DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

# WINDOWING


## PETR FELKEL

**FEL CTU PRAGUE**

**felkel@fel.cvut.cz**

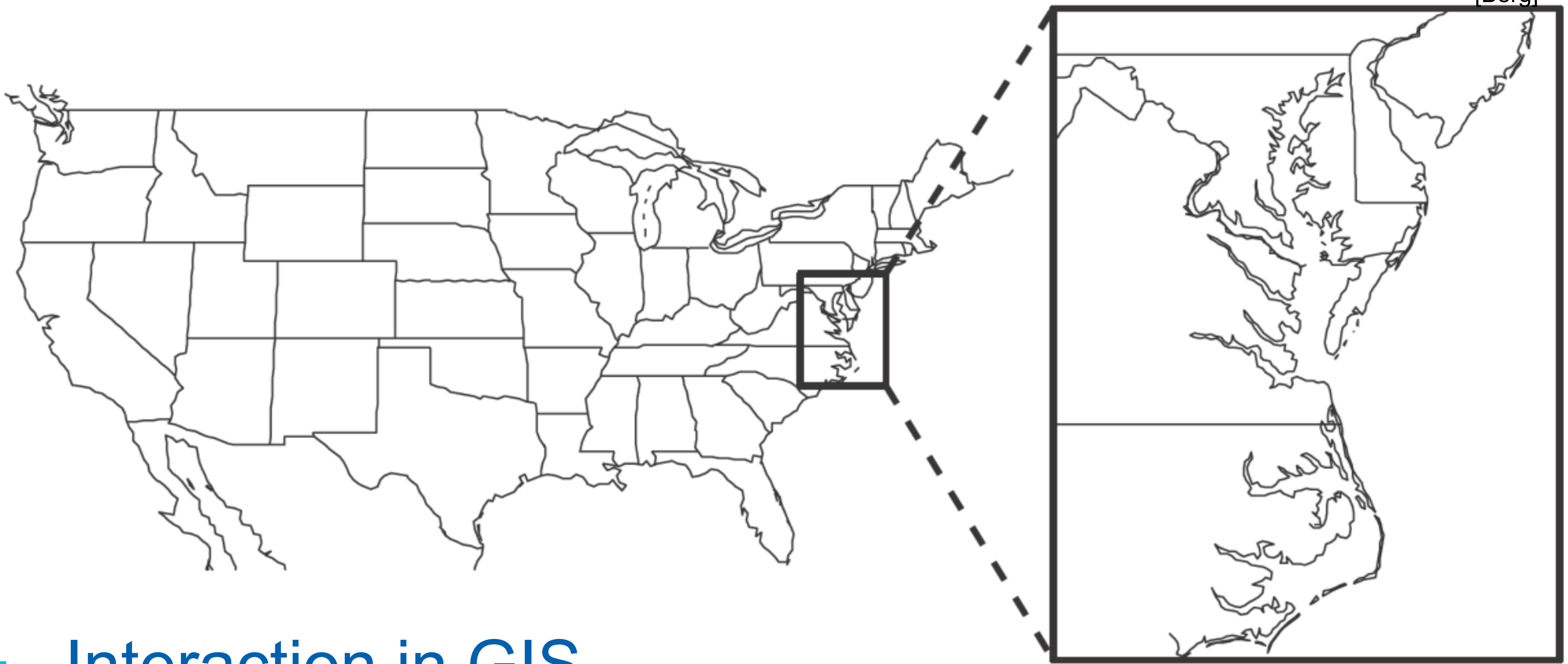
**https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start**


**Based on [Berg], [Mount]**

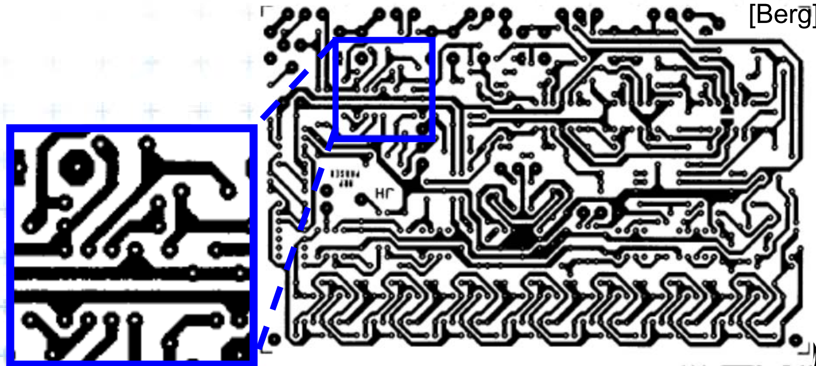
**Version from 15.12.2016**

# Windowing queries - examples

[Berg]



[Berg]



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

[Vakken]

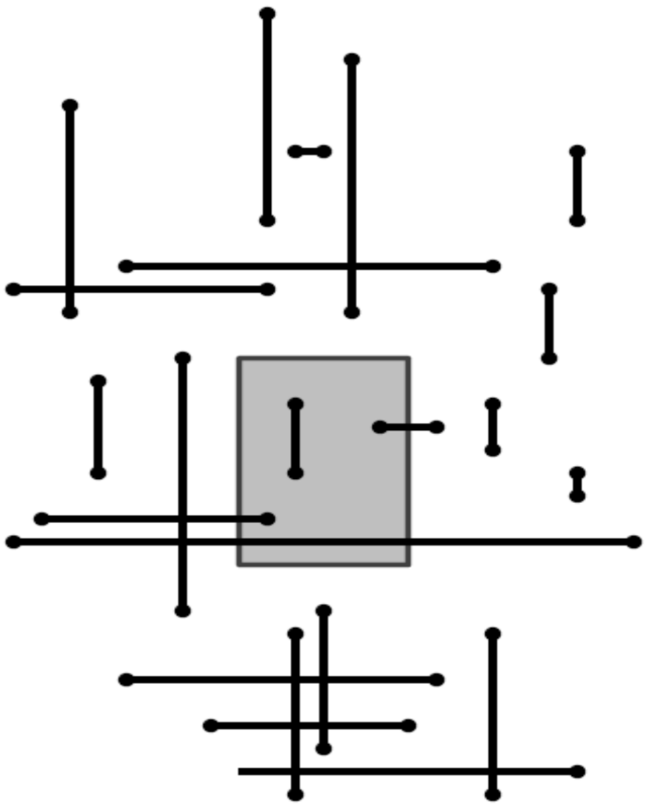
# Windowing versus range queries

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

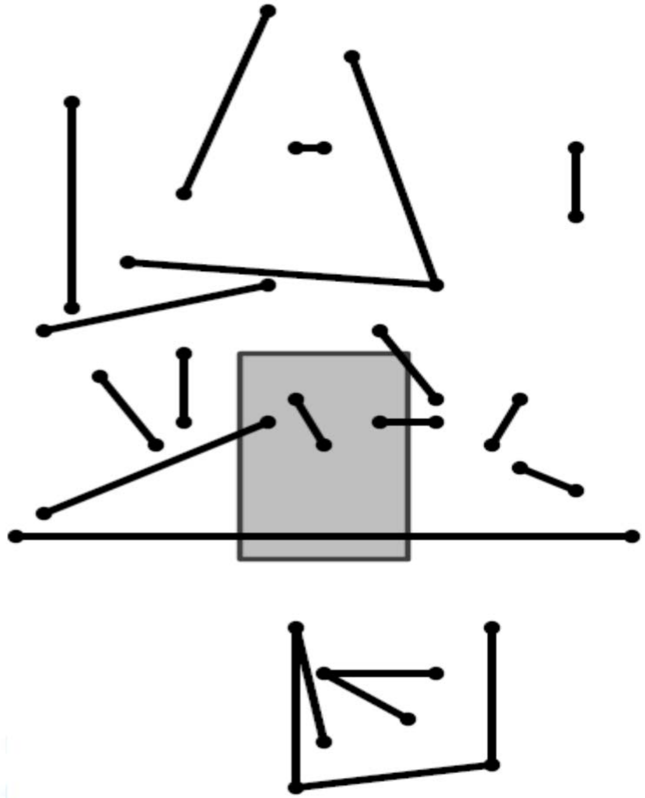
- The goal for both:
  Preprocess the data into a data structure
  - so that the objects intersected by the query rectangle can be reported efficiently

DCGI

# Windowing queries on line segments



1. Axis parallel line segments



2. Arbitrary line segments (non-crossing)

[Vakken]

# Talk overview

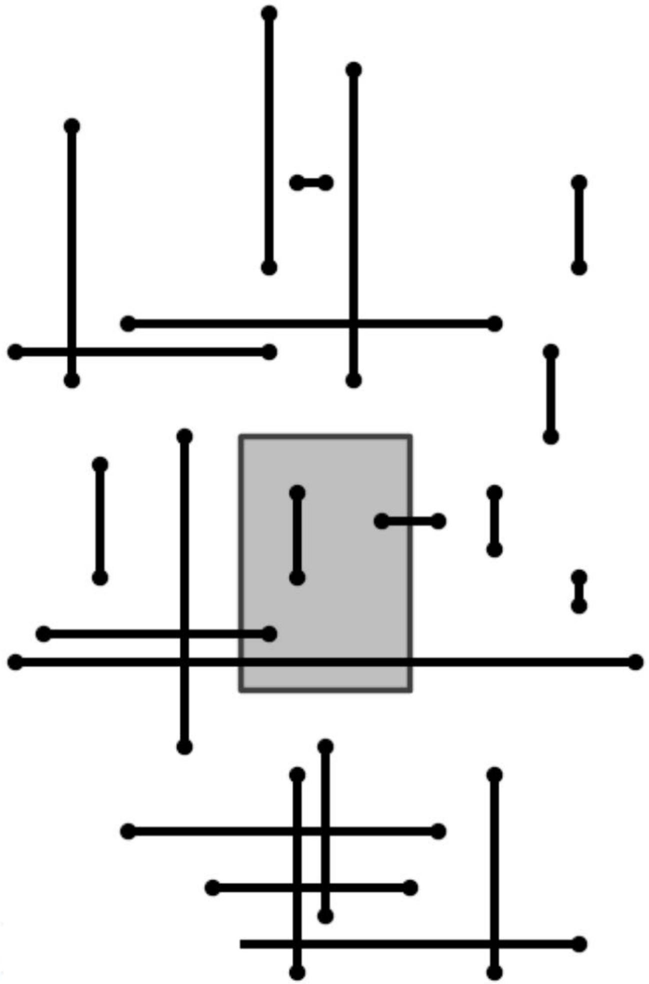
1. Windowing of axis parallel line segments in 2D
   – 3 variants of *interval tree – IT in x-direction*
   – Differ in storage of segment end points $M_L$ and $M_R$

