DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

GEOMETRIC SEARCHING PART 2: RANGE SEARCH

PETR FELKEL

FEL CTU PRAGUE

felkel@fel.cvut.cz

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

Based on [Berg] and [Mount]

Version from 22.10.2015

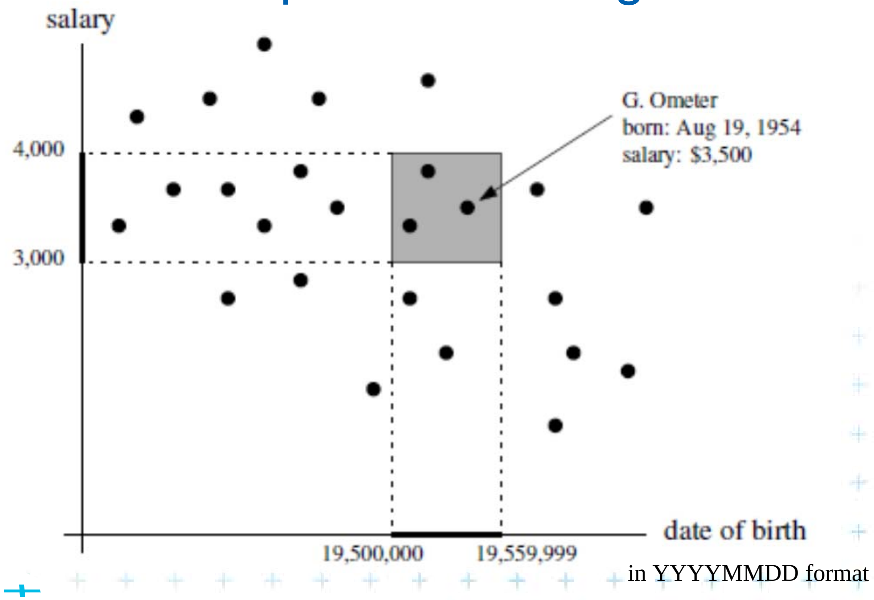
Range search

- Orthogonal range searching
- Canonical subsets
- 1D range tree
- Kd-tree
- 2D-nD Range tree
  - With fractional cascading (Layered tree)

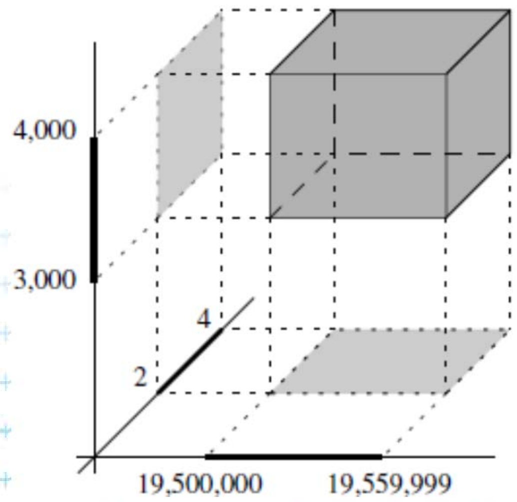
Orthogonal range searching

– Given a set of points P, find the points in the region Q
  - Search space: a set of points P (somehow represented)
  - Query: intervals Q (axis parallel rectangle)
  - Answer: points contained in Q
– Example: Databases (records->points)
  - Find the people with given range of salary, date of birth, kids, …

2D: axis parallel rectangle



3D: axis parallel box



[Berg]

Orthogonal range searching

- Query region = axis parallel rectangle
  - nDimensional search can be decomposed into set of 1D searches (separable)

Other range searching variants

- Search space S: set of
  - line segments,
  - rectangles, …
- Query region Q: any other region
  - disc,
  - polygon,
  - halfspace, …
- Answer: subset of S laying in Q

- We concentrate on points in orthogonal ranges

How to represent the search space?

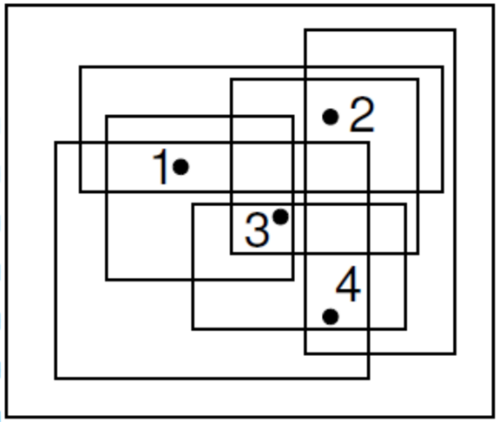
Basic idea:

- Not all possible combination can be in the output (not the whole power set)
- => Represent only the “selectable” things (a well selected subset –> one of the canonical subsets)

Subsets selectable by given range class

- The number of subsets that can be selected by simple ranges Q is limited
- It is usually much smaller than the power set of P
  - Power set of P where P = {1,2,3,4} (potenční množina) is {{ }, {1},{2},{3},{4}, {1,2},{1,3},{1,4}, {2,3},…,{2,3,4}, {1,2,3,4} } … $O(2^n)$
    i.e. set of all possible subsets
  - Simple rectangular queries are limited
    - Defined by max 4 points along 4 sides => $O(n^4)$ of $O(2^n)$ power set
    - Moreover – not all sets can be formed by □ query Q

      e.g. sets {1,4} and {1,2,4} cannot be formed



[Mount]

Canonical subsets S_i

- Search space S=(P,Q) represented as a collection of canonical subsets $\{S_1, S_2, \ldots, S_k\}$, each S_i Å S,
  - S_i may overlap each other (elements can be multiple times there)
  - Any set can be represented as disjoint union disjunktní sjednocení of canonical subsets S_i each element knows from which subset it came
  - Elements of disjoint union are ordered pairs (*x*, *i*) (every element *x* with index *i* of the subset S_i)
- S_i may be selected in many ways
  - from *n* singletons $\{p_i\}$ … O(n)
  - to power set of P … $O(2^n)$
  - Good DS balances between total number of canonical subsets and number of CS needed to answer the query

1D range queries (interval queries)

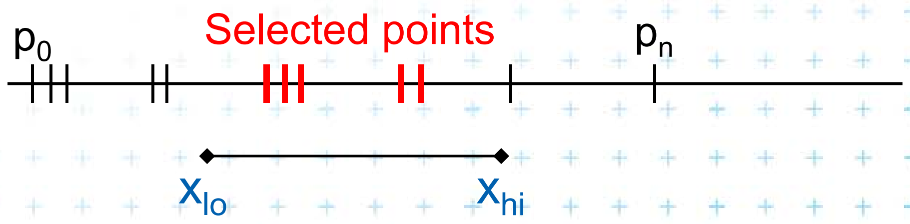
- Query: Search the interval $[x_{lo}, x_{hi}]$
- Search space: Points $P = \{p_1, p_2, \ldots, p_n\}$ on the line

