Range and windowing queries

- range searching
  - range trees
  - fractional cascading
- windowing queries
  - interval trees
  - priority search trees
  - segment trees

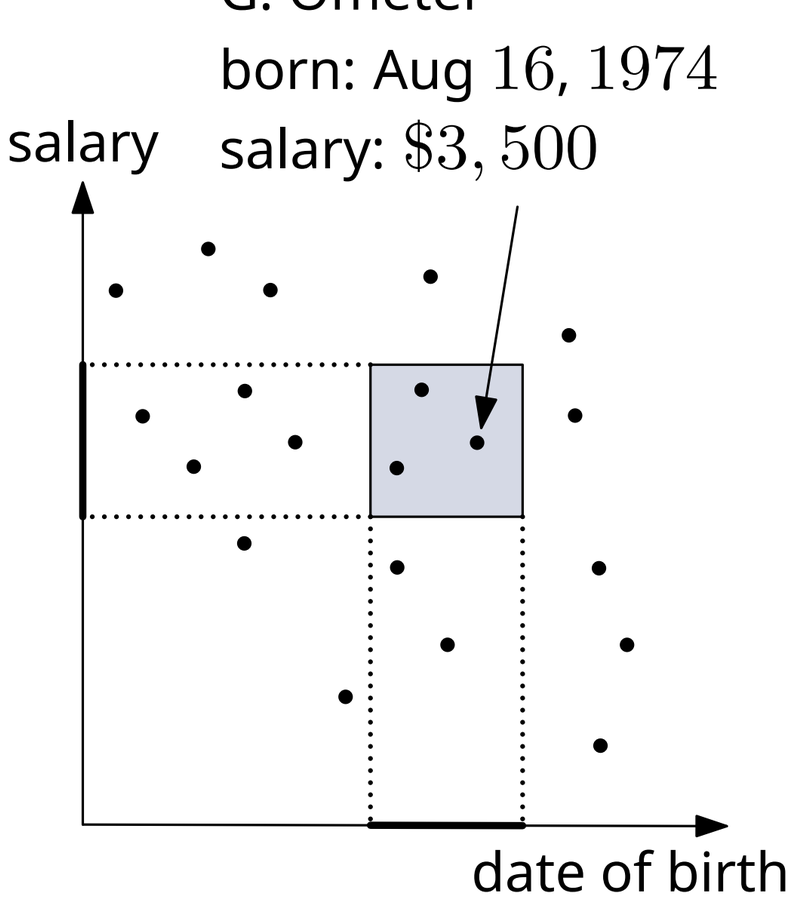
Range queries

range searching

- range trees
- fractional cascading

Database queries

A database query may ask for all employees with age between a_1 and a_2, and salary between s_1 and s_2



Data structures

Idea of data structures

- representation of structure, for convenience (like DCEL)
- preprocessing of data, to be able to solve future questions really fast (sub-linear time)

Data structures

Idea of data structures

- representation of structure, for convenience (like DCEL)
- preprocessing of data, to be able to solve future questions really fast (sub-linear time)

1D range query problem

Problem (1D range query): Preprocess a set of n points on the real line such that the ones inside a 1D query range (interval) can be found fast.

- the points $p_1, \ldots, p_n$ are known beforehand
- the query range $[x : x']$ will be known only later

1D range query problem

Problem (1D range query): Preprocess a set of n points on the real line such that the ones inside a 1D query range (interval) can be found fast.

- the points $p_1, \ldots, p_n$ are known beforehand
- the query range $[x : x']$ will be known only later

A solution to a query problem consists of three parts:

- a data structure
- a query algorithm
- a construction algorithm

1D range query problem

Problem (1D range query): Preprocess a set of n points on the real line such that the ones inside a 1D query range (interval) can be found fast.

- the points $p_1, \ldots, p_n$ are known beforehand
- the query range $[x : x']$ will be known only later

A solution to a query problem consists of three parts:

- a data structure
- a query algorithm
- a construction algorithm
- (an update algorithm)

1D range query problem

Problem (1D range query): Preprocess a set of n points on the real line such that the ones inside a 1D query range (interval) can be found fast.

- the points $p_1, \ldots, p_n$ are known beforehand
- the query range $[x : x']$ will be known only later

A solution to a query problem consists of three parts:

- a data structure
- a query algorithm
- a construction algorithm
- (an update algorithm)

Question: What are the most important factors for the efficiency of a solution?

1D range query problem

Problem (1D range query): Preprocess a set of n points on the real line such that the ones inside a 1D query range (interval) can be found fast.

- the points $p_1, \ldots, p_n$ are known beforehand
- the query range $[x : x']$ will be known only later

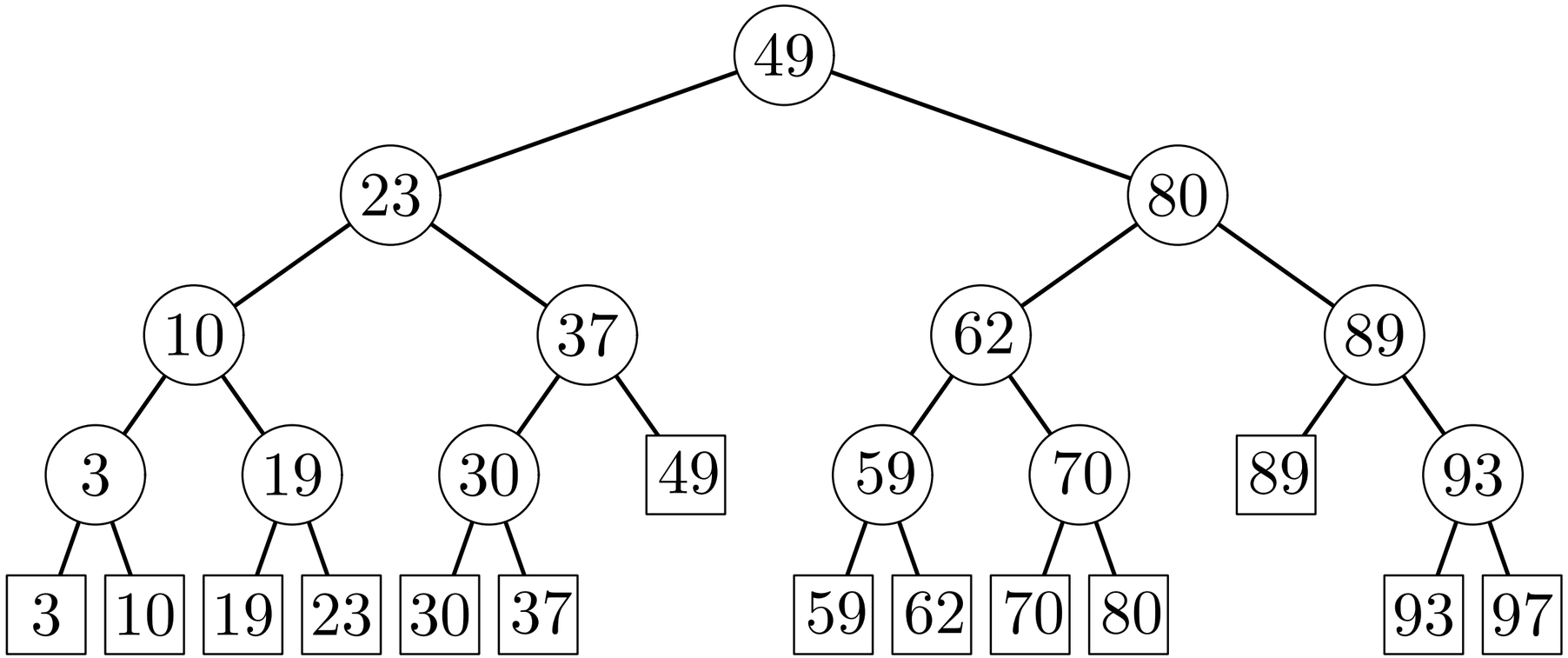
A solution to a query problem consists of three parts:

- a data structure $\rightarrow$ storage requirement
- a query algorithm $\rightarrow$ query running time
- a construction algorithm $\rightarrow$ construction running time
- (an update algorithm) $\rightarrow$ update running time

Question: What are the most important factors for the efficiency of a solution?

1D range query problem: data structure & query algorithm

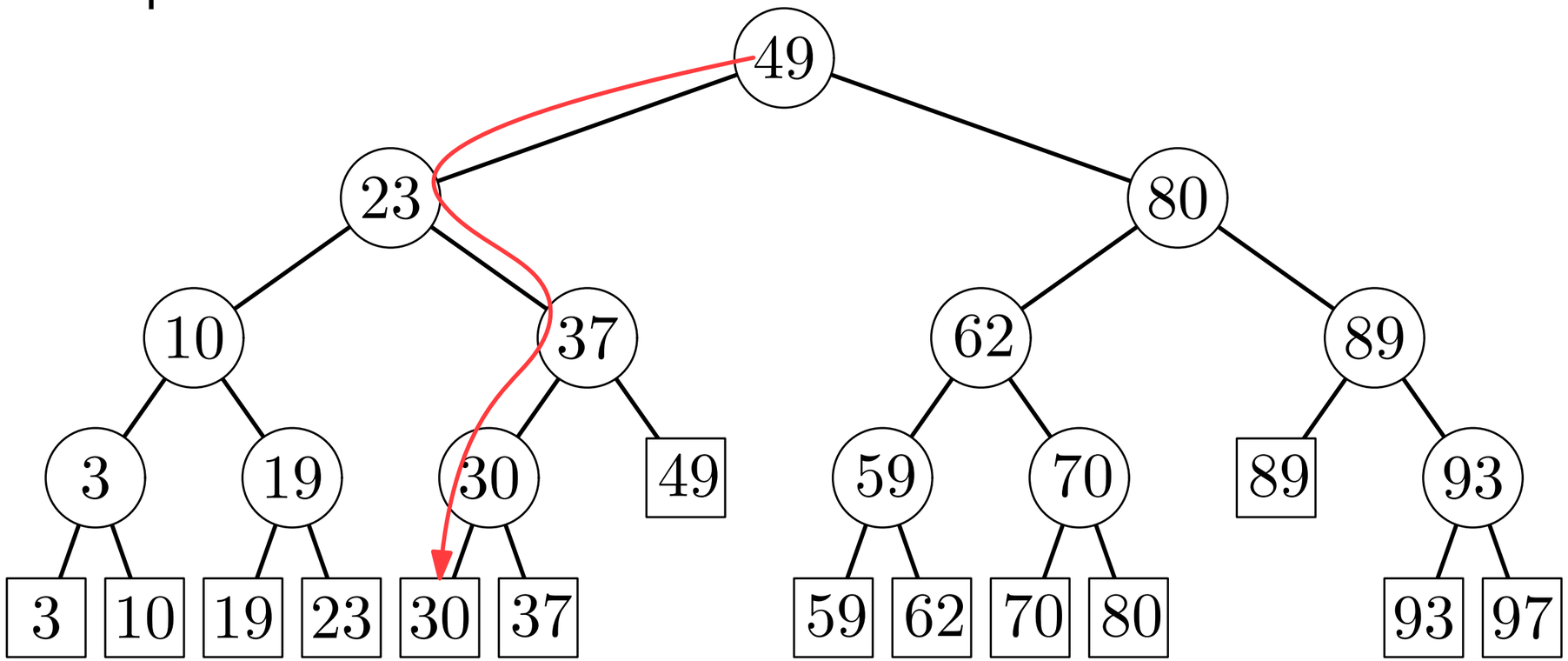
Data structure: balanced binary search tree (with points in leaves)



1D range query problem: data structure & query algorithm

Data structure: balanced binary search tree (with points in leaves)

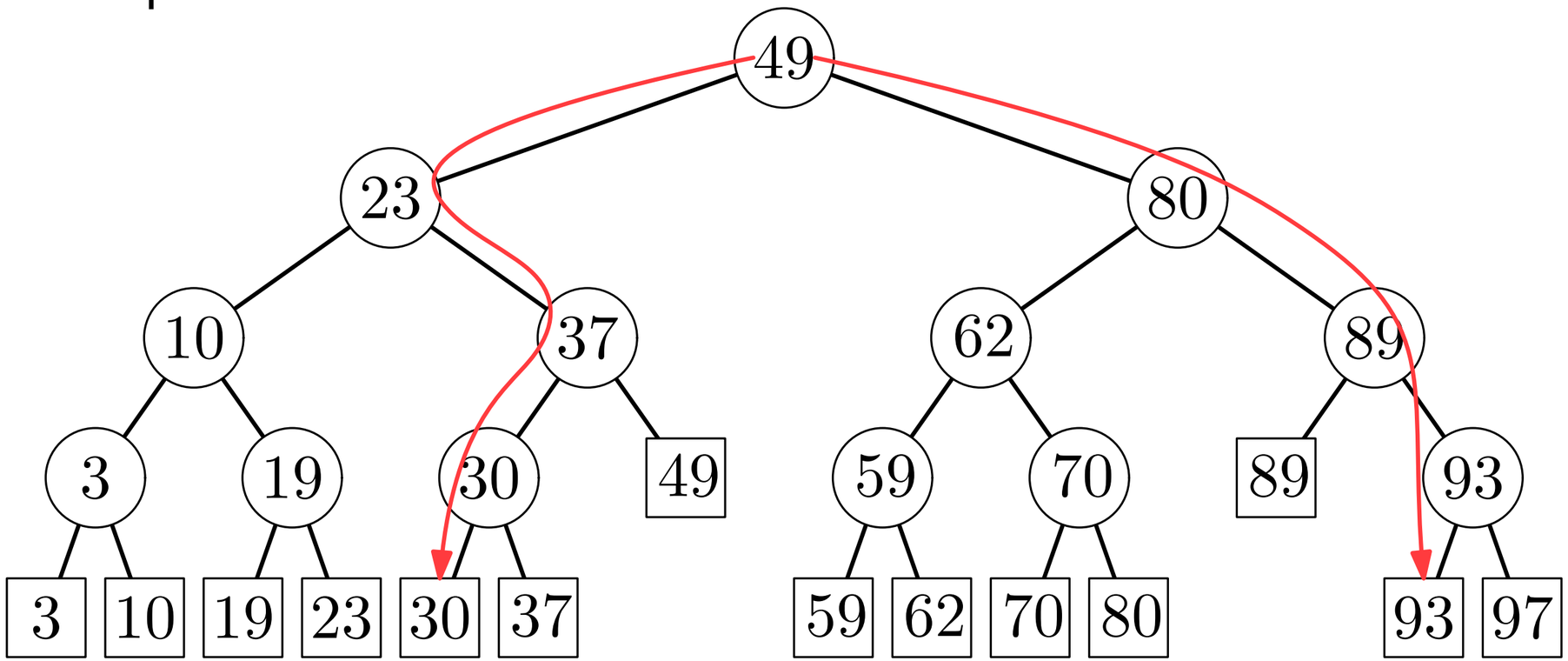
- search path for 25



1D range query problem: data structure & query algorithm

Data structure: balanced binary search tree (with points in leaves)

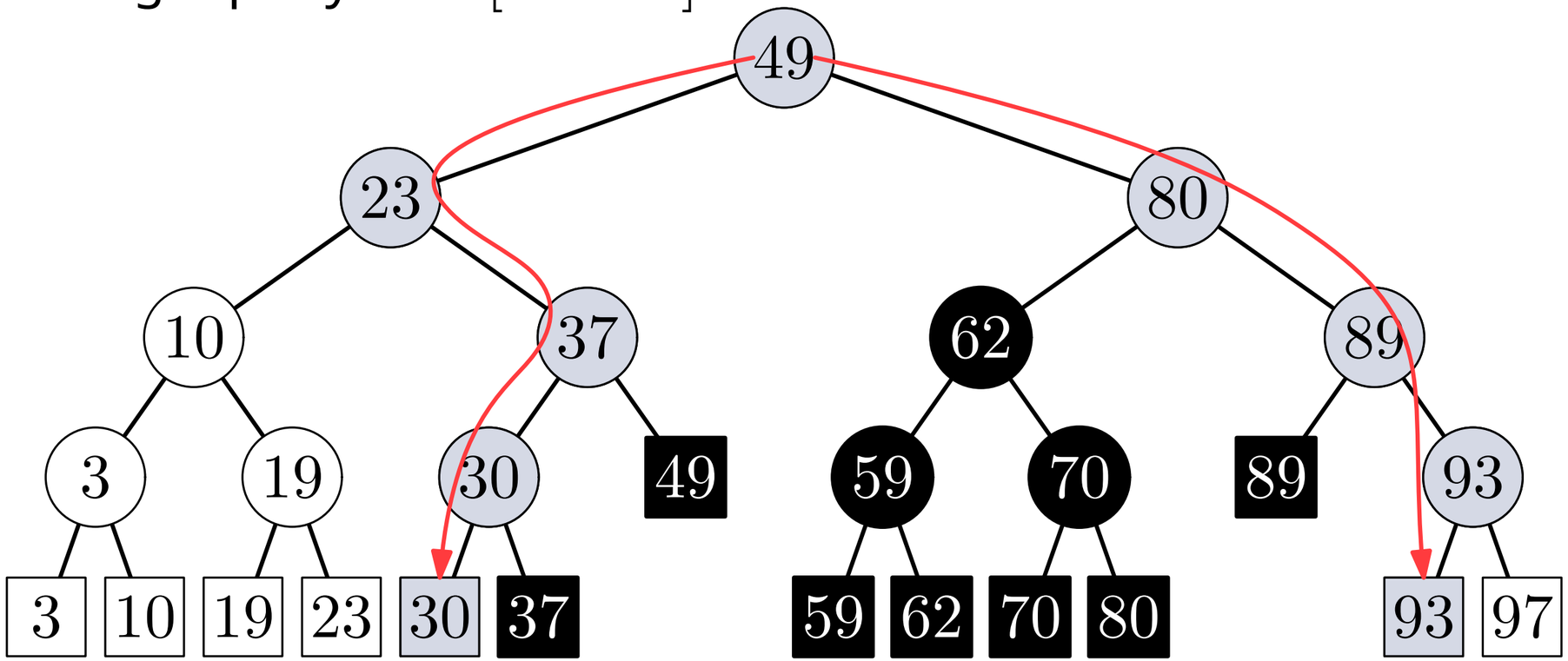
- search paths for 25 and 90



1D range query problem: data structure & query algorithm

Data structure: balanced binary search tree (with points in leaves)

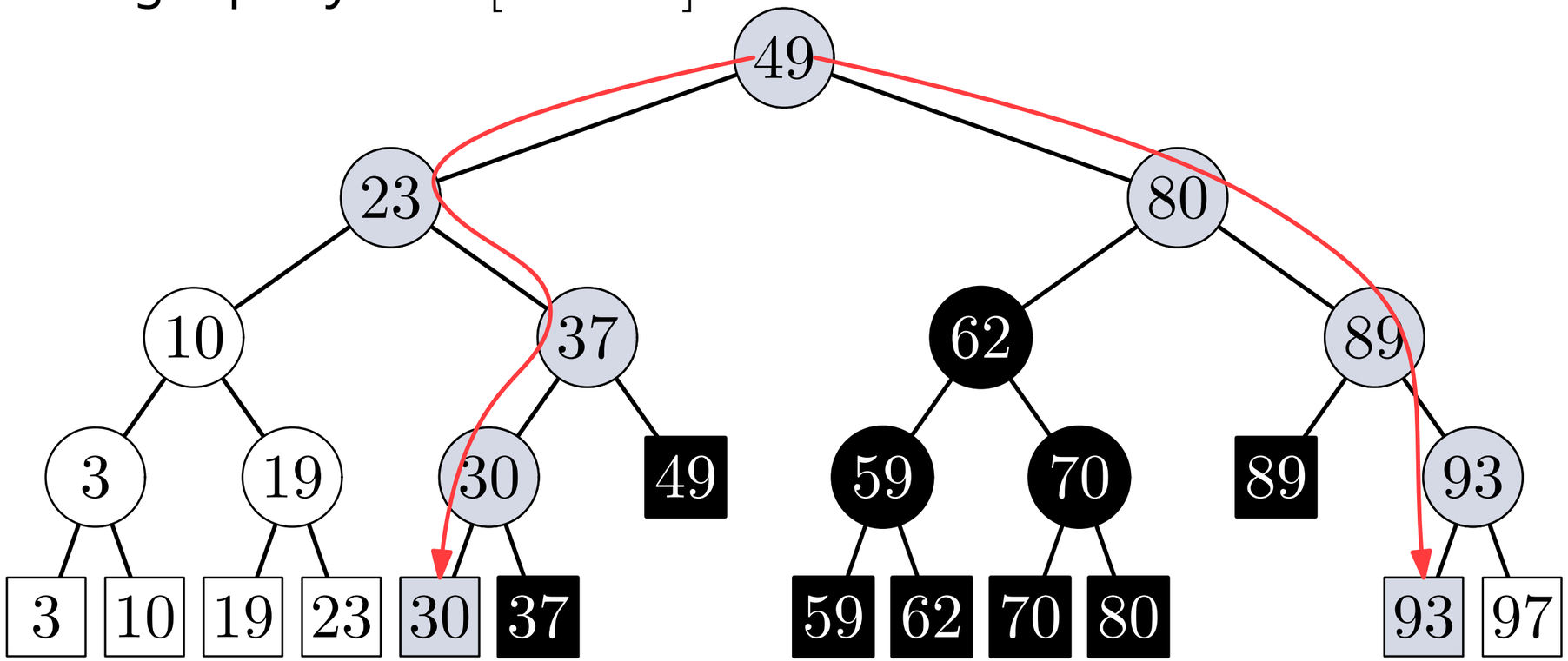
- 1D range query with $[25 : 90]$



1D range query problem: data structure & query algorithm

Data structure: balanced binary search tree (with points in leaves)

- 1D range query with $[25 : 90]$

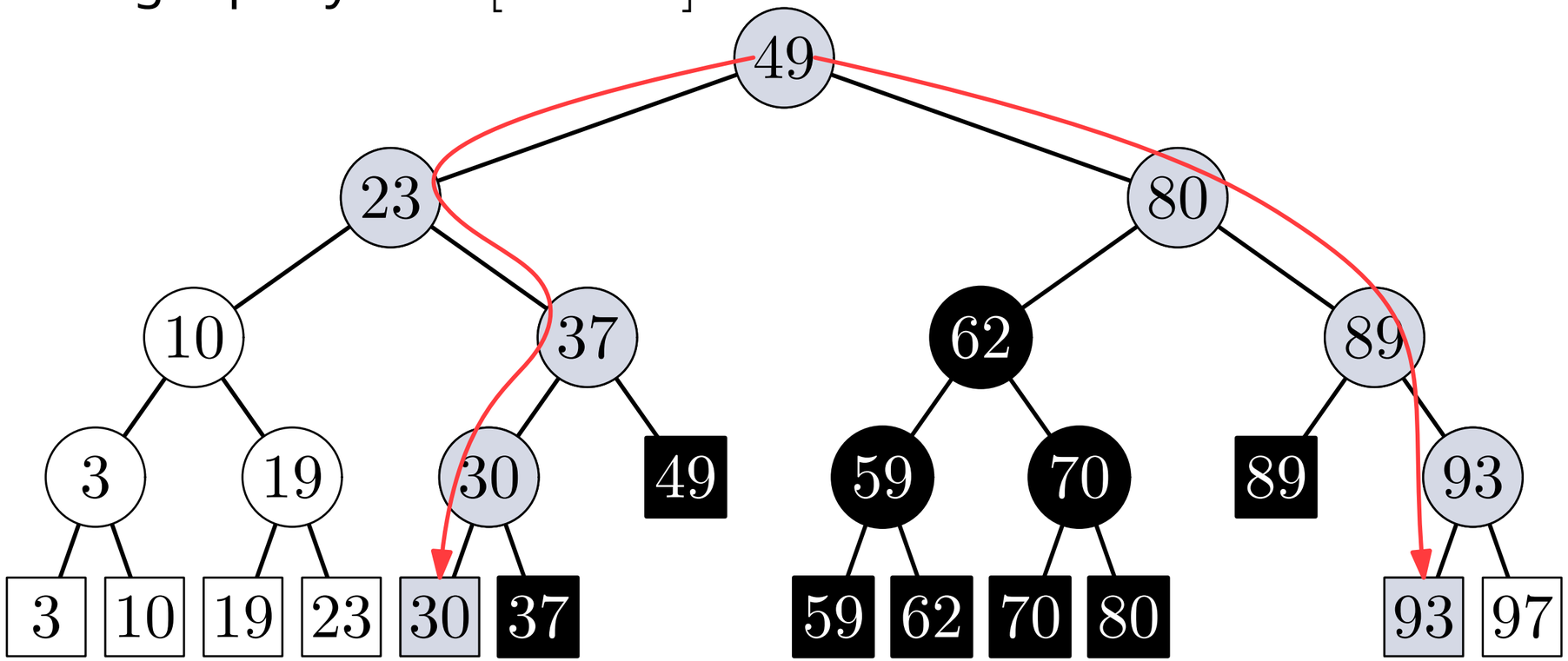


Question: What is the space requirement?

1D range query problem: data structure & query algorithm

Data structure: balanced binary search tree (with points in leaves)

- 1D range query with $[25 : 90]$



Question: What is the space requirement? $O(n)$

1D range query problem: query algorithm

Three types of nodes for a given query:

White nodes: never visited by the query
Grey nodes: visited by the query, unclear if they lead to output
Black nodes: visited by the query, whole subtree is output

1D range query problem: query algorithm

Three types of nodes for a given query:

White nodes: never visited by the query
Grey nodes: visited by the query, unclear if they lead to output
Black nodes: visited by the query, whole subtree is output

The query algorithm comes down to what we do at each type of node:

White nodes: N/A

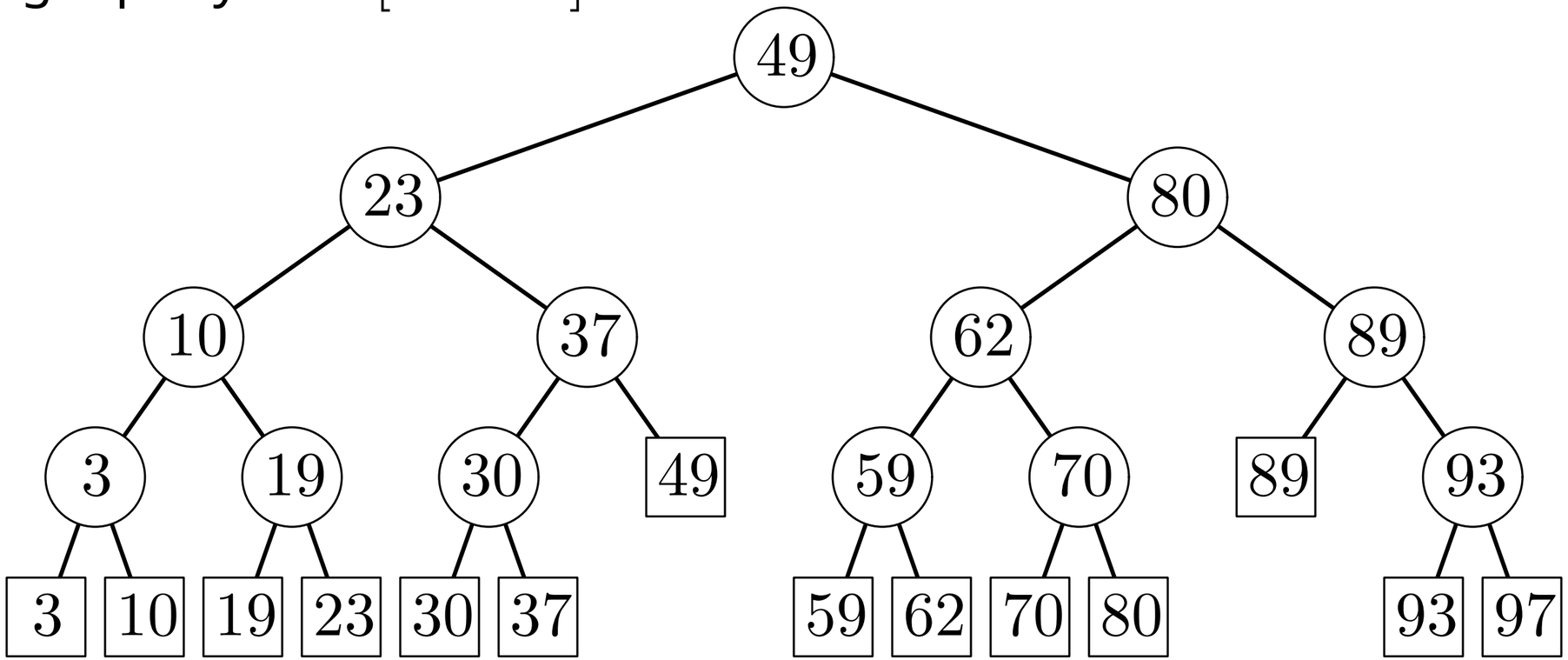
Grey nodes: use query range to decide how to proceed

- not visit a subtree (pruning),
- report a complete subtree, or
- continue to a left and/or right child

Black nodes: traverse and enumerate all points in the leaves

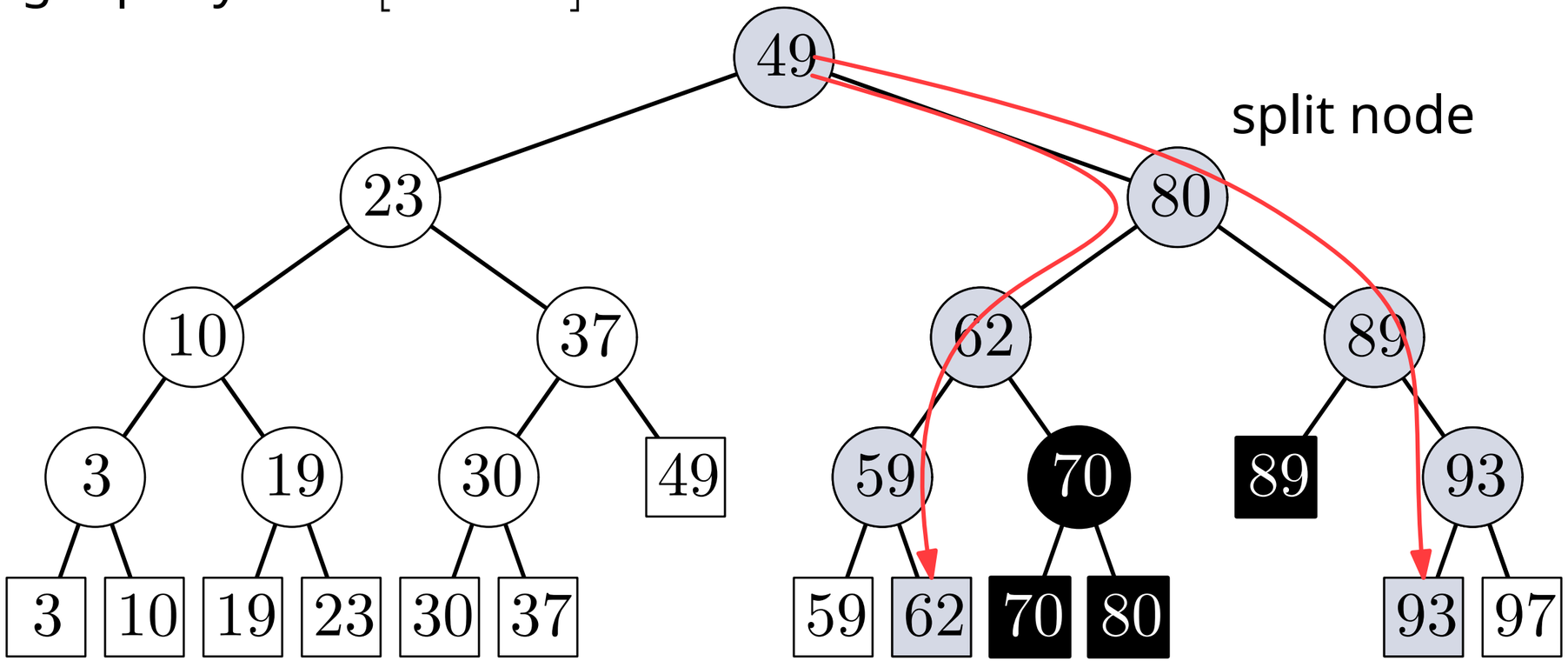
1D range query problem: split node

- 1D range query with $[61 : 90]$



1D range query problem: split node

- 1D range query with $[61 : 90]$



1D range query problem: query algorithm

Algorithm 1DRangeQuery$(\mathcal{T}, [x : x'])$

1: $v_{\text{split}} \leftarrow$ FindSplitNode$(\mathcal{T}, x, x')$
2: **if** v_{split} is a leaf **then**
3: check if the point in v_{split} must be reported
4: **else**
5: $v \leftarrow lc(v_{\text{split}})$
6: **while** v is not a leaf **do**
7: **if** $x \leq x_v$ **then**
8: ReportSubtree$(rc(v))$
9: $v \leftarrow lc(v)$
10: **else**
11: $v \leftarrow rc(v)$
12: check if the point stored in v must be reported
13: similarly, follow the path to x', and report the left subtrees

1D range query problem: query running time analysis

The efficiency analysis: count the numbers of nodes visited for each type

1D range query problem: query running time analysis

The efficiency analysis: count the numbers of nodes visited for each type

White nodes: never visited by the query $\rightarrow$ no time spent

1D range query problem: query running time analysis

The efficiency analysis: count the numbers of nodes visited for each type

White nodes: never visited by the query $\rightarrow$ no time spent

Grey nodes: visited by the query, unclear if they lead to output

$\rightarrow$ time determines dependency on n (the number of nodes)

1D range query problem: query running time analysis

The efficiency analysis: count the numbers of nodes visited for each type

White nodes: never visited by the query $\rightarrow$ no time spent

Grey nodes: visited by the query, unclear if they lead to output

$\rightarrow$ time determines dependency on n (the number of nodes)

Black nodes: visited by the query, whole subtree is output

$\rightarrow$ time determines dependency on k (the output size)

1D range query problem: query running time analysis

The efficiency analysis: count the numbers of nodes visited for each type

White nodes: never visited by the query $\rightarrow$ no time spent

Grey nodes: visited by the query, unclear if they lead to output

- $\rightarrow$ time determines dependency on n (the number of nodes)
- $\rightarrow$ occur only on two paths in the tree, and since the tree is balanced, we have $O(\log n)$ grey nodes

Black nodes: visited by the query, whole subtree is output

- $\rightarrow$ time determines dependency on k (the output size)

1D range query problem: query running time analysis

The efficiency analysis: count the numbers of nodes visited for each type

White nodes: never visited by the query $\rightarrow$ no time spent

Grey nodes: visited by the query, unclear if they lead to output

- $\rightarrow$ time determines dependency on n (the number of nodes)
- $\rightarrow$ occur only on two paths in the tree, and since the tree is balanced, we have $O(\log n)$ grey nodes

Black nodes: visited by the query, whole subtree is output

- $\rightarrow$ time determines dependency on k (the output size)
- $\rightarrow$ a (sub)tree with m leaves has $m - 1$ internal nodes; algo visits $O(m)$ nodes and finds m points for the output

1D range query problem: query running time analysis

The efficiency analysis: count the numbers of nodes visited for each type

White nodes: never visited by the query $\rightarrow$ no time spent

Grey nodes: visited by the query, unclear if they lead to output

- $\rightarrow$ time determines dependency on n (the number of nodes)
- $\rightarrow$ occur only on two paths in the tree, and since the tree is balanced, we have $O(\log n)$ grey nodes

Black nodes: visited by the query, whole subtree is output

- $\rightarrow$ time determines dependency on k (the output size)
- $\rightarrow$ a (sub)tree with m leaves has $m - 1$ internal nodes; algo visits $O(m)$ nodes and finds m points for the output

The time spent at each node is $O(1) \Rightarrow O(\log n + k)$ query time

1D range query problem: preprocessing

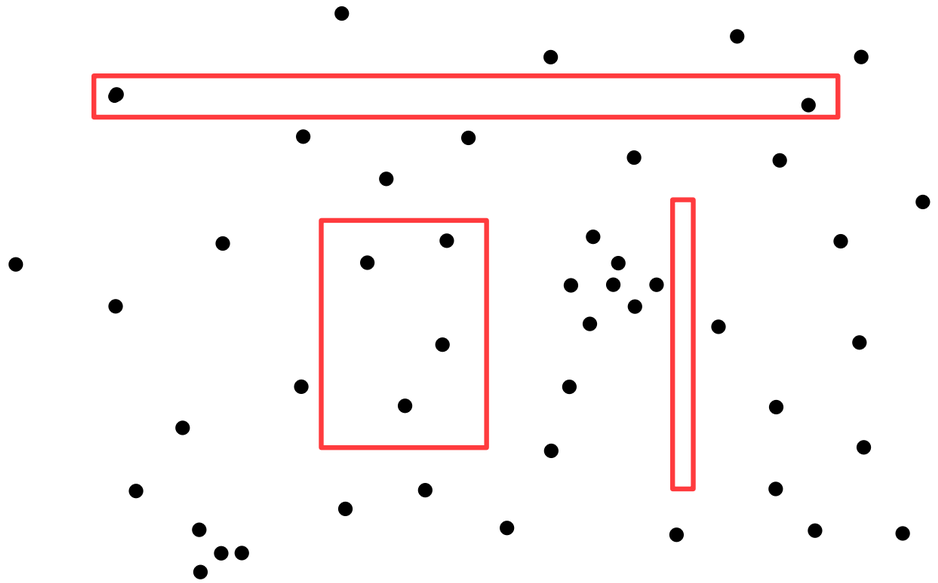
Preprocessing time: A (balanced) binary search tree storing n points can be built in $O(n \log n)$ time, or in $O(n)$ if the points are given in sorted order.

Result

Theorem: A set of n points on the real line can be preprocessed in $O(n \log n)$ time into a data structure of $O(n)$ size so that any 1D range query can be answered in $O(\log n + k)$ time, where k is the number of answers reported.

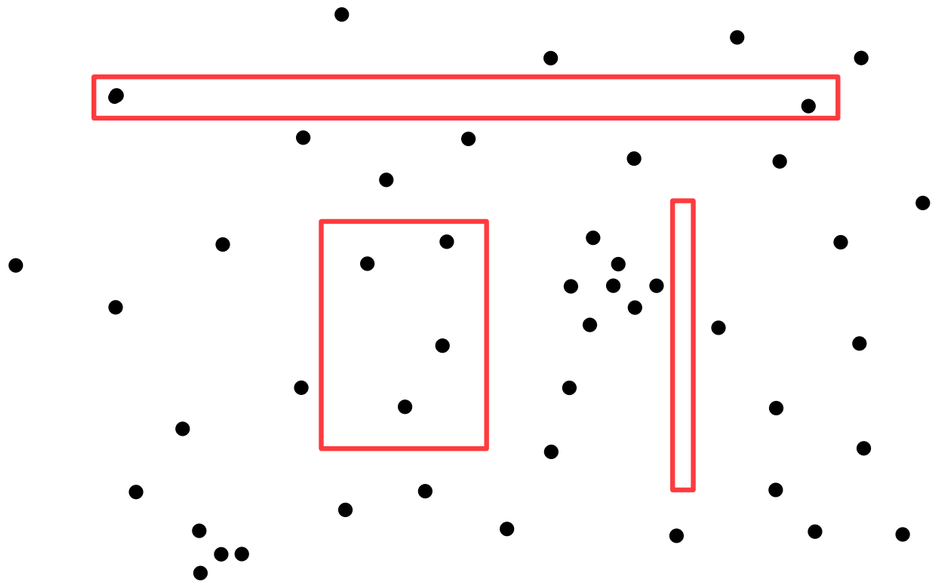
2D range query problem

Problem (2D range query): Preprocess a set of n points in 2D such that the ones inside a 2D query range (axis-aligned box) can be found fast.



2D range query problem

Problem (2D range query): Preprocess a set of n points in 2D such that the ones inside a 2D query range (axis-aligned box) can be found fast.



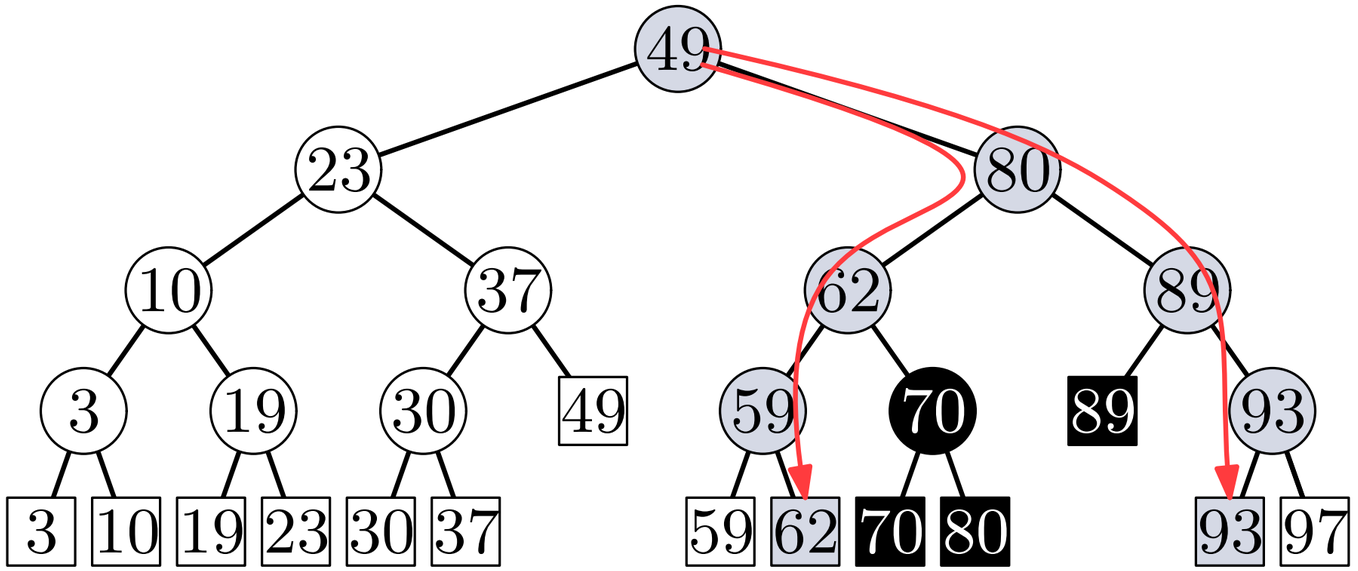
quadtrees: good in practice, but not so good worst-case query time

kd-trees: queries take $O(\sqrt{n} + k)$ (Chapter 5.2)

range trees: today

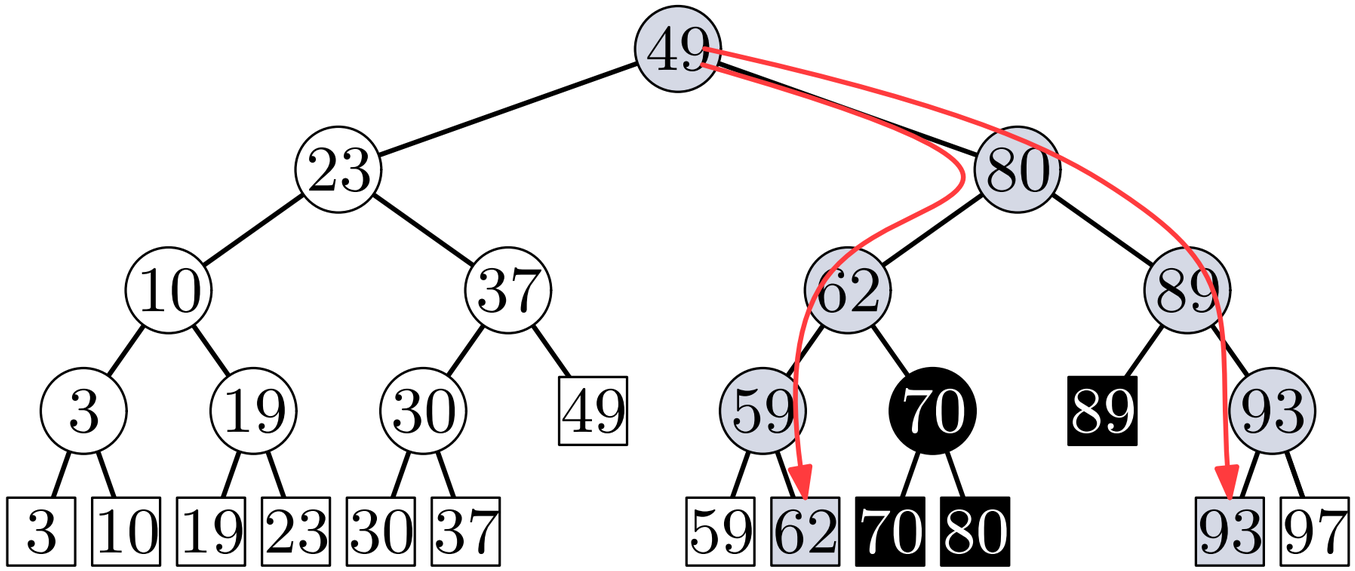
2D range query problem: back to 1D

- 1D range query with $[61 : 90]$



2D range query problem: back to 1D

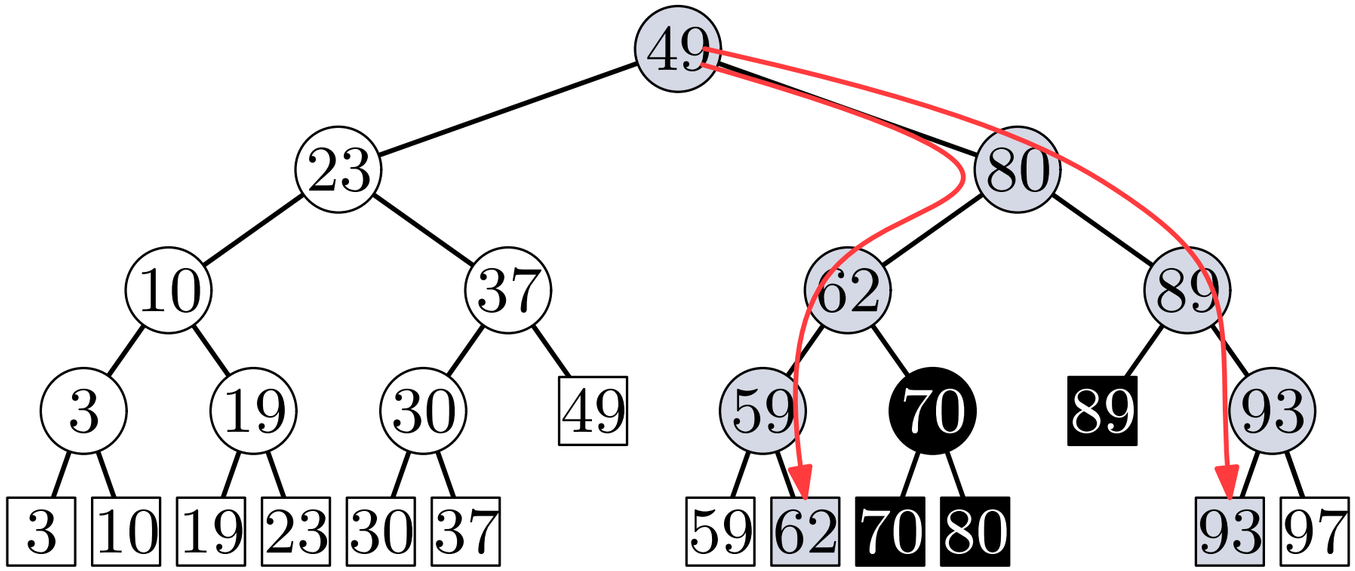
- 1D range query with $[61 : 90]$



Observation: Ignoring the search path leaves, all answers are jointly represented by the highest nodes strictly between the two search paths

2D range query problem: back to 1D

- 1D range query with $[61 : 90]$



Observation: Ignoring the search path leaves, all answers are jointly represented by the highest nodes strictly between the two search paths

Question: How many highest nodes between the search paths can there be?

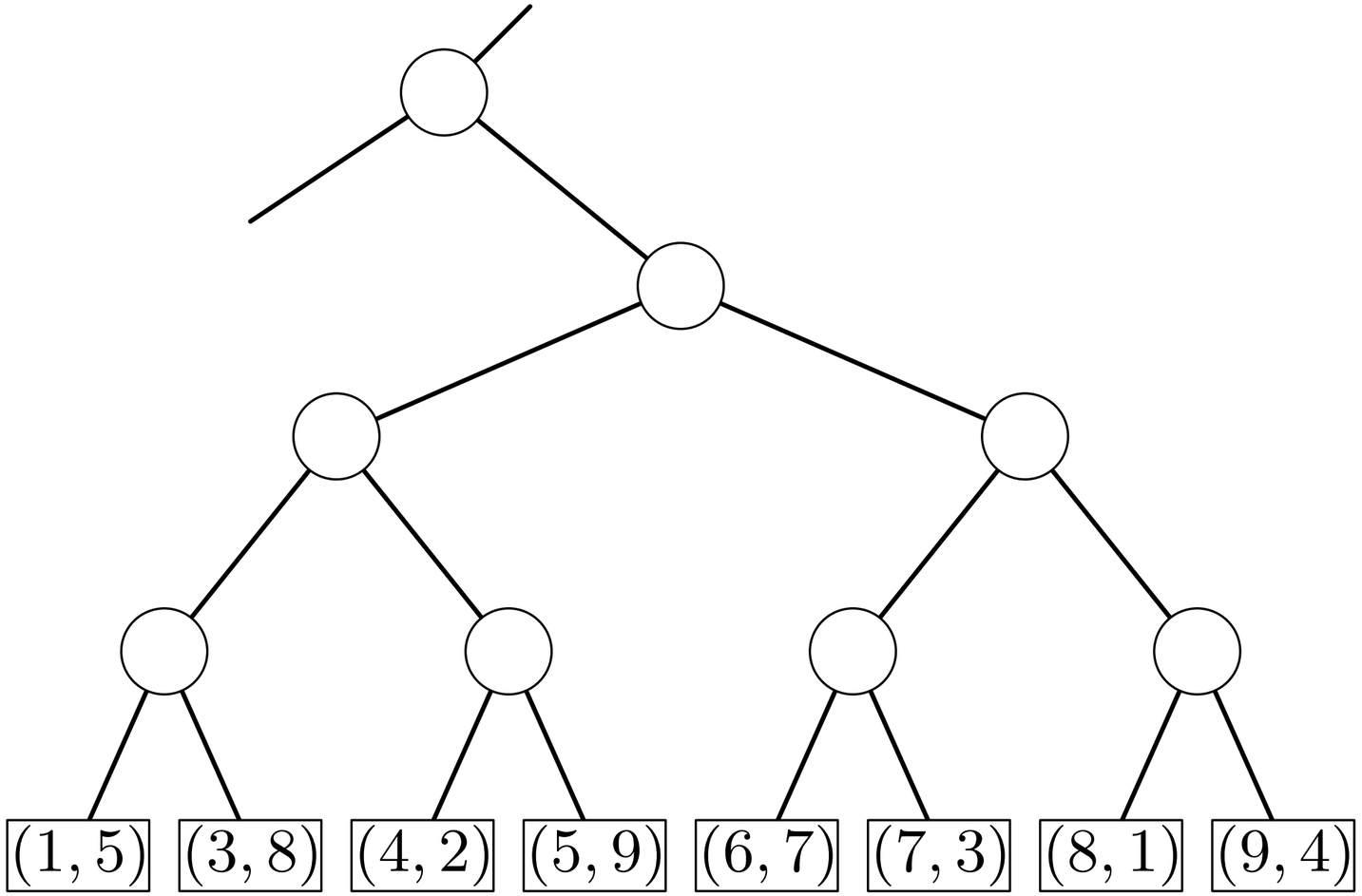
2D range query problem: examining 1D range queries

For any 1D range query, we can identify $O(\log n)$ nodes that together represent all answers to a 1D range query.

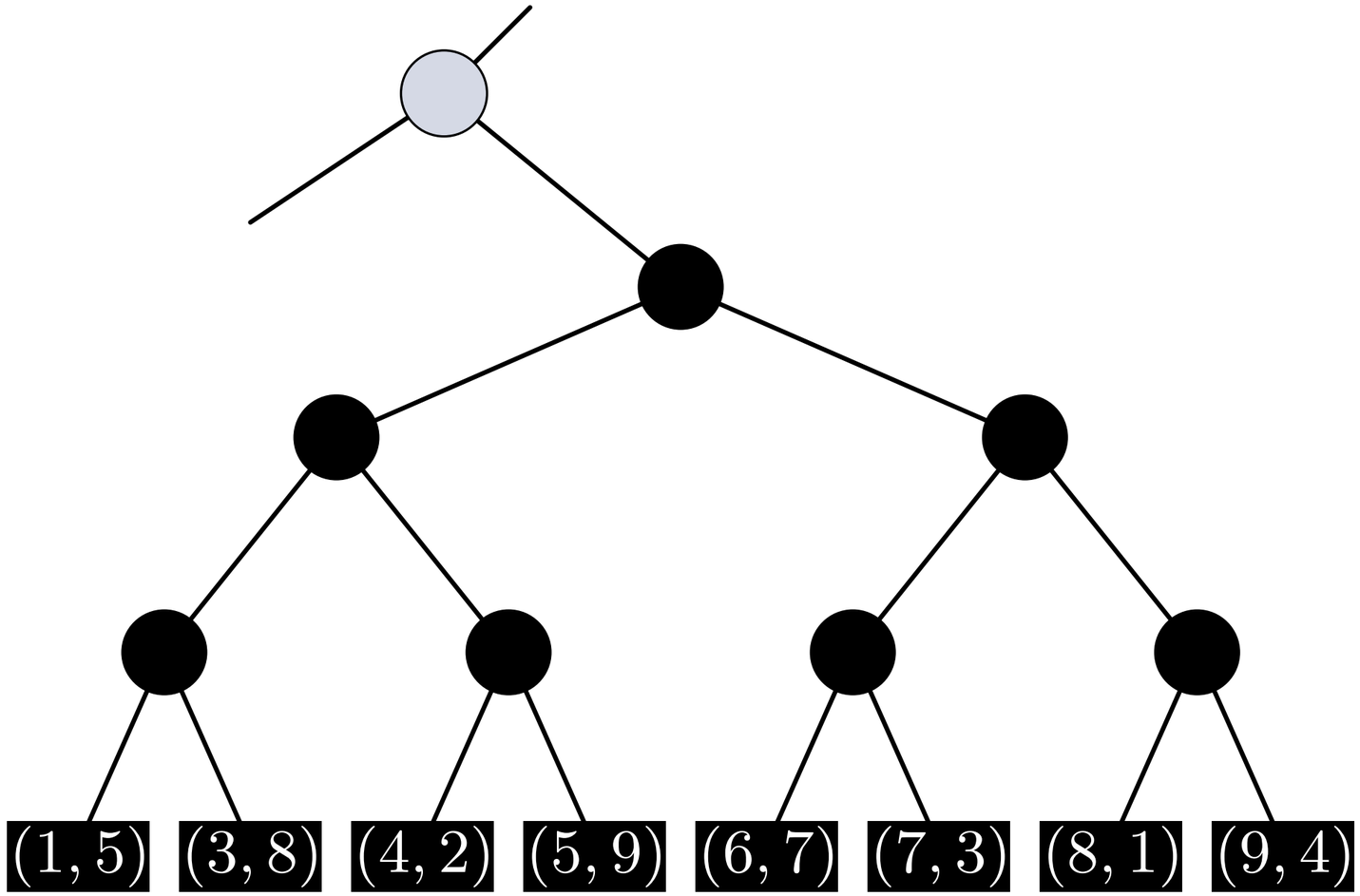
2D range query problem: toward 2D range queries

For any 2D range query, we can identify $O(\log n)$ nodes that together represent all points that have a correct first coordinate.

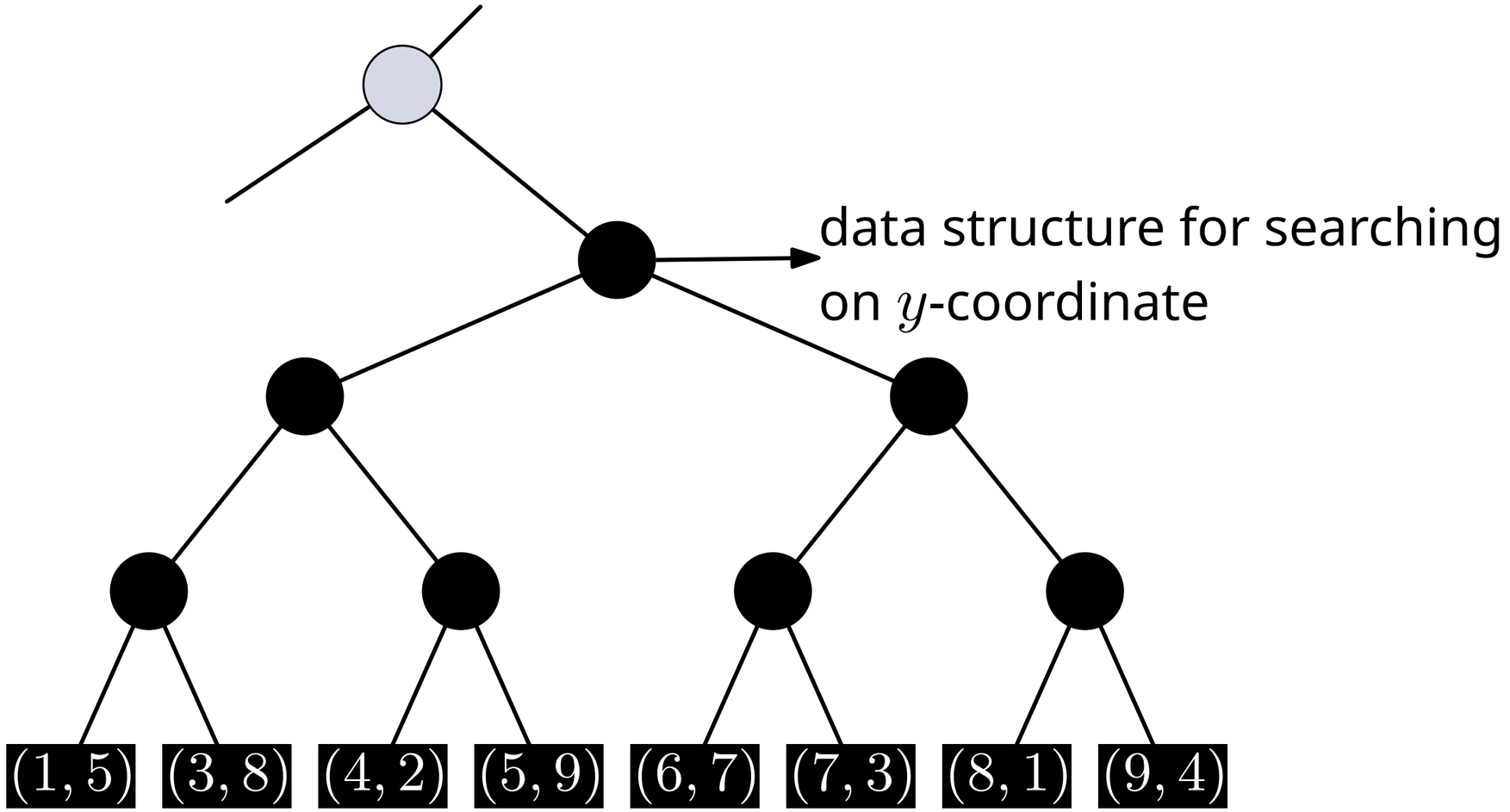
2D range query problem: toward 2D range queries



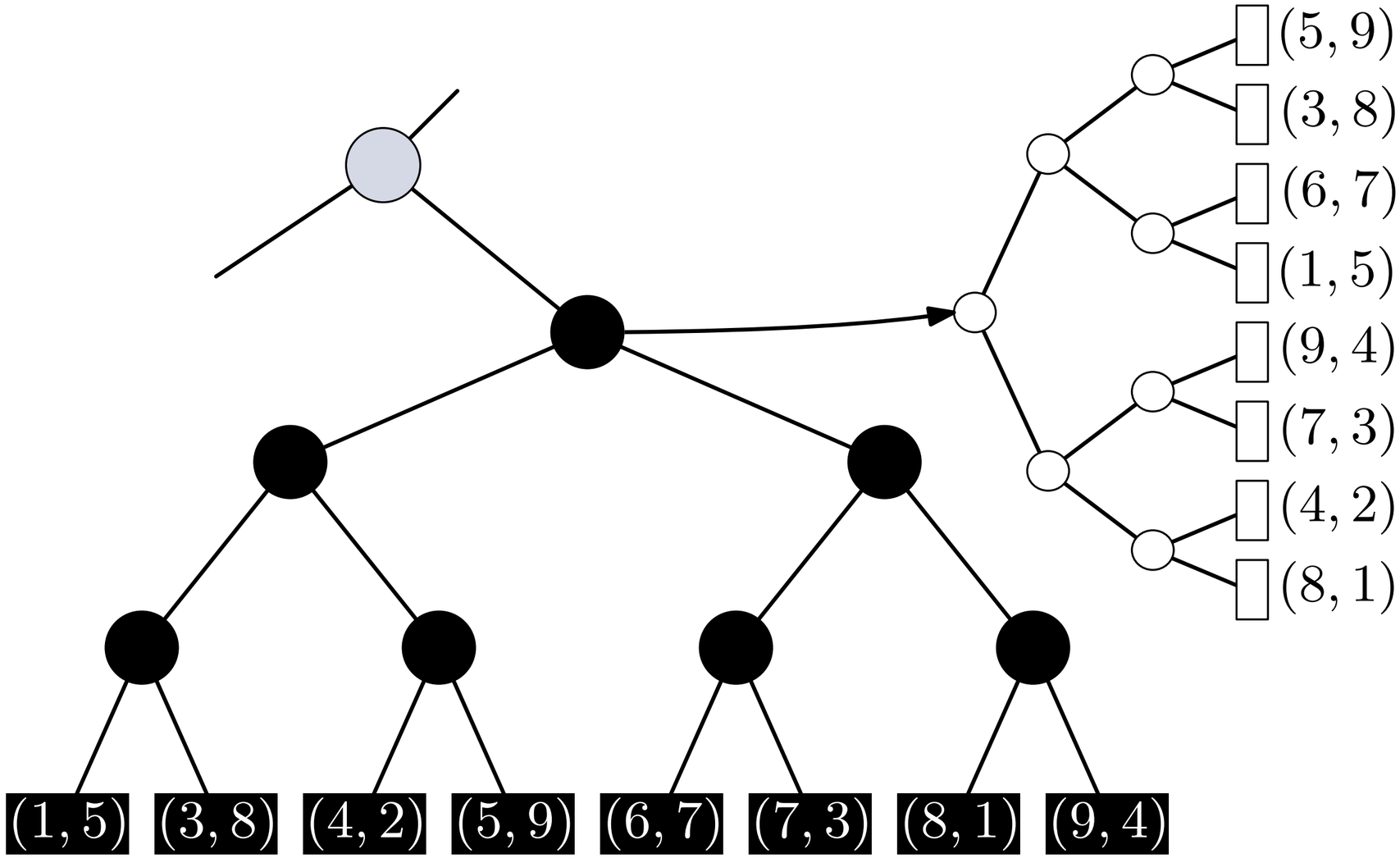
2D range query problem: toward 2D range queries



2D range query problem: toward 2D range queries



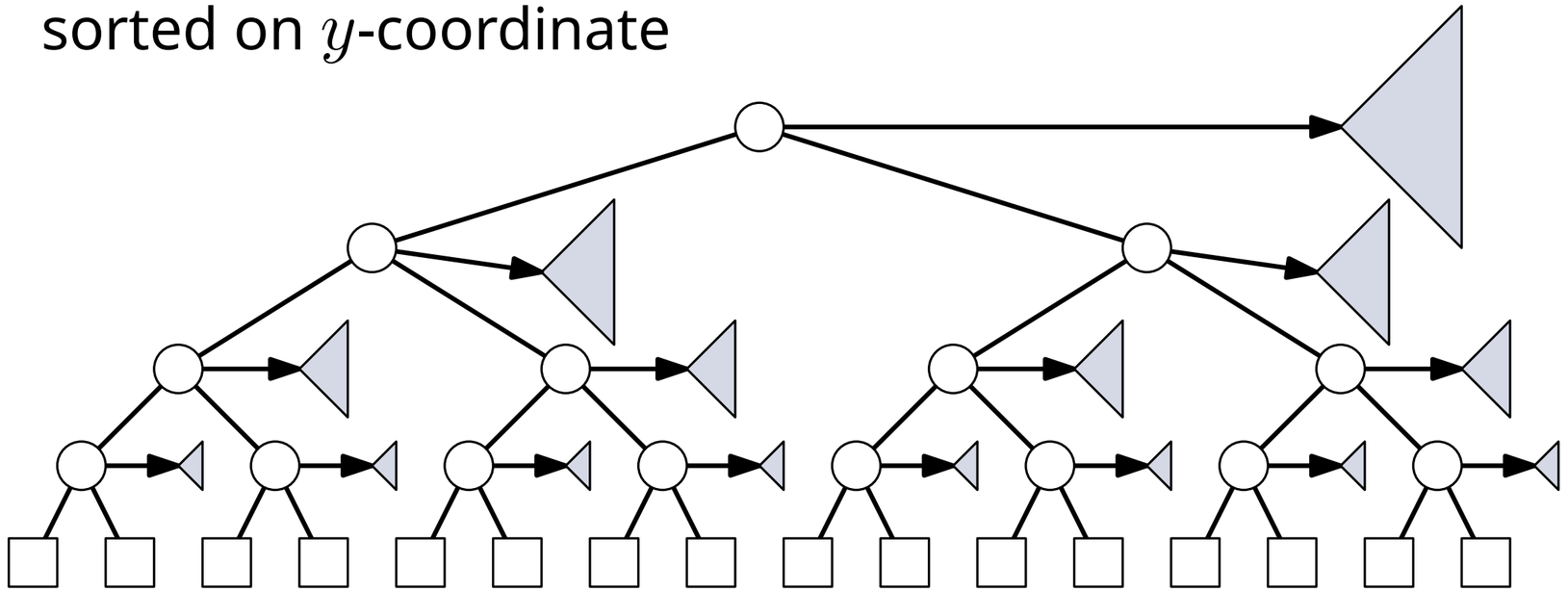
2D range query problem: toward 2D range queries



2D range query problem: data structure

Data structure: 2D range trees

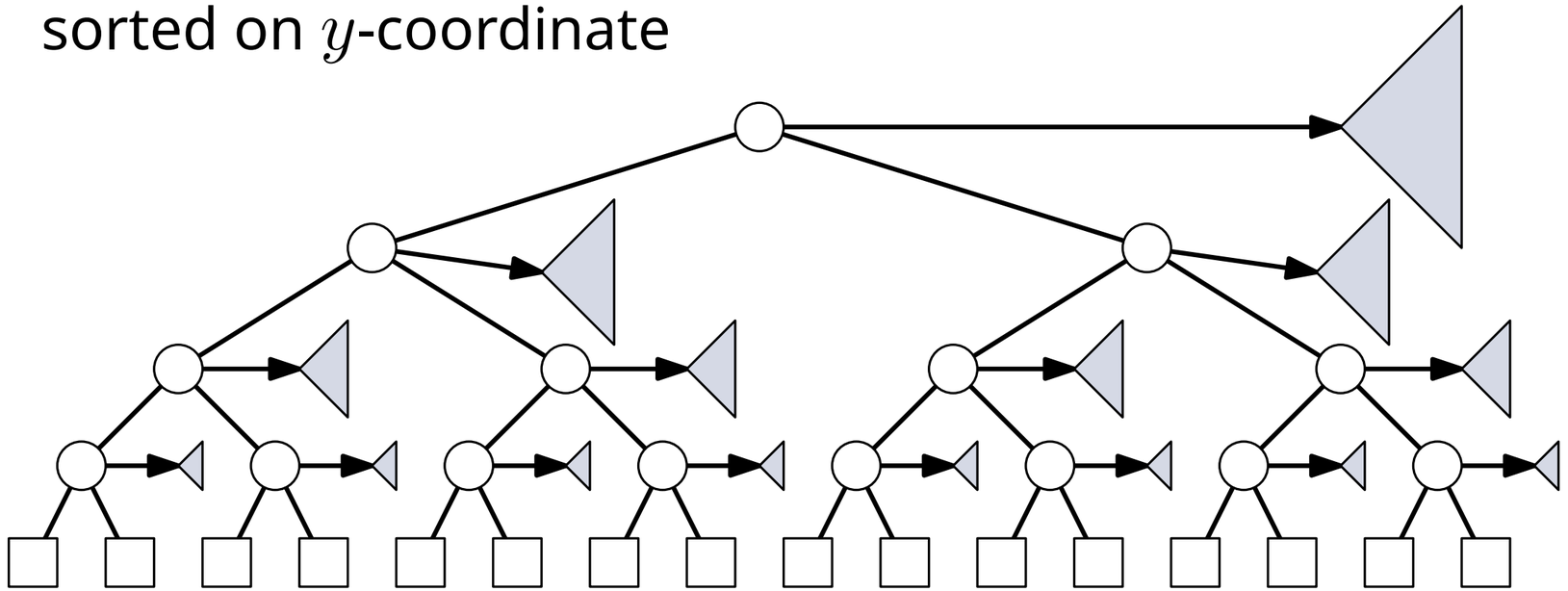
- balanced binary search tree on x-coordinate (with points in leafs)
- every internal node stores a tree in an associated structure, same points but sorted on y-coordinate



2D range query problem: data structure

Data structure: 2D range trees

- balanced binary search tree on x-coordinate (with points in leafs)
- every internal node stores a tree in an associated structure, same points but sorted on y-coordinate



Question: How much storage does this take?