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

DCGI

# 1. Windowing of axis parallel line segments

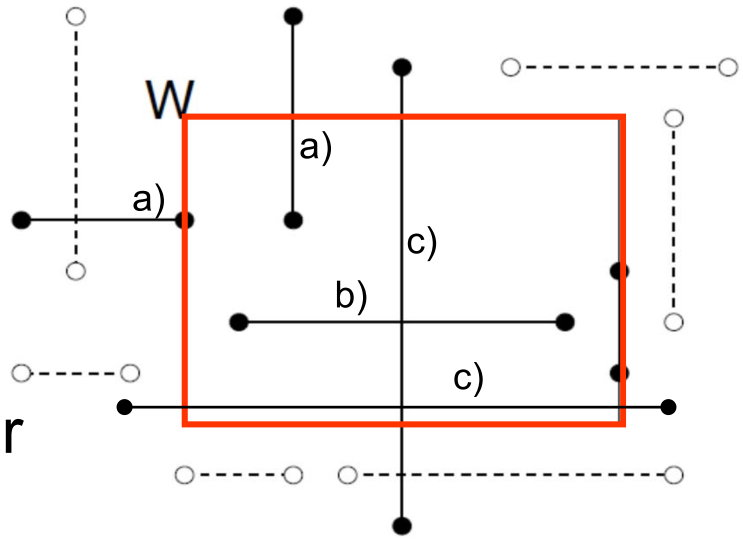


[Vakken]

# 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) 1 endpoint in
  b) 2 end points in – Included
  c) no end point in – Cross over

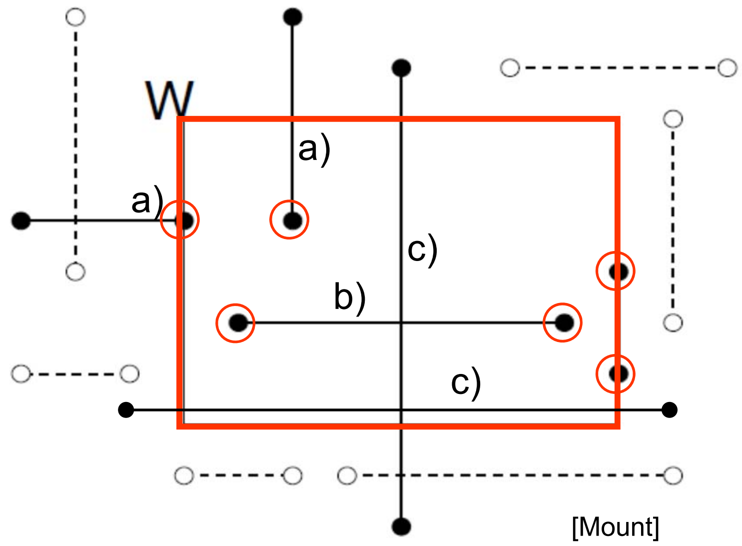



[Mount]

# Line segments with 1 or 2 points inside

a) 1 point inside

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



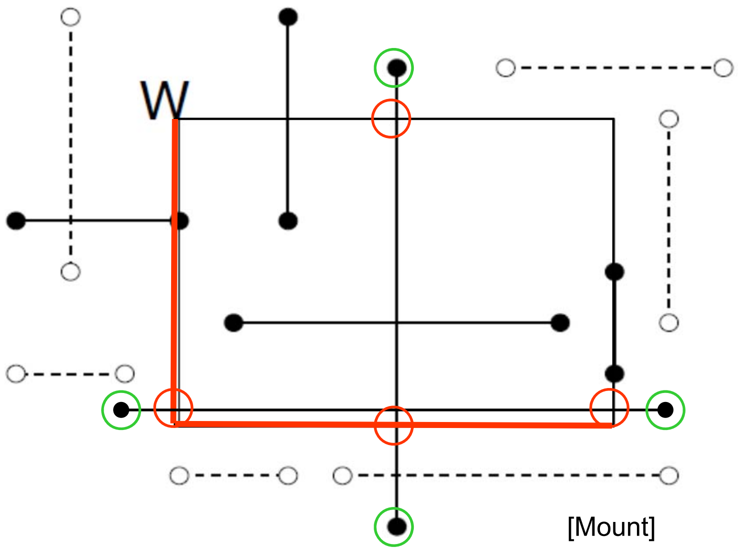

b) 2 points inside – as a) 1 point inside

- Avoid reporting twice

  1. Mark segment when reported (clear after the query)
  2. When end point found, check the other end-point. Report only the leftmost or bottom endpoint

# Line segments that cross over the window

## c) No points inside




- Such segments not detected using end-point range tree
- Cross the boundary twice
  or
  contain one boundary edge
- It is enough to
  detect segments intersected by the left and bottom boundary edges (not having end point inside)
- For left boundary: Report the segments intersecting vertical query *line segment* (1/ii.)
- Let's discuss vertical query *line* first (1/i.)
- Bottom boundary is rotated 90°

# Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   i. Line stabbing (standard *IT* with *sorted lists*)
   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
   – *segment tree*

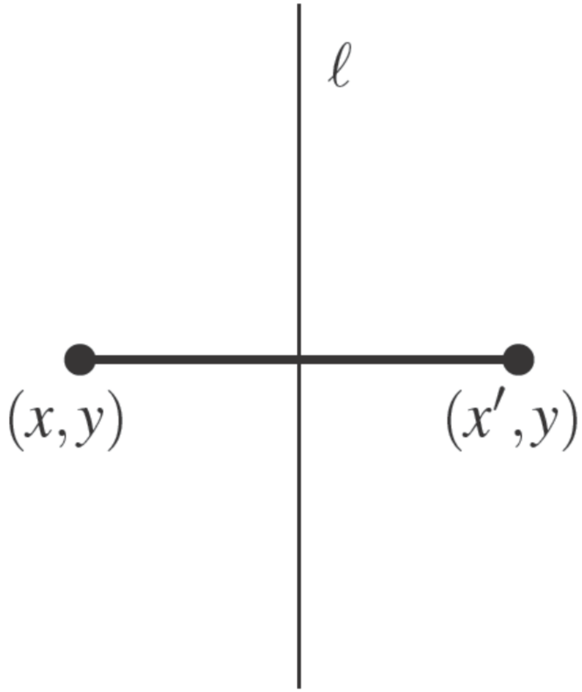
# i. Segment intersected by vertical line – 1D

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

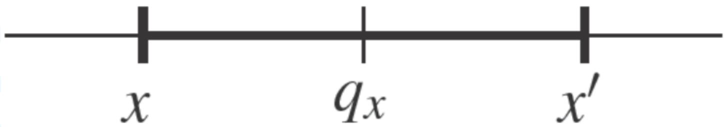
  Report the segments stabbed by a vertical line

  = 1 dimensional problem

  (ignore y coordinate)




⇒ Report the interval containing query point $q_x$

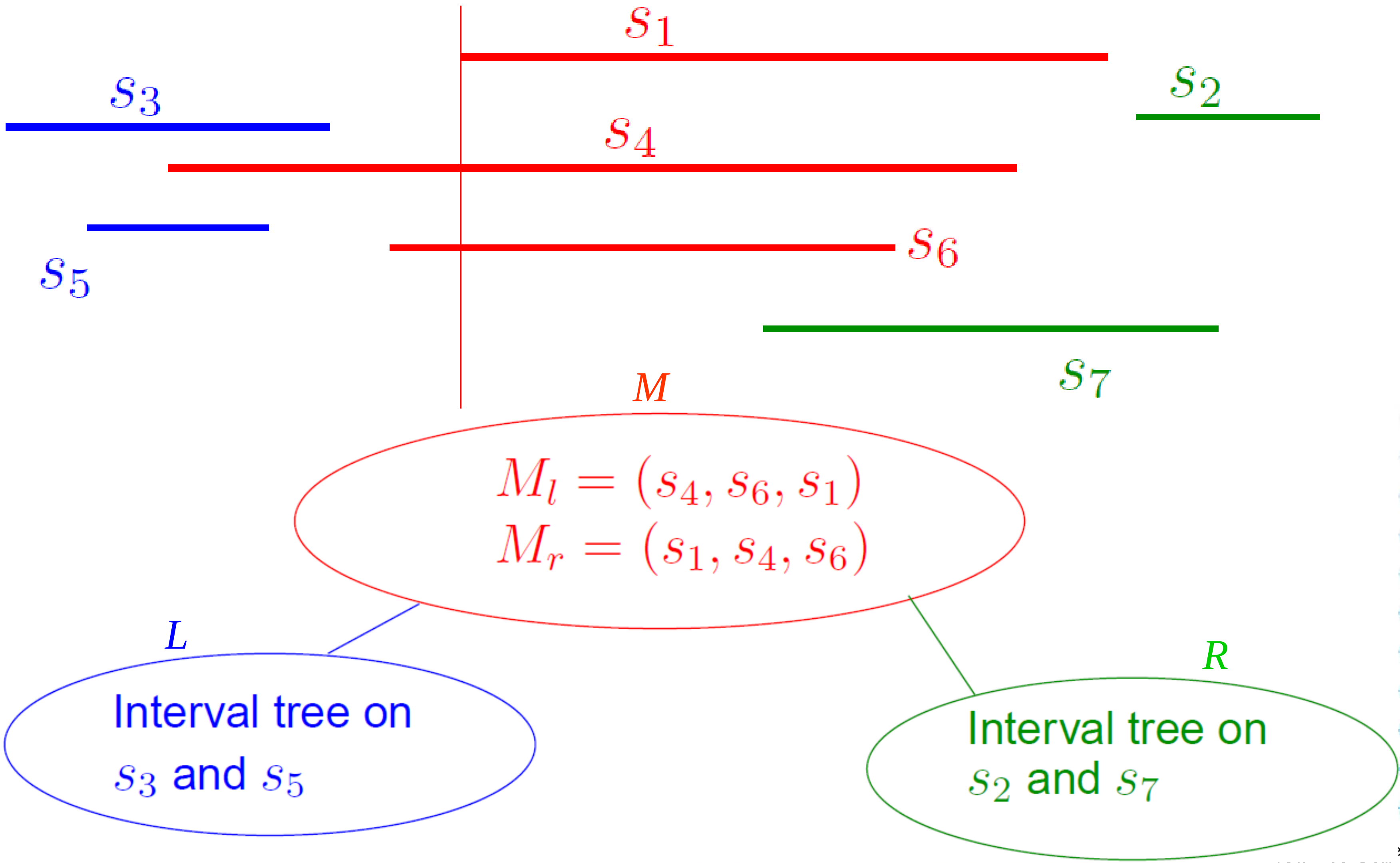



DS: Interval tree with sorted lists

[Mount]

# Interval tree principle

(see lecture 9 - intersections)

