DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

INTERSECTIONS OF LINE SEGMENTS AND AXIS ALIGNED RECTANGLES, OVERLAY OF SUBDIVISIONS

PETR FELKEL

FEL CTU PRAGUE

felkel@fel.cvut.cz

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

Based on [Berg], [Mount], [Kukral], and [Drtina]

Version from 19.11.2020

Talk overview

- Intersections of line segments (Bentley-Ottmann)
  - Motivation
  - Sweep line algorithm recapitulation
  - Sweep line intersections of line segments
- Intersection of polygons or planar subdivisions
  - See assignment [21] or [Berg, Section 2.3]
- Intersection of axis parallel rectangles
  - See assignment [26]

Geometric intersections – what are they for?

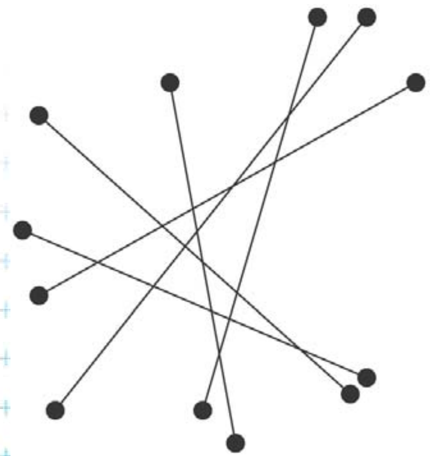
One of the most basic problems in computational geometry

- Solid modeling
  - Intersection of object boundaries in CSG
- Overlay of subdivisions, e.g. layers in GIS
  - Bridges on intersections of roads and rivers
  - Maintenance responsibilities (road network X county boundaries)
- Robotics
  - Collision detection and collision avoidance
- Computer graphics
  - Rendering via ray shooting (intersection of the ray with objects)
- …

Line segment intersection

Line segment intersection

- Intersection of complex shapes is often reduced to simpler and simpler intersection problems
- Line segment intersection is the most basic intersection algorithm
- Problem statement:
  Given *n* line segments in the plane, report all points where a pair of line segments intersect.
- Problem complexity
  - Worst case – $I = O(n^2)$ intersections
  - Practical case – only some intersections
  - Use an output sensitive algorithm
    - $O(n \log n + I)$ optimal randomized algorithm
    - $O(n \log n + I \log n)$ sweep line algorithm - %



[Berg]

Plane sweep line algorithm recapitulation

Postupový plán

- Horizontal line (sweep line, *scan line*) ℓ moves top-down (or vertical line: left to right) over the set of objects
- The move is not continuous, but ℓ jumps from one event point to another
  - Event points are in priority queue or sorted list (~y)
  - The (left) top-most event point is removed first
  - New event points may be created (usually as interaction of neighbors on the sweep line) and inserted into the queue

Status

- Scan-line status
  - Stores information about the objects intersected by ℓ
  - It is updated while stopping on event point

Line segment intersection - Sweep line alg.

- Avoid testing of pairs of segments far apart
- Compute intersections of neighbors on the sweep line only
- $O(n \log n + I \log n)$ time in $O(n)$ memory
  - $2n$ steps for end points,
  - I steps for intersections,
  - $\log n$ search the status tree
- Ignore “degenerate cases” (most of them will be solved later on)
  - No segment is parallel to the sweep line
  - Segments intersect in one point and do not overlap
  - No three segments meet in a common point

Line segment intersections

Status = ordered sequence of segments intersecting the sweep line ℓ — *Stav*

Events (waiting in the priority queue) — *Postupový plán*

= points, where the algorithm actually does something

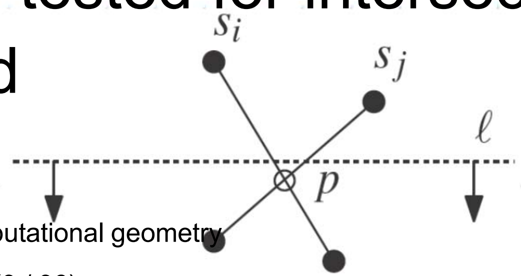
- Segment *end-points*
  - known at algorithm start
- Segment *intersections* between neighboring segments along SL
  - discovered as the sweep executes

Detecting intersections

- Intersection events must be detected and inserted to the event queue before they occur
- Given two segments *a, b* intersecting in point *p*, there must be a placement of sweep line ℓ prior to *p*, such that segments *a, b* are adjacent along ℓ (only adjacent will be tested for intersection)
  - segments *a, b* are not adjacent when the alg. starts
  - segments *a, b* are adjacent just before *p*

  => there must be an event point when *a,b* become adjacent and therefore are tested for intersection

  => All intersections are found

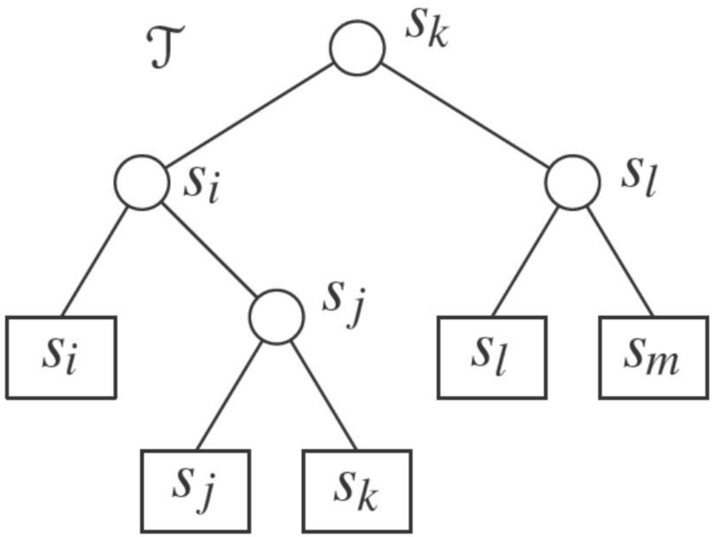


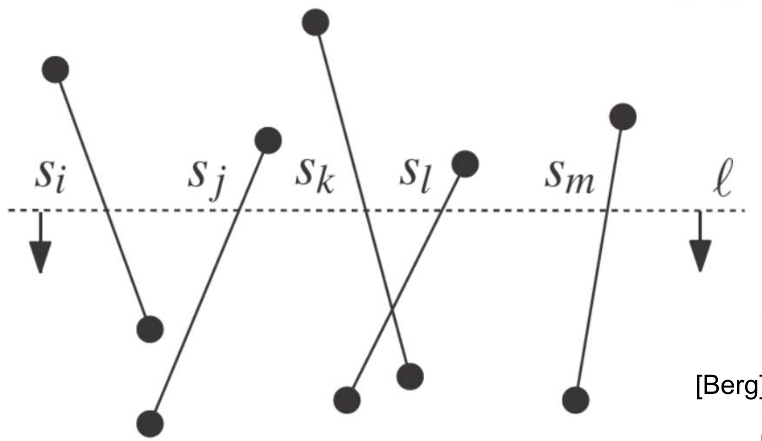
[Berg]

Data structures

Sweep line ℓ status = order of segments along ℓ

- Balanced binary search tree of segments
- Coords of intersections with ℓ vary as ℓ moves
  => store pointers to line segments in tree nodes
  - Position of ℓ is plugged in the $y=mx+b$ to get the x-key

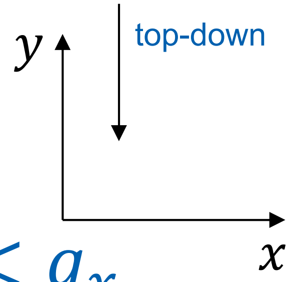




[Berg]

Data structures

Event queue (*postupový plán, časový plán*)



- Define: Order $\succ$ (top-down, lexicographic)

  $p \succ q$ **iff** $p_y > q_y$ **or** $p_y = q_y$ and $p_x < q_x$

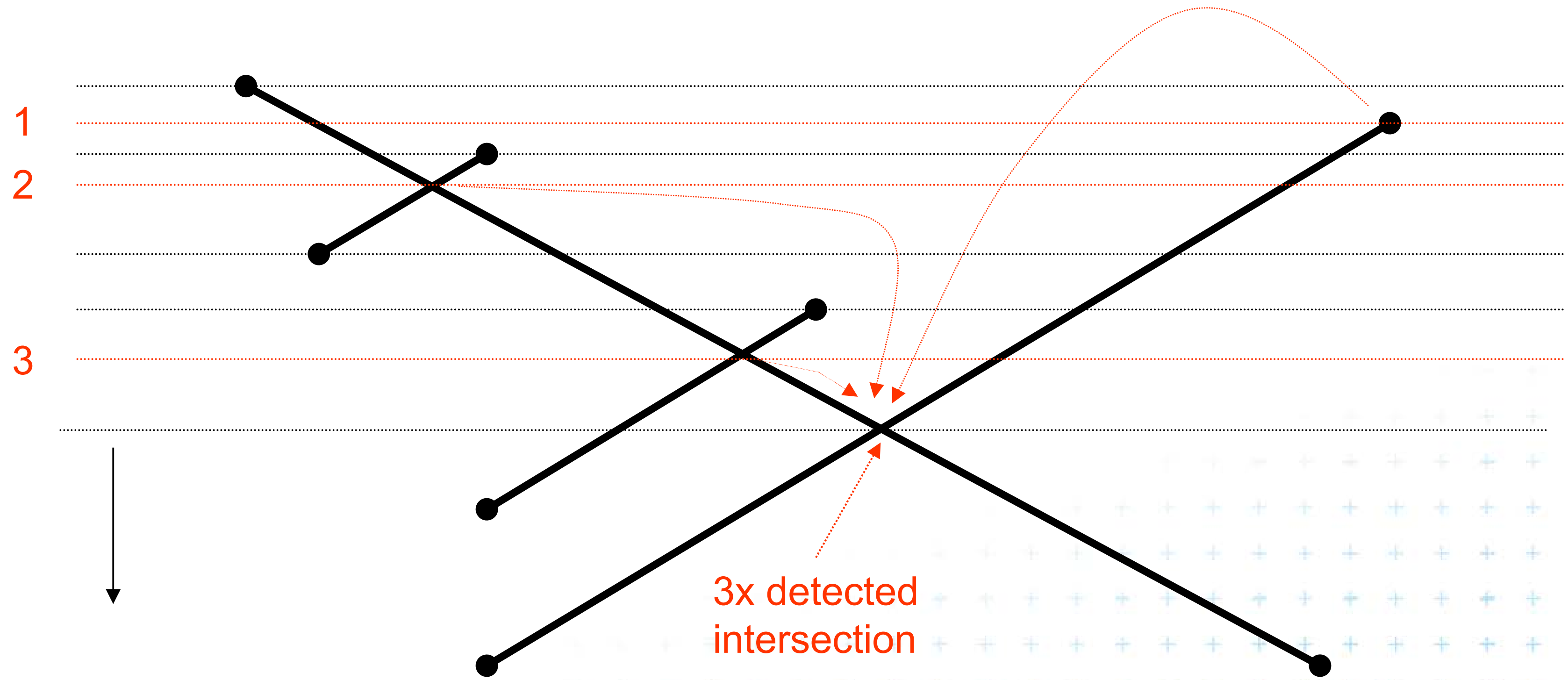
  top-down, left-right approach

  (points on ℓ treated left to right)

- Operations
  - Insertion of computed intersection points
  - Fetching the next event (highest y below ℓ or the leftmost right of e) — must have
  - Test, if the segment is already present in the queue (Locate and delete intersection event in the queue) — may have

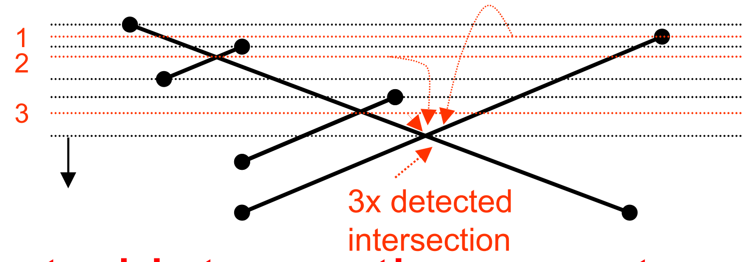
Problem with duplicities of intersections

Intersection may be detected many times



Data structures

Event queue data structure



a) Heap

- Problem: can not check duplicated intersection events (reinvented & stored more than once)
- Intersections processed twice or even more times
- Memory complexity up to O(n^2)

b) Ordered dictionary (balanced binary tree)

- Can check duplicated events (adds just constant factor)
- Nothing inserted twice
- If non-neighbor intersections are deleted
  i.e., if only intersections of neighbors along ℓ are stored
  then memory complexity just O(n)

Line segment intersection algorithm

top-down

FindIntersections(*S*)

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

Output: The set of intersection points + pointers to segments in each

1. init an empty event queue *Q* and insert the segment endpoints
2. init an empty status structure *T*
3. **while** Q in not empty
4.   remove next event *p* from *Q*
5.   handleEventPoint(*p*)

→ Upper endpoint

→ Intersection

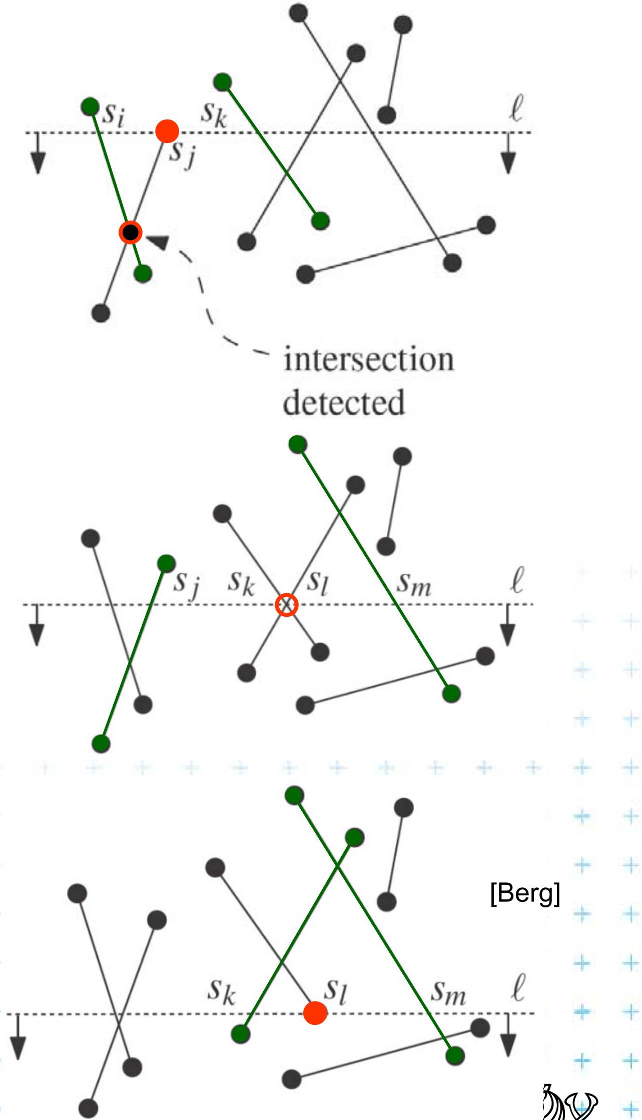
→ Lower endpoint

Improved algorithm:
Handles all in *p*
in a single step

Note: Upper-endpoint events store info about the segment

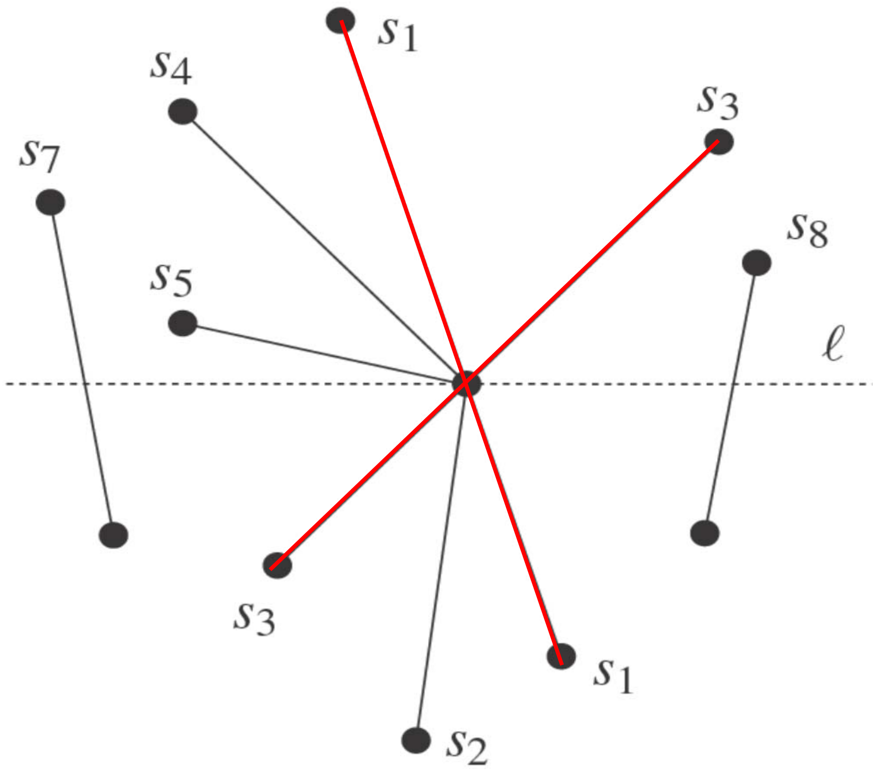
handleEventPoint() principle

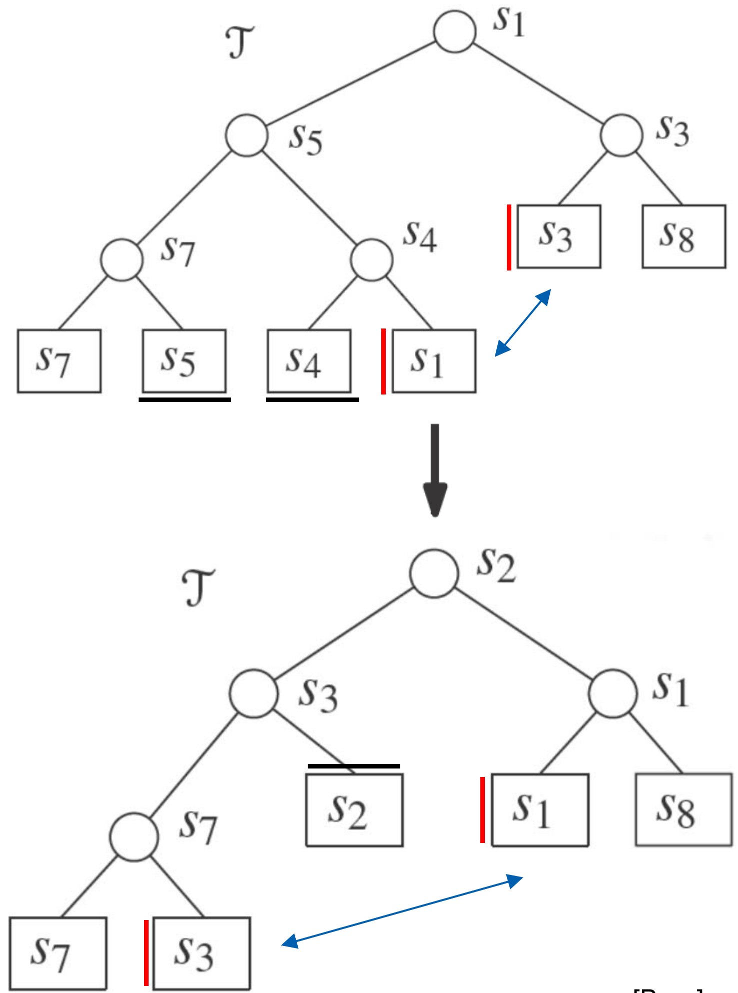
- Upper endpoint $U(p)$
  - insert p (on s_j) to status T
  - add intersections with left and right neighbors to Q
- Intersection $C(p)$
  - switch order of segments in T
  - add intersections with nearest left and nearest right neighbor to Q
- Lower endpoint $L(p)$
  - remove p (on s_l) from T
  - add intersections of left and right neighbors to Q



[Berg]

More than two segments incident





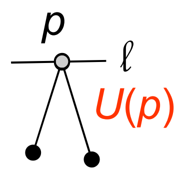
$U(p) = \{s_2\}$ — start here

$C(p) = \{s_1, s_3\}$ — cross on ℓ

$L(p) = \{s_4, s_5\}$ — end here

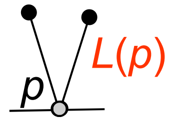
[Berg]

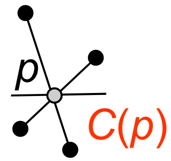
Handle Events [modified Berg, page 25]



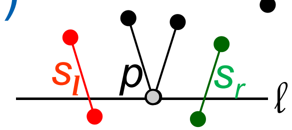
handleEventPoint(*p*) // precisely: handle all events with point *p*

1. Let *U*(*p*) = set of segments whose Upper endpoint is *p*.
   These segments are stored with the event point *p* (will be added to *T*)
2. Search *T* for all segments *S*(*p*) that contain *p* (are adjacent in *T*):
   Let *L*(*p*) ∪ *S*(*p*) = segments whose Lower endpoint is *p*
   Let *C*(*p*) ∪ *S*(*p*) = segments that Contain *p* in interior

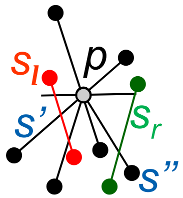




3. **if**(*L*(*p*) ∪ *U*(*p*) ∪ *C*(*p*) contains more than one segment)
4. report *p* as intersection ∘ together with *L*(*p*), *U*(*p*), *C*(*p*)
5. Delete the segments in *L*(*p*) ∪ *C*(*p*) from *T*



6. **if**(*U*(*p*) ∪ *C*(*p*) = ∅) then findNewEvent(s_l, s_r, *p*) // left & right neighbors
7. **else** Insert the segments in *U*(*p*) ∪ *C*(*p*) into *T* // reverse order of *C*(*p*) in *T*
   (order as below ℓ, horizontal segment as the last)
8. s' = leftmost segm. of *U*(*p*) ∪ *C*(*p*); findNewEvent(s_l, *s'*, *p*)
9. s'' = rightmost segm. of *U*(*p*) ∪ *C*(*p*); findNewEvent(*s''*, s_r, *p*)



Detection of new intersections

findNewEvent(s_l, s_r, p) ***// with handling of horizontal segments***

Input: two segments (left & right from p in T) and a current event point p

Output: updated event queue Q with new intersection ●

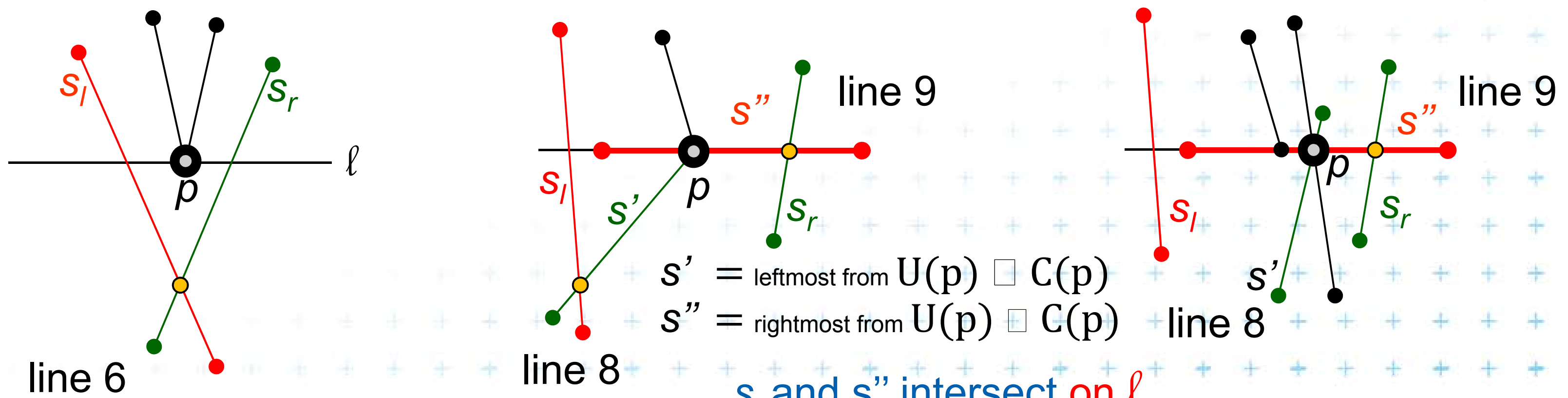
1. **if [(** s_l and s_r intersect below the sweep line ℓ) // intersection below ℓ

   or (s_r intersect s'' on ℓ and to the right of p)] // Non-overlapping horizontal segment

   and(the intersection ● is not present in Q)
2. **then**

   insert intersection ● as a new event into Q

○	Reported intersection - line 4
●	New intersection to Q - line 6,8,9



s' = leftmost from U(p) ∪ C(p)

s'' = rightmost from U(p) ∪ C(p)

s_l and s_r intersect below

s_r and s'' intersect on ℓ,
s'' is horizontal and to the right of p

DCGI

Line segment intersections

- Memory $O(I) = O(n^2)$ with duplicities in Q
  or O(n) with duplicities in Q deleted
- Operational complexity
  - $n + I$ stops
  - $\log n$ each

  => $O(I + n) \log n$ total
- The algorithm is by Bentley-Ottmann

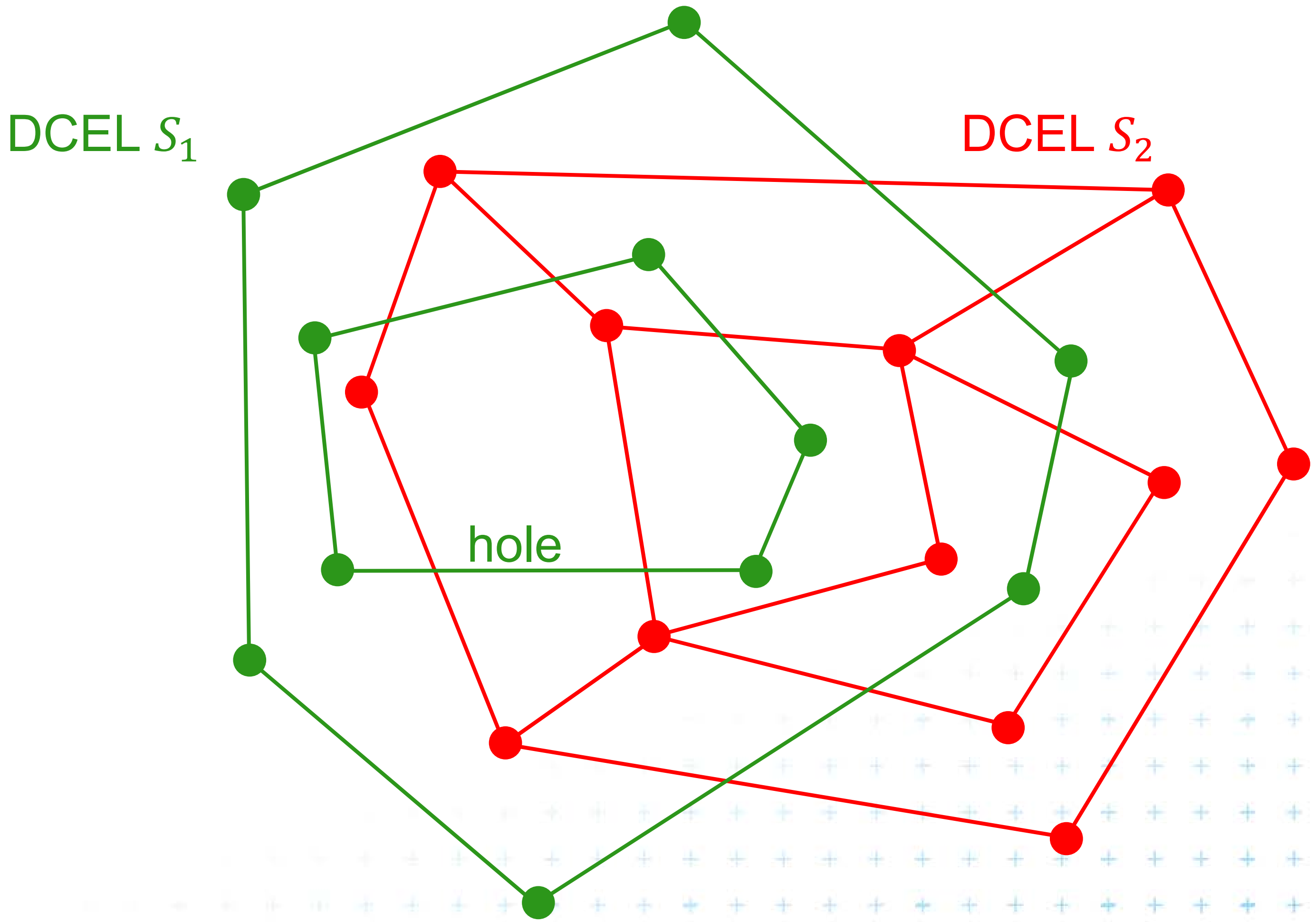
Bentley, J. L.; Ottmann, T. A. (1979), "Algorithms for reporting and counting geometric intersections", *IEEE Transactions on Computers* **C-28** (9): 643-647, doi:10.1109/TC.1979.1675432 .

