VE281

Data Structures and Algorithms

k-d Trees

Learning Objectives:

- Know what a k-d tree is and its difference over basic binary search tree
- Know how to implement search, insertion, and removal for a k-d tree

Multidimensional Search

- Example applications:
  - find person by **last name** and **first name** (2D)
  - find location by **latitude** and **longitude** (2D)
  - find book by **author**, **title**, **year published** (3D)
  - find restaurant by **city**, **cuisine**, **popularity**, **sanitation**, **price** (5D)

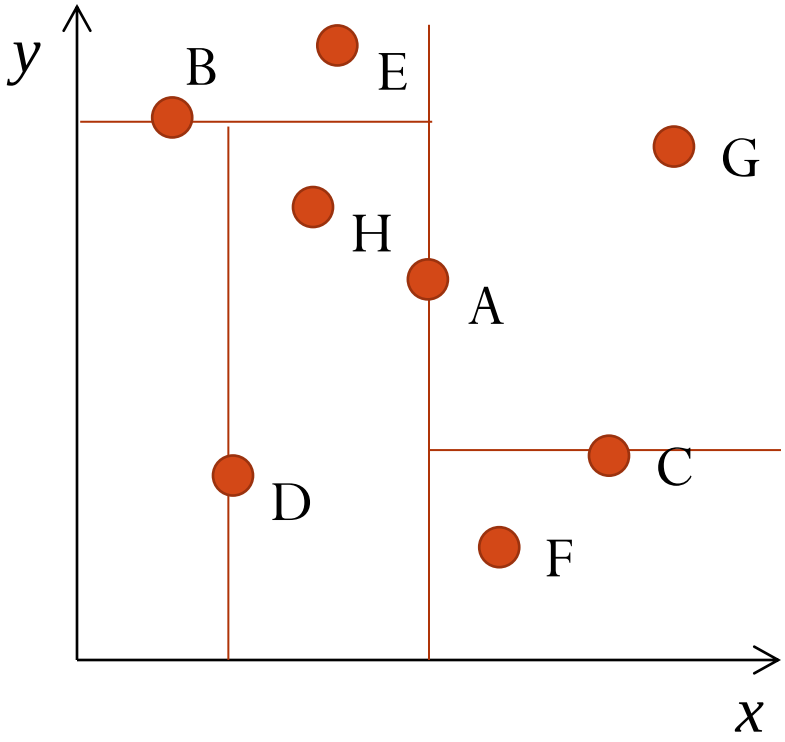
- Solution: k-d tree
  - $O(\log n)$ insert and search times

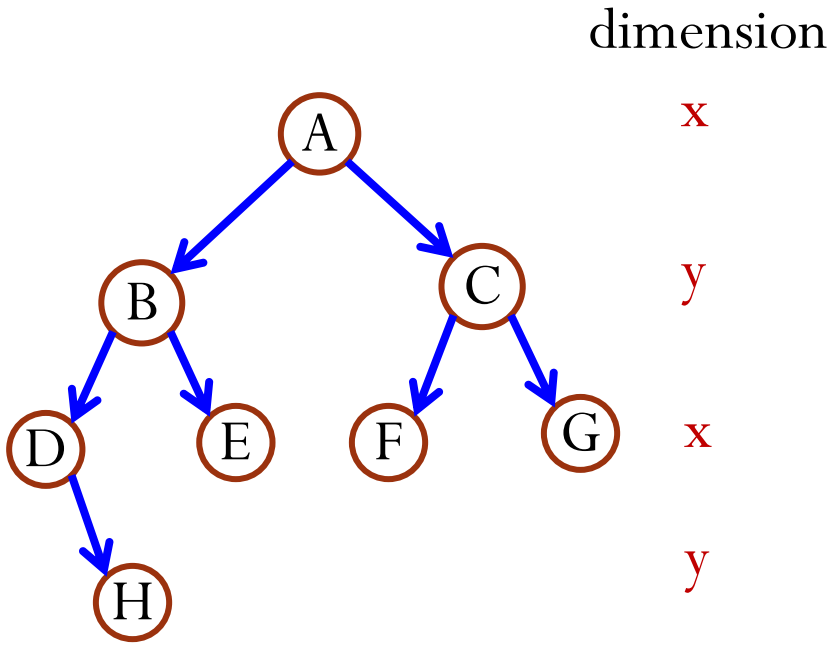
k-d Tree

- A k-d tree is a **binary search tree**
- At each level, keys from a different search dimension is used as the **discriminator**
  - Nodes on the left subtree of a node have keys with value < the node's key value **along this dimension**
  - Nodes on the right subtree have keys with value ≥ the node's key value **along this dimension**
- We **cycle** through the dimensions as we go down the tree
  - For example, given keys consisting of x- and y-coordinates, level 0 discriminates by the x-coordinate, level 1 by the y-coordinate, level 2 again by the x-coordinate, etc.

