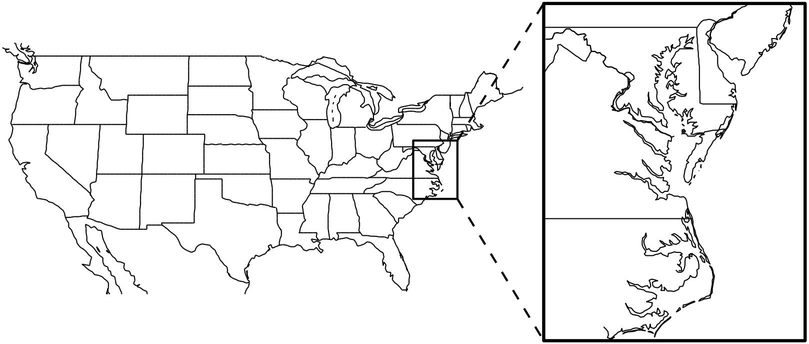
Windowing queries

Computational Geometry

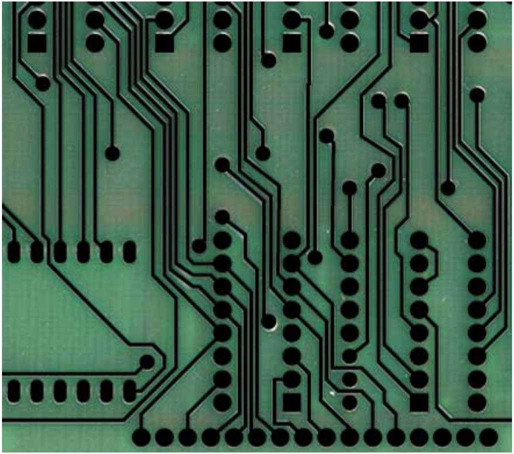
Lecture 14: Windowing queries

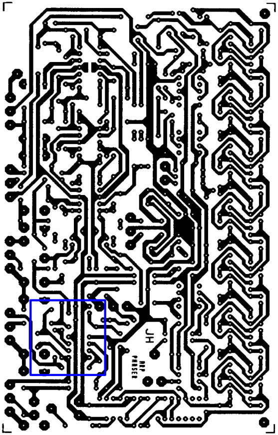
Windowing



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

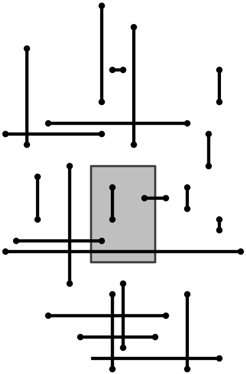
Windowing





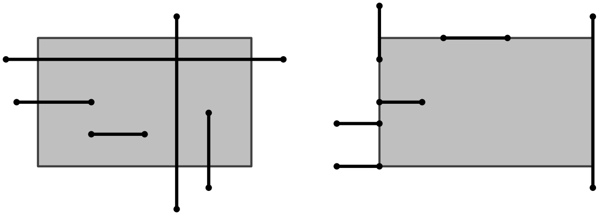
Windowing

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

How can a rectangle and an axis-parallel line segment intersect?

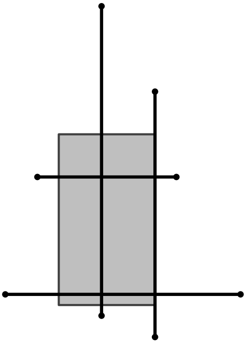


Windowing

Essentially two types:

- Segments whose endpoint lies in the rectangle (or both endpoints)
- Segments with both endpoints outside the rectangle

Segments of the latter type always intersect the boundary of the rectangle (even the left and/or bottom side)

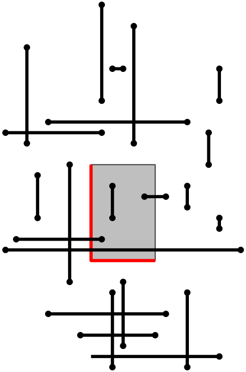


Windowing

Instead of storing axis-parallel segments and searching with a rectangle, we will:

- store the segment endpoints and query with the rectangle
- store the segments and query with the left side and the bottom side of the rectangle

Note that the query problem is at least as hard as rectangular range searching in point sets

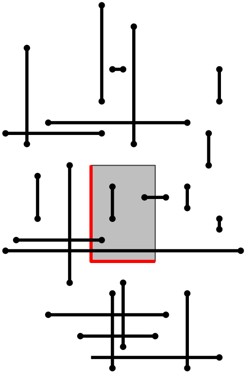


Windowing

Instead of storing axis-parallel segments and searching with a rectangle, we will:

- store the segment endpoints and query with the rectangle
- store the segments and query with the left side and the bottom side of the rectangle

Question: How often might we report the same segment?



Avoiding reporting the same segment several times

Use one representation of each segment, and store a mark bit with it that is initially FALSE

When we think we should report a segment, we first check its mark bit:

- if FALSE, then report it and set the mark bit to TRUE
- otherwise, don't do anything

After a query, we need to reset all mark bits to FALSE, for the next query (how?)

Windowing

Instead of storing axis-parallel segments and searching with a rectangle, we will:

- store the segment endpoints and query with the rectangle

  use range tree (from Chapter 5)

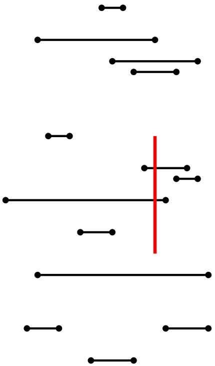
- store the segments and query with the left side and the bottom side of the rectangle

  need to develop data structure

Windowing

Current problem of our interest:

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



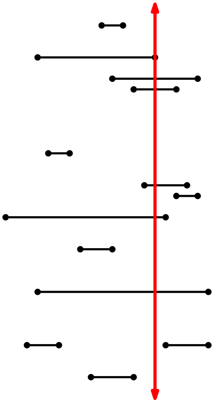
Question: Do we also need to store vertical segments for querying with vertical segments?

Windowing

Simpler query problem:

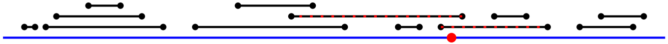
What if the vertical query segment is a full line?

