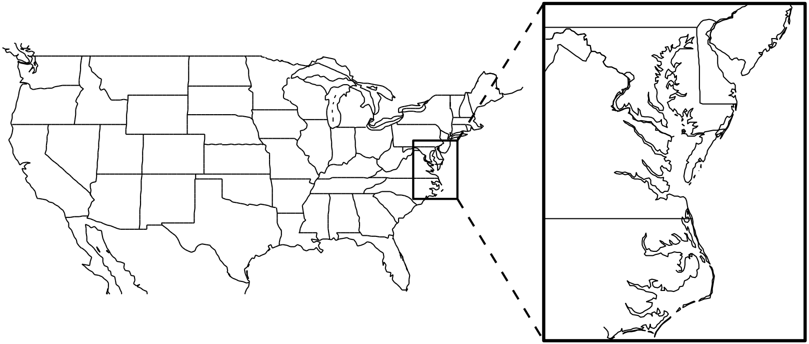
Windowing queries

Computational Geometry

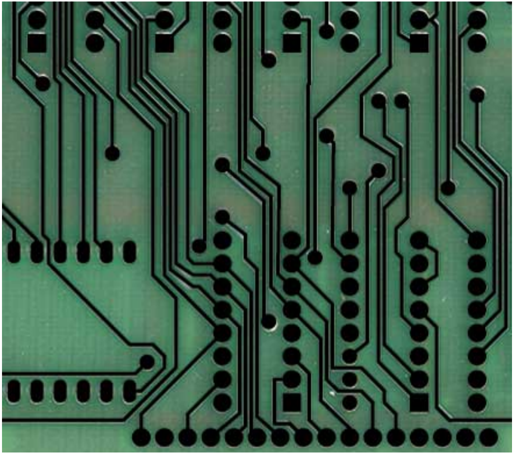
Lecture 15: Windowing queries

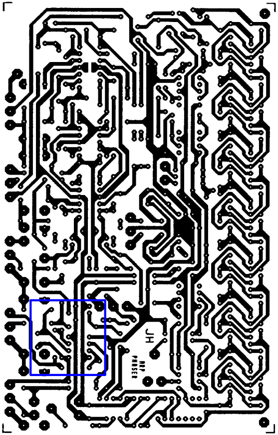
Windowing



Zoom in; re-center and zoom in; select by outlining

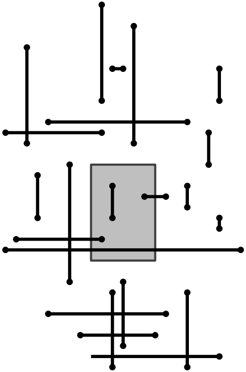
Windowing





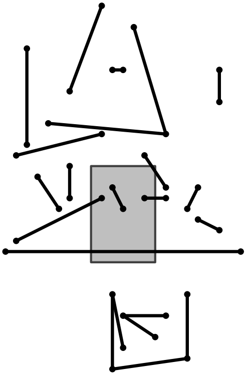
Windowing

Given a set of n axis-parallel line segments, preprocess them into a data structure so that the ones that intersect a query rectangle can be reported efficiently



Windowing

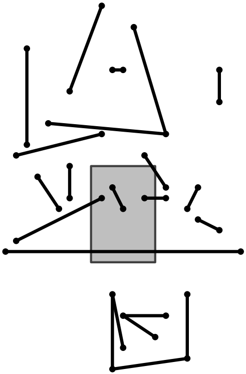
Given a set of n arbitrary, non-crossing line segments, preprocess them into a data structure so that the ones that intersect a query rectangle can be reported efficiently



Windowing

Two cases of intersection:

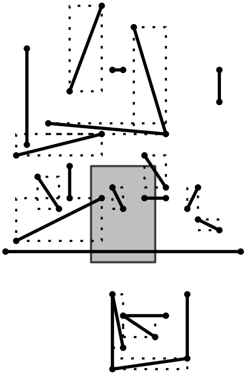
- An endpoint lies inside the query window; solve with range trees
- The segment intersects a side of the query window; solve how?



Using a bounding box?

