CSc 460 — Database Design
Spring 2024 (McCann)

https://cs.arizona.edu/classes/cs460/spring24/

Program #1: Creating and Exponentially–Searching a Binary File

Due Dates:

Part A	:	January 17th, 2024, at the beginning of class
Part B	:	January 24th, 2024, at the beginning of class

Overview: Program #2 will create an index on a binary file that this program creates. 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 (if not always the programmer!) 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 related *fields* is called a *record*). Making this happen in Java requires a bit of effort. Alongside the PDF version of this handout on the class web page, you'll find a sample Java binary file I/O program that shows the basics of working with binary files.

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

Part A Available on `lectura` is a file named `lunarcraters.csv` (see the Data section, below, for the location). This is a CSV text file from the Lunar and Planetary Institute consisting of over eight thousand lines of data describing impact craters on Earth's moon. Each lunar crater is described by ten fields of information, separated by commas (hence the `.csv` extension — comma–separated values).

Using Java 16 or earlier, write a complete, well–documented program named `Prog1A.java` that creates a binary file version of the provided text file's content whose records are stored in the same order as provided in the data file (which is sorted alphabetically by the names of the craters).

Some initial details (the following pages have many more!):

- For an input file named `file.csv`, name the binary file `file.bin`. (That is, keep the file name, but change the extension.) Do not prepend a path to the file name in your code; just let your program create the binary file in the current directory (which is the default behavior).
- Field types are limited to `double` and `String`. Specifically, if a field seems to contain real #s, it does. When necessary, pad strings on the right with spaces to reach the needed length(s) (see also 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, your program needs 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 define maximum string lengths, so we need to code defensively to accommodate changes in data. You may assume that the data file's field order, data types, and quantity of fields 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 the binary file. How you do this is up to you. 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 a bit simpler for Part B's program.
- How do you know that your binary file is correctly created? Here's one way: Write the first part of Part B! (Part B depends on Part A's output.)

Read the rest of this handout before starting Part A! Remember, Part A is due in just one week; start today!

Part B Write a second complete, well–documented Java 16 (or earlier) program named `Prog1B.java` that performs both of the following tasks:

1. Read directly from the binary file created in Part A (not from the provided `.csv` file!), and print to the screen the content of the Crater name, Diameter, Apparent depth, and Age fields of the first three records of data, the middle three records (or middle four records, if the quantity of records is even), and the last three records of data. Next, display the total number of records in the binary file, on a new line. Conclude the output with a list, in descending order by depth, of the names and apparent depths of the ten craters having the largest apparent depths. If there are ties, display all of the craters with apparent depths that tie the depth of the tenth–deepest crater (thus, the list may be longer than ten). The TAs will not be doing 'script grading,' so you do not need to worry about generating a specific output format, but it should be easily readable by human beings.

   If the binary file does not contain at least ten records, print as many as exist for each of the four groups of records. For example, if there are only two records, print the fields of both records four times — once as the "first three" records, a second time as the "middle three," a third time as the "last three," and a fourth time (in the appropriate order) as the list of "ten." (And, of course, also display the total quantity of records.)

2. Allow the user to provide, one at a time, zero or more crater names, and for each name locate within the binary file using exponential binary search (see below), and display to the screen the same four field values (name, diameter, apparent depth, and age) of the record identified by the given name, or display a polite 'not found' message.

A few more Part B details:

- Many craters do not have age strings. For them, display "null" as the age.
- Output the data one record per line, with each field value surrounded by square brackets (e.g., `[Lee M][73.33][2.766][null]`).
- `seek()` the Java API and ye shall find a method that will help you find the middle and last records of the binary file. (See also the previously–mentioned `BinaryIO.java` example.)
- Use a loop to prompt the user for the crater names, one crater name per iteration. Terminate the program when either "e" or "E" is entered as the name.

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/spring24/lunarcraters.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.)

(Continued . . .)

Each of the lines in the file contains ten fields of information. Here is an example line:

```
Lee M,72.33,-29.83,-39.70,62.20,39.28,2.766,4694.64,0.00,
```

This example demonstrates a few of the 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 ignore that line.
- Many `.csv` files have missing field values (usually represented with adjacent commas (e.g., ",,"), or lines that end with a comma, to show that no data exists for that field). In this data, the only missing data values we know of are many of the ages (the last column), which is why the example line above ends with a comma. Should you find any other fields with missing data, let us know and we'll tell you how to handle those situations.
- Finally, be aware that we have not combed through the data to see that it is all formatted perfectly. This is completely intentional! Corrupt and oddly–formatted data is a huge headache in data management and processing. 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 us questions about them as necessary.

Output: Basic output details for each program are stated in the Assignment section, above. Please ask (preferably in a public post 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 for `turnin` 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 thoroughly on lectura. Feel free to split up your code over additional files if doing so is appropriate to achieve good 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 and its data file are available from the class web page and from the `/home/cs460/spring24/` directory on lectura.
- https://www.lpi.usra.edu/scientific-databases/ — This is the source of our data; scroll down to the "Lunar Impact Crater Database (2015)" link. The full data file has many additional columns that we removed so that you wouldn't have to worry about them. (You're welcome!) You don't need to visit this page; we're providing it in case you're interested in learning more.

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 file locations. Another example: We may test your program with a file of just a few data records or even no data records. We expect that your program will handle such situations gracefully. As mentioned above, the characteristics of the fields (types, order, etc.) will not change.
- 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.

(Continued . . .)

- Not a fan of documenting code? Try this: Comment your code according to the style guidelines handout *as you write the code* (rather than 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: https://mccann.cs.arizona.edu/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 CSV files are plain text files; you can view them, and create new ones, 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!** There's a lot for you to do here, and 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 ordered 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