Yan-Bin Jia, Iowa State University

Segment Trees

Outline:

I. General windowing queries

II. Inapplicability of an interval tree

III. Locus approach

IV. Structure of a segment tree

V. Query and construction

VI. Solution of a general windowing query

VII. Analysis

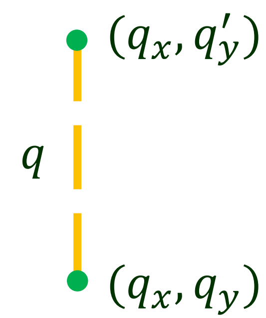
Textbook: *Computational Geometry: Algorithms and Applications (*3rd ed.) by M. de Berg et al., Springer-Verlag, 2008.

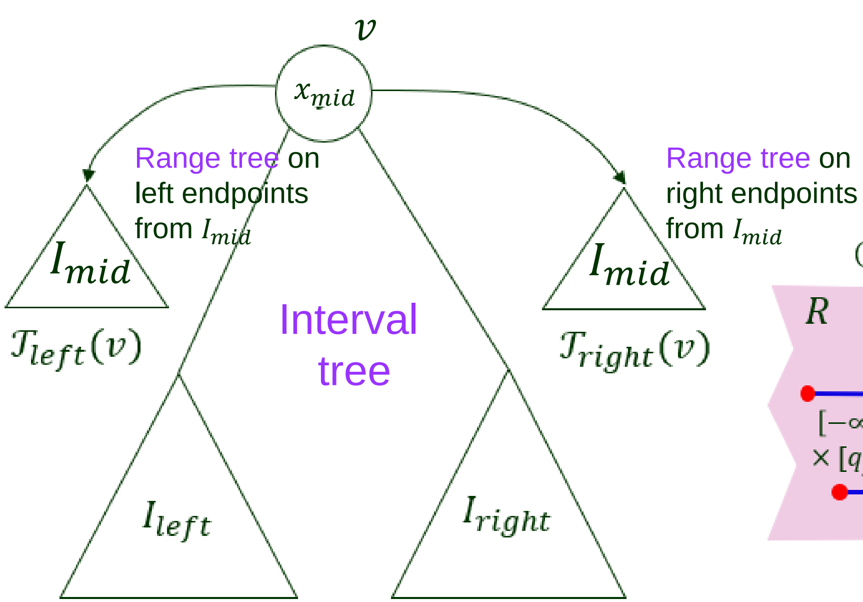
I. Windowing Query Over Horizontal Segments

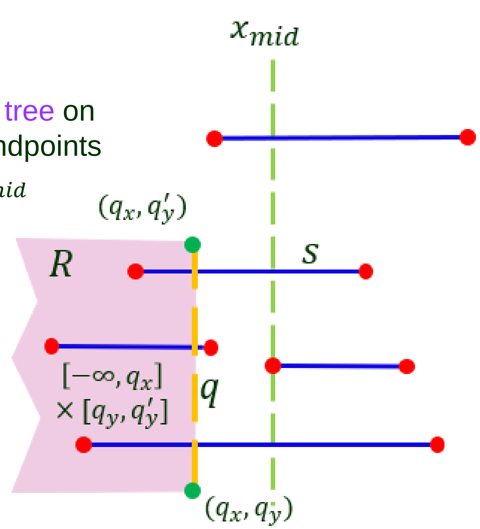
Interval tree + range trees

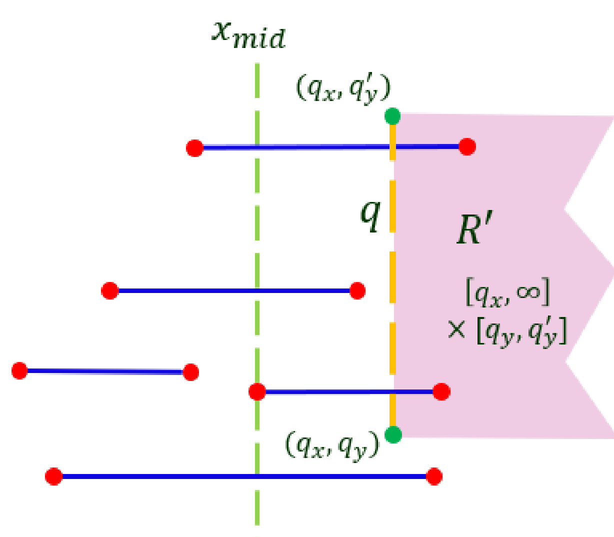
Query object: vertical segment $q_x \times [q_y, q'_y]$

Set: n horizontal line segments







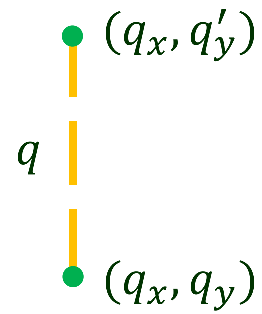


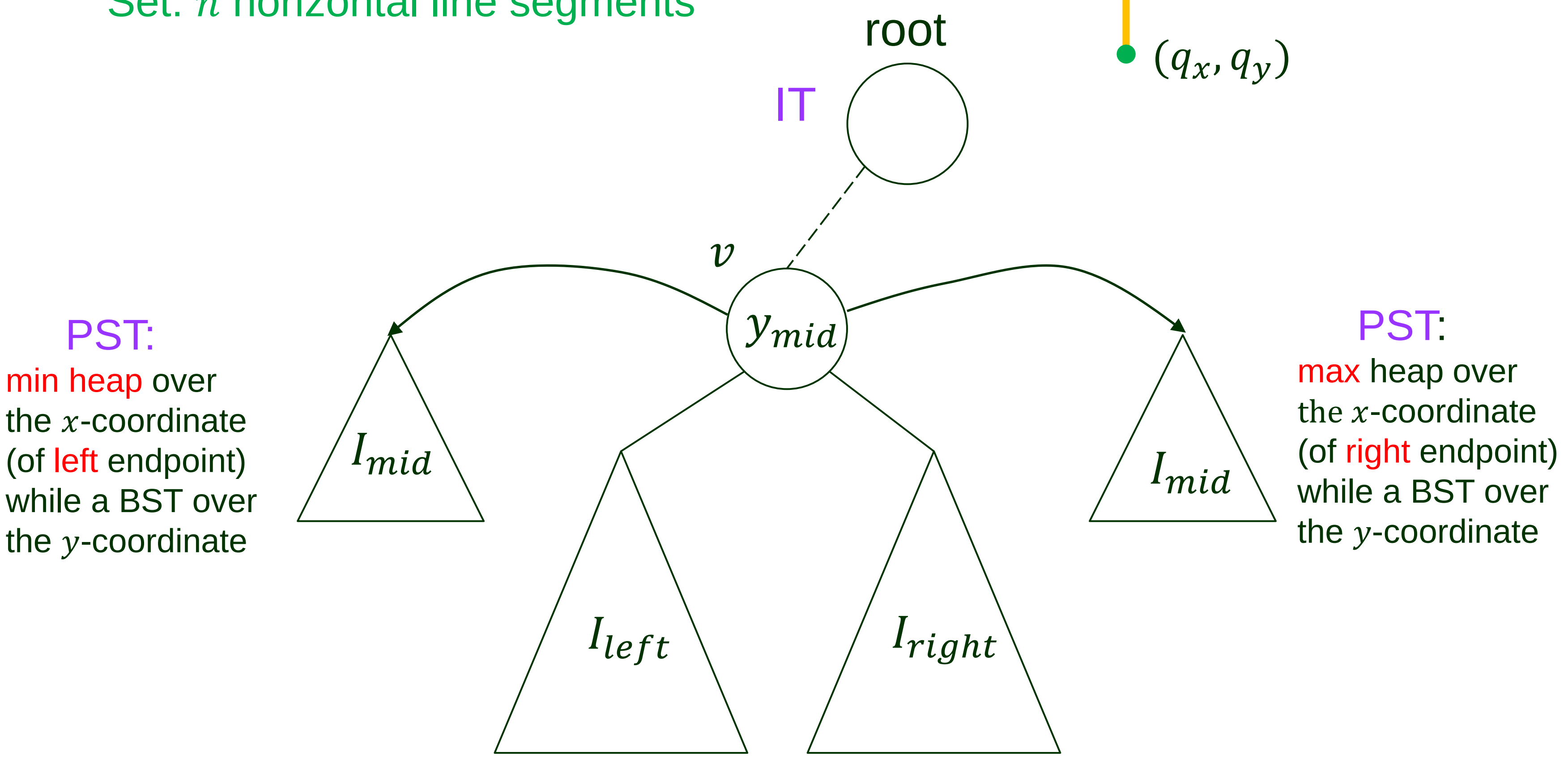
Interval Tree + Priority Search Trees

Applicable to axis-parallel segments only.

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

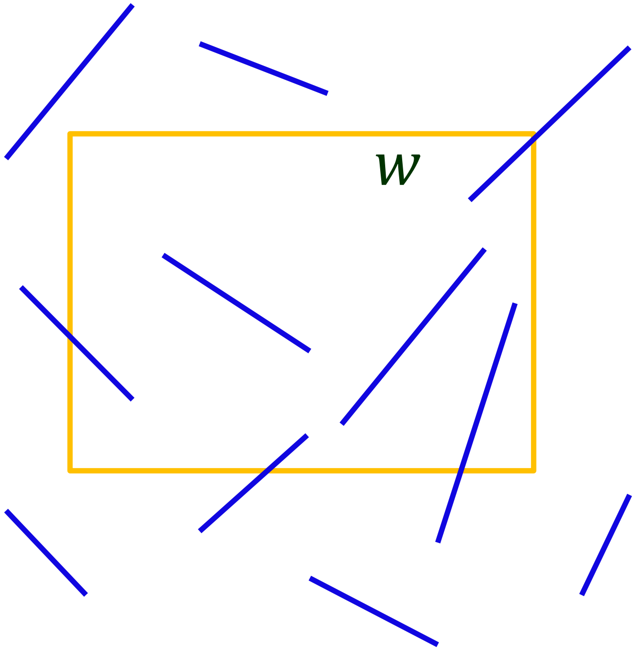
Set: n horizontal line segments





General Windowing Queries

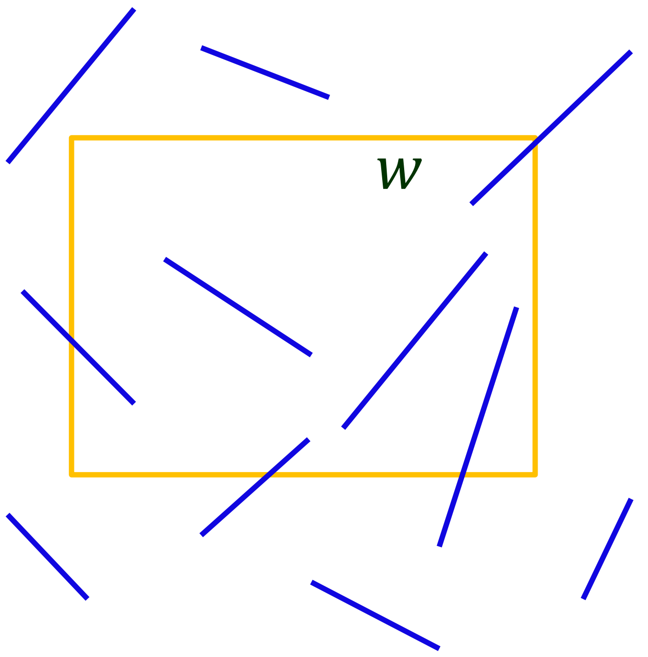
Line segments can have arbitrary orientations.



General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?

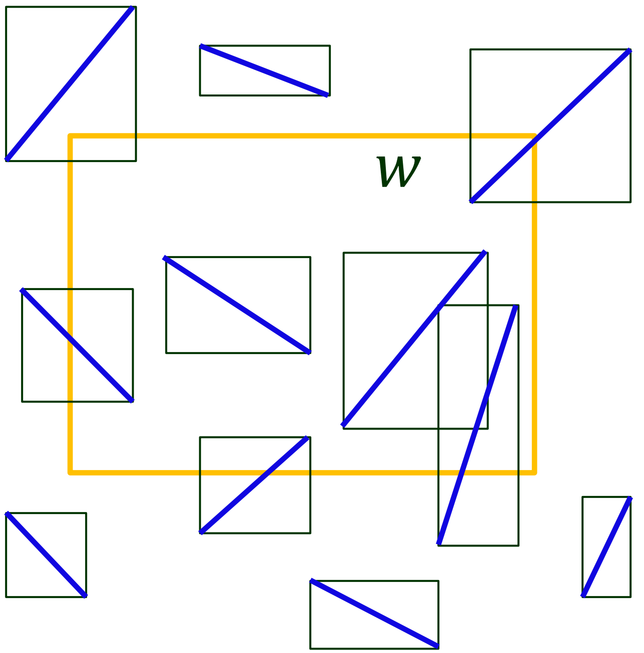


General Windowing Queries

Line segments can have arbitrary orientations.

A bounding box approach?

- Replace each segment with its bounding box.

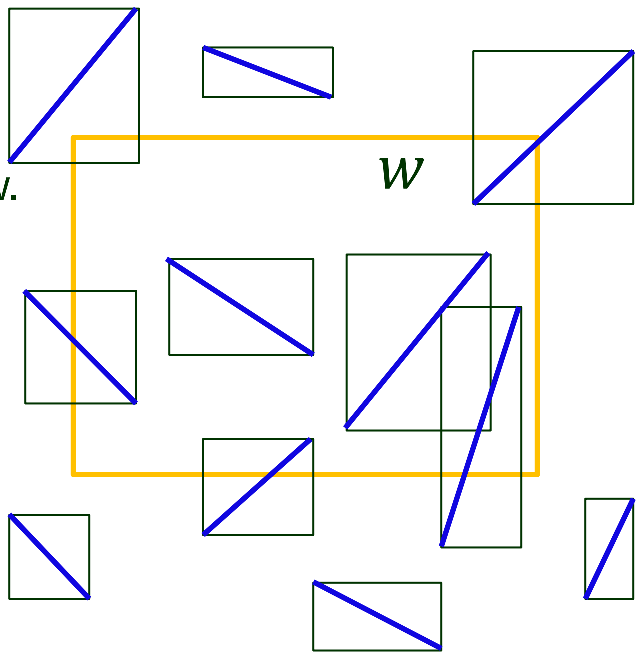


General Windowing Queries

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.



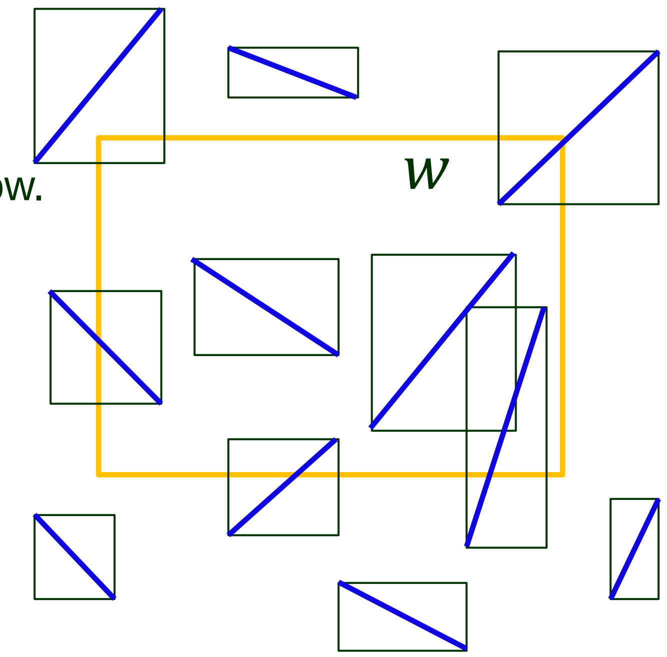
General Windowing Queries

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

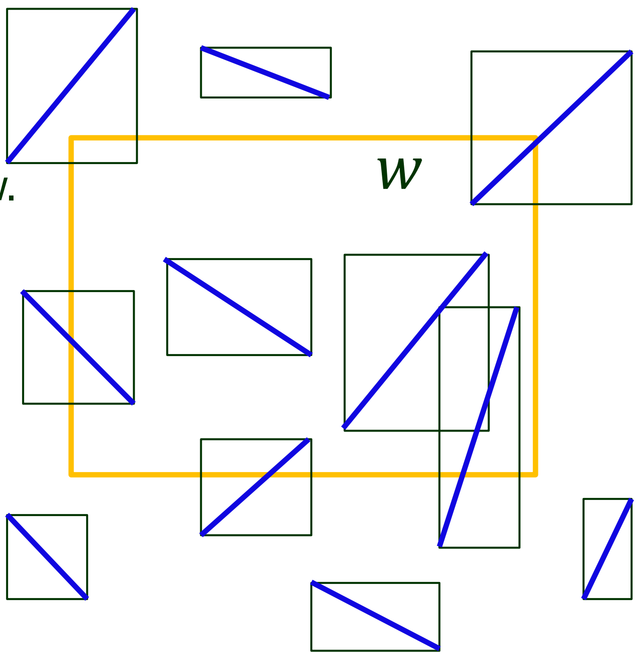
Line segments can have arbitrary orientations.

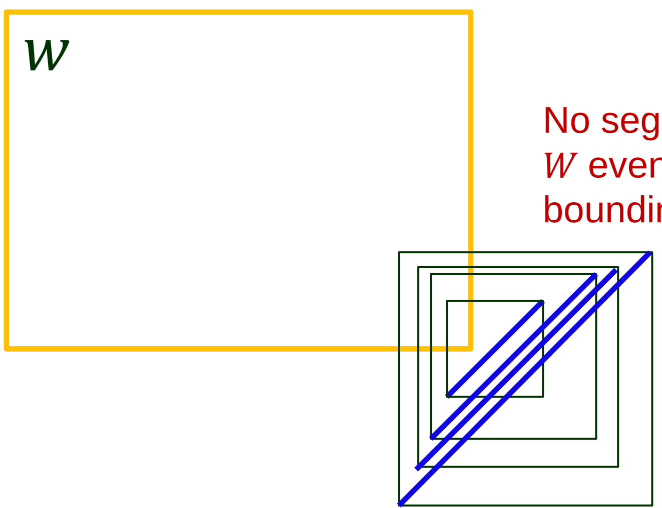
A bounding box approach?

- 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

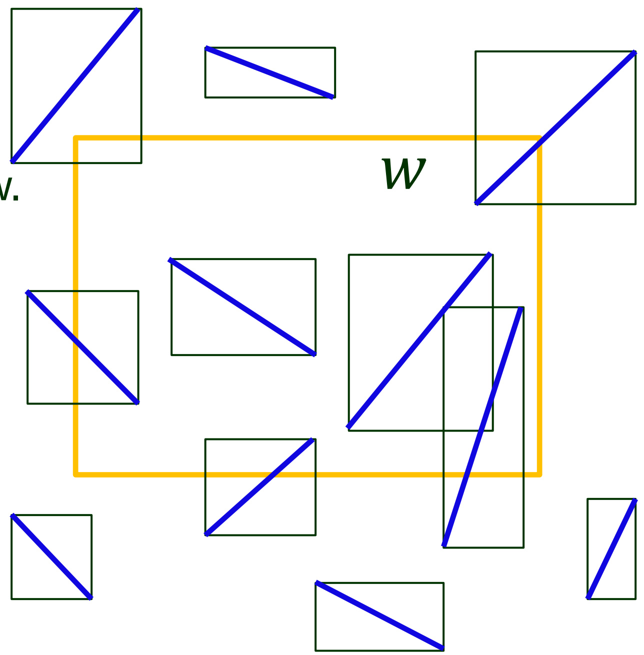
Line segments can have arbitrary orientations.

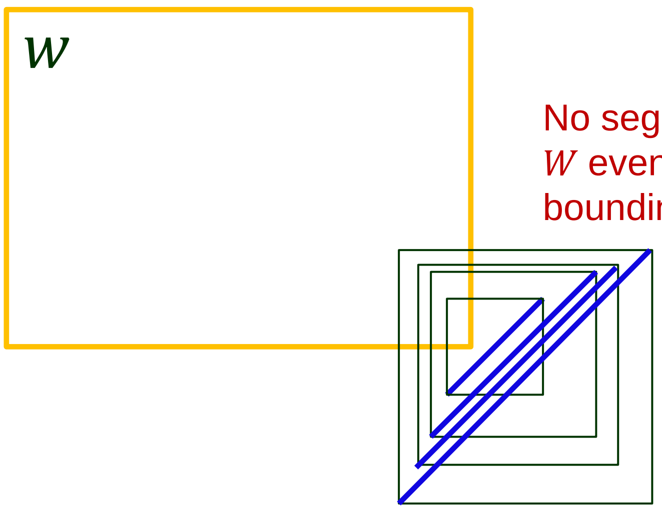
A bounding box approach?

- 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 ≥ 1 endpoints in W.

II. Thinking Top-Down

- Segments with ≥ 1 endpoints in W.

Range trees.

II. Thinking Top-Down

- Segments with ≥ 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 ≥ 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 ≥ 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 ≥ 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: $S = \{s_1, s_2, \ldots, s_n\}$ 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 $\Longrightarrow$ $I_{mid}(v)$ at node v

Inapplicability of an Interval Tree

An interval tree is not very helpful.

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

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

Supposing $x_{mid}(v) > q_x$

Inapplicability of an Interval Tree

An interval tree is not very helpful.

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

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

Supposing $x_{mid}(v) > q_x$

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

Inapplicability of an Interval Tree

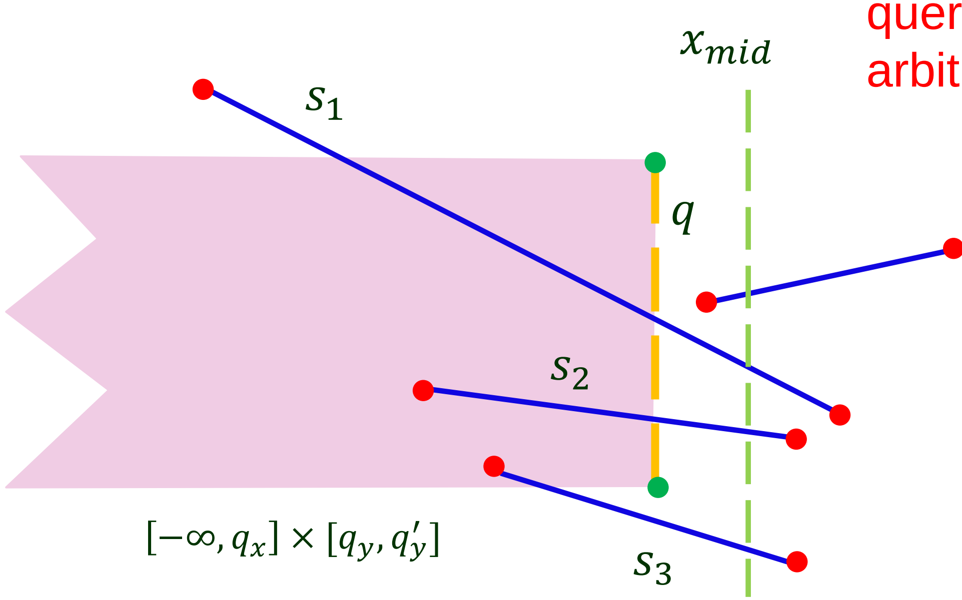
An interval tree is not very helpful.

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

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

Supposing $x_{mid}(v) > q_x$

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



Inapplicability of an Interval Tree

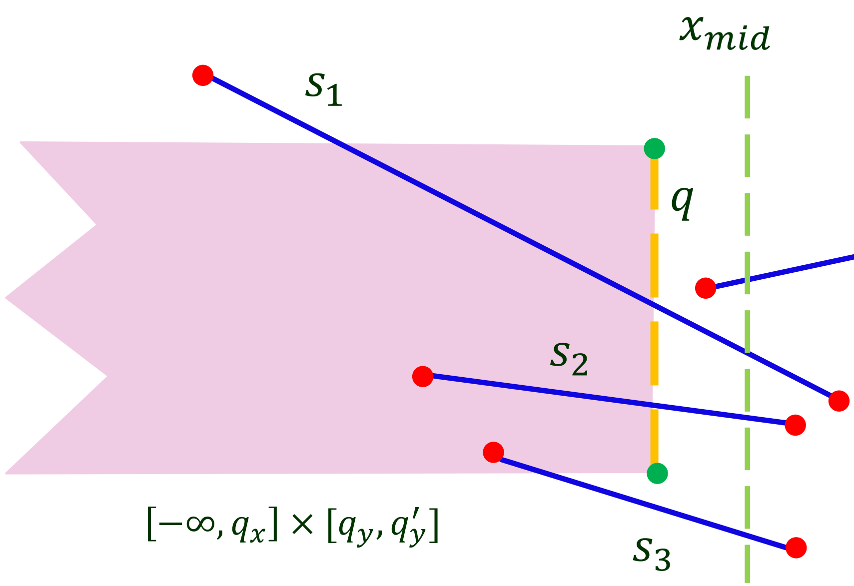
An interval tree is not very helpful.

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

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

Supporting $x_{mid}(v) > q_x$ (for a horizontal segment).

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



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

Inapplicability of an Interval Tree

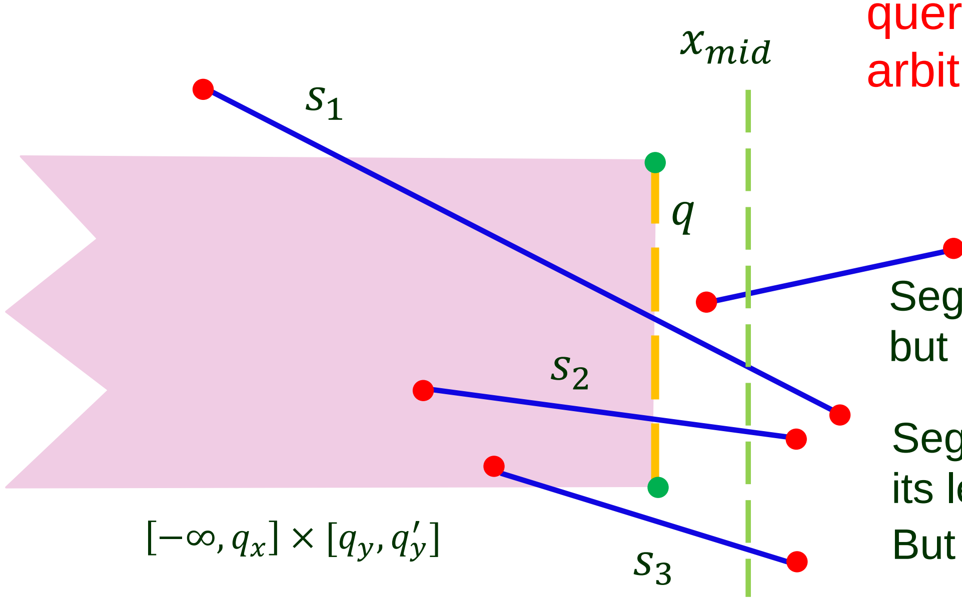
An interval tree is not very helpful.

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

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

Supposing $x_{mid}(v) > q_x$ (for a horizontal segment).

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



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

Segment $s_3 \in I_{mid}(v)$ and its left endpoint $\in (-\infty, q_x] \times [q_y, q_y']$. But it has no intersection with q.

III. 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.

III. 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.

Project all the segments onto the x-axis first.

III. 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.

Project all the segments onto the x-axis first.

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

III. 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.

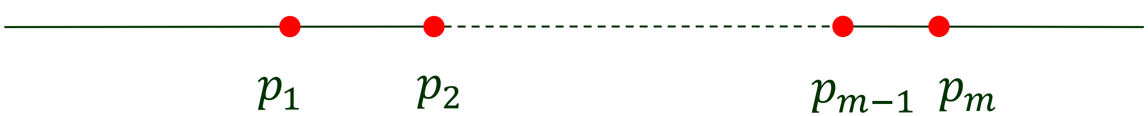
Project all the segments onto the x-axis first.

1D Query

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

Output: all intervals containing q_x

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



III. 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.

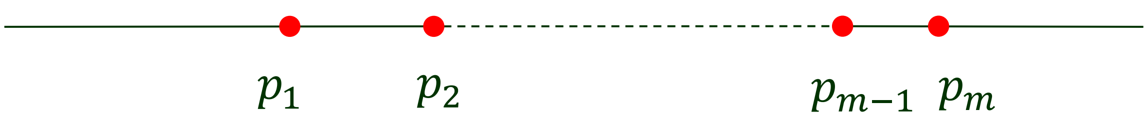
Project all the segments onto the x-axis first.

