Quad Tree

Ashish Pratap Singh

3 min read

Imagine you're building a system that tracks thousands of delivery drivers in a city. A naive approach would be to store each driver's coordinates (latitude, longitude) in a simple list or a database table.

Now, a customer requests a delivery. To find the nearest driver, your system would have to:

1. Get the customer's location.
2. Iterate through the **entire list** of drivers.
3. Calculate the distance from the customer to every single driver.
4. Find the minimum distance.

This is a full scan, an **O(n)** operation. It might work for 100 drivers, but it will be painfully slow for 100,000 drivers. As your data scales, this approach quickly becomes a performance bottleneck.

Similarly, imagine a game world with 1 million objects. To detect collisions, you need to quickly find all objects near the player without checking each one individually.

This is where spatial partitioning helps—and the Quad Tree is one of the most elegant solutions.

In this chapter, we’ll explore:

- What a QuadTrees is
- How QuadTrees work
- Their real-world applications
- A step-by-step implementation in Python
- Their advantages and trade-offs

---

1. What is a QuadTree?

A **Quad Tree** is a tree data structure where each internal node has exactly four children. It recursively partitions a 2D space into four equal quadrants, continuing to subdivide until each quadrant (a "leaf" node) contains a manageable number of points (e.g., less than 5) or reaches a maximum depth.

Each node in the tree represents a specific rectangular region of the space.

- The **root node** represents the entire 2D space (e.g., the whole city map).
- If a node contains more points than its capacity, it splits into **four child nodes**, representing the North-West, North-East, South-West, and South-East quadrants of its parent's region.
- This subdivision continues until a stopping condition is met, creating a hierarchy that adapts to the density of the data.

---

2. How QuadTrees Work

Building a Quad Tree is a straightforward, recursive process:

1. Start with a single node representing a large rectangular area (the "root").
2. Insert a data point. The tree finds the correct leaf node where the point belongs.
3. If adding this point causes the leaf node to exceed its

capacity (e.g., it now has 5 points, but the capacity is 4), the node is **subdivided** into four new child nodes (quadrants).

4. The points from the original overflowing node are then redistributed among the four new children.
5. This process repeats recursively for each subregion as more points are added.

Here’s a simple visual of how a space is partitioned:

```
+-----------------------+      +--------------------
|                       |      |         |
| .   .         .       |      | .   .   |      .
|                       | -->  |         |
| .         .   .       |      |---------+----------
|                       |      |         |
|     .                 |      | .       |  .   .
|                       |      |         |
+-----------------------+      +--------------------
(Many points in one region)    (Region is subdivide
```

This structure is powerful because it allows queries to quickly discard large, empty regions and "zoom in" on only the relevant areas of the map.

3. Operations on Quad Trees

The recursive structure of Quad Trees makes key operations very efficient.

a. Insertion

To insert a point, you start at the root and traverse down the tree, choosing the appropriate quadrant at each level until you find the correct leaf node. If that node exceeds its capacity after insertion, you split it.

