CS3114 (Fall 2013)
PROGRAMMING ASSIGNMENT #2

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

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

Updated: 10/10/2013

Assignment:

This project continues the theme of an earthquake notification system that was begun in Project 1. This time, the focus is organizing the watcher records into a database for fast search. Watcher records get searched for by name when they are deleted, and by location when an earthquake record is processed. The linked list that you used in Project 1 does not help to make either search efficient. Thus, your project will actually implement **two** trees: you will implement a Binary Search Tree to support searches by name, and you will implement a spatial data structure called a bintree to support searches by position.

Input to your program is identical to that of Project 1, except that there is an additional command that can appear in the watcher stream (see "Output Format" below).

Aside from this, your program will simply repeat Project 1, execept that you will replace the linked list for the watcher records with a BST and a bintree.

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 **not** use any built-in Java list classes for this assignment.

Data Structures and Design:

Note: in addition to the requirements below, if you lost points on your design or implementation related to the queue or the heap in Project 1, then you must correct those problems or you will lose those same points again.

Your BST **must** be implemented such that when you delete a node that has two non-empty children, you will replace it with the **greatest-valued** element in the **left** subtree. This requirement will result in everyone having the same BST for a given series of inserts and deletes.

To see how a bintree works, Module 7.11 in the OpenDSA textbook on bintrees.

Your bintree will assume a range in the X axis of [0.0, 360.0), and in the Y axis of [0, 180.0), Since the Earthquake API provides coodinates in standard longitude/latitude, you need to convert to the bintree's coordinate system by adding 180 to the longitude and adding 90 to the latitude. Be careful about handling the coordinates, so that you are consistent with the grader's data. For example, when splitting in the X dimension, a coordinate of exactly 180.0 would go the the right side of the dividing line, not the left.

The primary design challenge for this project is dealing with the fact that your watcher records can be associated with using the name for a key (in the BST), or with using the long/lat coordinate for a key (in the bintree). Thus, you cannot just implement the `comparable` interface because there is not just one thing in a record to compare. Instead, you should either implement a comparator class for your containers, or else store key/value pairs with your containers. Either approach is acceptable. See Module 6.3 in the OpenDSA tutorial for details.

Your bintree and BST must implement their insert, region search, and delete methods using recursion. You must use inheritance for your bintree node hierarchy, with separate classes for the internal and leaf nodes. Empty leaf nodes must be implemented using a flyweight. Empty leaf nodes may be implemented as a separate class or as a special instance of the leaf class. You may not store bounding box coordinates in bintree nodes, and bintree nodes may not store a pointer to that node's parent. You bintree search operation should visit no more nodes than necessary.

Program Invocation:

The program will be invoked from the command line as:

`java EqSpatial <watcher-file> <earthquake-stream>`

(Note that the `--all` option is not used in this project.)

The `<earthquake-stream>` parameter should either be the word `live` or a filename. If it is `live`, then your program should use the real-time data stream. Otherwise, it should use the given local file to simulate the real-time data stream.

Output Format

As with Project 1, during a given time step you will first process the watcher stream commands associated with that time step, and then process the earthquake report for that time step. Be sure that you process the watcher command in the order that they are received.

Output format is identical to Project 1, with the following modifications.

1. There is no `--all` parameter on command line invocation.

2. Attempts to add a watcher can be successful or unsuccessful. If the name duplicates a name of a current watcher in the system, or the coordinate duplicates the coordinate of a current watcher in the system, then the request should be rejected (and the watcher is not added to the database). You will attempt first to insert to the BST, and then to the bintree.

   When you process a new watcher add request, you will no longer print the message that the name is added to the watcher list. Instead, you should print the these two lines on a successful insert:

   ```
   <Watcher name> <coordinate> is added to the BST
   ```

   ```
   <Watcher name> <coordinate> is added to the bintree
   ```

   If the name is a duplicate, neither of those two lines will be printed. Instead you will print

   ```
   <Watcher name> duplicates a watcher already in the BST
   ```

   If the coordinate is a duplicate, then you will first print the message that the watcher was added to the BST. Then you will print:

   ```
   <coordinate> duplicates a watcher already in the bintree
   ```

   Finally, you will remove that record from the BST that you just added, and print:

   ```
   <Watcher name> is removed from the BST
   ```

3. When you process a watcher remove request, you will no longer print the message that the name is removed from the watcher list. Instead, you should print the these two lines:

   ```
   <Watcher name> <coordinate> is removed from the BST
   ```

```
<Watcher name> <coordinate> is removed from the bintree
```

Note that you first check the BST (since deletion is by name), from which you can then get the coordinate for deletion from the bintree. If the name is not in the BST, then you should print:

```
<Watcher name> does not appear in the BST
```

4. The output for the query command will be the same as in Project 1.

5. The watcher stream can contain the "debug" command. When you see a "debug" command, you will output the current state of your BST and your bintree using the following formats.

   For the bintree, you will produce a preorder traversal of the tree where internal nodes are marked with I, empty leaf nodes are marked with E, and leaf nodes that contain a data point output the name and coordinates of the point's position. Every node will be printed on a separate line.

   For the BST, you will produce an inorder traversal of the tree where each node is on a separate line, the name and coordinates of the watcher at that node is printed, and each line is preceeded by 2 periods for each level that the corresponding node is in the tree. If the BST is empty, then you will not print anything for its traversal.

6. When an earthquake is added to the heap and queue, you must do a traversal of the bintree to determine which watchers are notified. Your output for the notification messages should be changed from Project 1 to look as follows:

   ```
   <Earthquake> is close to the following watchers:

   <Watcher1>

   <Watcher2>
   ```

   That is, print out a message to describe the earthquake and the print (each on a separate line) the list of watchers that are notified. If no watcher is notified, then no watchers will be printed. After the notification messages (if any) are printed, you will output the following for every earthquake that is inserted:

   ```
   Earthquake inserted at <long, lat>.
   Watcher search caused <##> bintree nodes to be visited.
   ```

   A node is visited during the search if (and only if) the bounding box for the earthquake overlaps its bounding box.

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.