CSc 460 — Database Design
Fall 2018 (McCann)

http://u.arizona.edu/~mccann/classes/460

Program #2: Dynamic Hashing

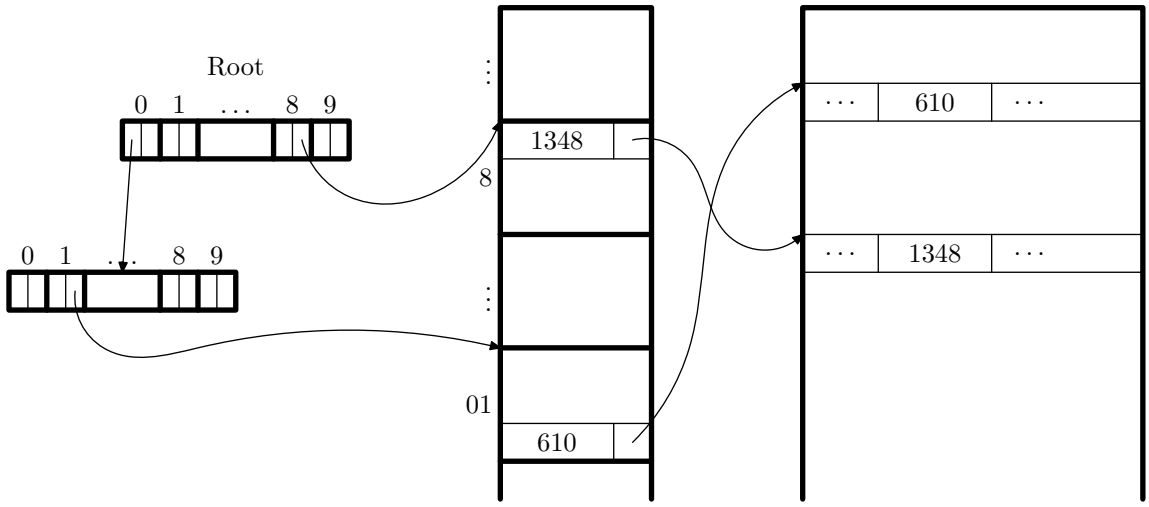
Due Date: September 20th, 2018, at the beginning of class

Overview: In our first programming assignment, you created a binary file of uniformly- sized records. Querying the content of such files can be made more efficient by using an index.

In class we talked about Dynamic Hashing indices under the assumption of a binary (2–way) tree. If your situation permits – and ours does – the use of an n–way tree can produce a shallower in–memory tree index and thus faster searches. Further, we can create indices by reading the content of the field being indexed in reverse order. Doing so can further improve querying performance.

Assignment: Write a complete, well-documented Java program that creates a Dynamic Hash index for the binary file of airline flights that your Program #1 created and uses it to satisfy a simple type of query. In particular, you are to index on the ARR_TIME field, using the digits in reversed (right to left) order, assuming the implied zeroes when necessary to create a four-digit value.

Recall that a dynamic hash structure consists of a bucket directory tree and a collection of buckets. In this assignment, the tree is held in–memory and the key field of the data consists of just numeric characters. Your directory tree should be structured with (eventually) one level per digit in the key, and 10 tree pointers and 10 hash file "pointers" (really just byte offsets into the binary file of hash buckets) per node. The following picture should help:



Bucket Directory Tree Hash Bucket File Program 1's Binary File

At the beginning, your tree should consist of just the root node with pointers to 10 empty buckets in the hash bucket file. As buckets are filled based on the right–most digit of the key of the record being inserted, new leaf nodes will be added to the tree and buckets will need to be split. To add new buckets to the hash file, all you need to do is append the new buckets to the end of the file (and, optionally, reuse the old bucket, to save space). For this program, use hash buckets that can hold 100 index records each. I recommend that each bucket be based on a data structure that consists of (a) a count of how many of the 100 slots are filled, and (b) an array of slots. The hash file will be a binary file of these bucket structures.

(Continued . . .)

Once the dynamic hashing structures (the in-memory tree and the on-disk bucket file) are created, the program is to process a simple variety of query using your dynamic hashing index. Prompt the user to enter, one at a time, zero or more ARR_TIME suffixes (e.g., an ARR_TIME of 359 has potential suffixes of 9, 59, 359, and 0359). Your program should display the UNIQUE_CARRIER, FL_NUM and ARR_TIME fields associated with the given suffix, and the total number of records that were printed. Sequentially scanning the binary file to find the matching records is not acceptable — the querying must use your index. Display the UNIQUE_CARRIER, FL_NUM and ARR_TIME fields using the same basic output format you used for Program #1.

Data: The binary file to be indexed is to be the one your Prog1A.java program from Program #1 creates. If you need to make changes to Prog1A.java to create a correct binary file for this program to index, be sure to submit your updated Prog1A.java in addition to this program.

If a record's ARR_TIME value is missing, skip the record (that is, do not add an entry for it into the index).

The queries will consist of suffixes as described above. A sequence of 0000 is the termination condition. Invalid (not legal four-digit times) suffixes should be handled reasonably.

As for dreaming up queries for testing, that's up to you. We'll test with a variety of query strings, of course, to make sure that your code is operating correctly and robustly. Thus, so should you.

Output: Apart from the various prompts to the user, the only displayed output from your program should be the query results. For each suffix, after the list of records, display a count of the number of records that match each query: "# records matched your query."

Hand In: You are required to submit your completed program file(s) using the `turnin` facility on lectura. The submission folder is `cs460p2`. Instructions are available from the document of submission instructions linked to the class web page. Because we will be grading your program on lectura, it needs to run on lectura. This means that you need to test it on lectura. Name your main program source file `Prog2.java` so that we don't have to guess which files to compile, but feel free to split up your code over additional files if you feel that doing so is appropriate. Submit each file as-is; that is, do not 'package' them into ZIP or TAR files.

Want to Learn More?

- There aren't many examples of our version of Dynamic Hashing available. I refer to this type of external hashing as "Dynamic Hashing" to distinguish it from Extendible Hashing, which is more well-known. But, Extendible Hashing is an example of the general concept of dynamic hashing, too.

Other Requirements and Hints:

- The bucket size of 100 records is insufficient to allow your program to index your entire binary file. When you reach the point that an insertion cannot be made, display to the user the message "Index capacity reached; binary file record X could not be inserted.", and move on to the query portion of the program. When testing on small binary files for which 100–record buckets are adequate, display the message "All Y records have been processed.", and again continue with the querying.
- Work on just a part of the program at a time; don't try to code it all before you test any of it. Don't be afraid to do things a little bit backwards; for example, it's nice to have a crude query algorithm in place to help you test the construction of your index.
- You can make debugging easier by using only a small amount of carefully selected data – and very small buckets to force early splitting – as you develop the code. Switch to the complete data file and full–size buckets when everything seems to be working.
- Repeating one of my standard pieces of advice: Comment your code according to the style guidelines *as you write the code* (not in the last few hours before the due date and time!). Doing this makes writing documentation far less tedious.
- As always: **Start early!** There's plenty to do here, and understanding the hash structures will take time.