See also http://wapedia.mobi/en/Bentley%E2%80%93Ottmann_algorithm

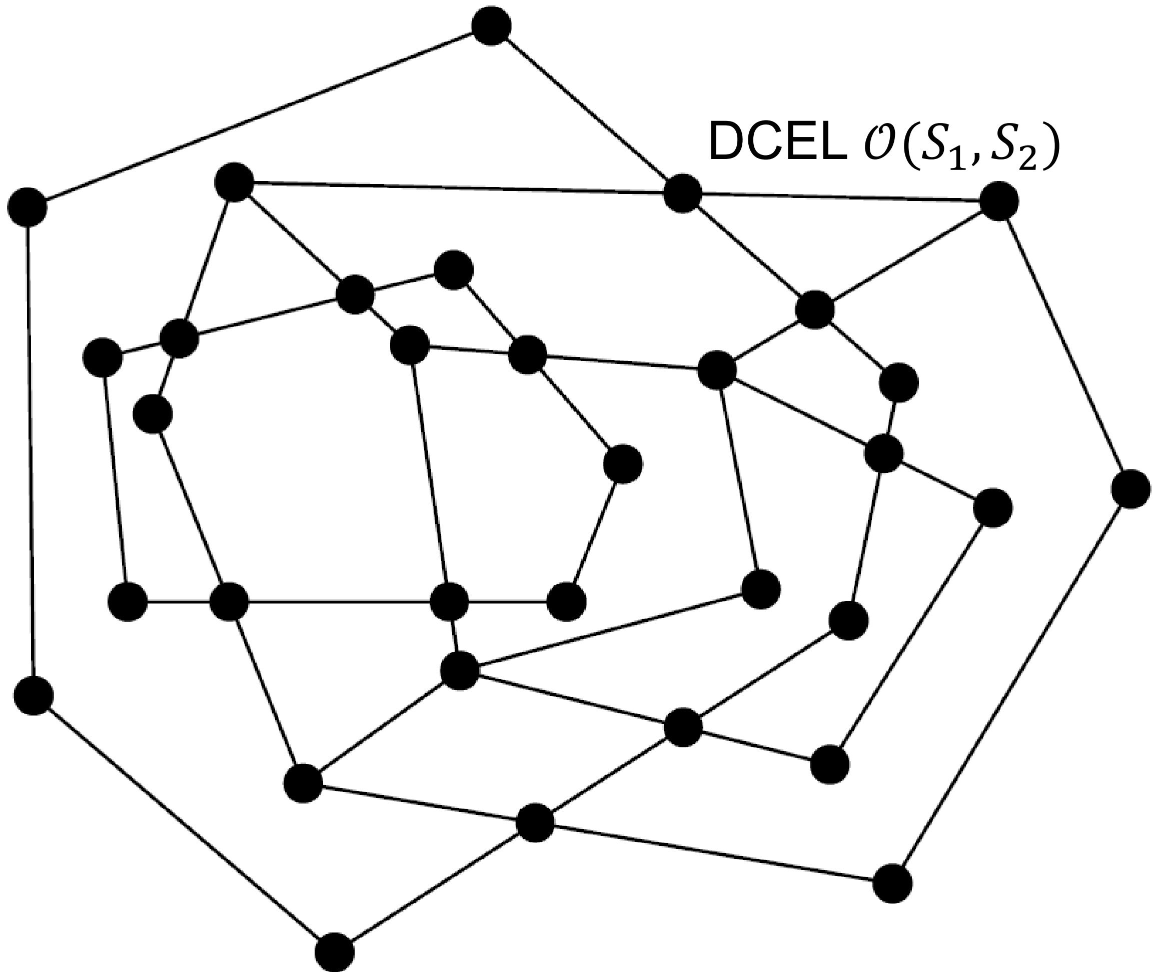
Overlay of two subdivisions

(intersection of DCELs)

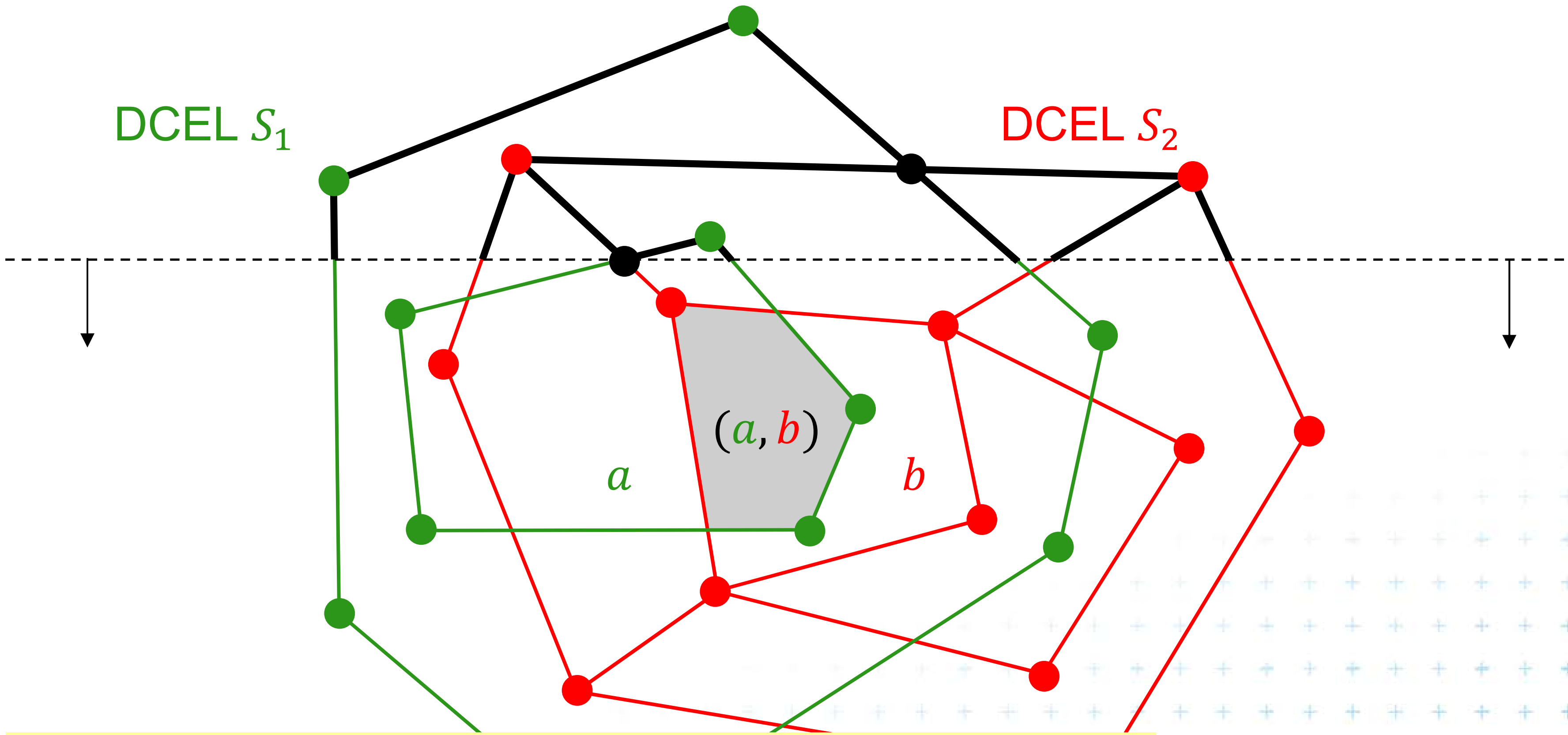
Overlay of two subdivisions



Overlay is a new planar subdivision



Sweep line overlay algorithm



Compute new planar subdivision

- Re-use not intersected half-edge records
- Compute intersections and new half-edge records
- Compute labels of new faces

The algorithm principle

Copy DCELs of both subdivisions to invalid DCEL $\mathcal{D}$

Transform the result into a valid DCEL for the subdivision overlay $\mathcal{O}(S_1, S_2)$

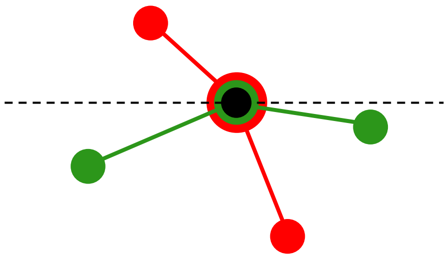
- Compute the intersection of edges (from different subdivisions $S_1 \cap S_2$)
- Link together appropriate parts of the two DCELs
  - Vertex and half-edge records
  - Face records

At an Event point

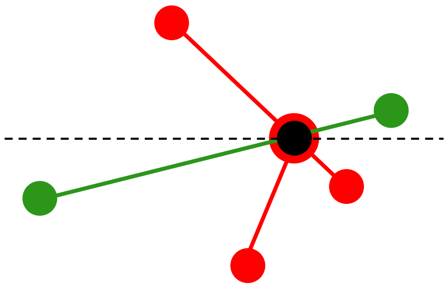
- Update queue Q (pop, delete intersections of separated edges below) and sweep line status tree $\mathcal{T}$ (add/remove/swap edges, compute intersections with neighbors) as in line segment intersection algorithm (cross pointers between edges in $\mathcal{T}$ and $\mathcal{D}$ to access part of $\mathcal{D}$ when processing an intersection)
- For vertex from one subdivision
  - No additional work
- For Intersection of edges from different subdivisions
  - Link both DCELs
  - Handle all possible cases

Three types of intersections

New are intersections of different subdivisions

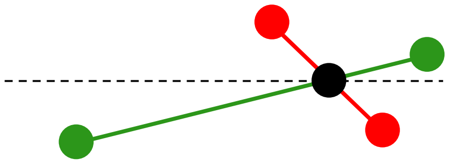


vertex – vertex: overlap of vertices

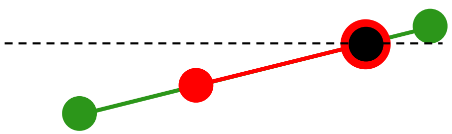


vertex – edge: edge passes through a vertex

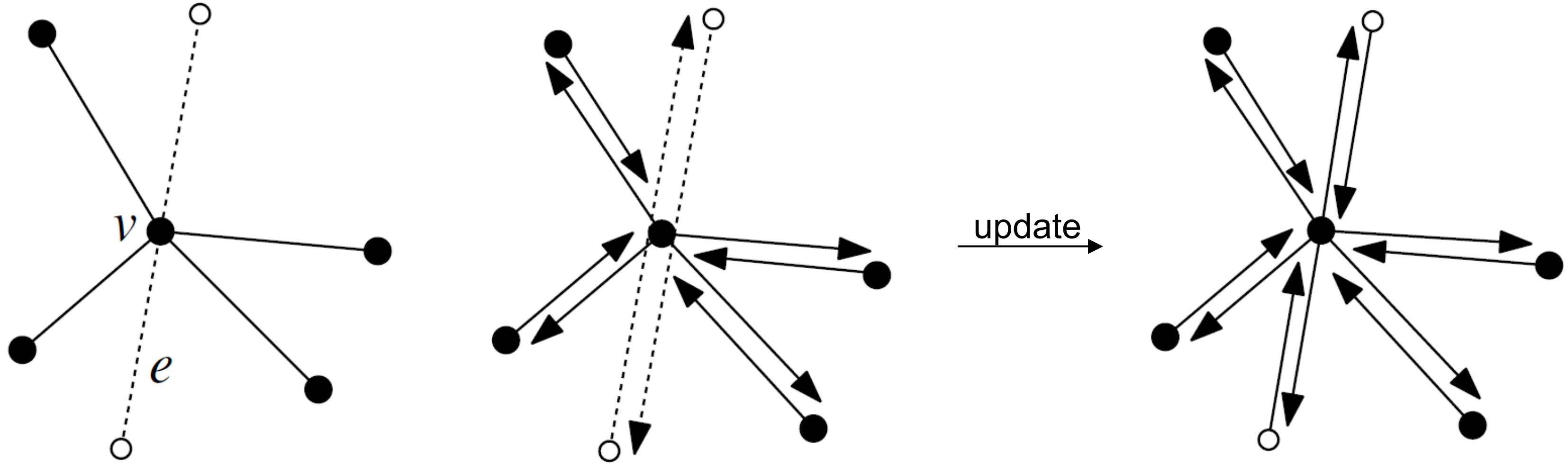
Let's discuss this case,
the other two are similar



edge – edge: edges intersect in their interior



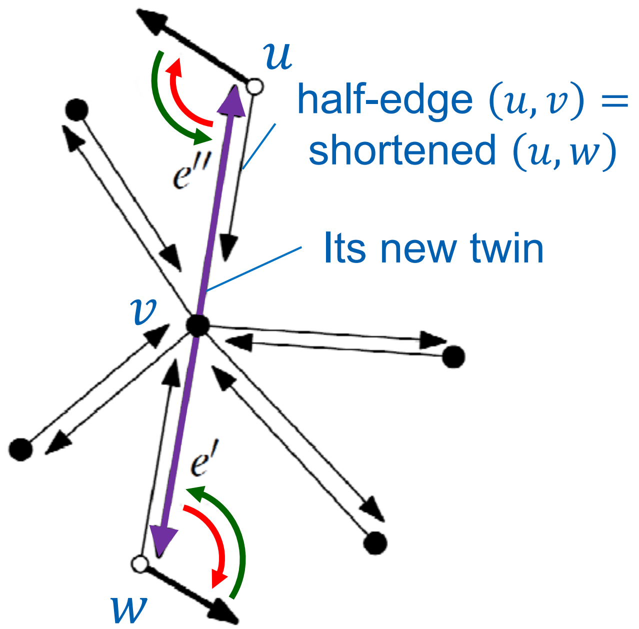
vertex – edge update – the principle



Before:	Before:	After:
The geometry	two half-edges	four half-edges (two shorter and two new)

Pointers around the end-points of edge e

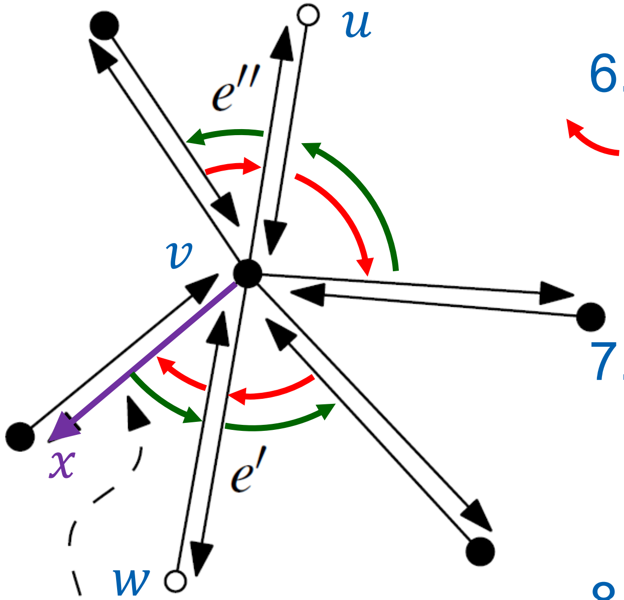
1. Edge $e = (u, w)$ splits into two edges e' and e'' at intersection v



$e' = (w, v)$ $\qquad$ $e'' = (v, u)$

2. Shorten half-edge (w, u) to (w, v)
   Shorten half-edge (u, w) to (u, v)
3. Create their twin (v, w) for (w, v)
   Create their twin (v, u) for (u, v)
4. Set new twin's next to former edge e next
   $\mathrm{next}(v, u) = \mathrm{next}(w, u)$ now in $\mathrm{next}(w, v)$
   $\mathrm{next}(v, w) = \mathrm{next}(u, w)$ now in $\mathrm{next}(u, v)$
5. Set prev pointers to new twins
   $\mathrm{prev}(\mathrm{next}(v, u)) = (v, u)$
   $\mathrm{prev}(\mathrm{next}(v, w)) = (v, w)$

Pointers around intersection v



first CW half-edge from e'

6. Find the next edge x for e' from half-edge (w, v)
   = first CW half-edge from e' with v as origin
   $\mathrm{next}(w, v) = x$
   $\mathrm{prev}(x) = (w, v)$
7. Find the prev edge for e' from half-edge (v, w)
   = first CCW half-edge from e' with v as destination
   $\mathrm{next}, \mathrm{prev}$ similarly
8. Find the next edge for e'' from half-edge (u, v)
   = first CW half-edge from e'' with v as origin
   $\mathrm{next}, \mathrm{prev}$ similarly
9. Find the prev edge for e'' from half-edge (v, u)
   = first CCW half-edge from e' with v as destination
   $\mathrm{next}, \mathrm{prev}$ similarly

Time cost for updating half-edge records

- All operations with splitting of edges in intersections and reconnecting of prev, next pointers take $O(1)$ time
- Locating of edge position in cyclic order
  - around single vertex v takes $O(\deg(v))$
  - which sums to $O(m)$ = number of edges processed by the edge intersection algorithm $= O(n)$
  - The overall complexity is not increased

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

$n = |S_1| + |S_2|$ k = complexity of the overlay (≈intersections)

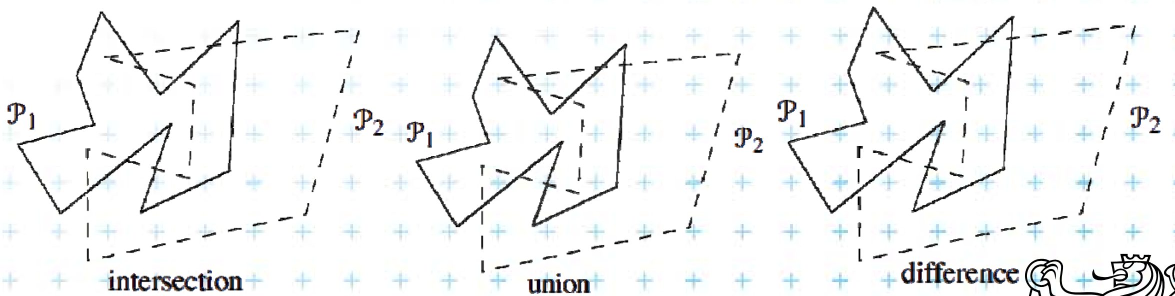
Complexity of input subdivisions

Face records for the overlay subdivision

- Create face records for each face f in $\mathcal{O}(S_1, S_2)$
  - Each face f has it unique outer boundary (CCW) (except the background that has none)
  - Each face has its *OuterComponent*(f) – store edge of it
  - Together faces = #outer boundaries + 1
- *InnerComponents*(f) – list of edges of holes (cw)
- Label of f in S_1
- Label of f in S_2

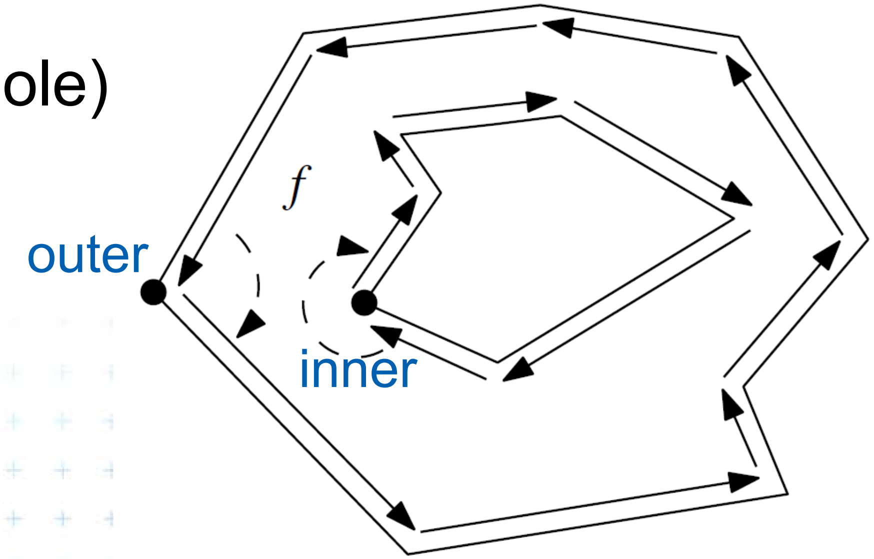
Used for Boolean operations such as $S_1 \cap S_2$, $S_1 \cup S_2$, $S_1 \setminus S_2$

Polygon examples:



Extraction of faces

- Traverse cycles in DCEL (Tarjan alg. DFS) ...*O(n)*
- Decide, if the cycle is outer or inner boundary
  - Find leftmost vertex of the cycle (bottom leftmost)
  - Incident face lies to the left of edges
  - Angle $< 180°$ ⇒ outer
  - Angle $> 180°$ ⇒ inner (hole)

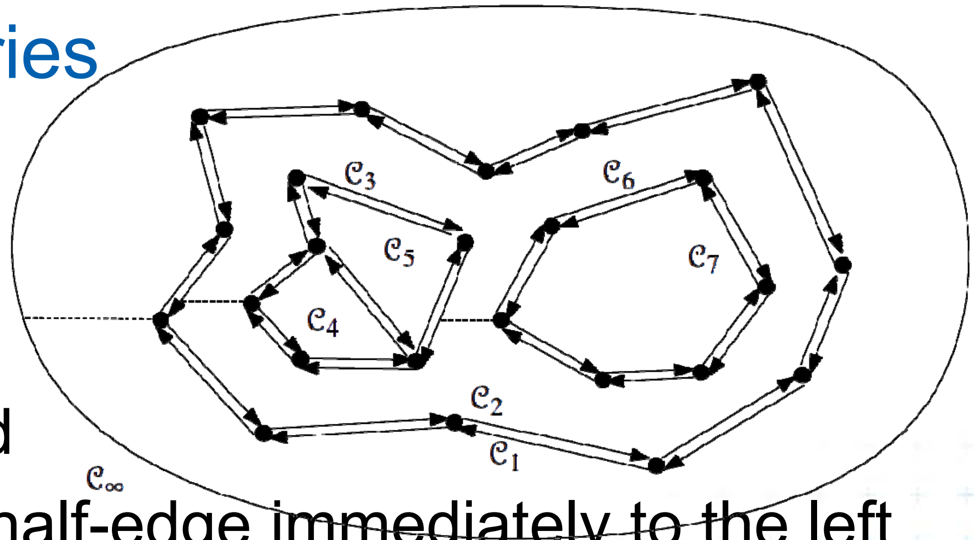


Which boundary cycles bound same face?

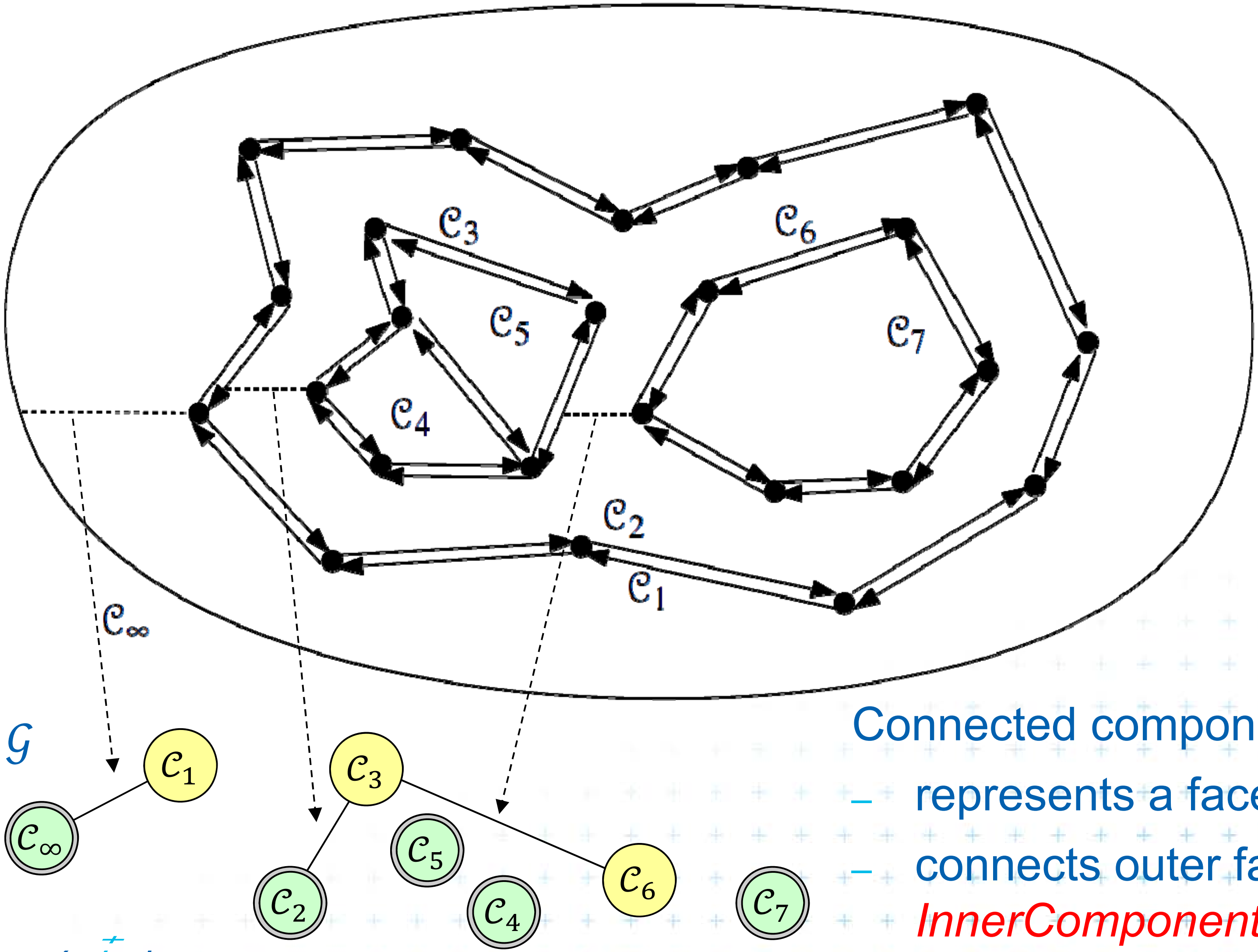
- Single outer boundary shares the face with its holes – inner boundaries
- Graph
  - Node for each cycle

    $\mathcal{C}_3$ inner

    $\mathcal{C}_2$ outer $\mathcal{C}_\infty$ unbounded
  - Arc if inner cycle has half-edge immediately to the left of the leftmost vertex
  - Each connected component – set of cycles of one face



Graph $\mathcal{G}$ of faces and their relations



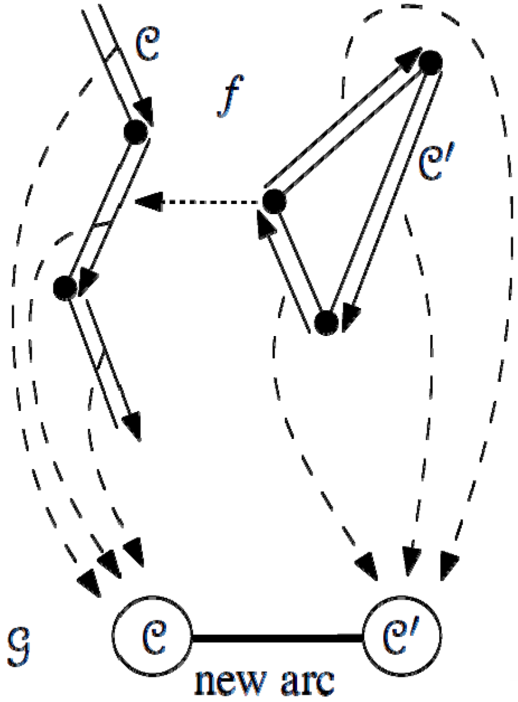
- $\mathcal{C}_3$ inner (cw)
- $\mathcal{C}_2$ outer (ccw)
- $\mathcal{C}_\infty$ unbounded

Connected component in $\mathcal{G}$

- represents a face f
- connects outer face with its holes
  InnerComponents(f)

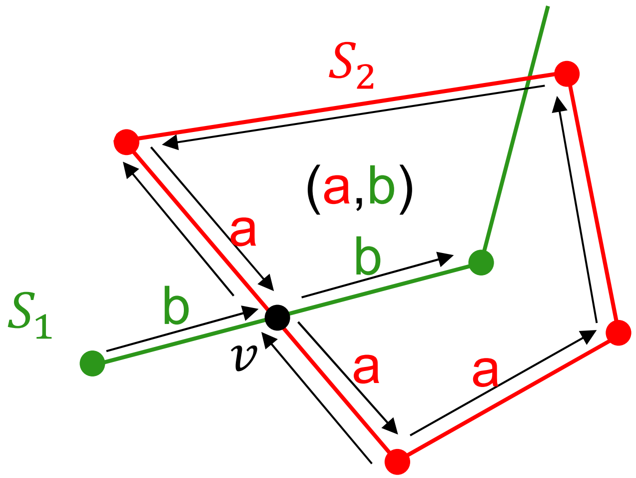
Graph $\mathcal{G}$ construction

Idea – during sweep line, we know the nearest left edge for every vertex v (and half-edge with origin v)



1. Make node for every cycle (graph traversal)
2. During plane sweep,
   – store pointer to graph node for each edge
   – remember the leftmost vertex and its nearest left edge
3. Create arc between cycles of the leftmost vertex an its nearest left edge
4.

Face label determination



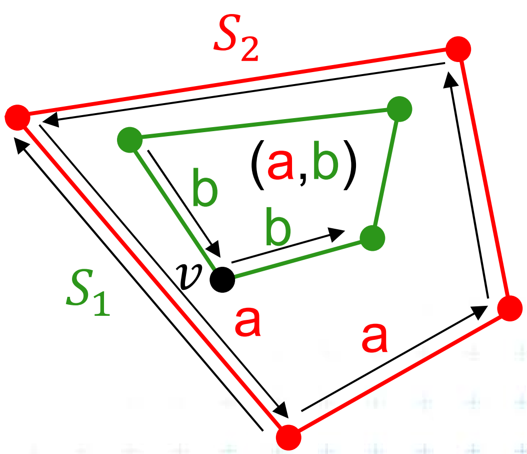
For intersection v of two edges:

During the sweep-line

- In both new pieces, remember the face of half-edge being split into two

After

- Label the face by both labels



For face in other face:

Known half-edge label only from S_1

Use graph $\mathcal{G}$ to locate outer boundary label for face from S_2
(or store containing face f of other subdivision for each vertex)

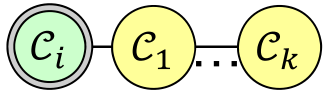
Map overlay algorithm

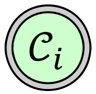
MapOverlay(S_1, S_2)

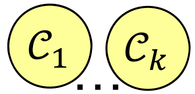
Input: Two planar subdivisions S_1 and S_2 stored in DCEL // complexity n

Output: The overlay of S_1 and S_2 stored in DCEL $\mathcal{D}$

1. Copy both DCELs for of S_1 and S_2 into DCEL $\mathcal{D}$ // $O(n)$
2. Use plane sweep to compute intersections of edges from S_1 and S_2 // $O(n \log n + k \log n)$ (intersection)
   - Update vertex and edge records in $\mathcal{D}$ when the event involves edges of both S_1, S_2
   - Store the half-edge to the left of the event point at the vertex in $\mathcal{D}$
3. Traverse $\mathcal{D}$ (depth-first search) to determine the boundary cycles // $O(n)$
4. Construct the graph $\mathcal{G}$ (boundary and hole cycles, immediately to the left of hole),
5. **for** each connected component in $\mathcal{G}$ **do**



6. $C \leftarrow$ the unique outer boundary cycle
7. $f \leftarrow$ the face bounded by the cycle C.
8. Create a face record for f
9. *OuterComponent*(f) $\leftarrow$ some half-edge of C, 

10. *InnerComponents*(f) $\leftarrow$ list of pointers to one half-edge e in each hole 

11. *IncidentFace*(e) $\leftarrow f$ for all half-edges bounding cycle C and the holes

(lines 5–11) // $O(k)$

12. Label each face of $O(S_1, S_2)$ with the names of the faces of S_1 and S_2 containing it

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

Running time

The overlay of two planar subdivisions with total complexity n can be constructed in

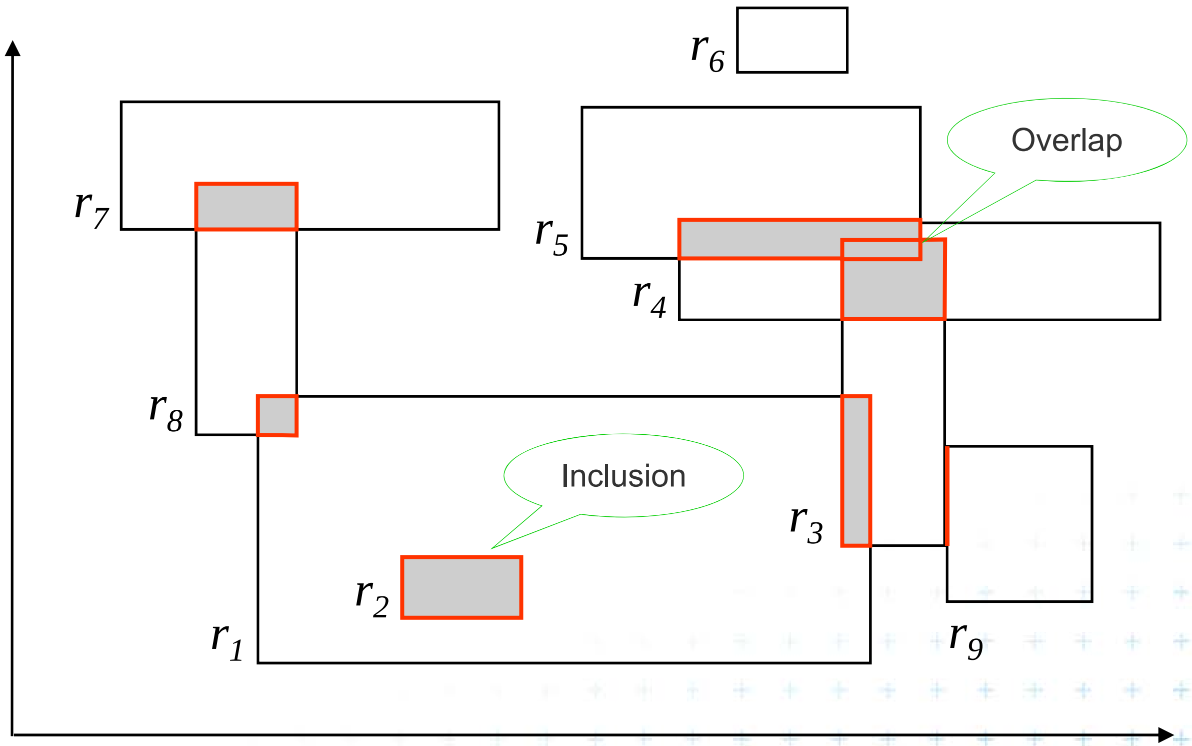
$$O(n \log n + k \log n)$$

where $k =$ complexity of the overlay (≈intersections)

Axis parallel rectangles intersection

Intersection of axis parallel rectangles

- Given the collection of *n isothetic* rectangles, report all intersecting parts



Alternate sides belong to two pencils of lines

(trsy přímek)

(often used with points in infinity = axis parallel)

2D => 2 pencils

[?]

