Indexer Design Spec
----

(1) *Input*

Normal run:
Command Input

```
          ./indexer  [TARGET_DIRECTORY] [RESULTS FILENAME]
```

Example command input:

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

`[TARGET_DIRECTORY] ./data/`
Requirement: The directory must exist and contain the output files of crawler component.
Usage: The indexer needs to inform the user if the directory is not found.

`[RESULTS FILENAME] index.dat`
Requirement: If the file exists, the caller must have read and write permission to the file.
Usage: The indexer needs to inform the user if the index building process failed (including the case when the file is does not have read and write permission)

Test run:
Command Input

```
./indexer  [TARGET_DIRECTORY] [RESULTS FILENAME] [RESULTS FILENAME] [RE
WRITEN FILENAME]
```

Example command input:

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

`[TARGET_DIRECTORY] [RESULTS FILENAME] ./data/ index.dat`
Requirement and Usage same as above.

`[REWRITEN FILENAME] new_index.dat`
Requirement: If the file exists, the caller must have write permission to the file.
Usage: The indexer needs to inform the user if the index building process failed (including the case when the file is does not have read and write permission)

(2) *Output*

Normal Run:
Indexer will write out to the file passed as the second argument, a line for every word found within the html portion of the crawler output files that is as big as or bigger than 3 letters with some format:

```
cat 2 3 4 7 6 …
```

"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.
This file may go on as such:

```
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
adjunct 6 4 1 5 1 43 1 44 1 139 1 142 3
```

Test Run:
The indexer will write out the same file as in the Normal Run, but will also create another file identical to this previous file with the name, `[REWRITEN FILENAME]` that was passed in as an argument. This file was created by re-reading the previously outputted file, re-building an inverted index, then outputting another file that should be identical to the previously outputted file from that re-built inverted index.

(3) *Data Flow*

Normal Run:

The html contents of one of the files within the directory passed in as the first argument will be parsed for words (outside html tags) and those words, documentID of where the words came from, and the occurrences of each word will be saved in an inverted index.

This process will continue until all of the files have been parsed and its words have been added to the inverted index.

The inverted index will then be converted into a file format and will outputs its contents appropriately following the specification of the file format explained in (2) *Output* section.

Test Run:

Test run will go through all the process from the Normal Run, but with additional processes after the steps of the Normal Run.

The output file from the Normal Run will be read to create an inverted index encoding the information of the file.

The inverted index will be converted back to a file format with a new name specified by the last argument. If the test is successful, the contents of this file and the output file from the Normal Run should be identical.s

(4) *Data Structures*

`invertedIndex` `testInvertedIndex`

A dictionary structure that holds words, which documentID it occurred and how many times it occurred at each document.

(5) *Pseudo Code*

```
1. Program parameter processing

2. Initialize data structures allocate Inverted_index, zero it, and set links to NULL.

3. LOG( "Building the index" );

4. Index *wordindex=buildIndexFromDirectory("argv[1]");

5. saveFile (argv[2], wordindex );
   LOG( "done!");

6. CleanDynamicList(wordindex)  // if testing then proceed

7. If (5 == argc) then

8. LOG ("Testing index\n");

9. Reload the index from the file and rewrite it to a new file       wordindex = readFile(argv[3]);

10. saveFile (argv[4], wordindex);
    LOG("test complete\n");

11. CleanDynamicList(wordindex);

12. Done.
```

Functional Specification

The indexer SHALL do the following for each file/document created by crawler in the TARGET_DIRECTORY :

- load the document from the file system, noting the unique document identifier (filename);
- use a provided function to parse the document discarding the HTML tags in order to count the occurrences of each word found in the document; and
- 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.

summary of error conditions detected and reported
----

1) With no arguments, error message should show to inform user of usage:
   Usage: ./indexer [TARGET_DIRECTORY] [RESULTS FILENAME]
   For test run: ./indexer [TARGET_DIRECTORY] [RESULTS FILENAME] [RESULTS FILENAME] [REWRITEN FILENAME]

2) With too few arguments, indexer will inform that the number of arguments must be 3 or 5:
   You must input 3 arguments for the normal run, or 5 arguments for the test run.

3) With too many arguments, indexer will inform that the number of arguments must be 3 or 5:
   You must input 3 arguments for the normal run, or 5 arguments for the test run.

4) With invalid directory argument, indexer will inform that the given directory is not a directory:
   Given "directory" is not a directory.

5) Any calloc failures, fopen failures, and any other failures of functions that indexer depends on:
   Failed index building or output processes.

test cases, expected results

Processing crawler output files at depth 0, 1, 2 and 3.
Expected Results: Outputs, index.dat and new_index.dat should have identical contents.