1D Query

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

Output: all intervals containing q_x

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



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

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

III. 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.

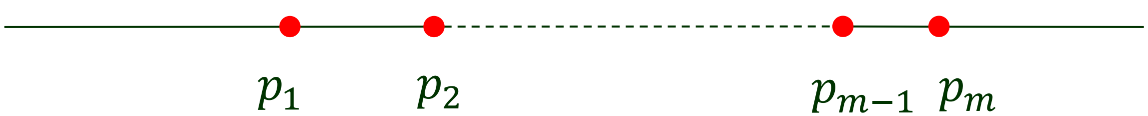
Project all the segments onto the x-axis first.

1D Query

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

Output: all intervals containing q_x

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

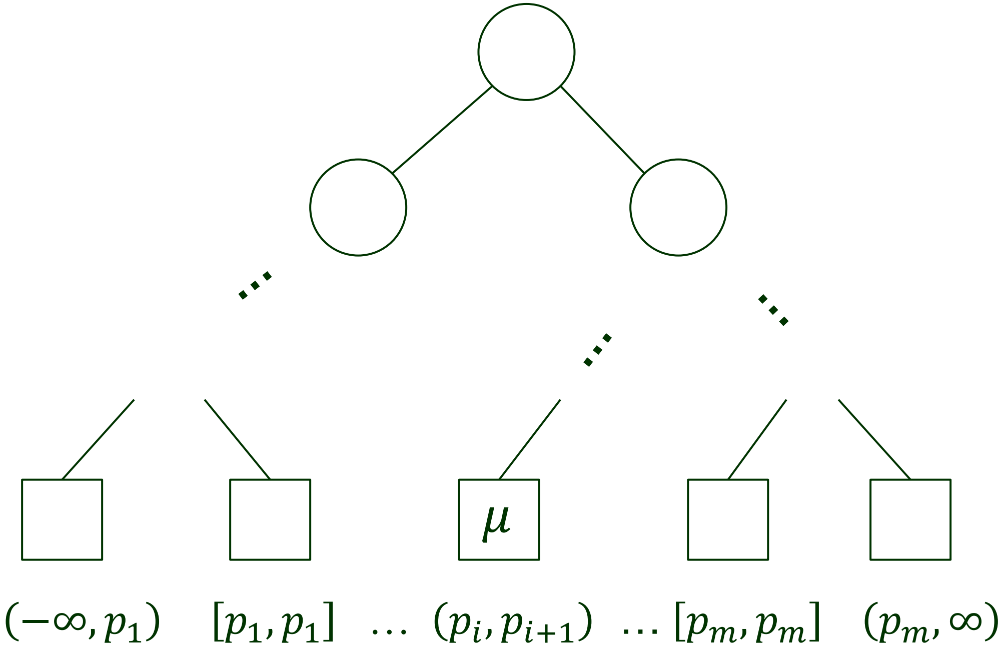


The parameter space $(-\infty, \infty)$ is partitioned into *elementary intervals*.

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

- Every open interval has two consecutive endpoints.
- Every closed interval consists of a single endpoint.
- Open intervals alternate with closed intervals.

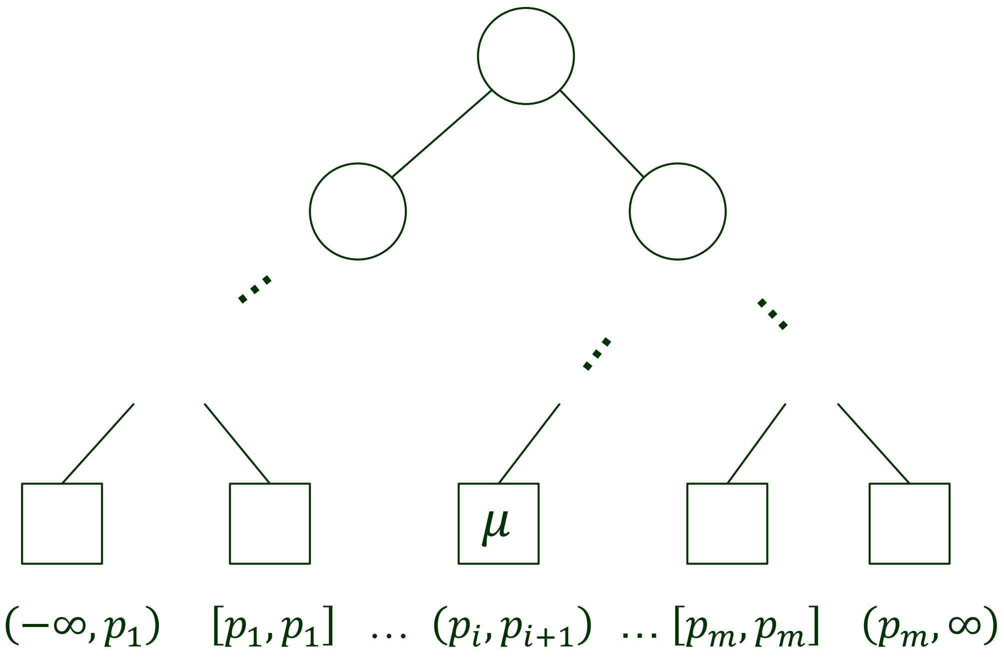
Using a Binary Search Tree?



μ : a leaf

Int(μ): elementary interval corresponding to μ.

Using a Binary Search Tree?

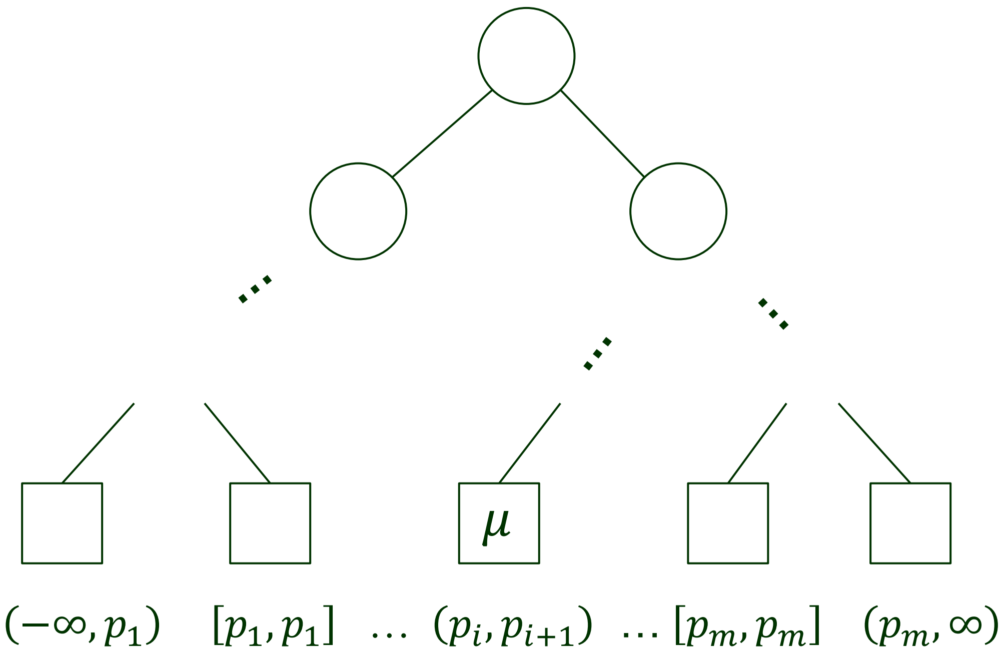


μ : a leaf

Int(μ): elementary interval corresponding to μ.

♠ Store at the leaf μ all the intervals in I that contain Int(μ).

Using a Binary Search Tree?



μ : a leaf

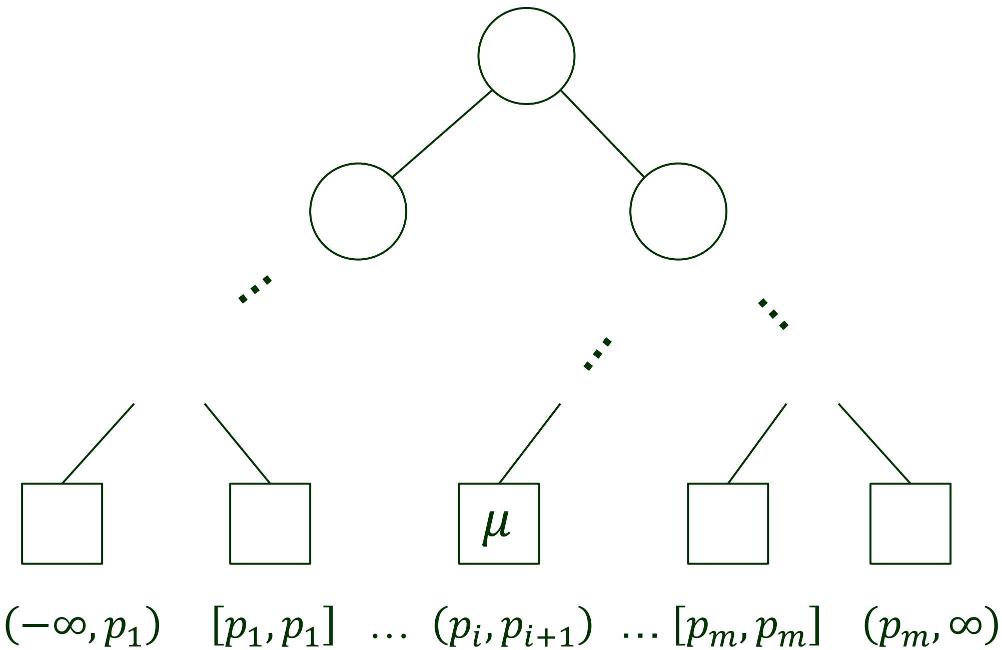
Int(μ): elementary interval corresponding to μ.

♠ Store at the leaf μ all the intervals in I that contain Int(μ).

♠ Query time

$$O(\log n + k)$$

Using a Binary Search Tree?



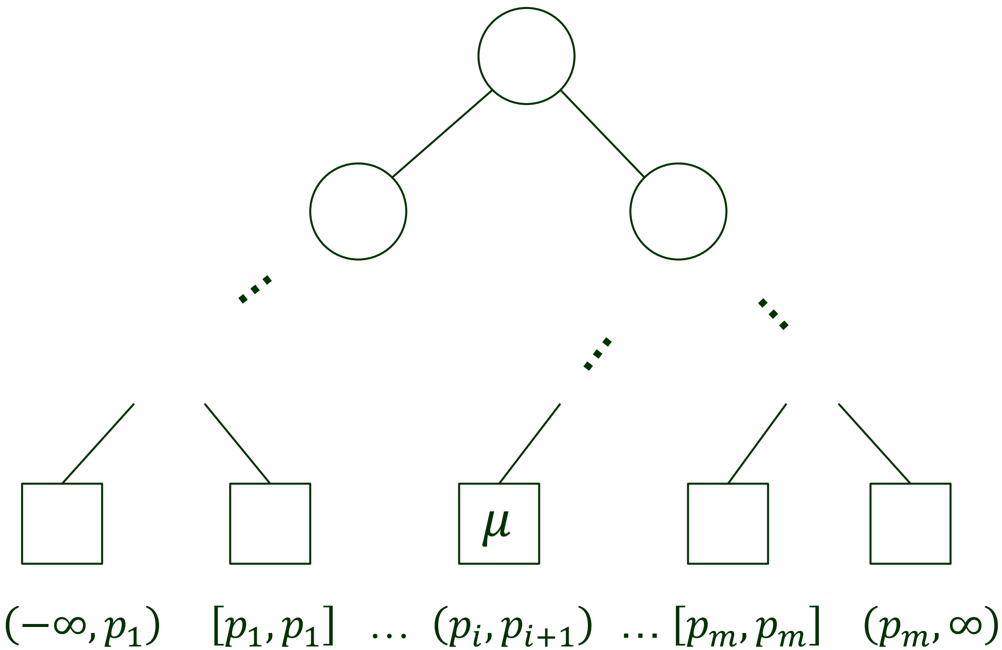
μ : a leaf

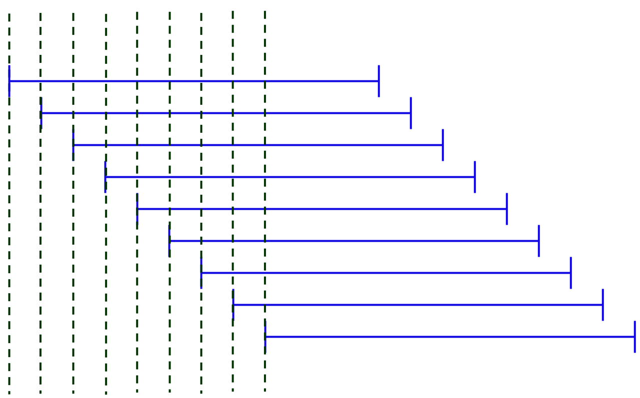
Int(μ): elementary interval corresponding to μ.

- Store at the leaf μ all the intervals in I that contain Int(μ).
- Query time

$$\underbrace{O(\log n}_{\text{BST search}} + \underbrace{k)}_{\text{\#reported intervals}}$$

Using a Binary Search Tree?





μ : a leaf
Int(μ): elementary interval corresponding to μ.

♠ Store at the leaf μ all the intervals in I that contain Int(μ).

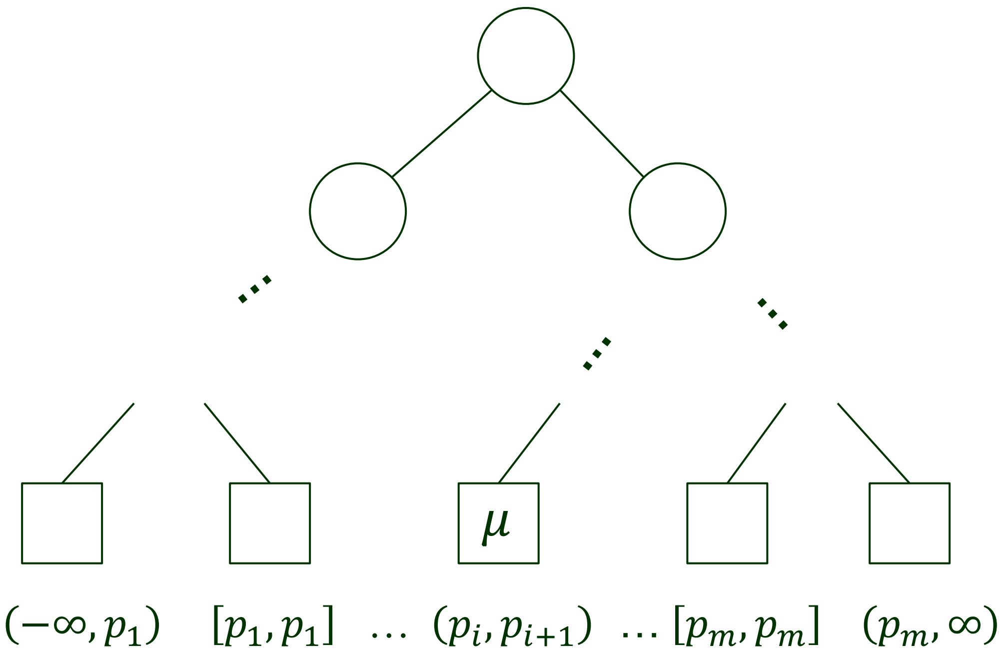
♠ Query time

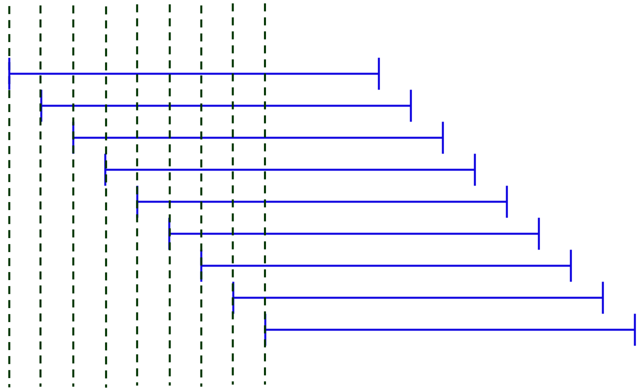
$$O(\log n + k)$$

BST search — #reported intervals

♠ High storage if the intervals overlap a lot.

Using a Binary Search Tree?





μ : a leaf

Int(μ): elementary interval corresponding to μ.

- Store at the leaf μ all the intervals in I that contain Int(μ).
- Query time

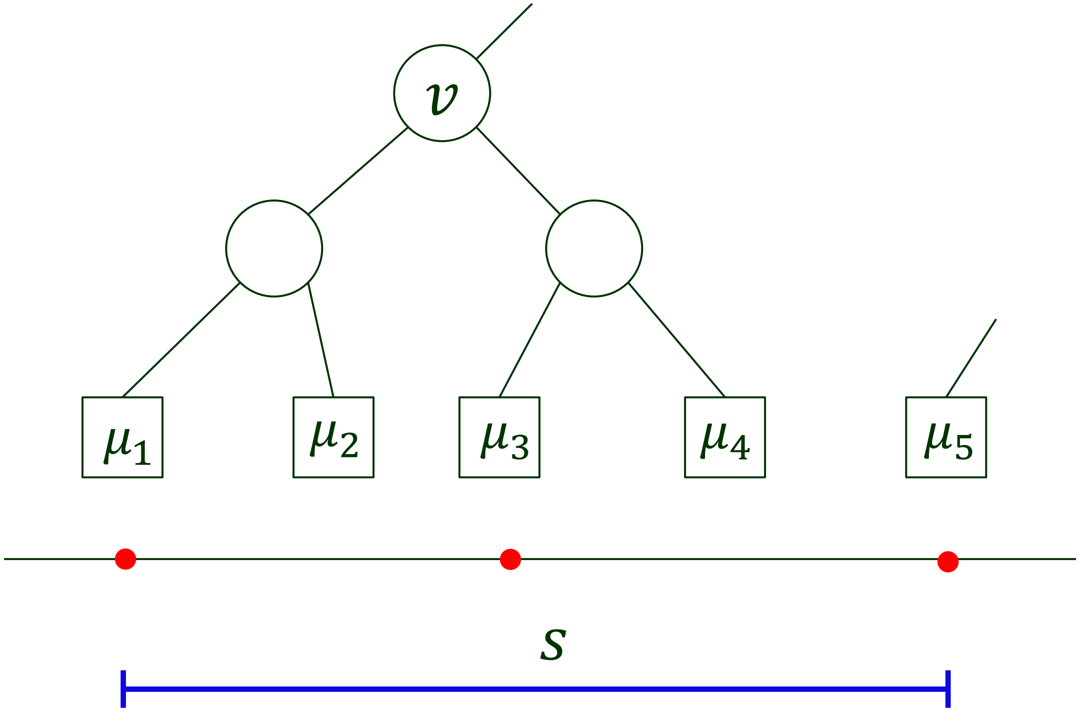
$$O(\log n + k)$$

BST search — #reported intervals

- High storage if the intervals overlap a lot.

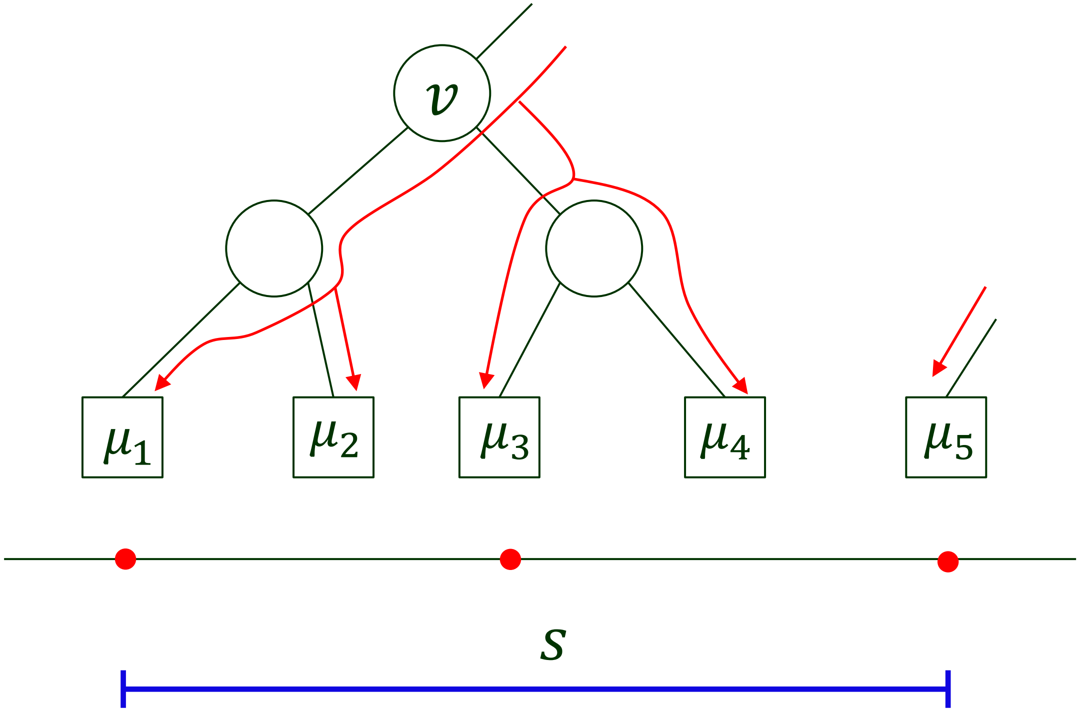
$O(n^2)$ possible!

Store an Interval As High as Possible



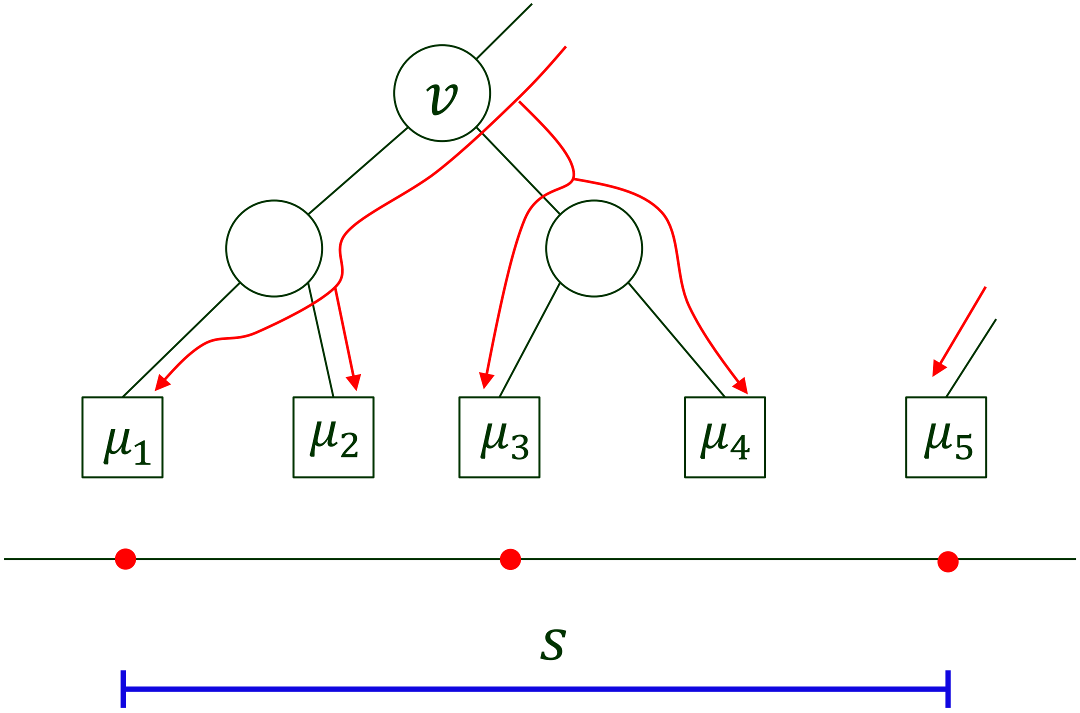
Interval s is stored five times at $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5$.

Store an Interval As High as Possible



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

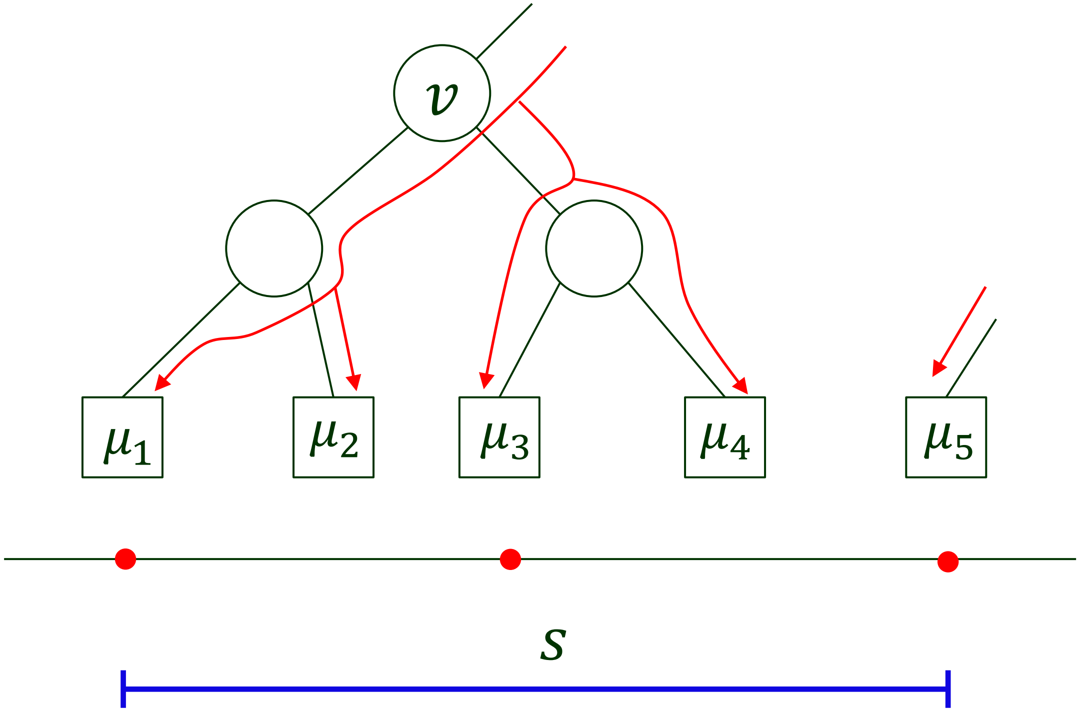
Store an Interval As High as Possible



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.

Store an Interval As High as Possible

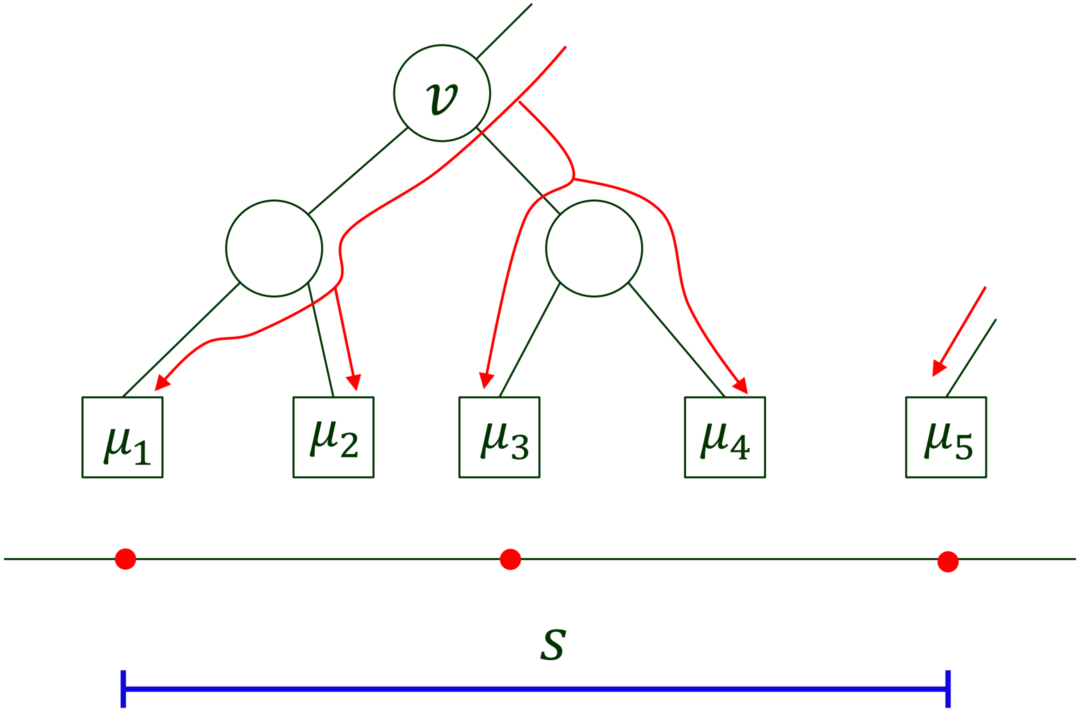


Interval s is stored five times at $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5$.