Then the problem is essentially 1-dimensional



Interval querying

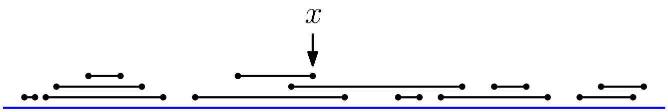
Given a set I of n intervals on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



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

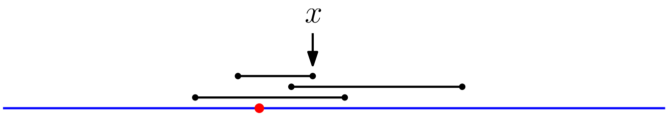
- the intervals I_{left} are stored in the left subtree of v
- the intervals I_{mid} are stored with v
- the intervals I_{right} are stored in the right subtree of v

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

How many intervals can be in I_{mid}? How should we store I_{mid}?

Interval tree: left and right lists

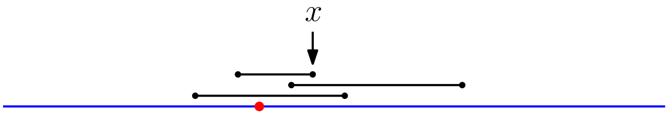
How is I_{mid} stored?



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

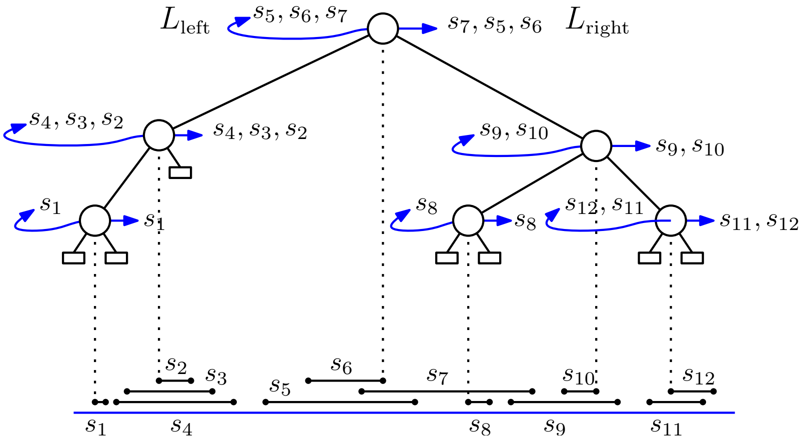


Make a list L_{left} using the left-to-right order of the *left endpoints* of I_{mid}

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

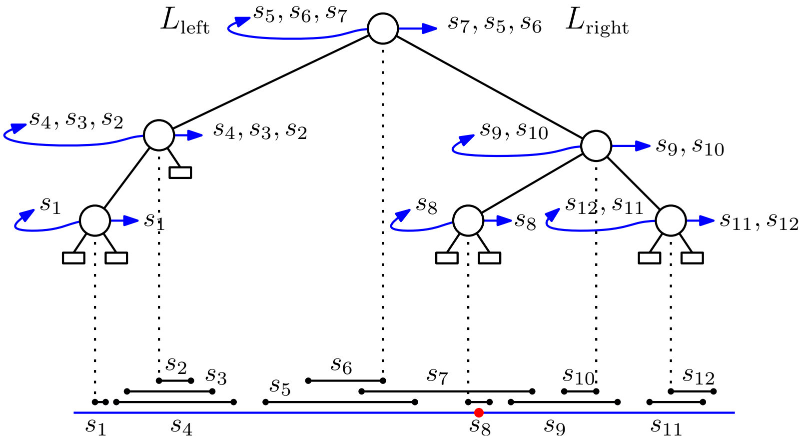
Consequently, the interval tree uses $O(n)$ storage

Interval querying

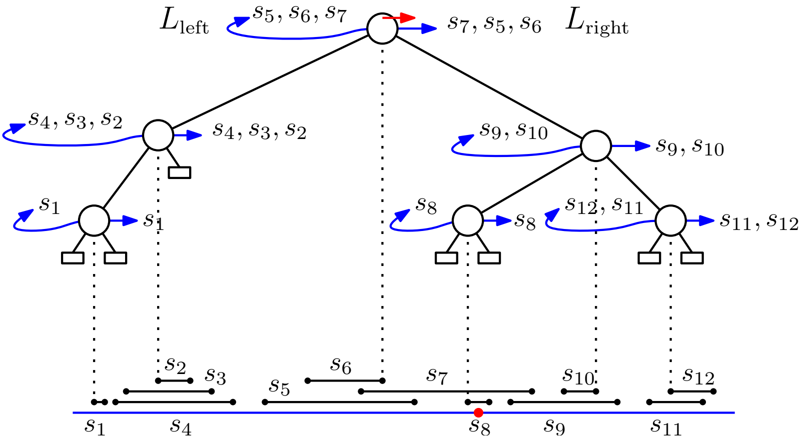
Algorithm $\textsc{QueryIntervalTree}(\nu, q_x)$

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. $\textsc{QueryIntervalTree}(rc(\nu), q_x)$

