Geometric Search:
- Nearest neighbors
- Range searching
- Point Location
- Intersection Search

Sofar:
- 1-dimensional keys
- Multi-dimensional data

Partition Trees:
- Tree structure based on hierarchical space partition
- Each node is associated with a region – cell
- Each internal node stores a splitter that subdivides the cell
- External nodes store points

Multi-Dim vs. 1-dim Search?
Similarities:
- Tree structure
- Balance \(O(\log n)\)
- Internal nodes split
- External nodes data

Differences:
- No (natural) total order
- Need other ways to discriminate and separate
- Tree rotation may not be meaningful

Point: A d-vector in \(\mathbb{R}^d\)

Representations:
- Scalars: Real numbers for coordinates, etc.
- Vectors: \(\mathbb{R}^d\) for other geometric objects

Class Point
- float[] coord // coords
- int getDim() \(\rightarrow\) coord.length
- float get(int i) \(\rightarrow\) coord[i]
- \(\ldots\) others: equality, distance, toString...
Point Quadtree:
- Each internal node stores a point
- Cell is split by horiz. + vert. lines through point

Quadtree:
- Partition trees
- Cell: Axis-parallel rectangle
- AABB: "Axis-aligned bounding box"

History: Bentley 1975
- Called it 2-d tree ($\mathbb{R}^2$)
- 3-d tree ($\mathbb{R}^3$)
- In short $kd$-tree (any dim)
- Where/which direction to split? → next

kd-Tree: Binary variant of quadtree
- Splitter: Horiz. or vert. line in 2-d (orthogonal plane ow.)
- Cell: Still AABB

Find/Pr Location:
Given a query point $q$, is it in tree, and if not which leaf cell contains it?
→ Follow path from root down (generalizing BST find)

Quadtrees & $kd$-Trees

Each external node corresponds to cell of final subdivision

Numerous variants!
- PR, PMR, QR, QX... see Sameti’s book
- Popular in 2-d apps
  (in 3-d, octrees)
- Don’t scale to high dim
- Out degree = 2d
- What to do for higher dims?
Kd-Tree Node:

```java
class KDNode {
    Point pt; // splitting point
    int cutDim; // cutting coordinate
    KDNode left; // low side
    KDNode right; // high side
}
```

**Example:**

```
q = (4,4) 
```

```
3,2
5,5
1,4
1,2
(3,2)
(1,4)
(5,5)
(2,1)
```

**Quad trees & Kd-Trees III**

- **Type:** Kd-Tree
- **Root:** Point
- **Leaf:** Point or null

**Analysis:** Find runs in time $O(h)$, where $h$ is height of tree.

**Theorem:** If pts are inserted in random order, expected height is $O(\log n)$

**Value**

```java
public Point find(Point q, KDNode p) {
    if (p == null) return null;
    else if (q == p.pt) return p.value;
    else if (p.onLeft(q)) return find(q, p.left);
    else return find(q, p.right);
}
```

**Helper:**

```java
class KDNode {
    boolean onLeft(Point q) {
        return q[cutDim] < pt[cutDim];
    }
}
```

Example:

```
3,2
5,5
1,4
1,2
(3,2)
(1,4)
(5,5)
(2,1)
```

**Example:**

```
q = (4,4)
```

```
3,2
5,5
1,4
1,2
```

**Analysis:** Find runs in time $O(h)$, where $h$ is height of tree.

**Theorem:** If pts are inserted in random order, expected height is $O(\log n)$

**Value**

```java
public Point find(Point q, KDNode p) {
    if (p == null) return null;
    else if (q == p.pt) return p.value;
    else if (p.onLeft(q)) return find(q, p.left);
    else return find(q, p.right);
}
```

**Helper:**

```java
class KDNode {
    boolean onLeft(Point q) {
        return q[cutDim] < pt[cutDim];
    }
}
```

**Quad trees & Kd-Trees III**

- **Type:** Kd-Tree
- **Root:** Point
- **Leaf:** Point or null

**Analysis:** Find runs in time $O(h)$, where $h$ is height of tree.

**Theorem:** If pts are inserted in random order, expected height is $O(\log n)$

**Value**

```java
public Point find(Point q, KDNode p) {
    if (p == null) return null;
    else if (q == p.pt) return p.value;
    else if (p.onLeft(q)) return find(q, p.left);
    else return find(q, p.right);
}
```