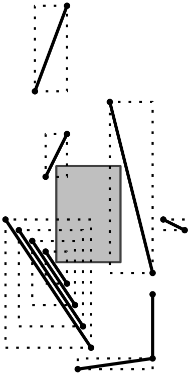
If the query window intersects the line segment, then it also intersects the bounding box of the line segment (whose sides are axis-parallel segments)

So we could search in the $4n$ bounding box sides



Using a bounding box?

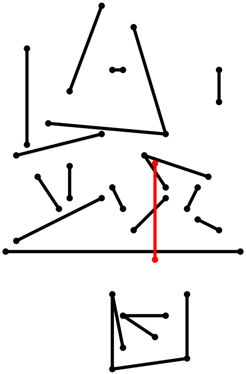
But: if the query window intersects bounding box sides does not imply that it intersects the corresponding segments



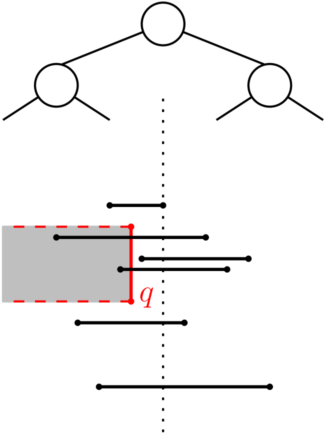
Windowing

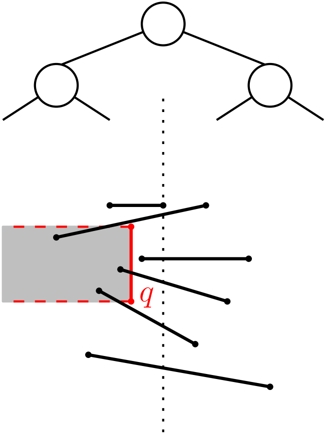
Current problem of our interest:

Given a set of arbitrarily oriented, non-crossing line segments, preprocess them into a data structure so that the ones intersecting a vertical (horizontal) query segment can be reported efficiently



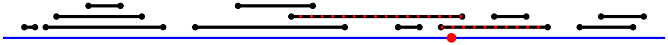
Using an interval tree?





Interval querying

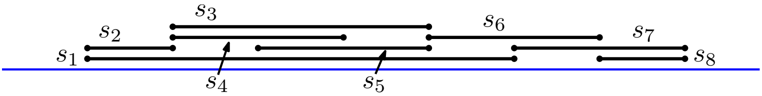
Given a set I of n intervals on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



We have the interval tree, but we will develop an alternative solution

Interval querying

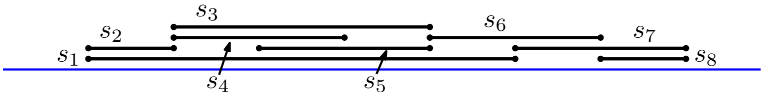
Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of n *segments* on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



The new structure is called the *segment tree*

Locus approach

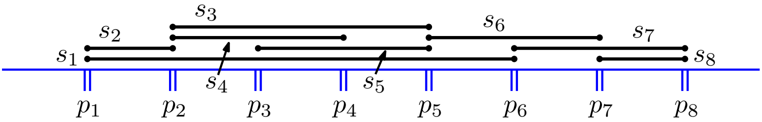
The locus approach is the idea to partition the solution space into parts with equal answer sets



For the set S of segments, we get different answer sets before and after every endpoint

Locus approach

Let $p_1, p_2, \ldots, p_m$ be the sorted set of unique endpoints of the intervals; $m \leq 2n$



The real line is partitioned into

$(-\infty, p_1)$, $[p_1, p_1]$, (p_1, p_2), $[p_2, p_2]$, (p_2, p_3), $\ldots$, $(p_m, +\infty)$,

these are called the elementary intervals

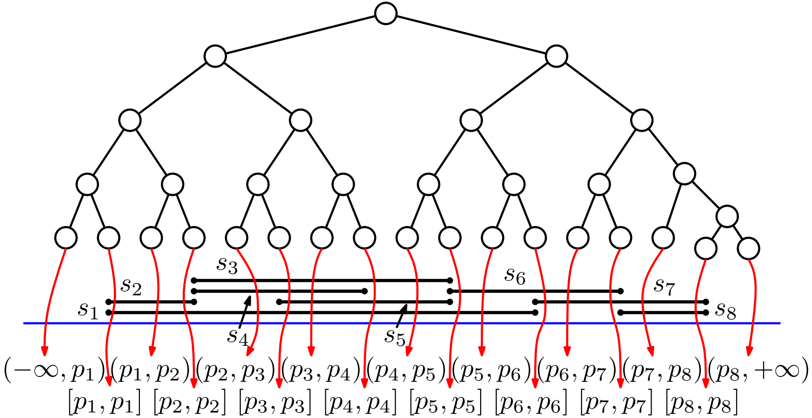
Locus approach

We could make a binary search tree that has a leaf for every elementary interval
$(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], (p_2, p_3), \ldots, (p_m, +\infty)$

