# CSE520: Computational Geometry
# Lecture 3
# Line Segment Intersection

Antoine Vigneron

Ulsan National Institute of Science and Technology

June 15, 2020

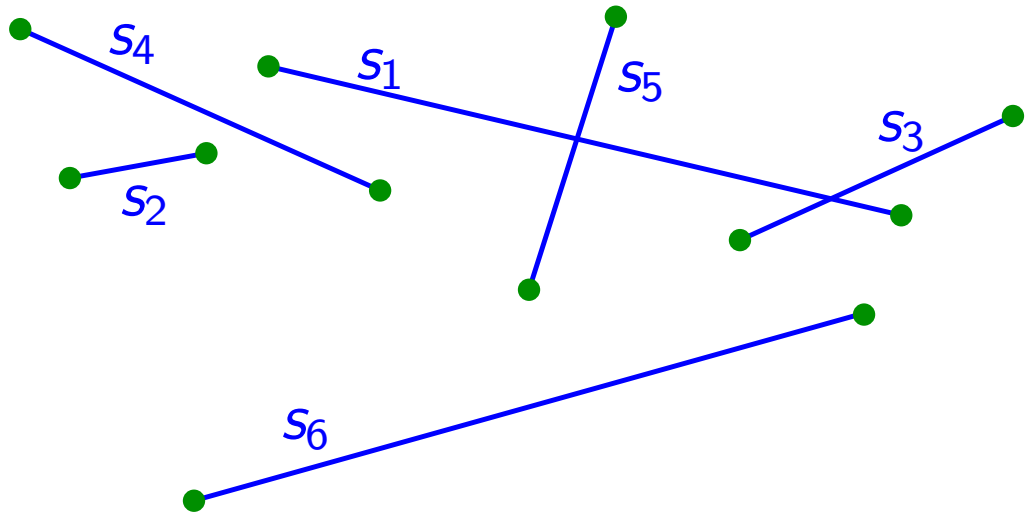
1. Introduction
2. Preliminary
3. Intersection Detection
4. Intersection reporting

# Course Information

- Reference for this Lecture: textbook Chapter 2.

# Problems

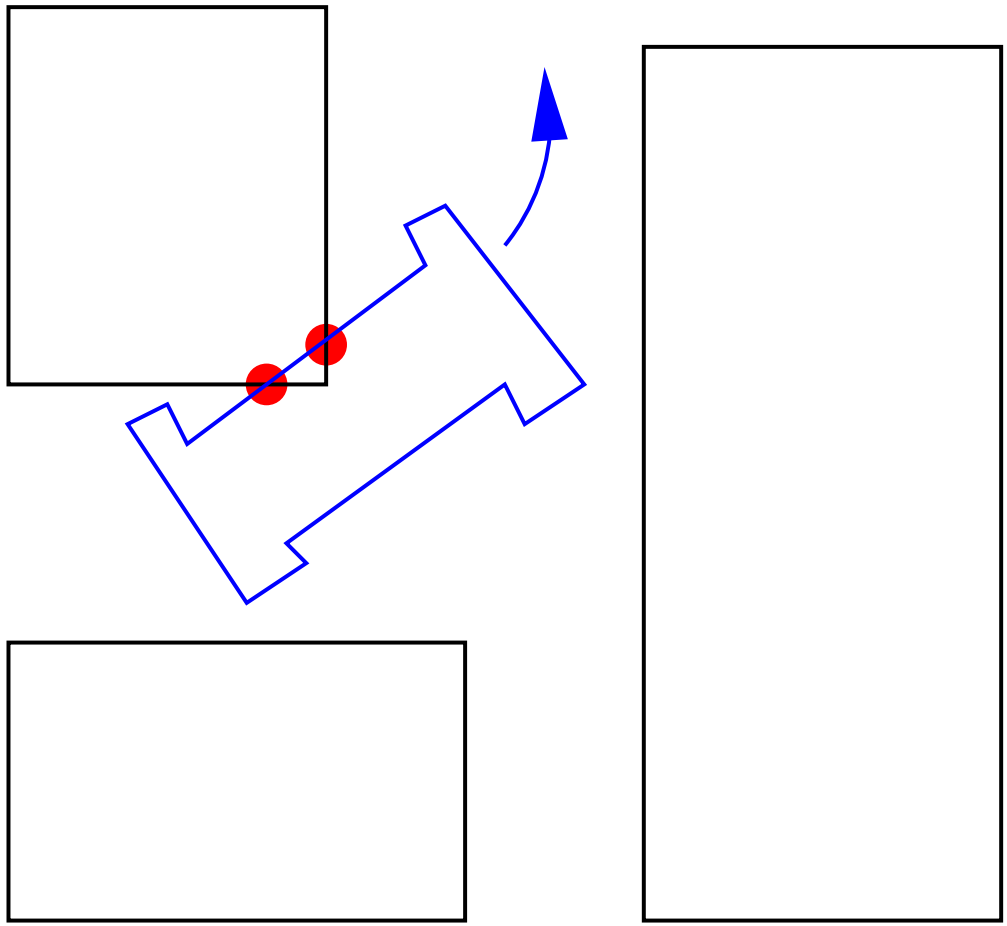
**INPUT:** a set $S = \{s_1, s_2, \ldots s_n\}$ of $n$ line segments in $\mathbb{R}^2$ given by the coordinates of their endpoints.




- *Intersection detection*: Is there a pair $(s_i, s_j) \in S^2$ such that $i \neq j$ and $s_i \cap s_j \neq \emptyset$?
- *Intersection reporting*: Find all pairs $(s_i, s_j) \in S^2$ such that $i \neq j$ and $s_i \cap s_j \neq \emptyset$.