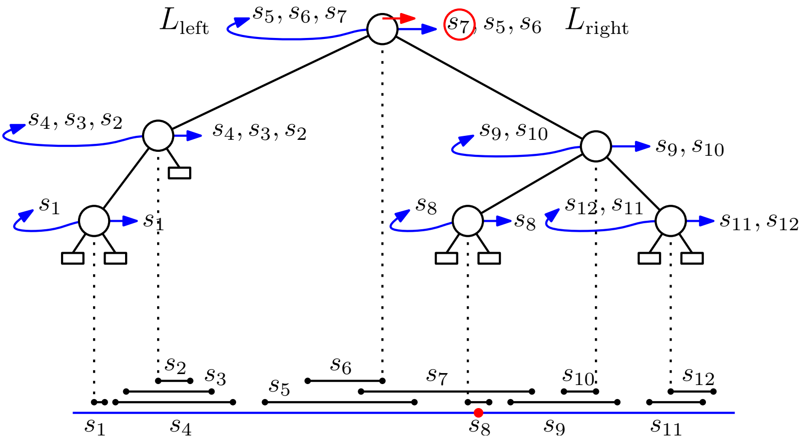
Interval tree: query example



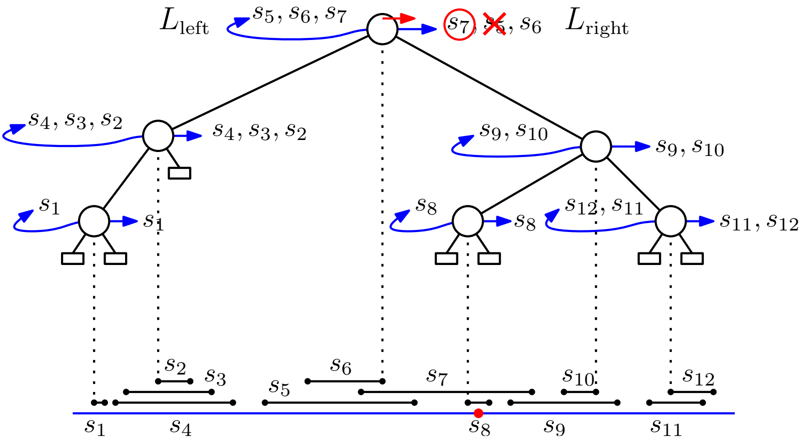
Interval tree: query example



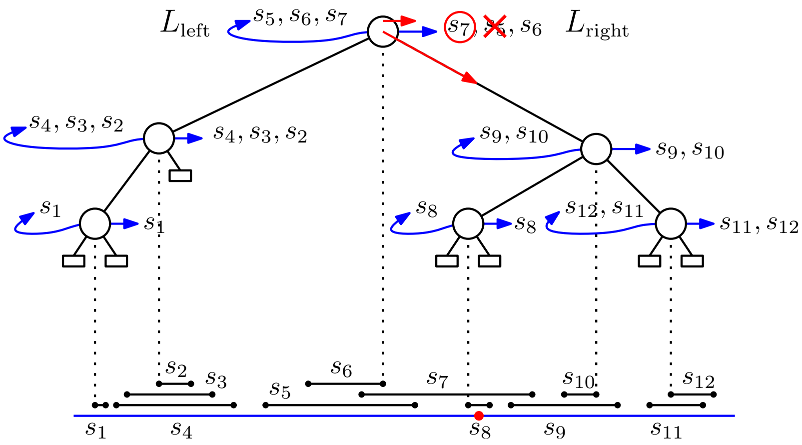
Interval tree: query example



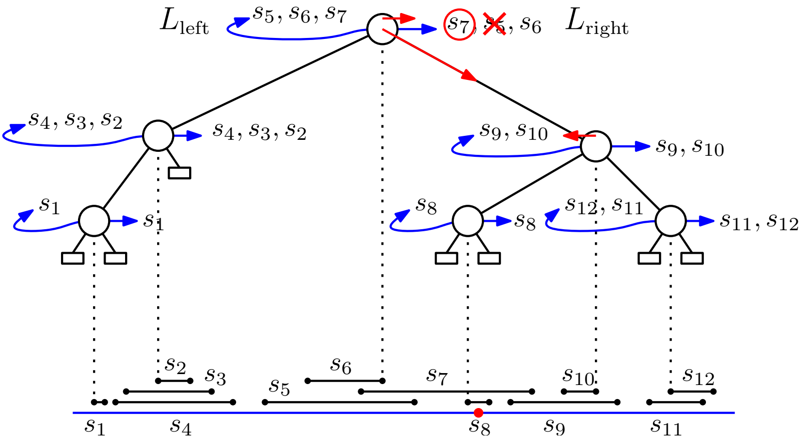
Interval tree: query example



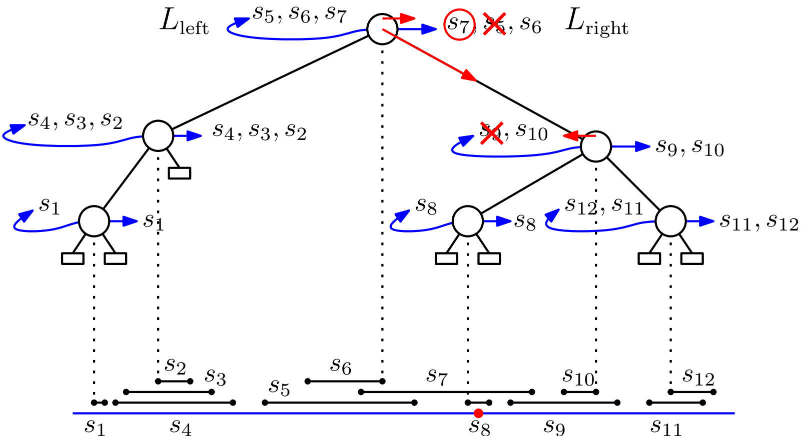
Interval tree: query example



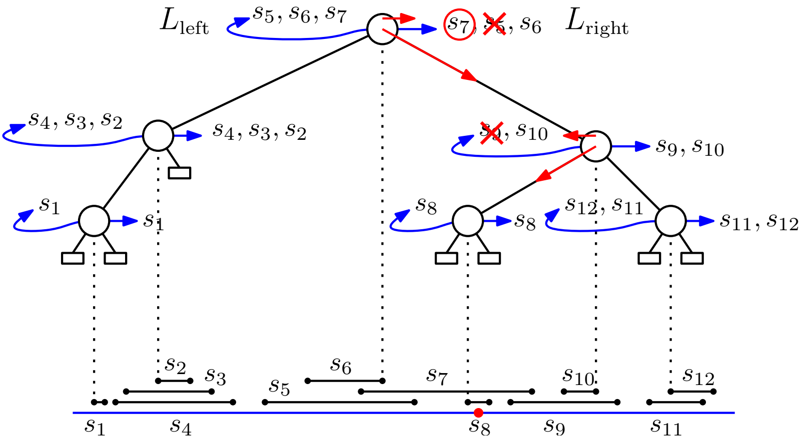
Interval tree: query example



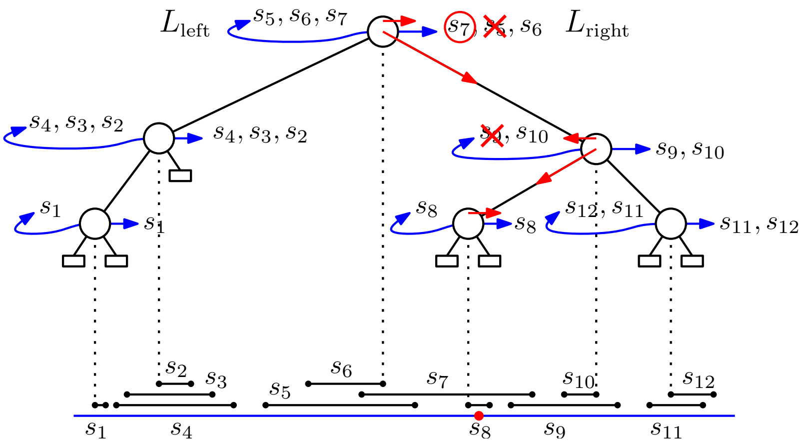
Interval tree: query example



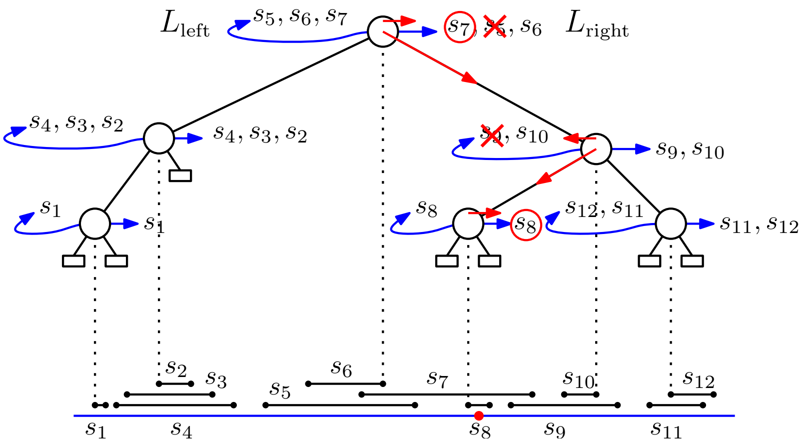
Interval tree: query example



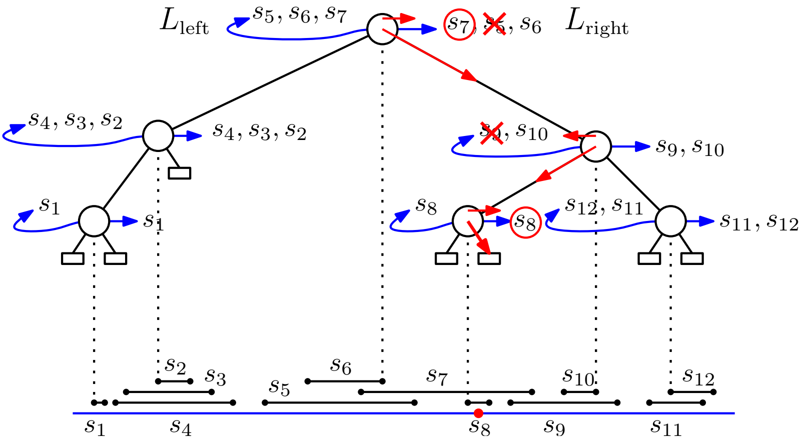
Interval tree: query example



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

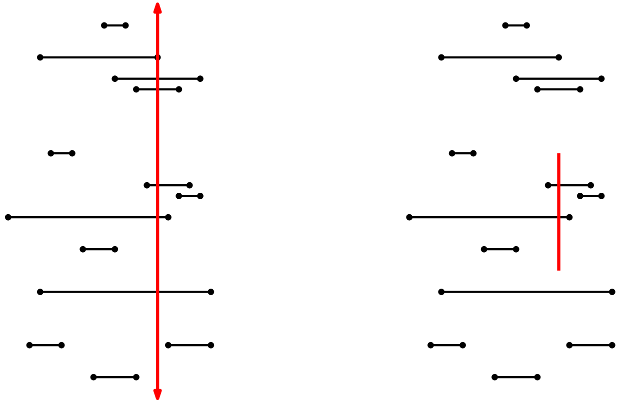
Algorithm ConstructIntervalTree(I)
Input. A set I of intervals on the real line