Example

- k-d tree for points in a 2-D plane





k-d Tree Insert

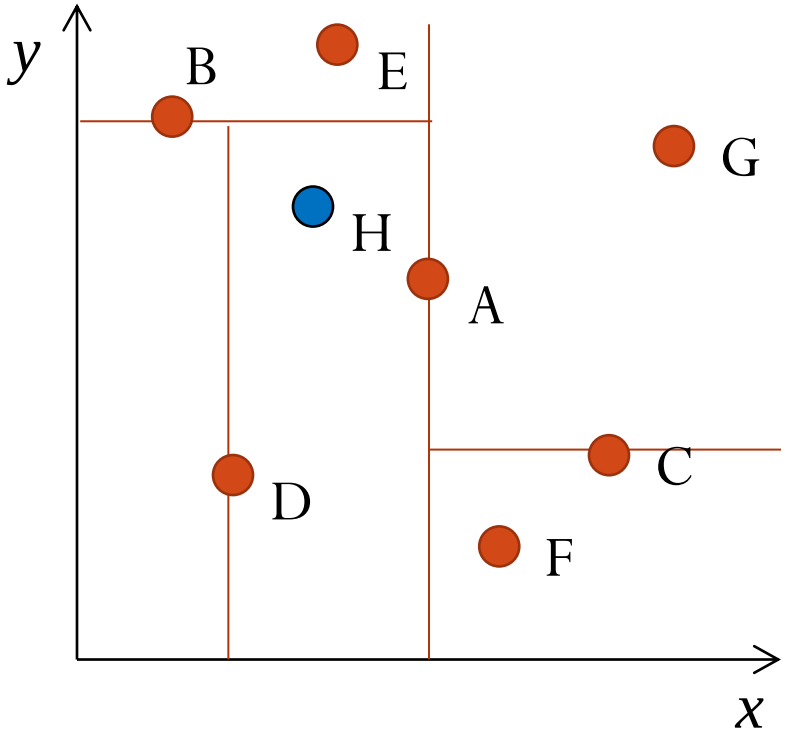
- If new item's key is equal to the root's key, return;
- If new item has a key smaller than that of root's along the dimension of the current level, recursive call on left subtree
- Else, recursive call on the right subtree
- In recursive call, cyclically increment the dimension

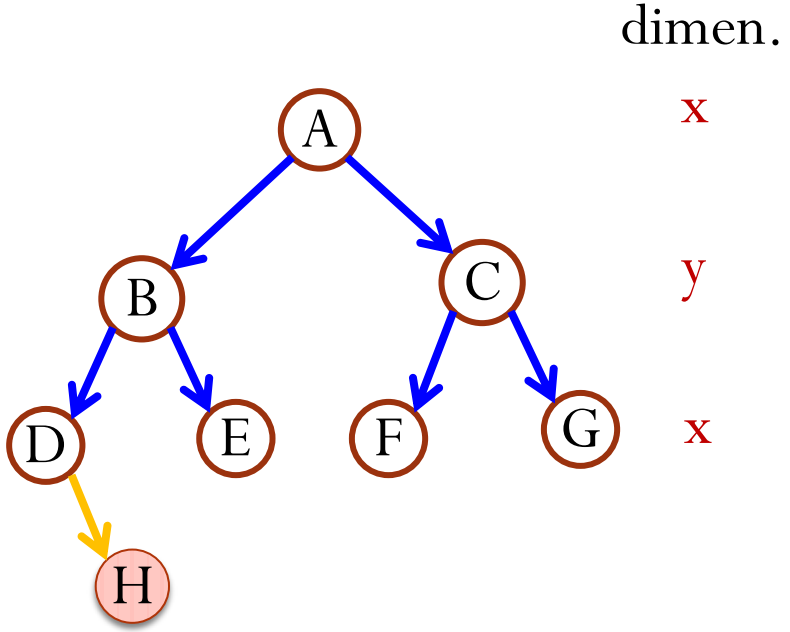
```
void insert(node *&root, Item item, int dim) {
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key == root->item.key) // equal in all
    return;                      // dimensions
  if(item.key[dim] < root->item.key[dim])
    insert(root->left, item, (dim+1)%numDim);
  else
    insert(root->right, item, (dim+1)%numDim);
}
```

dim refers to the dimension of the root

Insert Example

- Insert H
- Initial function call: insert(A, H, 0) // 0 indicates dimension x





k-d Tree Search

- Search works similarly to insert
  - In recursive call, cyclically increment the dimension

