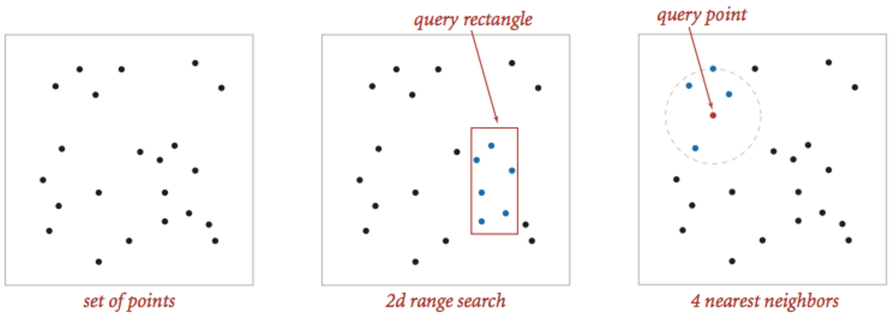
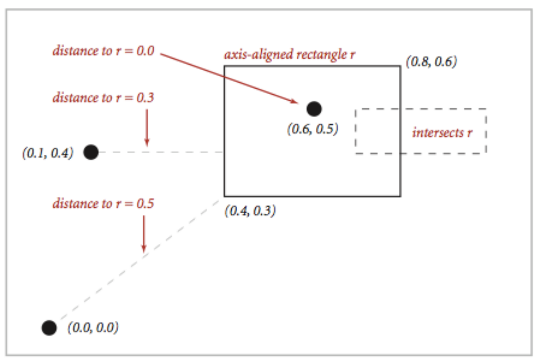
The purpose of this assignment is to create a symbol table data type whose keys are two-dimensional points. We'll use a 2d-tree to support efficient range search (find all the points contained in a query rectangle) and k-nearest neighbor search (find k points that are closest to a query point). 2d-trees 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.



Use the immutable data type `edu.princeton.cs.algs4.Point2D` for points in the plane. Here is the subset of its API that you may use:

method	description
`Point2D(double x, double y)`	construct the point (x, y)
`double x()`	x-coordinate
`double y()`	y-coordinate
`double distanceSquaredTo(Point2D that)`	square of Euclidean distance between this point and *that*
`Comparator<Point2D> distanceToOrder()`	a comparator that compares two points by their distance to this point
`boolean equals(Point2D that)`	does this point equal *that*?
`String toString()`	a string representation of this point

Use the immutable data type `edu.princeton.cs.algs4.RectHV` for axis-aligned rectangles. Here is the subset of its API that you may use:

method	description
`RectHV(double xmin, double ymin, double xmax, double ymax)`	construct the rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$
`double xmin()`	minimum x-coordinate of rectangle
`double xmax()`	maximum x-coordinate of rectangle
`double ymin()`	minimum y-coordinate of rectangle
`double ymax()`	maximum y-coordinate of rectangle
`boolean contains(Point2D p)`	does this rectangle contain the point p (either inside or on boundary)?
`boolean intersects(RectHV that)`	does this rectangle intersect *that* rectangle (at one or more points)?
`double distanceSquaredTo(Point2D p)`	square of Euclidean distance from point p to closest point in rectangle
`boolean equals(RectHV that)`	does this rectangle equal *that*?
`String toString()`	a string representation of this rectangle

Symbol Table API Here is a 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 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

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`).

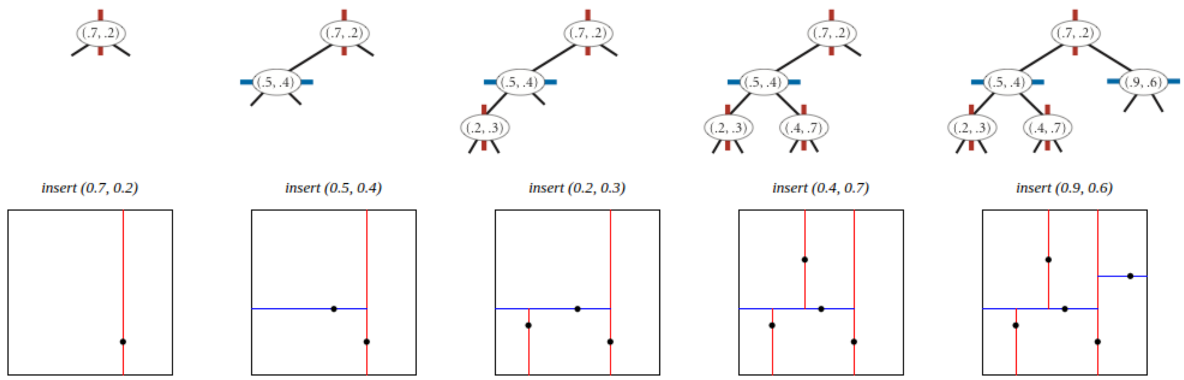
Corner cases. Throw a `java.lang.NullPointerException` if any argument is `null`.

