Indexer Design Spec

In the following Design Module we describe the input, data flow, and output specification for the indexer module. The pseudo code for the indexer module is also given.

The following are included in this Design Spec:

(1) Input: Any inputs to the module
(2) Output: Any outputs of the module
(3) Data Flow: Any data flow through the module
(4) Data Structures: Major data structures used by the module
(5) Pseudo Code: Pseudo code description of the module.

(1) *Input*

Command Input

```
./indexer [TARGET_DIRECTORY] [FILE]
./indexer [TARGET_DIRECTORY] [FILE] [FILE_2]
```

Example command input

```
indexer ~/cs50/labs/lab5/crawler/data index.dat
```

[TARGET_DIRECTORY] ~/cs50/labs/lab5/crawler/data
Requirement: The directory must exist, and contain all the crawled webpage files.
Usage: The indexer needs to inform the user if the directory cannot be found, or is empty.

[FILE] index.dat
Requirement: The file must exist.
Usage: The indexer needs to inform the user if the file cannot be found, or if it is not empty.

```
indexer ~/cs50/labs/lab5/crawler/data index.dat new_index.dat
```

* First two arguments are the same as above.

[FILE_2] new_index.dat
Requirement: The file must exist.
Usage: The indexer needs to inform the user if the file cannot be found, or if it is not empty.

(2) *Output*

 From all the files in the [TARGET DIRECTORY], the indexer will build an Inverted Index of words to documents, and output this information to [FILE]. If [FILE2] is passed, then the indexer reads from [FILE] to recreate the Inverted Index, and write the information to [FILE2], to test if the index storage is working correctly or not.

(3) *Data Flow*

The HTML in each file in [TARGET_DIRECTORY] is stored into a character array. Then, each word in each string is checked to see whether it is already in the InvertedIndex or not. If not, the word is added to the InvertedIndex. If the word is in the InvertedIndex, then the program checks whether the specific document the search word came from is associated with the word in the InvertedIndex. If it is, then the frequency of the word from the specific document is incremented. Otherwise, the word in the InvertedIndex is associated with the specific document and frequency of occurrence in this document is set to one.

We continue this process as long as there are words left from each HTML string for each [TARGET_DIRECTORY] file.

Afterwards, the InvertedIndex is saved to [FILE], where each line represents information about each unique word, such as the number of documents it appears in, which document, and the frequency.

If [FILE2] is passed, then the InvertedIndex is recreated from [FILE], and saved to [FILE2].

(4) *Data Structures*

WordNode – holds information about a word, such as in which document it appears, and how frequently.

DocumentNode – holds information about a document that a word appears in.

InvertedIndex – holds a table of all the words, where each word points to a list of documents that it appears in. This is essentially a HashTable.

(5) *Indexer* Pseudocode

```
 // check command line arguments
        // Inform the user if arguments are not present or invalid

    // initialize data structures / variables

    // loop through each file in [TARGET_DIRECTORY]
        // get the html from the file and store into a string
        // get the document_id from the file

        // get next word from the html string
        // update the InvertedIndex with regards to the word

        // free resources

    // free resources

    // save the InvertedIndex to [FILE]
    // check that [FILE2] has been passed
        // if true, read [FILE] and recreate the InvertedIndex
        // save the new InvertedIndex to [FILE2]
        // free resources

    // free resources
```

Indexer Functionality

Below are descriptions of all functions used to build the Indexer.

```
/*
 * LoadDocument - Load a document into a string.
 * @file_name: file to be loaded.
 *
 * Returns the string of html if successful.
 * If not successful, returns NULL.
 *
 * Pseudocode:
 *     1. Get the full filename, complete with the directory.
 *     2. Set the position of the file to the start of the third line.
 *     3. Read each line and add to a string variable.
 */
char *LoadDocument(char *);
```

```
/*
 * GetDocumentId - gets the unique doc_id of each file.
 * @file_name: file to get the doc_id from.
 *
 * Returns the doc_id, which is an integer.
 *
 * Assumptions:
 *     1. Passed file_name is not NULL.
 *
 * Pseudocode:
 *     1. Read in file_name as an integer.
 *     2. Return this integer as the doc_id.
 */
int GetDocumentId (char *);


/*
 * UpdateIndex - updates the InvertedIndex for each word of a document.
 * @word: word to be added to the InvertedIndex.
 * @documentId: document the word comes from.
 * @Index: pointer to the InvertedIndex.
 *
 * Returns 1 if successful, 0 if not successful.
 *
 * Assumptions:
 *     1. InvertedIndex has been initialized.
 *
 * Pseudocode:
 *     1. Check that the args are valid.
 *     2. Add the word to the Index.
 */
int UpdateIndex(char *, int, HashTable *);


/*
 * SaveIndexToFile - outputs the InvertedIndex to a file.
 * @Index: pointer to the InvertedIndex.
 * @file_name: file to be written to.
 *
 * Returns 1 if successful, 0 if not successful.
 *
 * Assumptions:
 *     1. InvertedIndex has been initialized.
 *
 * Pseudocode:
 *     1. Check that the args are valid.
 *     2. Add the word to the Index.
 */
int SaveIndexToFile(HashTable *, char *);


/*
 * 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 *);


/*
 * AddWord - adds a word to the InvertedIndex.
 * @WORD: word to be added.
 * @doc_ID: doc_ID of the doc in which the word was found.
 * @Index: pointer to the InvertedIndex.
 *
 * Returns 1 if successful, 0 if not.
 *
 * Assumptions:
 *     1. InvertedIndex has been initialized.
 *
 * Pseudocode:
 *     1. Compute the hash code.
 *     2. Check if the word already exists in the InvertedIndex.
 *     3. If it does, then see if there is a matching DocumentNode. Increment frequency if
found.
 *     4. If there is no matching DocumentNode, create a new one and add to the
InvertedIndex.
 *     5. If the word does not exist in the InvertedIndex, create a new WordNode and add
to the InvertedIndex.
 *
 */
int AddWord(char *, int, 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 *);
```

Indexer Error Conditions / Boundary Cases
----

1. This program checks that there are either 2 or 3 arguments passed.
2. This program checks that the [TARGET_DIRECTORY] exists.
3. This program checks that the [TARGET_DIRECTORY] is not empty.
4. This program checks that [FILE] and [FILE2] files are valid (ie, they exist).
5. This program checks that [FILE] and [FILE2] files are empty.
6. This program checks for sufficient memory space when memory is allocated.

Indexer Test Cases

Below are some test cases and the expected output.

1. Case of invalid number of arguments (not 2 or 3)

```
Input: ./indexer ~/cs50/labs/lab5/crawler/data
Output: Please input the correct number of arguments.
```

2. Case of nonexistent TARGET_DIRECTORY.
   For this test, ./testing was a nonexistent directory.

```
Input: ./indexer ./testing index.dat
Output: Please input a valid TARGET_DIRECTORY.
```

3. Case of empty TARGET_DIRECTORY.
   For this test, ./data was an empty directory.

```
Input: ./indexer ./data test.dat
Output: TARGET_DIRECTORY is empty.
```

4. Case of nonexistent index file(s).
   For this test, test.dat existed, but new_test.dat did not.

```
Input: ./indexer ~/cs50/labs/lab5/crawler/data test.dat new_test.dat
Output: Please input a valid new_index.dat file.
```

5. Case of non-empty index file(s).
   For non-empty index files, program prints out warning messages, and continues.

```
Output: index.dat is not empty. The contents will be overwritten.
        new_index.dat is not empty. The contents will be overwritten.
```