Indexer Design Spec
---

(1) *Input*

Command Input

```
./query [INDEXER OUTPUT FILE] [CRAWLER OUTPUT FILE DIRECTORY]
```

Example command input:

```
./query ./index.dat ./WebPageDir
```

`[INDEXER OUTPUT FILE]./index.dat`
Requirement: The file must exist and contain the output files of indexer.
Usage: The indexer needs to inform the user if the file is not found.

`[CRAWLER OUTPUT FILE DIRECTORY] ./WebPageDir`
Requirement: The directory must exist and contain the output files of crawler component.
Usage: The indexer needs to inform the user if the directory is not found.

(2) *Output*

query will write out all the results from a given query input, such as "dog AND cat" in ranked order.
The format is as follows:

Document ID:someNumber URL:http://someurl.com\n

As well as the total pages found right after all the URLs found.
The documents with rank numbers > 100 will be printed in Green, > 50 in Cyan, > 10 in Yellow, and <= 10 in Magenta.

(3) *Data Flow*

An inverted index will be created from reading the file specified in the second argument.

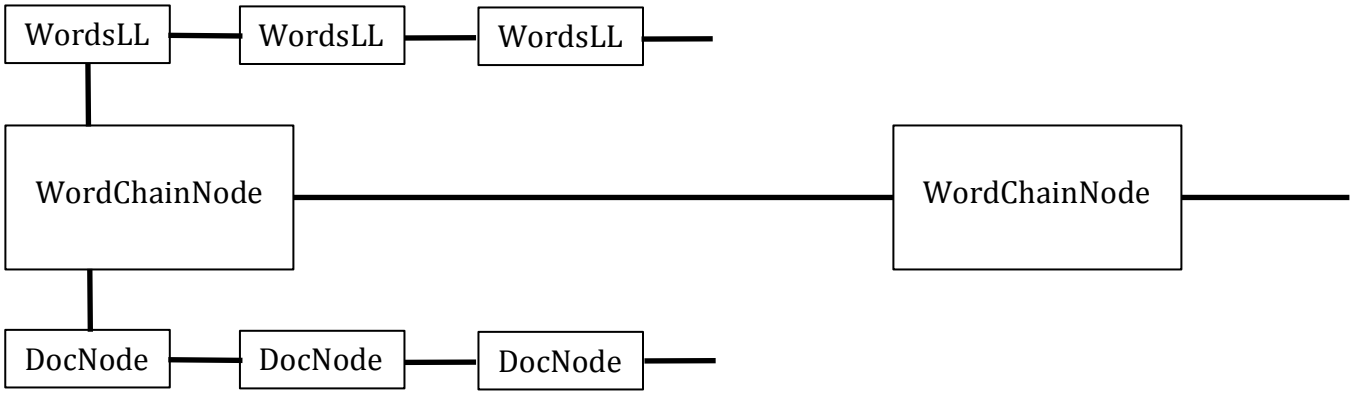
Query will take in input of a query from stdin, and will look for the appropriate words in the inverted index.

Query will then process the information gained from the inverted index using a OR and AND processing algorithm, and will print to stdout the query results in ranked order.

(4) *Data Structures*

`wordList`

A list structure with a node containing two linked list, one of WordsLL (char *) and another for DocNodes.
Diagram of structure:



Each WordChainNode's WordsLL contain words from the query that should be ANDed. By the end of the AND processes, each WordChainNode's DocNode list will contain the results of the ANDed DocNodes of all the words in WordsLL list.
Then the individual WordChainNodes will be ORed so that by the end of these processes, query will have a DocNode chain of all the results from the query input that can be sorted by rank and printed.

`invertedIndex`

A dictionary structure that holds words, which documentID it occurred and how many times it occurred at each do.

