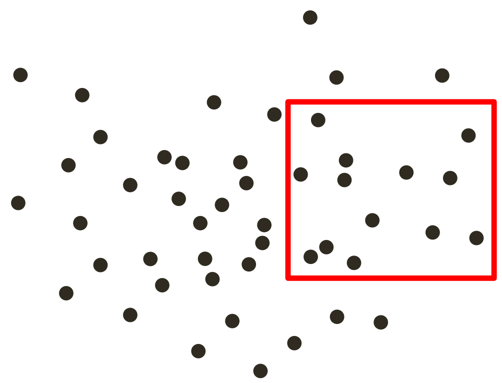
Geometric Searching

- Orthogonal Range Searching
- Point Location

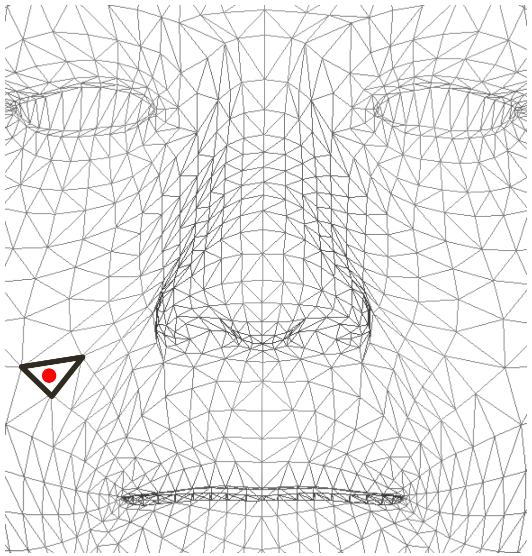
Types of Searching

Range searching



Report all the objects inside an axes-parallel query rectangle.

Point location



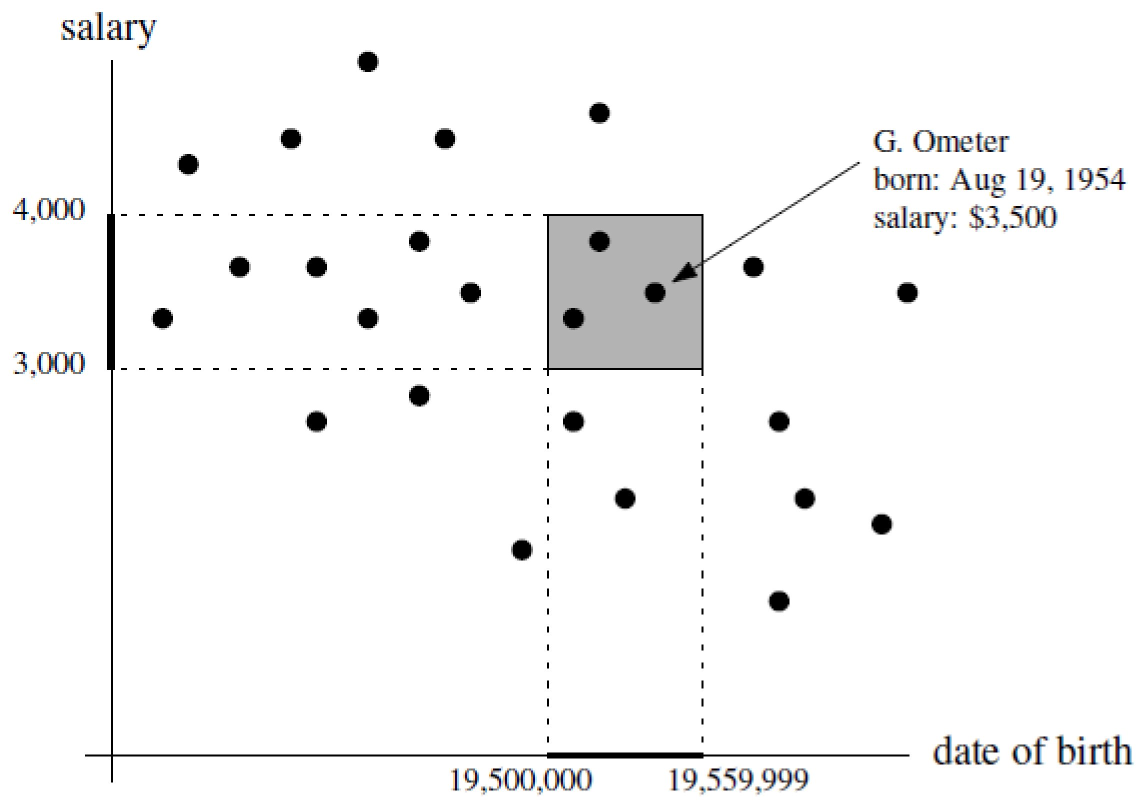
Report a face of a straight planar graph that contains the specified query point.

Range Searching

Query: $(X_{min}, Y_{min}, X_{max}, Y_{max})$

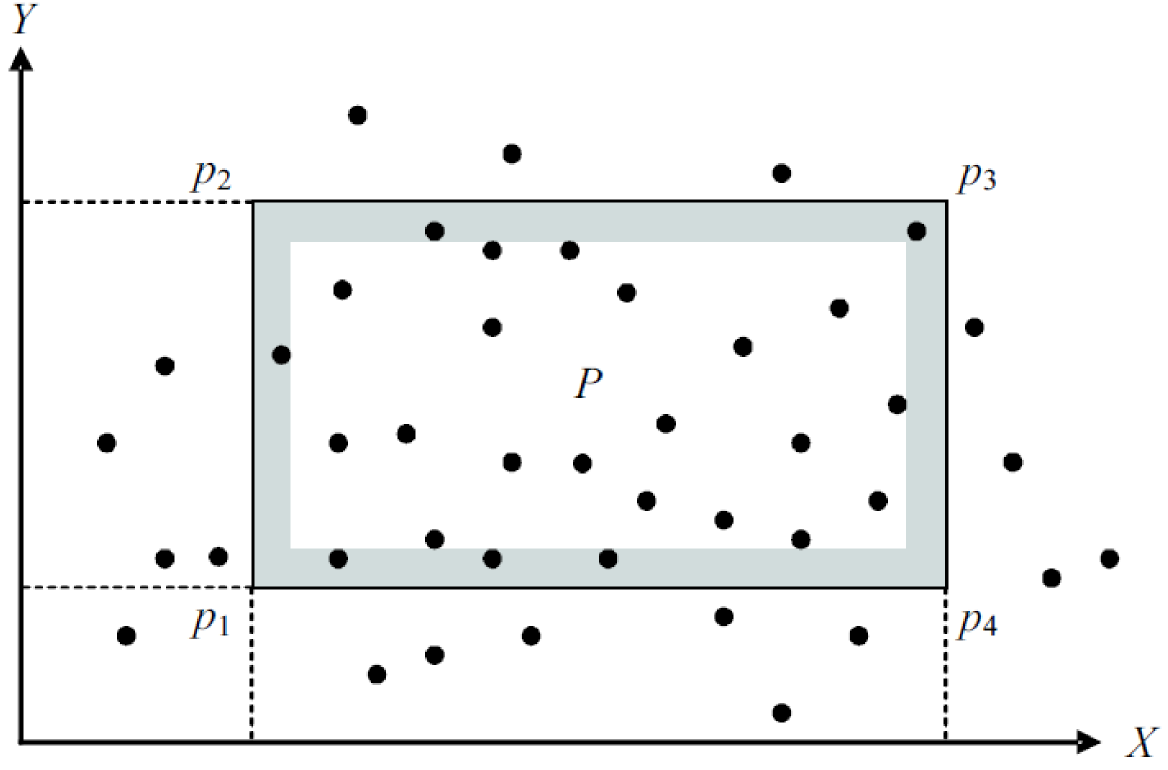
Multidimensional Query to a Database

Employee
name
date of birth
salary
sex



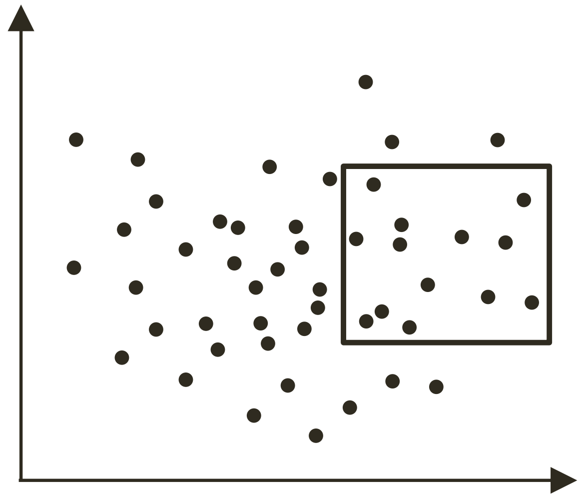
Points Counting

Given a set S of n points on the plane, report **how many** of them are inside the specified orthogonal query rectangle P?