Output. The root of an interval tree for I

1. **if** $I = \emptyset$
2. **then return** an empty leaf
3. **else** Create a node ν. Compute x_{mid}, the median of the set of interval endpoints, and store x_{mid} with ν
4. Compute I_{mid} and construct two sorted lists for I_{mid}: a list $L_{\mathrm{left}}(\nu)$ sorted on left endpoint and a list $L_{\mathrm{right}}(\nu)$ sorted on right endpoint. Store these two lists at ν
5. $lc(\nu) \leftarrow$ ConstructIntervalTree(I_{left})
6. $rc(\nu) \leftarrow$ ConstructIntervalTree(I_{right})
7. **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

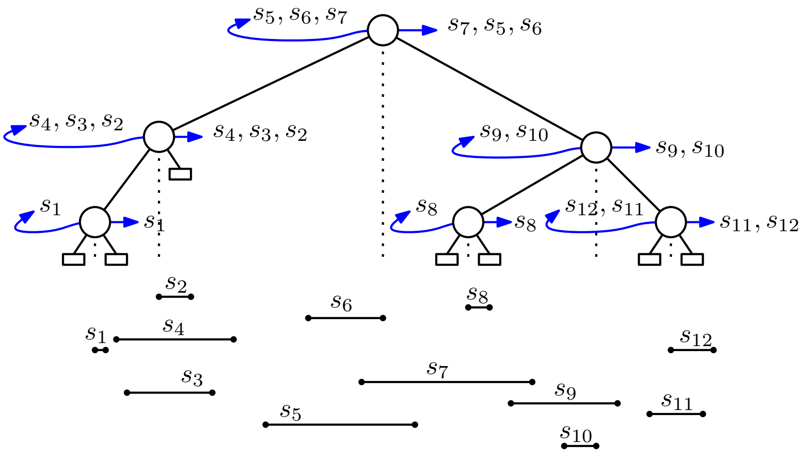


Back to the plane

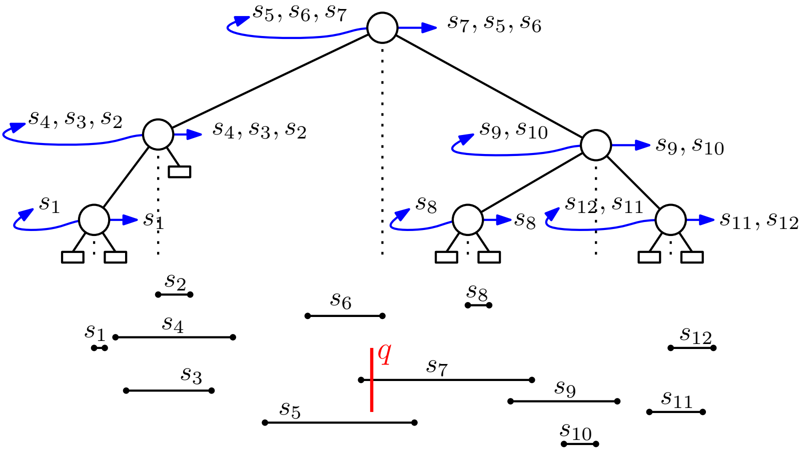
Suppose we use an interval tree on the x-intervals of the horizontal line segments?