2D range query problem: storage of 2D range trees

To analyze storage, two arguments can be used:

- By level: On each level, any point is stored exactly once. So all associated trees on one level together have $O(n)$ size.
- By point: For any point, it is stored in the associated structures of its search path. So it is stored in $O(\log n)$ of them.

2D range query problem: storage of 2D range trees

To analyze storage, two arguments can be used:

- By level: On each level, any point is stored exactly once. So all associated trees on one level together have $O(n)$ size.
- By point: For any point, it is stored in the associated structures of its search path. So it is stored in $O(\log n)$ of them.

Storage requirement: $O(n \log n)$

2D range query problem: construction algorithm

Algorithm BUILD2DRANGETREE(P)

1: construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P
2: **if** P contains only one point p **then**
3: create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v
4: **else**
5: split P into P_{left} and P_{right} by x-coordinate around median x_{mid}
6: $v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})
7: $v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})
8: create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v
9: **return** v

2D range query problem: construction algorithm

Algorithm BUILD2DRANGETREE(P)

1: construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P

2: **if** P contains only one point p **then** $O(1)$

3: create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v

4: **else**

5: split P into P_{left} and P_{right} by x-coordinate around median x_{mid}

6: $v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})

7: $v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})

8: create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v $O(1)$

9: **return** v

2D range query problem: construction algorithm

Algorithm BUILD2DRANGETREE(P)

1: construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P

2: **if** P contains only one point p **then** $O(1)$

3: create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v

4: **else**

5: split P into P_{left} and P_{right} by x-coordinate around median x_{mid}

6: $v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})

7: $v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})

8: create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v $O(1)$

9: **return** v

2D range query problem: construction algorithm

Algorithm BUILD2DRANGETREE(P)

1:	construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P	$O(n \log n)$
2:	**if** P contains only one point p **then**	$O(1)$
3:	create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	
4:	**else**	
5:	split P into P_{left} and P_{right} by x-coordinate around median x_{mid}	
6:	$v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})	
7:	$v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})	
8:	create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	$O(1)$
9:	**return** v	

2D range query problem: construction algorithm

Algorithm BUILD2DRANGETREE(P)

1:	construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P	$O(n \log n)$
2:	**if** P contains only one point p **then**	$O(1)$
3:	create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	
4:	**else**	
5:	split P into P_{left} and P_{right} by x-coordinate around median x_{mid}	$O(n)$
6:	$v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})	
7:	$v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})	
8:	create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	$O(1)$
9:	**return** v	

2D range query problem: construction algorithm

Algorithm BUILD2DRANGETREE(P)

1:	construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P	$O(n \log n)$
2:	**if** P contains only one point p **then**	$O(1)$
3:	create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	
4:	**else**	
5:	split P into P_{left} and P_{right} by x-coordinate around median x_{mid}	$O(n)$
6:	$v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})	$2T(n/2)$
7:	$v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})	
8:	create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	$O(1)$
9:	**return** v	

2D range query problem: efficiency of construction

The construction algorithm takes:

$$T(1) = O(1)$$

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

which solves to $O(n \log^2 n)$.

2D range query problem: efficiency of construction

The construction algorithm takes:

$$T(1) = O(1)$$

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

which solves to $O(n \log^2 n)$.

Question: Can we do better?

2D range query problem: efficiency of construction

The construction algorithm takes:

$$T(1) = O(1)$$

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

which solves to $O(n \log^2 n)$.

Question: Can we do better?

Idea: Suppose we pre-sort P on y-coordinate, and whenever we split P into P_{left} and P_{right}, we keep the y-order in both subsets...

2D range query problem: efficiency of construction

Algorithm BUILD2DRANGETREE(P)

1:	construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P	$O(n \log n)$
2:	**if** P contains only one point p **then**	$O(1)$
3:	create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	
4:	**else**	
5:	split P into P_{left} and P_{right} by x-coordinate around median x_{mid}	$O(n)$
6:	$v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})	$2T(n/2)$
7:	$v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})	
8:	create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	$O(1)$
9:	**return** v	

2D range query problem: efficiency of construction

0: presort P	$O(n \log n)$
Algorithm BUILD2DRANGETREE(P)	
1: construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P	$O(n \log n)$
2: **if** P contains only one point p **then**	$O(1)$
3: create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	
4: **else**	
5: split P into P_{left} and P_{right} by x-coordinate around median x_{mid}	$O(n)$
6: $v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})	$2T(n/2)$
7: $v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})	
8: create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v	$O(1)$
9: **return** v	

2D range query problem: efficiency of construction

0: presort P — $O(n \log n)$

Algorithm BUILD2DRANGETREE(P)

1: construct the associated structure: build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P — $O(n)$

2: **if** P contains only one point p **then** — $O(1)$

3: create a leaf v storing p, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v

4: **else**

5: split P into P_{left} and P_{right} by x-coordinate around median x_{mid} — $O(n)$

6: $v_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left}) — $2T(n/2)$

7: $v_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})

8: create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v — $O(1)$

9: **return** v

2D range query problem: efficiency of construction

The adapted construction algorithm takes:

$$T(1) = O(1)$$

$$T(n) = 2T(n/2) + O(n)$$

which solves to $O(n \log n)$.

2D range query problem: efficiency of construction

The adapted construction algorithm takes:

$$T(1) = O(1)$$

$$T(n) = 2T(n/2) + O(n)$$

which solves to $O(n \log n)$.

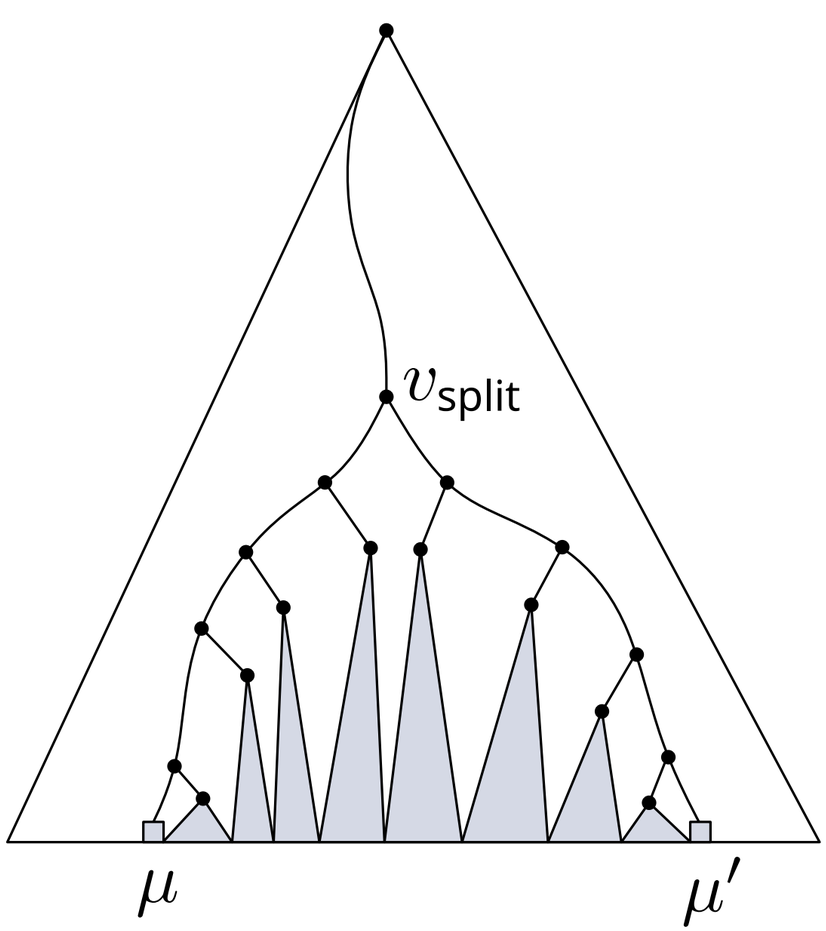
Total running time: $O(n \log n) + O(n \log n) = O(n \log n)$

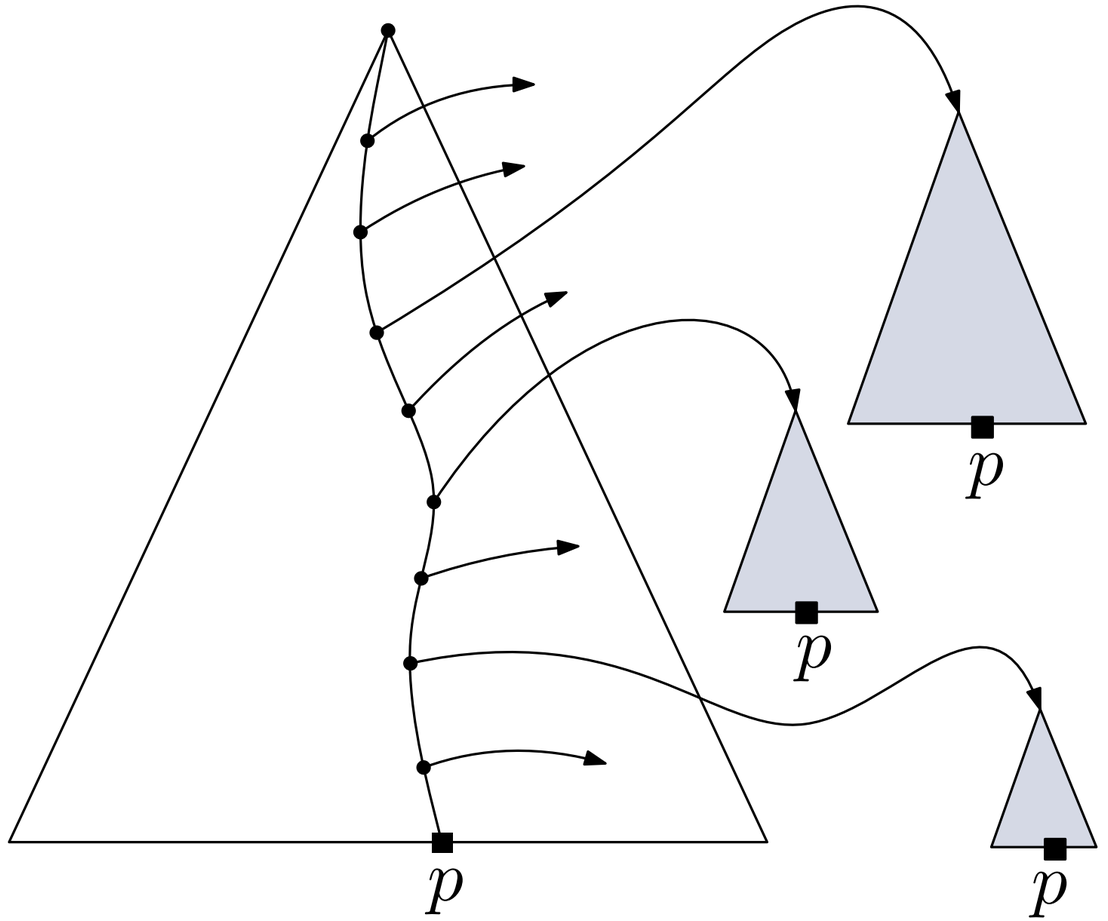
2D range query problem: query algorithm

How are queries performed and why are they correct?

- Are we sure that each answer is found?
- Are we sure that the same point is found only once?

2D range query problem: query algorithm





2D range query problem: query algorithm

Algorithm 2DRangeQuery($\mathcal{T}, [x : x'], [y : y']$)

1: $v_{\text{split}} \leftarrow$ FindSplitNode($\mathcal{T}, x, x'$)
2: **if** v_{split} is a leaf **then**
3: report the point stored at v_{split} if needed
4: **else**
5: $v \leftarrow lc(v_{\text{split}})$
6: **while** v is not a leaf **do**
7: **if** $x \leq x_v$ **then**
8: 1DRangeQuery($\mathcal{T}(rc(v)), [y : y']$)
9: $v \leftarrow lc(v)$
10: **else**
11: $v \leftarrow rc(v)$
12: check if the point stored at v must be reported
13: similarly, follow the path to x', and report the left subtrees

2D range query problem: query algorithm

Algorithm 2DRangeQuery($\mathcal{T}, [x : x'], [y : y']$)

1: $v_{\text{split}} \leftarrow$ FindSplitNode($\mathcal{T}, x, x'$)
2: **if** v_{split} is a leaf **then**
3: report the point stored at v_{split} if needed
4: **else**
5: $v \leftarrow lc(v_{\text{split}})$
6: **while** v is not a leaf **do**
7: **if** $x \leq x_v$ **then**
8: 1DRangeQuery($\mathcal{T}(rc(v)), [y : y']$)
9: $v \leftarrow lc(v)$
10: **else**
11: $v \leftarrow rc(v)$
12: check if the point stored at v must be reported
13: similarly, follow the path to x', and report the left subtrees

2D range query time

Question: How much time does a 2D range query take?

2D range query time

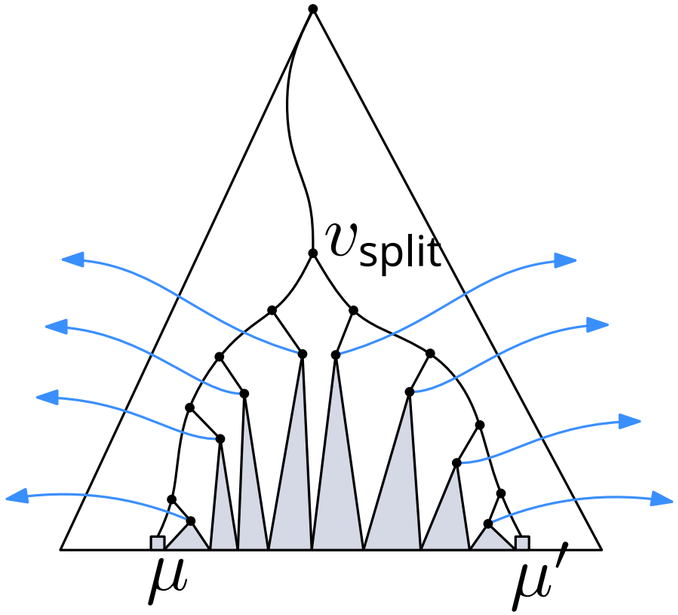
Question: How much time does a 2D range query take?

Subquestions: In how many associated structures do we search? How much time does each such search take?

2D range query time

Question: How much time does a 2D range query take?

Subquestions: In how many associated structures do we search? How much time does each such search take?



2D range query time

Question: How much time does a 2D range query take?

Subquestions: In how many associated structures do we search? How much time does each such search take?

We search in $O(\log n)$ associated structures to perform a 1D range query; at most two per level of the main tree

2D range query time

Question: How much time does a 2D range query take?

Subquestions: In how many associated structures do we search? How much time does each such search take?

We search in $O(\log n)$ associated structures to perform a 1D range query; at most two per level of the main tree

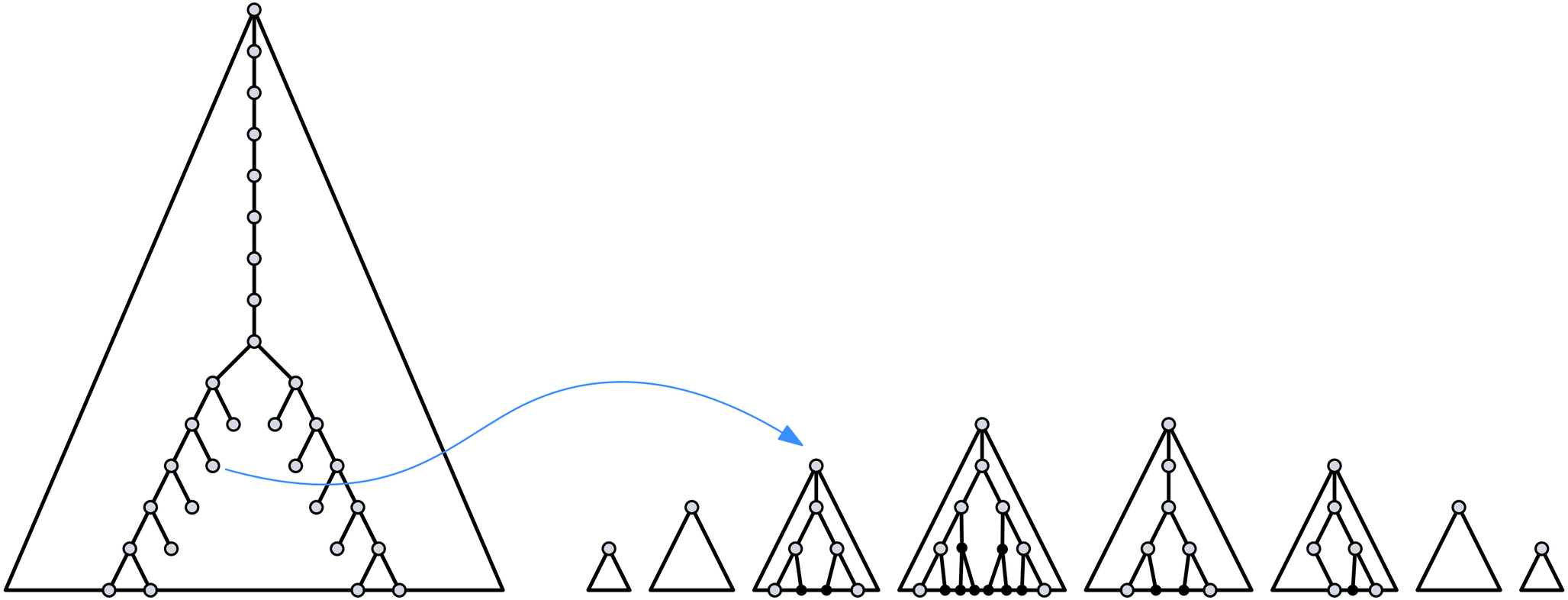
Query time is $O(\log n) \times O(\log m + k')$, or

$$\sum_v O(\log n_v + k_v)\,,$$

where $\sum k_v = k$ the number of points reported

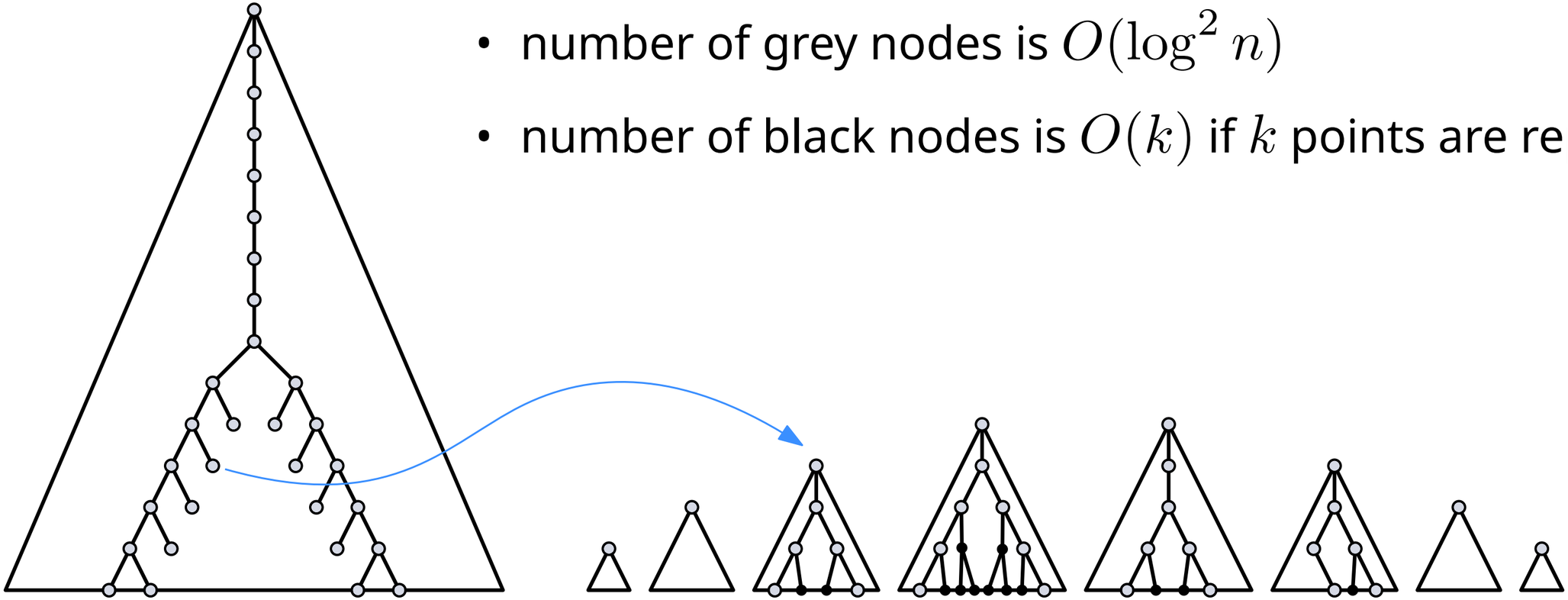
2D range query time

Use the concept of grey and black nodes again:



2D range query time

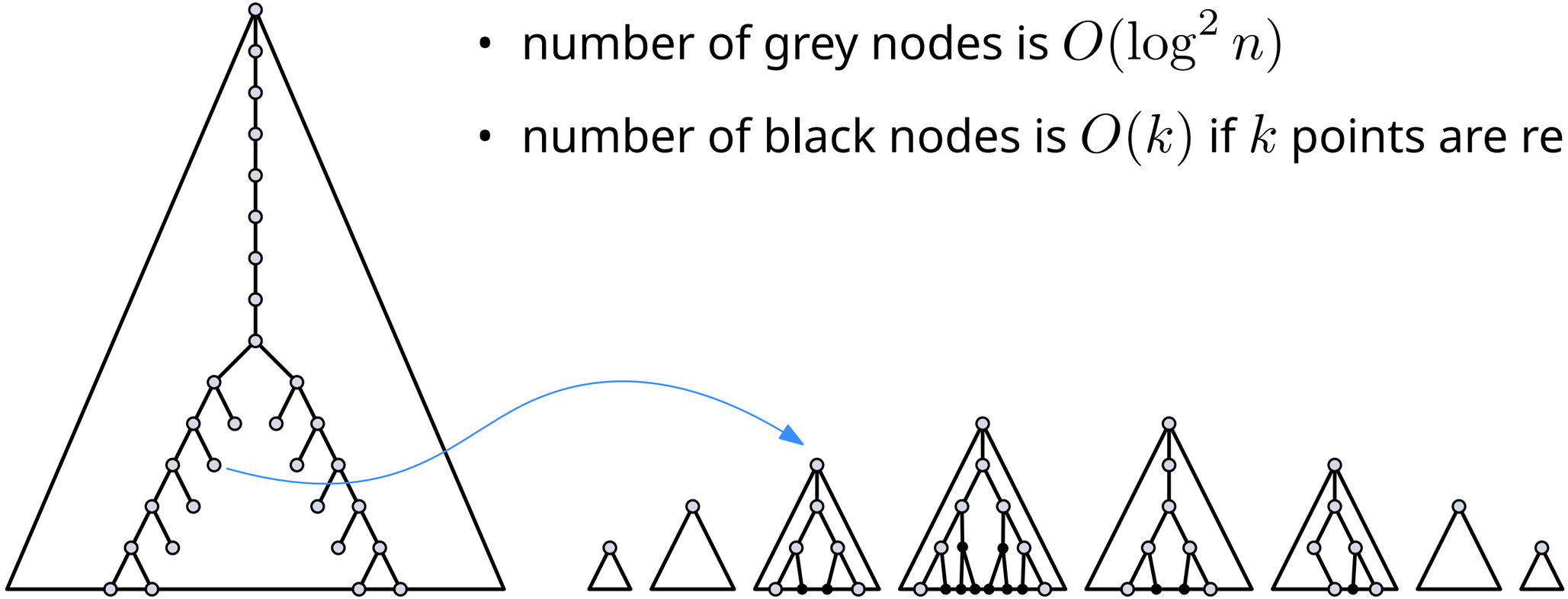
Use the concept of grey and black nodes again:



- number of grey nodes is $O(\log^2 n)$
- number of black nodes is $O(k)$ if k points are reported

2D range query time

Use the concept of grey and black nodes again:



- number of grey nodes is $O(\log^2 n)$
- number of black nodes is $O(k)$ if k points are reported

The query time is $O(\log^2 n + k)$, where k is the size of the output

Result

Theorem: A set of n points in the plane can be preprocessed in $O(n \log n)$ time into a data structure of $O(n \log n)$ size so that any 2D range query can be answered in $O(\log^2 n + k)$ time, where k is the number of points reported.

Result

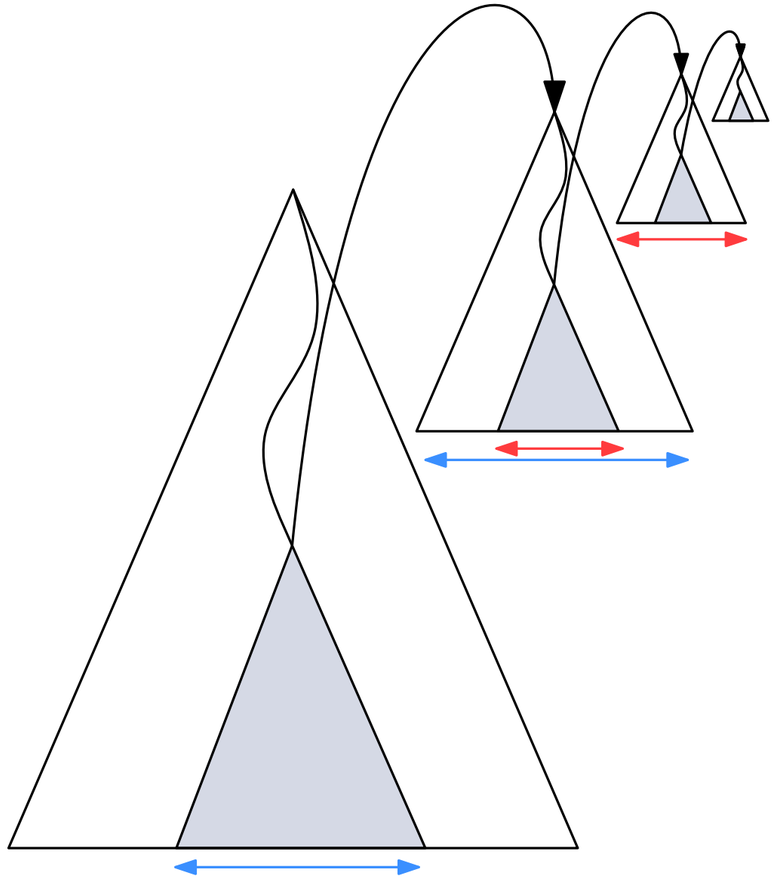
Theorem: A set of n points in the plane can be preprocessed in $O(n \log n)$ time into a data structure of $O(n \log n)$ size so that any 2D range query can be answered in $O(\log^2 n + k)$ time, where k is the number of points reported.

In contrast, a kd-tree has $O(n)$ size and answers queries in $O(\sqrt{n} + k)$ time (Chapter 5.2)

Higher dimensional range trees

A d-dimensional range tree has a main tree:

- a one-dimensional balanced binary search tree on the first coordinate,
- every node has a pointer to an associated structure which is a $(d-1)$-dimensional range tree on the other coordinates.



Storage

The size $S_d(n)$ of a d-dimensional range tree satisfies:

$$S_1(n) = O(n) \quad \text{for all } n$$

$$S_d(1) = O(1) \quad \text{for all } d$$

$$S_d(n) \leq 2 \cdot S_d(n/2) + S_{d-1}(n) \quad \text{for } d \geq 2$$

This solves to $S_d(n) = O(n \log^{d-1} n)$.

Query time

The number of grey nodes $G_d(n)$ satisfies:

$$G_1(n) = O(\log n) \quad \text{for all } n$$

$$G_d(1) = O(1) \qquad \text{for all } d$$

$$G_d(n) \leq 2 \cdot \log n + 2 \cdot \log n \cdot G_{d-1}(n) \quad \text{for } d \geq 2$$

This solves to $G_d(n) = O(\log^d n)$.

Result

Theorem: A set of n points in d-dimensional space with $d \geq 2$ can be preprocessed in $O(n \log^{d-1} n)$ time into a data structure of $O(n \log^{d-1} n)$ size so that any d-dimensional range query can be answered in $O(\log^d n + k)$ time, where k is the number of answers reported.

Improving the query time

We can improve the query time of a 2D range tree from $O(\log^2 n)$ to $O(\log n)$ by a technique called fractional cascading.

This automatically lowers the query time in d dimensions to $O(\log^{d-1} n)$ time.

Improving the query time

We can improve the query time of a 2D range tree from $O(\log^2 n)$ to $O(\log n)$ by a technique called fractional cascading.

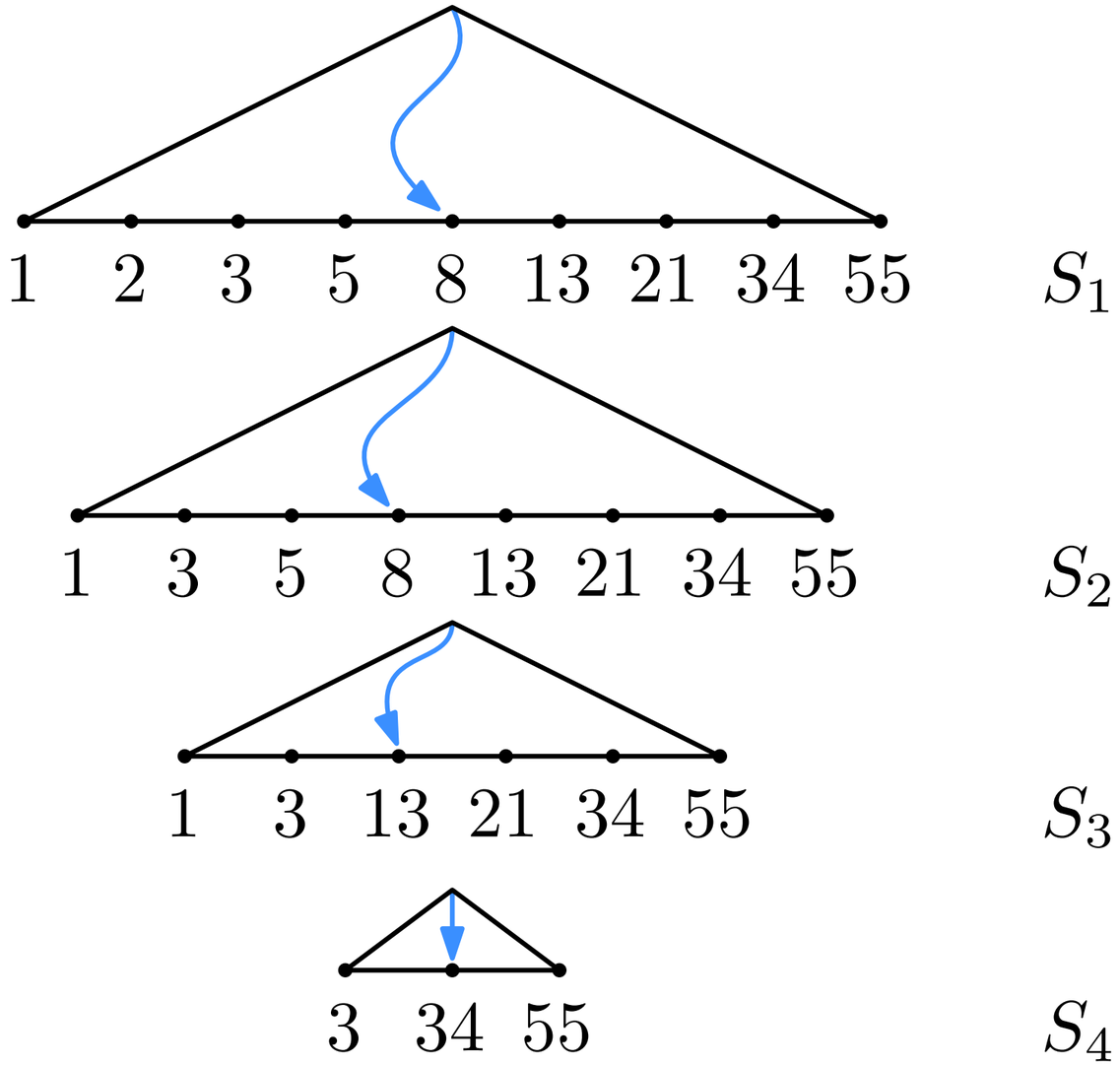
This automatically lowers the query time in d dimensions to $O(\log^{d-1} n)$ time.

The idea illustrated best by a different query problem:

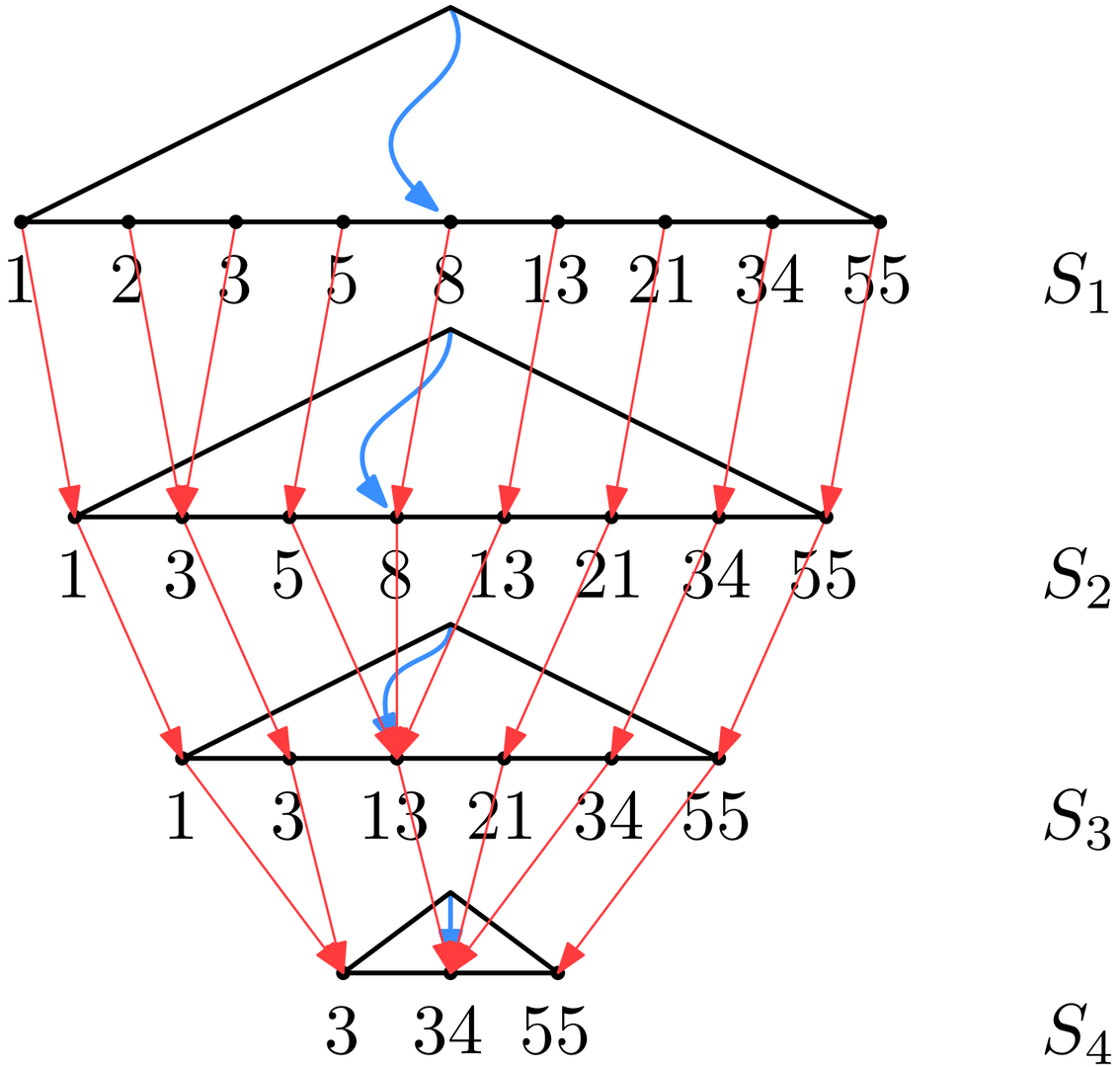
Suppose that we have a collection of sets $S_1, \ldots, S_m$, where $|S_1| = n$ and where $S_{i+1} \subseteq S_i$

We want a data structure that can report for a query number x, the smallest value $\geq x$ in all sets $S_1, \ldots, S_m$

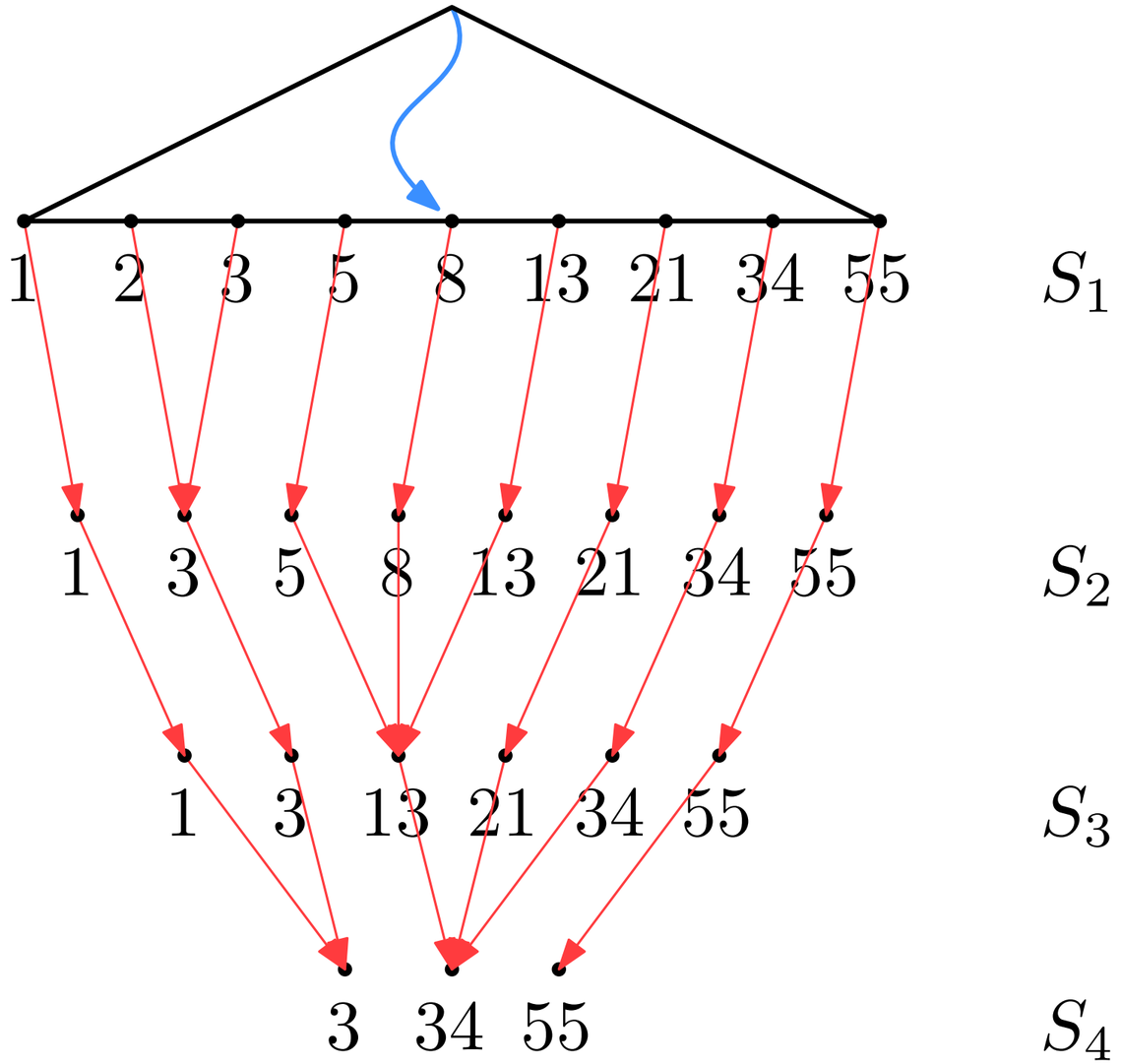
Improving the query time



Improving the query time



Improving the query time



Improving the query time

Suppose that we have a collection of sets $S_1, \ldots, S_m$, where $|S_1| = n$ and where $S_{i+1} \subseteq S_i$

We want a data structure that can report for a query value x, the smallest value $\geq x$ in all sets $S_1, \ldots, S_m$

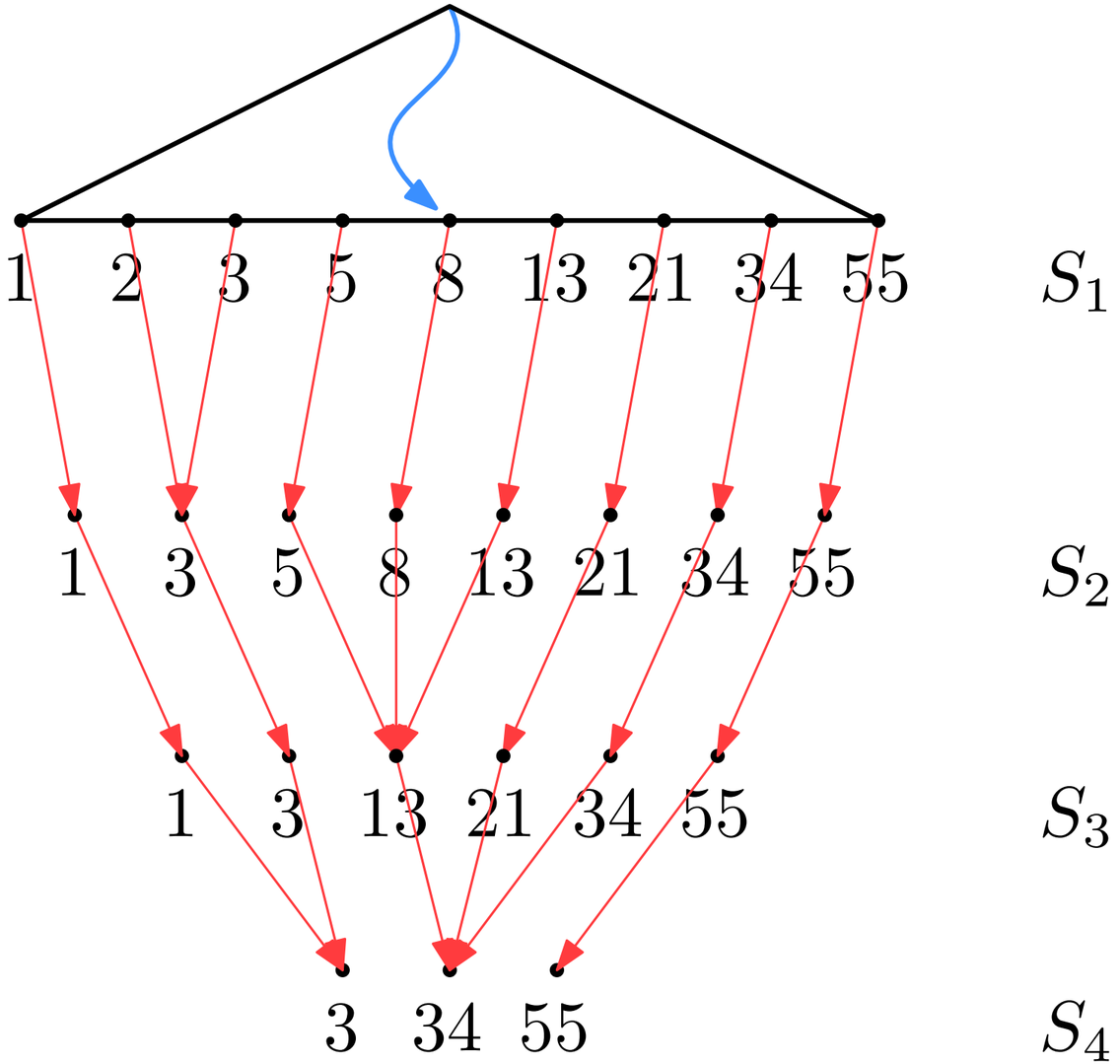
This query problem can be solved in $O(\log n + m)$ time instead of $O(m \cdot \log n)$ time

Improving the query time

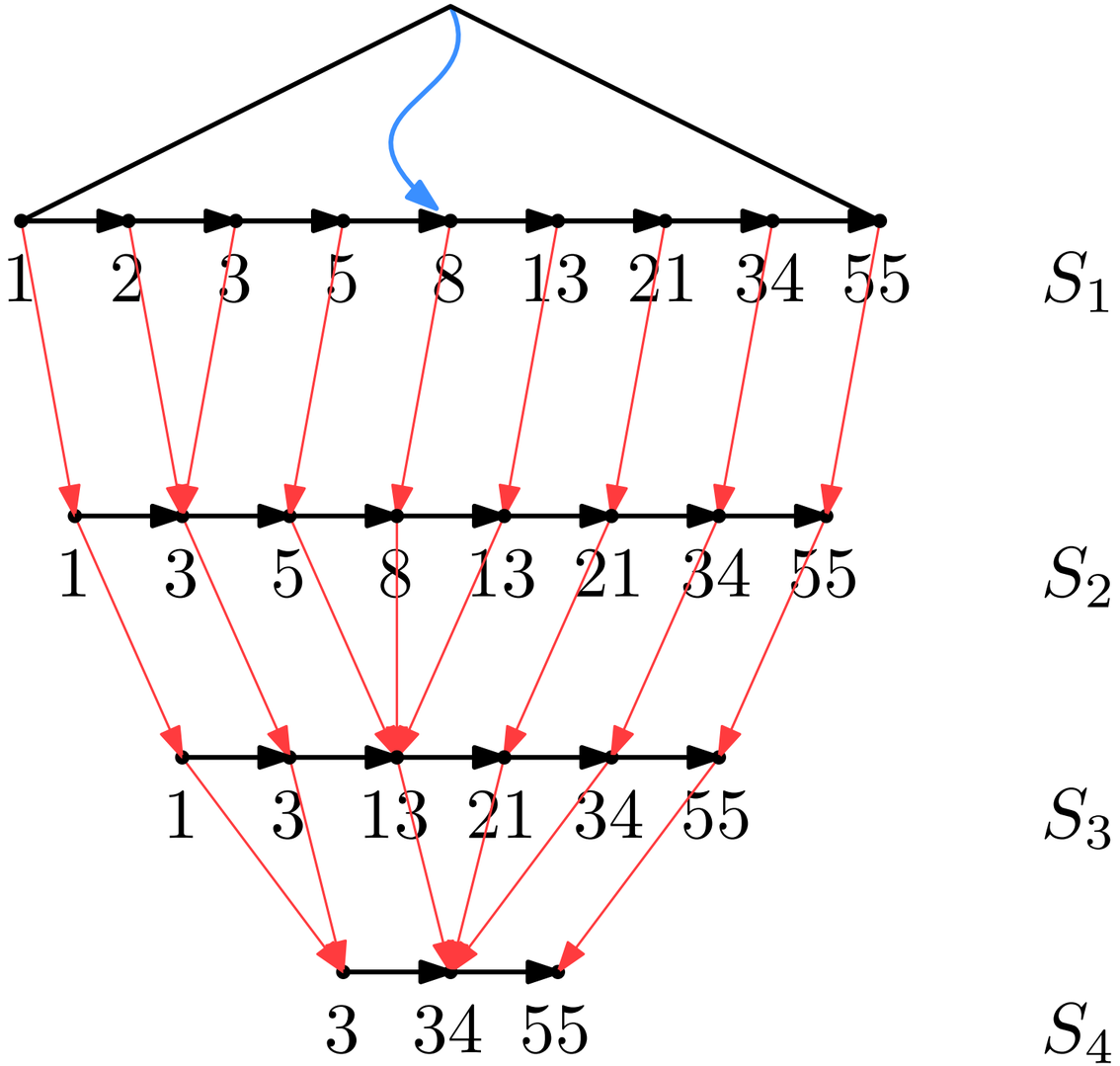
Can we do something similar for m 1-dimensional range queries on m sets $S_1, \ldots, S_m$?

We hope to get a query time of $O(\log n + m + k)$ with k the total number of points reported

Improving the query time

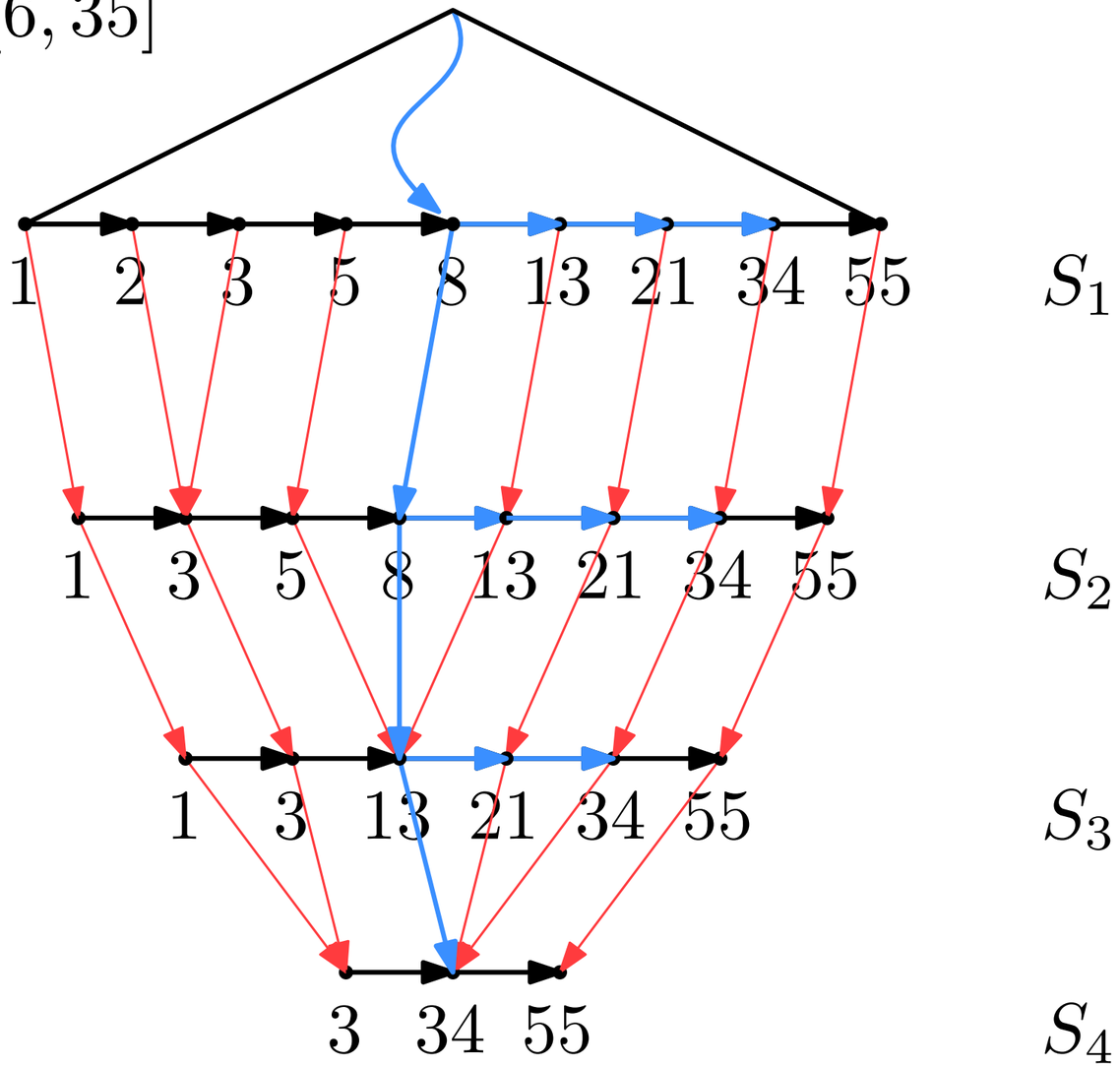


Improving the query time



Improving the query time

- range query for $[6, 35]$



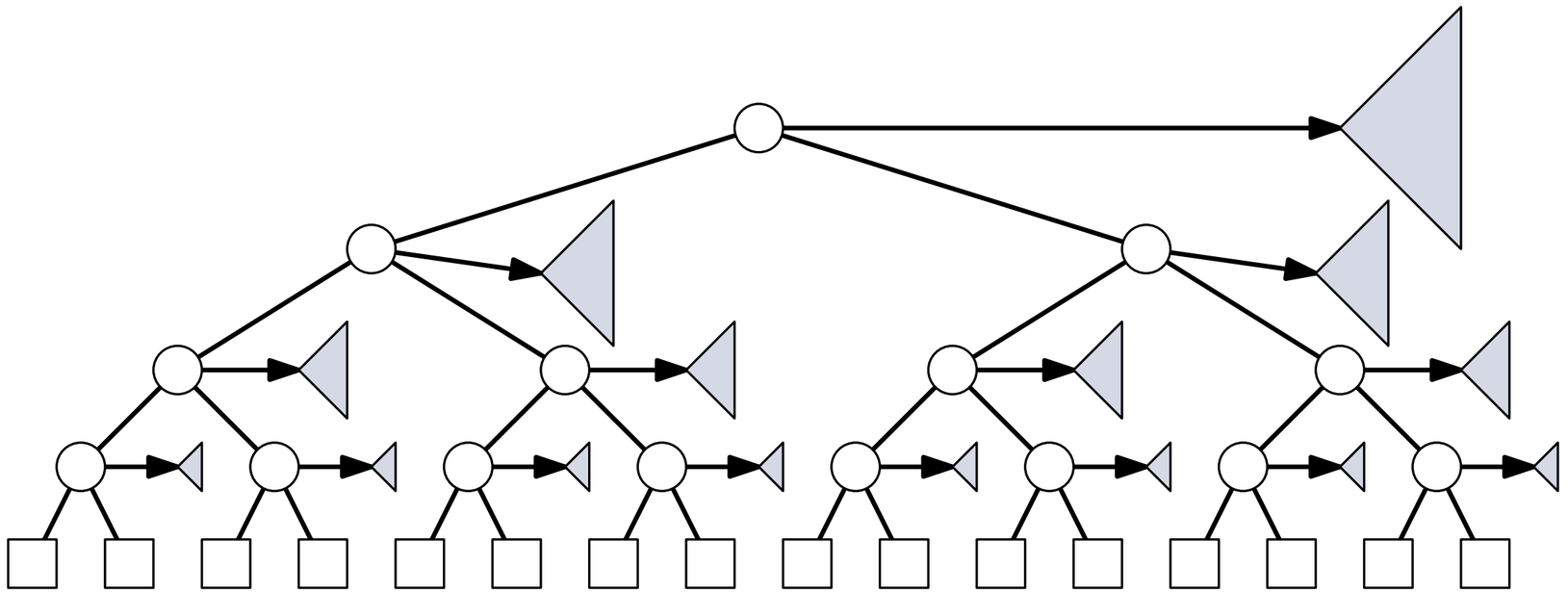
Fractional cascading

Now we do “the same” on the associated structures of a 2-dimensional range tree

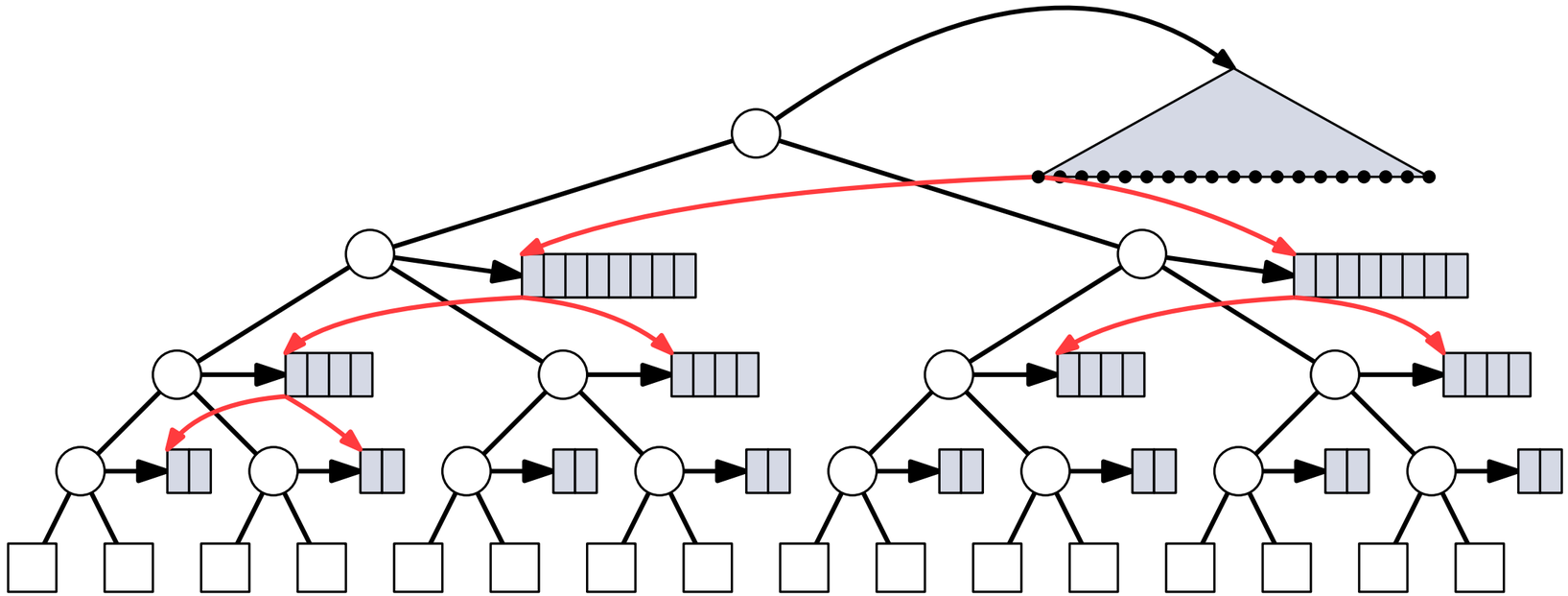
Note that in every associated structure, we search with the same values y and y'

- Replace all associated structures except for the root by a linked list
- For every list element (and leaf of the associated structure of the root), store two pointers to the appropriate list elements in the lists of the left child and of the right child

