Query Engine Design Spec

(1) Input

./query [INDEX FILE] [CRAWLER FILES DIR]

Example: ./query ../indexer/index.dat ../crawler/lvl1/

[INDEX FILE] ../indexer/index.dat Requirement: The file must exist, and be a valid index file.

[CRAWLER FILES DIR] ../crawler/lvl1 Requirement: The directory must exist and contain crawler files.

The query engine will then as for input, which should be in the format: wordA AND wordB wordA or wordB wordA wordB (which is the equivalent of an AND)

It will continue to do so and produce relevant output until an end-of-file is given.

(2) Output

Each time a search input is entered, query will display a list of relevant pages in the format: Document ID: [DOC ID] URL: [DOC URL] These pages will be sorted by relevance, with the most relevant pages to the given search terms at the top.

(3) Data Flow

When the query is run, it will first check for valid arguments. If the file and directory are given exist, then it will continue and create an index from the given index file. At that point it will then prompt the user to enter search words.

It will then process the input to determine be able to correctly calculate relevance. It will group by AND operator, adding any group of terms (i.e. cat AND dog, mouse deer pig, etc) to a List, and then adding each of these andLists to a List. Query will then go through each of the andLists, and create a single list from them of all the documents containing each word.

The result of this are documents lists separated by the OR operator. It will combine two of these lists at a time, and the end result will be a single list of documents with their calculated relevance to the search. This list of documents will then be moved into an array, where an insertion sort will sort them from highest to lowest.

The results will then be printed to the user, with the url being retrieved from the corresponding file as query goes through the array. Query will then prompt the user to enter another search, and the process will continue until an end of file is given.

(4) Data Structures

```
WordNode - a node containing a word and pointer to a DocumentList.

DocumentNode - a node containing a document ID, and the number of
           occurences of a specific word.

DocumentList - a list of all the documents containing a given word.

Index - the index of all of the words processed so far, and
        their corresonding DocumentLists.
```

(5) Pseudocode

```
1. Parameter processing and error checking
2. Reading index from file
3. Prompt user for search, until given EOF:
    1. Create lists of words by AND operator
    2. Process each list, creating a smaller number of lists to be processed by the OR operator
    3. Going through two at a time, create a single list out of the two based of OR relevance. Finish with a single list
    4. Copy list into an array, and use insertion sort to sort by the highest relevance.
    5. For each document in the array, print out the document ID and the page URL
4. Free the index memory
5. Done.
```

Query Engine Implementations Spec

(1) Useful constants and macros

```
//The number of "buckets" of the index hashtable
#define MAX_HASH_SLOT 10000

//Print debug/progress statements
#define DEBUG
```

(2) Data Structures and Variables

```
// Key data structures for storing words and the docs that contain them
typedef struct WordNode {
    struct WordNode *next;      //pointer to the next WordNode
    char *word;                         //the word
    List *docList;                      //pointer to the page list
    int numDocs;            //number of docs word was found in
} WordNode;

typedef struct DocumentNode {
    int doc_id;                         //document identifier
    int freq;                           //number of occurrences of the word
} DocumentNode;


// List data structure
typedef struct ListNode {
    void *data;             //generic data pointer
    struct ListNode *prev;          //pointer to previous node
    struct ListNode *next;          //pointer to next node
} ListNode;

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



//Hashtable data structure
typedef struct HashTableNode {
    void *data;                                 // holds a WordNode
    struct HashTableNode *next;                 // pointer to next node
} HashTableNode;

typedef struct HashTable {
    HashTableNode *table[MAX_HASH_SLOT];        // actual hashtable
} HashTable;

Hashtable index;                              // the inverted index itself
```

