Real-Time Rendering

Plane Sweep

WS 2019/20

Prof. Rhadamés Carmona

Universidad Central de Venezuela
Computer Graphics Center

Bauhaus-Universität Weimar

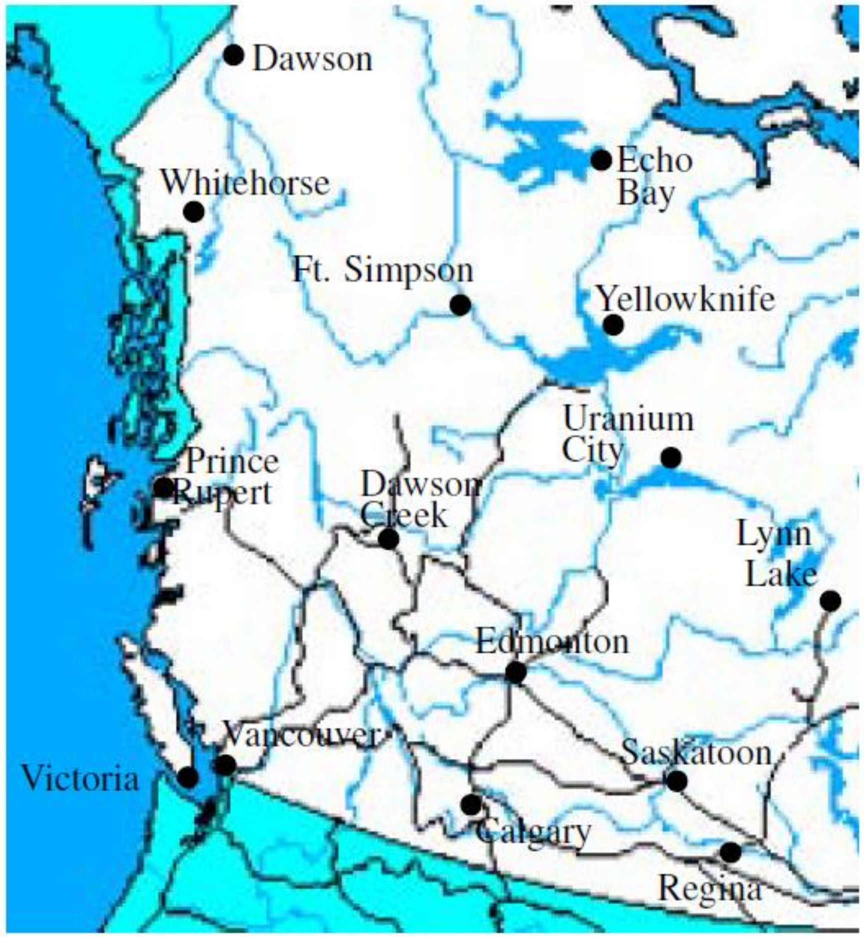
Virtual Reality and Visualization Research

Literature

- Computational Geometry, Algorithms and Applications (3rd edition), Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars, Springer 2008
  Course follows partially this book
  Lecture design and slide set by M. v. Kreveld
- Inspired by further lectures
  - Bernd Fröhlich, Real Time Rendering, 2012
  - Pjotr Indyk, MIT
  - Thomas Ottmann, Freiburg
  - Bernd Gärtner, Michael Hoffmann, ETH
  - Prof. Stefan Schirra, Madgeburg

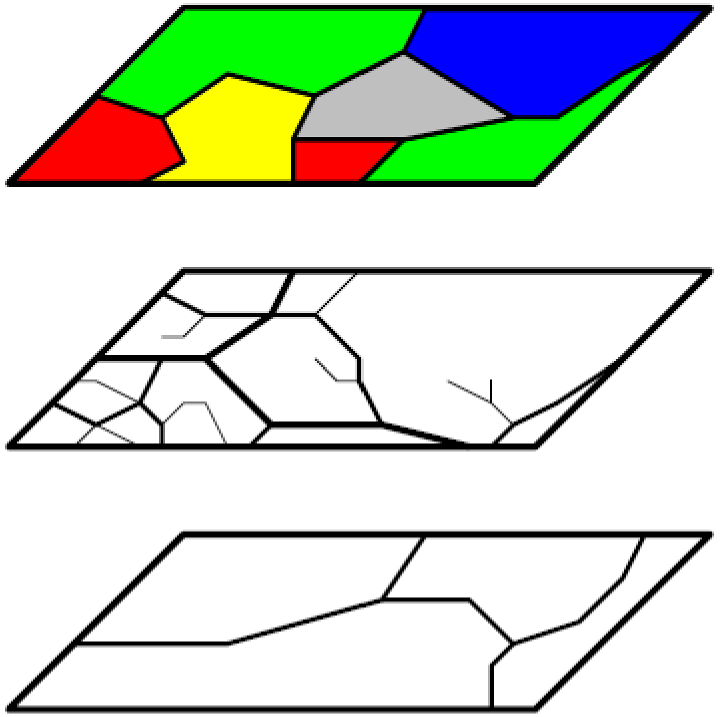
Motivation: maps

- Maps may contain cities, rivers, railroads, population info, epidemic clusters, hunger info, and so on
- It is difficult to find information when all information is together
- Information should be separated in someway



Motivation: GIS

- Geographic information systems (GIS) store data in separate layers
- A layer stores the geometric information about some aspect, like land cover, road network, municipality boundaries, red fox habitat, vegetation, rivers…
- Most information is stored as vector data, i.e. line segments. Curves are approximated by small segments

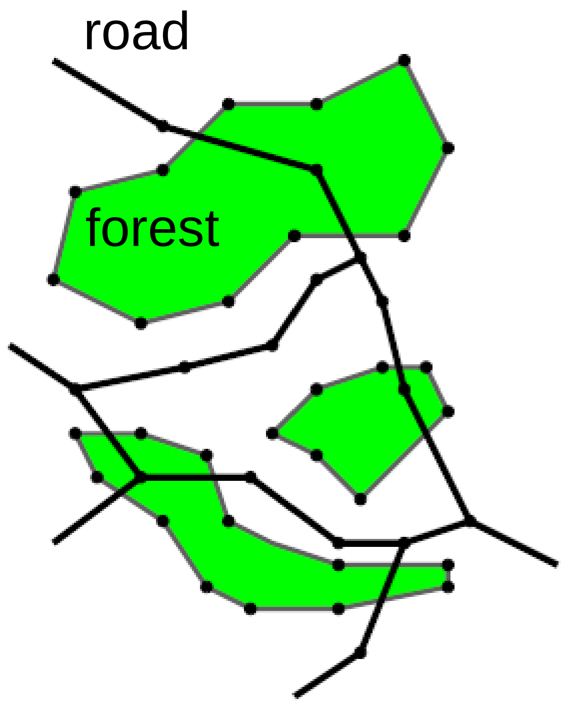


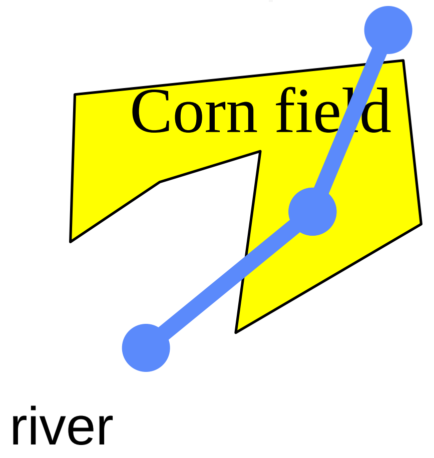
Motivation: GIS

- Map overlay is the combination of two (or more) map layers

- Needed to answer questions such as
  - What is the total length of roads through forests?
  - What is the total area of corn fields at the left of a river?

- Problem: need to compute (at least) intersection points of two layers (intersections of two sets of line segments)



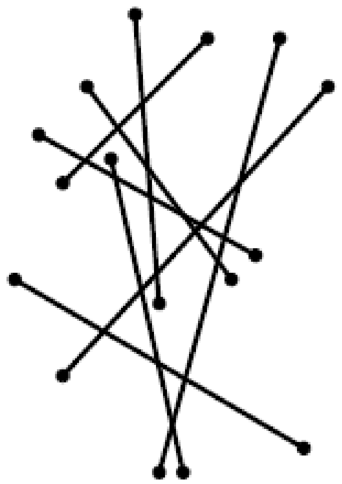


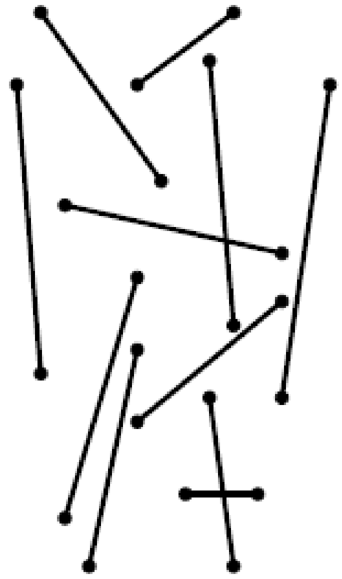
Problem

- Given 2 set of lines, find the intersection points between one set and the other set
- A more general approach is simply combining all lines in just one set, and find the intersection points between all lines. Intersections between lines of same set can be discarded or ignored during this process
- Brute force approach would compare every possible pair of segments $O(n^2)$
- In practical situations the number of intersections is very smaller than n^2
- May the running time depends of the output size?

Output-Sensitive Algorithms

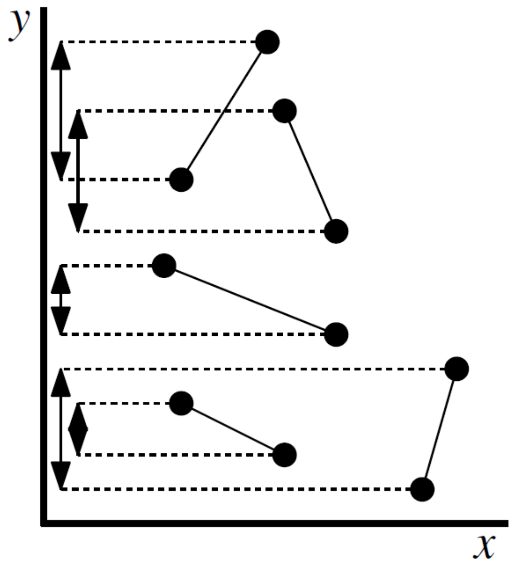
- The running time of an algorithm is always input-sensitive (depends on *n*)
- Output-sensitive algorithms: the running time depends on the size of the output
- In this problem, output-sensitive is “intersections-sensitive”, since the number of intersections determines the output size
- Question: may we avoid comparing every pair of lines?





Observation

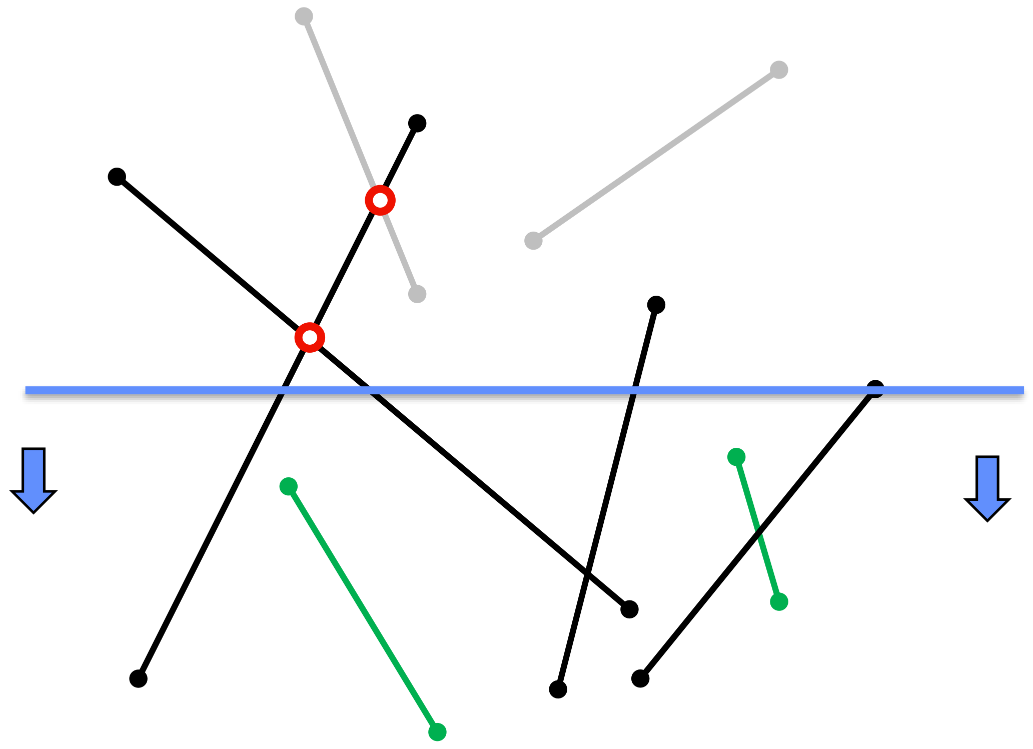
- Segments that are close together are candidates for intersection, unlike segments that are far apart
- We do not have to test every pair or segments for intersection
- First observation: if two lines do not overlap in y axis, they cannot intersect (they are far apart in y)



We can sort the segments by y axis, and process the lines one by one detecting overlapping in y axis. Instead of 5*4 pairs, how many pairs are tested in the example?

