Outline:

I. General windowing queries

II. Inapplicability of an interval tree

III. Structure of a segment tree

IV. Query and construction

V. Solution of a general windowing query

VI. Analysis
I. Solving a Windowing Query over Horizontal Segments

Interval tree + range trees

Query object: vertical segment $q_x \times [q_y, q'_y]$

Set: $n$ horizontal line segments
Interval Tree + Priority Search Trees

Applicable to axis-parallel segments only.

Query object: vertical segment $q_x \times [q_y, q_y']$

Set: $n$ horizontal line segments

\[ \text{IT} \]

\[ y_{mid} \]

\[ I_{mid} \]

\[ I_{left} \]

\[ I_{right} \]

\[ \text{PST: min heap over the } x\text{-coordinate (of left endpoint) while a BST over the } y\text{-coordinate} \]

\[ \text{PST: max heap over the } x\text{-coordinate (of right endpoint) while a BST over the } y\text{-coordinate} \]
General Windowing Queries

Line segments can have arbitrary orientations.
General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?
General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?

- Replace each segment with its bounding box.
General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?

- Replace each segment with its bounding box.
- Find all bounding boxes intersecting the query window.
- Check the segments defining these boxes.
General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?

- Replace each segment with its bounding box.
- Find all bounding boxes intersecting the query window.
- Check the segments defining these boxes.
- Works well generally in practice.
General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?

- Replace each segment with its bounding box.
- Find all bounding boxes intersecting the query window.
- Check the segments defining these boxes.

♦ Works well generally in practice.
♠ Worst case can be very bad.

No segment intersects \( W \) even though all the bounding boxes do!
General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?

- Replace each segment with its bounding box.
- Find all bounding boxes intersecting the query window.
- Check the segments defining these boxes.

- Works well generally in practice.
- Worst case can be very bad.

- No segment intersects $W$ even though all the bounding boxes do!
- No guarantee on a fast query time.
II. Thinking Top-Down

♦ Segments with $\geq 1$ endpoints in $W$. 

II. Thinking Top-Down

♦ Segments with $\geq 1$ endpoints in $W$.

Range trees.
II. Thinking Top-Down

- Segments with $\geq 1$ endpoints in $W$.

  Range trees.

- Segments intersecting the window boundary.

  One intersection query with each of the 4 boundary edges.
II. Thinking Top-Down

♦ Segments with $\geq 1$ endpoints in $W$.
  
  Range trees.

♦ Segments intersecting the window boundary.
  
  One intersection query with each of the 4 boundary edges.
  Focus on a vertical boundary edge.
II. Thinking Top-Down

- Segments with $\geq 1$ endpoints in $W$.
  
  Range trees.

- Segments intersecting the window boundary.
  
  One intersection query with each of the 4 boundary edges.
  
  Focus on a vertical boundary edge.

Query problem

Query object: a vertical line segment $q: q_x \times [q_y, q_y']$
II. Thinking Top-Down

- Segments with $\geq 1$ endpoints in $W$.
  
  Range trees.

- Segments intersecting the window boundary.
  
  One intersection query with each of the 4 boundary edges.
  
  Focus on a vertical boundary edge.

Query problem

Query object: a vertical line segment $q: q_x \times [q_y, q'_y]$ 

Set: $S = \{s_1, s_2, ..., s_n\}$ of $n$ segments
  
  - arbitrarily oriented
  - non-intersecting in the interior
  - possibly sharing endpoints
Inapplicability of an Interval Tree

An interval tree is not very helpful.
Inapplicability of an Interval Tree

An interval tree is not very helpful.

Search with $q_x$
Inapplicability of an Interval Tree

An interval tree is not very helpful.

Search with $q_x$ $\rightarrow$ $I_{\text{mid}}(v)$ at node $v$

at node $v$
Inapplicability of an Interval Tree

An interval tree is not very helpful.

Search with $q_x$ \implies \text{ at node } v
\text{ mid (v)dat node } v
\text{ Checking if left endpoint } \in (-\infty, q_x] \times [q_y, q_y']
\text{ (for a horizontal segment).}

$x \text{ mid } v > q_x$
$x xx \text{ mid d mid (v)} > q_x$
Inapplicability of an Interval Tree

An interval tree is not very helpful.

Search with $q_x$ \implies\text{I mid (v) at node } v 
\implies\text{Checking if left endpoint } \in (-\infty, q_x] \times [q_y, q_y'] 
(for a horizontal segment).

$x \text{ mid } v > q_x$ 
$x x \text{ mid } d \text{ mid (v)} > q_x$

No such reduction to range query when the segment is arbitrarily oriented
Inapplicability of an Interval Tree

An interval tree is not very helpful.

Search with $q_x$  \[\rightarrow\]  $I_{mid}(v)$ at node $v$

at node $v$

\[\rightarrow\]  Checking if left endpoint $\in (-\infty, q_x] \times [q_y, q'_y]$

(for a horizontal segment).

No such reduction to range query when the segment is arbitrarily oriented.
An interval tree is not very helpful.

Search with $q_x$ at node $v$

- Checking if left endpoint $\in (-\infty, q_x] \times [q_y, q_y']$ (for a horizontal segment).

No such reduction to range query when the segment is arbitrarily oriented.

Segment $s_1 \in I_{\text{mid}}(v)$ intersects $q$ but its left endpoint $\notin (-\infty, q_x] \times [q_y, q_y']$. 
Inapplicability of an Interval Tree

An interval tree is not very helpful.

Search with \( q_x \) at node \( v \)

\[ x_{\text{mid}}(v) > q_x \]

Checking if left endpoint \( \in (-\infty, q_x] \times [q_y, q_y'] \)

(for a horizontal segment).

No such reduction to range query when the segment is arbitrarily oriented.