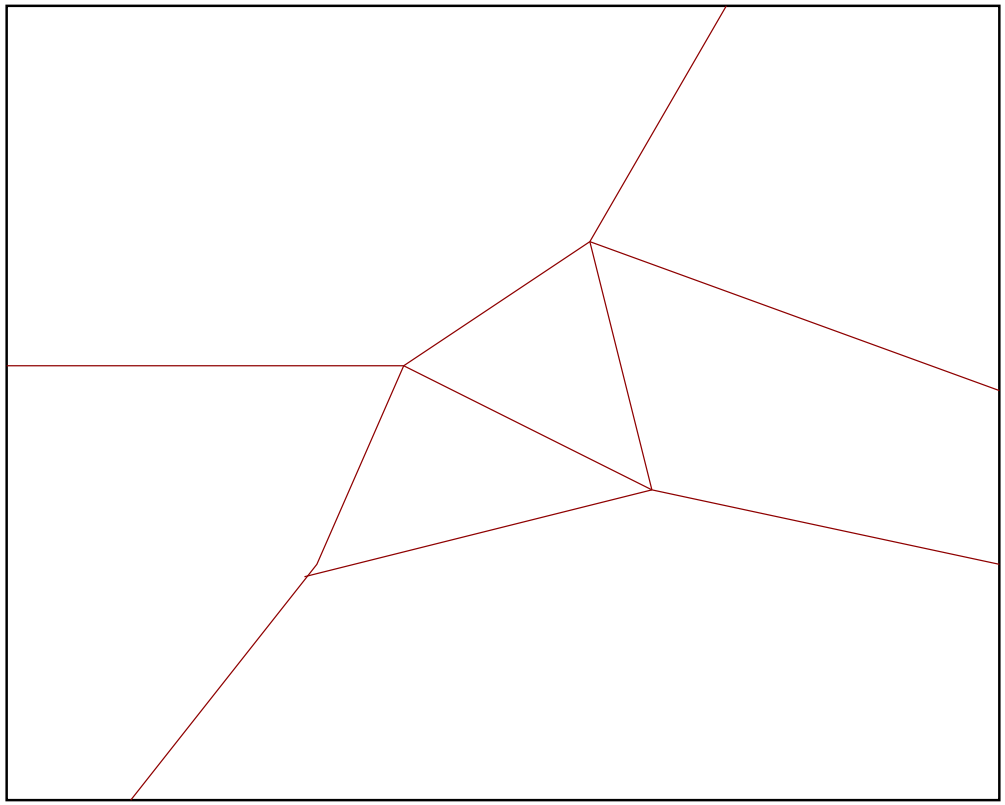
# Motivation

- Motion planning: collision detection



# Motivation

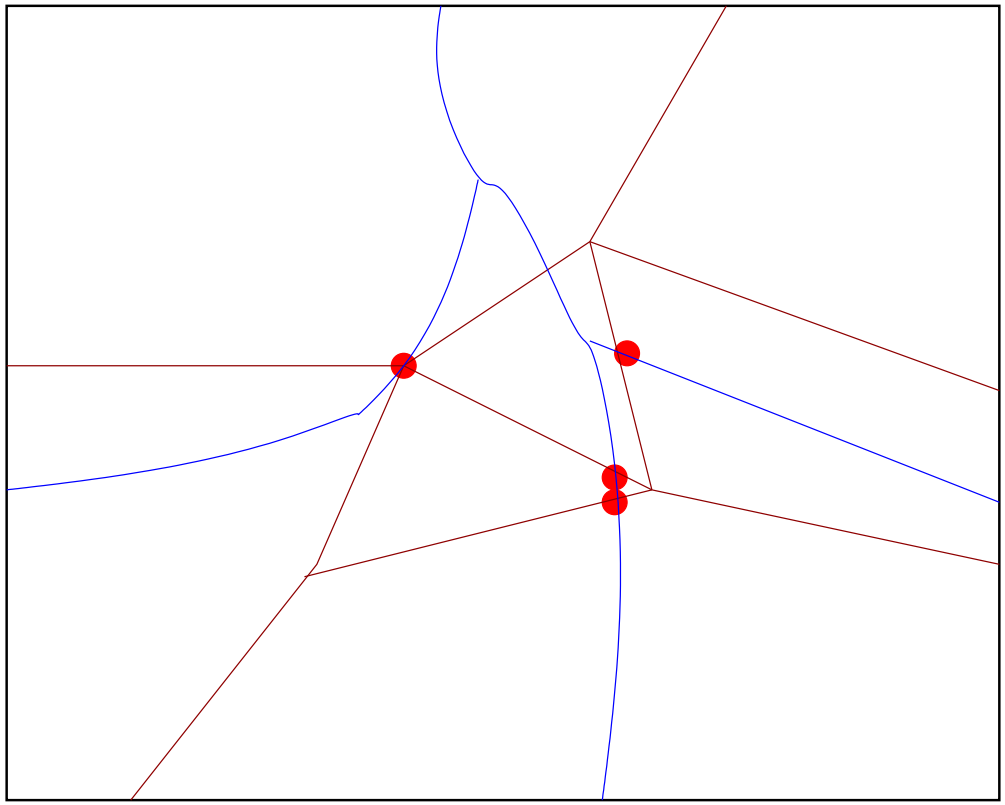
- Geographic Information Systems: map overlay
- More generally: spatial join in databases



ROAD MAP

# Motivation

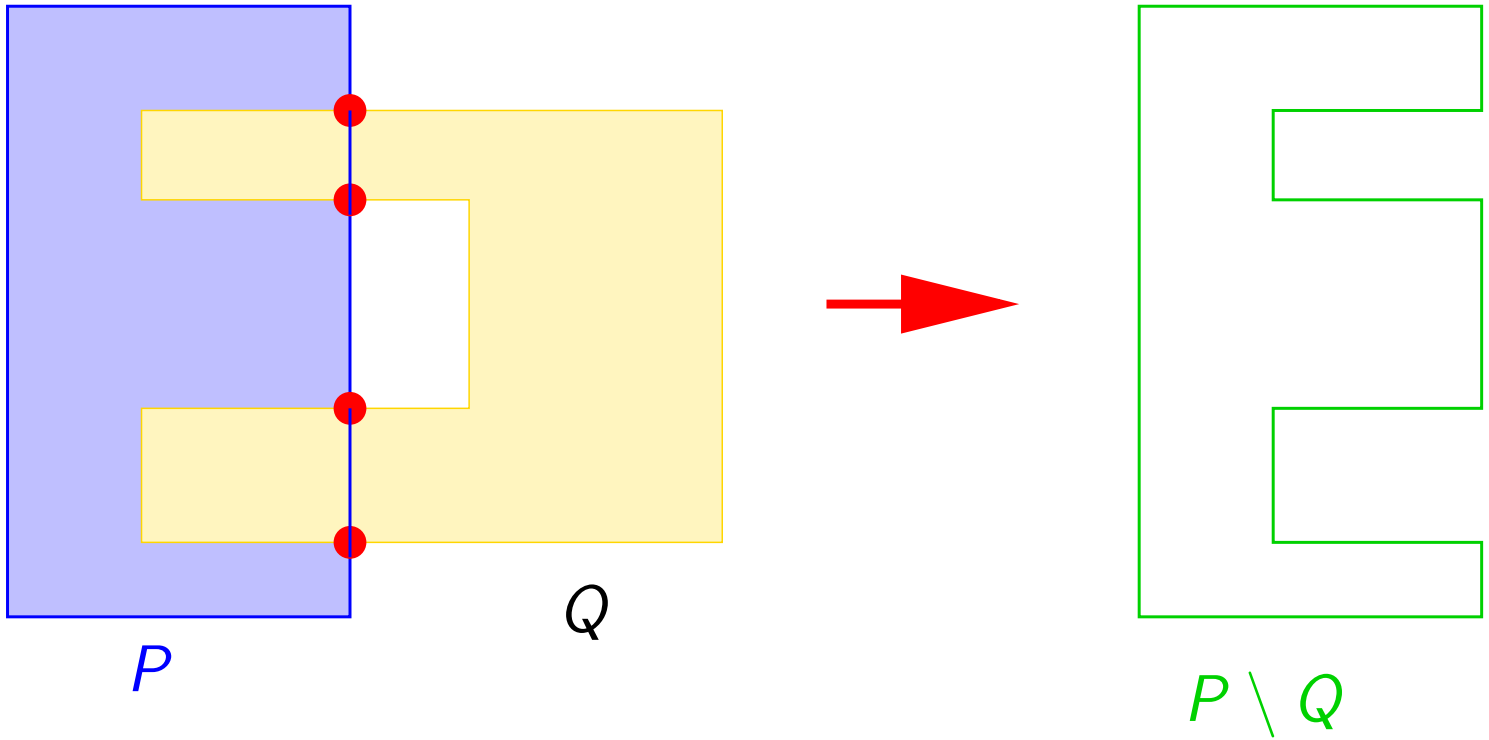
- Geographic Information Systems: map overlay
- More generally: spatial join in databases



