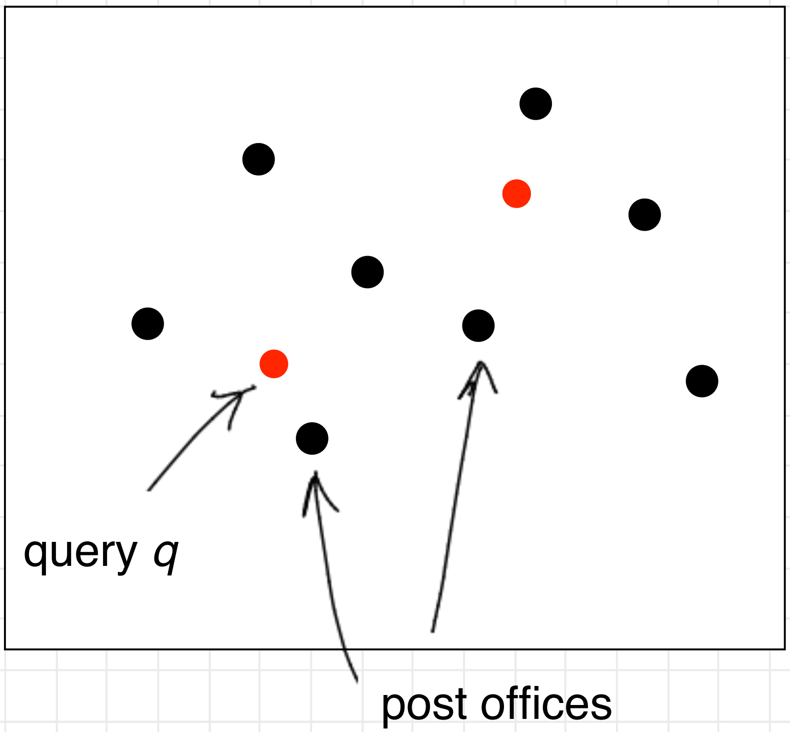
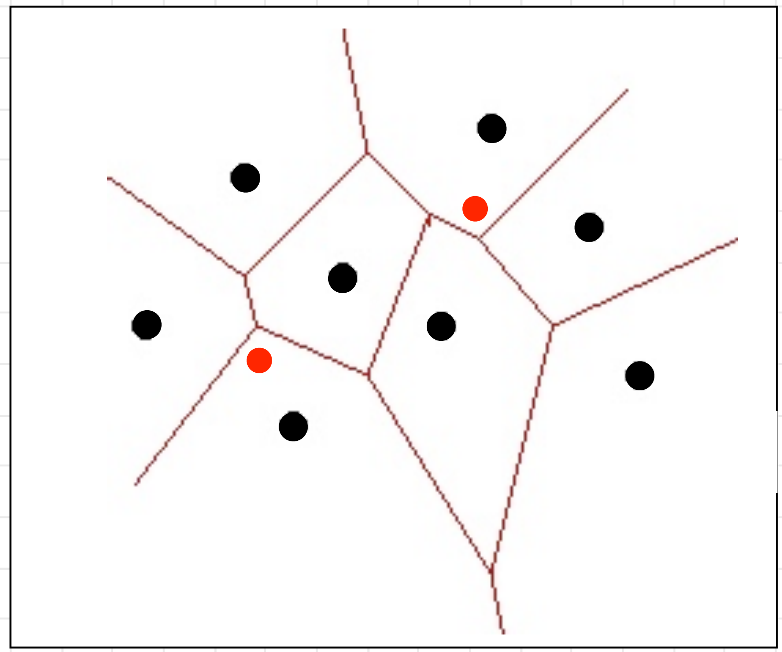
**Planar point location:** To answer, "Where am I?"
Given a planar subdivision (partition of the plane into disjoint regions by straight line segments), preprocess to quickly locate a query point.

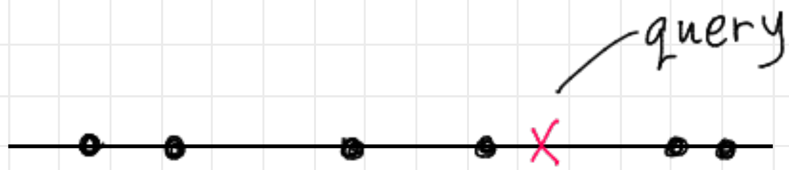
Example: the post office problem aka nearest neighbour problem.



Compute the Voronoi diagram.
Query becomes: which Voronoi region contains $q$?