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 μ_5?

Store an Interval As High as Possible



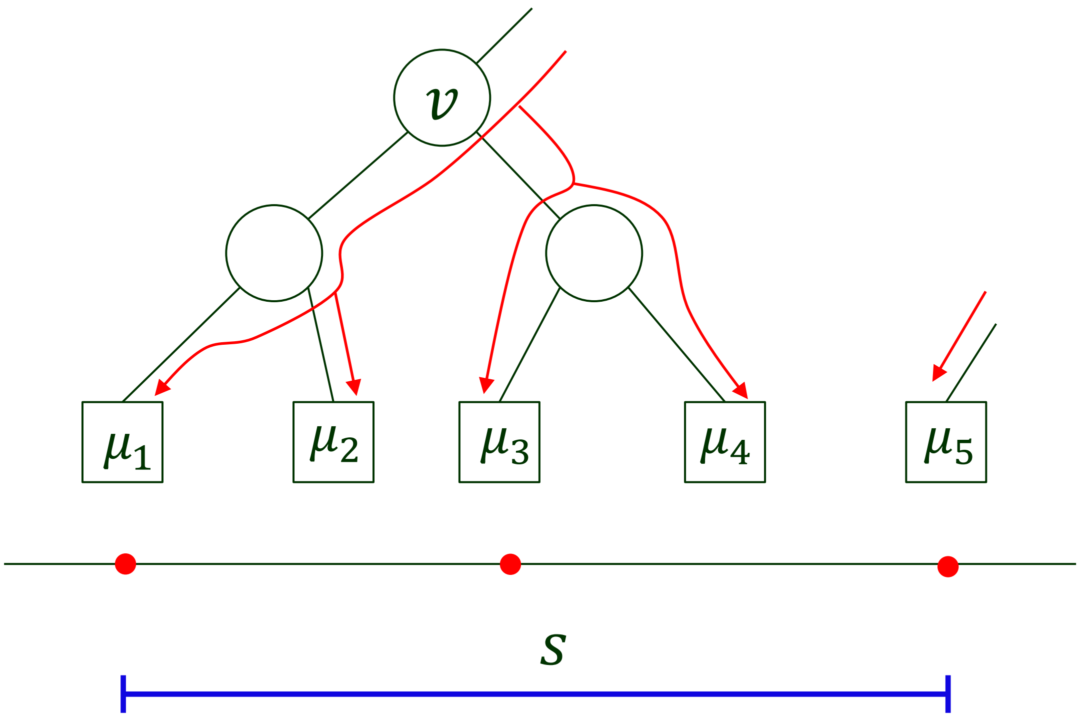
Interval s is stored five times at $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5$.

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 μ_5?

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

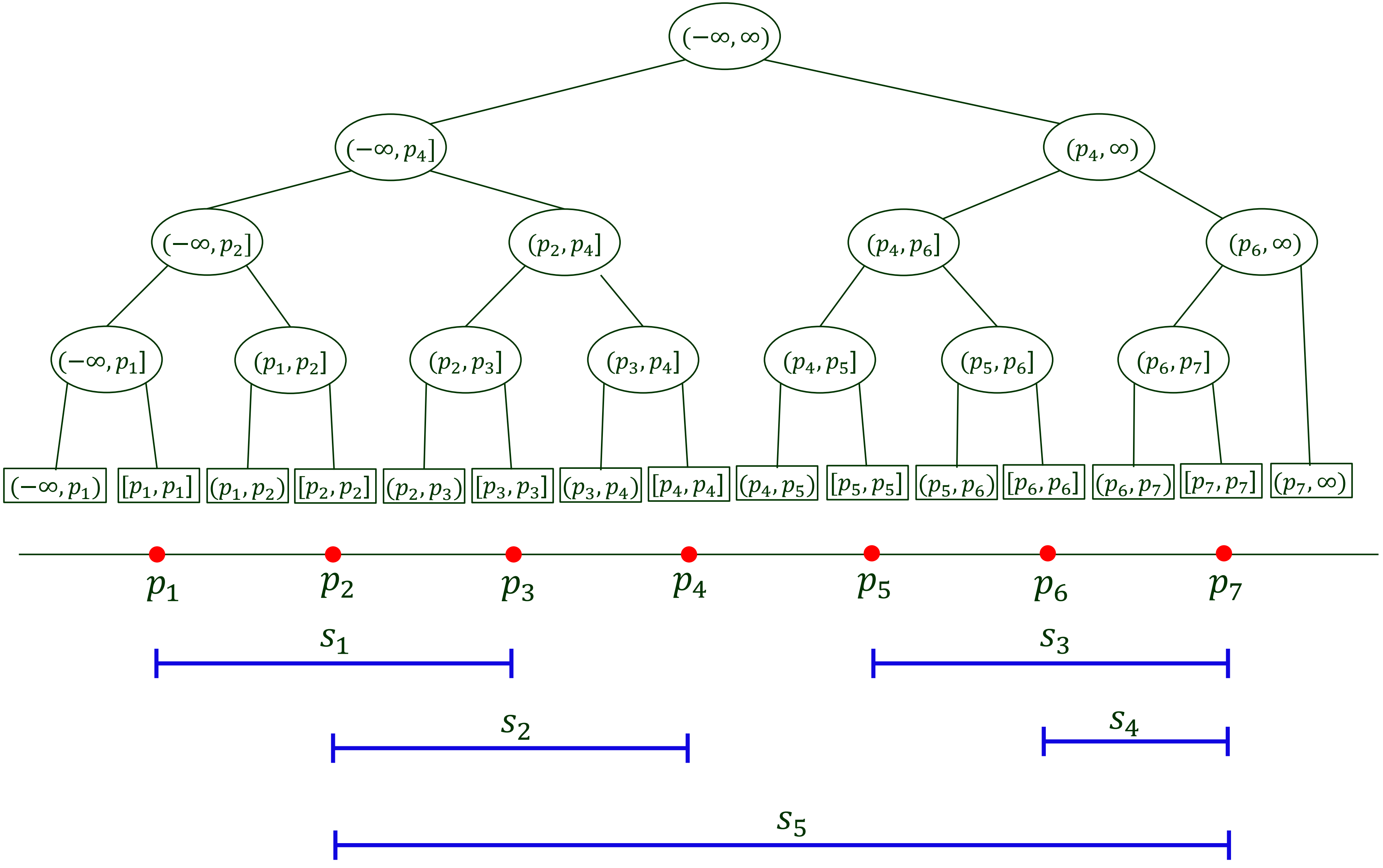
Why not store s only two times at v and μ_5?

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

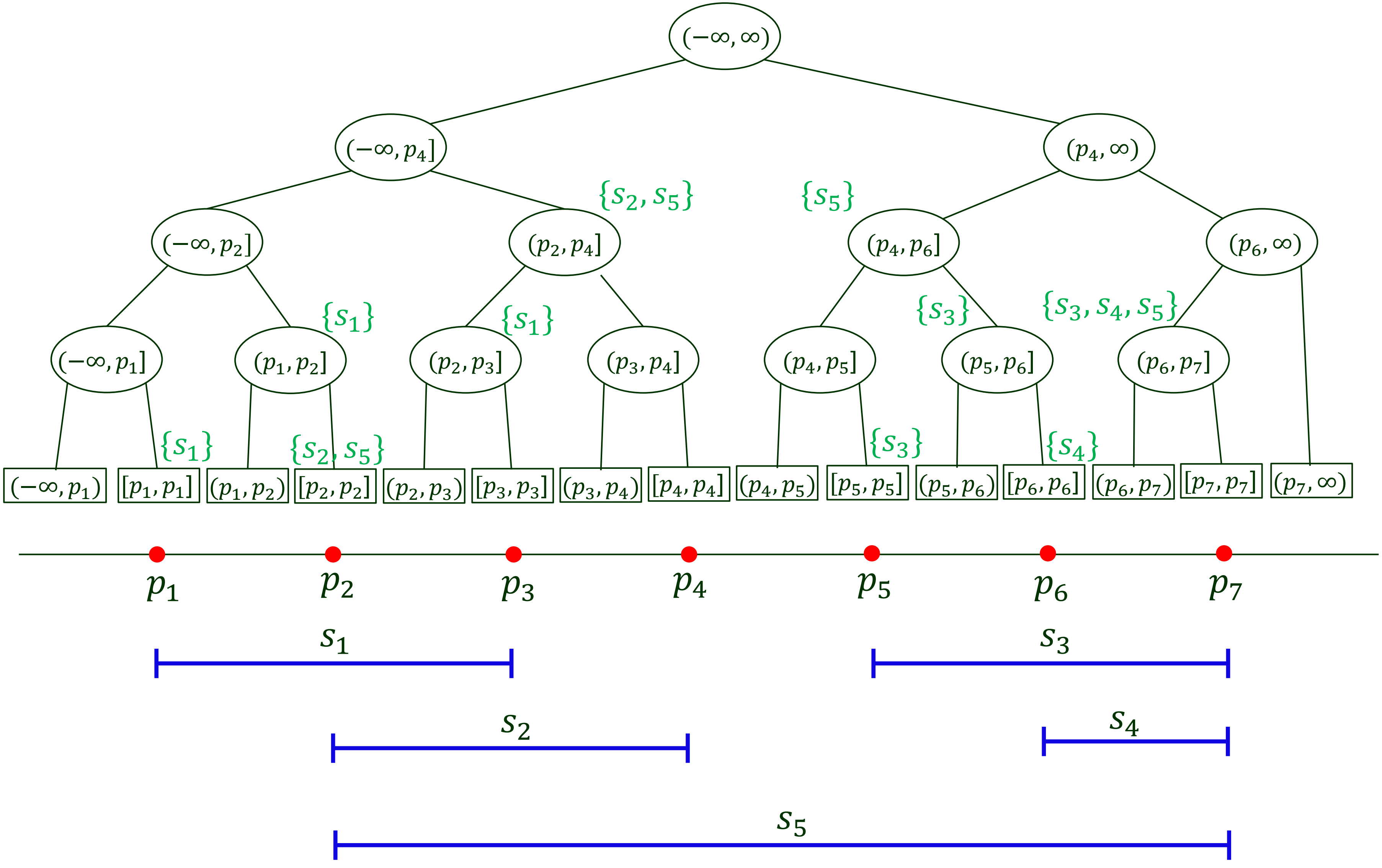
$\Downarrow$

These nodes must be as high as possible in the tree.

IV. Segment Tree

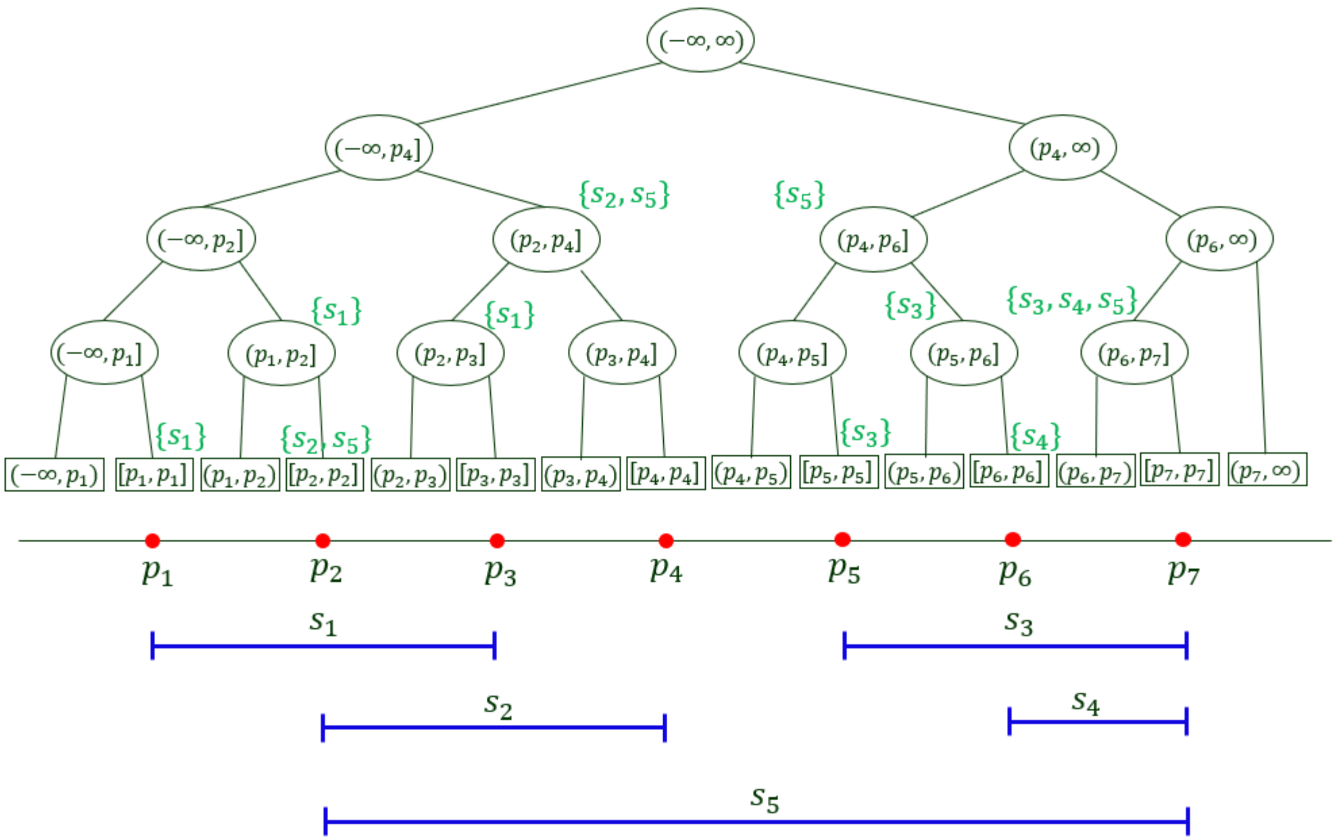


IV. Segment Tree



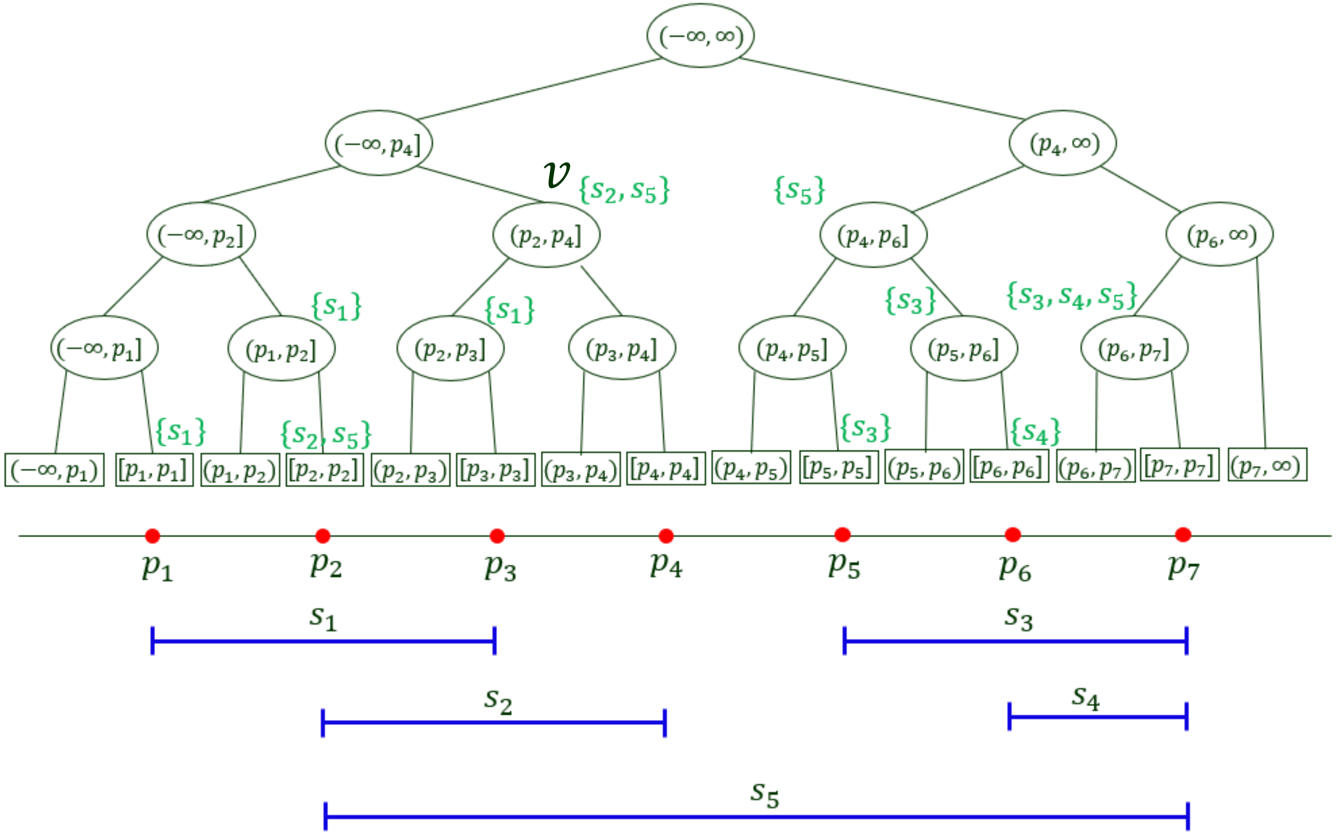
Tree Structure

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



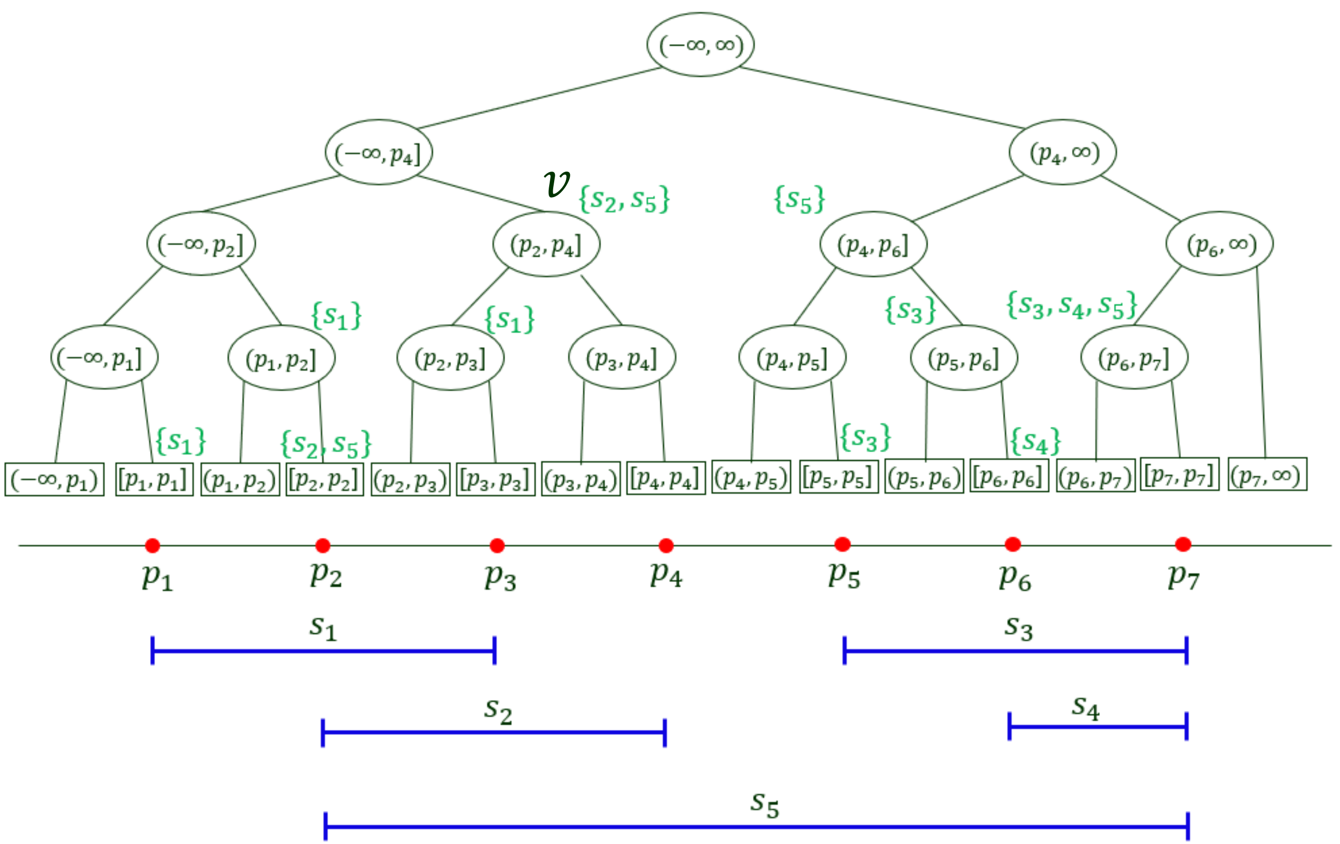
Tree Structure

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



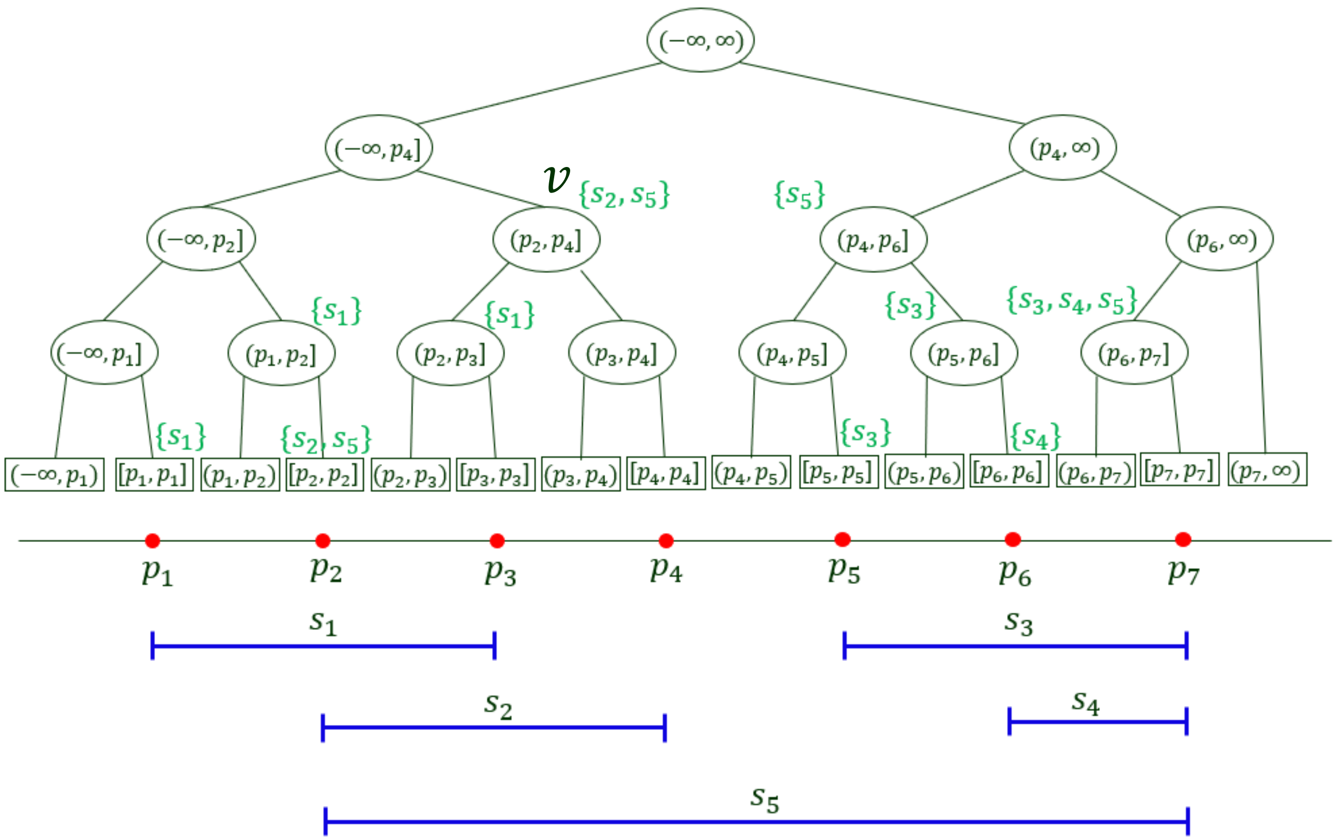
Tree Structure

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



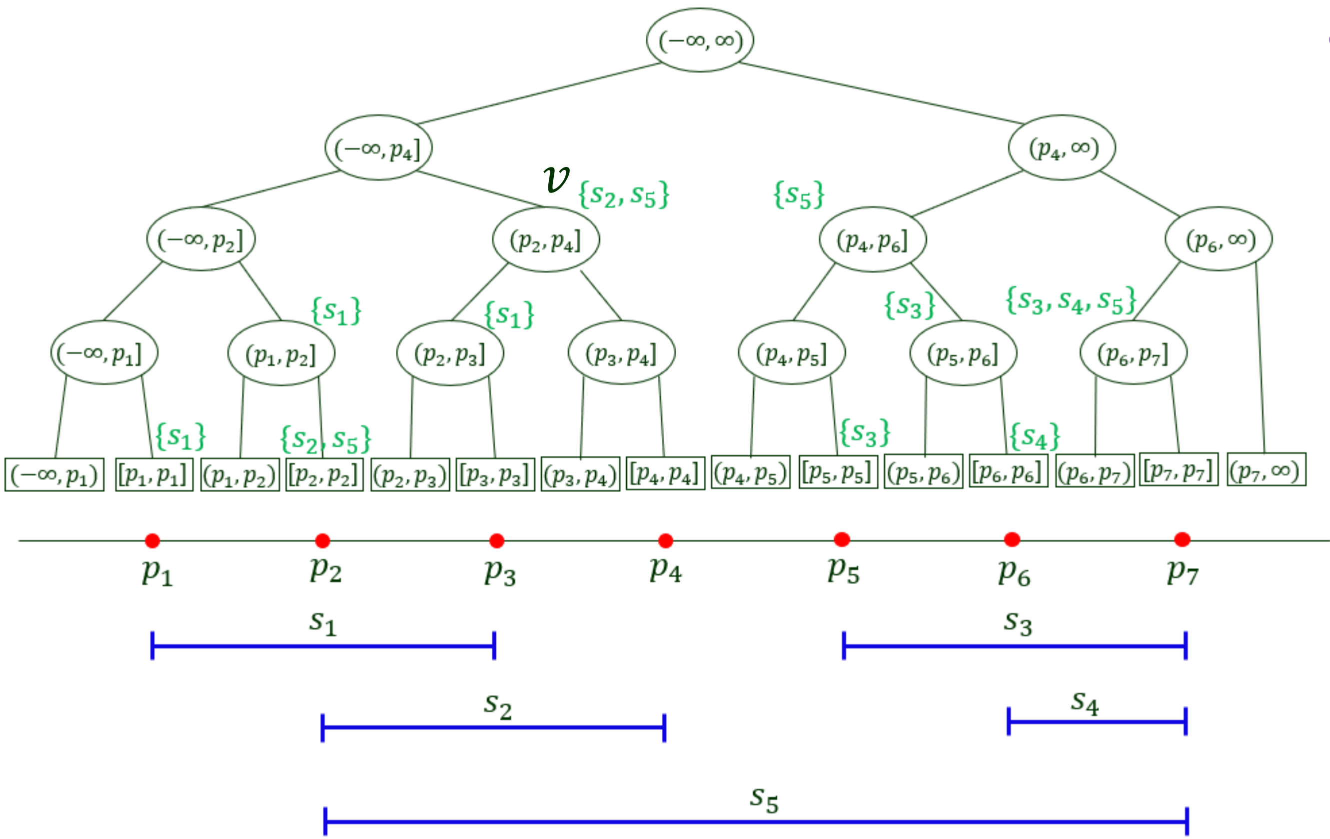
Tree Structure

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



Tree Structure

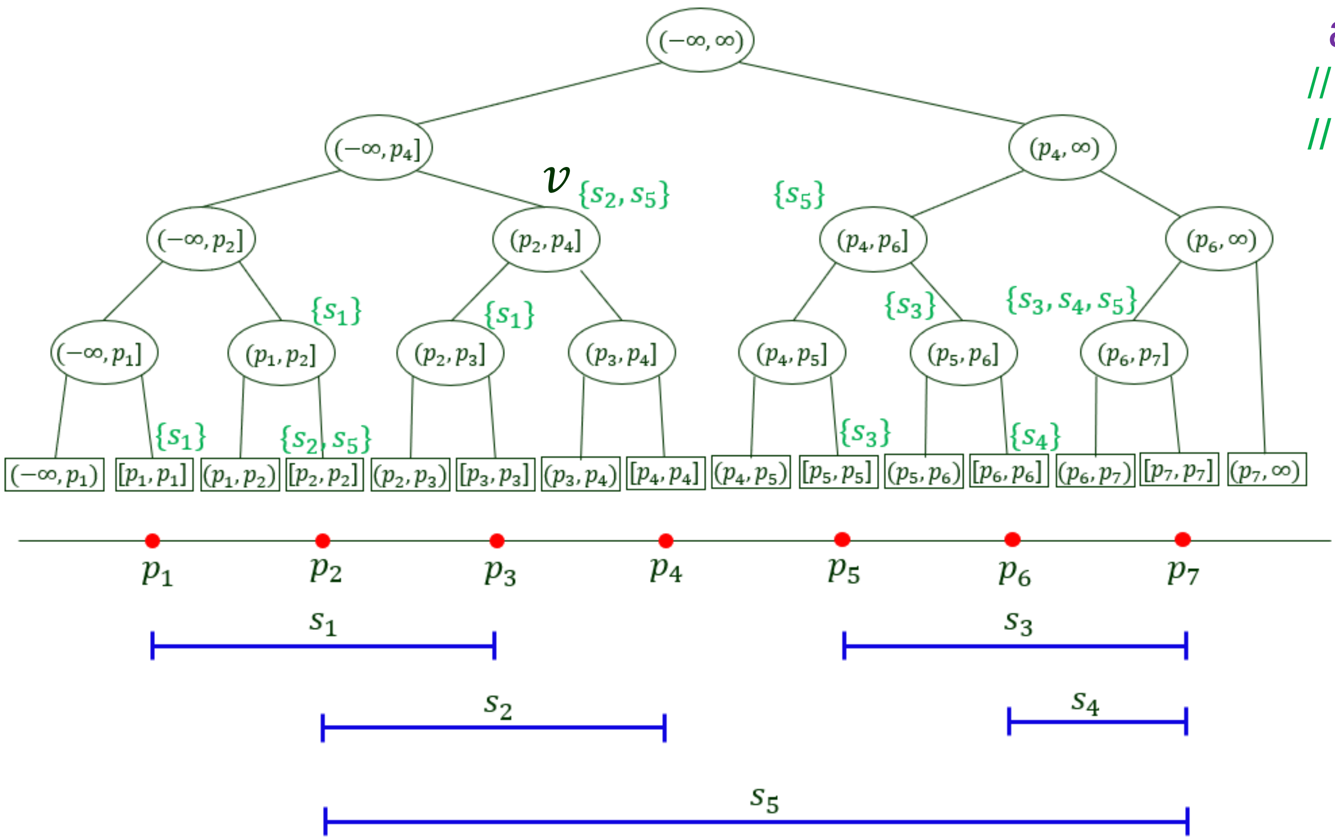
- Leaves ↔ elementary intervals (left-to-right)
- Internal node v ↔ union Int(v) of elementary intervals at the leaves in the subtree $\mathcal{T}(v)$ rooted at v.
- At every node or leaf v stores
  - Int(v)
  - the *canonical subset* (as a linked list): $C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq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



