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

So far: 1-dimensional keys  
- Multi-dimensional data  
- Applications:  
  - Spatial databases & maps  
  - Robotics & Auton. Systems  
  - Vision/Graphics/Games  
  - Machine Learning

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

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 + separate  
- Tree rotation may not be meaningful

Quadtree & k-d Trees:

Points: A d-vector in $\mathbb{R}^d$  
$p = (p_1, \ldots, p_d)$  
$p \in \mathbb{R}^d$

Representations:  
- Scalars: Real numbers for coordinates, etc.  
  float
- Points: $p = (p_1, \ldots, p_d)$ in real d-dim space $\mathbb{R}^d$
- Other geom objects: Built from these
**Point Quadtree:**
- Each internal node stores a point.
- Cell is split by horizon. + vertical lines through point.

Each external node corresponds to cell of final subdivision.

**Quadtree:**
- Partition trees
  - Cell: Axis-parallel rectangle
    - [AABB: "Axis-aligned bounding box"]
  - Splitter: Subdivides cell into four (generally $2^d$) subcells.

**kd-Tree:**
- Binary variant of quadtree
  - Splitter: Horizontal or vertical line in 2-d (orthogonal plane $aw$)
  - Cell: Still AABB
    - Left: left/below
    - Right: right/above

History: Bentley 1975
- Called it 2-d tree ($R^2$)
- 3-d tree ($R^3$)
- In short, kd-tree (any dim)

- Where/which direction to split? → next

**Quadtree Analysis:**
- Numerous variants!
  - PR, PMR, QR, QX... see Samet's book
  - Popular in 2-d apps
    - In 3-d, octrees
  - Don't scale to high dim
    - Out degree = $2^d$
  - What to do for higher dims?
Example:

Kd-Tree Node:

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

How do we choose cutting dim?:

- **Standard k-d tree**: cycle through them (e.g., d = 3: 1, 2, 3, 1, 2, 3...)
  based on tree depth
- **Optimized k-d tree**: (Bentley)
  - Based on widest dimension of pts in cell.

Analysis:

Find runs in time \( O(h) \), where \( h \) is the height of tree.

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

Value

```java
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);
}
```
KDTree Insertion:

```
KDNode insert(Point x, Value v, KDNode p, int cd) { 
    if (p == null)  // fell out? 
        return new KDNode(x, v, p, null); 
    else if (p.pt == x) 
        return newKDNode(x, v, p, null); 
    else if (p.onLeft(x)) 
        p.left = insert(x, v, p.left, (cd+1)%dim); 
    else 
        p.right = insert(x, v, p.right, (cd+1)%dim); 
    return p; 
}
```

Deletion:

- Descend path to leaf
- If found:
  - leaf node → just remove
  - internal node → recursively delete replacement

Example:

```
Example: insert(3,4)
```

Analysis:

Run time: \(O(h)\)

Can we balance the tree?

- Rotation does not make sense

```
Rotation does not make sense
```

Rebalance by Rebuilding:

- Rebuild subtrees as with scapegoat trees
- \(O(\log n)\) amortized
- Find: \(O(\log n)\) guaranteed