Yan-Bin Jia, Iowa State University

# Line Segment Intersection

## Outline

I. Intersecting two segments

II. Intersecting $n$ segment via a plane sweep

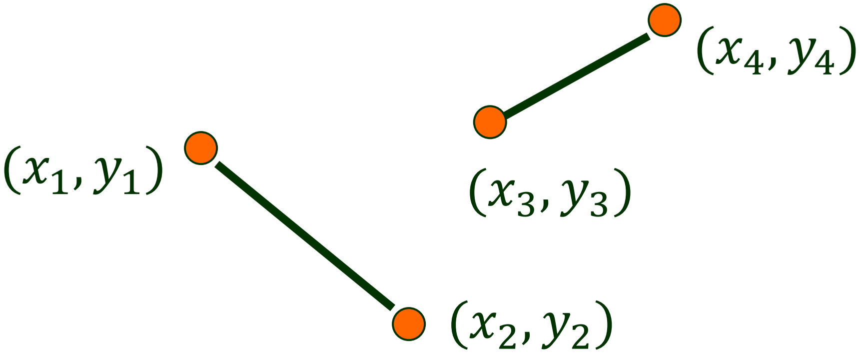
III. Data structures for the algorithm

IV. Running time and storage

Textbook: *Computational Geometry: Algorithms and Applications (*3rd ed.) by M. de Berg et al., Springer-Verlag, 2008.

# I. Two Segments Intersect?

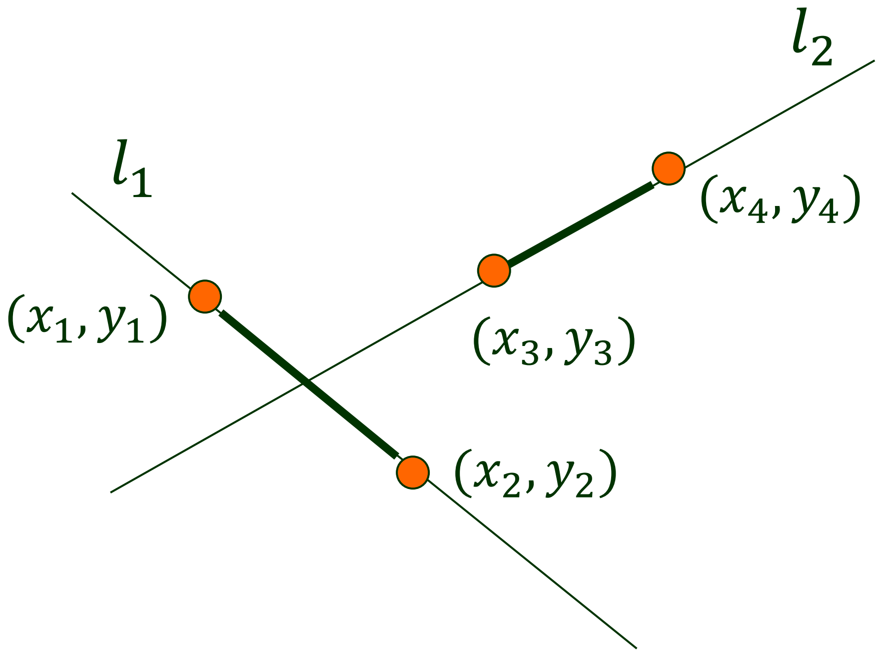
Straightforward approach:



# I. Two Segments Intersect?

Straightforward approach:

1. Construct the lines $l_1$ and $l_2$ containing the two segments.

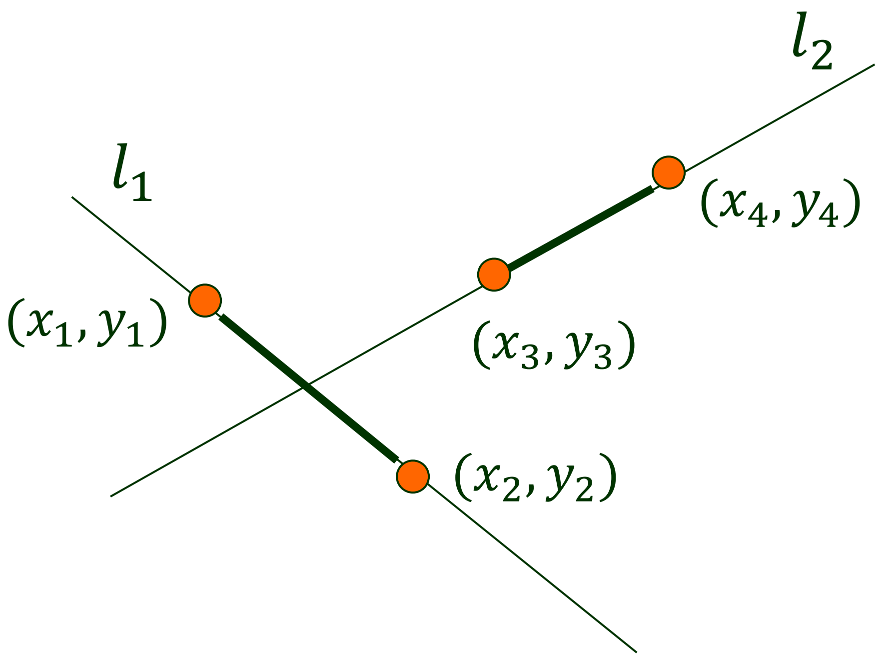


# I. Two Segments Intersect?

Straightforward approach:

1. Construct the lines $l_1$ and $l_2$ containing the two segments.

$$l_1 : (a_1, b_1, c_1) = (x_1, y_1, 1) \times (x_2, y_2, 1) \qquad // \ a_1 x + b_1 y + c_1 = 0$$



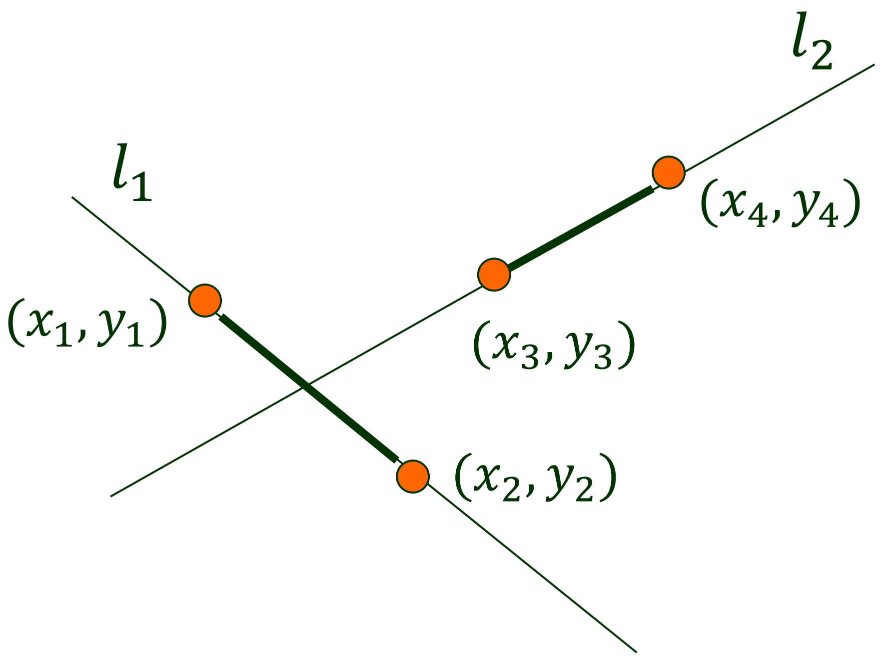
# I. Two Segments Intersect?

Straightforward approach:

1. Construct the lines $l_1$ and $l_2$ containing the two segments.

$$l_1: (a_1, b_1, c_1) = (x_1, y_1, 1) \times (x_2, y_2, 1) \qquad // \ a_1 x + b_1 y + c_1 = 0$$

$$l_2: (a_2, b_2, c_2) = (x_3, y_3, 1) \times (x_4, y_4, 1) \qquad // \ a_2 x + b_2 y + c_2 = 0$$



# I. Two Segments Intersect?

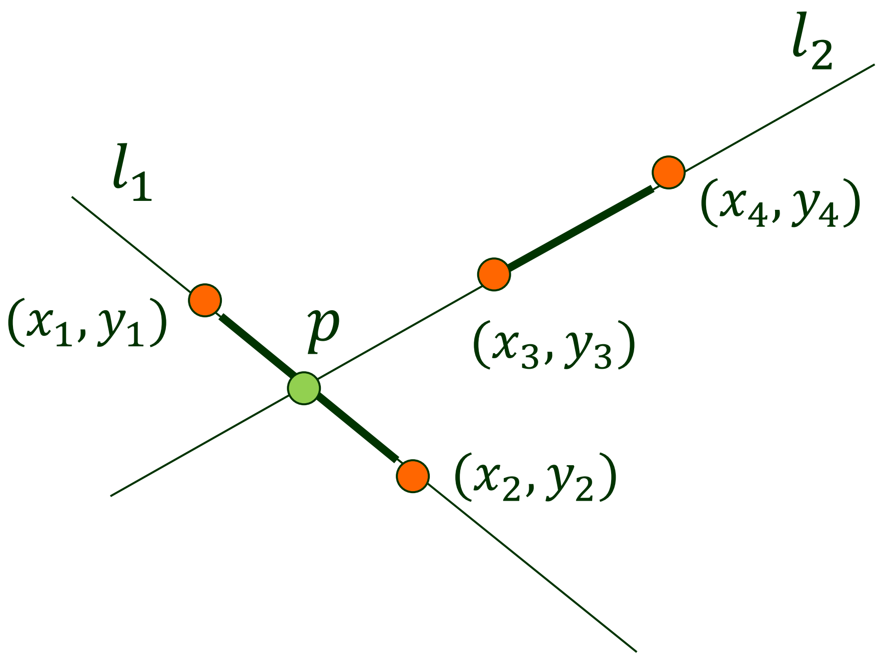
Straightforward approach:

1. Construct the lines $l_1$ and $l_2$ containing the two segments.

$$l_1: (a_1, b_1, c_1) = (x_1, y_1, 1) \times (x_2, y_2, 1) \qquad // \; a_1 x + b_1 y + c_1 = 0$$

$$l_2: (a_2, b_2, c_2) = (x_3, y_3, 1) \times (x_4, y_4, 1) \qquad // \; a_2 x + b_2 y + c_2 = 0$$

2. Find their intersection point $p$.



# I. Two Segments Intersect?

Straightforward approach:

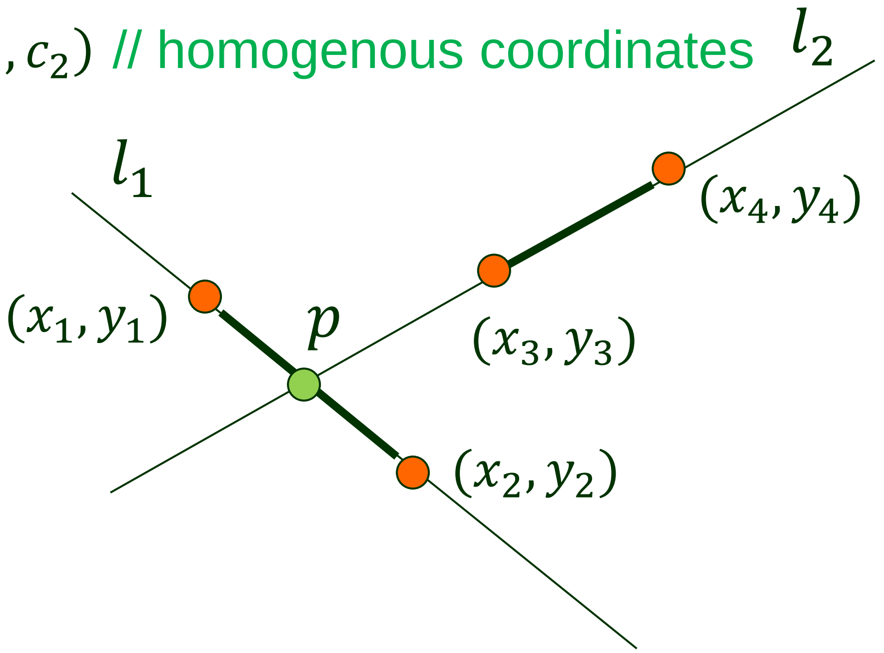
1. Construct the lines $l_1$ and $l_2$ containing the two segments.

$$l_1: (a_1, b_1, c_1) = (x_1, y_1, 1) \times (x_2, y_2, 1) \qquad // \; a_1 x + b_1 y + c_1 = 0$$

$$l_2: (a_2, b_2, c_2) = (x_3, y_3, 1) \times (x_4, y_4, 1) \qquad // \; a_2 x + b_2 y + c_2 = 0$$

2. Find their intersection point $p$.

$p$: $(a_1, b_1, c_1) \times (a_2, b_2, c_2)$ // homogenous coordinates



# I. Two Segments Intersect?

Straightforward approach:

1. Construct the lines $l_1$ and $l_2$ containing the two segments.

$$l_1: (a_1, b_1, c_1) = (x_1, y_1, 1) \times (x_2, y_2, 1) \qquad // \ a_1 x + b_1 y + c_1 = 0$$

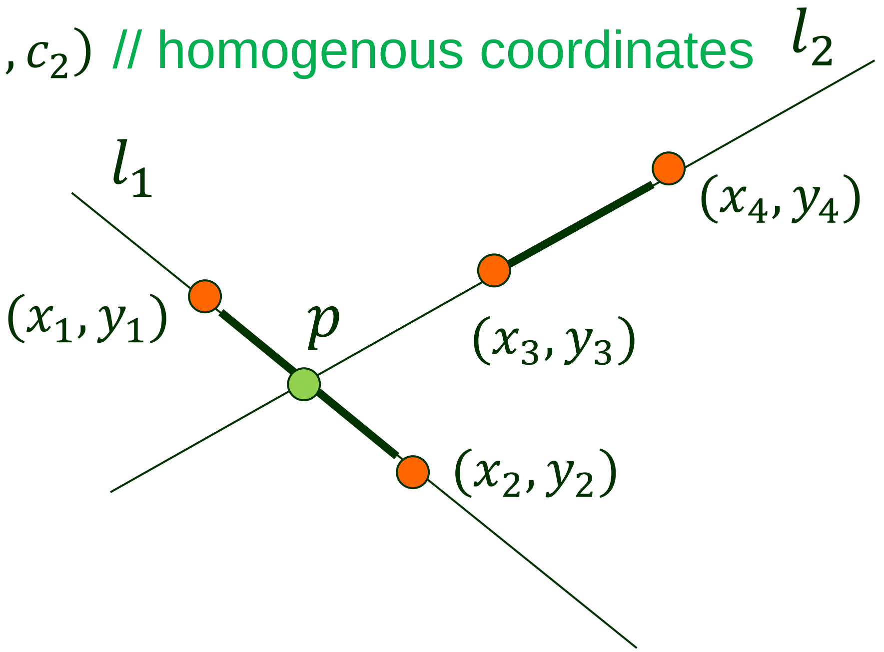
$$l_2: (a_2, b_2, c_2) = (x_3, y_3, 1) \times (x_4, y_4, 1) \qquad // \ a_2 x + b_2 y + c_2 = 0$$

2. Find their intersection point $p$.

$p$: $(a_1, b_1, c_1) \times (a_2, b_2, c_2)$ // homogenous coordinates

⇩

$$p = (p_x, p_y)$$



# I. Two Segments Intersect?

Straightforward approach:

1. Construct the lines $l_1$ and $l_2$ containing the two segments.

$$l_1 : (a_1, b_1, c_1) = (x_1, y_1, 1) \times (x_2, y_2, 1) \qquad // \; a_1 x + b_1 y + c_1 = 0$$

$$l_2 : (a_2, b_2, c_2) = (x_3, y_3, 1) \times (x_4, y_4, 1) \qquad // \; a_2 x + b_2 y + c_2 = 0$$

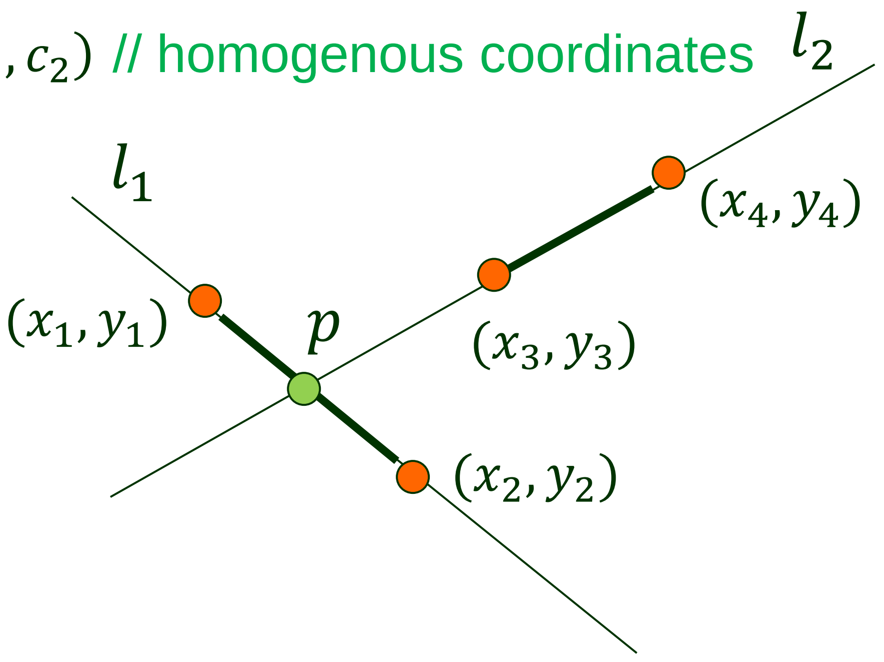
2. Find their intersection point $p$.

$p$: $(a_1, b_1, c_1) \times (a_2, b_2, c_2)$ // homogenous coordinates

$\Downarrow$

$$p = (p_x, p_y)$$

3. Check if $p$ lies on both segments, i.e., if both conditions below hold:



# I. Two Segments Intersect?

Straightforward approach:

1. Construct the lines $l_1$ and $l_2$ containing the two segments.

$$l_1: (a_1, b_1, c_1) = (x_1, y_1, 1) \times (x_2, y_2, 1) \qquad // \; a_1 x + b_1 y + c_1 = 0$$

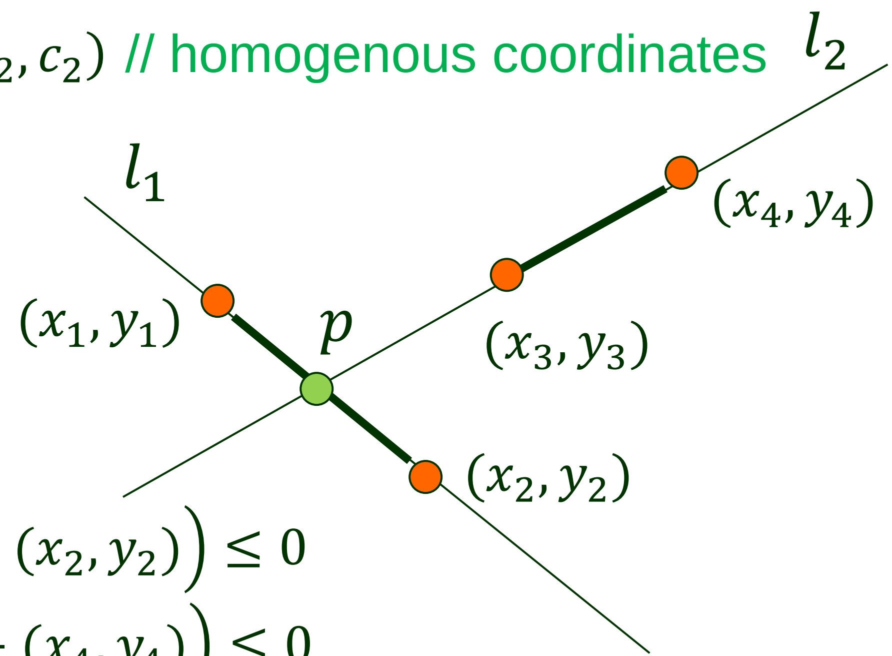
$$l_2: (a_2, b_2, c_2) = (x_3, y_3, 1) \times (x_4, y_4, 1) \qquad // \; a_2 x + b_2 y + c_2 = 0$$

2. Find their intersection point $p$.

$$p: \; (a_1, b_1, c_1) \times (a_2, b_2, c_2) \quad \text{// homogenous coordinates}$$

$$\Downarrow$$

$$p = (p_x, p_y)$$




3. Check if $p$ lies on both segments, i.e., if both conditions below hold:

$$\big((p_x, p_y) - (x_1, y_1)\big) \cdot \big((p_x, p_y) - (x_2, y_2)\big) \le 0$$

$$\big((p_x, p_y) - (x_3, y_3)\big) \cdot \big((p_x, p_y) - (x_4, y_4)\big) \le 0$$

# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

# Quick Rejection

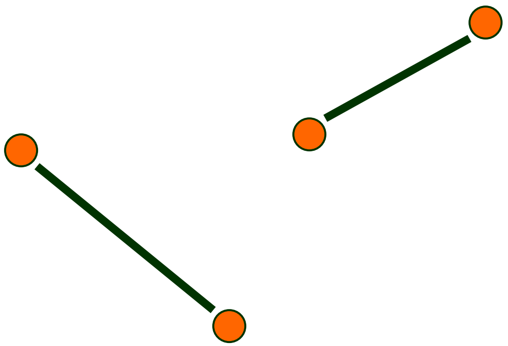
In practice, the two input segments often do ***not*** intersect.

