# kd-Trees

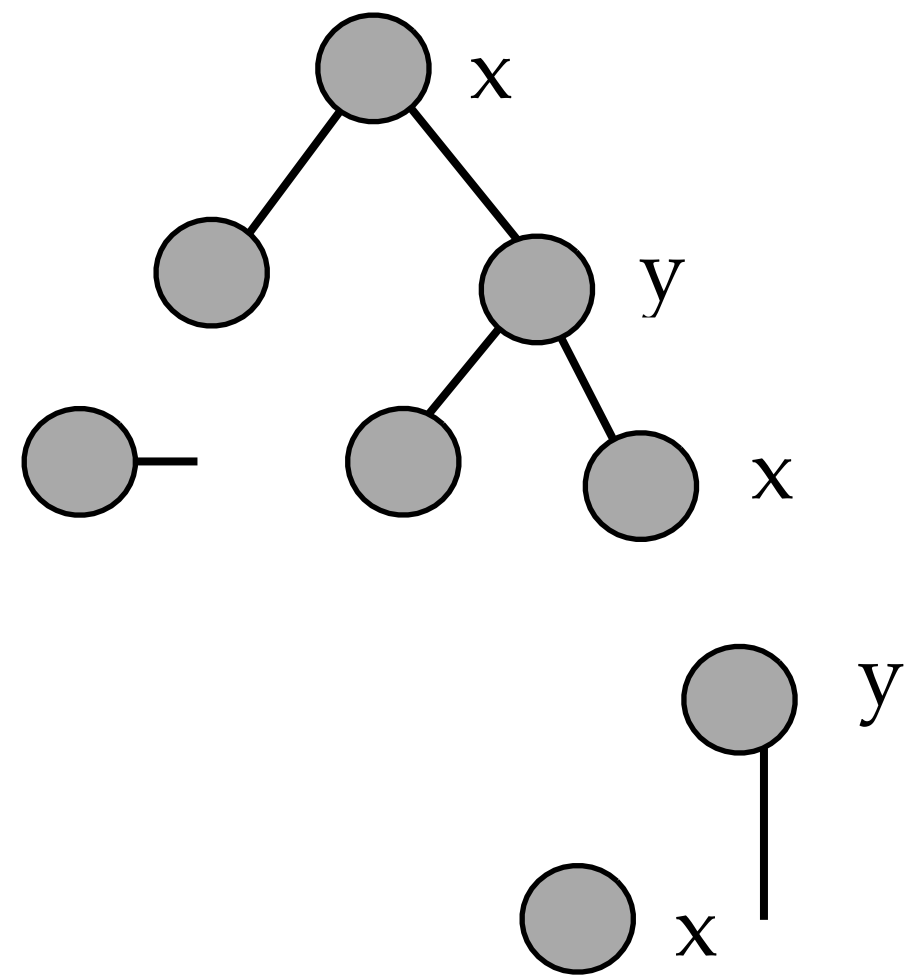
## kd-Trees

- 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 $2^d$)