Answer: (r_1, r_2) (r_1, r_3) (r_1, r_8) (r_3, r_4) (r_3, r_5) (r_3, r_9) (r_4, r_5) (r_7, r_8)

Brute force intersection

Brute force algorithm

Input: set S of axis parallel rectangles
Output: pairs of intersected rectangles

1. For every pair (r_i, r_j) of rectangles $\in S, i \neq j$
2. if $(r_i \cap r_j \neq \emptyset)$ then
3. report (r_i, r_j)

Analysis

Preprocessing: None.

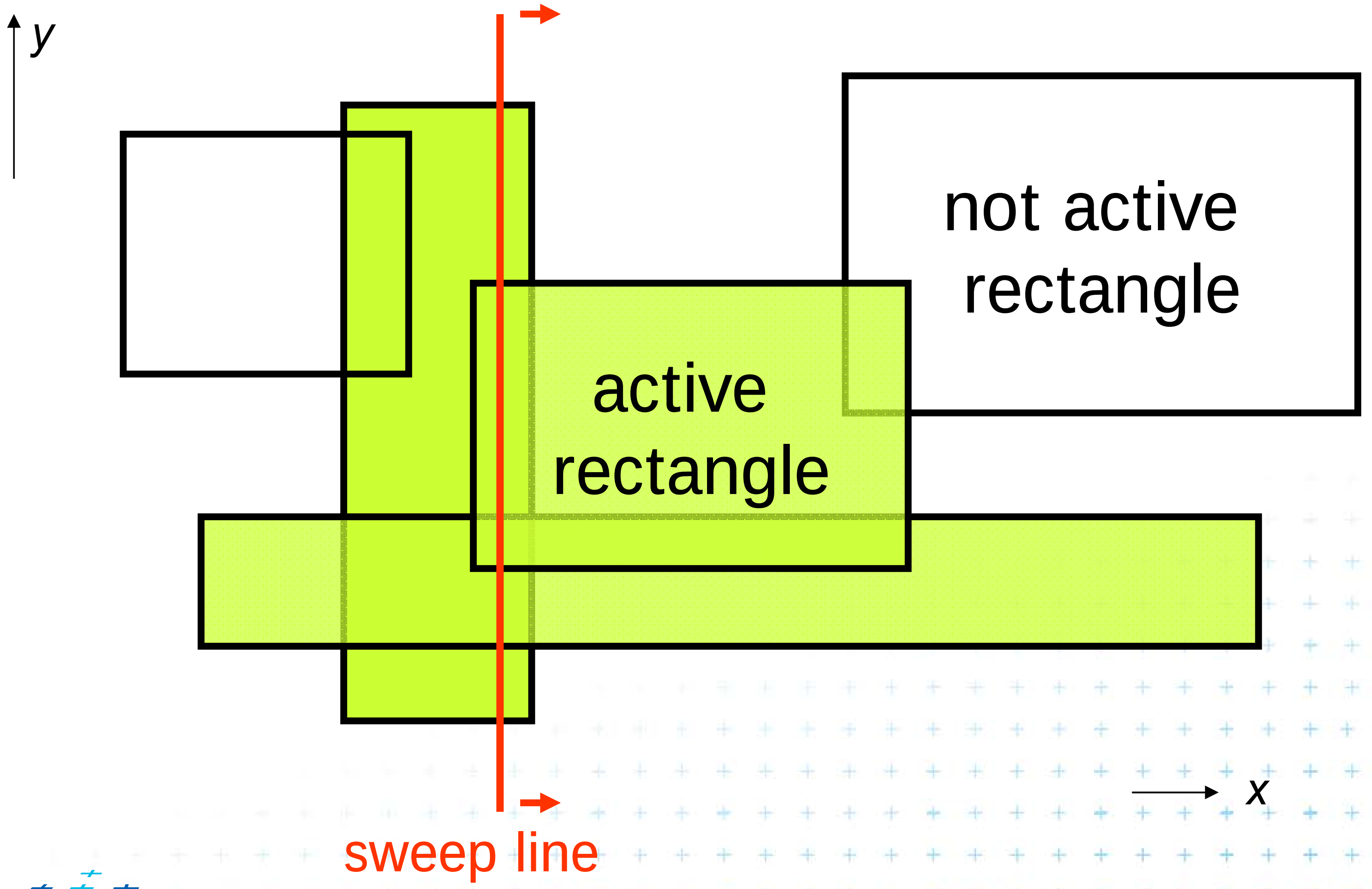
Query: $O(N^2)$ $\binom{N}{2} = \frac{N(N-1)}{2} \in O(N^2)$.

Storage: $O(N)$

Plane sweep intersection algorithm

- Vertical sweep line moves from left to right
- Stops at every x-coordinate of a rectangle (either at its left side or at its right side).
- active rectangles – a set
  = rectangles currently intersecting the sweep line
  - left side event of a rectangle – start
    => the rectangle is added to the active set.
  - right side – end
    => the rectangle is deleted from the active set.
- The active set used to detect rectangle intersection

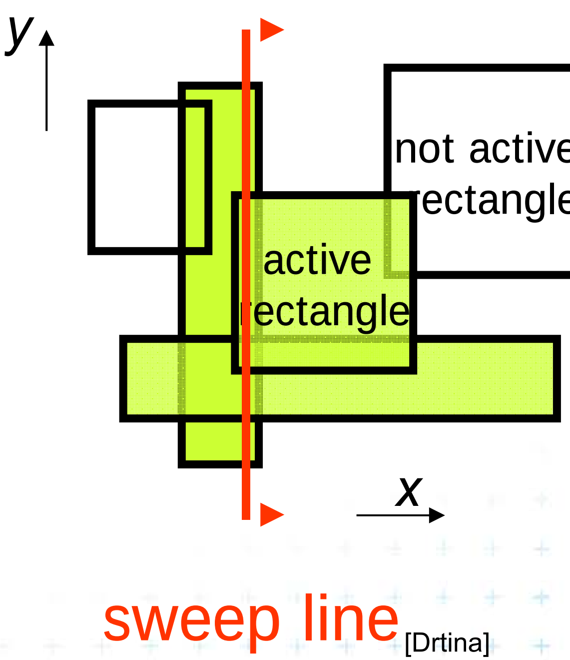
Example rectangles and sweep line

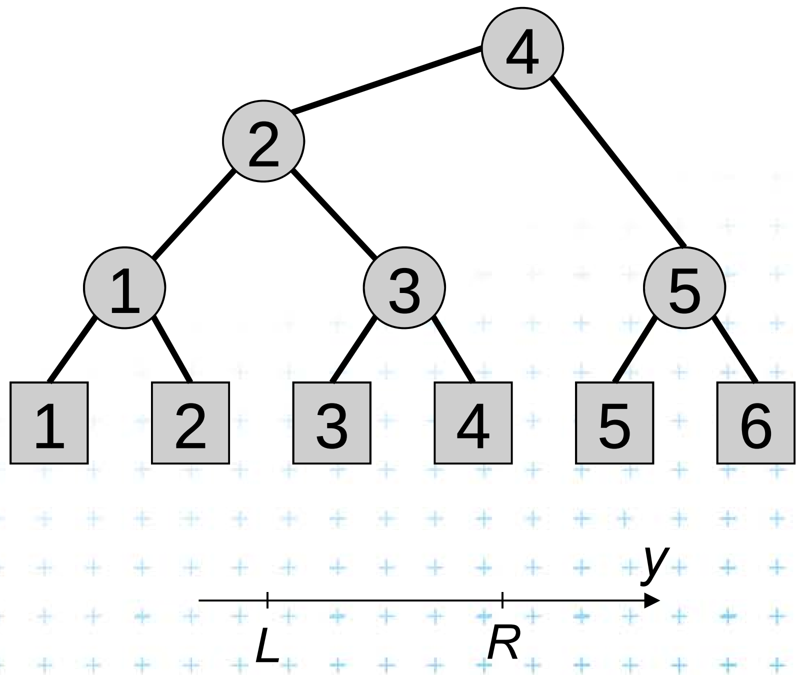


[Drtina]

Interval tree as sweep line status structure

- Vertical sweep-line => only *y*-coordinates along it
- The status tree is drawn horizontal - turn 90° right as if the sweep line (*y*-axis) is horizontal



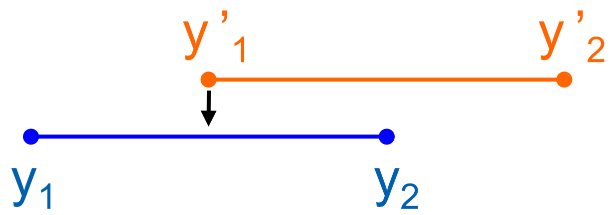


Intersection test – between pair of intervals

- Given two intervals $I = [y_1, y_2]$ and $I' = [y'_1, y'_2]$ the condition $I \cap I'$ is equivalent to one of these mutually exclusive conditions:

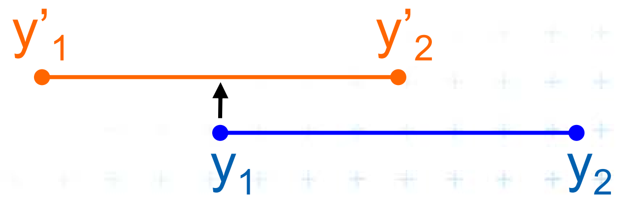
1st variant

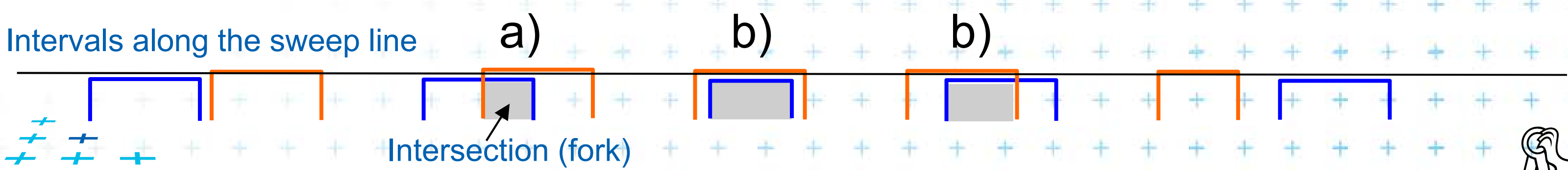
a) $y_1 \le y'_1 \le y_2$



OR

b) $y'_1 \le y_1 \le y'_2$



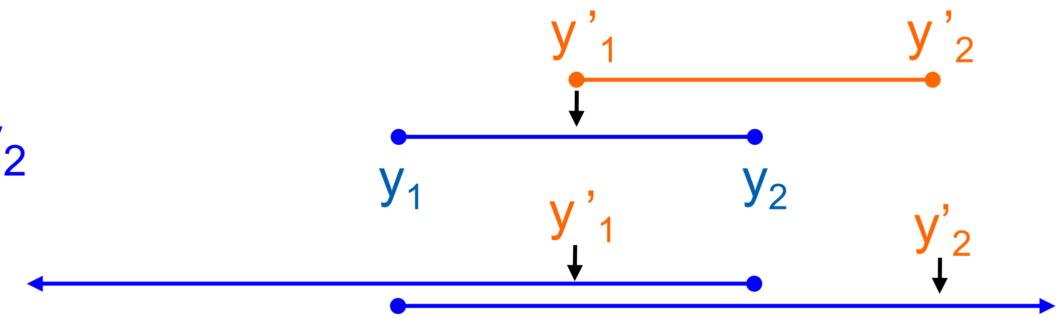


Intersection test – between pair of intervals

- Given two intervals $I = [y_1, y_2]$ and $I' = [y'_1, y'_2]$ the condition $I \cap I'$ is equivalent to both of these conditions simultaneously:

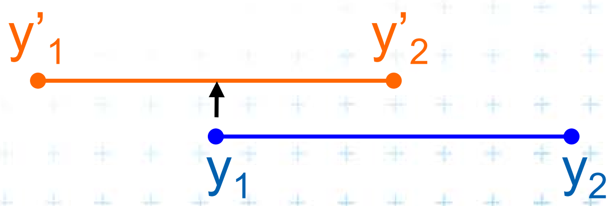
2nd variant

1) $y'_1 \le y_2$

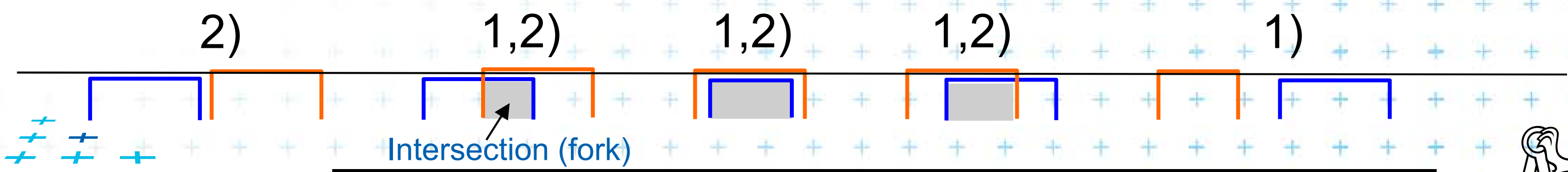


AND

2) $y_1 \le y'_2$

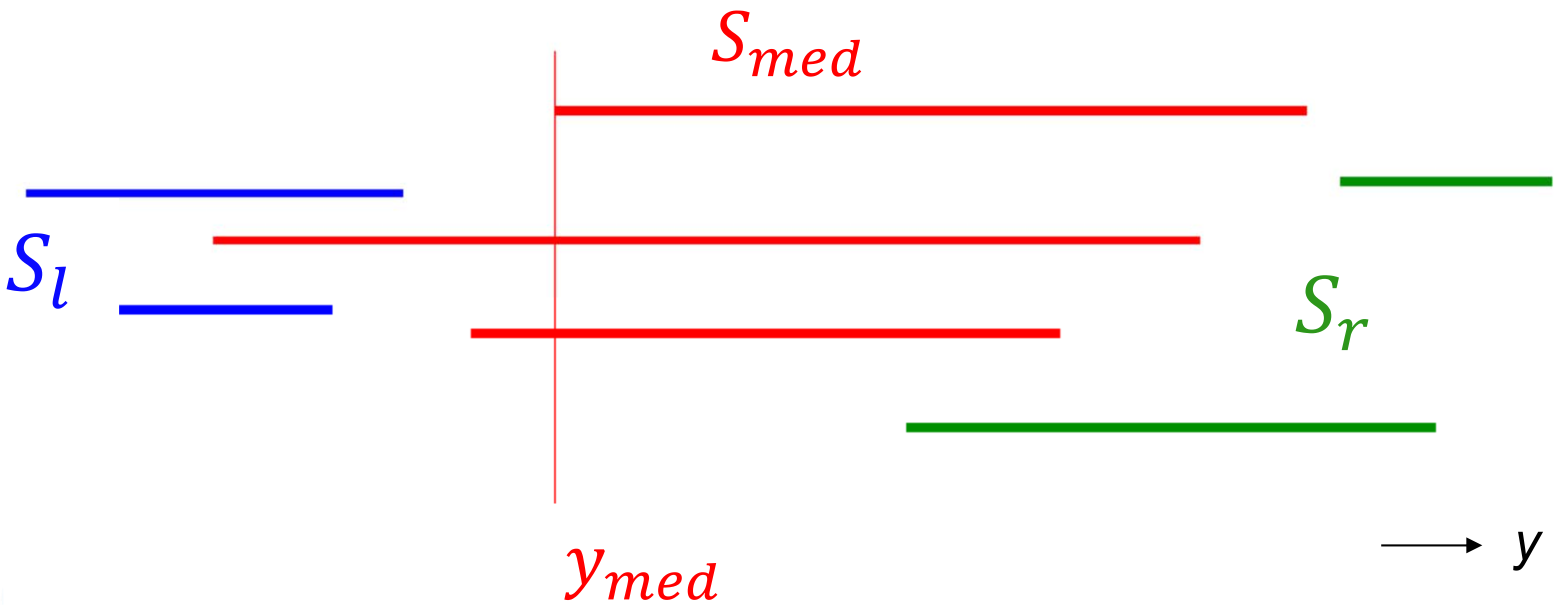


Intervals along the sweep line



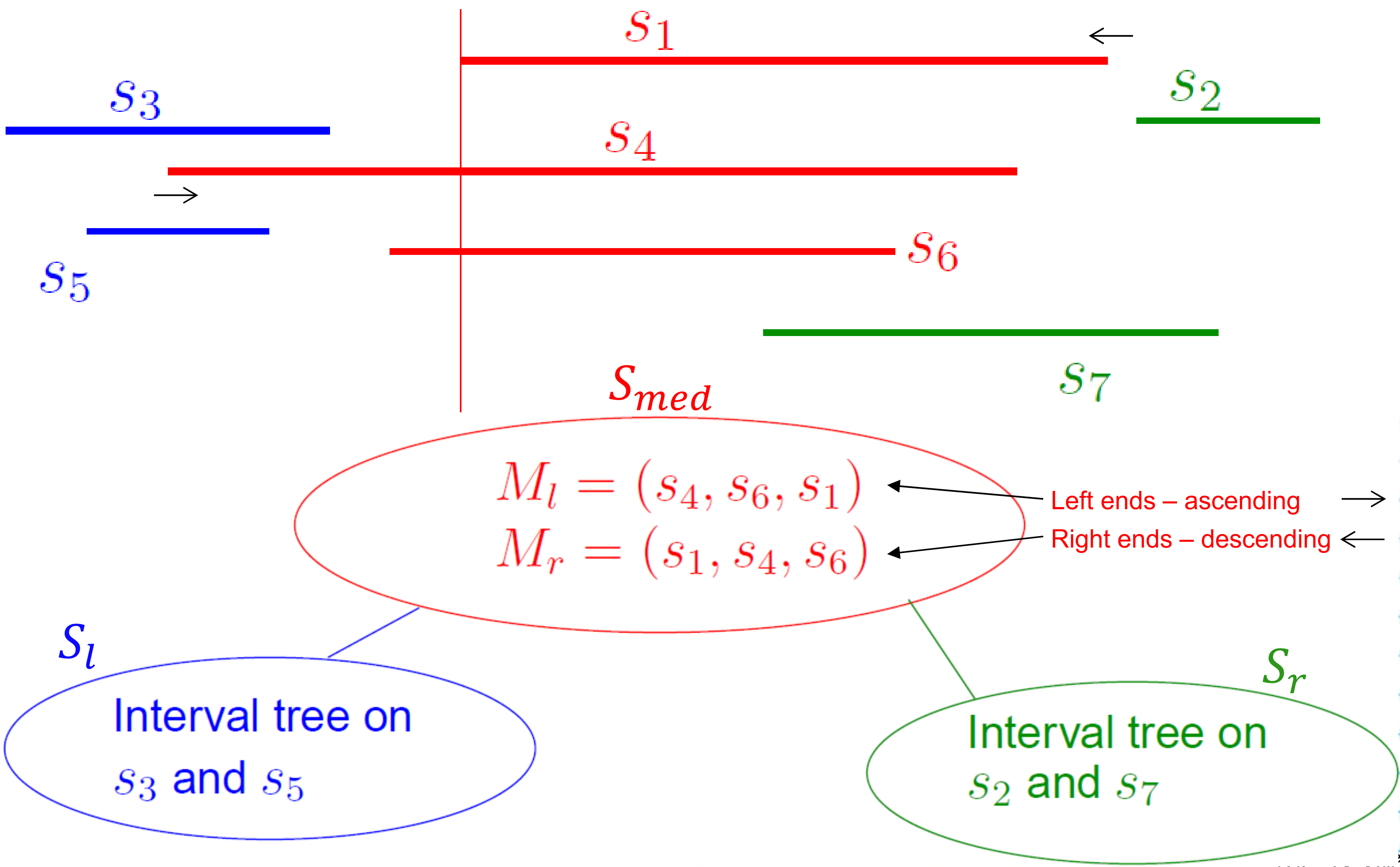
Static interval tree – stores all end point y_s

- Let $v = y_{med}$ be the median of end-points of segments
- S_l : segments of S that are completely to the left of y_{med}
- S_{med}: segments of S that contain y_{med}
- S_r : segments of S that are completely to the right of y_{med}



[Vigneron]

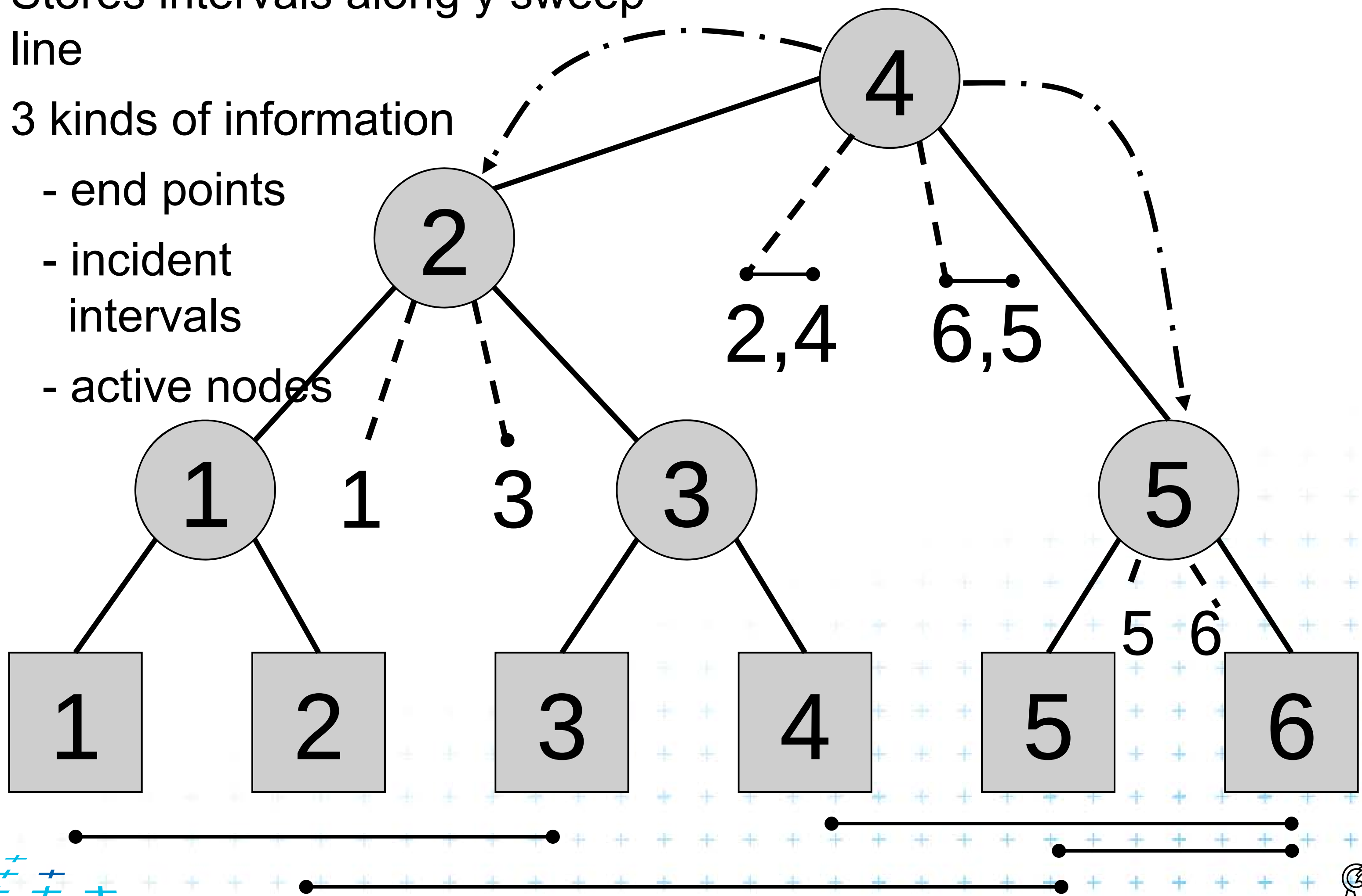
Static interval tree – Example



[Vigneron]

Static interval tree [Edelsbrunner80]

- Stores intervals along y sweep line
- 3 kinds of information
  - end points
  - incident intervals
  - active nodes

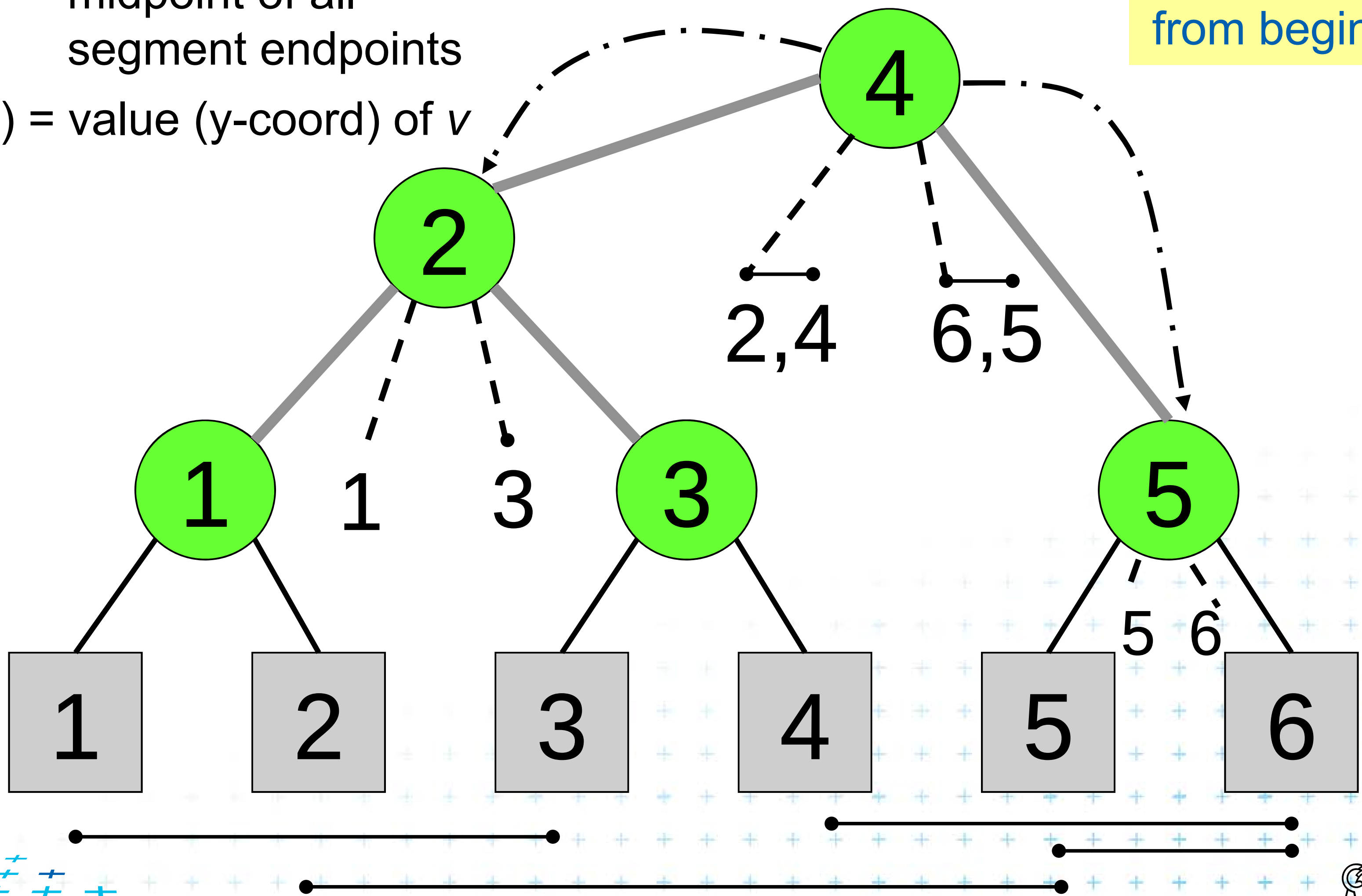


Primary structure – static tree for endpoints

v = midpoint of all segment endpoints

H(v) = value (y-coord) of *v*

Static – known from beginning



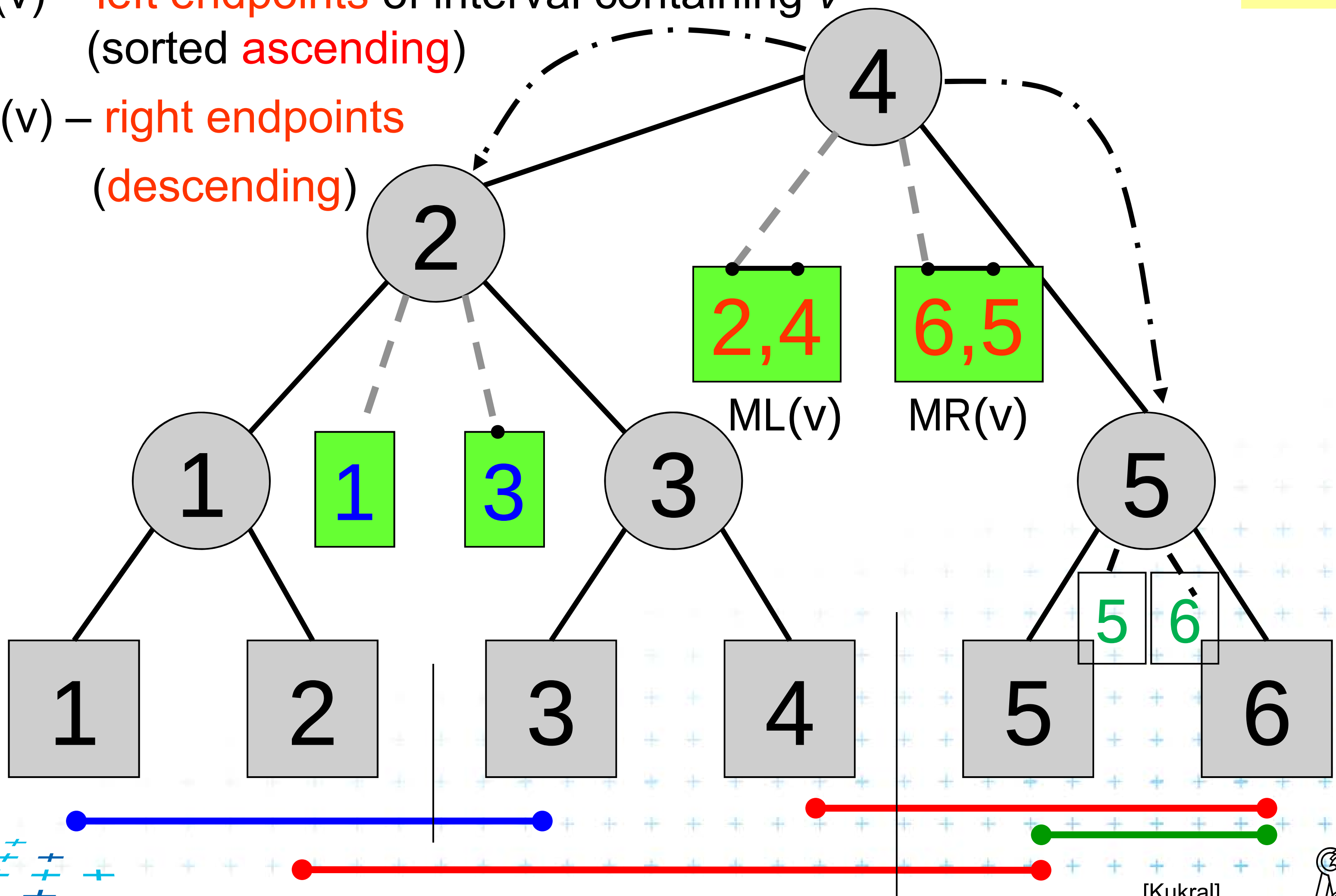
[Kukral]

Secondary lists of incident interval end-pts.

