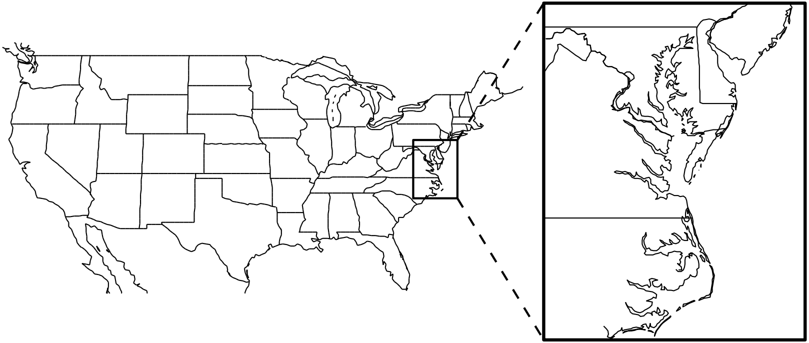
Windowing queries

Computational Geometry

Lecture 8: Windowing queries (part 2)

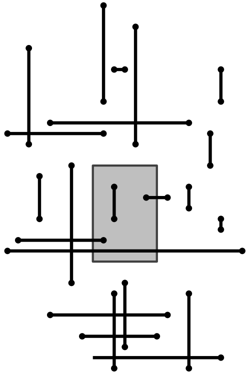
Windowing



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

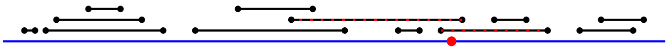
Windowing

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



Interval querying

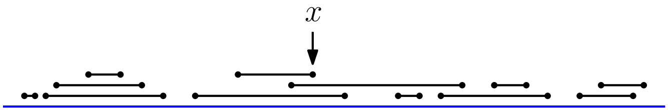
Given a set I of n intervals on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



Splitting a set of intervals

The *median* x of the $2n$ endpoints partitions the intervals into three subsets:

- Intervals I_{left} fully left of x
- Intervals I_{mid} that contain (intersect) x
- Intervals I_{right} fully right of x



Interval tree: recursive definition

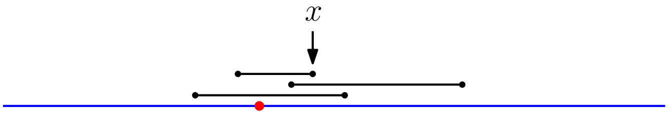
The interval tree for I has a root node v that contains x and

- the intervals I_{left} are stored in the left subtree of v
- the intervals I_{mid} are stored with v
- the intervals I_{right} are stored in the right subtree of v

The left and right subtrees are proper interval trees for I_{left} and I_{right}

Interval tree: left and right lists

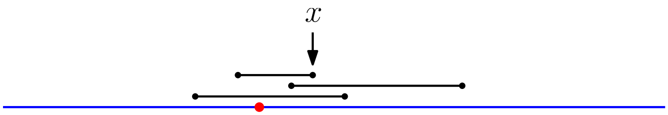
How is I_{mid} stored?



Observe: If the query point is left of x, then only the *left endpoint* determines if an interval is an answer

Symmetrically: If the query point is right of x, then only the *right endpoint* determines if an interval is an answer

Interval tree: left and right lists

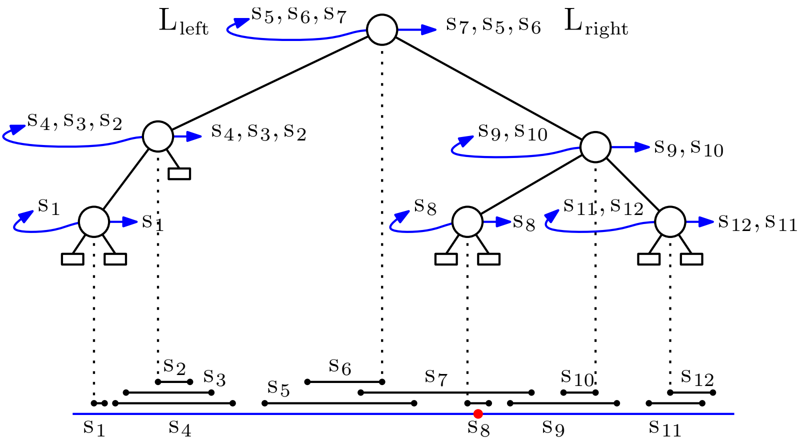


Make a list L_{left} using the left-to-right order of the *left endpoints* of I_{mid}

Make a list L_{right} using the right-to-left order of the *right endpoints* of I_{mid}

Store both lists as associated structures with ν

Interval tree: example



Interval tree: storage

The main tree has $O(n)$ nodes

The total length of all lists is $2n$ because each interval is stored exactly twice: in L_{left} and L_{right} and only at one node

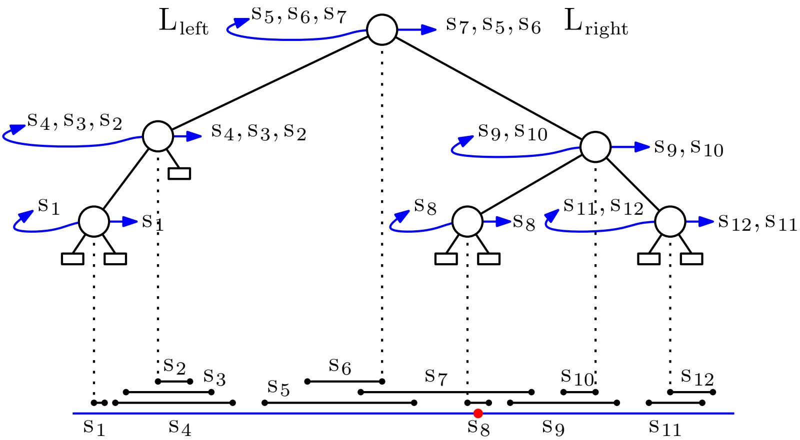
Consequently, the interval tree uses $O(n)$ storage

Interval querying

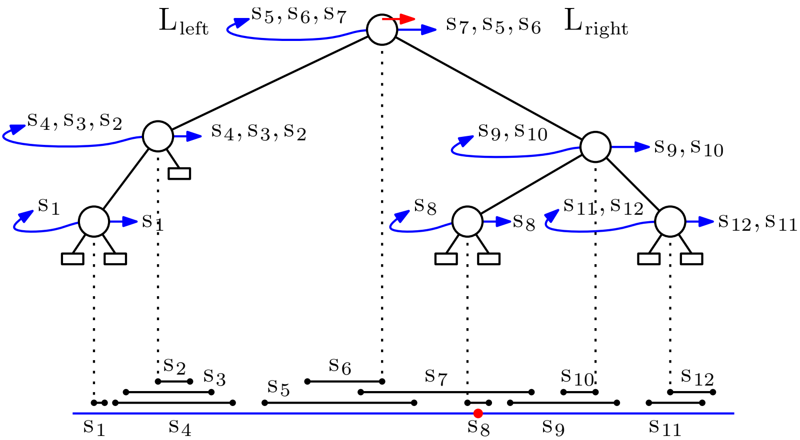
Algorithm $\textsc{QueryIntervalTree}(\nu, q_x)$

1. **if** ν is not a leaf
2. **then if** $q_x < x_{\text{mid}}(\nu)$
3. **then** Traverse list $L_{\text{left}}(\nu)$, starting at the interval with the leftmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
4. $\textsc{QueryIntervalTree}(lc(\nu), q_x)$
5. **else** Traverse list $L_{\text{right}}(\nu)$, starting at the interval with the rightmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
6. $\textsc{QueryIntervalTree}(rc(\nu), q_x)$