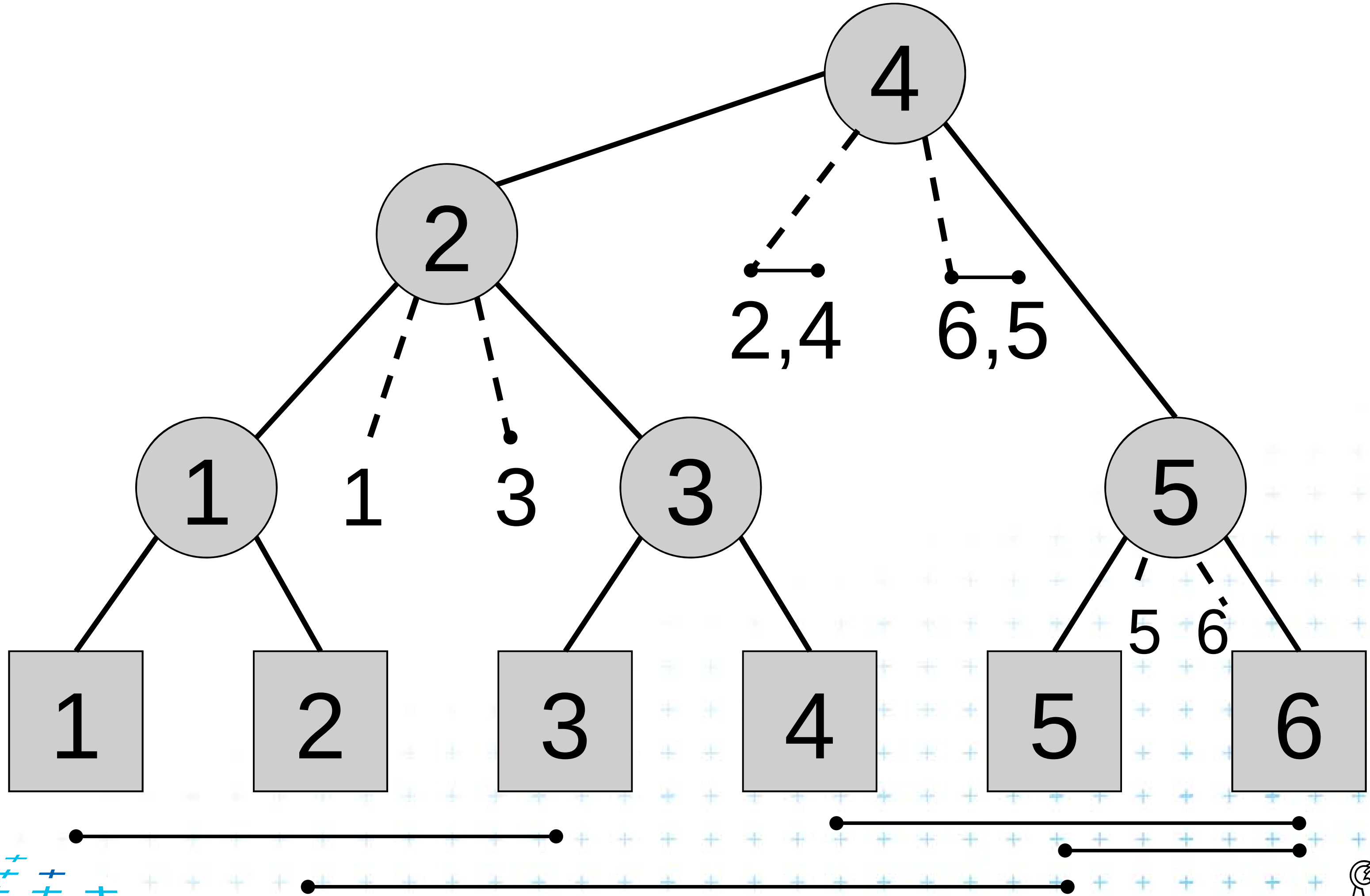



[Vigneron]

DCGI

# Static interval tree [Edelsbrunner80]

Tree over sorted segment end-points

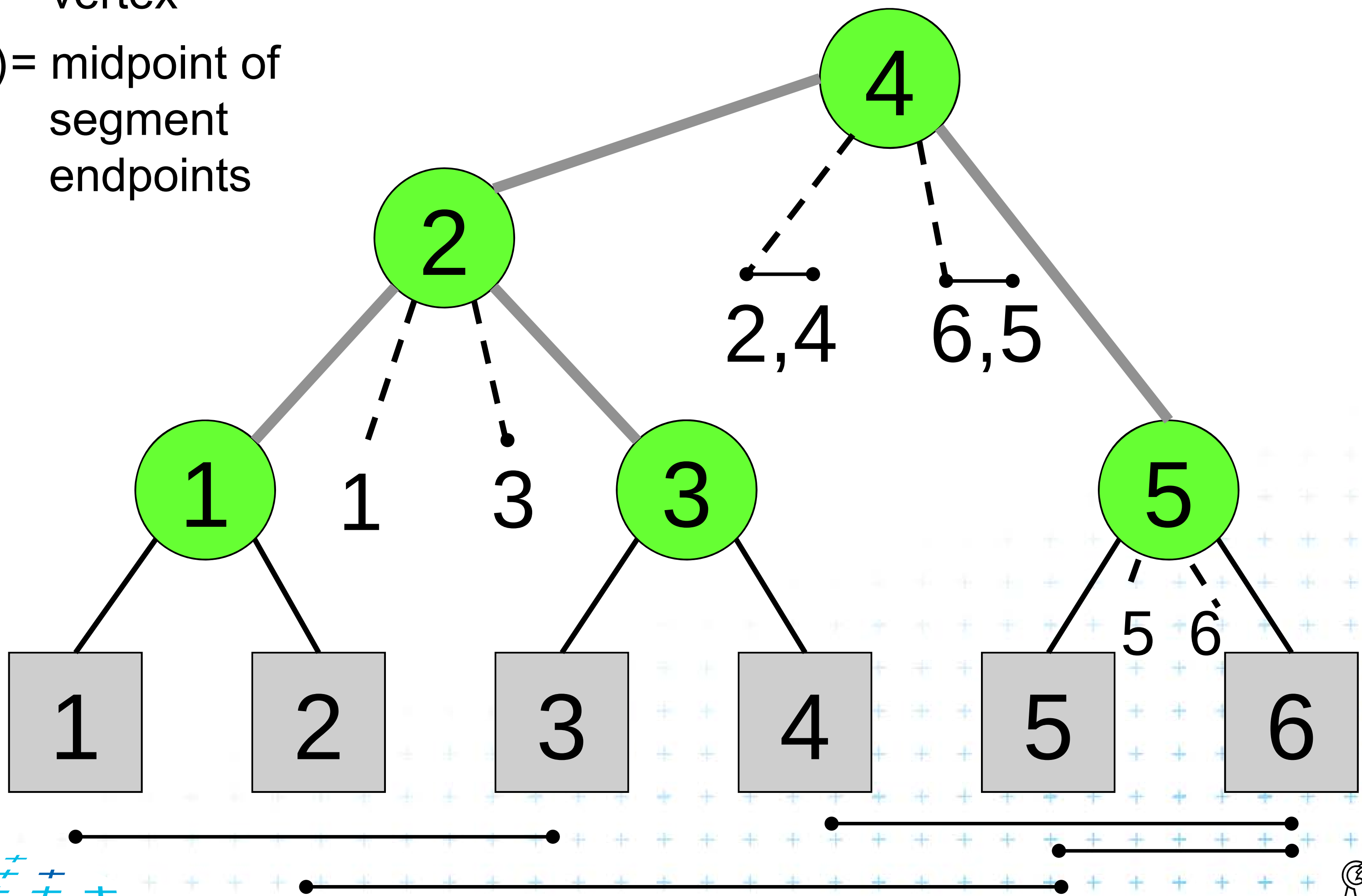



[Kukral]

# Primary structure – static tree for endpoints

v = vertex

d(v)= midpoint of segment endpoints

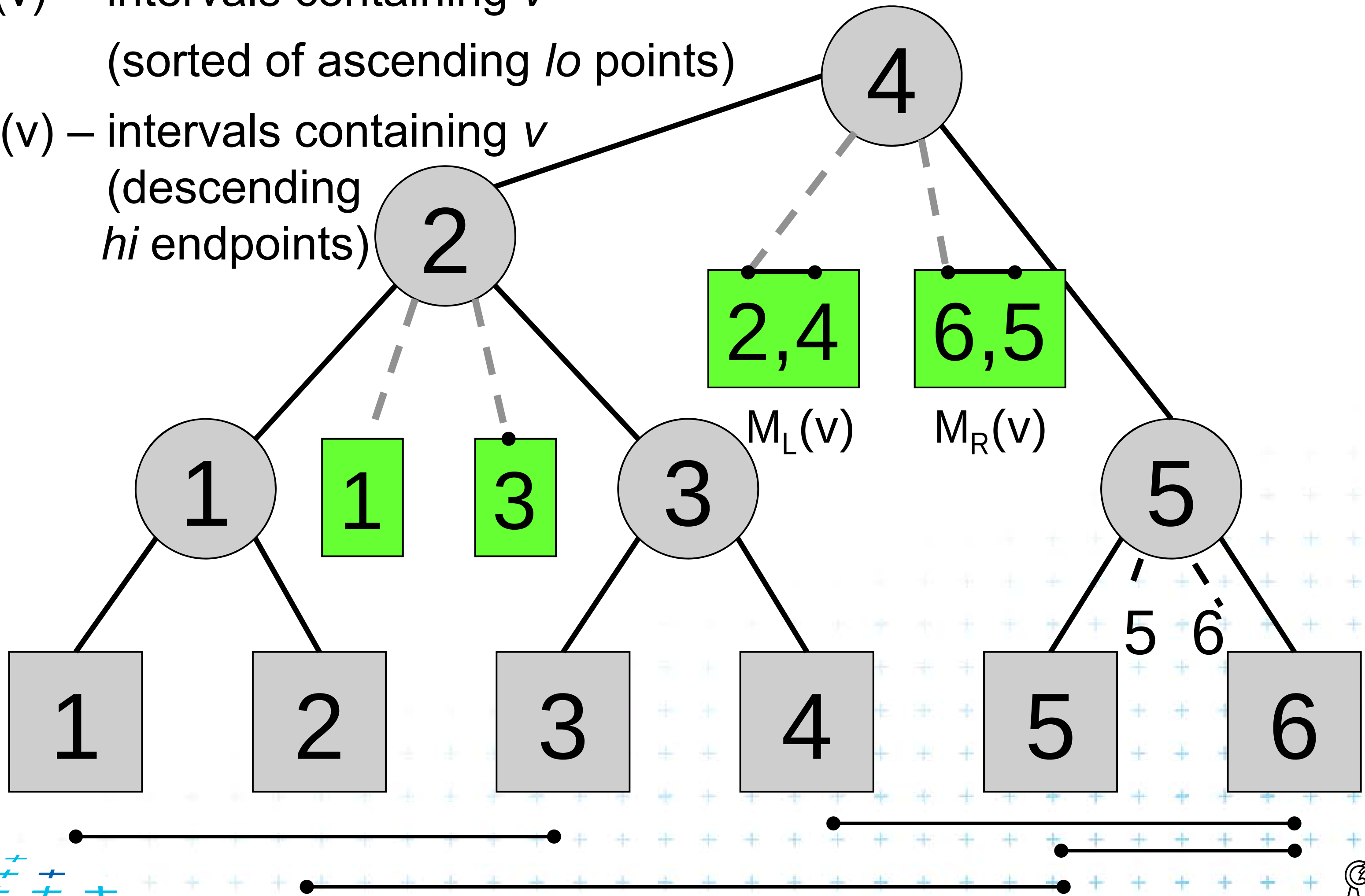



[Kukral]

