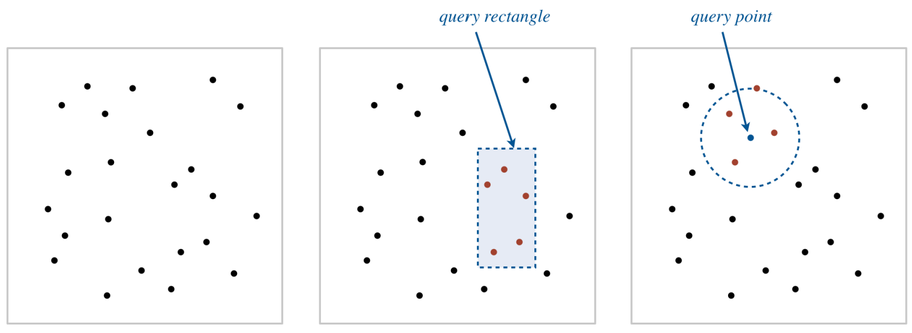
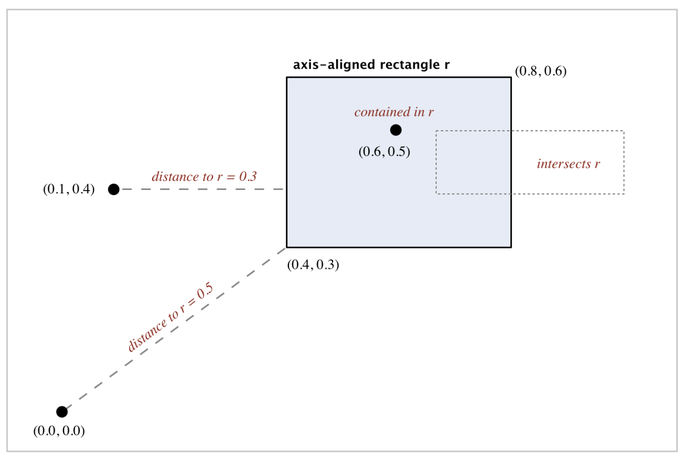
Write a data type to represent a set of points in the unit square (all points have *x*- and *y*-coordinates between 0 and 1) using a *2d-tree* to support efficient *range search* (find all of the points contained in a query rectangle) and *nearest-neighbor search* (find a closest point to a query point). 2d-trees have numerous applications, ranging from classifying astronomical objects to computer animation to speeding up neural networks to mining data to image retrieval.



Geometric primitives. To get started, use the following geometric primitives for points and axis-aligned rectangles in the plane.



- The immutable data type Point2D (part of `algs4.jar`) represents points in the plane. Here is the subset of its API that you may use:

```
public class Point2D implements Comparable<Point2D> {
   public Point2D(double x, double y)               // construct the point (x, y)
   public  double x()                               // x-coordinate
   public  double y()                               // y-coordinate
   public  double distanceTo(Point2D that)          // Euclidean distance between two points
   public  double distanceSquaredTo(Point2D that)   // square of Euclidean distance between two points
   public     int compareTo(Point2D that)           // for use in an ordered symbol table
   public boolean equals(Object that)               // does this point equal that object?
   public    void draw()                            // draw to standard draw
   public  String toString()                        // string representation
}
```

- The immutable data type RectHV (part of `algs4.jar`) represents axis-aligned rectangles. Here is the subset of its API that you may use:

```
public class RectHV {
   public    RectHV(double xmin, double ymin,      // construct the rectangle [xmin, xmax] x [ymin, ymax]
                    double xmax, double ymax)      // throw an IllegalArgumentException if (xmin > xmax) or (ymin > ymax)
   public  double xmin()                            // minimum x-coordinate of rectangle
   public  double ymin()                            // minimum y-coordinate of rectangle
   public  double xmax()                            // maximum x-coordinate of rectangle
   public  double ymax()                            // maximum y-coordinate of rectangle
   public boolean contains(Point2D p)               // does this rectangle contain the point p (either inside or on boundary)?
   public boolean intersects(RectHV that)           // does this rectangle intersect that rectangle (at one or more points)?
   public  double distanceTo(Point2D p)             // Euclidean distance from point p to closest point in rectangle
   public  double distanceSquaredTo(Point2D p)      // square of Euclidean distance from point p to closest point in rectangle
   public boolean equals(Object that)               // does this rectangle equal that object?
   public    void draw()                            // draw to standard draw
   public  String toString()                        // string representation
}
```

Do not modify these data types.