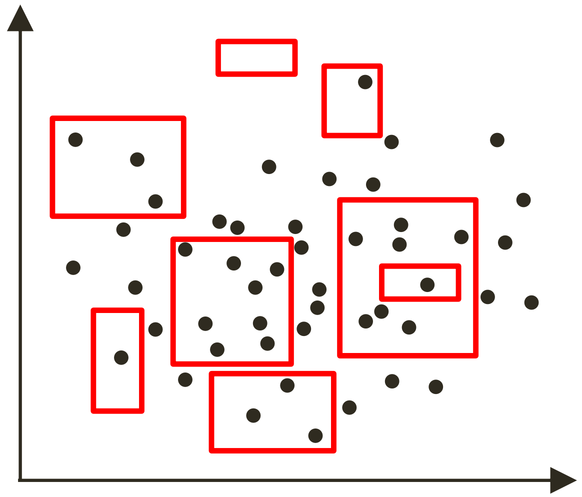


Query Types

Single



Multiple

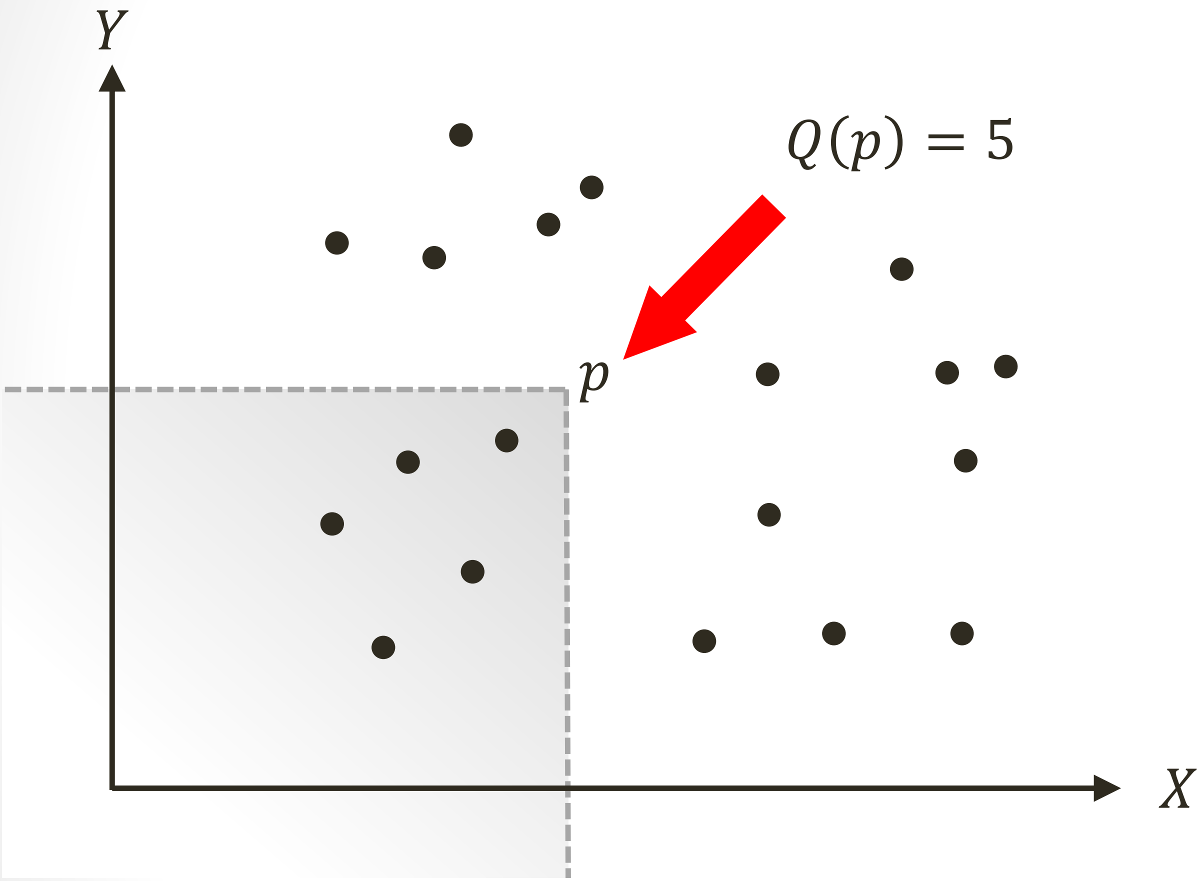


Metrics

- Query time
- Memory usage

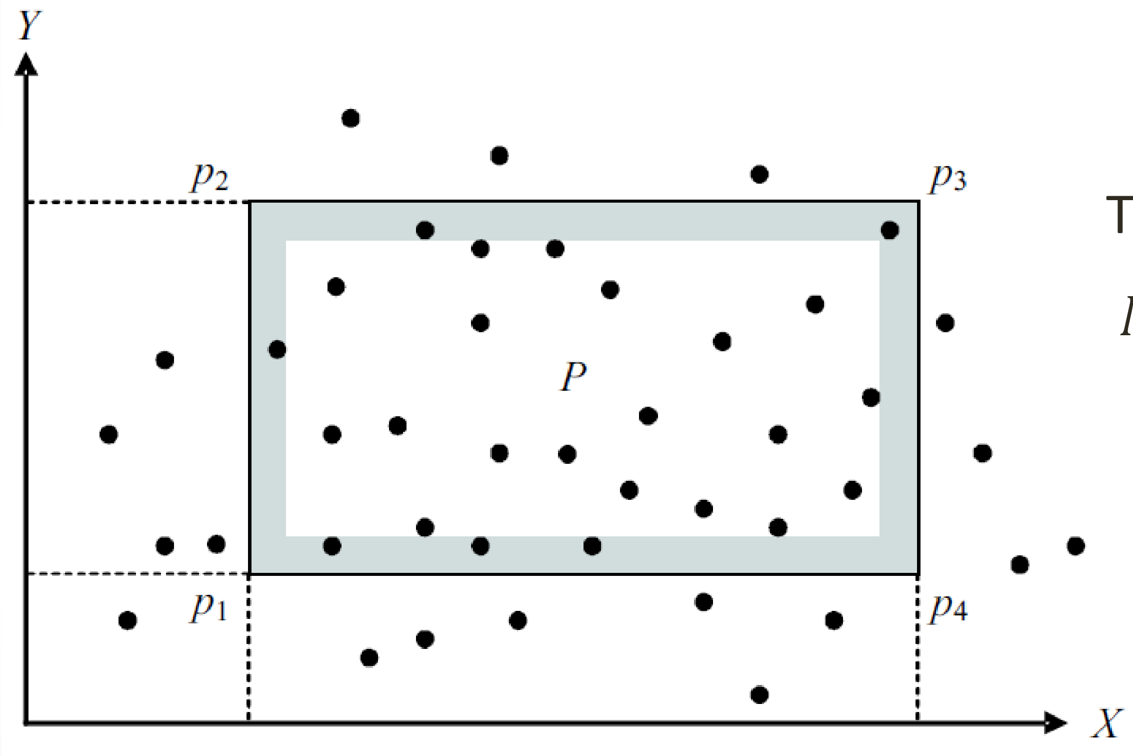
- Query time
- Memory usage
- Preprocessing time

Points Counting: Multiple Query



Domination number $Q(p)$ *of point* p: the number of points in S on the left and below of p.

Points Counting: Multiple Query

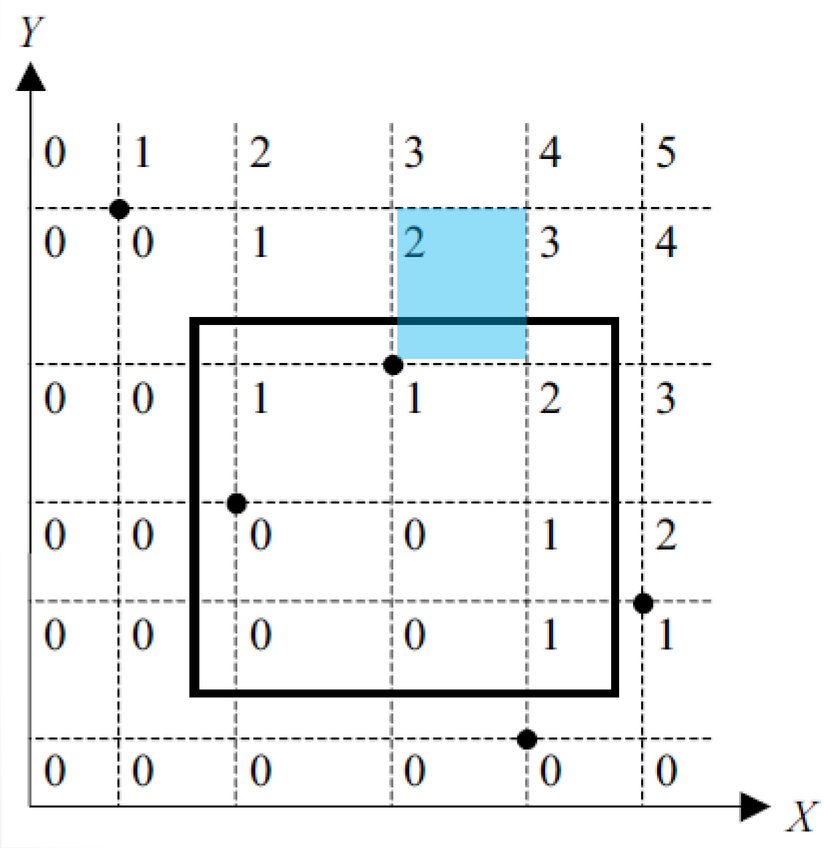


The number of points inside P:

$$N(P) = Q(p_3) - Q(p_2) - Q(p_4) + Q(p_1)$$

Locus Approach

Here, *locus* is a set of points with **the same domination number**: a rectangular domain.



Required domain search: $O(\log n)$.

In total:

- Query time: $O(\log n)$
- Memory usage: $O(\boldsymbol{n^2})$
- Preprocessing time: $O(\boldsymbol{n^2})$

Points Enumeration

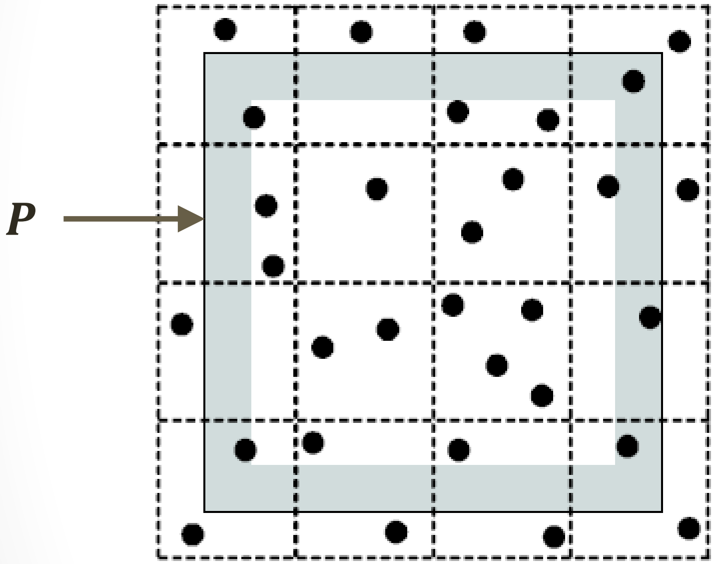
Given a set S of n points on the plane, **report all the points** of S inside a specified orthogonal query rectangle P.

Lower bounds:

- $\Omega(n)$ (when all the points are inside P).
- $\Omega(k)$, where k is the number of enumerated points (*output-sensitive* lower bound).

Our goal is to minimize the number of actions **beyond reporting the output points**.

Enumeration: Regular Grid



- Consider m^2 **identical** cells.
- The average number of points in a cell:

$$M = \frac{n}{m^2}$$

- Search of cells that intersect P:

$O(1)$ – constant!

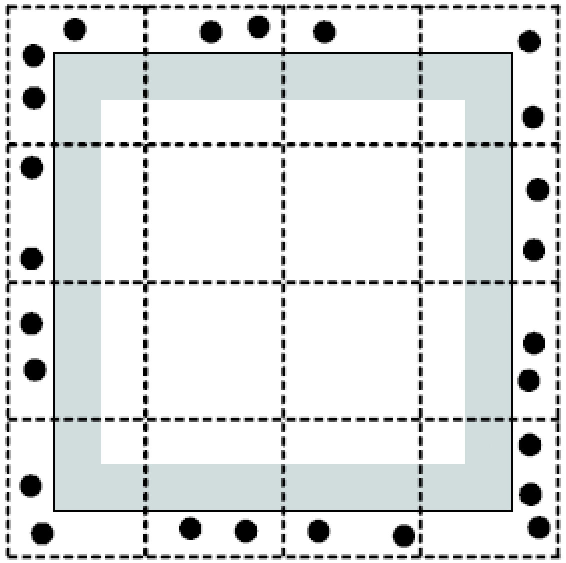
- Query time: $O(k + m_P)$,

where m_P is the number of intersected cells.

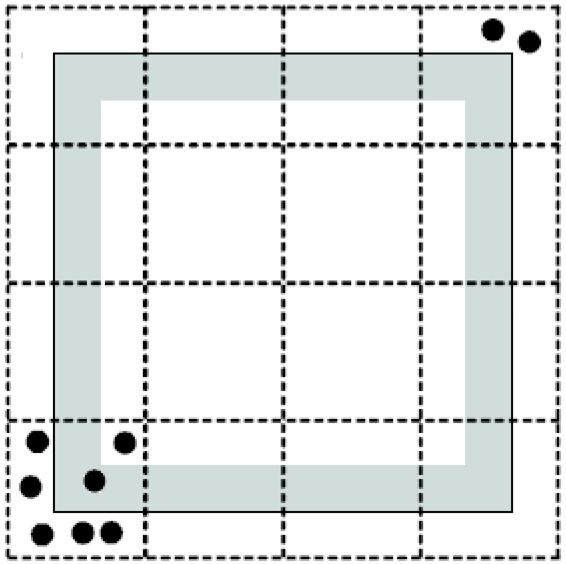
- Memory usage: $O(n + m^2)$
- Preprocessing time: $O(n + m^2)$

Regular Grid: Worst Cases

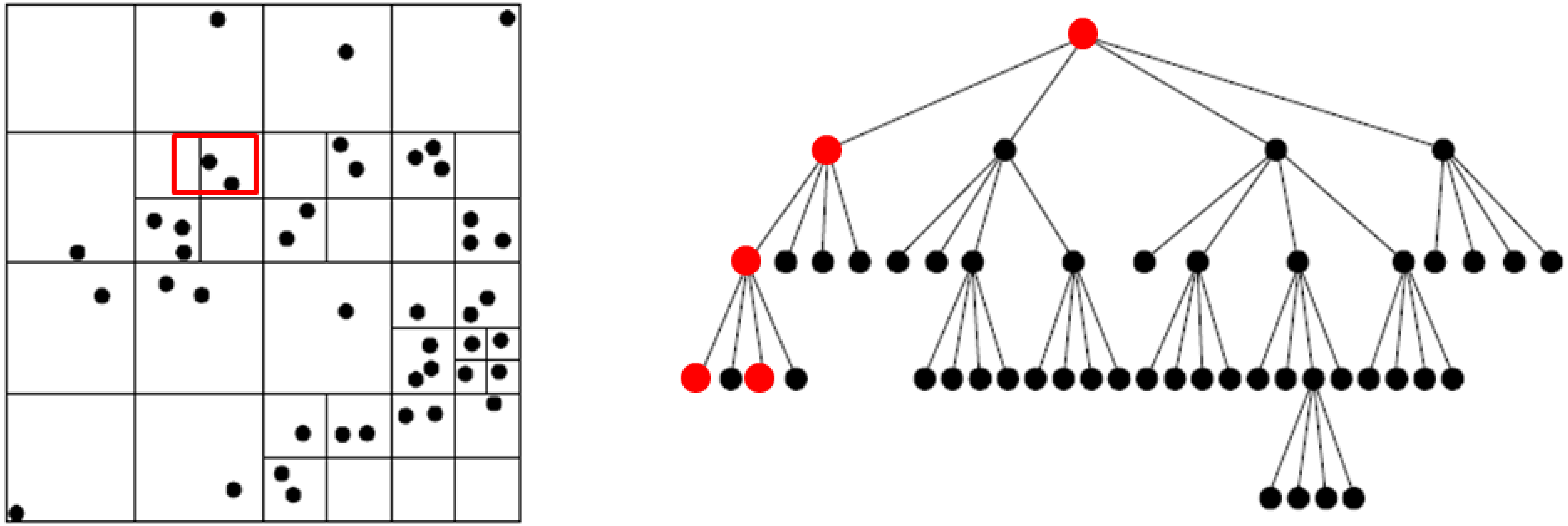
m^2 cells to be checked event if **no point is enumerated**.