**Stage 1**: quick rejection if their bounding boxes do not intersect

# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

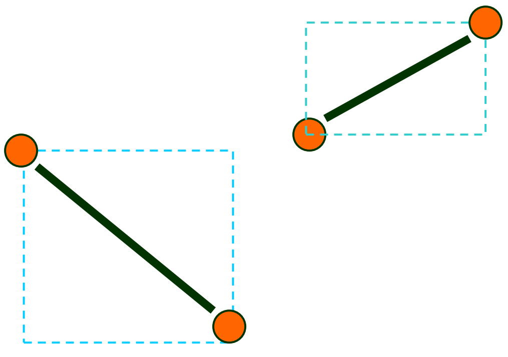
**Stage 1**: quick rejection if their bounding boxes do not intersect



# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

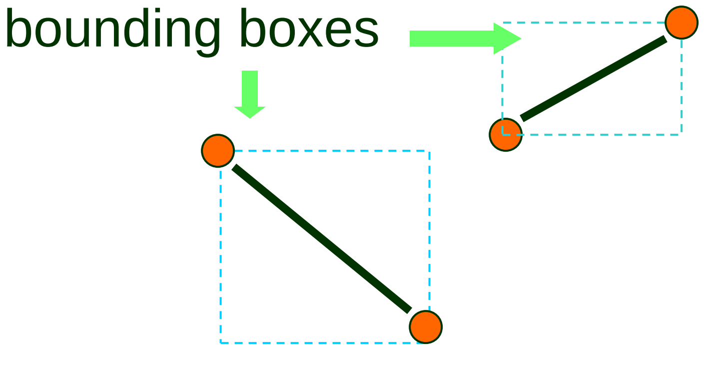
**Stage 1**: quick rejection if their bounding boxes do not intersect



# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

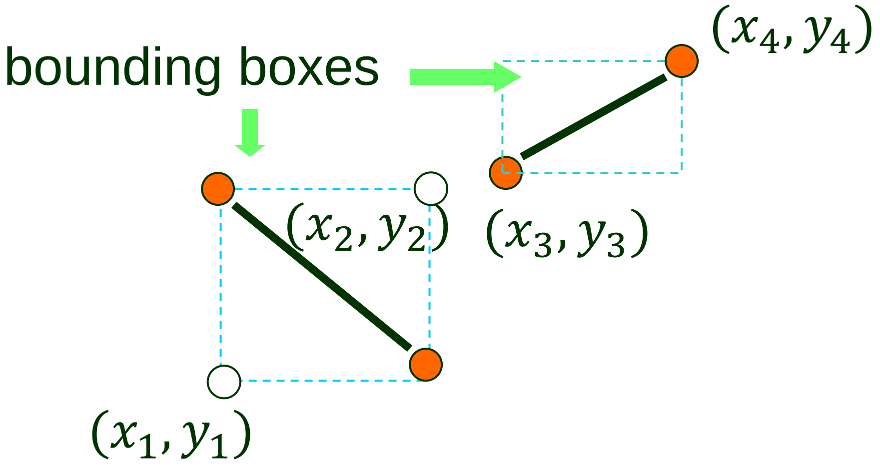
**Stage 1**: quick rejection if their bounding boxes do not intersect



# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

**Stage 1**: quick rejection if their bounding boxes do not intersect

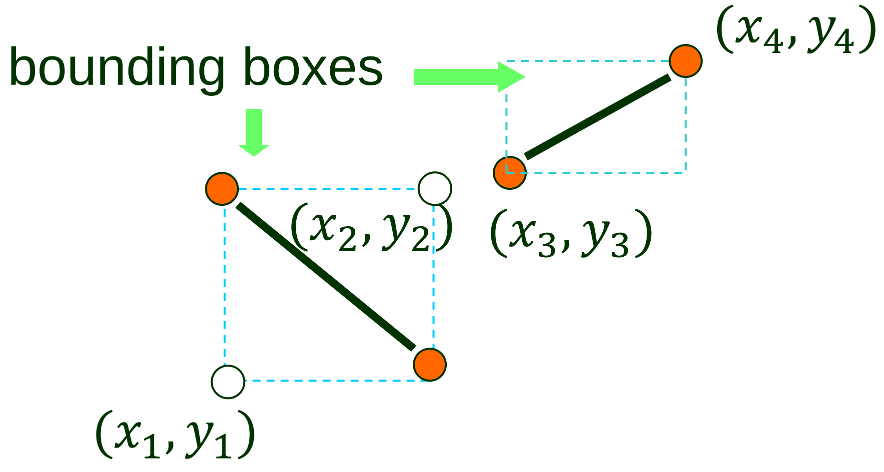


# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

**Stage 1**: quick rejection if their bounding boxes do not intersect

if and only if $x_4 < x_1 \lor x_3 > x_2 \lor y_4 < y_1 \lor y_3 > y_2$

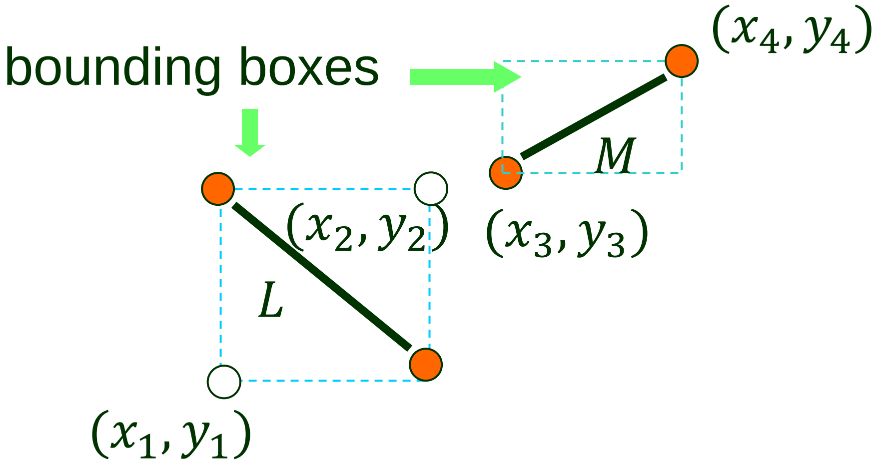


# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

**Stage 1**: quick rejection if their bounding boxes do not intersect

if and only if $x_4 < x_1 \lor x_3 > x_2 \lor y_4 < y_1 \lor y_3 > y_2$



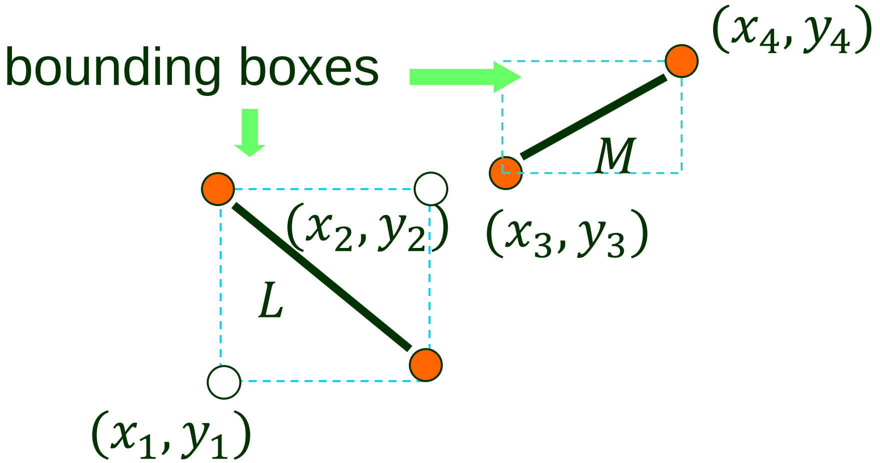
# Quick Rejection

In practice, the two input segments often do ***not*** intersect.

**Stage 1**: quick rejection if their bounding boxes do not intersect

if and only if $x_4 < x_1 \vee x_3 > x_2 \vee y_4 < y_1 \vee y_3 > y_2$

$L$ right of $M$?  $L$ left of $M$?  $L$ above $M$?  $L$ below $M$?



# Quick Rejection

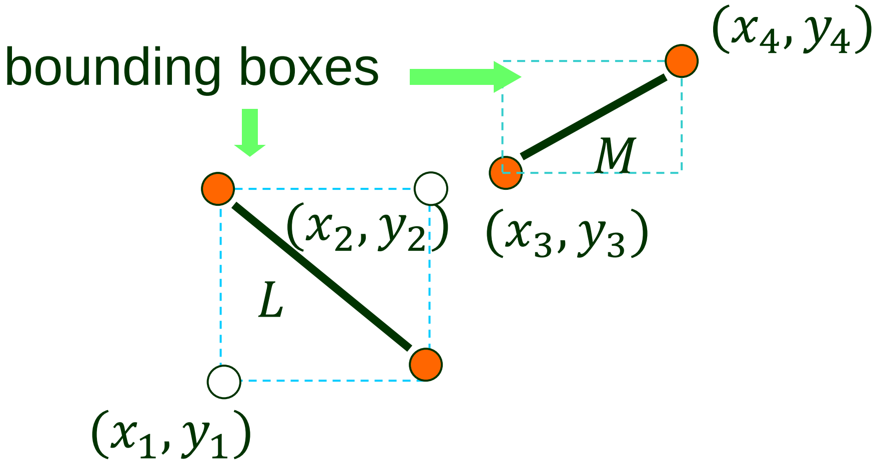
In practice, the two input segments often do ***not*** intersect.

**Stage 1**: quick rejection if their bounding boxes do not intersect

if and only if $x_4 < x_1 \lor x_3 > x_2 \lor y_4 < y_1 \lor y_3 > y_2$

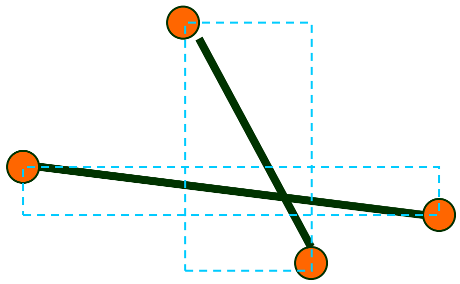
$L$ right of $M$? $L$ left of $M$? $L$ above $M$? $L$ below $M$?

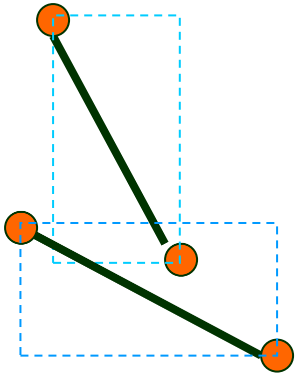
**Case 1**: bounding boxes do not intersect; neither will the segments**.**



# Bounding Box

**Case 2**: Bounding boxes intersect; the segments may or may not intersect. Needs to be further checked in Stage 2.

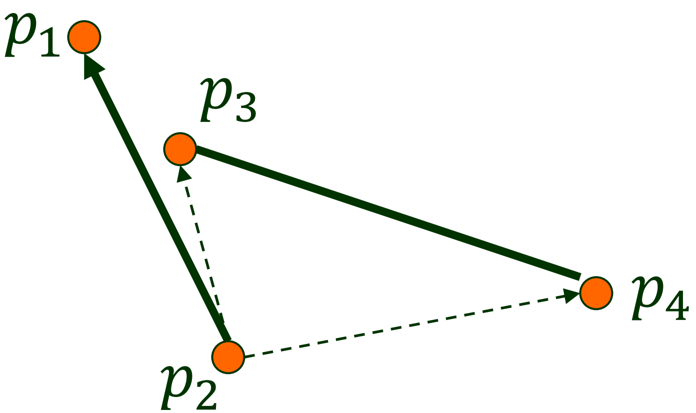




# Necessary and Sufficient Condition for No Intersection

Two line segments do ***not*** intersect if and only if

*one of them lies entirely to one side of the line containing the other one.*




The above condition is equivalent to that

*at least one of the two pairs of cross products below has the same sign:*

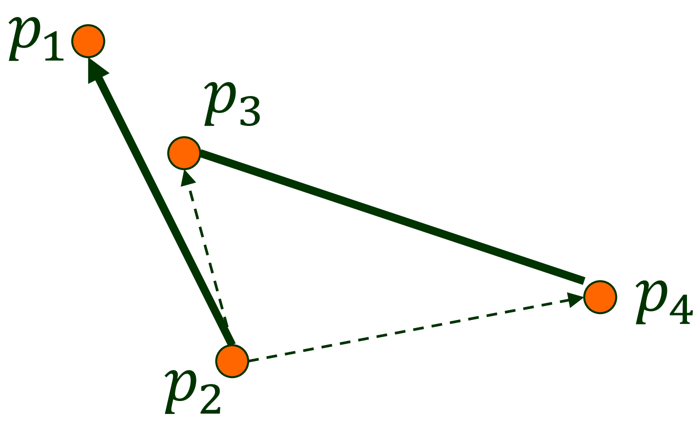
$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$

$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$

# Necessary and Sufficient Condition for No Intersection

Two line segments do ***not*** intersect if and only if

*one of them lies entirely to one side of the line containing the other one*.




$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$ are both positive!

The above condition is equivalent to that

*at least one of the two pairs of cross products below has the same sign:*

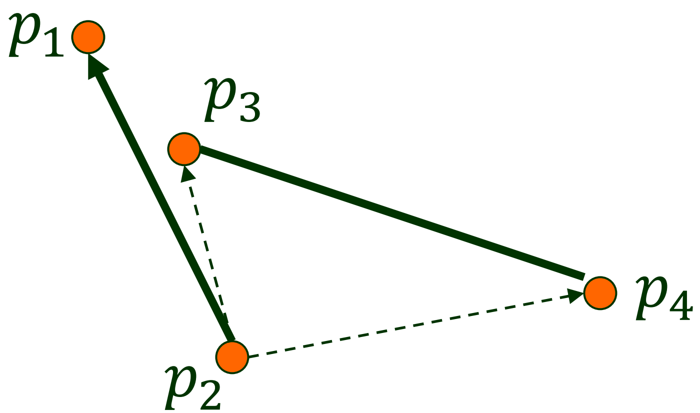
$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$

$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$

# Necessary and Sufficient Condition for No Intersection

Two line segments do ***not*** intersect if and only if

*one of them lies entirely to one side of the line containing the other one*.




$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$ are both positive!

The above condition is equivalent to that

*at least one of the two pairs of cross products below has the same sign:*

$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$

$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$

Two line segments intersect if and only if the condition is false.

# When the Necessary & Sufficient Condition is False

The cross products in either pair have different signs (or at least one cross product in the pair is 0).

$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$ // the line through $p_3, p_4$ // intersects $\overline{p_1 p_2}$.

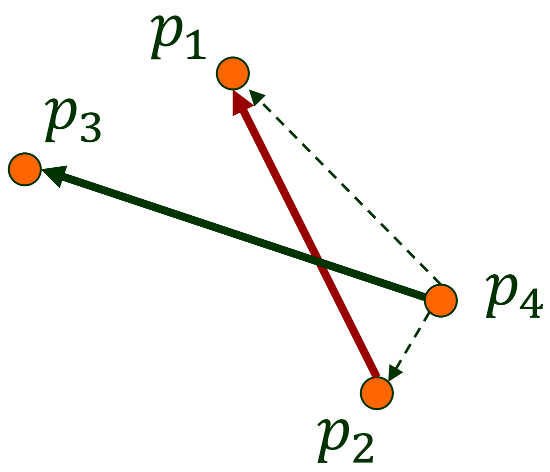
$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$ // the line through $p_1, p_2$ // intersects $\overline{p_3 p_4}$.

# When the Necessary & Sufficient Condition is False

The cross products in either pair have different signs (or at least one cross product in the pair is 0).

$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$ // the line through $p_3, p_4$ // intersects $\overline{p_1 p_2}$.

$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$ // the line through $p_1, p_2$ // intersects $\overline{p_3 p_4}$.

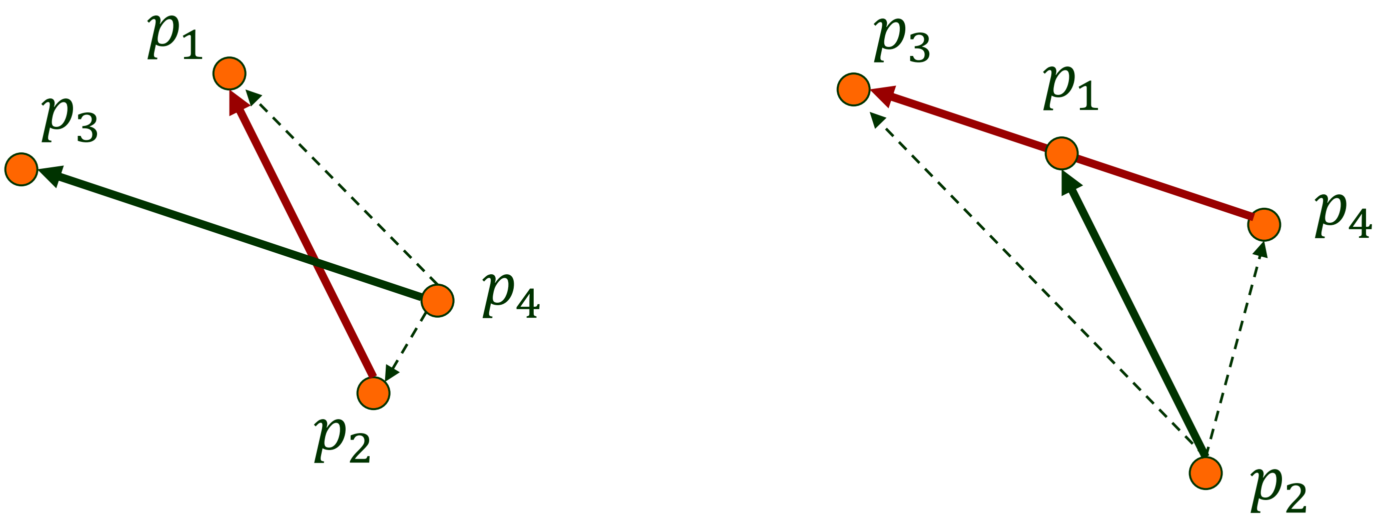


# When the Necessary & Sufficient Condition is False

The cross products in either pair have different signs (or at least one cross product in the pair is 0).

$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$ // the line through $p_3, p_4$ // intersects $\overline{p_1 p_2}$.

$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$ // the line through $p_1, p_2$ // intersects $\overline{p_3 p_4}$.

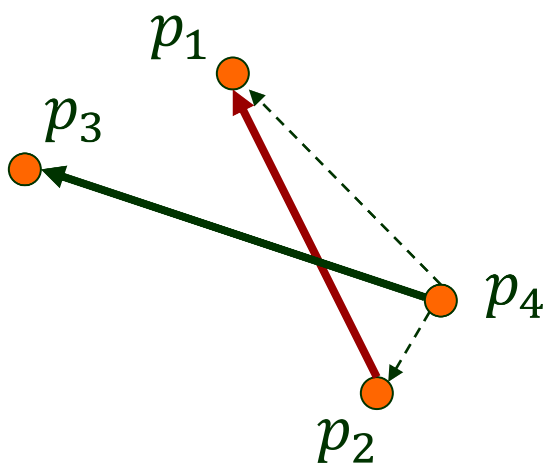


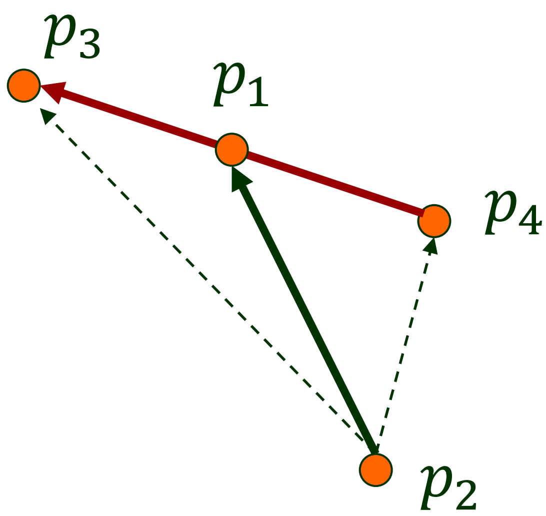
# When the Necessary & Sufficient Condition is False

The cross products in either pair have different signs (or at least one cross product in the pair is 0).

$(p_3 - p_2) \times (p_1 - p_2)$ and $(p_4 - p_2) \times (p_1 - p_2)$ // the line through $p_3, p_4$ // intersects $\overline{p_1 p_2}$.

$(p_1 - p_4) \times (p_3 - p_4)$ and $(p_2 - p_4) \times (p_3 - p_4)$ // the line through $p_1, p_2$ // intersects $\overline{p_3 p_4}$.



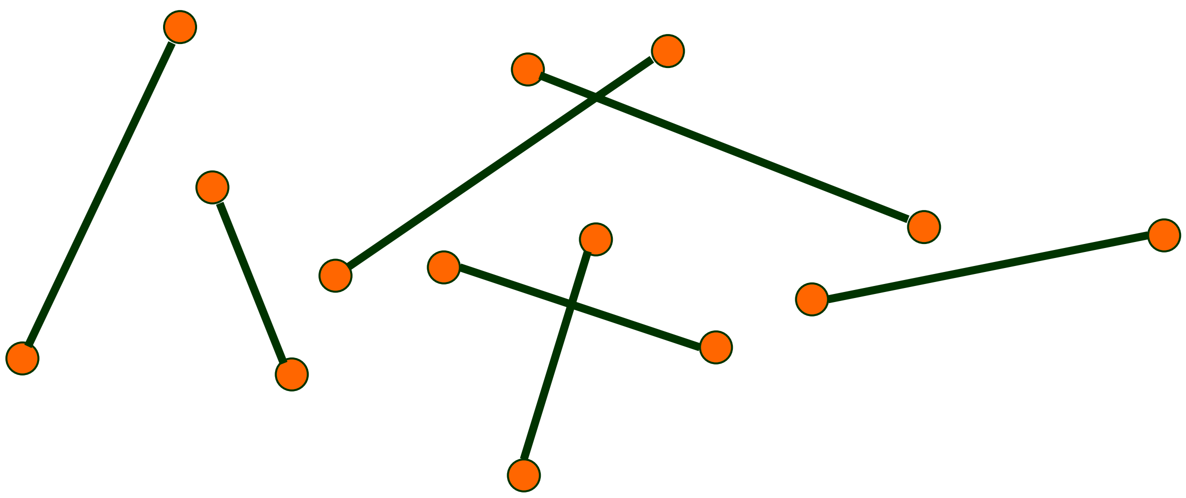




Not as convenient as testing the falsity of non-intersecting of the two segments.

# II. $n$ Line Segments

**Input:** a set of $n$ line segments in the plane.

**Output**: all intersections and for each intersection the involved segments.



# A Brute-Force Algorithm

Simply take every pair of segments, and check if they intersect.
If so, output the intersection.

# A Brute-Force Algorithm

Simply take every pair of segments, and check if they intersect.
If so, output the intersection.

Running time $\Theta(n^2)$.

# A Brute-Force Algorithm

Simply take every pair of segments, and check if they intersect.
If so, output the intersection.

Running time $\Theta(n^2)$.

Nevertheless, *sparse distribution* in practice:

# A Brute-Force Algorithm

Simply take every pair of segments, and check if they intersect.
If so, output the intersection.

Running time $\Theta(n^2)$.

Nevertheless, *sparse distribution* in practice:

Most segments do not intersect, or if they do,
only with a few other segments.

# A Brute-Force Algorithm

Simply take every pair of segments, and check if they intersect.
If so, output the intersection.

Running time $\Theta(n^2)$.

Nevertheless, *sparse distribution* in practice:

Most segments do not intersect, or if they do, only with a few other segments.

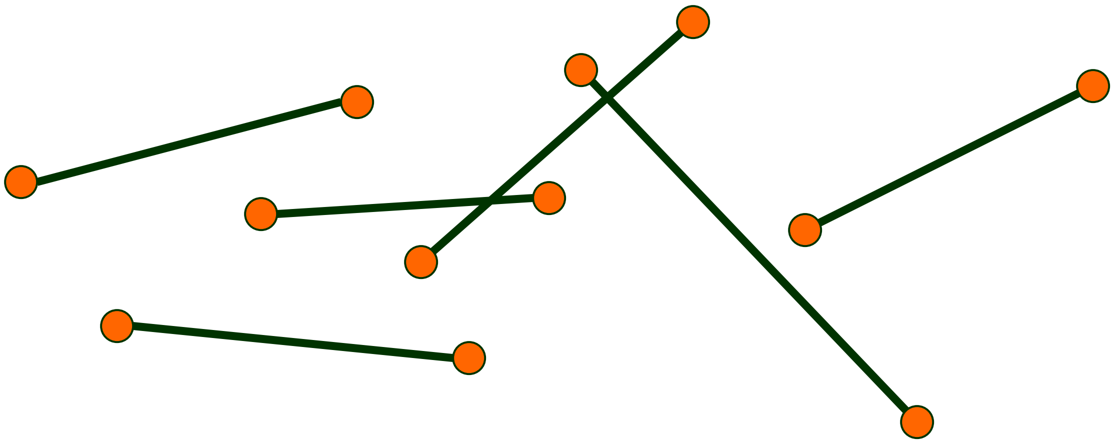
Need a faster algorithm to deal with such situations!

# A Plane Sweep Algorithm

Avoid testing pairs of segments that are *far apart*.

# A Plane Sweep Algorithm

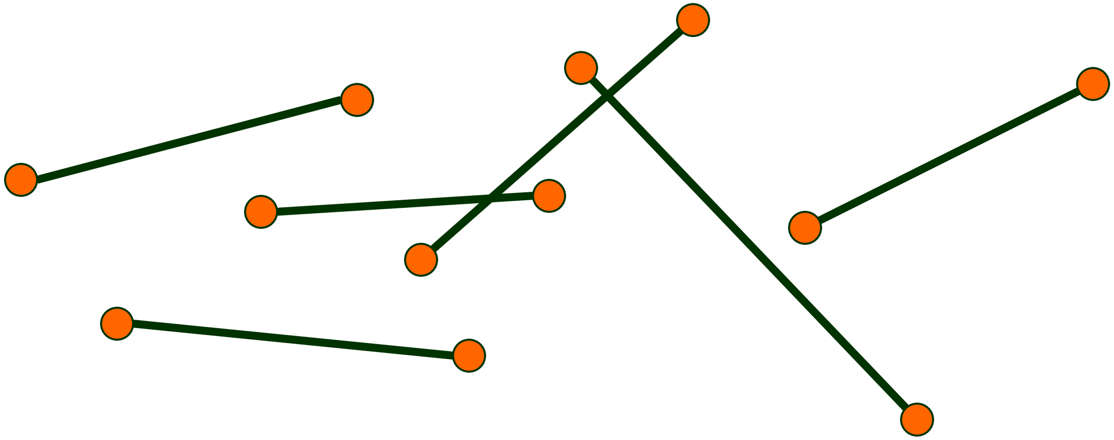
Avoid testing pairs of segments that are *far apart*.



