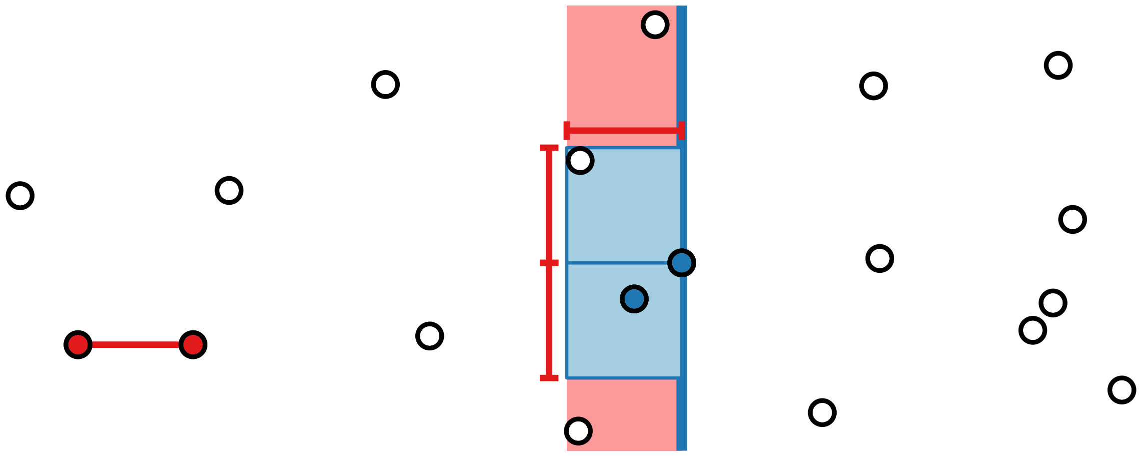
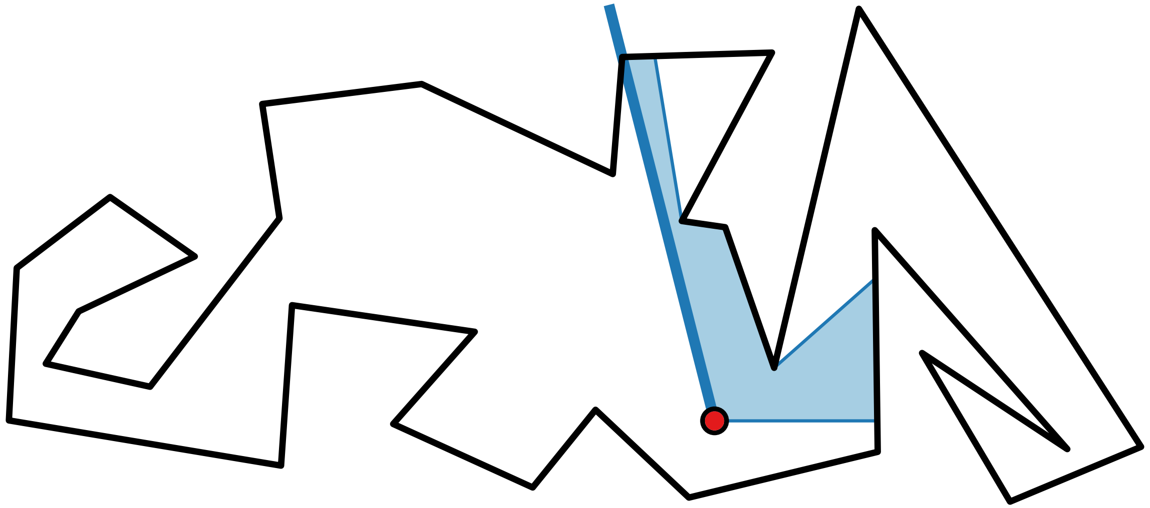
Advanced Algorithms

Computational Geometry
Sweep-Line Algorithms

Johannes Zink · WS23/24





Introduction

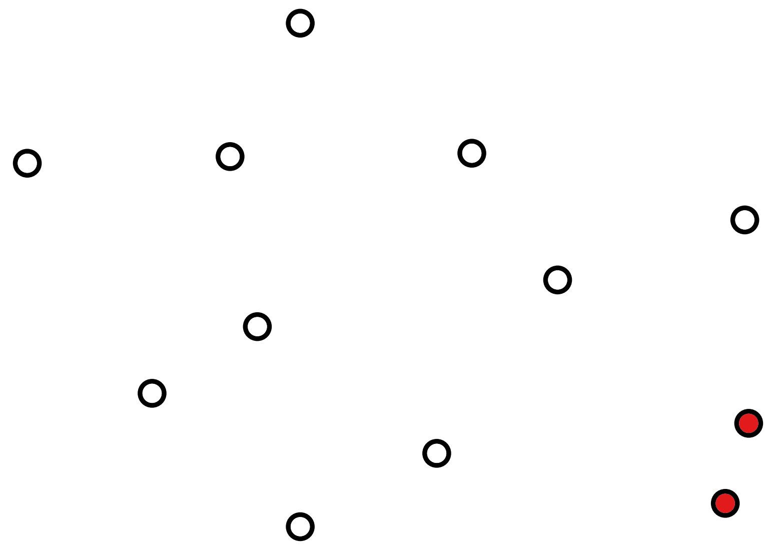
Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair

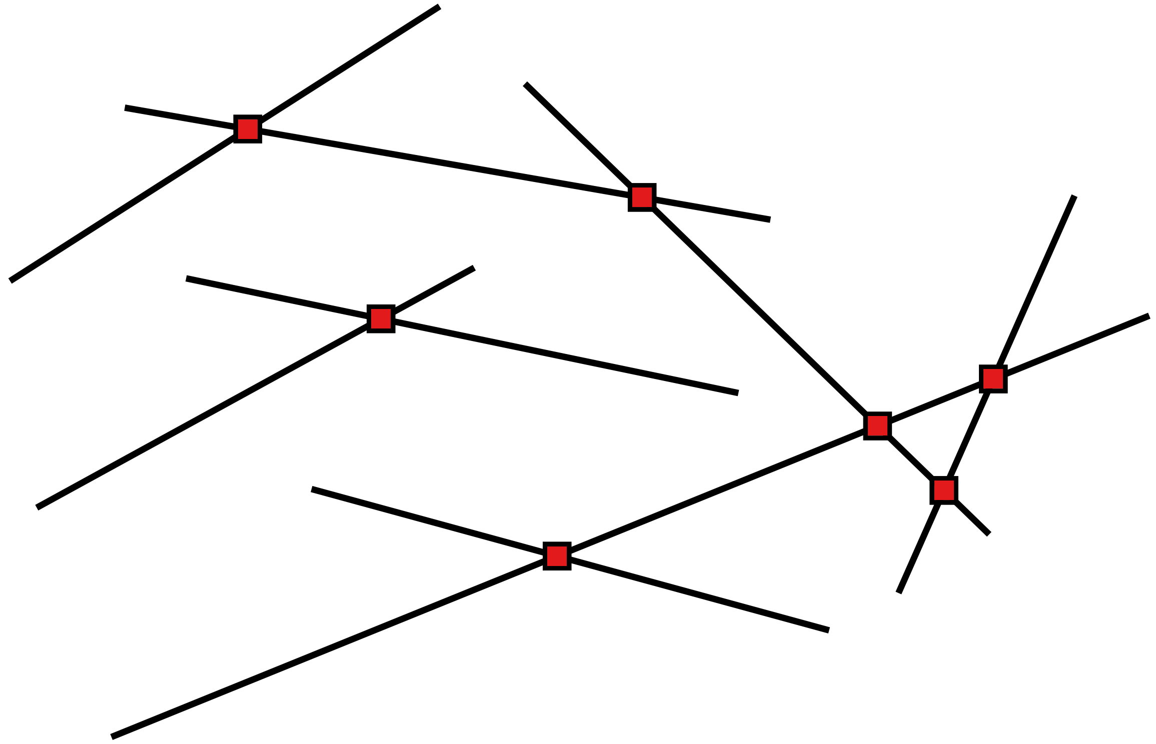


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection

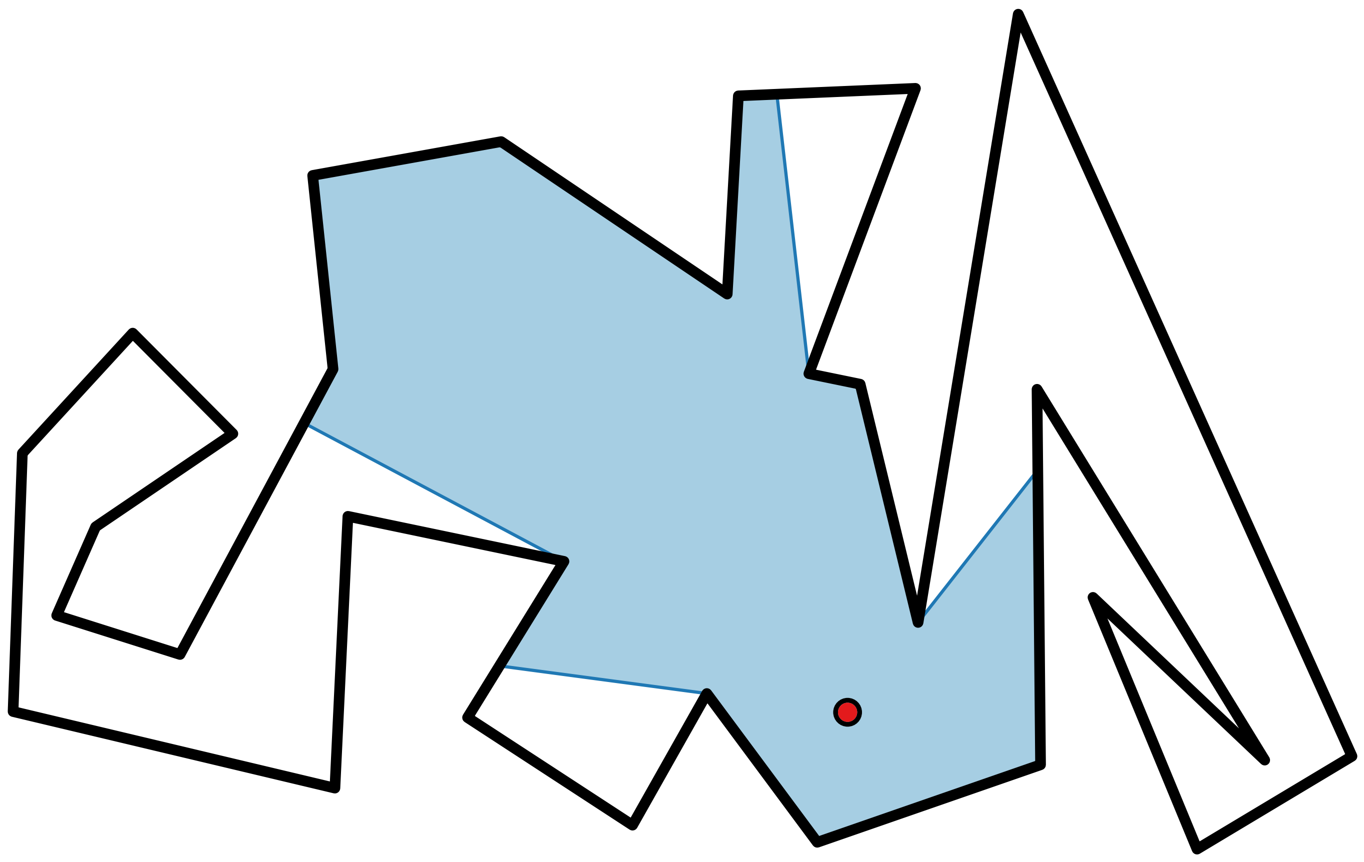


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility

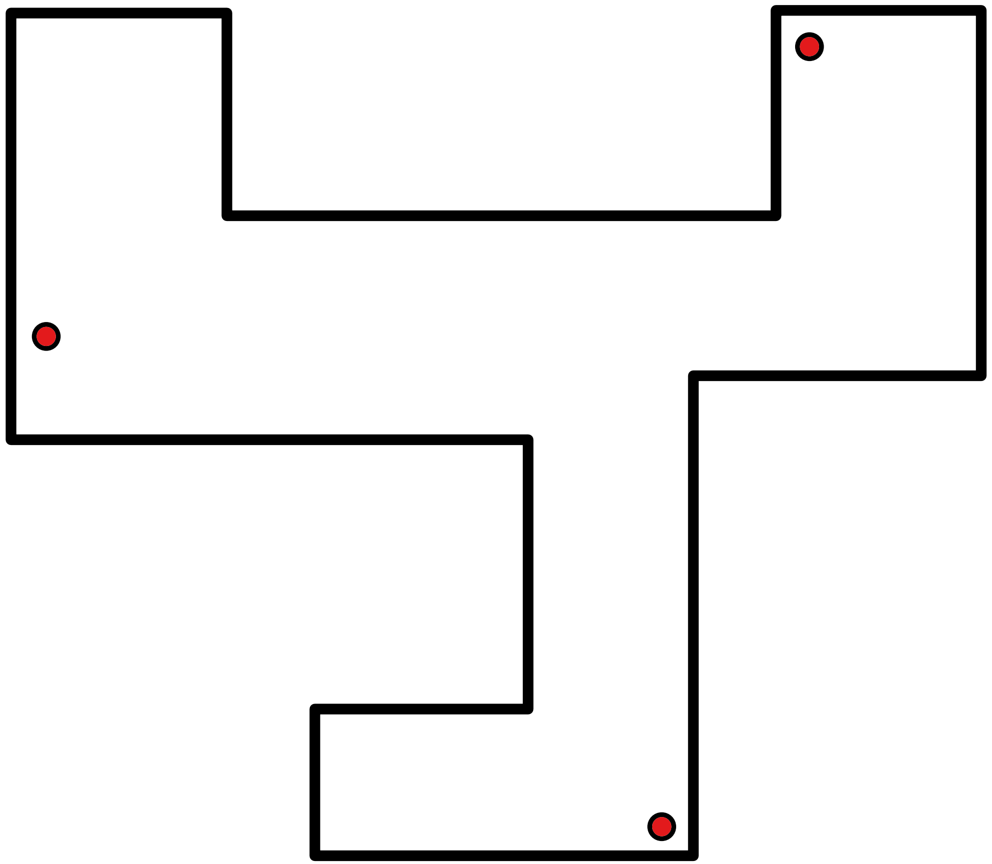


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery

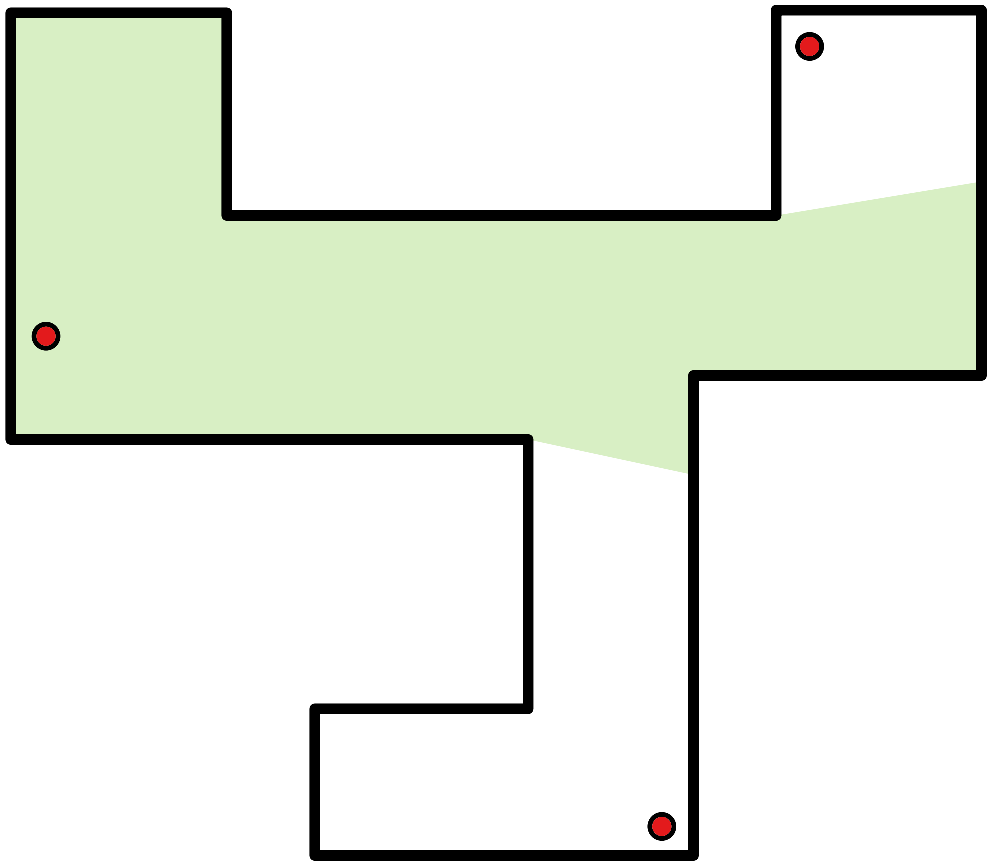


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery

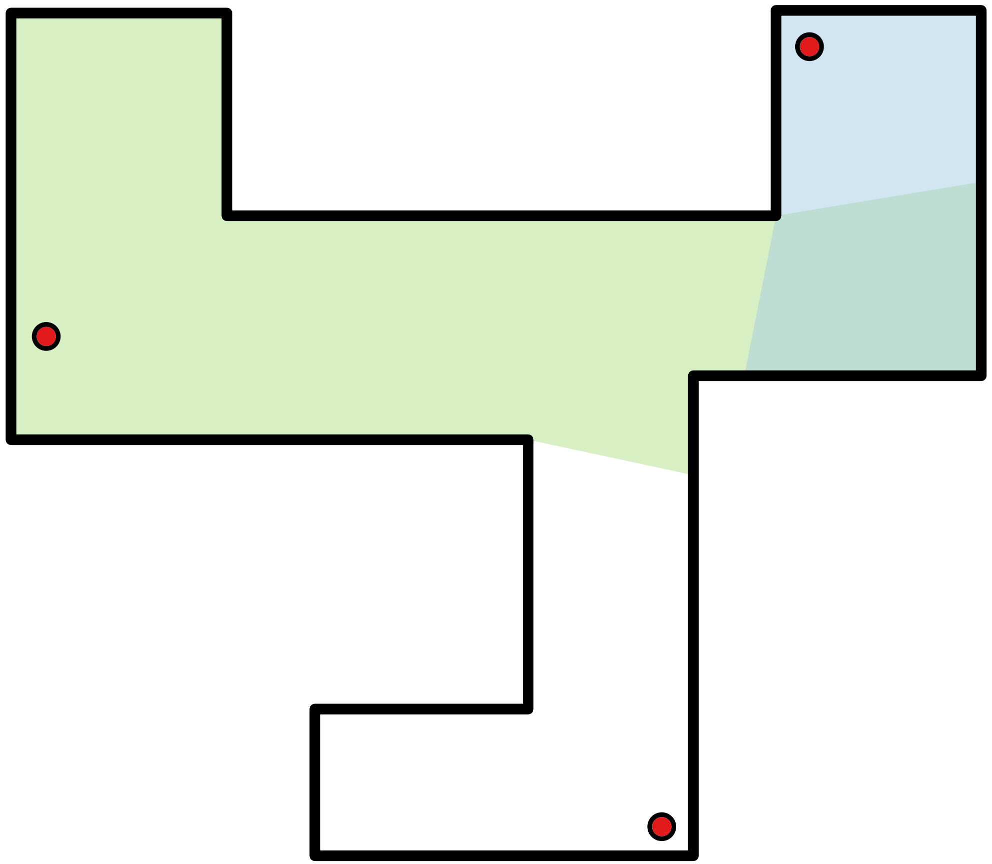


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery

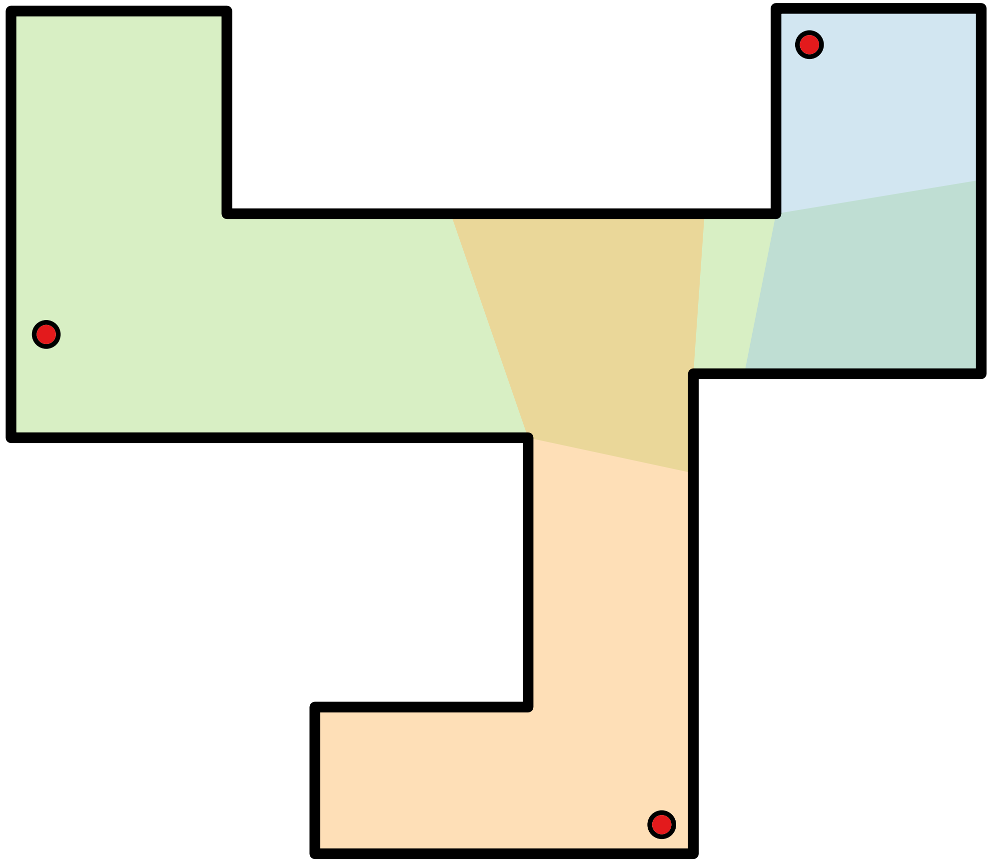


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery

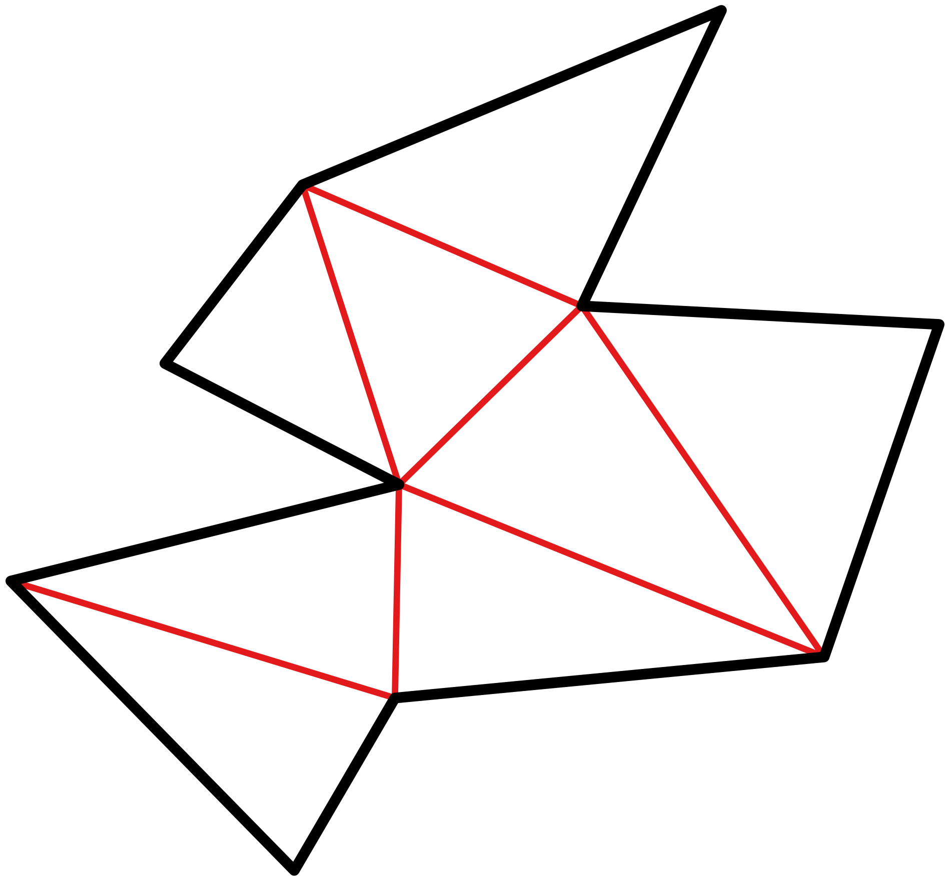


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery
- Triangulating a polygon

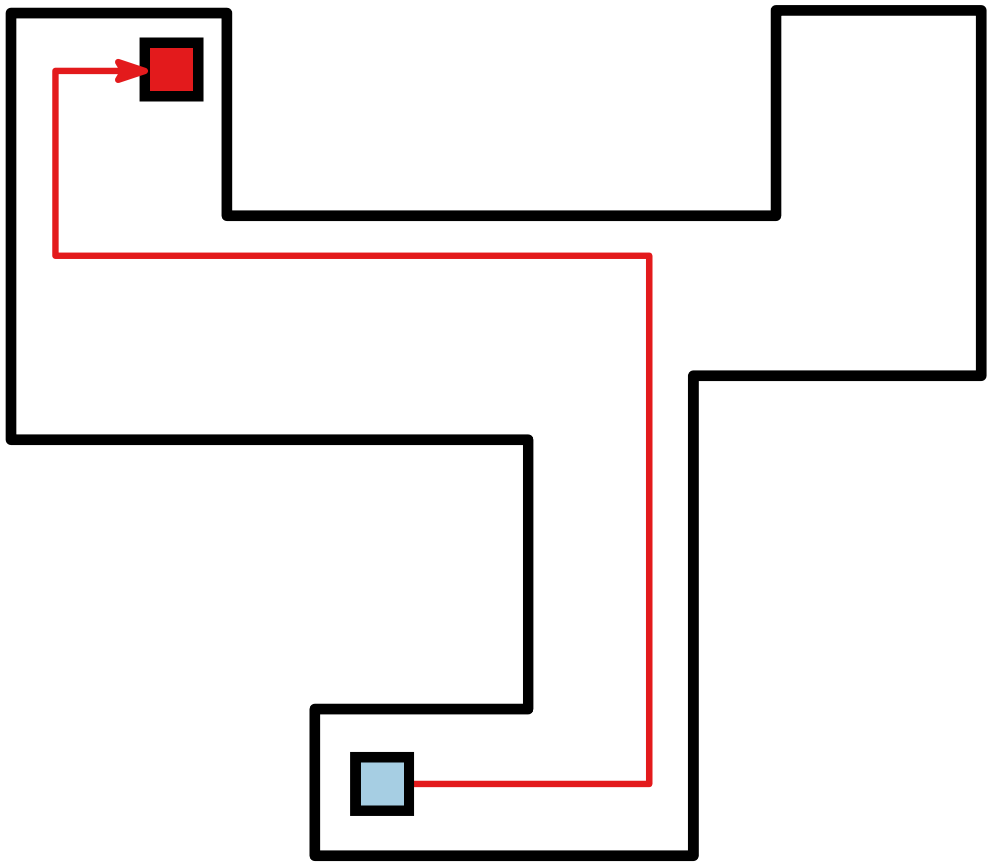


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery
- Triangulating a polygon
- Motion planning

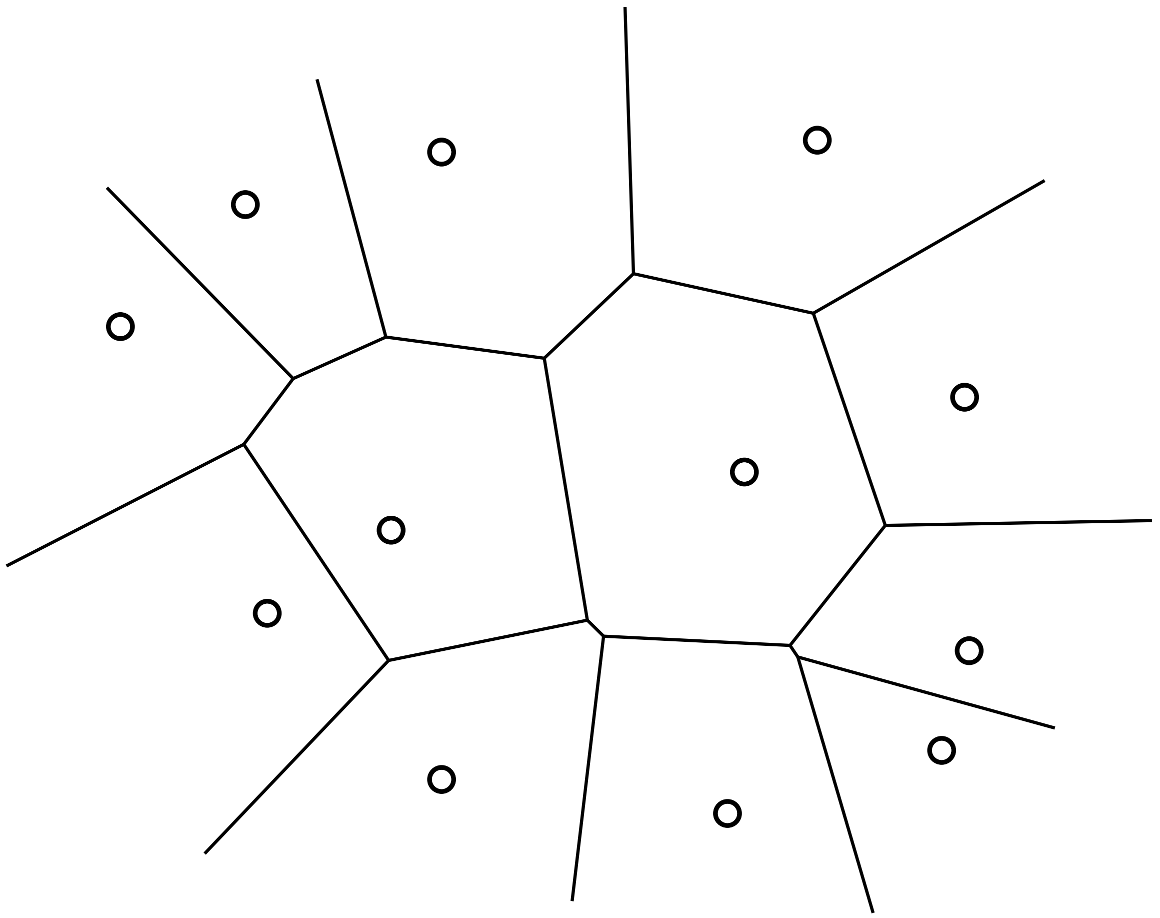


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, ...

