MLB Player Index

An index is a system used to make finding information easier. In this case, you will be given data files containing information about players in Major League Baseball and you will build in-memory index structures to support searches of records which match certain criteria.

The player records will contain the following information:

Figure 1: Player Data Fields

	Significance	Type/Format	Comments
	PlayerID	alphanumeric string	unique identifier for this player
	Year of birth	mm/dd/yy	date of player's birth
	Birth place	string	location of player's birth
	Birth country	string	country of player's birth
*	Year of death	mm/dd/yy	date of player's birth
*	Death place	string	location of player's death
*	Death country	string	country of player's death
	First name	string	commonly-used first name for player
	Last name	string	last name for player
	Weight	non-negative number	approximate playing weight (pounds)
	Height	non-negative number	approximate playing height (inches)
	Bats	'R' || 'L'	preferred batting orientation (right or left)
	Throws	'R' || 'L'	preferred throwing hand (right or left)
	Begin	mm/dd/yy	date of first appearance in MLB
*	End	mm/dd/yy	date of last appearance in MLB

Each record will occur on a single line. The record fields will be presented in a comma-separated-values format. The last field in each record will be followed immediately by a newline character. Here is a sample player record in comma-separated format:

```
aaronha01,2/5/1934, Mobile AL,USA,,,Hank,Aaron,180,72,R,R,4/13/1954,10/3/1976
```

The player record files are guaranteed to conform to this syntax, so there is no explicit requirement that you validate the files. On the other hand, some error-checking during parsing may help you detect errors in your parsing logic.

Note that it is logically possible that some data fields will be left empty. Some such fields are marked with an asterisk in the table above, and your implementation must deal gracefully with records in which those fields are, in fact, empty. It is possible other fields will be empty; however, the player ID and name fields will not.

Complete sample data files will be supplied on the course website.

Assignment:

You will implement a system that indexes and provides search features for a file of player records, as described above.

Your system will build and maintain several in-memory index data structures to support these operations:

- Importing new records into a database file of player records.
- Retrieving the player record that matches a given PlayerID.
- Retrieving all player records that match a given player name (first and last).
- Deleting the player record that matches a given PlayerID.

See the section Command File for details. You will implement a single Java program to perform all system functions.

Program Invocation:

The program will take the names of three files from the command line, like this:

```
java PlayerDB <db file name> <command script file name> <log file name>
```

The db file should be created, as an empty file, by your program. If the command script file is not found, the program should log a descriptive error message and exit. If a log file name is not specified, the program should write a descriptive error message to standard output and exit.

Data and File Structures:

To efficiently support PlayerID searches, you will need to build an in-memory index that relates a given PlayerID to a unique, relevant player record within the player data file; we will call this the PlayerID index. The PlayerID is a unique identifier, so there is at most one matching player record for any given PlayerID.

To efficiently support player name searches, you will need to build an in-memory index that relates a given player name (consisting, in the data files, of two separate strings) to a collection of relevant player records within the player data file; we will call this the PlayerName index. Player names are certainly not guaranteed to be unique identifiers and there may be many different player records that contain a given player name. There is no guaranteed limit on the number of player records that may match a given player name, so design your solution accordingly.

You will use an AVL tree as the underlying physical structure for each of the in-memory indices. The AVL used by the PlayerID index will store objects that encapsulate a single PlayerID and a single file offset. The AVL used by the PlayerName index will store objects that encapsulate a single player name (in a form you must decide) and a collection of one or more file offsets.

Neither index object will ever store a complete player record.

When building the index structures, your program will make one complete pass through the player record file. Aside from where specific data structures are required, you may use any suitable standard Java component you like.

Each index object should have the ability to write a nicely-formatted display of itself to an output stream. Consult the course notes for AVL trees for an example.

Note: your implementation will not simply store the complete player records in memory. The file on disk is only place that complete records will exist, aside from a small number of temporary objects created when servicing search requests.

Other System Elements:

There should of course, be a number of classes in your design and implementation. You are expected to apply the object-oriented design principles from your earlier courses, and the evaluation of your solution will take this into account. In particular, you should give careful consideration to providing suitable interfaces for each system component.

Command File:

The execution of the program will be driven by a script file. Lines beginning with a semicolon character (`';'`) are comments and should be ignored. Each non-comment line of the command file will specify one of the commands described below. Each line consists of a sequence of tokens, which will be separated by single tab characters. A newline character will immediately follow the final token on each line. The command file is guaranteed to conform to this specification, so you do not need to worry about error-checking when reading it. The following commands must be supported:

`import<tab><player record file><newline>`

Open the specified file, parse it, add each player record it contains to the database file if that record is not already present, and update the index structures as necessary. When the importing is completed, close the specified file and log a message reporting the number of records that were imported. If the specified file does not exist, log an appropriate error message.

`identify_by_name<tab><first name><tab><last name><newline>`

Log all the PlayerID values for records in the database file that match the specified name; log an informative message if no matches are found.

