Geometric Data Structures for Range Searching and Windowing

Balanced Binary Search Tree for 1-D Range Searching

Given: A set $P := \{p_1, \dots, p_n\} \subseteq \mathbb{R}$

Task: Process P into a data structure that supports range queries of the type: Report all points of P that lie in a query interval $[x, x']$

$\Rightarrow$ Balanced binary search tree $\mathcal{T}$
- Leaves of $\mathcal{T}$ store points of P
- Inner vertices store <u>splitting values</u>

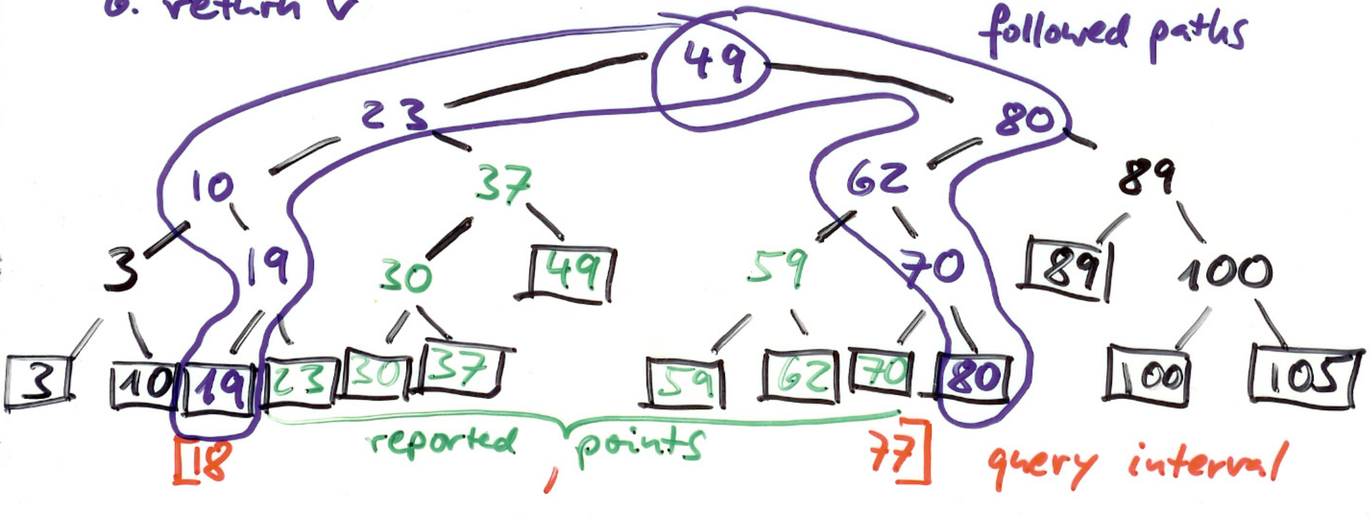
  Inner vertex v stores x_v such that
  - Left subtree of v contains all $p \in P$ with $p \leq x_v$
  - Right subtree of v contains all $p \in P$ with $p > x_v$

Find Split Node $(\mathcal{T}, x, x')$:

Input: $\mathcal{T}$ and $x \leq x'$

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

```
v := root(T)
while v is no leaf and (x' ≤ x_v or x_v < x) do
    if x' ≤ x_v then
        v := left_child(v)
    else v := right_child(v)
return v
```



Algorithm 1D_Range_Query(T, $[x, x']$)

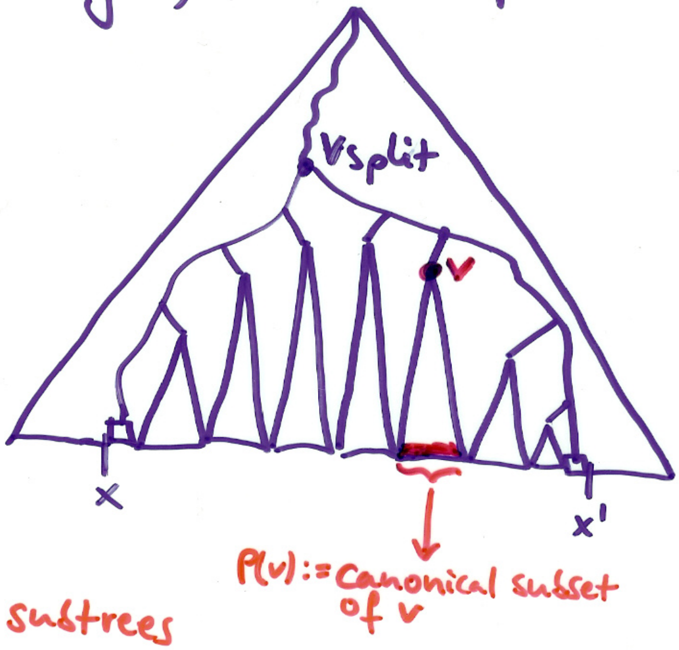
Input: A binary search tree T, a range $[x, x']$