ROAD MAP + RIVER MAP

# Motivation

- Computer Aided Design: boolean operations



# Intersection of Two Line Segments

Finding the intersection of two line segments:

- Find the intersection of their two support lines
  - Linear system, two variables, two equations.
- Check whether this intersection point is between the two endpoints of each line segment. If not, the intersection is empty.
- Degenerate case: same support line. Intersection may be a line segment.

$O(1)$ time.

# Intersection of Two Line Segments

Checking whether two line segments intersect without computing the intersection point:

- 4 $CCW(\cdot)$ tests in non-degenerate cases. (How?)

# First Approach

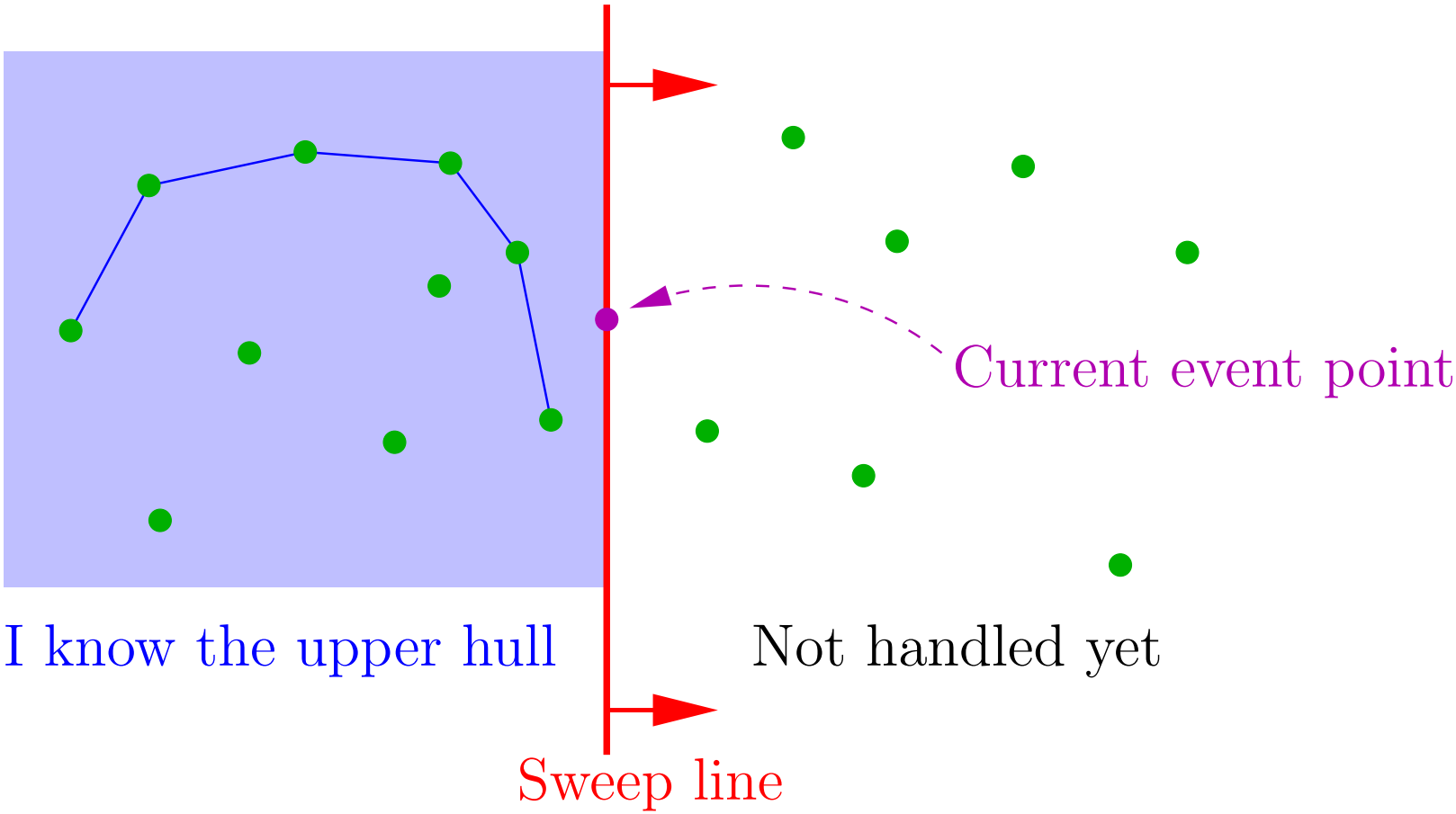
- Brute force algorithm: Check all pairs of segments for intersection.
- Running time:

$$\begin{pmatrix} n \\ 2 \end{pmatrix} \cdot \Theta(1) = \Theta(n^2)$$

- Can we do better?
  - Intersection detection:
    - ★ Yes, see later.
  - Intersection reporting:
    - ★ If all pairs intersect there are $\Omega(n^2)$ intersections, then our time bound is optimal as a function of $n$.
    - ★ So we will look for an output-sensitive algorithm.

# Plane Sweep Algorithms

- Intuition: A vertical line sweeps the plane from left to right and draws the picture.
- Example: last week's convex hull algorithm.



# Plane Sweep Algorithms

- The sweep line moves from left to right and stops at *event points*.
  - Convex hull: The event points are just the input points.
  - Sometimes they are not known from the start. (See intersection reporting.)
- We maintain invariants.
  - Convex hull: I know the upper hull of the points to the left of the sweep line.
- At each event, restore the invariants.