Performance requirements. Your implementation should support `put()`, `get()` and `contains()` in time proportional to the logarithm of the number of points in the set in the worst case; it should support `points()`, `range()`, and `nearest()` in time proportional to the number of points in the symbol table.

```
$ java BrutePointST 0.661633 0.287141 0.65 0.68 0.28 0.29 5 < data/input10K.txt
st.empty()? false
st.size() = 10000
First 5 values:
  3380
  1585
  8903
  4168
  5971
  7265
st.contains((0.661633, 0.287141))? true
st.range([0.65, 0.68] x [0.28, 0.29]):
  (0.663908, 0.285337)
  (0.661633, 0.287141)
  (0.671793, 0.288608)
st.nearest((0.661633, 0.287141)) = (0.663908, 0.285337)
st.nearest((0.661633, 0.287141), 5):
  (0.663908, 0.285337)
  (0.658329, 0.290039)
  (0.671793, 0.288608)
  (0.65471, 0.276885)
  (0.668229, 0.276482)
```

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

- *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.



- *Level-order traversal.* The `points()` method should return the points in level-order: first the root, then all children of the root (from left/bottom to right/top), then all grandchildren of the root (from left to right), and so forth. The level-order traversal of the 2d-tree above is (0.7, 0.2), (0.5, 0.4), (0.9, 0.6), (0.2, 0.3), (0.4, 0.7).

The prime advantage of a 2d-tree over a BST is that it supports efficient implementation of range search, nearest neighbor, and k-nearest neighbor search. Each node corresponds to an axis-aligned rectangle, which encloses all of the points in its subtree. The root corresponds to the infinitely large square from $[(-\infty, -\infty), (+\infty, +\infty)]$; the left and right children of the root correspond 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). That is, you should search a subtree 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, you should search a node 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 your recursive method so that when there are two possible subtrees to go down, you choose first the subtree that is on the same side of the splitting line as the query point; the closest point found while exploring the first subtree may enable pruning of the second subtree.

- *k-nearest neighbor search.* Use the technique from kd-tree nearest neighbor search described above.

Corner cases. Throw a `java.lang.NullPointerException` if any argument is `null`.

```
java KdTreePointST 0.661633 0.287141 0.65 0.68 0.28 0.29 5 < data/input10K.txt
st.empty()? false
st.size() = 10000
First 5 values:
  0
  2
  1
  4
  3
```

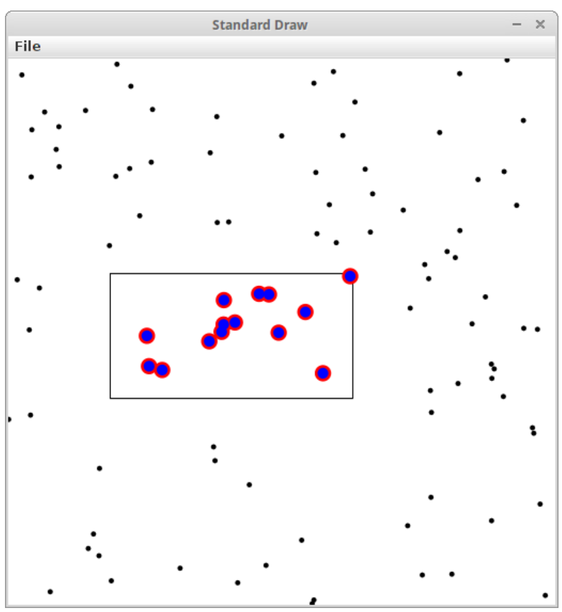
```
  62
st.contains((0.661633, 0.287141))? true
st.range([0.65, 0.68] x [0.28, 0.29]):
  (0.671793, 0.288608)
  (0.663908, 0.285337)
  (0.661633, 0.287141)
st.nearest((0.661633, 0.287141)) = (0.663908, 0.285337)
st.nearest((0.661633, 0.287141), 5):
  (0.668229, 0.276482)
  (0.65471, 0.276885)
  (0.671793, 0.288608)
  (0.658329, 0.290039)
  (0.663908, 0.285337)
```

Data Under the `data` directory, we provide several sample input files for testing.