Tree Structure

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

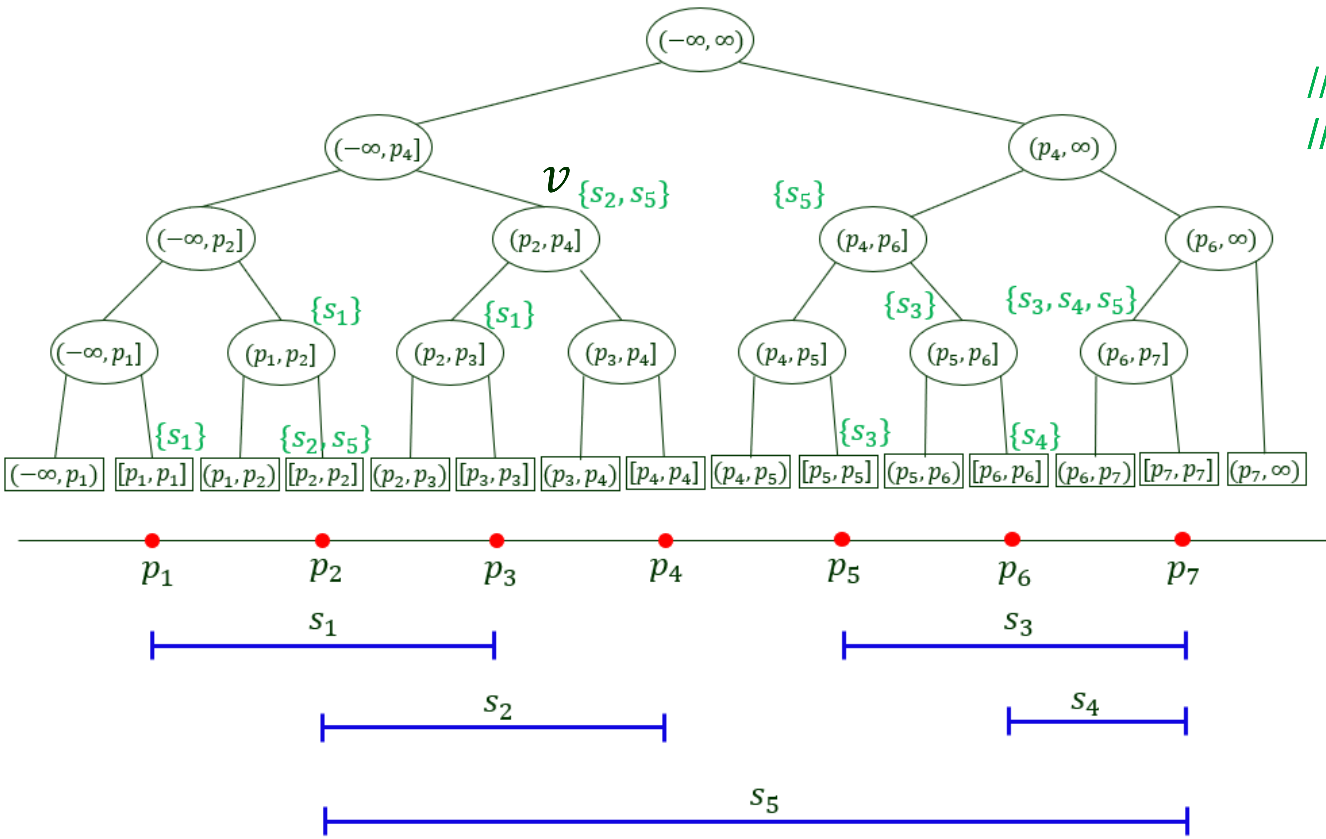
// intervals "covering" the node
// but not its parent node



Tree Structure

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

// intervals “covering” the node
// but not its parent node

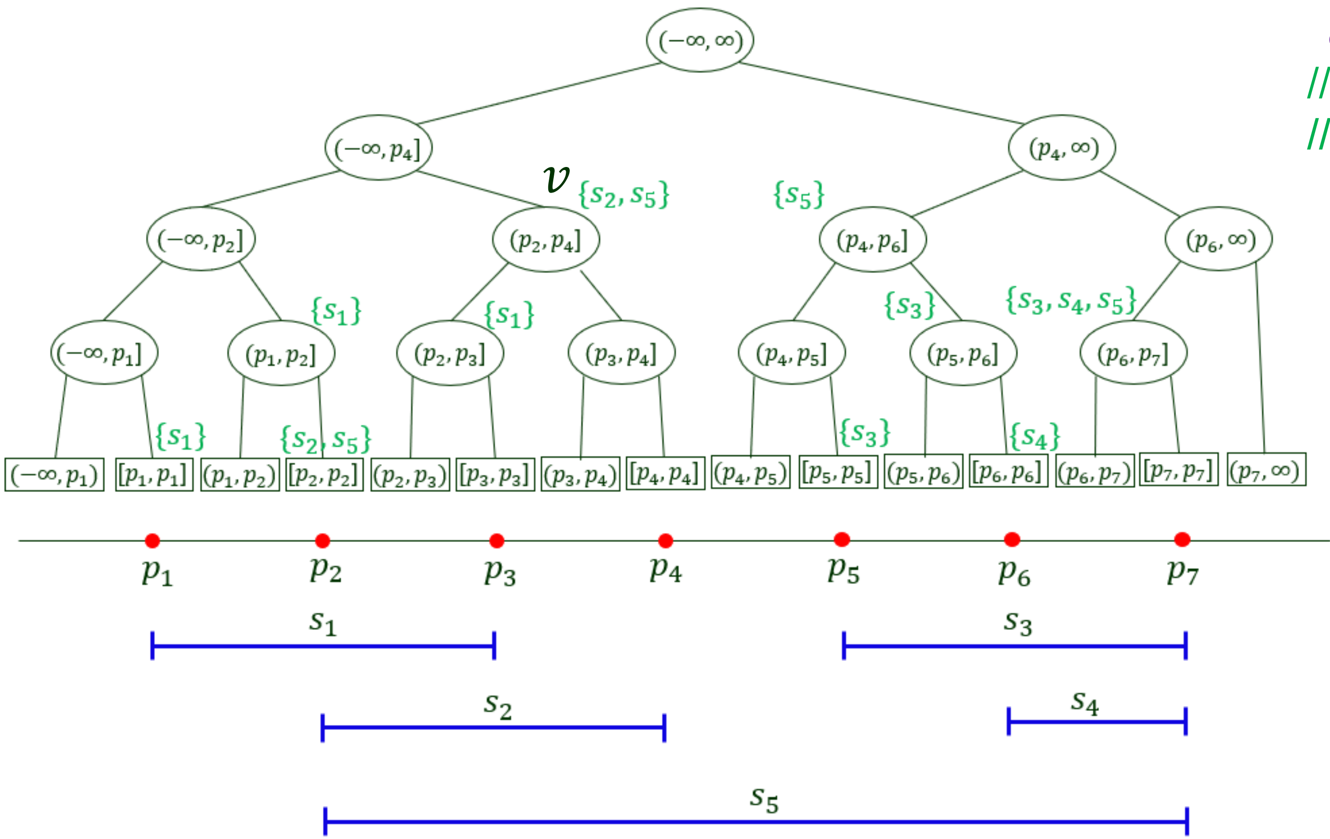


Store intervals at nodes as high as possible.

Tree Structure

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

// intervals “covering” the node
// but not its parent node



Store intervals at nodes as high as possible.

Examples

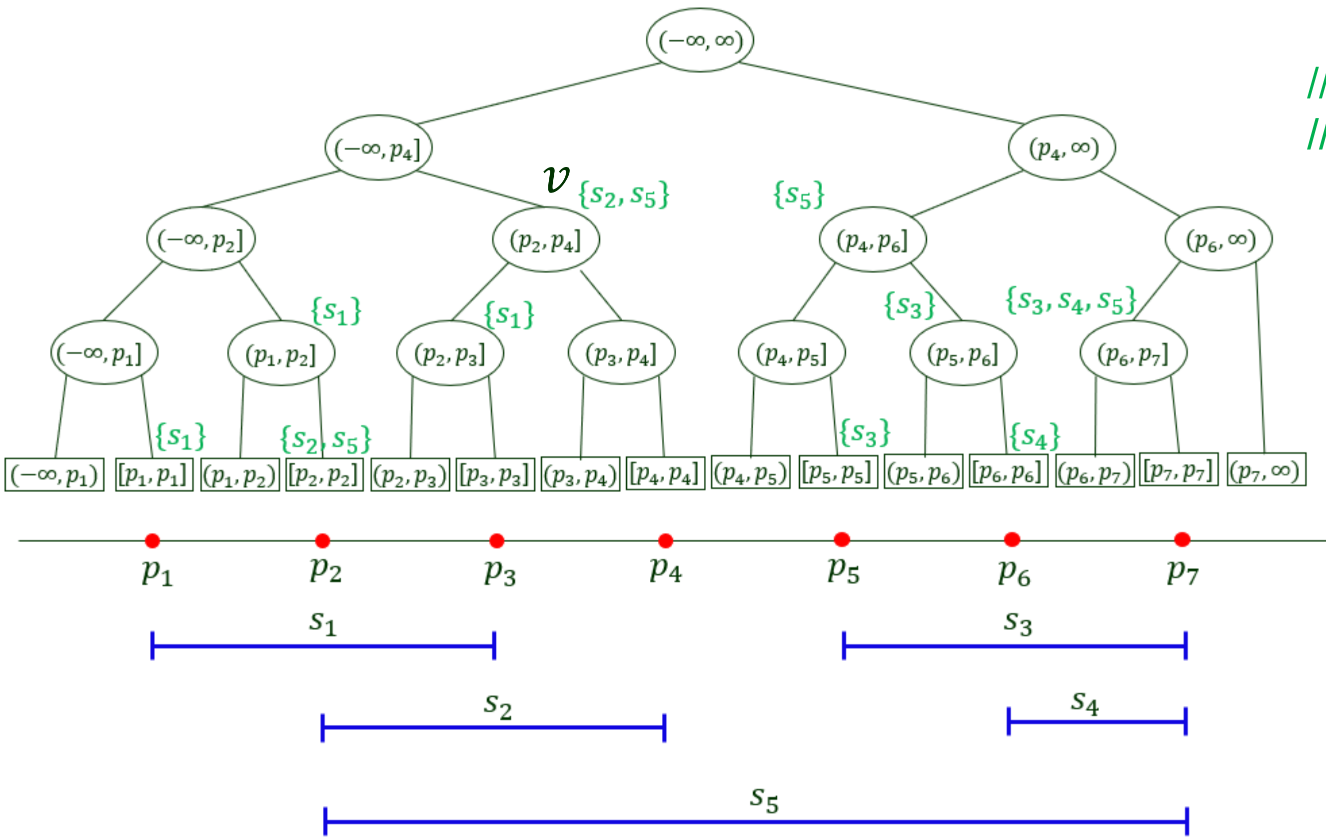
$\text{Int}(v) = [p_2, p_4] \subseteq s_2, s_5$

Tree Structure

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

// intervals "covering" the node
// but not its parent node

Store intervals at nodes as high as possible.



Examples

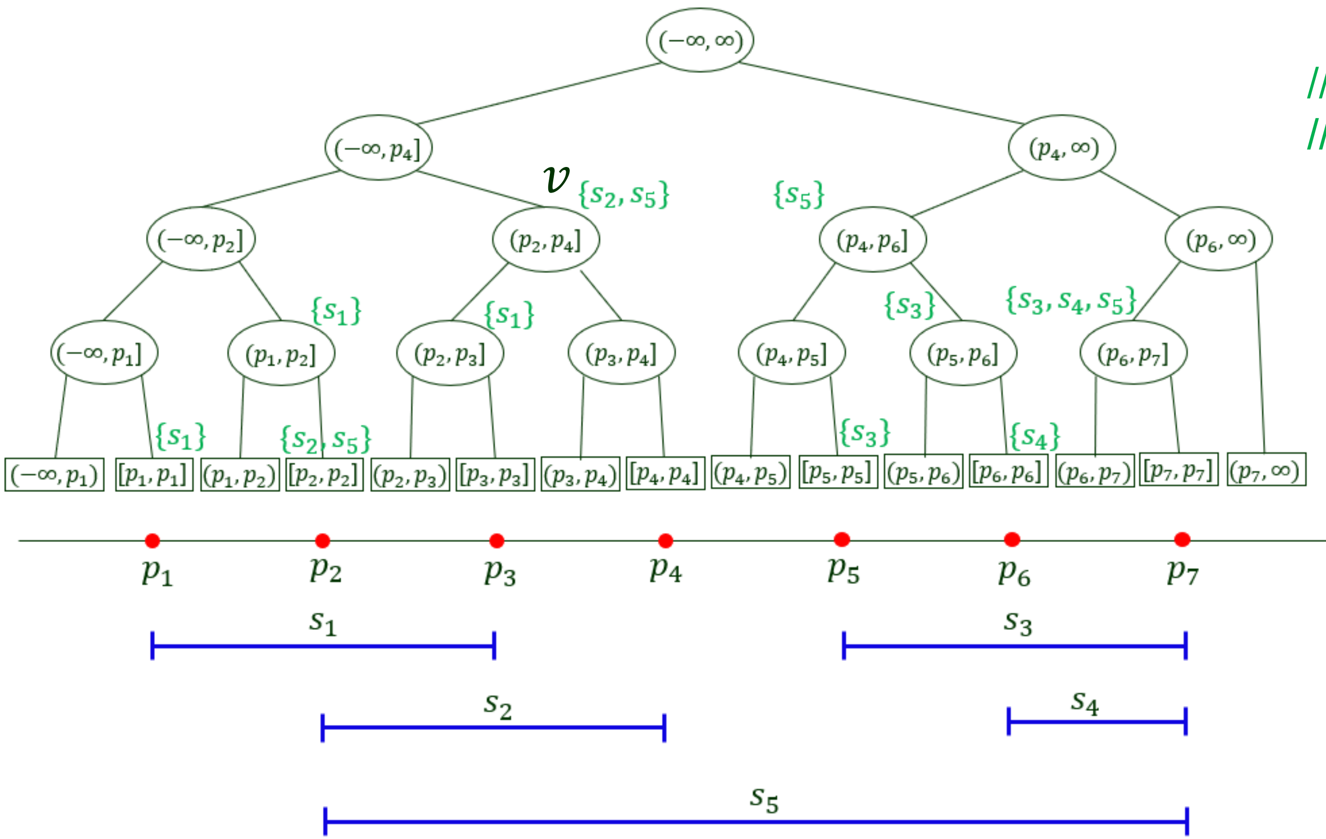
$\mathrm{Int}(v) = [p_2, p_4] \subseteq s_2, s_5$

$\mathrm{Int}(\mathrm{parent}(v)) = (-\infty, p_4]$
$\nsubseteq s_2, s_5$

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
  - $\text{Int}(v)$
  - the *canonical subset* (as a linked list): $C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x']$ and $\text{Int}(\text{parent}(v)) \not\subseteq [x, x']\}$

// intervals "covering" the node
// but not its parent node



Store intervals at nodes as high as possible.

Examples

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

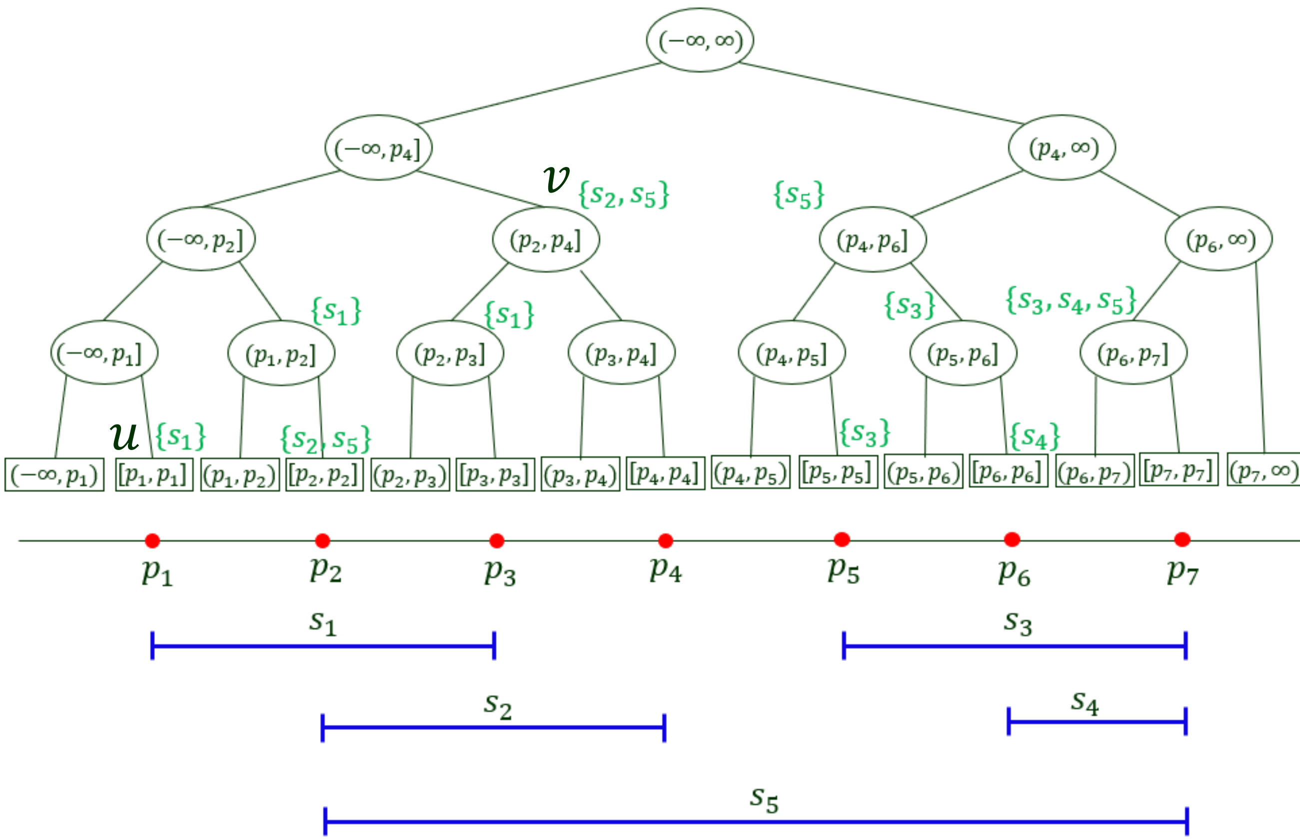
$\text{Int}(\text{parent}(v)) = (-\infty, p_4] \not\subseteq s_2, s_5$

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

Tree Structure

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

// intervals "covering" the node
// but not its parent node



Store intervals at nodes as high as possible.

Examples

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

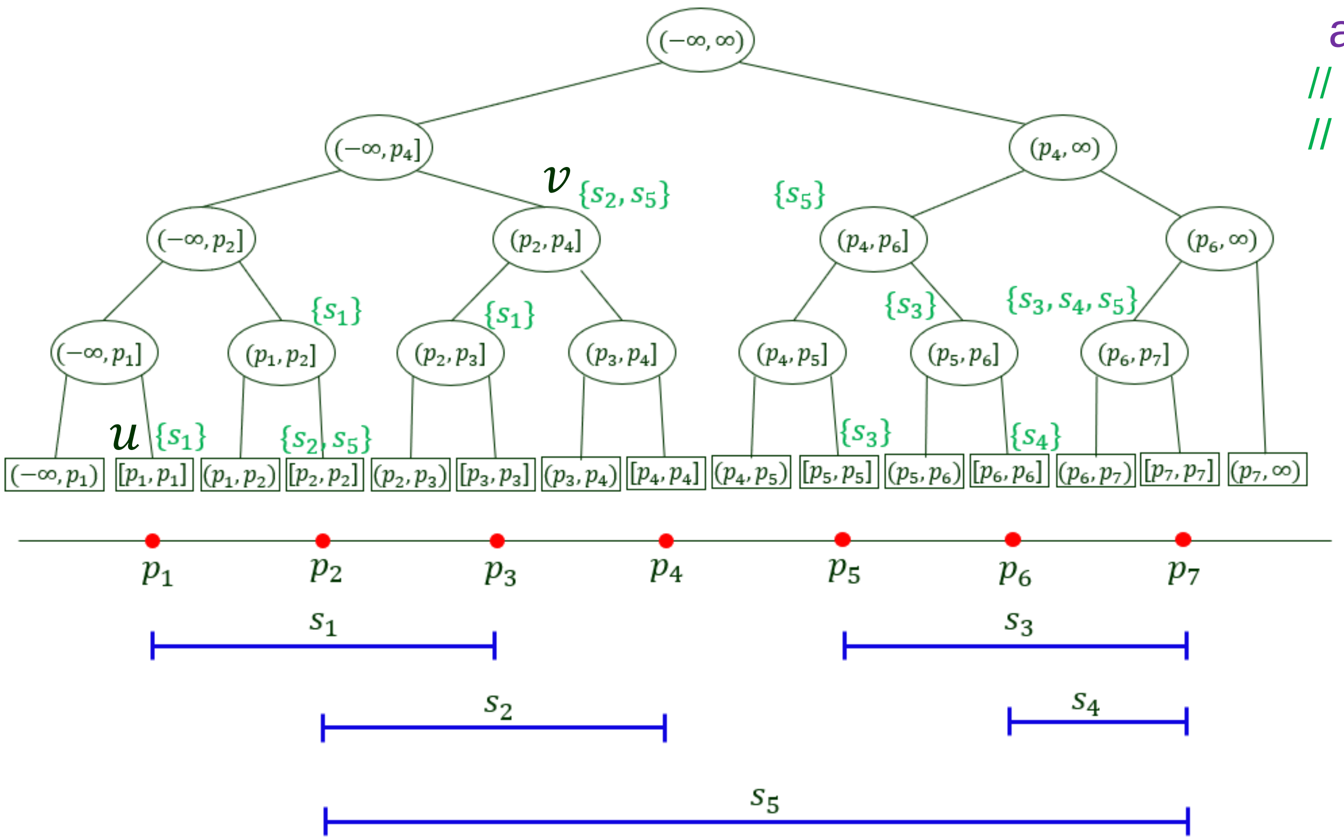
$\text{Int}(\text{parent}(v)) = (-\infty, p_4] \nsubseteq s_2, s_5$

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

Tree Structure

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

// intervals "covering" the node
// but not its parent node



Store intervals at nodes as high as possible.

