Assignment 2

In this assignment, we implemented a inverted indexer, which traverses a directory and tokenizes keywords, then stores the word count and filename where the token exists.

Implementation

-The tokenizing will be done by using read( ) to parse one character at a time. These tokens are not unique, and they are stored in an array. The unique tokens and their count will be put into a temporary linked list.

-The data about each unique token will be stored in a hash table of size 1000 that deals with collisions by chaining.

-The hash function sums up the values of each character in the string, then mod that value by 1000.

-Each unique token will have its own linked list, which contains data about filenames and the number of occurrences in that file.

Functions

traverseDir

void traverseDir(kNode** hashTable, char* path)

Parameters: -hashTable is a kNode** pointer to a hash table implementation
-the path (argv[2]) of the file/directory to open.

Notes:

-Checks if path is a file, directory, or invalid input. If path is a file, the tokenize(), sort(), and insertRecord() functions are called.

-If path is a directory, the directory is opened, and the directory's contents are read. For files, the same functions listed above are performed. For directories, TraverseDir is recursively called. If the filename passed as path doesn't exist, then the program exits.

Tokenize

char** tokenize(char* fileName, char** wordArray, int* wordCountPtr)

Parameters: -fileName is the file to be opened
-wordArray is a pointer to a char array
-wordCountPtr is a pointer to the number of words that the array will hold (the int at that address will be 0 when we pass it in).

Notes:

-This implementation happened because we realloc'd and need to return a pointer, but we also needed to return an int.

-the read() function reads from the buffer one character at a time. If the character is not alphanumeric, the program recognizes that a token has been completed. That token is stored in an array,

-realloc is used to account for cases where there are large numbers of characters in a token, and large numbers of tokens in a file
Returns: pointer to unsorted array of strings

sort
void sort(char** allStrings, int wordCount)
Parameters: -allStrings is a char** to unsorted string array
-wordCount is the number of words in the array
Notes:
-Insertion sort on unsorted array
Returns: void

removeDuplicates
node* removeDuplicates(char** allWords, int len)
Parameters: - allWords is a char** to a sorted string array.
-len is the length of the string array
Notes:
-traverses the array 1 time to find duplicates. Because array is sorted, duplicates are next to each other.
-stores the each token and its word count inside of a Linked List node.
Return: a pointer to the head of linked list

insertRecords
void insertRecords(kNode** hashTable, node* head, char* fileName)
Parameters: -hashTable is a kNode** pointer
-head is a pointer to a node* linked list containing keywords and count
-fileName is a string containing the file that was just read
Notes:
-this function takes in a node linked list containing keywords and count.
-the keywords are used to hash into the hash table. If a keyword doesn't already exist as a key, it is added to the hashtable as a kNode.
-A new node is created to store filename and count of that particular keyword. It is added to the kNode's fileList.
-If the filename already exists in the fileList, then do not add a new node to the fileList
Return: void

hashFunction
int hashFunction(char* str)
Parameters: str is a pointer to a token string
Notes:
-Each char has an integer value, so we add up all of the string's chars
-Take the resulting sum, mod by 1000
-This will tell us which bucket int the hashtable to place the keyword in.
Return: integer

createNode
node* createNode(char* str, int count, node* next)
Parameters: str is a string
count is an integer that counts the number of occurrence of a word in a file
next is the next node.
Notes:
-There is a function to create a new node*, but there is no function to create a new kNode*
-the str parameter is can be used to represent a keyword or a filename, depending on when it is used
-mallocs a new node and char*, uses strcpy to copy parameter into new string
Return: returns the created node

mergeSortKw
kNode* mergeSortKw(kNode* head)
Parameters: head is pointer to unsorted list of kNode*
Notes:
-Merge sorts the keywords into alphabetical order
Return: pointer to head of sorted linked list

mergeSortRecords
node* mergeSortRecords(node* head)
Parameters: head is pointer to unsorted list of node*
Notes:
-Merge sorts by putting filenames in alphabetical order
Return: pointer to head of sorted linked list

freeKNodes
void freeKNodes(kNode* head)
Parameters: head is pointer to head of kNode* linked list
Notes:
-freeKNodes calls freeNodes() to free each KNode's fileList.
-recursively calls freeKNodes() on each next kNode
Return: void

freeNodes
void freeNodes(node* head)
Parameters: head is pointer to head of node* linked list
Notes:
-recursively frees each next node
Return: void

clearBuffer
void clearBuffer(char* str, int len)
Parameters: str* is a buffer string
len is the length of str
Notes:
-sets each character in str to '\0'
Return: void