CMPT 813
Range Search

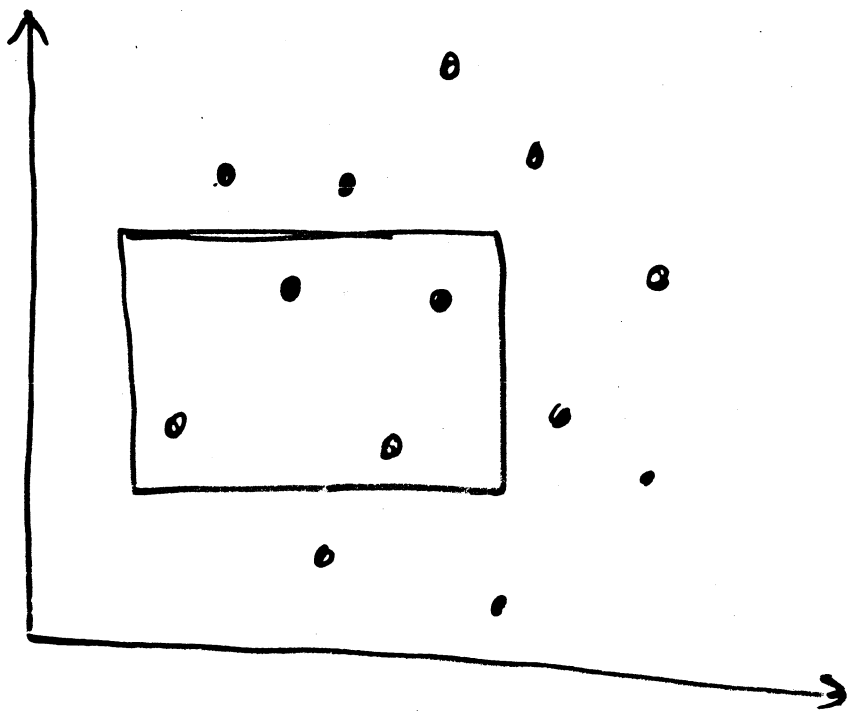
Week 4

Chapter 5

de Berg, van Kreveld, Overmars + Schwarzkopf

Orthogonal Range Searching
(querying a database)

- Data structure for a set of objects (points, rectangles, polygons) for efficient range queries.
  - Given a set of n points, build a data structure that for any query rectangle R, reports all points in R.



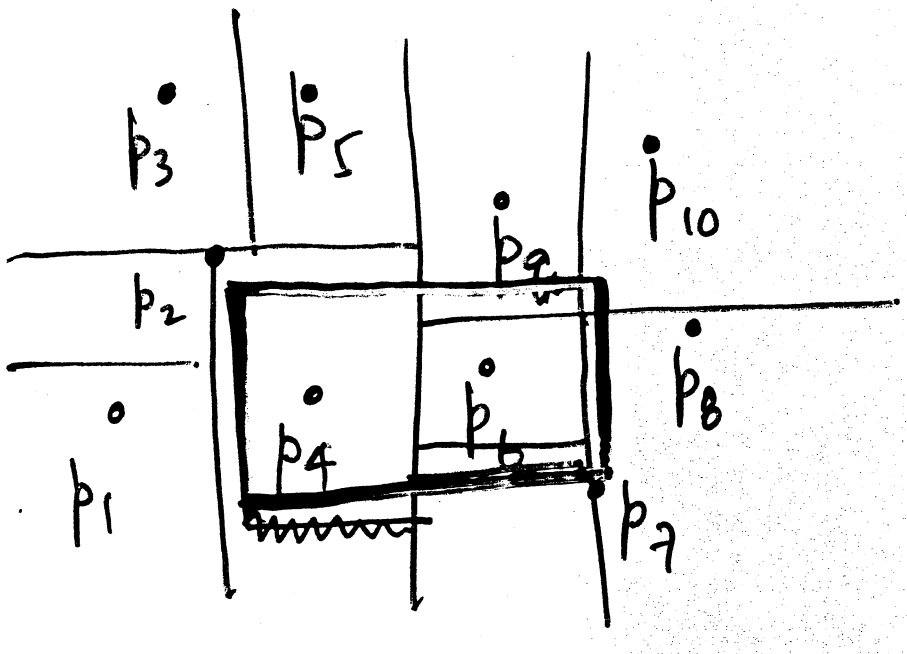
- Efficiency : depends on type of objects & queries.

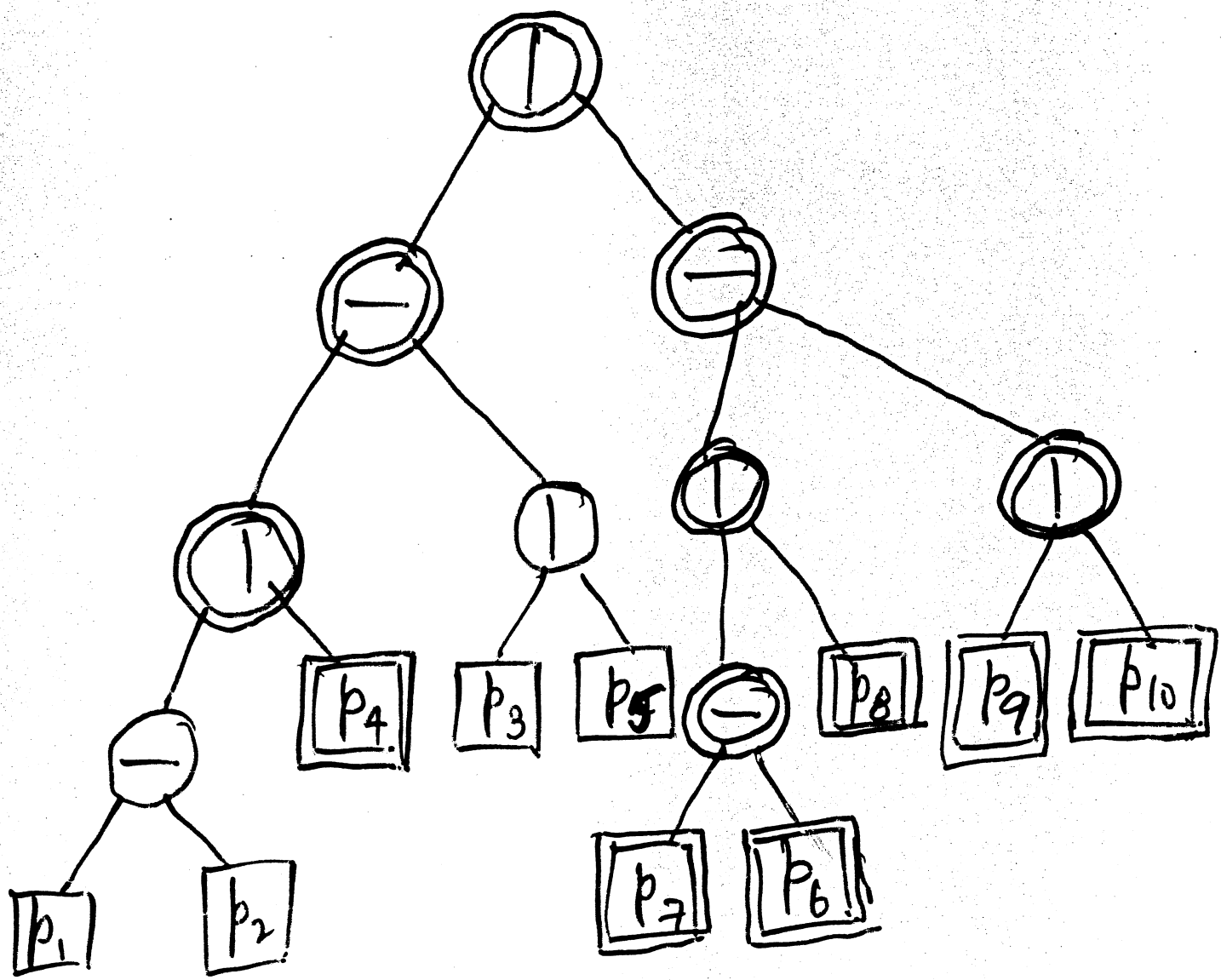
- Time-space tradeoff : the more we preprocess the data & store, the faster we can solve a query

