Indexer Design Spec

(1) *Input*

Command Input without testing:

```
./indexer [TARGET_DIRECTORY] [INDEX_FILE]
```

Example command input without testing:

```
./crawler ./data index.dat
```

Command Input with testing:

```
./indexer [TARGET_DIRECTORY] [INDEX_FILE] [INDEX_FILE] [RELOAD_FILE]
```

Example command input with testing:

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

[TARGET_DIRECTORY] ./data
Requirement: Must be an existing directory.
Usage: The indexer needs to inform the user if the directory cannot be found

[INDEX_FILE] index.dat
Requirement: Must be a non-existing file
Usage: The indexer needs to warn the user if the file already exists.

[RELOAD_FILE] new_index.dat
Requirement: Must be a non-existing file
Usage: The indexer needs to warn the user if the file already exists.

(2) *Output*
Output without testing:
The indexer will read through each file in the [TARGET_DIRECTORY] that has an integer as its name, skipping the first two lines of each. Information about each word, the documents it is included in, and the frequency in each of those documents will be written to the [INDEX_FILE]. Each line of the file will have the following format:
[word] [number of documents the word occurred in] [docID] [frequency in the doc] [docId] [frequency in the doc]…

Output with testing:
The [INDEX_FILE] will be produced in the exact same way as outlined above. It will then be read in order to recreate the data structure that produced it. The information from the new data structure will the written to [RELOAD_FILE]. If the test runs successfully, [INDEX_FILE] and [RELOAD_FILE] should be identical when sorted.

(3) *Data Flow*
The names of the files in the [TARGET_DIRECTORY] are retrieved.

Each file is opened, read, stored in a character buffer, and closed.

The character buffer will contain html if the files being indexed are produced from crawler. The html will be parsed for words.

As each word is encountered, it will be added to a HashTable data structure. The HashTable data structure uses the Jenkins Hash function to hash the word to its appropriate slot. A WordNode and a DocumentNode is created, or updated depending on whether or not the word-document pairing has been encountered before. WordNodes are stored as data in HashTableNodes.

Collisions are handled by chaining WordNodes together in a linked list. Each WordNode contains a linked list of DocumentNodes that represent each unique document the word is found in. The DocumentNodes keep track of how many times the word occurred in a given document.

Once the HashTable is completely constructed (there are no more files to process), the information it stores will be processed and written to the provided file. This is done by iterating through each slot in the HashTable and checking if anything has been hashed there. If there is, the linked list of WordNodes is traversed, and each list of DocumentNodes is traversed in order to get the appropriate information to write to the file.

If testing is being done (4 arguments are provided), the files produced above will be read back in. Each line is read, and a new HashTable is updated accordingly. Once every line has been processed, and the HashTable has been successfully constructed, the same process of writing the information to a file that was outlined above will take place.

(4) *Data Structures*

Index — HashTable struct that stores all of the data

word — pointer to the current word

WordNode structure holds the word, a pointer to the next WordNode, a pointer to the first DocumentNode in the list of DocumentNode

DocumentNode structure holds the document ID (the integer used as the file name), the frequency of the word that is linked to the DocumentNode in the document, a pointer to the next DocumentNode in a linked list of DocumentNodes

(5) *Indexer Pseudocode*

```
// check command line arguments

// initialize data structures

// get the file names from the target directory

// look at each file in the directory

    // load the file into a string

    // get the document ID of the file (and make sure its an integer)

    // while there are words to be read in the string

        // normalize the word

        // update the index with the current word, and its document ID

// free resources
```

Functions used:

char *LoadDocument(char *fileName) — loads HTML document from file. receives the name of the file to be loaded and returns a string containing the loaded document

int GetDocumentId(char *fileName) — gets the integer file name of the file provided and converts it from a string to an int. returns -1 if the file name is not an integer

int GetNextWord(const char *doc, int pos, char **word) — parses the string containing the loaded document skipping the HTML tags and returning the next word.

int UpdateIndex(char *word, int documentId, HashTable *index) — updates the data structure with the word-docID pair provided.

static int UpdateDocList(WordNode *word, int documentId) — updates the list of DocumentNodes pointed to by the provided word node. used in UpdateIndex.

int SaveIndexToFile(HashTable *index, char *filePath) — writes the data stored in the index to the file provided

int ReadFile(char *file) — recreates the data structure that produced the file provided.

Test Cases

Test: number of arguments provided is not 2 or 4
Expected Result: Alert the user in standard out, and exit the program

Test: the target directory is not a valid directory
Expected Result: Alert the user in standard out, and exit the program

Test: the index file provided already exists
Expected Result: Warn the user so that they do not overwrite the file and quite the program

Test: The reload file provided already exists
Expected Result: Warn the user so that they do not overwrite the file and quite the program

Test: the index file and the results file are the same
Expected Result: Warn the user, and exit the program

Test: one of the files in the directory has a name that is not an integer
Expected Result: Tell the user the words from file will not be included in the index file data

Test: if the file being read in testing mode is formatted incorrectly (e.g. line does not start with a word, every character after the word is not a number, there is not an even number of tokens separated by spaces in the line, the number of documents the word appears in is not the same as the number of docId-freq pairs in a line)
Expected Result: Alert the user to the specific problem, and exit the program