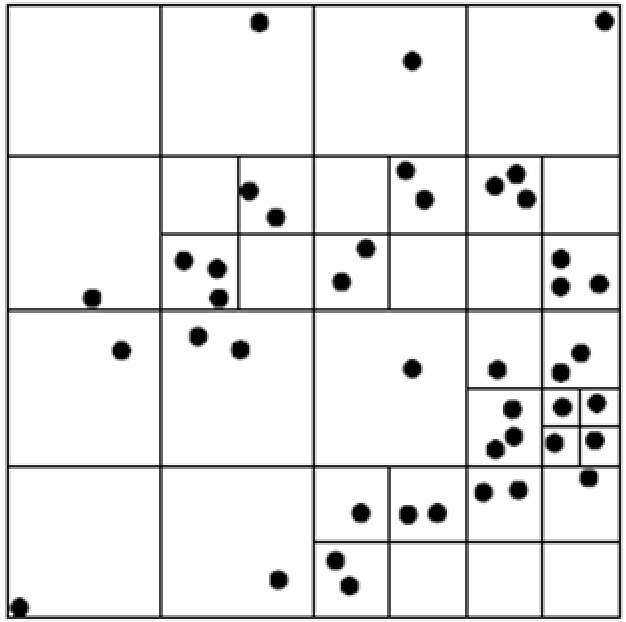
Inefficient memory usage if the points are distributed **non-regularly**.



Points Enumeration: the Quadtree

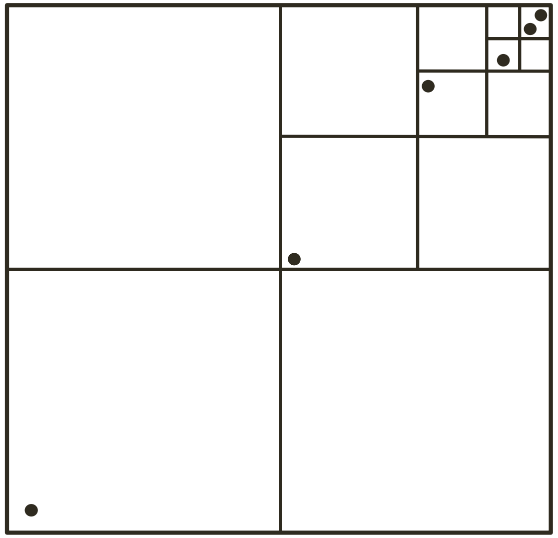


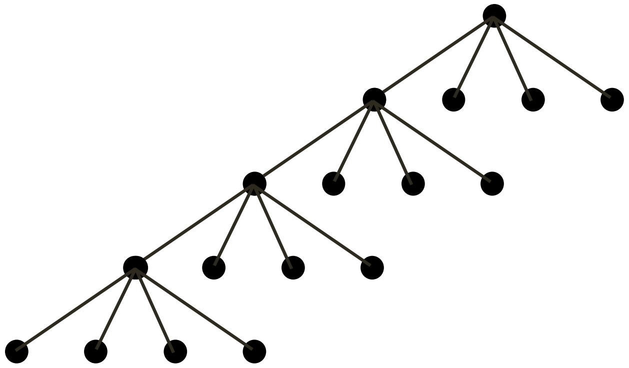
Quadtree: Analysis



- Max points in a cell **is a parameter**.
- Query time, **in average**: $O(\log_4 n)$
- Query time, **the worst case**: $O(n)$
- Memory usage: $O(n)$
- Preprocessing time: $O(n \log n)$

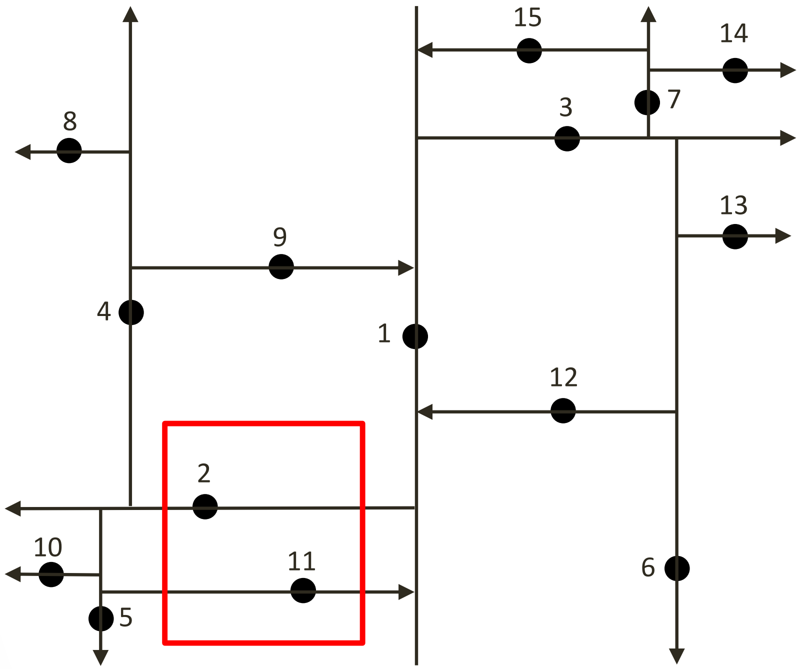
Quadtree: Analysis

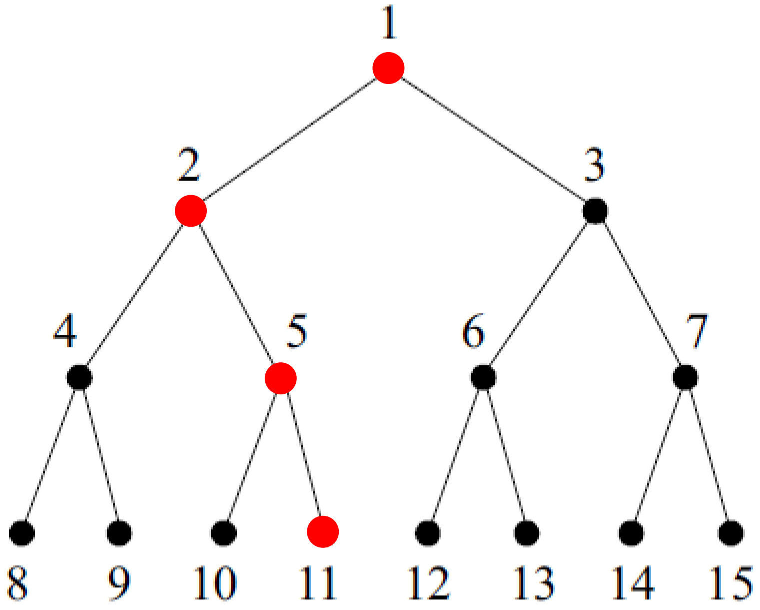




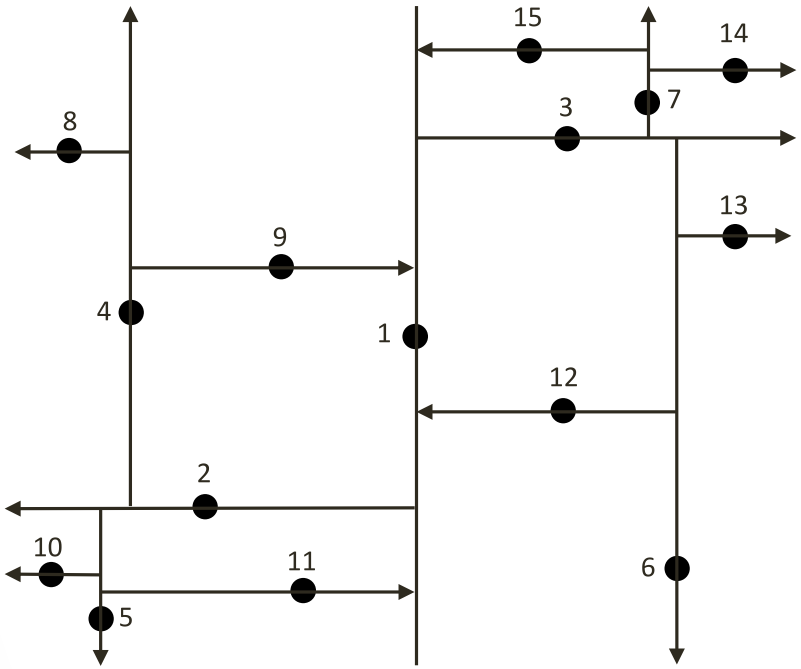
1. The algorithm does not consider points **position**, only the **number** of points per domain.
2. $O(n)$ query time in the worst case.
3. In the worst case $O(\sqrt{n})$ domains will be checked even if **no point is enumerated**.

Points Enumeration: 2-d-tree



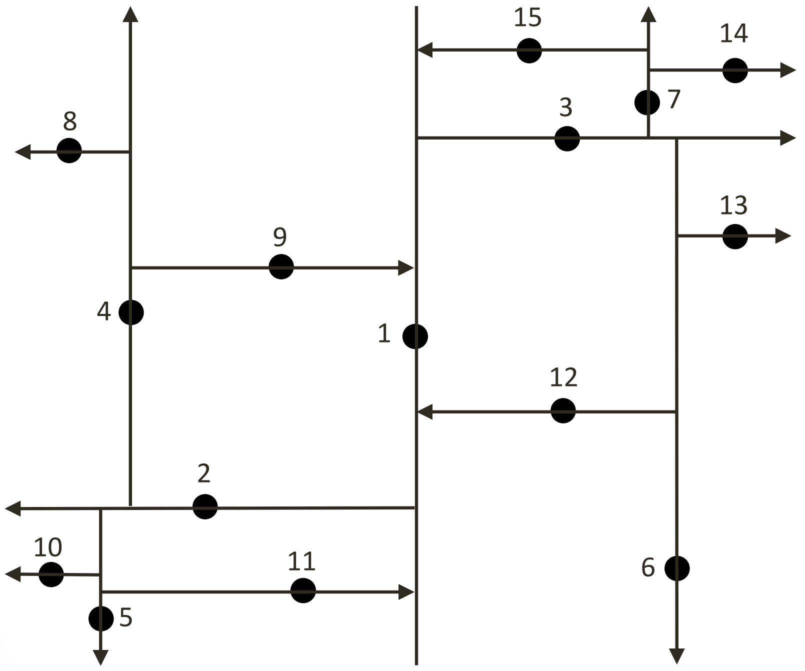


2-d-tree: Analysis



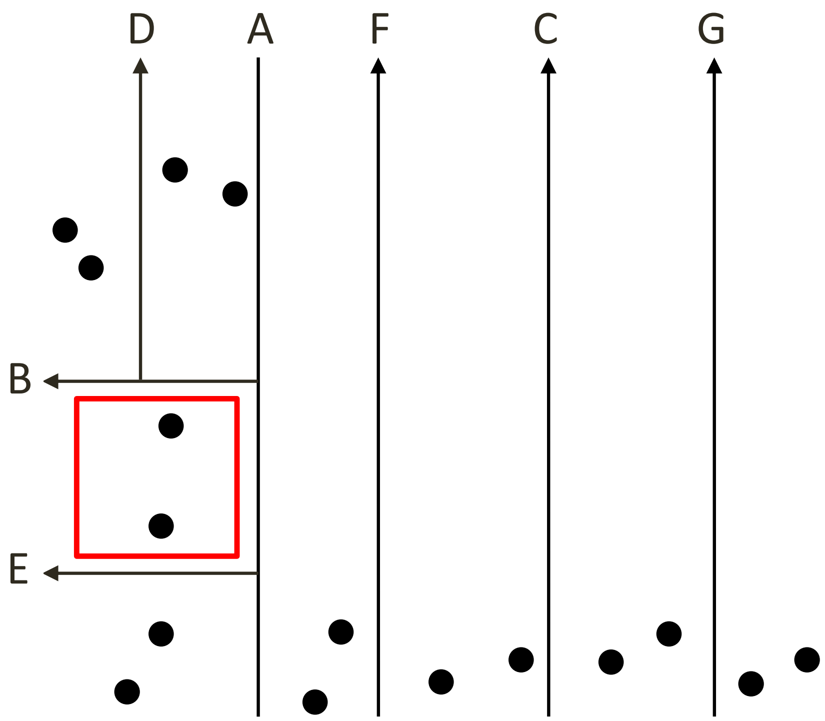
- The algorithm considers not only the number of points per domain, but also their **real position**.
- Query time, the worst case : $O(\log n)$
- Memory usage: $O(n)$
- Preprocessing time: $O(n \log n)$