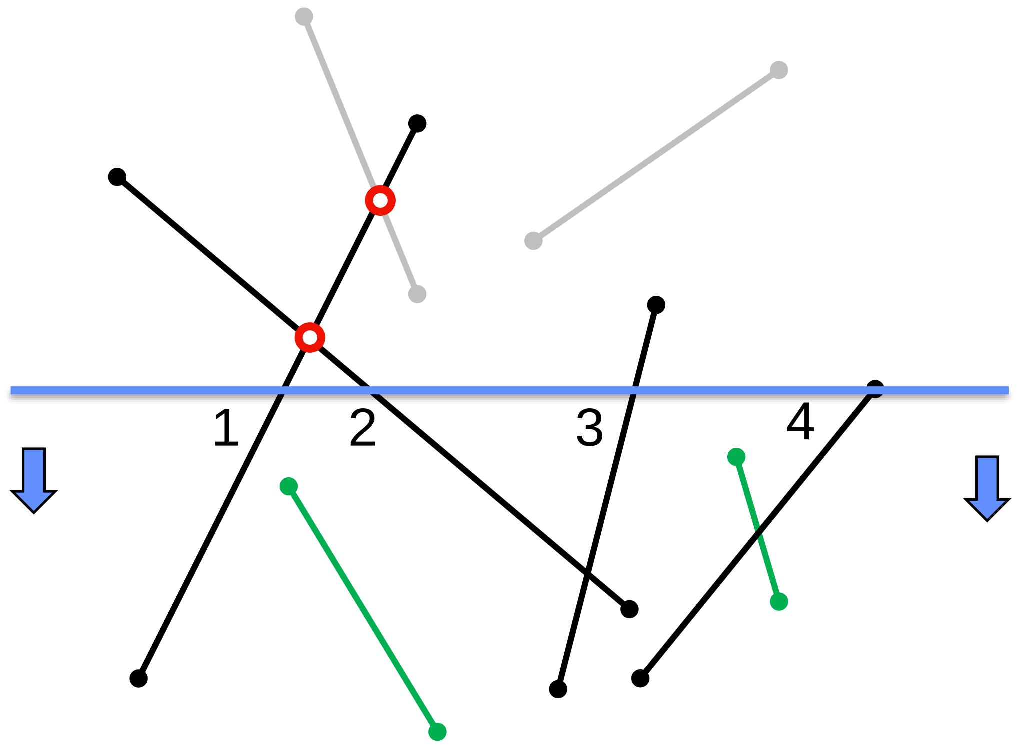
Plane-sweep or sweep-line Technique

- How to determine the pairs to process?
- Let's consider a sweep line, which is sweeping the plane, from ymax to ymin



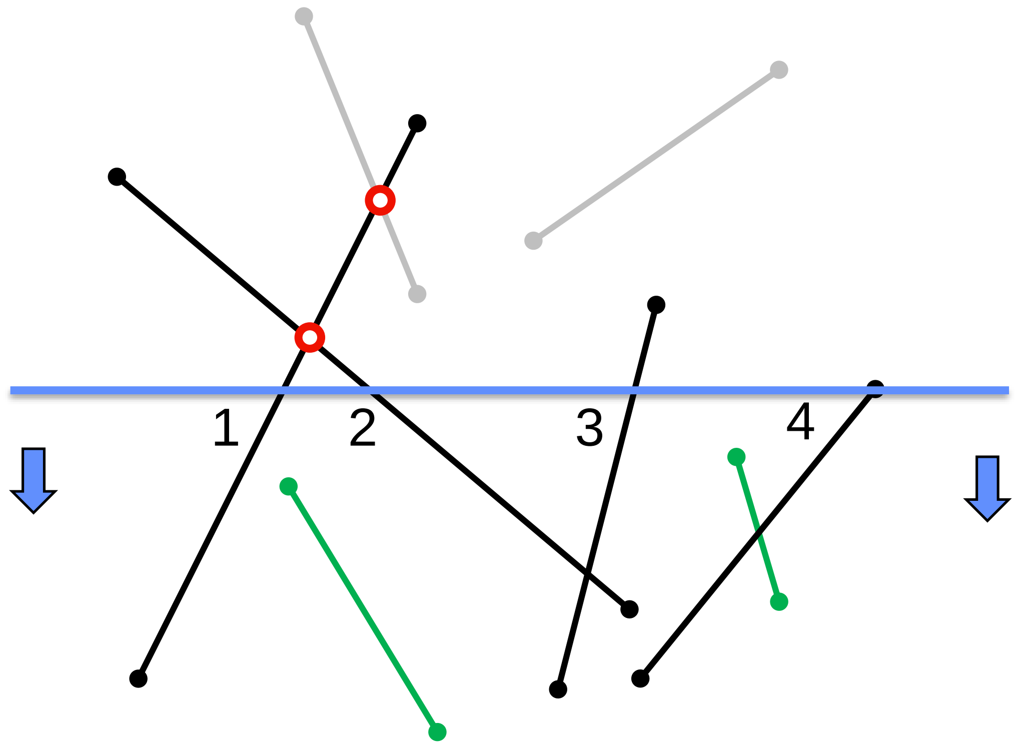
Plane-sweep or sweep-line Technique

- While the sweep line is moved from the top to the bottom, adjacent lines overlapping the sweep line may be “close” in x-direction and they are candidates to intersect



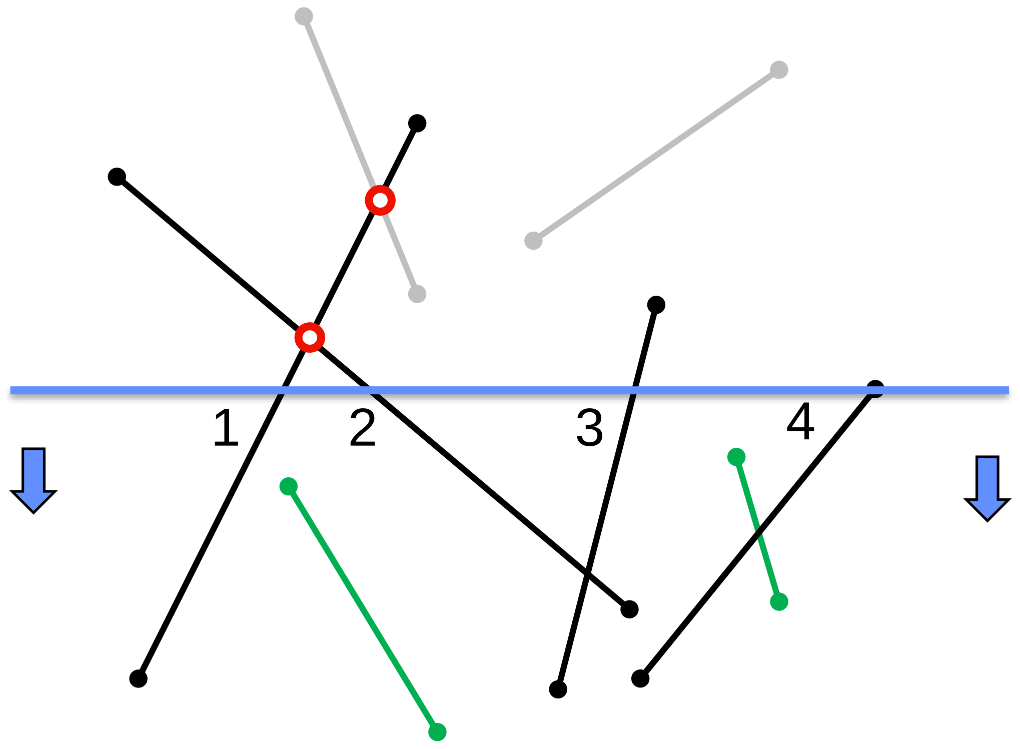
Plane-sweep or sweep-line Technique

- Active lines are sorted from left to right, because adjacent lines are candidates for intersection



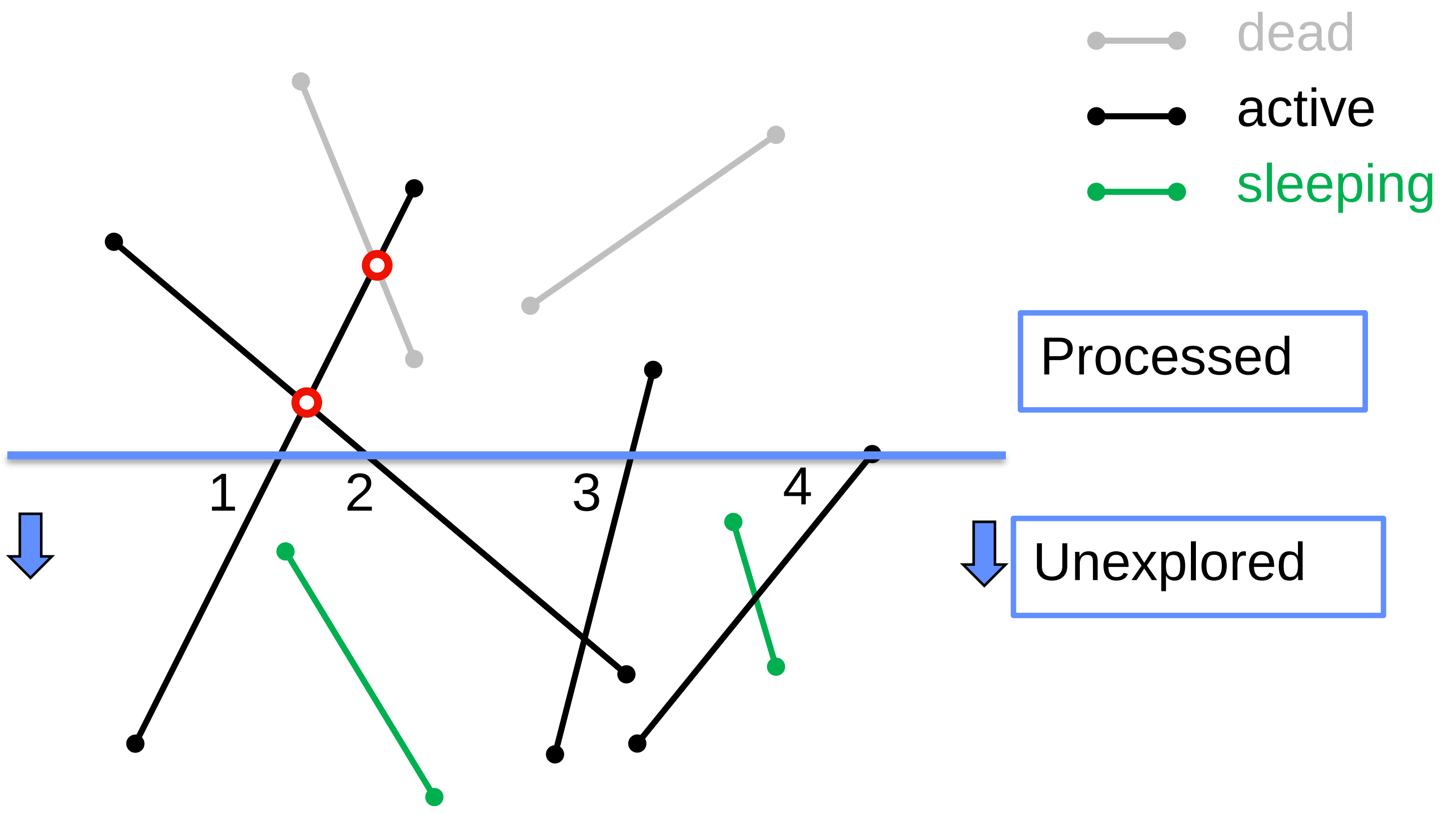
Plane-sweep or sweep-line Technique

- Considerations:
  - 1. We do not want to sort the active lines every time we move the sweep ➔ let's reuse the previous sorting when we move the sweep line, minimizing updates



Plane-sweep or sweep-line Technique

- Considerations:
  - 2. We want to move the sweep line at discrete positions ➔ let's consider the possible events (Q) that can modify the active list (T)



