Documentation for TinySearchEngine: Indexer

Chua Zheng Fu Edrei

Version 1.0: last update: 9 August 2015

I Introduction

We will discuss the Requirement Specifications and the command line execution of Indexer.

I.I Requirement Specification

Indexer is designed with the following requirement specifications:

The indexer shall do the following for each file/document created by crawler in the [$TARGET\_DIRECTORY$]

1. load the document from the file system, noting the unique document identifier (filename);
2. use the provided function to parse the document discarding the HTML tags in order to count the occurrences of each word found in the document; and
3. store each word and its frequency of occurrence in that document in an inverted index data structure so that the information can be easily retrieved.

After all the documents have been processed, the indexer will save the index information to a file. This file will be used by the query engine.

The index will need to be restored into the data structures for the query engine. To test the reconstruction of the index, indexer should have a test mode as follows:

1. create the index from html files (described above)
2. write the index to a file to store it, e.g., index.dat
3. reinitialize the index data structures,
4. read the index in from the file (i.e., index.dat in this case) and recreate the index data structures ; and write the index to a different file (e.g., new_index.dat) for comparison with the first index written out (e.g., index.dat). The two files should be identical.

I.II Command line execution

The design takes the following 2 options, described in the listing below

Listing 1: Command line option

```
/* command line option 1: default */

 ./indexer [TARGET_DIRECTORY] [RESULTS FILE NAME]

 // Example command input:
 // ./indexer ../../data/ index.dat

 /* command line option 2: testing */
./indexer [TARGET_DIRECTORY] [RESULTS FILENAME] [REWRITEN FILENAME]

//Example command input:
//./indexer ../../data/ index.dat new_index.dat
```

Option 1 is the regular option used to create an inverted index [$RESULTSFILENAME$]. Option 2 is the testing option used to test the reconstruction of an inverted index from the file [$RESULTSFILENAME$]. Both options extract files from [$TARGET\_DIRECTORY$].

[$RESULTSFILENAME$] and [$REWRITENFILENAME$] will then be saved in [$TARGET\_DIRECTORY$] as indicated in the lab instructions.

II Design Specification

In the design specification, we will talk about the Top level module (which includes the input, output and data flow specifications), the data structure that can be used and the pseudocode for Indexer.

II.I Top level module

In the following Design Module we describe the input, data flow, and output specification for the indexer module.

Now, lets look at how we define the inputs, outputs and the major data flow through the module:

Input

Input option 1:
./indexer [TARGET_DIRECTORY] [RESULTS FILE NAME]

Example command input:
./indexer ../../data/ index.dat

Input option 2:
./indexer [TARGET_DIRECTORY] [RESULTS FILENAME] [REWRITEN FILENAME]

Example command input:
./indexer ../../data/ index.dat new_index.dat

[$TARGET\_DIRECTORY$] ../../data
Requirements: The directory must already by created by user and contain html files of the format specified in the Special Considerations section below
Usage: The indexer needs to inform the user if the directory cannot be found, is invalid, files cannot be opened or the files are not in the correct format

[$RESULTSFILENAME$] [$REWRITENFILENAME$] index.dat new_index.dat
Requirements: both file names should be different so that the user doesn't overwrite the file
Usage: The indexer needs to inform the user if both file names are identical

Output

1) For command line option 1, output will be [RESULTS FILE NAME] with an inverted index of words in the html files stored in [TARGET_DIRECTORY] for first command line option
2) For command line option 2, output will be the above and [REWRITEN FILENAME] with an inverted index of words reconstructed from [RESULTS FILE NAME]

Note: [RESULTS FILE NAME] and [REWRITEN FILENAME] will be saved in [TARGET_DIRECTORY] as indicated in the lab instructions

The format of [RESULTS FILE NAME] and [REWRITEN FILENAME] will take the form of an inverted index, as described below:

cat 2 3 4 7 6
moose 1 5 7...

Let us consider the entry cat 2 2 3 4 5 . cat is the word, the number 2 is the number of documents containing the word cat , the following 3 4 means the document with identifier 3 has 4 occurrences of cat in it; the following 7 6 means that the document with identifier 7 has 6 occurrences of cat in it.

