More Geometric Data Structures

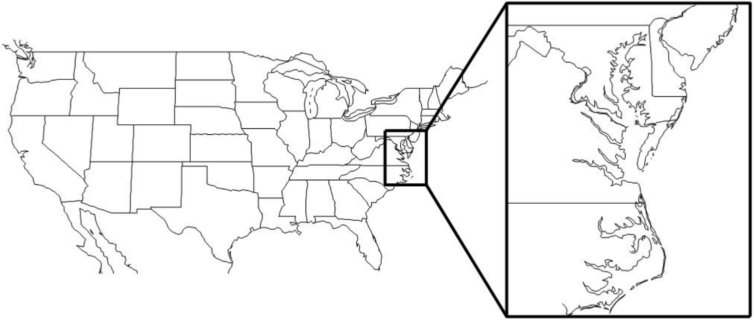
Mehrnoosh Abtahi & Zohreh Akbari

Department of Computer Science
Yazd University

May 13, 2013

Windowing

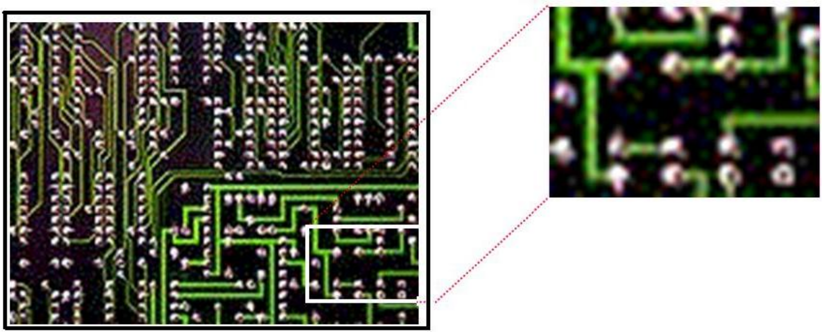
The geographic maps



given a rectangular region, or a *window*, the system must determine the part of the map(roads, cities, and so on) that lie in the window , and display them. This is called a *windowing query*.

Windowing

Design of printed circuit boards

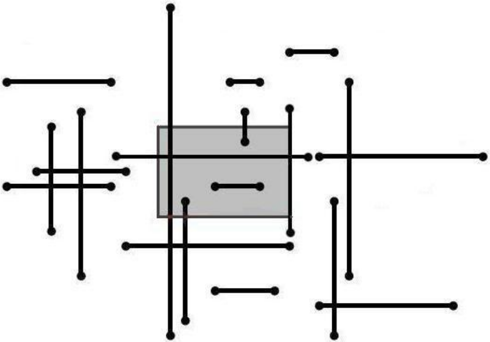


Windowing is required whenever one wants to inspect a small portion of a large, complex object.

Windowing

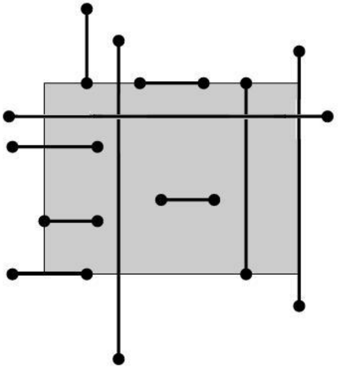
We assume that the query window is an axis-parallel rectangle, that is, a rectangle whose edges are axis-parallel.

Let S be a set of n axis-parallel line segments.



Windowing

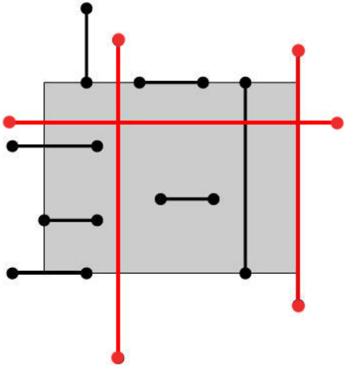
To solve windowing queries we need a data structure that stores S in such a way that the segments intersecting a query window $W{:=}[x:x']\times[y:y']$ can be reported efficiently.



Windowing

What ways a segment can intersect the rectangle ?

- Segments that have at least one endpoint inside the rectangle
- Segments with both endpoints outside the rectangle



Storing segments and searching with a rectangle

Segments with at least one endpoint in the rectangle can be found by building a $2d$ range tree on the $2n$ endpoints.

- Keep pointer from each endpoint stored in tree to the segments
- Mark segments as you output them, so that you don't output contained segments twice.

Lemma

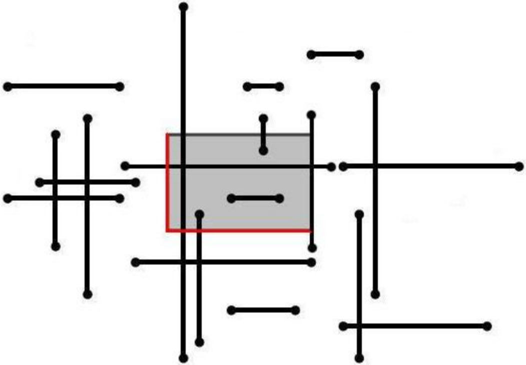
Lemma 10.1

Let S be a set of n axis-parallel line segments in the plane.The segments that have at least one endpoint inside an axis-parallel query window W can be reported in $O(\log n+k)$ time with a data structure that uses $O(n \log n)$ storage and preprocessing time, where k is the number of reported segments.

Storing segments and searching with a rectangle

Segments with both endpoints outside the rectangle :

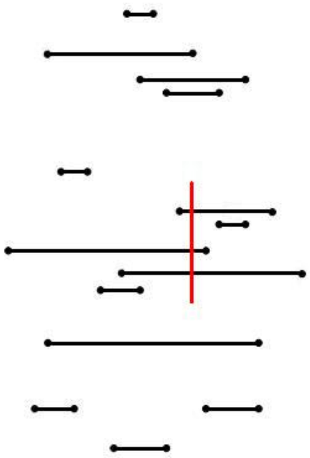
- Store the segments and query with the left side and the bottom side of the rectangle



Current problem:

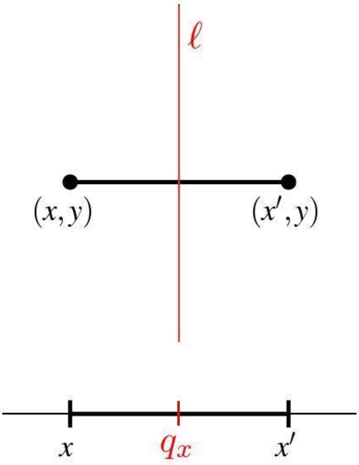
Given a set of horizontal (vertical) line segments, preprocess them into a data structure so that the ones intersecting a vertical (horizontal)query segment can be reported efficiently.

Consider the problem of finding the horizontal segments intersected by the left edge of W.



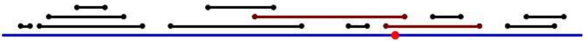
Simpler problem :

- The query segment is a full line.
- $\ell := (x = q_x)$ denote the query line.
- A horizontal segment $s := \overline{(x, y)(x', y)}$ is intersected by ℓ iff $x \leqslant q_x \leqslant x'$
- Then the problem is essentially 1-dimensional.



Interval querying

Given a set of intervals $I := \{[x_1 : x'_1], [x_2 : x'_2], ..., [x_n : x'_n]\}$ on the real line, report the ones that contain the query point q_x.

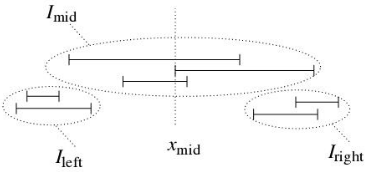


Classification a set of intervals

Let x_{mid} be the median of the $2n$ interval endpoints,

partition the intervals into three subsets :

- Intervals $I_{left} := \{[x_j : x'_j] \in I : x'_j < x_{mid}\}$
- Intervals $I_{mid} := \{[x_j : x'_j] \in I : x_j \leqslant x_{mid} \leqslant x'_j\}$
- Intervals $I_{right} := \{[x_j : x'_j] \in I : x_{mid} < x_j\}$



Construct a binary tree

- If the query value q_x lies to the left of x_{mid} then I_{right} do not contain q_x.
- Or if the query value q_x lies to the right of x_{mid} then I_{left} do not contain q_x.
- we construct a binary tree based on this idea.