# Secondary lists – sorted segments in M

ML(v) – intervals containing *v*
(sorted of ascending *lo* points)

MR(v) – intervals containing *v*
(descending
*hi* endpoints)




[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*

```
if (|S| == 0) return null                              // no more intervals
else
    xMed = median endpoint of intervals in S           // median endpoint
    L = { [xlo, xhi] in S | xhi < xMed }               // left of median
    R = { [xlo, xhi] in S | xlo > xMed }               // right of median
    M = { [xlo, xhi] in S | xlo <= xMed <= xhi }       // contains median
    ML = sort M in increasing order of xlo             // sort M
    MR = sort M in decreasing order of xhi
    t = new IntTreeNode(xMed, ML, MR)                  // this node
    t.left  = ConstructIntervalTree(L)                 // left subtree
    t.right = ConstructIntervalTree(R)                 // right subtree
    return t
```

# 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, xq)
*Input:* IntTreeNode t, Scalar xq
*Output:* prints the intersected intervals

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

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

[Mount]

# Complexity of line stabbing via interval tree

- 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)

DCGI

# Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree – IT*)
   i. Line stabbing (standard *IT* with *sorted lists* )
   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
   – *segment tree*

DCGI

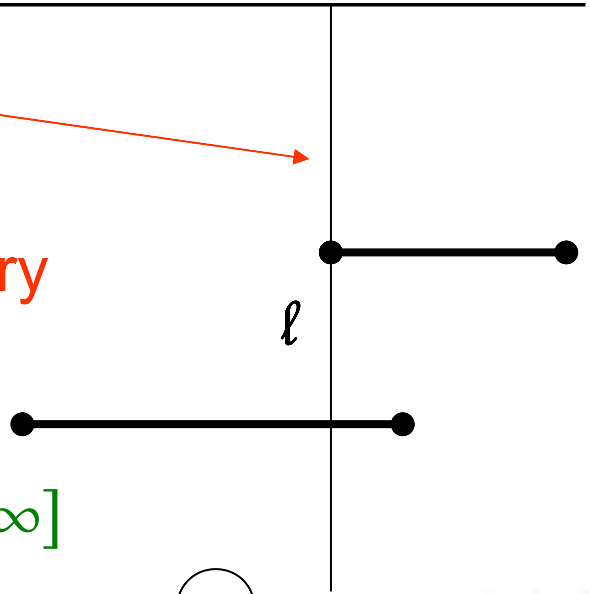
# Line segment stabbing (*IT* with *range trees*)

## Enhance 1D interval trees to 2D

- Change 1D test $q_x \in \langle x, x' \rangle$
  done by interval tree with sorted lists $M_L$ and $M_R$
  into 2D test $q_x \in (-\infty : q_x]$
- and change lines $q_x \times [-\infty : \infty]$ (no y-test)
  to segments $q_x \times [q_y : q'_y]$ (additional y-test)

# i. Segment intersected by vertical line - 2D

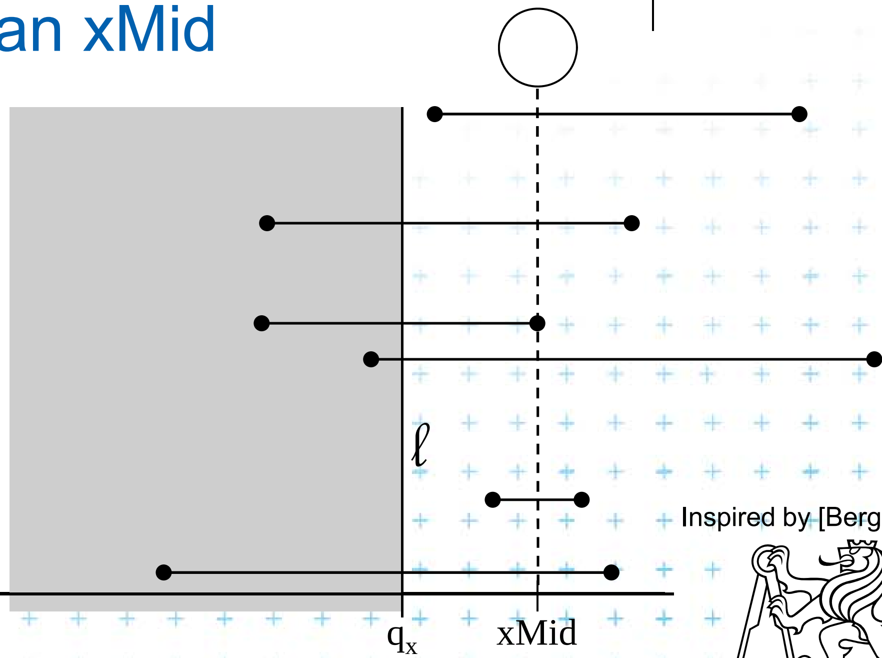
- Query line $\ell := q_x \times [-\infty : \infty]$




- Horizontal segment of *M* stabs the query line $\ell$ **iff** its left endpoint lies in halph-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 $\ell$ lies left from xMid
  - $M_R$: whose right point lies in $[q_x : +\infty)$ if $\ell$ lies right from xMid

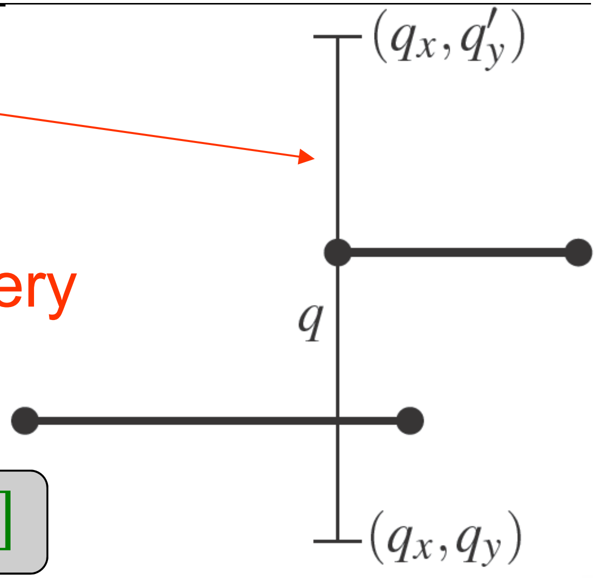



Inspired by [Berg]

DCGI

# ii. Segment intersected by vertical line segment

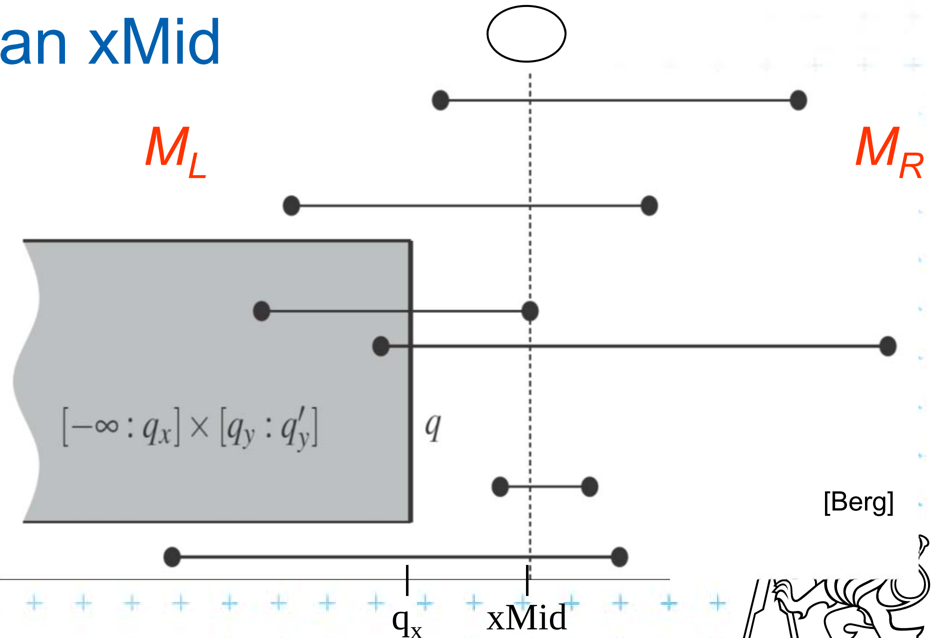
- Query segment $q := q_x \times [q_y : q'_y]$




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

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

- 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

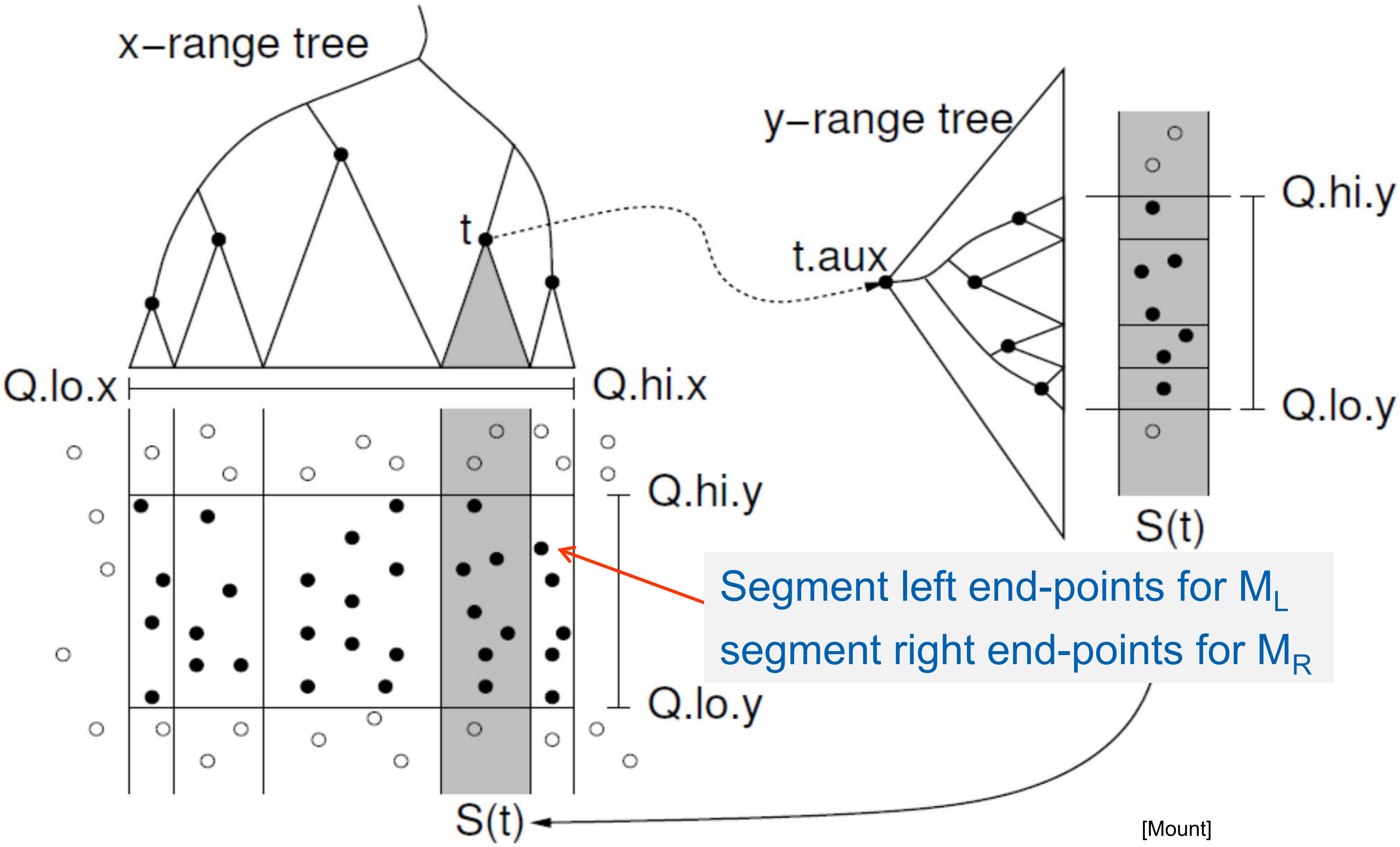



[Berg]