Examples

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

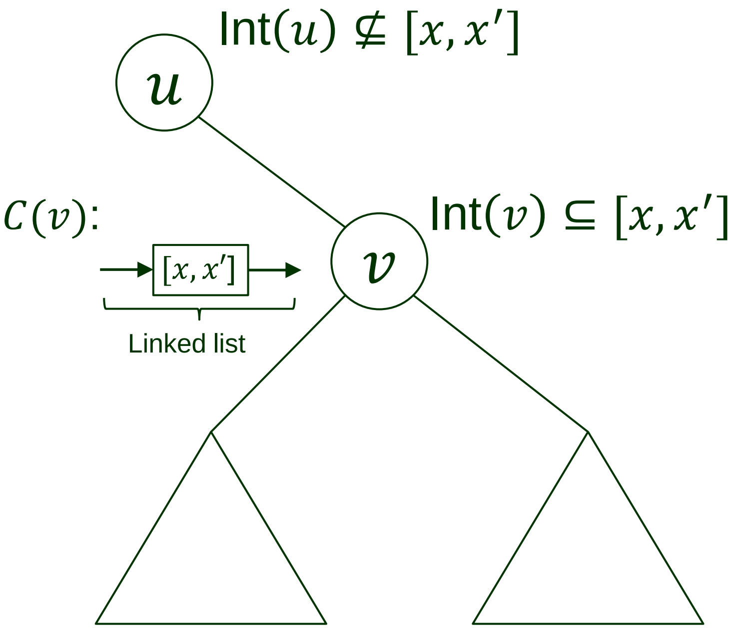
$\mathrm{Int}(\mathrm{parent}(v)) = (-\infty, p_4]$
$\not\subseteq s_2, s_5$

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

$C(u) = \{s_1\}$

Canonical Subset

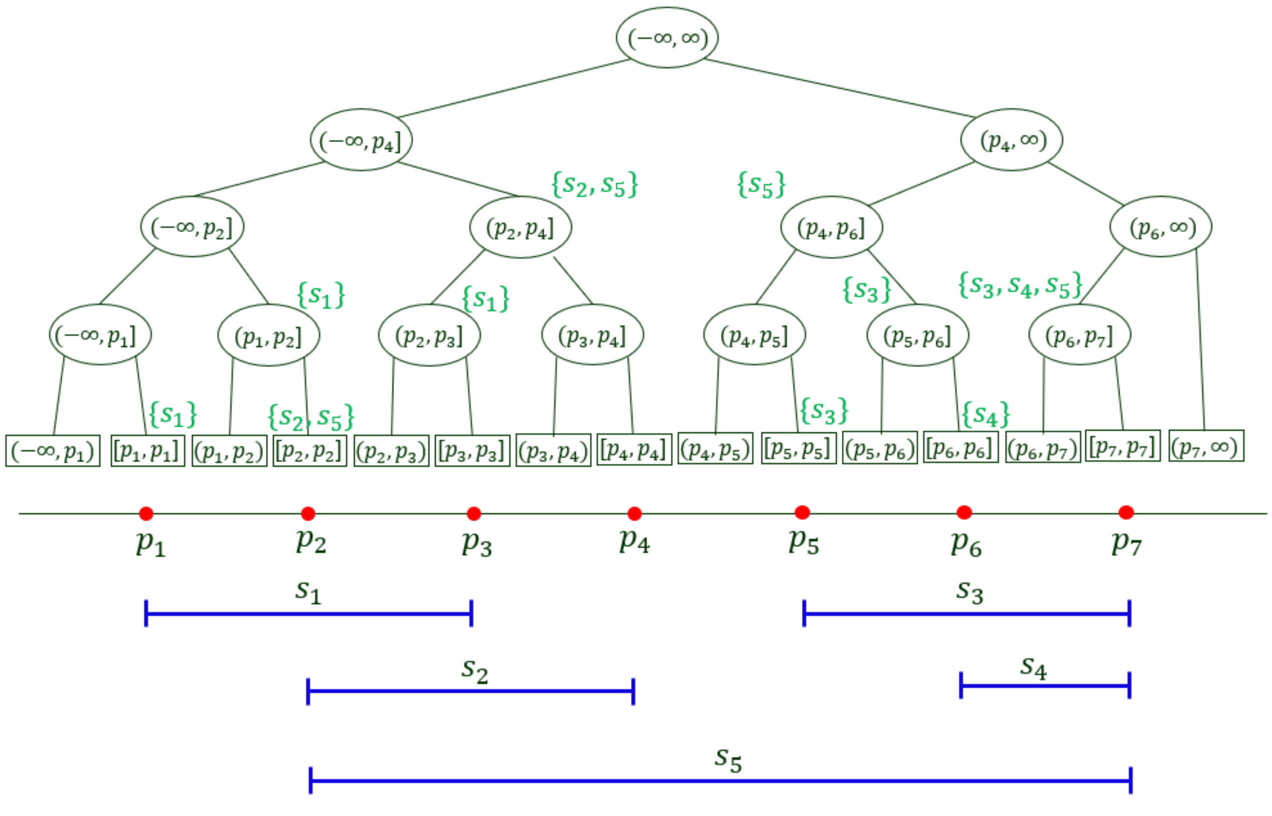
$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \nsubseteq [x, x']\}$



Leaf Node

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

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

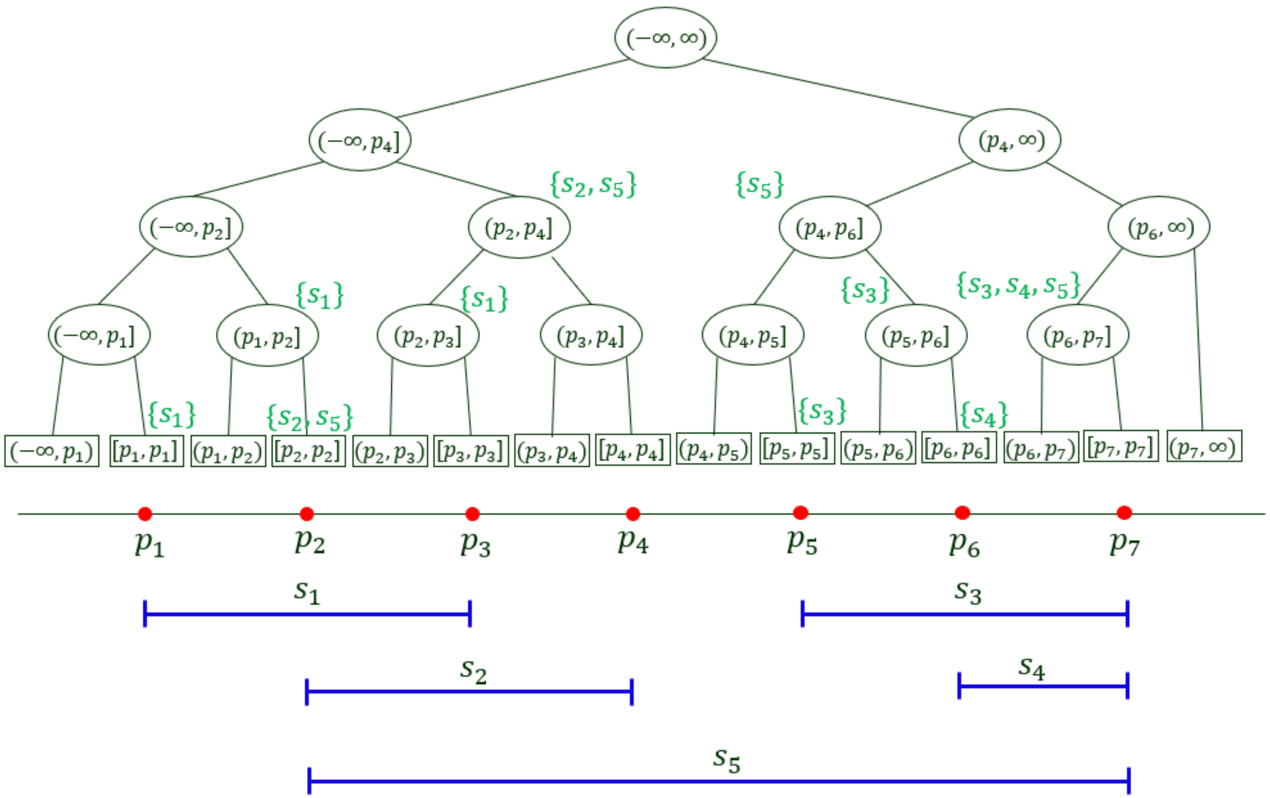


Leaf Node

$C(v) = \{[x, x'] \in I \mid \mathrm{Int}(v) \subseteq [x, x'] \text{ and } \mathrm{Int}(\mathrm{parent}(v)) \nsubseteq [x, x']\}$

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

(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.



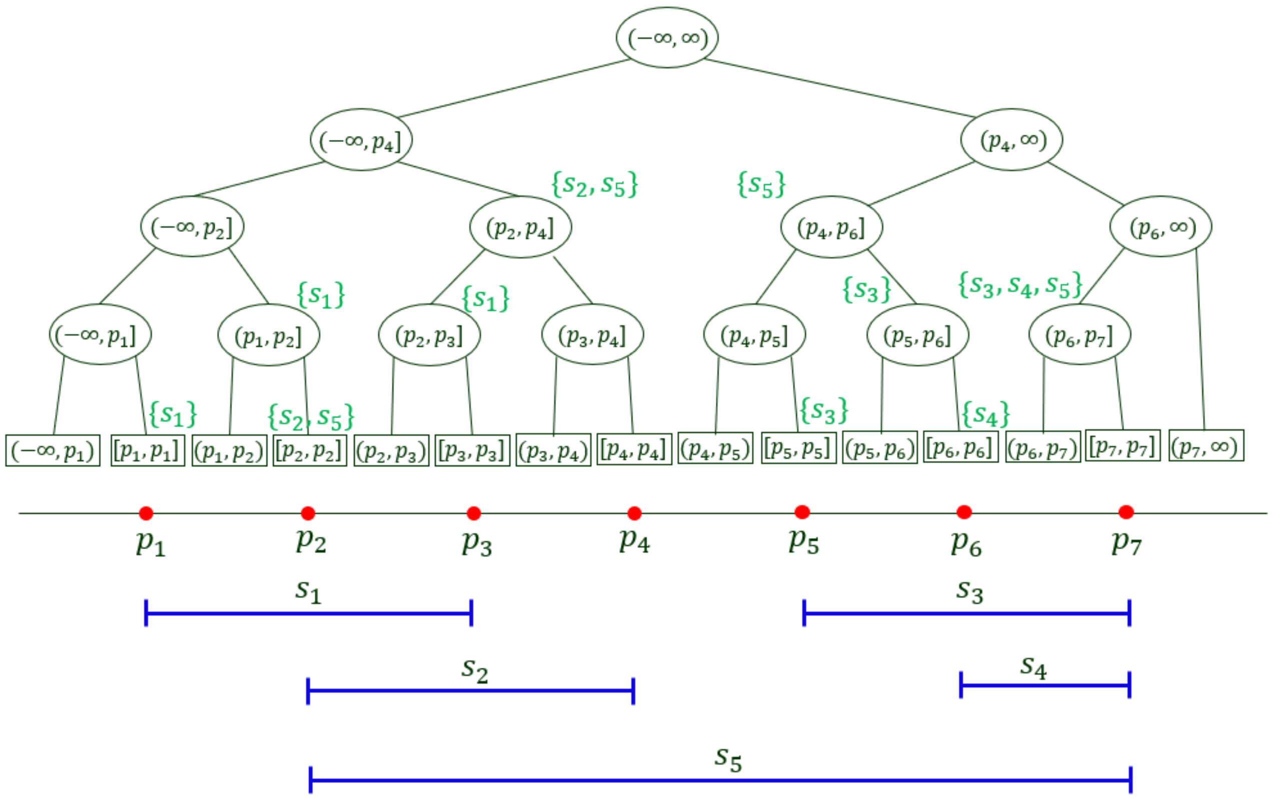
Leaf Node

$C(v) = \{[x, x'] \in I \mid \mathrm{Int}(v) \subseteq [x, x'] \text{ and } \mathrm{Int}(\mathrm{parent}(v)) \nsubseteq [x, x']\}$

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

(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.

- $\mathrm{Int}(\mu) = (p_i, p_{i+1})$?



Leaf Node

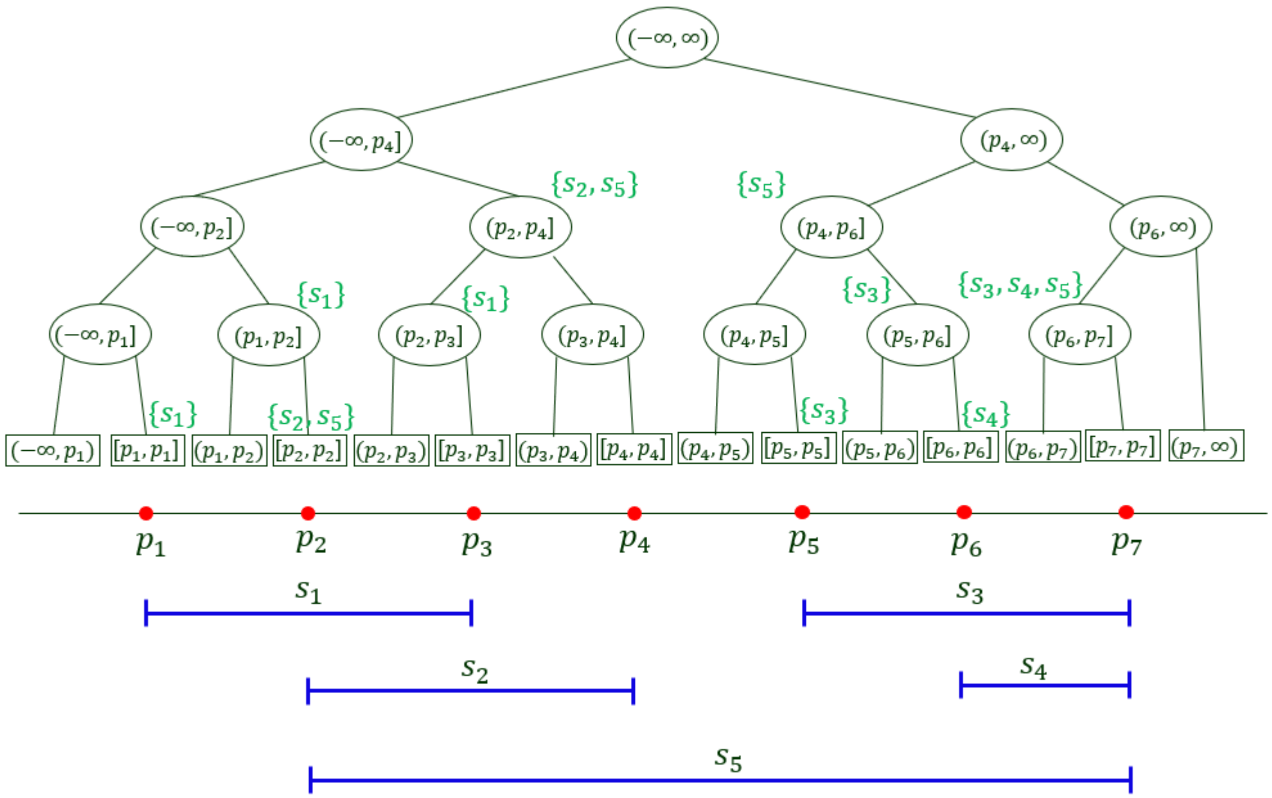
$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \nsubseteq [x, x']\}$

A leaf node μ 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.

(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.

- $\text{Int}(\mu) = (p_i, p_{i+1})$?

  Impossible because $(p_i, p_{i+1}) \subseteq [x, x']$



Leaf Node

$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and Int(parent}(v)) \nsubseteq [x, x']\}$

A leaf node μ 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.

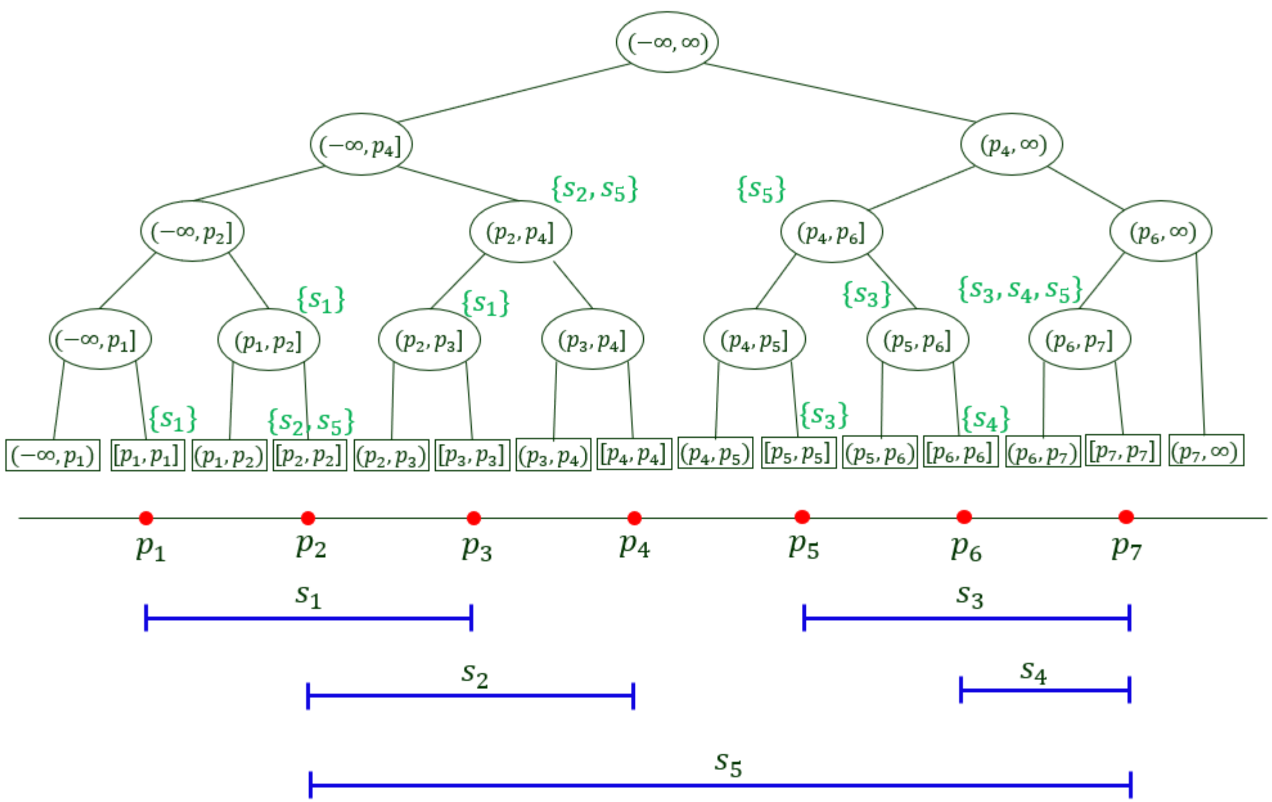
(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.

- $\text{Int}(\mu) = (p_i, p_{i+1})$?

  Impossible because $(p_i, p_{i+1}) \subseteq [x, x']$

  $\Downarrow$

  $\text{Int(parent}(\mu))$
  $= (p_i, p_{i+1}] \subseteq [x, x']$

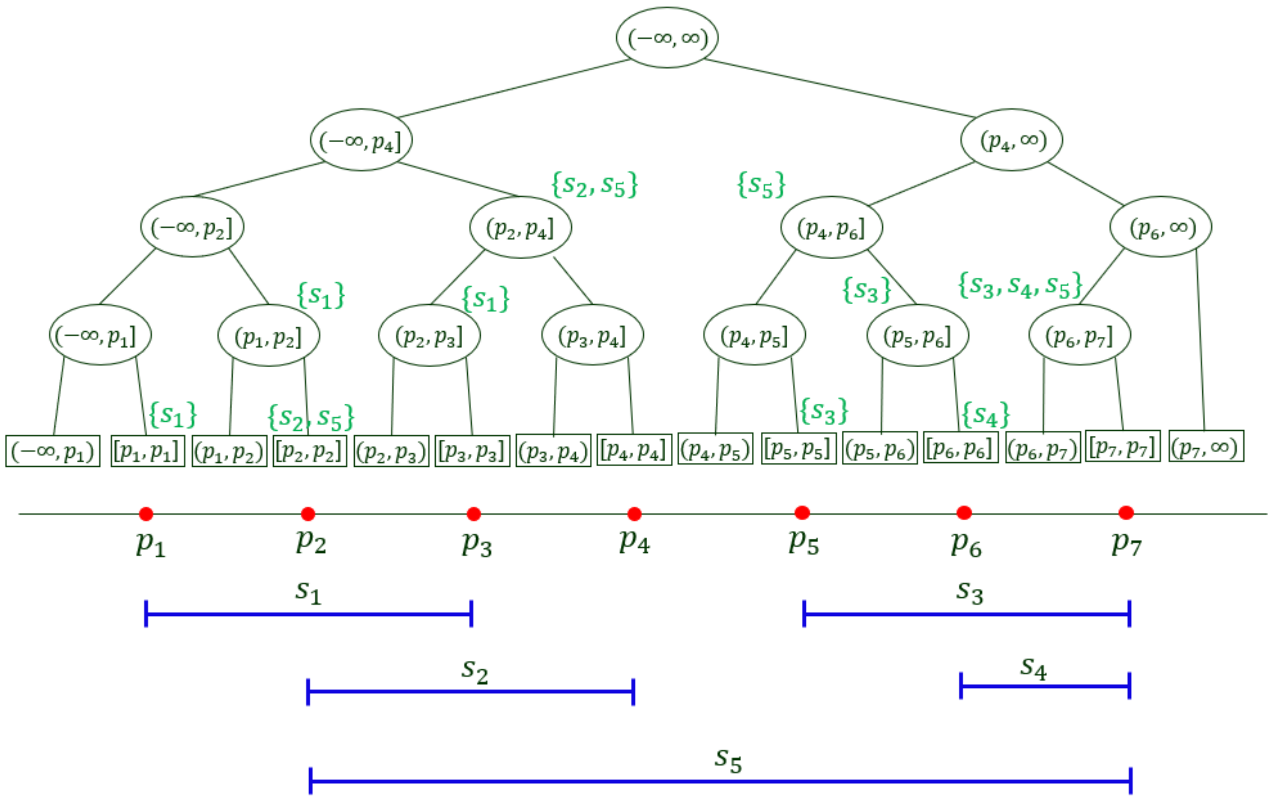


Leaf Node

$C(v) = \{[x,x'] \in I \mid \text{Int}(v) \subseteq [x,x'] \text{ and } \text{Int}(\text{parent}(v)) \nsubseteq [x,x']\}$

A leaf node μ 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.

(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.



- $\text{Int}(\mu) = (p_i, p_{i+1})$?

  Impossible because
  $(p_i, p_{i+1}) \subseteq [x, x']$

  $\Downarrow$

  $\text{Int}(\text{parent}(\mu))$
  $= (p_i, p_{i+1}] \subseteq [x, x']$

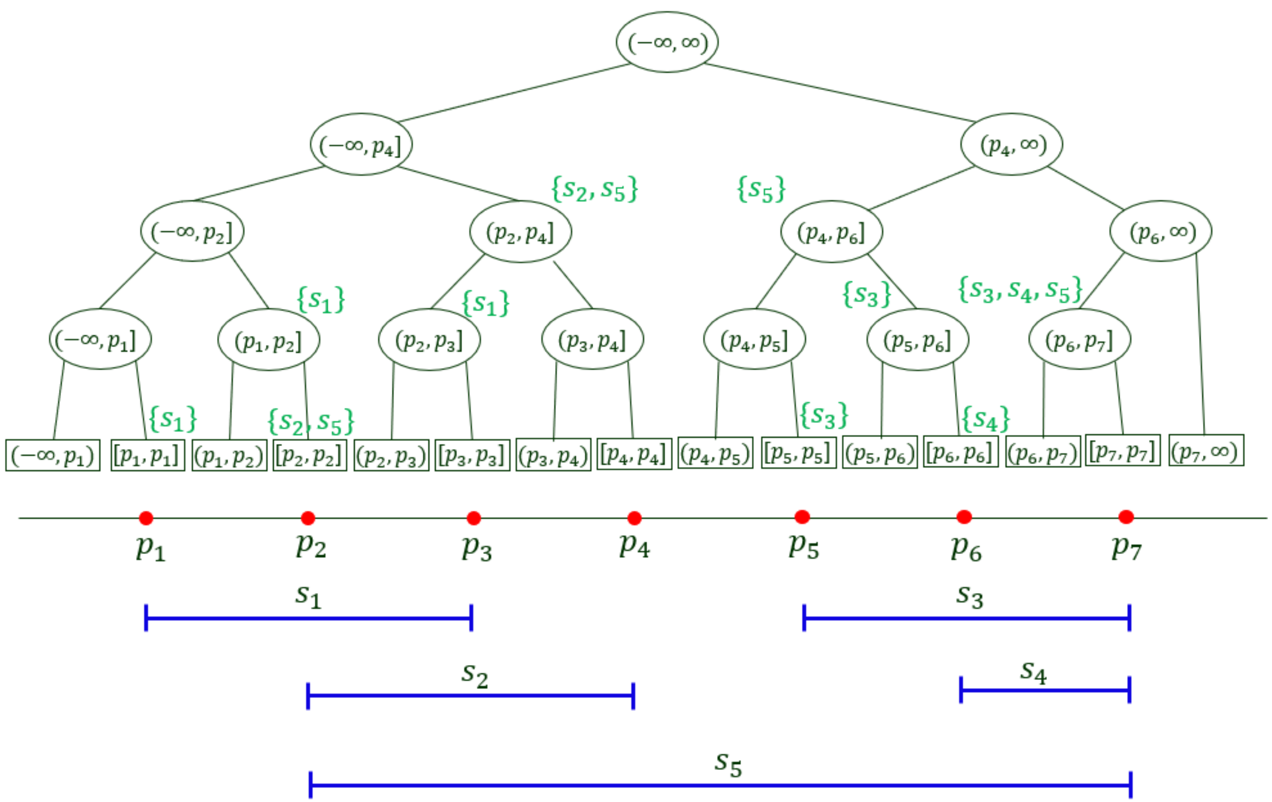
- $\text{Int}(\mu) = [p_i, p_i]$ is a right endpoint of some $[x, x']$?

Leaf Node

$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \nsubseteq [x, x']\}$

A leaf node μ 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.

(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.



- $\text{Int}(\mu) = (p_i, p_{i+1})$?

  Impossible because $(p_i, p_{i+1}) \subseteq [x, x']$

  $$\Downarrow$$

  $\text{Int}(\text{parent}(\mu))$
  $= (p_i, p_{i+1}] \subseteq [x, x']$

- $\text{Int}(\mu) = [p_i, p_i]$ is a right endpoint of some $[x, x']$?

  $$\Downarrow$$

  $x \leq p_{i-1} < x' = p_i$

Leaf Node

$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and Int(parent}(v)) \nsubseteq [x, x']\}$

A leaf node μ 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.

(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.

- $\text{Int}(\mu) = (p_i, p_{i+1})$?

  Impossible because $(p_i, p_{i+1}) \subseteq [x, x']$

  $\Downarrow$

  $\text{Int(parent}(\mu))$ $= (p_i, p_{i+1}] \subseteq [x, x']$

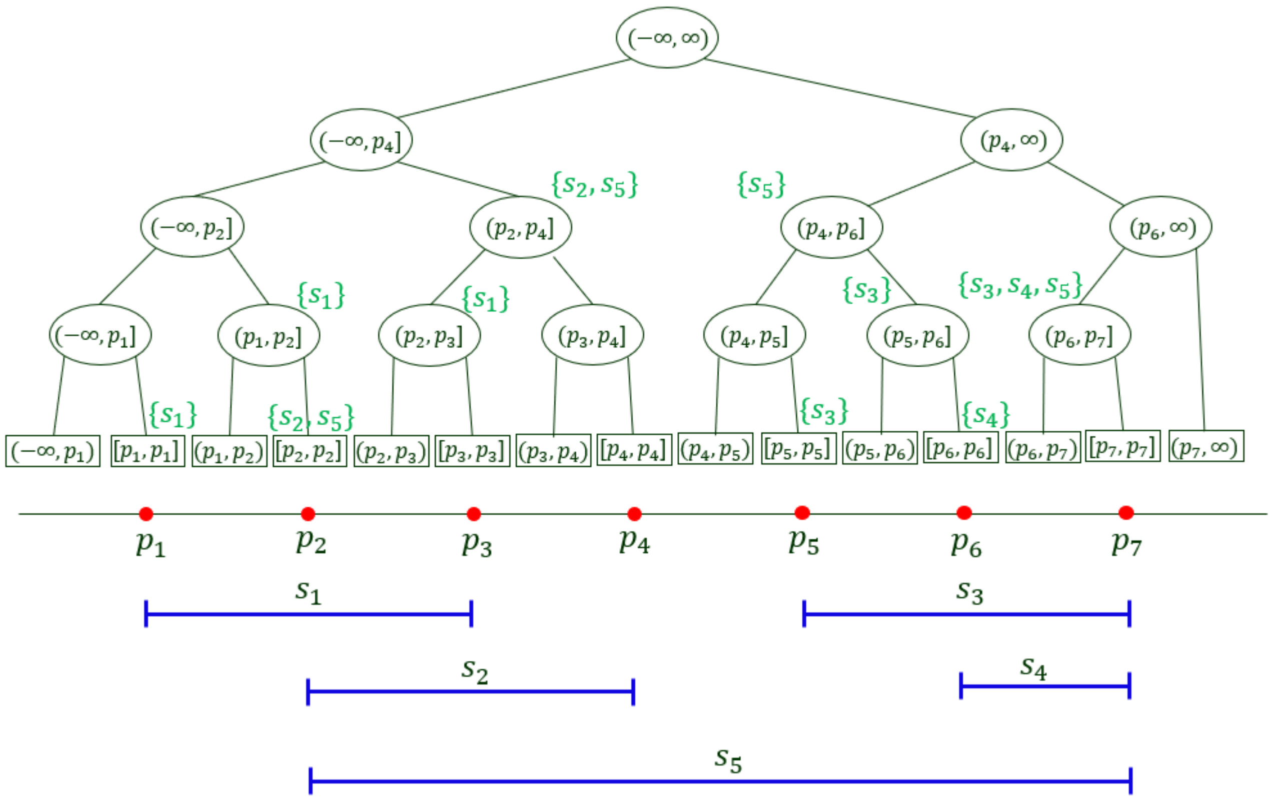
- $\text{Int}(\mu) = [p_i, p_i]$ is a right endpoint of some $[x, x']$?