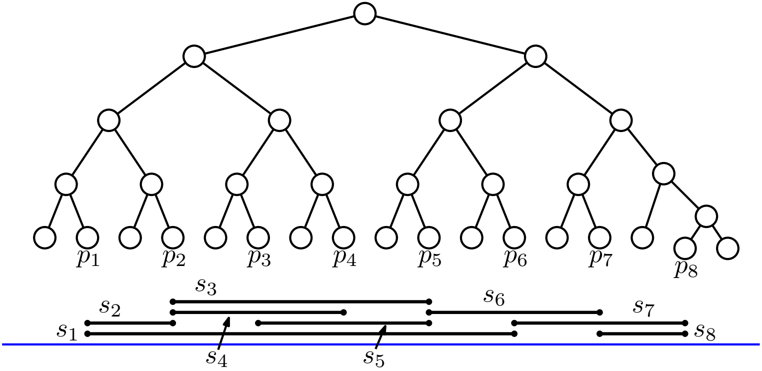
Each segment from the set S can be stored with all leaves whose elementary interval it contains: $[p_i, p_j]$ is stored with
$[p_i, p_i], (p_i, p_{i+1}), \ldots, [p_j, p_j]$

A *stabbing query* with point q is then solved by finding the unique leaf that contains q, and reporting all segments that it stores

Locus approach



Locus approach



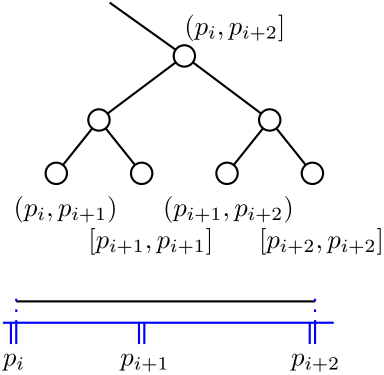
Locus approach

Question: What are the storage requirements and what is the query time of this solution?

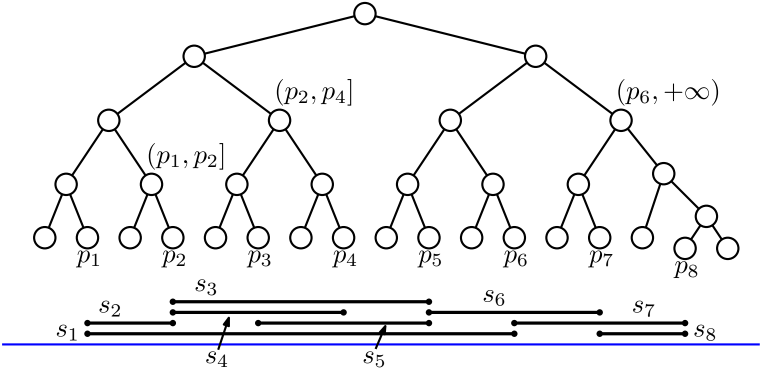
Towards segment trees

In the tree, the leaves store elementary intervals

But each internal node corresponds to an interval too: the interval that is the union of the elementary intervals of all leaves below it



