Computational Geometry

Lecture 14: Windowing queries
Zoom in; re-center and zoom in; select by outlining
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.
How can a rectangle and an axis-parallel line segment intersect?
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)
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.
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 $\text{FALSE}$

When we think we should report a segment, we first check its mark bit:
- if $\text{FALSE}$, then report it and set the mark bit to $\text{TRUE}$
- otherwise, don’t do anything

After a query, we need to reset all mark bits to $\text{FALSE}$, for the next query (how?)
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*
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?
**Simpler query problem:**

What if the vertical query segment is a full line?

Then the problem is essentially 1-dimensional.
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.
The median $x$ of the $2n$ endpoints partitions the intervals into three subsets:

- Intervals $I_{\text{left}}$ fully left of $x$
- Intervals $I_{\text{mid}}$ that contain (intersect) $x$
- Intervals $I_{\text{right}}$ fully right of $x$
The interval tree for \( I \) has a root node \( v \) that contains \( x \) and

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

The left and right subtrees are proper interval trees for \( I_{\text{left}} \) and \( I_{\text{right}} \)

How many intervals can be in \( I_{\text{mid}} \)? How should we store \( I_{\text{mid}} \)?
Interval tree: left and right lists

How is $I_{\text{mid}}$ 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 $L_{\text{left}}$ using the left-to-right order of the left endpoints of $I_{\text{mid}}$

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

Store both lists as associated structures with $\nu$
Interval tree: example
The main tree has $O(n)$ nodes.

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

Consequently, the interval tree uses $O(n)$ storage.
**Algorithm** \( \text{QUERY\_INTERVAL\_TREE}(\nu, q_x) \)

1. if \( \nu \) is not a leaf
2. \hspace{1em} then if \( q_x < \text{x_{mid}}(\nu) \)
3. \hspace{2em} 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. \hspace{1em} \text{QUERY\_INTERVAL\_TREE}(\text{lc}(\nu), q_x) \\
5. \hspace{1em} 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. \hspace{1em} \text{QUERY\_INTERVAL\_TREE}(\text{rc}(\nu), q_x)
Interval tree: query example

Motivation
- Interval trees
- Priority search trees

Definition
- Querying
- Construction

Interval tree: query example

L_left
- s_5, s_6, s_7

L_right
- s_7, s_5, s_6

s_4, s_3, s_2
- s_4, s_3, s_2

s_1
- s_1

s_2
- s_2

s_3
- s_3

s_4
- s_4

s_5
- s_5

s_6
- s_6

s_7
- s_7

s_8
- s_8

s_9
- s_9

s_10
- s_10

s_11
- s_11

s_12
- s_12
Interval trees

Priority search trees

Definition

Querying

Construction

Interval tree: query example
Motivation
Interval trees
Priority search trees
Definition
Querying
Construction

Interval tree: query example

$L_{left}$ $s_5, s_6, s_7$ $L_{right}$

$s_4, s_3, s_2$

$s_1$

$s_4$

$s_4, s_3, s_2$

$s_9, s_{10}$

$s_8$

$s_8, s_{12}, s_{11}$

$s_{11}, s_{12}$

$s_1$ $s_4$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$
Motivation
Interval trees
Priority search trees

Definition
Querying
Construction

Interval tree: query example
Motivation

Interval trees

Priority search trees

Definition

Querying

Construction

Interval tree: query example

$L_{\text{left}}$ $s_5, s_6, s_7$ $L_{\text{right}}$

$s_7, s_6$ $s_6$

$s_4, s_3, s_2$ $s_4, s_3, s_2$

$s_1$ $s_4$

$s_2$ $s_3$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$

$s_1$ $s_4$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$

Computational Geometry  Lecture 14: Windowing queries
Interval tree: query example

Motivation
Interval trees
Priority search trees
Definition
Querying
Construction

Computation Geometry  Lecture 14: Windowing queries
Interval tree: query example
Interval tree: query example

![Diagram of an interval tree with nodes labeled and intervals depicted.]
Interval tree: query example

- **Motivation**
- **Interval trees**
- **Priority search trees**

**Definition**

**Querying**

**Construction**

- **Interval tree:** query example

\[ L_{\text{left}} = s_5, s_6, s_7 \]
\[ s_7, \text{x}, s_6 \]
\[ L_{\text{right}} \]

\[ s_4, s_3, s_2 \]

\[ s_1 \]

- **s_1**
- **s_4**
- **s_3**
- **s_5**
- **s_6**
- **s_7**
- **s_8**
- **s_9**
- **s_10**
- **s_11**
- **s_12**