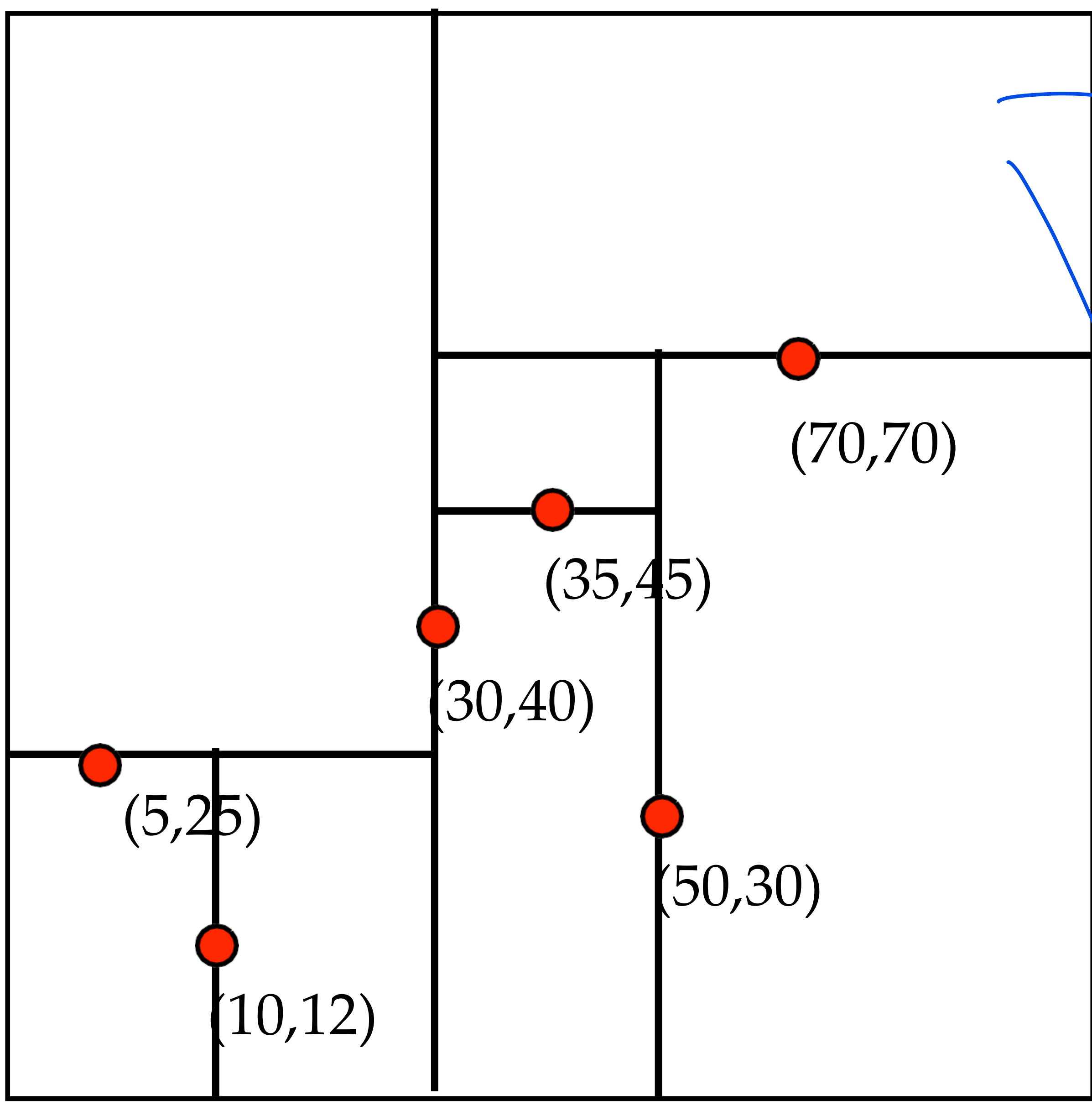
# kd-trees

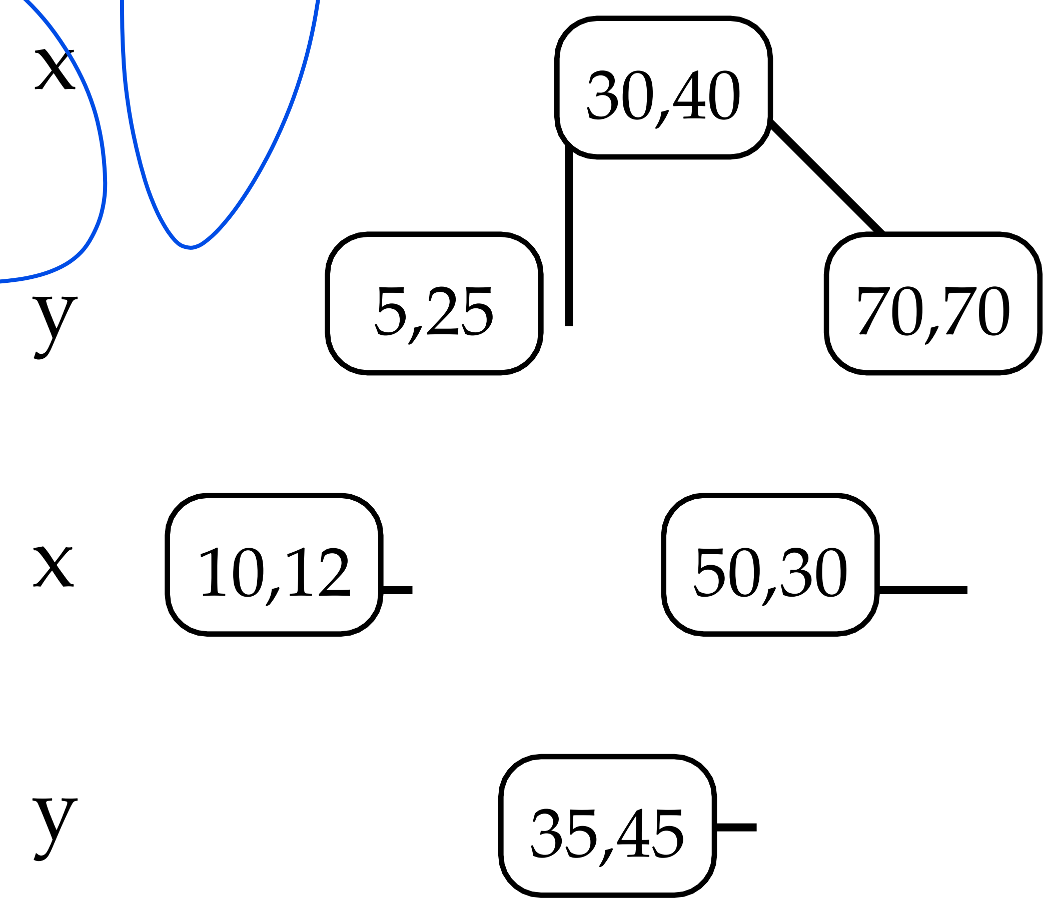
- Each level has a "cutting dimension"
- Cycle through the dimensions as you walk down the tree.
- Each node contains a point $P = (x,y)$
- To find $(x',y')$ you only compare coordinate from the cutting dimension
  - e.g. if cutting dimension is x, then you ask: is $x' < x$?



