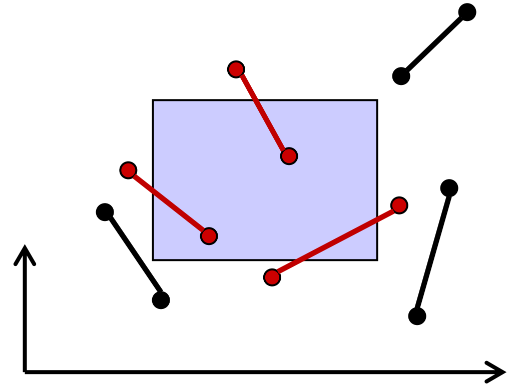
Computational Geometry



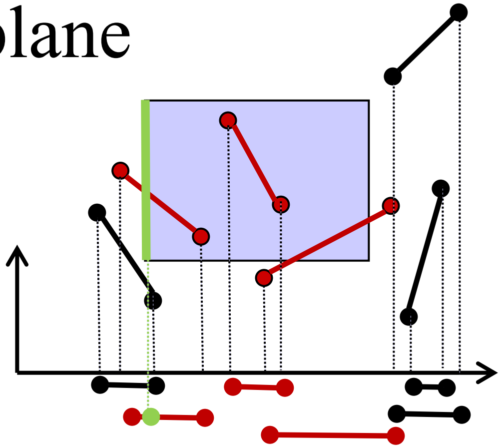
Windowing

Michael Goodrich

with slides from Carola Wenk and Kevin Buchin

Windowing

Input: A set S of n line segments in the plane



Query: Report all segments in S that intersect a given query window

Subproblem: Process a set of intervals on the line into a data structure which supports queries of the type: Report all intervals that contain a query point.

$\Rightarrow$ Interval trees

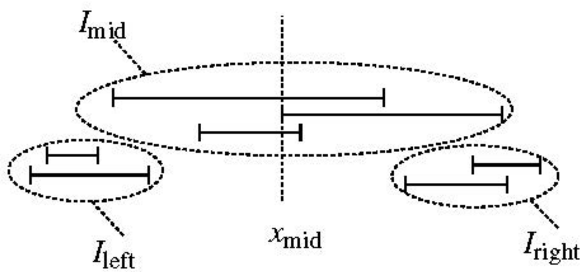
$\Rightarrow$ Segment trees

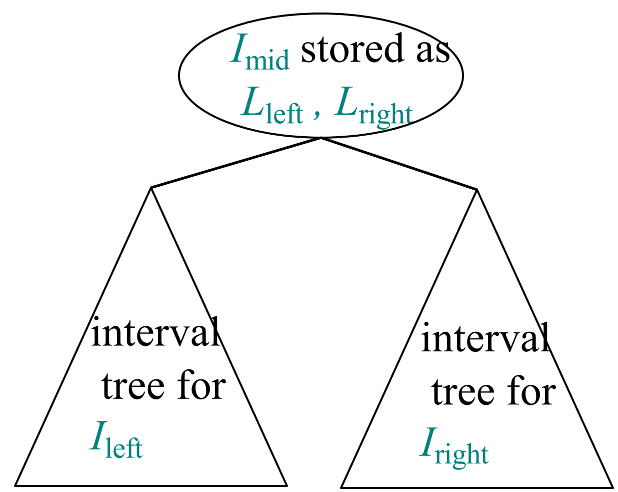
Interval Trees

Input: A set I of n intervals on the line.

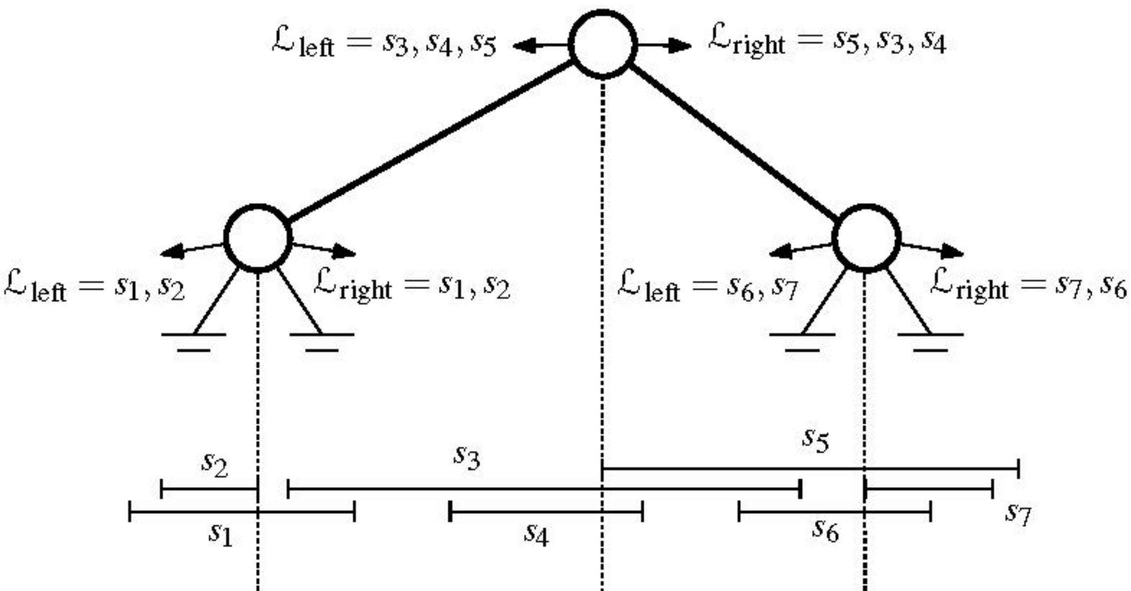
disjoint union

Idea: Partition I into $I_{left} \dot\cup I_{mid} \dot\cup I_{right}$ where x_{mid} is the median of the $2n$ endpoints.
Store I_{mid} twice as two lists of intervals: L_{left} sorted by left endpoint and as L_{right} sorted by right endpoint.





Interval Trees



Lemma: An interval tree on a set of n intervals uses $O(n)$ space and has height $O(\log n)$. It can be constructed recursively in $O(n \log n)$. time.

Proof: Each interval is stored in a set I_{mid} only once, hence $O(n)$ space. In the worst case half the intervals are to the left and right of x_{mid}, hence the height is $O(\log n)$. Constructing the (sorted) lists takes $O(|I^v| + |I^v_{\text{mid}}| \log |I^v_{\text{mid}}|)$ time per vertex v. □

Interval Tree Query

Algorithm QUERYINTERVALTREE(ν, q_x)
Input. The root ν of an interval tree and a query point q_x.
Output. All intervals that contain q_x.

if ν is not a leaf
  then if $q_x < x_{\text{mid}}(\nu)$
    then Walk along the list $\mathcal{L}_{\text{left}}(\nu)$, starting at the interval with the leftmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
      QUERYINTERVALTREE($lc(\nu), q_x$)
    else Walk along the list $\mathcal{L}_{\text{right}}(\nu)$, starting at the interval with the rightmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
      QUERYINTERVALTREE($rc(\nu), q_x$)

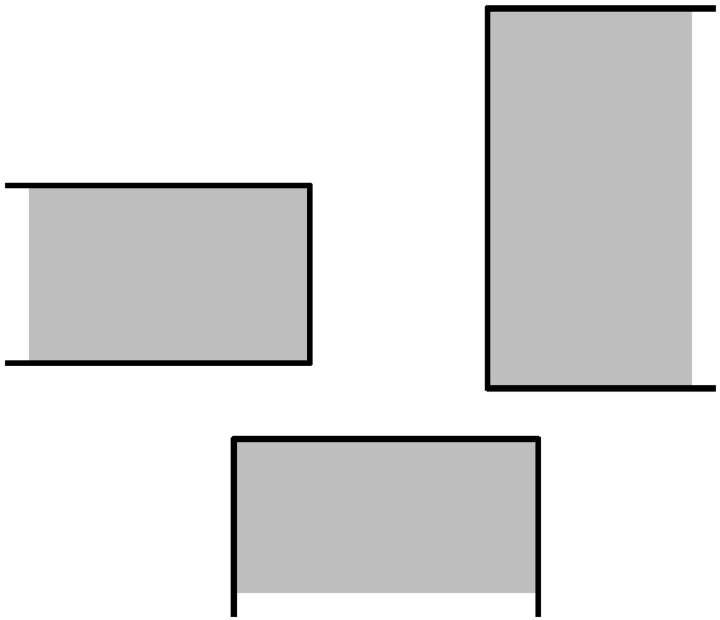
Theorem: An interval tree on a set of n intervals can be constructed in $O(n \log n)$ time and uses $O(n)$ space. All intervals that contain a query point can be reported in $O(\log n + k)$ time, where k = #reported intervals.

Proof: We spend $O(1+k_v)$ time at vertex v, where k_v = #intervals reported at v. We visit at most 1 node at any depth. □

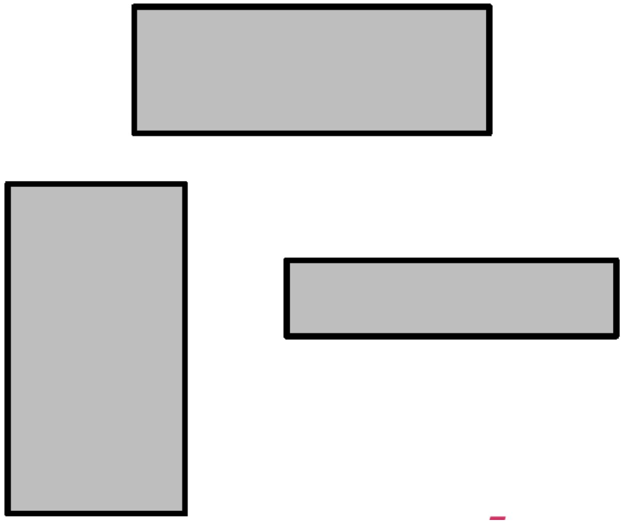
3-Sided Range Queries

- 3-sided range queries want all points in a range where one of the sides is unbounded.