Segment \( s_1 \in I_{\text{mid}}(v) \) intersects \( q \) but its left endpoint \( \notin (-\infty, q_x] \times [q_y, q_y'] \).

Segment \( s_3 \in I_{\text{mid}}(v) \) and its left endpoint \( \in (-\infty, q_x] \times [q_y, q_y'] \).

But it has no intersection with \( q \).
Locus Approach

Window $W: [q_x, q'_x] \times [q_y, q'_y]$ defined by four parameters.

**Idea:** Partition the parameter space into regions such that queries in the region have the same answer.
Locus Approach

Window $W: [q_x, q'_x] \times [q_y, q'_y]$ defined by four parameters.

Idea: Partition the parameter space into regions such that queries in the region have the same answer.

1D Query

Input: interval set: $I = \{[x_1, x'_1], [x_2, x'_2], \ldots, [x_n, x'_n]\}$

point $q_x$

Output: all intervals containing $q_x$
Locus Approach

Window $W$: $[q_x, q'_x] \times [q_y, q'_y]$ defined by four parameters.

Idea: Partition the parameter space into regions such that queries in the region have the same answer.

1D Query

Input: interval set: $I = \{[x_1, x'_1], [x_2, x'_2], \ldots, [x_n, x'_n]\}$

point $q_x$

Output: all intervals containing $q_x$

$p_1, p_2, \ldots, p_m$: distinct interval endpoints in the increasing order.

\[ \begin{array}{ccc}
p_1 & p_2 & p_{m-1} & p_m \\
\end{array} \]
Locus Approach

Window $W$: $[q_x, q'_x] \times [q_y, q'_y]$ defined by four parameters.

**Idea:** Partition the parameter space into regions such that queries in the region have the same answer.

1D Query

**Input:** interval set: $I = \{[x_1, x'_1], [x_2, x'_2], \ldots, [x_n, x'_n]\}$

point $q_x$

**Output:** all intervals containing $q_x$

$p_1, p_2, \ldots, p_m$: distinct interval endpoints in the increasing order.

The parameter space $(-\infty, \infty)$ is partitioned into elementary intervals.

$(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, (p_{m-1}, p_m), [p_m, p_m], (p_m, \infty)$
Locus Approach

Window $W: [q_x, q_x'] \times [q_y, q_y']$ defined by four parameters.

Idea: Partition the parameter space into regions such that queries in the region have the same answer.

1D Query

Input: interval set: $I = \{[x_1, x_1'], [x_2, x_2'], \ldots, [x_n, x_n']\}$

point $q_x$

Output: all intervals containing $q_x$

$p_1, p_2, \ldots, p_m$: distinct interval endpoints in the increasing order.

The parameter space $(-\infty, \infty)$ is partitioned into elementary intervals.

$(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, (p_{m-1}, p_m), [p_m, p_m], (p_m, \infty)$

- Every open interval has two consecutive endpoints.
- Every closed interval consists of a single endpoint.
- Open intervals alternate with closed intervals.
Using a Binary Search Tree?

\[ \mu : \text{a leaf} \]
\[ \text{Int}(\mu) : \text{elementary interval corresponding to } \mu. \]

\[ (-\infty, p_1) \quad [p_1, p_1] \quad \cdots \quad (p_i, p_{i+1}) \quad \cdots \quad [p_m, p_m] \quad (p_m, \infty) \]
Using a Binary Search Tree?

\[ \mu : \text{a leaf} \]

\[ \text{Int}(\mu) : \text{elementary interval corresponding to } \mu. \]

\[ \blacklozenge \text{ Store at the leaf } \mu \text{ all the intervals in } I \text{ that contain } \text{Int}(\mu). \]

\[ (\infty, p_1) \quad [p_1, p_1] \quad \ldots \quad (p_i, p_{i+1}) \quad \ldots \quad [p_m, p_m] \quad (p_m, \infty) \]
Using a Binary Search Tree?

\( \mu \): a leaf
Int(\( \mu \)): elementary interval corresponding to \( \mu \).

- Store at the leaf \( \mu \) all the intervals in \( I \) that contain \text{Int}(\( \mu \)).

- Query time \( O(\log n + k) \)

\((-\infty, p_1), [p_1, p_1], \ldots, (p_i, p_{i+1}), \ldots, [p_m, p_m], (p_m, \infty)\)
Using a Binary Search Tree?

\( \mu \) : a leaf
\( \text{Int}(\mu) \): elementary interval corresponding to \( \mu \).

- Store at the leaf \( \mu \) all the intervals in \( I \) that contain \( \text{Int}(\mu) \).

- Query time

\[ O(\log n + k) \]
Using a Binary Search Tree?

- Store at the leaf $\mu$ all the intervals in $I$ that contain $\text{Int}(\mu)$.

- Query time

\[ O(\log n + k) \]

- BST search

- #reported intervals

- High storage if the intervals overlap a lot.

$\mu$: a leaf

$\text{Int}(\mu)$: elementary interval corresponding to $\mu$. 
Using a Binary Search Tree?

\[ \mu : \text{a leaf} \]
\[ \text{Int}(\mu) : \text{elementary interval corresponding to } \mu. \]

\[ \blacklozenge \text{ Store at the leaf } \mu \text{ all the intervals in } I \text{ that contain } \text{Int}(\mu). \]

\[ \blacklozenge \text{ Query time } \quad O(\log n + k) \]

\[ \text{BST search} \quad \#\text{reported intervals} \]

\[ \blacklozenge \text{ High storage if the intervals overlap a lot.} \]

\[ O(n^2) \text{ possible!} \]
Store an Interval As High as Possible

Interval \( s \) is stored five times at \( \mu_1, \mu_2, \mu_3, \mu_4, \mu_5 \).
Store an Interval As High as Possible

Interval $s$ is stored five times at $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5$. 
Store an Interval As High as Possible

Interval \( s \) is stored five times at \( \mu_1, \mu_2, \mu_3, \mu_4, \mu_5 \).