(5) *Pseudo Code*

1. Make new inverted index from given index.dat file

2. Take input for query from user

3. Add the words appropriately to a SinLL structure

4. Process the words that should be ANDed

5. Process the AND processed DocNodes to be ORed

6. Sort the final DocNode in rank sorted order

7. Get individual URLS for the final DocNodes to print as the results.

8. Get user input for query again

9. Keep going until user inputs EOF.

10. Done.

Implementation Specification
---

(1) *Data Structures And Variables*

linked list that contains a linked list of words, and a linked list of DocNodes.

```
typedef struct WordChainNode {
   struct WordsLL *words;                  // head of the linkedlist of words
   struct WordChainNode *nextWords;        // next pointer to the next set of words and
DocNodes
   struct DocNode *docs;                   // head of the linkedlist of DocNodes
} WordChainNode;
```

linked list of words to be stored in WordChainNode. In query, these words represent the words to be ANDed.

```
typedef struct WordsLL {
   char *word;                             // word to be stored
   struct WordsLL *nextWord;               // points to next same structure
} WordsLL;

typedef struct SinLL {
      WordChainNode *head;                 // "beginning" of the list
      WordChainNode *tail;                 // "end" of the list
} SinLL;
```

(2) *Prototype Definitions*

queryProcess.c Functions:

```
/*
 * Compare function passed into merge sort function to compare the
 * DocNode's occurrences.
 * returns 1 if doc1 has a greater occurrences.
 * returns 0 if doc1 and doc2 has the same occurrences.
 * returns -1 if doc1 has a smaller occurrences.
 */
int compareOccurrences(DocNode *doc1, DocNode *doc2);

/*
 * Compare function passed into merge sort function to compare the
 * DocNode's documentID.
*/
int compareIDs(DocNode *doc1, DocNode *doc2);

/*
 *  Looks for the given word in the inverted index, gets its associated
 *  WordNode, get its docs field (Linked List of DocNodes), and returns
 *  this DocNodes.
 */
DocNode *DocsFromWordNode(char *word, HashTable *hashTab);

/*
 * Copies the DocNode provided to the function. Allocates this copy
 * on the heap. The caller is responsible for the freeing of this
 * memory.
*/
DocNode *CopyDocs(DocNode *docHead);

/*
 * Function used for AND and OR processes.
 * Takes in pointers to two DocNode lists, and merges them together
 * in sorted order according to their document IDs. (descending order)
*/
int DocMergedID(DocNode **doc1, DocNode **doc2);

/*
 * Function used for AND processing
 * Takes output from DocMergedID and leaves only the duplicates
 * Makes duplicates into one node by adding the second occurrence
 * to the first DocNode occurrence.
 */
int ProcessAND(DocNode **doc1);

/*
 * Function used for OR processing.
 * Takes output from DocMergedID and make duplicates into one by
```

```
 * adding the second occurrence to the first docnode occurrence.
 */
int ProcessOR(DocNode **doc1);

/*
 * Sorts given DocNode by occurrences "Rank".
 */
void SortByRank(DocNode **docToSort);

/*
 * For all document ID in DocNode of DocNode list, the function
 * will find the URL from specific crawler files, and will print out
 * a line per document formatted:
 *           Document ID: documentID URL: http://url.com \n
 */
void PrintQueryResult(DocNode *printHead, char *webPageDir);
```

LLMergeSort.c Functions:

```
/*
 * Overall sorting function that performs merge sort on a
 * linked list of DocNodes.
 */
DocNode *Merge(DocNode *doc1, DocNode *doc2, int(*compareFunc)(DocNode *doc1, DocNode
*doc2));

/*
 * Splits a given DocNode list in half and store the two halves
 * at firstHalf and lastHalf.
 */
int SplitHalf(DocNode *toSplit, DocNode **firstHalf, DocNode **lastHalf);

/*
 * Merges the DocNode list doc1 and doc2 in correct sorted order
 * according to compareFunc.
 */
void MergeSort(DocNode **docHead, int(*compareFunc)(DocNode *doc1, DocNode *doc2));
```

