CS3114 (Fall 2011)
PROGRAMMING ASSIGNMENT #2

Due Tuesday, October 11 @ 11:00 PM for 100 points
Due Monday, October 10 @ 11:00 PM for 10 point bonus
Initial Schedule due Thursday, September 22 @ 11:00 PM
Second Schedule due Tuesday, October 4 @ 11:00 PM

Revised 10/4/2011

Background:

This project continues the theme of a Geographic Information System that was begun in Project 1. This time, the focus is organizing the city records into a database for fast search. For each city you will store the name and its location (X and Y coordinates). Searches can be by either name or location. 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 variation on the PR Quadtree to support searches by position.

Consider what happens if you store the city records using a linked list. The cost of key operations (insert, remove, search) using this representation would be $\Theta(n)$ where n is the number of cities. For a few records this is acceptable, but for a large number of records this becomes too slow. Some other representation is needed that makes these operations much faster. In this project, you will implement a variation on one such representation, known as a PR Quadtree.

A binary search tree gives O(log n) performance for insert, remove, and search operations (if you ignore the possibility that it is unbalanced). This would allow you to insert and remove cities, and locate them by name. However, the BST does not help when doing a search by coordinate. In particular, the primary search operation that we are concerned with in this project is a form of range query called "regionsearch." In a regionsearch, we want to find all cities that are within a certain distance of a query point.

You might consider combining the (x, y) coordinates into a single key and store cities using this key in a second BST. That would allow search by coordinate, but does nothing to help regionsearch. The problem is that the BST only works well for one-dimensional keys, while a coordinate is a two-dimensional key.

To solve these problems, you will implement a variant of the PR Quadtree. The PR Quadtree (see Section 13.3.2 of the textbook) is one of many **hierarchical data structures** commonly used to store data such as city coordinates. It allows for efficient insertion, removal, and regionsearch queries. You should pay particular attention to the discussion regarding parameter passing in recursive functions, and to the discussion regarding flyweights, both in Section 13.3.2. Both of these discussions are relevent to your implementation.

Your version of the PR Quadtree differs from the one described in Section 13.3.2 in that your leaf nodes will store up to three points.

Invocation and I/O Files:

Your program must be named **PRprog**, and should be invoked as:

```
PRprog <filename>
```

where `<filename>` is a commandline argument for the name of the command file. Your program should write to standard output (`System.out`). The program should terminate when it reads the end of file mark from the input file.

The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. Commands are free format in that an arbitrary number of additional spaces may be interspersed between parameters. The input file may also contain blank lines, which your program should ignore. You do not need to check for syntax errors in the command lines (although you **do** need to check for logical errors such as duplicate insertions or removals of non-existent cities).

Each input command should result in meaningful feedback in terms of an output message. Each input command should be echo'ed to the output. In addition, some indication of success or error should be reported. Some of the command specifications below indicate particular additional information that is to be output.

Commands and their syntax are as follows. Note that a *name* is defined to be a string containing only upper and lower case letters and the underscore character.

insert x y *name*

A city at coordinate (x, y) with name *name* is entered into the database. That means you will store the city record once in the BST, and once in the PR Quadtree. Spatially, the database should be viewed as a square whose origin is in the upper left (NorthWest) corner at position (0, 0). The world is 2^{14} by 2^{14} units wide, and so x and y are integers in the range 0 to $2^{14}-1$. The x-coordinate increases to the right, the y-coordinate increases going down. Note that it is an error to insert two cities with identical coordinates, but **not** an error to insert two cities with identical names.

remove x y

The city with coordinate (x, y) is removed from the database (if it exists). That means it must be removed from both the PR Quadtree and the BST. Be sure to print the name and coordinates of the city that is removed.

remove *name*

A city with name *name* is removed from the database (if any exists). That means it must be removed from both the PR Quadtree and the BST. Be sure to print the name and coordinates of the city that is removed.

find *name*

Print the name and coordinates from all city records with name *name*. You would search the BST for this information.

search x y *radius*

All cities within *radius* distance from location (x, y) are listed. A city that is exactly *radius* distance from the query point should be listed. x and y are (signed) integers with absolute value less than 2^{14}. *radius* is a non-negative integer with value less than 2^{14}. You should also output a count of the number of PR Quadtree nodes looked at during the search process. If you need to determine the type of the node (internal, leaf, empty), then you should count it as "looked at".

debug

Print a listing of the PR Quadtree nodes in preorder. PR Quadtree children will appear in the order NW, NE, SW, SE. The node listing should appear as a single string (without internal newline characters or spaces) as follows:

- For an internal node, print "I".
- For an empty leaf node (the flyweight), print "E".
- For a leaf node containing one or more data points, for each data point print X,Y,NAME where X is the x-coordinate, Y is the y-coordinate, and NAME is the city name. After all of the city records for that node have been printed, print a "bar" or "pipe" symbol (the `|` character).

The tree corresponding to Figure 13.16 of the textbook would be printed as:
`I40,45,A|I70,10,C|E69,50,D|E15,70,B|IIEE55,80,E|80,90,F|EEE|`. (Note that the figure doesn't show what your tree would really look like for this example, because your leaf nodes hold up to three data points each.)

makenull

Initialize the database to be empty.

Example:

Note: in this example, statements enclosed in `{}` are comments to help you understand the example; comments do NOT appear in the data file!

```
insert 900 500  Blacksburg
    insert   500   140   Roanoke
 insert 910  510  New_York

 debug                      { print coords, name for 3 cities }
  remove  500 140           { its there to remove }
search 901  501  5          { print info for one city }
makenull                    { reinitialize }
```

Implementation:

Note that in Project 4 you will be re-implementing this project to store the PR Quadtree and the BST on disk. To avoid having to completely rewrite your tree class code, you should be sure to access tree nodes **ONLY** through `set` and `get` methods in the node classes. Make the child references to be private data members of the node class to insure that this happens.

All operations that traverse or descend the BST or the PR Quadtree structures MUST be implemented recursively.

You must use inheritance or an interface to implement PR Quadtree nodes. You should declare either abstract base class or an interface, and define separate subclasses for the internal nodes and the leaf nodes. A PR Quadtree internal node should store references to its children. Leaf nodes should store a (reference to a) city record object. Empty leaf nodes should be implemented by a flyweight. The empty leaf node can either be its own separate subclass, or an instance of the leaf node subclass.

The PR Quadtree and the BST should be implemented in some way that supports their generic use with records of various types. For the BST this should be easy, though you do have to deal with issues related to getting the key from a record and doing key comparisons, as discussed in class. The PR Quadtree is more specialized since there must be a mechanism to get two coordinate

values from the record. But it should be able to handle more than just hard-coded access to "city" records.

Programming Standards:

You must conform to good programming/documentation standards. Some specifics:

- You must include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- 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 code taken from the textbook. 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.

Testing:

A sample data file will be posted to the website to help you test your program. This is not the data file that will be used in grading your program. The test data provided to you will attempt to exercise the various syntactic elements of the command specifications. It makes no effort to be comprehensive in terms of testing the data structures required by the program. Thus, while the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

Deliverables:

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 will submit your project through the automated Web-CAT server. Links to the Web-CAT client and instructions for those students who are not developing in Eclipse are posted at the class website. If you make multiple submissions, only your last submission will be evaluated.

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.

Scheduling:

In addition to the project submission, you are also required to submit an initial project schedule, an intermediate schedule, and a final schedule with your project submission. The schedule sheet template for this project is posted at the course website. You won't receive direct credit for submitting the schedule as required, but each instance of failing to submit scheduling information as required will lose 10 points from the project grade.

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.