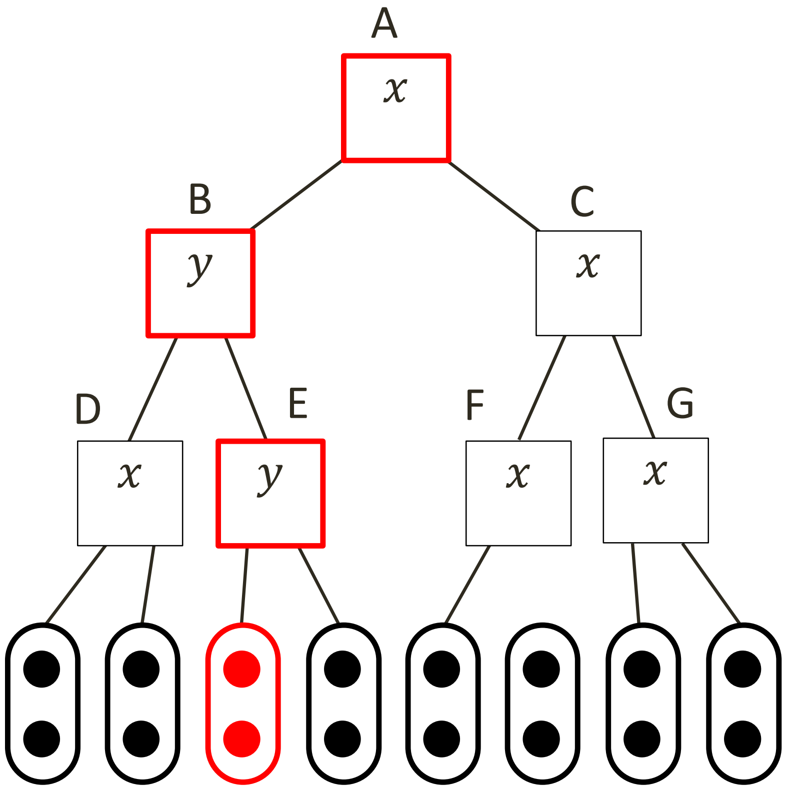
2-d-tree Properties



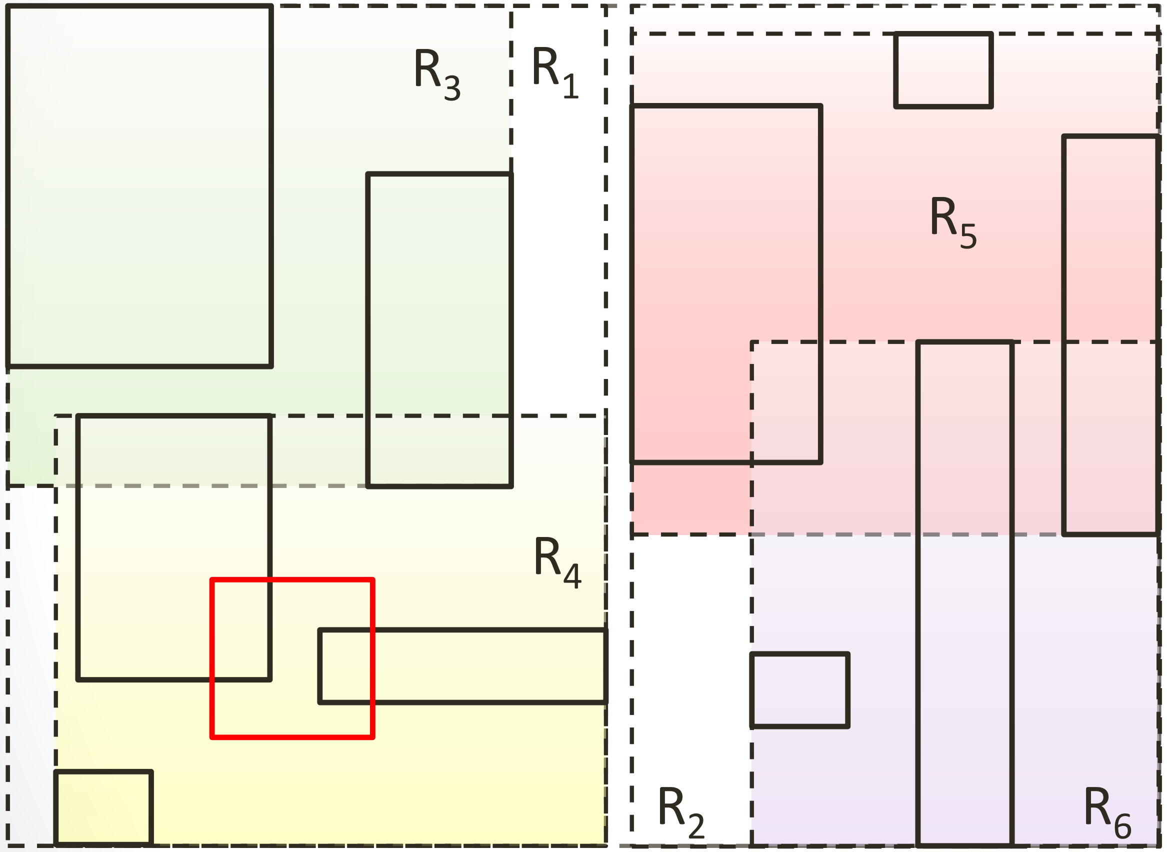
- All the points should be specified **prior** to the tree is created.
- Point removal makes the tree **inconsistent**.
- Instead of removal, points are marked (**virtual** removal) to be **physically** removed during periodical tree reconstruction.
- In the worst case $O(\sqrt{n})$ domains will be checked even if **no point is enumerated**.

