kd-Trees

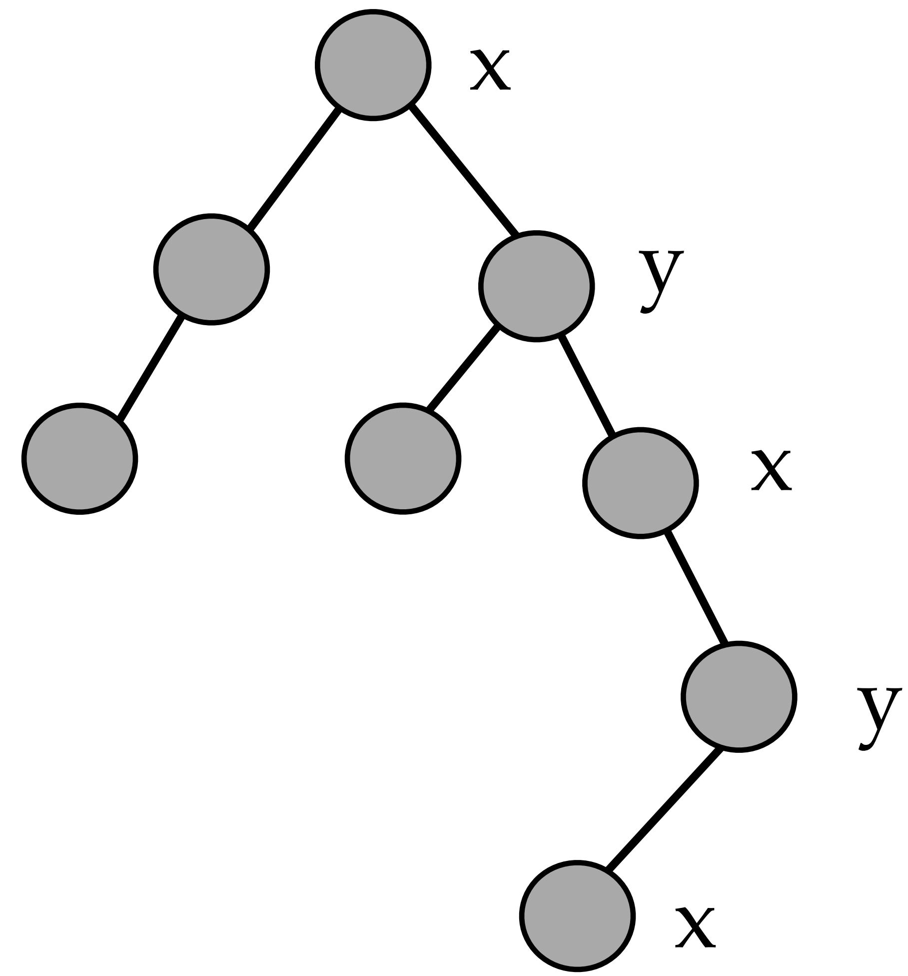
- Invented in 1970s by Jon Bentley
- Name originally meant "3d-trees, 4d-trees, etc" where k was the # of dimensions
- Now, people say "kd-tree of dimension d"

- Idea: Each level of the tree compares against 1 dimension.
- Let's us have only **two children** at each node (instead of 2^d)

kd-trees

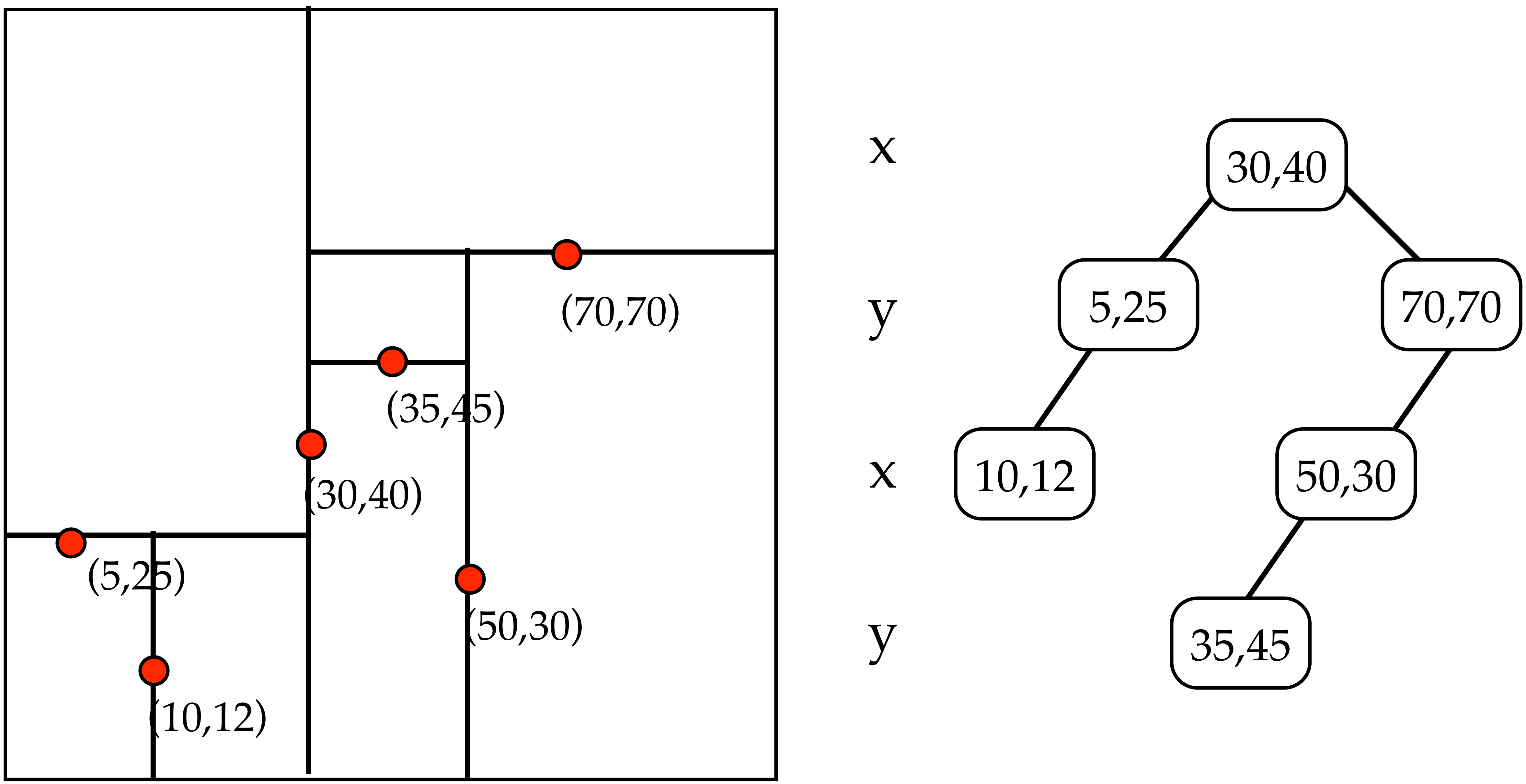
- Each level has a "cutting dimension"
- Cycle through the dimensions as you walk down the tree.
- Each node contains a point $P = (x,y)$

- To find (x',y') you only compare coordinate from the cutting dimension
  - e.g. if cutting dimension is x, then you ask: is $x' < x$?



kd-tree example

insert: (30,40), (5,25), (10,12), (70,70), (50,30), (35,45)



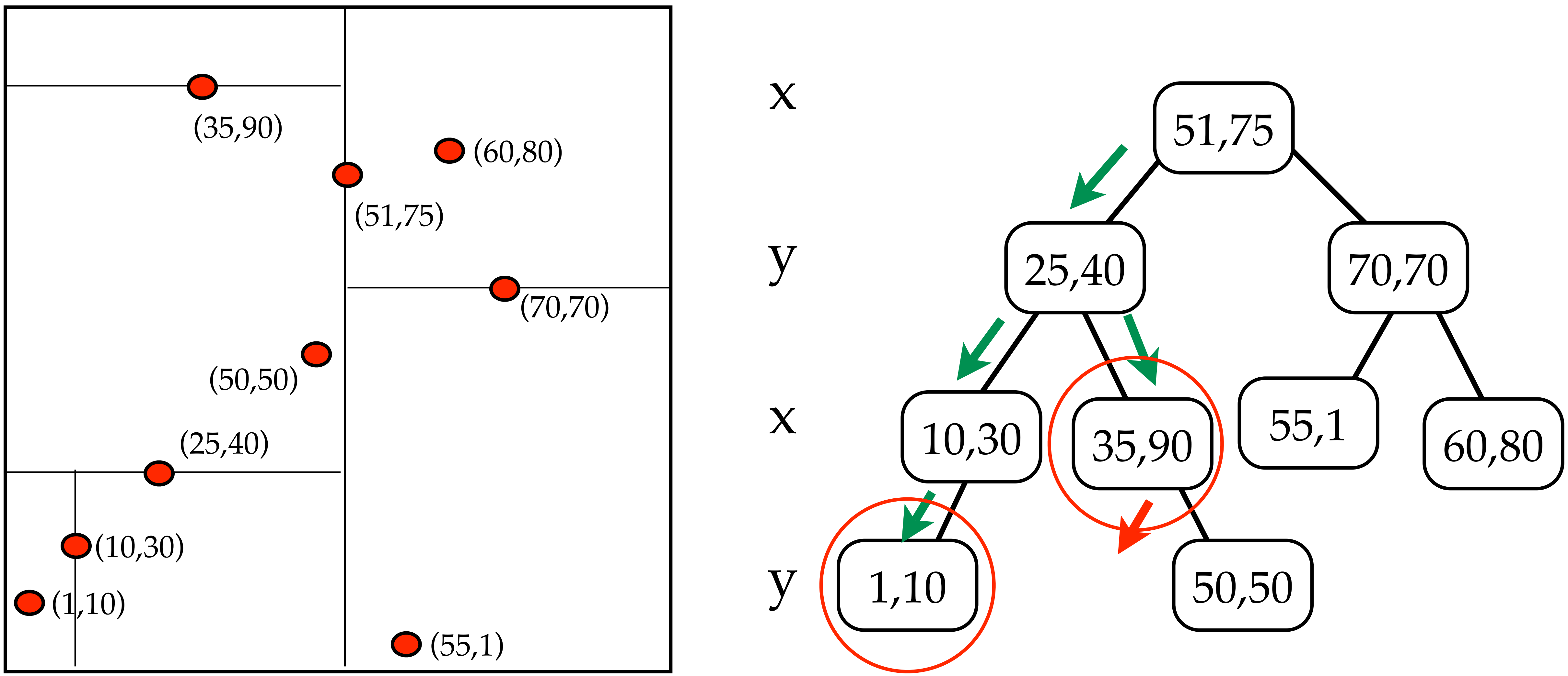
FindMin in kd-trees

- FindMin(d): find the point with the smallest value in the dth dimension.

