DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

WINDOWING

PETR FELKEL

FEL CTU PRAGUE

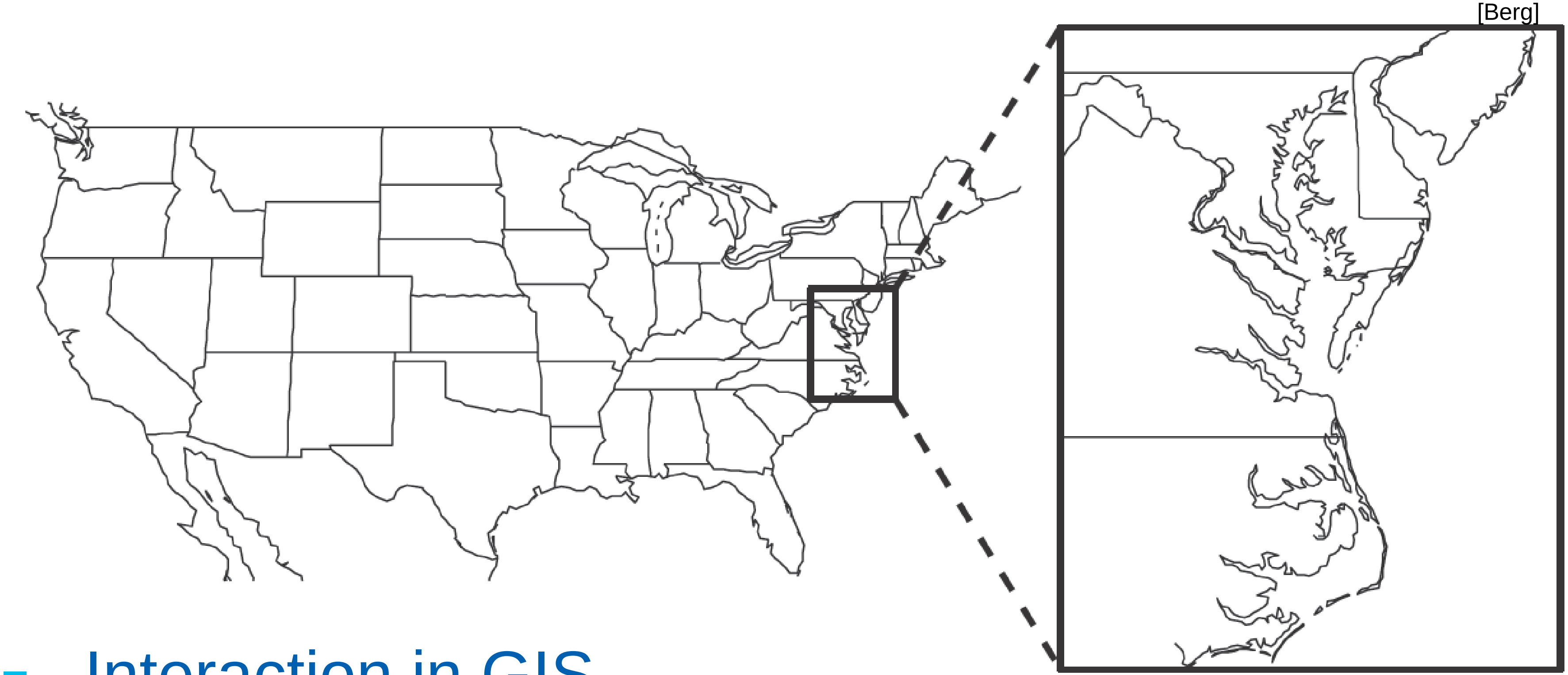
felkel@fel.cvut.cz

https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start

Based on [Berg], [Mount]

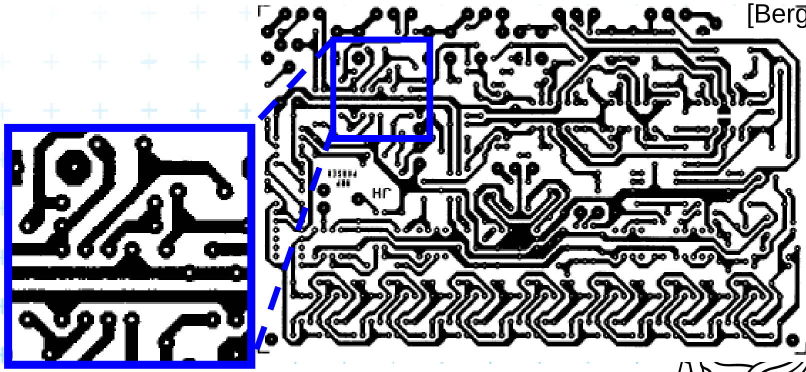
Version from 30.11.2022

Windowing queries - examples



[Berg]

- Interaction in GIS
  - Select subset by outlining
  - Zoom in and re-center
- Circuit board inspection,…



[Berg]

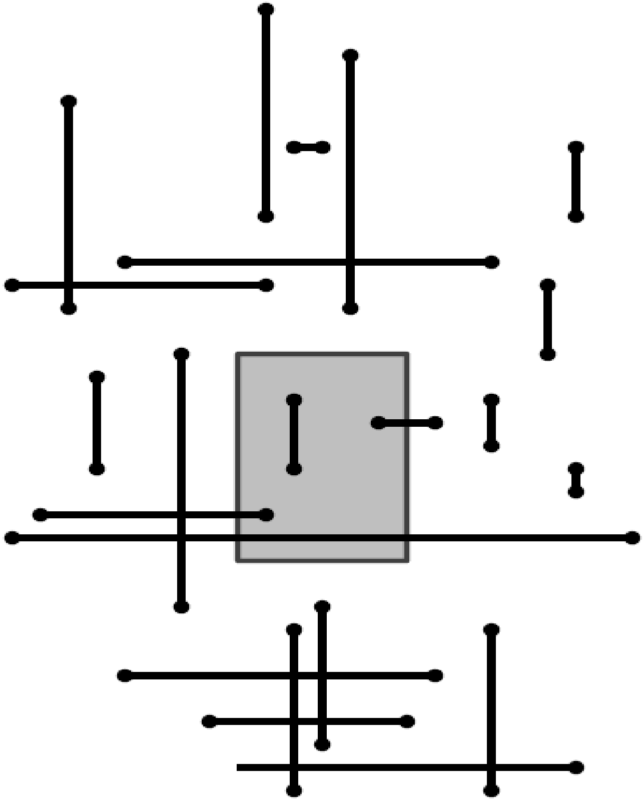
[Vakken]

Windowing versus range queries

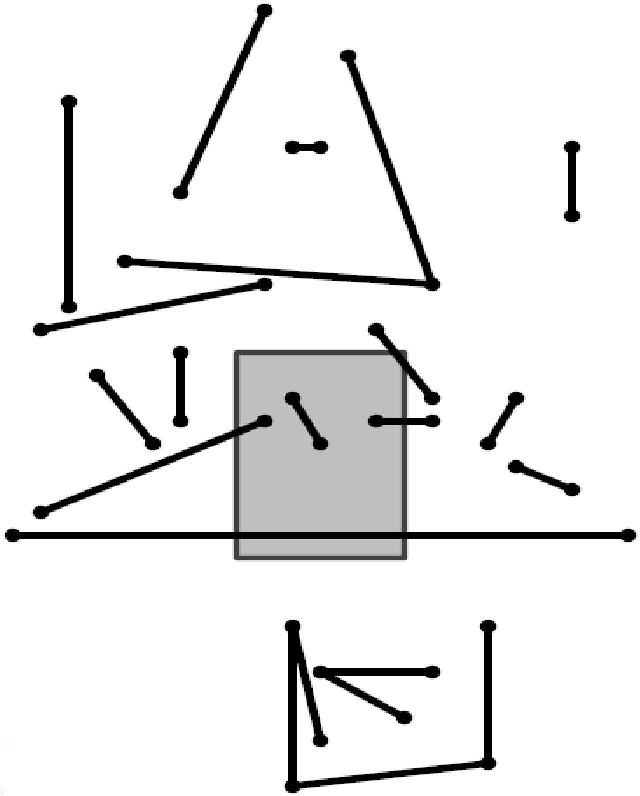
- Range queries (see range trees in Lecture 03)
  - Points
  - Often in higher dimensions
- Windowing queries
  - Line segments, curves, …
  - Usually in low dimension (2D, 3D)

- The goal for both:
  Preprocess the data into a data structure
  - so that the objects intersected by the query rectangle can be reported efficiently

Windowing queries on line segments



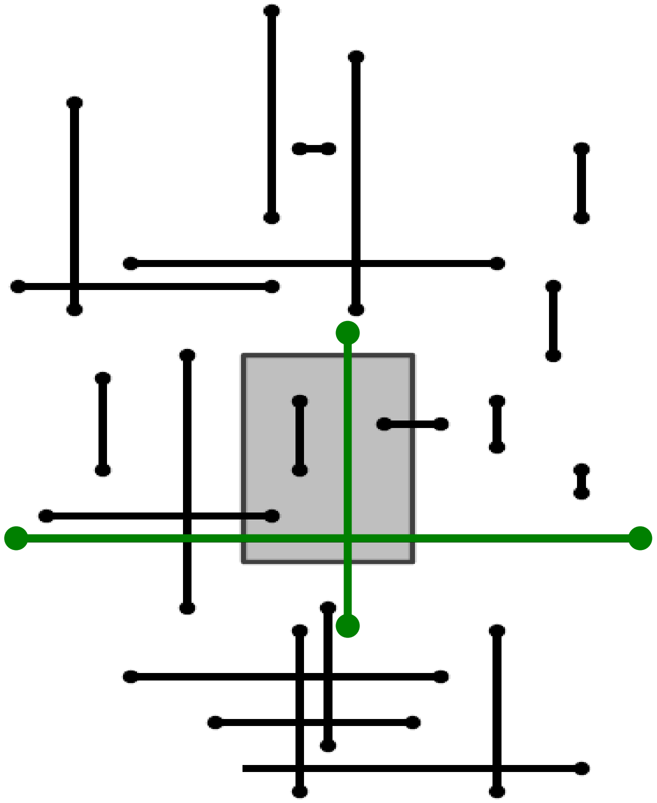
1. Axis parallel line segments



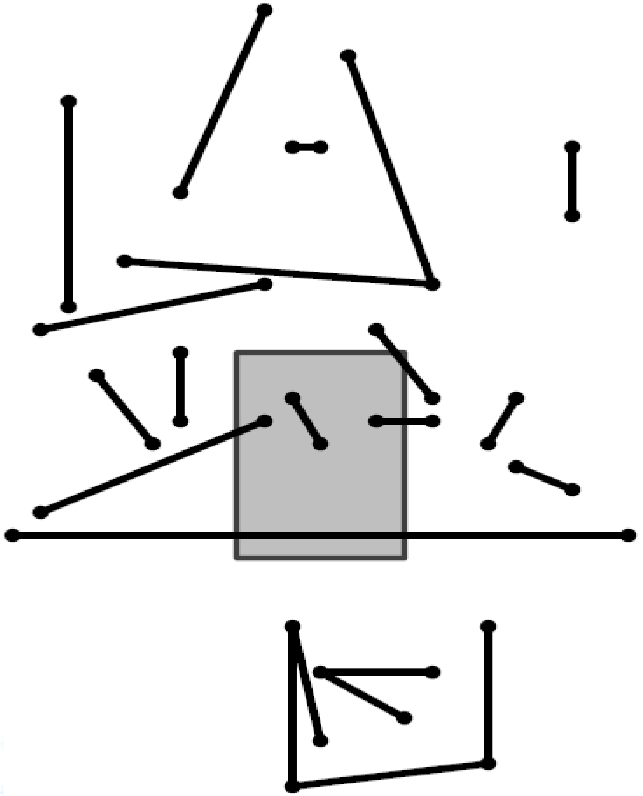
2. Arbitrary line segments (non-crossing)

[Vakken]

Windowing queries on line segments



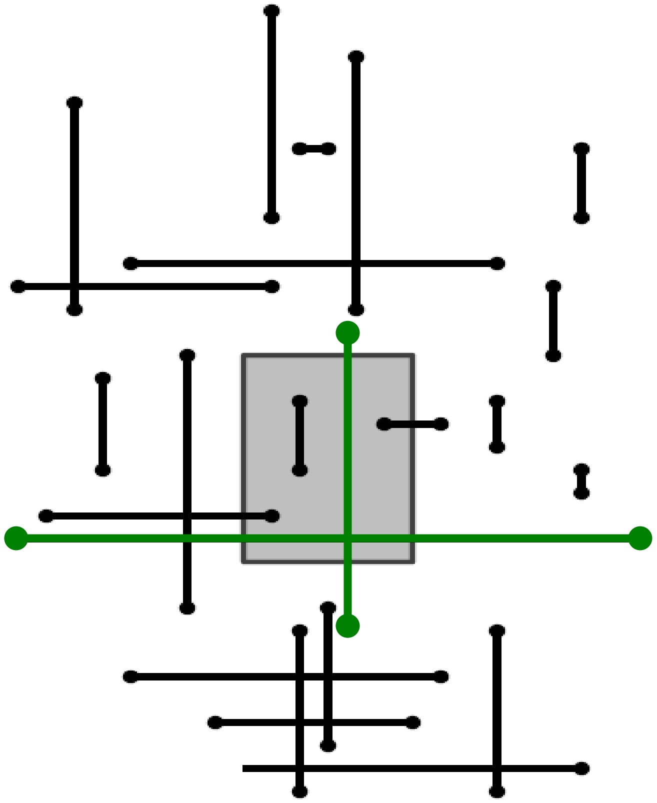
1. Axis parallel line segments



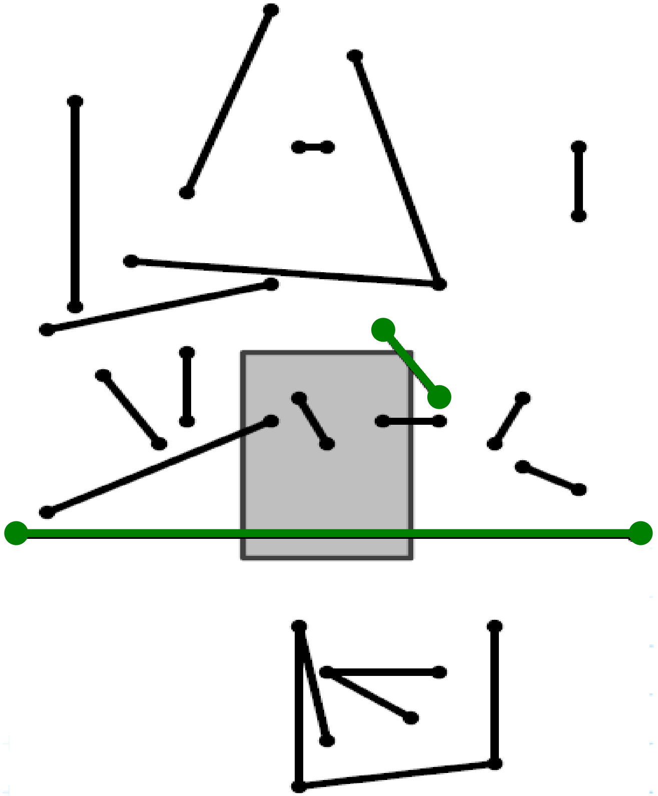
2. Arbitrary line segments (non-crossing)

[Vakken]

Windowing queries on line segments



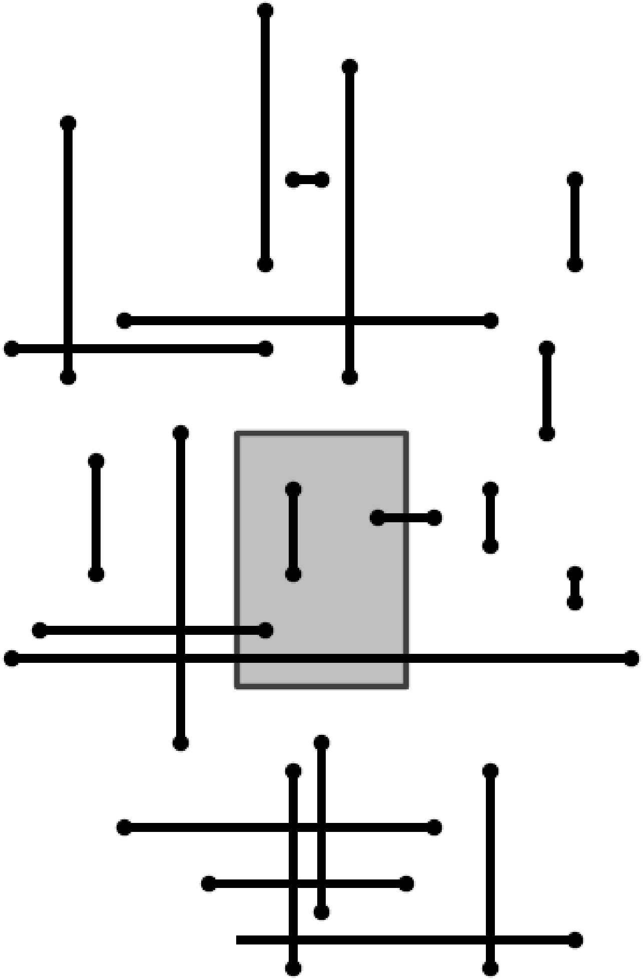
1. Axis parallel line segments



2. Arbitrary line segments (non-crossing)

[Vakken]

1. Windowing of axis parallel line segments

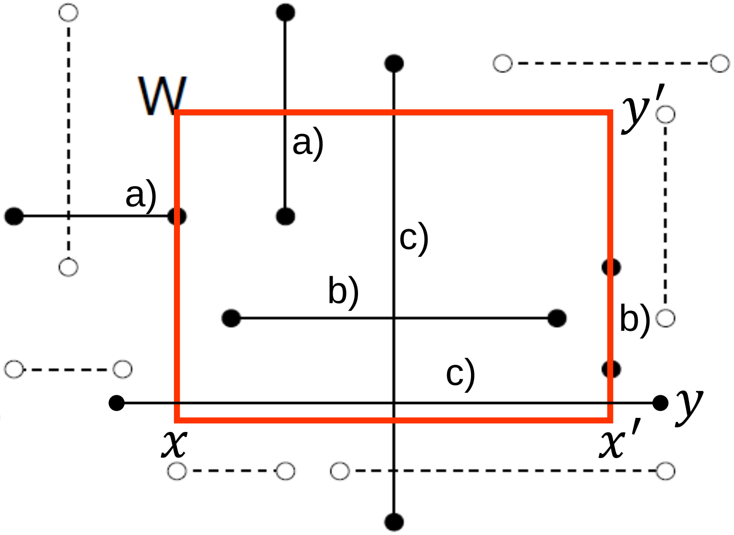


[Vakken]

1. Windowing of axis parallel line segments

Window query

- Given
  - a set of orthogonal line segments S (preprocessed),
  - and orthogonal query rectangle $W = [x : x'] \times [y : y']$
- Count or report all the line segments of S that intersect W
- Such segments have
  a) one endpoint in
  b) two end points in – included
  c) no end point in – cross over

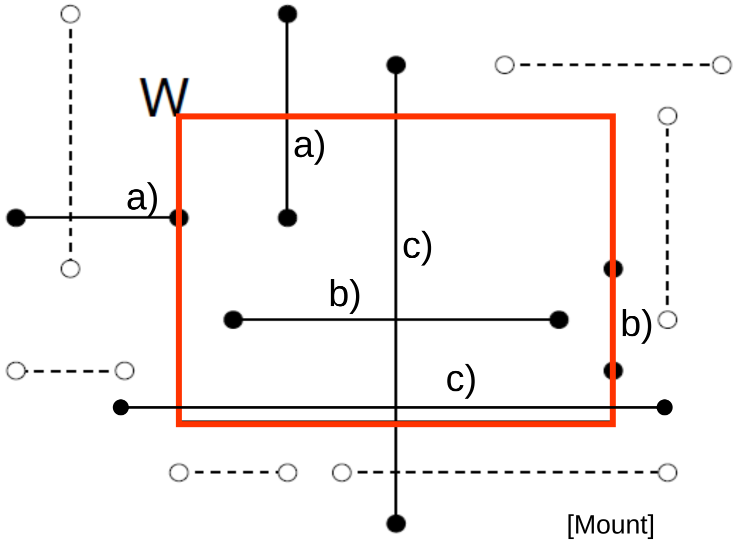


[Mount]

Line segments with 1 or 2 points inside

a) one point inside

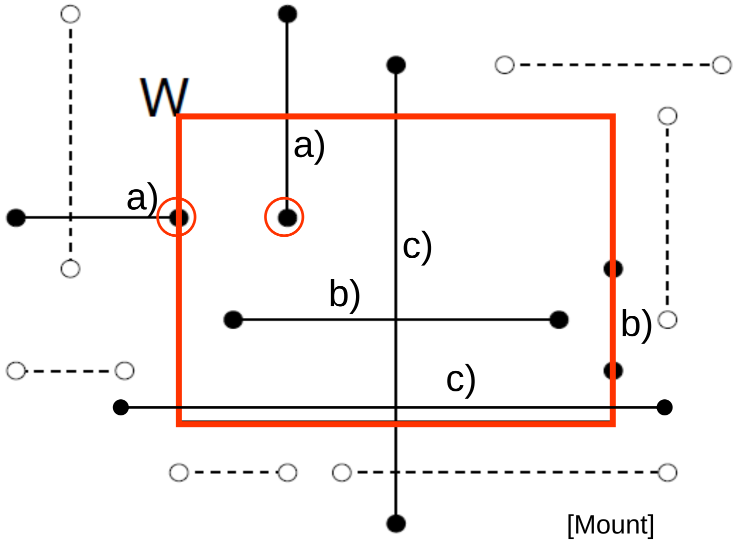
- Use a 2D range tree (lesson 3)
- $O(n \log n)$ storage
- $O(\log^2 n + k)$ query time or
- $O(\log n + k)$ with fractional cascading



Line segments with 1 or 2 points inside

a) one point inside

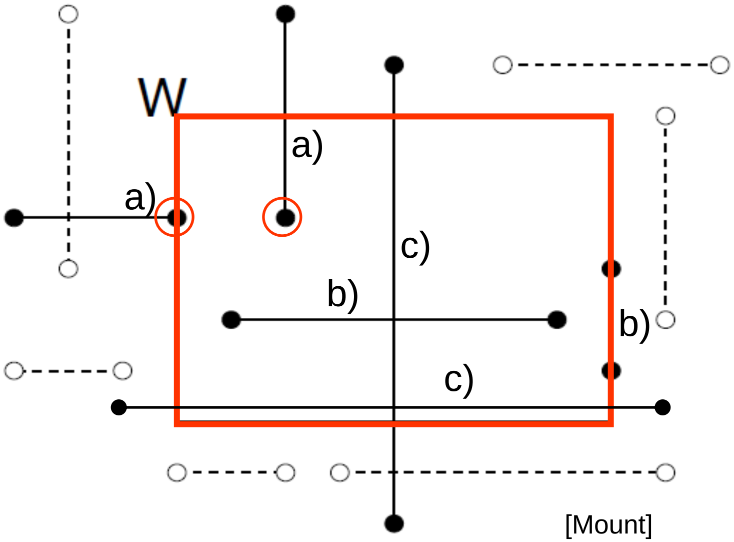
- Use a 2D range tree (lesson 3)
- $O(n \log n)$ storage
- $O(\log^2 n + k)$ query time or
- $O(\log n + k)$ with fractional cascading



Line segments with 1 or 2 points inside

a) one point inside

- Use a 2D range tree (lesson 3)
- $O(n \log n)$ storage
- $O(\log^2 n + k)$ query time or
- $O(\log n + k)$ with fractional cascading



b) two points inside – as a) one point inside

- Avoid reporting twice:

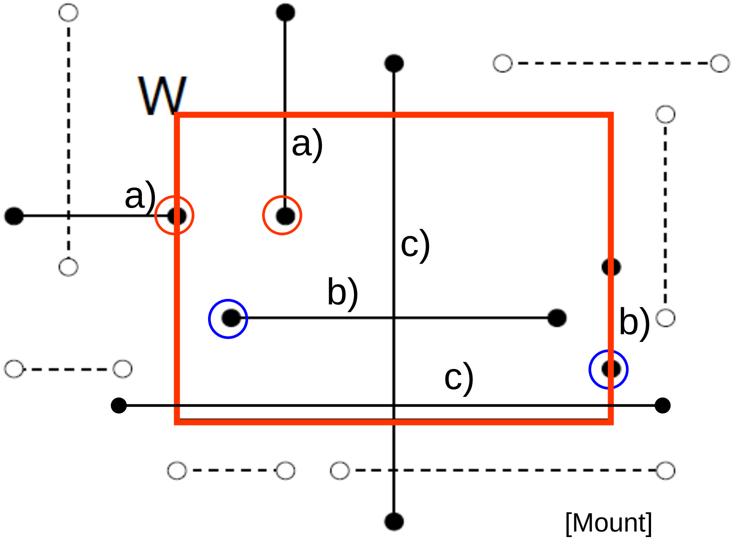
→ Mark segment when reported (clear after the query) and skip marked segments or

when end point found, check the other end-point and report only one of them (the leftmost or the bottom).

Line segments with 1 or 2 points inside

a) one point inside

- Use a 2D range tree (lesson 3)
- $O(n \log n)$ storage
- $O(\log^2 n + k)$ query time or
- $O(\log n + k)$ with fractional cascading



b) two points inside – as a) one point inside

- Avoid reporting twice:

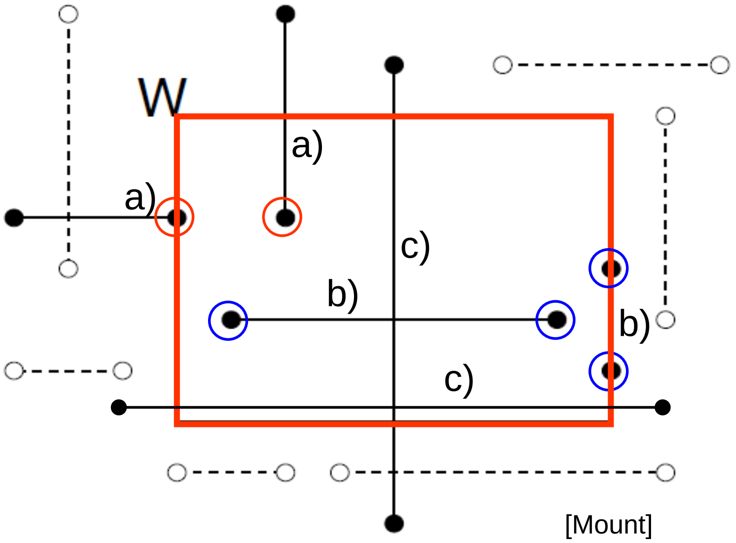
→ Mark segment when reported (clear after the query) and skip marked segments or

when end point found, check the other end-point and report only one of them (the leftmost or the bottom)

Line segments with 1 or 2 points inside

a) one point inside

- Use a 2D range tree (lesson 3)
- $O(n \log n)$ storage
- $O(\log^2 n + k)$ query time or
- $O(\log n + k)$ with fractional cascading



b) two points inside – as a) one point inside

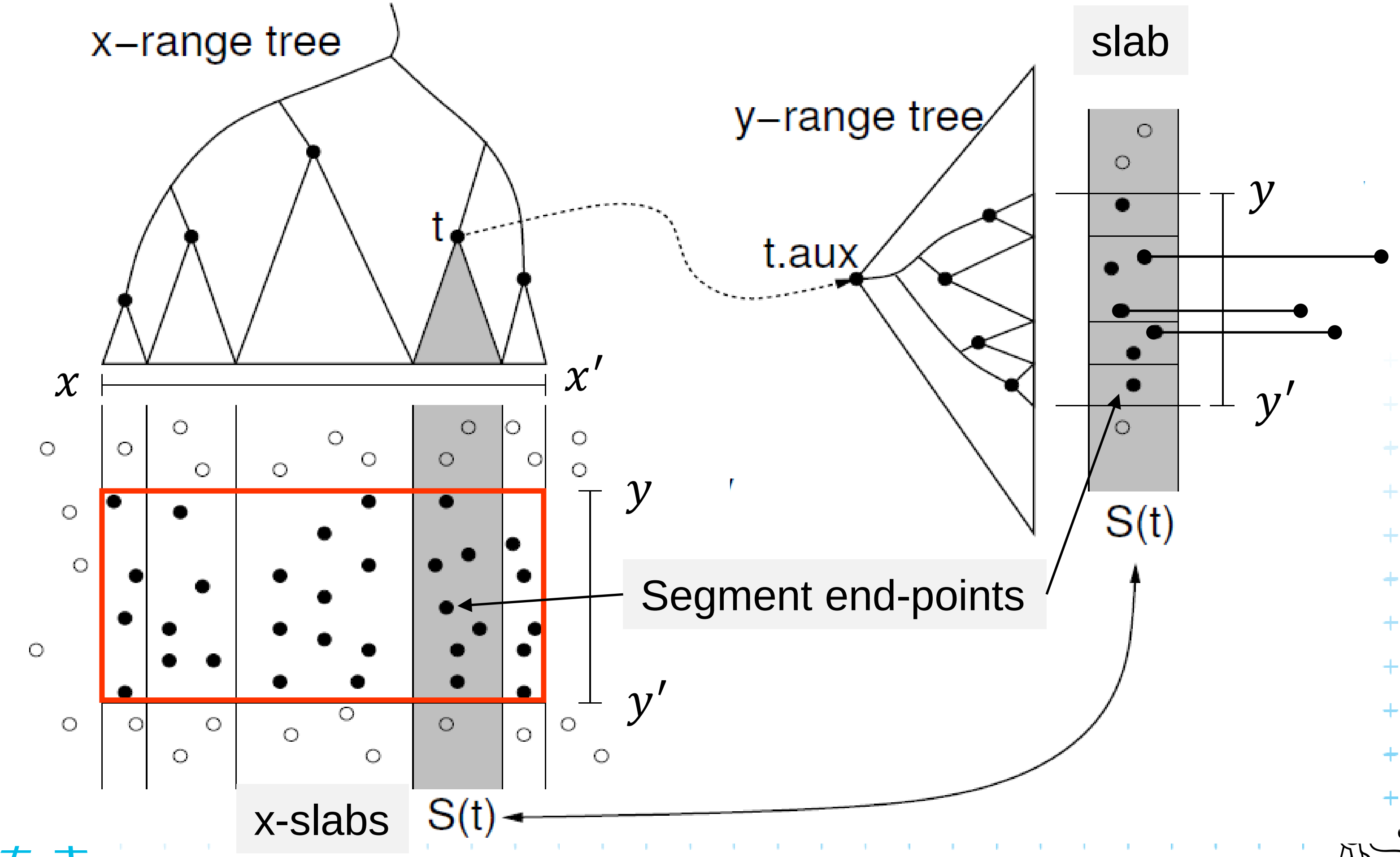
- Avoid reporting twice:
  - Mark segment when reported (clear after the query) and skip marked segments or
  - when end point found, check the other end-point and report only one of them (the leftmost or the bottom)

2D range tree (without fractional cascading-more in Lecture 3)

Search space: points

Query: Orthogonal intervals $[x : x'] \times [y : y']$

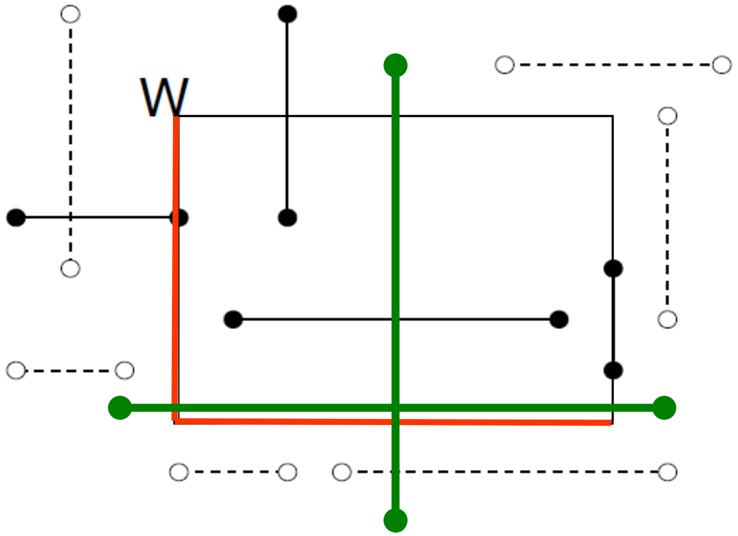
a), b)



Line segments that cross over the window

c) No points inside

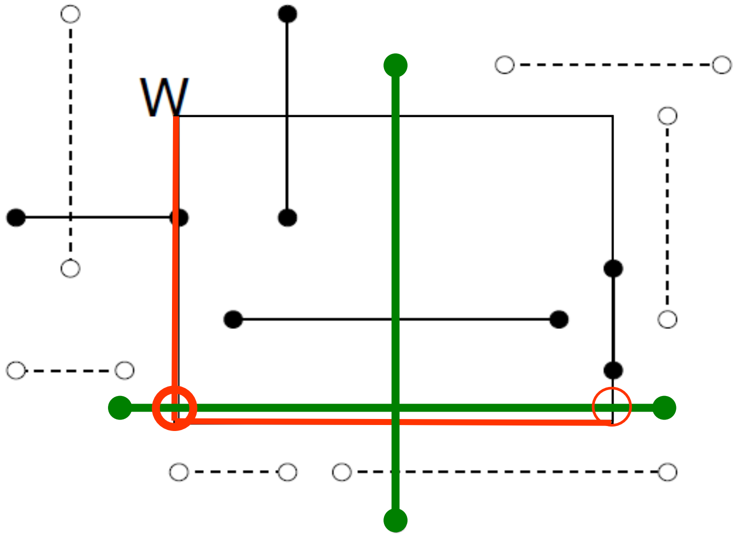
- Such segments not detected using end-point range tree
- Cross the boundary twice



Line segments that cross over the window

c) No points inside

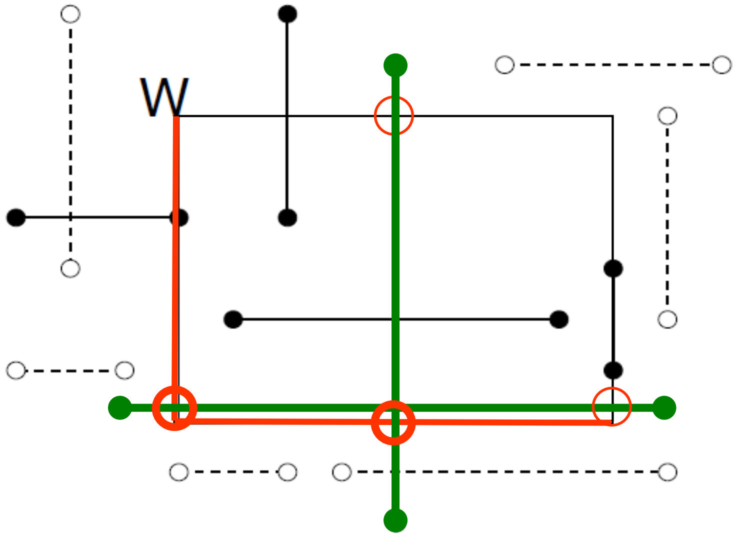
- Such segments not detected using end-point range tree
- Cross the boundary twice



Line segments that cross over the window

c) No points inside

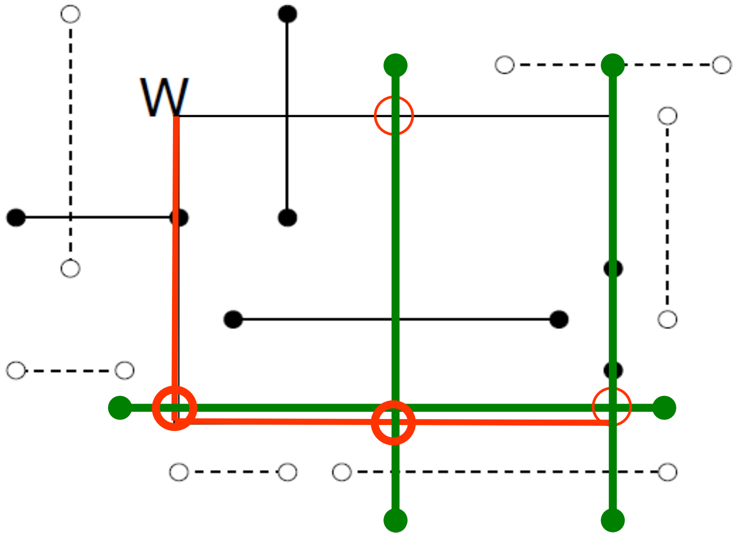
- Such segments not detected using end-point range tree
- Cross the boundary twice



Line segments that cross over the window

c) No points inside

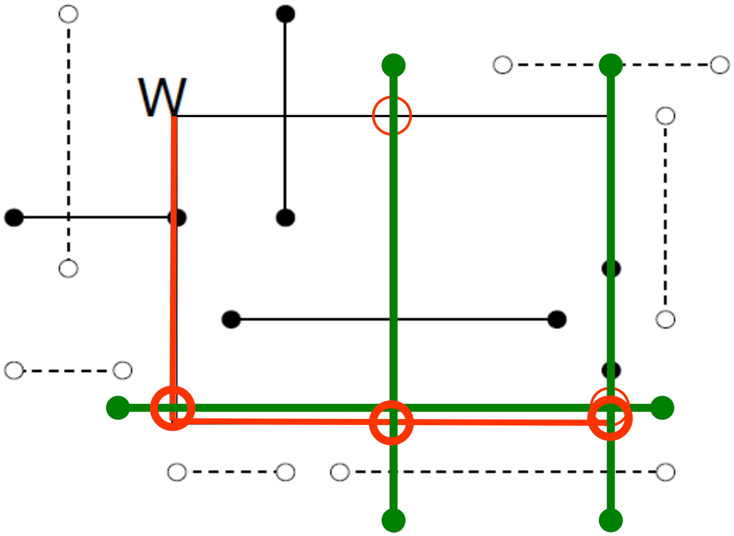
- Such segments not detected using end-point range tree
- Cross the boundary twice



Line segments that cross over the window

c) No points inside

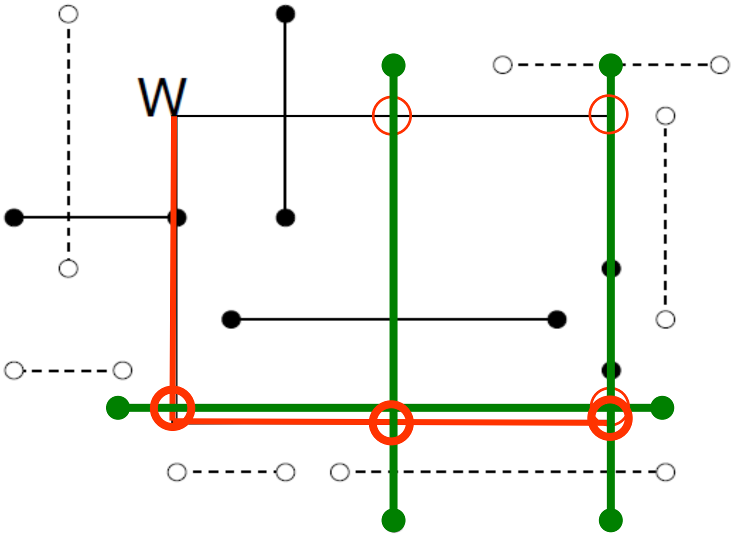
- Such segments not detected using end-point range tree
- Cross the boundary twice



Line segments that cross over the window

c) No points inside

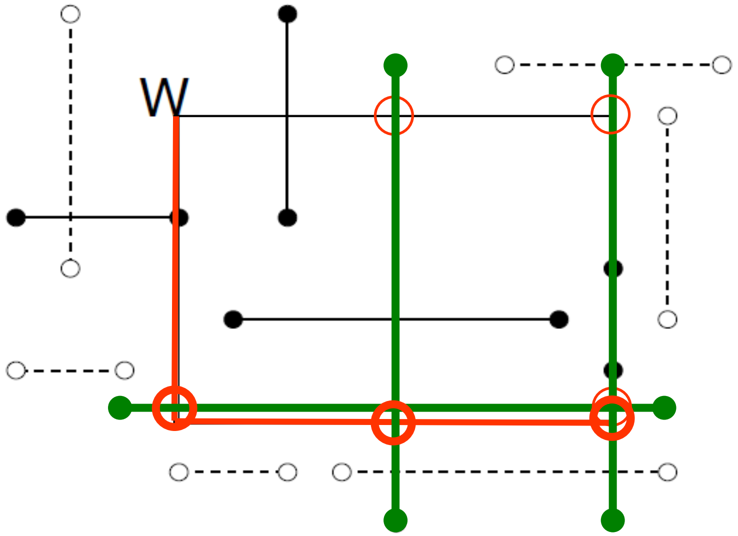
- Such segments not detected using end-point range tree
- Cross the boundary twice



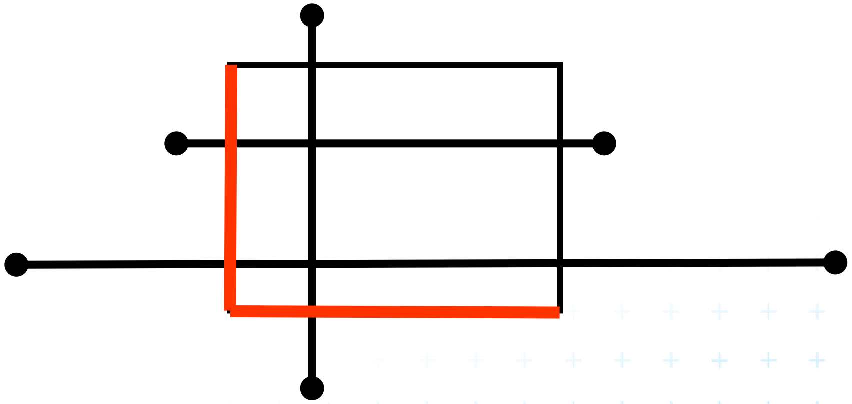
Line segments that cross over the window

c) No points inside

- Such segments not detected using end-point range tree
- Cross the boundary twice



For axis parallel segments

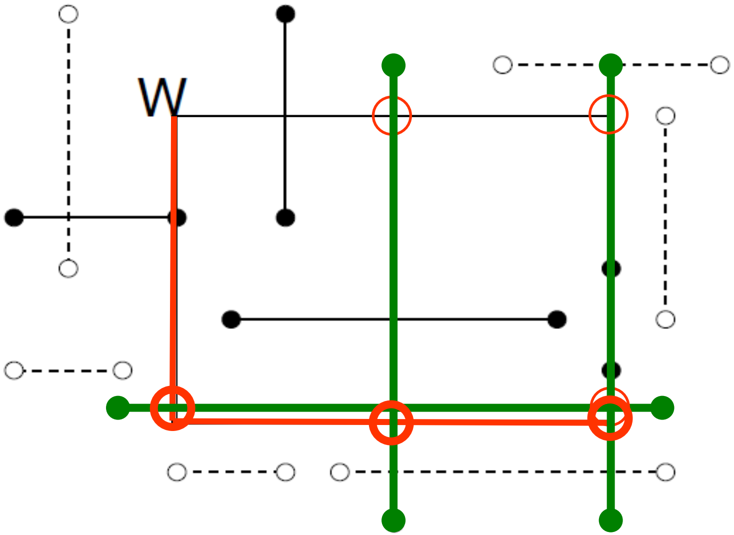


Check left and bottom boundary

Line segments that cross over the window

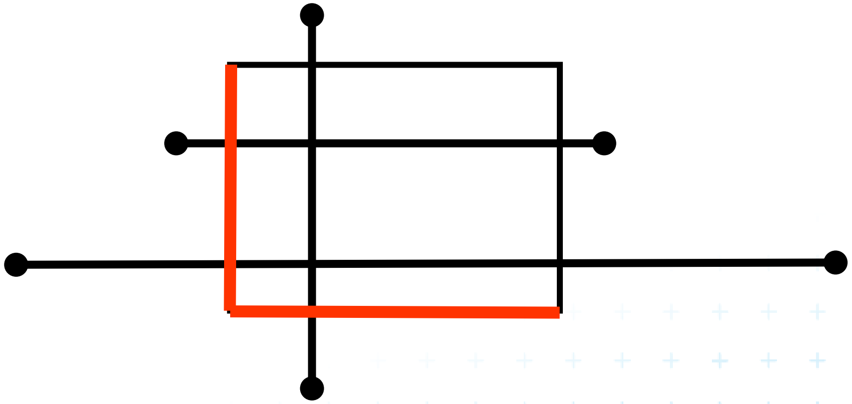
c) No points inside

- Such segments not detected using end-point range tree
- Cross the boundary twice

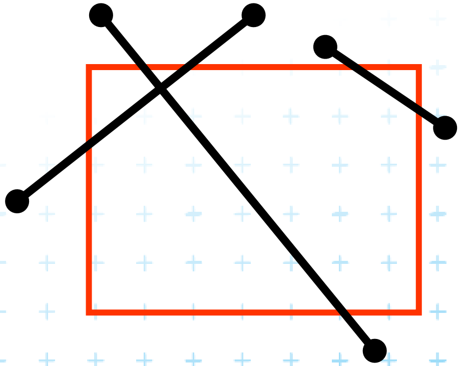


[Mount]

For axis parallel segments



Check left and bottom boundary

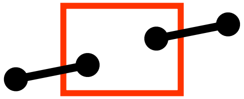


For non-parallel segments

Check all 4 boundaries

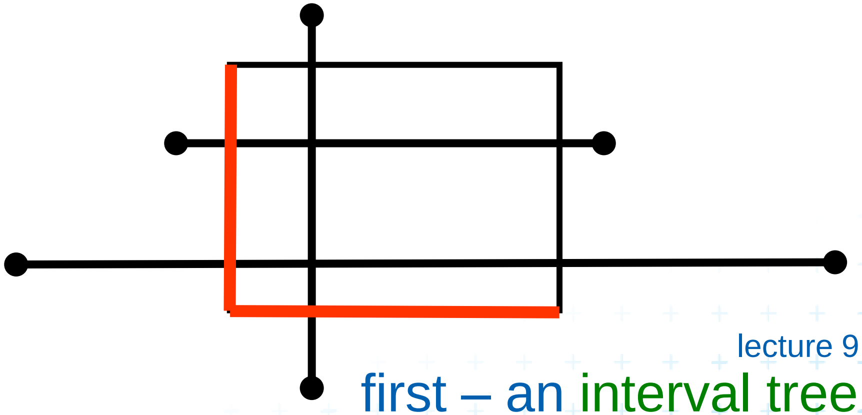
Windowing problem summary

Cases a) and b)



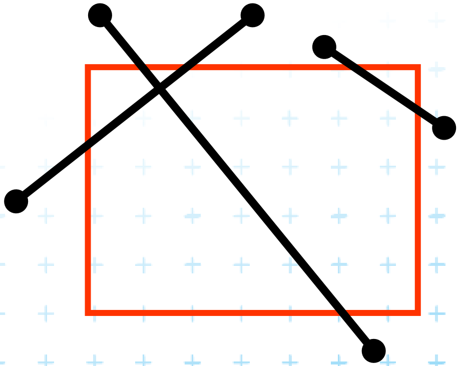
- Segment end-point in the query rectangle (window)
- Solved by 2D range trees (see lecture 3, $O(n \log n)$ time & memory)

- We will discuss only case c)
  - Segment crosses the window



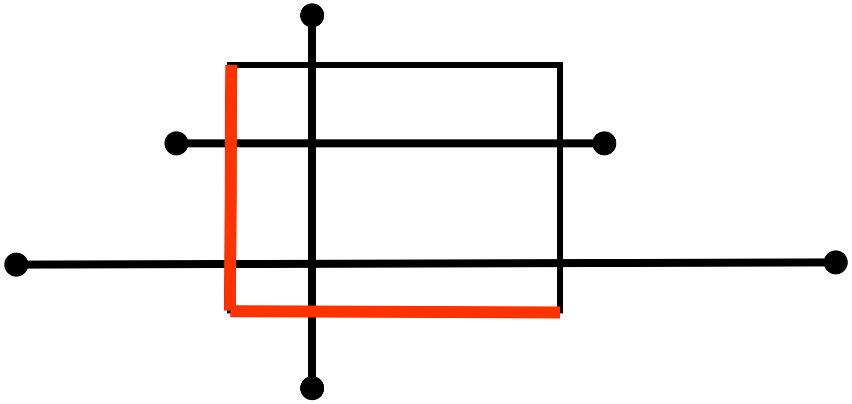
lecture 9

first – an interval tree (three variants)



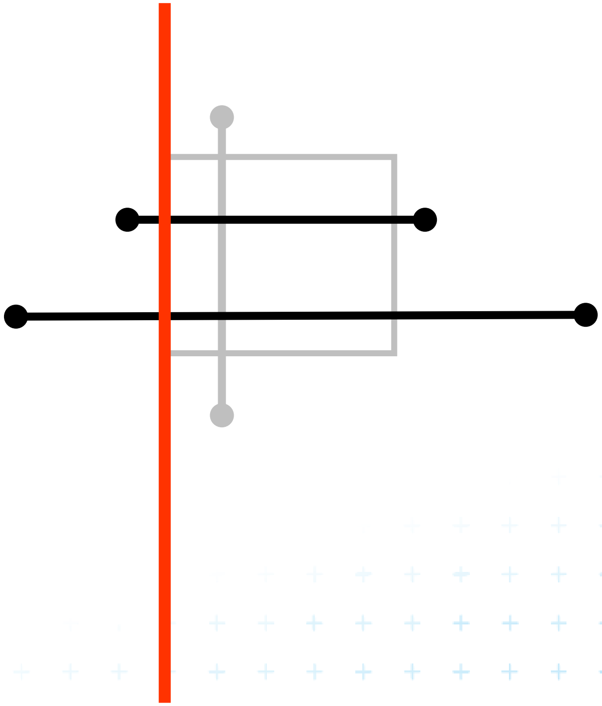
later – a segment tree

case c) principle



Segments cross the window

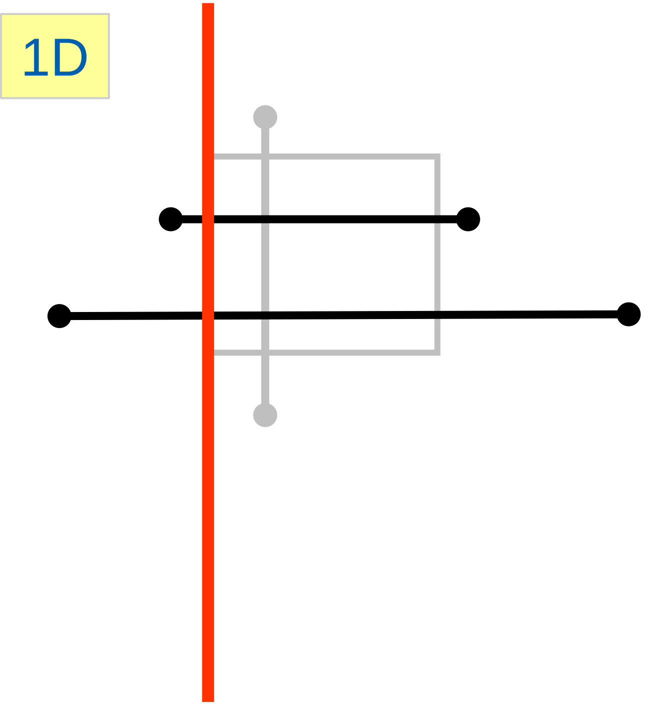
solved as

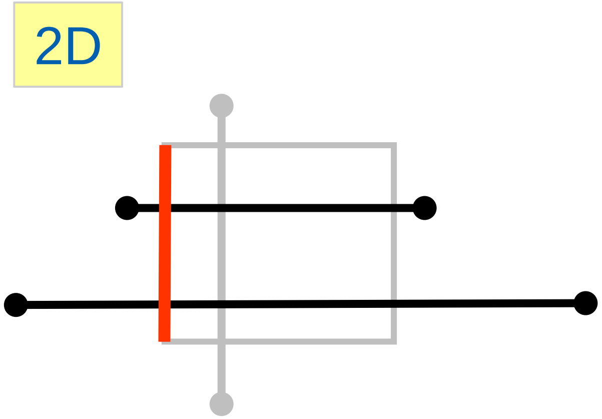


Line crosses the segments

(horizontal + vertical)

Talk Outline



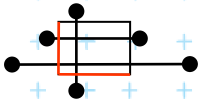


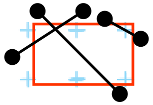
Line x line segments

interval tree

For heat-up

Line segment x line segments

2 variants of interval tree 

1 variant of segment tree 

Data structures for case c)

Interval tree (1D IT)

stores 1D intervals (end-points in sorted lists)

computes intersections with query interval

see intersection of axis angle rectangles – there is y-overlap used, here is x-overlap

We must extend Interval tree to 2D

variants differ in storage of interval end-points M_L, M_R

- 2D range trees
- priority search trees

Segment tree

splits the plane to slabs in x in elementary intervals

Talk overview

1. Windowing of axis parallel line segments in 2D

- 3 variants of *interval tree – IT in x-direction*
- Differ in storage of segment end points M_L and M_R

1D

i. Line stabbing (standard *IT* with *sorted lists*) lecture 9 - intersections

2D

ii. Line segment stabbing (*IT* with *range trees*)

iii. Line segment stabbing (*IT* with *priority search trees*)

2. Windowing of line segments in general position

2D

- *segment tree + BST*

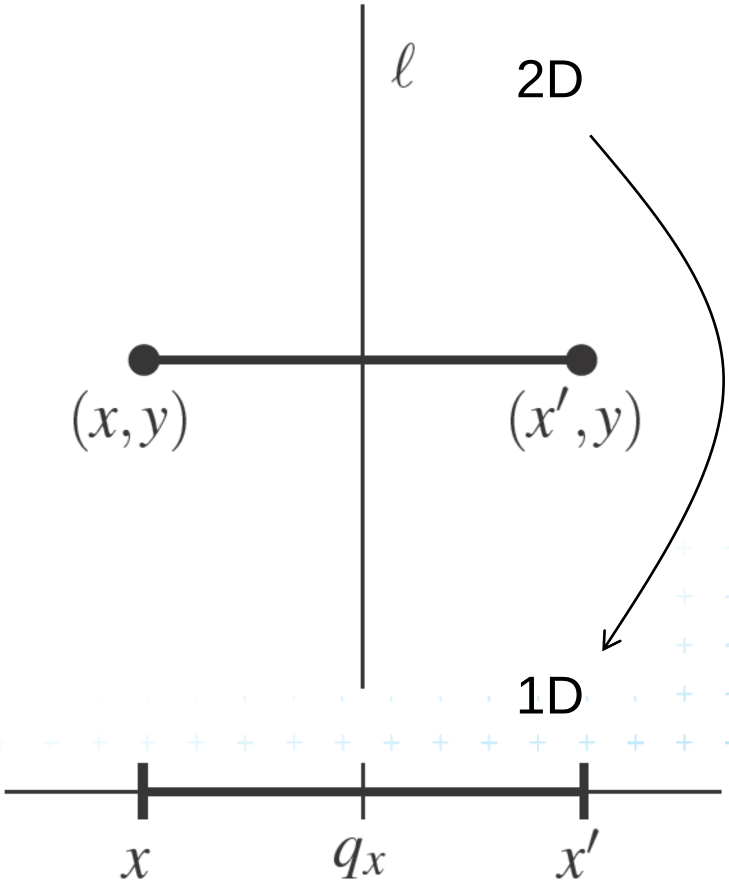
i. Segment intersected by vertical line

- Query line $\ell := \ (x = q_x)$

  Report the segments stabbed by a vertical line

  = 1 dimensional problem

  (ignore y coordinate)