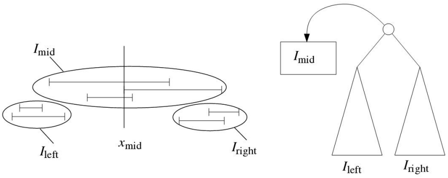
Interval tree

Recursively build subtrees on interval set as follows:

- the intervals I_{left} are stored in the left subtree
- the intervals I_{right} are stored in the right subtree

How should we store I_{mid} ?

we store the set I_{mid} in a separate structure and associate that structure with the root of our tree.

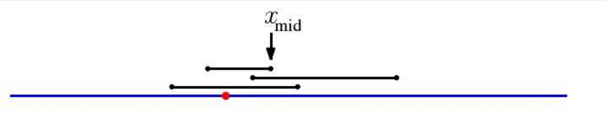


Interval tree : left and right lists

I_{mid} could be the same as I.

But there is a difference ,all the Intervals in I_{mid} contain x_{mid}.

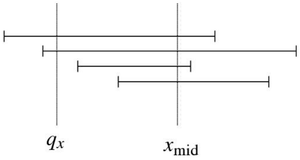
- If the query point(q_x is left of x_{mid}, then only the left endpoint determine if an interval is an answer



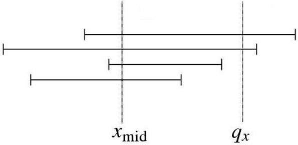
- Symmetrically : If the query point(q_x is right of x_{mid}, then only the right endpoint determine if an interval is an answer

Interval tree : left and right lists

- Make a list $\mathcal{L}_{left}$ sorted on increasing left endpoints of I_{mid}.



- Make a list $\mathcal{L}_{right}$ sorted on decreasing right endpoints of I_{mid}.



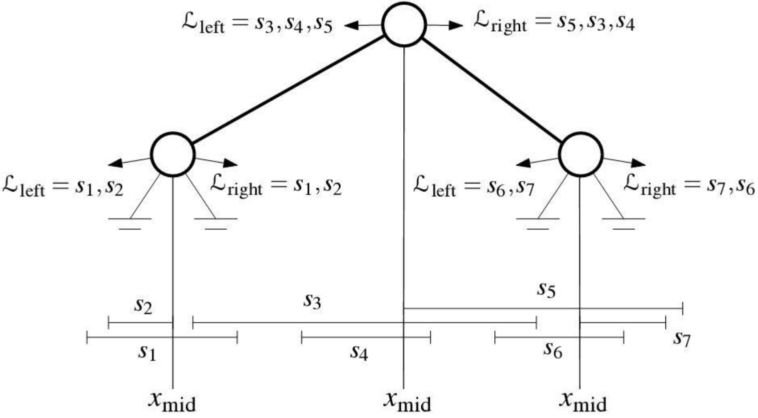
- we can simply walk along the sorted list reporting intervals, until we come to an interval that does not contain q_x.

Interval tree

The interval tree consist of a root node v storing x_{mid}.Furthermore,

- The set I_{mid} is stored twice; once in a list $\mathcal{L}_{left}$, and once in a list $\mathcal{L}_{right}$,
- The left subtree of v is an interval tree for the set I_{left},
- The right subtree of v is an interval tree for the set I_{right}.

Interval tree : example



Interval Tree : storage

Lemma 10.2

An interval tree on a set of n intervals uses $O(n)$ storage and has depth $O(\log n)$.

Proof.

- By choosing the median, we split the set of end points in half each time therefore depth is $O(\log n)$.
- each interval is only stored in a set I_{mid} onec and,hence, only appears once in the two sorted lists.consequently, the interval tree uses $O(n)$ storage.

□

Construct interval tree Algorithm

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 $\mathcal{L}_{\text{left}}(\nu)$ sorted on left endpoint and a list $\mathcal{L}_{\text{right}}(\nu)$ 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

Build time

Lemma 10.3

An interval tree on a set of n intervals can be built in $O(n \log n)$ time.

Build time

Lemma 10.3

An interval tree on a set of n intervals can be built in $O(n \log n)$ time.

Proof.

1. Presorting all of the interval endpoints requires $O(n \log n)$ time.

Build time

Lemma 10.3

An interval tree on a set of n intervals can be built in $O(n \log n)$ time.

Proof.

1. Presorting all of the interval endpoints requires $O(n \log n)$ time.
2. Compute I_{mid},I_{left},I_{right} takes $O(n)$ time.

   Over all $T(n) = O(n)+2T(n/2) = O(n \log n)$.

Build time

Lemma 10.3

An interval tree on a set of n intervals can be built in $O(n \log n)$ time.

Proof.

1. Presorting all of the interval endpoints requires $O(n \log n)$ time.
2. Compute I_{mid},I_{left},I_{right} takes $O(n)$ time.
   Over all $T(n) = O(n)+2T(n/2) = O(n \log n)$.
3. Create $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ takes $O(n_{mid} \log n_{mid})$ time , where $n_{mid}=$ card(I_{mid}). over all take $\sum O(n_{mid} \log n_{mid})$,
   since $\sum n_{mid} = n$, $\sum O(n_{mid} \log n_{mid}) \leqslant O(n \log n)$.

Build time

Lemma 10.3

An interval tree on a set of n intervals can be built in $O(n \log n)$ time.

Proof.

1. Presorting all of the interval endpoints requires $O(n \log n)$ time.
2. Compute I_{mid},I_{left},I_{right} takes $O(n)$ time.
   Over all $T(n) = O(n)+2T(n/2) = O(n \log n)$.
3. Create $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ takes $O(n_{mid} \log n_{mid})$ time , where $n_{mid}=$ card(I_{mid}). over all take $\sum O(n_{mid} \log n_{mid})$,
   since $\sum n_{mid} = n$, $\sum O(n_{mid} \log n_{mid}) \leqslant O(n \log n)$.
4. The total built time therefore becomes $O(n \log n)$.

□

Query Interval Tree Algorithm

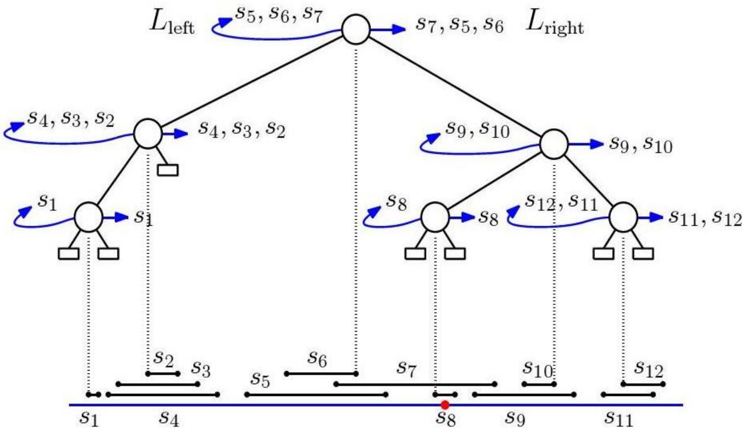
Algorithm QUERYINTERVALTREE(ν, q_x)

Input. The root ν of an interval tree and a query point q_x.

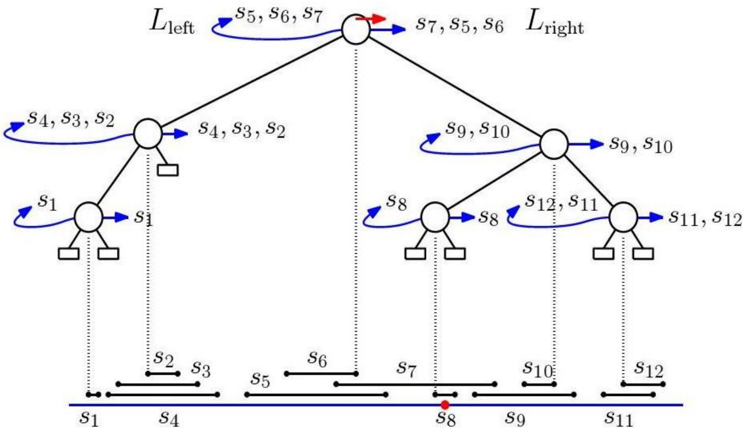
Output. All intervals that contain q_x.

