PR quadtree

This assignment involves implementing a point-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:

```java
// 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.J3.DS;

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

    abstract class prQuadNode { . . . }
    class prQuadLeaf extends prQuadNode {
       ArrayList<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.
    // Pre:   xMin < xMax and yMin < yMax
    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
    // 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:   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:  xLo < xHi and yLo < yHi
    // 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 ArrayList<T> find(long xLo, long xHi, long yLo, long yHi) {. . .}
}
```

You may safely <u>add</u> features to the given interface, but if you omit or modify members of the given public interface you will be likely to 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. That said, you must still use an `ArrayList` object to store the single data element in the leaf node; that will help you later when you convert this implementation to a fully bucketed approach.

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 <u>must</u> be designed to work with data objects that implement the following interface:

```
package Minor.J3.DS;

// The interface Compare2D is intended to supply facilities that are useful
// in supporting the use of a generic 2D spatial structure with a user-
// defined data type.  As a consequence, the interface contains a number of
// redundant features; it is unlikely that any particular PR quadtree
// implementation would make use of every feature shown here.
//
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);
}
```

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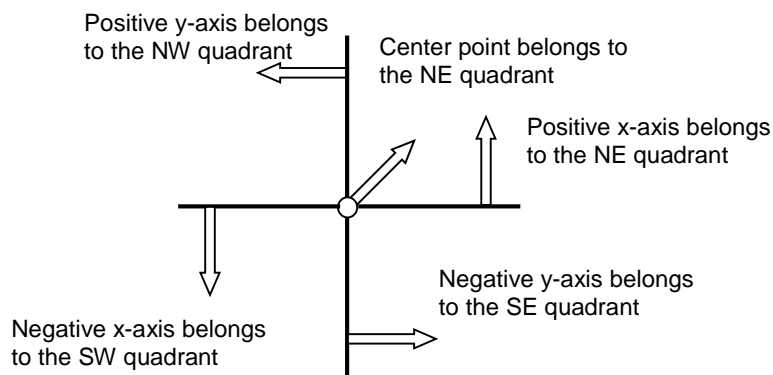
The actual test data will be created so that (ideally) no data points will ever lie on a region boundary. However, you should conform to the guideline above just to be safe.

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.J3.DS;

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

The use of an enumerated type allows us to use descriptive labels at critical points in the implementation. If you have not made use of this Java feature before, this is a good place to learn about it.

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

```java
package Minor.J3.Client;
import Minor.J3.DS.Compare2D;
import Minor.J3.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) { . . . }

    // Additional methods as needed...
    . . .
}
```

Note that if you use calls to `directionFrom()` or `inQuadrant()` or `inBox()` 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.J3.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*.

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 <u>not</u> allow duplicate (according to `equals()`) records to be inserted.
- The insertion logic must <u>not</u> allow the insertion of records that lie outside the world boundaries.
- 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 (the test harness will never call such a method).
- 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 <u>may not</u> make use of the `Rectangle` class from the Java library, but 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, since the testing code will sort the elements before considering them.
- Naturally, your implementation of region search should be efficient; therefore, it must not look at subtrees that could not possibly contain elements that fall in to the specified region. This will be checked by the course TAs.

Testing and Evaluation

We will be testing your implementation with a stand-alone test harness. A version of that harness is posted as a zip file:

`gradeJ3.sh`	shell script to drive testing process
`prQuadGenerator.jar`	test case generator
`LogComparator.jar`	tool for comparing and scoring test output
`testDriver.java`	main Java file for test harness
`Minor/J3/Client/Point.java`	data type for insertion into PR quadtree during testing*
`Minor/J3/DS/Compare2D.java`	type bound for PR quadtree generic parameter
`Minor/J3/DS/Direction.java`	enumerated type used in Compare2D
`Minor/J3/DS/prQuadTree.java`	PR Quadtree implementation*
`Minor/J3/DS/Lewis.class`	test methods

*   shell files that you must complete in order to run the testing code

After unpacking the test harness, you must implement the `Point` type and the PR quadtree in order to perform testing. Specifically, you must implement the parts of the `Point` type related to the elements of the `Compare2D` interface that your PR quadtree makes use of. You can restrict which methods in the PR quadtree are tested by commenting out sections of the main test driver file, `testDriver.java`.

You can run the test harness with the command: `./gradeJ3.sh [-rand | -repeat]`

Run the script first with the `-rand` switch; use that switch if you want to generate new randomized test cases created. Use the `-repeat` switch to use test cases created by an earlier testing run. Examine the various log files for the details of the scoring.

There is no guarantee that any particular run of the test harness will create every possible logical case. It is your responsibility to perform testing 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 <u>you</u> test all of the interface elements thoroughly, both in isolation and in interleaved fashion. Do not include any testing code in your submission, including calls to JUnit elements.

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 final submission will be evaluated by one of the TAs, who will assess a deduction (ideally zero) against your score from the testing harness.

What to turn in and how

This assignment will be auto-graded using a stand-alone test harness, running on a CentOS 7 system, using Java version 1.8.0u25 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.

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 following pledge statement at the beginning of the file that contains `main()`:

```
//     On my honor:
//
//     - I have not discussed the Java language code in my program with
//       anyone other than my instructor or the teaching assistants
//       assigned to this course.
//
//     - I have not used Java language code obtained from another student,
//       or any other unauthorized source, either modified or unmodified.
//
//     - If any Java language code or documentation used in my program
//       was obtained from another source, such as a text book or course
//       notes, that has been clearly noted with a proper citation in
//       the comments of my program.
//
//     - I have not designed this program in such a way as to defeat or
//       interfere with the normal operation of the grading code.
//
//     <Student's Name>
```

We reserve the option of assigning a score of zero to any submission that does not contain this statement.

Change Log Relative to Version 4.00

If changes are made to the specification, details will be noted below, the version number will be updated, and an announcement will be made on the course Forum board.

4.00   Base document. Posted on June 7.