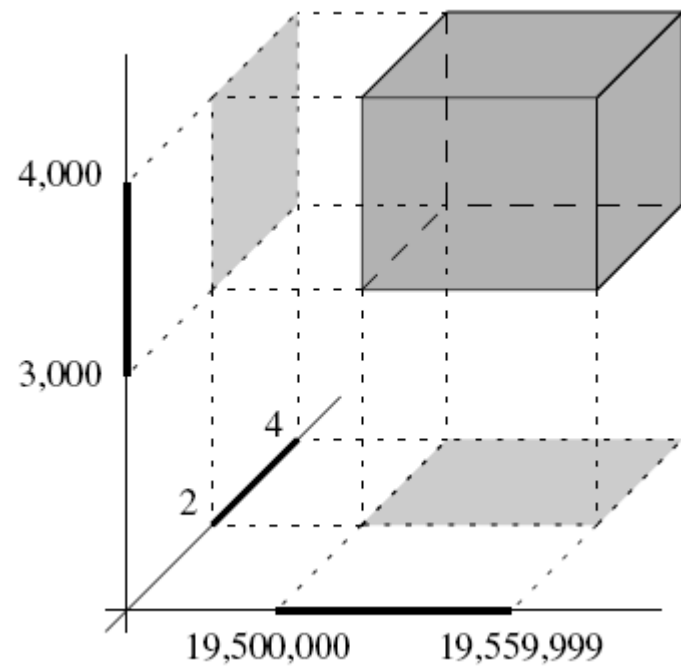
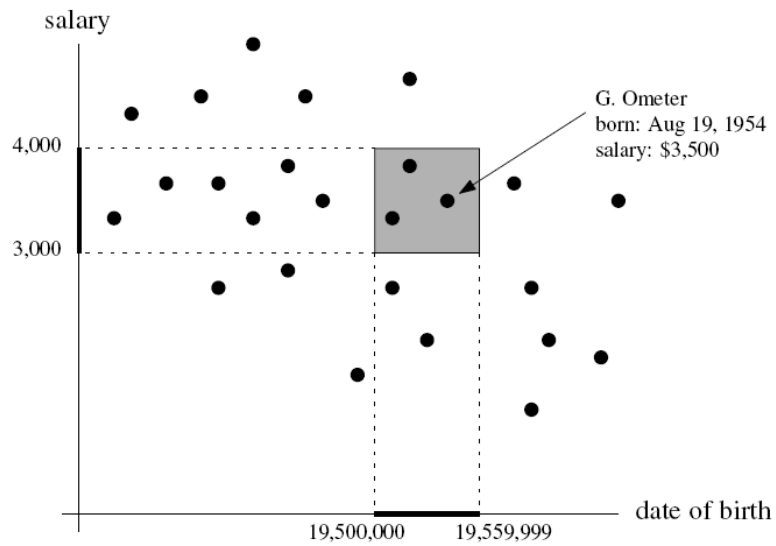


Orthogonal Range Search Problem

Nattee Niparnan

The Problem



The Problem : Formalizes

- Input
 - A set P of n -dimension points $\{x_i = (a_1, a_2, \dots, a_n)\}$
 - A “range” query a conjunction of
 - $\min_1 \leq a_1 \leq \max_1$
 - $\min_2 \leq a_2 \leq \max_2$
 - ...
 - $\min_n \leq a_n \leq \max_n$
- Output
 - All points in P satisfying the condition

