CSc 460 — Database Design
Fall 2020 (McCann)

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

Program #1: Creating and Exponentially Searching a Binary File

Due Dates:

Part A:	September 3rd, 2020, at the beginning of class
Part B:	September 10th, 2020, 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 formatted text file makes for a nice "shake off the rust" assignment, plus it provides a gentle (?) introduction to binary file processing for those of you who haven't used it before.

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 directly by people and doesn't need to be 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 values that we need to store, and each line's values 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 to provide more direct file I/O.) On the class web page you'll find a sample Java binary file I/O program to help you get started.

Assignment: To discourage procrastination, this assignment is in two parts, Part A and Part B:

Part A Available on `lectura` is a file named `MeteoriteLandings.csv` (see the Data section, below). This is a text file consisting of over 45,000 lines of data describing (you guessed it) meteorite impacts on Earth through 2013. Each landing has 10 fields of information, separated by commas (hence the `.csv` extension – comma–separated values).

Using Java 1.8, write a complete, well–documented program named `Prog1A.java` that creates a binary version of the provided text file's content that is sorted in ascending order by the seventh field of each record (the 'year' field; only use the year part).

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 the file name in your progam; just let your program create the file in the current directory (which is the default behavior).
- Field types are limited to `int`, `double`, and `String`. When necessary, pad strings on the right with spaces to reach the needed length(s) (see the next bullet point). (For example, `"abc␣␣"`, where `"␣"` represents the space character.)
- For each column, all values must consume the same quantity of bytes, so that records have a uniform size. This is easy for numeric columns (e.g., a `double` in Java is always eight bytes), but for alphanumeric columns we don't want to waste storage by choosing an excessive maximum length. Instead, you need to determine the number of characters in each string field's longest value, and use that as the length of each value in that field. 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 new data. You may assume that the field order, types, and quantity will not change.)

(Continued...)

- Because the maximum lengths of the string fields can be different for different input files, you will need to store these lengths somewhere within the binary file so that your Part B program can use them to successfully read it. One possibility is to store the maximum string field lengths at the end of the binary file (after the last data record). This allows the first data record to begin at offset zero in the binary file, which keeps the record location calculations simple for Part B's program.
- How do you test that your binary file is correctly created? Write the first part of Part B! Part B depends on Part A.

Repeating the info at the top of this handout: Part A is due in just one week; start today!

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

1. Reads from the binary file created in Part A (not directly from the provided text file!), and prints to the screen, the name, year (again, just the year part), and GeoLocation fields of the first five records of data, the middle five records (or middle four records, if the quantity of records is even), and the last five records of data. Conclude the output with the total number of records in the binary file, on a new line.

   If the binary file does not contain at least five records, print as many as exist for each of the three groups of records. For example, if there are only two records, print those three fields of both records three times — once as the "first five" records, once as the "middle five," and once as the "last five." Don't forget to also output the quantity of records as the last line of output!

2. Using one or more year values given by the user, locates within the binary file (using exponential binary search; see below) and displays to the screen the same three field values of all records having that year value.

A few details:

- Output the data one record per line, with each field value surrounded by square brackets (e.g., `[1938]`).
- `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 year values; one year per iteration. Terminate the program when the year 0 (zero) is entered.

Data: Write your programs to accept the complete data file pathname as a command line argument (for `Prog1A`, that will be the pathname of the data file, and for `Prog1B`, the pathname of binary file created by `Prog1A`). The complete `lectura` pathname of our data file is `/home/cs460/fall20/MeteoriteLandings.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 CS account.)

Each of the lines in the file contains 10 fields (columns) of information. Here is one of them:

```
South African Railways,23674,Valid,"Iron, IVA","47,000",Found,1/1/1938 0:00,,,
```

There are quite a few data situations that your program(s) will need to handle:

- The field names can be found in the first line of the file. Because that line contains only metadata, that line must not be stored in the binary file. Code your Part A program to read that line and ignore it.

(Continued . . .)