Dynamic

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

MR(v) – right endpoints (descending)

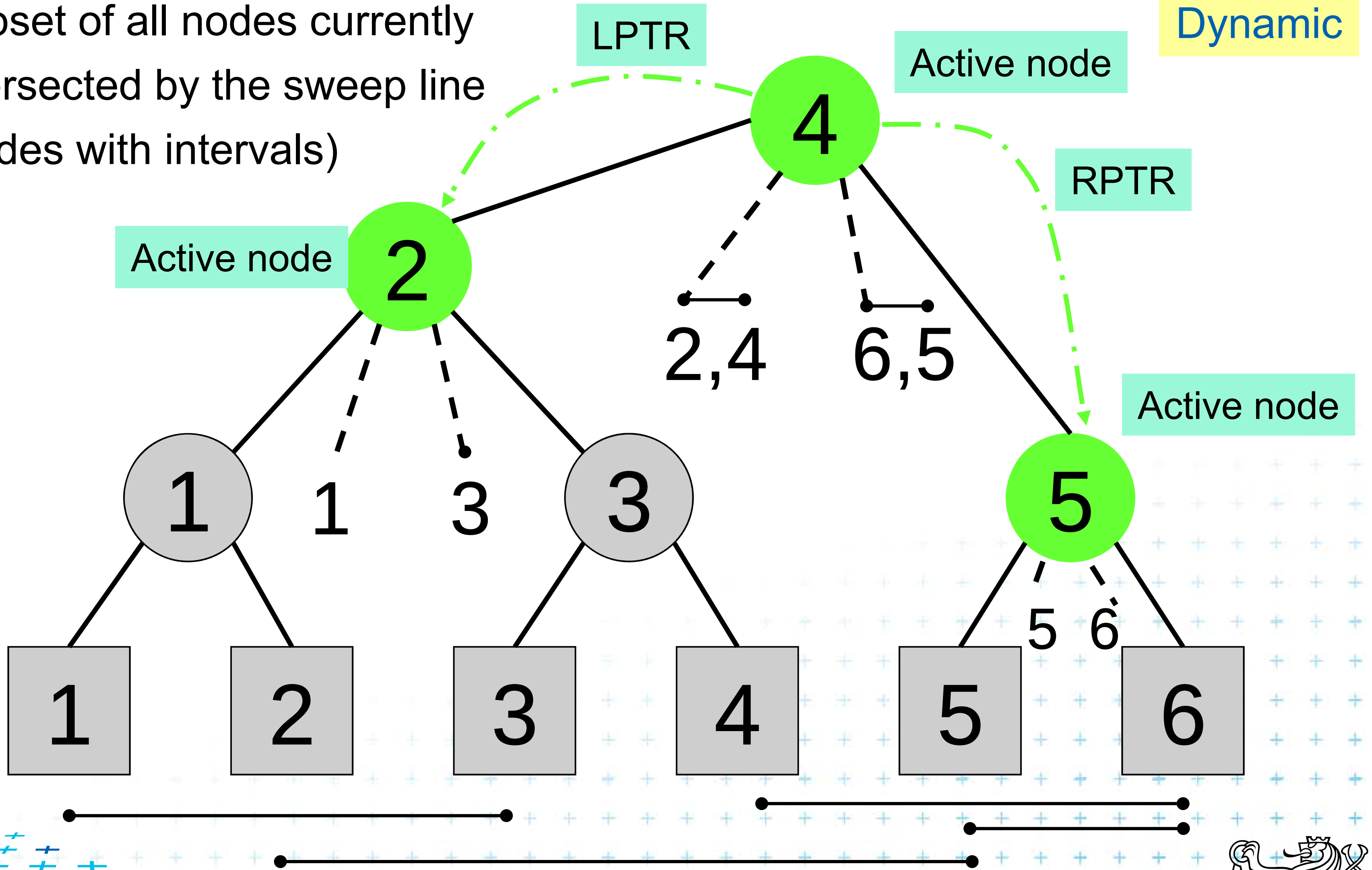


[Kukral]

Active nodes – intersected by the sweep line

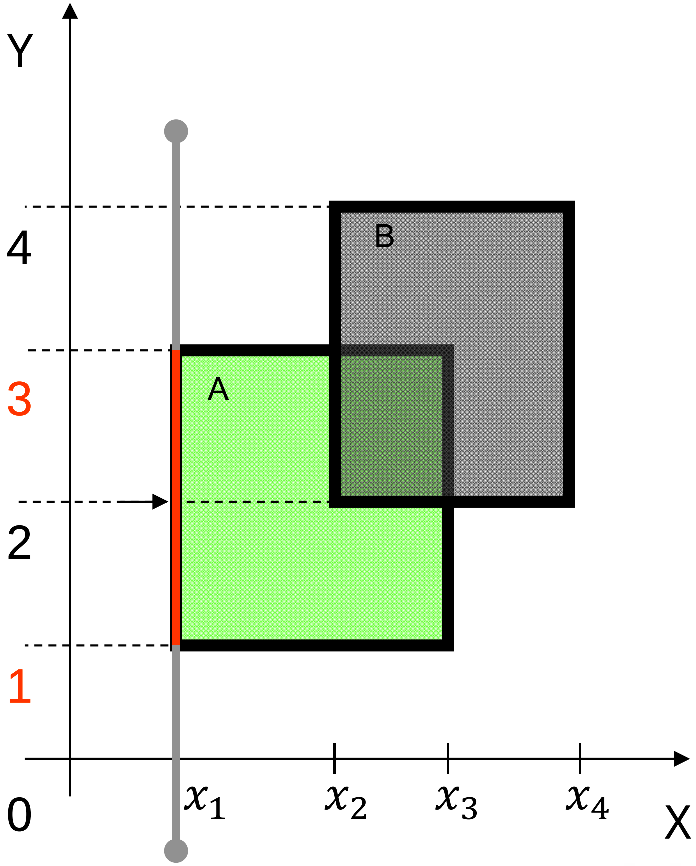
Subset of all nodes currently intersected by the sweep line (nodes with intervals)

Dynamic



[Kukral]

Entries in the event queue



$(x_i\ , y_{il}\ , y_{ir}\ , t)$

$(x_1\ , 1\ , 3\ , left)$

$(x_2\ , 2\ , 4\ , left)$

$(x_3\ , 1\ , 3\ , right)$

$(x_4\ , 2\ , 4\ , right)$

Static nodes in the SL status tree

1,2,3,4

Query = sweep and report intersections

Rectangle**Intersections****(S)**

Input: Set S of rectangles

Output: Intersected rectangle pairs

```
1. Preprocess( S )              // create the interval tree T  (for y-coords)
                                // and event queue Q           (for x-coords)
2. while ( Q ≠ ∅ ) do
3.     Get next entry (x_i, y_iL, y_iR, t) from Q       // t ∈ { left | right }
4.     if ( t = left )    // left edge
5.             a) QueryInterval (y_iL, y_iR, root(T))  // report intersections
6.             b) InsertInterval (y_iL, y_iR, root(T))  // insert new interval
7.     else               // right edge
8.             c) DeleteInterval (y_iL, y_iR, root(T))
```

Preprocessing

Preprocess(S)

Input: Set S of rectangles

Output: Primary structure of the interval tree T and the event queue Q

1. T = PrimaryTree(S) // Construct the static primary structure
   // of the interval tree -> sweep line STATUS T

2. // Init event queue Q with vertical rectangle edges in ascending order ~x
   // Put the left edges with the same x ahead of right ones
3. for i = 1 to n
4. insert$\left((x_{iL}, y_{iL,} y_{iR}, \text{left}), Q\right)$ // left edges of i-*th* rectangle
5. insert$\left((x_{iR}, y_{iL,} y_{iR}, right), Q\right)$ // right edges

DCGI

Interval tree – primary structure construction

PrimaryTree(*S*) // only the y-tree structure, without intervals

Input: Set *S* of rectangles

Output: Primary structure of an interval tree *T*

1. S_y = Sort endpoints of all segments in *S* according to *y*-coordinate
2. *T* = BST(S_y)
3. **return** *T*

BST(S_y)

1. **if**($|S_y| = 0$) **return** null
2. *yMed = median of* S_y // *the smaller item for even* S_y*.size*
3. *L* = endpoints $p_y \leq yMed$
4. R = endpoints $p_y > yMed$
5. *t* = *new* IntervalTreeNode*(yMed)*
6. *t.left* = BST(*L*)
7. *t.right* = BST(*R*)
8. **return** *t*

DCGI

Interval tree – search the intersections

QueryInterval (*b, e, T*)

Input: Interval of the edge and current tree *T*

Output: Report the rectangles that intersect [*b, e*]

if(*T* = null) **return**

i=0; **if**(b < H(v) < e) // forks at this node

    while ~~(*MR*(v).[i] >= b) &&~~ (i < Count(v)) // Report all intervals inM

        ReportIntersection; i++

    QueryInterval(*b,e,T.LPTR*) // jump to active

    QueryInterval(*b,e,T.RPTR*) // node below

else if (H(v) ≤ b < e) // search RIGHT (←)

    while (*MR*(v).[i] >= b) && (i < Count(v))

        ReportIntersection; i++

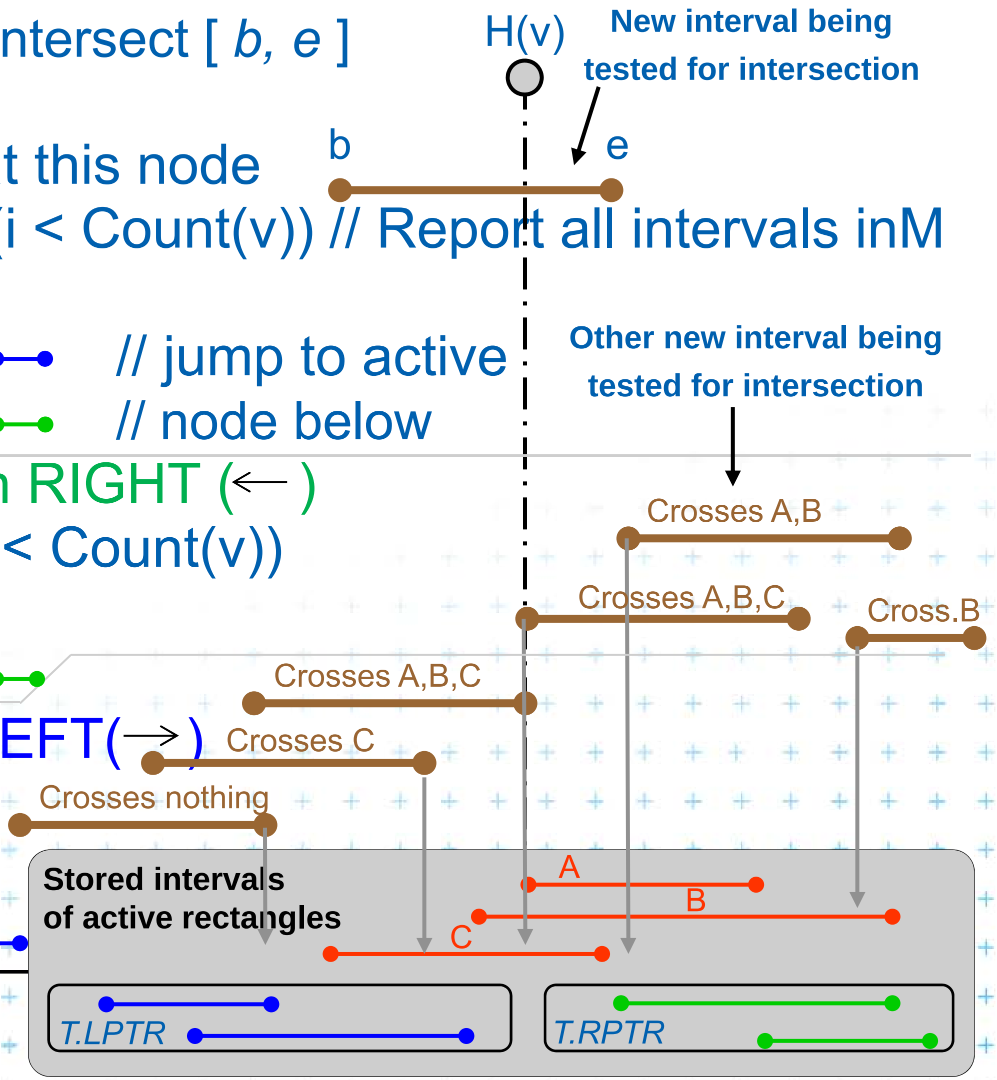
    QueryInterval(*b,e,T.RPTR*)

else // b < e ≤ H(v) //search LEFT(→)

    while (*ML*(v).[i] <= e)

        ReportIntersection; i++

    QueryInterval(*b,e,T.LPTR*)



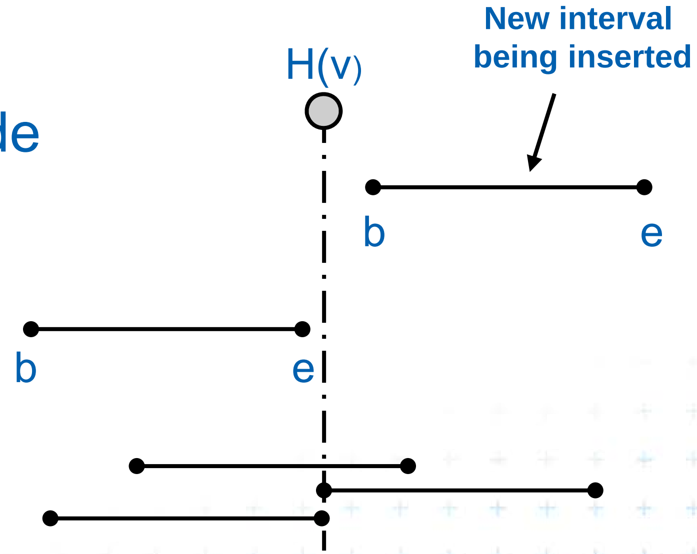
Interval tree - interval insertion

InsertInterval (*b, e, T*)

Input: Interval [*b*,*e*] and interval tree *T*

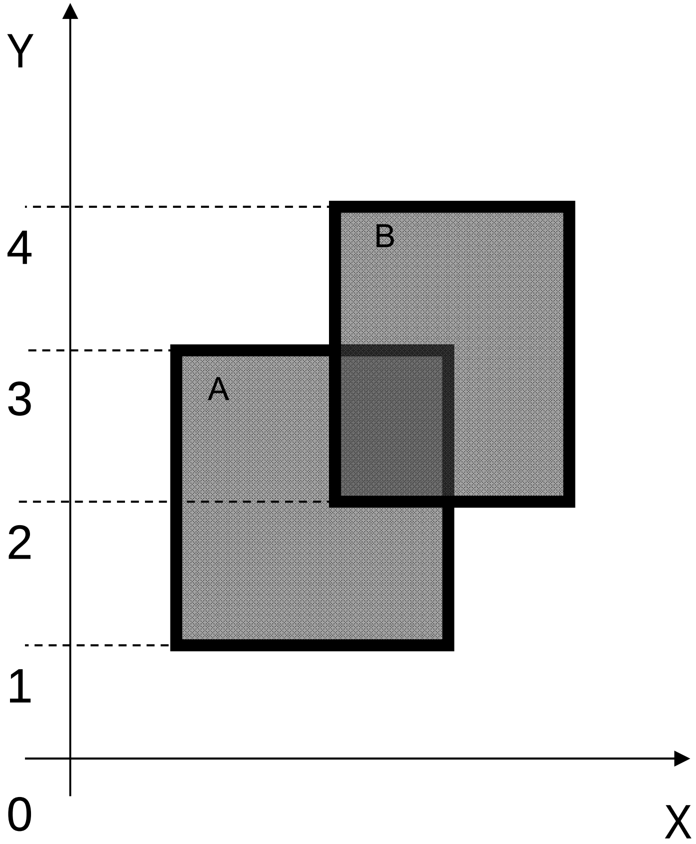
Output: *T* after insertion of the interval

```
v = root(T )
while( v != null )        // find the fork node
    if (H(v) < b < e)
        v = v.right       // continue right
    else if (b < e < H(v))
        v = v.left        // continue left
    else  // b ≤ H(v) ≤ e  // insert interval
        set v node to active
        connect LPTR resp. RPTR to its parent (active node above)
        insert [b,e] into list ML(v) – sorted in ascending order of b’s
        insert [b,e] into list MR(v) – sorted in descending order of e’s
        break
endwhile
return T
```

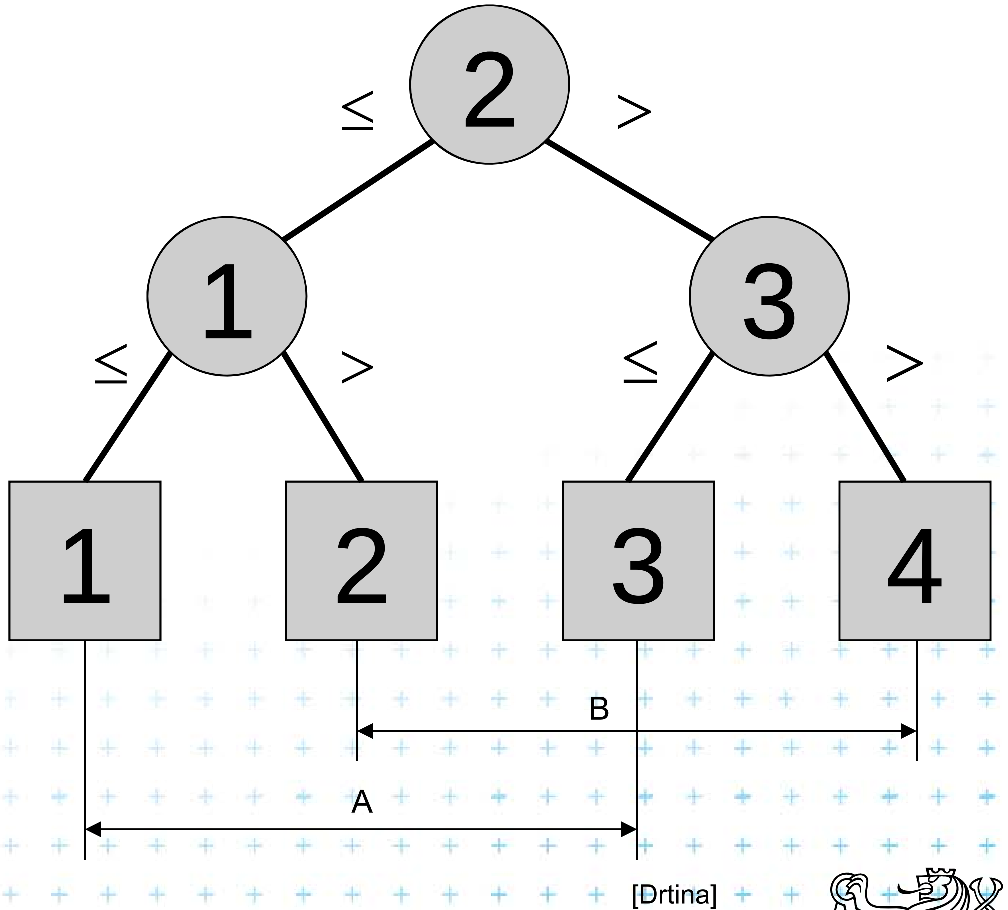


Example 1

Example 1 – static tree on endpoints



H(v) – value of node *v*



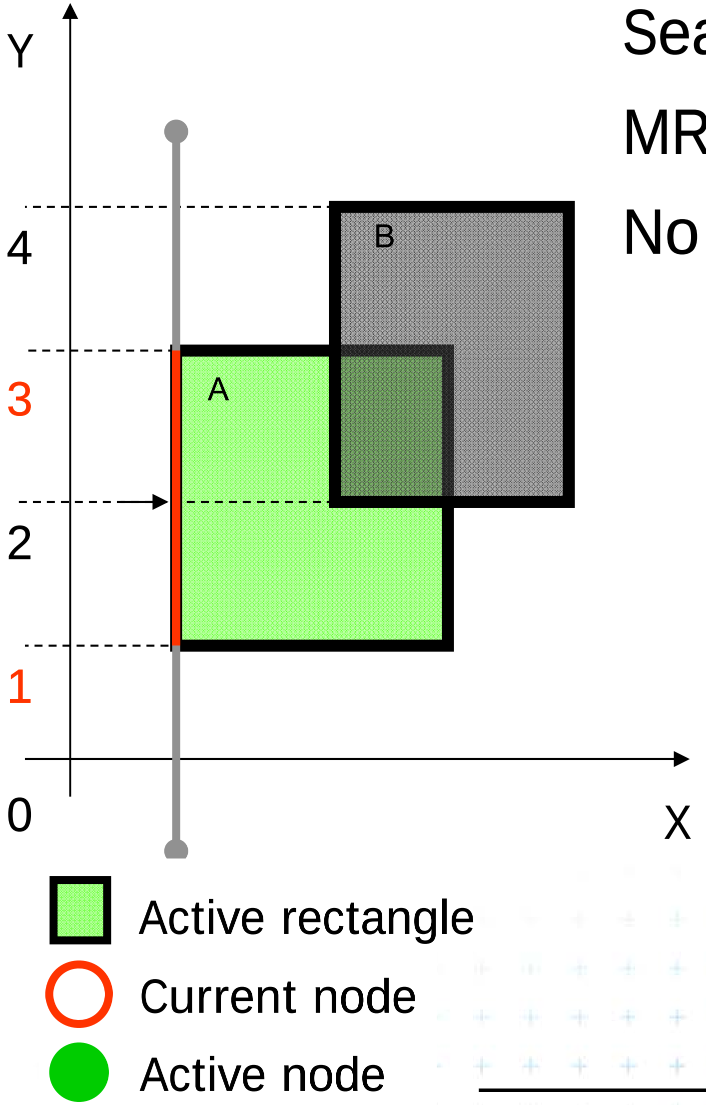
Interval insertion [1,3] a) Query Interval

