CSc 460 — Database Design
Spring 2023 (McCann)

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

Program #2: Dynamic Hashing

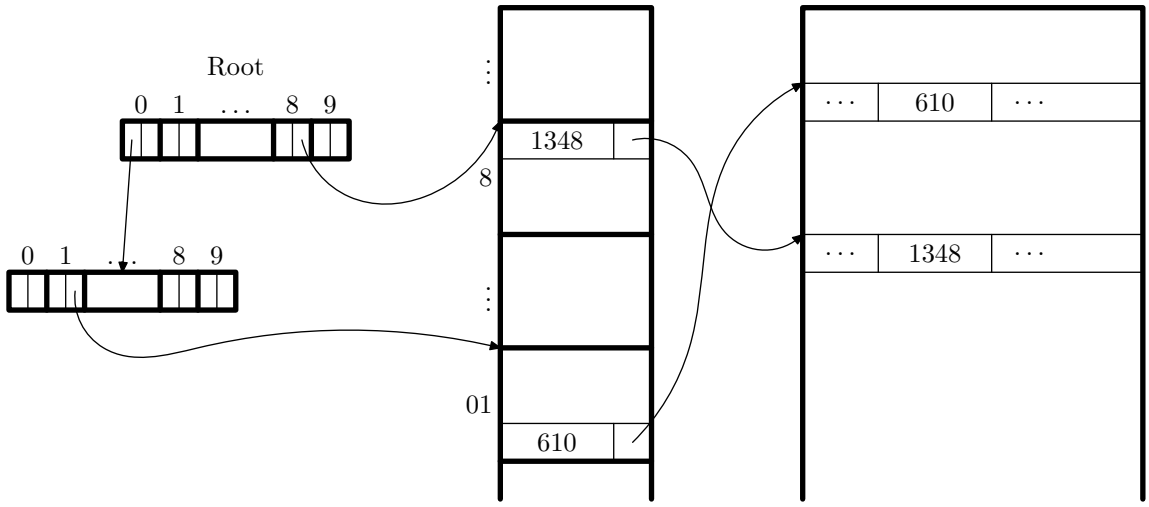
Due Date: February 9th, 2023, at the beginning of class

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 tree index and thus faster searches. Further, we can create indices by using the content of the field being indexed in reverse order. Doing so can further improve querying performance.

Assignment: Write a complete, well–documented Java 16 (or earlier) program named `Prog2.java` that creates a Dynamic Hash index for the binary file of women's basketball players that your Program #1 created and uses it to satisfy a simple type of query. In particular, you are to index on the player_id field, using the digits in *reversed* (right to left) order.

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 (player_id) consists of just integers. Your directory tree should be structured with (if necessary) 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 a maximum of 50 index records each. We recommend that each bucket be based on a data structure that consists of (a) a count of how many of the slots are filled, and (b) an array of slots. The hash file will be a binary file containing 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 player_id suffixes (e.g., a player_id of 123 has potential suffixes of 3, 23, 123, 0123, etc.). Your program should display all of the player_id, name, and hometown_clean fields associated with the given suffix, followed by the total number of records that were printed. Sequentially scanning the binary file (or the index file) to find the matching records is not acceptable — the querying must use your index hierarchically. Display the player_id, name, and hometown_clean fields using the same square–bracketed 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 player_id 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. An input sequence of 0000000 (seven zeroes) is the termination condition. Invalid input (illegal 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: As noted above, your program is to generate and store the hash bucket file as a binary file in the current directory. Apart from the various prompts to the user, the only displayed output from your program should be the query results, which are also described above.

Hand In: Using the `turnin` facility on lectura, submit your `Prog2.java` file, any additional source code files on which it depends (e.g., `.java` files with code of additional classes you wrote), and the complete version of your `Prog1A.java` program that creates the binary files that `Prog2` reads. We expect your code to work as specified and to follow the class programming style guidelines. The submission folder is `cs460p2`. Please submit each file as–is; that is, do not 'package' them into ZIP or TAR files. 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 before submitting it.

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:

- 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 bucket capacities 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 *before and 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, and should improve the quality of your code.
- As always: **Start early!** There's plenty to do here, and understanding the hash structures will take time.