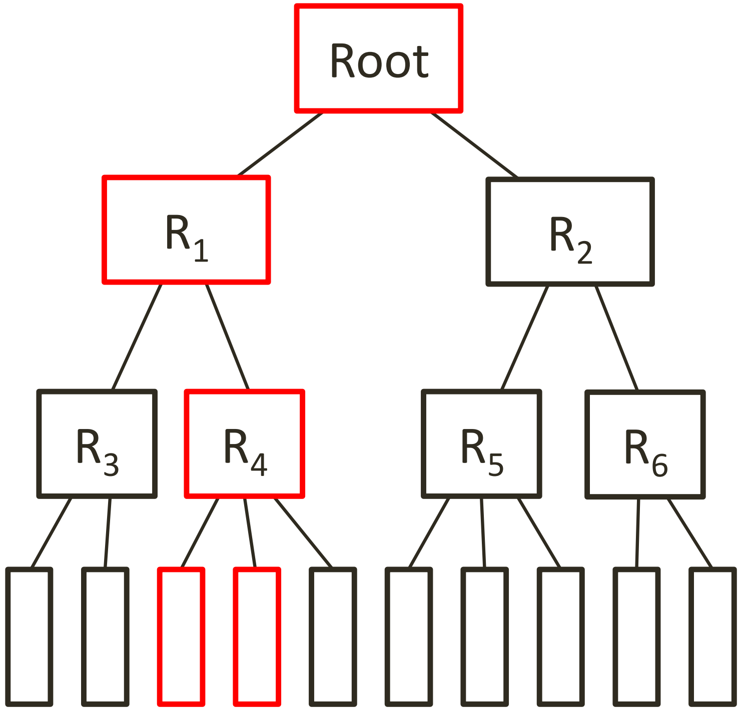
Adaptive 2-d-tree



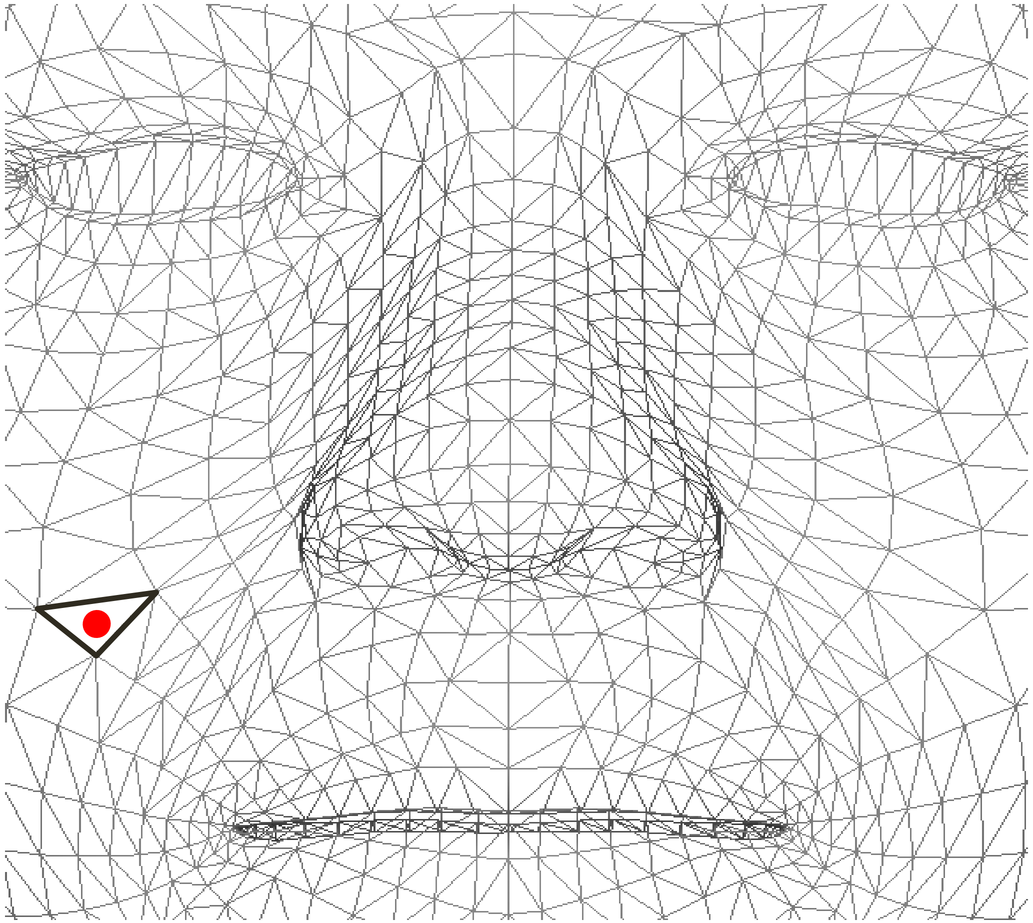


R-tree



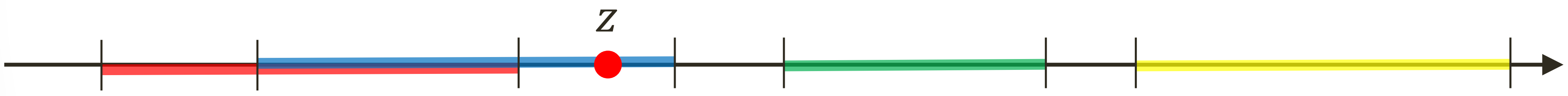


Point Location

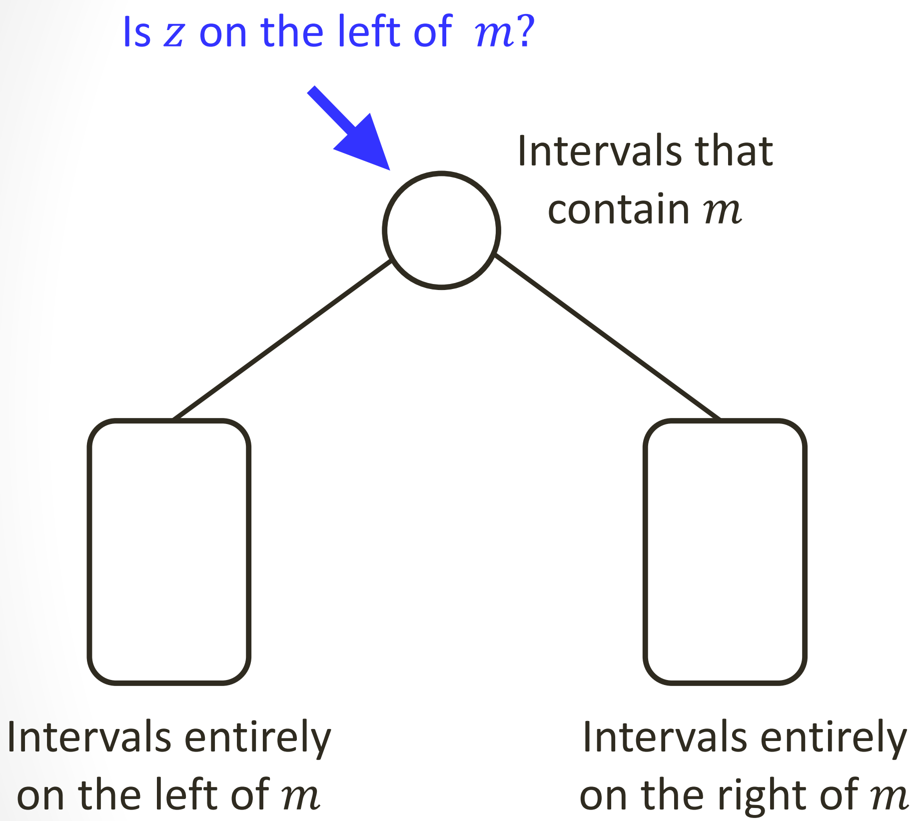


1-D Case

Consider linear intervals (*segments*) $[x_1, y_1], \ldots, [x_n, y_n]$ on a straight line which can overlap. Given a point z on the line, report intervals that contain z.



Interval Tree



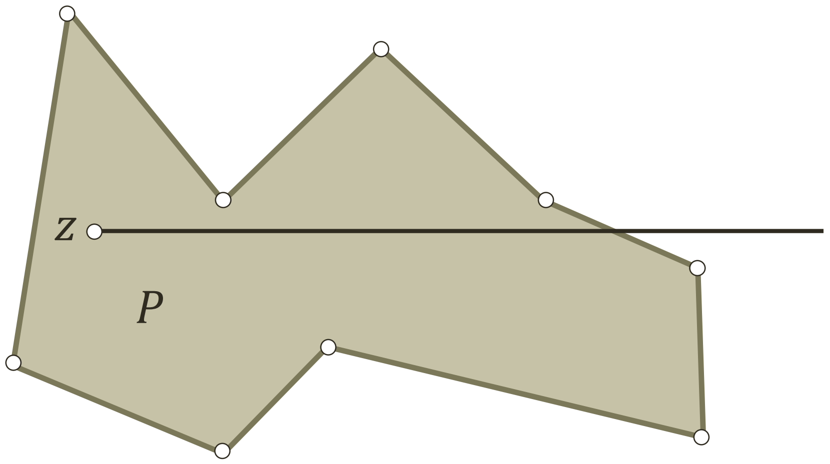
Query time $O(\log n + k)$, where k is the number of reported intervals.

- Find the **median** m of the endpoints of the intervals, takes $O(n)$ of preprocessing.
- Put the intervals that do not contain m to the left and right child trees of the root, respectively.
- The root itself contains two lists of intervals that contain m:
  1. Sorted by the left endpoint.
  2. Sorted by the right endpoint.
- If $\mathrm{z} > m$, then report the intervals from the list 2 whose right endpoints are on the right of z and consider recursively the right child. Consider similarly the case $\mathrm{z} < m$.

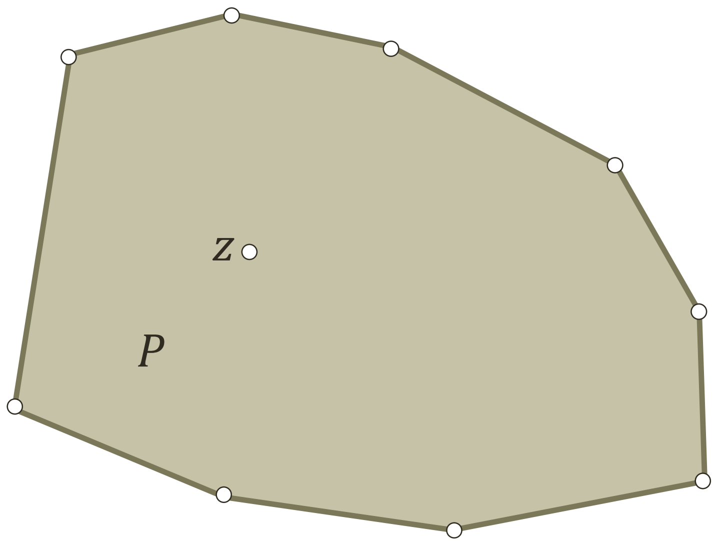
Simple Polygon

Given a simple polygon P with n vertices and a point z, determine whether z is inside P.

<u>Case 1</u>: single query time is $O(n)$.

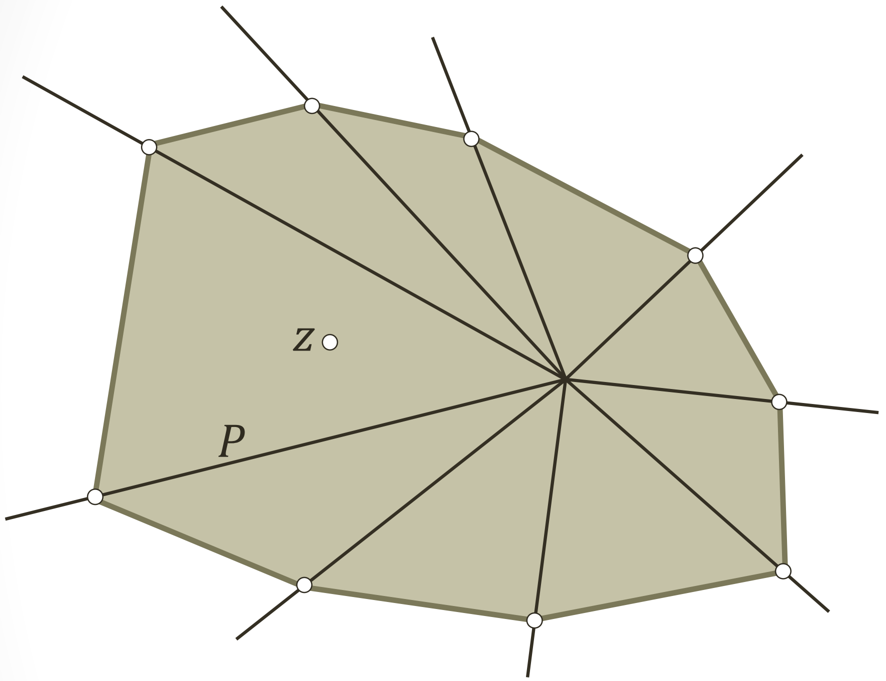


Convex polygon



Given a convex polygon P with n vertices and a point z, determine whether z is inside P.

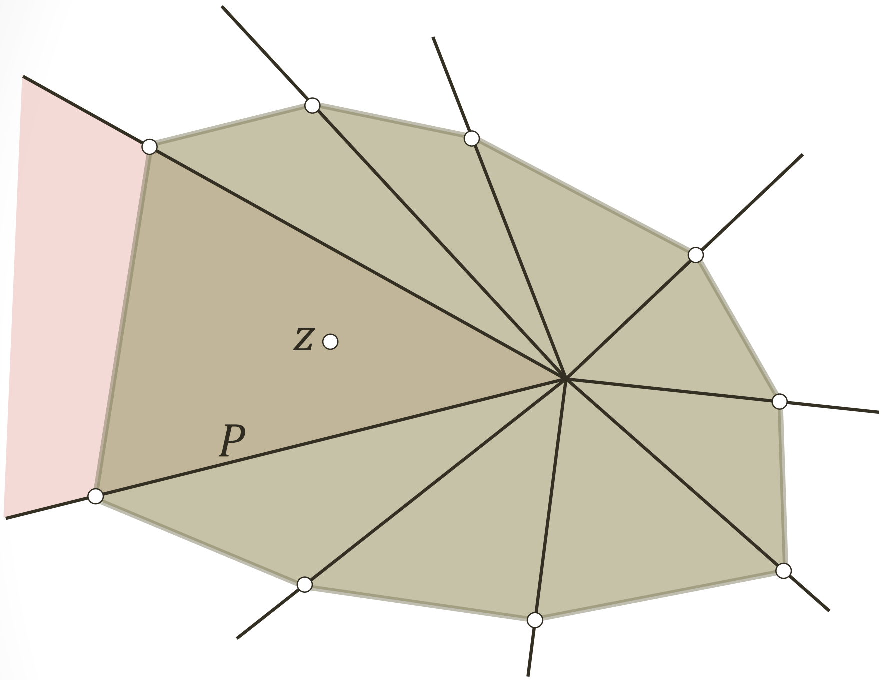
Convex Polygon - 1



Given a convex polygon P with n vertices and a point z, determine whether z is inside P.

1. **Preprocessing:** divide P into sectors relative to an arbitrary internal point, takes $O(n)$.

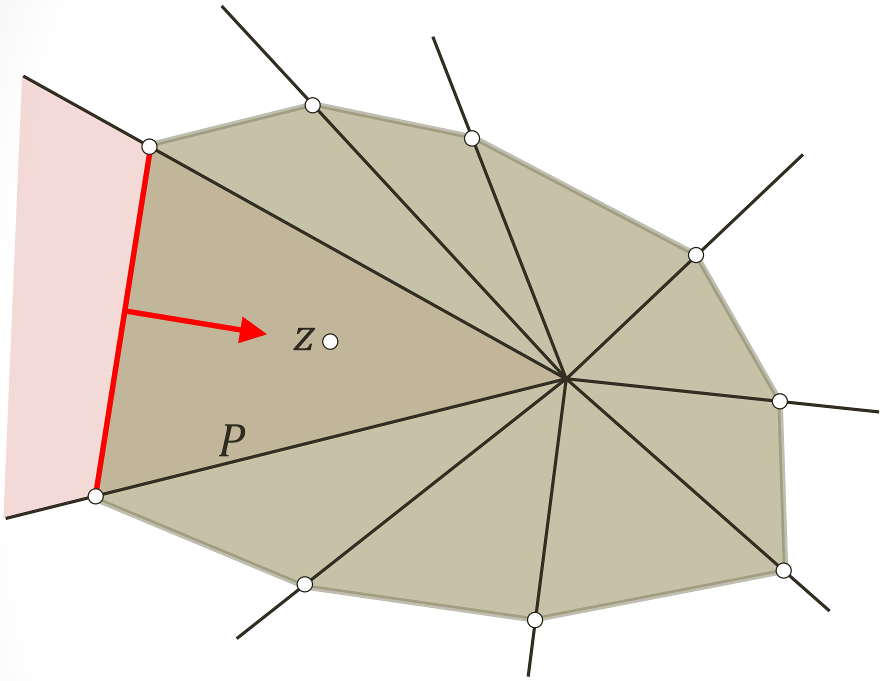
Convex Polygon - 1



Given a convex polygon P with n vertices and a point z, determine whether z is inside P.