a) Binary search in an array

- Simple, but
- not generalize to any higher dimensions

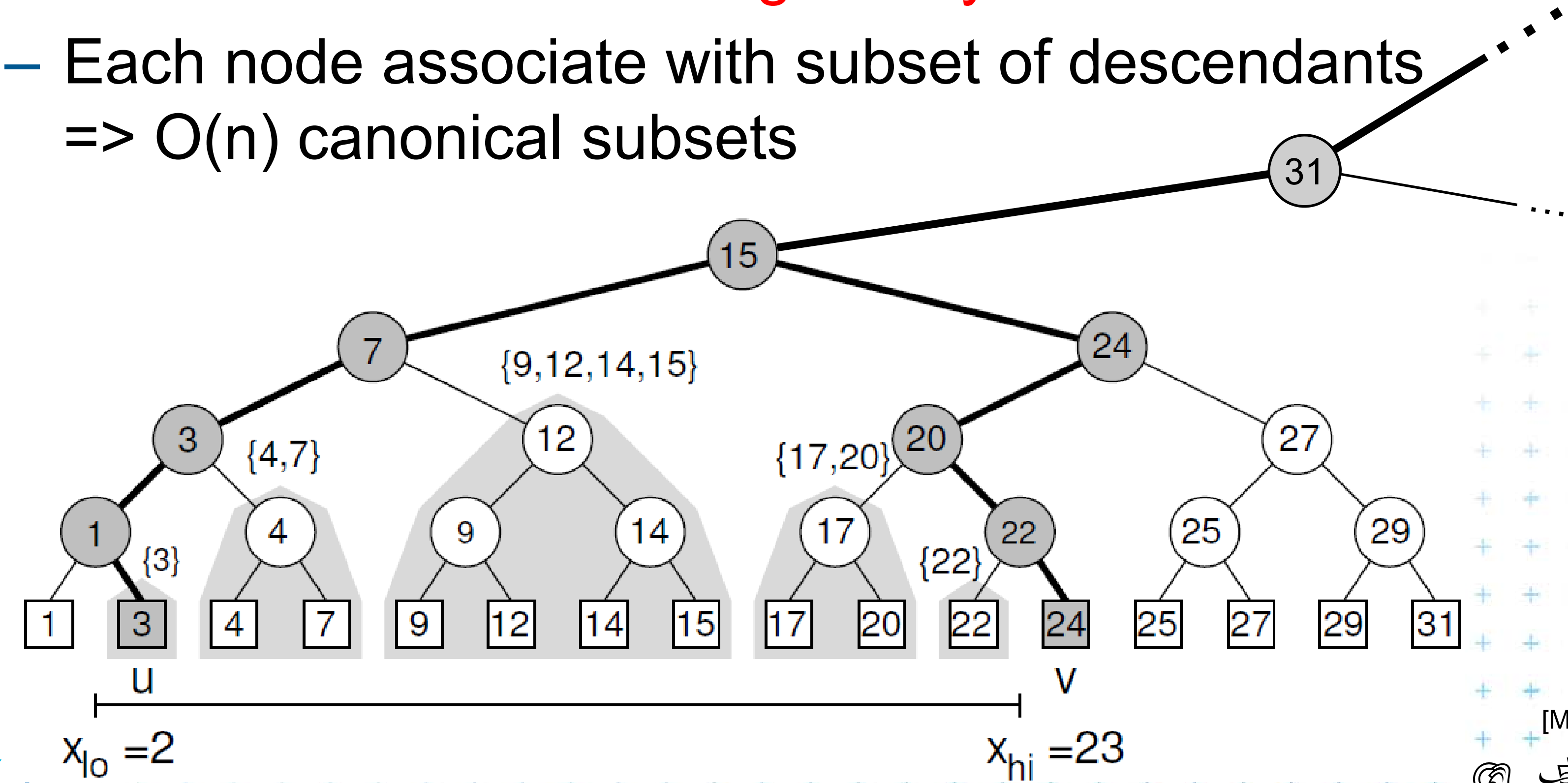
b) Balanced binary search tree

- 1D range tree
- maintains canonical subsets
- generalize to higher dimensions



1D range tree definition

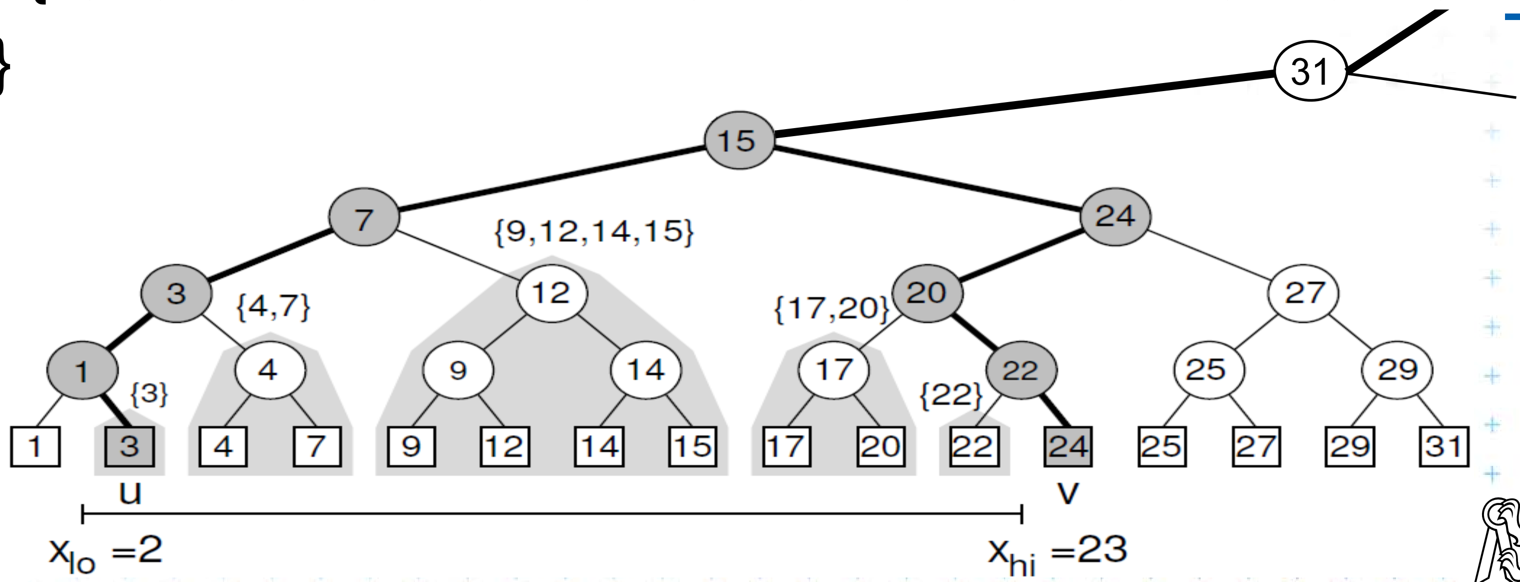
- Balanced binary search tree
  - leaves – sorted points
  - inner node label – the largest key in its left child
  - Each node associate with subset of descendants => O(n) canonical subsets



Canonical subsets and <2,23> search

- Canonical subsets for this subtree are

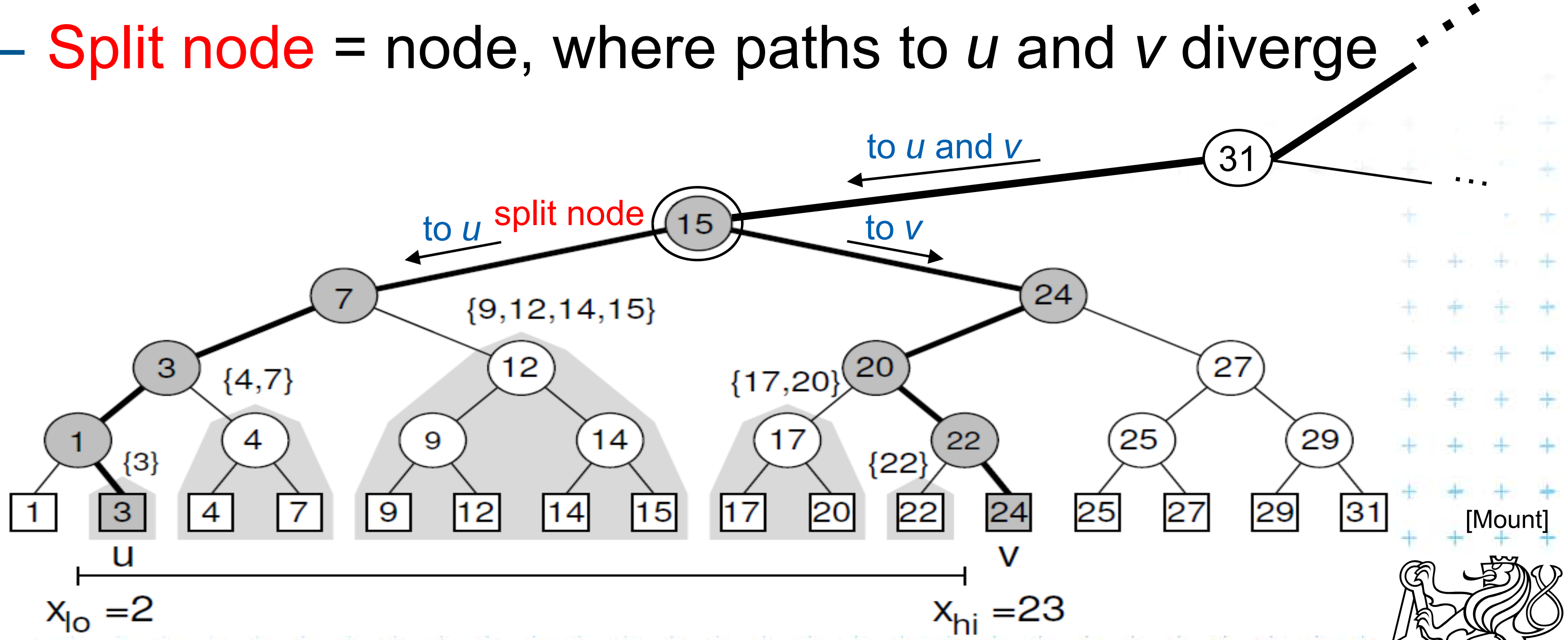
	#
{ {1}, {3}, …, {31},	16
{1, 3}, {4, 7}, …, {29, 31}	8
{1, 3, 4, 7}, {9, 12, 14, 15}, …, {25, 27, 29, 31}	4
{1, 3, 4, 7, 9, 12, 14, 15}, {17, 20, 22, 24, 25, 27, 29, 31}	2
{1, 3, 4, 7, 9, 12, 14, 15, 17, 20, 22, 24, 25, 27, 29, 31}	1
}	O(n)



[Mount]