# Data structure for endpoints

- Storage of $M_L$ and $M_R$
  - 1D Sorted lists not enough for line segments
  - Use two 2D range trees
- 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)




Segment left end-points for $M_L$
segment right end-points for $M_R$

[Mount]

# Complexity of 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) => 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
  2D range tree search with Fractional Cascading
  - One node processed in O(log $n$ + k'), k'=reported inter.
  - $v$-visited nodes in O($v$ log $n$ + k), k=total reported inter.
  - $v$ = interval tree height = O(log $n$)
  - 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

DCGI

# Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   i. Line stabbing (standard *IT* with *sorted lists* )
   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
   – *segment tree*

# iii. Priority search trees [McCreight85]

- Priority search trees – in case c) on slide 9
  – Exploit the fact that query rectangle in range queries is unbounded (in x direction)
  – Can be used as secondary data structures for both left and right endpoints (ML and MR) of segments in nodes of interval tree – one for ML, one for MR
  – Improve the storage to O(*n*) for horizontal segment intersection with window edge (Range tree has O(*n* log *n*))
- For cases a) and b) - O(*n* log *n*) remains
  – we need range trees for windowing segment endpoints

# Rectangular range queries variants

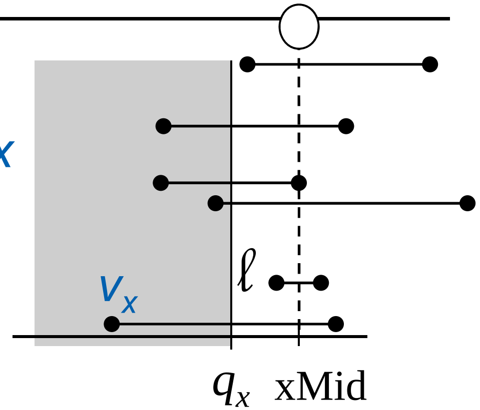
- Let $P = \{ p_1, p_2, \ldots, p_n \}$ is set of points in plane
- Goal: rectangular range queries of the form $(-\infty : q_x] \times [q_y ; q'_y]$
- In 1D: search for nodes $v$ with $v_x \in (-\infty : q_x]$
  - range tree — $O(\log n + k)$ time
  - ordered list — $O(1 + k)$ time (start in the leftmost, 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 $x \in (-\infty : q_x]$

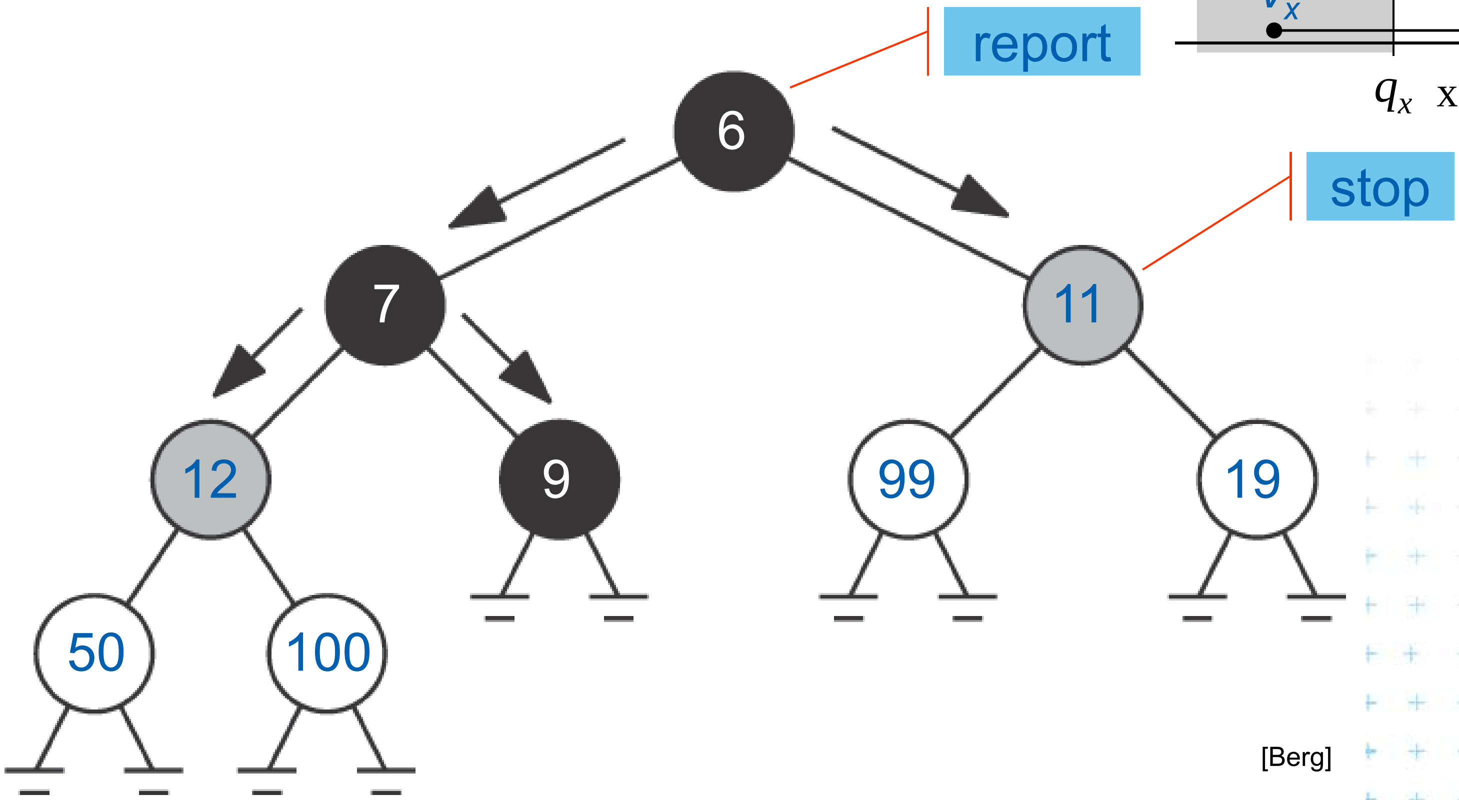
  \+ integrate information about y-coordinate

DCGI

# Heap for 1D unbounded range queries

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

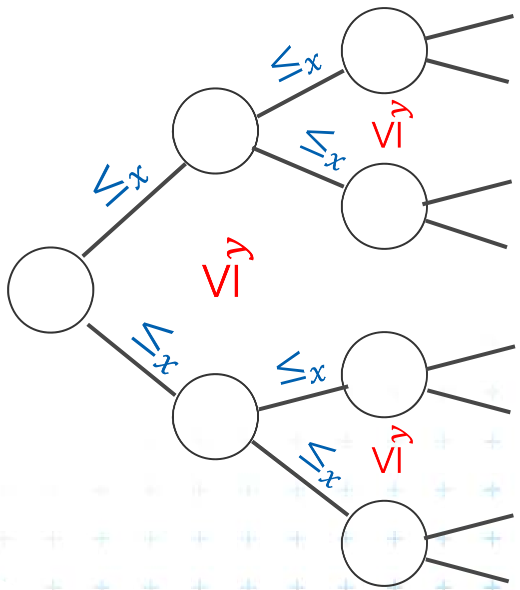



[Berg]

# Principle of priority search tree

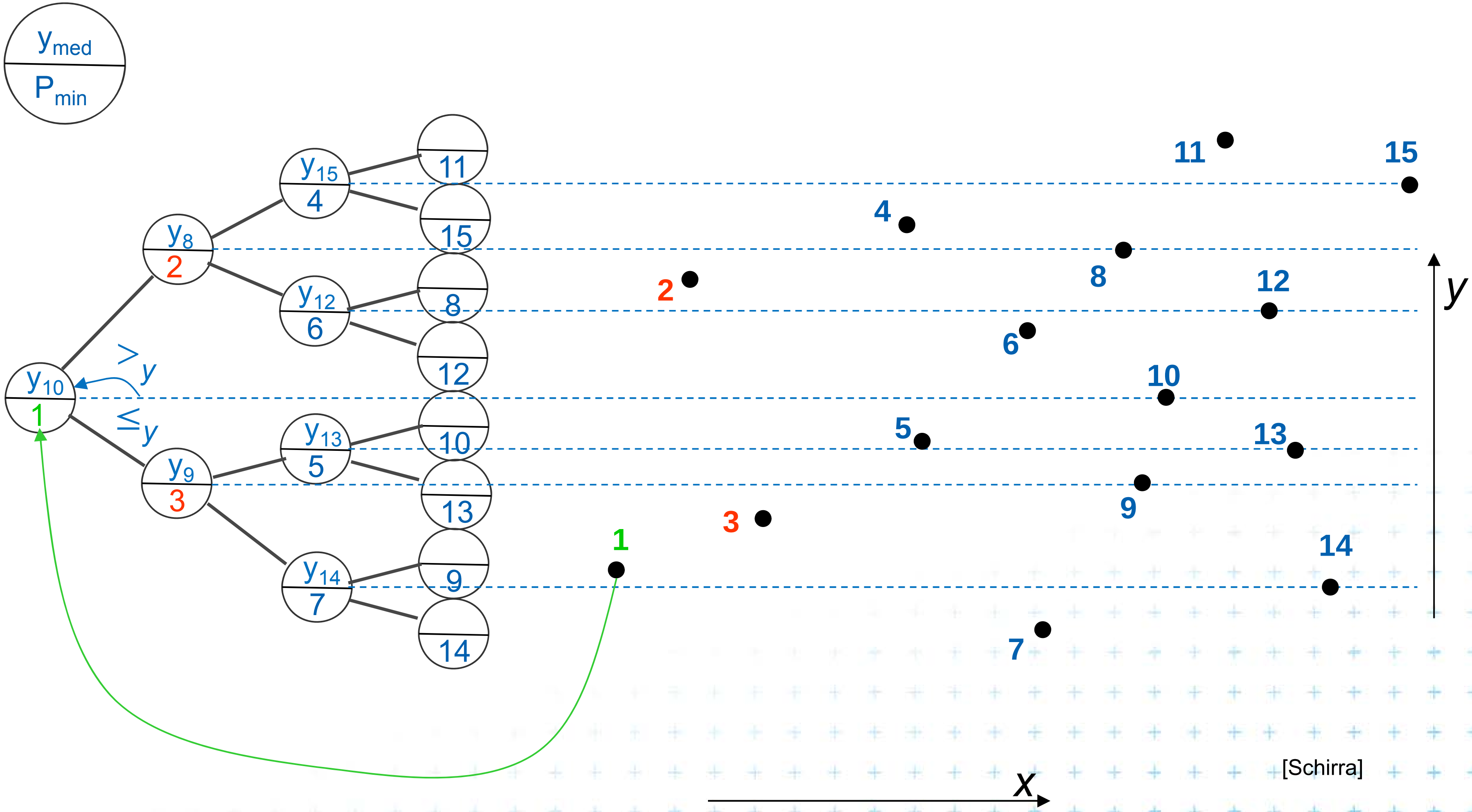
- Heap
  - relation between parent and its child nodes
  - no relation between the child nodes themselves
- Priority search tree
  - relate the child nodes according to y