1. **if** ν is not a leaf
2. **then if** $q_x < x_{\text{mid}}(\nu)$
3. **then** Walk along the list $\mathcal{L}_{\text{left}}(\nu)$, 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. QUERYINTERVALTREE($lc(\nu), q_x$)
5. **else** Walk along the list $\mathcal{L}_{\text{right}}(\nu)$, 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. QUERYINTERVALTREE($rc(\nu), q_x$)

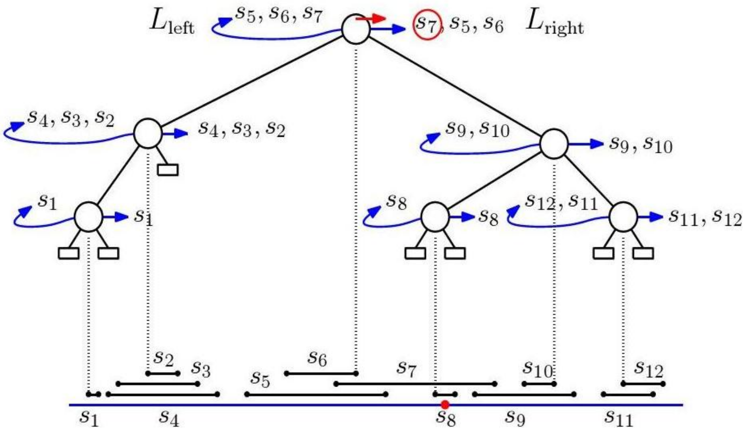
Interval tree:query example



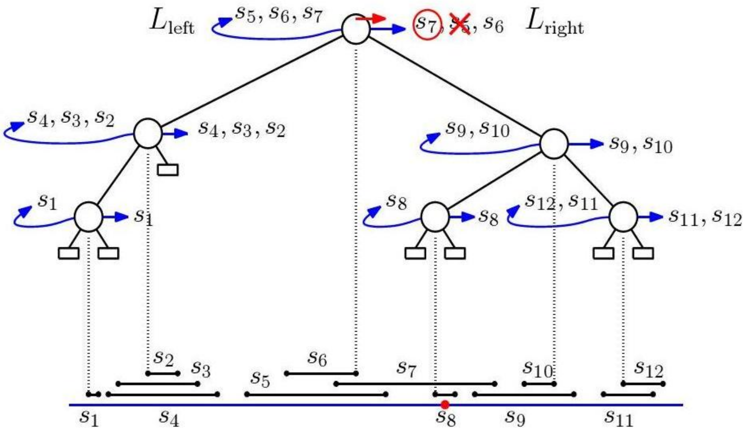
Interval tree:query example



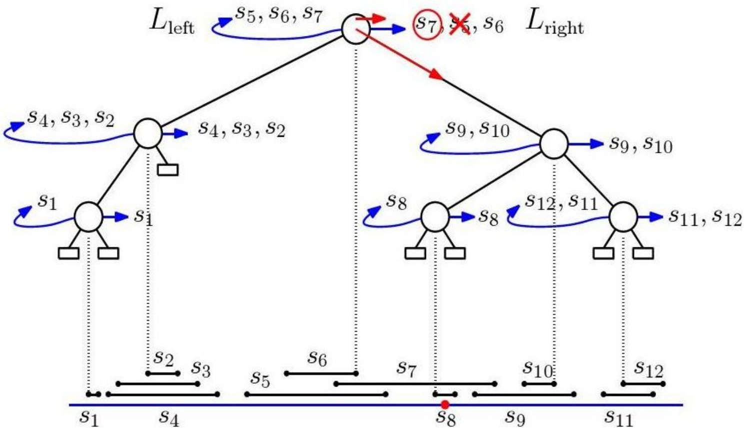
Interval tree:query example



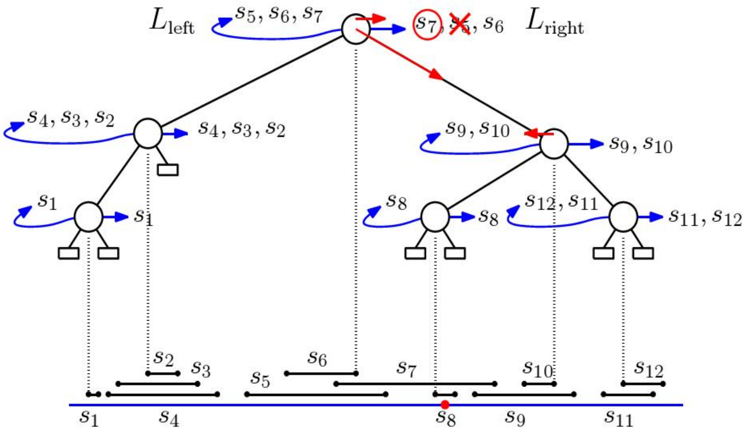
Interval tree:query example



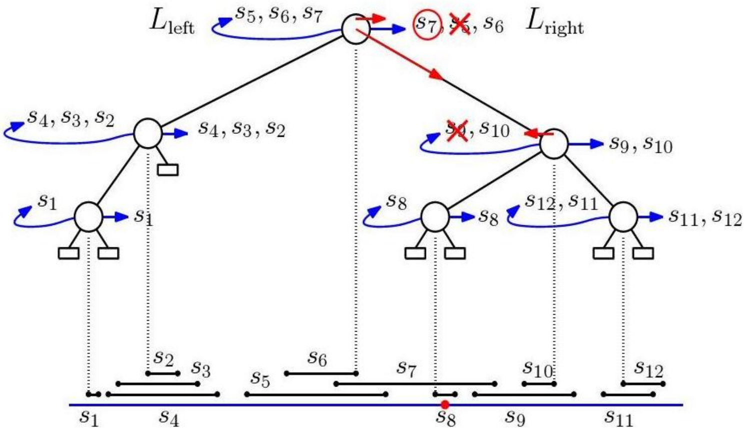
Interval tree:query example



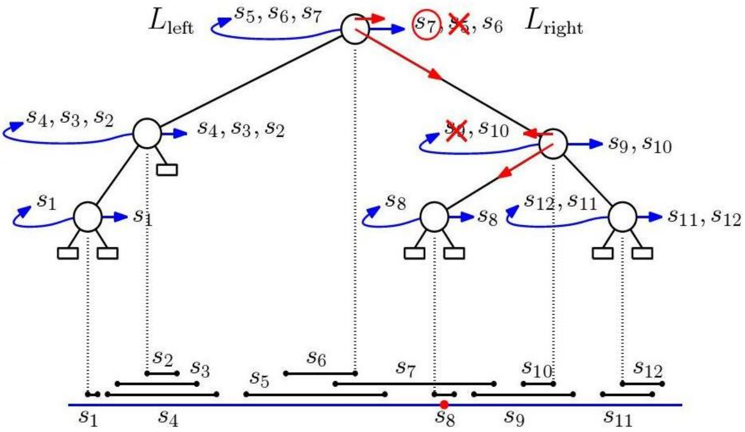
Interval tree:query example



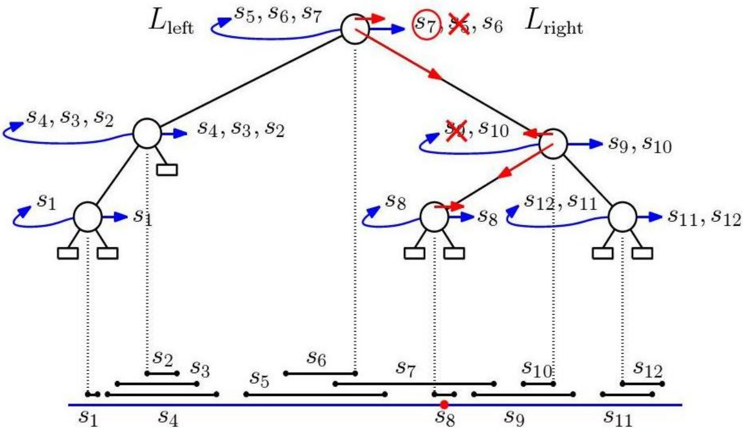
Interval tree:query example



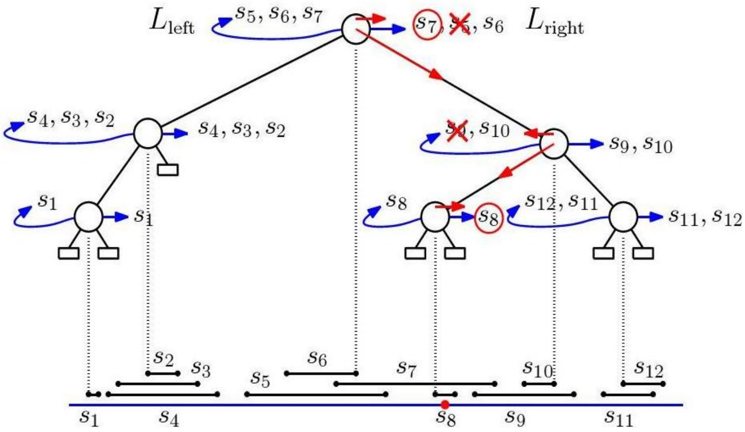
Interval tree:query example



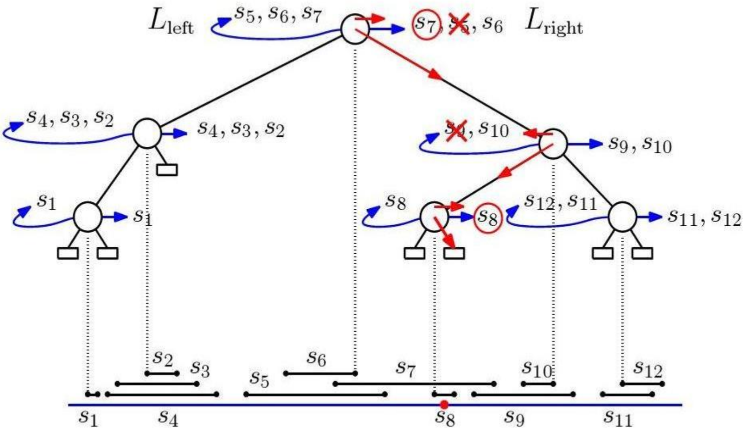
Interval tree:query example



Interval tree:query example



Interval tree:query example



Query time

- The total query time is $O(\log n + k)$, Since
  - At any node v that we visit we spend $O(1+ k_v)$ time,where k_v is the number of intervals that we report at v,
  - $\sum_v k_v = k$,
  - We visit at most one node at any depth of the tree,
  - The depth of the interval tree is $O(\log n)$,
  - So the total query time is $O(\log n + k)$.

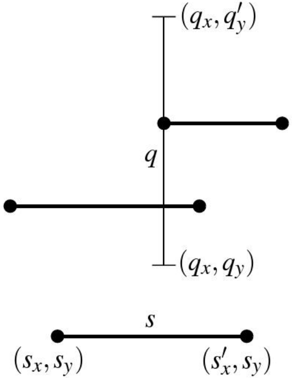
Interval tree : result

Theorem 10.4

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

Vertical Segment Searching

- Let $S_H \subseteq S$ be the subset of horizontal segments in S.
- And q be the vertical query segment $q_x \times [q_y : q'_y]$.
- For a segment $s := [s_x : s'_x] \times s_y$ in S_H, we call $s := [s_x : s'_x]$ the x-interval of the segment.



Vertical Segment Searching

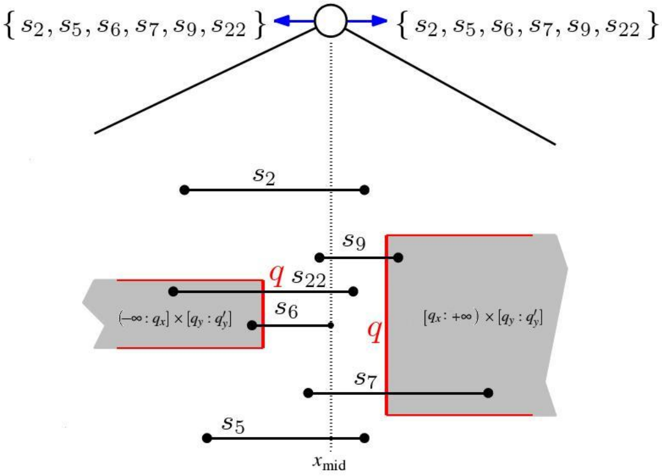
- Suppose we have stored the segments in S_H in an interval tree $\mathcal{T}$ according to their x-interval.

