Aditi Singh and Rumeet Goradia (as2811 and rug5)
Asst2 File Compression

In this assignment we designed a program that builds a Huffman tree that encrypts tokens from specified files. The resulting Huffman Codebook is used to condense such files into Huffman-encoded .hcz files and then later to decompress the .hcz files back into the original files. These operations are done either non-recursively or recursively over multiple files within directories and subdirectories.

fileCompressor.c is the main file. We decided that the best way to go about reading in the file's content was to store it all in a single string. The program then utilizes the *tokenize()* function that inserts tokens into a linked list in the case that the token is unique, or updates its frequency by searching through the linked list, if the token is repeated. For the build flag, a new token is created every time a delimiter is read, which will always happen since all files with any content end with a new line, while a guard is put into place in *main()* for empty files. In addition, each delimiting whitespace is inputted into the linked list using a unique string so as to not mess up the formatting of the HuffmanCodebook file. The other two flags utilize codes and tokens arrays that are populated using a similar tokenizing process on the HuffmanCodebook file provided. Tokenizing has a runtime efficiency of O(n), where n is the number of individual tokens in the given file, including whitespace. Based on the flag given, a specific function, whose definition is located in one of the library files, is called. Before any operation or tokenizing occurs, however, the presence of a recursion flag is checked. If it was given by the user, then the program checks if the path specified leads to a file or a directory. A file-leading path undergoes the same operation as if the recursion flag was not present. In the other case, the individual operation that would occur without a recursion flag would be applied to every subdirectory and file located within the given directory through the function *recursive_function()*.

Within the file fcdatastructs.c, we constructed a min-heap that correctly sorts the given tokens by their respective frequencies into a Huffman tree. The Huffman tree then assigns a bit sequence that is specific to each unique token as a code value. The worst case scenario for the min-heap is if all the data is inputted from greatest frequency to least frequency. In this case, the min-heap would have to compare the node that has to be inserted to the value of every node that is already present within the tree. As a consequence of this, since there are n nodes and it would take O(logn) to traverse the min-heap and insert, the overall time complexity would be O(nlogn) for the min-heap.

From this, the file tokenizer.c is then in charge of compressing and decompressing the file. *compress()* tokenizes the input of the file using the previously mentioned method. For each

token created, the corresponding code is written to the new .hcz file. This function has a runtime efficiency of O(n), where n is the number of tokens. Since white-spaces are fairly common within files, the function obtains the code for the three types of whitespace before entering the main loop so that the loop would not have to perform the same search every time an instance of whitespace is encountered. After this, the function *decompress()* is in charge of reading the characters from the input and if the input corresponds/matches to a code, then the token that corresponds to that code is written to the specified file. This method also has a runtime efficiency of O(n), where n is the number of characters in the given file.

The program has comprehensive error checking that checks all different types of errors, including, but not limited to, errors in the command line input and errors related to opening files and directories. The overall efficiency of the program is O(nlogn), as building the HuffmanCodebook file with the min-heap has the largest big-O of O(nlogn). Although *decompress()* has to analyze the input character by character, it does not take as long as building the codebook since the compressed files are most often shorter than the decompressed files.