Project 5 (Kd-Trees)

Clarifications and Hints

Prologue

Project goal: create a symbol table data type whose keys are two-dimensional points

The zip file (`http://www.swamiiyer.net/cs210/kd_trees.zip`) for the project contains

- project specification (`kd_trees.pdf`)
- starter files
  - `BrutePointST.java`
  - `KdTreePointST.java`
- test script (`run_tests.py`)
- test data (`data/`)
- visualization clients (`RangeSearchVisualizer`, `NearestNeighborVisualizer`, and `BoidSimulator`)
- report template (`report.txt`)

Problems

Java interface `PointST<Value>` specifying the API for a symbol table data type whose keys are two-dimensional points represented as `Point2D` objects

method	description
`boolean isEmpty()`	is the symbol table empty?
`int size()`	number of points in the symbol table
`void put(Point2D p, Value val)`	associate the value val with point p
`Value get(Point2D p)`	value associated with point p
`boolean contains(Point2D p)`	does the symbol table contain the point p?
`Iterable<Point2D> points()`	all points in the symbol table
`Iterable<Point2D> range(RectHV rect)`	all points in the symbol table that are inside the rectangle $rect$
`Point2D nearest(Point2D p)`	a nearest neighbor to point p; `null` if the symbol table is empty
`Iterable<Point2D> nearest(Point2D p, int k)`	k points that are closest to point p

Problems

Problem 1 (*Brute-force Implementation*) Write a mutable data type `BrutePointST` that implements the above API using a red-black BST (`edu.princeton.cs.algs4.RedBlackBST`).

Hints

- Instance variable
  - A binary search tree to store the key/value pairs, `RedBlackBST<Point2D, Value> bst`
- `BrutePointST()`
  - Initialize instance variable appropriately
- `boolean isEmpty()`
  - Return `true` if the symbol table is empty, and `false` otherwise
- `int size()`
  - Return the number of key/value pairs in the symbol table
- `void put(Point2D p, Value val)`
  - Insert the given key/value pair into the symbol table
- `Value get(Point2D p)`
  - Return the value corresponding to the given key, or `null`
- `boolean contains(Point2D p)`
  - Return `true` if the given key is in the symbol table, and `false` otherwise

Problems

- `Iterable<Point2D> points()`
  - Return an iterable object containing all the keys in the symbol table
- `Iterable<Point2D> range(RectHV rect)`
  - Return an iterable object containing all the keys (ie, points) in the symbol table that are contained inside the given rectangle
- `Point2D nearest(Point2D p)`
  - Return a key (ie, point) from the symbol table that is closest to (and different from) the given key
- `Iterable<Point2D> nearest(Point2D p, int k)`
  - Return an iterable object containing upto `k` keys (ie, points) from the symbol table that are closest to (and different from) the given key

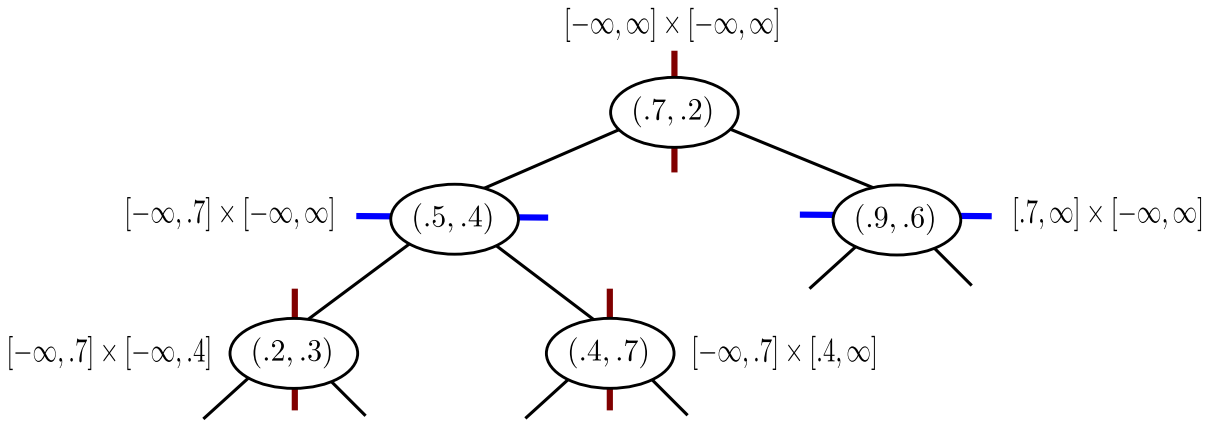
Problem 2 (*2d-tree Implementation*) Write a mutable data type `KdTreePointST` that uses a 2d-tree to implement the above symbol table API.