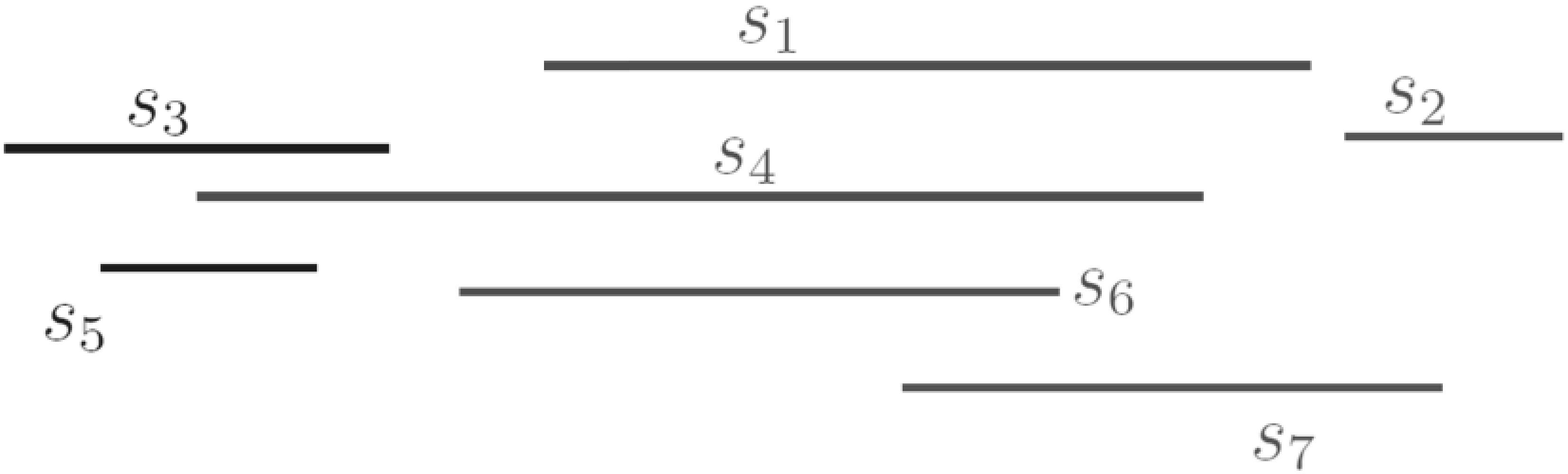
⇒ Report the interval $[x : x']$ containing query point q_x

DS: Interval tree with sorted lists

[Mount]

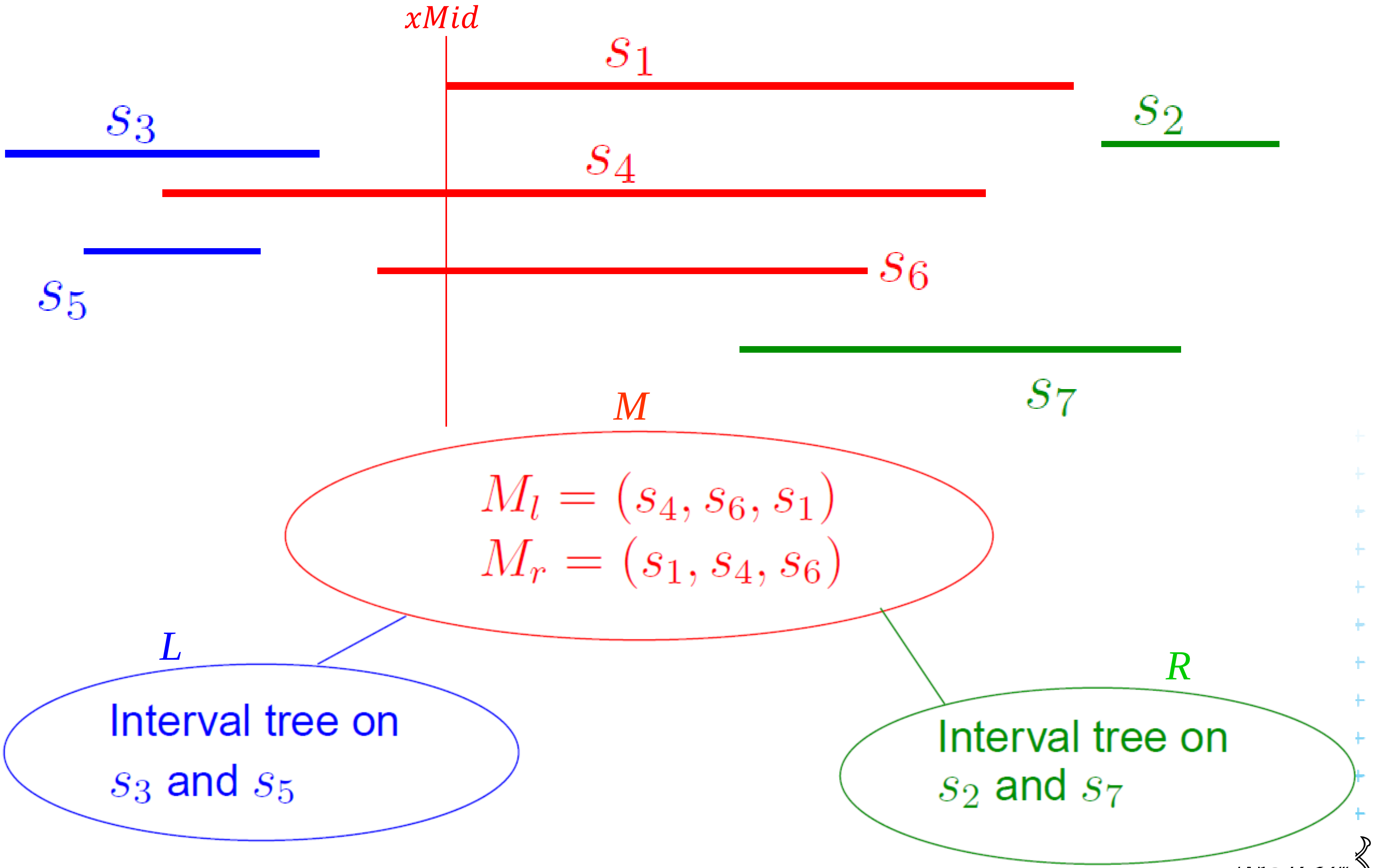
Interval tree principle

(see lecture 9 - intersections)



Interval tree principle

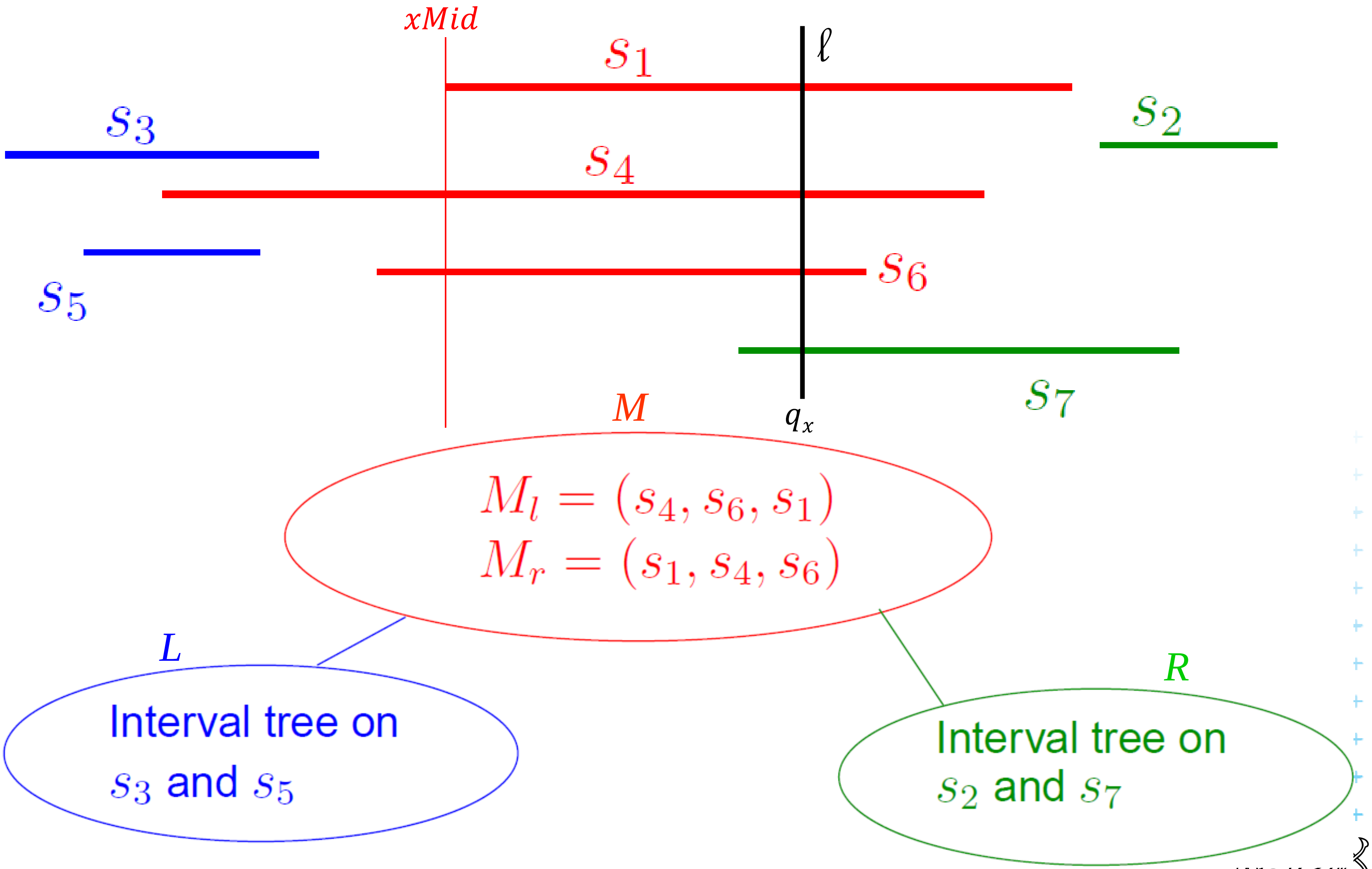
(see lecture 9 - intersections)



[Vigneron]

Interval tree principle

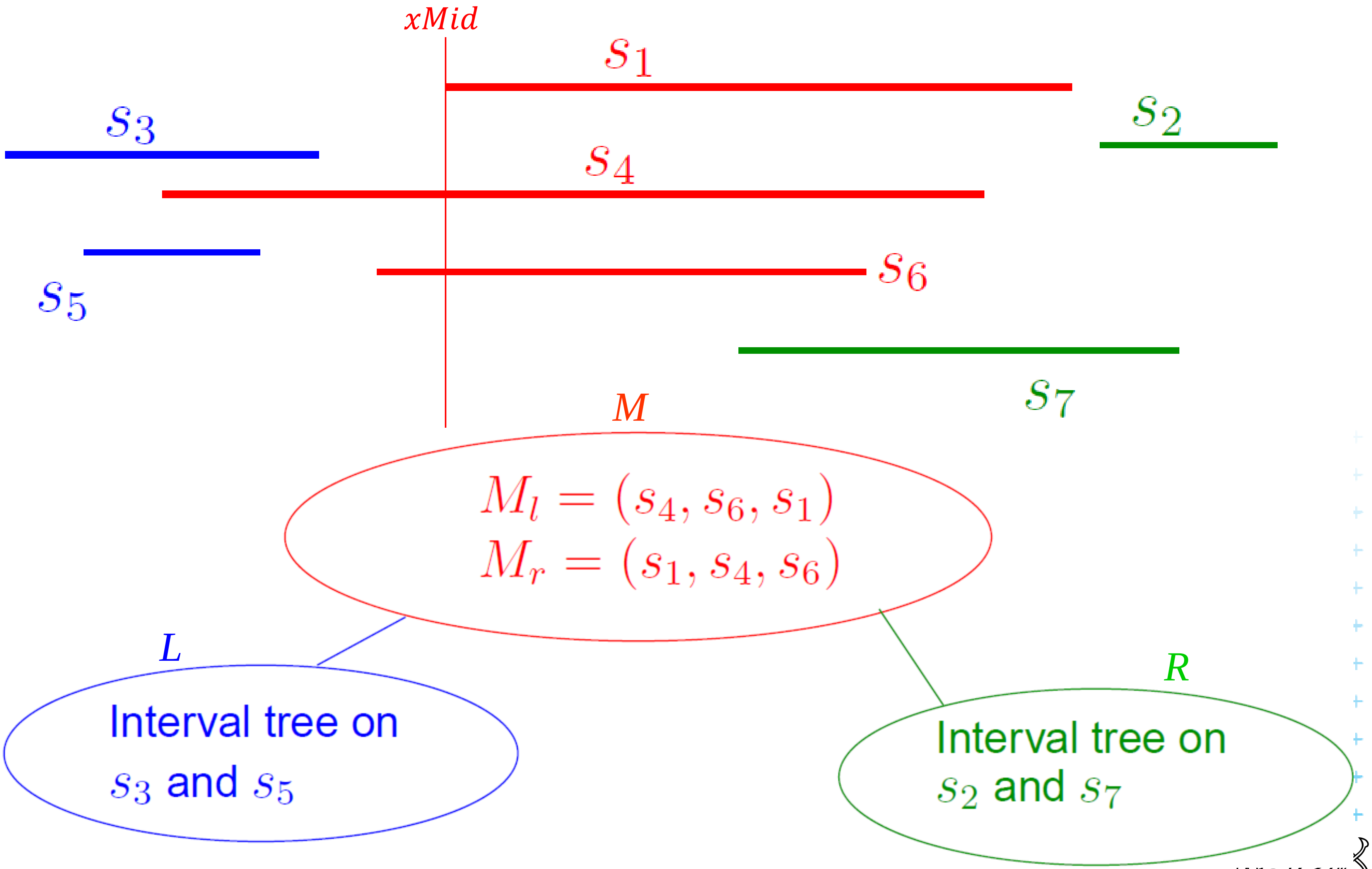
(see lecture 9 - intersections)



[Vigneron]

Interval tree principle

(see lecture 9 - intersections)



$M_l = (s_4, s_6, s_1)$

$M_r = (s_1, s_4, s_6)$

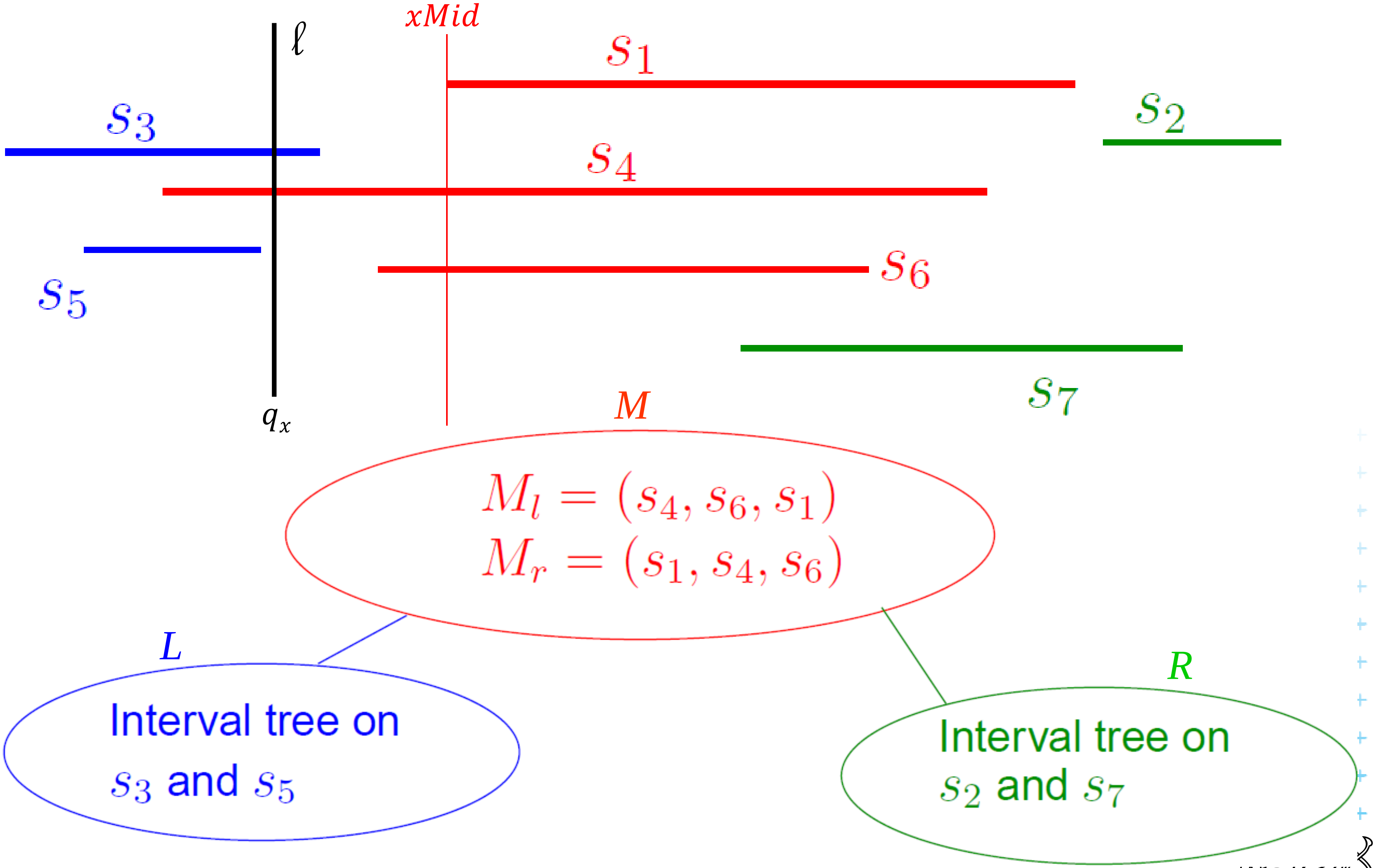
Interval tree on s_3 and s_5

Interval tree on s_2 and s_7

[Vigneron]

Interval tree principle

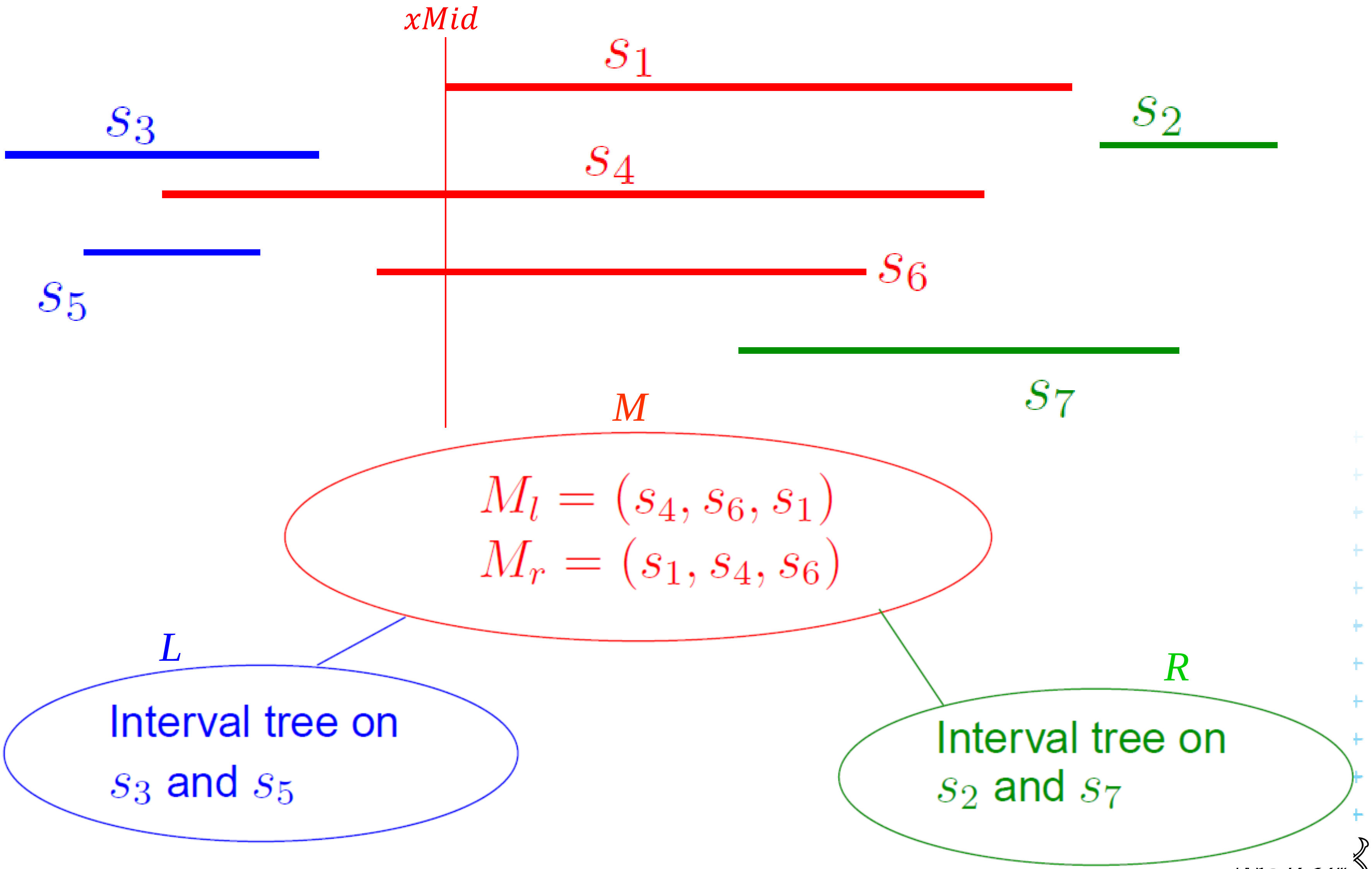
(see lecture 9 - intersections)



[Vigneron]

Interval tree principle

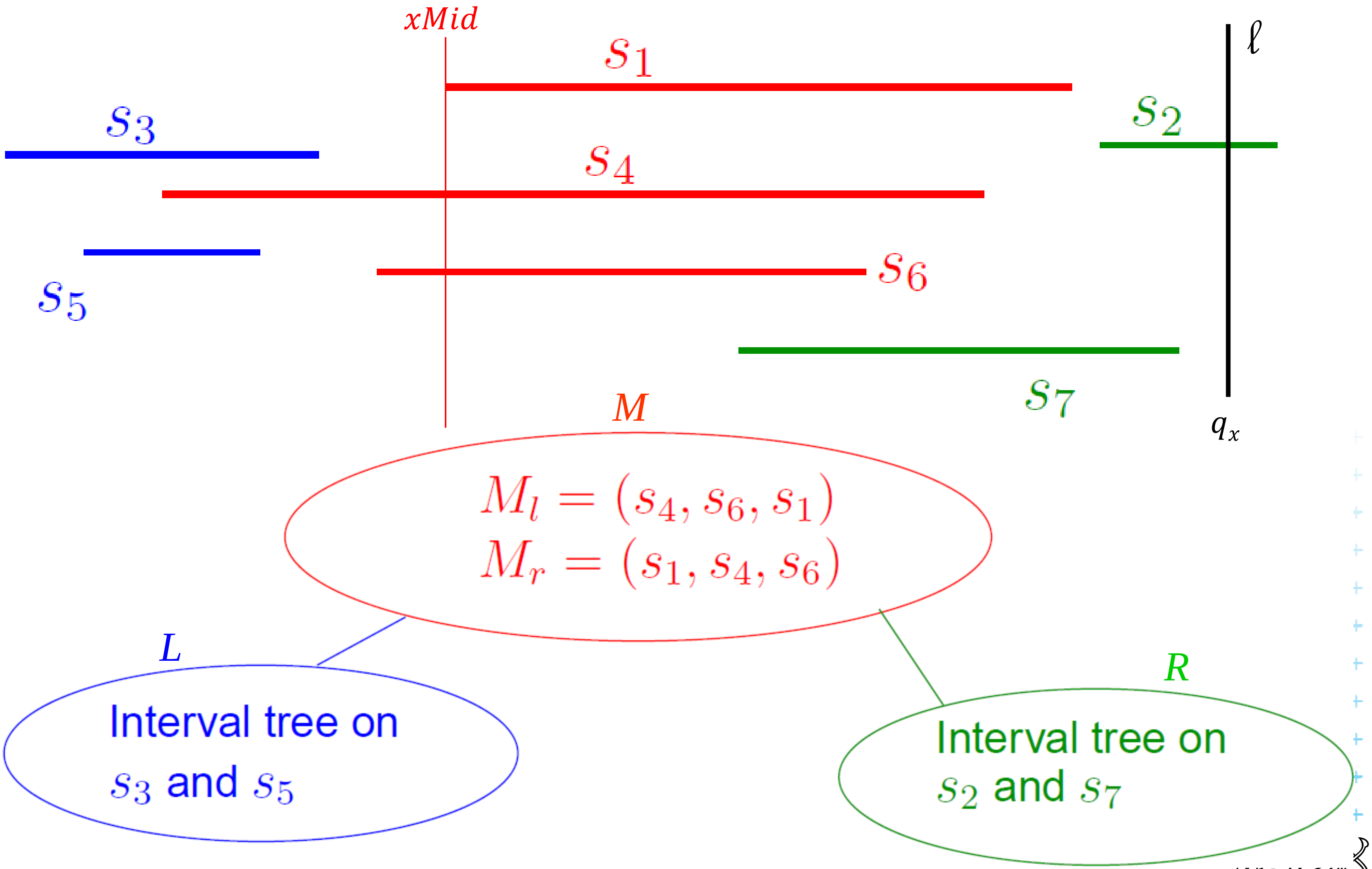
(see lecture 9 - intersections)



[Vigneron]

Interval tree principle

(see lecture 9 - intersections)



[Vigneron]

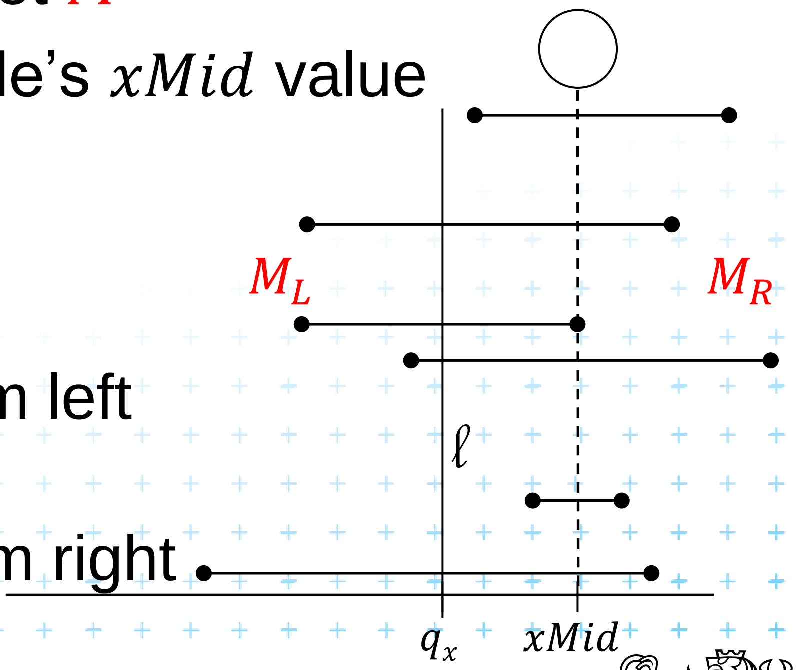
i. Segment intersected by vertical line

Principle

- Store input segments in static interval tree
- In each interval tree node
  - Check the segments in the set M
  - These segments contain node's $xMid$ value
    - M_L are left end-points
    - M_R are right end-points
  - q_x is the query value
  - If $(q_x < xMid)$ Sweep M_L from left

    p ∈ M_L: if $p_x \leq q_x \Rightarrow$ intersection
  - If $(q_x > xMid)$ Sweep M_R from right

    p ∈ M_R: if $p_x \geq q_x \Rightarrow$ intersection



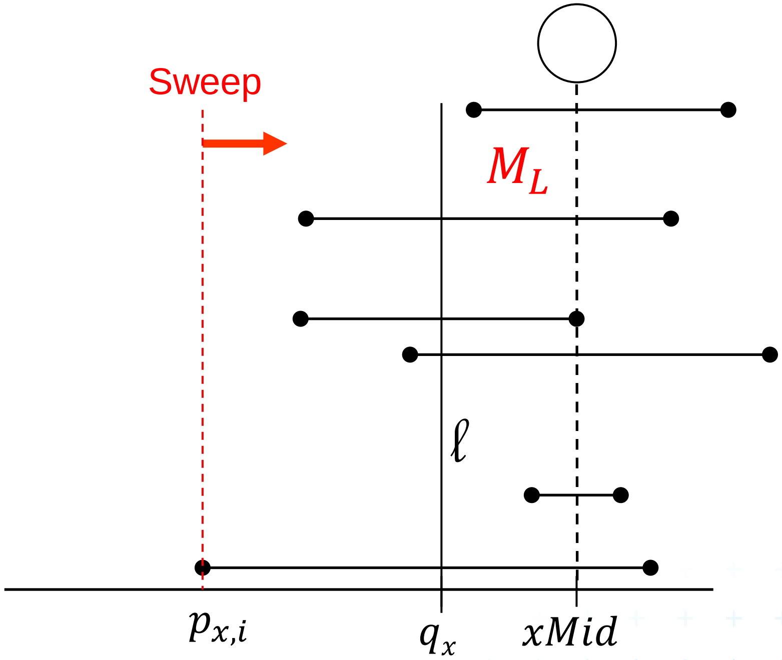
Inspired by [Berg]

Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

$\Rightarrow q_x$ must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

$\Rightarrow$ left endpoints $p_{x,i} \leq q_x \Rightarrow$ intersection



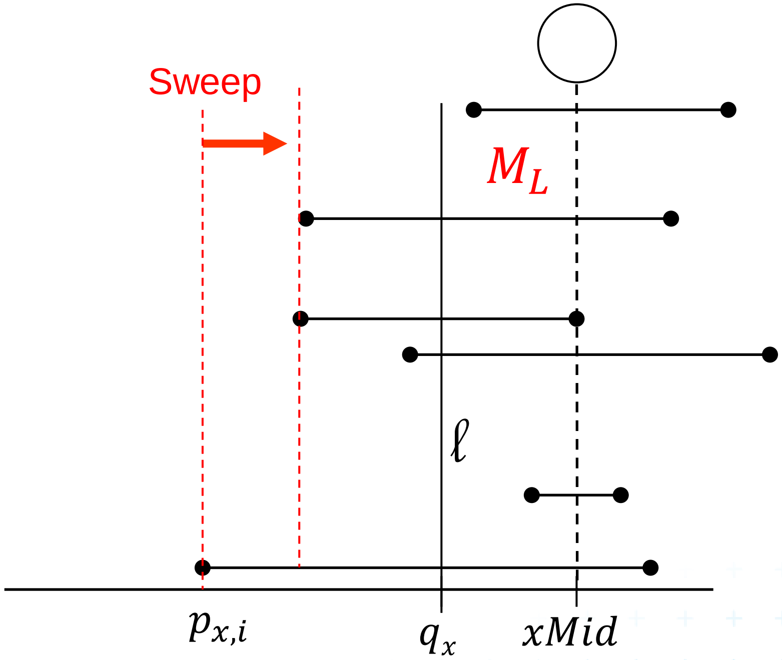
Inspired by [Berg]

Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

⇒ q_x must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

⇒ left endpoints $p_{x,i} \leq q_x$ ⇒ intersection



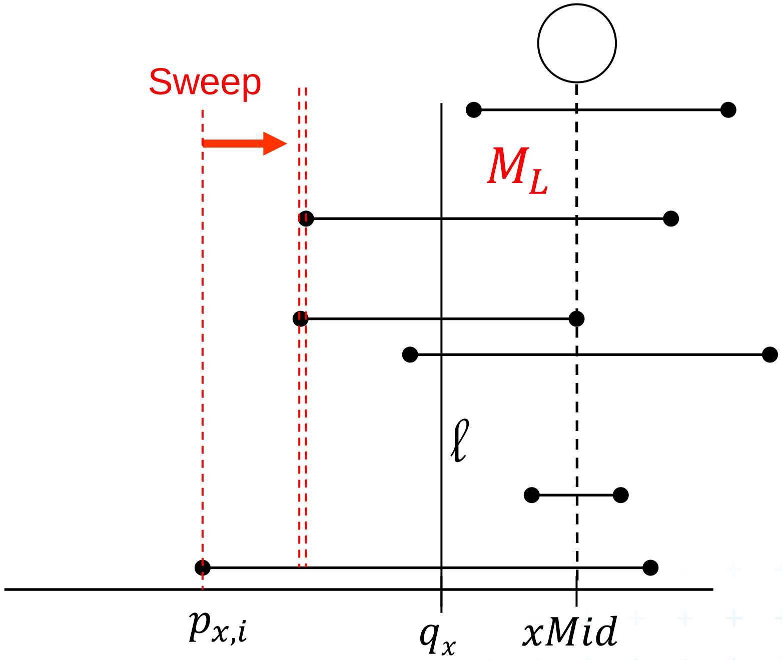
Inspired by [Berg]

Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

⇒ q_x must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

⇒ left endpoints $p_{x,i} \leq q_x$ ⇒ intersection



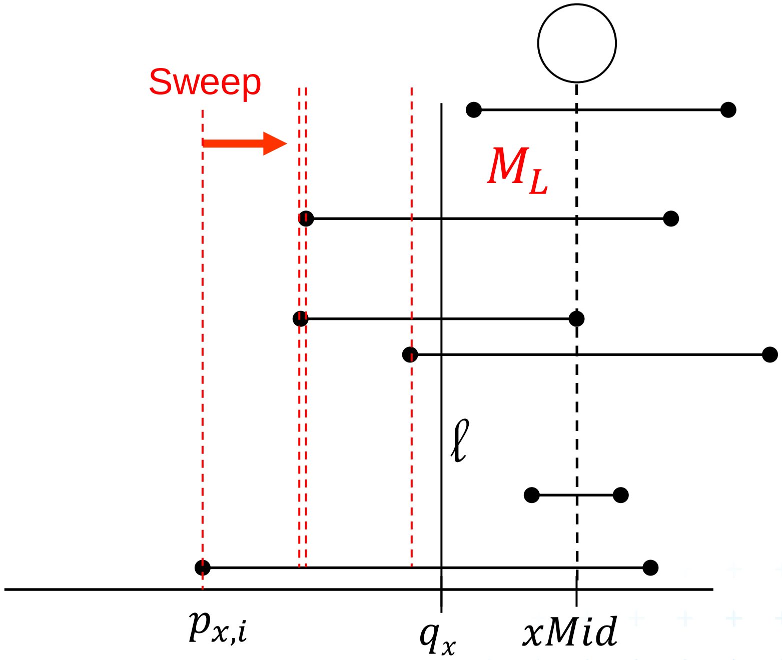
Inspired by [Berg]

Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

⇒ q_x must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

⇒ left endpoints $p_{x,i} \leq q_x$ ⇒ intersection



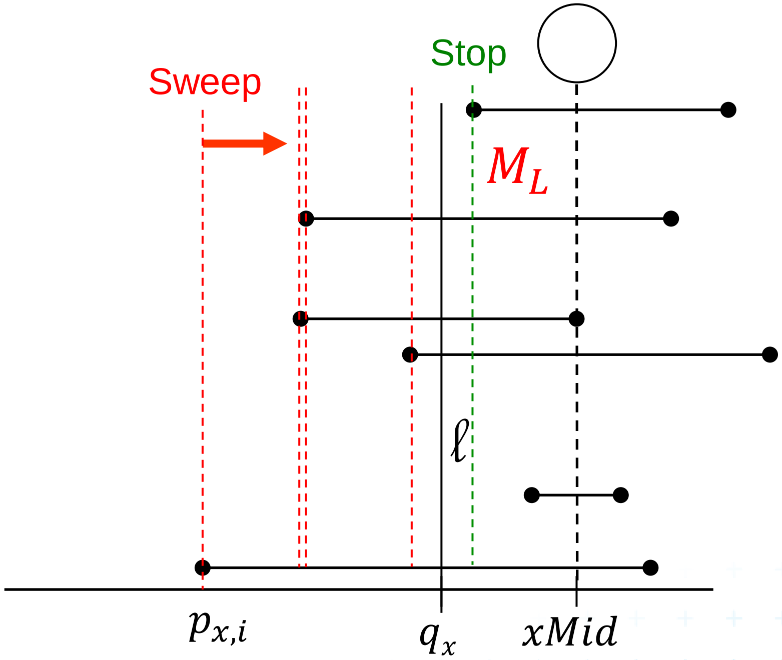
Inspired by [Berg]

Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

⇒ q_x must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

⇒ left endpoints $p_{x,i} \leq q_x$ ⇒ intersection



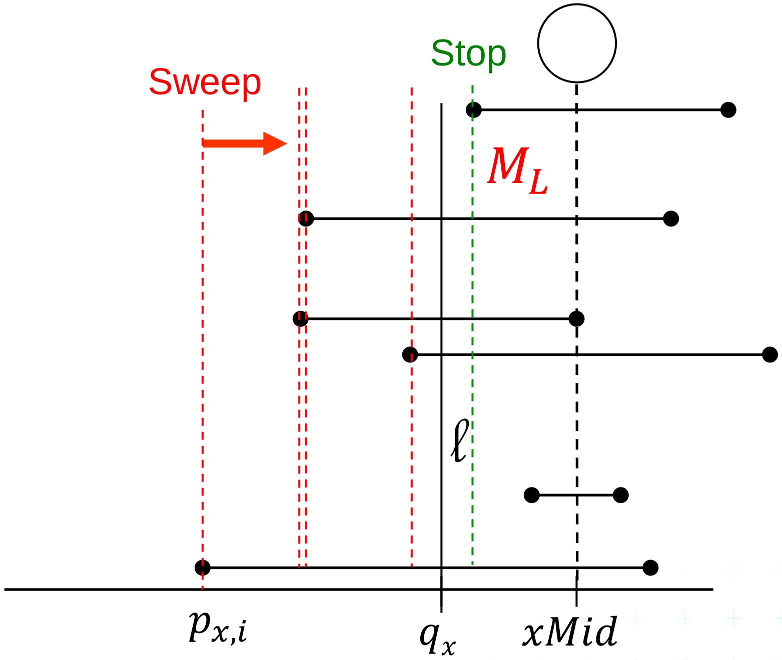
Inspired by [Berg]

Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

⇒ q_x must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

⇒ left endpoints $p_{x,i} \leq q_x$ ⇒ intersection



Intersection with line ℓ

$\ell := q_x \times [-\infty : \infty]$

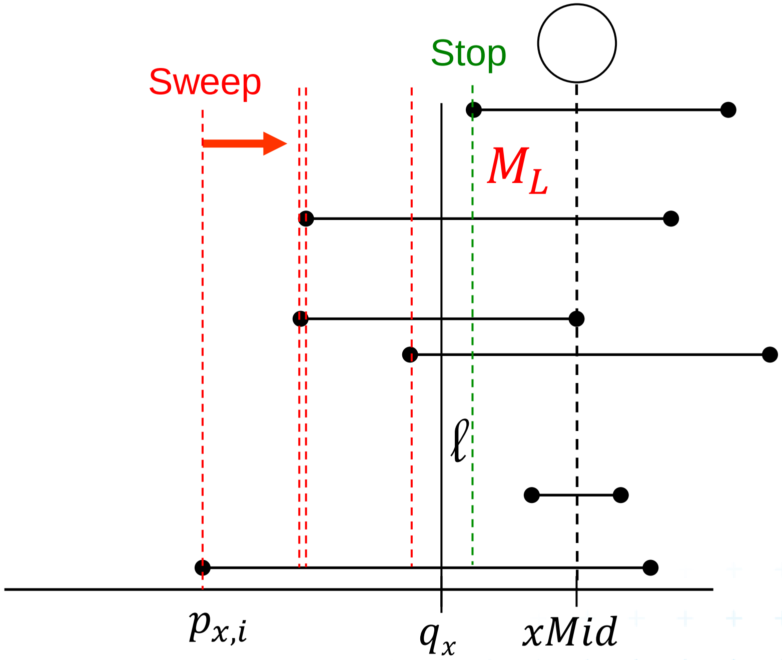
Inspired by [Berg]

Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

⇒ q_x must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

⇒ left endpoints $p_{x,i} \leq q_x$ ⇒ intersection



Intersection with line ℓ means

$$\ell := q_x \times [-\infty : \infty]$$

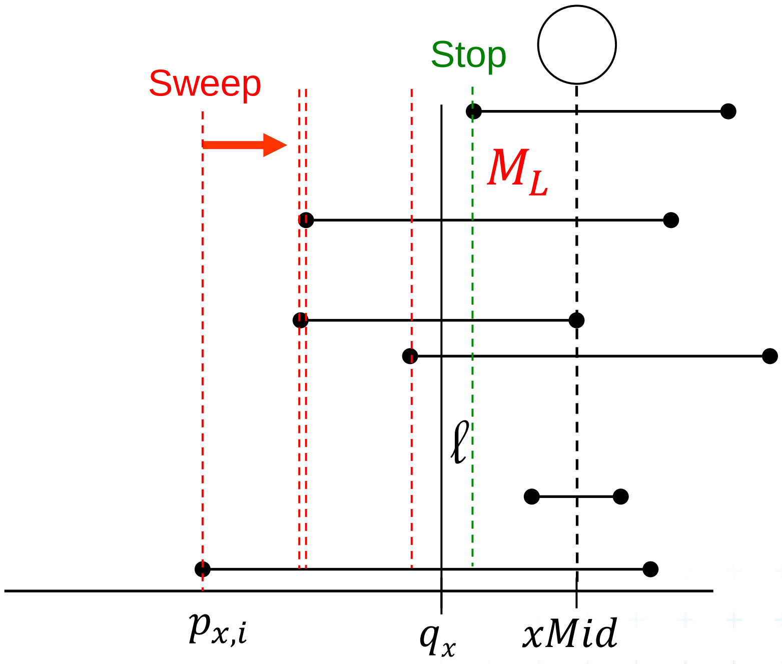
Inspired by [Berg]

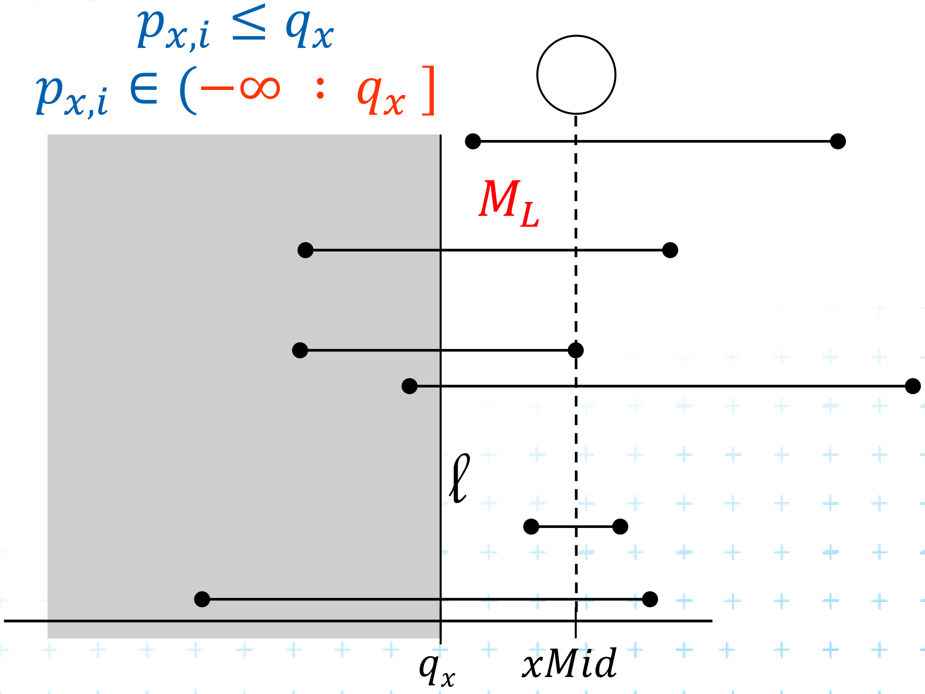
Segment intersection (left from $xMid$)

All line segments from M pass through $xMid$

⇒ q_x must be between $p_{x,i}$ and $xMid$ to intersect the line segment i

⇒ left endpoints $p_{x,i} \le q_x$ ⇒ intersection





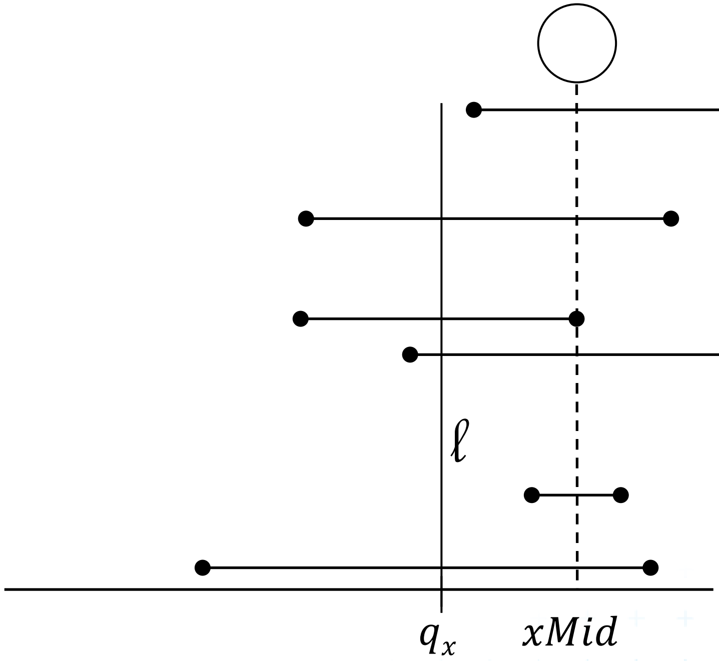
Intersection with line ℓ means Intersection with half space q

$\ell := q_x \times [-\infty : \infty]$ $q := (-\infty \;:\; q_x\,] \times [-\infty : \infty]$

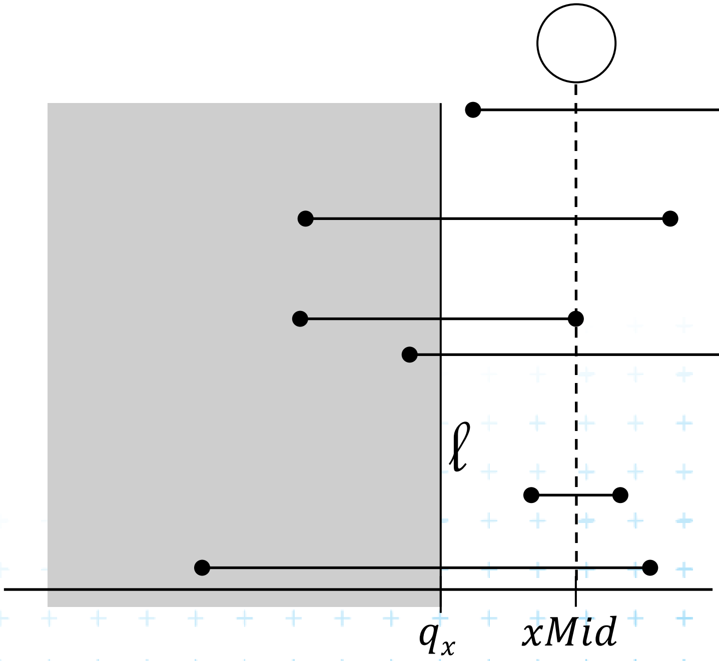
Inspired by [Berg]

Principle once more

Instead of
intersecting edges by line



search points in half-space

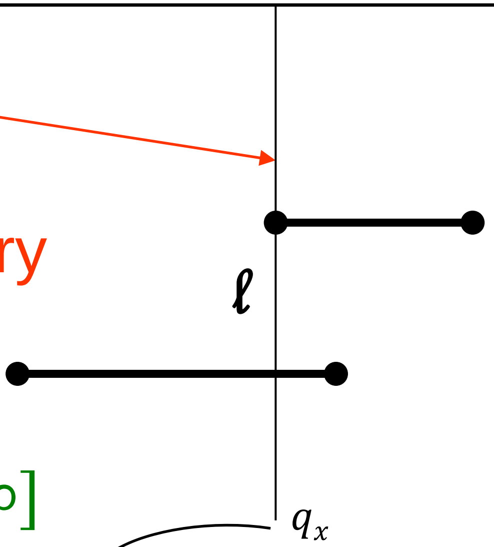


i. Segment intersected by vertical line

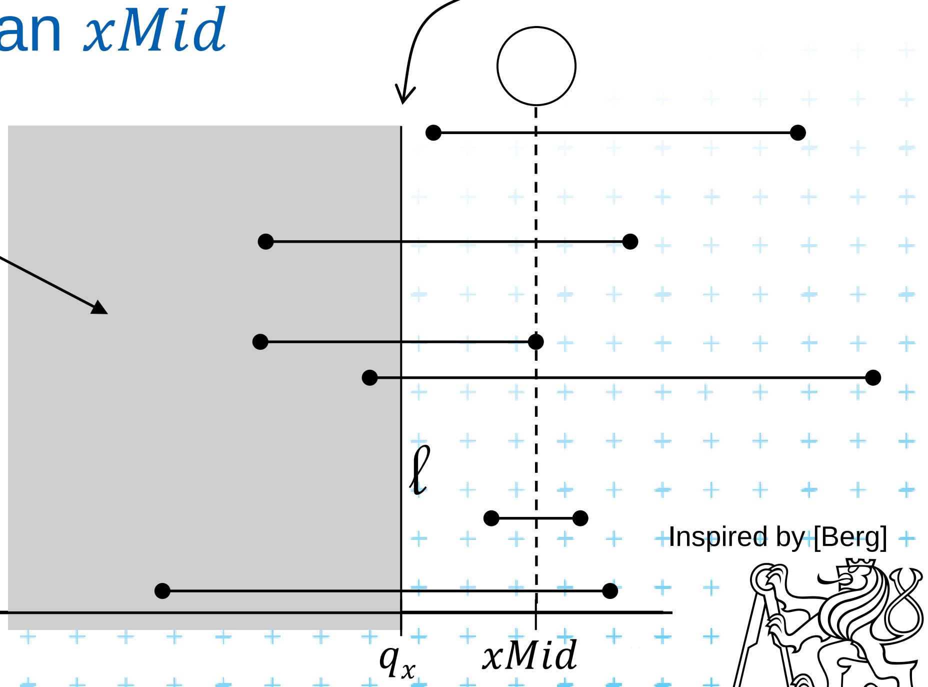
De facto a 1D problem

- Query line $\ell := q_x \cancel{\times [-\infty : \infty]}$
- Horizontal segment of M stabs the query line ℓ left of $xMid$ **iff** its (segment's) left endpoint lies in half-space

$$q := (-\infty : q_x] \times [-\infty : \infty]$$



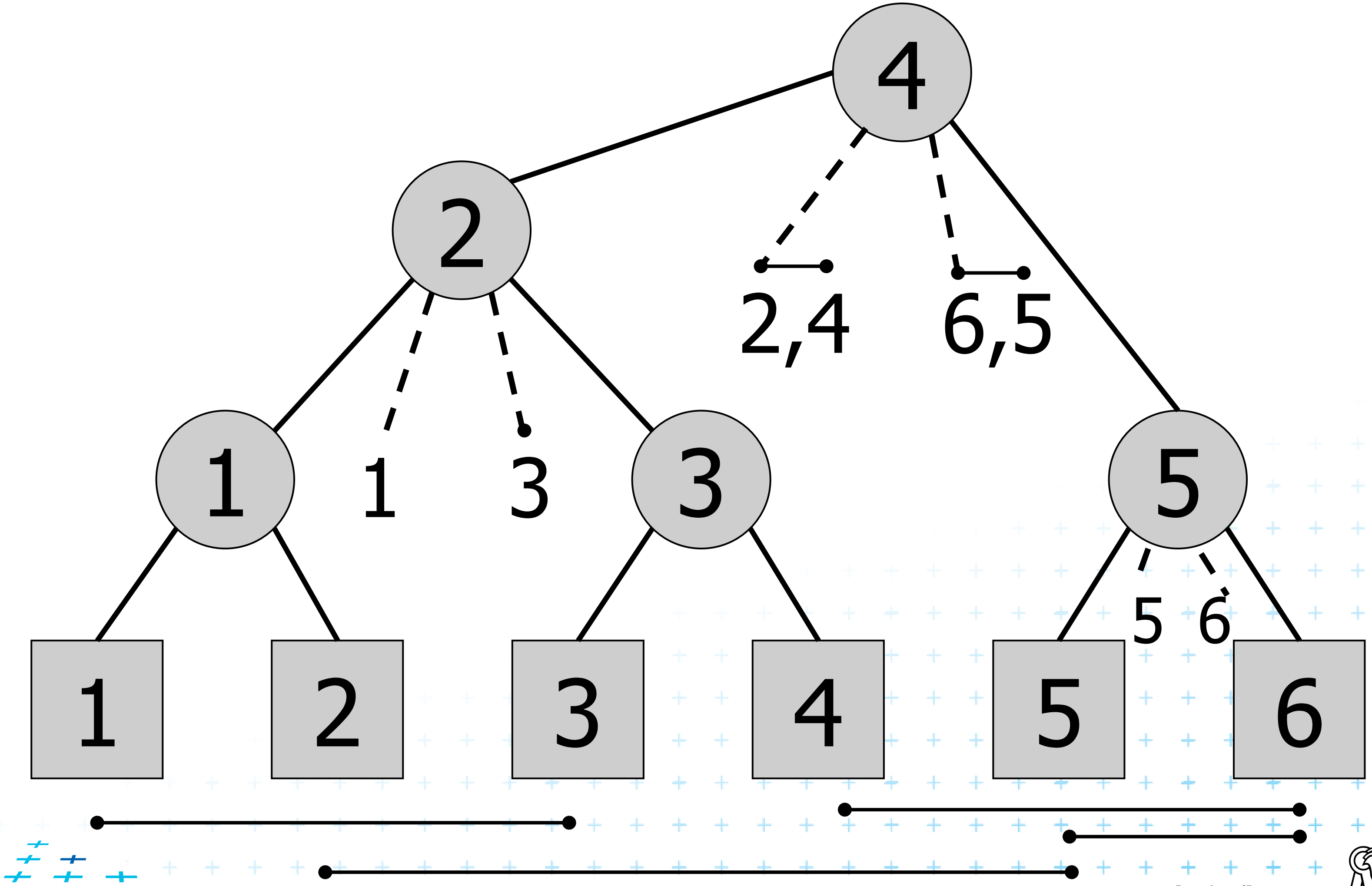
- In IT node with stored median $xMid$ report all segments from M
  - M_L: whose left point lies in $(-\infty \ : q_x]$ if ℓ lies left from xMid
  - M_R: whose right point lies in $[q_x : +\infty)$ if ℓ lies right from xMid



Inspired by [Berg]

Static interval tree [Edelsbrunner80]

Tree over sorted segment end-points



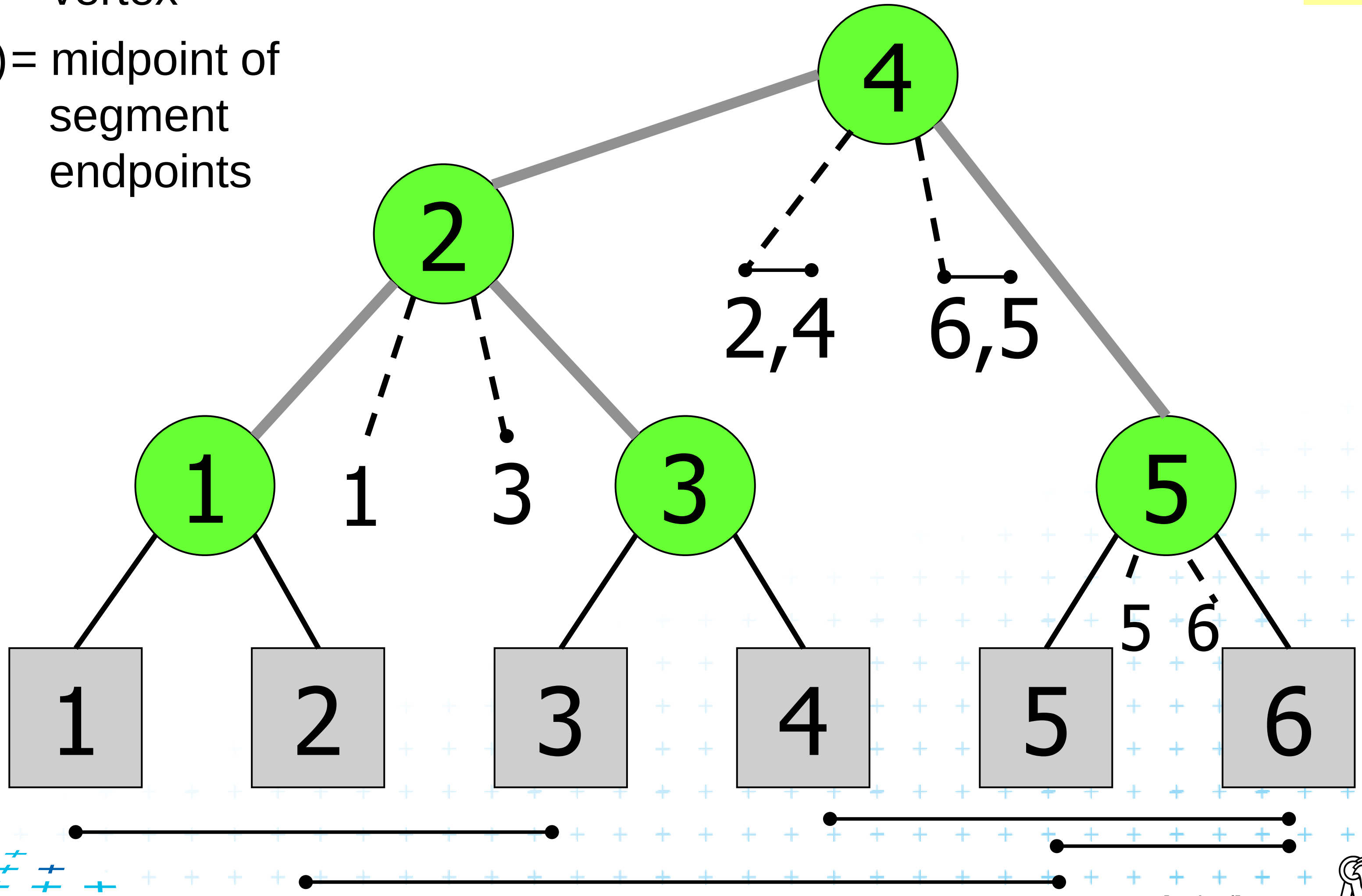
[Kukral]

Primary structure – static tree for endpoints

Static

v = vertex

d(v)= midpoint of segment endpoints



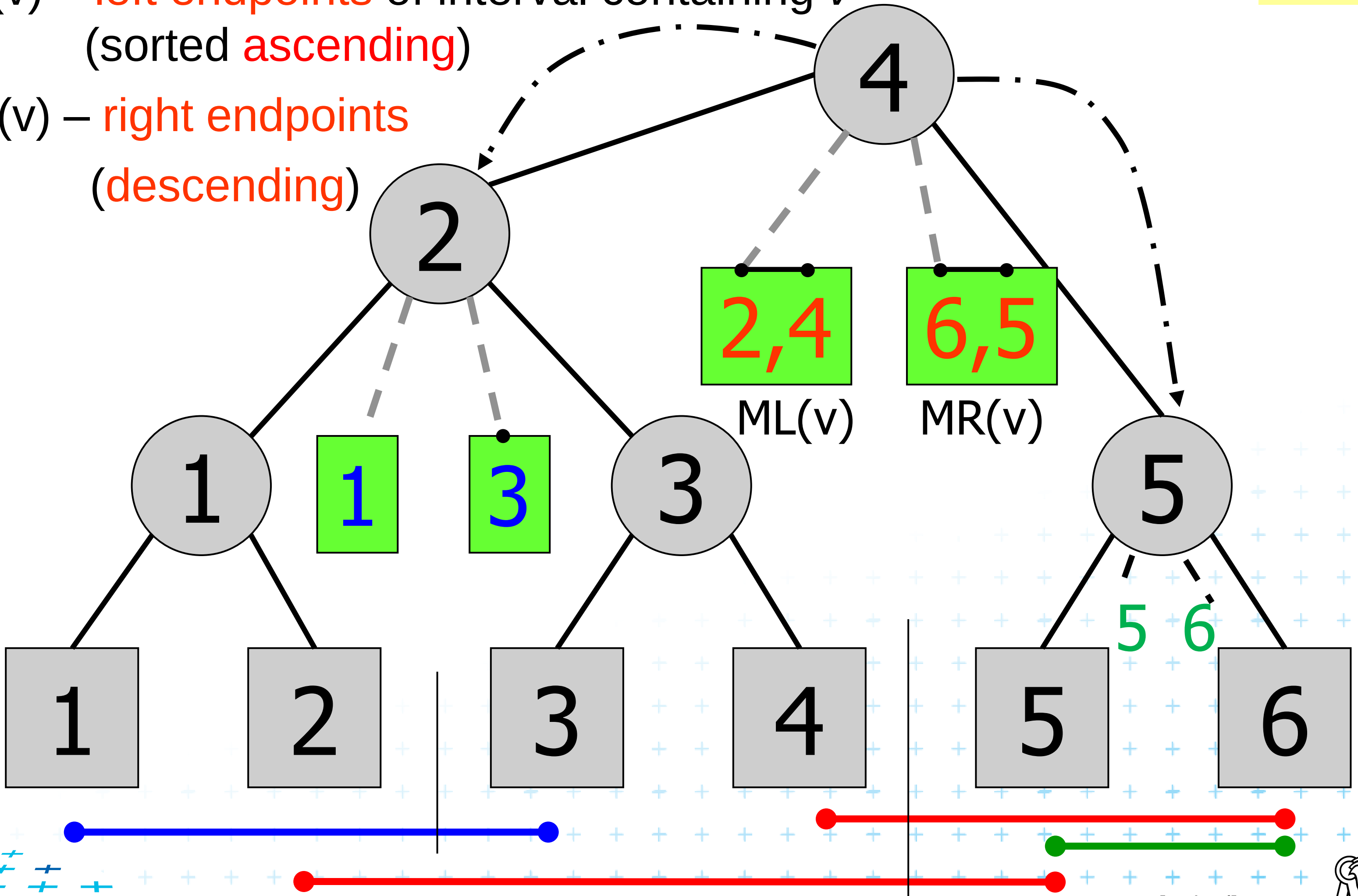
[Kukral]

Secondary lists of incident interval end-pts.

