Assignment 9: Indexing Files to Query

DUE: July 31, 2019, 6pm.

In A9, you're going to use what you built in A8 to build, basically, a search engine for a bunch of movies. When a user enters a word, we'll print out a list of all the movies that have that word in the title. The interaction will look like this:

```
adrienne@virtualbox$ ./main data_small/
Crawling directory tree starting at: ../data_small/
crawling dir: ../data_small/
crawling dir: ../data_small/abc/
adding file to map: ../data_small/abc/xca

[snip]

adding file to map: ../data_small/xcs
crawling dir: ../data_small/xyz/
crawling dir: ../data_small/xyz/foo/
adding file to map: ../data_small/xyz/foo/xbl
adding file to map: ../data_small/xyz/foo/xct
adding file to map: ../data_small/xyz/xco
Crawled 24 files.
Parsing and indexing files...
processing file: 0
processing file: 1

[snip]

processing file: 24
Took 0.138030 seconds to execute.
```

crawling

```
22197 entries in the index.

Enter a term to search for, or q to quit: seattle
Getting docs for movieset term: "seattle"
```

→ invoke user input

```
indexType: tvEpisode
7 items
  Seattle Seahawks: Road to 2008
  St. Louis Cardinals vs. Seattle Seahawks
  Seattle
  Seattle Seahawks vs. Los Angeles Raiders
  Seattle Seahawks vs. Dallas Cowboys
  Denver Broncos vs. Seattle Seahawks
  Wasted in Seattle

Enter a term to search for, or q to quit: sister
Getting docs for movieset term: "sister"

indexType: movie
1 items
  South of Heaven: Little Sister
indexType: video
2 items
  Sister, Sister
  My Little Sister
indexType: short
3 items
  Sadie's Sister
  Sister
  Searching for: Sister Gertrude Morgan
indexType: tvEpisode
1 items
  I Ran Away from Home with My Sister

Enter a term to search for, or q to quit: q
Thanks for playing!
```

We'll be starting with more than 5 million movies, in about 50 different files. The brute-force approach to solving this problem is the following:

- Get a word from the user
- For every file:
  - Open the file
  - Read in a line
  - Find the title field
  - Look at every word in the field to see if there's a word that matches the user provided word
  - If it does, print out the movie.

You should have enough experience now to realize that will be VERY slow. If Google did that every time you ran a query, we wouldn't be using Google.

Perhaps a better way to do this would be to read in all 5M rows into a LinkedList of Movie structs, and when we query, iterate through all those movies, checking each word in the title and seeing if it matches what we're looking for. While this is a simple approach (I encourage you to test it), it will still be really slow.

We can reduce the response time by taking an approach where we:

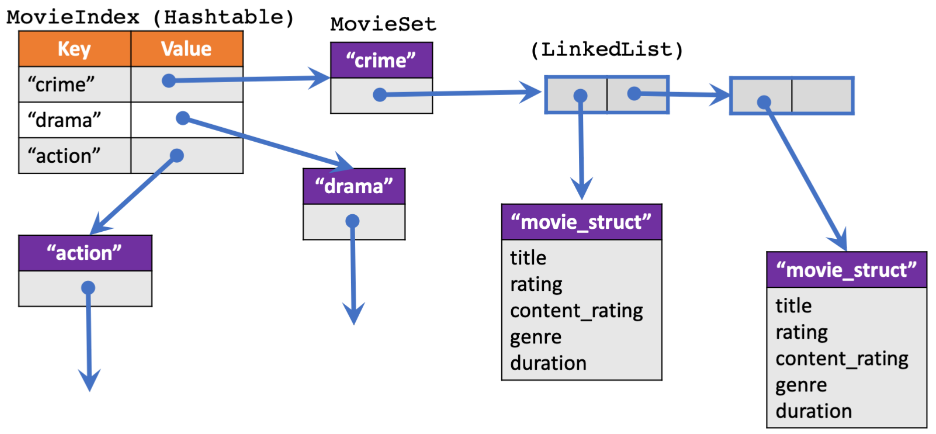
- Read in each of the 5M rows to a Movie struct
- As we read in the movie, create a hashtable that maps words in the title to a Movie struct.
- To query, return the list of Movie structs from the hashtable.

This has the downside of storing a lot of data about the Movies in memory. **How much memory would this take, assuming we have 5M movies?** In order to do all of this efficiently and respond quickly, we need to be a little clever.

