CSE 333 13su Homework #2

out: Thursday, July 11, 2013
due: Thursday, July 25, 2013 by **11:00 pm**.
[summary | part a | part b | part c | bonus | how to submit | grading]

Summary

For homework #2, you will use the LinkedList and HashTable modules that you built in homework #1 in order to finish our implementation of a file system crawler, indexer, and search engine. Homework #2 has three parts to it. In Part A, you will build a module that reads the content of a file into memory, parses it into a series of words, and builds a linked list of (word, position) information. In Part B, you will build modules that convert a series of these linked lists into an in-memory, inverted index. In Part C, you will use this in-memory, inverted index to build a query processor that has a console-based interface.

As before, pease read through this entire document before beginning the assignment, and please start early! There is a fair amount of coding you need to do to complete this assignment, and it will definitely expose any conceptual weaknesses you have with the prior material on C, pointers, malloc/free, and the semantics of the LinkedList and HashTable implementations.

In HW1, we asked you to look for out-of-memory errors and return an error code to the caller. To make life a bit easier, in HW2, you can use assert() to test for an out-of-memory error; this way, running out of memory will cause the program to halt, and you don't need to deal with returning an error code if one occurs.

Part A -- finish our fileparser.c

Context.

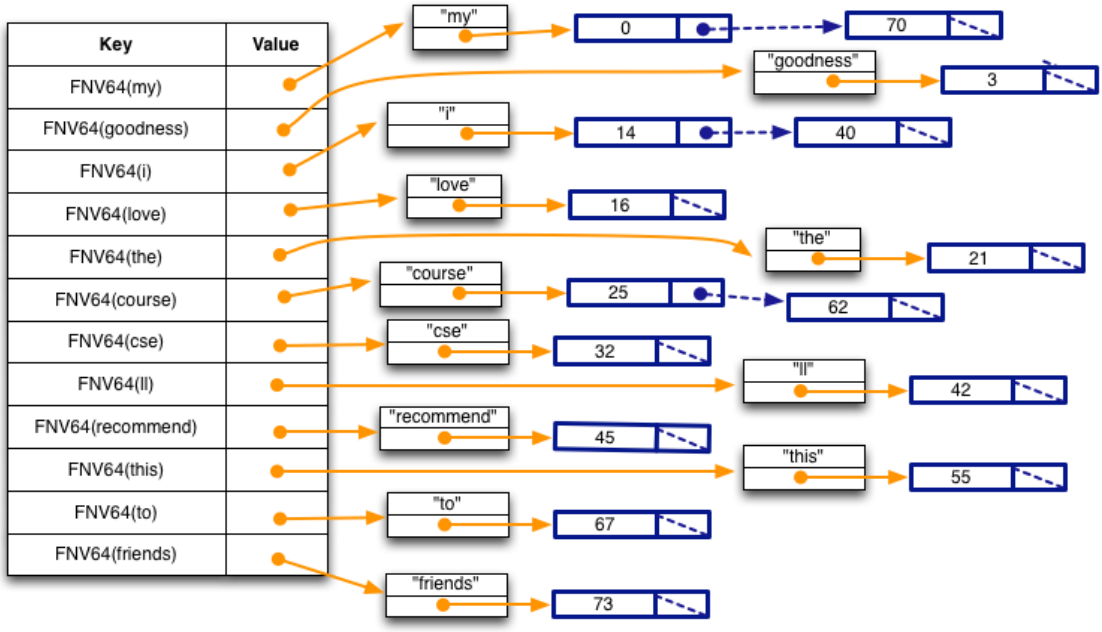
You're going to write a module that reads the contents of a text file into memory and then parses the text file to look for words within it. As it finds words, it will build up a HashTable that contains one record for each word. Each record will contain a lower-case copy of the word, and also a sorted linked list. Each record of the linked list contains an offset within the file that the word appeared in. (The first character in the file has offset zero.)

Our word parser won't be very smart. It will treat as a word any non-zero sequence of alphabetic characters separated by non-alphabetic characters.