Dynamic

ML(v) – left endpoints of interval containing *v*
(sorted ascending)

MR(v) – right endpoints
(descending)



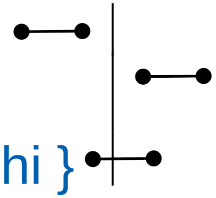
[Kukral]

Interval tree construction

Merged procedures from in lecture 09

- PrimaryTree(*S*) on slide 33
- InsertInterval (*b, e, T*) on slide 35

```
ConstructIntervalTree( S )        // Intervals all active – no active lists
Input:    Set S of intervals on the real line – on x-axis
Output:   The root of an interval tree for S
1.  if (|S| == 0) return null                              // no more intervals
2.  else
3.      xMed = median endpoint of intervals in S           // median endpoint
4.      L = { [xlo, xhi] in S | xhi < xMed }               // left of median
5.      R = { [xlo, xhi] in S | xlo > xMed }               // right of median
6.      M = { [xlo, xhi] in S | xlo <= xMed <= xhi }       // contains median
7.      ML = sort M in increasing order of xlo             // sort M
8.      MR = sort M in decreasing order of xhi
9.      t = new IntTreeNode(xMed, ML, MR)                  // this node
10.     t.left  = ConstructIntervalTree(L)                 // left subtree
11.     t.right = ConstructIntervalTree(R)                 // right subtree
12.     return t
```



steps 4.,5.,6. done in one step if presorted [Mount]

Line stabbing query for an interval tree

Less effective variant of QueryInterval (b, e, T)
on slide 34 in lecture 09
with merged parts: fork and search right

Stab(t, qx)

Input: IntTreeNode t, Scalar qx

Output: prints the intersected intervals

```
if (t == null) return                              // no leaf: fell out of the tree
if (qx < t.xMed)                                   // left of median?
    for (i = 0; i < t.ML.length; i++)              // traverse $M_L$ left end-points
            if (t.ML[i].lo ≤ qx) print (t.ML[i])   // ..report if in range
            else break                             // ..else done
    Stab (t.left, qx)                              // recurse on left subtre
else  // (qx ≥ t.xMed)                             // right of or equal to median
    for (i = 0; i < t.MR.length; i++) {            // traverse $M_R$ right end-points
            if (t.MR[i].hi ≥ qx) print (t.MR[i])   // ..report if in range
            else break                             // ..else done
    Stab (t.right, qx)                             // recurse on right subtree
```

Note: Small inefficiency for qx == t.xMed – recurse on right

[Mount]

Complexity of line stabbing via interval tree

with *sorted lists*

- Construction - $O(n \log n)$ time
  - Each step divides at maximum into two halves or less (minus elements of M) => tree of height $h = O(\log n)$
  - If presorted endpoints in three lists L,R, and M then median in O(1) and copy to new L,R,M in $O(n)$
- Vertical line stabbing query - $O(k + \log n)$ time
  - One node processed in $O(1 + k')$, k'reported intervals
  - v visited nodes in $O(v + k)$, k total reported intervals
  - $v = h =$ tree height = $O(\log n)$ $k = \Sigma k'$
- Storage - $O(n)$
  - Tree has $O(n)$ nodes, each segment stored twice (two endpoints)

Talk overview

1. Windowing of axis parallel line segments in 2D

– 3 variants of *interval tree – IT in x-direction*

– Differ in storage of segment end points M_L and M_R

1D i. Line stabbing (standard *IT* with *sorted lists*) lecture 9 - intersections

2D ii. Line segment stabbing (*IT* with *range trees*)

iii. Line segment stabbing (*IT* with *priority search trees*)

2. Windowing of line segments in general position

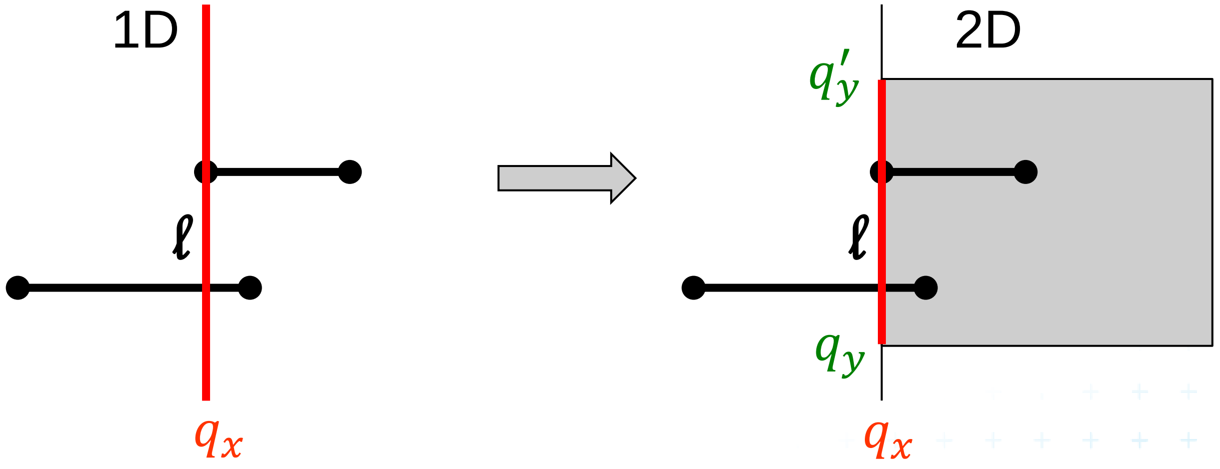
2D – *segment tree + BST*

Line segment stabbing (*IT* with *range trees*)

Enhance 1D interval trees to 2D

change lines

to segments



$q_x \times [-\infty : \infty]$ (no y-test)

$q_x \times [q_y : q'_y]$ (additional y-test)

Sorted lists

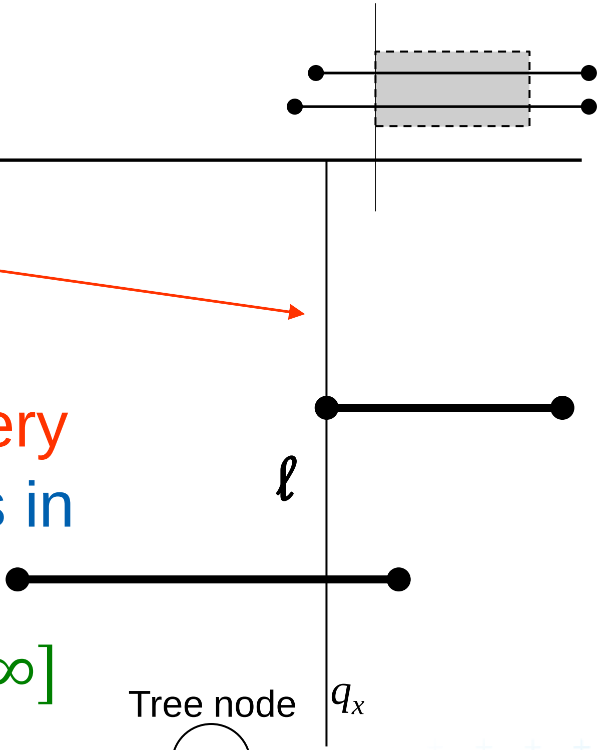
Range trees

i. Segments × vertical line

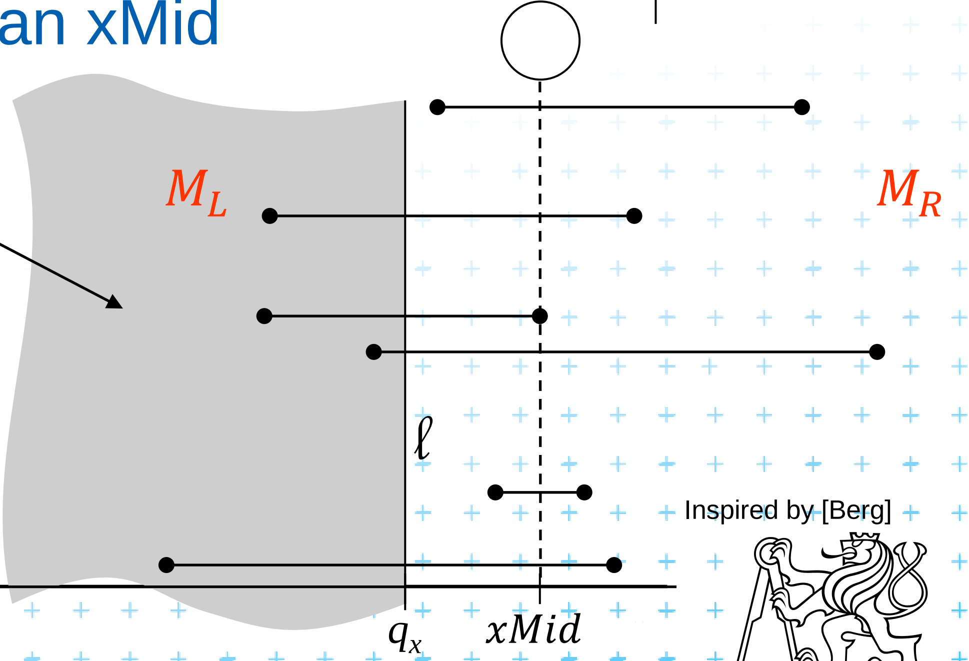
De facto a 1D problem

- Query line $\ell := q_x \times [-\infty : \infty]$
- Horizontal segment of M_L stabs the query line ℓ left of $xMid$ **iff** its left endpoint lies in half-space

$$q := (-\infty : q_x] \times [-\infty : \infty]$$



- In IT node with stored median xMid report all segments from M
  - M_L: whose left point lies in $(-\infty : q_x]$ if ℓ lies left from xMid
  - M_R: whose right point lies in $[q_x : +\infty)$ if ℓ lies right from xMid

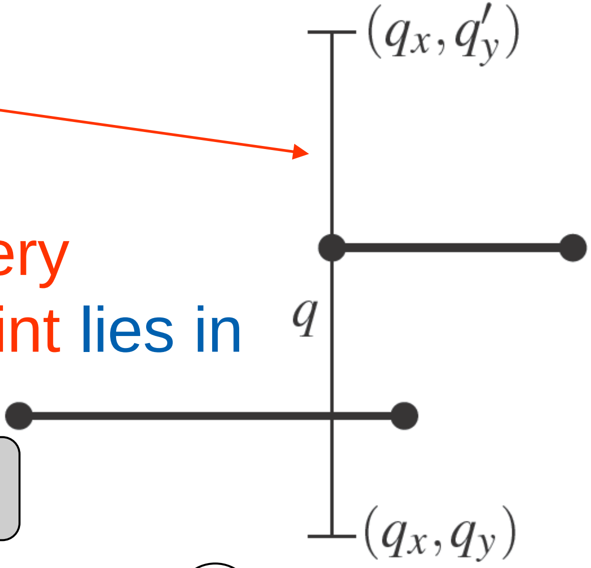


Inspired by [Berg]

ii. Segments × vertical line segment

A 2D problem

- Query segment $q \coloneqq q_x \times [q_y : q'_y]$

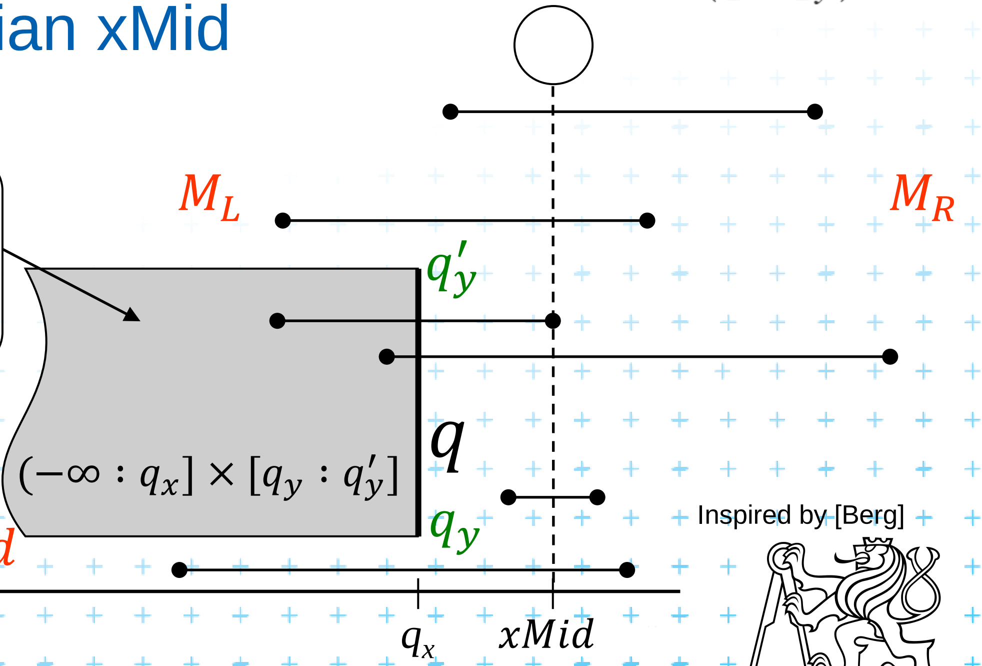


- Horizontal segment of M_L stabs the query segment q left of $xMid$ **iff** its left endpoint lies in semi-infinite rectangular region

New test

$$q \coloneqq (-\infty : q_x] \times [q_y : q'_y]$$

- In IT node with stored median xMid report all segments
  - M_L: whose left points lie in $(-\infty : q_x] \times [q_y : q'_y]$ where q_x lies left from $xMid$
  - M_R: whose right point lies in $[q_x : +\infty) \times [q_y : q'_y]$ where q_x lies right from $xMid$

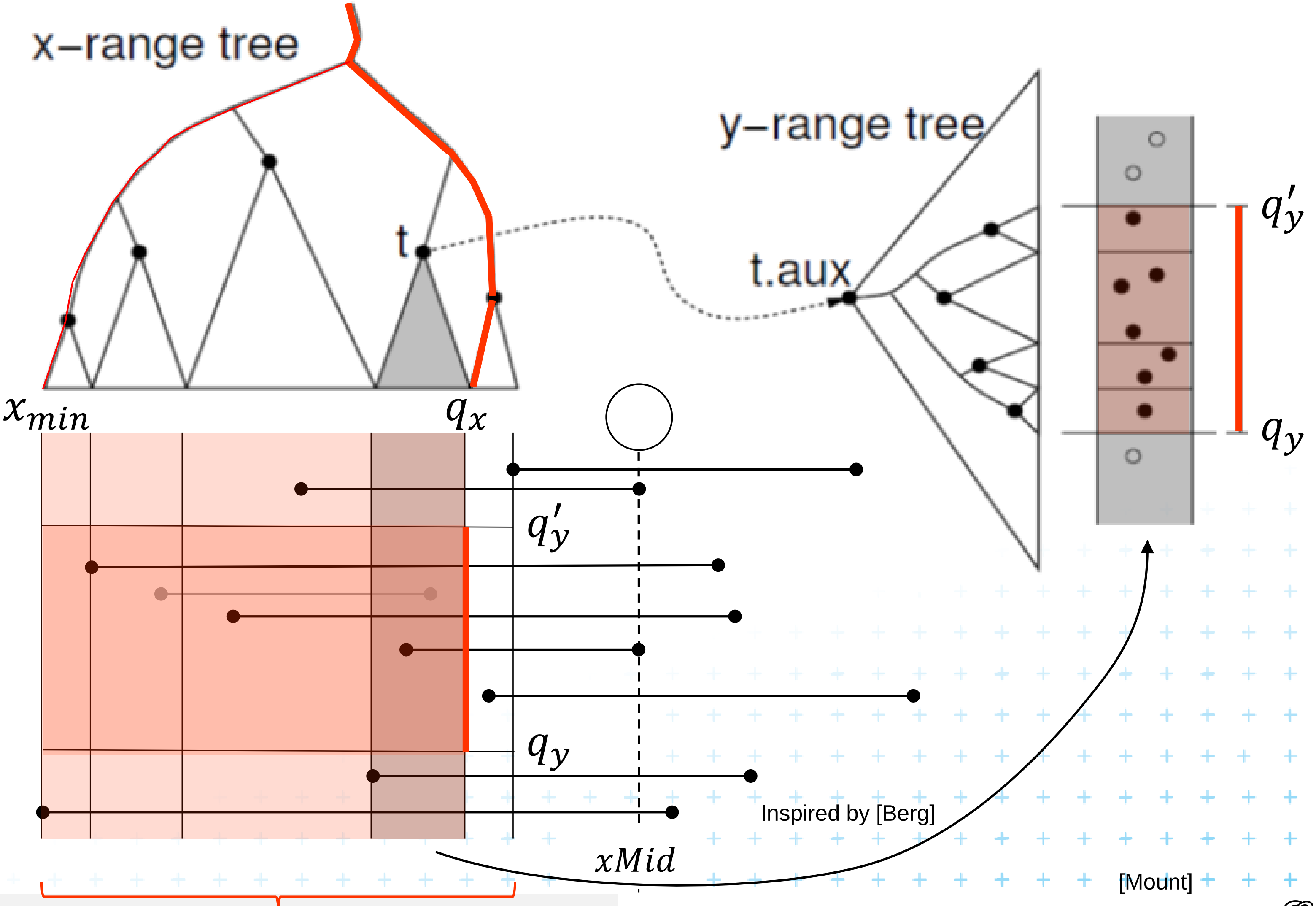


Inspired by [Berg]

Data structure for endpoints

- Storage of M_L and M_R
  - 1D Sorted lists is not enough for line segments
  - We need to test in y too
  - Use 2D range trees
    (one for M_L and one for M_R in each node)
- Instead $O(n)$ sequential search in M_L and M_R perform $O(\log n)$ search
  in range tree with fractional cascading

2D range tree (without fractional cascading-more in Lecture 3)



Inspired by [Berg]

Segment left end-points for M_L

[Mount]

Complexity of range tree line segment stabbing

- Construction - $O(n \log n)$ time
  - Each step divides at maximum into two halves L,R or less (minus elements of M) => int. tree height $O(\log n)$
  - If the range trees are efficiently build in $O(n)$ after points sorted
- Vertical line segment stab. q. - $O(k + \log^2 n)$ time
  - One node processed in $O(\log n + k')$, k' reported segm.

    interval tree

    2D range tree search with Fractional Cascading
  - v-visited nodes in $O(v \log n + k)$, k total reported segm.

    interval tree
  - v = interval tree height = $O(\log n)$

    $\mathrm{k} = \sum k'$
  - $O(k + \log^2 n)$ time - range tree with fractional cascading
  - $O(k + \log^3 n)$ time - range tree without fractional casc.
- Storage - $O(n \log n)$
  - Dominated by the range trees

Complexity of range tree line segment stabbing

- Construction - $O(n \log n)$ time
  - Each step divides at maximum into two halves L,R or less (minus elements of M) => int. tree height $O(\log n)$
  - If the range trees are efficiently build in $O(n)$ after points sorted
- Vertical line segment stab. q. - $O(k + \log^2 n)$ time
  - One node processed in $O(\underset{\text{interval tree}}{\log n} + \overset{\text{2D range tree search with Fractional Cascading}}{k'}), k'$ reported segm.
  - v-visited nodes in $O(v \log n + k), k$ total reported segm.

    interval tree
  - v = interval tree height = $O(\log n)$ $\qquad \mathrm{k} = \sum k'$
  - $O(k + \log^2 n)$ time - range tree with fractional cascading
  - $O(k + \log^3 n)$ time - range tree without fractional casc.
- Storage - $O(n \log n)$ Can be done better?
  - Dominated by the range trees

Talk overview

1. Windowing of axis parallel line segments in 2D
   - 3 variants of *interval tree – IT in x-direction*
   - Differ in storage of segment end points M_L and M_R

1D i. Line stabbing (standard *IT* with *sorted lists*) lecture 9 - intersections

2D ii. Line segment stabbing (*IT* with *range trees*)

iii. Line segment stabbing (*IT* with *priority search trees*)

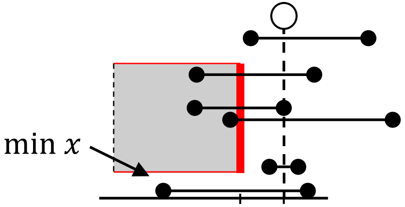
2. Windowing of line segments in general position

2D – *segment tree + BST*

iii. Priority search trees

[McCreight85]

- Another variant for case c) on slide 9
  - Exploit the fact that query rectangle in each node in interval tree is unbounded (in x direction)



- Priority search trees
  - as secondary data structure for both left and right endpoints (M_L and M_R) of segments in nodes of interval tree – one for ML, one for MR
  - Improve the storage to $O(n)$ for horizontal segment intersection with left window edge (2D range tree has $O(n \log n)$)
- For cases a) and b) - $O(n \log n)$ storage remains
  - we need range trees for windowing segment endpoints

Rectangular range queries variants

- Let $P = \{p_1, p_2, \dots, p_n\}$ is set of points in plane
- Goal: rectangular range queries of the form
  $\underbrace{(-\infty : q_x]}\ \times\ [q_y : q'_y\,]$ – unbounded (in x direction)
- In 1D: search for nodes v with $v_x \in (-\infty : q_x]$
  - range tree $O(\log n\ +\ k)$ time (search the end, report left)
  - ordered list $O(1\ +\ k)$ time 1 is for possibly fail test of the first
    (start in the leftmost, stop on v with $v_x > q_x$)
  - use heap $O(1\ +\ k)$ time !
    (traverse all children, stop when $v_x > q_x$)
- In 2D – use heap for points with $x \in (-\infty :\ q_x]$

+ integrate information about y-coordinate

Rectangular range queries variants

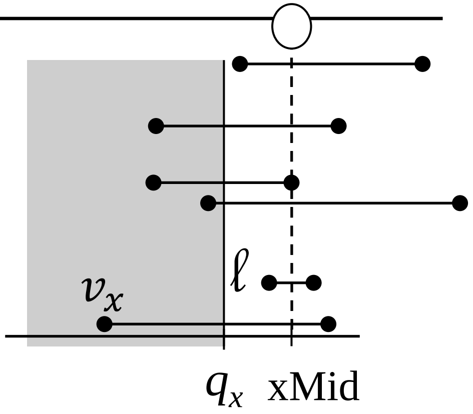
- Let $P = \{p_1, p_2, \dots, p_n\}$ is set of points in plane
- Goal: rectangular range queries of the form
  $\underbrace{(-\infty : q_x]}_{} \times [q_y : q'_y] -$ unbounded (in x direction)
- In 1D: search for nodes v with $v_x \in (-\infty : q_x]$
  - range tree $O(\log n + k)$ time (search the end, report left)
  - ordered list $O(1 + k)$ time 1 is for possibly fail test of the first
    (start in the leftmost, stop on v with $v_x > q_x$)
  - use heap $O(1 + k)$ time !
    (traverse all children, stop when $v_x > q_x$)
- In 2D – use heap for points with $x \in (-\infty : q_x]$

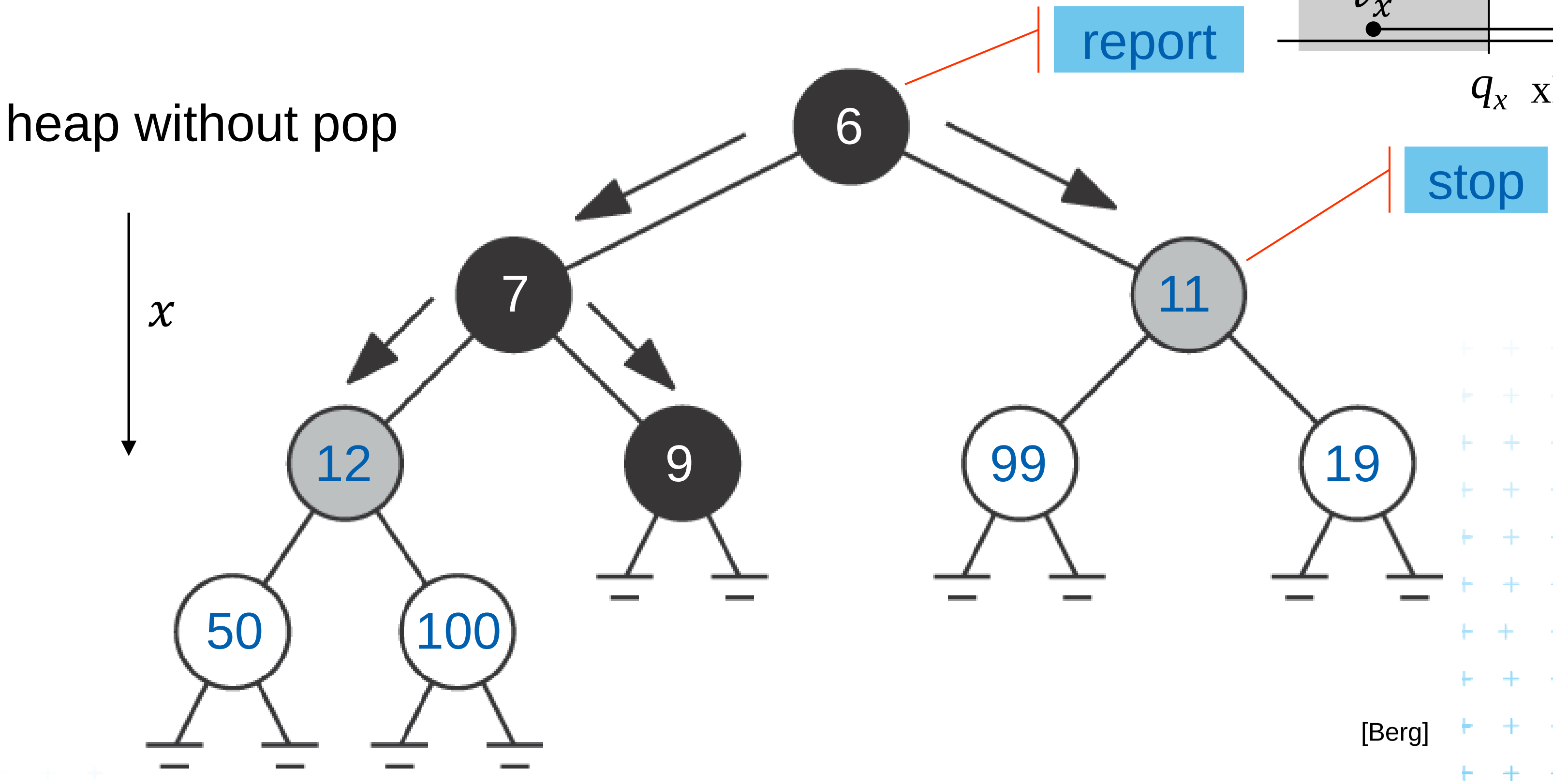
+ integrate information about y-coordinate

= Priority search tree

Heap for 1D unbounded range queries

- Traverse all children, stop if $v_x > q_x$
- Example: Query $(-\infty : 10]$, $q_x = 10$

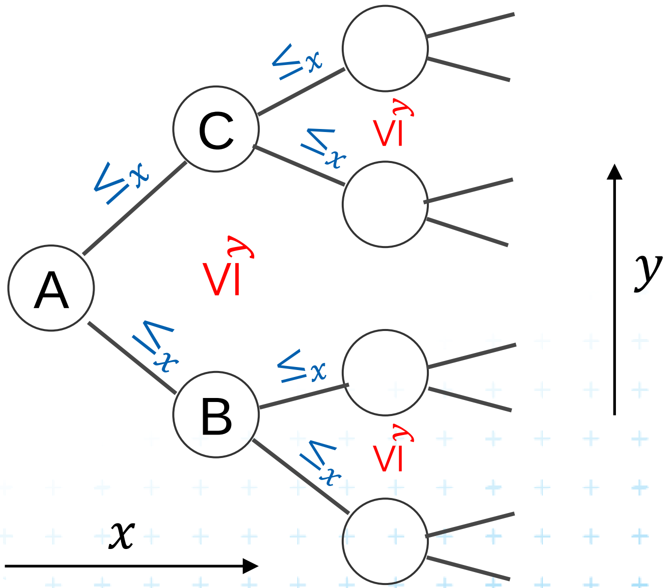




[Berg]

Principle of priority search tree

- Heap $\leq_x$
  - relation between parent and its child nodes only
  - no relation between the child nodes themselves
- Priority search tree
  - relate the child nodes according to y $\leq_y$



x Heap

A $\leq_x$ B

A $\leq_x$ C

y BVS

B $\leq_y$ A $\leq_y$ C $\Rightarrow$ B $\leq_y$ C

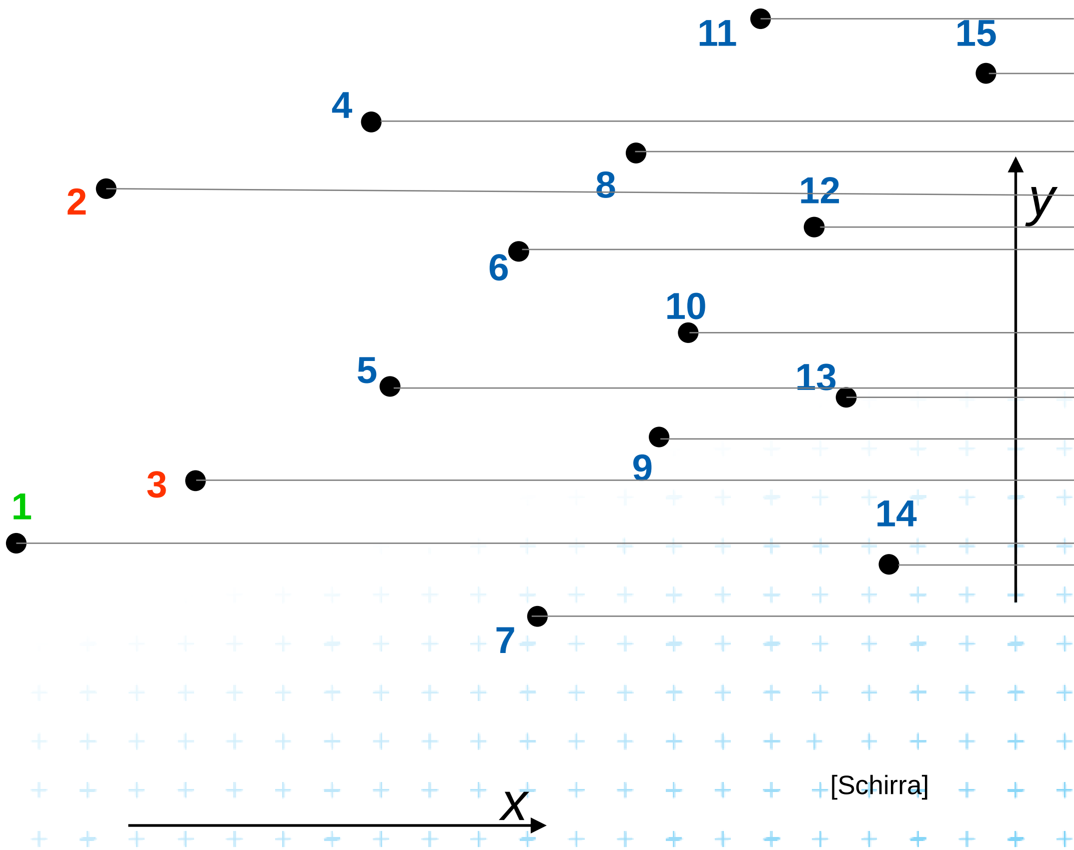
Priority search tree (PST)

= Heap in 2D that can incorporate info about both x, y

- BST on y-coordinate (horizontal slabs) ~ 1D range tree
- Heap on x-coordinate (minimum *x* from slab along *x*)

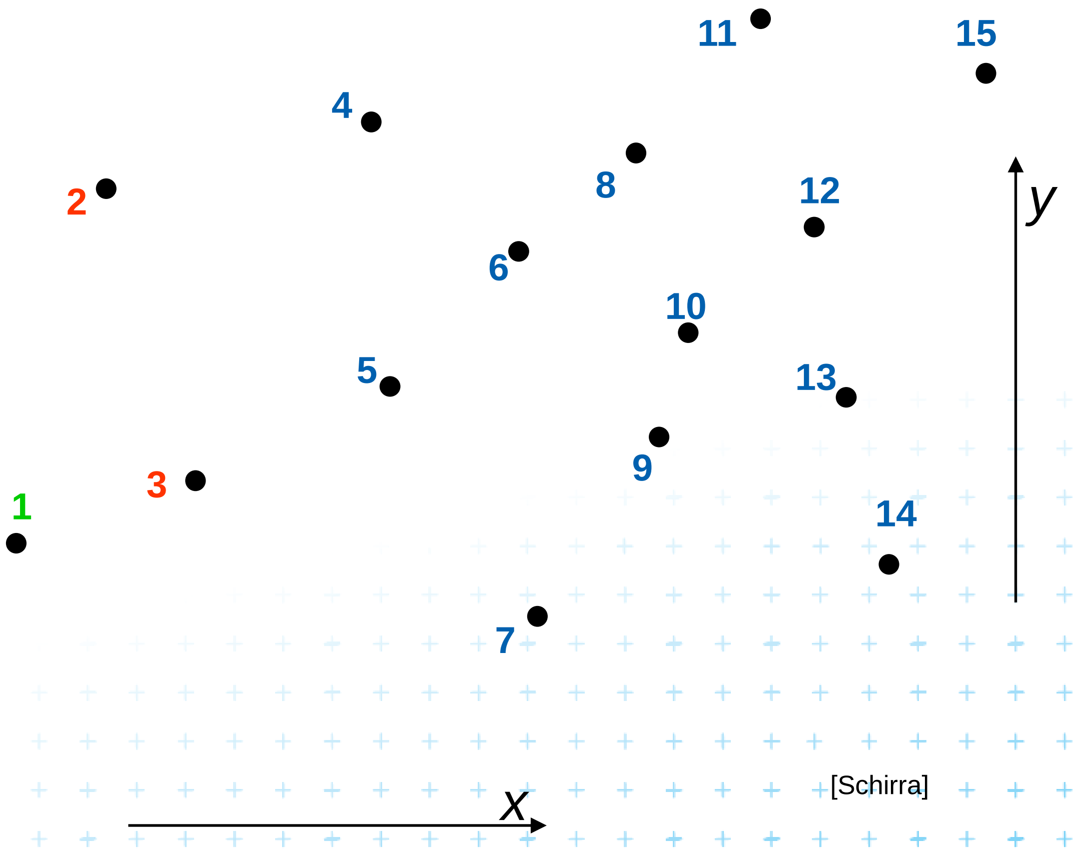
- If P is empty, PST is empty leaf
- else
  - p_{min} = point with smallest x-coordinate in P – a heap root
  - y_{med} = y-coord. median of points $P \setminus \{p_{min}\}$ – BST root
  - $P_{below} \coloneqq \{ p \in P \setminus \{p_{min}\} \ : p_y \leq y_{med} \}$
  - $P_{above} \coloneqq \{ p \in P \setminus \{p_{min}\} \ : p_y > y_{med} \}$
- Point p_{min} and scalar y_{med} are stored in the PST root
- The left subtree is PST of P_{below}
- The right subtree is PST of P_{above}

Priority search tree construction example



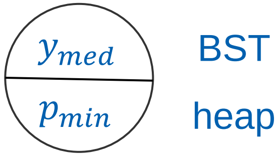
[Schirra]

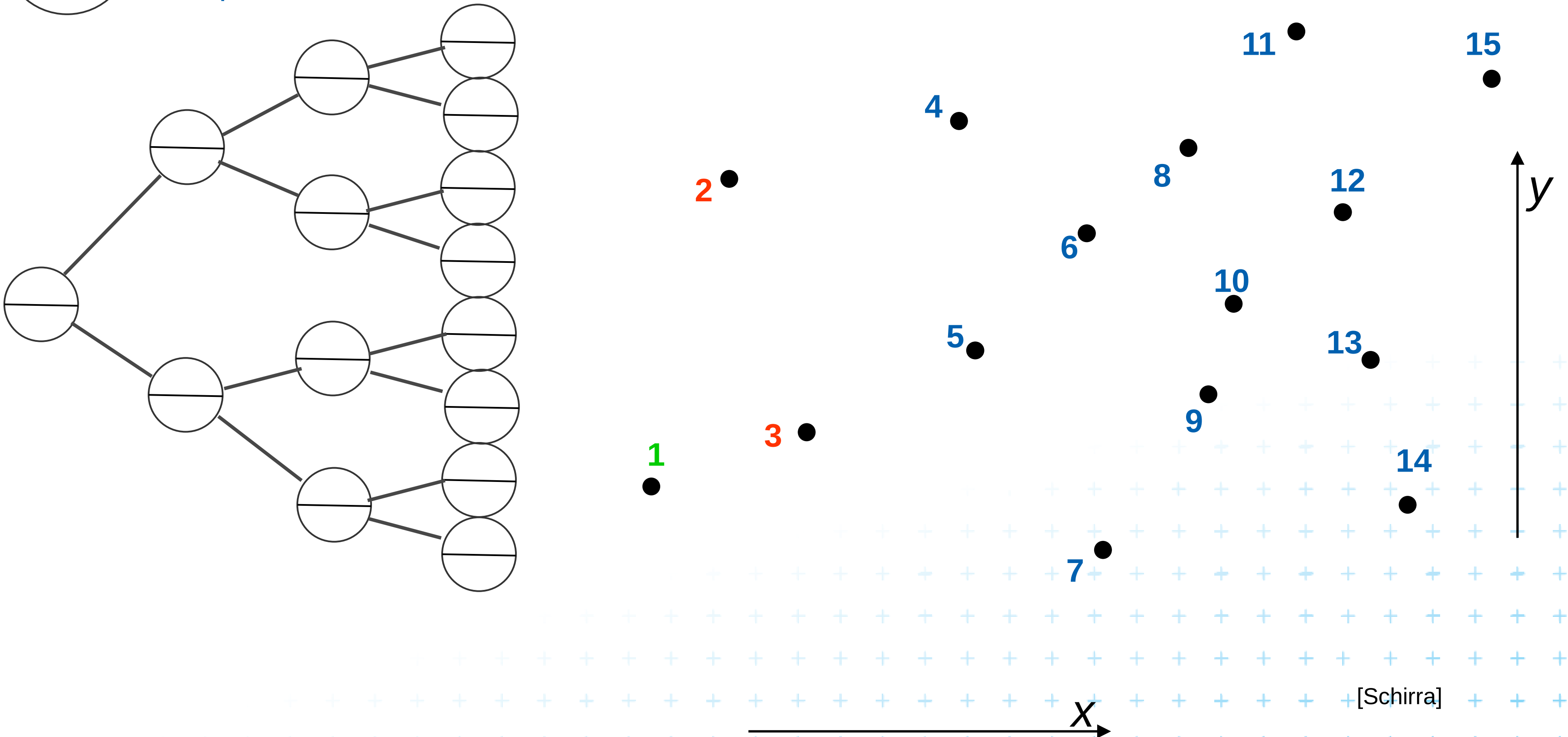
Priority search tree construction example



[Schirra]

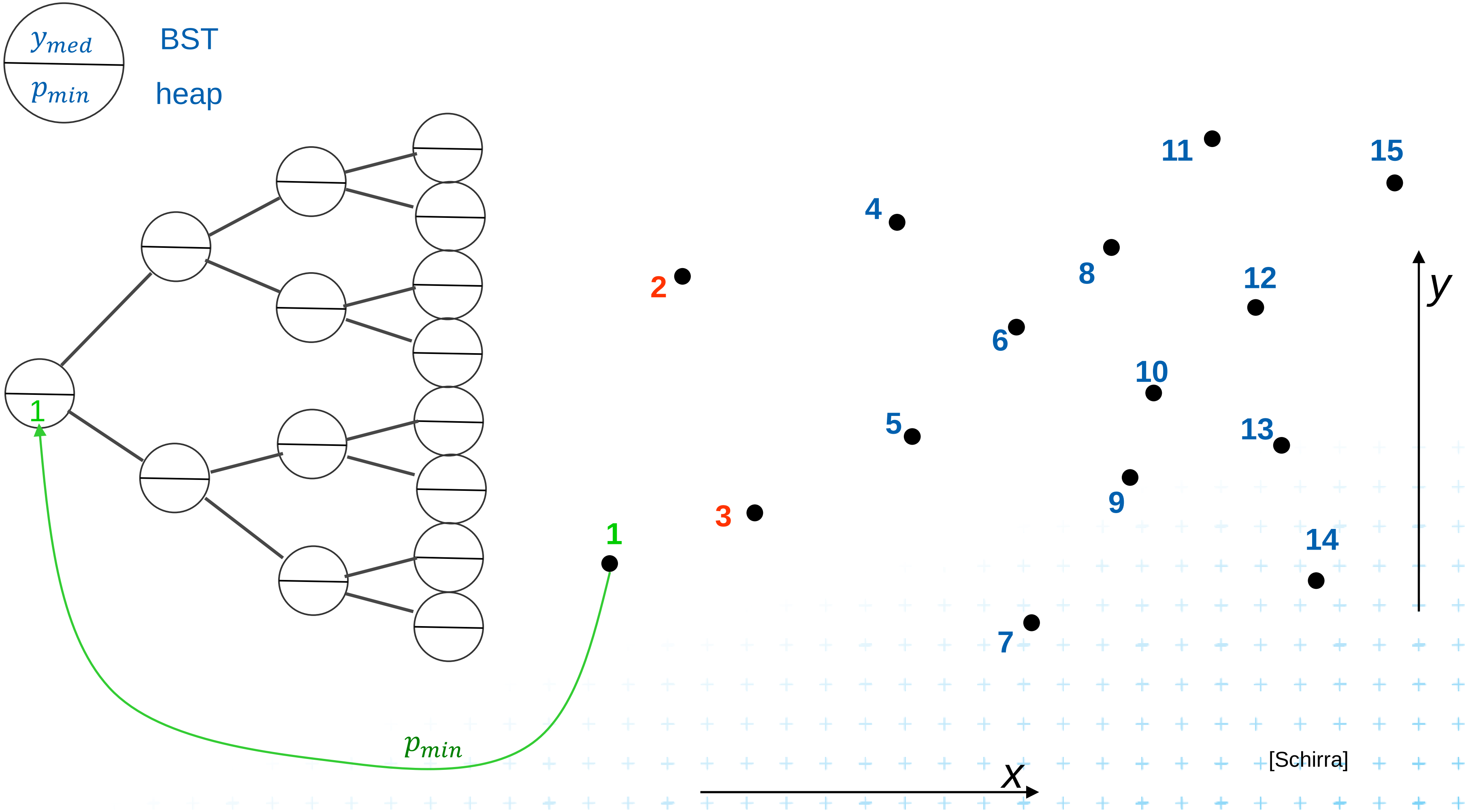
Priority search tree construction example





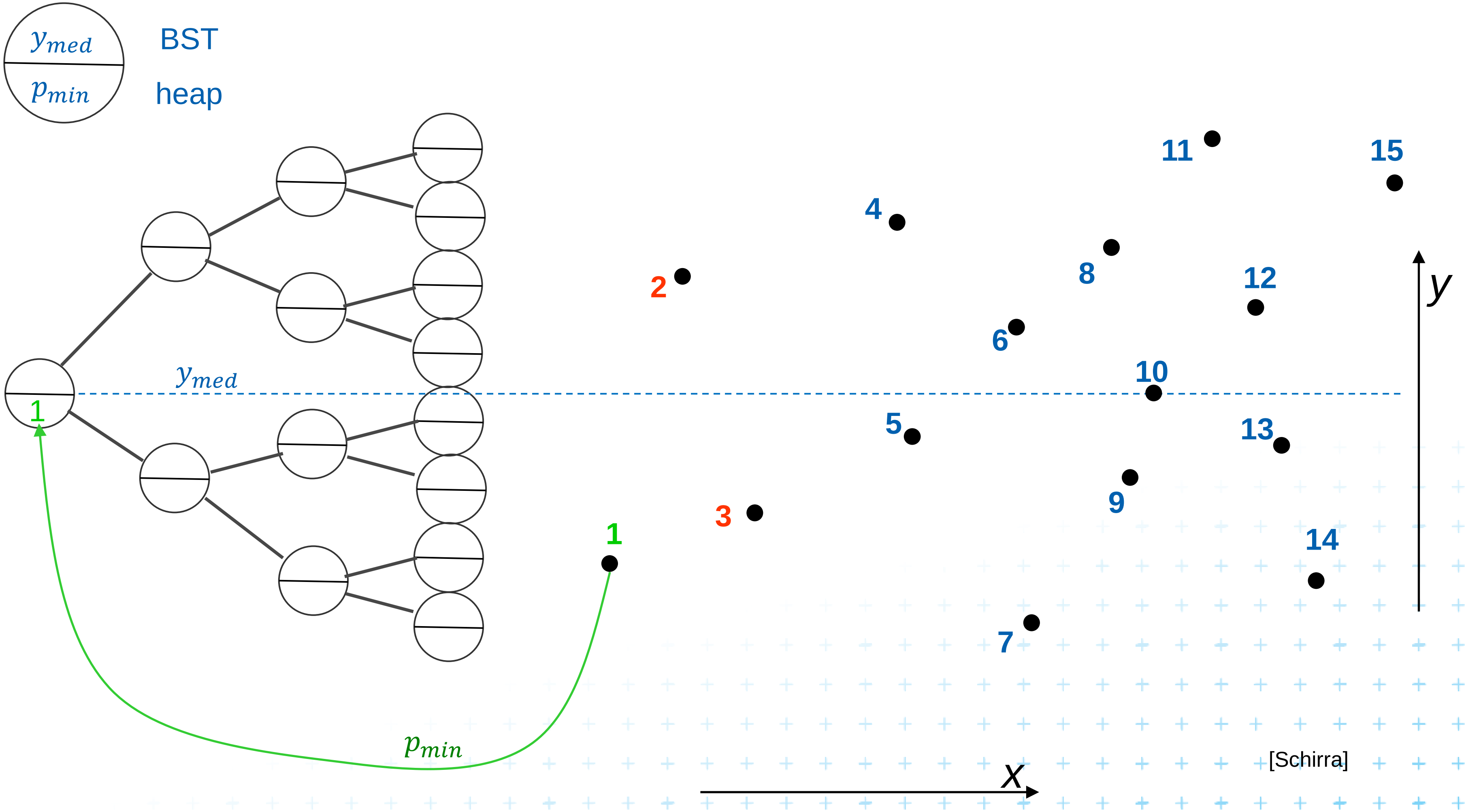
[Schirra]