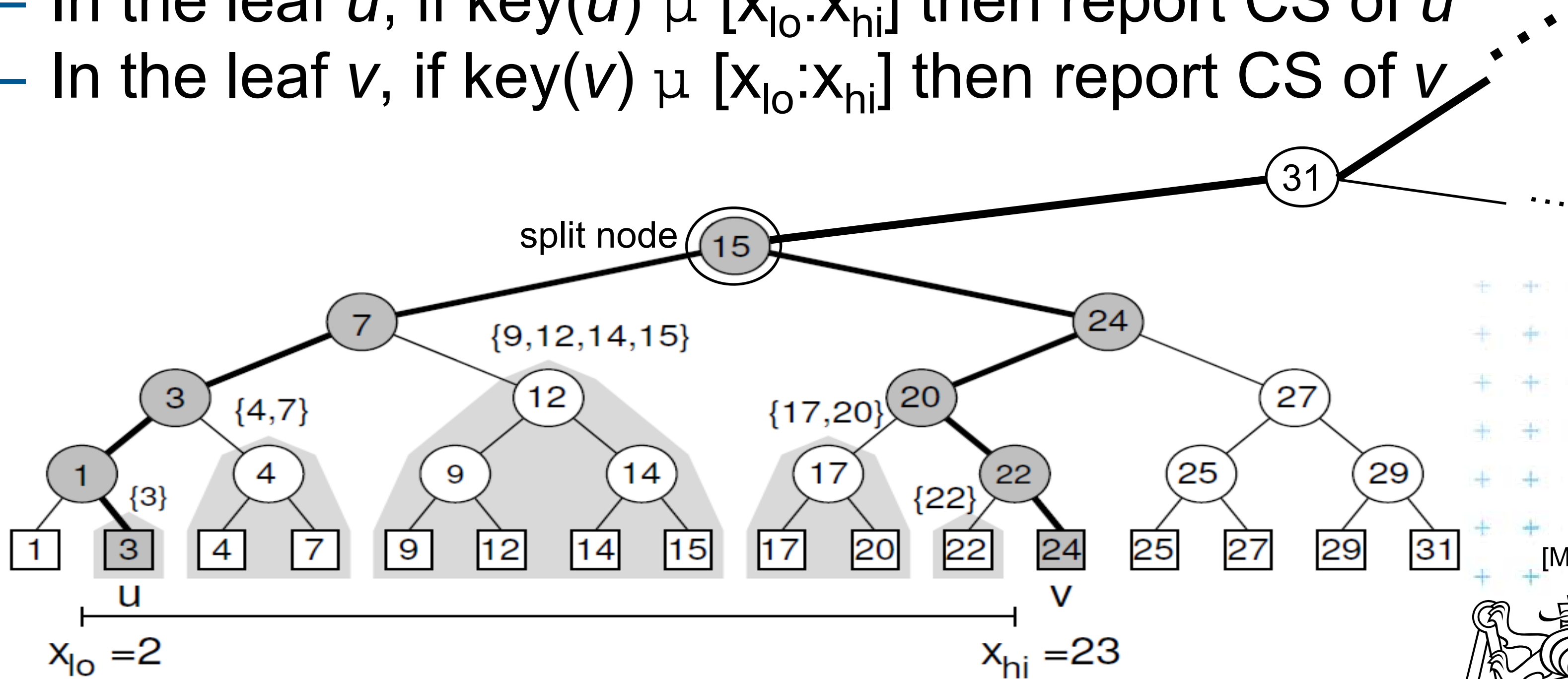
1D range tree search interval <2,23>

- Canonical subsets for any range found in O(log *n*)
  - Search x_{lo}: Find leftmost leaf *u* with key(*u*) – x_{lo} 2 -> 3
  - Search x_{hi}: Find leftmost leaf *v* with key(*v*) – x_{hi} 23 -> 24
  - Points between *u* and *v* lie within the range => report canon. subsets of maximal subtrees between *u* and *v*
  - Split node = node, where paths to *u* and *v* diverge



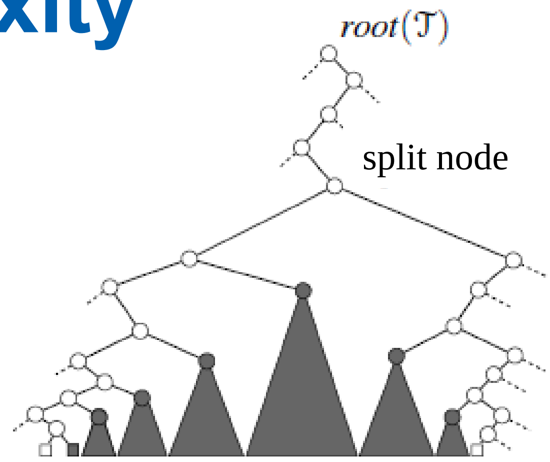
1D range tree search

- Reporting the subtrees (below the split node)
  - On the path to *u* whenever the *path goes left*, report the canonical subset (CS) associated to right child
  - On the path to *v* whenever the *path goes right*, report the canonical subset associated to left child
  - In the leaf *u*, if key(*u*) µ [x_{lo}:x_{hi}] then report CS of *u*
  - In the leaf *v*, if key(*v*) µ [x_{lo}:x_{hi}] then report CS of *v*



[Mount]

1D range tree search complexity



[Berg]

- Path lengths O(log n)

  => O(log n) canonical subsets (subtrees)
- Range counting queries
  - Return just the number of points in given range
  - Sum the total numbers of leaves stored in maximal subtree roots … O(log *n*) time
- Range reporting queries
  - Return all *k* points in given range
  - Traverse the canonical subtrees … O(log *n* + *k*) time
- O(*n*) storage, O(*n* log *n*) preprocessing (sort P)

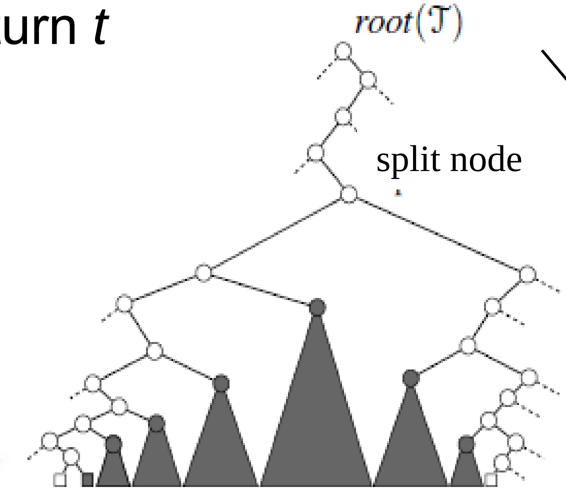
Find split node

FindSplitNode(*T*, [x:x'])

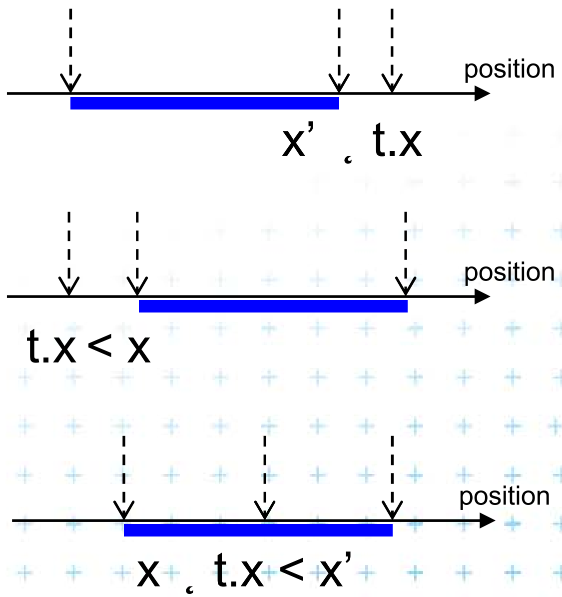
Input: Tree *T* and Query range [x:x'], x ¸ x'

Output: The node, where the paths to x and x' split or the leaf, where both paths end

1. *t = root(T)*
2. while(*t* is not a leaf **and (**x' ¸ t.x **or** t.x < x**)**) // out of the range [x:x']
3. if(*x'* ¸ t.x) *t = t.left*
4. else *t = t.right*
5. return *t*



[Berg]



1D range search

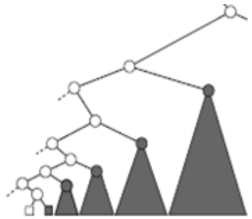
(2D on slide 30)

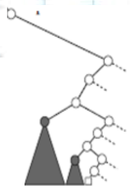
1dRangeQuery(*t*, [x:x'])

Input: 1d range tree *t* and Query range $[x : x']$

Output: All points in *t* lying in the range

```
tsplit = FindSplitNode( t, x, x' )          // find interval point t µ [x:x']
if( tsplit is leaf )   // e.g. Searching [16:17] or [16:16.5] both stops in the leaf 17 in the previous example
    check if the point in tsplit must be reported   // t_x ∈ [x : x']
else // follow the path to x, reporting points in subtrees right of the path
    t = tsplit.left
    while( t is not a leaf )
        if( x ¸ t.x)
            ReportSubtree( t.right )    // any kind of tree traversal
            t = t.left
        else t = t.right
    check if the point in leaf t must be reported
    // Symmetrically follow the path to x' reporting points left of the path
    t = tsplit.right …
```