- Recursively traverse the tree
- If cutdim(current_node) = d, then the minimum can't be in the right subtree, so recurse on just the left subtree
  - if no left subtree, then current node is the min for tree rooted at this node.
- If cutdim(current_node) ≠ d, then minimum could be in *either* subtree, so recurse on both subtrees.
  - (unlike in 1-d structures, often have to explore several paths down the tree)

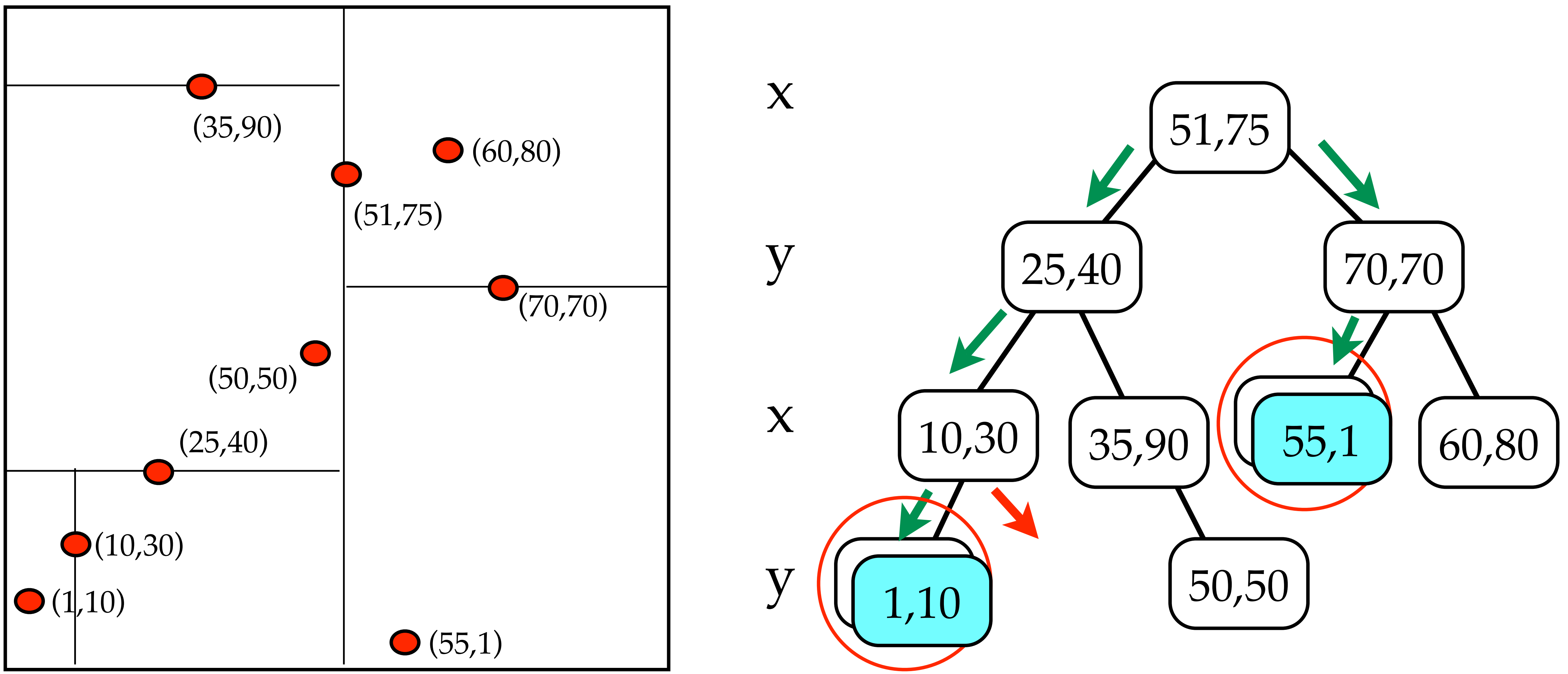
FindMin

FindMin(x-dimension):



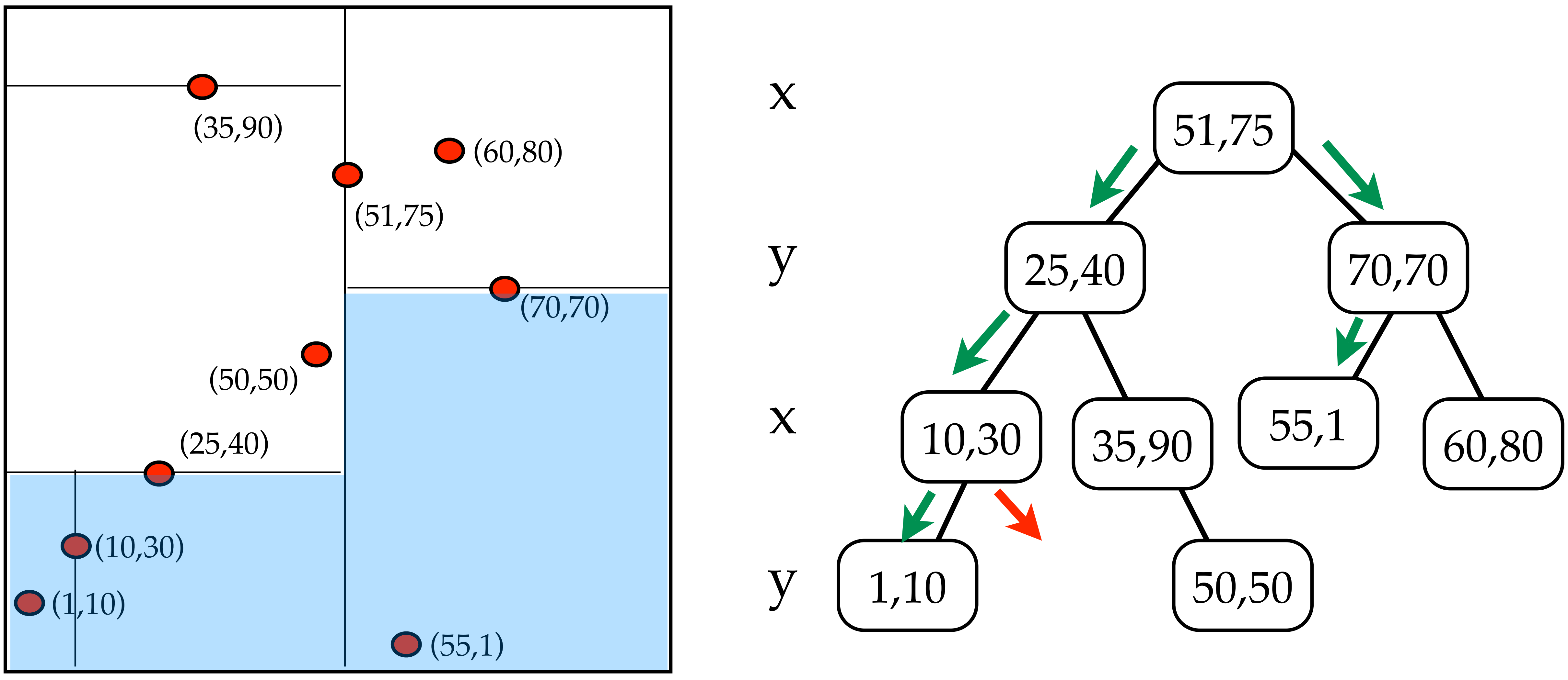
FindMin

FindMin(y-dimension):



FindMin

FindMin(y-dimension): space searched



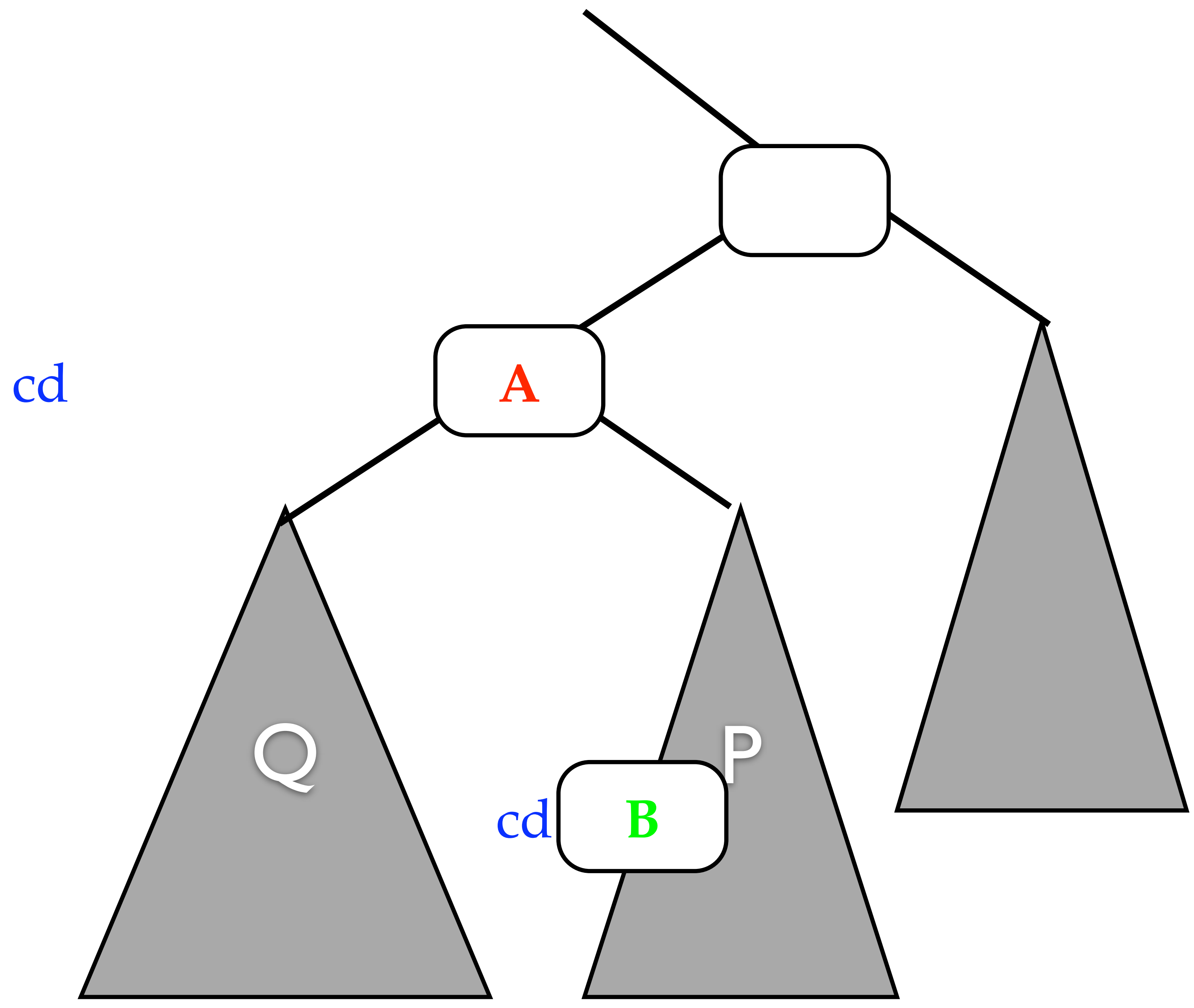
Delete in kd-trees

Want to delete node A. Assume cutting dimension of A is cd

In BST, we'd *findmin*(A.right).

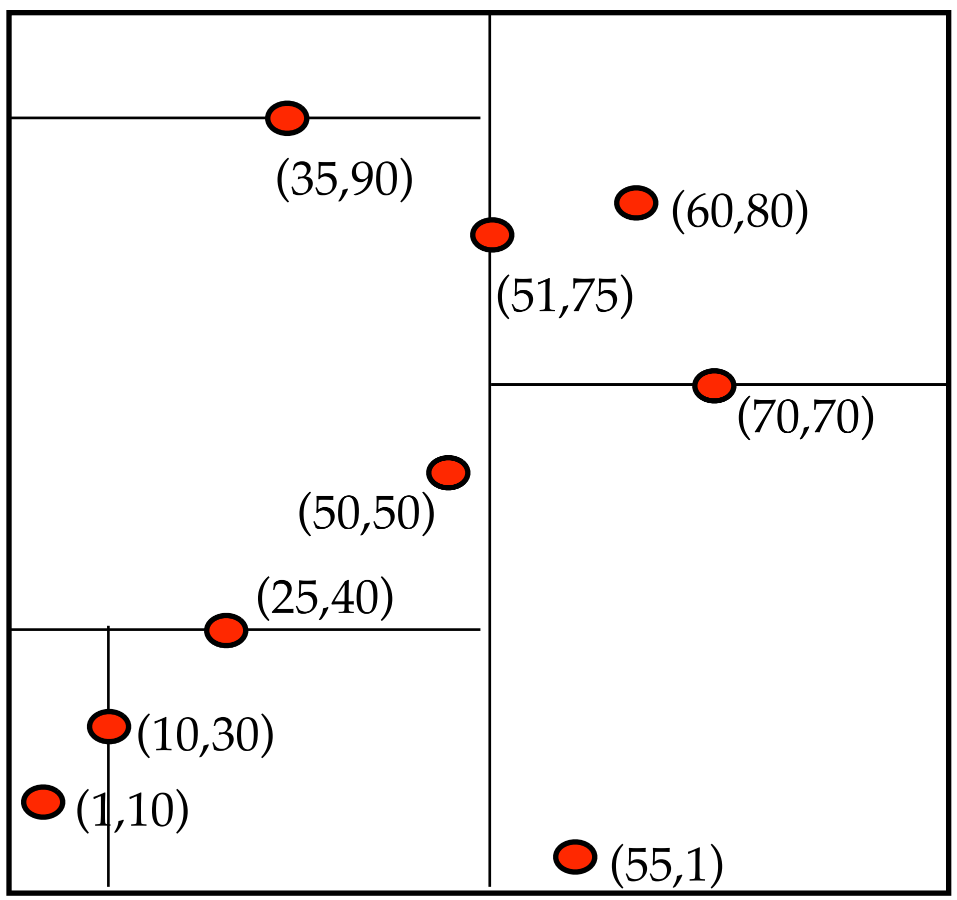
Here, we have to *findmin*(A.right, cd)