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

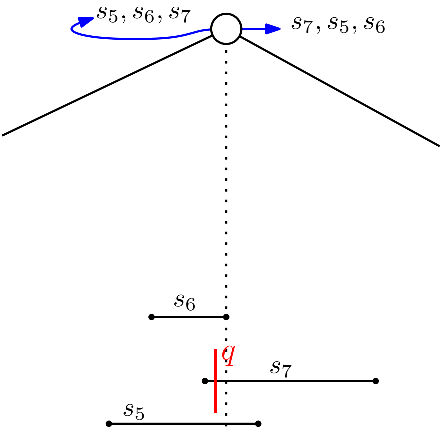
Back to the plane



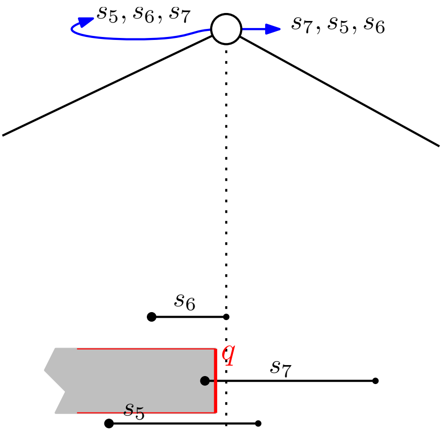
Back to the plane



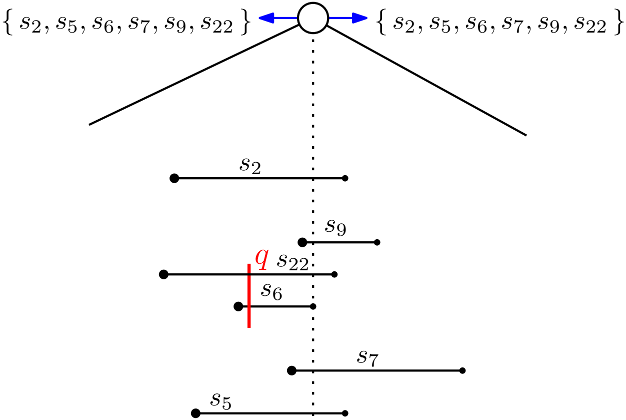
Back to the plane



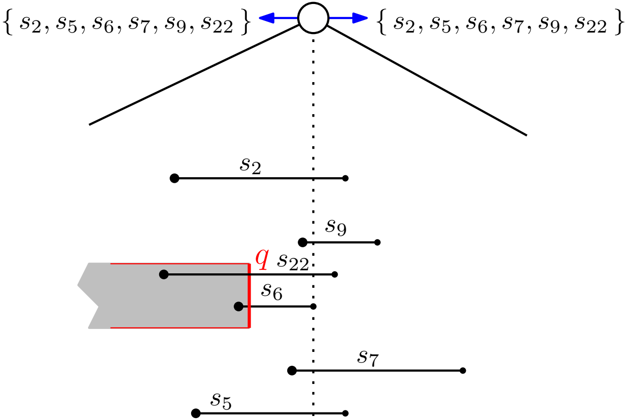
Back to the plane



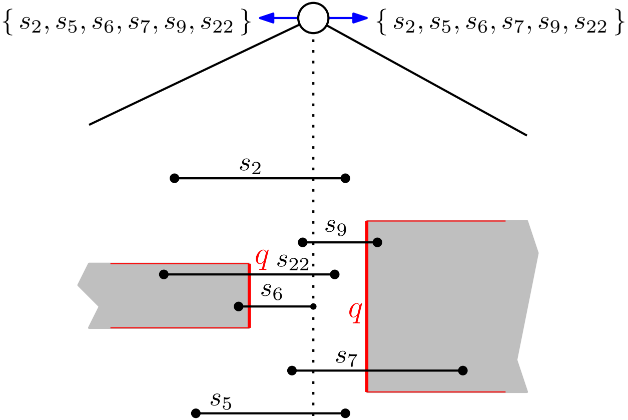
Back to the plane



Back to the plane



Back to the plane

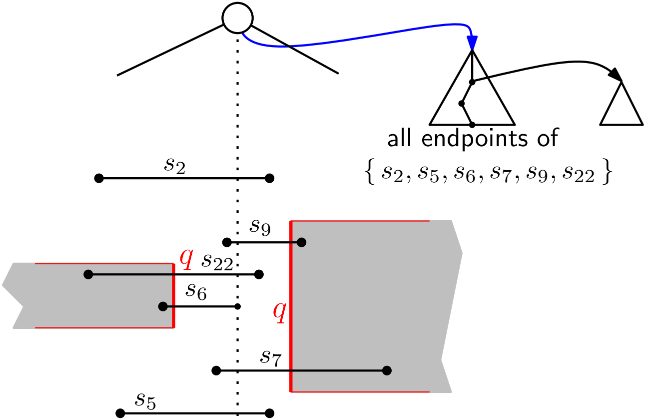


Segment intersection queries

We can use a *range tree* (chapter 5) 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

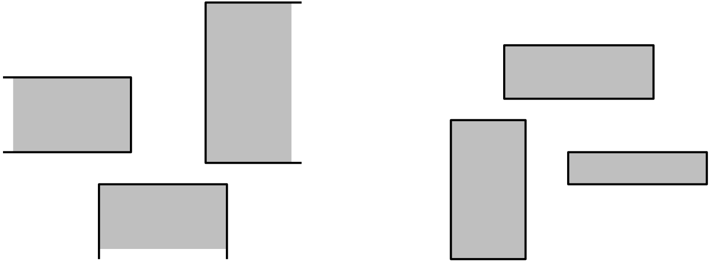
If fractional cascading is used in the associated structures, the overall query time is $O(\log^2 n + k)$

Question: How about the construction time?

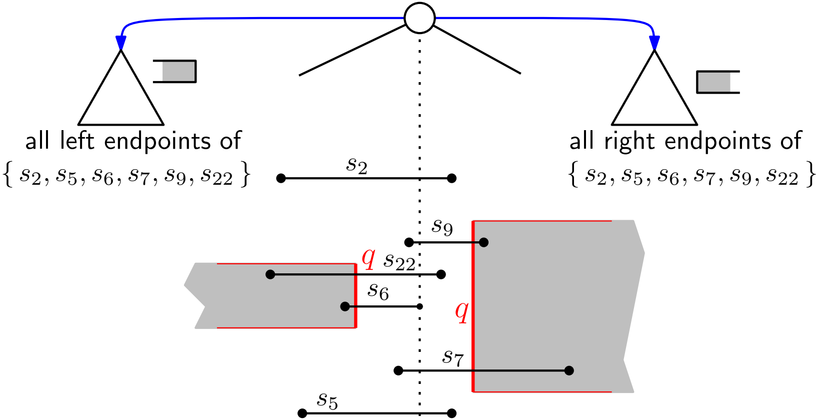
3- and 4-sided ranges

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?