- Orthogonal range searching deals with point sets and axis-aligned rectangle queries.

kd-trees (Bentley)

- Not the most efficient solution in theory.
- Everyone uses it in practice.
- Algorithm:
  - Choose x or y coordinate (alternate)
  - Choose the median of the coordinate.
    (the point set gets split into two equal halves.)
    This defines a horizontal or a vertical line.
  - Recurse on both sides.


p3
p5
p10
p9
p2
p8
p4
p6
p1
p7


p4
p3
p5
p8
p9
p10
p1
p2
p7
p6

Kd-tree : Range Queries

1. Recursive procedure, starting from v = root.

2. Search (v, R):

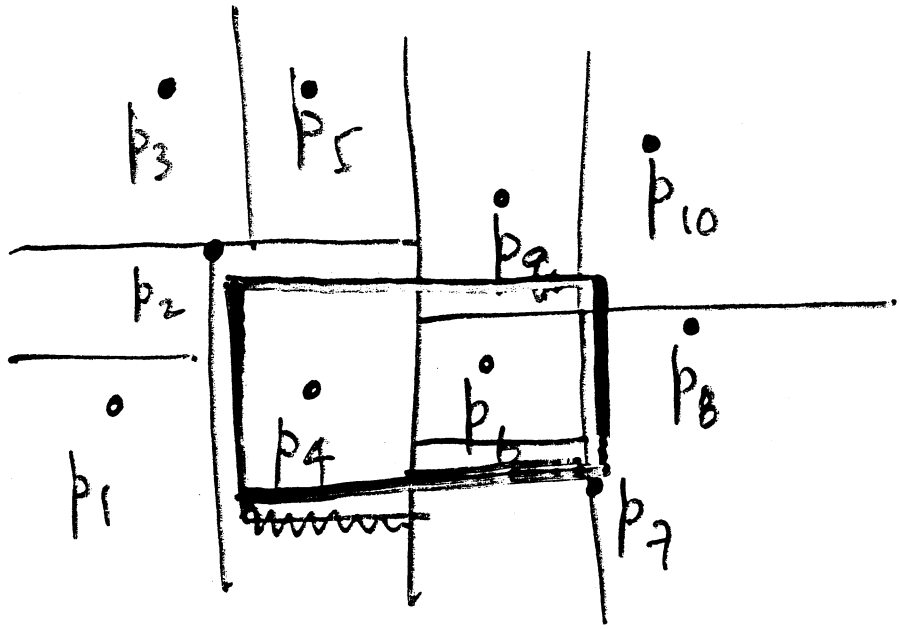
   a) if v is a leaf, report the pt. stored in v if it lies in R.

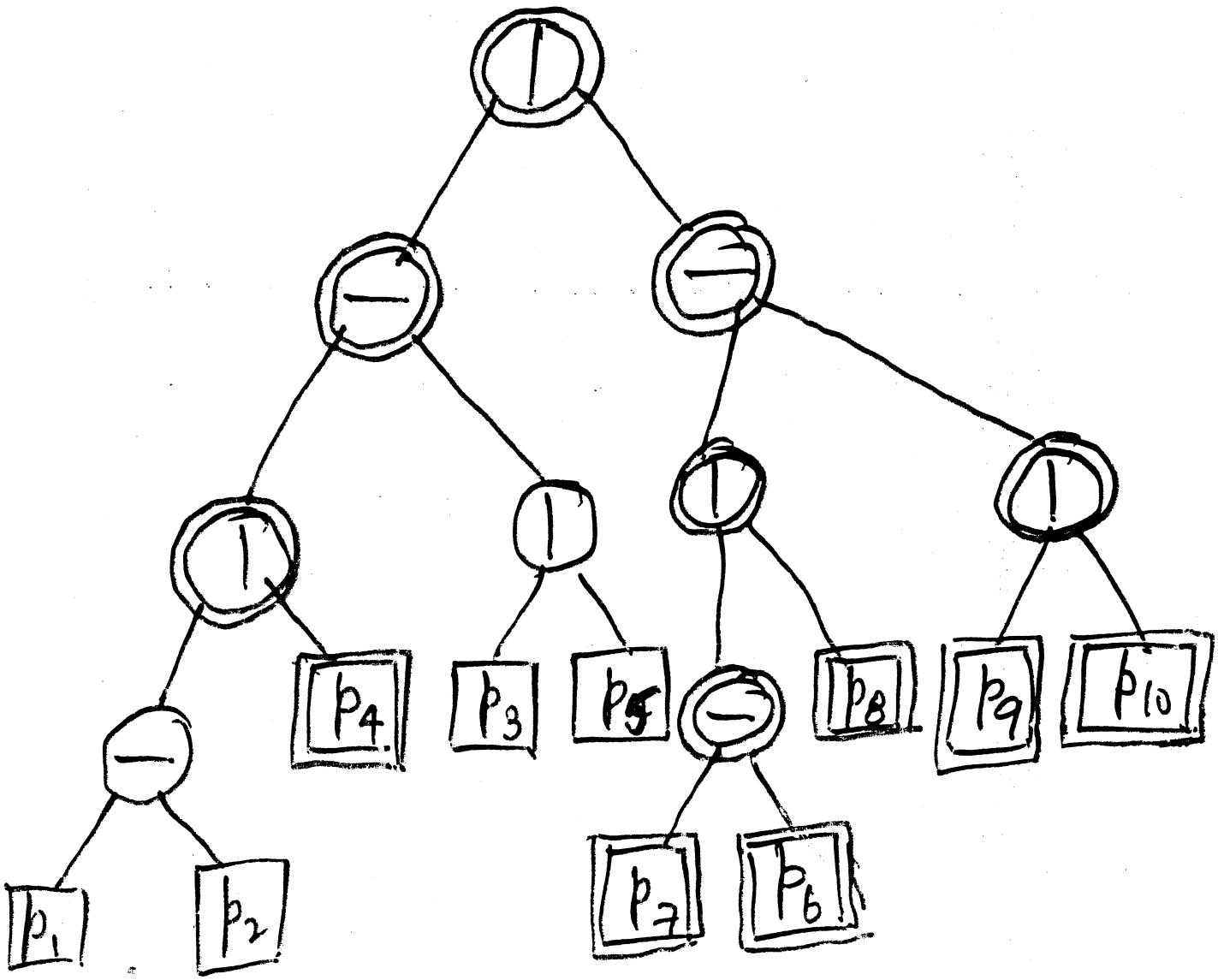
   b) Otherwise, if Regn(v) is contained in R, report all the points in the subtree of v.

   c) Otherwise:

      i) if Regn(left(v)) intersects R, search(left(v), R)

      ii) if Region(right(v)) intersects R, Search(right(v), R)


p3
p5
p10
p9
p2
p8
p4
p6
p1
p7


p4
p3
p5
p8
p9
p10
p1
p2
p7
p6

- When Region(v) is contained in R, the complexity is linear in output size.
- We just need to bound the number of nodes v st. Region(v) intersects R but is not contained in R. In other words, the boundary of R intersects the boundary of Region(v).
- We will make a gross overestimate. We will bound the number of Region(v) which are crossed by any one of the 4 horizontal/vertical lines of R.

- Let ℓ be the line defining one side of R.
- How many Region(v) can ℓ intersect?
- Consider two levels at a time
- Suppose the first cut is vertical, & second horizontal. We have 4 regions, each with $\frac{n}{4}$ points
- A line intersect exactly two region. (line is either vertical or horizontal) The other regions will be either outside or entirely inside R.
- The recurrence is

$$Q(n) = \begin{cases} 1 & \text{if } n=1 \\ 2Q\left(\frac{n}{4}\right)+2 & \text{Otherwise} \end{cases}$$

$\therefore Q(n) \in O(\sqrt{n})$.

- Kd-tree is an $O(n)$ space data structure that solves the orthogonal range query in worst-case time $O(\sqrt{n}+k)$, where k is the output size.

Extending Kd-tree to higher dimensions?

