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

So far: 1-dimensional keys
- Multi-dimensional data
- Applications: \((x, y, z)\)
  - 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.

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

Quadtrees & kd-Trees

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

Representations:
- Scalars: Real numbers for coordinates, etc.
- Floats

Java: \(p[0..d-1]\)

Points: \(p = (p_1, \ldots, p_d)\) in real \(d\)-dim space \(\mathbb{R}^d\)

Other geom objects:
- Built from these
Quadtree: (abstractly)
-Partition trees
-Cell: Axis-parallel rectangle
[AAAB - Axis-aligned bounding box]
   split
   NW NE SW SE
-Splitter: Subdivides cell into four (generally $2^d$) subcells

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

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 ($R^2$)
-k-d tree 3-d tree ($R^3$)
-In short k-d-tree (any dim)
-Where/which direction to split?
  → next

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

Leaf cell:

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)

Each external node corresponds to cell of final subdivision

Null ptr

Quadtrees & kd-Trees II

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 = $2^d$
-What to do for higher dims?
Example: Splittings rule: Alternate

```
Example: find((q) calls find((q, root))
```

**Point kd-tree**

```
Point pt // splitting point
int cutDim // cutting coordinate
KDNode left // low side
KDNode right // high side
```

**Quadtrees & kd-Trees III**

```
Find point q in subtree
rooted at p with cutDim cd:
```

```
Helper: class KDNode {

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

```
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);
}
```

**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)

**How do we choose cutting dim?**

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

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

```
Quad trees &
kd-Trees III
```

```
How do we choose cutting dim?
```

```
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);
}
```

```
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)

```
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)

```
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)

```
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)

```
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)
Kd-Tree Insertion:

(Similar to std. BSTs)

- Descend tree until cutting dimension to use
  - find pt ➔ Error: duplicate
  - falling out ➔ create new node
  - set cutting dim

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

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

Example:
insert(3,4)

KDNode insert(Point pt, KDNode p, int cd)
if (p == null) // fell out?
    p = new KDNode(pt, cd)
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

Analysis:
Run time: O(h)

Can we balance the tree?
- Rotation does not make sense!!
Announcements: 3/16

- Midterm - still grading
- Programming Assignment 2
  - Coming "soon"
  - Due Apr 5
  - kd-trees ...

Convention
Ties broken in favor of right.