Point location in 1D




use binary search in sorted array, or balanced binary search tree
(can handle dynamic case where points are added/deleted)

P = preprocessing time
S = space
Q = query time
(U = update time)

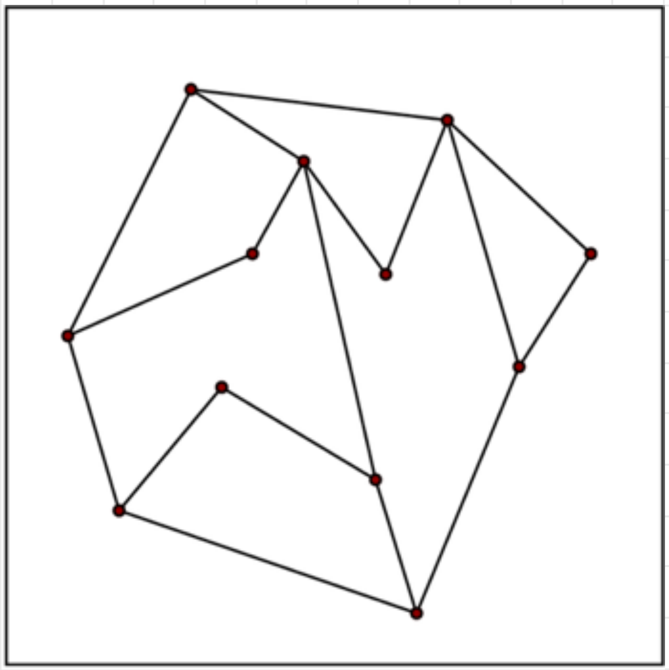
in 1D: P= $O(n \log n)$
S = $O(n)$
Q = $O(\log n)$

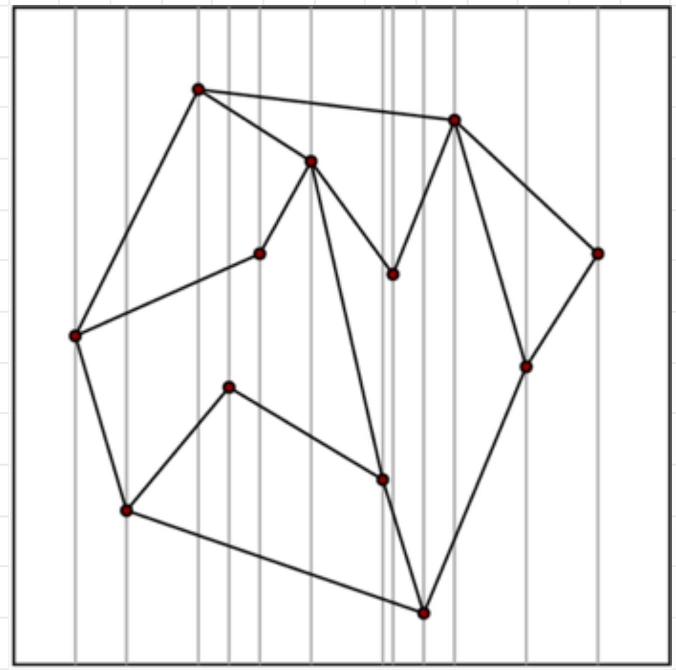
We can achieve the same bounds in 2D for **planar point location**.

1. slab method (not optimal)
2. persistence — I won't give details.
3. Kirkpatrick's triangulation refinement
4. trapezoidal map (expected good behaviour) — I won't give details.

**1. Slab method**: A basic solution to planar point location

Divide into vertical slabs at vertices.
Each slab is a 1D problem. — store each slab





P = $O(n^2)$ sort but then $O(n^2)$ output size

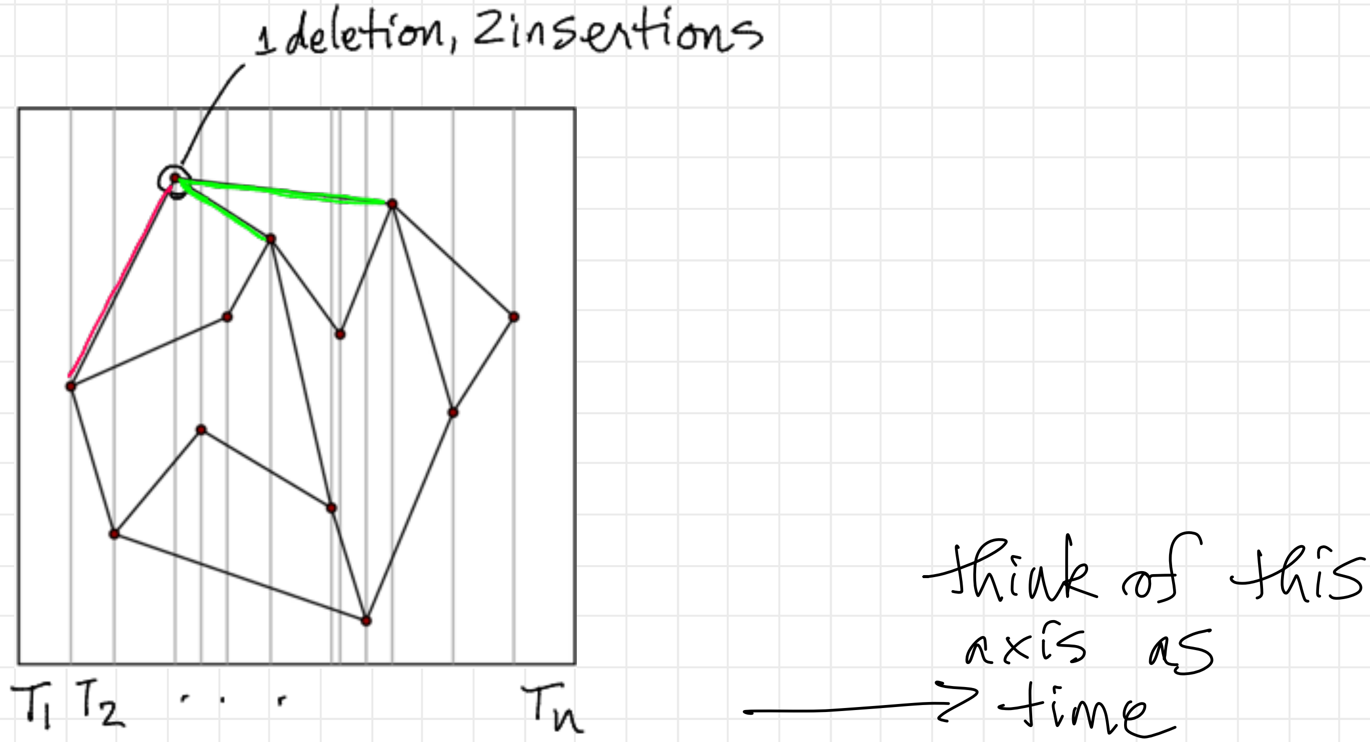
S = $O(n^2)$ — n slabs each of size $O(n)$