A sample of the output could be:

Listing 2: Sample output

```
address 19 2 1 8 4 21 1 22 1 37 1 38 1 54 1 61 1 72 2 79 3 84 1 85 1
88 1 90 1 109 2 126 1 153 1 159 1 188 1
addressed 2 138 1 140 1
addresses 1 101 1
adelfio 3 33 1 112 1 138 1
adelson 1 104 1
adhere 2 11 1 142 4
adhoc 1 88 1
adirondacks 1 191 1
aditikabir 1 51 1
```

Data flow

The html files in the directory will be opened and parsed to search for words. Next, the words together with the document ID of the files will be inserted into the data structure (an inverted index). The data structure is essentially a hash table with each of the node being a linked list, as will be described in the next section. Keep doing this while there are words to be parsed in the file. If the word has already been found in the same document, increment the document frequency associated with the document ID for that particular word. Repeat this cycle while there are files in the directory. Then print the data structure into a file.

To test the indexer, read the inverted index file and save the word, with the corresponding document ID and frequency into a data structure, and then print the data structure into a new file.

II.II Data Structure

The data structure are described below:

directory - a string (char*) containing the path of the directory
result_file - a string (char*) specifying the name of the result file
rewritten_file - a string (char*) specifying the name of the file to be rewritten

We will use a hash table to store the words.
HashTable has a bunch of WordNode*

We also need a data structure for each word, storing the word (char*), the number of documents with the word (int), a pointer to the next word in (WordNode*) and a pointer to the documents with the word (Docu-

mentNode*).

WordNode has the word, a next pointer, the document frequency of the word and a pointer to a DocumentNode.

The DocumentNode is a single linked list with a next pointer, the document ID, and the frequency of the word in the document.

The detailed data structure will be described in the functional specification section.

II.III Indexer Pseudocode

Here, we will described the pseudocode for indexer. The pseudocode for each function will be described in the Functional Specification section.

Listing 3: Pseudocode fro Indexer

```
/**
 * Pseudocode:         1. Check input parameters
 *                             a. Check number of arguments
 *                             b. Check if directory is valid
 *                       2. Assign variable names to input parameter for readability of code
 *                       3. Fill up array of filenames using the function GetFilenamesInDir
 *                       4. Create Inverted Index
 *                             a. get document ID using function GetDocumentId
 *                             b. combined filename with directory path using function getFileName
 *                             c. load the html file into a string using function LoadDocument
 *                             d. get the words in the html string using function GetNextWord
 *                             e. normalise the words using function NormalizeWord
 *                             f. insert the normalised word into the inverted index using
 *                                function HashTableInsert
 *                             g. free memory
 *                       5. Save Inverted Index to file using function SaveIndexToFile
 *                       6. Reconstruct inverted index from file if applicable (command line option 2)
 *                             a. read the file and create the index using function ReadFile
 *                             b. save the reconstructed index to file using function
 *                                SaveIndexToFile
 **/
```

III Functional Specification

The functional specification section will detailed the data structure and functions employed in the implementation of the indexer.

III.I Data Structure

The data structure outined in the previous section will be implemented as shown below:

Listing 4: Data structure: WordNode

```
typedef struct WordNode {
  struct WordNode *next;                  // pointer to the next word (for collisions)
  char *word;                             // the word
  int docfreq;                            // number of documents with the word
```

```
  struct DocumentNode *page;              // pointer to the first element of the page list.
} WordNode;
```

Listing 5: Data structure: DocumentNode

```
typedef struct DocumentNode {
  struct DocumentNode *next;      // pointer to the next member of the list.
  int doc_id;                     // document identifier
  int freq;                       // number of occurrences of the word
} DocumentNode;
```

Listing 6: Data structure: Hash Table

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

III.II Files in the package and function prototype

The files contained in the package are described below. For the .c and .h file, function prototypes are also included.

Listing 7: Files in package lab5/indexer

```
Files in package lab5/indexer

1) lab5/indexer/src
       a) indexer.c
              functions:     char* getFileName(char* fileName, char* dir);
                             char* LoadDocument(char* fileName);
                             int GetDocumentId (char* fileName);
                             void SaveIndexToFile(HashTable* index, char* filename);
                             int ReadFile(HashTable* index, char *file);

       b) file.c and file.h
              functions:     int IsDir(const char *path);
                             int IsFile(const char *path);
                             int GetFilenamesInDir(const char *dir, char ***filenames);

       c) list.c and list.h
              functions:     int ListInsert(int doc, DocumentNode** dnode);
                             void CleanUpList(DocumentNode* head);

       d) hashtable.c and hashtable.h
              functions:     unsigned long JenkinsHash(const char *str, unsigned long mod);
                             int HashTableInsert(char* word, int doc, HashTable* index);
                             void InitialiseHashTable(HashTable* hashTable);
                             void CleanUpHash(HashTable* hashtable);

       e) web.c and web.h
              functions:     int GetNextWord(const char* doc, int pos, char **word);
                             void NormalizeWord(char *word);

       f) common.h                                    // common functionality

2) BATS.sh                                            // Automated testing
```

```
3) README                                          // README for indexer

4) Makefile                                        // make and make clean functionality

5) indexer_doc.pdf                                 // Documentation

6) Sun_Aug_09_11:53:28_2015.log                    // Sample Test log from running BATS.sh

7) index.dat                                       // inverted index file generated
```