**Observation** A search path ends at \( \mu_1, \mu_2, \mu_3, \mu_4 \) if and only if it passes through \( v \).
Interval $s$ is stored five times at $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5$.

**Observation** A search path ends at $\mu_1, \mu_2, \mu_3, \mu_4$ if and only if it passes through $v$.

Why not store $s$ only two times at $v$ and $\mu_5$?
Store an Interval As High as Possible

Interval $s$ is stored five times at $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5$.

**Observation** A search path ends at $\mu_1, \mu_2, \mu_3, \mu_4$ if and only if it passes through $v$.

Why not store $s$ only two times at $v$ and $\mu_5$?

**Idea:** Store a segment $s$ at the *fewest nodes* whose corresponding intervals form a partitioning of $s$. 

---

$\mu_1$  $\mu_2$  $\mu_3$  $\mu_4$  $\mu_5$

$S$
Store an Interval As High as Possible

Interval $s$ is stored five times at $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5$.

**Observation** A search path ends at $\mu_1, \mu_2, \mu_3, \mu_4$ if and only if it passes through $v$.

Why not store $s$ only two times at $v$ and $\mu_5$?

**Idea**: Store a segment $s$ at the fewest nodes whose corresponding intervals form a partitioning of $s$.

These nodes must be as high as possible in the tree.
III. Segment Tree
III. Segment Tree
III. Segment Tree
III. Segment Tree

\[
\begin{align*}
&(-\infty, \infty) \\
&(-\infty, p_4) \\
&(-\infty, p_2) \\
&(-\infty, p_1) \\
&(p_1, p_2) \\
&(p_2, p_3) \\
&(p_3, p_4) \\
&(p_4, p_5) \\
&(p_5, p_6) \\
&(p_6, p_7) \\
&(p_7, \infty)
\end{align*}
\]

\[
\begin{align*}
&\{s_1\} \\
&\{s_2, s_5\} \\
&\{s_1\} \\
&\{s_1\} \\
&\{s_2, s_5\} \\
&\{s_5\} \\
&\{s_3\} \\
&\{s_3, s_4, s_5\}
\end{align*}
\]
III. Segment Tree
III. Segment Tree
Tree Structure

- Leaves ↔ elementary intervals (left-to-right)
Tree Structure

- Leaves $\leftrightarrow$ elementary intervals (left-to-right)
Tree Structure

- Leaves ↔ elementary intervals (left-to-right)
- Internal node $v$ ↔ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at $v$. 

![Tree Structure Diagram]

- $s_1$: $[p_1, p_2]$  
- $s_2$: $[p_2, p_3]$  
- $s_3$: $[p_4, p_5]$  
- $s_4$: $[p_6, p_7]$  
- $s_5$: $[p_8, p_9]$
Tree Structure

- Leaves $\leftrightarrow$ elementary intervals (left-to-right)
- Internal node $v$ $\leftrightarrow$ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at $v$.
- At every node or leaf $v$ stores $\text{Int}(v)$ and the *canonical subset* (as a linked list) defined as
Tree Structure

- Leaves $\leftrightarrow$ elementary intervals (left-to-right)
- Internal node $v$ $\leftrightarrow$ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at $v$.
- At every node or leaf $v$ stores $\text{Int}(v)$ and the canonical subset (as a linked list) defined as

$$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int(parent}(v)) \not\subseteq [x, x']\}$$
Tree Structure

- Leaves ↔ elementary intervals (left-to-right)
- Internal node \( v \) ↔ union \( \text{Int}(v) \) of elementary intervals at the leaves in the subtree \( \mathcal{T}(v) \) rooted at \( v \).
- At every node or leaf \( v \) stores \( \text{Int}(v) \) and the *canonical subset* (as a linked list) defined as
  \[
  C(v) = \{ [x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \not\subseteq [x, x'] \}
  \]

Store intervals at nodes as high as possible.
Tree Structure

- Leaves $\leftrightarrow$ elementary intervals (left-to-right)
- Internal node $v \leftrightarrow$ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at $v$.
- At every node or leaf $v$ stores $\text{Int}(v)$ and the *canonical subset* (as a linked list) defined as

$$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \not\subseteq [x, x']\}$$

Store intervals at nodes as high as possible.

**Examples**

$\text{Int}(v) = [p_2, p_4] \subseteq s_2, s_5$
Tree Structure

- Leaves ↔ elementary intervals (left-to-right)
- Internal node $v$ ↔ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at $v$.
- At every node or leaf $v$ stores $\text{Int}(v)$ and the canonical subset (as a linked list) defined as:
  \[
  C(v) = \{ [x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int(parent}(v)) \not\subseteq [x, x'] \}
  \]

Store intervals at nodes as high as possible.

Examples
- $\text{Int}(v) = [p_2, p_4] \subseteq s_2, s_5$
- $\text{Int(parent}(v)) = (-\infty, p_4] \not\subseteq s_2, s_5$
Leaves $\leftrightarrow$ elementary intervals (left-to-right)

Internal node $v$ $\leftrightarrow$ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at $v$.

At every node or leaf $v$ stores $\text{Int}(v)$ and the *canonical subset* (as a linked list) defined as

\[
C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \not\subseteq [x, x']\}
\]

Store intervals at nodes as high as possible.