1. **Preprocessing:** divide P into sectors relative to an arbitrary internal point, takes $O(n)$.
2. Find in time $O(\log n)$ the sector containing z.

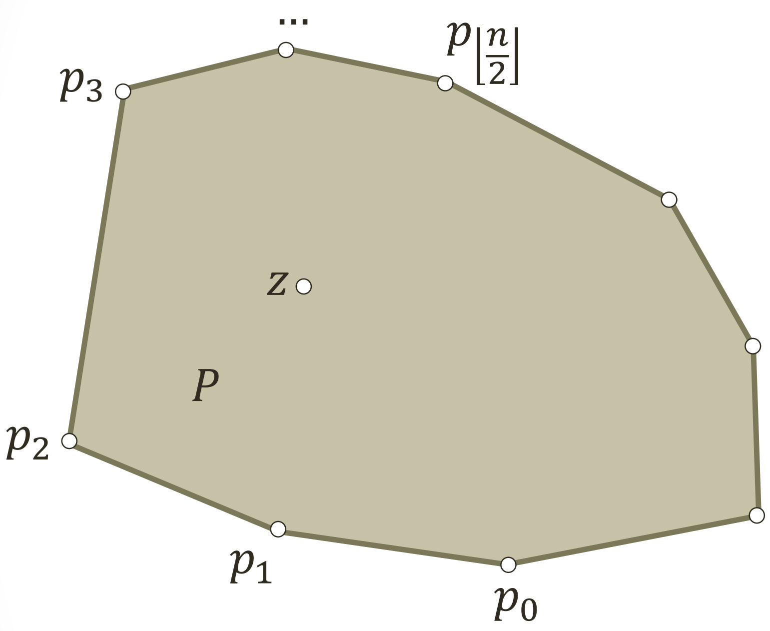
Convex Polygon - 1



Given a convex polygon P with n vertices and a point z, determine whether z is inside P.

1. **Preprocessing:** divide P into sectors relative to an arbitrary internal point, takes $O(n)$.
2. Find in time $O(\log n)$ the sector containing z.
3. Find in $O(1)$ the position of z relative to the corresponding edge of the polygon.

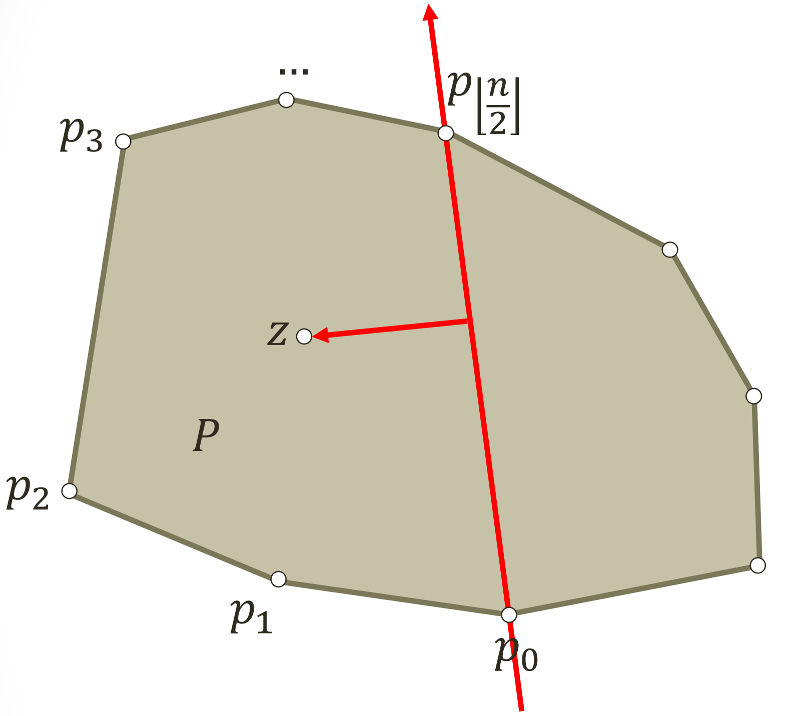
Convex Polygon - 2



Given a convex polygon P with n vertices and a point z, determine whether z is inside P.

1. If $n = 3$, determine whether z belongs to the triangle, takes $O(1)$.

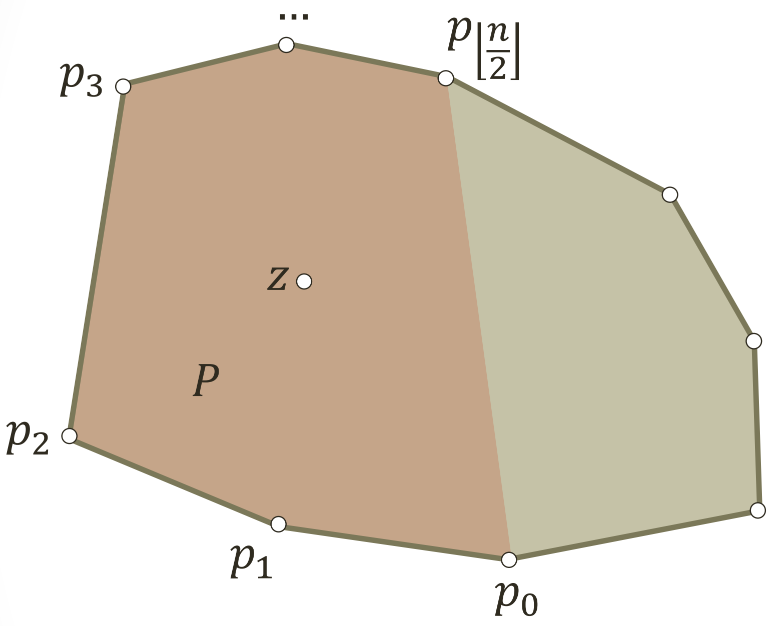
Convex Polygon - 2



Given a convex polygon P with n vertices and a point z, determine whether z is inside P.

1. If $n = 3$, determine whether z belongs to the triangle, takes $O(1)$.
2. If $n > 3$, determine at which side z is located regarding the line $\left(p_0, p_{\lfloor \frac{n}{2} \rfloor}\right)$.

Convex Polygon - 2



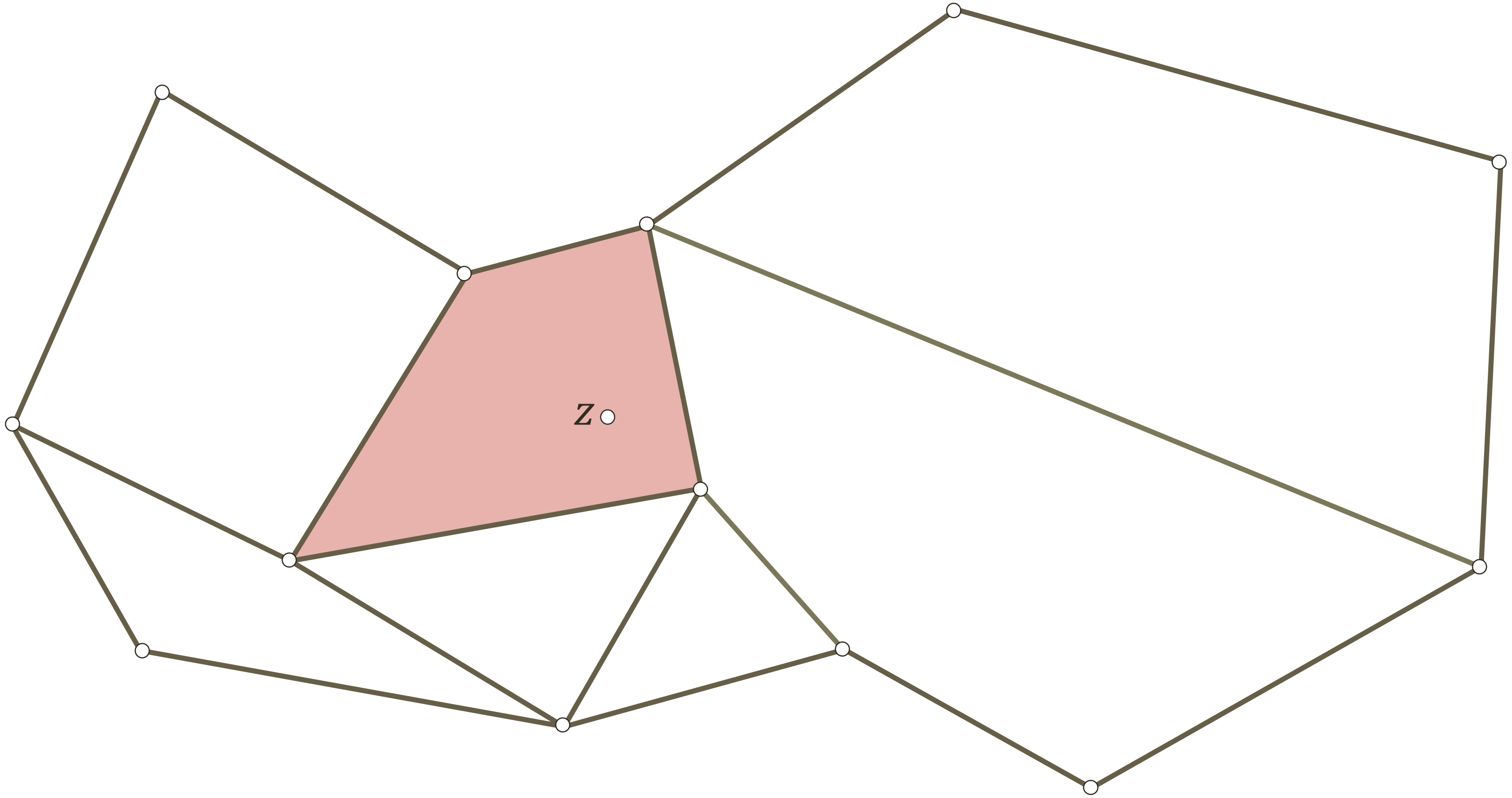
Given a convex polygon P with n vertices and a point z, determine whether z is inside P.

1. If $n = 3$, determine whether z belongs to the triangle, takes $O(1)$.
2. If $n > 3$, determine at which side z is located regarding the line $\left(p_0, p_{\lfloor \frac{n}{2} \rfloor}\right)$.
3. Recursively apply steps 1-3 to the corresponding part of P.

Query time: $O(\log n)$.