Vertical Segment Searching

- Suppose we have stored the segments in S_H in an interval tree $\mathcal{T}$ according to their x-interval.
- For a segment $s \in I_{mid}$ to be intersected by q , it is not sufficient that its left (right) endpoint lies to the left (right) of q; it is also required that its y-coordinate lies in the range $[q_y : q'_y]$.

Vertical Segment Searching

- Suppose we have stored the segments in S_H in an interval tree $\mathcal{T}$ according to their x-interval.
- For a segment $s \in I_{mid}$ to be intersected by q , it is not sufficient that its left (right) endpoint lies to the left (right) of q; it is also required that its y-coordinate lies in the range $[q_y : q'_y]$.
- Then the lists $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ are not suitable anymore to solve the query problem for the segments corresponding to I_{mid}.


$\{ s_2, s_5, s_6, s_7, s_9, s_{22} \}$
$\{ s_2, s_5, s_6, s_7, s_9, s_{22} \}$
s_2
s_9
q
s_{22}
$(-\infty : q_x] \times [q_y : q'_y]$
s_6
q
$[q_x : +\infty) \times [q_y : q'_y]$
s_7
s_5
x_{mid}

Vertical Segment Queries

- We need a assosiated structure.

Vertical Segment Queries

- We need a assosiated structure.
- The main structure is an interval tree $\mathcal{T}$ on the x-interval of the segments.

Vertical Segment Queries

- We need a assosiated structure.
- The main structure is an interval tree $\mathcal{T}$ on the x-interval of the segments.
- Instead of the sorted lists we have two range tree as the associated structure.

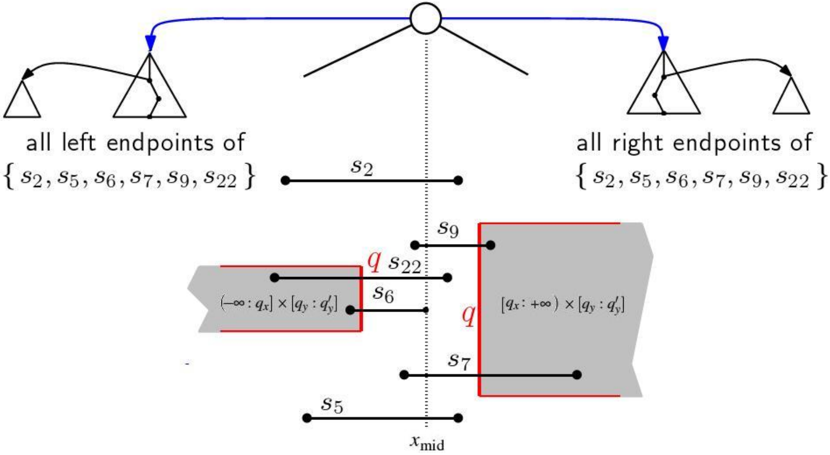
Vertical Segment Queries

- We need a assosiated structure.
- The main structure is an interval tree $\mathcal{T}$ on the x-interval of the segments.
- Instead of the sorted lists we have two range tree as the associated structure.
- A range tree $\mathcal{T}_{left}(v)$ on the left endpoints of the segments in $I_{mid}(v)$, and a range tree $\mathcal{T}_{right}(v)$ on the right endpoints of the segments in $I_{mid}(v)$.