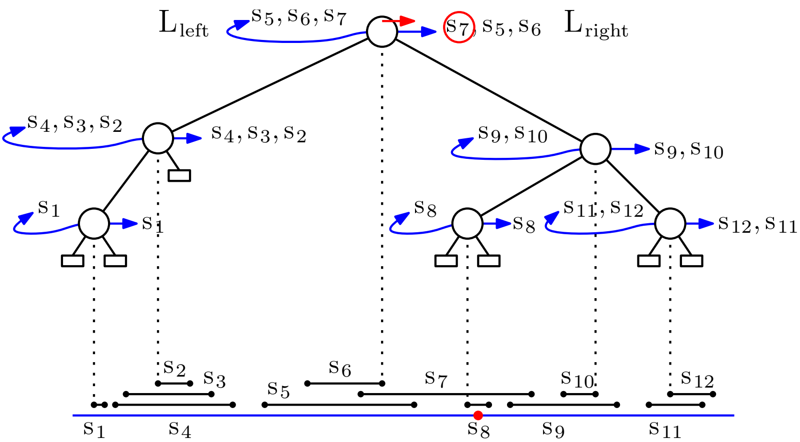
Interval tree: query example



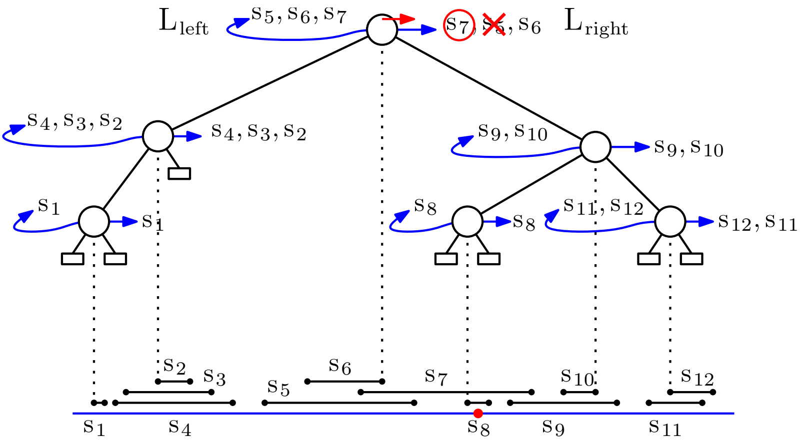
Interval tree: query example



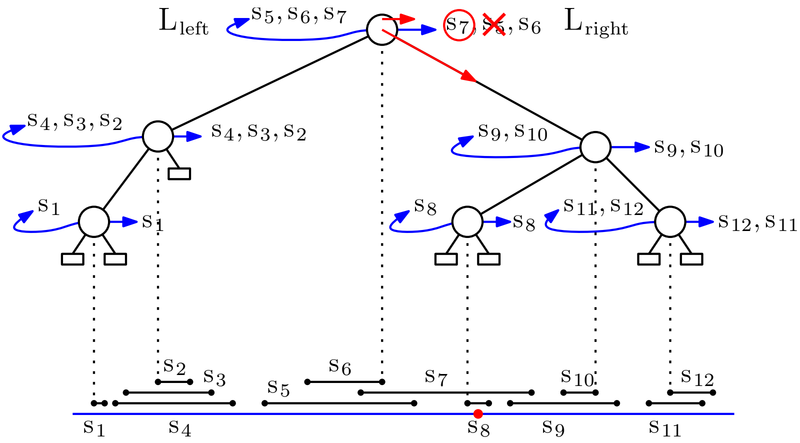
Interval tree: query example



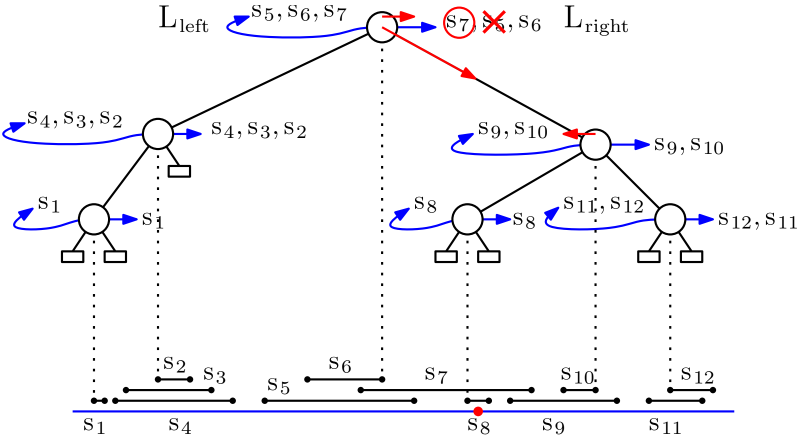
Interval tree: query example



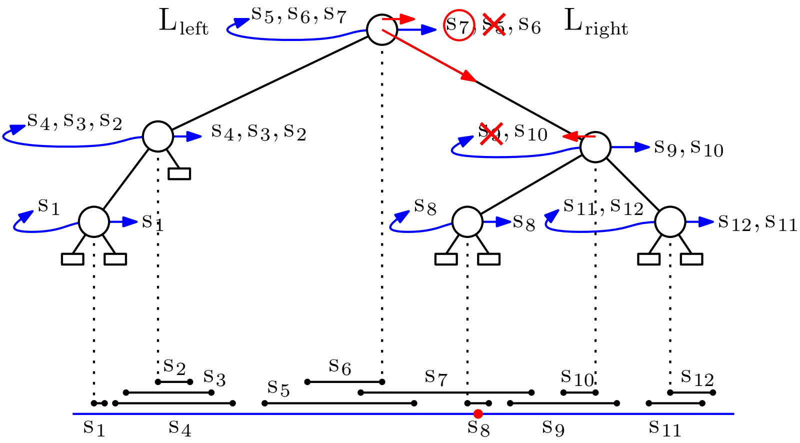
Interval tree: query example



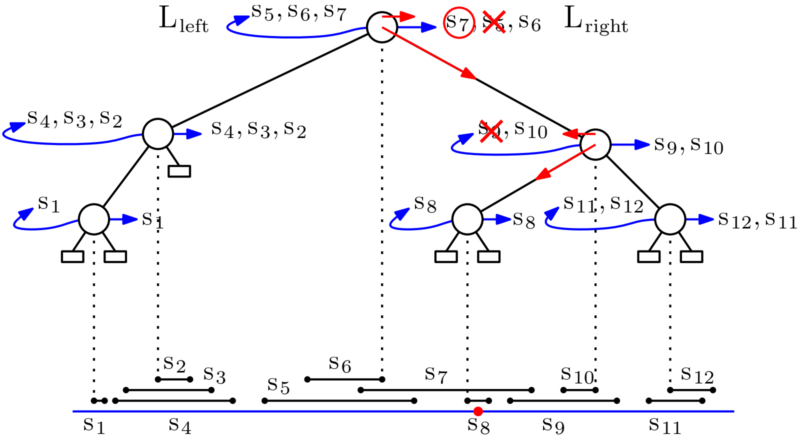
Interval tree: query example



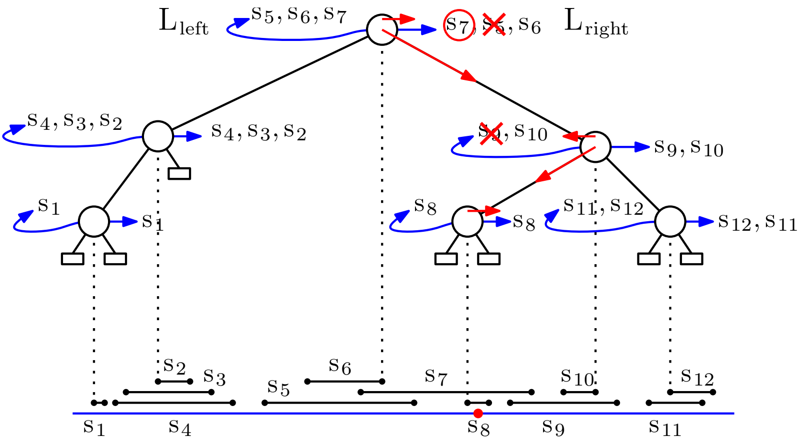
Interval tree: query example



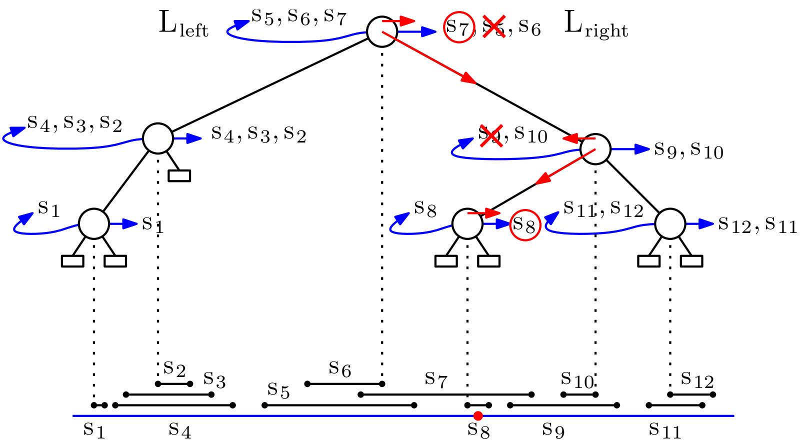
Interval tree: query example



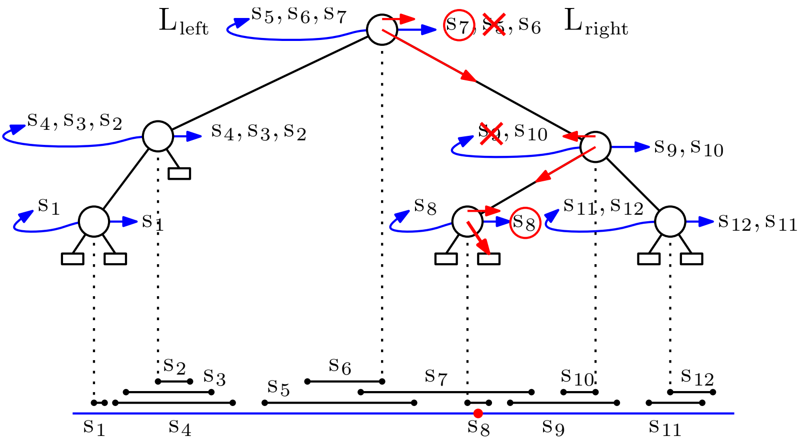
Interval tree: query example



Interval tree: query example



Interval tree: query example



Interval tree: query time

The query follows only one path in the tree, and that path has length $O(\log n)$

