Documentation of Query

Table of Contents

Query Design Specifications

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.

(1) Input

Command Input

Provide 3 arguments

./query [INDEX FILENAME] [DIRECTORY WHERE THE DATA FILES ARE]

Example command input

./query ../../lab5/indexer/index.dat ../../lab5/indexer/dir/

Input Details:

[INDEX FILENAME]
Requirement: Must be able to read file path given.
Usage: The query engine needs to inform the user if the index file cannot be opened and also if it is has the incorrect format.

[DIRECTORY WHERE THE DATA FILES ARE]
Requirement: The directory must be a directory. The files must conform to the specifications of crawler, i.e. numbered with format URL (enter) depth (enter) html.
Usage: The query engine must inform the user if the directory exists.

Queries are read by user input – should end program with CTRL+ D.
Queries can are up to 1000 characters long and are in this format:
"word AND word2 OR word …" / "word word word" (all AND)

(2) Output

The query program will the list of results, returning the document with the highest number of matches to the query string, then the second highest, and so on…

example:
QUERY>: noodles

Document ID: 1380 URL: http://old-www.cs.dartmouth.edu/~cs50/tse/wiki/Mike_Remlinger.html
Document ID: 1507 URL: http://old-www.cs.dartmouth.edu/~cs50/tse/wiki/The_College_of_William_%26_Mary.html
Document ID: 257 URL: http://old-www.cs.dartmouth.edu/~cs50/tse/wiki/Nutrition.html

(3) Data Flow

The index file specified by the first argument, [INDEX FILENAME], is loaded as an index. Then for each query written by the user, it is read from left to right one word at a time. The program will keep track of two lists: of results to be unioned with ORs, and a list of results (list of DocNodes) to be intersected with ANDs. Each word besides OR and AND is checked in the index and returns a list (if the word is found), and depending on whether an OR, a word, or an AND preceded it, the two lists will be updated.
- If an OR preceded this word, the current list to be intersected with (called tmp_list) is flushed, i.e. unioned with the current list of results to be unioned with (called search_results), and a new tmp_list is created from the list retrieved from the index of the search word. When unioning two lists together, the program goes through both lists and if there are any documents in both lists, a new document will take the maximum frequency of the documents in both lists and added to the new, unioned list.
- If an AND preceded this word, the current list to be intersected with (called tmp list) is intersected with the list retrieved from the index of the search word. When intersecting two lists together, the program goes through each document in one list and checks if it is in the other list. If it is, it adds the frequencies of the document in each list and a new document is created with that frequency and added to the new, intersected list.
After going through the entire query, the list to be unioned with (search_results) and the remaining list to be intersected with (tmp_list) is unioned to get the final list of documents. Then this is ordered by frequency and printed out as the search result. Then the program restarts and waits for a new query.

(4) Data Structures

hashTable – the inverted index containing all the words and their frequencies across documents.

- the hashTable points to a hashTableNode, which has a word as a key and a wordNode as a value (in this case).

wordNode – a pointer to a linked list of docNodes that the word(s) occur in.

docNode – stores a document ID and the frequency of the word, and a pointer to the next docNode, which has information of another document (file) that has some frequency of the same word.

(5) Pseudo Code

```
//check arguments
//if 3 arguments given:
        // check if 1st arg can be read and 2nd arg is a directory, and inform user
        if neither of these are valid
        // make indexer from 1st arg
```

```
// while something from standard input:
        // read query from left to right
                // initialize list of search results and temporary list results
                // for every word: check if OR or AND
                        // if so: mark "OR" or "AND" and continue.
                        // else:
                                // if previously marked "OR": update search
                                results by unionizing search results and
                                temporary list
                                // if marked "AND": update temporary list
                                results by intersecting temporary list and
                                the list results for the word
                                // else, first word of the query – load
                                temporary list results for the word.

        // when run out of words in query, unionize temporary list results
        and search results

        // order search results in terms of frequency
        // print out each document in order of its rank (relevance)

// get ready for next query from standard input
```

Query Implementation Specifications

In this query specification we define the prototypes and data structures
in detail. It defines the data structures in the design spec.
The header file of the library libtseutil.h includes all the data structures and prototypes we will use for the Tiny Search Engine. You can find individual source codes in the lab6/util/ directory. They are summarized below.
Hashtable.h has Hashtable structures and functions
Web.h normalizes the word (lowercase)
File.h has functions with files and directories
Common_index.h has WordNode and DocNode structures and functions
Common_query.h has functions dealing with querying specifically.