3-sided ranges:



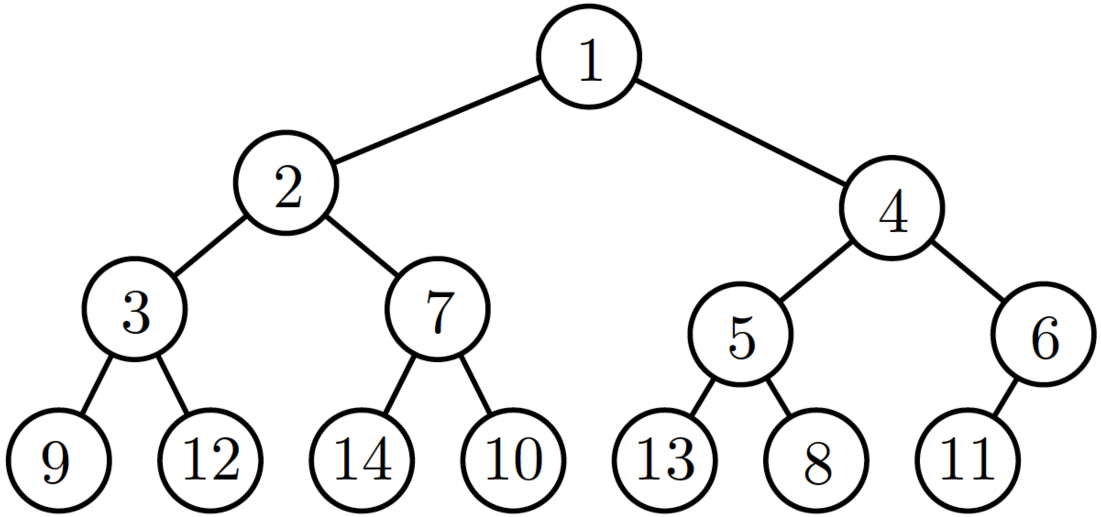
4-sided ranges:



Priority Search Trees

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

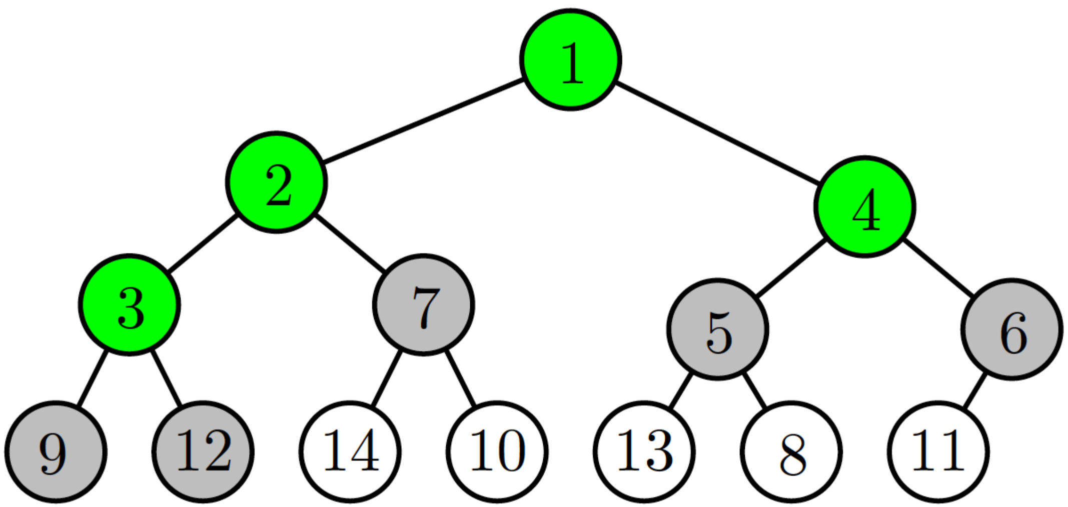
Recall the heap:



Priority Search Trees

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

Recall the heap:



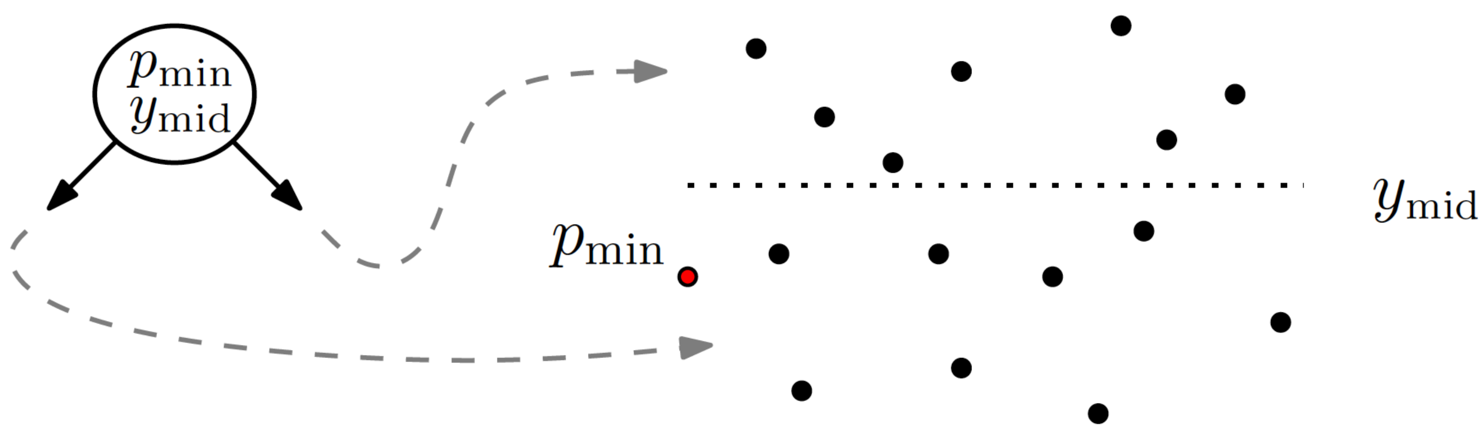
Report all values ≤ 4

Priority Search Trees

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

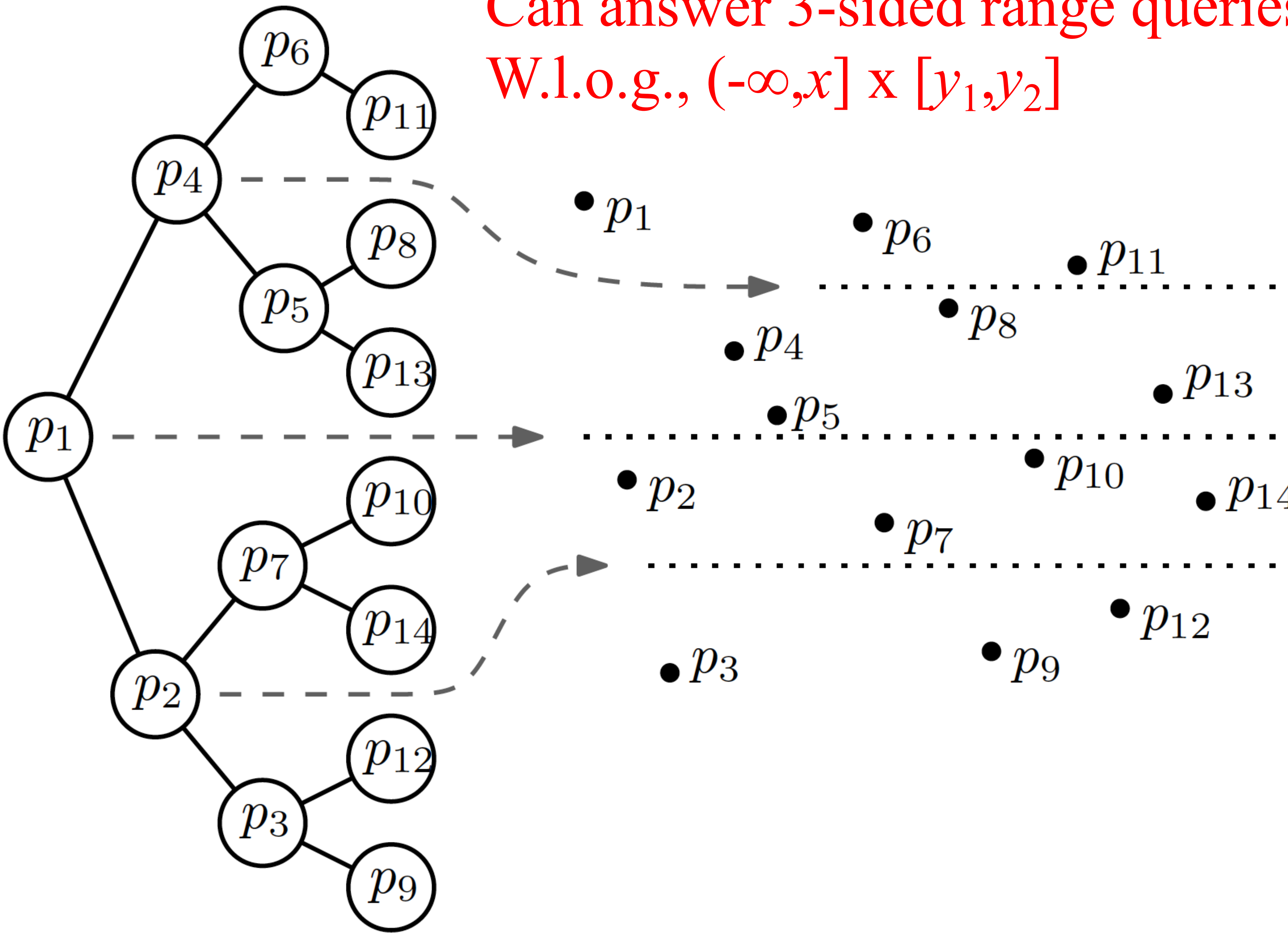
Otherwise, let $p_{\min}$ be the leftmost point in P, and let y_{mid} be the median y-coordinate of $P \setminus \{p_{\min}\}$

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 $P \setminus \{p_{\min}\}$ with y-coordinate $\leq y_{\text{mid}}$ and $> y_{\text{mid}}$