Brute-force implementation. Write a mutable data type `PointSET.java` that represents a set of points in the unit square. Implement the following API by using a red–black BST:

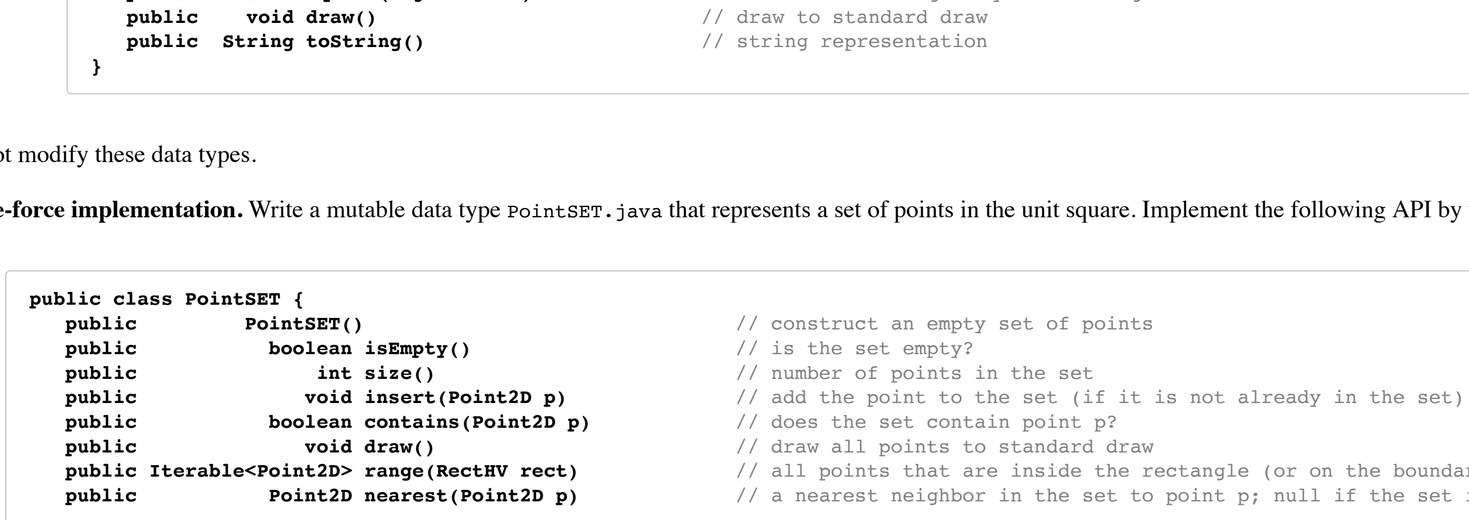
```
public class PointSET {
   public         PointSET()                               // construct an empty set of points
   public           boolean isEmpty()                      // is the set empty?
   public               int size()                         // number of points in the set
   public              void insert(Point2D p)              // add the point to the set (if it is not already in the set)
   public           boolean contains(Point2D p)            // does the set contain point p?
   public              void draw()                         // draw all points to standard draw
   public Iterable<Point2D> range(RectHV rect)             // all points that are inside the rectangle (or on the boundary)
   public           Point2D nearest(Point2D p)             // a nearest neighbor in the set to point p; null if the set is empty

   public static void main(String[] args)                  // unit testing of the methods (optional)
}
```

Implementation requirements. You must use either `SET` or `java.util.TreeSet`; do not implement your own red–black BST.

