Readme.txt

1. File Structure
    a. FileCompressor.c parses arguments, handles errors, and calls the appropriate method which is to build a codebook, compress, or decompress.
    b. BuildCodebook.c handles all of the code related to building the Huffman codebook.
    c. Compress.c handles all of the code related to using the Huffman codebook to compress a text file into a file with only 0's and 1's.
    d. Decompress.c handles all of the code related to using the Huffman codebook to decompress a file with only 0's and 1's into a text file.
2. BuildCodebook.c
    a. Data Structures
        i. Node
            1. The AVL tree, min heap, and Huffman Tree each used the same node data structure.
            2. Each node has char * string containing a unique token, an int frequency containing the amount of times that the token occurs, a struct Node * left containing the left child of the node, a struct Node * right containing the right child of the node, and an int height containing the height of the node in the tree that it is in.
        ii. AVL tree
            1. The program inserts tokens from each file that it needs to build the codebook for into an AVL tree.
            2. Time: Inserting a token into an AVLTree has a time complexity of O(log n).
            3. Space: The size of the AVL tree is the number of unique tokens in all of the files that the program needs to build a codebook for.
        iii. Min heap
            1. The program builds the Huffman Tree by using a min heap data structure.

2. Time: Inserting to and deleting from a min heap takes O(log n) time. Building a min heap takes O(n) time.
3. Space: The size of the min heap is the number of unique tokens in all of the files that the program needs to build a codebook for.

iv. Huffman Tree

1. The program uses a Huffman Tree to generate codes for each unique token, and places these codes in the codebook
2. Time: Generating the code for each token using the Huffman tree takes on average O(log n) time. However, in the worst case, generating the code takes O(n) because the program cannot guarantee that the structure of the tree is like an AVL tree.
3. Space: The size of the Huffman Tree is 2*the number of unique tokens.

b. Algorithm

i. First, the program determines whether it received a file or a directory.
ii. Next, the program will travel through the file or recursively travel through the files in the directory to insert unique tokens from the files into the AVL tree. O(number of characters in the file/ in all of the files of the directory) time to find unique tokens
iii. After, the program will convert the tree into a min heap. O(number of unique tokens) time
iv. Next, the program uses the min heap to build the Huffman tree.
v. After, for each leaf of the Huffman tree, which by default has a unique token from a file, the program will generate the code for it.
vi. Then, the program will append this code and the token that corresponds to it to a string that contains all of the information in the codebook. O(number of unique tokens) time
vii. Finally, the program will write that string to the codebook.

3. Compress.c

a. Data Structures

i. Linked List

1. Each element of the hash table has a linked list.

ii. Hash Table

1. The program inserts tokens and their corresponding bitcodes from the codebook into an array of linked lists, which we call a hash table.
2. Time: Inserting and searching for tokens in the hash table take O(1) time on average. However, in the worst case, these operations will take O(n) time if all of the tokens happen to belong in the same index of the hash table. We designed the hash function to be key % size where key=key*13+(int)word[i] such that word[i] is the letter at the ith position of a token, and size is the number of tokens in the codebook multiplied by 1.3.
3. Space: The hash table has a size of the number of tokens in the codebook, multiplied by 1.3. In the best case situation, the space that the hash table takes up is the number of tokens in the codebook multiplied by 1.3. In the worst case situation, the space that the hash table takes up is that, plus the number of tokens in the codebook, and this happens when all of the tokens are stored at the same index of the hash table.

b. Algorithm

i. For each token in the codebook, insert it into the hash table in O(1) time. The total amount of time that inserting into the hash table takes is O(number of tokens in the codebook) time. The total amount of time to find tokens in the codebook is O(number of characters in the codebook) because the program needs to read every character of the codebook.

ii. Check if the user inputted a directory or a file. If the user inputted a directory, recursively travel through it and compress each file.

iii. To compress a file: O(number of characters in the file) time to read each character

1. For each token in the file, search for it in the hash table, and find its corresponding bitcode.

2. Append that bitcode to a string.
3. After reading each token in the file and appending bitcodes corresponding to them to the string, write the string to a new file with a .hcz extension.

4. Decompress.c
    a. Data Structures
        i. Array
            1. The index of the array represents the value of the bitcode in decimal notation.
            2. The value stored at an index of the array will be the token that corresponds to the index or null, if no token corresponds to that index.
            3. We do not use an array of linked lists as in compress, because we know that none or only one token will correspond to each bitcode value, not more than one.
            4. Time: Inserting and searching a token in the array will take O(1) time because the program is manipulating or accessing a value at one index that is given.
            5. Space: The array has a size of $2^{\text{max length of the bit code string}}$ because that is the maximum value in decimal notation that a bit code string in the codebook can have.
    b. Algorithm
        i. For each token in the codebook, insert it into the array at the appropriate position.
            1. Finding the appropriate position will take O(length of bitcode string) time because the program needs to read every character in the bitcode string to convert the number into decimal notation.
            2. Inserting the token into the array will take O(1) time.
            3. The total amount of time that this operation takes, for all tokens, is O(number of tokens in the codebook*max length of bitcode string) time.

ii. Check if the user inputted a directory or a file. If the user inputted a directory, recursively travel through it and compress each file.
iii. To decompress a file: O(number of characters in the file) time to read each character
    1. Read one character and append it to a bit string.
    2. If the program, after converting the bit string into decimal notation, finds a token that corresponds to it in the array, then append that token to a string.
    3. Otherwise, if the program, after converting the bit string into decimal notation, does not find a token that corresponds to it in the array, then go back to step one.
    4. After reading all of the 0's and 1's in the file and appending tokens corresponding to them to the string, write the string to a new file without a .hcz extension.

5. fileCompressor.c
    a. Data Structures
        i. No data structures were used, and no while loops were used that would run more than five times. So, time is O(1).
    b. Algorithm
        i. First, the code checks if it was given valid inputs by checking to see if the flags given are used correctly.
        ii. Then, it takes the data in the file position and the codebook position, and passes it on to runProcesses.
        iii. This function first checks if the recursive flag was used when the input is a directory, and if it wasn't used, it returns an error.
        iv. Then it does a different operation depending on which flag was used, but for each one it first checks if it is a valid file or directory.
        v. If the -b flag was used, assuming it is a valid file or directory, it runs buildCodebook for the given input.

vi. If the -c flag was used, it initializes a hash table using the given codebook. It then performs recursive compression if it is a directory, or compression assuming it is not a .hcz file.

vii. If the -d flag was used, it initializes a hash array using the given codebook. It then performs recursive decompression if it is a directory, or decompression if it is a .hcz file.

6. Possible errors:
    a. Wrong number of arguments
    b. Invalid flags provided (-b, c, d R)
    c. No path provided with recursive argument
    d. Invalid Codebook name
    e. .hcz file for build codebook or compress
    f. Non-.hcz file for decompress
    g. Invalid path or file name provided for operation
    h. Not able to read file for build codebook or compress
    i. Not able to open file to write .hcz file
    j. Could not for bitCode in code book for compressing