“readme.pdf” that describes the design and implementation of your fileCompressor
- include an analysis of the time and space usage and complexity of your program. You do not need to analyze every size function, but you should analyze the overall program's time and space used as a function of the size of the input.

Design/Implementation:

Adam and I designed our algorithm based off of an AVL implemented Huffman Coding algorithm. We first tokenize the input file, then convert it to an AVL to count occurrences. Then, we convert to a Minheap for the Huffman Coding. Once in the minheap, I chose to implement a bnode (a binary tree node) as a part of the minheap, allowing us to use the minheap for both sorts. This allowed us to create the right structure to build our codebook. For compressing and decompressing, we used an array of binary nodes (bnodes) to store the tokens and encodings from the HuffmanCodebook file. After that, we searched through them and wrote() to the correct files, whether for encoding or decoding (compressing or decompressing).

Time/Space Usage:

AVL: n∗log(n) total time complexity for inserting, sorting and translating over to minheap.
Minheap: n∗log(n) total time complexity with inserting, deleting, sifting up and down
Binary Tree: n ∗ n = n^2 for the compressing and decompressing because the binary nodes are put into an array of binary nodes. Having a O(n^2) is not ideal, but it was necessary to fulfill our implementation in a timely manner, and with our other algorithms at optimal speeds, it works well.

Complexity:

The algorithm is long, as there are many moving parts, however it is modularized and very easy to read. There are three files that contain information, the filecompressor.c , avltree.c and the minheap.c, the avltree.c and the minheap.c hold all our data structures while the filecompressor.c holds some tokenizer functions and the main driver.

All in all, it was difficult but manageable and I believe we did a very good job!