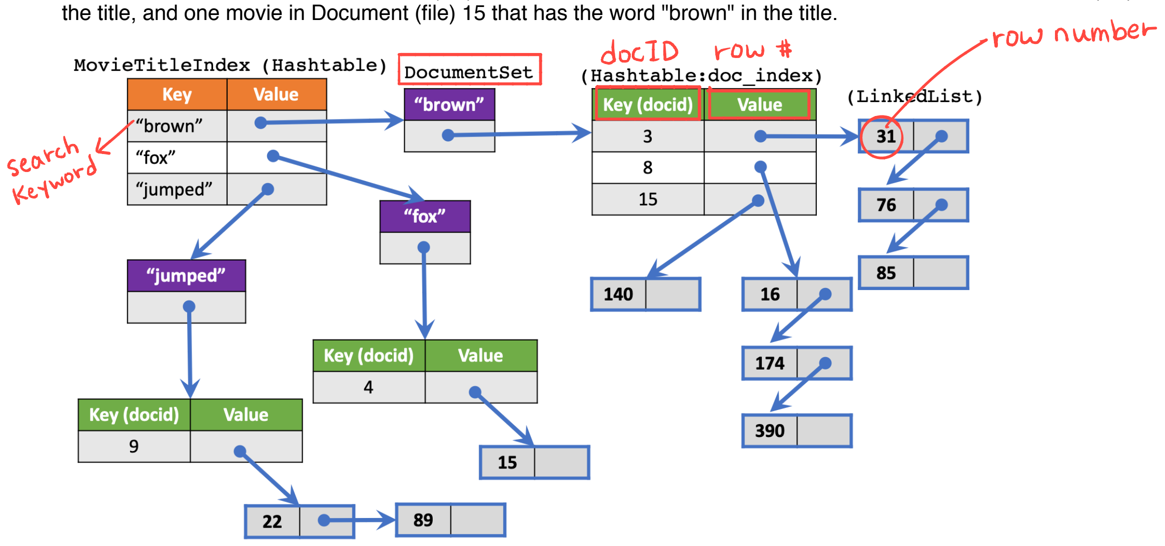
To address the memory issue, instead of loading all the Movies into memory, we can keep the data in the files, on the drive. Instead, in memory, we will keep track of which file and row a particular word appears in. Instead of our hashtable using a word as a key and a list of Movie structs as the value (that is, the MovieSet), the hashtable will use a word for a key, and DocumentSet as a value. The DocumentSet struct will hold a doc Id and a row number. We'll utilize another hashtable that maps a doc id to a filename. When we query, we'll get the list of DocInfo structs for that word, then use the DocIdMap to get the filename, open the file, move to the appropriate row, and pull out the information we need.

In A8, you created a flexible Hashtable implementation. Then you used that Hashtable to build a new, specific data structure-- a MovieIndex-- based on that Hashtable.

For the MovieIndex, we used values of a field of the movie stucture, such as Type, Year or Genre, to group movies together. MovieIndex was a Hashtable, where the key of the Hashtable was one of those values ("movie" vs "tvEpisode", or "1984" or "1995"), while the value of the Hashtable was a LinkedList of movies that shared that value for that attribute. (As depicted in the image below):

