The file compressor can operate given a direct file path, or a directory path (as long as given a -R flag along with it). When given the arguments on the command line, the user must give one of these flags: [-b, -c, -d], and an optional -R flag for recursively operating through directories and their files. Then the following argument is the file or directory path, in which we use the stat syscall to check whether we are given a valid path or not, and whether or not it is a regular file or directory. If it is a directory, we use openDir to open the directory and readDir to see the contents inside of it. We then check for each nested file types to decide whether to recursively traverse into nested directories, or use open(), read(), and close() syscalls to open files and perform one of the following three methods on them: build, compress, or decode. For flags [-c, -d], they require an additional command line argument for the path of the generated huffman codebook.

Filecompressor first goes through all the files needed to build the codebook, and tokenizes all of the strings to create a complete linked list of every token in the searched files and the amount of times those tokens appear. Those tokens are then given to the genBook method which creates a tree by looping through twice to find the two smallest frequencies out of the remaining nodes, then combining the two nodes into one, until there is only one node left, which it returns. Because these nodes are not sorted at any point there is a minor hit in run time, however there is an improvement in space and speed efficiency in that the treeNodes themselves act as a linked list, removing the need to convert them from a linked list into an additional array of treeNodes. Once the tree is created, the publish method recursively goes down every branch and keeps track of the path it took, then prints out the path and token whenever it reaches an existing token.

The compress method opens the file to compress and reads through char by char. It tokenizes the strings in a similar way to the string tokenizer before. Every token is then sent to the search method which finds the token in the tree created before. The path taken to find the token is recorded and written in the output file. The overall string is only looped through once, and as per design, it is faster to search through the tree for tokens that appear frequently, minimizing runtime. Besides the strings written and the tree designed from the given codebook, no other space is needed

Decode works in a similar manner, where the file is read char by char, however this time each char progresses down the tree depending on if the char read is a 1 or a 0. Whenever the pointer reaches a treeNode that has a token in it, the token is written in the output file and the pointer is reset to the top of the tree. Because the method loops through the same way and also progresses through the same tree, it has a similar efficiency to compress. The method also has similar space efficiency.