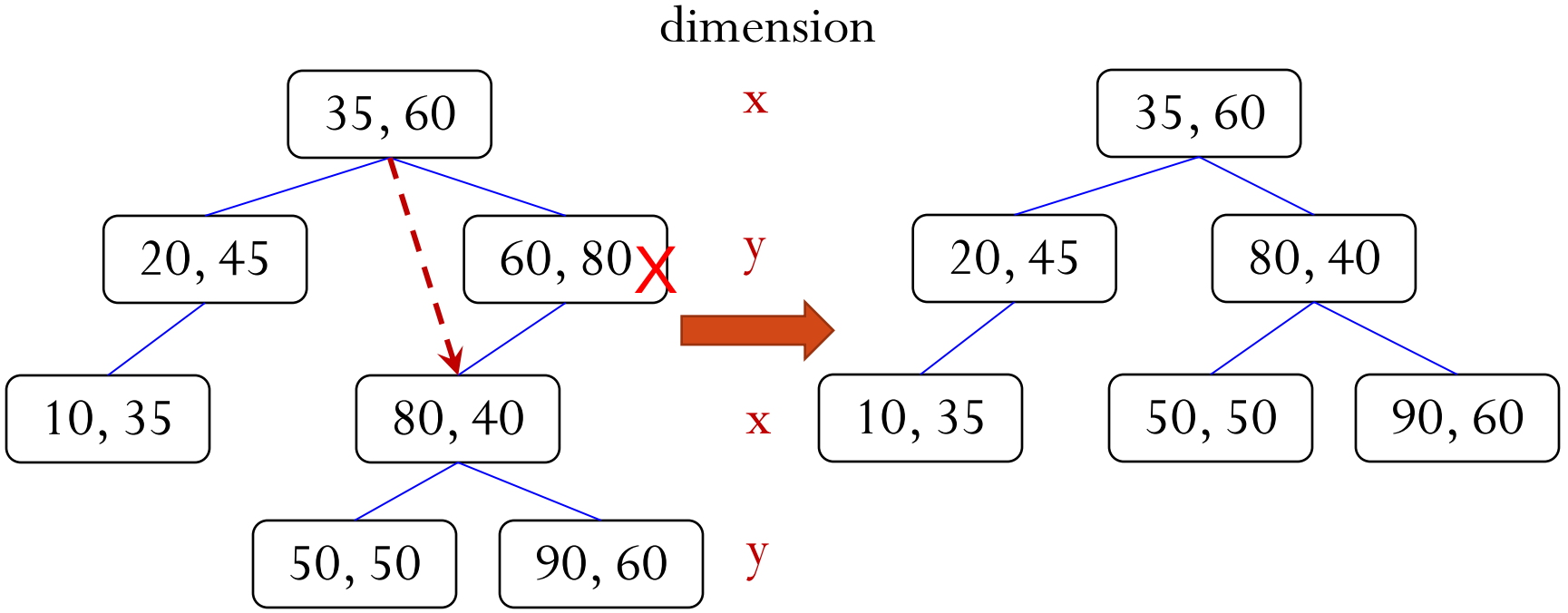
```
node *search(node *root, Key k, int dim) {
  if(root == NULL) return NULL;
  if(k == root->item.key)
    return root;
  if(k[dim] < root->item.key[dim])
    return search(root->left, k, (dim+1)%numDim);
  else
    return search(root->right, k, (dim+1)%numDim);
}
```

Time complexities of insert and search are all $O(\log n)$

k-d Tree Remove

- If the node is a leaf, simply remove it (e.g., remove (50,50))
- If the node has only one child, can we do the same thing as BST (i.e., connect the node's parent to the node's child)?
  - Consider remove (60, 80)

