Read Me:
Overall Algorithm:
Create a hashtable to store frequencies and tokens throughout all files. This is needed to build the codebook. Check what flags are called and how many there are. Return errors if the flags are invalid input ex. -R -b -d <file> Codebook
If input is proper, perform what the flags say. If there is a -R, then call the recursive function that walks through the directory. When a file is reached, call the method for the appropriate flag. (Call compressor if given -c decompressor for -d, build for -b).

Recursion:
Takes in flags as an int representation (1=b, 2=c, 3=d). Opens the directory path if it was a valid path to a directory. The program traverses through each file/folder in the directory and that level and concatnates the directory name to a string called path. There is a while loop inside the method, that searches thorugh the directory given (while readdir). When a directory is found, a recursive call is made by sending in that directory. When a file is reached, the path that is created for that file is passed in as a parameter to the given flag's function.

Build Code Book:
Structures Used:
Hashtables - Used to store tokens and update their frequency while files are being parsed

Linked List - Struct that contains a struct HeapNode* tree and a *next. Used to sort the minheap trees.

Min Heap - Struct HeapNode* has fields: frequency, name, height, *left child, *right child. Used to create a min heap of the tokens and their frequencies.

Huffman Tree - Struct HeapNode*, a tree that is organized by frequency. Each parent node is given a unique name, based on the order of insertion into the Huffman Tree, in order to make searching for the correct tree possible while building the tree.

Algorithm:
1. Store tokens and frequencies
Parse the file using getNextToken and insert each token into a hashtable

2. Build Minheap Array

3. Build Huffman Tree

4. Build Codebook
Traverse through the huffman tree, using an int array to store all of the directions taken thus far (in 1/0s), when a leaf node is reached, the full contents of the array is written to the codebook file, plus a tab, the token, and then a newline. Because recursion is used, each time the function returns to a previous call, it also returns to a prior index in the int array, so that the index of the values written in to the array correspond to the level of the tree that is being traversed.

```
Compress:
        Takes a file to compress and codebook. Check if the file is a regular file, if it
is then copy the filepath and concatenate ".hcz." Now get each token one by one from the
file and send it into a method called retcode that will parse the codebook and check if
the target string is in the codebook. If it is not it will return "3" for error and
compress will remove the .hcz file created. If it is in the codebook then the huffman code
is returned and compress will write this code in to the compressed file.

Decompress:
        Takes in the file to decompress the codebook and a struct pointer to the huffman
tree. It builds the huffman tree if it does not already exist. It reads the codebook token
by token to create the Huffman tree, following the binary key in order to build each path.
Once the tree is created, it parses through the file to decompress character by character.
If the char is a 0, the left branch of the Huffman tree is followed, if 1 go right, else
return error. If the .hcz file leads our HUffman tree pointer past the end of a branch of
the tree, an error message is displayed to stdout.

Global Var:
        HashTable[10000] - Stores the tokens and their frequencies throughout all files.
Useful as a global rather than passing it into functions as a pointer because almost every
function in the build method needed it. Passing via pointers led to errors when trying to
access the table as it was being passed into several functions that were inside of each
other.
                ie: create table in main, pass into build() as a pointer, build passes it
to hashToArr() to create a minheap array, which passes it into its helper functions etc.

        numToks - Stores the number of tokens inserted into the hash table throughout all
files, so that we know how big to make our min heap array.
                This is an int pointer that was being passed to many different functions
all within each other as well. It caused a lot of errors this way, so it is simpler to
make it into a global variable.

Time Complexity: (t is the number of tokens in the file mentioned)
        Build Codebook: O(t log t)
                        Parsing the file:O(t)
                        Storing it to hashtable: Best Case: O(1), Worst Case: O(t)
                        Heapify: Heapification is log t, and t such calls are made: t log
t
                        Building Huffman tree: O(t log t)
                                        - log t to find the smallest two values, insert
them into the tree, performed on t tokens
                        Writing codebook from Huffman tree: O(t)

        Compress:O(t)
                        Parsing the file:O(t)
                        Reading through decompressed codebook to find token: Best Case:
O(1), Worst Case: O(t)
                        Writing compressed file: O(t)
        Decompress:O(t log t)
                        Recreating the Huffman Tree:O(t log t)
                                -Have to find each path for each node, traverse down that
path to insert the new node
                        Parsing the compressed file:O(t)
                        Searching the tree: t*
```

```
                        Writing decompressed file: O(t)
        Recurse:
                d - # of directories(dt_reg + dt_dir)
                        while loop: O(d)
                                - goes through all of the directories(dt_reg + dt_dir) in
a folder
                        recursive call
                        if dt_reg
                                O() of the function that is being called (O(compress),
O(decompress) O(buildCodebook))


Space Complexity: (t is the number of tokens in the file entered)
        Hashtable: Static array of size 10000, with t many nodes hanging off of it.
        Heap: Array of size t.
        Linked list: Used to build Huffman tree, composed of at most log t nodes, each
with a tree inside. There are t many tree nodes overall.
        Huffman tree: Composed of t many nodes
```