Computer Science 225
Advanced Programming
Siena College
Spring 2022

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

Implementations of spatial data structures could include other operations such as finding points near to a given set of coordinates.

To ensure that we have coordinates available and to keep the data structures 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 `PointSet2D` example, the `PointSet2D` interface defines the small public interface of our sets of points in two dimensions.

A few items to note about this interface:

- It is an interface that `extends` another interface.
- The type parameter has its own `extends`. This specification of `E` allows it to represent any type that either is or extends the `Point2D.Double` class. It is only used in the interface for the parameter to the `add` method, which must be of the specific type that is provided as the type parameter on construction of an instance of an implementing class. Other methods take a `Point2D,Double` as their parameter, which means they could be an `E`, but also could be a `Point2D.Double` or any other class that extends it. The `iterator` method, however, will return an `Iterator<E>`.
- A total of 5 methods are required of implementing classes: the four specifically listed in the interface, plus the one it inherits from `Iterable`, the `iterator` method.

List-Based Point Sets

First, we will consider `PointSetList`, which is an `ArrayList`-based implementation of the interface that we will work through in class. Its only instance variable is the `List` where we will store the points that are added. This allows for very straightforward implementations of the methods required by `PointSet2D`, as the list does much of the work for us.

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

Since the list of points in the `PointSetList` 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 (needed by the `get` and `remove` operations), 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

In the `PointSet2D` example, `PointSetSortedList` is a modificaton of `PointSetList` that can sort by either coordinate. This allows us to use a binary search based on the sorting order (though that code is not completed in the example).

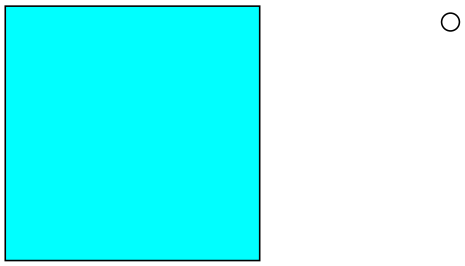
Better than this would be an implementation built on a balance binary search tree, allowing worst-case $O(\log n)$ implementations of `add`, `get`, and `remove` operations.

This is better, but still does not take advantage of the two-dimensional nature of our data. Instead of using a list or a traditional BST, we will turn to a tree structure to improve the efficiency of our point collection.

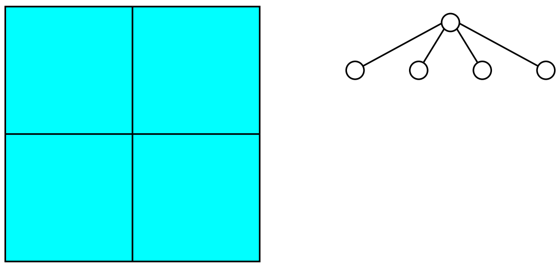
Quadtrees

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.

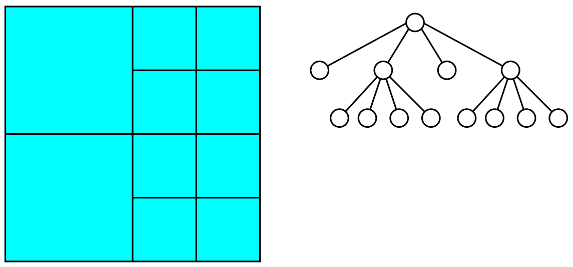
The tree's root represents the entire domain, which, in the case of this little example, is a square.

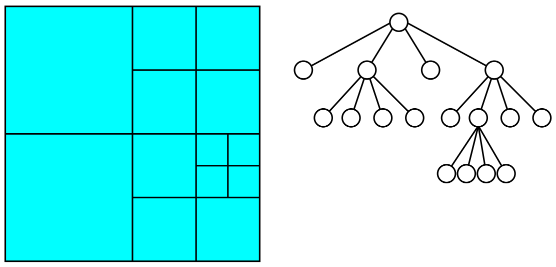


The four children of the root each represent one quarter of the space taken by the root.



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.

We can see this idea in action with METAL's visualization of quadtree construction.

In `PointSet2D` we will examine such an implementation that satisfies the `PointSet2D` interface.

Traversals

Strictly speaking, these point sets do not need to define an ordering of the points. However, we can take advantage of orderings induced by such a structure to come up with a one-dimensional ordering of, in this case, two-dimensional data.

We will use METAL HDX visualizations to experiment with these orderings.

We will also expand in class and in an upcoming assignment on the iterator in the quadtree.