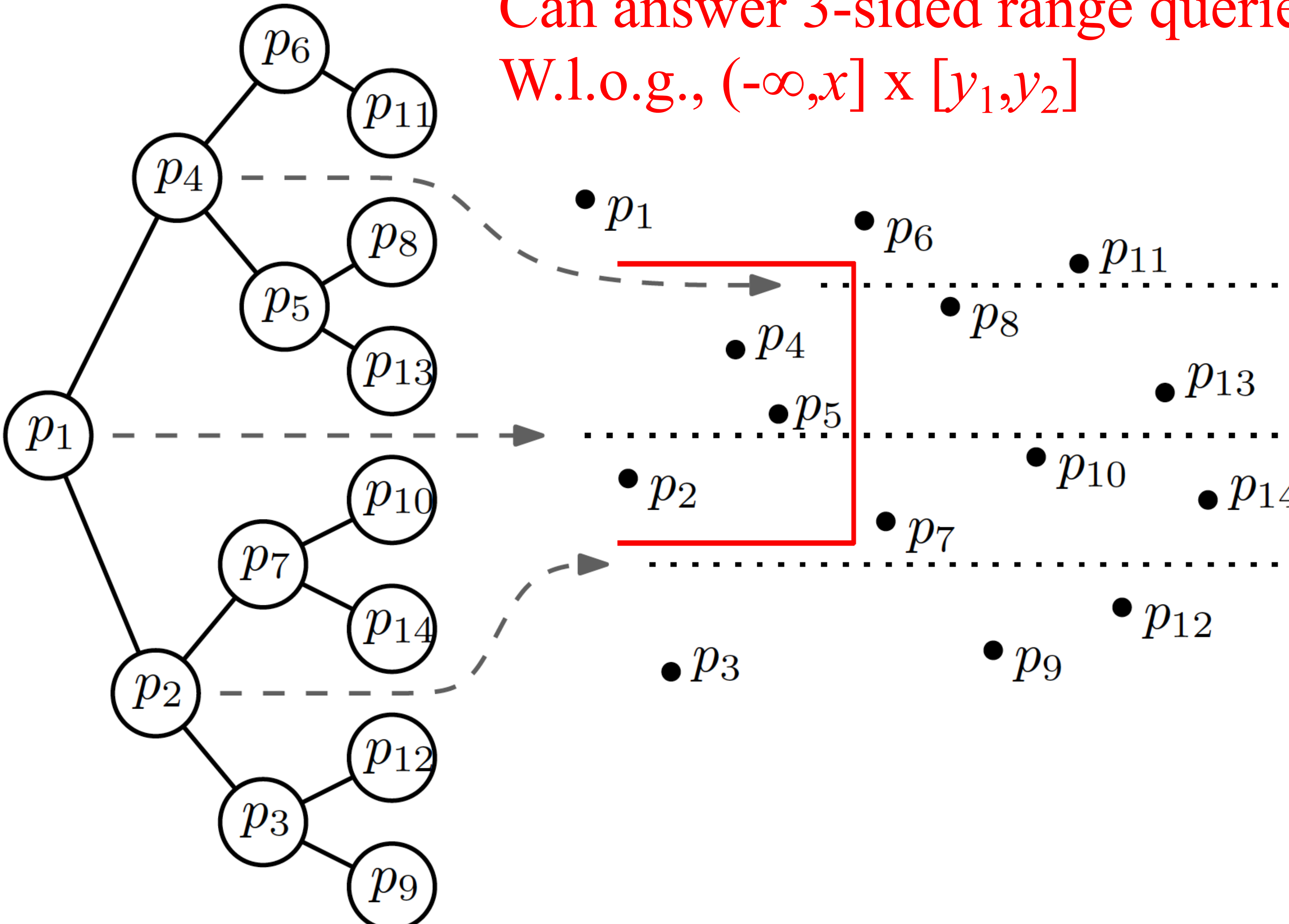
Can answer 3-sided range queries.

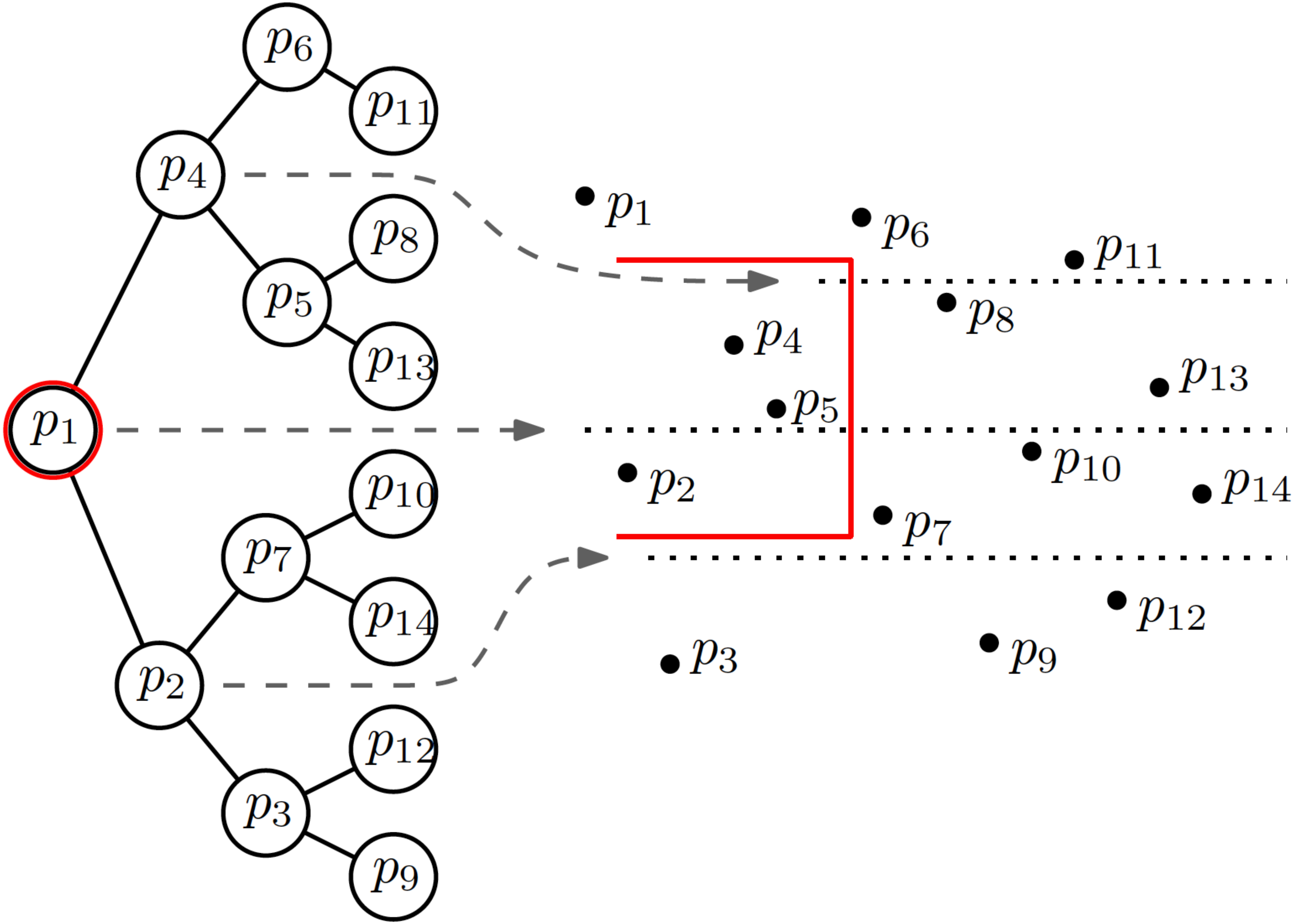
W.l.o.g., $(-\infty,x] \times [y_1,y_2]$



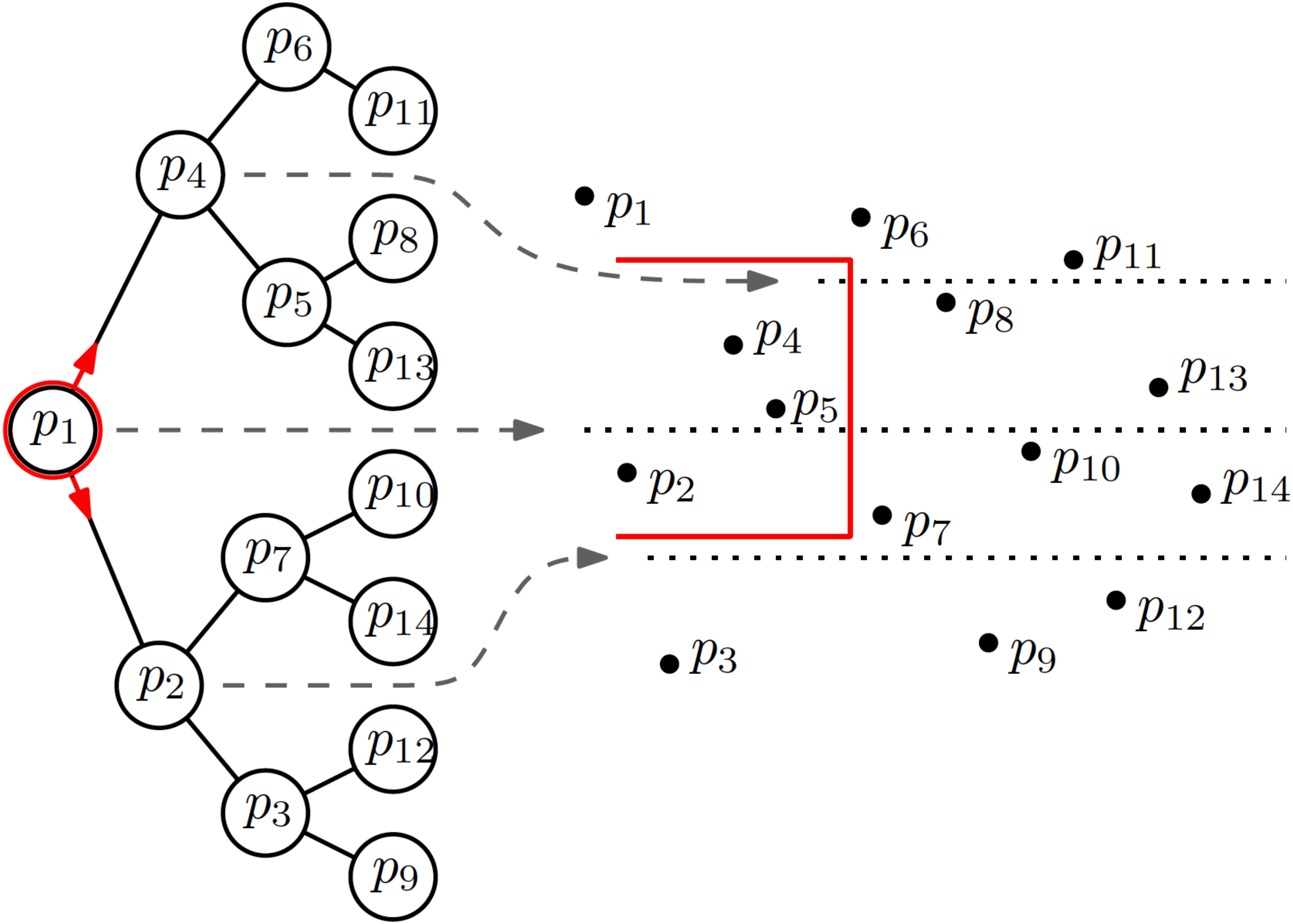
Can answer 3-sided range queries.

