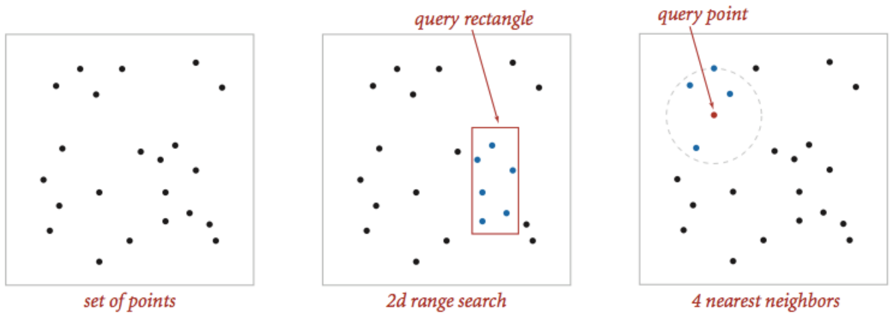
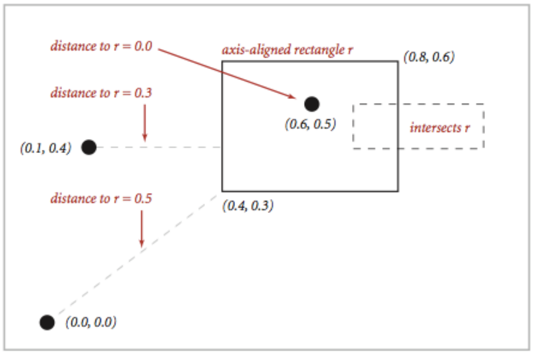
This document only contains the description of the project and the project problems. For the programming exercises on concepts related to the project, please refer to the project checklist .

The purpose of this project is to create a symbol table data type whose keys are two-dimensional points. We'll use a 2dTree 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). 2dTrees have numerous applications, ranging from classifying astronomical objects to computer animation to speeding up neural networks to mining data to image retrieval.



Geometric Primitives We will use the data types `dsa.Point2D` and `dsa.RectHV` to represent points and axis-aligned rectangles in the plane.



Symbol Table API Here is a Java interface `PointST<Value>` specifying the API for a symbol table data type whose keys are `Point2D` objects and values are generic objects:

PointST<Value>	
`boolean isEmpty()`	returns `true` if this symbol table is empty, and `false` otherwise
`int size()`	returns the number of key-value pairs in this symbol table
`void put(Point2D p, Value value)`	inserts the given point and value into this symbol table
`Value get(Point2D p)`	returns the value associated with the given point in this symbol table, or `null`
`boolean contains(Point2D p)`	returns `true` if this symbol table contains the given point, and `false` otherwise
`Iterable<Point2D> points()`	returns all the points in this symbol table
`Iterable<Point2D> range(RectHV rect)`	returns all the points in this symbol table that are inside the given rectangle
`Point2D nearest(Point2D p)`	returns the point in this symbol table that is different from and closest to the given point, or `null`
`Iterable<Point2D> nearest(Point2D p, int k)`	returns up to `k` points from this symbol table that are different from and closest to the given point

Problem 1. (*Brute-force Implementation*) Develop a data type called `BrutePointST` that implements the above API using a red-black BST (`RedBlackBST`) as the underlying data structure.

BrutePointST<Value> implements PointST<Value>	
`BrutePointST()`	constructs an empty symbol table

Corner Cases

- The `put()` method should throw a `NullPointerException()` with the message `"p is null"` if p is `null` and the message `"value is null"` if $value$ is `null`.
- The `get()`, `contains()`, and `nearest()` methods should throw a `NullPointerException()` with the message `"p is null"` if p is `null`.
- The `rect()` method should throw a `NullPointerException()` with the message `"rect is null"` if $rect$ is `null`.

Performance Requirements

- The `isEmpty()` and `size()` methods should run in time $T(n) \sim 1$, where n is the number of key-value pairs in the symbol table.
- The `put()`, `get()`, and `contains()` methods should run in time $T(n) \sim \log n$.
- The `points()`, `range()`, and `nearest()` methods should run in time $T(n) \sim n$.