  $\Downarrow$

  $x \leq p_{i-1} < x' = p_i$

  $\Downarrow$

  $\text{Int(parent}(\mu))$ $= (p_{i-1}, p_i] \subseteq [x, x']$

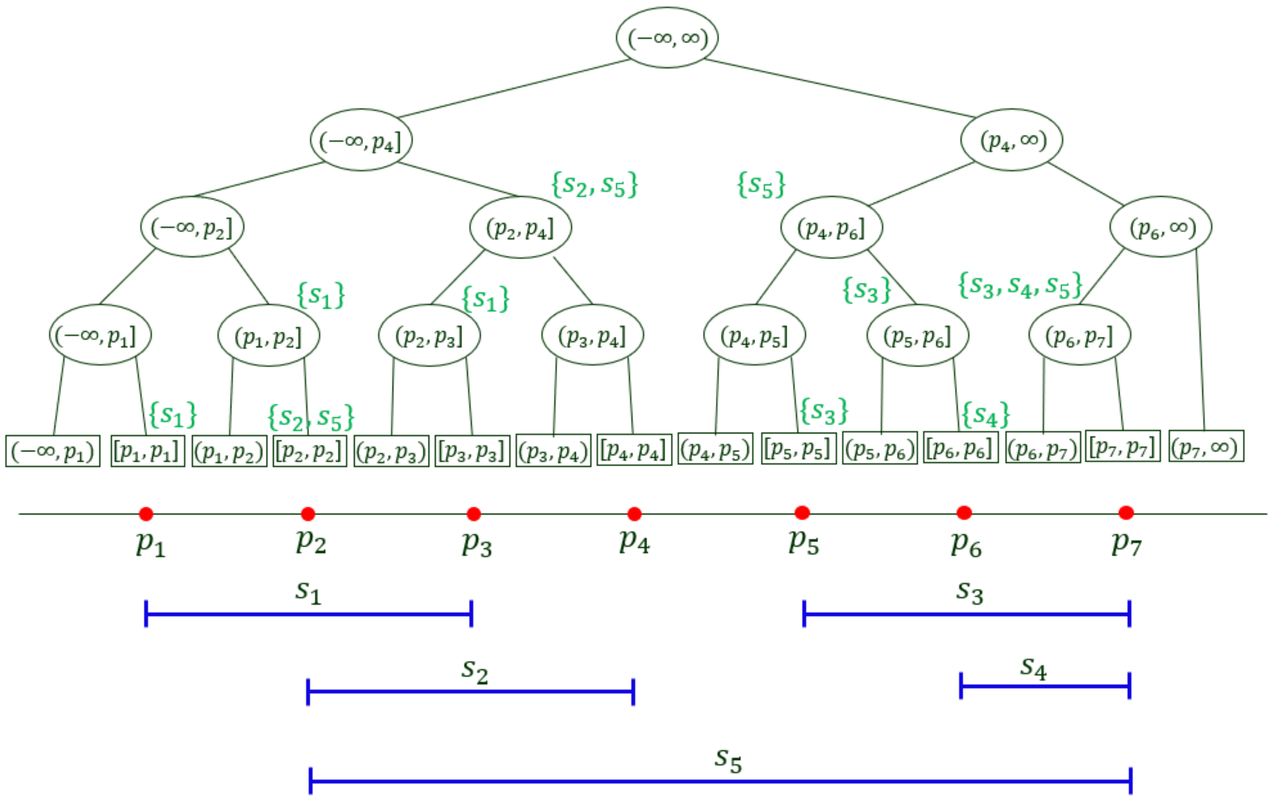


Leaf Node

$C(v) = \{[x, x'] \in I \mid \text{Int}(v) \subseteq [x, x'] \text{ and } \text{Int}(\text{parent}(v)) \nsubseteq [x, x']\}$

A leaf node μ 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.

(Proof of necessity) Suppose $C(\mu) \neq \emptyset$.



- $\text{Int}(\mu) = (p_i, p_{i+1})$?

  Impossible because $(p_i, p_{i+1}) \subseteq [x, x']$

  $\Downarrow$

  $\text{Int}(\text{parent}(\mu)) = (p_i, p_{i+1}] \subseteq [x, x']$

- $\text{Int}(\mu) = [p_i, p_i]$ is a right endpoint of some $[x, x']$?

  $\Downarrow$

  $x \le p_{i-1} < x' = p_i$

  $\Downarrow$

  $\text{Int}(\text{parent}(\mu)) = (p_{i-1}, p_i] \subseteq [x, x']$

  Also, impossible.

Number of Leaves

$n = |I|$: #segments
$m \leq 2n$: #distinct endpoints

$[p_i, p_i]$	m
(p_i, p_{i+1})	$m - 1$
$(-\infty, p_1)$	1
(p_m, ∞)	1

Number of Leaves

$n = |I|$: #segments
$m \leq 2n$: #distinct endpoints

$[p_i, p_i]$	m
(p_i, p_{i+1})	$m - 1$
$(-\infty, p_1)$	1
(p_m, ∞)	1

- $2m + 1 \leq 4n + 1$ leaves

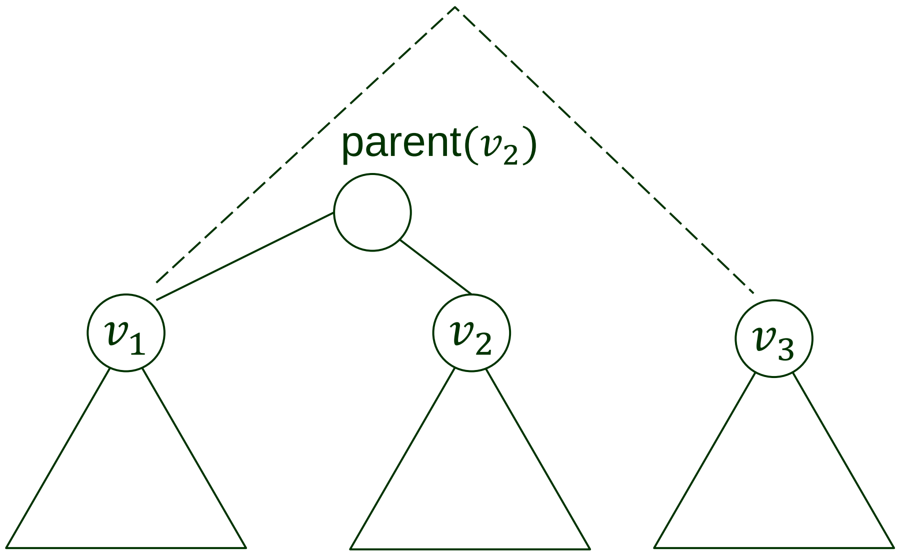
Storage of an Interval

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

Storage of an Interval

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

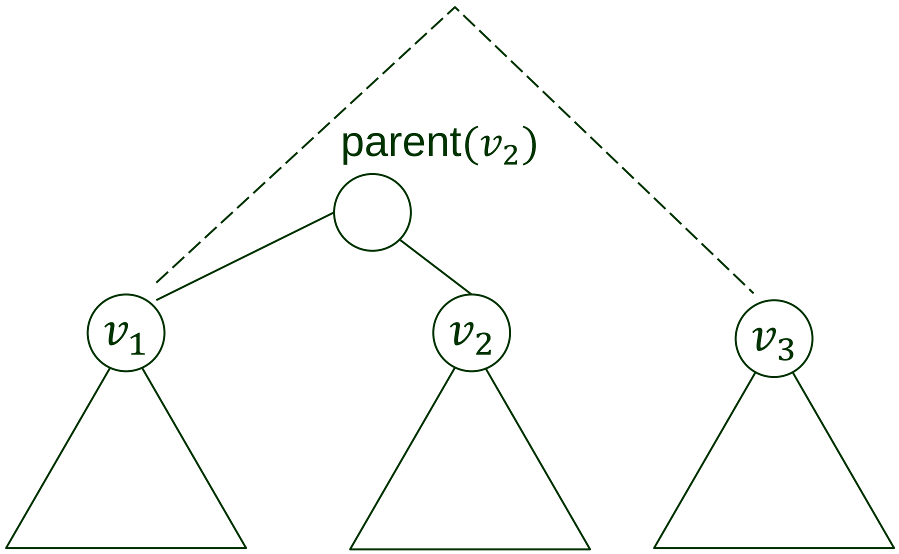
Proof Suppose the interval is stored at the nodes $v_1, v_2, \ldots, v_k$, $k \geq 3$, at the same depth in the left to right order.



Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at ≤ 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.

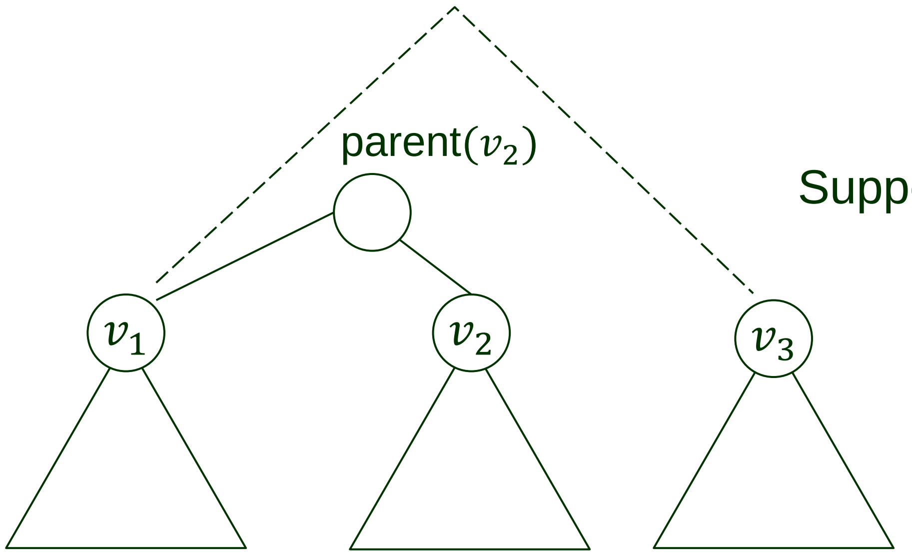


Then v_2 must be a sibling of either v_1 or v_3.

Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at ≤ 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.



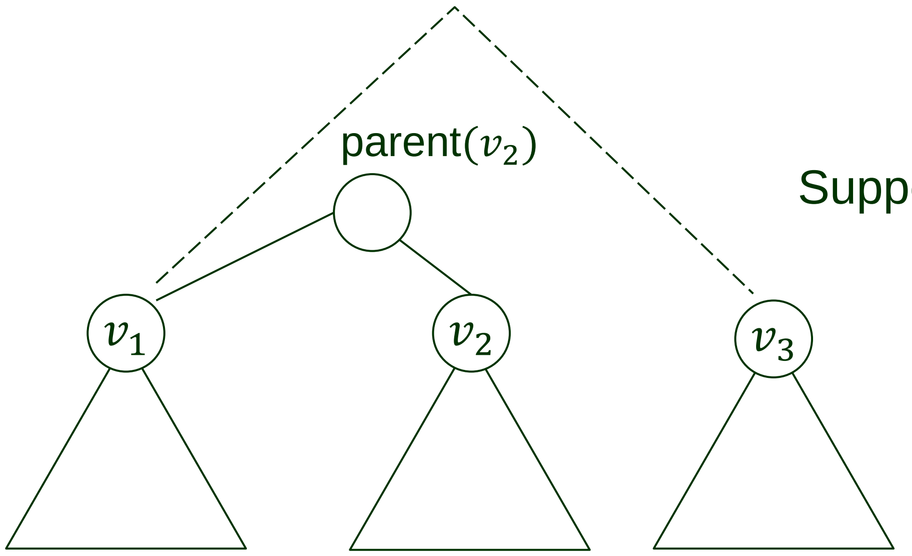
Then v_2 must be a sibling of either v_1 or v_3.

Suppose its sibling is v_1 without loss of generality.

Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at ≤ 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.

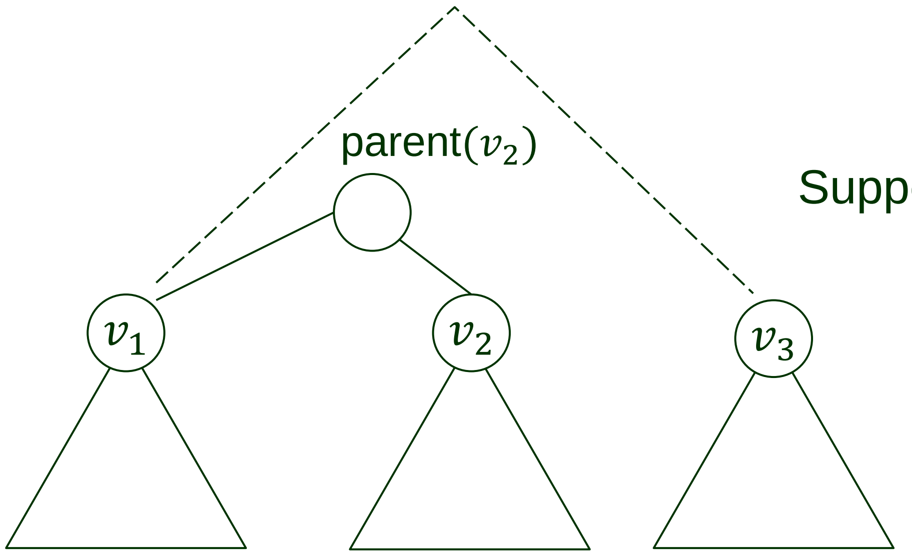
$\Downarrow$

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

Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at ≤ 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.

$$\Downarrow$$

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

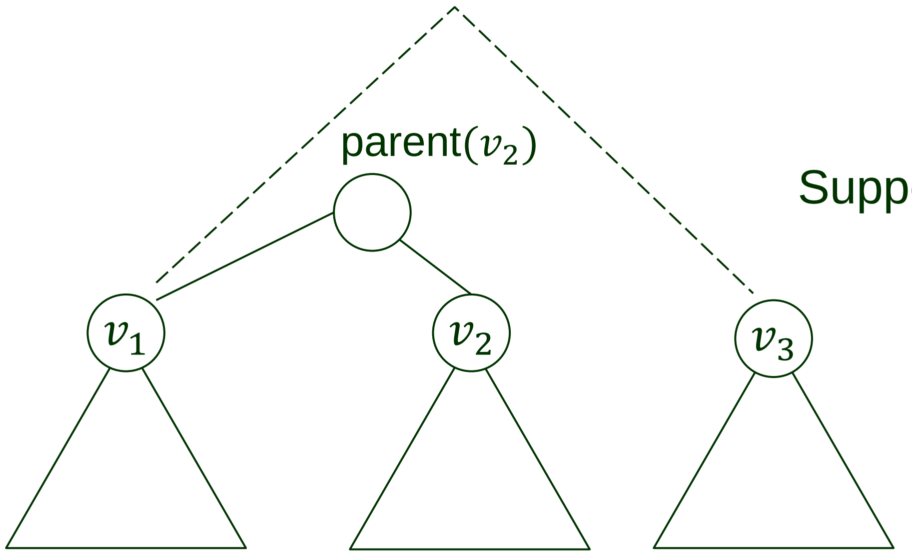
$$\Downarrow$$

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

Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at ≤ 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.

$$\Downarrow$$

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

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

$$\Downarrow$$

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

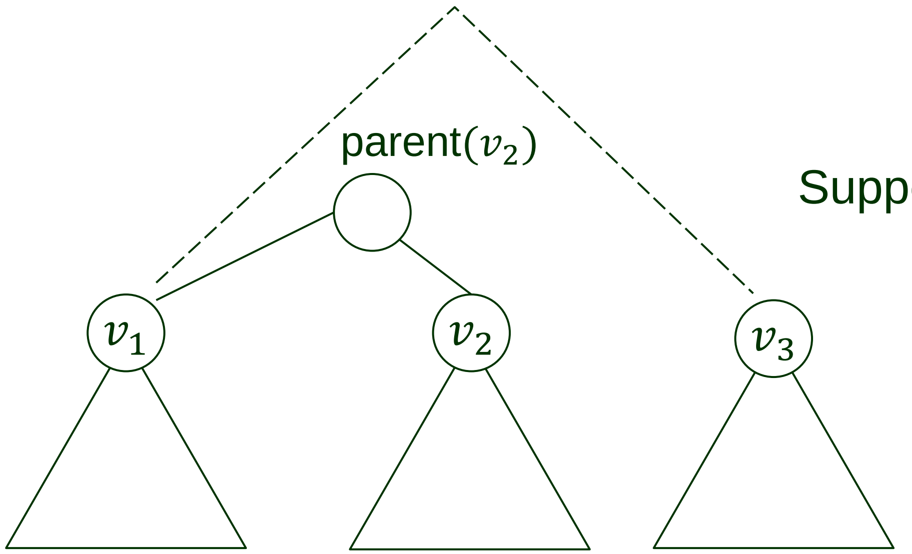
$$\Downarrow$$

$[x, x']$ should be stored at parent(v_2) or above instead of at v_2. Contradiction.

Storage of an Interval

Claim Each interval $[x, x'] \in I$ is stored at ≤ 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.

$$\Downarrow$$

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

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

$$\Downarrow$$

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

$$\Downarrow$$

$[x, x']$ should be stored at parent(v_2) or above instead of at v_2. Contradiction. $\square$

Total Storage

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

Total Storage

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

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

Total Storage

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

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 depth.

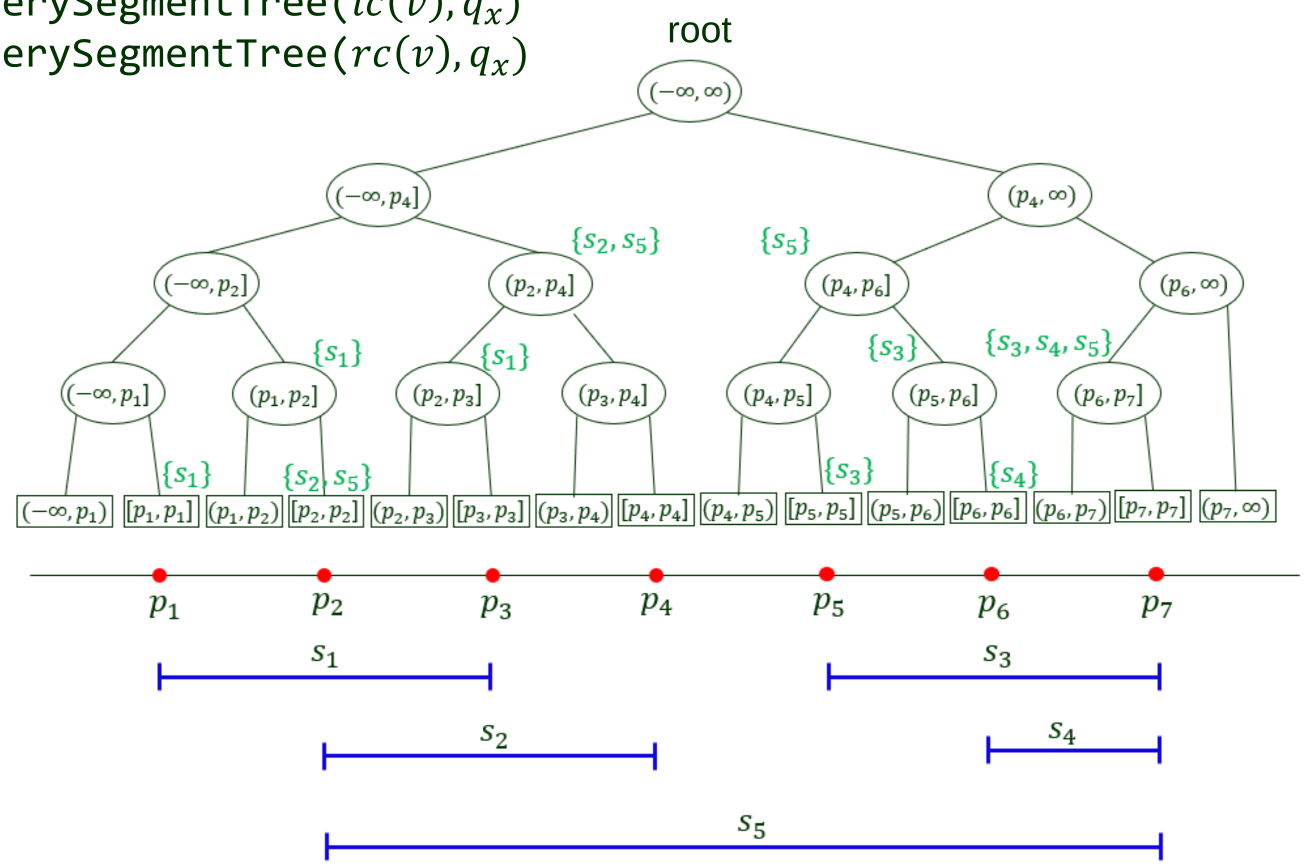
The required storage at each depth is $O(n)$.

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

$$O(n \log n)$$

V. Query Algorithm

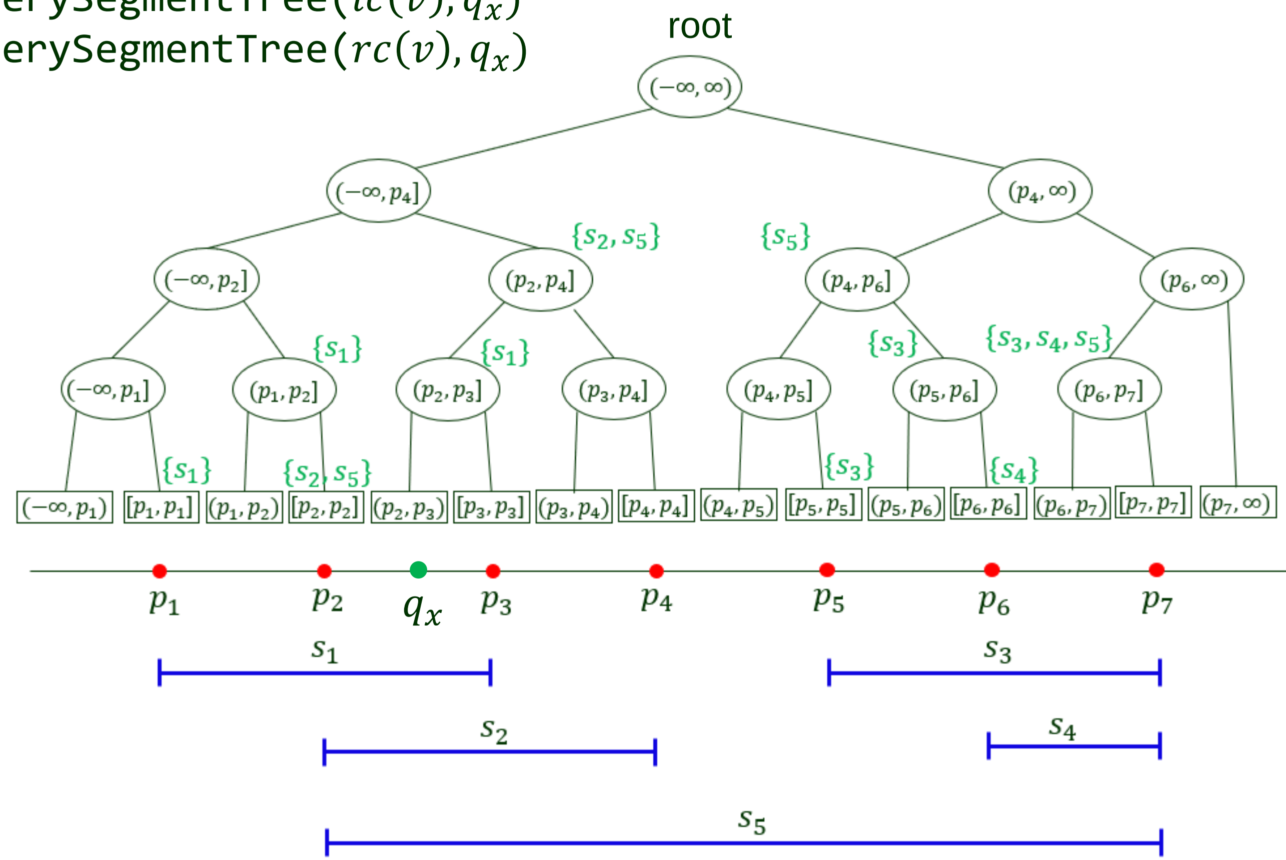
QuerySegmentTree(v, q_x)
// precondition: $q_x \in \mathrm{Int}(v)$
report all the intervals in $C(v)$ // canonical subset
if v is not a leaf
    then if $q_x \in$ Int($lc(v)$)
        then QuerySegmentTree($lc(v), q_x$)
        else QuerySegmentTree($rc(v), q_x$)



V. Query Algorithm

QuerySegmentTree(v, q_x)
// precondition: $q_x \in \mathrm{Int}(v)$
report all the intervals in $C(v)$ // canonical subset
if v is not a leaf
    then if $q_x \in \mathrm{Int}(lc(v))$
        then QuerySegmentTree($lc(v), q_x$)
        else QuerySegmentTree($rc(v), q_x$)

Example: q_x

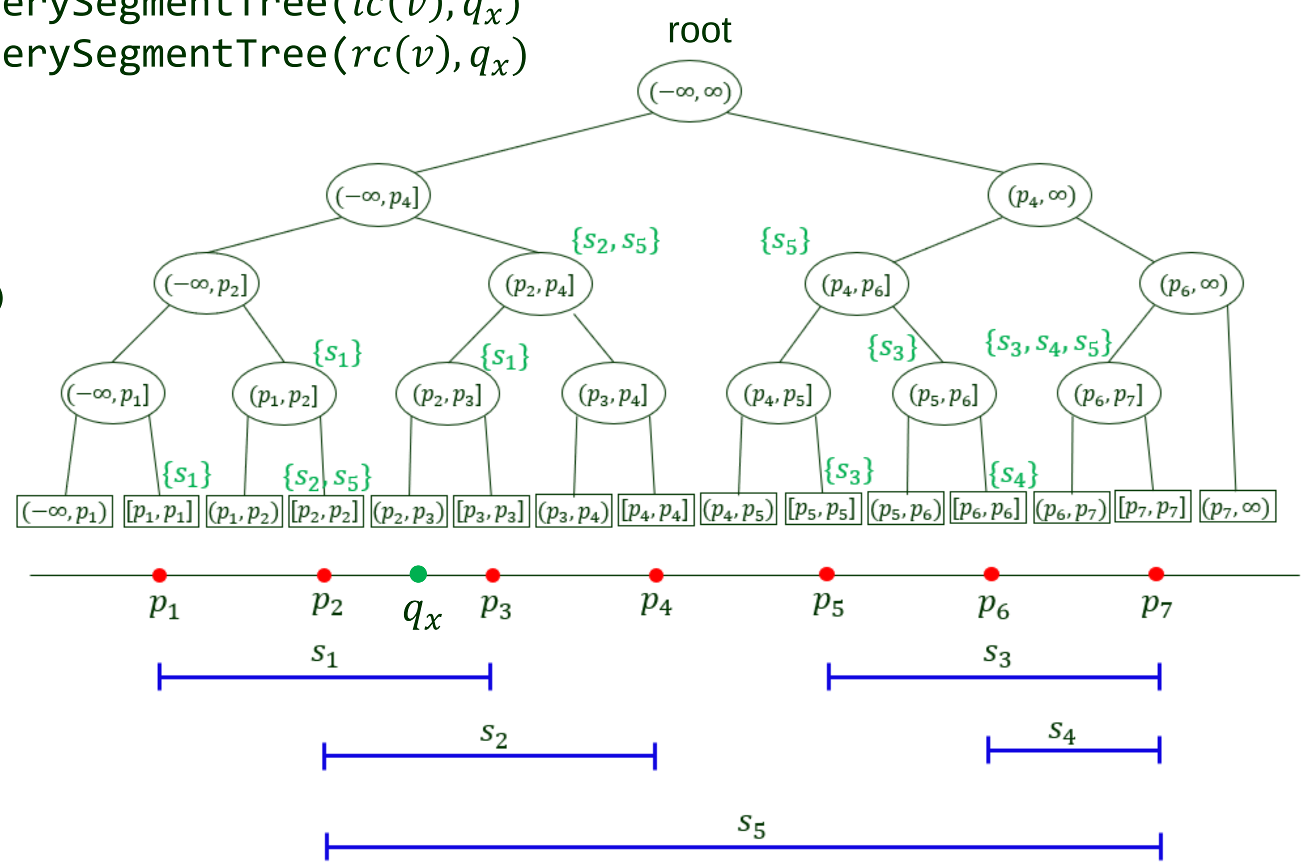


V. Query Algorithm

```
QuerySegmentTree(v, q_x)
// precondition: q_x ∈ Int(v)
report all the intervals in C(v) // canonical subset
if v is not a leaf
    then if q_x ∈ Int(lc(v))
            then QuerySegmentTree(lc(v), q_x)
            else QuerySegmentTree(rc(v), q_x)
```

Example: q_x

QuerySegmentTree(root, q_x)