Orthogonal

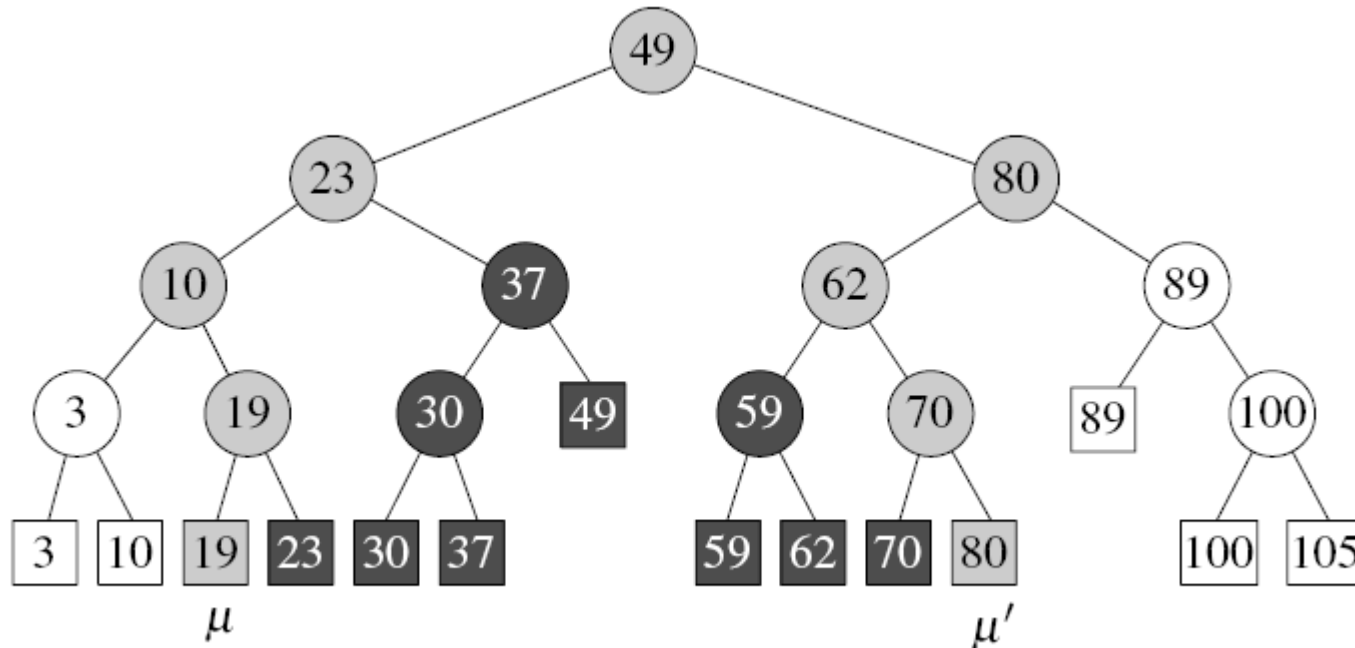
- Constraints are parallel to the axis

Nature of the Problem

- The set P is seldom updated
- There will be several queries

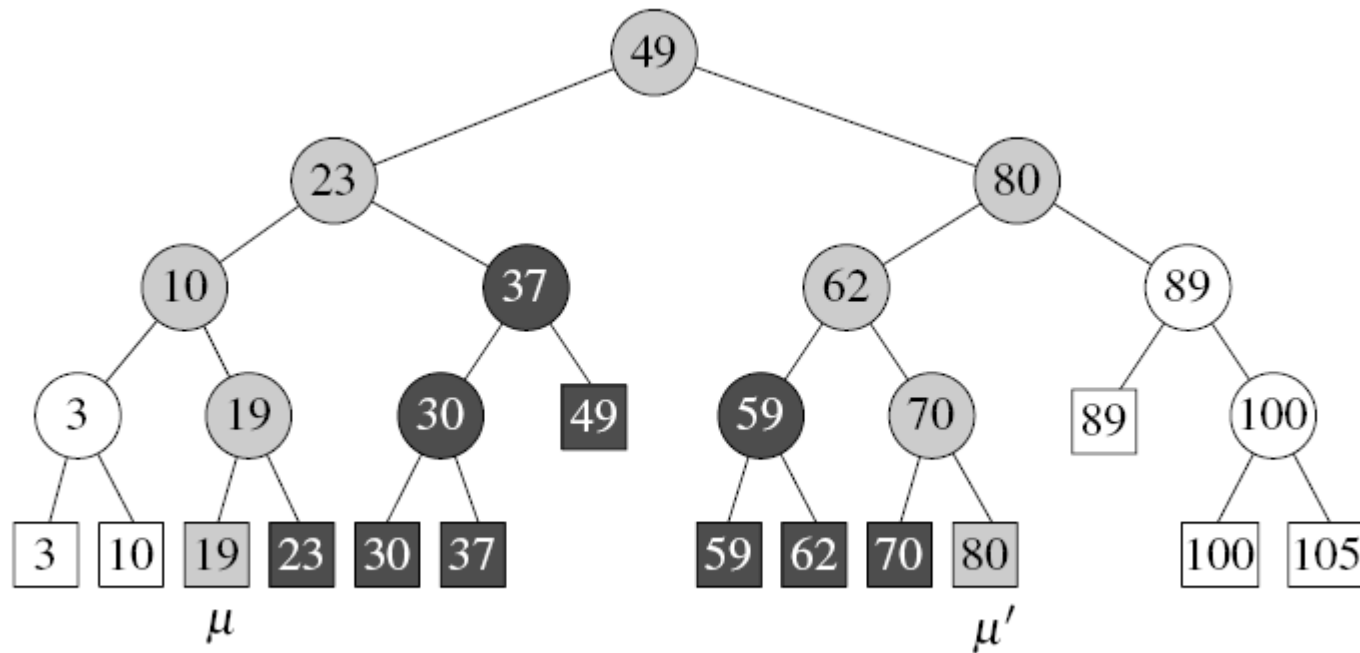
1-D Version

- Store data in an balanced tree
- Leaf node contains data
- Internal node contains max of left subtree
 - μ = largest data not exceeding the lower bound
 - μ' = smallest data not less than the upper bound



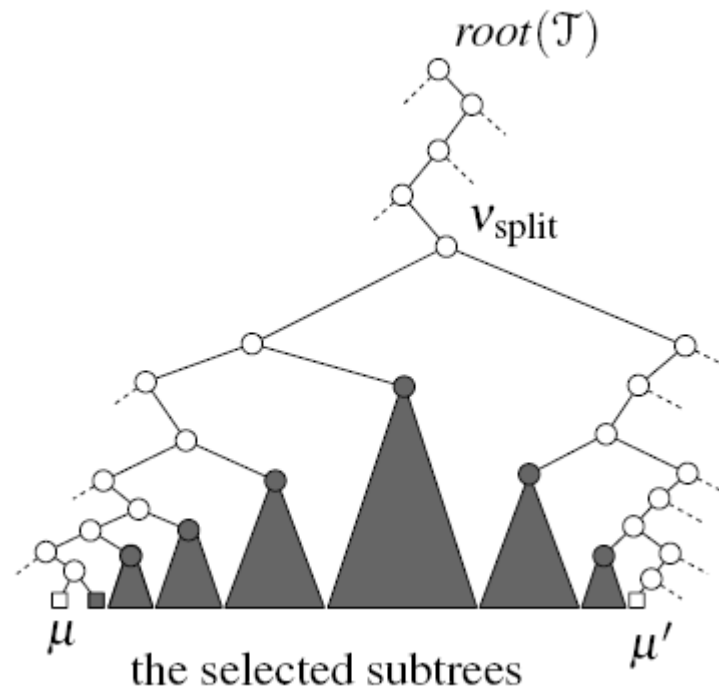
1-D Version

- Report everything between μ and μ'
 - Might include μ and μ'



The Algorithm

- Find the “split node”
 - Split node = node such that μ and μ' are on different subtree



Finding Split node

FINDSPLITNODE(\mathcal{T}, x, x')

Input. A tree \mathcal{T} and two values x and x' with $x \leq x'$.

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

1. $v \leftarrow \text{root}(\mathcal{T})$
2. **while** v is not a leaf **and** $(x' \leq x_v \text{ or } x > x_v)$
3. **do if** $x' \leq x_v$
4. **then** $v \leftarrow lc(v)$
5. **else** $v \leftarrow rc(v)$
6. **return** v

Report the Search

- For the left subtree containing μ
 - The result is the leaves of right subtree of the node that μ is on the left subtree
- For the right subtree containing μ'
 - The result is the leaves of left subtree of the node that μ' is on the right subtree

The Full Algorithm

Algorithm 1DRANGEQUERY($\mathcal{T}, [x : x']$)

Input. A binary search tree \mathcal{T} and a range $[x : x']$.

Output. All points stored in \mathcal{T} that lie in the range.

1. $v_{\text{split}} \leftarrow \text{FINDSPLITNODE}(\mathcal{T}, x, x')$
2. **if** v_{split} is a leaf
3. **then** Check if the point stored at v_{split} must be reported.
4. **else** (* Follow the path to x and report the points in subtrees right of the path. *)
5. $v \leftarrow lc(v_{\text{split}})$
6. **while** v is not a leaf
7. **do if** $x \leq x_v$
8. **then** REPORTSUBTREE($rc(v)$)
9. $v \leftarrow lc(v)$
10. **else** $v \leftarrow rc(v)$
11. Check if the point stored at the leaf v must be reported.
12. Similarly, follow the path to x' , report the points in subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.