- Try 3-dimensional case, & then generalize.
- The recurrence is

$$Q(n) = 2^{d-1} Q\left(\frac{n}{2^d}\right) + 2^{d-1}$$

which solves to $Q(n) \in O(n^{1-\frac{1}{d}})$

(d is fixed)

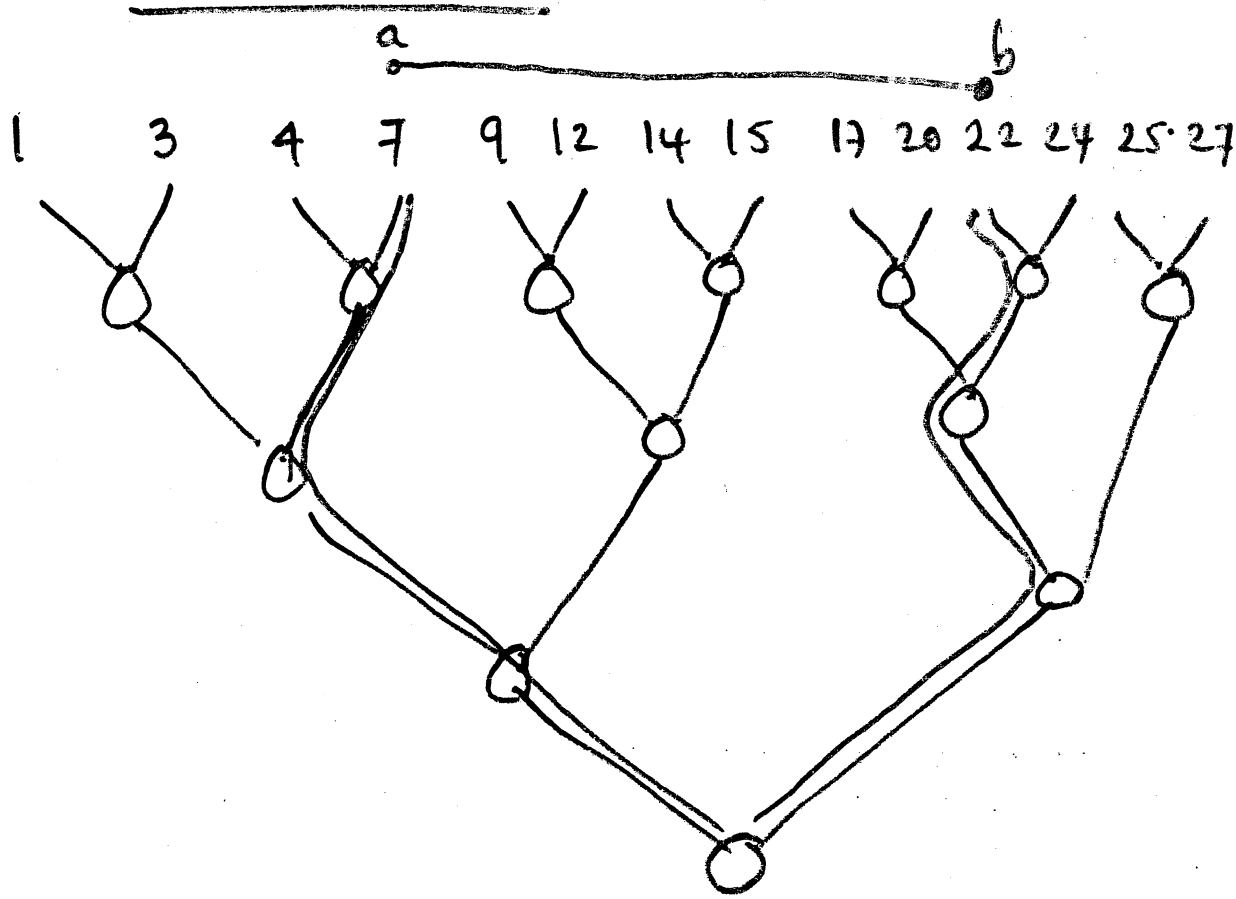
- Kd-tree is an $O(dn)$ space data structure that solves d-dim range query in worst-case time $O(n^{1-\frac{1}{d}} + k)$, where k is the output size.

1-dimensional range search.

- Points on a line
- Queries are intervals.
- Preprocessing : Sort the points
- Each query can be answered in $O(\log n + k)$ time where the interval contains k points.

  (Does not extend to higher dimension.)

Tree Search.



- build a balanced binary tree on the sorted list of points.
- given an interval [a,b], search down the tree for a & b.
- all leaves between the two form the answer.

Analysis

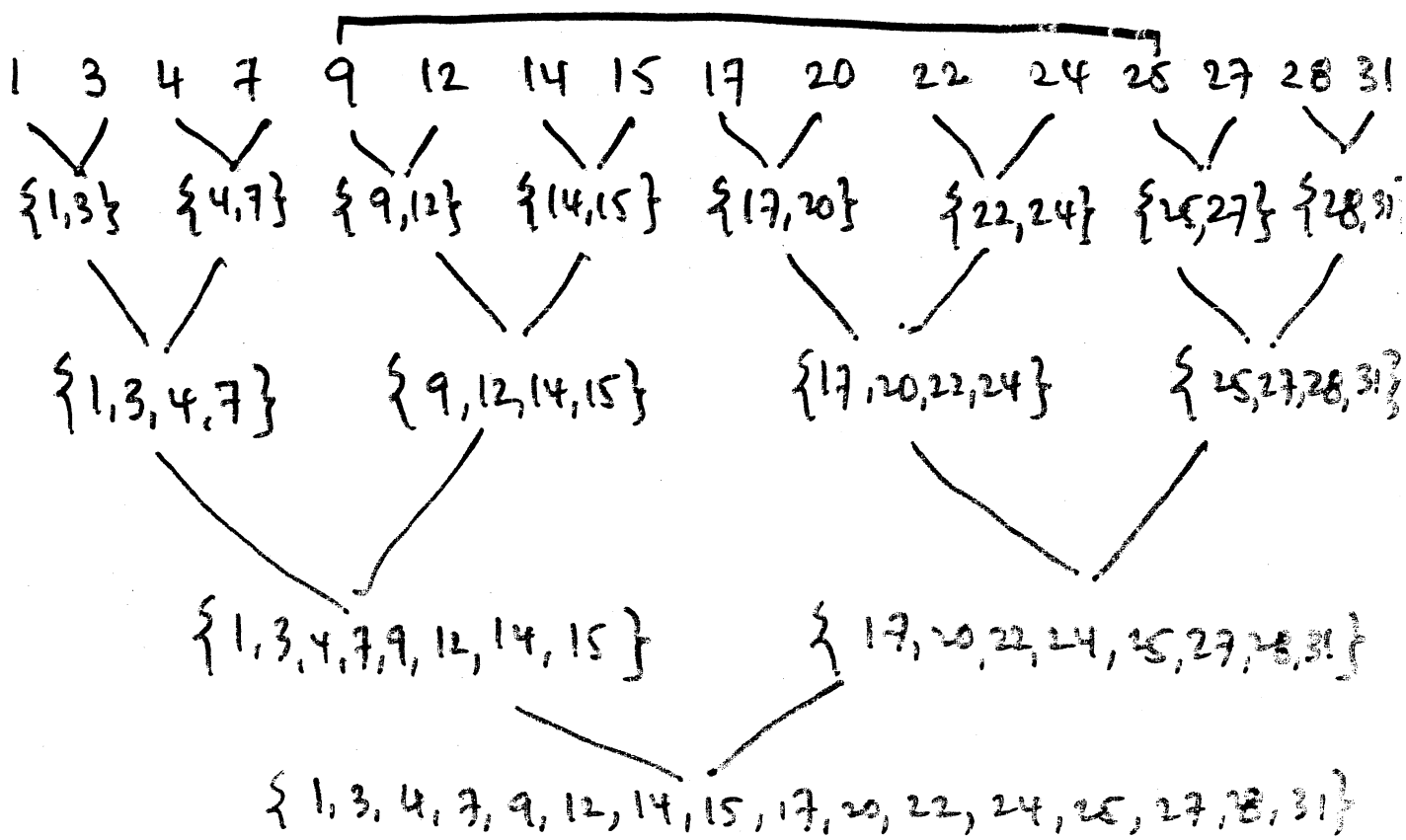
- Search time : $O(\log n + k)$
- Preprocessing time : $O(n \log n)$
- Storage space : $O(n \log n)$
  (canonical sets)

1-d Range query

- Given query $[a, b]$, search down the tree for the leftmost leaf $u \geq a$ and the leftmost leaf $v \geq b$.
- All leaves between u & v are in the range.
  - If $u = a$ or $v = b$, include the canonical set stored at u & v.
  - The remainder nodes are the union of at most $2 \log n$ canonical sets.