Output: All points stored in leaves of T that lie in $[x, x']$

```
vsplit := FindSplitNode(T, x, x')
if vsplit is a leaf then
    Check if point stored at vsplit must be reported
else // Follow path to x and report points in subtrees right of path
    v := left_child(vsplit)
    while v is no leaf do
        if x ≤ x_v then
            Report_Subtree(right_child(v))
            v := left_child(v)
        else v := right_child(v)
    Check if point stored at leaf v must be reported
Similarly follow path to x' and report points in subtrees left of path
```

Analysis:

- Preprocessing:
  - $O(n \log n)$ construction time
  - $O(n)$ storage
- Query:
  - $O(k + \log n)$ time to report all k points in query range



$O(\log n)$ subtrees

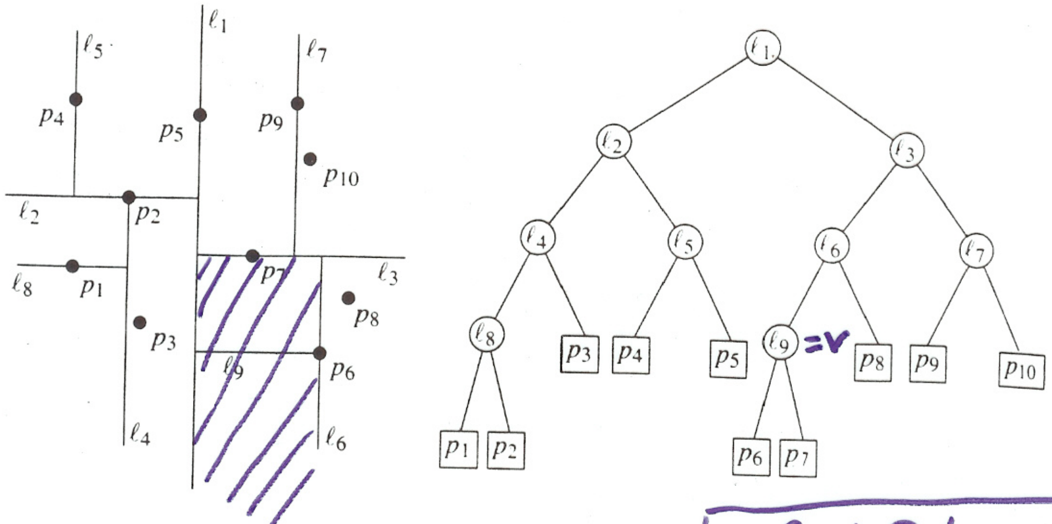
$\Rightarrow$ Union of canonical subsets = output

KD-Trees for 2-D Range Searching

Given: A set $P := \{p_1, \dots, p_n\} \subseteq \mathbb{R}^2$

Task: Process P into a data structure that allows fast 2D range queries: Report all points in P that lie in the query rectangle $[x, x'] \times [y, y']$

$\Rightarrow$ Recursively split P into two sets of same size, alternatingly along a vertical or horizontal line



region(v)

- Sort P by x- and by y-coordinate in advance
- Use these two sorted lists to find median
- Pass sorted lists into recursive calls

$$\Rightarrow T(n) = \begin{cases} O(1) & , n = 1 \\ O(n) + 2T(\lceil n/2 \rceil) & , n > 1 \end{cases}$$