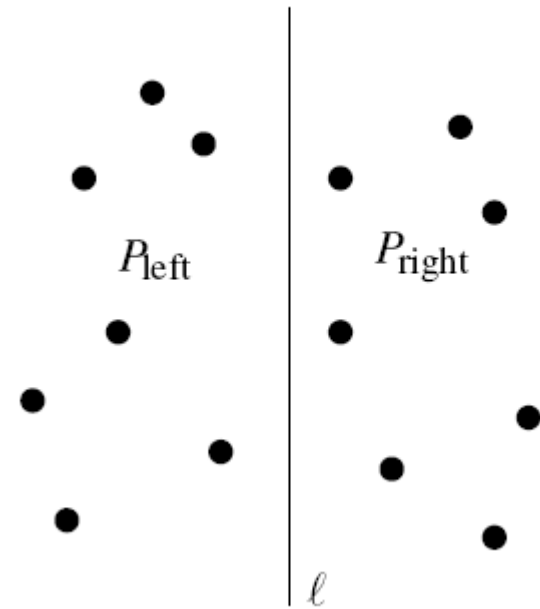
Analysis

- The space requirement is $O(n)$
 - Because it is a balanced tree
- The time for building the tree is $O(n \log n)$
- Reporting
 - $O(\log n + K)$
 - Where K is the number of points satisfying the conditions

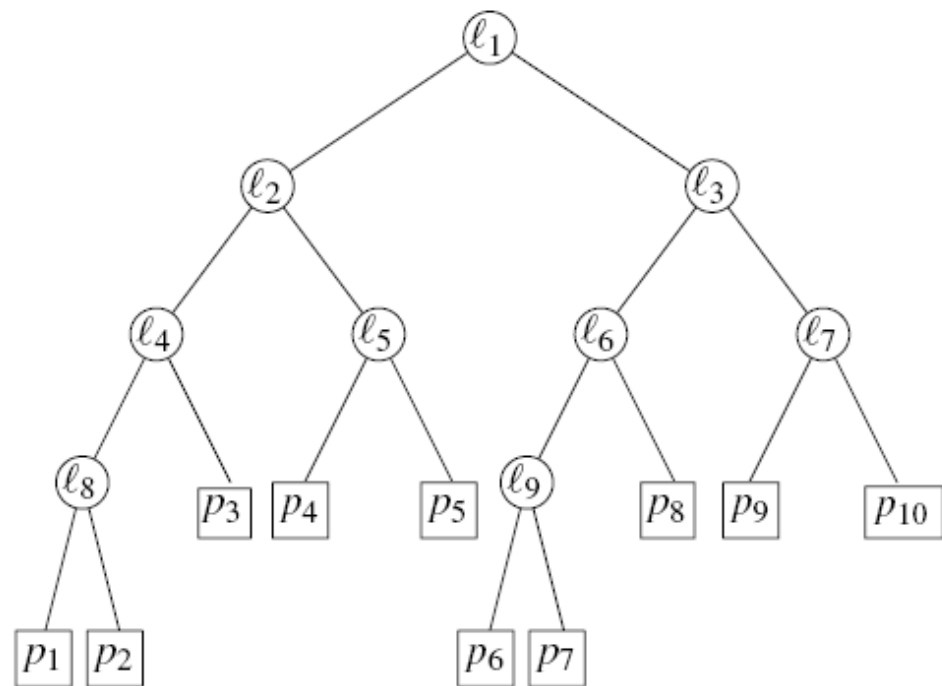
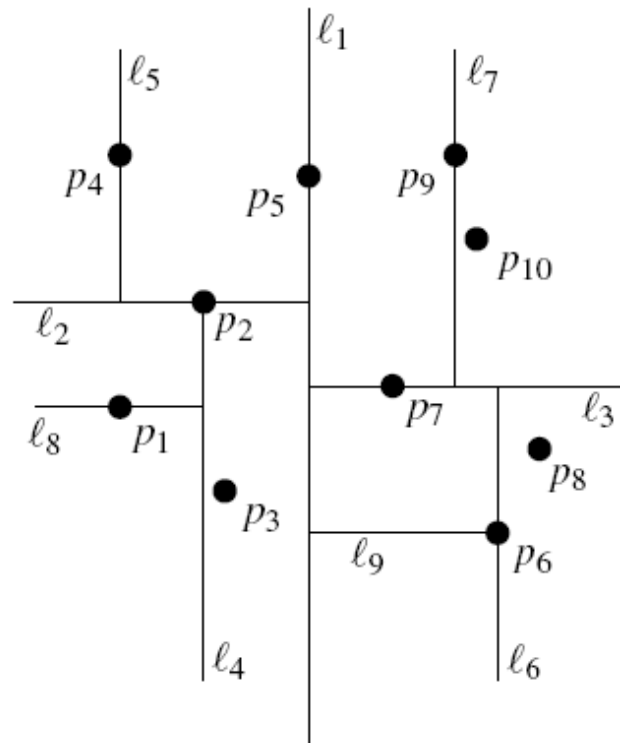
Generalizes to 2-dimension

