Design: Our program uses a recursive function to open up directories to reach the final file / directory that we are going for. Once it reaches there, it takes the file / files and reads a buffer string and calls upon tokenizer to seperate real words and inserting them into a static sized hashtable of size 26, with a hashkey of its respective asciivalue (being the letter that it started with), and inserting another node off of that node to be the file name with its counter (if it did not occur in the linked list). We then respectively sort the filenames with a priority of weight, then an alphanumeric sorting pattern of a-z, 0-9, and a '.'. Finally we print out all the nodes from the static hashtable with the respective nodes that each word node has stored in its specific pointer. More details on everything below.

STRUCTURES:
wordNode: WordNode is a node struct that is used for direct insertion to our static hashtable and contains the token string, a next pointer for its next wordNode in the hashtable (linked list), and a string_LL pointer called FILE_LL that points to a seperate linked list that contains the string_LL struct.

stringLL: StringLL is a string_LL struct that is used as a file name and counter data holder that is linked off of the original wordNode node and contains an integer for a counter of how many times the word has appeared in a file, a filename string, and a next pointer for its next stringLL node in the linked list.

MAIN PROGRAM METHODS:
insert: Insert takes the token string, the filename string and takes the token string and inserts it sorted (alphabetically) into the hashtable. If the token string exists, then we search its linked list for the file name and if thats found, we update the counter. If the filename is not found, we wait to sort it and basically attach it at the head of the linked list for easier searches for that word in that specific file. If the token string does not exist, we insert it alphabetically into our hashtable, and update it with a new stringLL struct for the current file its traversing through.

SetNode: adds current stringLL node into hash table. If the key already has a value other than null, it updates the counter

sortNode: sorts files where each word is located in by frequency (descending order)

freeNodes: frees current stringLL nodes being used by application

freeAll: goes through each word, and calls freeNodes for each of the lists before freeing the wordNode

filesorter: Because our stringLL nodes keep getting updated until the program finishes indexing files, we need to apply a final filesorter method at the end to sort these filenames in an arrangement of a-z, 0-9, and '.'. It calls upon a self made stringcompare method to base a-z, 0-9, ad '.'. Filesorter itself manages to sort based on the counter integer.

indexDirectory: IndexDirectory calls upon the string of the directory of itself, and uses the DIR* to access the pointers located inside the directory, and struct dir to access the names, and the file types. If its another directory, we modify the original string, go furthur down and open that directory in a recursive method. If its a file, we index the file.

printmethod: Final method call in our main method, prints out our hashtable if the key is not NULL,

and for every wordNode, we print out its string_LL FILE_LL linked list, in a XML format.

indexFile: takes a string, and a file name string, and it uses a hard coded substring method to cut off a string that fits

main: main function checks if there are enough arguments given, as well as whether the second argument given is either a singular file or a directory containing other potential files and directories

HELPER METHODS:
stringcompare: A recursive method to return an integer based on which of the two given strings are greater. It returns a negative to help insert find a node that is 'lesser' than the node we are trying to insert so, prev will point to the node we insert, and the node we insert will point to ptr.

isChar: Tells if a char is a character based on the ascii table.

isNum: Tells if a char is a number based on the ascii table.

Due to it being a hash table implementation, runtimes for inserting and sorting will at most go for O(n) time for each word, though a lot of memory is used up to hold all tokens.