Examples

\[
\begin{align*}
\text{Int}(v) &= [p_2, p_4] \subseteq s_2, s_5 \\
\text{Int}(\text{parent}(v)) &= (-\infty, p_4] \\
C(v) &= \{s_2, s_5\}
\end{align*}
\]
Tree Structure

- Leaves $\leftrightarrow$ elementary intervals (left-to-right)
- Internal node $v \leftrightarrow$ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at $v$.
- At every node or leaf $v$ stores $\text{Int}(v)$ and the canonical subset (as a linked list) defined as

$$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int(parent}(v)) \not\subseteq [x, x']\}$$

Store intervals at nodes as high as possible.

Examples

$$\text{Int}(v) = [p_2, p_4] \subseteq s_2, s_5$$
$$\text{Int(parent}(v)) = (-\infty, p_4] \not\subseteq s_2, s_5$$
$$C(v) = \{s_2, s_5\}$$
Tree Structure

- Leaves ↔ elementary intervals (left-to-right)
- Internal node $v$ ↔ union $\text{Int}(v)$ of elementary intervals at the leaves in the subtree $T(v)$ rooted at $v$.
- At every node or leaf $v$ stores $\text{Int}(v)$ and the canonical subset (as a linked list) defined as

$$ \mathcal{C}(v) = \{ [x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \not\subseteq [x, x'] \} $$

Store intervals at nodes as high as possible.

Examples

$\text{Int}(v) = [p_2, p_4] \subseteq s_2, s_5$

$\text{Int}(\text{parent}(v)) = (-\infty, p_4] \not\subseteq s_2, s_5$

$\mathcal{C}(v) = \{s_2, s_5\}$

$\mathcal{C}(u) = \{s_1\}$
Canonical Subset

\[ C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \not\subseteq [x, x'] \} \]

A leaf node \( \mu \) has a non-empty canonical subset if and only if \( \text{Int}(\mu) = [p_i, p_i] \), where \( p_i \) is the left endpoint of some interval.
Number of Leaves

\( n = |I|: \# \text{segments} \)

\( m \leq 2n: \# \text{distinct endpoints} \)

\[
\begin{align*}
[p_i, p_i] & \quad m \\
(p_i, p_{i+1}) & \quad m - 1 \\
(-\infty, p_1) & \quad 1 \\
(p_m, \infty) & \quad 1
\end{align*}
\]
Number of Leaves

\[ n = |I|: \text{#segments} \]
\[ m \leq 2n: \text{#distinct endpoints} \]

\[
\begin{align*}
[p_i, p_i] & \quad m \\
(p_i, p_{i+1}) & \quad m - 1 \\
(-\infty, p_1) & \quad 1 \\
(p_m, \infty) & \quad 1
\end{align*}
\]

- \(2m + 1 \leq 4n + 1\) leaves
Claim Each interval $[x, x'] \in I$ is stored at $\leq 2$ nodes at any depth.
Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at $\leq 2$ nodes at any depth.

Proof Suppose the interval is stored at the nodes $v_1, v_2, \ldots, v_k$, $k \geq 3$, at the same depth in the left to right order.
Storage of an Interval

**Claim** Each interval $[x, x'] \in I$ is stored at $\leq 2$ nodes at any depth.

**Proof** Suppose the interval is stored at the nodes $v_1, v_2, \ldots, v_k$, $k \geq 3$, at the same depth in the left to right order.

Then $v_2$ must be a sibling of either $v_1$ or $v_3$. 

![Diagram showing a tree with nodes $v_1$, $v_2$, and $v_3$. $v_2$ is a sibling of either $v_1$ or $v_3$. The parent of $v_2$ is indicated.]
Claim Each interval \([x, x']\) ∈ \(I\) is stored at \(\leq 2\) nodes at any depth.

Proof Suppose the interval is stored at the nodes \(v_1, v_2, \ldots, v_k, k \geq 3\), at the same depth in the left to right order.

Then \(v_2\) must be a sibling of either \(v_1\) or \(v_3\). Suppose its sibling is \(v_1\) without loss of generality.
Claim Each interval \([x, x'] \in I\) is stored at \(\leq 2\) nodes at any depth.

Proof Suppose the interval is stored at the nodes \(v_1, v_2, ..., v_k, k \geq 3\), at the same depth in the left to right order.

Then \(v_2\) must be a sibling of either \(v_1\) or \(v_3\).
Suppose its sibling is \(v_1\) without loss of generality.

\[
\begin{align*}
\text{Int}(v_1) & \subseteq [x, x'] \\
\text{Int}(v_2) & \subseteq [x, x']
\end{align*}
\]
Storage of an Interval

**Claim** Each interval $[x, x'] \in I$ is stored at $\leq 2$ nodes at any depth.

**Proof** Suppose the interval is stored at the nodes $v_1, v_2, ..., v_k$, $k \geq 3$, at the same depth in the left to right order.

Then $v_2$ must be a sibling of either $v_1$ or $v_3$. Suppose its sibling is $v_1$ without loss of generality.

$\text{Int}(v_1) \subseteq [x, x']$

$\text{Int}(v_2) \subseteq [x, x']$

$\text{Int}($parent$(v_2)) = \text{Int}(v_1) \cup \text{Int}(v_2) \subseteq [x, x']$
Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at $\leq 2$ nodes at any depth.

Proof Suppose the interval is stored at the nodes $v_1, v_2, \ldots, v_k$, $k \geq 3$, at the same depth in the left to right order.

Then $v_2$ must be a sibling of either $v_1$ or $v_3$.

Suppose its sibling is $v_1$ without loss of generality.

\[
\begin{align*}
\text{Int}(v_1) & \subseteq [x, x'] \\
\text{Int}(v_2) & \subseteq [x, x'] \\
\text{Int}(\text{parent}(v_2)) & = \text{Int}(v_1) \cup \text{Int}(v_2) \subseteq [x, x']
\end{align*}
\]

$[x, x']$ should be stored at $\text{parent}(v_2)$ or above instead of at $v_2$. Contradiction.
Claim: Each interval \([x, x']\) \(\in I\) is stored at \(\leq 2\) nodes at any depth.

