CSc 460 — Database Design
Spring 2023 (McCann)

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

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

Due Dates:

Part A	:	January 19th, 2023, at the beginning of class
Part B	:	January 26th, 2023, at the beginning of class

Overview: In Program #2, 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 (if not 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 *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 `wbbrosters2022.csv` (see also the Data section, below). This is a text file consisting of nearly five thousand lines of data describing the rosters of the women's Division I college basketball teams, as of mid–December of 2022. Each player is described by thirteen 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 sorted in ascending numeric order by the values of the 'player_id' field.

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 file in the current directory (which is the default behavior).
- Field types are limited to `int` and `String`. Specifically, if a field looks to contain integers, 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., an `int` in Java is always four 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 define maximum string lengths, so we need to code defensively to accommodate different 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. 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 test that your binary file is correctly created? Here's one way: Write the first part of Part B! (Part B depends on Part A.)

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 'player_id,' 'name,' and 'hometown_clean' 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. Next, display the total number of records in the binary file, on a new line. Conclude the output with a list, in descending order, of the five most common state abbreviations and the number of times each appeared in the 'state_clean' field. 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 five records, print as many as exist for each of the three groups of records. For example, if there are only two records, print the fields of both records three times — once as the "first five" records, once as the "middle four," and once more as the "last five." The same applies to the state occurrences. (And, of course, also display the total quantity of records.)

2. Allow the user to provide, one at a time, zero or more player_id values, and for each locate within the binary file using interpolation search (see below), and display to the screen the same three field values (player_id, name, and hometown_clean) of all records having the given player_id value.

A few details:

- Output the data one record per line, with each field value surrounded by square brackets (e.g., `[14551][Cate Reese      ][CYPRESS, TEXAS          ]`).
- `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 player_id values, one value per iteration. Terminate the program when the value -1 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/spring23/wbbrosters2022.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 thirteen fields of information. Here is an example line:

```
361,Long Island,3328,Dia Dénes,14,6,1,FORWARD,Freshman,0,"BUDAPEST, HUNGARY",,HUNGARY
```

(Continued . . .)

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.
- Some of the string values include non–ASCII UNICODE characters, such as "é" in "Dia Dénes." 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 the example data, above, you'll notice that field values containing a comma are delimited with double–quotes. This is done to keep such commas from being mistaken as field separators. Do not store the enclosing double–quotes in your binary file (they aren't data), but do retain and store the commas found within string values in your binary file.
- In some records, some field values are missing or may be malformed (in the example data line, notice the pair of adjacent commas). However, the binary file requires values be written for all fields, to maintain the common record length. For missing or malformed player_id values, just skip the line (that is, if such a value is not an integer, do not store any part of its line in the binary file). For missing numeric fields, store the value 0. For missing strings, store a string containing the appropriate quantity of spaces.
- 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, 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 us questions about them as necessary.

Output: Basic output details for each program are stated in the Assignment section, above. Please ask (preferably in public posts 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 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/spring23/` directory on lectura.
- https://github.com/Sports-Roster-Data/womens-college-basketball — This is the source of our data. The full data file has many more fields and lines; we cut it down. (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 records, or even no 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...)

- 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 view it, 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.

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 $0 + \left\lceil \frac{125000-100}{150000-100} \cdot (59999 - 0) \right\rceil = 49993$.