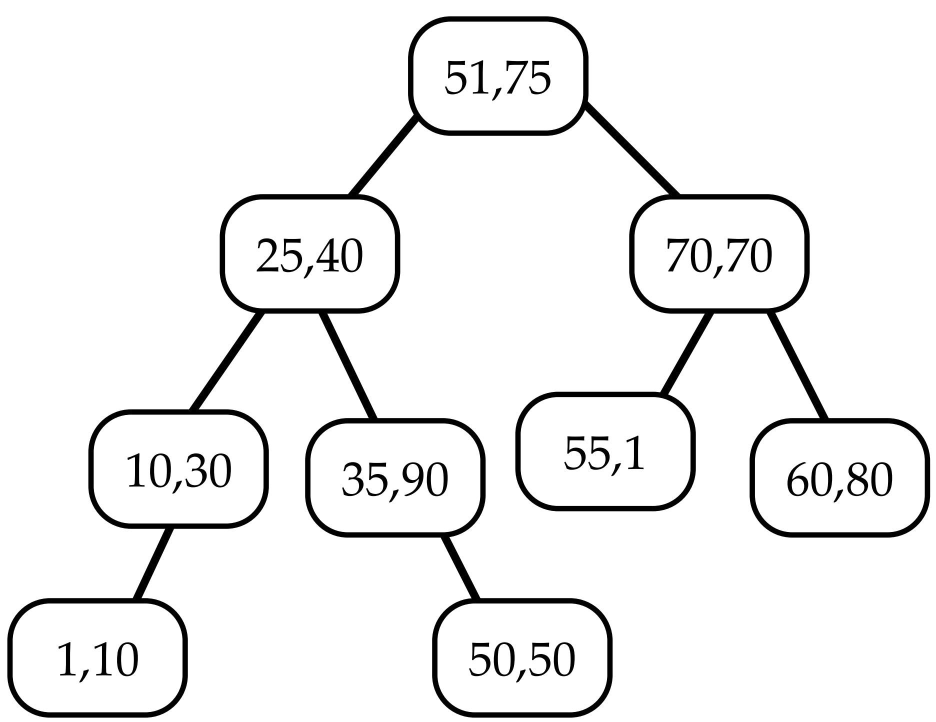
Everything in Q has cd-coord < B, and everything in P has cd-coord ≥ B



Nearest Neighbor Searching in kd-trees

- Nearest Neighbor Queries are very common: given a point Q find the point P in the data set that is closest to Q.
- Doesn't work: find cell that would contain Q and return the point it contains.
  - Reason: the nearest point to P in space may be far from P in the tree:
  - E.g. NN(52,52):





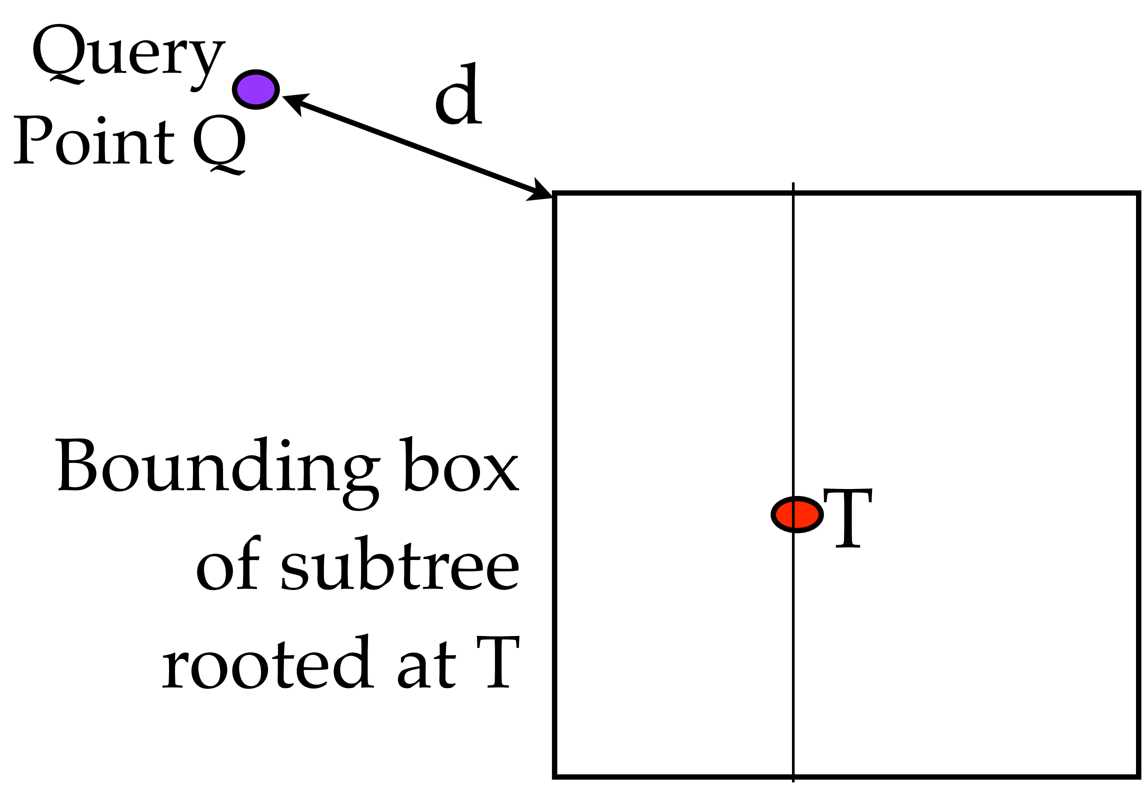
kd-Trees Nearest Neighbor

- Idea: traverse the whole tree, **BUT make two modifications to prune to search space:**

1. Keep variable of closest point C found so far. Prune subtrees once their bounding boxes say that they can't contain any point closer than C

2. Search the subtrees in order that maximizes the chance for pruning

Nearest Neighbor: Ideas, continued



If d > dist(C, Q), then no point in BB(T) can be closer to Q than C. Hence, no reason to search subtree rooted at T.

Update the best point so far, if T is better:
if dist(C, Q) > dist(T.data, Q), C := T.data

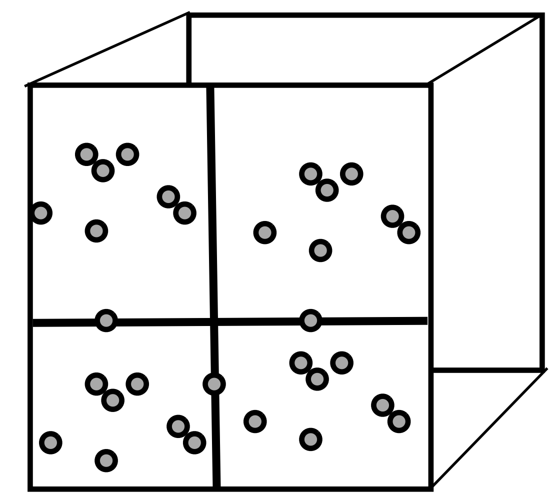
Recurse, but start with the subtree "closer" to Q:
First search the subtree that would contain Q if we were inserting Q below T.

Nearest Neighbor Facts

- Might have to search close to the whole tree in the worst case. [O(n)]
- In practice, runtime is closer to:
  - $O(2^d + \log n)$
  - log n to find cells "near" the query point
  - 2^d to search around cells in that neighborhood
- Three important concepts that reoccur in range / nearest neighbor searching:
  - *storing partial results*: keep best so far, and update
  - *pruning*: reduce search space by eliminating irrelevant trees.
  - *traversal order*: visit the most promising subtree first.

kd-tree Variants

- How do you pick the cutting *dimension*?
  - kd-trees cycle through them, but may be better to pick a different dimension
  - e.g. Suppose your 3d-data points all have same Z-coordinate in a give region:

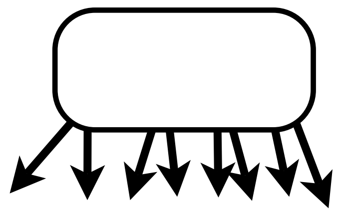


- How do you pick the cutting *value?*
  - kd-trees pick a key value to be the cutting value, based on the order of insertion
  - *optimal kd-trees:* pick the key-value as the median
  - Don't need to use key values => like PR Quadtrees => PR kd-trees

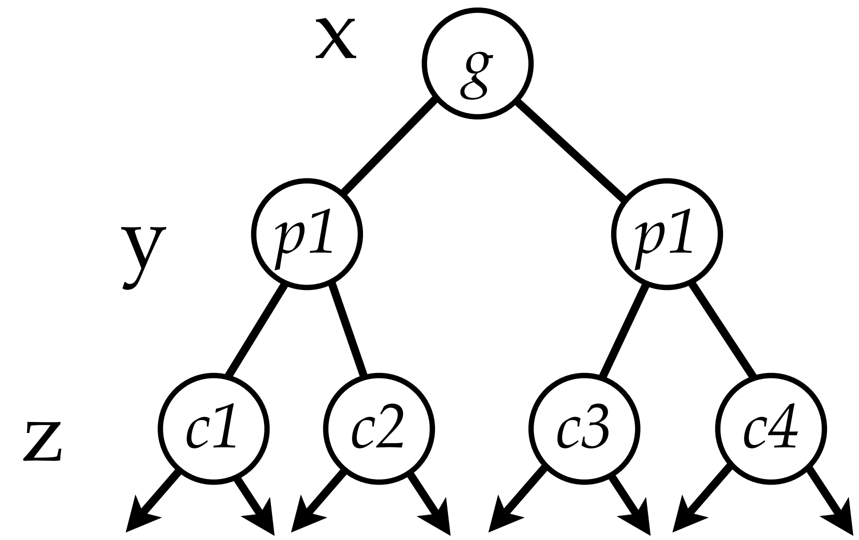
- What is the size of leaves?
  - if you allow more than 1 key in a cell: *bucket kd-trees*
- kd-trees: discriminator = (hyper)plane;
  quadtrees (and higher dim) discriminator complexity grows with *d*

kd-Trees vs. Quadtrees, another view

Consider a 3-d data set



Octtree



kd-tree

kd-tree splits the decision up over d levels

don't have to represent levels (pointers) that you don't need

Quadtrees: one *point* determines all splits

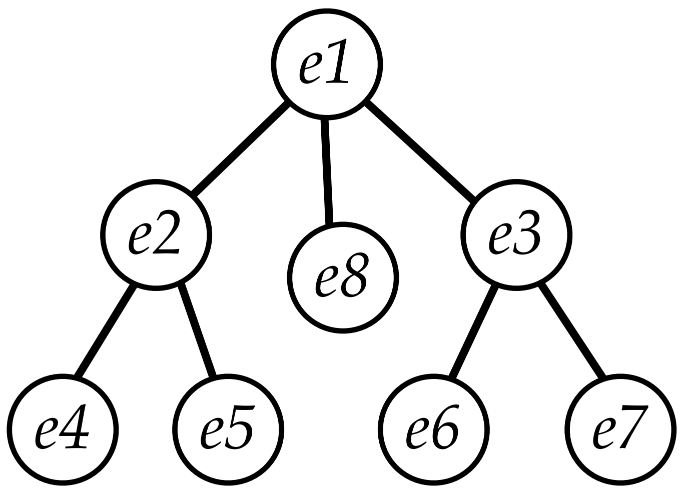
kd-trees: flexibility in how splits are chosen

Generalized Nearest Neighbor Search

- Saw last time: nearest neighbor search in kd-trees.
- What if you want the k-nearest neighbors?
- What if you don't know k?
  - E.g.: Find me the closest gas station with price < $3.25 / gallon.
  - Approach: go through points (gas stations) in order of distance from me until I find one that meets the $ criteria
- Need a NN search that will find points in order of their distance from a query point q.
- Same idea as the kd-tree NN search, just more general

Generalized NN Search

- A feature of all spatial DS we've seen so far: decompose space hierarchically.
  No matter what the DS, we get something like this:



Let the items in the hierarchy be e1,e2,e3...

