# Topic Notes: Spatial Data Structures

Our next example of object-oriented design involves a *spatial data structure*. The task is to design data structures that can store points in the two-dimensional plane. These could easily be generalized to three dimensions.

To keep it simple, we will support the following operations on the collection of points:

- add

- get entry with same coordinates

- remove entry at coordinates

- size

- iteration

To ensure that we have coordinates available and to keep the structure very flexible, we will require that all objects in our spatial data structures extend the `Point2D.Double` class, from `java.awt.geom` in the standard Java API.

In the `Points2D` example, the `Point2DCollection` interface defines the small public interface, and `PointsList` is an `ArrayList`-based implementation of the interface.

There are also test cases: a small set of `Point2D.Double` objects in `SmallTest` and one that works with METAL data in `WaypointTesting`.

---

# Quadtrees

Since the list of points in the `PointsList` structure has no enforced ordering, a linear search running in $O(n)$ time is needed for the `get` and `remove` operations.

Since we are guaranteed to have coordinate data, this could be used in a number of ways to enforce an ordering, allowing for a more efficient search, likely at the expense of a more complex `add` operation.

Some ideas:

- maintain in sorted order by x-coordinate

- maintain in sorted order by y-coordinate

- maintain in sorted order by distance to the origin

Instead of these, we will turn to a tree structure to improve the efficiency of our point collection.

A *quadtree* is a spatial tree data structure. Each node in the tree is called a *quadrant*. Leaf nodes are called *terminal quadrants*. These terminal quadrants will contain collections of points whose coordinates fall within that quadrant's bounds.
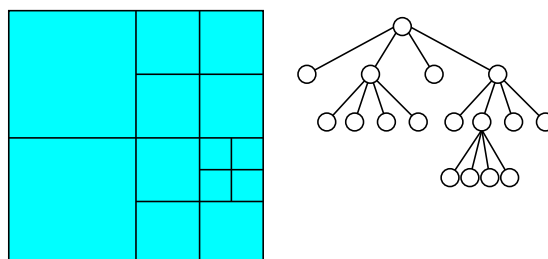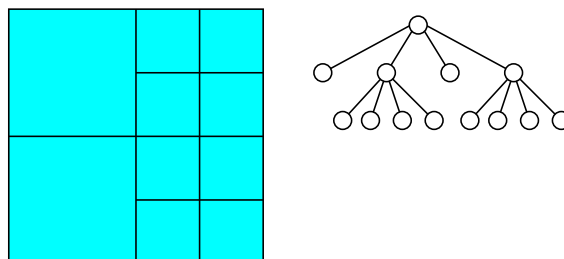
The tree's root represents the entire domain, which, in the case of this little example, is a square.
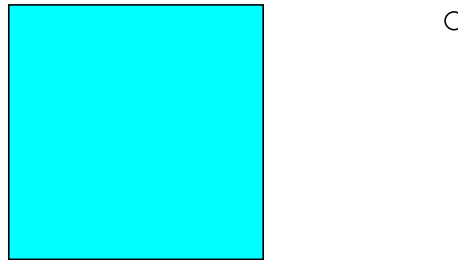
The four children of the root each represent one quarter of the space taken by the root.
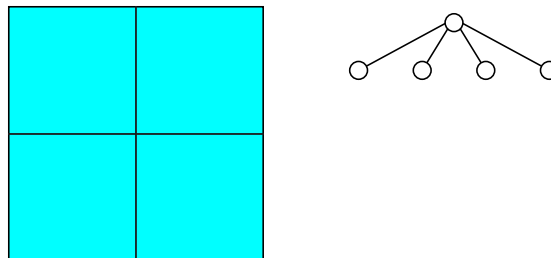
These children can then be divided in four, continuing down as many levels as desired. Different parts of the domain may be refined to different levels.

A quadtree can be used to store the points for our collection, organized by their coordinates as follows:

- The bounding box of the root quadrant covers the entire universe of possible points (so we must have a reasonable bound before adding points).

- Points are stored only in leaf quadrants.

- For non-leaf quadrants, we know in which child quadrant a point could be found/should be added based on its coordinates and the fact that the child quadrants are each 4 equal subsets of the quadrant.

- An upper limit on the number of points that can be stored in a leaf quadrant, called the *refinement threshold*, is used to make sure the (unordered) lists of points within each leaf quadrant does not get too long.

- When the number of points in a leaf exceeds the refinement threshold, that quadrant creates 4 child quadrants and redistributes its own points among its new children based on their coordinates.

This leads to much more efficient search operations on our structure, at the expense of an add operation that now must search its way down the tree for the correct leaf quadrant, and will occasionally trigger a more expensive refinement operation.

Since our quadtree structure only refines individual leaf quadrants when they exceed the refinement threshold, this is an *adaptive* quadtree structure.

`Points2D` has such an implementation that satisfies the `Point2DCollection` interface.