W.l.o.g., $(-\infty, x] \times [y_1, y_2]$

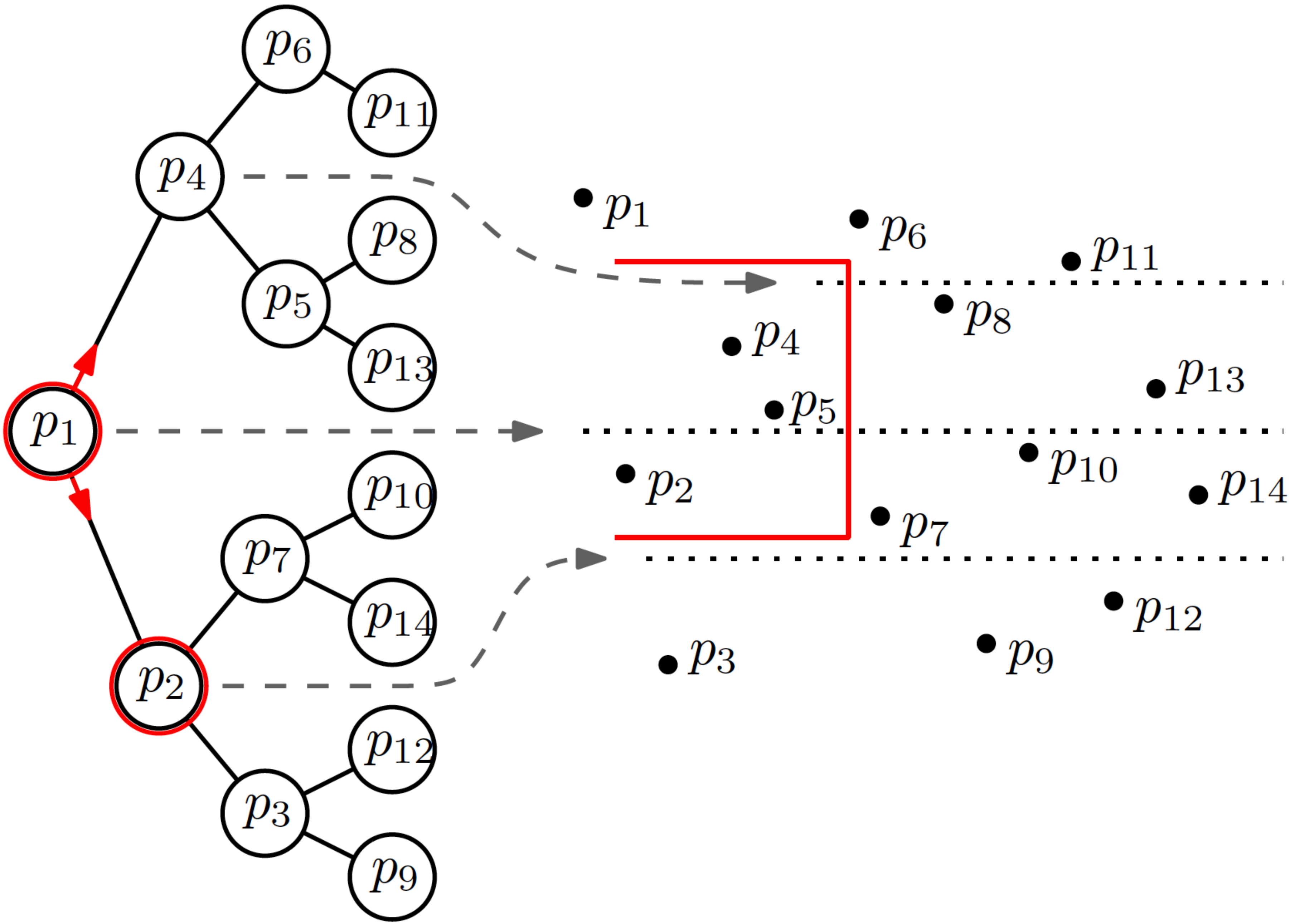


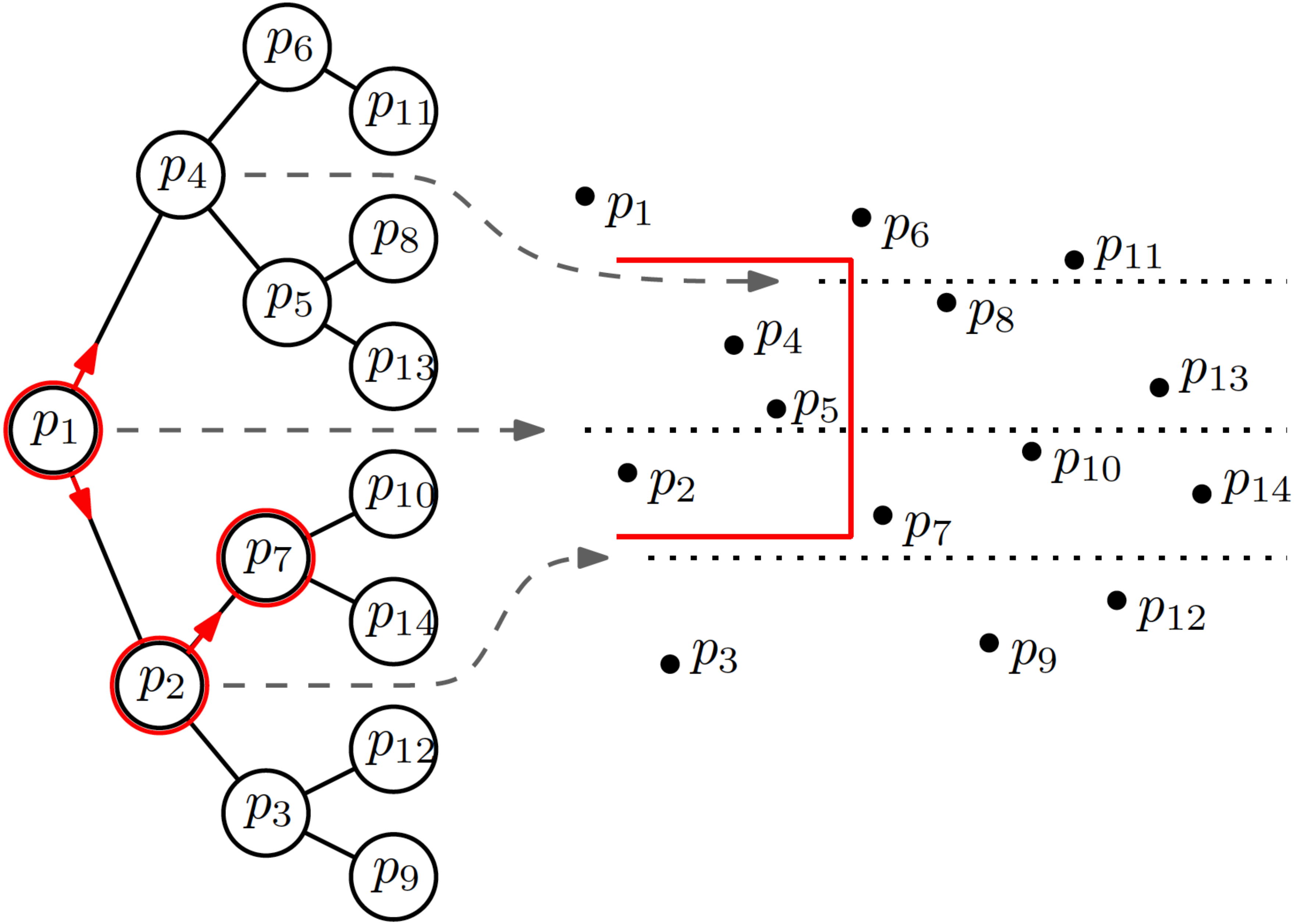


p_6
p_{11}
p_4
p_8
p_5
p_{13}
p_1
p_{10}
p_7
p_{14}
p_2
p_{12}
p_3
p_9

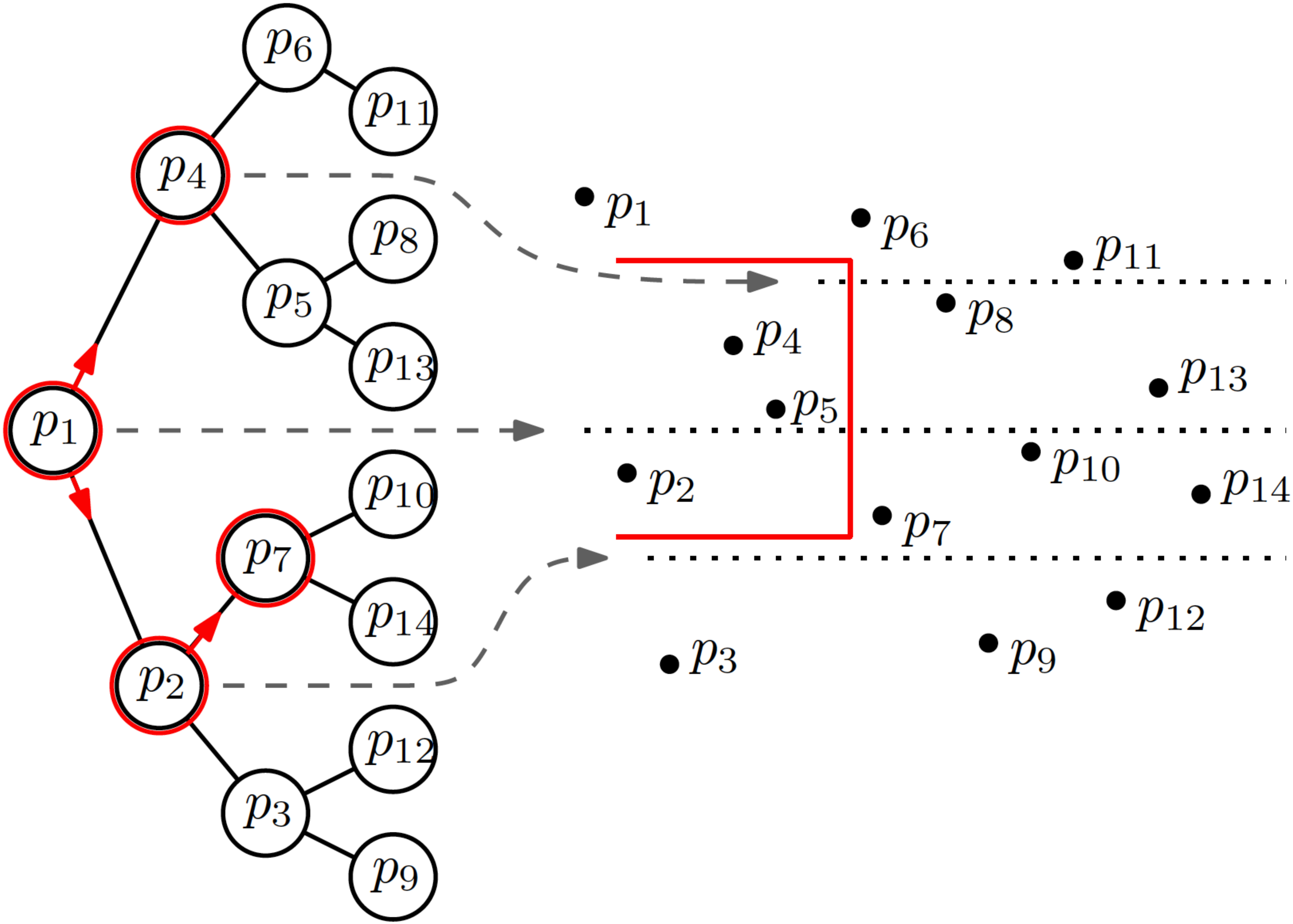


p_6
p_{11}
p_4
p_8
p_5
p_{13}
p_1
p_{10}
p_7
p_{14}
p_2
p_{12}
p_3
p_9
p_1
p_6
p_{11}
p_8
p_4
p_{13}
p_5
p_{10}
p_2
p_{14}
p_7
p_{12}
p_3
p_9

