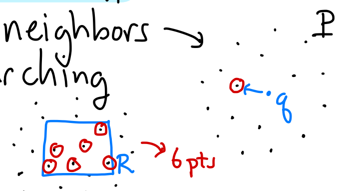
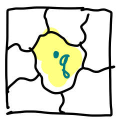
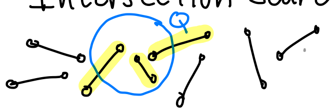
Geometric Search:

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





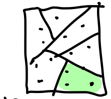


So far: 1-dimensional keys

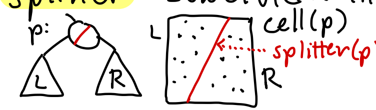
- Multi-dimensional data
- Applications:
  - Spatial databases + maps
  - Robotics + Auton. Systems
  - Vision / Graphics / Games
  - Machine Learning
- ...

Quadtrees & kd-Trees I

Partition Trees:



- Tree structure based on hierarchical space partition
- Each node is associated w. a region — cell
- Each internal node stores a splitter — subdivides the cell



- External nodes store pts.

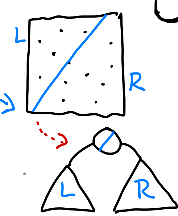
Point: A d-vector in $\mathbb{R}^d$

$p = (p_1, \ldots, p_d) \quad p_i \in \mathbb{R}$

Multi-Dim vs. 1-dim Search?

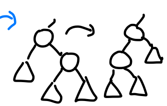
Similarities:

- Tree structure
- Balance $O(\log n)$
- Internal nodes - split
- External nodes - data



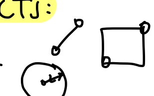
Differences:

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



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

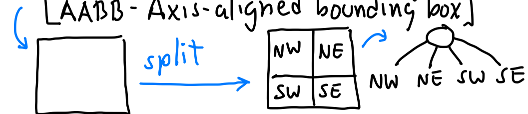


```
class Point {
    float[] coord               // coords
    Point(int d)
        ....> coord = new float[d]
    int getDim() --> coord.length
    float get(int i) --> coord[i]
    .... others: equality, distance
             toString ...
}
```

Quadtrees & kd-Trees II

Quadtrees: (abstractly)

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

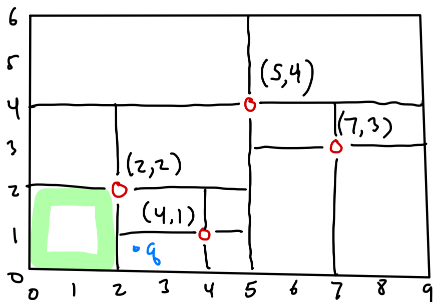


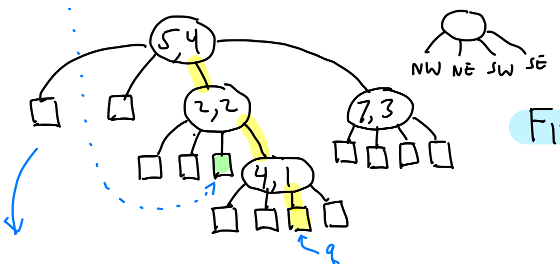
- Splitter: Subdivides cell into four (genlly 2^d) subcells

Point Quadtree:

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

(5,4)
(2,2)
(7,3)
(4,1)





Each external node corresps. to cell of final subdivision

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)

Quadtrees - Analysis

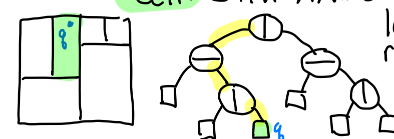
- Numerous variants!
  PR, PMR, QR, QX, ... see Samet's book
- Popular in 2-d apps
  (in 3-d, octtrees)
- Don't scale to high dim
  - out degree = 2^d
- What to do for higher dims?