# kd-tree example

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



# Insert Code

```
insert(Point x, KDNode t, int cd) {
   if t == null
      t = new KDNode(x)
   else if (x == t.data)
      // error! duplicate
   else if (x[cd] < t.data[cd])
      t.left = insert(x, t.left, (cd+1) % DIM)
   else
      t.right = insert(x, t.right, (cd+1) % DIM)
   return t
}
```

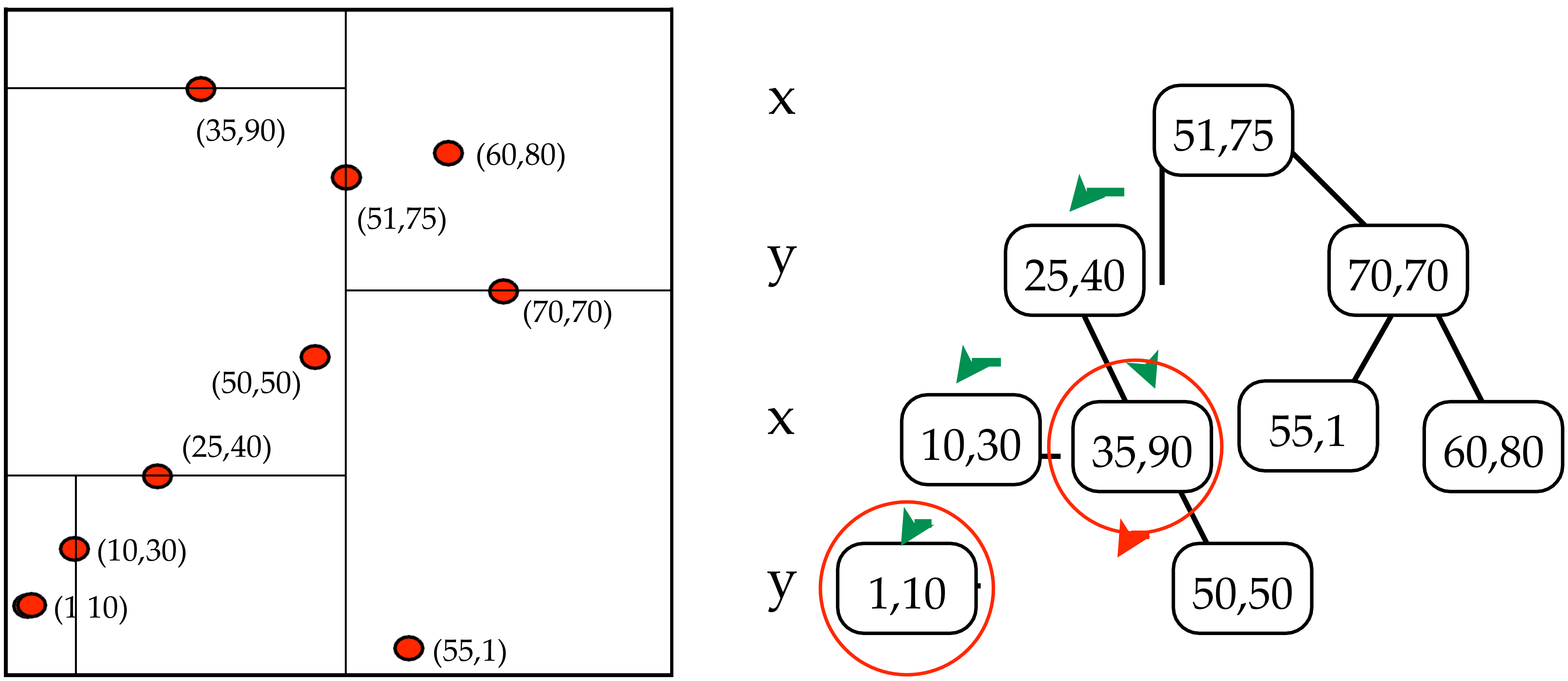
# FindMin in kd-trees

- 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)

