CSc 460 — Database Design
Spring 2017 (McCann)

http://www.cs.arizona.edu/classes/cs460/spring17

Program #1: Interpolation Searching a Binary File

Due Dates:

Part A:	January 25th, 2017, at the beginning of class
Part B:	February 1st, 2017, at the beginning of class

Overview: In the not–to–distant future, you will be writing a program to create an index on a binary file. I could just give you the binary file, or merge the two assignments, but creating the binary file from a generically–formatted text file makes for a nice "shake off the rust" assignment, plus it provides a gentle (?) introduction to binary file processing.

A basic binary file contains information in the same format in which the information is held in memory. (In a standard text file, all information is stored as ASCII or UNICODE characters.) As a result, binary files are generally faster and easier for a program to read and write than are text files. When your data is well-structured, doesn't need to be read by people or ported to a different type of system, binary files are usually the best way to store information.

For this program, we have lines of data items that we need to store, and each line's items need to be stored as a group (in a database, such a group of *fields* is called a *record*). Making this happen in Java requires a bit of effort; Java wasn't originally designed for this sort of task. By comparison, "systems" languages like C can interact more directly with the operating system and provide more convenient file I/O. On the class web page you'll find a sample Java binary file I/O program, which should help get you started.

Assignment: To discourage procrastination, this assignment is in two parts:

Part A Available from `lectura` is a file named `chicagotaxi-nov2016.csv` (see the Data section, below). This is a text file consisting of over 250K lines of data on taxi trips in Chicago, IL that started during the last week of November of 2016.

Using Java 1.8, write a program named `Prog1A.java` that creates a binary version of the text file's content that is sorted in ascending order by the first field of each record (the 'Trip ID' field). Additional details:

- For an input file named `file.csv`, name the binary file `file.bin`. (That is, keep the file name, but change the extension.) Don't put a path on it; just create it in the current directory.)
- Field types are limited to `int`, `double`, and `String`. All money fields are to be doubles, and pad Strings on the right with spaces to reach the needed length(s). (For example, `"abc␣␣"`.)
- For each column, all values must consume the same quantity of bytes. This is easy for numeric columns, but for String columns we don't want to waste storage. For those columns, you need to determine the number of characters in the longest value, and use that to size each value in the column for storage. This must be done for each execution of the program. (Why? The data doesn't provide field sizes, so we need to code defensively to accommodate unexpected changes.)

To repeat the info at the top of this handout: Part A is due in just one week.

(Continued...)

Part B Write another complete, well-documented Java 1.8 program named `Prog1B.java` that performs both of the following tasks:

1. Reads and prints to the screen the Trip IDs and Trip Totals of the first five records of data, the middle record (or both middle records, if the quantity of records is even), and the last five records of data in the binary file, **plus** the total number of records in the binary file.
2. Using one or more Trip ID prefixes given by the user, locates within the binary file (using interpolation search; see below) and displays to the screen the Trip IDs and Trip Totals of all records having that Trip ID prefix.

Some details:

- Output the data one record per line, with a single space between the two field values.
- `seek()` the Java API and ye shall find a method that will help you read the middle and last records of the binary file.
- Use a loop to prompt the user for the Trip ID prefixes.

Data: Write your programs to accept the complete data file pathname as a command line argument (for `Prog1A`, that will be the path of the data file, and for `Prog1B`, the path of binary file created by `Prog1A`). The complete `lectura` pathname of our data file is `/home/cs460/spring17/chicagotaxi-nov2016.csv`. `Prog1A` (when running on lectura, of course) can read the file directly from that directory; just provide that path when you run your program. (There's no reason to waste disk space by making a copy of the file in your account.)