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

↑ find slab ↑ search in slab (1D search) with sidedness test.

**2. Persistence** [Tarjan and Sarnak, 1986]

Observe that the binary search trees for successive slabs do not change much.




We know how to update binary search trees at O(log *n* ) per insertion/deletion.

New issue: query may take place not in "current" tree but in any previous tree. $T_i$

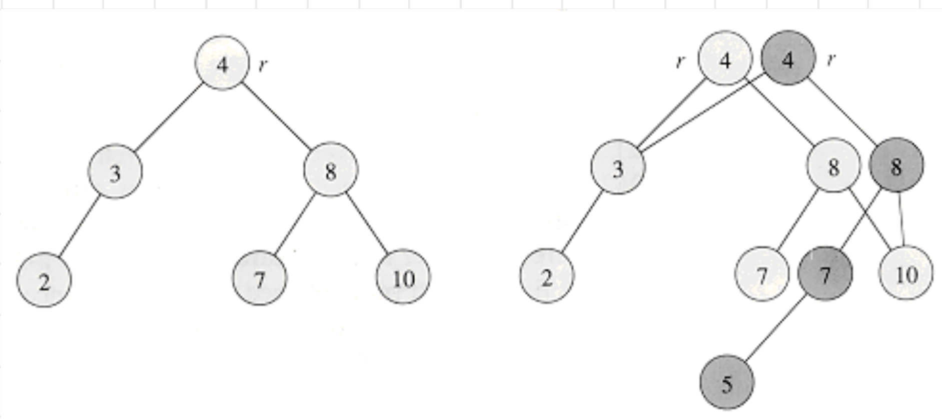
## Persistent data structure

Allow insertions and deletions over time (as in a usual dynamic data structure) BUT allow queries in old versions. The query specifies the time.

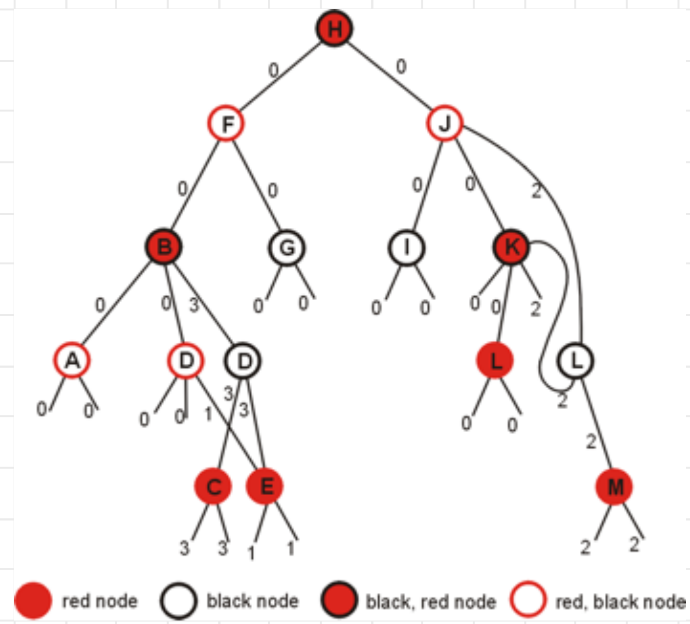
## Persistent search trees

**Idea 1**: make new tree share as much as possible with old tree

**Idea 2**: give each node one extra pointer to save making new copy



(a) (b)