Canonical Subsets

- $S_1, S_2, \ldots, S_t$ are canonical subsets $S_i \subseteq P$, if the answer to any range query can be written as the disjoint union of some S_i's.
- The canonical subsets may overlap.
- Only $O(\log n)$ subsets are reported for any range query.

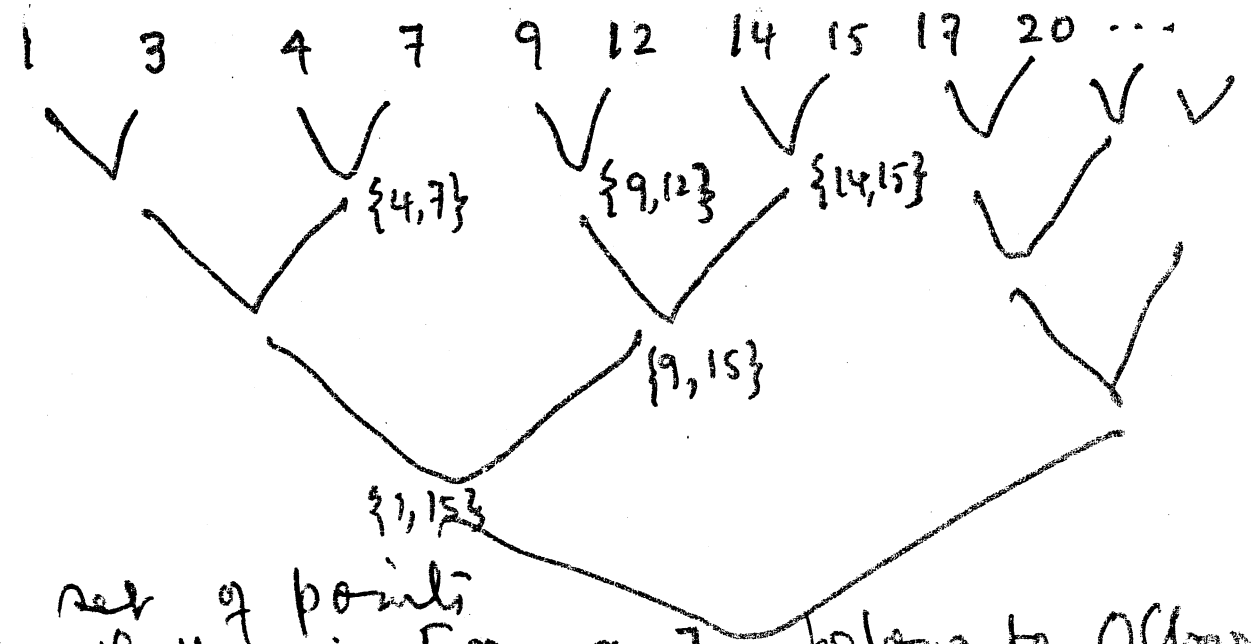


Orthogonal Range Trees

- Generalizes 1-D search trees to dimension d.
- Each search recursively decomposes into multiple lower dimensional searches.
- Search complexity is $O((\log n)^d + k)$ where k is the answer size.
- Space and time complexity is $O(n(\log n)^{d-1})$.
- Fractional Cascading eliminates one $\log n$ factor from search time.
- We focus on 2-d. The generalization step is straightforward.

2-dimensional Range Trees

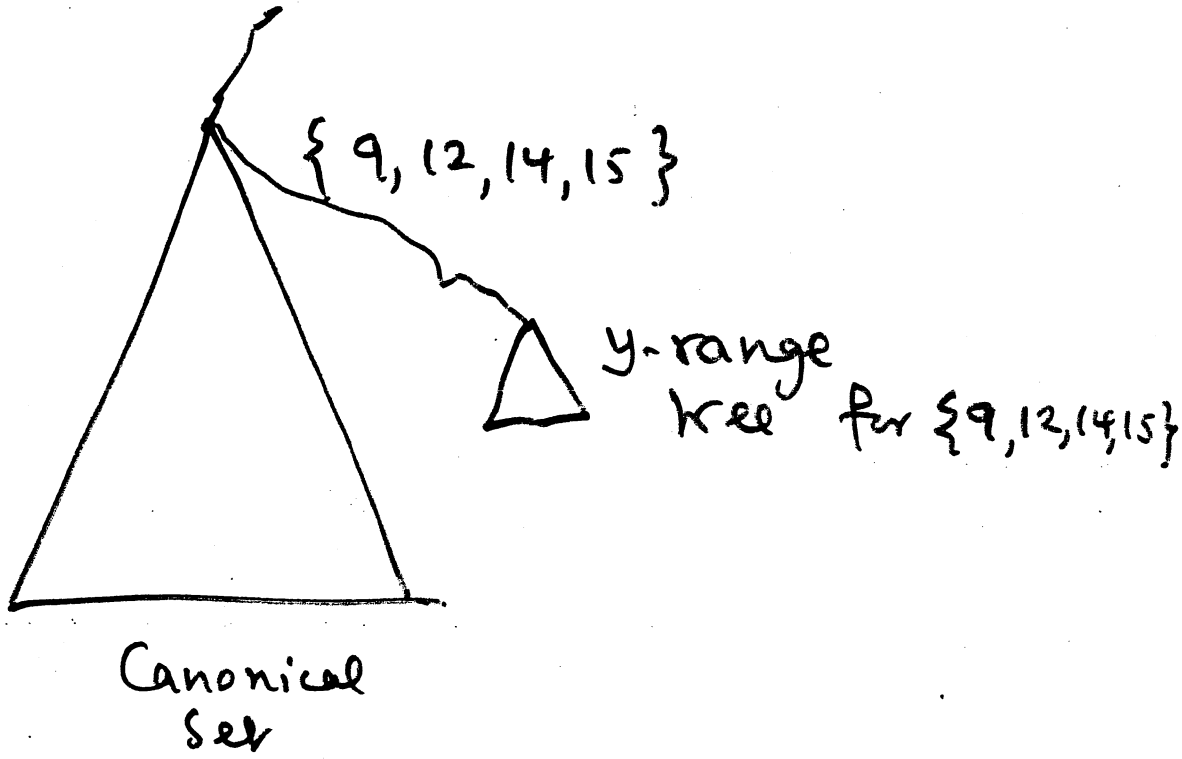
- Suppose $S = \{p_1, \dots, p_n\}$ set of points in the plane.
- Eg. The generic query is
$$R = [x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$$
- Ignore the y-coordinates first. Build a 1-dimensional x-range tree on S.



The set of points that fall in $[x_{lo}, x_{hi}]$ belong to $O(\log n)$ canonical sets. This is a superset of the final answer.

Level 2 trees

- Collect points of each canonical set & build a y-range tree.



- Search each of the $O(\log n)$ canonical sets that include points for x-range $[x_{lo}, x_{hi}]$ using their y-range trees for range $[y_{lo}, y_{hi}]$

- The y-range searches list out the points of P in the query rectangle

  (No duplicates)

