Kathy Dong
CS50 – Winter 15
Design and Implementation Spec – Indexer

TINY SEARCH ENGINE DESIGN

Tiny search engine is a 3-part lab: crawler, indexer, and queryEngine.

- **Crawler:** takes a URL and retrieves the html contents of the webpage and subsequent nested webpages (up to a certain depth). The contents are then saved into a directory file, which includes the URL, depth, and html content.
- **Indexer:** takes the crawled files and created an index of all of the different words from the crawled pages, as well as the frequency they occur in each document.
- **QueryEngine:** takes the index.dat file generated from indexer and the crawled files generated from crawler to create a command-line search. QueryEngine parses the user query appropriately (taking into consideration false words, capitalization, AND and OR commands, etc.), and then searches for the documents that include the highest frequencies of the search input. The documents and URLs are then displayed in a sorted list.

All of these components can be run independently, but the components are closely related and build off of each other.

Tiny Search Engine master Makefile: The TSE master Makefile calls all of the subsequent Makefiles that are necessary to compile the full search engine. Specifically, the Makefile calls the crawler, indexer, and queryEngine Makefiles.

QUERY ENGINE DESIGN

1. INPUT

To compile in shell, enter:
```
make
```

The makefile will call another makefile in the utils directory to create libsteutils.a library, which includes major functions included in file.c, web.c, and index.c. The library provides essential functions for queryEngine.

To run with 3 arguments, enter:
```
./queryEngine [index.dat file] [directory with files]
```

Arguments:

- argv[0] = ./queryEngine
- argv[1] = index.dat file (gives word, document count, and documents/frequencies) **generated from indexer
- argv[2] = directory of previously crawled files **generated from crawler

Notes: queryEngine will run if there is a mismatch between the depth of index.dat and the directory; however,
the user will be told that the file does not exist and the search is invalid.

Requirements:

- must be run with either 3 arguments
- argv[1] = index.dat file must exist
- argv[2] = directory must be valid; if invalid, will return an error to the user directory must not be empty; if empty, return error

Example Command Line:

3 args: ./queryEngine index.dat /net/class/cs50/public_html/tse/crawler/lvl1

2. OUTPUT

QueryEngine is set in an infinite loop; to exit the program, the user must issue an EOF Ctrl+D call.

QueryEngine takes the user query and performs a search to find the most relevant results. The output to stdout shows all of the documents that are relevant to the particular search, as well as the corresponding URLs in sorted order.

Example Output:

```
QUERY:> computer
Document ID: 2 - URL: http://old-
www.cs.dartmouth.edu/~cs50/tse/wiki/Computer_science.html
Document ID: 7 - URL: http://old-
www.cs.dartmouth.edu/~cs50/tse/wiki/Linked_list.html
Document ID: 3 - URL: http://old-
www.cs.dartmouth.edu/~cs50/tse/wiki/C_(programming_language).html
Document ID: 6 - URL: http://old-
www.cs.dartmouth.edu/~cs50/tse/wiki/Hash_table.html
Document ID: 4 - URL: http://old-www.cs.dartmouth.edu/~cs50/tse/wiki/Unix.html
Document ID: 5 - URL: http://old-
www.cs.dartmouth.edu/~cs50/tse/wiki/Dartmouth_College.html
Document ID: 1 - URL: http://old-www.cs.dartmouth.edu/~cs50/tse/

QUERY:> coconut mangoes OR grapefruits
No documents found

QUERY:> xavier
Document ID: 1 - URL: http://old-www.cs.dartmouth.edu/~cs50/tse/

QUERY:> animation OR rendering AND AND OR AND OR linguistics
Document ID: 2 - URL: http://old-
www.cs.dartmouth.edu/~cs50/tse/wiki/Computer_science.html
Document ID: 7 - URL: http://old-
www.cs.dartmouth.edu/~cs50/tse/wiki/Linked_list.html
```

3. DATA FLOW

QueryEngine is set in an infinite loop; to exit the program, the user must issue an EOF Ctrl+D call.

3 arguments design procedure:

- Argument checking
- Rebuilds InvertedIndex hashtable from index.dat
- Gets query input entered by user in stdin
- Parses query input, grouping words appropriately based on AND or OR, puts all of the words into QueryNodes
- Intersects the documents for all of the AND words (finds the common documents between the words, discards rest)
- Unionizes the documents for all OR words
- Sorts all of the valid documents
- Print out document id and URL (URLs extracted from crawler files in given directory)

Specific new functions include the UnionList and IntersectList included in index.c. These functions are called in the more general query functions from query.c.

4. DATA STRUCTURES

- **InvertedIndex:** hashtable that will include all of the information regarding word, docID, and freq. InvertedIndex will only directly include WordNodes in a linked list format for collisions.