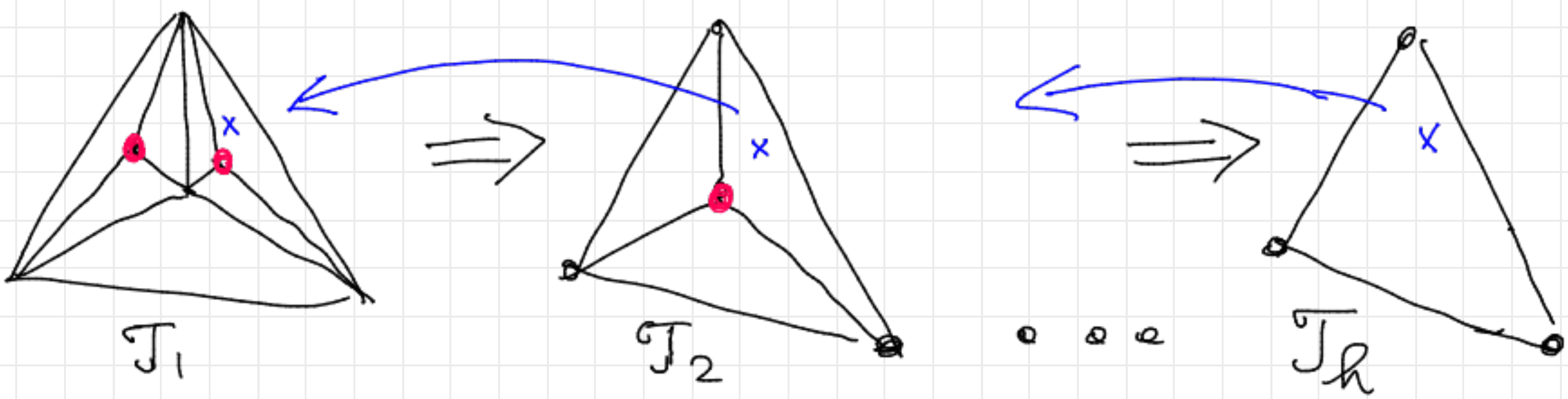



achieve: P= O($n$ log $n$ )
S = O($n$ )
Q = O(log $n$ )

**3. Kirkpatrick's triangulation refinement**, 1983

First triangulate the planar subdivision in O($n$ log $n$ ) time.
Also add a big bounding triangle.

**Idea**: make rougher and rougher versions by deleting vertices, until we have only the bounding triangle. Then search for query point starting backwards.



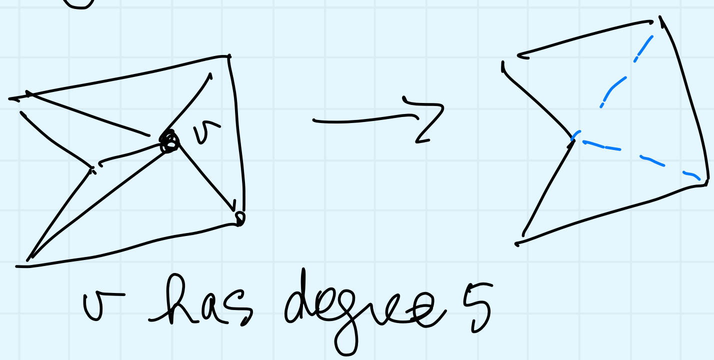

To make this efficient:

- want $h$ small. $h$ = # triangulations = length of search sequence
- want each query step $T_i$ to $T_{i-1}$ to be efficient.
  each triangle $T_i$ to intersect few triangles in $T_{i-1}$

Plan: At each stage remove some vertices

1. remove $\frac{n}{c}$ vertices, $c$ constant
   then $h$ (sequence length) is $O(\log n)$

2. only remove vertices of degree $\le d$, $d$ constant




add new edges to triangulate
- get $d-2$ triangles

each new triangle intersects at most $d$ old triangles

3. remove <u>independent</u> set of vertices
   (no two joined by an edge)
   So the regions to be re-triangulated are disjoint.

Assuming the plan is possible, here's the analysis of *h*, S and Q

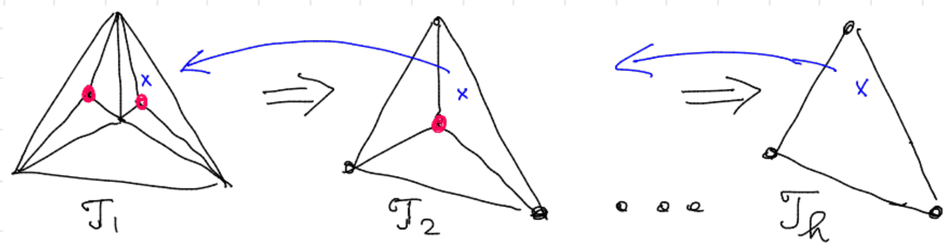
# vertices in each triangulation:

$$n \quad n(1-\frac{1}{c}) \quad n(1-\frac{1}{c})^2 \quad \ldots$$

Thus *h* = O(log *n* ) triangulations in total

Total size of all triangulations: $O(n(\sum_{i=0}(1-\frac{1}{c})^i)) = O(n)$ Thus S = O(*n* )

Time per query:




$$O(\log n) \times O(1)$$

# triangulations

to go from triangulation $\boldsymbol{T}_i$ to $\boldsymbol{T}_{i-1}$

Keep pointers from each triangle in $\boldsymbol{T}_i$
to all *d* intersecting triangles in $\boldsymbol{T}_{i-1}$
and check which one contains the query point

Thus Q = O(log *n* )

**Lemma.** There exist constants *c*, *d*, such that for any triangulation ***T*** (= $\mathcal{T}$) on *n* vertices, we can find, in O(*n*) time, a set of ≥ *n* /*c* vertices each of degree ≤ *d* that form an independent set.