The query traverses $O(\log n)$ lists. Traversing a list with k' answers takes $O(1+k')$ time

The total time for list traversal is therefore $O(\log + k)$, with the total number of answers reported (no answer is found more than once)

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

Interval tree: query example

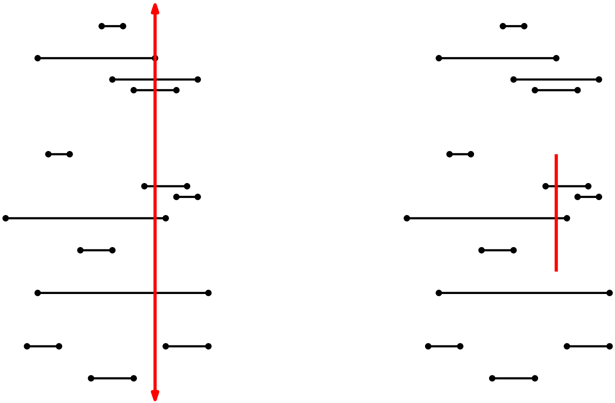
Algorithm CONSTRUCTINTERVALTREE(I)
Input. A set I of intervals on the real line
Output. The root of an interval tree for I

```
if I = ∅
    then return an empty leaf
    else Create a node ν. Compute x_mid, the median of the
         set of interval endpoints, and store x_mid with ν
         Compute I_mid and construct two sorted lists for I_mid:
         a list L_left(ν) sorted on left endpoint and a list
         L_right(ν) sorted on right endpoint. Store these two
         lists at ν
         lc(ν) ← CONSTRUCTINTERVALTREE(I_left)
         rc(ν) ← CONSTRUCTINTERVALTREE(I_right)
         return ν
```

Interval tree: result

Theorem: An interval tree for a set I of n intervals uses $O(n)$ storage and can be built in $O(n \log n)$ time. All intervals that contain a query point can be reported in $O(\log n + k)$ time, where k is the number of reported intervals.

Back to the plane

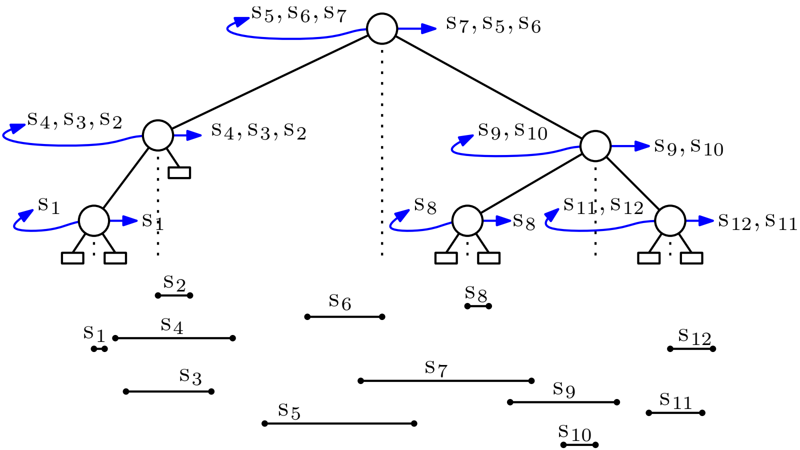


Back to the plane

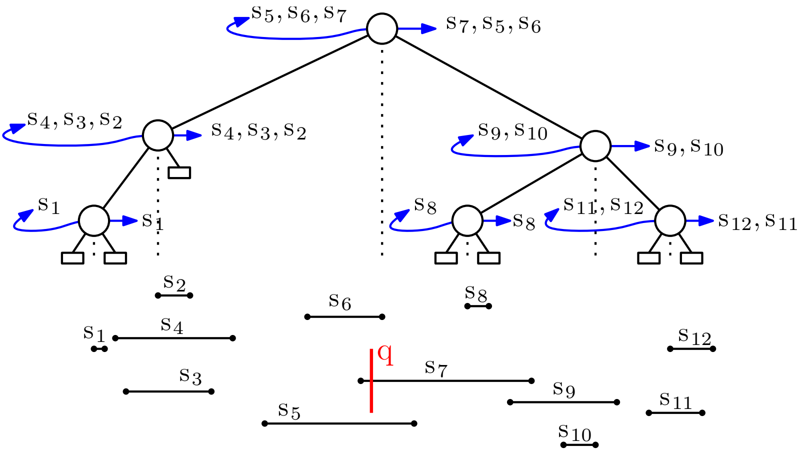
Suppose we use an interval tree on the x-intervals of the horizontal line segments?

Then the lists L_{left} and L_{right} are not suitable anymore to solve the query problem for the segments corresponding to I_{mid}