Proof: Suppose the interval is stored at the nodes \(v_1, v_2, \ldots, v_k, k \geq 3\), at the same depth in the left to right order.

Then \(v_2\) must be a sibling of either \(v_1\) or \(v_3\).

Suppose its sibling is \(v_1\) without loss of generality.

\[
\text{Int}(v_1) \subseteq [x, x'] \\
\text{Int}(v_2) \subseteq [x, x']
\]

\[
\text{Int}(\text{parent}(v_2)) = \text{Int}(v_1) \cup \text{Int}(v_2) \subseteq [x, x']
\]

\([x, x']\) should be stored at \(\text{parent}(v_2)\) or above instead of at \(v_2\). Contradiction.
Total Storage

Following the claim, any interval is stored at most twice at any given length.
Following the claim, any interval is stored at most twice at any given length.

The required storage at each depth is $O(n)$. 
Total Storage

Following the claim, any interval is stored at most twice at any given length.

The required storage at each depth is $O(n)$.

Maximum tree depth (i.e., height) is $O(\log n)$.
Total Storage

Following the claim, any interval is stored at most twice at any given length.

The required storage at each depth is $O(n)$.

Maximum tree depth (i.e., height) is $O(\log n)$.

$O(n \log n)$
IV. Query Algorithm

QuerySegmentTree(\(v, q_x\))

1. report all the intervals in \(C(v)\) // canonical subset
2. if \(v\) is not a leaf
3. then if \(q_x \in \text{Int}(lc(v))\)
4. then QuerySegmentTree(lc(v), q_x)
5. else QuerySegmentTree(rc(v), q_x)
IV. Query Algorithm

QuerySegmentTree($v, q_x$)

1. report all the intervals in $C(v)$ // canonical subset
2. if $v$ is not a leaf
3. then if $q_x \in \text{Int}(lc(v))$
4. then QuerySegmentTree($lc(v), q_x$)
5. else QuerySegmentTree($rc(v), q_x$)

Example: $q_x$
IV. Query Algorithm

QuerySegmentTree(𝑣, 𝑞𝑥)

1. report all the intervals in 𝐶(𝑣) // canonical subset
2. if 𝑣 is not a leaf
3. then if 𝑞𝑥 ∈ Int(lc(𝑣))
4. then QuerySegmentTree(lc(𝑣), 𝑞𝑥)
5. else QuerySegmentTree(rc(𝑣), 𝑞𝑥)

Example: 𝑞𝑥
QuerySegmentTree(root, 𝑞𝑥)
IV. Query Algorithm

QuerySegmentTree(\(v, q_x\))

1. report all the intervals in \(C(v)\) // canonical subset
2. if \(v\) is not a leaf
3. then if \(q_x \in \text{Int}(lc(v))\)
4. then QuerySegmentTree(lc(v), \(q_x\))
5. else QuerySegmentTree(rc(v), \(q_x\))

Example: \(q_x\)
QuerySegmentTree(root, \(q_x\))
IV. Query Algorithm

QuerySegmentTree(\(v, q_x\))

1. report all the intervals in \(C(v)\) // canonical subset
2. if \(v\) is not a leaf
3. then if \(q_x \in \text{Int}(lc(v))\)
4. then QuerySegmentTree(\(lc(v), q_x\))
5. else QuerySegmentTree(\(rc(v), q_x\))

Example: \(q_x\)
QuerySegmentTree(root, \(q_x\))

Output in order: \(s_2, s_5, s_1\)
IV. Query Algorithm

QuerySegmentTree\( (v, q_x) \)

1. report all the intervals in \( C(v) \) // canonical subset
2. if \( v \) is not a leaf
3. then if \( q_x \in \text{Int}(lc(v)) \)
4. then QuerySegmentTree\( (lc(v), q_x) \)
5. else QuerySegmentTree\( (rc(v), q_x) \)

Example: \( q_x \)

QuerySegmentTree\( (\text{root}, q_x) \)

Output in order:
\( s_2, s_5, s_1 \)

Query time:
\( O(\log n + k) \)
IV. Query Algorithm

QuerySegmentTree($v, q_x$)

1. report all the intervals in $C(v)$ // canonical subset
2. if $v$ is not a leaf
3. then if $q_x \in \text{Int}(lc(v))$
4. then QuerySegmentTree($lc(v), q_x$)
5. else QuerySegmentTree($rc(v), q_x$)

Example: $q_x$

QuerySegmentTree(root, $q_x$)

Output in order: $s_2, s_5, s_1$

Query time:

$O(\log n + k)$

#reported intervals
Segment Tree Construction

- Sort endpoints from $I$ to yield elementary intervals.
Segment Tree Construction

- Sort endpoints from $I$ to yield elementary intervals.

$O(n \log n)$
Segment Tree Construction

- Sort endpoints from $I$ to yield elementary intervals.
  
  $O(n \log n)$

- Construct a balanced BST in a bottom-up way.
Segment Tree Construction

- Sort endpoints from $I$ to yield elementary intervals.
  
  $O(n \log n)$

- Construct a balanced BST in a bottom-up way.

  Determine interval $I(v)$ for each node $v$. 
Segment Tree Construction

- Sort endpoints from $I$ to yield elementary intervals.
  \[ O(n \log n) \]
- Construct a balanced BST in a bottom-up way.
  Determine interval $I(v)$ for each node $v$.
  \[ O(n) \] for all the nodes together
Segment Tree Construction

- Sort endpoints from $I$ to yield elementary intervals.
  
  $\mathcal{O}(n \log n)$

- Construct a balanced BST in a bottom-up way.