# A Plane Sweep Algorithm

Avoid testing pairs of segments that are *far apart*.

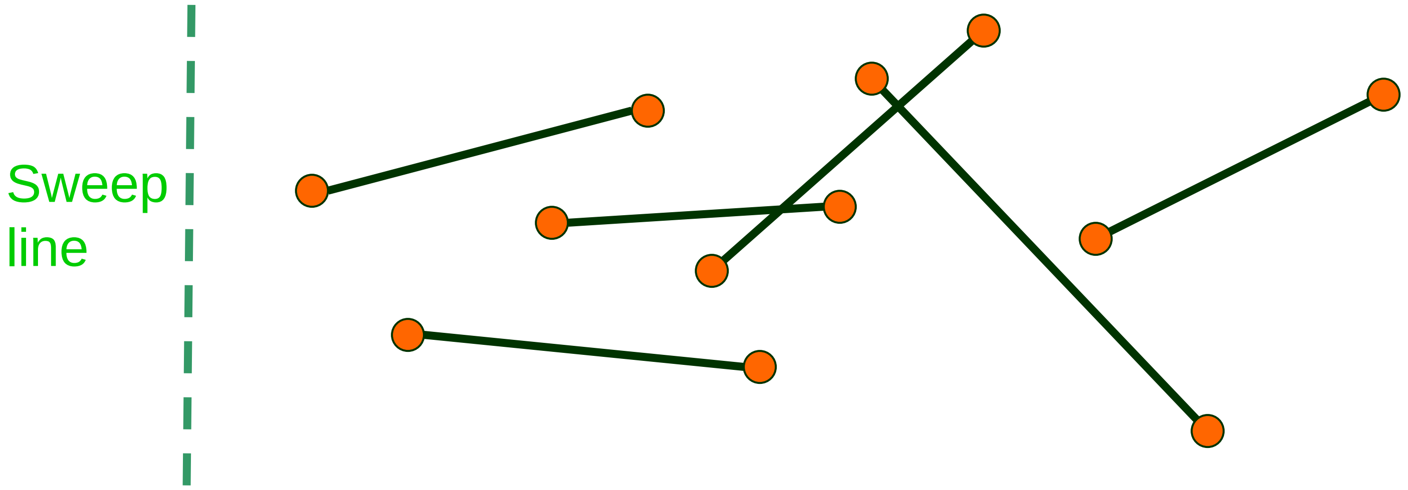
**Idea**: *Imagine* a vertical sweep line passes through the given set of line segments, from left to right.



# A Plane Sweep Algorithm

Avoid testing pairs of segments that are *far apart*.

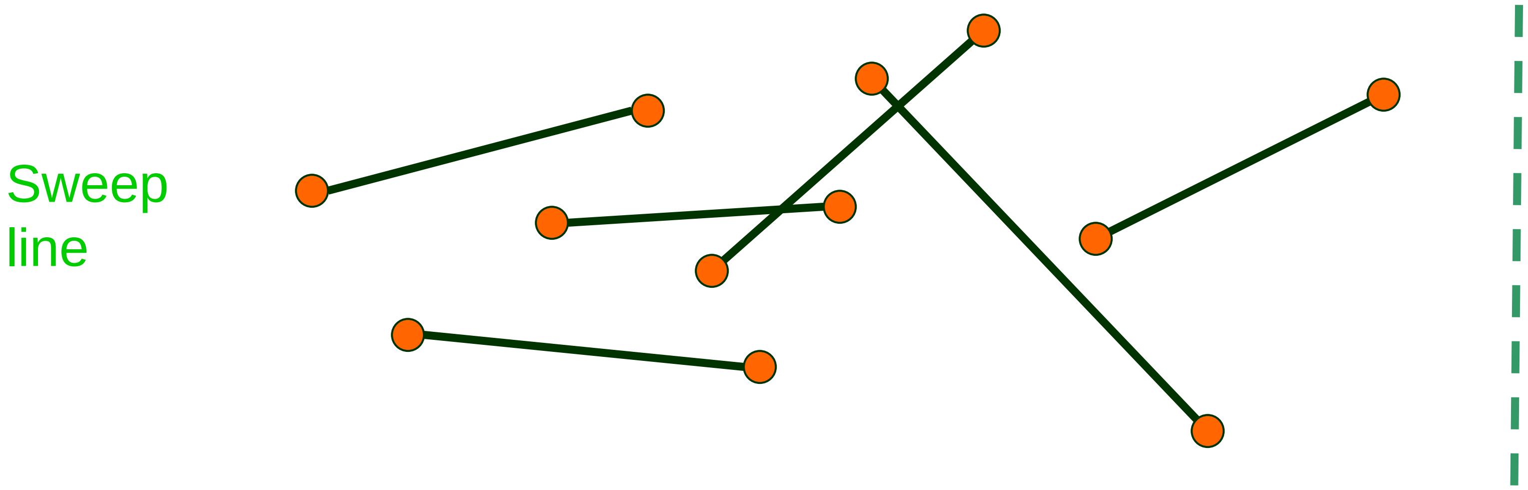
**Idea**: *Imagine* a vertical sweep line passes through the given set of line segments, from left to right.



# A Plane Sweep Algorithm

Avoid testing pairs of segments that are *far apart*.

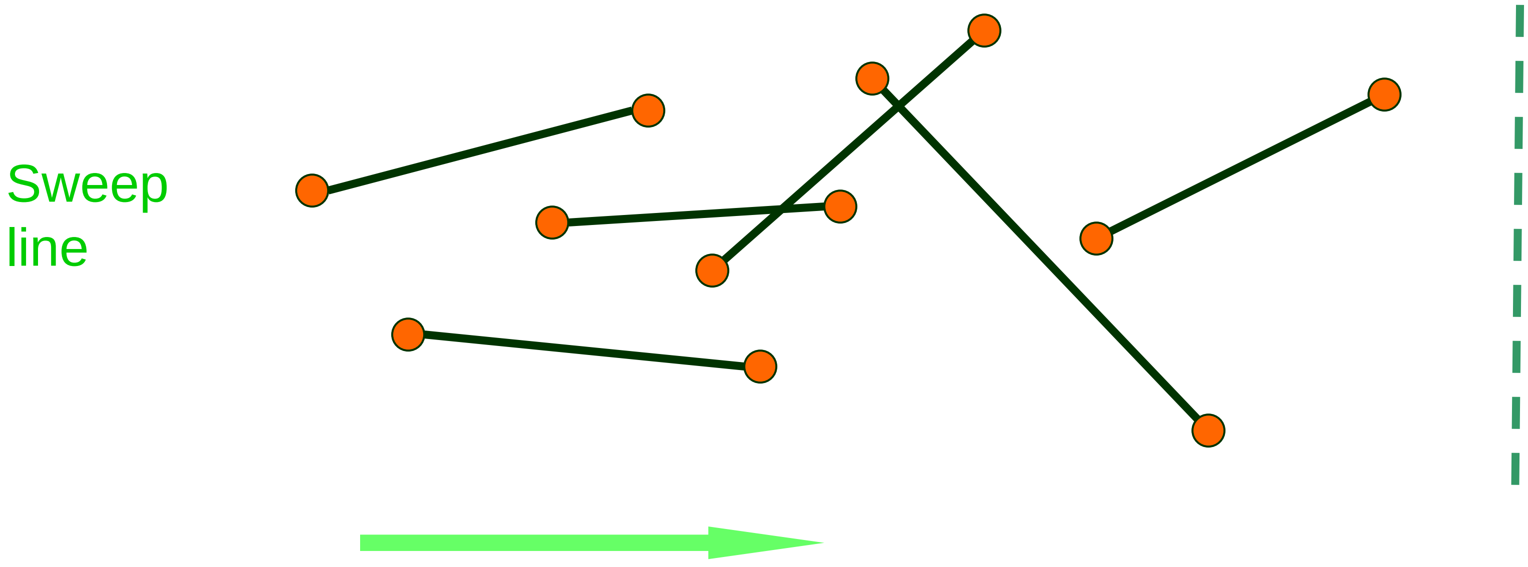
**Idea**: *Imagine* a vertical sweep line passes through the given set of line segments, from left to right.



# A Plane Sweep Algorithm

Avoid testing pairs of segments that are *far apart*.

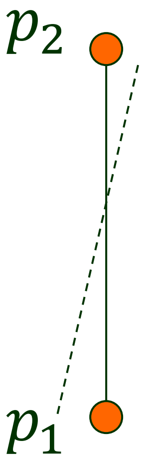
**Idea**: *Imagine* a vertical sweep line passes through the given set of line segments, from left to right.



# Handling Non-degeneracy

If $\geq 1$ vertical segment, imagine all segments are rotated clockwise by a tiny angle and then test for intersection.

For each vertical segment, the sweep line will hit its lower endpoint before upper point.

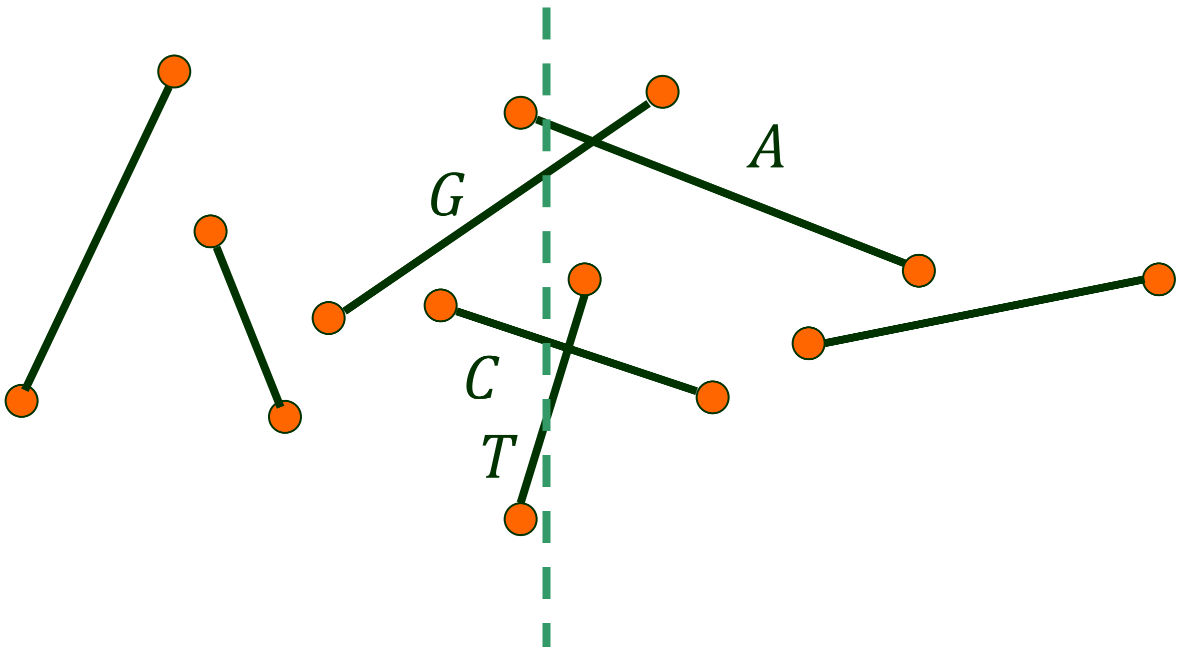


# Sweep Line Status

*The set of segments intersecting the sweep line.*

# Sweep Line Status

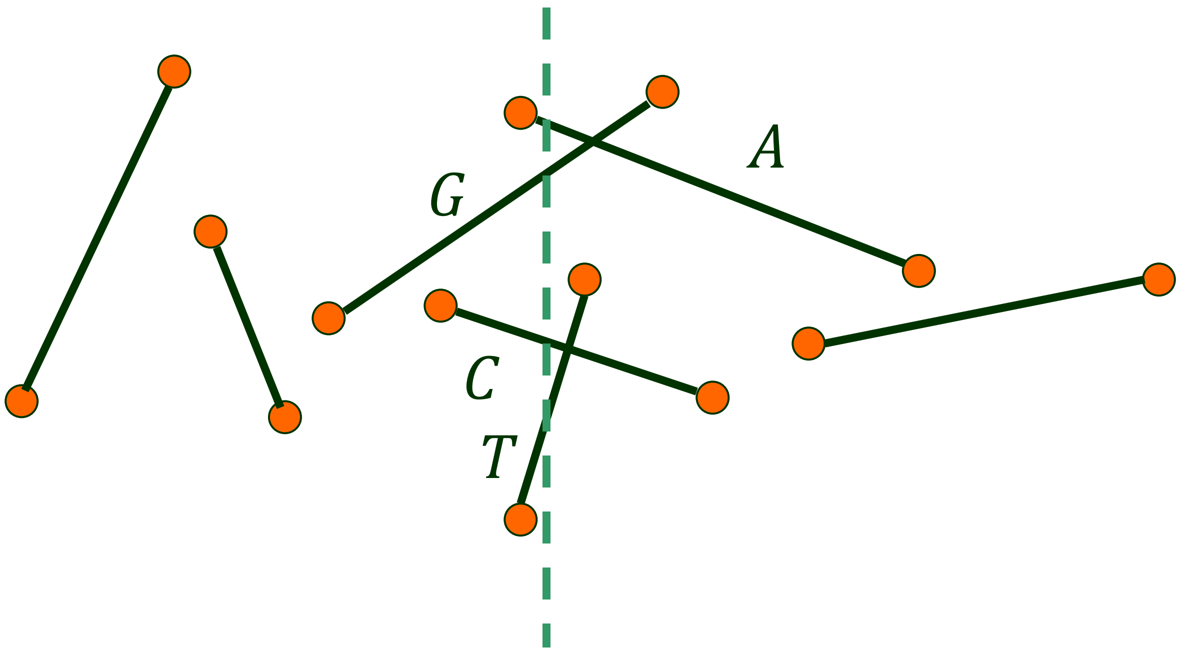
*The set of segments intersecting the sweep line.*



# Sweep Line Status

*The set of segments intersecting the sweep line.*

It changes as the sweep line moves, but *not continuously*.

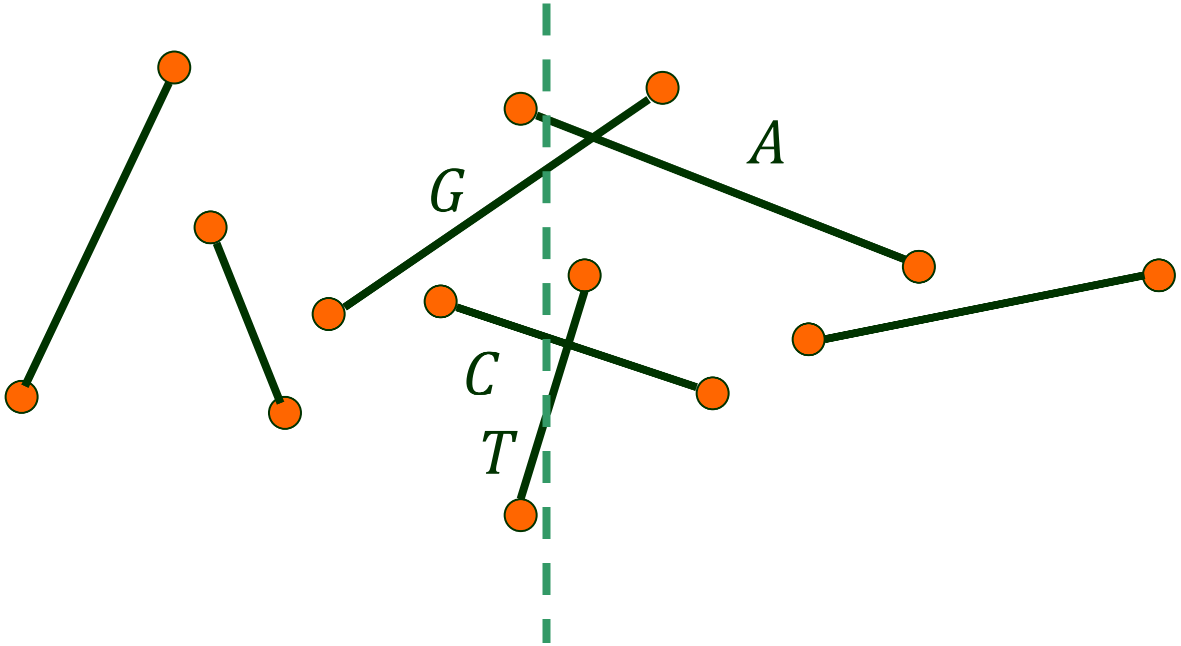


# Sweep Line Status

*The set of segments intersecting the sweep line.*

It changes as the sweep line moves, but *not continuously*.

Updates of status happen only at *event points*.



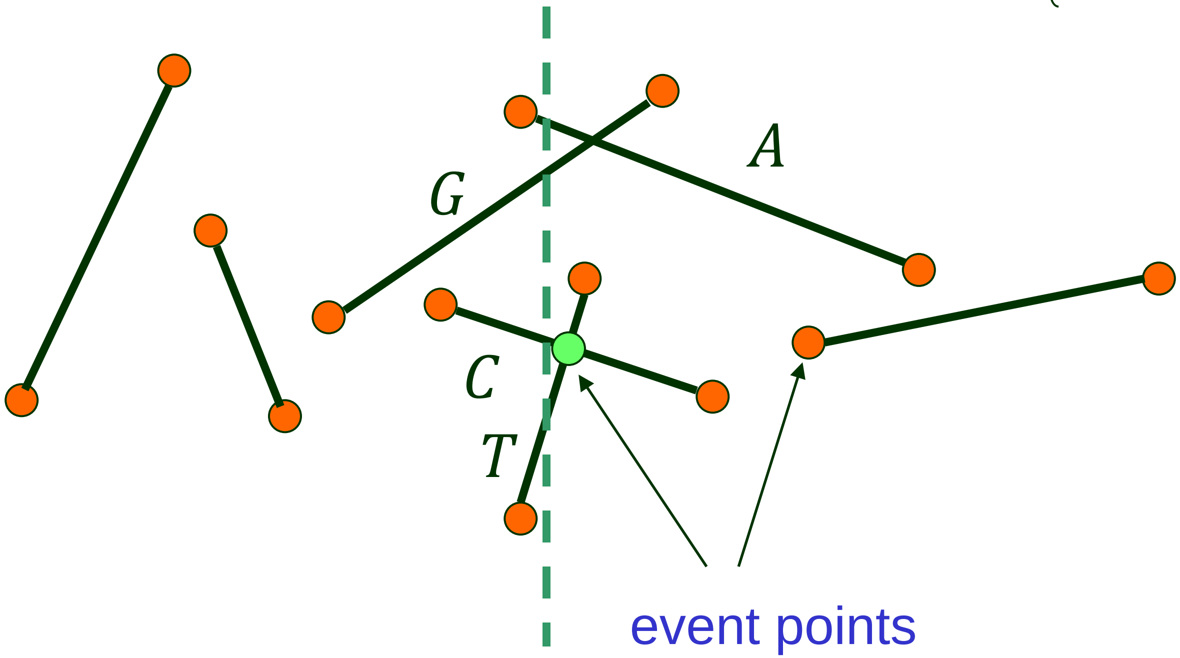
# Sweep Line Status

*The set of segments intersecting the sweep line.*

It changes as the sweep line moves, but *not continuously*.

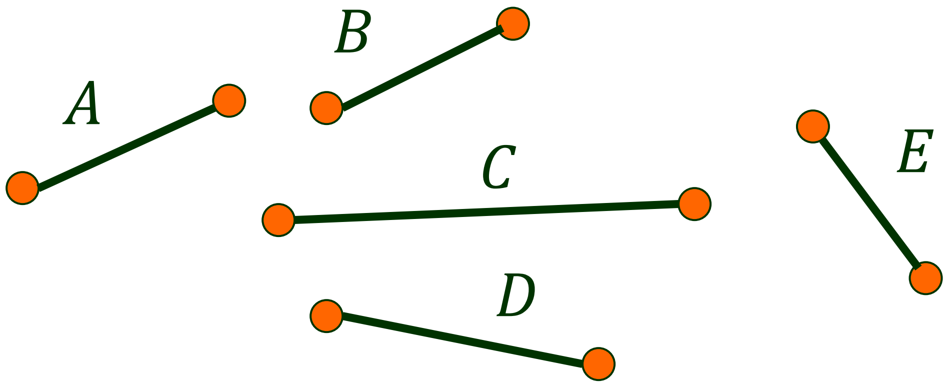
Updates of status happen only at *event points*.

- endpoints
- intersections



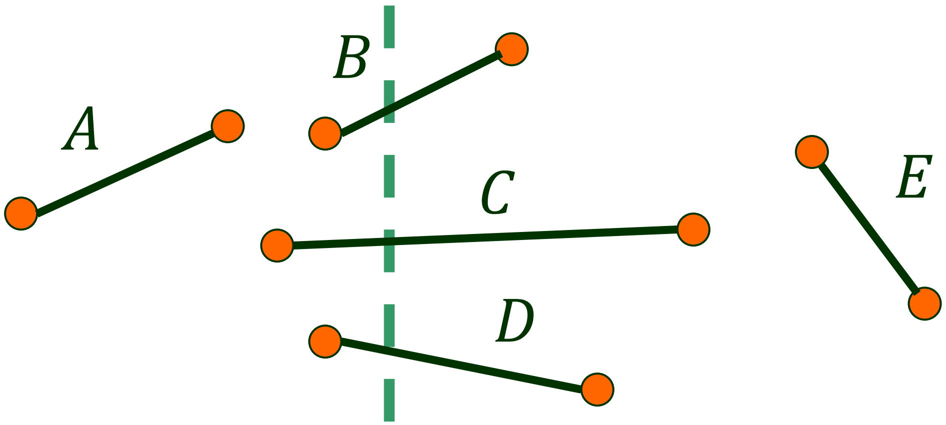
# Ordering Segments

A *total order* over the segments that intersect the current position of the sweep line:



# Ordering Segments

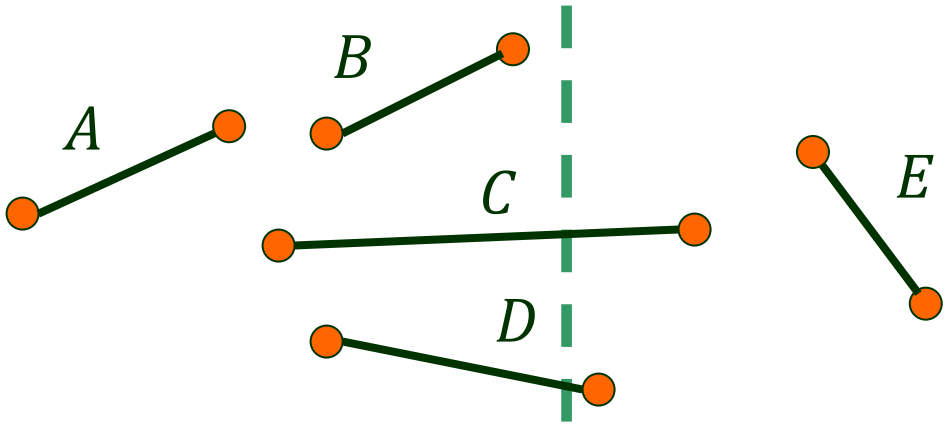
A *total order* over the segments that intersect the current position of the sweep line:




$D < C < B$
($A$ and $E$ not in the ordering)

# Ordering Segments

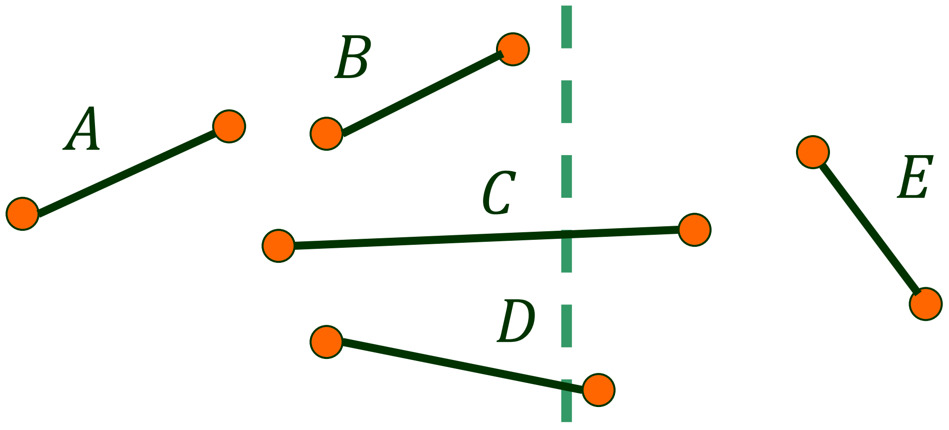
A *total order* over the segments that intersect the current position of the sweep line:




$D < C$
($B$ drops out of the ordering)

# Ordering Segments

A *total order* over the segments that intersect the current position of the sweep line:



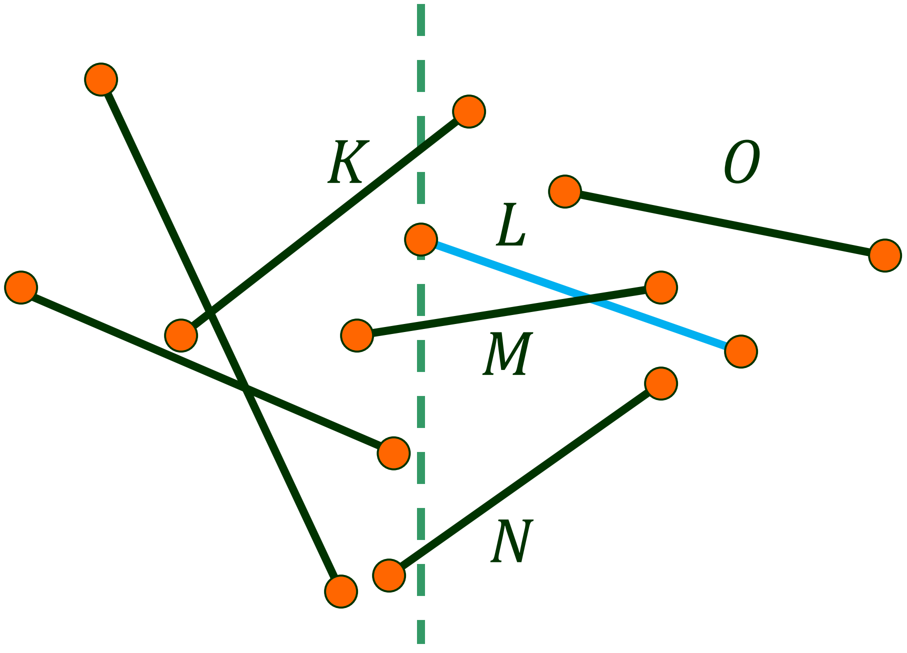

$D < C$
($B$ drops out of the ordering)

At an event point, the sequence of segments changes:

- Update the status.
- Detect the intersections.