Vertical Segment Queries

- We need a assosiated structure.
- The main structure is an interval tree $\mathcal{T}$ on the x-interval of the segments.
- Instead of the sorted lists we have two range tree as the associated structure.
- A range tree $\mathcal{T}_{left}(v)$ on the left endpoints of the segments in $I_{mid}(v)$, and a range tree $\mathcal{T}_{right}(v)$ on the right endpoints of the segments in $I_{mid}(v)$.
- Instead of traversing $\mathcal{L}_{left}$ or $\mathcal{L}_{right}$, we perform a query in the range tree $\mathcal{T}_{left}$ or $\mathcal{T}_{right}$.

Vertical Segment Queries



Vertical Segment Queries: Storage and query time

- The total amount of storage for the data structure becomes $O(n \log n)$, Since
  - The total amount of storage for a rangr tree is $O(n_v \log n_v)$,
  - $\sum n_v = n$,
  - $\sum O(n_v \log n_v) \leqslant O(n \log n)$.

Vertical Segment Queries: Storage and query time

- The total amount of storage for the data structure becomes $O(n \log n)$, Since
  - The total amount of storage for a rangr tree is $O(n_v \log n_v)$,
  - $\sum n_v = n$,
  - $\sum O(n_v \log n_v) \leqslant O(n \log n)$.

- The total query time becomes $O(\log^2 n + k)$,Since
  - There are $O(\log n)$ nodes v on the search path,
  - At each node v have to do an $O(\log n + k)$ search on a range tree (assuming your range trees use fractional cascading),
  - The total query time therefore becomes $O(\log^2 n + k)$.

Result

Theorem 10.5

Let S be a set of n horizontal segments in the plane. The segments intersecting a vertical query segment can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time.

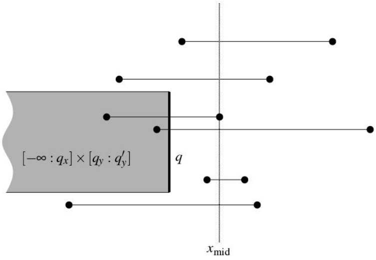
Priority search trees reduce the storage to $O(n)$.

Corollary 10.6

Let S be a set of n axis-parallel segments in the plane. The segments intersecting a axis-parallel rectangular query window can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time.

Property : queries are unbounded on one side

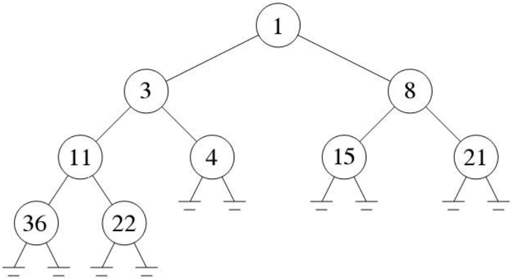
Using priority search tree, that uses this property, instead of range trees in the data structure for windowing reduces the storage bound in Theorem 10.5 to $O(n)$.



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.

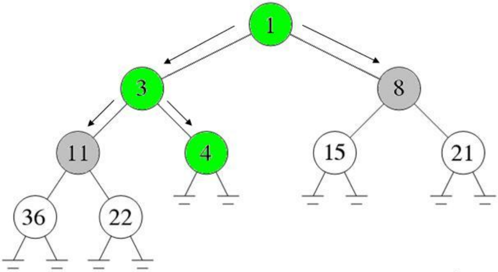
Recall the heap :



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 :



Example query : $(-\infty : 5]$

Report All values $\leqslant$ 5

A heap has the query time $O(1+k)$.

Priority search tree

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane.

Priority search tree

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane.

If $P = \emptyset$ then the priority search tree is an empty leaf.otherwise,let

Priority search tree

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane.

If $P = \emptyset$ then the priority search tree is an empty leaf.otherwise,let

- p_{min} := point with the smallest x-coordinate,
- y_{mid} := median of y-coordinates of points in $P - \{p_{min}\}$,
- $P_{below} := \{p \in P - \{p_{min}\} : p_y < y_{mid}\}$,
- $P_{above} := \{p \in P - \{p_{min}\} : p_y > y_{mid}\}$

Priority search tree

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane.

If $P = \emptyset$ then the priority search tree is an empty leaf.otherwise,let

- p_{min} := point with the smallest x-coordinate,
- y_{mid} := median of y-coordinates of points in $P - \{p_{min}\}$,
- $P_{below} := \{p \in P - \{p_{min}\} : p_y < y_{mid}\}$,
- $P_{above} := \{p \in P - \{p_{min}\} : p_y > y_{mid}\}$

The priority search tree has a root node v where the point p_{min} and the value y_{mid} are stored.

Priority search tree

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane.

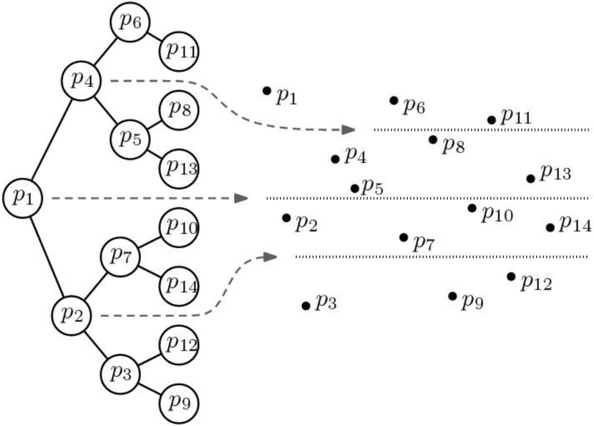
If $P = \emptyset$ then the priority search tree is an empty leaf.otherwise,let

- p_{min} := point with the smallest x-coordinate,
- y_{mid} := median of y-coordinates of points in $P - \{p_{min}\}$,
- $P_{below} := \{p \in P - \{p_{min}\} : p_y < y_{mid}\}$,
- $P_{above} := \{p \in P - \{p_{min}\} : p_y > y_{mid}\}$

The priority search tree has a root node v where the point p_{min} and the value y_{mid} are stored.

The left subtree of v is a priority search tree for the set P_{below} and right subtree of v is a priority search tree for the set P_{above}.

Priority search tree

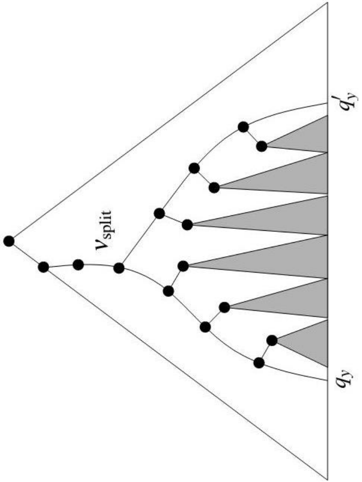


Priority search tree can be built in $O(n \log n)$ time.

Querying a priority search tree

A query with a range $(-\infty : q_x] \times [q_y : q'_y]$ in a PST :

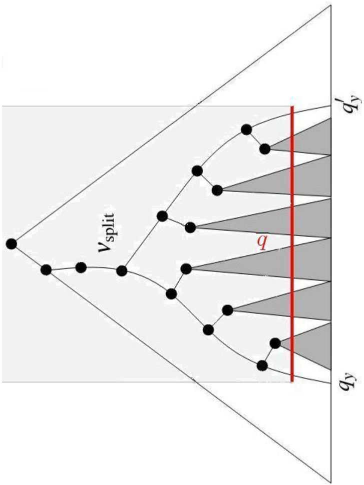
- First, we find all the points that lie in $[q_y : q'_y]$(shaded subtrees).
- Then, we search those subtrees based on x-coordinate only (heap on x-coordinate).
- Also, must check each node along both paths because they store points.