Towards segment trees



Towards segment trees

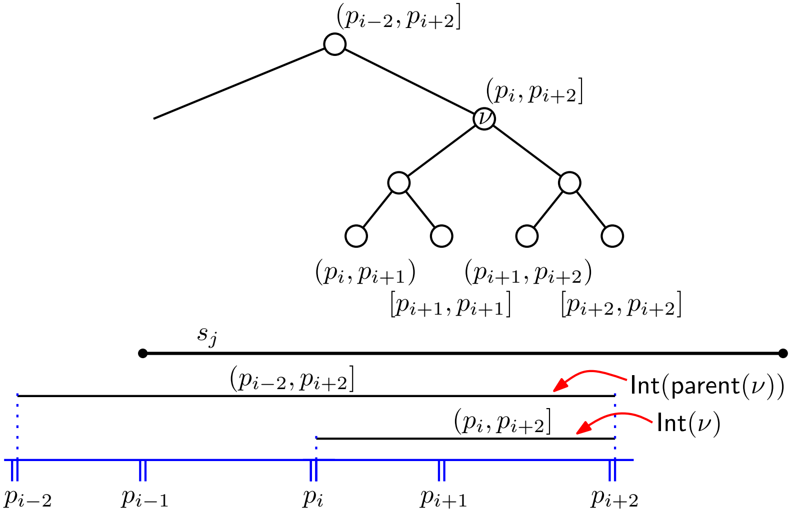
Let Int(v) denote the interval of node v

To avoid quadratic storage, we store any segment s_j as high as possible in the tree whose leaves correspond to elementary intervals

More precisely: s_j is stored with v if and only if

$\text{Int}(v) \subseteq s_j$ but $\text{Int}(\text{parent}(v)) \not\subseteq s_j$

Towards segment trees



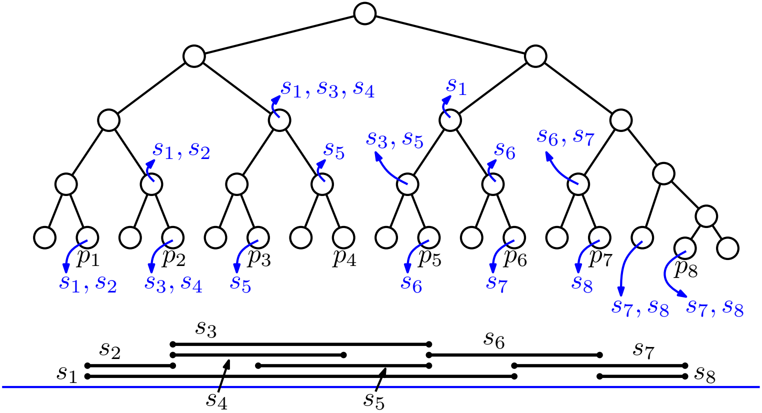
Segment trees

A segment tree on a set S of segments is a balanced binary search tree on the elementary intervals defined by S, and each node stores its interval, and its *canonical subset* of S in a list (unsorted)

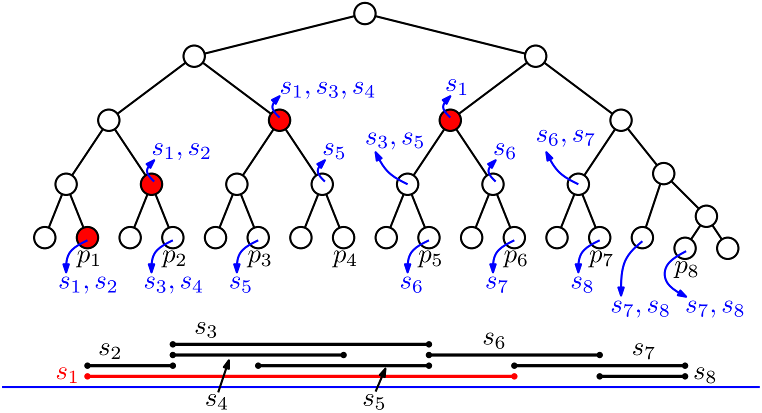
The canonical subset (of S) of a node ν is the subset of segments s_j for which

$\mathrm{Int}(\nu) \subseteq s_j$ but $\mathrm{Int}(\mathrm{parent}(\nu) \not\subseteq s_j$

Segment trees



Segment trees



Segment trees

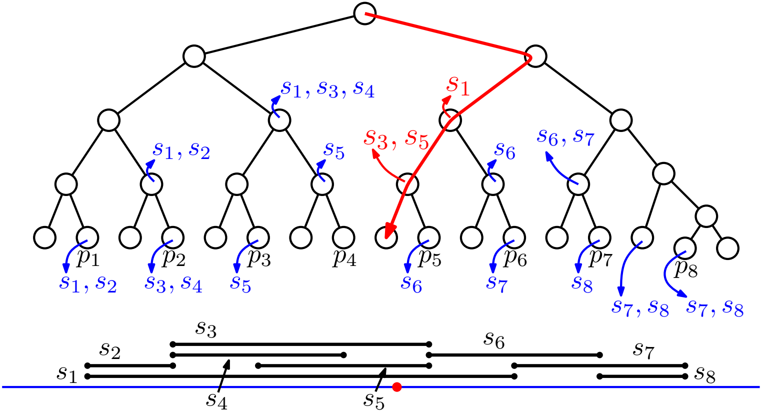
Question: Why are no segments stored with nodes on the leftmost and rightmost paths of the segment tree?

