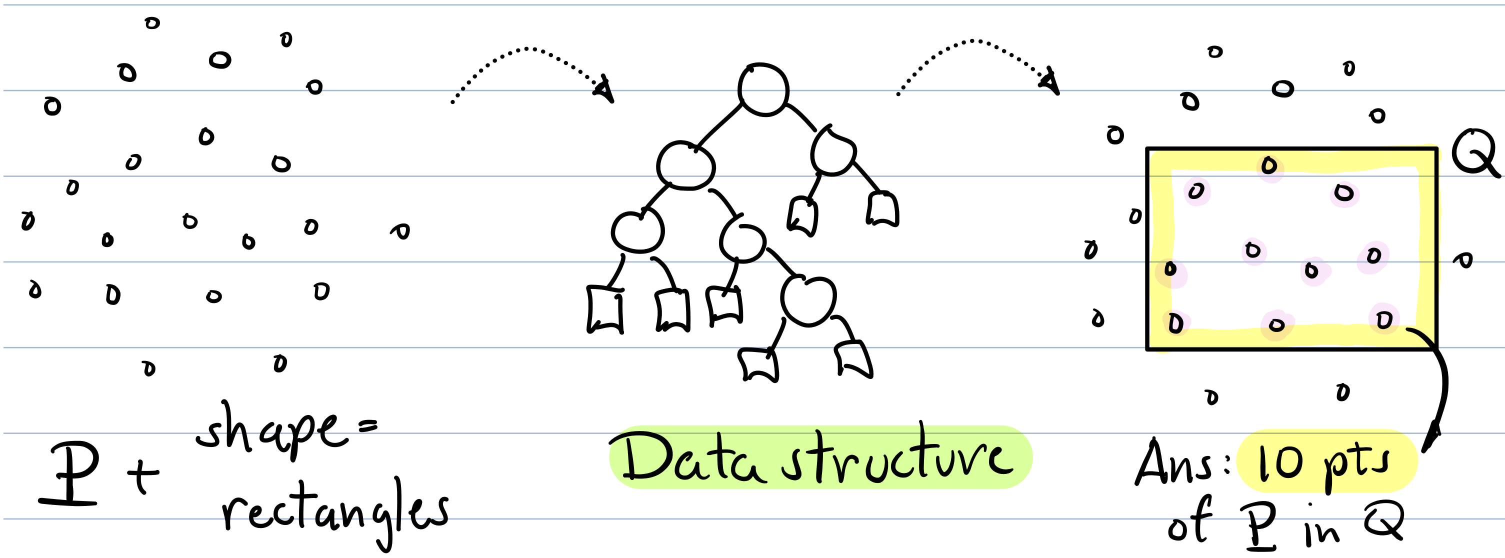
CMSC 754 - Computational Geometry
Lecture 14 - Orthogonal Range Search + kd-Trees

Range Searching: (Data structure problem)

- Given a point set $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^d$
- Given a class of shapes
  (e.g. rectangles, balls, triangles, halfspaces)
- Build a data structure so that:
  - Given any query region Q from the class, quickly identify the points of P in Q



What types of Queries?

- **Emptiness**: Any pts of P in Q?
- **Counting**: How many? $|P \cap Q|$
- **Weighted count**: Each $p \in P$ has weight $w(p)$. Return total weight

$$\sum_{p \in P \cap Q} w(p)$$

- **Semigroup weight**: Any commutative + associative function of wts:

Eg. max-query: $\max_{p \in P \cap Q} w(p)$

- **Reporting**: List the pts of $P \cap Q$
- **Top-k**: List just the highest k pts of $P \cap Q$ based on weights

Complexity Bounds:

Space: Total space needed to store points + data structure

Query time: Time needed to answer a query

Construction time: Time to build structure

Common: (Space bound) $\cdot O(\log n)$

"**Gold standard**": $O(n)$ space
$O(\log n)$ query time
$O(n \log n)$ constr. time

Many geometric structures are inferior w.r.t. space: $O(n \log^2 n)$
$O(n \log^d n)$ in $\mathbb{R}^d$
$O(n^2)$