Querying a priority search tree

A query with a range $(-\infty : q_x] \times [q_y : q'_y]$ in a PST :

- First, we find all the points that lie in $[q_y : q'_y]$(shaded subtrees).
- Then, we search those subtrees based on x-coordinate only (heap on x-coordinate).
- Also, must check each node along both paths because they store points.



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 $(p(\nu))_x \leqslant q_x$
2. **then** Report $p(\nu)$.
3. REPORTINSUBTREE($lc(\nu), q_x$)
4. REPORTINSUBTREE($rc(\nu), q_x$)

Lemma 10.7

ReportInSubTree (v, q_x) reports in $O(1+k_v)$ time all points in the subtree rooted at v whose x-coordinate is at most q_x, where k_v is the number of reported points.

Lemma 10.7

ReportInSubTree (v, q_x) reports in $O(1+k_v)$ time all points in the subtree rooted at v whose x-coordinate is at most q_x, where k_v is the number of reported points.

Proof.

- All points with x-coordinate at most q_x are reported.
- All points that are reported have x-coordinate at most q_x.

Lemma 10.7

REPORTINSUBTREE (v, q_x) reports in $O(1+k_v)$ time all points in the subtree rooted at v whose x-coordinate is at most q_x, where k_v is the number of reported points.

Proof.

- All points with x-coordinate at most q_x are reported.
- All points that are reported have x-coordinate at most q_x.
- The query time for a subtree is like query time for a heap, namely $O(1+k_v)$.

$\square$

Algorithm QUERYPRIOSEARCHTREE($\mathcal{T}, (-\infty : q_x] \times [q_y : q'_y]$)
Input. A priority search tree and a range, unbounded to the left.
Output. All points lying in the range.

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

Lemma 10.8

Algorithm QUERYPRIOSEARCHTREE reports the points in a query range $(-\infty : q_x] \times [q_y : q'_y]$ in $O(\log n + k)$ time, where k is the number of reported points.

Lemma 10.8

Algorithm QUERYPRIOSEARCHTREE reports the points in a query range $(-\infty : q_x] \times [q_y : q'_y]$ in $O(\log n + k)$ time, where k is the number of reported points.

Proof.

- Any point that is reported by the algorithm lies in the query range.
- Any point that lies in the range is reported by the algorithm.

Lemma 10.8

Algorithm QUERYPRIOSEARCHTREE reports the points in a query range $(-\infty : q_x] \times [q_y : q'_y]$ in $O(\log n + k)$ time, where k is the number of reported points.

Proof.

- Any point that is reported by the algorithm lies in the query range.
- Any point that lies in the range is reported by the algorithm.
- The search paths to q_y and q'_y have $O(\log n)$ nodes. At each node $O(1)$ time is spent.
- The time taken by all executions of REPORTINSUBTREE is $O(\log n + k)$.
- The total query time is $O(\log n + k)$.

□

Priority search tree: result

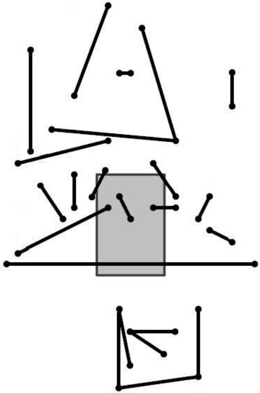
Theorem 10.9

A priority search tree for a set P of n points in the plane uses $O(n)$ storage and can report all points in a query range of the form $(-\infty : q_x] \times [q_y : q'_y]$ in $O(\log n + k)$ time, where k is the number of reported points.

Arbitrarily oriented segments

Two cases of intersection:

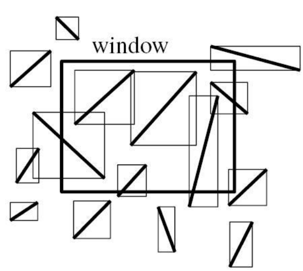
- An endpoint lies inside the query window; solve with range trees
- The segment intersects the window boundary; solve how?



Arbitrarily Oriented Segments

A simple solution:

Replace each line segment by its bounding box.

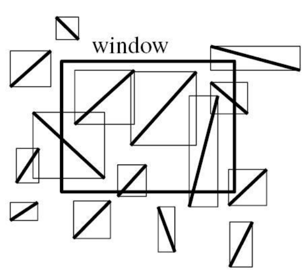


So we could search in the $4n$ bounding box sides.

Arbitrarily Oriented Segments

A simple solution:

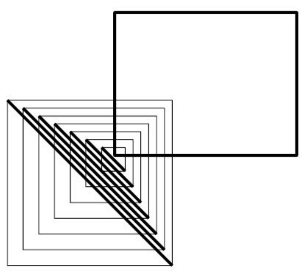
Replace each line segment by its bounding box.



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

In the worst case:

The solution is quite bad:

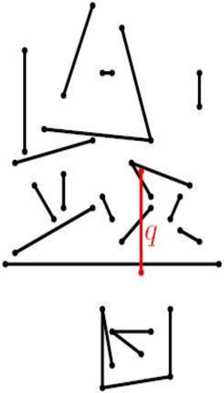


All bounding boxes may intersect W whereas none of the segments do.

Current problem:

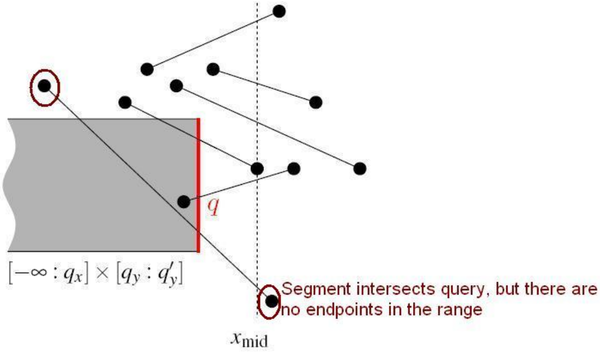
Current problem of our intesect:

Given a set S of line segments with arbitrary orientations in the plane, and we want to find those segments in S that intersect a vertical query segment $q := q_x \times [q_y : q'_y]$.

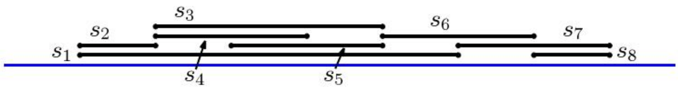


Why don't interval trees work ?

If the segments have arbitrary orientation, knowing that the right endpoint of a segment is to the right of q doesn't help us much.



Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of n segments(Intervals) 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**.

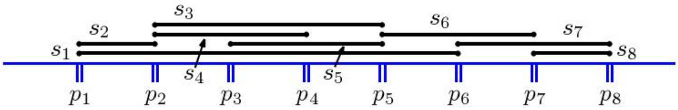
Locus approach

The locus approach is the idea to partition the parameter space into regions where the answer to a query is the same.

Locus approach

The locus approach is the idea to partition the parameter space into regions where the answer to a query is the same.

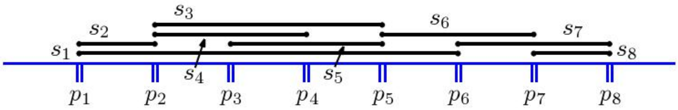
Our query has only one parameter, q_x, so the parameter space is the real line. Let $p_1, p_2, ..., p_m$ be the list of distinct interval endpoints, sorted from left to right; $m \leqslant 2n$



Locus approach

The locus approach is the idea to partition the parameter space into regions where the answer to a query is the same.

Our query has only one parameter, q_x, so the parameter space is the real line. Let $p_1, p_2, ..., p_m$ be the list of distinct interval endpoints, sorted from left to right; $m \leqslant 2n$



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.

Locus approach

- We could make a binary search tree that has a leaf for every elementary interval.
- We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.