**Helper:**

```java
class KDNode {
    boolean onLeft(Point q) {
        return q[cutDim] < pt[cutDim];
    }
}
```
KDNode insert(Point pt, KDNode p, int cd)
{
    if (p == null)  // fell out?
        p = new KDNode(pt, cd)  // new leaf node
    else if (p.point == pt)
        Error! Duplicate key
    else if (p.onLeft(pt))
        p.left = insert(pt, p.left, (cd+1)%dim)
    else
        p.right = insert(pt, p.right, (cd+1)%dim)

    return p
}

Kd-Tree Insertion:
(Similar to std. BSTs)
- Descend tree until
  - Descend path to leaf
  - If found:
    - leaf node → just remove
    - internal node
      → find replacement
      → copy here
      → recur: delete replacement
  - Error! Duplicate key
- Create new node
- Set cutting dim
- Recbalance by Rebuilding:
  - Rebuild subtrees as with scapegoat trees
  - O(log n) amortized
  - Find: O(log n) guaranteed.

Analysis:
Runtime: O(h)
Can we balance the tree?
- Rotation does not make sense!!
**Kd-Trees:**
- Partition trees
- Orthogonal split
- Alternate cutting
  - Dimension \( x, y, z, \ldots \)
- Cells are axis-aligned rectangles (AABB)

**Rectangle methods for kd-cells:**
- Split a cell \( r \) by a split pt \( s \in r \), along cut dimension \( d \)
- Contains pt \( q \in \mathbb{R}^d \) iff
  - Low: \( l \leq q_i \leq \text{high} \) for \( i \in \{1, \ldots, d\} \)

**Queries?**
- Orthogonal range queries
  - Given query rect. \((AABB)\)
  - Count/report pts in this rect.
- Other range queries?
  - Circular disks
  - Halfplane
- Nearest neighbor queries
  - Given query pt, return closest pt in the set
  - Find \( k \)th closest point
  - Find farthest point from \( q \)

This Lecture: \( O(n) \) time alg. for orthog. range counting queries in \( \mathbb{R}^2 \)
- General \( \mathbb{R}^d \): \( O(n^{1-1/d}) \)

**Kd-Tree Queries**

**Axis-Aligned Rect in \( \mathbb{R}^d \)**
- Defined by two pts: \( \text{low}, \text{high} \)
  - Contains pt \( q \in \mathbb{R}^d \) iff
    - Low: \( l \leq q_i \leq \text{high} \) for \( i \in \{1, \ldots, d\} \)

**Useful methods:**
- Let \( r, c \): Rectangles
- **Point**
  - \( r \text{.contains}(q) \)
  - \( r \text{.contains}(c) \)
  - \( r \text{.isDisjointFrom}(c) \)
Orthogonal Range Query

Assume:
- Each node \( p \) stores:
  - \( p.pt \): splitting point
  - \( p.cutDim \): cutting dimension
  - \( p.size \): no. of pts in \( p \)’s subtree
- Tree stores ptr. to root and bounding box for all pts.
- Recursive helper stores current node \( p \) + \( p \)’s cell.

Cases:
- \( p == \) null → fell out of tree → 0
- Query rect is disjoint from \( p \)’s cell
  → return 0
  → no point of \( p \) contributes to answer
- Query rect contains \( p \)’s cell
  → return \( p.size \)
  → every point of \( p \)’s subtree contributes to answer.
- Otherwise:
  Rect + cell overlap → Recurse on both children

class Rectangle {
    private Point low, high
    public Rect (Point l, Point h)
    " boolean contains(Point q)
    " boolean contains(Rect c)
    " Rect leftPart(int cd, Points)
    " Rect rightPart()
    }

Kd-Tree Queries

Cases:
- \( p == \) null → fell out of tree → 0
- Query rect is disjoint from \( p \)’s cell
  → return 0
  → no point of \( p \) contributes to answer
- Query rect contains \( p \)’s cell
  → return \( p.size \)
  → every point of \( p \)’s subtree contributes to answer.
- Otherwise:
  Rect + cell overlap → Recurse on both children

\[
\text{int rangeCount(Rect R, KDNode p, Rect cell)}
\]

\[
\begin{cases}
\text{if (} p == \text{null) return 0 // fell out of tree} \\
\text{else if (R is disjoint from (cell)) return 0 // overlap} \\
\text{else if (R.contains(cell)) return p.size // take all} \\
\text{else } \{ \text{int } ct = 0 \text{ // partial overlap} \\
\text{if (R.contains(p.pt)) } ct++ \text{ // pt in range} \\
\text{ct} += \text{rangeCount}(R, p.left, cell.leftPart(p.cutDim, p.pt)) \\
\text{ct} += \text{rangeCount}(R, p.right, cell.rightPart..) \\
\text{return ct}
\end{cases}
\]