Analysis

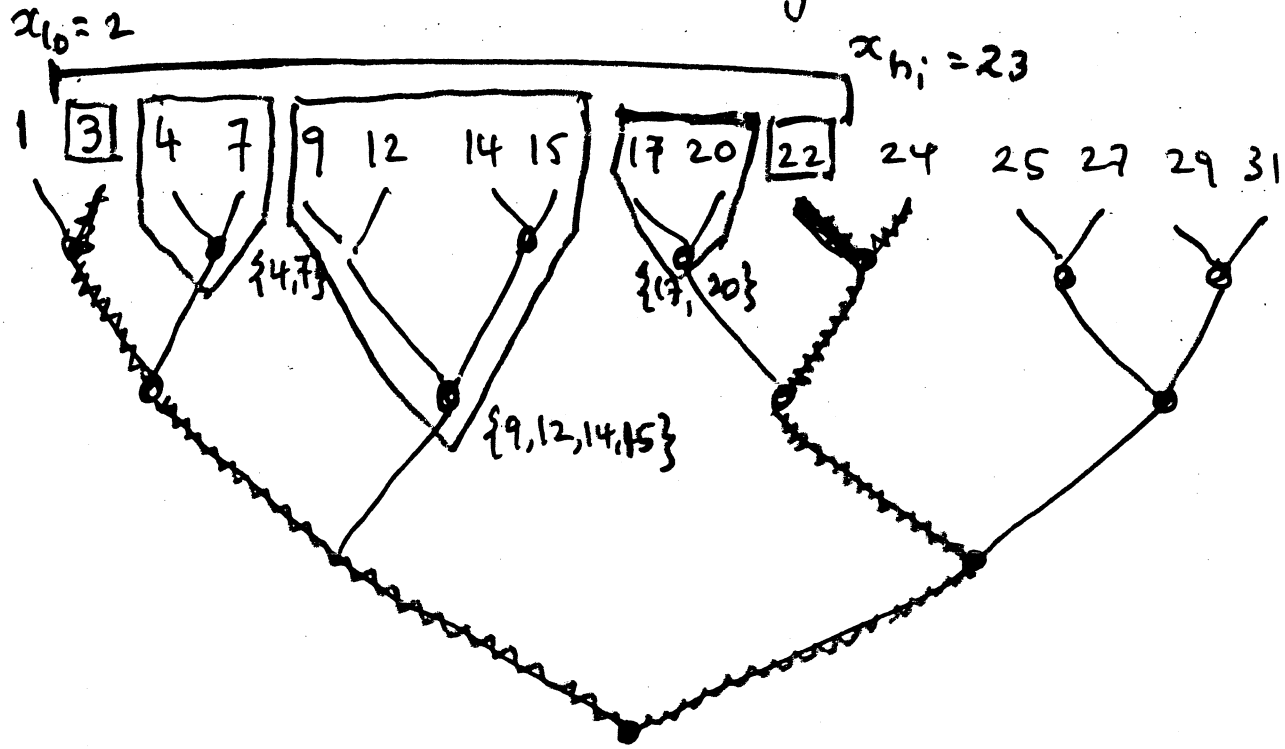
- Query time: $O(\log^2 n)$
  - $O(\log n)$ x-range query
  - Each canonical set y-range query takes $O(\log n)$ time.

- Space complexity: $O(n \log n)$
  - each point appears in $O(\log n)$ canonical sets.

- Preprocessing Complexity: $O(n \log n)$

2-dimensional Range Trees

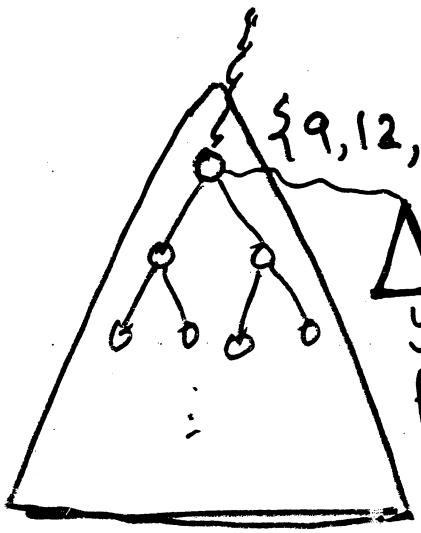
- Suppose $S = \{p_1, p_2, \cdots, p_n\}$ set of points in the plane.
- The generic query is $R = [x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$
- Ignore the y-coordinates first. Build a 1-dimensional x-range tree on P.



- The set of points that fall in $[x_{lo}, x_{hi}]$ belong to $O(\log n)$ canonical sets.
- This is a superset of the final answer.

Level 2 trees

- Collect points of each canonical set, & build a y-range tree

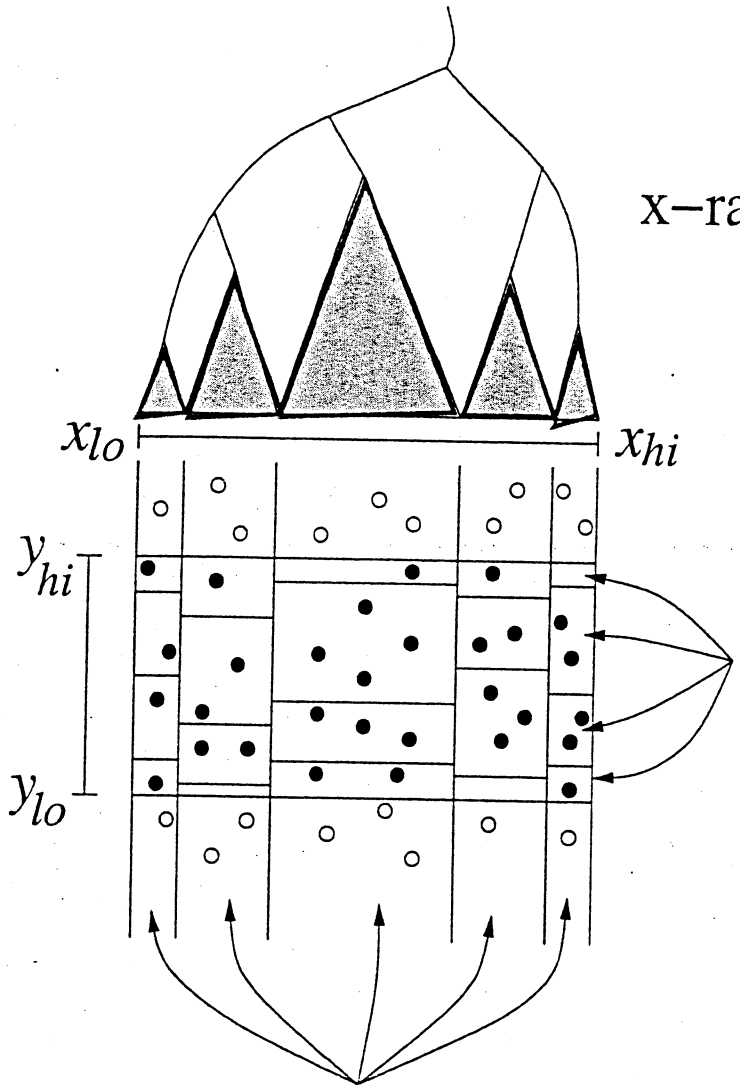


Canonical Set.

- We search each of the $O(\log n)$ canonical sets that include points for x-range $[x_{lo}, x_{hi}]$ using their y-range trees for range $[y_{lo}, y_{hi}]$.
- The y-range searches list out the points of P in the query rectangle.

(No duplicates.

Canonical Sets



Analysis

- Query time is $O(\log^2 n)$
  - $O(\log n)$ x-range query
  - Each canonical set y-range query takes $O(\log n)$ time.

- Space complexity is $O(n \log n)$.
  - Total canonical set size is $O(n \log n)$
    { Each point of S appears in $O(\log n)$ canonical sets.)
  - Each canonical set of size m requires $O(m)$ space for the y-range tree.

Preprocessing time complexity. $O(n \log n)$.

- x-range tree can be built in $O(n \log n)$ time.
  (Building them bottom up)

- y-range trees of all canonical sets can be constructed in $O(n \log n)$ time.

d-dimensional range trees

- The multilevel range tree extends naturally to any dimension d.
- Build the x-range tree on the first coordinate.
- For each canonical set build the (d-1)-dimensional range tree on the remaining (d-1)-dimensions; $x_2, x_3, \cdots, x_d$.

$$T_d(n) = O(n \log n) + O(\log n) \cdot T_{d-1}(n)$$

- Complexity grows by one log n factor for each dimension.
- Search cost is $O((\log n)^d)$ (query cost)
- Space Complexity : $O(n(\log n)^{d-1})$

  Preprocessing Time : $O(n(\log n)^{d-1})$.