# Status Update (1)

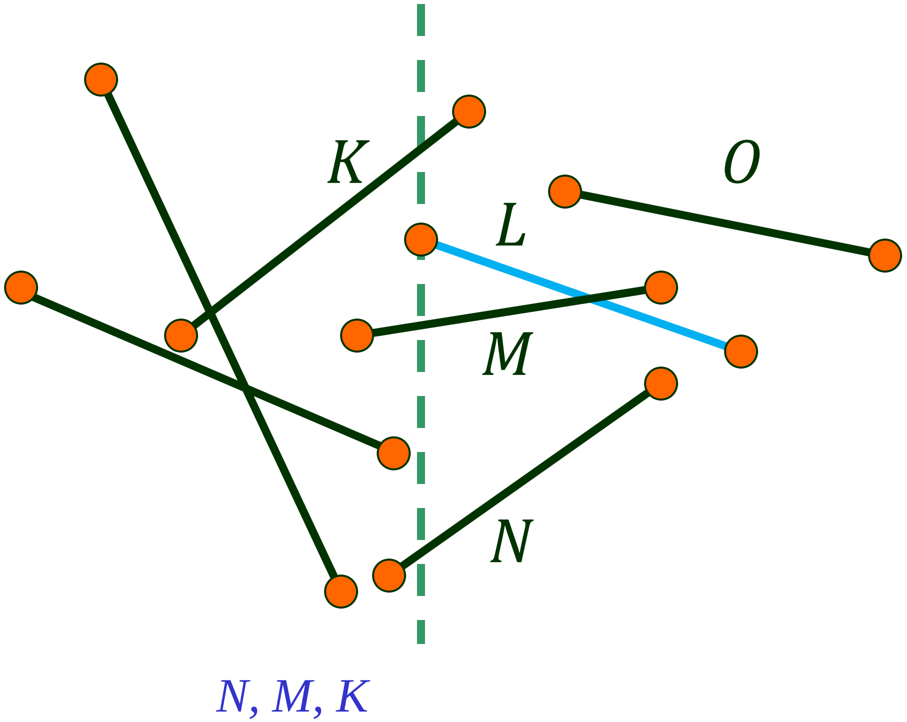
Event point is the left endpoint of a segment.




*N*, *M*, *K*

# Status Update (1)

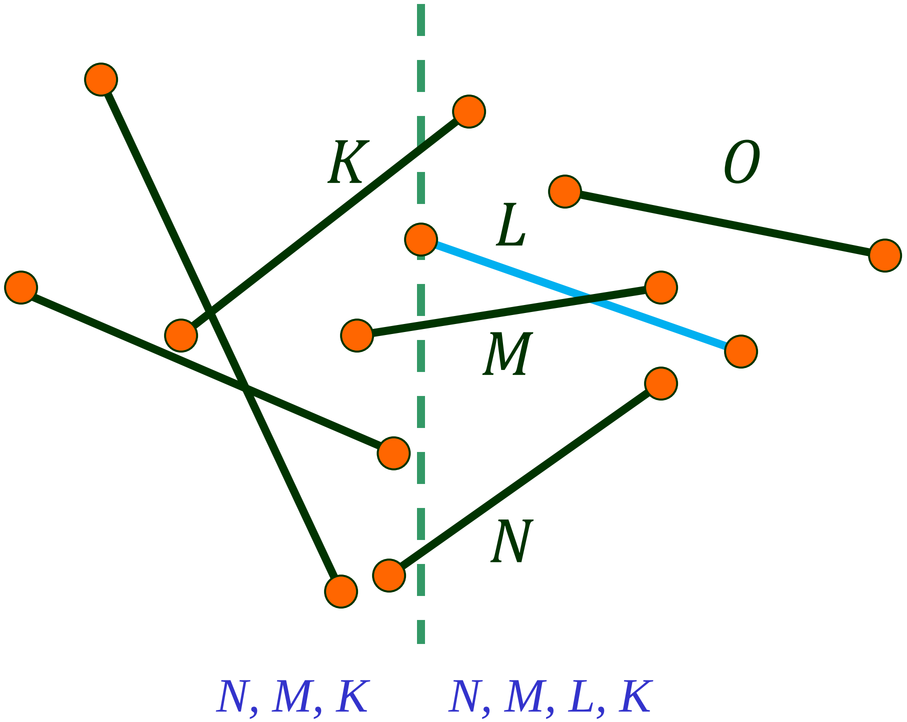
Event point is the left endpoint of a segment.




- A new segment *L* intersects the sweep line.

# Status Update (1)

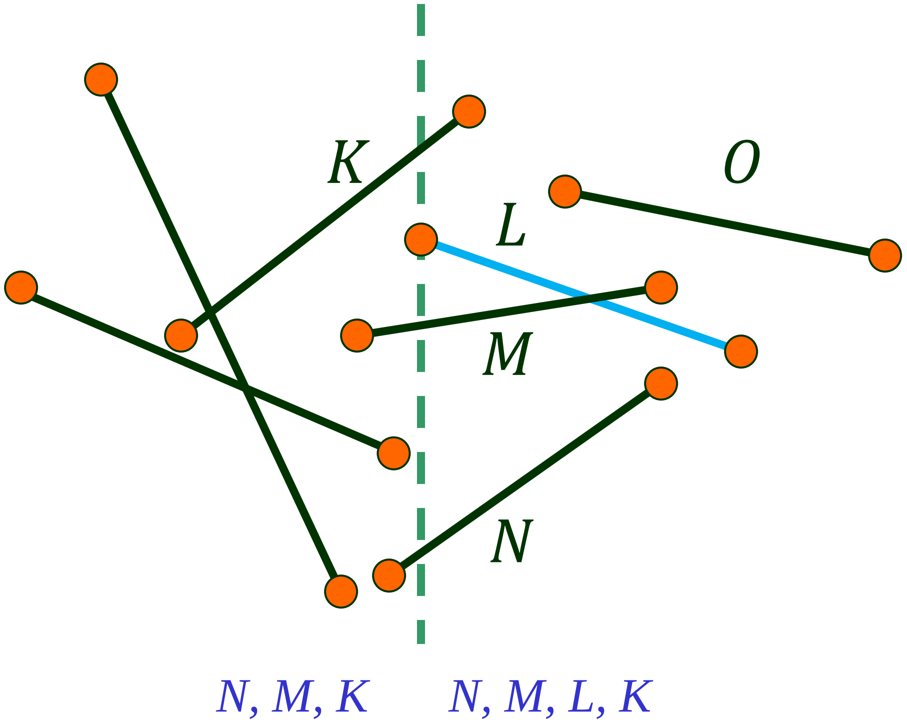
Event point is the left endpoint of a segment.




- A new segment $L$ intersects the sweep line.

# Status Update (1)

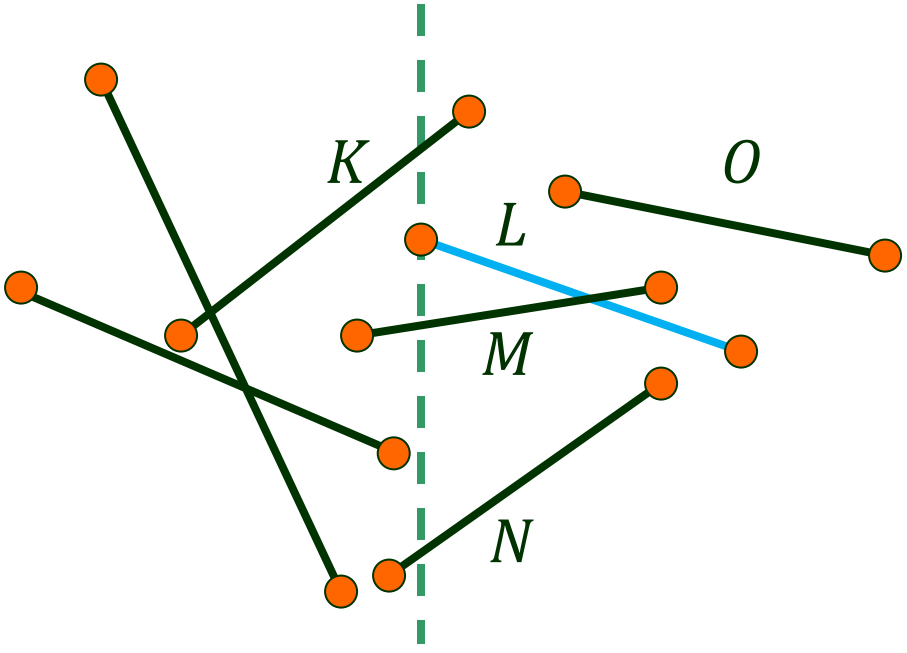
Event point is the left endpoint of a segment.




- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).

# Status Update (1)

Event point is the left endpoint of a segment.

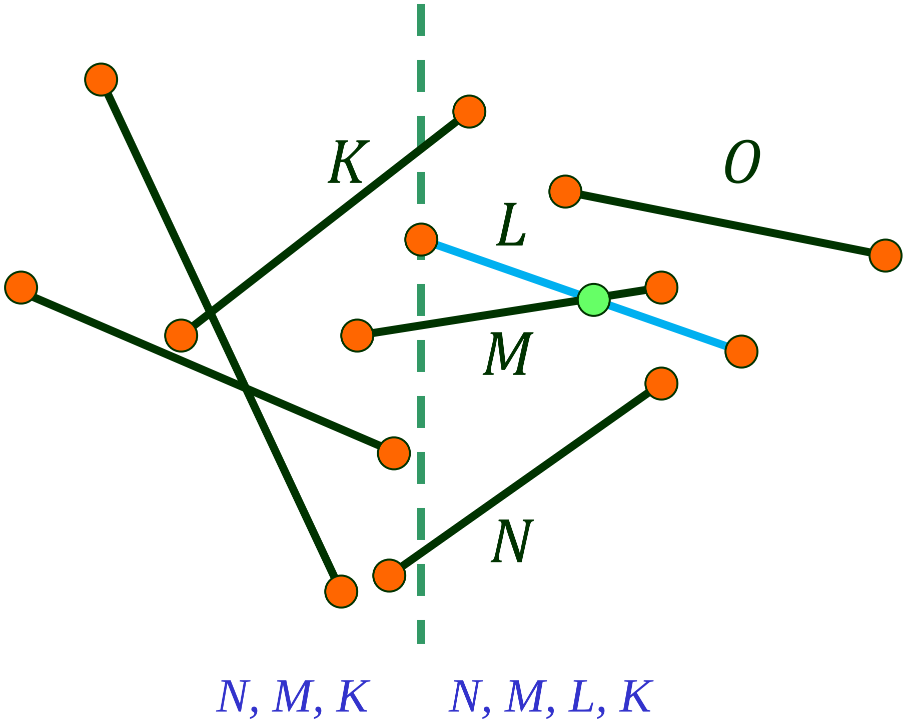



- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).

Two segments must have become neighbors before they can intersect as the sweep line moves.

# Status Update (1)

Event point is the left endpoint of a segment.

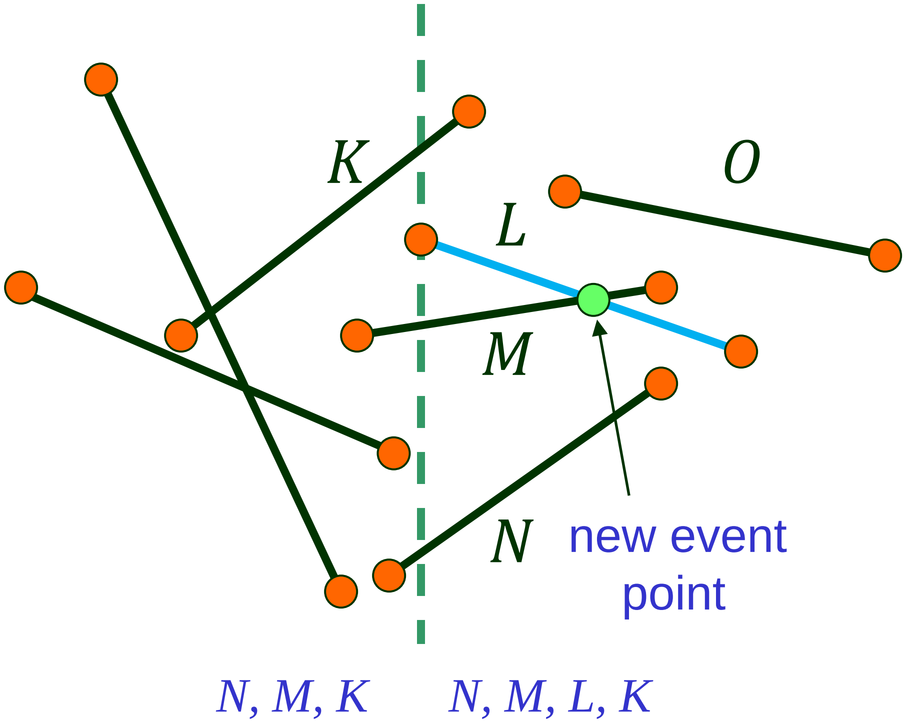



- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).

Two segments must have become neighbors before they can intersect as the sweep line moves.

# Status Update (1)

Event point is the left endpoint of a segment.

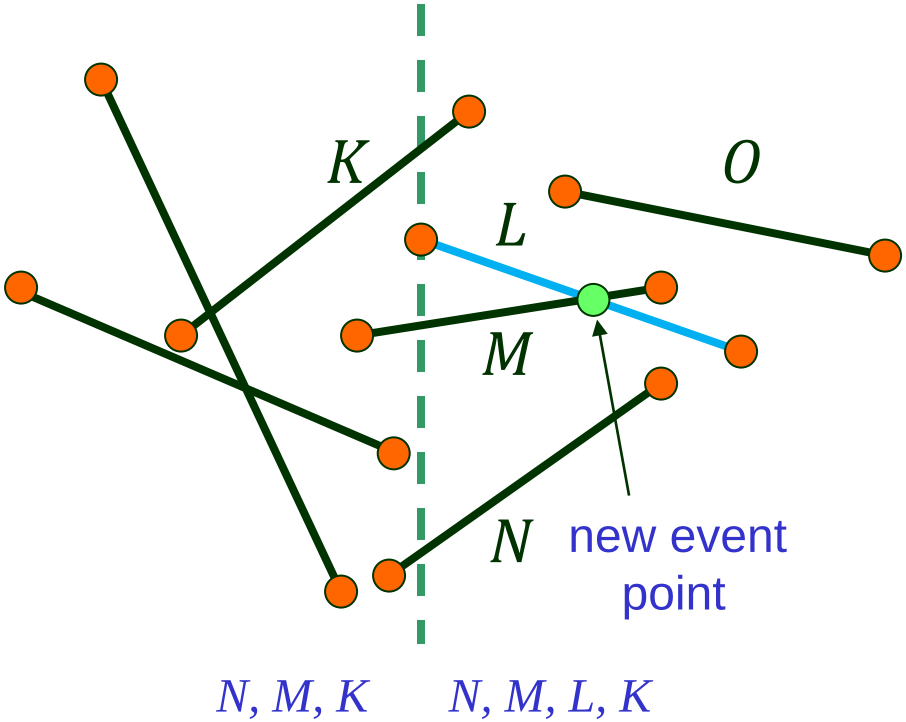



- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).

Two segments must have become neighbors before they can intersect as the sweep line moves.

# Status Update (1)

Event point is the left endpoint of a segment.



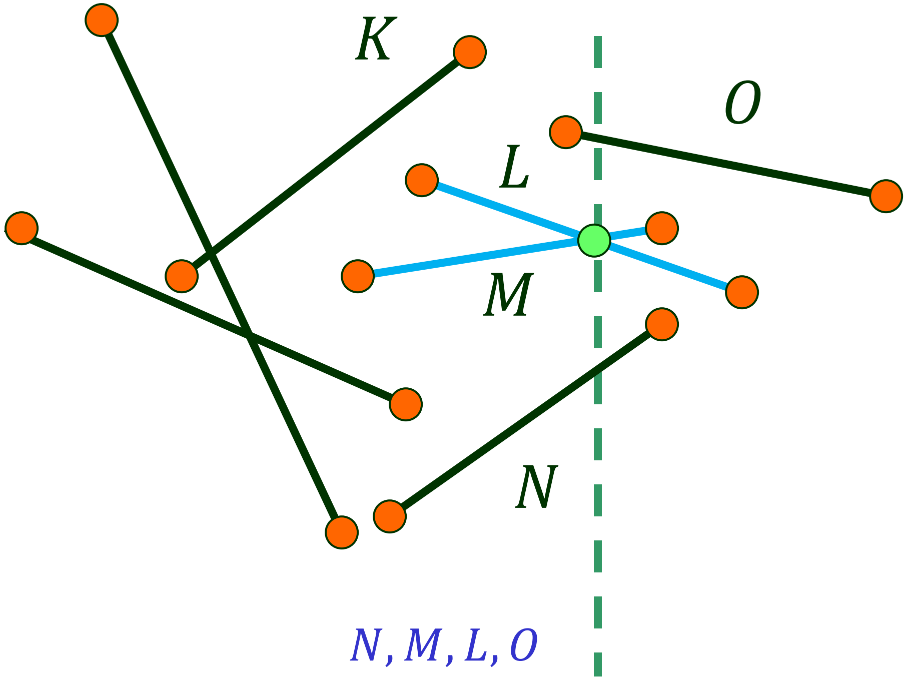

- A new segment $L$ intersects the sweep line.
- Check if $L$ intersects with the segment above ($K$) and the segment below ($M$).

Two segments must have become neighbors before they can intersect as the sweep line moves.

- Intersection(s) are new event points.

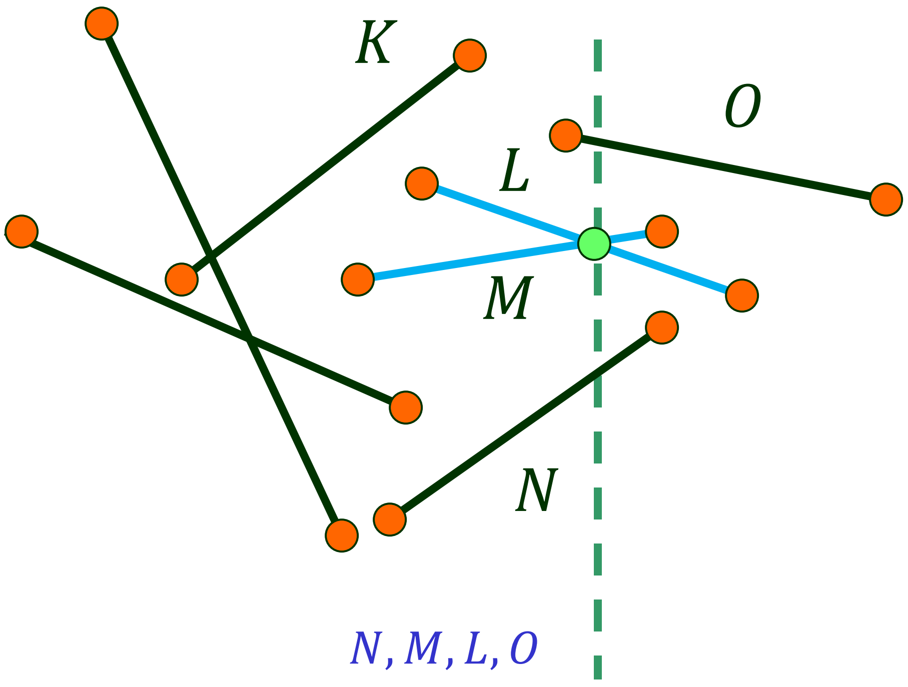
# Status Update (2)

Event point is an intersection.



# Status Update (2)

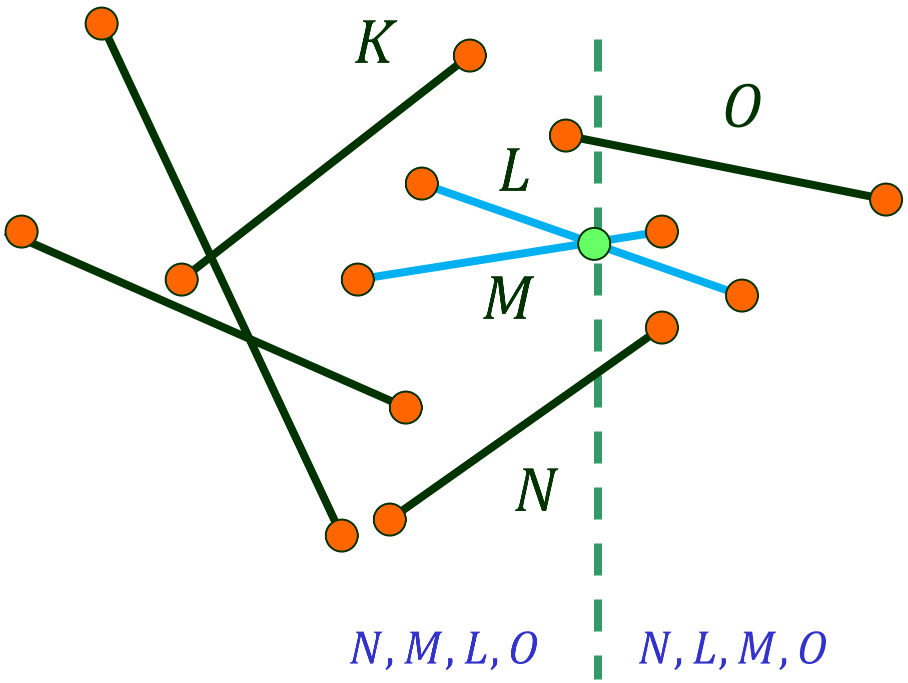
Event point is an intersection.