**Proof.** $\mathcal{T}$ has ≤ 3n-6 edges (Euler)

So average degree is $\frac{2(3n-6)}{n} < 6$

smallest degree is ≥ 2 (3 if no collinearities)

Thus $< \frac{n}{2}$ vertices have degree ≥ 10

Let Z = vertices of degree ≤ 9 $|Z| \geq \frac{n}{2}$

Use greedy algorithm to pick independent vertices $Z' \subseteq Z$

pick $v \in Z$, delete $v$ and neighbours (≤ 9 neighbours)

repeat

$|Z'| \geq \frac{|Z|}{10} \geq \frac{n}{20}$

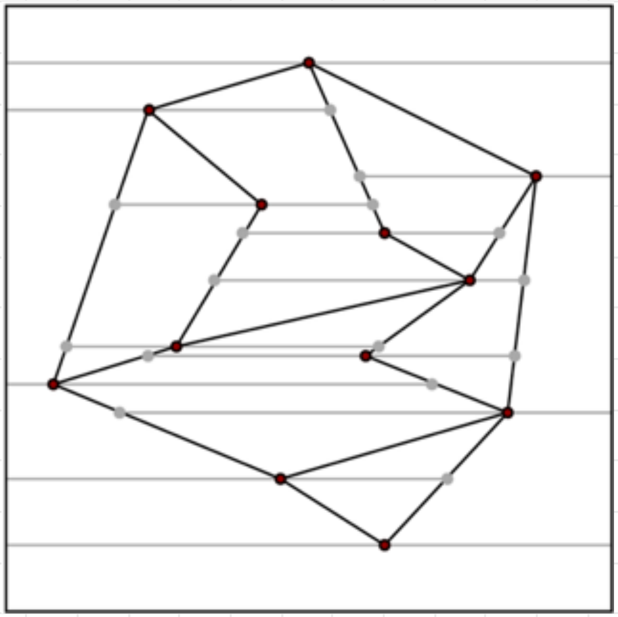
Get c = 20 d = 9.

Time is O(n) ← # vertices in $\mathcal{T}$

Total time: O(n) PE O(n log n)

**4. Trapezoidal decomposition** (good expected case behaviour)

Recall we saw trapezoidization of a polygon. Same idea for planar subdivision.



extend a horizontal line left and right of each point until we hit an edge

size is $O(n)$

Note: if we can locate the trapezoid containing a point, this gives the region containing the point.

Randomized incremental algorithm to build trapezoidal decomposition (add segments one by one in random order) AND point location data structure.

Note: To build the trapezoidal decomposition we use the point location structure.

Can achieve expected bounds $P = O(n \log n)$
$S = O(n)$
$Q = O(\log n)$

skipping details.

**Summary on planar point location**

P= O($n$ log $n$)
S = O($n$)
Q = O(log $n$)

- persistence
- Kirkpatrick's triangulation refinement
- trapezoidal map (expected case behaviour)

There are other methods.
Also, the constant inside the O(log $n$) query time can be made 1.

Seidel, Raimund, and Udo Adamy. "On the exact worst case query complexity of planar point location." Journal of Algorithms 37.1 (2000): 189-217.
http://www.sciencedirect.com/science/article/pii/S0196677400911015

**Dynamic planar point location.** Support updates to the planar subdivision. In 1D, balanced binary search trees support updates in O(log $n$), but it's harder in 2D.

for possible projects, see [Handbook]

**OPEN.** Achieve the above P, S, Q for point location in 3D.

**Localization.** Problem from robotics/vision: Determine your coordinates from local (visible) geometry.

**Other geometric data structures problems**

Handbook of Discrete and Computational Geometry [Handbook]:

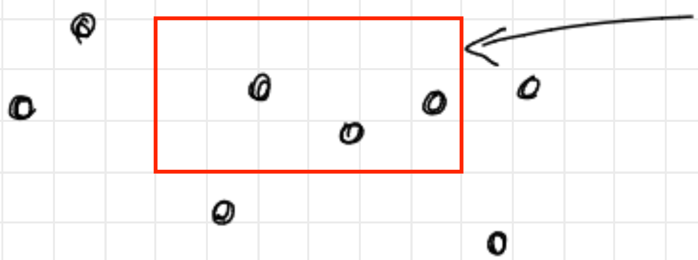


We will touch on range searching.
Huge amount of practical and of theoretical work.

## Range Searching

Given points in $R^d$ preprocess to quickly answer a query of the form: given a *range*, return the points in it.