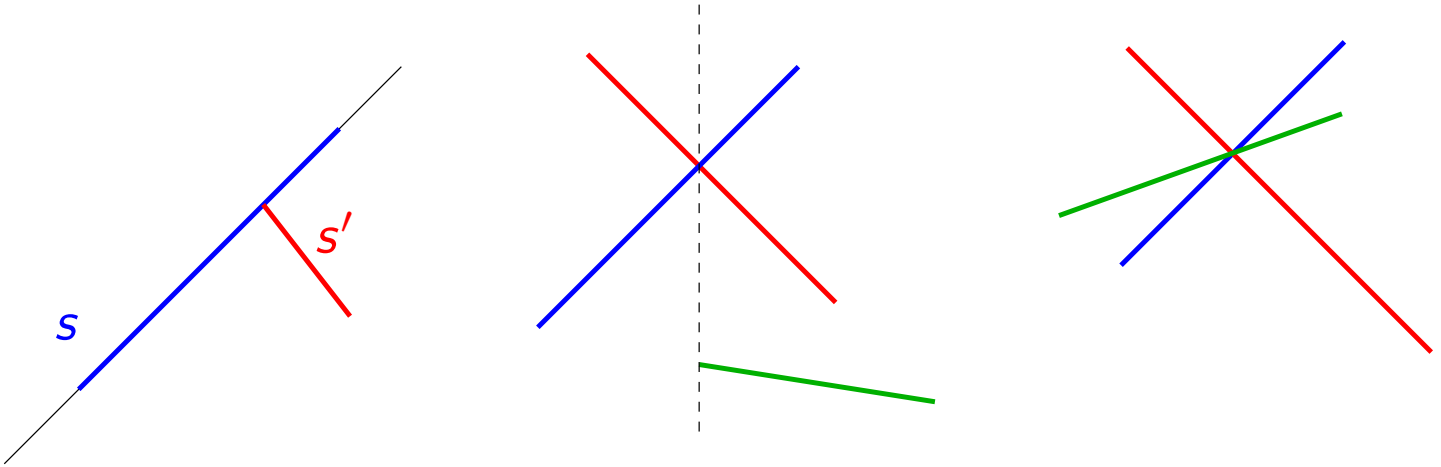
# General Position Assumptions

- No three endpoints are collinear.

Let $E$ be the set of the endpoints. Let $I$ be the set of intersection points.

- No two points in $E \cup I$ have same $x$-coordinate.
- No three segments intersect at the same point.

## Degenerate Cases



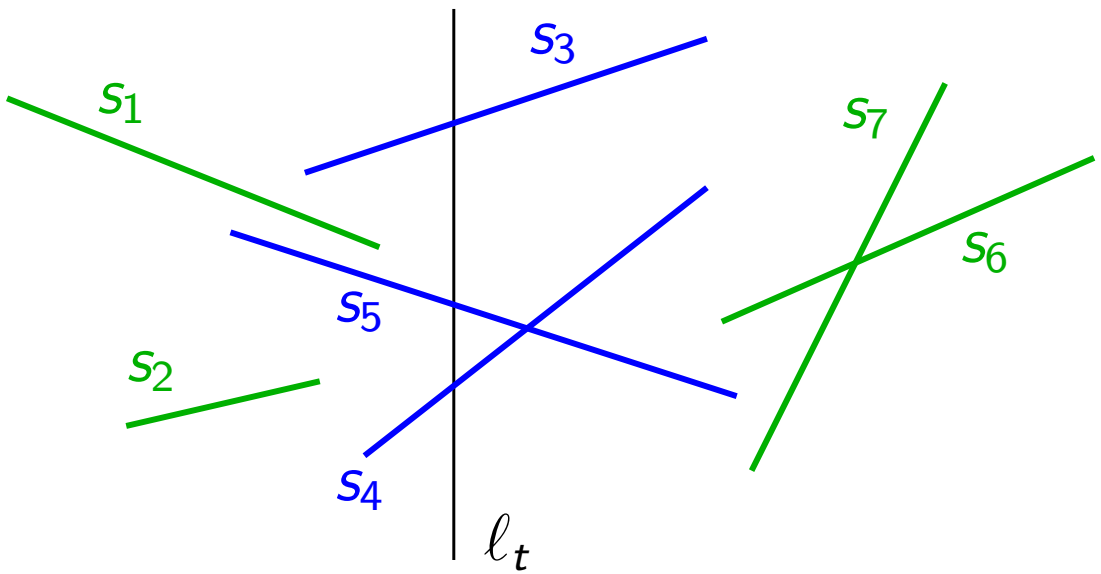

3 collinear endpoints  same x-coordinate  multiple intersection

# Intersection Detection

Plane sweep algorithm

- Let $\ell_t$ be the vertical line with equation $x = t$.
- $S_t$ is the sequence of segments that intersect $\ell_t$, ordered by the $y$-coordinates of their intersections with $\ell_t$.