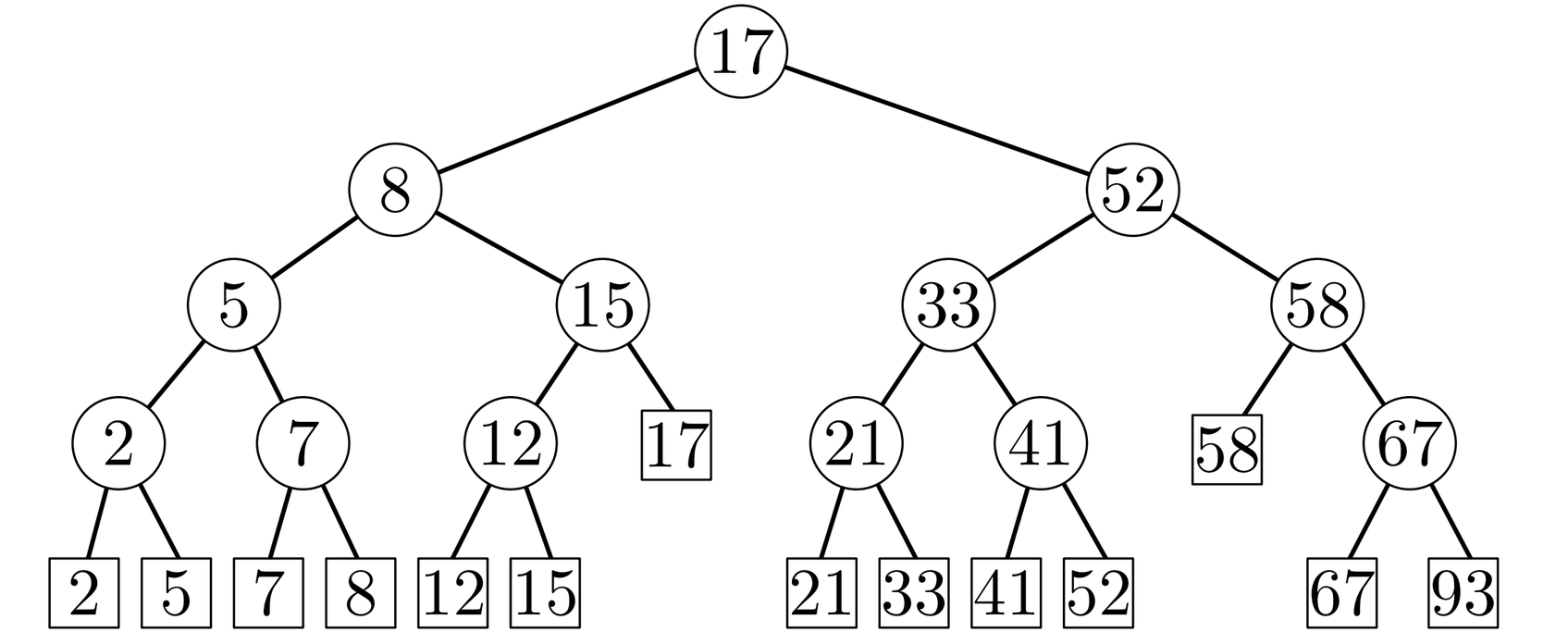
Fractional cascading



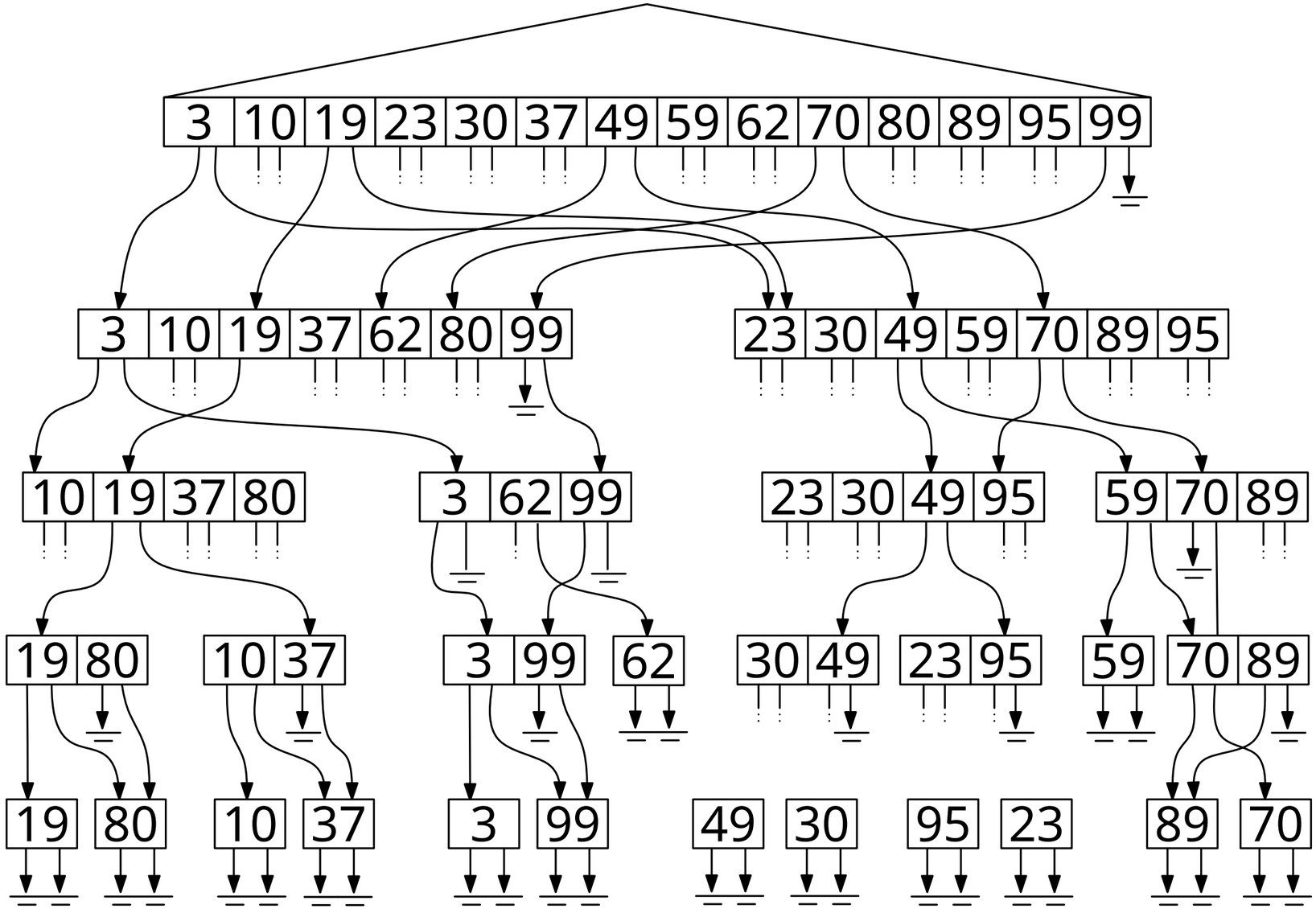
Fractional cascading



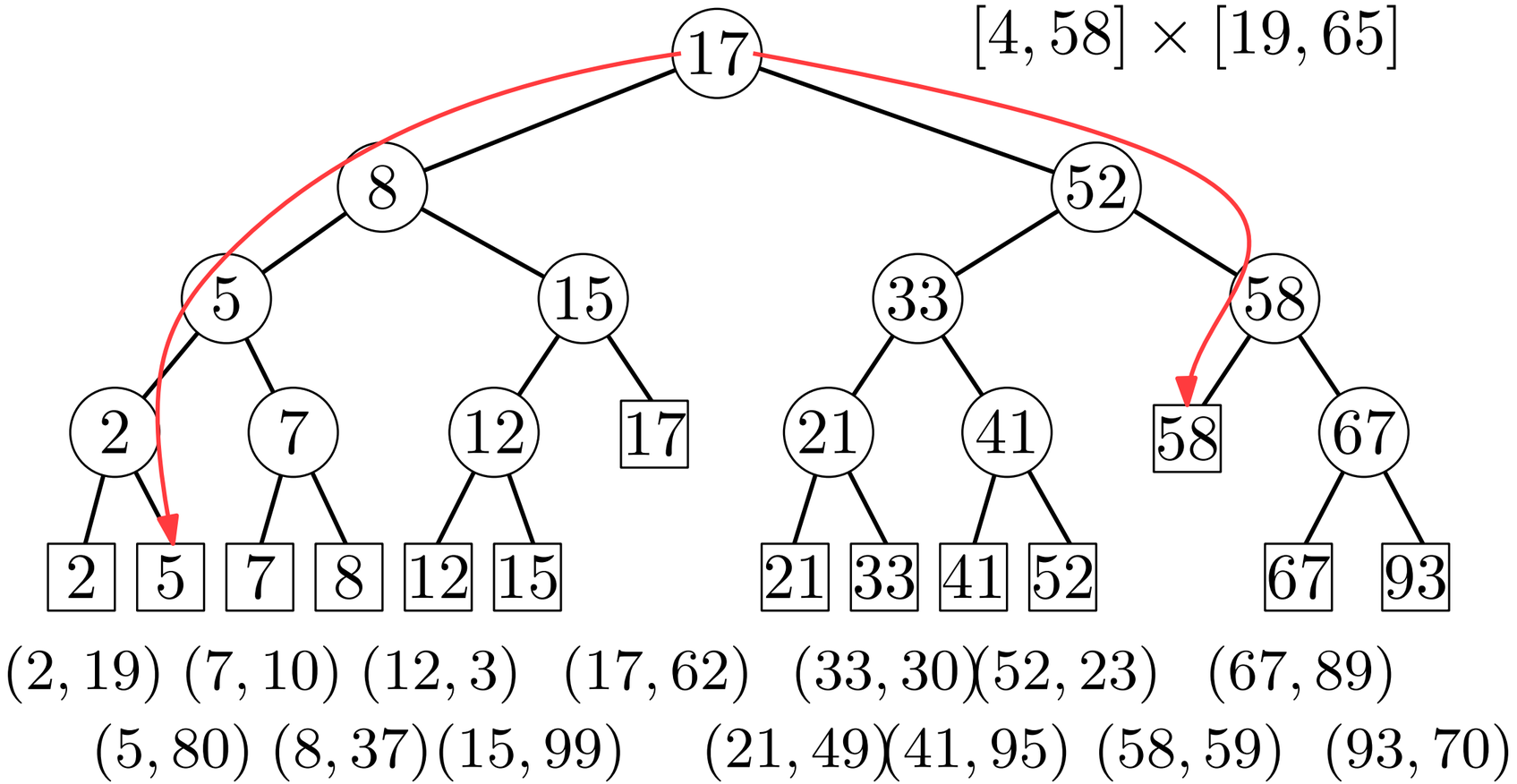
Fractional cascading



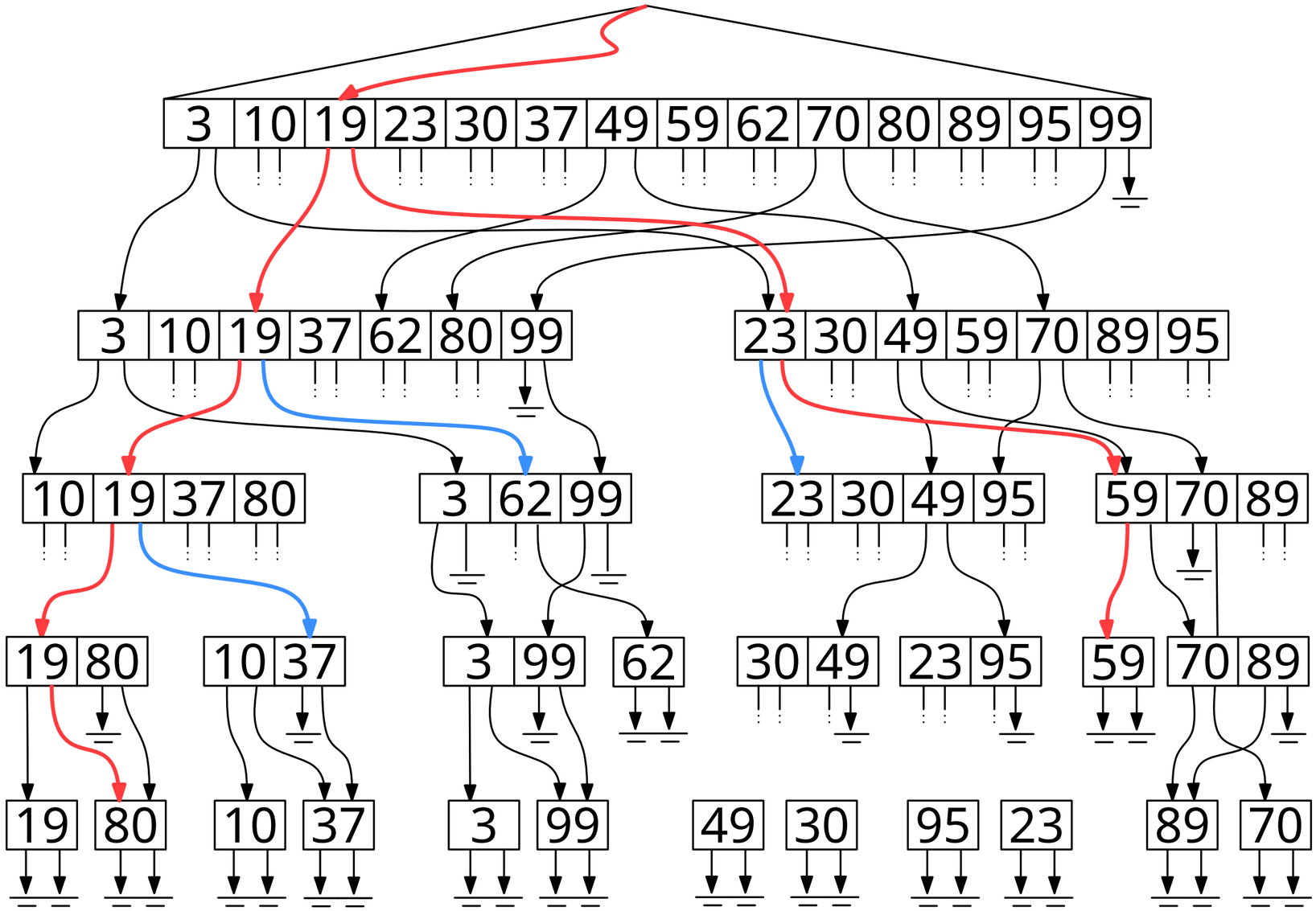
Fractional cascading



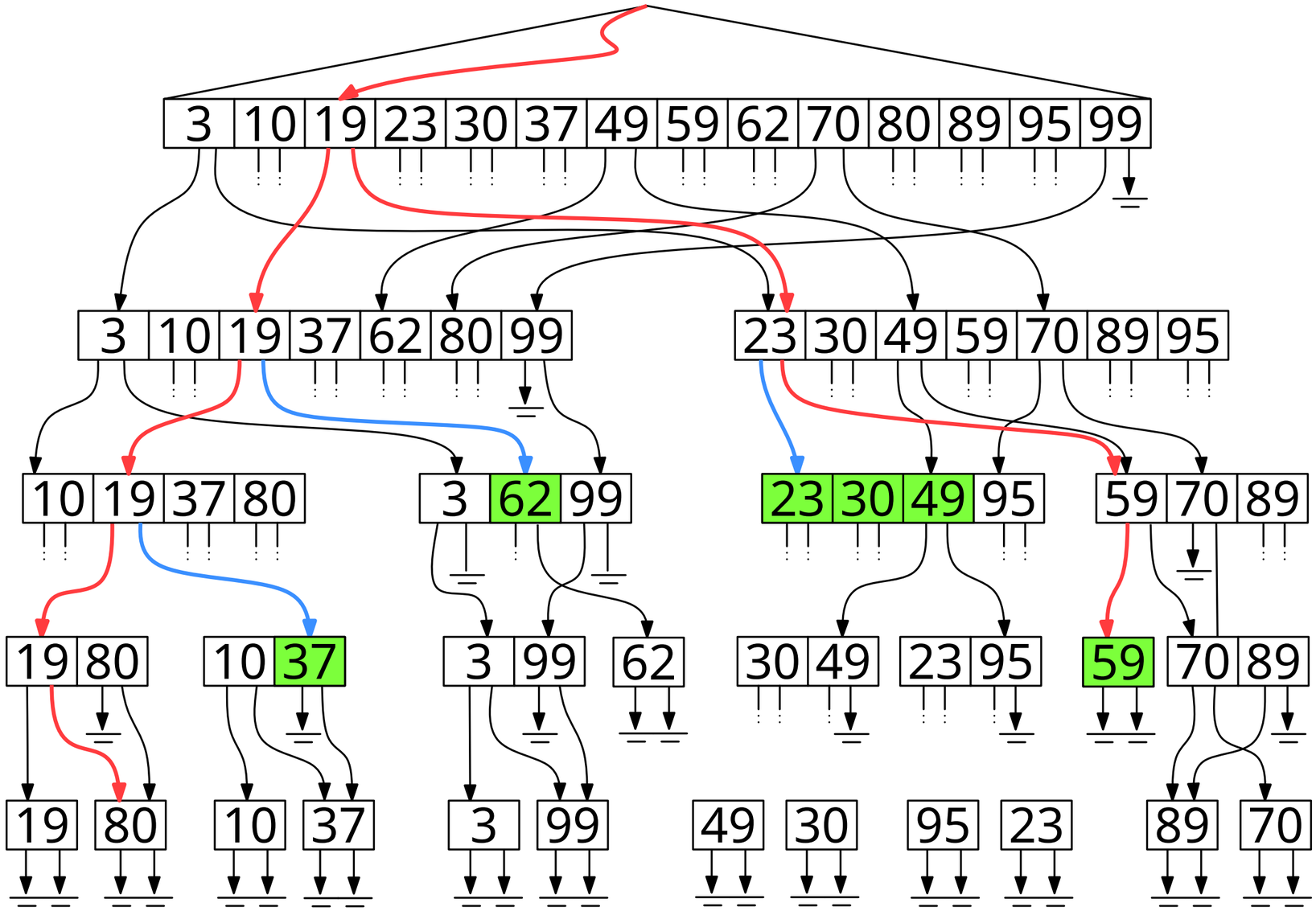
Fractional cascading



Fractional cascading



Fractional cascading



Fractional cascading

Instead of doing a 1D range query on the associated structure of some node v, we find the leaf where the search to y would end in $O(1)$ time via the direct pointer in the associated structure in the parent of v

The number of grey nodes reduces to $O(\log n)$

Result

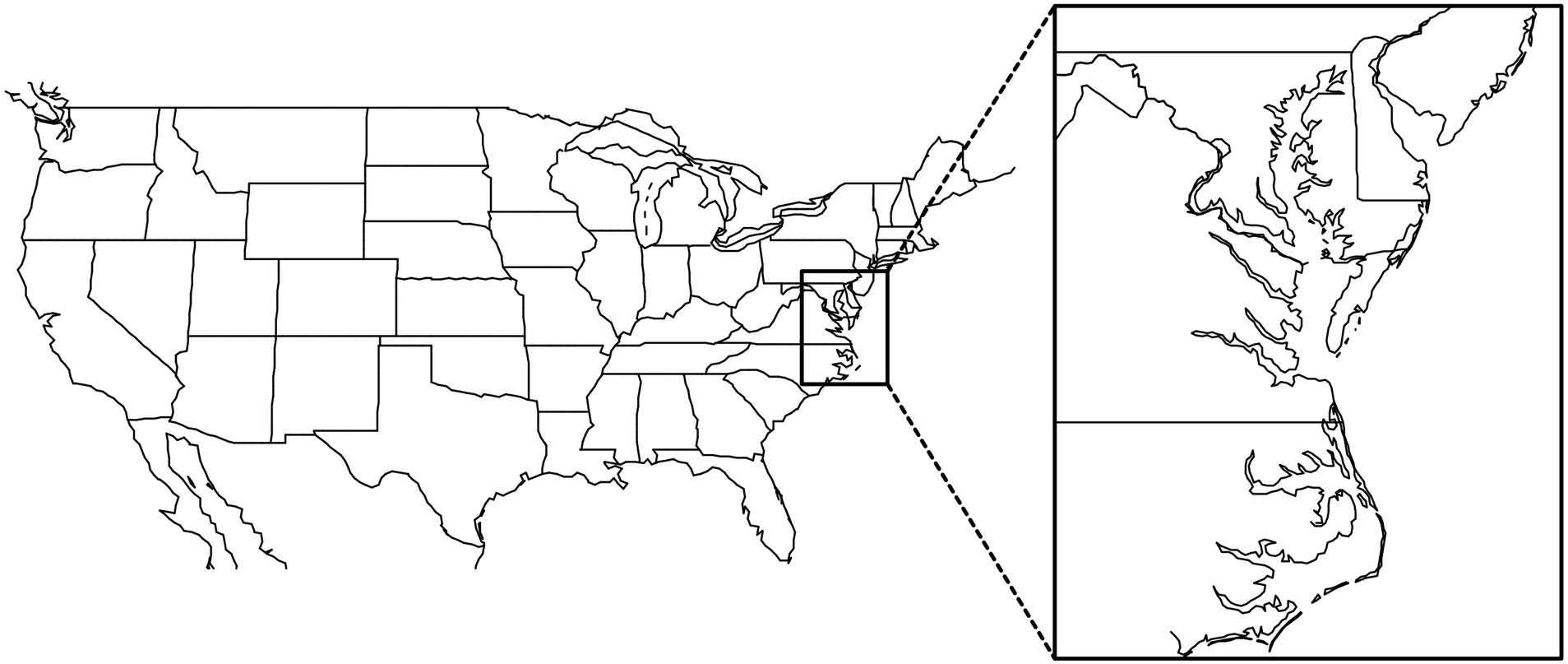
Theorem: A set of n points in d-dimensional space with $d \geq 2$ can be preprocessed in $O(n \log^{d-1} n)$ time into a data structure of $O(n \log^{d-1} n)$ size so that any d-dimensional range query can be answered in $O(\log^{d-1} n + k)$ time, where k is the number of answers reported.

Multiple points with the same x- or y-coordinate need to be handled with care.

Range and Windowing Queries

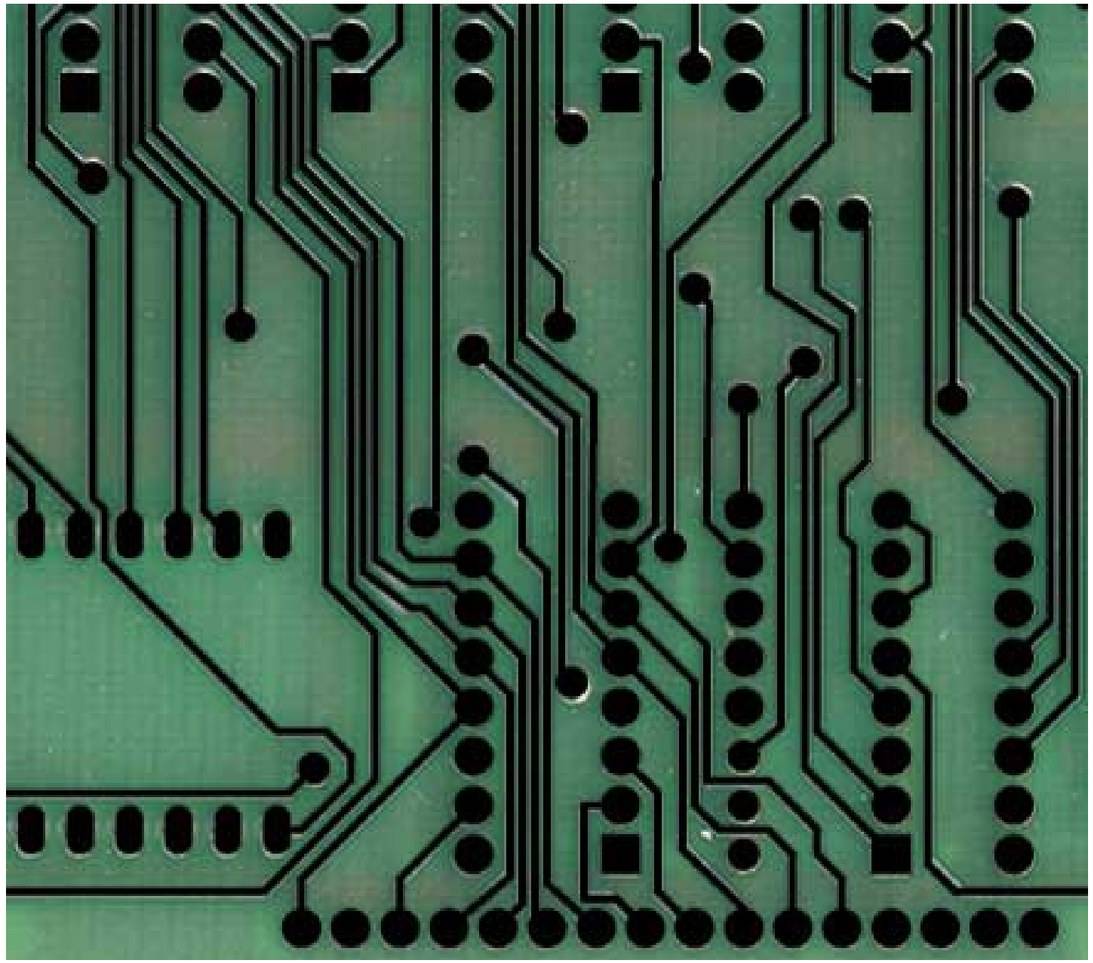
Interval trees for windowing queries

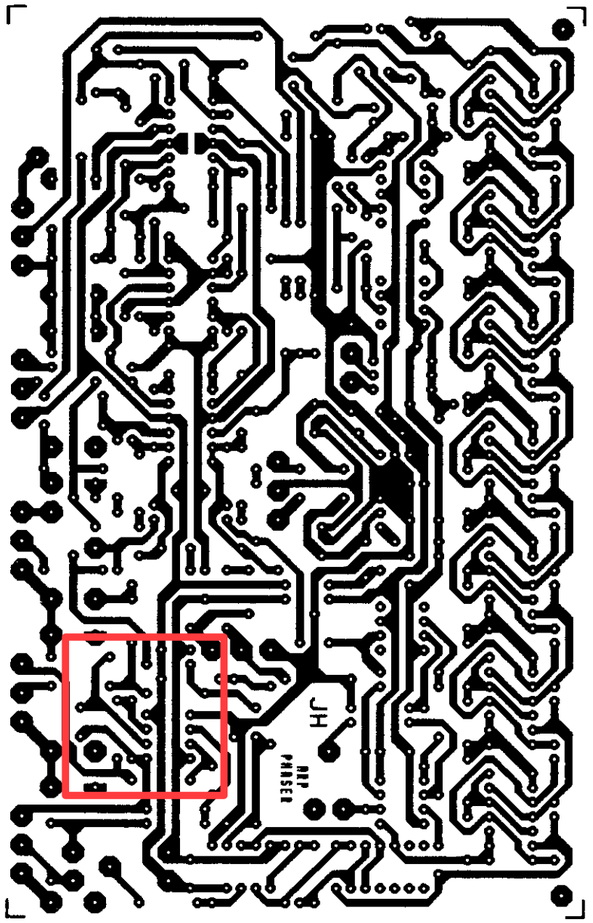
Windowing



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

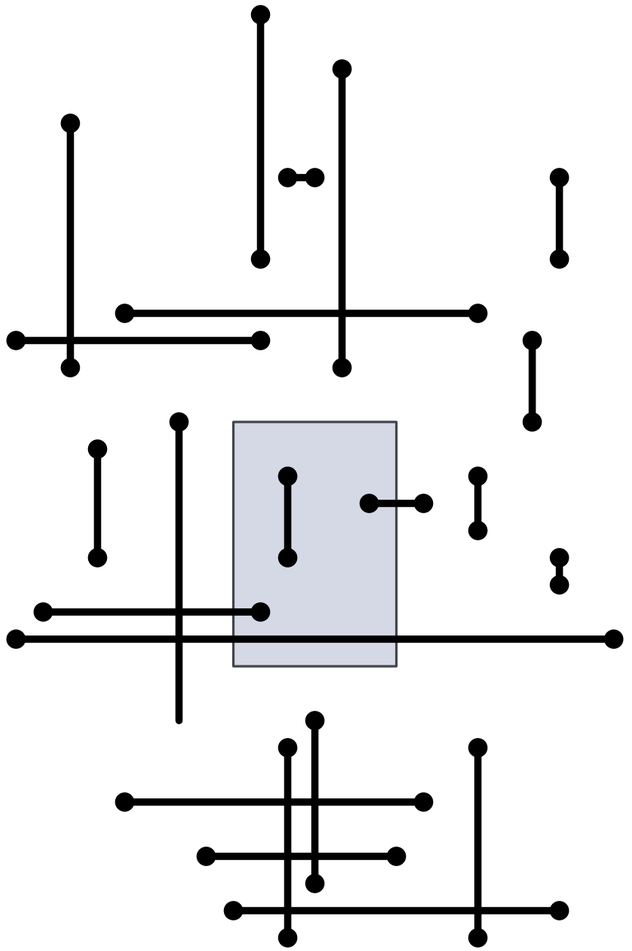
Windowing





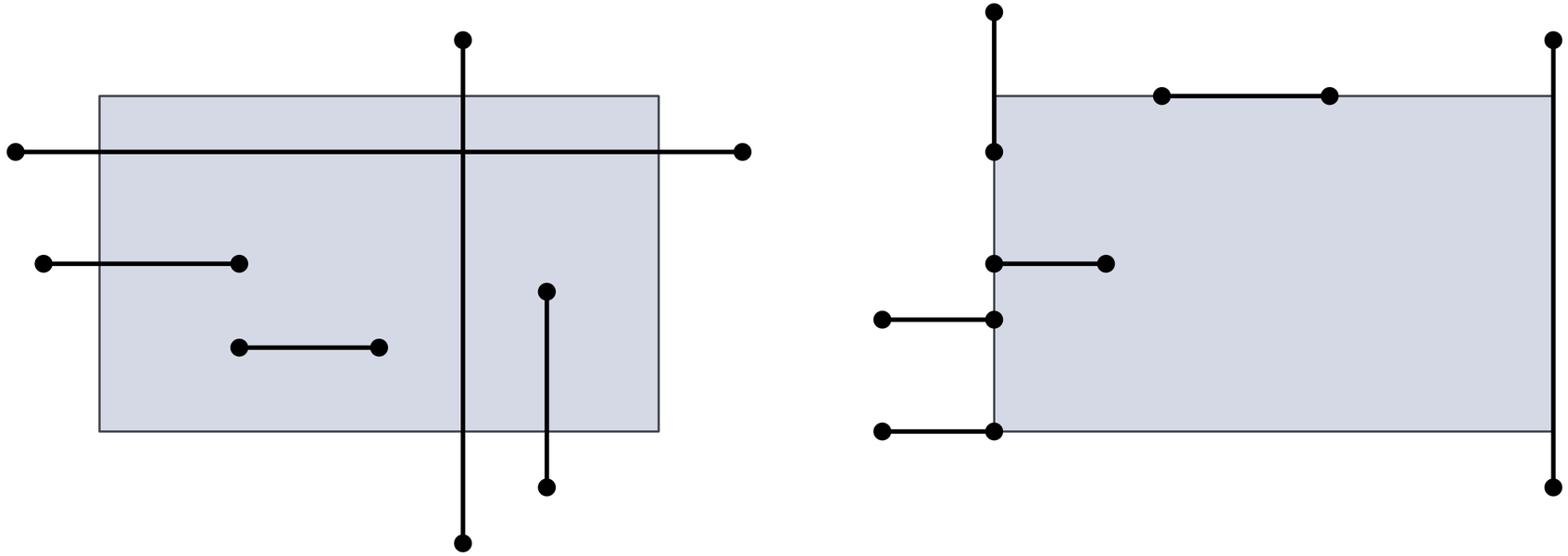
Windowing

Given a set of n axis-parallel line segments, preprocess them into a data structure so that the ones that intersect a query rectangle can be reported efficiently



Windowing

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

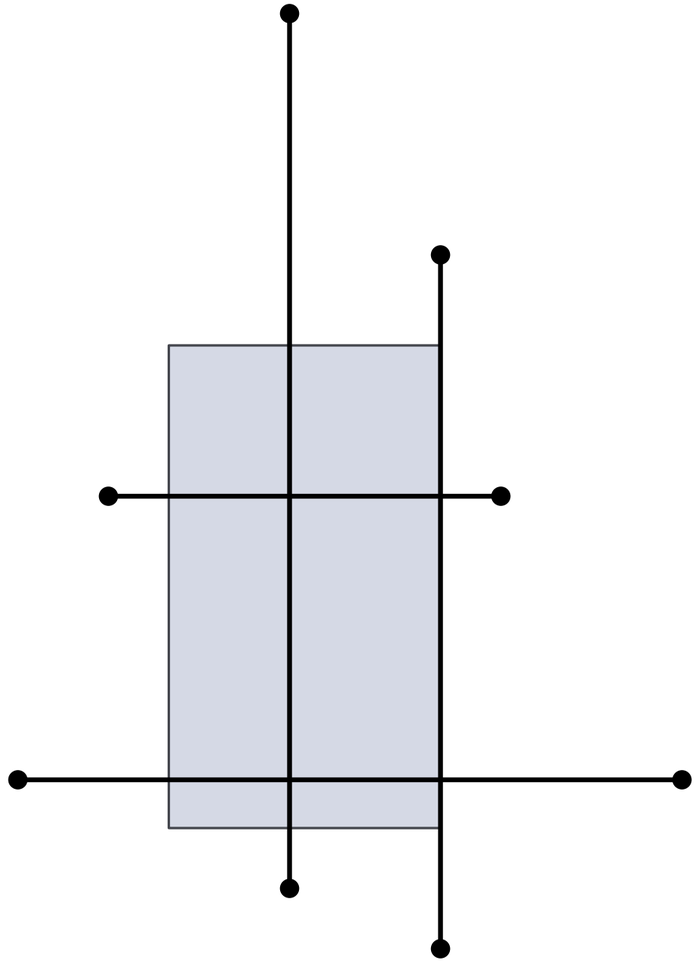


Windowing

Essentially two types:

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

Segments of the latter type always intersect the boundary of the rectangle twice (or contain a boundary edge)

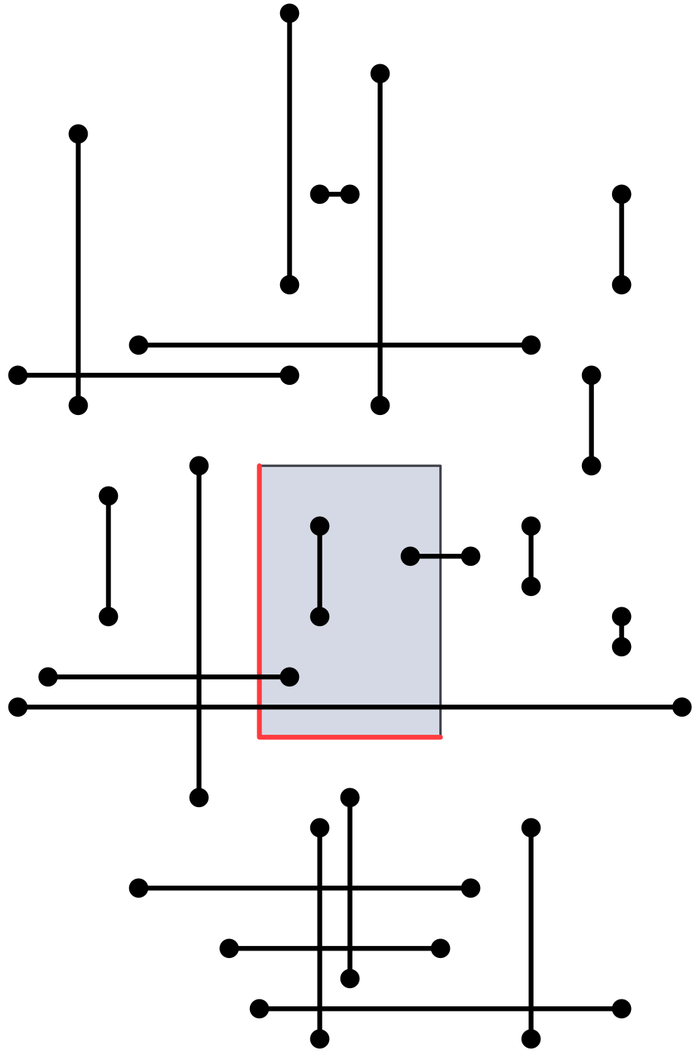


Windowing

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

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

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

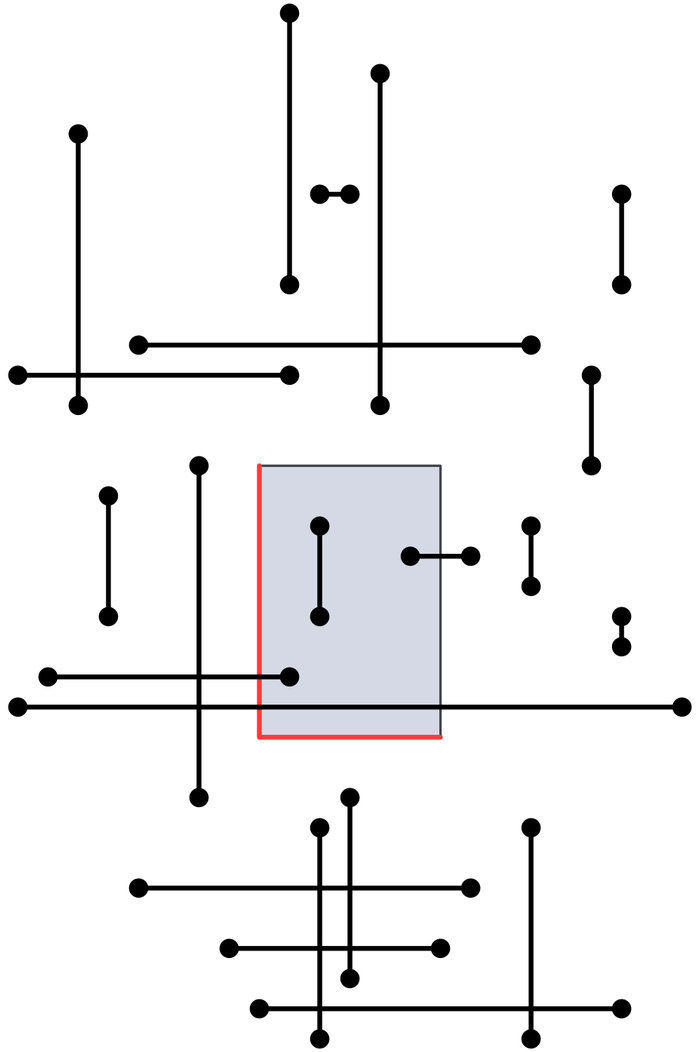


Windowing

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

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

Question: How often might we report the same segment?



Windowing

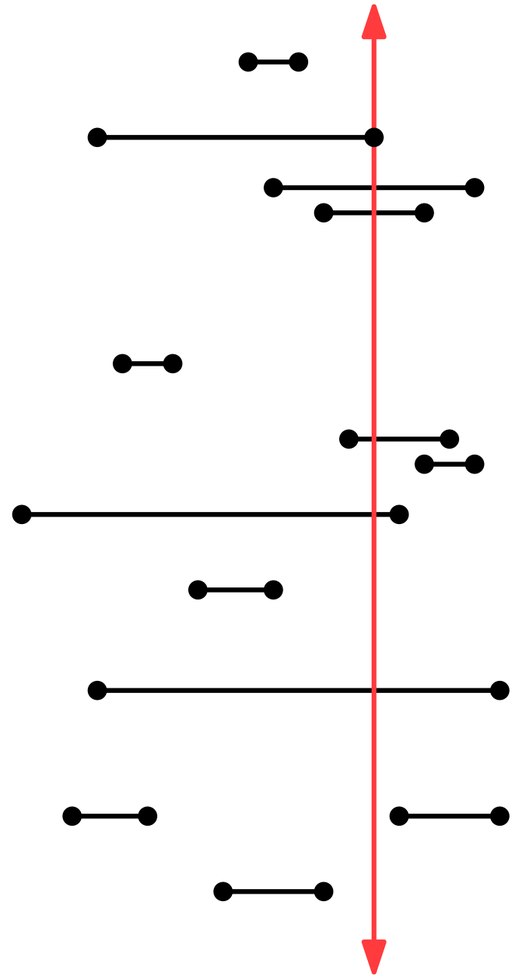
Instead of storing axis-parallel segments and searching with a rectangle, we will:

- store the segment endpoints and query with the rectangle
  use 2D range tree
- store the segments and query with the left side and the bottom side of the rectangle
  need to develop data structure

Windowing

Current problem of our interest:

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

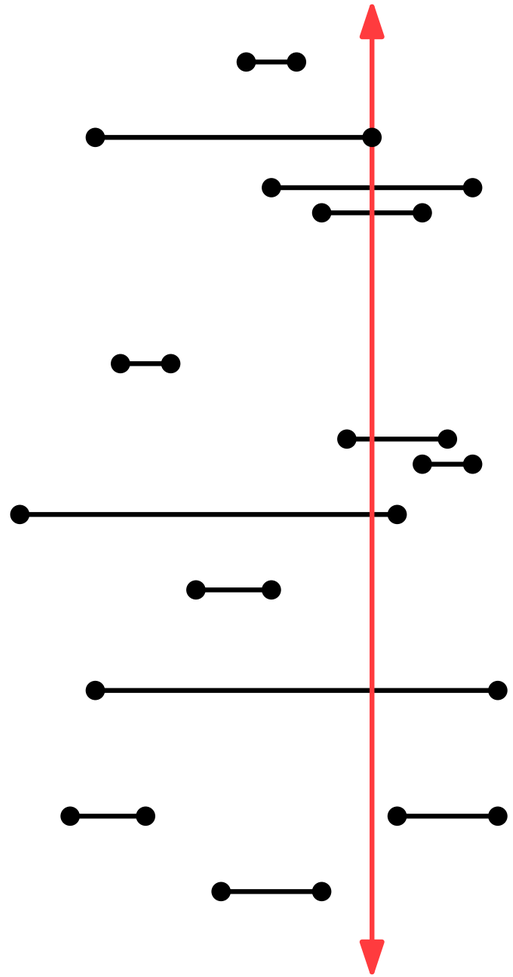


Windowing

Simpler query problem:

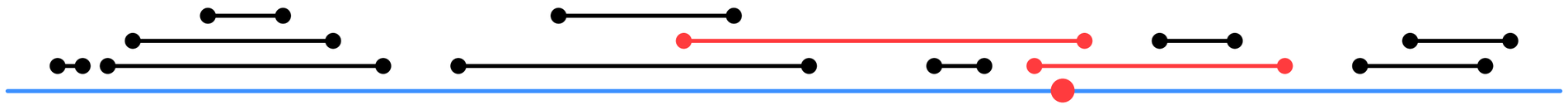
What if the vertical query segment is a full line?

Then the problem is essentially 1-dimensional



Interval querying

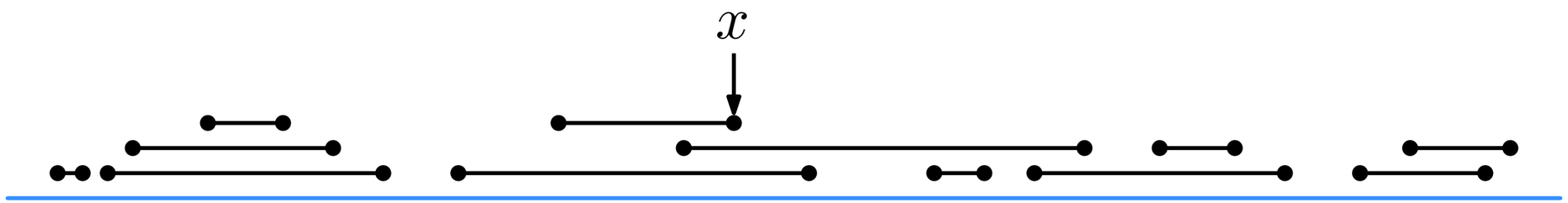
Given a set I of n intervals on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



Splitting a set of intervals

The median x of the $2n$ endpoints partitions the intervals into three subsets:

- Intervals I_{left} fully left of x
- Intervals I_{mid} that contain (intersect) x
- Intervals I_{right} fully right of x



Interval tree: recursive definition

The interval tree for I has a root node v that contains x and

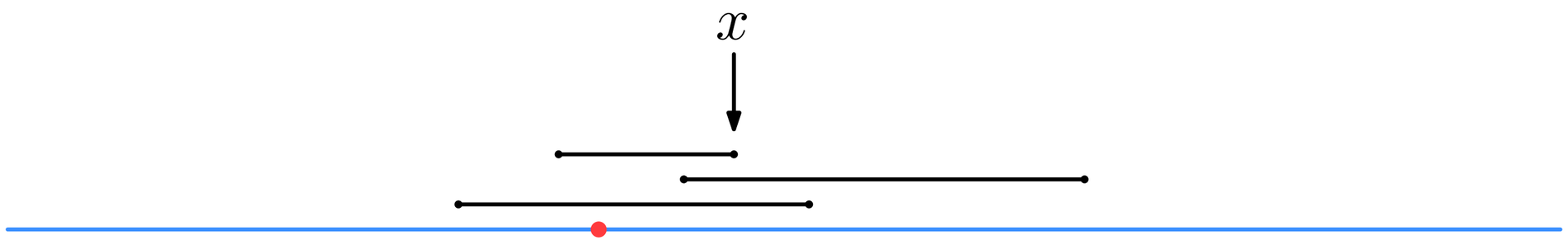
- the intervals I_{left} are stored in the left subtree of v,
- the intervals I_{mid} are stored with v,
- the intervals I_{right} are stored in the right subtree of v.

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

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

Interval tree: left and right lists

How is I_{mid} stored?

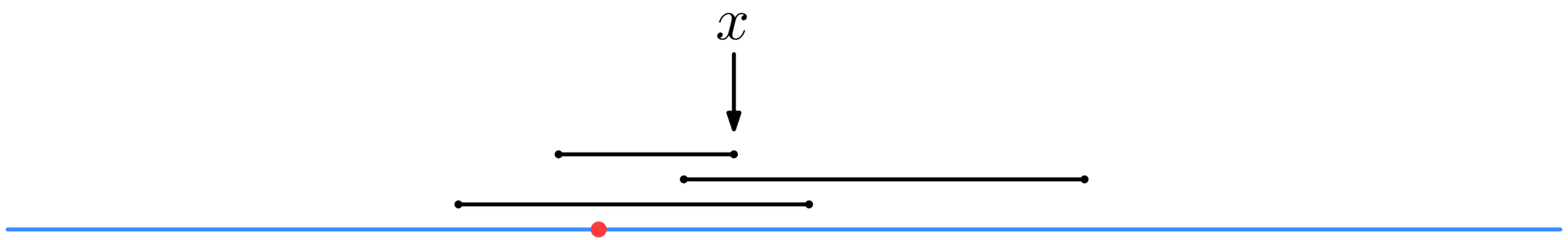


Observe: If the query point is left of x, then only the left endpoint determines if an interval is an answer

Symmetrically: If the query point is right of x, then only the right endpoint determines if an interval is an answer

Interval tree: left and right lists

How is I_{mid} stored?

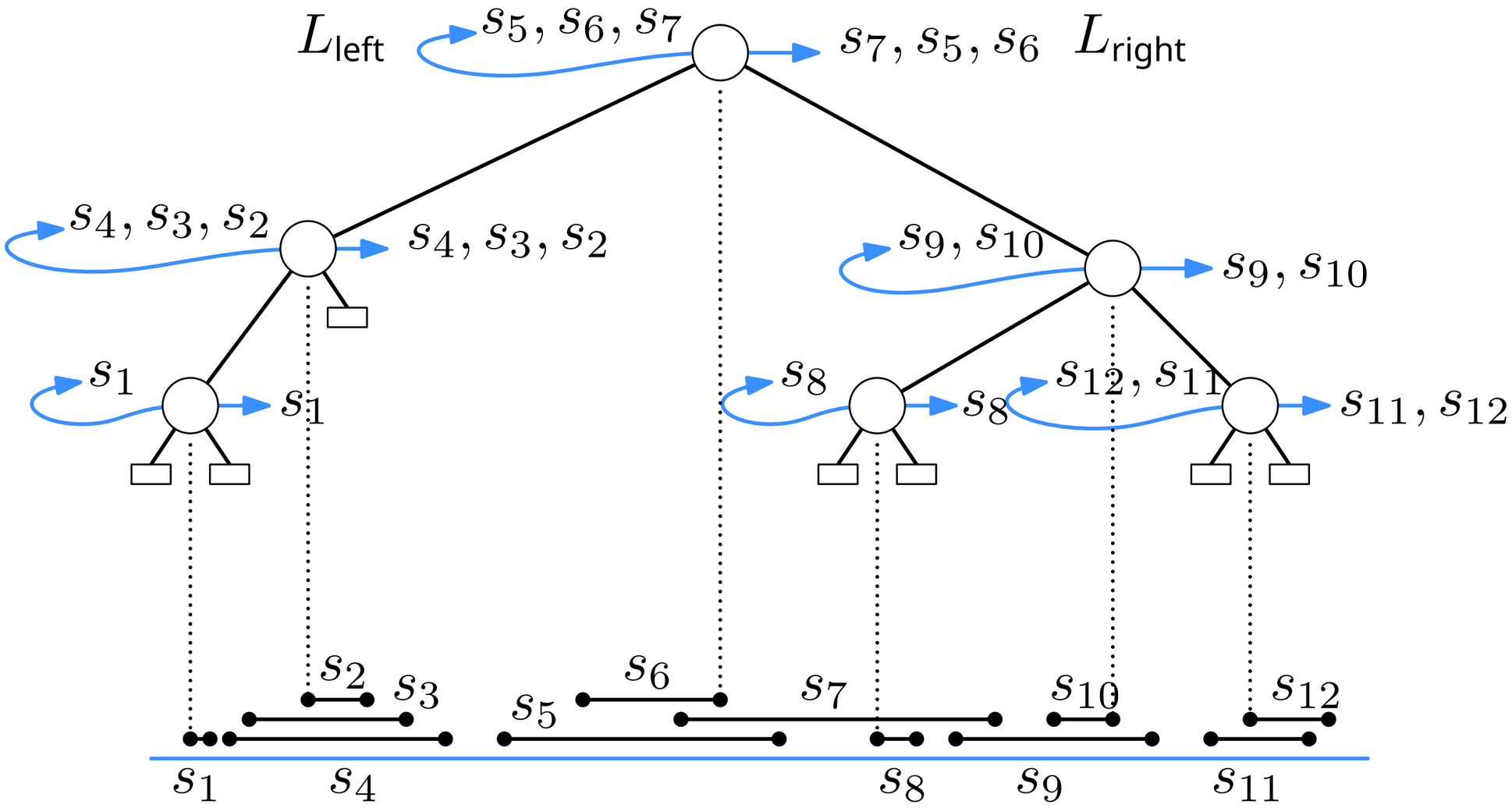


Make a list L_{left} using the left-to-right order of the left endpoints of I_{mid}

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

Store both lists as associated structures with v

Interval tree: example



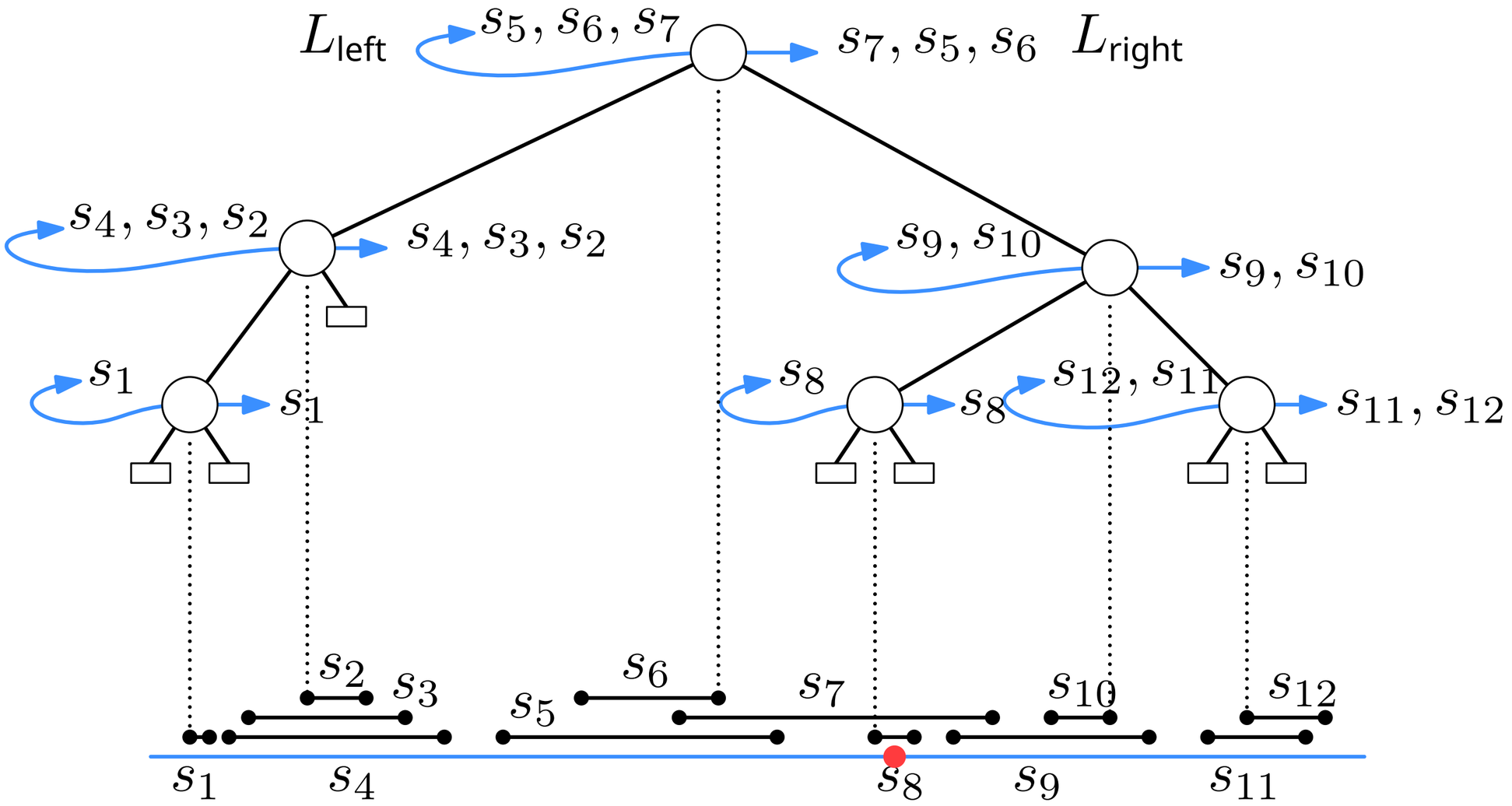
Interval tree: storage

The main tree has $O(n)$ nodes

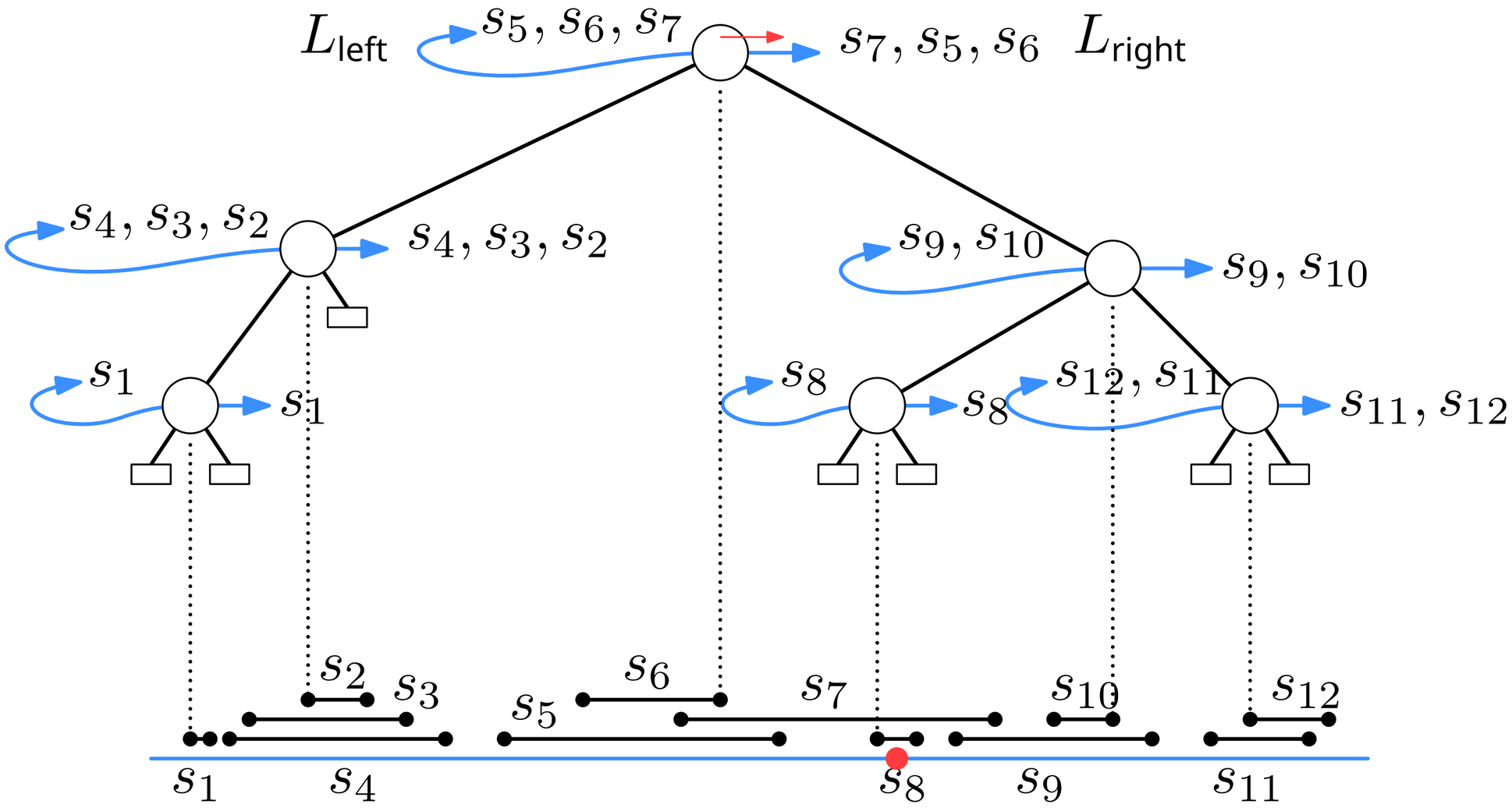
The total length of all lists is $2n$ because each interval is stored exactly twice: in L_{left} and L_{right} and only at one node

