CS3114 (Fall 2014)
PROGRAMMING ASSIGNMENT #2

Due Wednesday, October 15 @ 11:00 PM for 100 points

Due Tuesday, October 14 @ 11:00 PM for 10 point bonus

Updated: Tuesday 9/23

Assignment:

This project builds on Project 1. In Project 1, the hash table was great for keeping you from storing into the memory pool multiple copies of a given artist name or a given song name. But there was no support to answer questions like "What are all of the songs by artist X?" or "What artists recorded the song (with name) Y?". In this project you will add a data structure that supports **range queries**, so that these questions can be answered. The hash tables and the memory manager will be almost unchanged from Project 1.

Invocation and I/O Files:

The program would be invoked from the command-line as:

```
java SearchTree {initial-hash-size} {block-size} {command-file}
```

The name of the program is `SearchTree`. Parameter `{initial-hash-size}` is the initial size of the hash table (in terms of slots). Parameter `{block-size}` is the initial size of the memory pool (in bytes). The meaning of the parameters is identical to Project 1.

Your program will read from text file `{command-file}` a series of commands, with one command per line. The program should terminate after reading the end of the file. The formats for all commands are identical to Project 1. However, these new commands are added.

list `{artist|song} {name}`

If the first parameter is `artist` then all songs by the artist with name `{name}` are listed. If the first parameter is `song` then all artists who have recorded that song are listed. They should be listed in the order that they appear in the 2-3$^+$ tree.

delete `{artist-name}<SEP>{song-name}`

Delete the specific record for this particular song by this particular artist. This means removing two records from the 2-3$^+$ tree, undoing the insert. If this is the last instance of that artist or of that song, then it needs to be removed from its hash table and the memory pool. In contrast, the `remove` command removes all instances of a given artist or song from the 2-3$^+$ tree as well as the hash table and memory pool.

print `tree`

You will print out a pre-order traversal of the 2-3$^+$ tree as follows. First, before printing the tree, you should print out on its own line: "`Printing 2-3 tree:`". Each node is printed on its own line. Each node is indented by its depth times two spaces. So the root is not indented, its children are indented 2 spaces, its grandchildren 4 spaces, and so on. Internal nodes list the (one or two) placeholder records (both the key and the value for each one). That is, if an internal node stores a key (a handle) whose corresponding string is at position 10 in the memory pool, and a value handle with memory pool position 20, then you would print 10 20. Leaf nodes print, for each record it stores, the two handle positions. So they will print either 2 or 4 numbers. The numbers should be separated by a space.

The 2-3$^+$ Tree

You will implement a 2-3$^+$ tree to support the search query. The data records to be stored are objects of the class `KVPair`, where the key is a handle, and the value is another handle. Your Handle class should be made to support the Comparable interface, and the actual thing that will be compared is the start position for the string in the memory manager's array. In other words, ultimately, the keys for the strings are their locations in the memory manager's array. However, to simplify implementation, we want to avoid storing records with duplicate keys. So while we use the "key" field when comparing records, if they have the same value in the key (because they refer to the same artist, or to the same song), we will then use the value of the handle in the value position of the KVPair to break the tie. For example, if one record has handle 10 in the key field and and handle 30 in the value field, while another record has handle 10 in the key field and handle 40 in the value field, then the first one would appear in list of leaf nodes before the second.

Whenever an insert command is called with a song/artist pair, the handles for these two strings will then be used to enter **two** new entries into the 2-3$^+$ tree. If the song handle is named `songHandle` and the artist's name handle is named `artistHandle`, then you will cerate and insert a `KVPair` object with the `songHandle` as the key and the `artistHandle` as the value field. You will then create and insert a second `KVPair` object with the `artistHandle` as the key and the `songHandle` as the value field. However, be sure not to insert a duplicate artist/song record into the database. If there already exists a `KVPair` object in the 2-3$^+$ tree with those same handles, then you will not add the song pair again.

When a `list` command is processed, you will first get the handle for the associated string from the hash table. You will then search for that handle in the 2-3$^+$ tree and print out all records that have been found with that key value.

When a `remove` command is processed, you will be removing all records associated with a given artist or song name. To (greatly!) simplify removing the records from the 2-3+ tree, you will remove them one at a time. So, you will search for the first record matching the corresponding handle for the name that you want to delete, then call the (single record) delete operation on it, and repeat this process for as long as there is a record matching that handle in the tree. Also, whenever you remove a particular record (say, an artist/song pair), you will immediately after remove the corresponding song/artist record. Finally, whenever you remove the last instance of either an artist or a song, you always should remove the associated record from the hash table and the string from the memory pool.

When implementing the 2-3$^+$ tree, all major functions (insert, delete, search) must be implemented recursively. You may **not** store a parent pointer for the nodes. Your nodes must be implemented as a class hierarchy with a node base class along with leaf and internal node subclasses. Internal nodes store two values (of type `KVPair`) and three pointers to the node base class. Leaf nodes store two `KVPair` records.

Programming Standards:

You must conform to good programming/documentation standards. Note that Web-CAT will provide feedback on its evaluation of your coding style. While Web-CAT will not be used to define your coding style grade, the grader will take note of Web-CAT's style grade when evaluating your style. Some specific advice on a good standard to use:

- You should include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment should describe what your program does; don't just plagiarize language from this spec.
- You should include a comment explaining the purpose of every variable or named constant you use in your program.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as "camel casing".
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or the codebase provided by the instructor. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.

Deliverables:

You will submit your project through the automated Web-CAT server. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will **not** be given a copy of grader's test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

When structuring the source files of your project (be it in Eclipse as a "Managed Java Project," or in another environment), use a flat directory structure; that is, your source files will all be contained in the project root. Any subdirectories in the project will be ignored. If you used a makefile to compile your code, or otherwise did something that won't automatically compile in Eclipse, be sure to include any necessary files or instructions so that the TAs can compile it.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix `zip` command) or else a tar'ed and gzip'ed archive. Either way, your archive should contain all of the source code for the project, along with any files or instructions necessary to compile the code. If you need to explain any pertinent information to aid the TA in the grading of your project, you may include an optional "readme" file in your submitted archive.

You are permitted (and ecouraged) to work with a partner on this project. When you work with a partner, then **only one member of the pair** will make a submission. Be sure both names are included in the documentation. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used source code obtained from another student,
//   or any other unauthorized source, either modified or
//   unmodified.
//
// - All source code and documentation used in my program is
//   either my original work, or was derived by me from the
//   source code published in the textbook for this course.
//
// - I have not discussed coding details about this project with
//   anyone other than my partner (in the case of a joint
//   submission), instructor, ACM/UPE tutors or the TAs assigned
//   to this course. I understand that I may discuss the concepts
//   of this program with other students, and that another student
//   may help me debug my program so long as neither of us writes
//   anything during the discussion or modifies any computer file
//   during the discussion. I have violated neither the spirit nor
//   letter of this restriction.
```

Programs that do not contain this pledge will not be graded.