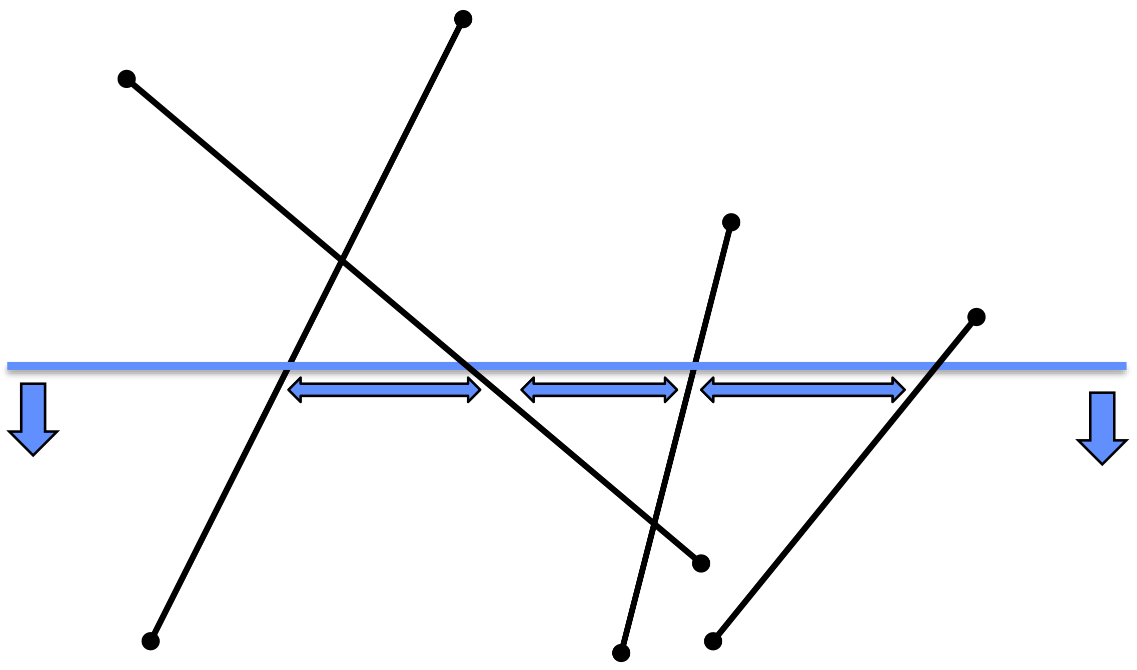
Plane-sweep or sweep-line Technique

- Object status
  - Dead: objects completely passed by the sweep line
  - **Active**: objects intersected by the sweep line
  - **Sleeping**: objects not yet reached by the sweep line

- **Event structure Q** maintains discrete events, called event points or transition points, where the status of the sweep line with respect to the objects changes
- **Status structure T** maintains information on the interaction of the sweep line with the active objects

Plane Sweep for Segment Intersections

- **Event structure Q** contains initially all start and end points of all line segments. Intersections are added on the fly
- **Status structure T** maintains the list of active segments in sorted x-order along the the sweep line in its current position

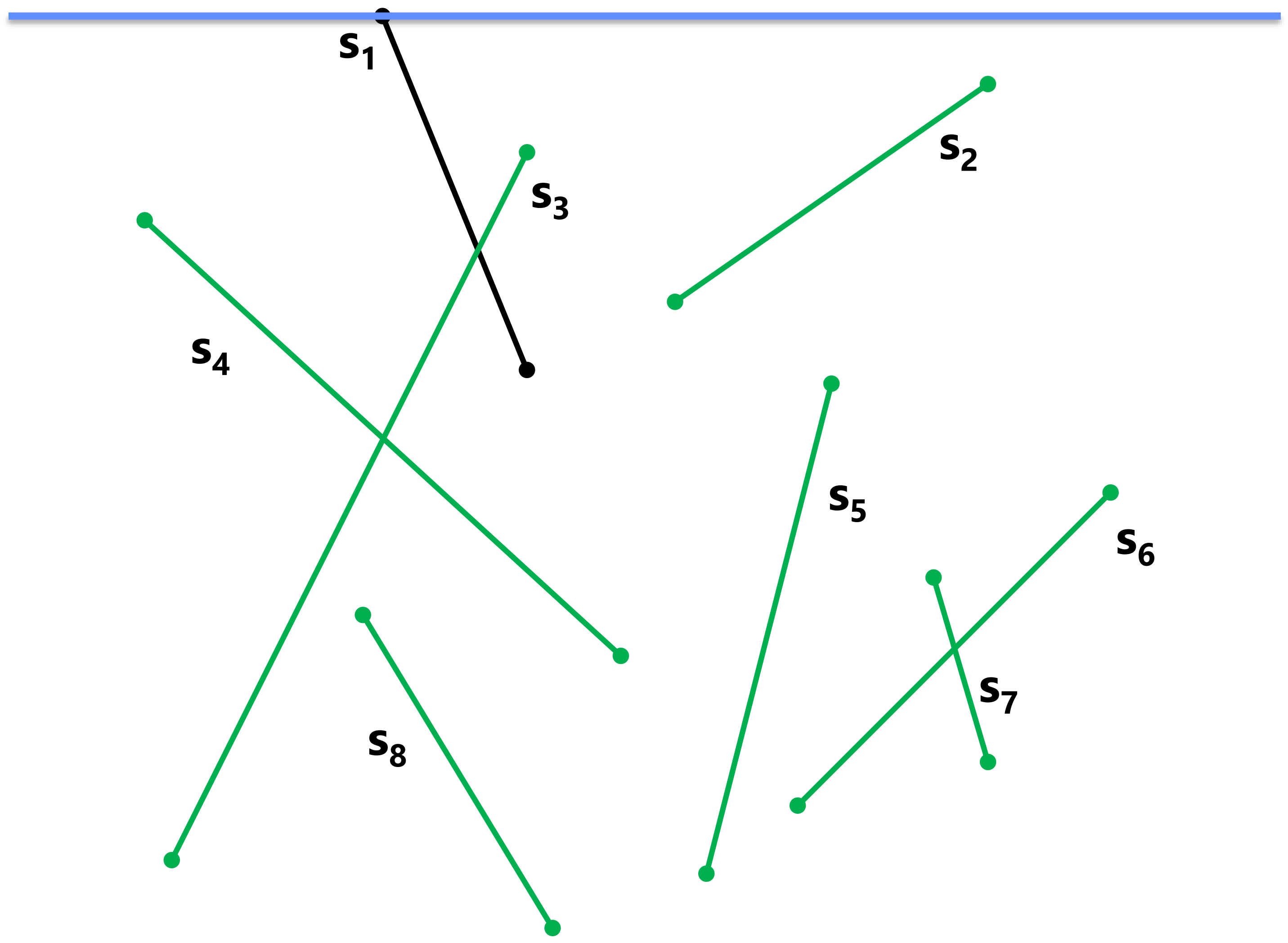


Let’s find out how it should work

- Considering the initial ideas:
  - Q contains the endpoints of the segments, sorted by y-axis. These endpoints are processed one by one. Eventually, a intersection point is inserted into this structure
  - T contains the segments intersecting the current sweep line, sorted from left to right. When a segment is included in T (from the upper point of the segment) such a segment is adjacent to 0, 1 or 2 segments. Thus intersection point(s) may be generated and inserted into Q accordingly
  - When the lower endpoint of a segment is processed, the segment is removed from T, and its left and right neighbors are now adjacent. Thus an intersection point may be generated and inserted into Q accordingly
  - When an intersection point is processed, its segments are swapped in T structure. Intersection may occur with their new neighbors. Intersection point(s) may be generated…

$T = \{s_1,\}$

$Q = \{\cancel{\uparrow s_1}, \uparrow s_2, \uparrow s_3, \uparrow s_4, \downarrow s_2, \downarrow s_1, \uparrow s_5, \uparrow s_6, \ldots\}$



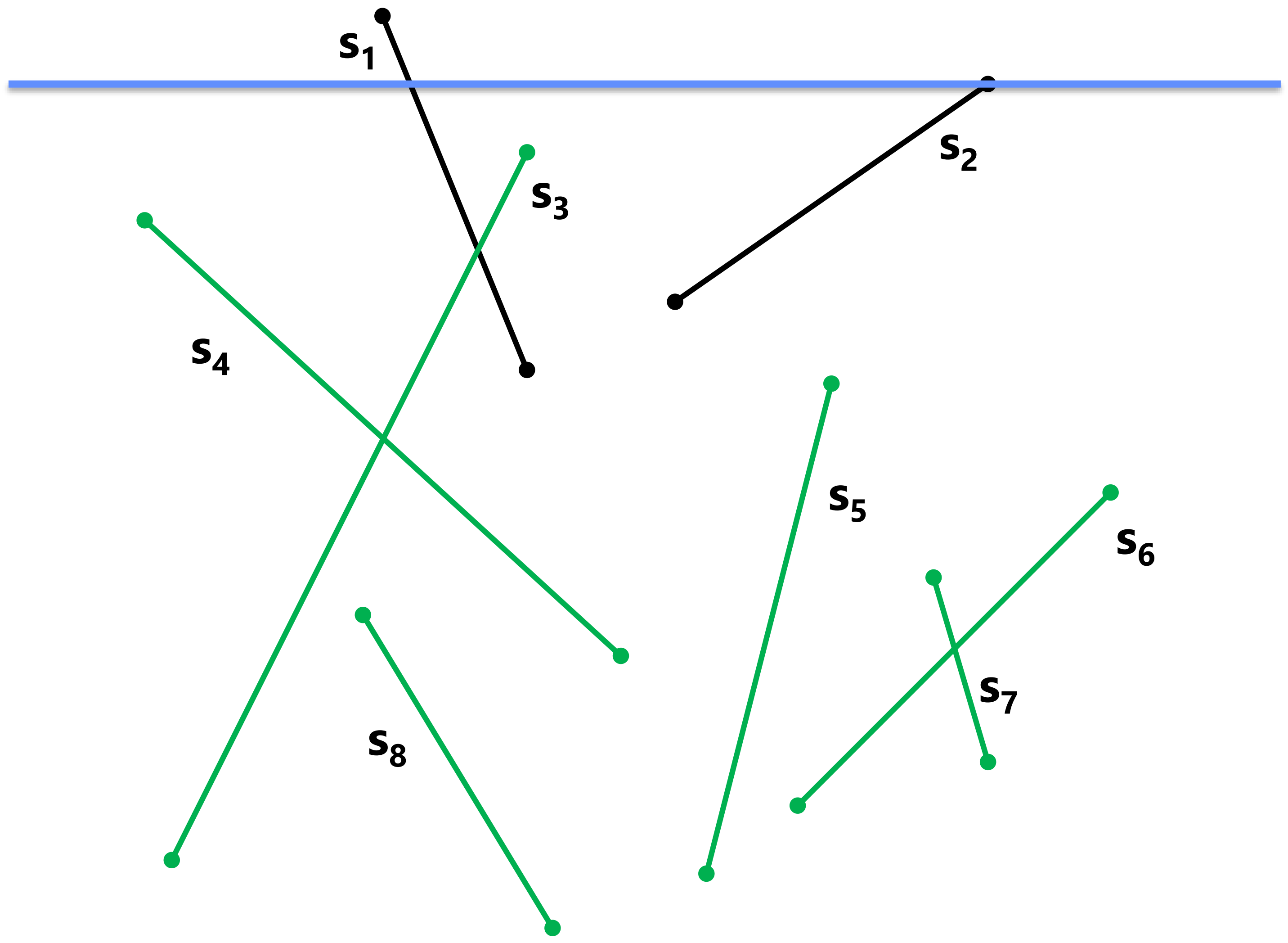
Operations on status structure T
Insert s_1

Operations on event structure Q
del $\uparrow s_1$

s_1 is now “active”

$T = \{s_1, s_2,\}$

$Q = \{$~~$\uparrow s_1, \uparrow s_2,$~~$\uparrow s_3, \uparrow s_4, \downarrow s_2, \downarrow s_1, \uparrow s_5, \uparrow s_6, \ldots\}$

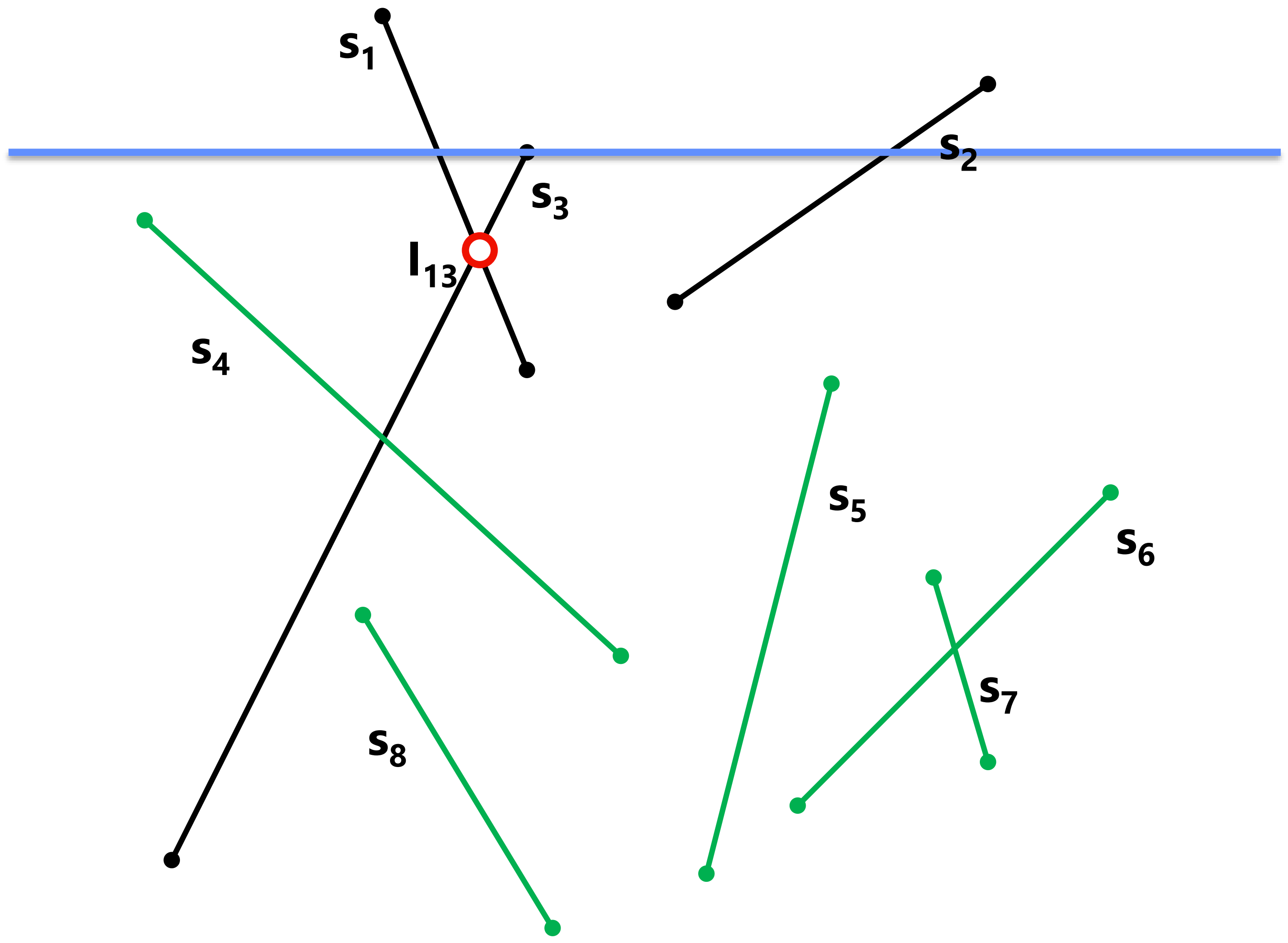


Operations on status structure T
Insert **s_2**
Operations on event structure Q
del $\uparrow$**s_2**

s_1 and **s_2** do not intersect… **s_2** is now “active”