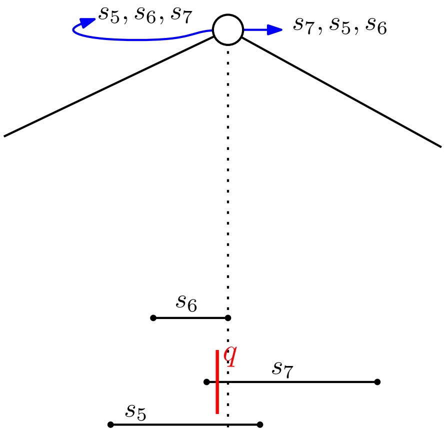
Back to the plane



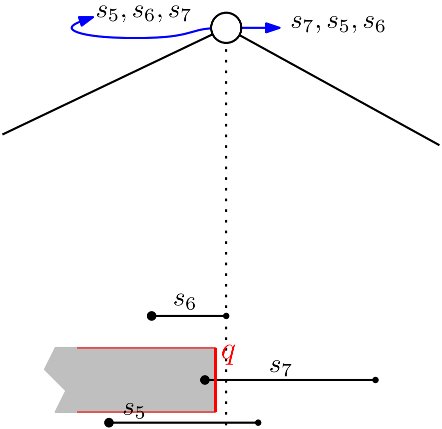
Back to the plane



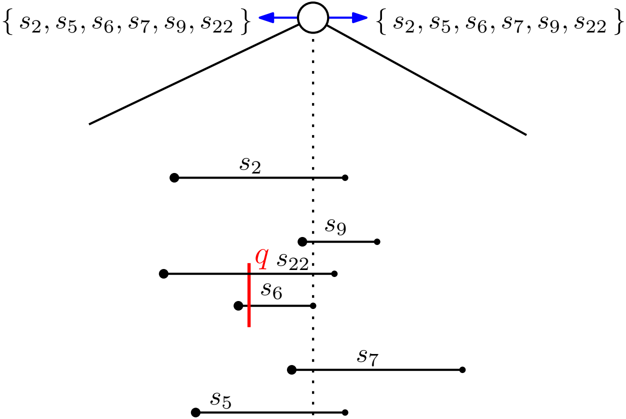
Back to the plane



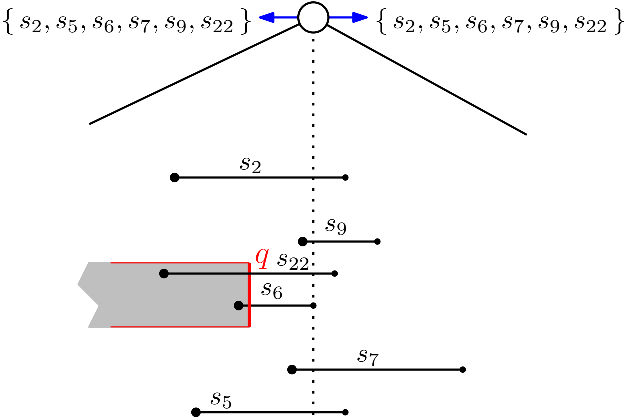
Back to the plane



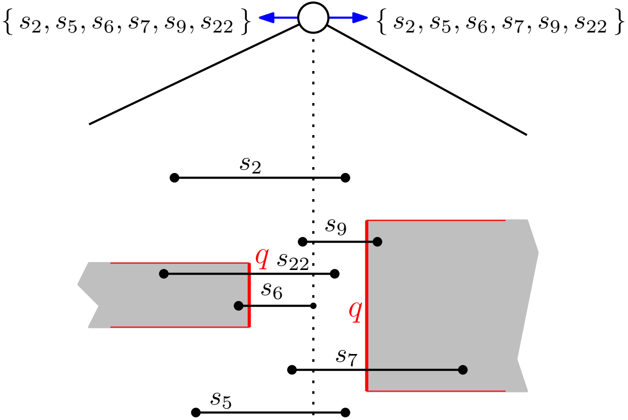
Back to the plane



Back to the plane



Back to the plane

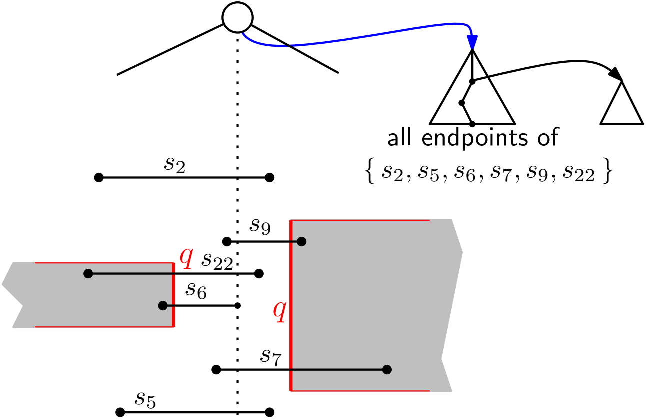


Segment intersection queries

We can use a *range tree* as the associated structure; we only need one that stores all of the endpoints, to replace L_{left} and L_{right}

Instead of traversing L_{left} or L_{right}, we perform a query with the region left or right, respectively, of q

Segment intersection queries



Segment intersection queries

In total, there are $O(n)$ range trees that together store $2n$ points, so the total storage needed by all associated structures is $O(n \log n)$

A query with a vertical segment leads to $O(\log n)$ range queries

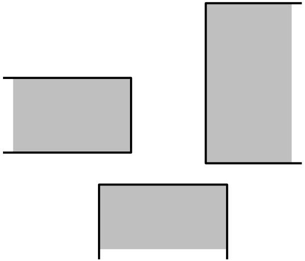
If fractional cascading is used in the associated structures, the overall query time is $O(\log^2 n + k)$

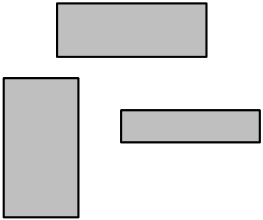
Question: How about the construction time?

3- and 4-sided ranges

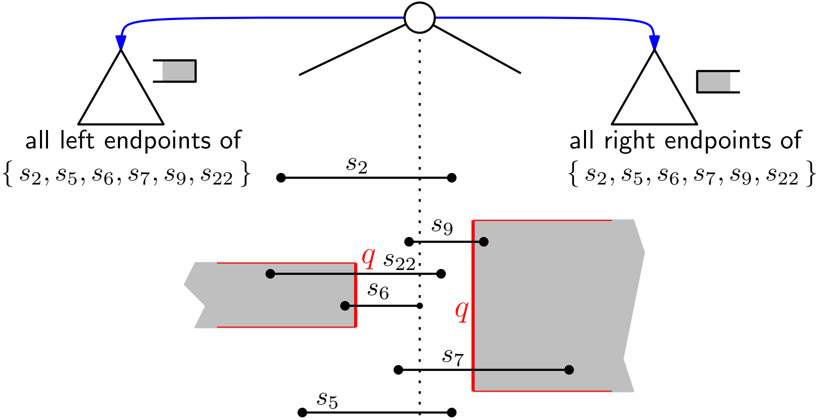
Considering the associated structure, we only need 3-sided range queries, whereas the range tree provides 4-sided range queries

Can the 3-sided range query problem be solved more efficiently than the 4-sided (rectangular) range query problem?





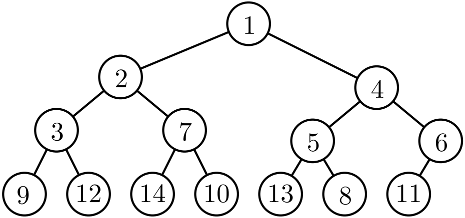
Scheme of structure



Heap and search tree

A priority search tree is like a heap on x-coordinate and binary search tree on y-coordinate at the same time

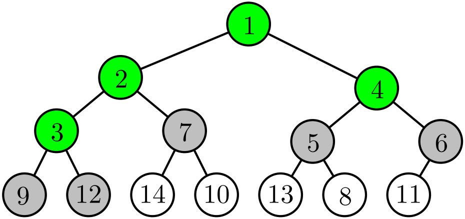
Recall the heap:



Heap and search tree

A priority search tree is like a heap on x-coordinate and binary search tree on y-coordinate at the same time

Recall the heap:



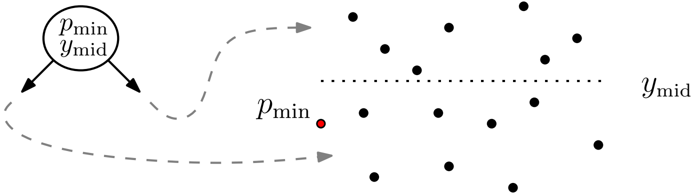
Report all values ≤ 4

Priority search tree

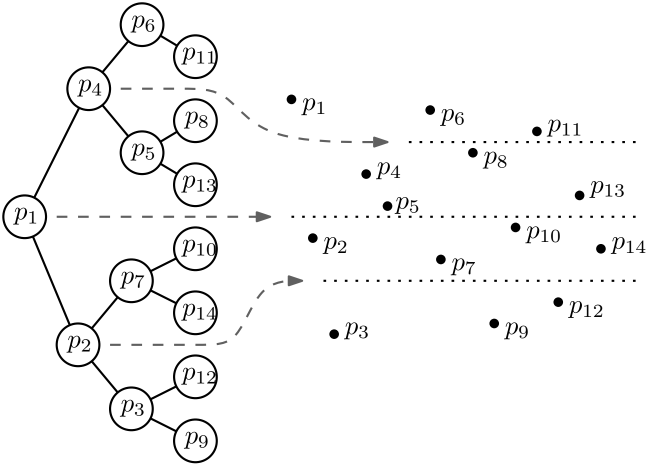
If $P = \emptyset$, then a priority search tree is an empty leaf

Otherwise, let $p_{\min}$ be the leftmost point in P, and let y_{mid} be the median y-coordinate of $P \setminus \{p_{\min}\}$

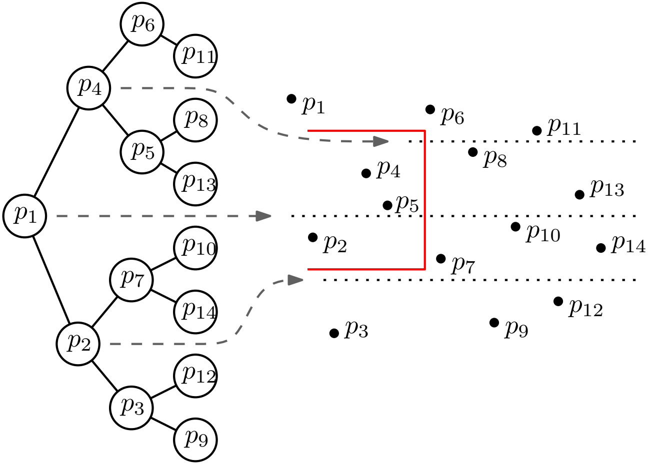
The priority search tree has a node v that stores $p_{\min}$ and y_{mid}, and a left subtree and right subtree for the points in $P \setminus \{p_{\min}\}$ with y-coordinate $\leq y_{\mathrm{mid}}$ and $> y_{\mathrm{mid}}$



