Programming Assignment 5 Checklist: Kd-Trees

Frequently Asked Questions

What should I do if a point has the same x-coordinate as the point in a node when inserting / searching in a 2d-tree? Go the right subtree as specified.

Can I assume that all x- or y-coordinates of points inserted into the `KdTree` will be between 0 and 1? Yes. You may also assume that the `insert()`, `contains()`, and `nearest()` methods in `KdTree` are passed points with x- and y-coordinates between 0 and 1. You may also assume that the `range()` method in `KdTree` is passed a rectangle that lies in the unit box.

What should I do if a point is inserted twice in the data structure? The data structure represents a *set* of points, so you should keep only one copy.

How should I scale the coordinate system when drawing? Don't, please keep the default range of 0 to 1.

How should I set the size and color of the points and rectangles when drawing? Use `StdDraw.setPenColor(StdDraw.BLACK)` and `StdDraw.setPenRadius(0.01)` before before drawing the points; use `StdDraw.setPenColor(StdDraw.RED)` or `StdDraw.setPenColor(StdDraw.BLUE)` and `StdDraw.setPenRadius()` before drawing the splitting lines.

What should `range()` return if there are no points in the range? It should return an `Iterable<Point2D>` object with zero points.

How much memory does a `Point2D` object use? For simplicity, assume that each `Point2D` object uses 32 bytes—in reality, it uses a bit more because of the `Comparator` instance variables.

How much memory does a `RectHV` object use? You can look at the code and analyze it.

I run out of memory when running some of the large sample files. What should I do? Be sure to ask Java for additional memory, e.g., `java -Xmx1600m RangeSearchVisualizer input1M.txt`.

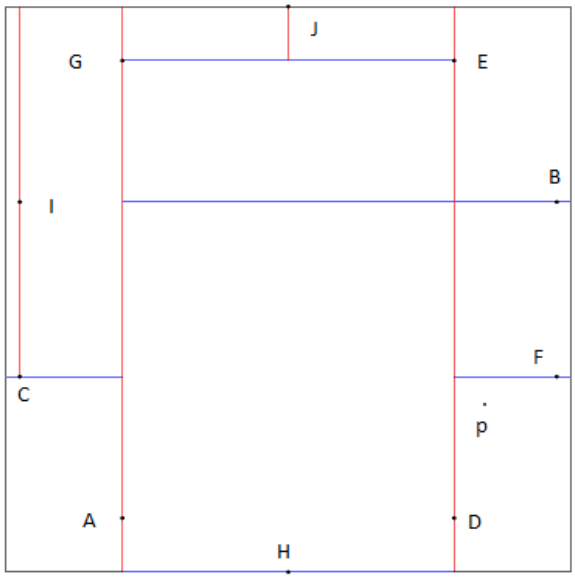
Testing

Testing. A good way to test `KdTree` is to perform the same sequence of operations on both the `PointSET` and `KdTree` data types and identify any discrepancies. The sample clients RangeSearchVisualizer.java and NearestNeighborVisualizer.java take this approach.

Sample input files. The directory kdtree contains some sample input files in the specified format. For convenience, kdtree-testing.zip contains all of these files bundled together.

- `circleN.txt` contains `n` points on the circumference of the circle centered on (0.5, 0.5) of radius 0.5.

The result of calling `draw()` on the points in `circle10.txt` should look like the following:



If `nearest()` is called with `p = (0.81, 0.30)` the number of nodes visited in order to find that F is nearest is 5.

Starting with `circle10k.txt` if nearest is called with `p = (0.81, 0.30)` the number of nodes visited in order to find that K is nearest is 6.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. **Node data type.** There are several reasonable ways to represent a node in a 2d-tree. One approach is to include the point, a link to the left/bottom subtree, a link to the right/top subtree, and an axis-aligned rectangle corresponding to the node.

   ```
   private static class Node {
      private Point2D p;      // the point
      private RectHV rect;    // the axis-aligned rectangle corresponding to this node
      private Node lb;        // the left/bottom subtree
      private Node rt;        // the right/top subtree
   }
   ```

   Unlike the `Node` class for `BST`, this `Node` class is static because it does not refer to a generic `Key` or `Value` type that depends on the object associated with the outer class. This saves the 8-byte inner class object overhead. (Making the `Node` class static in `BST` is also possible if you make the `Node` type itself generic as well). Also, since we don't need to implement the rank and select operations, there is no need to store the subtree size.

2. **Writing KdTree.** Start by writing `isEmpty()` and `size()`. These should be very easy. From there, write a simplified version of insert() which does everything except set up the `RectHV` for each node. Write the `contains()` method, and use this to test that `insert()` was implemented properly. Note that `insert()` and `contains()` are best implemented by using private helper methods analogous to those found on page 399 of the book or by looking at BST.java. We recommend using orientation as an argument to these helper methods.

   Now add the code to `insert()` which sets up the RectHV for each Node. Next, write `draw()`, and use this to test these rectangles. Finish up KdTree with the nearest and range methods. Finally, test your implementation using our interactive client programs as well as any other tests you'd like to conduct.

Optimizations

These are many ways to improve performance of your 2d-tree. Here are some ideas.

- **Squared distances.** Whenever you need to compare two Euclidean distances, it is often more efficient to compare the squares of the two distances to avoid the expensive operation of taking square roots. Everyone should implement this optimization because it is both easy to do and likely a bottleneck.

- **Range search.** Instead of checking whether the query rectangle intersects the rectangle corresponding to a node, it suffices to check only whether the query rectangle intersects the splitting line segment: if it does, then recursively search both subtrees; otherwise, recursively search the one subtree where points intersecting the query rectangle could be.

- **Save memory.** You are not required to explicitly store a `RectHV` in each 2d-tree node (though it is probably wise in your first version).