CSc 460 — Database Design
Spring 2022 (McCann)

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

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

Due Dates:

Part A	:	January 19th, 2022, at the beginning of class
Part B	:	January 26th, 2022, 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. 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 `Offsets-Database.csv` (see also the Data section, below). This is a text file consisting of over 5,800 lines of data describing global carbon offset projects, current as of last month. Each project has 13 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 tenth field of each record (the 'Total Credits Issued' 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 put a path on the file name in your program; 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, the last four fields are all to be stored as integers; the rest stored as strings. 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., 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 new 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 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.)

Remember: The rest of this handout has more info that will help you with Part A, and 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 from the binary file created in Part A (not directly from the provided text file!), and print to the screen the content of the Project ID, Project Name, and Total Credits Issued 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. The TAs will not be doing script–grading, so you do not need to worry about generating a specific output format.

   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 three fields of both records three times — once as the "first five" records, once as the "middle four," and once more as the "last five." And, of course, also display the total quantity of records.

2. Allow the user to provide, one at a time, zero or more Total Credits Issued values and for each locate within the binary file using ternary search (see below), and display to the screen the same three field values (Project ID, Project Name, and Total Credits Issued) of all records having the given Total Credits Issued value.

A few details:

- Output the data one record per line, with each field value surrounded by square brackets (e.g., `[CAR994    ][Dairy Dreams            ][65502]`).
- `seek()` the Java API (or see the `BinaryIO.java` example program) 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 Total Credits Issued values, one value per iteration. Terminate the program when -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/spring22/Offsets-Database.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 13 fields (columns) of information. Here is one example, displayed across multiple lines for clarity:

```
VCS594,"TIST Program in Kenya, VCS 001",Registered,Forestry & Land Use,
Afforestation/Reforestation,AR-ACM0001,Sub Saharan Africa,Kenya,,"61,903",
"61,853",50,2010
```

(Continued . . .)

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 ignore that line.
- Some of the strings values include non–ASCII UNICODE characters. 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 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), and also ignore commas found within numeric fields (that is, store `"12,345"` as the four–byte integer 12345), but do retain and store the commas found within string values into your binary file.
- In some records, some field values are missing (notice the pair of adjacent commas in the example). However, the binary file requires values be written for all fields, to maintain the common record length. 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 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 is available from the class web page and from `/home/cs460/spring22/BinaryIO.Java` on lectura.
- https://gspp.berkeley.edu/faculty-and-impact/centers/cepp/projects/berkeley-carbon-trading-project/offsets-database — This is the source of our data. The full data file has many more fields; we cut it down to just 13. (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. 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 (types, order, etc.) 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 view 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!** There's a lot for you to do here, and file processing can be tricky.

Ternary Search

Ternary Search is a slightly enhanced binary search that has the expectations of binary search — sorted data in a direct–access data structure (such as an array or a binary file of uniformly–sized records). The slight enhancement is this: While binary search probes a single location at a time, ternary search probes two locations. Those locations are one–third and two–thirds of the way into the search region.

For example, assume that the initial search region has a low index of 0 and a high index of 100. For this example, assume that one–third of the way into this region is index $0 + \left\lfloor \frac{100-0}{3} \right\rfloor = 0 + \lfloor 33.\overline{3} \rfloor = 33$ and two–thirds is $0 + \left\lfloor \frac{2 \cdot (100-0)}{3} \right\rfloor = 0 + \lfloor 66.\overline{6} \rfloor = 66$. Our target value could be found at one (or even both!) of those two indices, or in the three regions they define.

Let's continue the example by assuming that, if the target is in the data, it must be between the one–third and two–thirds points. The next pair of probe indices would be $34 + \left\lfloor \frac{65-34}{3} \right\rfloor = 34 + \lfloor 10.\overline{3} \rfloor = 44$ and $34 + \left\lfloor \frac{2 \cdot (65-34)}{3} \right\rfloor = 34 + \lfloor 20.\overline{6} \rfloor = 54$.

Like binary search, ternary search stops either when the target value is found at one of the probe indices, or when there is no region left to search. Also like binary search, ternary search can be tricky to code correctly!