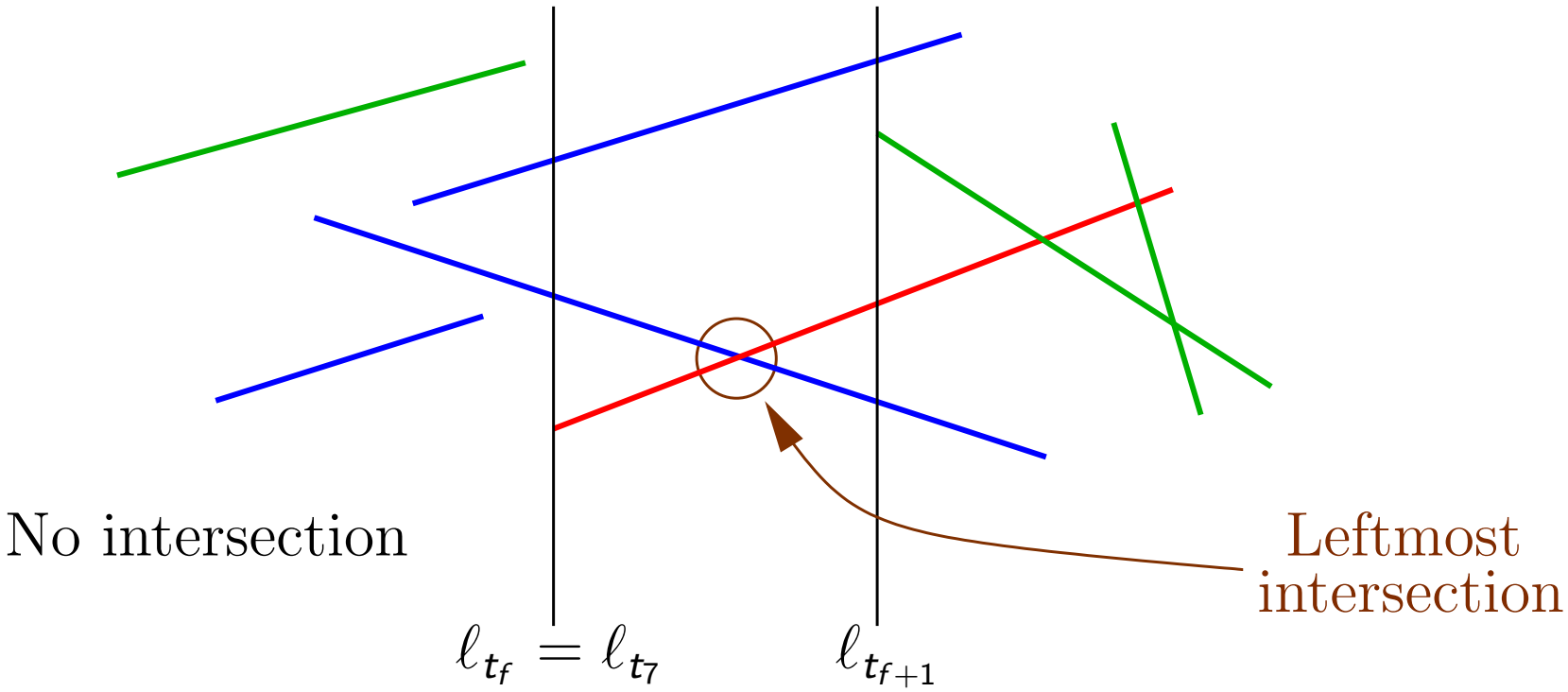
$$S_t = (s_4, s_5, s_3)$$



# Intersection Detection

- Idea: Maintain $S_t$ while $\ell_t$ moves from left to right until an intersection is found.
- Invariants:
  - There is no intersection to the left of $\ell_t$.
  - We know $S_t$.
- Let $t_1 < t_2 < \cdots < t_{2n}$ be the $x$-coordinates of the endpoints.
- $\ell_t$ stops each time $t = t_i$ for some $i$.
- Let $t = t_f$ be the last time at which the sweep-line $\ell_t$ stops.
- In other words, $f$ is the largest index such that there is no intersection point with $x$-coordinate smaller than $t_f$.

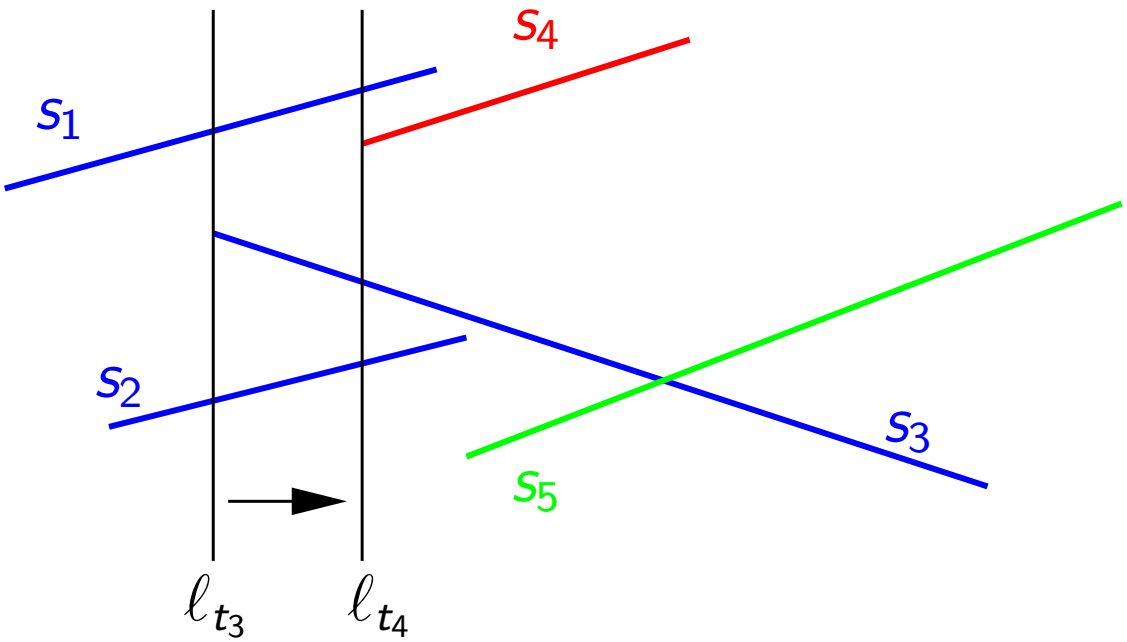
# Example




- Knowing $S_{t_i}$ for some $i < f$, we can easily find $S_{t_{i+1}}$. (See next two slides.)

# First Case: Left Endpoint

If $t_{i+1}$ corresponds to the left endpoint of $s$, insert $s$ into $S_{t_i}$ in order to obtain $S_{t_{i+1}}$.

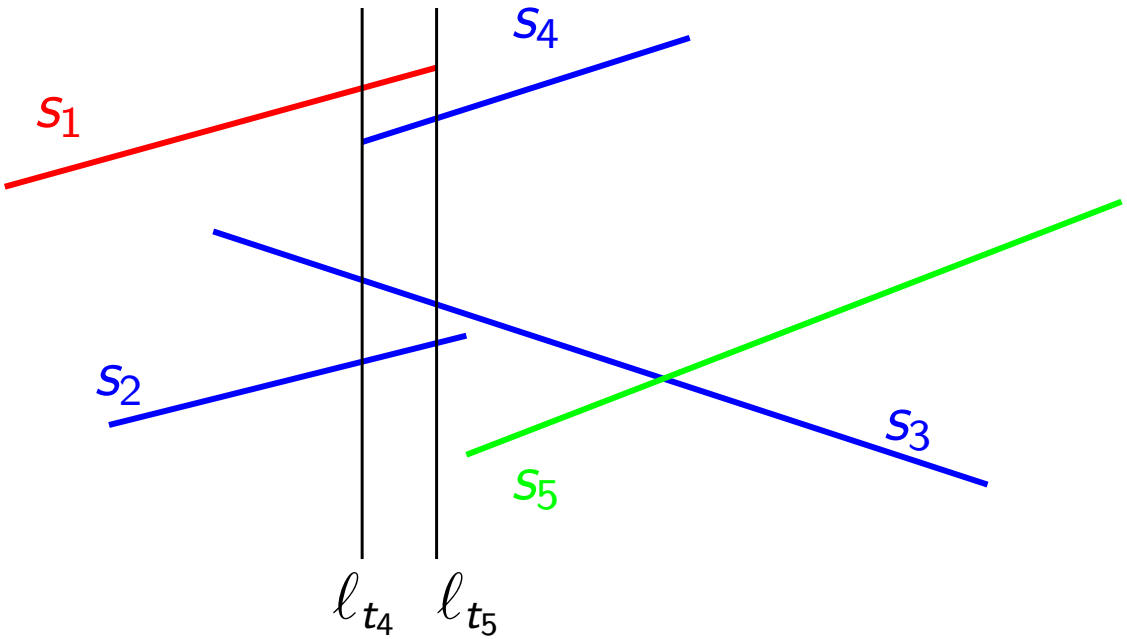



$$S_{t_3} = (s_2, s_3, s_1)$$

$$S_{t_4} = (s_2, s_3, s_4, s_1)$$

# Second Case: Right Endpoint

Delete the corresponding segment in order to obtain $S_{t_{i+1}}$.