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery
- Triangulating a polygon
- Motion planning
- Finding the closest post office

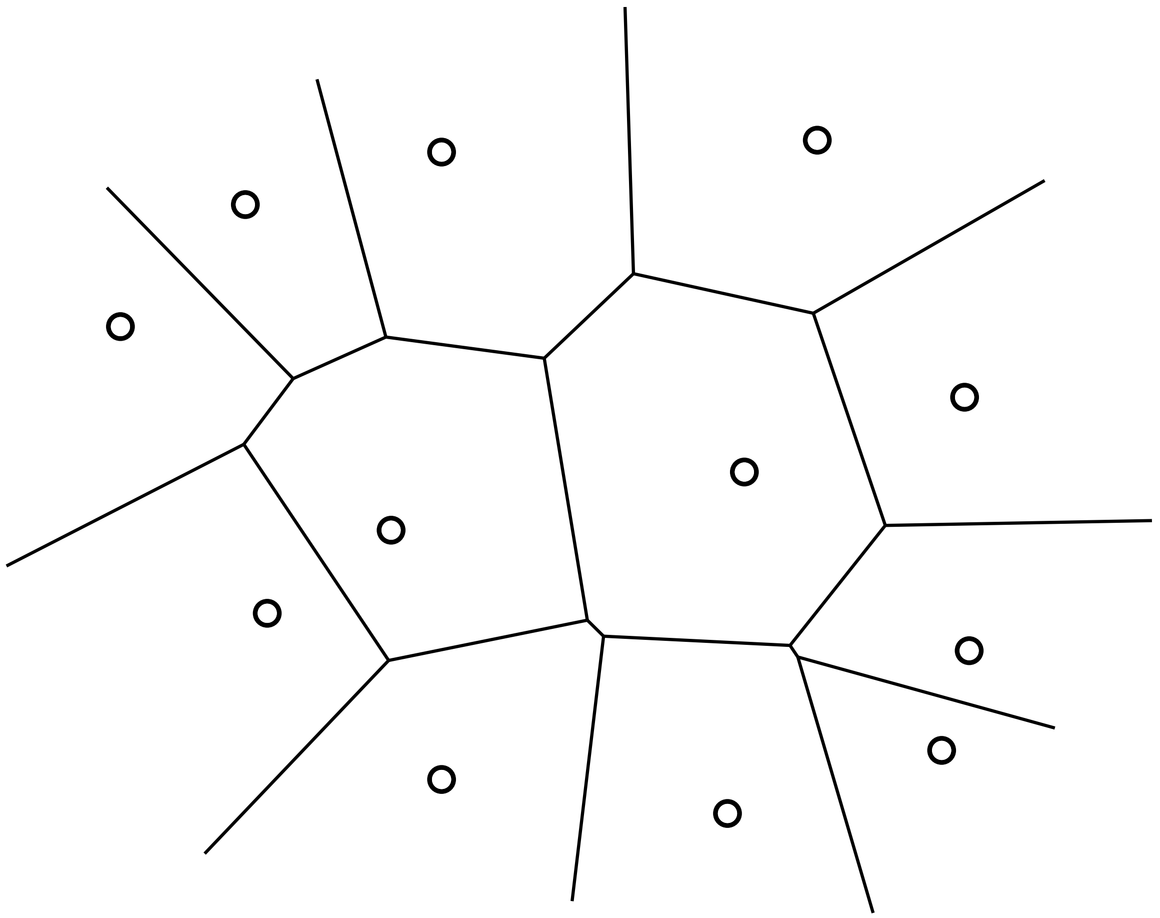


Introduction

Computational geometry is about algorithmic problems that involve geometric objects such as points, line segments, lines, polygons, circles, planes, polyhedra, . . .

Some problems:

- Closest Pair
- Line segment intersection
- Determining visibility
- Guarding an art gallery
- Triangulating a polygon
- Motion planning
- Finding the closest post office
- and many more.

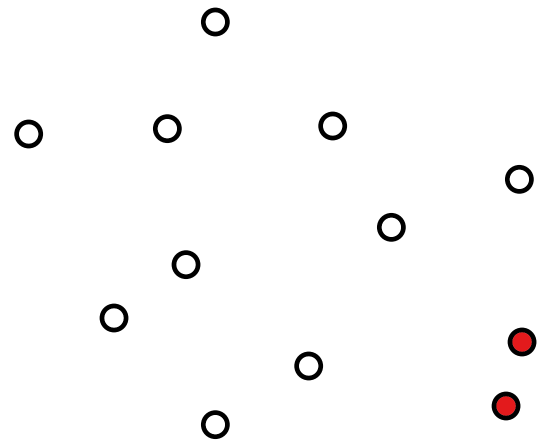


We offer an entire course on computational geometry in the winter term!

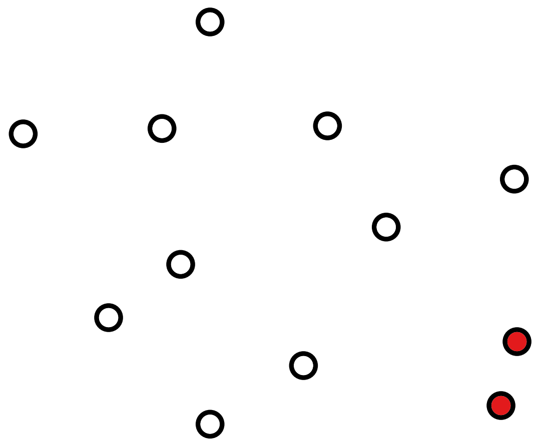
Closest Pair

Given: (multi-)set of points $P \subseteq \mathbb{R}^2$.

Task: Find a pair of distinct elements $p_a, p_b \in P$ such that the Euclidean distance $||p_a - p_b||$ is minimum.



Closest Pair



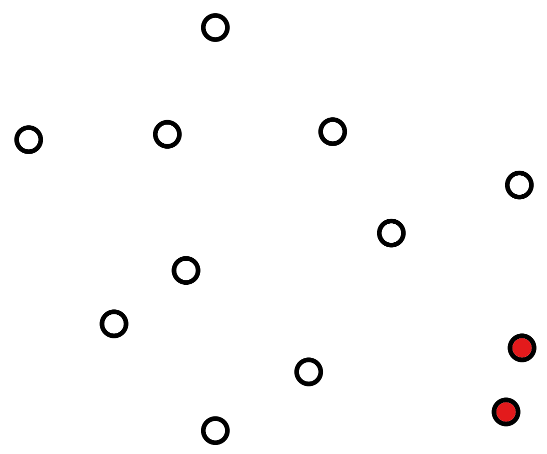
Given: (multi-)set of points $P \subseteq \mathbb{R}^2$.

Task: Find a pair of distinct elements $p_a, p_b \in P$ such that the Euclidean distance $||p_a - p_b||$ is minimum.

Deterministic algorithms:

Brute-force $\mathcal{O}(n^2)$

Closest Pair



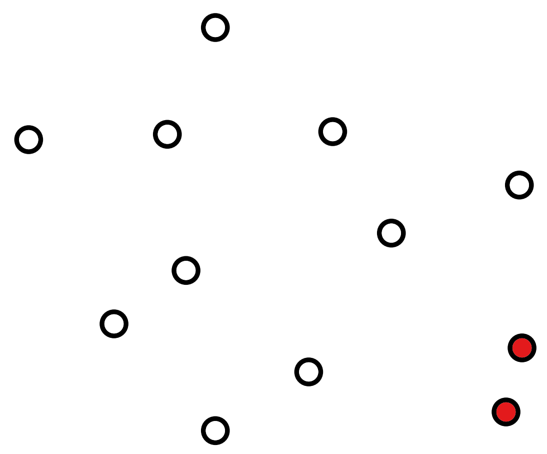
Given: (multi-)set of points $P \subseteq \mathbb{R}^2$.

Task: Find a pair of distinct elements $p_a, p_b \in P$ such that the Euclidean distance $||p_a - p_b||$ is minimum.

Deterministic algorithms:

Brute-force	$\mathcal{O}(n^2)$	
Divide and conquer (recall from ADS)	$\mathcal{O}(n \log n)$	(optimal)

Closest Pair



Given: (multi-)set of points $P \subseteq \mathbb{R}^2$.

Task: Find a pair of distinct elements $p_a, p_b \in P$ such that the Euclidean distance $||p_a - p_b||$ is minimum.

Deterministic algorithms:

Brute-force	$\mathcal{O}(n^2)$	
Divide and conquer (recall from ADS)	$\mathcal{O}(n \log n)$	(optimal)

Randomized algorithm:

Randomized incremental construction	$\mathcal{O}(n)$	(expected runtime)

later in this course!

A Randomized Incremental Algorithm for Closest Pair

Define $P_i = \{p_1, p_2, \ldots, p_i\}$ and let δ_i be the dis… …osest pair in P_i.

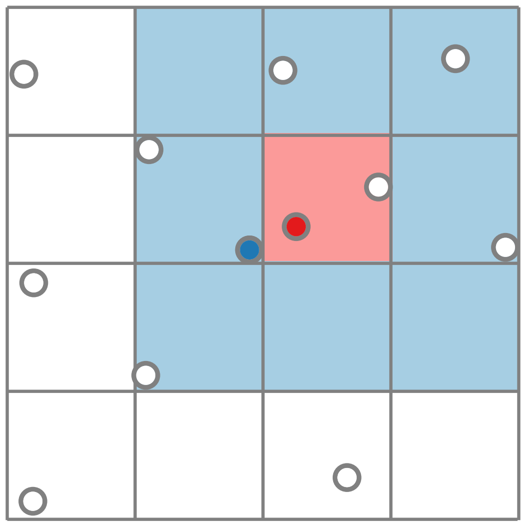
Idea: $\delta_2 = ||p_1, p_2||$. Compute $\delta_3, \delta_4, \ldots, \delta$… …g the points iteratively.

Suppose we have already determin…

Consider a square grid with c… …$\delta_{i-1} \times \delta_{i-1}$.

Add the point p_i. If δ_i… …n p_i must be part of each closest pa…

Moreover, p… …n the cell of p_i or one of the adjacen…



(simple) exercise

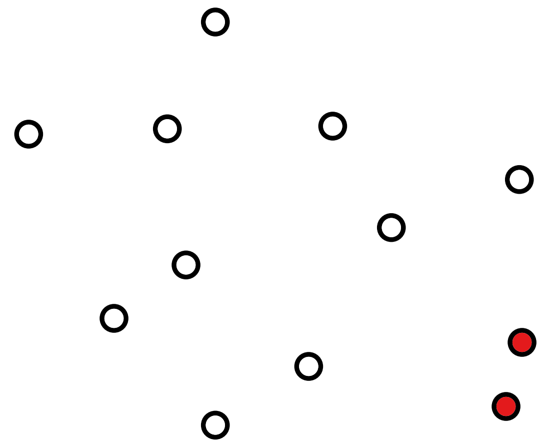
Ea… …e cells contains at most $\mathcal{O}(1)$ points of P_{i-1} ($\Leftarrow$ packing argument).

The co…inates of the cell of p_i can be determined in $\mathcal{O}(1)$ time assuming the floor function can be computed in $\mathcal{O}(1)$ time.

$\Rightarrow$ The test $\delta_i < \delta_{i-1}$ can be performed in $\mathcal{O}(1)$ time assuming P_{i-1} is stored in a suitable dictionary for the nonempty cells (implementable via dynamic perfect hashing).

Upcoming lecture on randomized algorithms

Closest Pair



Given: (multi-)set of points $P \subseteq \mathbb{R}^2$.

Task: Find a pair of distinct elements $p_a, p_b \in P$ such that the Euclidean distance $||p_a - p_b||$ is minimum.

Deterministic algorithms:

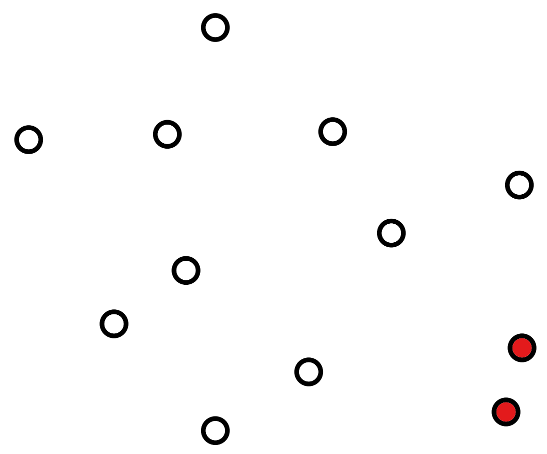
Brute-force	$\mathcal{O}(n^2)$	
Divide and conquer (recall from ADS)	$\mathcal{O}(n \log n)$	(optimal)

Randomized algorithm:

Randomized incremental construction	$\mathcal{O}(n)$	(expected runtime)

later in this course!

Closest Pair



Given: (multi-)set of points $P \subseteq \mathbb{R}^2$.

Task: Find a pair of distinct elements $p_a, p_b \in P$ such that the Euclidean distance $||p_a - p_b||$ is minimum.

Deterministic algorithms:

Brute-force	$\mathcal{O}(n^2)$	
Divide and conquer (recall from ADS)	$\mathcal{O}(n \log n)$	(optimal)
Sweep line	$\mathcal{O}(n \log n)$	(optimal) **now!**

Randomized algorithm:

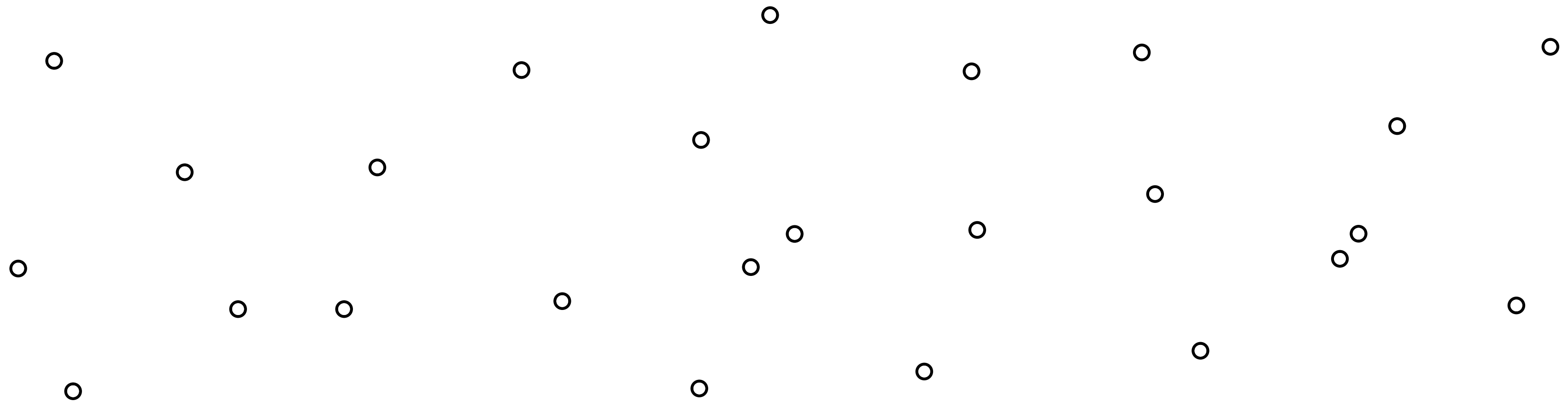
Randomized incremental construction	$\mathcal{O}(n)$	(expected runtime)

later in this course!

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

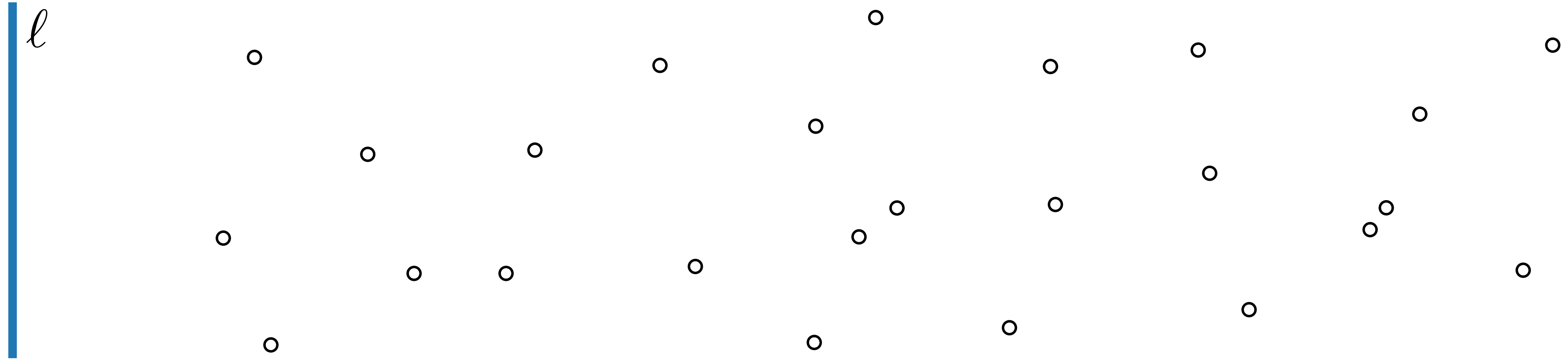
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

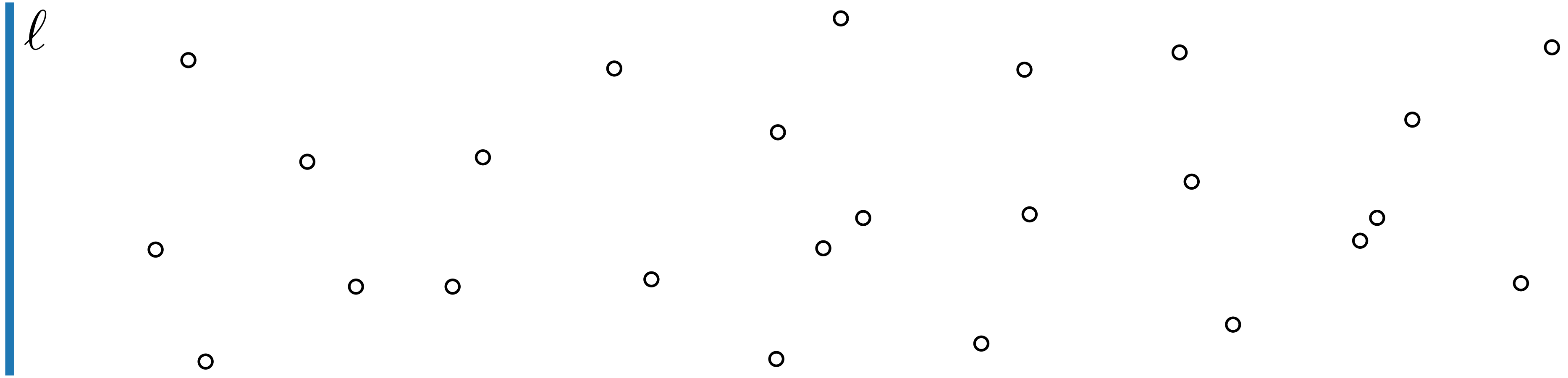
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

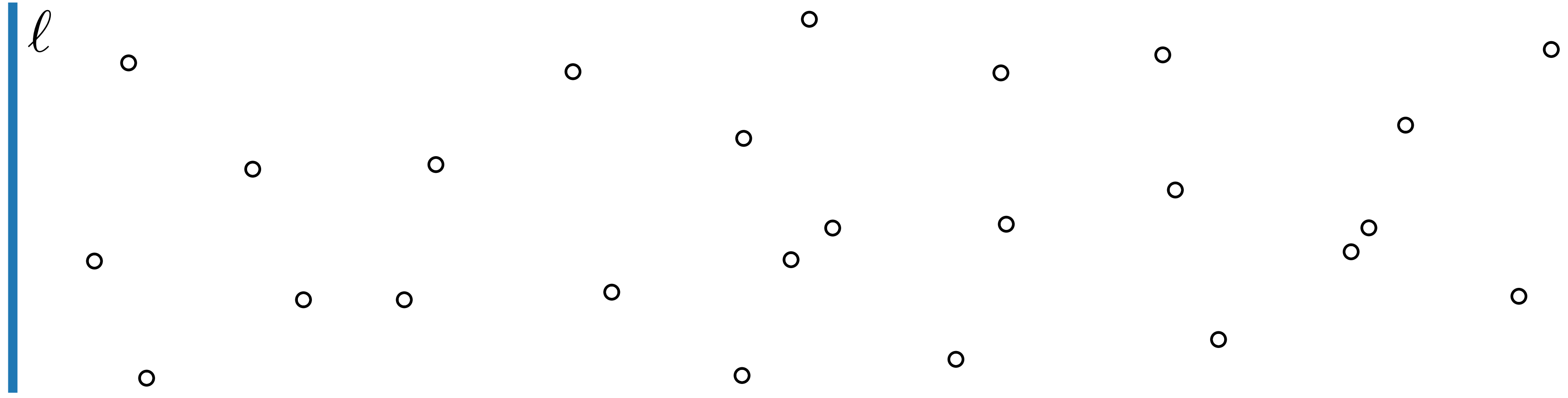
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

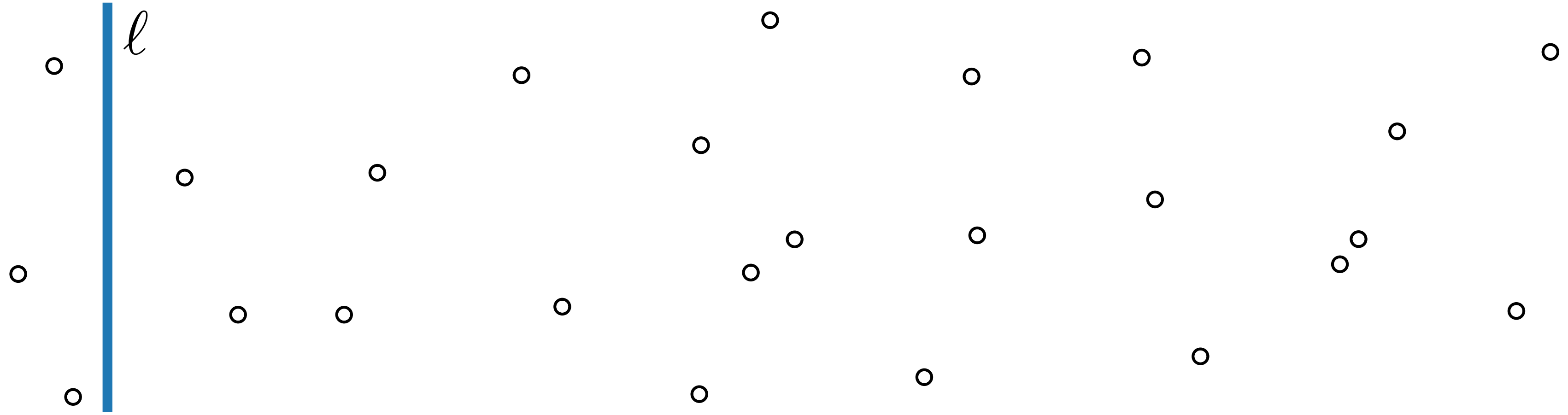
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

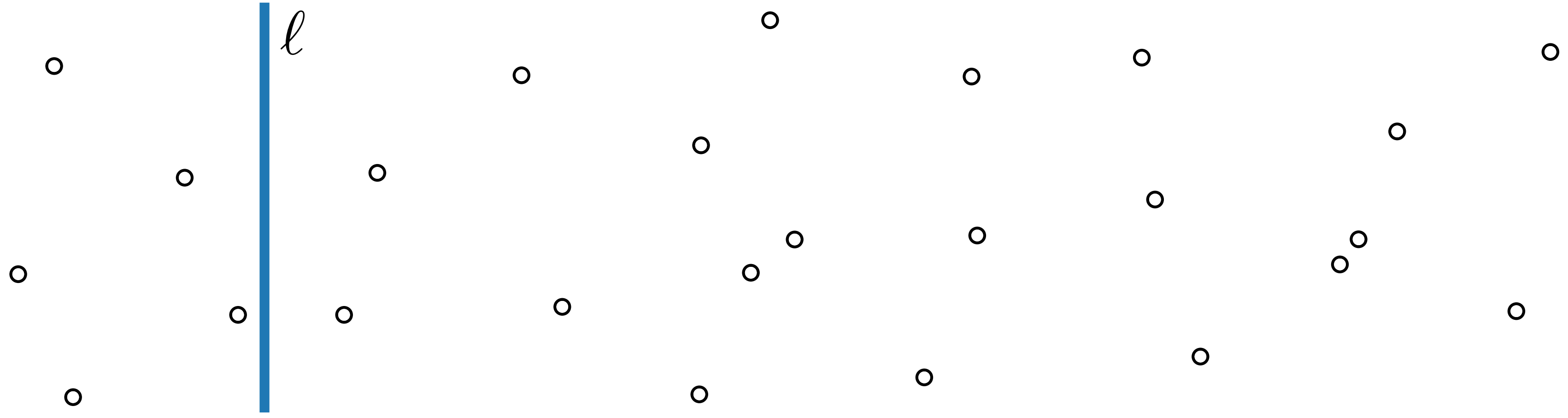
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

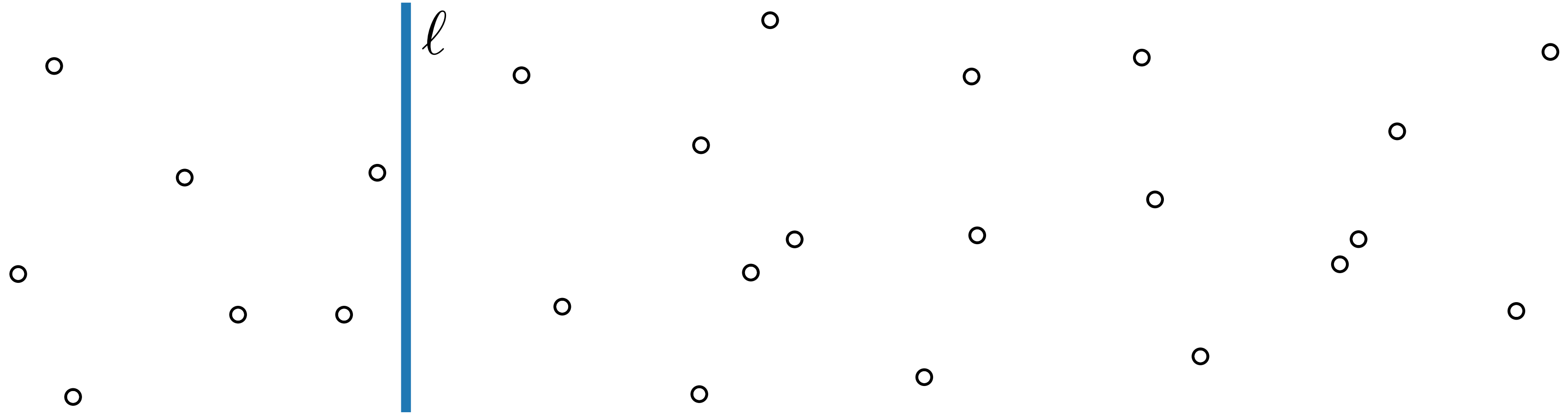
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

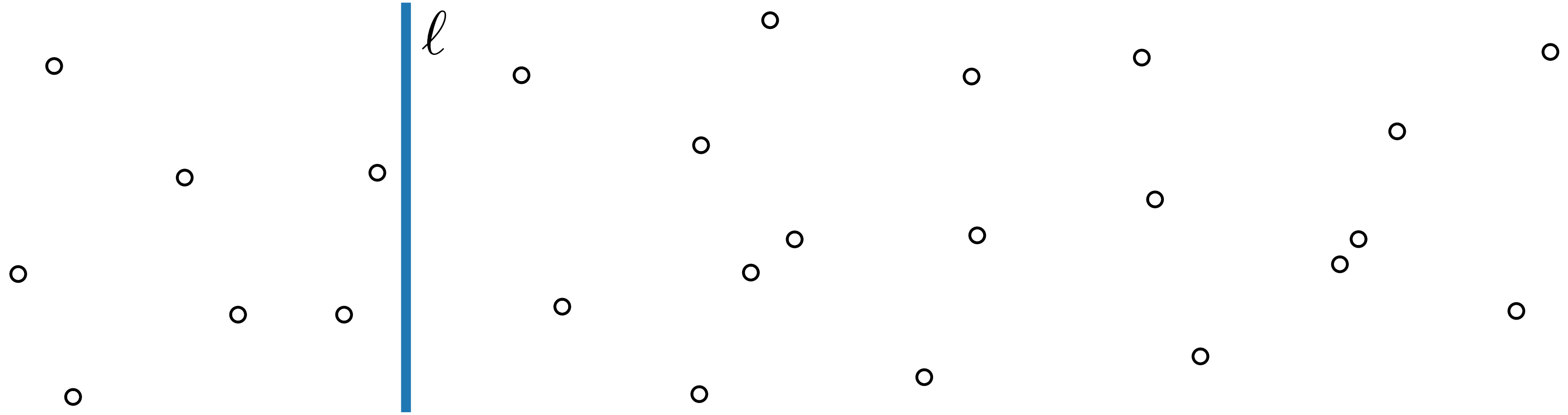
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).