**Orthogonal range searching**. A *range* is a rectangle.
E.g. in database query, find everyone between 30 and 40 years old making between \$50K and \$90K.

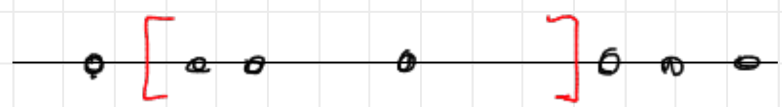


query rectangle

As before, we care about

- P = preprocessing time
- S = space
- Q = query time
- (U = update time)

In 1D
- P = O($n \log n$)
- S = O($n$)
- Q = O($\log n + t$), $t$ = output size
- U = O($\log n$)



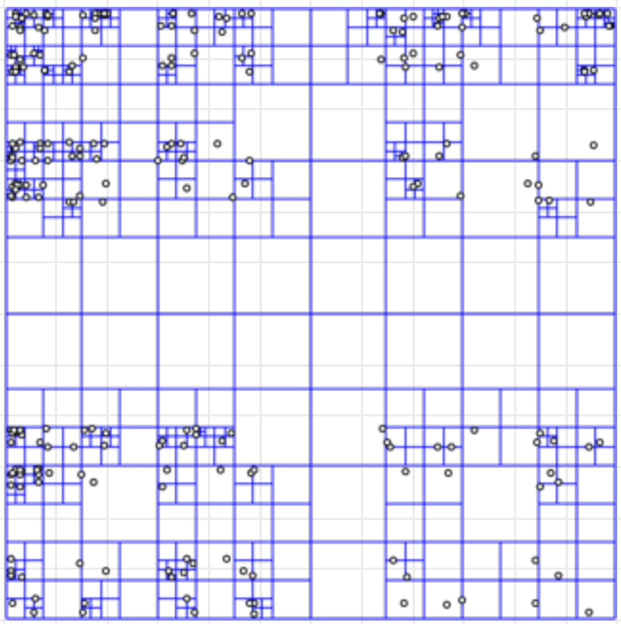
a rectangle is an interval

## Orthogonal range queries in 2D

Methods: quadtrees, kd trees, range trees

## Quad trees

divide squares into 4 subsquares. Repeat until each square has 0 or 1 points.



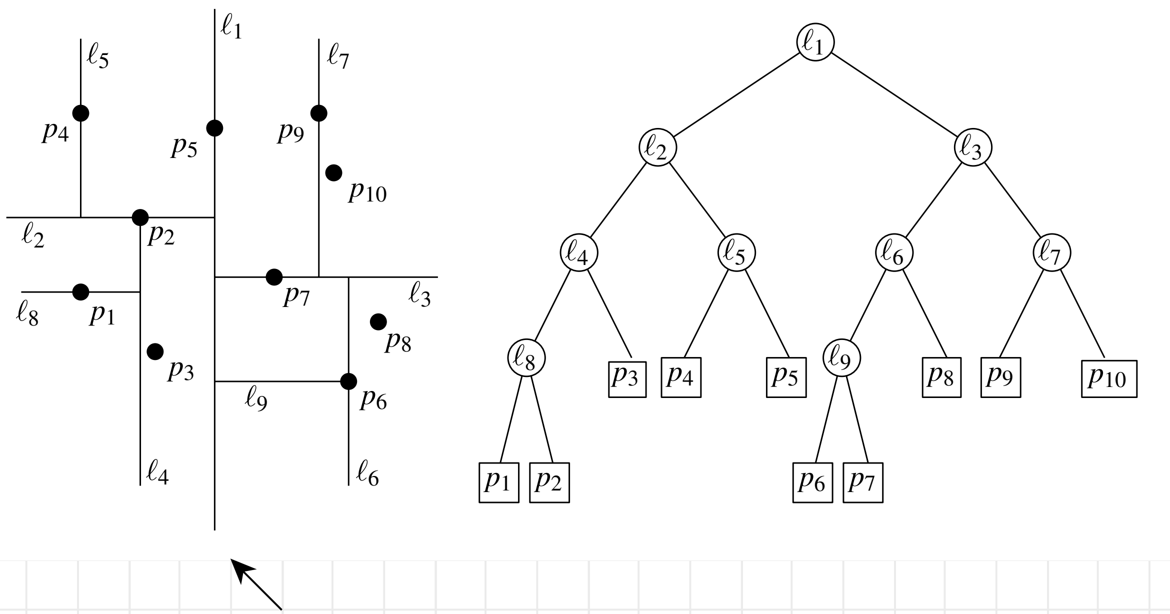
W https://en.wikipedia.org/wiki/Quadtree

P = O($n$ log $n$)
S = O($n$)
Q = O($\sqrt{n} + t$), $t$ = output size
U = O(log $n$)