$$S_{t_4} = (s_2, s_3, s_4, s_1)$$

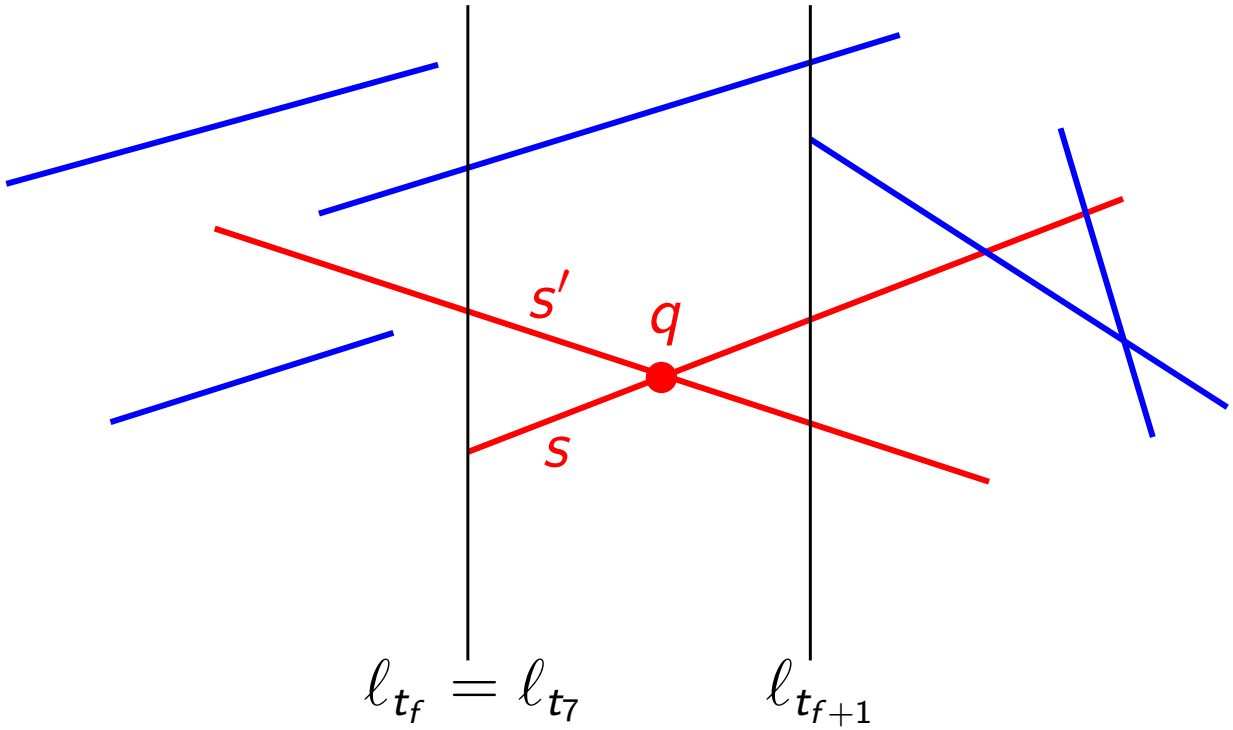
$$S_{t_5} = (s_2, s_3, s_4)$$

# Data Structure

- We maintain $S_t$ in a balanced binary search tree.
- For $i < f$, we obtain $S_{t_{i+1}}$ from $S_{t_i}$ by performing an insertion or a deletion.
- Each takes $O(\log n)$ time.
- We did not check intersections. How to do it?

# A Geometric Observation

- Let $q = s \cap s'$ be the leftmost intersection point.




- $s$ and $s'$ are adjacent in $S_{t_f}$ (proof next slide).

# Proof by Contradiction

- Assume that $S_{t_f} = (\ldots s \ldots s'' \ldots s' \ldots)$.
- The right endpoint of $s''$ cannot be to the left of $q$, by definition of $t_f$.
- If $q$ is below $s''$, then $s''$ intersect $s'$ to the left of $q$, which contradicts the fact that $q$ is the leftmost intersection.
- Similarly, $q$ cannot be above $s''$.
- We reached a contradiction.

# Checking Intersection

- We store $S_t$ in a balanced binary search tree $\mathcal{T}$, the order is the vertical order along $\ell_t$.
- Given a segment in $\mathcal{T}$, we can find in $O(\log n)$ time the next and previous segment in vertical order.
- When deleting a segment in $\mathcal{T}$, two segments $s$ and $s'$ become adjacent. We can find them in $O(\log n)$ time and check if they intersect.
- When inserting a segment $s_i$ in $\mathcal{T}$, it becomes adjacent to (at most) two segments $s$ and $s'$. Check if $s_i \cap s \neq \emptyset$ and if $s_i \cap s' \neq \emptyset$.
- In any case, if we find an intersection, we are done.

# Pseudocode

## Line Segment Intersection Detection

```
1:  procedure DetectIntersection(S)
2:      (e_1, e_2 ... e_2n) ← endpoints, ordered by increasing x-coordinate
3:      T ← empty balanced BST
4:      for i ← 1, 2n do
5:          if e_i is the left endpoint of some s ∈ S then
6:              insert s into T
7:              if s intersects pred(s) or succ(s) then
8:                  return True
9:          if e_i is the right endpoint of some s ∈ S then
10:             if next(s) intersects pred(s) then
11:                 return True
12:             delete s from T
13:     return False
```

# Analysis

- Line 1: $\Theta(n \log n)$ time by mergesort
- Line 2: $O(1)$ time
- Line 5 and 6: $O(\log n)$ time
- Loop 3–11: $O(n \log n)$ time
- So our algorithm runs in $\Theta(n \log n)$ time
- We will see later that it is optimal in some sense.