$T = \{s_1, s_3, s_2, \}$

$Q = \{$~~$\uparrow s_1, \uparrow s_2, \uparrow s_3,$~~$\uparrow s_4, I_{13}, \downarrow s_2, \downarrow s_1, \uparrow s_5, \uparrow s_6, \ldots\}$



Operations on status structure T

insert $\mathbf{s_3}$

Operations on event structure Q

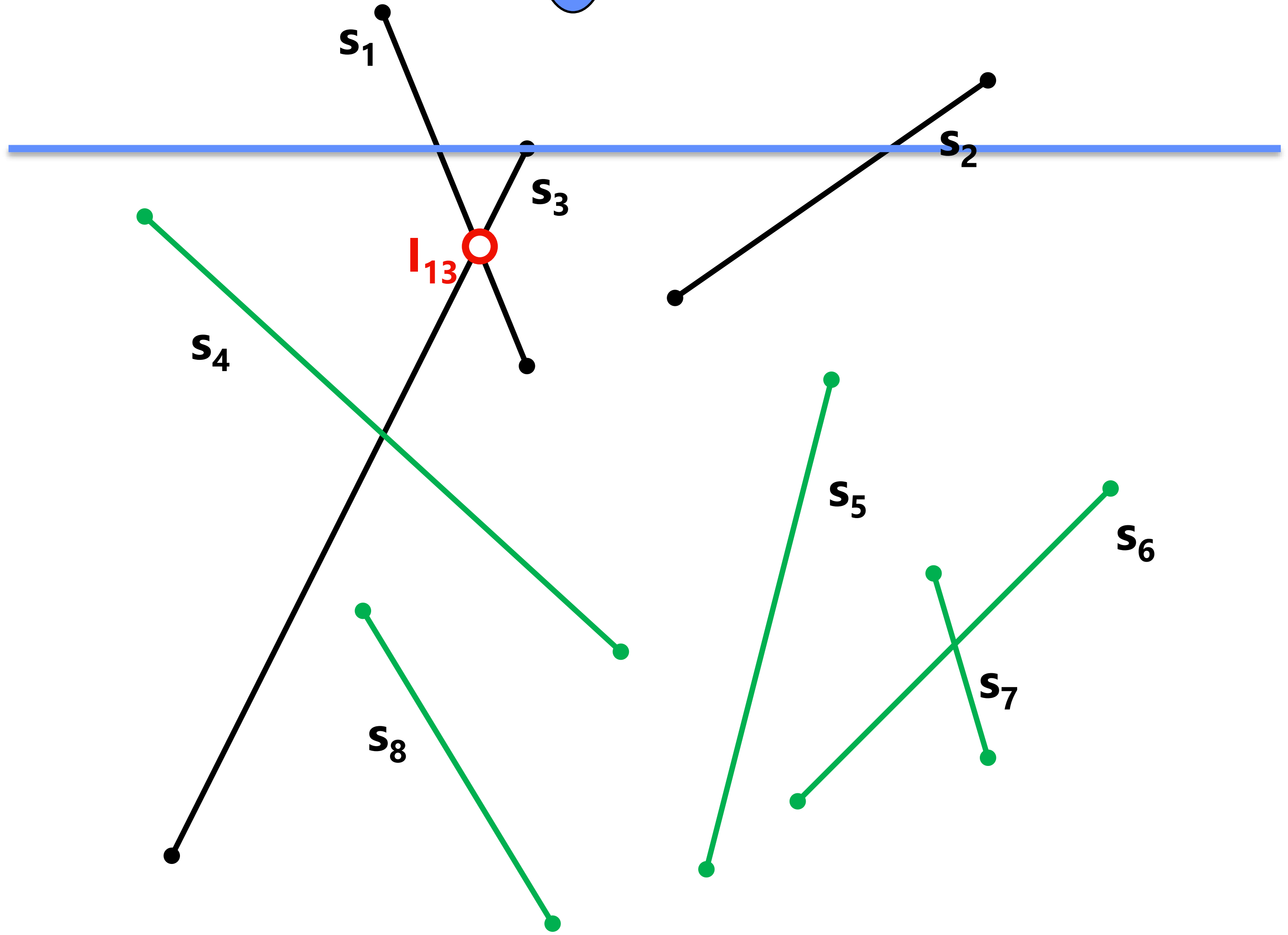
del $\uparrow \mathbf{s_3}$
insert $\mathbf{I_{13}}$

$\mathbf{s}_1$ and $\mathbf{s}_3$ intersect in $\mathbf{I_{13}}$; $\mathbf{s}_3$ is now “active”

Q is sorted by “y” axis

T = $\{s_1, s_3, s_2, \}$

Q = $\{$~~$\uparrow s_1, \uparrow s_2, \uparrow s_3,$~~ $\uparrow s_4, I_{13}, \downarrow s_2, \downarrow s_1, \uparrow s_5, \uparrow s_6, \ldots\}$

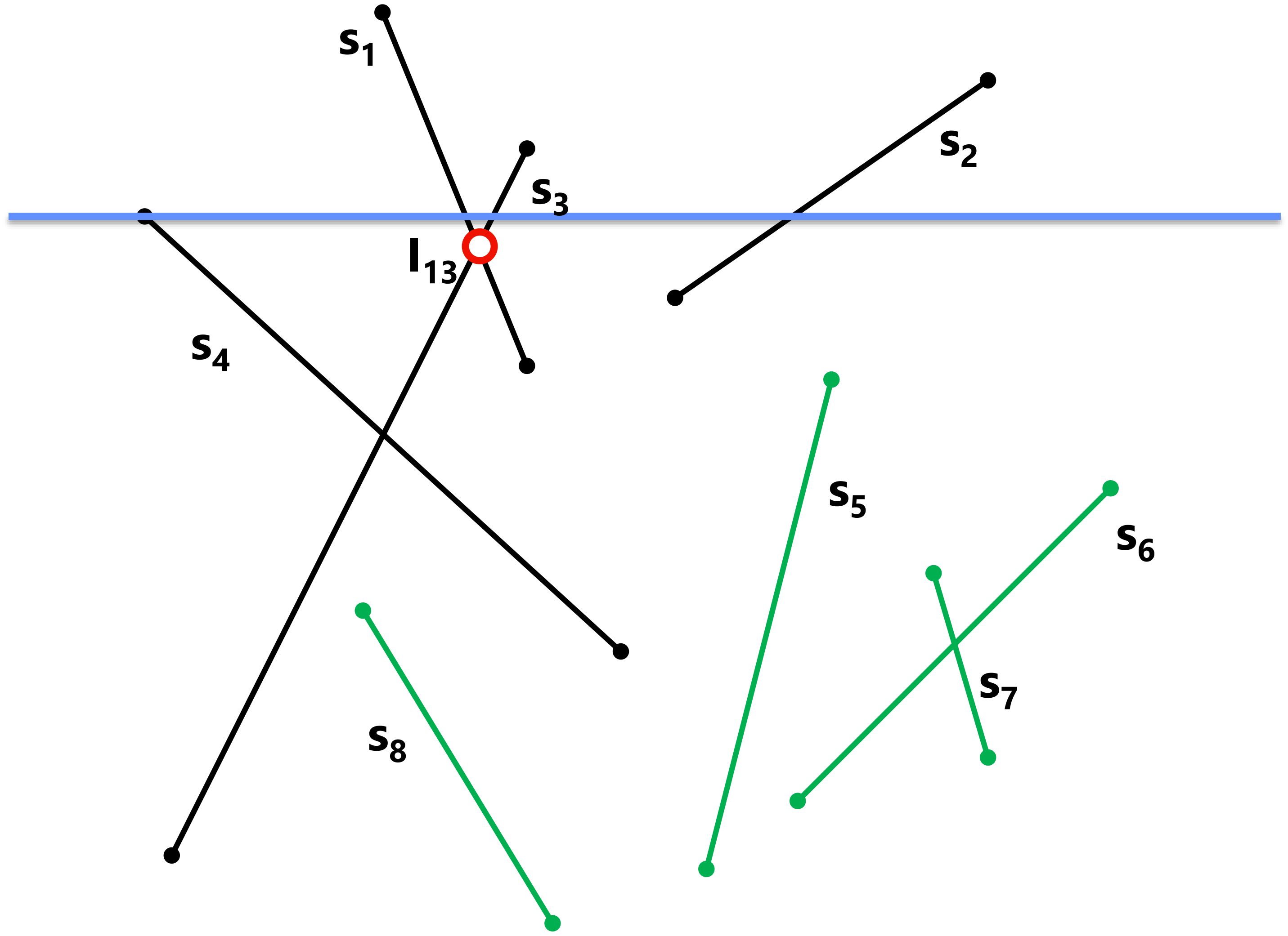


s_1 and **s_3** intersect in **I_{13}**; **s_3** is now “active”

Operations on status structure T
insert **s_3**
Operations on event structure Q
del $\uparrow$ **s_3** insert **I_{13}**

$T = \{s_4, s_1, s_3, s_2, \}$

$Q = \{$~~$\uparrow s_1, \uparrow s_2, \uparrow s_3, \uparrow s_4,$~~ $\mathbf{I_{13}}, \downarrow s_2, \downarrow s_1, \uparrow s_5, \uparrow s_6, \ldots\}$



Operations on status structure T
insert $\mathbf{s_4}$

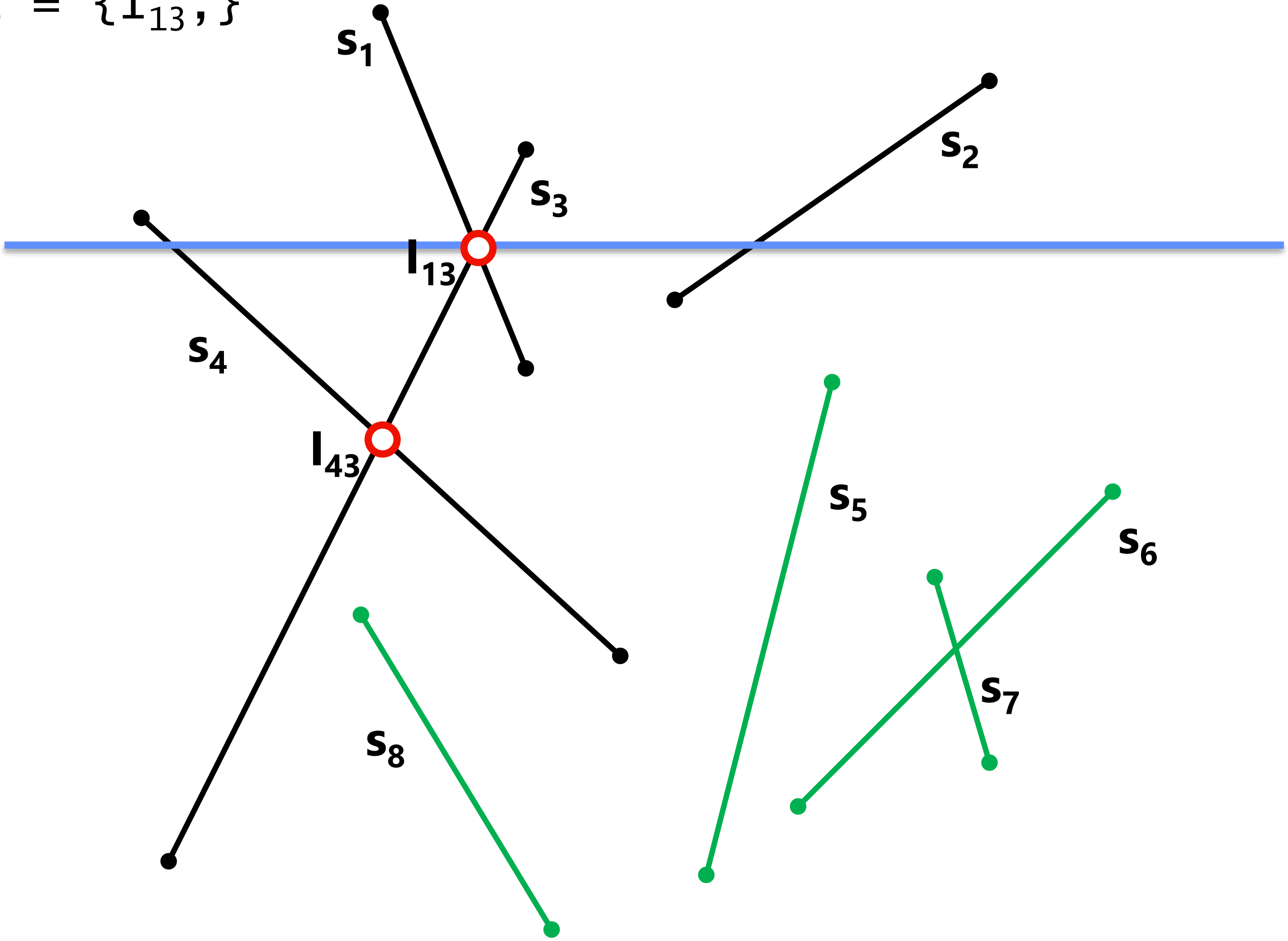
Operations on event structure Q
del $\uparrow \mathbf{s_4}$

