Structs

There are three different structs which were used in this program:

```
HashNode
        struct HashNode* next
        char* word
        fileNode* fileList

FileNode
        struct FileNode* next
        int* count
        char* filename

node
        node* next
        char* word
```

HashNodes are used in the Table hash table. A single HashNode is created for every word that appears in all the files that are traversed through.

FileNodes are used as linked lists inside of a HashNode. This keeps tracks of which filenames a word occurs in, as well as how frequently that word occurs in those filenames.

Nodes are used in the List linked list.

Data Structures

There are two main data structures which are used in this program:

```
HashNode* Table[BUCKETSIZE]
node* List
```

Table is a hash table which is used for storing HashNodes. Bucket placement is determined by summing up the ascii value of each character of the word.

List is a linked list of every word sorted alphabetically. This second structure is necessary since a hash table is incapable of sorting the words alphabetically.

File Traversal

Traversal is done using the ftw library. Specifically, the function nftw is used, which traverses through the given directory or file, and calls on the given function on every directory and file it finds.

Important Functions

int f(const char* fullpath, const struct stat* temp1, int filetype, struct FTW* temp2)

This function is called on every directory and file found by nftw. Some of the arguments passed by nftw aren't needed, and are called temp. If fullpath refers to a directory, this function immediately returns 0. Otherwise, the filename is obtained from the path, and the file is opened using a file pointer. Tokens are then read from the file, refined according to specifications (must start with a letter, only alphanumeric characters, etc.), and hashed. If a HashNode* is created, the token is also added to List.

int AddtoHash(char* word, const char* file)

This function takes the word and hashes it to the hash table. If it finds a HashNode for the word, it adds the file to the filelist of the HashNode. If no HashNode is found, a HashNode is created in the bucket.

Void AddtoFlist(HashNode* F, const char* file)

This function adds the given file into the filelist linked list of the given HashNode. It adds it in order from most frequent to least frequent. If two files share the same frequency, they are ordered alphanumerically.

Void AddtoList(char* word)

This function is only called if a HashNode is created in function f. It adds a node into List in alphanumeric order.

Time and Space Analysis

There are a couple factors which affect the efficiency of the program. The first one is the function nftw. This function opens directories when it is traversing them, and this uses up memory. However, the flag FTW_DEPTH is set, which limits the number of directories the function can have open at once. The function must close directories to open more once the limit is reached (15 in our case). This trades time for memory efficiency. Next, we look at the data structures used, mainly linked lists and hash tables. The hash table is very quick, and is sorted by words. Therefore, having a large amount of words will not make the program run too slowly. However, the drawback for this setup is in filenames. Filenames are stored in a sorted linked list, so having many filenames with similar words in them will slow the program down as it must traverse the entire linked list.