Fractional Cascading

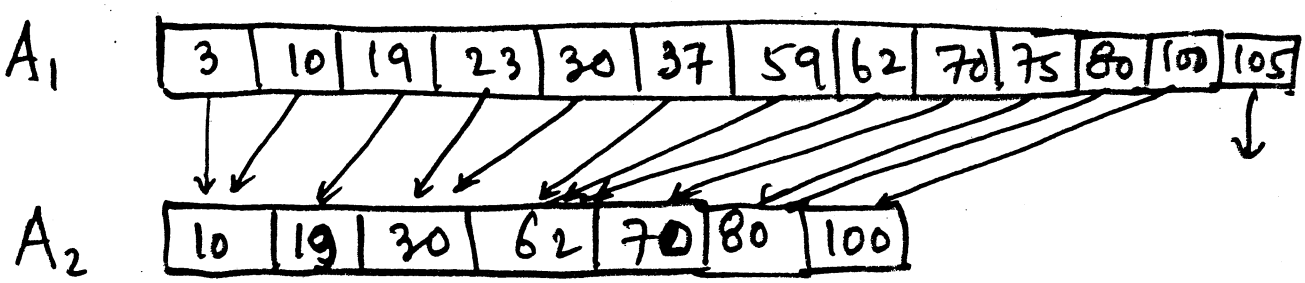
- Allows the 2-dimensional problem to be solved in
  - a) $O(\log n + k)$ query time
  - b) $O(n \log n)$ storage space & preprocessing time.
- This allows the d-dimensional orthogonal range tree problem to be solved in
  - a) $O(\log^{d-1} n + k)$ query time
  - b) $O(n(\log n)^{d-1})$ storage space & preprocessing time.

Basic idea.

- 2-range tree first finds the set of canonical sets lying in $[x_{lo}, x_{hi}]$.
- Each canonical set is searched using the ~~2-tree~~ y-range tree for the same range $[y_{lo}, y_{hi}]$.
- Since each set is searched for the same key, we can improve the search cost to $O(1)$, amortized time per set.
- Key is to attach pointers ~~list~~ linking the search structures for the canonical sets.

Simple Example

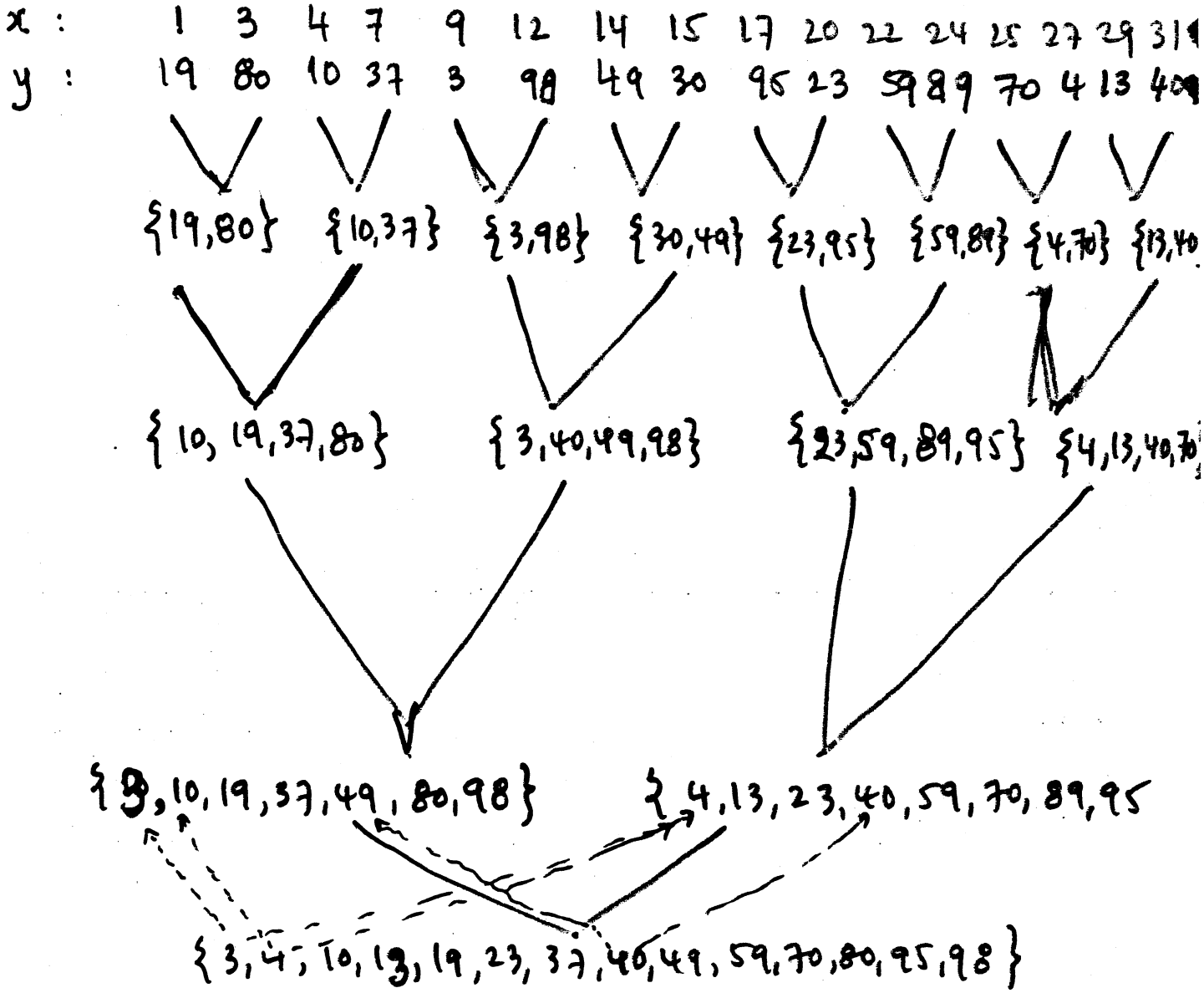
- We have two sets of numbers A_1, A_2, $A_1 \supseteq A_2$; both ~~set~~ sorted.
- Given a range $[x, x']$, we want to report all keys in A_1, A_2 that lie in the range $[x, x']$.
- 🧠 $2 \log n + k$ solution is easy.



- Point $A_1[i]$ to $A_2[j]$ ~~for all i, where~~ ~~$A_1[i] \geq A_2[j]$~~ for all i where $A[j]$ is the smallest key $\geq A_1[i]$.

- Suppose we want keys in range $[y, y']$.
- Search A_1 for y and walk until past y'. $O(\log n + k_1)$
- A_1 search for y ended at $A_1[i]$. Use the pointer at $A_1[i]$ to start search in A_2. This takes $O(1 + k_2)$ time.

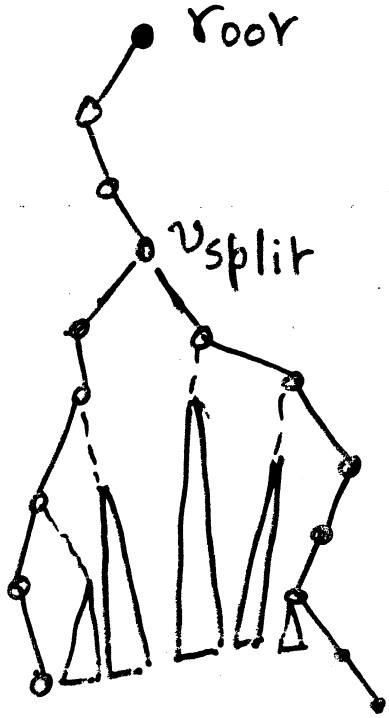
Fractional Cascading in Range Trees



- Canonical subsets $S(l(v))$ & $S(r(v))$ are subsets of canonical set $S(v)$.
- Each entry in $A(v)$ stores two pointers into arrays $A(l(v))$ & $A(r(v))$. $A(v)[i]$ points to the left most entry of $A(l(v))[j]$ with smallest value $\geq A(v)[i]$. Same for $A(r(v))$.

Fractional Cascading Search.

- Consider range $R = [x, x'] \times [y, y']$.
- Search x, x' in the main x-range tree.
- Let v_{split} be the node where the two search paths diverge.



- At v_{split}, do binary search to locate y in $A(v_{split})$.
- We can then search (for y in each) canonical set attached to the paths in $O(1)$ time per canonical set.
- Report all points in rectangle query in $O(\log n + k)$ time.

∴ d-dimensional orthogonal range search problem can be solved in

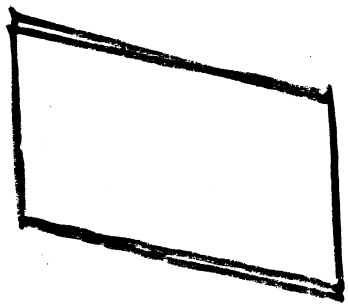
a) $O((\log n)^{d-1} + k)$ query time

b) $O(n(\log n)^{d-1})$ storage space

c) $O(n(\log n)^{d-1})$ preprocessing time.