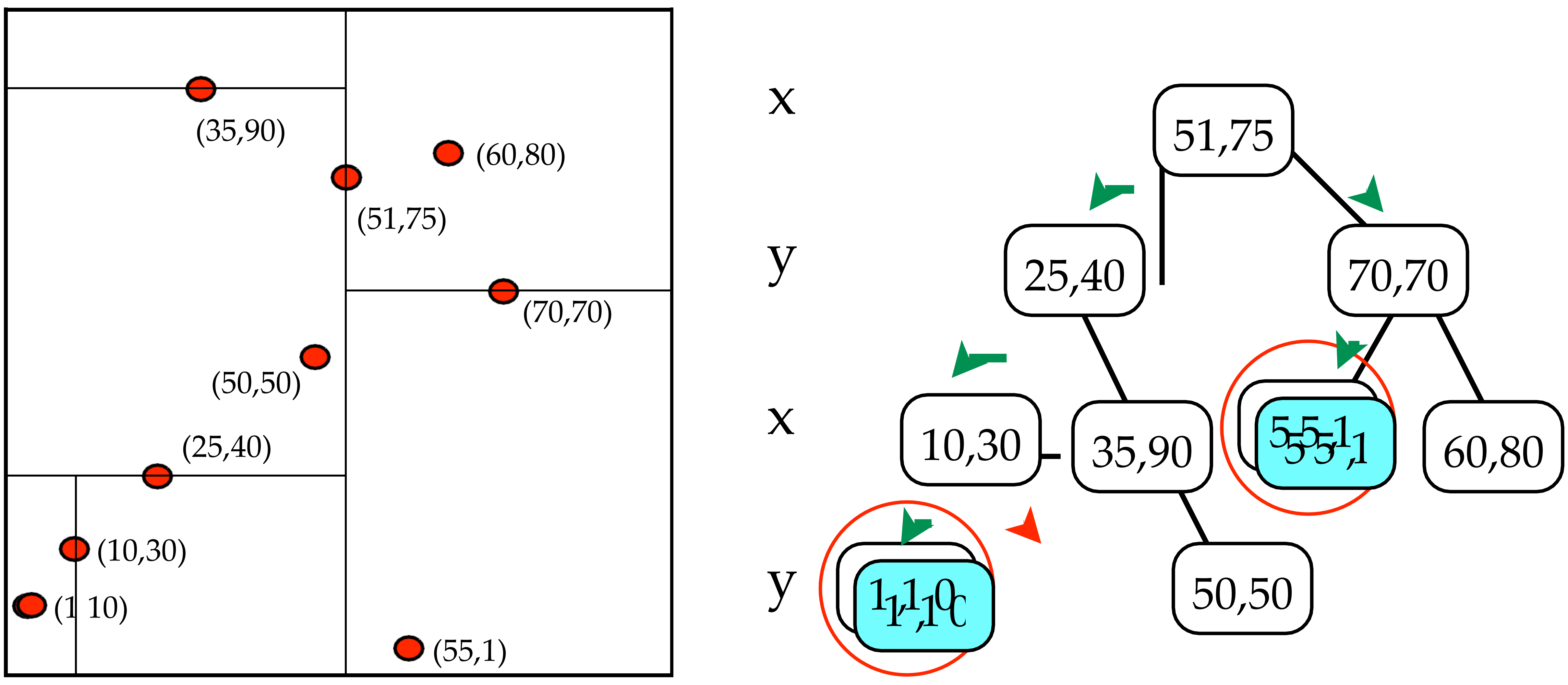
# FindMin

FindMin(x-dimension):



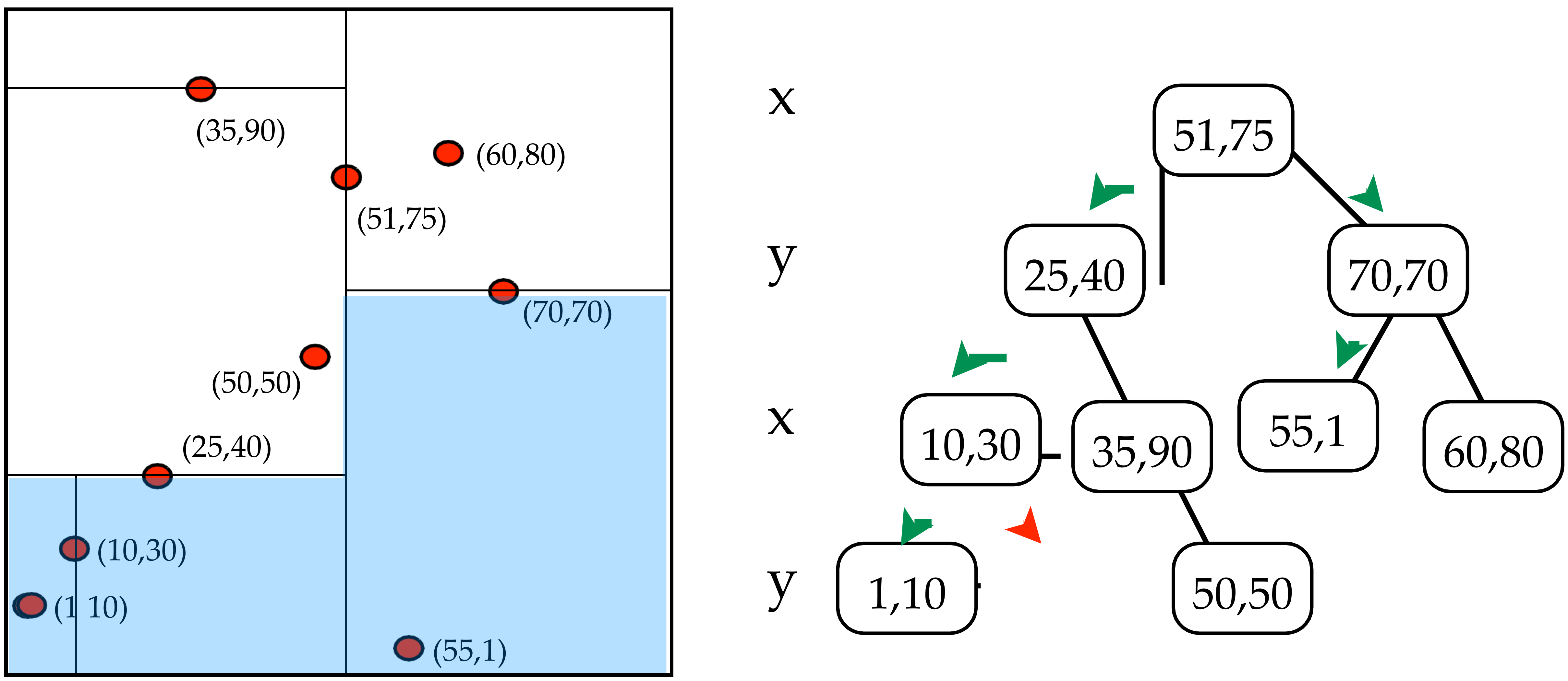
# FindMin

FindMin(y-dimension):