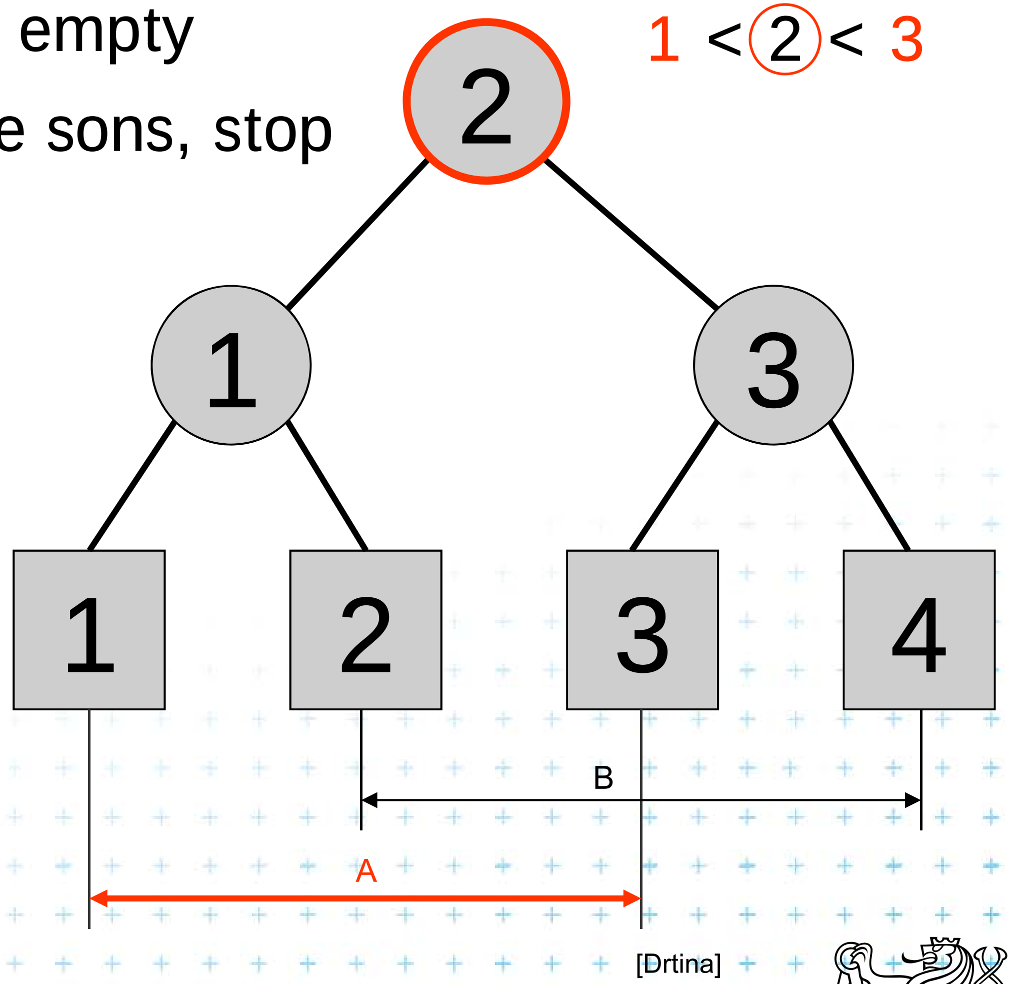
Search MR(v) or ML(v): ← b < H(v) < e

MR(v) is empty

No active sons, stop

1 < (2) < 3

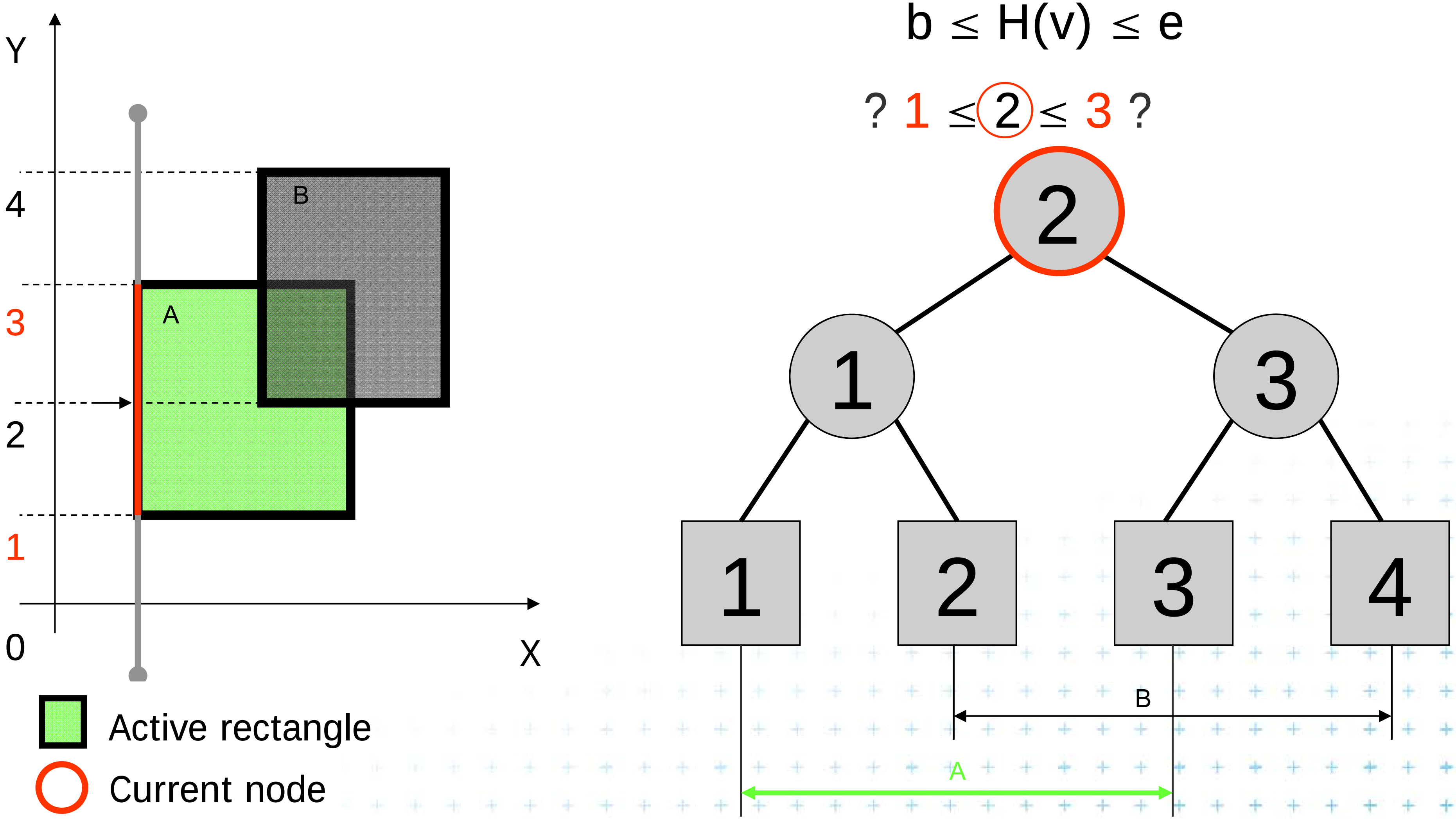




Interval insertion [1,3]   b) Insert Interval

$b \le H(v) \le e$

$? \ 1 \le 2 \le 3 \ ?$



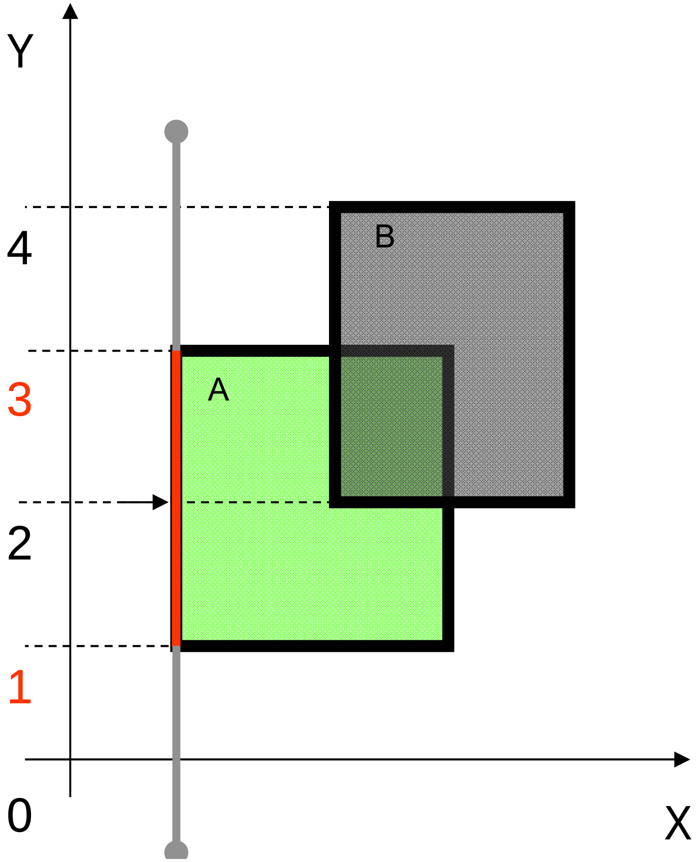
Active rectangle

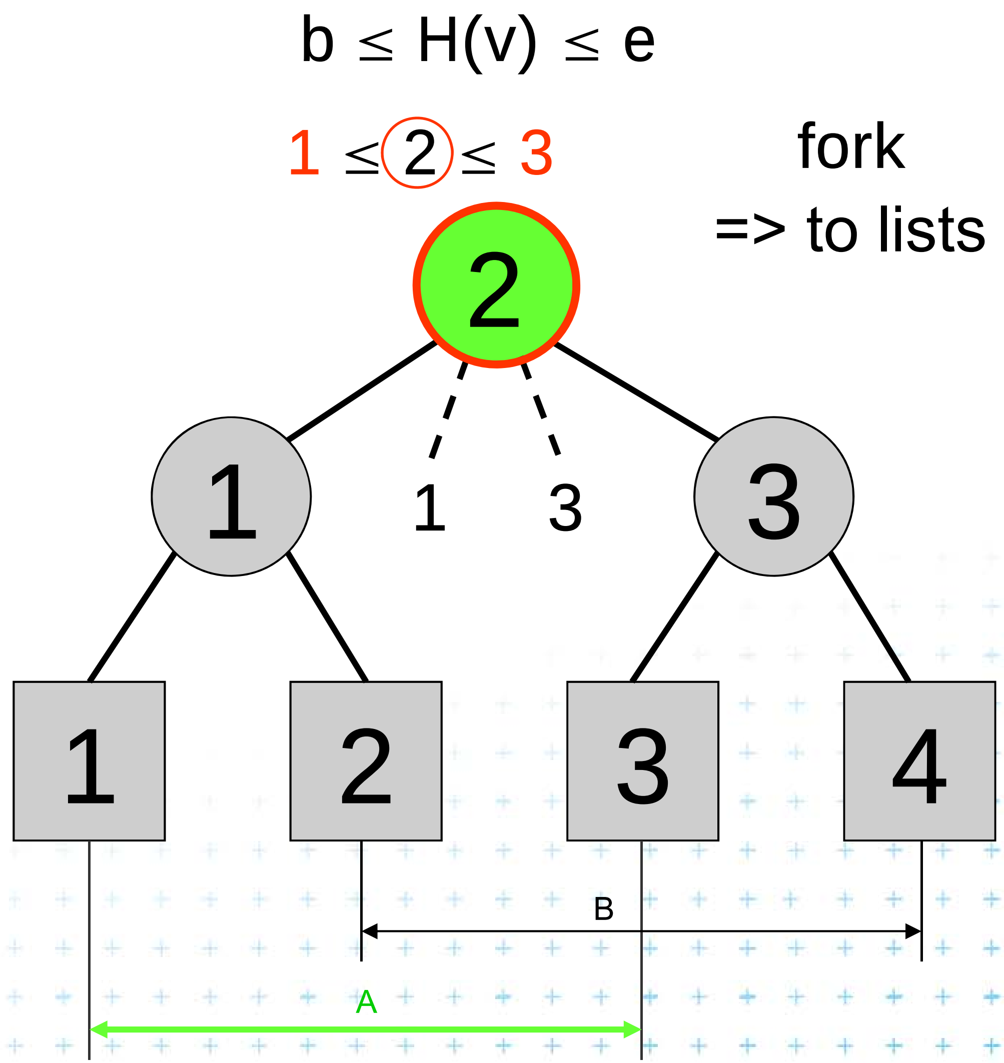
Current node

Active node

[Drtina]

Interval insertion [1,3]   b) Insert Interval

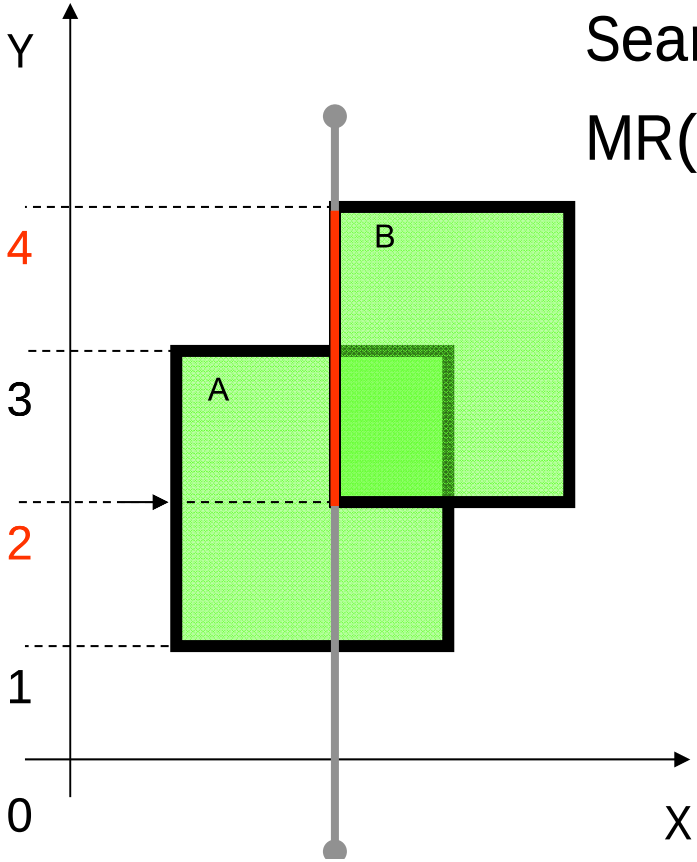




- Active rectangle
- Current node
- Active node

[Drtina]

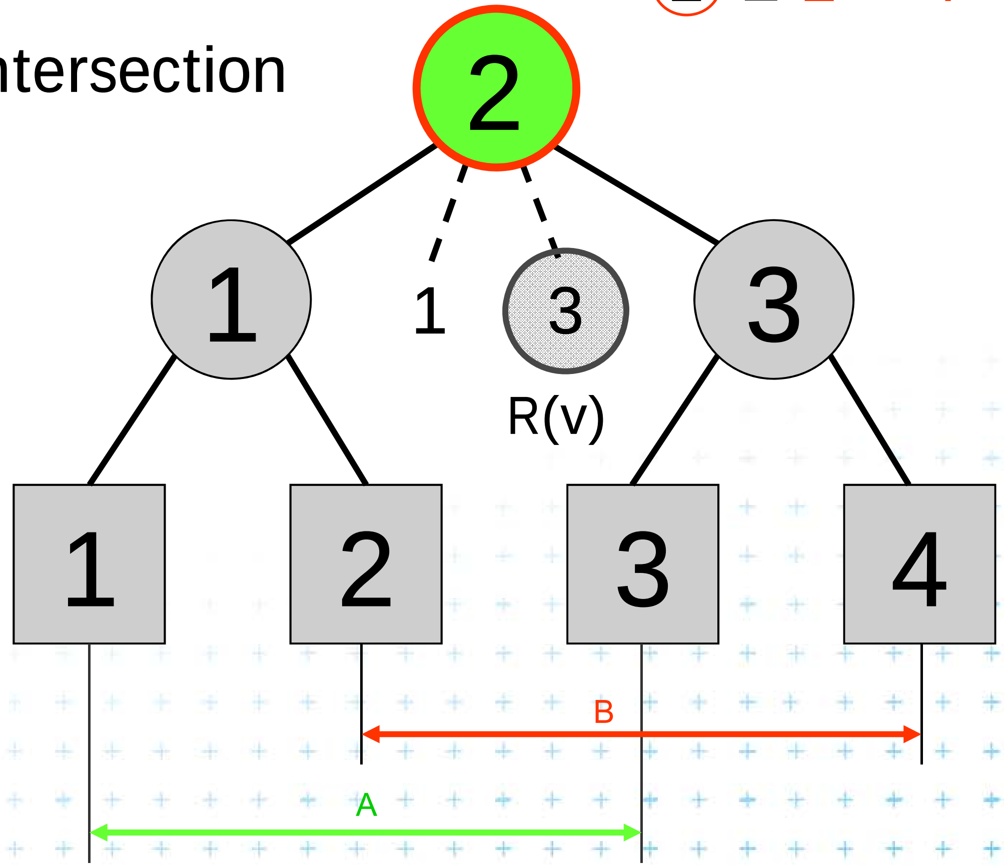
Interval insertion [2,4] a) Query Interval



Search MR(v) only: ← H(v) ≤ b < e

MR(v)[1] = 3 ≥ 2? ② ≤ 2 < 4

=> intersection



- Active rectangle
- Current node
- Active node

[Drtina]

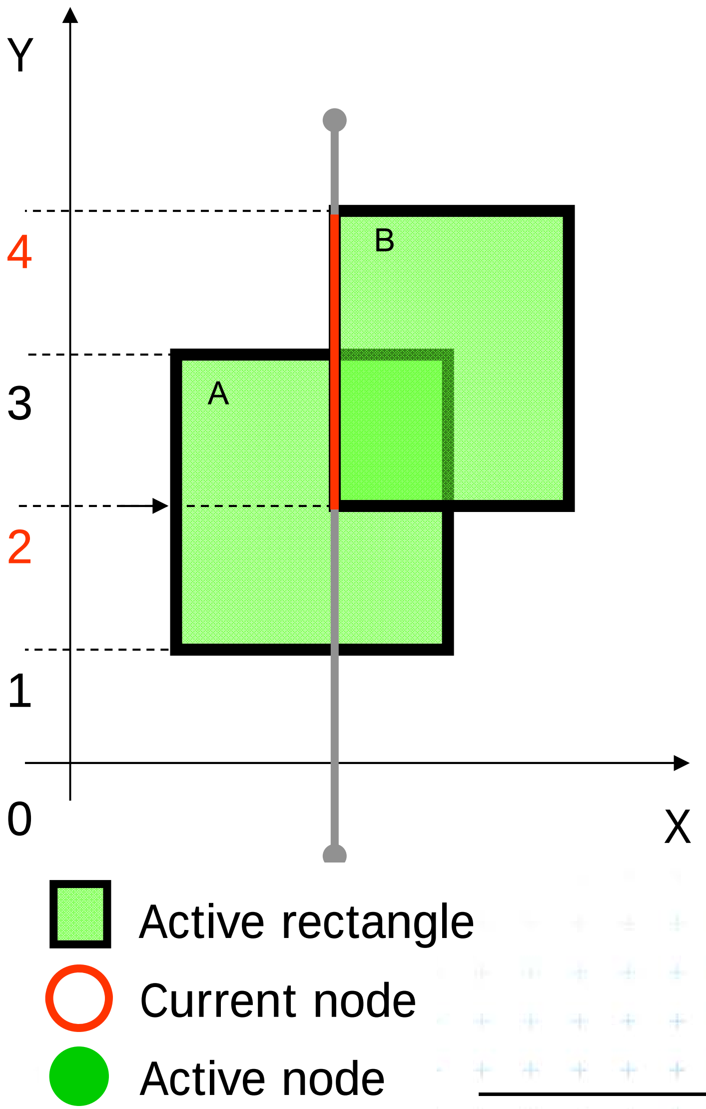
Interval insertion [2,4] b) Insert Interval

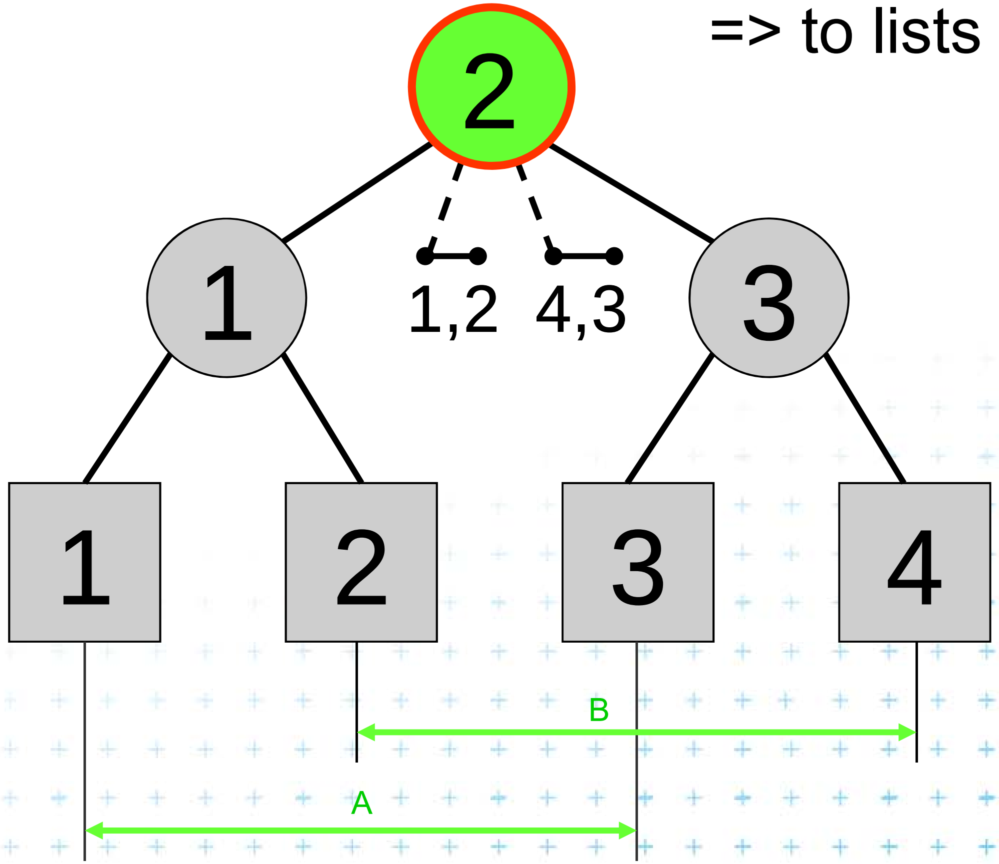
$b \leq H(v) \leq e$

$2 \leq 2 \leq 4$

fork

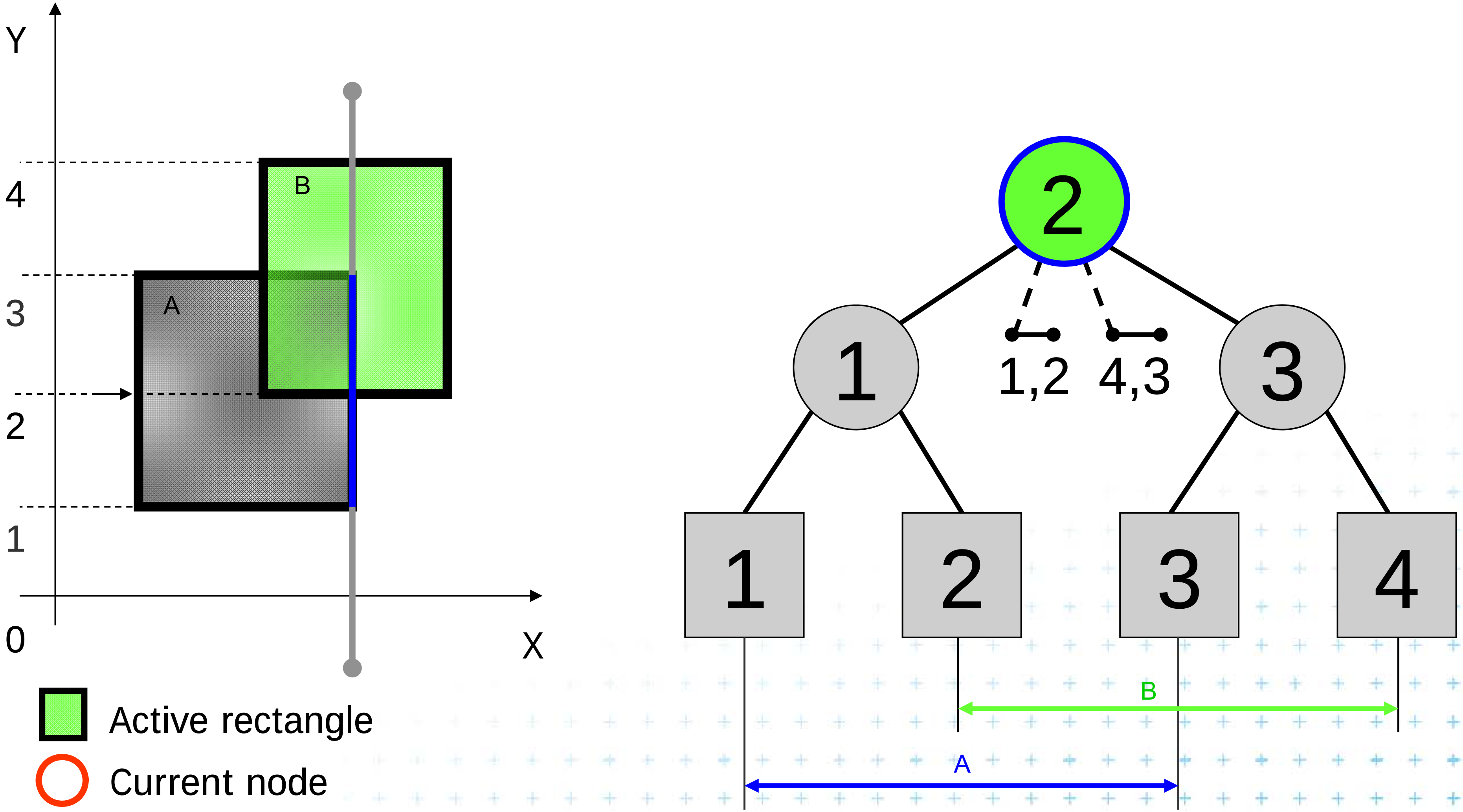
=> to lists





[Drtina]

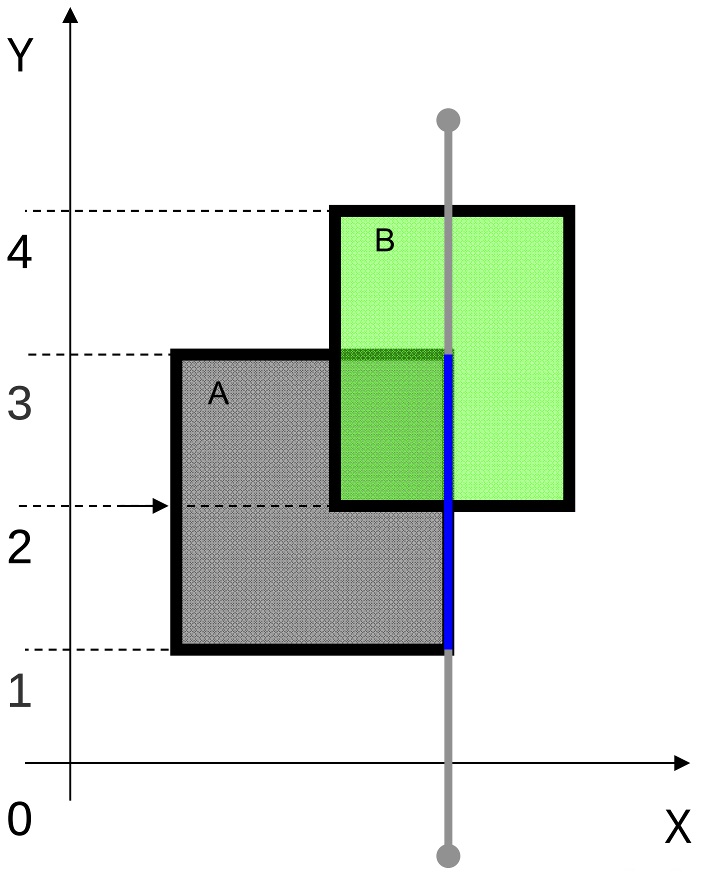
Interval delete [1,3]

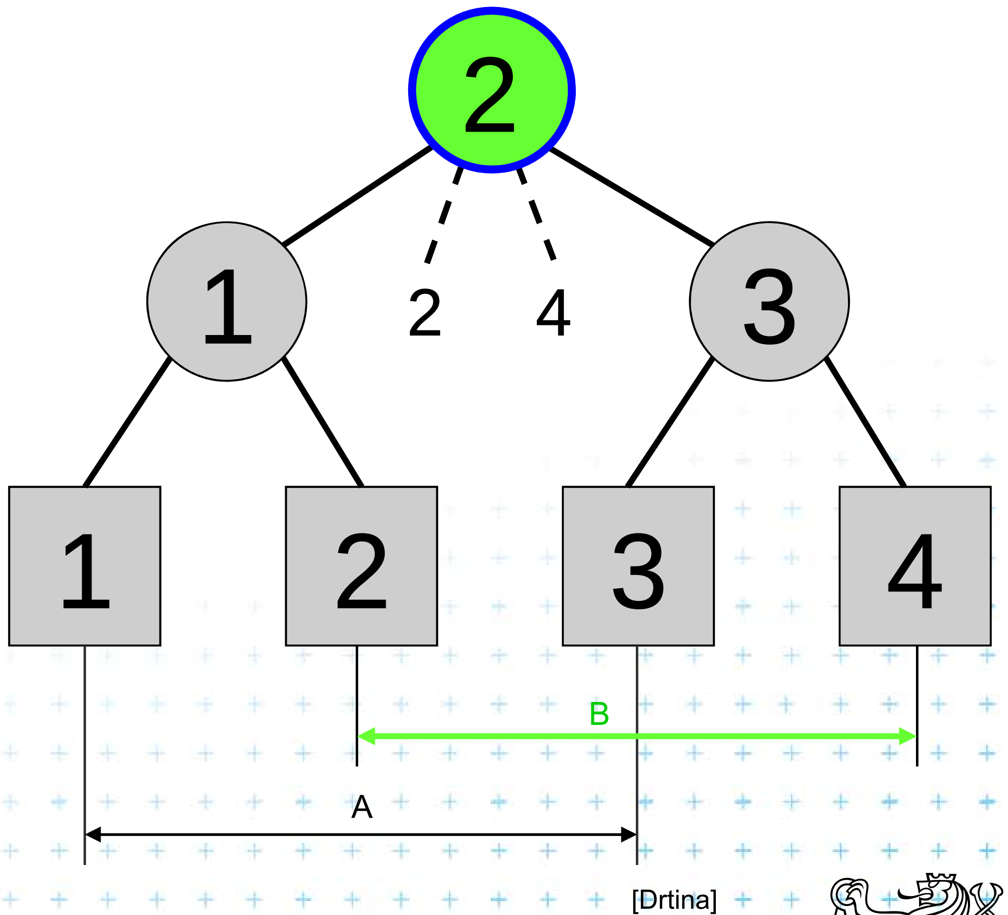


Active rectangle
Current node
Active node

[Drtina]

Interval delete [1,3]



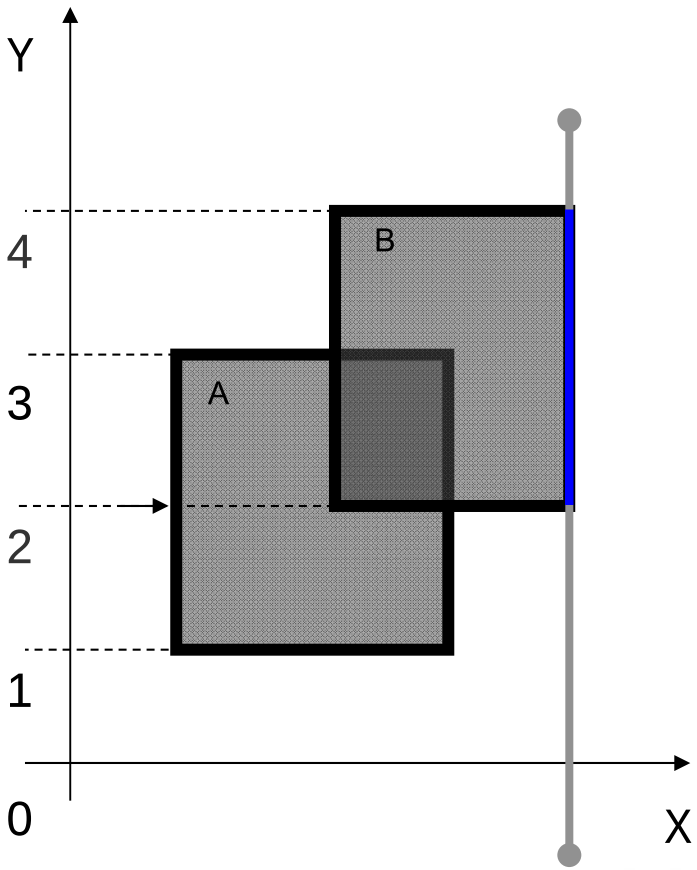


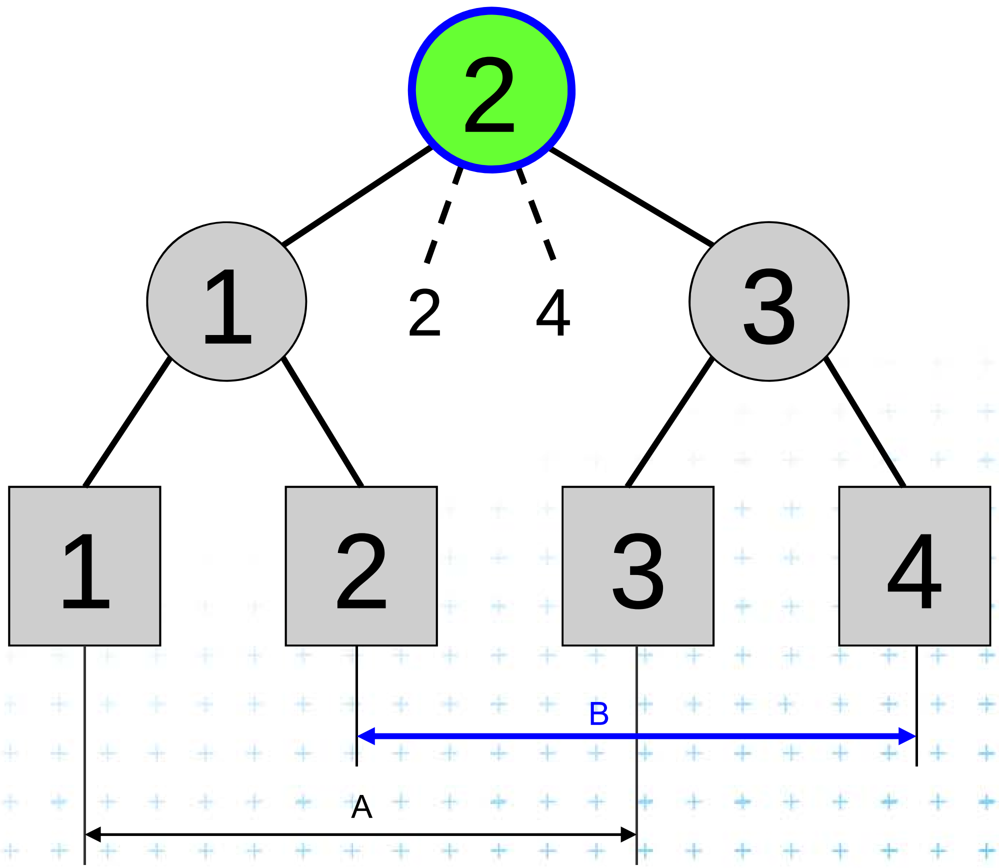
Active rectangle

Current node

Active node

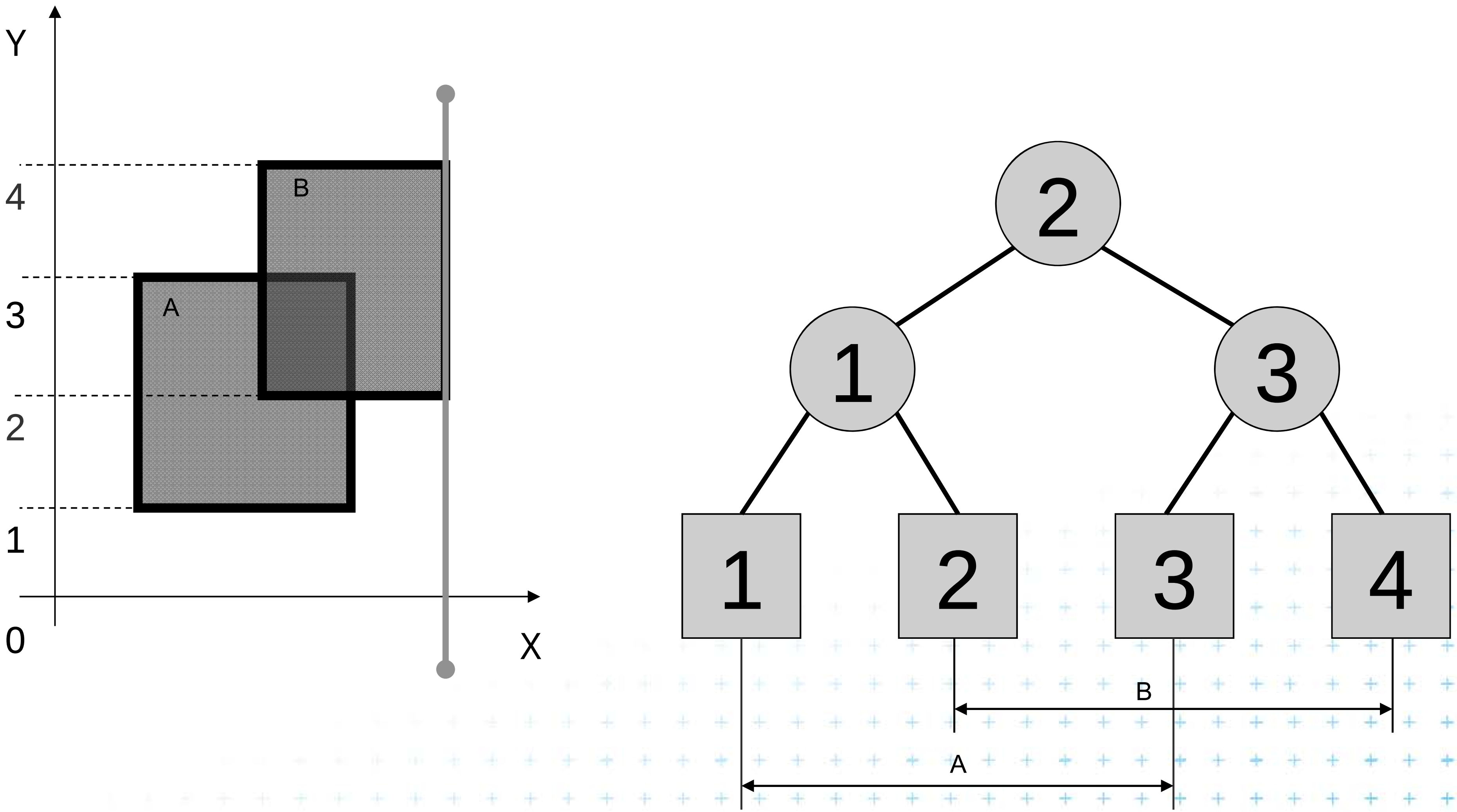
Interval delete [2,4]