Other query object ?

1) Parallelogram of the type.

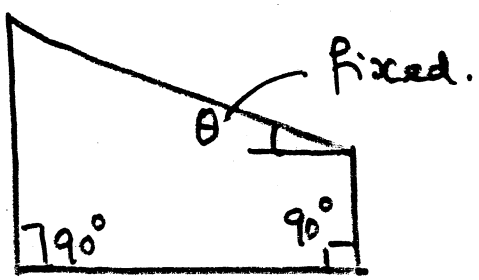


In 2-d

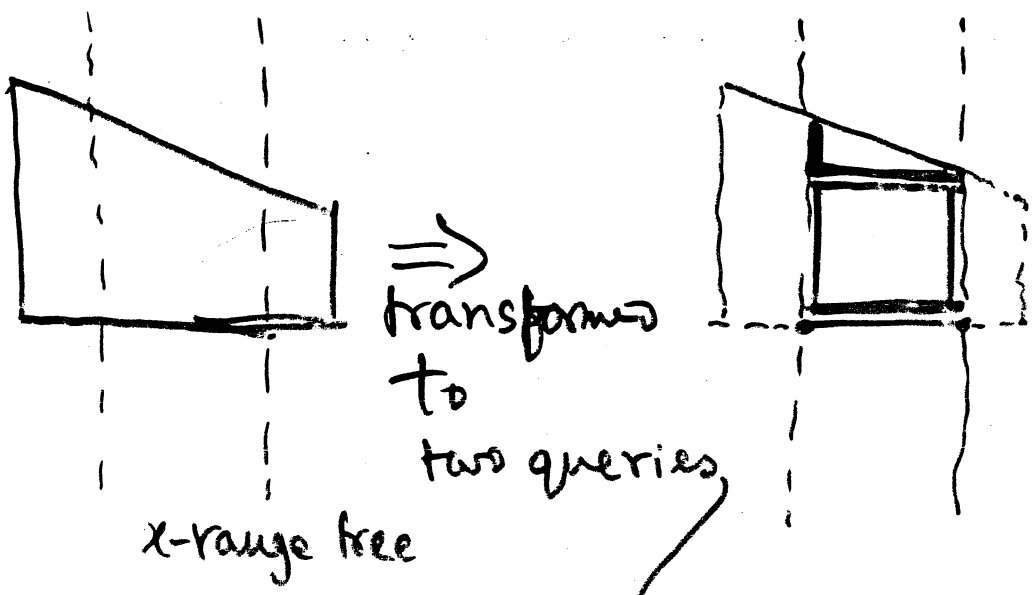
a) $O(\log n + k)$ query time

b) $O(n \log n)$ storage space + preprocessing time.

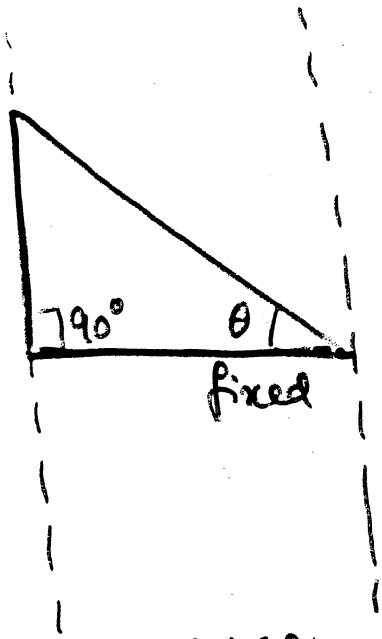
2. Trapezoid query object ?



Searching within a canonical set.



(2) Right-angled triangle.

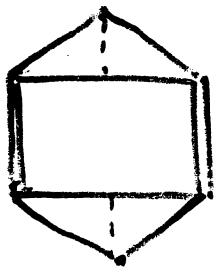


Using the priority-search tree data structure, it is possible to answer the query (i.e. reporting the points inside the triangle) in $O(\log n + k)$ time.

°° The trapetoid query can be answered in $O(\log^2 n + k)$ time using $O(n \log n)$ storage space + preprocessing time.

(Octagon)

(4) Recular Hexagon, query object



Can be answered by quering

a) a rectangle

b) 4 right angled triangle.
with one fixed acute angle.

Query time : $O(\log^2 n + k)$

Storage space : $O(n \log n)$

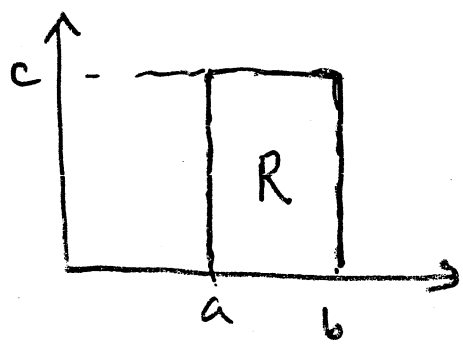
Preprocessing time : $O(n \log n)$

McCreight's Priority Search Trees

Priority Search Trees, SIAM J. on Computing Vol. 14, 257-276, 1985.

Problem Given a set of points (x,y), find all points inside a testing rectangle.

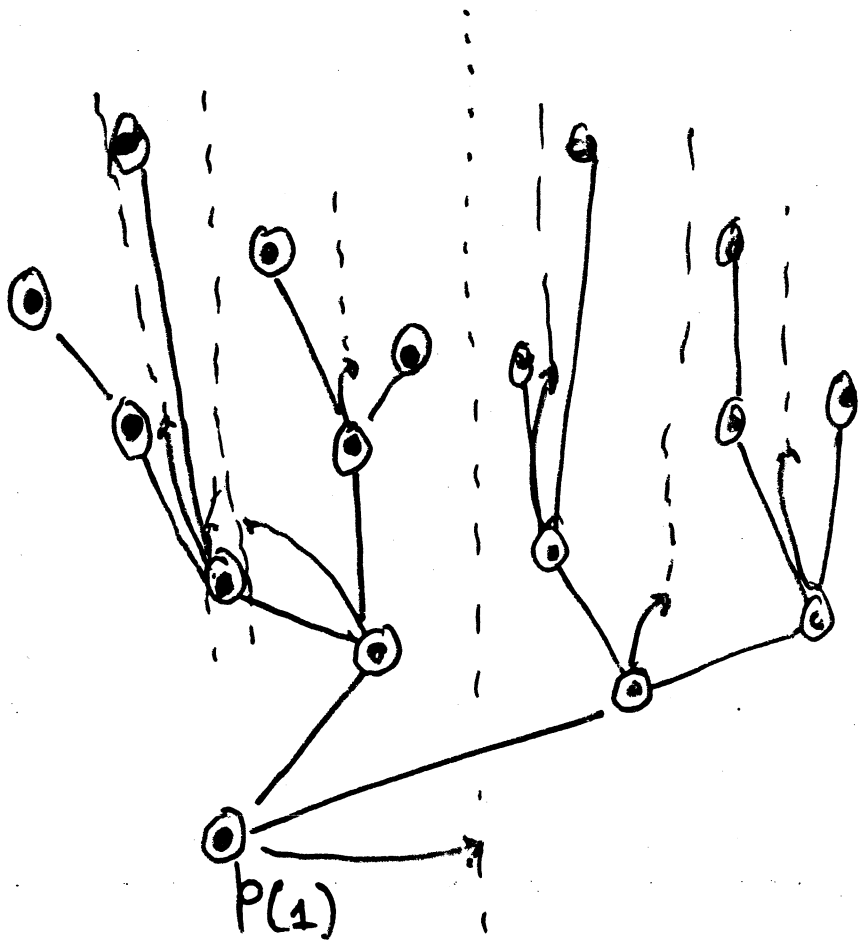
- allows insertion & deletions of points in $O(\log n)$ time.
- $O(\log n + k)$ query time for rectangle $R = \{x \mid a \le x \le b \text{ & } 0 \le y \le c\}$



- Search tree allows in $O(\log n)$ time the following operations (Given a rectangle query)
  - ~~minimum~~ point with minimum x-coordinate
  - largest x-coordinate point in R
  - ~~largest~~ smallest y-coordinate point in R
    (No largest y-coordinate point)

Preprocessing Stage ($S = \{p_1, \ldots, p_n\}$ is the set of points)

- Select the point with the smallest y-coordinate. (Say $p_{(1)}$). $p_{(1)}$ is the root of the search tree.
- The remaining points $S - \{p_{(1)}\}$ are assigned to the left or to the right subtree of $p_{(1)}$ according to whether they are to the left or to the right of a vertical line determined by the x-median of $S - \{p_{(1)}\}$ points. The root also contains the x-range of S, along with the vertical line.
- Recursively construct the left priority search tree of $p_{(1)}$
- Recursively construct the right priority search tree of $p_{(1)}$.