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

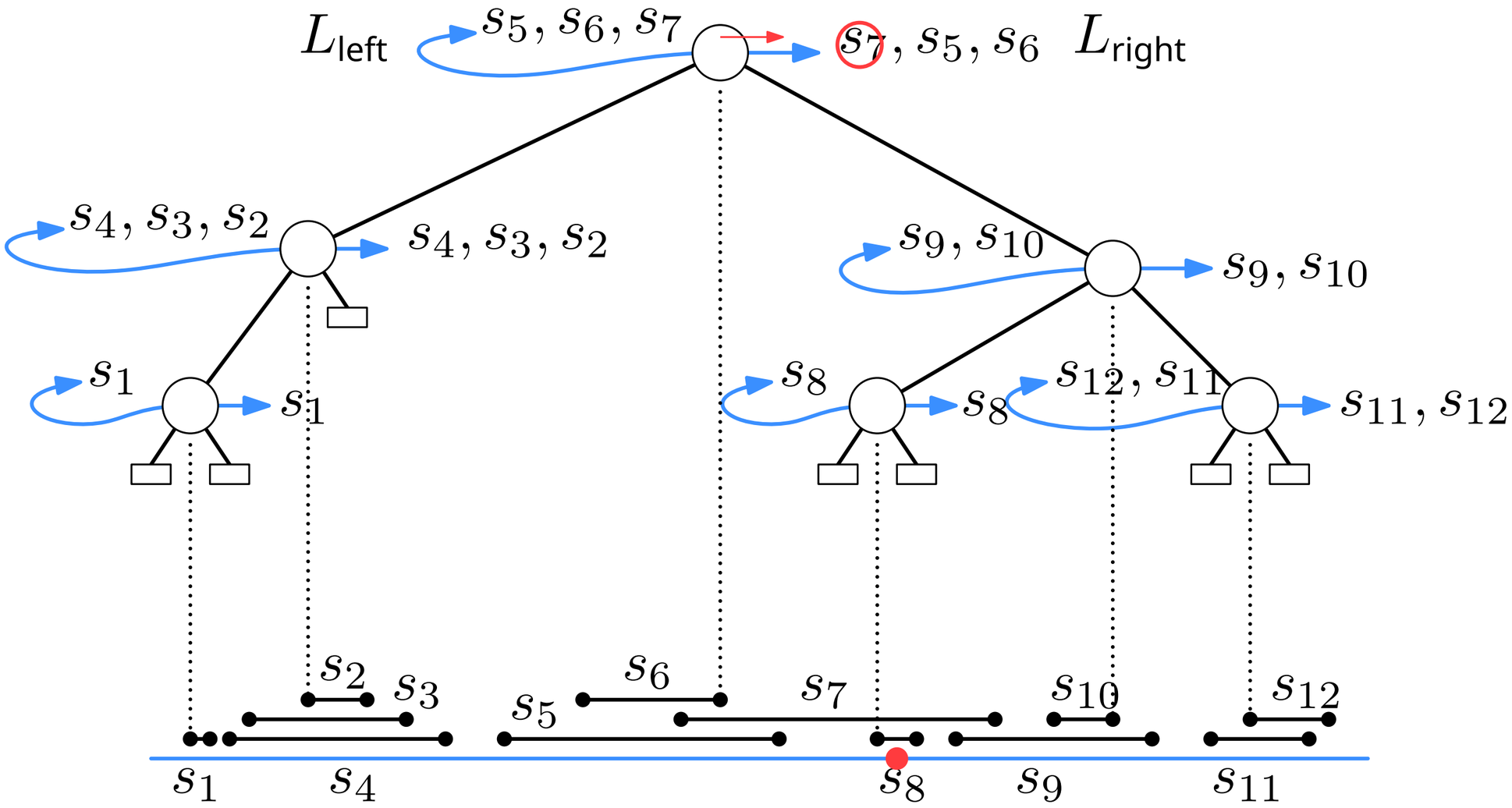
Interval tree: query example



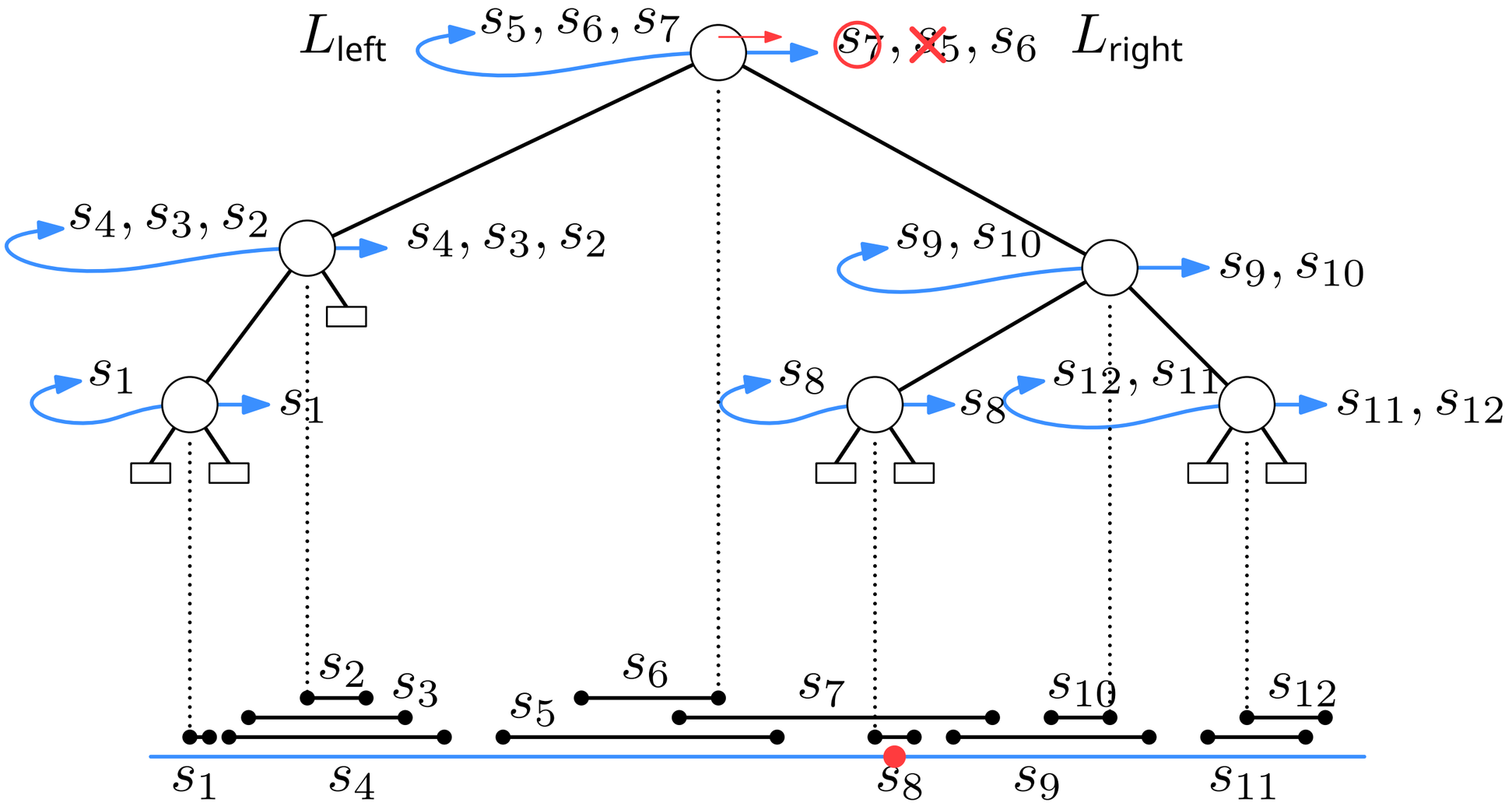
Interval tree: query example



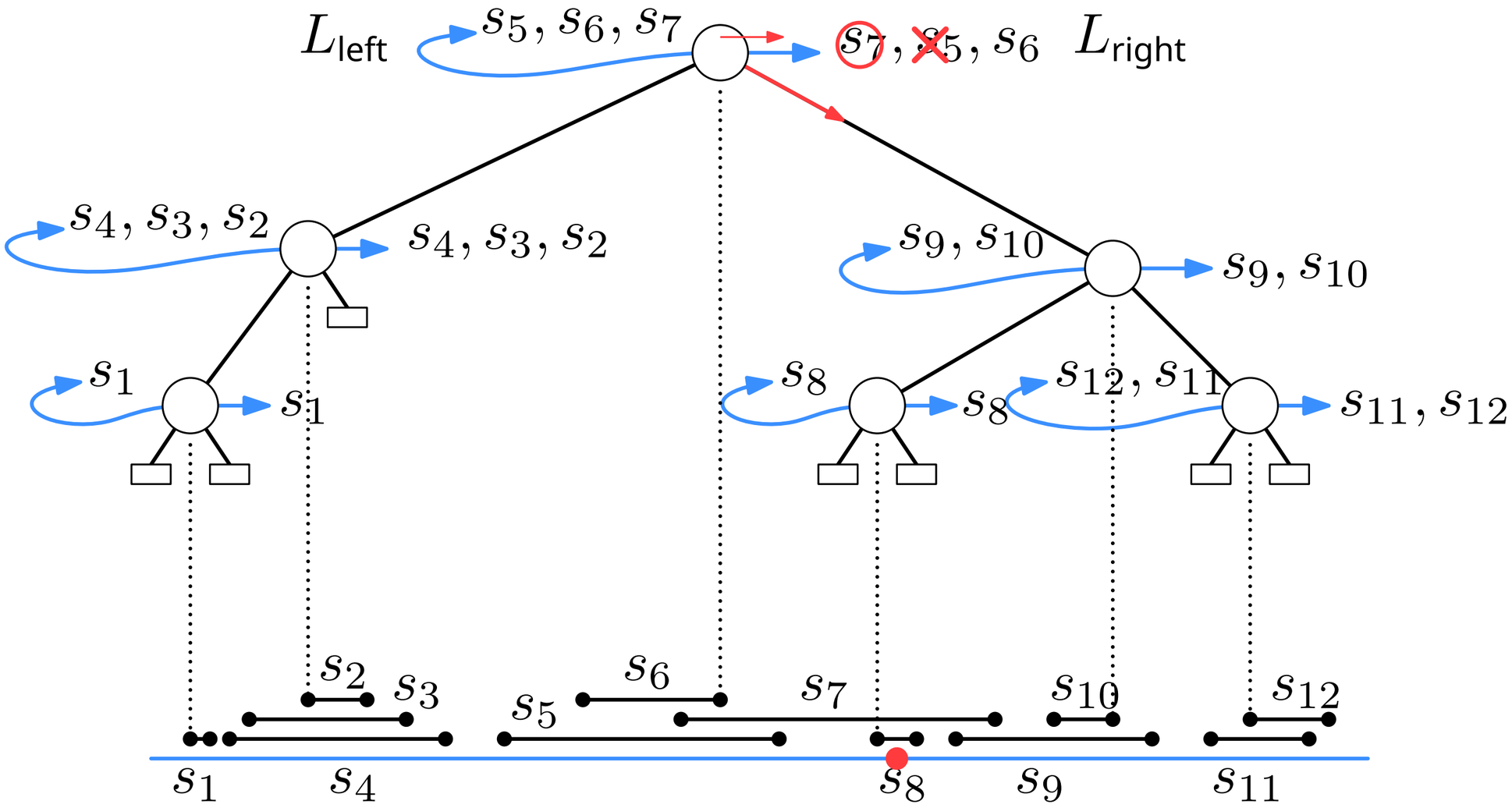
Interval tree: query example



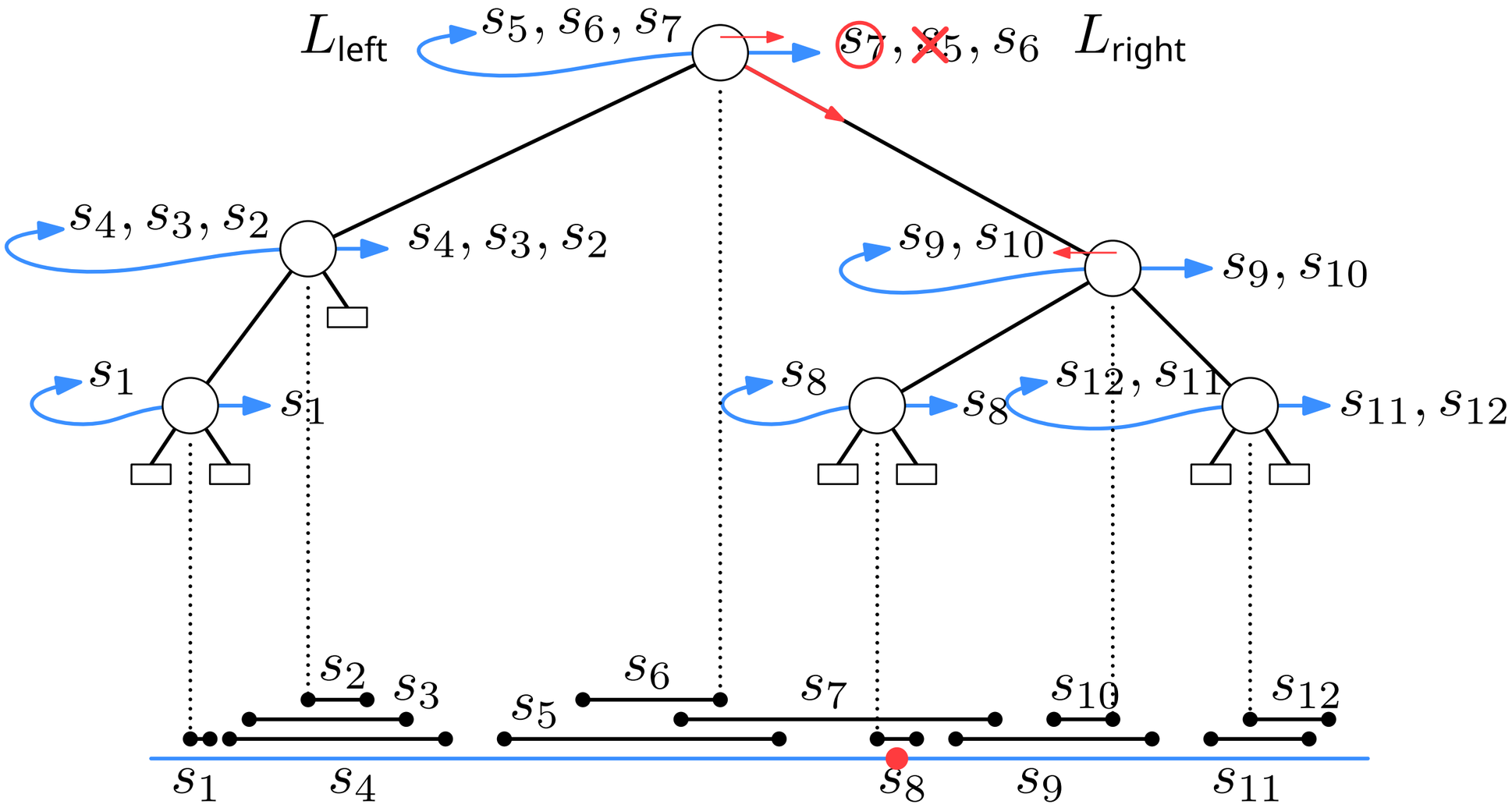
Interval tree: query example



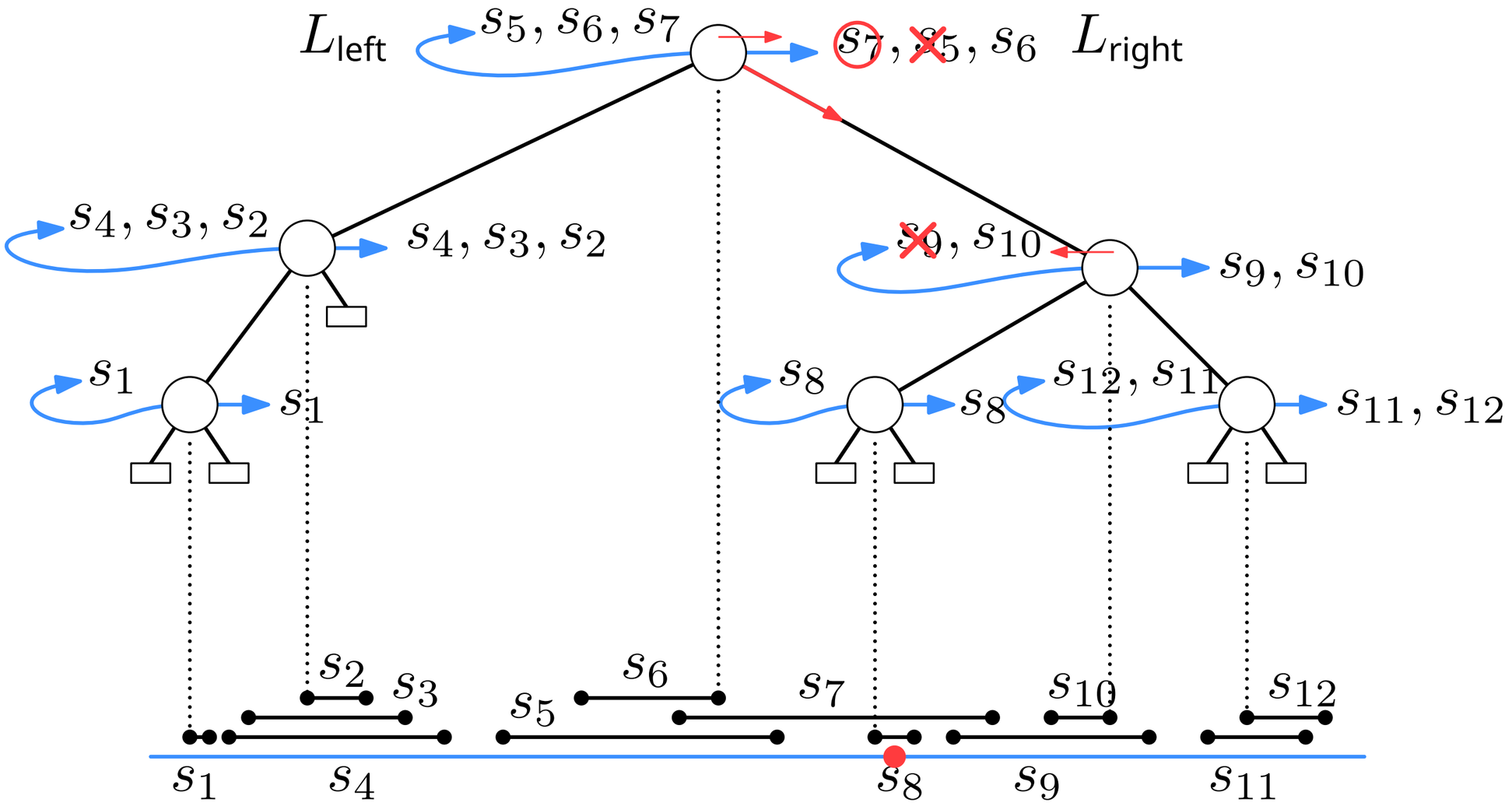
Interval tree: query example



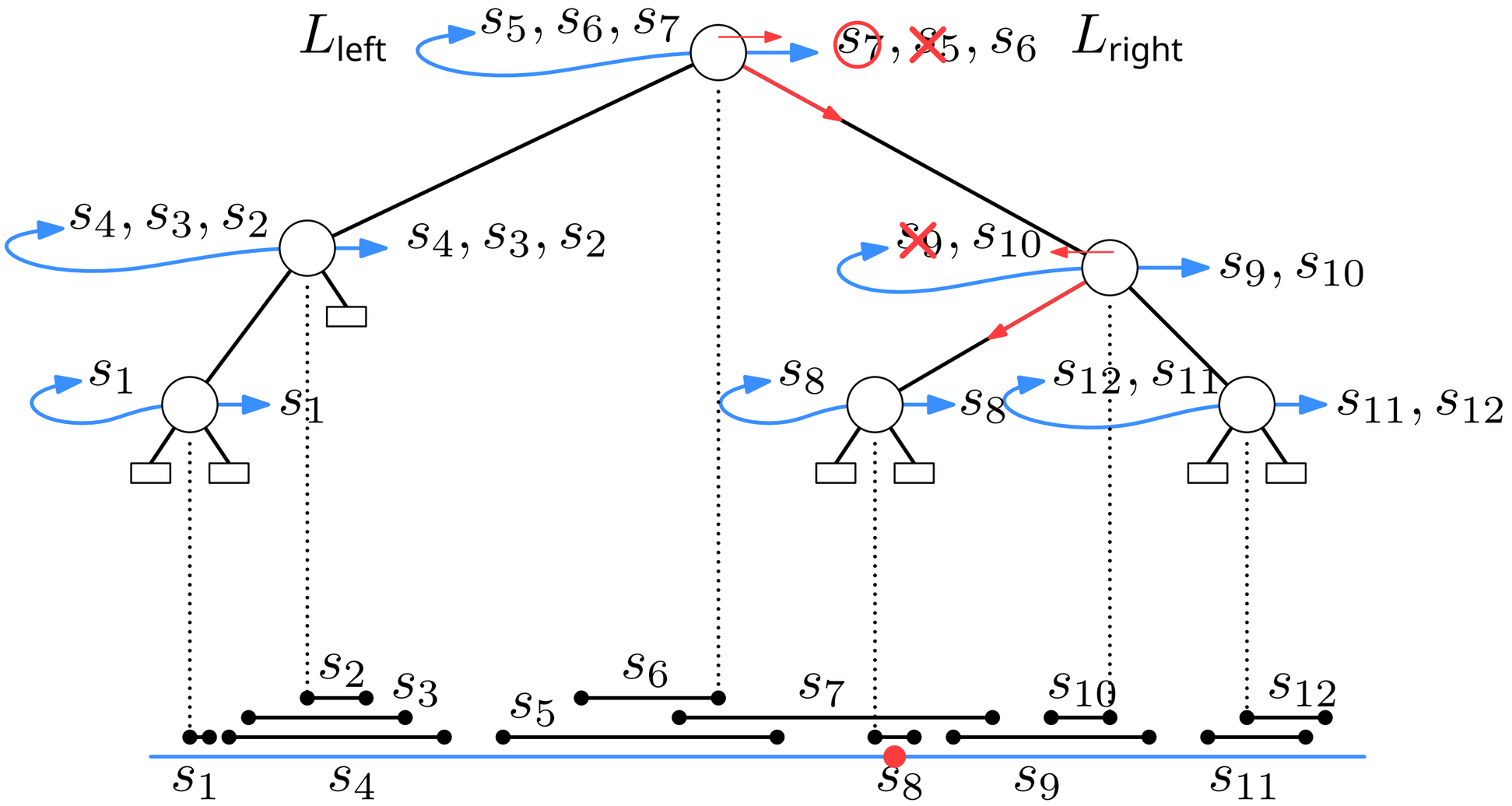
Interval tree: query example



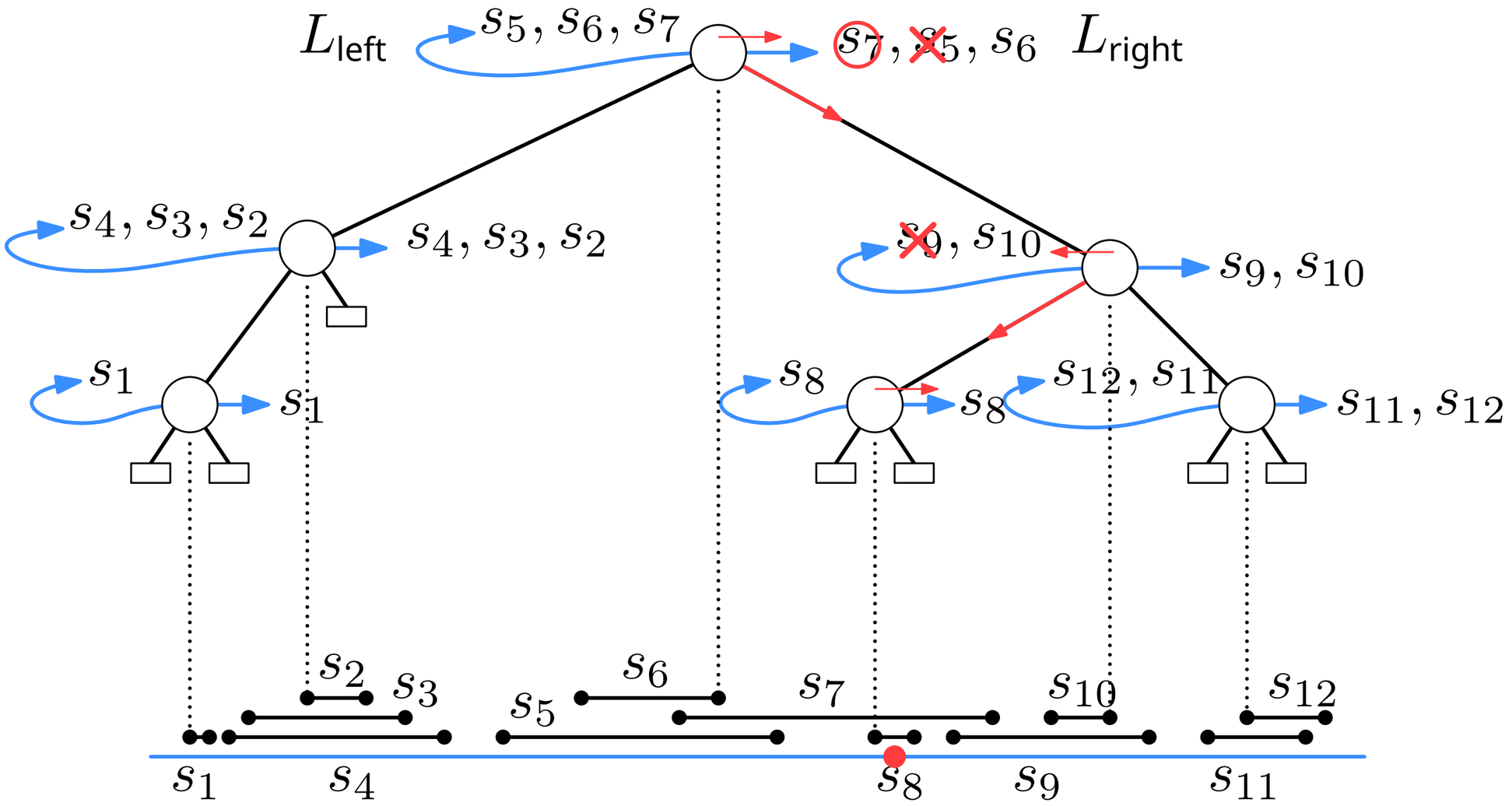
Interval tree: query example



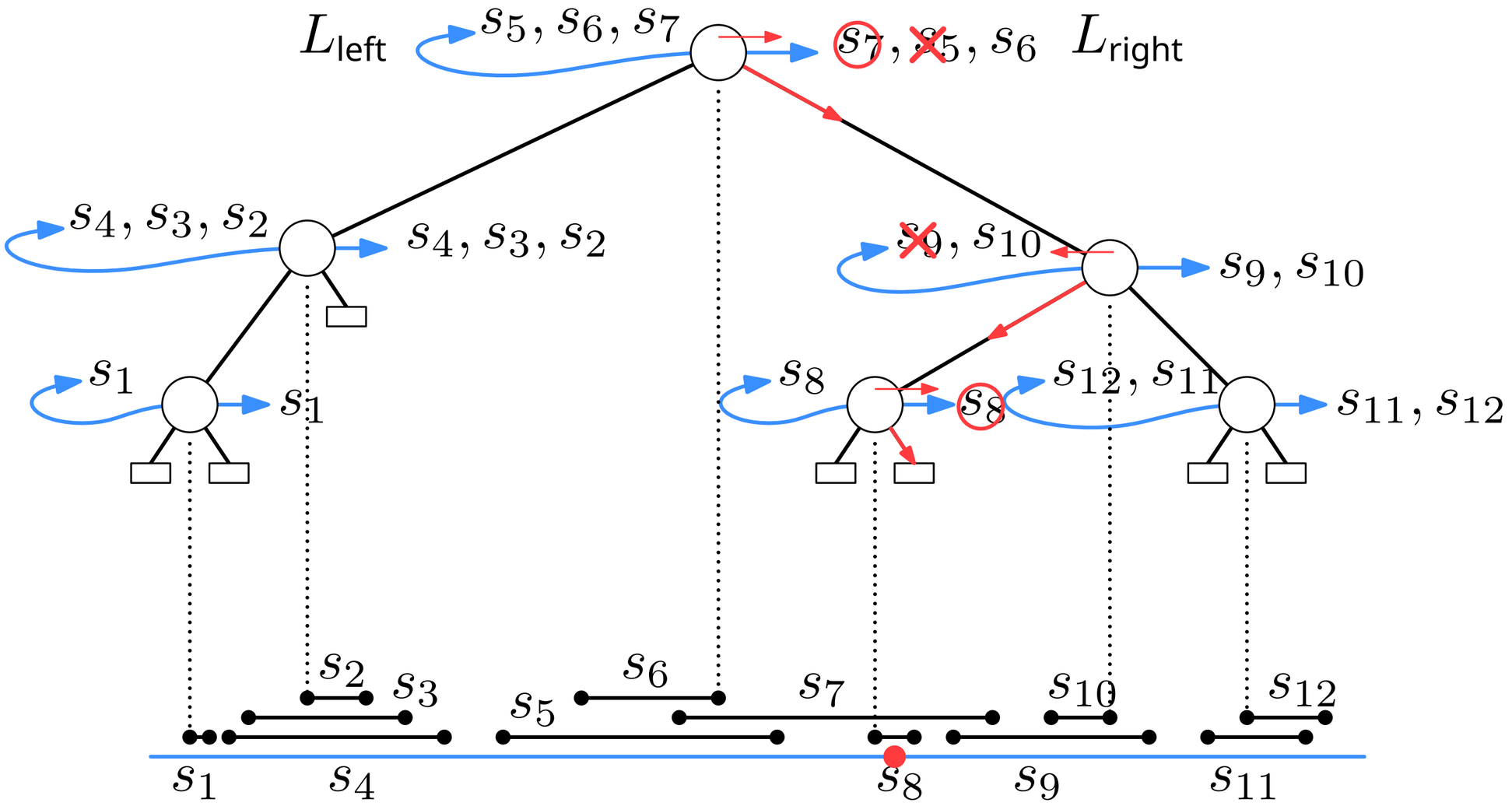
Interval tree: query example



Interval tree: query example



Interval tree: query example



Interval querying

Algorithm QUERYINTERVALTREE(v, q_x)

1: **if** v is not a leaf **then**
2: **if** $q_x < x_{\text{mid}}(v)$ **then**
3: Traverse list $L_{\text{left}}(v)$, starting at the interval with the leftmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
4: QUERYINTERVALTREE$(lc(v), q_x)$
5: **else**
6: Traverse list $L_{\text{right}}(v)$, starting at the interval with the rightmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
7: QUERYINTERVALTREE$(rc(v), q_x)$

Quiz: Interval trees queries

objects: intervals s_i

query object: 1d point q_x

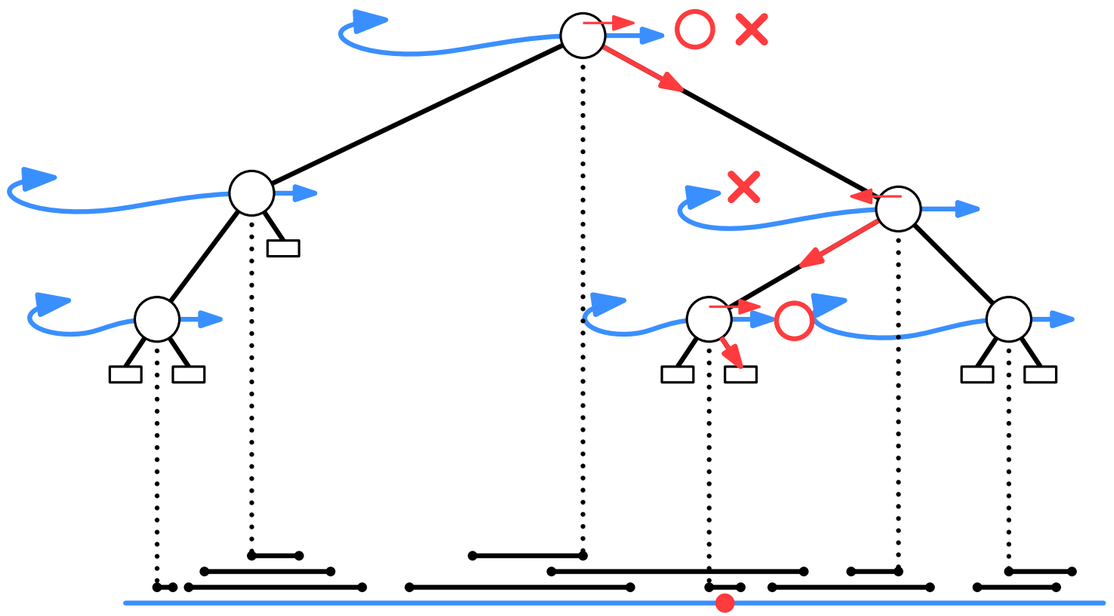
query: report all s_i with $q_x \in s_i$

Question: What is the query time when using an interval tree?

A: $\Theta(\log n + k)$

B: $\Theta(\log^2 n + k)$

C: $\Theta(\log^3 n + k)$



Quiz: Interval trees queries

objects: intervals s_i

query object: 1d point q_x

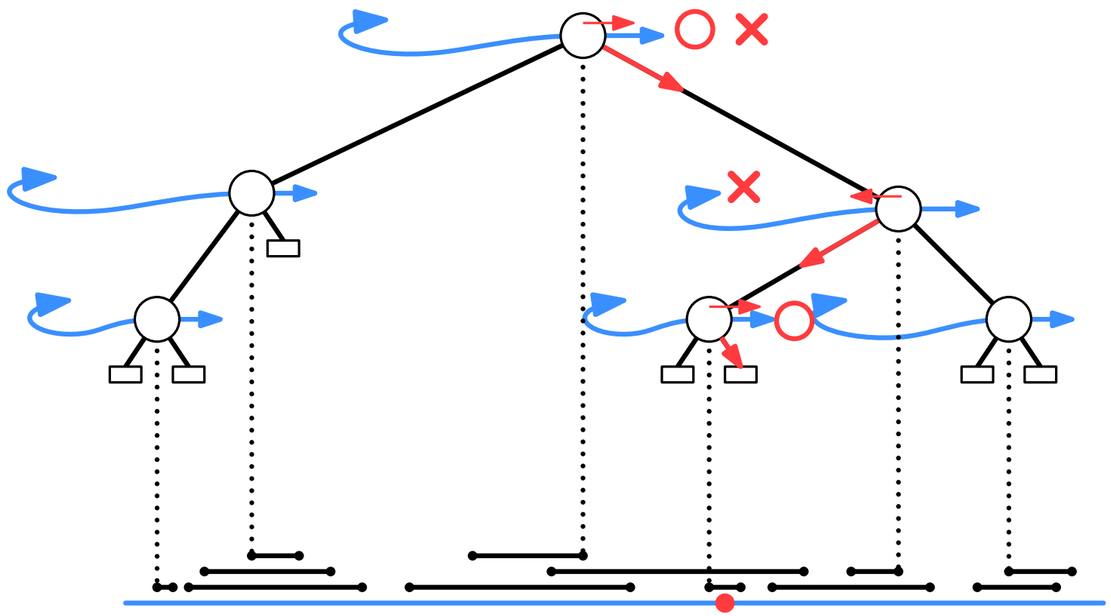
query: report all s_i with $q_x \in s_i$

Question: What is the query time when using an interval tree?

A: $\Theta(\log n + k)$

B: $\Theta(\log^2 n + k)$

C: $\Theta(\log^3 n + k)$



Interval tree: query time

The query follows only one path in the tree, and that path has length $O(\log n)$

The query traverses $O(\log n)$ lists. Traversing a list with k' answers takes $O(1 + k')$ time

The total time for list traversal is therefore $O(\log n + k)$, with the total number of answers reported (no answer is found more than once)

The query time is $O(\log n) + O(\log n + k) = O(\log n + k)$

Interval tree: query example

Algorithm ConstructIntervalTree(I)

Input: A set I of intervals on the real line

Output: The root of an interval tree for I

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

Result

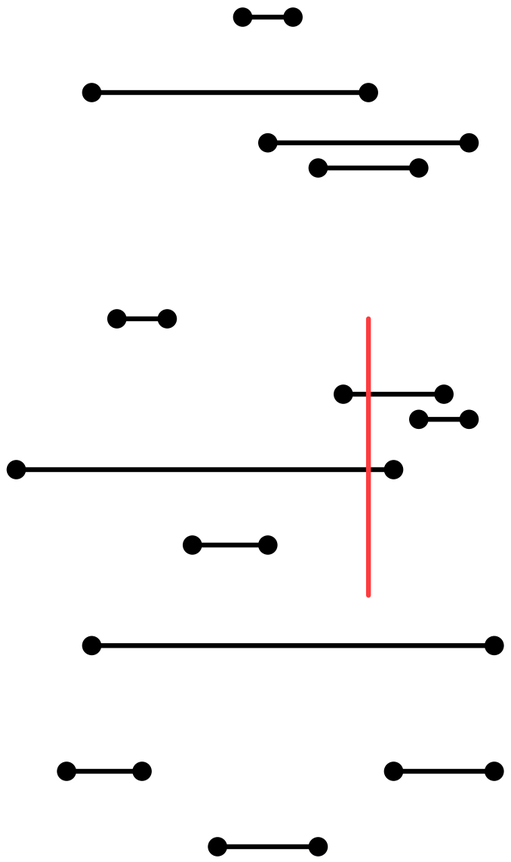
Theorem: An interval tree for a set I of n intervals uses $O(n)$ storage and can be built in $O(n \log n)$ time. All intervals that contain a query point can be reported in $O(\log n + k)$ time, where k is the number of reported intervals.

Range and Windowing Queries

Interval trees + range trees for windowing queries

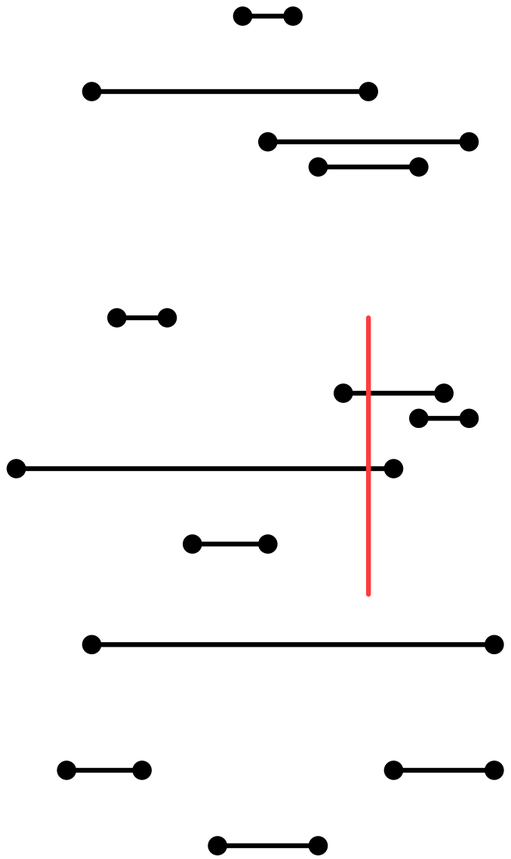
The problem

What we want:

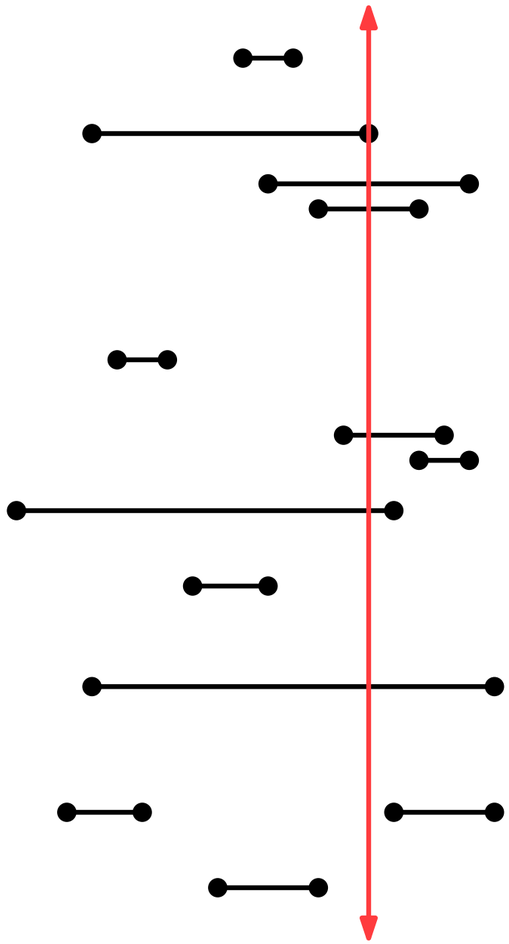


The problem

What we want:

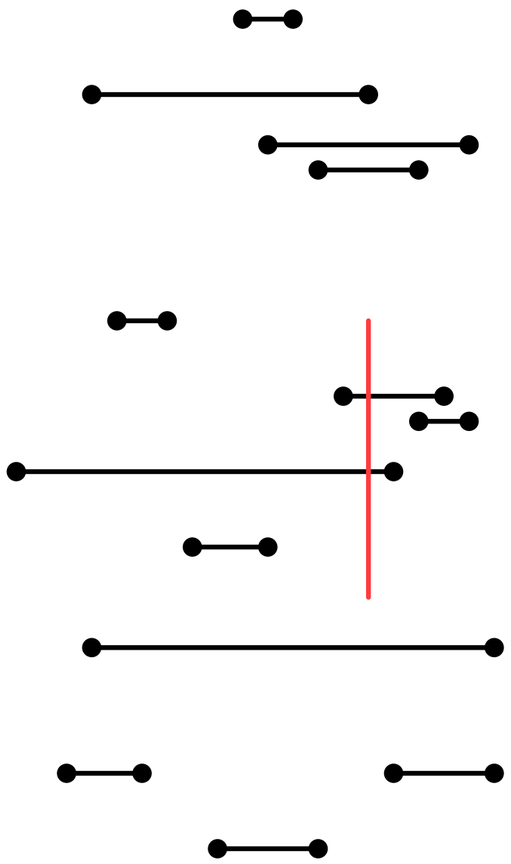


What interval trees give us:

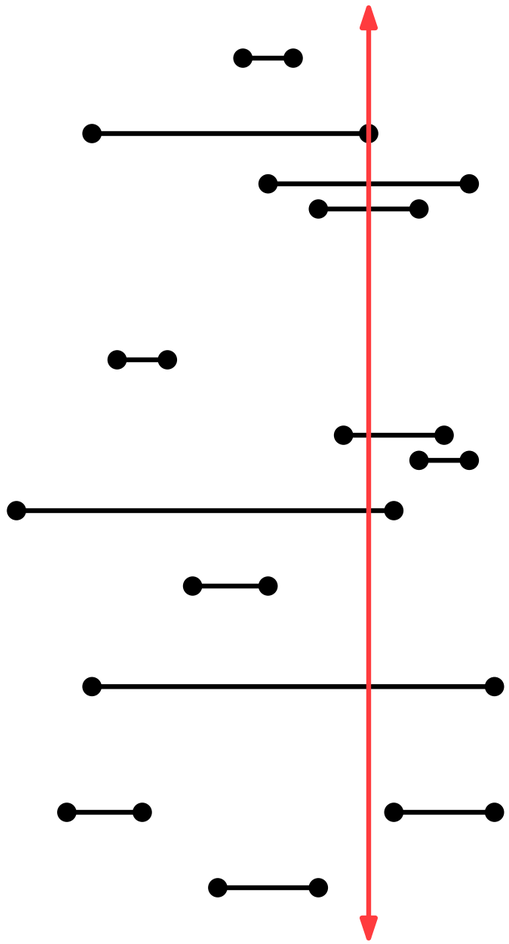


The problem

What we want:



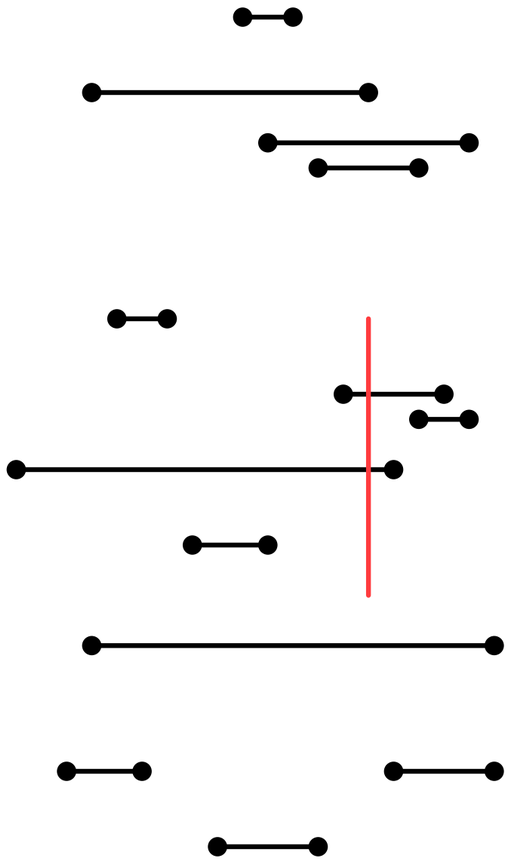
What interval trees give us:



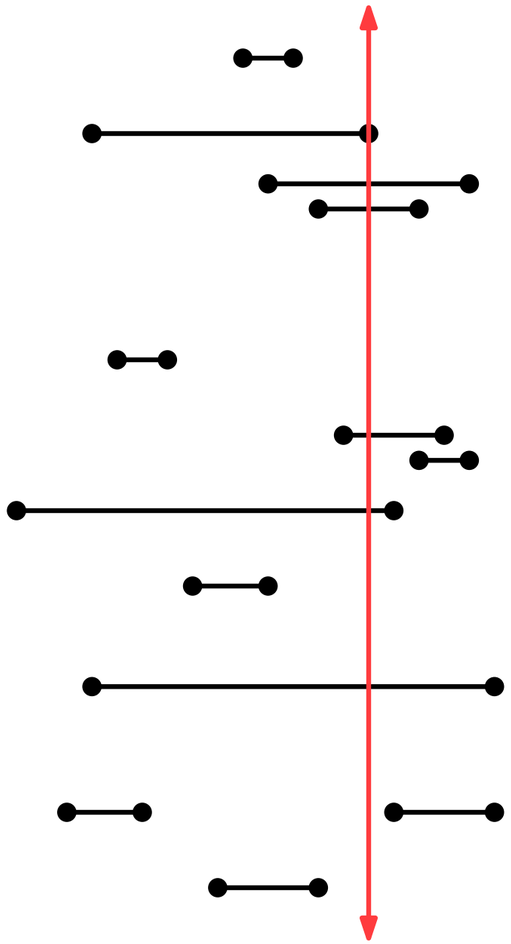
Suppose we use an interval tree on the x-intervals of the horizontal line segments, . . .

The problem

What we want:



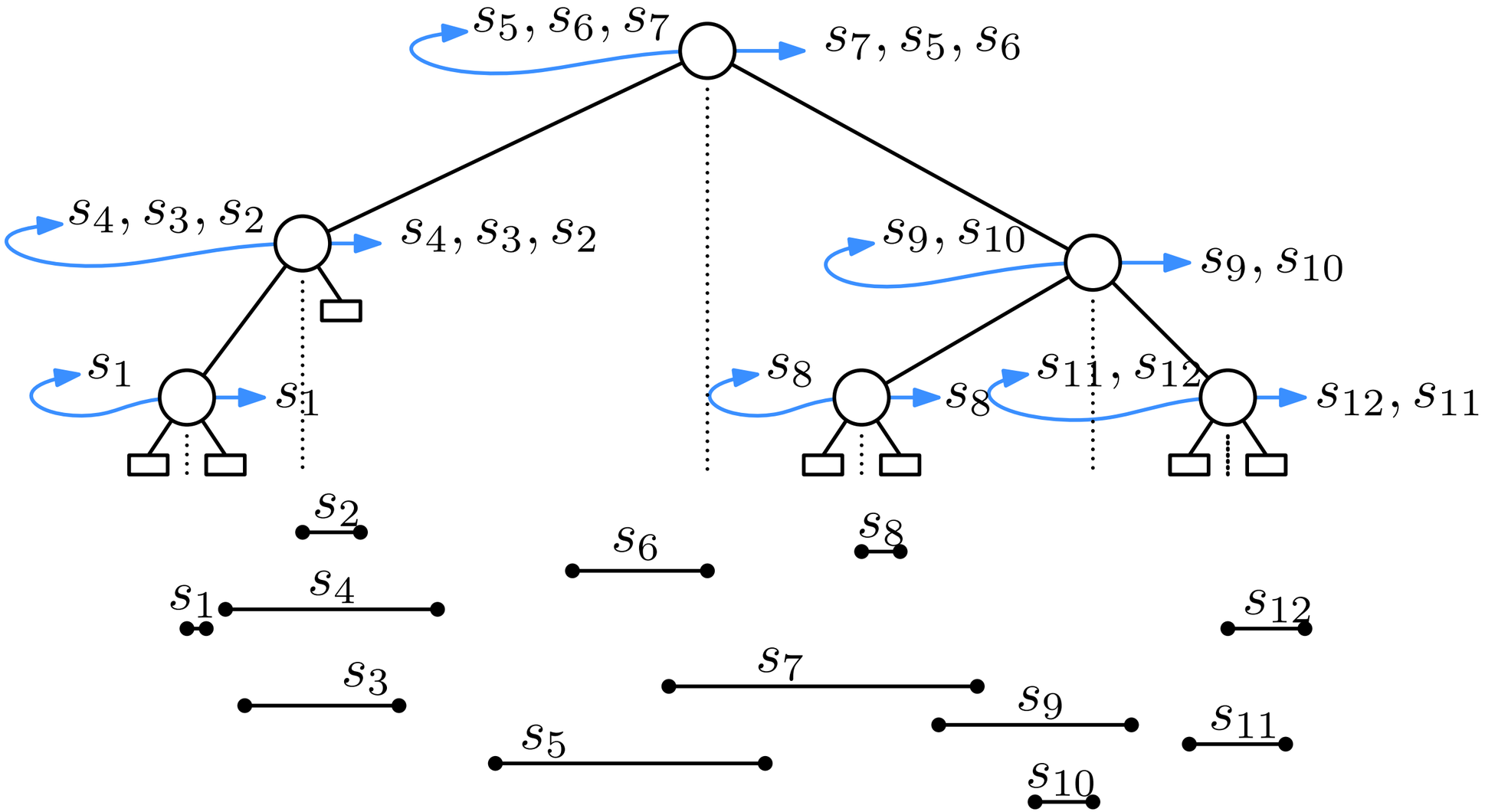
What interval trees give us:



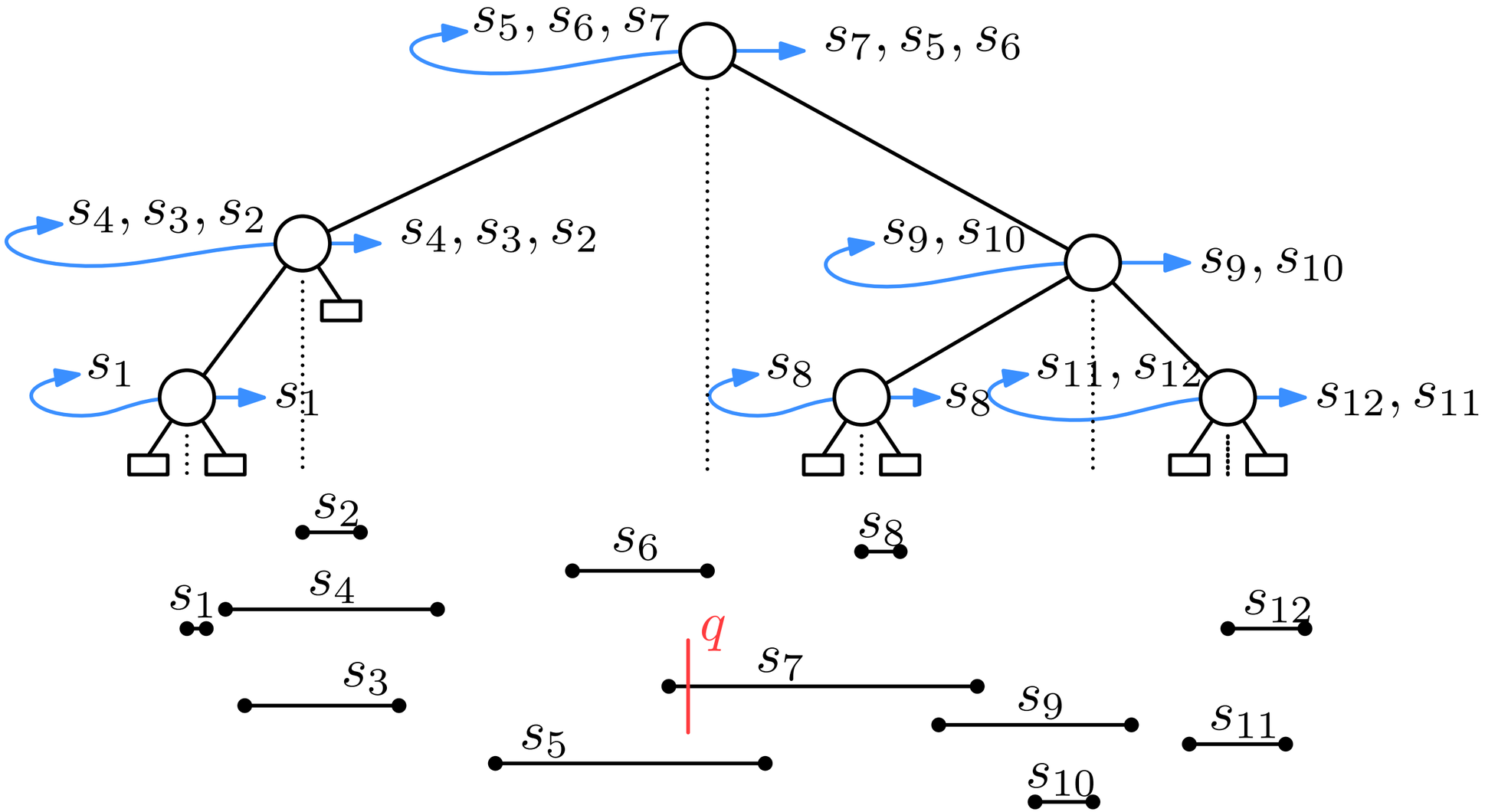
Suppose we use an interval tree on the x-intervals of the horizontal line segments, . . .

What do we need to change/extend/combine?

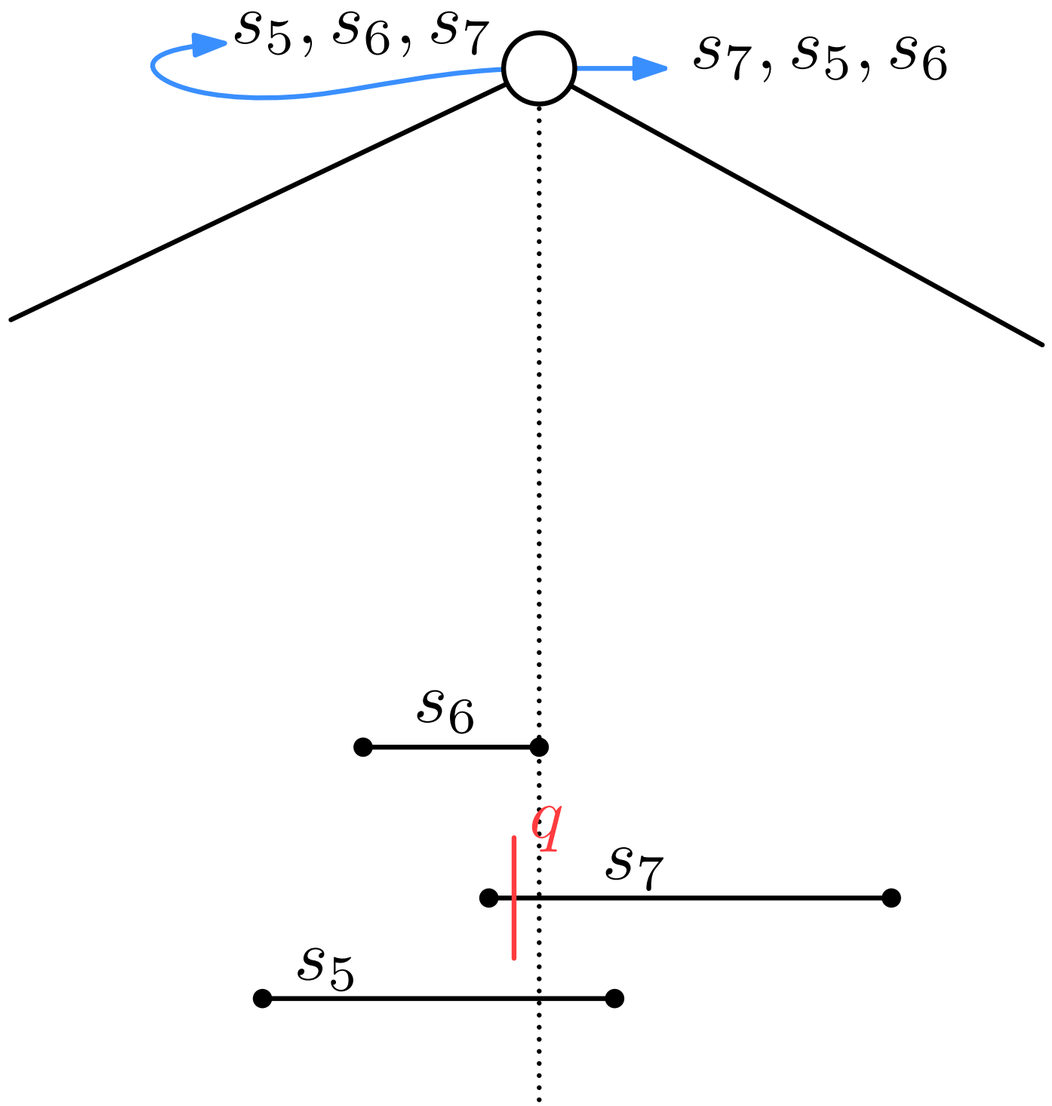
The query problem



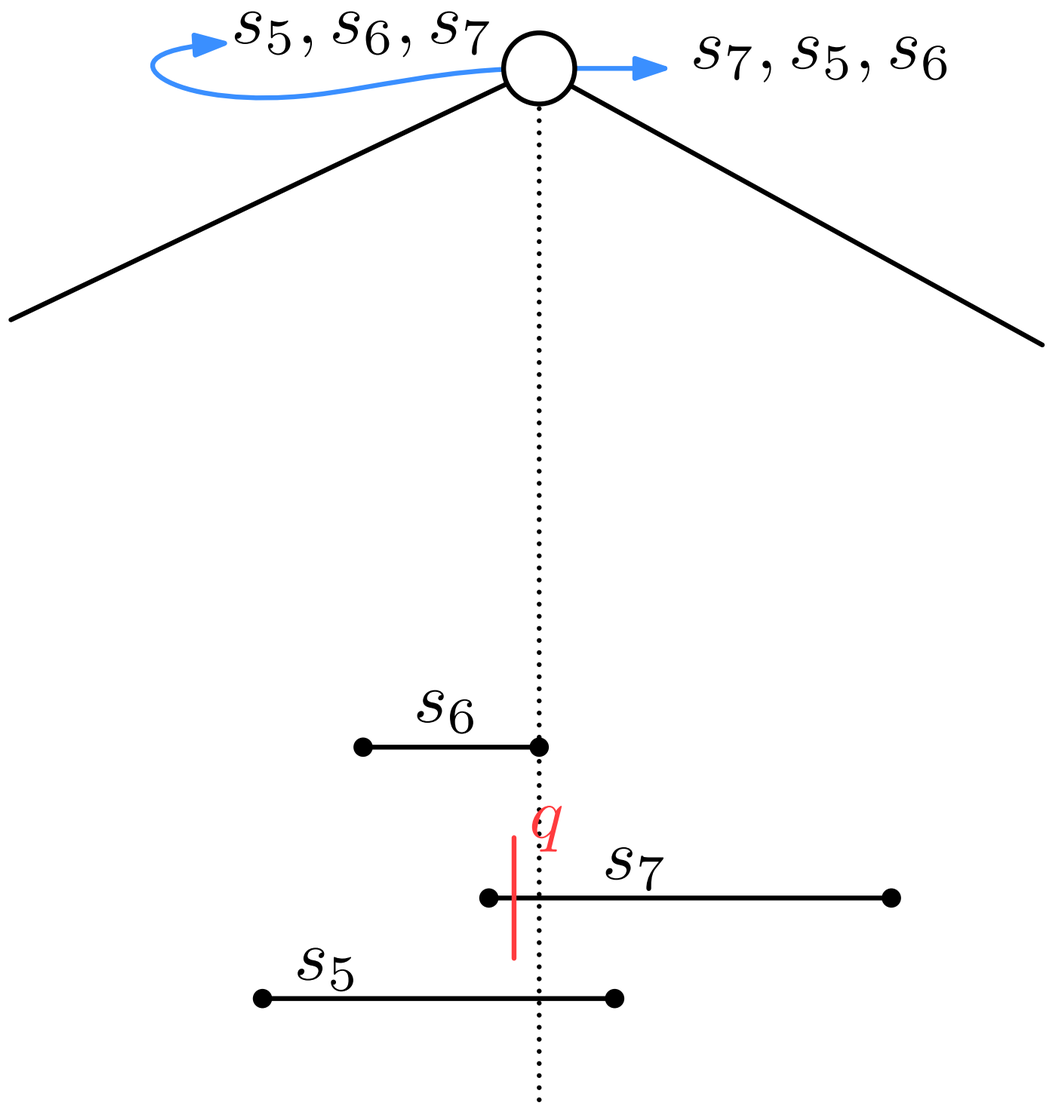
The query problem



The query problem

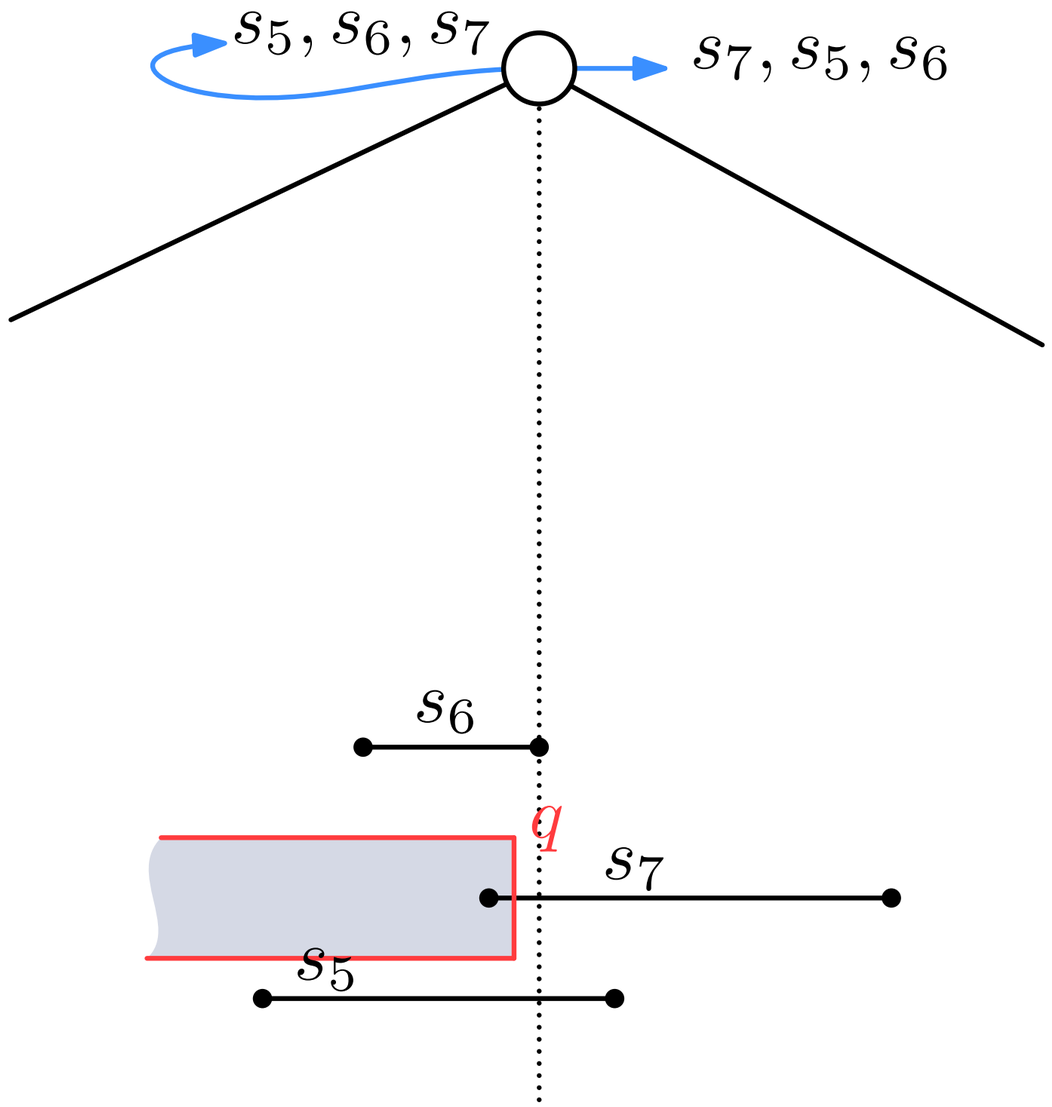


The query problem

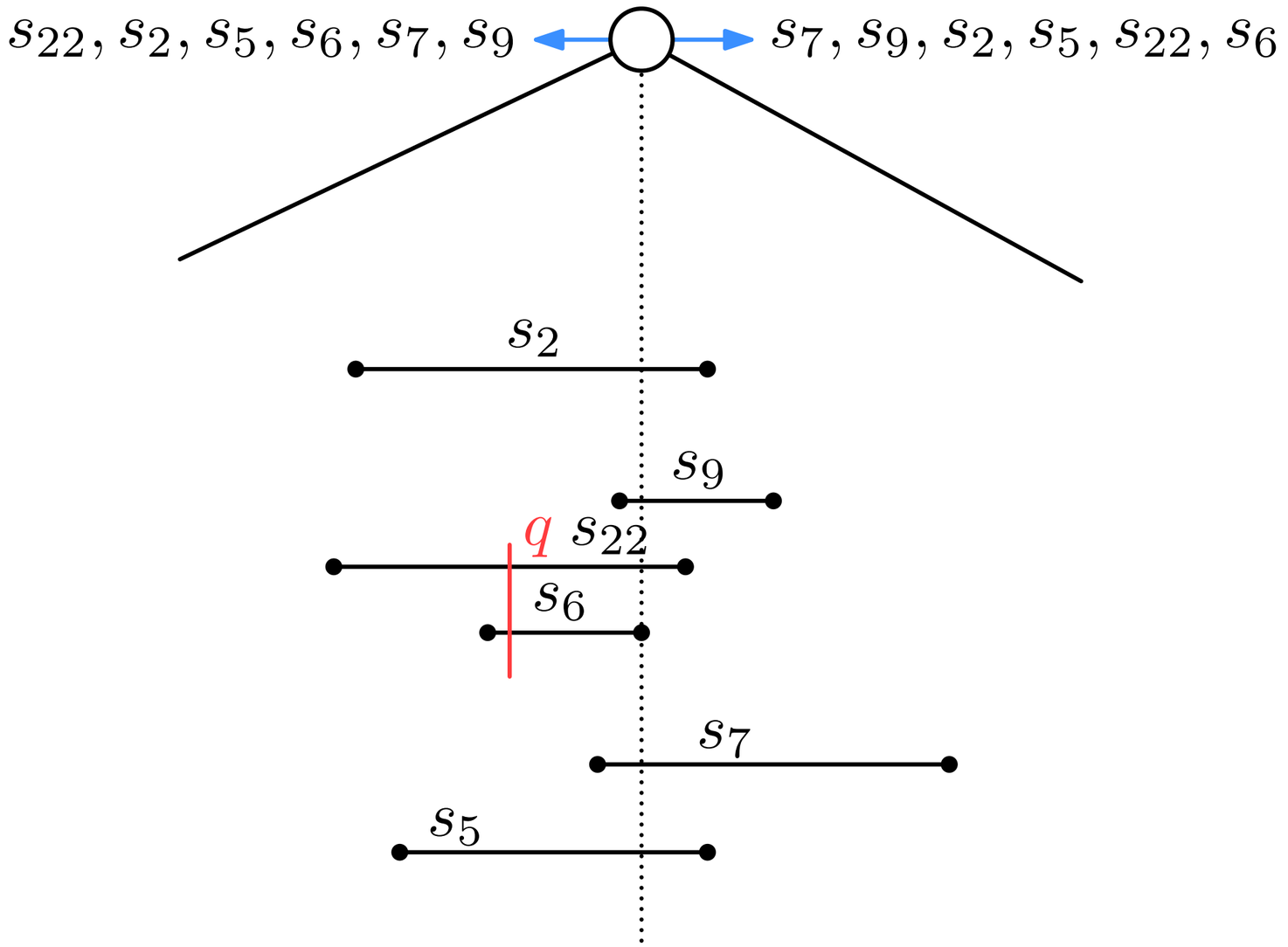


Lists L_{left} and L_{right} no longer suitable for segments of I_{mid}

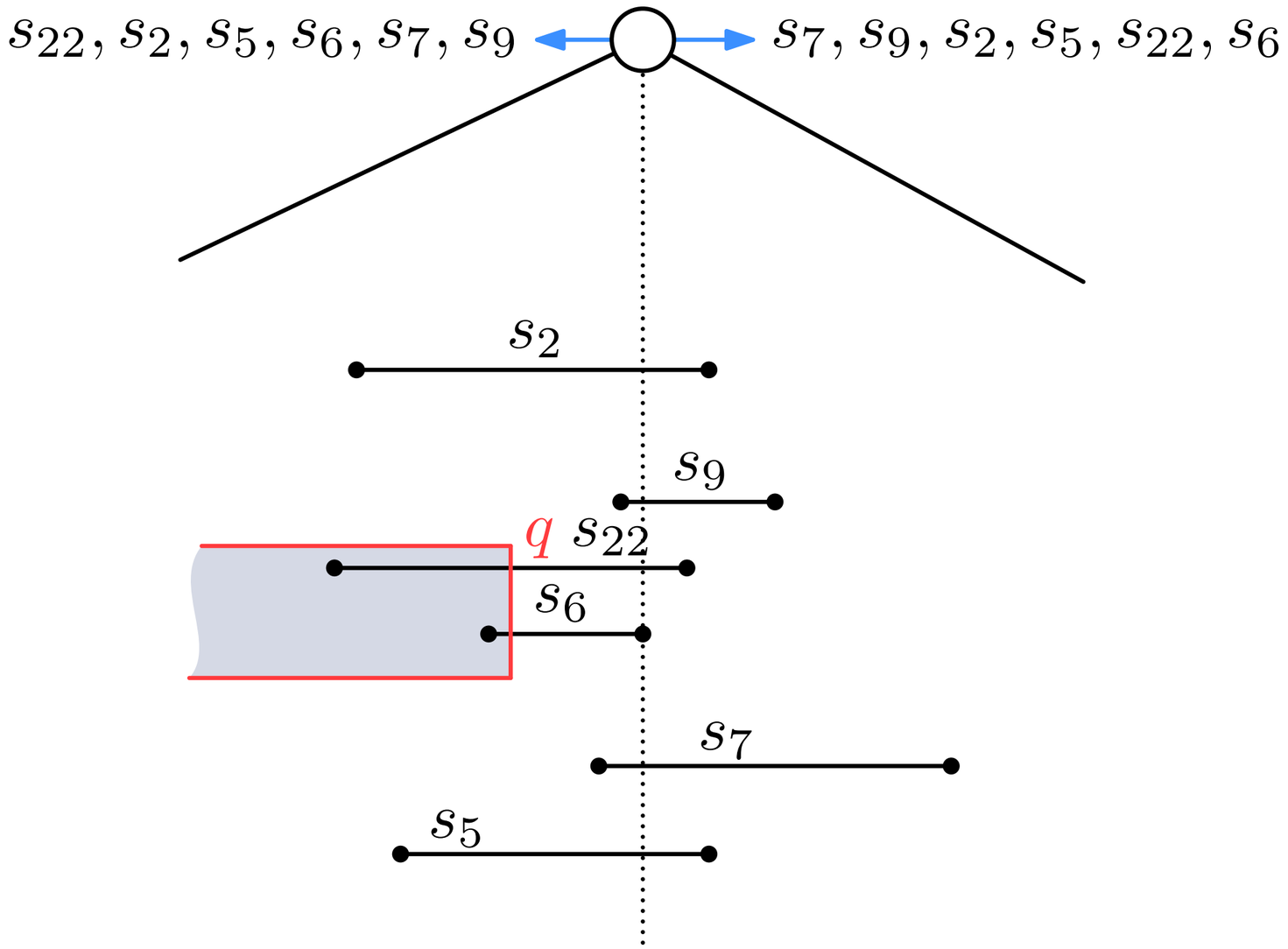
The query problem



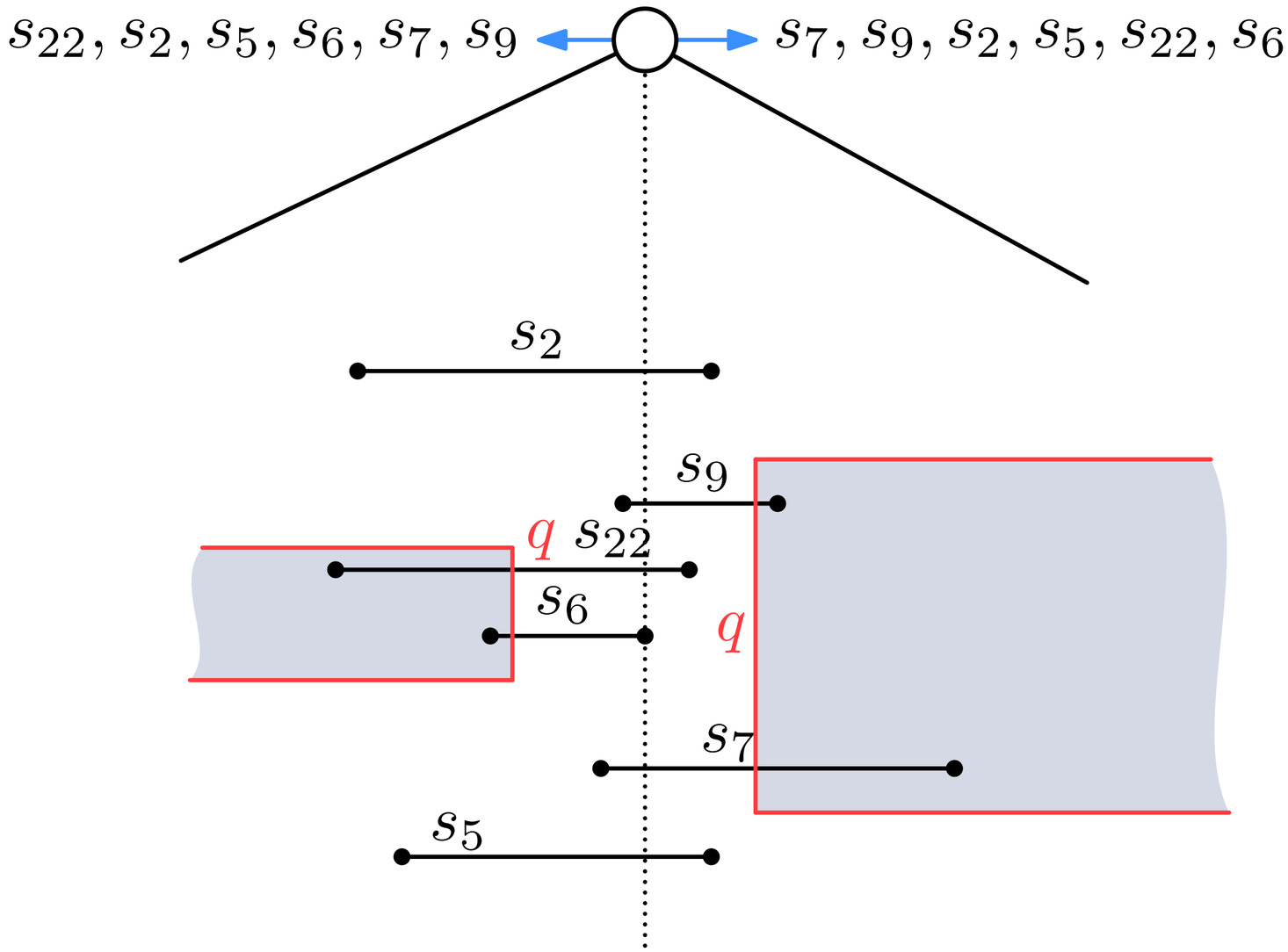
The query problem



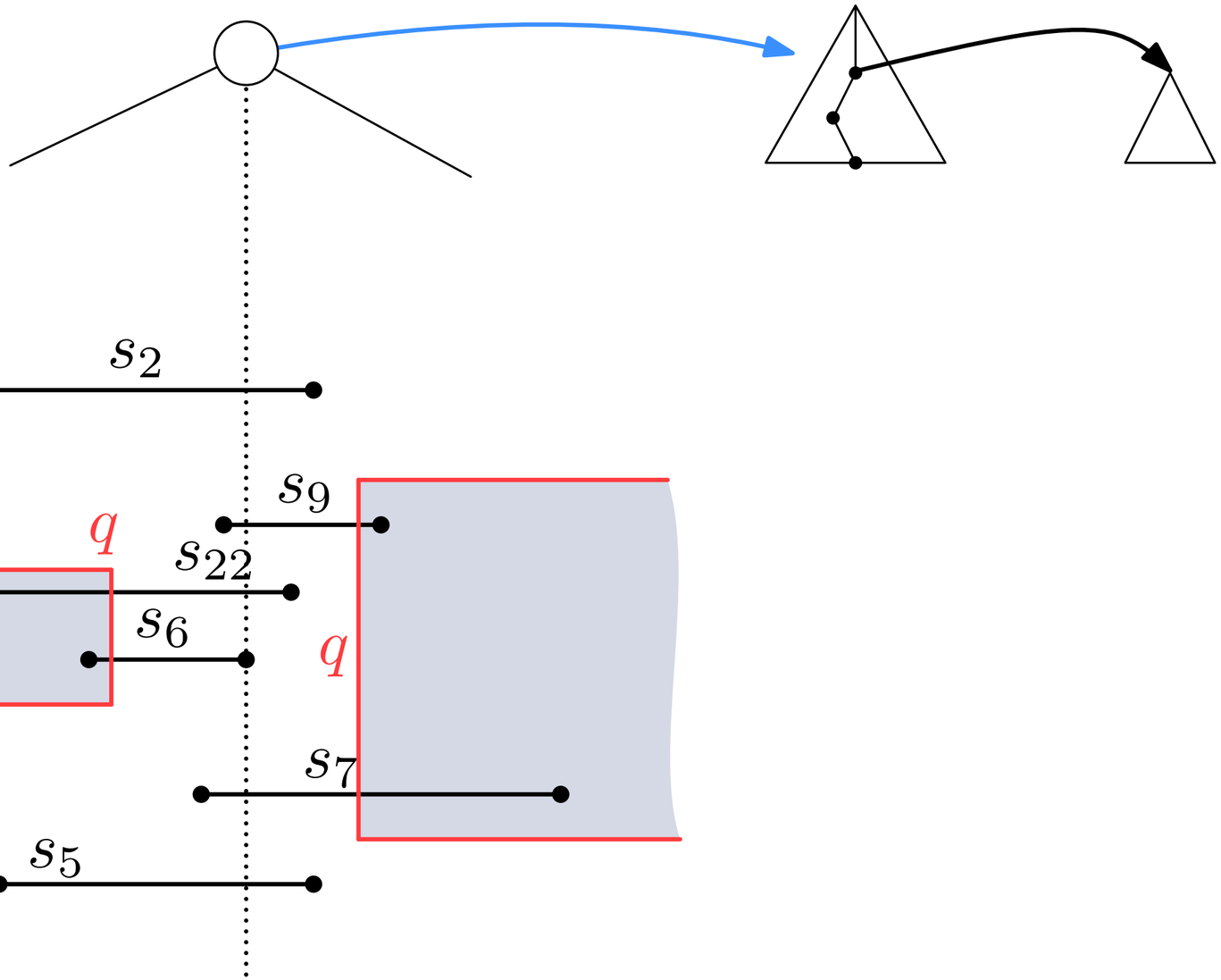
The query problem



The query problem



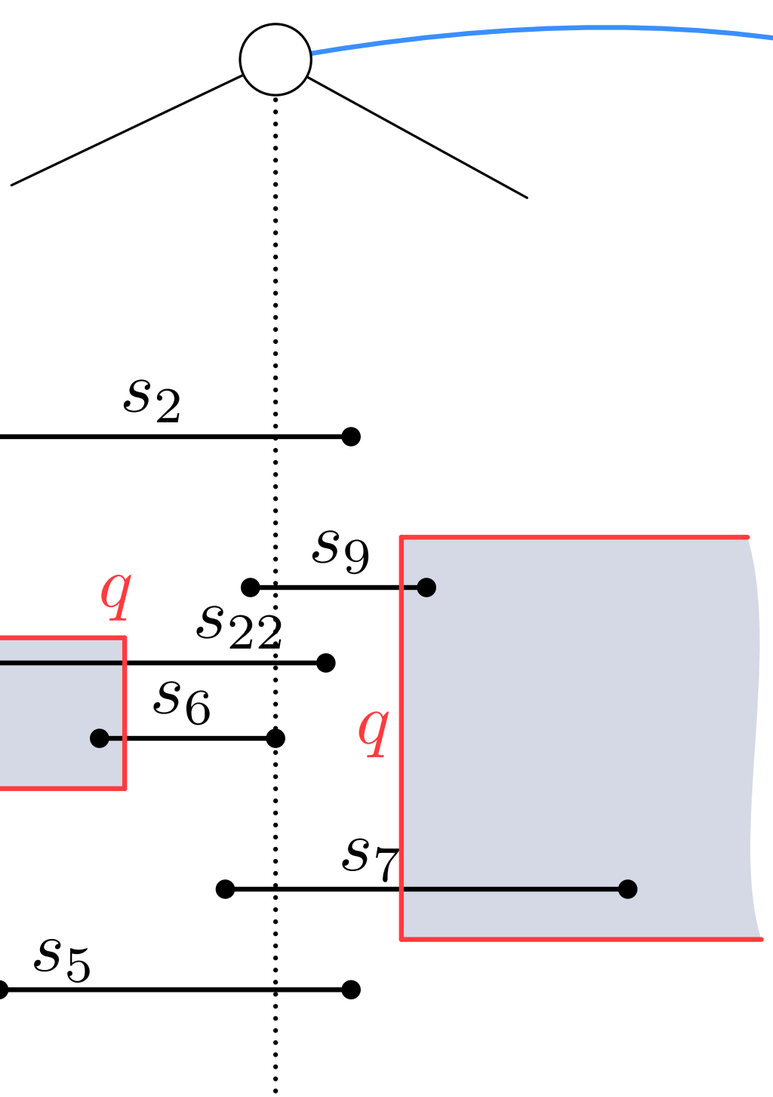
Range trees as associated structure

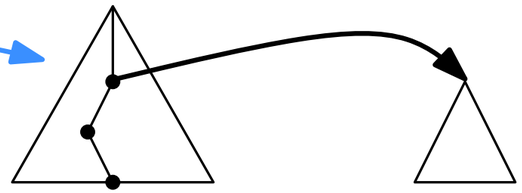


all endpoints of

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

Range trees as associated structure

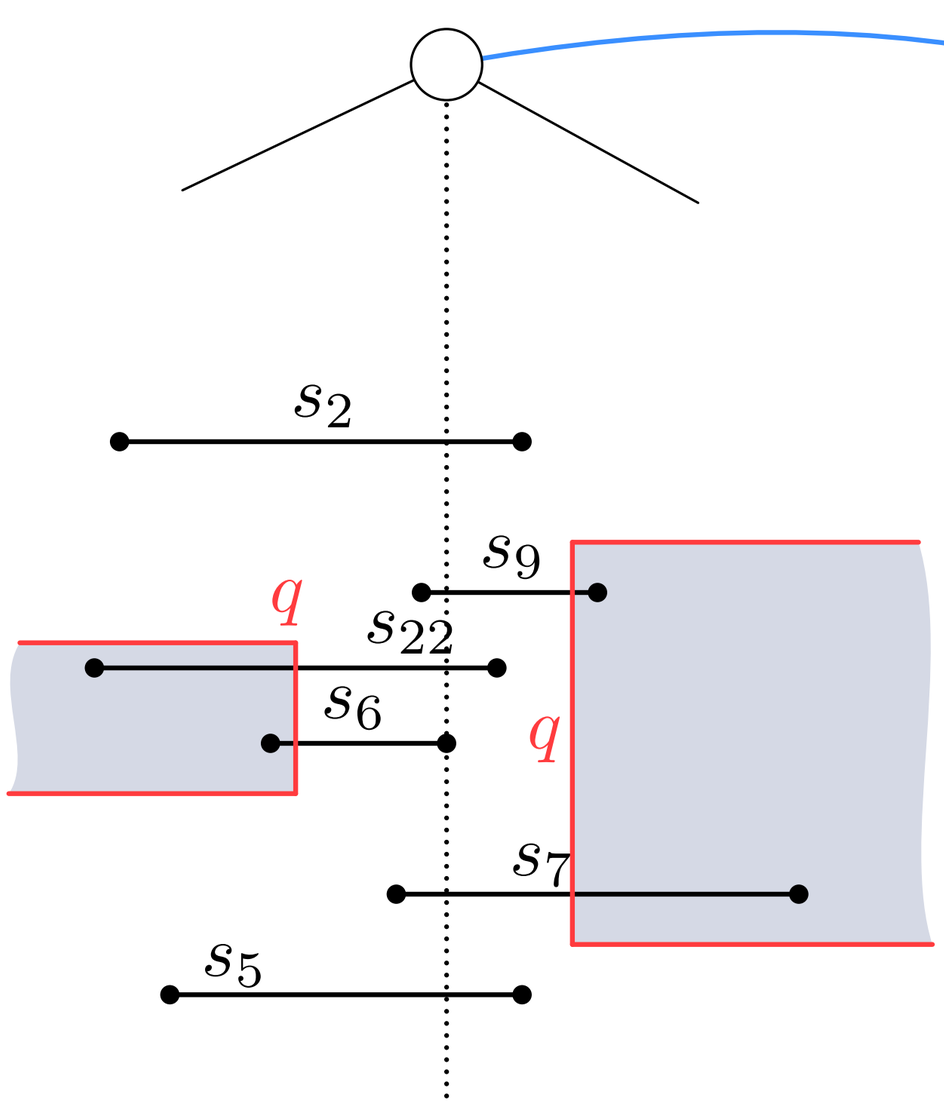


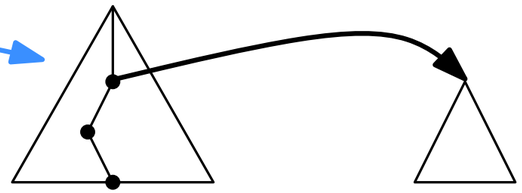


all endpoints of $\{s_2, s_5, s_6, s_7, s_9, s_{22}\}$

Use a range tree as associated structure: only need to store all of the endpoints, to replace L_{left} and L_{right}