Multidimensional range searching

- Equal principle – find the largest subtrees contained within the range
- Separate one *n*-dimensional search into *n* 1-dimensional searches
- Different tree organization
  - Kd tree
  - Orthogonal (Multilevel) range search tree
    e.g. nd range tree

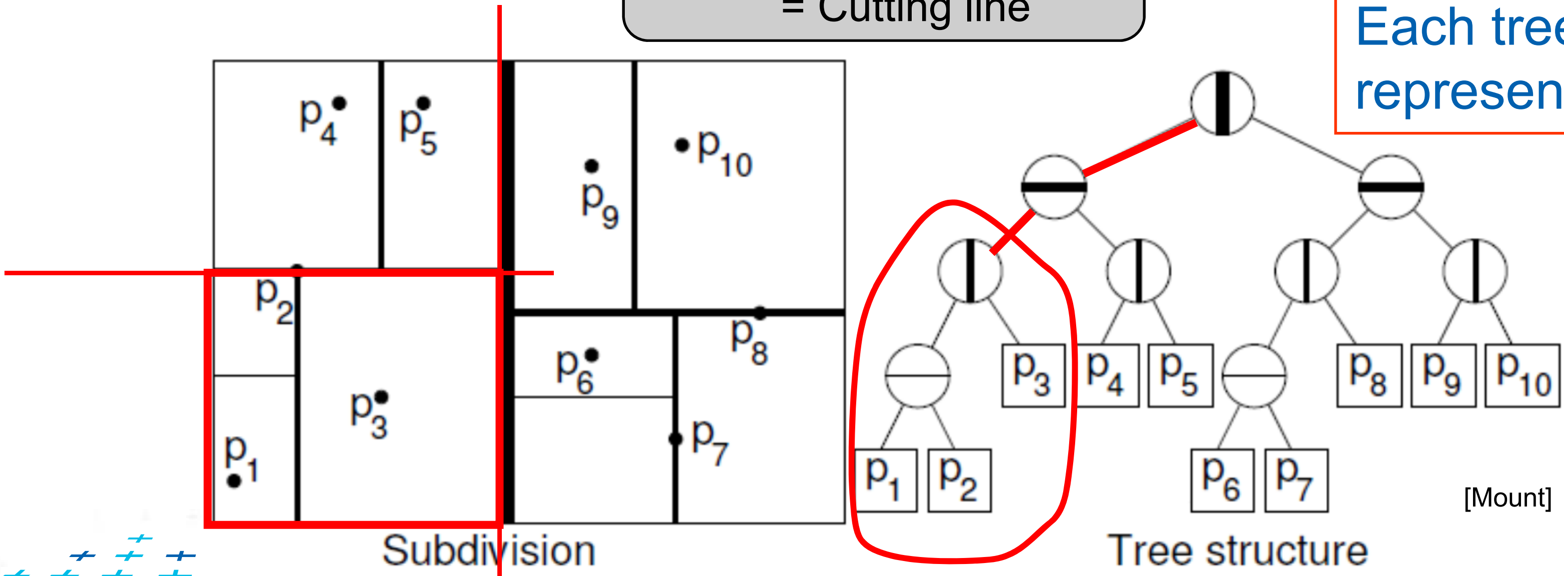
Kd-tree

- Easy to implement
- Good for different searching problems (counting queries, nearest neighbor,…)
- Designed by Jon Bentley as k-dimensional tree (2-dimensional kd-tree was a 2-d tree, …)
- Not the asymptotically best for orthogonal range search (=> range tree is better)
- Types of queries
  - Reporting – points in range
  - Counting – number of points in range

Kd-tree principle

- Subdivide space according to different dimension (*x*-coord, then *y*-coord, …)
- This subdivides space into rectangular cells
  => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))
  = Cutting line

Each tree node represents a region



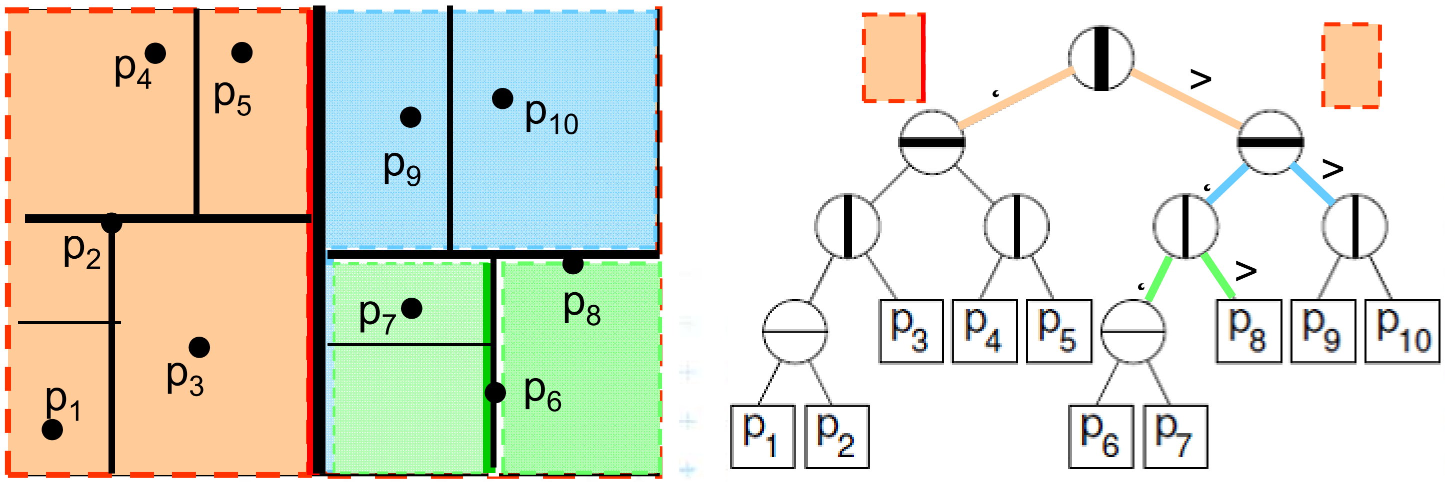
Where is a mistake in the figure?

Kd-tree principle

- Which dimension to cut? (cutDim)
  - Cycle through dimensions (round robin)
    - Save storage – cutDim is implicit ~ depth in the tree
    - May produce elongated cells (if uneven data distribution)
  - Greatest spread (the largest difference of coordinates)
    - Adaptive
    - Called “Optimal kd-tree”
- Where to cut? (cutVal)
  - Median, or midpoint between upper and lower median -> `P`(n)
  - Presort coords of points in each dimension (x-, y-,...) for `P`(1) median – resp. `P`(d) for all d dimensions

Kd-tree principle

- What about points on the cell boundary?
  - Boundary belongs to the left child
  - Left: $p_{cutDim} \le cutVal$
  - Right: $p_{cutDim} > cutVal$



Subdivision

Tree structure [Mount]

Kd-tree construction in 2-dimensions

BuildKdTree(*P, depth*)

Input: A set of points *P* and current *depth.*

Output: The root of a kD tree storing P.

```
If (P contains only one point)   [or small set of (10 to 20) points]
    then return a leaf storing this point
    else if (depth is even)
        then split P with a vertical line l through median x into two subsets
             P1 and P2 (left and right from median)
        else split P with a horiz. line l through median y into two subsets
             P1 and P2 (below and above the median)
    t_left  = BuildKdTree(P1, depth+1)
    t_right = BuildKdTree(P2, depth+1)
    create node t storing l, t_left and t_right children     // l = cutDim, cutVal
    return t
```

Split according to (*depth%max_dim)* dimension

If median found in O(1) and array split in O(n)