# Priority search tree (PST)

- Heap in 2D can incorporate info about both *x,y*
  - BST on *y*-coordinate (horizontal slabs) ~ 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
  - $P_{below} := \{ p \in P \setminus \{p_{min}\} : p_y \leq y_{med}\}$
  - $P_{above} := \{ p \in P \setminus \{p_{min}\} : p_y > y_{med}\}$
- 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

**PrioritySearchTree( *P* )**

*Input:* set *P* of points in plane

*Output:* priority search tree *T*

1. if $P = \emptyset$ then PST is an empty leaf
2. else
3. $p_{min}$ = point with smallest x-coordinate in *P* // heap on x root
4. $y_{med}$ = *y*-coord. median of points $P \setminus \{p_{min}\}$ // BST on y root
5. Split points $P \setminus \{p_{min}\}$ into two subsets – according to $y_{med}$
6. $P_{below} := \{ p \in P \setminus \{p_{min}\} : p_y \leq y_{med} \}$
7. $P_{above} := \{ p \in P \setminus \{p_{min}\} : p_y > y_{med} \}$
8. *T* = newTreeNode()
9. $T.p = p_{min}$ // point [ *x*, *y* ]
10. $T.y = y_{mid}$ // skalar
11. *T.left* = PrioritySearchTree( $P_{below}$ )
12. *T.rigft* = PrioritySearchTree( $P_{above}$ )

13. O( *n* log *n* ) , but O( *n* ) if presorted on *y*-coordinate and bottom up

Notation in alg:
… p(v)
… y(v)
… lc(v)
… rc(v)

# Query Priority Search Tree

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

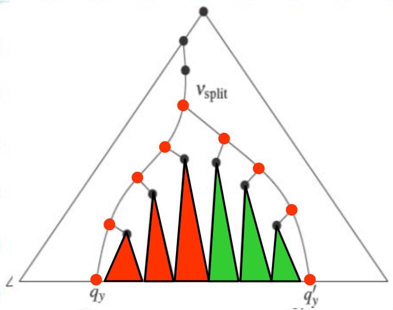
*Input:* A priority search tree and a range, unbounded to the left

*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( $rc(v)$, $q_x$ ) // 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( $lc(v)$, $q_x$ ) // rep. left subtree




[Berg]

DCGI

# Reporting of subtrees between the 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 in the subtree with $x$-coordinate at most $q_x$.

1. if $v$ is not a leaf and $x(\ p(v)\ ) \leq q_x$ // $x \in (-\infty : q_x]$ -- heap condition
2. Report $p(v)$.
3. ReportInSubtree( $lc(v)$, $q_x$ )
4. ReportInSubtree( $rc(v)$, $q_x$ )

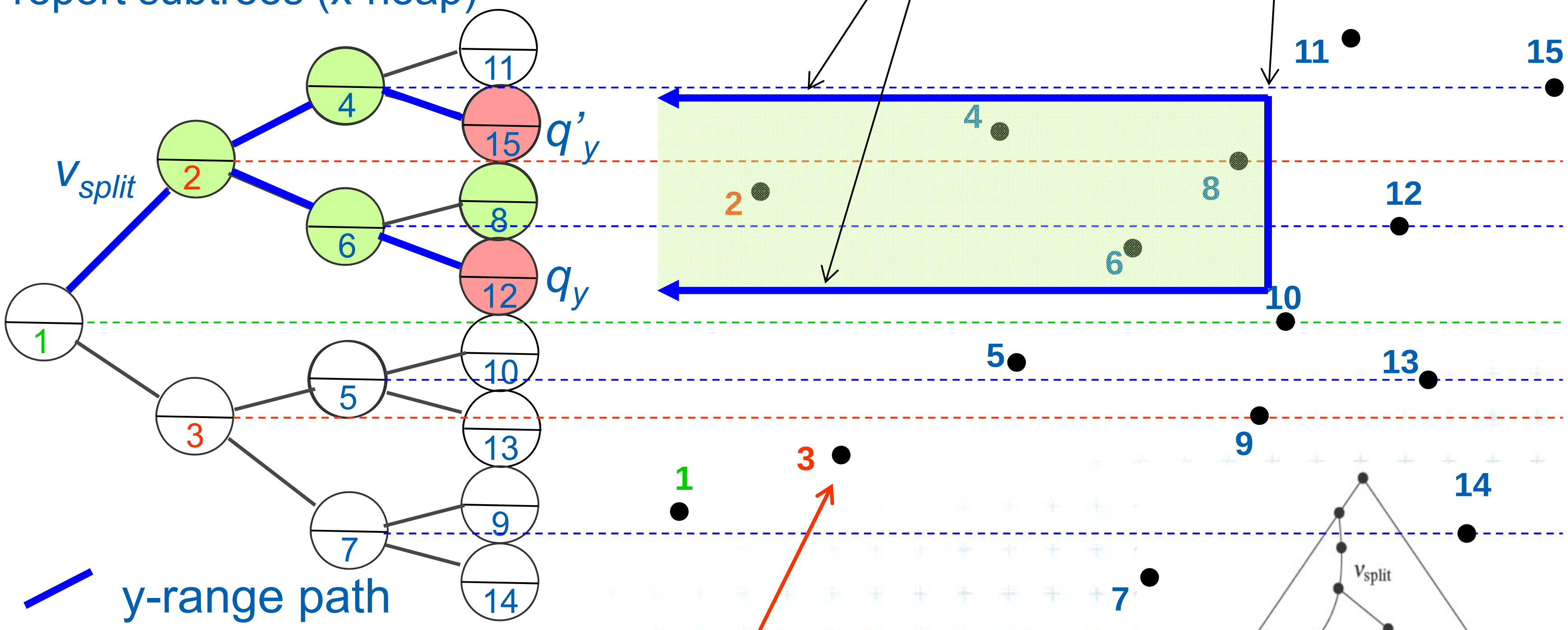
DCGI

# Priority search tree query

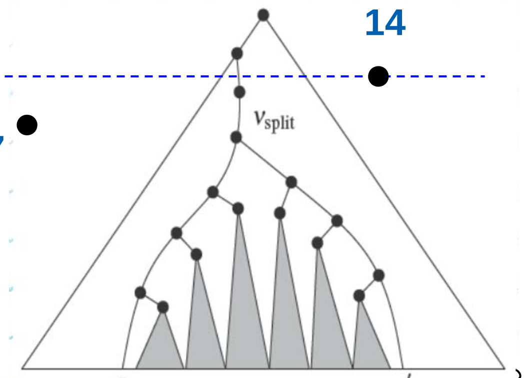
1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)
3. report subtrees (x-heap)

Given interval $y_{min}..y_{max}$

Given $x_{max}$




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

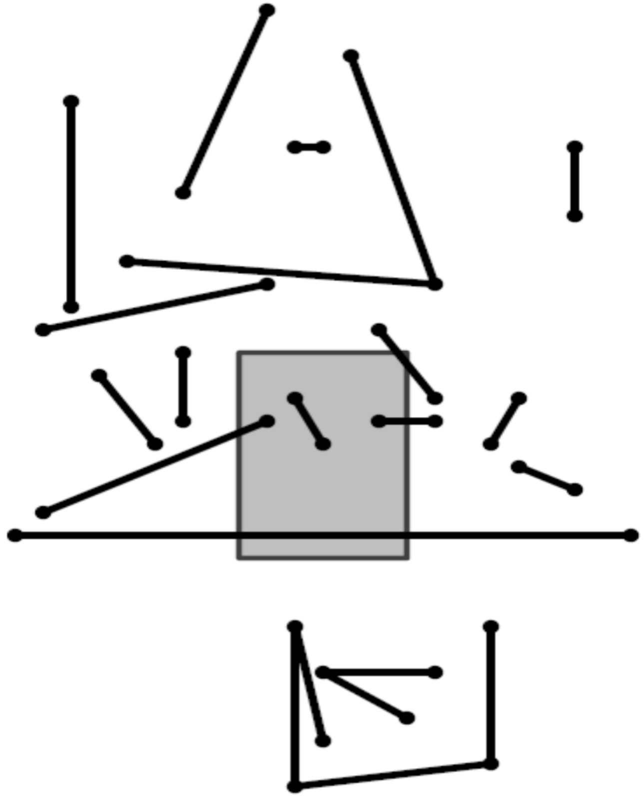
- Use Priority search tree as associated data structure for interval trees for storage of M (one for $M_L$, one for $M_R$)

DCGI

# Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   i. Line stabbing (standard *IT* with *sorted lists* )
   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
   – *segment tree*

# 2. Windowing of line segments in general position



[Vakken]

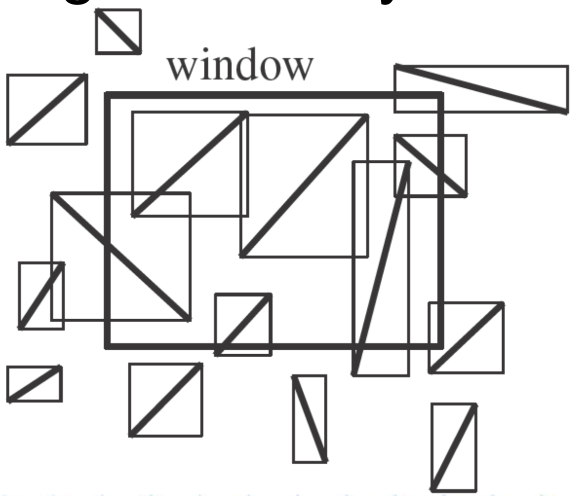
# Windowing of arbitrary oriented line segments