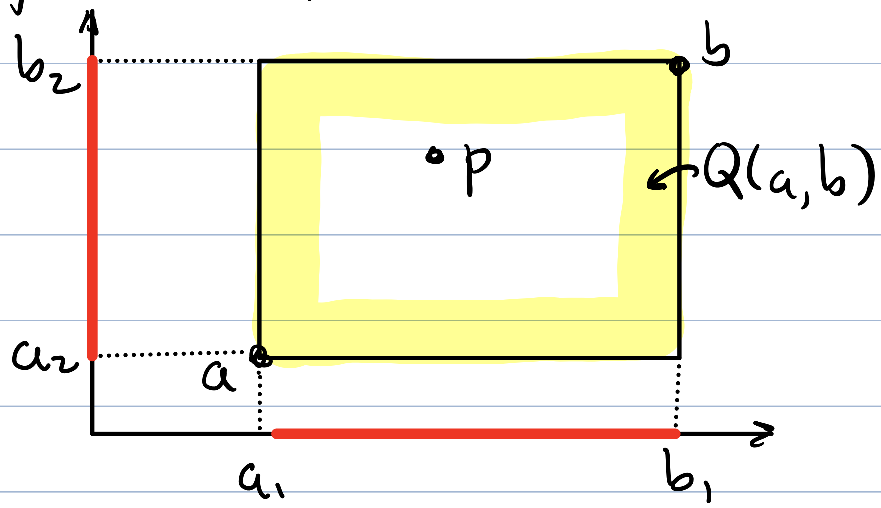
or Query time:

$O(\log^2 n)$
$O(\sqrt{n})$
$O(n^{1-1/d})$ in $\mathbb{R}^d$

Orthogonal Range Queries:

Query region is axis-aligned rectangle

Eg. Given pts $a, b \in \mathbb{R}^d$ s.t. $a_i < b_i \ \forall i$



Query rectangle is product of intervals:

$$Q(a,b) = \{ p \in \mathbb{R}^d \mid a_i \le p_i \le b_i \}$$

$$= [a_1, b_1] \times \dots \times [a_d, b_d]$$

Common in database queries:

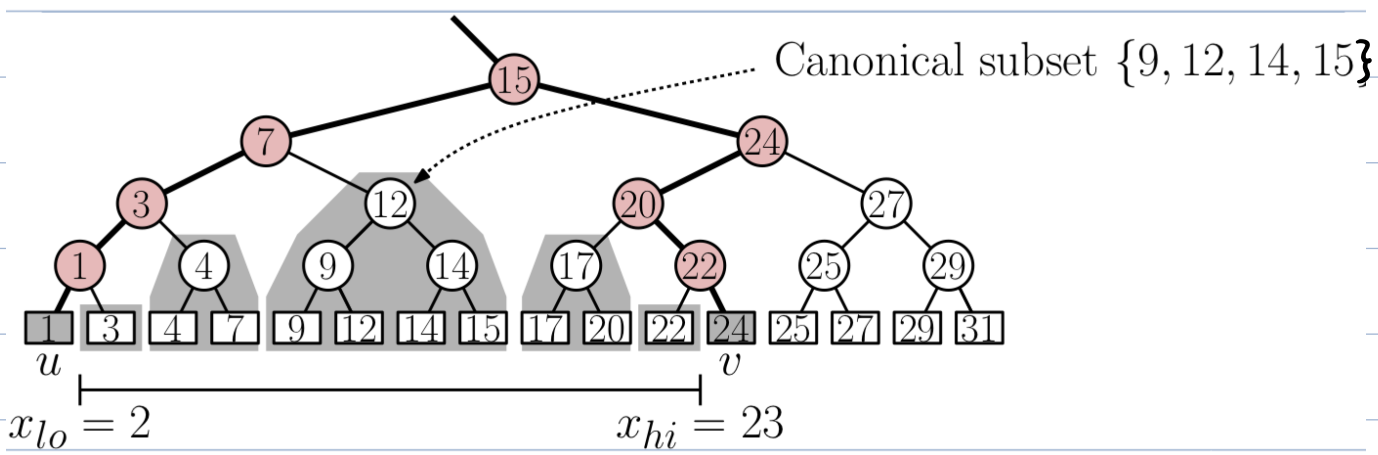
How many patients with age $\in [25, 35]$
weight $\in [100, 200]$
blood pressure $\in [80, 120]$