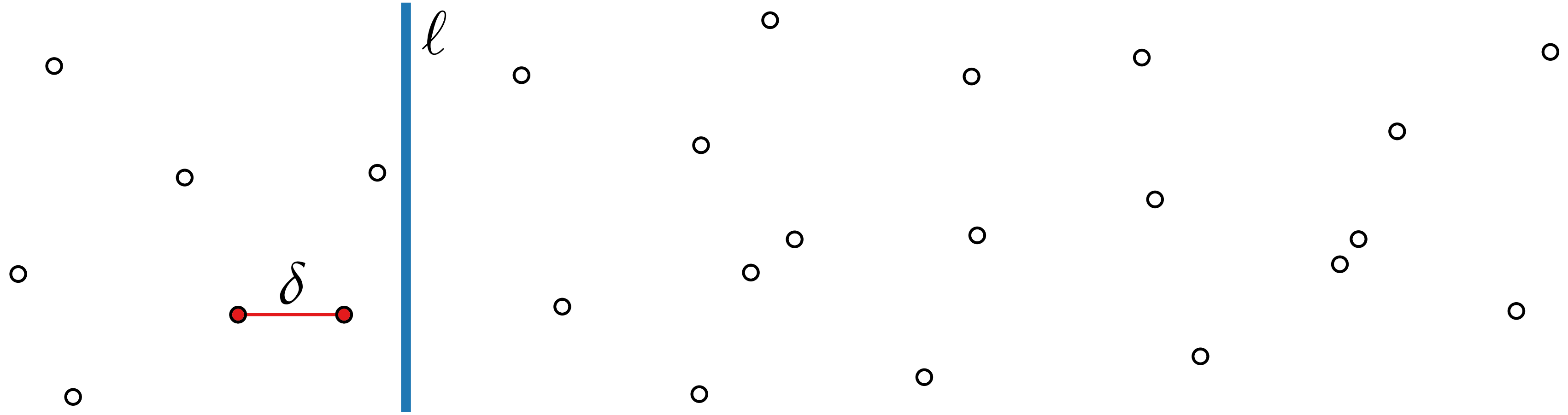


Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).

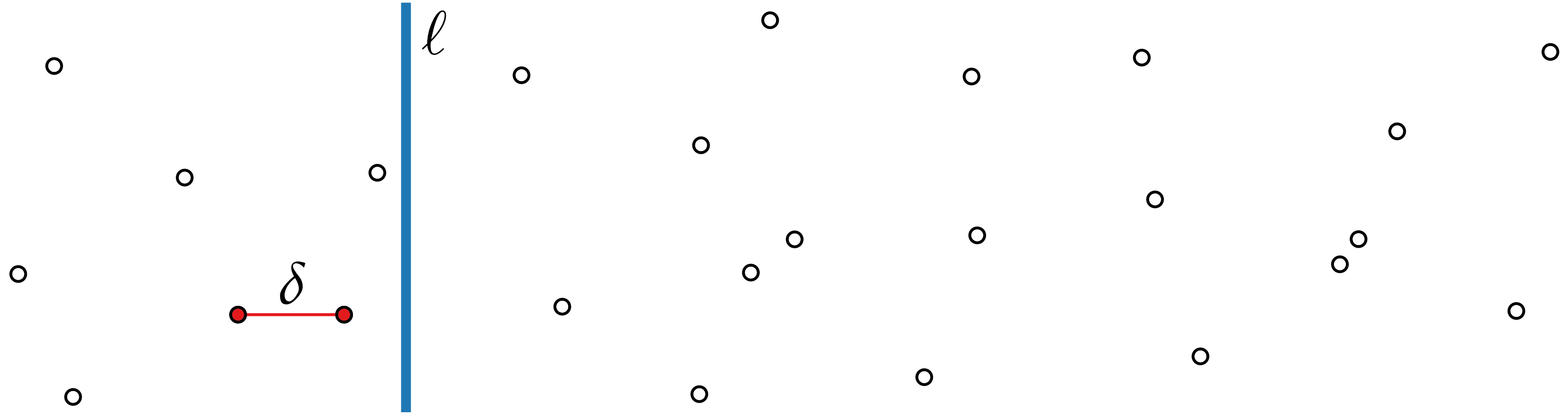


Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

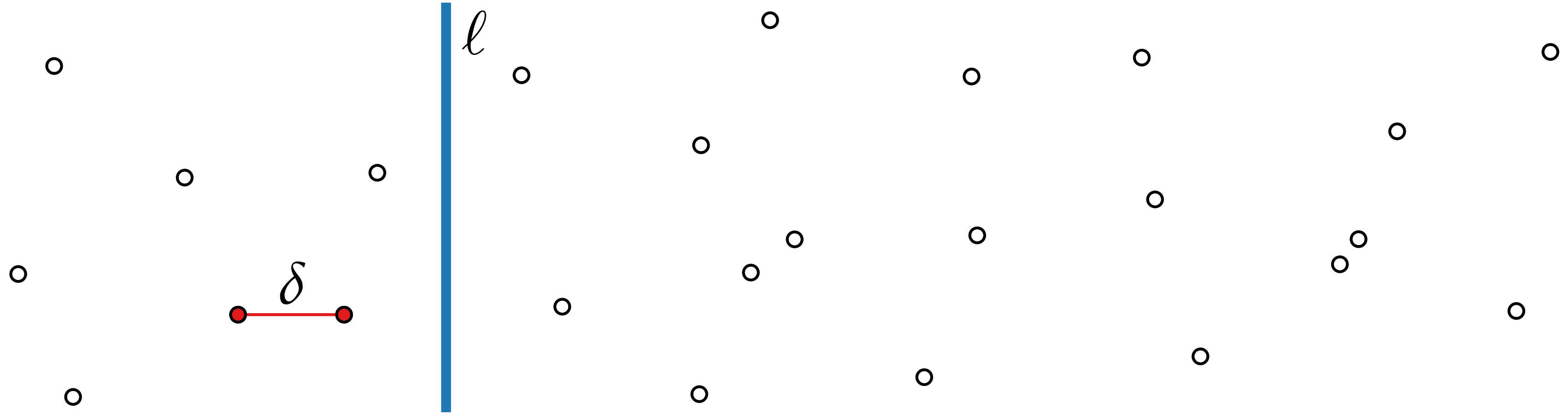
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.

A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

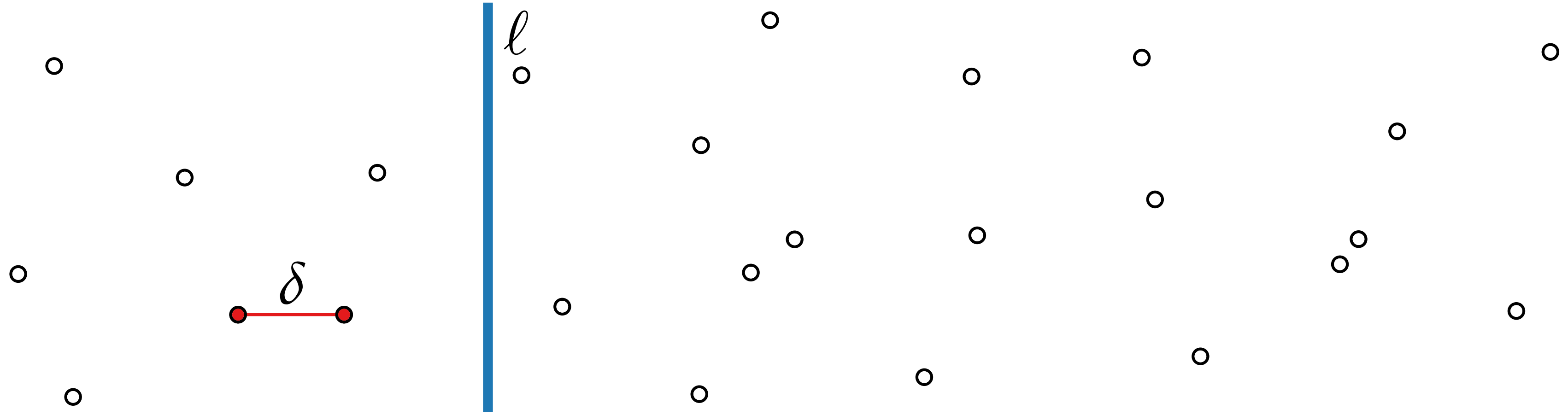
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

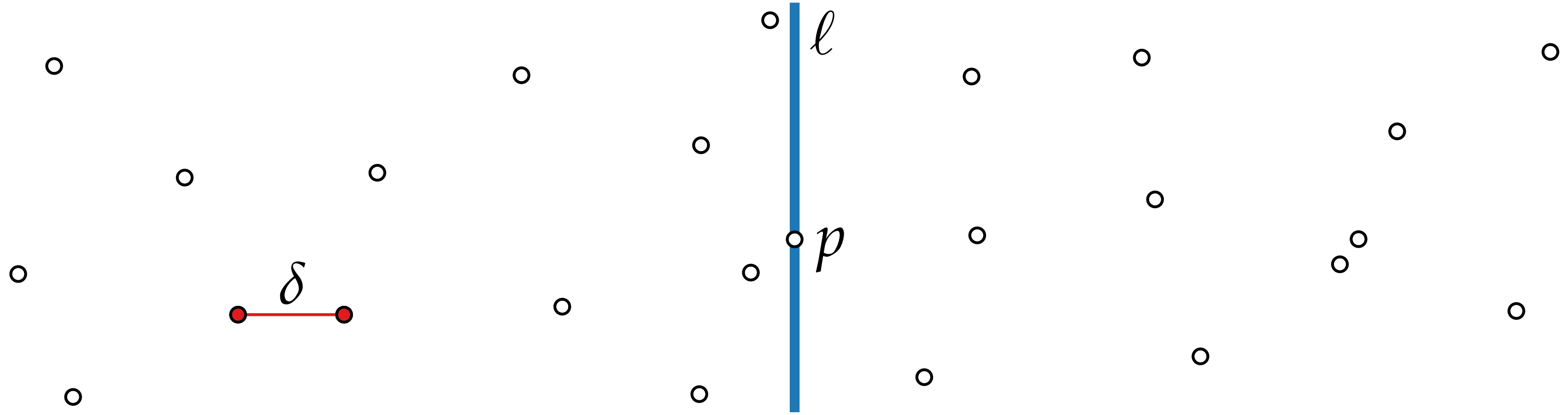
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.

A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

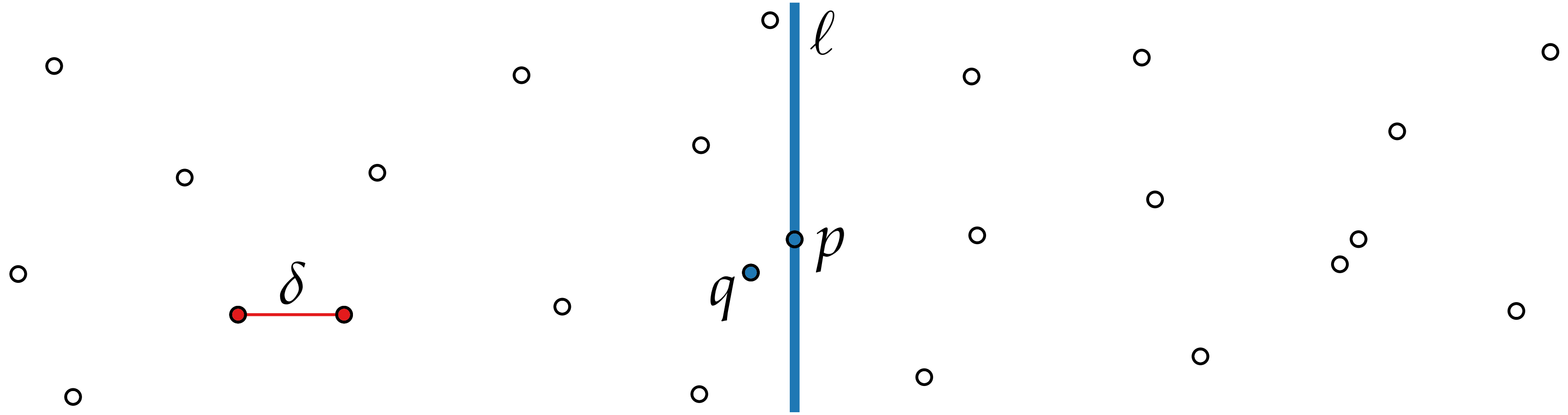
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

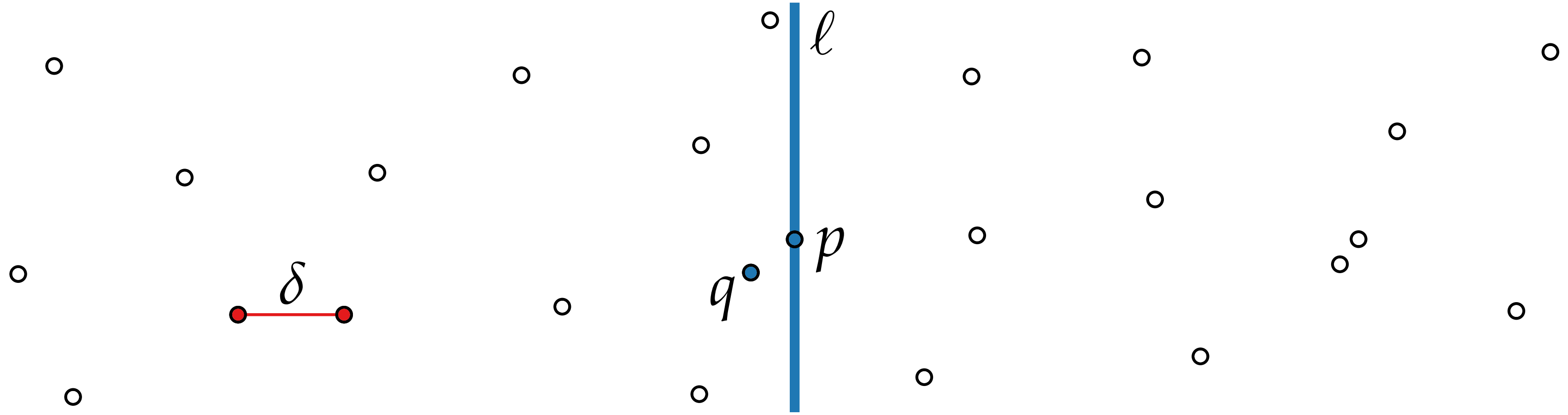
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q

A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

Observations:

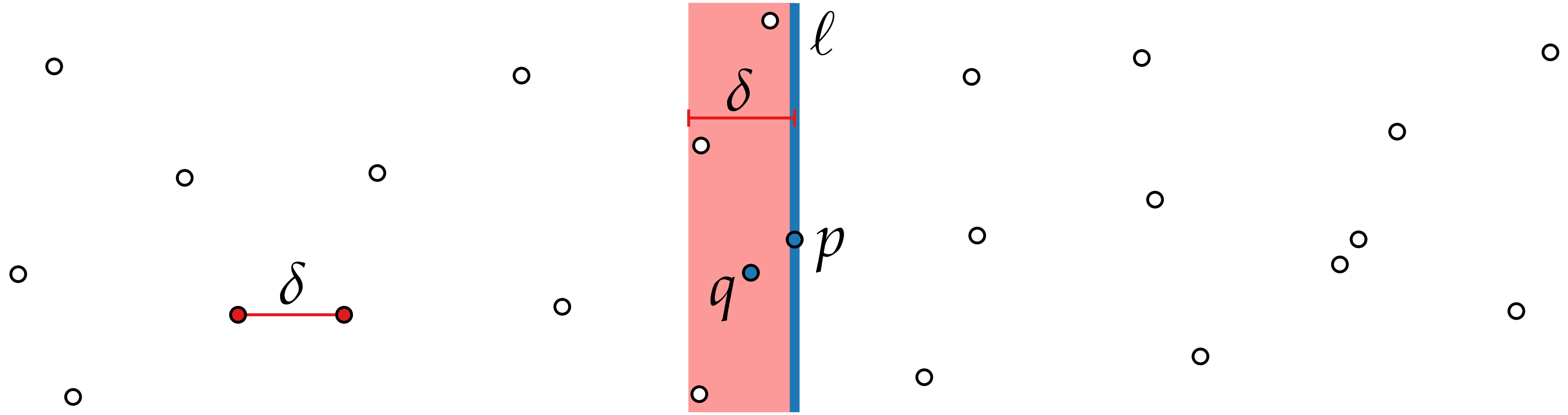
- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q

What do we know about the location of q?

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

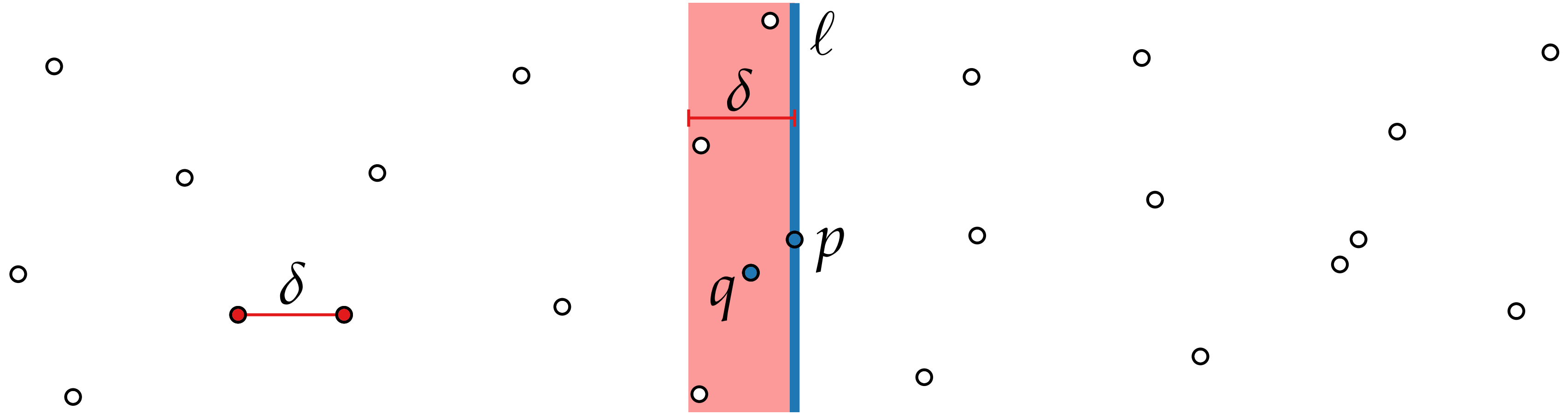
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q with distance $< \delta$ to ℓ.

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

Observations:

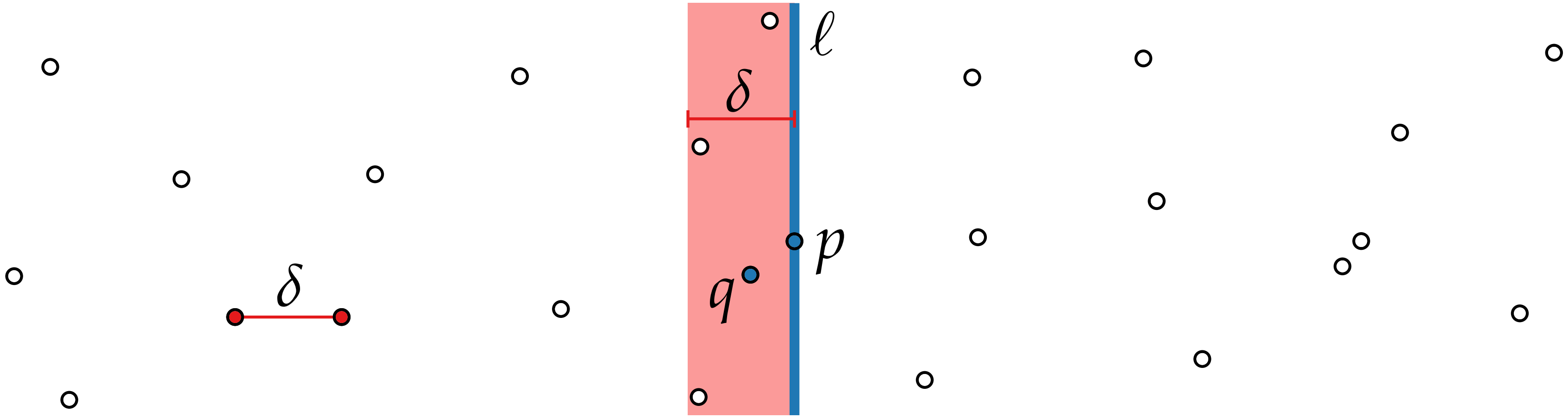
- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q with distance $< \delta$ to ℓ.

How many points can be in this vertical slab?

A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

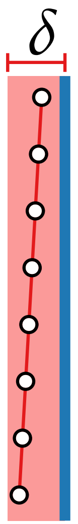
Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

Observations:

- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q with distance $< \delta$ to ℓ.



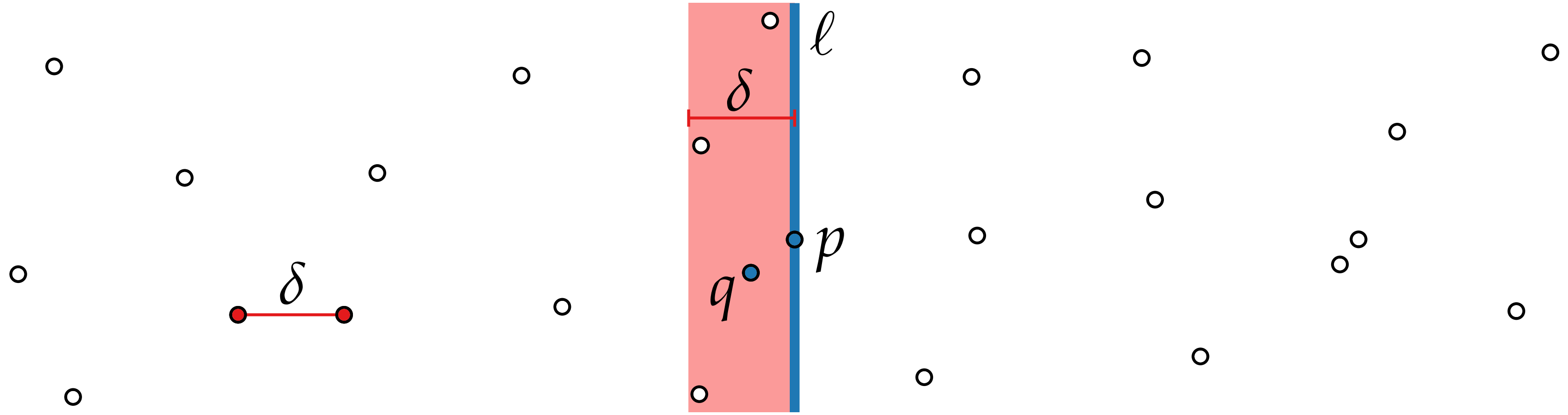
How many points can be in this vertical slab?

All of them!

A Sweep Line Approach for CLOSEST PAIR

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

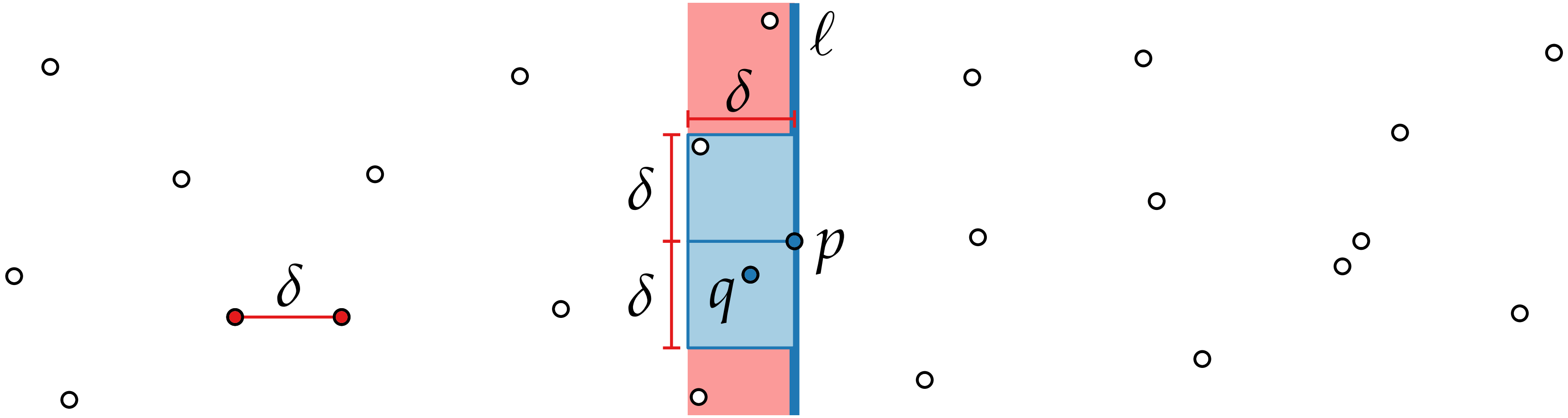
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q with distance $< \delta$ to ℓ.

