Answering Queries with kd-Trees
Previously, we introduced the \textbf{kd-tree}, a spatial binary partition tree:

- Stores a set of points in \textit{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 \textit{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 $d$-dimensional axis parallel rectangle $R$
  - **Range counting query:** How many points of $P$ lie within $R$?
  - **Range reporting query:** Report all the points of $P$ that lie within $R$. (Java: Return an iterator for the set $P \cap R$)

- **Nearest-neighbor query:**
  - Given a point set $P$ stored in a kd-tree, a query consists of a point $q$
  - **Nearest-distance query:** What is the distance to $q$’s closest point in $P$
  - **Nearest-neighbor query:** Report the point that is closest to $q$
  - $k$-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 biomedical statistics (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**
  - In a large database of *documents*, each document is encoded as a *vector* describing document properties (e.g., trigrams: number of occurrences of triples of characters)
  - Given a sample document \( q \), we want to find similar documents in the database
  - 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 $low[i] \leq q[i] \leq high[i]$, for $0 \leq i \leq d - 1$
Orthogonal Range Queries

A Rectangle Class

- Some useful functions:
  - `r.contains(Point q)`: true if \( r \) contains point \( q \)
  - `r.contains(Rectangle c)`: true if \( r \) contains rectangle \( c \)
    - Test \( r.\text{low}[i] \leq c.\text{low}[i] \) and \( c.\text{high}[i] \leq r.\text{high}[i] \), for all \( i \)
  - `r.isDisjointFrom(Rectangle c)`: true if \( r \) has no overlap with rectangle \( c \)
    - Test \( r.\text{high}[i] < c.\text{low}[i] \) or \( r.\text{low}[i] > c.\text{high}[i] \), for any \( i \)
    - (Not the same as \(!r.contains(c)\))
  - `r.distanceFrom(Point q)`: Min distance from \( q \), or 0 if \( q \) lies within \( r \)
Orthogonal Range Queries

A Rectangle Class

- For manipulating kd-tree cells:
  - Given a rectangle \( r \), a point \( s \) lying within \( r \), and a cutting dimension \( cd \)
    - \( r\.leftPart(int cd, Point s) \): Portion of \( r \) left of (below) \( s[cd] \)
      low is unchanged; high is same except high[cd] = s[cd]
    - \( r\.rightPart(int cd, Point s) \): Portion of \( r \) right of (above) \( s[cd] \)
      high is unchanged; low is the same except low[cd] = s[cd]
Orthogonal Range Queries

A Rectangle Class

- Basic signature of the 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)     // min distance to point q
    public Rectangle leftPart(int cd, Point s) // left part from s
    public Rectangle rightPart(int cd, Point s) // right part from s
}
```
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 root’s cell.
  - Use the *cell-range relationship* to avoid visiting subtrees whenever possible.

![Diagram](image)
Orthogonal Range Queries

Answering Queries

- **Cases:**
  - **Cell disjoint from range:** No overlap with range. **Return 0**
  - **Cell contained in range:** All 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 range:**
    - Check whether the **node’s point** lies in the range - **if so count it**
    - **Recurse** on both children

![Diagram](cell is disjoint from range) ![Diagram](cell is contained within range) ![Diagram](cell partially overlaps range)
Orthogonal Range Queries

Answering Queries

```c
int rangeCount(Rectangle r, KDNode p, Rectangle cell) {
    if (p == null) return 0;            // empty subtree
    else if (r.isDisjointFrom(cell))    // no overlap?
        return 0;
    else if (r.contains(cell))          // range contains our entire cell?
        return p.size;                  // …include all points in the count
    else {                               // partial overlap?
        int count = 0;
        if (r.contains(p.point))        // check 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.
Orthogonal Range Queries

Analysis

- Useful observation:
  - In 2D, if an axis-parallel line stabs a node $u$, then it stabs at most 2 of $u$’s grandchildren.
  - Therefore, the number of nodes stabbed at level $2^i$ is at most $2^i$. 

