PR Quadtree

This assignment involves implementing a region quadtree (specifically the PR quadtree as described in section 3.2 of Samet's paper) as a Java generic. Because this assignment will be auto-graded using a test harness we will provide, your implementation must conform to the public interface below, and include at least all of the public and private members that are shown:

```
// The test harness will belong to the following package; the quadtree
// implementation must belong to it as well.  In addition, the quadtree
// implementation must specify package access for the node types and tree
// members so that the test harness may have access to it.
//
package Minor.P3.DS;

public class prQuadtree< T extends Compare2D<? super T> > {

   abstract class prQuadNode { . . . }
   class prQuadLeaf extends prQuadNode {
      Vector<T> Elements;
   }
   class prQuadInternal extends prQuadNode {
      prQuadNode NW, NE, SE, SW;
   }

   prQuadNode root;
   long xMin, xMax, yMin, yMax;
   . . .

   // Initialize quadtree to empty state, representing the specified region.
   public prQuadtree(long xMin, long xMax, long yMin, long yMax) {
   . . .
   }

   // Pre:   elem != null
   // Post:  If elem lies within the tree's region, and elem is not already
   //        present in the tree, elem has been inserted into the tree.
   // Return true iff elem is inserted into the tree.
   public boolean insert(T elem) {
      . . .
   }


   // Pre:  elem != null
   // Post: If elem lies in the tree's region, and a matching element occurs
   //       in the tree, then that element has been removed.
   // Returns true iff a matching element has been removed from the tree.
   public boolean delete(T Elem) {
      . . .
   }

   // Pre:  elem != null
   // Returns reference to an element x within the tree such that
   // elem.equals(x)is true, provided such a matching element occurs within
   // the tree; returns null otherwise.
   public T find(T Elem) {
      . . .
   }
```

```
   // Pre:  xLo, xHi, yLo and yHi define a rectangular region
   // Returns a collection of (references to) all elements x such that x is
   //in the tree and x lies at coordinates within the defined rectangular
   // region, including the boundary of the region.
   public Vector<T> find(long xLo, long xHi, long yLo, long yHi) {
      . . .
   }
}
```

You may safely add features to the given interface, but if you omit or modify members of the given interface you will almost certainly face compilation errors when you submit your implementation for testing.

You must implement tree traversals recursively, not iteratively. You will certainly need to add a number of private recursive helper functions that are not shown above. Since those will never be called directly by the test code, the interfaces are up to you.

For this assignment, the bucket size is 1; that is, each leaf node will store exactly one data object.

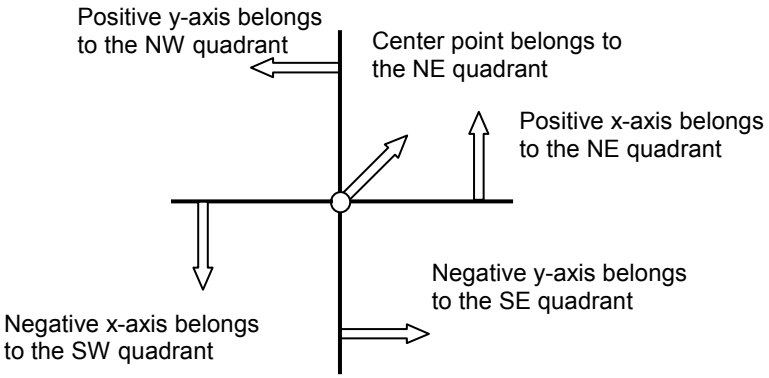
Note that the test harness for this assignment is shallower than the one for the BST project. That is, aside from checking the root pointer, it does not attempt to examine the internal structure of your quadtree at all. Instead, tree displays, written by the test driver, are compared to determine whether your tree has the correct structure. Therefore, the restrictions on the node types are extremely flexible.

On the other hand, your implementation must be designed to work with data objects that implement the following interface:

```
package Minor.P3.DS;

// The interface Compare2D is intended to supply facilities that are useful in
// supporting the the use of a generic spatial structure with a user-defined
// data type.
//
public interface Compare2D<T> {

   // Returns the x-coordinate field of the user data object.
   public long getX();

   // Returns the y-coordinate field of the user data object.
   public long getY();

   // Returns indicator of the direction to the user data object from the
   // location (X, Y) specified by the parameters.
   // The indicators are defined in the enumeration Direction, and are used
   // as follows:
   //
   //    NE:  vector from (X, Y) to user data object has a direction in the
   //         range [0, 90) degrees (relative to the positive horizontal axis
   //    NW:  same as above, but direction is in the range [90, 180)
   //    SW:  same as above, but direction is in the range [180, 270)
   //    SE:  same as above, but direction is in the range [270, 360)
   //    NOQUADRANT:  location of user data object is equal to (X, Y)
   //
   public Direction directionFrom(long X, long Y);
```

```
    // Returns indicator of which quadrant of the rectangle specified by the
    // parameters that user data object lies in.
    // The indicators are defined in the enumeration Direction, and are used
    // as follows, relative to the center of the rectangle:
    //
    //    NE:  user data object lies in NE quadrant, including non-negative
    //         x-axis, but not the positive y-axis
    //    NW:  user data object lies in the NW quadrant, including the positive
    //         y-axis, but not the negative x-axis
    //    SW:  user data object lies in the SW quadrant, including the negative
    //         x-axis, but not the negative y-axis
    //    SE:  user data object lies in the SE quadrant, including the negative
    //         y-axis, but not the positive x-axis
    //    NOQUADRANT:  user data object lies outside the specified rectangle
    //
    public Direction inQuadrant(double xLo, double xHi, double yLo,
                                double yHi);

    // Returns true iff the user data object lies within or on the boundaries
    // of the rectangle specified by the parameters.
    public boolean   inBox(double xLo, double xHi, double yLo, double yHi);

    // Overrides the user data object's inherited equals() method with an
    // appropriate definition; it is necessary to place this in the interface
    // that is used as a bound on the type parameter for the generic spatial
    // structure, otherwise the compiler will bind to Object.equals(), which
    // will almost certainly be inappropriate.
    public boolean equals(Object o);
}
```