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

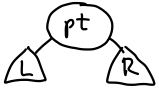
- splitter: Horiz. or vertic. line in 2-d (orthogonal plane o.w.)
- cell: Still AABB



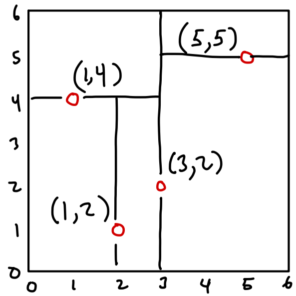
left: left/below
right: right/above

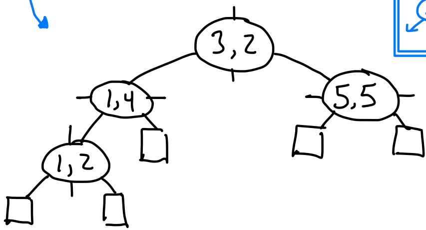
Quadtrees & kd-Trees III

Example:



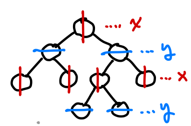
(3,2)
(1,4)
(5,5)
(1,2)



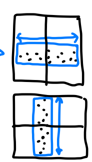


How do we choose cutting dim?

- **Standard kd-tree**: cycle through them (eg. d=3: 1,2,3,1,2,3...) based on tree depth

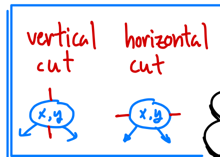


- **Optimized kd-tree**: (Bentley)
  - Based on widest dimension of pts in cell.



Kd-Tree Node:

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



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

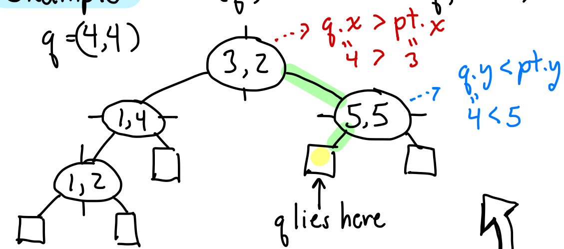
- if q == p.point ⇒ found!
- if q[cd] < p.point[cd] ⇒ left
- if q[cd] ≥ p.point[cd] ⇒ right

Helper:

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

Example: find(q) $\xrightarrow{\text{calls}}$ find(q, root)

q = (4,4)



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)          ···> all coords match?
        return p.value
    else if (p.onLeft(q))
        return find(q, p.left)
    else
        return find(q, p.right)
}
```

Quadtrees & kd-Trees IV

Kd-Tree Insertion:

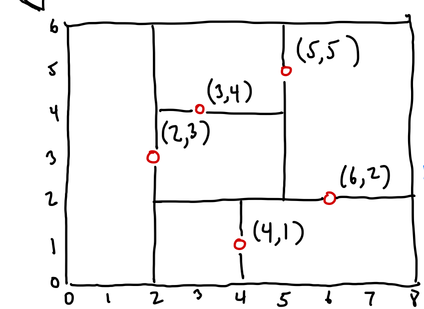
(Similar to std. BSTs)

- Descend tree until
  - → find pt → Error - duplicate
  - → falling out ← (Although we draw extended trees, lets assume standard trees)
    - → create new node
    - → set cutting dim

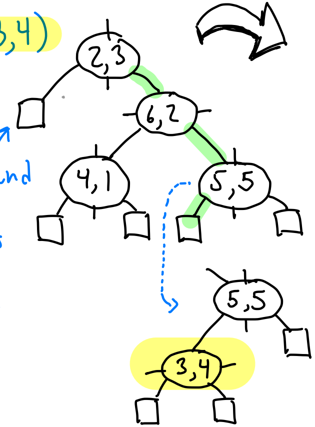
```
KDNode insert(Point pt,
              KDNode p, int cd){        // cutting dimension to use
  if(p == null)  //fell out?
    p = new KDNode(pt, cd)
            // new leaf node
  else if(p.point == pt)
    Error! Duplicate key
  else if(p.onLeft(pt))
    p.left = insert(pt, p.left, (cd+1) % dim)   // dimension of point
  else
    p.right = insert(pt, p.right,
                     (cd+1) % dim)
  return p
}
```

Example:

insert(3,4)



pretend ext. nodes are null

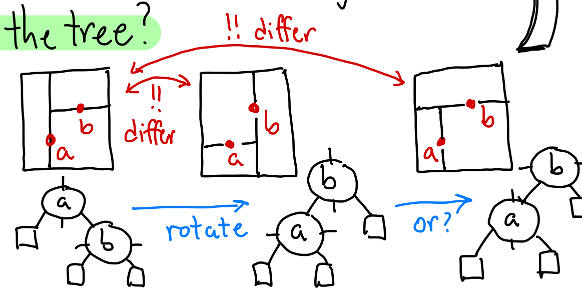


Analysis:

Run time: $O(h)$ (Tree height)

Can we balance the tree?

- Rotation does not make sense !!



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.