**Orthogonal range queries in 2D**

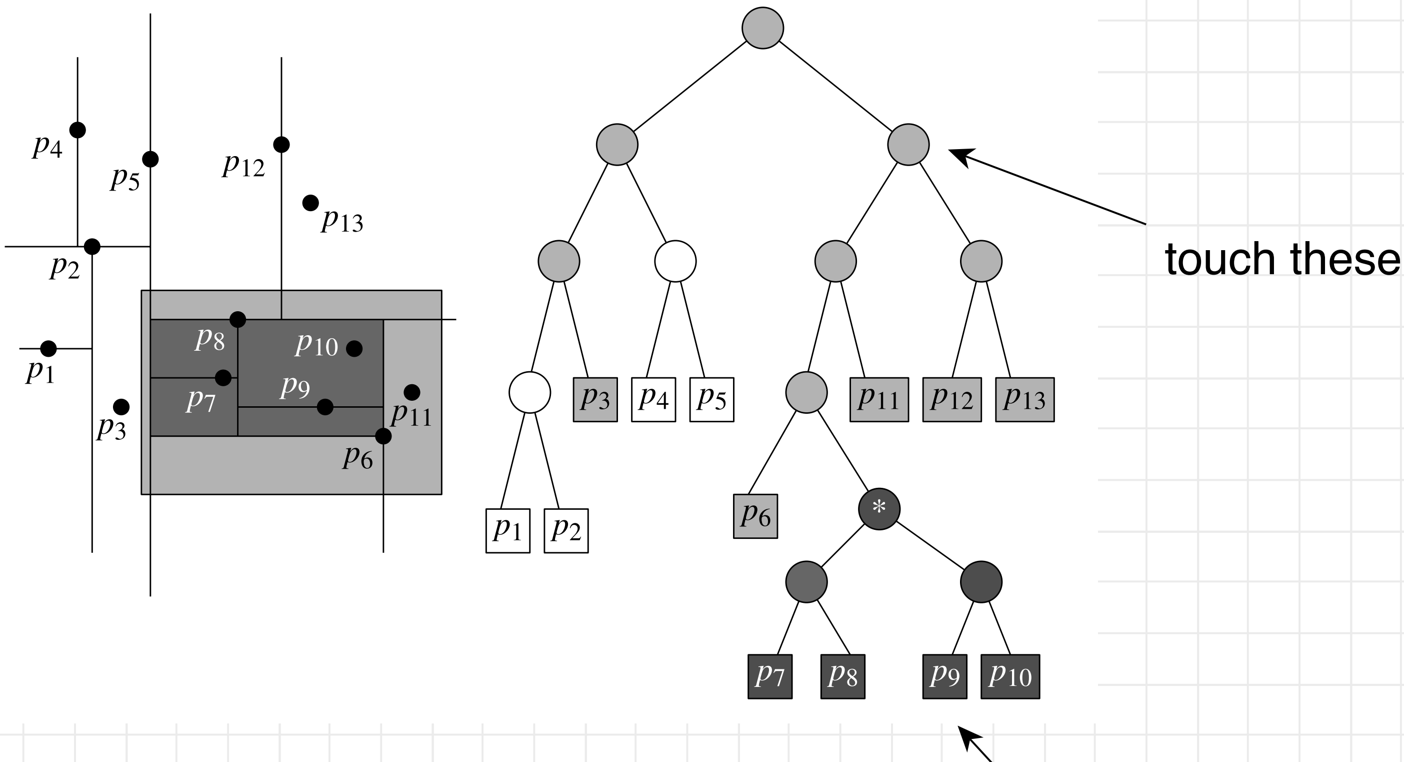
**kd tree**

alternately divide points in half vertically then horizontally




P = O($n$ log $n$ )
S = O($n$ )