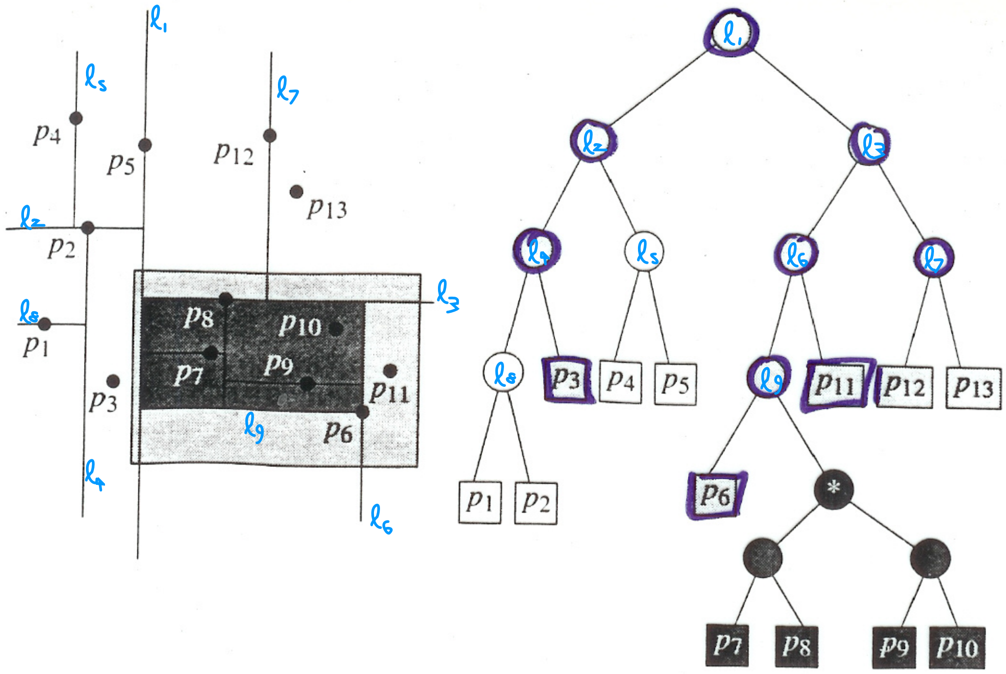
$= O(n \log n)$ time

Algorithm BUILDKDTREE(P, *depth*)

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

Output. The root of a kd-tree storing P.

```
if P contains only one point
   then return a leaf storing this point
   else if depth is even
           then Split P into two subsets with a vertical line ℓ through the
                median x-coordinate of the points in P. Let P1 be the set of
                points to the left of ℓ or on ℓ, and let P2 be the set of points
                to the right of ℓ.
           else Split P into two subsets with a horizontal line ℓ through
                the median y-coordinate of the points in P. Let P1 be the
                set of points below ℓ or on ℓ, and let P2 be the set of points
                above ℓ.
        v_left ← BUILDKDTREE(P1, depth + 1)
        v_right ← BUILDKDTREE(P2, depth + 1)
        Create a node v storing ℓ, make v_left the left child of v, and make
        v_right the right child of v.
        return v
```



$lc(v)$ = left_child(v)
$rc(v)$ = right_child(v)

Algorithm SEARCHKDTREE(ν, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below ν that lie in the range.

```
if ν is a leaf
    then Report the point stored at ν if it lies in R.
    else if region(lc(ν)) is fully contained in R
            then REPORTSUBTREE(lc(ν))
            else if region(lc(ν)) intersects R
                    then SEARCHKDTREE(lc(ν), R)
         if region(rc(ν)) is fully contained in R
            then REPORTSUBTREE(rc(ν))
            else if region(rc(ν)) intersects R
                    then SEARCHKDTREE(rc(ν), R)
```

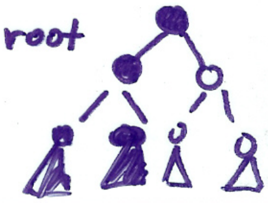
// $\text{region}(lc(v)) = \text{region}(v) \cap \ell(v)^{\text{left}}$
$\rightsquigarrow$ compute on the fly

Theorem: A kd-tree for a set P of n points can be constructed in $O(n \log n)$ time and uses $O(n)$ space. A rectangular range query can be answered in $O(\sqrt{n} + k)$ time; k = # reported points.
(Generalization to d dimensions: Also $O(n)$ storage, $O(n \log n)$ construction time, but $O(n^{1-1/d} + k)$ query time.)

Proof Sketch:

- Sum of # of visited vertices in ReportSubtree $= O(k)$
- # visited vertices that are not in one of the reported subtrees $= O(\#\text{regions}(v)$ intersected by a line$)$