So, graphically, what your module will take a text file that contains something like this. (Whenever you see a "\n" in the text file, that represents a "newline" control character that appears at the end of each line of the text file.)

```
My goodness!  I love the course CSE333.\n
I'll recommend this course to my friends.\n
```

and produces a data structure that looks like this:



Specifically, note a few things:

- Each key in the hash table is the result of calling the hashtable module's FNVHash64() function, passing the string as the first argument, and the strlen(string) as the second argument.
- Each element in the hash table is a pointer to a heap-allocated structure that contains two fields; a string and a linked list. Note the string is lower-cased, and that our parser is not very smart: because it treats any sequence of alphabetic characters, surrounded by non-alphabetic characters, as words, the word ***I'll*** was misparsed as the two words ***i*** and ***ll***.
- Each element in the linked list is an integer that contains a position in the text file that the word starts at. So, the word "my" starts at offset 0 in the text file, the word "i" appears twice, once at offset 14 and once at offset 40, and the word "course" appears twice, once at offset 25 and once at offset 62.
- Each list is sorted in ascending order.

What to do.

You should follow these steps to do this assignment:

1. Navigate to the directory containing your hw1 directory. Click (or right-click if needed) on this hw2.tar.gz link to download the archive containing the starter code for hw2 and the directory containing test data for the remaining parts of the project. Extract its contents and verify that you have everything.

   ```
   bash$ tar xzf hw2.tar.gz
   BASH$ ls
   clint.py  gtest  hw1  hw1.tar.gz  hw2  hw2.tar.gz  LICENSE.TXT  projdocs
   ```

   (It's ok if you've previously deleted the hw1.tar.gz archive with the hw1 starter files, but you still need the hw1 directory itself. hw2 won't build properly without it.)

2. Take a look inside hw2, and note a few things:

   - There is a subdirectory called libhw1/. It has symbolic links to header files and your libhw1.a from ../hw1. (Therefore, you should make sure you have compiled everything in ../hw1 while working on hw2.)
   - If you don't think your HW1 solution is up to the job, you can use ours instead. Our libhw1.a is in ../hw1/solution_binaries; just copy it over your ../hw1/libhw1.a.
   - Just like with HW1, there is a Makefile that compiles the project, a series of files (test_*) that contain our unit tests, and some files (doctable.c, doctable.h, filecrawler.c, filecrawler.h, fileparser.c, fileparser.h, memindex.c, memindex.h, searchshell.c) that contain our partial implementation of the project.
   - Type "make" to compile the project, and try running the test suite by running ./test_suite. It should fail, since most of the implementation is missing!

3. Also note there is a new subdirectory called "test_tree/" that contains a bunch of text files and subdirectories containing more text files. Explore this subdirectory and its contents; start with the README.TXT file.

4. Your job in Part A is to finish the implementation of fileparser.c. Start by reading through fileparser.h and make sure you understand the semantics of the functions. Also, look at the WordPositions structure typedef'ed in fileparser.h and compare it to the figure above. The function BuildWordHT() builds a HashTable that looks like the figure, and each value in the HashTable contains a heap-allocated WordPositions structure.

5. Similar to HW1, look through fileparser.c to get a sense of it's layout, and look for all occurrences of PART X (e.g., PART 1, PART 2, ...) for where you need to add code. Be sure to read the full file before adding any code, so you can see the full structure of what we want you to do. Once you're finished adding code, run the test suite and you should see three suites succeed: FPTestReadFile, FPTestBuildWordHT, and FPTestBigBuildWordHT.

6. As before, in the subdirectory solution_binaries/ we've provided you with linux executables (test_suite and searchshell) that were compiled with our complete, working version of HW2. You can run them to see what should happen when your HW2 is working.

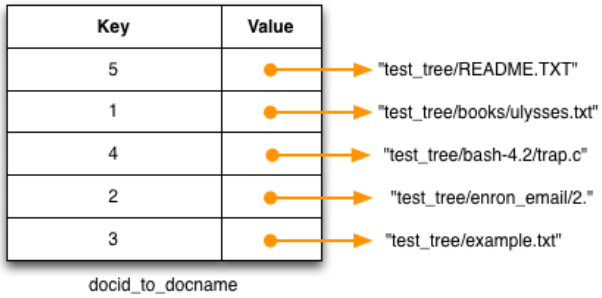
Part B -- finish our file crawler and indexer

Context.