# FindMin

FindMin(y-dimension): space searched



# FindMin Code

```
Point findmin(Node T, int dim, int cd):
    // empty tree
    if T == NULL: return NULL

    // T splits on the dimension we’re searching
    // => only visit left subtree
    if cd == dim:
        if t.left == NULL: return t.data
        else return findmin(T.left, dim, (cd+1)%DIM)

    // T splits on a different dimension
    // => have to search both subtrees
    else:
        return minimum(
            findmin(T.left, dim, (cd+1)%DIM),
            findmin(T.right, dim, (cd+1)%DIM)
            T.data
        )
```

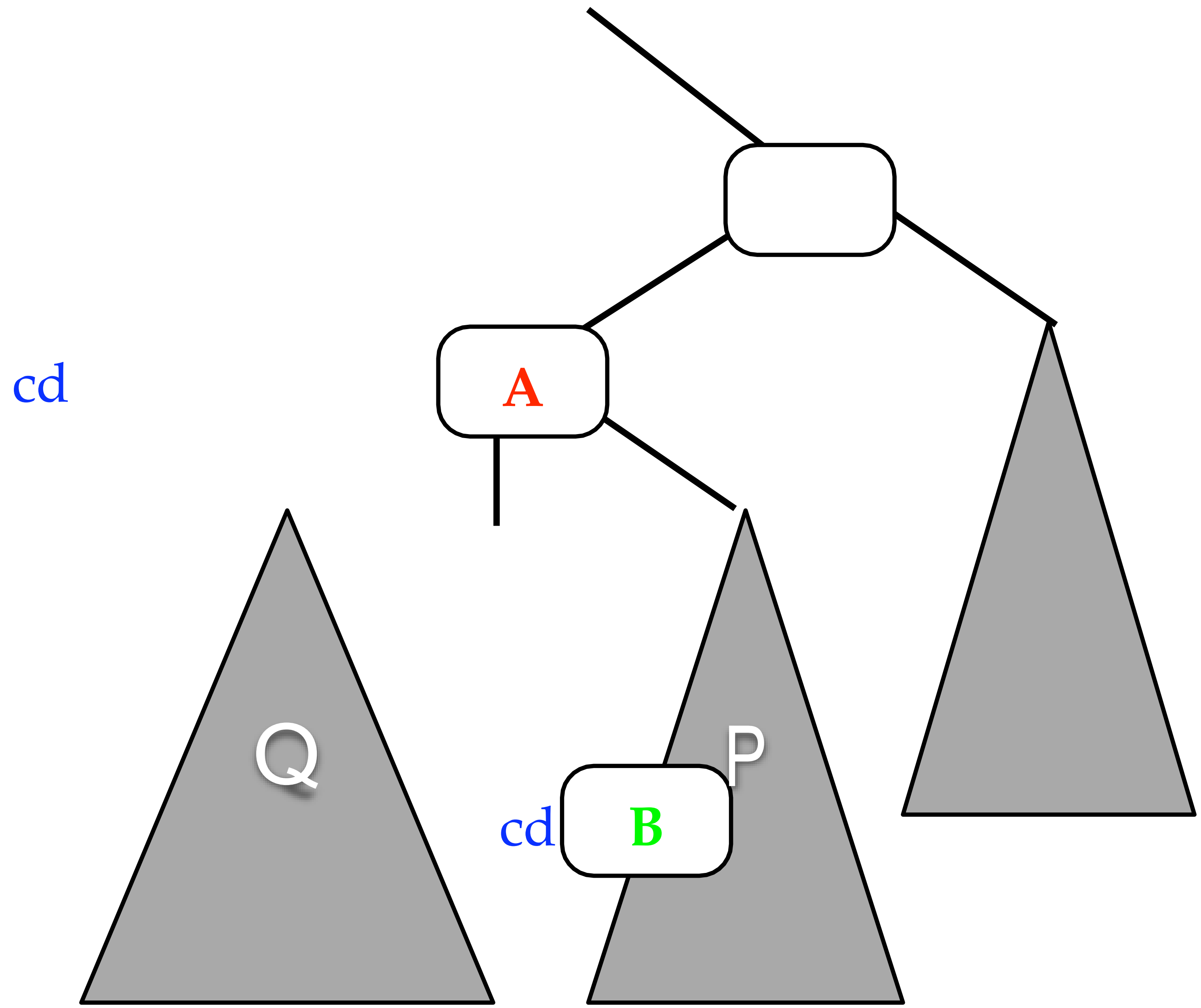
# Delete in kd-trees

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

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

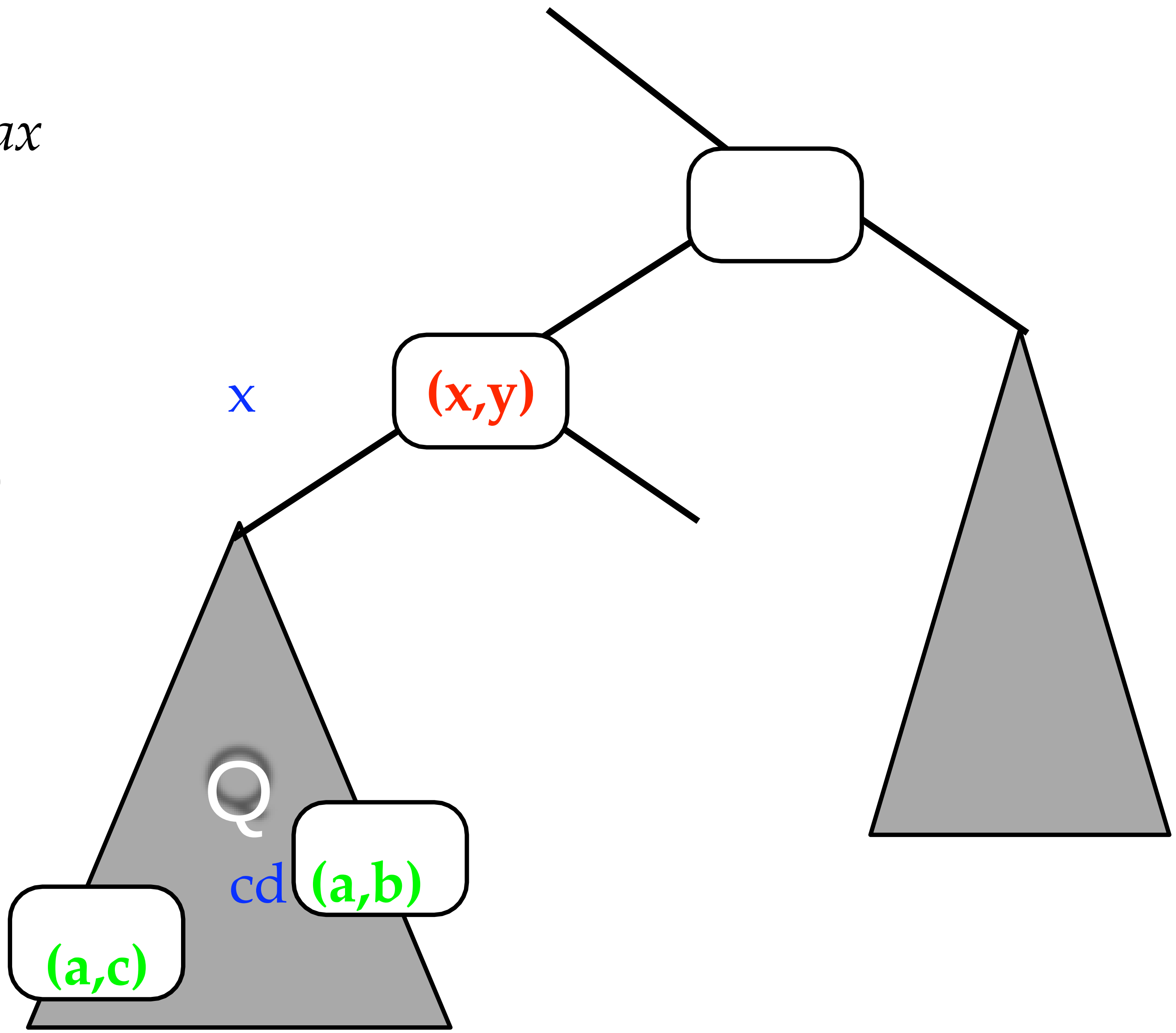
Here, we have to *findmin*(A.right, cd)

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



# Delete in kd-trees --- No Right Subtree

- What is right subtree is empty?
- Possible idea: Find the *max* in the left subtree?
  - Why might this not work?
- Suppose I findmax(T.left) and get point (a,b):

