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

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

Sofar: 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

External nodes store pts.

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

Quadtrees & kd-Trees

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

Class Point
```java
float[] coord // coords
Point(int d)
  // coord = new float[d]
int getDim() \rightarrow coord.length
float get(int i) \rightarrow coord[i]
  // others: equality, distance
toString...
```
Quadtree:
- Each internal node stores a point
- Cell is split by horiz. + vertic. lines through point
- Quadtree: (abstractly)
  - Partition trees
  - Cell: Axis-parallel rectangle
  - Splitter: Subdivides cell into four (generally $2^d$) subcells

Find/Pt 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)

Point Quadtree:
- Each internal node stores a point
- Cell is split by horiz. + vertic. lines through point

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

Quadtree & $kd$-Trees

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

Find/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 - Analysis:
- Numerous variants!
  - $PR$, $PMR$, $QR$, $QX$, ... see Samet's book
- Popular in 2-d apps
  - In 3-d, octtrees
- Don't scale to high dim
  - Out degree = $2^d$

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

Find/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)

Each external node corresponds to cell of final subdivision

Quadtree:
- Each internal node stores a point
- Cell is split by horiz. + vertic. lines through point

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

Find/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 - Analysis:
- Numerous variants!
  - $PR$, $PMR$, $QR$, $QX$, ... see Samet's book
- Popular in 2-d apps
  - In 3-d, octtrees
- Don't scale to high dim
  - Out degree = $2^d$

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

Find/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)
**Example:**

Kd-Tree Node:

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

**Example:**

```java
find(q) calls find(q, root)
```

**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:**

```
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);
}
```

**Quad trees & KD Trees III**

**How do we choose cutting dim?**

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

```
if (q == p.pt) found!
if q[cd] < p.pt[cd] => left
if q[cd] > p.pt[cd] => right
```

```
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)
        return p
            // 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 cutting,
  → find pt → Error: duplicate
  → falling out (Although we draw extended trees, let’s assume standard trees)
  → create new node
  → set cutting dim

Quadtrees & Kd-Trees

Deletion:
- Descend path to leaf
- If found:
  - leaf node → just remove
  - internal node → find replacement
    → copy here
    → recur. delete replacement

This is the hardest part.
See Latex notes.

Rebalance by Rebuilding:
- Rebuild subtrees as with scapegoat trees
- O(log n) amortized
- Find: O(log n) guaranteed.

Example:
insert(3,4)

Analysis:
Runtime: O(h)

Can we balance the tree?
- Rotation does not make sense!!

0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7

Graft Node insertion: [1 2 3 4 5 6 7]