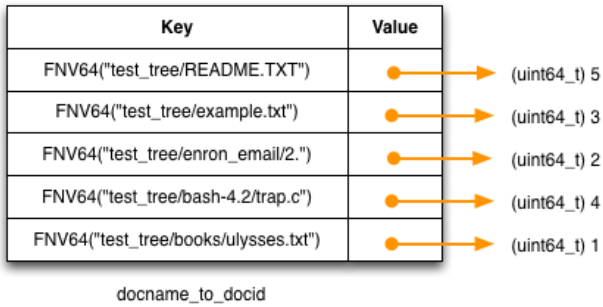
At a high-level, a search engine is really simple. It has three major components: a crawler, an indexer, and a query processor. A crawler explores the world, looking for documents to index. The indexer takes a set of documents found by the crawler, and produces something called an **inverted index** out of them. A query processor asks a user for a query, and processes it using the inverted index to figure out a list of documents that match the query.

File system crawler: Your file system crawler will be provided with the name of a directory in which it should start crawling. Its job is to look through the directory for documents (text files) and to hand them to the indexer, and to look for subdirectories; it recursively descends into each subdirectory to look for more documents and sub-sub-directories. For each document it encounters, it will assign the document a unique "document ID", or "docID". A docID is just a 64-bit unsigned integer.

Your crawler itself will build two hash tables in memory, adding to them each time it discovers a new text file. The two hash tables map from docID to document filename, and from document filename to docID:



docid_to_docname

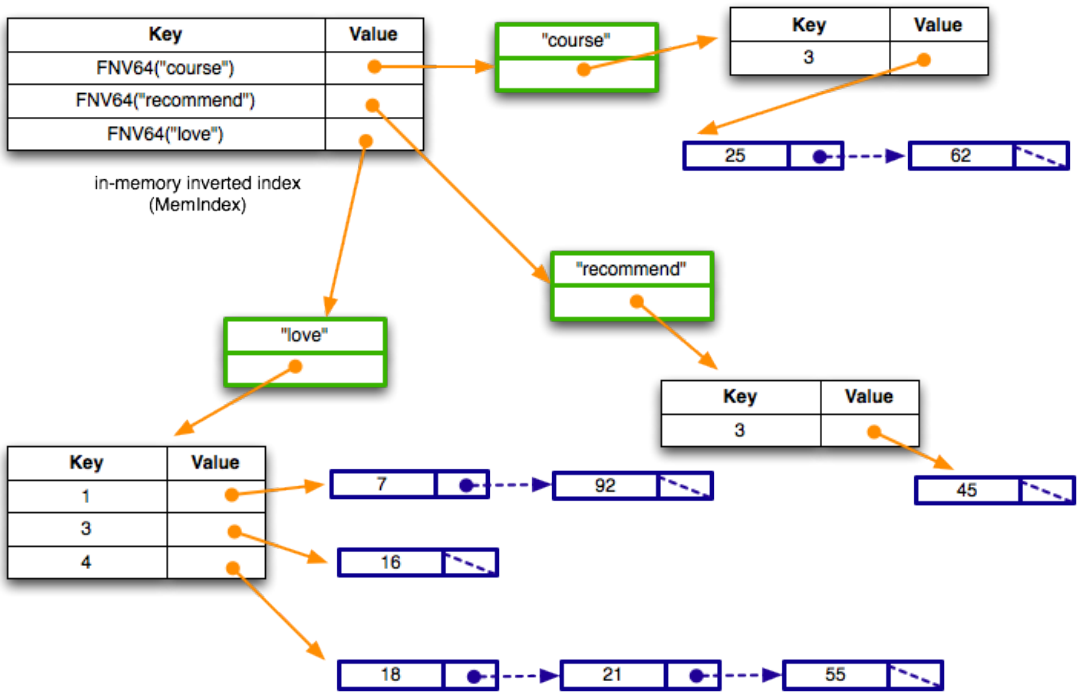


docname_to_docid

For each document the crawler finds, it will make use of your part B code to produce a word hashtable using BuildWordHT().