Range trees as associated structure





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

Use a range tree as associated structure: only need to store all of the endpoints, to replace L_{left} and L_{right}

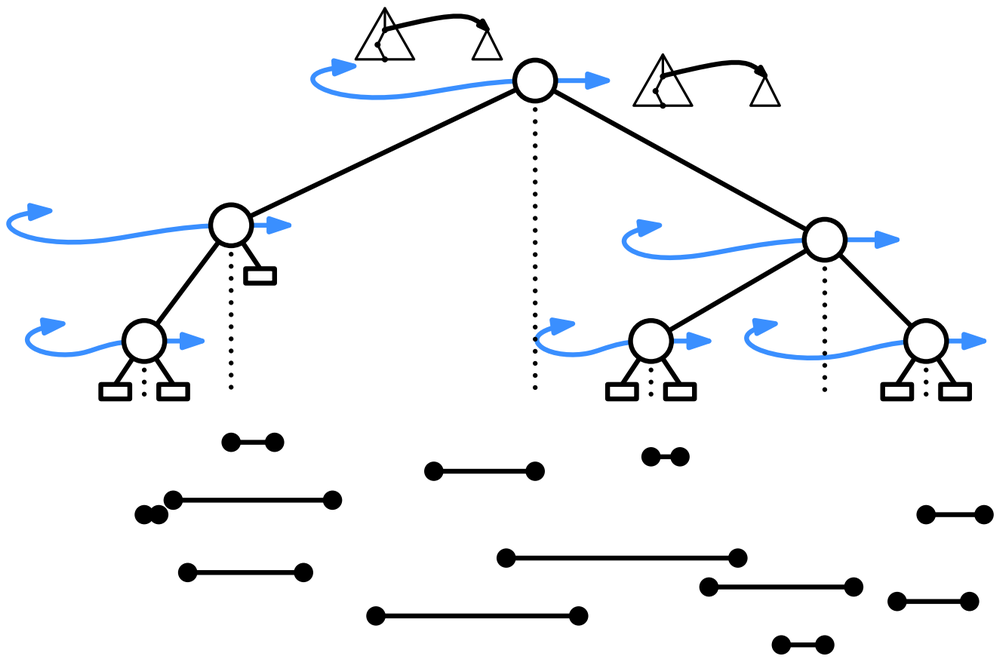
Instead of traversing L_{left} or L_{right}, we perform a query with the region left or right, respectively, of q

Analysis

objects: horizontal segments s_i

query object: vertical segment q

query: report all s_i intersecting q

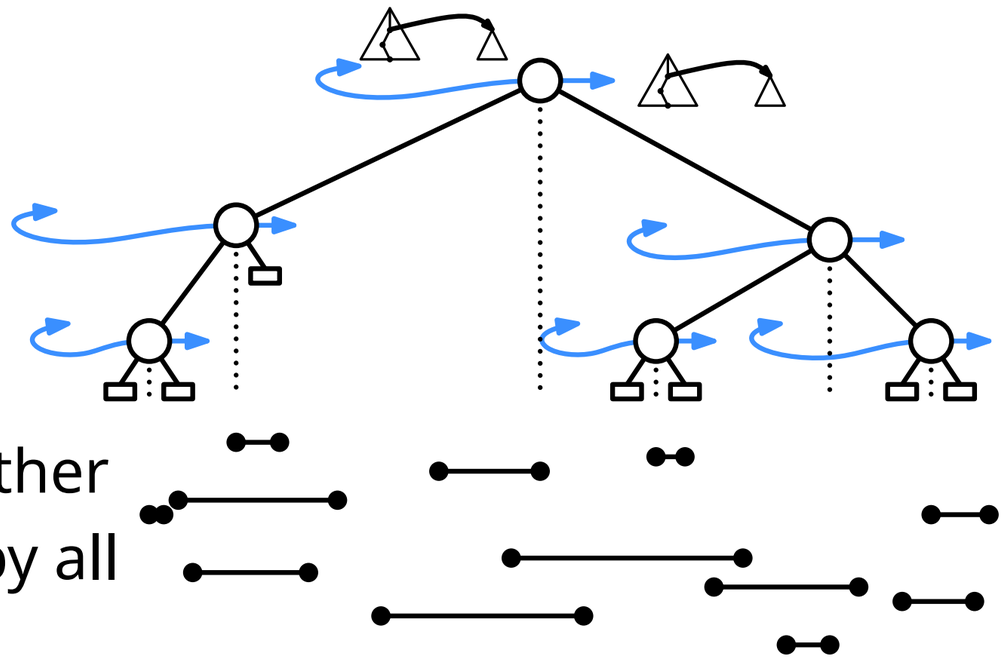


Analysis

objects: horizontal segments s_i

query object: vertical segment q

query: report all s_i intersecting q



Space

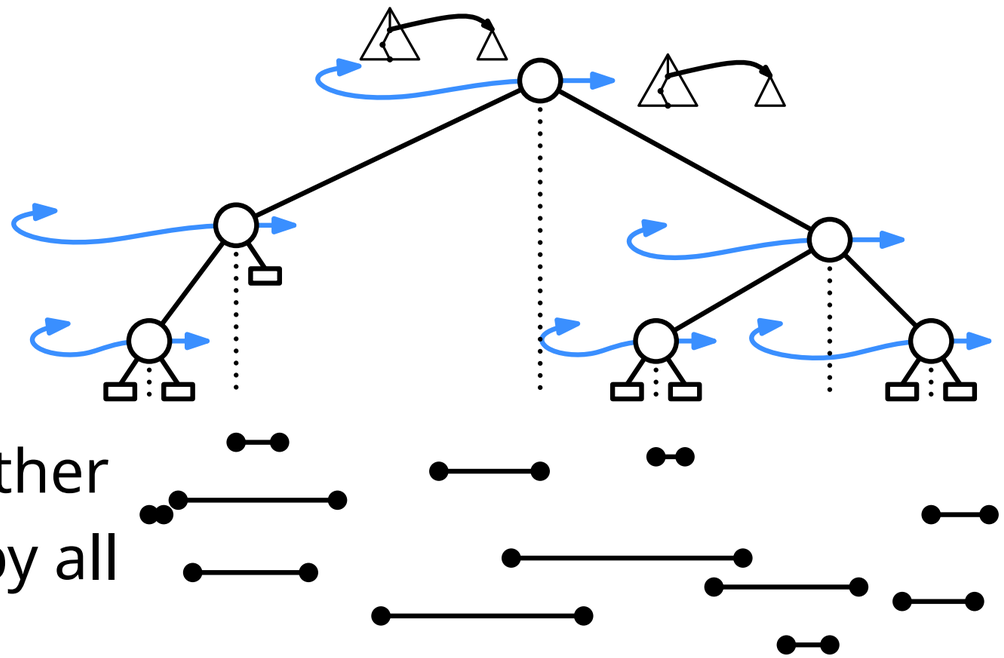
In total, there are $O(n)$ range trees that together store $2n$ points, so the total storage needed by all associated structures is $O(n \log n)$

Analysis

objects: horizontal segments s_i

query object: vertical segment q

query: report all s_i intersecting q



Space

In total, there are $O(n)$ range trees that together store $2n$ points, so the total storage needed by all associated structures is $O(n \log n)$

Query time

A query with a vertical segment leads to $O(\log n)$ range queries

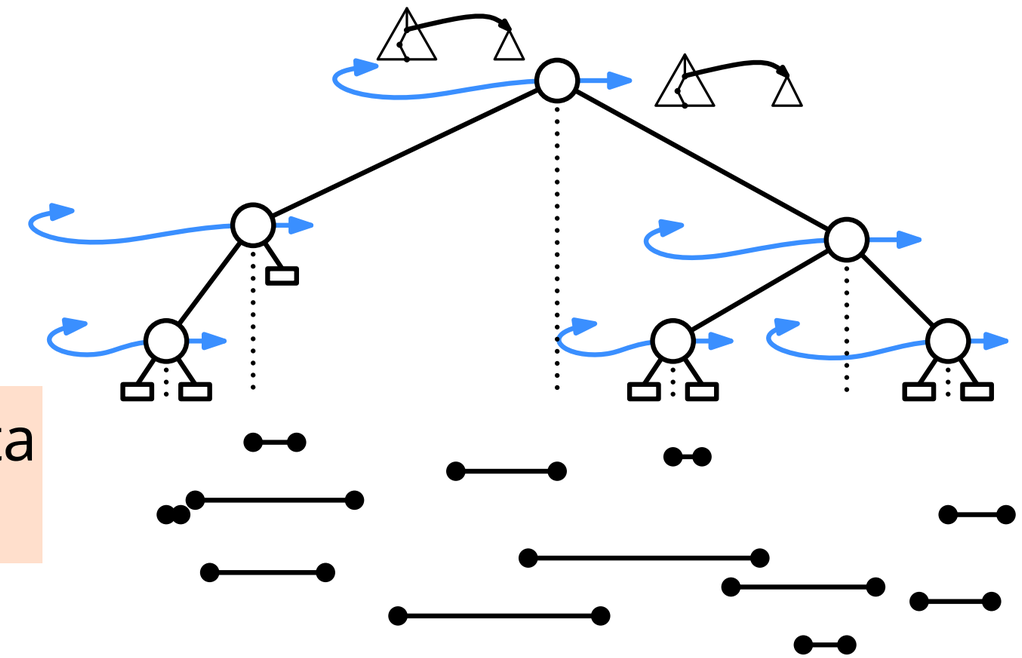
If fractional cascading is used in the associated structures, the overall query time is $O(\log^2 n + k)$

Quiz: Interval tree + Range tree

objects: horizontal segments s_i

query object: vertical segment q

query: report all s_i intersecting q



Question: How fast can we construct this data structure?

A: $\Theta(n \log n)$

B: $\Theta(n \log^2 n)$

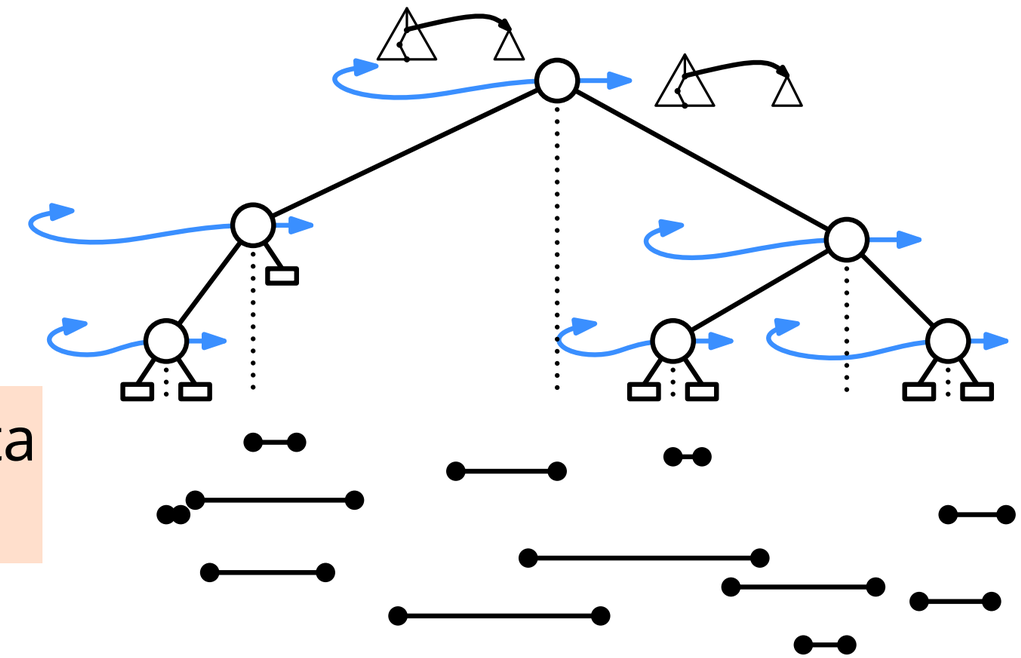
C: $\Theta(n^2)$

Quiz: Interval tree + Range tree

objects: horizontal segments s_i

query object: vertical segment q

query: report all s_i intersecting q



Question: How fast can we construct this data structure?

A: $\Theta(n \log n)$

B: $\Theta(n \log^2 n)$

C: $\Theta(n^2)$

Result

Theorem: A set of n horizontal line segments can be stored in a data structure of size $O(n \log n)$ such that intersection queries with a vertical line segment can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported. The data structure can be build in $O(n \log n)$ time.

Result

Theorem: A set of n horizontal line segments can be stored in a data structure of size $O(n \log n)$ such that intersection queries with a vertical line segment can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported. The data structure can be build in $O(n \log n)$ time.

Theorem: A set of n axis-parallel line segments can be stored in a data structure of size $O(n \log n)$ such that windowing queries can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported. The data structure can be build in $O(n \log n)$ time.

Result

Theorem: A set of n horizontal line segments can be stored in a data structure of size $O(n \log n)$ such that intersection queries with a vertical line segment can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported. The data structure can be build in $O(n \log n)$ time.

Theorem: A set of n axis-parallel line segments can be stored in a data structure of size $O(n \log n)$ such that windowing queries can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported. The data structure can be build in $O(n \log n)$ time.

necessary?

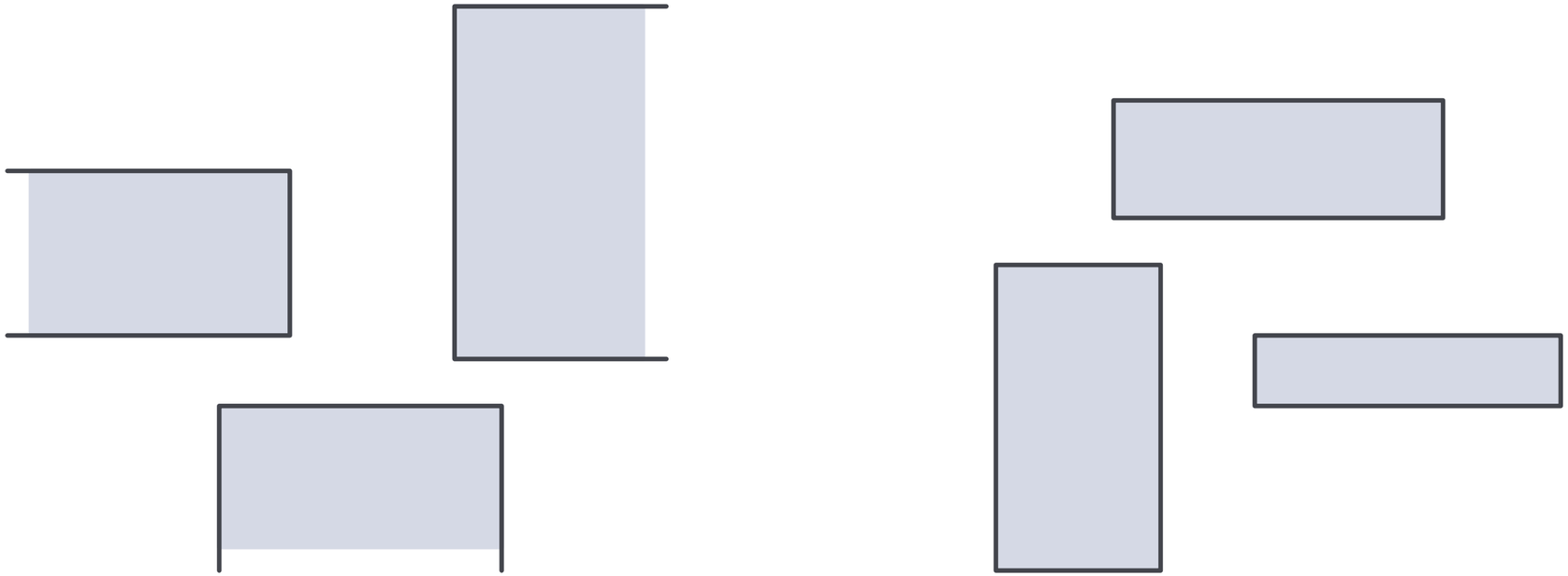
Range and Windowing Queries

Priority search trees for windowing queries

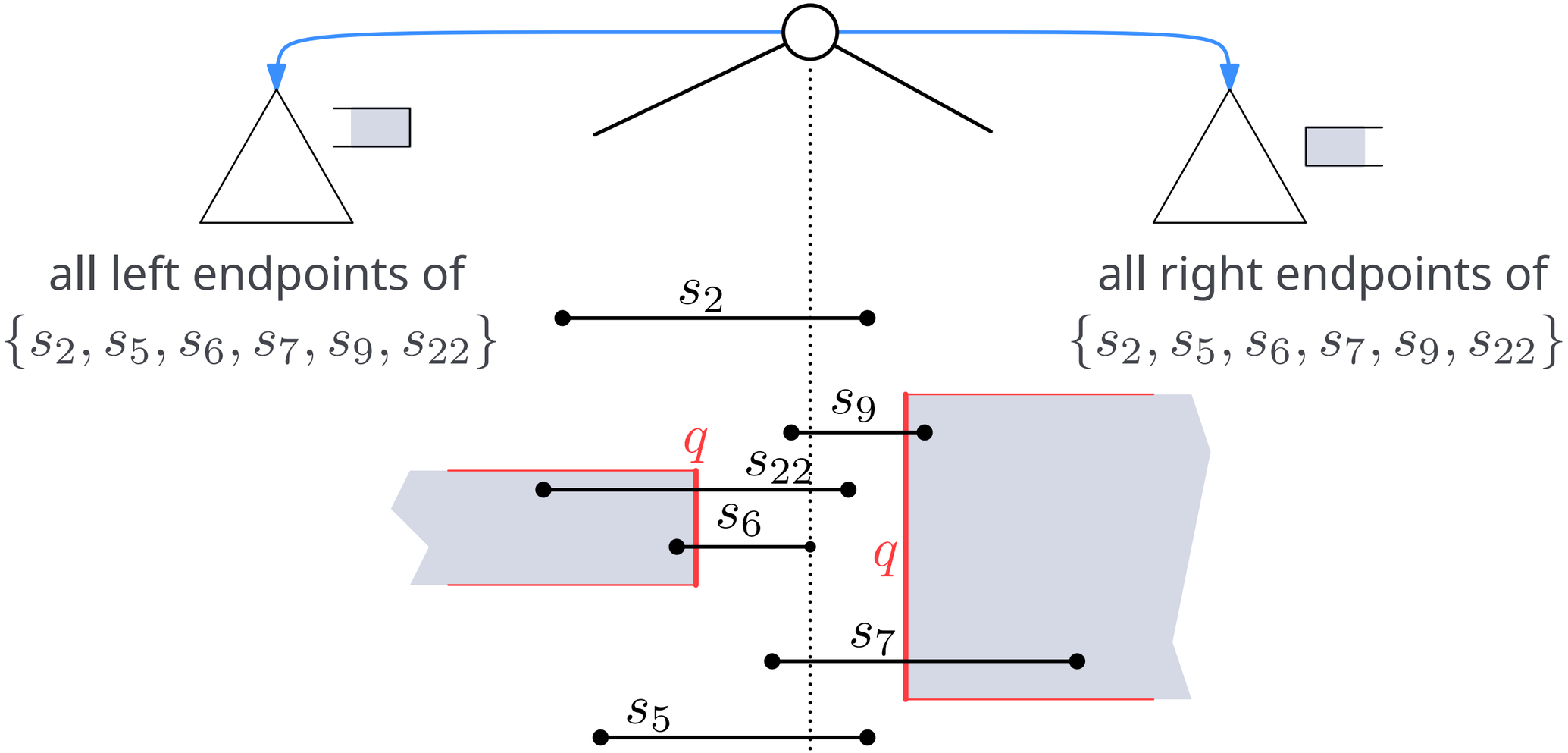
3- and 4-sided ranges

Considering the associated structure, we only need 3-sided range queries, whereas the range tree provides 4-sided range queries

Can the 3-sided range query problem be solved more efficiently than the 4-sided (rectangular) range query problem?



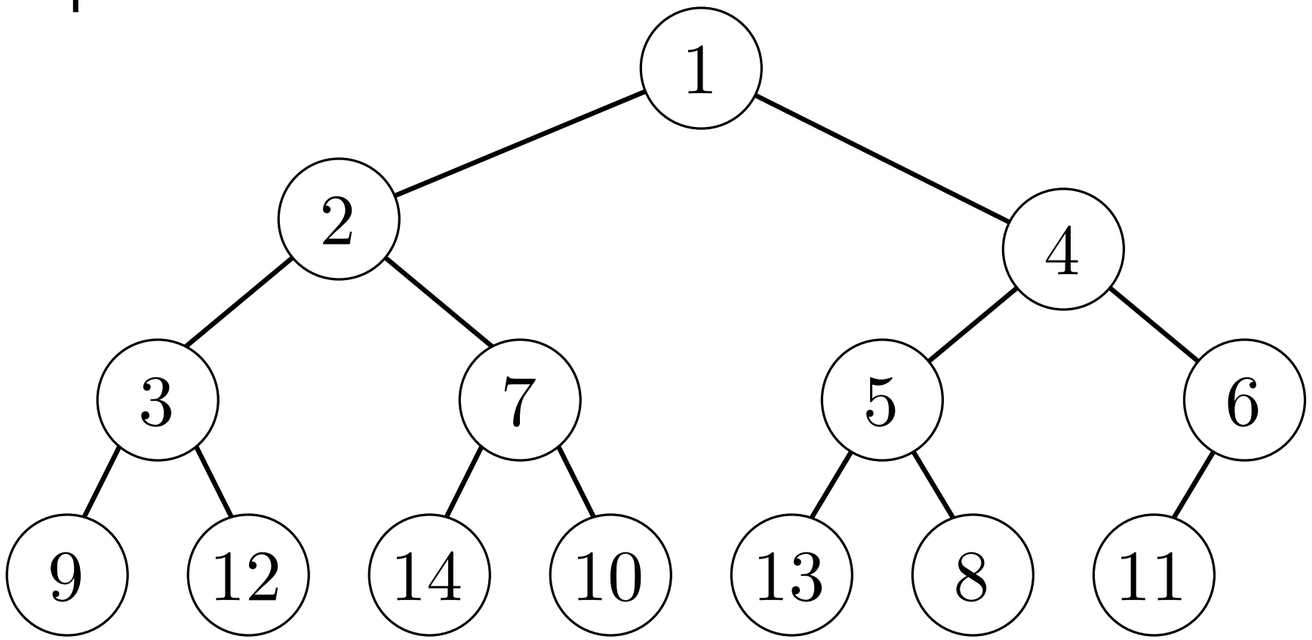
Windowing Data Structure



Heap and search tree

A priority search tree is like a heap on x-coordinate and a binary search tree on y-coordinate at the same time

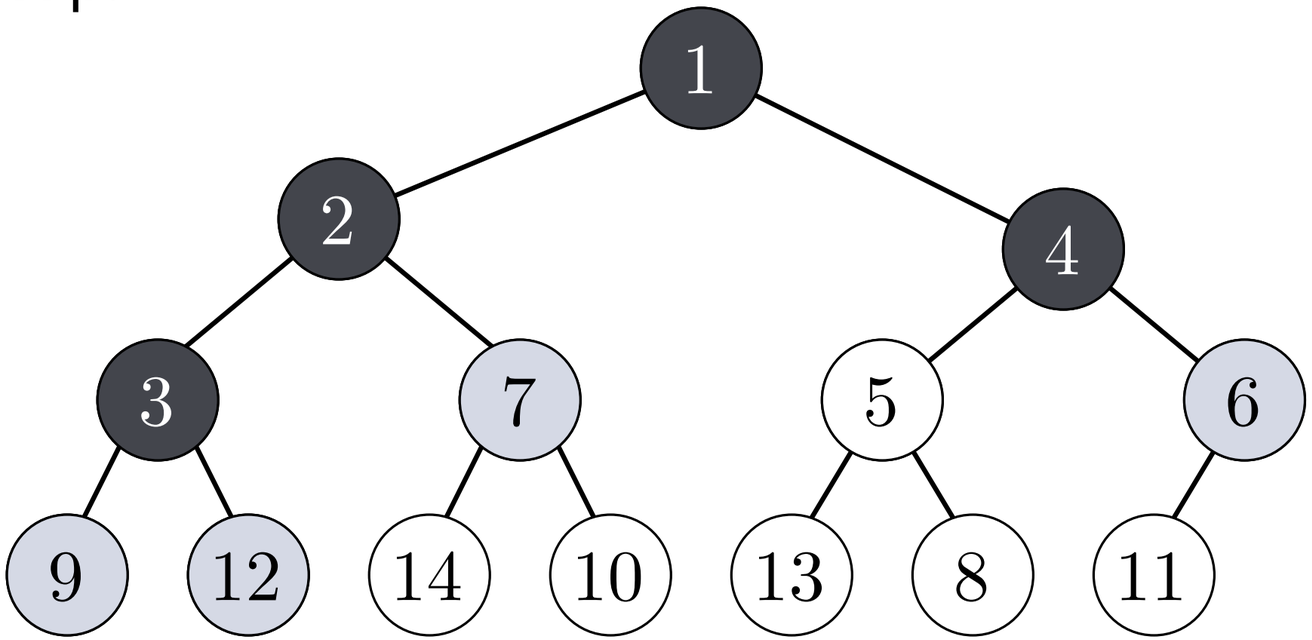
Recall the heap:



Heap and search tree

A priority search tree is like a heap on x-coordinate and a binary search tree on y-coordinate at the same time

Recall the heap:



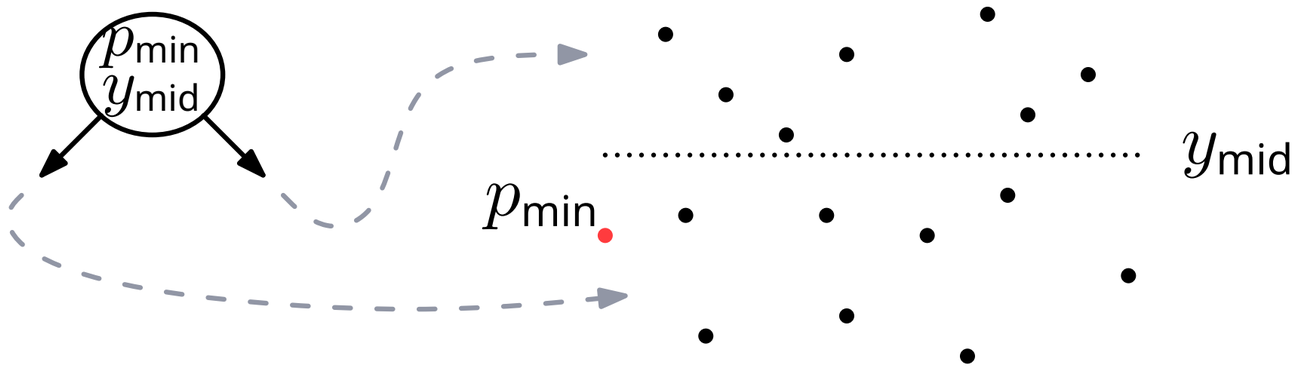
report all values ≤ 4

Priority search tree: query

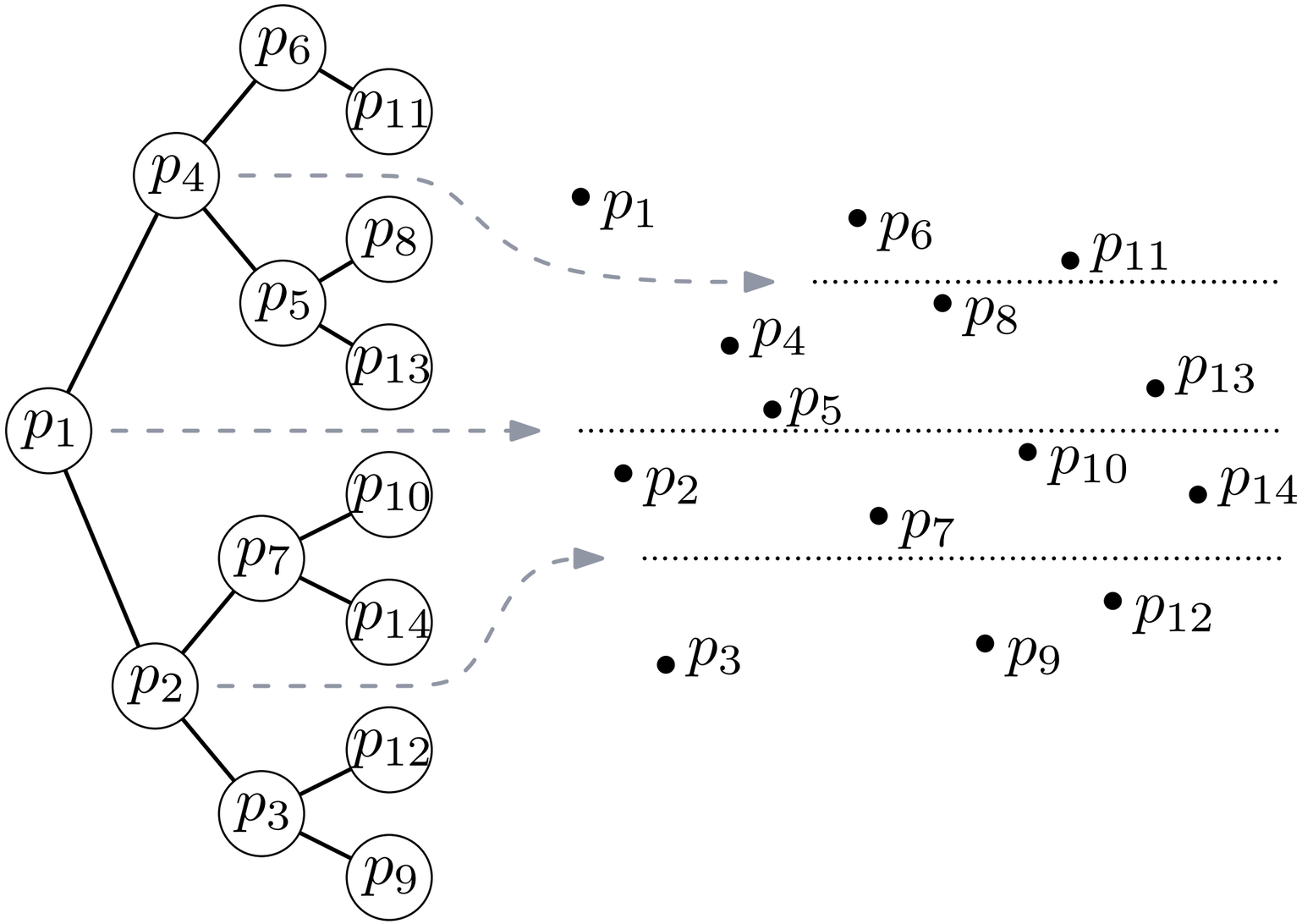
If $P = \varnothing$, then a priority search tree is an empty leaf

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

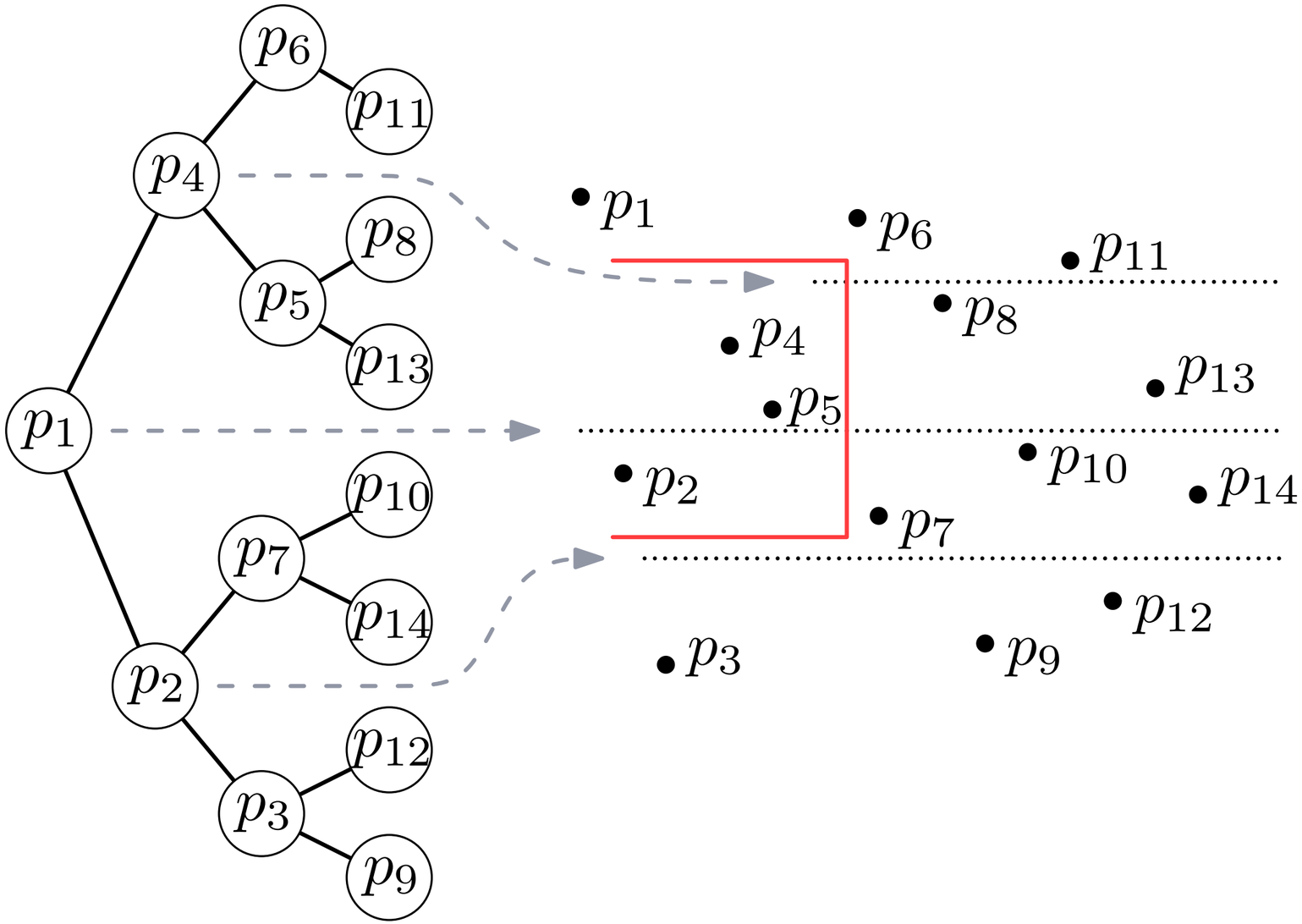
The priority search tree has a node v that stores p_{min} and y_{mid}, and a left subtree and right subtree for the points in $P \backslash \{p_{\text{min}}\}$ with y-coordinate $\leq y_{\text{mid}}$ and $> y_{\text{mid}}$



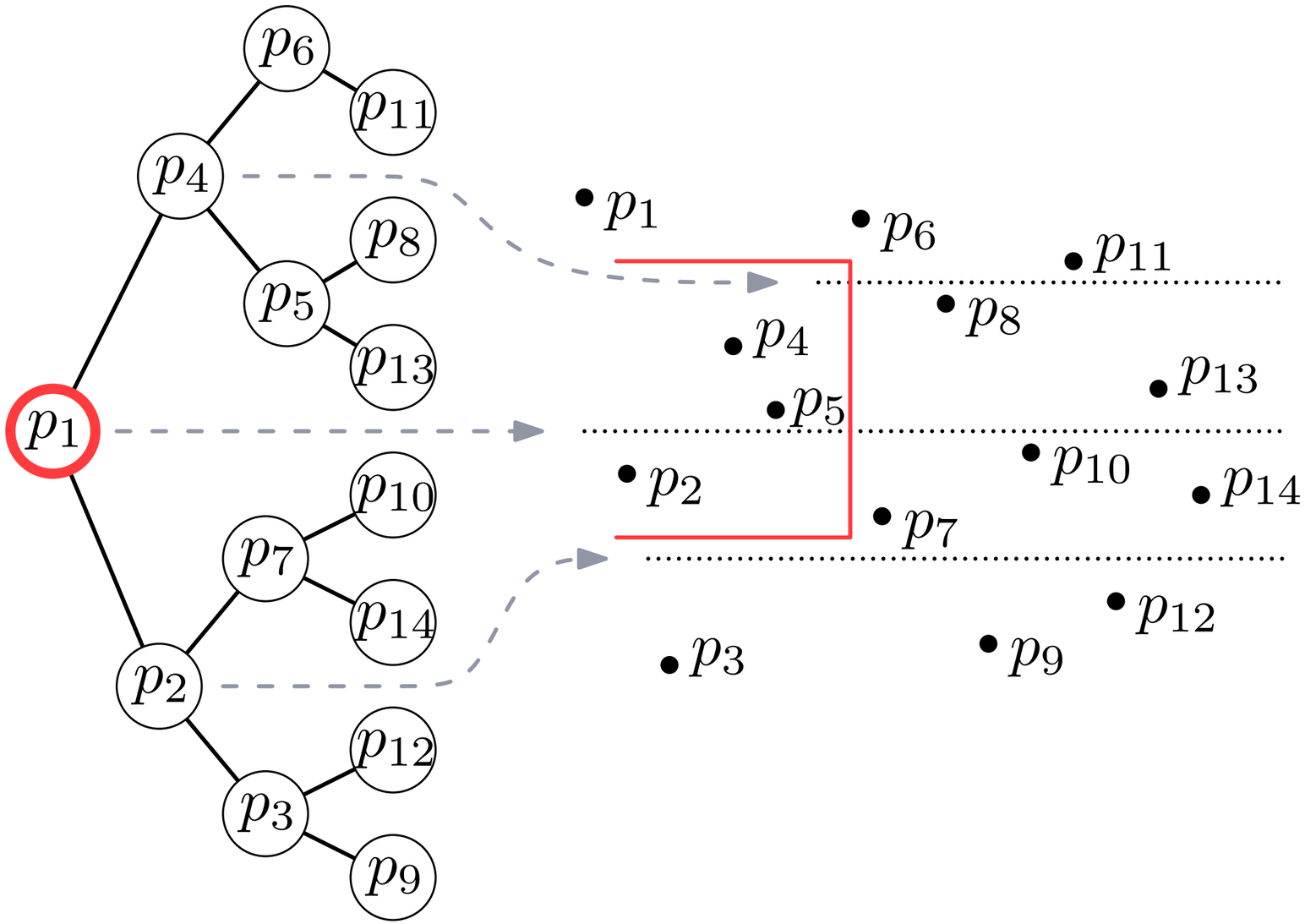
Priority search tree: query



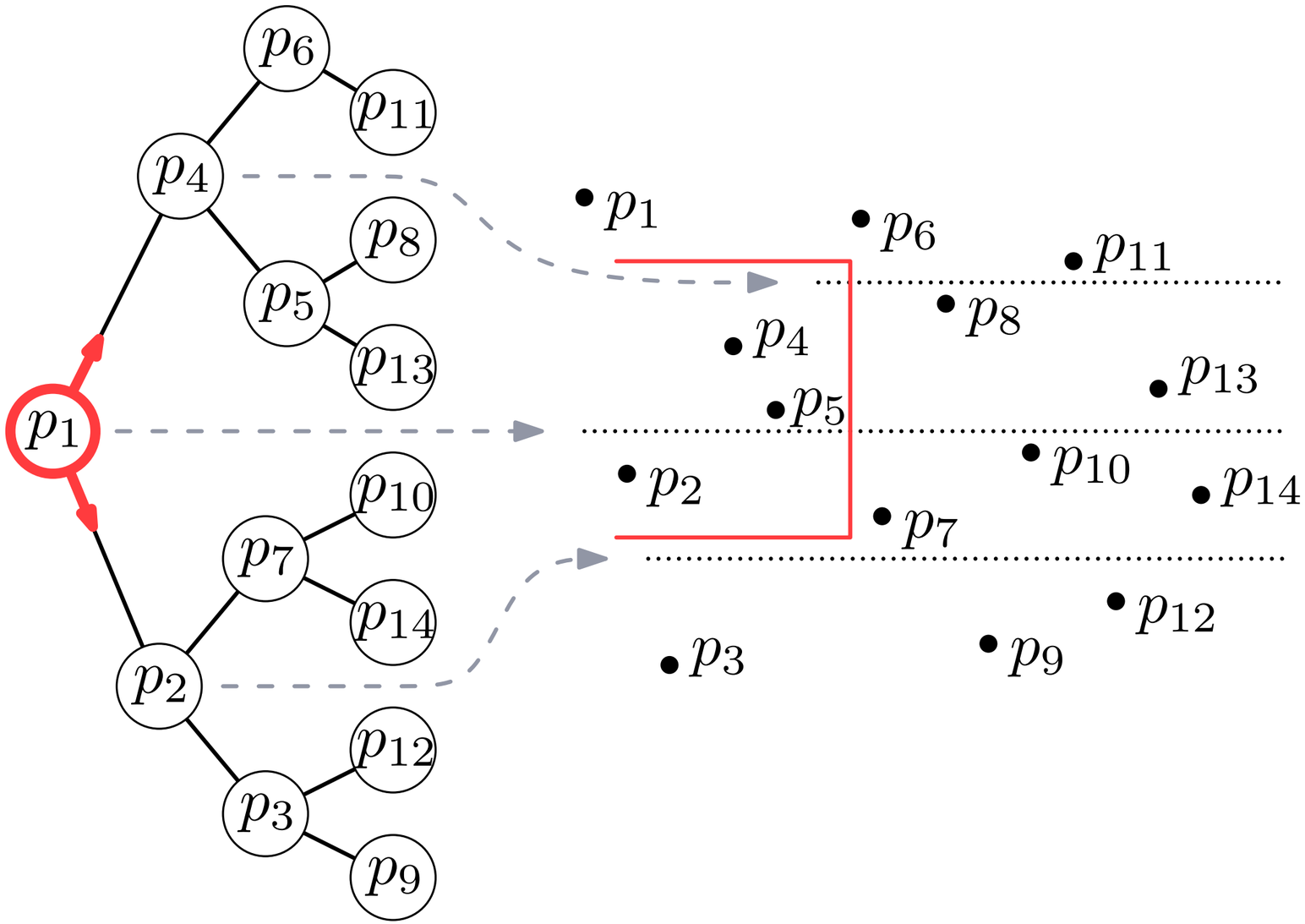
Priority search tree: query



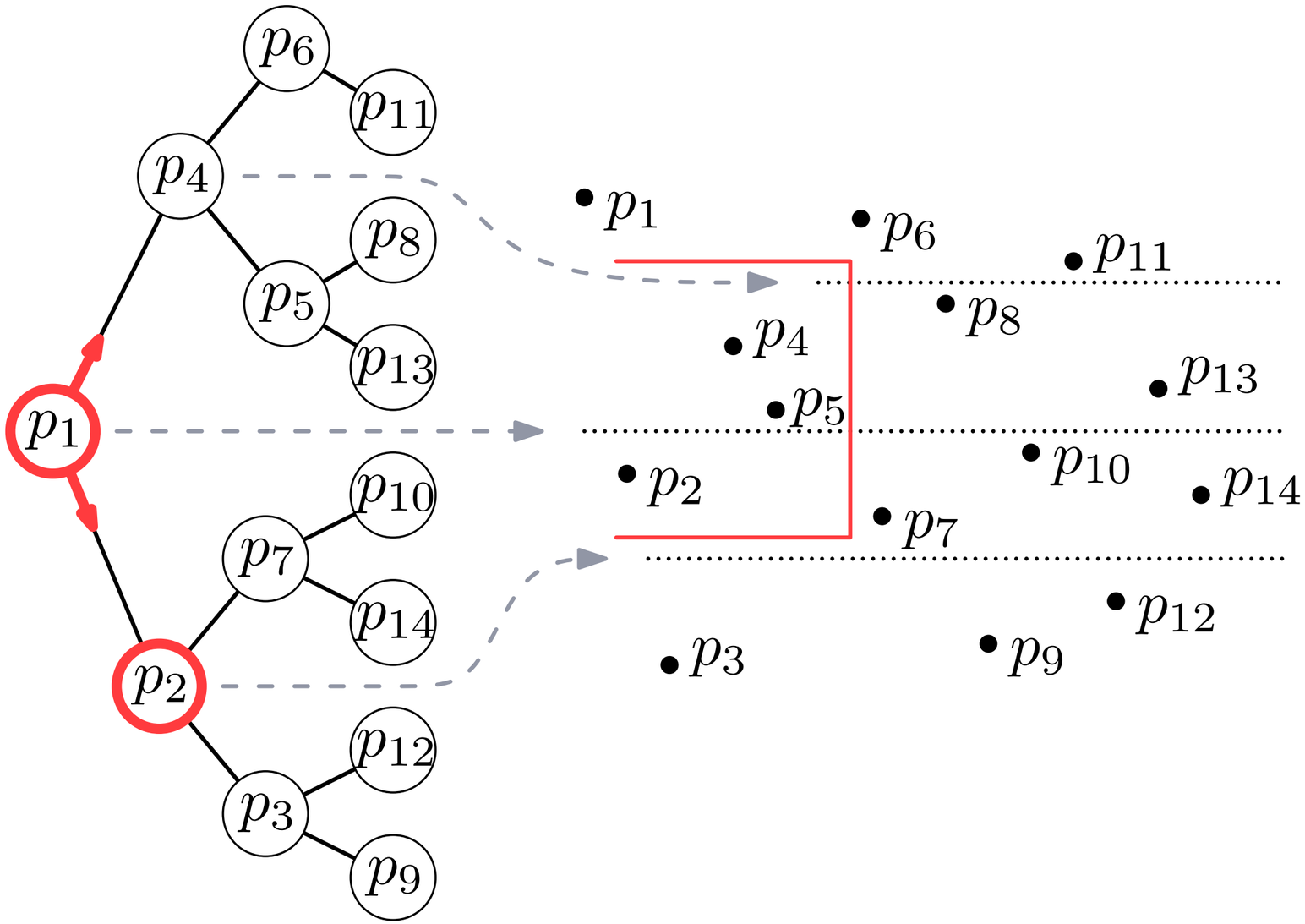
Priority search tree: query



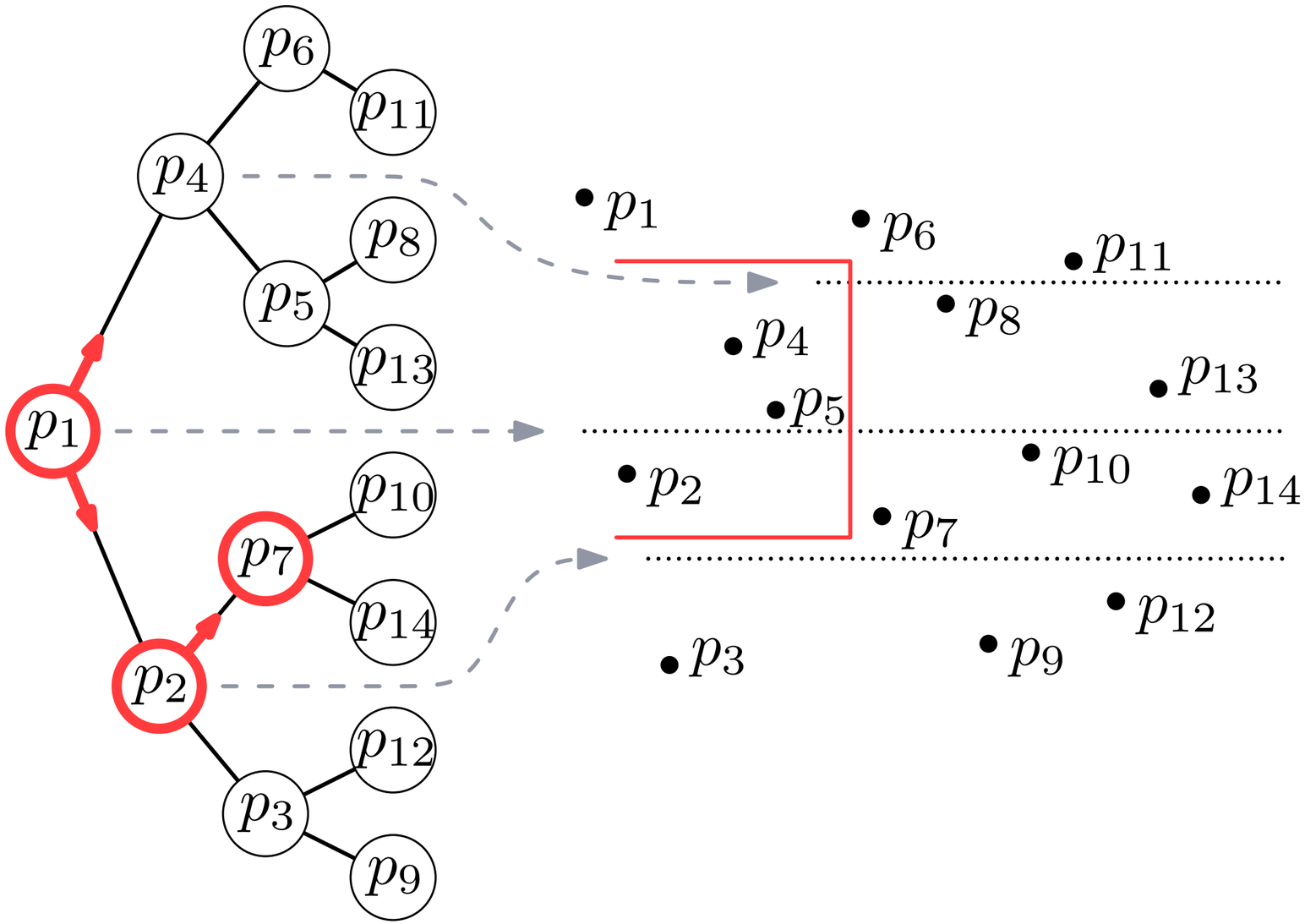
Priority search tree: query



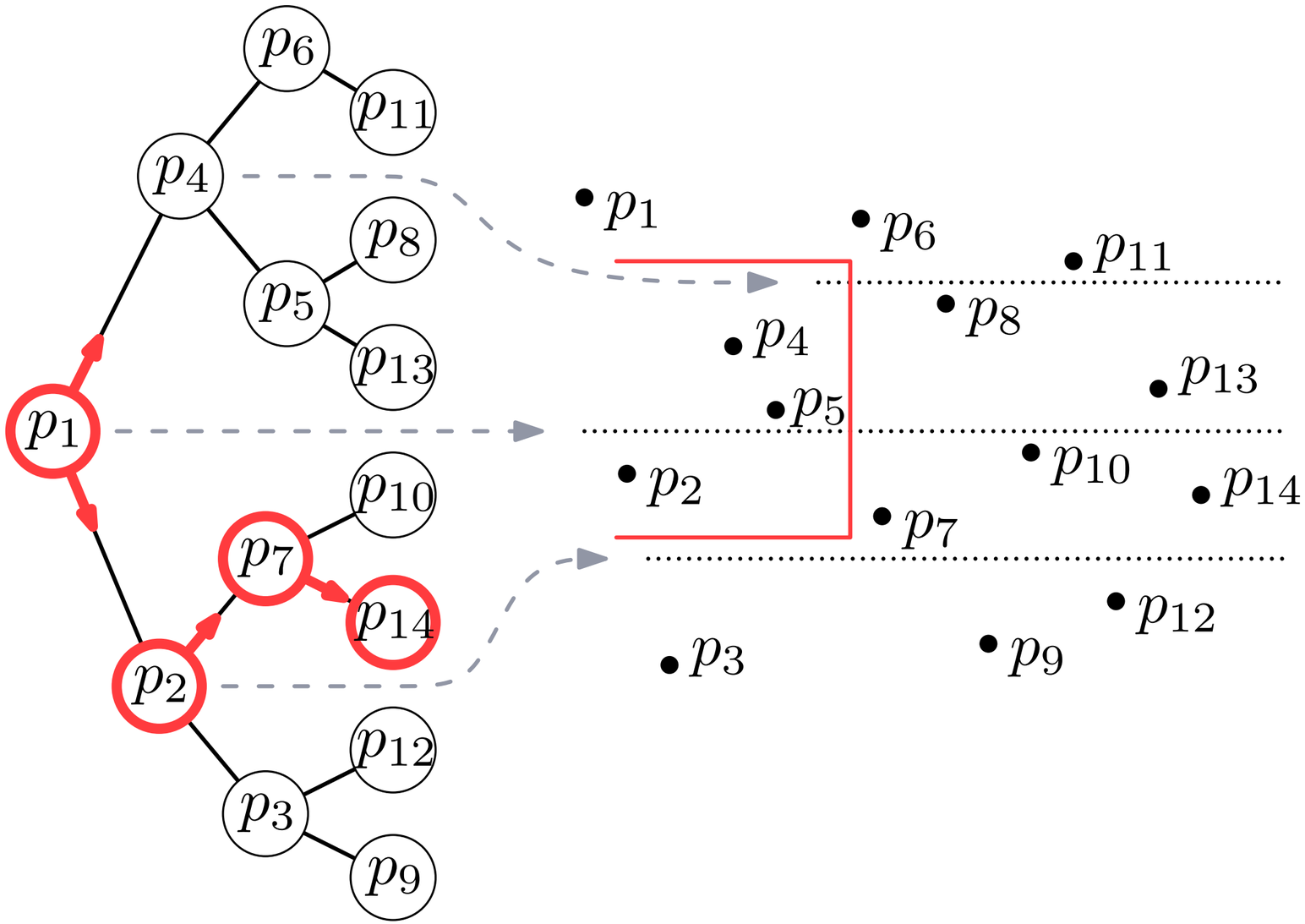
Priority search tree: query



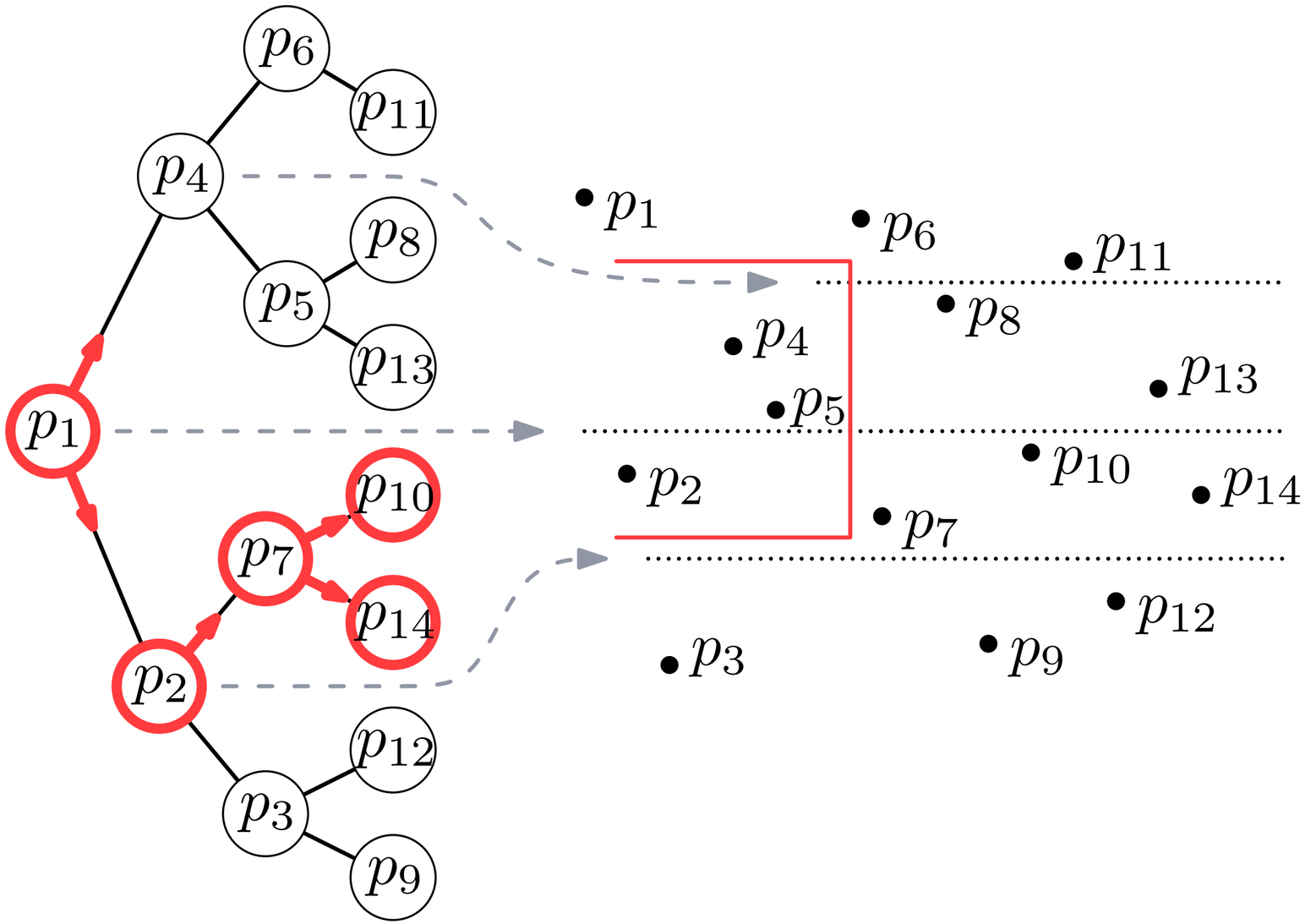
Priority search tree: query



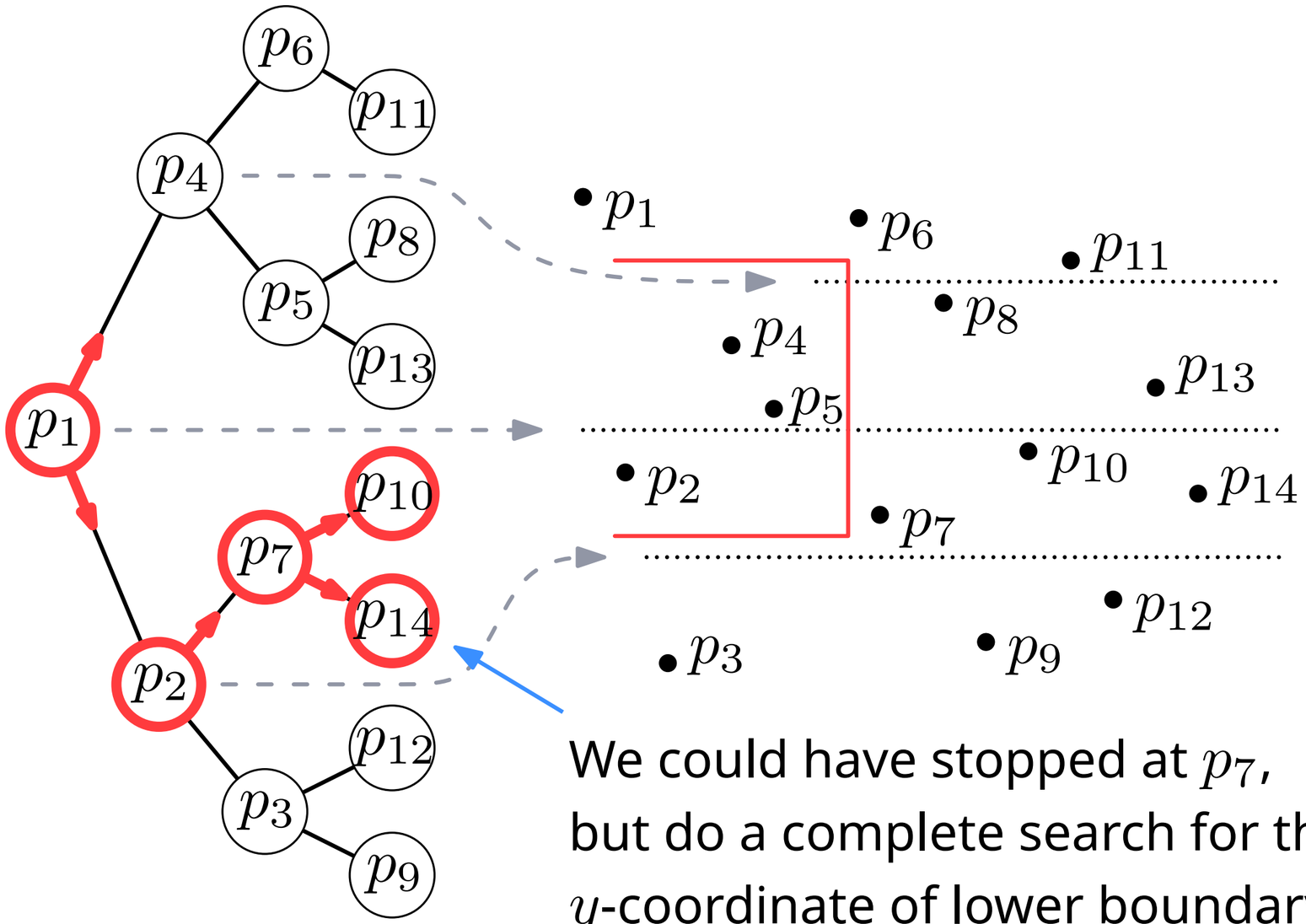
Priority search tree: query



Priority search tree: query

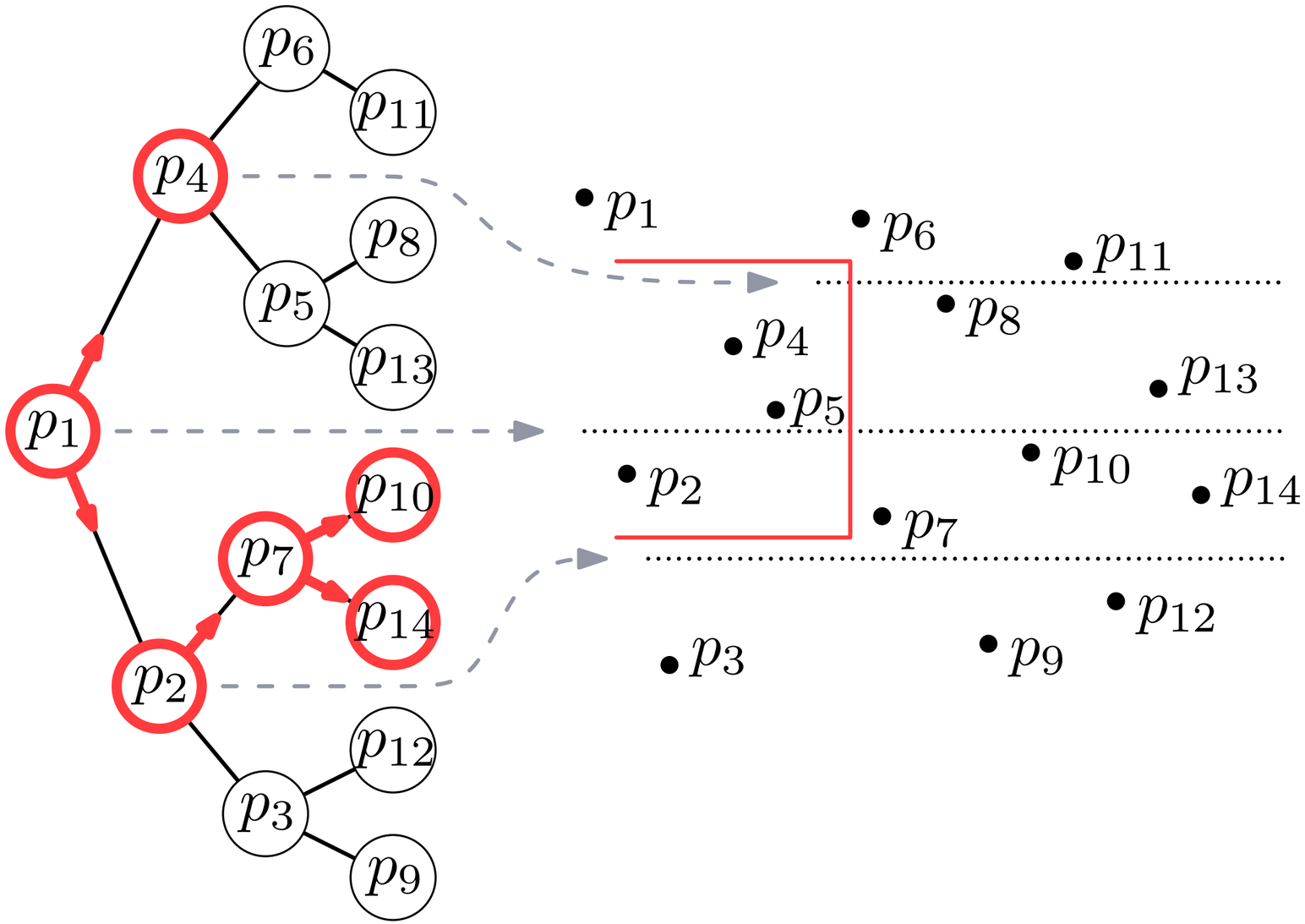


Priority search tree: query

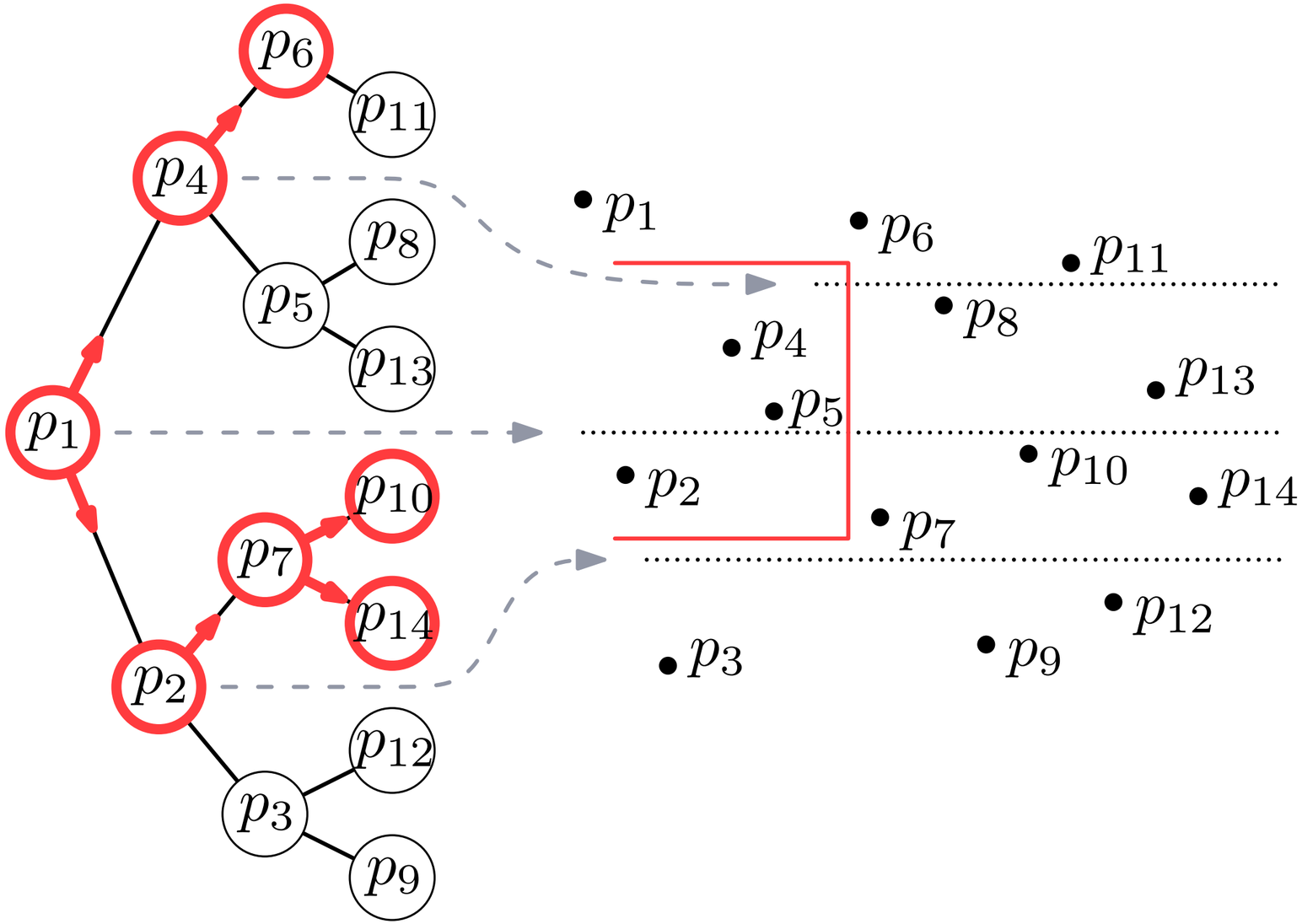


We could have stopped at p_7, but do a complete search for the y-coordinate of lower boundary