General approach to answering range queries:

- Too slow to count pts one by one
- Too much space to precompute answer to every possible query

- Canonical subsets:
  Carefully select an (ideally small) collection of subsets of P so that the answer to any query can be formed as (disjoint) union of a small number of subsets.

Example: 1-dimensional range query
$P = p_1 < p_2 < \dots < p_n$ in $\mathbb{R}$
- Store P as leaves of a balanced tree
- Leaves of each subtree form canonical set



- The answer to any 1-dim range query can be expressed as the disjoint union of $O(\log n)$ canonical subsets.

- Example: $Q = [x_{lo}, x_{hi}] = [2, 23]$

  $P \cap Q = \{3\} \cup \{4, 7\} \cup \{9, 12, 14, 15\} \cup \{17, 20\} \cup \{22\}$

  - Cover the range with maximal subtrees
  - Take union of the assoc. canonical subsets
  - $O(\log n)$ subtrees always suffice.
  - $O(n)$ nodes $\Rightarrow$ $O(n)$ canon. subsets

Compose the Answer to Query from Subsets:

Counting query: Node stores # of leaves

Weighted count: Node stores total weight of leaves

Max query: Node stores max of all weights in leaves

...

Can answer queries in $O(\log n)$ time by combining subtree results (assuming you can identify the canon. subsets for query + precompute info.)

Kd-Trees: A natural generalization of 1-d trees to higher dim.

1-d tree, 2-d tree, …, k-d tree

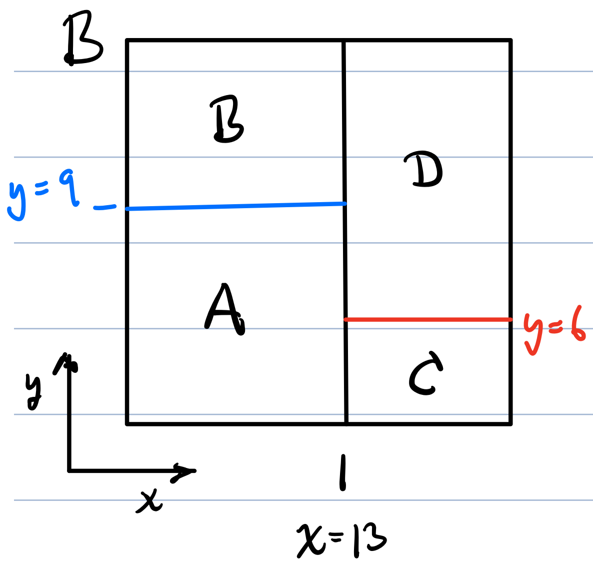
Jon Bentley (1975)

Numerous variants – we present one

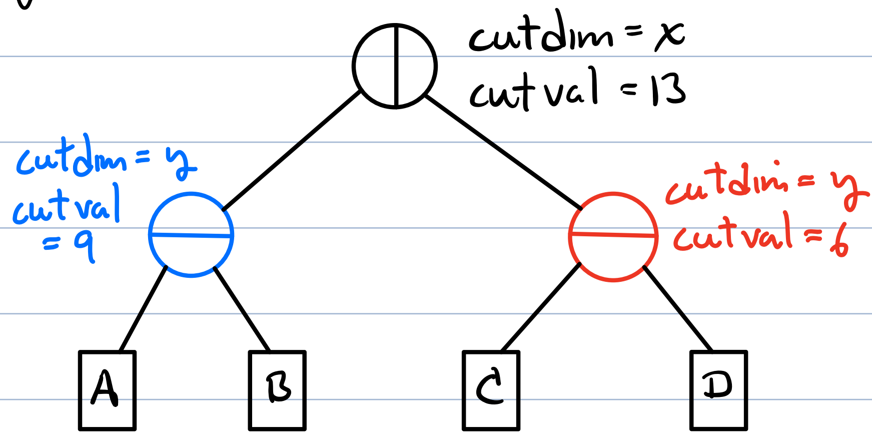
- Assume have large **bounding box** B containing P
- **Recursively split space** by axis-orthogonal hyperplane