![Diagram showing orthogonal range queries in 2D with levels and stabs at different grandchildren nodes.]
Lemma: Given a balanced kd-tree with $n$ points in 2D, the number of nodes stabbed by any axis-parallel line is $O(\sqrt{n})$.

Proof:

- Let $h \approx \lg n$ be the tree height. Let $l$ be an axis parallel line
- If $l$ stabs a node $u$, then it stabs at most 2 of $u$’s grandchildren
- For every two levels of the tree, the number of stabbed nodes at most doubles
- Total number of stabbed nodes is roughly:
  $$\sum_{i=0}^{h/2} 2^i \approx 2^{h/2} = \left(2^h\right)^{1/2} \approx \left(2^{\lg n}\right)^{1/2} = (n)^{1/2} = \sqrt{n}$$

Proof of Theorem:

- Each of the 4 sides of the range stabs $O(\sqrt{n})$ nodes. Total time $\sim O(4\sqrt{n}) = O(\sqrt{n})$
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:
    \[
    \text{dist}(p, q) = \sqrt{(p_1 - q_1)^2 + \cdots + (p_d - q_d)^2}
    \]
  - Unfortunately, worst case is \( O(n) \), which happens if almost all points at same distance.
    In practice, much better
Overview:

- For simplicity, we will compute just the distance to the nearest neighbor.
  - Computing the actual point is a simple extension.
- Search operates recursively, starting from the root.
- Keep track of the minimum distance to the query seen so far - bestDist.
- Minimize the number of nodes visited:
  - Visit the subtree (left or right) that is closer to the query point first.
  - Don’t visit the other child if it cannot possibly contribute a closer point.
Nearest-Neighbor Searching

Answering Queries

- float nearNeighbor(Point q, Node p, Rectangle cell, float bestDist)
  - If p is null - return bestDist (empty subtree, no change in best)
  - Else:
    - Compute dist(q, p.point) and update bestDist if this is smaller
    - Compute child cells, leftPart and rightPart
    - Determine which child is closer to the query point (which side is q w.r.t. splitter)
    - Recursively visit the closer child - Update bestDist
    - Visit the farther child only if it is sufficiently close - Update bestDist
    - Return bestDist
Nearest-Neighbor Searching
Answering Queries

- float nearNeighbor(Point q, Node p, Rectangle cell, float bestDist)
float nearNeighbor(Point q, KDNode p, Rectangle cell, float bestDist) {
    if (p != null) {
        float thisDist = q.distanceTo(p.point);  // distance to p's point
        bestDist = Math.min(thisDist, bestDist);  // keep smaller distance

        int cd = p.cutDim;                       // cutting dimension
        Rectangle leftCell = cell.leftPart(cd, p.point);  // left child's cell
        Rectangle rightCell = cell.rightPart(cd, p.point);  // right child's cell

        if (q[cd] < p.point[cd]) {               // q is closer to left
            bestDist = nearNeighbor(q, p.left, leftCell, bestDist);
            if (rightCell.distanceTo(q) < bestDist) { // worth visiting right?
                bestDist = nearNeighbor(q, p.right, rightCell, bestDist);
            }
        } else {                                 // q is closer to right
            /* ... left-right symmetrical ... */
        }
    }
    return bestDist;
}
Nearest-Neighbor Searching

Example
Nearest-Neighbor Searching

Example
Nearest-Neighbor Searching

Example
Nearest-Neighbor Searching

Example

![Diagram of nearest-neighbor searching with points and a query point q. The diagram shows a tree structure with points at various levels, and the final closest point is marked.]
Summary

- Answering Queries with kd-trees
  - Principles:
    - Use recursion to visit subtrees
    - Maintain intermediate results
    - Avoid visiting subtrees whenever possible
  - Orthogonal range (counting) queries
  - Nearest-neighbor queries