Priority search tree construction example



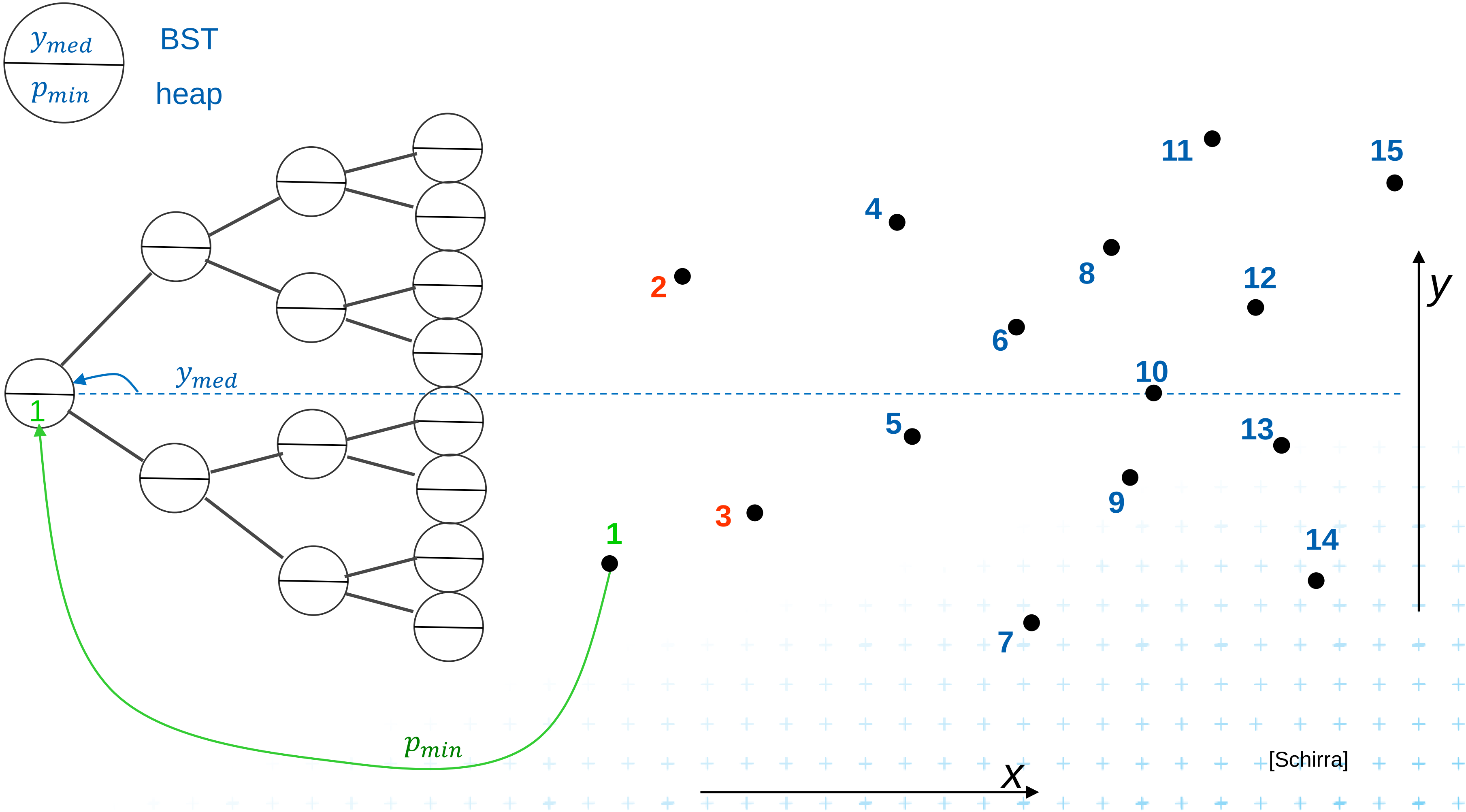
[Schirra]

Priority search tree construction example



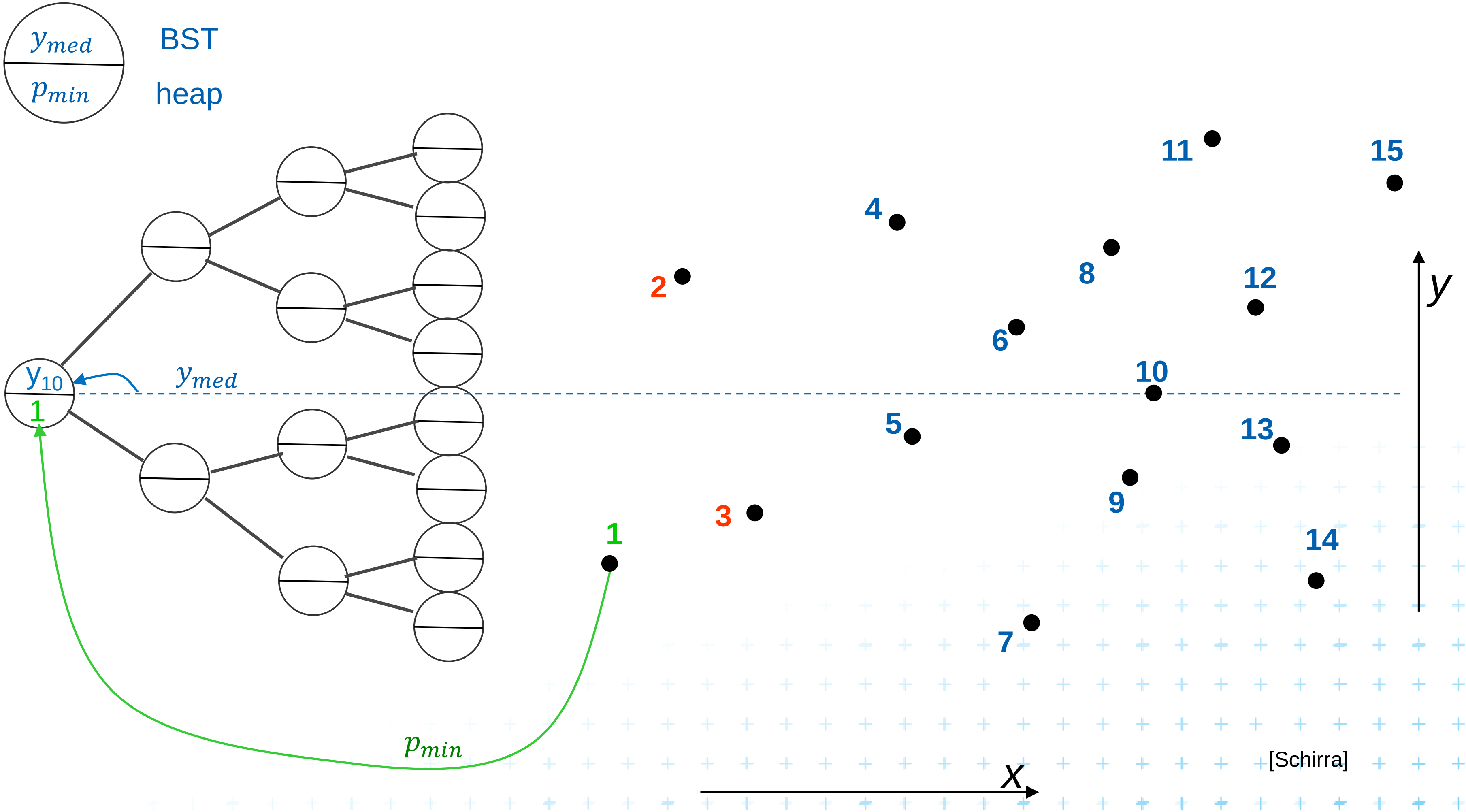
[Schirra]

Priority search tree construction example



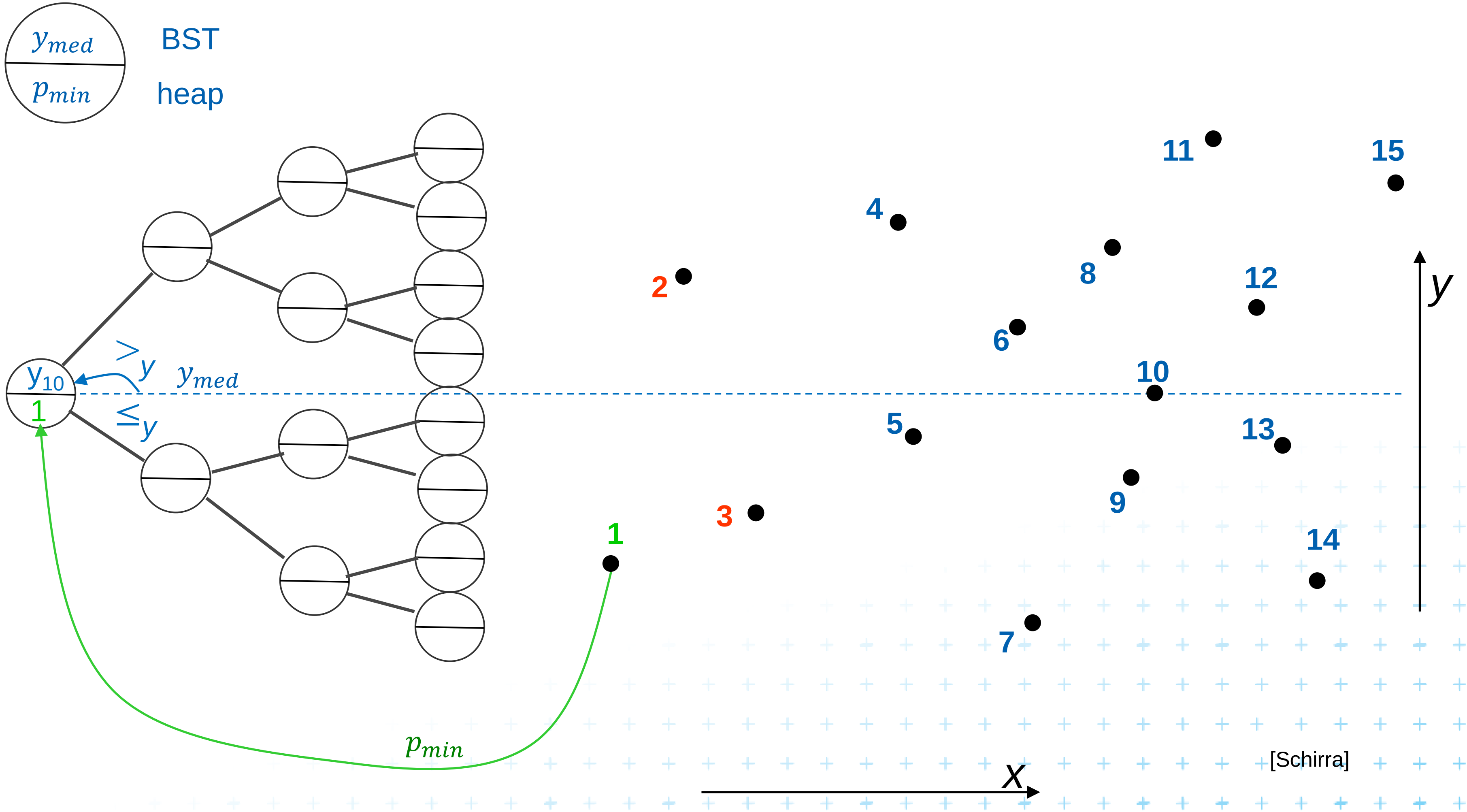
[Schirra]

Priority search tree construction example



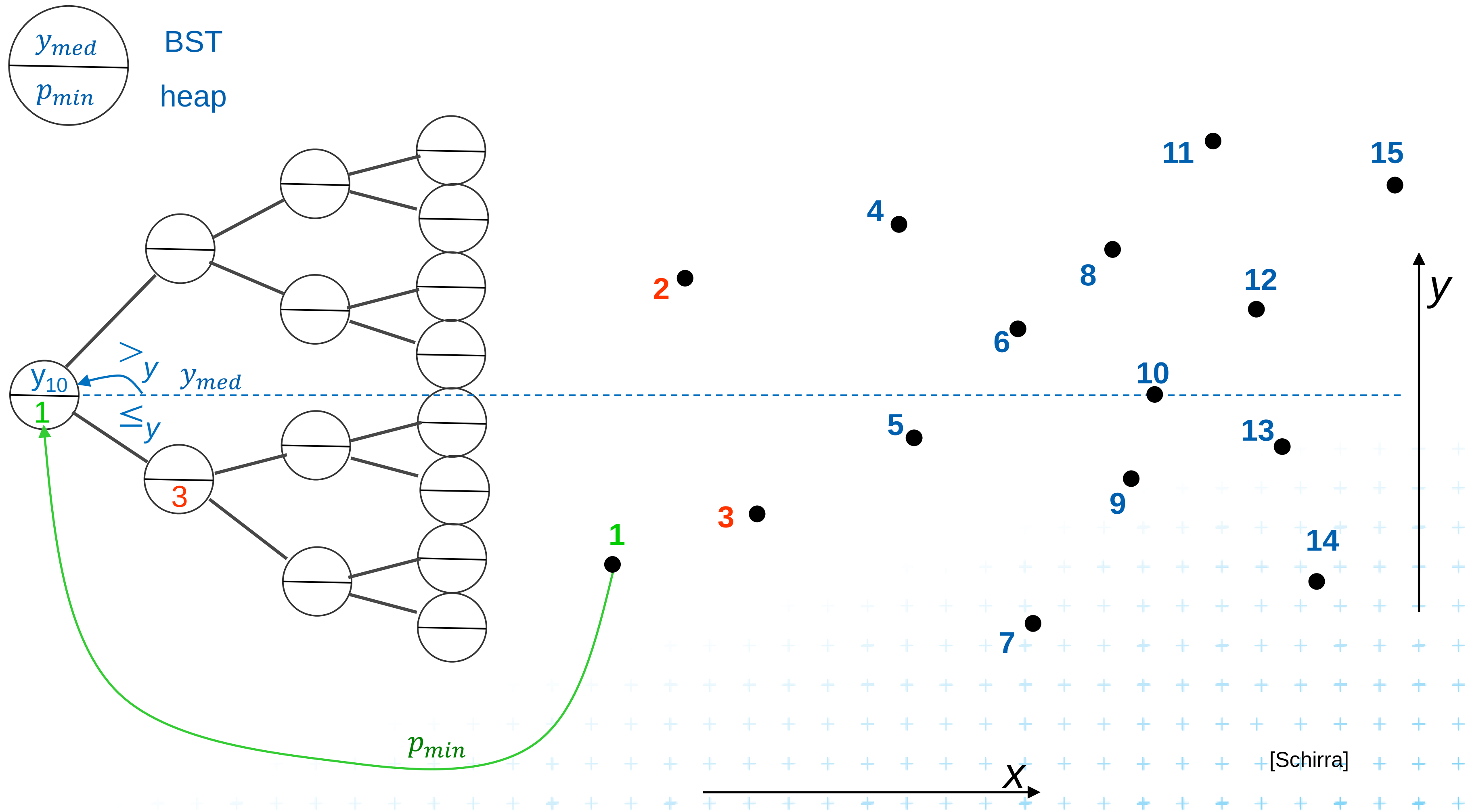
[Schirra]

Priority search tree construction example



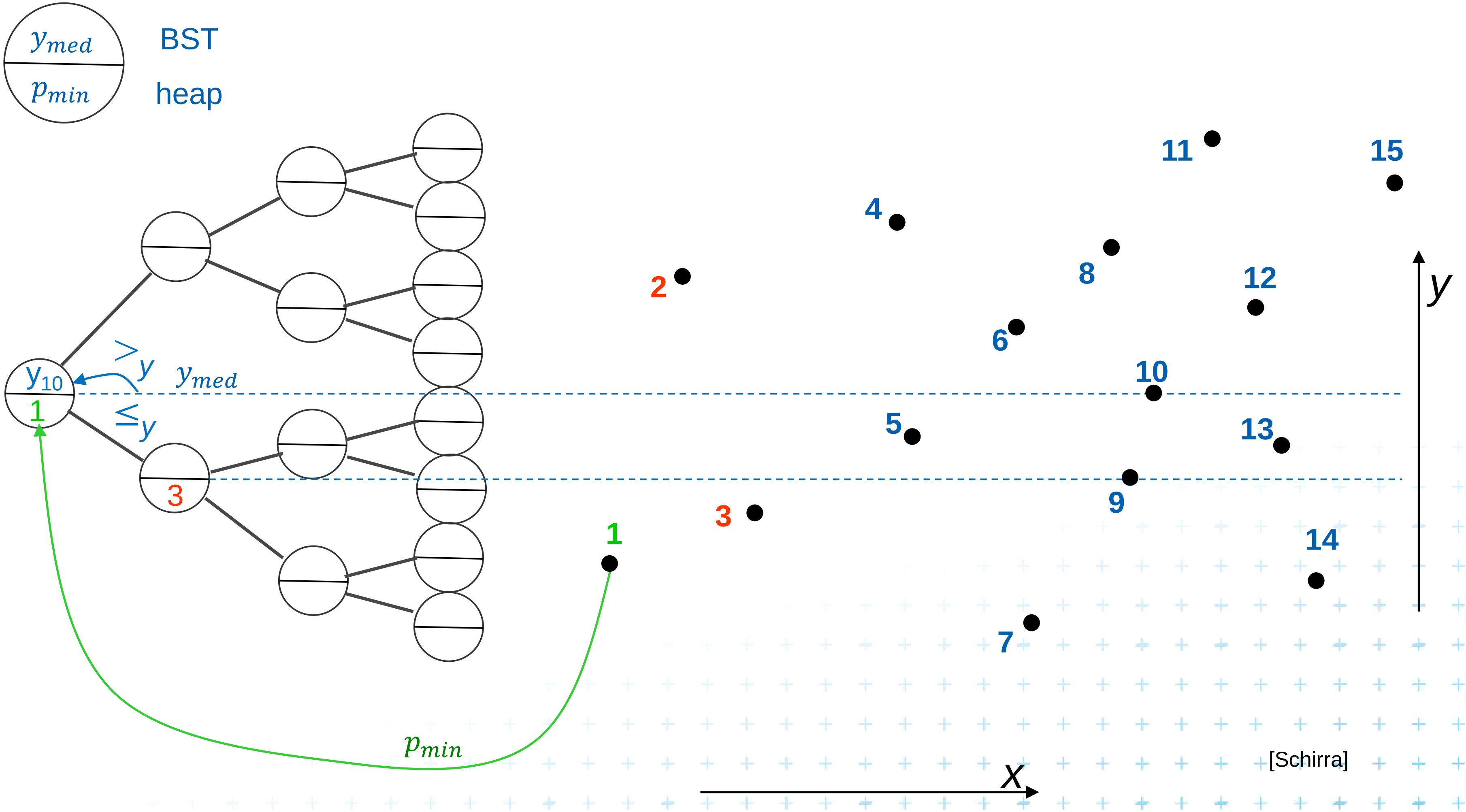
[Schirra]

Priority search tree construction example



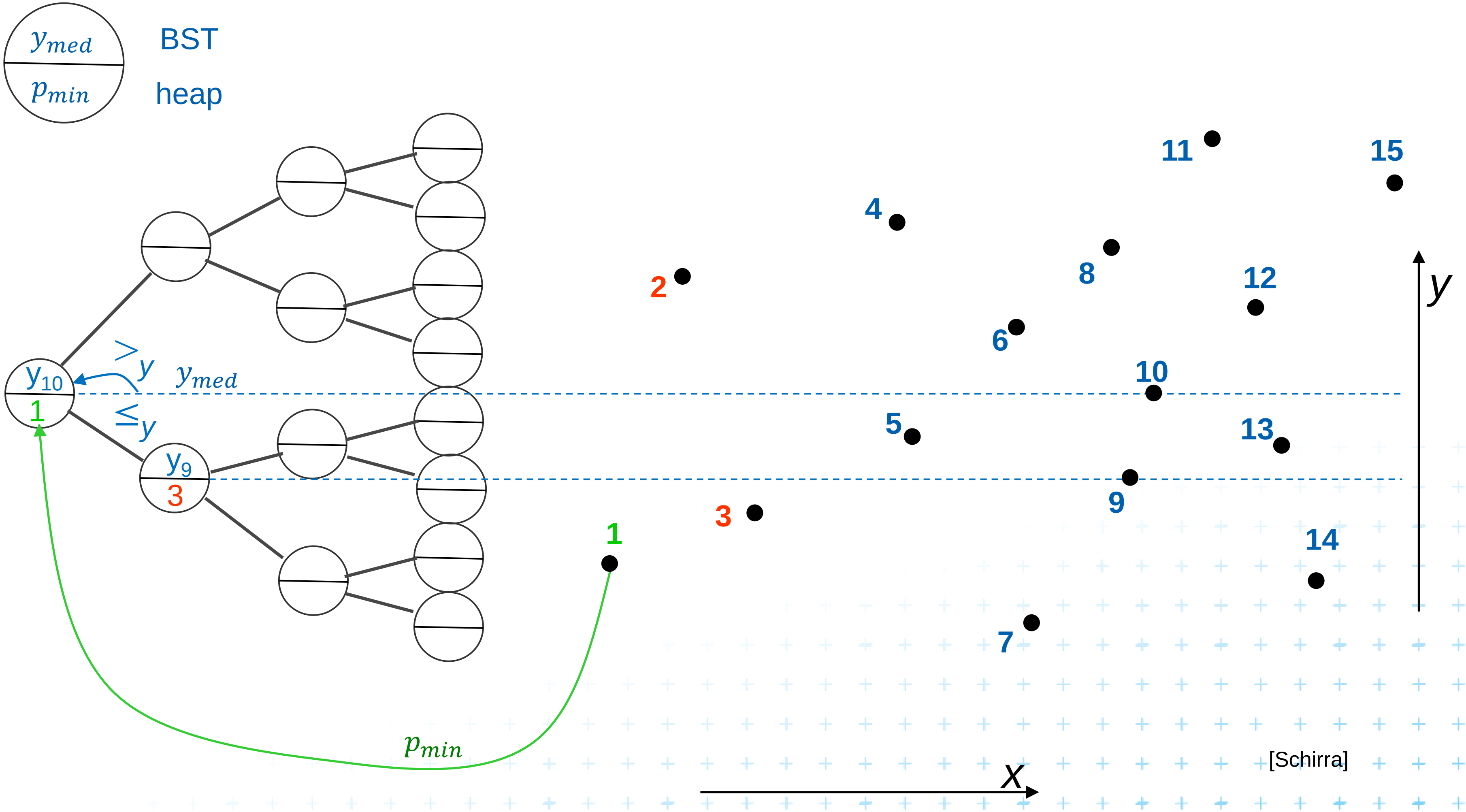
[Schirra]

Priority search tree construction example



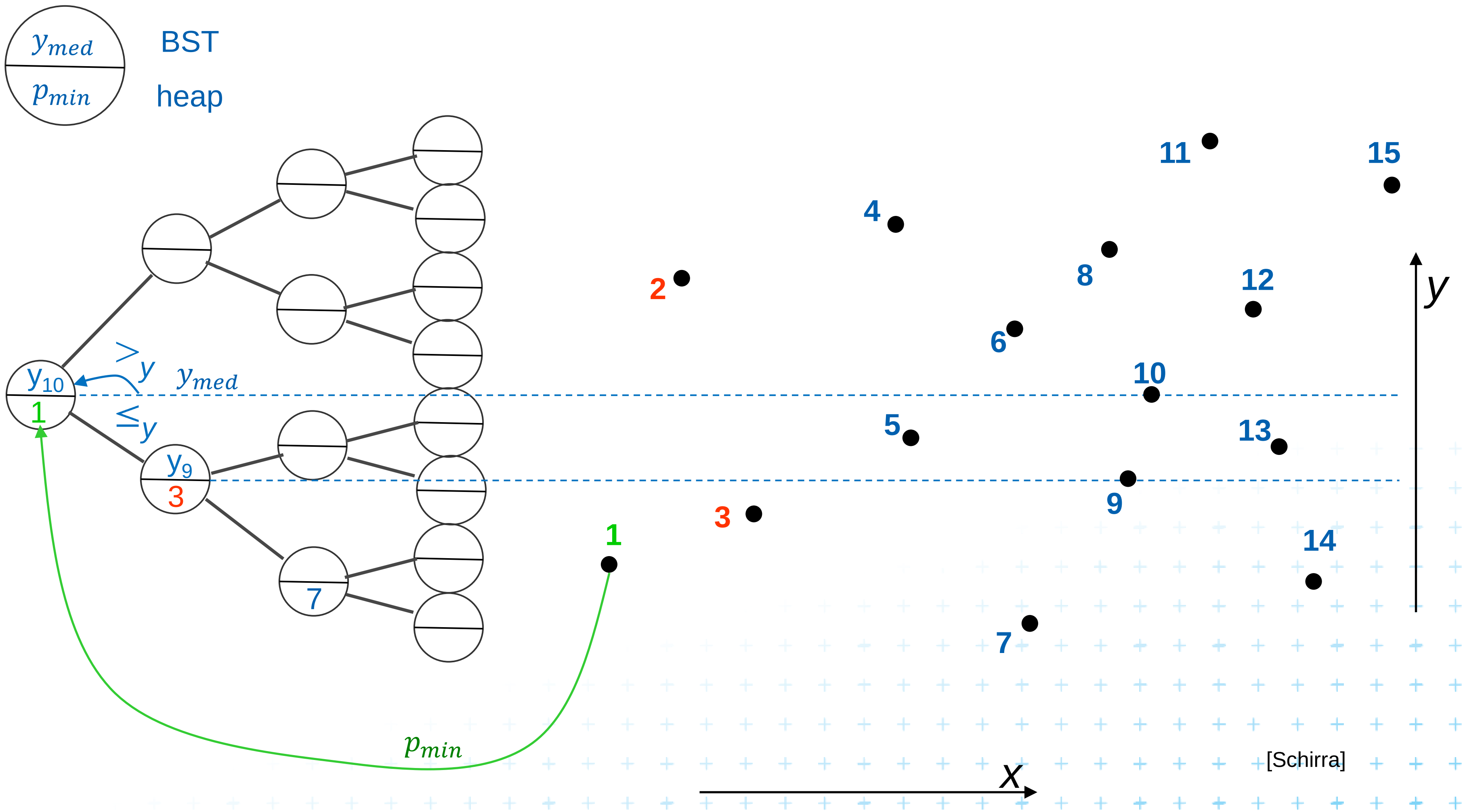
[Schirra]

Priority search tree construction example



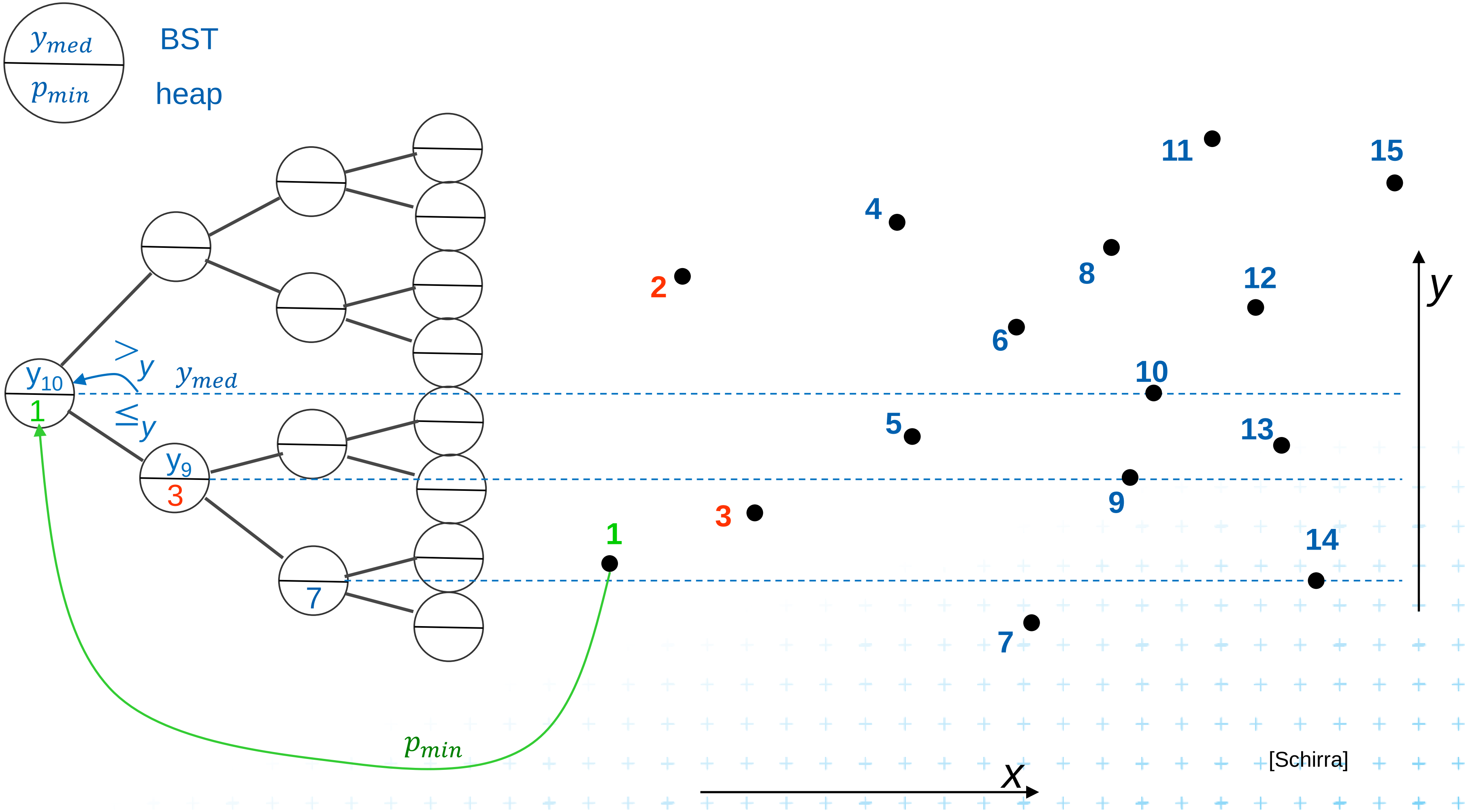
[Schirra]

Priority search tree construction example



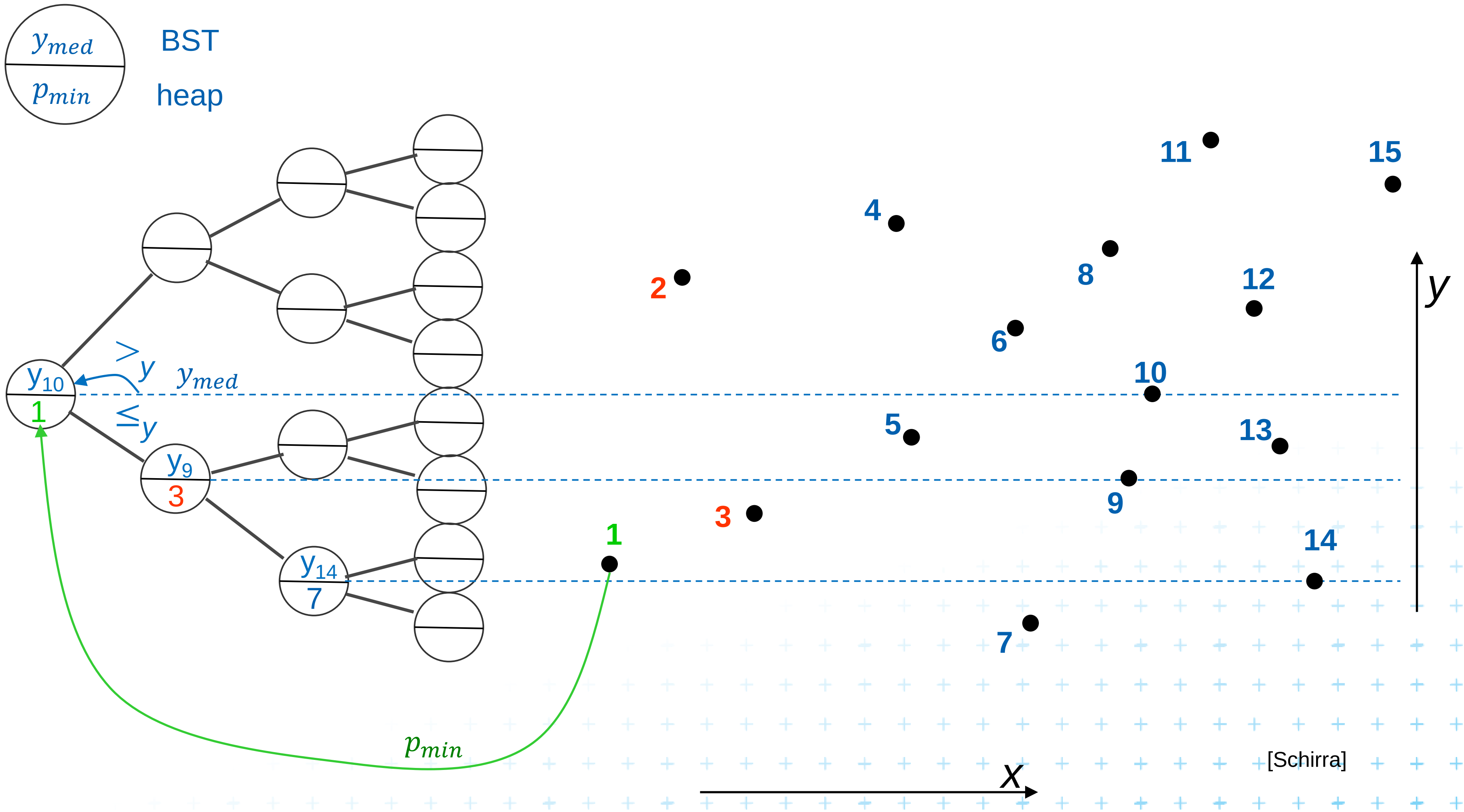
[Schirra]

Priority search tree construction example



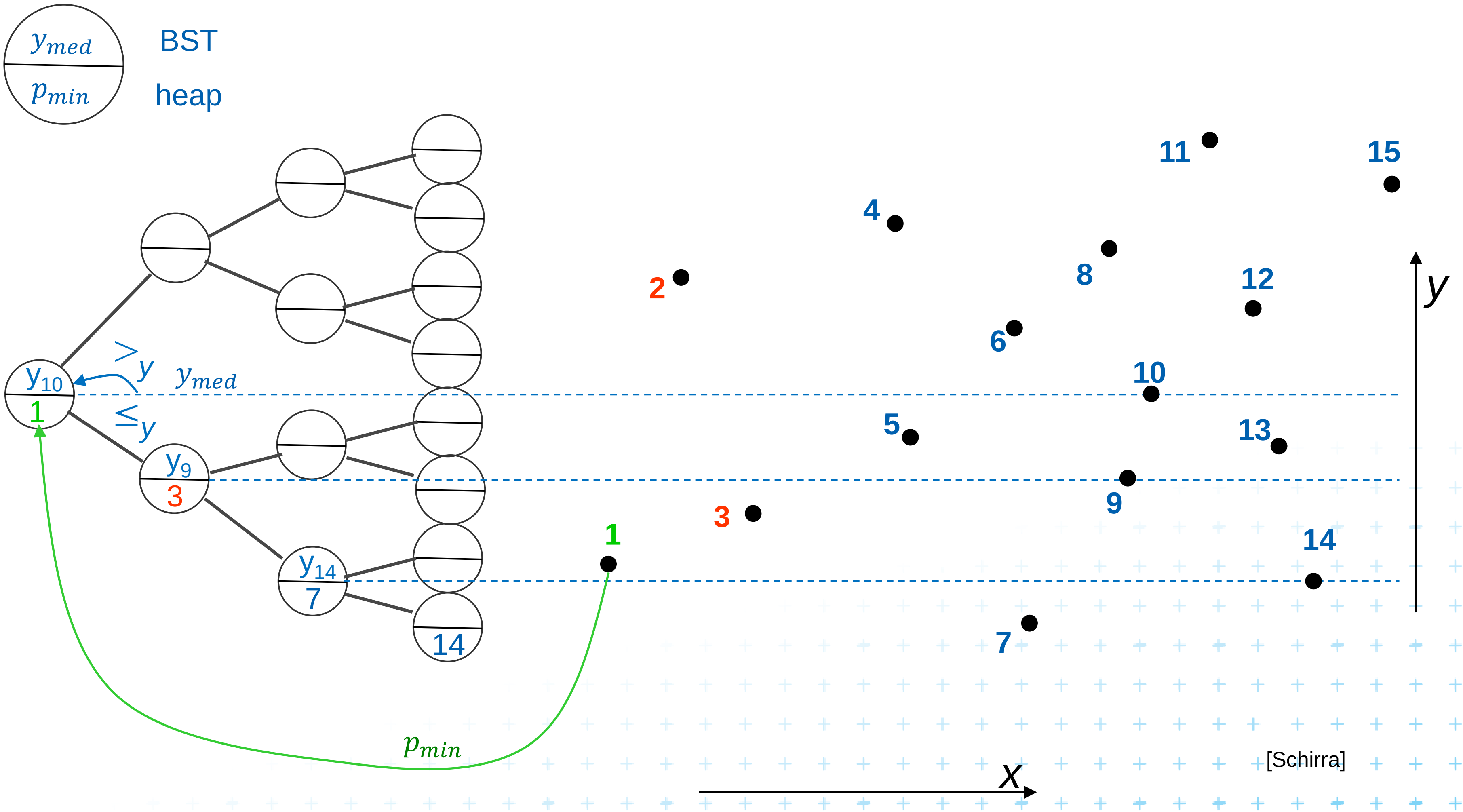
[Schirra]

Priority search tree construction example



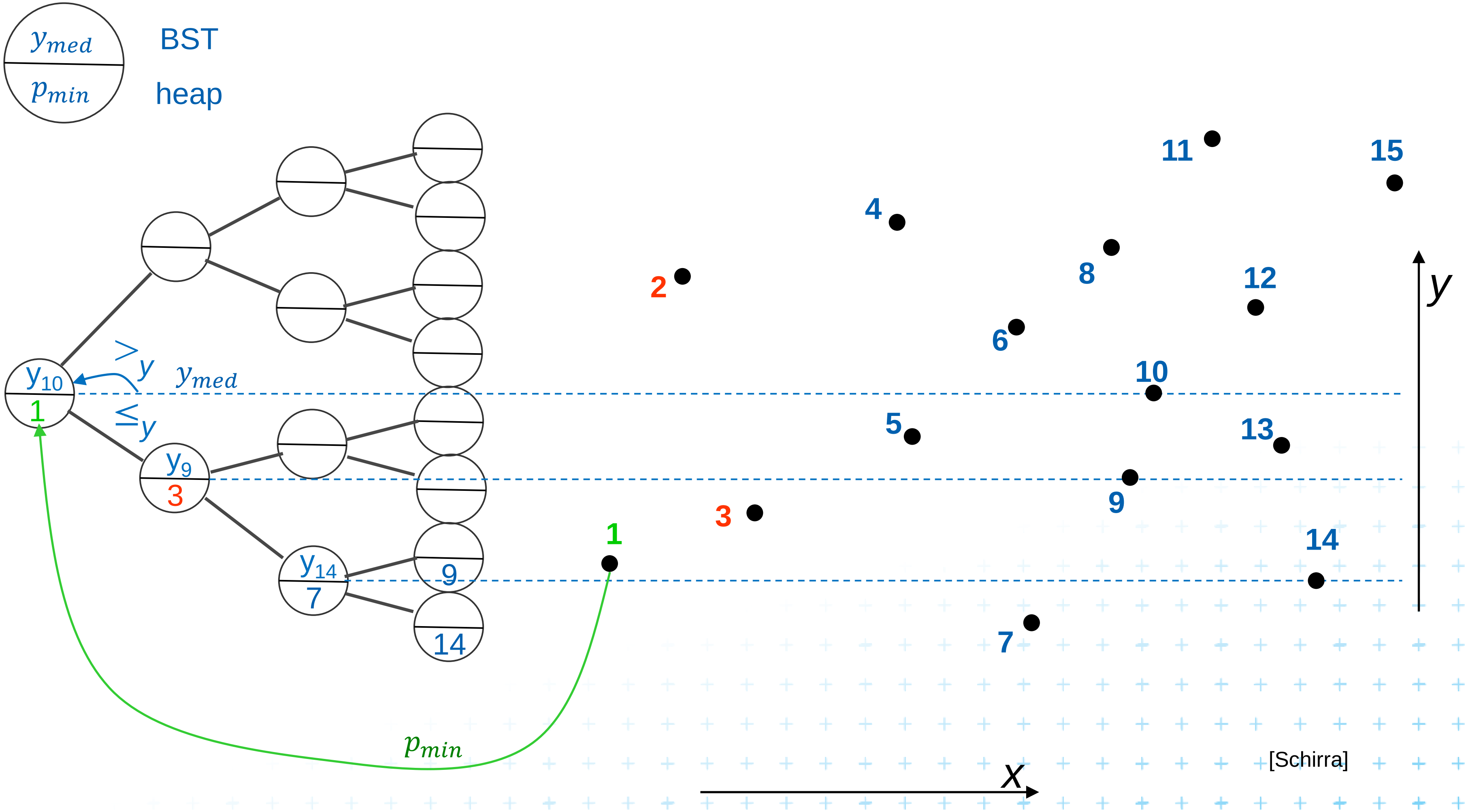
[Schirra]

Priority search tree construction example

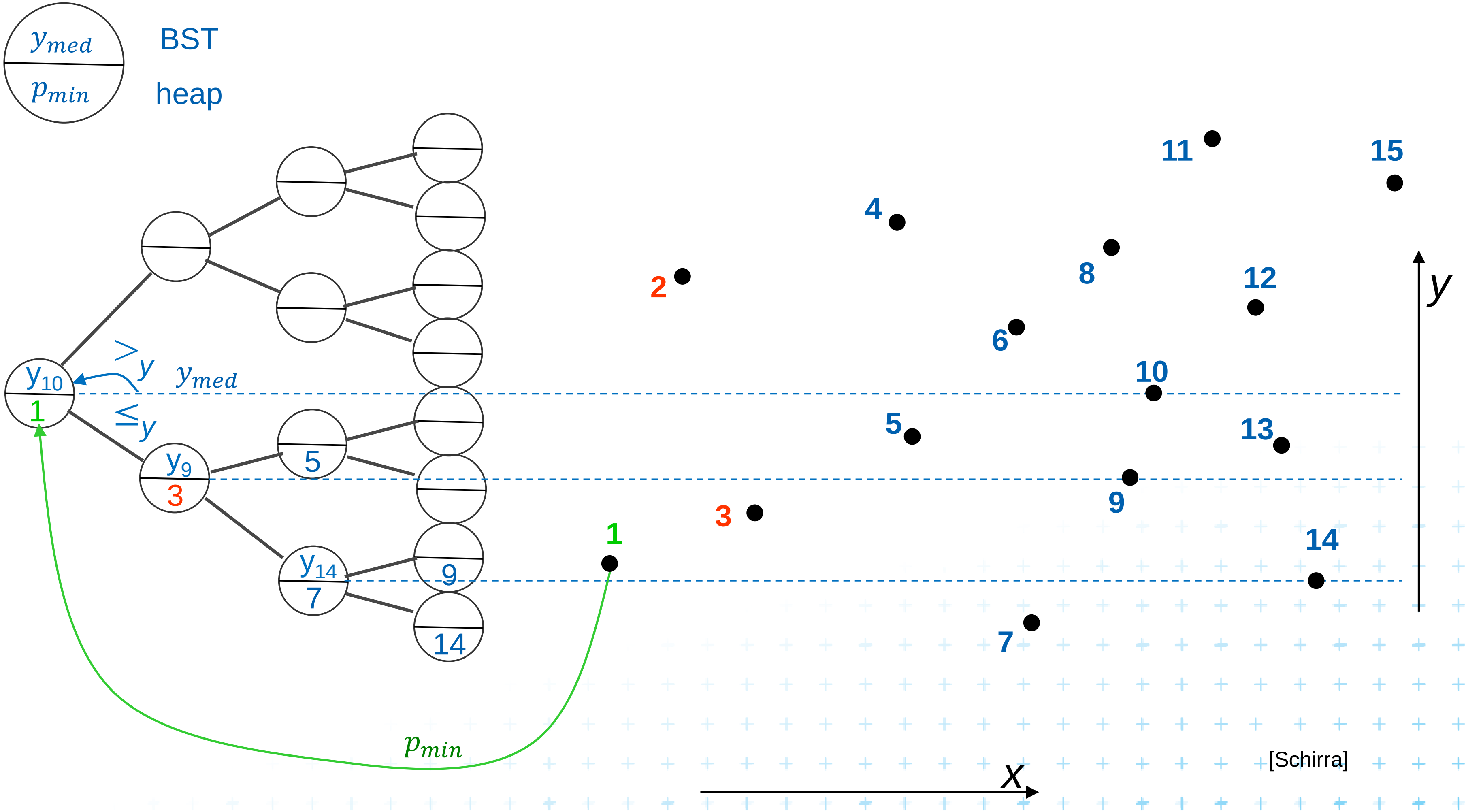


[Schirra]

Priority search tree construction example

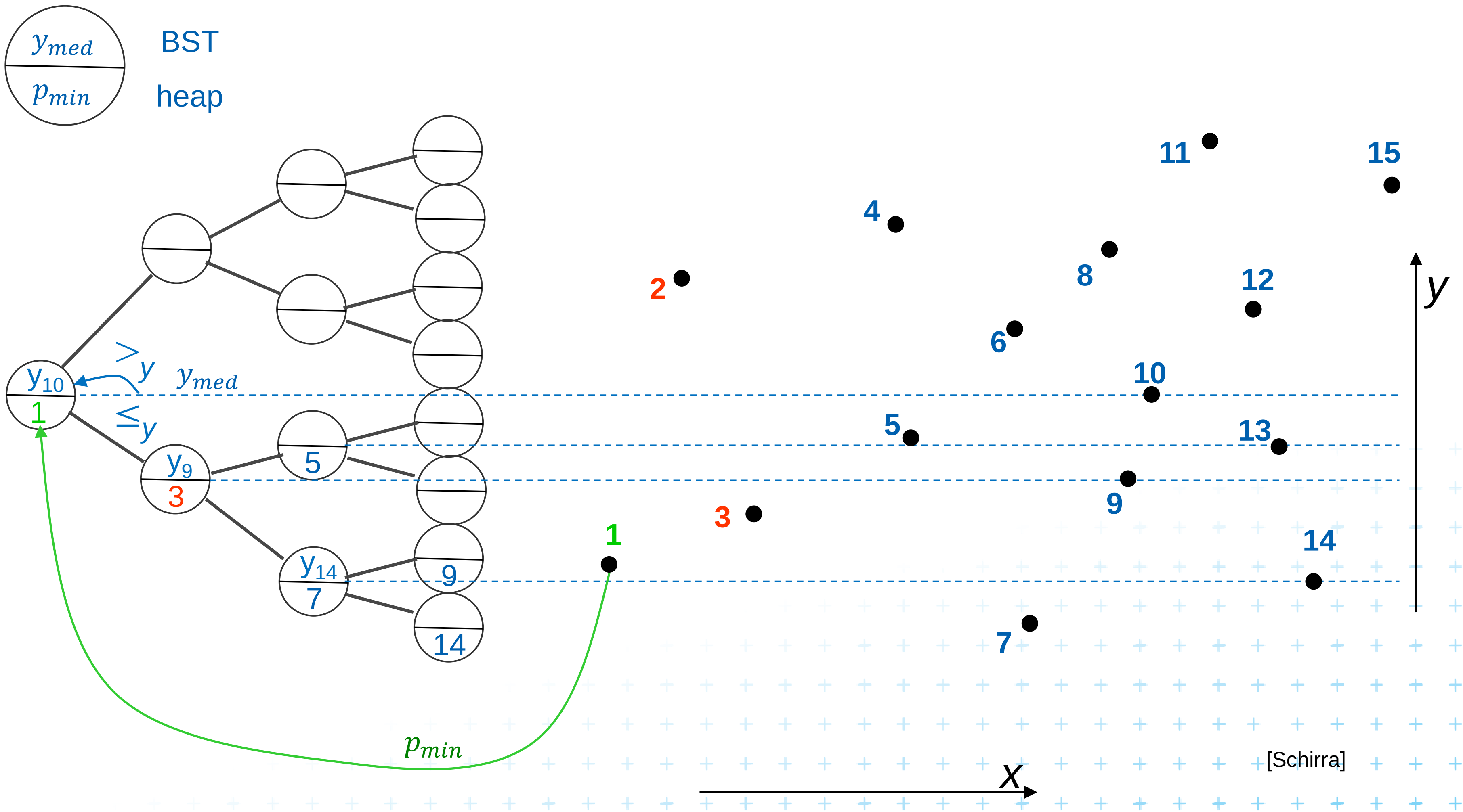


Priority search tree construction example



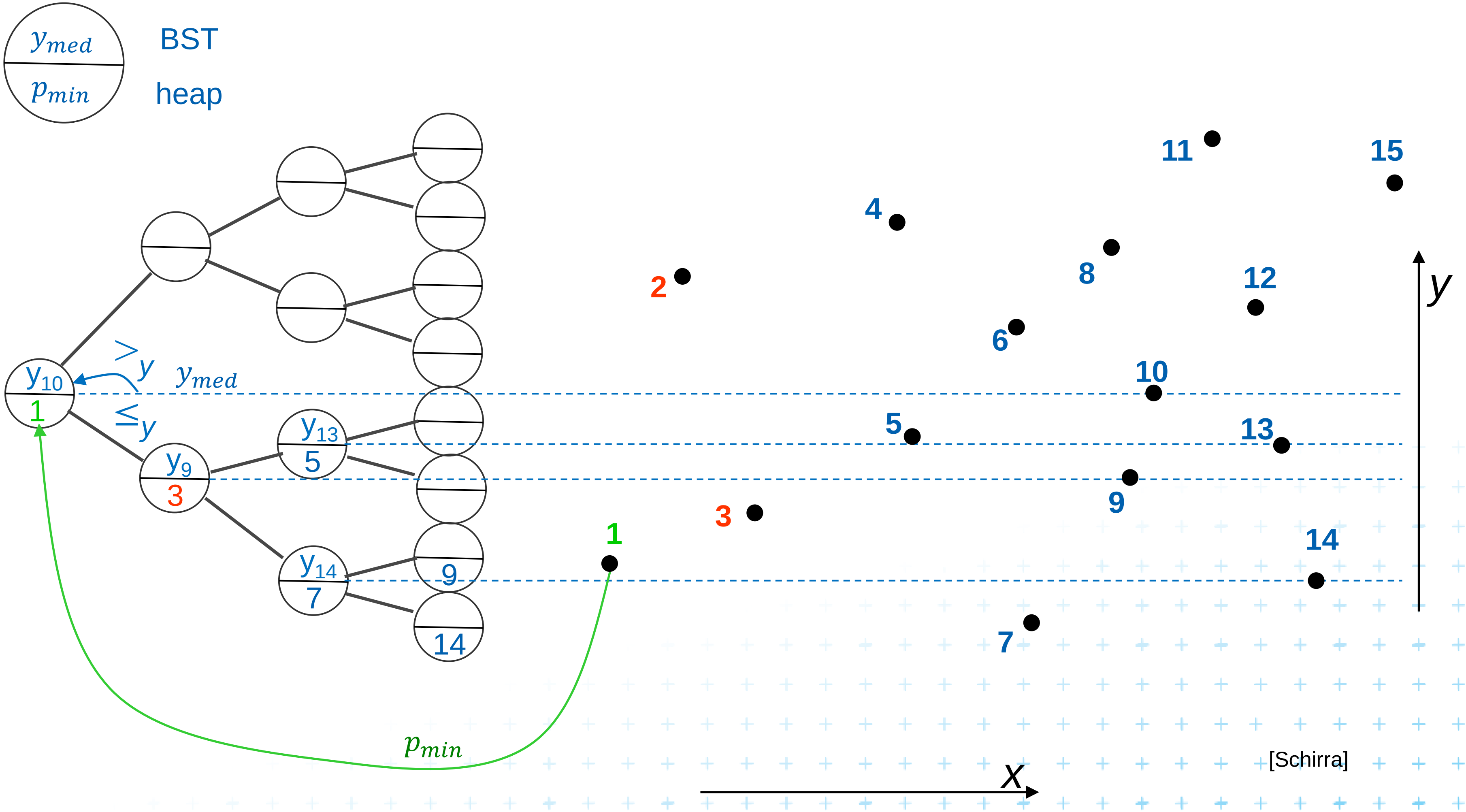
[Schirra]

Priority search tree construction example



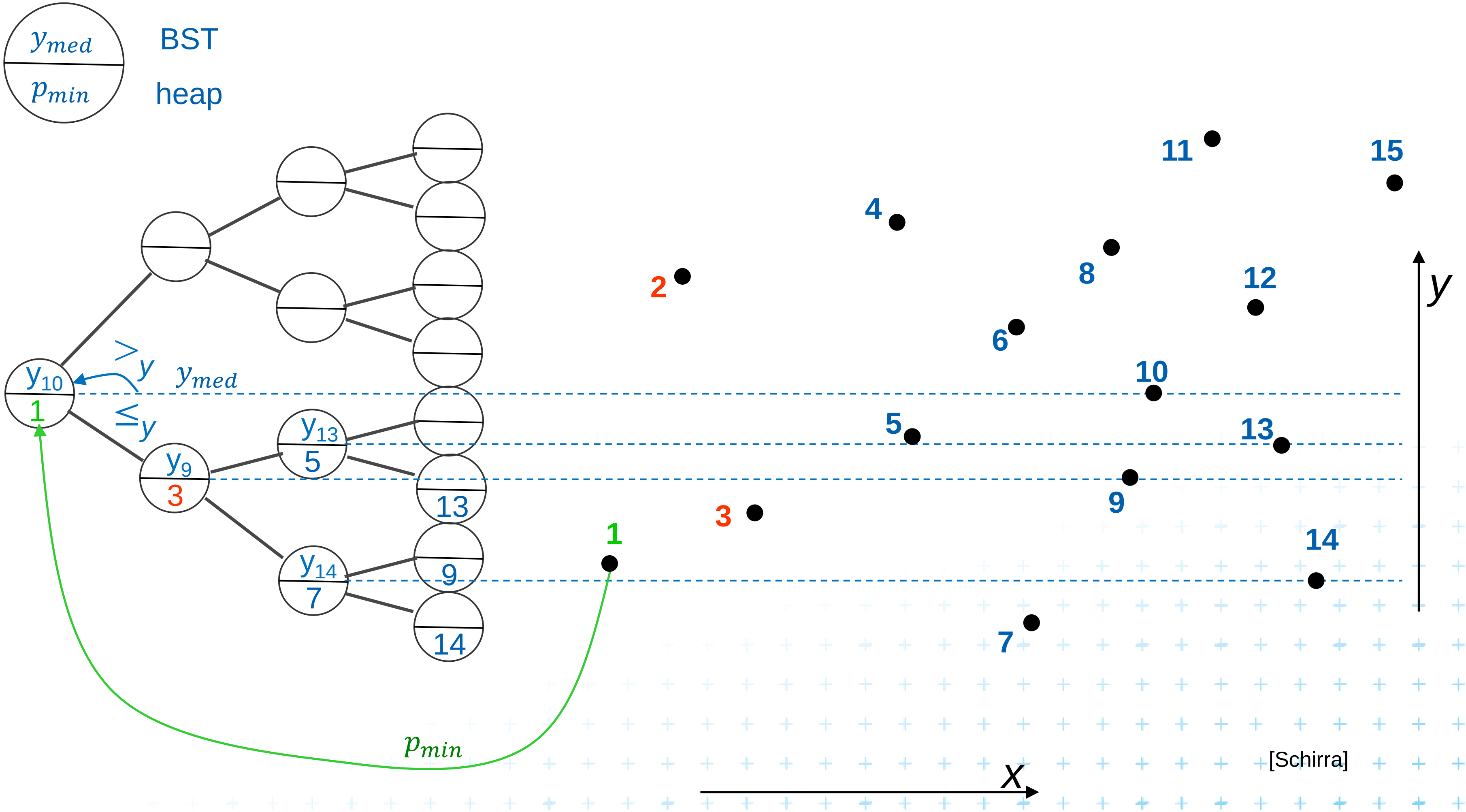
[Schirra]