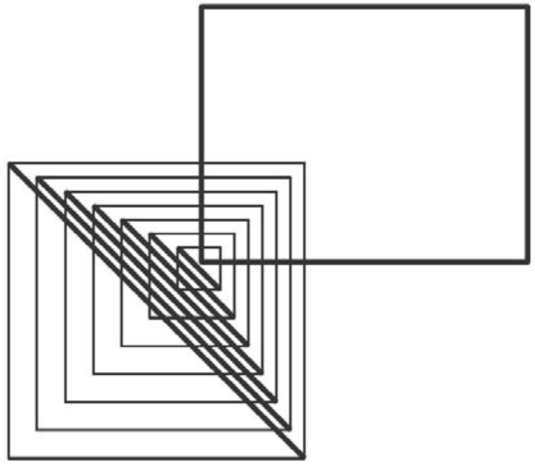
- Two cases of intersection

  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
  – But segments may not intersect the window –> query y



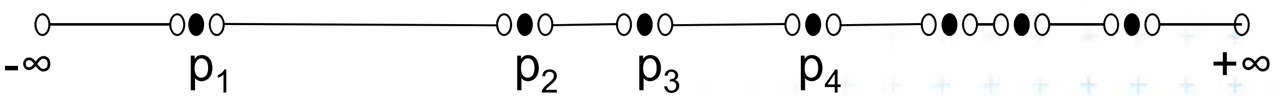
# Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   i. Line stabbing (*IT* with *sorted lists* )
   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
   – *segment tree*

# 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




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

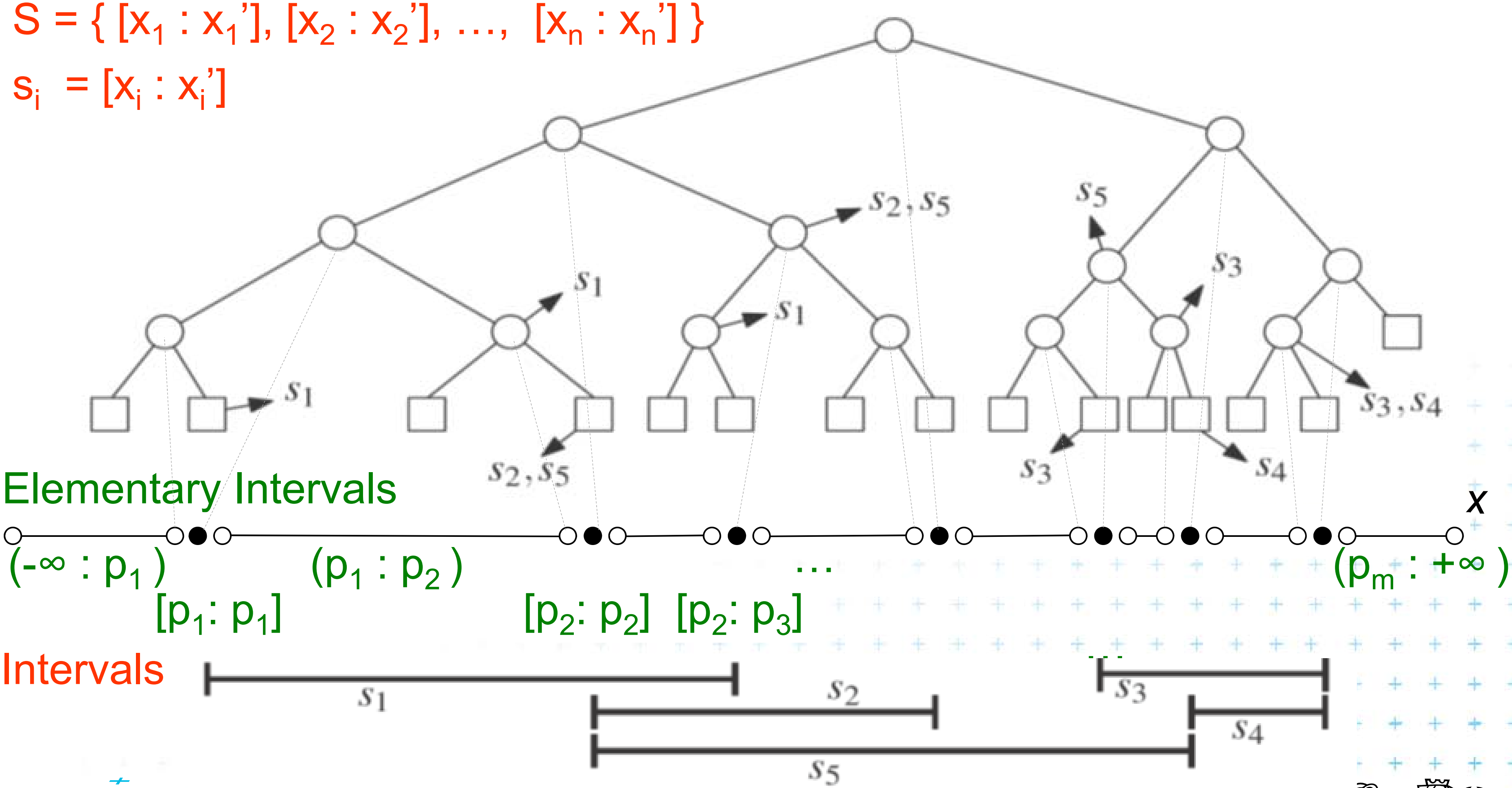
  - Stores intervals $s_i$ with the elementary intervals
  - Reports the intervals $s_i$ containing query point $q_x$.

# Segment tree example

Intervals

$S = \{ [x_1 : x_1'], [x_2 : x_2'], \ldots, [x_n : x_n'] \}$

$s_i = [x_i : x_i']$




[Berg]



DCGI

# Segment tree definition

Segment tree

- Skeleton is a balanced binary tree *T*
- Leaves ~ elementary intervals Int(v)
- Internal nodes *v*
  ~ union of elementary intervals of its children
  - Store: 1. interval Int(v) = union of elementary intervals of its children
    2. canonical set $S(v)$ of intervals $[x : x'] \in S$ (segments $s_i$)
  - Holds $\text{Int}(v) \subseteq [x : x']$ and $\text{Int}(\text{parent}(v)) \not\subseteq [x : x']$
    (node interval is not larger than the segment)
  - Intervals $[x : x']$ are stored as high as possible, such that Int($v$) is completely contained in the segment

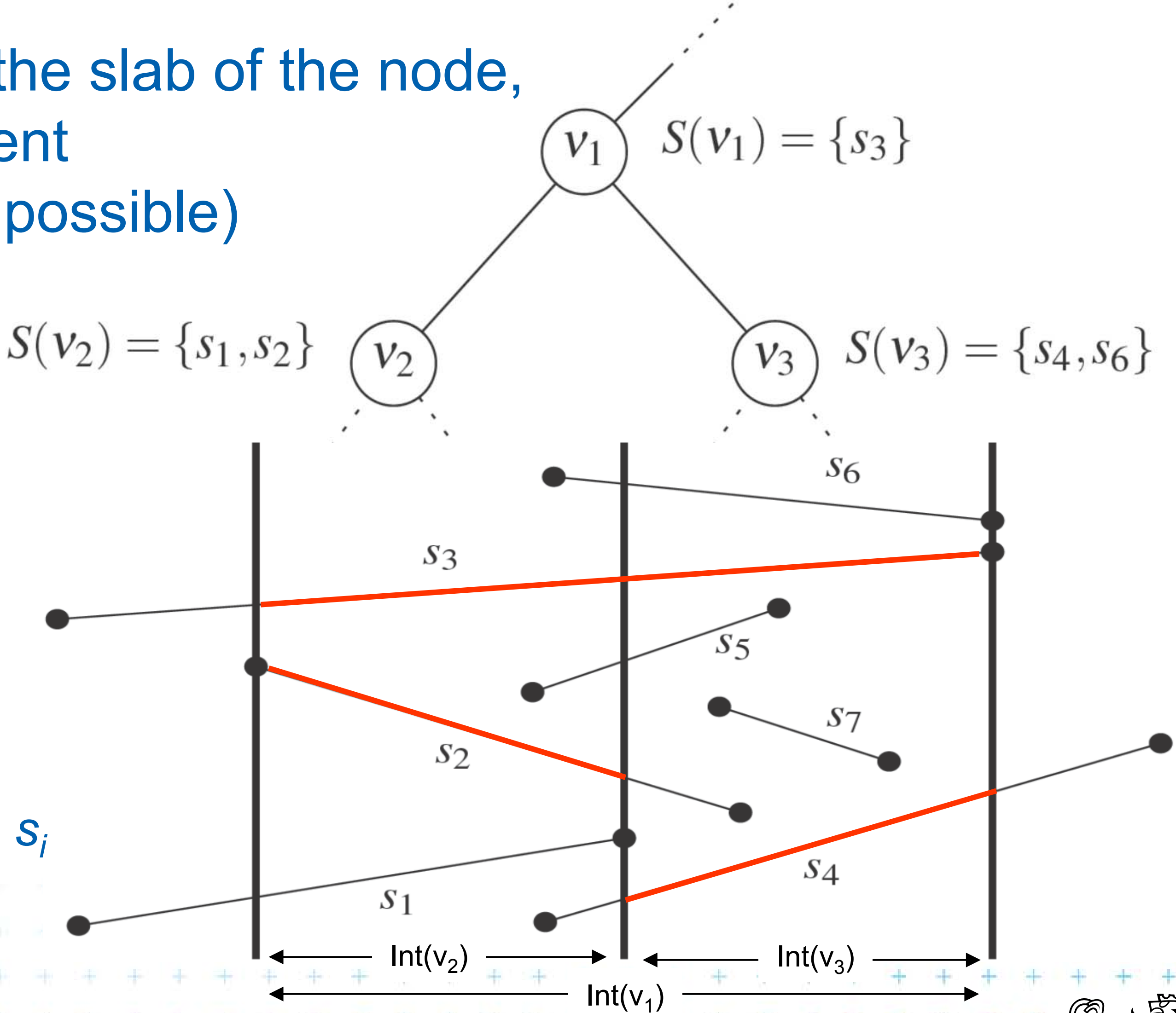
# Segments span the slab

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

$\text{Int}(v_j) \subseteq s_i$

and

$\text{Int}(\text{parent}(v_j)] \not\subseteq s_i$




[Berg]

# Query segment tree – stabbing query

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

Report all the intervals $s_i$ in $S(v)$. // current node
**if** $v$ is not a leaf
    if $q_x \in$ Int( $lc(v)$ ) // go left
        QuerySegmentTree( $lc(v)$, $q_x$ )
    else // or go right
        QuerySegmentTree( $rc(v)$, $q_x$ )