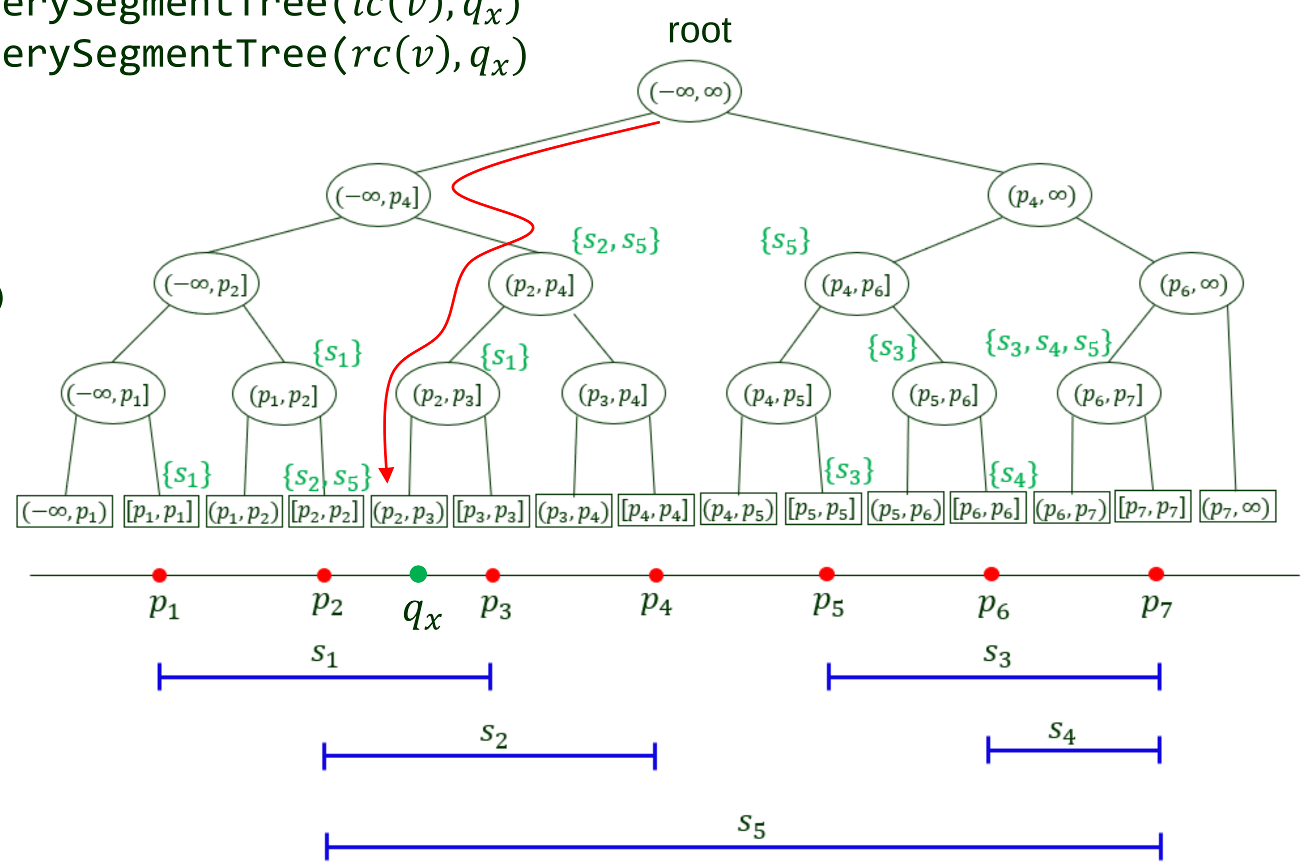


V. Query Algorithm

```
QuerySegmentTree(v, q_x)
// precondition: q_x ∈ Int(v)
report all the intervals in C(v) // canonical subset
if v is not a leaf
    then if q_x ∈ Int(lc(v))
            then QuerySegmentTree(lc(v), q_x)
            else QuerySegmentTree(rc(v), q_x)
```

Example: q_x

QuerySegmentTree(root, q_x)



V. Query Algorithm

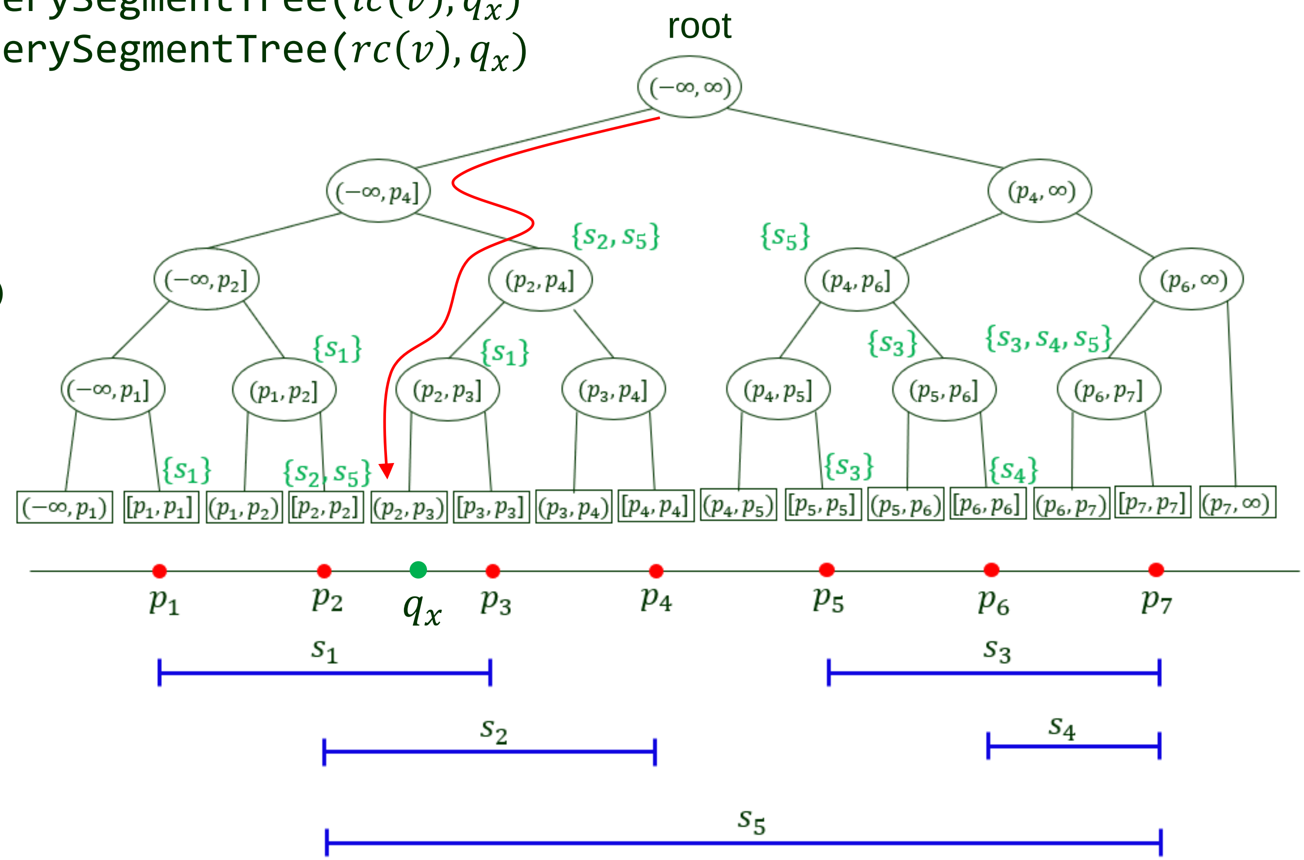
```
QuerySegmentTree(v, q_x)
// precondition: q_x ∈ Int(v)
    report all the intervals in C(v) // canonical subset
    if v is not a leaf
        then if q_x ∈ Int(lc(v))
                then QuerySegmentTree(lc(v), q_x)
                else QuerySegmentTree(rc(v), q_x)
```

Example: q_x

QuerySegmentTree(root, q_x)

Output in order:

s_2, s_5, s_1



V. Query Algorithm

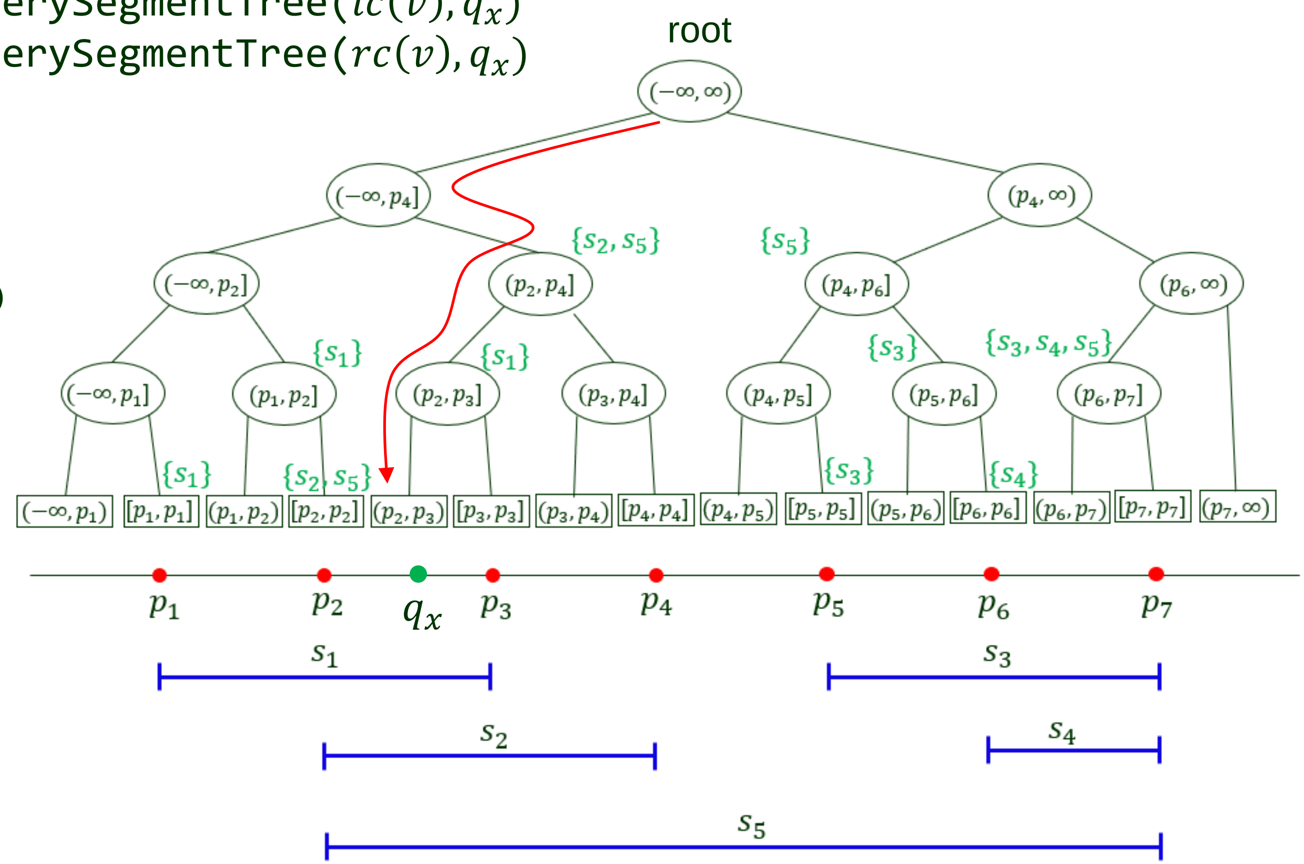
QuerySegmentTree(v, q_x)
// precondition: $q_x \in \mathrm{Int}(v)$
report all the intervals in $C(v)$ // canonical subset
if v is not a leaf
  then if $q_x \in \mathrm{Int}(lc(v))$
    then QuerySegmentTree$(lc(v), q_x)$
    else QuerySegmentTree$(rc(v), q_x)$

Example: q_x

QuerySegmentTree(root, q_x)

Output in order:

s_2, s_5, s_1



Query time:

$O(\log n + k)$

V. Query Algorithm

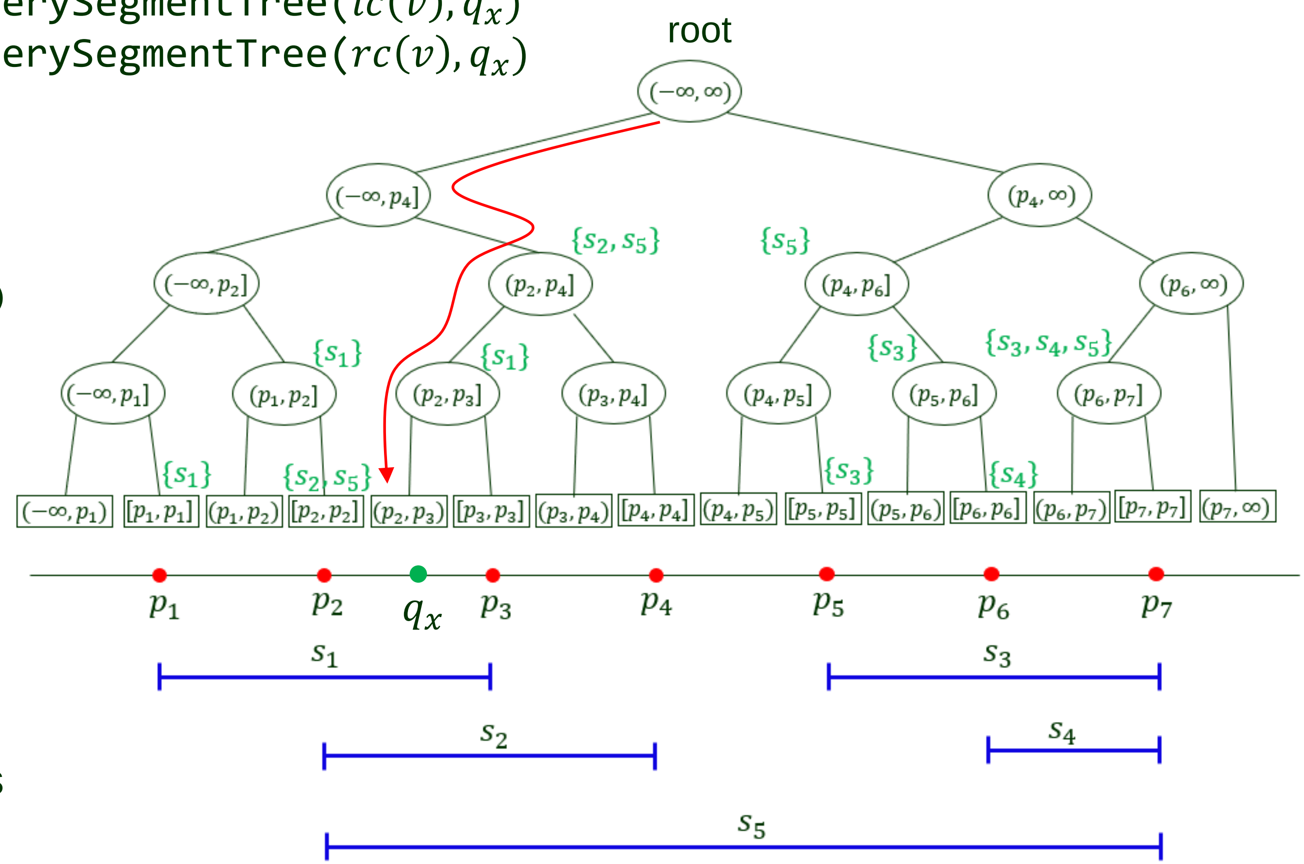
```
QuerySegmentTree(v, q_x)
// precondition: q_x ∈ Int(v)
report all the intervals in C(v) // canonical subset
if v is not a leaf
    then if q_x ∈ Int(lc(v))
             then QuerySegmentTree(lc(v), q_x)
             else QuerySegmentTree(rc(v), q_x)
```

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.

$$O(n \log n)$$

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

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

$O(n)$ for all the nodes together

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

Segment Tree Insertion

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

if Int(v)$\subseteq [x, x']$ // Int(parent(v))$\not\subseteq [x, x']$ holds
    then store $[x, x']$ at v
    else if Int($lc(v)$)$\cap [x, x'] \neq \emptyset$
        then InsertSegmentTree($lc(v), [x, x']$)
      if Int($rc(v)$)$\cap [x, x'] \neq \emptyset$
        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

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

if Int$(v) \subseteq [x, x']$ // Int(parent(v)) $\not\subseteq [x, x']$ holds
    then store $[x, x']$ at v
    else if Int$(lc(v)) \cap [x, x'] \neq \emptyset$
        then InsertSegmentTree$(lc(v), [x, x'])$
        if Int$(rc(v)) \cap [x, x'] \neq \emptyset$
        then InsertSegmentTree$(rc(v), [x, x'])$

- At each visited node v, either $[x, x']$ is stored or Int(v) contains an endpoint of $[x, x']$.
  - An interval is stored ≤ 2 times at each depth.

Segment Tree Insertion

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

if Int$(v) \subseteq [x, x']$ // Int(parent(v))$\not\subseteq [x, x']$ holds
  then store $[x, x']$ at v
  else if Int$(lc(v)) \cap [x, x'] \neq \emptyset$
    then InsertSegmentTree$(lc(v), [x, x'])$
   if Int$(rc(v)) \cap [x, x'] \neq \emptyset$
    then InsertSegmentTree$(rc(v), [x, x'])$

- At each visited node v, either $[x, x']$ is stored or Int(v) contains an endpoint of $[x, x']$.
  - An interval is stored ≤ 2 times at each depth.
  - At each depth, there exist

