CSc 460 — Database Design
Spring 2026 (McCann)

https://cs.arizona.edu/classes/cs460/spring26/

Program #2: Linear Hashing Lite

Due Date: February 12th, 2026, at the beginning of class

Overview: In class we cover Dynamic Hashing and Extendible Hashing indices. Both require a directory. In 1980, Witold Litwin introduced Linear Hashing, a directory–free hash–based indexing structure. In this assignment, you'll implement what we'll call Linear Hashing Lite[1] (LHL) and use it to assist the querying of a binary file. See the last page of this handout for the description of LHL.

Assignment: This assignment is in two parts, both of which have the same due date.

Part 1: Write a program named `Prog21.java` that creates, in a binary file named `lhl.idx`, a Linear Hashing Lite index for the `Dataset2.bin` database file created by your `Prog1A.java` program. Write your program to store `lhl.idx` in the current directory (that is, do not add any path information to the index file name), and to accept the complete path of the `Dataset2.bin` file as the command–line argument. The index is to be constructed using the 'Data.entry' field (the second field) as the key.

For this implementation of LHL, the buckets will have a maximum capacity of 30 index records. Initially, the index will consist of two empty buckets, $H = 0$, and the hash function will be $h_H(k) = |\ k.\texttt{hashCode()}\ |\ \%\ (2^{H+1})$, where $|\ k.\texttt{hashCode()}\ |$ is the absolute value of the result of `java.lang.String`'s `hashCode()` method executed on k, a Data.entry string.

After the index file has been created, display, labeled clearly and in this order, (a) the number of buckets in the index, (b) the number of records in the lowest–occupancy bucket, (c) the number of records in the highest–occupancy bucket, and (d) the mean and the median of the occupancies across all buckets, to two decimal places.

Part 2: The second task is to write a program named (you guessed it) `Prog22.java` that processes a simple variety of query with the help of your LHL index. The complete paths to both the `lhl.idx` index file and the `Dataset2.bin` database file are provided, in that order, as command–line arguments. Use a loop to prompt the user to enter any number of target values, one at a time. If the target matches a Data.entry value in the LHL index, display the same three values from the corresponding DB file record in the database file as you displayed in Prog1B, and in the same format. Otherwise, display the message "The target value '`#####`' was not found." (Of course, display the actual target value, not a bunch of pound signs!) Control `Prog22.java` the same way you controlled the execution of `Prog1B.java`, terminating when a target value of -1000 is entered.

In order to successfully write `Prog22.java`, you may decide that you need to communicate to it some metadata about the index created by `Prog21.java`. Program #1 gave you some experience doing that sort of thing (with the max string lengths). You may do the same sort of thing in this assignment. Keep in mind that your index's size will vary based on the number of records in the DB file (binary file) your `Prog1A.java` creates, much as your DB file's size varies based on the CSV file's content.

Data: We will use your own `Dataset2.bin` file to test your programs, which means that you will need to submit it using `turnin`. You may also submit your current `Prog1A.java` program, if you wish to do so. Why would you want to? If the TAs have a problem with your binary file and also have the current version of the program that generates it, they can try to recreate it themselves.

As for dreaming up queries for testing, that's up to you (but shouldn't be too hard). We'll test with a variety of Data.entry field values (present in the data and not). Thus, so should you.

(Continued...)

[1] One–third fewer headaches than regular Linear Hashing, but the same great idea!

Output: The basic output expectations of `Prog21.java` and `Prog22.java` are given with their descriptions, above.

Hand In: You are required to submit your completed program files (`Prog21.java` and `Prog22.java`), and your `Dataset2.bin` file on which those programs operate, using the `turnin` facility on lectura. The submission folder is `cs460p2`. Optionally, you may also provide your current `Prog1A.java` program. Submit all files as–is; that is, **do not 'package' them** into ZIP or TAR files, and **do not** just drop your project's subdirectory from your IDE — find the .java files and submit just those.

Because we will be *grading* your program on lectura, it needs to *run* on lectura, and so you need to *test* it on lectura. Name your main program source files as directed above, 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.

Want to Learn More?

> *Remember: What this assignment requires is* ***not*** *the full version of Linear Hashing described in these papers; I'm referencing them to satisfy the curious among you.*

- Lots of copies of Litwin's original Linear Hashing paper are floating around the internet, because the official source isn't readily available. Google Scholar can point you to a copy: `https://scholar.google.com/scholar?q="Linear+Hashing+A+New+Tool+for+File+and+Table+Addressing."`
- A somewhat more approachable description of Linear Hashing can be found as part of the paper "Dynamic hash tables" by Per–Ake (Paul) Larson: `https://dl.acm.org/doi/10.1145/42404.42410`

Other Requirements and Hints:

- I will be providing a video in which I will explain how to get started with the construction of a linear hashing lite index; look for it in the Video module of the Content area in Brightspace.