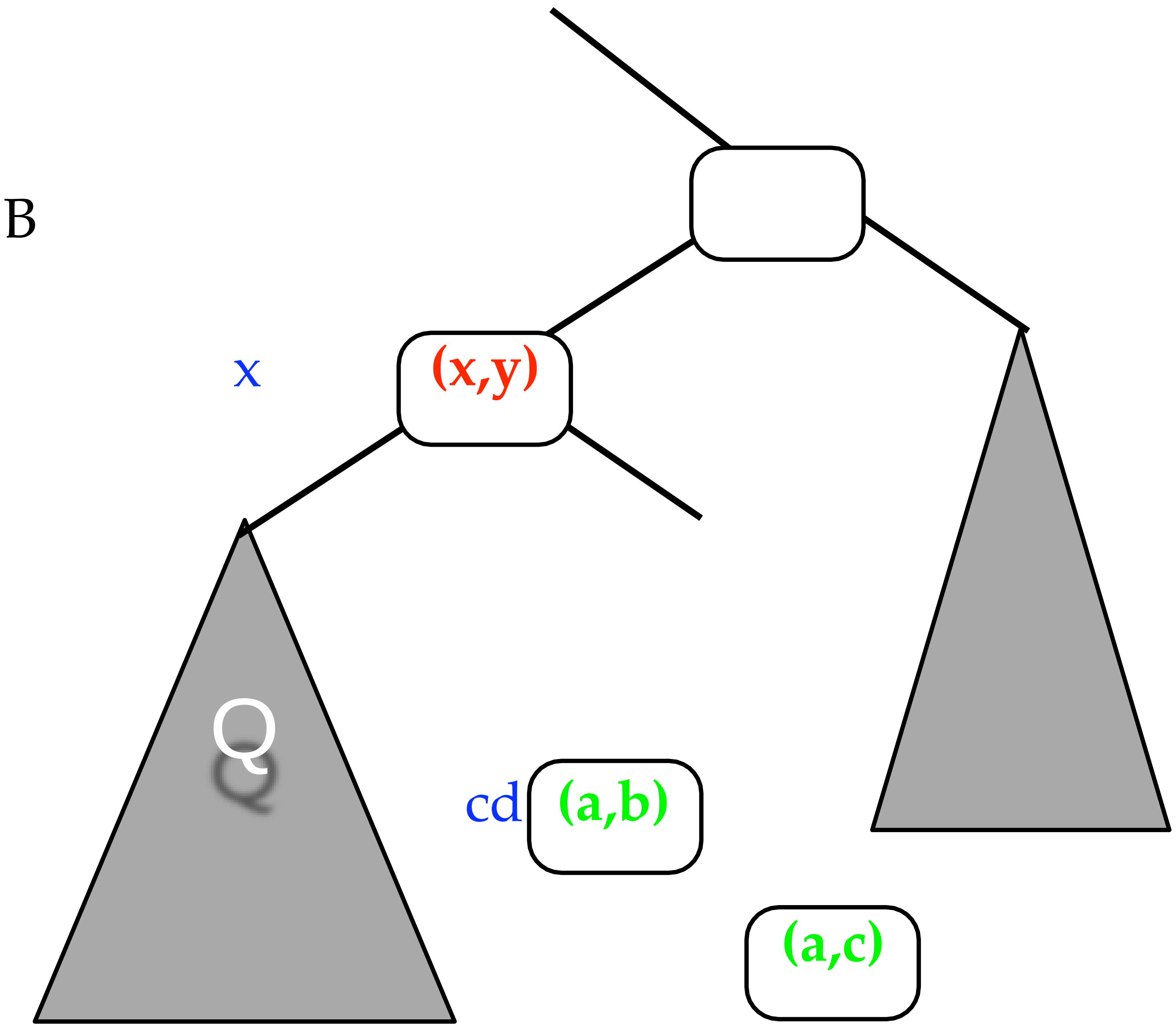



It's possible that T.left contains *another* point with x = a.

Now, our equal coordinate invariant is violated!

# No right subtree --- Solution

- Swap the subtrees of node to be deleted
- $B = \textit{findmin}(\text{T.left})$
- Replace deleted node by B