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 *S* - segments
*Output:* segment tree

1. Sort endpoints of segments in *S* -> get elemetary 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):
4. v = root( *T* )
5. for all segments $s_i = [x : x'] \in S$
6. InsertSegmentTree( *v*, $[x : x']$ )

# Segment tree construction – interval insertion

InsertSegmentTree( $v$, [$x$ : $x'$] )
*Input:* The root of (a subtree of) a segment tree and an interval.
*Output:* The interval will be stored in the subtree.

```
if Int(v) ⊆ [ x : x' ]                        // Int(v) contains s_i = [ x : x' ]
    store [ x : x' ] at v
else if Int( lc(v) ) ∩ [ x : x' ] ≠ ∅
        InsertSegmentTree( lc(v), [x : x' ] )
      if Int( rc(v) ) ∩ [ x : x' ] ≠ ∅
        InsertSegmentTree(rc(v), [x : x' ] )
```

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

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

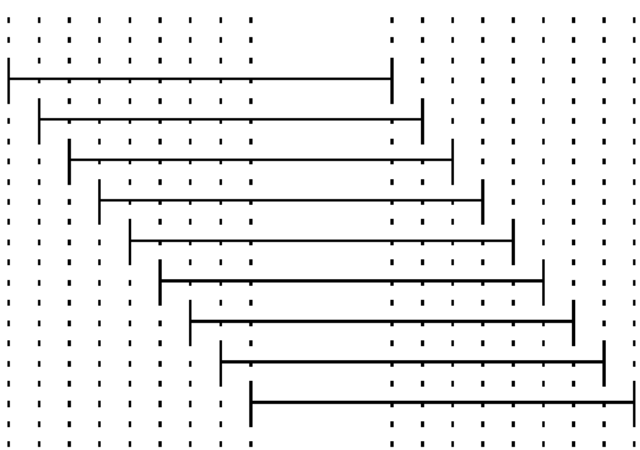
Construction total $O(n \log n)$

Storage $O(n \log n)$

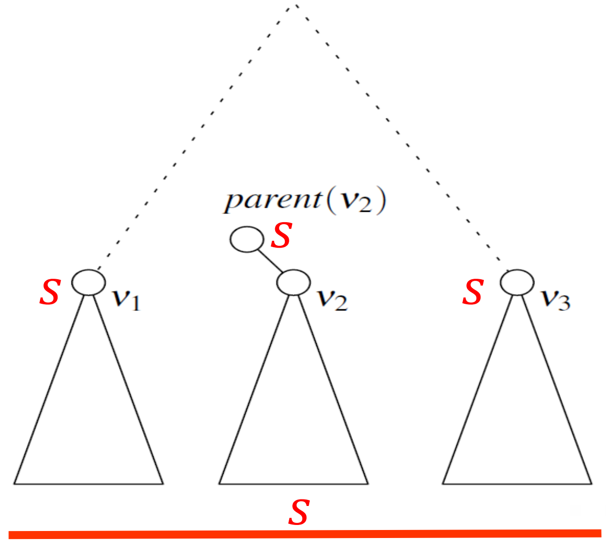
Tree height $O(\log n)$, name stored max 2x in one level

Storage total $O(n \log n)$ – see next slide

# Space complexity - notes



[Berg]




[Berg]

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

But

Store segments as high, as possible

Segment max 2 times in one level ⇐

- $s$ 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

DCGI

# Segment tree versus Interval tree

- Segment tree
  - O(*n* log *n* ) storage x O(*n*) of Interval tree
  - But returns exactly the intersected segments $s_i$, interval tree must search the lists ML and/or MR
- 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

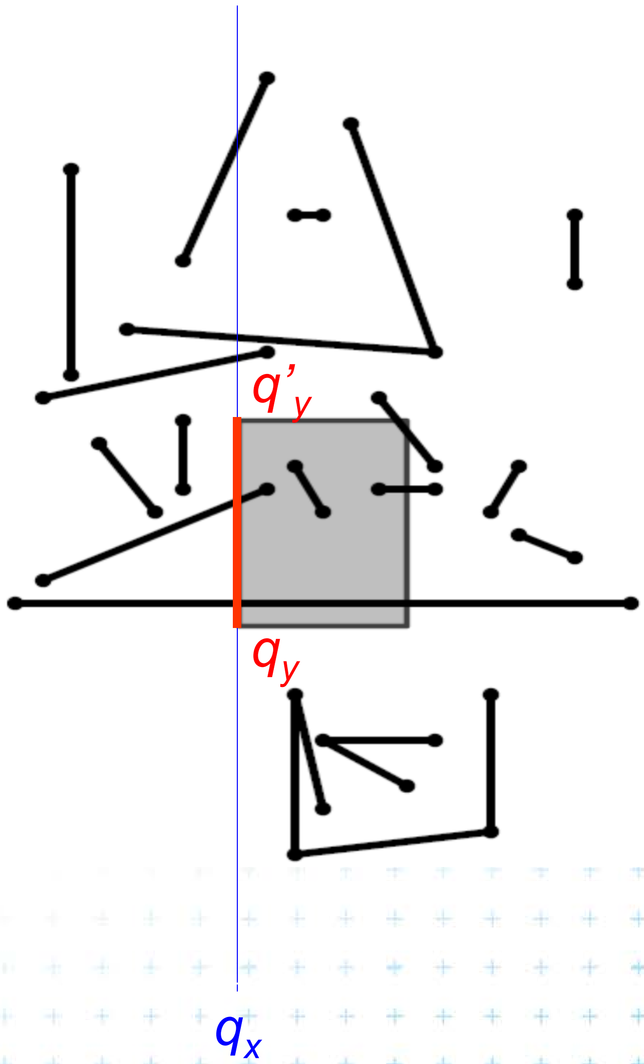
     (Interval and priority search trees do not exist in ^dims)

DCGI

# Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   i. Line stabbing (standard *IT* with *sorted lists* )
   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
   – *segment tree*
   – the algorithm

# 2. Windowing of line segments in general position

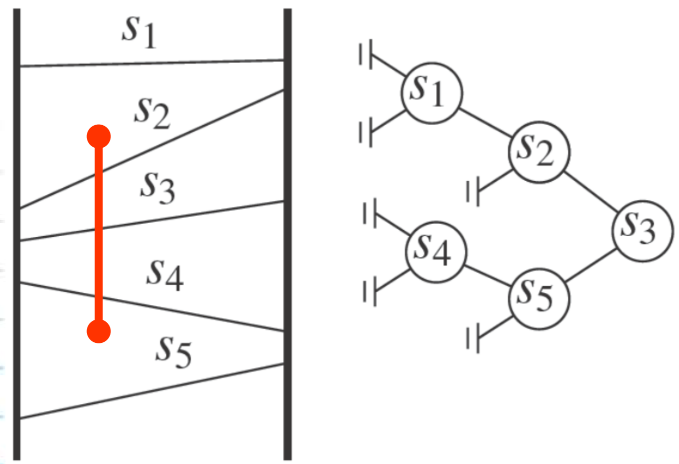



[Vakken]

# 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 $q := q_x \times [q_y : q'_y]$
- Segment tree $T$ on $x$ intervals of segments in $S$
  - node $v$ of $T$ corresponds to vertical slab $\text{Int}(v) \times (-\infty : \infty)$
  - 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]

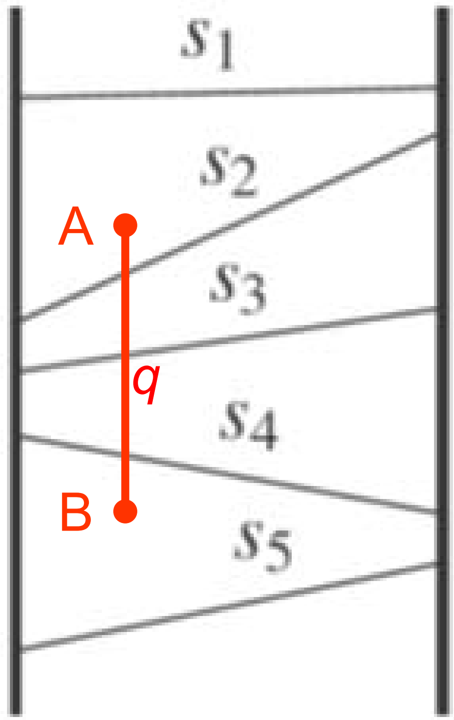
# Segments between vertical segment endpoints

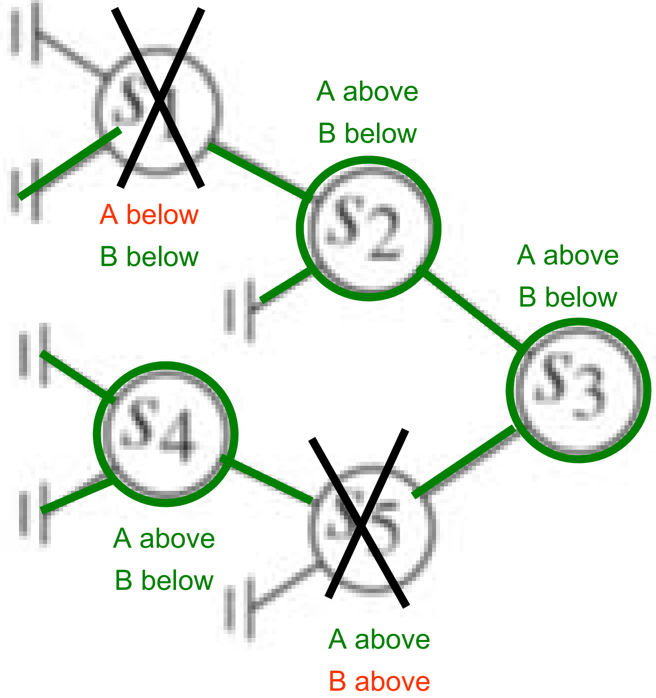
- Segments (in the slab) do not mutually intersect
  => segments can be vertically ordered and stored in BST
  - Each node *v* of the x segment tree has an associated y BST
  - BST *T*(*v*) of node *v* stores the canonical subset *S*(*v*) according to the vertical order
  - Intersected segments can be found by searching *T*(*v*) in O( $k_v$ + log *n*), $k_v$ is the number of intersected segments

DCGI

# 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]

# 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 $O( k + \log^2 n)$
  - Report all segments that contain a query point
  - $k$ is number of reported segments

DCGI

# Windowing of line segments in 2D – conclusions

Construction: all variants O(n logn)

| 1. Axis parallel | Search | Memory |
|---|---|---|
| i. Line (*sorted lists* ) | $O(k + \log n)$ | $O(n)$ |
| 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

| | | |
|---|---|---|
| – *segment tree* | $O(k + \log^2 n)$ | $O(n \log n)$ |

DCGI

# 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] David Mount, - CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland, Lectures 7,22, 13,14, and 30. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml

[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