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

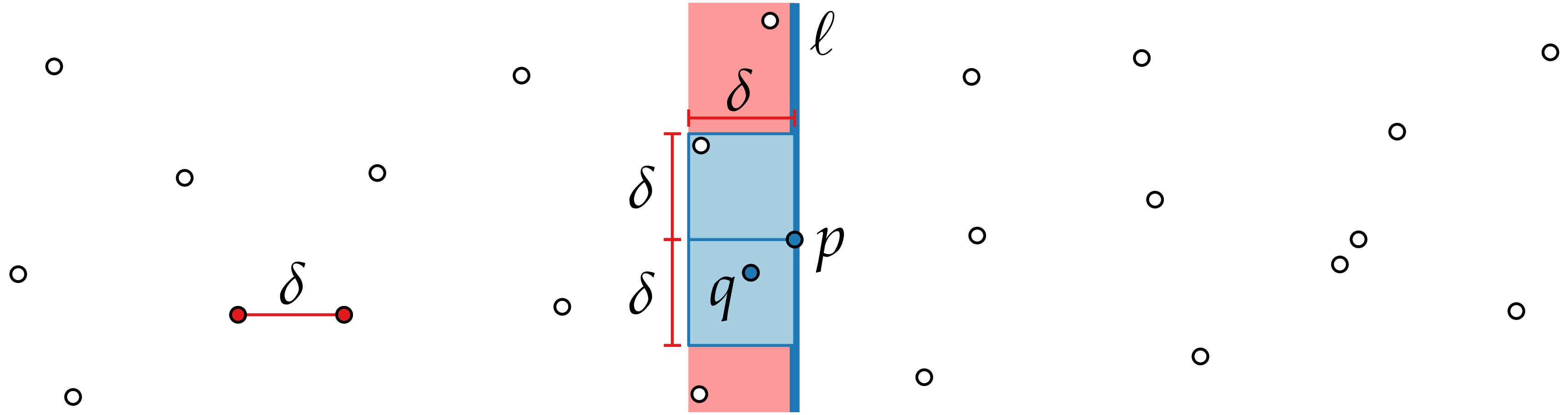
Observations:

- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q with distance $< \delta$ to ℓ.
- q needs to be located in a $\delta \times 2\delta$ rectangle R to the left of p.

A Sweep Line Approach for Closest Pair

Assumption: The points in P have pairwise distinct x-coordinates.

Idea: Sweep the plane from left to right with a vertical line ℓ (the **sweep line**).



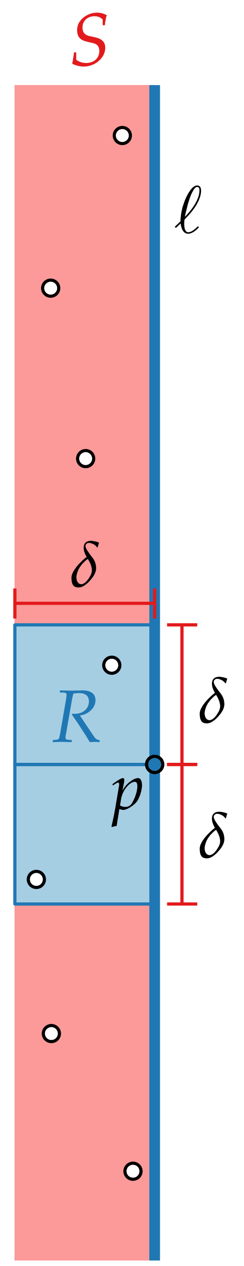
Invariant: a closest pair of the points to the left of ℓ and its distance δ is already known.

Observations:

- This partial solution can only change when ℓ sweeps a point p of P.
- Each new closest pair consists of p and a point q with distance $< \delta$ to ℓ.
- q needs to be located in a $\delta \times 2\delta$ rectangle R to the left of p.
- R contains $\mathcal{O}(1)$ points of $P \setminus \{p\}$ since their pairwise distance is $\geq \delta$. (packing argument)

Computing the Points in R Efficiently

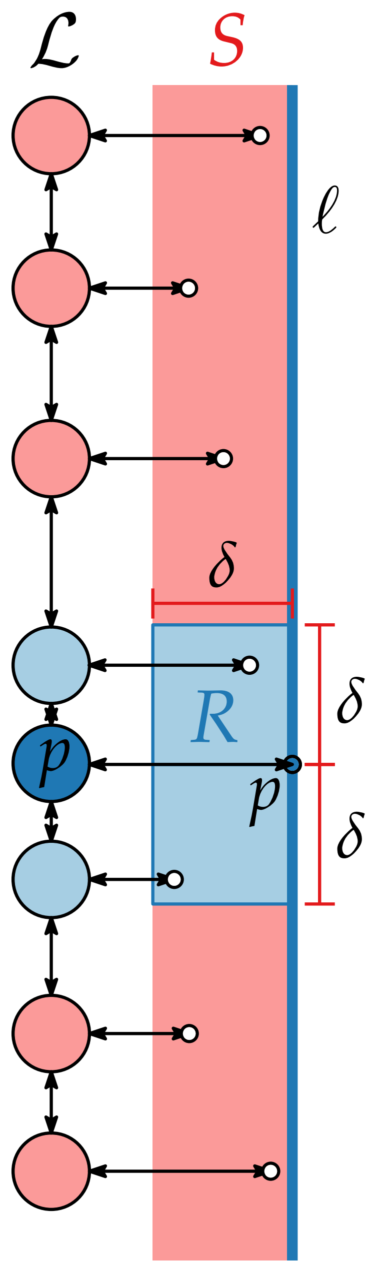
Let S denote the vertical slab of width δ to the left of ℓ.



Computing the Points in R Efficiently

Let S denote the vertical slab of width δ to the left of ℓ.

Assume that the points $P \cap S$ are stored in a **linked list** $\mathcal{L}$ sorted according to their y-coordinates.

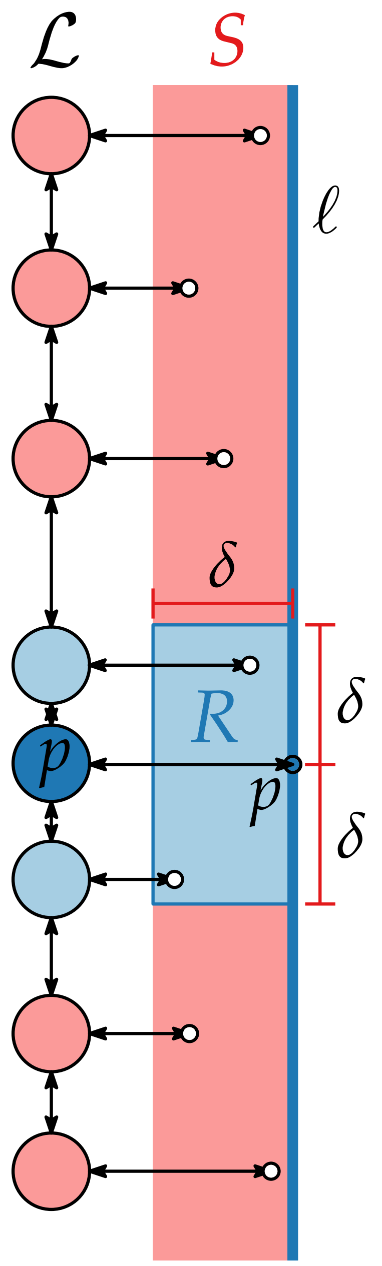


Computing the Points in R Efficiently

Let S denote the vertical slab of width δ to the left of ℓ.

Assume that the points $P \cap S$ are stored in a **linked list** $\mathcal{L}$ sorted according to their y-coordinates.

$\Rightarrow$ Given a pointer to p, we can determine the points in R by searching the interval $[\mathrm{y}(p) - \delta,\ \mathrm{y}(p) + \delta]$. This takes $\mathcal{O}(1)$ time since R contains $\mathcal{O}(1)$ points.



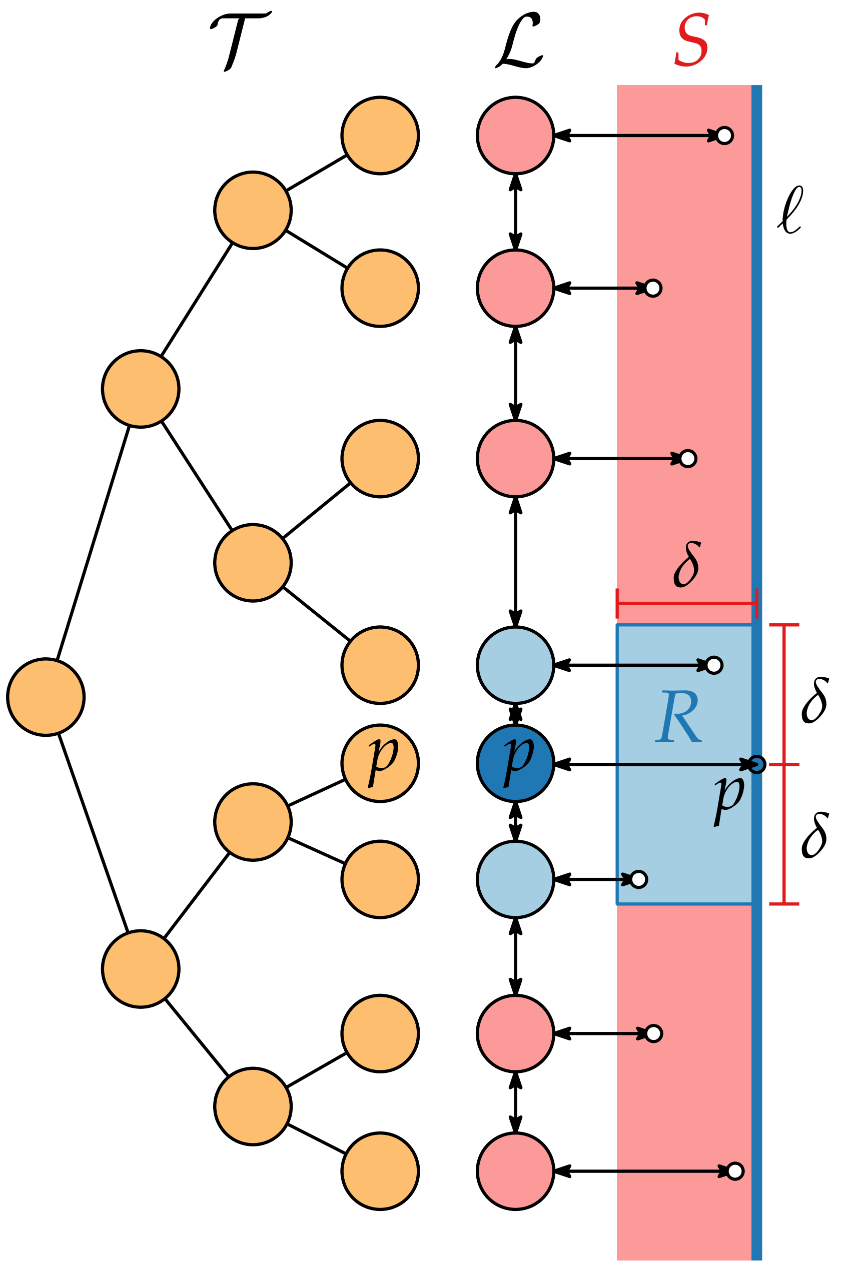
Computing the Points in R Efficiently

Let S denote the vertical slab of width δ to the left of ℓ.

Assume that the points $P \cap S$ are stored in a **linked list** $\mathcal{L}$ sorted according to their y-coordinates.

$\Rightarrow$ Given a pointer to p, we can determine the points in R by searching the interval $[\mathrm{y}(p) - \delta,\ \mathrm{y}(p) + \delta]$. This takes $\mathcal{O}(1)$ time since R contains $\mathcal{O}(1)$ points.

To ensure that $\mathcal{L}$ can be updated efficiently, we additionally store the points $P \cap S$ in a **balanced binary search tree** $\mathcal{T}$ using the y-coordinates as keys.



Computing the Points in R Efficiently

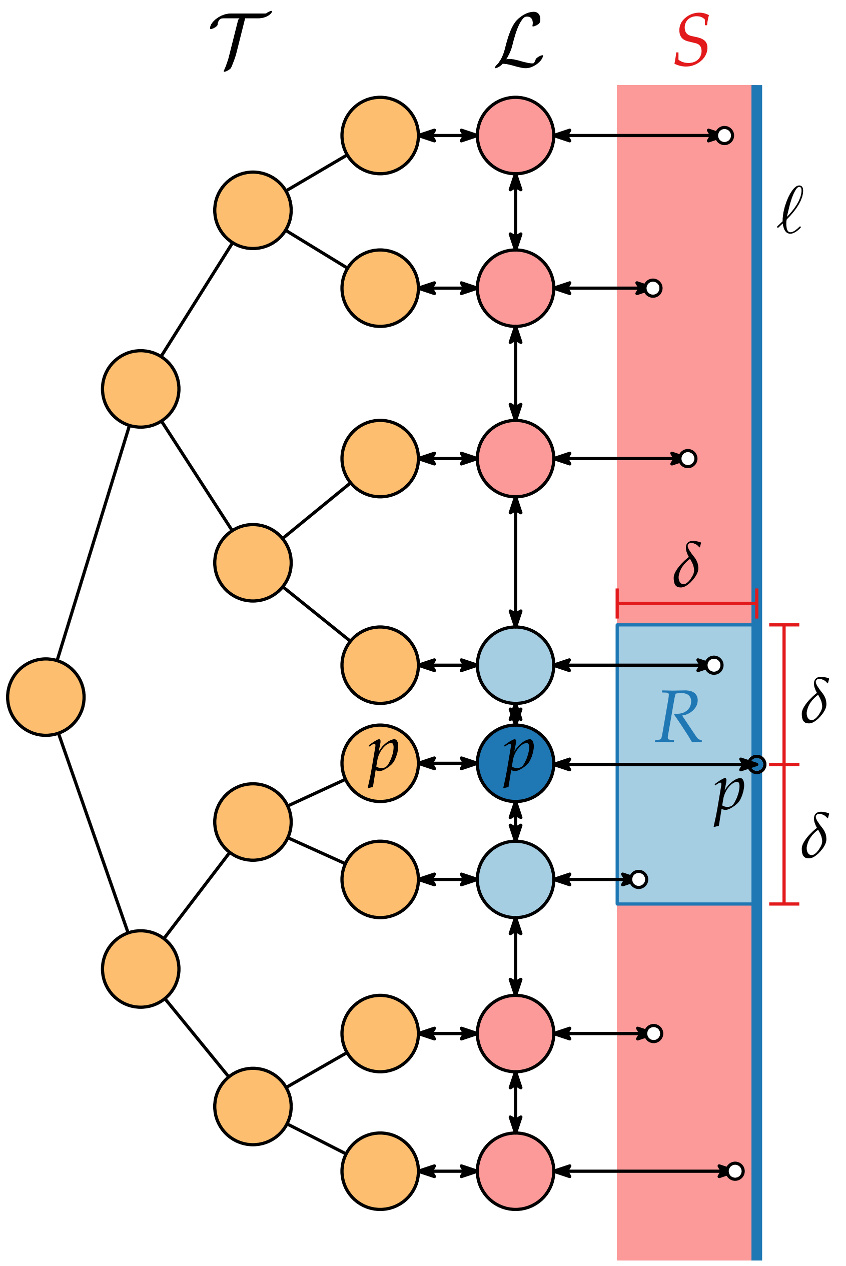
Let S denote the vertical slab of width δ to the left of ℓ.

Assume that the points $P \cap S$ are stored in a **linked list** $\mathcal{L}$ sorted according to their y-coordinates.

$\Rightarrow$ Given a pointer to p, we can determine the points in R by searching the interval $[\mathrm{y}(p) - \delta,\ \mathrm{y}(p) + \delta]$. This takes $\mathcal{O}(1)$ time since R contains $\mathcal{O}(1)$ points.

To ensure that $\mathcal{L}$ can be updated efficiently, we additionally store the points $P \cap S$ in a **balanced binary search tree** $\mathcal{T}$ using the y-coordinates as keys.

The corresponding elements in $\mathcal{L}$ and $\mathcal{T}$ are linked.



Computing the Points in R Efficiently

Let S denote the vertical slab of width δ to the left of ℓ.

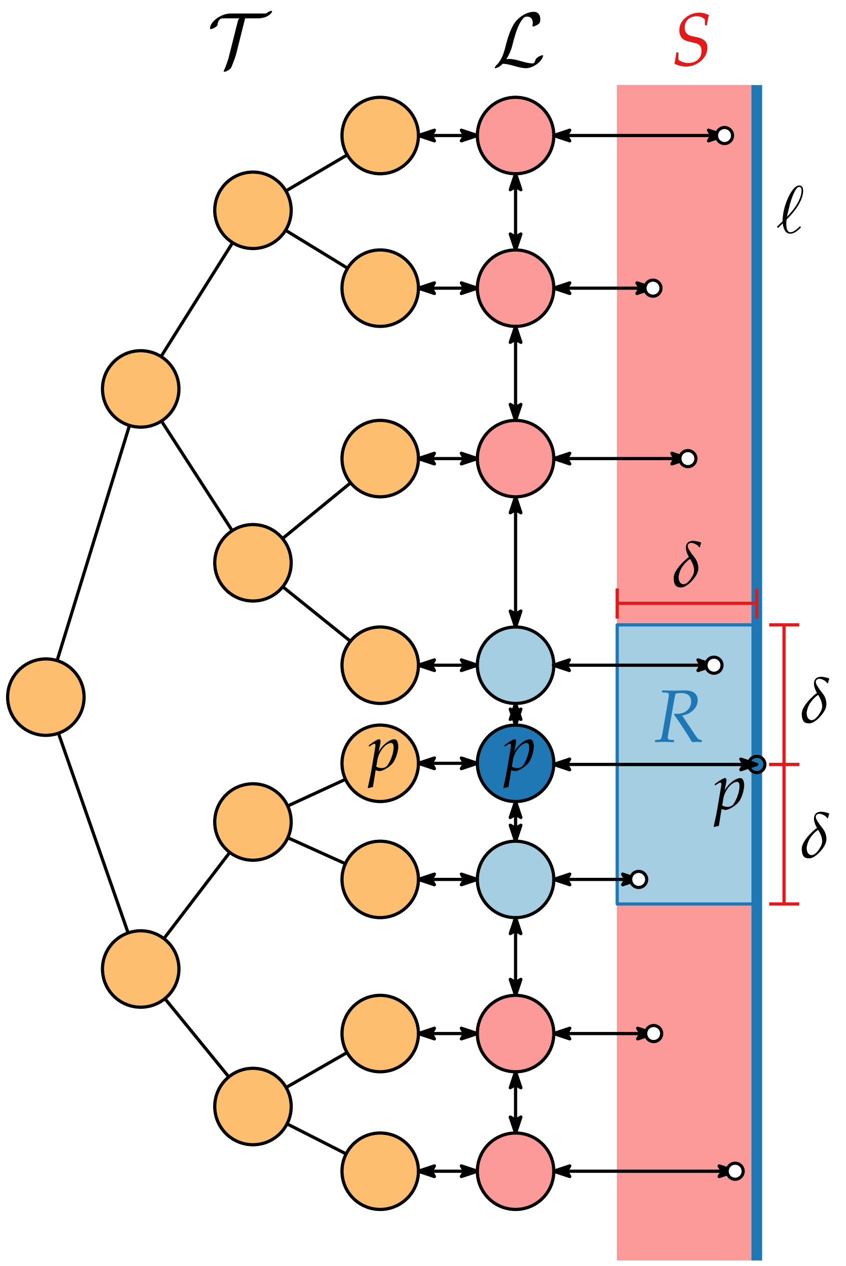
Assume that the points $P \cap S$ are stored in a **linked list** $\mathcal{L}$ sorted according to their y-coordinates.

$\Rightarrow$ Given a pointer to p, we can determine the points in R by searching the interval $[\mathrm{y}(p) - \delta,\ \mathrm{y}(p) + \delta]$. This takes $\mathcal{O}(1)$ time since R contains $\mathcal{O}(1)$ points.

To ensure that $\mathcal{L}$ can be updated efficiently, we additionally store the points $P \cap S$ in a **balanced binary search tree** $\mathcal{T}$ using the y-coordinates as keys.

The corresponding elements in $\mathcal{L}$ and $\mathcal{T}$ are linked.

$\Rightarrow$ when a point is inserted in $\mathcal{T}$ in $\mathcal{O}(\log n)$ time, its according position in $\mathcal{L}$ can be determined in $\mathcal{O}(1)$ time.



Computing the Points in R Efficiently

Let S denote the vertical slab of width δ to the left of ℓ.

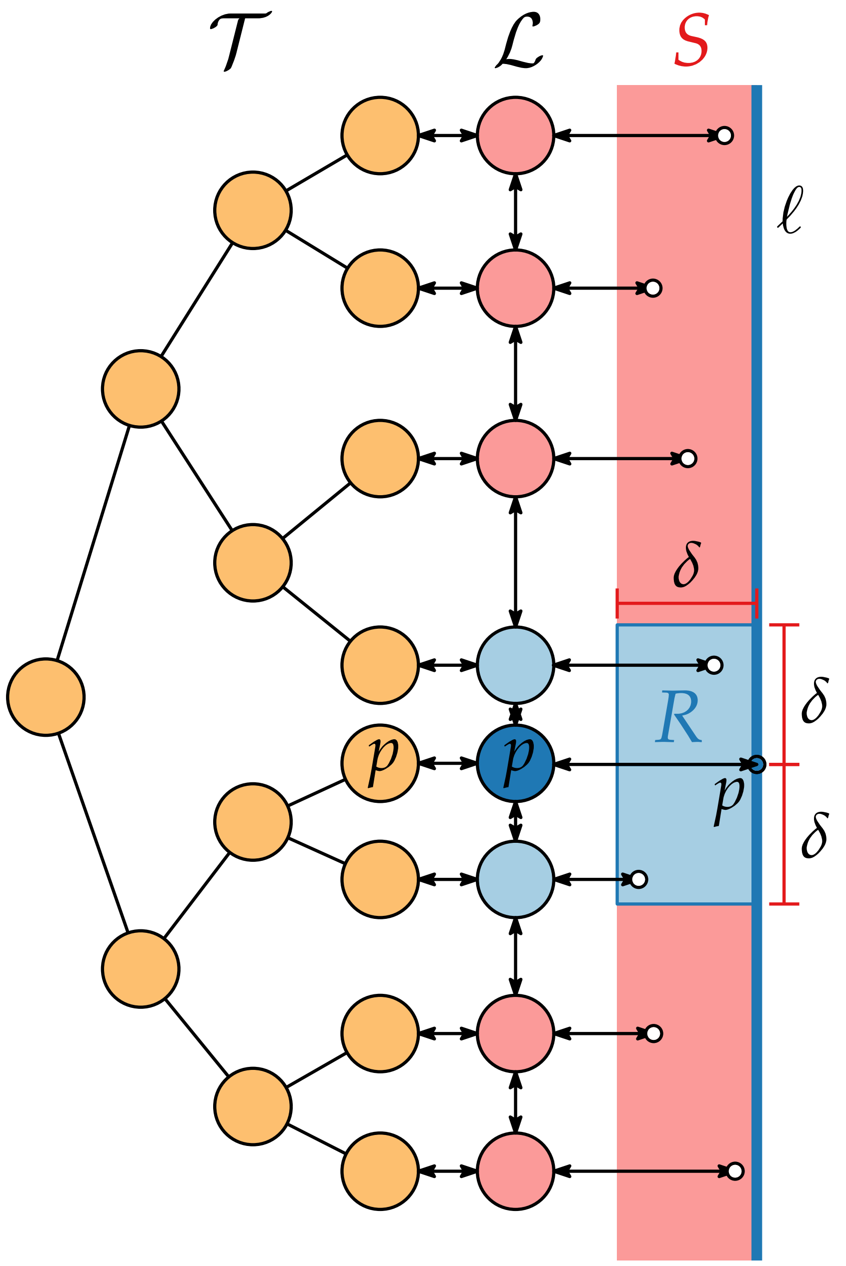
Assume that the points $P \cap S$ are stored in a **linked list** $\mathcal{L}$ sorted according to their y-coordinates.

$\Rightarrow$ Given a pointer to p, we can determine the points in R by searching the interval $[\mathrm{y}(p) - \delta,\ \mathrm{y}(p) + \delta]$. This takes $\mathcal{O}(1)$ time since R contains $\mathcal{O}(1)$ points.

To ensure that $\mathcal{L}$ can be updated efficiently, we additionally store the points $P \cap S$ in a **balanced binary search tree** $\mathcal{T}$ using the y-coordinates as keys.

The corresponding elements in $\mathcal{L}$ and $\mathcal{T}$ are linked.

$\Rightarrow$ when a point is inserted in $\mathcal{T}$ in $\mathcal{O}(\log n)$ time, its according position in $\mathcal{L}$ can be determined in $\mathcal{O}(1)$ time.



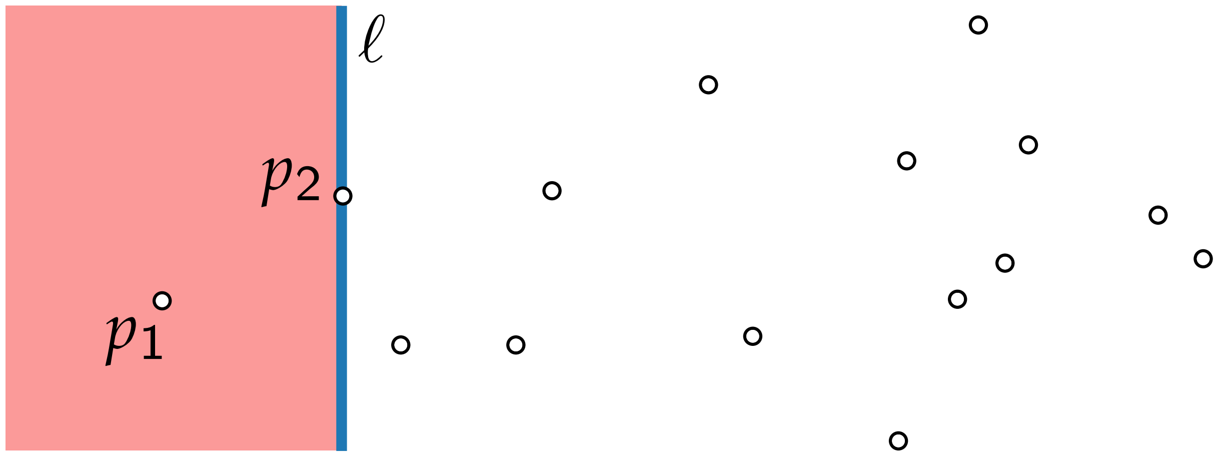
Invariant 2: when we reach a point p, $\mathcal{T}$ and $\mathcal{L}$ contain exactly the points in $P \cap S$.

Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates
$P_{\min} =$ nil // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
    insert p_i into $\mathcal{L}$ and $\mathcal{T}$
    for $p_j \in [\mathrm{y}(p_i) - \delta,\ \mathrm{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
        if $||p_j - p_i|| < \delta$ **do**
            $P_{\min} = \{p_j, p_i\};\ \ \delta = ||p_j - p_i||$
    while $\mathrm{x}(p_k) < \mathrm{x}(p_{i+1}) - \delta$ **do**
        delete p_k from $\mathcal{L}$ and $\mathcal{T}$
        $k = k + 1$
return $P_{\min}$

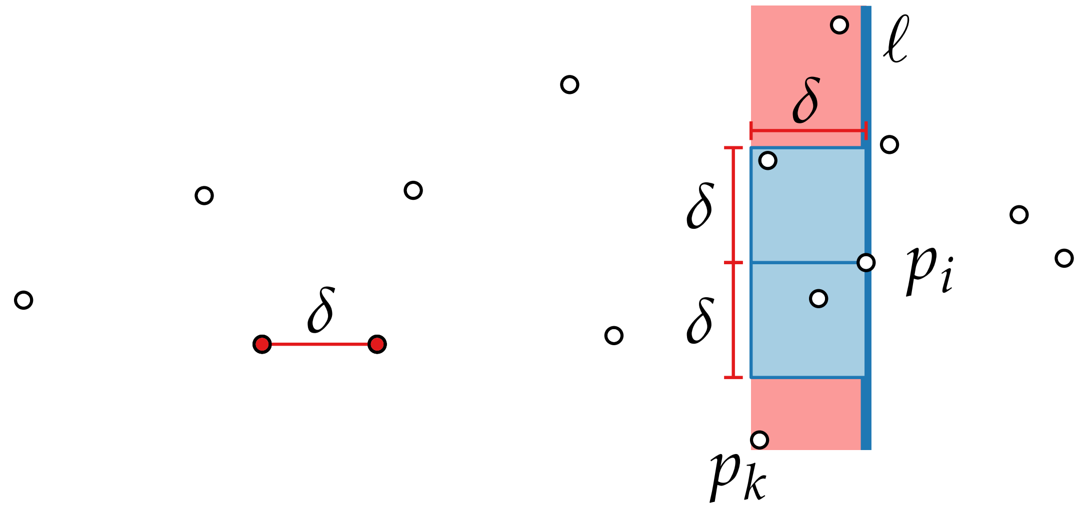
Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates
$P_{\text{min}} =$ nil // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$
  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
    if $||p_j - p_i|| < \delta$ **do**
      $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return P_{min}



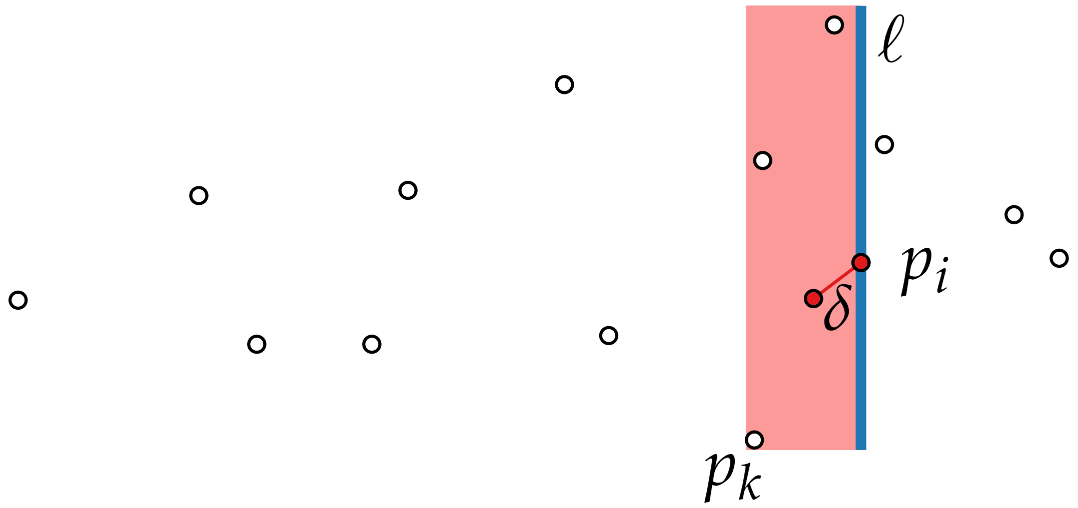
Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates
P_{min} = nil // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$
  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
    if $||p_j - p_i|| < \delta$ **do**
      $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return P_{min}



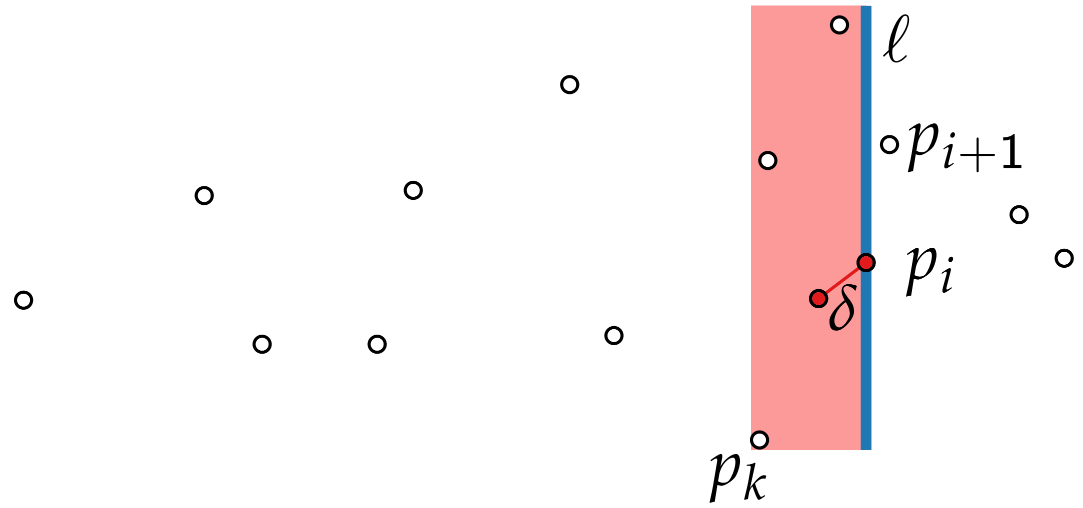
Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates
$P_{\text{min}} = \text{nil}$ // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$
  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
    if $||p_j - p_i|| < \delta$ **do**
      $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return P_{min}



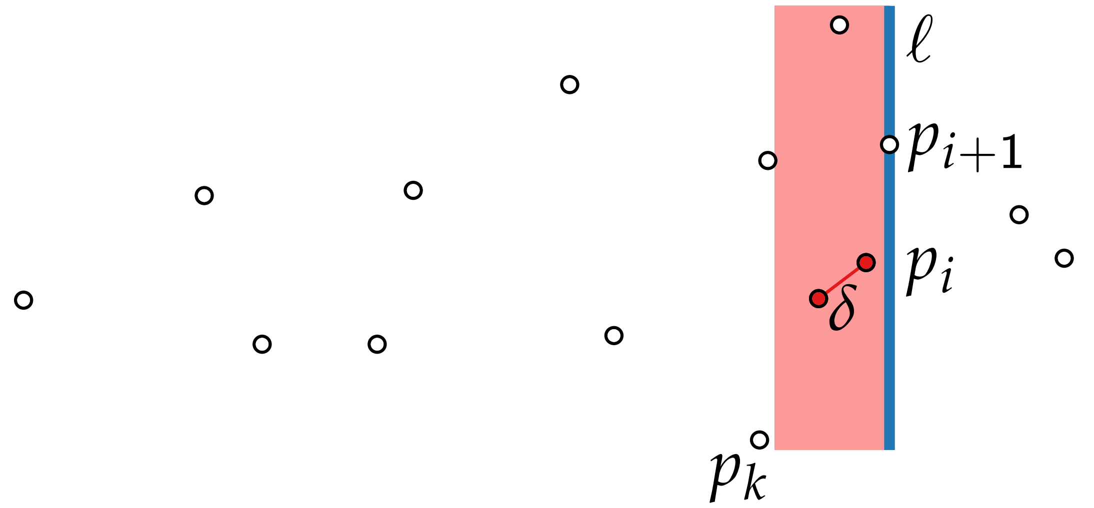
Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates
P_{min} = nil // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$
  for $p_j \in [\mathrm{y}(p_i) - \delta,\ \mathrm{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
    if $||p_j - p_i|| < \delta$ **do**
      $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
  while $\mathrm{x}(p_k) < \mathrm{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return P_{min}



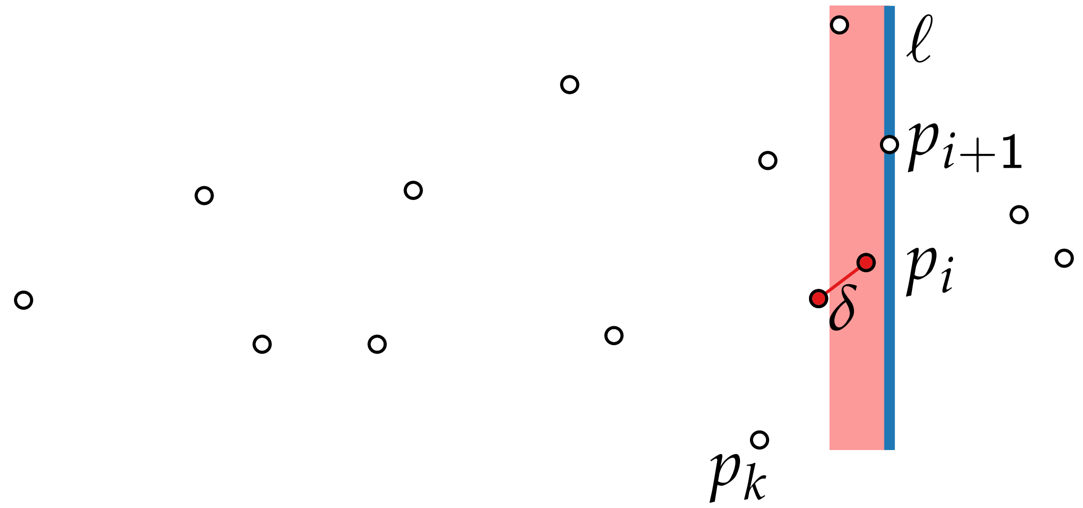
Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates
$P_{\min} = \text{nil}$ // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$
  for $p_j \in [\mathrm{y}(p_i) - \delta,\ \mathrm{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
    if $||p_j - p_i|| < \delta$ **do**
      $P_{\min} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
  while $\mathrm{x}(p_k) < \mathrm{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return $P_{\min}$



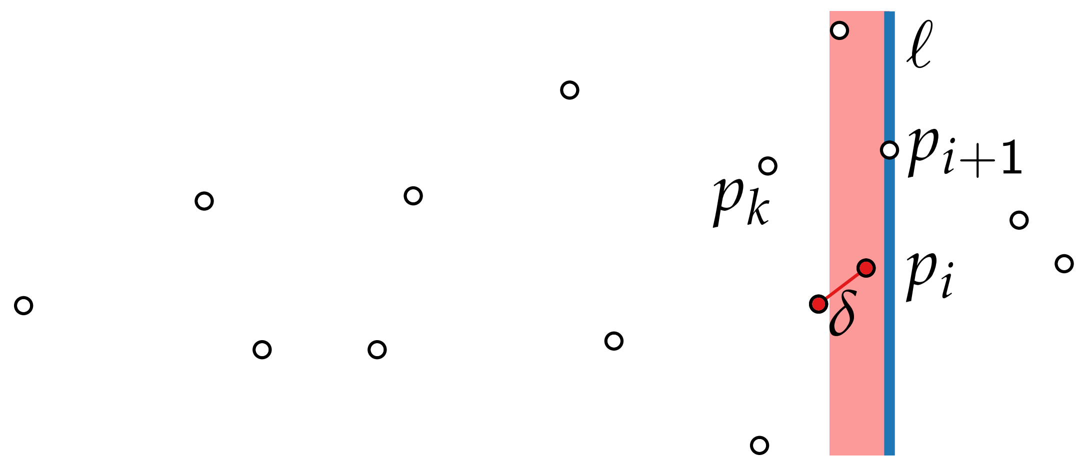
Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates
$P_{\text{min}} = \text{nil}$ // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$
  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
    if $||p_j - p_i|| < \delta$ **do**
      $P_{\text{min}} = \{p_j, p_i\};\ \ \delta = ||p_j - p_i||$
  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return P_{min}



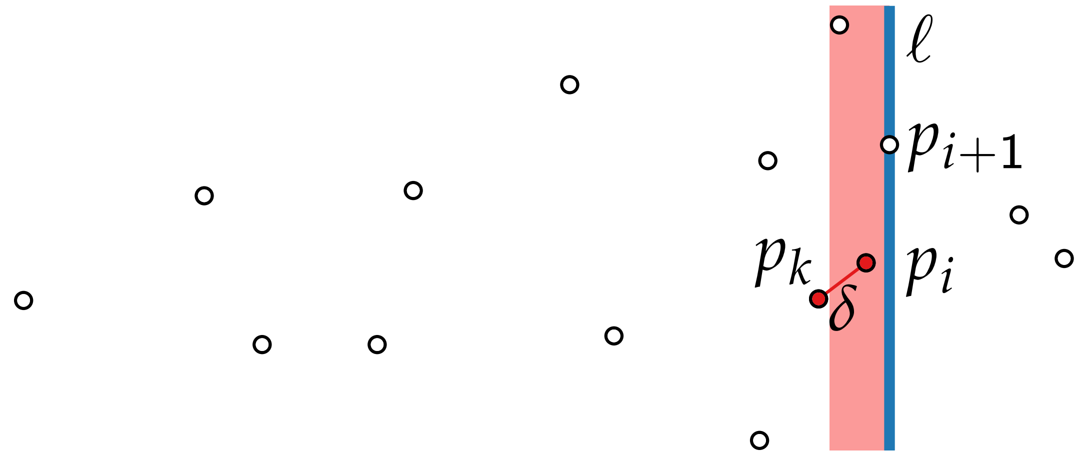
Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates
$P_{\text{min}} = \text{nil}$ // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$
  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
    if $||p_j - p_i|| < \delta$ **do**
      $P_{\text{min}} = \{p_j, p_i\};\quad \delta = ||p_j - p_i||$
  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return P_{min}



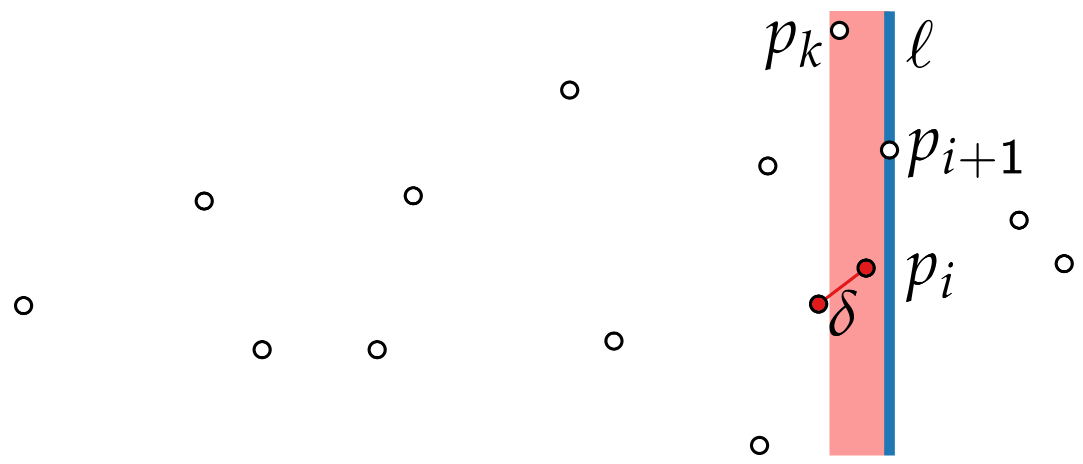
Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates
$P_{\text{min}} = \text{nil}$ // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
    insert p_i into $\mathcal{L}$ and $\mathcal{T}$
    for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
        if $||p_j - p_i|| < \delta$ **do**
            $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
    while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
        delete p_k from $\mathcal{L}$ and $\mathcal{T}$
        $k = k + 1$
return P_{min}



Algorithm

$p_1, p_2, \dots, p_n$ = points of P sorted according to their x-coordinates
P_{min} = nil // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \dots, n$ **do**
    insert p_i into $\mathcal{L}$ and $\mathcal{T}$
    for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**
        if $||p_j - p_i|| < \delta$ **do**
            $P_{\text{min}} = \{p_j, p_i\};\ \ \delta = ||p_j - p_i||$
    while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
        delete p_k from $\mathcal{L}$ and $\mathcal{T}$
        $k = k + 1$
return P_{min}



Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$

$P_{\text{min}} = \text{nil}$ // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

for $i = 2, 3, \ldots, n$ **do**

  insert p_i into $\mathcal{L}$ and $\mathcal{T}$

  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do**

    if $||p_j - p_i|| < \delta$ **do**

      $P_{\text{min}} = \{p_j, p_i\}; \quad \delta = ||p_j - p_i||$

  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**

    delete p_k from $\mathcal{L}$ and $\mathcal{T}$

    $k = k + 1$

return P_{min}

Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$

$P_{\min} =$ nil // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

for $i = 2, 3, \ldots, n$ **do**

  insert p_i into $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$

  for $p_j \in [\mathrm{y}(p_i) - \delta,\ \mathrm{y}(p_i) + \delta] \setminus \{p_i\}$ **do**

    if $||p_j - p_i|| < \delta$ **do**

      $P_{\min} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$

  while $\mathrm{x}(p_k) < \mathrm{x}(p_{i+1}) - \delta$ **do**

    delete p_k from $\mathcal{L}$ and $\mathcal{T}$

    $k = k + 1$

return $P_{\min}$

Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$
$P_{\text{min}} = \text{nil}$ // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
    insert p_i into $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$
    for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do** $\mathcal{O}(1)$
        if $||p_j - p_i|| < \delta$ **do**
            $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
    while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
        delete p_k from $\mathcal{L}$ and $\mathcal{T}$
        $k = k + 1$
return P_{min}

Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$
$P_{\text{min}} =$ nil // current closest pair
$\delta = \infty$ // distance of current closest pair
$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$
initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1
for $i = 2, 3, \ldots, n$ **do**
  insert p_i into $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$
  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do** $\mathcal{O}(1)$
    if $||p_j - p_i|| < \delta$ **do** $\mathcal{O}(1)$
      $P_{\text{min}} = \{p_j, p_i\};\ \ \delta = ||p_j - p_i||$
  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do**
    delete p_k from $\mathcal{L}$ and $\mathcal{T}$
    $k = k + 1$
return P_{min}

Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$

$P_{\text{min}} = \text{nil}$ // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

for $i = 2, 3, \ldots, n$ **do**

  insert p_i into $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$

  for $p_j \in [\mathrm{y}(p_i) - \delta,\ \mathrm{y}(p_i) + \delta] \setminus \{p_i\}$ **do** $\mathcal{O}(1)$

    if $||p_j - p_i|| < \delta$ **do** $\mathcal{O}(1)$

      $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$

  while $\mathrm{x}(p_k) < \mathrm{x}(p_{i+1}) - \delta$ **do**

    delete p_k from $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$

    $k = k + 1$

return P_{min}

Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$

P_{min} = nil // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

```
for i = 2, 3, ..., n do
    insert p_i into L and T                                  O(log n)
    for p_j ∈ [y(p_i) − δ, y(p_i) + δ] \ {p_i} do            O(1)
        if ||p_j − p_i|| < δ do                                         O(1)
            P_min = {p_j, p_i};  δ = ||p_j − p_i||
    while x(p_k) < x(p_{i+1}) − δ do                         O(n) in total
        delete p_k from L and T                              O(log n)
        k = k + 1
return P_min
```

Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates — $\mathcal{O}(n \log n)$

P_{min} = nil // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

for $i = 2, 3, \ldots, n$ **do**

- insert p_i into $\mathcal{L}$ and $\mathcal{T}$ — $\mathcal{O}(\log n)$
- **for** $p_j \in [\mathrm{y}(p_i) - \delta,\ \mathrm{y}(p_i) + \delta] \setminus \{p_i\}$ **do** — $\mathcal{O}(1)$
  - **if** $||p_j - p_i|| < \delta$ **do**
    - $P_{\text{min}} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$
  - $\mathcal{O}(1)$
- **while** $\mathrm{x}(p_k) < \mathrm{x}(p_{i+1}) - \delta$ **do** — $\mathcal{O}(n)$ in total
  - delete p_k from $\mathcal{L}$ and $\mathcal{T}$ — $\mathcal{O}(\log n)$
  - $k = k + 1$
  - $\mathcal{O}(n \log n)$ in total

return P_{min}

Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates — $\mathcal{O}(n \log n)$

$P_{\text{min}} = \text{nil}$ // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

for $i = 2, 3, \ldots, n$ **do** — $\mathcal{O}(n)$

insert p_i into $\mathcal{L}$ and $\mathcal{T}$ — $\mathcal{O}(\log n)$

for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do** — $\mathcal{O}(1)$

if $||p_j - p_i|| < \delta$ **do**

$P_{\text{min}} = \{p_j, p_i\}; \quad \delta = ||p_j - p_i||$

$\mathcal{O}(1)$

while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do** — $\mathcal{O}(n)$ in total

delete p_k from $\mathcal{L}$ and $\mathcal{T}$ — $\mathcal{O}(\log n)$

$k = k + 1$

$\mathcal{O}(n \log n)$ in total

return P_{min}

Algorithm

$p_1, p_2, \ldots, p_n$ = points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$

$P_{\min} = \text{nil}$ // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

for $i = 2, 3, \ldots, n$ **do** $\mathcal{O}(n)$

  insert p_i into $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$

  for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do** $\mathcal{O}(1)$

    if $||p_j - p_i|| < \delta$ **do**

      $P_{\min} = \{p_j, p_i\}$; $\delta = ||p_j - p_i||$

  $\mathcal{O}(1)$

  $\mathcal{O}(n \log n)$

  while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do** $\mathcal{O}(n)$ in total

    delete p_k from $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$

    $k = k + 1$

  $\mathcal{O}(n \log n)$ in total

return $P_{\min}$

Algorithm

$p_1, p_2, \ldots, p_n =$ points of P sorted according to their x-coordinates $\mathcal{O}(n \log n)$

$P_{\text{min}} =$ nil // current closest pair

$\delta = \infty$ // distance of current closest pair

$k = 1$ // index of the left-most point in $\mathcal{L}$ and $\mathcal{T}$

initialize $\mathcal{L}$ and $\mathcal{T}$ with p_1

for $i = 2, 3, \ldots, n$ **do** $\mathcal{O}(n)$

 insert p_i into $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$

 for $p_j \in [\text{y}(p_i) - \delta,\ \text{y}(p_i) + \delta] \setminus \{p_i\}$ **do** $\mathcal{O}(1)$

  if $||p_j - p_i|| < \delta$ **do**

   $P_{\text{min}} = \{p_j, p_i\}; \quad \delta = ||p_j - p_i||$

 $\mathcal{O}(1)$ $\mathcal{O}(n \log n)$

 while $\text{x}(p_k) < \text{x}(p_{i+1}) - \delta$ **do** $\mathcal{O}(n)$ in total

  delete p_k from $\mathcal{L}$ and $\mathcal{T}$ $\mathcal{O}(\log n)$

  $k = k + 1$

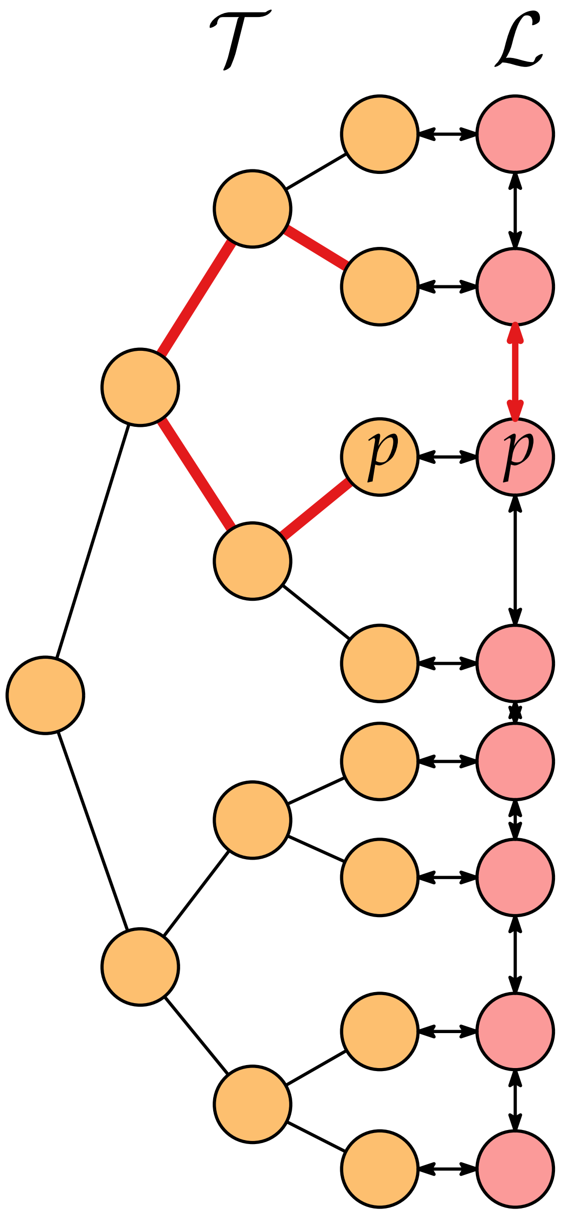
 $\mathcal{O}(n \log n)$ in total

return P_{min}

$\Rightarrow$ **Total runtime:** $\mathcal{O}(n \log n)$

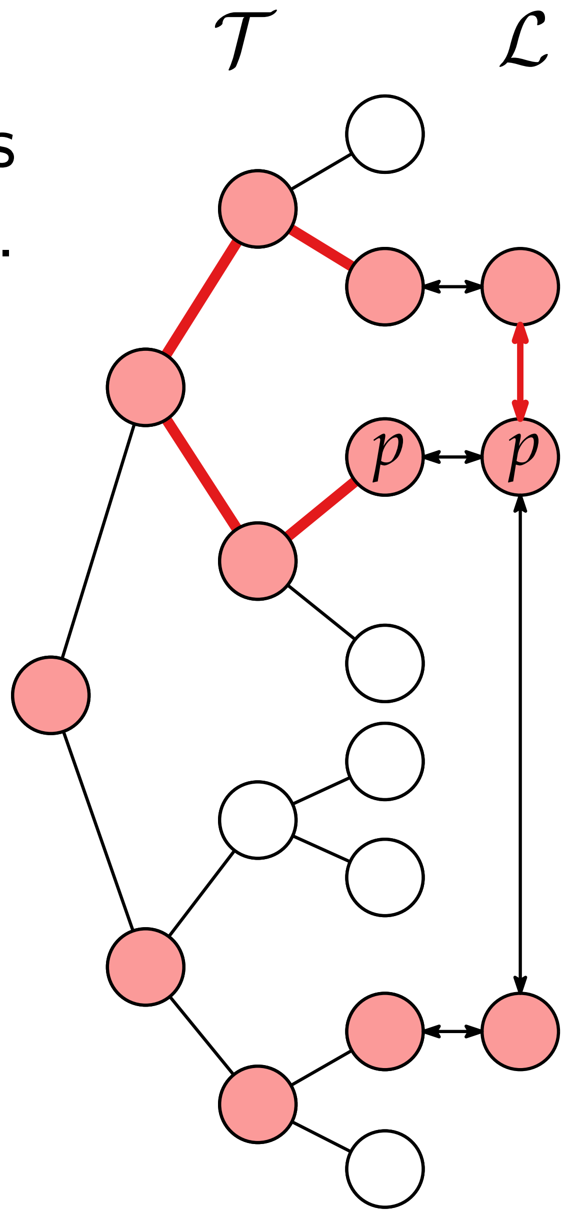
Remarks on the Implementation

- The list $\mathcal{L}$ is actually not necessary: given a point p in $\mathcal{T}$, its neighbors in the ordering can be determined in $\mathcal{O}(\log n)$ time.



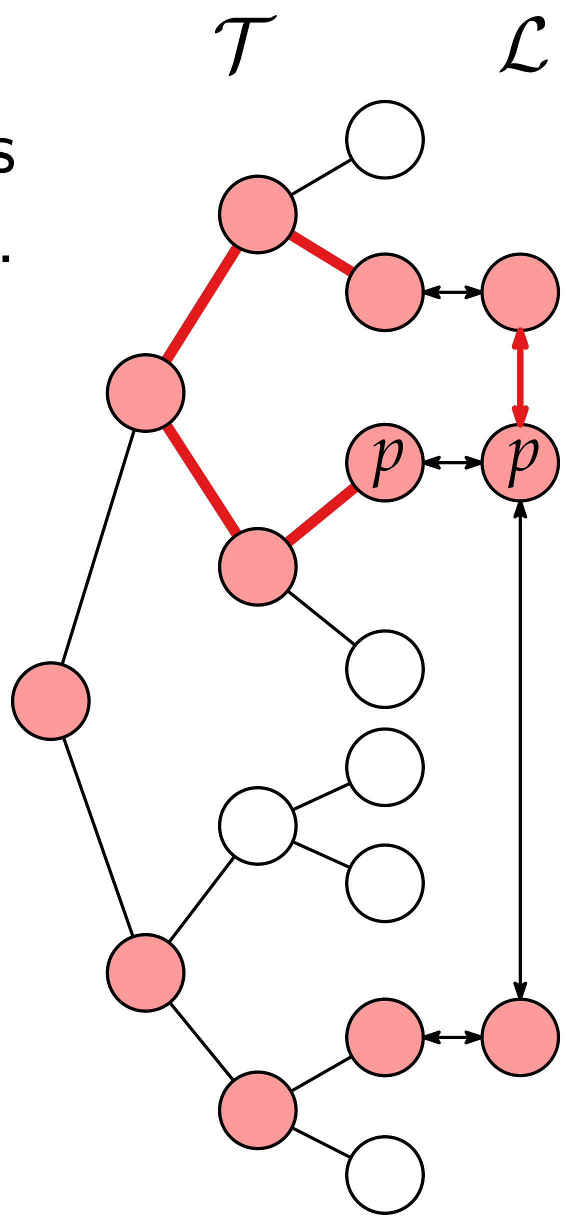
Remarks on the Implementation

- The list $\mathcal{L}$ is actually not necessary: given a point p in $\mathcal{T}$, its neighbors in the ordering can be determined in $\mathcal{O}(\log n)$ time.
- The tree $\mathcal{T}$ does not need to be dynamic! A static tree on all points suffices if each point currently in S and all its ancestors are marked. $\rightarrow$ simple and space efficient (1 bit of extra information / node).



Remarks on the Implementation

- The list $\mathcal{L}$ is actually not necessary: given a point p in $\mathcal{T}$, its neighbors in the ordering can be determined in $\mathcal{O}(\log n)$ time.
- The tree $\mathcal{T}$ does not need to be dynamic! A static tree on all points suffices if each point currently in S and all its ancestors are marked.
  $\rightarrow$ simple and space efficient (1 bit of extra information / node).
- We assumed that the points in P have pairwise distinct x-coordinates. This situation can be established by rotating P or tilting ℓ slightly.
  Simply, visit the points in lexicographical order!



Summary and Discussion

The **sweep line approach** is an important design paradigm (like divide and conquer, prune and search, dynamic programming, greedy, . . .) in computational geometry.