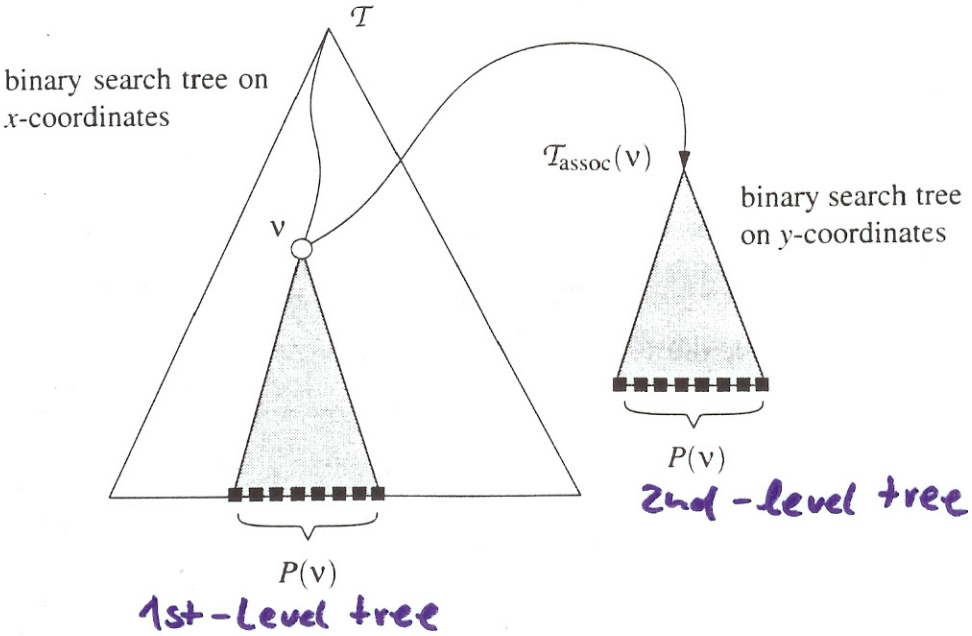
$\rightsquigarrow Q(n) :=$ # intersected regions in kd-tree of n points whose root contains vertical splitting line



$\rightsquigarrow Q(n) = 2 + 2Q\left(\frac{n}{4}\right),\ n > 1 \Rightarrow Q(n) = O(\sqrt{n})$

Range Trees in 2D

Task: Answer 2D rectangular range query within $O(\log^2 n + k)$ query time



Algorithm BUILD2DRANGETREE(P)
Input. A set P of points in the plane.
Output. The root of a 2-dimensional range tree.

1. Construct the associated structure: Build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates of the points in P. Store at the leaves of $\mathcal{T}_{\text{assoc}}$ not just the y-coordinate of the points in P_y, but the points themselves.
2. **if** P contains only one point
3.   **then** Create a leaf ν storing this point, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of ν.
4.   **else** Split P into two subsets; one subset P_{left} contains the points with x-coordinate less than or equal to x_{mid}, the median x-coordinate, and the other subset P_{right} contains the points with x-coordinate larger than x_{mid}.
5. $\nu_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})
6. $\nu_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})
7.   Create a node ν storing x_{mid}, make ν_{left} the left child of ν, make ν_{right} the right child of ν, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of ν.
8. **return** ν

Lemma: $O(n \log n)$ storage; $O(n \log n)$ construction time

Proof:
- $O(n)$ storage at every level of $\mathcal{T}$ $\rightsquigarrow$ $O(n \log n)$ altogether
- Presort P into two lists, sorted by x- or y-coordinate
  - $\rightsquigarrow$ Construct search trees in linear time
  - $\rightsquigarrow$ Construction time proportional to storage

$\square$

Algorithm 2DRANGEQUERY($\mathcal{T}$, $[x:x'] \times [y:y']$)
Input. A 2-dimensional range tree $\mathcal{T}$ and a range $[x:x'] \times [y:y']$.
Output. All points in $\mathcal{T}$ that lie in the range.

$\nu_{\text{split}} \leftarrow$ FINDSPLITNODE($\mathcal{T}, x, x'$)
if ν_{split} is a leaf
  then Check if the point stored at ν_{split} must be reported.
  else (* Follow the path to x and call 1DRANGEQUERY on the subtrees right of the path. *)
    $\nu \leftarrow lc(\nu_{\text{split}})$
    while ν is not a leaf
      do if $x \leqslant x_\nu$
        then 1DRANGEQUERY($\mathcal{T}_{\text{assoc}}(rc(\nu))$, $[y:y']$)
          $\nu \leftarrow lc(\nu)$
        else $\nu \leftarrow rc(\nu)$
    Check if the point stored at ν must be reported.
    Similarly, follow the path from $rc(\nu_{\text{split}})$ to x', call 1DRANGEQUERY with the range $[y:y']$ on the associated structures of subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.

Theorem: A range tree storing n points in the plane can be constructed in $O(n \log n)$ time and storage. A rectangular range query can be answered in $O(\log^2 n + k)$ time; $k = \#$ reported points.

Proof: Query time $= \sum_{\text{visited } v} O(\log n + k_v) = O\Big(\underbrace{\sum_{\text{visited } v} \log n}_{= \log^2 n} + \underbrace{\sum_{\text{visited } v} k_v}_{= k}\Big)$

Search paths in $\mathcal{T}$ have length $\log^2 n$

Generalizing Range Trees to Higher Dimensions

- d Levels of trees; one level per coordinate

  1st level: balanced bin. search tree on 1st coordinate

  2nd Level: For each v in 1st level construct $(d-1)$-dim. range tree for points in $P(v)$ restricted to last $(d-1)$ coordinates

  ⋮

- Query recursively as before $\rightsquigarrow O(\log^d n + k)$ time

Theorem: A range tree storing n points in $\mathbb{R}^d$ can be constructed in $O(n \log^{d-1} n)$ time and storage. A rectangular range query can be answered in $O(\log^d n + k)$ time.