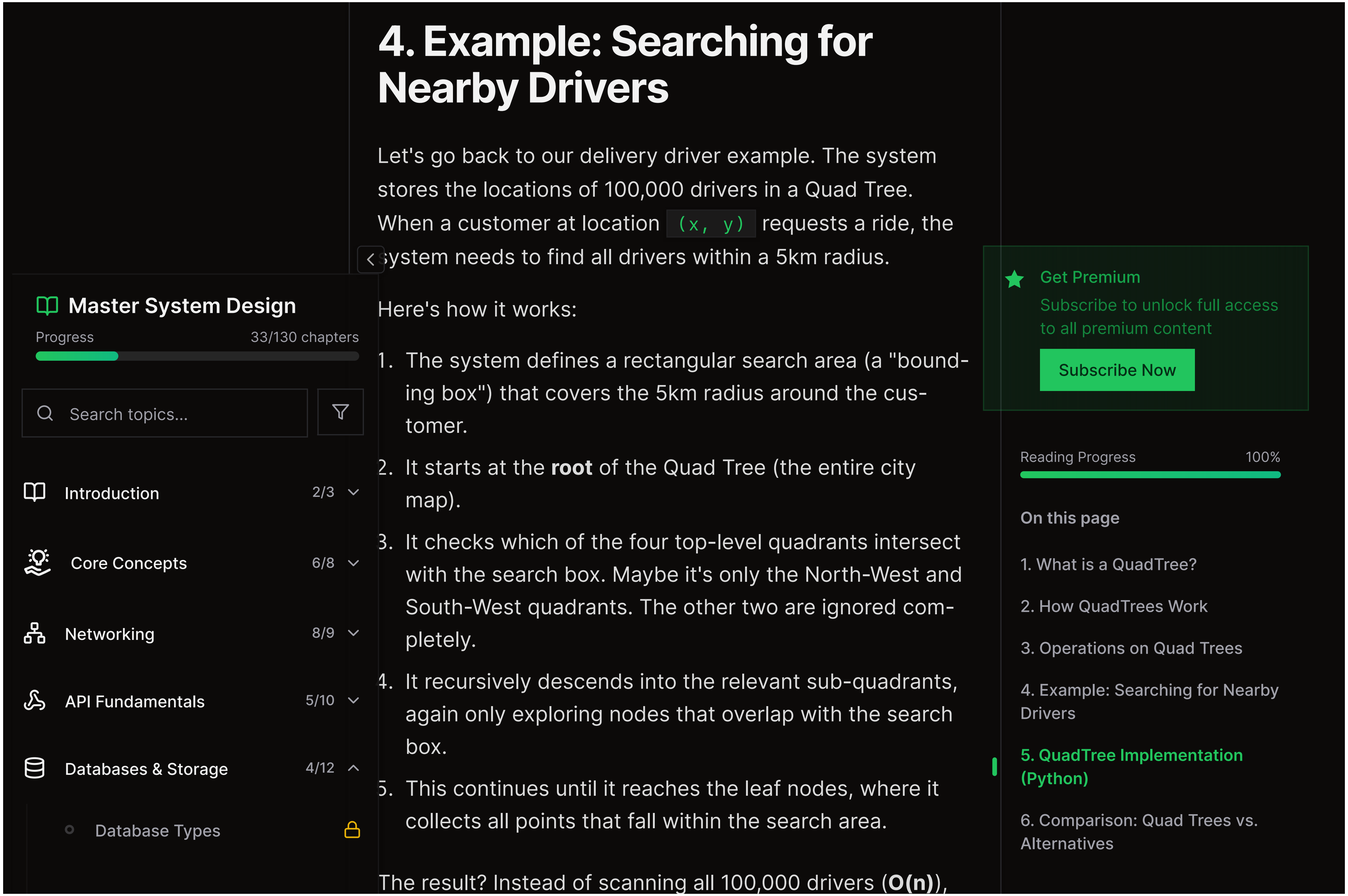
b. Range Query

This is where Quad Trees shine. To find all points within a certain area (a "range query"), you start at the root and check which of its four child quadrants **intersect** with your search area. You then recursively search *only* in those intersecting children, completely ignoring the others.

This pruning of the search space is what makes Quad Trees so fast.

c. Deletion

