CSc 460 — Database Design
Spring 2026 (McCann)

https://cs.arizona.edu/classes/cs460/spring26/

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

Due Dates:

Part A	:	January 22nd, 2026, at the beginning of class
Part B	:	January 29th, 2026, 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 done 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 `Dataset2.csv` (see the Data section, below, for the location). This is a CSV text file containing data on bat caves around the globe, as collected by Krizler C. Tanalgo et. al. Each cave is described by 11 fields of information, separated by commas (hence the `.csv` extension — comma–separated values).

Using Java 25 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 order by Data.entry. The choice of sorting algorithm used by `Prog1A.java` for this purpose is up to you.

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 `int`, `double`, and `String`. Specifically, if a field seems to contain numbers, 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 the 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, your program needs to determine the number of characters in each `String` field's longest value, and use that as the length of each string of each record for that field. This must be done as part of each execution of the program. (Why? The data doesn't define maximum string lengths, so we need to code defensively to accommodate changes in field values. You may assume that the data file's field order, data types, and quantity of fields will not change, but the quantity of rows may.)

(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 25 (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 Dataset_sequence_ID, Country.record, and Cave.site fields of the first four records of data, the middle four records (or middle five records, if the quantity of records is odd), and the last four records of data. Next, compute from the file size, and display on a new line, the total number of records in the binary file. Conclude the output with a list of the Country.record, Cave.site, and Latitude values of the ten distinct cave sites that are furthest from the equator. (That is, no single cave site is to appear more than once.) Order the list in descending order of distance (e.g., the first cave listed should be the most distant). If there are ties with the tenth most distant cave, lengthen the list if necessary so that all of those ties are displayed. The TAs will not be doing 'script grading,' so you do not need to worry about generating an exact output format, but your output should be easily readable by human beings. A few more output details are given below.

   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 appropriate fields of both records four times — once as the "first four" records, a second time as the "middle four," a third time as the "last four," 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 Data.entry values, and for each value locate within the binary file using exponential binary search (see below), and display to the screen, the Country.record, Cave.site, and Species.name fields identified by the given Data.entry, or display a polite 'not found' message.

A few more Part B details:

- Some fields may not have values. For missing numeric field values, store and display -1000. For missing string field values, display the string "null".
- Output the data one record per line, with each field value surrounded by square brackets (e.g., `[2031][United States][Fixin-to-die- Cave]`).
- `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 Data.entry values, one Data.entry value per iteration. Terminate the program when -1000 is entered as the value.

Data: Write your programs to accept the complete data file pathname plus filename (including the extension) 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 pathname of our data file is `/home/cs460/spring26/Dataset2.csv` on `lectura`. `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 CSV file contains 11 fields of information. Here are two example lines:

```
64,BC 065,S 005,Indomalayan,Asia,TSMB,Laos DPR,Tom Quaie,17.55417,104.825,Rhinolophus thomasi
65,BC 066,S 005,Indomalayan,Asia,TSMB,Laos DPR,"Ban Pontong, Mauang",17.57778,104.825,Cynopterus sphinx
```

These examples demonstrate 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.
- The second sample data line, above, demonstrates 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.
- Many `.csv` files have missing field values (usually represented with leading, adjacent (e.g., ",,"), or trailing commas, to show that no data exists for a field). Store -1000 for any missing numeric values (as numeric fields can't be valueless in binary files). Store the appropriate quantity of spaces for any missing string values.
- 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 file processing. We hope that this file holds a few 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 `.java` 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 your `.java` files as–is, *without* associated IDE files and directories, and *without* manually packaging them into `.zip`, `.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 (`binaryIO.csv`) are available from the class web page and from the `/home/cs460/spring26/` directory on lectura.
- https://observablehq.com/@observablehq/darkcides-bats-in-caves — This is the source of our data subset. The full data file has some 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.
- The corresponding paper's citation: Tanalgo, K.C., Tabora, J.A.G., de Oliveira, H.F.M. et al. DarkCideS 1.0, a global database for bats in karsts and caves. Sci Data 9, 155 (2022). https://doi.org/10.1038/s41597-022-01234-4

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.

(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. Again, see `BinaryIO.java`.
- 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.
- Part A will be worth about 15% of your Program #1 score, and will be 'graded' based on the answers to these two questions: (a) does the submitted code demonstrate that you have made progress on the assignment, and (b) is that submitted code well–structured and well–documented? The TAs will not be running (or even compiling!) your Part A submission. The purpose of having a one–week partial due date is to encourage you to get started. So, ...
- **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 normal 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