- **Left branch:**
  - \( s_1, s_4, s_3, s_2 \)
  - \( s_1 \)
  - \( s_4, s_3, s_2 \)
  - \( s_1 \)

- **Right branch:**
  - \( s_8, s_9, s_{10}, s_{11}, s_{12} \)
  - \( s_8 \)
  - \( s_{12}, s_{11} \)
  - \( s_8 \)

- **Node labels:**
  - \( s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12} \)

- **Windowing queries**

**Computational Geometry**

Lecture 14: Windowing queries
Interval tree: query example

- **Motivation**
  - Interval trees
  - Priority search trees

- **Definition**
  - Querying
  - Construction

- **Computational Geometry**
  - Lecture 14: Windowing queries
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 $O(\log n + k)$, with the total number of answers reported (no answer is found more than once)

The query time is $O(\log n) + O(\log n + k) = O(\log n + k)$
Algorithm $\text{ConstructIntervalTree}(I)$

*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 $\nu$. Compute $x_{\text{mid}}$, the median of the set of interval endpoints, and store $x_{\text{mid}}$ with $\nu$
4. Compute $I_{\text{mid}}$ and construct two sorted lists for $I_{\text{mid}}$: a list $L_{\text{left}}(\nu)$ sorted on left endpoint and a list $L_{\text{right}}(\nu)$ sorted on right endpoint. Store these two lists at $\nu$
5. $lc(\nu) \leftarrow \text{ConstructIntervalTree}(I_{\text{left}})$
6. $rc(\nu) \leftarrow \text{ConstructIntervalTree}(I_{\text{right}})$
7. return $\nu$
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.
Suppose we use an interval tree on the $x$-intervals of the horizontal line segments?

Then the lists $L_{\text{left}}$ and $L_{\text{right}}$ are not suitable anymore to solve the query problem for the segments corresponding to $I_{\text{mid}}$. 
Motivation
Interval trees
Priority search trees

Back to the plane

Computational Geometry Lecture 14: Windowing queries
Computational Geometry
Lecture 14: Windowing queries
Back to the plane
Back to the plane

Motivation
Interval trees
Priority search trees

$q$

$s_5, s_6, s_7$

$s_7, s_5, s_6$

$s_6$

$s_5, s_7$

$s_7$

Computational Geometry
Lecture 14: Windowing queries
Motivation
Interval trees
Priority search trees
Back to the plane

\{ s_2, s_5, s_6, s_7, s_9, s_{22} \} \quad \leftrightarrow \quad \{ s_2, s_5, s_6, s_7, s_9, s_{22} \}

Computational Geometry
Lecture 14: Windowing queries
Back to the plane

\[ \{ s_2, s_5, s_6, s_7, s_9, s_{22} \} \leftrightarrow \{ s_2, s_5, s_6, s_7, s_9, s_{22} \} \]
Back to the plane

\{ s_2, s_5, s_6, s_7, s_9, s_{22} \} 

\{ s_2, s_5, s_6, s_7, s_9, s_{22} \}
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 $L_{\text{left}}$ and $L_{\text{right}}$

Instead of traversing $L_{\text{left}}$ or $L_{\text{right}}$, we perform a query with the region left or right, respectively, of $q$
Segment intersection queries

all endpoints of \{ s_2, s_5, s_6, s_7, s_9, s_{22} \}

\[ q \]

Motivation
Interval trees
Priority search trees
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

all left endpoints of \( \{ s_2, s_5, s_6, s_7, s_9, s_{22} \} \)

all right endpoints of \( \{ s_2, s_5, s_6, s_7, s_9, s_{22} \} \)
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:

```
1
/   \
/     \  
2       4
/ \\     /   \\
3 7 5 6
/ \  \\
9 12 10 8
```

14 13 11
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:

```
  1
 /\ \
2  4
 / \ /  \
3  7 5  6
 /  \ /  \\
9 12 14 10 13 8 11
```

Report all values $\leq 4$
If $P = \emptyset$, then a priority search tree is an empty leaf.