Priority search tree construction example



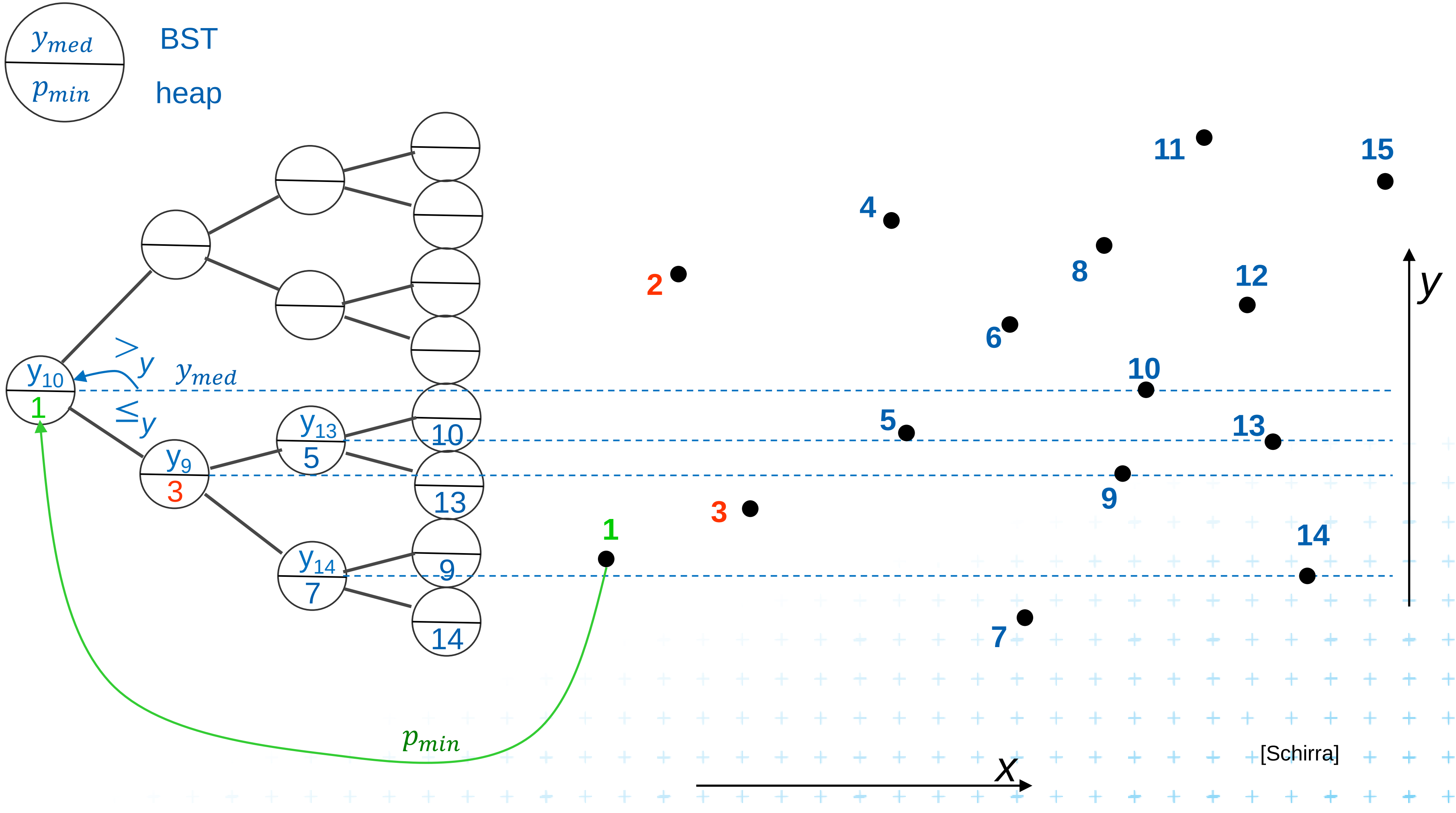
[Schirra]

Priority search tree construction example



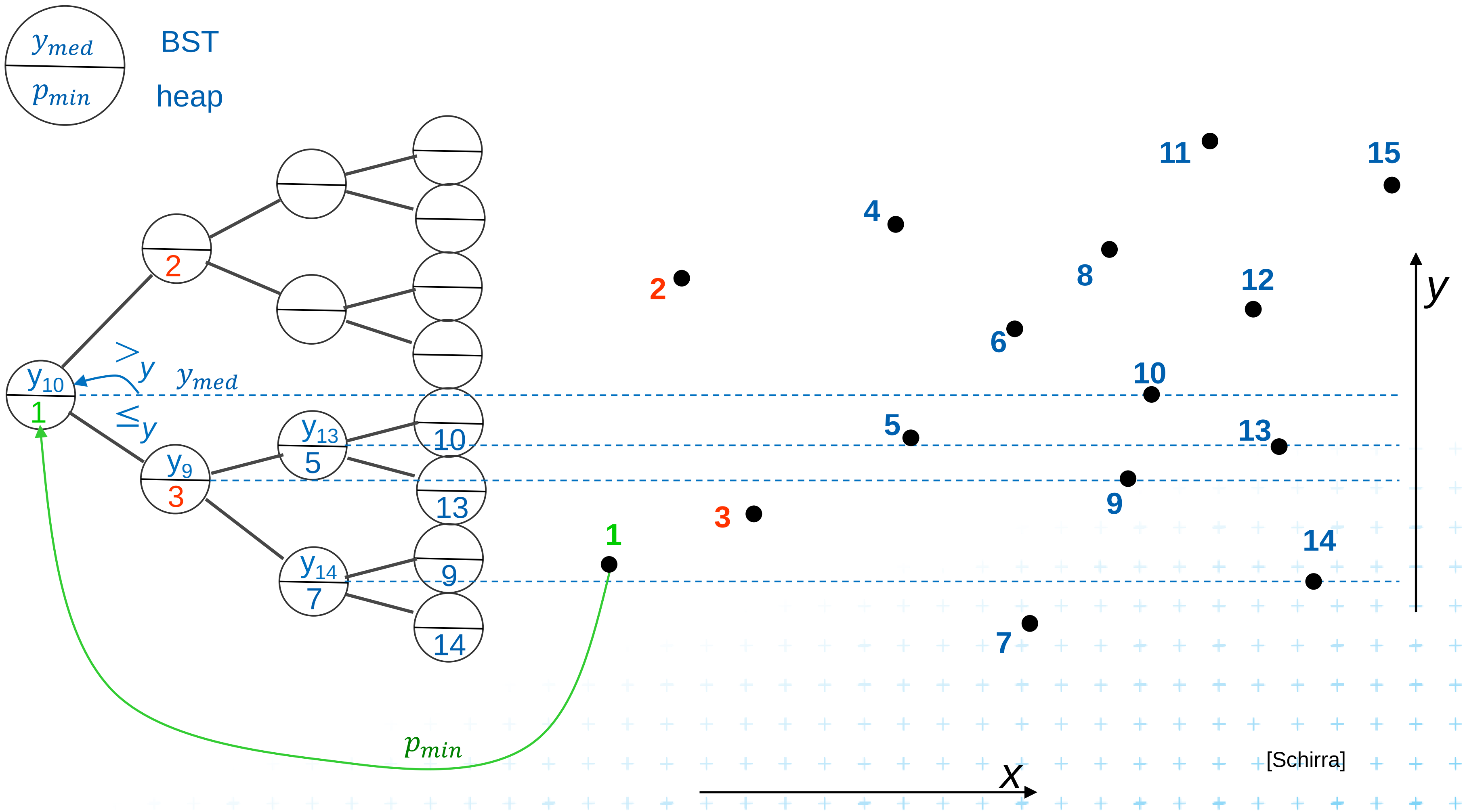
[Schirra]

Priority search tree construction example



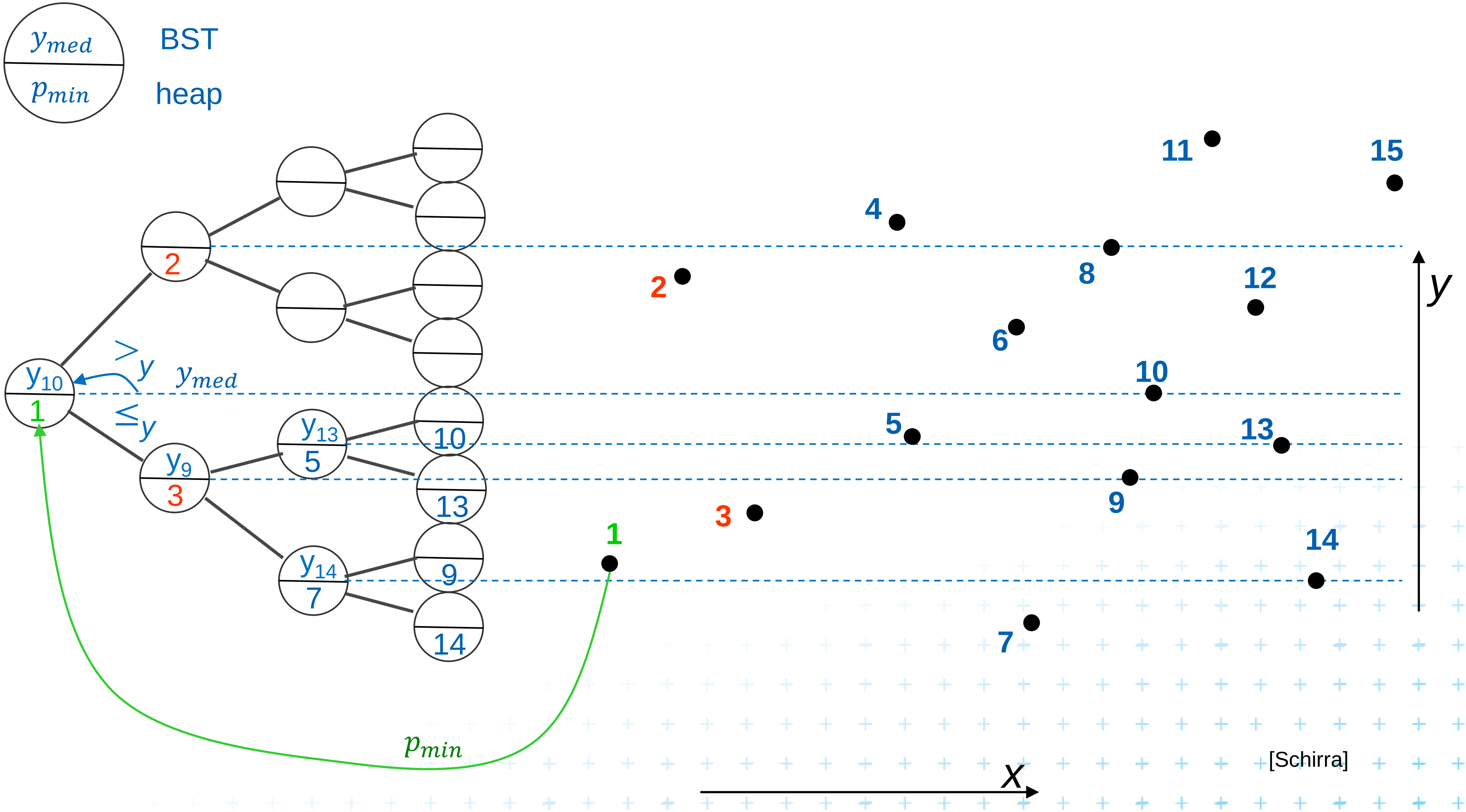
[Schirra]

Priority search tree construction example



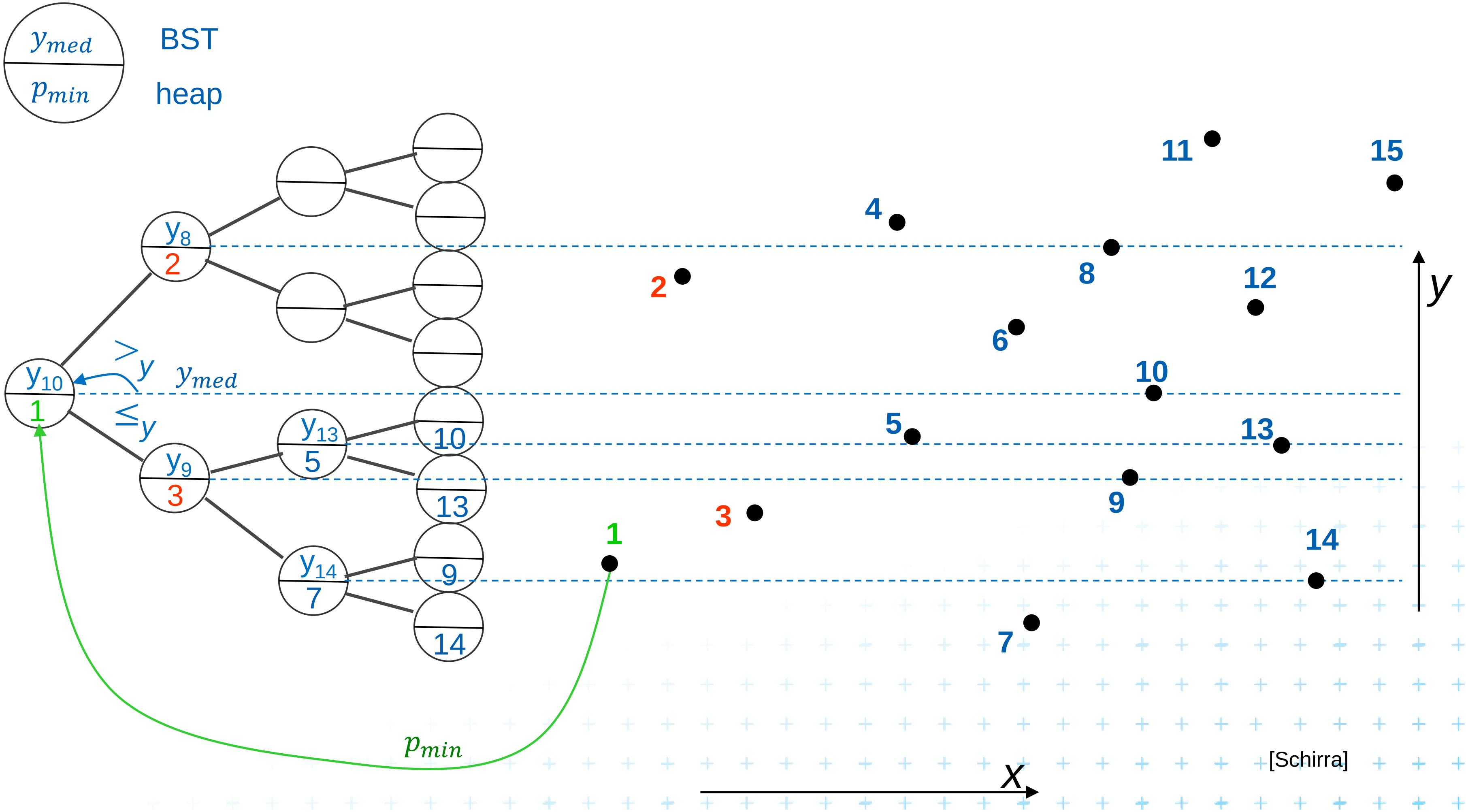
[Schirra]

Priority search tree construction example



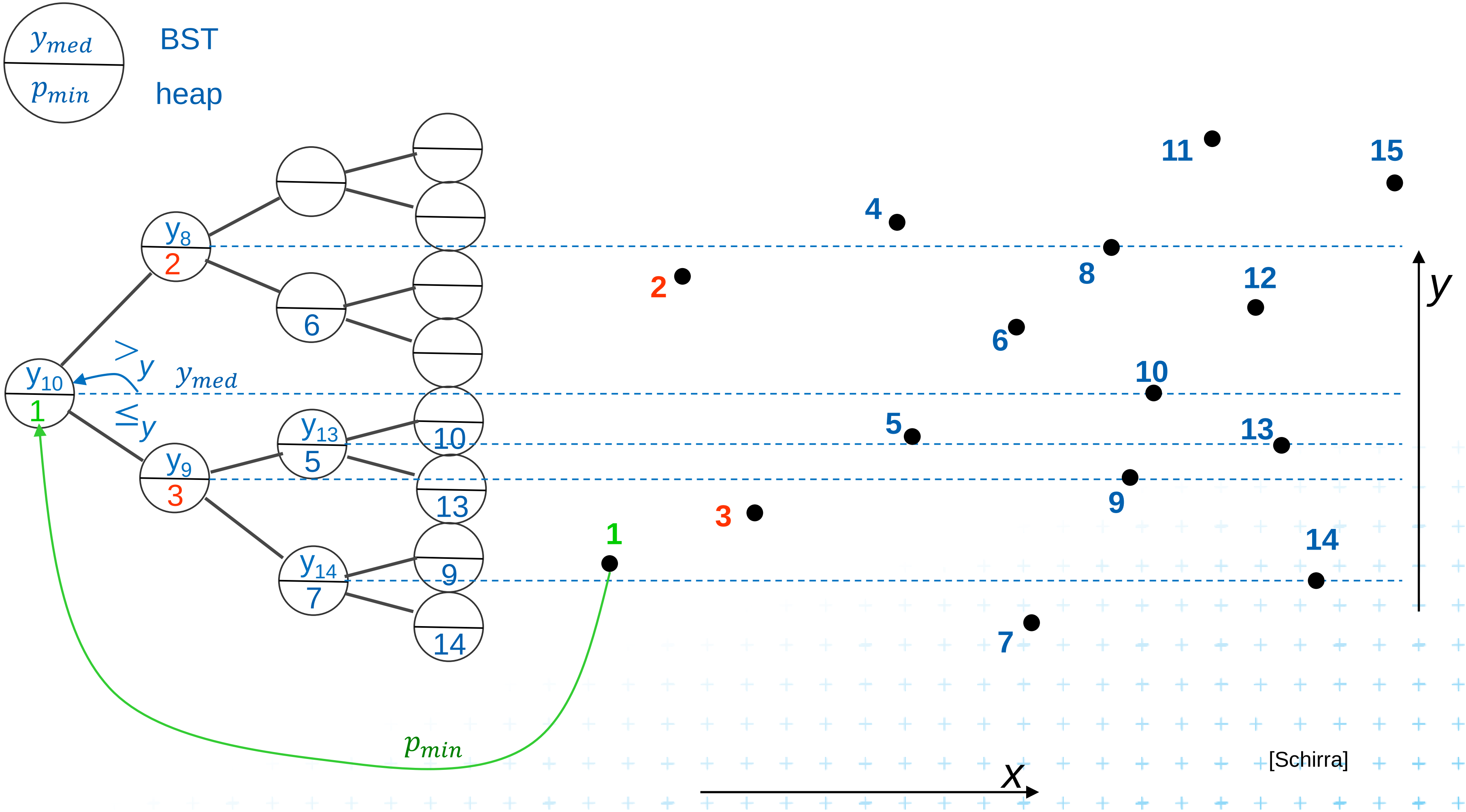
[Schirra]

Priority search tree construction example



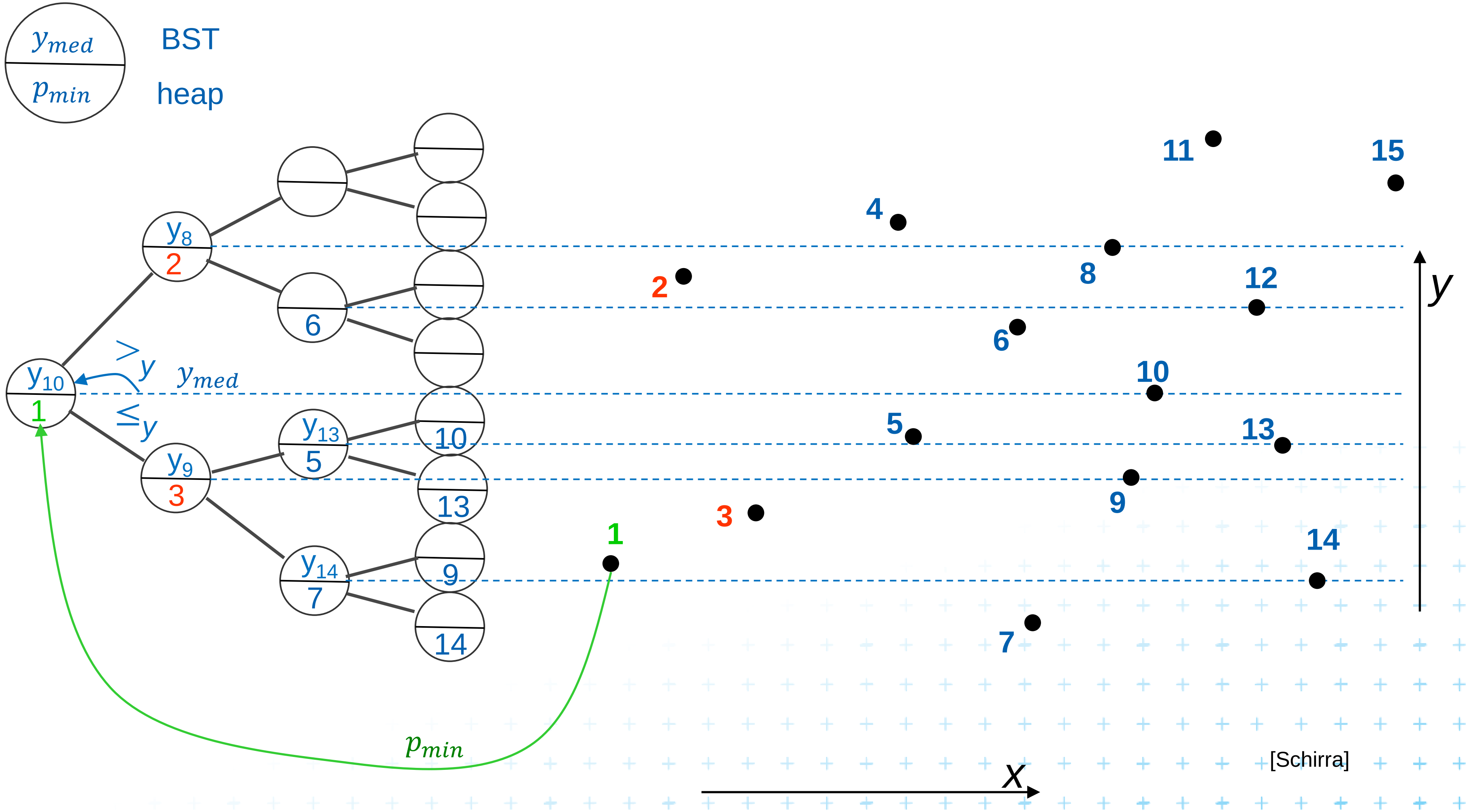
[Schirra]

Priority search tree construction example



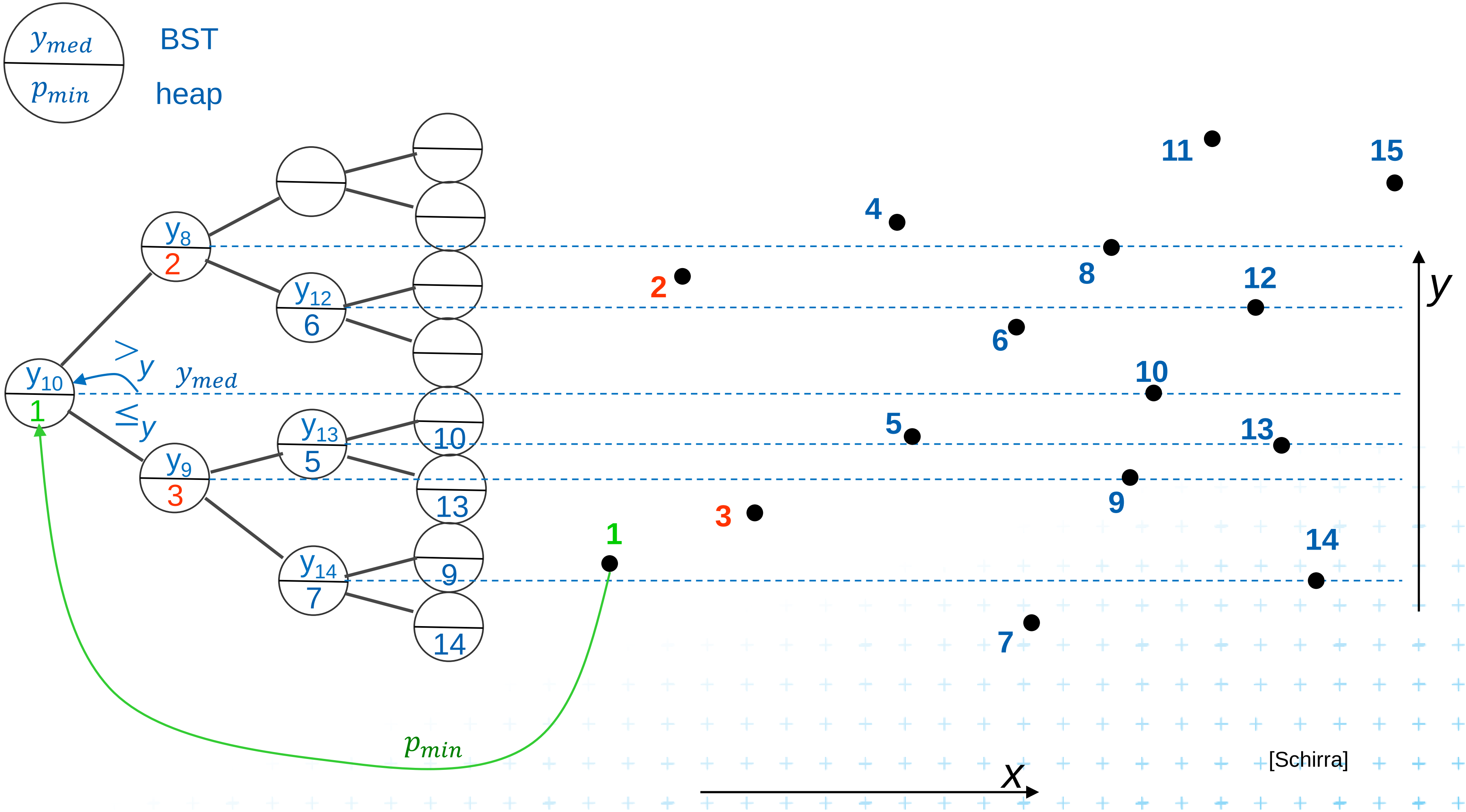
[Schirra]

Priority search tree construction example

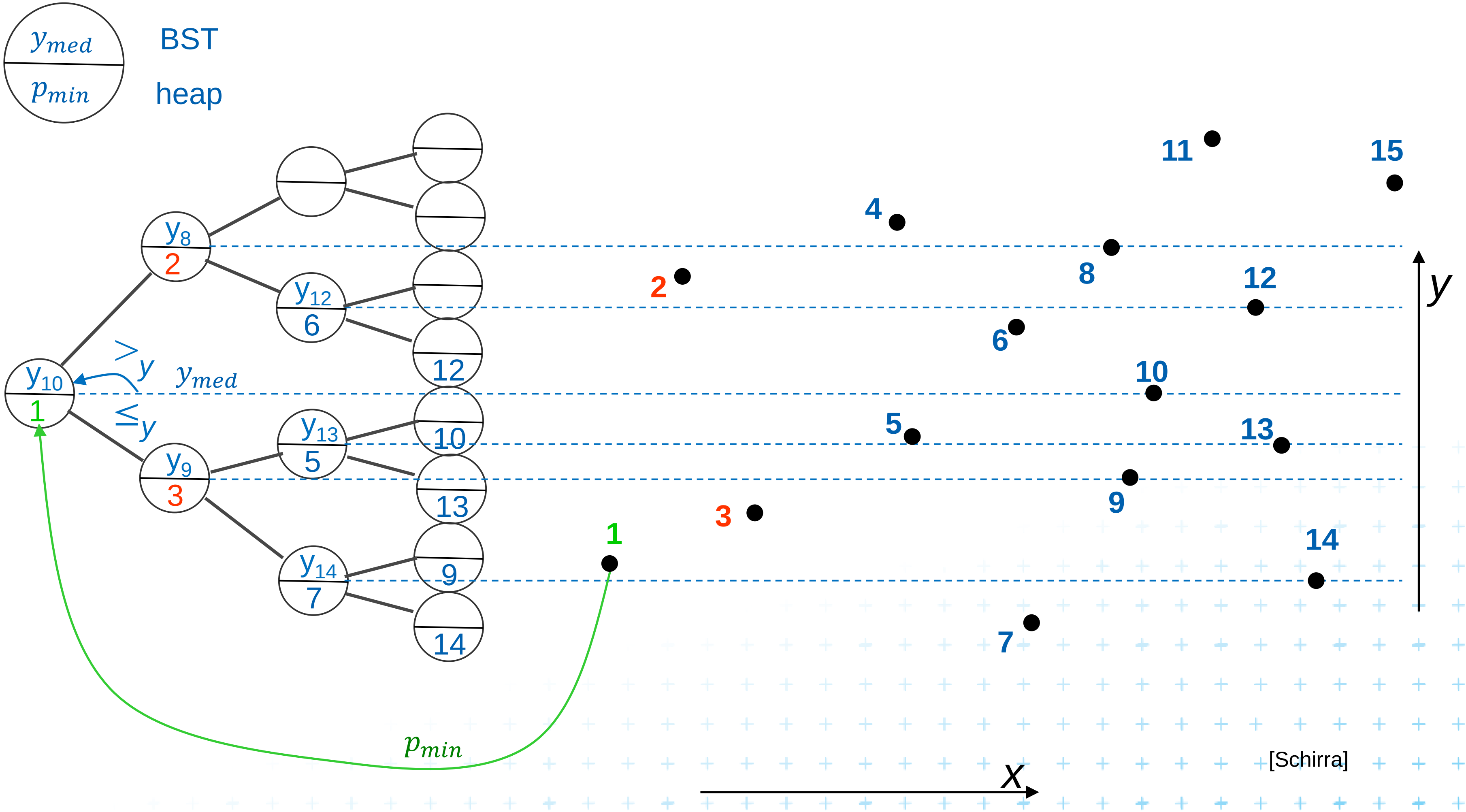


[Schirra]

Priority search tree construction example

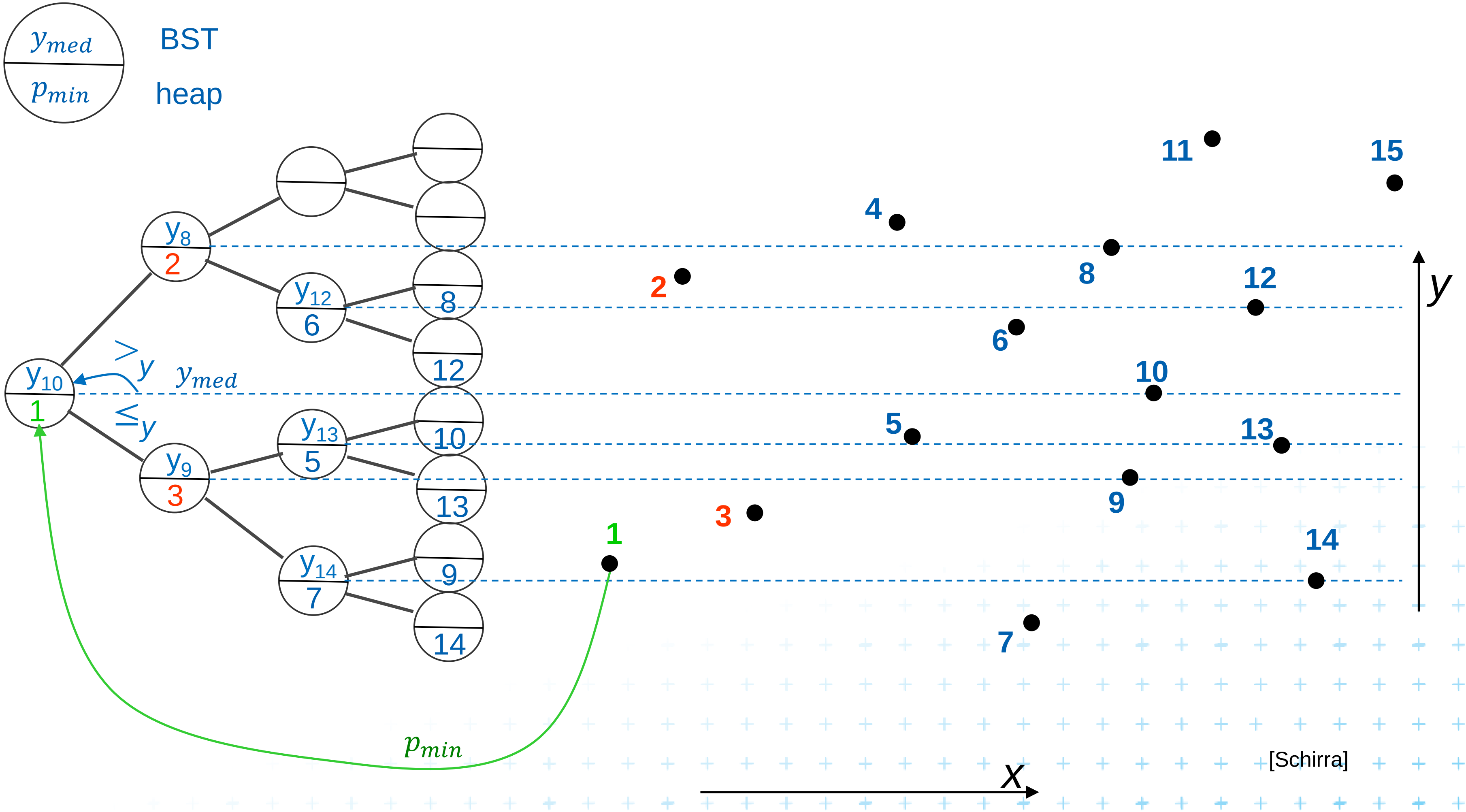


Priority search tree construction example



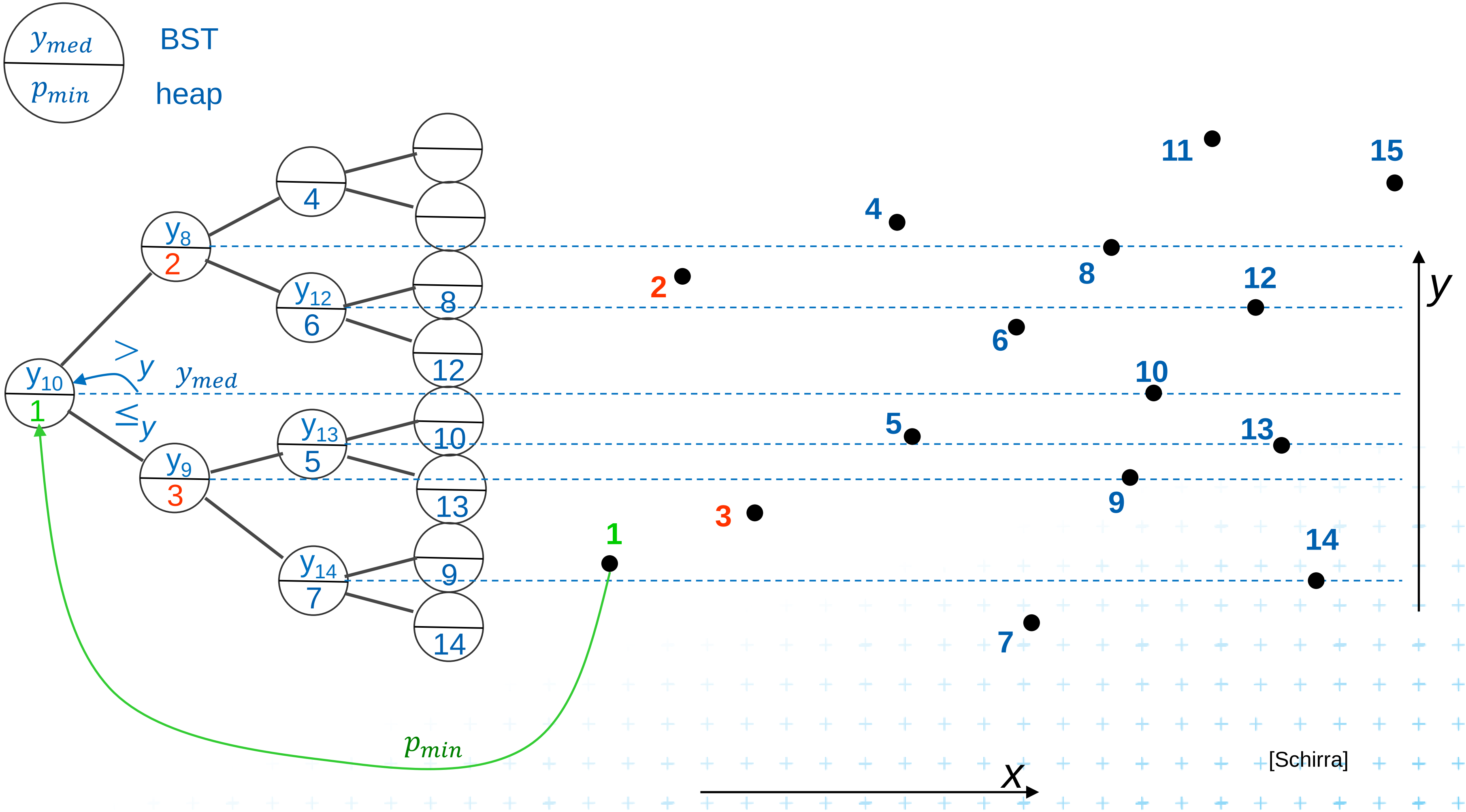
[Schirra]

Priority search tree construction example



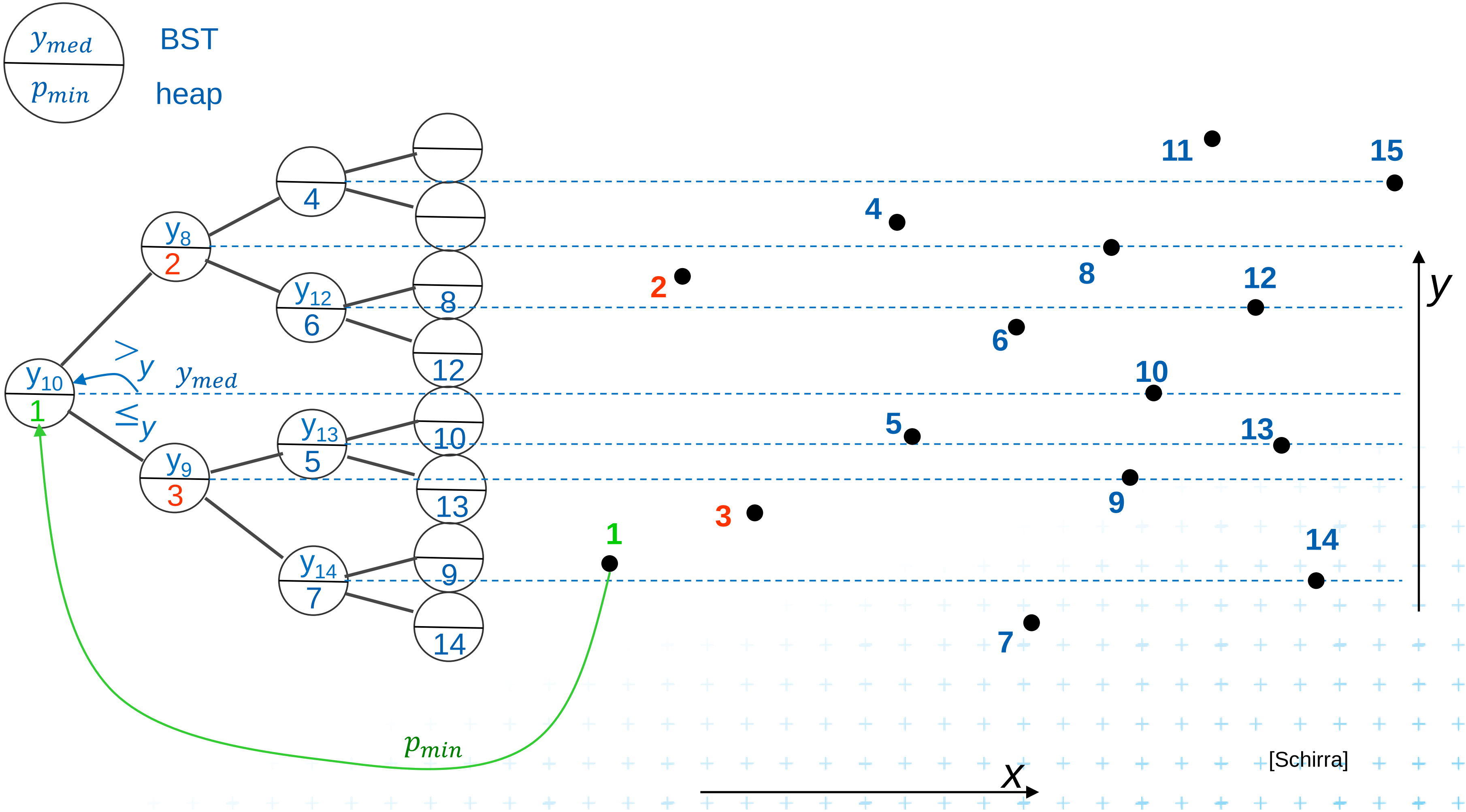
[Schirra]

Priority search tree construction example



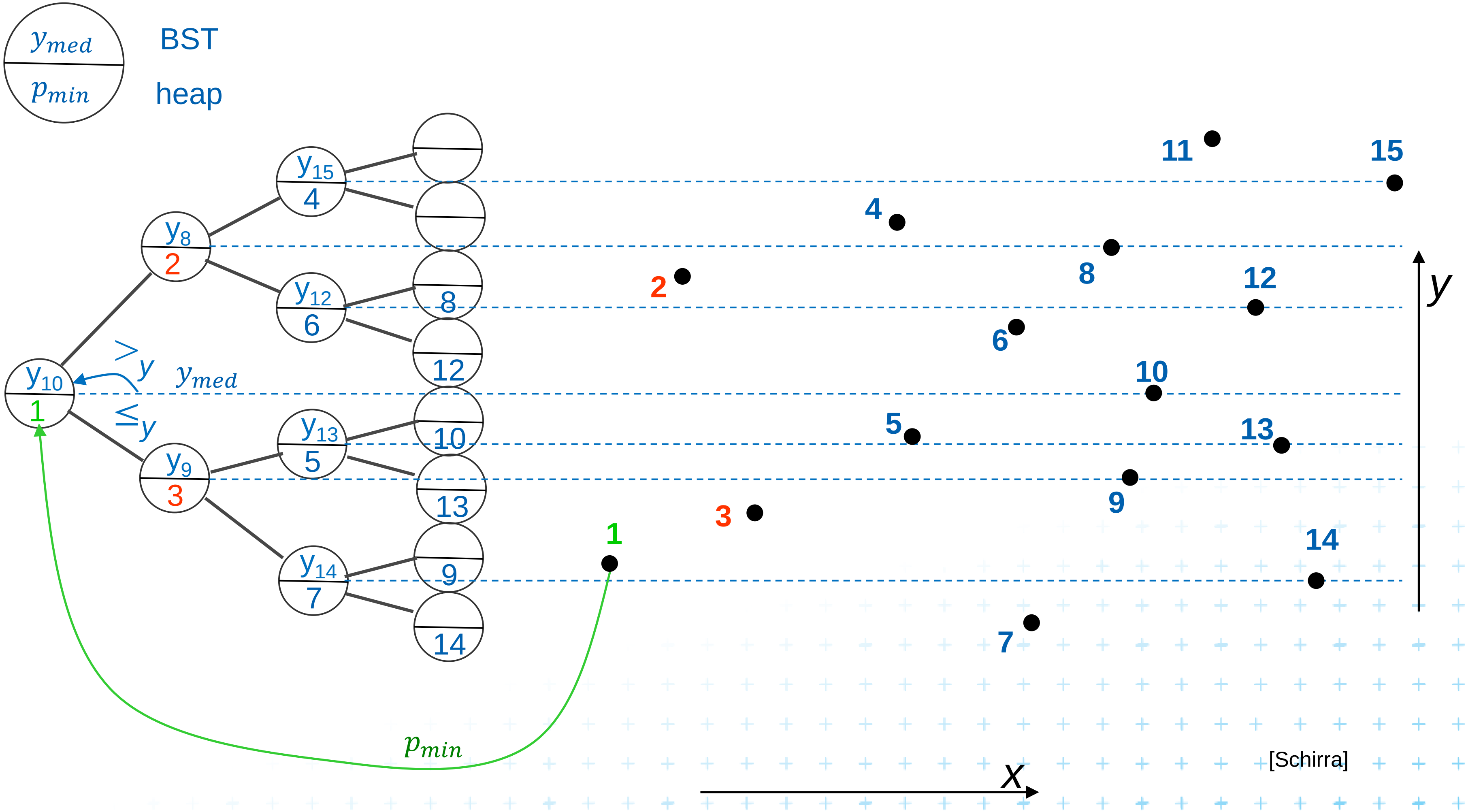
[Schirra]

Priority search tree construction example



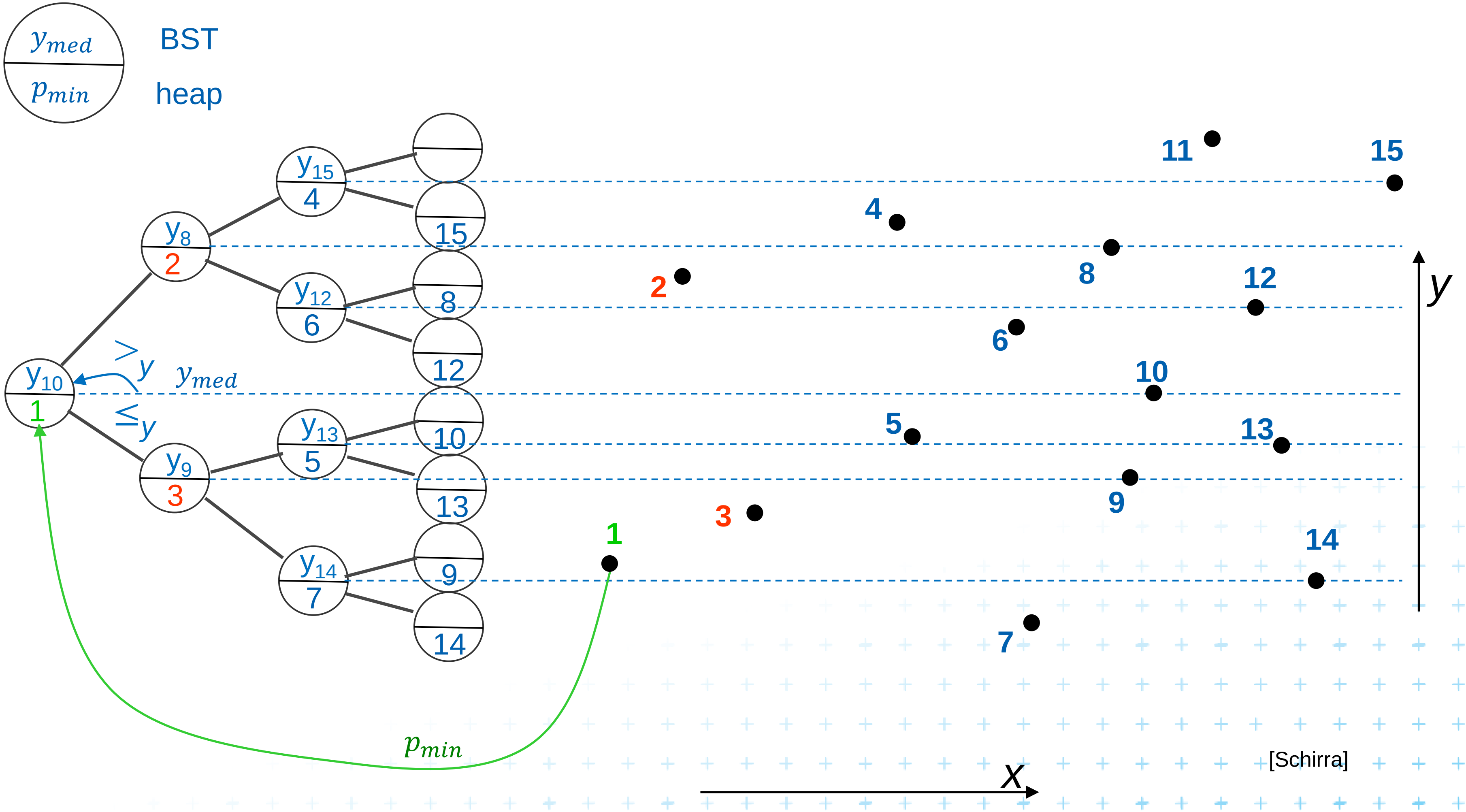
[Schirra]

Priority search tree construction example



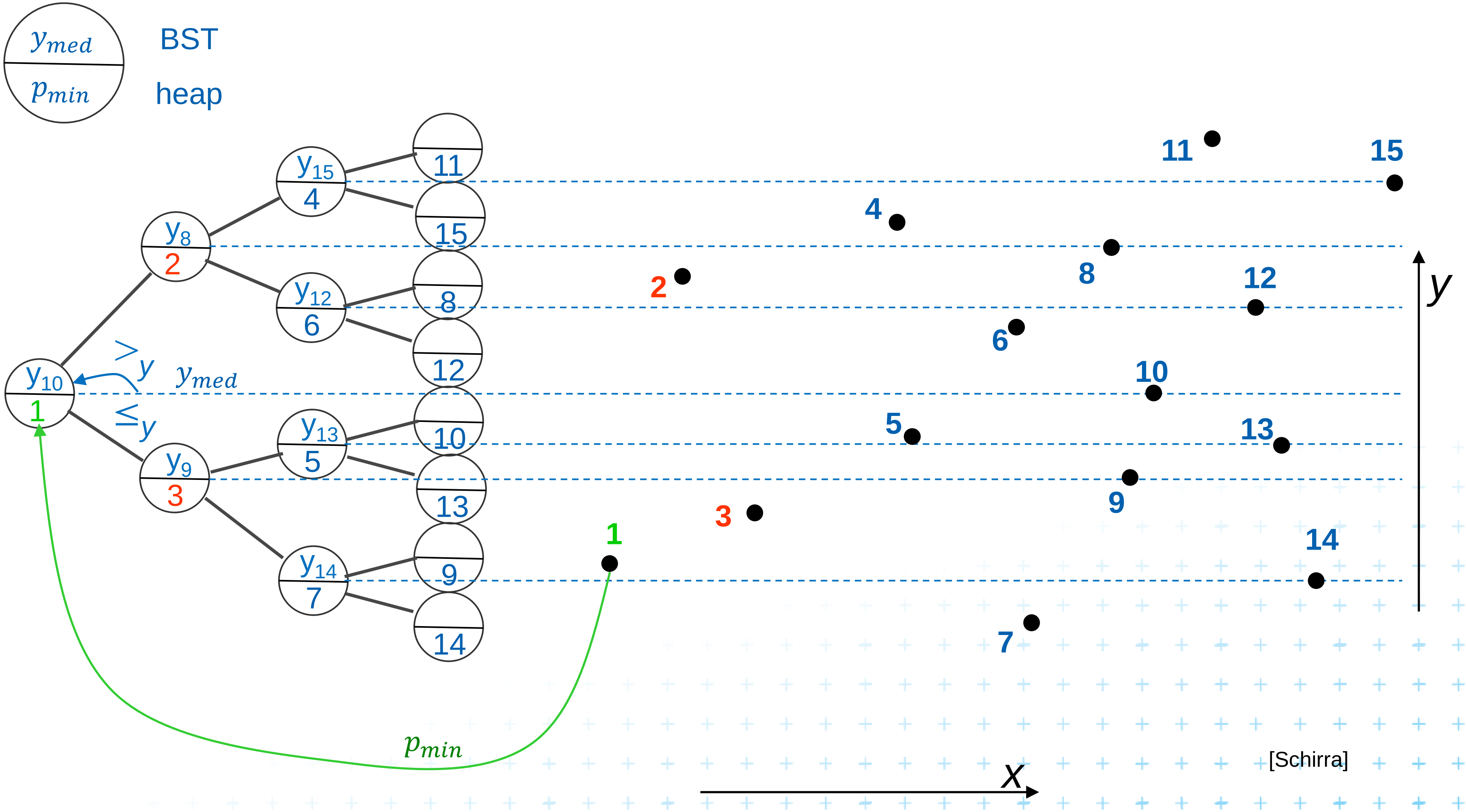
[Schirra]

Priority search tree construction example



[Schirra]

Priority search tree construction example



[Schirra]

Priority search tree construction

```
PrioritySearchTree( $P$ )
Input:   set $P$ of points in plane
Output:  priority search tree $T$
if $P = \emptyset$ then PST is an empty leaf
else
    $p_{min}$ = point with smallest $x$-coordinate in $P$      // heap on $x$ root
    $y_{med}$ = $y$-coord. median of points $P \setminus \{p_{min}\}$      // BST on $y$ root
    Split points P \ {$p_{min}$} into two subsets – according to $y_{med}$
        $P_{below} \coloneqq \{ p \in P \setminus \{p_{min}\} \ : p_y \leq y_{med} \}$
        $P_{above} \coloneqq \{ p \in P \setminus \{p_{min}\} \ : p_y > y_{med} \}$
    T = newTreeNode()
    T.p = $p_{min}$      // point [ $x, y$ ]
    T.y = $y_{med}$      // scalar
    T.left  = PrioritySearchTree( $P_{below}$ )
    T.rigft = PrioritySearchTree( $P_{above}$ )
```

… Notation on the next slide:

… $p(v)$, v = tree node

… $y(v)$

… $l(v)$

… $r(v)$

$O(n \log n)$, but $O(n)$ if presorted on y-coordinate and bottom up

Query Priority Search Tree

QueryPrioritySearchTree(T**,** $(-\infty : q_x] \times [q_y : q'_y\,]$**)**

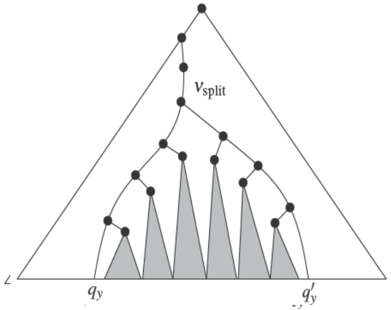
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
1. Search with $q_y$ and $q'_y$ in $T$      // BST on $y$-coordinate – select $y$ range
   Let $v_{split}$ be the node where the two search paths split (split node)

2. for each node $v$ on the search path of $q_y$ or $q'_y$   // points along the paths
3.     if $p(v) \in (-\infty : q_x] \times [q_y : q'_y\,]$ then Report $p(v)$   // starting in tree root

4. for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
5.     if the search path goes left at $v$
6.         ReportInSubtree( $r(v)$, $q_x$ )   // report right subtree
7. for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
8.     if the search path goes right at $v$
9.         ReportInSubtree( $l(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y : q'_y\,]$**)**

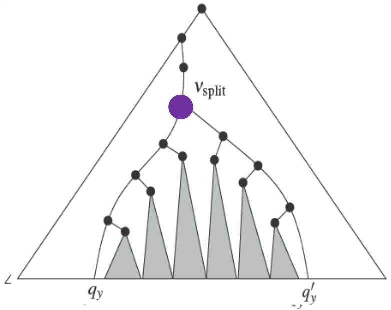
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$        // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ ● be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$    // points along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y : q'_y\,]$ then Report $p(v)$   // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $r(v)$, $q_x$ )   // report right subtree
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $l(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T**,** $(-\infty : q_x] \times [q_y : q'_y]$**)**

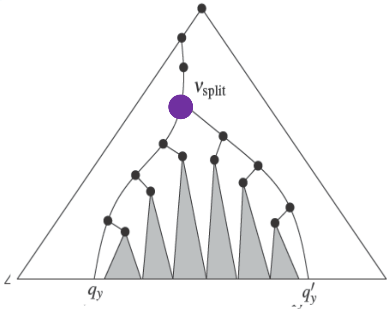
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$      // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ ● be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$   // points • along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y : q'_y]$ then Report $p(v)$   // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $r(v)$, $q_x$ )   // report right subtree
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $l(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y : q'_y\,]$**)**

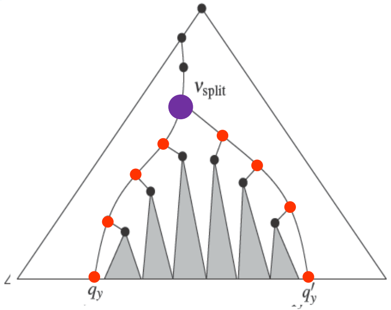
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
1.  Search with $q_y$ and $q'_y$ in $T$        // BST on $y$-coordinate – select $y$ range
    Let $v_{split}$ be the node where the two search paths split (split node)

