Query Design Spec

In the following Design Module we describe the input, data flow, and output specification for the query module. The pseudo code for the query module is also given.

The Design spec should include:
[1] Input: Any inputs to the module
[2] Output: Any outputs of the module
[3] Data Flow: Any data flow through the module
[4] Data Structures: Major data structures used by the module
[5] Pseudo Code: Pseudo code description of the module

[1] **Input**

Command Input
./query [INDEX FILE] [SOURCE DIRECTORY]

Example command input
Query ../indexer/index.dat ../crawler/data

[INDEX FILE] -> ../indexer/index.dat
Requirement: The file must be valid and in the correct format in order to be read correctly
Usage: The query needs to inform the user if the file is not found

[SOURCE DIRECTORY] -> ../crawler/data
Requirement: The directory must exist, have files in it, and be already created by the user
Usage: The query needs to inform the user if the directory can not be found or if the directory is empty.

[2] **Output**

For the input words that the user enters, the query will search through the given index and the list of urls in the directory and will give the user a list of urls with positive matches ranked on frequency.

The output will include the document ID, the frequency and the associated url. The user will be able to continuously enter more words to search until they decide to quit. Words written continuously or separated with AND will be clumped together and words separated with OR will be tallied separately at first and then combined later in the program before the output is sent to the screen.

[3] **Data Flow**

The index file is read and the information is stored into a hashtable data structure that holds the doc_id and frequency for each word in the index. The program then prompts the user to enter words to be searched.

The program will then parse through the user's input and separate words into groups based on which inputs are separated by OR. Those will be clumped together and put into a wordlist data structure. Once this is done for all user inputs, the program will add the words to a docList data structure that will keep track of all the doc_id's and frequencies for the words that we are searching for.

Once the list is made, dealing with the differences in AND and OR inputs the docList will be passed to a bubble sort. The components will be rearranged such that the high frequency docNodes come first.

Once the sorting finishes, they will be passed to a print function that will take the information given and open the right file in the source directory. It will print to the user the results of the find and give the doc_id, freq, and url.

[4] **Data Structures**

For the functionality of this program, several data structures were created to hold relevant information.

WNode – is a data structure that links other WNodes and the main purpose of this node is to keep track of all the words that we will be searching for

ListNode – is a data structure that links to other ListNodes and the main purpose of this node is to create a list of all the words that the user inputted

List – is a data structure used to hold references to ListNodes

wordListNode – is a data structure that holds WNode's and also links to other wordLIstNodes

wordlist – is a data structure used to hold references to wordListNodes

docNode – is a data structure that links other docNodes and the main purpose of this node is to keep track of the doc_id's and frequencies of the words that we are searching for in their respective urls.

[5] **Query Pseudocode**

```
//check command line arguments
Inform the user if the arguents are not present or invalid.
IF the index file or source directory do not exist, inform
the user of usage and exit failed

//Enter a while loop
This program will run continuously until the user decides
to quit

//take user input
Allow the user to input up to 1000 characters

//parse the user input
Split the user input based on space delimiters to get a
record of all the words that the user inputted

//Deal with AND and OR
group words based on if they are separated by spaces,
AND's, or OR's

//Add word to list
add the words to a list that will be grouped

//create a doclist
for each word, go through and check to see if they match
with any words in the directory files and if they do, add
them to the doclist

//sort
sort the results in the doclist based on a bubblesort.
Rank the outputs based on frequency of search success with
the highest coming first

//print
print the results to the user

//free
```

Free all of the data structures and memory that we allocated during this program.

Query Implementation Spec
In this implementation specification we define the prototypes and data structures in details. It defines the abstract data types (ADT) for the query design module. This specification code should compile without errors as the query.

The “top” header file querylist.h includes common stuff:
[1] data structures
[2] external functions

Let’s begin with the detailed language dependent data structures.

```
//--------------------Structures/Types
typedef struct WNode{
  struct WNode *next;
  char *word;
} WNode;

typedef struct ListNode {
  char *word;
  struct ListNode *next;     //pointer to the next node
} ListNode;

typedef struct List{
  ListNode *head;            //"beginning" of the list
  ListNode *tail;            //"end" of the list
} List;

typedef struct wordListNode{
  WNode *wnode;
  struct wordListNode *next;
} wordListNode;

typedef struct wordList{
  wordListNode *head;
  wordListNode *tail;
} wordList;

typedef struct docNode{
  struct docNode *next;
  int doc_id;
  int freq;
```

```
} docNode;

typedef struct docList{
  docNode *head;
} docList;
```

Next, we define some of the detailed prototype specifications

```
//This function initialized the list by setting the head
and tail to null
wordList *initializeList();

//This function will take the given page and add it as a
node to the end of the current list
void addtolist (List *currentlist, char *currentword);

//This function frees the word list
void freewordlist(wordList *currentlist);

//This function frees the list
void freelist(List *currentlist);
#endif // LIST_H
```

These are all the necessary functions and data structures that must be declared outside of the main c.file, query.c