Summary and Discussion

The **sweep line approach** is an important design paradigm (like divide and conquer, prune and search, dynamic programming, greedy, ...) in computational geometry.

Main idea: Sweep the plane with a line ℓ while maintaining two invariants:

- A **partial solution** for the input to the left of ℓ is known.
- The part of the input to the left of ℓ that is still relevant for updating the partial solution is encoded in a suitable data structure (**sweep line status**).

Summary and Discussion

The **sweep line approach** is an important design paradigm (like divide and conquer, prune and search, dynamic programming, greedy, ...) in computational geometry.

Main idea: Sweep the plane with a line ℓ while maintaining two invariants:

- A **partial solution** for the input to the left of ℓ is known.
- The part of the input to the left of ℓ that is still relevant for updating the partial solution is encoded in a suitable data structure (**sweep line status**).

The partial solution and the sweep line status only change at specific positions (**events**) that may be part of the input or arise during the execution of the algorithm.

Summary and Discussion

The **sweep line approach** is an important design paradigm (like divide and conquer, prune and search, dynamic programming, greedy, ...) in computational geometry.

Main idea: Sweep the plane with a line ℓ while maintaining two invariants:

- A **partial solution** for the input to the left of ℓ is known.
- The part of the input to the left of ℓ that is still relevant for updating the partial solution is encoded in a suitable data structure (**sweep line status**).

The partial solution and the sweep line status only change at specific positions (**events**) that may be part of the input or arise during the execution of the algorithm.

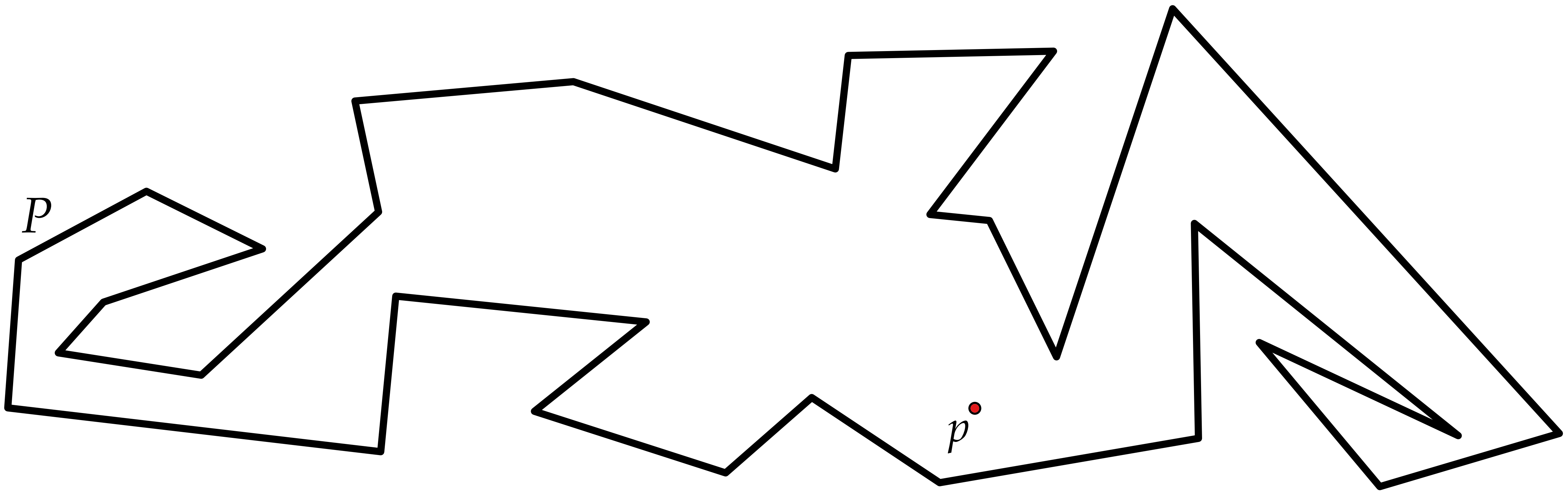
The sweep line paradigm is **powerful** and leads to **simple** algorithms for many problems:
computing Voronoi diagrams, crossings in an arrangement of line segments,
intersection/union of two polygons, decompositions of polygons,
certain triangulations, visibility polygons, ...

Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

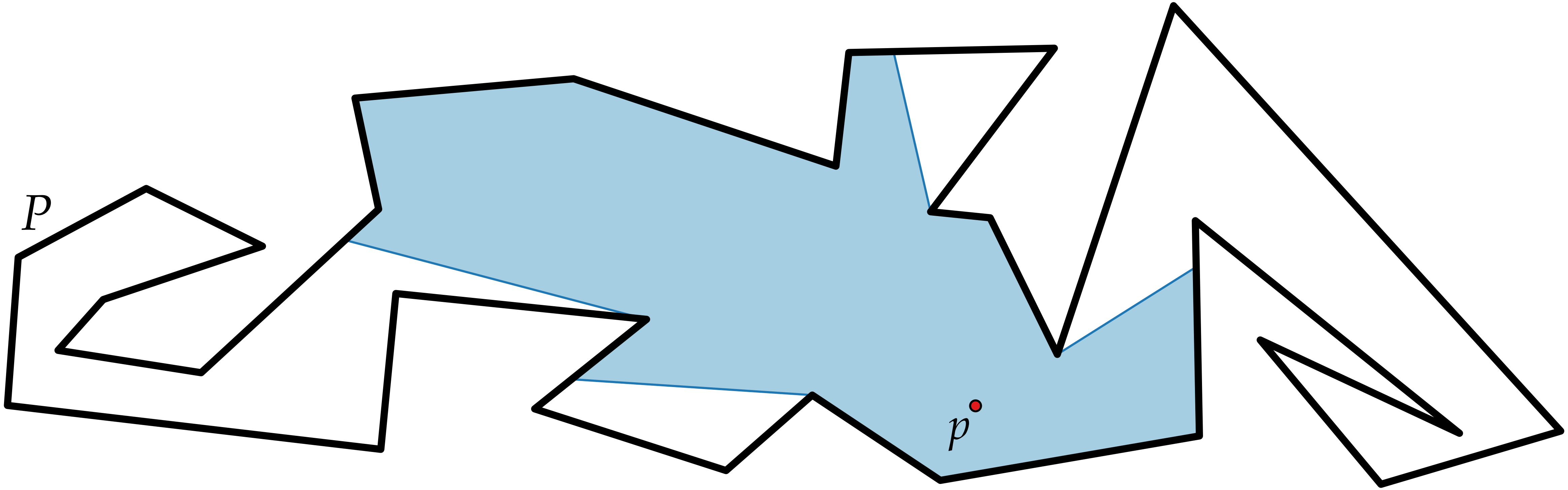


Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

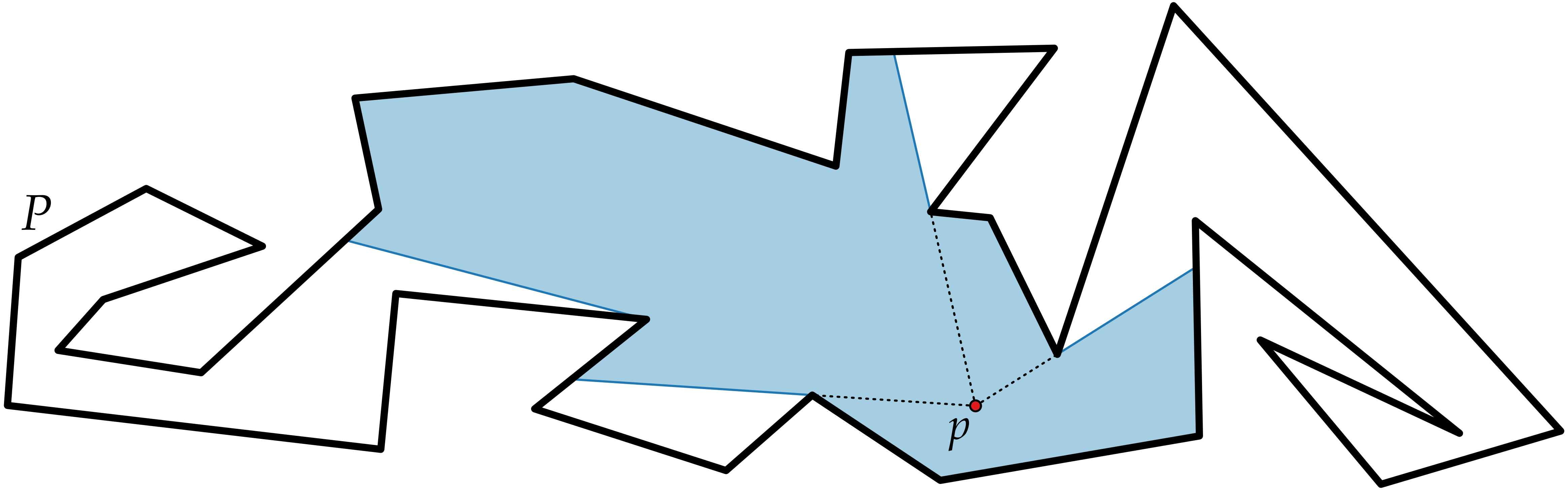


Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

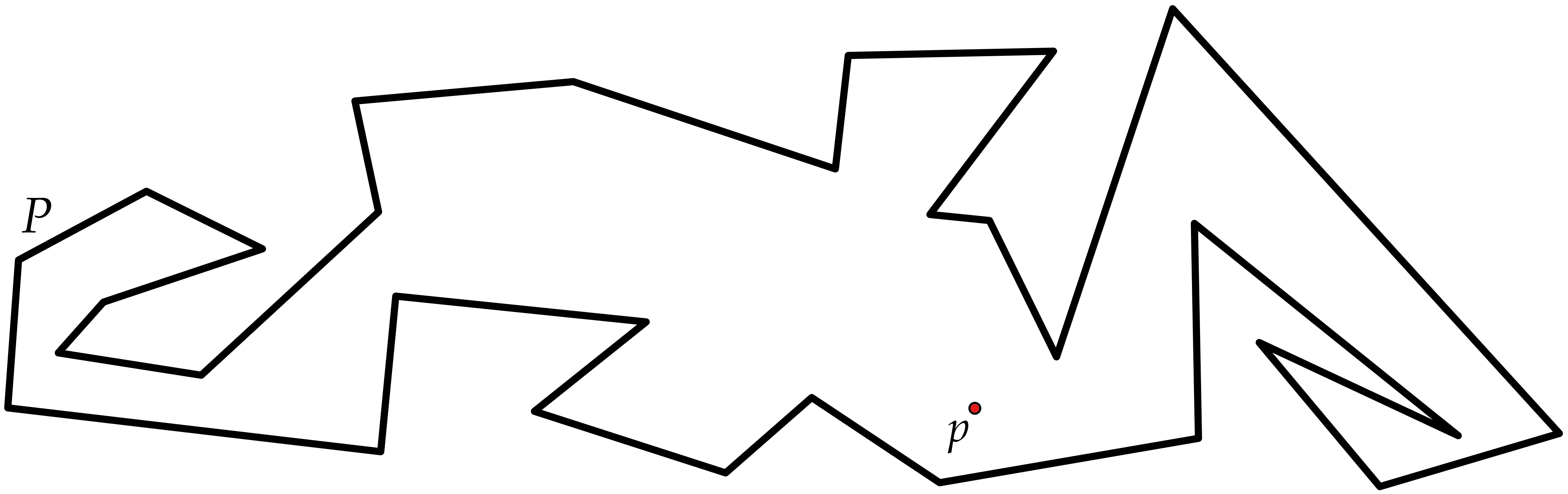


Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.



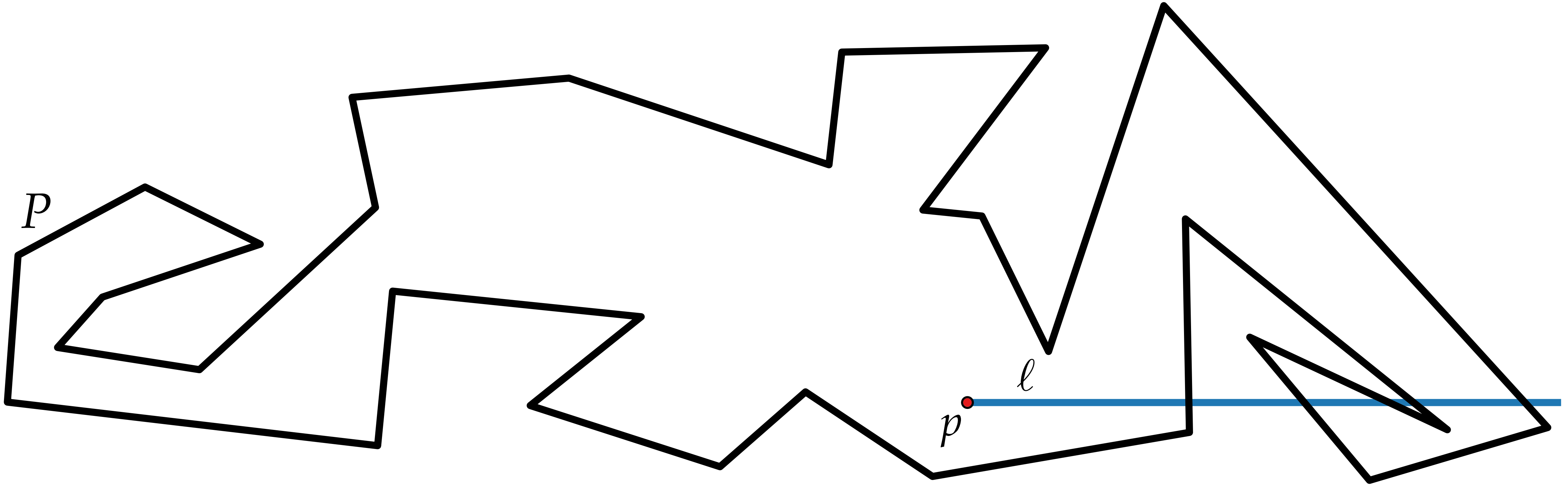
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



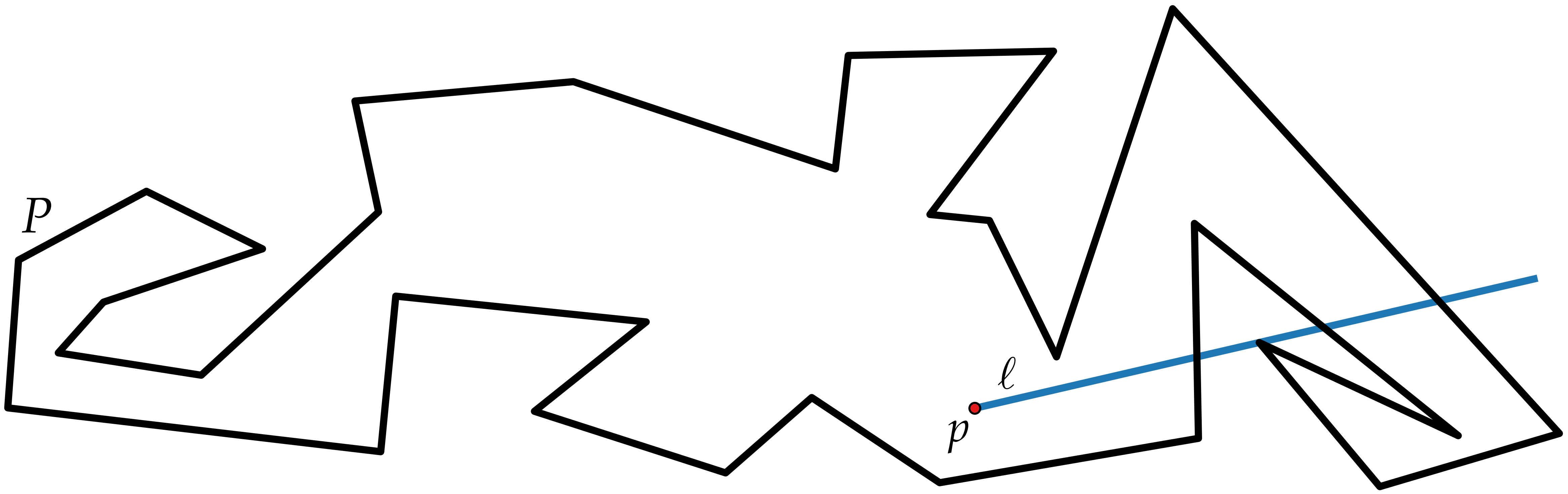
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



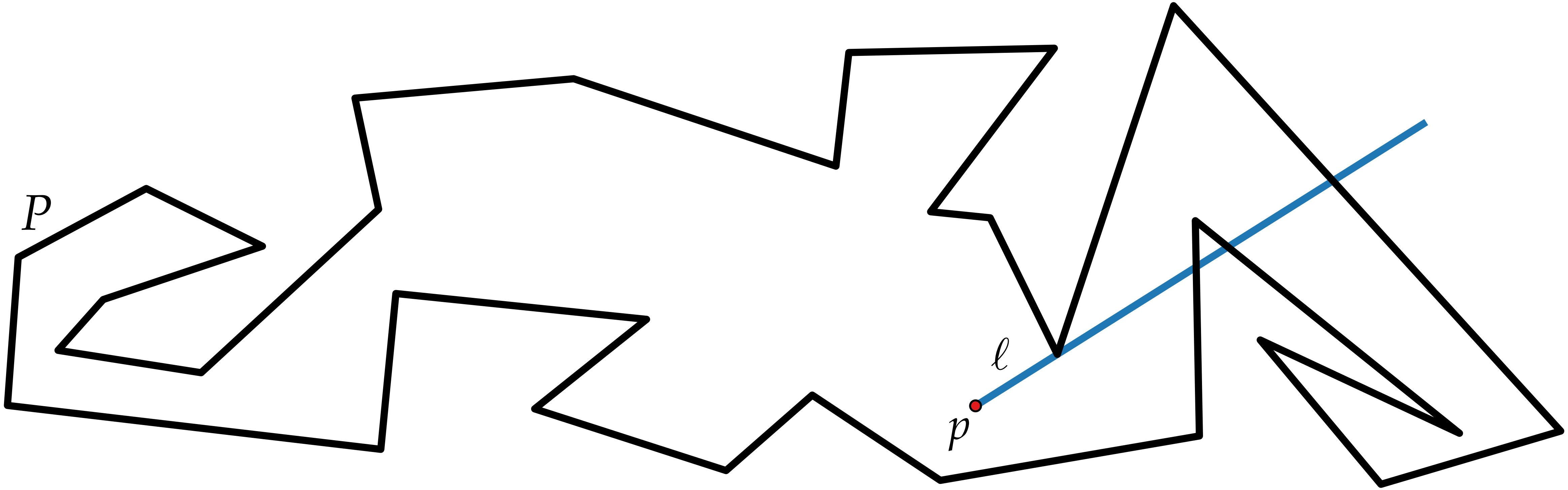
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



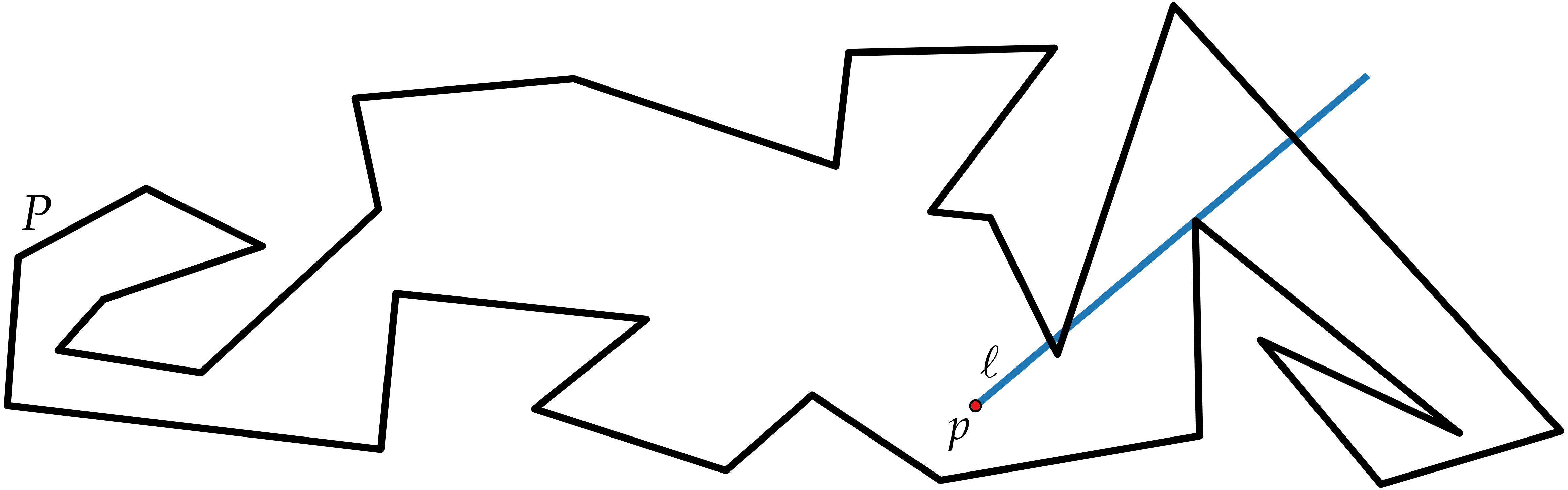
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



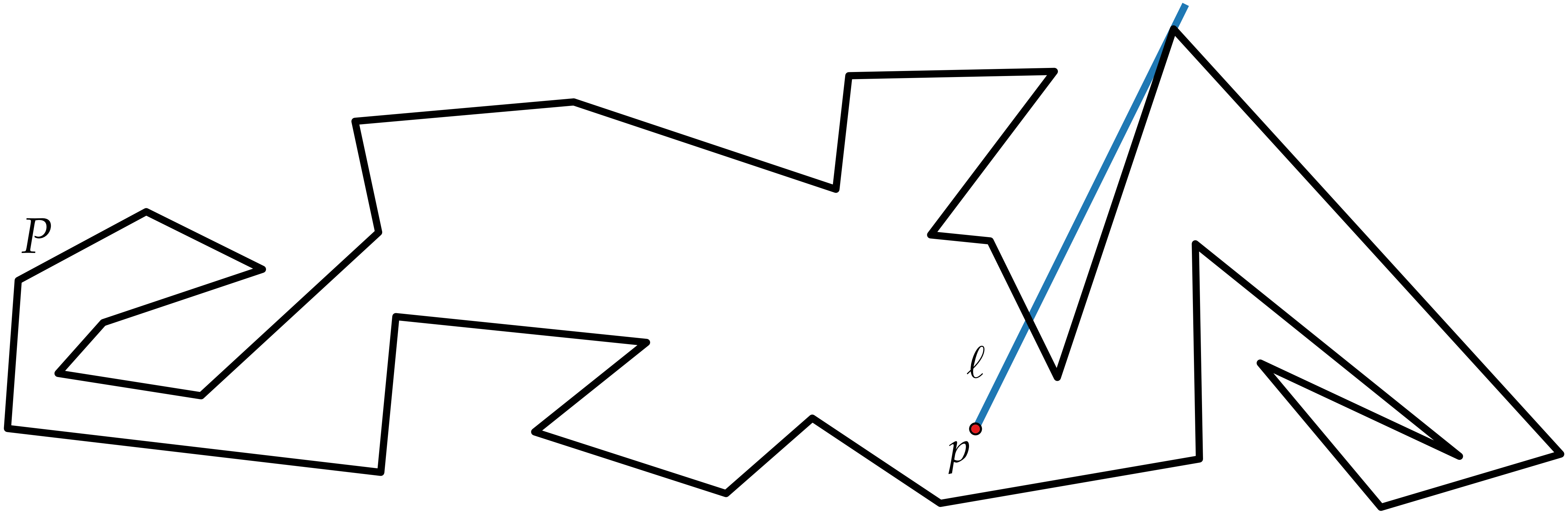
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



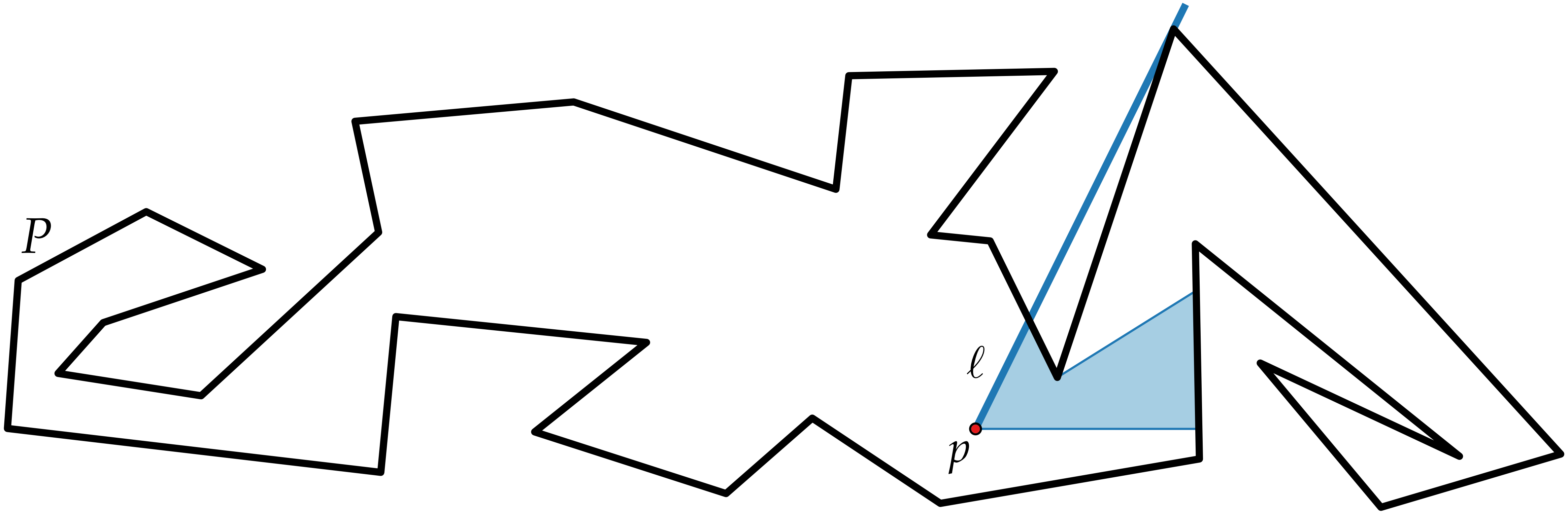
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



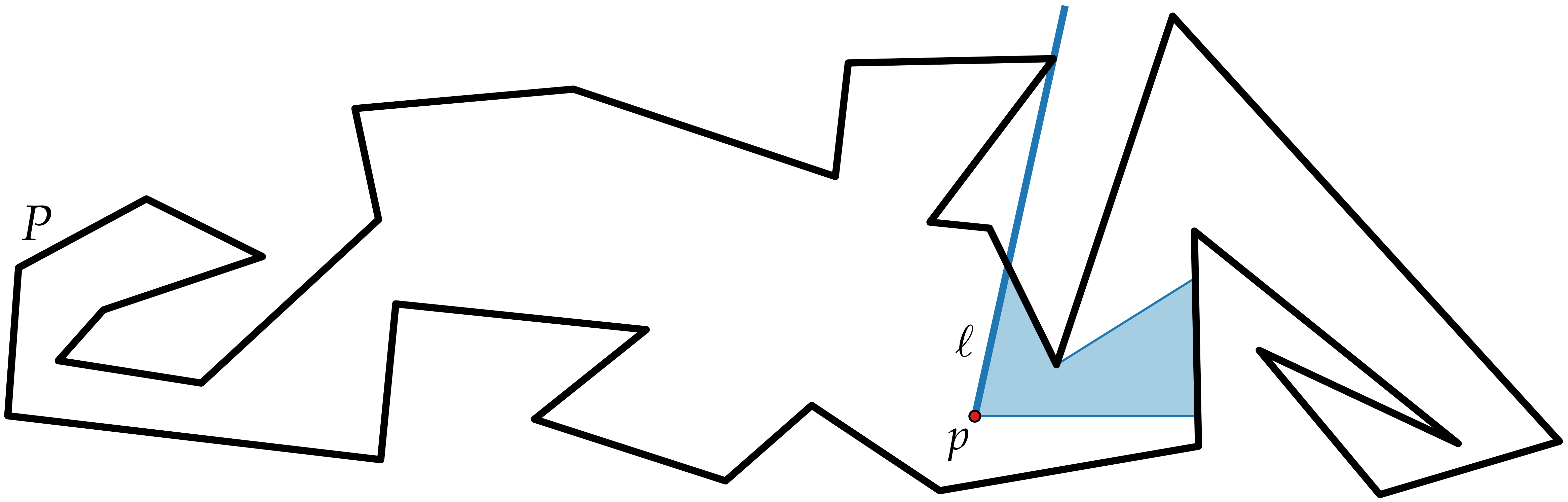
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