$\mathbf{s_4}$ and $\mathbf{s_1}$ do not intersect…$\mathbf{s_4}$ is now “active”

$T = \{s_4, \overset{\curvearrowright}{s_3, s_1}, s_2,\}$

$Q = \{$ ~~$\uparrow s_1, \uparrow s_2, \uparrow s_3, \uparrow s_4, I_{13}$~~ $, \downarrow s_2, \downarrow s_1, \uparrow s_5, I_{43}, \uparrow s_6, \ldots\}$

$L = \{I_{13},\}$



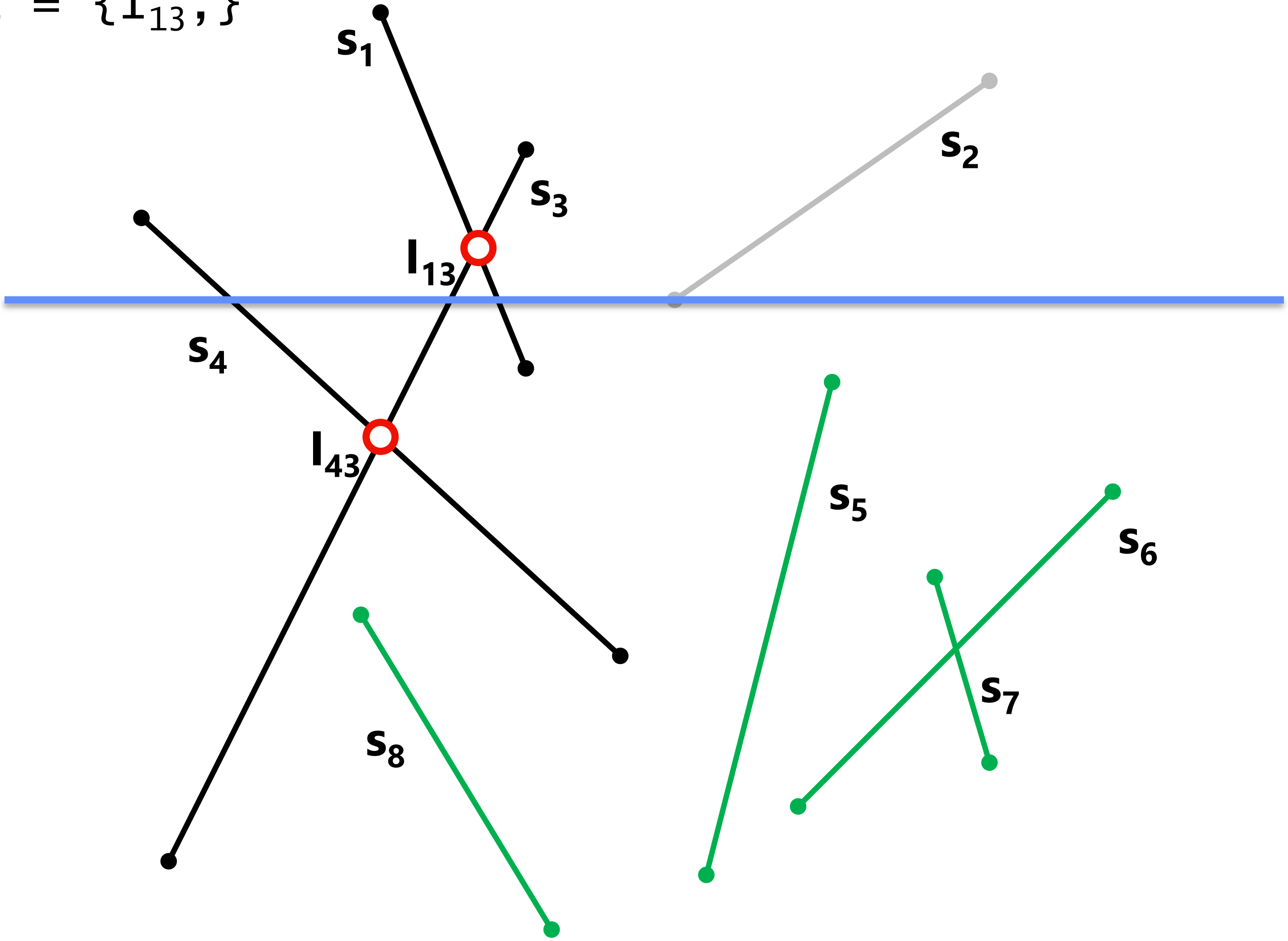
s_4 and **s_3** intersect in **I_{43}**…

Operations on status structure T
report **I_{13}** swap **s_1,s_3**
Operations on event structure Q
del **I_{13}** insert **I_{43}**

$T = \{s_4, s_3, s_1\}$

$Q = \{$~~$\uparrow s_1, \uparrow s_2, \uparrow s_3, \uparrow s_4, I_{13}, \downarrow s_2,$~~$\downarrow s_1, \uparrow s_5, I_{43}, \uparrow s_6, \ldots\}$

$L = \{I_{13}, \}$



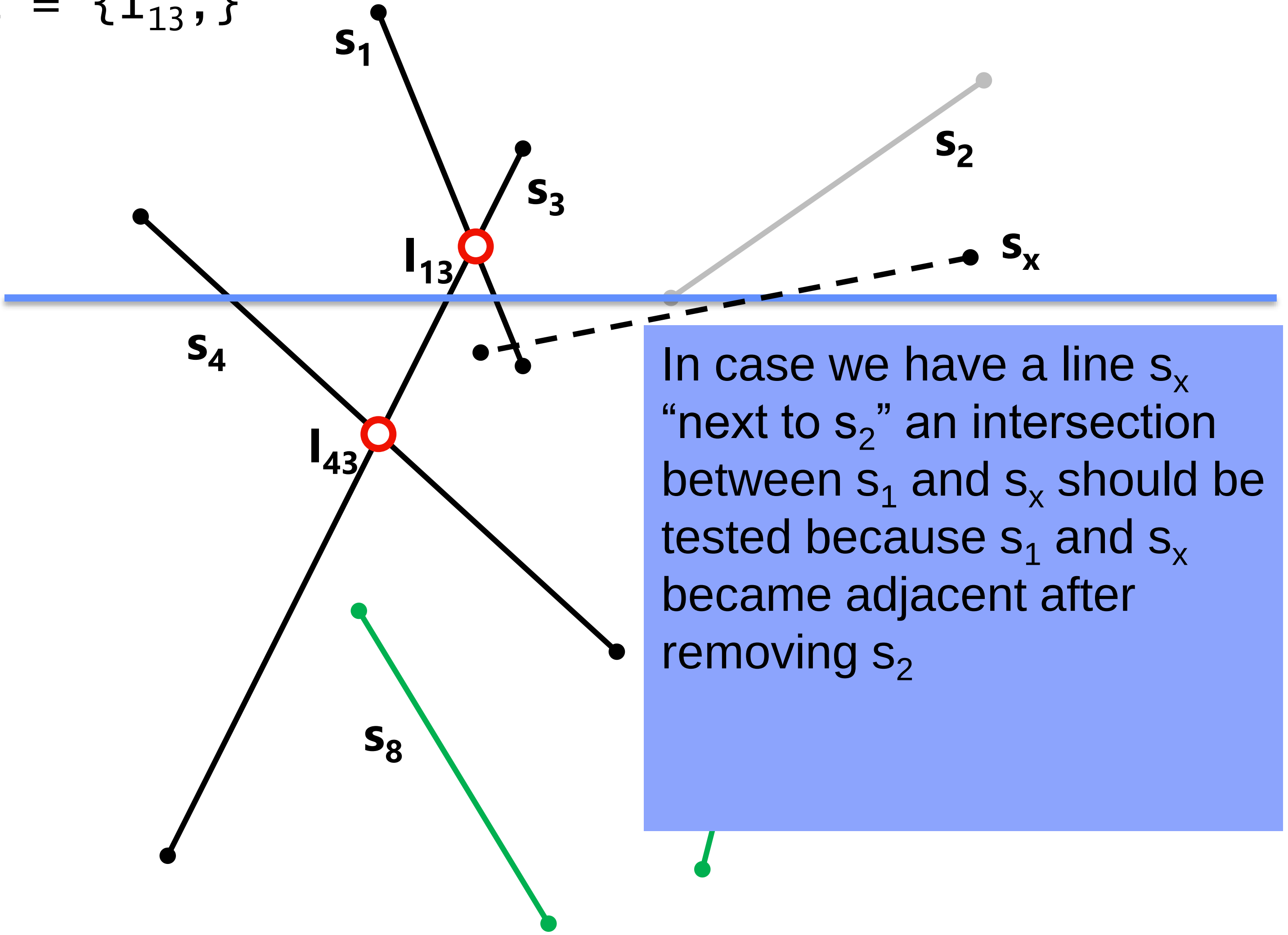
Operations on status structure T	del **s**$_2$
Operations on event structure Q	del **s**$_2$ end

Zero insertions; **s**$_2$ is now “dead”

$T = \{s_4, s_3, s_1\}$

$Q = \{$~~$\uparrow s_1, \uparrow s_2, \uparrow s_3, \uparrow s_4, I_{13}, \downarrow s_2,$~~$\downarrow s_1, \uparrow s_5, I_{43}, \uparrow s_6, \ldots\}$

$L = \{I_{13},\}$



In case we have a line s_x "next to s_2" an intersection between s_1 and s_x should be tested because s_1 and s_x became adjacent after removing s_2

Operations on status structure T
del **s_2**
Operations on event structure Q
del **s_2** end

Zero insertions; **s_2** is now "dead"

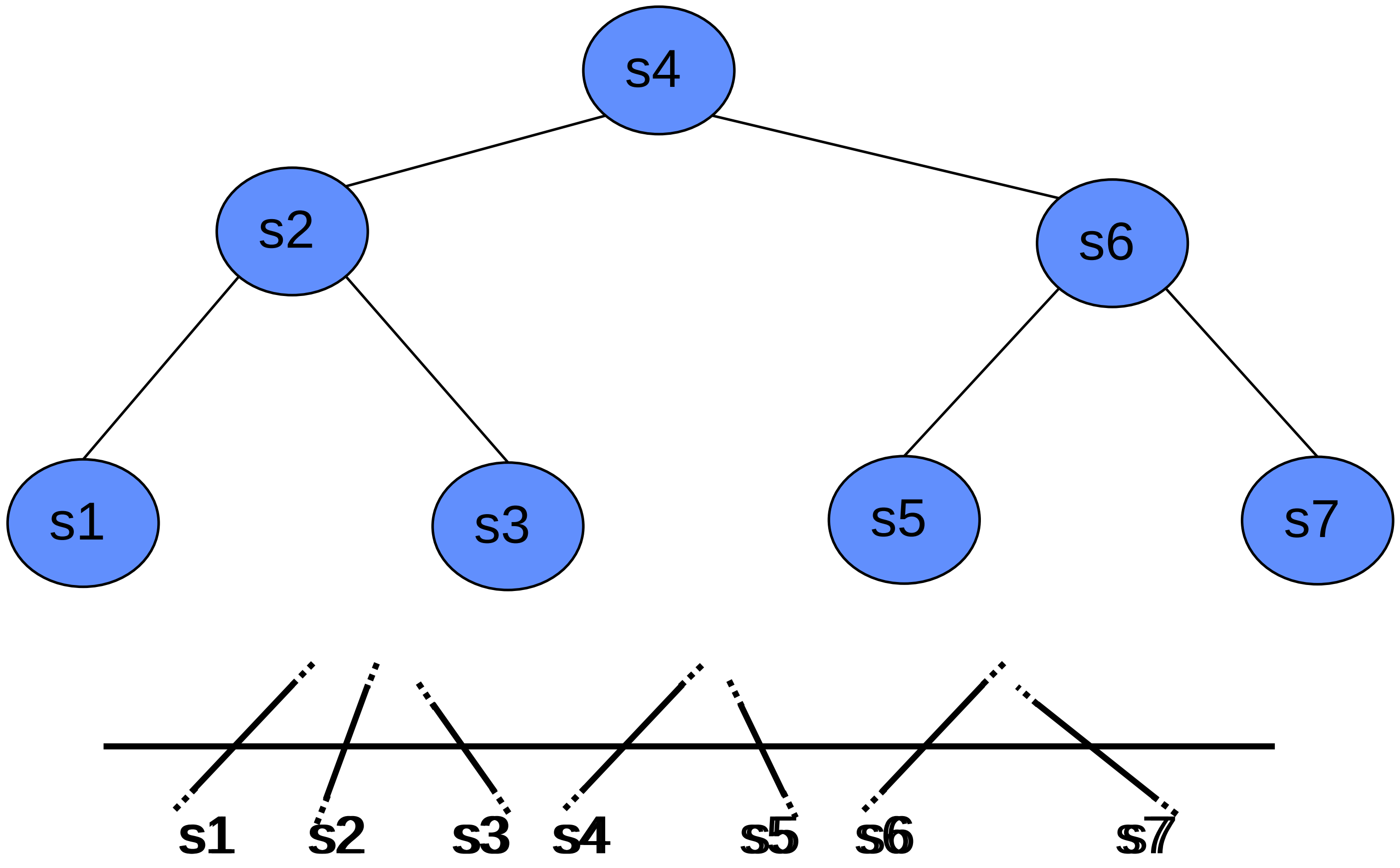
Events

- Events are generated when the sweep line encounters
  - an upper endpoint of a line segment
  - a lower endpoint of a line segment
  - an intersection point of line segments
- Events trigger changes of the status of the sweep line and generate output