Determine interval $I(v)$ for each node $v$.

$\mathcal{O}(n)$ for all the nodes together

- Compute canonical subsets by inserting original intervals $[x, x']$ from $I$ one by one.
Segment Tree Insertion

InsertSegmentTree(v, [x, x'])

1. if Int(v) ⊆ [x, x'] // Int(parent(v)) ⊈ [x, x'] holds
2. then store [x, x'] at v
3. else if Int(lc(v)) ∩ [x, x'] ≠ ∅
4. then InsertSegmentTree(lc(v), [x, x'])
5. if Int(rc(v)) ∩ [x, x'] ≠ ∅
6. then InsertSegmentTree(rc(v), [x, x'])
InsertSegmentTree(v, [x, x'])

1. if Int(v) ⊆ [x, x'] // Int(parent(v)) ∉ [x, x'] holds
2. then store [x, x'] at v
3. else if Int(lc(v)) ∩ [x, x'] ≠ ∅
4. then InsertSegmentTree(lc(v), [x, x'])
5. if Int(rc(v)) ∩ [x, x'] ≠ ∅
6. then InsertSegmentTree(rc(v), [x, x'])

- At each visited node v, either [x, x'] is stored or Int(v) contains an endpoint of [x, x'].
Segment Tree Insertion

\[ \text{InsertSegmentTree}(v, [x, x']) \]

1. \textbf{if} \( \text{Int}(v) \subseteq [x, x'] \) \quad \text{// \( \text{Int}(\text{parent}(v)) \not\subseteq [x, x'] \) holds}
2. \textbf{then} \quad \text{store} \ [x, x'] \ \text{at} \ v
3. \textbf{else if} \ \text{Int(lc}(v)\text{)) \cap [x, x'] \not= \emptyset
4. \textbf{then} \quad \text{InsertSegmentTree(lc}(v), [x, x'])
5. \textbf{if} \ \text{Int(rc}(v)\text{)) \cap [x, x'] \not= \emptyset
6. \textbf{then} \quad \text{InsertSegmentTree(rc}(v), [x, x'])

- At each visited node \( v \), either \( [x, x'] \) is stored or \( \text{Int}(v) \) contains an endpoint of \( [x, x'] \).
  - An interval is stored \( \leq 2 \) times at each level.
Segment Tree Insertion

\[\text{InsertSegmentTree}(v, [x, x'])\]

1. if \(\text{Int}(v) \subseteq [x, x']\) // \(\text{Int}(\text{parent}(v)) \not\subseteq [x, x']\) holds
2. \hspace{1em} then store \([x, x']\) at \(v\)
3. \hspace{1em} else if \(\text{Int}(\text{lc}(v)) \cap [x, x'] \neq \emptyset\)
4. \hspace{2em} then \(\text{InsertSegmentTree}(\text{lc}(v), [x, x'])\)
5. \hspace{1em} if \(\text{Int}(\text{rc}(v)) \cap [x, x'] \neq \emptyset\)
6. \hspace{2em} then \(\text{InsertSegmentTree}(\text{rc}(v), [x, x'])\)

- At each visited node \(v\), either \([x, x']\) is stored or \(\text{Int}(v)\) contains an endpoint of \([x, x']\).
  - An interval is stored \(\leq 2\) times at each level.
  - At each depth, there exist
Segment Tree Insertion

InsertSegmentTree(\(v, [x, x']\))

1. if \(\text{Int}(v) \subseteq [x, x']\) // \(\text{Int}(\text{parent}(v)) \notin [x, x']\) holds
2. then store \([x, x']\) at \(v\)
3. else if \(\text{Int}(\text{lc}(v)) \cap [x, x'] \neq \emptyset\)
4. then InsertSegmentTree(\(\text{lc}(v), [x, x']\))
5. if \(\text{Int}(\text{rc}(v)) \cap [x, x'] \neq \emptyset\)
6. then InsertSegmentTree(\(\text{rc}(v), [x, x']\))

- At each visited node \(v\), either \([x, x']\) is stored or \(\text{Int}(v)\) contains an endpoint of \([x, x']\).
  - An interval is stored \(\leq 2\) times at each level.
  - At each depth, there exist \(\leq 1\) node \(u\) such that \(x \in \text{Int}(u)\)
Segment Tree Insertion

\[ \text{InsertSegmentTree}(v, [x, x']) \]

1. \( \text{if } \text{Int}(v) \subseteq [x, x'] \) // \( \text{Int(parent}(v)) \not\subseteq [x, x'] \) holds
2. \( \text{then store } [x, x'] \text{ at } v \)
3. \( \text{else if } \text{Int}(lc(v)) \cap [x, x'] \neq \emptyset \)
4. \( \text{then InsertSegmentTree}(lc(v), [x, x']) \)
5. \( \text{if } \text{Int}(rc(v)) \cap [x, x'] \neq \emptyset \)
6. \( \text{then InsertSegmentTree}(rc(v), [x, x']) \)

- At each visited node \( v \), either \( [x, x'] \) is stored or \( \text{Int}(v) \) contains an endpoint of \( [x, x'] \).
  - An interval is stored \( \leq 2 \) times at each level.
  - At each depth, there exist
    \[ \leq 1 \text{ node } u \text{ such that } x \in \text{Int}(u) \]
    \[ \leq 1 \text{ node } u' \text{ such that } x' \in \text{Int}(u') \]
Construction Time

- $\leq 2$ storage actions + $\leq 2$ containments
Construction Time

- $\leq 2$ storage actions + $\leq 2$ containments

\[ \downarrow \]

$\leq 4$ nodes visited per level.
Construction Time

- \( \leq 2 \) storage actions + \( \leq 2 \) containments

\[ \downarrow \]

\( \leq 4 \) nodes visited per level.

\[ \downarrow \]

\( O \log n \) time to insert an interval.