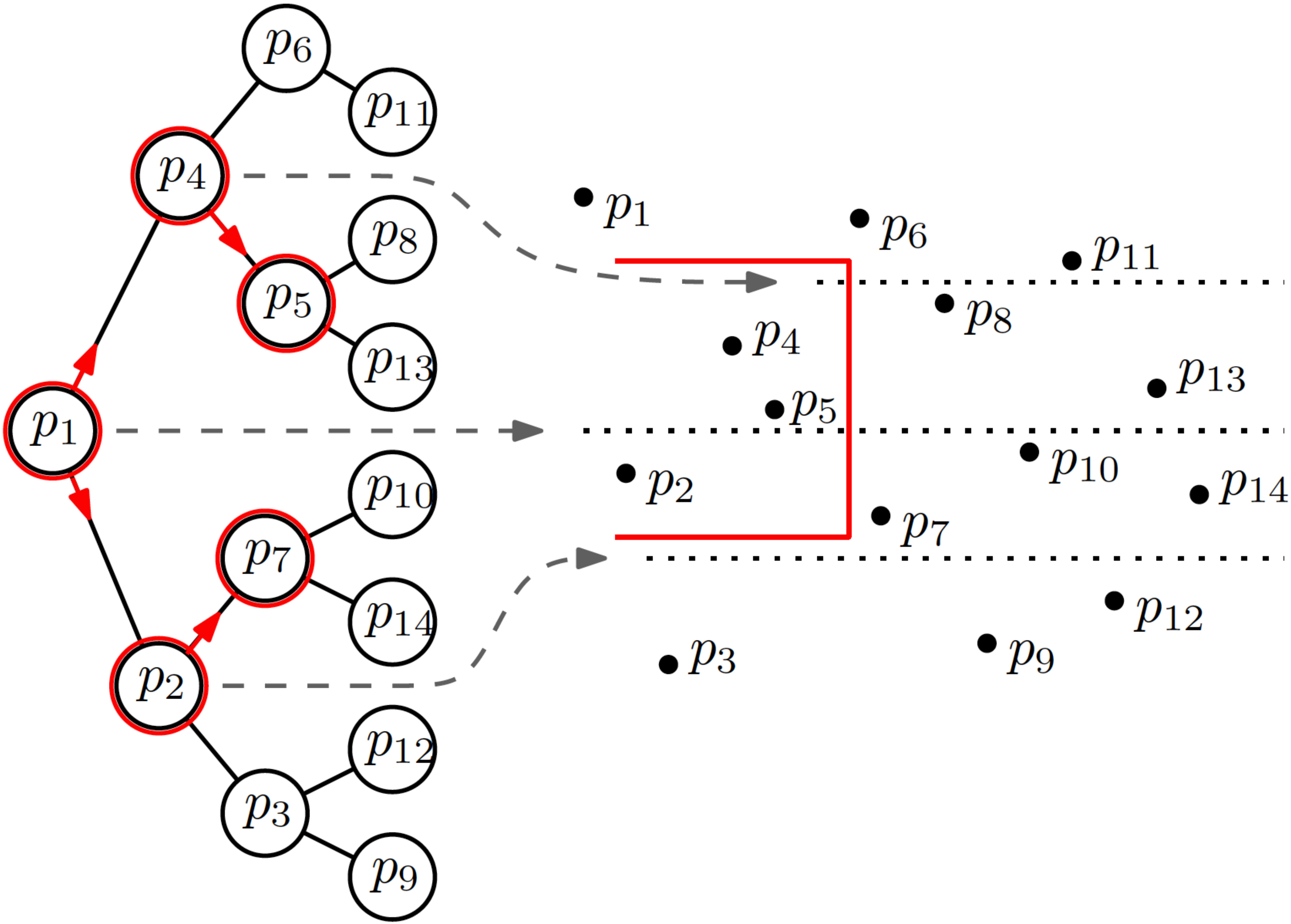

p_6
p_{11}
p_4
p_8
p_5
p_{13}
p_1
p_{10}
p_7
p_{14}
p_2
p_{12}
p_3
p_9
p_1
p_6
p_{11}
p_8
p_4
p_{13}
p_5
p_{10}
p_2
p_{14}
p_7
p_{12}
p_3
p_9



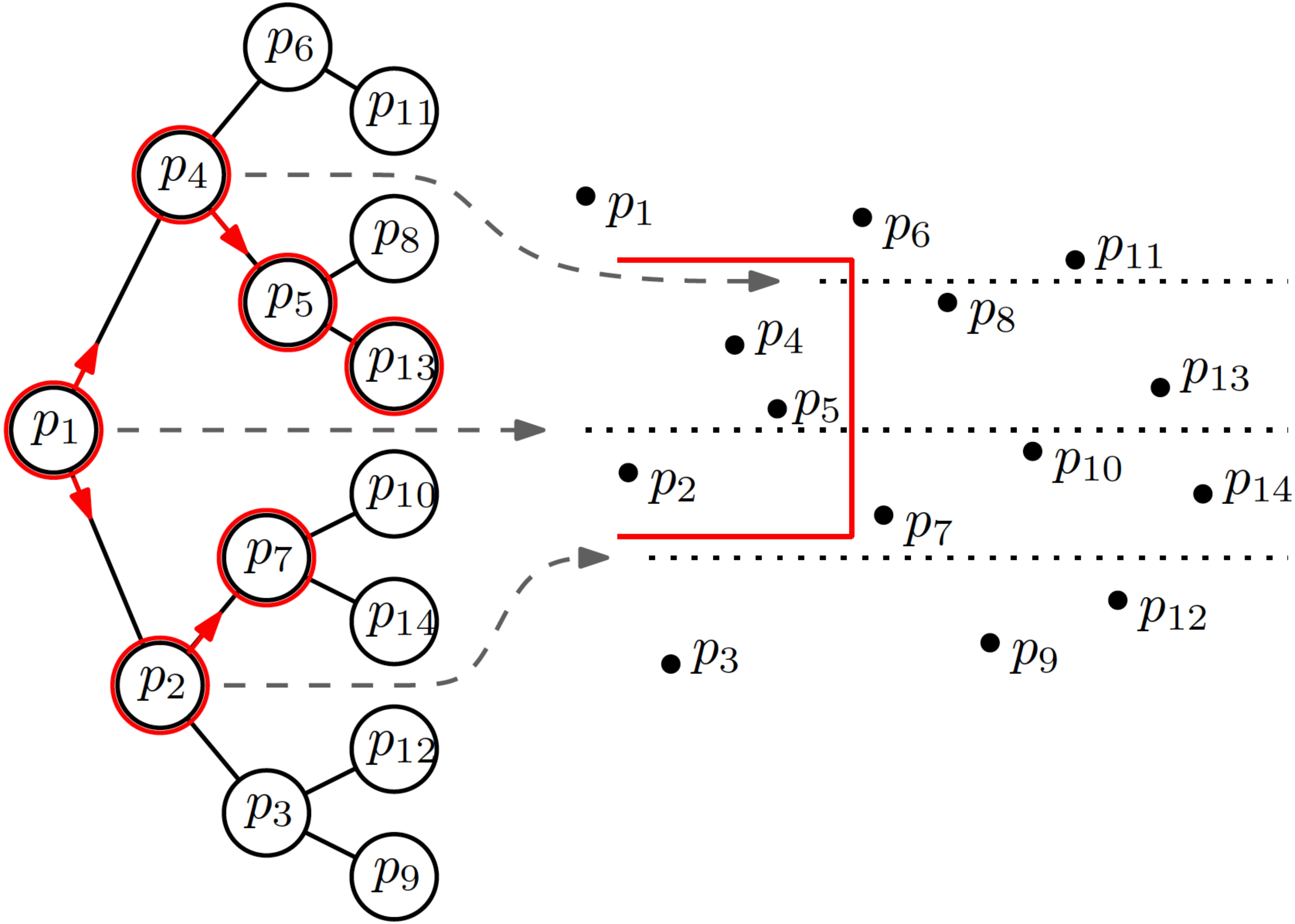
p_6
p_{11}
p_4
p_8
p_5
p_{13}
p_1
p_{10}
p_7
p_{14}
p_2
p_{12}
p_3
p_9
p_1
p_6
p_{11}
p_8
p_4
p_{13}
p_5
p_{10}
p_2
p_{14}
p_7
p_{12}
p_3
p_9