III.III Function description

Description of each function will be outlined here:

Functions in indexer.c

Listing 8: Functions in indexer.c

```
/*
 * getFileName: function to get file name combined with directory path
 * @fileName: the file name
 * @dir: the directory to store the file
 *
 * Returns the new file name (file name combined with directory path)
 * Special consideration: It is the caller's responsibility to free the filename
 *
 * Pseudocode: 1. If directory ends with a '\', append filename with directory path
 *                          2. If directory does not end with a '\', append filename with directory
 *                                 path and include a '\' in between
 *
 */
char* getFileName(char *fileName, char* dir);

/*
 * LoadDocument: function to load the html file into a string
 * @fileName: the file name of the html file
 *
 * Returns a string with the html file loaded
 * Special consideration: It is the caller's responsibility to free string
 *
 * Pseudocode: 1. Check if file exists
 *                          2. Check if file is valid
 *                          3. Allocate memory to string
 *                          4. Copy the file into string
 *
 */
char* LoadDocument(char *fileName);

/*
 * GetDocumentId: function to get the document ID from the filename
 * @fileName: the file name of the html file
 *
 * Returns an integer that is the document ID
 * Assumption: the file name is an integer
 *
```

```
 * Pseudocode: 1. Check if file name is an integer
 *                          2. Returns the integer as the document id if 1. is true
 *
 */
int GetDocumentId (char *fileName);

/*
 * SaveIndexToFile: function to convert the inverted index into a file
 * @index: the inverted index
 * @fileName: the file name of the html file
 *
 * Prints a file (filename) with the inverted index into the target directory
 *
 * Pseudocode: for all rows in the hash table{
 *                                  while there are still WordNodes in the row{
 *                                         print the word and the number of documents with the word
 *                                         print the document id and frequency while there are
     DocumentNodes linked to the WordNode
 *                                  }
 *                                  print a new line
 *                          }
 *                          clean up
 *
 */
void SaveIndexToFile(HashTable* index, char* filename);

/*
 * ReadFile: function to read the file and reconstruct the inverted index
 * @index: the reconstructed inverted index (should be NULL when passed into the function)
 * @file: the file name of the file with the inverted index
 *
 * Returns the reconstructed inverted index
 * Special consideration: index should be declared NULL before being passed into the function
 * Assumptions: the file is in the format specified in the documentation
 *
 * Pseudocode: 1. Open the file and check if it exist
 *                          2. Scan for the word, the number of documents with the word, the document
     id
 *                                  and the frequency of word occurrences
 *                          3. Input the parameters to reconstruct the index using HashTableInsert
 *
 */
int ReadFile(HashTable* index, char *file);
```

Functions in hashtable.c and hashtable.h

Listing 9: Functions in hashtable.c and hashtable.h

```
/*
 * 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);

/*
 * HashTableInsert - function to insert words and document id into the data structure
 * @word: word to insert into the inverted indes
 * @doc: document id of the associated word
 * @index: pointer to the hashtable
 *
 * Returns the hashtable index with the updates
 * function returns 1 if the word is found for the first time;
 *            2 if the word is found in the document with id doc for the first time;
 *            0 if the word has already been found in the same document previously
 *
 * Special Consideration: It is the user responsibility to pass the hashtable to be updated
 *                                                  into the function
 *
 * Pseudocode: 1. Obtain the node in the hashtable by looking up the hash number
 *                      2. If the node is NULL, word is found for the first time
 *                                  a. Initialise word node
 *                                  b. Create DocumentNode
 *                      3. If node is not NULL
 *                                  a. search through the WordNodes
 *                                  b. if at any point the same word occur, increment the word freq
    and we are done
 *                                  c. if we reached the end of the list and the word is still not
    found, create
 *                                                  a new WordNode and DocumentNode
 */
int HashTableInsert(char* word, int doc, HashTable* index);

/*
 * InitialiseHashTable - function to initialise hash table
 * @hashTable: HashTable to be initialised
 *
 * Returns a HashTable with NULL nodes
 *
 */
void InitialiseHashTable(HashTable* hashTable);


/*
 * CleanUpHash - function to free the Hash table
 * @hashtable: pointer to the hashtable
 *
```

```
 * Pseudocode: for all rows in the hashtable
 *                                  Save the pointer to the next WordNode, and free the previous node.
 *                                  Keep doing this until the next node is a NULL pointer
 *
 */
void CleanUpHash(HashTable* hashtable);
```