  cutting dimension: which axis

  cutting value: where to cut



Spatial subdivision

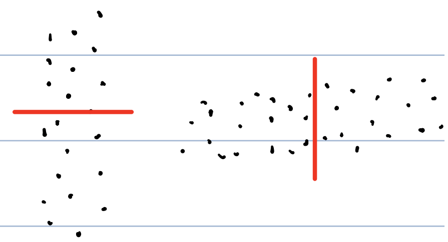


Tree structure

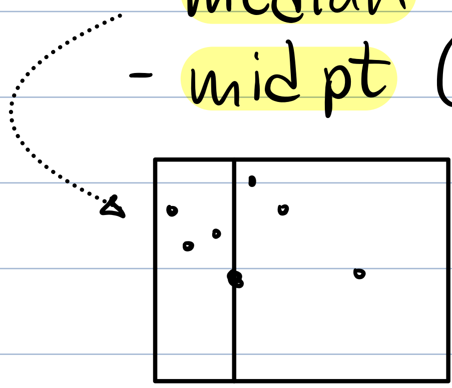
Cell: Each tree node represents a rectangular region

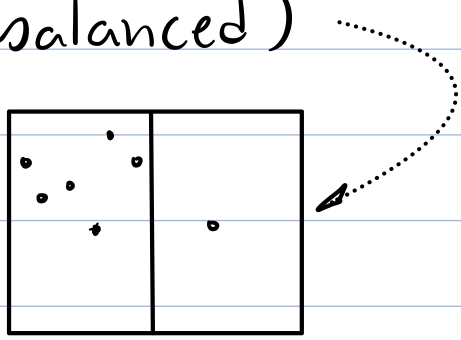
Design choices:

- Where are points stored?
  - internal nodes (used for splitting)
  - external nodes (leaves)
    - ↳ Permits more flexibility in where to split
- How is cutting dim chosen?
  - alternate: x, y, x, y, … or x, y, z, x, y, z, …
  - select based on point distribution



- How is cutting value chosen?
  - median (balanced height)
  - mid pt (geom. balanced)





Our structure:

- Points stored at leaves (external nodes)
- Alternate splitting axes
- Split at median

Construction:

Tree can be built in $O(n \log n)$ time

$$T(n) = n + 2T(n/2)$$

find median splitting coord

recursively build subtrees

$$= O(n \log n)$$

Slight improvement: Presort the points d times into d lists - one for each coordinate + cross-link entries

- Faster in practice

Space: $O(n)$

- n leaves (one per point)
- (n-1) internal nodes
- $O(1)$ info per node

Range Search:

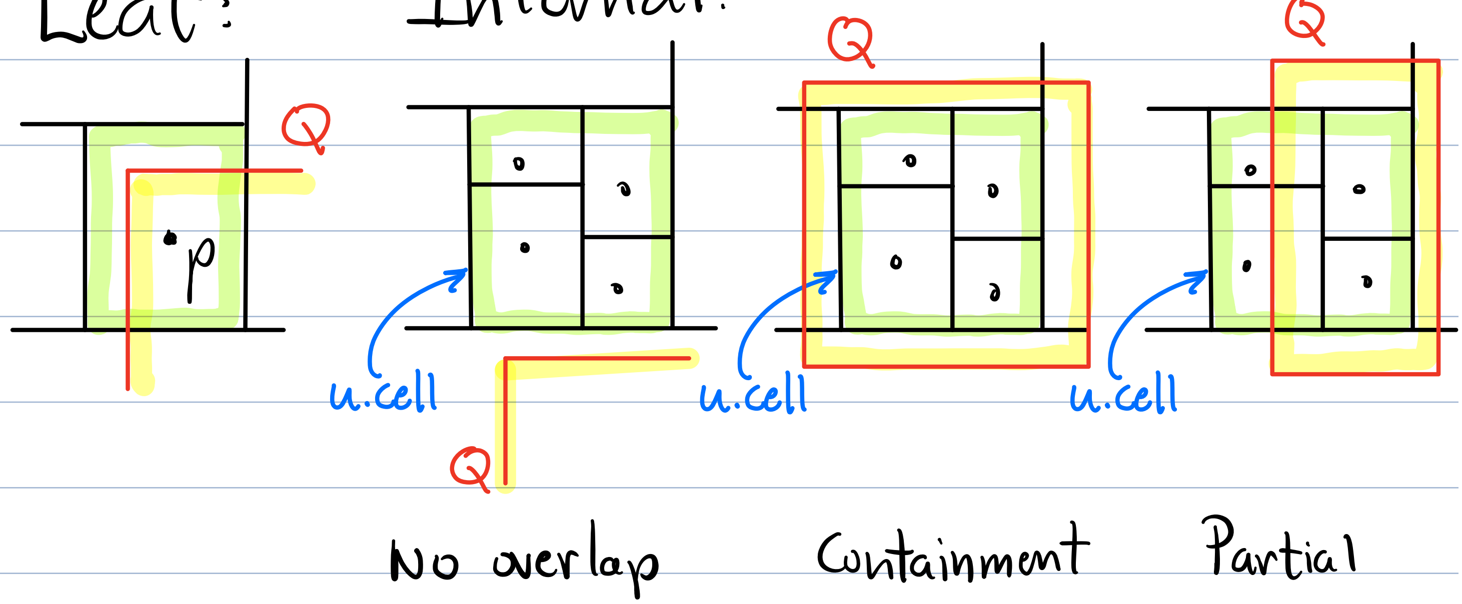
Key: If node's cell does not overlap Q → Don't visit

If node's cell completely in Q → count all its pts

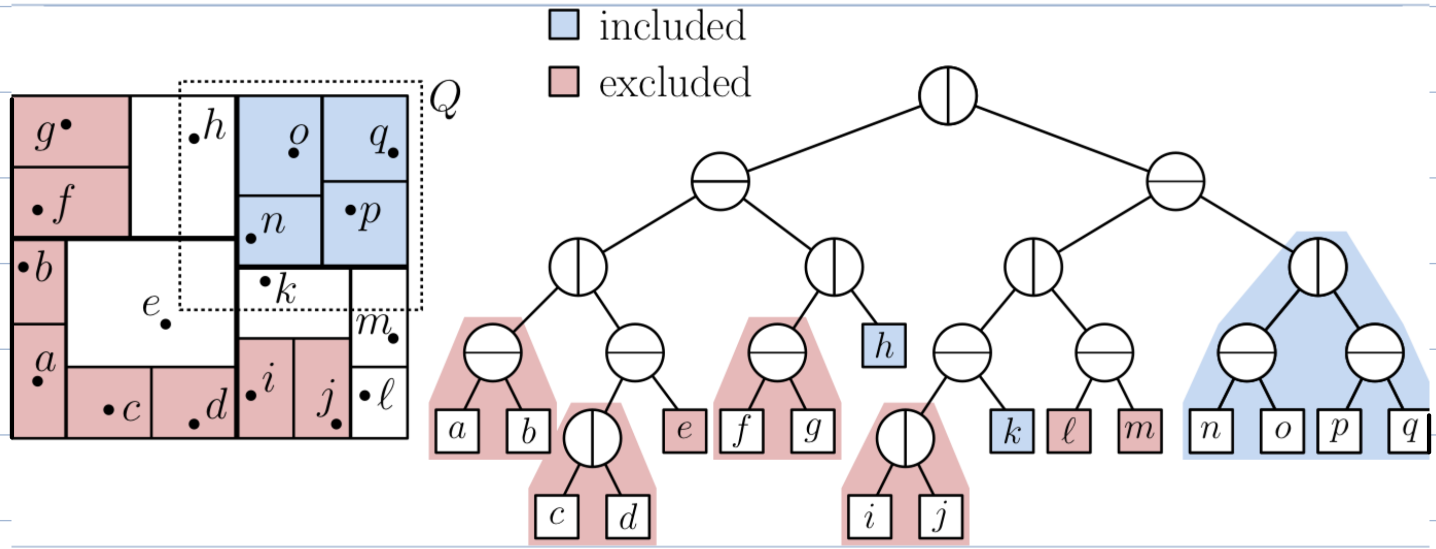
Algorithm: Weighted range count in kd-tree

```
range-count (Rect Q, KdNode u)
    if (u is leaf)
        if (u ∈ Q) return u.point.weight
        else return 0
    else (u is internal)
        if (u.cell ∩ Q = ∅)
            return 0     (no overlap)
        else if (u.cell ⊆ Q)
            return u.weight    (total weight)
        else
            return range-count (Q, u.left)
                   + range-count (Q, u.right)
```