SinLL.c Functions:

```
/*
 * Allocates memory for the list and returns a pointer to
 * this memory.
 */
SinLL *CreateSinLL();

/*
 * append a word to the current node (tail) of the linked list
 */
int appendWord(char *word, SinLL *linkedList);
```

```
/*
 * creates a new WordChain in the linked list, not just a
 * new WordsLL node like appendWord function.
 */
int appendNewWordChain(char *word, SinLL *linkedList);

/* remove from top of SLL and put the DocNode chain that was on that
 * node to tempDocPtr.
 */
int removeTopDoc(SinLL *linkedList, DocNode **tempDocPtr);

/*
 * Deletes the full node of the linked list.
 */
int DeleteWordChainNode(WordChainNode *toFreeNode);

/*
 * Deletes the word linked list withing the node of the linked list.
 */
int DeleteWordsLLChain(WordsLL *toFreeWords);

/* checks if list is empty
 */
int IsEmptySinLL(SinLL *linkedList);

/*
 * Adds a given DocNode list to the given WordChainNode.
*/
int AddDocNodeChain(WordChainNode *toAddNode, DocNode *docNodeChain);
```

Functional Specification
---

query.c

The operations performed by the query engine are as follows:

- it should load a previously generated index from the file system;
- it should receive user queries from input;
- it should convert capital letters of words of the query to lower case (i.e., we do not distinguish between dog and Dog;
- for each query, it should check the index and retrieve the results matching the query;
- it should rank the list of results, returning the document with the highest number of matches to the query string, then the second highest, and so on; and
- it should display the results to the user.

queryProcess.c/h

Defines all primary functions used to process query prompts. Functions defined are compareOccurrences, compareIDs, DocFromWordNode, CopyDocs, DocMergeID. ProcessAND, ProcessOR, SortByRank, and PrintQueryResult. All these functions called by query when processing query inputs for print the results to the user.

LLMergeSort.c/h

Implements merge sort on a linked list of DocNodes Functions defined are MergeSort, Merge, and SplitHalf, all needed to make the recursive merge sort work. Merge sort is used in queryProcess to process for OR and AND, and when sorting the results by Rank.

SinLL.c/h

Defines a linked list structure that contains a list of words and DocNodes. Functions defined are create, appendWord, appendNewWordChain, removeTopDoc, DeleteWordChainNode, DeleteWordsLLChain, IsEmptySinLL, and AddDocNodeChain. All called by query to store the words and DocNodes of the words for processing by queryProcess,c

summary of error conditions detected and reported
---

1) With no arguments, error message should show to inform user of usage.
   Usage: query [INDEXER OUTPUT FILE] [CRAWLER OUTPUT FILE DIRECTORY]

2) With too many args, error message should show to inform user of usage.
   Usage: query [INDEXER OUTPUT FILE] [CRAWLER OUTPUT FILE DIRECTORY]

3) With invalid directory argument, query will inform that the given directory is not a directory.
   Second argument is not a directory.

4) With invalid indexer file argument, the query will inform that the givem file is not valid.
   First argument is not a valid file.

5) With query of more than 1000 characters, query will inform the user to give a query of less than 1000 characters.
   Query length is over the maximum 1000 characters!

6) Any calloc failures of inverted index that query depends on:
   Failed inverted index building.

test cases, expected results
---

1) Query: cat AND dog (testing AND condition)
   Should have found 12 files.

2) Query: CAT AND DOG (testing for lowercasing)
   Should have found 12 files.

3) Query: or dog (testing space condition which should AND, and a non-existing first word)
   Should have found 0 files.

4) Query: or OR dog (non-existing first word ORed with a valid word)
   Should have found 55 files.

5) Query: cat OR dog (testing OR condition)
   Should have found 257 files.

6) Query: cat and dog (testing space condition which should AND, and lowercase operators)
   Should have found 12 files.

7) Query: cat dog (testing space condition which should AND)
   Should have found 12 files.

8) Query: cat AND dog OR head AND tail (testing longer cases with combination of OR and AND)
   Should have found 178 files.

9) Query:
   Should have found 0 files.

10) Query: ksvjsngvaen (testing non sense input)
    Should have found 0 files.