Andrew Walters
CSC 2053
Kd-Tree Part 2

Project Overview:

The described API for KdTreeST.java was successfully implemented and tested. Several binary search tree functions that were applicable to the kd tree, but not required, were not included, most notably delete. The existing API is functional enough to demonstrate the success of the nearest and range methods.

I will provide a brief description of the implementation of the range and nearest methods to make them more easily understandable. It is important to note that the Node in KdTreeST is the same as the Node in BST. The node class only contains a Point2D (key), Value, Size, and Left and Right pointers. When rectangles are needed, they are created and destroyed in the class methods.

Explanation of Implementation:

Both range and search contain a common piece of code that creates two RectHV's. This code uses the following information:

- Parent RectHV (myRect)
- Dimension being compared at Parent (True for X and False for Y)
- Coordinates of the children

The code then creates leftRect which contains all points in the left subtree, and rightRect which contains all points in the right subtree.

Range:

The private range helper method is invoked from the public range method by the following call:

```
return range(rect, new RectHV(0,0,1,1), root, cmpX);
```

The parameters respectively are: the RectHV that you are interested in, the RectHV of the parent, the parent Node, and the Dimension being compared at this Node. The method returns all of the points contained in the subtree with a root at the parent node. The method follows the following algorithm:

- If the point is null, return an empty queue
- If the point is contained in rect, append this point to the queue
- Divide myRect into two child rectangles: leftRect and rightRect

- If leftRect intersects rect (the interested area), then recursively call range with the following parameters: `(rect, leftRect, x.left, !dim)`
- The calling method appends the queue that is returned by the recursive call onto its own queue
- The two above steps are repeated for the right subtree
- The final queue is returned

Nearest:

The private nearest helper method is invoked from the public nearest method by the following call:

```
return nearest(root, p, rect, cmpX);
```

The parameters respectively are: the subtree to be searched, the point whose nearest neighbor we are searching for, the rectangle which contains all of the points in the subtree, and the dimension being compared at this node. The method returns the nearest point in the subtree specified by the first parameter. The method follows the following algorithm:

- Determine if the point we are searching for is contained in the right or left subtree
- Divide myRect into two child rectangles: leftRect and rightRect
- If a subtree exists, search the subtree that contains the point we are looking for by recursively calling nearest
- The recursive call will return the closest point to the point we are interested in from the given subtree, so compare the distance from that point to the interested point to the distance from the current node to the interested point
- Set the closer point to a temporary point variable
- For the subtree that does not contain the point we are searching for, search only if it exists and the rectangle it specifies is closer to the point we are searching for than the current closest known point. Search by following the above two steps
- Return the closest known point

Memory and Timing Analysis:

My implementation saves memory at the expense of computing time. The nodes are small because the corresponding rectangles and dimensions (vertical or horizontal) aren't saved, but rather computed in the class methods.

Memory Analysis:
Given in terms of N points contained in the KdTreeST

`private class Node`

Object	Memory Per Object (bytes)	Total Memory Usage (bytes)
Object Overhead	8	8
Point2D	32	32
Value	V	V
Node (reference)	8	2*8
int	4	4
TOTAL		52 + V

`public class KdTreeST`

Object	Memory Per Object (bytes)	Total Memory Usage (bytes)
Object Overhead	8	8
Node	36+V	N*(52+V)
TOTAL		N*(52+V) + 8

Running Time Analysis:

`insert()`, `get()`, and `contains()` methods:

All three of these methods should have a worst case running time of N (linear tree) and an average case of log(N). The readme.txt file indicates that their running time should be greater than PointST, which is also log(N), however I see no reason that this implementation of a binary tree would create any difference in the above three methods. The reason for using a KdTree is for faster implementation of the range and nearest methods.

Creating a 2D Tree:
StdRandom was used to generate N coordinates, which were then read from two arrays.

N	Elapsed Time (s)
100	0.003
200	0.003
400	0.005
800	0.008
1600	0.013
3200	0.021
6400	0.023
12800	0.031
25600	0.041

Nearest Neighbor Search:

PointST implementation performing 1000 nearest neighbor operations

INPUT	TEST 1 (s)	TEST 2 (s)	AVERAGE (s)	OPS PER S
input100K	48.13	49.61	48.87	20.46
input1M	48.47	46.93	47.70	20.96

KdTreeST implementation performing 10000000 nearest neighbor operations

INPUT	TEST 1 (s)	TEST 2 (s)	AVERAGE (s)	OPS PER S
input100K	24.11	24.81	24.46	408830
input1M	40.47	40.62	40.55	246609

Sample Output:

Draw is implemented as described, with red verticals and blue horizontals. Range is represented with a gray box in the lower left being the search area, and the pink points being points contained in that area. The cyan line from the center represents the result of the nearest method.