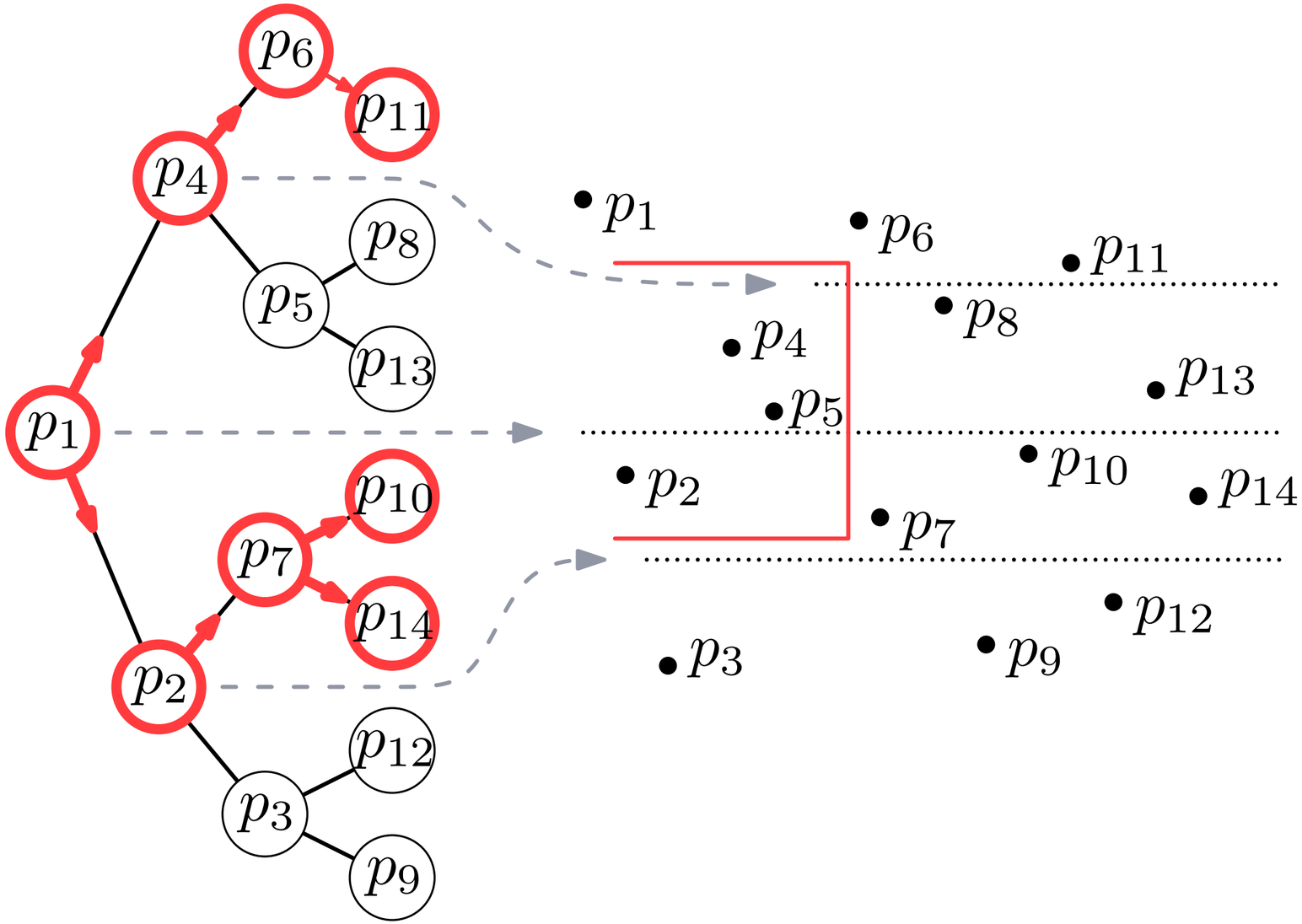
Priority search tree: query



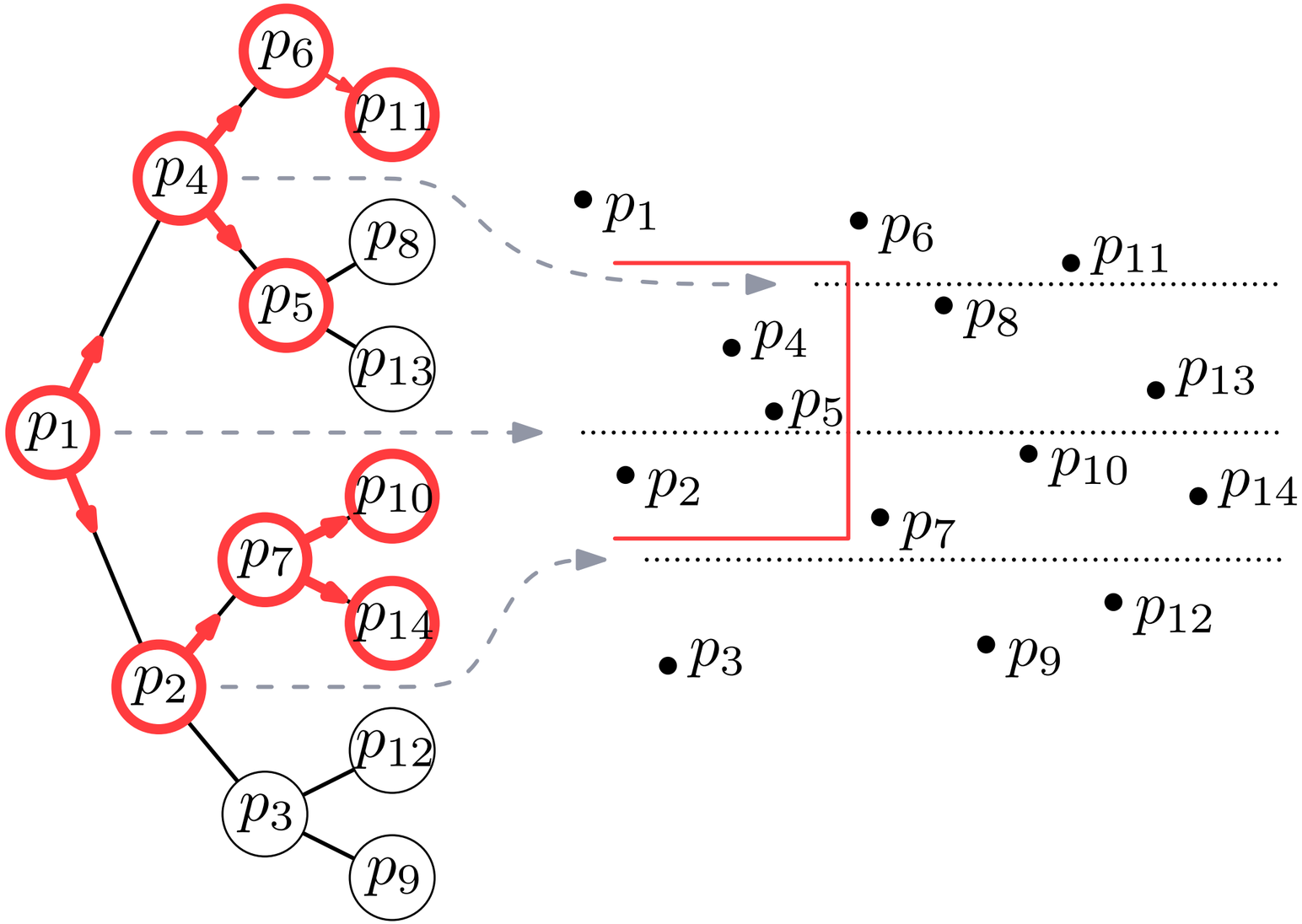
Priority search tree: query



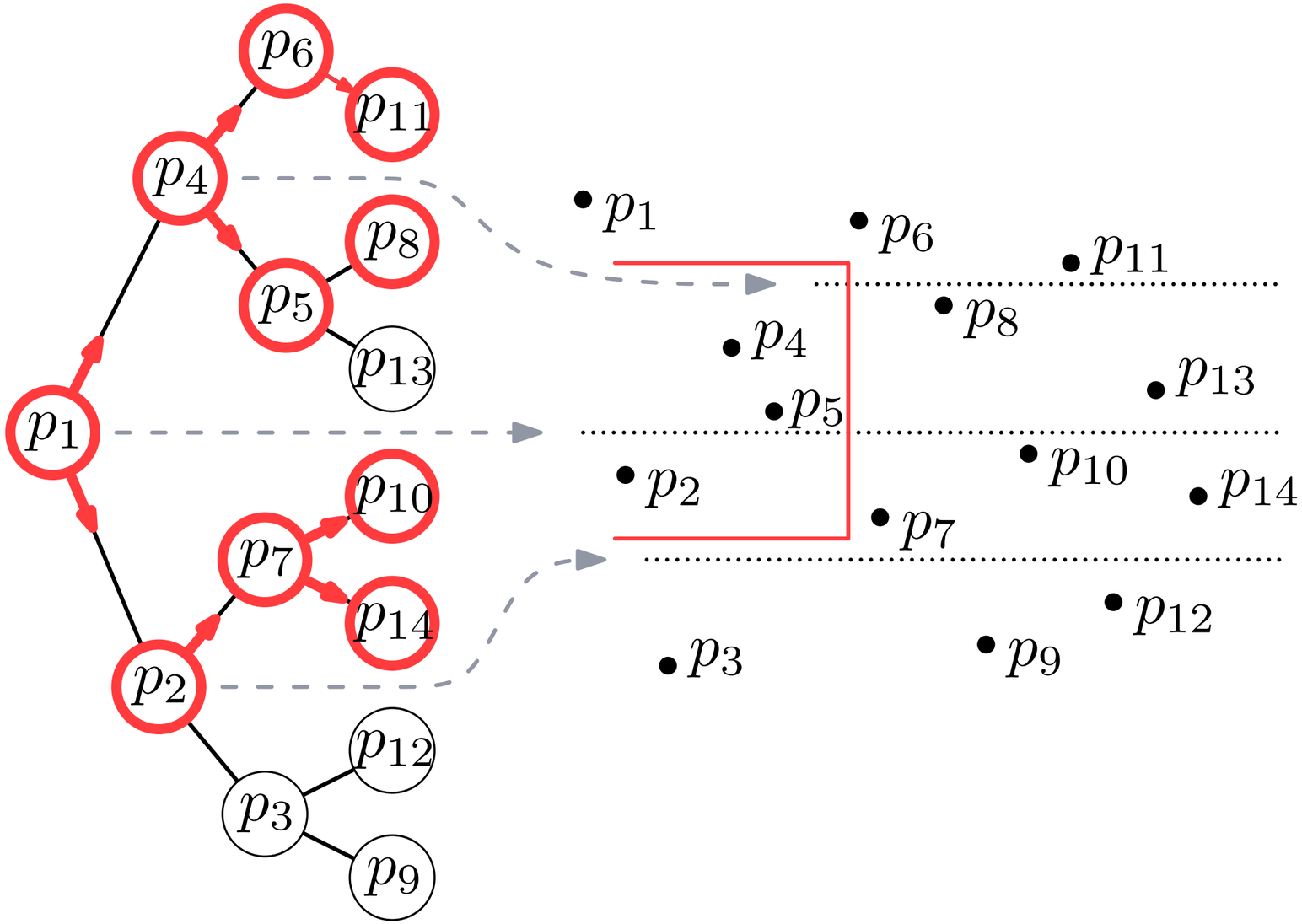
Priority search tree: query



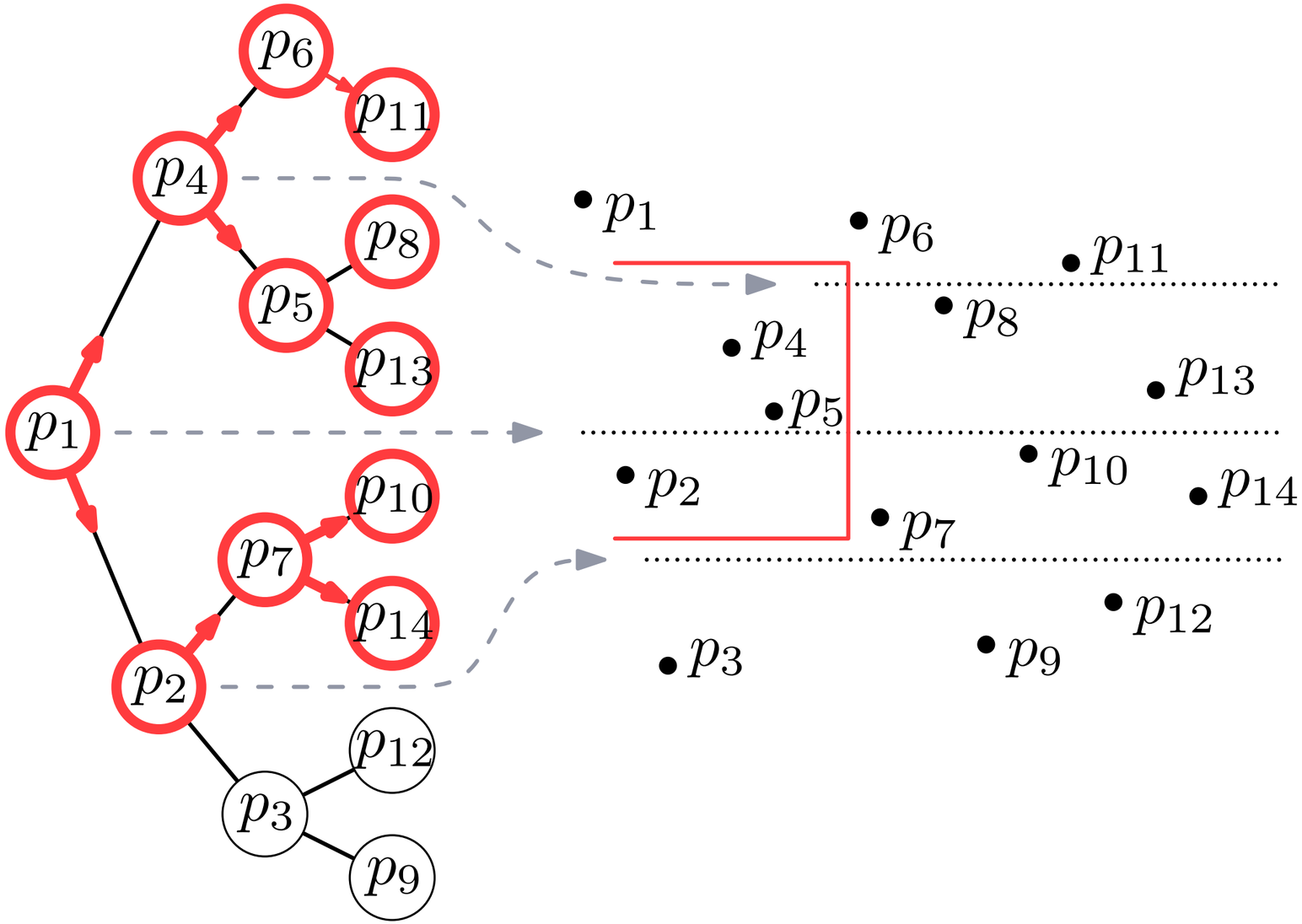
Priority search tree: query



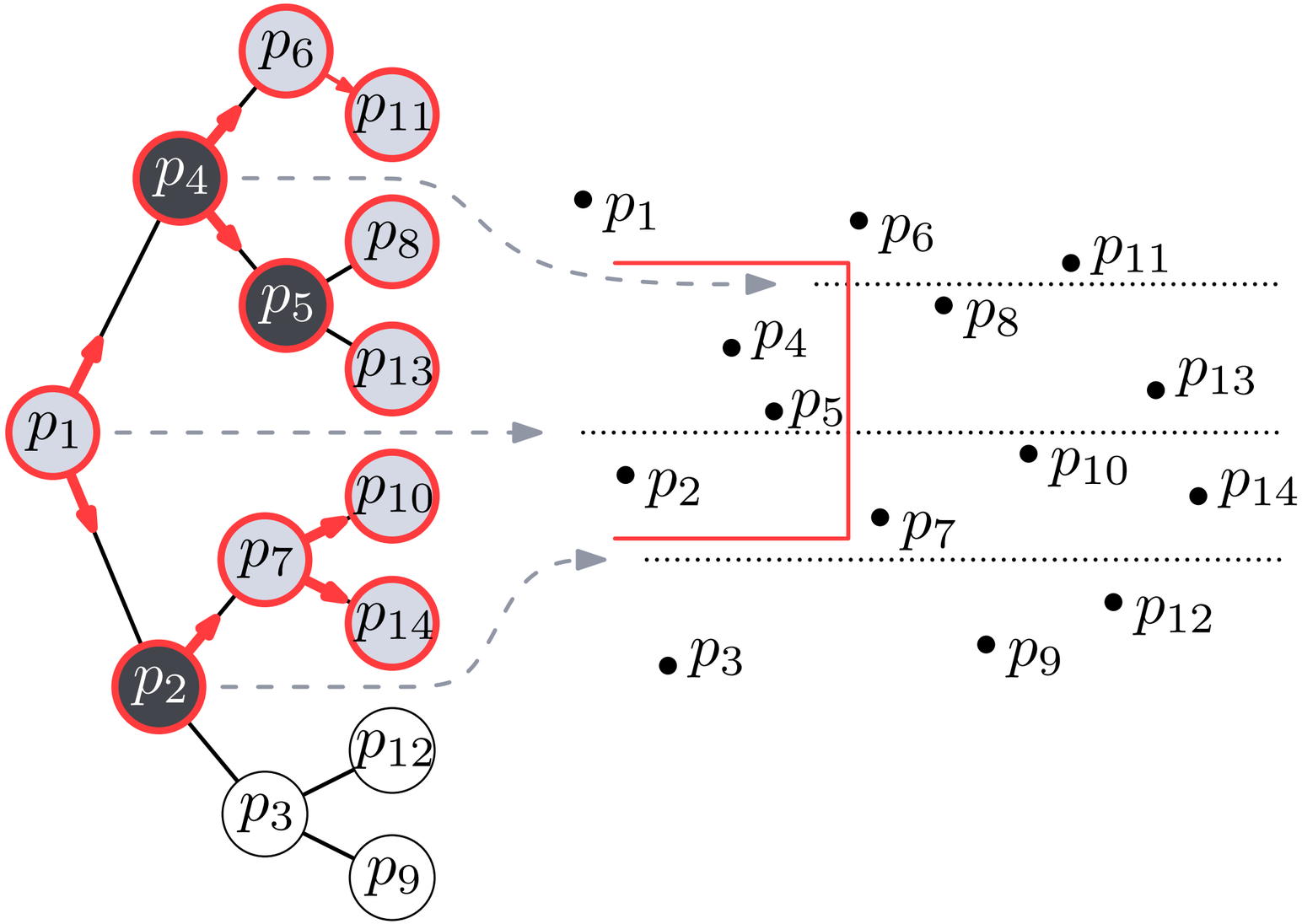
Priority search tree: query



Priority search tree: query



Priority search tree: query

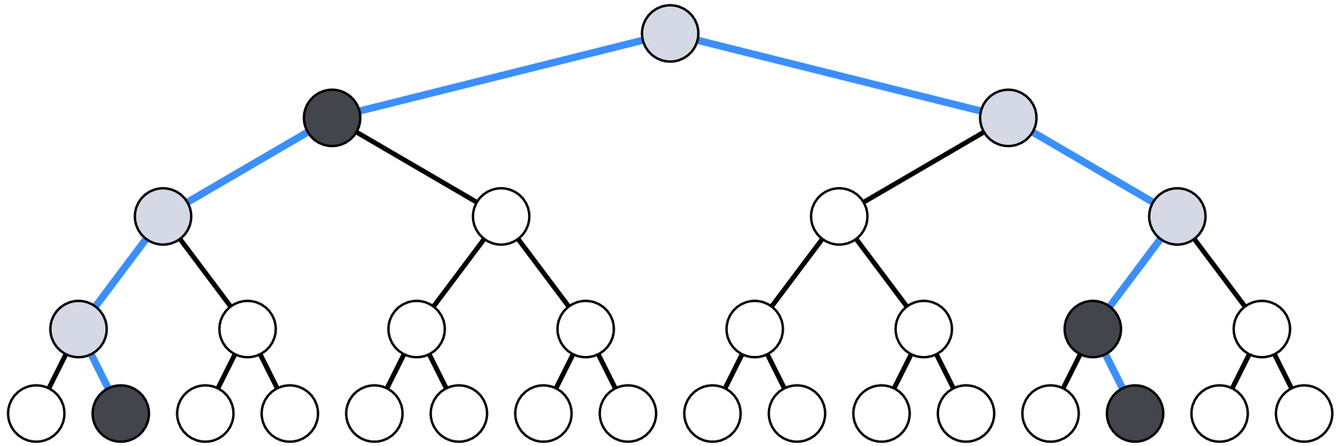


Query algorithm

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

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

Structure of the query



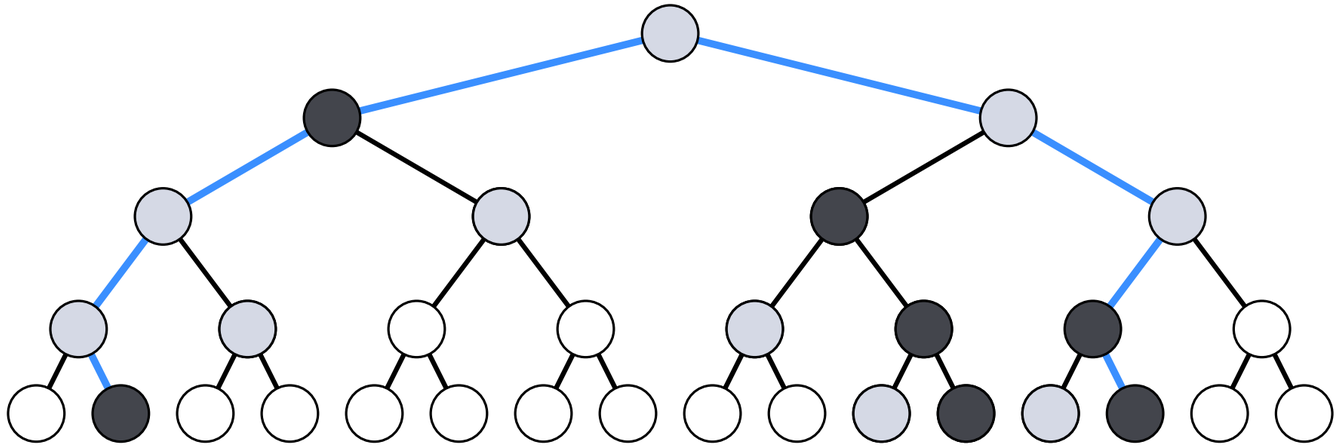
black: reported

grey: visited, not reported

white: not visited

1. Search path for horizontal boundaries: $O(\log n)$ visited, possible reported

Structure of the query



black: reported

grey: visited, not reported

white: not visited

1. Search path for horizontal boundaries: $O(\log n)$ visited, possible reported

2. between boundaries: stop when encountering a node
that is visited but not reported. $|\text{grey}| \leq 2\ |\text{black}|$

Query algorithm

REPORTINSUBTREE(v, q_x)

Input: The root v of a subtree of a priority search tree and a value q_x

Output: All points in the subtree with x-coordinate at most q_x

1: **if** v is not a leaf and $(p(v))_x \leq q_x$ **then**
2: Report $p(v)$
3: REPORTINSUBTREE$(lc(v), q_x)$
4: REPORTINSUBTREE$(rc(v), q_x)$

Query algorithm

ReportInSubtree(v, q_x)

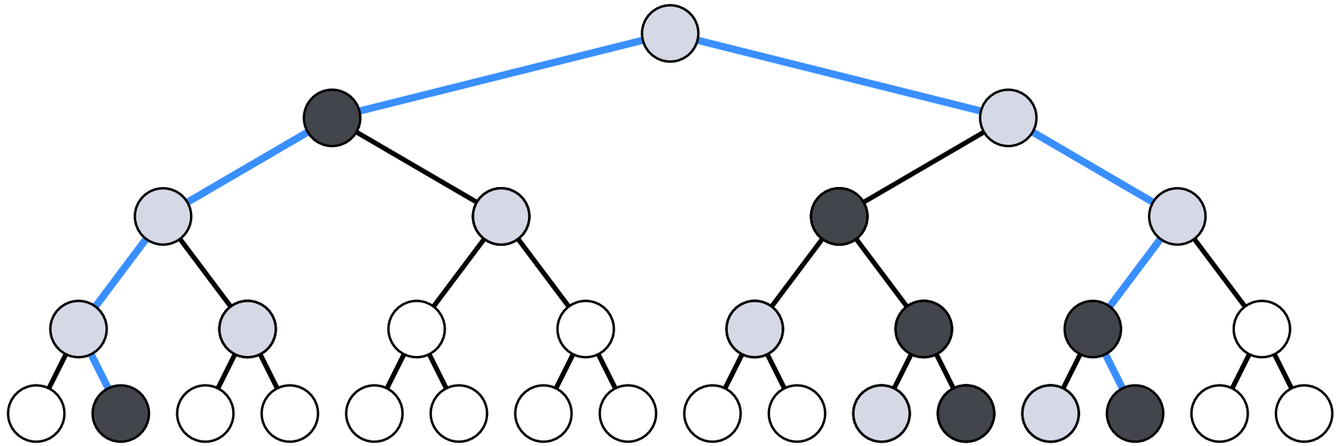
Input: The root v of a subtree of a priority search tree and a value q_x

Output: All points in the subtree with x-coordinate at most q_x

1: **if** v is not a leaf and $(p(v))_x \leq q_x$ **then**
2: Report $p(v)$
3: ReportInSubtree$(lc(v), q_x)$
4: ReportInSubtree$(rc(v), q_x)$

This subroutine takes $O(1 + k)$ time, for k reported answers

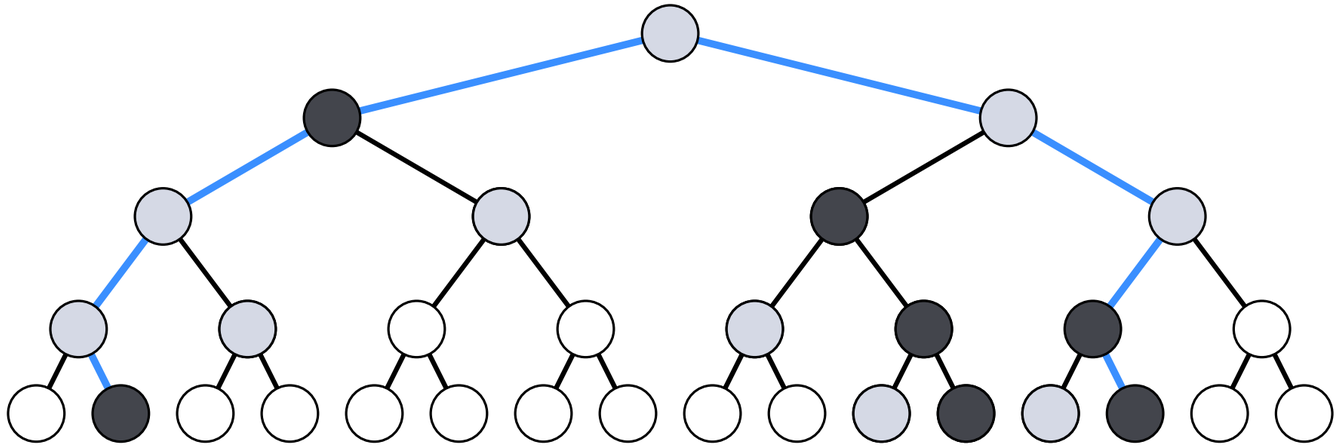
Query algorithm



The search paths to y and y' have $O(\log n)$ nodes.

At each node $O(1)$ time is spent

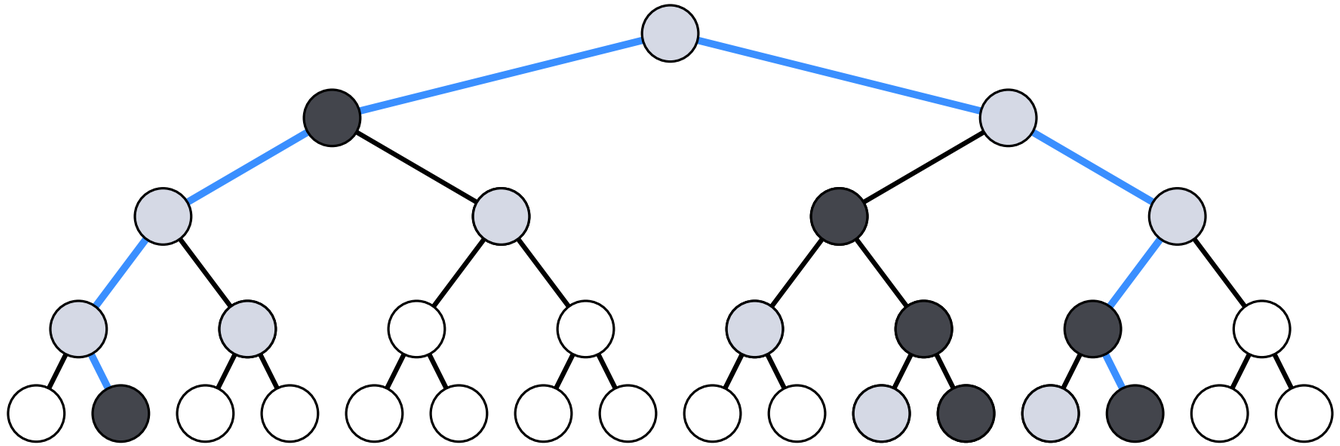
Query algorithm



The search paths to y and y' have $O(\log n)$ nodes.
At each node $O(1)$ time is spent

No nodes outside the search paths are ever visited

Query algorithm

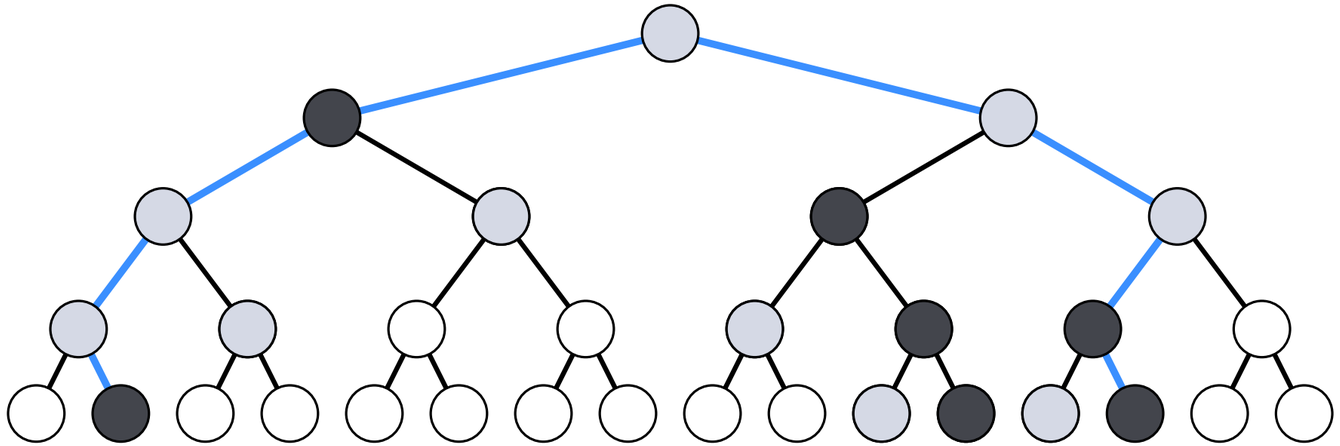


The search paths to y and y' have $O(\log n)$ nodes.
At each node $O(1)$ time is spent

No nodes outside the search paths are ever visited

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

Query algorithm



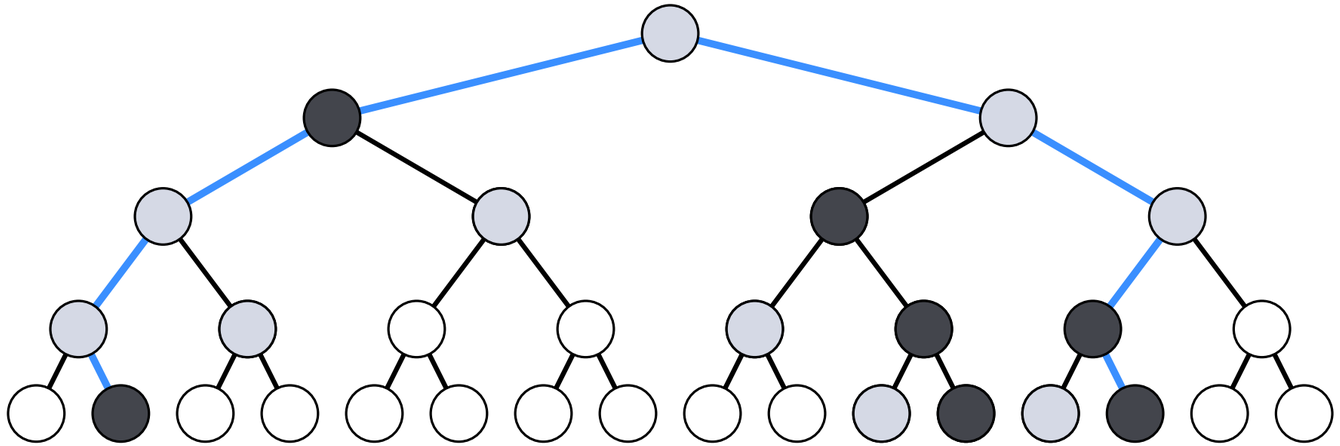
The search paths to y and y' have $O(\log n)$ nodes.
At each node $O(1)$ time is spent

No nodes outside the search paths are ever visited

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

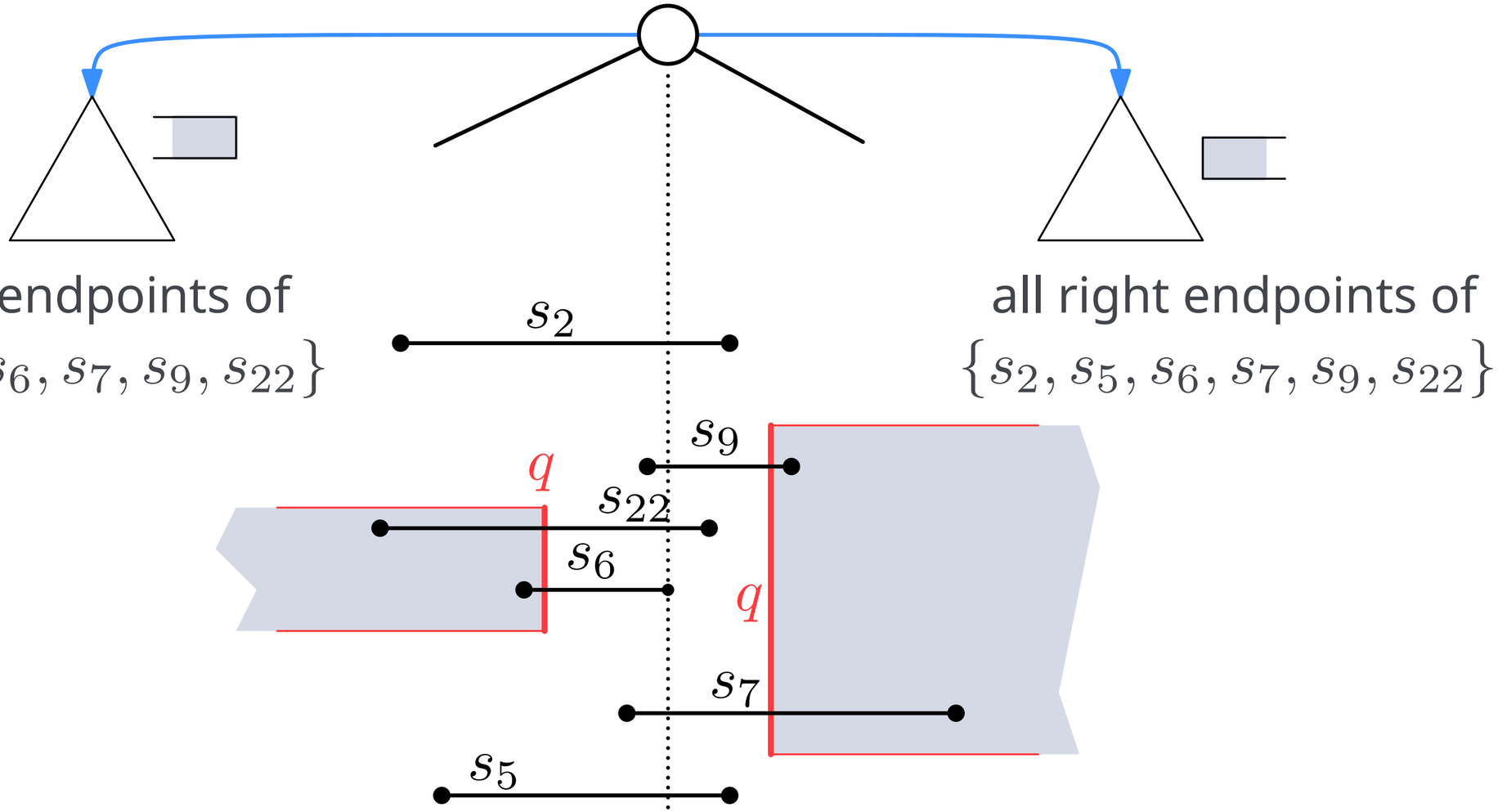
The total query time is $O(\log n + k)$, if k points are reported

Priority search tree: result



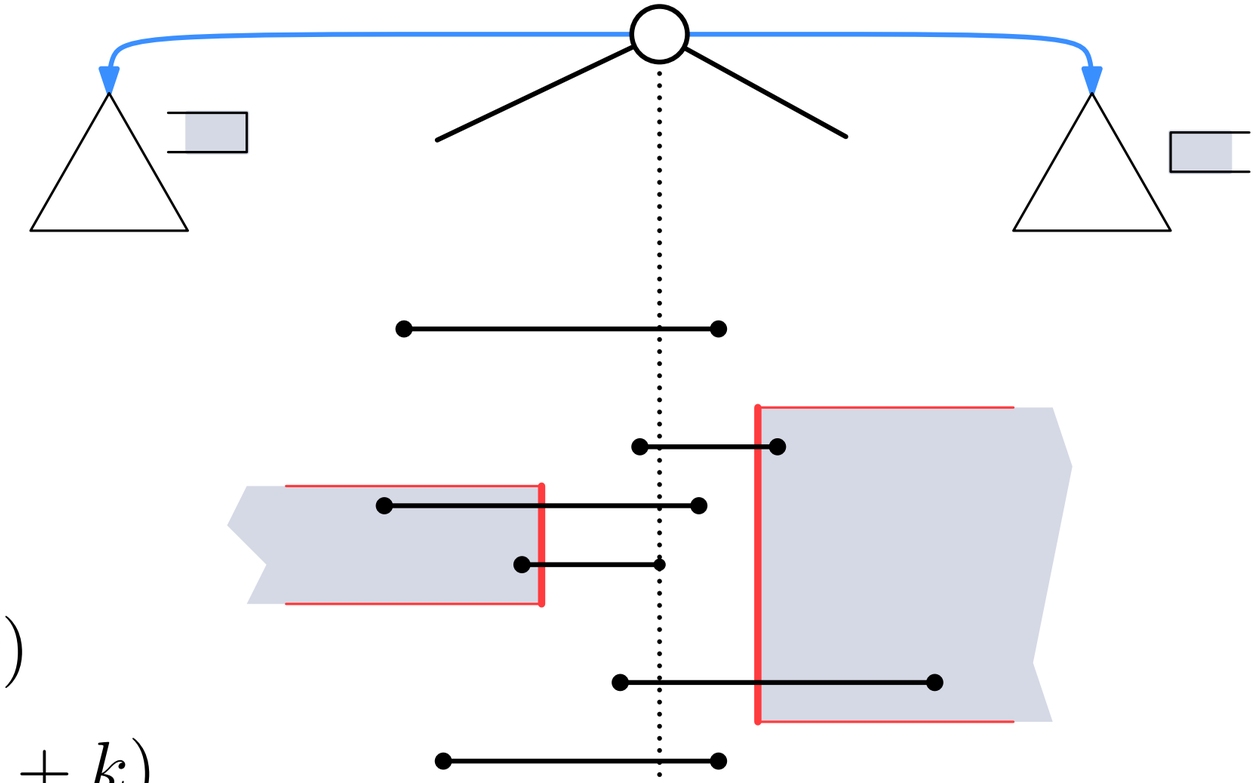
Theorem: A priority search tree for a set P of n points uses $O(n)$ storage and can be built in $O(n \log n)$ time. All points that lie in a 3-sided query range can be reported in $O(\log n + k)$ time, where k is the number of reported points.

Scheme of structure



Quiz

Question: What are the storage requirements and query time for this structure?



A: $O(n)$ and $O(\log n + k)$

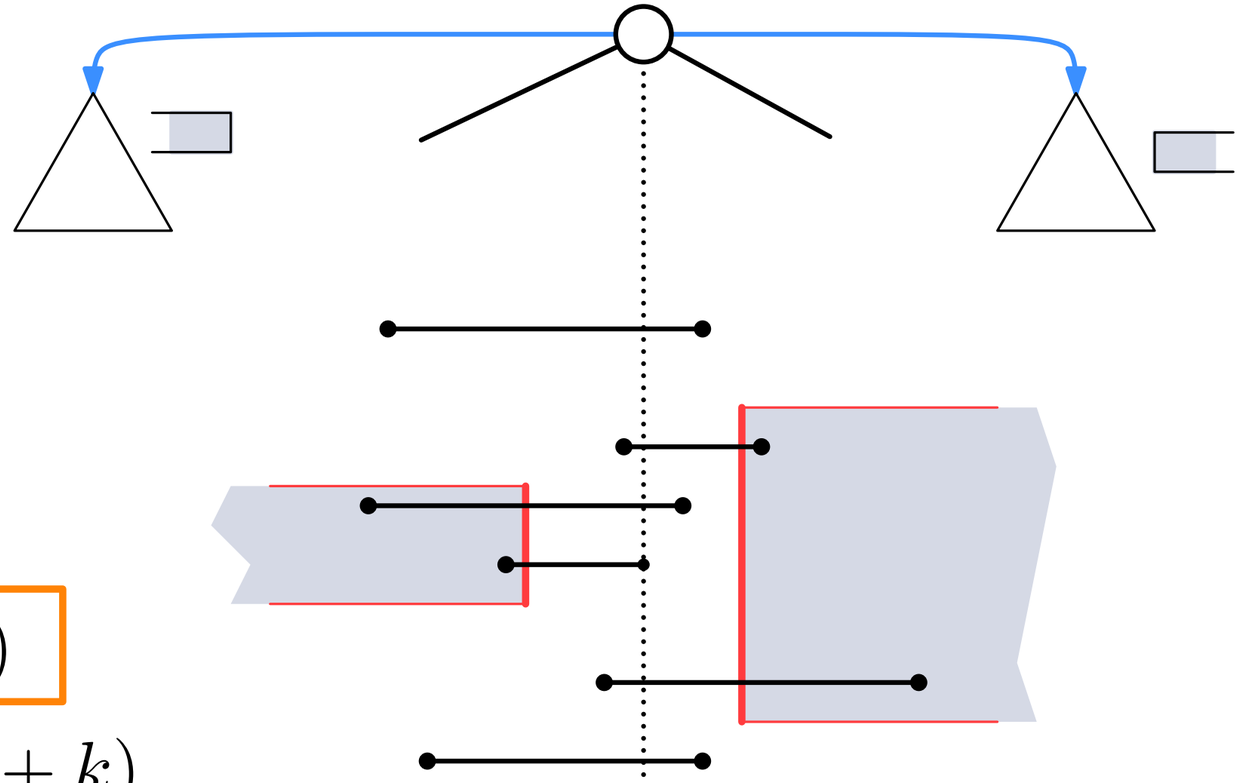
B: $O(n)$ and $O(\log^2 n + k)$

C: $O(n \log n)$ and $O(\log n + k)$

D: $O(n \log n)$ and $O(\log^2 n + k)$

Quiz

Question: What are the storage requirements and query time for this structure?



A: $O(n)$ and $O(\log n + k)$

B: $O(n)$ and $O(\log^2 n + k)$

C: $O(n \log n)$ and $O(\log n + k)$

D: $O(n \log n)$ and $O(\log^2 n + k)$

Result

Theorem: A set of n horizontal line segments can be stored in a data structure with size $O(n)$ such that intersection queries with a vertical line segment can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported.

Result

Theorem: A set of n horizontal line segments can be stored in a data structure with size $O(n)$ such that intersection queries with a vertical line segment can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported.

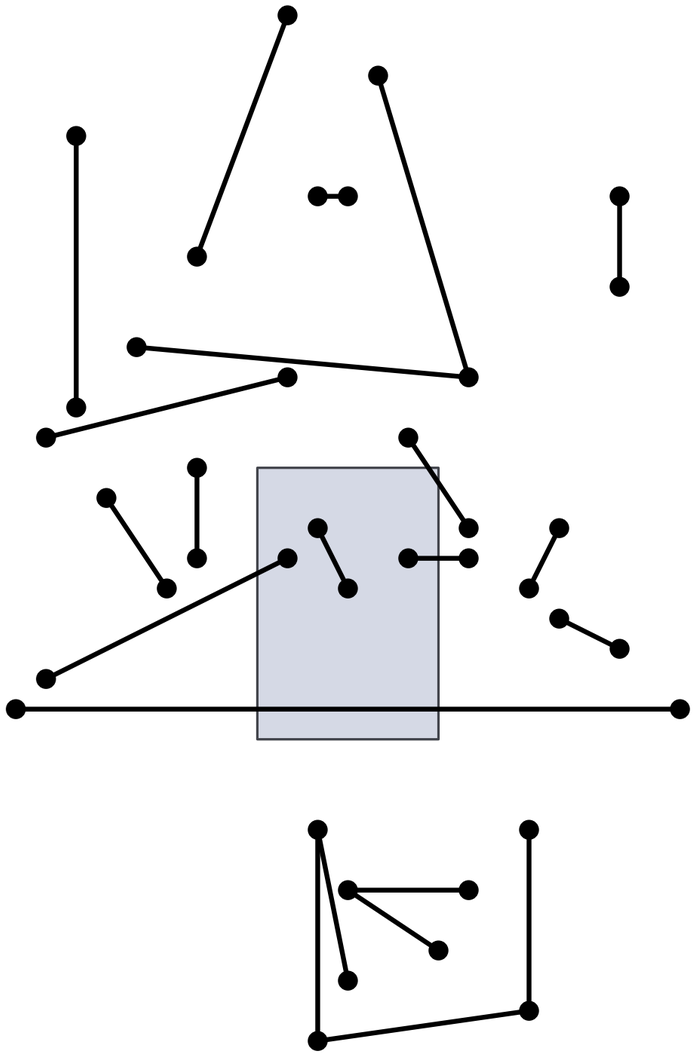
Recall that the windowing problem is solved with a combination of a range tree and the structure just described

Theorem: A set of n axis-parallel line segments can be stored in a data structure with size $O(n \log n)$ such that windowing queries can be performed in $O(\log^2 n + k)$ time, where k is the number of segments reported.

Range and Windowing Queries

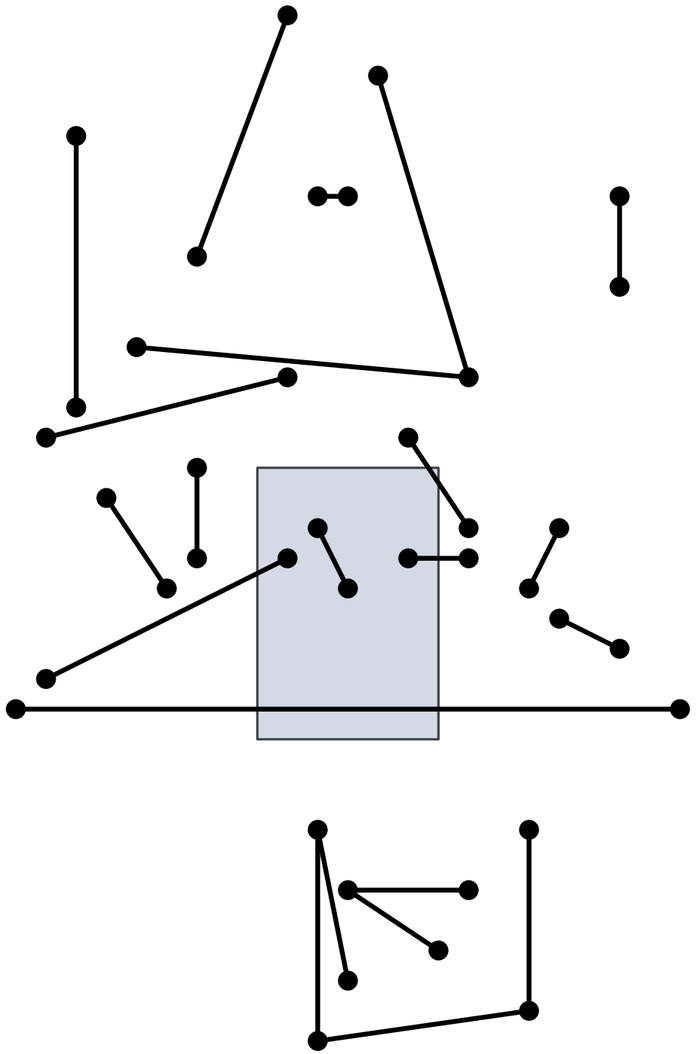
Windowing Queries for arbitrary, non-intersecting segments

Windowing



Given a set of n arbitrary, non-crossing line segments, can we preprocess them into a data structure so that the ones that intersect a query rectangle can be reported efficiently?

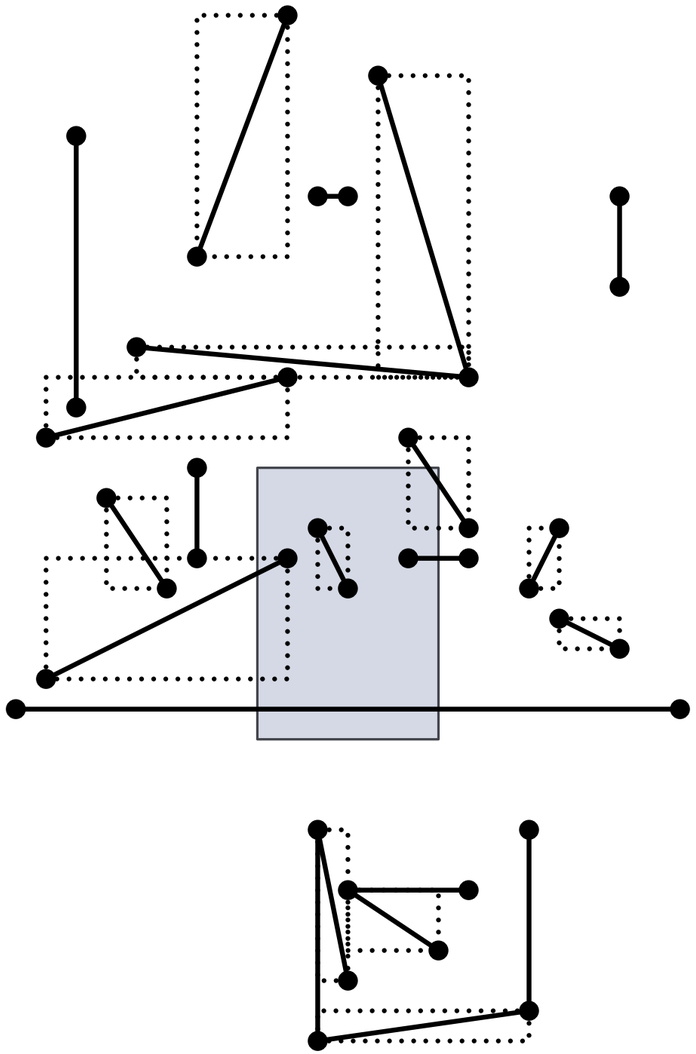
Windowing



Two cases of intersection:

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

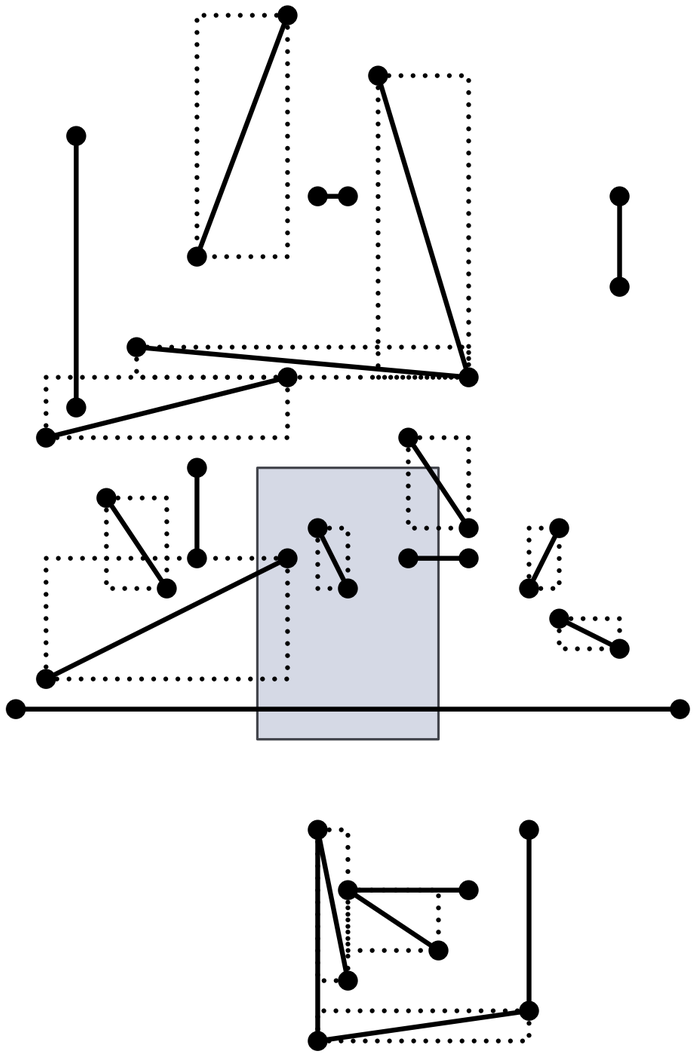
Using a bounding box?



If the query window intersects the line segment, then it also intersects the bounding box of the line segment (whose sides are axis-parallel segments)

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

Using a bounding box?



If the query window intersects the line segment, then it also intersects the bounding box of the line segment (whose sides are axis-parallel segments)

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

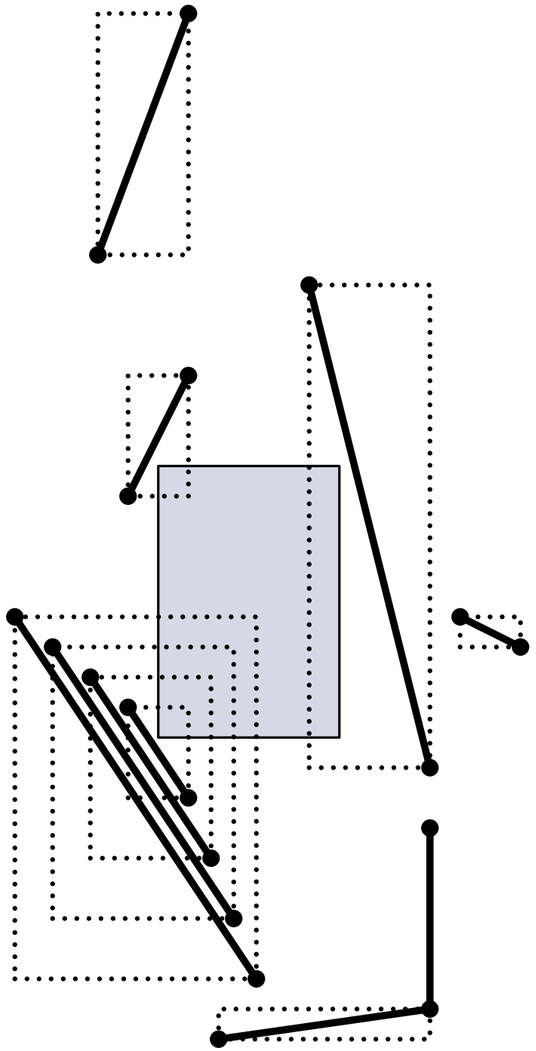
What is the main problem with this approach?

A: A segment might not intersect the query rectangle, while its bounding box does

B: Vertical segments don’t have bounding boxes

C: There are too many bounding boxes

Using a bounding box?



If the query window intersects the line segment, then it also intersects the bounding box of the line segment (whose sides are axis-parallel segments)

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

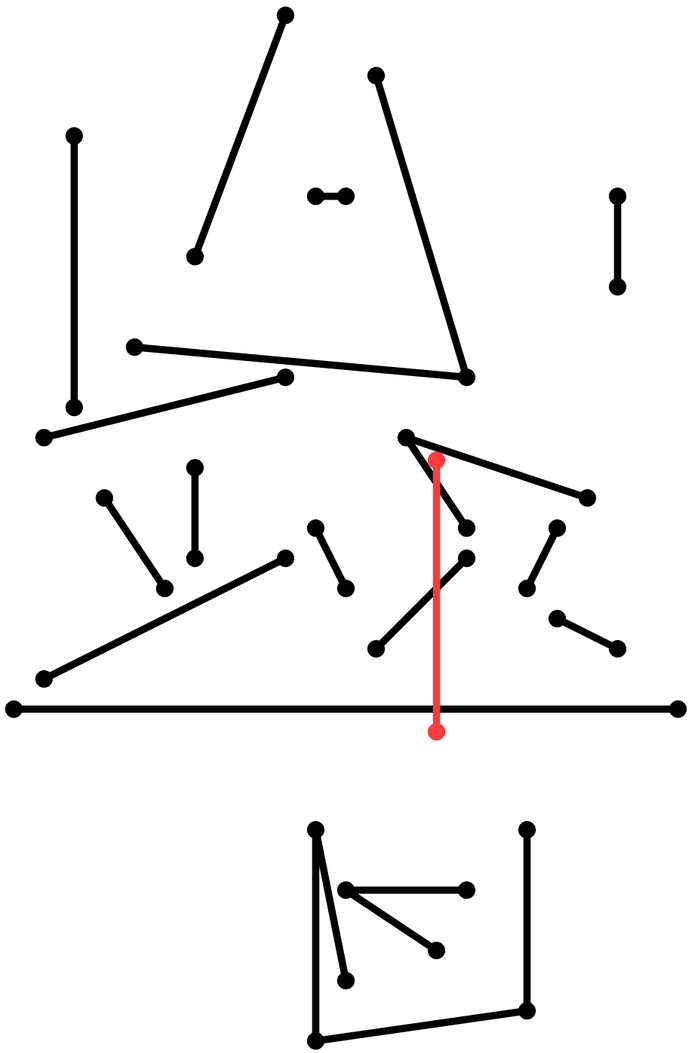
What is the main problem with this approach?