Items may represent points, or bounding boxes, or ...

Let Type(e) be an abstract "type" of the object:

we use the type to determine which distance function to use

E.g: if Type = "bounding box" then we'd use the point-to-rectangle distance function.

A concrete example: in a Quadtree: internal nodes have type "bounding box"

Leaves would have type "point"

Generalized, Incremental NN

Let IsLeaf(), Children(), and Type() represent the decomposition tree

Let $d_t(q,e_t)$ be the distance function appropriate to compare points with elements of type t.

Idea: keep a priority queue that contains *all elements* visited so far (points, bounding boxes)

Priority queue (heap) is ordered by distance to the query point

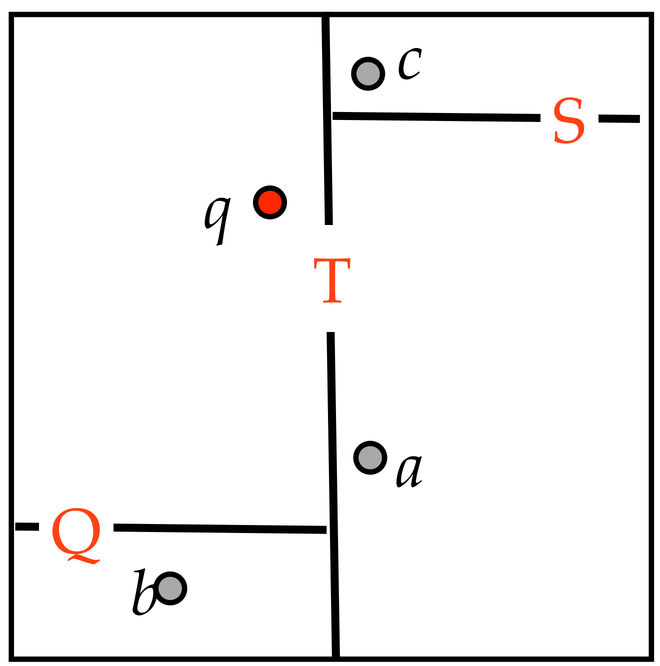
When you dequeue a point (leaf), it will be the next closest

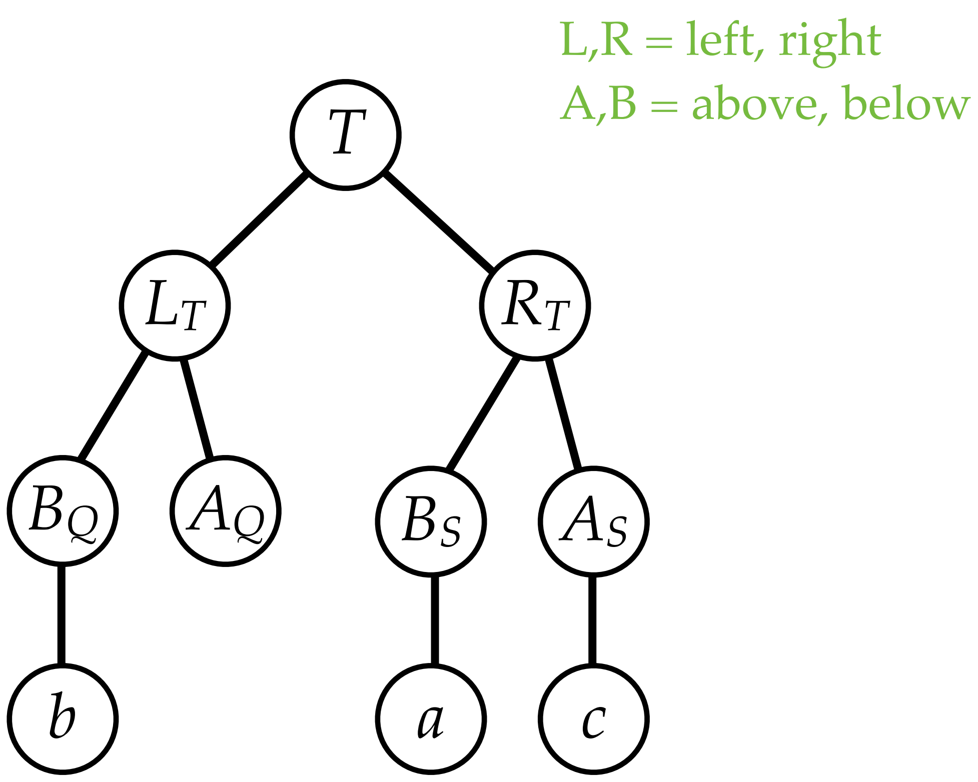
```
HeapInsert(H, root, 0)
while not Empty(H):
   e := ExtractMin(H)
   if IsLeaf(e):
      output e as next nearest
   else
      foreach c in Children(e):
        t = Type(c)
        HeapInsert(H, c, dt(q,c))
```

$d_t(q,c)$ may be the distance to the bounding box represented by *c*, e.g.

Incremental, Generalized NN Example

Some spatial data structure:



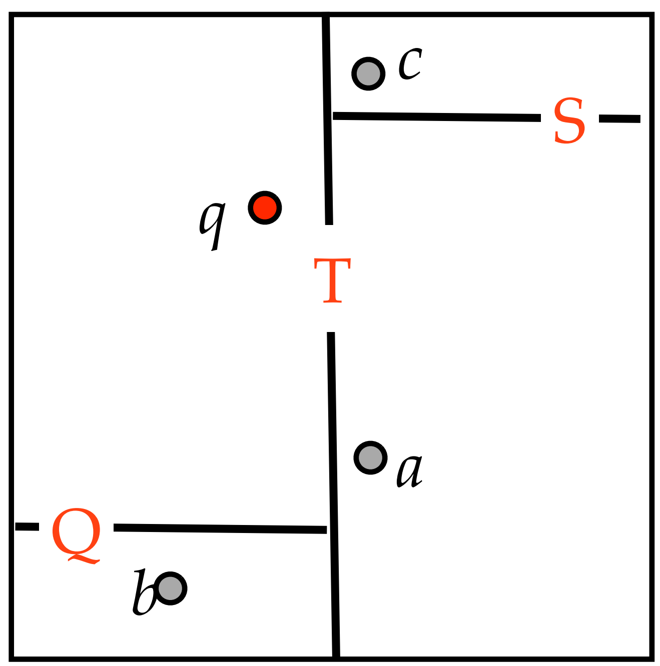


It's spatial decomposition (**NOT** the actual data structure)

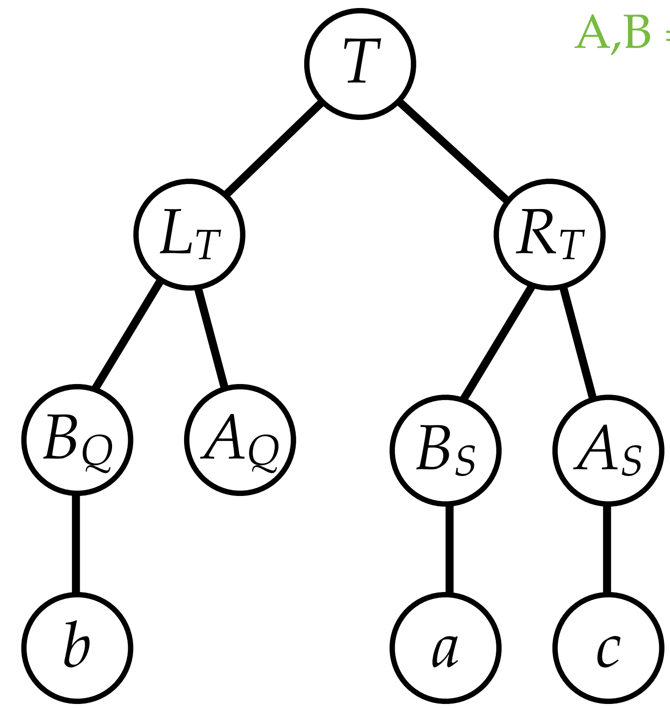
```
HeapInsert(H, root, 0)
while not Empty(H):
   e := ExtractMin(H)
   if IsLeaf(e):
      output e as next nearest
   else
      foreach c in Children(e):
        t = Type(c)
        HeapInsert(H, c, dt(q,c))
```

Incremental, Generalized NN Example

Some spatial data structure:



L,R = left, right
A,B = above, below



Its spatial decomposition (**NOT** the actual data structure)

```
HeapInsert(H, root, 0)
while not Empty(H):
   e := ExtractMin(H)
   if IsLeaf(e) && IsPoint(e):
      output e as next nearest
   else
      foreach c in Children(e):
        t = Type(c)
        HeapInsert(H, c, d_t(q,c))
```

$H = []$

$H = [T]$

$H = [L_T\ R_T]$

$H = [A_Q\ R_T\ B_Q]$

$H = [R_T\ B_Q]$

$H = [B_S\ A_S\ B_Q]$

$H = [A_S\ a\ B_Q]$

$H = [c\ a\ B_Q]$

$H = [c\ a\ b]$

$H = [a\ b]$

$H = [b]$

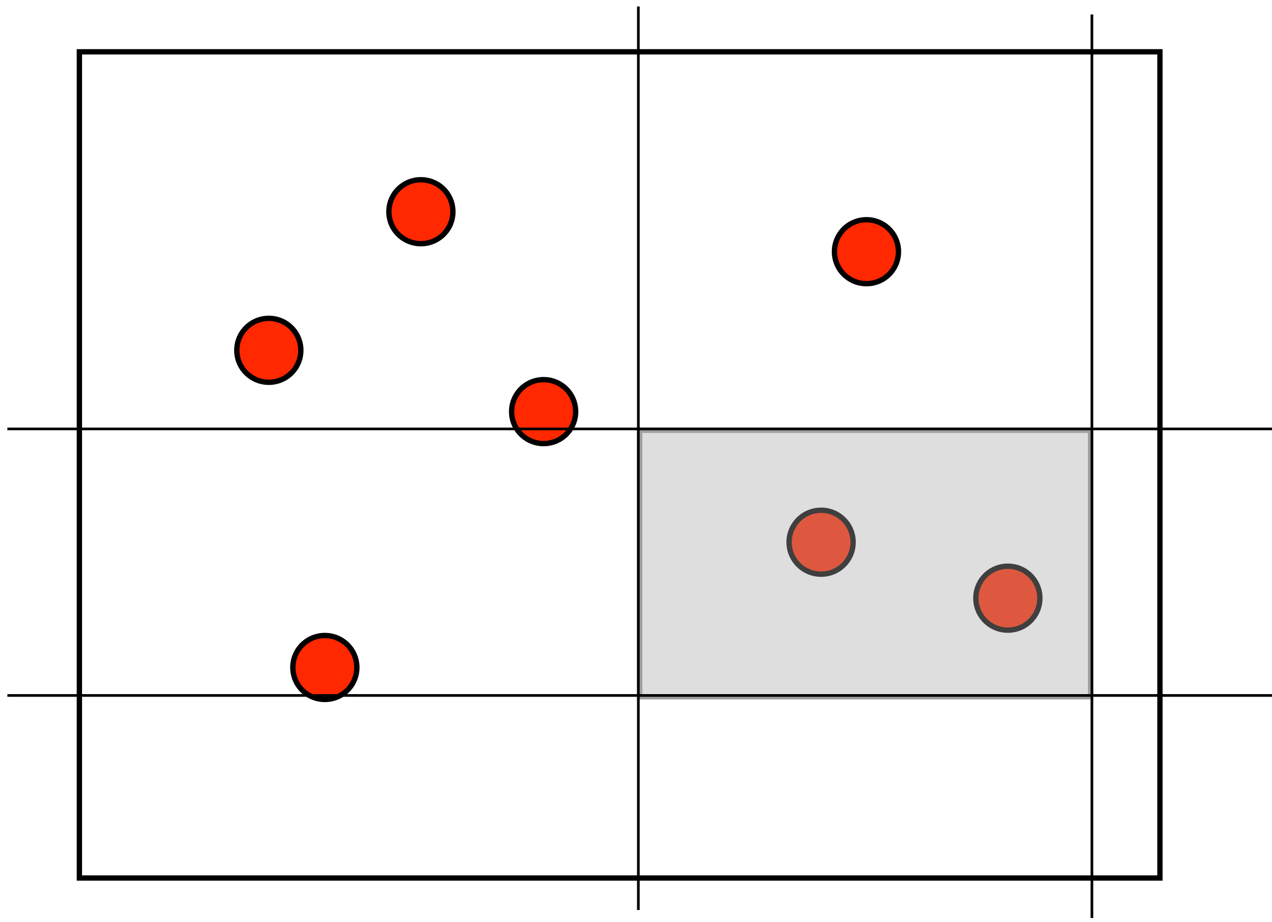
$H = []$

Range Searching in kd-trees

- Range Searches: another extremely common type of query.