- The two intersecting segments ($L$ and $M$) change order.

# Status Update (2)

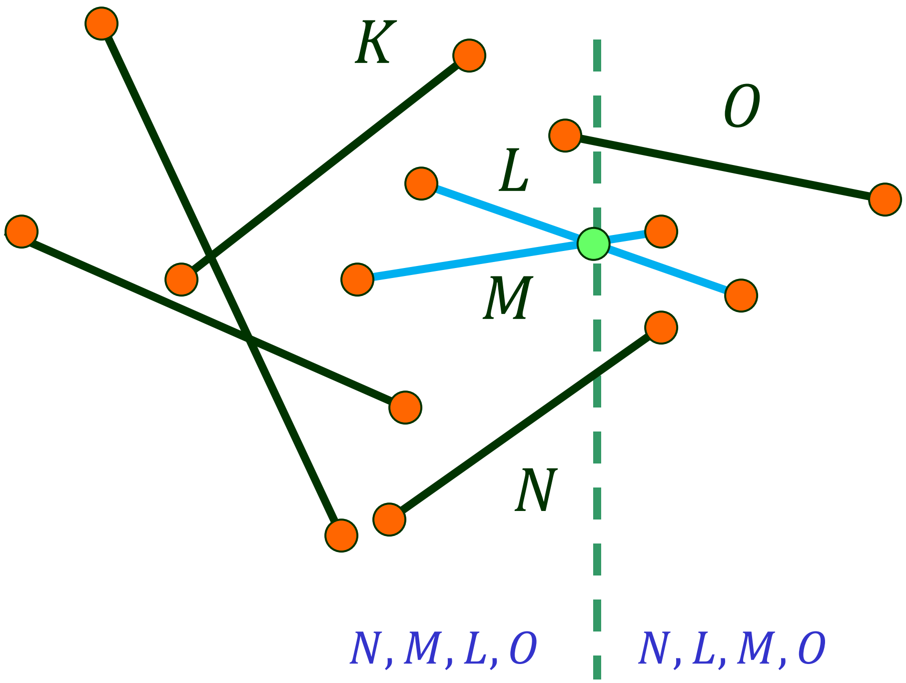
Event point is an intersection.




- The two intersecting segments ($L$ and $M$) change order.

# Status Update (2)

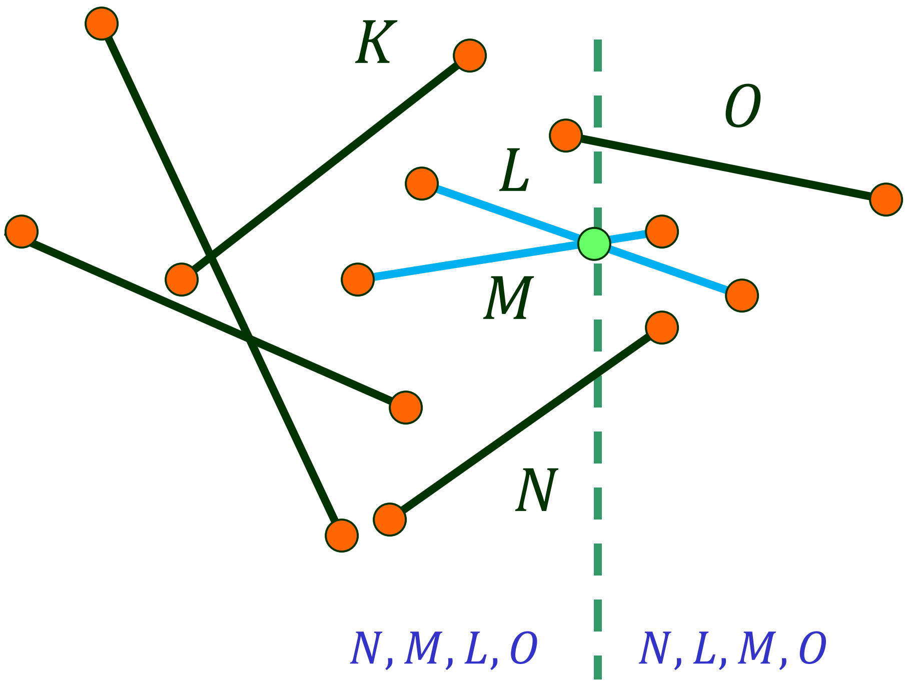
Event point is an intersection.




- The two intersecting segments ($L$ and $M$) change order.
- Check intersection with new neighbors ($M$ with $O$ and $L$ with $N$).

# Status Update (2)

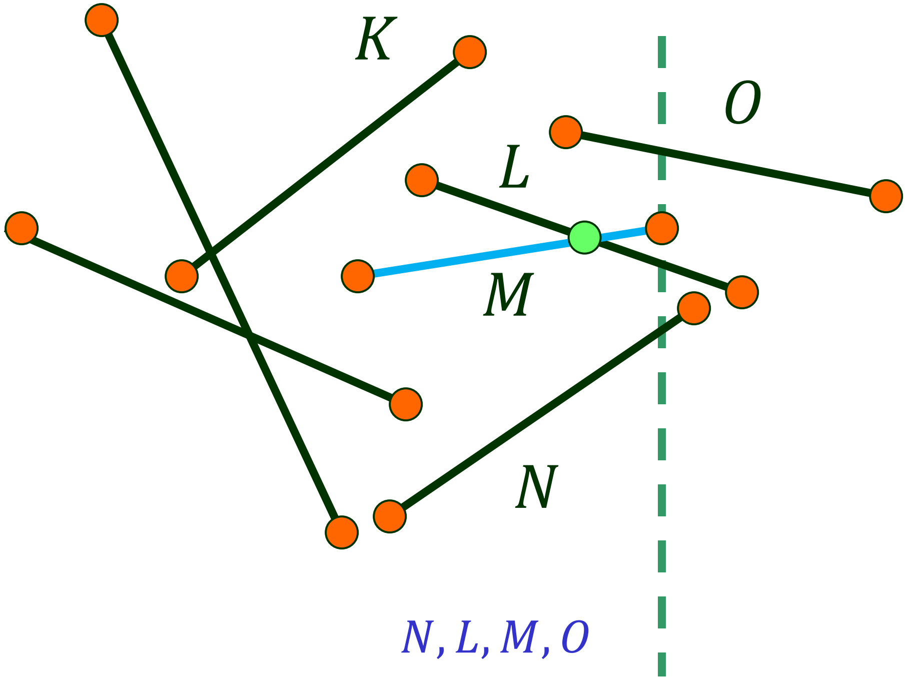
Event point is an intersection.




- The two intersecting segments ($L$ and $M$) change order.
- Check intersection with new neighbors ($M$ with $O$ and $L$ with $N$).
- Intersection(s) are new event points.

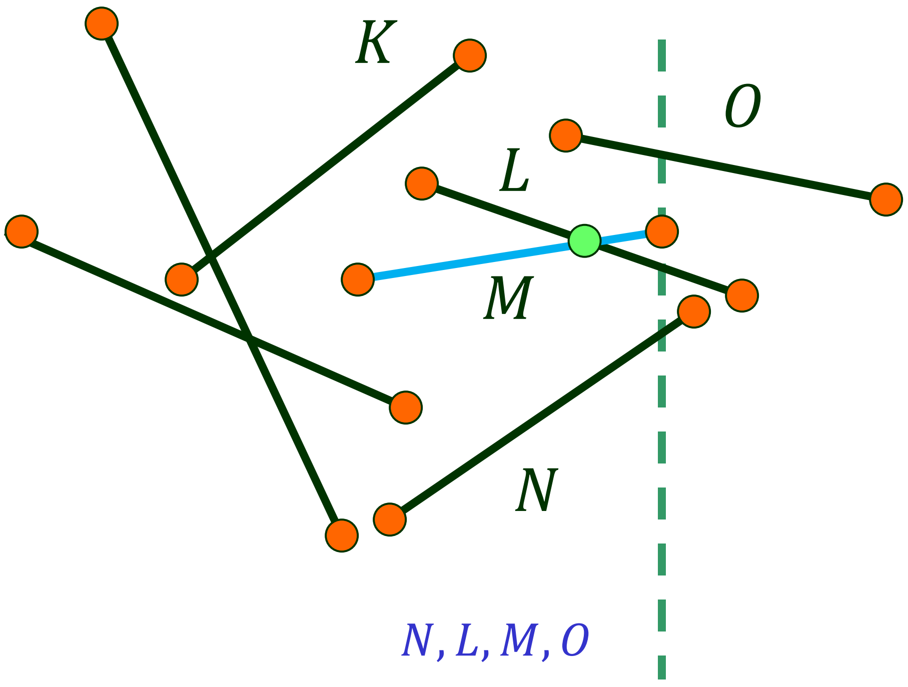
# Status Update (3)

Event point is a right endpoint of a segment.



# Status Update (3)

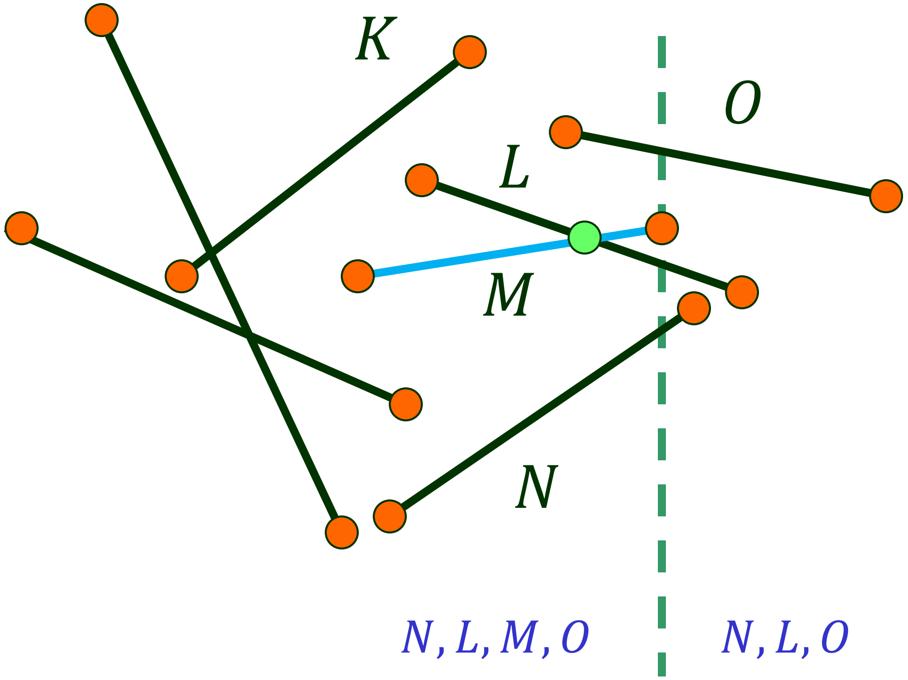
Event point is a right endpoint of a segment.




- The two neighbors ($O$ and $L$) become adjacent.

# Status Update (3)

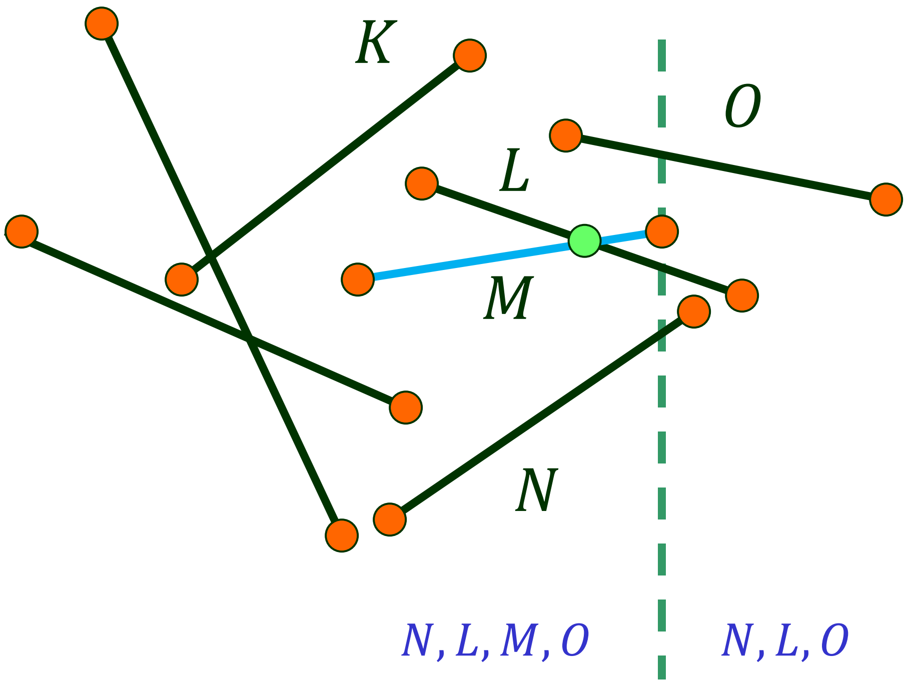
Event point is a right endpoint of a segment.




- The two neighbors ($O$ and $L$) become adjacent.

# Status Update (3)

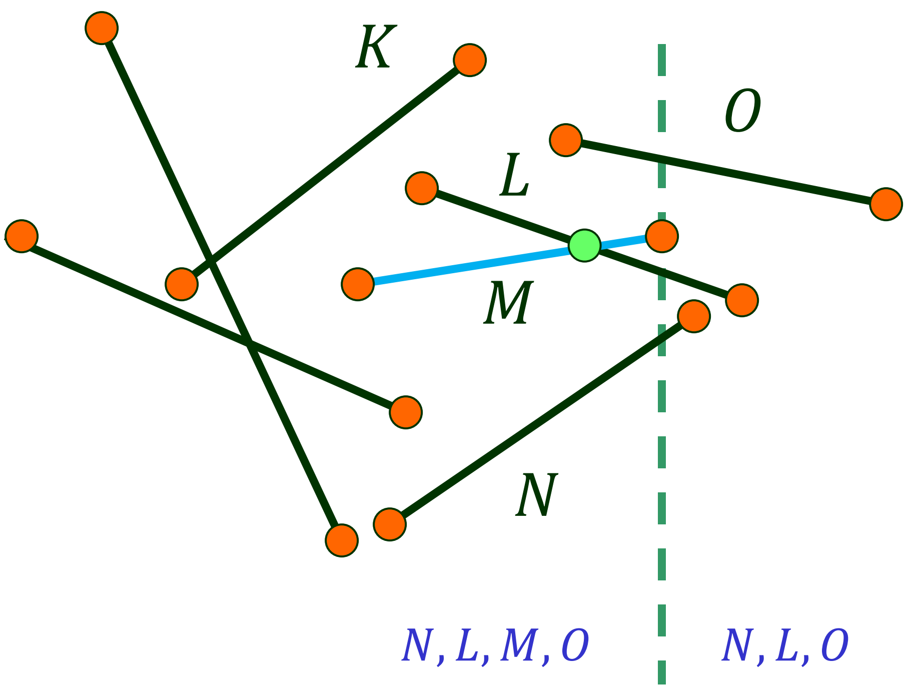
Event point is a right endpoint of a segment.




- The two neighbors ($O$ and $L$) become adjacent.
- Check if they ($O$ and $L$) intersect.

# Status Update (3)

Event point is a right endpoint of a segment.




- The two neighbors ($O$ and $L$) become adjacent.
- Check if they ($O$ and $L$) intersect.
- Intersection is new event point.

# Correctness

***Invariant*** at any time during the plane sweep:

All intersection points to the left of the sweep line have been computed correctly.

The correctness of the algorithm thus follows.

# III. Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

# III. Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment $s$.

# III. Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment *s*.

- fetching the next event
- inserting an event

# III. Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment *s*.

- fetching the next event
- inserting an event

Every event point $p$ is stored with all segments starting at $p$.

# III. Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment *s*.

- fetching the next event
- inserting an event

Every event point $p$ is stored with all segments starting at $p$.

Data structure: balanced binary search tree (e.g., red-black tree, AVL tree).

# III. Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment *s*.

- fetching the next event
- inserting an event

Every event point $p$ is stored with all segments starting at $p$.

Data structure: balanced binary search tree (e.g., red-black tree, AVL tree).

$m$ = #event points in the queue

# III. Data Structure for Event Queue

Ordering of event points:

- by $x$-coordinates
- by $y$-coordinates in case of a tie in $x$-coordinates.

Supports the following operations on a segment *s*.

- fetching the next event // $O(\log m)$
- inserting an event // $O(\log m)$

Every event point $p$ is stored with all segments starting at $p$.

Data structure: balanced binary search tree (e.g., red-black tree, AVL tree).

$m$ = #event points in the queue

# Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

# Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

- Use a balanced binary search tree $T$ to support the following operations on a segment $s$.

# Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

- Use a balanced binary search tree $T$ to support the following operations on a segment $s$.

  Insert($T$, $s$)
  Delete($T$, $s$)
  Above($T$, $s$)    // segment immediately above $s$
  Below($T$, $s$)    // segment immediately below $s$

# Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

- Use a balanced binary search tree $T$ to support the following operations on a segment $s$.

  Insert(*T*, *s*)
  Delete(*T*, *s*)
  Above(*T*, *s*)    // segment immediately above *s*
  Below(*T*, *s*)    // segment immediately below *s*

- *e.g*, Red-black trees, AVL trees (key comparisons replaced by cross-product comparisons).

# Data Structure for Sweep-line Status

- Describes the relationships among the segments intersected by the sweep line.

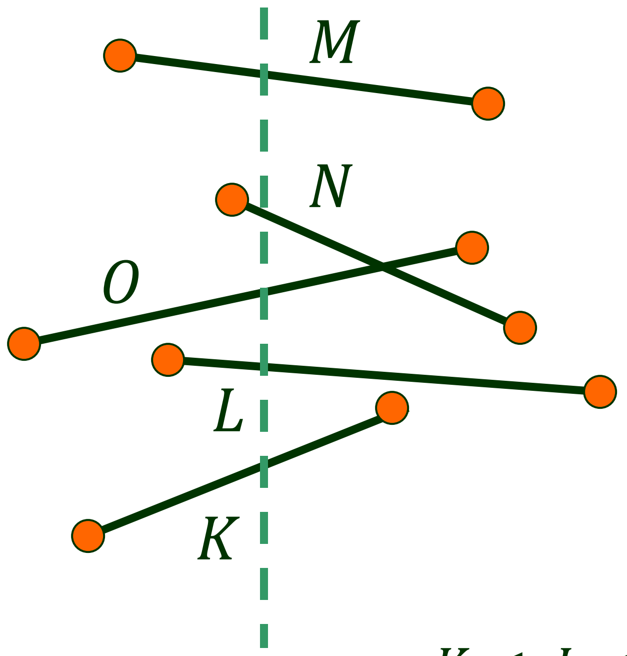
- Use a balanced binary search tree $T$ to support the following operations on a segment $s$.

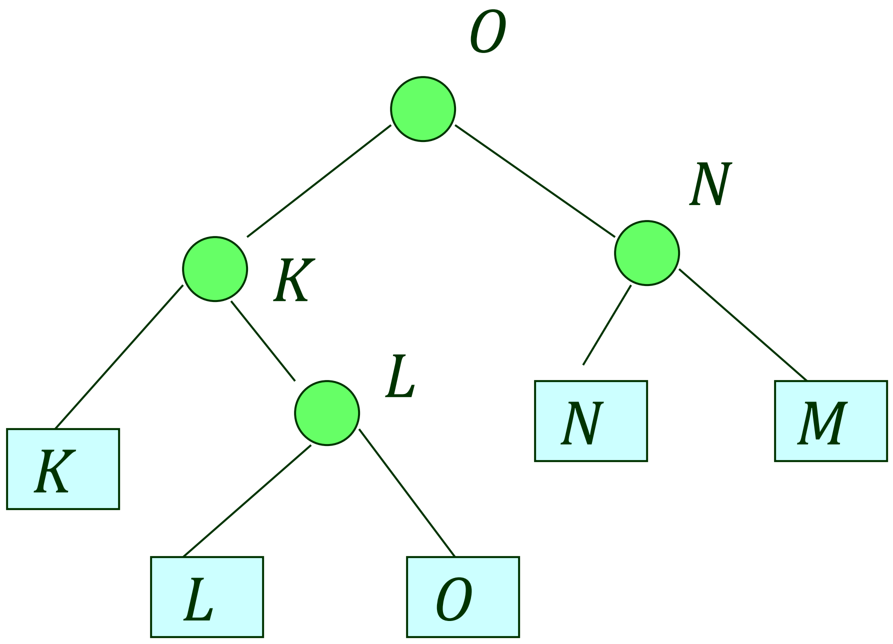
  Insert(*T*, *s*)
  Delete(*T*, *s*)
  Above(*T*, *s*)  // segment immediately above *s*
  Below(*T*, *s*)  // segment immediately below *s*

- *e.g*, Red-black trees, AVL trees (key comparisons replaced by cross-product comparisons).

- $O(\log n)$ for each operation.

