MLB Player Index v 2.0

This project extends Project 1, and all of the specific functional requirements for that assignment are still in force, unless explicitly modified in this specification, and will not be repeated here.

For this assignment, we will add data on batting statistics for MLB players. These records are also supplied in comma-separated format, and contain the following information:

Figure 1: Player Batting Data Fields

Significance	Type/Format	Comments
* PlayerID	alphanumeric string	unique identifier for this player
* Year	4-digit integer	year being reported in record
* Stint	integer	order of appearance in batting order?
* Team	string	player's team ID
* League	mm/dd/yy	player's league ID
* Games	integer	# of games in which player batted
* At bats	integer	# of at bats for player
* Runs	integer	# of runs player scored
* Hits	integer	# of hits player made
* Doubles	integer	#of doubles player made
* Triples	integer	# of triples player made
* Home runs	integer	# of home runs player made
* RBI	integer	# of runs player batted in
Stolen bases	integer	# of bases player stole
Caught stealing	integer	# of times player was caught trying to steal
Base on balls	integer	#of times player was walked
Strikeouts	integer	# of times player struck out
Intentional walks	integer	# of times player was walked intentionally
Hit by pitch	integer	# of times player was hit by a pitch
Sacrifice hits	integer	# of sacrifice hits player made
Sacrifice flies	integer	# of sacrifice fly balls player made
Grnd into DP	integer	# of times player grounded into double play
G played field	integer	# of games player played in field

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

```
ruthba01,1927,1,NYA,AL,151,540,158,192,29,8,60,164,7,6,137,89,,0,14,,,151
```

The player batting 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. Fields that are mandatory are marked with an asterisk character '*' in the table above.

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

Assignment:

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

Your system will build and maintain several in-memory index data structures to support all of the search options from Project 1 and these new operations:

- Importing batting records into a second database file
- Reporting batting data that corresponds to a given PlayerID
- Providing a compact summary of all the batting data for a given player, specified by PlayerID
- Providing a compact summary of all the batting data that corresponds to all indexed PlayerIDs that fall in a given range.

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 <m> <main db file> <batting db file> <command script file> <log file>
```

The db files should be created, as empty files, by your program. If files with the specified names already exist, they should be deleted. 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:

The PlayerID and PlayerName index structures will be retained, without changes (but see below) from Project 1.

You will implement a new index structure to support searches for player batting data. This BattingData index will store records that relate a single PlayerID to a collection of corresponding batting data records are stored in the batting data dB file. In order to make the storage of the B^+-tree nodes on disk more efficient, all batting records corresponding to a particular PlayerID will be stored adjacently in the batting data files that you will be importing. In addition, if multiple imports of batting data files are performed, no two separate files will never contain records for the same PlayerID.

The underlying physical structure for the BattingData index will be a B^+-tree, whose order, *m*, is specified on the command-line as shown above. For full credit, the B^+-tree must be stored in a file on disk.

When importing a file, your program will make one complete pass through the relevant record file and update the relevant indices as needed. 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, and see the log files on the course website for a B^+-tree example.

Optional approach: you may combine the PlayerID and BattingData indices into a single entity, using a B^+-tree as the physical structure. If you do this, all other requirements are still in force.

Note: your implementation will not simply store the complete player records in memory. The files on disk are only places 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. In addition to the commands from Project 1, the following commands must be supported:

`import_batting<tab><batting data file><newline>`

Open the specified file, parse it, add each batting data record it contains to the batting data dB 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.

`show_batting_details_for<tab><PlayerID>`

For the specified PlayerID, log the following data fields, for each relevant year: year, team ID, games, at bats, singles, doubles, triples, home runs, and batting average.

`show_batting_summary_for<tab><PlayerID>`

For the specified PlayerID, log the following data: number of years played, total games, total at bats, total singles, total doubles, total triples, total home runs, and overall batting average.

`show_batting_summary_for_range<tab><PlayerID1><tab><PlayerID2>`

Show the same results as specified for the previous command, but do so for every individual PlayerID in the index that falls between PlayerID1 and PlayerID2, inclusive.

`show_index_for<tab>[PlayerID | PlayerName | Batting ]`

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.

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

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.

B^+-tree 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.

For full credit, you must store the B^+-tree nodes in a disk file and retrieve tree nodes as needed to perform searches.

Recall that the entire justification for the B^+-tree structure is that the "wide" nodes enable efficient disk access times, but that will only be true if an entire node can be retrieved from disk and written to disk in a single read or write transaction. The easiest way to achieve that is to make sure that all of the B^+-tree internal nodes occupy the same number of bytes on disk, and so do the leaf nodes (although they will be different sizes).

The comment earlier that all the batting data records for a given player will be stored consecutively in the batting data files implies that you only need to store one file offset to batting data for each PlayerID. (You might want to store some additional information as well…) So, the data records that are stored in the internal nodes of the B^+-tree should all be the same size, and all the data records stored in the leaf nodes should be the same size, no matter how you handle your design…

… if you store (in the B^+-tree file) the numeric values in binary format and you store each of the PlayerIDs by using the same number of characters. You may assume that no PlayerID contains more than 10 characters.

You should give careful thought to the design of the B^+-tree nodes (on disk), and the overall organization of the B^+-tree (on disk) before you commit too much of your design to code.

Finally, it is necessary to implement a Java class that represents the interface to the B^+-tree and the in-memory portion of the B^+-tree while it is being used. You are also required to implement your in-memory representation of the tree using formal Java generics.

Project 1 Tie-in

This project retains all of the functionality of Project 1, except for the requirement for instrumenting your search paths, and so we will be retesting those things for Project 3, possibly with different data.

If your solution for Project 1 failed some of the tests, but your solution for Project 3 passes the corresponding tests, we will give you back 50% of the relevant points you lost for that test on Project 1.

On the other hand, if you failed tests on Project 1 and do not fix the problem, then you will probably lose points during the testing of Project 3.

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
TBA	Initial plan
TBA	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 B^+-tree using formal Java generics (for the in-memory representation)
- 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.