PR Quadtree Project

Background:

For this project you will build a simple Geographic Information System for storing point data. The focus is organizing 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 could combine 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 PQ 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 inovoked 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. You should also output a count of the number of PR Quadtree nodes looked at during the search process. x and y are (signed) integers with absolute value less than 2^{14}. *radius* is a non-negative integer.

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 13.13 of the textbook would be printed as: `IEIE110,25,E|EI98,35,C|EE117,52,D|30,90A|95,85B|`. (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 under 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 to implement PR Quadtree nodes. You should have an abstract base class with separate subclasses for the internal nodes,the leaf nodes, and empty 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 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.