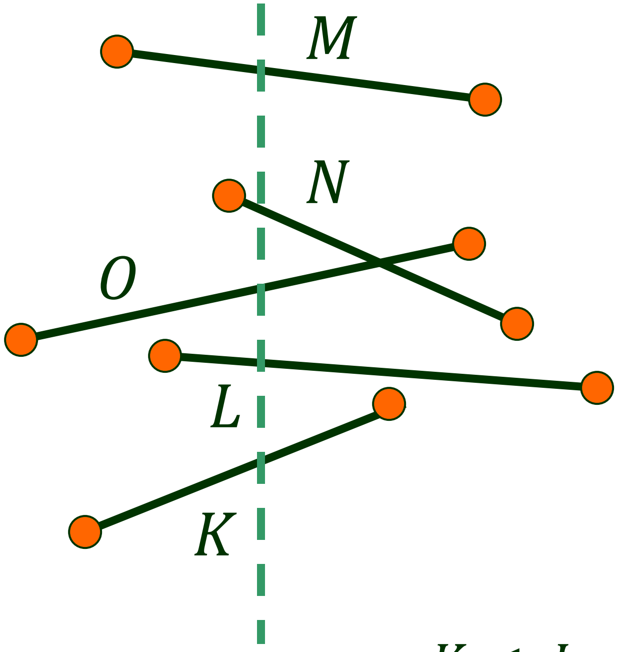
# Example

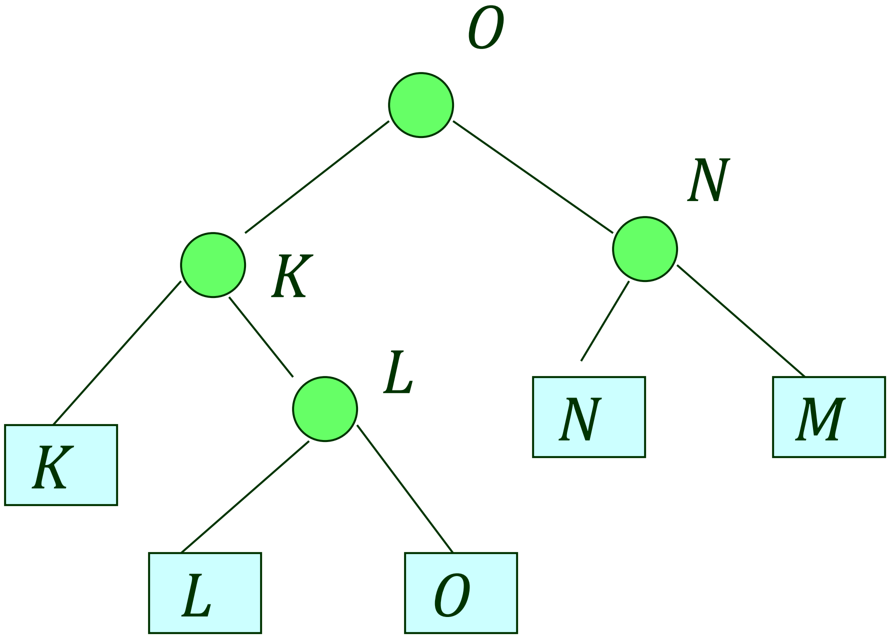



$K < L < O < N < M$

# Example

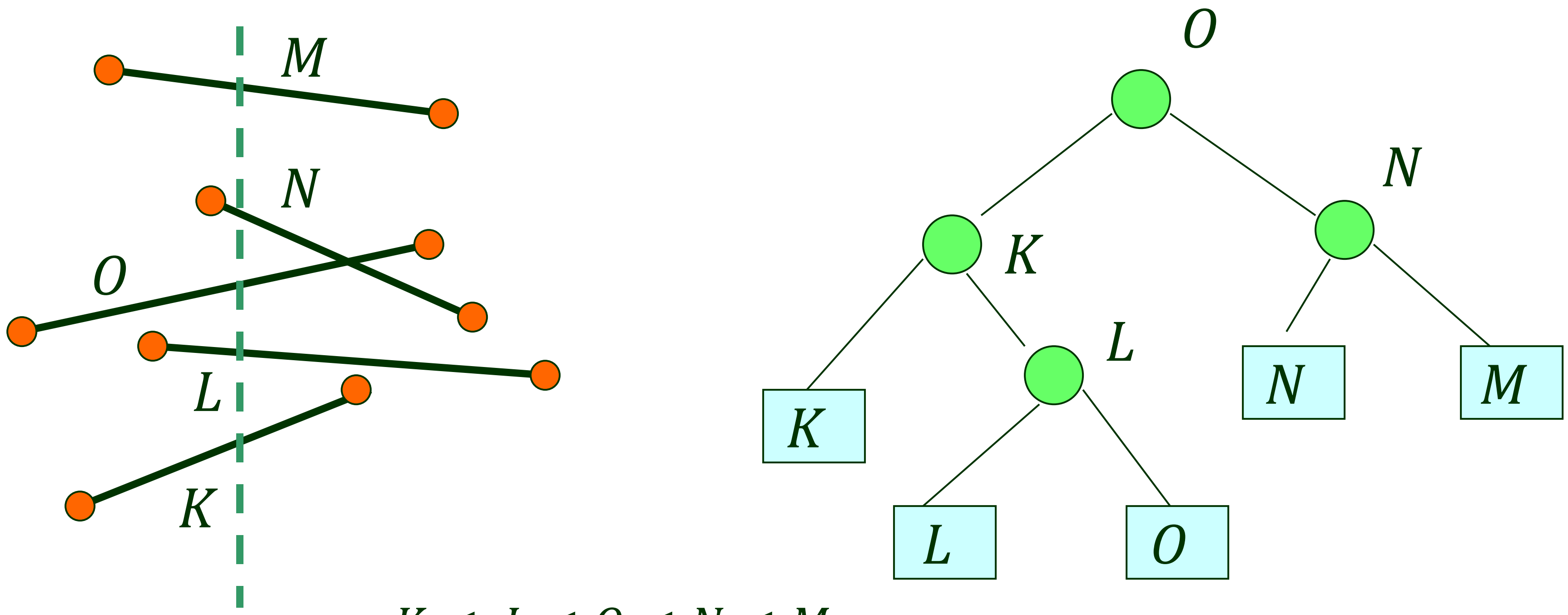



$K < L < O < N < M$

- The bottom-up order of the segments corresponds to the *left-to-right* order of the leaves in the tree $T$.

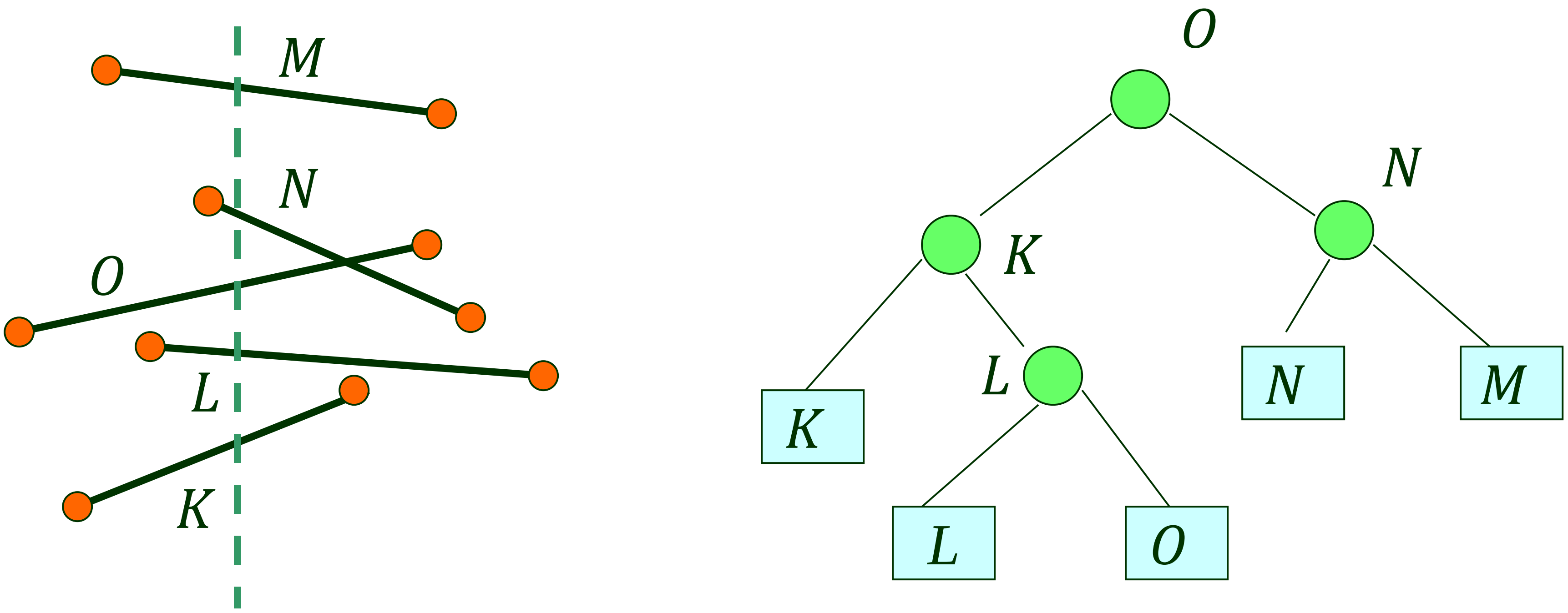
# Example




$K < L < O < N < M$

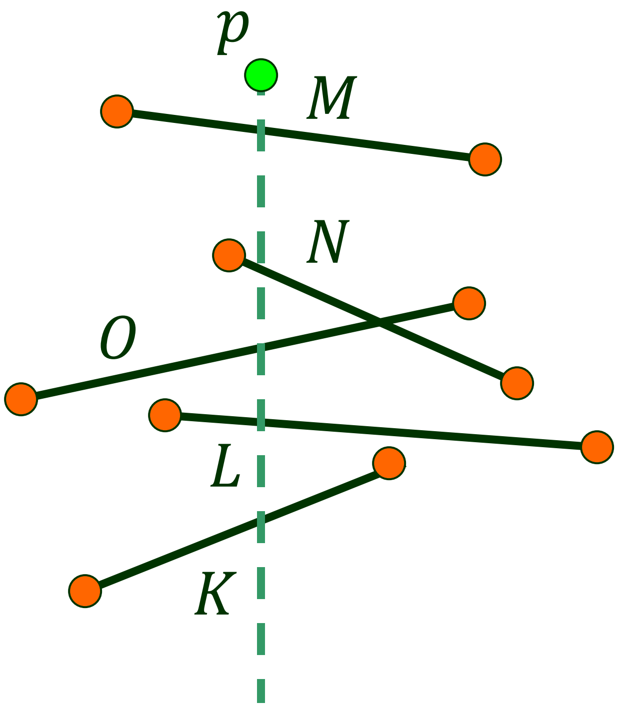
- The bottom-up order of the segments corresponds to the *left-to-right* order of the leaves in the tree $T$.
- Each internal node stores the segment from the *rightmost leaf* in its left subtree.

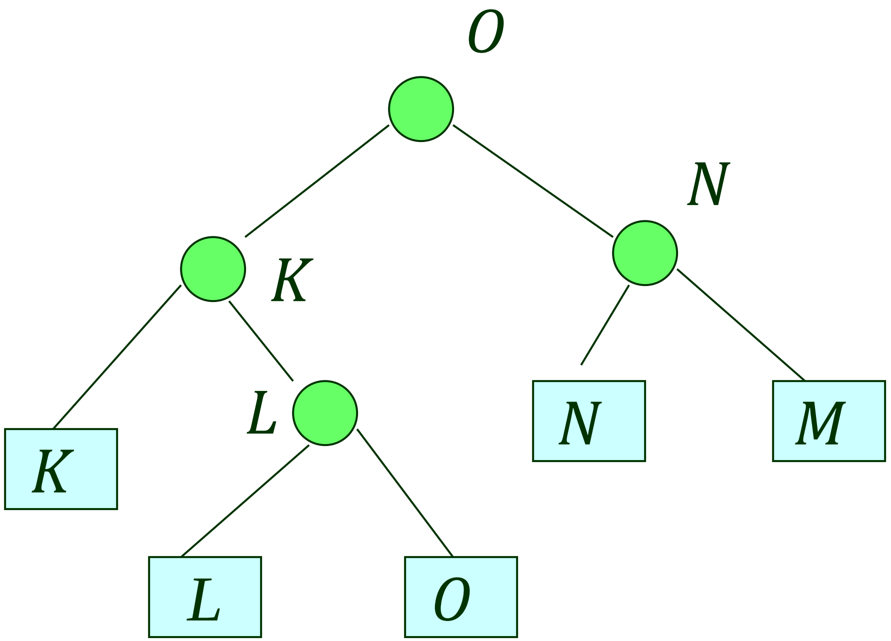
# Additional Operation




Searching for the segment immediately below some point $p$ on the sweep line.

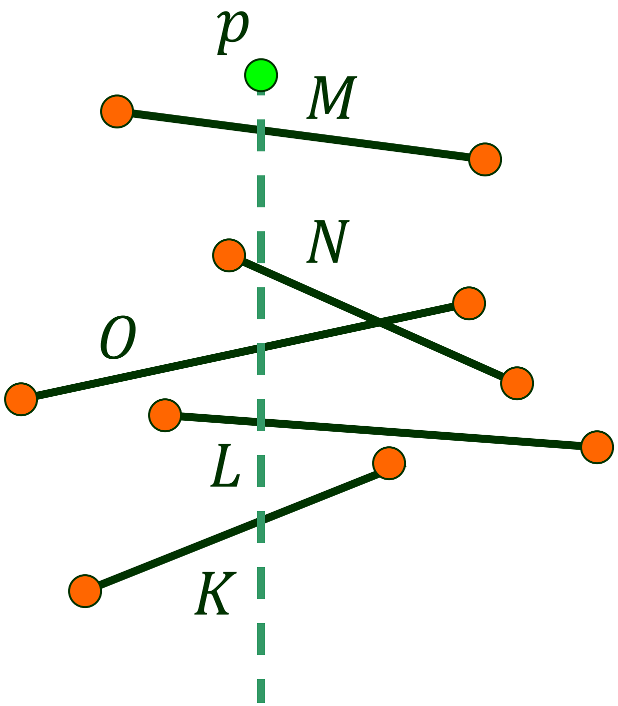
# Additional Operation

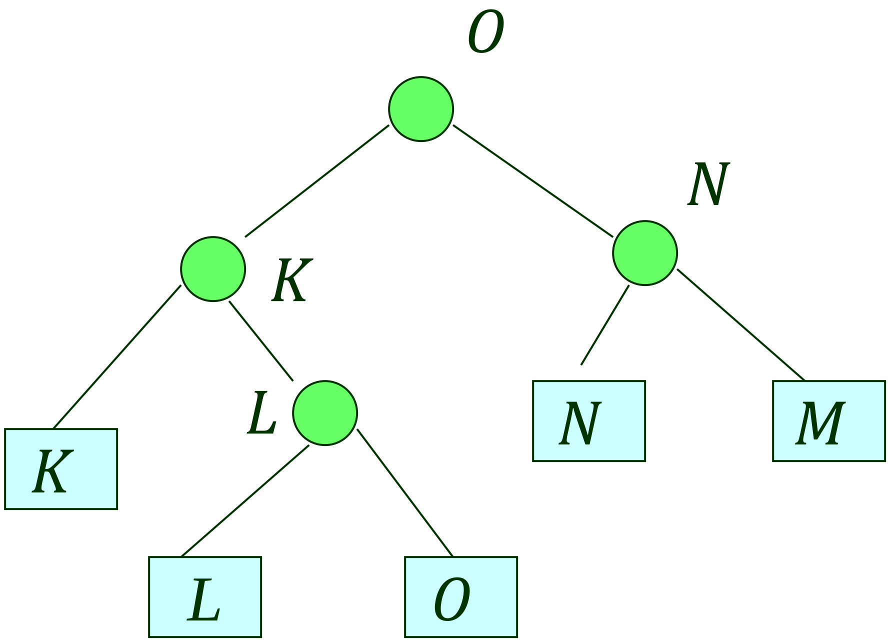



Searching for the segment immediately below some point $p$ on the sweep line.

# Additional Operation

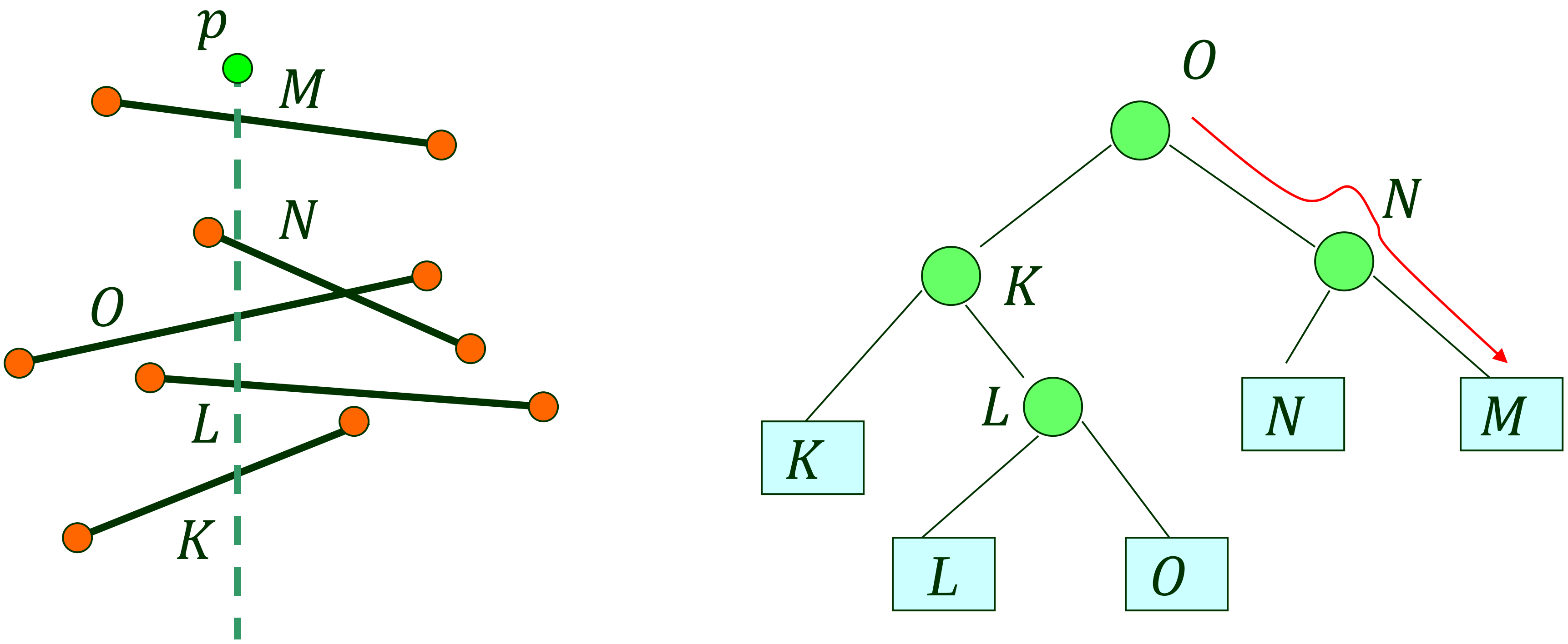



Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.

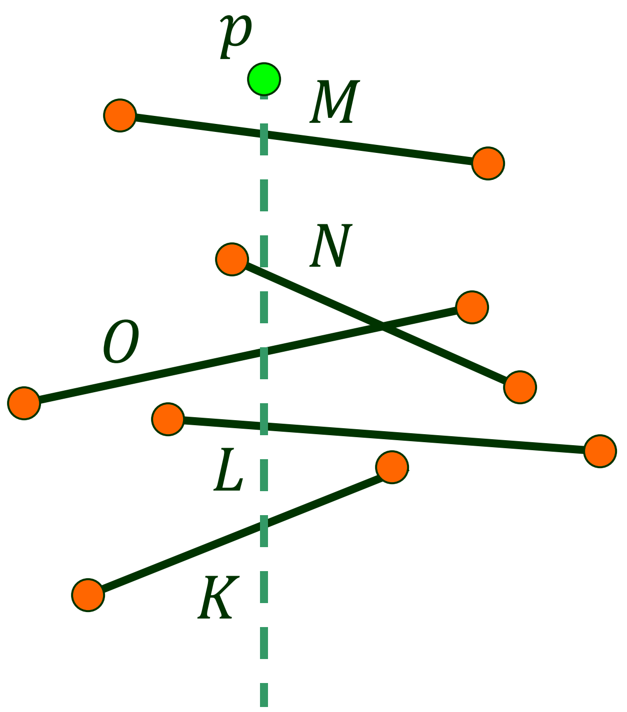
# Additional Operation

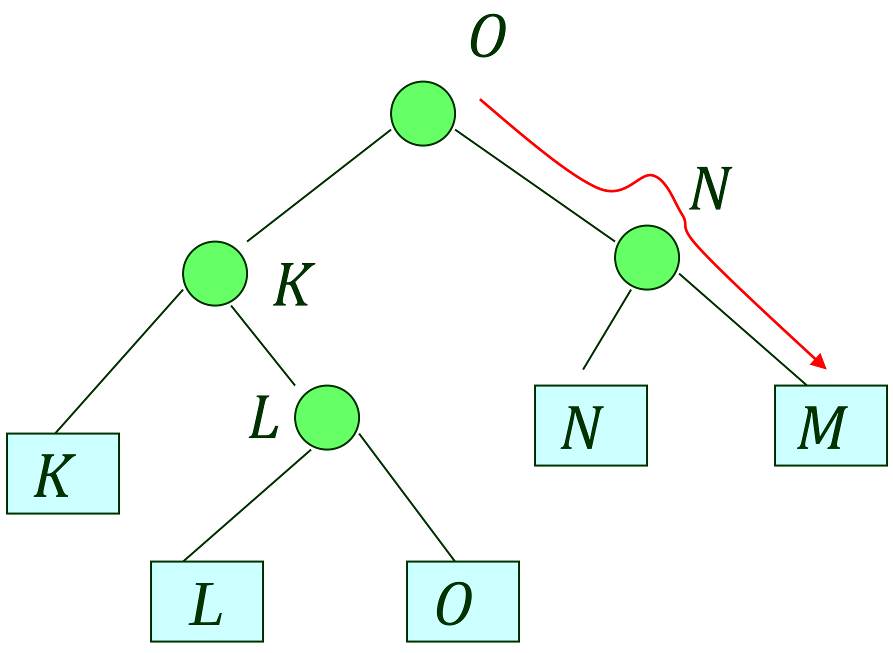



Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.

# Additional Operation

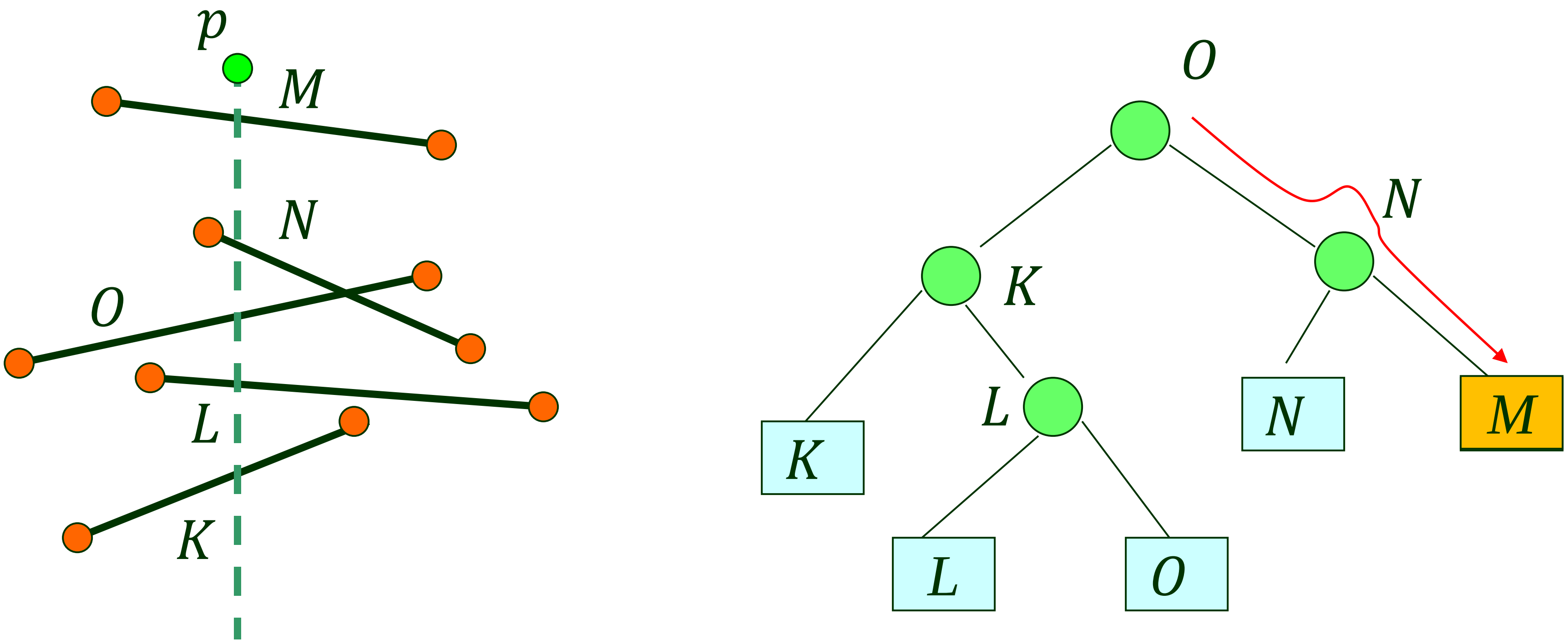



Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($p$) or the leaf immediately to its left ($q$).