The interface is intended to allow your implementation to either take coordinates from the data object (via `getX()` and `getY()`) and use those values to make directional decisions, or to use the `directionFrom()` method supplied by the data object for the same purpose. The only restriction is that if you take the first approach you must make sure that your logic follows the following pattern when partitioning the world space:



The actual test data will be created so that (ideally) no data points will ever lie on a region boundary. However, you should conform to the guideline above just to be safe.

One more note, we specifically require that the `equals()` and `directionFrom()` methods are consistent, in the sense that `equals()` returns true if and only if `directionFrom()` returns `NOQUADRANT`. We are specific about this because the Java language specification does, in fact, allow an inconsistency between `equals()` and `compareTo()`.

The `Compare2D` interface depends upon the following enumerated type:

```
package Minor.P3.DS;

public enum Direction {NW, SW, SE, NE, NOQUADRANT};
```

During our testing of your implementation, we will use the following data object type; a more complete version is posted on the course website.

```
package Minor.P3.Client;
import Minor.P3.DS.Compare2D;
import Minor.P3.DS.Direction;

public class Point implements Compare2D<Point> {

   private long xcoord;
   private long ycoord;

   public Point() {
         xcoord = 0;
         ycoord = 0;
   }

   public Point(long x, long y) {
         xcoord = x;
         ycoord = y;
   }

   public long getX() {
         return xcoord;
   }

   public long getY() {
         return ycoord;
   }

   public Direction directionFrom(long X, long Y) { . . . }

   public Direction inQuadrant(double xLo, double xHi,
                               double yLo, double yHi) { . . . }

   public boolean   inBox(double xLo, double xHi,
                          double yLo, double yHi) { . . . }

   public String toString() { . . . }

   public boolean equals(Object o) { . . . }
}
```

Note that if you use calls to `directionFrom()`or `inQuadrant()` or `inBox()` or `equals()` in your quadtree you must implement those methods to be consistent with the directional guidelines given on the previous page.

BTW, I put the class `Point` in the package `Minor.P3.Client` because, on the one hand, it needs to be in a package so my test class can import it, and, on the other hand, it doesn't seem to make sense to put a user *data type* into a package intended for *data structures*.

Displaying the tree

It is necessary that your quadtree implementation display the tree in a specific form, so that the testing protocols can evaluate the structure correctly. The posted shell for the PR quadtree class includes a complete implementation of display methods that will produce satisfactory results. The given code is essentially the same as that in the course notes; the only significant change is that hyphens are used to make the indentation of each node clearer.

Design and implementation requirements

There are some explicit requirements, in addition to those on the *Programming Standards* page of the course website:

- You must implement the PR quadtree to conform to the specification.
- The insertion logic must not allow duplicate (according to `equals()`) records to be inserted.
- The insertion logic must not allow the insertion of records that lie outside the world boundaries.
- Deletion must be handled exactly as described in the course notes. In particular, pay attention to the possibility that a branch may contract more than a single level when a leaf node is deleted.
- Under no circumstances should any of the PR quadtree member functions write output, except for a `display()` method and its helper, if you choose to implement that.
- When carrying out a region search, you must determine whether the search region overlaps with the region corresponding to a subtree node before descending into that subtree. The Java libraries include a Rectangle class which could be (too) useful. You may make use of the Rectangle class, but you will be penalized on the GIS project later in the term if your submitted solution to that assignment makes use of any of the following Rectangle methods: contains(), intersection(), and intersects(). Note though, that it is acceptable to make use of those methods during development of a solution to this assignment.
- When carrying out a region search, the order in which you add elements to the vector does not matter.

Testing

We will be testing your implementation with our own test driver. We may (or may not) release information about that driver before the assignment is due. In any case, it is your responsibility to design and carry out a sensible test of your implementation before submitting it. For that purpose, you may share test code (**but absolutely no tree code!!**) via the class Forum.

Be sure you test all of the interface elements thoroughly, both in isolation and in interleaved fashion.

Evaluation

You should document your implementation in accordance with the *Programming Standards* page on the course website. It is possible that your implementation will be evaluated for documentation and design, as well as for correctness of results. If so, your submission that achieved the highest score will be evaluated by one of the TAs, who will assess a deduction (ideally zero) against your score from the Curator.

Note that the evaluation of your project may depend substantially on the quality of your code and documentation.

What to turn in and how

This assignment will be auto-graded using a test harness on the Curator system. The testing will be done under Windows (which should not matter at all) using Java version 1.6.21 or later.

Submit a single `.java` file (not zipped) containing your PR quadtree generic to the Curator System. Submit nothing else. Your solution should not write anything to standard output.

Your submitted source file will be placed in the appropriate subdirectory with the packaged test code, and will then be compiled with the test harness using the following command, executed in the root of the source directory tree:

```
javac testDriver.java
```

Instructions, and the appropriate link, for submitting to the Curator are given in the *Student Guide* at the Curator website:

http://www.cs.vt.edu/curator/.

You will be allowed to submit your solution multiple times; the highest score will be counted.

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the Programming Standards page in one of your submitted files.