`show_stats_for<tab><PlayerID>`

Log all the data fields in the unique player record in the database file that matches the given `<PlayerID>`. The display should be well-formatted and clearly labeled. If no matching record exists, log an informative message.

`remove<tab><PlayerID>`

Update the PlayerID index to remove the entry for this PlayerID. Update the PlayerName index to remove all references to this PlayerID. If no matching record exists, log an informative message. There is no requirement that you make any modifications to the database file.

`show_index_for<tab>[PlayerID | PlayerName ]`

Log the contents of the specified index structure in a fashion that makes the internal structure and contents of the index clear. It is not necessary to be overly verbose here, but it would be useful to include information like key values and file offsets where appropriate.

`exit<tab>`

Terminate program execution.

Sample command scripts will be given on the course website. As a general rule, every command should result in some output.

Instrumentation:

Each index (or its aggregated container) must be instrumented so that it logs information about each search it performs. The information should identify each index record that is accessed during the index search, and should be written to the log file.

This can be accomplished in either of two ways. You may modify the interface of the AVL tree so that the search method takes a Java I/O object (whatever you use to front for the log file) as a parameter. Or, you may modify the AVL so that it stores a Java I/O object as a member, set via a constructor or a mutator. Neither approach requires violating the encapsulation of the tree.

Log File Description:

Since this assignment will be graded by TAs, rather than by an automated system, the format of the output is left up to you. Of course, your output should be clear, concise, well labeled, and correct. You should begin the log with a few lines identifying yourself, and listing the names of the input files that are being used.

The remainder of the log file output should come directly from your processing of the command file. You are required to echo each command that you process to the log file so that it's easy to determine which command each section of your output corresponds to. Each command should be numbered, starting with 1, and the output from each command should be well formatted, and delimited from the output resulting from processing other commands. A complete sample log will be posted shortly on the course website.

File Navigation in Java

See the course notes on Java file access on the course website.

AVL Requirements

You are welcome to use relevant code from your textbook as a basis for your implementation. However, this assignment is not "open-web" and you are expected to not use other sources unless specifically authorized by your instructor.

In the interest of efficiency, you are required to use a Java enumerated type to represent the balance factor information in your AVL nodes (this differs from the implementation shown in the code in the Weiss book).

You are also required to implement your AVL tree using formal Java generics.

Administrative Issues:

Project Schedule:

You will find a template for planning a development schedule on the course website. You are required to submit three versions by the due date for the project solution:

Deadline	Description
Sept 14	Initial plan
Sept 25	Revised plan
Final project submission	Summary of the actual schedule you followed, up to the point you completed the project. This should be submitted shortly after you have submitted your final solution for the assignment.

Failure to submit each of these by the specified deadline will result in a deduction of 5% from your final score on the project.

Submission:

You will submit this assignment to the Curator System (read the *Student Guide*), where it will be archived for grading at a demo with a TA.

For this assignment, you must submit a zip (or jar?) file containing all the source code files for your implementation (i.e., the `java` files). Submit nothing else.

In order to correct submission errors and late-breaking implementation errors, you will be allowed up to five submissions for this assignment. Unless you notify us otherwise, the TAs will evaluate the last version you submit.

The Student Guide and link to the submission client can be found at: `http://www.cs.vt.edu/curator/`

Evaluation:

The TAs will evaluate the correctness of your results. In addition, the TAs will evaluate your project for good internal documentation and software engineering practice.

Remember that your implementation will be tested using version 1.6.0_16 of `javac`. If you use a different development platform, it is entirely your responsibility to make sure your implementation works correctly in the lab.

Note that the evaluation of your project will depend substantially on the quality of your code and documentation. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Pedagogic points:

The goals of this assignment include, but are not limited to:

- documented planning of a formal timeline for completing a development project
- implementation of a AVL using formal Java generics
- design of appropriate index classes to wrap the containers
- design of appropriate data objects to store in each index
- understanding how to navigate a file in Java
- understanding how to parse an interesting text file in Java
- creation of a sensible OO design for the overall system, including the identification of a number of useful classes not explicitly named in this specification
- implementation of such an OO design into a working system
- incremental testing of the basic components of the system in isolation
- satisfaction when the entire system comes together in good working order

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the Submitting Assignments page of the course website.

Comments:

1. The two index structures do not store complete player records. Rather, they provide logical "pointers" to where the corresponding records are found in the database file that the program creates.
2. It would be extremely poor design if either index structure were simply an AVL tree; the interface is inappropriate for one thing. Of course, each index structure will make good use of an AVL tree.
3. It's up to you to decide how to deal with player names in your design of the player name index. In the player records, a name consists of two parts, a first name and a last name. You may keep it that way in your index, or you may do something else.
4. There is nothing wrong, either from a design perspective or from a practical perspective, with a container providing the client with access to the data objects it organizes… they belong to the client, not to the tree. Of course, if the client uses that access to modify members that are used in the comparison logic of the data objects, then the logic of the container may be broken… if so, that's entirely the fault of the client.