2.  for each node $v$ on the search path of $q_y$ or $q'_y$   // points • along the paths
3.      if $p(v) \in (-\infty : q_x] \times [q_y : q'_y\,]$ then Report $p(v)$   // starting in tree root

4.  for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
5.      if the search path goes left at $v$
6.          ReportInSubtree( $r(v)$, $q_x$)   // report right subtree
7.  for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
8.      if the search path goes right at $v$
9.          ReportInSubtree( $l(v)$, $q_x$)   // rep. left subtree
```



[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T**,** $(-\infty : q_x] \times [q_y : q'_y\,]$**)**

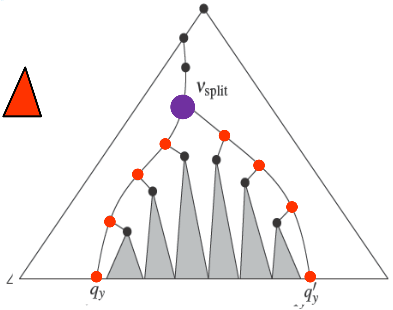
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$        // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ ● be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$   // points • along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y : q'_y\,]$ then Report $p(v)$   // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$  // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $r(v)$, $q_x$ )   // report right subtree ▲
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $l(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y : q'_y\,]$**)**

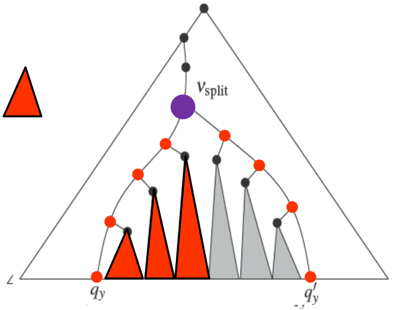
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$        // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ ● be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$    // points • along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y : q'_y\,]$ then Report $p(v)$   // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $r(v)$, $q_x$ )   // report right subtree ▲
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $l(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y : q'_y\,]$)

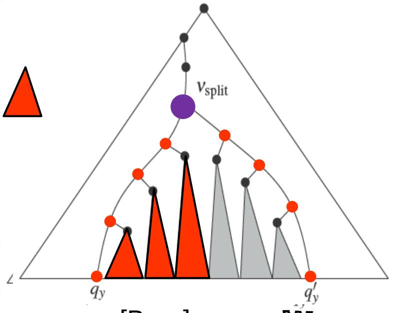
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
1.  Search with $q_y$ and $q'_y$ in $T$        // BST on $y$-coordinate – select $y$ range
    Let $v_{split}$ ● be the node where the two search paths split (split node)

2.  for each node $v$ on the search path of $q_y$ or $q'_y$   // points • along the paths
3.      if $p(v) \in (-\infty : q_x] \times [q_y : q'_y\,]$ then Report $p(v)$   // starting in tree root

4.  for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
5.      if the search path goes left at $v$
6.          ReportInSubtree( $r(v)$, $q_x$)   // report right subtree ▲
7.  for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
8.      if the search path goes right at $v$
9.          ReportInSubtree( $l(v)$, $q_x$)   // rep. left subtree ▲
```



[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T**,** $(-\infty : q_x] \times [q_y : q'_y\,]$**)**

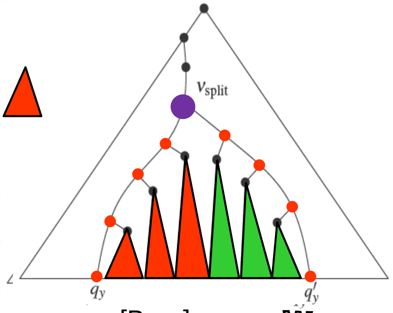
Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$        // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ ● be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$   // points ● along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y : q'_y\,]$ then Report $p(v)$  // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $r(v)$, $q_x$)   // report right subtree ▲
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $l(v)$, $q_x$)   // rep. left subtree ▲
```



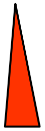
[Berg]

Reporting of subtrees between the y-paths

ReportInSubtree(*v*, *q_x*)

Input: The root v of a subtree of a priority search tree and a value q_x.
Output: All points p in the subtree with x-coordinate at most q_x.

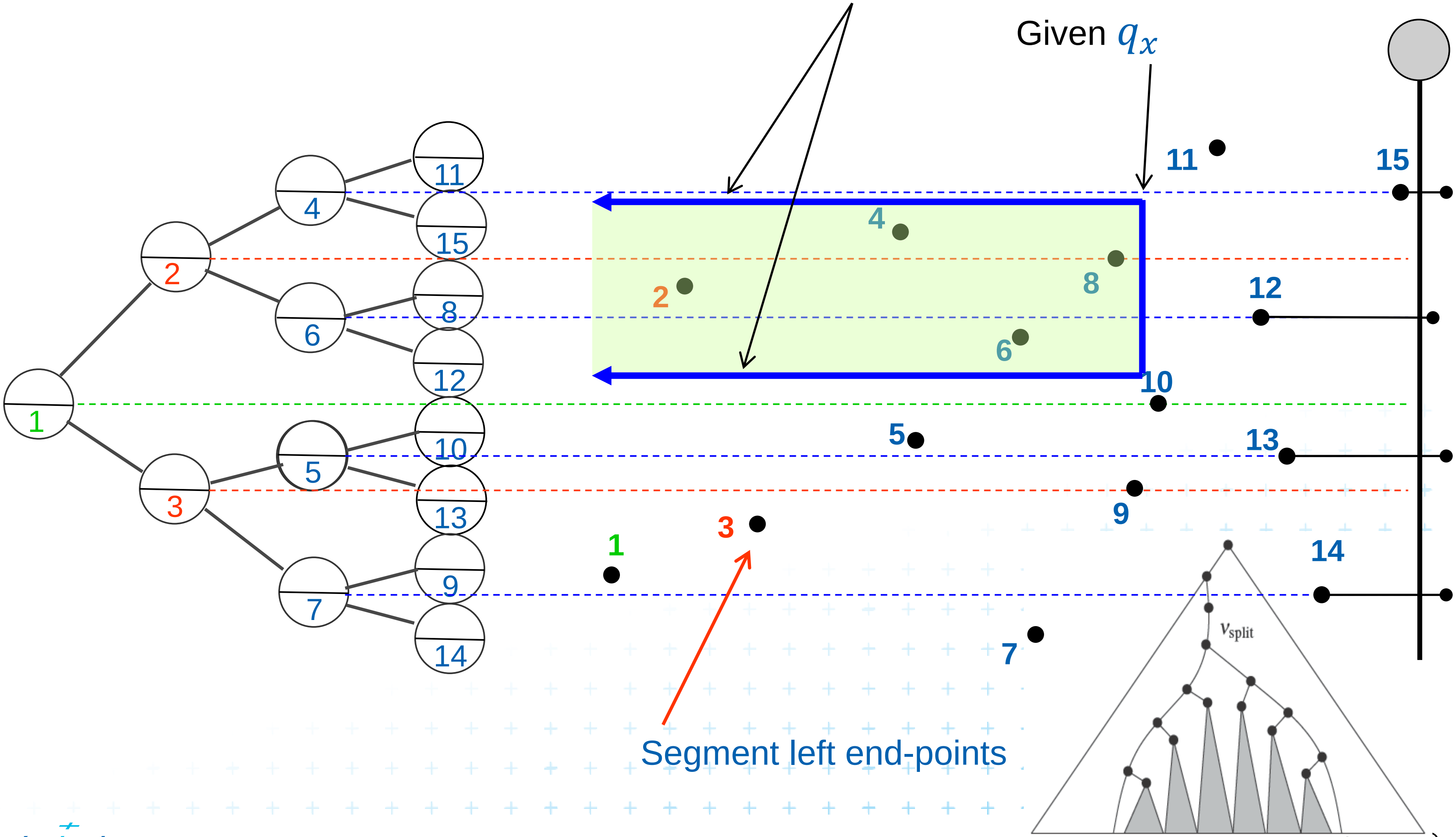
```
if $x(p(v)) \le q_x$        // $x \in (-\infty : q_x]$   -- heap condition
    Report point $p(v)$.
if $v$ is not a leaf
    ReportInSubtree( $l(v), q_x$)
    ReportInSubtree( $r(v), q_x$)
```



Search according to x in the heap

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

-

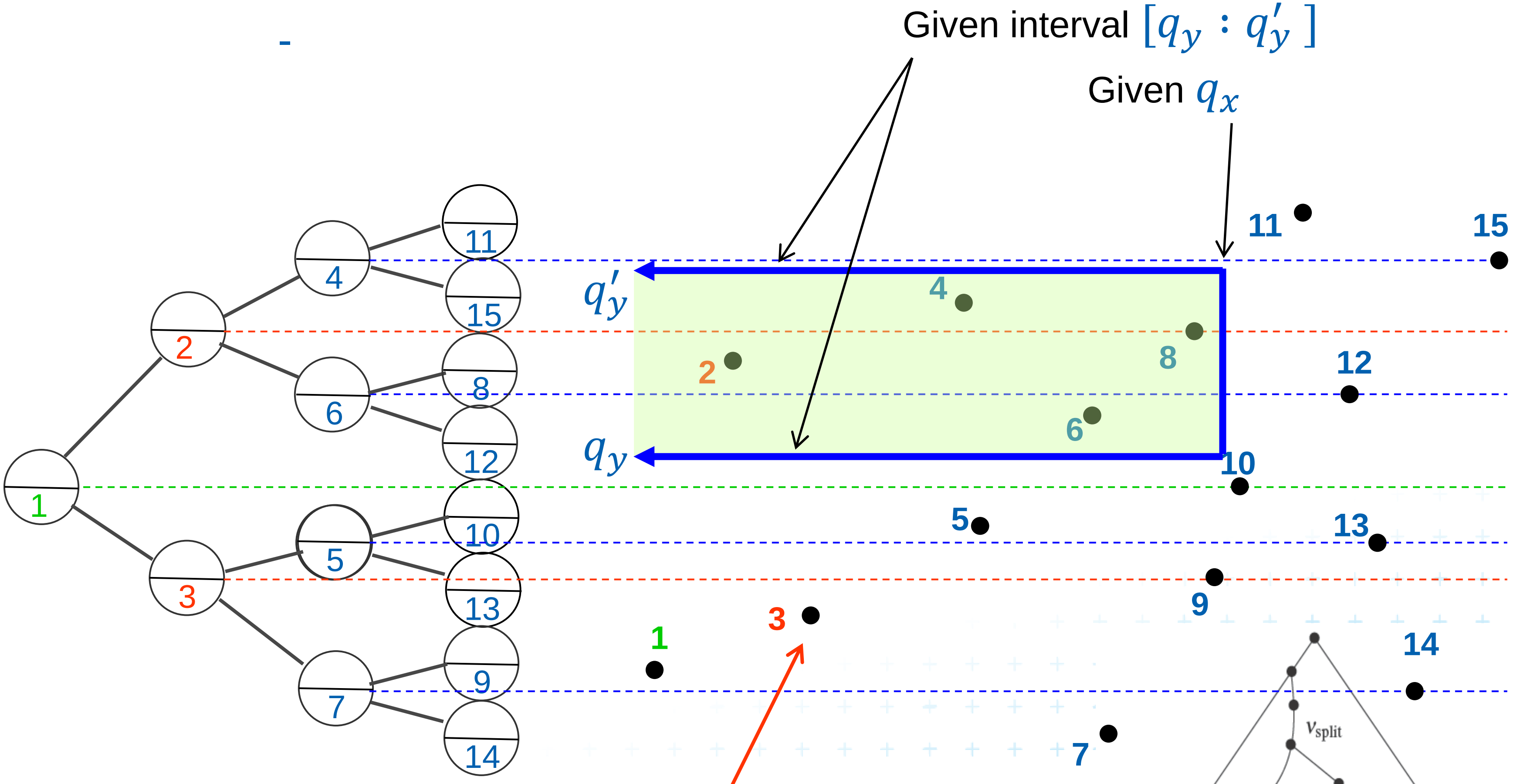


Based on [Schirra]

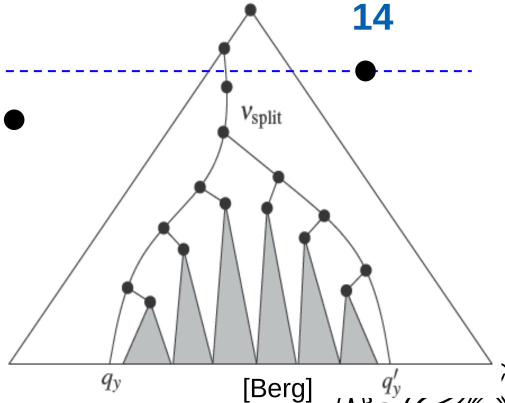
[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

-



Segment left end-points

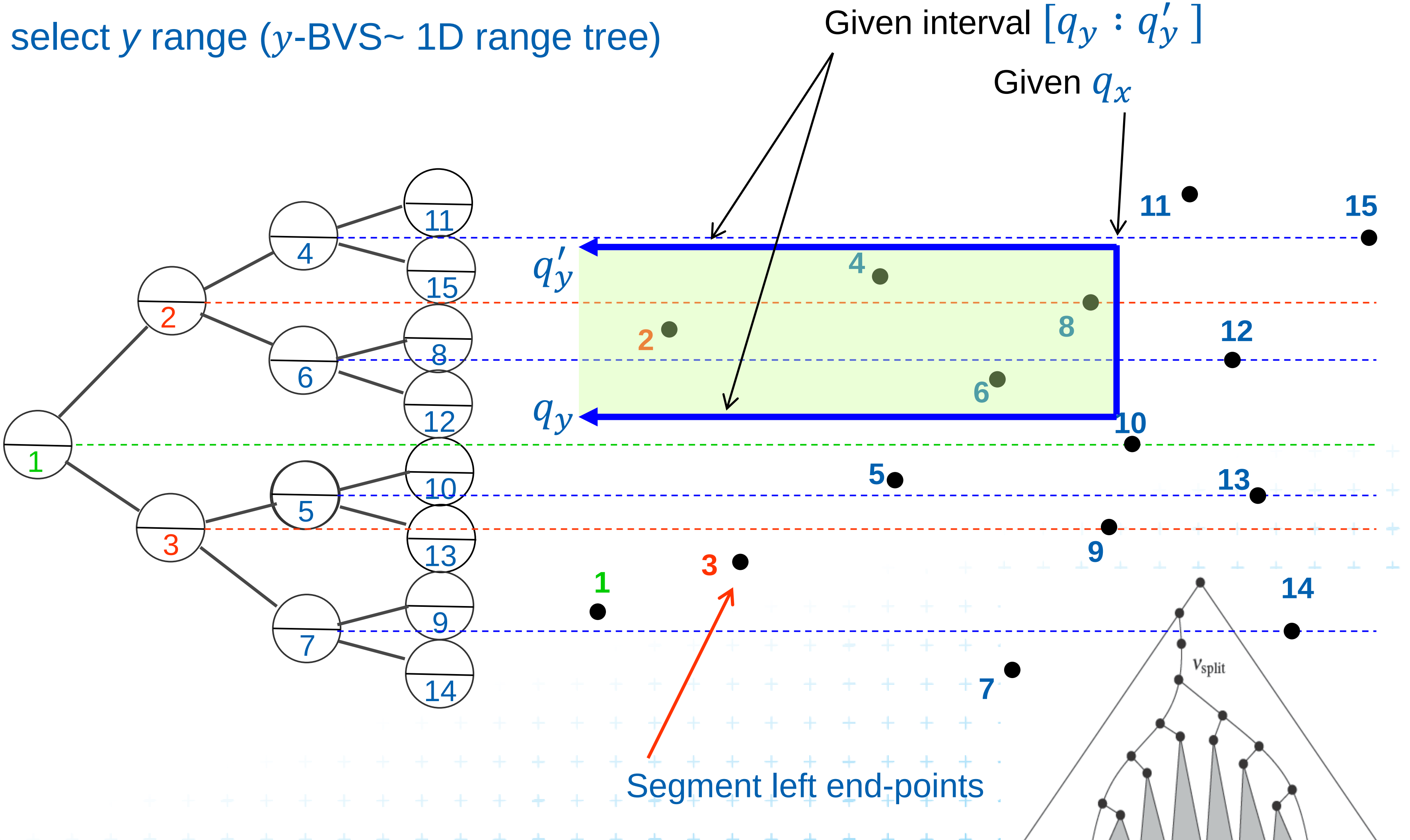


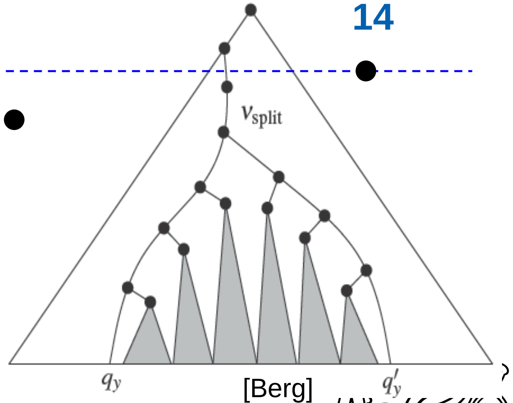
[Berg]

Based on [Schirra]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (*y*-BVS~ 1D range tree)



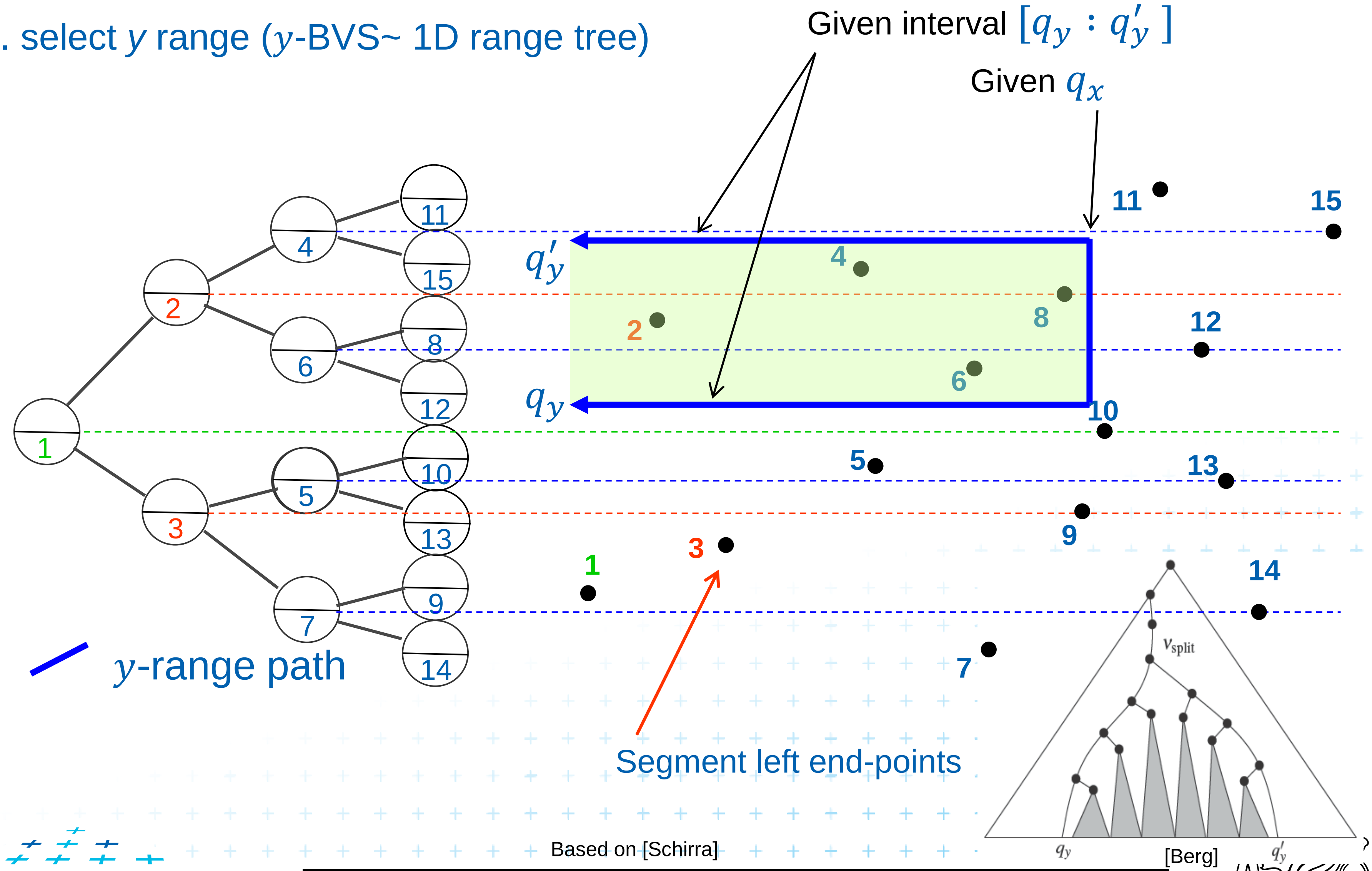


[Berg]

Based on [Schirra]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

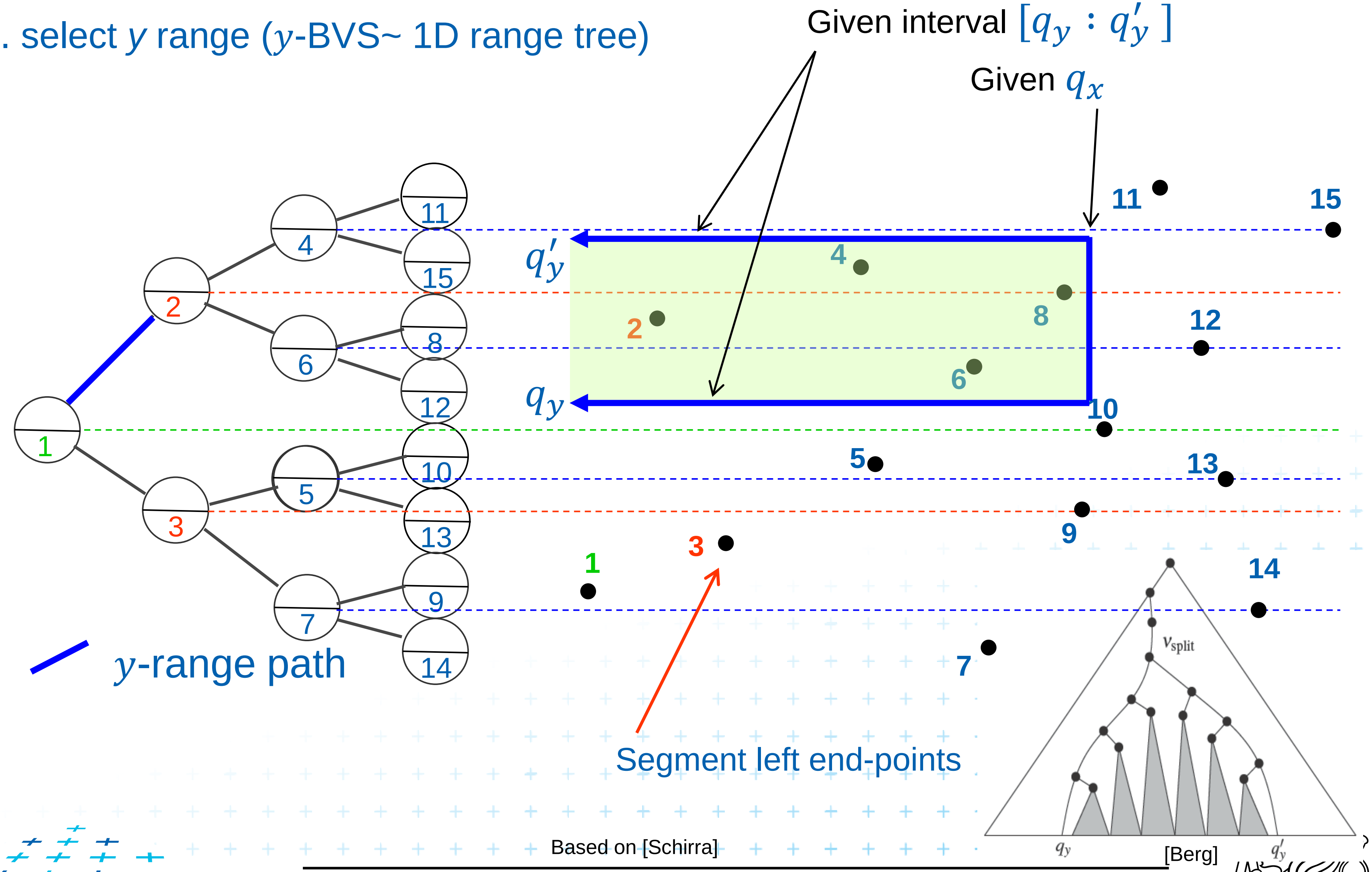
1. select *y* range (y-BVS~ 1D range tree)



Based on [Schirra]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y\,]$

1. select *y* range (y-BVS~ 1D range tree)



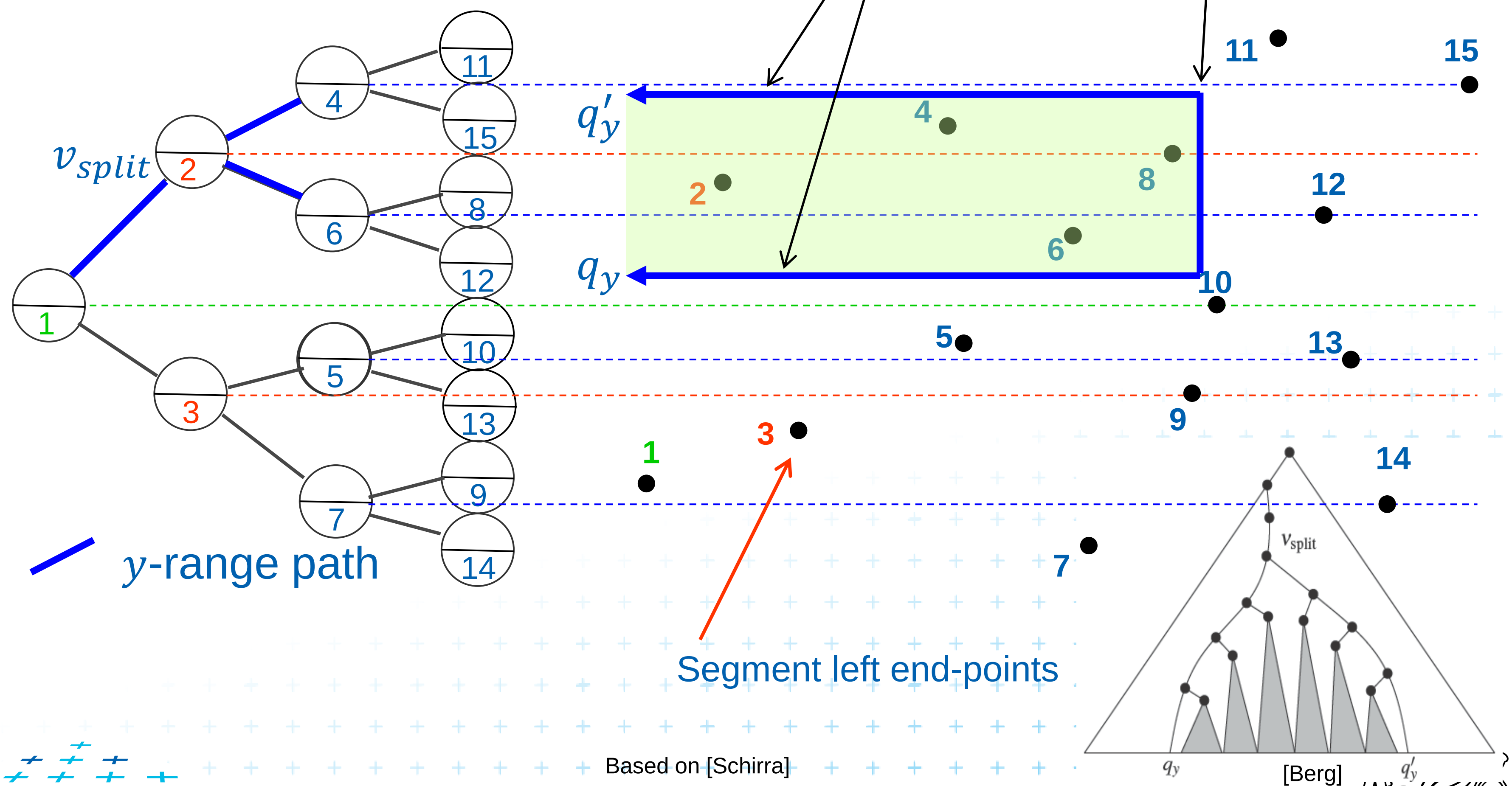
Based on [Schirra]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (*y*-BVS~ 1D range tree)

Given interval $[q_y : q'_y]$

Given q_x



y-range path

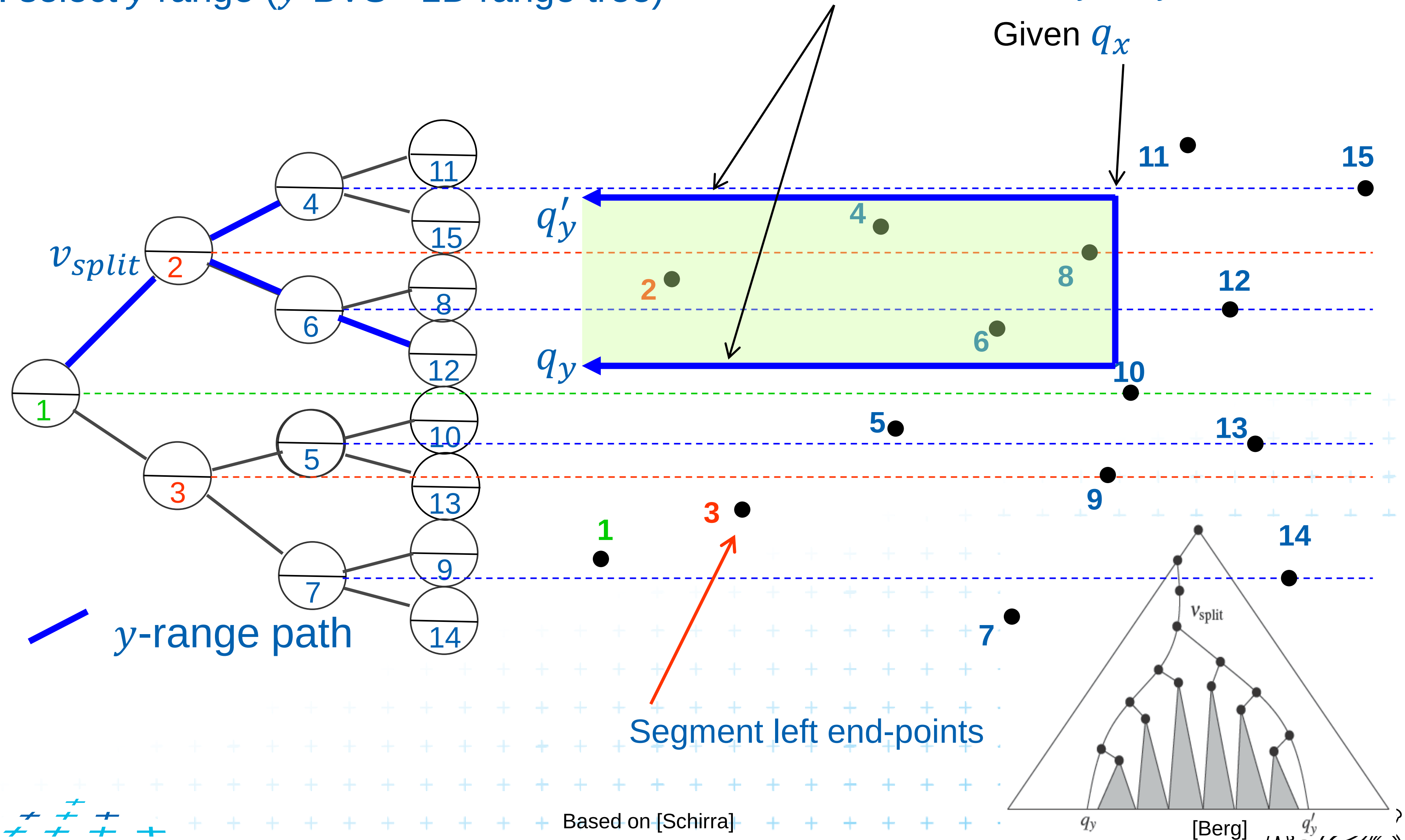
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (*y*-BVS~ 1D range tree)



Given interval $[q_y : q'_y]$

Given q_x

y-range path

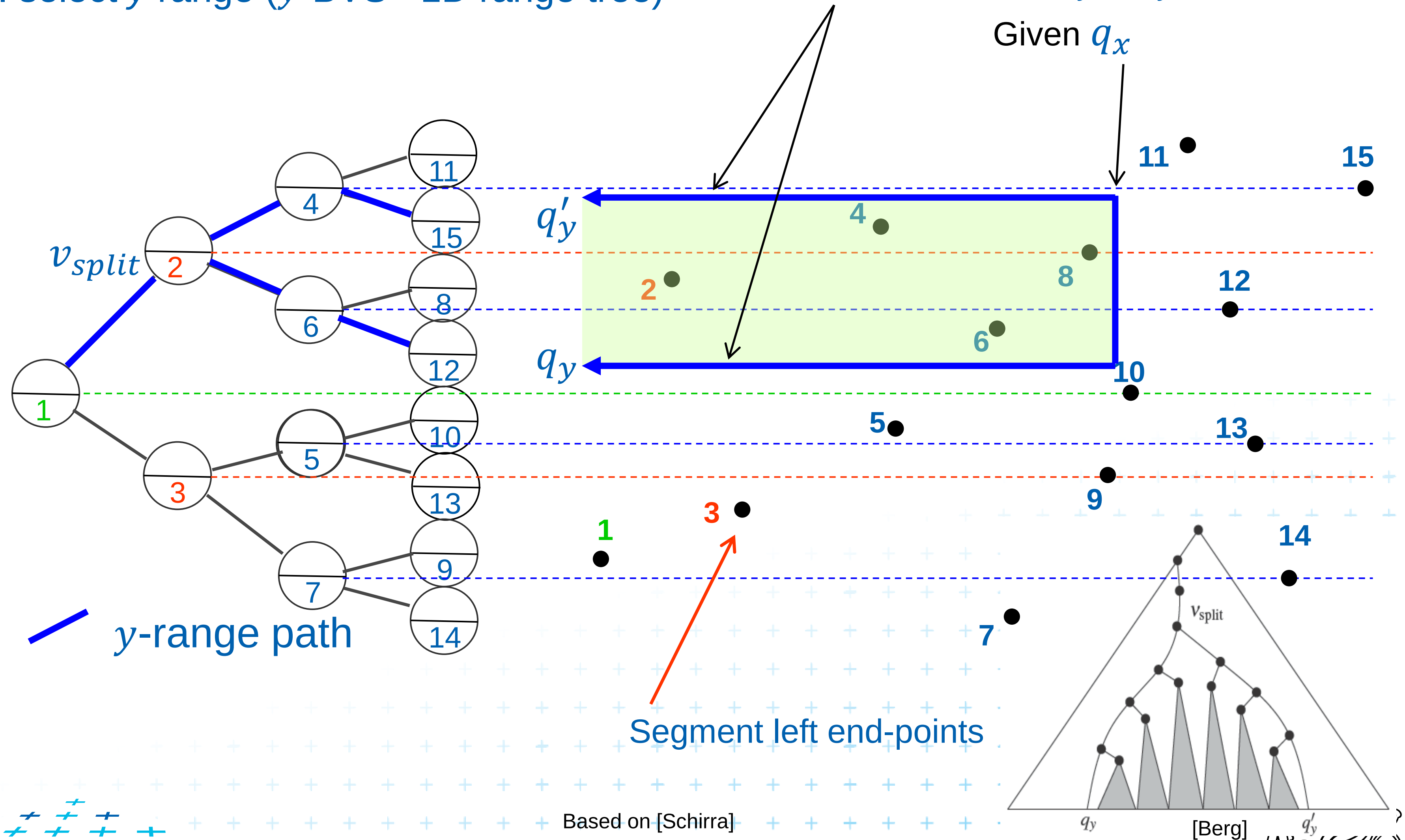
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (y-BVS~ 1D range tree)



Given interval $[q_y : q'_y]$

Given q_x

y-range path

Segment left end-points

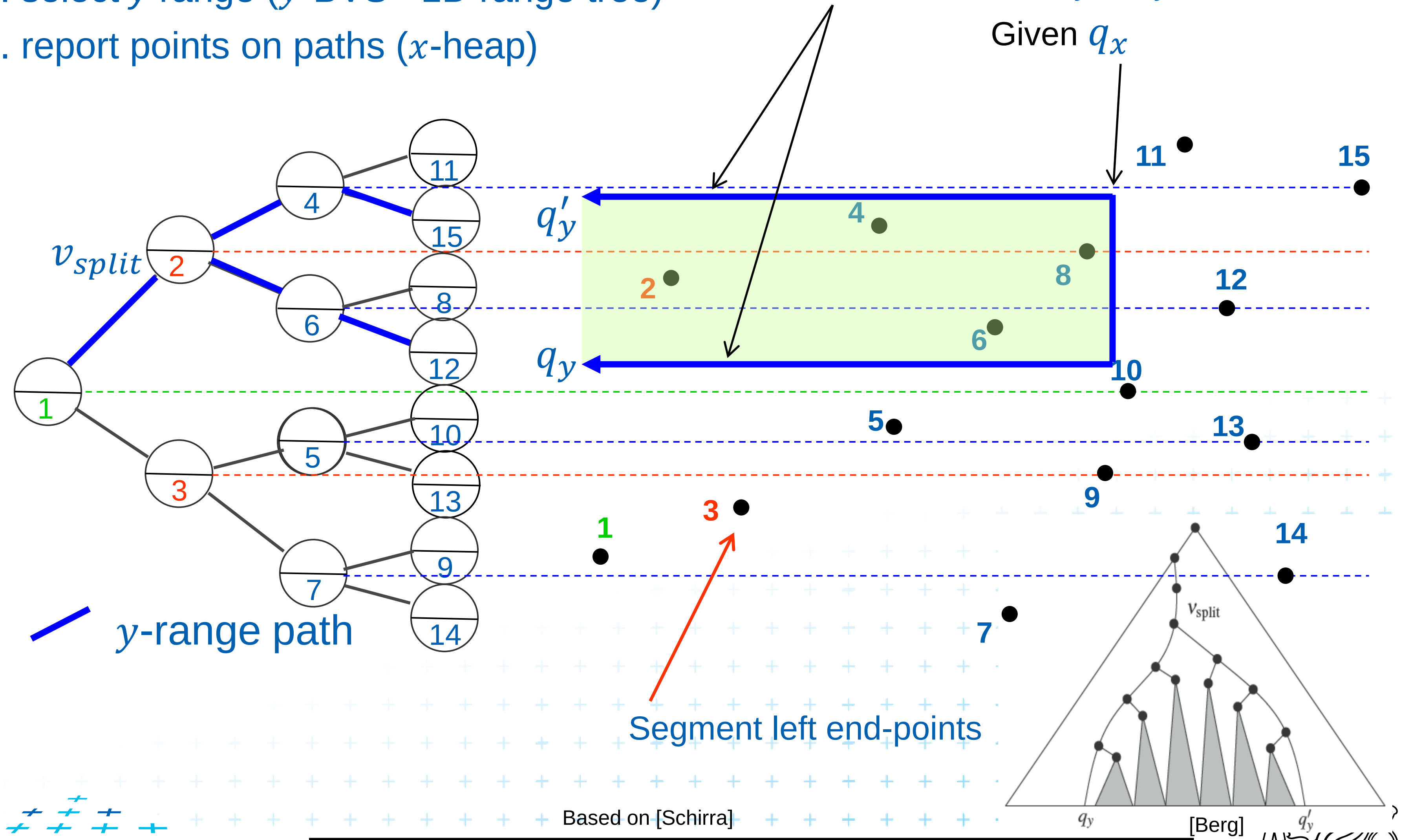
Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (y-BVS~ 1D range tree)

2. report points on paths (x-heap)



Given interval $[q_y : q'_y]$

Given q_x

y-range path

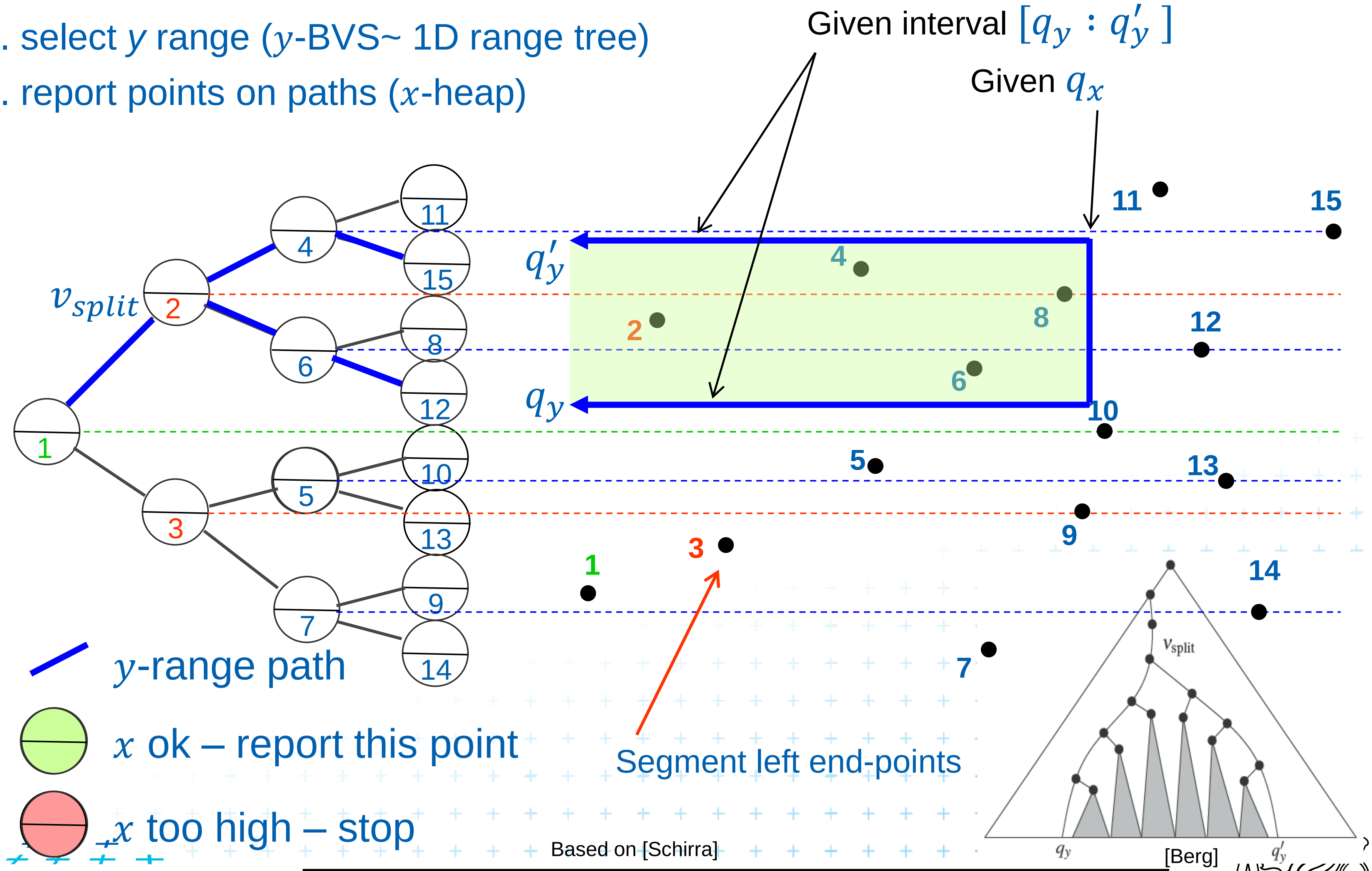
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y\,]$

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)


Given interval $[q_y : q'_y\,]$

Given q_x

Segment left end-points

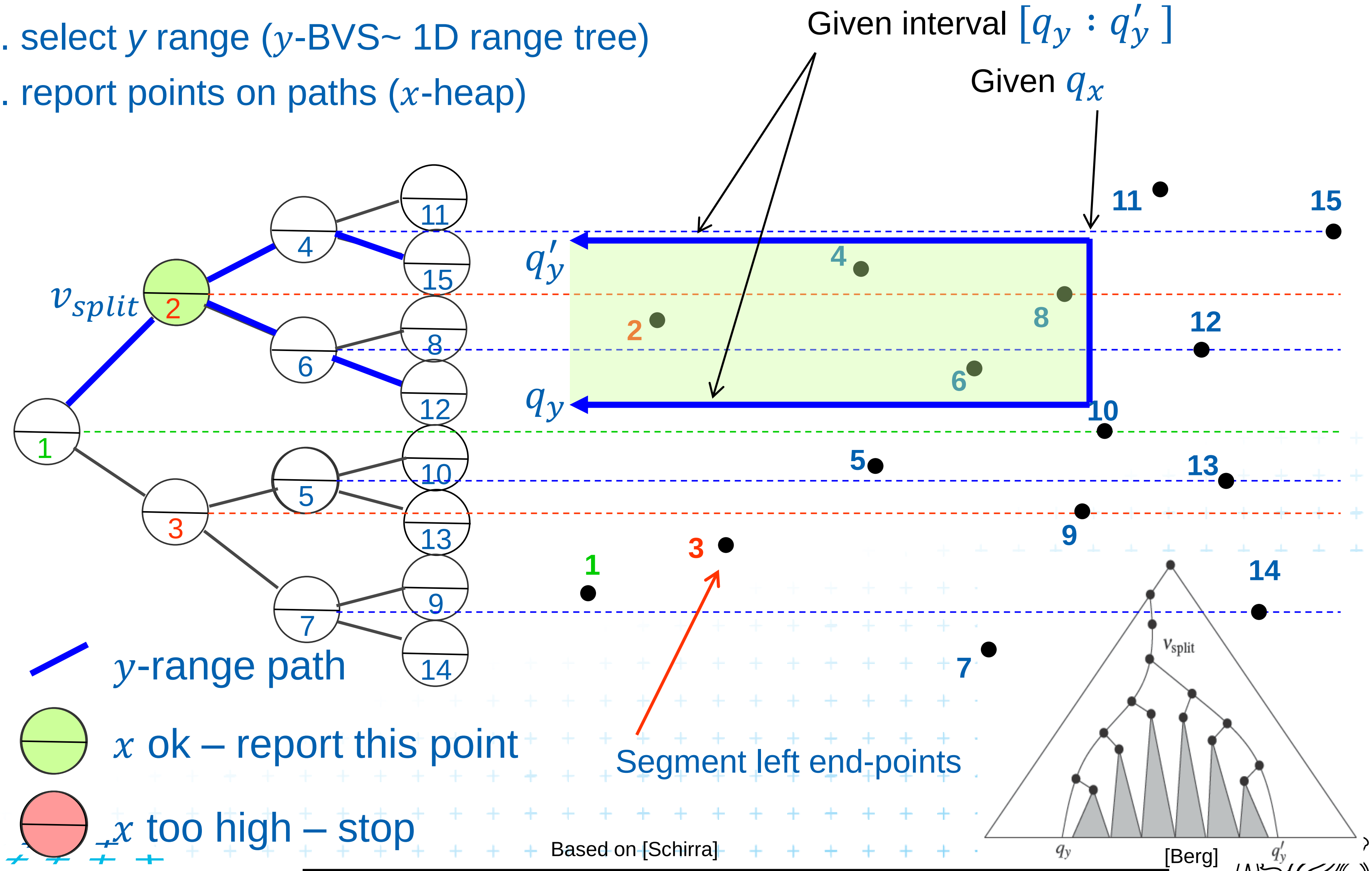
- y-range path
- 🟩 x ok – report this point
- 🟥 x too high – stop

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



Given interval $[q_y : q'_y]$

Given q_x

Segment left end-points

y-range path

🟩 x ok – report this point

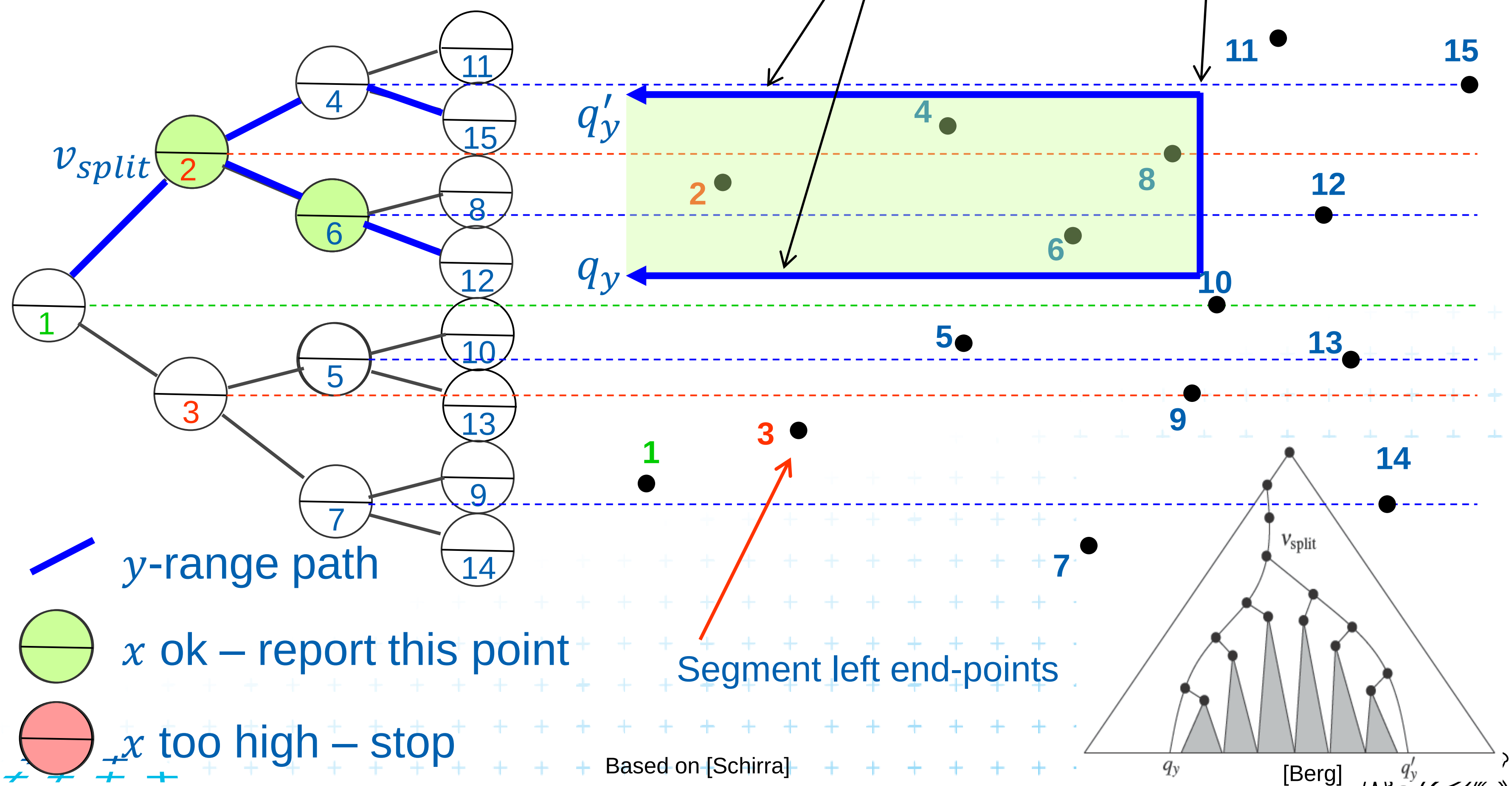
🟥 x too high – stop

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select y range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



Given interval $[q_y : q'_y]$

Given q_x

Segment left end-points

— y-range path

🟩 x ok – report this point

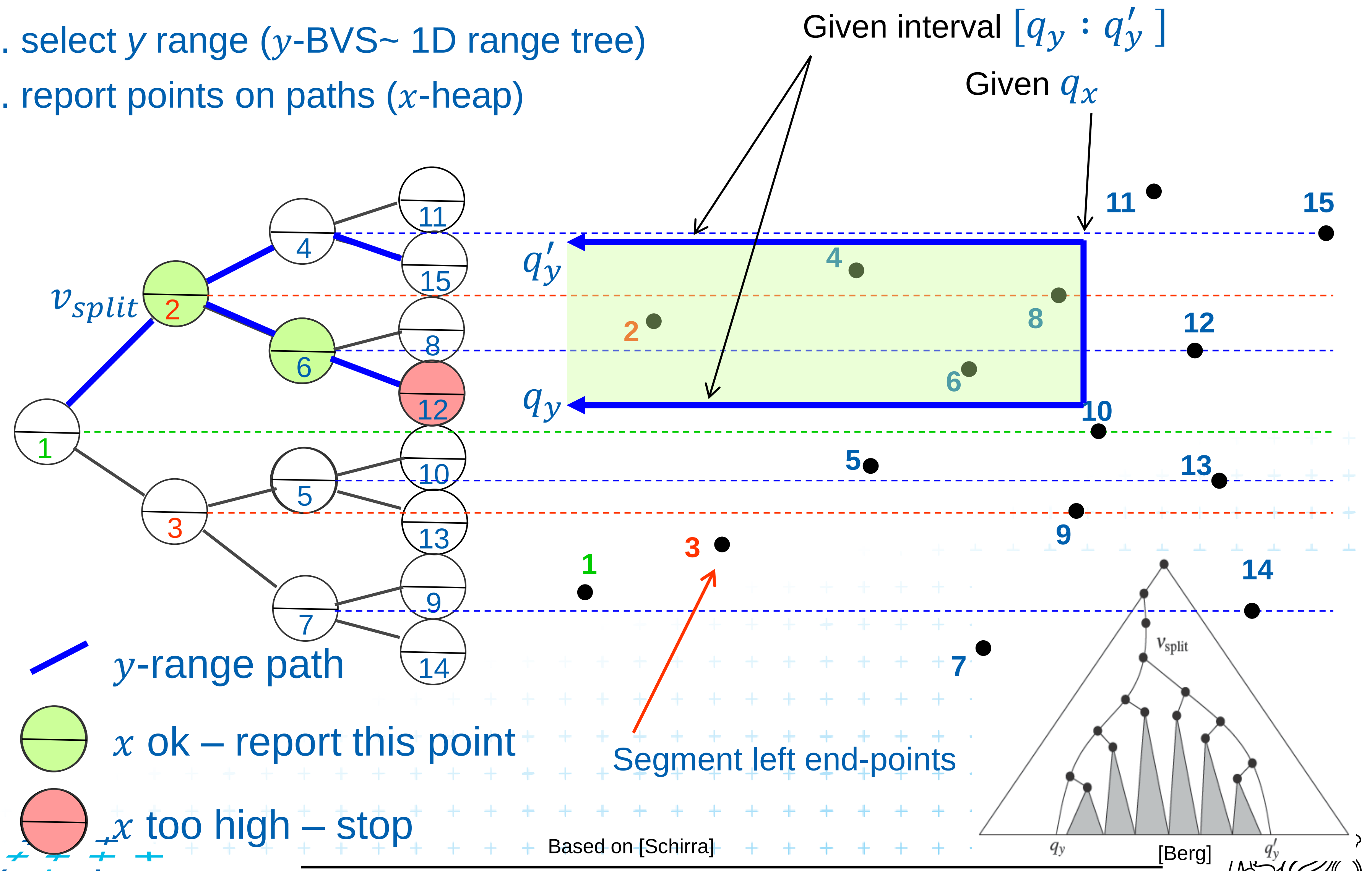
🟥 x too high – stop

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)

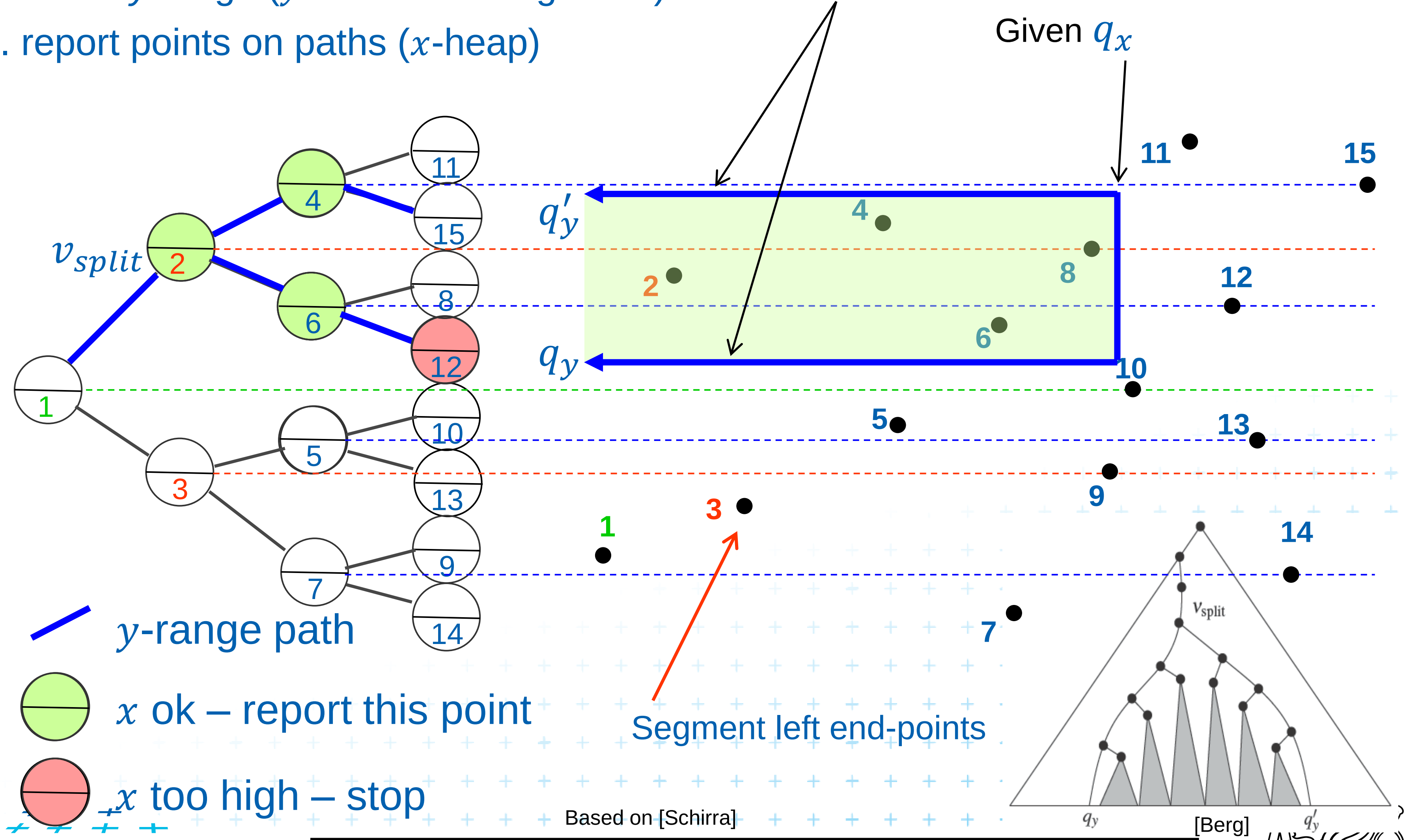


Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



Given interval $[q_y : q'_y]$

Given q_x

Segment left end-points

y-range path

x ok – report this point

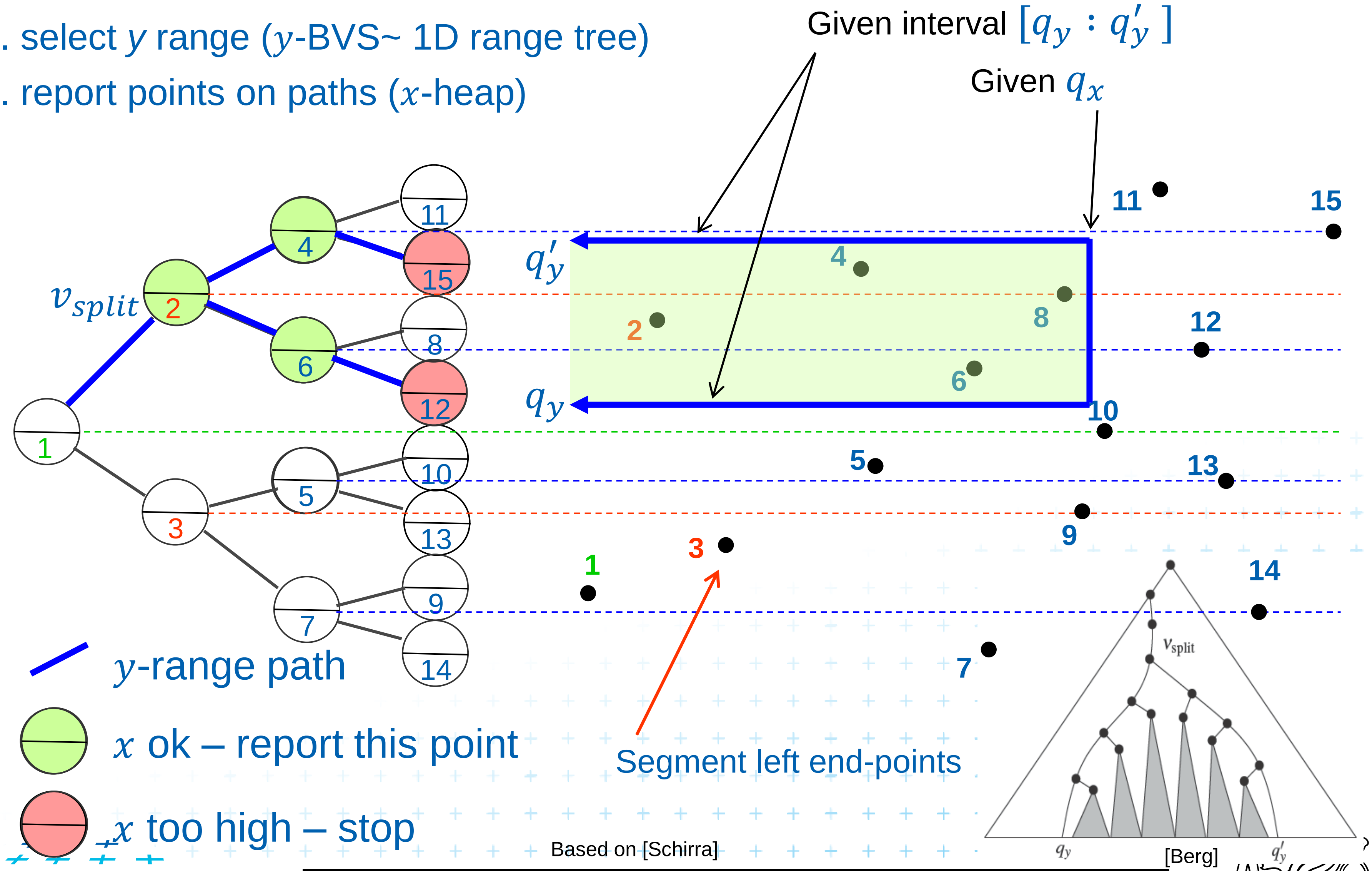
x too high – stop

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



▬ y-range path

🟩 x ok – report this point

🟥 x too high – stop

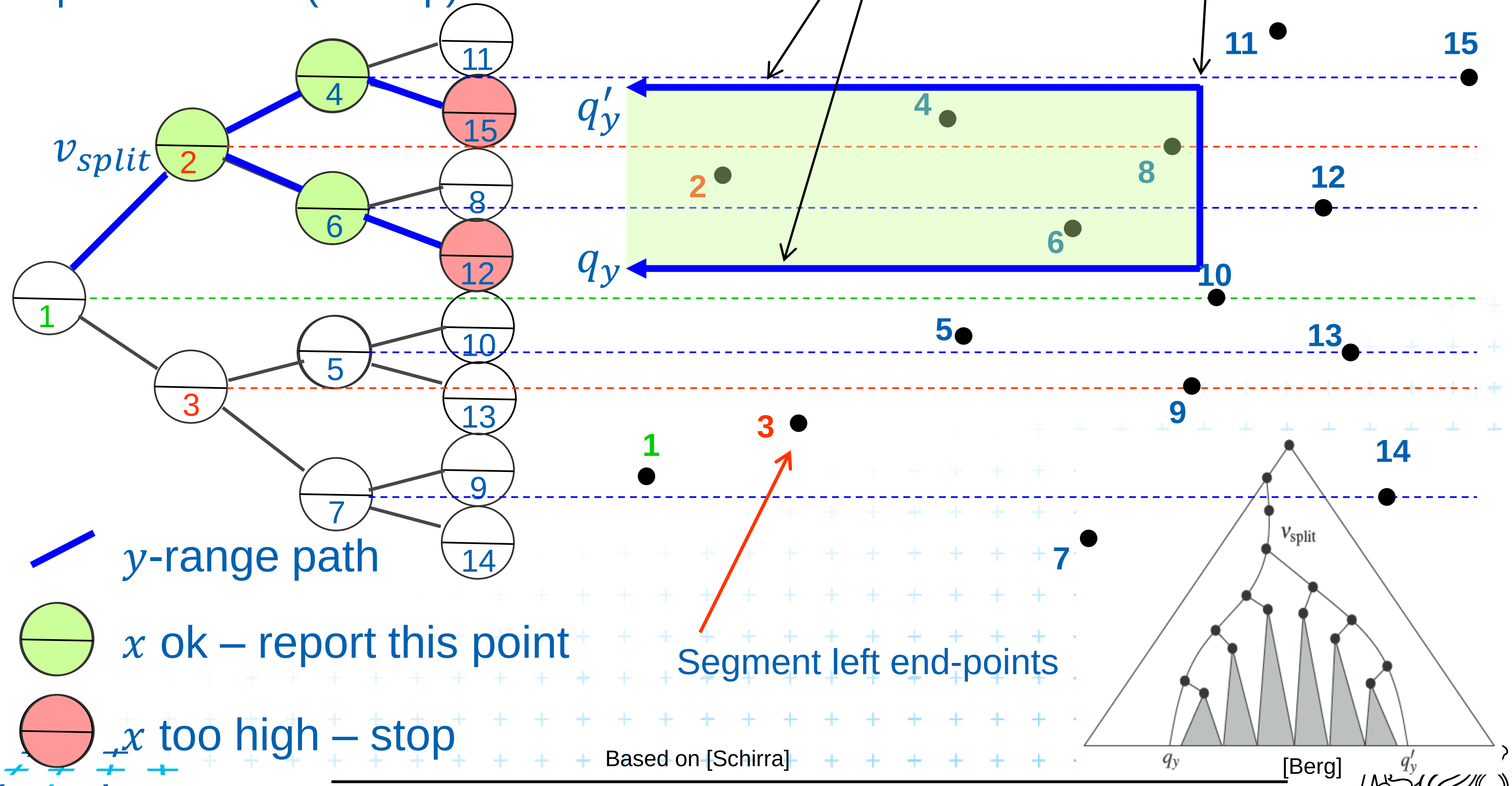
Based on [Schirra]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select y range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)
3. report subtrees (x-heap)

Given interval $[q_y : q'_y]$

Given q_x



━ y-range path

🟩 x ok – report this point

🟥 x too high – stop

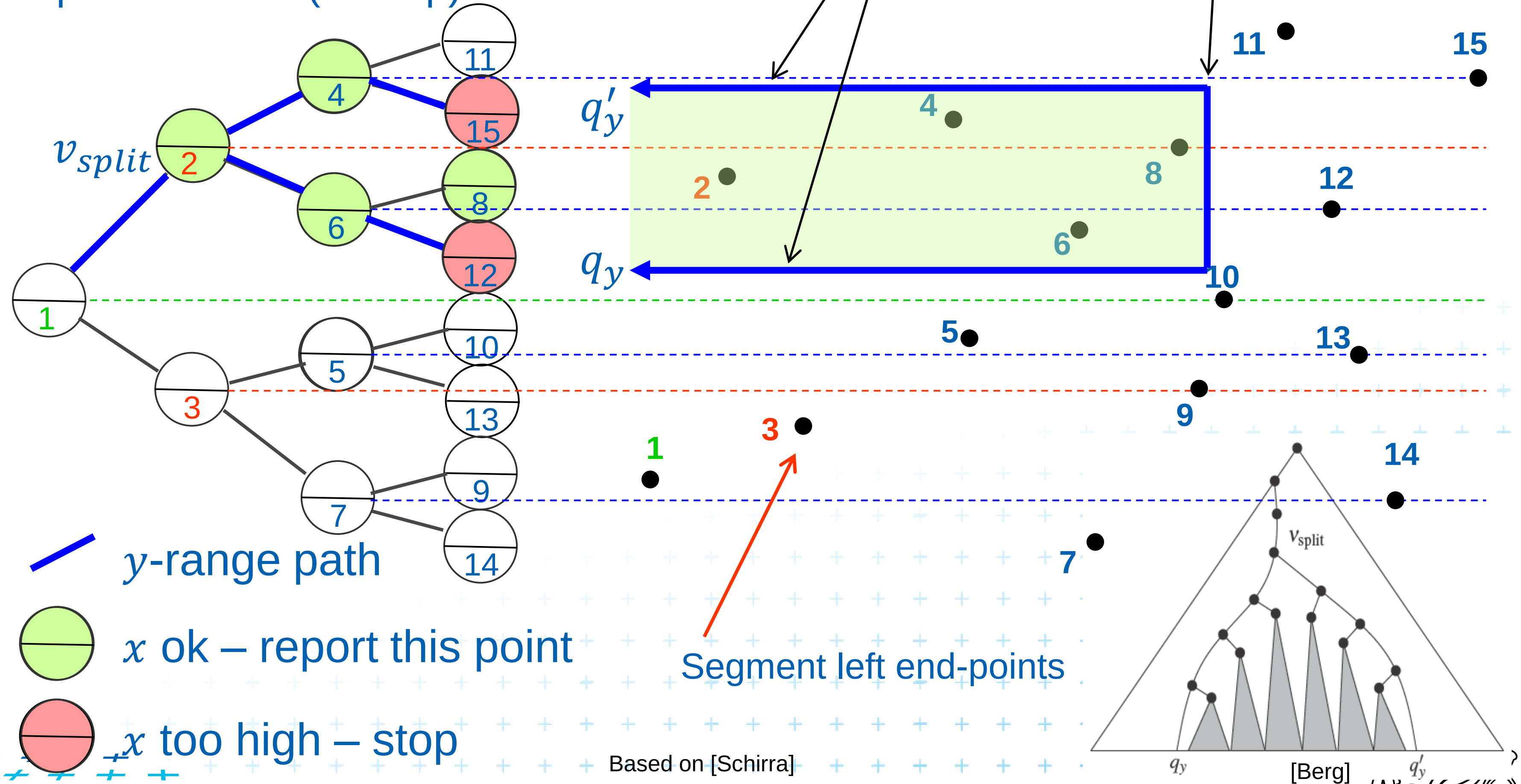
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query $(-\infty : q_x] \times [q_y : q'_y]$

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)
3. report subtrees (x-heap)



Given interval $[q_y : q'_y]$

Given q_x

Segment left end-points

━ y-range path

🟩 x ok – report this point

🟥 x too high – stop

Based on [Schirra]

[Berg]

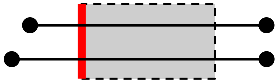
Priority search tree complexity

For set of n points in the plane

- Build $O(n \log n)$
- Storage $O(n)$
- Query $O(k + \log n)$
  - points in query range $(-\infty : q_x] \times [q_y : q'_y]$
  - k is number of reported points

- Use Priority search tree as associated data structure for interval trees for storage of set M (one for M_L, one for M_R)

Talk overview



1. Windowing of axis parallel line segments in 2D
   - 3 variants of *interval tree – IT in x-direction*
   - Differ in storage of segment end points M_L and M_R

1D i. Line stabbing (standard *IT* with *sorted lists*) lecture 9 - intersections

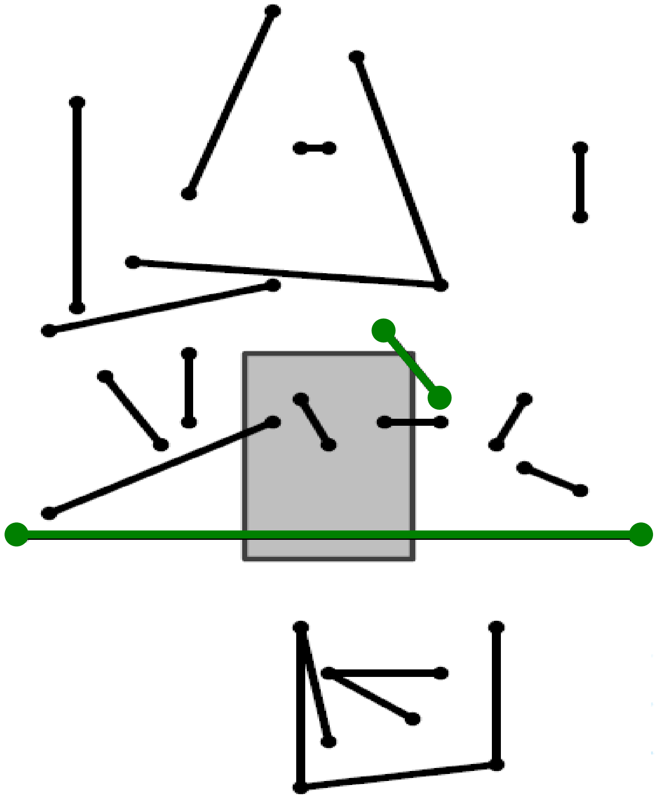
2D ii. Line segment stabbing (*IT* with *range trees*)

iii. Line segment stabbing (*IT* with *priority search trees*)

2. Windowing of line segments in general position

2D – *segment tree + BST*

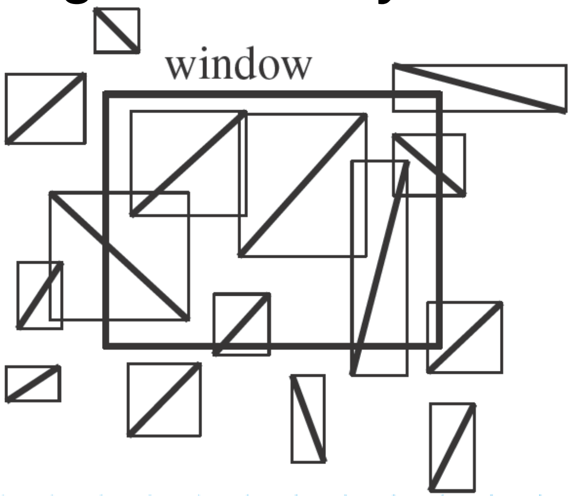
2. Windowing of line segments in general position

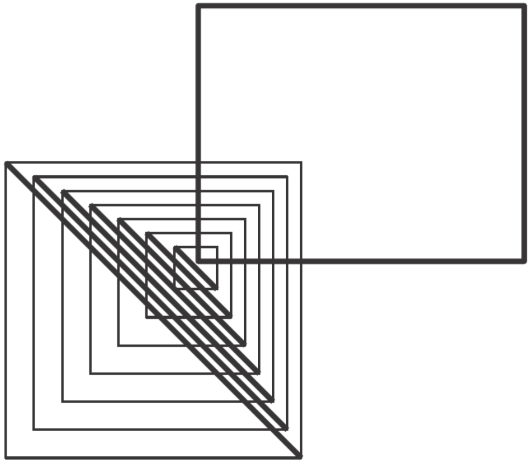


[Vakken]

Windowing of arbitrary oriented line segments

- Two cases of intersection
  - a,b) Endpoint inside the query window => range tree
  - c) Segment intersects side of query window => ???
- Intersection with BBOX (segment bounding box)?
  - – Intersection with 4n sides of the segment BBOX?
  - – But segments may not intersect the window –> query y





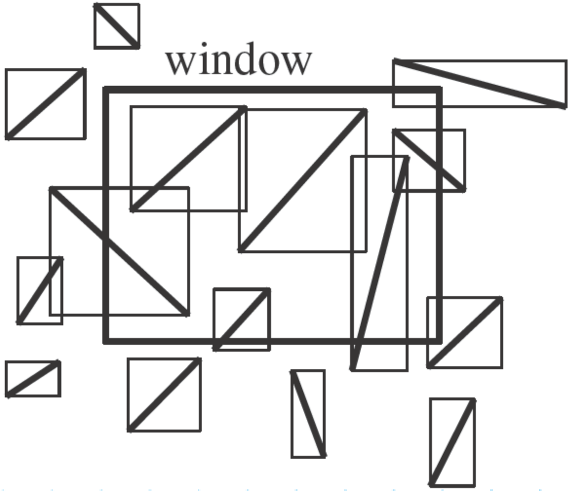
Windowing of arbitrary oriented line segments

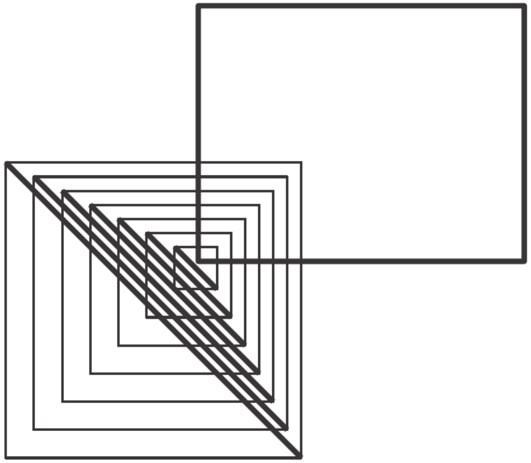
- Two cases of intersection

  a,b) Endpoint inside the query window => range tree

  c) Segment intersects side of query window => ???

- Intersection with BBOX (segment bounding box)?
  - Intersection with 4n sides of the segment BBOX?
  - But segments may not intersect the window –> query y





NOT

Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)

1D i. Line stabbing (*IT* with *sorted lists*)

2D ii. Line segment stabbing (*IT* with *range trees*)

iii. Line segment stabbing (*IT* with *priority search trees*)

2. Windowing of line segments in general position

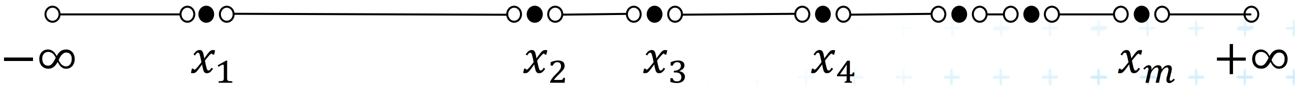
2D – *segment tree*

Note: segment = interval
it consists of elementary intervals

Segment tree

[Bentley, 1977]

- Exploits locus approach
  - Partition parameter space into regions of same answer
  - Localization of such region = knowing the answer
- For given set S of n intervals (segments) on real line
  - Finds m elementary intervals (induced by interval end-points)
  - Partitions 1D parameter space into these elementary intervals



$$(-\infty : x_1), [x_1 : x_1], (x_1 : x_2), [x_2 : x_2], \ldots,$$
$$(x_{m-1} : x_m), [x_m : x_m], (x_m : +\infty)$$

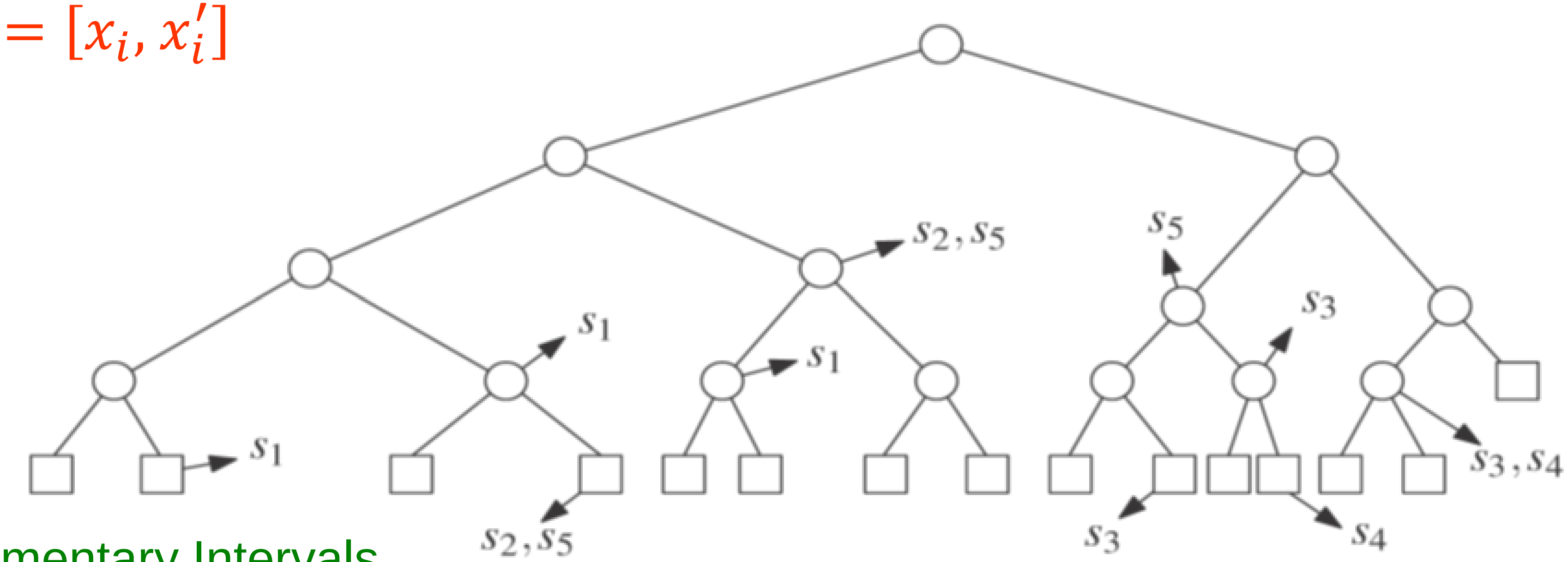
  - Stores line segments s_i with the elementary intervals
  - Reports the segments s_i containing query point q_x.

Plain is partitioned into vertical slabs

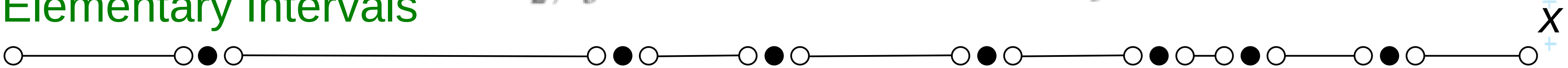
Segment tree example

Segments $S = \{s_1, s_2, \dots, s_n\}$

$s_i = [x_i, x_i']$



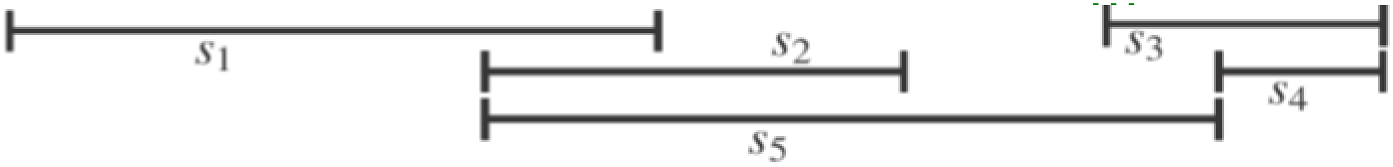
Elementary Intervals



$(-\infty : x_1)$ $(x_1 : x_2)$ $\dots$ $(x_m : +\infty)$

$[x_1 : x_1]$ $[x_2 : x_2]$ $[x_3 : x_3]$ $[x_m : x_m]$

Intervals

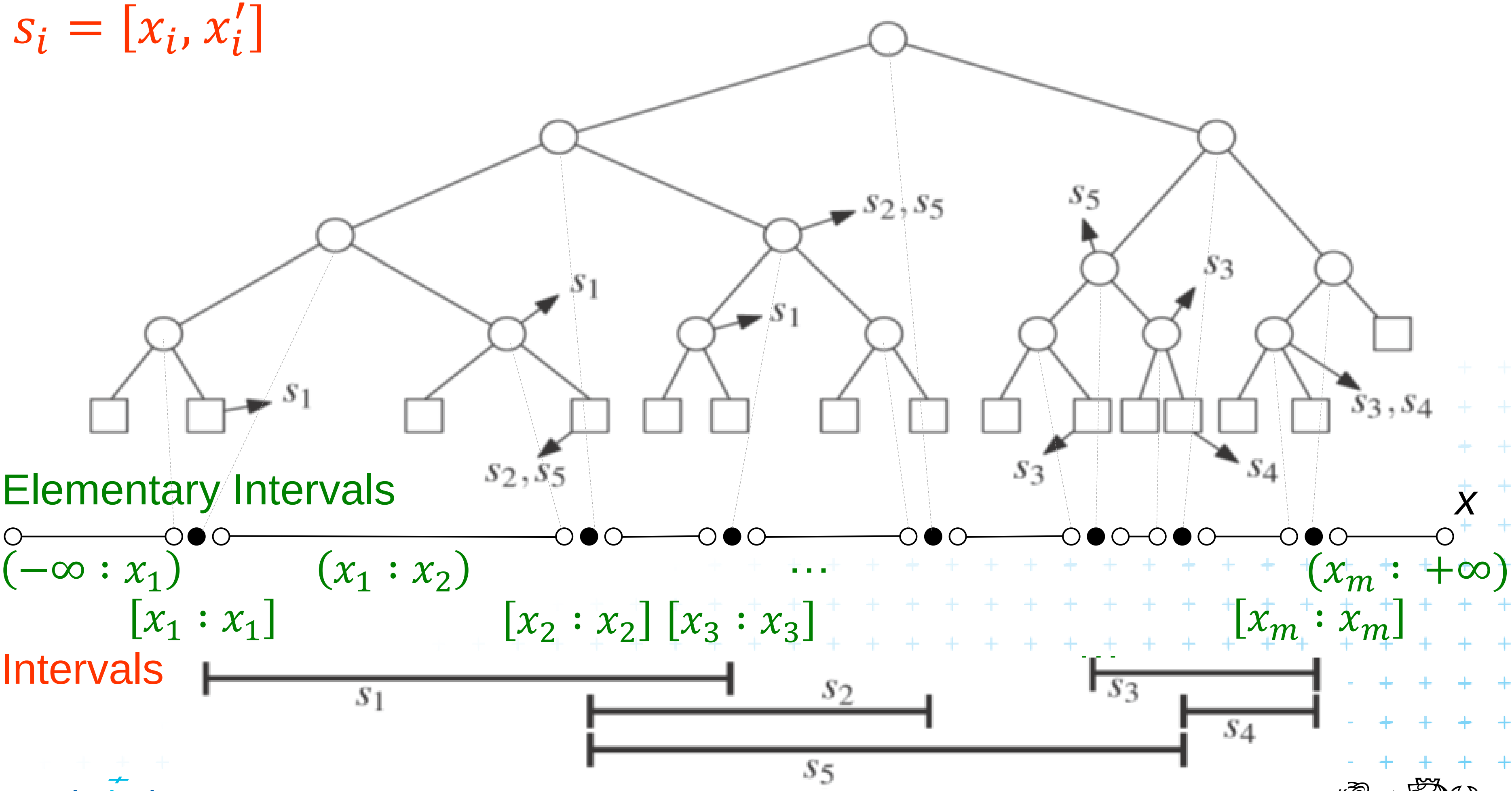


[Berg]

Segment tree example

Segments $S = \{s_1, s_2, \dots, s_n\}$

$s_i = [x_i, x_i']$



Elementary Intervals

x

$(-\infty : x_1)$ $(x_1 : x_2)$ $\dots$ $(x_m : +\infty)$

$[x_1 : x_1]$ $[x_2 : x_2]$ $[x_3 : x_3]$ $[x_m : x_m]$

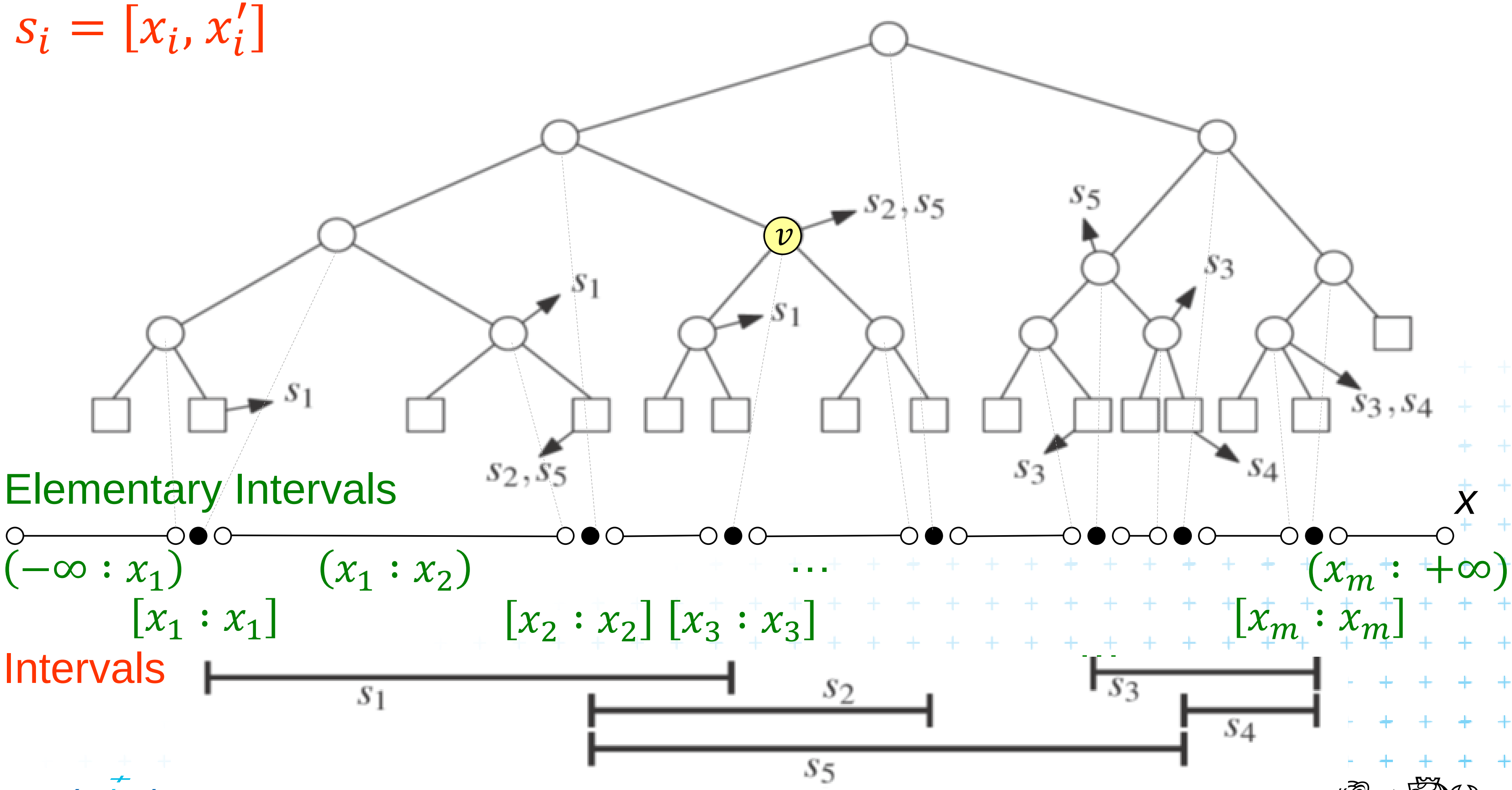
Intervals

[Berg]

Segment tree example

Segments $S = \{s_1, s_2, \dots, s_n\}$

$s_i = [x_i, x_i']$



Elementary Intervals

$(-\infty : x_1)$ $[x_1 : x_1]$ $(x_1 : x_2)$ $[x_2 : x_2]$ $[x_3 : x_3]$ … $(x_m : +\infty)$ $[x_m : x_m]$

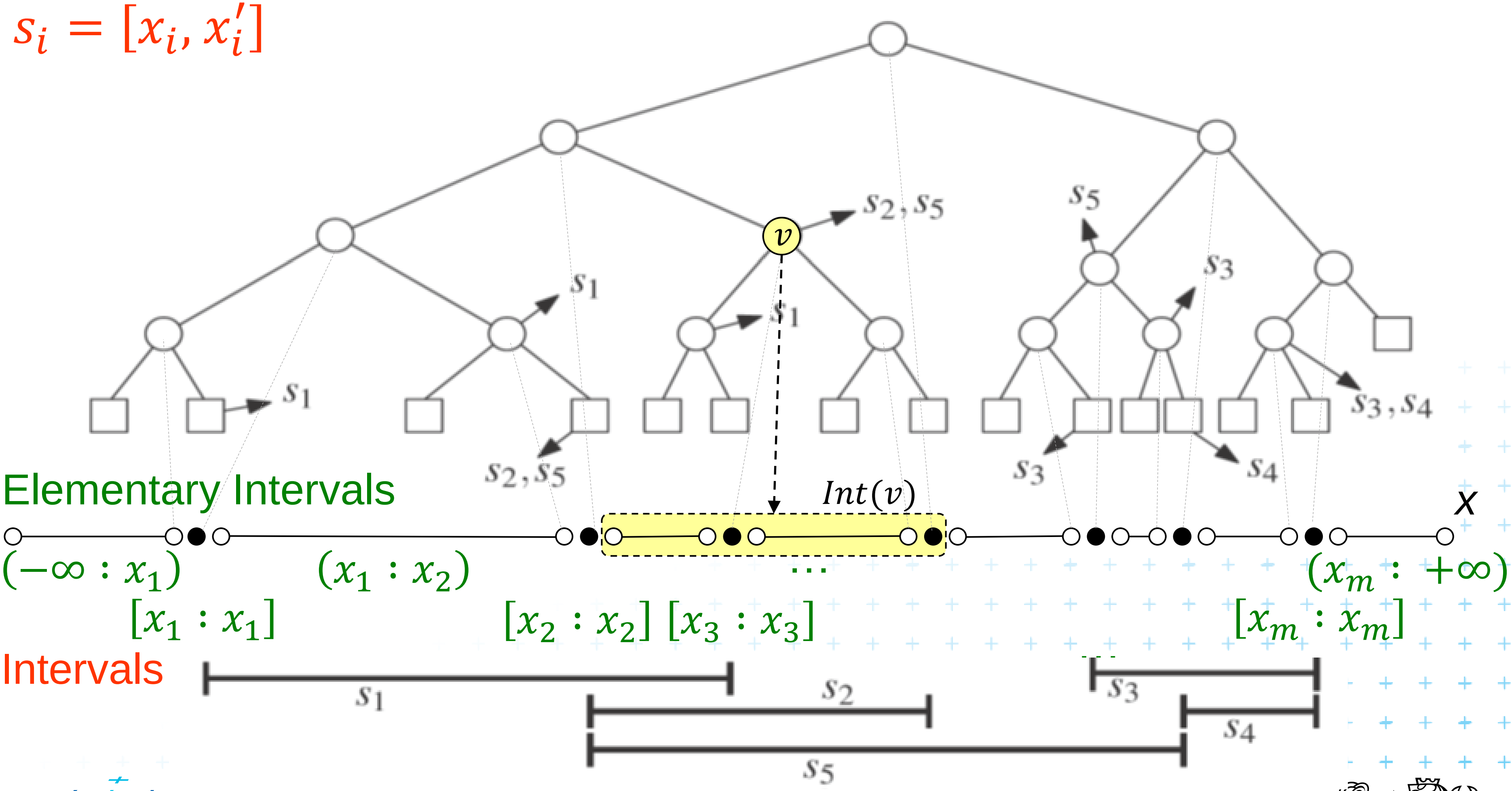
Intervals

[Berg]

Segment tree example

Segments $S = \{s_1, s_2, \dots, s_n\}$

$s_i = [x_i, x'_i]$



Elementary Intervals

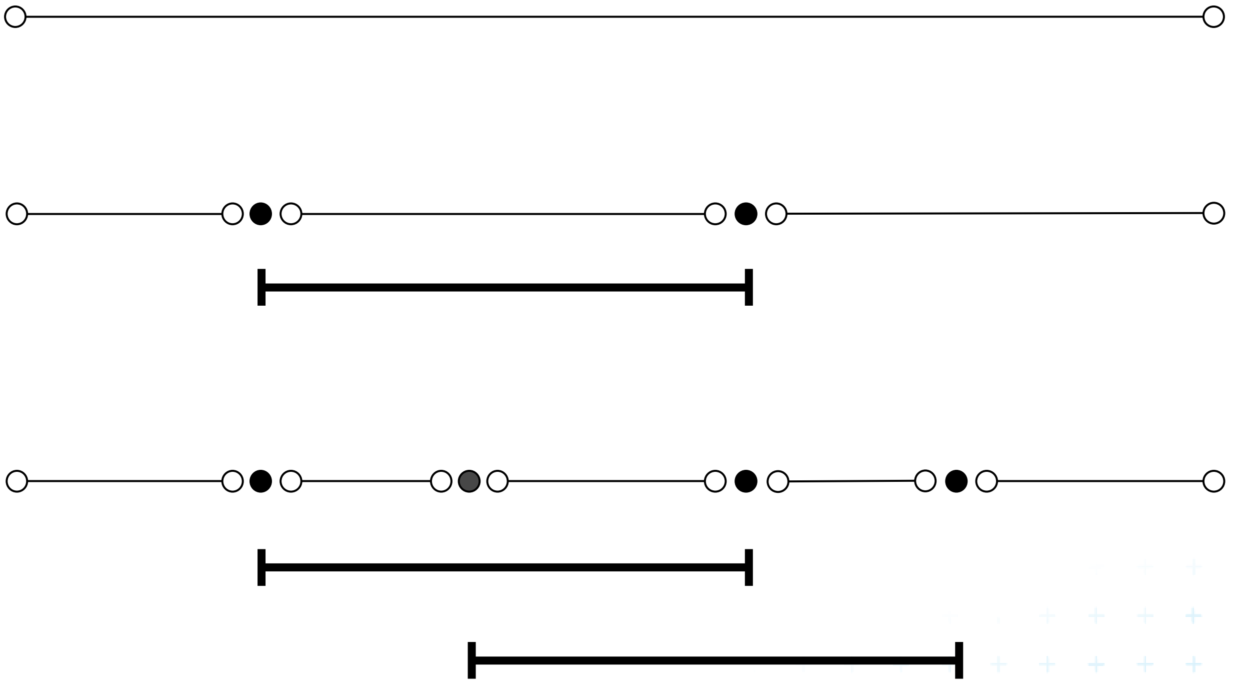
$(-\infty : x_1)$ $(x_1 : x_2)$ $\dots$ $(x_m : +\infty)$

$[x_1 : x_1]$ $[x_2 : x_2]$ $[x_3 : x_3]$ $[x_m : x_m]$

Intervals

[Berg]

Number of elementary intervals for n segments



$n = 0$ $\# = 1$

$n = 1$ $\# = 4 + 1$

$n = 2$ $\# = 4 * 2 + 1$

n Each end-point adds two elementary intervals $\# = 4n + 1$

Each segment four…

Segment tree definition

Segment tree

- Skeleton is a balanced binary tree T
- Leaves ~ elementary intervals
- Internal nodes v
  ~ union of elementary intervals of its children
  - Store: 1. interval $Int(v)$ = union of elementary intervals of its children

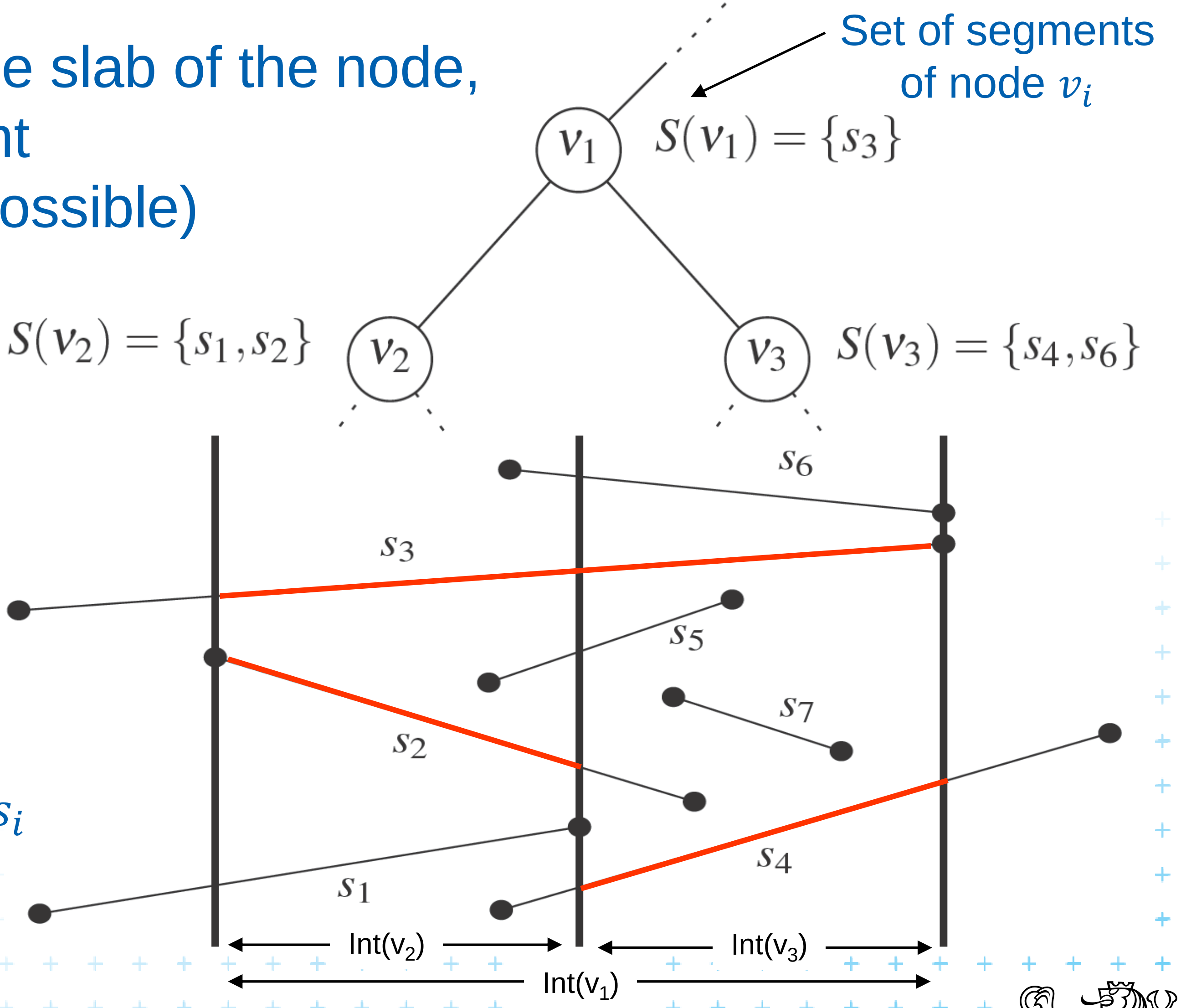
    segments s_i

    2. canonical set $S(v)$ of segments $[x_i : x_i']\in S$
  - Holds $Int(v) \subseteq [x_i : x_i']$ and $Int(\text{parent}(v)] \nsubseteq [x_i : x_i']$ (node interval is not larger than the segment)
  - Segments $[x_i : x_i']$ are stored as high as possible, such that $Int(v)$ is completely contained in the segment

DCGI

Segments span the slab

Segments span the slab of the node,
but not of its parent
(stored as up as possible)



$Int(v_j) \subseteq s_i$

and

$Int(\text{parent}(v)) \nsubseteq s_i$

[Berg]

Query segment tree – stabbing query (1D)

QuerySegmentTree(v, q_x)
Input: The root of a (subtree of a) segment tree and a query point q_x
Output: All intervals (=segments) in the tree containing q_x.