Degenerate Cases

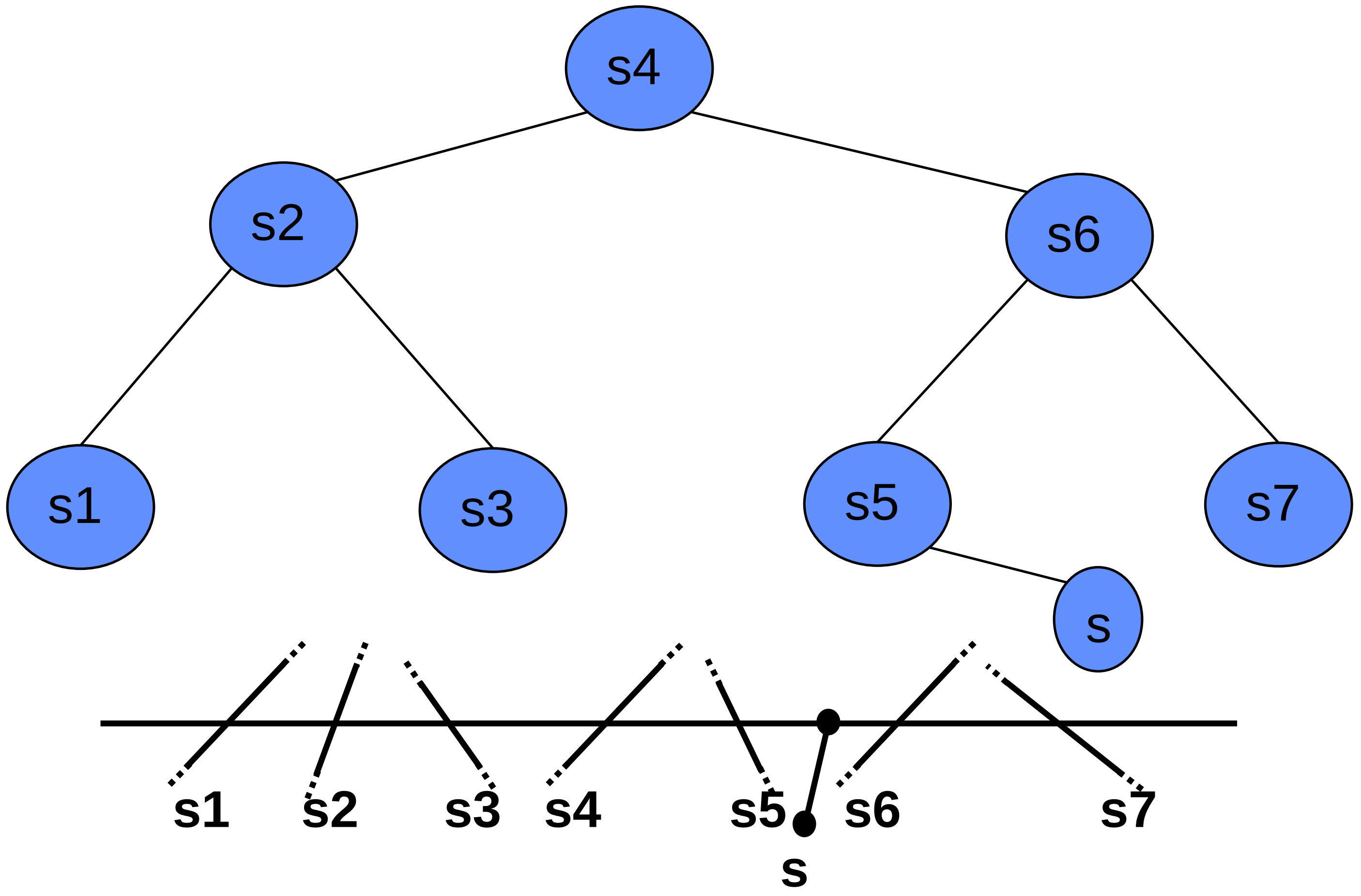
- They are ignored during the design phase of the algorithm. Simplifying assumptions:
  - no horizontal segments
  - no intersection at endpoints
  - no three (or more) segments with a common intersection
  - endpoints and intersection points have different y-coordinates
  - no two overlapping segments

(T)Status Structure Updates



- Implemented as a balanced binary search tree, e.g. red-black tree or AVL
- Storing O(n), insertion O(log n), deletion O(log n)

(T)Status Structure Updates

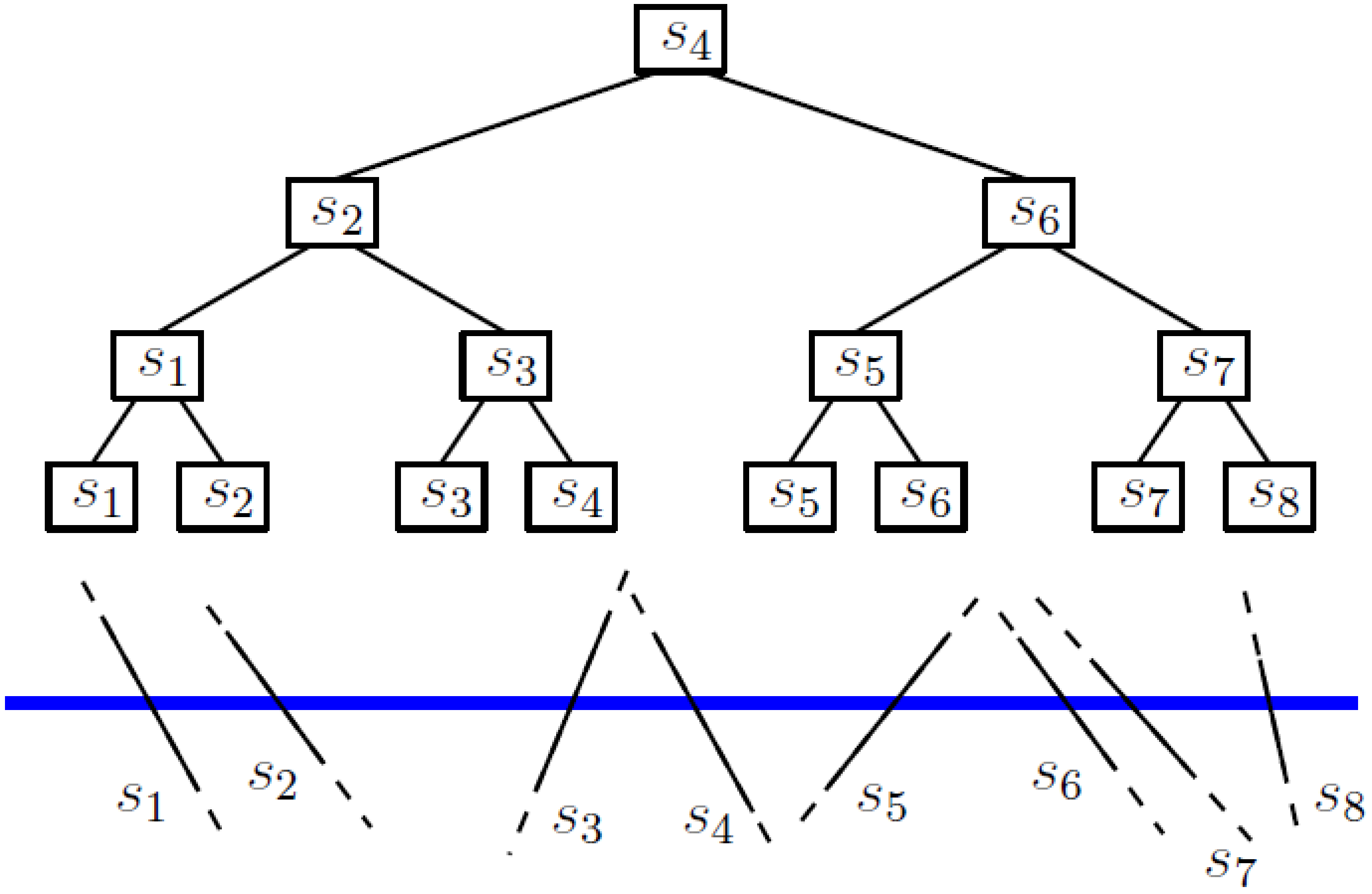


- Inserting s

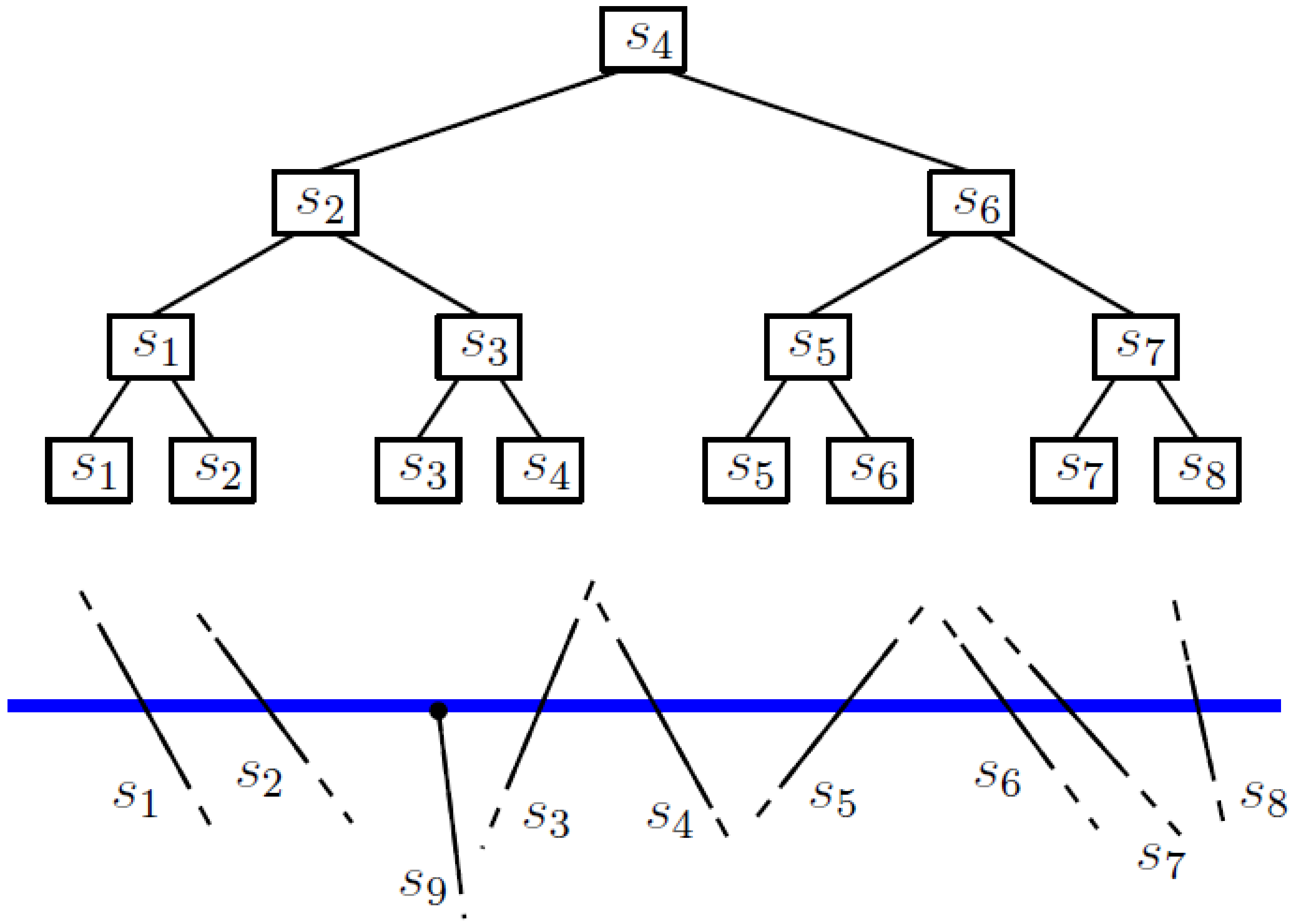
(T)Status Structure Updates

- Another way to implement the balanced search tree (BST) is storing the segments in the leaves (n leaves), but selecting inner nodes with this criterion: the segment from the rightmost leaf in its left subtree