  I noticed that I forgot to clarify the abbreviated notation that I used for writing the hash functions on the board for the video. The "k" in "k%2", for example, is the result of the absolute value of the hashCode() of the Data.entry strings. It's not meant to be a mod of the string itself. See also the second paragraph of the Part 1 section on the first page of this handout.
- Comment your code according to the style guidelines *as you write the code* (not an hour before class!).
- Work on just a part of the assignment 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 basic query program in place to help you test the construction of your index.
- You can make debugging easier by using only a small amount of data with very small buckets as you develop the code, and switch to the complete data file and full–size buckets when everything seems to be working.
- As always: **Start early!** We don't have an early, first–part due–date on this one; you'll have to do your own planning.

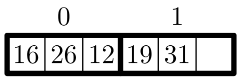
(Continued...)

Linear Hashing Lite: The Basics

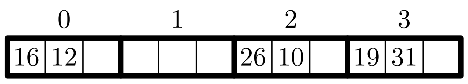
Like Dynamic and Extendible Hashing, Linear Hashing was created for indexing. Unlike them, Linear Hashing does not use a directory as its hash function. Instead, it relies on a characteristic of division–based hash functions, thus so does our simplified version, described on this page using generic data.

• **Insertion**

Consider two hash buckets, which, for this demonstration, are each capable of holding at most $bf = 3$ index records, and the simple hash function $h(k) = k \,\%\, 2$. Assuming that the key values being hashed are the integers 16, 19, 26, 31, and 12, the result is:



To store the key value 10, we need to expand the hash table and change the hash function. Specifically, we will change the hash function to be $h(k) = k \,\%\, 4$ (double the divisor), and double the number of buckets in the table (to match the range of the new function). We also need to re–distribute (re–hash) the existing key values. Because of our choice of hash functions, this isn't a typical re–hash: The values currently in bucket 0 will either stay there, or will move to bucket 2. Similarly, those in bucket 1 will stay or move to bucket 3. (Why this happens is left as an exercise for the reader.) After the re–hashing, and after the insertion of 10, the table looks like this:



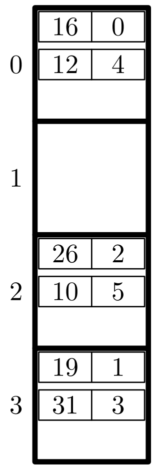
Each time we need to insert into a full bucket, the same steps are performed: We double the divisor of the hash function, double the number of buckets in the hash table, re–hash the existing values, and insert the new key. Each time we re–hash the content of a bucket, the entries either stay in the same bucket, or move to just one of the new buckets. This property helps minimize I/O operations.

Performing insertions for this assignment is a bit more involved. We discover the keys to be inserted by sequentially reading the records in the binary file created by `Prog1A.java` (the 'Database File' in the figure below). As we read the records, we also note their locations (here, their record numbers, if you imagine the file to be an array of records). Together, the key and the record number form the index record that is inserted into the hash bucket. It is helpful to parameterize the hash function. That is, instead of $h(k)$, express the hash function as $h(k, H) = k \,\%\, (2^{H+1})$, where $H = 0$ initially and increases by one whenever the hash table grows. (Note that the number of buckets in the hash table is the same as the hash function's divisor, 2^{H+1}.)

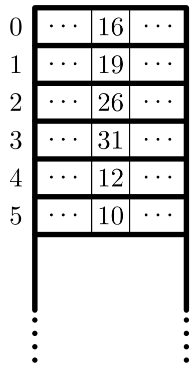
• **Searching**

Searching a Linear Hashing Lite index is straight–forward: Using the given target to be located and the last value of H, locate and read the bucket of the hash file that contains (or would contain) the target. Sequentially search the bucket content to see if the target is there. If it is, use the paired DB file record number to access the DB file fields you need to output.

For example, consider the figures to the right (an enhanced version of the above insertion example) and the target value 31. Our last value of H was 1. $h(k, H) = h(31, 1) = 31 \,\%\, (2^{1+1}) = 3$. Reading and searching bucket 3 locates 31's index record, which contains the record number of 31's complete record in the database file (in this example, the record number is 3). Reading that record provides 31's associated data, allowing the query to be completed.



LHL Hash File
($H = 1, bf = 3$)



Database File