A: A segment might not intersect the query rectangle, while its bounding box does

B: Vertical segments don't have bounding boxes

C: There are too many bounding boxes

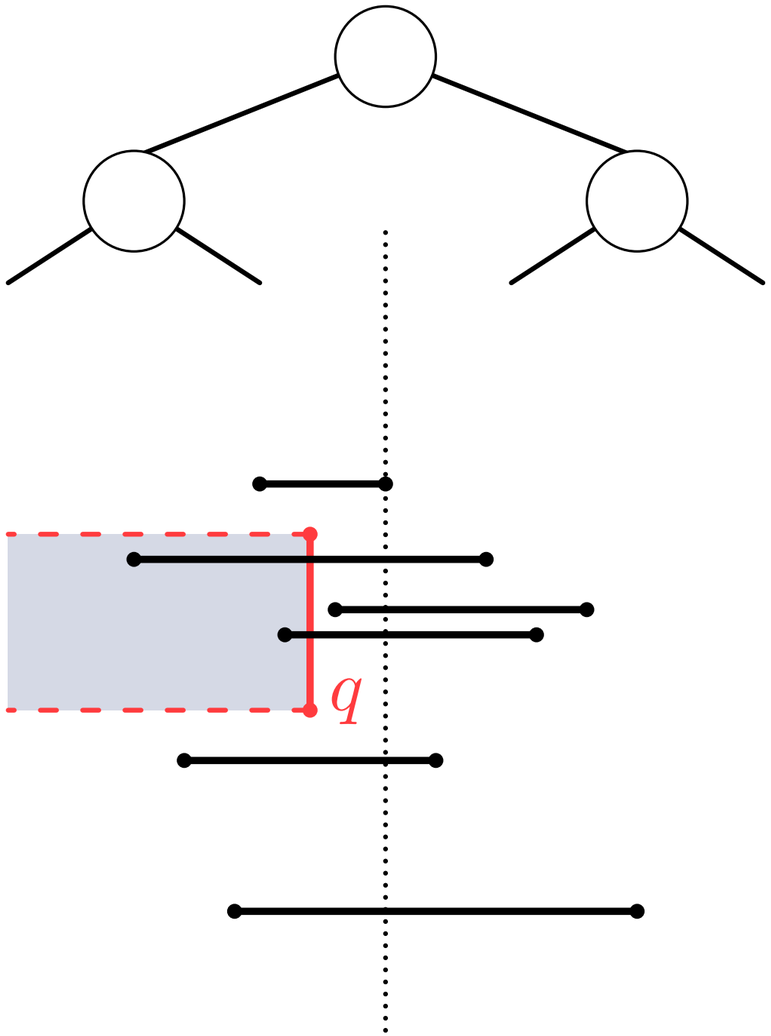
Windowing

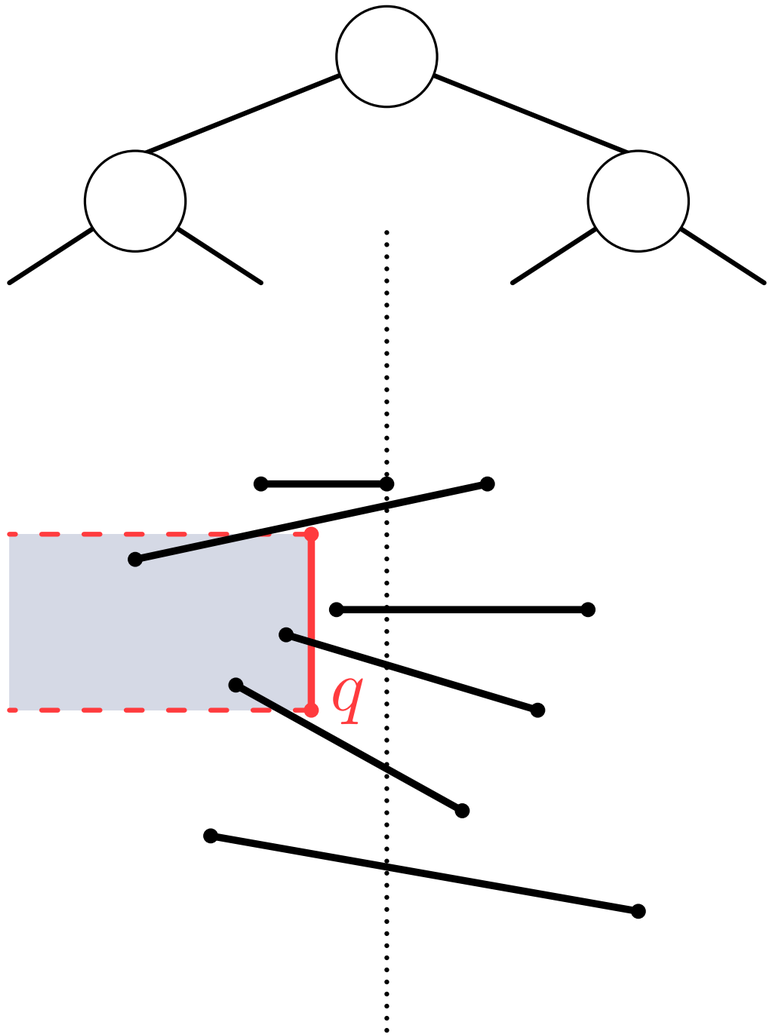


Current problem of our interest:

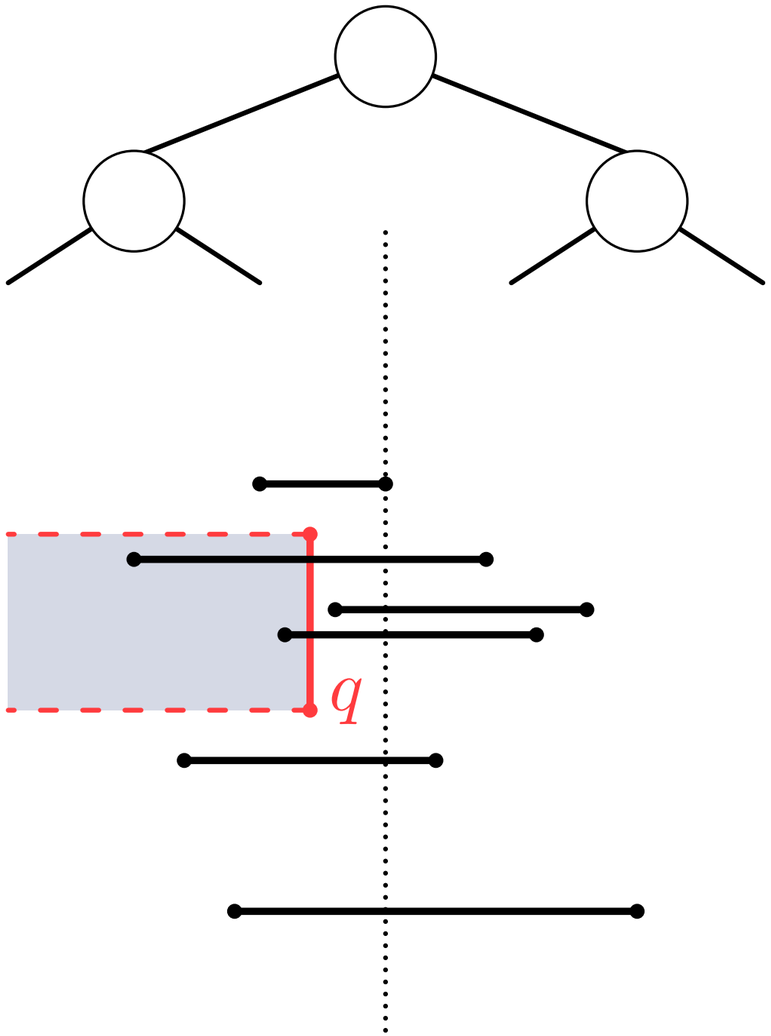
Given a set of arbitrarily oriented, non-crossing line segments, preprocess them into a data structure so that the ones intersecting a vertical (horizontal) query segment can be reported efficiently

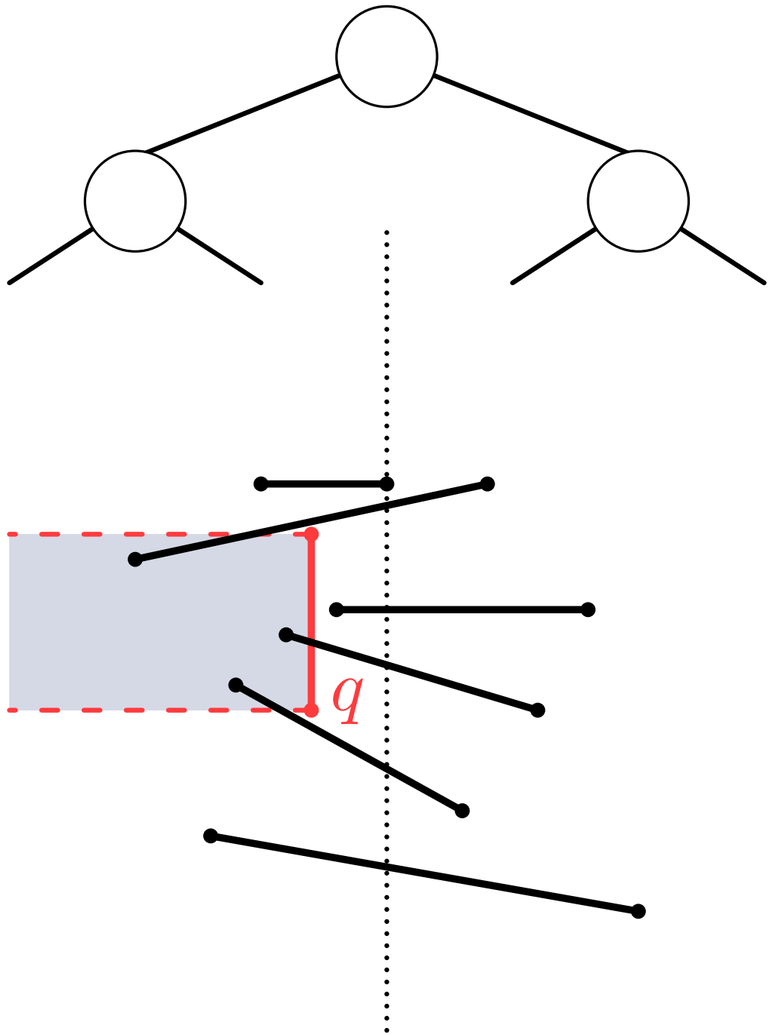
Using an interval tree?





Using an interval tree?





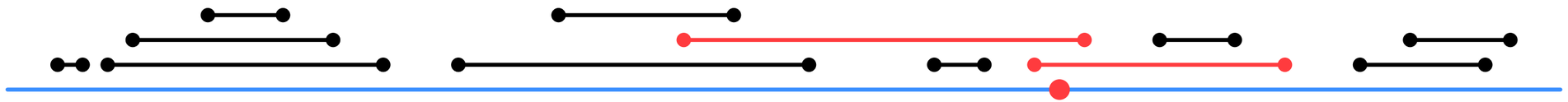
next: segment trees

Range and Windowing Queries

Segment trees: Locus approach

Interval querying

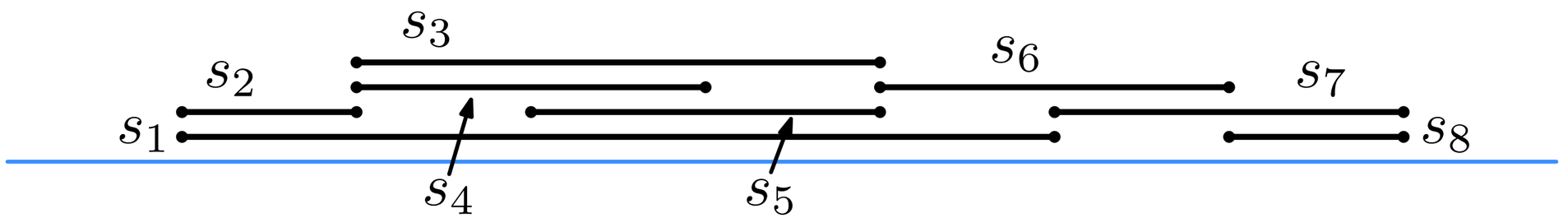
Given a set I of n intervals on the real line, preprocess them into a data structure so that the ones containing a query point (value) can be reported efficiently



We have the interval tree, but we will develop an alternative solution

Locus approach

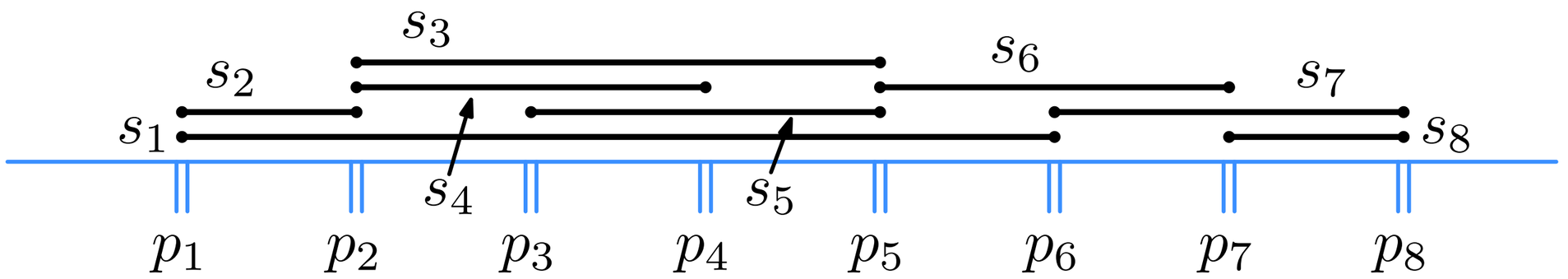
The locus approach is the idea to partition the solution space into parts with equal answer sets



For the set S of segments, we get different answer sets before and after every endpoint

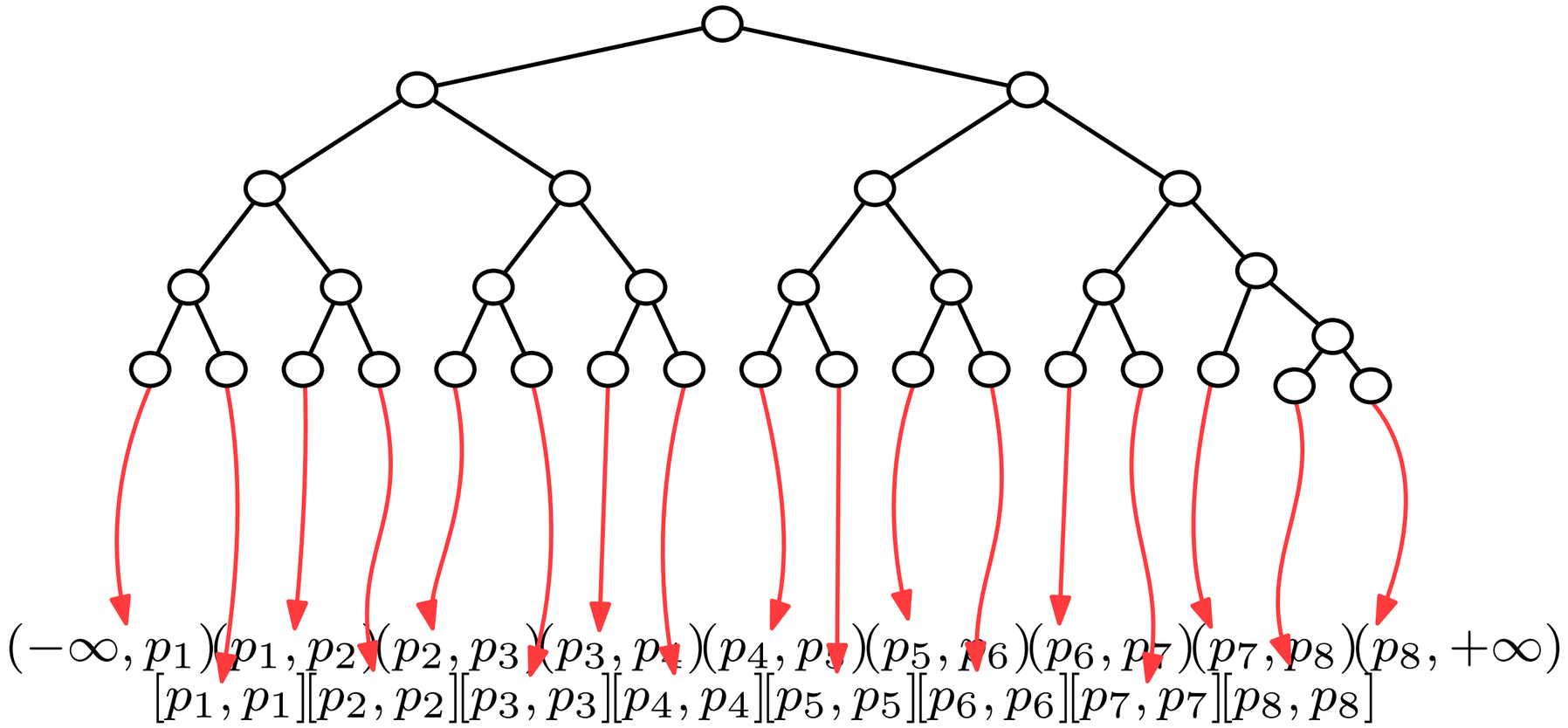
Locus approach

Let $p_1, p_2, \ldots, p_m$ be the sorted set of unique endpoints of the intervals; $m \leq 2n$



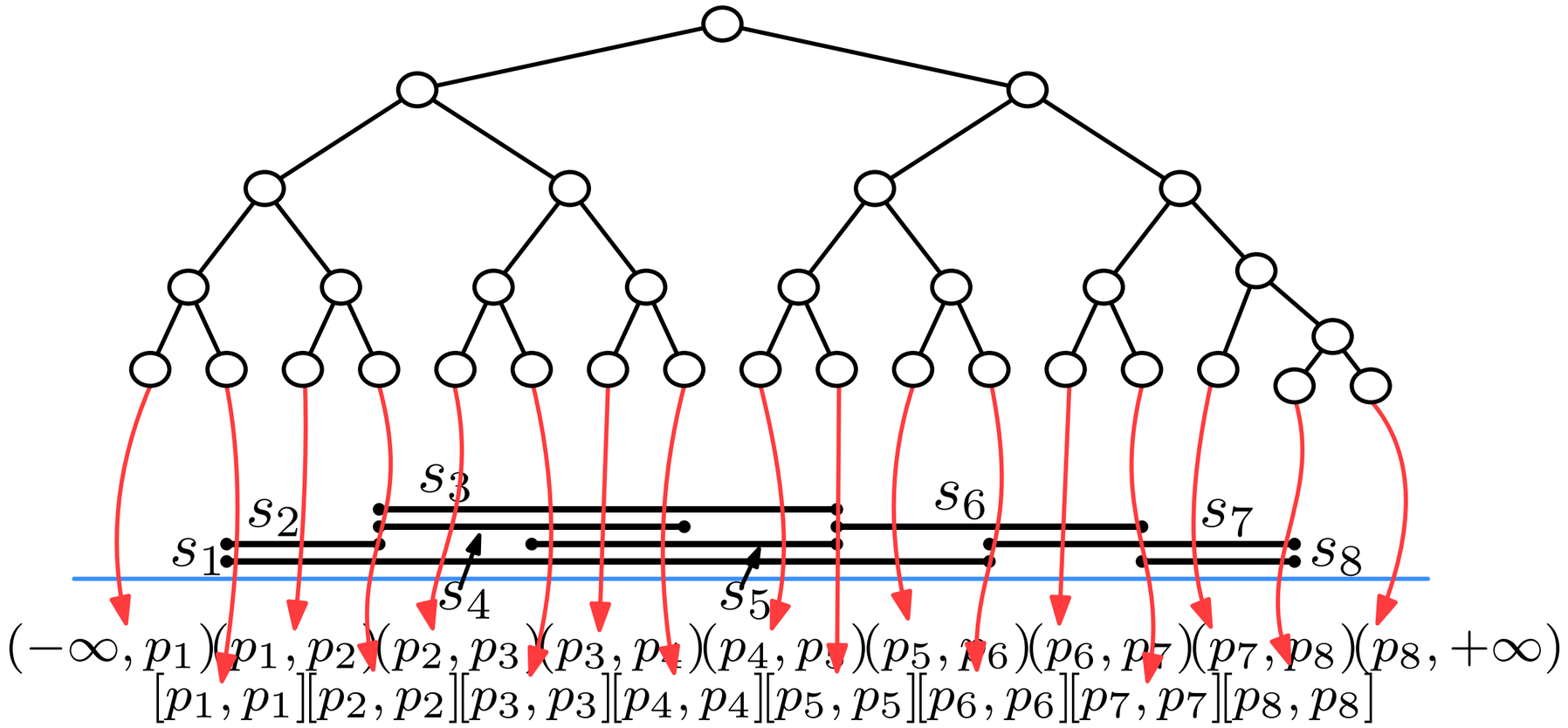
The real line is partitioned into $(-\infty, p_1)$, $[p_1, p_1]$, (p_1, p_2), $[p_2, p_2]$, (p_2, p_3), …, $(p_m, +\infty)$, these are called the elementary intervals

Locus approach



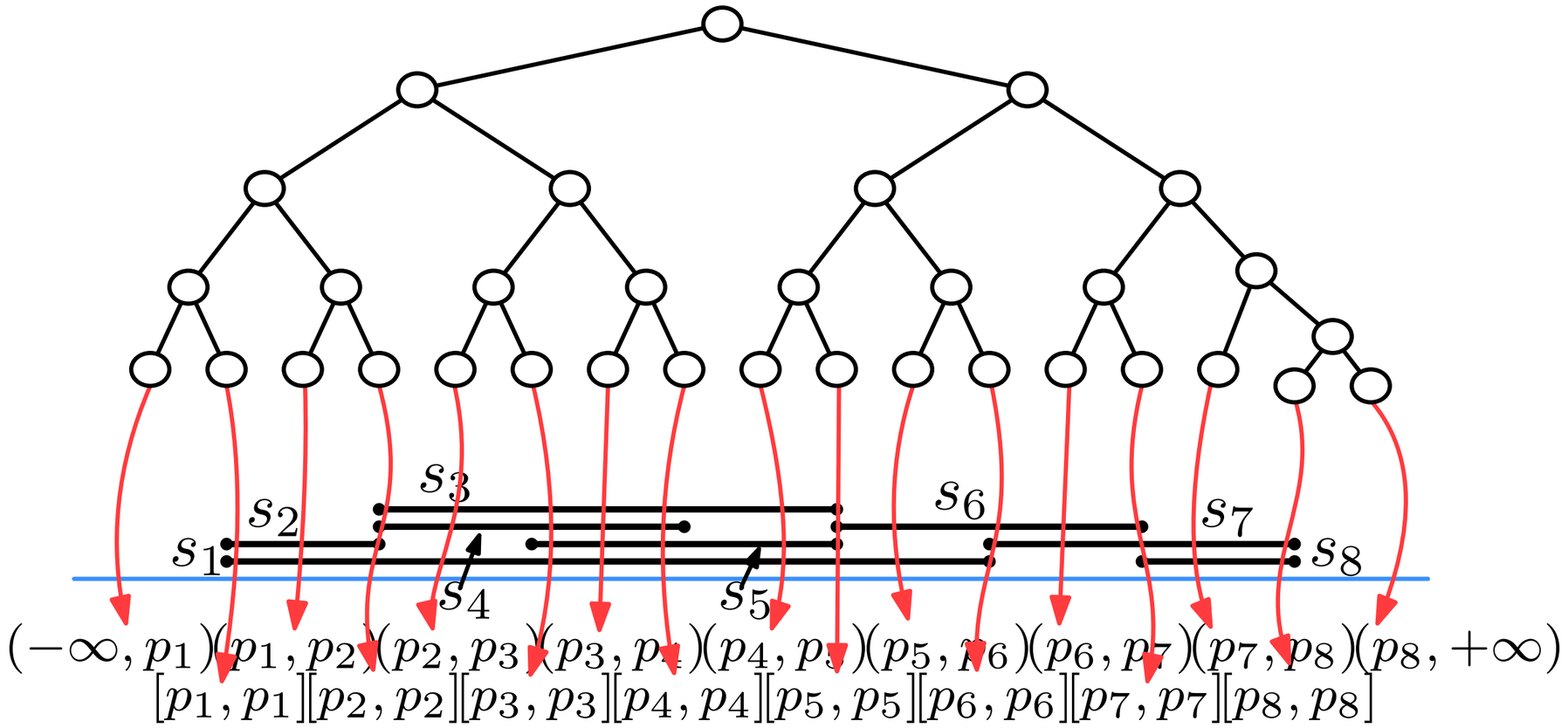
We could make a binary search tree that has a leaf for every elementary interval $(-\infty, p_1)$, $[p_1, p_1]$, (p_1, p_2), $[p_2, p_2]$, (p_2, p_3), …, $(p_m, +\infty)$

Locus approach



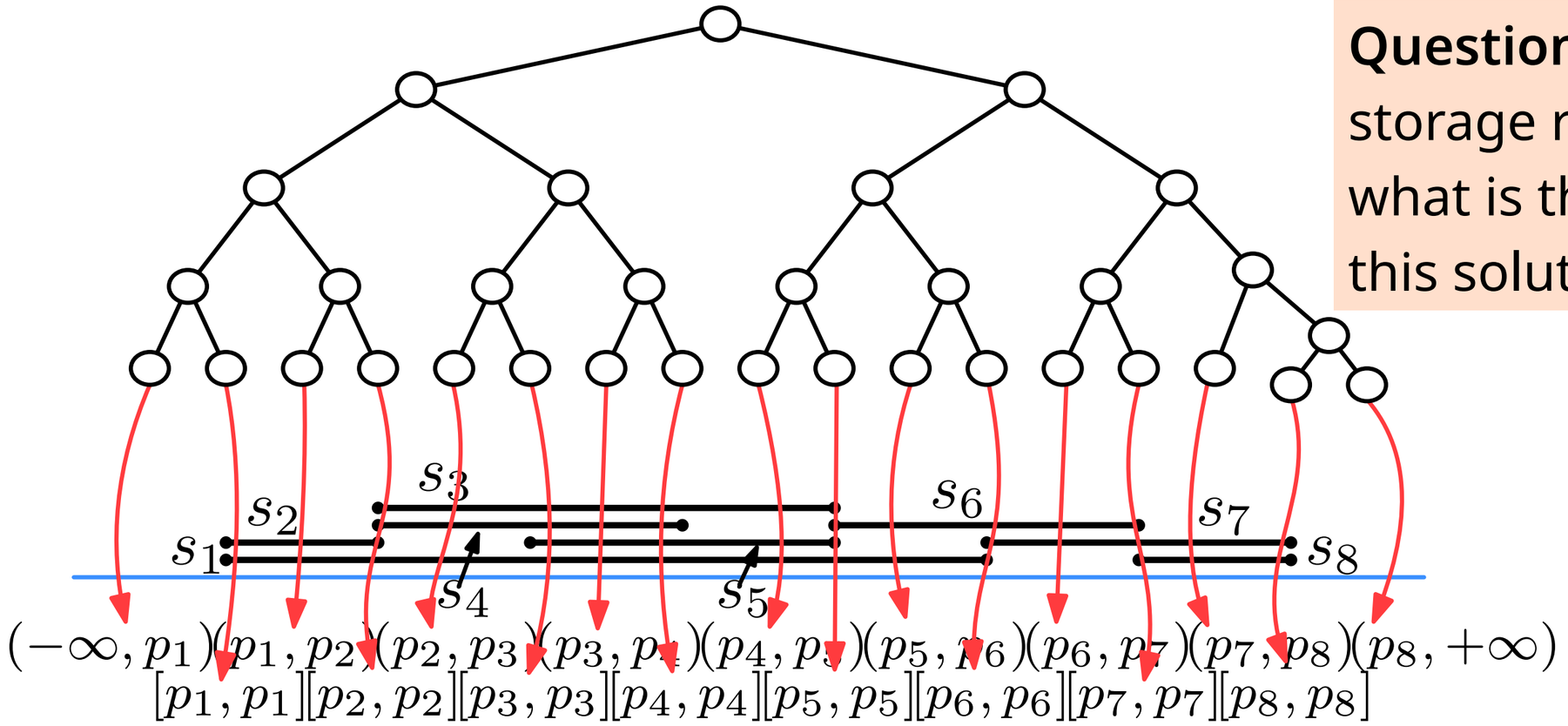
Each segment from the set S can be stored with all leaves whose elementary interval it contains: $[p_i, p_j]$ is stored with $[p_i, p_i]$, (p_i, p_{i+1}), …, $[p_j, p_j]$

Locus approach



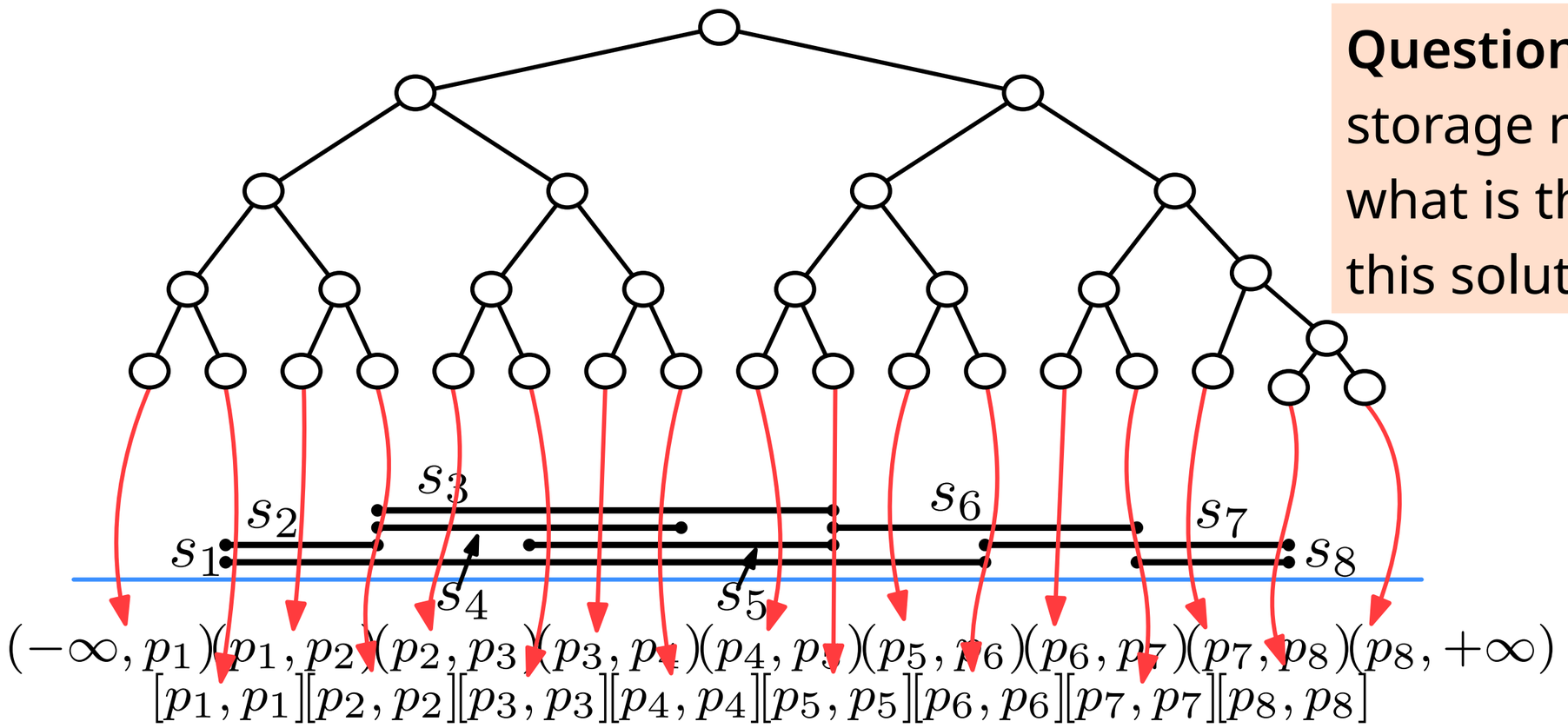
A stabbing query with point q is then solved by finding the unique leaf that contains q, and reporting all segments that it stores

Locus approach



Question: What are the storage requirements and what is the query time of this solution?

Locus approach



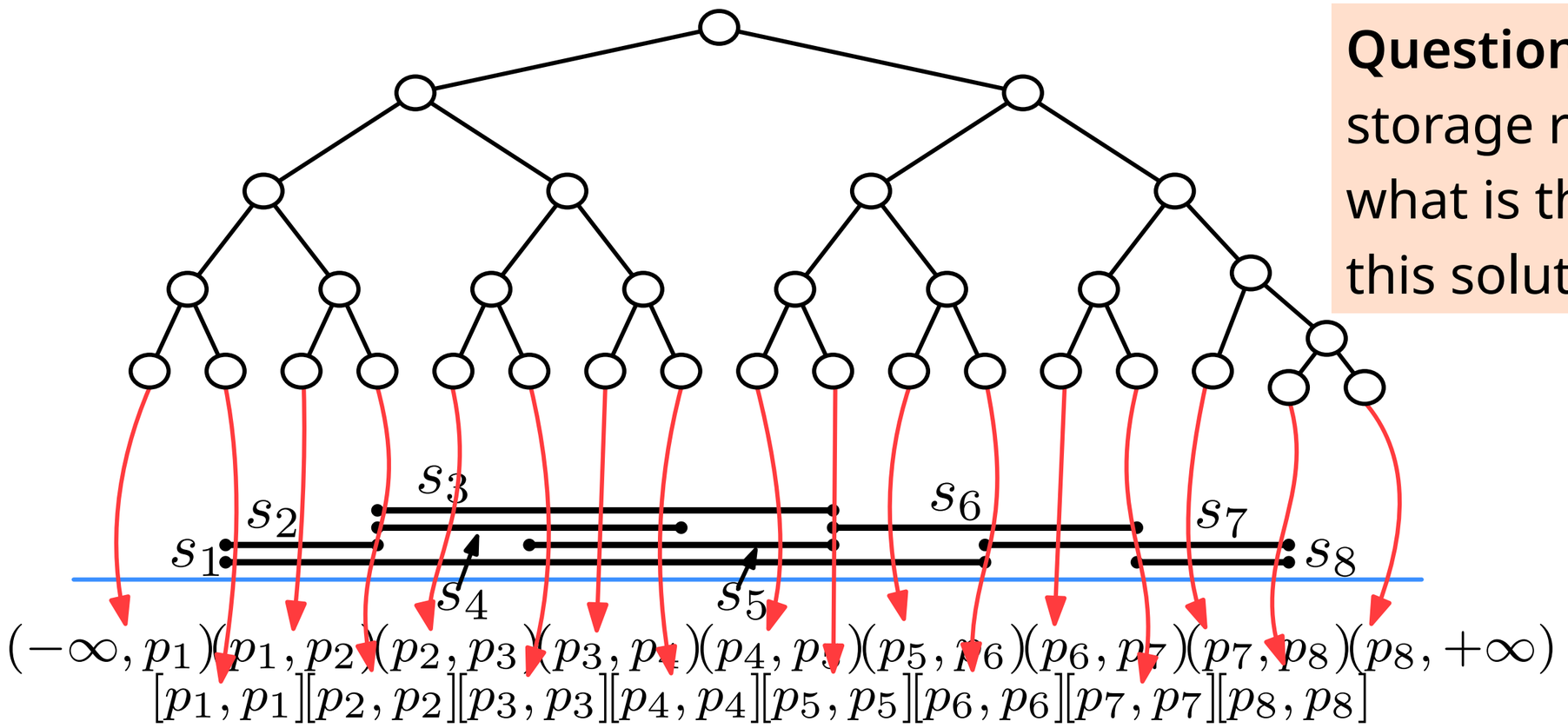
Question: What are the storage requirements and what is the query time of this solution?

A: $O(n)$ and $O(\log n + k)$

B: $O(n \log n)$ and $O(\log^2 n + k)$

C: $O(n^2)$ and $O(\log n + k)$

Locus approach



Question: What are the storage requirements and what is the query time of this solution?

A: $O(n)$ and $O(\log n + k)$

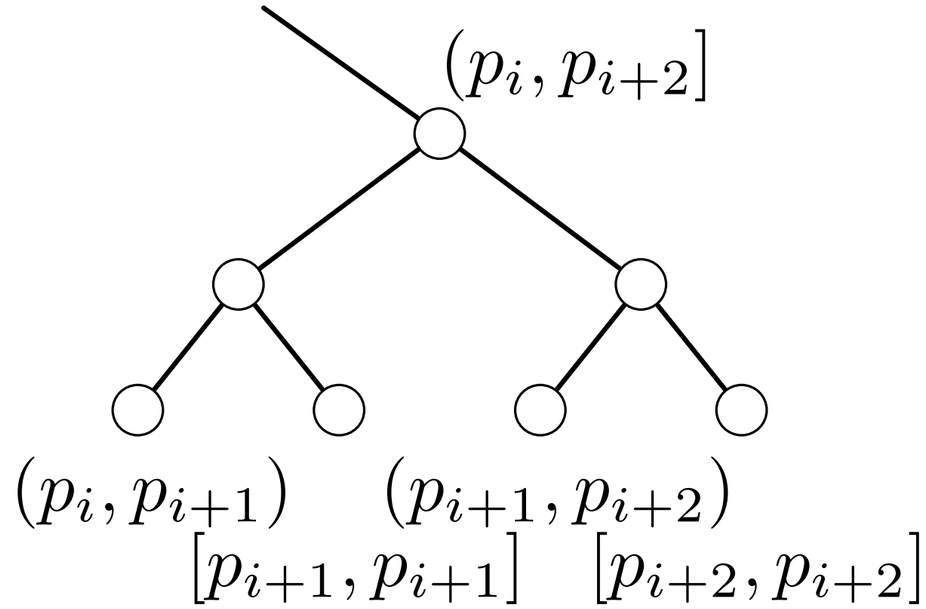
B: $O(n \log n)$ and $O(\log^2 n + k)$

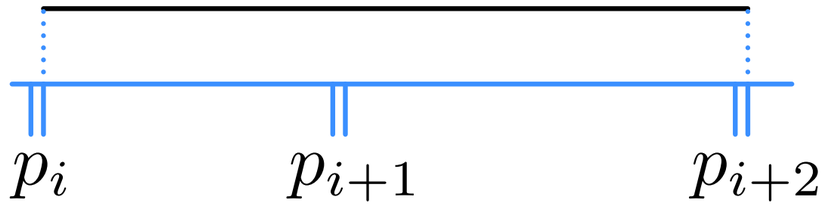
C: $O(n^2)$ and $O(\log n + k)$

Towards segment trees

In the tree, the leaves store elementary intervals

But each internal node corresponds to an interval too: the interval that is the union of the elementary intervals of all leaves below it

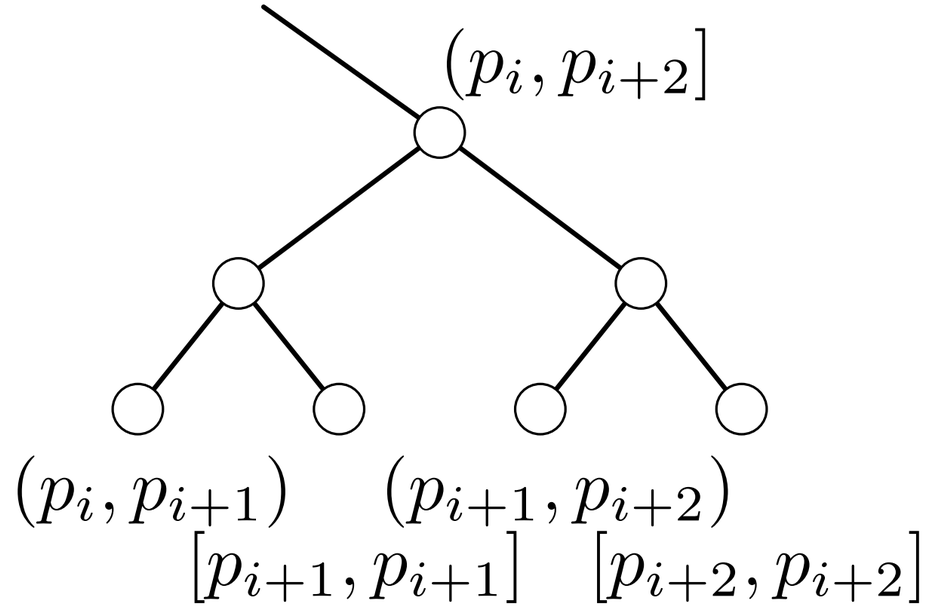


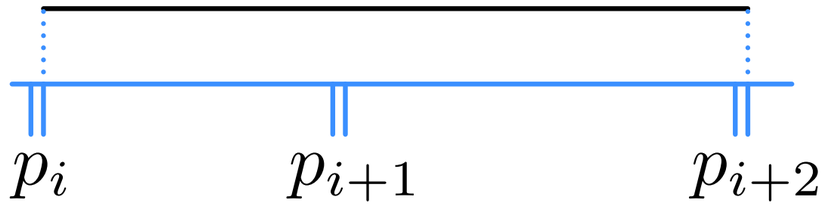


Towards segment trees

In the tree, the leaves store elementary intervals

But each internal node corresponds to an interval too: the interval that is the union of the elementary intervals of all leaves below it



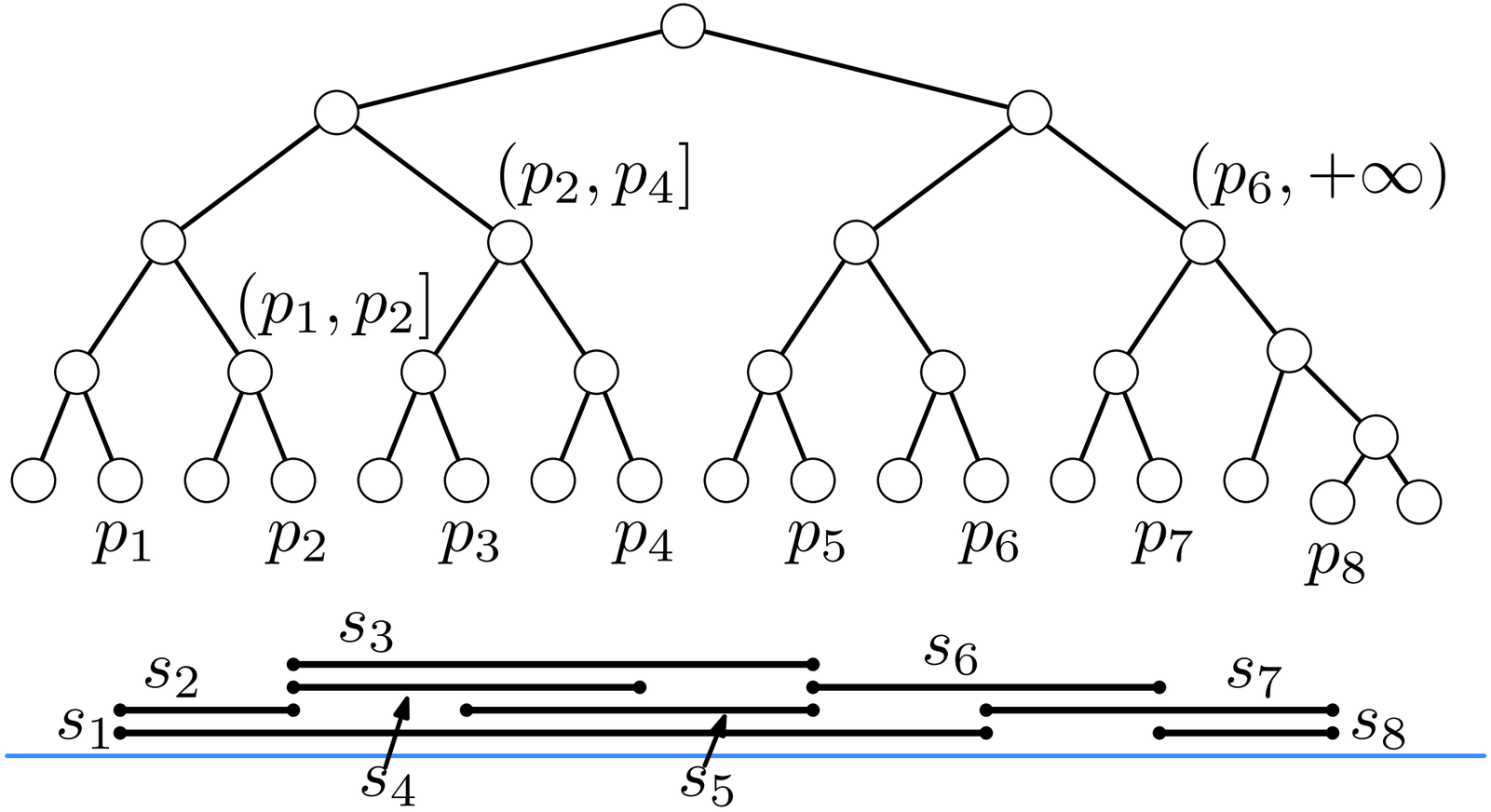


next: How can we use this to save storage?

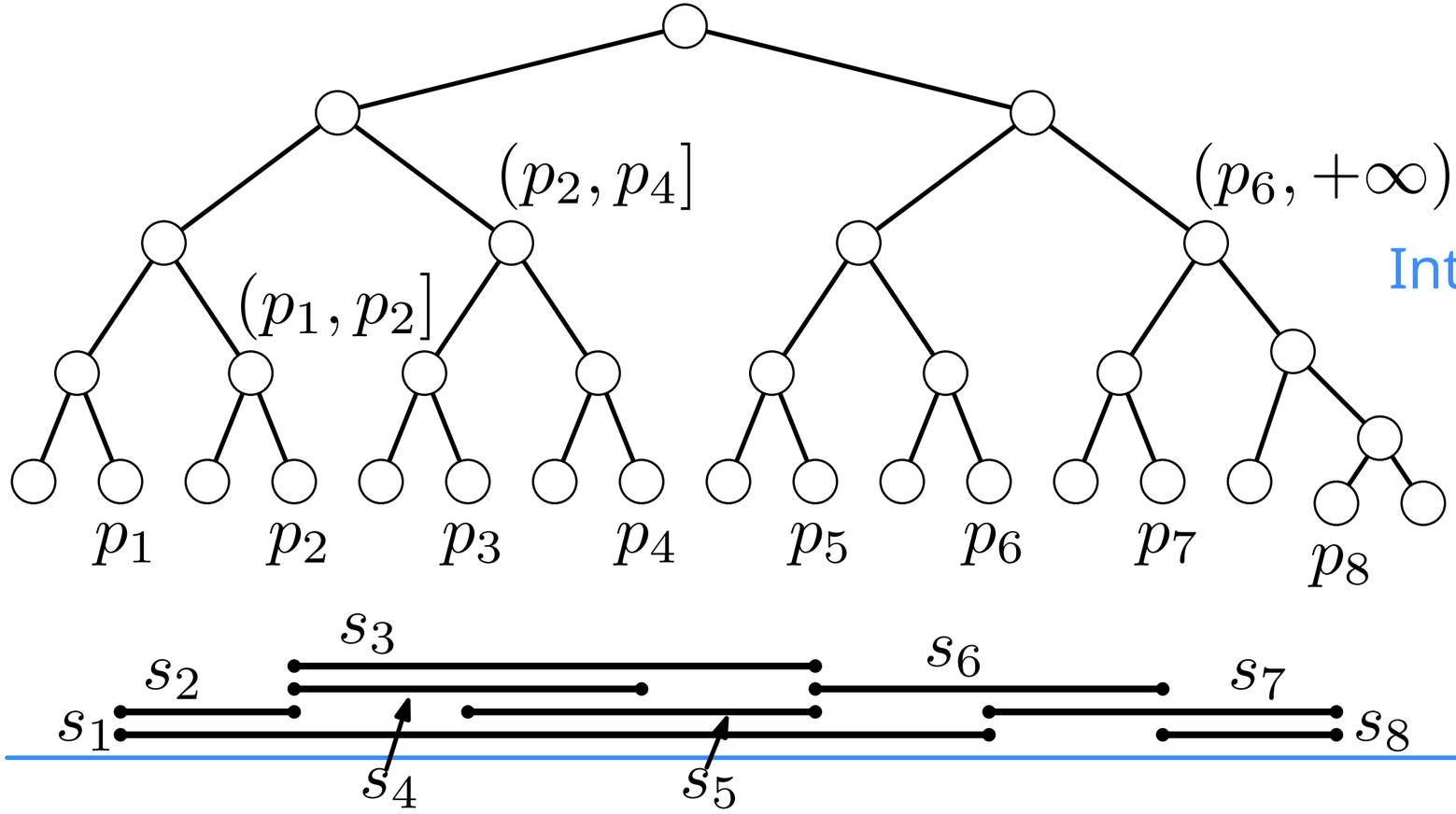
Range and Windowing Queries

Segment trees: Canonical intervals

Towards segment trees

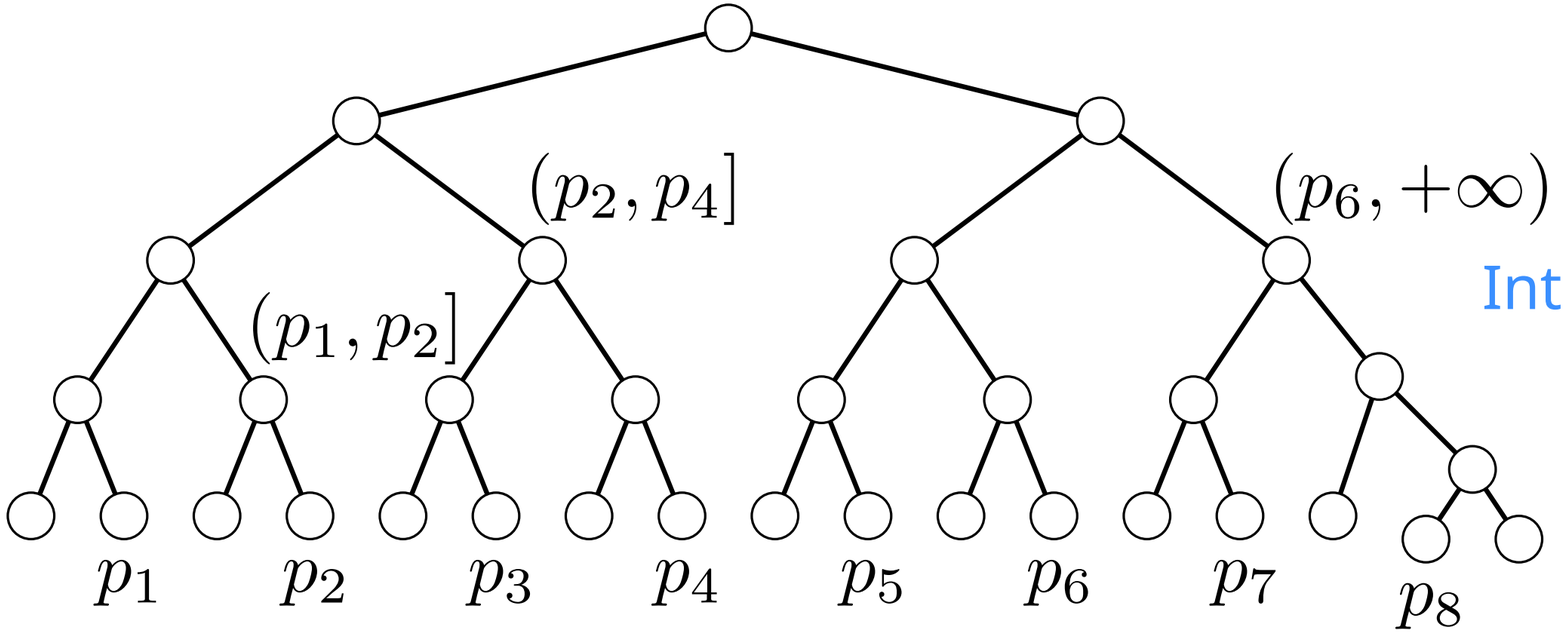


Towards segment trees

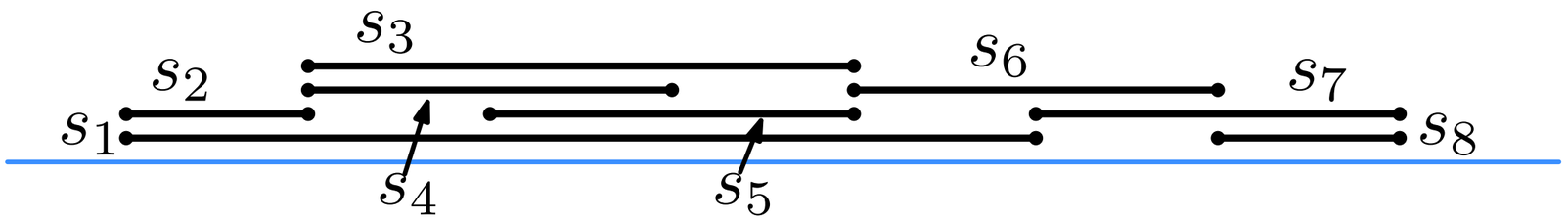


$\mathrm{Int}(v)$: interval of node v

Towards segment trees

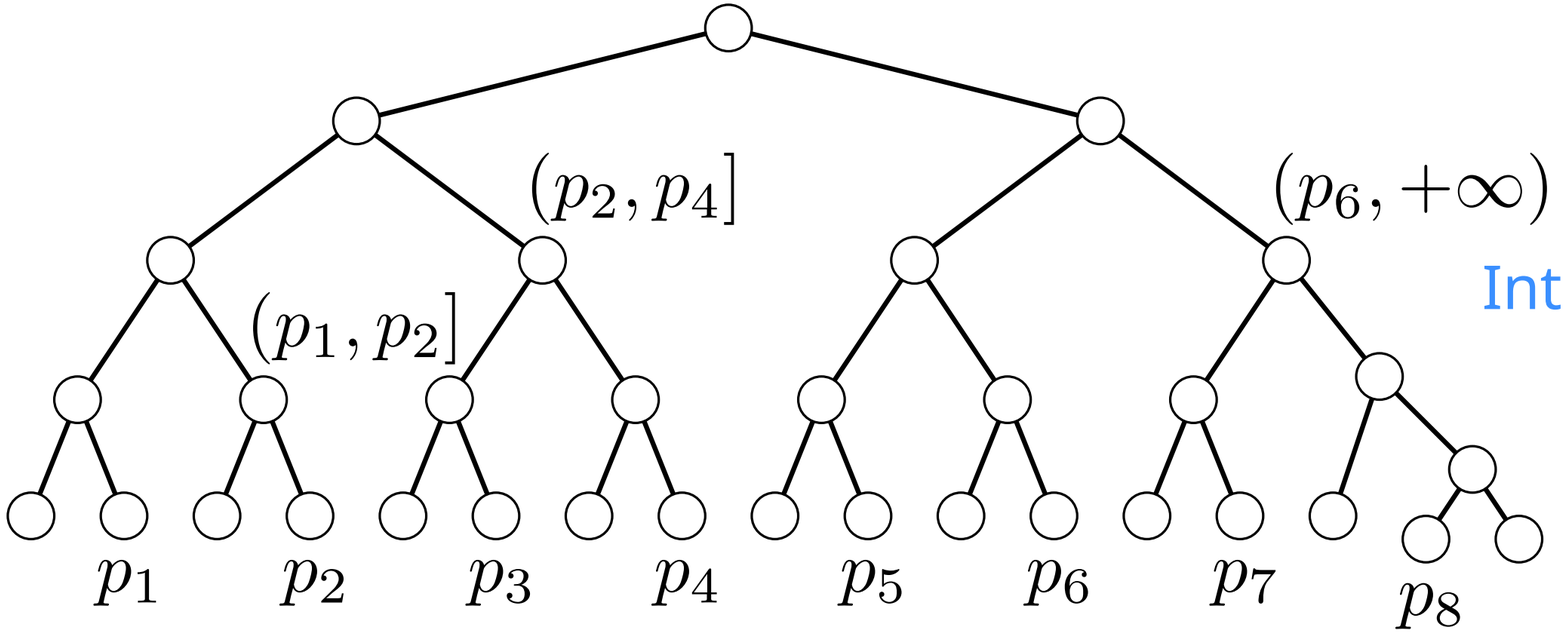


$\mathrm{Int}(v)$: interval of node v

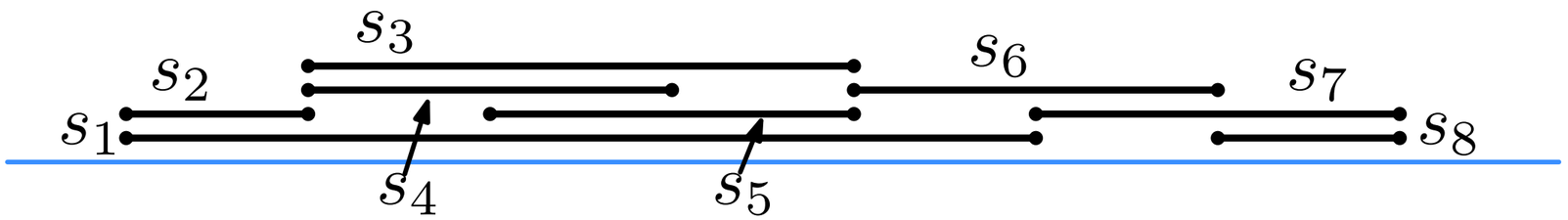


Avoid quadratic storage: store any segment s_j as high as possible in the tree

Towards segment trees



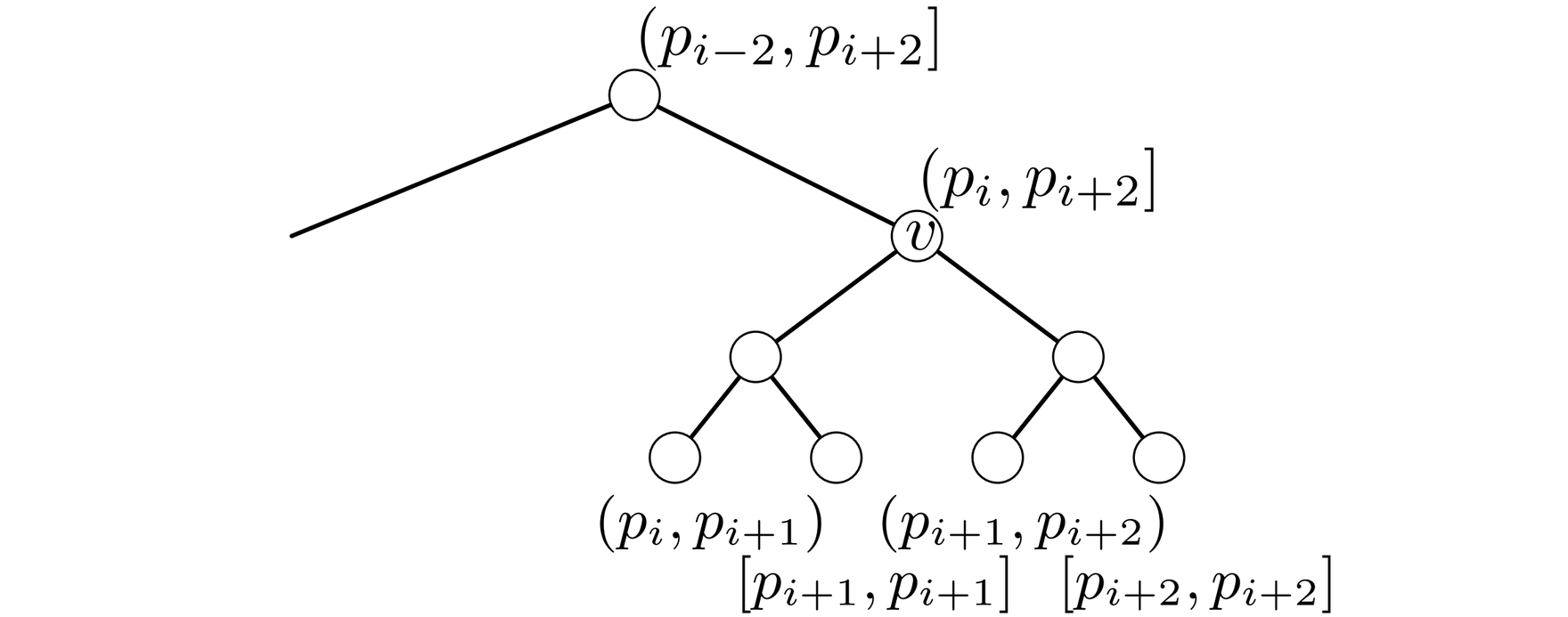
$\mathrm{Int}(v)$: interval of node v

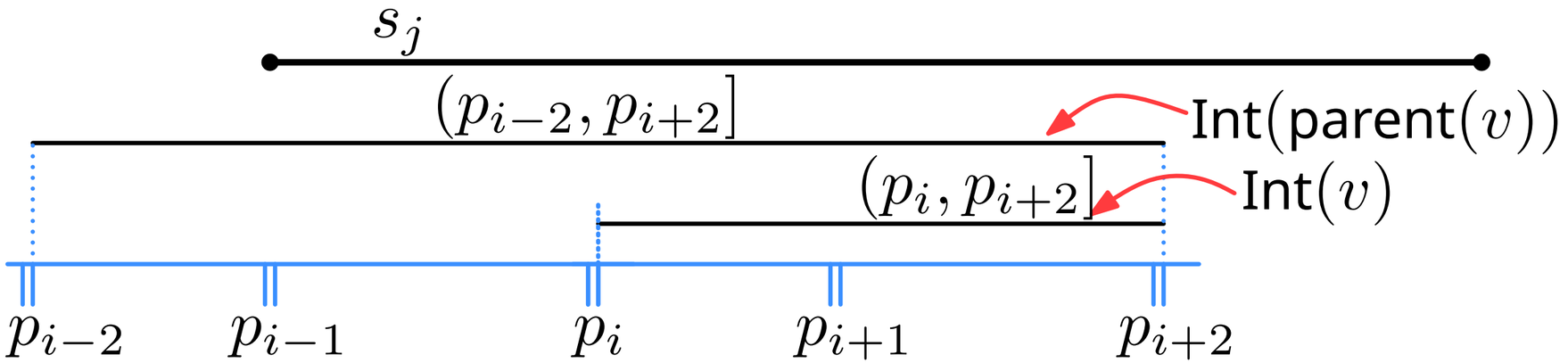


Avoid quadratic storage: store any segment s_j as high as possible in the tree

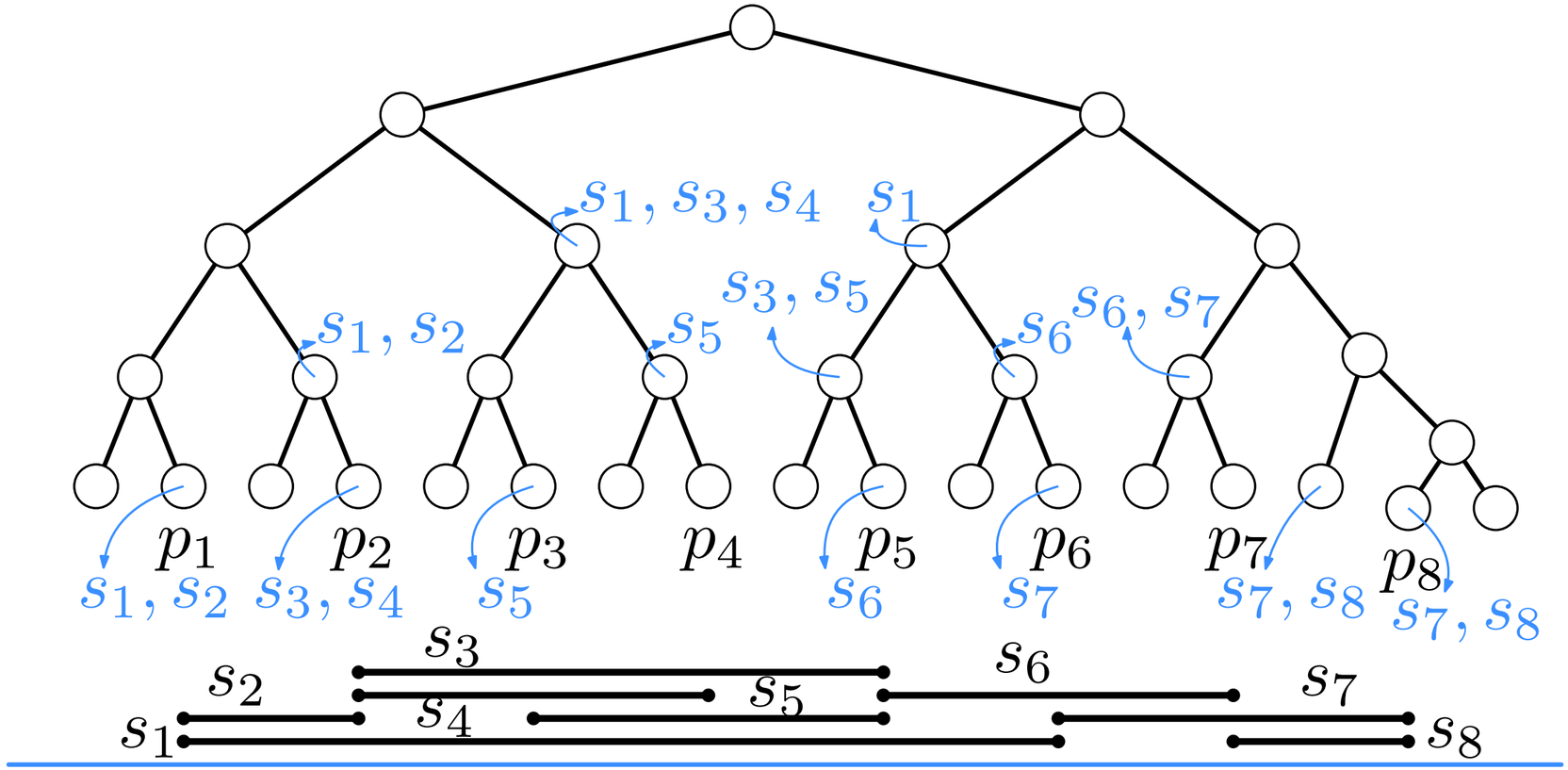
s_j is stored with $v \Leftrightarrow \mathrm{Int}(v) \subseteq s_j$ but $\mathrm{Int}(\mathrm{parent}(v)) \not\subseteq s_j$