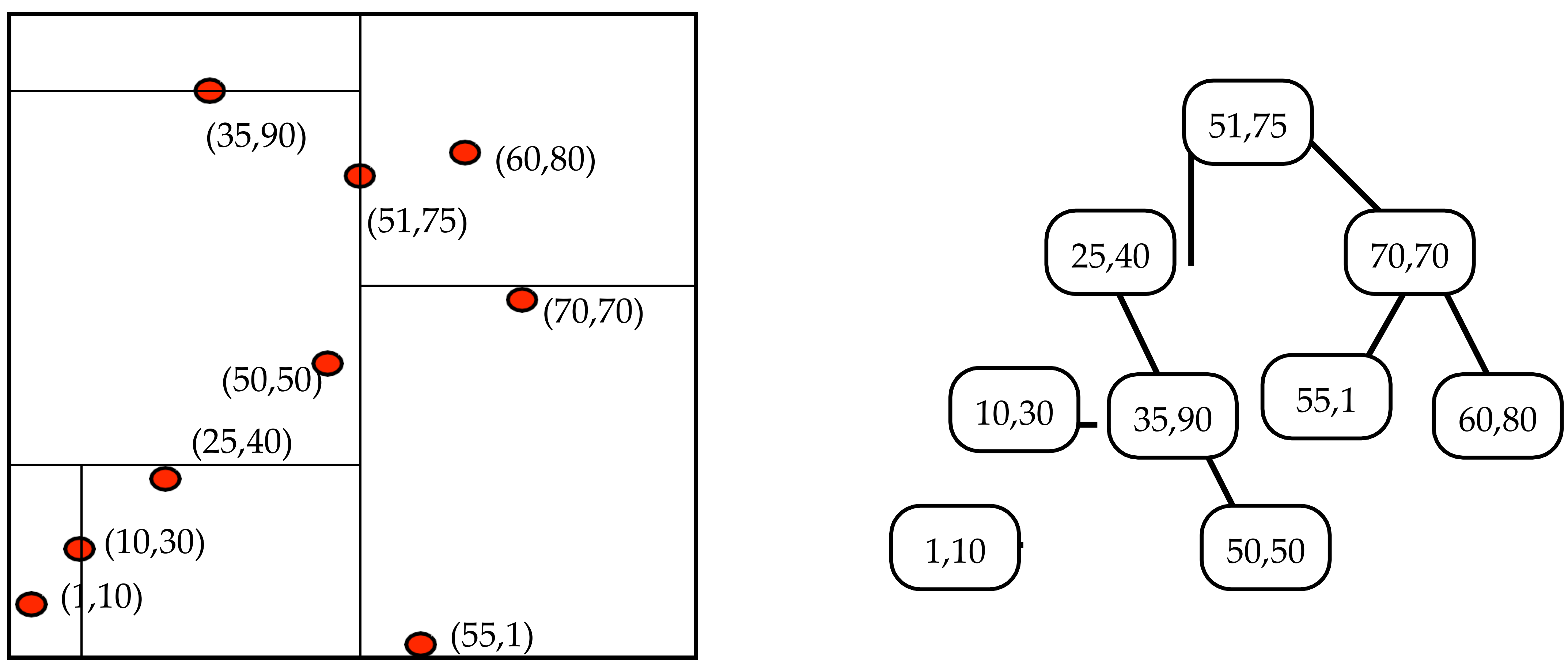



Now, if there is another point with x=a, it appears in the right subtree, where it should

```
Point delete(Point x, Node T, int cd):
   if T == NULL: error point not found!
   next_cd = (cd+1)%DIM

   // This is the point to delete:
   if x = T.data:
      // use min(cd) from right subtree:
      if t.right != NULL:
         t.data = findmin(T.right, cd, next_cd)
         t.right = delete(t.data, t.right, next_cd)
      // swap subtrees and use min(cd) from new right:
      else if T.left != NULL:
         t.data = findmin(T.left, cd, next_cd)
         t.right = delete(t.data, t.left, next_cd)
      else
         t = null    // we're a leaf: just remove

   // this is not the point, so search for it:
   else if x[cd] < t.data[cd]:
      t.left = delete(x, t.left, next_cd)
   else
      t.right = delete(x, t.right, next_cd)

   return t
```

# 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):



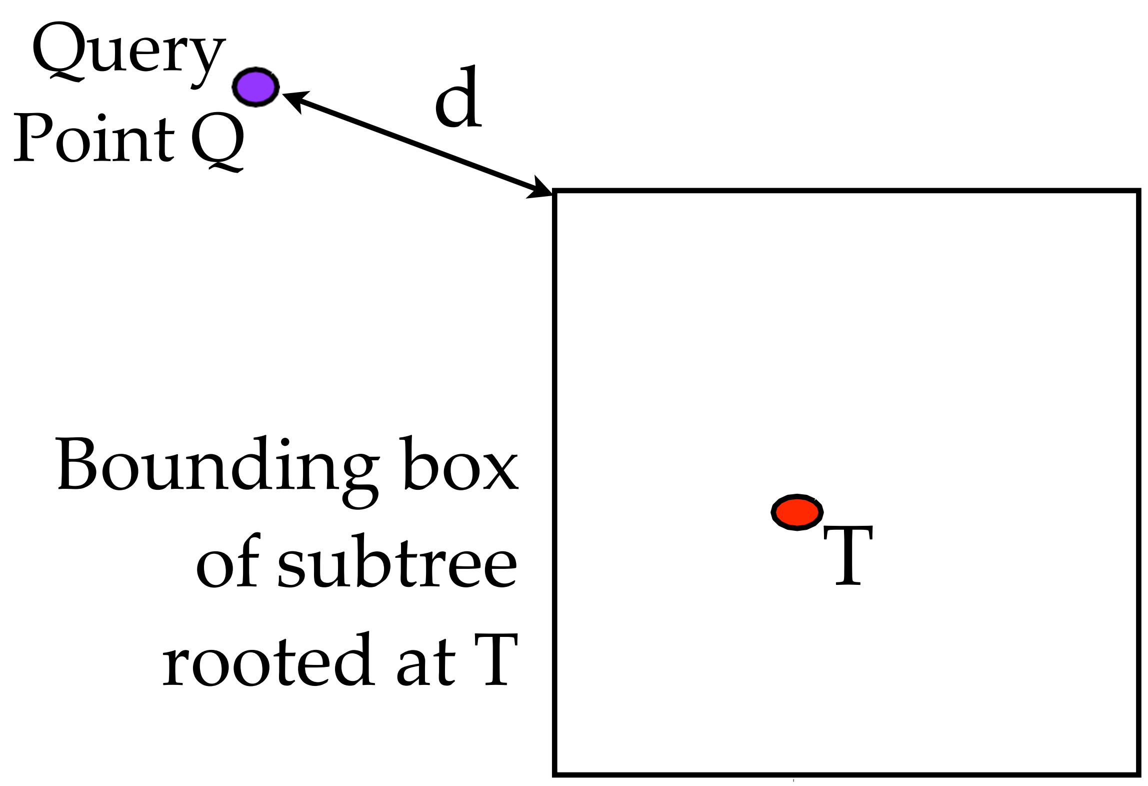
## kd-Trees Nearest Neighbor

- 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, Code

best, best_dist are global var
(can also pass into function calls)

```
def NN(Point Q, kdTree T, int cd, Rect BB):

    // if this bounding box is too far, do nothing
    if T == NULL or distance(Q, BB) > best_dist: return

    // if this point is better than the best:
    dist = distance(Q, T.data)
    if dist < best_dist:
        best = T.data
        best_dist = dist
    // visit subtrees is most promising order:
    if Q[cd] < T.data[cd]:
        NN(Q, T.left, next_cd, BB.trimLeft(cd, t.data))
        NN(Q, T.right, next_cd, BB.trimRight(cd, t.data))
    else:
        NN(Q, T.right, next_cd, BB.trimRight(cd, t.data))
        NN(Q, T.left, next_cd, BB.trimLeft(cd, t.data))
```

Following Dave Mount's Notes (page 77)

# 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.