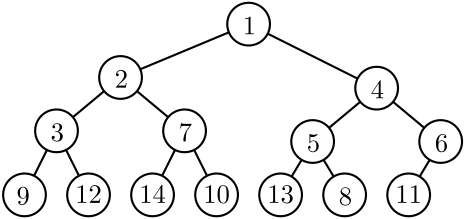
Scheme of structure



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

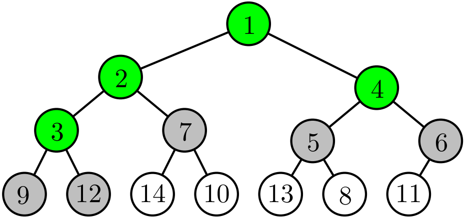
Recall the heap:



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:



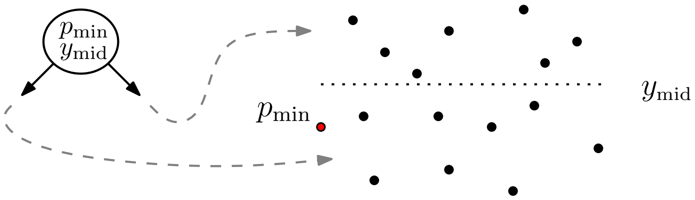
Report all values ≤ 4

Priority search tree

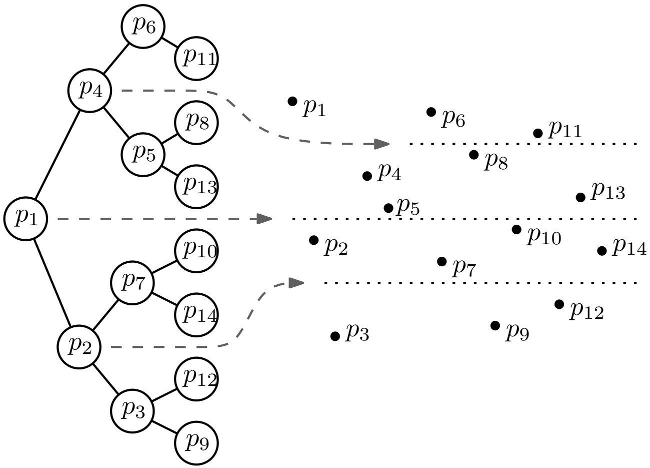
If $P = \emptyset$, then a priority search tree is an empty leaf

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

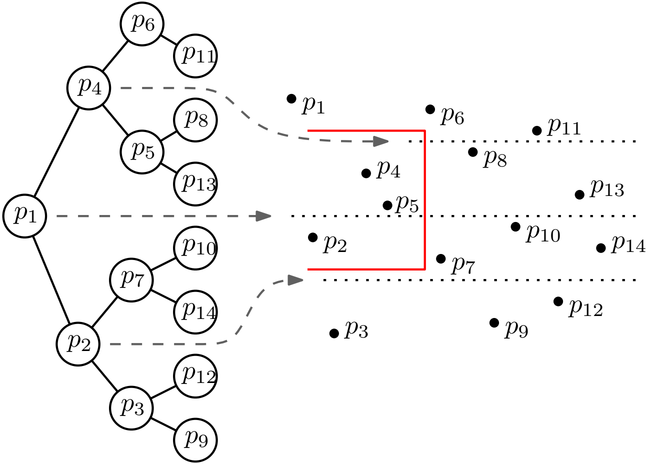
The priority search tree has a node ν that stores $p_{\min}$ and y_{mid}, and a left subtree and right subtree for the points in $P \setminus \{p_{\min}\}$ with y-coordinate $\leq y_{\mathrm{mid}}$ and $> y_{\mathrm{mid}}$