Hints

- Instance variables
  - Reference to the root of the 2d-tree, `Node root`
  - Number of nodes (ie, key/value pairs) in the tree, `int N`
- `KdTreePointST()`
  - Initialize instance variables appropriately
- `boolean isEmpty()`
  - Return `true` if the symbol table is empty, and `false` otherwise
- `int size()`
  - Return the number of key/value pairs in the symbol table

Problems

- Axis-aligned rectangles for the running example in the assignment writeup



- `void put(Point2D p, Value val)`
  - Call the private helper method `put()` with appropriate arguments to insert the key/value pair into the 2d tree; the parameter `lr` in this and other helper methods represents if the currrent node is x-aligned (`lr = true`) or y-aligned (`lr = false`)
- `Node put(Node x, Point2D p, Value val, RectHV rect, boolean lr)`
  - If `x = null`, return a new `Node` object built appropriately
  - If the key in `x` is the same as the given key, update the value in `x` to the given value
  - Make a recursive call to `put()` with appropriate arguments to insert the given key/value pair into the left subtree `x.lb` or the right subtree `x.rt` depending on how the values of `x.x`/`x.y` and `p.x`/`p.y` compare (use `lr` to decide which alignment to consider)
  - Return `x`

Problems

- `Value get(Point2D p)`
  - Call the private helper method `get()` with appropriate arguments to return the value corresponding to the given key, or `null`
- `Value get(Node x, Point2D p, boolean lr)`
  - If `x = null`, return `null`
  - If the key in `x` is the same as the given key, return the value in `x`
  - Make a recursive call to `get()` with appropriate arguments to find and return the value corresponding to the given key in the left subtree `x.lb` or the right subtree `x.rt` depending on how the values of `x.x`/`x.y` and `p.x`/`p.y` compare

Problems

- `boolean contains(Point2D p)`
  - Return `true` if the given key is in the symbol table, and `false` otherwise
- `Iterable<Point2D> points()`
  - Return an iterable object containing all the keys in the symbol table, enumerated in level order using a queue
- `Iterable<Point2D> range(RectHV rect)`
  - Call the private helper method `range()` passing it an empty queue of `Point2D` objects as one of the arguments, and return the queue
- `void range(Node x, RectHV rect, Queue<Point2D> q)`
  - If `x = null`, simply return
  - If `rect` contains the key in `x`, enqueue the key into `q`
  - Make recursive calls to `range()` starting at the left subtree `x.lb` and the right subtree `x.rt`
  - Incorporate the *range search* pruning rule mentioned in the assignment writeup

Problems

- `Point2D nearest(Point2D p)`
  - Return the key (ie, point) closest to (and different from) the given key by making a call to the private helper method `nearest()` with appropriate arguments
- `Point2D nearest(Node x, Point2D p, Point2D nearest, double nearestDistance, boolean lr)`
  - If `x = null`, return `nearest`
  - If the key in `x` is different from the given key and the squared distance between the two is smaller than the `nearestDistance`, update `nearest` and `nearestDistance` appropriately
  - Make recursive call to `nearest()` starting at the left subtree `x.lb`
  - Make recursive call to `nearest()` starting at the right subtree `x.rt`, using the value returned by the first call in an appropriate manner
  - Incorporate the *nearest neighbor* pruning rules mentioned in the assignment writeup

Problems

- `Iterable<Point2D> nearest(Point2D p, int k)`
  - Call the private helper method `nearest()` passing it an empty max-pq (consisting of `Point2D` objects and built with a suitable comparator from `Point2D`) as one of the arguments, and return the pq
- `void nearest(Node x, Point2D p, int k, MaxPQ<Point2D> pq, boolean lr)`
  - If `x = null` or if the size of pq is greater than or equal to `k`, simply return
  - If the key in `x` is different from the given key, insert it into pq
  - If the size of pq exceeds `k`, remove the maximum key from the pq
  - Make recursive calls to `nearest()` starting at the left subtree `x.lb` and the right subtree `x.rt`
  - Incorporate the *nearest neighbor* pruning rules

Epilogue

The `data` directory contains a number of sample input files, each with N points (x, y), where $x, y \in (0, 1)$

```
$ more data/input100.txt
0.042191 0.783317
0.390296 0.499816
0.666260 0.752352
0.369388 0.827540
0.196858 0.784460
0.972447 0.184467
0.761545 0.622944
0.257585 0.436938
0.458838 0.569394
0.817388 0.635645
0.958454 0.324192
0.403121 0.700930
...
```

Epilogue

We provide three visualization clients that serve as large-scale visual traces and we highly recommend using them for testing and debugging your solutions

1. `RangeSearchVisualizer` reads a sequence of points from a file (specified as a command-line argument), inserts those points into `BrutePointST` (red) and `KdTreePointST` (blue) based symbol tables, performs range searches on the axis-aligned rectangles dragged by the user, and displays the points obtained from the symbol tables in red and blue

```
$ java RangeSearchVisualizer data/input100.txt
```

2. `NearestNeighborVisualizer` reads a sequence of points from a file (specified as a command-line argument), inserts those points into `BrutePointST` (red) and `KdTreeSPointT` (blue) based symbol tables, performs k- (specified as the second command-line argument) nearest neighbor queries on the point corresponding to the location of the mouse, and displays the neighbors obtained from the symbol tables in red and blue

```
$ java NearestNeighborVisualizer data/input100.txt 5
```

3. `BoidSimulator` simulates the flocking behavior of birds, using a `BrutePointST` or `KdTreePointST` data type; the first command-line argument specifies which data type to use (`brute` or `kdtree`), the second argument specifies the number of boids, and the third argument specifies the number of friends each boid has

```
$ java BoidSimulator brute 100 10
```

Epilogue

Your project report (use the given template, `report.txt`) must include

- time (in hours) spent on the project
- short description of how you approached each problem, issues you encountered, and how you resolved those issues
- acknowledgement of any help you received
- other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Before you submit your files

- make sure your programs meet the input and output specifications by running the following command on the terminal

```
$ python run_tests.py -v [<problems>]
```

- make sure your programs meet the style requirements by running the following command on the terminal

```
$ check_style <program>
```

- make sure your report isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling/grammatical mistakes