Leaf: Internal:



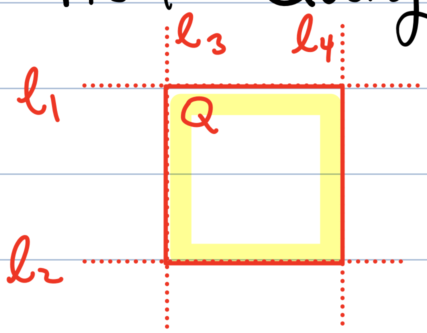
Example:



Query Time:

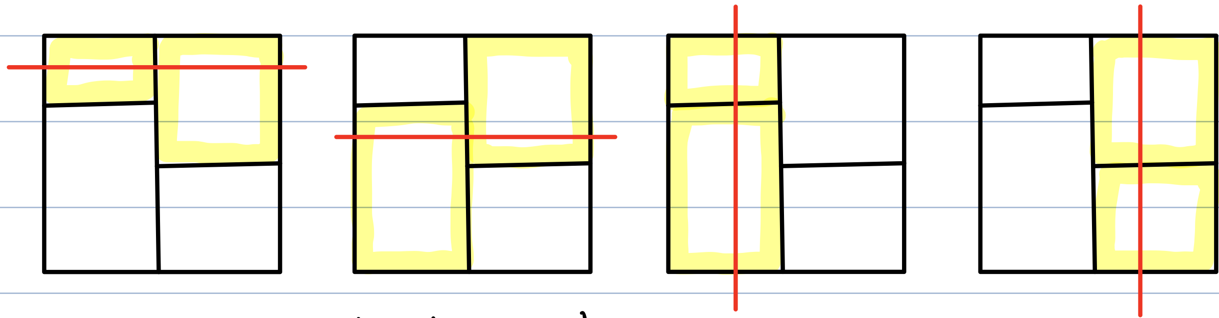
Thm: Given a height-balanced kd-tree in $\mathbb{R}^2$ using alternating splitting axes, orthog. counting queries can be answered in $O(\sqrt{n})$ time.
[Reporting queries in time $O(k + \log n)$, where
k = # of points reported]

Proof: Query rectangle bounded by 4 lines



We'll show that each line stabs $\le \sqrt{n}$ cells of tree $\Rightarrow O(4 \cdot \sqrt{n})$

Key: Because we alternate cutting dim for every 2 levels of tree, any axis parallel line can stab at most 2 out of 4 grandchild cells



Since we use balanced splitting

parent	n pts
child	$n/2$ pts
grandchild	$n/4$ pts

$\Rightarrow$ Query time:

$$T(n) = \underbrace{2T(n/4)}_{\text{recurse on 2 of 4 grandchildren}} + \underbrace{1}_{\text{constant time per cell}}$$

$$= O(\sqrt{n})$$ [see lect. notes for details]