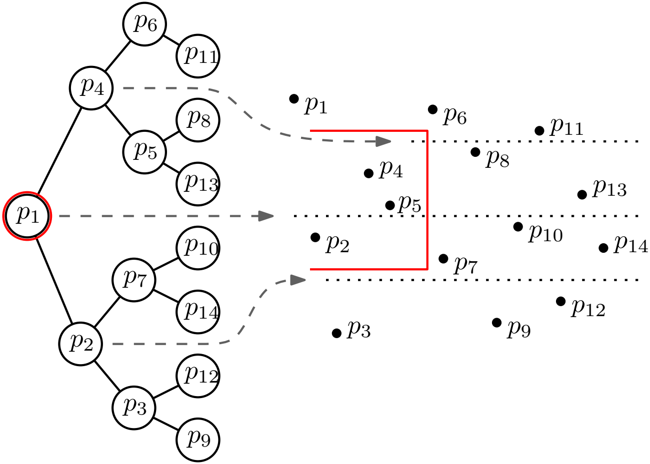
Priority search tree



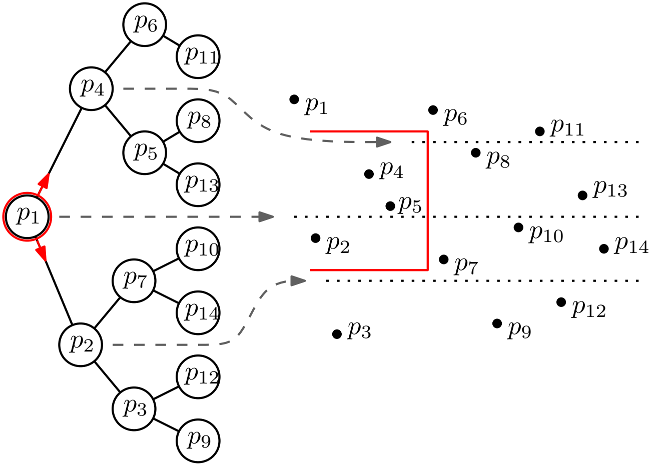
Priority search tree



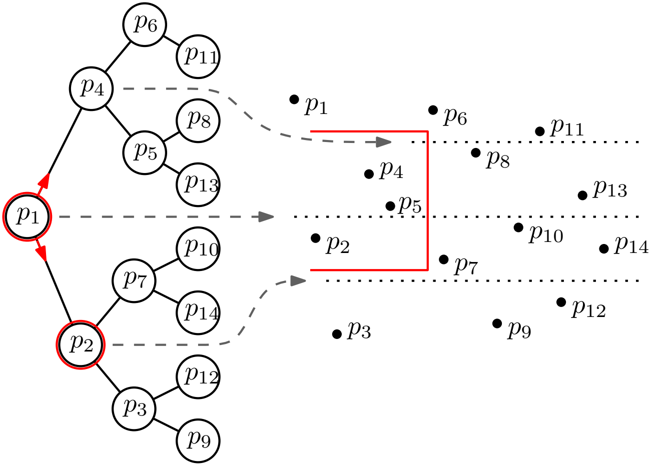
Priority search tree



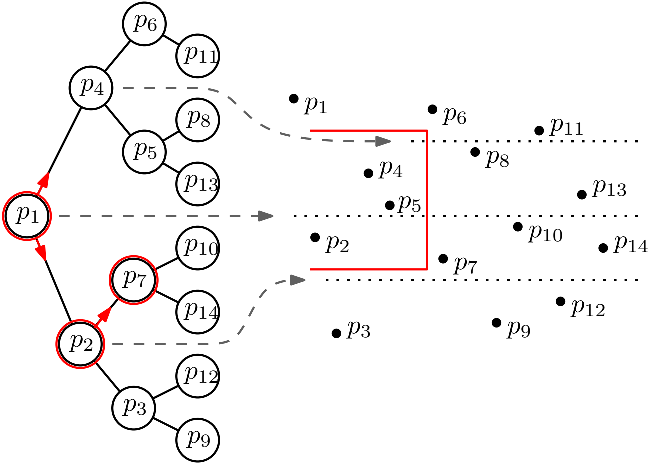
Priority search tree



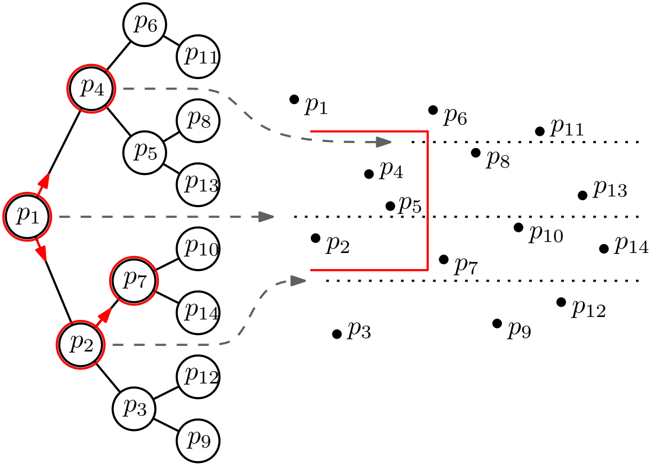
Priority search tree



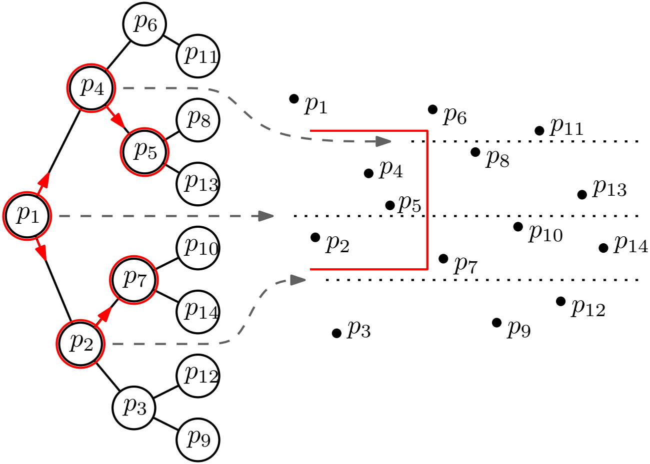
Priority search tree



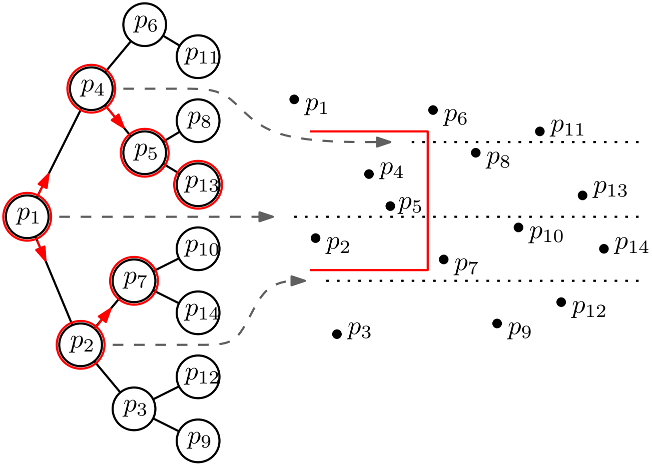
Priority search tree



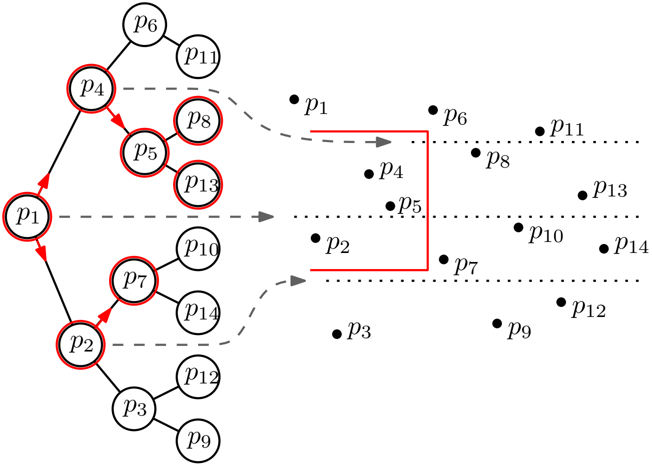
Priority search tree



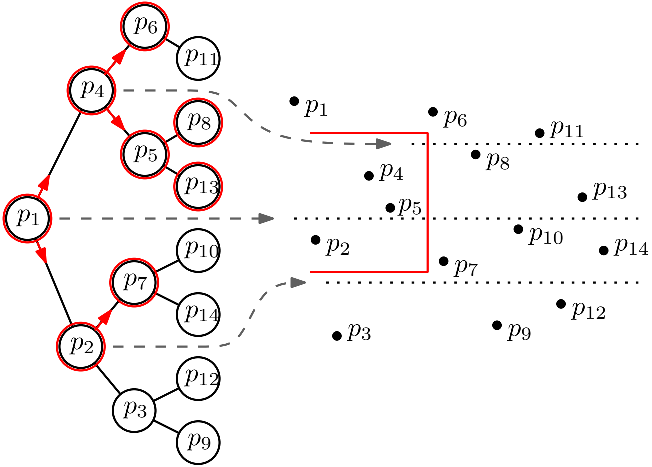
Priority search tree



Priority search tree



Priority search tree

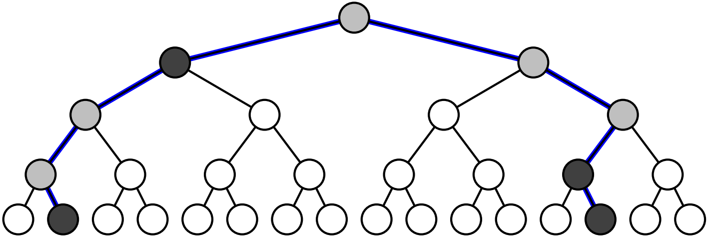


Query algorithm

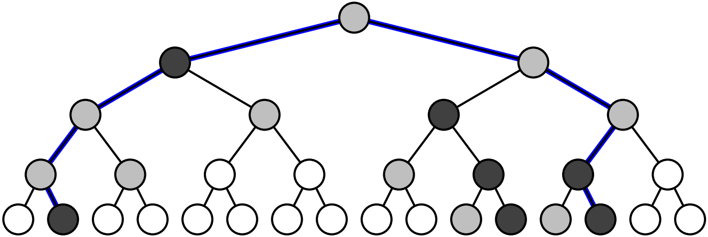
Algorithm $\text{QUERYPRIOSEARCHTREE}(\mathcal{T}, (-\infty : q_x] \times [q_y : q'_y])$