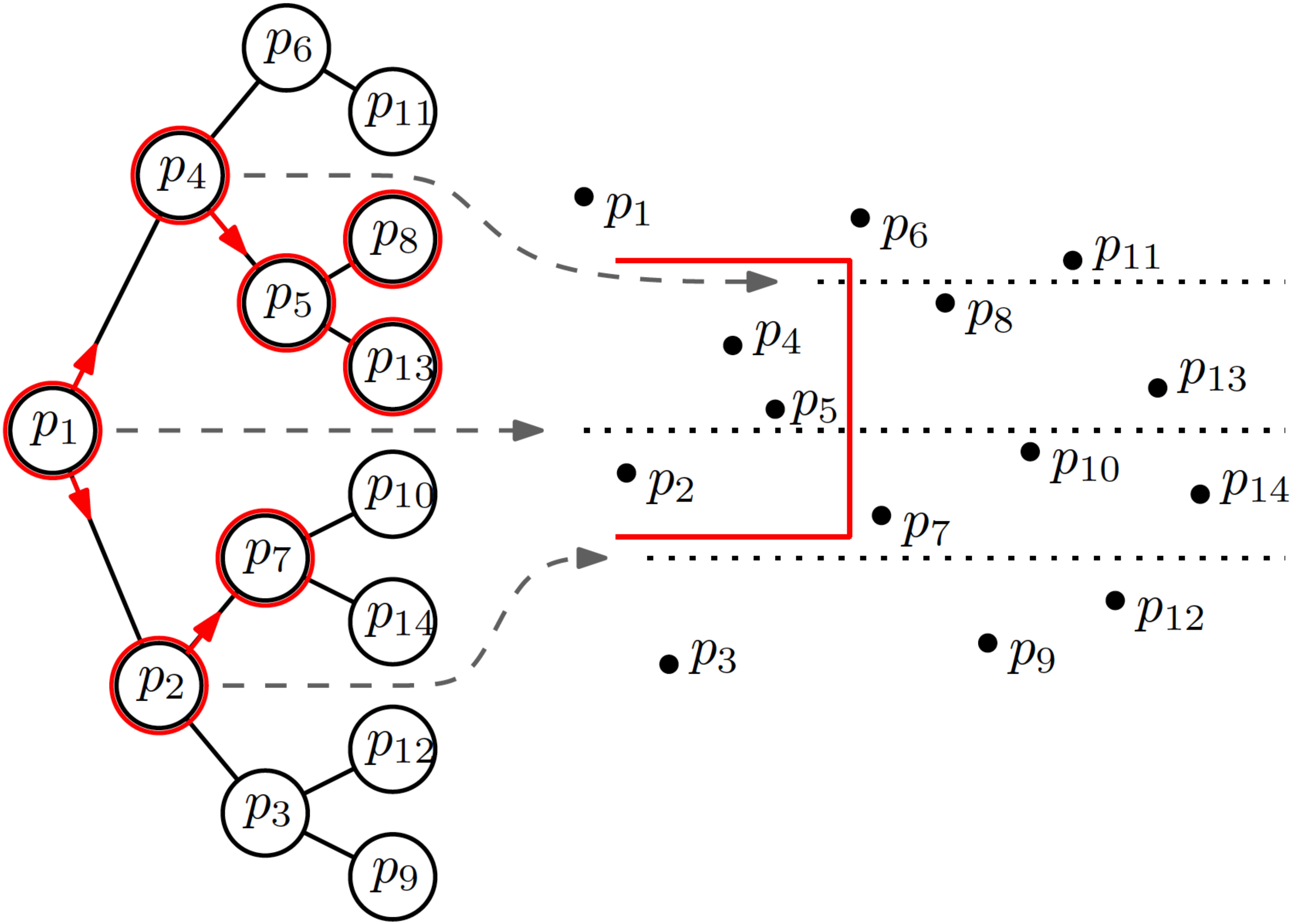
p_6
p_{11}
p_4
p_8
p_5
p_{13}
p_1
p_{10}
p_7
p_{14}
p_2
p_{12}
p_3
p_9
p_1
p_6
p_{11}
p_8
p_4
p_{13}
p_5
p_{10}
p_2
p_{14}
p_7
p_{12}
p_3
p_9



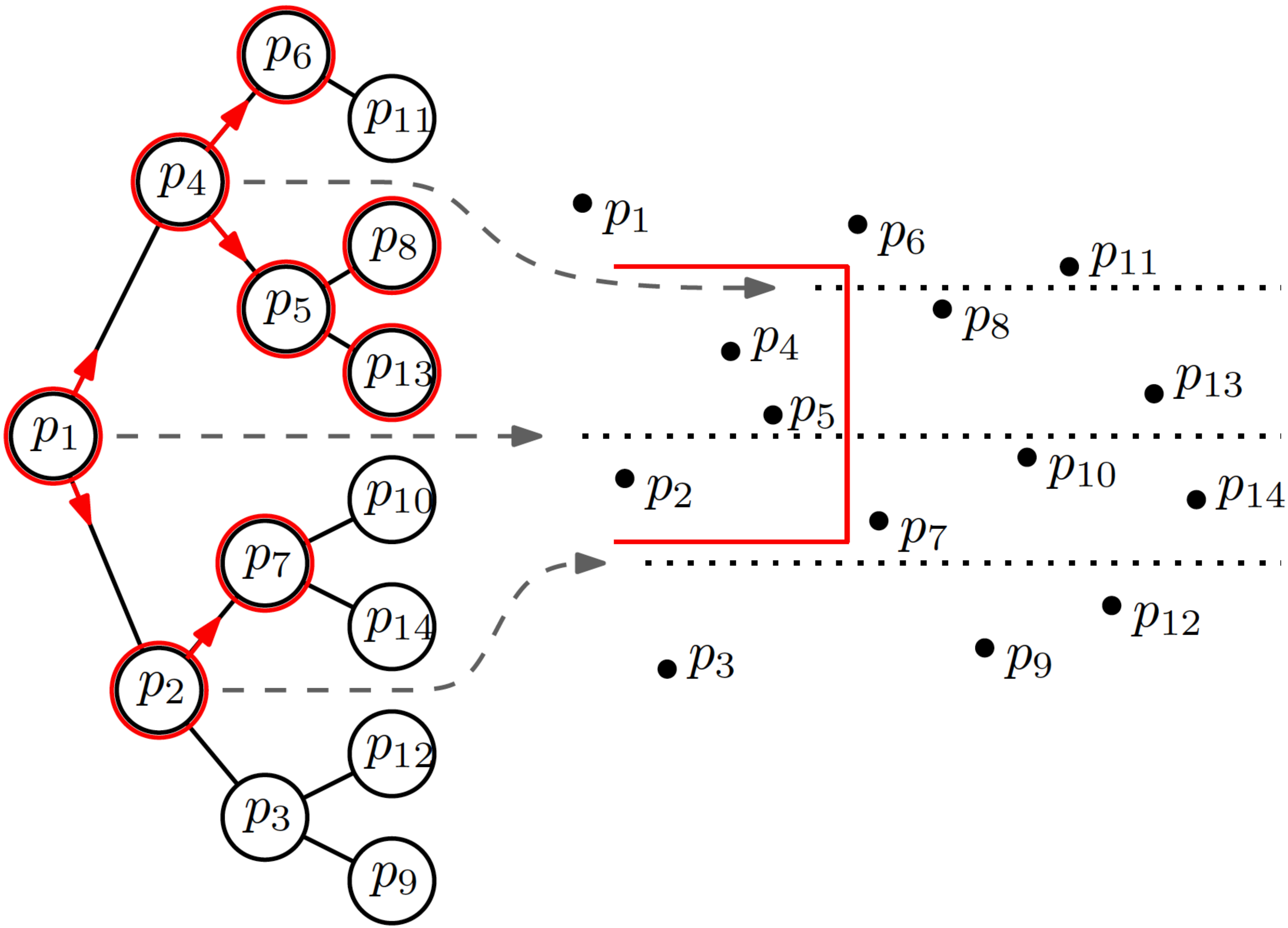
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
p_9
p_{10}
p_{11}
p_{12}
p_{13}
p_{14}



p_6
p_{11}
p_4
p_8
p_5
p_{13}
p_1
p_{10}
p_7
p_{14}
p_2
p_{12}
p_3
p_9
p_1
p_6
p_{11}
p_8
p_4
p_{13}
p_5
p_{10}
p_2
p_{14}
p_7
p_{12}
p_3
p_9



p6
p11
p4
p8
p5
p13
p1
p10
p7
p14
p2
p12
p3
p9
p1
p6
p11
p8
p4
p13
p5
p10
p2
p14
p7
p12
p3
p9



p_6
p_{11}
p_4
p_8
p_5
p_{13}
p_1
p_{10}
p_7
p_{14}
p_2
p_{12}
p_3
p_9
p_1
p_6
p_{11}
p_8
p_4
p_{13}
p_5
p_{10}
p_2
p_{14}
p_7
p_{12}
p_3
p_9

Query algorithm

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

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

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 $(p(v))_x \leq q_x$
2. then 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

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

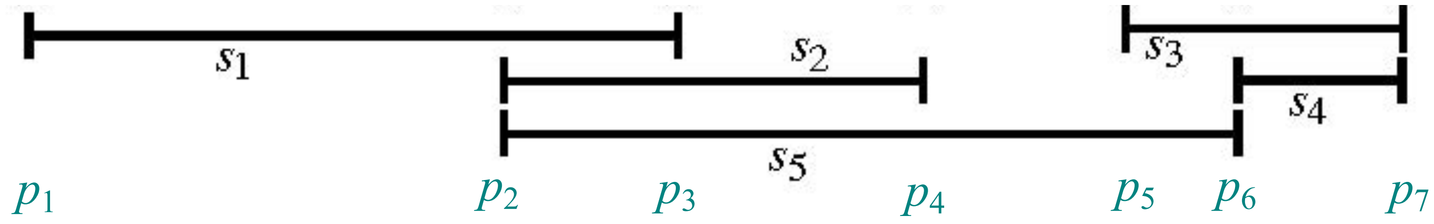
Theorem: A priority search tree for a set P of n points uses $O(n)$ storage and can be built in $O(n \log n)$ time. All points that lie in a 3-sided query range can be reported in $O(\log n + k)$ time, where k is the number of reported points

Segment Trees

- Let $I=\{s_1,\ldots,s_n\}$ be a set of n intervals (segments), and let $p_1, p_2, \ldots, p_m$ be the sorted list of distinct interval endpoints of I.
- Partition the real line into elementary intervals:

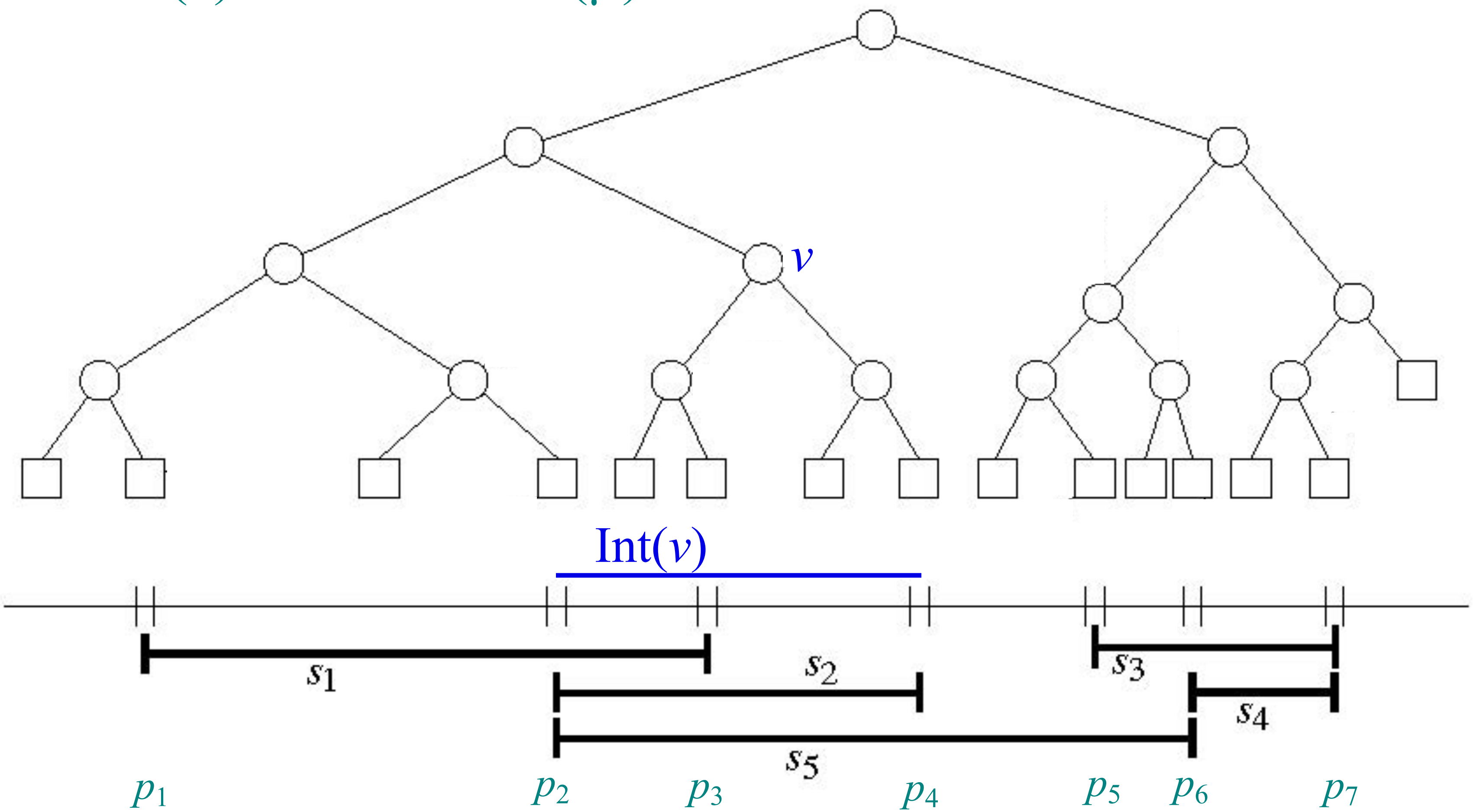
$$(-\infty, p_1), [p_1, p_1], (p_1, p_2), \ldots, (p_{m-1}, p_m), [p_m, p_m], (p_m, \infty)$$

- Construct a balanced binary search tree T with leaves corresponding to the elementary intervals



Elementary Intervals

- Int(μ):=elementary interval corresponding to leaf μ
- Int(v):=union of Int(μ) of all leaves in subtree rooted at v

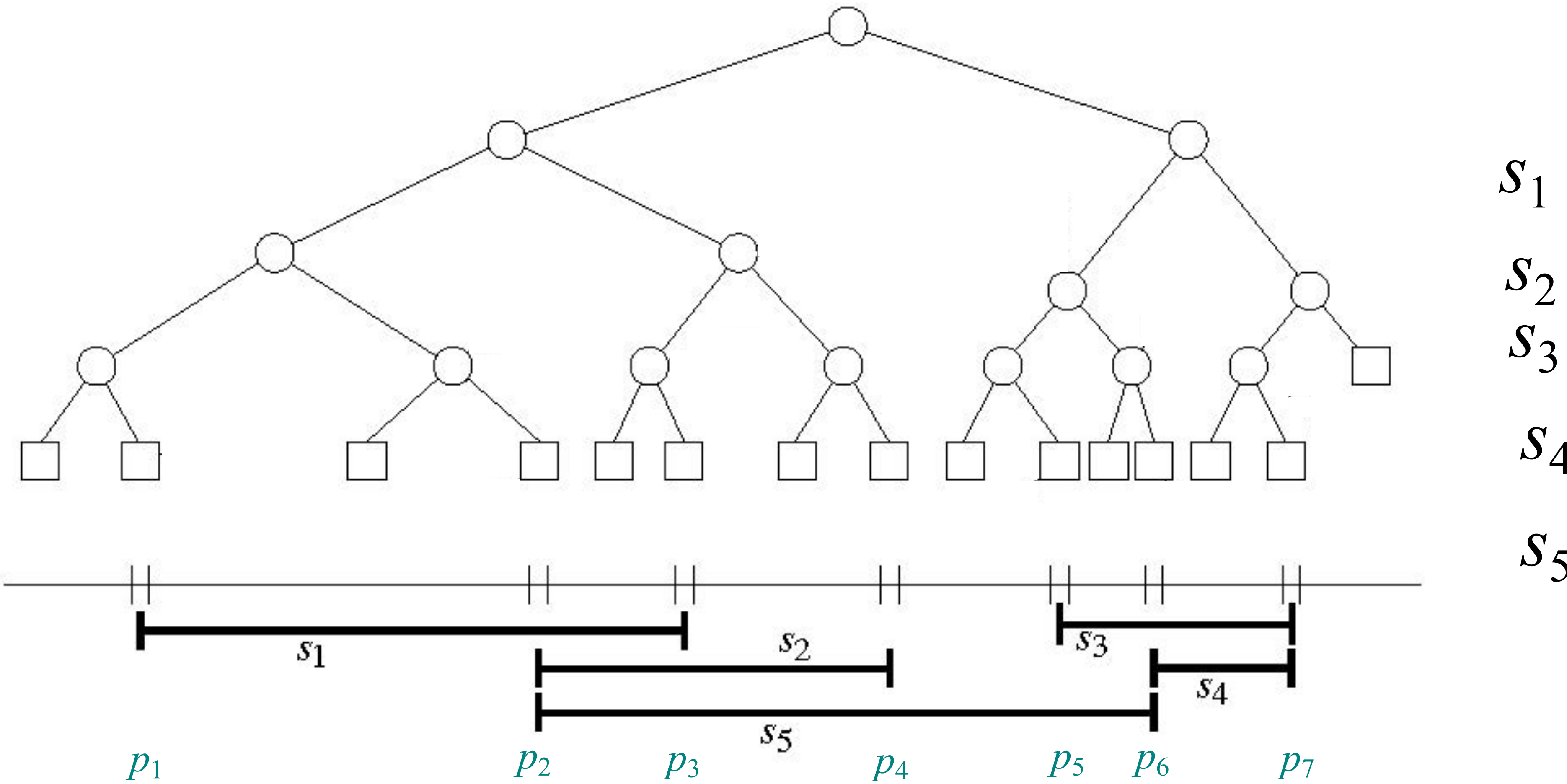


Segment Trees

Store segments as high as possible

Each vertex v stores (1) Int(v) and (2) the canonical subset I(v)⊆I:

$$I(v) := \{s \in I \mid Int(v) \subseteq s \text{ and } Int(parent(v)) \not\subset s\}$$

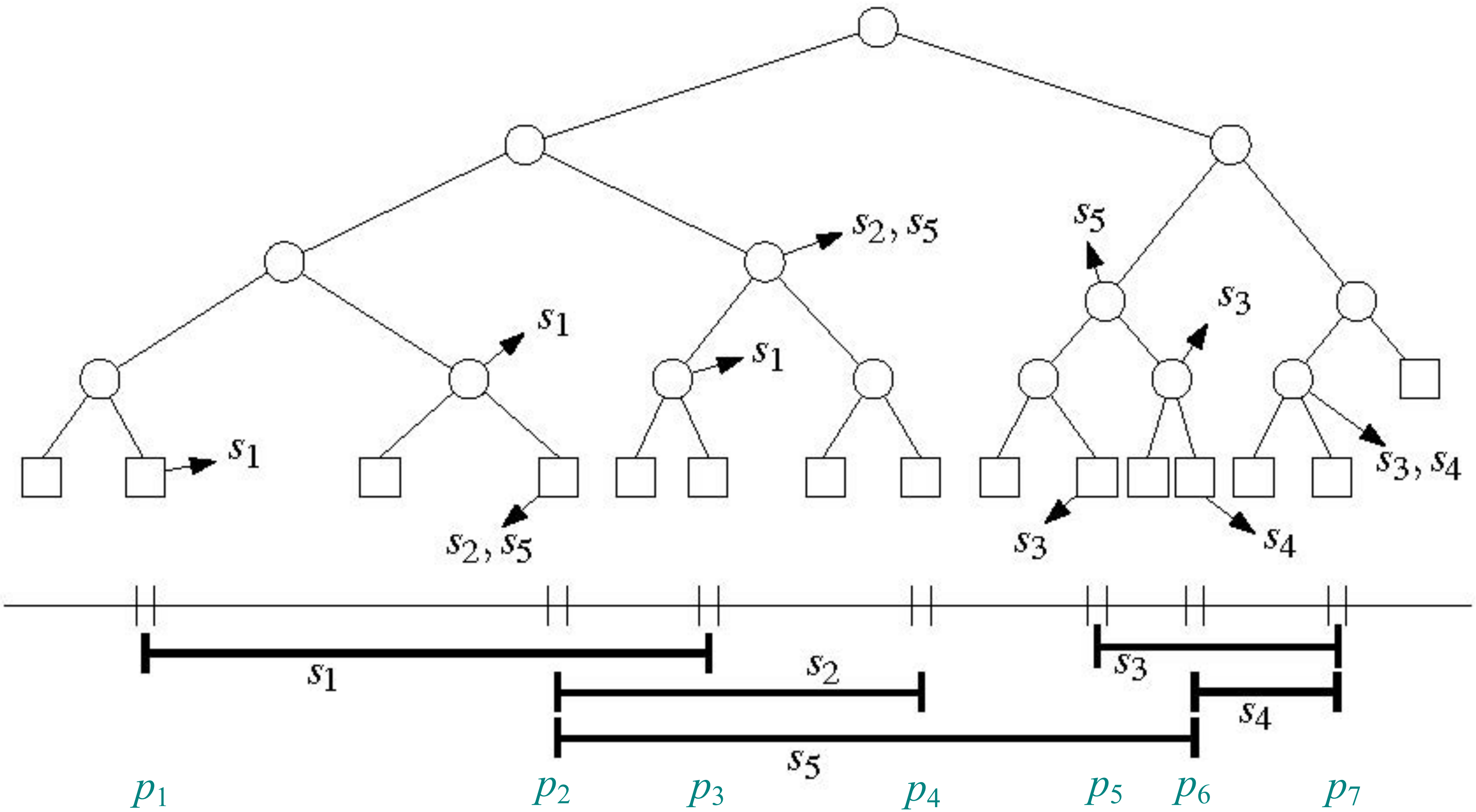


Segment Trees

Store segments as high as possible

Each vertex v stores (1) Int(v) and (2) the canonical subset I(v)⊆I:

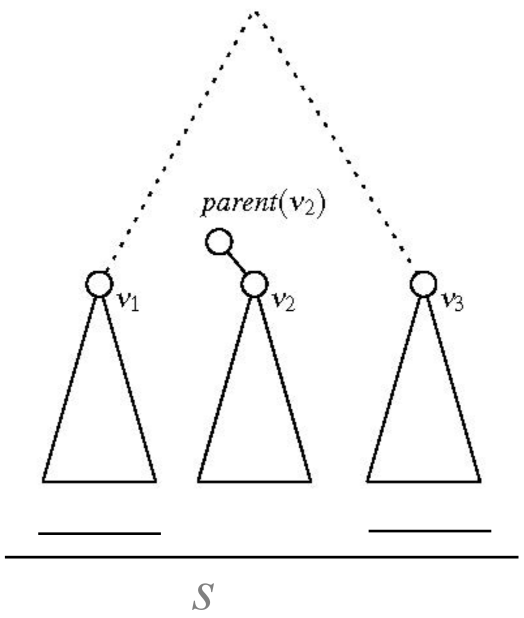
$$I(v) := \{ s \in I \mid Int(v) \subseteq s \text{ and } Int(parent(v)) \not\subseteq s \}$$



Space

Lemma: A segment tree on n intervals uses $O(n \log n)$ space.