Me to insert an interval.
Construction Time

- $\leq 2$ storage actions + $\leq 2$ containments

  $\Downarrow$

  $\leq 4$ nodes visited per level.

  $\Downarrow$

  $O \log n$ time to insert an interval.

  $\Downarrow$

  $O(n \log n)$ time for segment tree construction.
Construction Time

- $\leq 2$ storage actions + $\leq 2$ containments
  \[ \downarrow \]
  - $\leq 4$ nodes visited per level.
    \[ \downarrow \]
    $O \log n$ time to insert an interval.
    \[ \downarrow \]
    $O(n \log n)$ time for segment tree construction.

Compared with an interval tree, a segment tree has
- the same query time $O(\log n + k)$
- a larger storage $O(n \log n)$ than $O(n)$,
Construction Time

- ≤ 2 storage actions + ≤ 2 containments
  - ↓
  - ≤ 4 nodes visited per level.
  - ↓
  - $O \log n$ time to insert an interval.
  - ↓
  - $O(n \log n)$ time for segment tree construction.

Compared with an interval tree, a segment tree has
- the same query time $O(\log n + k)$
- a larger storage $O(n \log n)$ than $O(n)$,

but it allows to answer more complicated queries.
V. Back to Windowing

Query segment: \( q = q_x \times [q_y, q'_y] \)

Over \( n \) arbitrarily oriented segments

\( q \)

\( (q_x, q_y) \)
V. Back to Windowing

Query segment: \( q = q_x \times [q_y, q'_y] \)

Over \( n \) arbitrarily oriented segments

Construct a segment tree \( \mathcal{T} \).

\( (q_x, q'_y) \)

\( (q_x, q_y) \)
Query segment: \( q = q_x \times [q_y, q'_y] \)

Over \( n \) arbitrarily oriented segments

Construct a segment tree \( T \).

- On the \( x \)-intervals of the segments in \( S \).
- Canonical subset \( C(v) \) at a vertex \( v \) store segments rather than their \( x \)-intervals.
Query segment: \( q = q_x \times [q_y, q'_y] \)  

Over \( n \) arbitrarily oriented segments

Construct a segment tree \( \mathcal{T} \).

- On the \( x \)-intervals of the segments in \( S \).
- Canonical subset \( C(v) \) at a vertex \( v \) store segments rather than their \( x \)-intervals.

Segment \( s \in C(v) \) if it crosses the slab \( S(v): \text{Int}(v) \times (-\infty, \infty) \) but does not cross its parent’s slab.
V. Back to Windowing

Query segment: \( q = q_x \times [q_y, q'_y] \)

Over \( n \) arbitrarily oriented segments

Construct a segment tree \( T \).

- On the \( x \)-intervals of the segments in \( S \).
- Canonical subset \( C(v) \) at a vertex \( v \) store segments rather than their \( x \)-intervals.

Segment \( s \in C(v) \) if it crosses the slab \( S(v) \): \( \text{Int}(v) \times (-\infty, \infty) \)

but does not cross its parent’s slab.
Search with $q_x$ in $T$. 

Segments on the Search Path

Segment tree

$(-\infty, \infty)$
Segments on the Search Path

- Search with $q_x$ in $T$. 

Segment tree

$(-\infty, \infty)$
Segments on the Search Path

- Search with \( q_x \) in \( T \).
- \( O(\log n) \) canonical sets together include all the segments intersected by the vertical line \( x = q_x \).
Segments on the Search Path

- Search with $q_x$ in $\mathcal{T}$.
- $O(\log n)$ canonical sets together include all the segments intersected by the vertical line $x = q_x$.
- $v$ is a node on the search path.
Segments on the Search Path

- Search with \( q_x \) in \( \mathcal{T} \).
- \( O(\log n) \) canonical sets together include all the segments intersected by the vertical line \( x = q_x \).
- \( v \) is a node on the search path.
- Segment \( s \in C(v) \) is intersected by \( q \) iff
Segments on the Search Path

- Search with $q_x$ in $\mathcal{T}$.
- $O(\log n)$ canonical sets together include all the segments intersected by the vertical line $x = q_x$.
- $v$ is a node on the search path.
- Segment $s \in C(v)$ is intersected by $q$ iff
  
  \[(q_x, q_y) \text{ below } s \text{ and } (q_x, q'_y) \text{ above } s\]
VI. Storage of a Canonical Set

- Segments in $C(v)$ do not intersect each other.
- Each segment is over an $x$-interval containing $\text{Int}(v)$. 

\[ (-\infty, \infty) \]

\[ v \]

\[ C(v) \]
VI. Storage of a Canonical Set

- Segments in $C(v)$ do not intersect each other.
- Each segment is over an $x$-interval containing $\text{Int}(v)$.
- These segments span the slab $\text{Int}(v) \times [-\infty, \infty]$.
- They do not intersect each other in the interior.
VI. Storage of a Canonical Set

- Segments in $C(v)$ do not intersect each other.
- Each segment is over an $x$-interval containing $\text{Int}(v)$.
- These segments span the slab $\text{Int}(v) \times [-\infty, \infty]$.
- They do not intersect each other in the interior.
- Store the segments in a balanced BST $B(v)$ in the vertical order.
VI. Storage of a Canonical Set

- Segments in $C(v)$ do not intersect each other.
- Each segment is over an $x$-interval containing $\text{Int}(v)$.
- These segments span the slab $\text{Int}(v) \times [-\infty, \infty]$.
- They do not intersect each other in the interior.
- Store the segments in a balanced BST $B(v)$ in the vertical order.