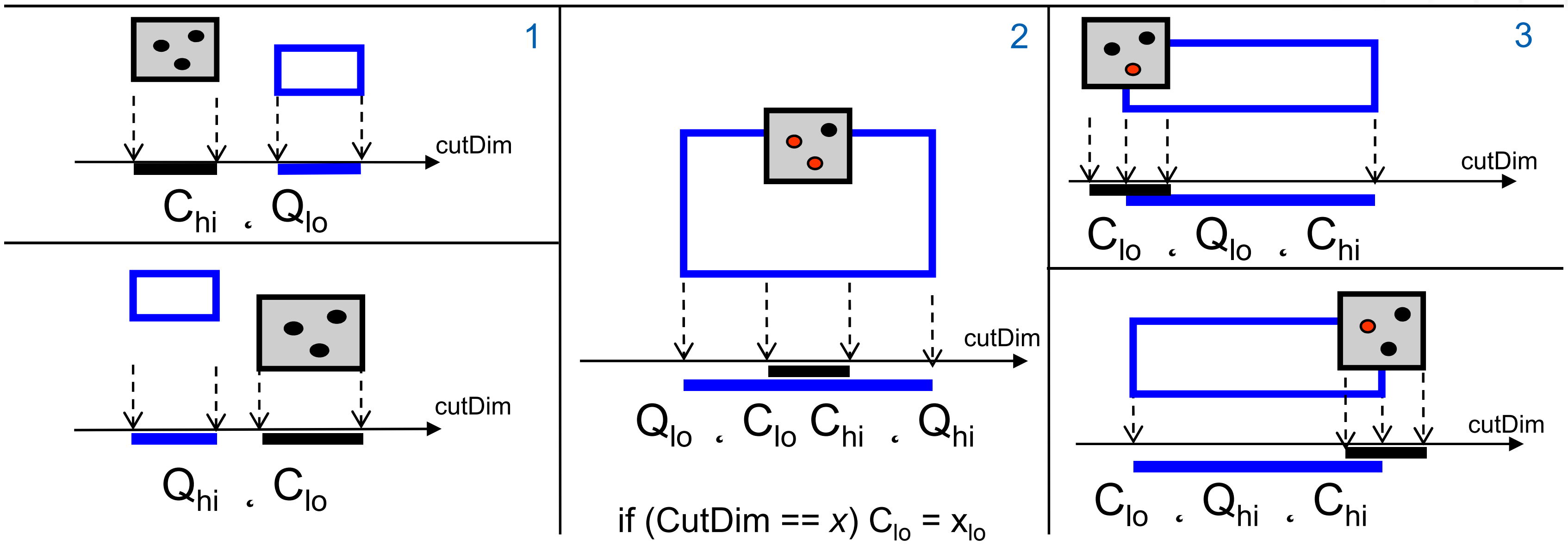
$T(n) = 2\,T(n/2) + n \Rightarrow O(n \log n)$ construction

Kd-tree test variants

Test interval-interval

a) Compare rectang. array Q with rectangular cells C

- Rectangle C:[x_{lo}, x_{hi}, y_{lo}, y_{hi}] – computed on the fly
- Test of kD node cell C against query Q (in one cutDim)
  1. if cell is disjoint with Q … $C \cap Q = \emptyset$ … stop
  2. If cell C completely inside Q … $C \subseteq Q$ … stop and report cell points
  3. else cell C overlaps Q … recurse on both children
- Recursion stops on the largest subtree (in/out)



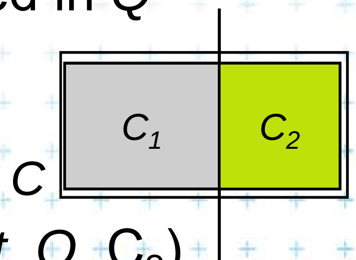
Kd-tree rangeCount (with rectangular cells)

int rangeCount(*t*, *Q*, *C*)

Input: The root *t* of kD tree, query range *Q and t's* cell C.

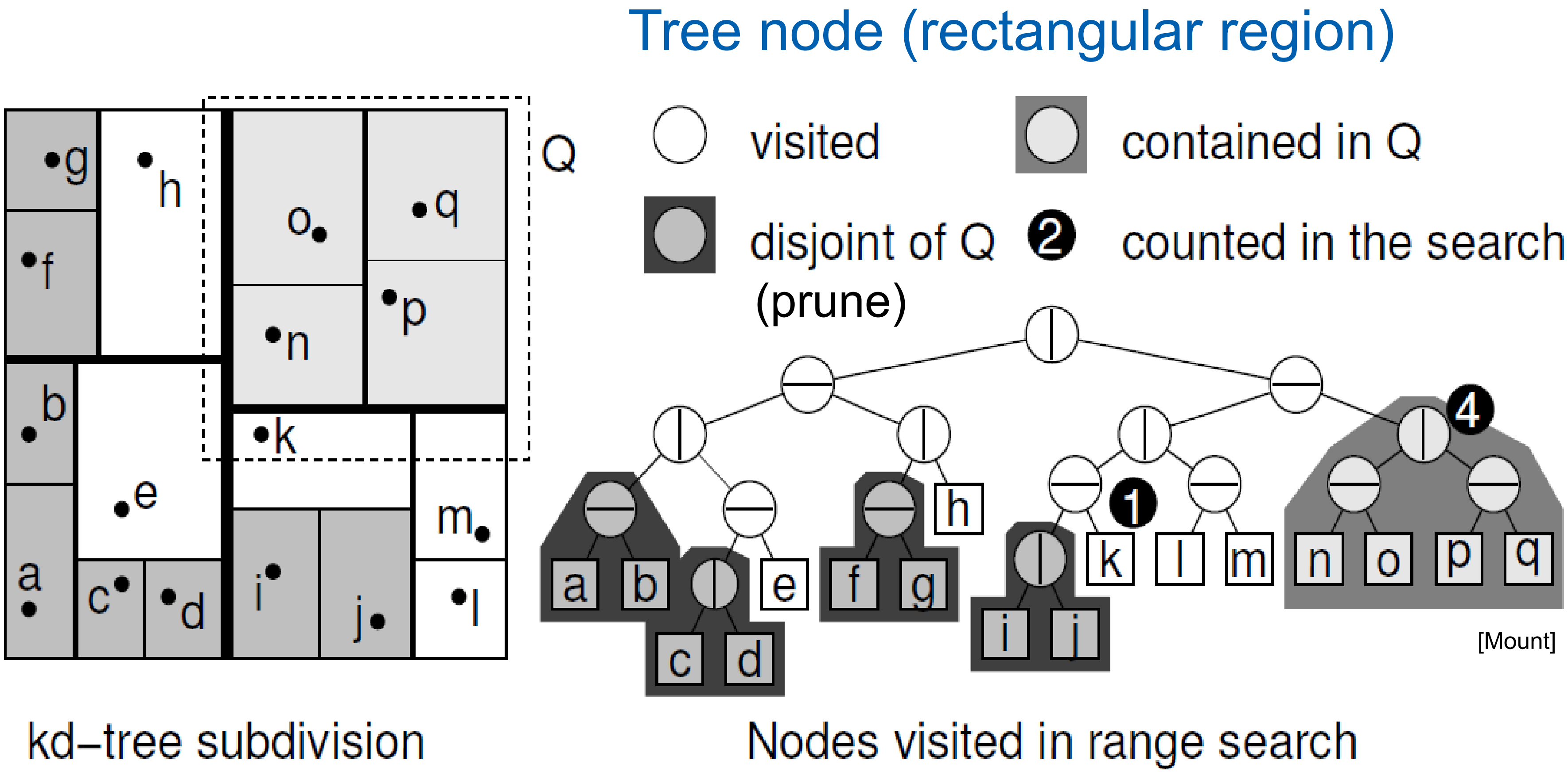
Output: Number of points at leaves below *t* that lie in the range.

```
1.  if (t is a leaf)
2.      if (t.point lies in Q) return 1     // or loop this test for all points in leaf
3.      else return 0                       // visited, not counted
4.  else  // (t is not a leaf)
5.      if (C ý Q = þ )  return 0           … disjoint
6.      else if (C Å Q)  return t.size      … C is fully contained in Q
7.      else
8.          split C along t's cutting value and dimension,
            creating two rectangles C1 and C2.
9.          return rangeCount(t.left, Q, C1) + rangeCount(t.right, Q, C2)
```



// (pictograms refer to the next slide)

Kd-tree rangeCount example

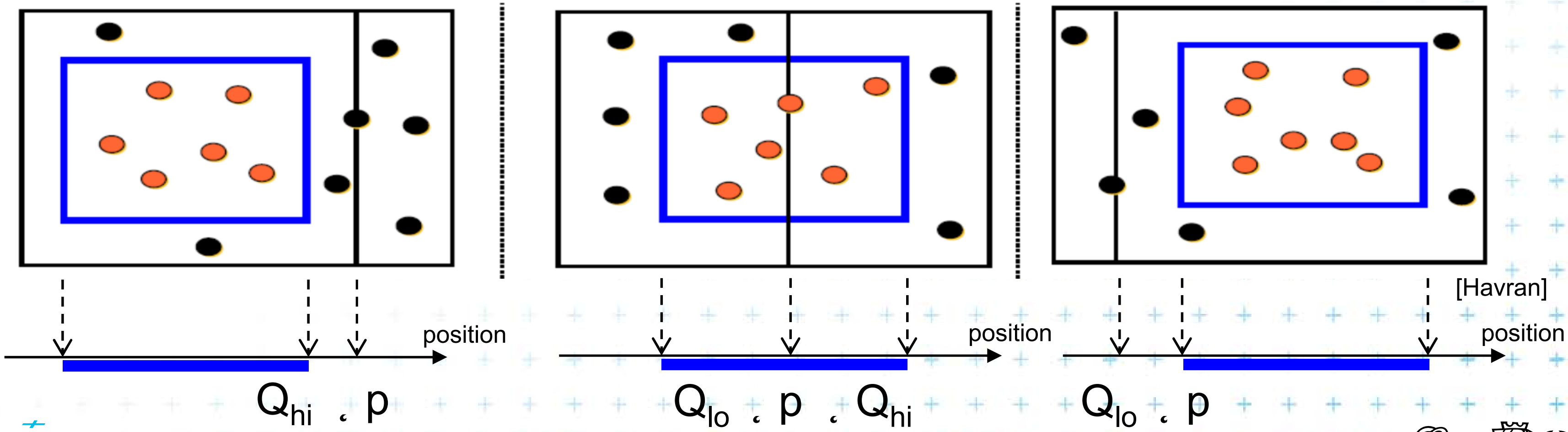


Kd-tree test variants

Test point-interval

b) Compare Q with cutting lines

- Line = Splitting value *p* in one of the dimensions
- Test of single position given by dimension against Q
  1. Line *p* is right from Q … recurse on left child only (prune right child)
  2. Line *p* intersects Q … recurse on both children
  3. Line *p* is left from Q … recurse on right child only (prune left ch.)
- Recursion stops in leaves - traverses the whole tree



Kd-tree rangeSearch (with cutting lines)

int rangeSearch(*t*, *Q*)

Input: The root *t* of (a subtree of a) kD tree and query range *Q*.