To delete a point, you find it and remove it. Afterward, you can optionally check if the parent node's children are now collectively under-capacity. If so, you can **merge** the children back into a single leaf node to save memory.



4. Example: Searching for Nearby Drivers

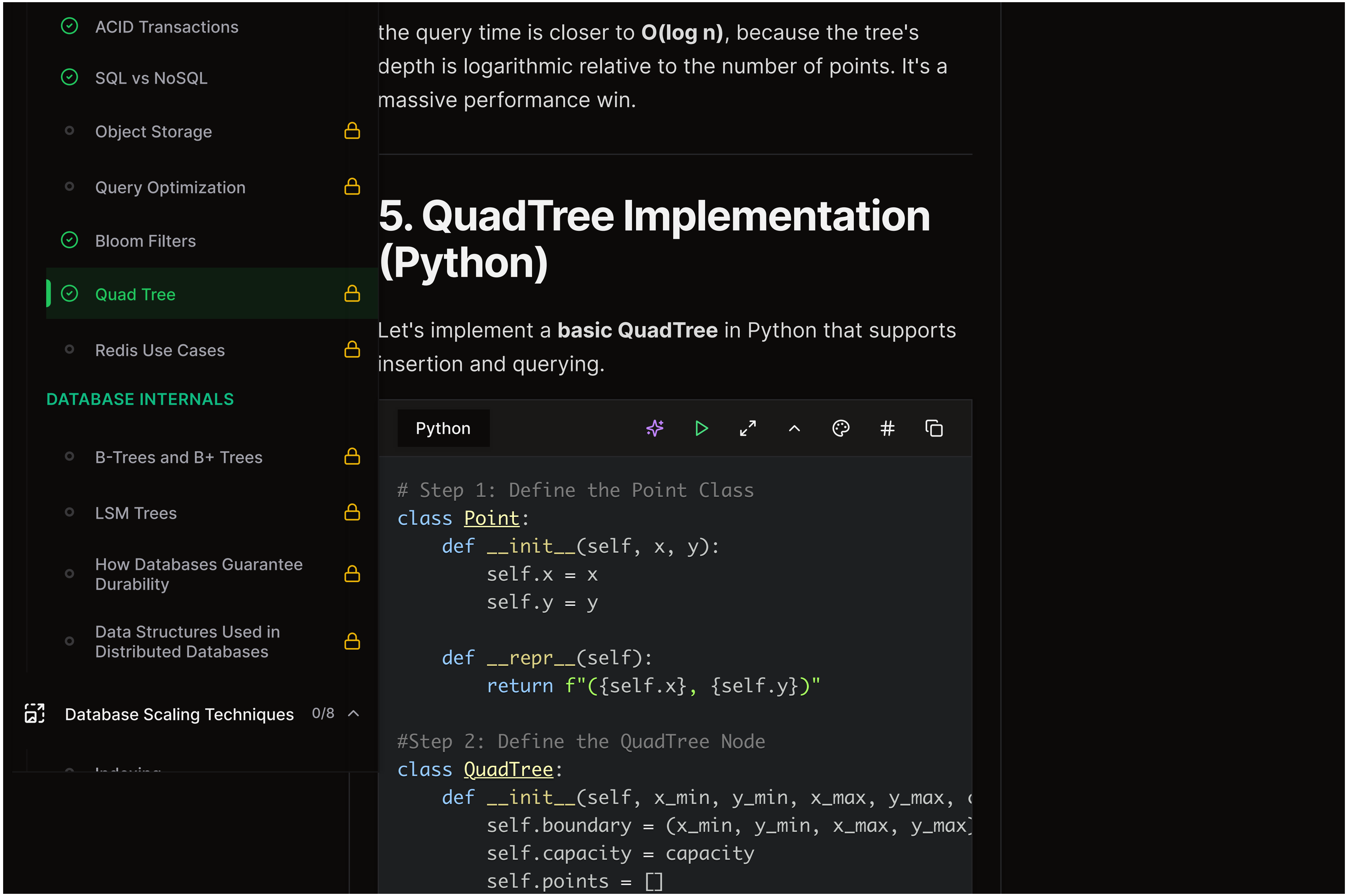
Let's go back to our delivery driver example. The system stores the locations of 100,000 drivers in a Quad Tree. When a customer at location `(x, y)` requests a ride, the system needs to find all drivers within a 5km radius.

