Homework 6: Due: Tuesday April 17 by 11:59PM via email to grader (dkc237@nyu.edu)

This homework assignment generalizes binary search trees to allow searching on multiple keys. Such a tree is called a k-d tree for k-dimensional tree. In this assignment we will start by inserting 2 dimensional points into a tree, so k = 2.

For example, let points \( p_k = (x_k, y_k) \), for \( k = 1, \ldots, n \). A regular Binary Search Tree can only insert the points searching using either the \( x \) or \( y \) coordinate, but not both. A k-d tree first searches on the \( x \) coordinate to decide whether to go to the left or right of a node, then at the next level searches on the \( y \) coordinate. Then it’s back to \( x \) again. Generalizing to \( k \) dimensions you cycle through all \( k \) keys at \( k \) successive levels before repeating from the beginning.

Here is an example of some points and the k-d tree associated with it. For this example we will assume all points are in the unit square from \([0,0]\) to \([1,1]\), and inserted in the order given.

Points

\[
\begin{align*}
A &= (.5,.4) \\
B &= (.2,.6) \\
C &= (.6,.8) \\
D &= (.6,.7) \\
E &= (.65,.7)
\end{align*}
\]

The usefulness of a k-d tree is that it enables fast range queries - i.e. it can answer the following kind of question quickly: Tell me all points in the rectangular region from \((\text{low}_x, \text{low}_y)\) to \((\text{high}_x, \text{high}_y)\). This kind of query is useful for example in video games, when the game needs to see if thousands of objects might potentially collide with something at each step of a simulation, and thus need special processing (e.g. the fireworks hit the house and an explosion should be drawn). Another example that uses k-d tree is ray tracing, a method in computer graphics for drawing an image by following a ray of light and drawing its effects (e.g. shadows, reflections) when it hits an object.

The `printRange` question can be answered by traversing the tree, and not taking any paths that clearly are outside the rectangular region of interest. (If this is not enough of a description to figure it out, draw some examples from the figure above. We can also discuss further in class, or you can post questions to the class email list). The query should return an ArrayList of nodes with data in the requested region. The calling method (your main method) should then print how many points were retrieved, and print the result. In the game example above, the objects would be represented by the points of their bounding boxes, and each object would have a velocity. When the boxes are retrieved by the printRange query, those boxes can be checked more precisely using their real geometry to see if an intersection will occur in the next step.

For homework, write the code needed to create, traverse, insert, and printRange for a k-d tree. Your code should work for \( k \) dimensions, even if we only test it for 2 dimensions. Your
KDT tree class should take nodes of type T that implement the `KDTPoint` interface, defined here:

```java
public interface KDTPoint{
    public double getDim(int d); // return the double in the dth dimension
}
```

In other words, the KDT tree must have nodes that are \( k \)-tuples of doubles. For example the two dimensional point class that you should write should have a `getDim` method that returns the \( x \) coordinate when \( d = 0 \), and the \( y \) coordinate when \( d = 1 \).

You should debug your code by inserting a few specific points into the tree. When you submit it to the grader you should insert 1000 randomly generated points into the tree, and have a `printRange` command that looks for anything in the region \((.42, .5)\) to \((.48, .54)\) (these should be variables set to these values at the top of your code, so they can be easily changed for testing, not hardwired in the middle of your main method.)

**Honors section:** As with a binary search tree, if the nodes are not inserted into the KDT tree in a good order, an imbalanced tree may result. The honors section should write additional methods that inserts points in a way that results in a balanced KDT tree. To do this, all the points have to be known in advance, which is usually not a problem. The algorithm is the following:

1. Find an approximation to the median of all the points. This is usually done by taking a constant number of points (e.g. the first, middle point and last point) and picking the middle one as a guess of the median.
2. Partition the array into points less than and greater than the approximate median from step 1.
3. Recursively repeat on the left and right subarrays. Continue inserting until there are no more points left to insert. (Mergesort may be a useful resource here).

Use the same set of points to insert with the regular KDT tree algorithm and the balanced KDT algorithm. Compare the two trees that result by looking at the maximum depth, and the results of some `printRange` queries.

To see more of a difference, you might insert points that are not uniformly distributed but are not centered. For example, try using the `Random.nextGaussian()` random number generator instead of `Math.random()`. Another thing you might do is use the seed, so you can debug with the same set of numbers. (A seed allows the random number generator to start in the same way, and thus generate the same numbers, each time it is called with that seed).

Note that there are many subparts to this algorithm. Try to use an efficient method to do the subparts. Include a detailed description of what you did in your submission.