Output: Points at leaves below *t* that lie in the range.

```
if (t is a leaf)
    if (t.point lies in Q) report t.point   // or loop test for all points in leaf
    else return
else  (t is not a leaf)
    if (Qhi ¸ t.cutVal) rangeSearch(t.left, Q)   // go left only
    if (Qlo > t.cutVal) rangeSearch(t.right, Q) // go right only
    else
        rangeSearch(t.left, Q)                  // go to both
        rangeSearch(t.right, Q)
```

Kd-tree - summary

- Orthogonal range queries in the plane (in balanced 2d-tree)
  - Counting queries O(ô n) time
  - Reporting queries O(ô n + k) time, where k = No. of reported points
  - Space O(n)
  - Preprocessing: Construction O(n log n) time
    (Proof: if presorted points to arrays in dimensions. Median in O(1) and split in O(n) per level, log n levels of the tree)
- For d−2:
  - Construction O(d n log n), space O(dn), Search O(d n^(1-1/d) + k)

Orthogonal range tree (RT)

- DS highly tuned for orthogonal range queries
- Query times in plane

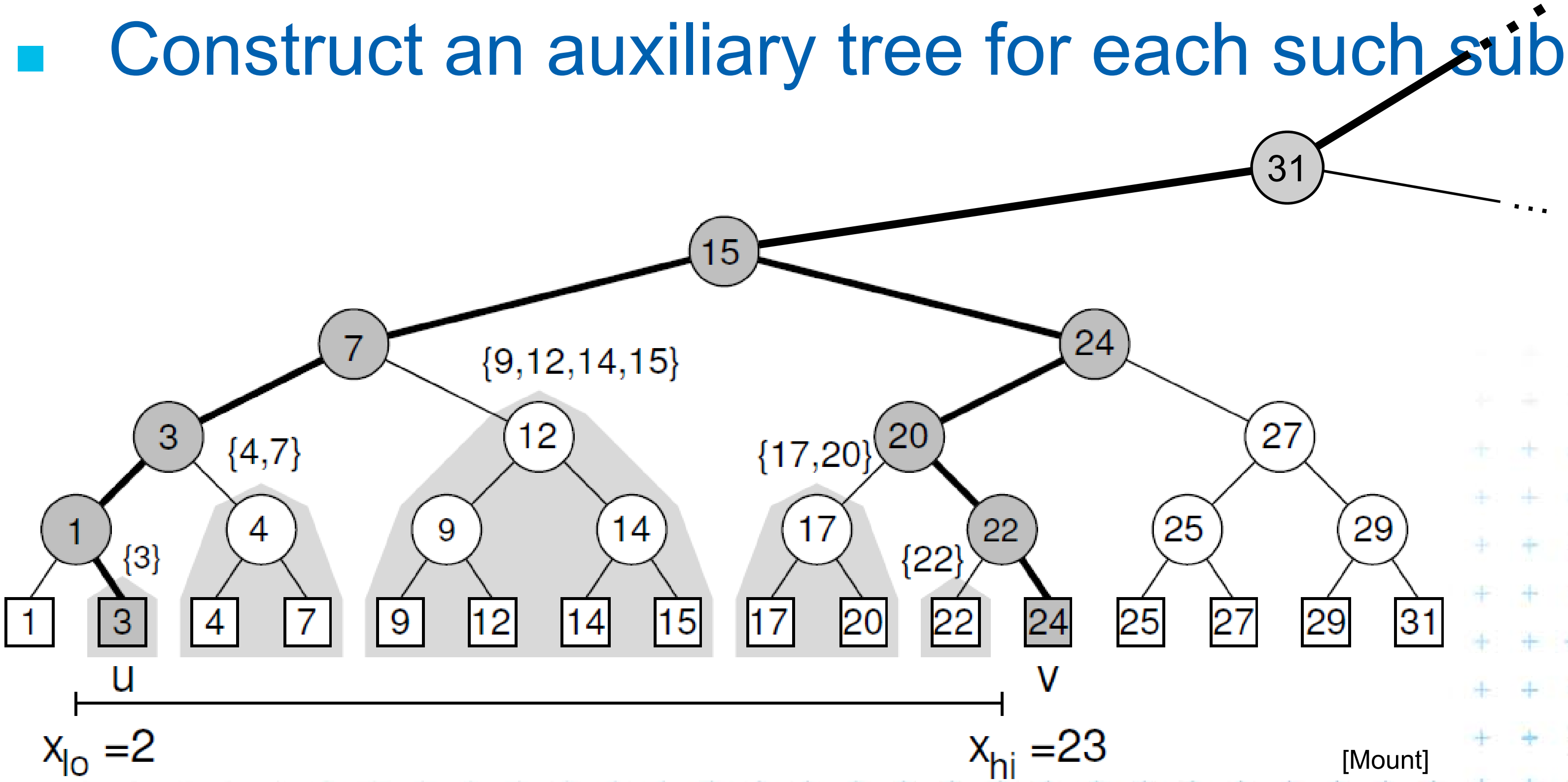
2d tree	versus	range tree
O(ô n + k) time of Kd	>	O(log *n*) time query
O(*n*) space of Kd	<	O(*n* log *n*) space

