Answering Queries with kd-Trees
Previously, we introduced the **kd-tree**, a spatial binary partition tree:

- Stores a set of **points in d-dimensional real space**, where each point \( p \) is represented as a \( d \)-element Java vector \( p[0, \ldots, d-1] \)
- Each node stores a point \( p \) and a **cutting dimension** \( i \), where \( 0 \leq i \leq d - 1 \)
- The left subtree contains points \( x \) such that \( x[i] < p[i] \) and the right contains points such that \( x[i] \geq p[i] \)
- Cutting dimension **varies** from node to node (e.g., cycles from 0 through \( d - 1 \), but other strategies are possible)
- What **other queries** can we answer?
Overview

Queries

- **Orthogonal range query:**
  - Given a point set P stored in a kd-tree, a query consists of a \textit{d-dimensional axis parallel rectangle} R
  - \textit{Range counting query}: How many points of P lie within R?
  - \textit{Range reporting query}: Report all the points of P that lie within R. (Java: Return an iterator for this set)

- **Nearest-neighbor query:**
  - Given a point set P stored in a kd-tree, a query consists of a point q
  - \textit{Nearest-distance query}: What is the distance to q’s closest point in P
  - \textit{Nearest-neighbor query}: Report the point that is closest to q
  - \textit{k-th Nearest-neighbor query}: Report the k closest points of P to q
Overview

Queries

- Orthogonal range queries
  - Given a medical database. Each patient associated with a vector of numbers (weight, height, blood pressure,...)
  - Want to count the number of patients whose weight, height, BP, etc. are within a given range of values
  - This is range counting query
Overview

Queries

- Nearest-neighbor queries
  - Given a medical database. Each patient associated with a vector of numbers (weight, height, blood pressure,...)
  - Given a sample patient $q$, we want to find other patients with similar health statistics, in order to determine likely outcomes.
  - This is a nearest-neighbor query
Orthogonal Range Queries

A Rectangle Class

- d-dimensional axis-aligned (hyper-)rectangles are useful geometric objects
- Rectangle class:
  - Defined by two d-dimensional points, low and high
  - The rectangle consists of the points $q$, such that $\text{low}[i] \leq q[i] \leq \text{high}[i]$, for $0 \leq i \leq d - 1$
Orthogonal Range Queries

A Rectangle Class

- Useful functions:
  - \( r \text{.contains}(\text{Point } q) \): true if \( r \) contains the point \( q \)
  - \( r \text{.contains}(\text{Rectangle } c) \): true if \( r \) contains the rectangle \( c \)
  - \( r \text{.isDisjointFrom}(\text{Rectangle } c) \): true if \( r \) has no overlap with rectangle \( c \)
Orthogonal Range Queries

A Rectangle Class

- **More useful functions:**
  - `r.distanceFrom(Point q)`: Minimum distance from q. Returns 0 if q lies within r

- **Functions useful for generating kd-tree cells:**
  - Given a rectangle r, a point x lying within r, and a cutting dimension cd
  - `r.leftPart(int cd, Point x)`: The subrectangle of r to the left (below) of x with respect to the cutting dimension cd
  - `r.rightPart(int cd, Point x)`: The subrectangle of r to the left (below) of x with respect to the cutting dimension cd
Orthogonal Range Queries

A Rectangle Class

- **Skeleton code for Rectangle class:**

```java
public class Rectangle {
    Point low;                                  // lower left corner
    Point high;                                 // upper right corner

    public Rectangle(Point low, Point high)     // constructor
        public boolean contains(Point q)            // do we contain q?
        public boolean contains(Rectangle c)        // do we contain rectangle c?
        public boolean isDisjointFrom(Rectangle c)  // disjoint from rectangle c?
        public float distanceTo(Point q)            // minimum distance to point q
        public Rectangle leftPart(int cd, Point x)  // left part from x
        public Rectangle rightPart(int cd, Point x) // right part from x
}
```
Orthogonal Range Queries

Answering Queries

- Intuition:
  - Each node of the kd-tree is associated with a cell, a rectangular region of space based on the intersection of the cuts of its ancestors
  - As a starting point, assume that there is a bounding box, the cell for the root of the tree
  - Use the cell-range relationship to avoid visiting subtrees whenever possible
Orthogonal Range Queries

Answering Queries

- **Cases:**
  - Node’s cell is disjoint from range: No need to visit this subtree
  - Cell is contained in the range: All of the points in this subtree lie in the range. Count them all. (Assume each node p stores its subtree size, p.size)
  - Cell partially overlaps the range:
    - Check whether the point in this node lies in the range - if so count it
    - Recurse on both children
int rangeCount(Rectangle r, KDNode p, Rectangle cell) {
    if (p == null) return 0;                // empty subtree
    else if (r.isDisjointFrom(cell))     // no overlap with range
        return 0;
    else if (r.contains(cell))          // range contains our entire cell?
        return p.size;                   // include all points in the count
    else {                               // range partially overlaps cell
        int count = 0;
        if (r.contains(p.point))        // consider this point
            count++;
        // apply recursively to children
        count += rangeCount(r, p.left,  cell.leftPart(p.cutDim, p.point));
        count += rangeCount(r, p.right, cell.rightPart(p.cutDim, p.point));
        return count;
    }
}
Orthogonal Range Queries

Example
Orthogonal Range Queries

Analysis

- Theorem: Given a balanced kd-tree with n points in 2D, range counting queries can be answered in $O(\sqrt{n})$ time.

- Terminology:
  - A node $p$ is stabbed by a line if the line intersects the interior of $p$’s cell
  - Observe that if a node is not stabbed by any of the four lines bounding the range, we will never recurse into this node

- Lemma: Given a balanced kd-tree with n points in 2D, the number of nodes stabbed by any axis-parallel line is $O(\sqrt{n})$.

- The above theorem follows directly from this.
Nearest Neighbor Searching

- Nearest Neighbors
  - Given a kd-tree and a query point \( q \), compute the closest point in the kd-tree to \( q \)
  - We assume that distances are measured using the Euclidean metric:
    \[
    \text{dist}(p, q) = \sqrt{(p_1 - q_1)^2 + \cdots + (p_d - q_d)^2}
    \]
  - How fast: Unfortunately, worst case is \( O(n) \). In practice, much better
Nearest Neighbor Searching

– To be continued
Summary