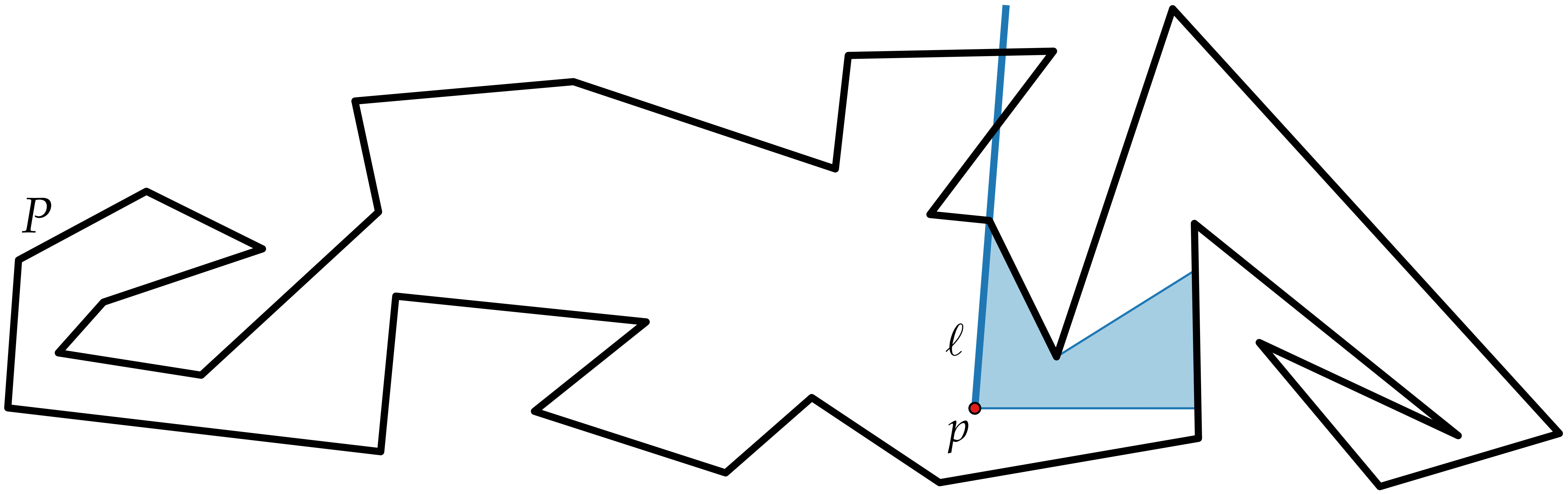
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



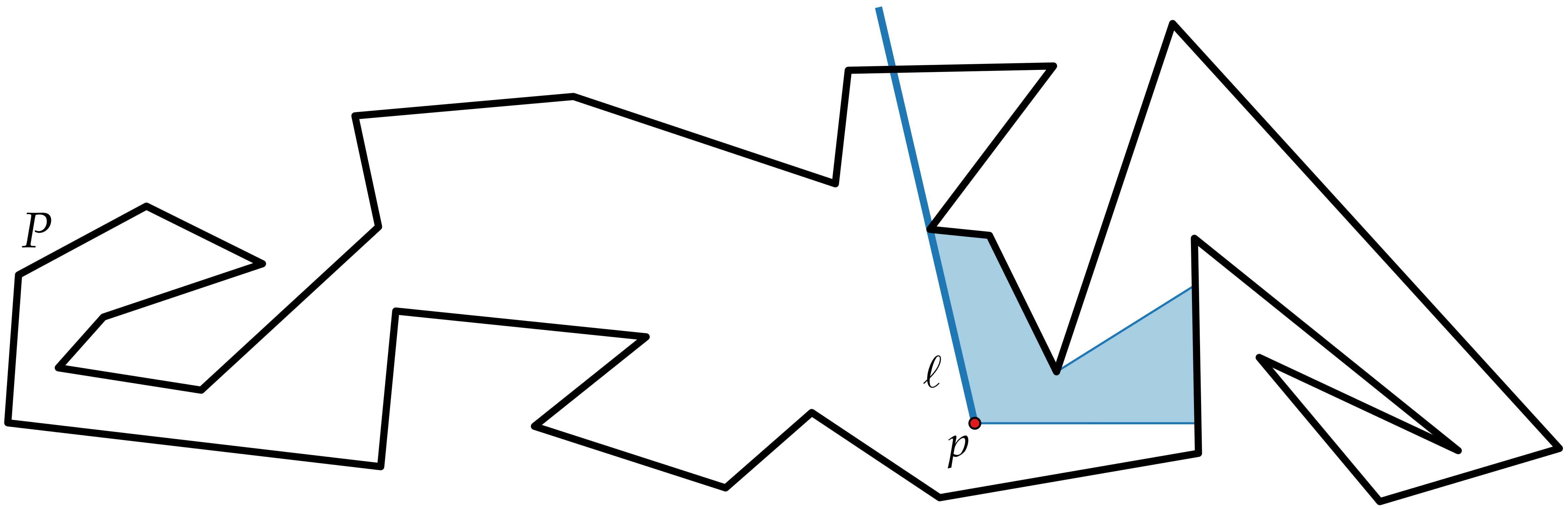
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



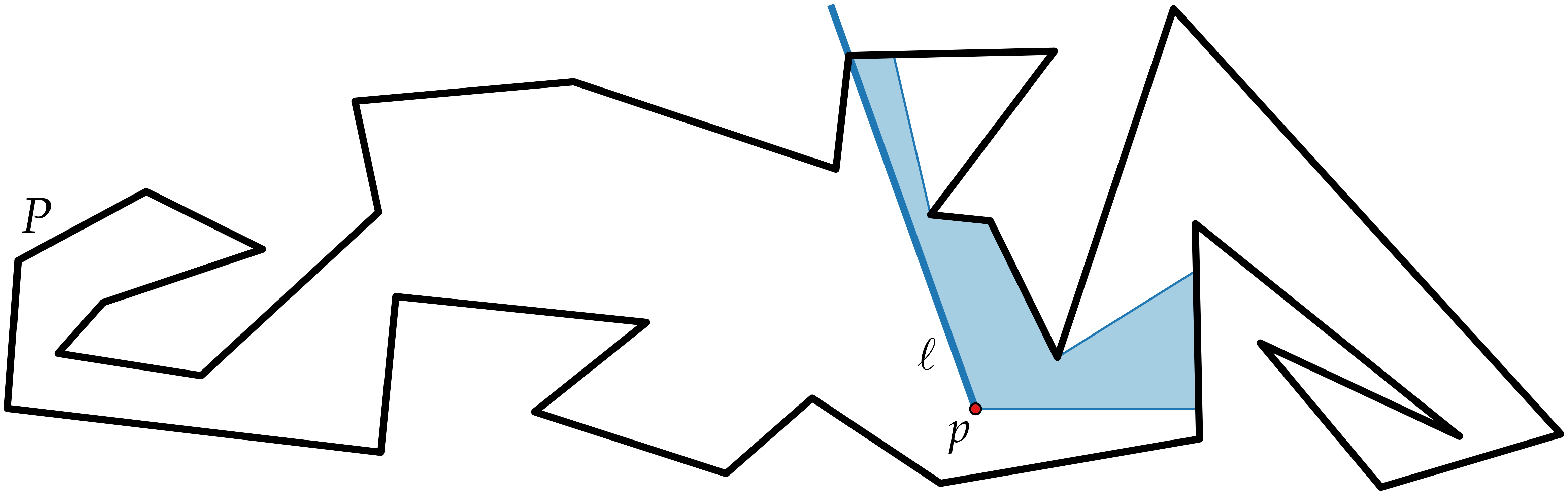
Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.



Outlook: Computing Visibility Polygons

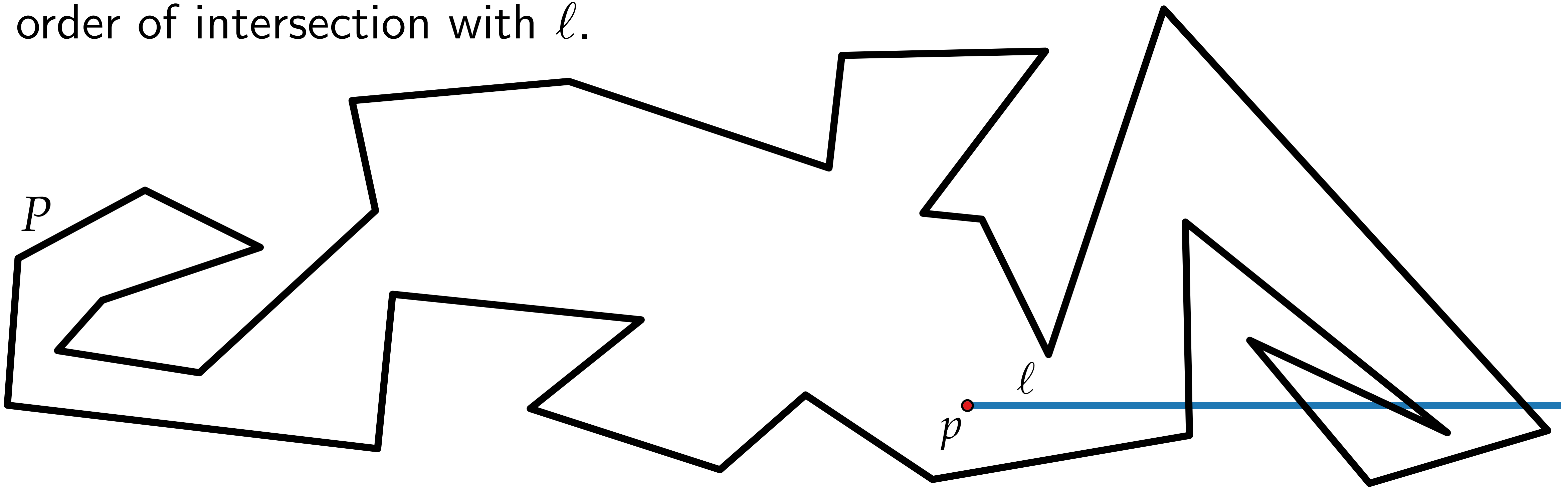
The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.



Outlook: Computing Visibility Polygons

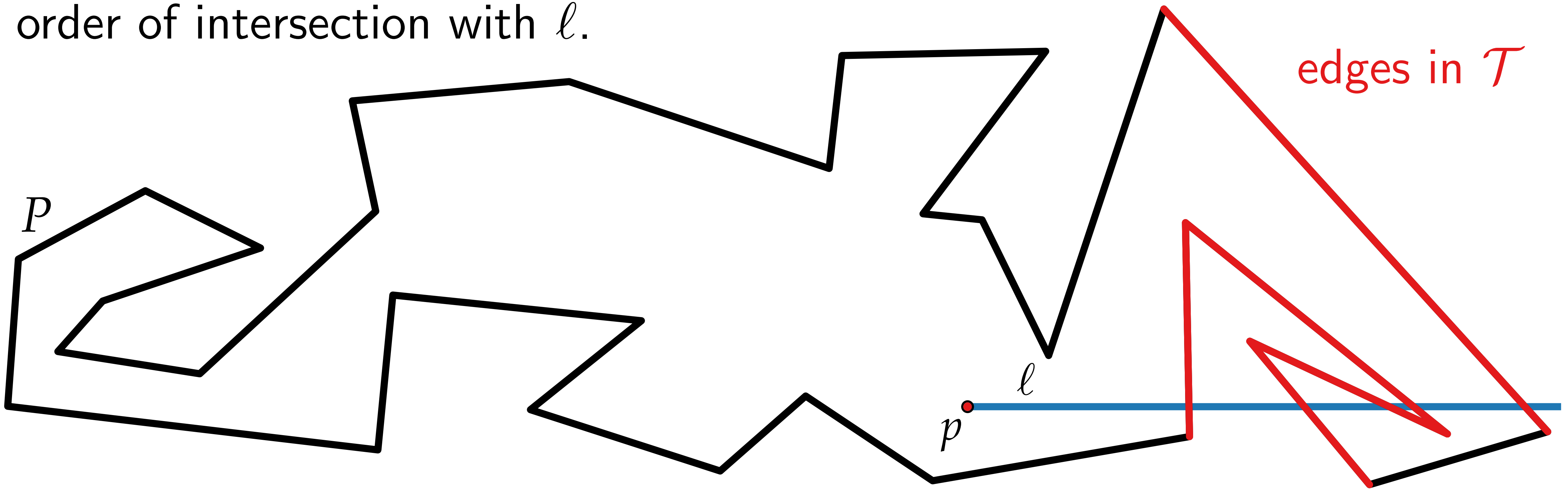
The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

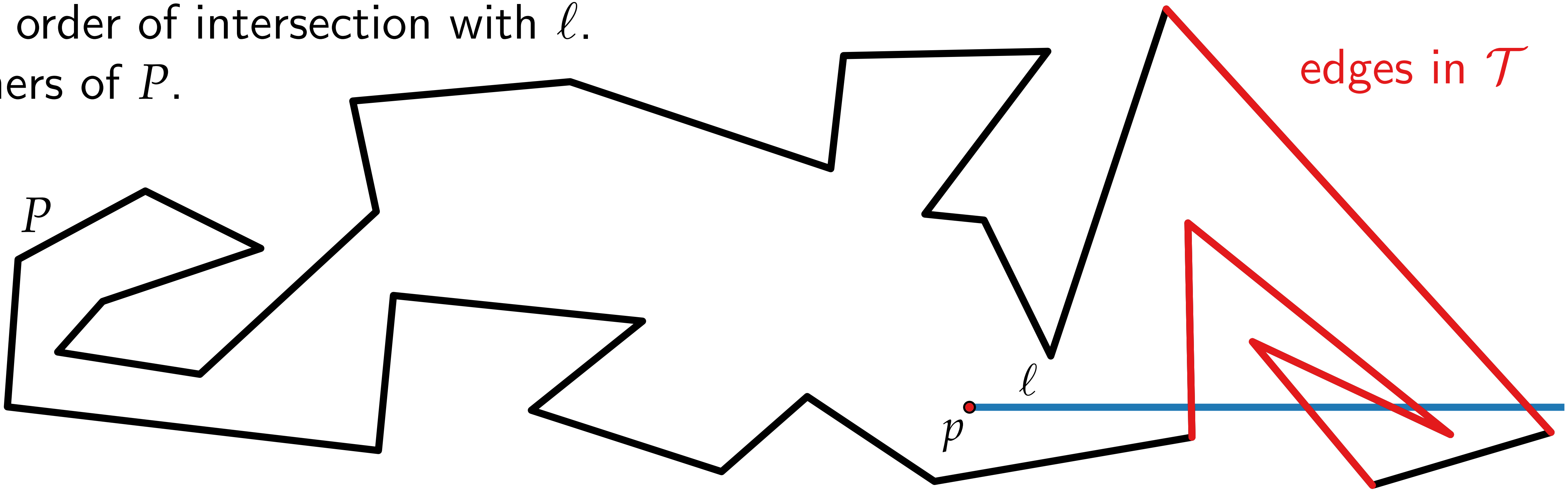
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

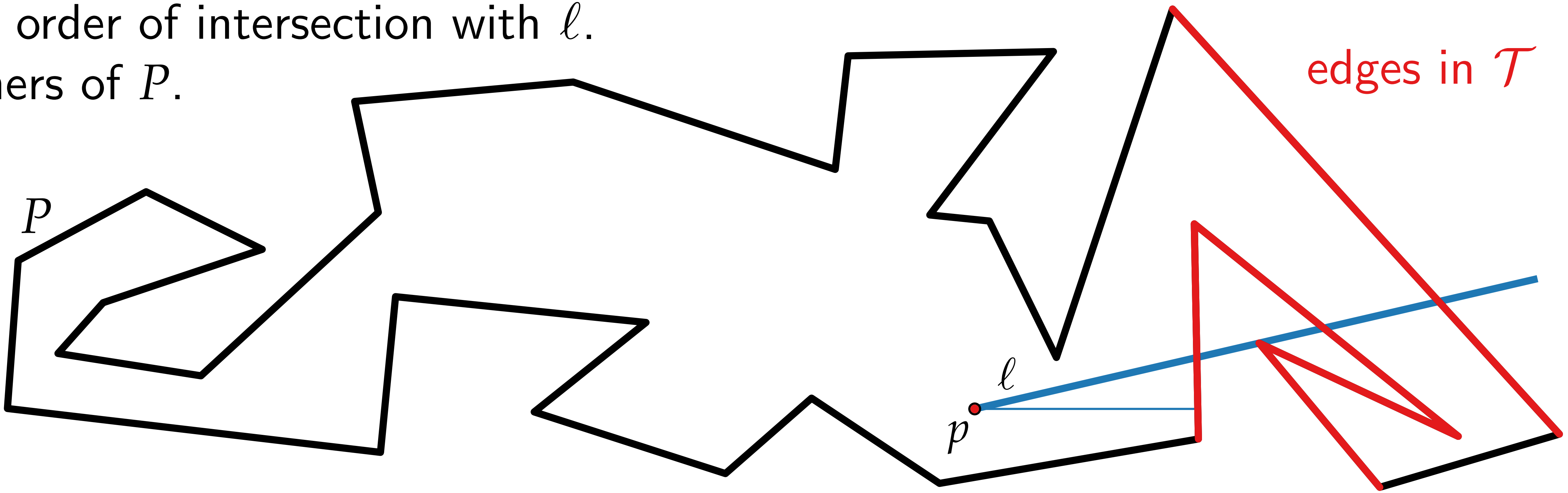
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

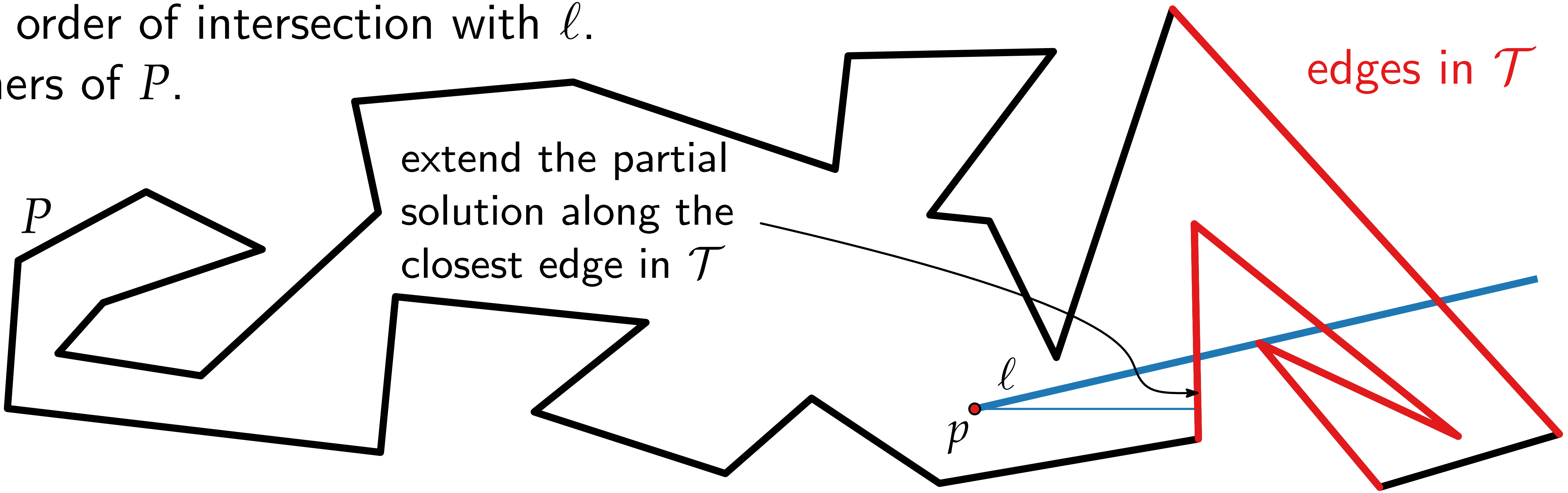
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

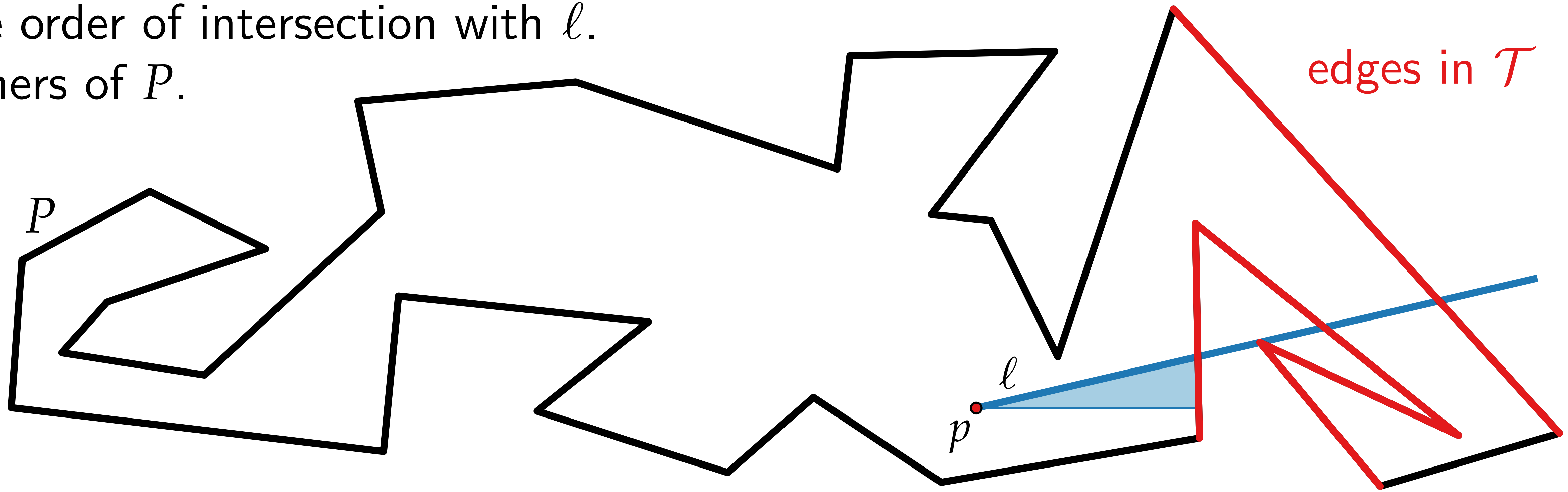
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

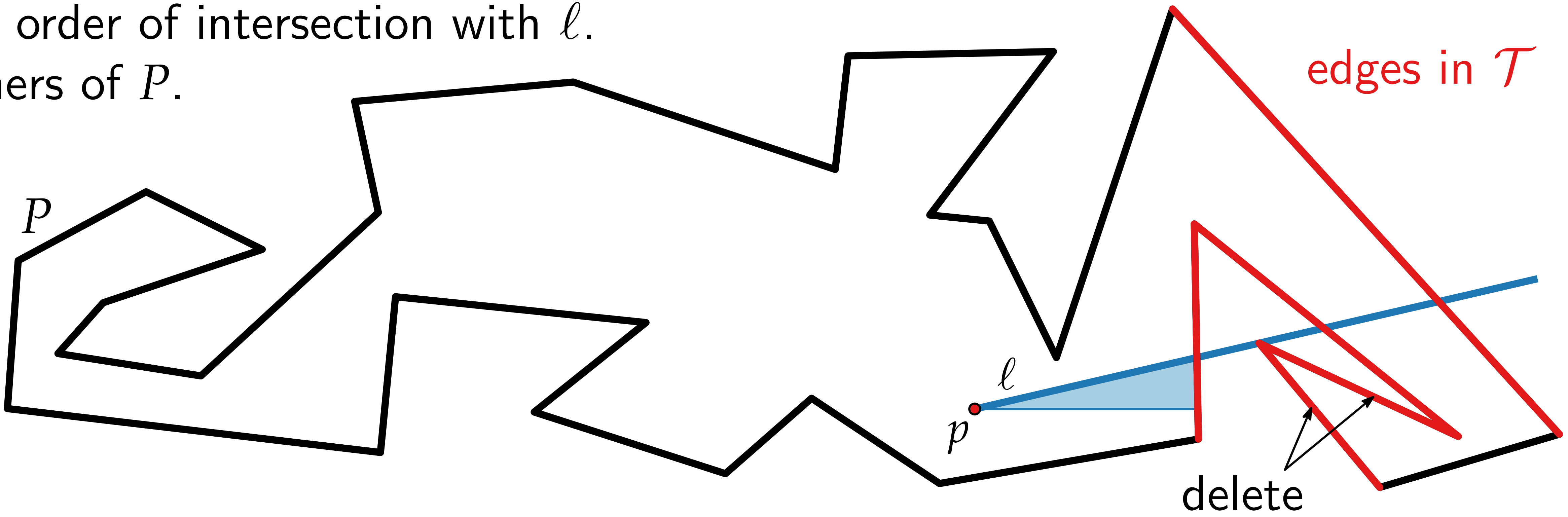
The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.
Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

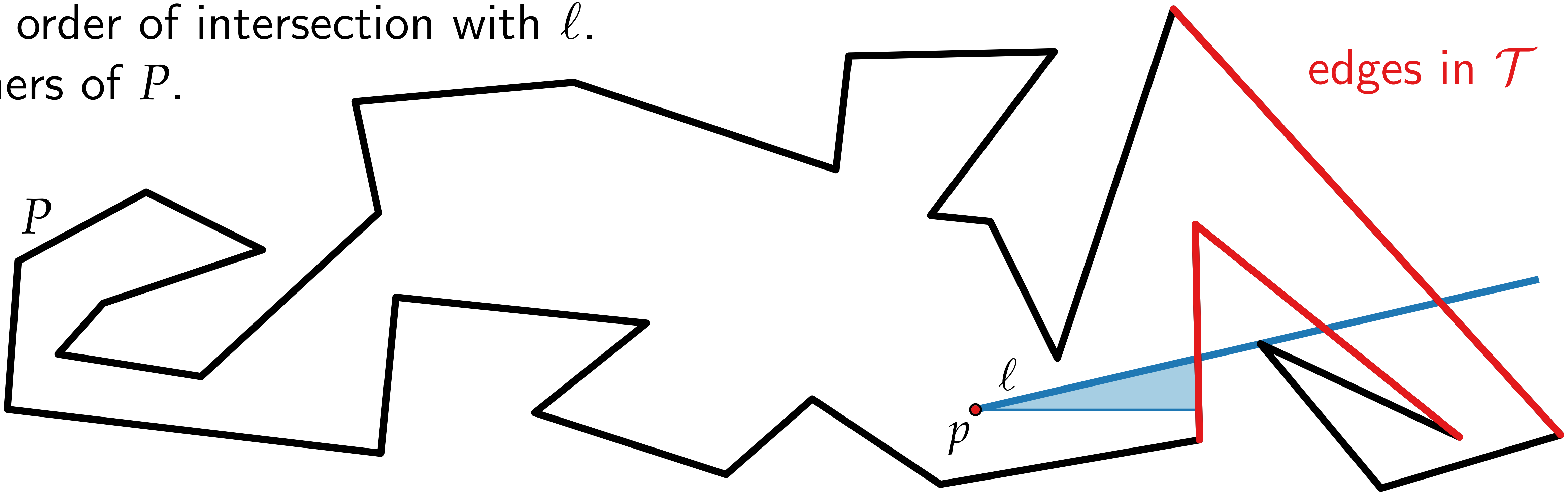
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

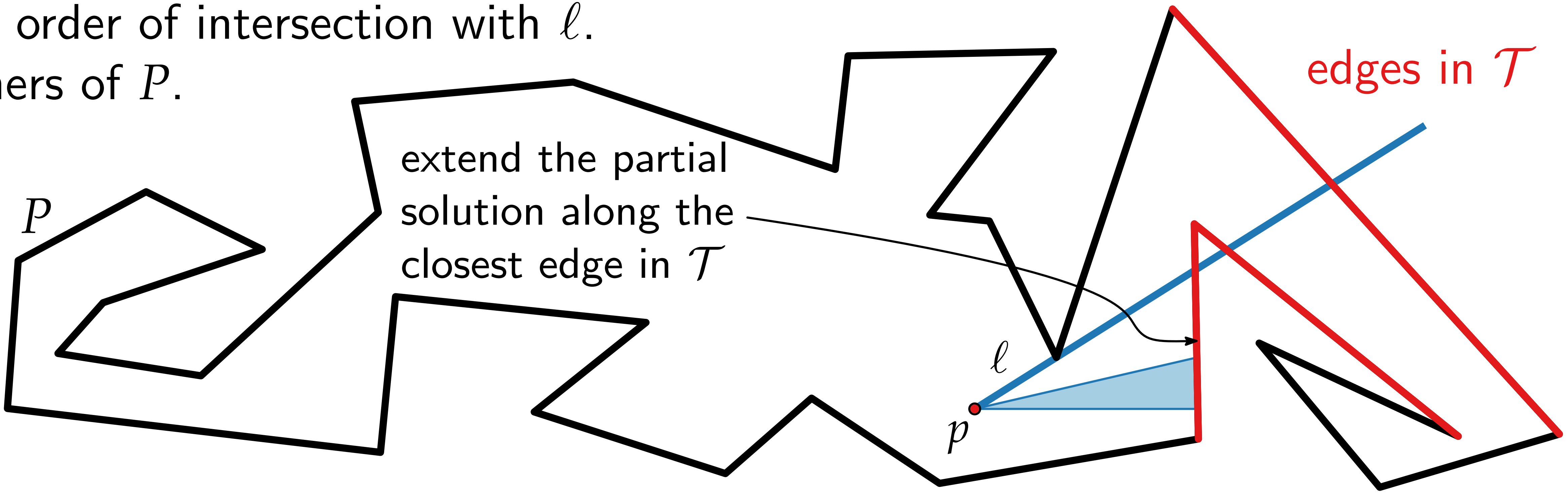
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