- Orthogonal range queries:
  - Given axis-aligned rectangle
  - Return (or count) all the points inside it

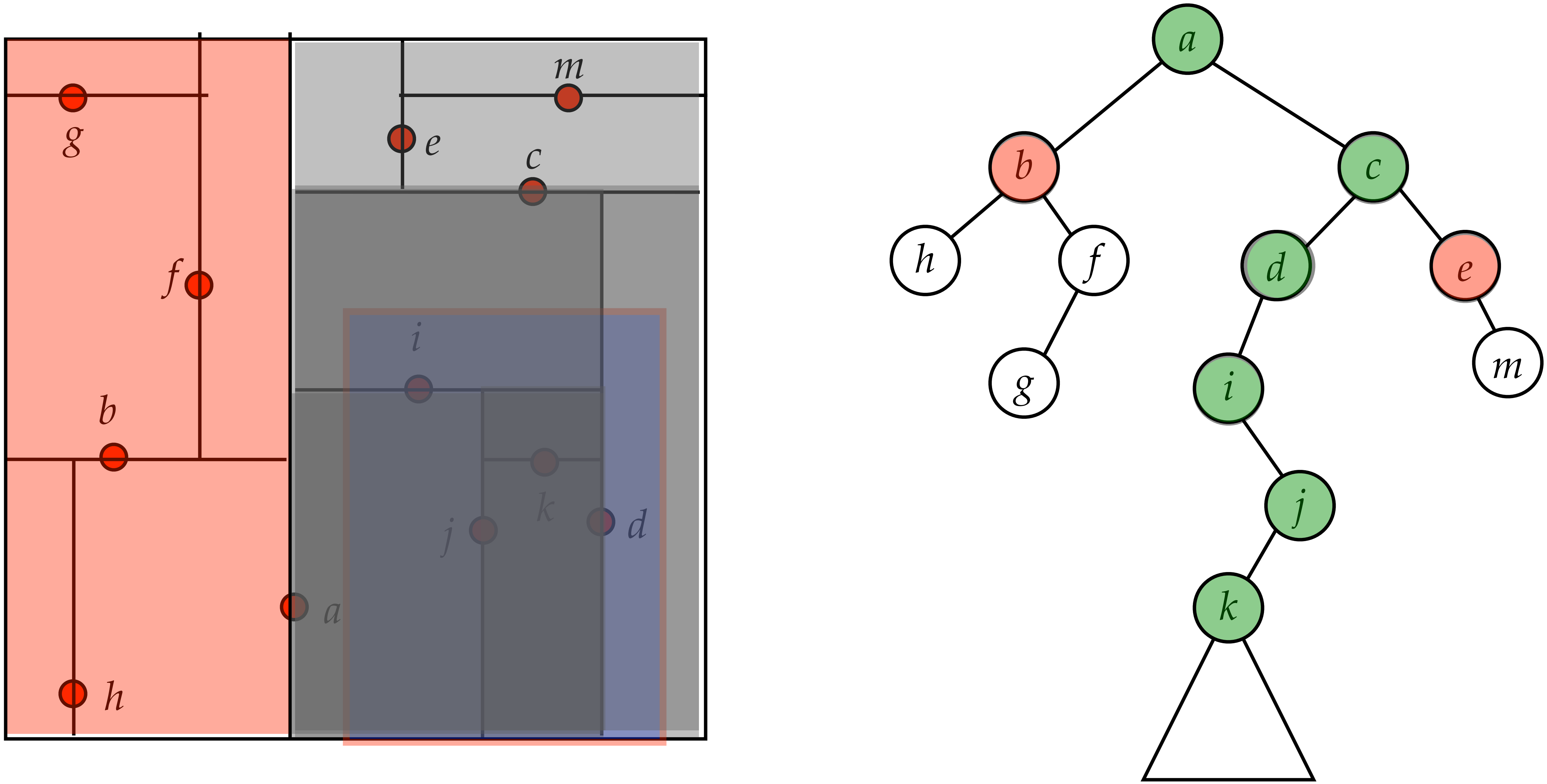
- **Example**: find all people between 20 and 30 years old who are between 5′8″ and 6′ tall.



Range Searching in kd-trees

- Basic algorithmic idea:
  - traverse the whole tree, **BUT**
    - prune if bounding box doesn't intersect with Query
    - stop recursing or print all points in subtree if bounding box is entirely inside Query

Range Searching Example



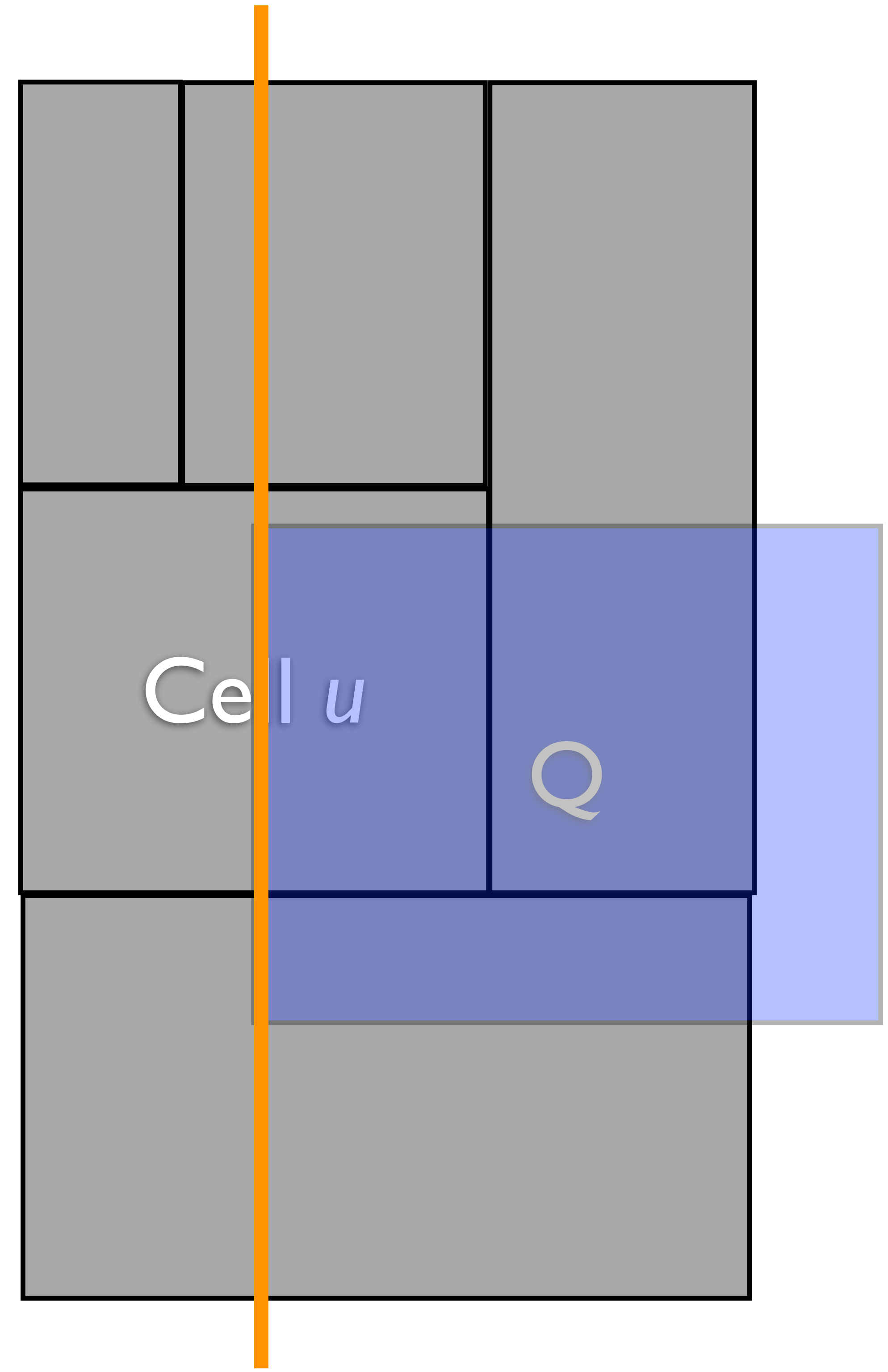
If query box doesn't overlap bounding box, stop recursion

If bounding box is a subset of query box, report all the points in current subtree

If bounding box overlaps query box, recurse left and right.

Expected # of Nodes to Visit

- Completely process a node only if query box intersects bounding box of the node's cell:
- In other words, one of the edges of Q must cut through the cell.
- # of cells a vertical line will pass through ≥ the number of cells cut by the left edge of Q.
- Top, bottom, right edges are the same, so bounding # of cells cut by a vertical line is sufficient.



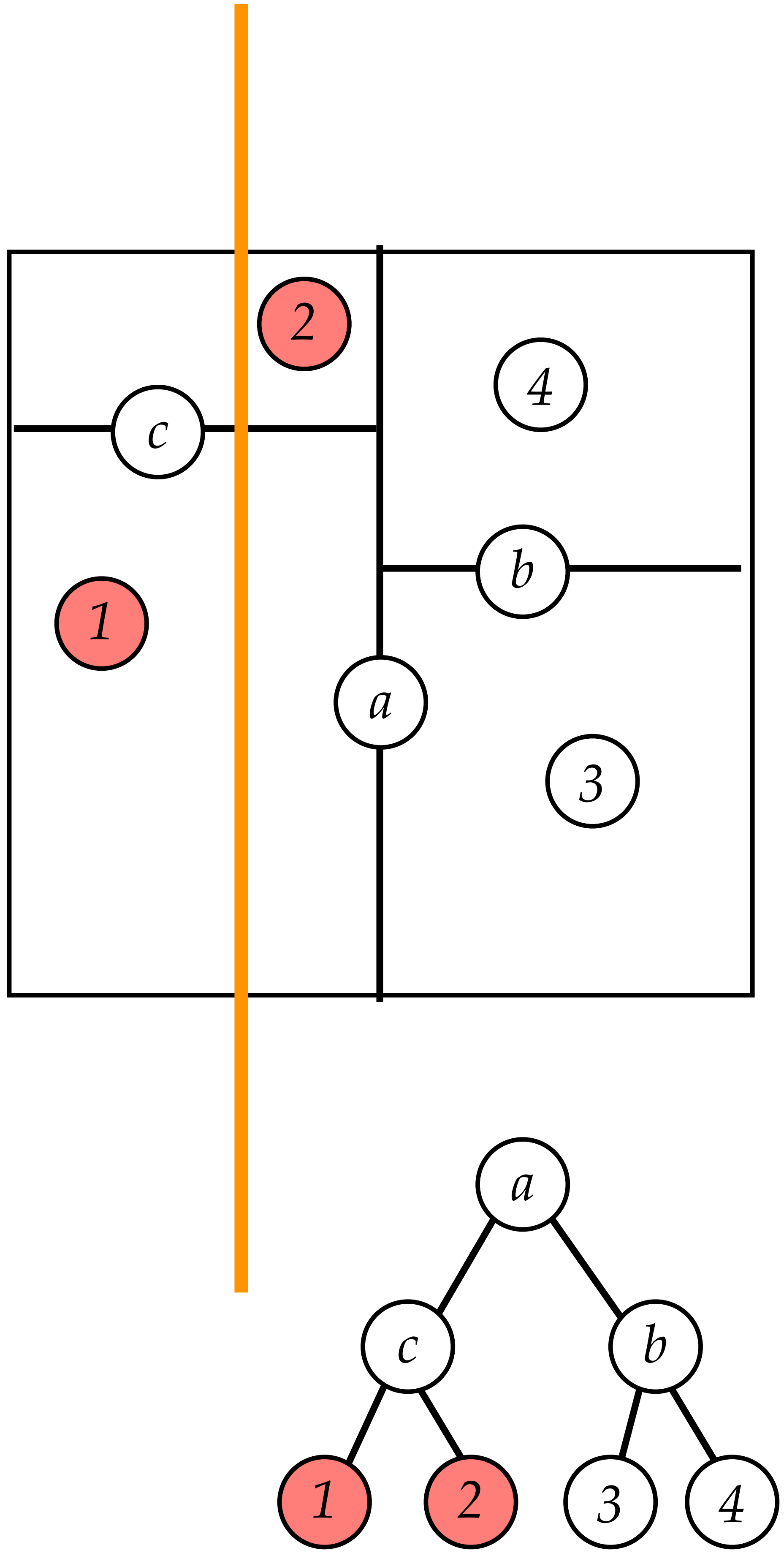
of Stabbed Nodes = O(√n)