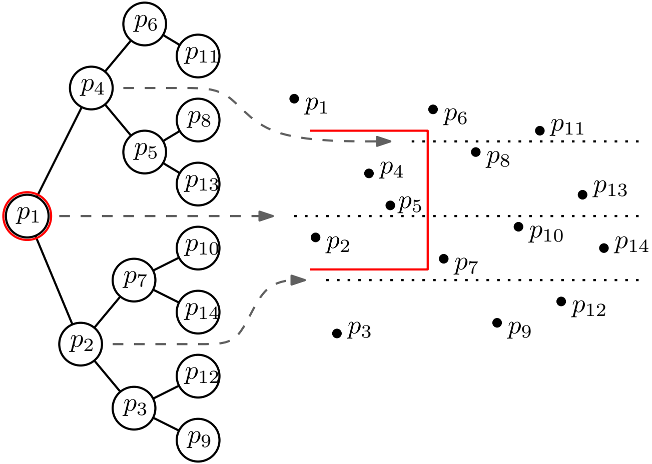
Priority search tree



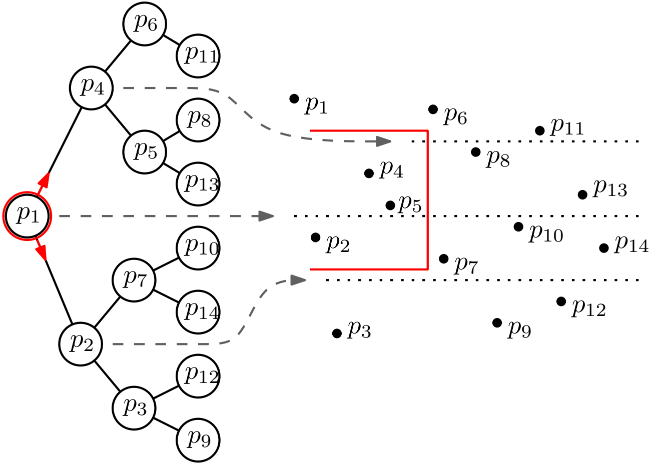
Priority search tree



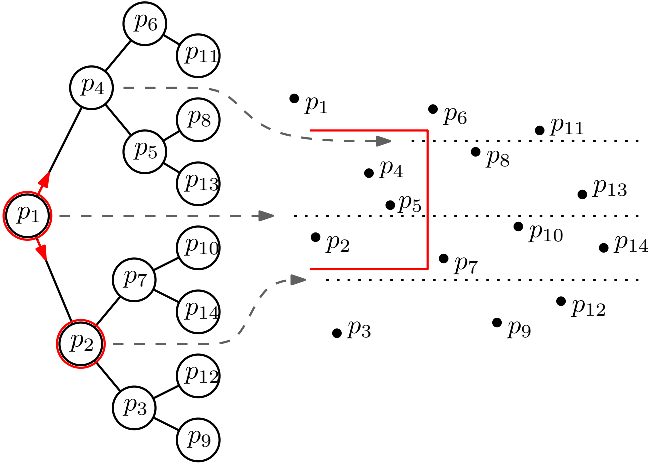
Priority search tree



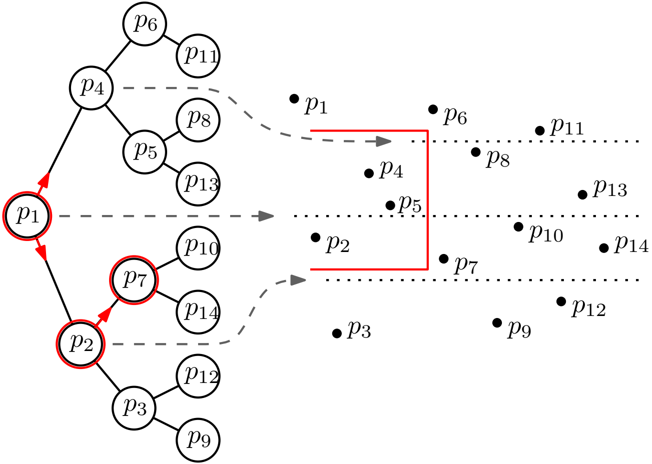
Priority search tree



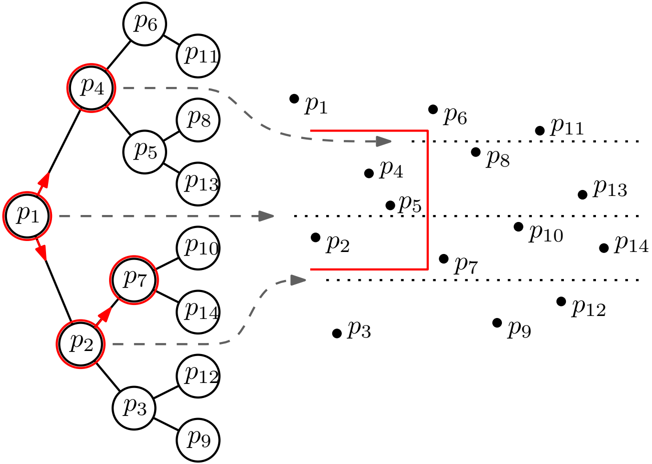
Priority search tree



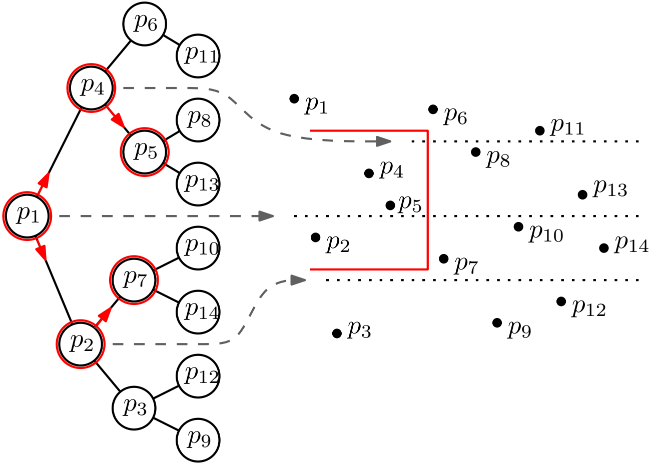
Priority search tree



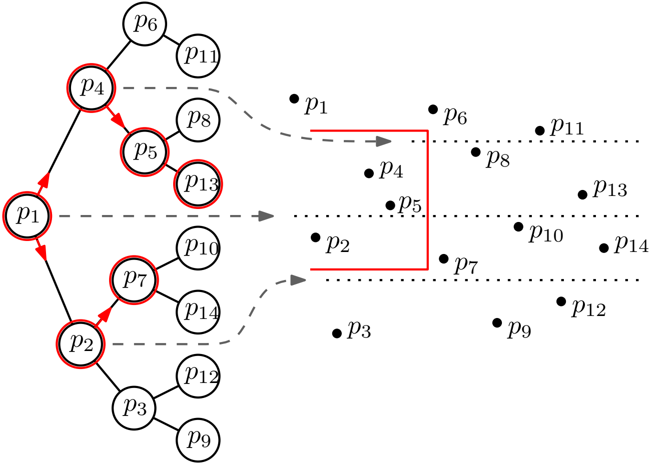
Priority search tree



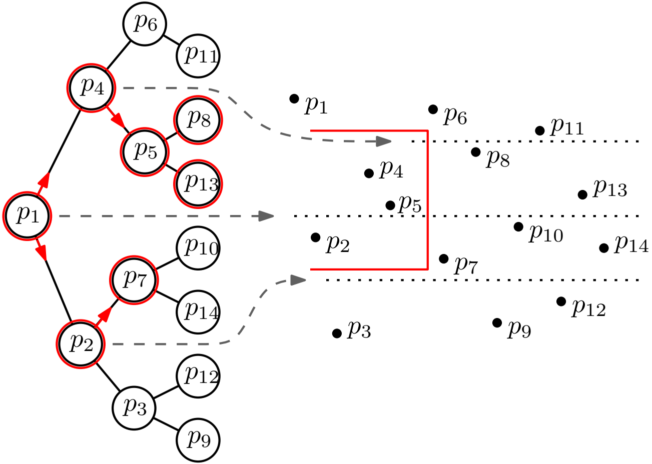
Priority search tree



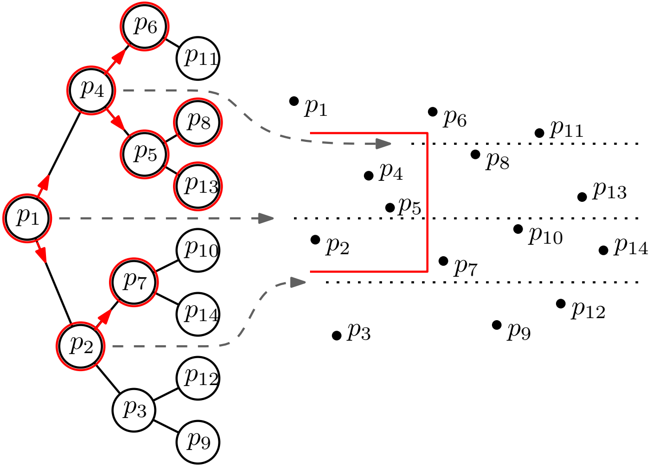
Priority search tree



Priority search tree



Priority search tree

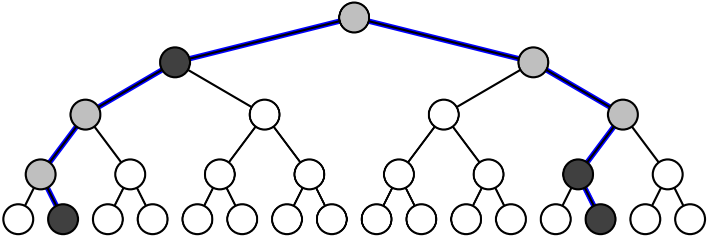


Query algorithm

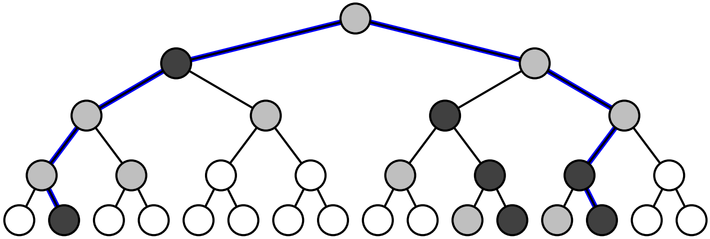
Algorithm $\textsc{QueryPrioSearchTree}(\mathcal{T}, (-\infty : q_x] \times [q_y : q'_y])$

```
Search with q_y and q'_y in T
Let v_split be the node where the two search paths split
for each node v on the search path of q_y or q'_y
    do if p(v) ∈ (−∞ : q_x] × [q_y : q'_y] then report p(v)
for each node v on the path of q_y in the left subtree of v_split
    do if the search path goes left at v
        then ReportInSubtree(rc(v), q_x)
for each node v on the path of q'_y in the right subtree of v_split
    do if the search path goes right at v