# Intersection Reporting

- Brute force: $\Theta(n^2)$ time, which is optimal in the worst case, when there are $k = \Omega(n^2)$ intersections.
- If there is $\leqslant 1$ intersection, the detection algorithm does it in $O(n \log n)$ time which is better.
- We will be able to interpolate between these two algorithms, and find an output-sensitive algorithm.

# Output Sensitive Algorithm

- Let $k = \#$intersecting pairs.
- Here $k = \#$ intersection points, since we assume general position.
- $k$ is the output size.
- Sweep line algorithm reports intersections in $O((n+k)\log n)$ time (see later).
- This is an output sensitive algorithm.
- $\Omega(n+k)$ is a lower bound
  $\Rightarrow$ nearly optimal (within an $O(\log n)$ factor).

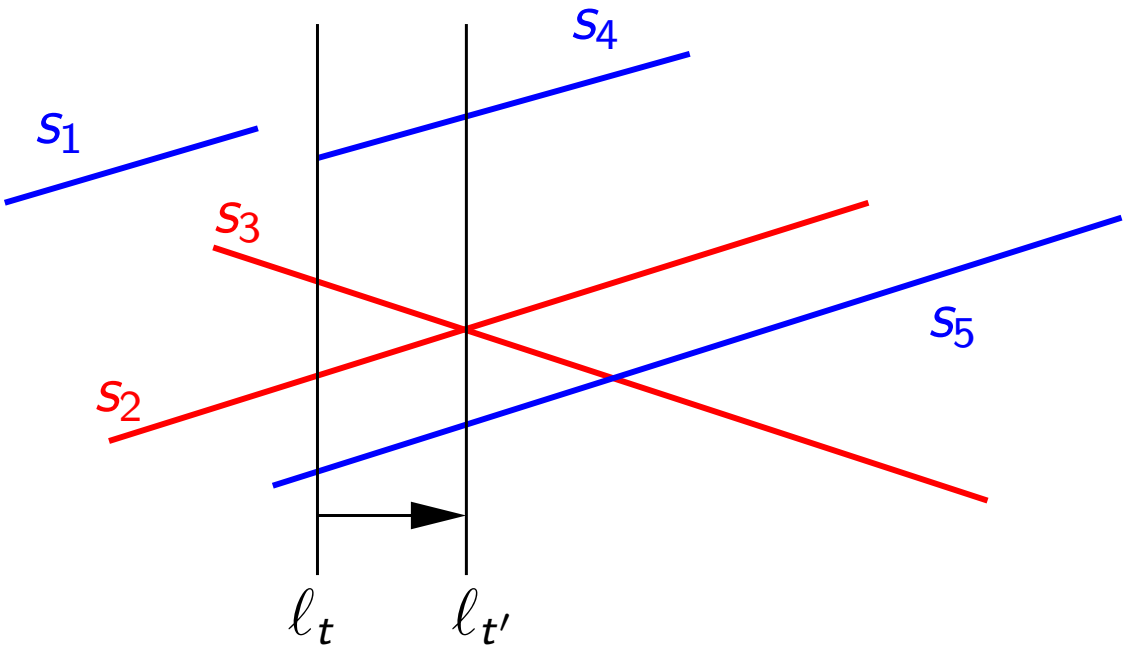
# Algorithm

- Similar to intersection detection.
- Two types of event points: endpoints and intersection points.
- We do not know intersection points in advance.
  - We cannot sort them all in advance.
  - We will use an *event queue* $\mathcal{Q}$.
- $\mathcal{Q}$ contains event points, events with smaller $x$-coordinate having higher priority.
- Implementation: a min-heap.
- We can insert an event point in $O(\log n)$ time.
- We can dequeue the event point with smallest $x$-coordinate in $O(\log n)$ time.

# Algorithm

- Initially, $\mathcal{Q}$ contains all the endpoints.
- The sweep line moves from left to right.
- It stops at each event point of $\mathcal{Q}$. When it does, we dequeue the corresponding event point.
- Each time we find an intersection, we insert it into $\mathcal{Q}$.
- When we reach an intersection point, we swap the corresponding segments in $\mathcal{T}$, and check the newly adjacent segments for intersection.

# Intersection Event




$$S_t = (s_5, s_2, s_3, s_4)$$

$$\downarrow$$

$$S_{t'} = (s_5, s_3, s_2, s_4)$$

$$\downarrow$$

Check $s_4 \cap s_2$ and $s_3 \cap s_5$

At time $t$, the event queue $\mathcal{Q}$ contains all the endpoints with abscissa larger than $t$ and the intersection point $s_2 \cap s_3$. At time $t'$, we insert $s_3 \cap s_5$ into $\mathcal{Q}$.

# Pseudocode

## Line Segment Intersection Reporting

1: **procedure** ReportIntersection($S$)
2: $\mathcal{T} \leftarrow$ empty balanced BST
3: $\mathcal{Q} \leftarrow$ new priority queue ▷ Event queue
4: Insert all endpoints into $\mathcal{Q}$
5: **while** $\mathcal{Q} \neq \emptyset$ **do**
6: $p \leftarrow \mathcal{Q}$.delete-min ▷ According to $x$-coordinate
7: **if** $p$ is the left endpoint of some $s \in S$ **then**
8: HandleLeftEndpoint($s, \mathcal{T}, \mathcal{Q}$)
9: **if** $p$ is the right endpoint of some $s \in S$ **then**
10: HandleRightEndpoint($s, \mathcal{T}, \mathcal{Q}$)
11: **if** $p = s \cap s'$ for some $s \in S$ and $s' = \mathcal{T}$.succ($s$) **then**
12: HandleIntersection($s, s', \mathcal{T}, \mathcal{Q}$)