Functions in list.c and list.h

Listing 10: Functions in list.c and list.h

```
/*
 * ListInsert: function to insert DocumentNode into the list
 * @doc: the document id
 * @dnode: the pointer to the head of the first DocumentNode in the WordNode
 *
 * Returns 1 if doc_id is unique, returns 0 for repeated cases
 * Special consideration: dnode should be declared and memory allocated before passing into the
    function
 *
 * Pseudocode: 1. Initialise node if the word is found for the first time
 *                           2. If not, search through the single linked list
 *                                        a. if a document node with the same id is found, increment
    the freq
 *                                        b. if no nodes with the same id can be found, create a new
    node and initialise it
 *
 */
int ListInsert(int doc, DocumentNode** dnode);


/*
 * CleanUpList - function to free the Single linked list
 * @head: the head of the single linked list
 * Special Consideration: It is the caller responsibility to ensure that the head of the
 *            linked list, and not an intermediate node, is passed into the function to prevent
 *            memory leak
 *
 * Pseudocode: Save the pointer to the next node, and free the previous node. Keep doing
 *                           this until the next node is a NULL pointer
 *
 */
void CleanUpList(DocumentNode* head);
```

Functions in web.c and web.h

Listing 11: Functions in web.c and web.h

```
/*
 * GetNextWord - returns the next word from doc[pos] into word
 * @doc: pointer to an html document
 * @pos: current position in the document
 * @word: a pointer to a c-style string, used to pass the word back out
 *
 * Returns the current position searched so far in doc; otherwise, returns < 0;
 * The doc should be a valid character buffer. The pos argument should be 0 on the
 * initial call. The word argument should be a NULL pointer. On successful parse
 * of the html in doc, word will contain a newly allocated character buffer;
 * may be NULL on failed return. The caller is responsible for free'ing this
 * memory.
 *
 * Usage example: (retrieve all words in a page)
 * int pos = 0;
 * char *word;
 * char *doc = "<ruh>Vox clamantis <roh> 3.0 < 5.0 /> in deserto.<raggy>";
 *
 * while((pos = GetNextWord(doc, pos, &word)) > 0) {
 *     // do something with word
 *     free(word);
 * }
 */
int GetNextWord(const char *doc, int pos, char **word);

/*
 * 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);
```

Functions in file.c and file.h

Listing 12: Functions in file.c and file.h

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

/*
 * GetFilenamesInDir - populates a list of filenames found in the directory dir
 * @dir: directory path to search
 * @filenames: a pointer to an array of c-style strings
 *
 * Returns the number of filenames found; otherwise, a negative number indicates
 * failure. The filenames array should not be preallocated, but will be created
 * by this function. It is the caller's responsibility to free the filenames and
 * the array containing them.
 *
 * Usage example:
 * char **filenames = NULL;
 * int num_files = 0;
 *
 * num_files = GetFilenamesInDir(".", &filenames);
 * if(num_files < 0) {
 *     // failed to get any filenames (various reasons)
 * } else {
 *     for(int i = 0; i < num_files; i++) {
 *         printf("File: %s\n", filenames[i]);
 *         free(filenames[i]);
 *     }
 *     free(filenames);
 * }
 */
int GetFilenamesInDir(const char *dir, char ***filenames);
```

IV Summary of error conditions

There are no major error conditions observed at the time of writing of this documentation. Boundary cases are tested for (and described in the Testing section). There are also no memory leaks.

V Test cases and expected results

Testing was performed using the default command line option, the testing option and the automated testing script (BATS.sh)

1) From running the default: ./indexer ../../data index.dat

Listing 13: Running indexer using default