(3) Prototype Definitions

```
// List functions

/*
 *@list: a List structure
 *
 *Initializes the given list.
 */
void initList(List *list);

/*
 *@list: the List to add to
 *@data: the data to insert into the new node
 *
 *Adds a new node containing the given data at the end
 *of the given list
 */
void appendToList(List *list, void *data);

/*
 *@list: the list to pop from
 *
 *Returns a void pointer to the data in the first node and
 *removes that node
 */
void * popFromList(List *list);

/*
 *@list: the list to check
 *
 *Returns 0 if the List is empty, 1 otherwise
 */
int listHasNext(List *list);

/*
 *@list: the list to be freed
 *
 *Frees all the allocated memory in the list.
 */
void freeList(List *list);



// Hashtable functions --------------------------------------------

/*
 * jenkins_hash - Bob Jenkins' one_at_a_time hash function
 * @str: char buffer to hash
 * @mod: desired hash modulus
 *
 * Returns hash(str) % mod. Depends on str being null terminated.
 * Implementation details can be found at:
 *     http://www.burtleburtle.net/bob/hash/doobs.html
 */
unsigned long JenkinsHash(const char *str, unsigned long mod);

/*
 *@hash: a HashTable structure
 *
 *Initializes the given HashTable
 */
void initHash(HashTable *hash);

/*
 *@hash: the HashTable to add to
 *@data: the data to insert into the new node
 *
 *Adds a new node to the corresponding Hash slot
 */
void addToHash(HashTable *hash, void *data);

/*
 *@hash: the HashTable to check in
 *@data: the data to be checked for
 *
 *Returns 0 if the HashTable contains the given data, 1 otherwise
 */
int checkHash(HashTable *hash, void *data);

/*
 *@hash: the hashtable to be freed
 *
 *Frees the allocated memory in the HashTable
 */
void freeHash(HashTable *hash);


//------------------- web.c word parsing functions -------------------------

/*
 * NormalizeWord - lowercases all the alphabetic characters in word
 * @word: the character buffer to normalize
 *
 * Word is modified in-place, with all uppercase letters lowered.
 *
 * Usage example:
 * char *str = "HELLO WORLD!";
 * NormalizeWord(str);
 * // str should now be "hello world!"
 */
void NormalizeWord(char *word);


//---------------------------- File functions ----------------------------


/*
 * IsDir - determine if path is a directory
 * @path: path to check
 *
 * Returns non-zero if path is a directory; otherwise, 0.
 *
 * Usage example:
 * if(IsDir(".")) {
 *     // "." is a directory
 * }
 */
int IsDir(const char *path);



/*
 * IsFile - determine if path is a file
 * @path: path to check
 *
 * Returns non-zero if path is a file; otherwise, 0.
 *
 * Usage example:
```

```c
 * if(IsFile("/etc/passwd")) {
 *     // "/etc/passwd" is a file
 * }
 */
int IsFile(const char *path);


//--------------------------- Indexer functions -----------------------------

/*
 *@file: a preexisting index file
 *
 * Creates an index from a preexisting index file
 *
 * Returns an index of all the data in the file
 */
HashTable * indexFromFile(char *file);


/*
 *@index: the index to be freed
 *
 * Frees the allocated memory in the index
 */
void freeIndex(HashTable *index);


//--------------------------- Query functions -------------------------------


/*
 *@index: a valid index, from indexFromFile()
 *@line: a search line
 *@dir: the directory where corresponding crawler files are located
 *
 *Takes in a query and displays links by relevance
 *
 */
int processQuery(HashTable *index, char *line, char *dir);


/*
 *@line: the input to be checked
 *
 * Checks the line for invalid input
 *
 * Returns 1 if line is invalid, 0 otherwise
 */
int checkLine(const char *line);


/*
 *@index: a valid index
 *@word: the word to find in the index
 *
 * Returns the given WordNode for a word, or NULL if non-existant
 */
WordNode * findWordNode(HashTable *index, char *word);

/*
 *@index: creates an orList of WordNodes
 *@inputList: a list of words and OR operators
 *
 *Creates an orList from the input list by processing all of the and operators
 *
 *Returns a list of WordNodes that the OR operator must be performed on
 */
List * createOrList(HashTable *index, List *inputList);


/*
 *@orList: a list of WordNodes that the OR operator must be performed on
 *
 * Processes the orList, creating a single combined list
 *
 * Returns a list of DocumentNodes for the given query
 */
List * processOrList(List *orList);


/*
 *@node1: an existing WordNode
 *@node2: an existing WordNode
 *
 * Performs the AND operator on the two nodes
 *
 * Returns a single WordNode with a new DocumentNode List
 */
WordNode * processAND(WordNode *node1, WordNode *node2);


/*
```

```
 *@list1: a List of DocumentNodes
 *@list2: a List of DocumentNodes
 *
 * Combines the two lists into a single list based on the OR operator
 *
 * Returns the combined list of DocumentNodes
 */
List * processOR(List *list1, List *list2);


/*
 *@curWord: the WordNode to be copied
 *
 * Makes a copy of the given WordNode
 *
 * Returns a pointer to the newly made copy
 */
WordNode * copyWordNode(WordNode *curWord);


/*
 *@docList: a List of DocumentNodes to be sorted
 *
 * Sorts the docList from low to high frequency
 */
void sortDocs(List *docList);


/*
 *@docId: the id of the desired file
 *@dir: the directory the file is located in
 *
 * Retrieves the url from the given file
 *
 * Returns the found url, or NULL if an error occured
 */
char * getFileUrl(int docId, char *dir);
```

Error Conditons

1. Invalid given file or directory
2. Invalid index file format

3. Input in the form of: AND word word AND word OR word AND OR word2 etc.
4. Index file not for corresponding query directory

Test cases and Results

```
/*
  Test Harness Spec:

  Uses these functions, but does not unit test them
    void initList(List *list);
    void appendToList(List *list, void *data);
    void * popFromList(List *list);

  It tests the following functions:

  int checkLine(const char *line);
  WordNode * processAND(WordNode *node1, WordNode *node2);
  List * processOR(List *list1, List *list2);
  void sortDocs(List *docList);
  char *getFileUrl(int docId, char *dir);



  If any of the tests fail it prints status
  If all tests pass it prints status.


  Test Cases:
  -----------

  The test harness runs a number of test cases to test, by setting up
  the environment for the test, invoking the functions being tested,
  and using SHOULD_BE to validate the return.


  Cases for:
  int checkLine(const char *line);

  Test case: checkLine:1
  -This checks for AND at the beginning of the input

  Test case: checkLine:2
  -This checks for OR at the beginning of the input
  Test case: checkLine:3
  -This checks for AND at the end of input

  Test case: checkLine:4
  -This checks for OR at the end of input

  Test case: checkLine:5
  -This checks for a consecutive AND OR

  Test case: checkLine:6
  -Checks with valid input



  Cases for:
  WordNode * processAND(WordNode *node1, WordNode *node2);

  Test case: processAND:1
  -This checks to see if processAND correctly creates a new node containing
   the correct list of documents

   Cases for:
   List * processOR(List *list1, List *list2);

  Test case: processOR:1
  -This checks to see if processOR correctly creates a new list of documents,
   with documents found in both lists having combined frequencies


   Cases for:
   void sortDocs(List *docList);

   Test case: sortDocs:1
   -Tests sortDocs when given an empty list

   Test case: sortDocs:2
   -Tests sortDocs when given a list of a single document

   Test case: sortDocs:3
   -Checks that sortDocs correctly sorts a list of multiple documents
*/
```