```
Report all the intervals $s_i$ in $S(v)$.        // covered by the current node
if $v$ is not a leaf                             //     root covers "all" $(-\infty, +\infty)$
    if $q_x \in$ Int( $l(v)$ )                    // go left
        QuerySegmentTree( $l(v)$, $q_x$ )
    else                                          // or go right
        QuerySegmentTree( $r(v)$, $q_x$ )
```

Query time $O(\log n + k)$, where k is the number of reported intervals

$O(1 + k_v)$ for one node

Height $O(\log n)$

Segment tree construction

ConstructSegmentTree(S)
Input: Set of intervals (segments) S
Output: segment tree

```
1. Sort endpoints of segments in $S$, get elementary intervals … $O(n \log n)$
2. Construct a binary search tree $T$ on elementary intervals … $O(n)$
   (bottom up) and determine the interval $Int(v)$ it represents
3. Compute the canonical subsets for the nodes
   (lists of their segments $s_i$):
4.     v = root( $T$ )
5.     for all segments $s_i = [x_i : x_i'] \in S$
6.         InsertSegmentTree( $v$, $[x_i : x_i']$)
```

Segment tree construction – interval insertion

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

Input: The root of (a sub-tree of) a segment tree and an interval.

Output: The interval will be stored in the sub-tree.