Proof: Any interval s is stored in at most two sets $I(v_1)$, $I(v_2)$ for two different vertices v_1, v_2 at the same level of T. [If s was stored in $I(v_3)$ for a third vertex v_3, then s would have to span from left to right, and $Int(parent(v_2)) \subseteq s$, hence s is cannot be stored in v_2.]
The tree is a balanced tree of height $O(\log n)$.



□

Segment Tree Query

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

Runtime Analysis:

- Visit one node per level.
- Spend $O(1+k_v)$ time per node v.

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

Segment Tree Construction

$O(n \log n)$ {steps 1–3}

1. Sort interval endpoints of I. $\rightarrow$ elementary intervals
2. Construct balanced BST on elementary intervals.
3. Determine Int(v) bottom-up.
4. Compute canonical subsets by incrementally inserting intervals $s=[x,x']\in I$ into T using InsertSegmentTree:

Algorithm INSERTSEGMENTTREE(v, s)
Input. The root of a (subtree of a) segment tree and an interval.
Output. The interval will be stored in the subtree.

```
if Int(v) ⊆ s
    then store s at v
    else if Int(lc(v)) ∩ s ≠ ∅
            then INSERTSEGMENTTREE(lc(v), s)
         if Int(rc(v)) ∩ s ≠ ∅
            then INSERTSEGMENTTREE(rc(v), s)
```

Segment Trees

Runtime:

- Each interval stored at most twice per level
- At most one node per level that contains the left endpoint of s (same with right endpoint)

→ Visit at most 4 nodes per level

→ $O(\log n)$ per interval, and $O(n \log n)$ total

Theorem: A segment tree for a set of n intervals can be built in $O(n \log n)$ time and uses $O(n \log n)$ space. All intervals that contain a query point can be reported in $O(\log n + k)$ time.

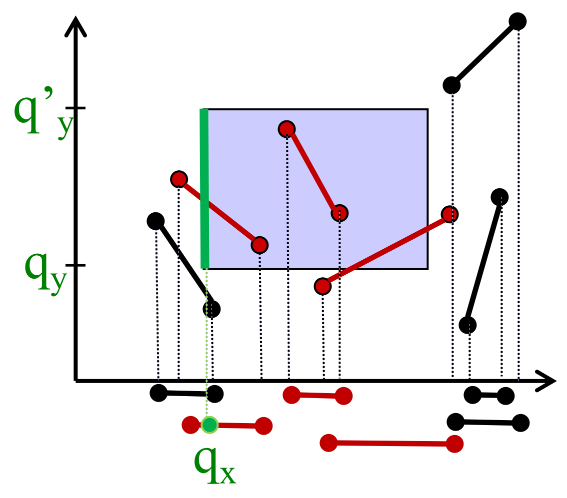
2D Windowing Revisited

Input: A set S of n disjoint line segments in the plane

Task: Process S into a data structure such that all segments intersecting a

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

can be reported efficiently.

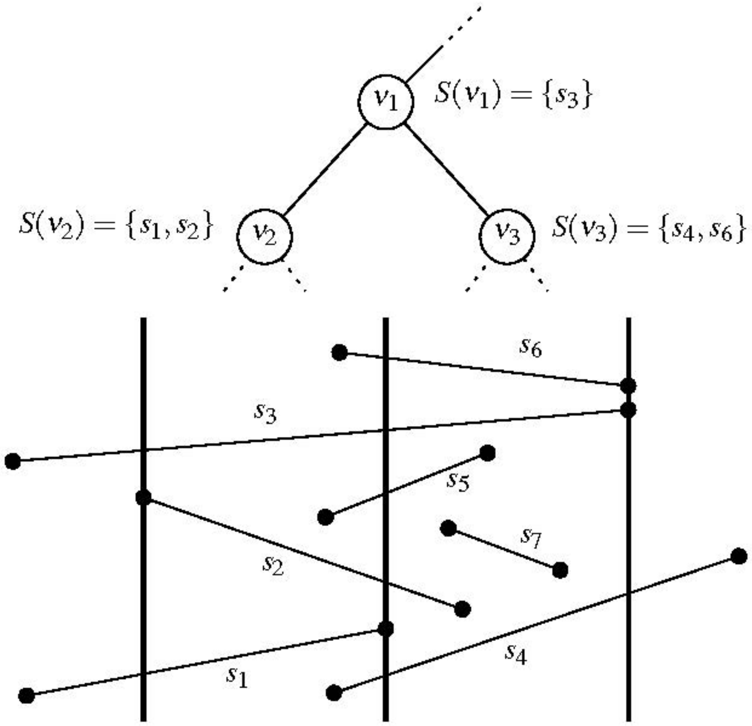


2D Windowing Revisited

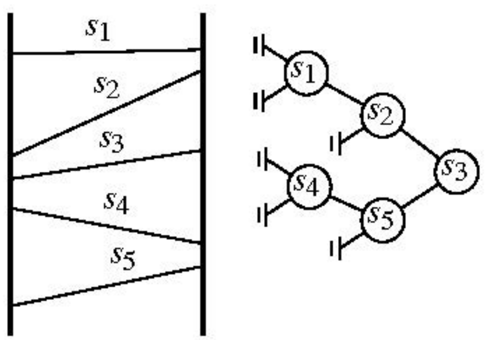
Solution:

Segment tree with nested range tree

- Build segment tree *T* based on *x*-intervals of segments in *S*.
  → each Int(v)≅ Int(v)×(-∞,∞) vertical slab
- I(v)≅S(v) canonical set of segments spanning vertical slab



- Store S(v) in 1D range tree (binary search tree) T(v) based on vertical order of segments



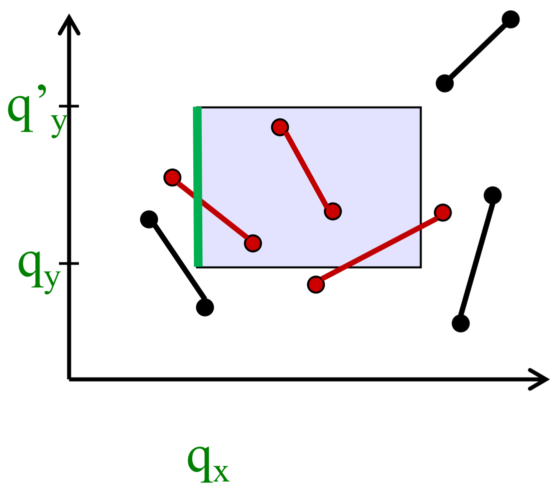
2D Windowing Revisited

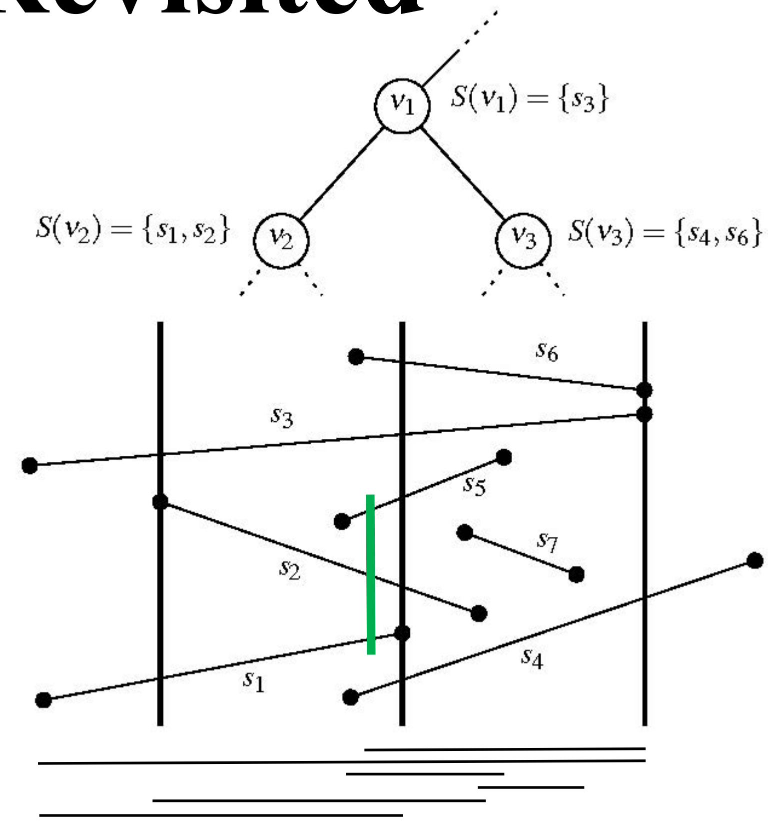
Query algorithm:

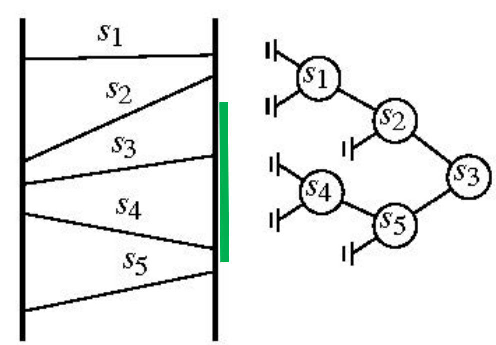
- Search regularly for q_x in T
- In every visited vertex v report segments in $T(v)$ between q_y and q'_y (1D range query)

$\Rightarrow O(\log n + k_v)$ time for $T(v)$

$\Rightarrow O(\log^2 n + k)$ total







2D Windowing Summary

Theorem: Let S be a set of (interior-) disjoint line segments in the plane. The segments intersecting a vertical query segment (or an axis-parallel rectangular query window) can be reported in $O(\log^2 n + k)$ time, with $O(n \log n)$ preprocessing time and $O(n \log n)$ space.