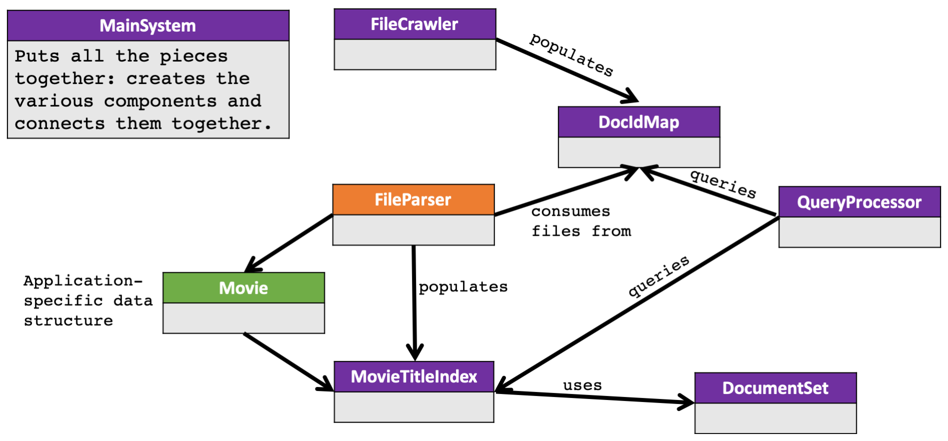


For A9, we're going to use our Hashtable again to build a similar kind of data structure as MovieIndex. This new data structure is called a MovieTitleIndex. The key will be a word, such as "sleepless" or "seattle", or as in the example below, "brown". The value will be another Hashtable. This second Hashtable will have a Document ID (such as 3, 8, 15) as a key, and a linked list of RowIDs (such as 31-76-85). This will represent the fact that the word "brown" is in the title of a Movie stored in Document (file) 3, rows 31, 76, and 85; there are also 3 movies in the Document (file) with ID 8 that have the word "brown" in the title, and one movie in Document (file) 15 that has the word "brown" in the title.



WHAT DOES THE OVERALL SYSTEM LOOK LIKE?

The following picture is an overview of the entire system:



You've already built parts of it (and you're provided with a library that gives you the functionality of A8). The big-picture steps you need to do:

1. Implement the FileCrawler. The FileCrawler is going to start at one directory, and find all the files in that directory, and all directories below that directory, and put them in a DocIdMap. The output of the FileCrawler will look something like this (the first part of the output above):

```
adrienne@virtualbox$ ./main data_small/
Crawling directory tree starting at: ../data_small/
crawling dir: ../data_small/
crawling dir: ../data_small/abc/
adding file to map: ../data_small/abc/xca
[snip]
adding file to map: ../data_small/xcs
crawling dir: ../data_small/xyz/
crawling dir: ../data_small/xyz/foo/
adding file to map: ../data_small/xyz/foo/xbl
adding file to map: ../data_small/xyz/foo/xct
adding file to map: ../data_small/xyz/xco
Crawled 24 files.
```

2. Modify your FileParser (which is now a DirectoryParser) to parse all files in the DocIdMap. As you're parsing the files, you'll index them (putting them in the MovieTitleIndex) based on their titles.

```
Parsing and indexing files...
processing file: 0
processing file: 1
[snip]
processing file: 24
Took 0.138030 seconds to execute.
22197 entries in the index.
```

3. Finish building a QueryProcessor. The QueryProcessor will take in queries from a user, and return back a list of movies with that word in the title.

```
Enter a term to search for, or q to quit: seattle
Getting docs for movieset term: "seattle"

indexType: tvEpisode
7 items
  Seattle Seahawks: Road to 2008
  St. Louis Cardinals vs. Seattle Seahawks
  Seattle
  Seattle Seahawks vs. Los Angeles Raiders
  Seattle Seahawks vs. Dallas Cowboys
  Denver Broncos vs. Seattle Seahawks
  Wasted in Seattle

Enter a term to search for, or q to quit: sister
Getting docs for movieset term: "sister"

indexType: movie
1 items
  South of Heaven: Little Sister
indexType: video
2 items
  Sister, Sister
  My Little Sister
indexType: short
3 items
  Sadie's Sister
  Sister
  Searching for: Sister Gertrude Morgan
indexType: tvEpisode
1 items
  I Ran Away from Home with My Sister
```

```
Enter a term to search for, or q to quit: q
Thanks for playing!
```

SUMMARIZING THE ASSIGNMENT

Once again, to summarize:

We have a unique DocumentSet for every unique word in every movie title. The DocumentSet maintains a list of DocId's (which document contains the movie with this word) and a list of rows in that document (which row the specific movie is in). That is, in the `doc_index` hashtable, the key is an int that is a docId, and the value is a LinkedList where the payload is an int (row number).

When we ask the MovieTitleIndex for a set of SearchResults for a given word, a list of docId-rowId pairs is returned. Then, we use that information to go look up the specific row in the file and return that row to the customer.

This is the overall flow:

1. The FileCrawler starts at a given directory and puts all the files in the DocIdMap, which assigns a unique ID to that file.
2. The DirectoryParser gets all the files from the DocIdMap, reads in each row (to an application-specific data structure call a movie) and gives that to the MovieTitleIndex.
3. The MovieTitleIndex chooses how to index the Movie datastructure, storing the information using the DocSet.
4. When all the files have been indexed, the QueryProcessor gets a request from a customer (e.g. "Give me all movies with SISTER in the title"), asks the MovieTitleIndex for the relevant records, and returns them to the customer.