```
if $\mathrm{Int(v)} \subseteq [x : x']$                 // $\mathrm{Int(v)}$ contains $s_i = [x : x']$
    store $s_i = [x : x']$ at $v$
else if $\mathrm{Int}(\,\mathrm{l(v)}) \cap [x : x'] \neq \emptyset$   // part of $s_i$ to the left
        InsertSegmentTree( $\mathrm{l(v)}$, $[x : x']$ )
      if $\mathrm{Int}(\,\mathrm{r(v)}) \cap [x : x'] \neq \emptyset$     // part of $s_i$ to the right
        InsertSegmentTree( $\mathrm{r(v)}$, $[x : x']$ )
```

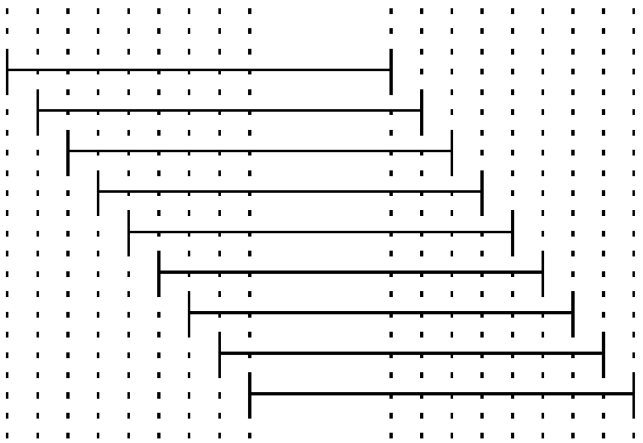
One interval is stored at most twice in one level =>

- Single interval insert $O(\log n)$, insert n intervals $O(\cancel{2}n \log n)$
- Construction total $O(n \log n)$

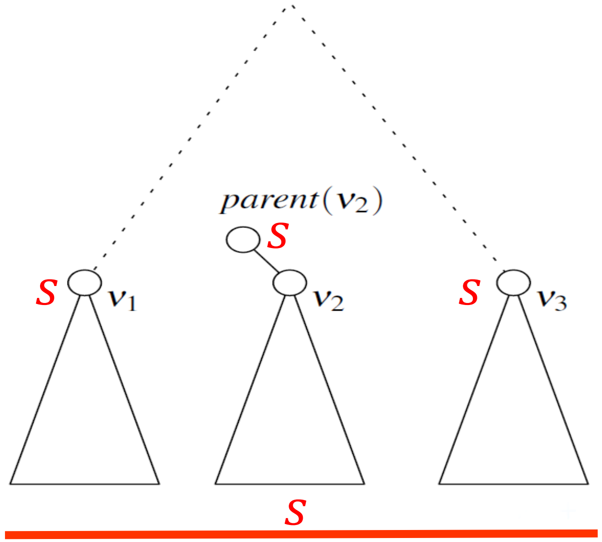
Storage $O(n \log n)$

- Tree height $O(\log n)$, name stored max 2x in one level
- Storage total $O(n \log n)$ – see next slide

Space complexity - notes



[Berg]



[Berg]

Worst case – $O(n^2)$ segments in leafs

But

Store segments as high, as possible

Segment max 2 times in one level $\Leftarrow$

s covered by v_1 and v_3

$\Rightarrow v_2$ covered, $Int(v_2) \in s$

As v_2 lies between v_1 and v_3

$\Rightarrow Int(parent(v_2)) \in s \Rightarrow$ segment s will not be stored in v_2

$\max 4n + 1$ elementary intervals (leaves)

$\Rightarrow O(n)$ space for the tree

$\Rightarrow O(n \log n)$ space for interval names

Segment tree complexity

A segment tree for set S of n intervals in the plane,

- Build $O(n \log n)$
- Storage $O(n \log n)$
- Query $O(k + \log n)$
  - Report all intervals that contain a query point
  - k is number of reported intervals

Segment tree versus Interval tree

- Segment tree
  - $O(n \log n)$ storage versus $O(n)$ of Interval tree
  - But returns exactly the intersected segments s_i, interval tree must search the lists M_L and/or M_R
- Good for
  1. extensions (allows different structuring of intervals)
  2. stabbing counting queries
     - store number of intersected intervals in nodes
     - $O(n)$ storage and $O(\log n)$ query time = optimal
  3. higher dimensions – multilevel segment trees
     (Interval and priority search trees do not exist in ^dims)

Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)

1D
   i. Line stabbing (standard *IT* with *sorted lists*)

2D
   ii. Line segment stabbing (*IT* with *range trees*)
   iii. Line segment stabbing (*IT* with *priority search trees*)

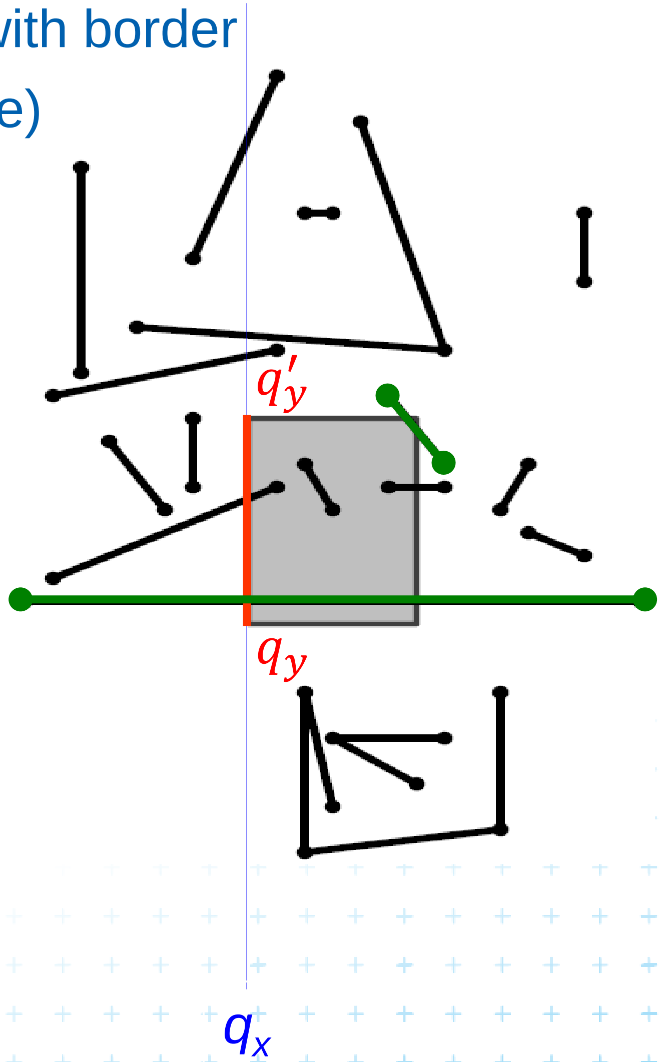
2. Windowing of line segments in general position

2D
   – *segment tree*
   – the windowing algorithm

2. Windowing of line segments in general position

Test intersection with border

Done 4x (rectangle)

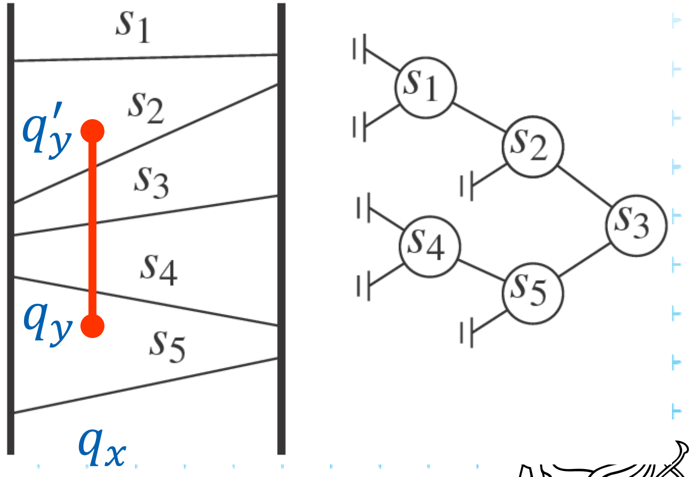


[Vakken]

Windowing of arbitrary oriented line segments

- Let S be a set of arbitrarily oriented line segments in the plane.
- Report the segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$ – window border
- Segment tree T on x intervals of segments in S
  - node v of T corresponds to vertical slab $Int(v) \times (-\infty : \infty)$
  - segments span the slab of the node, but not of its parent
  - segments do not intersect

  => segments in the slab (node) can be vertically ordered – BST

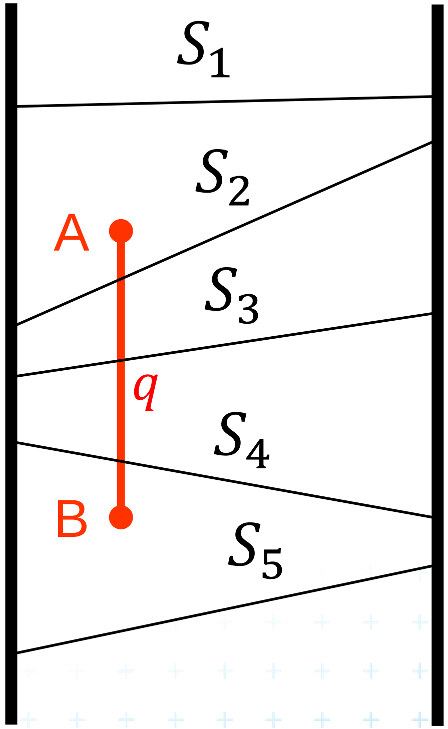


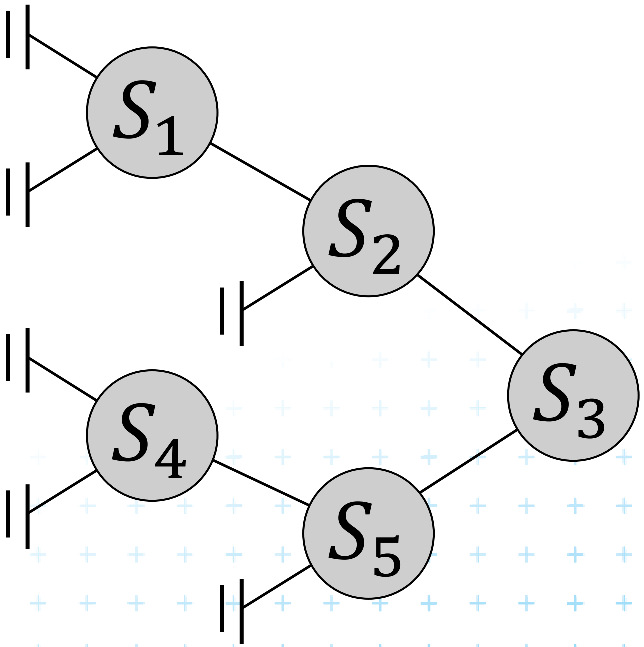
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



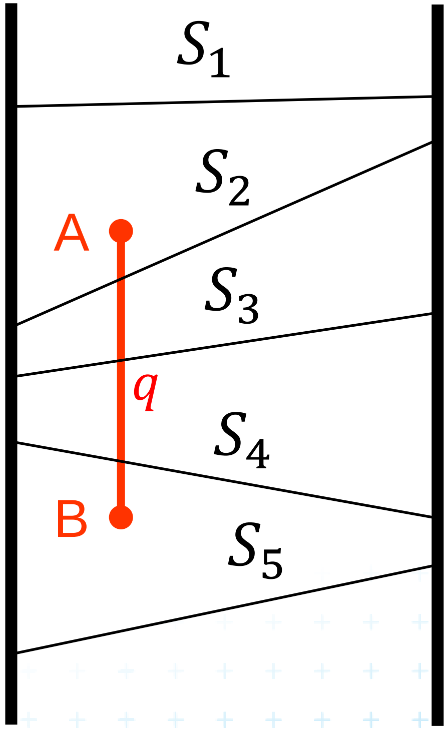


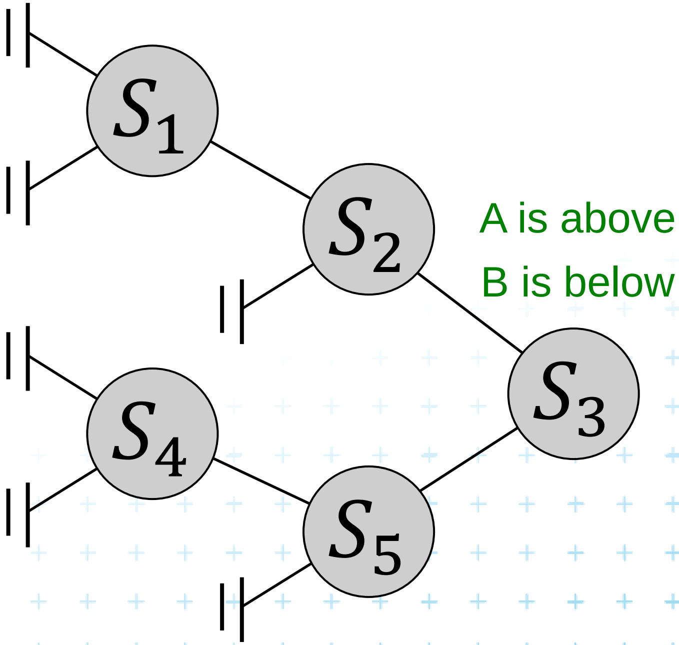
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



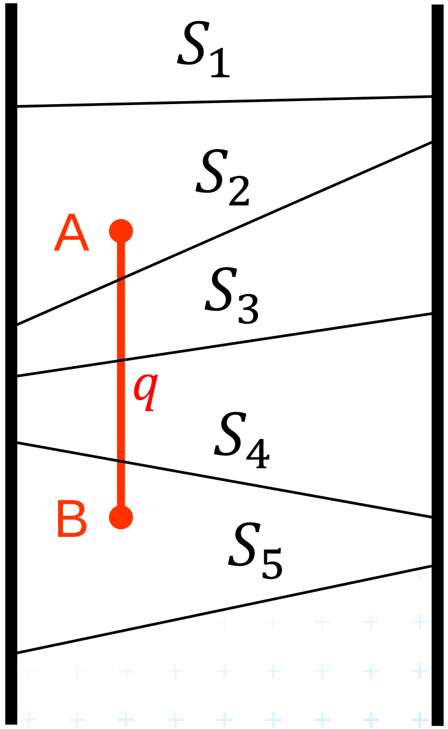


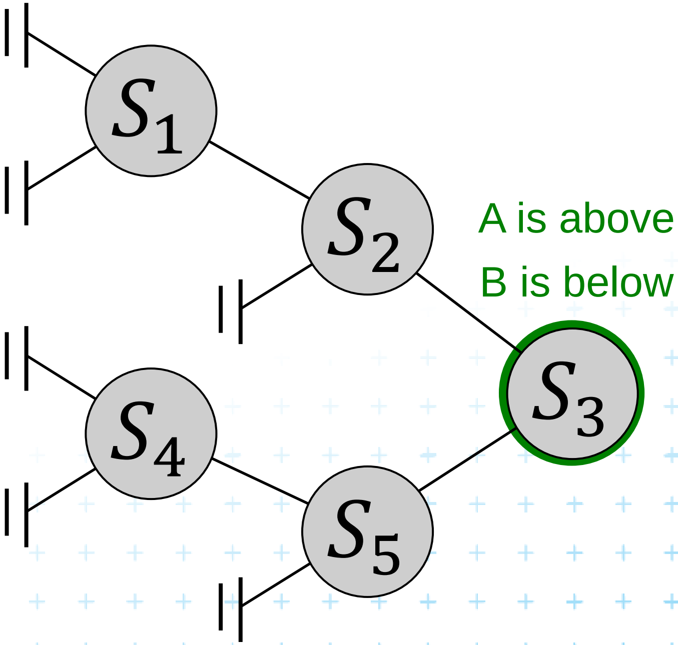
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



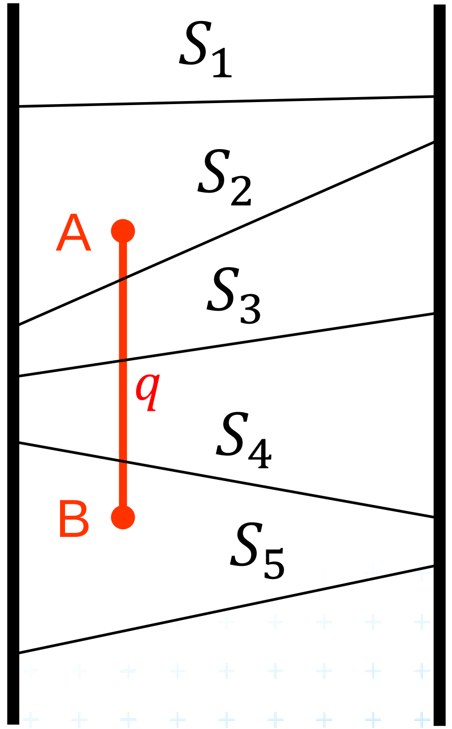


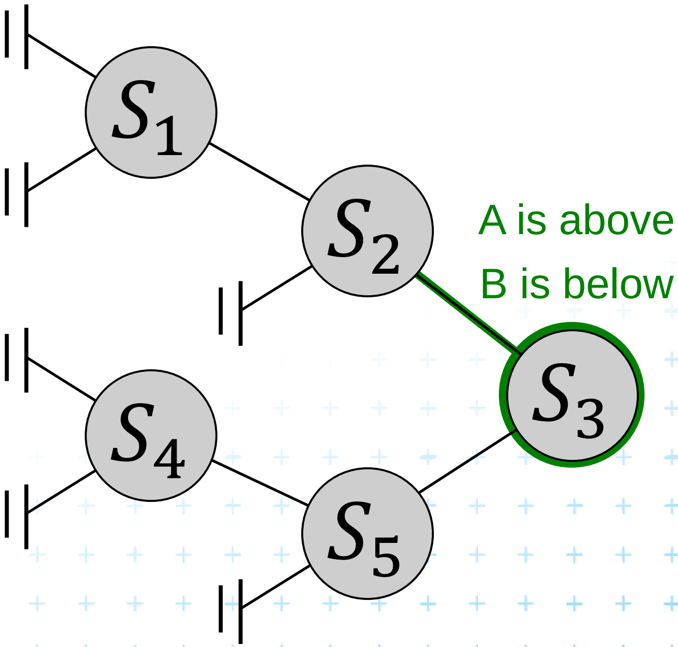
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



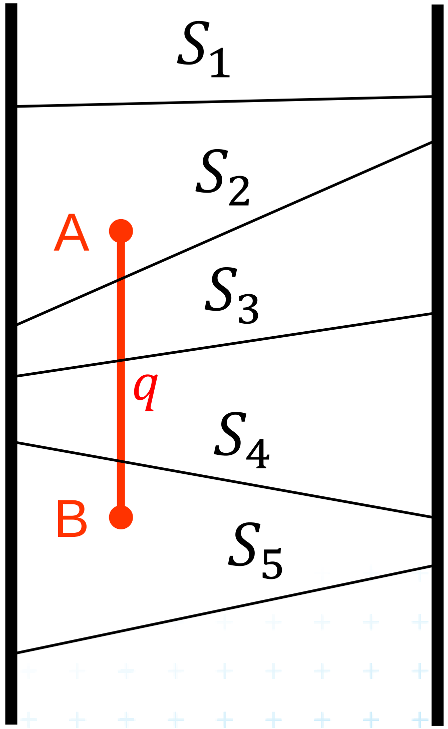


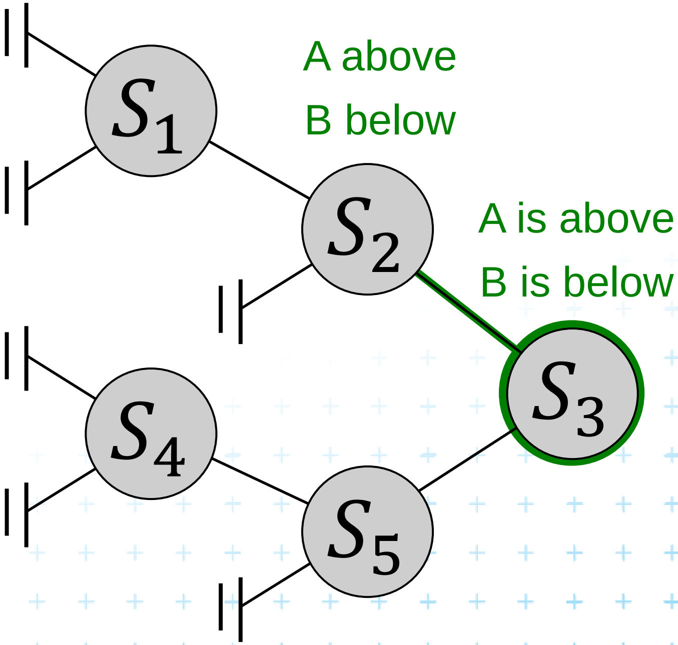
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



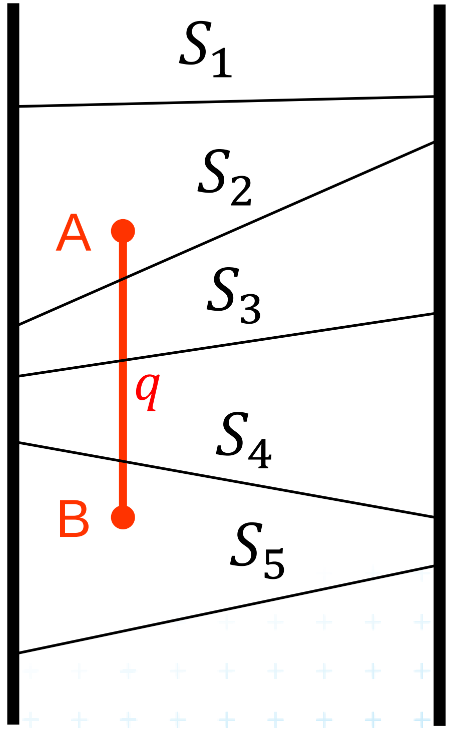


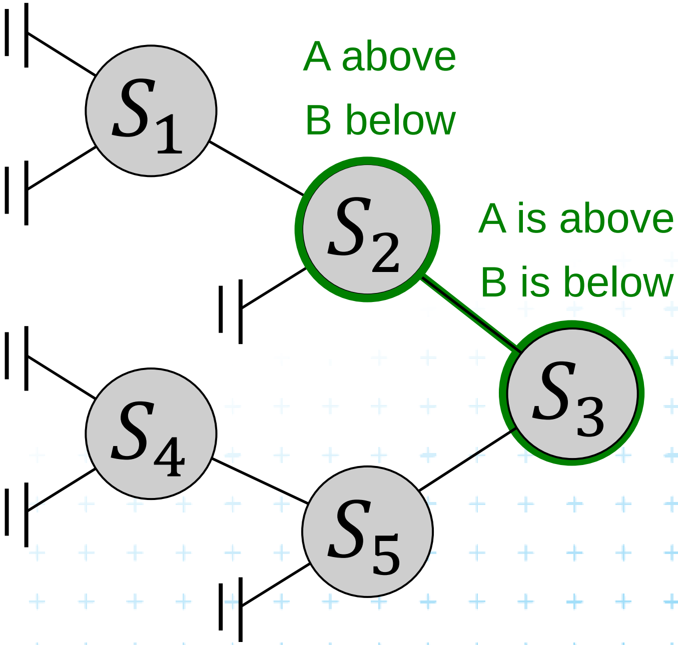
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



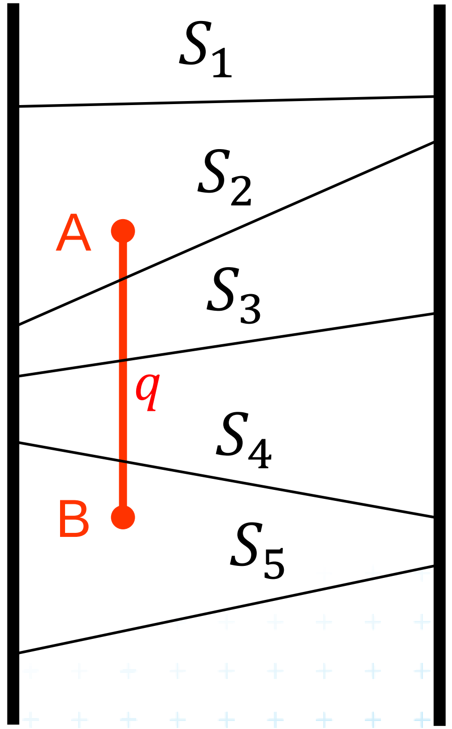


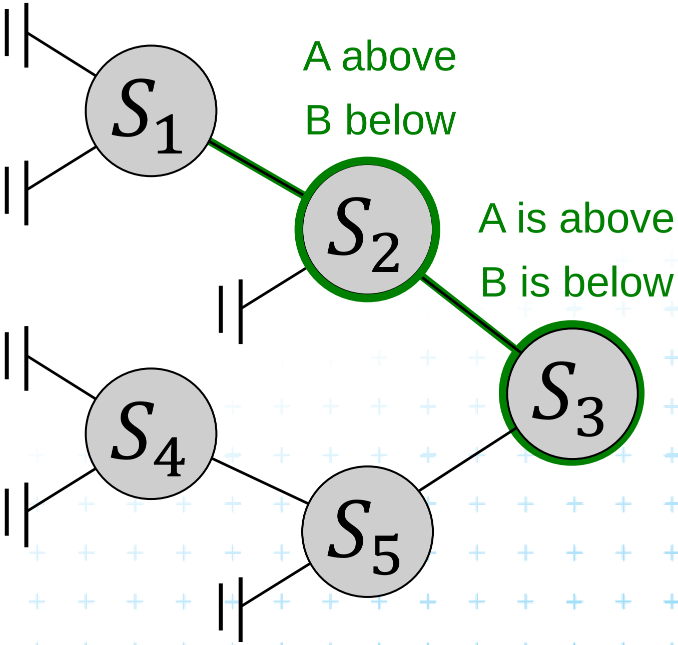
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



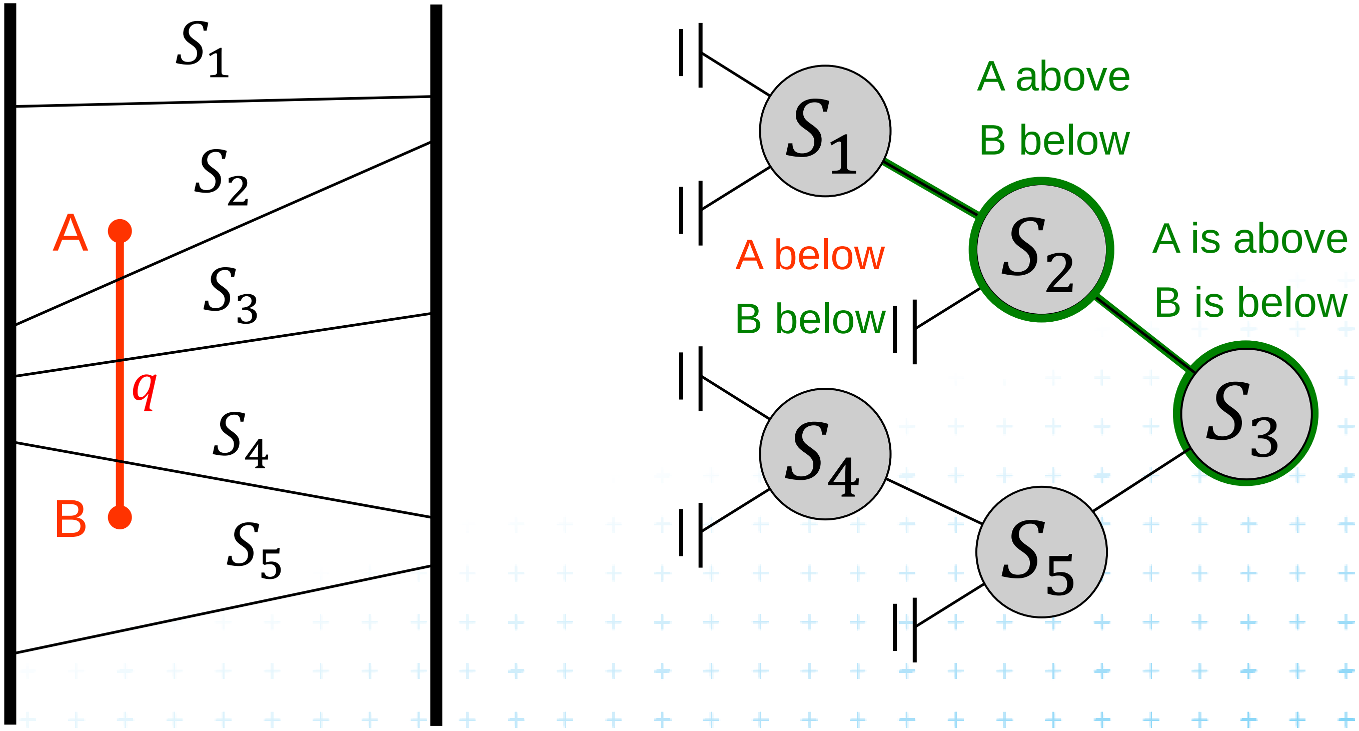


[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s

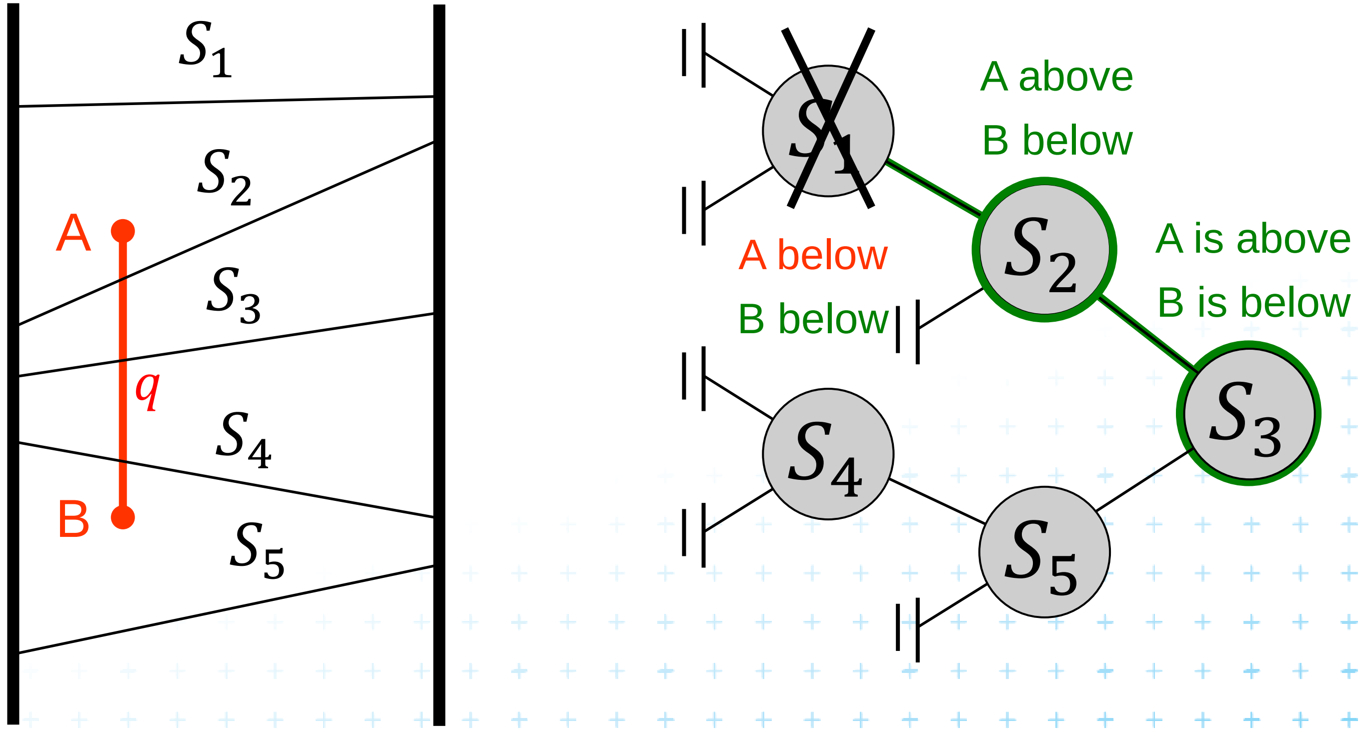


[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s

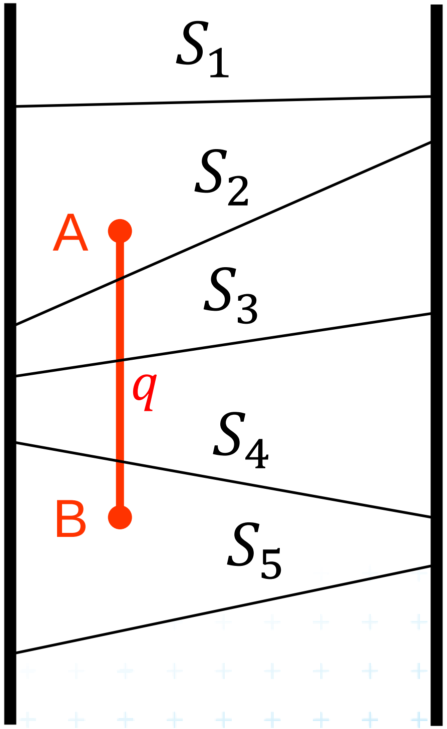


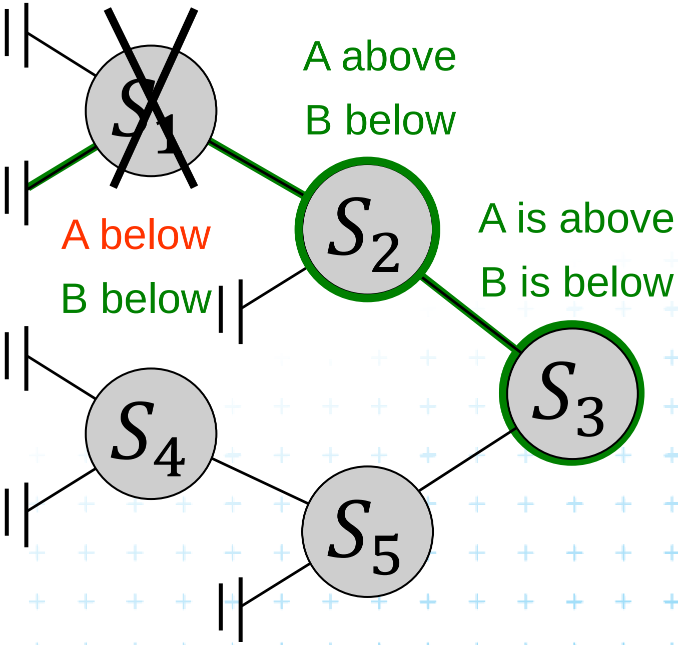
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



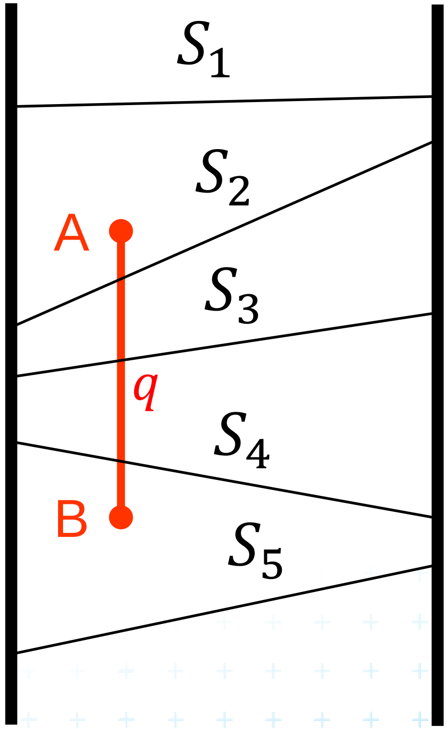

A above
B below
A below
B below
A is above
B is below

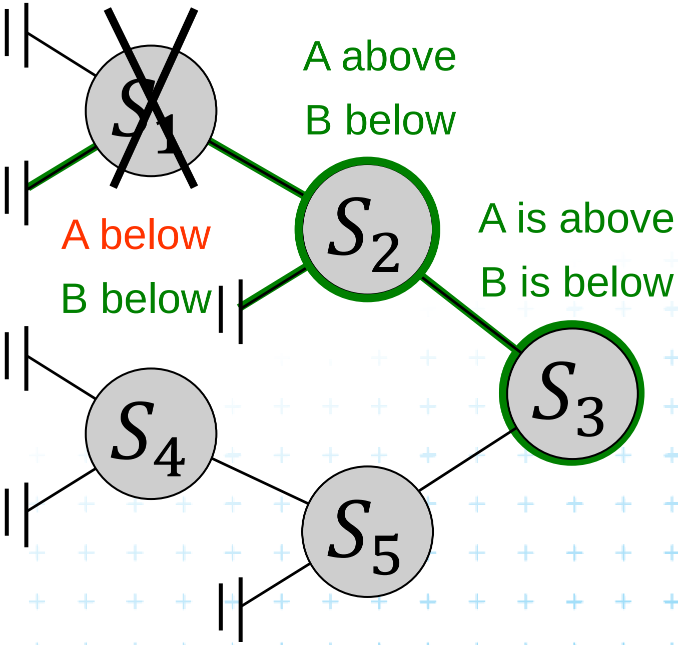
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



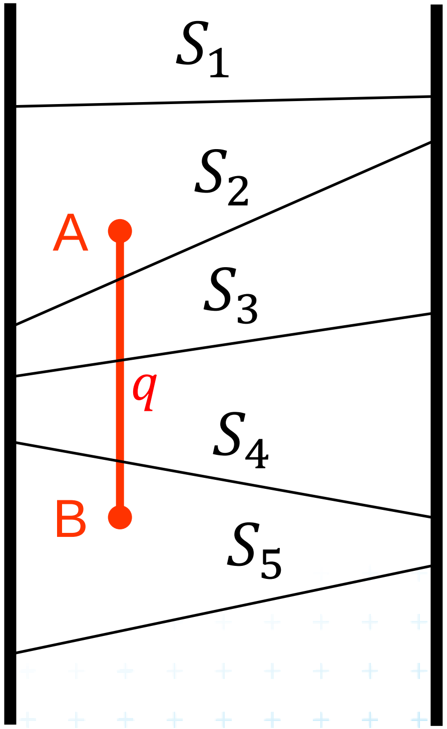


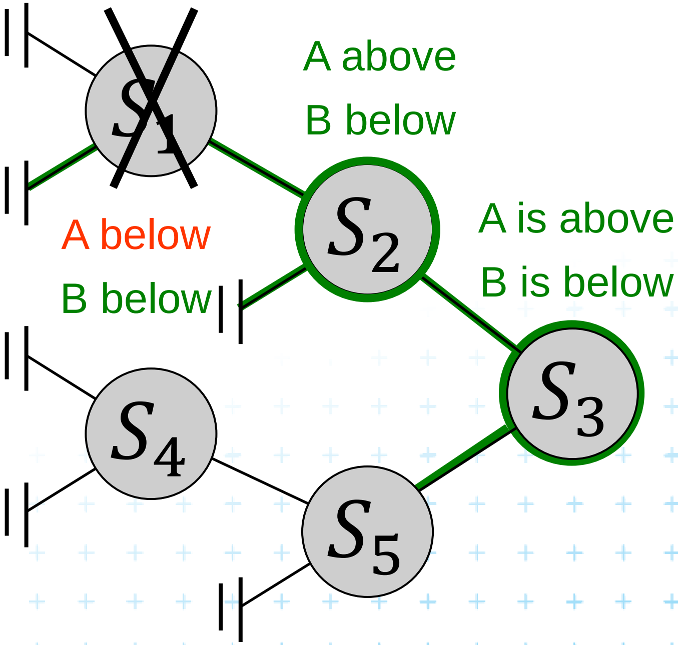
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



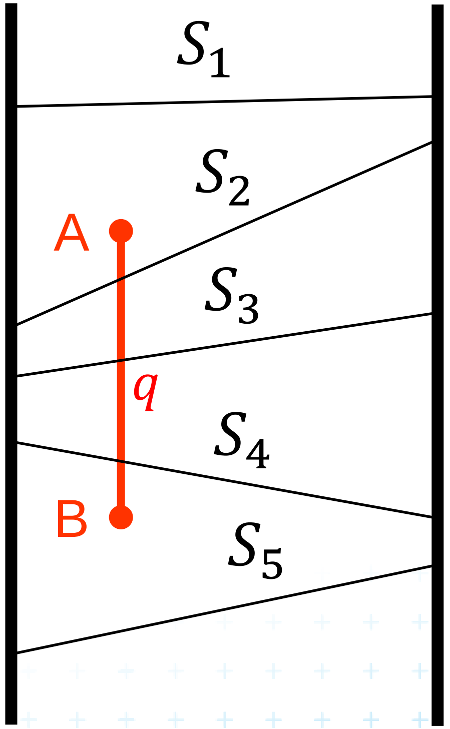


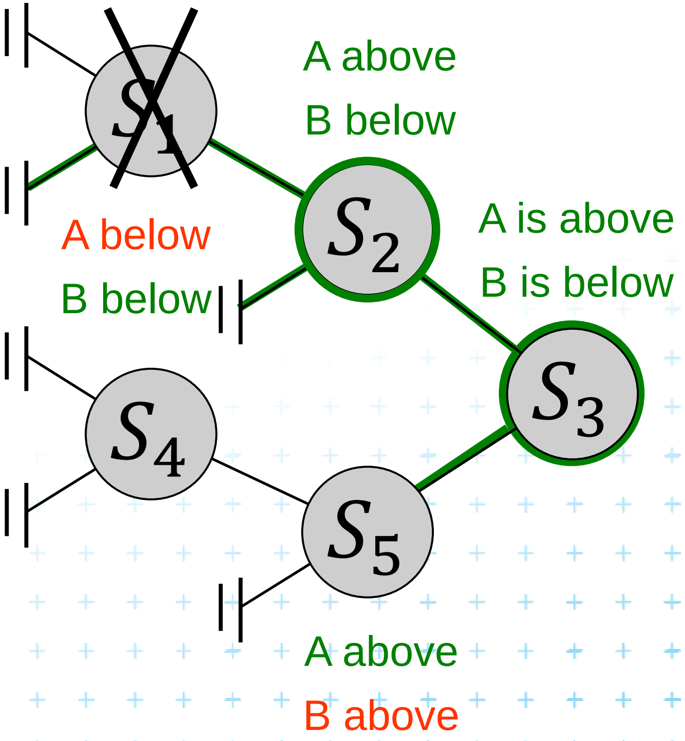
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



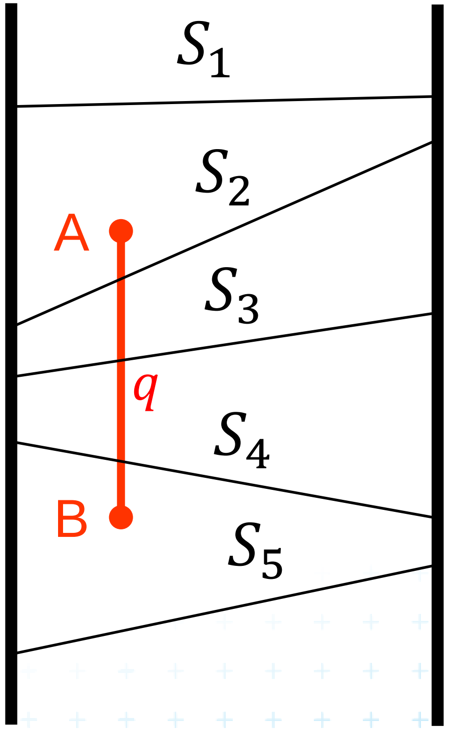


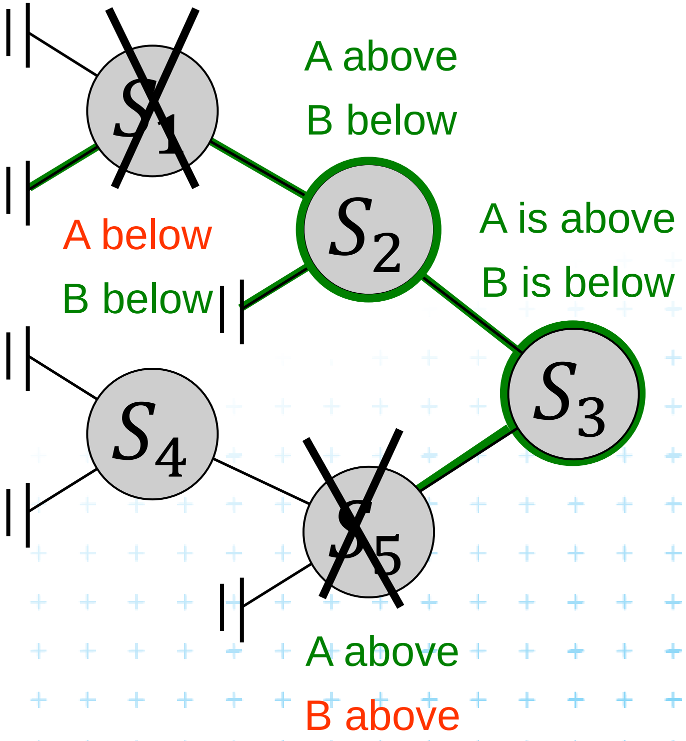
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s





A above
B below

A below
B below

A is above
B is below

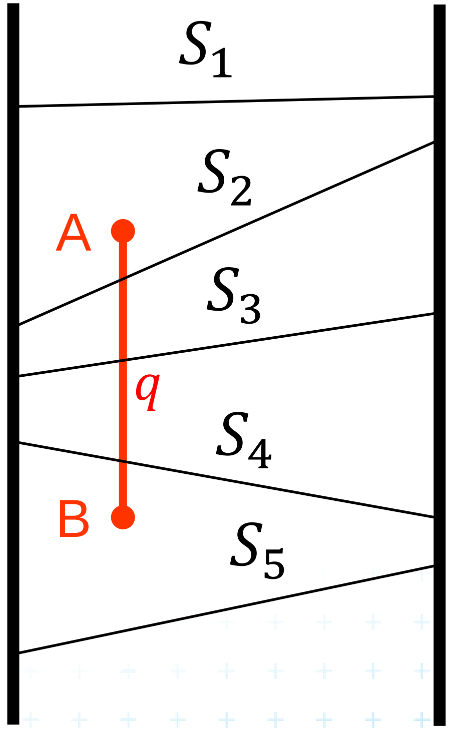
A above
B above

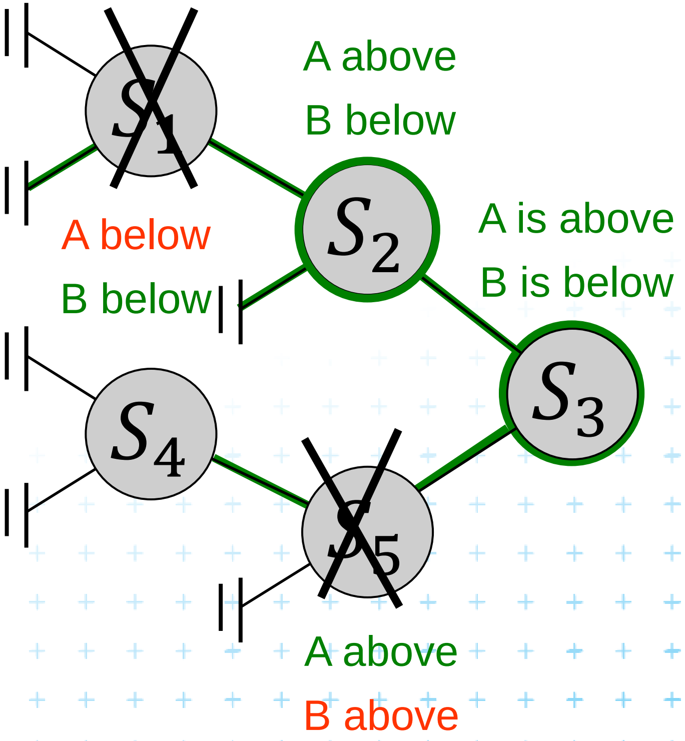
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



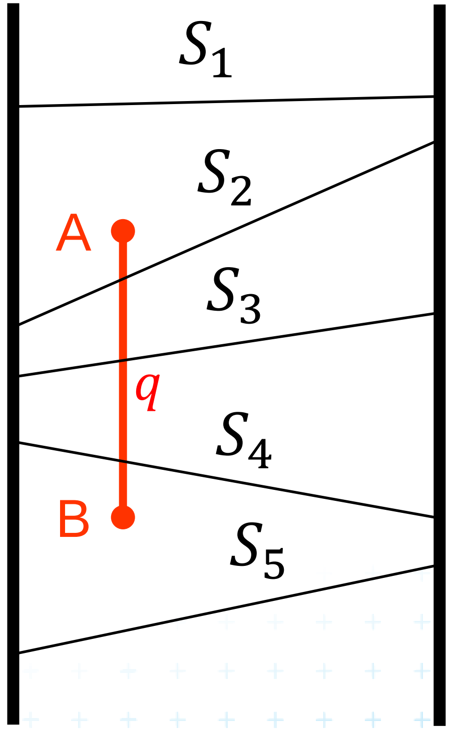


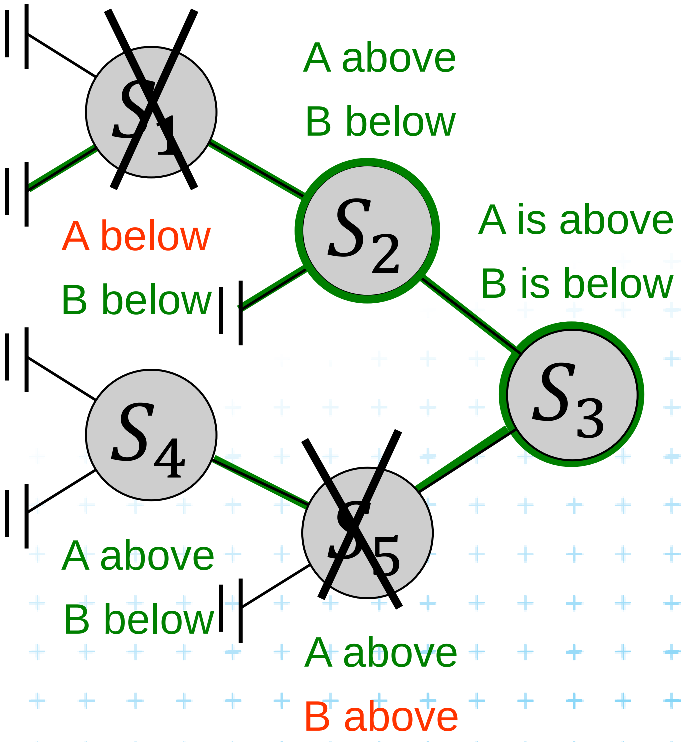
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



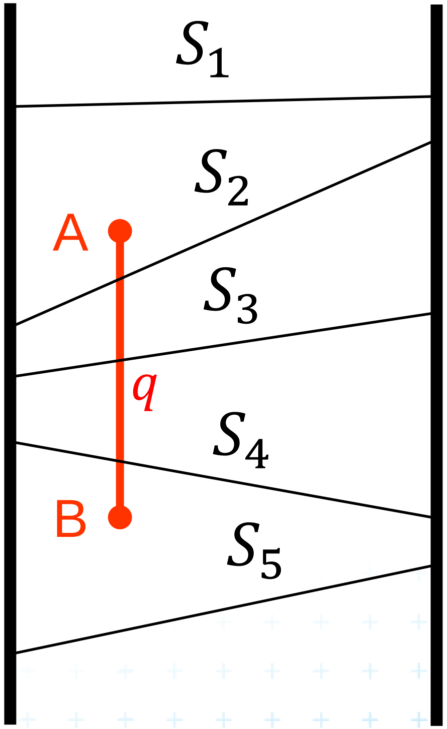


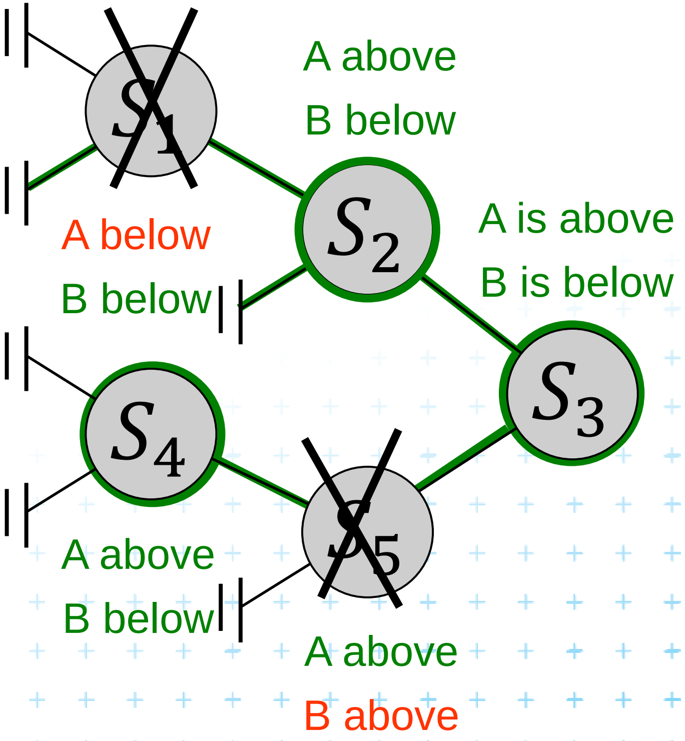
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



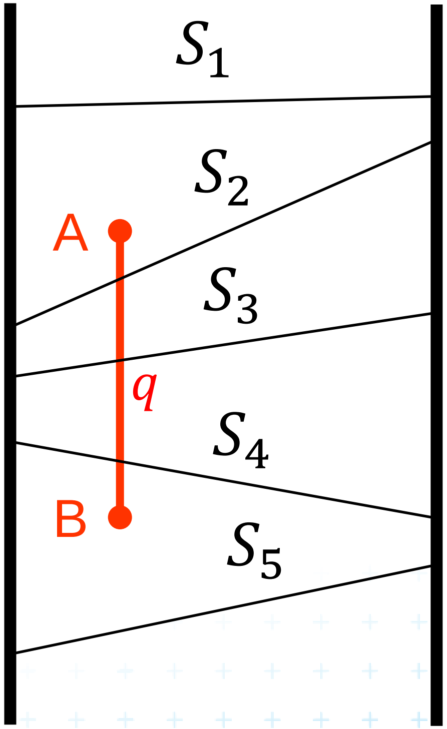


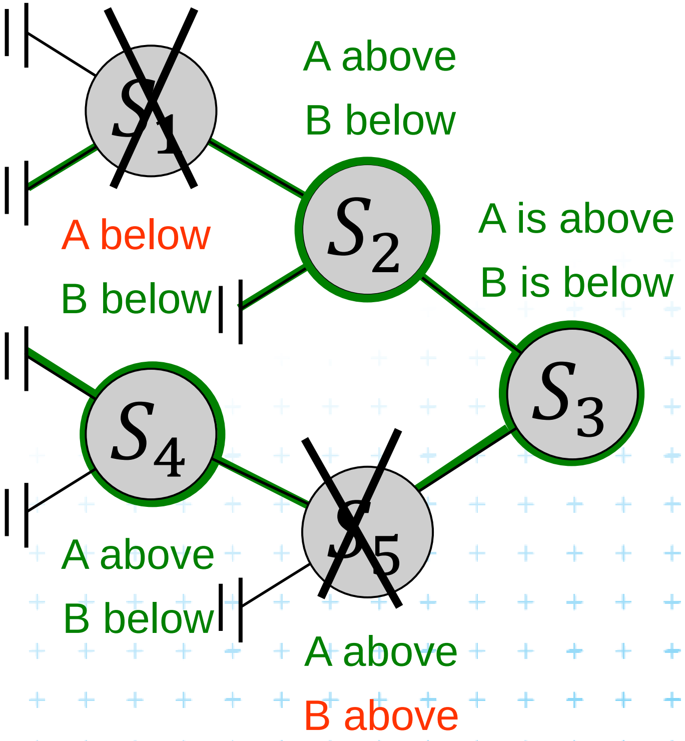
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s



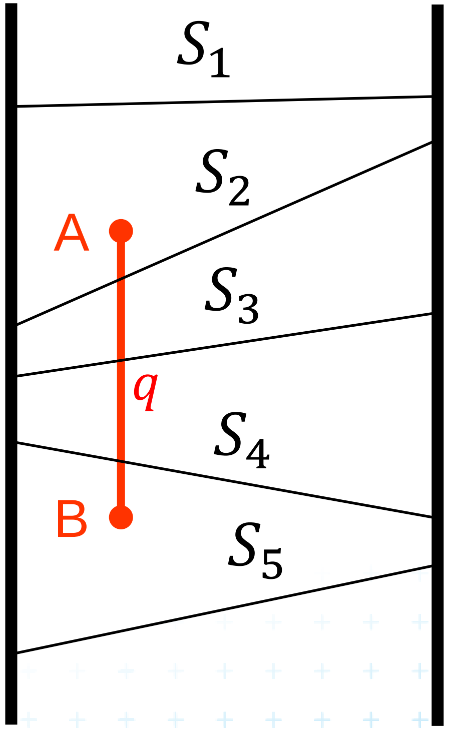


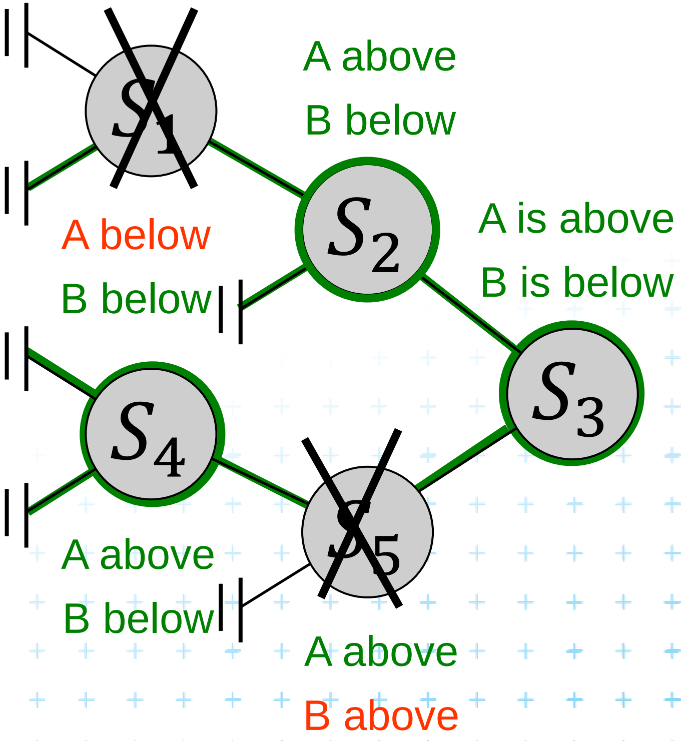
[Berg]

Segments between vertical segment endpoints

Segment s is intersected by vert.query segment q iff

- The lower endpoint (B) of q is below s and
- The upper endpoint (A) of q is above s





A above
B below

A is above
B is below

A below
B below

A above
B below

A above
B above

[Berg]

Segments between vertical segment endpoints

- Segments (in the slab) do not mutually intersect

  => segments can be vertically ordered and stored in BST

  - Each node v of the x segment tree (vertical slab) has an associated y-BST
  - BST $T(v)$ of node v stores the canonical subset $S(v)$ according to the vertical order
  - Intersected segments can be found by searching $T(v)$ in $O(\,k_v + \log n),\; k_v$ is the number of intersected segments

Windowing of arbitrary oriented line segments complexity

Structure associated to node (BST) uses storage linear in the size of $S(v)$

- Build $O(n \log n)$
- Storage $O(n \log n)$
- Query $O(k + \log^2 n)$ … $O(\log n)$ segm tree +$O(\log n)$ BST
  - Report all segments that contain a query point
  - k is number of reported segments

Windowing of line segments in 2D – conclusions

Construction: all interval tree variants $O(n \log n)$

	1. Axis parallel	Search	Memory
1D	i. Line (*sorted lists*)	$O(k + \log n)$	$O(n)$
2D	ii. Segment (*range trees*)	$O(k + \log^2 n)$	$O(n \log n)$
	iii. Segment (*priority s. tr.*)	$O(k + \log n)$	$O(n)$
	2. In general position		
2D	– *segment tree + BST*	$O(k + \log^2 n)$	$O(n \log n)$

References

[Berg] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry: Algorithms and Applications, Springer-Verlag, 3rd rev. ed. 2008. 386 pages, 370 fig. ISBN: 978-3-540-77973-5, Chapters 3 and 9, http://www.cs.uu.nl/geobook/

[Mount] Mount, D.: *Computational Geometry Lecture Notes for Fall 2016*, University of Maryland, Lecture 33.
http://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf

[Rourke] Joseph O´Rourke: Computational Geometry in C, Cambridge University Press, 1993, ISBN 0-521- 44592-2
http://maven.smith.edu/~orourke/books/compgeom.html

[Vigneron] Segment trees and interval trees, presentation, INRA, France,
http://w3.jouy.inra.fr/unites/miaj/public/vigneron/cs4235/slides.html

[Schirra] Stefan Schirra. Geometrische Datenstrukturen. Sommersemester 2009 http://wwwisg.cs.uni-magdeburg.de/ag/lehre/SS2009/GDS/slides/S10.pdf