[Drtina]

Interval delete [2,4]



[Drtina]

Example 2

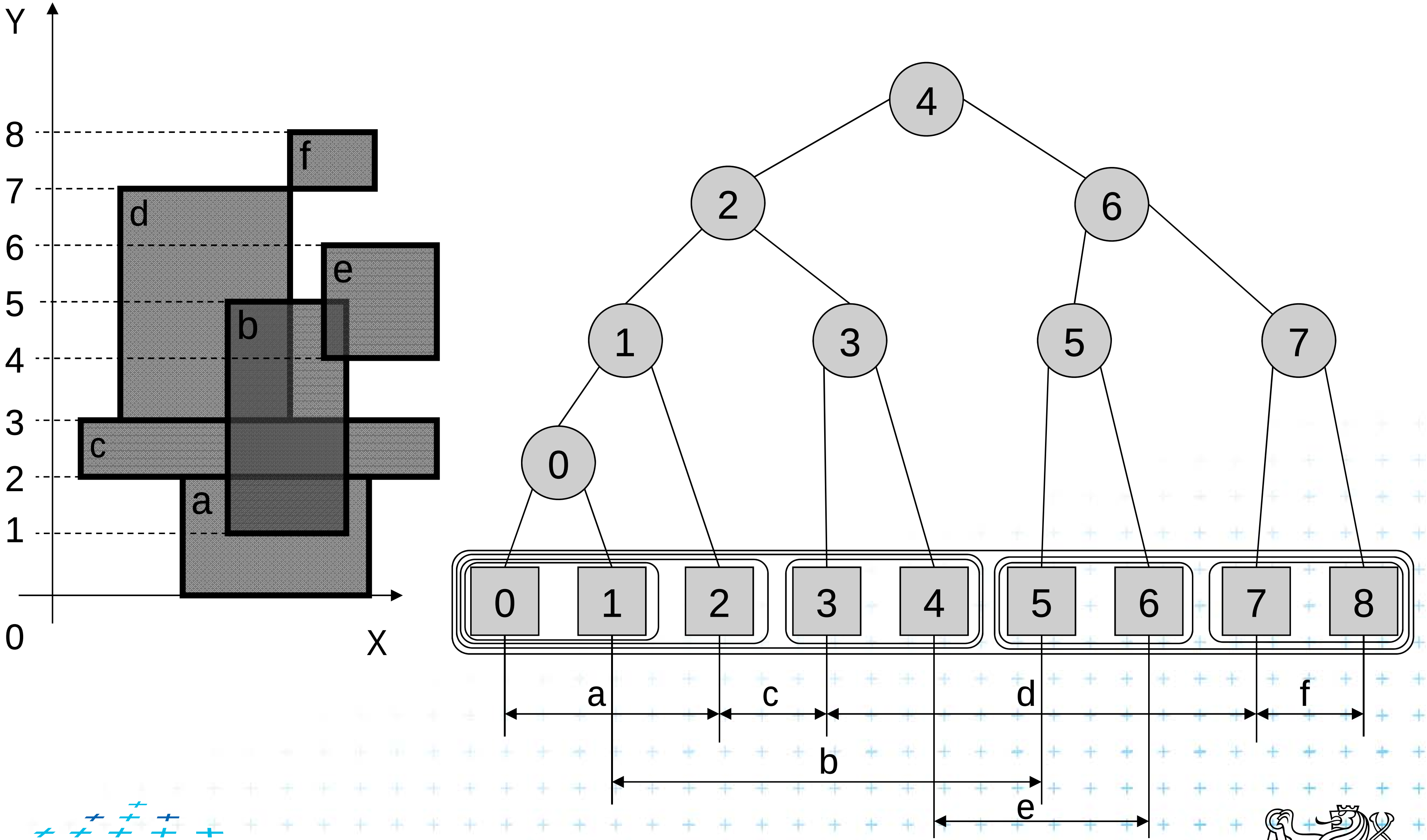
Query = sweep and report intersections

RectangleIntersections**(S)**
Input: Set S of rectangles
Output: Intersected rectangle pairs

// this is a copy of the slide before
// just to remember the algorithm

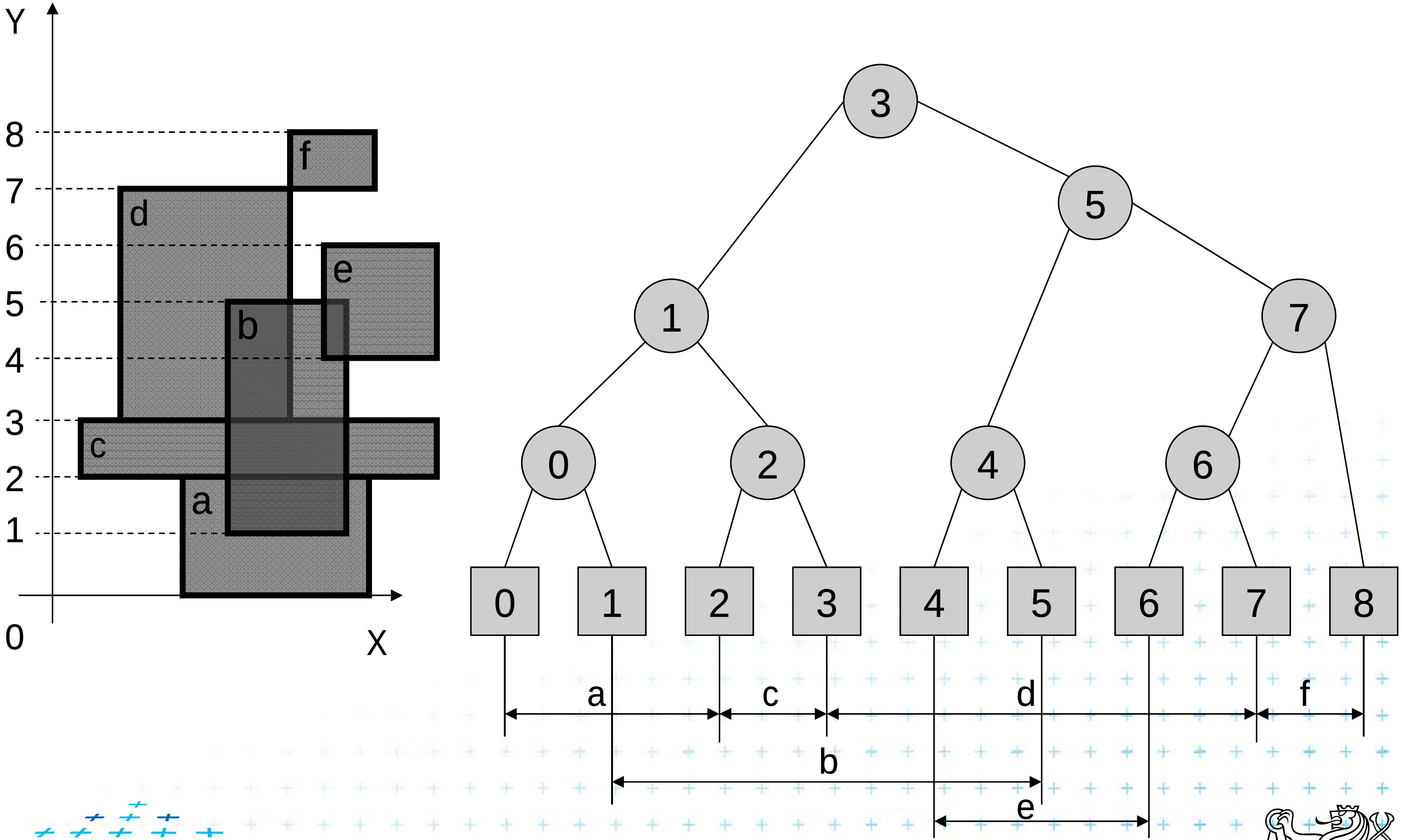
1. Preprocess(S) // create the interval tree T (for y-coords)
   // and event queue Q (for x-coords)
2. **while** ($Q \neq \emptyset$) do
3. Get next entry (x_i, y_{iL}, y_{iR}, t) from Q // $t \in$ { *left* | *right* }
4. **if** (t = left) // left edge
5. a) QueryInterval $(y_{iL}, y_{iR}, \text{root}(T))$ // report intersections
6. b) InsertInterval $(y_{iL}, y_{iR}, \text{root}(T))$ // insert new interval
7. **else** // right edge
8. c) DeleteInterval $(y_{iL}, y_{iR}, \text{root}(T))$

Example 2 – tree created by PrimaryTree(S)



[Drtina]

Example 2 – slightly unbalanced tree



[Drtina]

Insert [2,3] – empty => b) Insert Interval

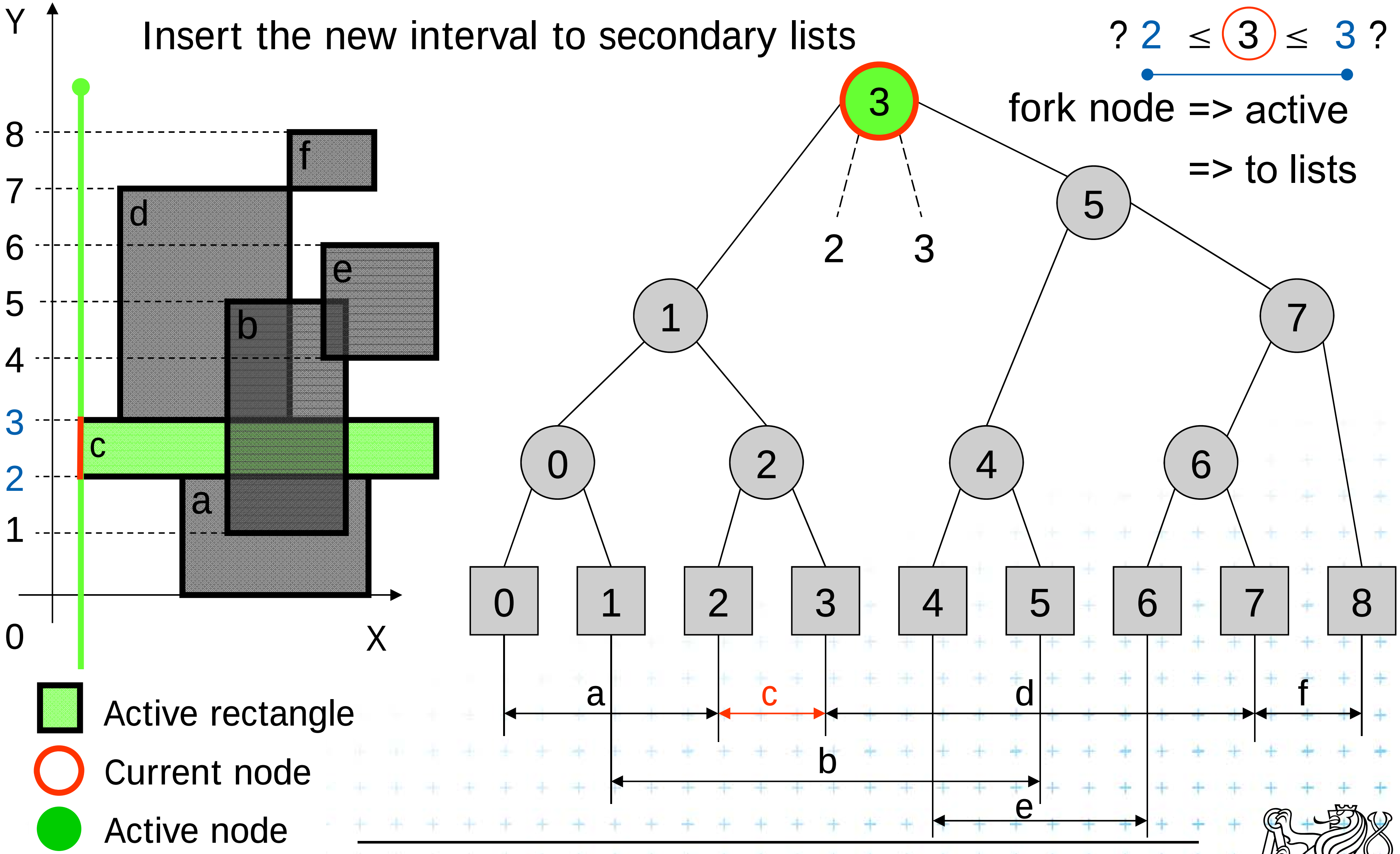
$b \le H(v) \le e$

Insert the new interval to secondary lists

? 2 ≤ 3 ≤ 3 ?

fork node => active

=> to lists



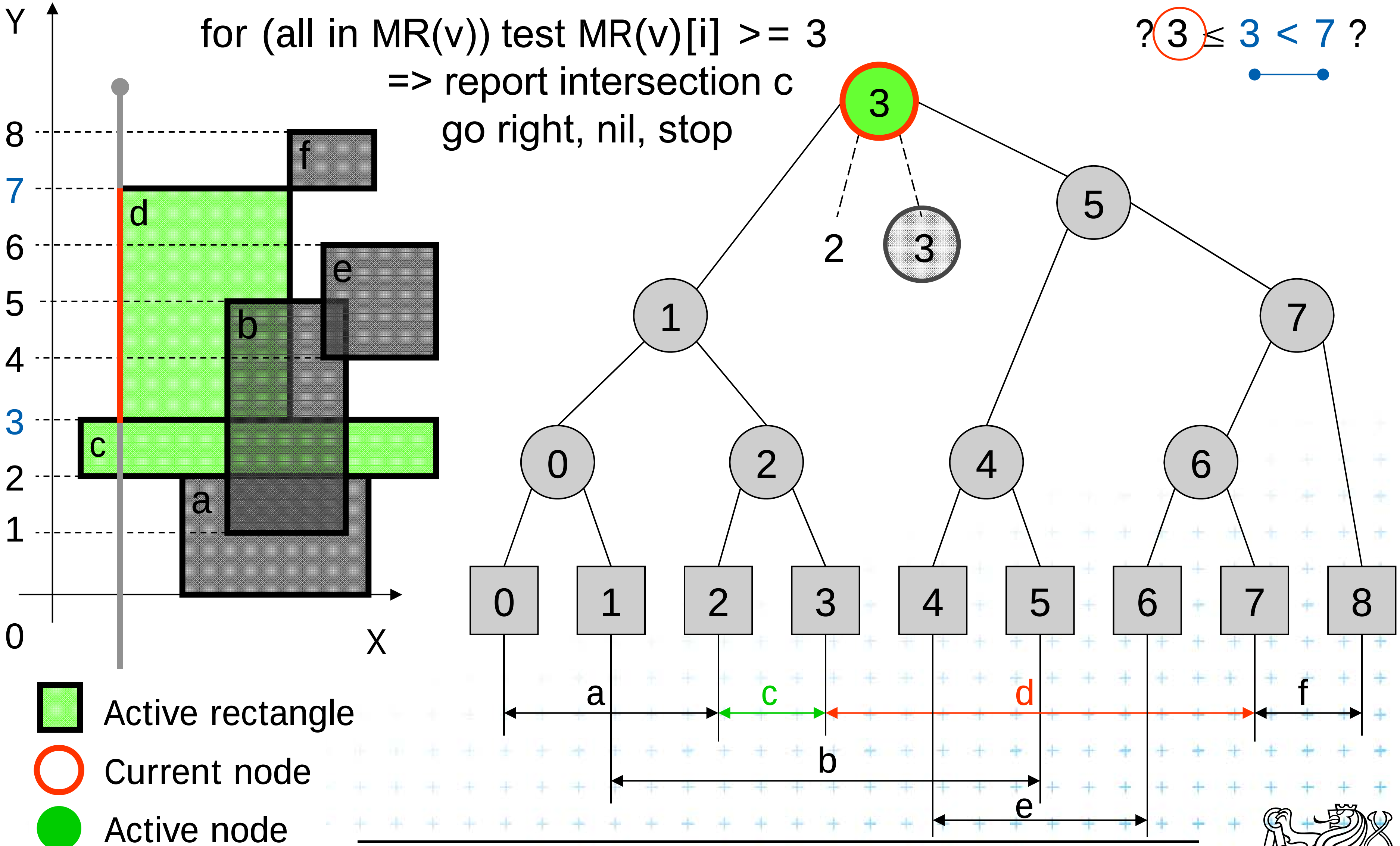
[Drtina]

Insert [3,7] a) Query Interval

$H(v) \leq b < e$

for (all in MR(v)) test MR(v)[i] >= 3
=> report intersection c
go right, nil, stop

? 3 ≤ 3 < 7 ?



Insert [3,7] b) Insert Interval

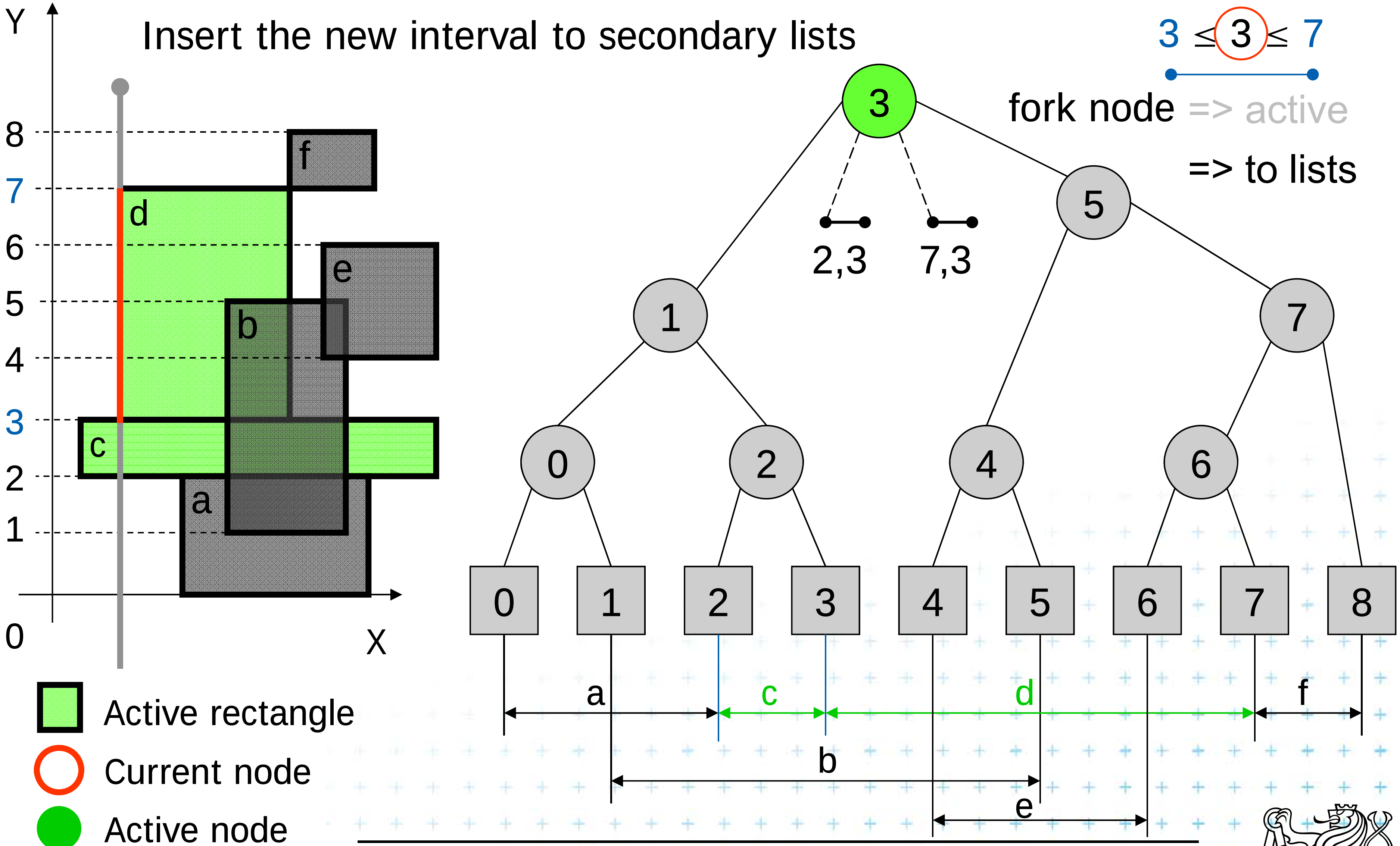
$b \le H(v) \le e$

Insert the new interval to secondary lists

$3 \le 3 \le 7$

fork node => active

=> to lists



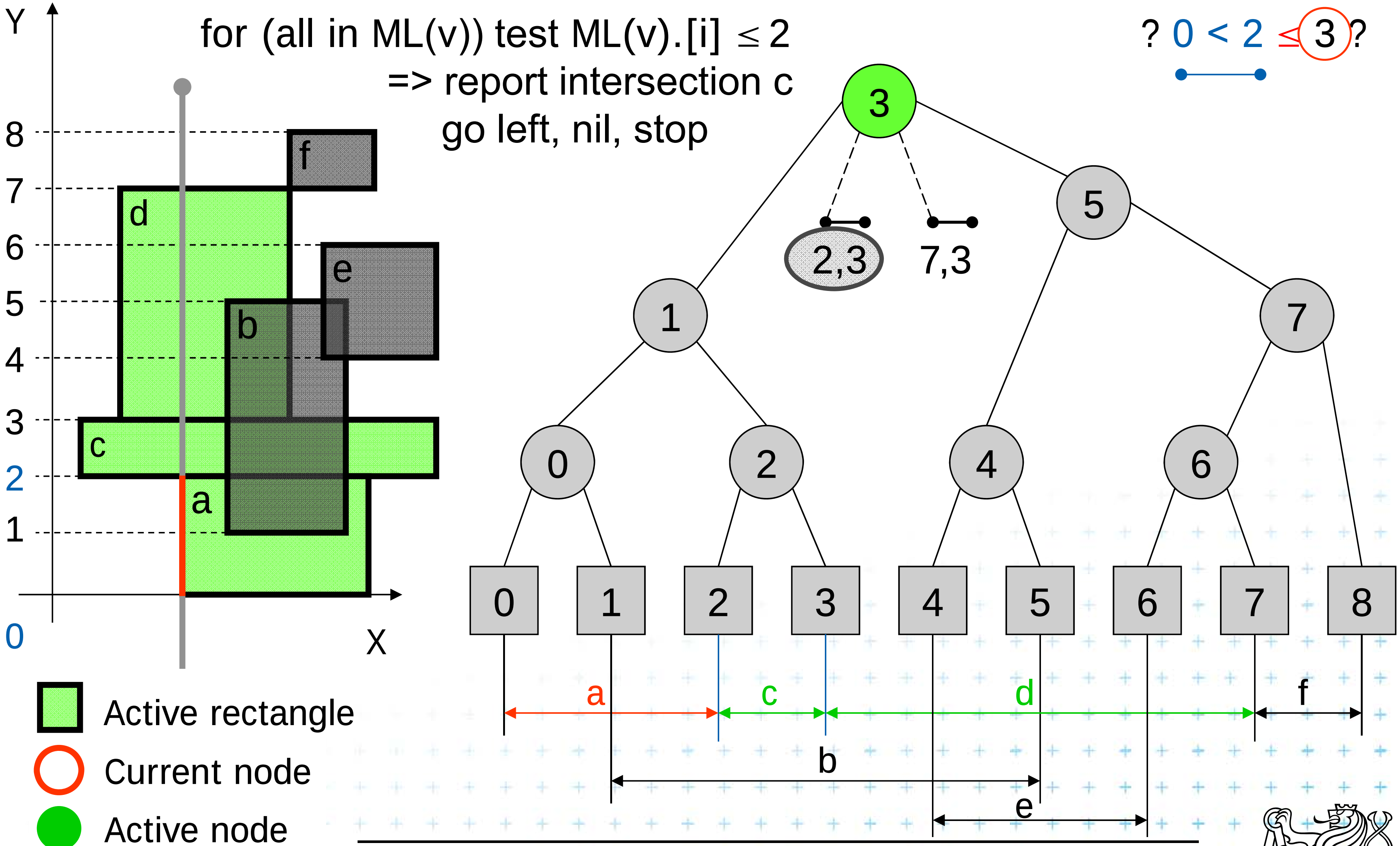
[Drtina]

Insert [0,2] a) Query Interval

b < e ≤ H(v)

for (all in ML(v)) test ML(v).[i] ≤ 2
=> report intersection c
go left, nil, stop

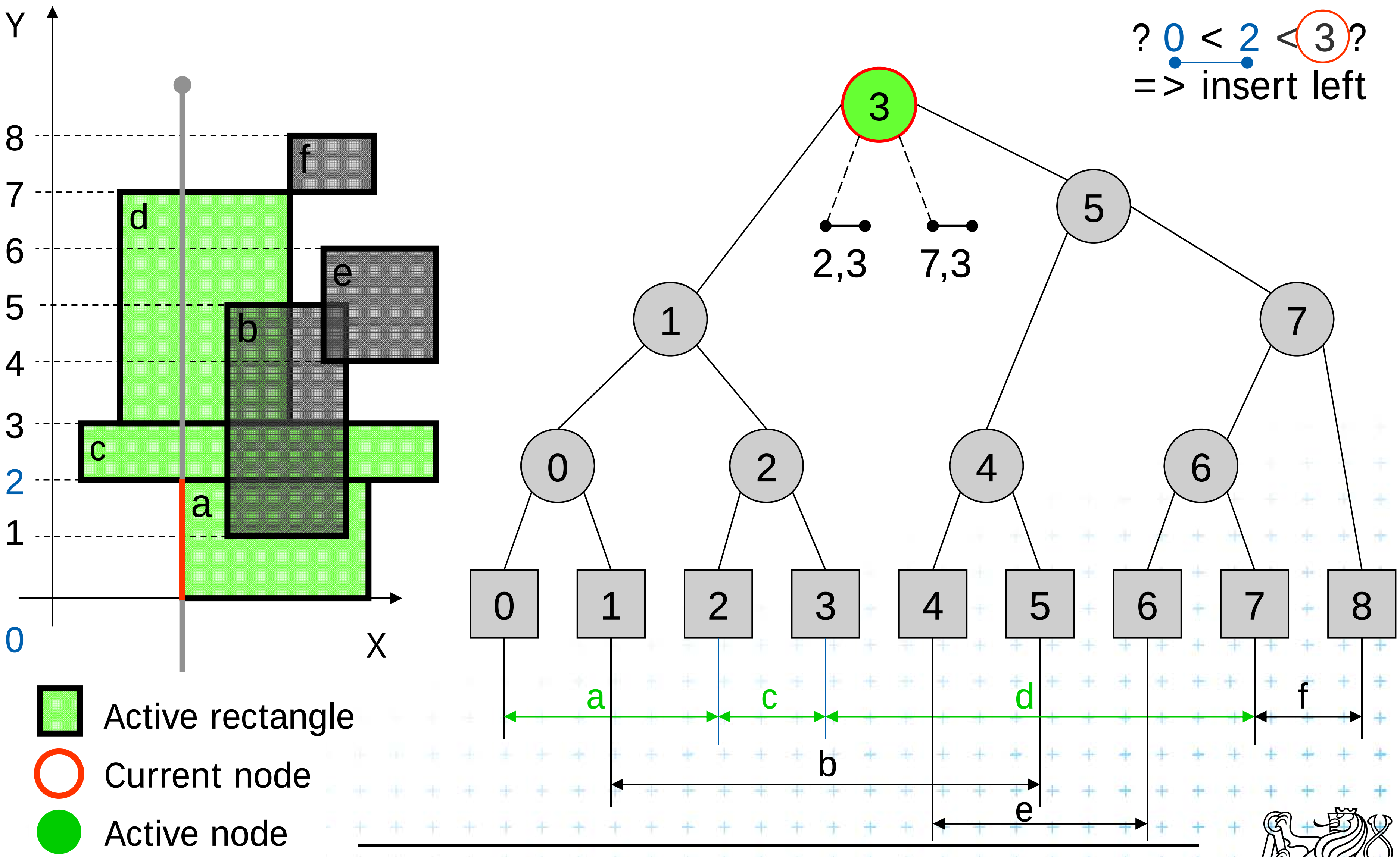
? 0 < 2 ≤ 3 ?



