Design and Implementation:

Building Code book

Our implementation of the fileCompressor is a pretty straight forward approach. For building the codebook, we create a struct variable that is in charge of storing all tokens, frequency of tokens, and child nodes that will be used in building the Huffman tree. The first step is scanning through the file, tokenizing it, then storing the tokens in the struct array. If a token already exists, we just add to the frequency of the specific token. Once the array of structs is made, we sort it using insertion sort. This could have been faster, and I wish I had opted for a minheap, but it is still plenty fast enough for building the codebook. Next, we build the Huffman tree with the array, and each time a node is added back in we use insertion sort to keep the array in order. This ensured the Huffman tree built is valid. Lastly is traversing the made tree, recording 0 if we travel to a left child, and 1 if we travel to a right node. Altogether, accomplishing everything here will take O(n^2) runtime.

Compress and Decompress

Implementing the compression and decompression was much simpler then building the codebook. For compression, what we do is open and write all the tokens and bit sequences from the HuffmanCodeBook into an array of structs. Each struct having its own individual token and bitSequence. Then we go into the file to be compressed, compress all the tokens using the same tokenizer that builds the HuffmanCodeBook, and compare each token with the tokens from the HuffmanCodeBook. This process runs in O(n*m) runtime, where n is the size of the huffmancodebook struct (which will just be the number of tokens in the HuffmanCodeBook), and m is the side of the array made by the file. We also make sure that all tokens from a file are within the code book. If there is ever a case where there is a token in a file that is not in the code book, we stop compressing and delete anything up to that point. That file will no longer be compressed/decompressed unless the codebook is rebuilt.

For Decompressing, the process is exactly the same as compressing. We create the array of structs from the code book (once again this will always be the number of tokens in the HuffmanCodeBook), and traverse through the compressed file. This process is again a O(n*m) runtime, where n is the size of the array of structs, and m is the number of chars in the file.

Misc.

Space Usage:

We have the array of structs that hold the tokens and frequencies when building the codebook, an array of chars for each file, and another array of structs for the HuffmanCodeBook tokens and bit sequences.

Other:

There are some functions that are used multiple times or every time, for building the codebook, compressing and decompressing. I think one of the smartest ideas we had was that the first function that always gets called is a function that will pass the correct file to all the other functions used in compress, decompress, or building the code book. If the call made is not recursive, then it will just pass the file along. If the call made includes the ‘R’ flag, it will recursively ascend through the directories, adding on the path as it goes, and send the files with

the paths all the other functions. This simplified the implementation quite a bit, and after this was made we only had to focus on doing the job (compressing/decompressing/building the codebook) for a single file.