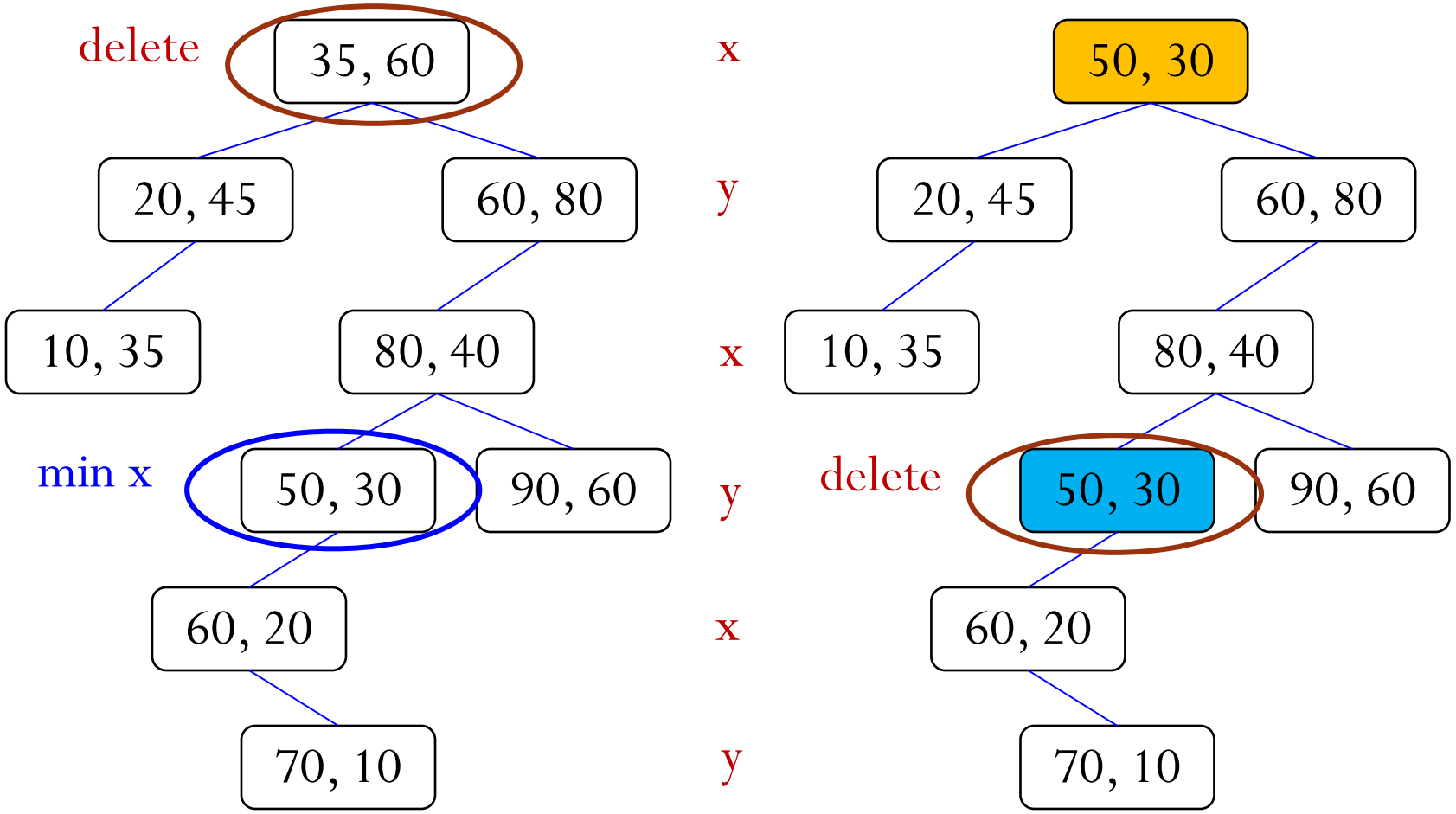
Answer: No!