Consider a node *a* with cutting dimension = x

Vertical line can intersect exactly one of *a*'s children (say *c*)

But will intersect *both* of c's children.

Thus, line will intersect at most 2 of *a*'s grandchildren.



of Stabbed Nodes = O(√n)

So: you at most double #
of cut nodes every 2 levels

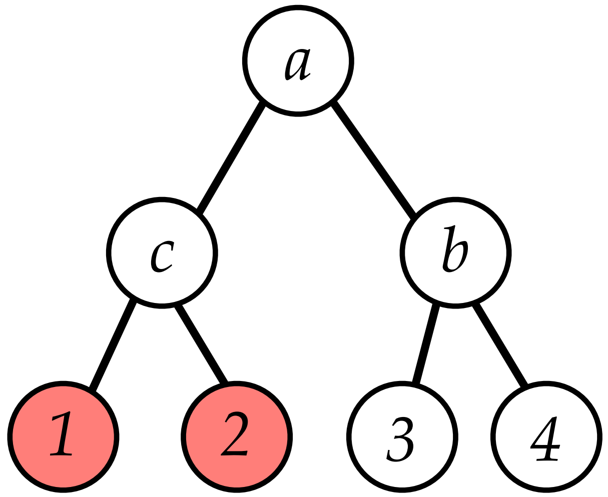
If kd-tree is balanced, has
O(log n) levels

Cells cut

$= 2^{(\log n)/2}$

$= 2^{\log \sqrt{n}}$

$= \sqrt{n}$



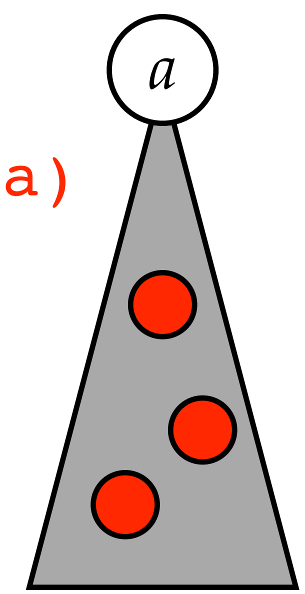
Assuming random input, or all points known ahead of time, you'll get a balanced tree.

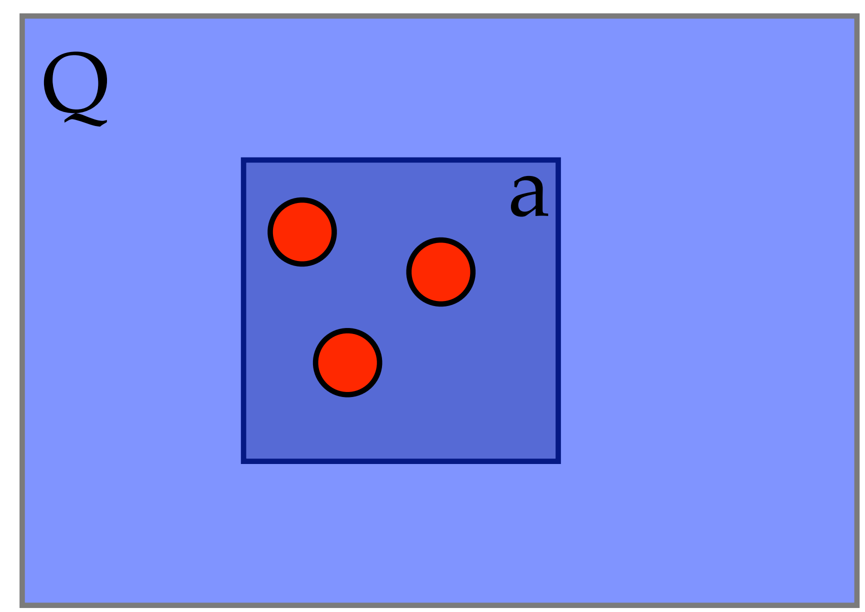
Each side of query rectangle stabs < O(√n) cells. So whole query stabs at most O(4√n) = O(√n) cells.

Suppose we want to output all points in region

- Then cost is O(k + √n)
  - where k is # of points in the query region.
- Why? Because: you visit every stabbed node [O(√n) of them] + every node in the subtrees rooted in the contained cells.
  - Takes linear time to traverse such subtrees
- Example of *output sensitive* running time analysis: running time depends on size of the *output*.

AllNodesUnder(a)





kd-tree Summary:

- Use $O(n)$ storage [1 node for each point]

- If all points are known in advance, balanced kd-tree can be built in $O(n \log n)$ time
  - Recall: sort the points by x and y coordinates
  - Always split on the median point so each split divides remaining points nearly in half.
  - Time dominated by the initial sorting.

- Can be orthogonal range searched in $O(\sqrt{n} + k)$ time.

- Can we do better than $O(\sqrt{n})$ to range search?
  - (possibly at a cost of additional space)

1-Dimensional Range Trees

- Suppose you have "points" in 1-dimension (aka numbers)
- Want to answer range queries: "Return all keys between x_1 and x_2."
- How could you solve this?

Balanced Binary Search Tree

Range Queries on Binary Search Trees

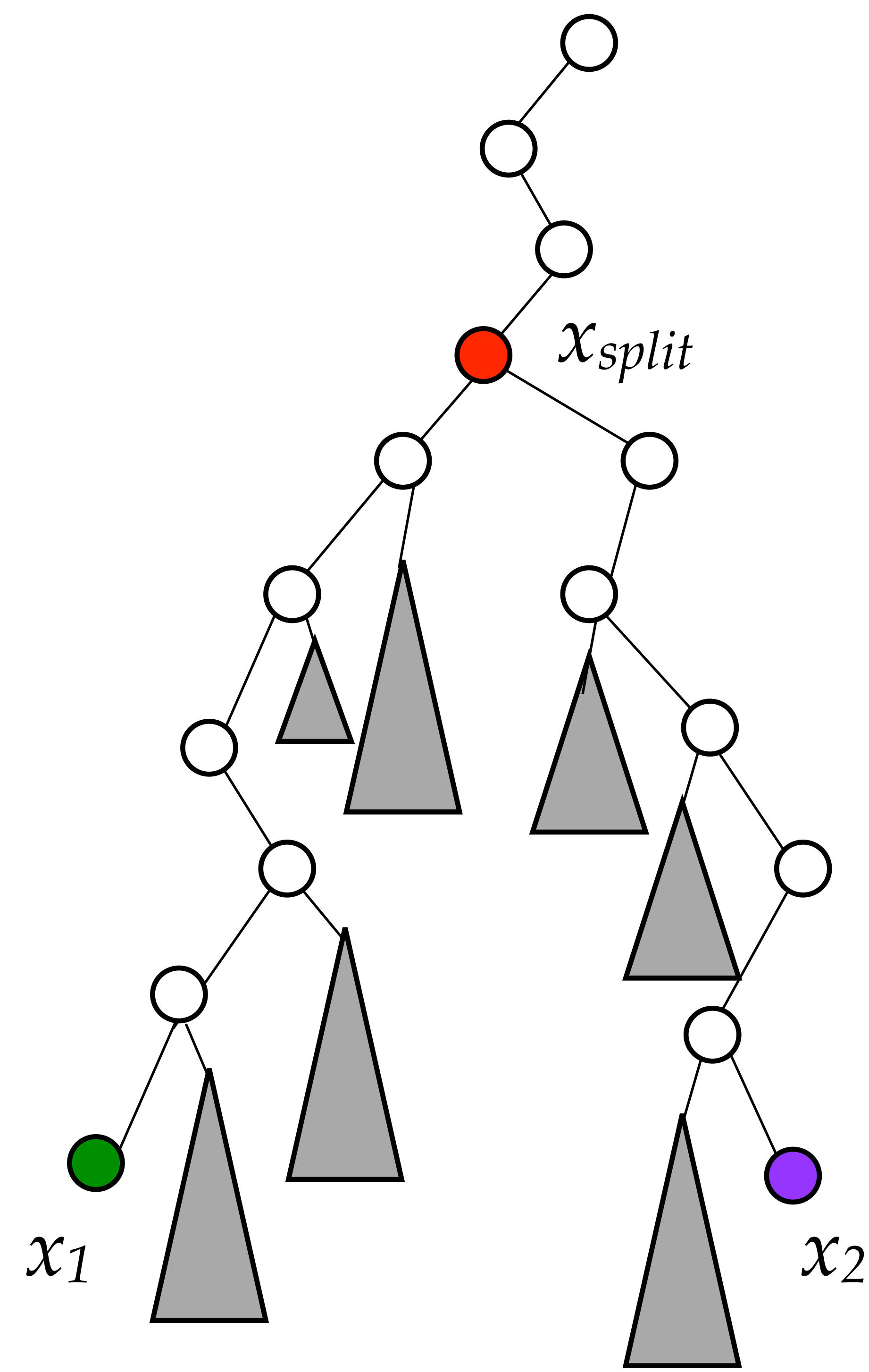
Assume all data are in the leaves

Search for x_1 and x_2

Let x_{split} be the node were the search paths diverge

Output leaves in the right subtrees of nodes on the path from x_{split} to x_1

Output leaves in the left subtrees of nodes on the path from x_{split} to x_2

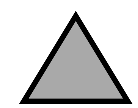


1-D Query Time

- O(k + log n), where k is the number of points output.
  - Tree is balanced, so depth is O(log n)
  - Length of paths to x1 and x2 are O(log n)
  - Therefore visit O(log n) nodes to find the roots of subtrees to output
  - Traversing the subtrees is linear, O(k), in the number of items output.