n = number of points

k = number of reported points

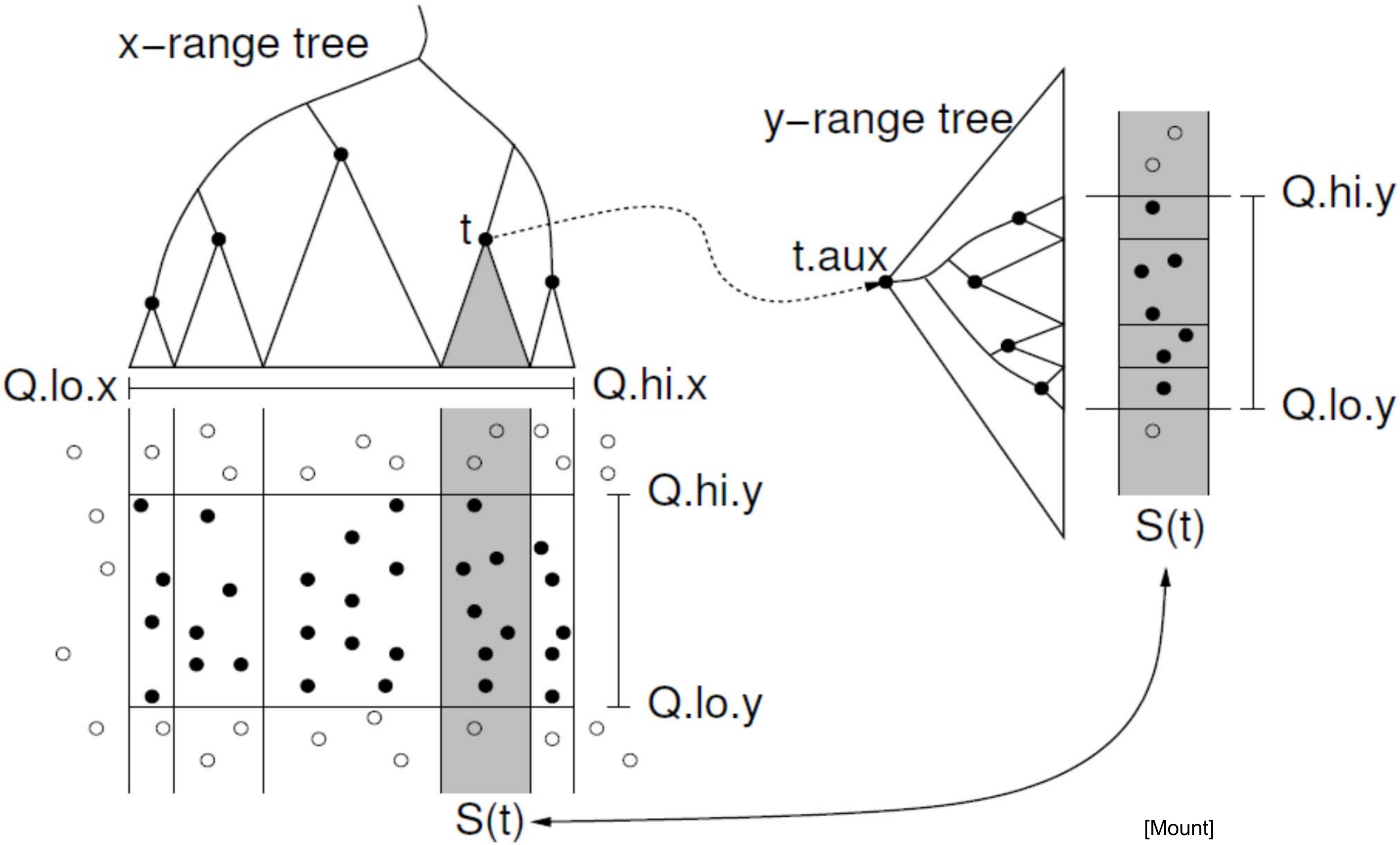
From 1D to 2D range tree

- Search points from [Q.x_{lo}, Q.x_{hi}] [Q.y_{lo}, Q.y_{hi}]
- 1d range tree: log n canonical subsets based on x
- Construct an auxiliary tree for each such subset y



[Mount]

2D range tree



[Mount]

2D range search

2dRangeQuery(*t*, [x:x’] ∘ [y:y’])

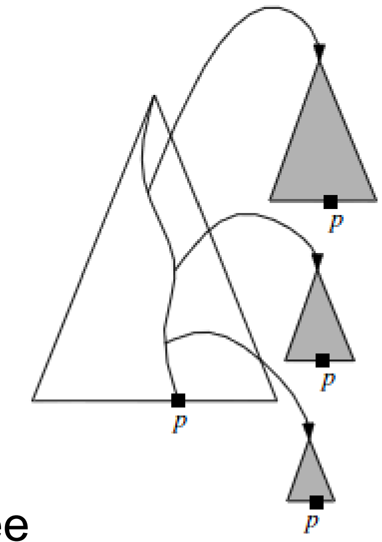
Input: 2d range tree *t* and Query range

Output: All points in *t* laying in the range

```
t_split = FindSplitNode( t, x, x’ )
if( t_split is leaf )
    check if the point in t_split must be reported      … t.x µ [x:x’], t.y µ [y:y’]
else // follow the path to x, calling 1dRangeQuery on y
    t = t_split.left    // path to the left
    while( t is not a leaf )
        if( x ¸ t.x)
            1dRangeQuerry( t_assoc( t.right ), [y:y’] ) // check associated subtree
            t = t.left
        else t = t.right
  check if the point in leaf t must be reported      … t.x ¸ x’, t.y µ [y:y’]
  Similarly for the path to x’      … // path to the right
```

2D range tree

- Search $O(\log^2 n + k)$ – log n in x-, log n in y
- Space O(n log n)
  - O(n) the tree for x-coords
  - O(n log n) trees for y-coords
    - Point p is stored in all canonical subsets along the path from root to leaf with p,
    - once for x-tree level (only in one x-range)
    - each canonical subsets is stored in one auxiliary tree
    - log n levels of x-tree => O(n log n) space for y-trees



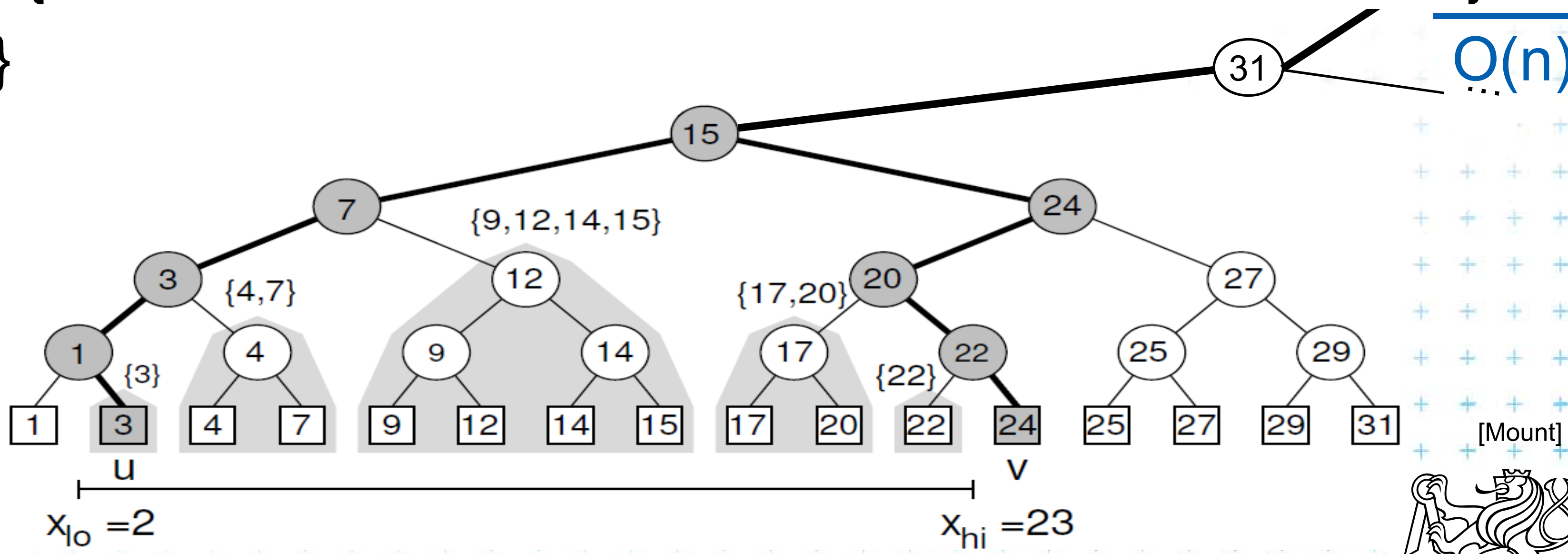
[Berg]

- Construction - O(n log n)
  - Sort points (by *x* and by *y*). Bottom up construction

