Documentation Assignment 2: Keyspace Construction

Problem:
In this project we were tasked with constructing an inverted index of files and words from a specified argument directory. This program utilizes several structures native and user defined to solve this problem. The input argument is first tested if it is a file or a directory and if it is a file a quick and simple inverted index is created by reading the file into a buffer and tokenizing it. The tokenized string is then used to create an inverted index. If the input argument happens to be a directory, recursion is used to navigate the directory and the same process of tokenizing and creating an inverted index happens again.

Approach:
To be able to read all the files in the directory/file, the *recursiveFileGather* function is used. This function first reads the input argument given by the user. The function determines if the input argument is a file or a directory by using the opendir() function. If opendir() returns null then there was an error opening the directory, this could mean that the file is either a regular file or some other error. This error file checking is handled later in the program but in the *recursiveFileGather* function all paths that return NULL when sent as a parameter to openDir are considered as regular files. When *recursiveFileGather* function finds a file it first updates the folder array so that the correct file path is recorded when the file is to be open later. Then a *FileProcessElement* structure is created with the file path and file name as parameters. Each of these *FileProcessElement* strucures are nodes in a linked list so after each one is created it is placed at the beginning of the linked list of files that must be processed. After the linked list is created the first node is sent into the *readFile* function. This function opens each *fileProcessElement* structure in the linked list and uses *fopen* to read one word at a time and either creates a new *FileListElement* in the HashMap or updates the frequency of one of the *FileListElement* structure. After the frequency is updated the files are sorted in order of decreasing frequency for each word. If two files have the same frequency then they are sorted in reverse alphanumerical order.

Defined Structs

`struct fileProcessElement`	
char * rootFolder	Holds the file path for each file that has to be processed
char* fileName	Holds the file name for each file
struct fileProcessElement * next	Pointer to next file in the linked list
`struct HashNode`	
char * word	Word that was found in a file
struct FileListElement* fileElement	Pointer to first node in linked list of all the files that contain a specific word and its frequencies in files
struct HashNode * next	Pointer to next node in HashMap
`struct FileListElement`	
char * filename	Name of the file
struct FileListElement * next	Pointer to next file

int frequency	Number of times a specific word occurs in the file specified by filename

Functions

```
FileProcessElement* recursiveFileGather(char* rootDirFile,
FileProcessElement* root)
```
This function reads through the input argument and gathers all the files that must be read. It places all these files in a linked list of fileProcessElement structures and returns the pointer to the first node.

```
void readFile(char* filepathtofolder,  FileProcessElement*
root,struct HashNode* map)
```
This function opens each file and reads word by word and sends words to the tokenize function

```
void tokenize(char *word, char* file, struct HashNode* root)
```
This function makes sure each token is up to the correct specifications (i.e. is not all numbers or doesn’t start with a number)

```
struct HashNode* insertNode(char* word, char* fileName, struct
HashNode* root)
```
This function checks if the hash map already contains a node for the specified word and creates a new node if the node doesn’t exist for the word

```
void insertSorted( struct HashNode* newNode,  struct HashNode*
root)
```
This function sorts the hash nodes based on the word they represent. The hash map must be in alphanumerical order and this function inserts new nodes into the correct spot so that the hash map stays in the proper order.

```
void sortFiles(struct HashNode* hashPtr)
```
This function sorts the list of FileListElements for each HashNode based on frequency or file name if two files have the same frequency for a specific word