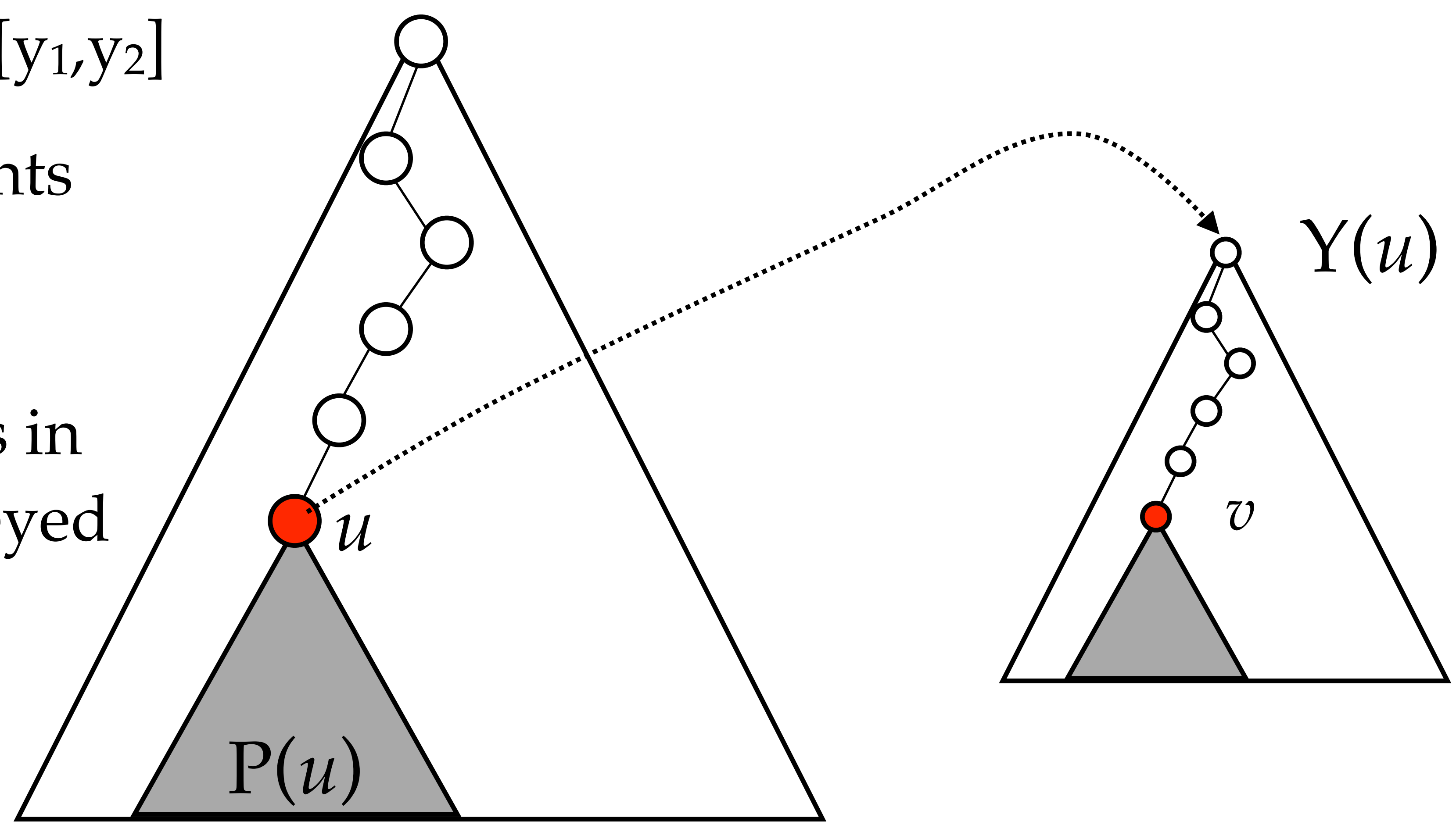
How would you generalize to 2d?

2d Range Trees

- Treat range query as 2 nested one-dimensional queries:
  - $[x_1,x_2]$ by $[y_1,y_2]$
  - First ask for the points with x-coordinates in the given range $[x_1,x_2]$ => a set of subtrees 
  - Instead of all points in these subtrees, only want those that fall in $[y_1,y_2]$

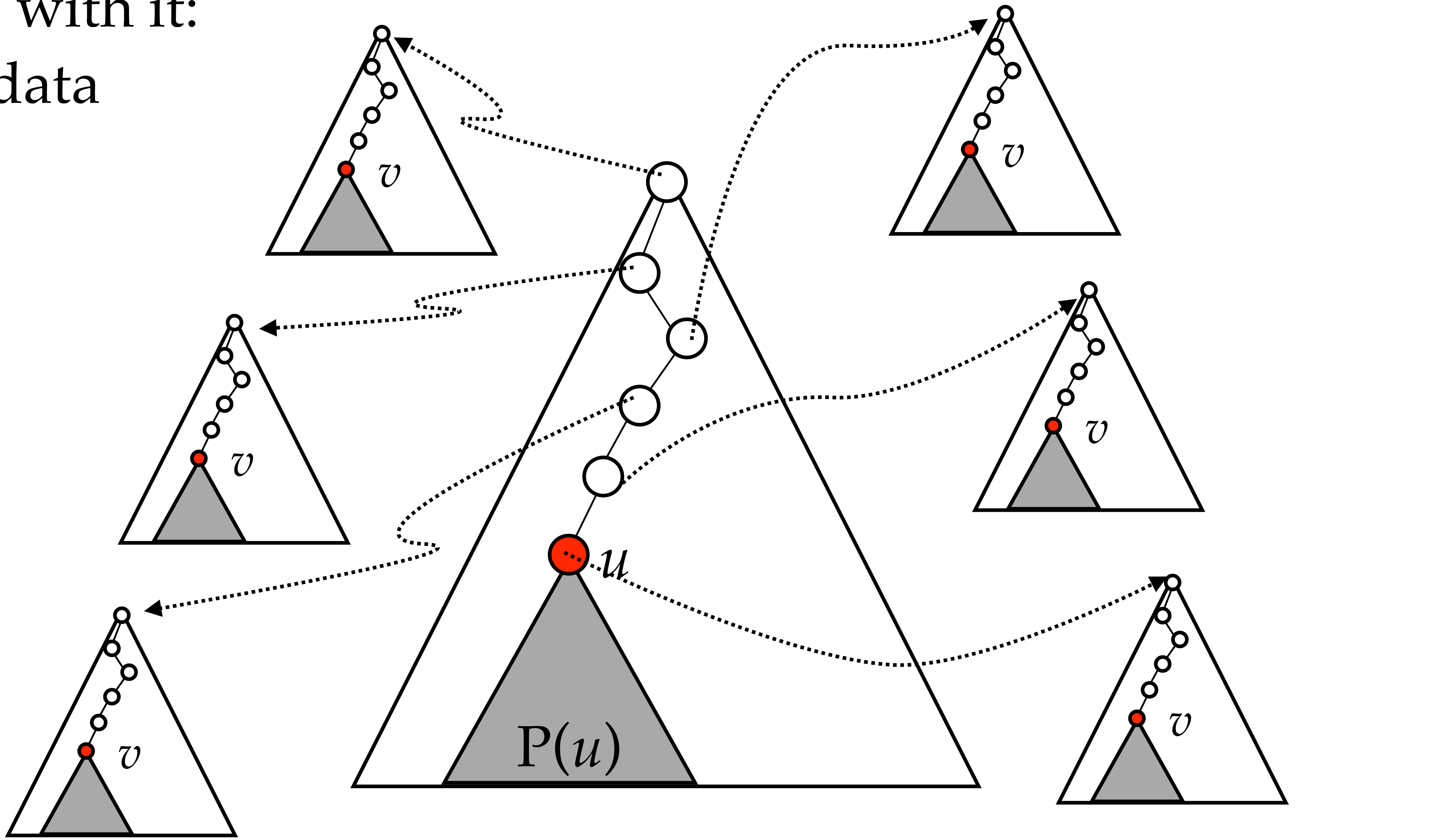
P(u) is the set of points under *u*

We store *those* points in another tree Y(u), keyed by the y-dimension

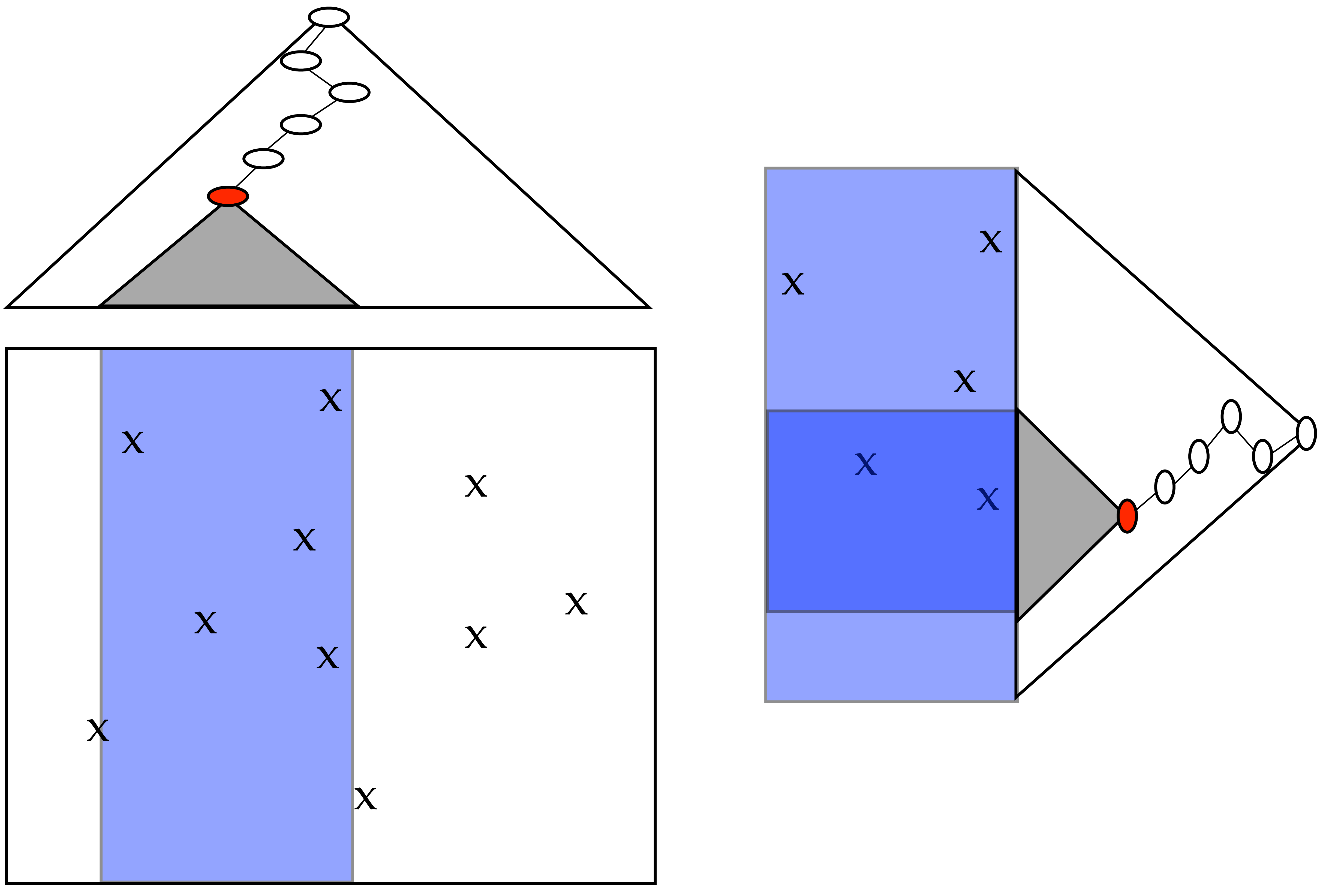


2-D Range Trees, Cont.

Every node has a tree associated with it: *multilevel* data structure



Range Trees, continued.