---

1. THE DISCOVERY PHASE: FILE CRAWLING

The goal of this step is to find all the files that contain data we care about. We do this by writing a program that traverses a given directory, and create a data structure that contains a mapping between a unique document ID and a filename.

In this step, use the provided code in `FileCrawler.c, FileCrawler.h, DocIdMap.c, DocIdMap.h` to implement the relevant method(s).

After the FileCrawler runs, the DocId map will be populated with all of the files in a specified directory, and they will all be assigned a unique DocId number.

A sample directory (`movies_small`) is provided in the resources repo for you to use.

FileCrawler will generate a DocId map that is conceptually this:

`docidmap (Hashtable)`

Key (docid)	Value (filename)
1	"movies/title_ab.txt"
2	"movies/title_ac.txt"
3	"movies/title_ad.txt"

Notes:

- Remember, a file directory is like a tree. Please traverse it in an alphabetical, depth-first manner.
- An library including a LinkedList and Hashtable implementation is provided for you (libhtll.a)
- `main.c` provides an example of how it will be used (line 38):

```
    DocIdMap docs = CreateDocIdMap();
    CrawlFilesToMap(dir, docs);
```

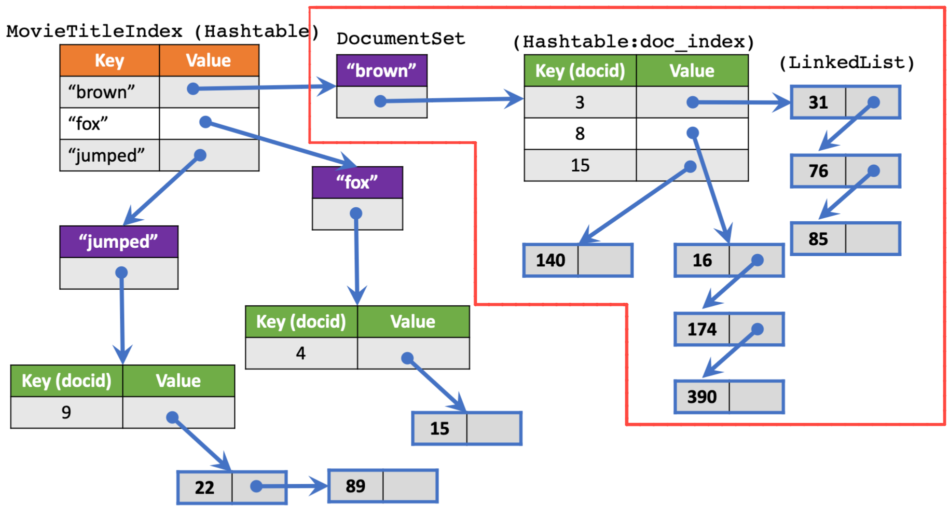
1. Make the FileCrawler.* tests pass
   - After the FileCrawler tests pass, also make sure valgrind is clean.
   - `make test`
   - `./test_suite --gtest_filter=FileCrawler.*`
   - `./test_suite --gtest_filter=DocIdMap.*`
   - `valgrind ./test_suite --gtest_filter=FileCrawler.*`
   - `valgrind ./test_suite --gtest_filter=DocIdMap.*`
   - `DocIdMap.c`
   - `FileCrawler.c`

Follow the steps identified in the code, such as "STEP 1:" and "STEP 2:".

```
grep TODO *.c
```

2. BUILD THE DOCSET AND MOVIETITLEINDEX

Next we'll build the data structure we'll use to index all the Movies. As a reminder, this is a picture of the structure we're building:



You'll see that MovieTitleIndex is mostly implemented for you. It's not much different than MovieIndex that you built in A8. What's different here is the payload: DocSet, instead of MovieSet.

Make sure you understand the DocSet, and write the code to add and retrieve data from the DocSet.

1. Make the DocSet.* and MovieTitleIndex.* tests pass
   - After the DocSet tests pass, also make sure valgrind is clean.
   - `make test`
   - `./test_suite --gtest_filter=MovieTitleIndex.*`
   - `./test_suite --gtest_filter=DocSet.*`
   - `valgrind ./test_suite --gtest_filter=MovieTitleIndex.*`
   - `valgrind ./test_suite --gtest_filter=DocSet.*`
   - `DocSet.c`
   - `MovieTitleIndex.c`

3. THE INDEXING PHASE: FILE PROCESSING

Now that we've found all the files that contain data we care about, we want to process them. We're going to build a DirectoryParser, which is not unlike the FileParser that you built in A8. Rather than parsing a single file, we're going to add a function that allows it to parse ALL the files that are listed in a DocIdMap.