1. Search with q_y and q'_y in $\mathcal{T}$
2. Let ν_{split} be the node where the two search paths split
3. **for** each node ν on the search path of q_y or q'_y
4.   **do if** $p(\nu) \in (-\infty : q_x] \times [q_y : q'_y]$ **then** report $p(\nu)$
5. **for** each node ν on the path of q_y in the left subtree of ν_{split}
6.   **do if** the search path goes left at ν
7.   **then** $\text{REPORTINSUBTREE}(rc(\nu), q_x)$
8. **for** each node ν on the path of q'_y in the right subtree of ν_{split}
9.   **do if** the search path goes right at ν
10.   **then** $\text{REPORTINSUBTREE}(lc(\nu), q_x)$

Structure of the query



Structure of the query



Query algorithm

ReportInSubtree(ν, q_x)
Input. The root ν of a subtree of a priority search tree and a value q_x
Output. All points in the subtree with x-coordinate at most q_x
1. **if** ν is not a leaf and $(p(\nu))_x \leq q_x$
2. **then** Report $p(\nu)$
3. ReportInSubtree($lc(\nu), q_x$)
4. ReportInSubtree($rc(\nu), q_x$)

This subroutine takes $O(1+k)$ time, for k reported answers

Query algorithm

The search paths to y and y' have $O(\log n)$ nodes. At each node $O(1)$ time is spent

No nodes outside the search paths are ever visited

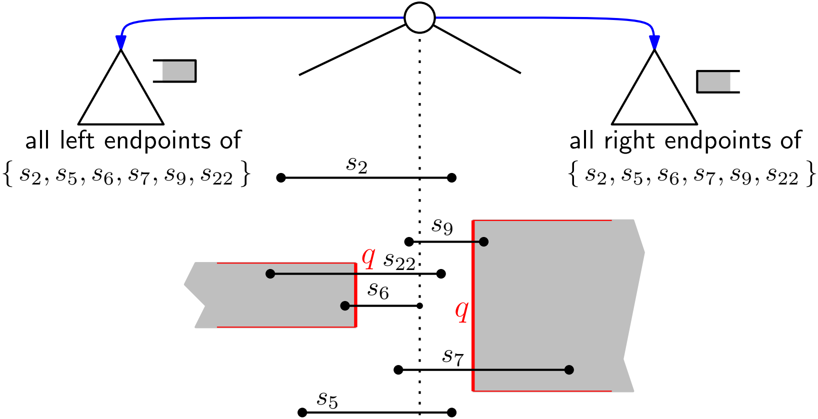
Subtrees of nodes between the search paths are queried like a heap, and we spend $O(1+k')$ time on each one

The total query time is $O(\log n + k)$, if k points are reported

Priority search tree: result

Theorem: A priority search tree for a set P of n points uses $O(n)$ storage and can be built in $O(n \log n)$ time. All points that lie in a 3-sided query range can be reported in $O(\log n + k)$ time, where k is the number of reported points

Scheme of structure



Storage of the structure

Question: What are the storage requirements of the structure for querying with a vertical segment in a set of horizontal segments?

Query time of the structure

Question: What is the query time of the structure for querying with a vertical segment in a set of horizontal segments?

Result

Theorem: A set of n horizontal line segments can be stored in a data structure with size $O(n)$ such that intersection queries with a vertical line segment can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported

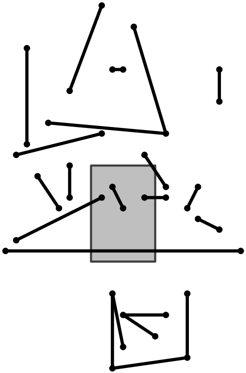
Result

Recall that the **windowing problem** is solved with a combination of a range tree and the structure just described

Theorem: A set of n axis-parallel line segments can be stored in a data structure with size $O(n \log n)$ such that windowing queries can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported

Windowing

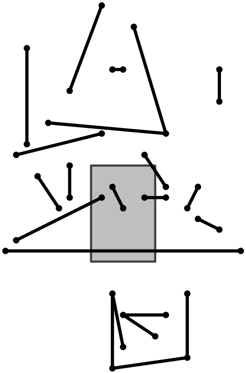
Given a set of n arbitrary, non-crossing line segments, preprocess them into a data structure so that the ones that intersect a query rectangle can be reported efficiently



Windowing

Two cases of intersection:

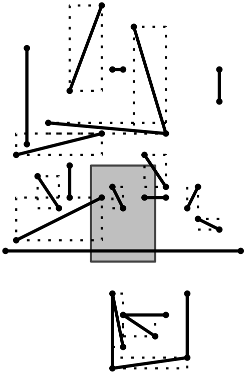
- An endpoint lies inside the query window; solve with range trees
- The segment intersects a side of the query window; solve how?



Using a bounding box?

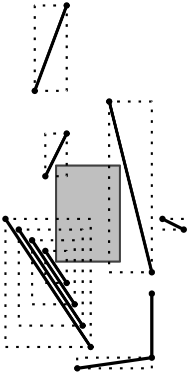
If the query window intersects the line segment, then it also intersects the bounding box of the line segment (whose sides are axis-parallel segments)

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



Using a bounding box?

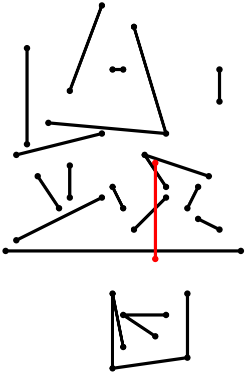
But: if the query window intersects bounding box sides does not imply that it intersects the corresponding segments



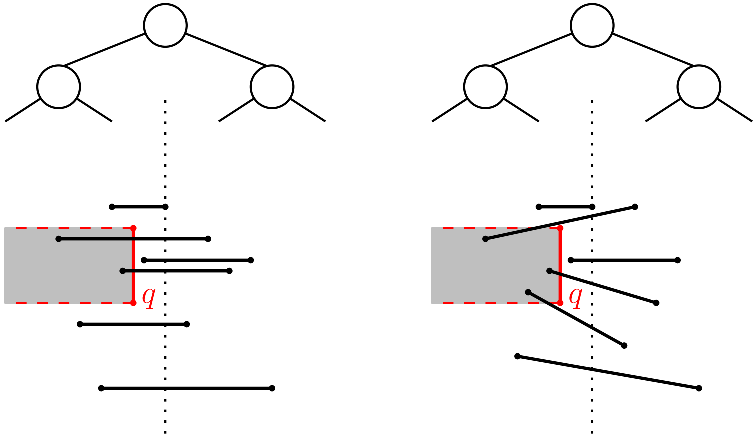
Windowing

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

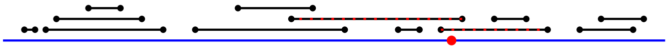


Using an interval tree?