        then ReportInSubtree(lc(v), 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

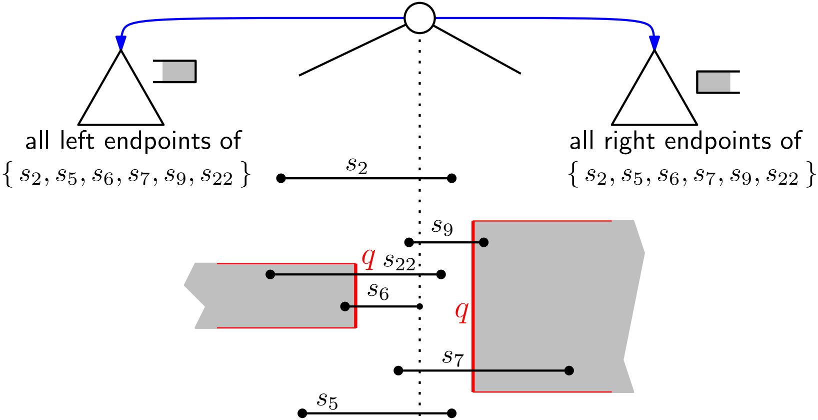
Subtrees of nodes between the search paths are queried like a heap, and we spend $O(1+k')$ time on each one

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

Interesting

Just to confuse you (even more)....

A priority search tree can be used to solve the interval stabbing problem (store 1-dim intervals, query with a point) (!?)

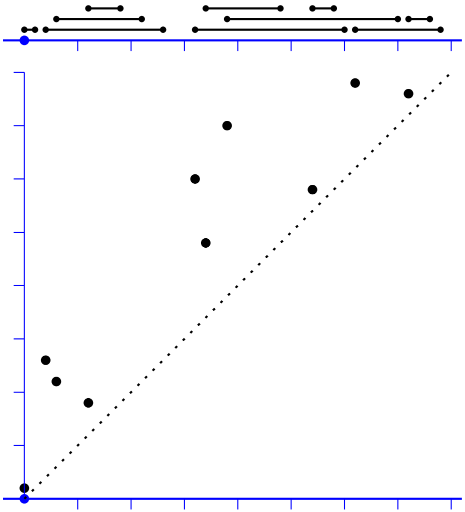
Transformation

Let I be a set of n intervals. Transform each 1-dim interval $[a,b]$ to the point (a,b) in the plane

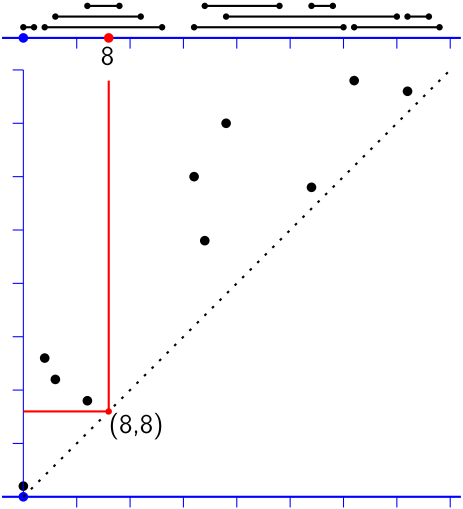
A query with value q is transformed to the 2-sided range $(-\infty,q] \times [q,+\infty)$

Correctness: $q \in [a,b]$ if and only if $(a,b) \in (-\infty,q] \times [q,+\infty)$

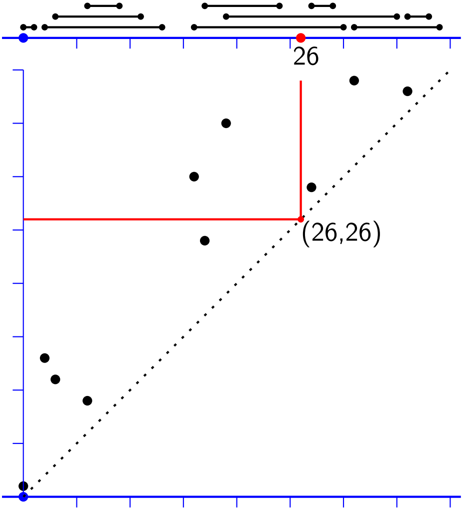
Transformation



Example query



Example query



Food for thought

Question: Can an interval tree be used (after some transformation) to answer 3-sided range queries?

Question: Can the priority search tree be used as the main tree for the structure that queries with a vertical line segment in horizontal line segments?

Question: Can the priority search tree or the interval tree be augmented for interval stabbing *counting* queries?