Proof: $T_d(n) = \underset{\text{constmct bal. bin. search tree}}{O(n \log n)} + \underset{\text{\#levels}}{O(\log n)} \cdot \underset{\text{construction time per level}}{T_{d-1}(n)}$ construction time

$T_2(n) = O(n \log n) \Rightarrow T_d(n) = O(n \log^{d-1} n)$

Query time $Q_d(n) = \underset{\text{1st level}}{O(\log n)} + \underset{\text{\#subtrees}}{O(\log n)} \cdot Q_{d-1}(n)$

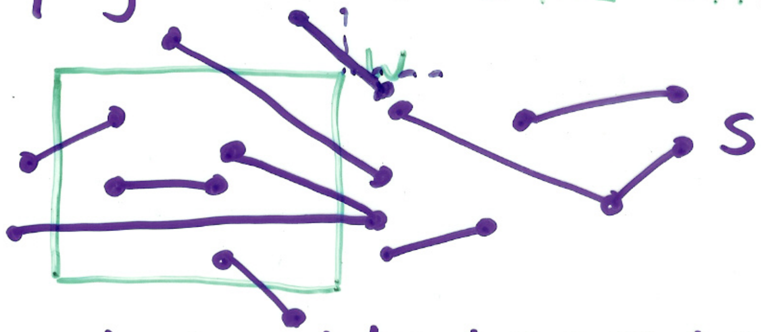
$Q_2(n) = O(\log^2 n) \Rightarrow Q_d(n) = O(\log^d n)$

Theorem: Using fractional cascading the query time of a d-dimensional range tree can be sped up to $O(\log^{d-1} n + k)$; thus to $O(\log n + k)$ for $d = 2$.

Windowing Problem

Given: A set S of n (non-intersecting) line segments in the plane

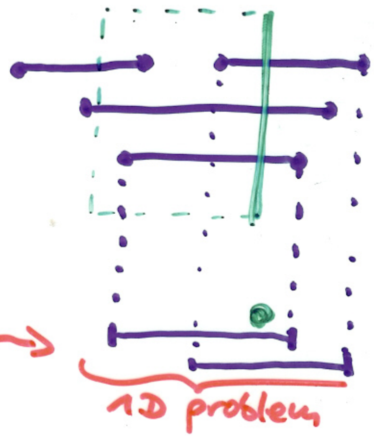
Task: Process S into a data structure such that the following windowing query can be answered efficiently:
Report all segments in S that intersect a given query window $W := [x, x'] \times [y, y']$



Segments having at least one endpoint in W can be found by range queries in range trees
→ $O(\log n + k)$ time (with fractional cascading)

Subproblem I (for horizontal segments)

Process a set of horizontal line segments s.t. segments intersecting a vertical query segment can be reported efficiently.
⤳ Consider query line instead of segment →



Subproblem II (1 dimensional):

Given: A set $I = \{[x_1, x_1'], \dots, [x_n, x_n']\}$ of intervals in $\mathbb{R}$

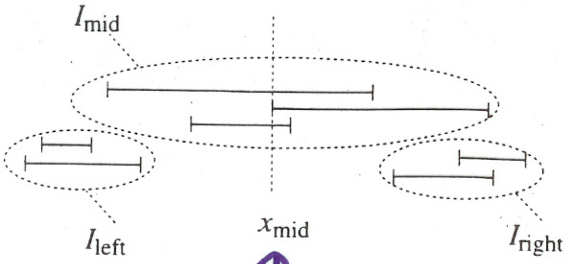
Task: Process I into a data structure which supports queries of the type: Report all intervals that contain a query point