Corner cases. Throw an `IllegalArgumentException` if any argument is null. *Performance requirements*. Your implementation should support `insert()` and `contains()` in time proportional to the logarithm of the number of points in the set in the worst case; it should support `nearest()` and `range()` in time proportional to the number of points in the set.

2d-tree implementation. Write a mutable data type `KdTree.java` that uses a 2d-tree to implement the same API (but replace `PointSET` with `KdTree`). A *2d-tree* is a generalization of a BST to two-dimensional keys. The idea is to build a BST with points in the nodes, using the *x*- and *y*-coordinates of the points as keys in strictly alternating sequence.

- *Search and insert*. The algorithms for search and insert are similar to those for BSTs, but at the root we use the *x*-coordinate (if the point to be inserted has a smaller *x*-coordinate than the point at the root, go left; otherwise go right); then at the next level, we use the *y*-coordinate (if the point to be inserted has a smaller *y*-coordinate than the point in the node, go left; otherwise go right); then at the next level the *x*-coordinate, and so forth.



- *Draw*. A 2d-tree divides the unit square in a simple way: all the points to the left of the root go in the left subtree; all those to the right go in the right subtree; and so forth, recursively. Your `draw()` method should draw all of the points to standard draw in black and the subdivisions in red (for vertical splits) and blue (for horizontal splits). This method need not be efficient—it is primarily for debugging.

The prime advantage of a 2d-tree over a BST is that it supports efficient implementation of range search and nearest-neighbor search. Each node corresponds to an axis-aligned rectangle in the unit square, which encloses all of the points in its subtree. The root corresponds to the unit square; the left and right children of the root corresponds to the two rectangles split by the *x*-coordinate of the point at the root; and so forth.

- *Range search*. To find all points contained in a given query rectangle, start at the root and recursively search for points in *both* subtrees using the following *pruning rule*: if the query rectangle does not intersect the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). A subtree is searched only if it might contain a point contained in the query rectangle.

- *Nearest-neighbor search*. To find a closest point to a given query point, start at the root and recursively search in *both* subtrees using the following *pruning rule*: if the closest point discovered so far is closer than the distance between the query point and the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). That is, search a node only only if it might contain a point that is closer than the best one found so far. The effectiveness of the pruning rule depends on quickly finding a nearby point. To do this, organize the recursive method so that when there are two possible subtrees to go down, you always choose *the subtree that is on the same side of the splitting line as the query point* as the first subtree to explore—the closest point found while exploring the first subtree may enable pruning of the second subtree.

Clients. You may use the following interactive client programs to test and debug your code.

- KdTreeVisualizer.java computes and draws the 2d-tree that results from the sequence of points clicked by the user in the standard drawing window.
- RangeSearchVisualizer.java reads a sequence of points from a file (specified as a command-line argument) and inserts those points into a 2d-tree. Then, it performs range searches on the axis-aligned rectangles dragged by the user in the standard drawing window.
- NearestNeighborVisualizer.java reads a sequence of points from a file (specified as a command-line argument) and inserts those points into a 2d-tree. Then, it performs nearest-neighbor queries on the point corresponding to the location of the mouse in the standard drawing window.

Analysis of running time and memory usage (optional and not graded).

- Give the total memory usage in bytes (using tilde notation) of your 2d-tree data structure as a function of the number of points *n*, using the memory-cost model from lecture and Section 1.4 of the textbook. Count all memory that is used by your 2d-tree, including memory for the nodes, points, and rectangles.
- Give the expected running time in seconds (using tilde notation) to build a 2d-tree on *n* random points in the unit square. (Do not count the time to read in the points from standard input.)
- How many nearest-neighbor calculations can your 2d-tree implementation perform per second for input100K.txt (100,000 points) and input1M.txt (1 million points), where the query points are random points in the unit square? (Do not count the time to read in the points or to build the 2d-tree.) Repeat this question but with the brute-force implementation.

Web submission. Submit a .zip file containing only `PointSET.java` and `KdTree.java`. We will supply `algs4.jar`. Your may not call library functions except those in those in `java.lang`, `java.util`, and `algs4.jar`.