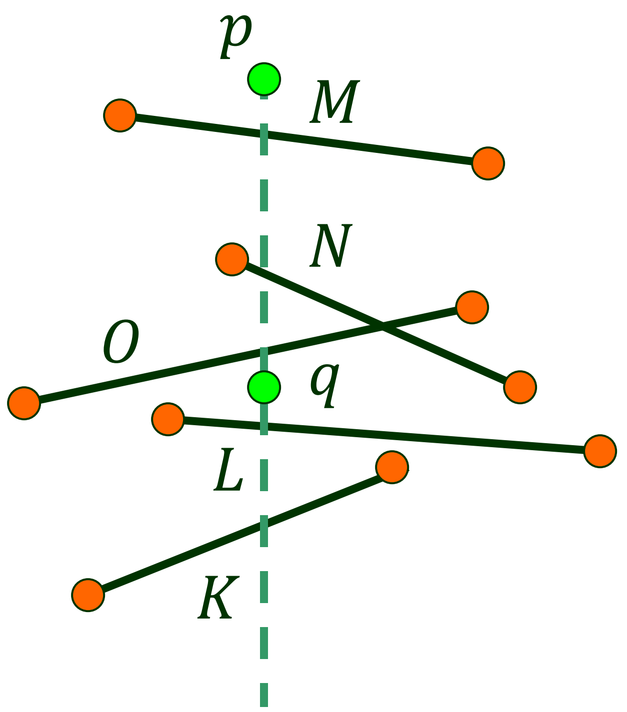
# Additional Operation

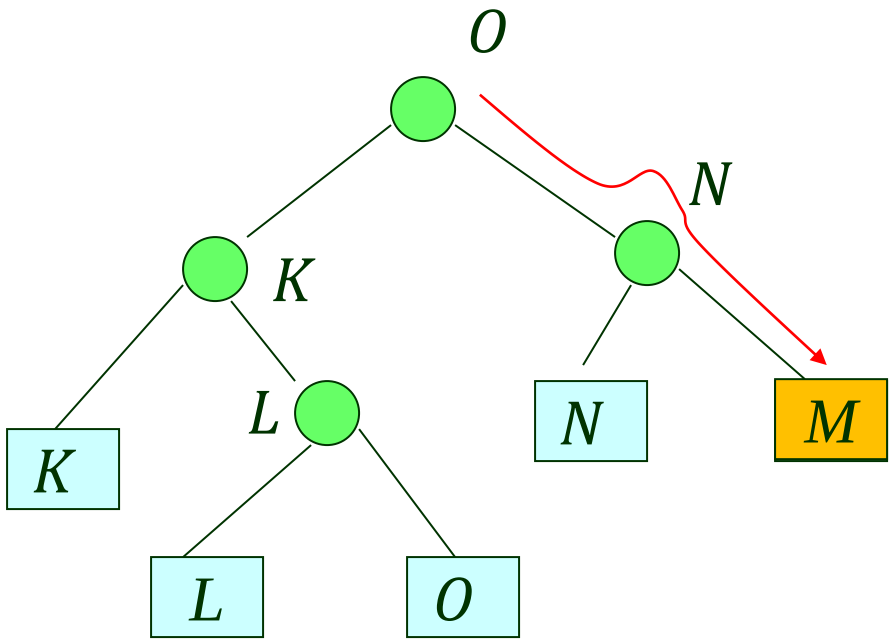



Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($p$) or the leaf immediately to its left ($q$).

# Additional Operation

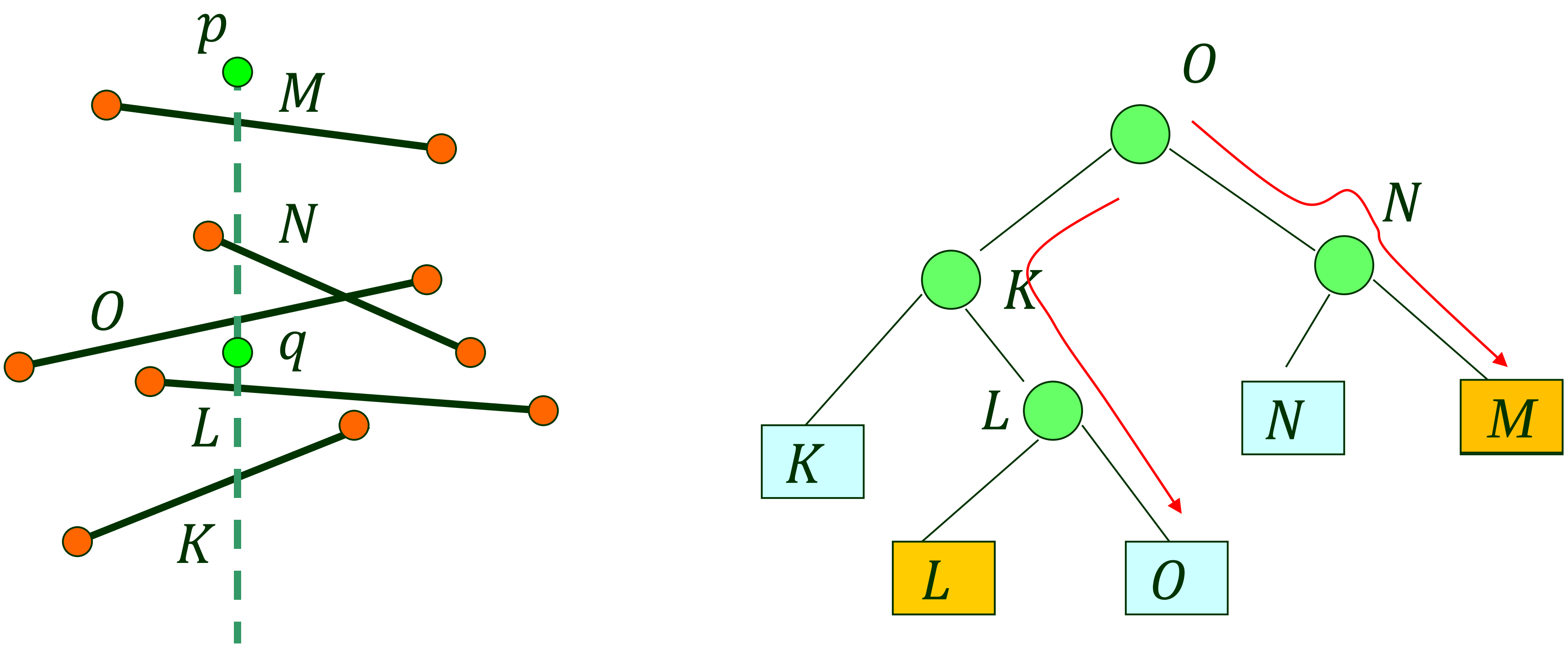



Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($p$) or the leaf immediately to its left ($q$).

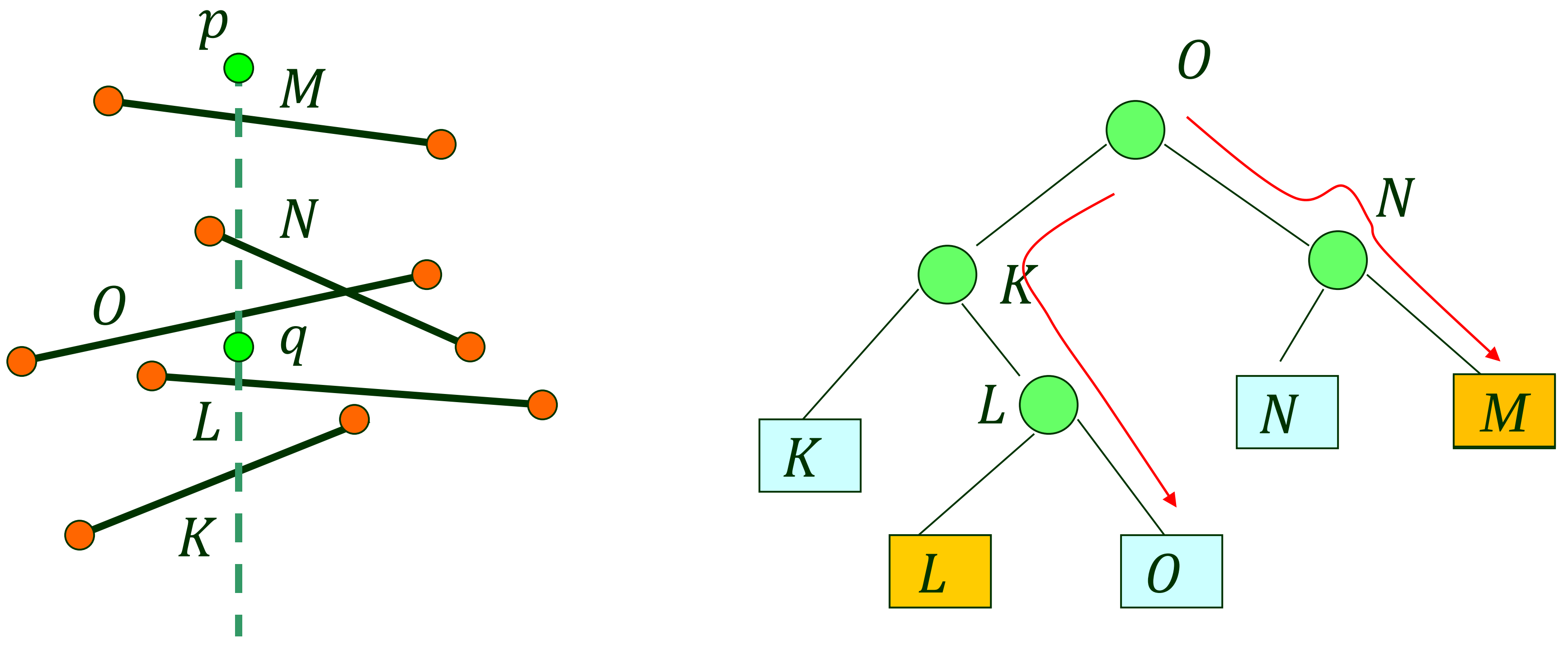
# Additional Operation




Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($p$) or the leaf immediately to its left ($q$).

# Additional Operation




Searching for the segment immediately below some point $p$ on the sweep line.

- Descend binary search tree all the way down to a leaf.
- Outputs either this leaf ($p$) or the leaf immediately to its left ($q$).

$O(\log n)$ time

# The Algorithm

FindIntersections($S$)

**Input**: a set $S$ of line segments

**Ouput**: all intersection points and for each intersection the segment containing it.

1. $Q \leftarrow \emptyset$ // initialize an empty event queue
2. Insert the segment endpoints into $Q$ // store with every left // endpoint the **corresponding segments**
3. $T \leftarrow \emptyset$ // initialize an empty status structure
4. while $Q \neq \emptyset$
5. do extract the next event point $p$
6. $Q \leftarrow Q - \{p\}$
7. HandleEventPoint($p$)

# Handling Event Points

Status updates (1) – (3) presented earlier.

# Handling Event Points

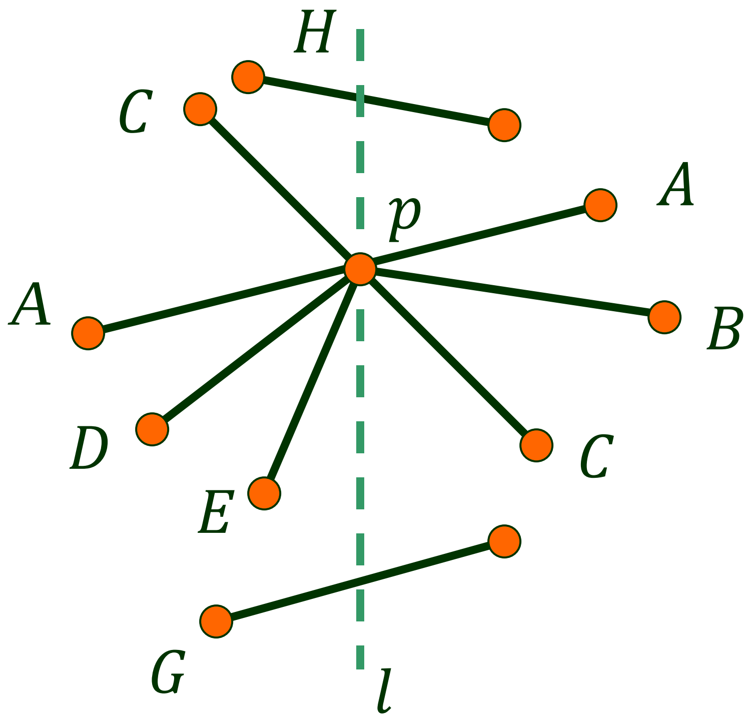
Status updates (1) – (3) presented earlier.

Degeneracy: Several segments are involved in one event point (tricky).

# Handling Event Points

Status updates (1) – (3) presented earlier.

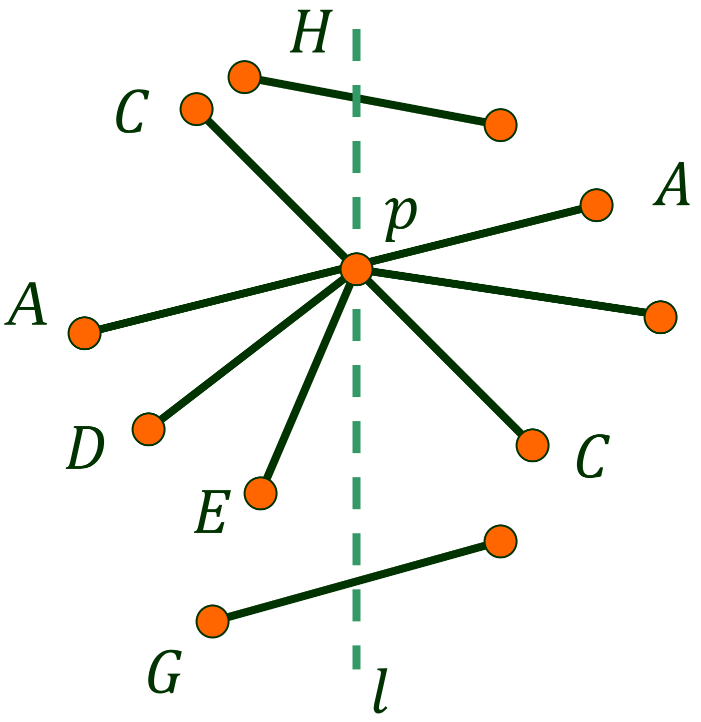
Degeneracy: Several segments are involved in one event point (tricky).

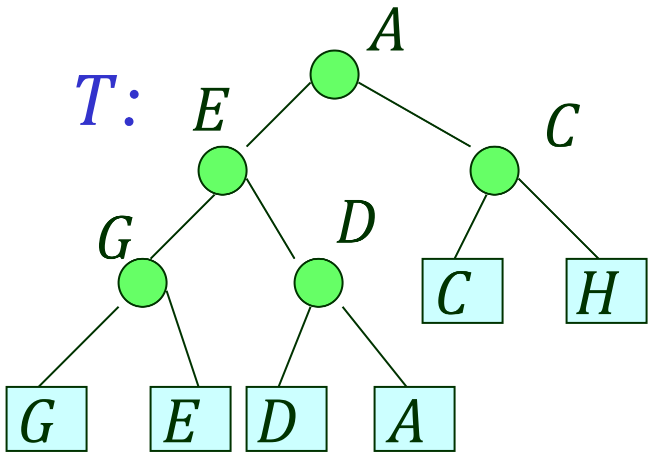


# Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: Several segments are involved in one event point (tricky).

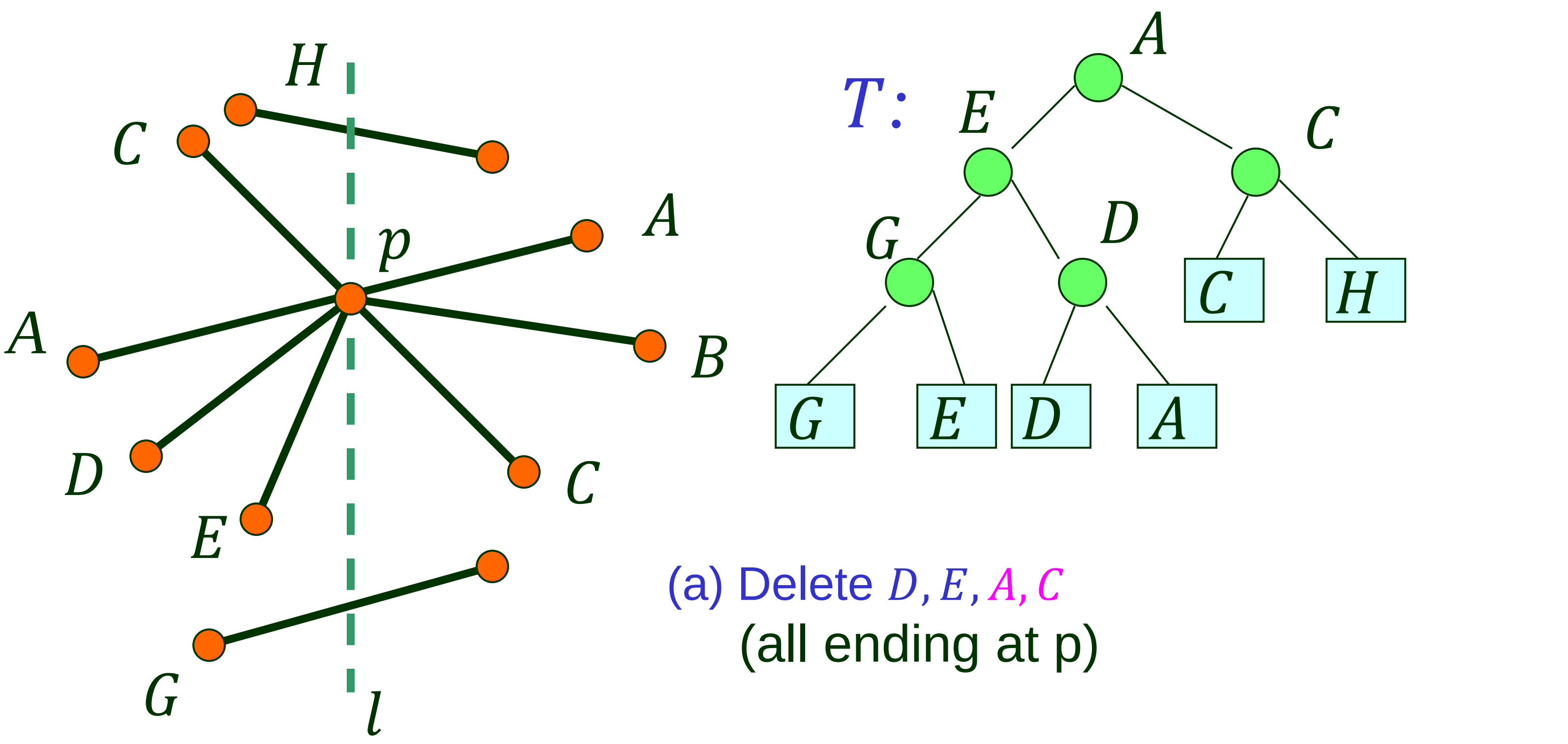


# Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: Several segments are involved in one event point (tricky).

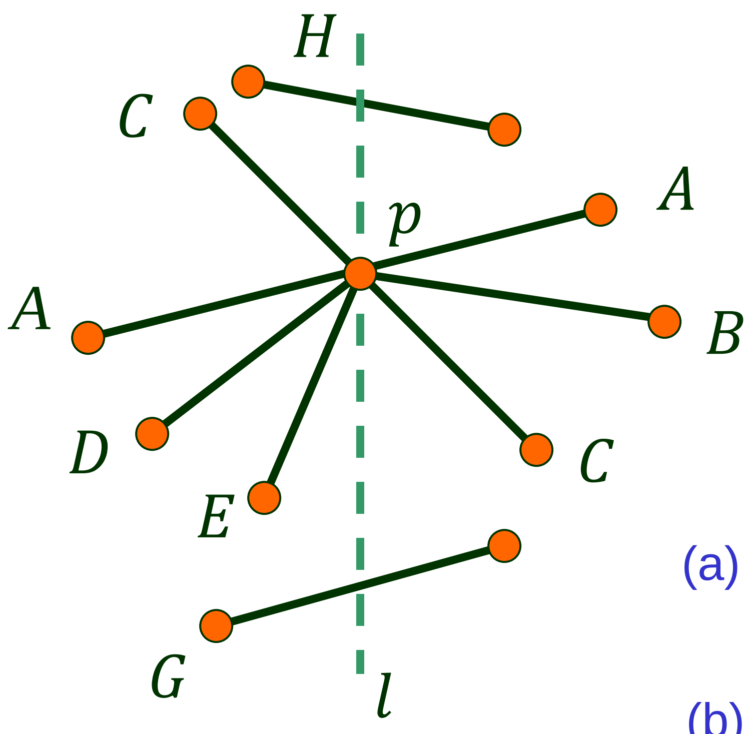


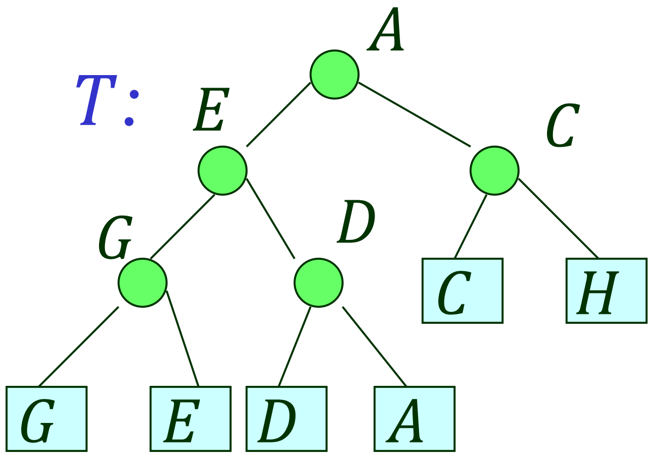

(a) Delete $D, E, A, C$
(all ending at p)

# Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: Several segments are involved in one event point (tricky).



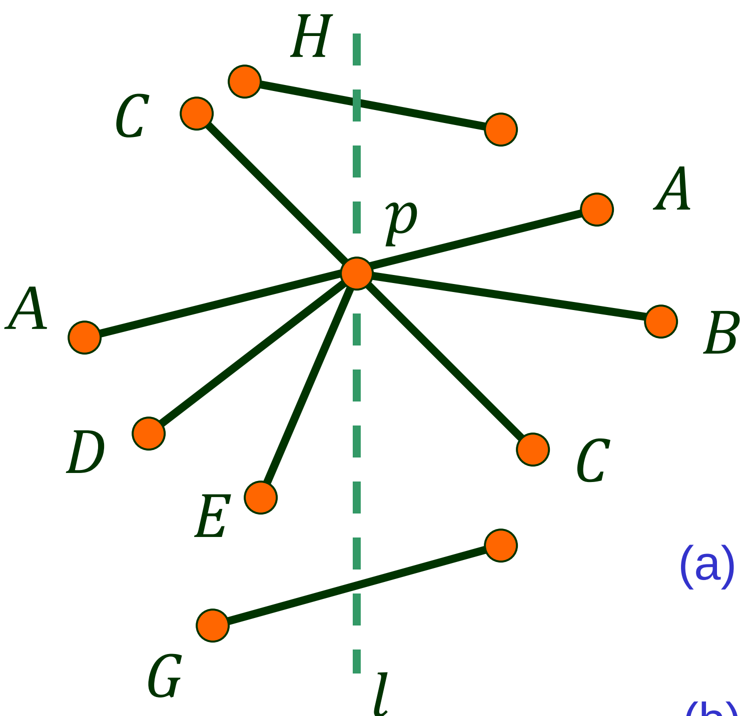

(a) Delete $D, E, A, C$
(all ending at p)

(b) Insert $B, A, C$
(all starting at p)

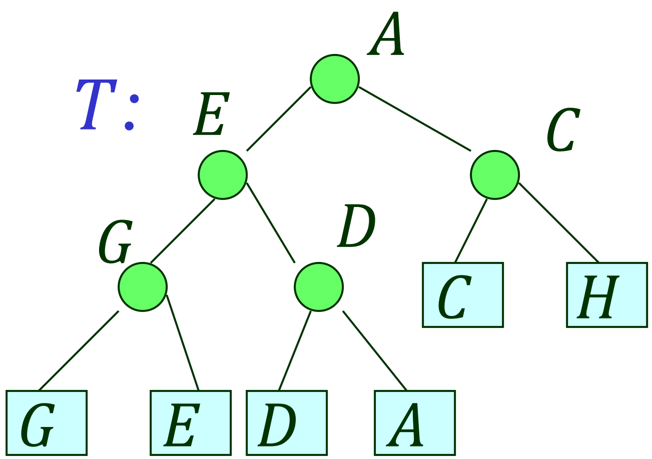
# Handling Event Points

Status updates (1) – (3) presented earlier.

Degeneracy: Several segments are involved in one event point (tricky).