→ Interval trees

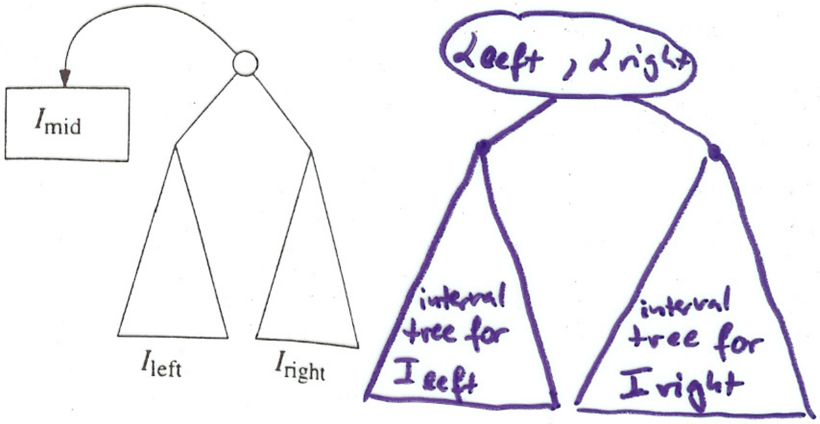
→ Segment trees

Interval Trees

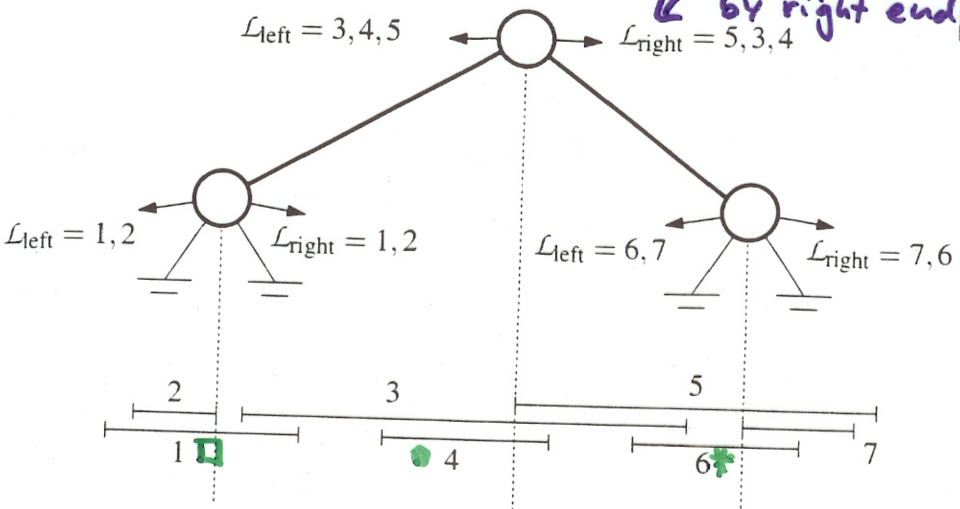
$I = I_{left} \dot\cup I_{mid} \dot\cup I_{right}$



↑ median of $2n$ interval endpoints



List of intervals in I_{mid} sorted by right endpoint



Interval Tree

Lemma: An interval tree on a set of n intervals uses $O(n)$ storage and has depth $O(\log n)$.

Proof: Each interval is stored in a set I_{mid} only once $\rightarrow O(n)$ storage.

Algorithm CONSTRUCTINTERVALTREE(I)
Input. A set I of intervals on the real line.
Output. The root of an interval tree for I.

```
if I = ∅
   then return an empty leaf
   else Create a node ν. Compute x_mid, the median of the set of interval
        endpoints, and store x_mid with ν.
        Compute I_mid and construct two sorted lists for I_mid: a list L_left(ν)
        sorted on left endpoint and a list L_right(ν) sorted on right endpoint.
        Store these two lists at ν.
        lc(ν) ← CONSTRUCTINTERVALTREE(I_left)
        rc(ν) ← CONSTRUCTINTERVALTREE(I_right)
        return ν
```

Time analysis: $O(\underset{|I|}{\cancel{n}} + |I_{mid}| \cdot \log |I_{mid}|)$ per vertex $\Rightarrow O(n \log n)$

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

```
if ν is not a leaf
   then if q_x < x_mid(ν)
           then Walk along the list L_left(ν), starting at the interval with the
                leftmost endpoint, reporting all the intervals that contain
                q_x. Stop as soon as an interval does not contain q_x.
                QUERYINTERVALTREE(lc(ν), q_x)
           else Walk along the list L_right(ν), starting at the interval with the
                rightmost endpoint, reporting all the intervals that contain
                q_x. Stop as soon as an interval does not contain q_x.
                QUERYINTERVALTREE(rc(ν), q_x)
```

Time analysis:

- $O(1+k_v)$ time at vertex v; $k_v :=$ #intervals reported at v
- Visit ≤ 1 node at any depth

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

Theorem: An interval tree for a set of n intervals ~~in the plane~~ can be constructed in $O(n \log n)$ time and uses $O(n)$ storage. All intervals that contain a query point can be reported in $O(\log n + k)$ time; $k =$ # reported intervals.

