Design & Implementation:

This program takes in one or two flags, a path or file, and a codebook (depending on the flags). The program has three main functions: build codebook, compress, and decompress. The build function (-b flag) will create a single Huffman Codebook containing the data from the given path or file and their respective Huffman codes. The compress function (-c flag) takes in a path or file along with a codebook and compresses the given data based on their Huffman codes and outputs files of the same name but with a ".hcz" extension. The files are created alongside the original file. The compress function only compresses files without a ".hcz" extension. The decompress function (-d flag) takes in a path or file along with a codebook and decompresses the given data based on their respective strings and outputs files of the same but without the ".hcz" extension. The files are created alongside the original file. The decompress function only decompresses ".hcz" files. When entering a directory, recursive mode (-R flag) must be set before one of the function flags in order to recursively go through the directory and any subdirectories and files.

The program first checks to see if the input is entered correctly. If entering a file, only one of the function flags (-b, -c, -d) can be used. Multiple function flags can not be given or this will result in an error. In addition, recursive mode can not be used with files. This will cause the program to stop running and print an error. If entering a directory path, recursive mode (-R) must be set before entering a function flag. If recursive mode is not set when a directory path is given, this will result in an error. Only one function flag can be given after setting recursive mode. If the given file or directory does not exist, an error message will be printed.

If the build flag (-b) is given, the file given or all of the files in the directory given will be used to create a Huffman Codebook. First, the data in the file is read and broken up into tokens. Tokens are separated by spaces, new lines, and tabs. These delimiters are also considered tokens. As a token is created, it is inserted into an AVL tree. Each node is given a balance factor and a height so that the program is able to check whether or not the tree is balanced and balance it if need be. Each token is also searched for in the tree. If a token already exists in the tree, its frequency will be incremented by 1 and it will not be added again to the tree. If the token is a space, new line, or tab, a special control code is used to store into the tree. "_\\", "_\\n", and "_\\t" are stored to represent spaces, new lines, and tabs respectively. This is done so that a space, new line, or tab is not actually printed. Once all of the data in the file is tokenized and inserted into the tree, an array of pointers is created. Each block in the array points to a node in the tree. After, the array is turned into a min heap by using the sift down method. Each node's frequency value is less than that of its children. The smallest frequency node will be at the top of the heap (start of the array). Once this is done, a Huffman tree is created from the array. The function takes the two smallest frequency nodes, adds up their frequencies, and creates a new node with that frequency. This new node serves as the head of the two smaller nodes and is inserted back into the array. The sift down algorithm is run again so that the smallest frequency is at the top of the heap and this

process repeats until one Huffman tree is created. Once the tree is created, each node is given a code. This is done by traversing the tree and giving a “0” for the left branch and a “1” for the right branch. After, each node has a Huffman code. This tree is then used to create a Huffman Codebook. By traversing through the tree, each code and token string is written into a new file named “HuffmanCodebook”.

If the compress flag (-c) is given, the file given or all of the files in the directory given will be used to compress each of the files using the codebook. This function only works on files that do not have a “.hcz” extension. Any files with that extension are ignored. First, all of the tokens and their respective codes from the codebook are inserted into an AVL tree. Once this is completed, the data in the file is read and is tokenized by spaces, new lines, and tabs. Each token is then searched for in the AVL tree and the token’s code is retrieved. This code is then written into a new file with the same file name but with a “.hcz” extension. The token’s code essentially replaces the token in the file. Once all of the files' data is read and the codes are written in the new file, the program exits.

If the decompress flag (-d) is given, the file given or all of the files in the directory given will be used to decompress each of the files using the codebook. This function only works on files that have a “.hcz” extension. Any files without that extension are ignored. First, all of the tokens and their respective codes from the codebook are inserted into an AVL tree. Once this is completed, the data in the files is read. Since the files contain the codes to each of the tokens in the codebook, each character is read one at a time and is compared to each of the codes in the AVL tree. All of the token’s Huffman codes are unique, so the possible problem of having two tokens with the same code will not occur. If the code is not found in the tree, then the next character in the file is appended onto the previous character(s) and this new string is compared with the codes in the tree. If the code is found in the tree, its string is then written into a new file with the same file name but without the “.hcz” extension. The token’s name essentially gets replaced by its code. Once all of the files' data is read and the tokens are written in the new file, the program exits.

Complexity of Program:

An AVL tree is the main data structure that is used in the program to handle all of the files’ data. It is used to store and search for all of the tokens for the build function and it is used to store the codebook’s tokens and their respective codes for the compress and decompress functions. In terms of space usage, the average case and worst case space complexity for an AVL tree is O(n), n being the total number of tokens in the tree. For the build function, the total number of tokens comes from each unique token in the files. For the compress and decompress functions, the total number of tokens comes from each listed token in the codebook. Additionally, the build function creates space for an array of pointers. The space complexity of this is also O(n) since each of the

nodes in the AVL are stored in the array. The worst case time complexity for an AVL tree is O(log(n)) for both insert and search. This is due to the fact that the tree is height balanced, meaning that each node has a left branch height and a right branch height that only differs by at most 1. This causes the insert and the search to be faster than if you are to use a binary search tree for example. With a binary search tree, the worst case insert and search time is O(n) due to the fact that the tree could be completely skewed to either the left or right. This causes insert and search to take longer than it would with an AVL tree.