Indexer: This is the heart of the search engine. The job of the indexer is to take each word hashtable produced by BuildWordHT(), and fold its contents in to an inverted index. An inverted index is easy to understand; it's just a hash table that maps from a word to a "posting list," and a posting list is just a list of places that word has been found.

Specifically, the indexer should produce an in-memory hash table that looks roughly like this:



Walking through it, the inverted index is a hash table that maps from a (hash of a) word to a structure. The structure (shown in green) contains the word as a string, and also a HashTable. The HashTable maps from the docID (**not** the hash of docID) to a LinkedList. The LinkedList contains each position that word appeared in that docID.

So, based on the figure, you can see that the word "course" appeared in a single document with docID 3, at byte offsets 25 and 62 from the start of file. Similarly, the word "love" appears in three documents: docID 1 at positions 7 and 92, docID 3 at position 16, and docID 4 at positions 18, 21, and 55.

What to do.

The bulk of the work in this homework is in this step. We'll tackle it in parts.

1. Take a look at doctable.h; this is the public interface to the module that builds up the docID-to-docname HashTable and the docname-to-docID HashTable. Make sure you understand the semantics of everything in that header file; note that a single DocTable structure contains both of these tables, so when you implement AllocateDocTable(), you'll end up malloc'ing a structure that contains two HashTables, and you'll allocate each of those HashTables.

2. Take a look inside doctable.c; this is our partially completed implementation. Be sure to read the full file. Your job, as always, is to look for the "STEP X" comments and finish our implementation. Once you've finished the implementation, re-compile and re-run the test_suite executable to see if you pass our DTTestSimple test suite. If not, go back and fix some bugs!

3. Now, take a look inside filecrawler.h; this is the public interface to our file crawler module. Make sure you understand the semantics of everything in that header file. Next, read through the full filecrawler.c and then complete our implementation. Once you're ready, re-compile and re-run the test_suite executable to see if you pass our FCTestSimple test suite. If not, go back and fix some bugs!

4. Finally, take a look inside memindex.h. This is the public interface to the module that builds up the in-memory inverted index. Make sure you understand the semantics of everything in that header file. Next, read the full memindex.c, and then complete our implementation. (This is the most involved part of the assignment.) Once you're ready, re-compile and re-run the test_suite executable to see if you pass our MITestSimple test suite. If not, go back and fix some bugs!
   Once you've passed all of the test suites, re-rerun the test suites under valgrind and make sure you don't have any memory leaks.

   Congrats, you've passed part B of the assignment!

Part C -- finish our query processor

Context.

Now that you have a working inverted index, you're ready to build your search engine. The job of a search engine is to receive queries from a user, and return a list of matching documents in some rank order.

For us, a query is just a string of words, such as:

```
course friends my
```

The goal of our search engine is to find all of the documents that contain **all** of the words. So, if a document has the words "course" and "my" but not the word "friends," it shouldn't match.
The way to execute a query is really simple. First, the query processor must split the query up into a list of words (the strtok() function is useful for this). Next, it looks up in the inverted index the list of documents that match the first word. This is our initial matching list.

Next, for each additional word in the query, the query processor uses the inverted index to get access to the HashTable that contains the set of matching documents. For each document in the matching list, the query processor tests to see if that document is also in the HashTable. If so, it keeps the document in the matching list, and if not, it removes the document from the matching list.

Once the processor has processed all of the words, it's done. Now, it just has to rank the results, sort the results by rank, and return the sorted result list to the user.

For us, our ranking function is very simple: given a document that matches against a query, we sum up the number of times each query word appears in the document, and that's our rank. So, if the user provides the query "foo" and that words appears on a document 5 times, the rank for that document given that query is 5. If the user provides a multi-word query, our sum includes the number of times each of the words in the query appears. So, if the user provides the query "foo bar", the word "foo" appears in the document 5 times, and the word "bar" appears 10 times, the rank of the document for that query is 15. The bigger the sum, the higher the rank, and therefore the better the search result.