Segment Trees

Let $p_1, p_2, \dots, p_m$ the sorted list of distinct interval endpoints of I

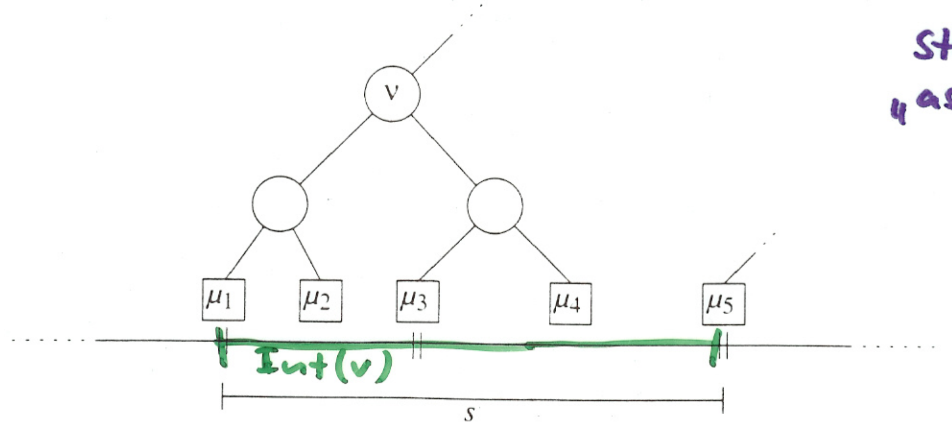
→ Consider partitioning into elementary intervals

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

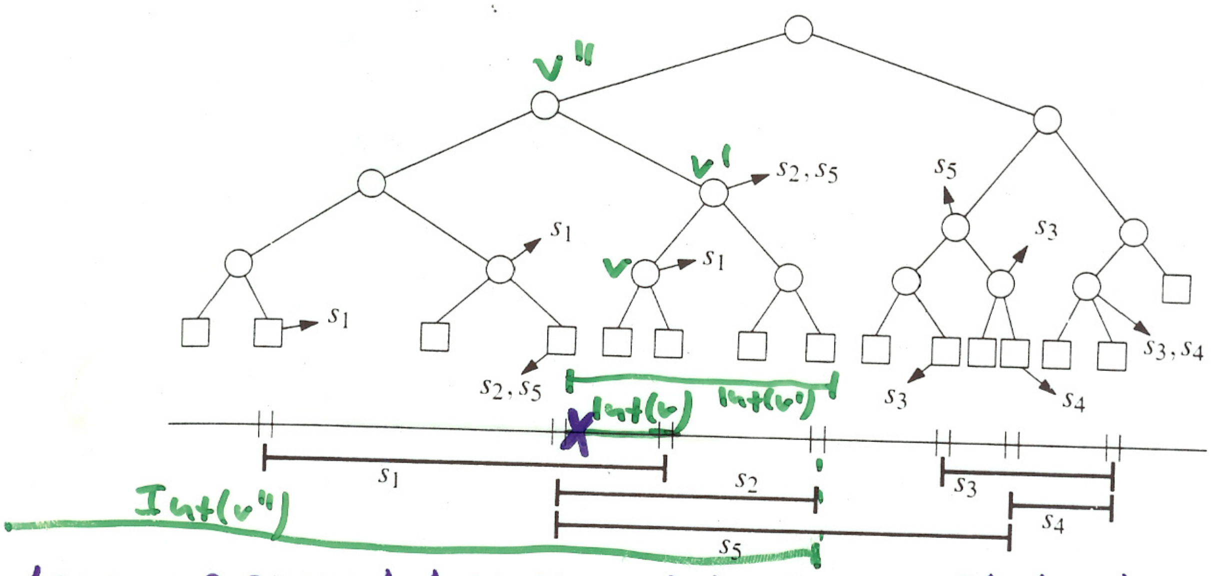
- Balanced binary search tree $\mathcal{T}$ with leaves corresponding to elementary intervals
- $\mathrm{Int}(\mu) :=$ elementary interval corresponding to leaf μ

  $\mathrm{Int}(v) :=$ union of $\mathrm{Int}(\mu)$ of all leaves in subtree rooted at v
- Each node or leaf v stores
  - $\mathrm{Int}(v)$
  - the canonical subset $I(v) \subseteq I$:

    $I(v) := \{[x, x'] \in I \mid \mathrm{Int}(v) \subseteq [x, x'] \text{ and } \mathrm{Int}(\mathrm{parent}(v)) \not\subseteq [x, x']\}$



Store intervals „as high as possible".



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

Proof idea: Any interval is stored in a set $I(v)$ for at most two nodes at the same level of $\mathcal{T}$.

