CS3114 (Fall 2011)
PROGRAMMING ASSIGNMENT #4

Due Tuesday, December 10 @ 11:00 PM for 150 points

Due Monday, December 9 @ 11:00 PM for 15 point bonus

Updated: 12/05/2013

Please note that this assignment is worth more than the others. It is not intended that this assignment be more difficult than the others. Rather, this should be an opportunity to make up for any past problems. However, the more defects that were in your prior projects, the more work you will have now.

In this project, you will re-implement a Bintree for storing watcher records from Project 2, where this Bintree is stored in a disk file. The context will be a simplified version of Project 2, in that there will be no earthquake service. There will simply be a series of commands to process (similar to Project 2's Watcher service in principle). The main operations will be inserting watchers, deleting watchers, locating all watchers within a certain distance of a search point, and outputting a traversal of the Bintree for debugging purposes. There will be no heap and no BST for this project.

While the functionality on the Bintree is roughly the same as in Project 2, this time the Bintree and its Watcher records will reside on disk. A buffer pool (using the LRU replacement strategy) will mediate access to the disk file, and a memory manager will decide where in the disk file to store the Bintree nodes as well as the Watcher records. Another way to look at this project: You will write a memory manager, which is a way to store data in a big array. You will store the Bintree nodes and watcher recrods in the memory manager's array. The memory manager's array is really a disk file, and so you will use a buffer pool to manage the disk I/O.

You should read the OpenDSA Memory Management chapter to prepare for implementing this project.

Input and Output:

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

```
java DiskBintree <command-file-name> <numb-buffers> <buffersize>
```

The `<command-file-name>` parameter is the name of the command input file that provides a series of commands to process. This is rather similar to the Watcher file from Project 2, but does not use the WatcherFile interface. The `<numb-buffers>` parameter is the number of buffers in the buffer pool, and will be in the range 1–20. Parameter `<buffersize>` is the size of a buffer in the buffer pool (and therefore determines the amount of information read/written on each disk I/O operation).

You will need to create and maintain a disk file which stores the memory manager's array. The buffer pool acts as the intermediary for this file. The name of this disk file must be "p4bin.dat". After completing all commands in the input file, all unwritten blocks in the buffer pool should be written to disk, and "p4bin.dat" should be closed, **not deleted**.

Your buffer pool should need only three changes from Project 3. First, the size for the buffers is determined by the command line parameter. Second, since the data entities stored on disk are of variable size (differerent tree node types, and the Watcher records contain a variable-length string), your buffer pool must be able to handle storing messages that span block boundaries, or that might even span multiple blocks. Third, you need to be careful not to read in blocks from the file that do not yet exist. This can happen when the memory manager decides to "grow" the size of its memory pool (and therefore directs the buffer pool to write a message to a part of the file that does not yet exist). file that does not yet

The command file will contain a series of commands (some with associated parameters, separated by spaces), one command for each line. The commands will be read from the file given in command line parameter 1, and the output from the command will be written to standard output. The format and interpretation for the commands will be similar Project 2, but there are some differences.

`add <x> <y> <name>`

Adds a watcher record to the bintree. As in Projects 1 and 2, the X and Y coordinates are longitude and latitude, respectively, and so range between (-180, 180) and (-90, 90), resprectively.

On successfully adding a record, your program should print

`<name> <x> <y> is added to the bintree`

If the record duplicates the X and Y coordinates of an existing record then you should reject it and print

`<name> <x> <y> duplicates a watcher already in the bintree`

Next is the delete command, which looks as follows.

`delete <x> <y>`

If successful, print

`<name> <x> <y> is removed from the bintree`

If a record with those coordiantes does not exist, then print

`There is no record at <x> <y> in the bintree`

The search command looks as follows.

`search <x> <y> <radius>`

First you will print out the line

`Search <x> <y> <radius> returned the following watchers:`

Then you will print out each record that lies in the search circle listing its name and position, one record to each line. After you have completed searching, you will print out

`Watcher search caused <number> bintree nodes to be visited.`

Finally, there is the `debug` command. You will print a traversal for the bintree nodes in the same format as you used for Project 2. After that, you will print the block IDs of the blocks currently contained in the bufferpool in order from most recently to least recently used. Note that "Block ID" simply refers to the block number, starting with 0. Thus, if the block size is 1024 bytes, then bytes 0-1023 are in Block 0, bytes 1024-2047 are in Block 1, and so on.

After you have processed all of the commands in the watcher file, you should print out summary statistics for the number of cache hits and misses and the number of disk reads and writes, just as you did for Project 3.

Implementation:

All operations that traverse or descend the Bintree structure must be implemented recursively. The Bintree itself will have its nodes stored in the memory manager's space on disk, and not in main memory. This is the primary difference from Project 2. The nodes will be of variable length, and where a node is stored on disk will be determined by the memory manager. When implementing the Bintree nodes, access to a node (perhaps you did this in Project 2 by using a "node.getchild()" method) will now mean a request to the memory manager to return the node contents from the memory pool. Creation or alteration of a node will require writing to the memory pool. From the point of view of the memory manager and the buffer pool, communications are in the form of variable-length "messages" that must be stored.

The location for placing the next message within the memory pool should be determined using circular first fit. The list of the free blocks will be maintained in main memory, and adjacent free blocks should be merged together.

Initially, the memory pool (and the "p4bin.dat" file) should have length 0. Whenever a request is made to the memory manager that it cannot fulfill with existing free blocks, the size of the memory pool should grow by one (or more if necessary) disk blocks (of size `<buffersize>`) to meet the request.

The memory manager will be managing data residing in the memory pool, and this memory pool will reside on disk, but the memory manager does not actually have direct access to the disk. All disk access is through the buffer pool. Thus, the flow of control for a node access is as follows: The Bintree will request a "message" from the memory manager via a handle, the memory manager will ask the buffer pool for the data at the physical location stored in the handle, the buffer pool will give the contents of the "message" to the memory manager, which will in turn give the contents of the "message" back to the Bintree.

Note that the memory manager will actually add at the beginning of each message that it stores a two-byte field to indicate the length of the message before storing it in the memory pool. Handles will always be a 4-byte quantity that indicates the starting byte position of the message in the disk file.

The layout of the Bintree node messages sent to the memory manager **MUST** be as described in the following paragraphs.

Internal nodes will store 9 bytes: a one-byte field used to distinguish internal from leaf nodes, followed by two 4-byte fields to store handles for the two children.

Empty nodes will be represented by storing in the empty node's parent a handle value that is recognized as representing an empty leaf node (the flyweight). The flyweight may be actually represented as a physical node on disk, or you may use a special handle value that is simply recognized as the flywight.

Leaf nodes that contain a watcher record will require 5 bytes, storing the following fields: a one-byte field used to distinguish internal from leaf nodes, and a 4-byte field that stores the handle to the watcher record.

Each Watcher record will be stored as a separate message in the memory pool. Its format will be as follows. First, an 8-byte double value for the Watcher's X coordinate, then an 8-byte double value for the Watcher's Y-coordinate, then a series of characters that store the Watcher's name. The Watcher name should just store the letters in the name. You should not store any termination character or length value for the name (you already can deduce the name length from knowing the length of the entire message).

Java Code:

For this project, you may only use standard Java classes, Java code that you have written yourself, and Java code supplied by the CS3114 instructor (see the class website for the distribution).

You may not use other third-party Java code. You may use standard Java list classes for this assignment.

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.