Interval querying

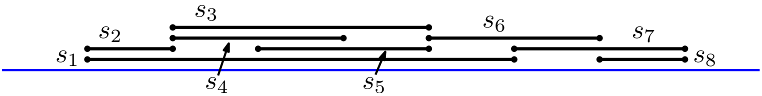
Given a set I of n intervals on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



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

Interval querying

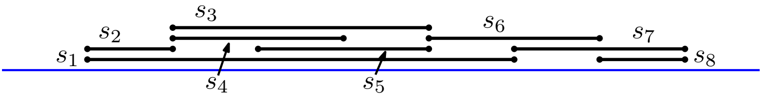
Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of n *segments* on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



The new structure is called the *segment tree*

Locus approach

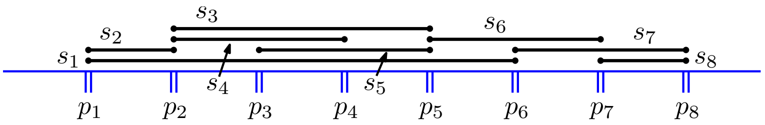
The locus approach is the idea to partition the solution space into parts with equal answer sets



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

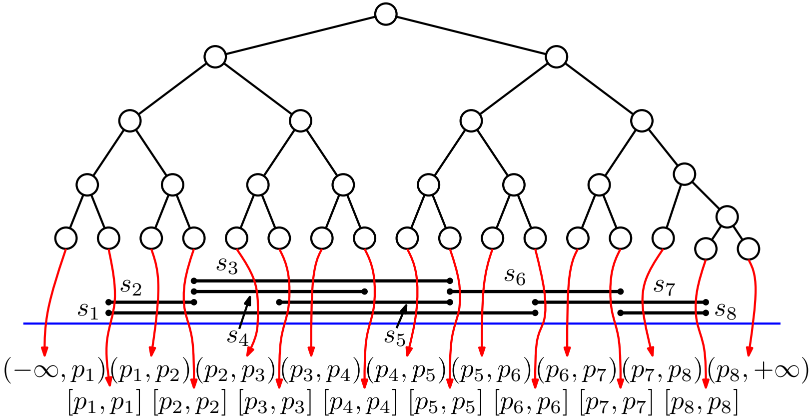
Locus approach

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

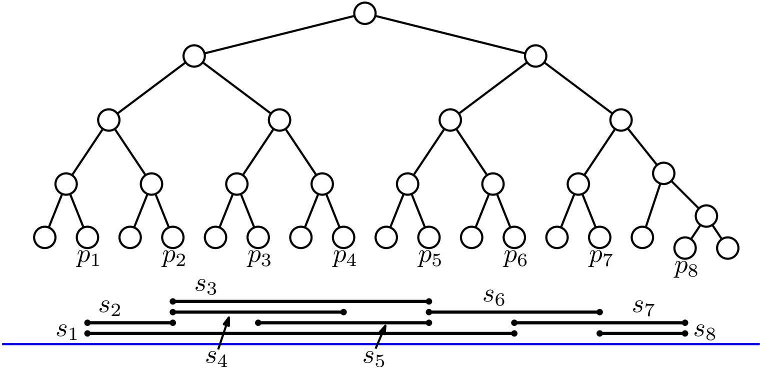
Each segment from the set S can be stored with all leaves whose elementary interval it contains: $[p_i, p_j]$ is stored with
$[p_i, p_i], (p_i, p_{i+1}), \ldots, [p_j, p_j]$

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



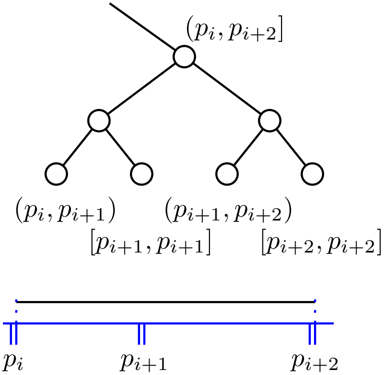
Locus approach

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

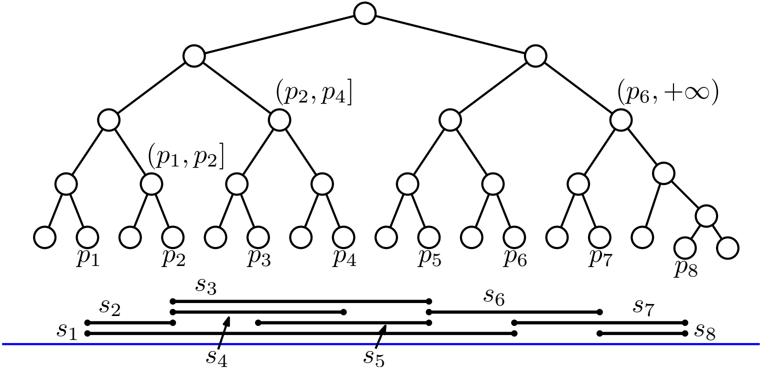
Towards segment trees

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

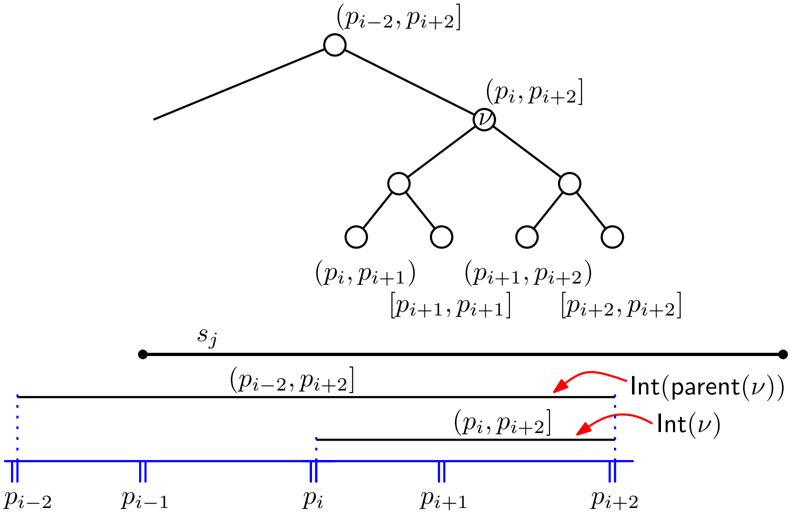
Let $\mathrm{Int}(v)$ denote the interval of node v

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

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

Towards segment trees



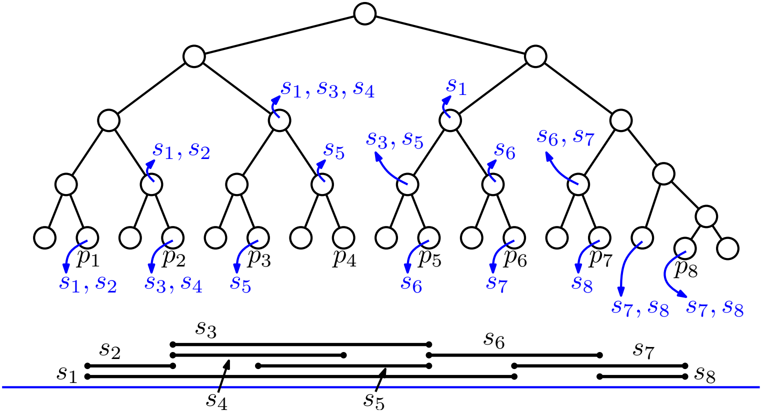
Segment trees

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)

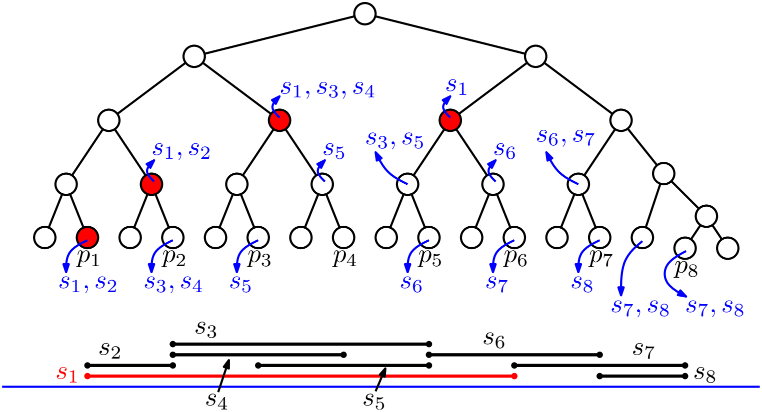
The canonical subset (of S) of a node ν is the subset of segments s_j for which

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

Segment trees



Segment trees



Segment trees

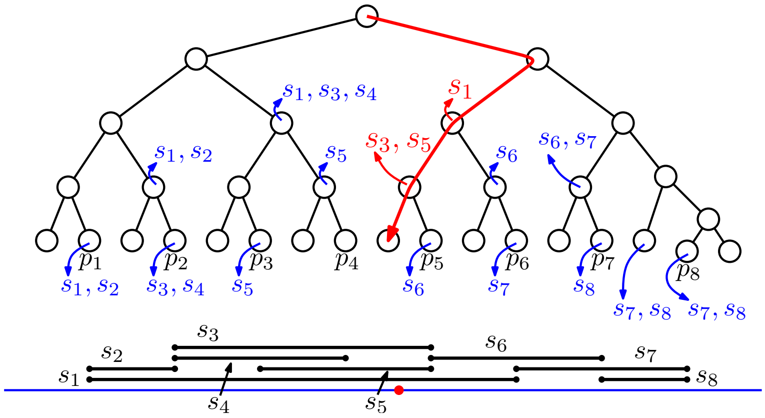
Question: Why are no segments stored with nodes on the leftmost and rightmost paths of the segment tree?