Query algorithm

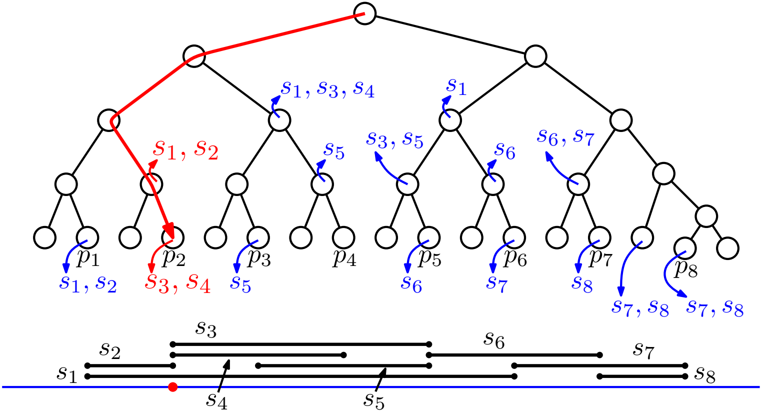
The query algorithm is trivial:

For a query point q, follow the path down the tree to the elementary interval that contains q, and report all segments stored in the lists with the nodes on that path

Example query



Example query



Query time

The query time is $O(\log n + k)$, where k is the number of segments reported

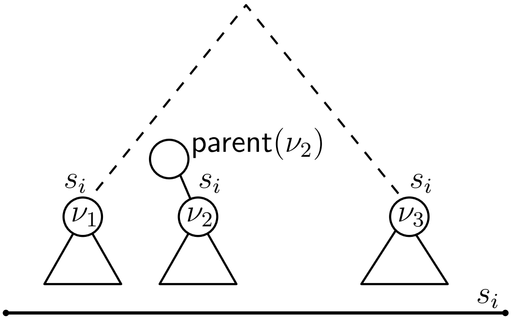
Segments stored at many nodes

A segment can be stored in several lists of nodes. How bad can the storage requirements get?

Segments stored at many nodes

Lemma: Any segment can be stored at up to two nodes of the same depth

Proof: Suppose a segment s_i is stored at *three* nodes v_1, v_2, and v_3 at the *same depth* from the root



Segments stored at many nodes

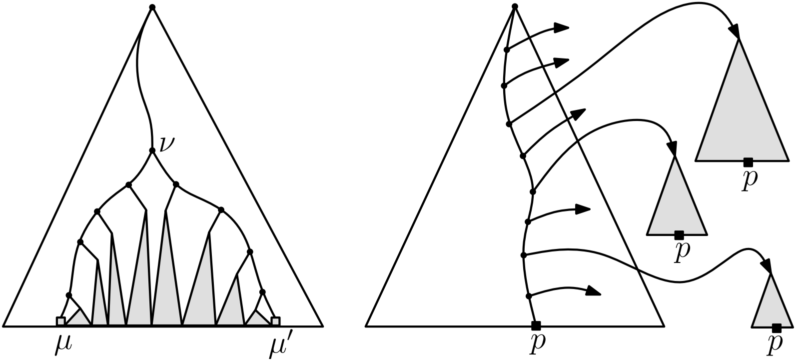
If a segment tree has depth $O(\log n)$, then any segment is stored in at most $O(\log n)$ lists $\Rightarrow$ the total size of all lists is $O(n \log n)$

The main tree uses $O(n)$ storage

The storage requirements of a segment tree on n segments is $O(n \log n)$

Segments and range queries

Note the correspondence with 2-dimensional range trees



Result

Theorem: A segment tree storing n segments ($=$intervals) on the real line uses $O(n \log n)$ storage, can be built in $O(n \log n)$ time, and stabbing queries can be answered in $O(\log n + k)$ time, where k is the number of segments reported