Locus approach

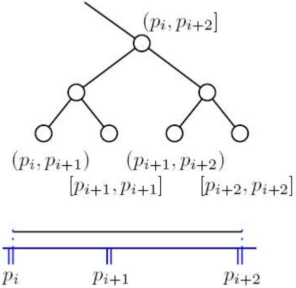
- We could make a binary search tree that has a leaf for every elementary interval.
- We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.
- all the segments (intervals)in S containing $Int(\mu)$ are stored at the leaf μ

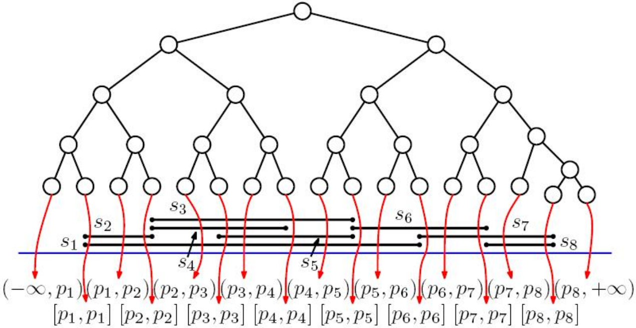
Locus approach

- We could make a binary search tree that has a leaf for every elementary interval.
- We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.
- all the segments (intervals)in S containing $Int(\mu)$ are stored at the leaf μ
- each internal node corresponds to an interval that is the union of the elementary intervals of all leaves below it

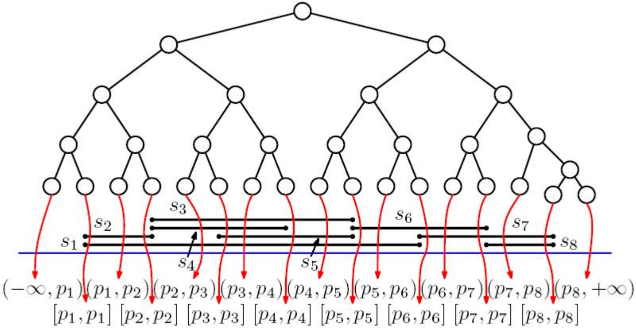
Locus approach

- We could make a binary search tree that has a leaf for every elementary interval.
- We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.
- all the segments (intervals)in S containing $Int(\mu)$ are stored at the leaf μ
- each internal node corresponds to an interval that is the union of the elementary intervals of all leaves below it



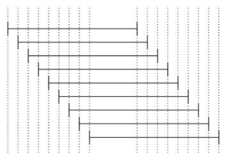


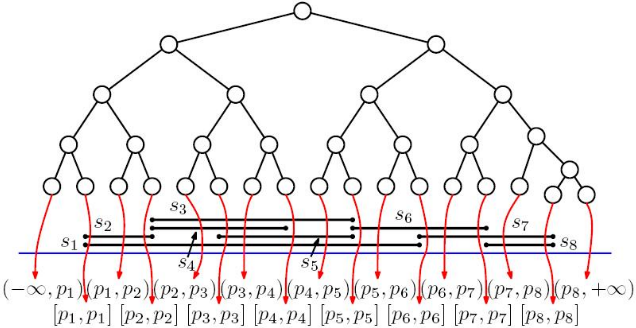
s_1
s_2
s_3
s_4
s_5
s_6
s_7
s_8
$(-\infty, p_1)$ (p_1, p_2) (p_2, p_3) (p_3, p_4) (p_4, p_5) (p_5, p_6) (p_6, p_7) (p_7, p_8) $(p_8, +\infty)$
$[p_1, p_1]$ $[p_2, p_2]$ $[p_3, p_3]$ $[p_4, p_4]$ $[p_5, p_5]$ $[p_6, p_6]$ $[p_7, p_7]$ $[p_8, p_8]$



Storage

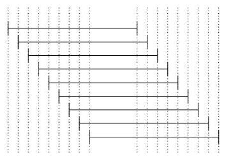
$O(n^2)$ storage in the worst case:





Storage

$O(n^2)$ storage in the worst case:

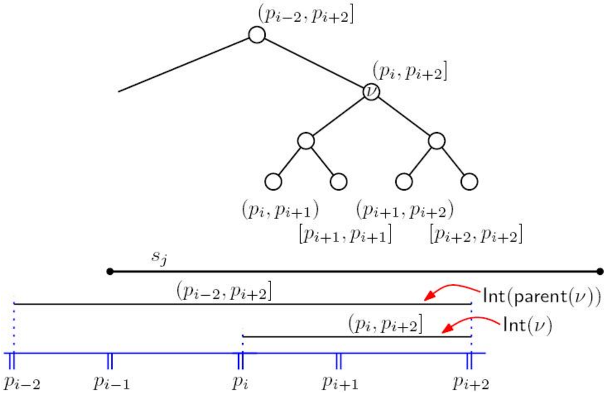


Query time

We can report the k intervals containing q_x in $O(\log n + k)$ time.

Reduce the amount of storage

To avoid quadratic storage, we store any segment s_j with v iff $Int(v) \subseteq s_j$ but $Int(parent(v)) \not\subseteq s_j$.



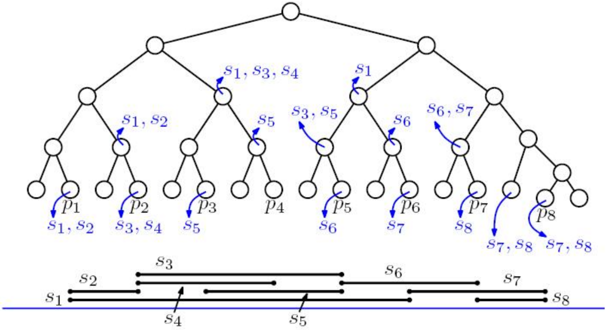
The data structure based on this principle is called a segment tree.

Segment tree

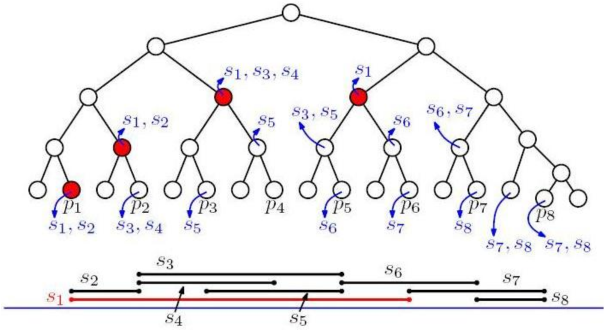
A segment tree on a set S of segments is a balanced binary search tree on the elementary intervals defined by S, and each node stores its interval, and its canonical subset of S in a list.

The canonical subset of a node v contains segments s_j such that $Int(v) \subseteq s_j$ but $Int(parent(v)) \nsubseteq s_j$

Segment tree



Segment tree

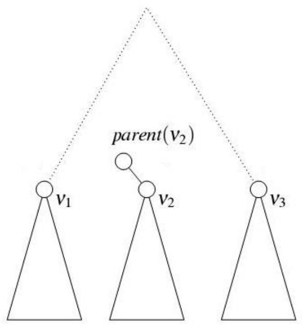


Lemma 10.10

A segment tree on a set of n intervals uses $O(n \log n)$ storage.

Proof.

We claim that any segment is stored for at most two nodes at the same depth of the tree.

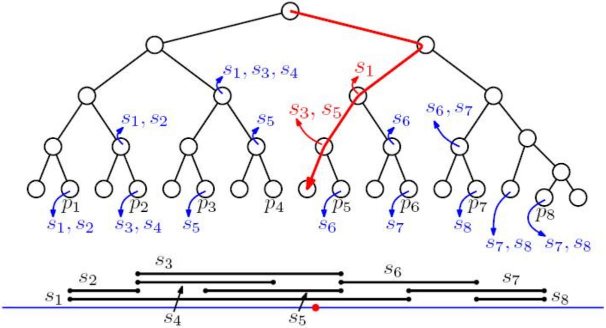


□

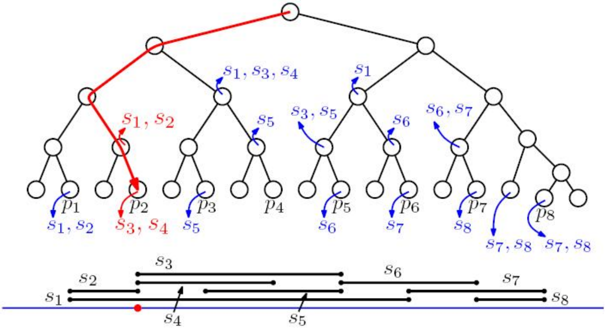
Query algorithm

Algorithm QUERYSEGMENTTREE(ν, q_x)
Input. The root of a (subtree of a) segment tree and a query point q_x.
Output. All intervals in the tree containing q_x.
1. Report all the intervals in $I(\nu)$.
2. **if** ν is not a leaf
3. **then if** $q_x \in \mathrm{Int}(lc(\nu))$
4. **then** QUERYSEGMENTTREE($lc(\nu), q_x$)
5. **else** QUERYSEGMENTTREE($rc(\nu), q_x$)

Example query



Example query



Lemma 10.11

Using a segment tree, the intervals containing a query point q_x can be reported in $O(\log n + k)$ time, where k is the number of reported intervals.