2d-range tree space requirements

- Sum of the sizes of Y(u) for u at a given depth is O(n)
  - Each point stored in the Y(u) tree for at most one node at a given depth
- Since main tree is balanced, has O(log n) depth
- Meaning total space requirement is O(n log n)

2d Range Tree Range Searches

1. First find trees that match the x-constraint;
2. Then output points in those subtrees that match the y-constraint (by 1-d range searching the associated Y(u) trees)

- Step 1 will return at most O(log n) subtrees to process.
- Step 2 will thus perform the following O(log n) times:
  - Range search the Y(u) tree. This takes $O(\log n + k_u)$, where k_u is the number of points output for that Y(u) tree.
- Total time is $\sum_u O(\log n + k_u)$ where u ranges over O(log n) nodes. Thus the total time is $O(\log^2 n + k)$.

kd-tree vs. Range Tree

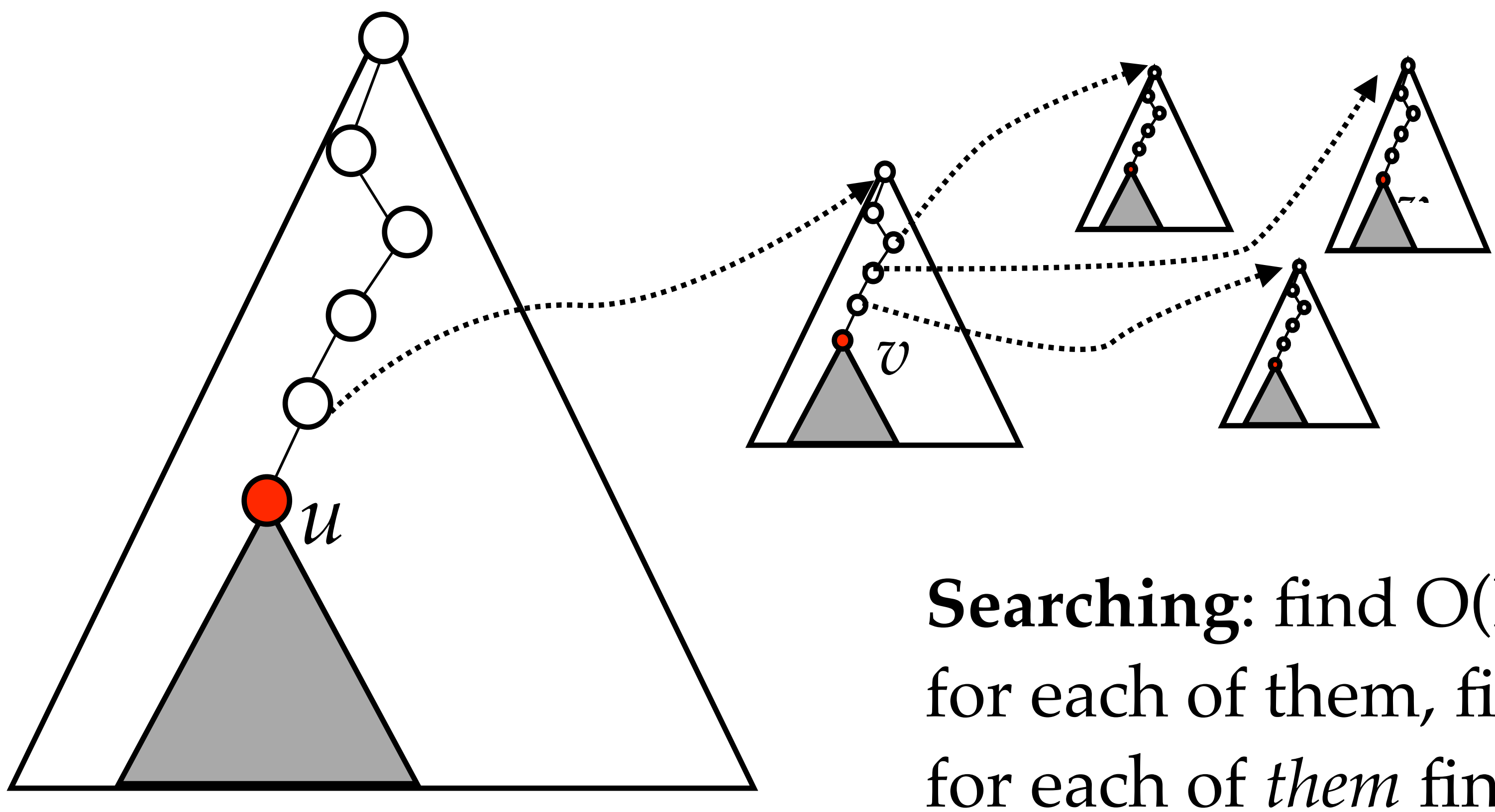
- 2d kd-tree:
  - Space = $O(n)$
  - Range Query Time = $O(k + \sqrt{n})$
  - Inserts $O(\log n)$

- 2d Range Tree:
  - Space = $O(n \log n)$
  - Range Query Time = $O(k + \log^2 n)$
  - Inserts $O(\log^2 n)$

How would you extend this to > 2 dimensions?

Range Trees for d > 2

- Now, your associated trees Y(u) themselves have associated trees Z(v) and so on:



Searching: find O(log n) nodes in first tree
for each of them, find another O(log n) sets
for each of *them* find another log n sets

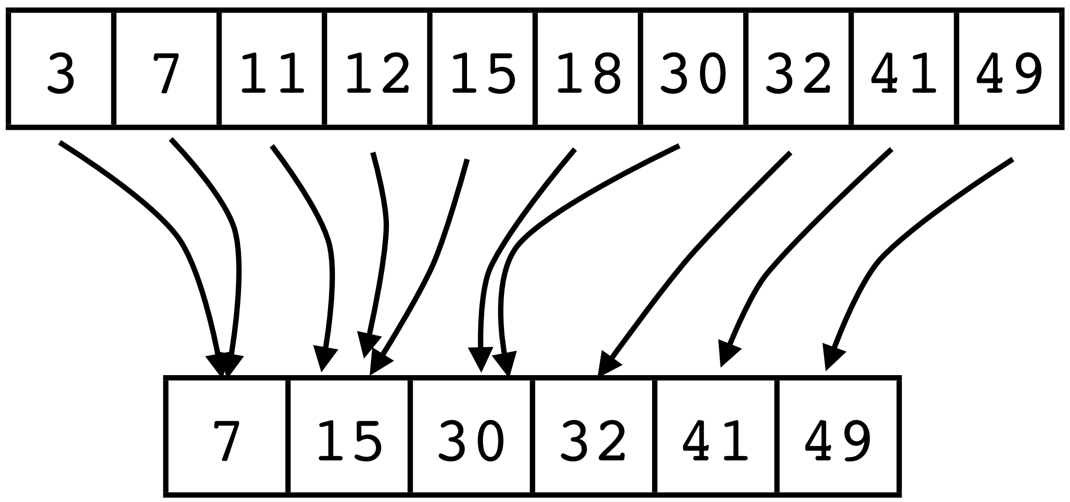
Leads to $O(k+\log^d n)$ search time
Space: $O(n \log^{d-1} n)$ space

Fractional Cascading Speed-up: Idea

- Suppose you had two sorted arrays A_1 A_2
  - Elements in A_2 are subset of those in A_1
  - Want to range search in both arrays with the same range: $[x_1,x_2]$
- Simple:
  - Binary Search to find x_1 in both A_1 and A_2
  - Walk along array until you pass x_2
- O(log n) time for each Binary Search,
  - have to do it twice though

Can do better:

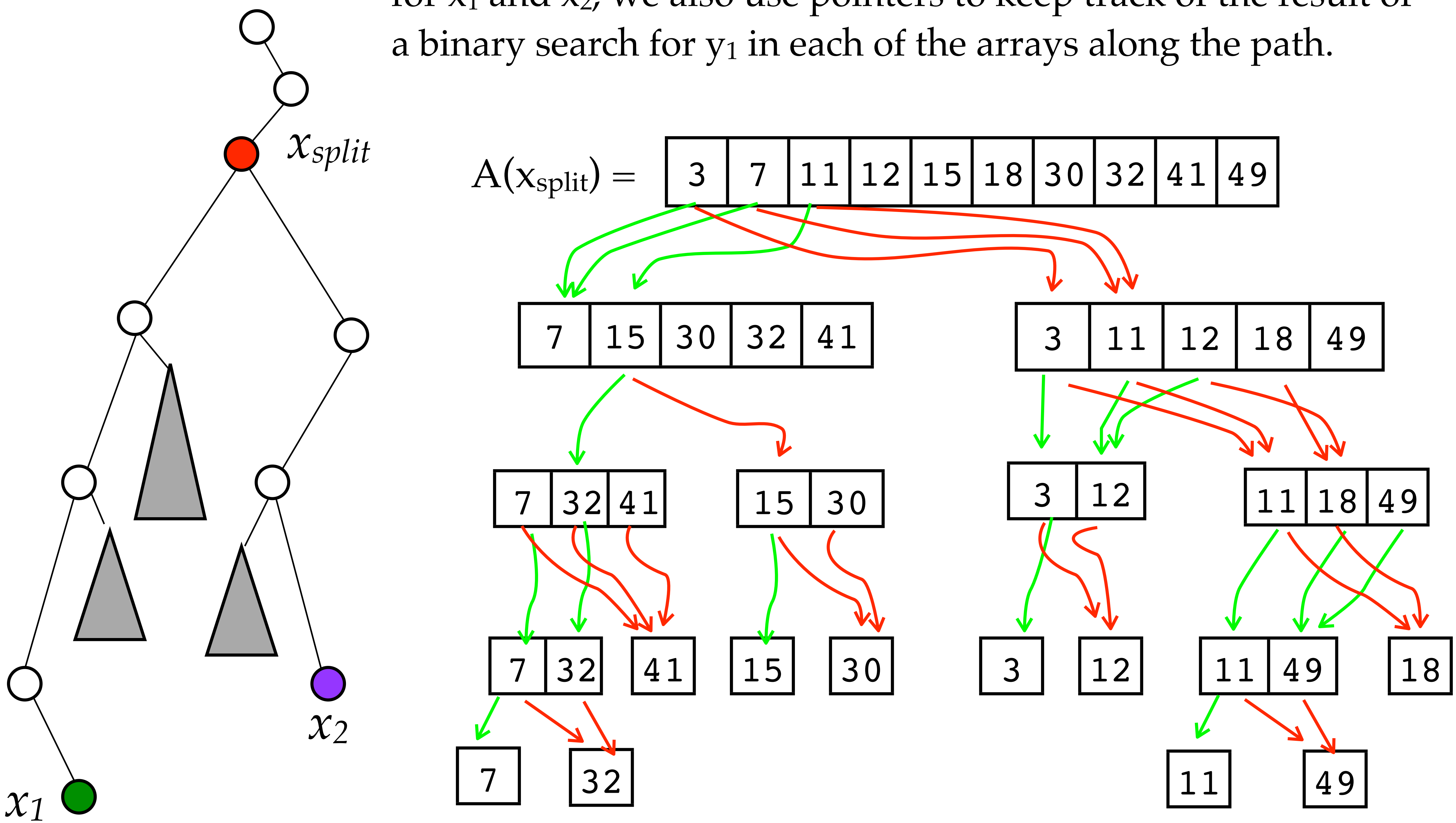
- Since A_2 subset of A_1:
  - Keep pointer at each element u of A_1 pointing to the smallest element of A_2 that is $\geq u$.



  - After Binary Search in A_1, use pointer to find where to start in A_2
- Can do similar in Range Trees to eliminate an O(log n) factor (see next slides)

Fractional Cascading in Range Trees

Instead of an aux. tree, we store an array, sorted by Y-coord. At x_{split}, we do a binary search for y_1. As we continue to search for x_1 and x_2, we also use pointers to keep track of the result of a binary search for y_1 in each of the arrays along the path.



(Only subset of pointers are shown)

Fractional Cascading Search

- RangeQuery([x1,x2] by [y1,y2]):
  - Search for x_{split}
  - Use binary search to find the first point in $A(x_{split})$ that is larger that y_1.
  - Continue searching for x_1 and x_2, following the now diverged paths
  - Let u_1--u_2--u_3--u_k be the path to x_1. While following this path, use the "cascading" pointers to find the first point in each $A(u_i)$ that is larger than y_1. [similarly with the path v_1--v_2--v_m to x_2]
  - If a child of u_i or v_i is the root of a subtree to output, then use a cascading pointer to find the first point larger than y_1, output all points until you pass y_2.

Fractional Cascading: Runtime

- Instead of $O(\log n)$ binary searches, you perform just one

- Therefore, $O(\log^2 n)$ becomes $O(\log n)$

- 2d-rectangle range queries in $O(\log n + k)$ time

- In d dimensions: $O(\log^{d-1} n + k)$