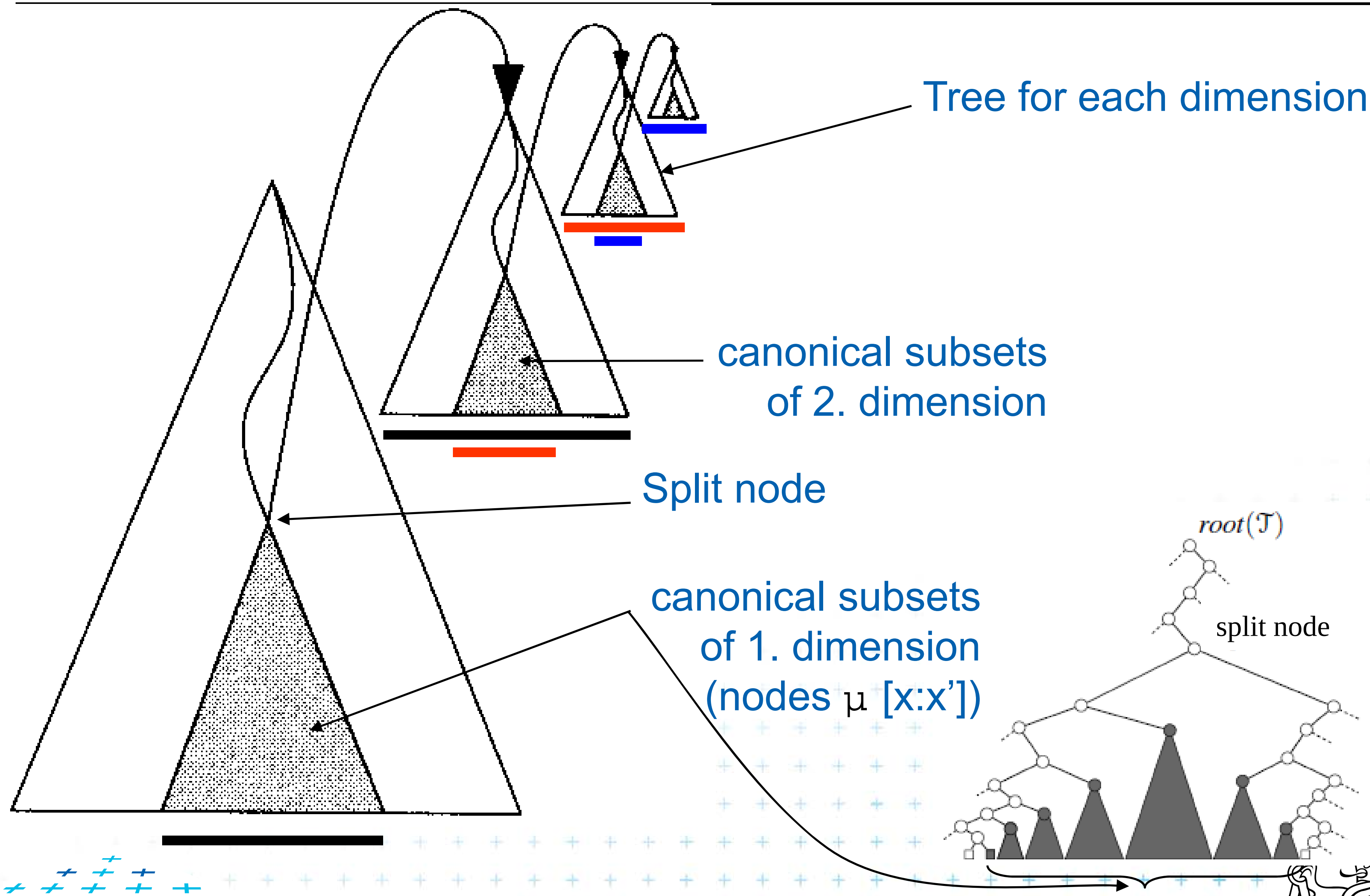
Canonical subsets

- Canonical subsets for this subtree are

	#
{ {1}, {3}, …, {31},	16
{1, 3}, {4, 7}, …, {29, 31}	8
{1, 3, 4, 7}, {9, 12, 14, 15}, …, {25, 27, 29, 31}	4
{1, 3, 4, 7, 9, 12, 14, 15}, {17, 20, 22, 24, 25, 27, 29, 31}	2
{1, 3, 4, 7, 9, 12, 14, 15, 17, 20, 22, 24, 25, 27, 29, 31}	1
}	O(n)



nD range tree (multilevel search tree)

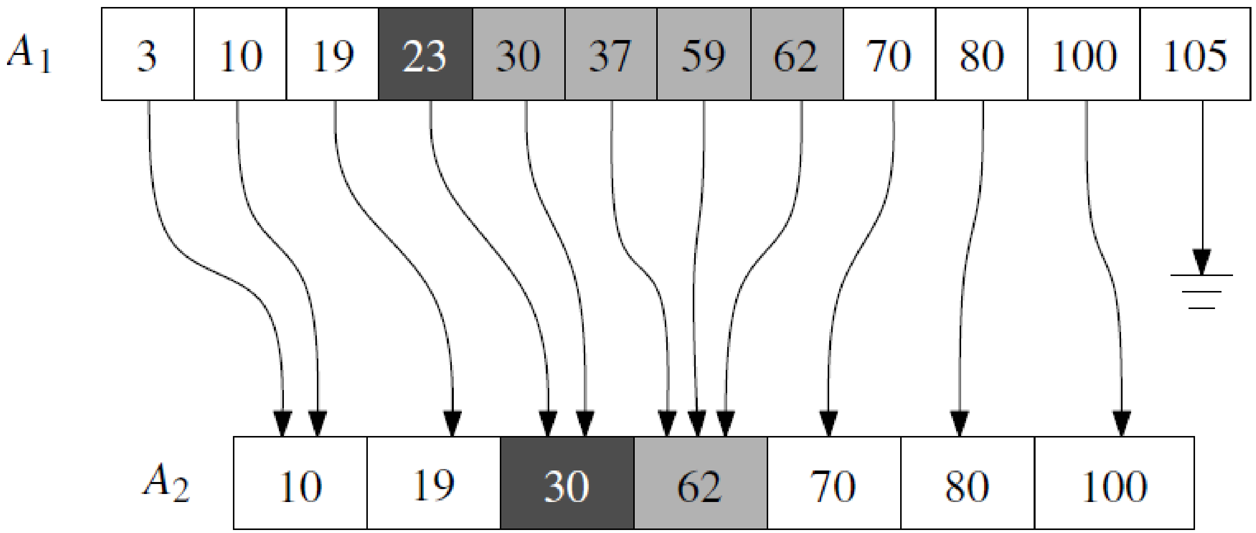


Fractional cascading - principle

- Two sets S_1, S_2 stored in sorted arrays A_1, A_2
- Report objects in both whose keys in [y:y']
- Naïve approach
  - $O(\log n_1 + k_1)$ – search in A_1 + report k_1 elements
  - $O(\log n_2 + k_2)$ – search in A_2 + report k_2 elements
- Fractional cascading – adds pointers from A_1 to A_2
  - $O(\log n_1 + k_1 + 1 + k_2)$ – search in A_1 + report k_1 elements
  - $O(1 + k_2)$ – jump to A_2 + report k_2 elements
  - Saves the $O(\log n_2)$ – search

Fractional cascading – principle for arrays

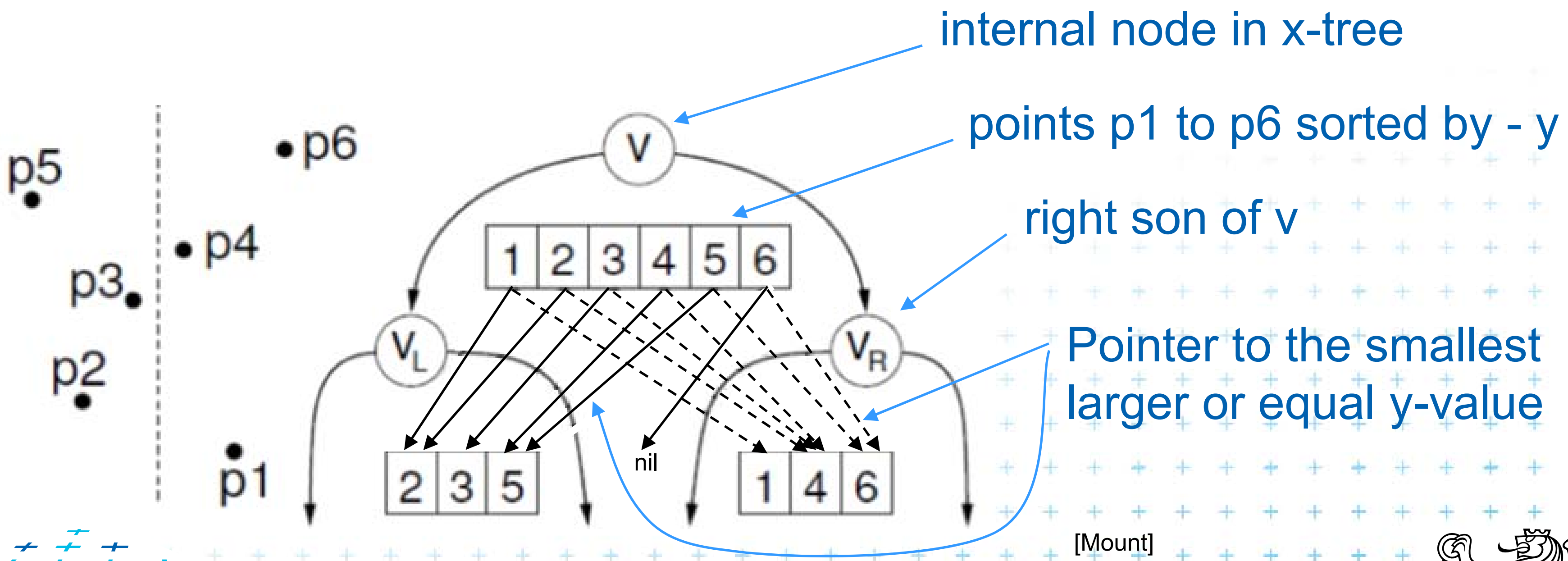
- Add pointers from A_1 to A_2
  - From element in A_1 with a key y_i point to the element in A_2 with the smallest key *larger or equal* to y_i
- Example query with the range [20 : 65]



[Berg]

Fractional cascading in the 2D range tree

- How to save one log n during last dim. search?
  - Store canonical subsets in arrays sorted by y
  - Pointers to subsets for both child nodes v_L and v_R
  - O(1) search in lower levels => in two dimensional search O($\log^2 n$) time -> O($2 \log n$)



Orthogonal range tree - summary

- Orthogonal range queries in plane
  - Counting queries O($\log^2 n$) time,
    or with fractional cascading O($\log n$) time
  - Reporting queries plus O(k) time, for k reported points
  - Space O($n \log n$)
  - Construction O($n \log n$)
- Orthogonal range queries in d-dimensions, d–2
  - Counting queries O($\log^d n$) time,
    or with fractional cascading O($\log^{(d-1)} n$) time
  - Reporting queries plus O(k) time, for k reported points
  - Space O($n \log^{(d-1)} n$)
  - Construction O($n \log^{(d-1)} n$) time

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, Chapter 5, http://www.cs.uu.nl/geobook/
- [Mount] David Mount, - CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland , Lectures 17 and 18. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml
- [Havran] Vlastimil Havran, Materiály k předmětu Datové struktury pro počítačovou grafiku, přednáška č. 6, Proximity search and its Applications 1, CTU FEL, 2007