Each of the lines in the file contains 23 fields (pieces) of information. Here is a sample line, displayed across four lines for clarity:

```
e73823f3a3453f0dba1aac4933ba76f61ad99ffb,
0cabbef72826456f4ae3c990831eb3e71299165d64d0fe43939030a342a295f7d021e0ce959152684d2034bee8579dc034476f7635455a6b853d97e8d6a3b7e2,
11/30/2016 06:00:00 PM,11/30/2016 06:00:00 PM,660,1.7,,17031081403,,8,$9.25,$2.00,$0.00,$0.00,$11.75,Credit Card,,,,,41.890922026,
-87.618868355,POINT (-87.618868 41.890922)
```

Some file information:

- The field names can be found in the first line of the file; of course, that first line must not be stored in the binary file.
- In some records, such as this sample, some field values are missing (see the adjacent commas?). However, the binary file requires values to maintain the common record length. For missing numeric fields, store -1. For missing strings, store a string containing the appropriate quantity of spaces.
- Please keep in mind that we have not combed through the data to see that it is all formatted perfectly. This is completely intentional, and not (just!) because we are lazy. Corrupt data is a huge problem in data management, and we hope that this file is no exception. We want you to think about how to deal with malformed data, and to ask questions as necessary.

Output: Output details for each program are stated in the Assignment section, above. Please ask (preferably on Piazza) if you need additional details.

Hand In: You are required to submit your completed program files using the `turnin` facility on lectura. The submission folder is `cs460p1`. Instructions are available from the document of submission instructions linked to the class web page. In particular, because we will be grading your program on lectura, it needs to run on lectura, so be sure to test it on lectura. Feel free to split up your code over additional files if doing so is appropriate to achieve acceptable code modularity. Submit all files as-is, *without* packaging them into `.zip`, `.jar`, `.tar`, etc., files.

(Continued...)

Want to Learn More?

- `https://data.cityofchicago.org/d/wrvz-psew` — The City of Chicago data portal viewer for the taxi data. I used the "Filter" option to restrict the data, and "Export" to convert it to a CSV file. Hovering your mouse over the circled-i symbols next to each field label will give you additional information about that field.

Other Requirements and Hints:

- Don't "hard–code" values in your program if you can avoid it. For example, don't assume a certain number of records in the input file or the binary file. Your program should automatically adapt to simple changes, such as more or fewer lines in a file or changes to the file names or locations. For example, we may test your program with a file of just a few records, or even no records. We expect that your program will handle such situations gracefully.
- Once in a while, a student will think that "create a binary file" means "convert all the data into the characters '0' and '1'." The binary I/O functions in Java will read/write the data in binary format automatically.
- Comment your code according to the style guidelines *as you write the code* (not an hour before class!). The requirements and some examples are available from: `http://www.cs.arizona.edu/~mccann/style.html`
- You can make debugging easier by using only a few lines of data from the data file for your initial testing. Try running the program on the complete file only when you can process that reduced data file.
- Late days can be used on each part of the assignment, if necessary. For example, you could burn two late days by turning in Part A two days late, and another by turning in Part B a day late.
- Finally: Start early! File processing can be tricky.

Interpolation Search

Interpolation Search is an enhanced binary search. To be most effective, these conditions must exist: (1) The data is stored in a direct-access data structure (such as a binary file of uniformly-sized records), (2) the data is in sorted order by the search key, (3) the data is uniformly distributed, and (4) there's a *lot* of data. In such situations, the reduction in quantity of probes over binary search is likely to be particularly beneficial given the inherent delay that exists in file accesses. Our data falls short on (3) and (4), but that's OK; the search will still work.

Interpolation Search is just like binary search, with one change: Instead of probing the data at the midpoint (one-half of low index plus high index), we use the following probe index calculation:

$$\text{probe\_index} = \text{low\_index} + \left\lceil \frac{\text{target} - \text{key[low\_index]}}{\text{key[high\_index]} - \text{key[low\_index]}} \cdot (\text{high\_index} - \text{low\_index}) \right\rceil$$

For example, consider a binary file of 60,000 records (indices 0 through 59,999), with keys that range from 100 through 150,000, and a target of 125,000. Thus, `low_index` = 0, `high_index` = 59999, `key[low_index]` = 100, `key[high_index]` = 150000, and `target` = 125000. Our first probe into the file would be into record number $0 + \left\lceil \frac{125000-100}{150000-100} \cdot (59999 - 0) \right\rceil = 49993$.