# Pseudocode

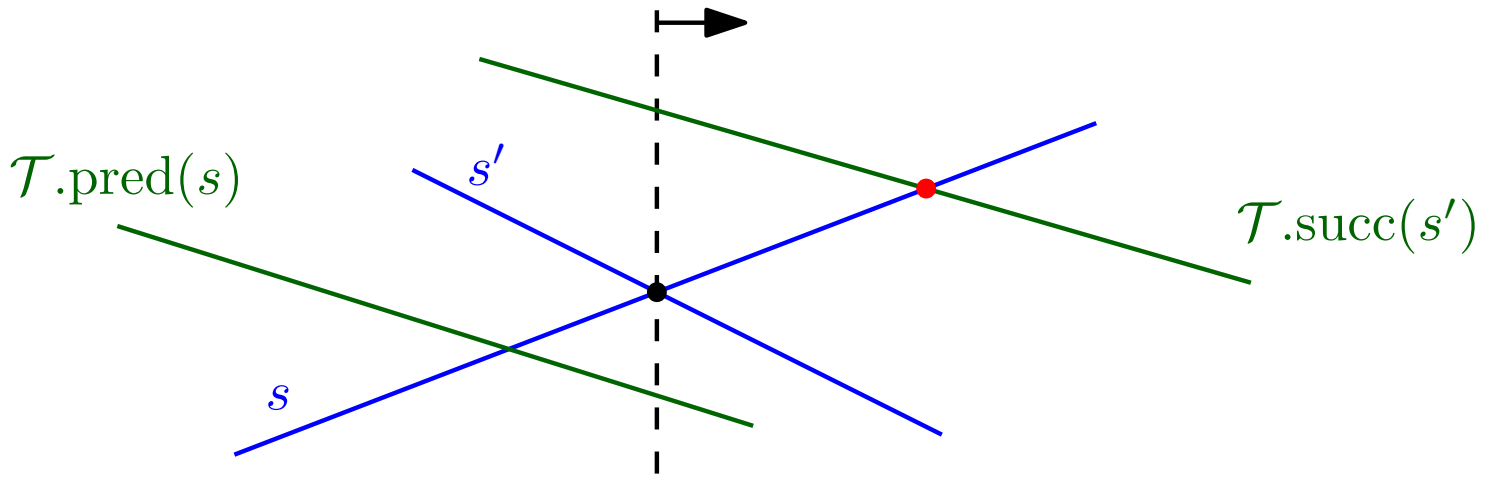
## Handling a Left Endpoint

1: **procedure** HandleLeftEndpoint($s, \mathcal{T}, \mathcal{Q}$)
2: insert $s$ into $\mathcal{T}$
3: **if** $s$ intersects $\mathcal{T}$.pred($s$) **then**
4: insert the intersection point into $\mathcal{Q}$
5: **if** $s$ intersects $\mathcal{T}$.succ($s$) **then**
6: insert the intersection point into $\mathcal{Q}$

## Handling a Right Endpoint

1: **procedure** HandleRightEndpoint($s, \mathcal{T}, \mathcal{Q}$)
2: **if** $\mathcal{T}$.pred($s$) intersects $\mathcal{T}$.succ($s$) **then**
3: insert the intersection point into $\mathcal{Q}$
4: delete $s$ from $\mathcal{T}$

# Pseudocode

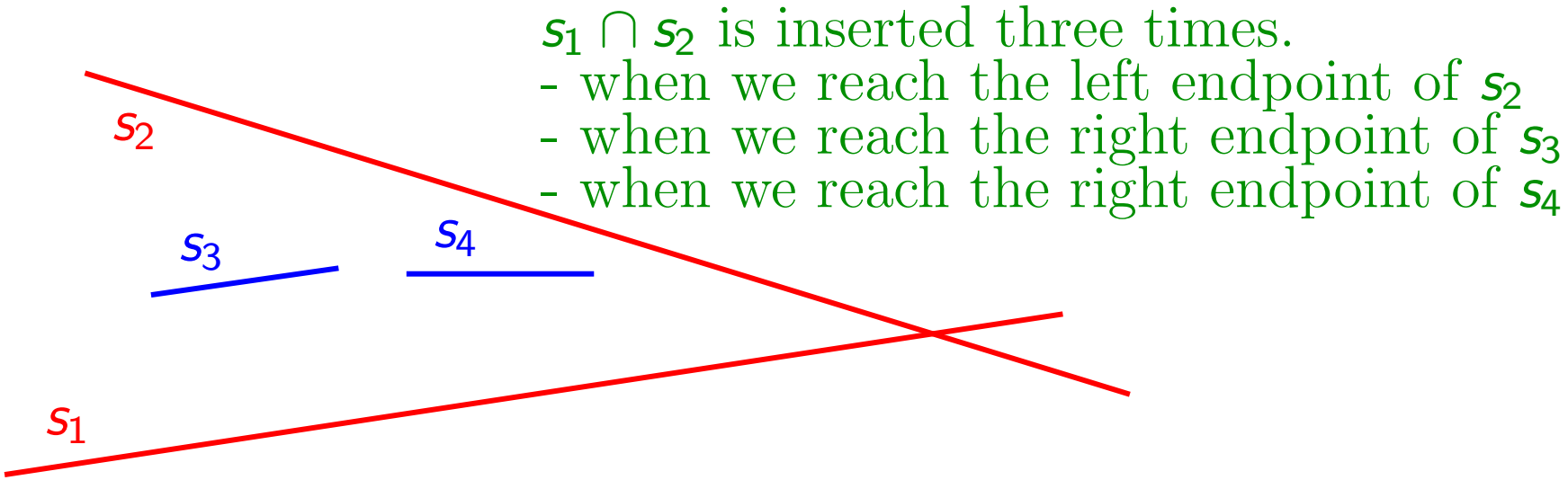



## Handling an Intersection Point

1: **procedure** HandleIntersectionPoint($s, s', \mathcal{T}, \mathcal{Q}$)
2: report $s \cap s'$
3: **if** $s$ intersects $\mathcal{T}$.succ($s'$) **then**
4: insert the intersection point into $\mathcal{Q}$
5: **if** $s'$ intersects $\mathcal{T}$.pred($s$) **then**
6: insert the intersection point into $\mathcal{Q}$
7: Swap $s$ and $s'$ within $\mathcal{T}$

# Analysis

- Each event is inserted or deleted in $O(\log(n+k))$ time.
- Problem: An intersection point may be inserted several times into $\mathcal{Q}$.




- When we reach an event point that is present several times in $\mathcal{Q}$, we just extract it repeatedly until a new event is found.

# Analysis

- Some intersection points are inserted several times into $\mathcal{Q}$. How many times does it happen?
- At each right endpoint, we may insert one crossing point that was inserted before.
- At each crossing point, we may insert two crossing points that were inserted before.
- So in total at most $3n + 3k$ events are created. ($2n$ endpoints and $n + 3k$ crossings.)
- It follows that we insert $O(n + k)$ events into $\mathcal{Q}$.
- So the algorithm runs in $O((n + k) \log(n + k))$ time.
- $\log(n + k) < \log(n + n^2) = O(\log n)$
- The running time is $O((n + k) \log n)$.

# Space Complexity

- In the worst case, $\mathcal{Q}$ contains $\Theta(n+k)$ points.
- This algorithm requires $\Theta(n+k)$ *space*.
- Can we do better?
- Yes: At time $t$, only keep intersections between segments that are adjacent in $S_t$.
- Then the algorithm requires only $\Theta(n)$ space.

# Conclusion

- We introduced a new technique: *plane sweep*.
- Idea: a sweep line moves from left to right and draws the picture.
- It stops at a finite set of event points, where the data structure is updated.
- The event points are stored in a priority queue.
- It can be used for other problems. For instance:
  - Map overlay: compute a full description of the map.
  - Fixed-radius near neighbor searching in 2D.