CSc 460 — Database Design
Fall 2018 (McCann)

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

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

Due Dates:

Part A:	August 30^{th}, 2018, at the beginning of class
Part B:	September 6^{th}, 2018, 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.

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 items that we need to store, and each line's items 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 convenient file I/O. On the class web page you'll find a sample Java binary file I/O program, which should help get you started.

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

Part A Available from `lectura` is a file named `jan2018ontime.csv` (see also the Data section, below). This is a CSV (Comma–Separated Value) text file consisting of 570,119 lines of 19 values each about airline flights in January of 2018.

Using Java 1.8, write a program named `Prog1A.java` that creates a binary version of the text file's content that is sorted (using a stable sort) in ascending order by the fifth field of each record (the 'FL_NUM' field). Additional 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 it; just create it in the current directory.
- Field types are limited to `int` , `double`, and `String`. Follow the lead of the data. For example, if values of a field are in quotes, assume that the field is a string (and don't store the quote characters).
- For each column, all values must consume the same quantity of bytes. This is easy for numeric columns, but for alphanumeric columns we don't want to waste storage by choosing a huge maximum length. Instead, you need to determine the number of characters in each 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? Your program needs to work on other flight data files, too. You may assume that the field order, types, and quantity will not change.) When necessary, pad strings on the right with spaces to reach the needed length(s). (For example, `"abc␣␣"`, where `"␣"` represents the space character.
- Because the maximum lengths of the text fields can be different for different input files, you will need to store these lengths in the binary file so that your Part B program can access them. One possibility is to store the maximum lengths at the end of the binary file (after the last data record).

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

(Continued...)

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

1. Reads and prints to the screen the UNIQUE_CARRIER, FL_NUM, ORIGIN, and DEST field values of the first five records of data, the middle three records (or middle four records, if the quantity of records is even), and the last five records of data in the binary file. Conclude the output with the total number of records in the binary file, on a new line.
2. Using one or more FL_NUM values given by the user, locates within the binary file (using interpolation search; see below), and displays to the screen, the same four field values of all records having the given FL_NUM value.

Use a loop to prompt the user for the FL_NUM values. Terminate the program when a zero is entered. An example of the expected output format is provided in the Output section, below.

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/fall18/jan2018ontime.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. (That is, there's no reason to waste disk space by making a copy of the file in your account.)

Each of the lines in the file contains 19 fields (columns) of information. Here is one of them:

```
2018-01-01,"EV",20366,"N604QX","2853","SAF","DFW","",,,"","",,"",,1.00,"A",,551.00,
```

Some file information:

- The field names can be found in the first line of the file. Because that line contains metadata, its content must not be stored in the binary file.
- In some records, such as this sample, some field values are missing (see the adjacent commas?). However, the binary file requires values to maintain the common record length. For missing numeric fields, store -1. For missing strings, store a string containing the appropriate quantity of spaces.
- Please keep in mind that we have not combed through the data to see that it is all formatted perfectly. This is completely intentional. Corrupt data is a huge problem in data management. We hope that this file holds a few surprises. We want you to think about how to deal with malformed data, and to ask questions of us as necessary.

(Continued. . .)

Output: Part A's output is the binary file; no on-screen output is required, but you are welcome to add some if you'd care to do so. Here is some sample output from Part B, provided so that you can see the expected formatting:

```
FIRST FIVE RECORDS:
[0]: EV, 2817, BTR, CLT
[1]: EV, 2817, DFW, BTR
[2]: EV, 2819, DFW, BTR
[3]: EV, 2820, DFW, ROW
[4]: EV, 2820, ROW, DFW
---------------------------------
MIDDLE FOUR RECORDS:
[8]: EV, 2823, DFW, BHM
[9]: EV, 2824, FSM, DFW
[10]: EV, 2825, BTR, DFW
[11]: EV, 2826, DFW, MHK
---------------------------------
LAST FIVE RECORDS:
[15]: EV, 2833, MHK, DFW
[16]: EV, 2834, CLT, TYS
[17]: EV, 2834, TYS, CLT
[18]: EV, 2836, BHM, DFW
[19]: EV, 2836, DFW, BHM
---------------------------------
There are 20 record(s) in the file.
Enter a flight number (FL_NUM) that you would like to search i.e. 2817. Enter zero(0) to end your search.
2819
[2]: EV, 2819, DFW, BTR
Enter a flight number (FL_NUM) that you would like to search i.e. 2817. Enter zero(0) to end your search.
2833
[14]: EV, 2833, DFW, MHK
[15]: EV, 2833, MHK, DFW
Enter a flight number (FL_NUM) that you would like to search i.e. 2817. Enter zero(0) to end your search.
2000
No result(s) found
Enter a flight number (FL_NUM) that you would like to search i.e. 2817. Enter zero(0) to end your search.
0
End of search.
End of Program.
```

Note that the records are prefixed with their record numbers. If the number of records is odd, change the word FOUR to THREE.

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 or compression.

Want to Learn More?

- `https://www.bts.gov/topics/airlines-and-airports-0` — The provided sample data was downloaded from the U.S. Department of Transportation's Bureau of Transportation Statistics web site. Other fields are available. If you're curious: Click the "On-Time" drop down, and choose "Airline On-Time Performance Databases". On the next page, click "Download". On the following page, select the desired fields and click "Download" to create CSV.

(Continued. . .)

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 and paths. For example, we may test your program with a file of just two records, or even no records. We expect that your program will handle such situations gracefully.
- 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. See the example program on the class web page.
- Try this: Comment your code according to the style guidelines *as you write the code* (not an hour 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. The requirements and some examples are available from: `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.
- 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.
- Finally: **Start early!** 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 $0 + \left\lceil \frac{125000-100}{150000-100} \cdot (59999 - 0) \right\rceil = 49993$.