Towards segment trees



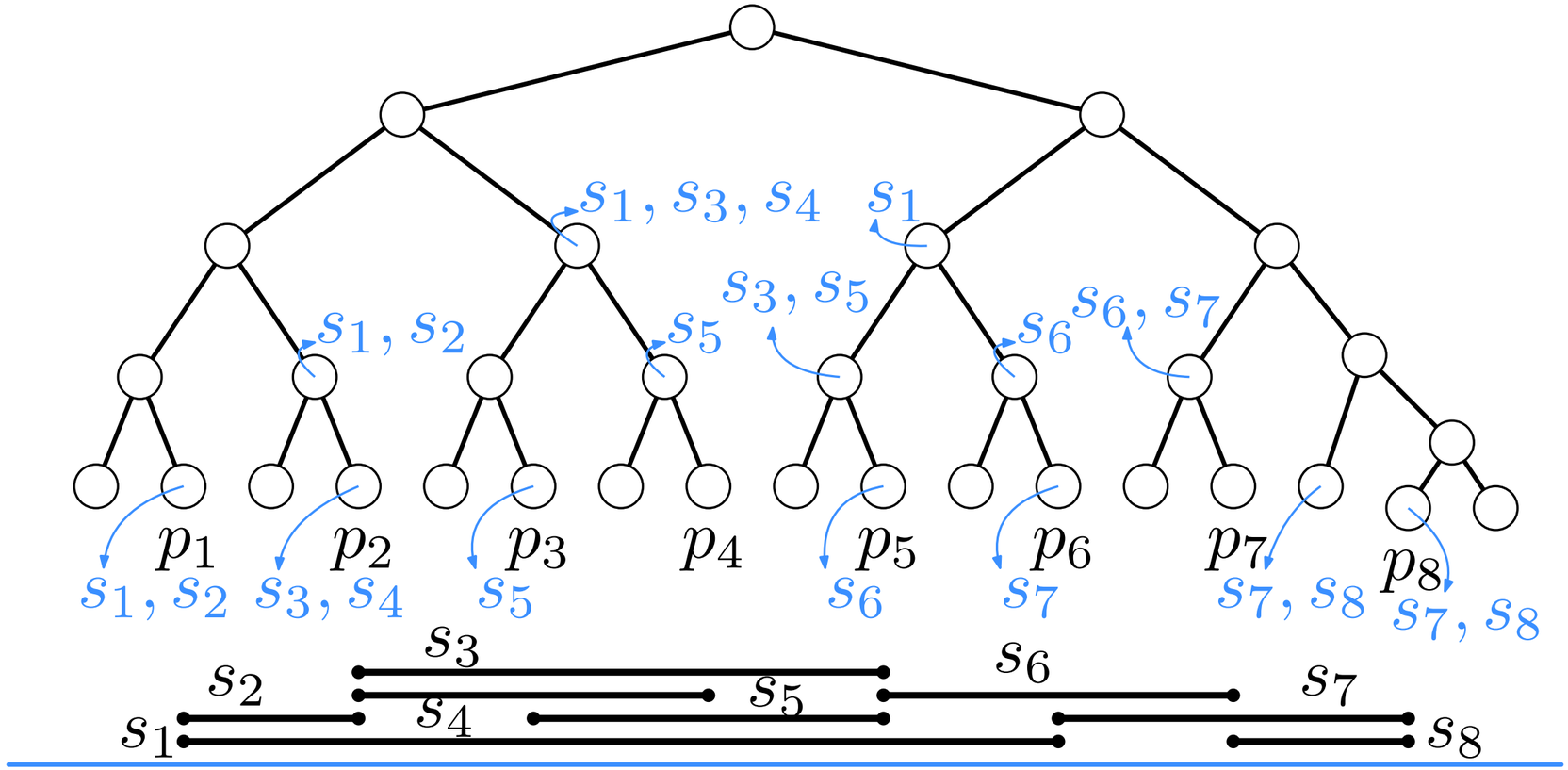


Segment trees



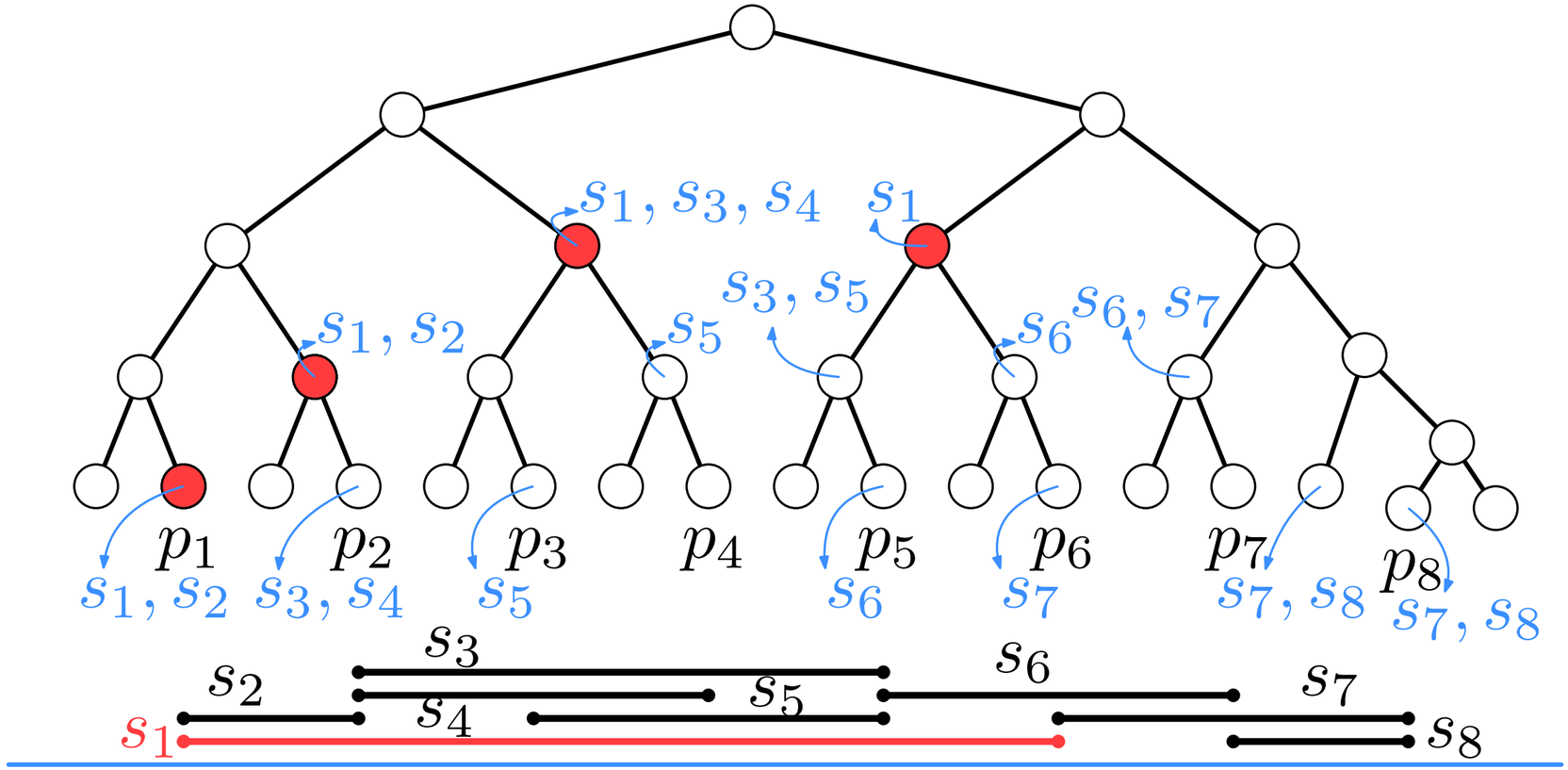
A segment tree on a set S of segments is a balanced binary search tree on the elementary intervals defined by S, and each node stores its interval, and its canonical subset of S in a list (unsorted)

Segment trees



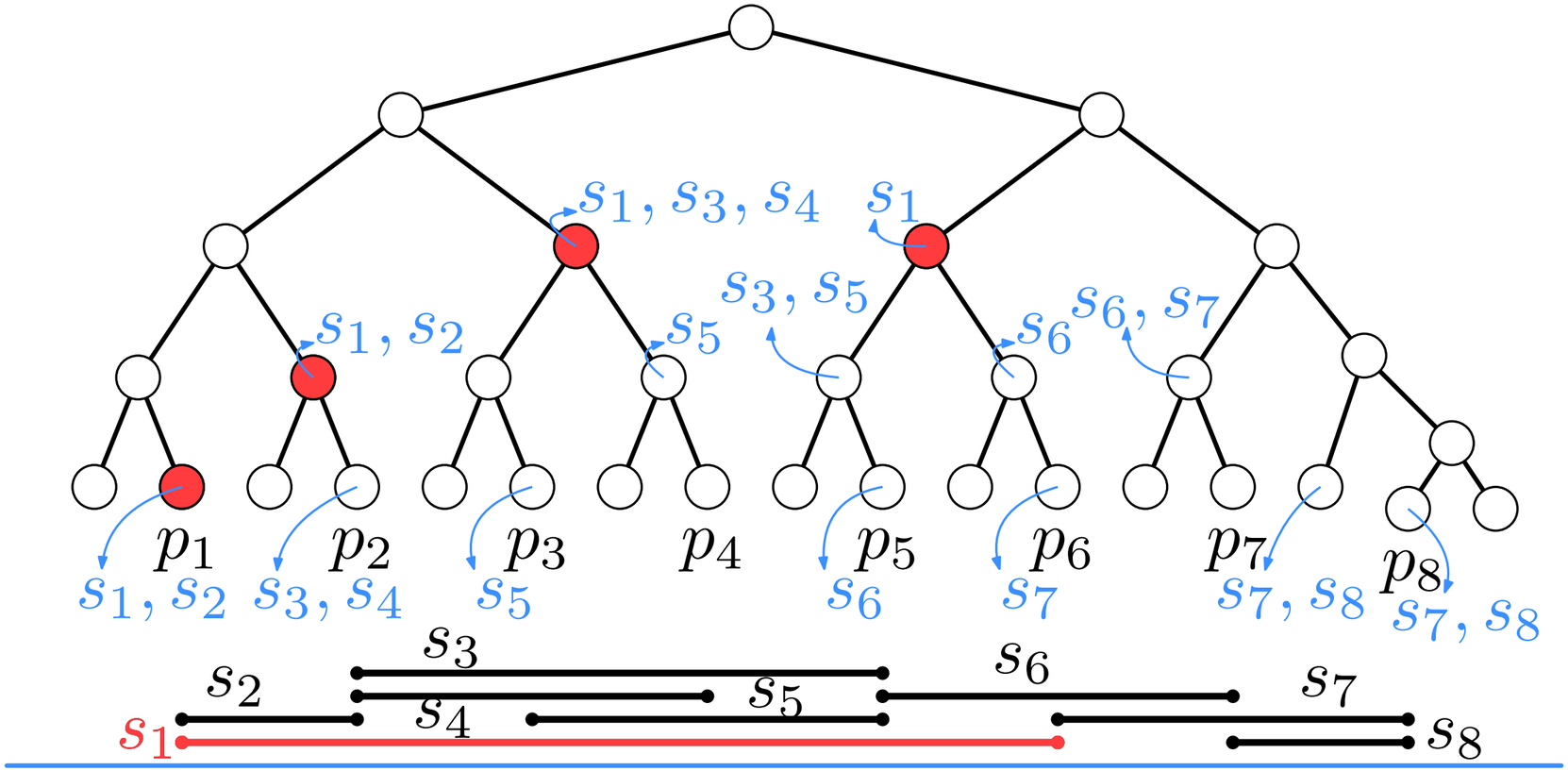
The canonical subset (of S) of a node v is the subset of segments s_j for which $\text{Int}(v) \subseteq s_j$ but $\text{Int}(\text{parent}(v)) \not\subseteq s_j$

Segment trees



The canonical subset (of S) of a node v is the subset of segments s_j for which $\text{Int}(v) \subseteq s_j$ but $\text{Int}(\text{parent}(v)) \not\subseteq s_j$

Segment trees



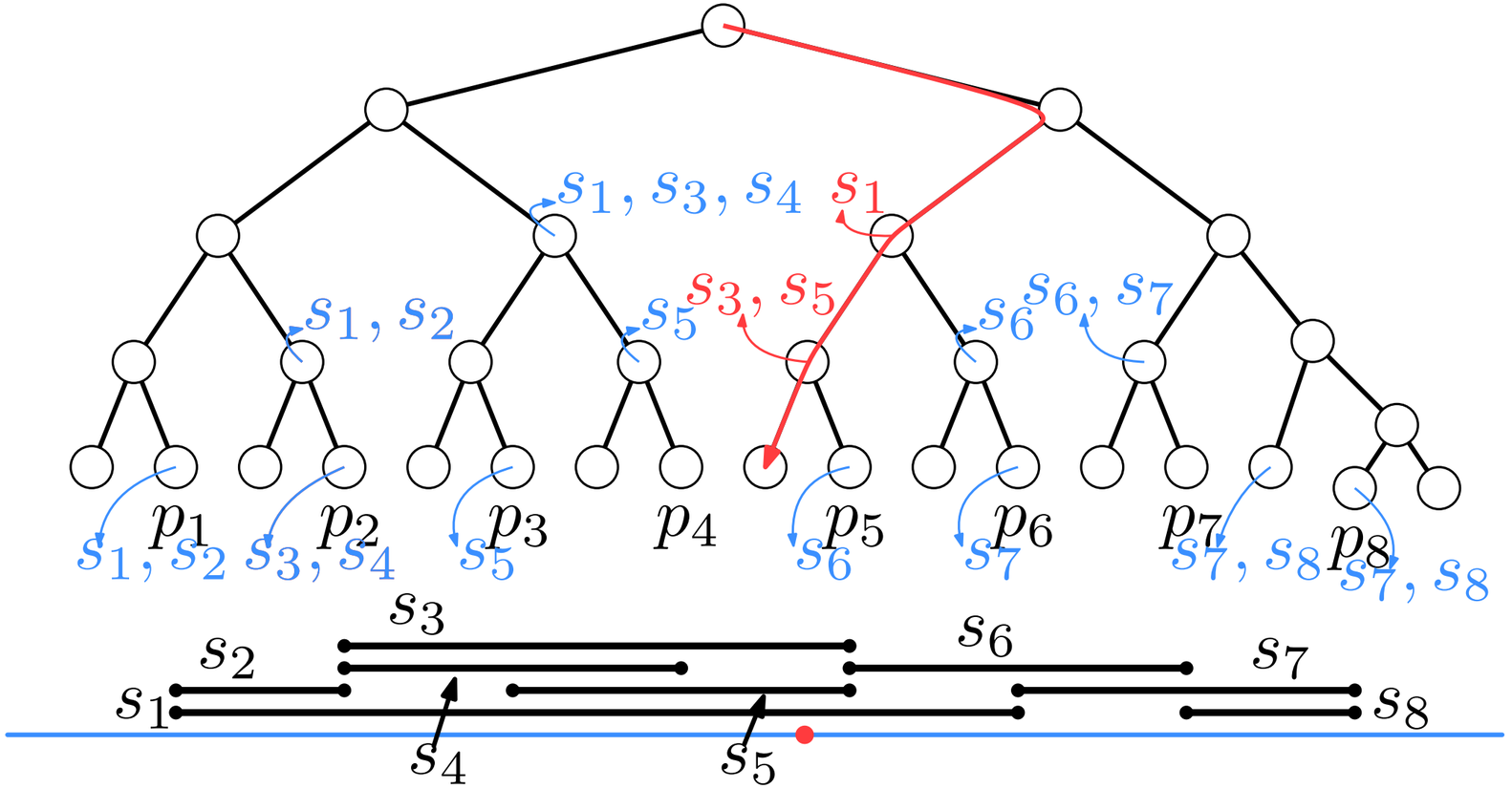
Why are no segments stored with nodes on the leftmost and rightmost paths of the segment tree?

Query algorithm

queries are easy:

For a query point q, follow the path down the tree to the elementary interval that contains q, and report all segments stored in the lists with the nodes on that path

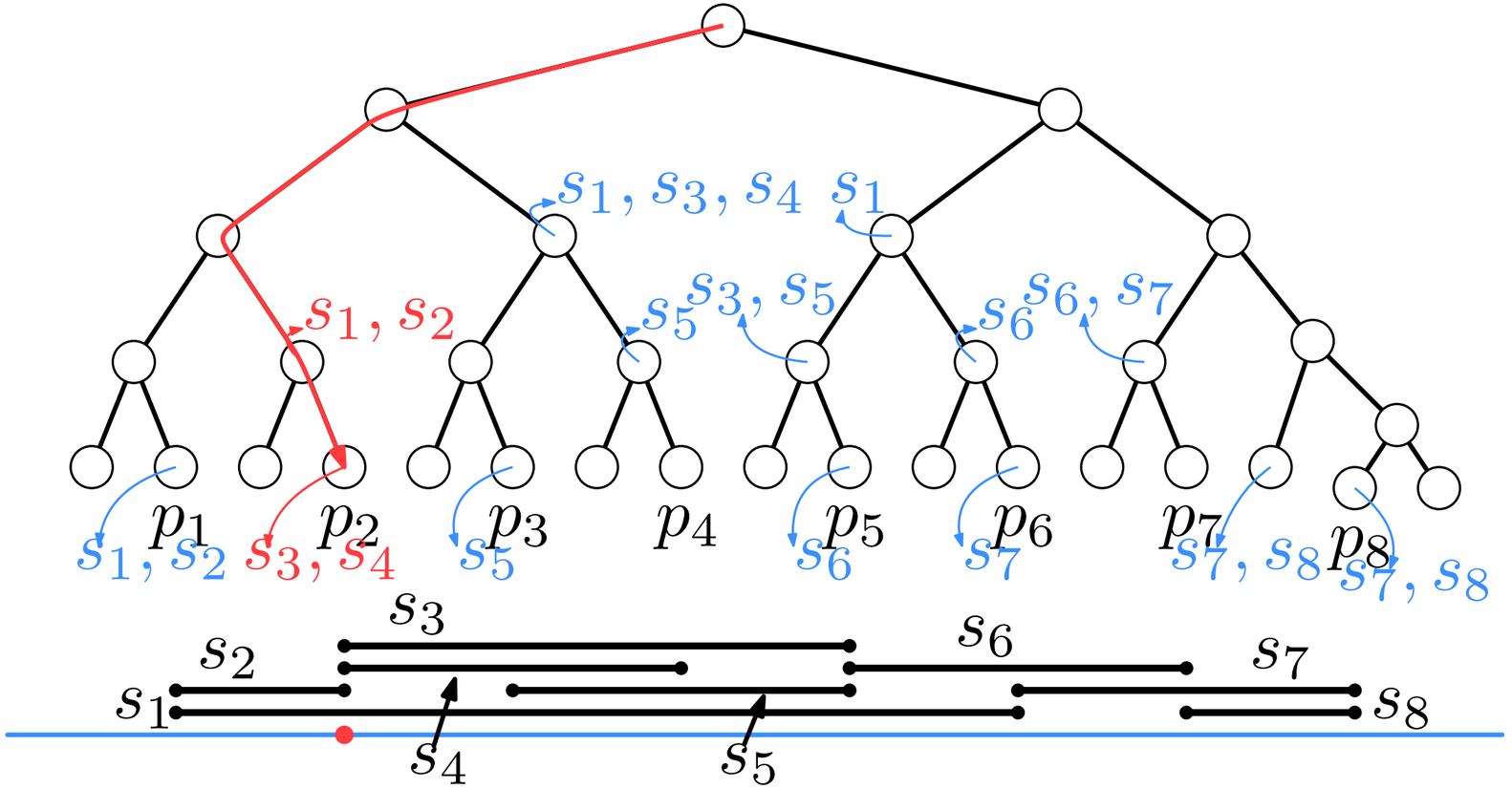
Example query



queries are easy:

For a query point q, follow the path down the tree to the elementary interval that contains q, and report all segments stored in the lists with the nodes on that path

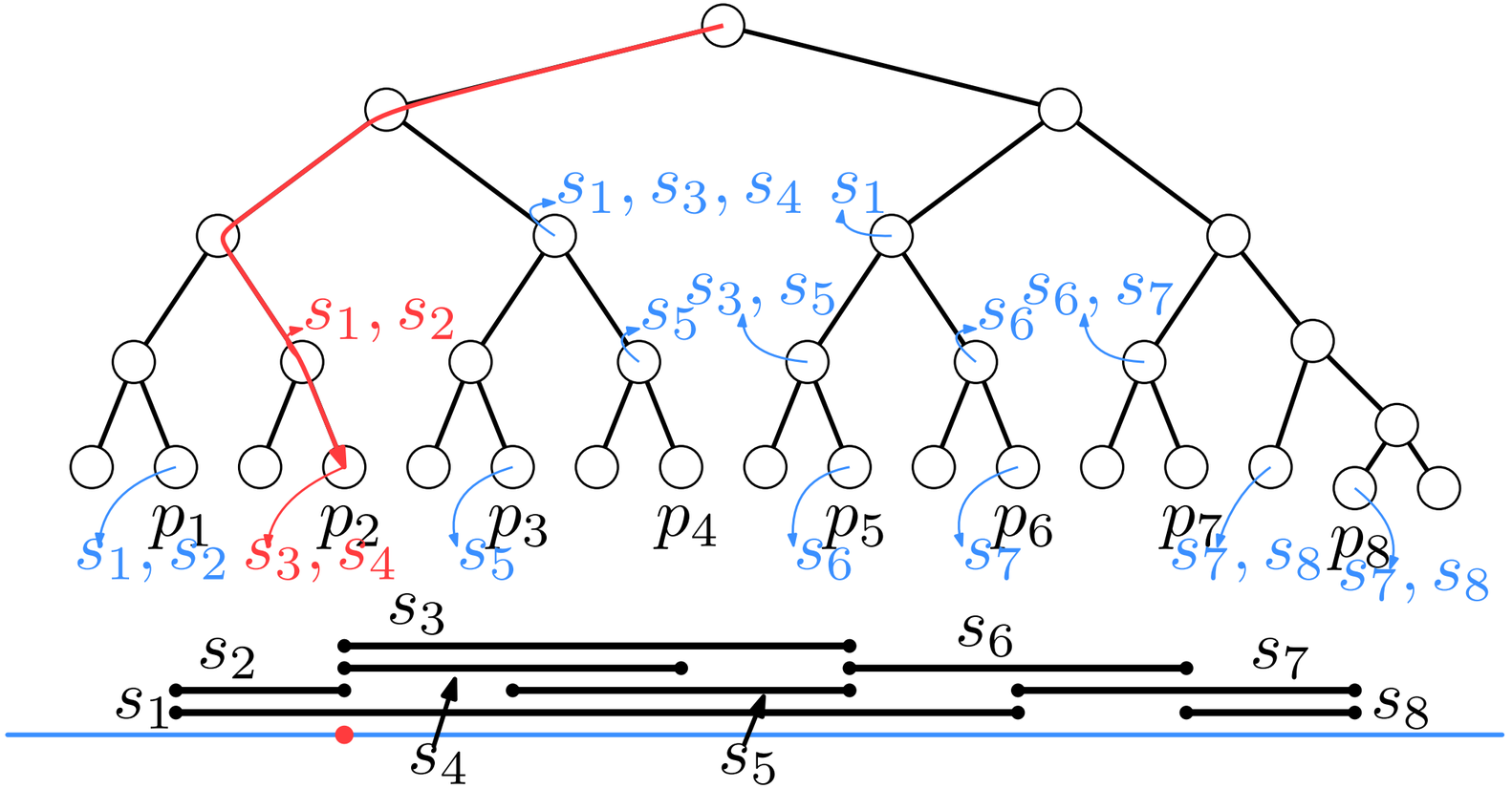
Example query



queries are easy:

For a query point q, follow the path down the tree to the elementary interval that contains q, and report all segments stored in the lists with the nodes on that path

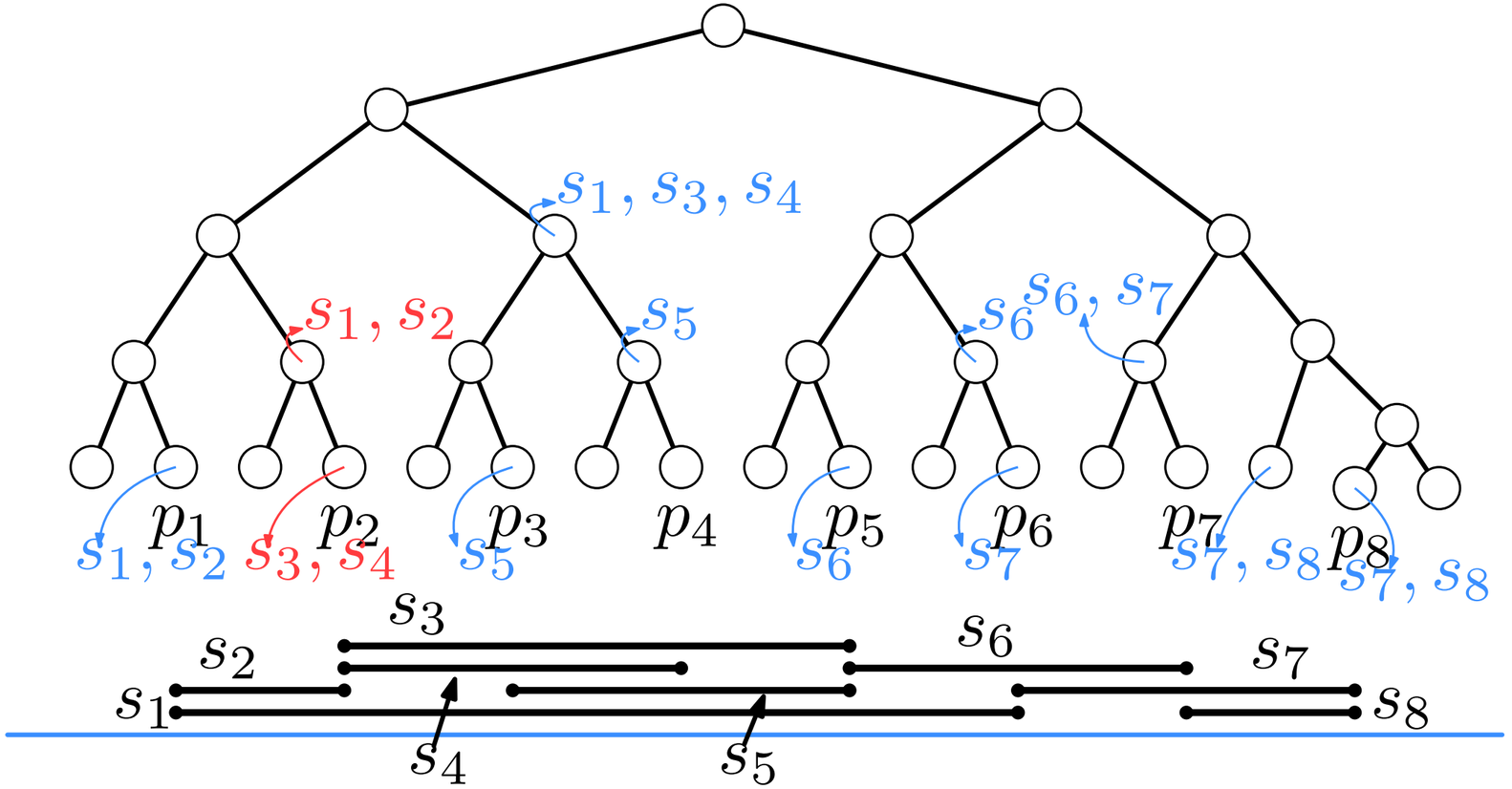
Example query



queries are easy:

The query time is $O(\log n + k)$, where k is the number of segments reported

Storage?



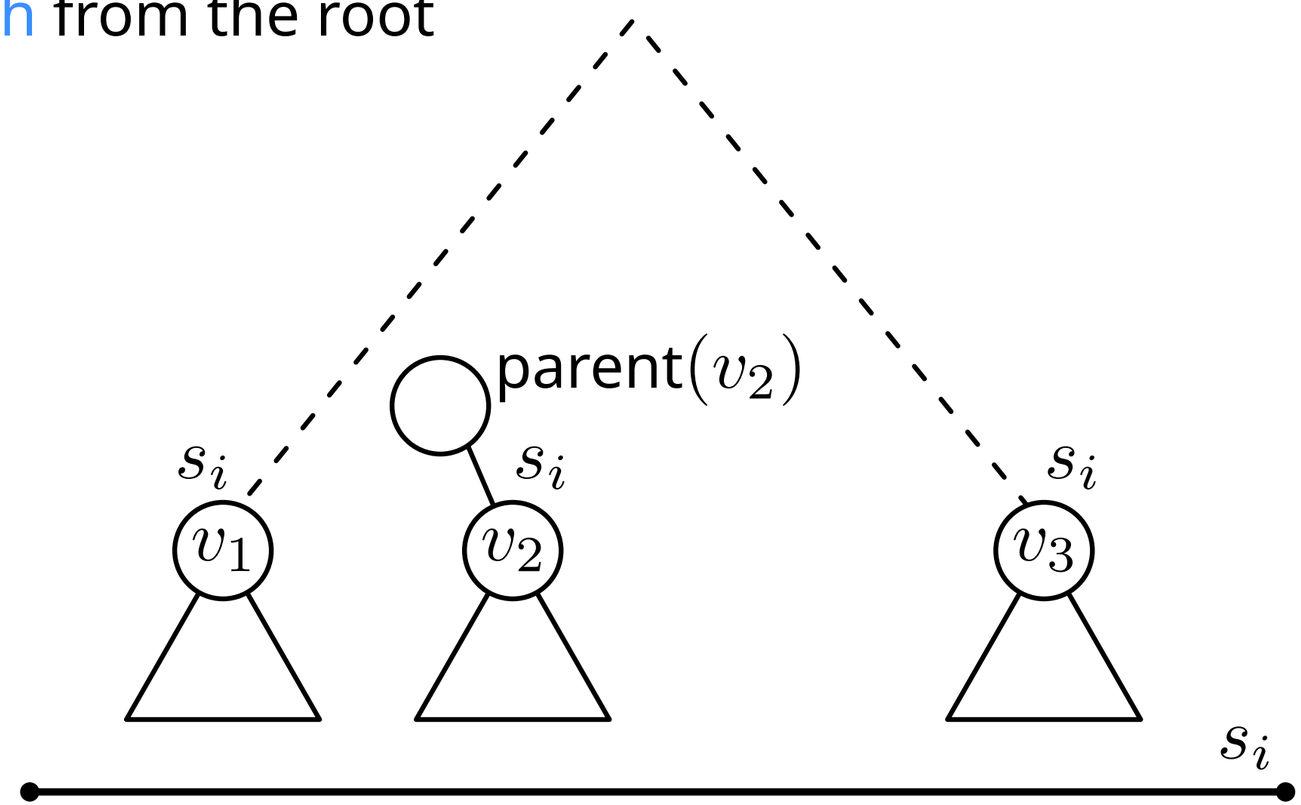
A segment can be stored in several lists of nodes.

How bad can the storage requirements get?

Storage

Lemma: Any segment can be stored at up to two nodes of the same depth

Proof: Suppose a segment s_i is stored at three nodes v_1, v_2, and v_3 at the same depth from the root



Storage

How bad can the storage requirements get?

A: $O(n)$

B: $O(n \log n)$

C: $O(n^2)$

Storage

How bad can the storage requirements get?

A: $O(n)$

B: $O(n \log n)$

C: $O(n^2)$

Storage

How bad can the storage requirements get?

A: $O(n)$

B: $O(n \log n)$

C: $O(n^2)$

If a segment tree has depth $O(\log n)$, then any segment is stored in at most $O(\log n)$ lists $\Rightarrow$ the total size of all lists is $O(n \log n)$

The main tree uses $O(n)$ storage

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

Result

Theorem: A segment tree storing n segments (= intervals) on the real line uses $O(n \log n)$ storage, can be built in $O(n \log n)$ time, and stabbing queries can be answered in $O(\log n + k)$ time, where k is the number of segments reported.

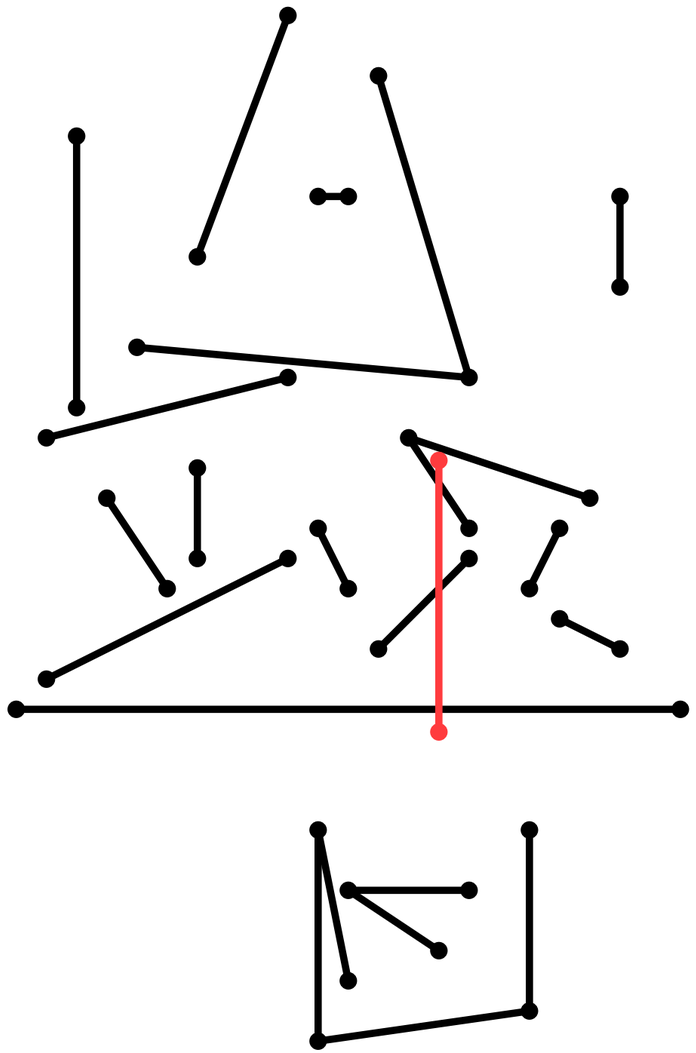
Property: For any query, all segments containing the query point are stored in the lists of $O(\log n)$ nodes.

next: Application to windowing

Range and Windowing Queries

Segment trees for windowing queries

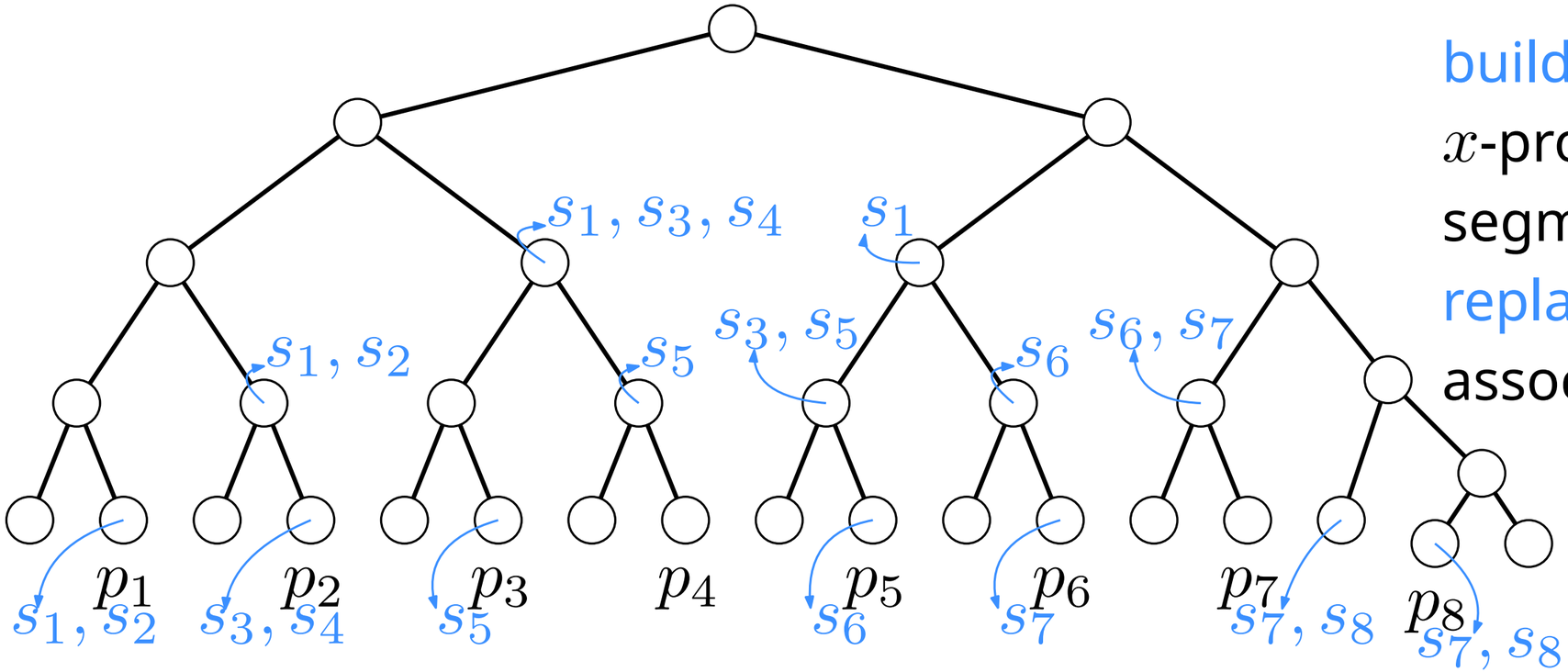
Back to windowing



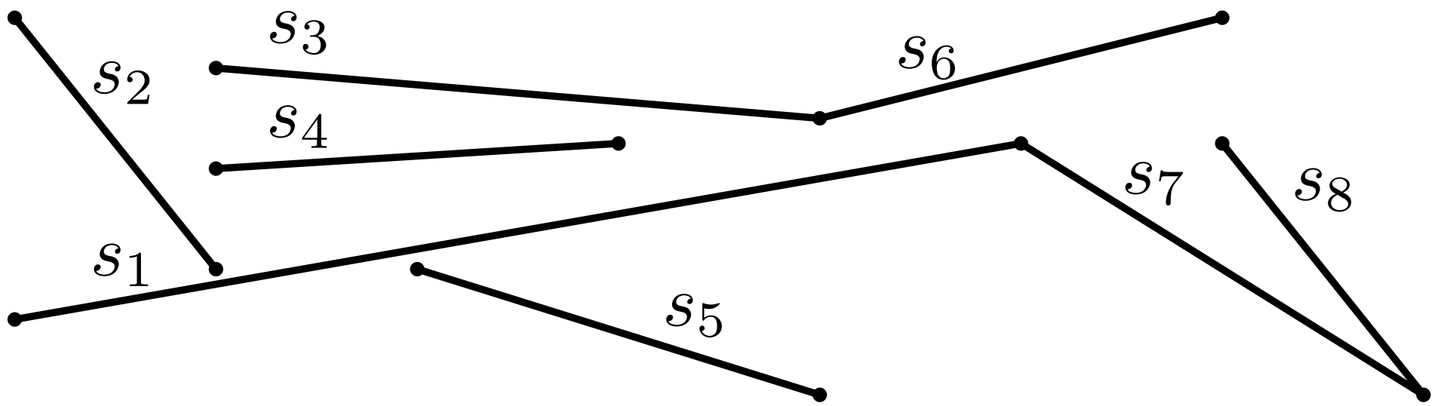
Problem arising from windowing:

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

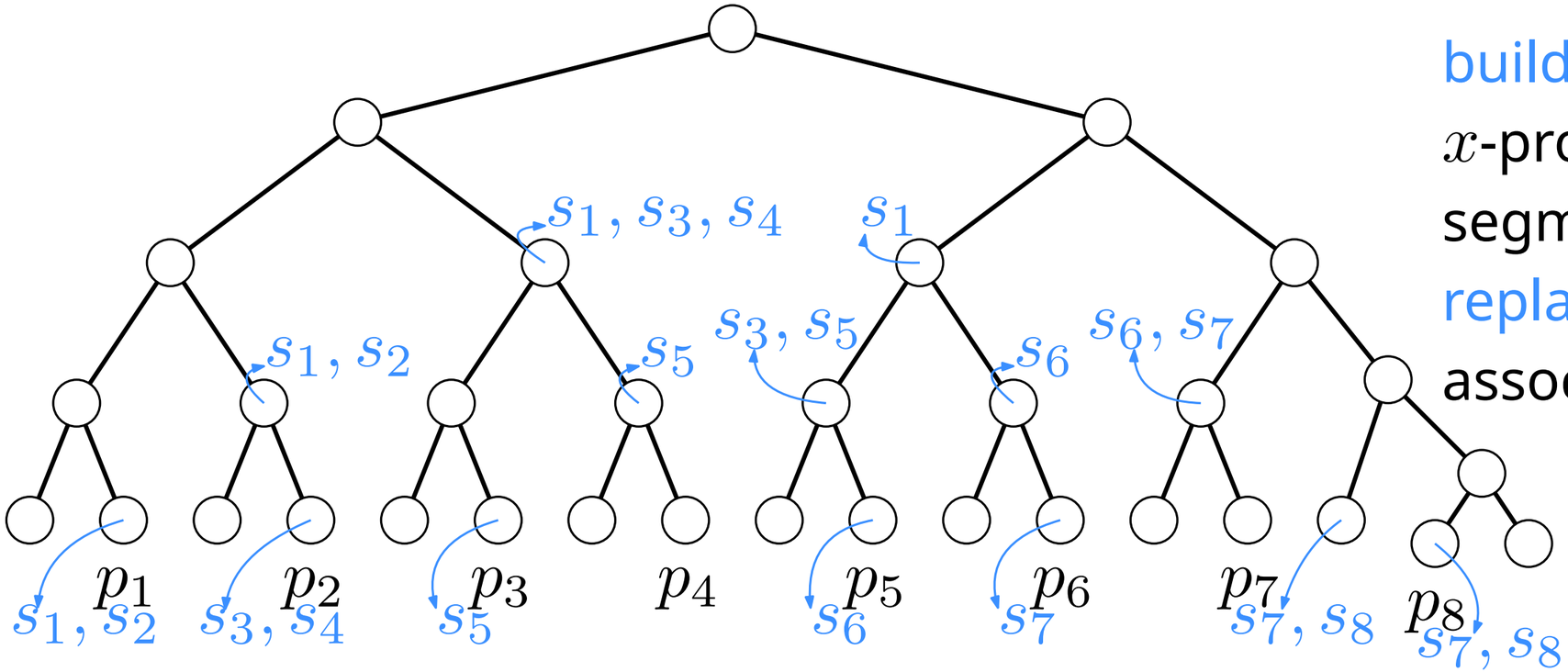
Idea for solution



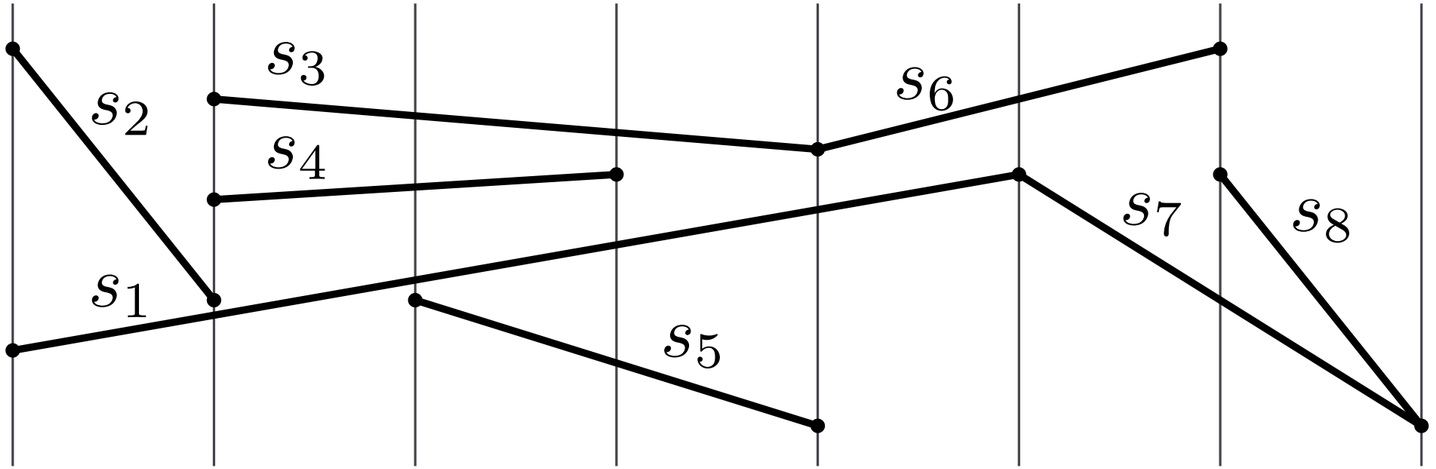
build segment tree on the x-projections of the segments and replace lists by suitable associated data structure



Idea for solution



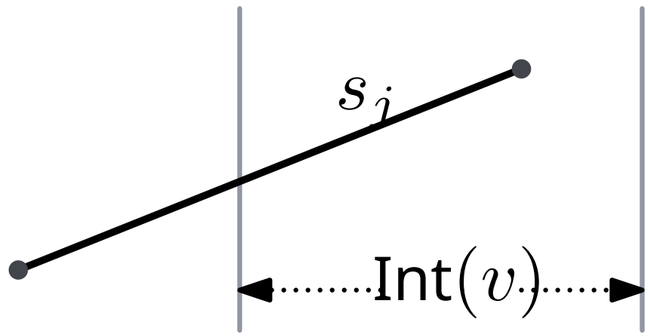
build segment tree on the x-projections of the segments and replace lists by suitable associated data structure



Idea for solution

Observe that nodes now correspond to vertical slabs of the plane, and:

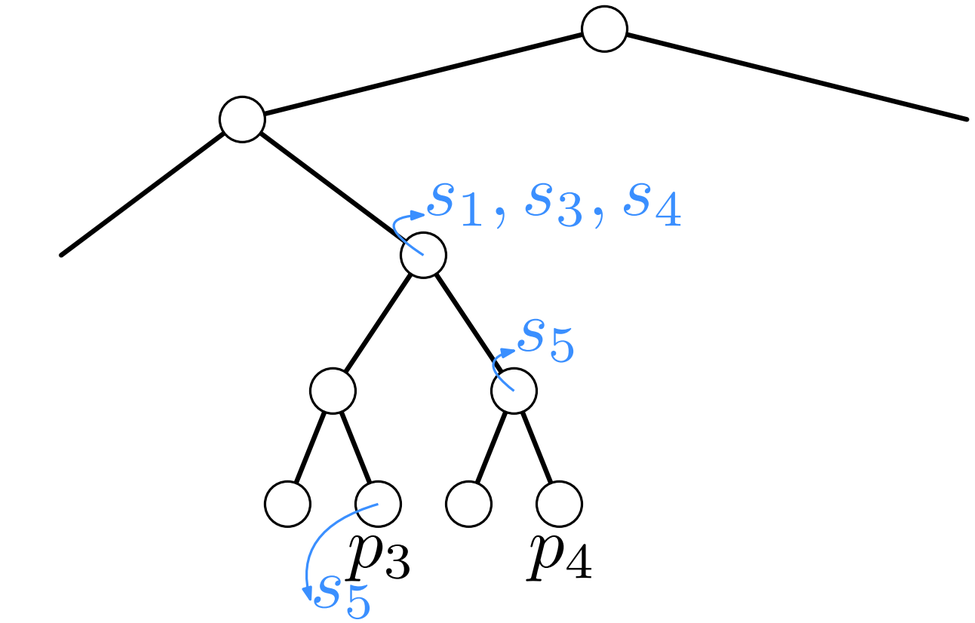
- if a segment s_i is stored with a node v, then it crosses the slab of v completely, but not the slab of the parent of v

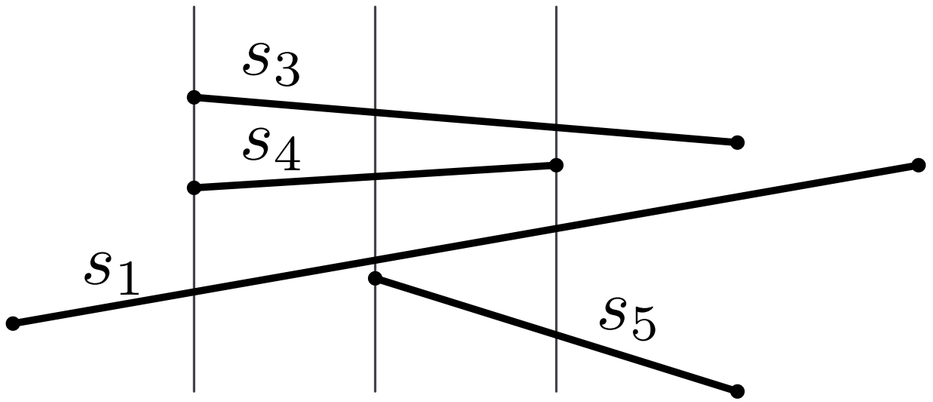


Example: s_j is stored at one or more nodes below v

Idea for solution

Observe that nodes now correspond to vertical slabs of the plane, and:

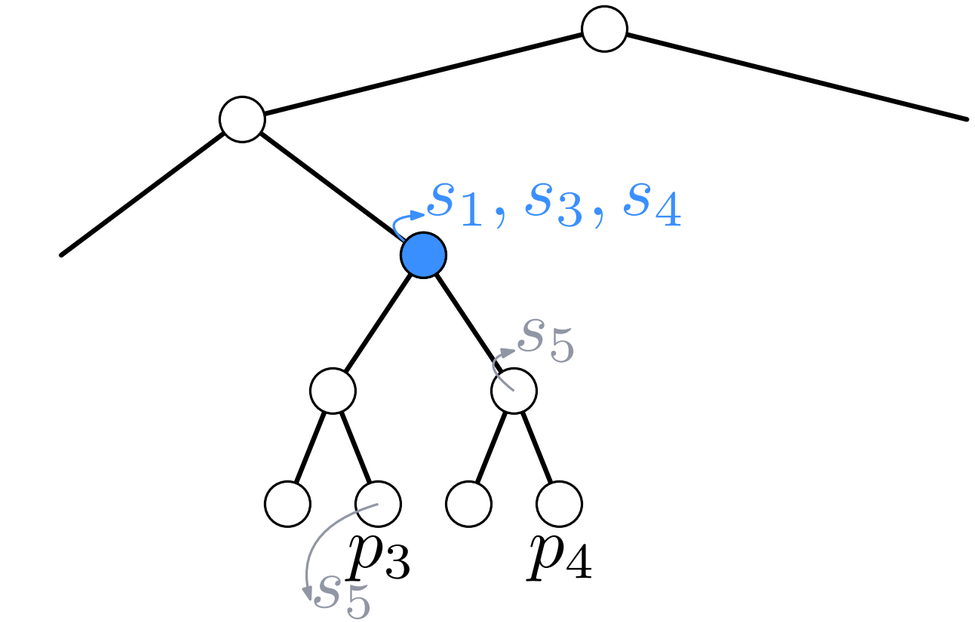


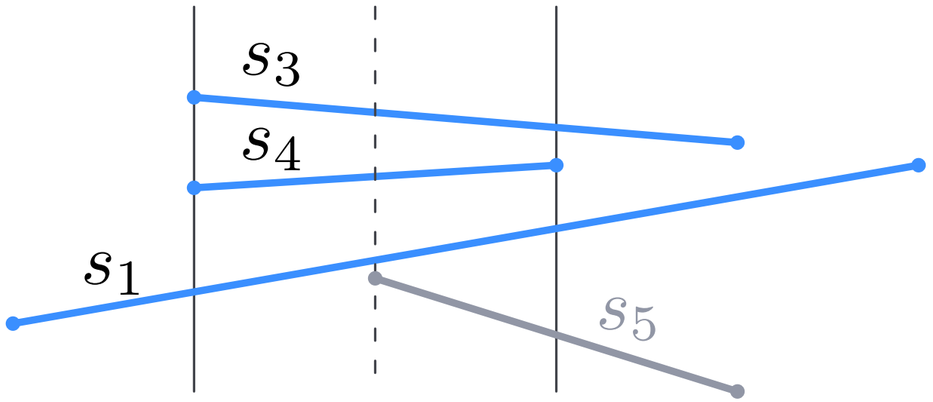


- if a segment s_i is stored with a node v, then it crosses the slab of v completely, but not the slab of the parent of v

Idea for solution

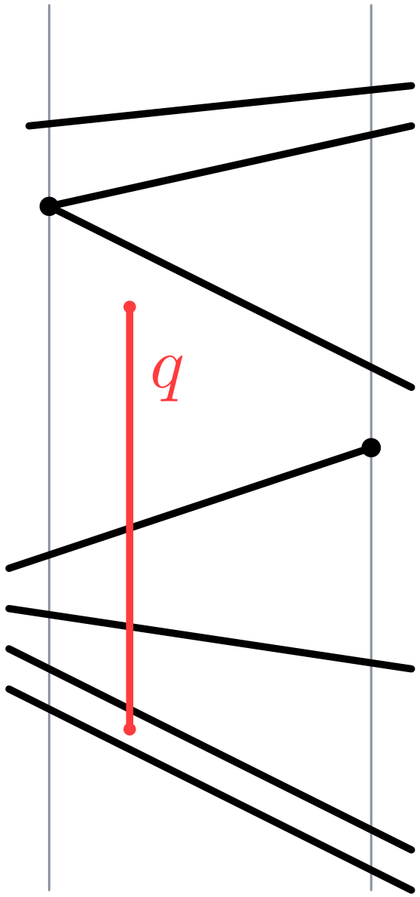
Observe that nodes now correspond to vertical slabs of the plane, and:





- if a segment s_i is stored with a node v, then it crosses the slab of v completely, but not the slab of the parent of v
- the segments crossing a slab have a well-defined top-to-bottom order

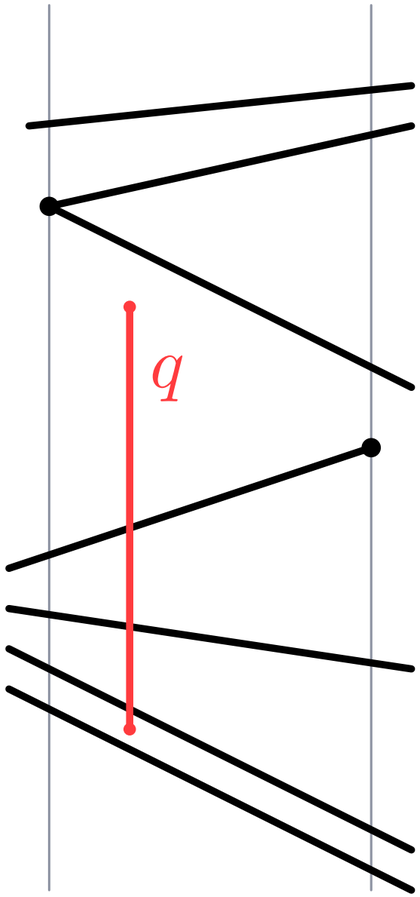
Querying



Query with a vertical line segment q

Only need segments stored with nodes on the path down the tree using the x-coordinate of q

Querying

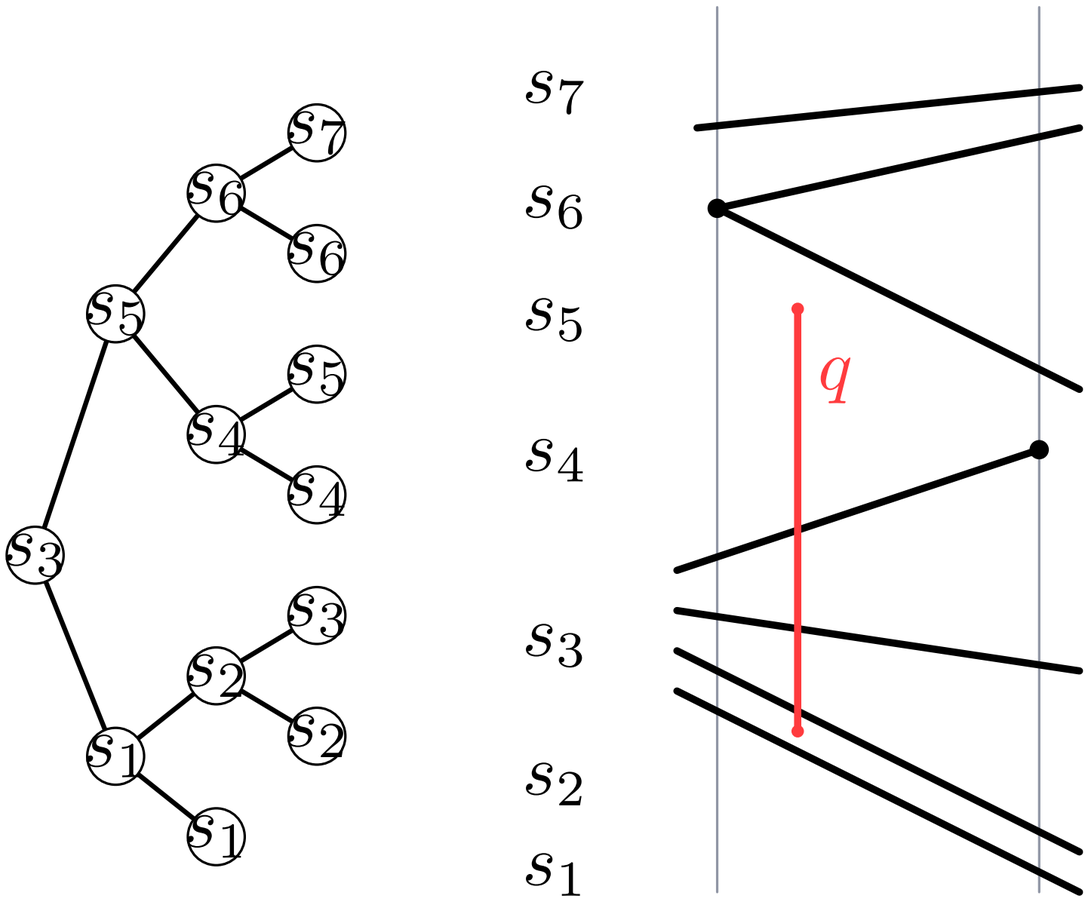


Query with a vertical line segment q

Only need segments stored with nodes on the path down the tree using the x-coordinate of q

At any such node, the query problem is: which of the segments (that cross the slab completely) intersects the vertical query segment q?

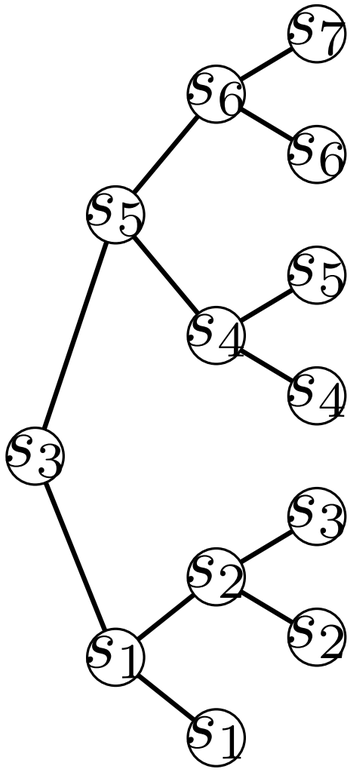
Querying

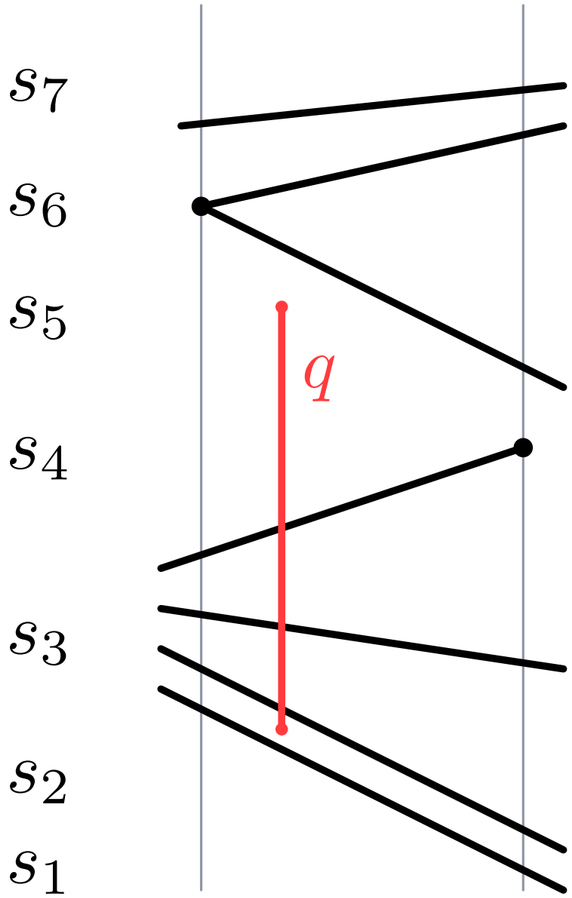


$1d$-range queries:
We store the canonical subset of a node v in a balanced binary search tree that follows the bottom-to-top order in its leaves

At any such node, the query problem is: which of the segments (that cross the slab completely) intersects the vertical query segment q?

Querying



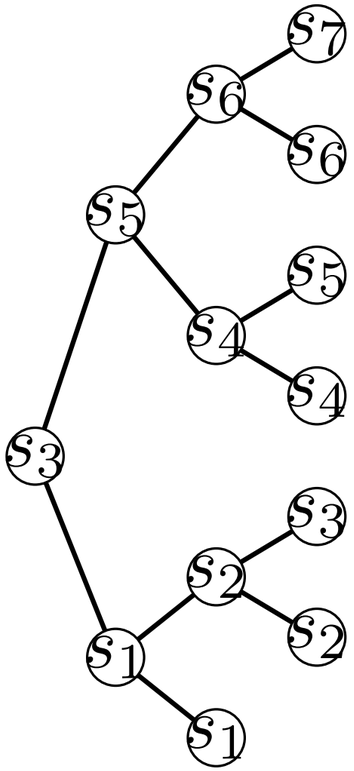


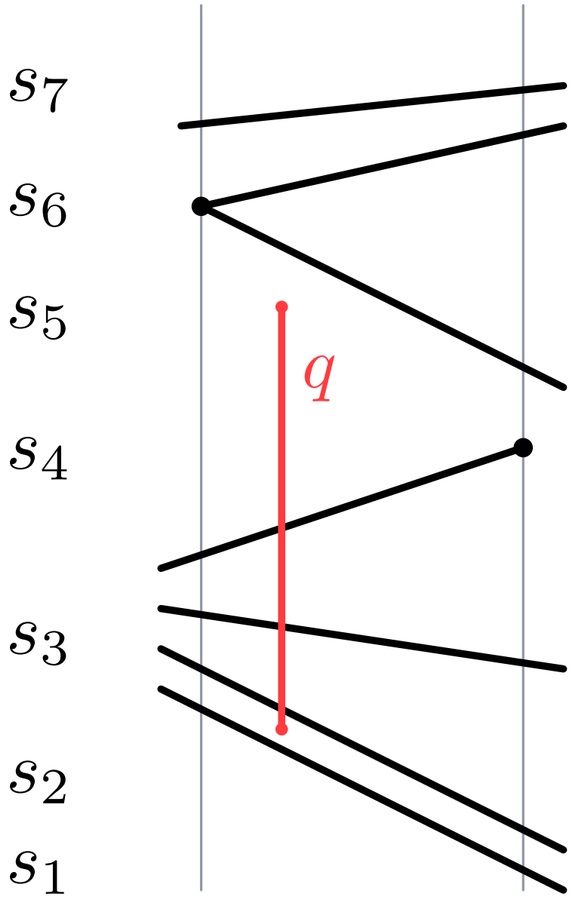
What is the overall query time for a windowing query?

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

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

Querying



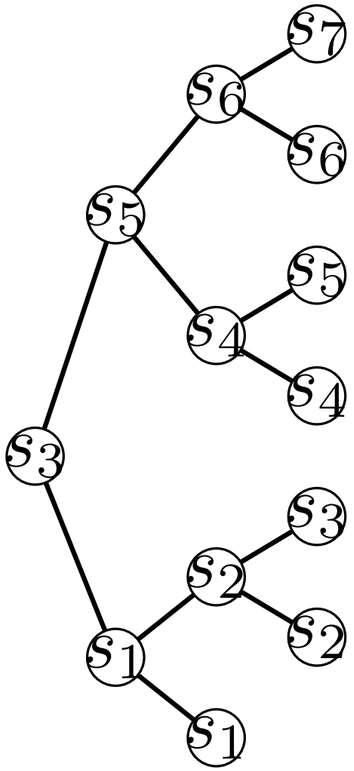


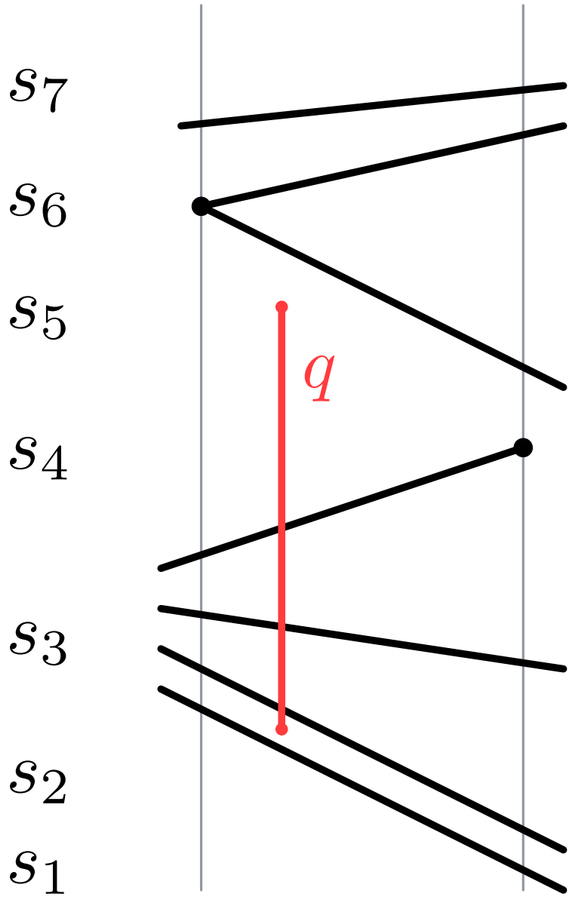
What is the overall query time for a windowing query?

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

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

Querying





What is the overall query time for a windowing query?

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

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

path down segment tree: $O(\log n)$ nodes

At each node: $1d$-range query in associated tree

Data structure: Summary

The data structure for intersection queries with a vertical query segment in a set of non-crossing line segments is a segment tree where the associated structures are binary search trees (i.e. $1d$-range trees) on the bottom-to-top order of the segments in the corresponding slab

The query time is $O(\log^2 n + k)$

Since it is a segment tree with lists replaced by trees, the storage remains $O(n \log n)$

Result

Theorem: A set of n non-crossing line segments can be stored in a data structure of size $O(n \log n)$ so that intersection queries with a vertical query segment can be answered in $O(\log^2 n + k)$ time, where k is the number of answers reported

Theorem: A set of n non-crossing line segments can be stored in a data structure of size $O(n \log n)$ so that windowing queries can be answered in $O(\log^2 n + k)$ time, where k is the number of answers reported

Summary of query data structures

Data structure	Objects	Query object	Query	Query time	Storage	Preprocessing
1D range tree	1D pts x_i	interval I	$x_i \in I$	$O(\log n + k)$	$O(n)$	$O(n \log n)$
2D range tree	2D pts p_i	axis-aligned rectangle R	$p_i \in R$	$O(\log^2 n + k)$, fr.ca.:$O(\log n + k)$	$O(n \log n)$	$O(n \log n)$
Range tree in $\mathbb{R}^d$	pts p_i in $\mathbb{R}^d$	axis-aligned box R	$p_i \in R$	$O(\log^d n + k)$, fr.ca.:$O(\log^{d-1} n + k)$	$O(n \log^{d-1} n)$	$O(n \log^{d-1} n)$
Priority search tree	2D pts p_i	2- or 3-sided range R	$p_i \in R$	$O(\log n + k)$	$O(n)$	$O(n \log n)$
Interval tree	1D intervals s_i	point p	$s_i \ni p$	$O(\log n + k)$	$O(n)$	$O(n \log n)$
Segment tree	1D intervals s_i	point p	$s_i \ni p$	$O(\log n + k)$	$O(n \log n)$	$O(n \log n)$
Interval tree + 2D RT as assoc.DS	horizontal segments s_i	vertical segment s	$s_i \cap s \neq \emptyset$	$O(\log^2 n + k)$	$O(n \log n)$	$O(n \log n)$
Interval tree + PST as assoc.DS	horizontal segments s_i	vertical segment s	$s_i \cap s \neq \emptyset$	$O(\log^2 n + k)$	$O(n)$	$O(n \log n)$
Segment tree + 1D RT as assoc.DS	non-crossing arbitrary segments s_i	vertical segment s	$s_i \cap s \neq \emptyset$	$O(\log^2 n + k)$	$O(n \log n)$	$O(n \log n)$
2D windowing DS ● 2D range tree ● 2 × (IT + PST)	horiz. and vert. segments s_i	axis-aligned box R	$s_i \in R$	$O(\log^2 n + k)$	$O(n \log n)$	$O(n \log n)$
2D windowing DS ● 2D range tree ● 4 × (ST + BST)	non-crossing arbitrary segments s_i	axis-aligned box R	$s_i \in R$	$O(\log^2 n + k)$	$O(n \log n)$	$O(n \log n)$