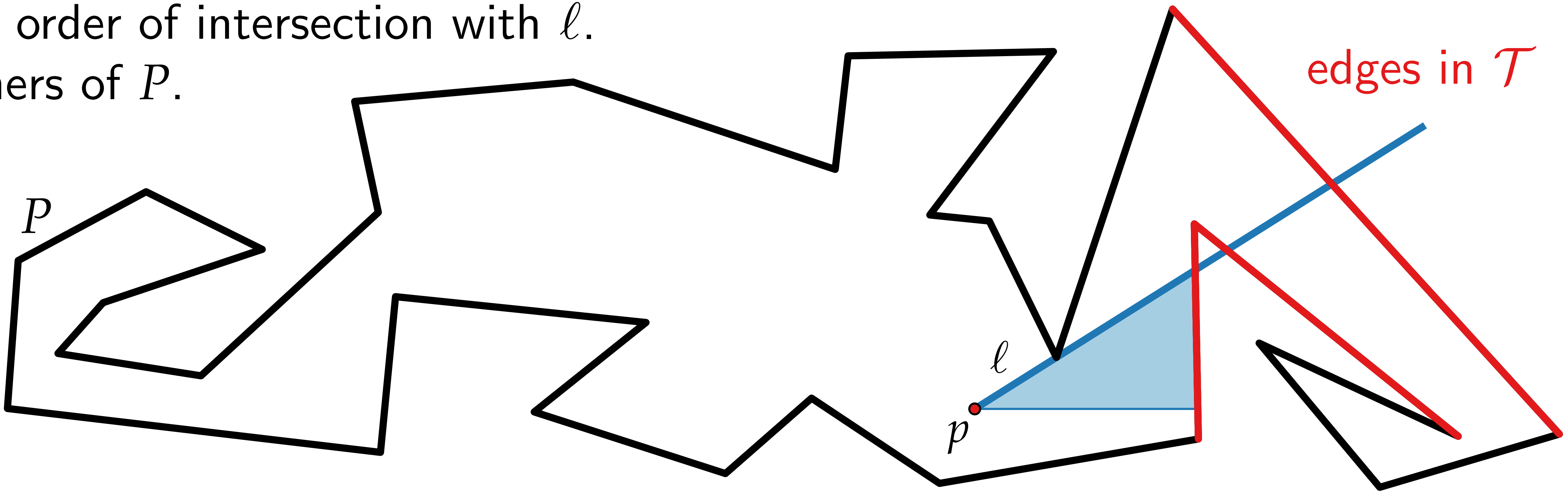
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

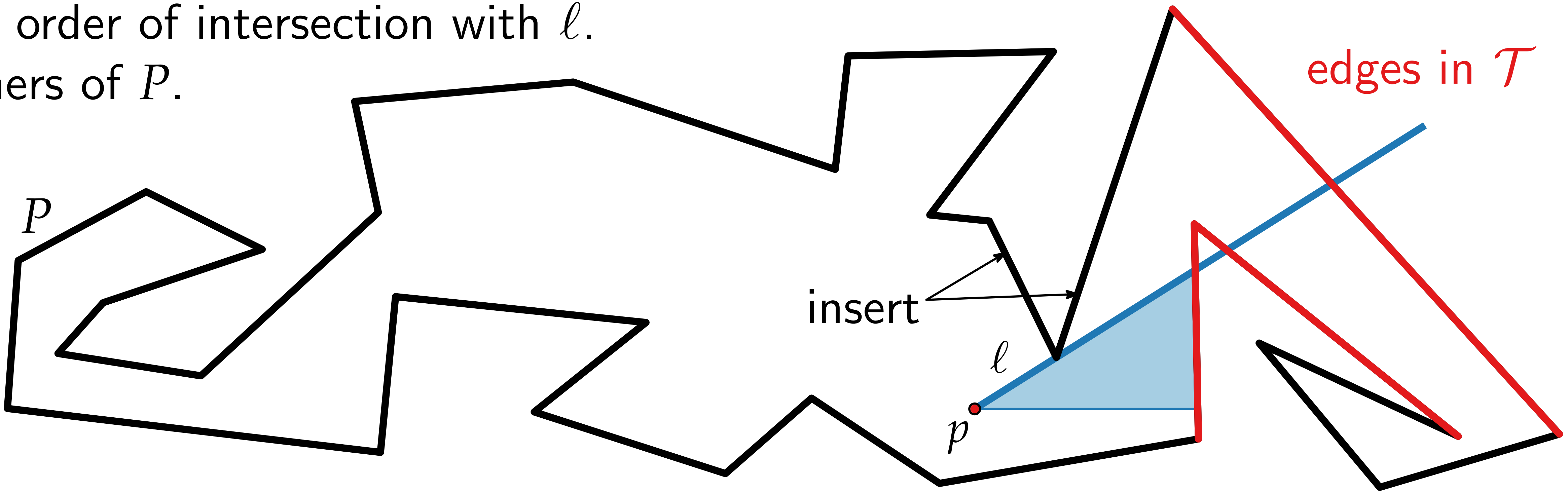
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

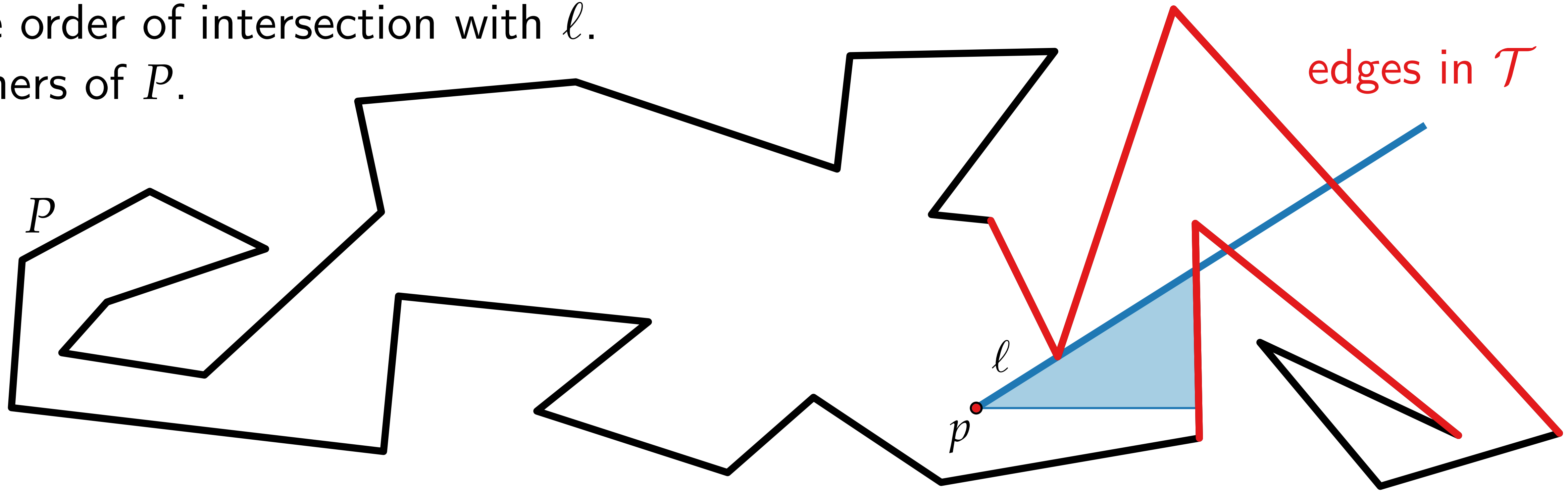
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

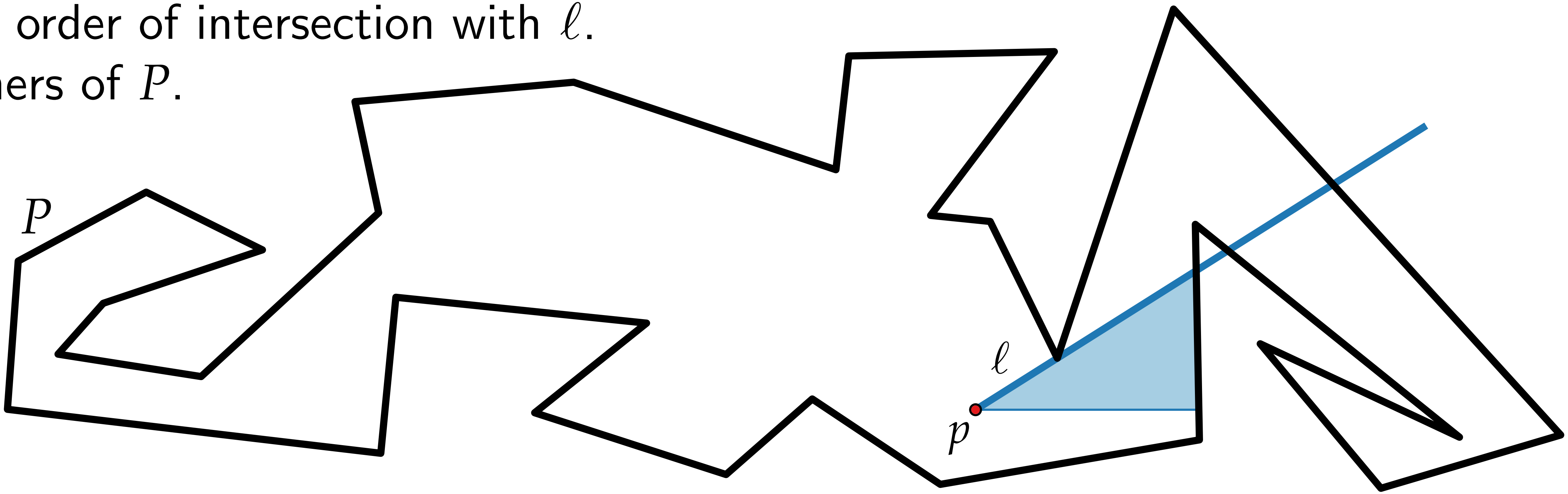
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

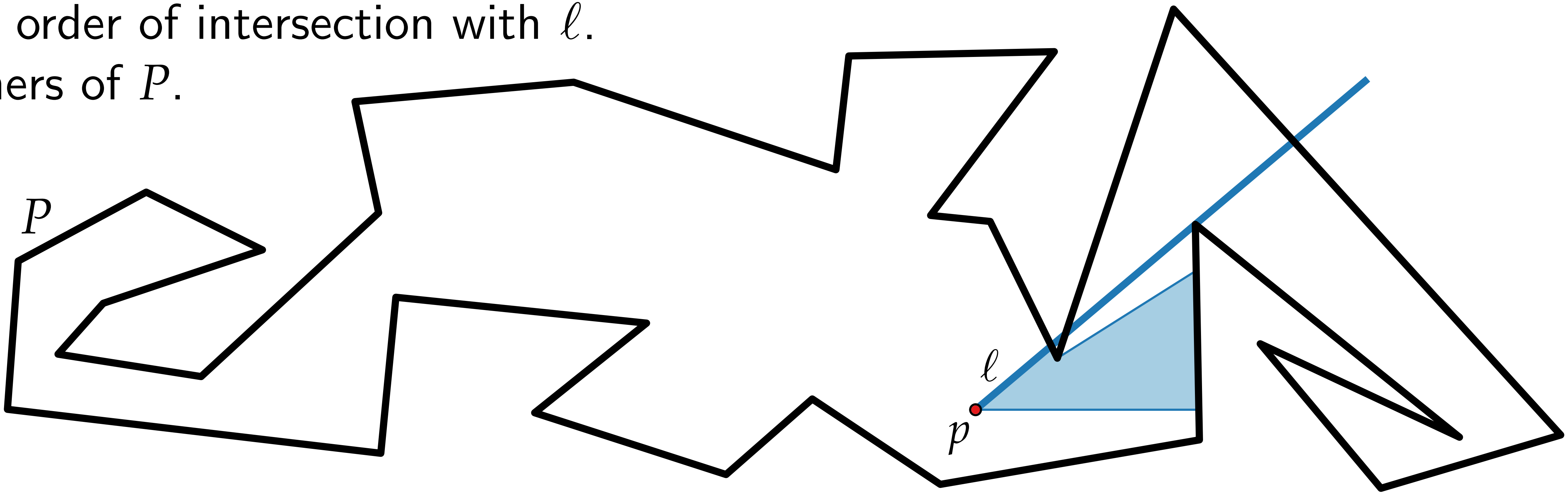
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

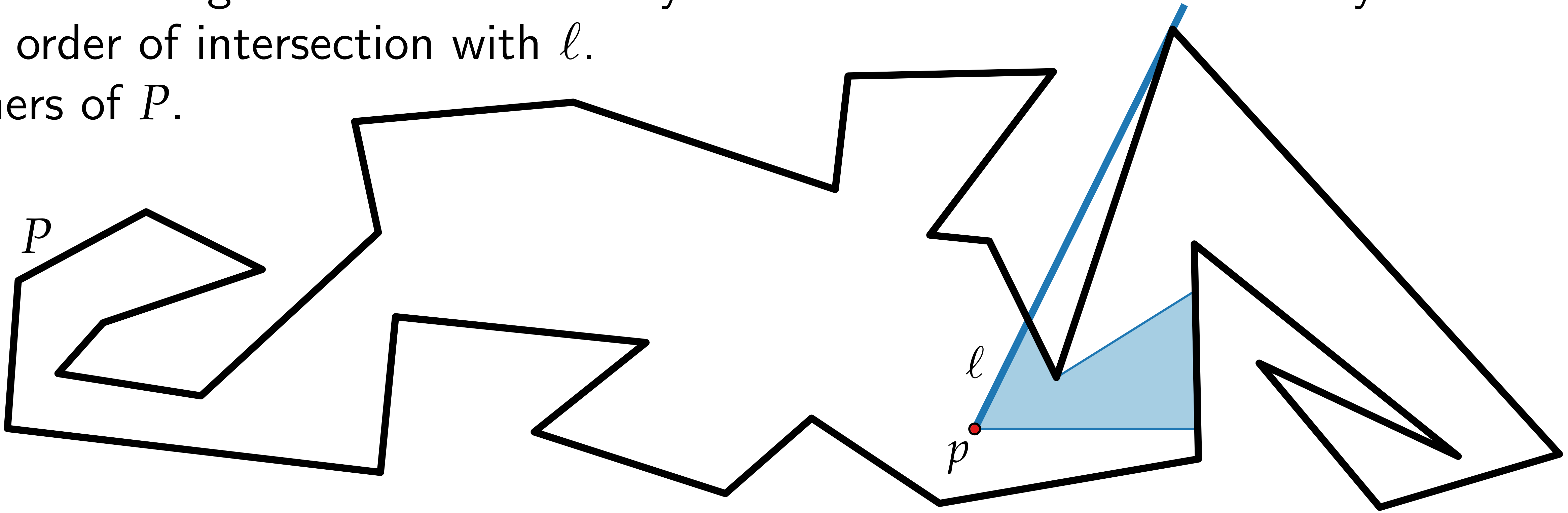
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

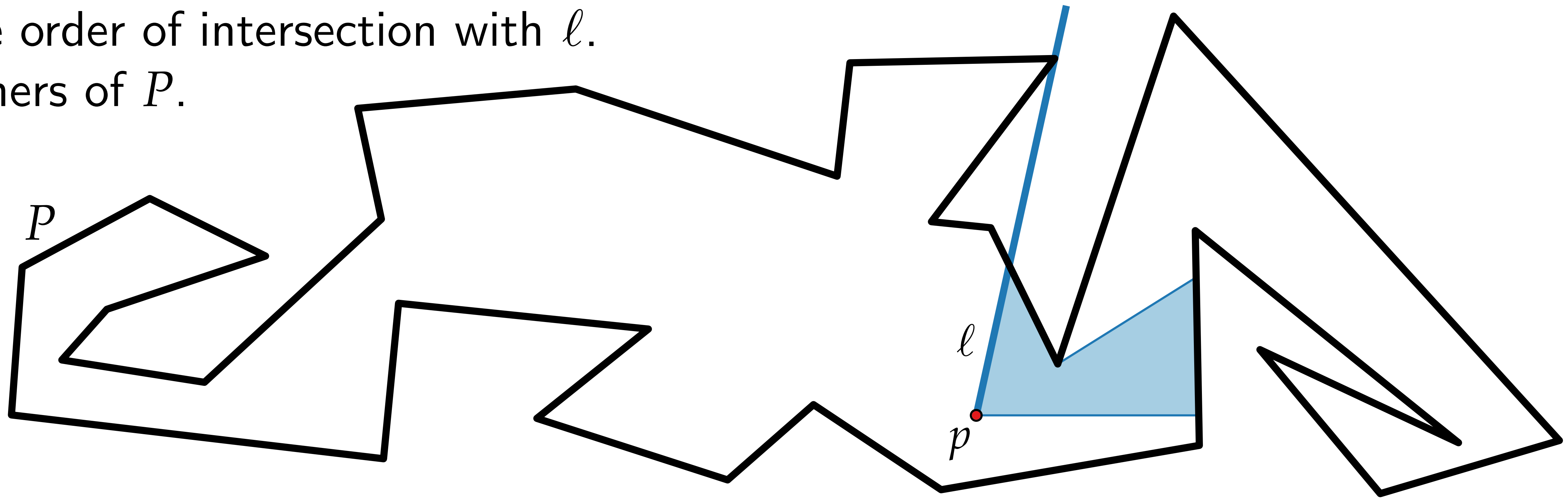
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

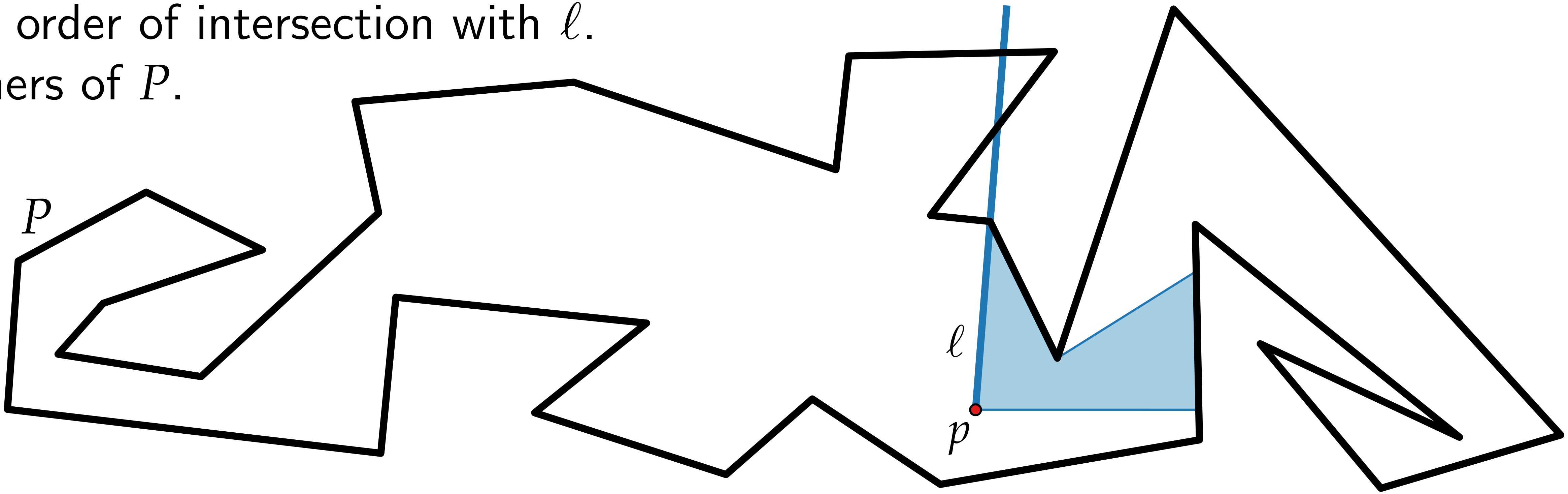
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

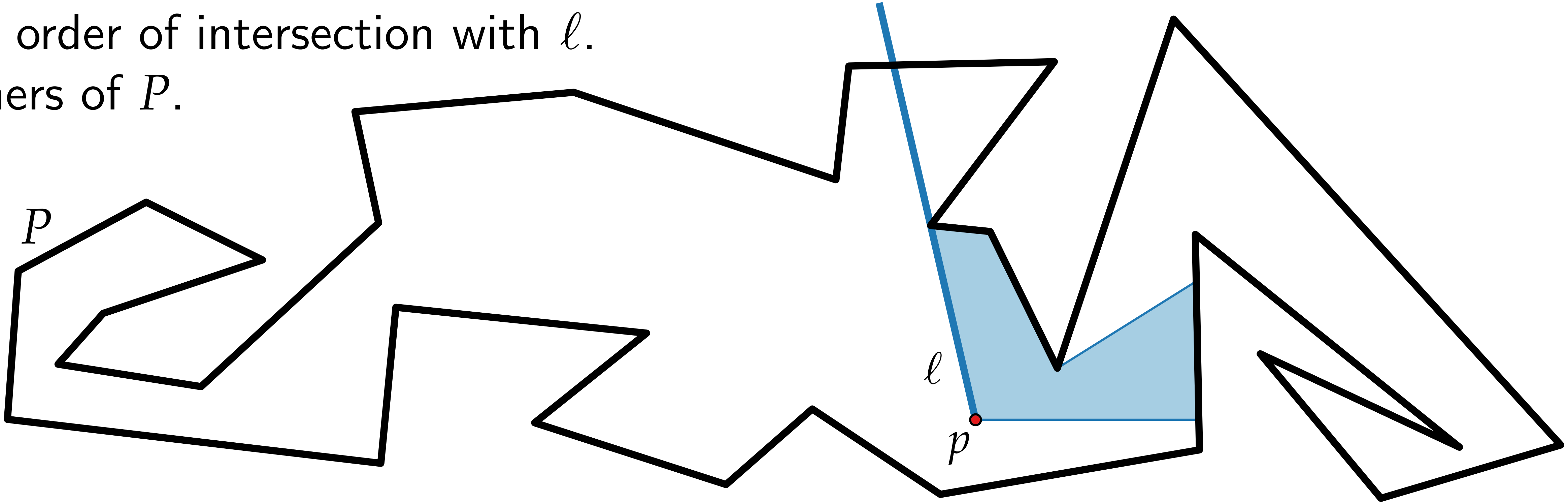
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

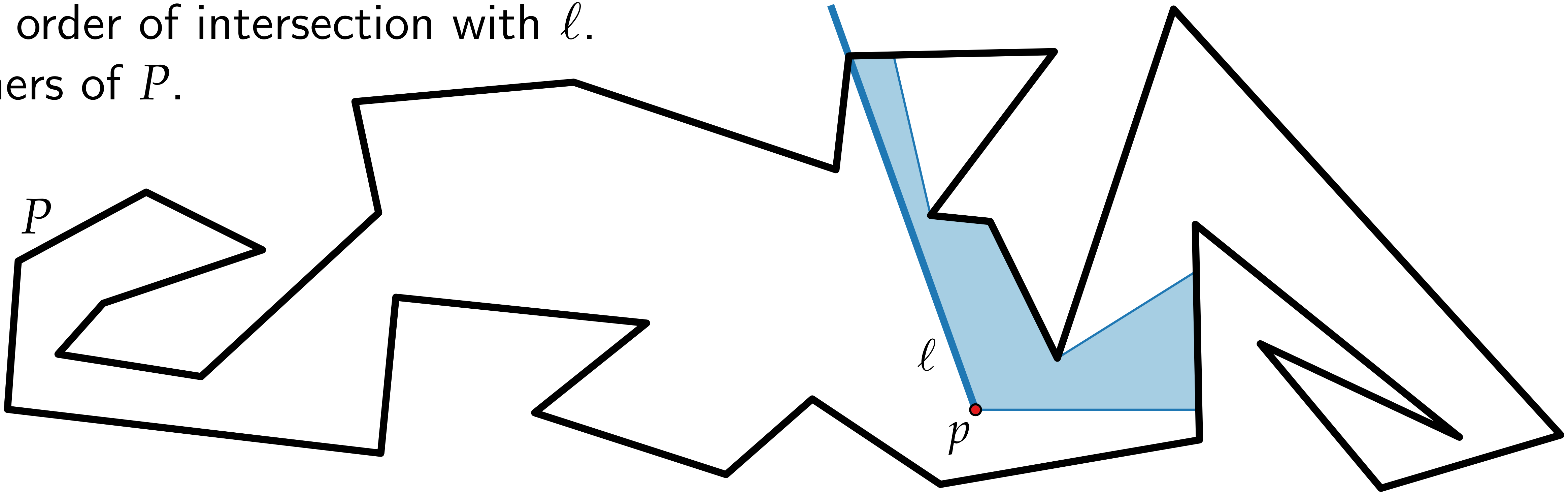
Given: A polygon P with n corners and a point p in its interior.

Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Outlook: Computing Visibility Polygons

The sweep "line" does not always have to move from left to right!

Given: A polygon P with n corners and a point p in its interior.

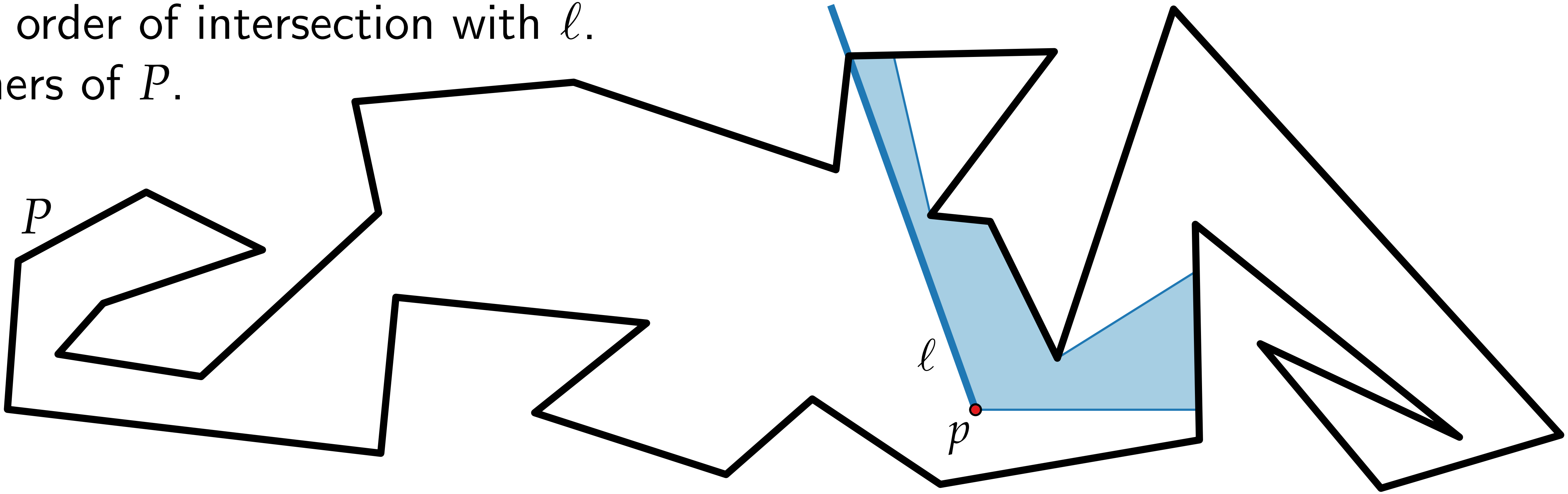
Task: Compute the visibility polygon of p with respect to P.

Idea: Sweep a **ray** ℓ radially around p.

Total runtime: $\mathcal{O}(n \log n)$

Sweep line status: Edges of P intersected by ℓ are stored in a balanced binary search tree $\mathcal{T}$ in the order of intersection with ℓ.

Events: Corners of P.



Literature

Rolf Klein. Algorithmische Geometrie: Grundlagen, Methoden, Anwendungen. Springer Verlag 2005.