Property: For any query, all segments containing the query point are stored in the lists of $O(\log n)$ nodes

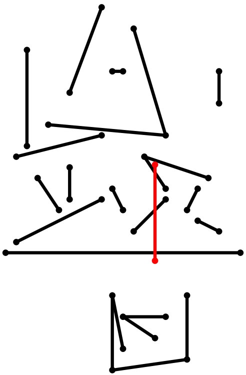
Stabbing counting queries

Question: Do you see how to adapt the segment tree so that stabbing *counting* queries can be answered efficiently?

Back to windowing

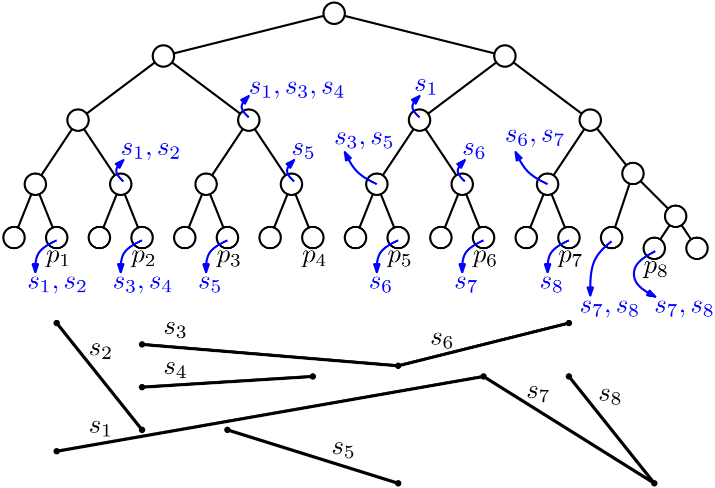
Problem arising from windowing:

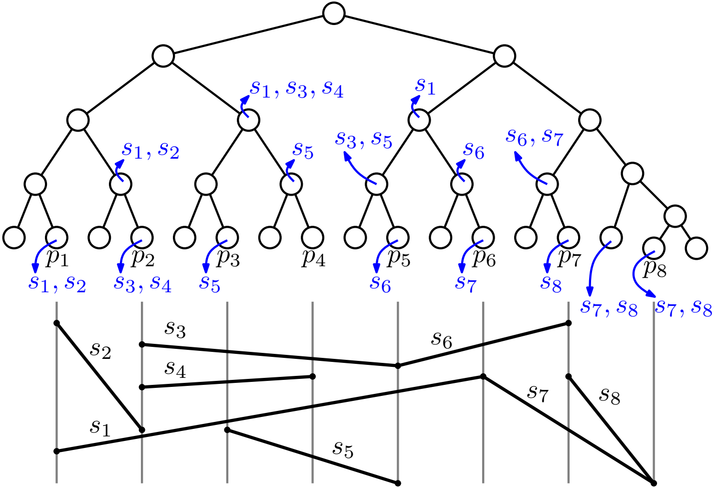
Given a set of arbitrarily oriented, non-crossing line segments, preprocess them into a data structure so that the ones intersecting a vertical (horizontal) query segment can be reported efficiently



Idea for solution

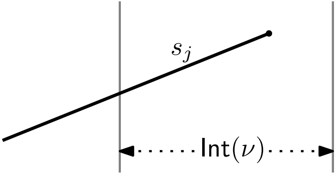
The main idea is to build a segment tree on the x-projections of the 2D segments, and replace the associated lists with a more suitable data structure


s_1, s_3, s_4
s_1
s_1, s_2
s_5
s_3, s_5
s_6
s_6, s_7
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
s_1, s_2
s_3, s_4
s_5
s_6
s_7
s_8
s_7, s_8
s_7, s_8
s_3
s_6
s_2
s_4
s_7
s_8
s_1
s_5