- Some of the name values include non–ASCII UNICODE characters (that's how the accents are included on some of the letters). Use Java's `Normalizer` class to replace UNICODE characters in strings containing them with their corresponding ASCII characters:

```
output = Normalizer.normalize(input, Normalizer.Form.NFKD).replaceAll("[^\\p{ASCII}]", "");
```

- In our example data line, above, you'll notice that two field values containing commas are delimited with double–quotes. This is done to keep such commas from being mistaken as field separators. You need to retain and store those commas in string values written to your binary file, but you are to discard them from numeric values (e.g., convert `"47,000"` to the integer `47000`). Never store the delimiting double–quotes.

- The year field values in the `.csv` file contain the year, but also contain many other characters that are extraneous. When writing the year to the binary file, only write the year, as an integer. For example, `1/1/1938 0:00` in the `.csv` file should be written as the integer `1938` in the binary file.

- In some records, such as the above sample, some field values are missing (see the adjacent commas and the end?). However, the binary file requires values be written for all fields, to maintain the common record length. For missing numeric fields, store -1 or -1.0 (to match the field's type). For missing strings, store a string containing the appropriate quantity of spaces.

- Finally, 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 because we are lazy. Corrupt and oddly–formatted data is a huge headache in data management and processing, as illustrated by the preceding special cases that your program needs to handle. We hope that this file holds a few additional surprises, because we want you to think about how to deal with additional data issues you may find in the CSV file, and to ask questions of us about them as necessary.

Output: Basic 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.

Want to Learn More?

- `BinaryIO.java` — New to binary file IO, or need a refresher? This example program is available from the class web page.

- https://data.nasa.gov/Space-Science/Meteorite-Landings/gh4g-9sfh — NASA's Open Data Portal's Meteorite Landings page, the source of the data we're providing for this assignment. You don't need to visit this page; we're providing it to be transparent, and in case you might be curious to learn more.

- https://meteoritical.org/ — NASA got the data from the Meteoritical Society.

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. As mentioned above, the characteristics of the fields will not change.

(Continued...)

- Once in a while, a student will think that "create a binary file" means "convert all the data into the characters '0' and '1'." Don't do that! The binary I/O functions in Java will read/write the data in binary format automatically.
- Try this: Comment your code according to the style guidelines *as you write the code* (not just before the due date and time!). Explaining in words what your code must accomplish *before* you write that code is likely to result in better code sooner. The documentation requirements and some examples are available here: http://u.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 a few reduced data files. The data file is a plain text file; you can open it with any text editor.
- Late days can be used on each part of the assignment, if necessary, but we are limiting you to at most two late days on Part A. For example, you could burn one late day by turning in Part A 18 hours late, and three more by turning in Part B two and a half days late. Of course, it's best if you don't use any late days at all; you may need them later!
- Finally: **Start early!** File processing can be tricky.

Exponential Binary Search

Exponential Binary Search is an extension of normal binary search originally intended for searching unbounded data, but it works for bounded data, too, such as might be stored in a binary file. The algorithm has two stages:

Stage 1: For $i = 0, 1, \ldots$, probe at index $2(2^i - 1)$ until the index is invalid or an element $\geq$ the desired target is found. (If the probed element equals the target, the search is complete.)

Stage 2: Perform binary search on the range of indices from $2(2^{i-1} - 1) + 1$ to $\min(2(2^i - 1) - 1, \text{largest index})$, inclusive.

To better understand how this works, sketch an array of 16 integers on a piece of paper, and imagine that you're searching for an item toward the far end of the array.

The algorithm can be extended to have as many repetitions of Stage 1 as desired to further narrow the range that Stage 2 needs to search. For our purposes, this basic version is adequate.

Reference: Bentley and Yao, "An Almost Optimal Algorithm for Unbounded Searching," Information Processing Letters 5(3), 1976, pp. 82-87. https://www.slac.stanford.edu/cgi-bin/getdoc/slac-pub-1679.pdf