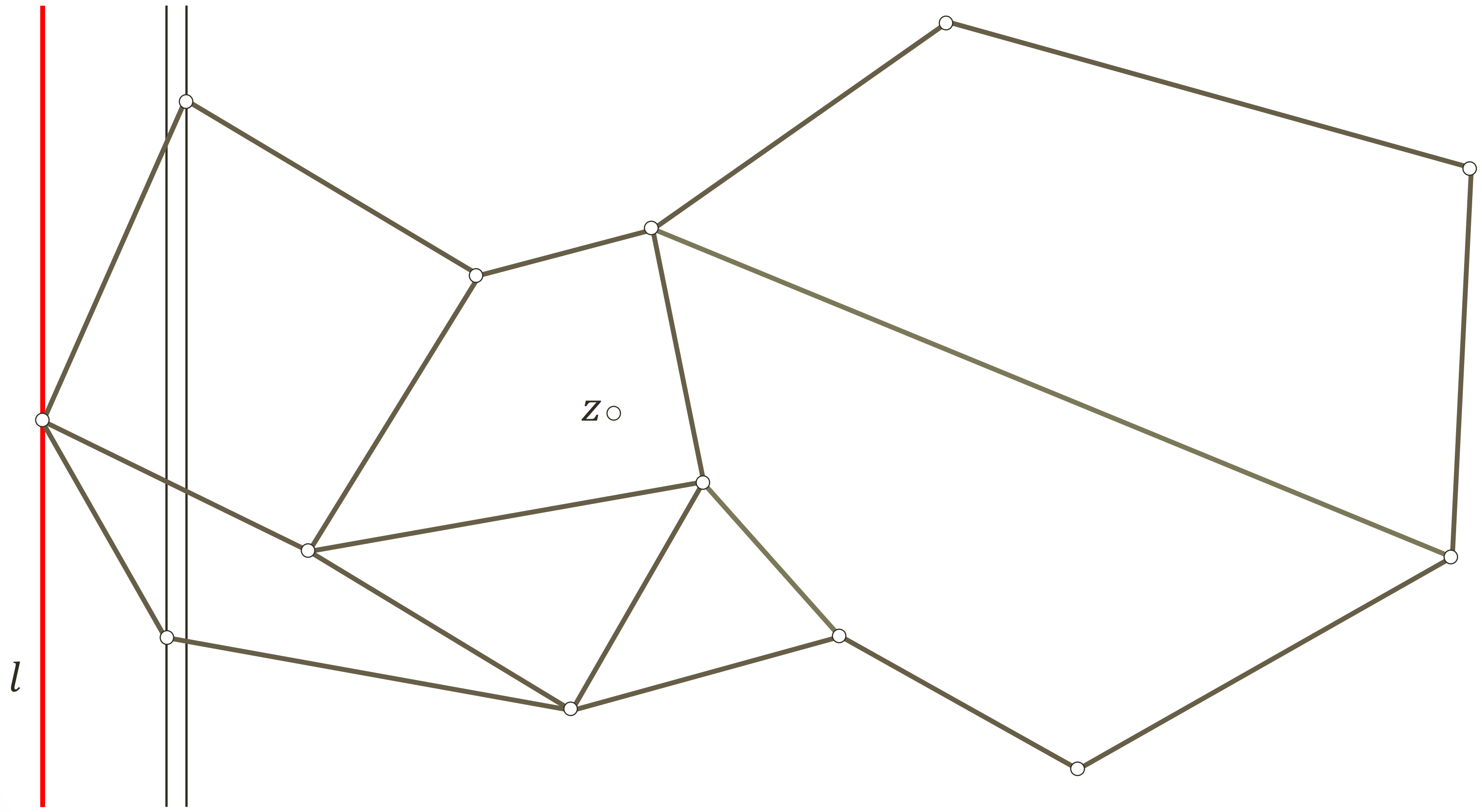
Point Location on a Plane Partition

Given planar straight line graph P with n vertices and a point z, find a facet of P that contains z.



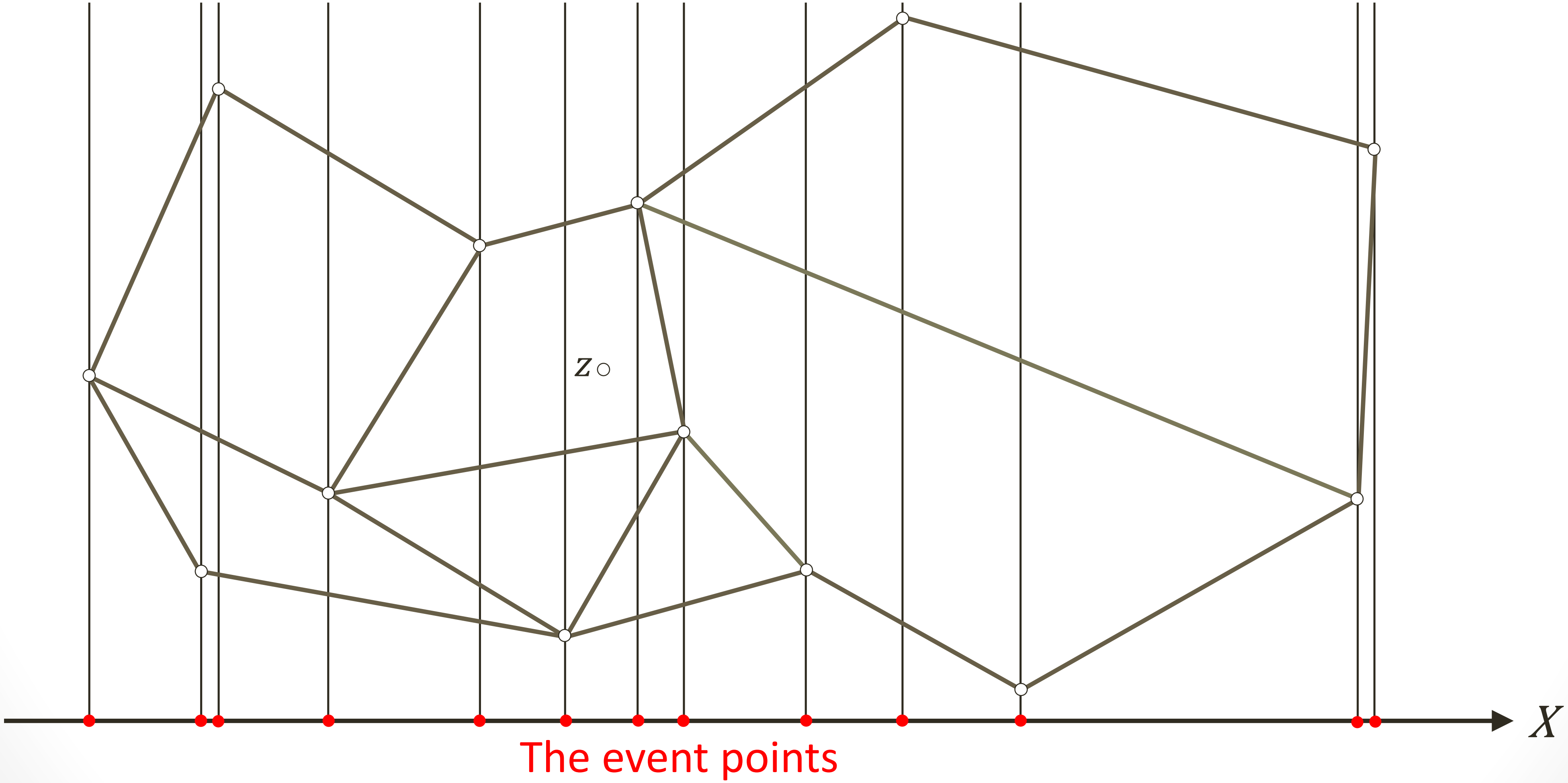
Sweep Line Technique

The idea: move a straight line across the plane and analyze the objects this line intersects.



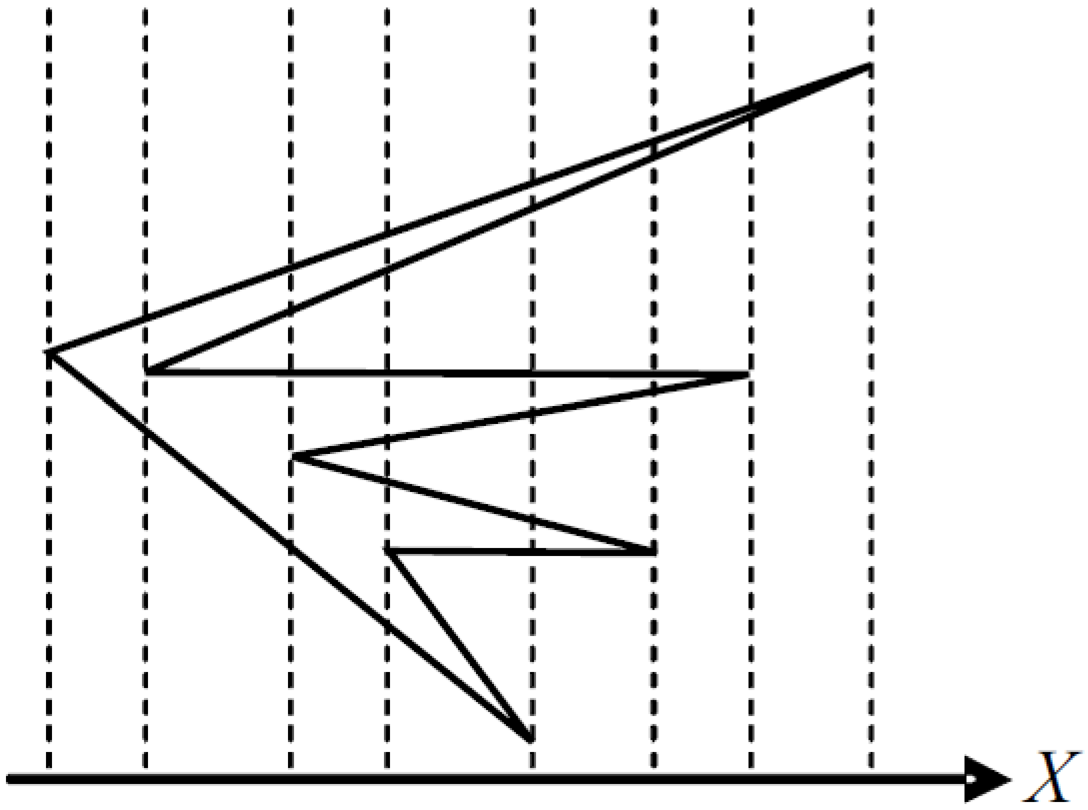
Sweep Line Technique

The idea: move a straight line across the plane and analyze the objects this line intersects.



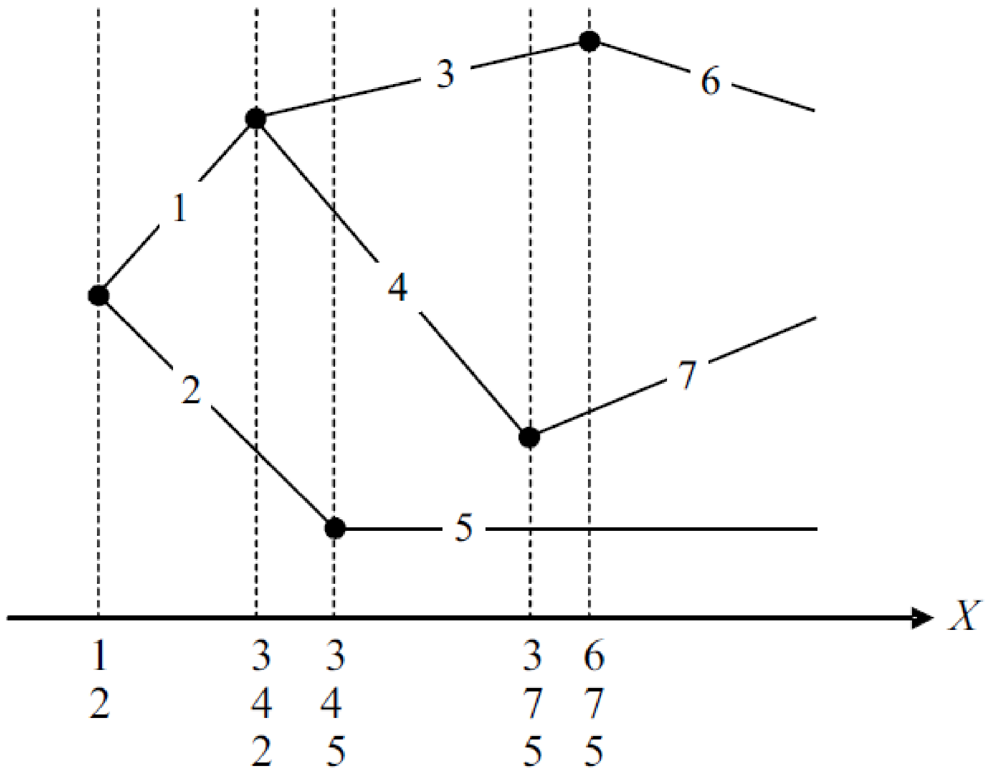
Complexity

1. Query time: $O(\log n)$
2. Preprocessing:
   - Sorting the vertices by X: $O(n \log n)$.
   - Sorting the edges inside each strip: $O(n^2 \log n)$, but can be improved by the **sweep line technique**.



Sweep Line Technique

1. The sweeping line (SL) is moving discreetly passing through the event points.
2. In an event point some event takes place that modifies the state of the SL. NO events happen between the event points.
3. Here: the *state* of the SL is the set of edges intersecting the SL sorted by their Y-coordinate.



Complexity

- Insertion and deletion of edges: $O(n \log n)$.
- Storing edges in the strips: $O(n^2)$.