(1) DATA STRUCTURES AND VARIABLES

```
1. General HashTable Structure

typedef struct HashTableNode {

    char *key;                                  // key

    void *value;                                 //in this case, WordNode *

    struct HashTableNode *next;                 // pointer to next node
```

```
} HashTableNode;


typedef struct HashTable {
    HashTableNode *table[MAX_HASH_SLOT];     // array of pointers to no
des
} HashTable;
2. WordNode and DocNodes
typedef struct DocNode {
        int docID;                      // document word is found in
        int freq;                       // frequency of word in the doc
        struct DocNode *next;               // pointer to next node
} DocNode;


typedef struct WordNode {
        DocNode *head;                  // points to first docnode in lis
t
} WordNode;
/*  * These structs should all be called dynamically, see prototypes.
*/
```

(2) PROTOTYPE DEFINITIONS

```
/* HashTable Functions */


// Implementation details can be found at:
// http://www.burtleburtle.net/bob/hash/doobs.html
unsigned long JenkinsHash(const char *str, unsigned long mod);


//creates hash table dynamically
HashTable *init_hash();
```

```
// returns 1 if key is in unique, 0 if not unique (key already in the h
ashtable)
int unique_hash(char *key, HashTable *ht);


// get value of a hashTableNode based on the key
// returns NULL if not found
void * get_value(char *key, HashTable *ht);
// adds a hash table node to the hash table
// @key: key of hashTableNode to be added
// @value: value of hashTableNode to be added
// returns 0 if error, 1 if success
int add_hash(char *key, void * value, HashTable *ht);


// frees hashtable
int free_table(HashTable *ht);


/* WordNode and DocNode Functions */
// create WordNode dynamically
WordNode * init_list();


// get length of the list (number of DocNodes)
int num_docs(WordNode * list);


//adds a DocNode to an existing WordNode list
// @ID: the filename (id) of the current document
// @freq: the frequency of the word in the document
// @list: the list of documents the document will be added to
int add_doc(int ID, int freq, WordNode* list);
```

```
//checks if a word has been seen before in the document.
// @ID: the filename (id) of the current document
// @list: the list of documents the word has been in
//returns 1 if the word has been seen before, returns 0 if not. this me
ans the a new docNode needs to be created for this word.
int in_doc(int ID, WordNode *list);

// returns a document node specified by its docID in the list. returns
NULL if not present in list.
DocNode *get_node(int ID, WordNode *list);

// merge sort code, borrowed from
// http://c.happycodings.com/sorting-searching/code10.html
// returns the head of the list.
DocNode *mergesort(DocNode *head);
DocNode *merge(DocNode *head_one, DocNode *head_two);

// frees list
int free_list(WordNode* list);

/* Web Functions */
// NormalizeWord - lowercases all the alphabetic characters in word
void NormalizeWord(char *word);

/* File Functions */
// IsFile - determine if path is a file
int IsFile(const char *path);

//IsDir - determine if path is a directory
```

```
int IsDir(const char *path);
```

(3) CONSTANTS

```
// Make the hash large in comparison to the number of words  Access is
O(1). Fast!

#define MAX_HASH_SLOT 10000

// Sets a max buffer for a line using fgets when reading in any file (H
TML or index).

#define BUF_MAX 100000

// Sets a max buffer for a query from user input.

#define QUERY_MAX 1000
```

Special Considerations and Tests

- If a query ends with an AND or OR, it is just ignored.
    - e.g. "list AND include AND" and "list AND include OR" have the same output as "list AND include"
- If program ends with CTRL + C, this works but results in memory leaks because the program exits prematurely. (still reachable)
- If the program gets an empty input, returns "No input specified".
- The program exits, if given too long of input (QUERY_MAX: default 1000 char).
-

Tests: all written in QEBATS.sh, will output an queryTestLog.date

- QEBATS calls query_test.c to do some unit testing for the query engine.
- Unit tests done on:
    - Intersection
    - Unionize
    - Mergesort – in common_index.c
    - Find index
    - GetURL
    - ProcessQuery
- Besides mergesort, these functions are all in common_query.c/h
- Test invalid inputs:
    - Invalid number of inputs
    - Invalid file
    - Invalid directory