Algorithm QUERYSEGMENTTREE(ν, 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 in $I(\nu)$.
if ν is not a leaf
    then if $q_x \in \mathrm{Int}(lc(\nu))$
        then QUERYSEGMENTTREE($lc(\nu), q_x$)
        else QUERYSEGMENTTREE($rc(\nu), q_x$)

Time analysis:
- Visit one node per level
- Spend $O(1 + k_v)$ per node v

$\Big\}$ $O(\log n + k)$ time

Segment_Tree_Construction:

1) Sort interval endpoints of I $\rightsquigarrow$ elementary intervals
2) Construct balanced bin. search tree on elem. intervals

$\Big\}$ $O(n \log n)$

3) Determine $\mathrm{Int}(v)$ bottom-up $O(n)$

4) Compute canonical subsets by incrementally inserting the intervals $[x, x'] \in I$ into T, using Insert Segment Tree

Algorithm INSERTSEGMENTTREE($\nu, [x : x']$)

Input. The root of a (subtree of a) segment tree and an interval.

Output. The interval will be stored in the subtree.

if $\mathrm{Int}(\nu) \subseteq [x : x']$
    then store $[x : x']$ at ν
    else if $\mathrm{Int}(lc(\nu)) \cap [x : x'] \neq \emptyset$
            then INSERTSEGMENTTREE($lc(\nu), [x : x']$)
        if $\mathrm{Int}(rc(\nu)) \cap [x : x'] \neq \emptyset$
            then INSERTSEGMENTTREE($rc(\nu), [x : x']$)

Time analysis:
- Spend constant time per node
- If we don't store $[x, x']$ at v, then $x \in \mathrm{Int}(v)$ or $x' \in \mathrm{Int}(v)$
- Each interval stored $\leq$ twice at each level.

At most one node per level whose interval contains x (similar for x')

$\rightsquigarrow$ Visit ≤ 4 nodes per level $\Rightarrow O(\log n) \Rightarrow O(n \log n)$ together

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

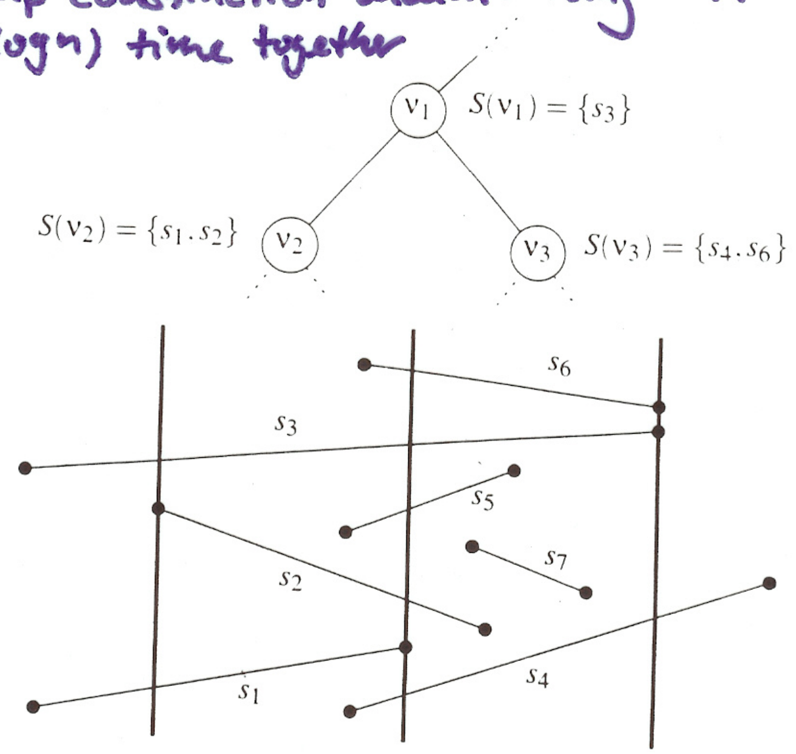
2D Windowing Revisited

Given: A set S of n disjoint segments in the plane

Task: Process S into a data structure such that all segments intersecting a vertical query segment $q := q_x \times [q_y, q_y']$ can be reported efficiently

- Build segment tree T based on x-intervals of segments in S
  $\rightsquigarrow$ each $Int(v) \hat{=} Int(v) \times (-\infty, \infty)$ vertical slab
- $I(v) \hat{=} S(v)$ canonical subset of segments spanning vertical slab at v
- Store $S(v)$ in binary search tree $T(v)$ based on vertical order of segments

Analysis:
- Storage $O(n \log n)$
- Bottom-up construction maintaining vertical order of segments
  $\rightsquigarrow O(n \log n)$ time together



Query algorithm:
- Search regularly for q_x in T
- In every visited node v report segments in $T(v)$ between q_y and q_y' (1D range query)
  $\rightarrow O(\log n + k_v)$ time for $T(v)$ $\rightsquigarrow O(\log^2 n + k)$ altogether

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