Segment Tree Insertion

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

if Int(v) ⊆ [x, x'] // Int(parent(v)) ⊈ [x, x'] holds
    then store [x, x'] at v
    else if Int(lc(v)) ∩ [x, x'] ≠ ∅
             then InsertSegmentTree(lc(v), [x, x'])
         if Int(rc(v)) ∩ [x, x'] ≠ ∅
             then InsertSegmentTree(rc(v), [x, x'])
```

- At each visited node v, either $[x, x']$ is stored or Int(v) contains an endpoint of $[x, x']$.
  - An interval is stored ≤ 2 times at each depth.
  - At each depth, there exist

    ≤ 1 node u such that $x \in$ Int(u)

Segment Tree Insertion

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

if Int(v)$\subseteq [x, x']$ // Int(parent(v))$\not\subseteq [x, x']$ holds
    then store $[x, x']$ at v
    else if Int($lc(v)$)$\cap [x, x'] \neq \emptyset$
        then InsertSegmentTree($lc(v), [x, x']$)
        if Int($rc(v)$)$\cap [x, x'] \neq \emptyset$
        then InsertSegmentTree($rc(v), [x, x']$)

- At each visited node v, either $[x, x']$ is stored or Int(v) contains an endpoint of $[x, x']$.
    - An interval is stored ≤ 2 times at each depth.
    - At each depth, there exist

      ≤ 1 node u such that $x \in$ Int(u)

      ≤ 1 node u' such that $x' \in$ Int(u')

Construction Time

- ≤ 2 storage actions + ≤ 2 containments

Construction Time

- ≤ 2 storage actions + ≤ 2 containments

$$\Downarrow$$

≤ 4 nodes visited per level.

Construction Time

- ≤ 2 storage actions + ≤ 2 containments

$$\Downarrow$$

≤ 4 nodes visited per level.

$$\Downarrow$$

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

Construction Time

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

$$\Downarrow$$

≤ 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

- ≤ 2 storage actions + ≤ 2 containments

$$\Downarrow$$

≤ 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

$$\Downarrow$$

≤ 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 $O(\log n + k)$

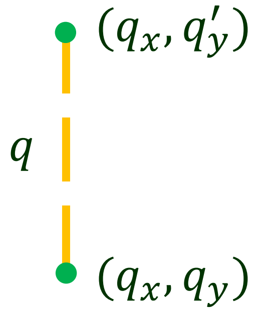
♣ a larger storage $O(n \log n)$ than $O(n)$,

but it allows to answer more complicated queries.

VI. Back to Windowing

Query segment: $q = q_x \times [q_y, q_y']$

n *arbitrarily oriented* segments

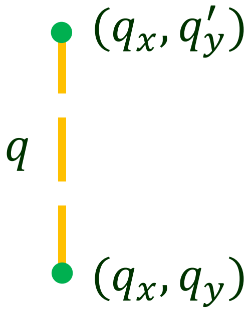


VI. Back to Windowing

Query segment: $q = q_x \times [q_y, q_y']$

n *arbitrarily oriented* segments

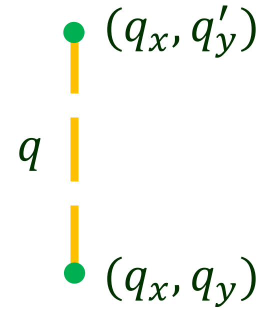
Construct a segment tree $\mathcal{T}$.



VI. Back to Windowing

Query segment: $q = q_x \times [q_y, q_y']$

n *arbitrarily oriented* segments



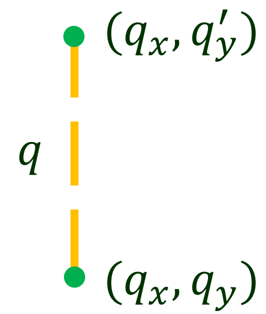
Construct a segment tree $\mathcal{T}$.

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

VI. Back to Windowing

Query segment: $q = q_x \times [q_y, q_y']$

n *arbitrarily oriented* segments



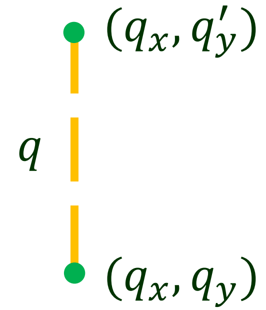
Construct a segment tree $\mathcal{T}$.

- 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 $S(v)$: Int$(v) \times (-\infty, \infty)$ but does not cross its parent's slab.

VI. Back to Windowing

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

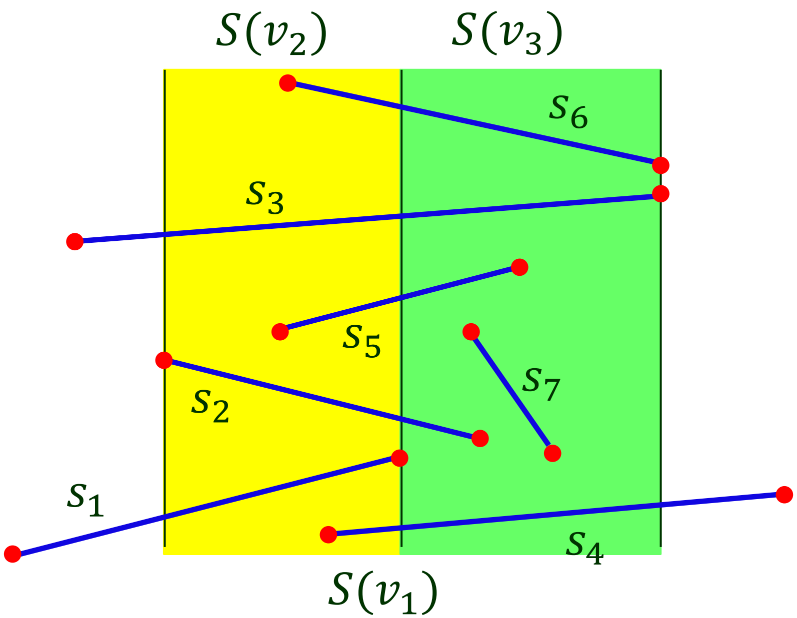


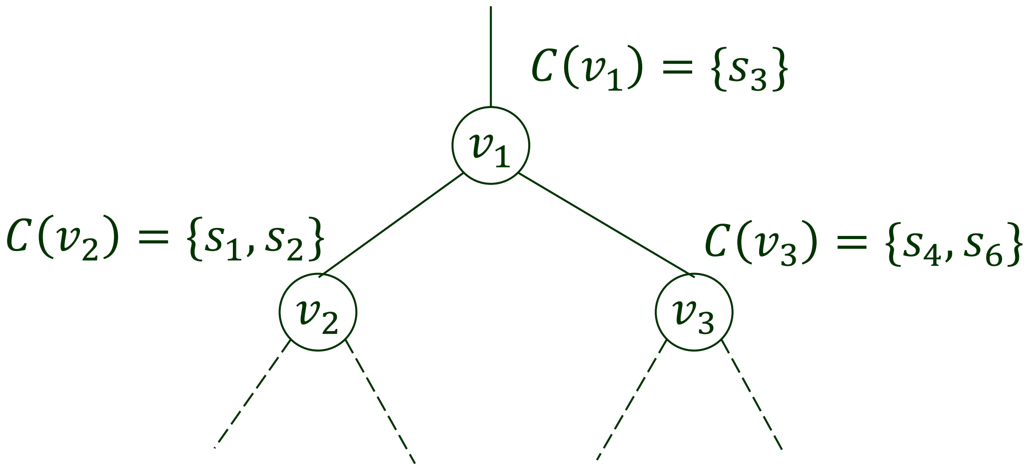
n *arbitrarily oriented* segments

Construct a segment tree $\mathcal{T}$.

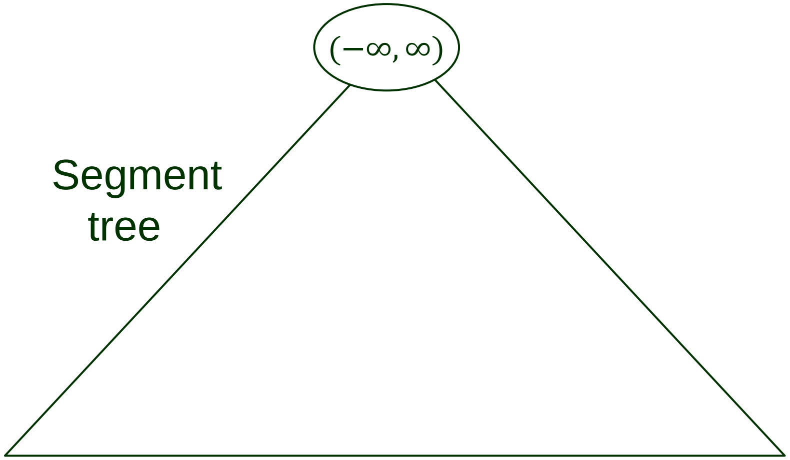
- On the x-intervals of the segments in S.
- Canonical subset $C(v)$ at a vertex v store segments rather than their x-intervals.

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.



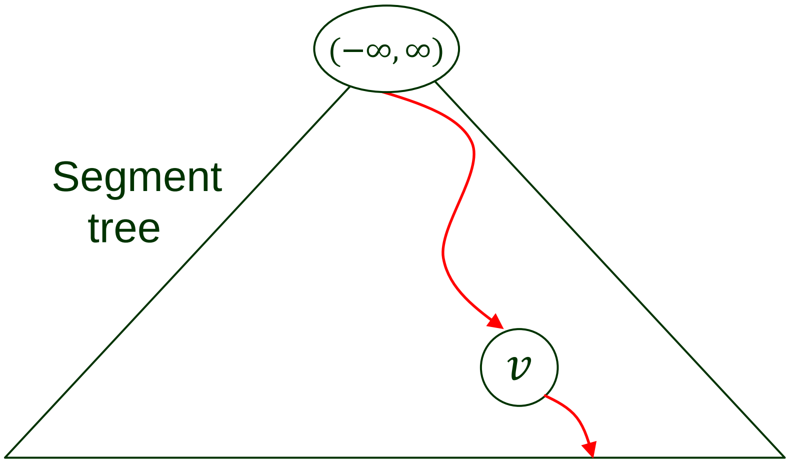


Segments on the Search Path



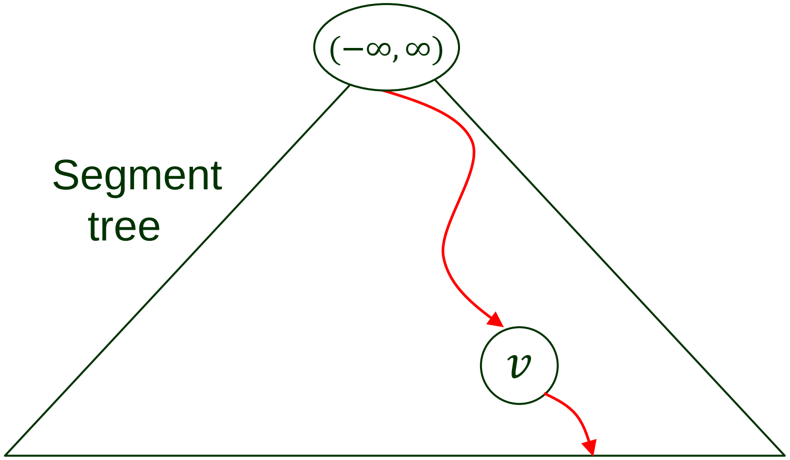
- Search with q_x in $\mathcal{T}$.

Segments on the Search Path



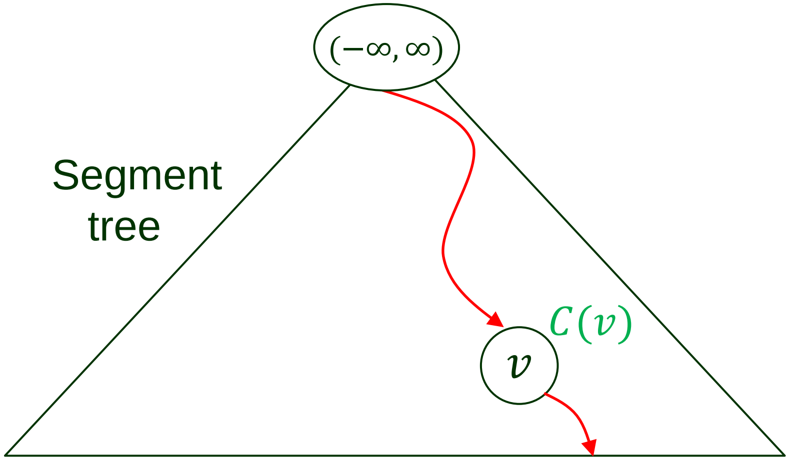
- Search with q_x in $\mathcal{T}$.

Segments on the Search Path



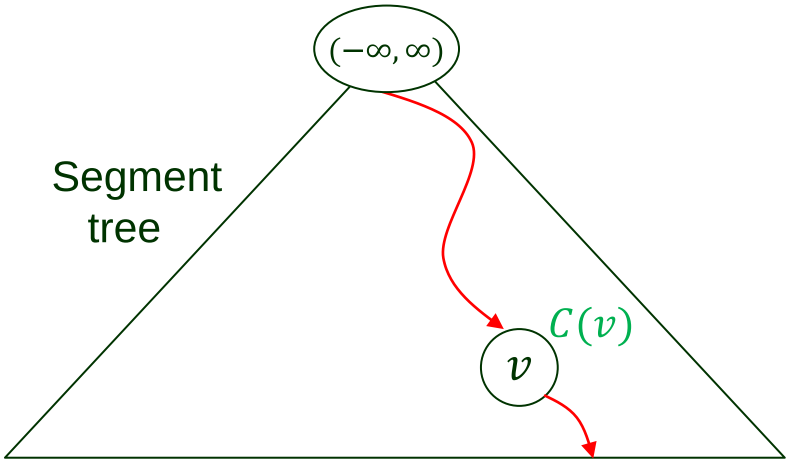
- Search with q_x in $\mathcal{T}$.
- $O(\log n)$ canonical sets together include all the segments intersected by the vertical line $x = q_x$.

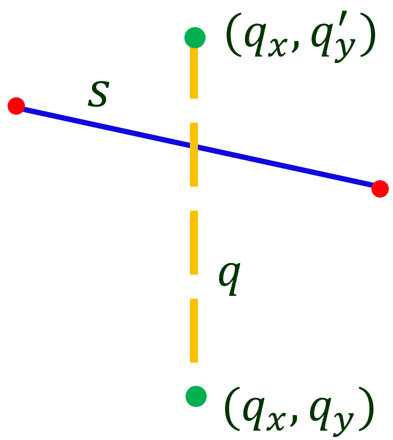
Segments on the Search Path



- 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.

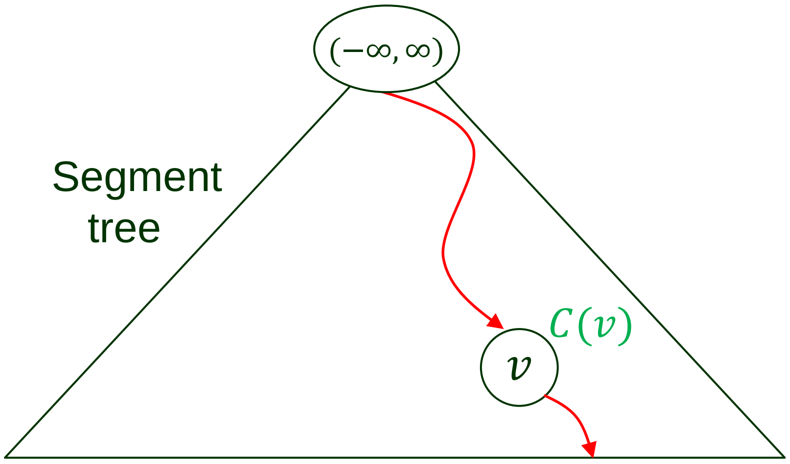
Segments on the Search Path



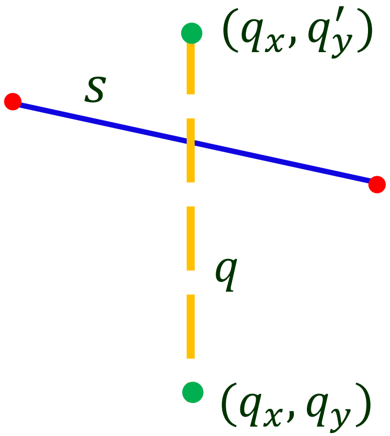


- Search with q_x in $\mathcal{T}$.
- $O(\log n)$ canonical sets together include all the segments intersected by the vertical line $x = q_x$.
- v is a node on the search path.
- Segment $s \in C(v)$ is intersected by q iff

Segments on the Search Path



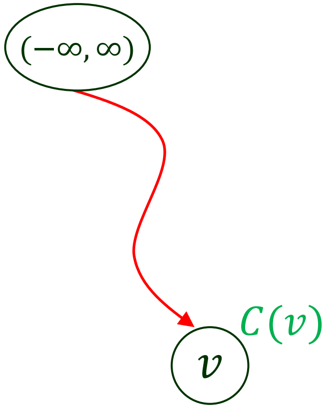
- Search with q_x in $\mathcal{T}$.
- $O(\log n)$ canonical sets together include all the segments intersected by the vertical line $x = q_x$.
- v is a node on the search path.



- Segment $s \in C(v)$ is intersected by q iff

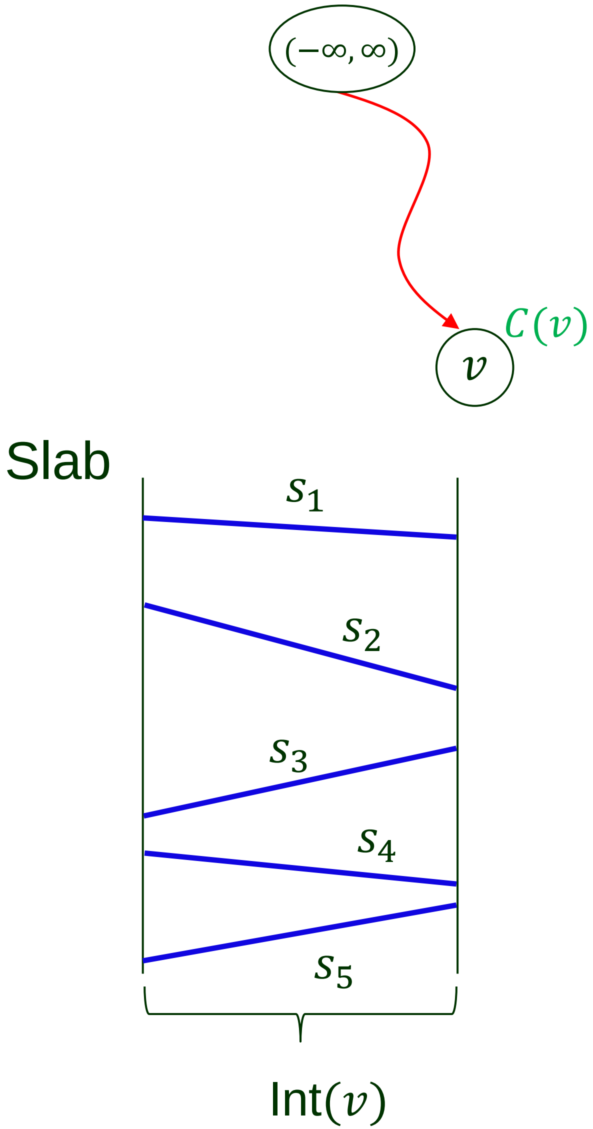
(q_x, q_y) below s and (q_x, q'_y) above s

VII. Storage of a Canonical Set



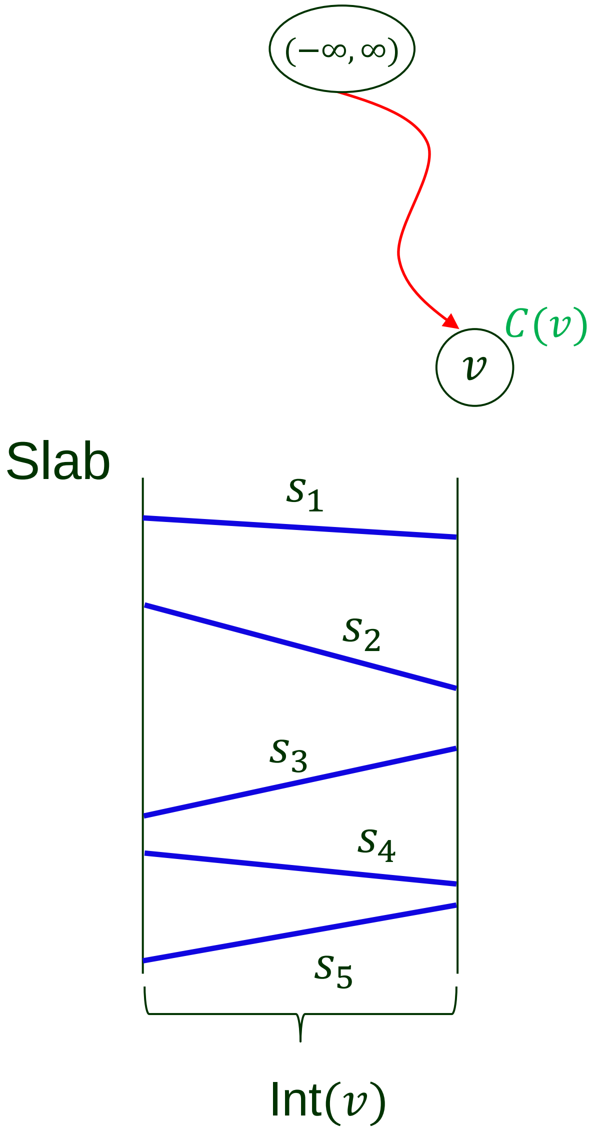
- Segments in $C(v)$ do not intersect each other.
- Each segment is over an x-interval containing Int(v).

VII. Storage of a Canonical Set



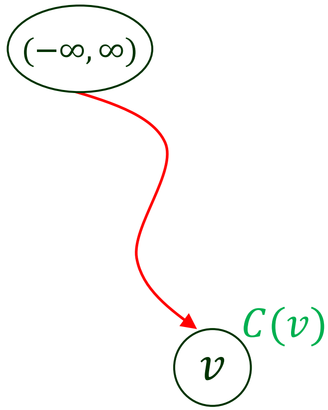
- Segments in $C(v)$ do not intersect each other.
- Each segment is over an x-interval containing Int(v).
- These segments span the slab Int$(v) \times [-\infty, \infty]$.
- They do not intersect each other in the interior.

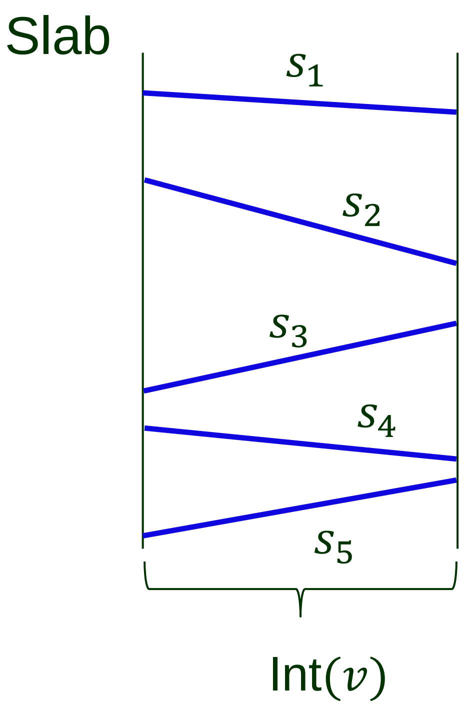
VII. Storage of a Canonical Set



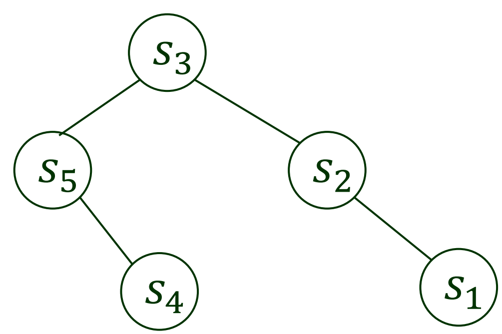
- Segments in $C(v)$ do not intersect each other.
- Each segment is over an x-interval containing $\text{Int}(v)$.
- These segments span the slab $\text{Int}(v) \times [-\infty, \infty]$.
- They do not intersect each other in the interior.
- Store the segments in a balanced BST $\mathcal{B}(v)$ in the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

VII. Storage of a Canonical Set



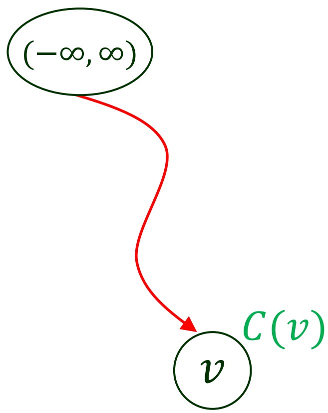


- Segments in $C(v)$ do not intersect each other.
- Each segment is over an x-interval containing Int(v).
- These segments span the slab Int$(v) \times [-\infty, \infty]$.
- They do not intersect each other in the interior.
- Store the segments in a balanced BST $\mathcal{B}(v)$ in the vertical order.

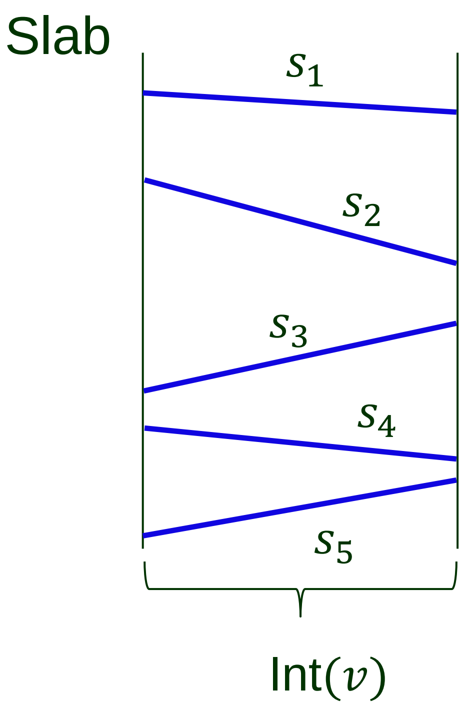


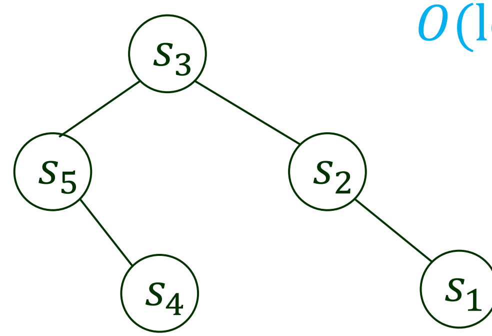
associate structure $\mathcal{B}(v)$

VII. Storage of a Canonical Set



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





associate structure $\mathcal{B}(v)$

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

#intersected segments in $C(v)$

Segment Tree Storage Summary

- The set S of segments is stored in a segment tree $\mathcal{T}$ based on

  their x-intervals.

- The canonical subset $C(v)$ of every internal node v in $\mathcal{T}$ is stored in a BST $\mathcal{B}(v)$ based on

  the vertical order within the slab $\mathrm{Int}(v) \times [-\infty, \infty]$.

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

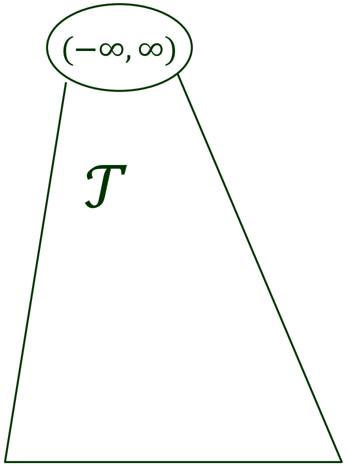
- Search with q_x in a segment tree $\mathcal{T}$.

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 $\mathcal{T}$.

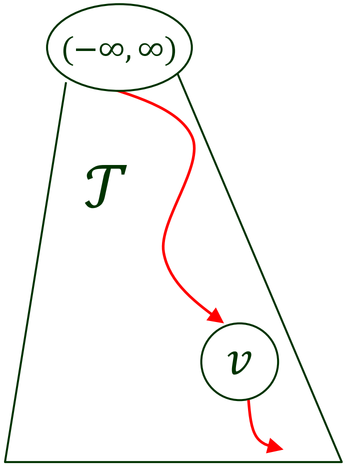


Query Algorithm

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 $\mathcal{T}$.

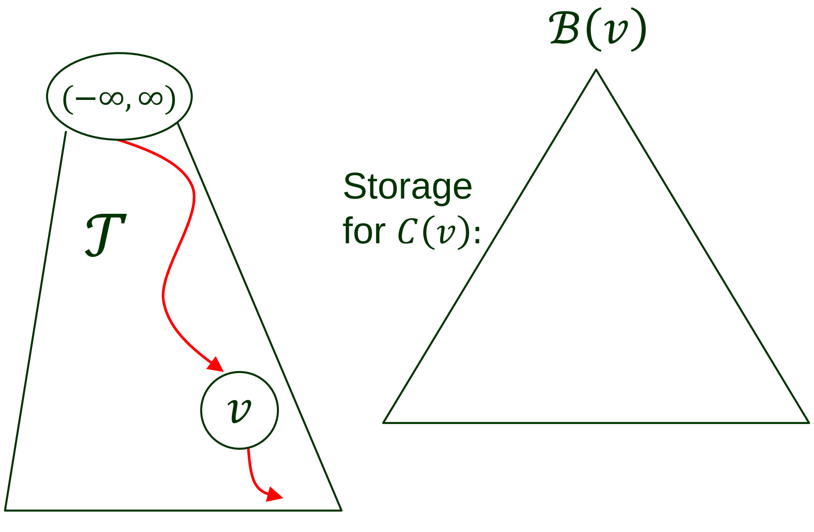


Query Algorithm

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 $\mathcal{T}$.

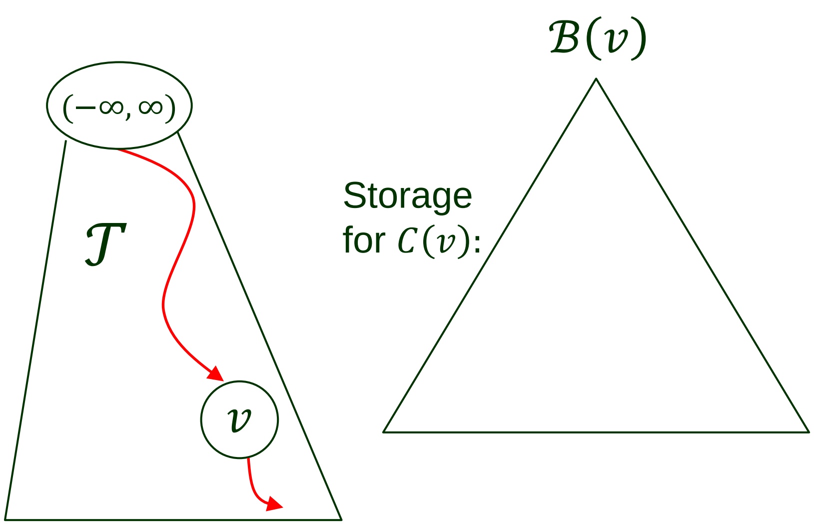


Query Algorithm

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 $\mathcal{T}$.
- 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.

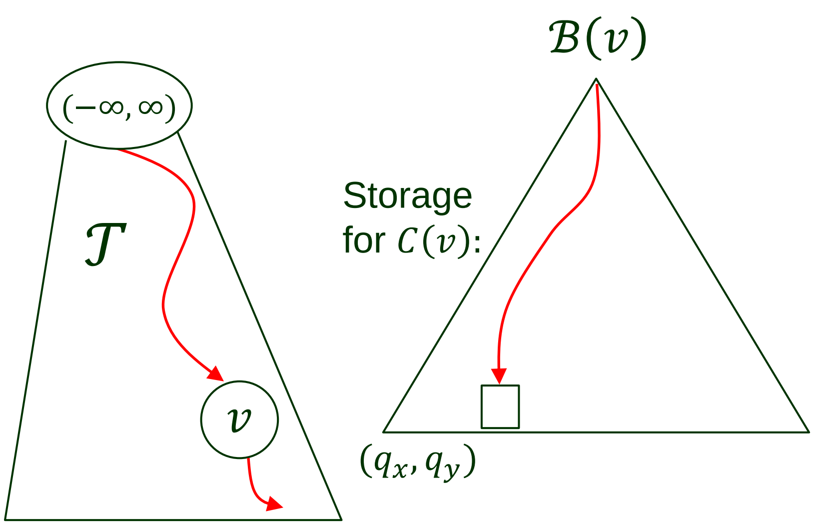


Query Algorithm

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 $\mathcal{T}$.
- 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.

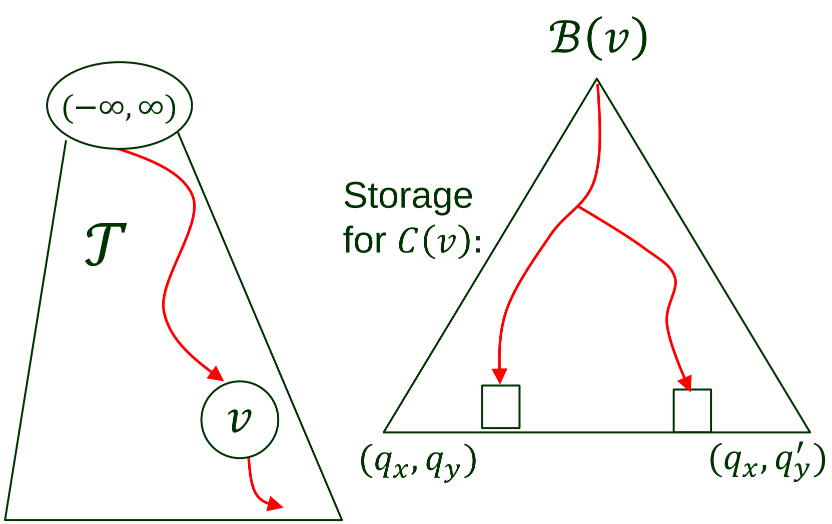


Query Algorithm

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 $\mathcal{T}$.
- 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.



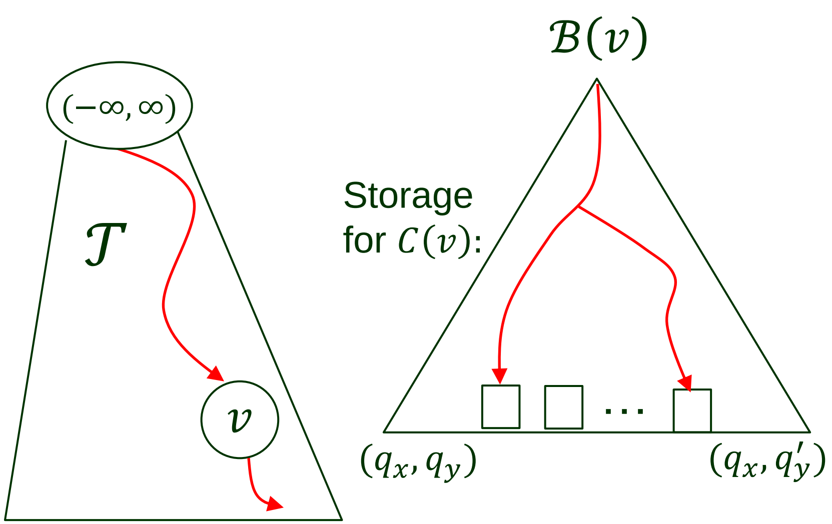
Query Algorithm

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 $\mathcal{T}$.

- 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.



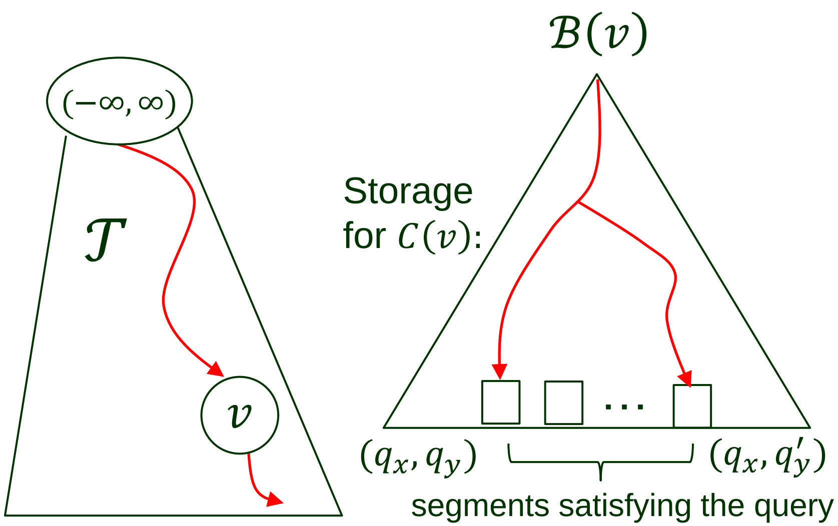
Query Algorithm

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 $\mathcal{T}$.

- 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.



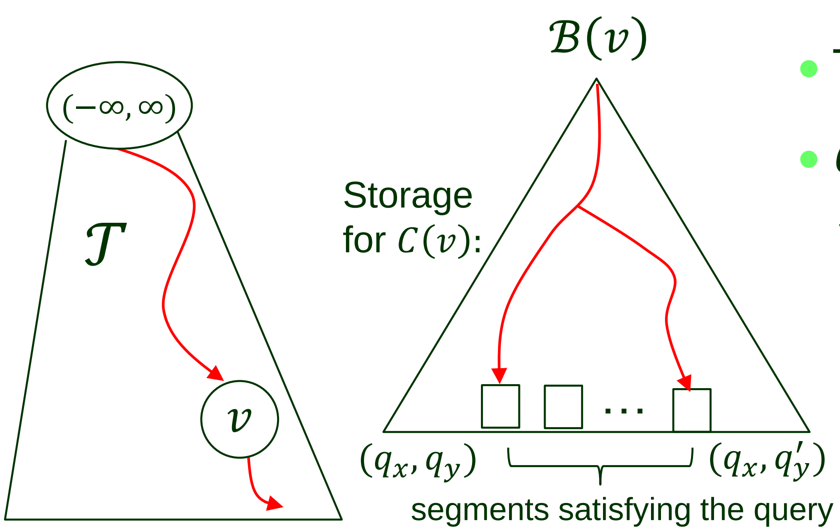
Query Algorithm

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 $\mathcal{T}$.

- 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.



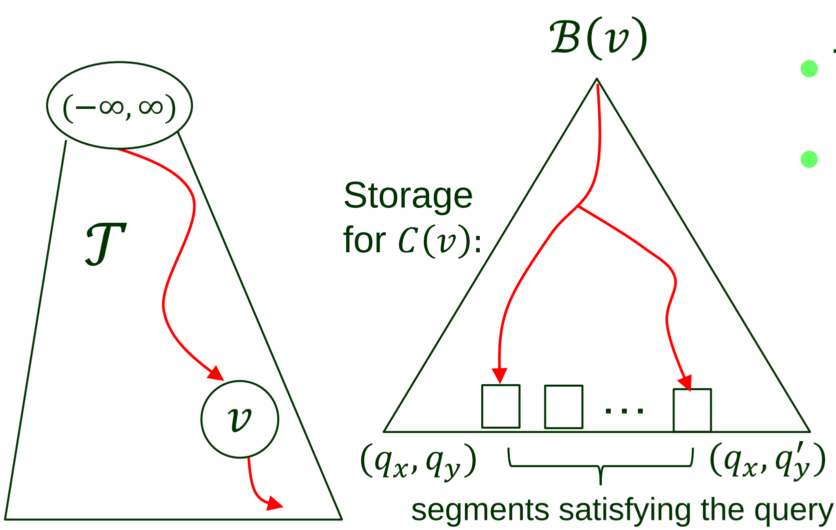
- The search takes $O(\log n + k_v)$ time.
- $O(\log n)$ nodes on the search path, each requiring such a search.

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 $\mathcal{T}$.
- 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.

$\Downarrow$

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