What to do.

We have provided you a partially working query processor that uses the Linux console to interact with the user. When you run the processor (called searchshell -- you can try a working searchshell in the solution_binaries/ directory), it takes from a command line argument the name of a directory to crawl. After crawling the directory and building the inverted index, enters a query processing loop, asking the user to type in a search string and printing the results.

For example, here's a transcript of a session I had with the searchshell program.

1. Take a look at searchshell.c, and get a sense for the overall structure of the program.
2. As before, search for "STEP X" to find the missing pieces and implement them.
3. Try running ./searchshell to interact with it, and see if your output matches the transcript from our search session with our solution. Alternatively, compare your searchshell to the searchshell we provided in the solution_binaries/ directory.

Optional bonus part D

We're offering two extra credit tasks. These are purely optional; if you choose not to do either, your grade won't be negatively affected at all. These are just if you happen to have the time and interest! You can do either or both of the bonus tasks.

1. When searching text documents for a given query, some words in the query do not add any value to the quality of the search results. These words are known as *stop words* and include common function words such as *the*, *is*, *and*, *at*, *which*, *there*, *on*, and so on. Some Web search engines filter these stop words, excluding them from both their indexes and ignoring them in queries; this results in both better search quality and significantly more efficient indexes and query execution.
   For extra credit, implement a stop word filter. The search shell should accept a second, *optional* argument "-s" that will act as a flag for turning the filter on. When the flag is not specified, your program should not filter any stop words. It is up to you to decide how you will implement the stop word filter (and where you'll get your list of stop words), but be sure to explain in your README file what changes you had to make and how your filter works. Stop words that have apostrophes in them should be handled the same way that you've handled the words in the documents.
2. You probably noticed that we went to a lot of trouble to have you include word position information in our inverted index posting lists, but we didn't make use of it. In this bonus task, you will. In addition to letting you search for words, modern search engines also let you search for phrases. For example, I could specify the following query:

   ```
   alice "cool fountains" flowers
   ```

   This query would match all documents that contain all of the following:

   - the word ***alice***, and,
   - the phrase ***cool fountains***; i.e., the words **cool** and **fountains** right next to each other, in that order, and
   - the word ***flowers***.

   Using the positions information in the inverted index postings list, implement support for phrase search. You'll have to also modify the query processor to support phrase syntax; phrases are specified by using quotation marks. Be sure to explain in your README file what changes you had to make to get phrasing to work.

What to turn in

When you're ready to turn in your assignment, do the following:

1. In the hw2 directory, run "make clean" to clean out any object files and emacs detritus; what should be left are your source files.
2. In the hw2 directory:

```
bash$ make clean
bash$ cd ..
bash$ tar czf hw2_<username>.tar.gz hw2
bash$ # make sure the tar file has no compiler output files in it, but
bash$ # does have all your source
bash$ tar tzf hw2_<username>.tar.gz
```

3. Turn in hw2_<*username*>.tar.gz hw2 using the course dropbox linked on the main course webpage.

Grading

We will be basing your grade on several elements:

- The degree to which your code passes the unit tests contained in test_suite.c. If your code fails a test, we won't attempt to understand why: we're planning on just including the number of points that the test drivers print out.
- We have some additional unit tests that test a few additional cases that aren't in the supplied test drivers. We'll be checking to see if your code passes these.
- The quality of your code. We'll be judging this on several qualitative aspects, including whether you've sufficiently factored your code and whether there is any redundancy in your code that could be eliminated.
- The readability of your code. For this assignment, we don't have formal coding style guidelines that you must follow; instead, attempt to mimic the style of code that we've provided you. Aspects you should mimic are conventions you see for capitalization and naming of variables, functions, and arguments, the use of comments to document aspects of the code, and how code is indented. We strongly suggest using the clint.py tool to check your code for style issues.