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

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

Computational Geometry
Lecture 14: Windowing queries
Priority search trees

Diagram of a priority search tree with labeled nodes and arrows indicating the structure and hierarchy of the tree.
Priority search tree

Motivation
Interval trees
Priority search trees
Definition
Querying

Computational Geometry  Lecture 14: Windowing queries
Motivation

Interval trees

Priority search trees

Definition

Querying

Priority search tree

Computational Geometry  Lecture 14: Windowing queries
Priority search tree
Priority search tree
Priority search tree
Priority search tree

Computational Geometry Lecture 14: Windowing queries
Algorithm \textsc{QueryPrioSearchTree}(\mathcal{T}, (-\infty : q_x] \times [q_y : q_y'))

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

*Input.* The root $\nu$ 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** $\nu$ 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
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.
**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

all left endpoints of \{s_2, s_5, s_6, s_7, s_9, s_{22}\}

all right endpoints of \{s_2, s_5, s_6, s_7, s_9, s_{22}\}

Left endpoints:  
- s_2
- s_5
- s_6
- s_{22}

Right endpoints:  
- s_2
- s_5
- s_6
- s_7
- s_{22}

Query point q

Motivation
Interval trees
Priority search trees

Computational Geometry Lecture 14: Windowing queries
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?
**Question:** What is the query time of the structure for querying with a vertical segment in a set of horizontal segments?
**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.
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.
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 $(a, b) \in (-\infty, q] \times [q, +\infty)$.
Example query

Motivation
Interval trees
Priority search trees

Computational Geometry Lecture 14: Windowing queries
Example query

(26,26)
**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?
Computational Geometry

Lecture 15: Windowing queries
Zoom in; re-center and zoom in; select by outlining
Motivation
Segment trees
Windowing again

Windowing

Windowing queries
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.
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.
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?
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.
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?

\[
\text{Motivation} \quad \text{Segment trees} \quad \text{Windowing again} \quad \text{Windowing queries}
\]
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.
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*.
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.
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
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

Motivation
Segment trees
Windowing again

Definition
Querying
Storage

Towards segment trees

[p2, p4]

(p1, p2]

(p6, +∞)

(p1, p2]

(p2, p4]
Towards segment trees

Let $\text{Int}(\nu)$ denote the interval of node $\nu$

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 $\nu$ if and only if

$\text{Int}(\nu) \subseteq s_j$ but $\text{Int}(\text{parent}(\nu)) \not\subseteq s_j$
Towards segment trees

\[(p_{i-2}, p_{i+2}]\]

\[(p_i, p_{i+2}]\]

\[(p_i, p_{i+1})\] \[(p_{i+1}, p_{i+2})\]

\([p_{i+1}, p_{i+1}]\) \([p_{i+2}, p_{i+2}]\)

\(s_j\)

Computational Geometry  Lecture 15: Windowing queries
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 $v$ is the subset of segments $s_j$ for which $\text{Int}(v) \subseteq s_j$ but $\text{Int}(\text{parent}(v)) \not\subseteq s_j$.
Segment trees

Motivation
Segment trees
Windowing again

Definition
Querying
Storage

Computational Geometry
Lecture 15: Windowing queries
Segment trees

Motivation

Segment trees

Windowing again

Definition

Querying

Storage

Computational Geometry

Lecture 15: Windowing queries
**Question:** Why are no segments stored with nodes on the leftmost and rightmost paths of the segment tree?
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

[Diagram of a segment tree with nodes labeled s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8 and edges labeled p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8.]

Motivation
Segment trees
Windowing again

Definition
Querying
Storage

Computational Geometry
Lecture 15: Windowing queries
The query time is $O(\log n + k)$, where $k$ is the number of segments reported.
A segment can be stored in several lists of nodes. How bad can the storage requirements get?
Lemma: Any segment can be stored at up to two nodes of the same depth.

Proof: Suppose a segment $s_i$ is stored at three nodes $v_1$, $v_2$, and $v_3$ at the same depth from the root.
If a segment tree has depth $O(\log n)$, then any segment is stored in at most $O(\log n)$ lists $\Rightarrow$ the total size of all lists is $O(n\log n)$

The main tree uses $O(n)$ storage

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

Note the correspondence with 2-dimensional range trees
**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.
Question: Do you see how to adapt the segment tree so that stabbing \textit{counting} queries can be answered efficiently?
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.
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.
Computational Geometry | Lecture 15: Windowing queries
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 $\nu$, then it crosses the slab of $\nu$ completely, but not the slab of the parent of $\nu$
- the segments crossing a slab have a well-defined top-to-bottom order

$s_j$ is stored at one or more nodes below $\nu$
Motivation
Segment trees
Windowing again

Segment tree variation
Querying
Storage

Computational Geometry Lecture 15: Windowing queries
Motivation
Segment trees
Windowing again

Segment tree variation
Querying
Storage

Computational Geometry
Lecture 15: Windowing queries
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 \( q \)?
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.
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 $O(\log^2 n + k)$. 
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 $O(n \log n)$. 


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.