Here's how it works:

1. The system defines a rectangular search area (a "bounding box") that covers the 5km radius around the customer.
2. It starts at the **root** of the Quad Tree (the entire city map).
3. It checks which of the four top-level quadrants intersect with the search box. Maybe it's only the North-West and South-West quadrants. The other two are ignored completely.
4. It recursively descends into the relevant sub-quadrants, again only exploring nodes that overlap with the search box.
5. This continues until it reaches the leaf nodes, where it collects all points that fall within the search area.

The result? Instead of scanning all 100,000 drivers (**O(n)**),

Reading Progress 100%



the query time is closer to **O(log n)**, because the tree's depth is logarithmic relative to the number of points. It's a massive performance win.

---

5. QuadTree Implementation (Python)

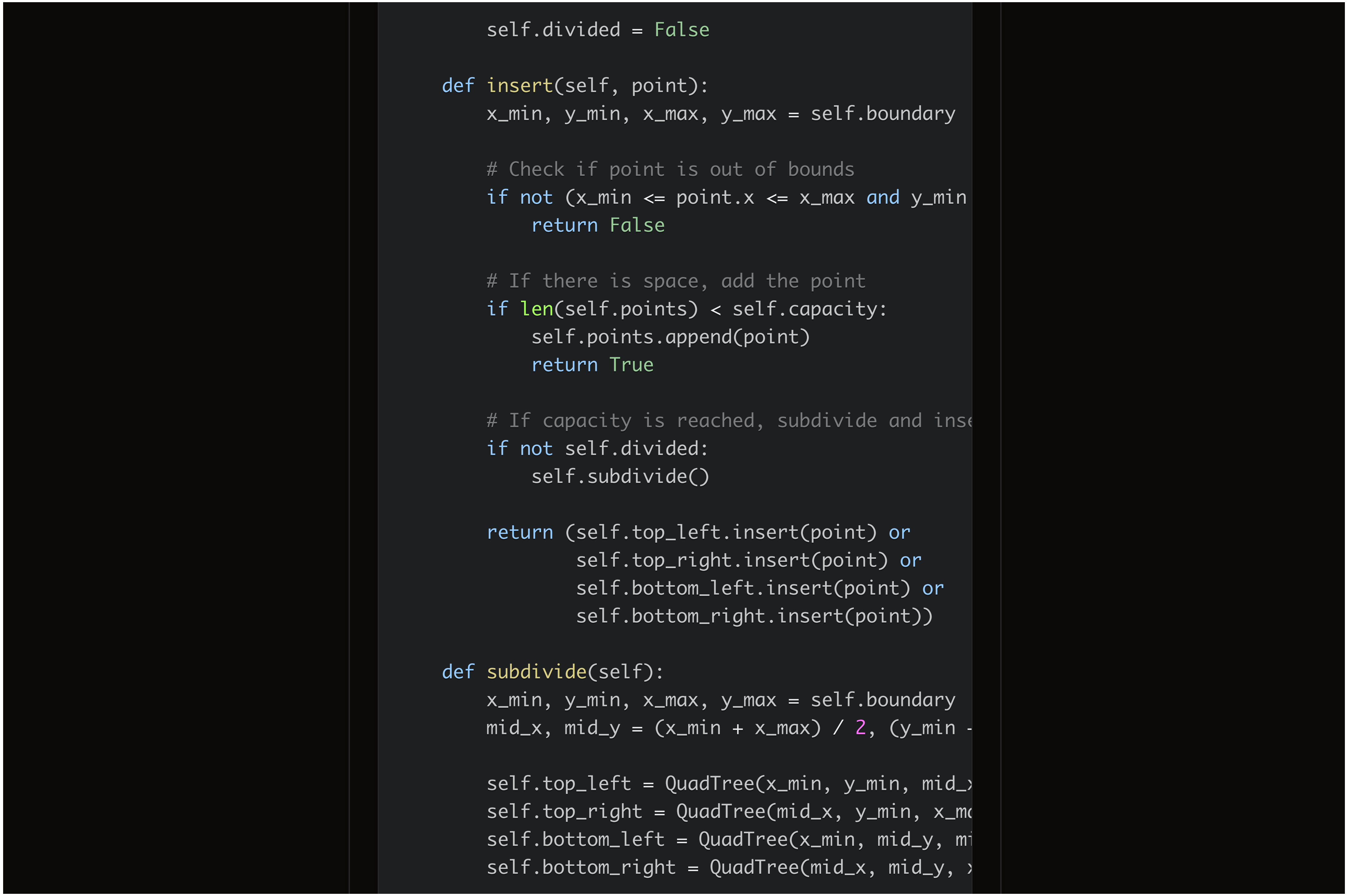
Let's implement a **basic QuadTree** in Python that supports insertion and querying.

Python

```python
# Step 1: Define the Point Class
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"({self.x}, {self.y})"

#Step 2: Define the QuadTree Node
class QuadTree:
    def __init__(self, x_min, y_min, x_max, y_max, 
        self.boundary = (x_min, y_min, x_max, y_max)
        self.capacity = capacity
        self.points = []
```



```
        self.divided = False

    def insert(self, point):
        x_min, y_min, x_max, y_max = self.boundary

        # Check if point is out of bounds
        if not (x_min <= point.x <= x_max and y_min
            return False

        # If there is space, add the point
        if len(self.points) < self.capacity:
            self.points.append(point)
            return True

        # If capacity is reached, subdivide and inse
        if not self.divided:
            self.subdivide()

        return (self.top_left.insert(point) or
                self.top_right.insert(point) or
                self.bottom_left.insert(point) or
                self.bottom_right.insert(point))

    def subdivide(self):
        x_min, y_min, x_max, y_max = self.boundary
        mid_x, mid_y = (x_min + x_max) / 2, (y_min -

        self.top_left = QuadTree(x_min, y_min, mid_
        self.top_right = QuadTree(mid_x, y_min, x_ma
        self.bottom_left = QuadTree(x_min, mid_y, mi
        self.bottom_right = QuadTree(mid_x, mid_y,
        self.divided = True

    def __repr__(self):
        return f"QuadTree({self.boundary}, Points: 

# Step 3: Testing the QuadTree
qt = QuadTree(0, 0, 10, 10)

# Insert random points
points = [Point(1, 1), Point(5, 5), Point(7, 8), Poi
for p in points:
    qt.insert(p)

print(qt)  # Should show the QuadTree structure
```

6. Comparison: Quad Trees vs. Alternatives

Quad Trees are not the only way to partition space. Here's how they stack up against other common structures.

Struc-ture	Use Case	Strength	Weakness
			Inefficient for

Grid (Uniform Grid)	Simple spatial partitioning	Very easy to implement; fast neighbor lookups.	unevenly distributed data; wastes memory in empty regions.
Quad Tree	Dynamic 2D spatial partitioning	Adapts to data density, saving memory.	Can become unbalanced with certain data patterns, leading to deep trees.
BSP Tree	3D game rendering, geometry	Handles arbitrary splitting planes, not just axes-aligned splits.	More complex logic and computation.
Geospatial databases (e.g., PostGIS)	Excellent for indexing non-point data like rectangles and polygons.	Excellent for indexing non-point data like rectangles and polygons.	More complex insertion and balancing algorithms than Quad Trees.

In interviews, a key insight to highlight is that **Quad Trees adapt dynamically to the data's density**, unlike fixed grids that can be wasteful in sparse regions and overloaded in dense ones.