(T)Status Structure Updates

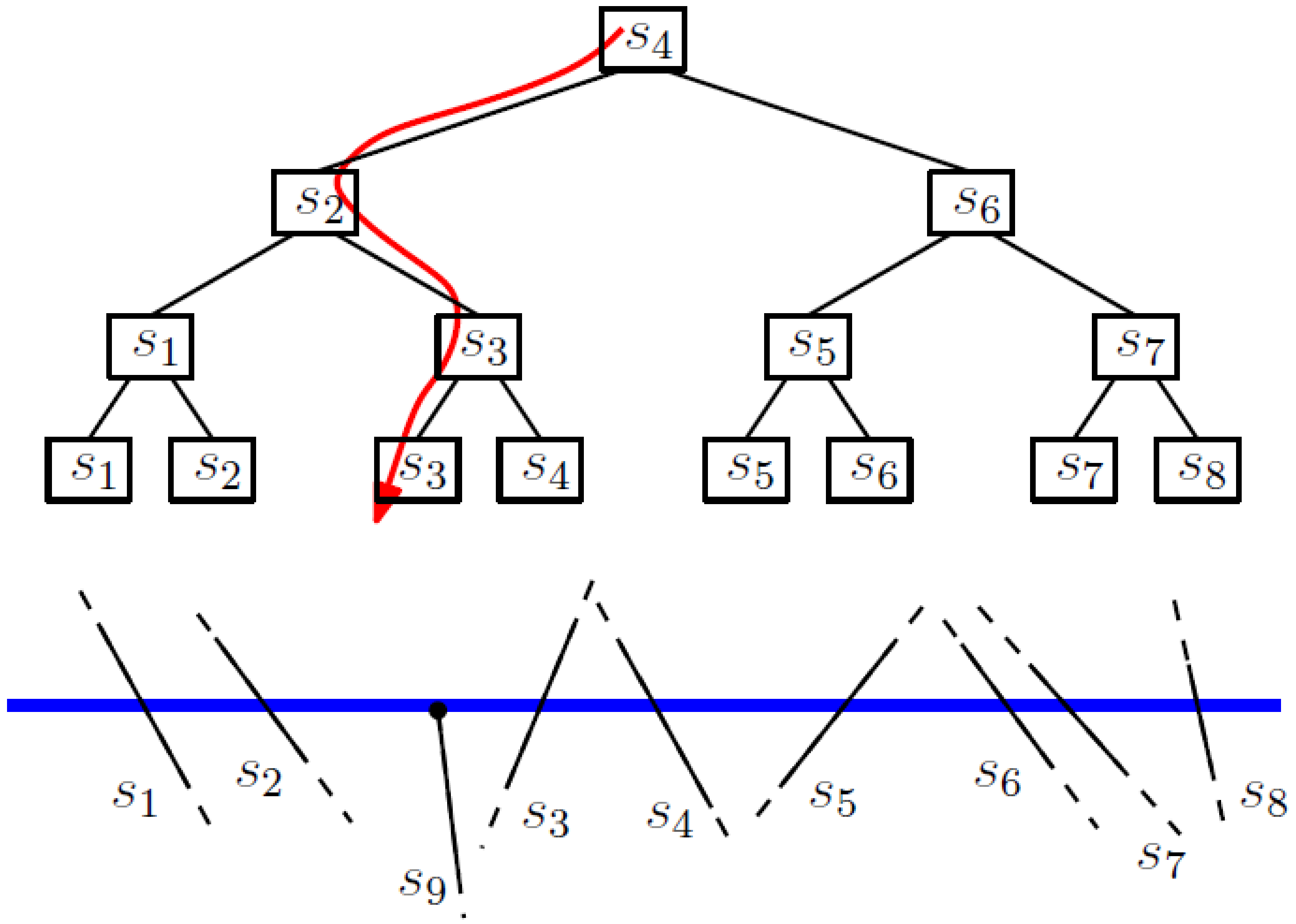


(T)Status Structure Updates



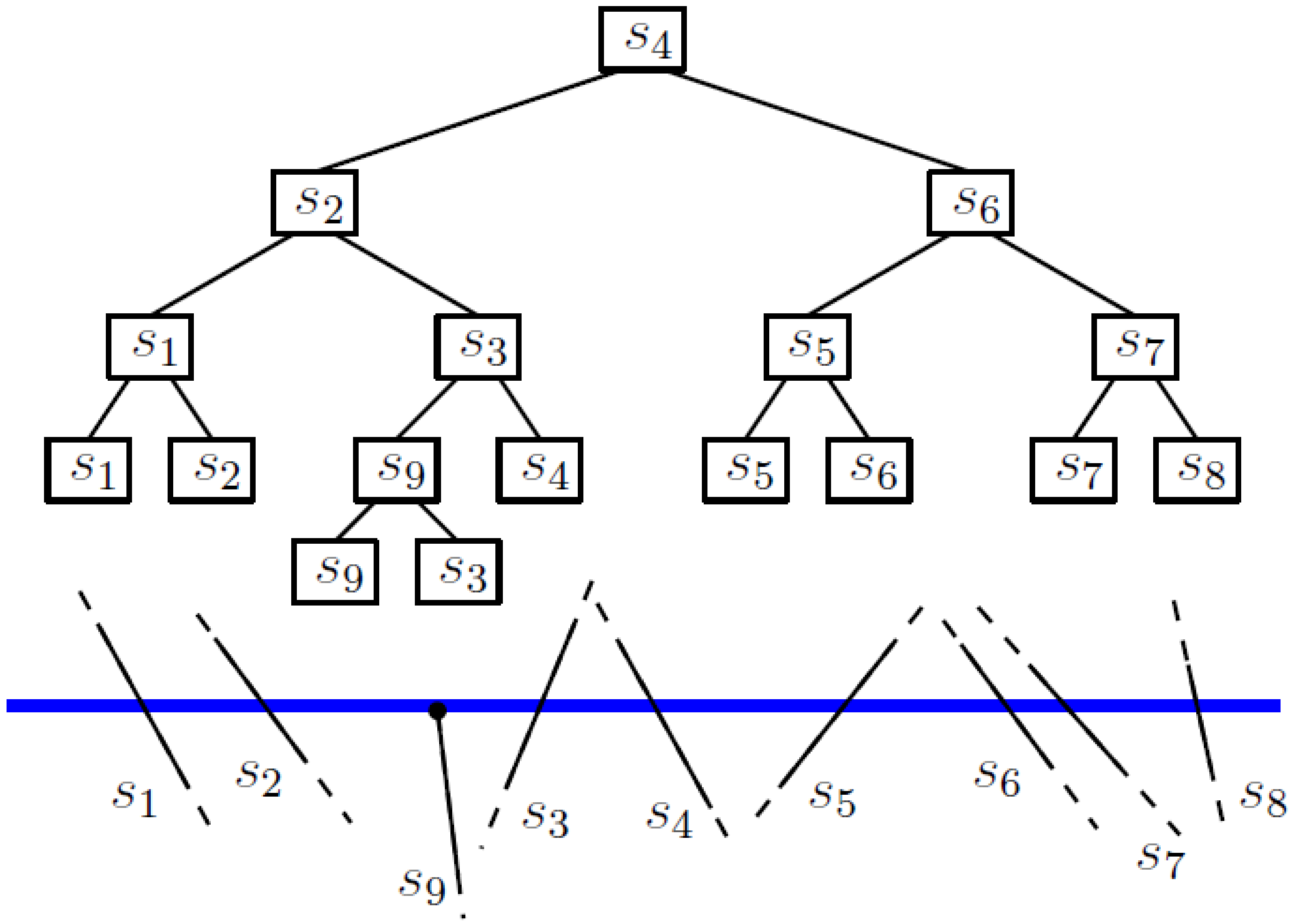
- Upper end of line segment s_9 is found

(T)Status Structure Updates



- Search appropriate location …

(T)Status Structure Updates



- Search appropriate location and insert s_9
- …

(T)Status Structure Updates

- Upper end of line segment is found
  - Insert the corresponding segment in the tree
  - O(log n)

- Lower end of line segment is found
  - Delete corresponding segment from status structure (a balanced tree)
  - O(log n)

- Intersection point is found
  - Report intersection point
  - Using the first BST implementation ➔ swap the corresponding two nodes in the tree.
  - Using the second BST implementation ➔swap the corresponding two leaves in the tree (and update the search path)
  - …

(Q)Event Structure

- Q is implemented as a **balanced search tree** too, with order $\mathbf{p} < \mathbf{q} \Leftrightarrow p_y > q_y \vee (p_y = q_y \wedge p_x < q_x)$
- It means, events are sorted by y axis (in ascendant order), but If two event points have the same y-coordinate, then the one with smaller x-coordinate will be first considered
- We **do not use a heap** to implement the event queue, because we have to be able to test whether a given event is already present in Q
- Two event points can coincide. For example, the upper endpoints of two distinct segments may coincide. It is convenient to treat this as one event point. Hence, an insertion must be able to check whether an event is already present in Q. This case will be handled later

(Q)Event Structure

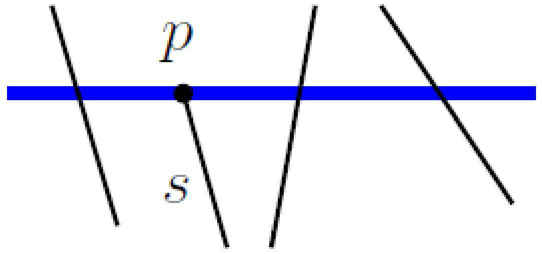
- Operations
  - Initialization (sort a sequence of upper and lower endpoints of segments in decreasing y-order) takes O(*n* log *n*) time
  - Find and remove the head of queue takes O(log *n*) time
  - Intersection points need to be potentially inserted when segments become adjacent in T. Due to Q is a balanced tree, insertion of an intersection point takes O(log *n*) time
- Space (for now): O(*n* + *k*), *k* = #intersections
- How large could k be?

Algorithm

```
1 Initialize list of intersection L=0
2 Insert upper and lower endpoints of all segments
  into event structure Q
3 initialize status structure T=0
4 while (Q is not empty) do
5     p = Q.head(); Q.del_head();
6     Update Q, L, T on each case:
7             p is an upper endpoint
8             p is an intersection point
9             p is a lower endpoint
```

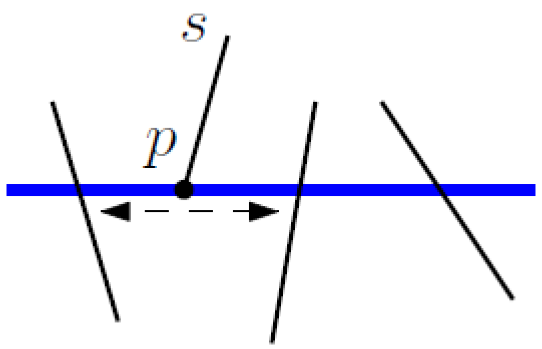
Event Handling: Upper Endpoint

- If the event is an upper endpoint event, and **s** is the line segment that starts at **p**:
  - Search with **p** in T, and insert **s**
  - If **s** intersects its left neighbor in T, then determine the intersection point and insert in Q
  - If **s** intersects its right neighbor in T, then determine the intersection point and insert in Q



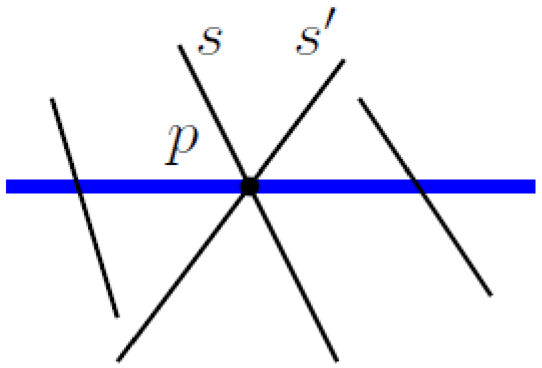
Event Handling: Lower Endpoint

- If the event is a lower endpoint event, and **s** is the line segment that ends at **p**:
  - Search with **p** in T, and delete **s**
  - Let $\mathbf{s_l}$ and $\mathbf{s_r}$ be the left and right neighbors of **s** in T (before deletion). They have now become neighbors and if they intersect ***below*** the sweep line, then insert their intersection point as an event in Q