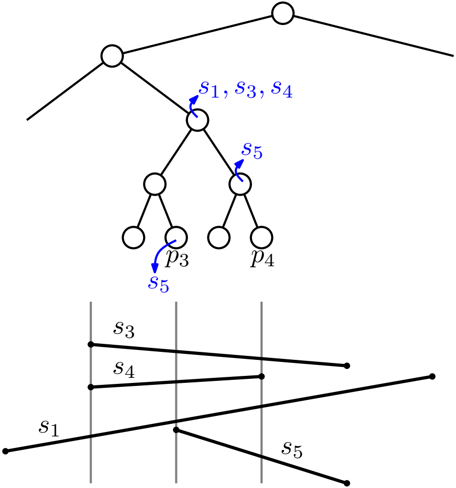

s_1, s_3, s_4
s_1
s_1, s_2
s_5
s_3, s_5
s_6
s_6, s_7
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
s_1, s_2
s_3, s_4
s_5
s_6
s_7
s_8
s_7, s_8
s_7, s_8
s_3
s_2
s_6
s_4
s_7
s_8
s_1
s_5

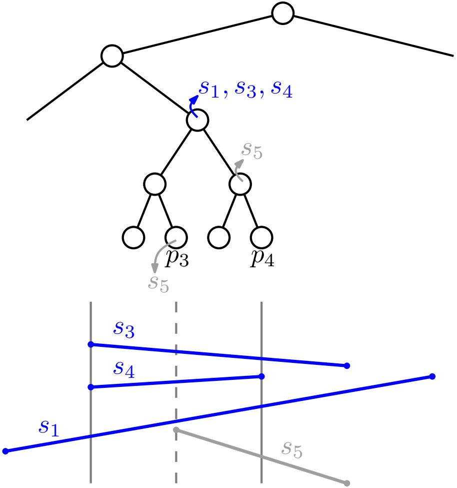
Observe that nodes now correspond to vertical slabs of the plane (with or without left and right bounding lines), and:

- if a segment s_i is stored with a node v, then it crosses the slab of v completely, but not the slab of the parent of v
- the segments crossing a slab have a well-defined top-to-bottom order



s_j is stored at one or more nodes below ν


s_1, s_3, s_4
s_5
p_3
p_4
s_5
s_3
s_4
s_1
s_5

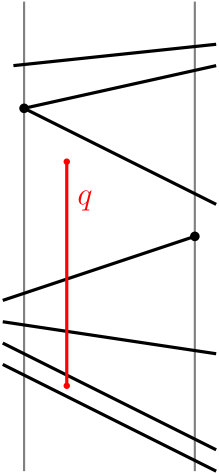

s_1, s_3, s_4
s_5
p_3
p_4
s_5
s_3
s_4
s_1
s_5

Querying

Recall that a query is done with a vertical line segment q

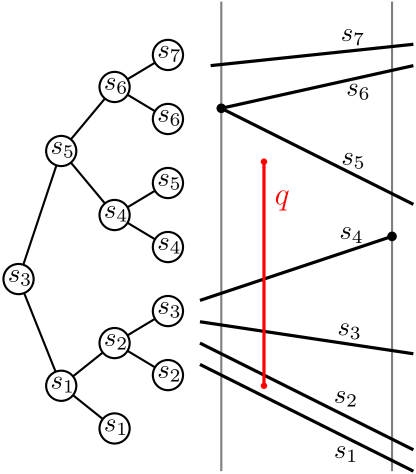
Only segments of S stored with nodes on the path down the tree using the x-coordinate of q can be answers

At any such node, the query problem is: which of the segments (that cross the slab completely) intersects the vertical query segment q?



Querying

We store the canonical subset of a node ν in a balanced binary search tree that follows the bottom-to-top order in its leaves



Data structure

A query with q follows one path down the main tree, using the x-coordinate of q

At each node, the associated tree is queried using the endpoints of q, as if it is a 1-dimensional range query

The query time is $O(\log^2 n + k)$

Data structure

The data structure for intersection queries with a vertical query segment in a set of non-crossing line segments is a segment tree where the associated structures are binary search trees on the bottom-to-top order of the segments in the corresponding slab

Since it is a segment tree with lists replaced by trees, the storage remains $O(n \log n)$

Result

Theorem: A set of n non-crossing line segments can be stored in a data structure of size $O(n \log n)$ so that intersection queries with a vertical query segment can be answered in $O(\log^2 n + k)$ time, where k is the number of answers reported

Theorem: A set of n non-crossing line segments can be stored in a data structure of size $O(n \log n)$ so that windowing queries can be answered in $O(\log^2 n + k)$ time, where k is the number of answers reported