Query algorithm

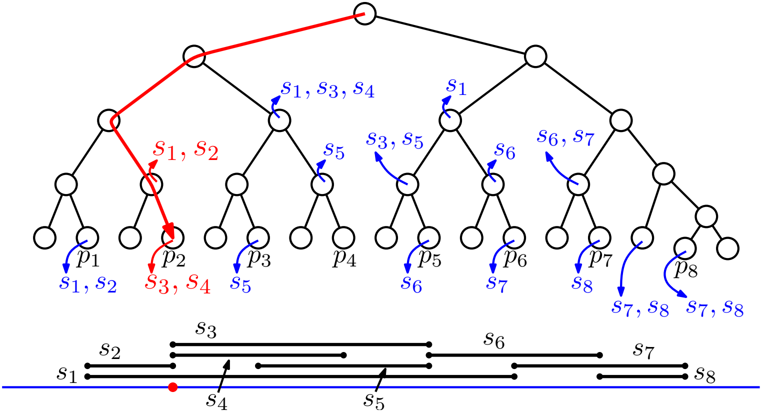
The query algorithm is trivial:

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

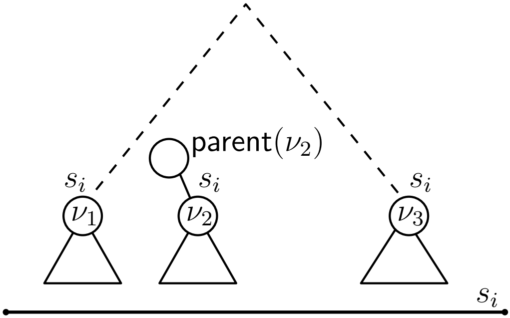
Segments stored at many nodes

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 ν_1, ν_2, and ν_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 $O(n \log n)$

Result

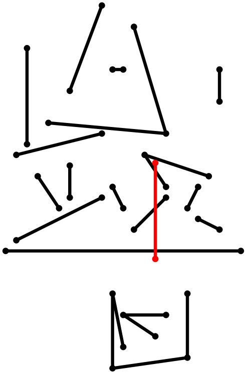
Theorem: A segment tree storing n segments (=intervals) on the real line uses $O(n \log n)$ storage, can be built in $O(n \log n)$ time, and stabbing queries can be answered in $O(\log n + k)$ time, where k is the number of segments reported

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

Back to windowing

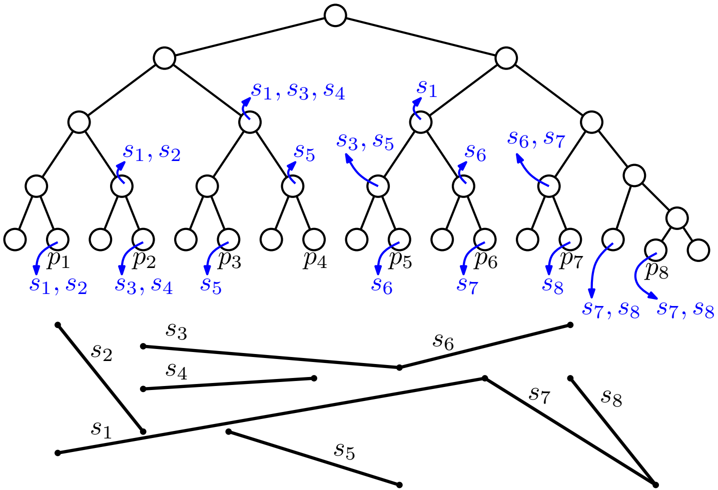
Problem arising from windowing:

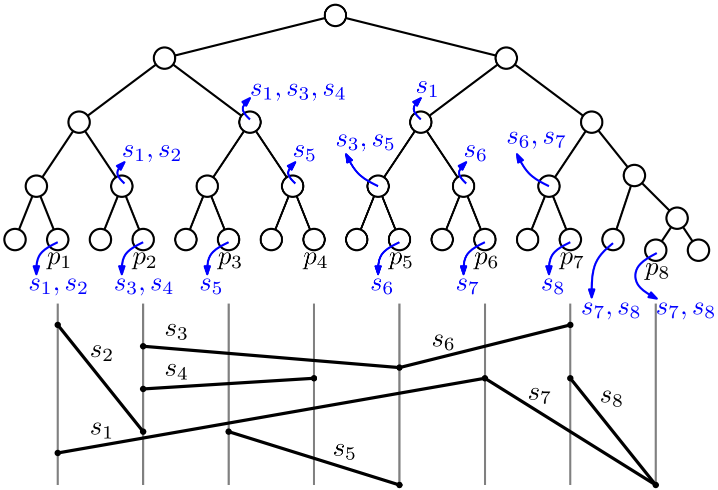
Given a set of arbitrarily oriented, non-crossing line segments, preprocess them into a data structure so that the ones intersecting a vertical (horizontal) query segment can be reported efficiently



Idea for solution

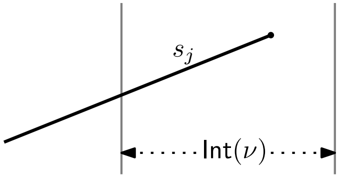
The main idea is to build a segment tree on the x-projections of the 2D segments, and replace the associated lists with a more suitable data structure


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_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8

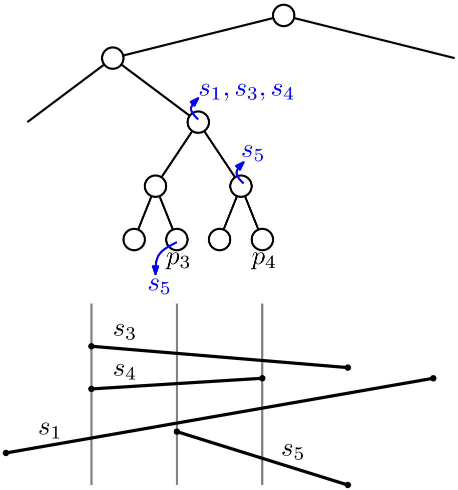

s_1, s_3, s_4
s_1
s_1, s_2
s_5
s_3, s_5
s_6
s_6, s_7
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
s_1, s_2
s_3, s_4
s_5
s_6
s_7
s_8
s_7, s_8
s_7, s_8
s_3
s_2
s_4
s_6
s_1
s_7
s_8
s_5

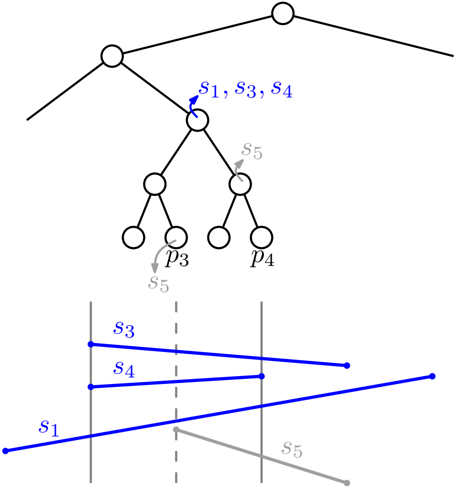
Observe that nodes now correspond to vertical slabs of the plane (with or without left and right bounding lines), and:

- if a segment s_i is stored with a node ν, then it crosses the slab of ν completely, but not the slab of the parent of ν
- 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

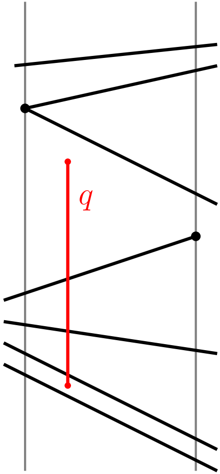

s_1, s_3, s_4
s_5
p_3
p_4
s_5
s_3
s_4
s_1
s_5

Querying

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

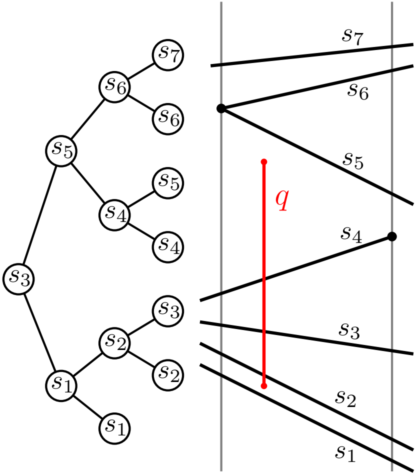
Only segments of S stored with nodes on the path down the tree using the x-coordinate of q can be answers

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 v 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