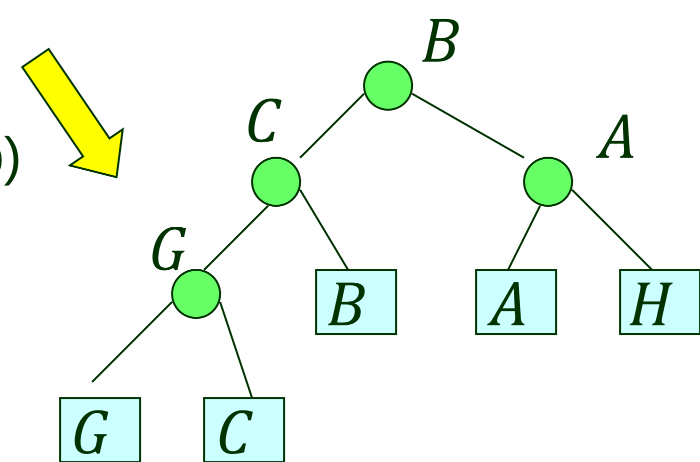



$T$:




(a) Delete $D, E, A, C$
(all ending at p)

(b) Insert $B, A, C$
(all starting at p)



# The Code for Event Handling

HandleEventPoint($p$)

$L(p) \leftarrow \{$ segments with left endpoint $p$ $\}$ // they are stored with $p$

$R(p) \leftarrow \{$ segments with right endpoint $p$ $\}$

$C(p) \leftarrow \{$ segments with $p$ in interior $\}$

**if** $|L \cup R \cup C| > 1$

  **then** report $p$ as an intersection along with $L, R, C$

$T \leftarrow T - (R \cup C)$

$T \leftarrow T \cup (L \cup C)$ // order as intersected by sweep line just to the right of $p$.
// segments in $C(p)$ have order reversed.

**if** $L \cup C = \emptyset$ // right endpoint only

  **then** let $s_a$ and $s_b$ be the neighbors right above and below $p$ in $T$

    FindNewEvent($s_b, s_a, p$)

  **else** $s' \leftarrow$ lowest segment in $L \cup C$

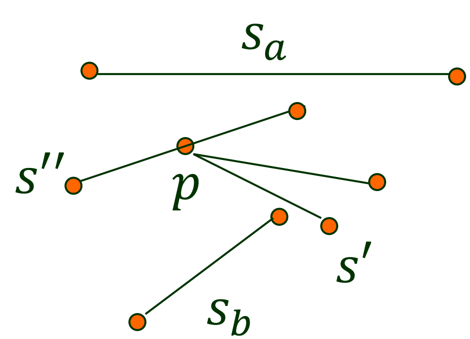
    $s_b \leftarrow$ segment right below $s'$

    FindNewEvent($s_b, s', p$)

    $s'' \leftarrow$ highest segment in $L \cup C$

    $s_a \leftarrow$ segment right above $s''$

    FindNewEvent($s'', s_a, p$)



# Finding New Event

FindNewEvent($s_l, s_r, p$)

1. if $s_l$ and $s_r$ intersect to the right of $p$ // sweep line position
2. then insert the intersection point as an event in $Q$

# IV. Time & Storage

The tree $T$ stores every segment once. $O(n)$

The size of the event queue $Q$: $O(n+I)$.

↑ # intersections

# IV. Time & Storage

The tree $T$ stores every segment once. $O(n)$

The size of the event queue $Q$: $O(n+I)$.

↑ # intersections

Reduce the storage to $O(n)$.

# IV. Time & Storage

The tree $T$ stores every segment once. $O(n)$

The size of the event queue $Q$: $O(n+I)$.

# intersections

Reduce the storage to $O(n)$.

- Store intersections among adjacent segments in the event queue.
- Delete those of segments that stop being adjacent.
- Before the deleted point is reached, the segments must have become adjacent again, resulting in the addition of the point to the even queue.

# IV. Time & Storage

The tree $T$ stores every segment once. $O(n)$

The size of the event queue $Q$: $O(n+I)$.

↑ # intersections

Reduce the storage to $O(n)$.

- Store intersections among adjacent segments in the event queue.
- Delete those of segments that stop being adjacent.
- Before the deleted point is reached, the segments must have become adjacent again, resulting in the addition of the point to the even queue.

**Theorem** All $I$ intersections of $n$ line segments in the plane can be reported in $O((n+I)\log n)$ time and $O(n)$ space.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.
  $R(p)$ and $C(p)$ are stored in $T$ and will be found.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.
  $R(p)$ and $C(p)$ are stored in $T$ and will be found.

} All involved are determined correctly.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.
  $R(p)$ and $C(p)$ are stored in $T$ and will be found.
  } All involved are determined correctly.

- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.
  $R(p)$ and $C(p)$ are stored in $T$ and will be found.

  } All involved are determined correctly.

- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$.

  All involved segments have $p$ in interior. Order them by polar angle.

# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

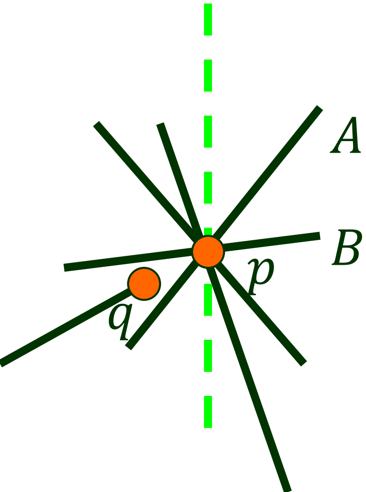
- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.
  $R(p)$ and $C(p)$ are stored in $T$ and will be found.

  } All involved are determined correctly.

- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$.

  All involved segments have $p$ in interior. Order them by polar angle.

  Let $A$ and $B$ be neighboring segments.



# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.
  $R(p)$ and $C(p)$ are stored in $T$ and will be found.

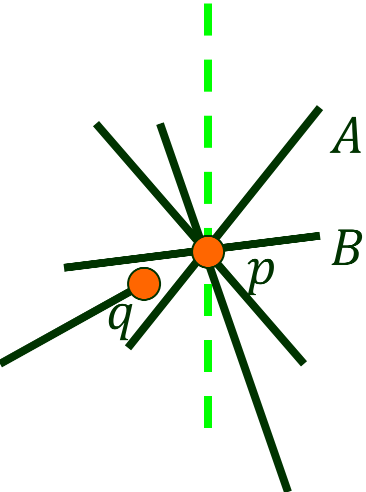
  } All involved are determined correctly.

- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$.

  All involved segments have $p$ in interior. Order them by polar angle.

  Let $A$ and $B$ be neighboring segments.

  $\Rightarrow$There exists event point $q < p$ after which $A$ and $B$ become adjacent..



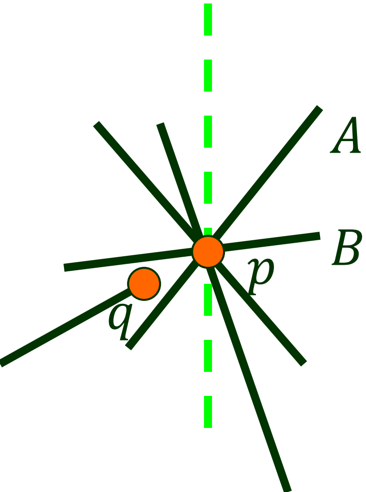
# Correctness

**Lemma** Algorithm FindIntersections computes all interesections and their containing segments correctly.

**Proof** By induction. Let $p$ be an intersection point and assume all intersections with a higher priority have been computed correctly.

- $p$ is an endpoint. $\Rightarrow$ stored in the event queue $Q$.
  $L(p)$ is also stored in $Q$.
  $R(p)$ and $C(p)$ are stored in $T$ and will be found.
  } All involved are determined correctly.

- $p$ is not an endpoint. We show that $p$ will be inserted into $Q$.




All involved segments have $p$ in interior. Order them by polar angle.

Let $A$ and $B$ be neighboring segments.

$\Rightarrow$ There exists event point $q < p$ after which $A$ and $B$ become adjacent..

$\Rightarrow$ By induction, $q$ was handled correctly and $p$ is detected and stored in $Q$

# Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

# Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time $O((n+k)\log n)$

# Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time $O((n+k)\log n)$

↑

output size (intersections and their containing segments)

# Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time $O((n+k)\log n)$

↑

output size
(intersections and their containing segments)

But one intersection may consist of $\Theta(n)$ segments.

# Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time $O((n+k)\log n)$

↑

output size
(intersections and their containing segments)

But one intersection may consist of $\Theta(n)$ segments.

When this happens, running time becomes $O(n^2 \log n)$

# Output Sensitivity

An algorithm is *output sensitive* if its running time is sensitive to the size of the output.

The plane sweeping algorithm is output sensitive.

running time $O((n+k)\log n)$

↑

output size
(intersections and their
containing segments)

But one intersection may consist of $\Theta(n)$ segments.

When this happens, running time becomes $O(n^2 \log n)$

Not tight! – the total number of intersections may still be $\Theta(n)$.

# A Tighter Bound

The running time is $O(n \log n + I \log n)$.

# A Tighter Bound

The running time is $O(n \log n + I \log n)$.

↑ # intersections

# A Tighter Bound

The running time is $O(n \log n + I \log n)$.

↑ # intersections

**Idea in analysis** all time cost is spent on maintaining the two data structures: 1) the event queue $Q$, and 2) the status structure tree $T$.

# A Tighter Bound

The running time is $O(n \log n + I \log n)$.

↑ # intersections

**Idea in analysis** all time cost is spent on maintaining the two data structures: 1) the event queue $Q$, and 2) the status structure tree $T$.

```
FindIntersections(S)
Q ← ∅      // initialize an empty event queue
Insert the segment endpoints into Q // balanced binary search tree O(n log n)
T ← ∅      // initialize an empty status structure
while Q ≠ ∅
    do extract the next event point p
       Q ← Q − {p}      // deletion from Q takes time O(log n).
       HandleEventPoint(p)  // 1 or 2 calls to FindNewEvent
```

# A Tighter Bound

The running time is $O(n \log n + I \log n)$.

# intersections

**Idea in analysis** all time cost is spent on maintaining the two data structures: 1) the event queue $Q$, and 2) the status structure tree $T$.

```
FindIntersections(S)
Q ← ∅      // initialize an empty event queue
Insert the segment endpoints into Q // balanced binary search tree O(n log n)
T ← ∅      // initialize an empty status structure
while Q ≠ ∅
    do extract the next event point p
        Q ← Q − {p}     // deletion from Q takes time O(log n).
        HandleEventPoint(p)  // 1 or 2 calls to FindNewEvent
                        // each call may result in an insertion into Q: O(log n).
```

# A Tighter Bound

The running time is $O(n \log n + I \log n)$.

↑ # intersections

**Idea in analysis** all time cost is spent on maintaining the two data structures: 1) the event queue $Q$, and 2) the status structure tree $T$.

```
FindIntersections(S)
Q ← ∅     // initialize an empty event queue
Insert the segment endpoints into Q // balanced binary search tree O(n log n)
T ← ∅     // initialize an empty status structure
while Q ≠ ∅
    do extract the next event point p
        Q ← Q − {p}      // deletion from Q takes time O(log n).
        HandleEventPoint(p)  // 1 or 2 calls to FindNewEvent

                  // each call may result in an insertion into Q: O(log n).
                  // each operation on T – insertion, deletion, neighbor
                  // finding – takes time O(log n).
```

# A Tighter Bound

The running time is $O(n \log n + I \log n)$.

↑ # intersections

**Idea in analysis** all time cost is spent on maintaining the two data structures: 1) the event queue $Q$, and 2) the status structure tree $T$.

```
FindIntersections(S)
1.  Q ← ∅     // initialize an empty event queue
2.  Insert the segment endpoints into Q // balanced binary search tree O(n log n)
3.  T ← ∅     // initialize an empty status structure
4.  while Q ≠ ∅
5.      do extract the next event point p
6.          Q ← Q − {p}      // deletion from Q takes time O(log n).
7.          HandleEventPoint(p)  // 1 or 2 calls to FindNewEvent

                      // each call may result in an insertion into Q: O(log n).
                      // each operation on T – insertion, deletion, neighbor
                      // finding – takes time O(log n).
                      // #operations on T is Θ(|L(p) ∪ R(p) ∪ C(p)|)
```

# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

**Claim** $m = O(n + I)$.

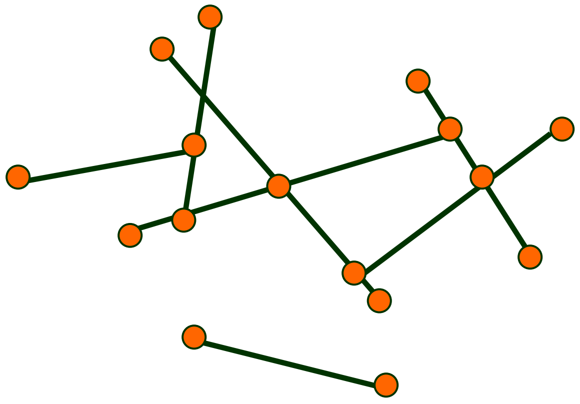
# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

**Claim** $m = O(n+I)$.

**Proof** View the set of segments as a *planar graph*.



# Total Number of Tree Operations

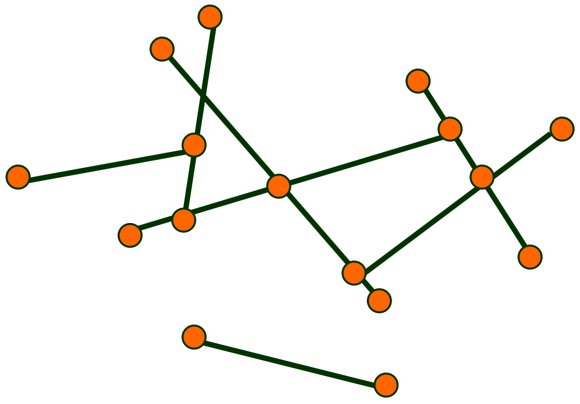
Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

**Claim** $m = O(n+I)$.

**Proof** View the set of segments as a *planar graph*.

$|L(p) \cup R(p) \cup C(p)| \leq \deg(p)$



# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

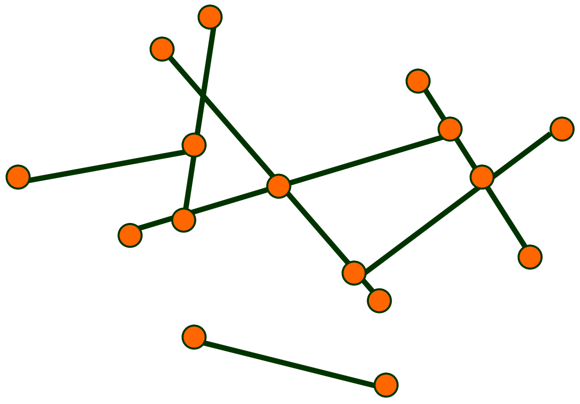
Then the running time is $O\big((m+n)\log n\big)$.

**Claim** $m = O(n+I)$.

Each in the set contributes 2 to the degree

**Proof** View the set of segments as a *planar graph*.

$|\, L(p) \cup R(p) \cup C(p)| \leq \deg(p)$



# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

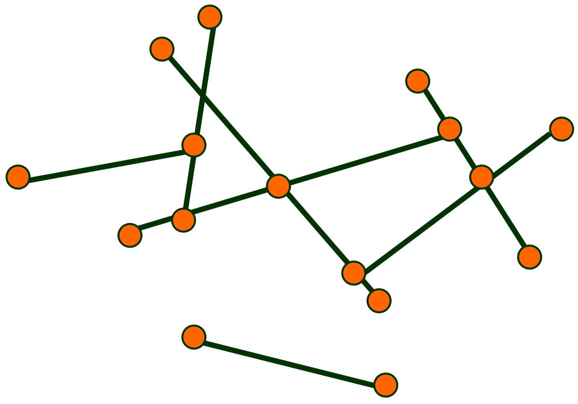
Each in the set contributes
2 to the degree

**Claim** $m = O(n+I)$.

**Proof** View the set of segments as a *planar graph*.

$|L(p) \cup R(p) \cup C(p)| \le \deg(p)$

$\Longrightarrow m \le \sum_p \deg(p)$



# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

**Claim** $m = O(n+I)$.

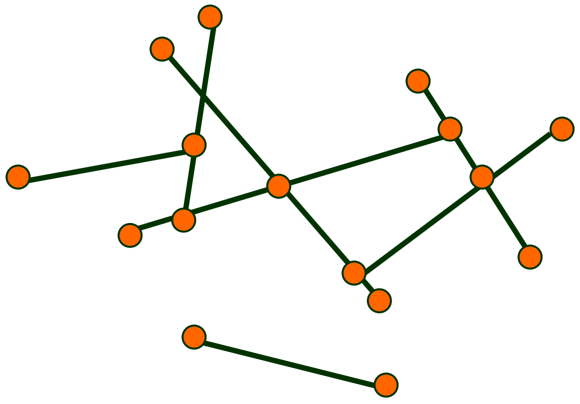
Each in the set contributes
2 to the degree

**Proof** View the set of segments as a *planar graph*.

$|L(p) \cup R(p) \cup C(p)| \le \deg(p)$

$\Longrightarrow m \le \sum_p \deg(p)$

$n_e$ = #edges $n_v$ = #vertices $n_f$ = #regions



# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

**Claim** $m = O(n+I)$.

Each in the set contributes
2 to the degree

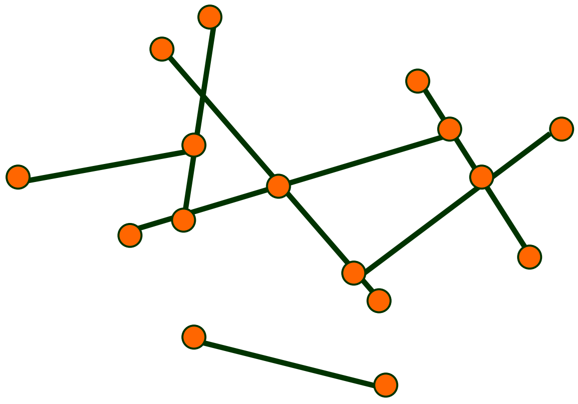
**Proof** View the set of segments as a *planar graph*.

$|L(p) \cup R(p) \cup C(p)| \leq \deg(p)$

$\Longrightarrow m \leq \sum_p \deg(p)$

$n_e$ = #edges  $n_v$ = #vertices  $n_f$ = #regions

Every edge contributes one to the degree of each of its two vertices.



# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

**Claim** $m = O(n + I)$.

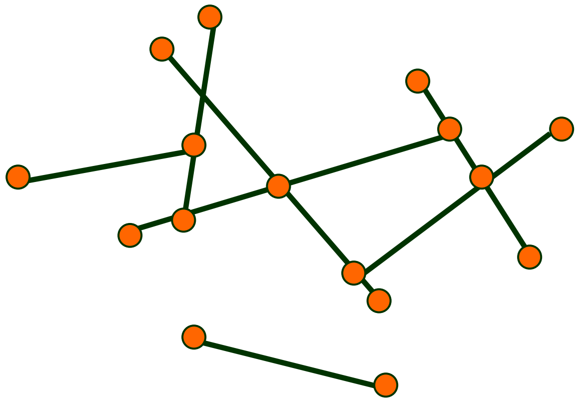
Each in the set contributes 2 to the degree

**Proof** View the set of segments as a *planar graph*.

$|L(p) \cup R(p) \cup C(p)| \leq \deg(p)$

$\Longrightarrow \; m \leq \sum_p \deg(p)$

$n_e$ = #edges $n_v$ = #vertices $n_f$ = #regions



Every edge contributes one to the degree of each of its two vertices.

$$\sum_p \deg(p) = 2n_e$$

# Total Number of Tree Operations

Let $m = \sum_p |L(p) \cup R(p) \cup C(p)|$

Then the running time is $O\big((m+n)\log n\big)$.

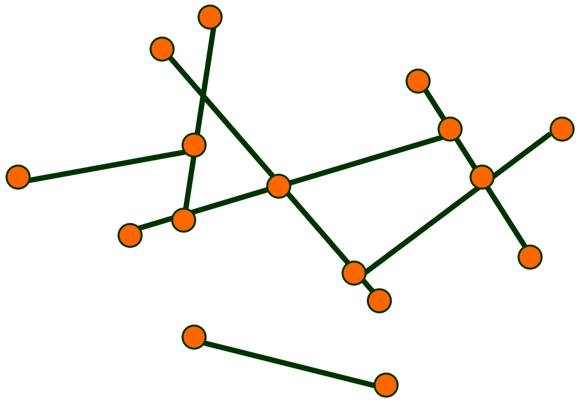
**Claim** $m = O(n + I)$.

Each in the set contributes 2 to the degree

**Proof** View the set of segments as a *planar graph*.

$|L(p) \cup R(p) \cup C(p)| \le \deg(p)$

⟹ $m \le \sum_p \deg(p)$



$n_e = \#\text{edges}$ $n_v = \#\text{vertices}$ $n_f = \#\text{regions}$

Every edge contributes one to the degree of each of its two vertices.

$$\sum_p \deg(p) = 2n_e$$

⟹ $m \le 2n_e$

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e$$

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3} n_e$$

$n_v \leq 2n + I$ (some endpoints may be also intersections)

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \le 2n_e \Rightarrow n_f \le \frac{2}{3} n_e$$

$$n_v \le 2n + I$$ (some endpoints may be also intersections)

Euler's equation:

# connected components

$$n_v - n_e + n_f = l + 1 \ge 2$$

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e$$

$$n_v \leq 2n + I$$ (some endpoints may be also intersections)

Euler's equation:

# connected components ↓

$$n_v - n_e + n_f = l + 1 \geq 2$$

⇓ $\frac{2}{3}n_e \geq n_f$

$$n_v - n_e + \frac{2}{3}n_e \geq n_v - n_e + n_f \geq 2$$

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3} n_e$$

$$n_v \leq 2n + I$$ (some endpoints may be also intersections)

Euler's equation:

# connected components ↓

$$n_v - n_e + n_f = l + 1 \geq 2$$

⇓ $\frac{2}{3} n_e \geq n_f$

$$n_v - n_e + \frac{2}{3} n_e \geq n_v - n_e + n_f \geq 2$$

⇓

$$n_e \leq 3n_v - 6$$

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e$$

$n_v \leq 2n + I$ (some endpoints may be also intersections)

Euler's equation:

# connected components ↓

$$n_v - n_e + n_f = l + 1 \geq 2$$

⇓ $\frac{2}{3}n_e \geq n_f$

$$n_v - n_e + \frac{2}{3}n_e \geq n_v - n_e + n_f \geq 2$$

⇓

$$n_e \leq 3n_v - 6$$

⇓ $n_v \leq 2n + I$

$$n_e \leq 6n + 3I - 6$$

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e$$

$$n_v \leq 2n + I$$ (some endpoints may be also intersections)

Euler's equation:

# connected components ↓

$$n_v - n_e + n_f = l + 1 \geq 2$$

⇓ $\frac{2}{3}n_e \geq n_f$

$$n_v - n_e + \frac{2}{3}n_e \geq n_v - n_e + n_f \geq 2$$

⇓

$$n_e \leq 3n_v - 6$$

⇓ $n_v \leq 2n + I$

$$n_e \leq 6n + 3I - 6$$

⇓

$$m \leq 2n_e \leq 12n + 6I - 12 = O(n + I)$$

# Cont'd

Meanwhile, a region is bounded by at least three edges, while an edge bounds at most two regions.

$$3n_f \leq 2n_e \Rightarrow n_f \leq \frac{2}{3}n_e$$

$n_v \leq 2n + I$ (some endpoints may be also intersections)

Euler's equation:

\# connected components

$$n_v - n_e + n_f = l + 1 \geq 2$$

$$\Downarrow \quad \frac{2}{3}n_e \geq n_f$$

$$n_v - n_e + \frac{2}{3}n_e \geq n_v - n_e + n_f \geq 2$$

$$\Downarrow$$

$$n_e \leq 3n_v - 6$$

$$\Downarrow \quad n_v \leq 2n + I$$

$$n_e \leq 6n + 3I - 6$$

$$\Downarrow$$

$$m \leq 2n_e \leq 12n + 6I - 12 = O(n + I)$$