- **WordNode:** direct content in InvertedIndex nodes; also included in QueryNodes
  - Next: pointer to the next wordNode in case of collisions
  - Num_count: number of documents that include the word
  - Word: pointer to character array of word
  - DocumentNode: data structure (described below)

- **DocumentNode:** content inside of WordNode
  - Next: pointer to next DocumentNode that includes the word
  - Doc_id: integer id for documents
  - Freq: integer number of times the word occurs in the particular document

- **QueryNode:** QueryNodes are used to keep track of groups of user queries (word groupings that are separated by OR)
  - WordNode: the words that are valid (can be found in the InvertedIndex) from the query search
  - num_words; integer for number of words included in the QueryNode; if the Node becomes invalidated when a word that cannot be found is searched for, num_words is set to 0
  - rel_docs: relevant documents (all of the documents that are still included in the search after performing the AND operator)
  - Next: pointer to next QueryNode

5. PSEUDOCODE

```
// check command line arguments; notify users if there are command line errors

// create InvertedIndex hashtable from index.dat supplied by user

// enters infinite loop until Ctrl + D command is issued; user enters query

        // get query entered by user

        // get query structure; populate QueryNodes based on parsed input from the users

        // find the documents that satisfy the AND command (the documents are found within
        all of the words); with AND commands, the documents are combined across all of the
        WordNodes within each queryNode

                // make a shallow copy of each WordNode so we can use an updated *next
                pointer to the next WordNode

        // get all documents that satisfy the OR command

                // make shallow copies of the DocumentNodes for flexibility with *next

                // update all of the frequencies

        // put all of the DocumentNodes into an array of pointers to DocumentNodes

        // perform a basic bubble sort to sort the documents by frequency

        // print all of the documents and URLs in sorted order
                // find URLS by extracting the first line in all crawled files
```

Important Notes and Clarifications:

- QueryNodes include all of the words until the OR or EOF
- An OR command will always create a new QueryNode and start adding words to that node until another OR command or EOF is reached. Thurs, a search such as OR OR OR OR will produce 4 empty QueryNodes
- AND commands are simply ignored; words are continuously added into the QueryNodes until an OR command is found
- If any word is not found, the QueryNode becomes invalidated (num_words set to 0); we then loop through all of the words until the next OR or EOF is reached

INDEXER IMPLEMENTATION SPEC

We will be describing prototypes, functions, and data structures in detail.

The header files include (1) Constants and macros; (2) Definition of major data structures.

1. DATA STRUCTURES AND VARIABLES

- **InvertedIndex:** the InvertedIndex structure is to replicate the information that is provided from the index.dat file

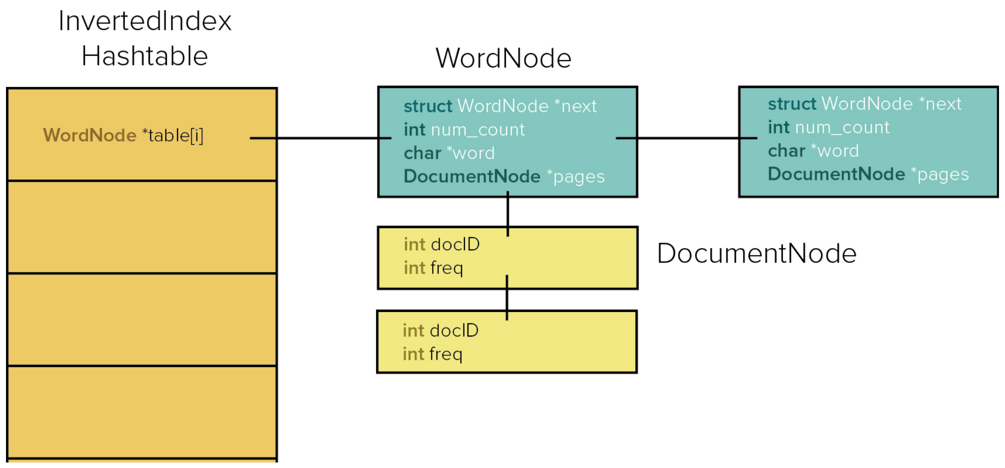
  -
    ```
    typedef struct DocumentNode {
            struct DocumentNode *next;
            int doc_id;
            int freq;
    } DocumentNode;
    ```

  -
    ```
    typedef struct WordNode {
            struct WordNode *next;
            int num_count;
            char *word;
            DocumentNode *pages;
    } WordNode;
    ```

  -
    ```
    typedef struct InvertedIndex {
            WordNode *table[MAX_HASH_SLOT];
    } InvertedIndex;
    ```

  InvertedIndex index;

VISUALIZATION OF INVERTEDINDEX



The InvertedIndex is not altered throughout the entirety of the program. The main purpose for the index is to provide the necessary information as the search is being processed. Any copies of the information will be made in shallow copies to preserve data in the InvertedIndex.

- **QueryNode:** The QueryNode is used to store all of the valid (found) words from the user entered query. For example, if the search is “cat AND dog OR cookies”…
  - QueryNode[0] = cat; dog
  - QueryNode[1] = cookies
- Elements in QueryNodes include…
  - **typedef struct QueryNode {**