[Drtina]

Insert [0,2] b) Insert Interval 1/2

b < e <H(v)



[Drtina]

Insert [0,2] b) Insert Interval 2/2

b ≤ H(v) ≤ e

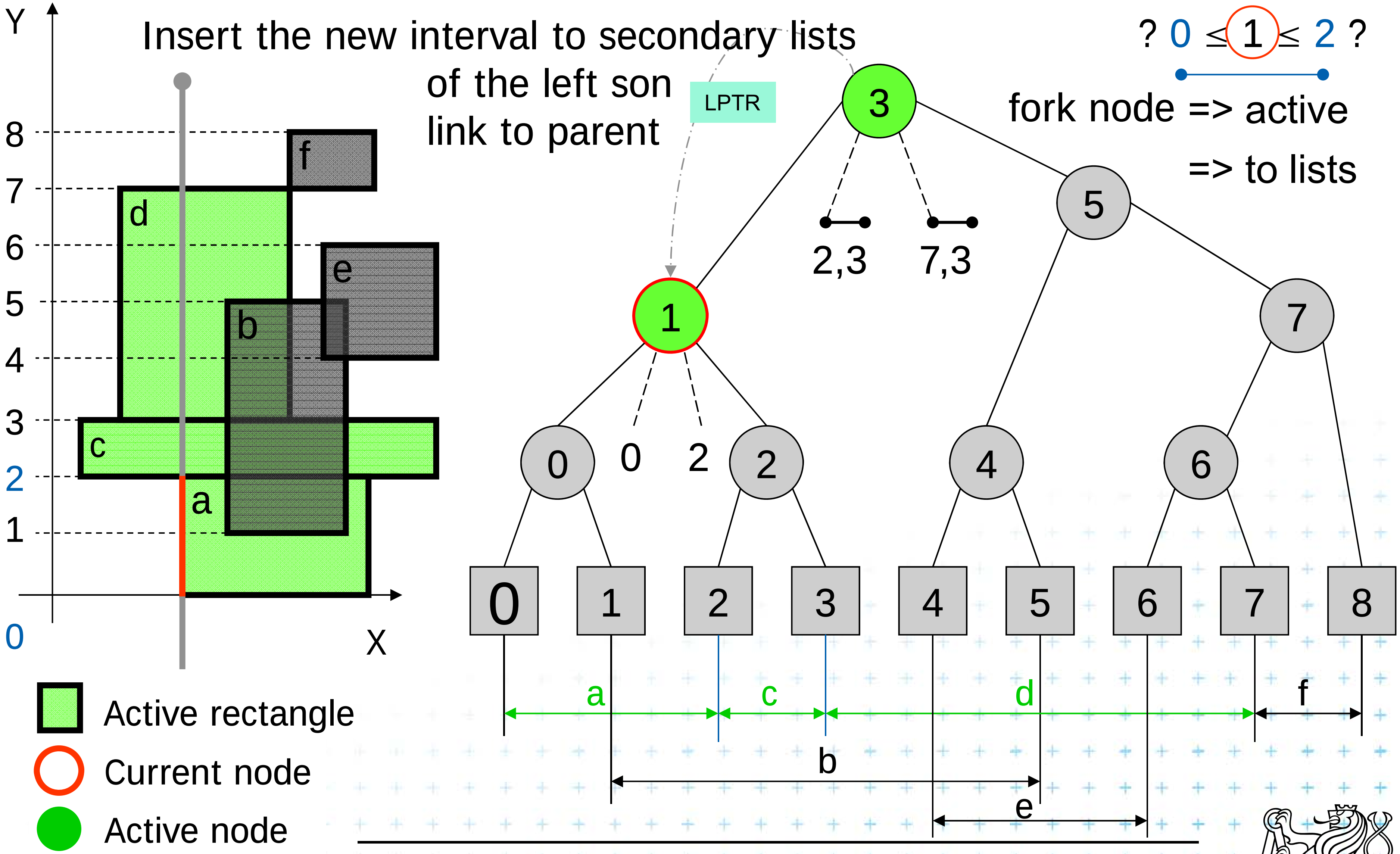
Insert the new interval to secondary lists of the left son

link to parent

? 0 ≤ 1 ≤ 2 ?

fork node => active

=> to lists



[Drtina]

Insert [1,5] a) Query Interval 1/2

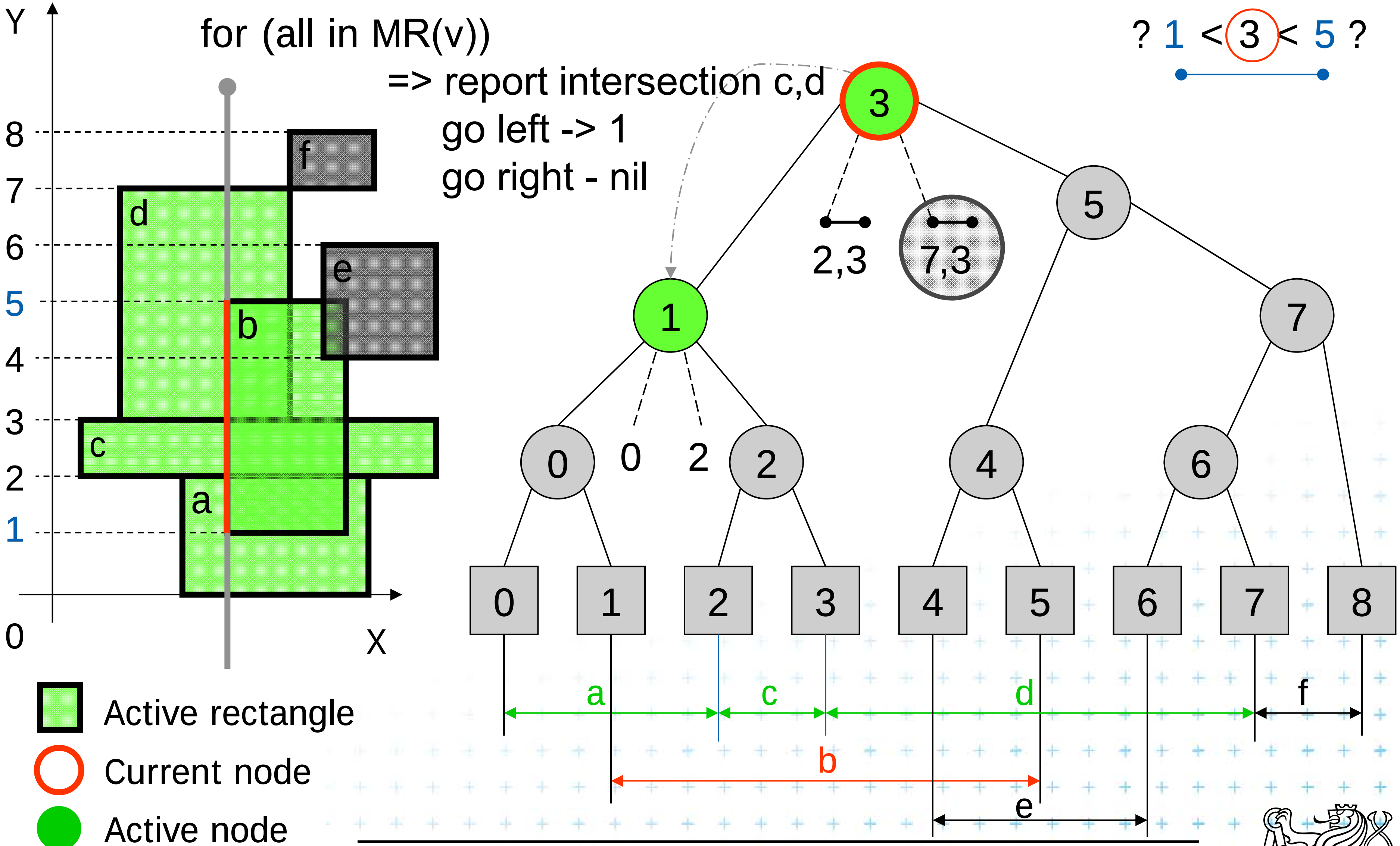
b < H(v) < e

for (all in MR(v))

=> report intersection c,d

go left -> 1

go right - nil

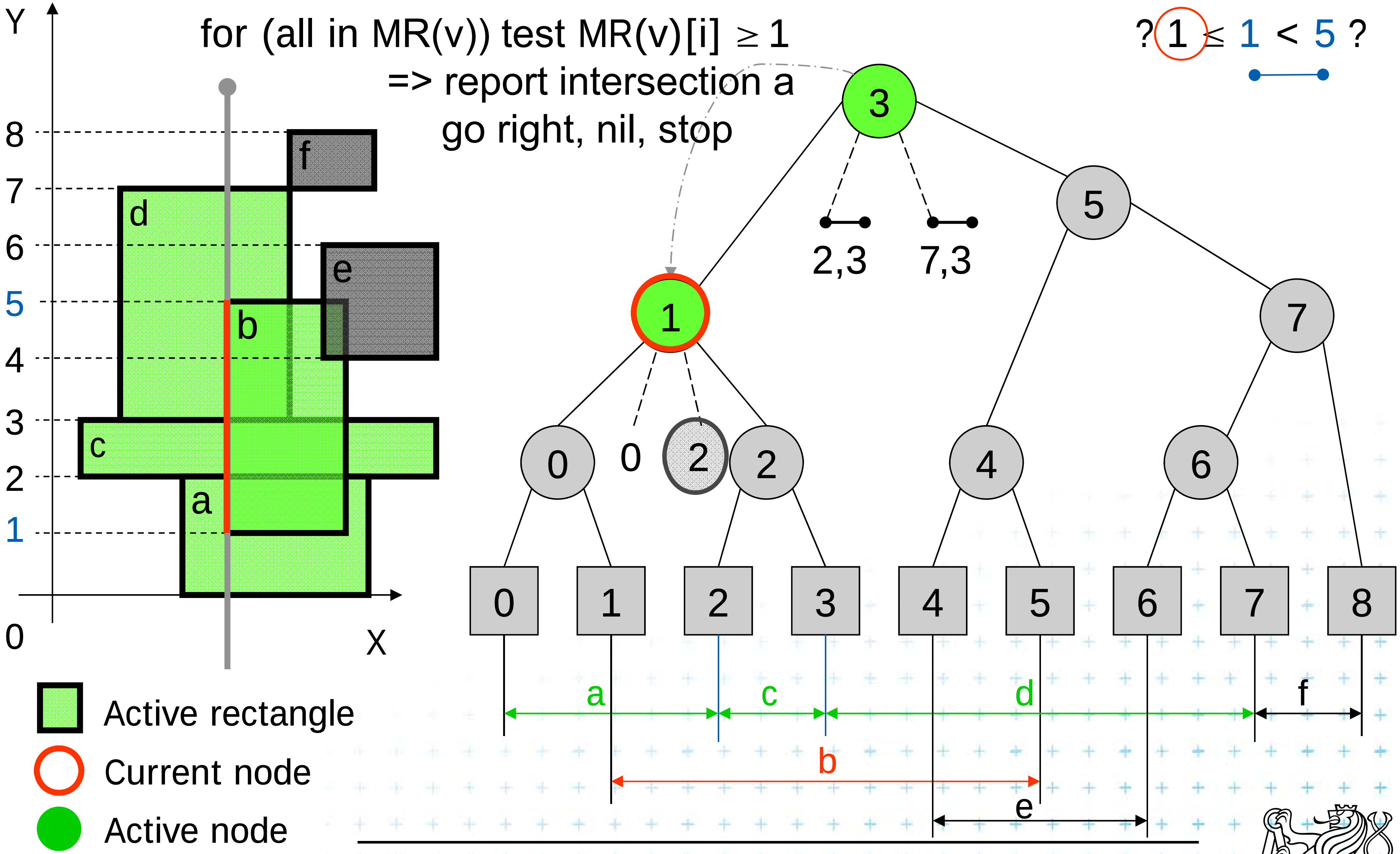


Insert [1,5] a) Query Interval 2/2

H(v) ≤ b < e

for (all in MR(v)) test MR(v)[i] ≥ 1
=> report intersection a
go right, nil, stop

? 1 ≤ 1 < 5 ?

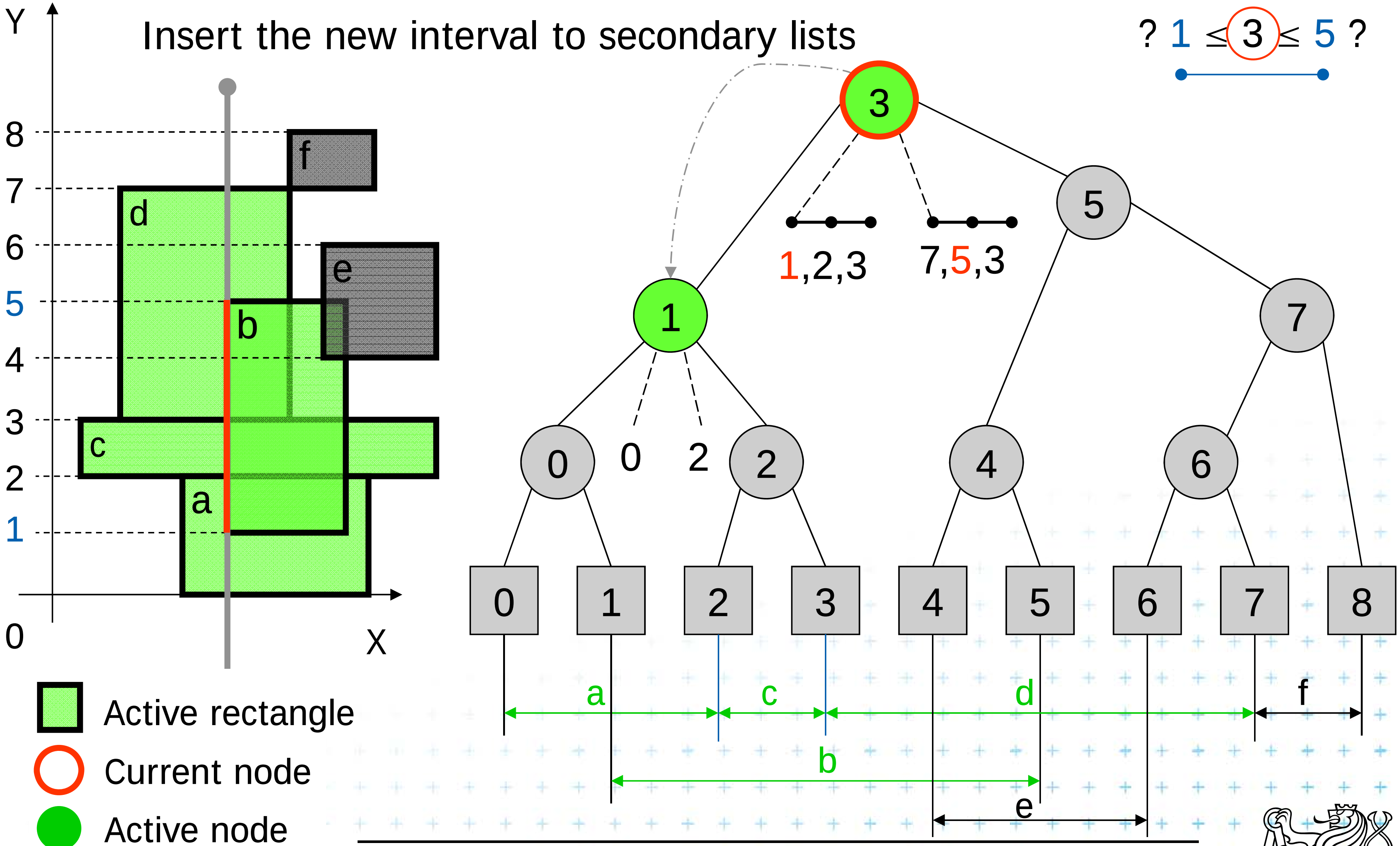


Insert [1,5] b) Insert Interval

b ≤ H(v) ≤ e

Insert the new interval to secondary lists

? 1 ≤ 3 ≤ 5 ?



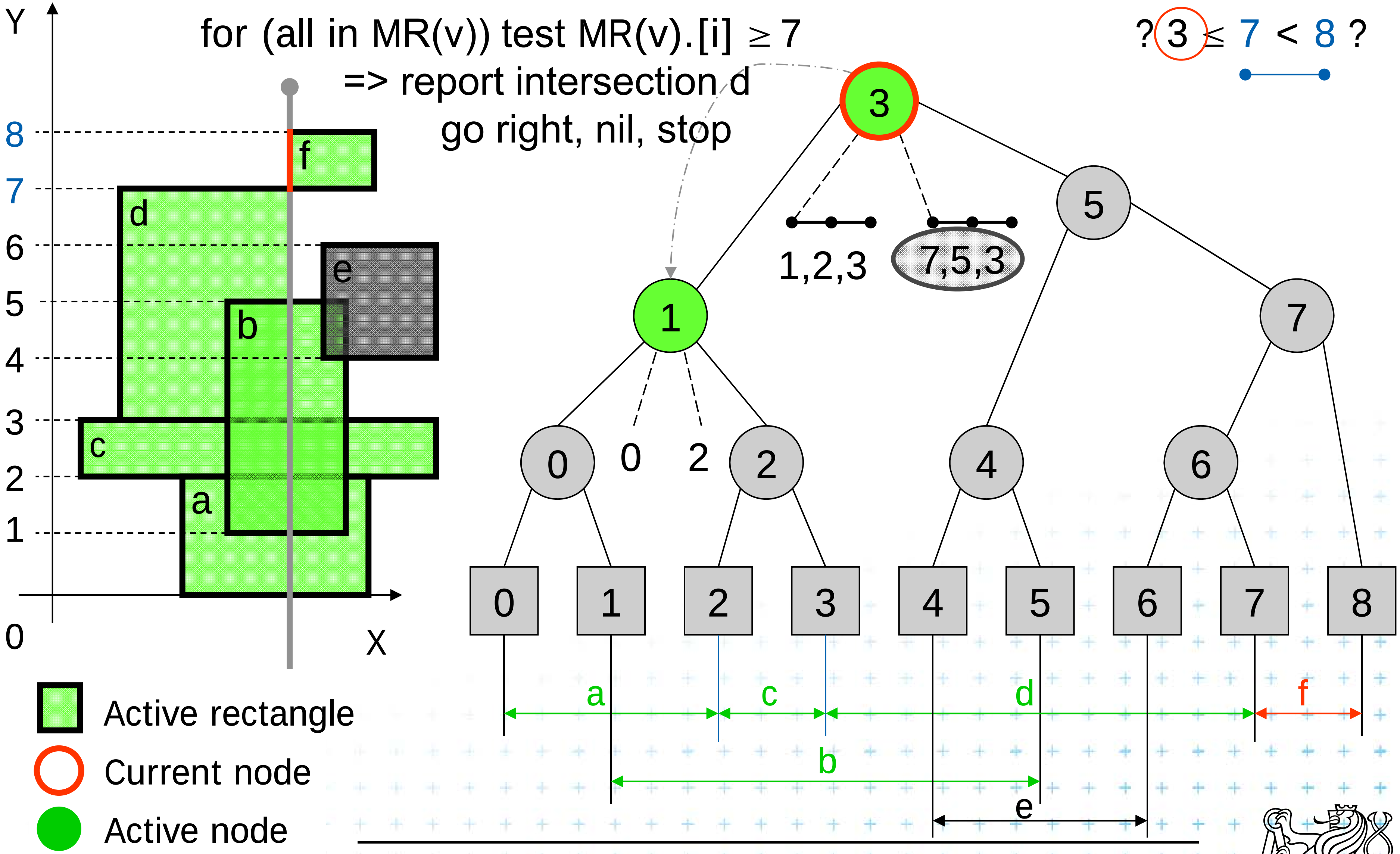
[Drtina]

Insert [7,8] a) Query Interval

H(v) ≤ b < e

for (all in MR(v)) test MR(v).[i] ≥ 7
=> report intersection d
go right, nil, stop

? 3 ≤ 7 < 8 ?

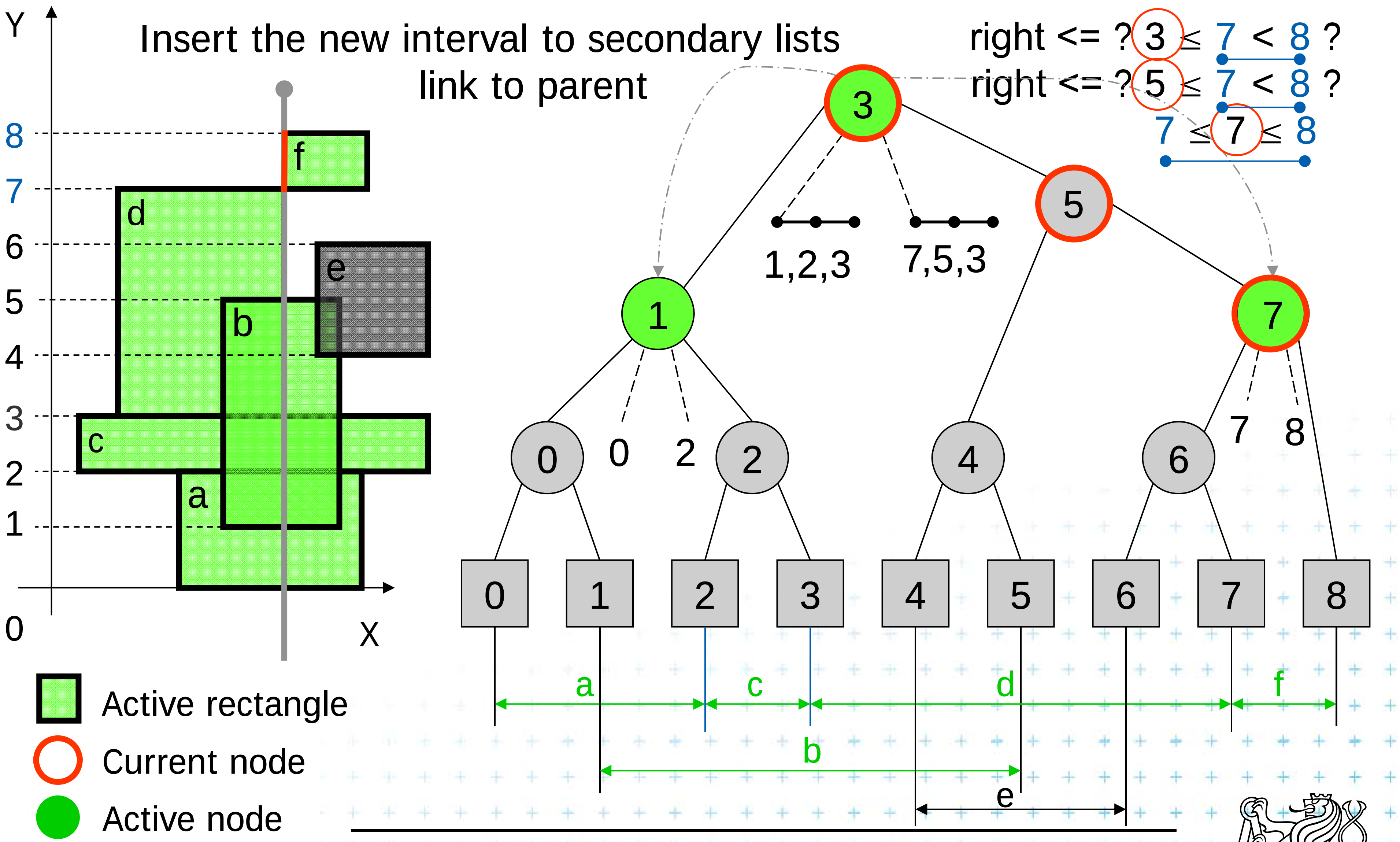


[Drtina]

Insert [7,8] b) Insert Interval

$b \leq H(v) \leq e$

Insert the new interval to secondary lists
link to parent



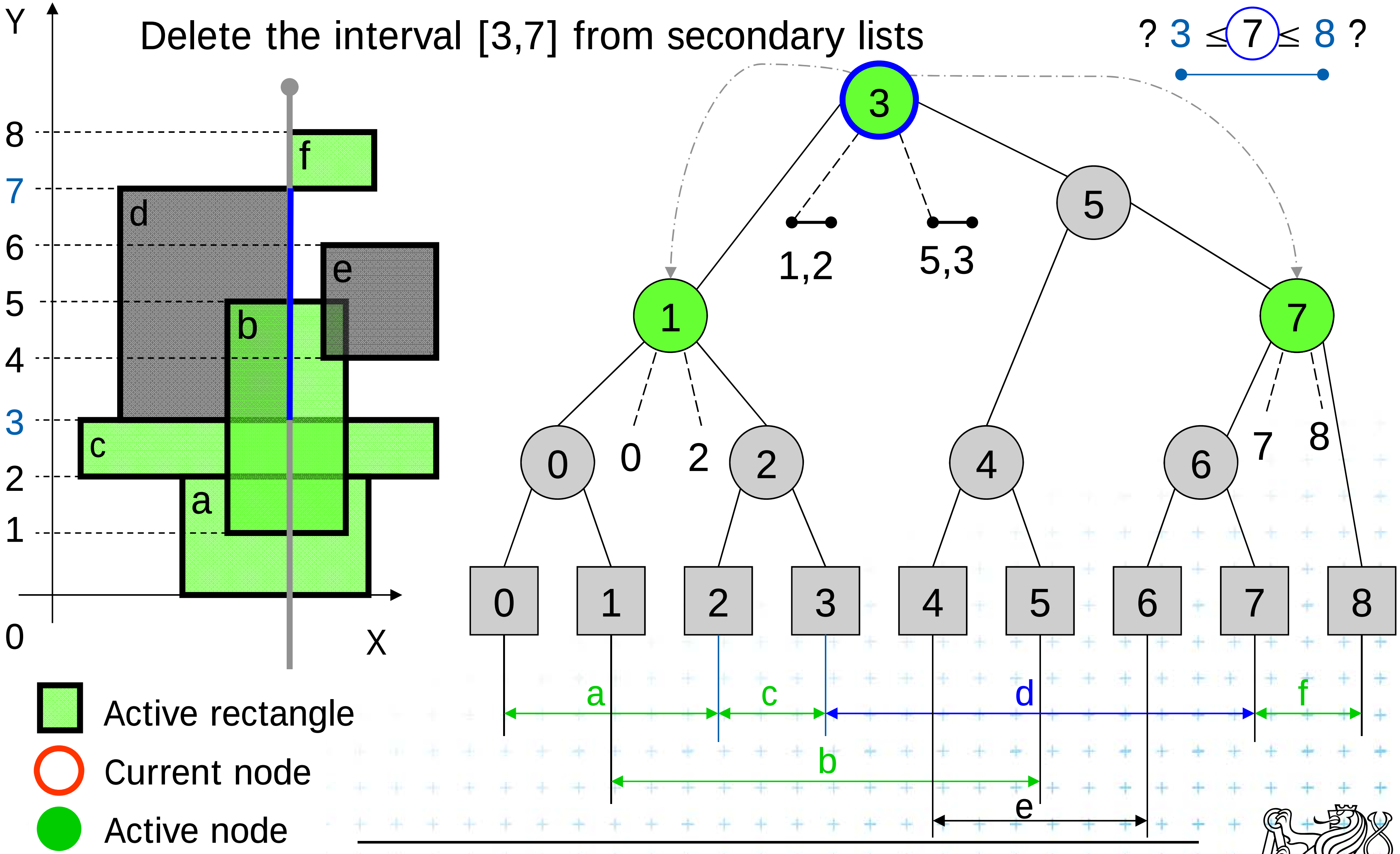
[Drtina]

Delete [3,7] Delete Interval

$b \le H(v) \le e$

Delete the interval [3,7] from secondary lists

? 3 ≤ 7 ≤ 8 ?

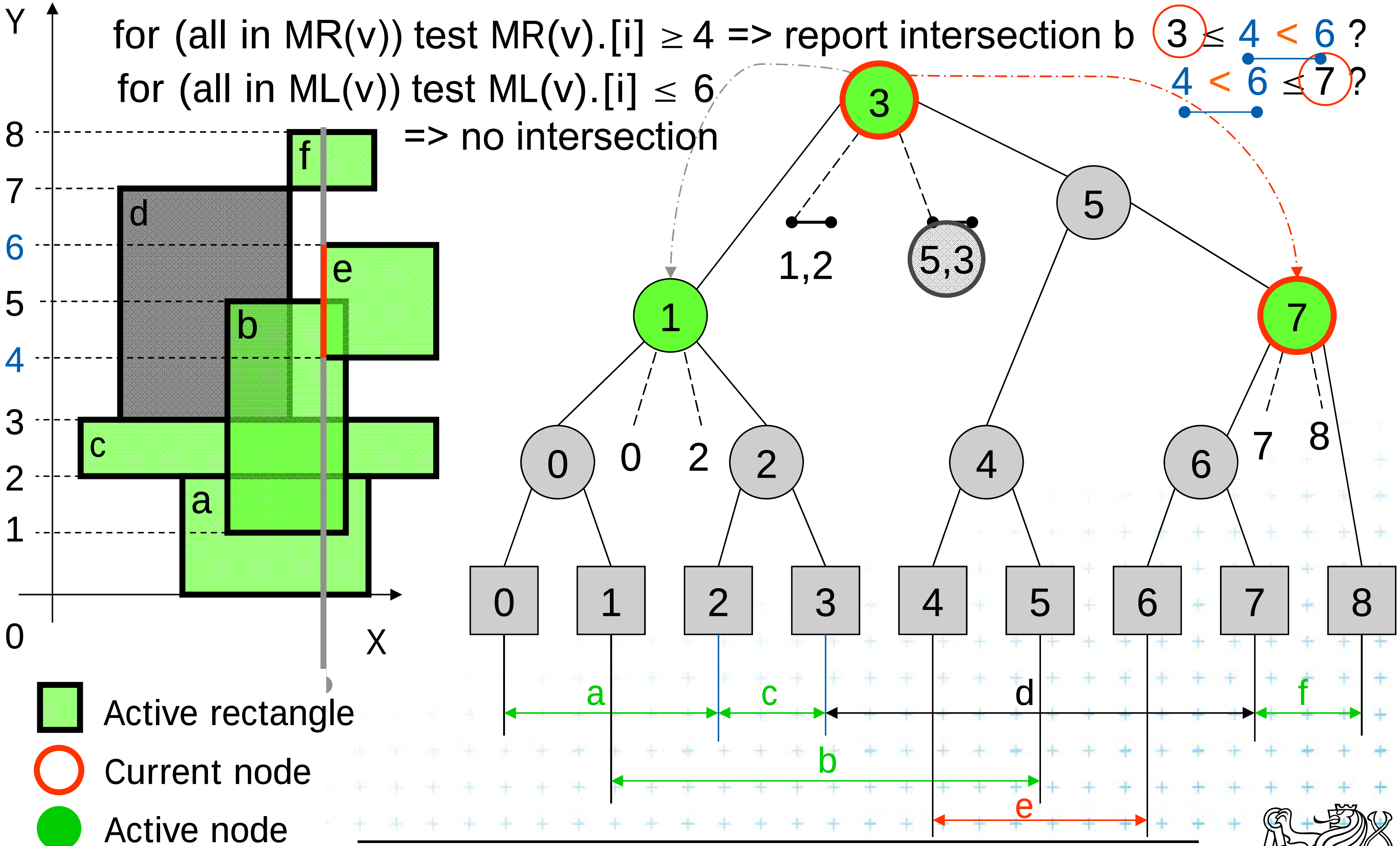


Insert [4,6] a) Query Interval

$H(v) \leq b < e$

for (all in MR(v)) test MR(v).[i] $\geq$ 4 => report intersection b
for (all in ML(v)) test ML(v).[i] $\leq$ 6
=> no intersection

$3 \leq 4 < 6$?
$4 < 6 \leq 7$?