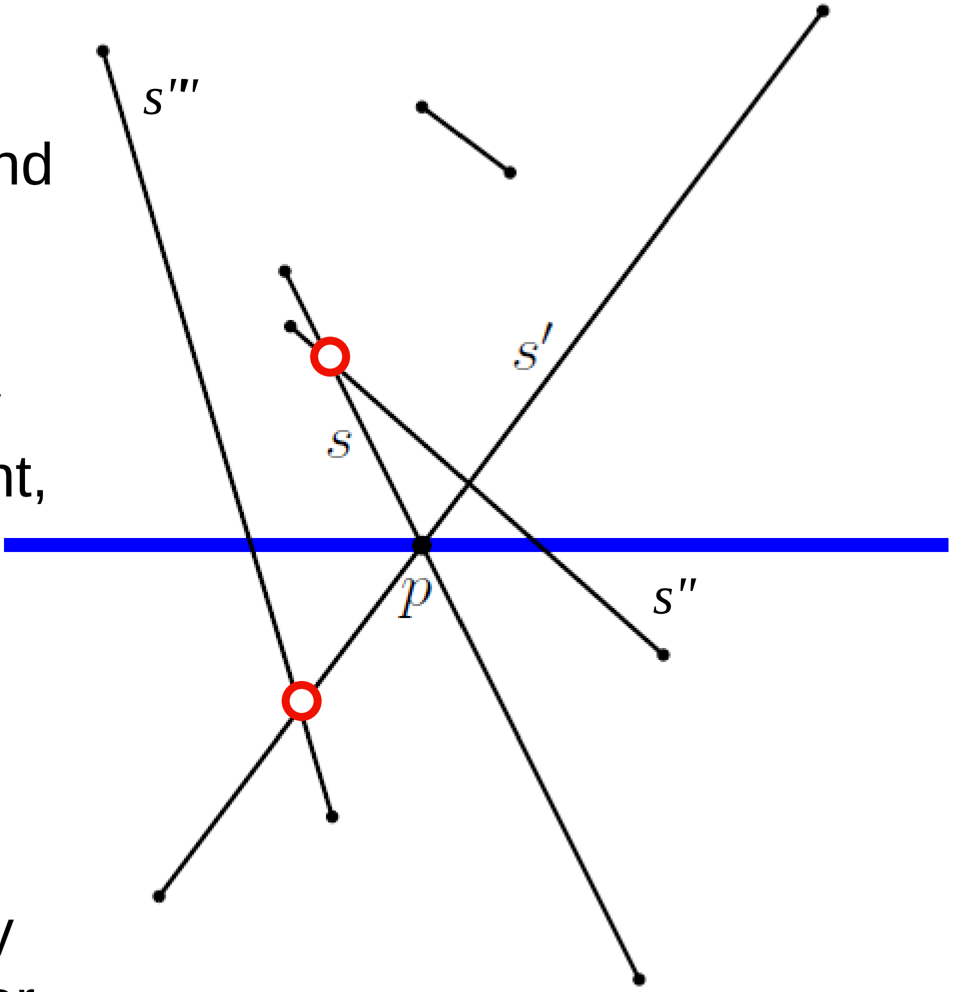
Event Handling: Intersection Point

- If the event is an intersection point event wher **s** and **s'** intersect at **p**:
  - Exchange **s** and **s'** in T
  - If **s'** and its new left neighbor in T intersect below the sweep line, then insert this intersection point in Q
  - If **s** and its new right neighbor in T intersect ***below*** tl sweep line, then insert this intersection point in Q
  - Report the intersection point



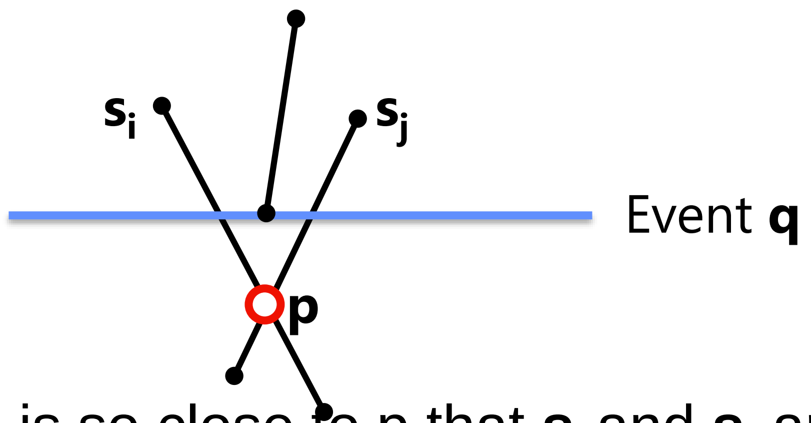
Event Handling: Surprises?

- Can it be that new horizontal neighbors already intersected above the sweep line? See ***s*** and ***s***''
- Can it be that we insert a newly detected intersection point event, but it already occurs in Q? See ***s***''' and ***s***'
- Notice that we need to check if an intersection point already exists in Q. That's why a priority queue (heap) is not an option for implementing Q



Correctness: Neighborhood Lemma

- Lemma: Let $\mathbf{s_i}$ and $\mathbf{s_j}$ be two non-horizontal segments intersecting in a single point **p** and no third segment passing through **p**. Then there is an event point above **p** where $\mathbf{s_i}$ and $\mathbf{s_j}$ become adjacent and are tested for intersection



- Proof: The sweep line is so close to p that $\mathbf{s_i}$ and $\mathbf{s_j}$ are next to each other. Since $\mathbf{s_i}$ and $\mathbf{s_j}$ are not yet adjacent at the beginning of the algorithm there is an event **q** where $\mathbf{s_i}$ and $\mathbf{s_j}$ become adjacent and tested for intersection

Number of Operation / Time Complexity

- Handling an event requires
  - at most one search in T and/or one insertion, deletion, or swap
  - at most twice finding a neighbor in T
  - at most one deletion from and two insertions in Q
- Since T and Q are balanced binary search trees, handling an event takes only O(log *n*) time
- How many events?
  - 2*n* for the upper and lower endpoints
  - *k* for the intersection points, if there are *k* of them
- In total: O(*n*+*k*) events

Number of Operation / Time Complexity

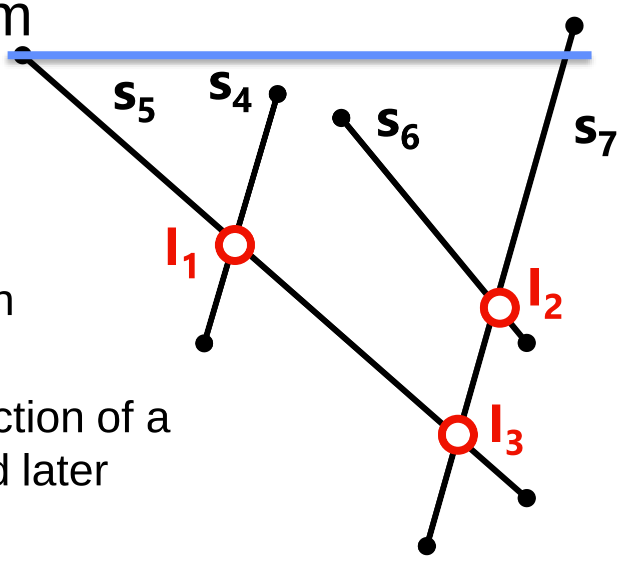
- Initialization takes $O(n \log n)$ time (to insert all upper and lower endpoint events in Q)
- Each of the $O(n+k)$ events takes $O(\log n)$ time
- The algorithm takes $O(n \log n + k \log n)$ time

- The total time required to carry out the plane-sweep algorithm for computing all k intersections of a set of n line segments is $O((n+k) \log n)$.
- The plane-sweep algorithm is **output sensitive**!

- Note: if k is really large, the brute force $O(n^2)$ time algorithm is more efficient

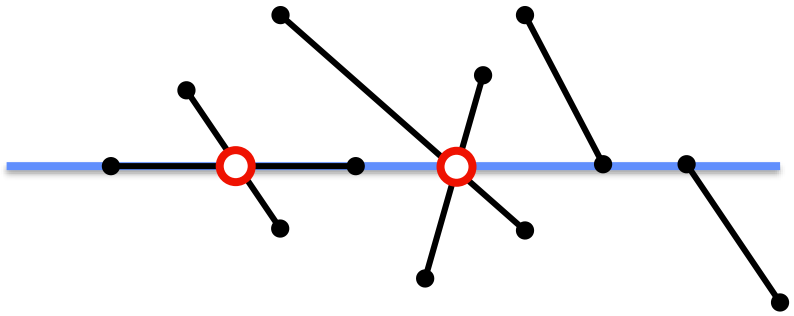
Space Complexity

- The status structure T requires O(*n*) space
- The event structure Q needs in a naïve implementation of the plane-sweep algorithm O(*n*+*k*) space, and *k* could be large
  - How to improve the situation?
  - Idea 1
    - For each line segment store the next intersection in decreasing y-order with the segment
    - Keep only intersections in Q if it is the next intersection of a segment. Further intersections will be rediscovered later
  - Idea 2
    - Just store intersections of adjacent segments and remove an intersection if segments are no longer adjacent. The intersection will be added again later, when they become adjacent
  - Thus, the space of Q is limited to O(*n*)



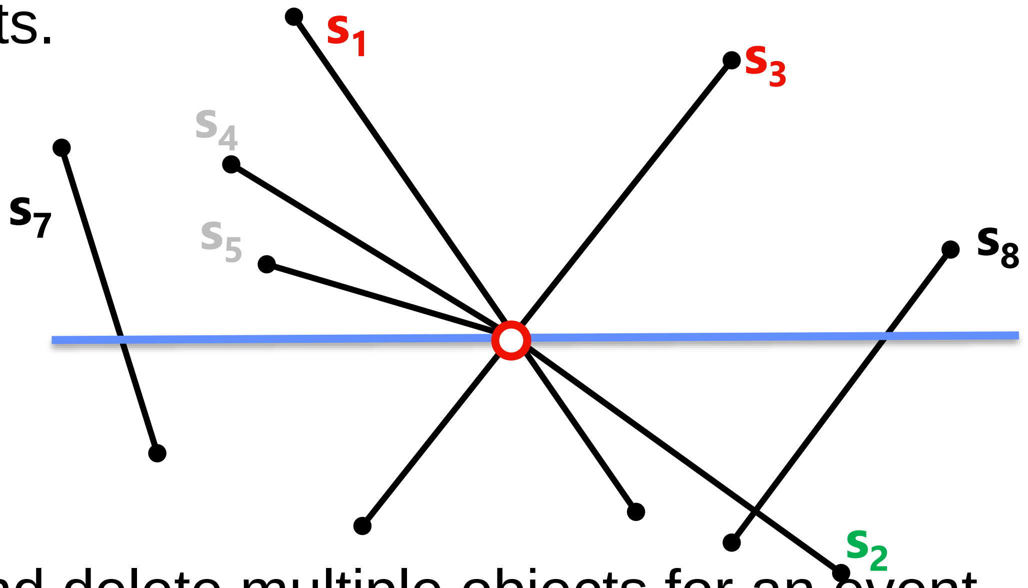
Degenerate Cases

- … were ignored during the design phase of the algorithm.
- Horizontal segments and multiple events with the same y-coordinate
  - Two different events with the same y-coordinate are treated from left to right (i.e. the upper endpoint of a horizontal line segment is its left endpoint)



Degenerate Cases

- More than 2 line segments meet. One event for multiple segments.



- Insert and delete multiple objects for an event
  - Delete $\mathbf{s_4}$, $\mathbf{s_5}$; insert $\mathbf{s_2}$; swap $\mathbf{s_1}$,$\mathbf{s_3}$
  - Find neighbor of leftmost line segment $\mathbf{s_3}$ and check for intersection of $\mathbf{s_3}$ and $\mathbf{s_7}$
  - Find neighbor of rightmost line segment $\mathbf{s_2}$ and check for intersection of $\mathbf{s_2}$ and $\mathbf{s_8}$

Plane-Sweep (or Sweep-Line) Technique

- Widely applicable paradigm to design geometric algorithms
- Plane sweep is similar to an incremental construction
  - Processing geometric objects is started as the objects are encountered by the sweep line and a partial solution is maintained
  - Partial solution is updated on each event with a few operations
- Plane sweep uses knowledge on the relative position of objects to perform an efficient computation
- In $\mathbf{R}^2$, the plane is (conceptually) swept by a straight line. Hence, the two different names for the technique
- In $\mathbf{R}^3$, the space is (conceptually) swept by a plane
  - Space-Sweep (or Sweep-Plane) technique

Design of Sweep Algorithms

- Define what constitutes the status
- Choose the status structure and the event list
- Figure out how events must be handled (use sketches!)
- Perform analysis: determine the number of events and how much time they take
- Deal with degenerate cases

A little bit of the past …

- What about if you want to fill a list of 2D polygon? May we take some ideas from this lecture?. For example, keeping a list of active triangles?
- In the past, there was not enough memory to store the full z-buffer and color-buffer. So, we filled only one line of pixels of the output window on each step considering the triangles intersecting that line (incremental algorithm). So, we use O(n) pixels of memory, instead of O(n^2). A line is simply sweeping from the top to the bottom screen line generating the whole output image

End

- My Bauhaus-Universität Weimar e-mail is rhadames.elias.carmona.suju@uni-weimar.de

- You may also write me to recs34@gmail.com