Once again, the format of the data is as below:

`-` represents an null value. The genre field is a comma-seperated list of genres: it should be parsed.

```
id|type|title1|title2|_|year|_|duration|genre
```

A few rows:

```
tt7356766|tvEpisode|Walker Cup|Walker Cup|0|2017|-|-|Sport
tt7356768|tvSeries|Aaron Olivo's Skits|Aaron Olivo's Skits|0|2017|-|-|Comedy
tt7356770|short|Ducko|Ducko|0|2018|-|1|Comedy,Short
tt7356774|tvEpisode|Kitchen Disaster to Master|Kitchen Disaster to Master|0|2017|-|-|Comedy,Game-Show,Reality-TV
tt7356776|tvEpisode|Carolina Panthers at San Francisco 49ers|Carolina Panthers at San Francisco 49ers|0|2017|-|-|Spor
tt7358226|tvEpisode|Caméra invisible|Caméra invisible|0|1994|-|-|Adventure,Family,Thriller
```

Your new `DirectoryParser.c` will include functions to:

- Iterate through all the documents in a DocId
- For every row of every document, create a Movie
- Add that Movie to a MovieTitleIndex

1. Make the DirectoryParser tests pass
2. `DirectoryParser.c`

4. THE FINDING PHASE: PROCESSING QUERIES

Here's where we handle the queries that are passed in by users. While this can get very complex, we're going to keep it simple: users enter a single word, and we print a list of movies that have that word in the title.

You're provided with a starting point of `QueryProcessor.h, .c`.

The basic idea of what you're implementing:

- The user enters a query term (we're keeping it to a single word at this point)
- QueryProcessor uses MovieTitleIndex to find the entry that has a corresponding key: This returns a Hashtable[DocId, LinkedList[RowIds]].
- Iterate through the returned hashtable for each document:
  - Use the DocIdMap to find the filename of the document
  - Open the file, find the correct row
  - Create a Movie struct from the row

    - Put it into a MovieIndex (which we're using just to print out nicely)
    - Repeat for all RowIds in the LinkedList[RowIds].
- After all Movies have been loaded, use the MovieIndex to PrintReport nicely.

A SLIGHT ASIDE: UTILIZING A8 TO PRINT OUT A NICE REPORT

After you've built the index, use the functions in `MovieReport.c`, which is based on the code we wrote in A8, to use the hashtable/index to generate a report of the indexed files. This is provided for you; it should be straightforward. Ask questions if it's not obvious how to use or it's not working correctly for some reason.

```
Genre: Crime
3 items
  The Shawshank Redemption
  The Usual Suspects
  American History X
Genre: Adventure
2 items
  The Lord of the Rings: The Fellowship of the Ring
  Back to the Future
...
```

```
Type: Short
3 items
  The Shawshank Redemption
  The Usual Suspects
  American History X
Type: tvEpisode
2 items
  The Lord of the Rings: The Fellowship of the Ring
  Back to the Future
Type: movie
5 items
...
```

- Print out the indexed type/value on one line, then the number of items in that value, then the movie name for each movie in that genre.

1. Make the QueryProcessor.* tests pass
    - After the QueryProcessor tests pass, also make sure valgrind is clean.
    - `make test`
    - `./test_suite --gtest_filter=QueryProcessor.*`
    - `valgrind ./test_suite --gtest_filter=QueryProcessor.*`

QUESTIONS TO ANSWER

- Why do we create this mapping between a DocId and a filename, rather than just storing a list of filenames?
- [More questions are coming]

The `main.c` file is provided for you, that sets up the index and runs a loop for the user to query the index.

RELEVANCE/SIMILARITIES TO THE REAL WORLD

In the bigger scheme of things, we're putting pieces together to build a search engine. A search engine has a few components: a file crawler, a file processor, an indexer, and a query processor. The file crawler starts in one place, and traverses to find other files. On the web, this means starting at one page and following all the links on the page; in a file system, it means starting in one directory and traversing through the directory structure until all files have been found. The file processor takes each of those files and finds significant words or concepts in the file. The indexer creates a data structure that (essentially) populates a data structure that makes it easy to find documents that have similar topics, or find documents that contain a certain word. The query processor lets us work with that data structure to get the results based on what we ask for.