```
            struct WordNode *word;
            int num_words;
            struct DocumentNode *rel_docs;
            struct QueryNode *next;
    } QueryNode;
```

2. PROTOTYPE DEFINITIONS

QUERY.C FUNCTIONS

```
/*
 * Creates a new QueryNode
 * input: void
 * output: QueryNode*
 */
```

QueryNode* makeQNode()

```
/*
 * gets the query input from the user
 * the function prompts the user enters a string of words; we then perform a check to see if
 * it surpasses the max number of words allowed.
 * input: none (prompts user within function)
 * output: query string entered by user
 */
```

char* GetQuery()

```
/*
 * Parse query and group appropriate words into queryNodes
 * input: user entered query, index generated from index.dat
 * output: pointer to the first QueryNode
 *         QueryNodes contain all of the user inputs parsed appropriately
 */
```

QueryNode *getQueryStructure(char *query, InvertedIndex *index)

```
/*
 * function to deal with AND operator; gets all of the documents that include all of the words
in the QueryNode
 * this is the more general Intersect function from query.c; it handles intersections for all of
the QueryNodes
 * there is a more specific help function called IntersectList (found in index.c) which is called
 * input: *head, pointer to first QueryNode
 * output: *queryNode with updated relevant document lists
 */
```

QueryNode *Intersect(QueryNode *head)

```
/*
 * gets unions of all of the words (find all of the documents that are valid for each queryNode
and aggregates them together)
```

 * inputs: pointer to head of queryNode, int flag which marks whether relevant documents have been found
 * output: linked list of document nodes that includes all of the relevant documents for all QueryNodes
 */
DocumentNode *Unionize(QueryNode *curr_QNode, int *flag)

/*
 * prints all of the document IDs and URLs
 * URLs are extracted from the first line of all of the previously crawled files
 * input:
 * doc_count: total number of documents
 * *path: the appropriate path to the directory which includes the crawled files
 * **final_sort: array of pointers to DocumentNodes
 * output: prints all of the information within function, returns void
 */
void printURL(int doc_count, char *path, DocumentNode **final_sort)

/*
 * free all of the nodes in the queryNode structure
 * inputs: pointer to queryNode head
 * outputs: none
 */
void freeQueryNode(QueryNode *query)

<u>INDEX.C FUNCTIONS</u>
// Majority of functions are the same as those in indexer
// New functions include…

/*
 * performs a shallow copy of a wordNode in order to better manipulate
 * word->next without interrupting links in the InvertedIndex hashtable.
 * input: original wordNode
 * output: shallow copy of the wordNode
 */
WordNode* copyWord(WordNode *original)

/*
 * performs a shallow copy; copies over content of a DocumentNode
 * input: original DocumentNode
 * output: copy of DocumentNode
 */
DocumentNode* copyDoc(DocumentNode *original)

/*
 * Essentially function for AND operator
 * input: two lists of documentNodes
 * output: documentNode with all of the intersections between the two lists
 */
DocumentNode* IntersectList(DocumentNode *list1, DocumentNode *list2)

/*
 * Function for union (OR)
 * inputs: two lists of documents (first list generally will remain static; set to current "final list")
 * outputs: documentNode with all of the relevant documents
 */
DocumentNode* UnionList (DocumentNode* list1, DocumentNode* list2)

WEB.C FUNCTIONS
// No changes from provided functions

FILE.C FUNCTIONS
// No changes from provided functions.

1. **MACROS**
   #define MAX_HASH_SLOT 10000

   // number of slots for InvertedIndex hashtable

***** ERROR CONDITIONS *****

QueryEngine checks the following error conditions...

1. **Proper Arguments:** must have 3 arguments
2. **Proper Directories:** directory must be existent and have valid files

***** TESTING *****

Tested by running different conditions. Comprehensive testing done in QEBATS.sh (refer to BATSlog_KD for testing log).

Conditions tested include...
1. Wrong number of inputs
2. Invalid file
3. Invalid directory
4. Empty directory
5. User input = "enter"
6. Inputs not found
7. Valid inputs AND case
8. Valid inputs AND OR case
9. Valid inputs, multiple OR cases
10. String greater than 1000 characters
11. Valid inputs on depth2
12. Valid inputs, but index.dat and crawled directory depths do not match

UnitTestingQE.c performs a unit test on multiple essential functions found in Query.c. The tests perform a number of different conditions.

Functions tested are...
1. getQueryStructure
2. Intersect
3. Unionize
4. PrintURLs