Height of the tree is $O(\log n)$ (Balanced)

Search for a point $p = (x_p, y_p)$

- Check if $y_p = y_{p(1)}$
- If yes, stop.
- Else if $x_p <$ divider (p_1),
  search left sub-tree
  else search right sub-tree.

Insert a point $p = (x_p, y_p)$

- If $y_p < y_{p(1)}$, (a) exchange p with $p_{(1)}$
  (b) Insert $p_{(1)}$ in $S - \{p_{(1)}\}$.
- If $y_p \geq y_{p(1)}$, insert p in $S - \{p_{(1)}\}$.

The search will terminate at a null link which gets replaced by the point being inserted.

Cost: $O(\log n)$ time.

Delete a point p

- Locate p in the tree.
- Delete p from the tree. (say at node v)
  (Leaves a hole in the structure)
- Fill it up by promoting the child of v with the smallest y-coordinate.
  (The hole moves down to the position of that child)
- Continue until the hole moves out of the bottom of the tree.

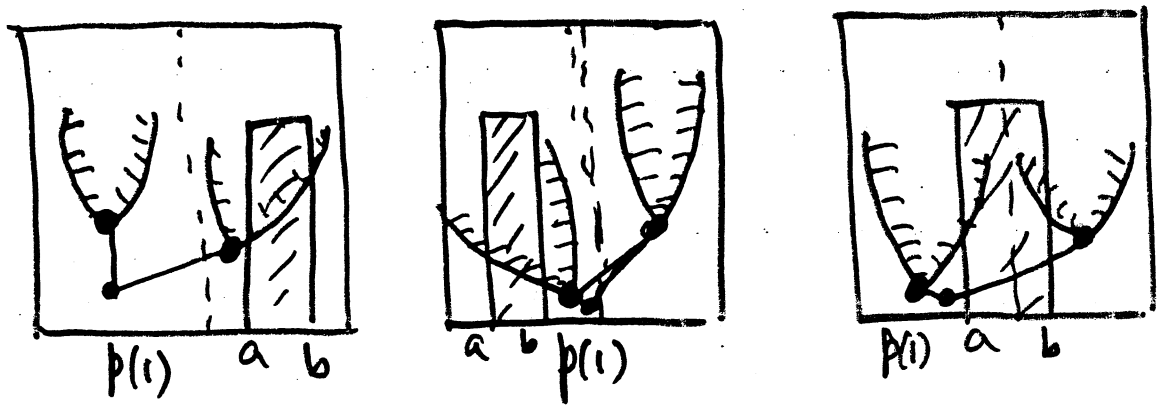
Cost $O(\log n)$

(Balancing the tree is required after insertion & deletion operations are performed.)

Balancing can be avoided if all the points are known beforehand.

Find the point of S with smallest y-coordinate inside the query rectangle $R = [a,b] \times [0,c]$.

- The root is in R: Stop, $p_{(1)}$ is the point
- Otherwise we have three possibilities.

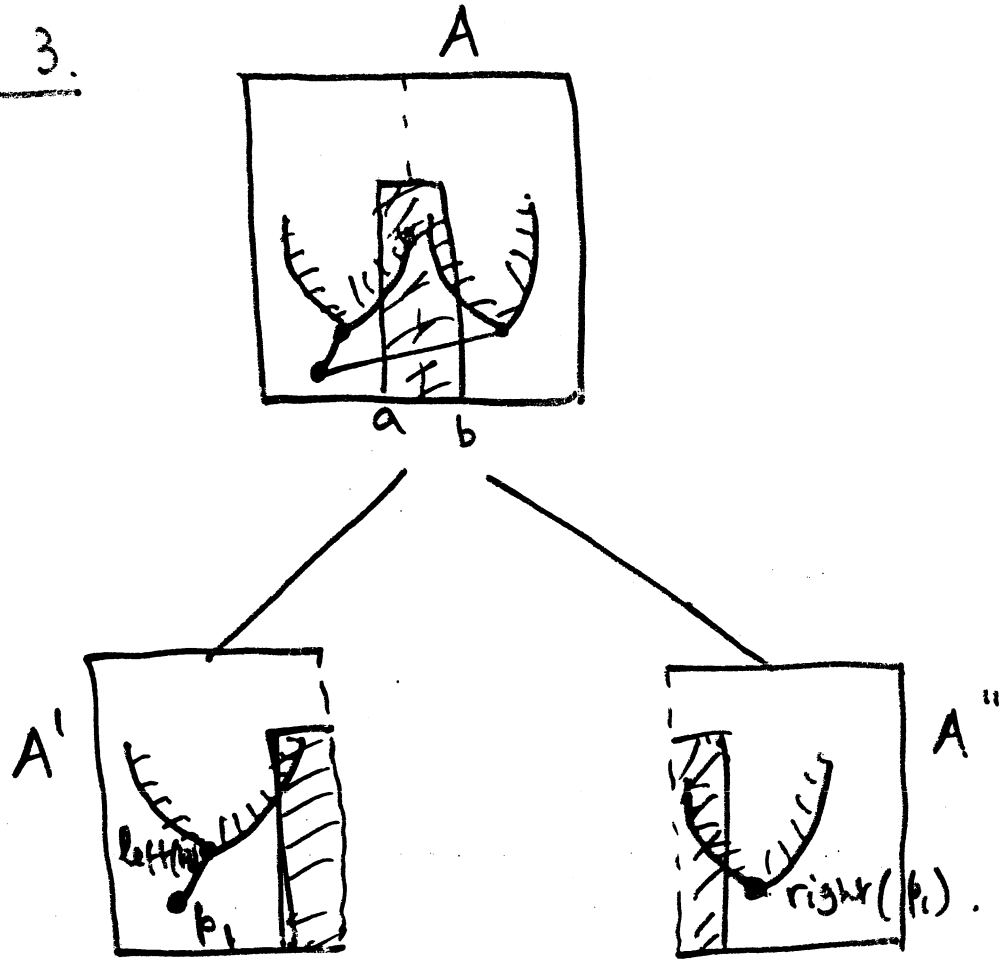


First two cases: ignore either the left or the right.

Third case: (a) Find the lowest point in each subtree.

(b) Pick the smallest of the two.

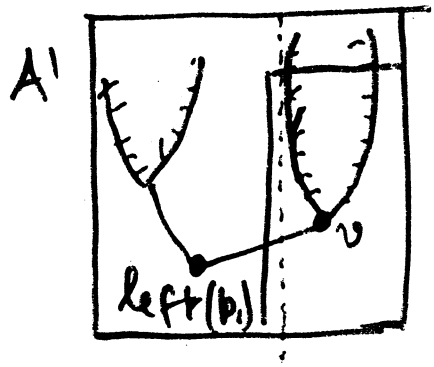
Case 3.



The problem reduces to two smaller similar subproblems.

In A' if the vertical line for the left subtree left($p_{(1)}$) does not intersect the shaded rectangle, the left subtree can be rejected. If the line intersects the rectangle, the shaded region contains the x-range of the right subtree of left(p_1).

Case 3 (contd.)



We then test v, the right child of left($p_{(1)}$) & see if it is inside R. If yes, search the left subtree. Otherwise we ignore the search tree rooted at v. Search the left subtree. In any case we are left with one subtree of left(p_1). Total time spent at left(v_1) is $O(1)$.

Similarly, ~~O(1)~~ after $O(1)$ time, we are left with one subtree of right(p_1).

°°° Lemma: The point with smallest y coordinate in a query rectangle R can be found in $O(\log n)$ time.