integer value, which is 97. It would continue this process to create a histogram which stores the occurrences of each character. Afterwords, it would create the binary tree by looking at the smallest elements inside the array. The reason why the array stores an advanced data type rather than an integer is because when getting the smallest value, value being the occurrence, it is not necessary to sort the array, but it is necessary to reduce the number of comparisons to get the smallest. Therefore, if a character, for instance character 'b', is used, then the index where character b is located at would switch places with the buffer end of the stack being the array which stores the occurrences, which therefore limits the amount of comparisons required as more elements would be added to the binary tree; the process of this will still be O(n^2), with n being 256. But because the stack size is changing, it would have a C value lower than one when modeling it based off of C * g(n), with g(n) being n^2. Afterwards, the program will use binary operations to store the direction of the characters. I am planning to store them into ints or longs, which can store up to 64 or 128 directions inside the binary tree if it is on a 64 bit machine. I would write to the file every time the 64 or 128 bytes are all used up, keeping the space complexity of the compression to O(1).

For decompressing it, I've been trying to understand how the header will store the table information. The table information includes the occurrence of each character to it knows. Based off the structure of the code, it would be best to store the occurrence with the character, instead of storing 256 integers inside a file, each specified to being unsigned int that consumes 4 bytes based off the inttypes library. This therefore will only require to append less than 256 different characters inside the binary tree instead of 256. Also, it might be best to sort the the character used in pre-order or post-order form so the algorithm will be quicker at storing the elements. Afterwords, it would decompress the file.