- The KD-Tree
 - Split the tree by half
 - Alternately switch between x coord and y coord
 - Root splits the y coord
 - 1st level splits the x coord
 - 2nd level splits the y coord
 - 3rd level split the x coord

Assume that all points
don't have same x-
coord or y-coord!!!!



2D kd-Tree



2D kd-Tree

- For vertical split line
 - Point on the line is stored in the left subtree
- For horizontal split line
 - Point on the line is stored in the bottom subtree

Building the kd-Tree

Algorithm BUILDKDTREE($P, depth$)

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

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

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** $depth$ is even
4. **then** Split P into two subsets with a vertical line ℓ through the median x -coordinate of the points in P . Let P_1 be the set of points to the left of ℓ or on ℓ , and let P_2 be the set of points to the right of ℓ .
5. **else** Split P into two subsets with a horizontal line ℓ through the median y -coordinate of the points in P . Let P_1 be the set of points below ℓ or on ℓ , and let P_2 be the set of points above ℓ .
6. $v_{\text{left}} \leftarrow \text{BUILDKDTREE}(P_1, depth + 1)$
7. $v_{\text{right}} \leftarrow \text{BUILDKDTREE}(P_2, depth + 1)$
8. Create a node v storing ℓ , make v_{left} the left child of v , and make v_{right} the right child of v .
9. **return** v

Choosing the Median

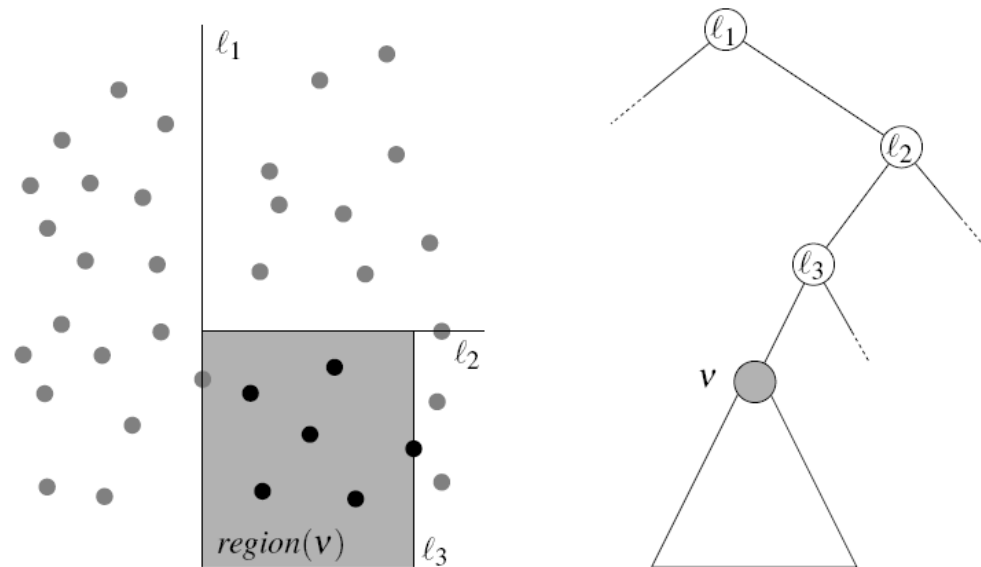
- Since the left subtree also contains the points lying on the split line
- The median should be the $\text{floor}(n/2)^{\text{th}}$ smallest member
 - Try the case of 3 points