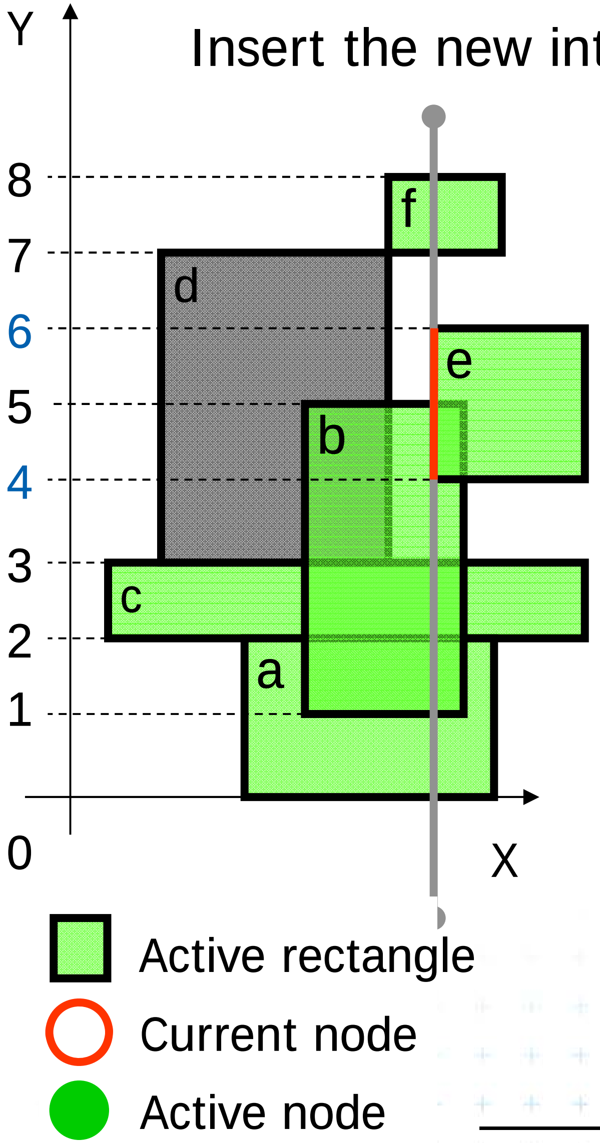


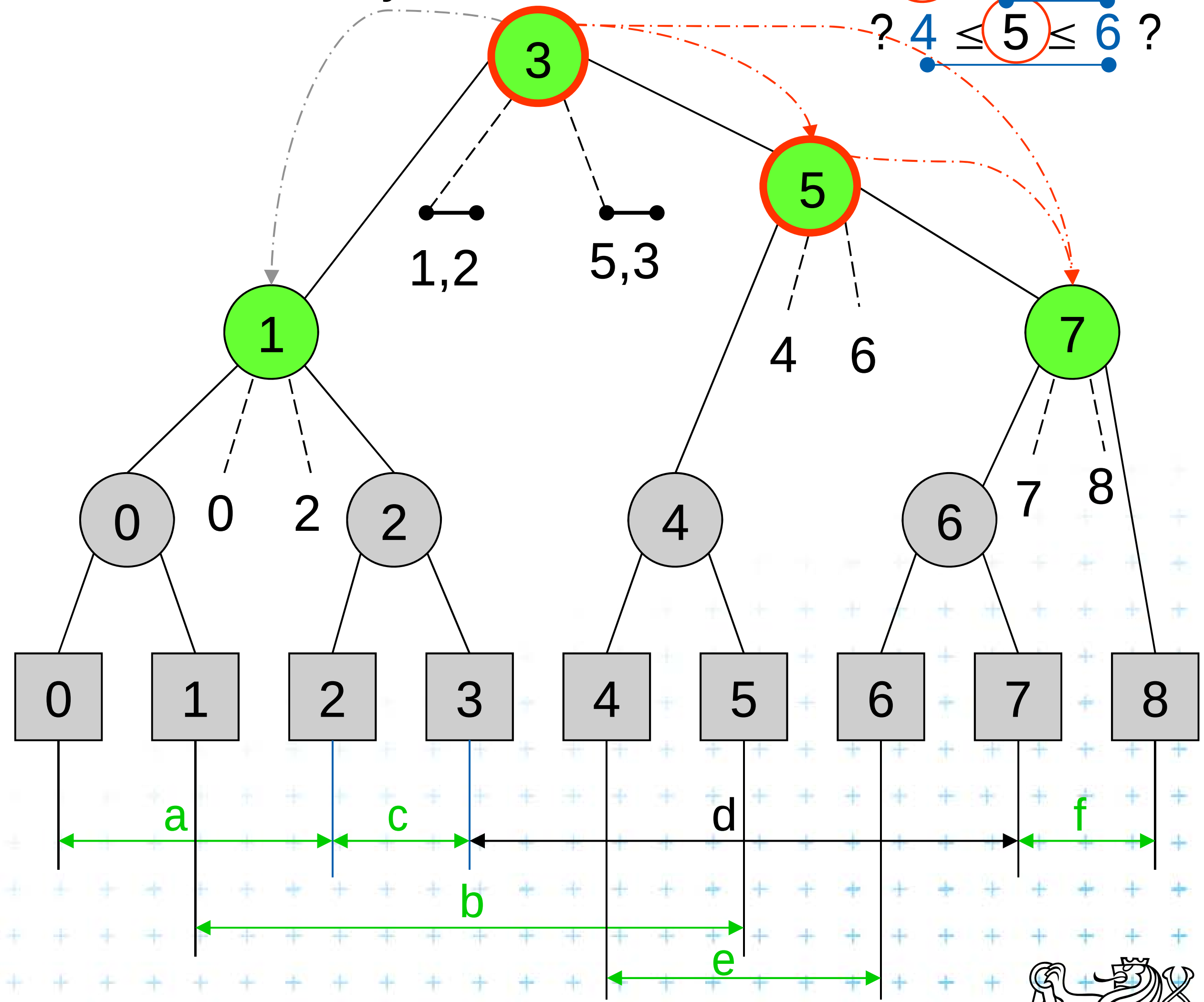
[Drtina]

Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists





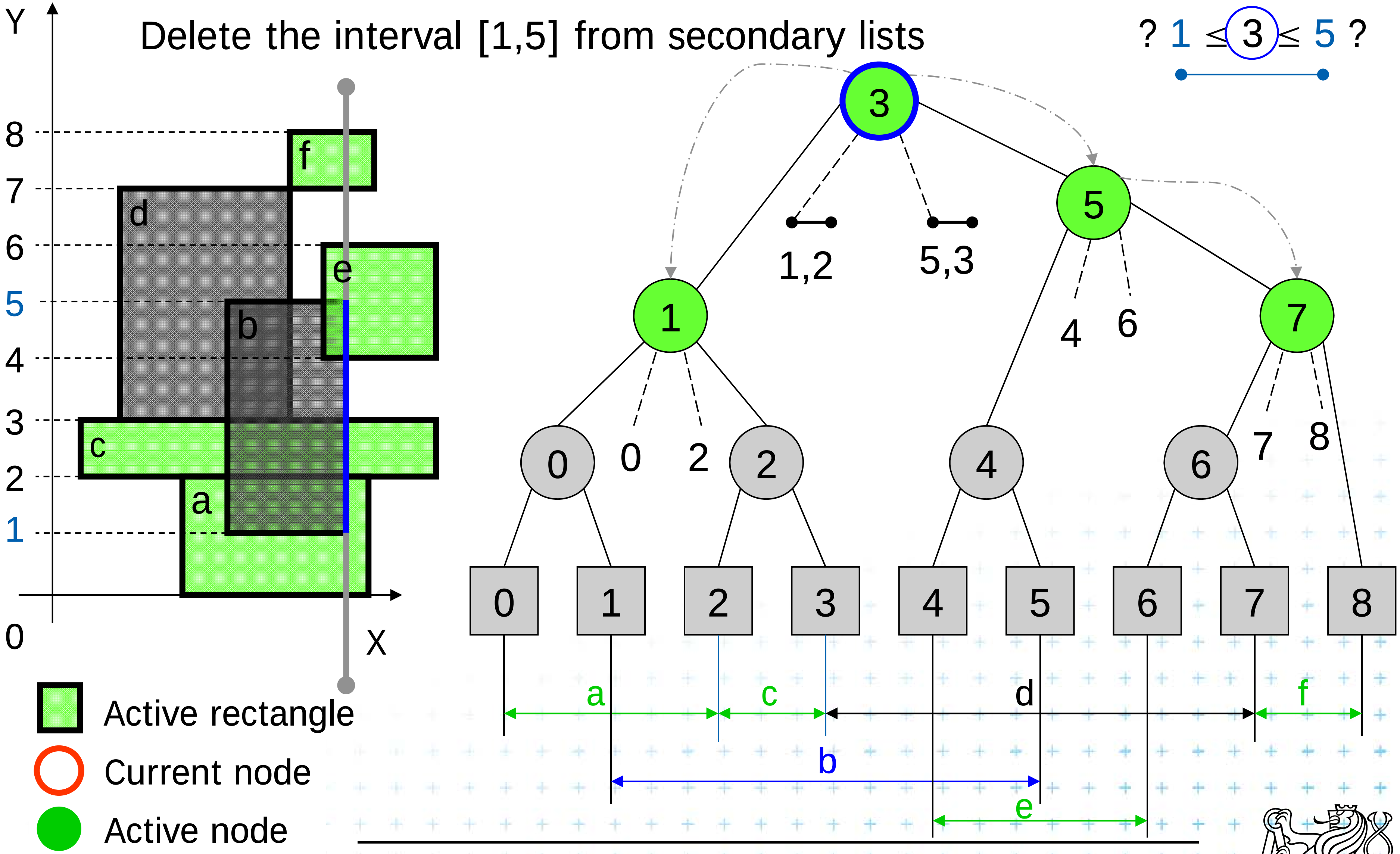
[Drtina]

Delete [1,5] Delete Interval

$b \leq H(v) \leq e$

Delete the interval [1,5] from secondary lists

? 1 ≤ 3 ≤ 5 ?

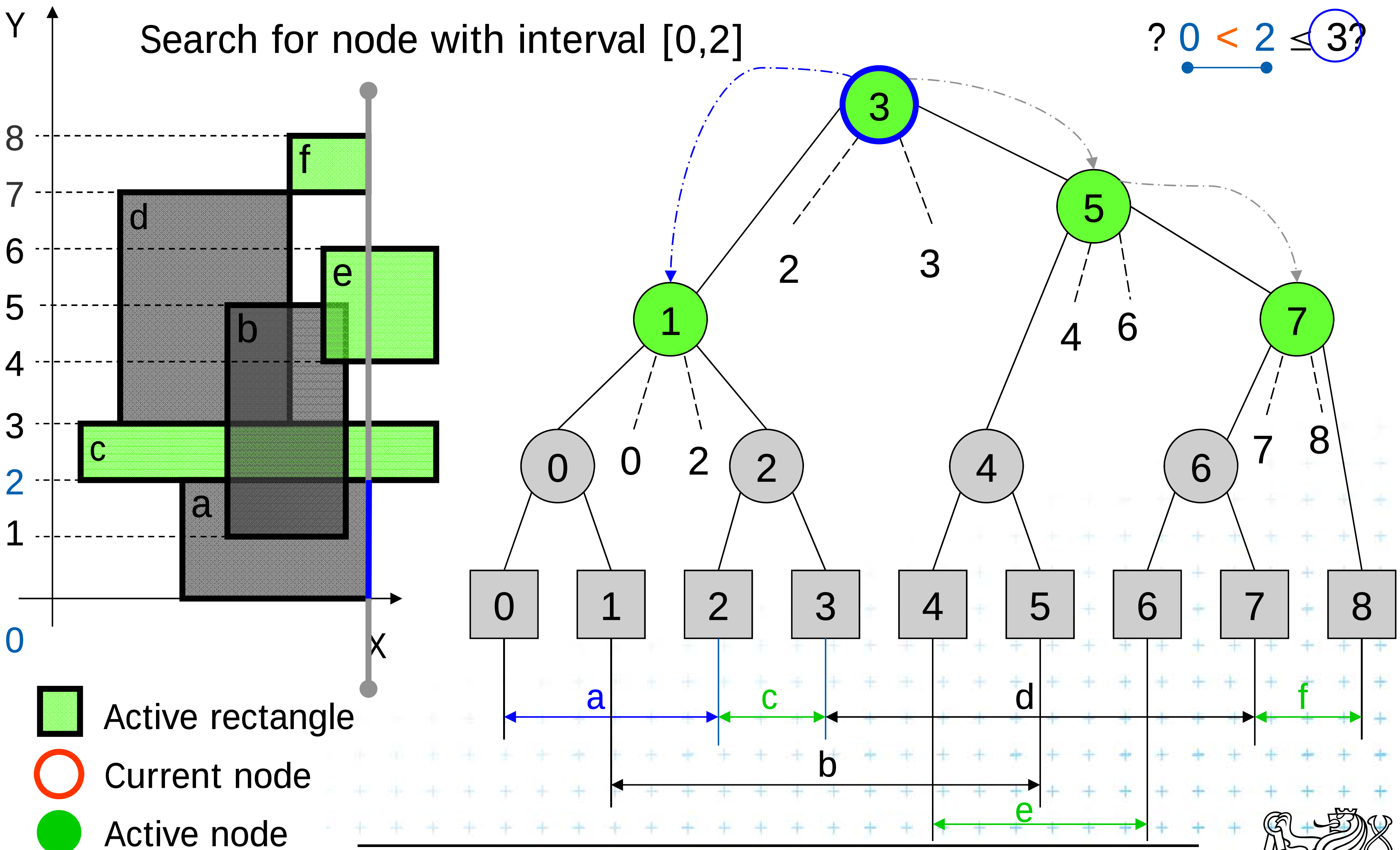


Delete [0,2] Delete Interval 1/2

b < e ≤ H(v)

Search for node with interval [0,2]

? 0 < 2 ≤ 3?

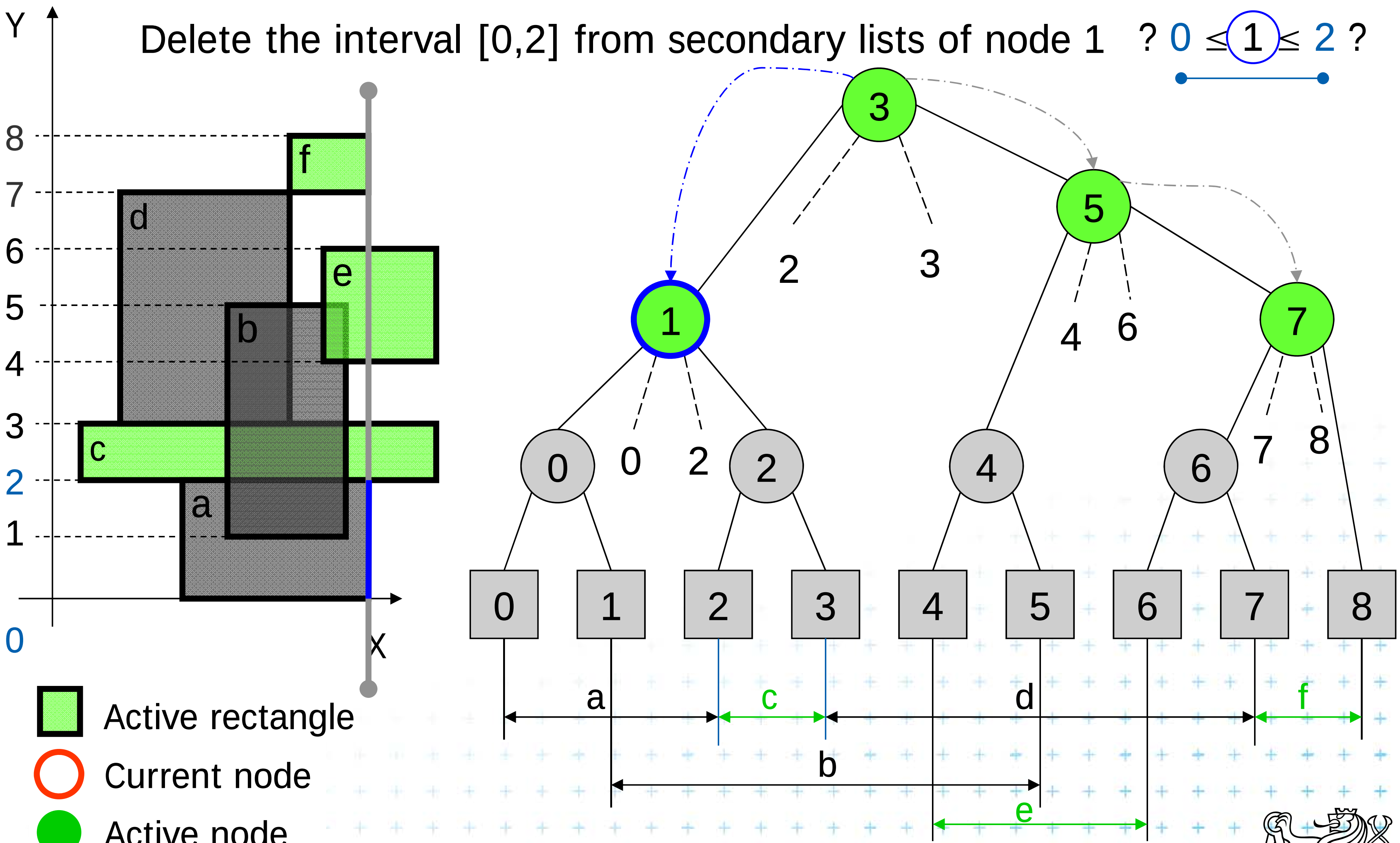


[Drtina]

Delete [0,2] Delete Interval 2/2

$b \le H(v) \le e$

Delete the interval [0,2] from secondary lists of node 1 ? $0 \le 1 \le 2$?

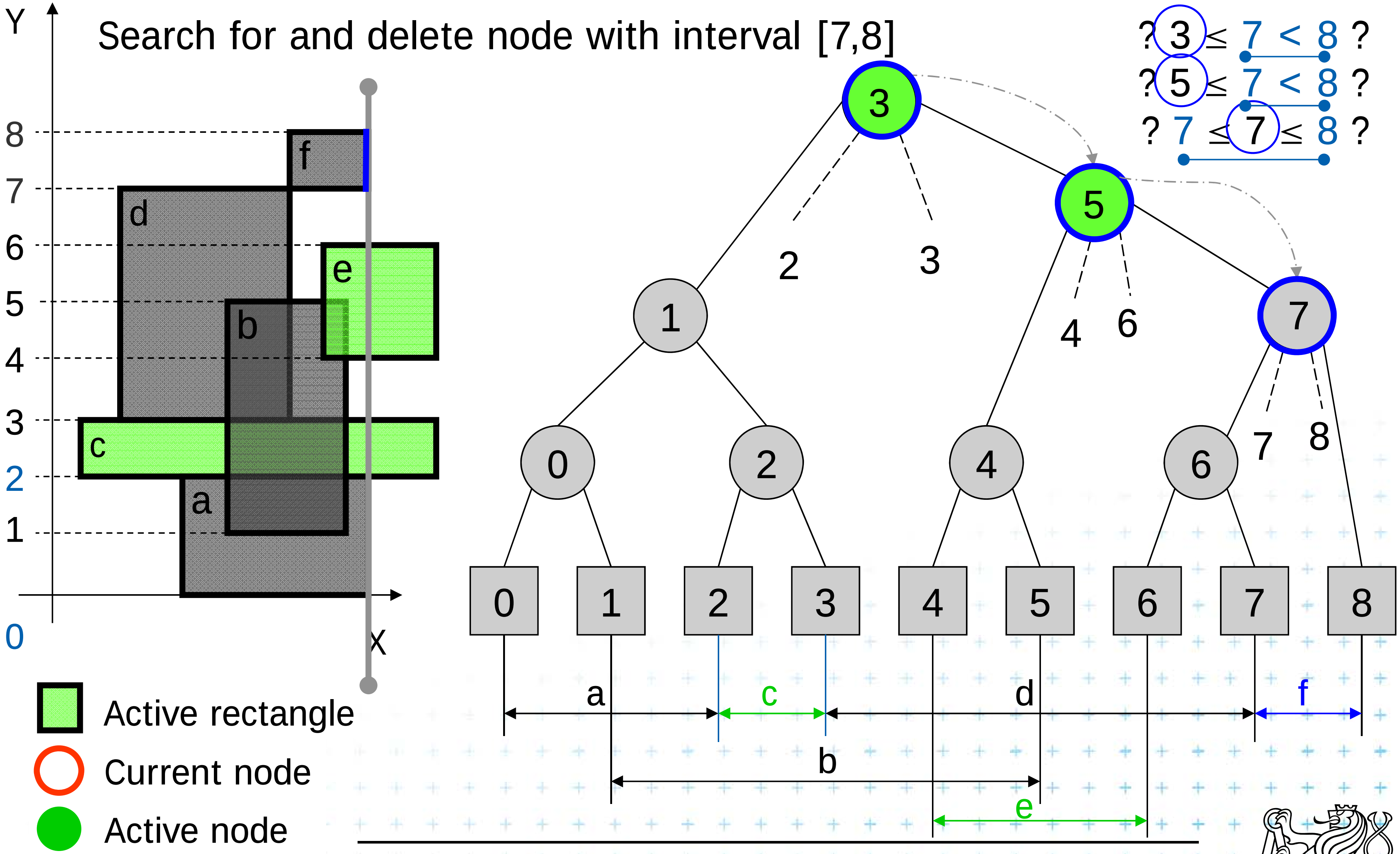


[Drtina]

Delete [7,8] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [7,8]



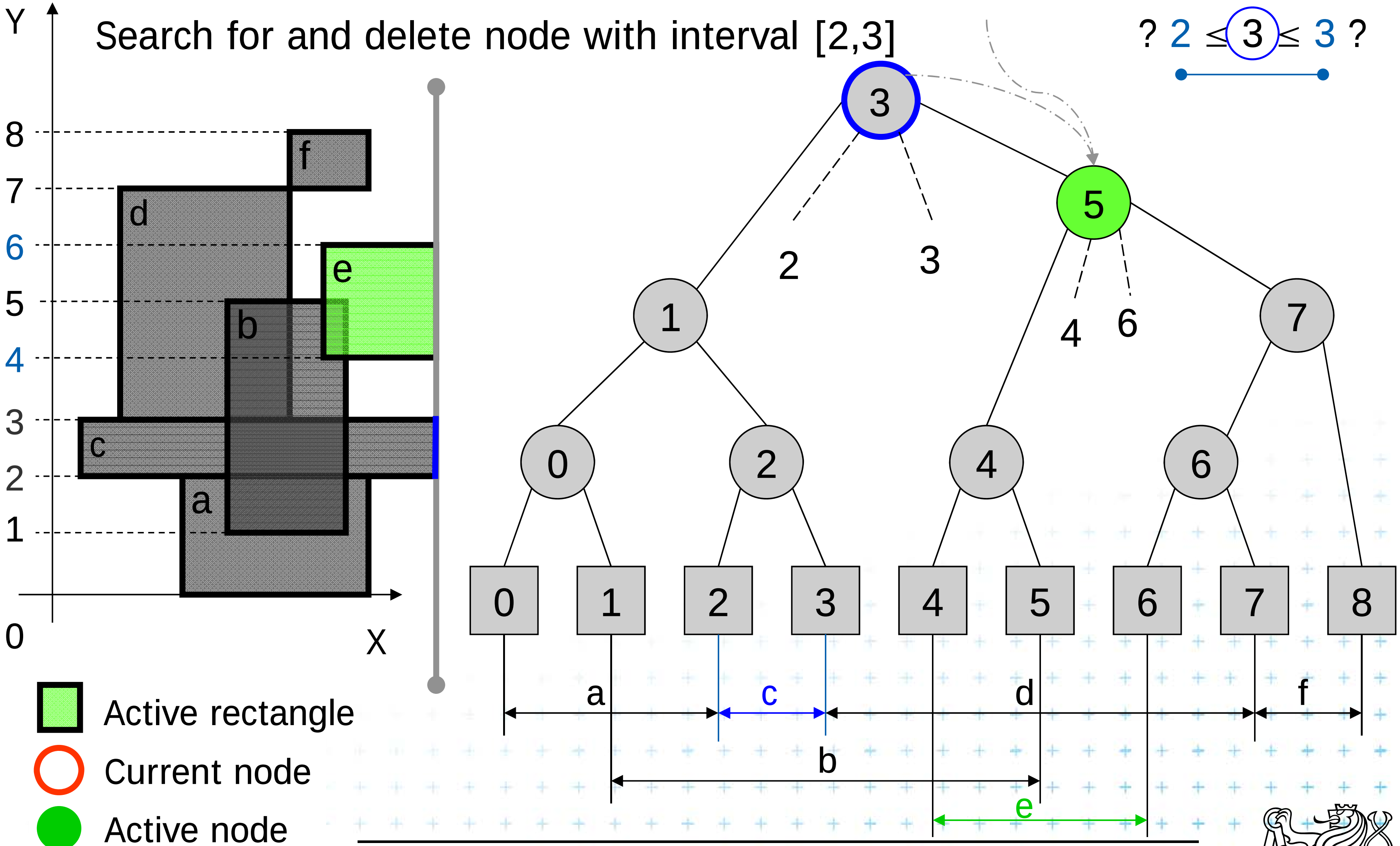
[Drtina]

Delete [2,3] Delete Interval

$b \le H(v) \le e$

Search for and delete node with interval [2,3]

? 2 ≤ 3 ≤ 3 ?



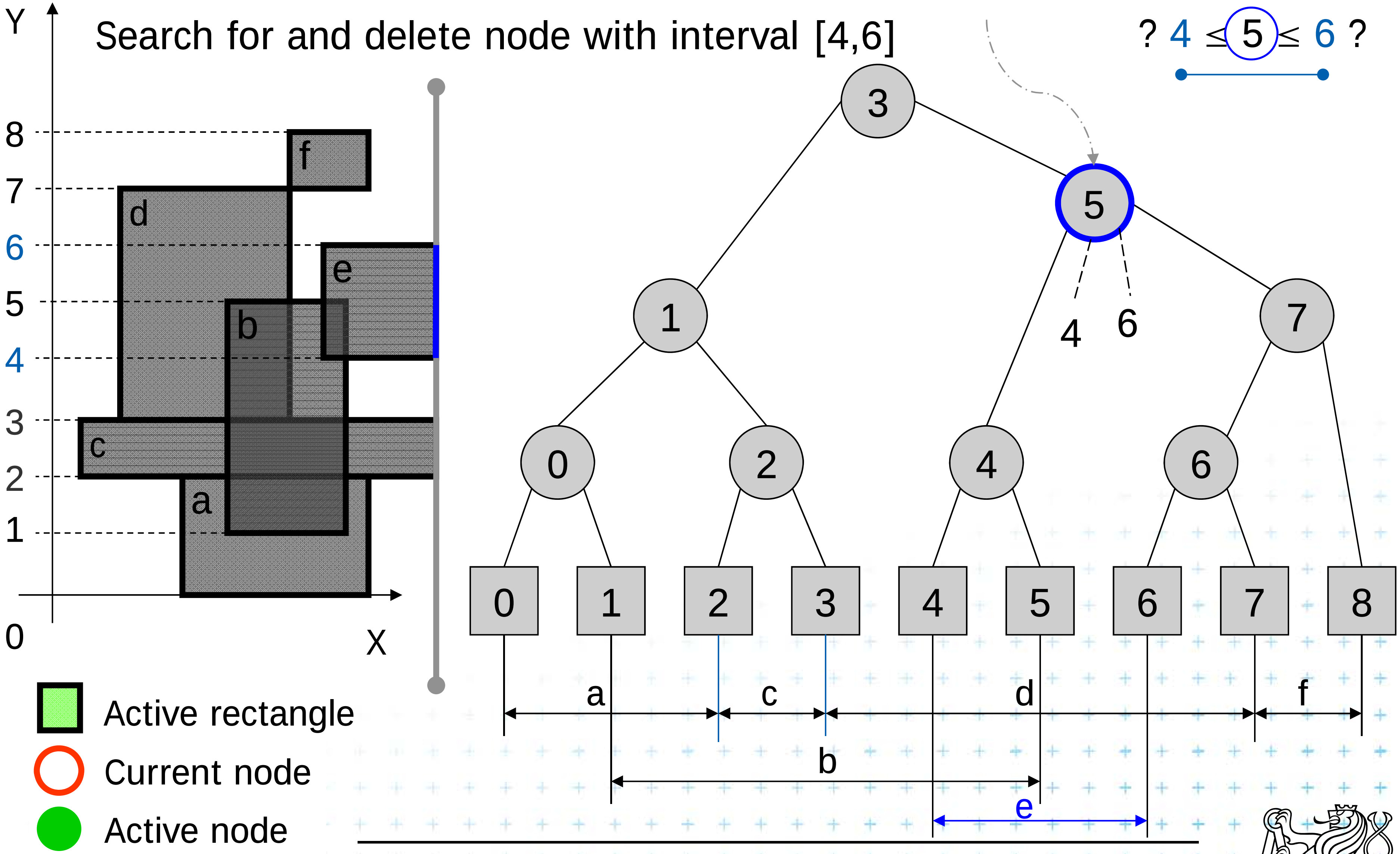
[Drtina]

Delete [4,6] Delete Interval

$b \le H(v) \le e$

Search for and delete node with interval [4,6]

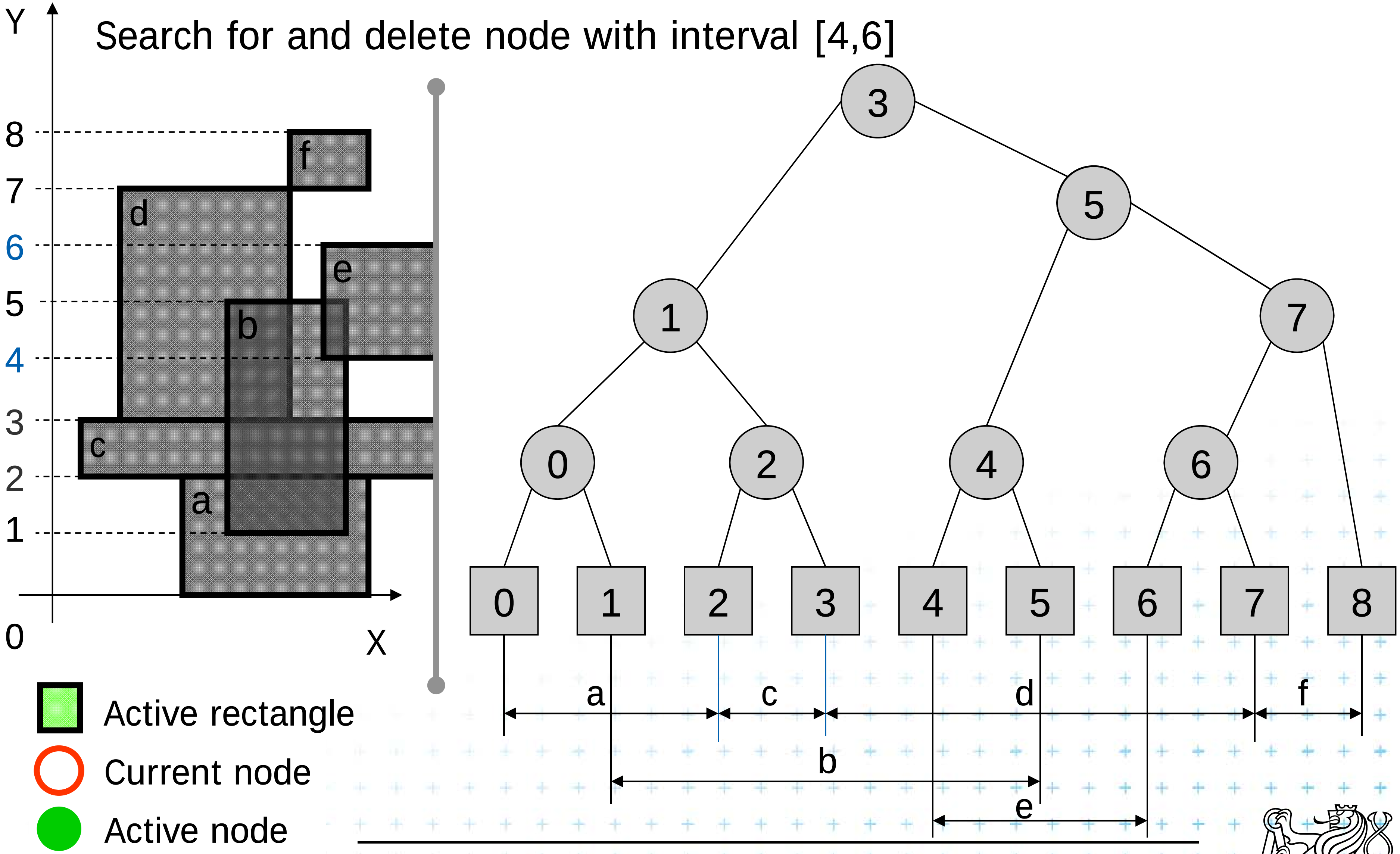
? 4 ≤ 5 ≤ 6 ?



[Drtina]

Empty tree

Search for and delete node with interval [4,6]



[Drtina]

Complexities of rectangle intersections

- *n* rectangles, *s* intersected pairs found
- O(*n* log *n*) preprocessing time to separately sort
  - x-coordinates of the rectangles for the plane sweep
  - the y-coordinates for initializing the interval tree.
- The plane sweep itself takes O(*n* log *n* + *s*) time, so the overall time is O(*n* log *n* + *s*)
- O(*n*) space
- This time is optimal for a decision-tree algorithm (i.e., one that only makes comparisons between rectangle coordinates).

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 5. 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

[Drtina] Tomáš Drtina: Intersection of rectangles. Semestral Assignment. Computational Geometry course, FEL CTU Prague, 2006

[Kukral] Petr Kukrál: Intersection of rectangles. Semestral Assignment. Computational Geometry course, FEL CTU Prague, 2006

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