C++ Implementation

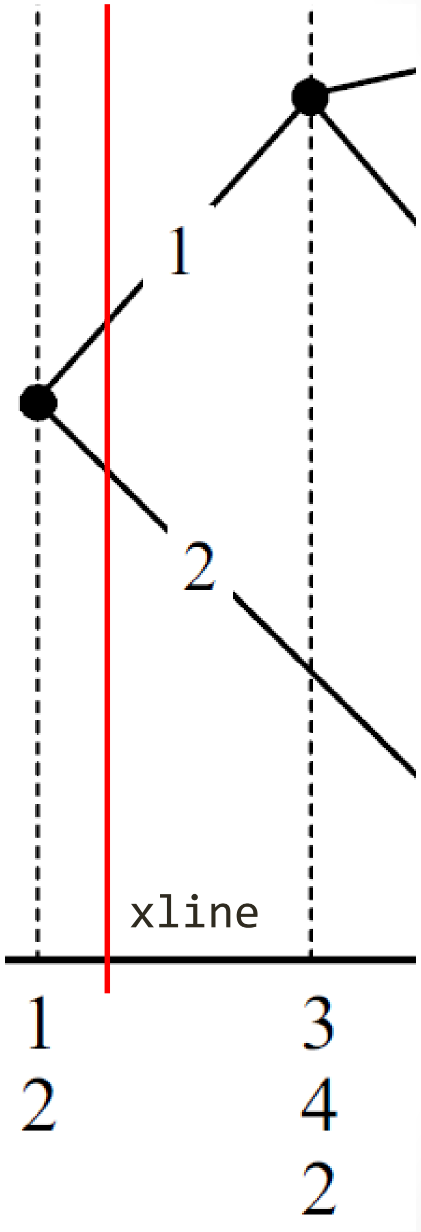
SL state is dynamic STL container providing insertion, deletion and searching in time $O(\log n)$.

```
typedef std::function<bool(int, int)> SegmentComparator;
typedef std::map<double, int, SegmentComparator> SweepLineStatus;

// Returns y-coordinate of the intersection point of the segment
// iseg and the SL with x-coordinate equal to xline.
double intersection_y(int iseg, double xline);

double xline; // x-coordinate of the SL

SweepLineStatus sw([&](int l, int r)
    {
        return intersection_y(l, xline) < intersection_y(r, xline);
    }
);
```



Point Location (continued)

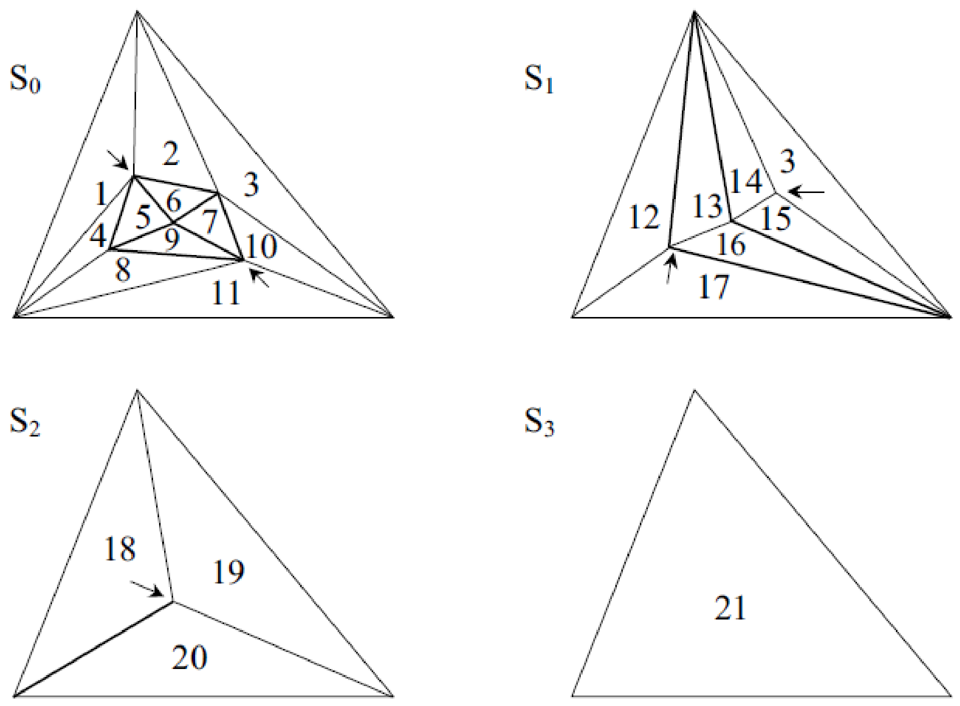
Triangulation refinement technique

Kirkpatrick (1983):

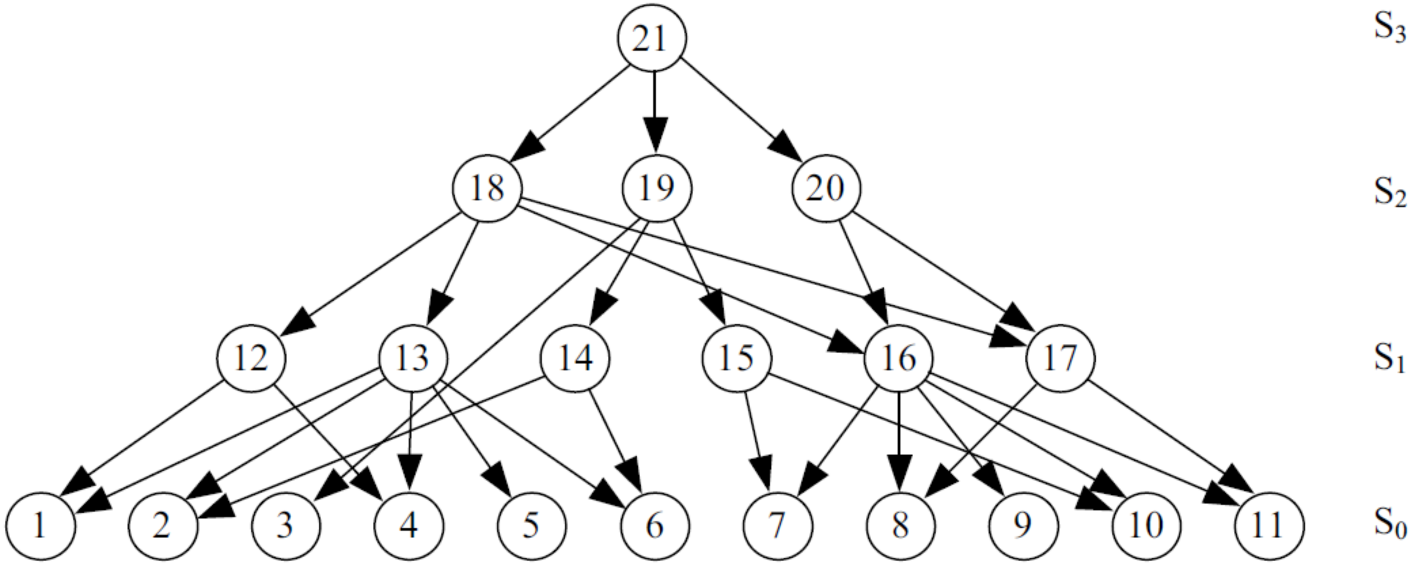
- Query time: $O(\log n)$
- Preprocessing time: $O(n \log n)$
- Memory usage: $O(n)$.

Triangulation Refinement

1. Build a triangle that encompasses the initial triangulation.
2. Triangulate the area between this triangle and the initial triangulation.
3. Build a sequence of triangulations $S_0, \ldots, S_{h(n)}$:
   - S_0 is the initial triangulation.
   - Remove from S_i some set of non-adjacent vertices and the corresponding incident edges.
   - Build a new triangulation S_{i+1} by triangulating empty areas coming out from these deletions.



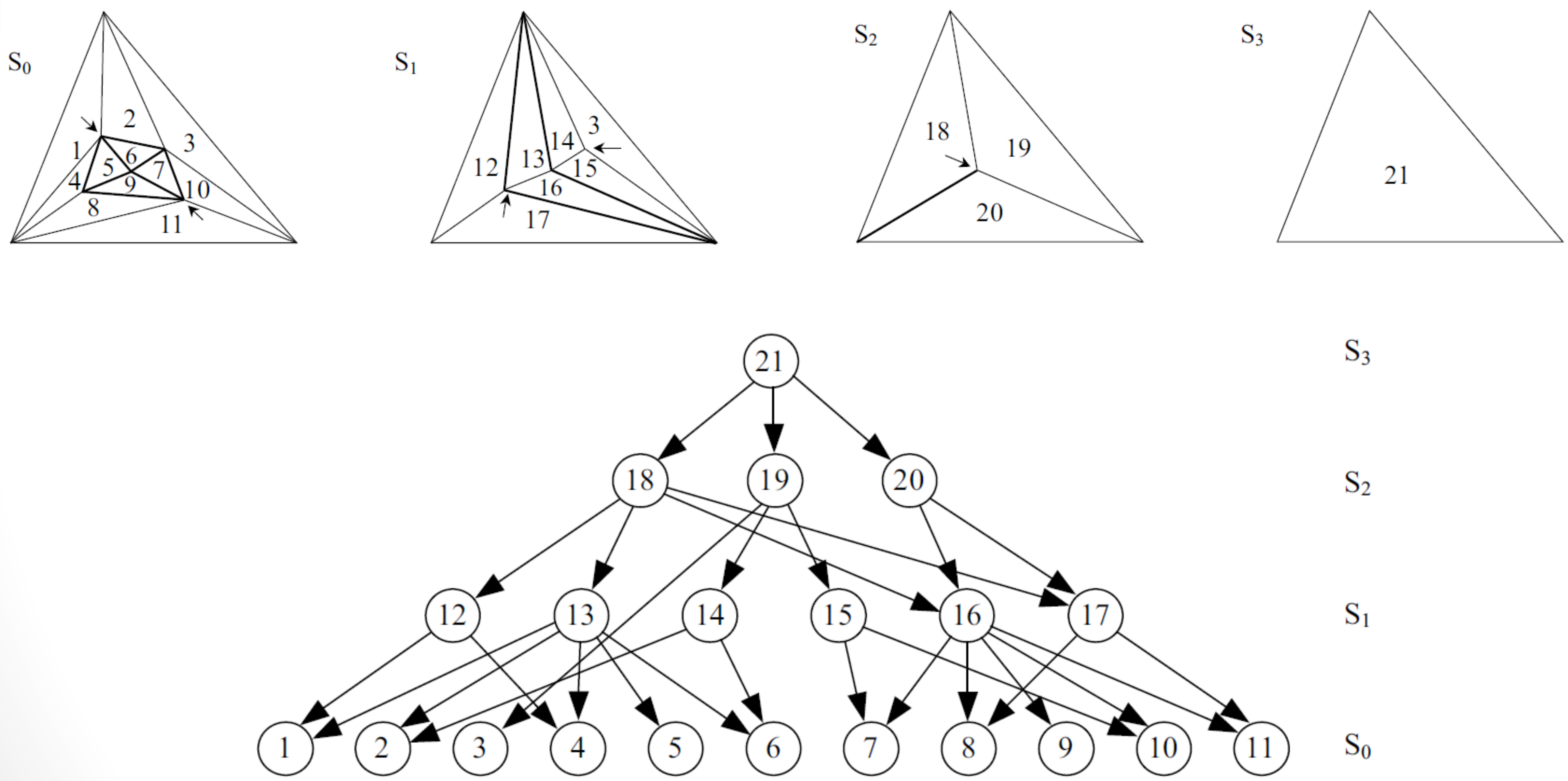
Triangulation Refinement



4. Build an oriented graph:
   - The vertices correspond to the triangles of the triangulations.

- An arc $(t_i,\ t_j)$ is created if the triangles t_i и t_j overlap, t_i has been removed from the previous triangulation, and t_j has been created while triangulating the empty areas.

Triangulation Refinement



Triangulation Refinement

Запрос:

Point-Location(z, T)

$v \leftarrow root[T]$
if $z \notin t(v)$ **then**
    return NIL
while $c(v) \neq \emptyset$ **do**
    for (по всем) $u \in c(v)$ **do**
        if $z \in t(u)$ **then**
            $v \leftarrow u$
return v

Triangulation Refinement

Point location on a planar straight line graph can be performed in time $O(\log n)$ using $O(n)$ memory and $O(n \log n)$ time for preprocessing.

Questions?