Slab

$\text{Int}(v)$

$C(v)$

$(\infty, \infty)$

Associate structure $B(v)$

$s_1$

$s_2$

$s_3$

$s_4$

$s_5$
VI. Storage of a Canonical Set

- Segments in $C(v)$ do not intersect each other.
- Each segment is over an $x$-interval containing $\text{Int}(v)$.
- These segments span the slab $\text{Int}(v) \times [−\infty, \infty]$.
- They do not intersect each other in the interior.
- Store the segments in a balanced BST $B(v)$ in the vertical order.

Slab

$\text{Int}(v)$

$s_1$

$s_2$

$s_3$

$s_4$

$s_5$

$C(v)$

$(-\infty, \infty)$

$O(\log n + k_v)$ query time within $B(v)$

#intersected segments in $C(v)$

associate structure $B(v)$
The set $S$ of segments is stored in a segment tree $T$ based on their $x$-intervals.

The canonical subset $C(v)$ of every internal node $v$ in $T$ is stored in a BST $B(v)$ based on the vertical order within the slab $\text{Int}(v) \times [-\infty, \infty]$.

Total storage: $O(n \log n)$

Construction time: $O(n \log n)$
Query Algorithm

Query object: a vertical line segment \( q: q_x \times [q_y, q'_y] \)
Set \( S = \{s_1, s_2, \ldots, s_n\} \) of arbitrarily oriented segments
Query Algorithm

Query object: a vertical line segment $q$: $q_x \times [q_y, q'_y]$
Set $S = \{s_1, s_2, \ldots, s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$. 
Query object: a vertical line segment \( q: q_x \times [q_y, q'_y] \)
Set \( S = \{s_1, s_2, ..., s_n\} \) of arbitrarily oriented segments

\[ \textcolor{red}{\blacklozenge} \text{Search with } q_x \text{ in a segment tree } T. \]
Query Algorithm

Query object: a vertical line segment $q$: $q_x \times [q_y, q_y']$
Set $S = \{s_1, s_2, \ldots, s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$. 

```latex
\begin{tikzpicture}
    \node (v) at (0,0) [circle, draw] {$v$};
    \node (c) at (0,-1) [circle, draw] {$C(v)$};
    \draw[->, red] (c) to [bend left=30] node [auto] {$(\neg \infty, \infty)$} (v);
\end{tikzpicture}
```
Query Algorithm

Query object: a vertical line segment $q$: $q_x \times [q_y, q'_y]$
Set $S = \{s_1, s_2, \ldots, s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$.
- At every node $v$ on the search path, search with the two endpoints of $q$ (essentially $q_y$ and $q'_y$) to report segments in $C(v)$ intersected by $q$. 
Query Algorithm

Query object: a vertical line segment $q$: $q_x \times [q_y, q'_y]$ 
Set $S = \{s_1, s_2, \ldots, s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$.
- At every node $v$ on the search path, search with the two endpoints of $q$ (essentially $q_y$ and $q'_y$) to report segments in $C(v)$ intersected by $q$. 

Diagram:

- $\mathcal{B}(v)$
- $\mathcal{C}(v)$
- $(-\infty, \infty)$
- $v$
Query object: a vertical line segment \( q: q_x \times [q_y, q'_y] \)
Set \( S = \{s_1, s_2, \ldots, s_n\} \) of arbitrarily oriented segments

- Search with \( q_x \) in a segment tree \( T \).
- At every node \( v \) on the search path, search with the two endpoints of \( q \) (essentially \( q_y \) and \( q'_y \)) to report segments in \( C(v) \) intersected by \( q \).
Query Algorithm

Query object: a vertical line segment $q: q_x \times [q_y, q_y']$
Set $S = \{s_1, s_2, ..., s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$.
- At every node $v$ on the search path, search with the two endpoints of $q$ (essentially $q_y$ and $q_y'$) to report segments in $C(v)$ intersected by $q$. 

![Diagram of segment tree with query algorithm steps](attachment:query_algorithm_diagram.png)
Query Algorithm

Query object: a vertical line segment $q: q_x \times [q_y, q_y']$
Set $S = \{s_1, s_2, \ldots, s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$.
- At every node $v$ on the search path, search with the two endpoints of $q$ (essentially $q_y$ and $q_y'$) to report segments in $C(v)$ intersected by $q$. 

```
B(v)
```

```
C(v)
```

```
(q_x, q_y)
```

```
(q_x, q_y')
```
Query Algorithm

Query object: a vertical line segment $q: q_x \times [q_y, q_y']$

Set $S = \{s_1, s_2, ..., s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$.

- At every node $v$ on the search path, search with the two endpoints of $q$ (essentially $q_y$ and $q_y'$) to report segments in $C(v)$ intersected by $q$.
Query Algorithm

Query object: a vertical line segment $q: q_x \times [q_y, q'_y]$
Set $S = \{s_1, s_2, \ldots, s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$.
- At every node $v$ on the search path, search with the two endpoints of $q$ (essentially $q_y$ and $q'_y$) to report segments in $C(v)$ intersected by $q$.

- The search takes $O(\log n + k_v)$ time.
- $O(\log n)$ nodes on the search path, each requiring such a search.
Query Algorithm

Query object: a vertical line segment $q$: $q_x \times [q_y, q'_y]$
Set $S = \{s_1, s_2, \ldots, s_n\}$ of arbitrarily oriented segments

- Search with $q_x$ in a segment tree $T$.

- At every node $v$ on the search path, search with the two endpoints of $q$ (essentially $q_y$ and $q'_y$) to report segments in $C(v)$ intersected by $q$.

- The search takes $O(\log n + k_v)$ time.

- $O(\log n)$ nodes on the search path, each requiring such a search.

Query time: $O(\log^2 n + k)$