Analysis

- How to find the median?
 - Can we find the median in $O(N)$?
 - Yes, but quite tricky
- It is better to take P as two sorted list
 - First list sorted by x
 - The second list sorted by y
- Resulting in $O(n \log n)$ build time

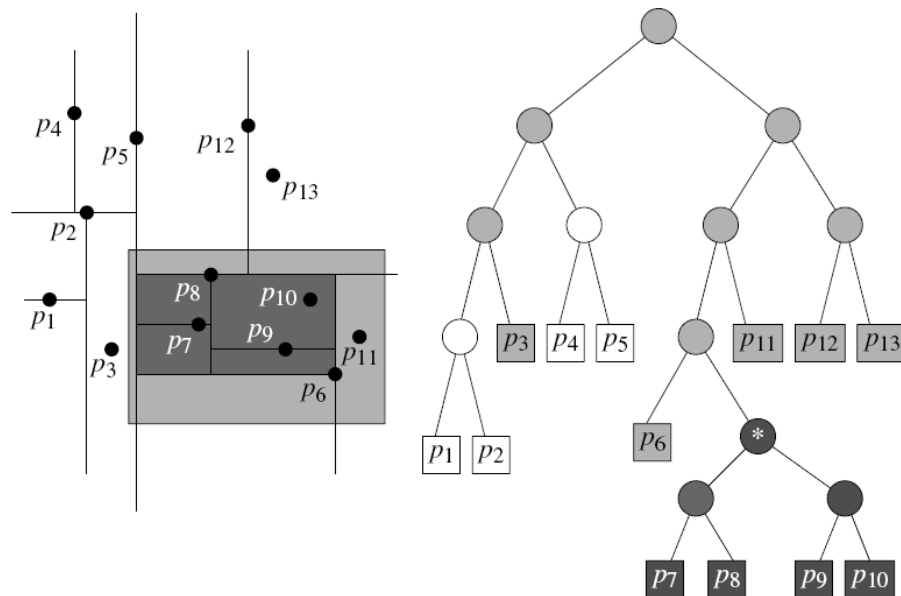
Searching in kd-Tree

- Each internal vertex represent a rectangular region
 - Might be unbounded



Searching in kd-Tree

- Traverse the tree
 - Visit only the node that intersect with the query
 - At leaf, check whether the point is contained in the query



The Algorithm

Algorithm SEARCHKDTREE(v, R)

Input. The root of (a subtree of) a kd-tree, and a range R .

Output. All points at leaves below v that lie in the range.

1. **if** v is a leaf
2. **then** Report the point stored at v if it lies in R .
3. **else if** $region(lc(v))$ is fully contained in R
4. **then** REPORTSUBTREE($lc(v)$)
5. **else if** $region(lc(v))$ intersects R
6. **then** SEARCHKDTREE($lc(v), R$)
7. **if** $region(rc(v))$ is fully contained in R
8. **then** REPORTSUBTREE($rc(v)$)
9. **else if** $region(rc(v))$ intersects R
10. **then** SEARCHKDTREE($rc(v), R$)

Region of an internal vertex

- We can do brute force
 - Pre-process
 - Not quite necessary
- We can compute on the go
 - $\text{region}(\text{lc}(v)) = \text{region}(v) \cap l(v)^{\text{left}}$
 - Where $l(v)$ is the splitting line of v
 - $l(v)^{\text{left}}$ is the left halfplane including $l(v)$

Analysis

- Searching takes $O(n^{0.5} + K)$
 - How?
 - Count the number of internal vertex intersected by the region
 - We actually count the number of node intersected by a vertical line (an upper bound)
 - Let $Q(n)$ be the number of intersected region
 - $Q(n)$ solves to $O(n^{0.5})$