Segment Tree Construction

- Build tree :

Segment Tree Construction

- Build tree :
  - Sort the endpoints of the segments take $O(n \log n)$ time.This give us the elementary intervals.
  - Construct a balanced binary tree on the elementary intervals,this can be done bottom-up in $O(n)$ time.

Segment Tree Construction

- Build tree :
  - Sort the endpoints of the segments take $O(n \log n)$ time.This give us the elementary intervals.
  - Construct a balanced binary tree on the elementary intervals,this can be done bottom-up in $O(n)$ time.
- Compute the canonical subset for the nodes.To this end we insert the intervals one by one into the segment tree by calling :

Segment Tree Construction

- Build tree :
  - Sort the endpoints of the segments take $O(n \log n)$ time. This give us the elementary intervals.
  - Construct a balanced binary tree on the elementary intervals, this can be done bottom-up in $O(n)$ time.
- Compute the canonical subset for the nodes. To this end we insert the intervals one by one into the segment tree by calling :

Algorithm INSERTSEGMENTTREE($\nu, [x : x']$)
Input. The root of a (subtree of a) segment tree and an interval.
Output. The interval will be stored in the subtree.
1. **if** Int(ν) $\subseteq [x : x']$
2.   **then** store $[x : x']$ at ν
3.   **else** **if** Int($lc(\nu)$) $\cap [x : x'] \neq \emptyset$
4.   **then** INSERTSEGMENTTREE($lc(\nu), [x : x']$)
5.   **if** Int($rc(\nu)$) $\cap [x : x'] \neq \emptyset$
6.   **then** INSERTSEGMENTTREE($rc(\nu), [x : x']$)

How much time does it take to insert an interval $[x : x']$ into the segment tree?

- an interval is stored at most twice at each level of $\mathcal{T}$
- There is also at most one node at every level whose corresponding interval contains x and one node whose interval contains x'.
- So we visit at most 4 nodes per level.
- Hence, the time to insert a single interval is $O(\log n)$, and the total time to construct the segment tree is $O(n \log n)$.

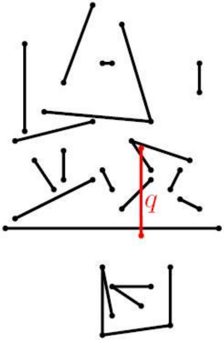
Result

Theorem 10.12

A segment tree for a set I of n intervals uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time. Using the segment tree we can report all intervals that contain a query point in $O(\log n + k)$ time, where k is the number of reported intervals.

Back to windowing problem

Let S be a set of arbitrarily oriented, disjoint segments in the plane. We want to report the segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$

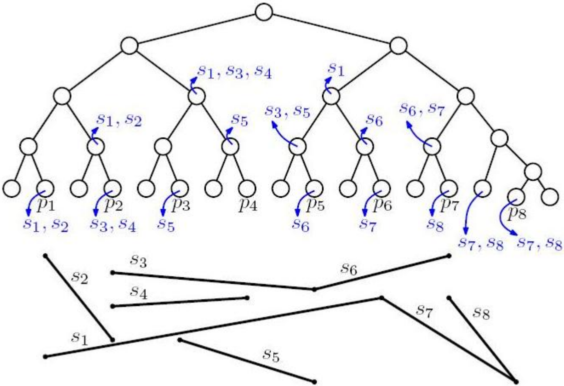


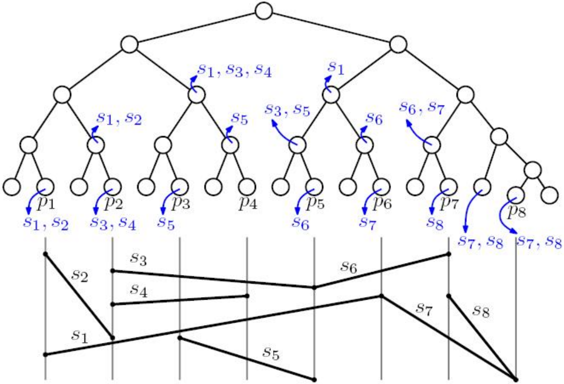
- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.

- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.
- A node v in $\mathcal{T}$ can now be considered to correspond to the vertical slab $Int(v) \times (-\infty : +\infty)$.

- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.
- A node v in $\mathcal{T}$ can now be considered to correspond to the vertical slab $Int(v) \times (-\infty : +\infty)$.
- A segment s_i is in the canonical subset of v, if it crosses the slab of v completely, but not the slab of the parent of v.

- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.
- A node v in $\mathcal{T}$ can now be considered to correspond to the vertical slab $Int(v) \times (-\infty : +\infty)$.
- A segment s_i is in the canonical subset of v, if it crosses the slab of v completely, but not the slab of the parent of v.
- We denote canonical subset of v with $S(v)$.

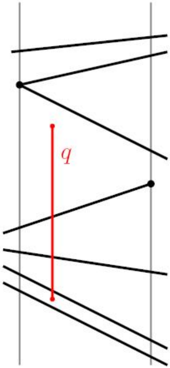

s_1, s_3, s_4
s_1
s_1, s_2
s_5
s_3, s_5
s_6
s_6, s_7
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
s_1, s_2
s_3, s_4
s_5
s_6
s_7
s_8
s_7, s_8
s_7, s_8
s_3
s_2
s_6
s_4
s_7
s_8
s_1
s_5



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

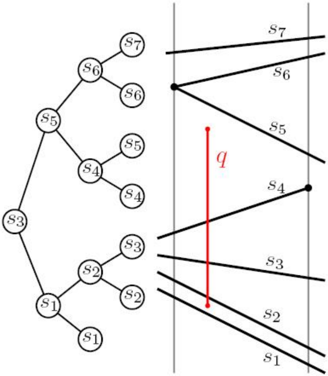
Querying

- When we search with q_x in $\mathcal{T}$ we find $O(\log n)$ canonical subsets that collectively contain all the segments whose x-interval contains q_x.
- A segment s in such a canonical subset is intersected by q if and only if the lower endpoint of q is below s and the upper endpointof q is above s.



Querying

- segments in the canonical subset $S(v)$ do not intersect each other. This implies that the segments can be ordered vertically.
- we can store $S(v)$ in a search tree $\mathcal{T}(v)$ according to the vertical order.



Query time

- A query with q_x follows one path down the main tree(segment tree)
- And at every node v on the search path we search with endpoints of q in $\mathcal{T}(v)$ to report the segments in $S(v)$ intersected by q (a 1-dimensional range query).
- The search in $\mathcal{T}(v)$ takes $O(\log n + k_v)$ time,wherek_v is the number of reported segments at (v).
- Hence, the total query time is $O(\log^2 n + k)$.

Storage

- Because the associated structure of any node v uses storage linear in the size of $S(v)$, the total amount of storage remains $O(n \log n)$.

- Data structure can be build in $O(n \log n)$ time.

Result

Theorem 10.13

Let S be a set of n disjoint segments in the plane. The segments intersecting a vertical query segment can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time.

Result

Theorem 10.13

Let S be a set of n disjoint segments in the plane. The segments intersecting a vertical query segment can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time.

Corollary 10.14

Let S be a set of n segments in the plane with disjoint interiors. The segments intersecting an axis-parallel rectangular query window can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time

END