Visualization Clients In addition to the test clients provided in `BrutePointST` and `KdTreePointST`, you may use the following interactive client programs to test and debug your code:

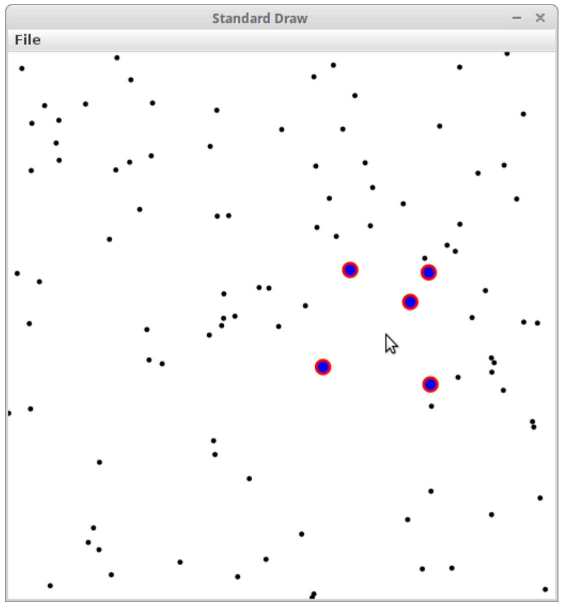
- `RangeSearchVisualizer` reads a sequence of points from a file (specified as a command-line argument) and inserts those points into `BrutePointST` and `KdTreePointST` based symbol tables `brute` and `kdtree` respectively. Then, it performs range searches on the axis-aligned rectangles dragged by the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

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



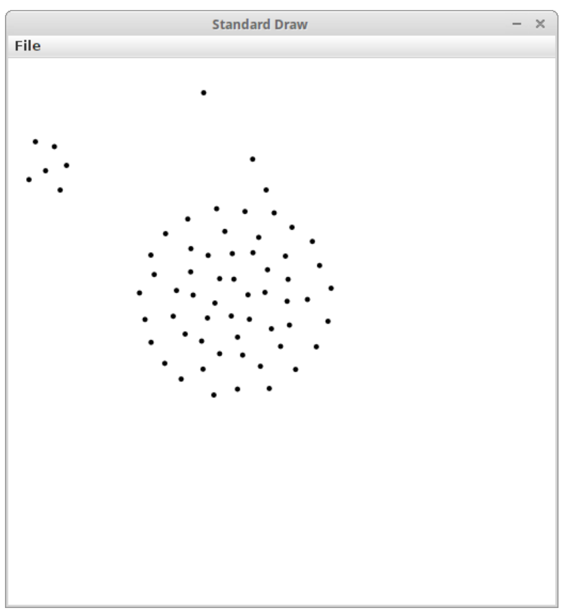
- `NearestNeighborVisualizer` reads a sequence of points from a file (specified as a command-line argument) and inserts those points into `BrutePointST` and `KdTreeSPointT` based symbol tables `brute` and `kdtree` respectively. Then, it performs k- (specified as the second command-line argument) nearest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

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



- `BoidSimulator` is an implementation of Craig Reynold's Boids program[1] to simulate the flocking behavior of birds, using a `BrutePointST` or `KdTreePointST` data type. The first command-line argument specifies which data type to use (`brute` for `BrutePointST` or `kdtree` for `KdTreePointST`), the second argument specifies the number of boids, and the third argument specifies the number of friends each boid has. [2]

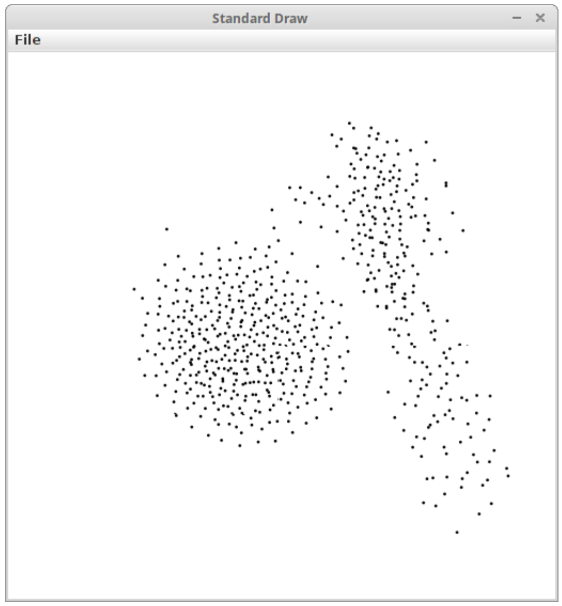
```
$ java BoidSimulator brute 100 10
```




```
$ java BoidSimulator kdtree 1000 10
```

[1] See www.en.wikipedia.org/wiki/Boids.

[2] Note that the program does not scale well with the number of boids when using `BrutePointST`, which is after all a brute-force implementation. However, the program does scale quite well when using `KdTreePointST`.



Files to Submit:

1. `BrutePointST.java`
2. `KdTreePointST.java`
3. `report.txt`

Before you submit:

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

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

  where the optional argument `<problems>` lists the problems (`Problem1`, `Problem2`, etc.) you want to test; all the problems are tested if no argument is given.

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

```
$ check_style <program>
```

  where `<program>` is the `.java` file whose style you want to check.

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

Acknowledgements This project is an adaptation of the Kd-Trees assignment developed at Princeton University by Kevin Wayne, with boid simulation by Josh Hug.