querying a kd-tree



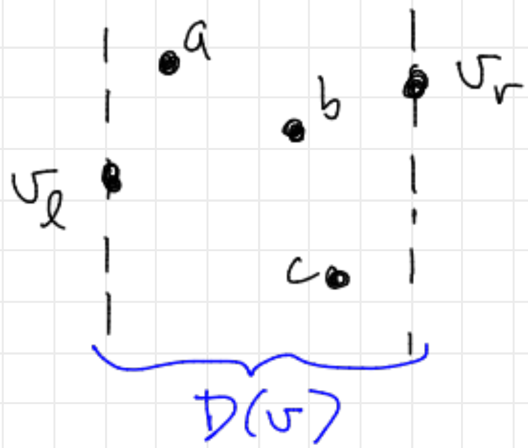

Can show Q = O($\sqrt{n}$ + $t$ ), $t$ = output size

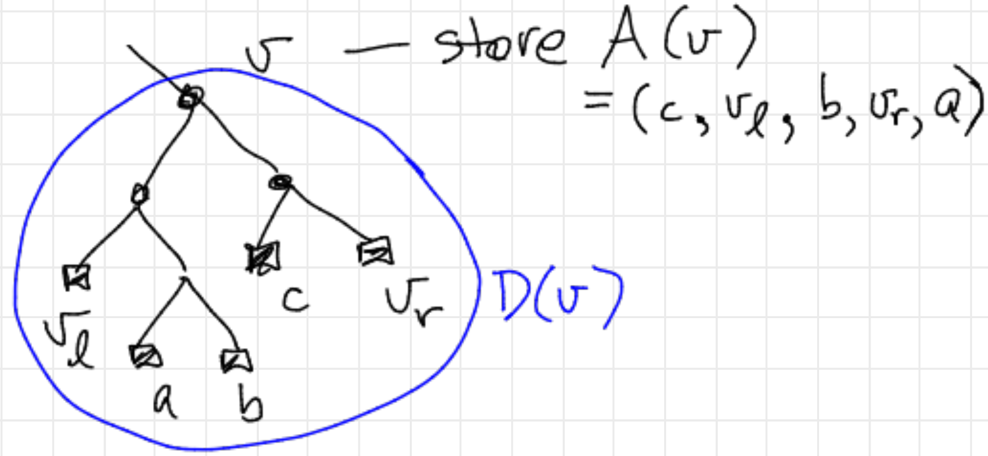
$$\sqrt{n} = 2^{(\log n)/2}$$

## Orthogonal range queries in 2D

**Range Tree.** Improve Q at the expense of S.
Make a balanced binary search tree. Leaves = points sorted by x-coordinate.

D($v$) = ***descendants*** of node $v$
associated with ***slab*** from $v_l$ to $v_r$

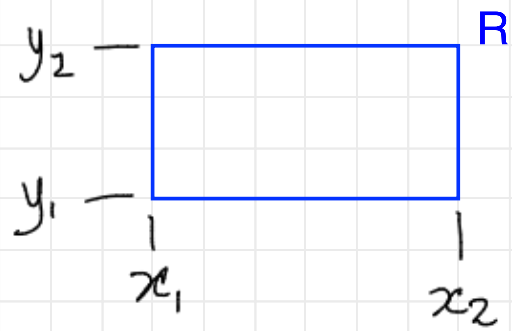


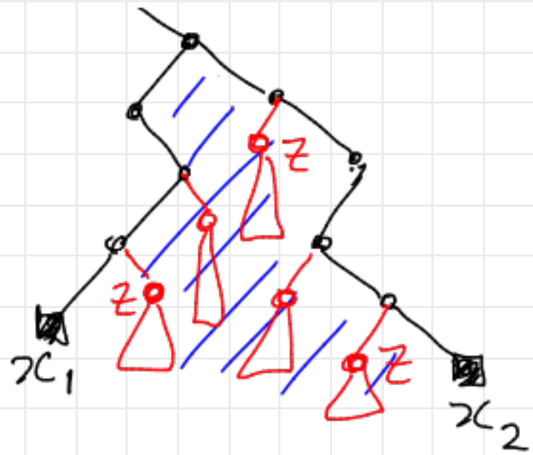




At node $v$, attach array A($v$) — points in D($v$) sorted by y-coordinate

S = O($n$ log $n$) — each point is in D($v$) for (log $n$) $v$'s
P = O($n$ log $n$) — sort by x to make the tree; sort by y to make the lists A($v$)

**Range Tree.** How to query rectangle R

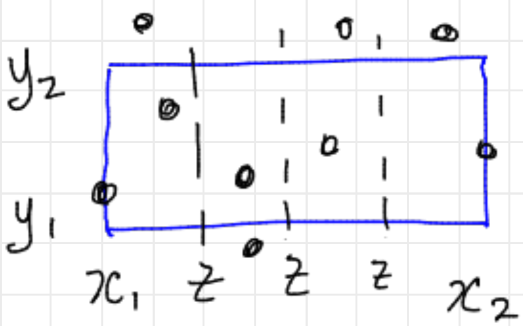



- search the tree for $x_1$ and $x_2$
- the points we want are at the leaves between $x_1$ and $x_2$, but we must filter to get between $y_1$ and $y_2$

Look at nodes $z$
- right children of nodes on search path to $x_1$
- left children of nodes on search path to $x_2$

There are O(log $n$ ) of them. They give disjoint slabs with union $[x_1, x_2]$.




- for each $z$ (each slab) do binary search in A($z$ ) to get points between $y_1$ and $y_2$

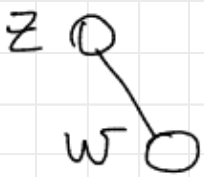
O(log $n$ + output) per slab. Since the slabs are disjoint, we don't repeat output, so total is $Q = O(\log^2 n + t)$, $t$ = output size.

**Range Tree.** Fractional cascading.

How to improve Q from $O(\log^2 n + t)$ to $O(\log n + t)$.

**Idea:** in each slab list A($z$), we repeat the binary search for the same $y_1$ and $y_2$. That's wasteful!

Consider node $z$, child $w$




sorted y coordinates of points

A($z$) = 1, 3, 4, 7, 11, 12

A(w) = 1, 4, 7, 11

- keep a pointer from each element in A($z$) to the corresponding element (or next higher) in A($w$)

This gives $Q = O(\log n + t)$
— we search once for $y_1$ and $y_2$ in A(root) and then follow pointers

Summary

- planar point location
- range searching

References

- [CGAA] Chapter 5
- [Handbook]

There are many possibilities for projects.