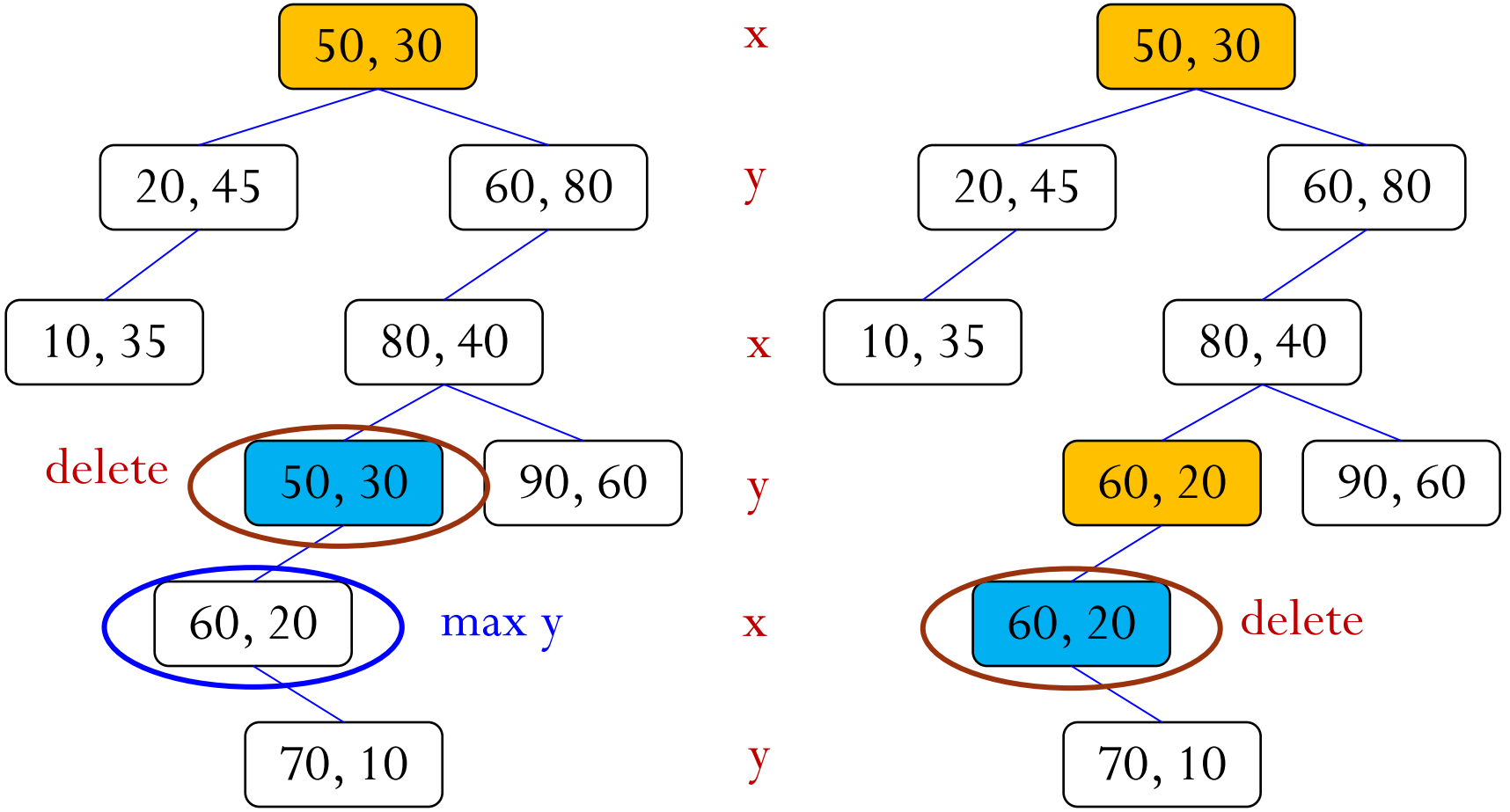
k-d Tree Removal of Non-leaf Node

- If the node R to be removed has right subtree, find the node M in right subtree with the **minimum** value of the current dimension
  - Replace the value of R with the value of M
  - Recurse on M until a leaf is reached. Then remove the leaf
- Else, find the node M in left subtree with the **maximum** value of the current dimension. Then replace and recurse