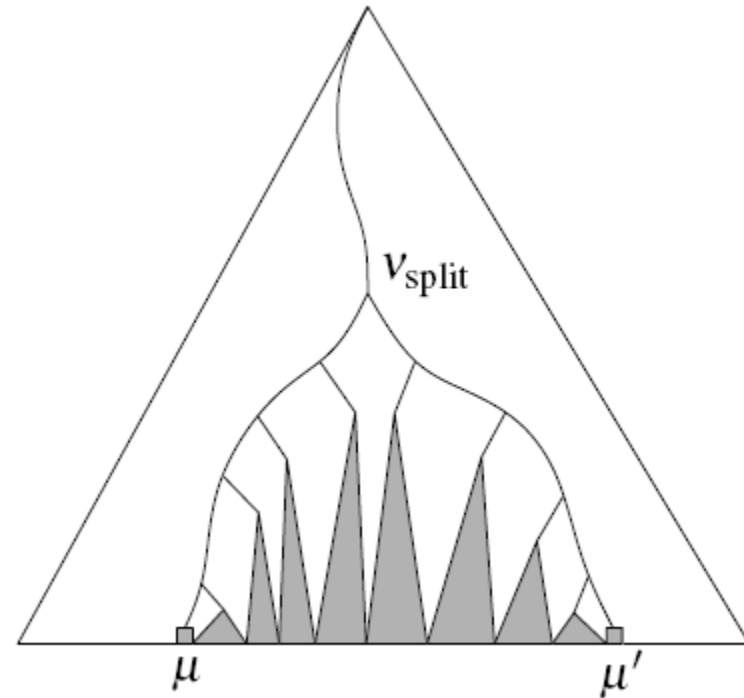
$$Q(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1. \end{cases}$$

Range Tree

Can we do better than kd-Tree?

Range Tree

- Having $O(\log^2 n + K)$ search time
- The idea
 - Let the P be the set of points
 - First, search P for points satisfying the x coordinate constrain
 - We won't bother with y coord yet
 - Let the result be P'
 - Then, search P' for points satisfying the y coordinate

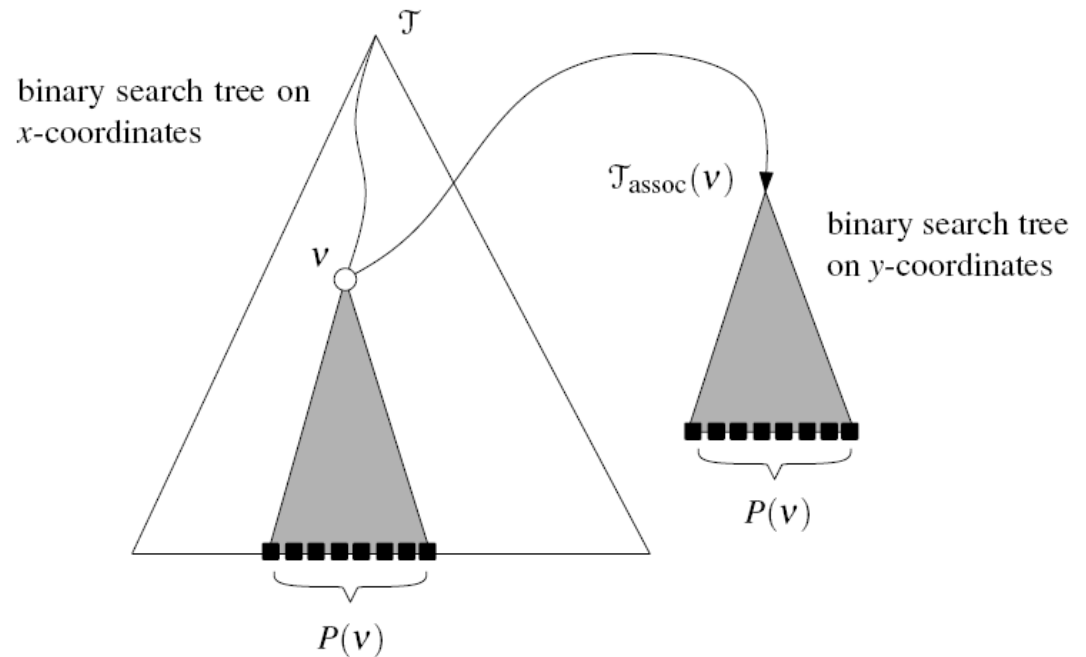


The canonical subset

- A set of leaves of a specific subtree

Range Tree

- For each vertex v
 - A canonical subset of a subtree rooted at v is stored in “another” range tree
 - Called “associated tree”



Building the Tree

Algorithm BUILD2DRANGETREE(P)