```
$ ./indexer ../../data index.dat

Reading file: ../../data/1
Reading file: ../../data/10
Reading file: ../../data/100
Reading file: ../../data/1000
Reading file: ../../data/1001
Reading file: ../../data/1002
...
...
...
Reading file: ../../data/998
Reading file: ../../data/999
Inverted index saved in ../../data/index.dat
```

2) From running the test option: ./indexer ../../data index.dat new_index.dat

Listing 14: Running indexer using test option

```
$ ./indexer ../../data index.dat new_index.dat

Reading file: ../../data/1
Reading file: ../../data/10
Reading file: ../../data/100
Reading file: ../../data/1000
Reading file: ../../data/1001
Reading file: ../../data/1002
...
...
...
Reading file: ../../data/998
Reading file: ../../data/999
Inverted index saved in ../../data/index.dat

Reconstructed inverted index saved in ../../data/new\_index.dat
```

3) Logfile from a successful test using BATS.sh

Listing 15: Log file from BATS.sh

```
start testing indexer
rm -f *~
```

```
rm -f *#
rm -f *.o
cd ../../data; rm -f *.dat;
gcc -Wall -pedantic -std=c11 -c ./src/indexer.c ./src/list.c ./src/web.c ./src/hashtable.c ./src/file.c
gcc -Wall -pedantic -std=c11 -c ./src/indexer.c ./src/list.c ./src/web.c ./src/hashtable.c ./src/file.c
gcc -Wall -pedantic -std=c11 -o indexer indexer.o web.o list.o hashtable.o file.o -lcurl
build indexer successfully
test input parameters
test input parameters successfully
inverted index reconstructed from file successfully
Indexes have been built, read and rewritten correctly!
```

4) Valgrind test for memory leak

Listing 16: Valgrind test for memory leak

```
==19623==
==19623== HEAP SUMMARY:
==19623==    in use at exit: 0 bytes in 0 blocks
==19623==  total heap usage: 11,860,113 allocs, 11,860,113 frees, 325,059,391 bytes allocated
==19623==
==19623== All heap blocks were freed -- no leaks are possible
==19623==
==19623== For counts of detected and suppressed errors, rerun with: -v
==19623== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5) Boundary cases

Same file names for test option

Listing 17: Boundary cases

```
[edrei@tahoe]$ indexer ../../data2 index.dat index.dat
Reading file: ../../data2/1
Reading file: ../../data2/2
Reading file: ../../data2/3
Reading file: ../../data2/4
Reading file: ../../data2/5
Reading file: ../../data2/6
Reading file: ../../data2/7
Inverted index saved in ../../data2/index.dat
Error: file name for [RESULTS FILENAME] and [REWRITEN FILENAME] are the same. They should be different
```

Incorrect file format

Listing 18: Boundary cases

```
[edrei@tahoe]$ indexer ../../data2 index.dat index.dat
```

```
Reading file: ../../data2/1
Reading file: ../../data2/2
Reading file: ../../data2/3
Reading file: ../../data2/4
Reading file: ../../data2/5
Reading file: ../../data2/6
Reading file: ../../data2/7
Reading file: ../../data2/8
Error: file format is incorrect ../../data2/8
```

Directory cannot be found

Listing 19: Boundary cases

```
[edrei@tahoe]$ indexer ../../dat index.dat index.dat
Error: Directory cannot be found
```

VI Special considerations

1) It is assumed that the html files have integer-valued names

2) It is assumed that the html files come in the below format:
url
page id
html doc

The first 2 lines will be ignored in the parsing of the words for the inverted index

3) The user should changed the directory path under datapath for BATS.sh and under UTILDIR for Makefile to reflect the directory path used. The default is "../../data", which assumes that there is a directory "data" two branch above the current directory that contains the html files

VII References

Dartmouth Canvas lab instruction for Indexer. Accessed on 9 August 2015. Retrieved from:
https://canvas.dartmouth.edu/courses/9618/assignments/46576

Dartmouth Canvas lecture on Indexer Design. Accessed on 9 August 2015. Retrieved from:
https://canvas.dartmouth.edu/courses/9618/pages/lecture-16-tinysearchengine-indexer-design

Dartmouth Canvas template on Design and Implementation specifications for Crawler. Accessed on 9 August 2015. Retrieved from:
https://canvas.dartmouth.edu/courses/9618/pages/lecture-13-tinysearchengine-crawler-design