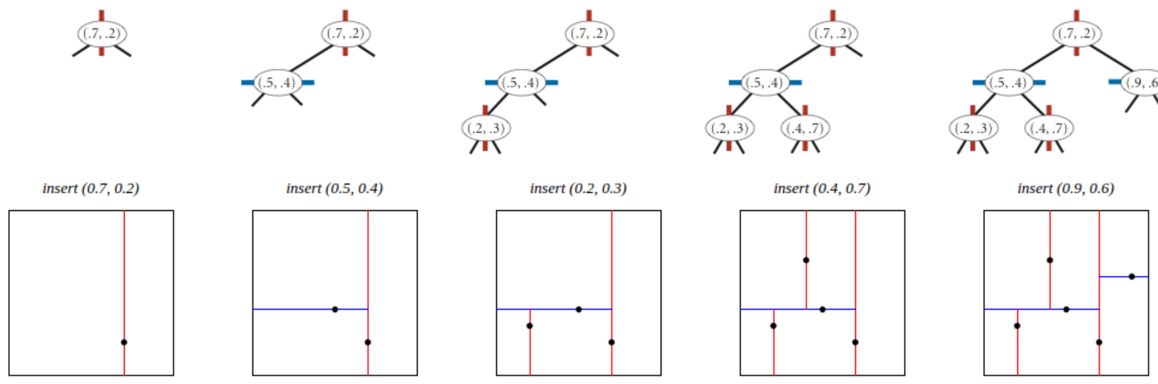
```
>_ ~/workspace/project5
$ java BrutePointST 0.975528 0.345492 5 < data/circle10.txt
st.size() = 10
st.contains((0.975528, 0.345492))? true
st.range([-1.0, -1.0] x [1.0, 1.0]):
  (0.5, 0.0)
  (0.206107, 0.095492)
  (0.793893, 0.095492)
  (0.024472, 0.345492)
  (0.975528, 0.345492)
  (0.024472, 0.654508)
  (0.975528, 0.654508)
  (0.206107, 0.904508)
  (0.793893, 0.904508)
  (0.5, 1.0)
st.nearest((0.975528, 0.345492)) = (0.975528, 0.654508)
st.nearest((0.975528, 0.345492), 5):
  (0.975528, 0.654508)
  (0.793893, 0.095492)
  (0.793893, 0.904508)
  (0.5, 0.0)
  (0.5, 1.0)
```

Problem 2. (*2dTree Implementation*) Develop a data type called `KdTreePointST` that uses a 2dTree to implement the above symbol table API.

KdTreePointST<Value> implements PointST<Value>	
`KdTreePointST()`	constructs an empty symbol table

A 2dTree 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 2dTree above is $(.7, .2)$, $(.5, .4)$, $(.9, .6)$, $(.2, .3)$, $(.4, .7)$.

The prime advantage of a 2dTree 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 nearest neighbor search described above.

Corner Cases

- The `put()` method should throw a `NullPointerException()` with the message `"p is null"` if p is `null` and the message `"value is null"` if *value* is `null`.

- The `get()`, `contains()`, and `nearest()` methods should throw a `NullPointerException()` with the message `"p is null"` if p is `null`.

- The `rect()` method should throw a `NullPointerException()` with the message `"rect is null"` if *rect* is `null`.

Performance Requirements

- The `isEmpty()` and `size()` methods should run in time $T(n) \sim 1$, where n is the number of key-value pairs in the symbol table.

- The `put()`, `get()`, `contains()`, `range()`, and `nearest()` methods should run in time $T(n) \sim \log n$.

- The `points()` method should run in time $T(n) \sim n$.

```
>_ ~/workspace/project5

$ java KdTreePointST 0.975528 0.345492 5 < data/circle10.txt
st.empty()? false
st.size() = 10
st.contains((0.975528, 0.345492))? true
st.range([-1.0, -1.0] x [1.0, 1.0]):
  (0.206107, 0.095492)
  (0.024472, 0.345492)
  (0.024472, 0.654508)
  (0.975528, 0.654508)
  (0.793893, 0.095492)
  (0.5, 0.0)
  (0.975528, 0.345492)
  (0.793893, 0.904508)
  (0.206107, 0.904508)
  (0.5, 1.0)
st.nearest((0.975528, 0.345492)) = (0.975528, 0.654508)
st.nearest((0.975528, 0.345492), 5):
  (0.5, 1.0)
  (0.5, 0.0)
  (0.793893, 0.904508)
  (0.793893, 0.095492)
  (0.975528, 0.654508)
```

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