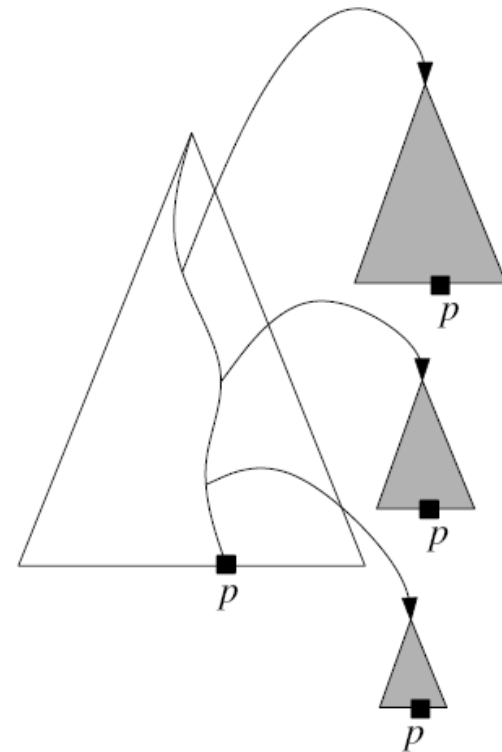
Input. A set P of points in the plane.

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 v storing this point, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v .
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. $v_{\text{left}} \leftarrow \text{BUILD2DRANGETREE}(P_{\text{left}})$
6. $v_{\text{right}} \leftarrow \text{BUILD2DRANGETREE}(P_{\text{right}})$
7. Create a node v storing x_{mid} , make v_{left} the left child of v , make v_{right} the right child of v , and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v .
8. **return** v

Analysis

- The space requirement of the Range tree is
 - $O(n \log n)$



Searching the Range Tree

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.

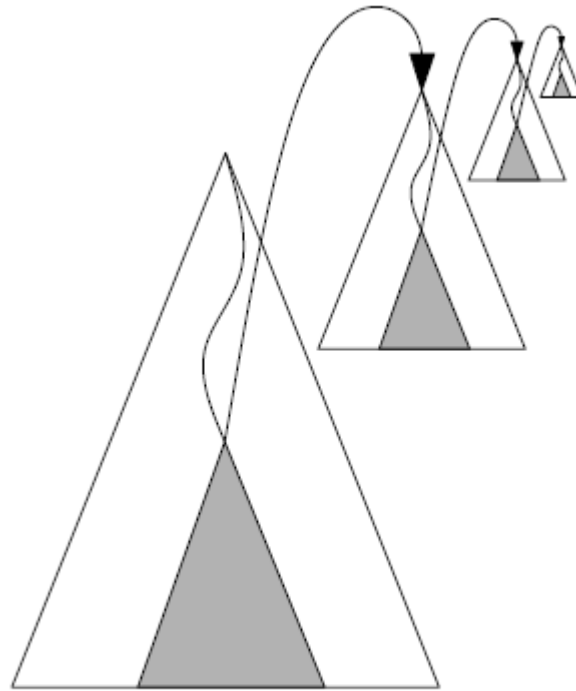
1. $v_{\text{split}} \leftarrow \text{FINDSPLITNODE}(\mathcal{T}, x, x')$
2. **if** v_{split} is a leaf
3. **then** Check if the point stored at v_{split} must be reported.
4. **else** (* Follow the path to x and call 1DRANGEQUERY on the subtrees right of the path. *)
5. $v \leftarrow lc(v_{\text{split}})$
6. **while** v is not a leaf
7. **do if** $x \leq x_v$
8. **then** 1DRANGEQUERY($\mathcal{T}_{\text{assoc}}(rc(v)), [y : y']$)
9. $v \leftarrow lc(v)$
10. **else** $v \leftarrow rc(v)$
11. Check if the point stored at v must be reported.
12. Similarly, follow the path from $rc(v_{\text{split}})$ 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.

Analysis

- For each node in the main tree
 - We spend $O(\log n + K)$ to search the T_{assoc}
 - There are $2 * \log n$ such node
 - One for the left subtree of the split node
 - One for the right subtree of the split node
 - Hence, it is $O(\log^2 n + K)$

Generalizes to Higher Dimention

- Simple



Fixing the unique x,y assumption

- Define new ordering scheme
- Composite Number Space

$$(a|b) < (a'|b') \Leftrightarrow a < a' \text{ or } (a = a' \text{ and } b < b').$$

- Change from $p = (x,y)$ to
 - $p = ((x|y) , (y|x))$
- Change the query as well
 - $[x:x'] * [y:y']$ into
 - $[(x|-\infty):(x'|+\infty)] * [(y|-\infty):(y'|+\infty)]$