***Query Implementation Spec***
------------------------------------------

### Key Data Structures

```
typedef struct DocumentNode {
 struct DocumentNode *next;
 int doc_id;
 int freq;
} DocumentNode;

typedef struct WordNode {
 struct WordNode *next;
 char *word;
 DocumentNode *page;
} WordNode;

typedef struct HashTableNode {
   void *data;
} HashTableNode;

typedef struct HashTable {
   HashTableNode *table[MAX_HASH_SLOT];
} HashTable;
```

### Prototype Definitions

```
/*
 * ReadFile - Read from a file to create an InvertedIndex.
 * @file_name: file to be read.
 * @New_Index: pointer to an InvertedIndex to be created.
 *
 * Returns a pointer to an InvertedIndex.
 *
 * Pseudocode:
 *     1. Read each line of file_name.
 *     2. Parse for the word and the number of docs the word appears in.
 *     3. Create a WordNode and add to the InvertedIndex.
 *     4. Loop through the rest of the line, parsing for doc_id and freq.
 *     5. Create a DocumentNode and add to the InvertedIndex.
 */
HashTable *ReadFile(char *, HashTable *);
```

```
/*
 * InHashTable - checks whether a word is in the InvertedIndex.
 * @WORD: word to be searched.
 * @Index: pointer to the InvertedIndex.
 *
 * Returns 0 if the word is not in the InvertedIndex.
 * Returns i if match is found, where i refers to the position of the matching WordNode in the bin.
 *                      For instance, if the matching WordNode is the second WordNode in the bin, then i = 2.
 *
 * Assumptions:
 *     1. InvertedIndex has been initialized.
 *
 * Pseudocode:
 *     1. Compute the hash code.
 *     2. Loop through all the WordNodes of the bin.
 *     3. If the matching WordNode is found, then return.
 *     4. Else, return 0.
 *
 */
int InHashTable(char *, HashTable *);


/*
 * CleanHashTable - frees all WordNodes and DocumentNodes in the InvertedIndex.
 * @Index: pointer to the InvertedIndex.
 *
 * Returns 0 at program termination.
 *
 * Assumptions:
 *     1. InvertedIndex has been initialized.
 *
 * Pseudocode:
 *     1. Loop through the entire InvertedIndex.
 *     2. Add WordNode and word pointers to arrays of WordNode and word pointers.
 *     3. Add DocumentNode pointers of each WordNode to array of DocumentNode pointers.
 *     4. Looping backwards through the three arrays, free the content in them.
 *
 */
int CleanHashTable(HashTable *);
```

```
/*
 * FreeHashTable - frees all HashTableNodes of the InvertedIndex.
 * @Index: pointer to the InvertedIndex.
 *
 * Returns 0 after function is run.
 *
 * Assumptions:
 *     1. Each HashTableNode is empty, and is not pointing to any structure.
 *
 * Pseudocode:
 *     1. Loop through each bin of the InvertedIndex.
 *     2. Free HashTableNode.
 */
int FreeHashTable(HashTable *);


/*
 * InitializeHashTable - initializes the InvertedIndex by creating empty HashTableNodes.
 * @Index: pointer to the InvertedIndex.
 *
 * Returns 0 after function is run.
 *
 * Assumptions:
 *     1. InvertedIndex is empty / has not been initialized.
 *
 * Pseudocode:
 *     1. Loop through each bin of the InvertedIndex.
 *     2. Declare and initialize empty HashTableNodes.
 */
int InitializeHashTable(HashTable *);


/*
 * GetLinks - get all matched DocumentNodes for a given query.
 * @line: query line to be searched for.
 * @Index: InvertedIndex containing all word-document pairs.
 *
 * Returns 1 if successful.
 * Returns 0 if not successful.
 *
 * Pseudocode:
 *     1. Read in each word of the query.
 *     2. Save all matched DocumentNodes in temp_list.
 *     2. If an AND is passed, or there is no logical operator between two words,
intersection operation is done on temp_list.
```

```
 *     3. If an OR is passed, data in temp_list is flushed out to final_list using the union
operation.
 *     4. Continue until end of line.
 */
int GetLinks(char *, HashTable *);


/*
 * And - Perform intersection operation.
 * @word: word to get new list of matching DocumentNodes from.
 * @Index: InvertedIndex containing all word-document pairs.
 *
 * Pseudocode:
 *     1. Find the start of the list of matching DocumentNodes for the word.
 *     2. For each DocumentNode in temp_list, go through the new list looking for doc_id
matches.
 *     2. If there is a match, keep that DocumentNode in temp_list and increment
frequency.
 *     3. If there is no match, delete that DocumentNode from temp_list.
 *     4. Continue until end of temp_list.
 */
void And(char *, HashTable *);


/*
 * Or - Perform union operation.
 *
 * Returns 0 and terminates.
 *
 * Pseudocode:
 *     1. If final_list is NULL, set final_list to temp_list, temp_list to NULL, and return.
 *     2. For each DocumentNode in temp_list, go through final_list looking for doc_id
matches.
 *     2. If there is a match, keep that DocumentNode in final_list and increment
frequency.
 *     3. If there is no match, create a copy DocumentNode and add to end of final_list.
 *     4. Continue until end of temp_list.
 */
int Or();
```

```
/*
 * Sort - Sort the list of DocumentNodes by rank, from highest to lowest.
 *
 * Pseudocode:
 *     1. Put DocumentNodes into an array.
 *     2. Sort the array by rank.
 *     2. Refactor sorted array into list of DocumentNodes.
 */
void Sort();


/*
 * key_compare - key compare function for qsort.
 * @e1: element to compare.
 * @e2: element to compare.
 *
 * Returns -1 if e1>e2, 1 if e1<e2, 0 if e1==e2.
 *
 * Pseudocode:
 *     1. Get the frequency of the two passed DocumentNodes.
 *     2. Return appropriate comparison value.
 */
int key_compare(const void *, const void *);


/*
 * FreeList - free memory of DocumentNode lists.
 * @choice: the type of list to free. If 0, free temp_list, and if 1, free final_list.
 *
 * Returns 0 and terminates.
 *
 * Pseudocode:
 *     1. For the appropriate list, loop through each DocumentNode.
 *     2. Free DocumentNode and move on to the next one.
 *     3. Free until list is empty.
 */
int FreeList(int);


/*
 * Display - display query matches to output.
 *
 * Returns 1 if successful.
 * Returns 0 if not.
 *
 * Pseudocode:
```

```
 *     1. Get each DocumentNode of final_list.
 *     2. Get filename and open a stream to that file.
 *     3. Read in the URL address from stream.
 *     4. Output doc_id and url to stdout.
 *     5. Cleanup memory and close stream.
 */
int Display();
```

Standard Linux system calls used

```
 *
* void free(void *ptr);
 * void *malloc(size_t size);
 * void *calloc(int num, size_t size);
 *
 */
```

Macros

```
#define MAX 1000
// Max number of characters for a single query search.

#define MATH_MAX(X, Y) (((X) > (Y)) ? (X) : (Y))
// Returns max of the two numbers passed.
// Used to get the rank of a document in case of union operation.

#define MAX_HASH_SLOT 10000
// Max number of bins of the InvertedIndex.
```