MLB Player Index

An database 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 for records which match certain criteria.

The player records will contain the following information (abbreviated from the earlier project):

Description of field	Format
Lahman ID; unique to each player.	positive integer
A unique code asssigned to each player. The playerID links the data in this file with records in the other files.	alphanumeric string
Player's birthdate	mm/dd/yyyy
Country where player was born	alphabetic string
State where player was born	alphabetic string
City where player was born	alphabetic string
Player's deathdate	mm/dd/yyyy
Country where player died	alphabetic string
State where player died	alphabetic string
City where player died	alphabetic string
Player's first name	alphabetic string
Player's last name	alphabetic string
Player's weight in pounds	positive integer
Player's height in inches	positive integer
Player's batting hand (left, right, or both)	'R' or 'L'
Player's throwing hand (left or right)	'R' or 'L'
Date that player made first major league appearance (debut)	mm/dd/yyyy or yyyy-dd-mm
Date that player made last major league appearance (finale)	mm/dd/yyyy or yyyy-dd-mm

Figure 1: Player Data Fields

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

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

Note that it is logically possible that some data fields will be left empty. In the sample record above, the player is alive and as of this date, so the death date, country, state and city are all empty. There are still tab characters surrounding those empty fields, though. 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 last 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 last name
- retrieving all player records that match a given year for the first MLB appearance

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 write 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 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 build an in-memory index that relates a given player last name to a collection of relevant player records within the player data file; we will call this the PlayerName index. Player's last 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.

To efficiently support searches for records that match a given year, you will build an in-memory index that relates a given year to a collection of relevant player records withing the player data file; we will call this the PlayerDebut index. There will typically be many player records that match a given debut year.

You will use an AVL tree as the underlying physical structure for the PlayerID index. The AVL used by the PlayerID index will store objects that encapsulate a single PlayerID and a single file offset.

You will use a hash table as the underlying physical structure for the PlayerName index. The hash table used by the PlayerName index will store objects that encapsulate a last name and a collection of one or more file offsets.

You will use an array as the underlying physical structure for the PlayerDebut index. For our purposes, assume that no player's debut year is earlier than 1869 or later than 2014. The array will store objects that encapsulate a collection of one or more player IDs ~~file offsets~~. (There is no need for these objects to store a value for the year if you do this sensibly.)

No 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 and earlier project specifications for examples of how to write a useful display of an AVL tree and a hash table.

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><name of player record file>`
: 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 import 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><last name>`
: Log the record offset, first name and last name value for each record in the database file that matches the specified last 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.

`show_debuts_for<tab><YYYY>`
: Log the record offset, first name and last name of each player whose first appearance in a MLB game was in the given year. The display should be well-formatted and clearly labeled. If no matching record exists, log an informative message.

`show_index_for<tab>[PlayerID | PlayerName | PlayerDebut ]`
: 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. Every command should result in some informative 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. Records can be identified by displaying their "key" value.

This can be accomplished in either of two ways. For example, 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. Do not implement this by having a function build a `String` object representing the structure and then printing the whole thing at once. That is extremely inefficient in any case, and the limitations of the Java `String` class make it even worse. If you insist on using this approach, research the `StringBuilder` class.

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.

AVL and Hash Table 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.

You are not required to use the interfaces specified for the earlier BST and hash table projects, although those would be suitable for this assignment as well. You may use any sensible hash function you like, and employ either probing or chaining in your hash table.

What to turn in and how:

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 jar file containing:

- all the Java source code files for your implementation (i.e., the `java` files)
- a file named `readme.txt` containing any special information that might help us test your submission
- a file named `Pledge.txt` containing the Honor Pledge statement quoted below

Submit nothing else. Extra files will irritate the person evaluating your submission and result in penalties.

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 1.7.0 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.

Maximizing Your Results

Ideally you will produce a fully complete and correct solution. If not, there are some things you can do that will improve your score:

- Make sure your program does not crash on any valid input, even if it cannot produce the correct results. If you ensure that your program processes all the posted test files, it is extremely unlikely it will encounter anything in our test data that would cause it to crash. On the other hand, if your program does crash on any of the posted test files, it will almost certainly do so during our testing.

  We will not invest time or effort in diagnosing the cause of such a crash during our testing. It's your responsibility to make sure we don't encounter such crashes.

  One good idea is to wrap as much of your `main()` function as possible in try-catch structure. Be sure to include a catch clause for `Exception`, and write the exception message to standard output.

- If there is a command that your solution cannot handle, document that in the `readme.txt` file you will include in your submission, and make sure that your solution simply ignores such commands (preferably with a message in the log file indicating it's doing that).

Pedagogic points:

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

- implementation of a AVL using formal Java generics
- implementation of a hash table 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 any index structure were simply a container object; the interface of a generic container is inappropriate for an index. Of course, each index structure will make good use of an underlying container.
3. 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 container. 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.
4. You may substitute a BST for the AVL tree, for a penalty of 30%.