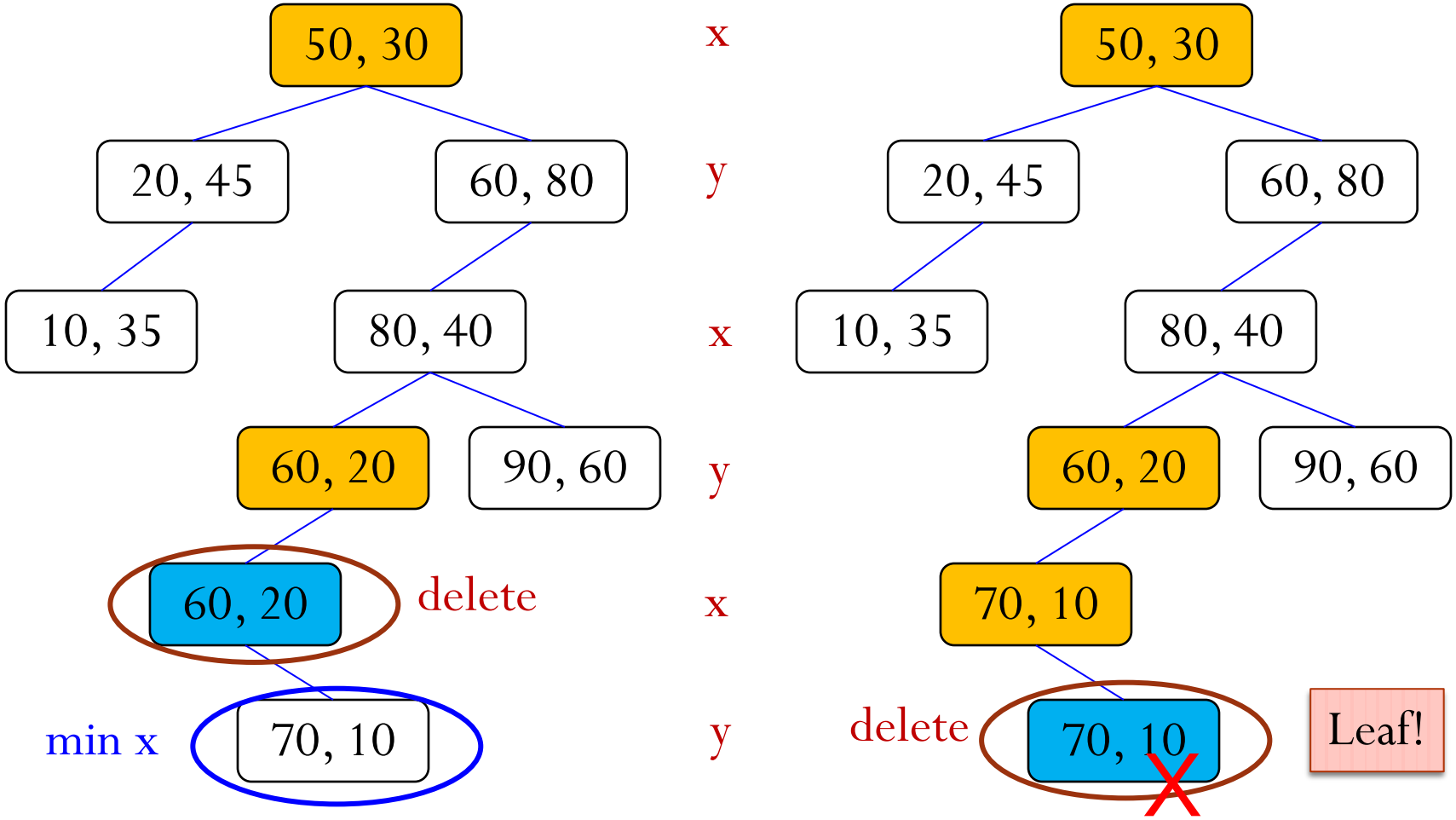
k-d Tree Removal Example



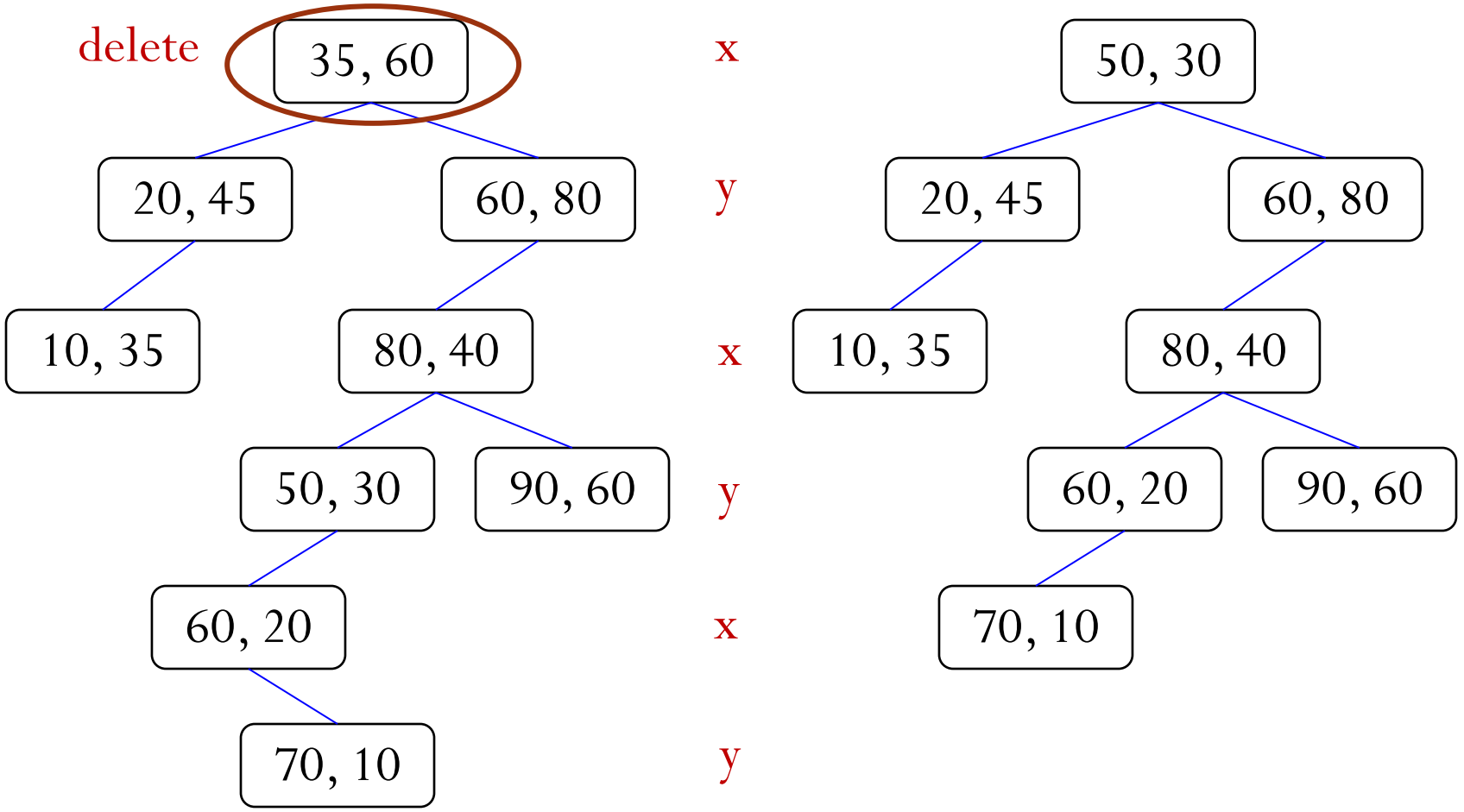
k-d Tree Removal Example



k-d Tree Removal Example

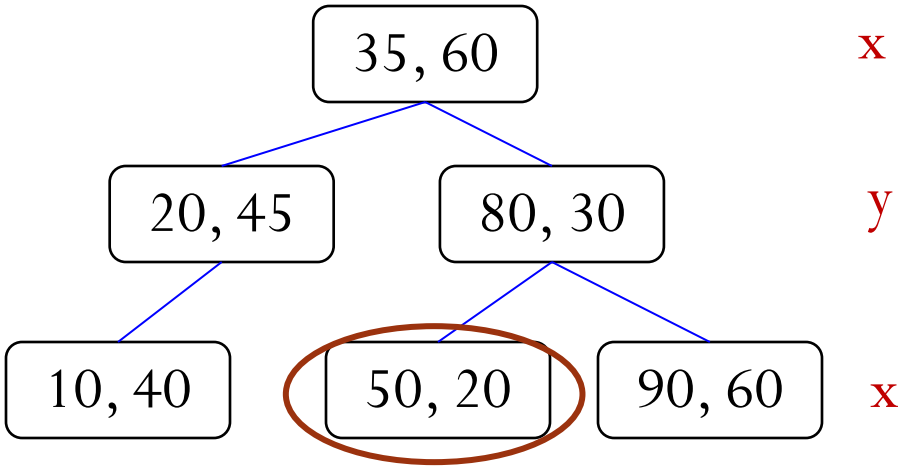


k-d Tree Removal Example: Summary



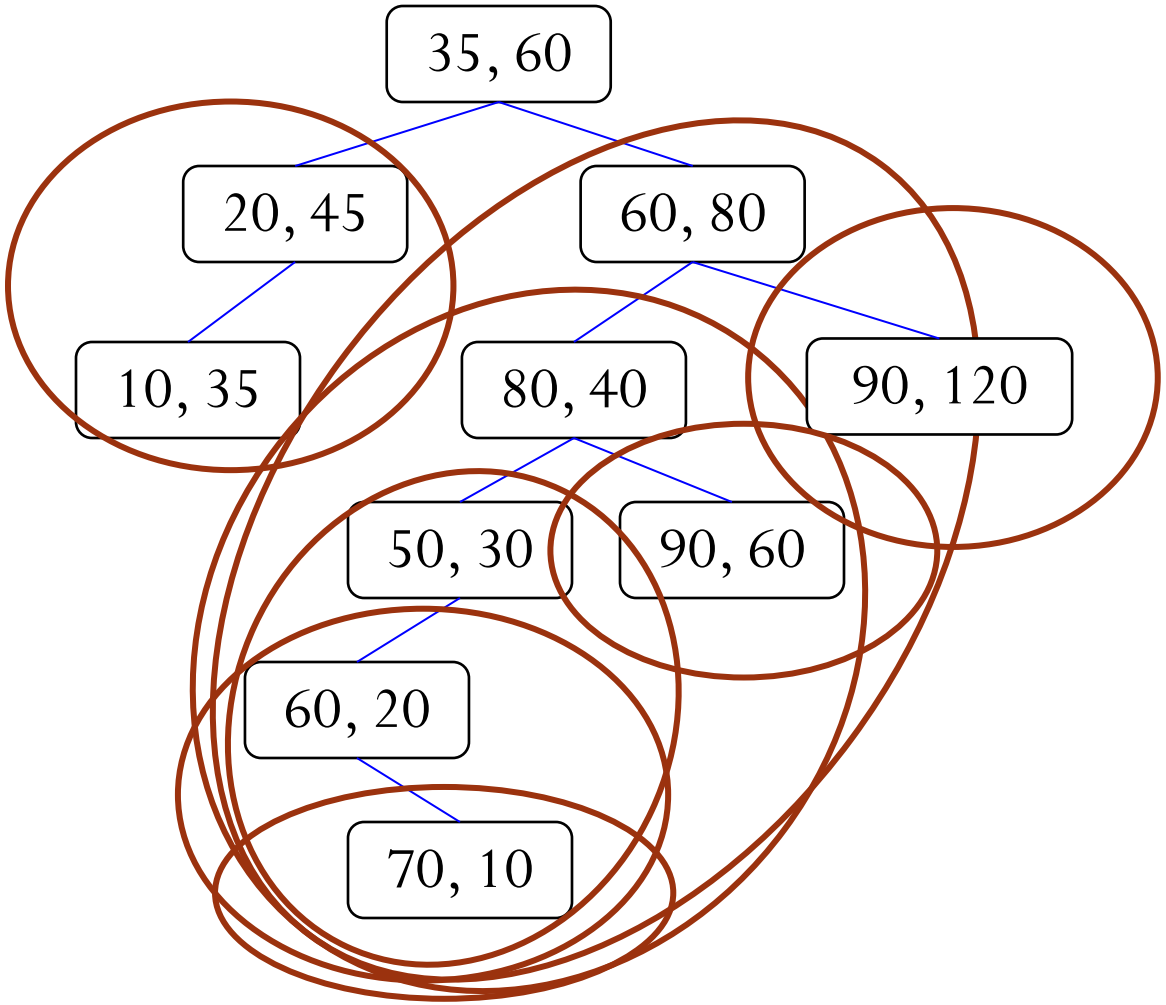
Find Minimum Value in a Dimension

- Different from the basic BST, because it may not be the left-most descendent.



Find the node with minimum value in dimension y

Find Min-Y



Find Minimum Value in a Dimension

```
node *findMin(node *root, int dimCmp, int dim) {
// dimCmp: dimension for comparison
  if(!root) return NULL;
  node *min =
    findMin(root->left, dimCmp, (dim+1)%numDim);
  if(dimCmp != dim) {
    rightMin =
     findMin(root->right, dimCmp, (dim+1)%numDim);
    min = minNode(min, rightMin, dimCmp);
  }
  return minNode(min, root, dimCmp);
}
```

- **minNode** takes two nodes and a dimension as input, and returns the node with the smaller value in that dimension

Multidimensional Range Search

- Example
  - Buy ticket for travel between certain dates and certain times
  - Look for apartments within certain price range, certain districts, and number of bedrooms
  - Find all restaurants near you

- k-d tree supports efficient range search, which is similar to that of basic BST but more complex!

k-d Tree Range Search

```
void rangeSearch(node *root, int dim,
  Key searchRange[][2], Key treeRange[][2],
  List results)
```

- Cycle through the dimensions as we go down the level
- **`searchRange[][2]`** holds two values (min, max) per dimension
  - Define a hyper-cube
  - min of dimension **`j`** at **`searchRange[j][0]`**, max at **`searchRange[j][1]`**
- **`treeRange[][2]`** holds lower bound and upper bound per dimension for the tree rooted at **`root`**.
  - Need to be updated as we go down the levels
  - Need to check if a search range overlaps a subtree range