VE281

Data Structures and Algorithms

k-d Trees; Tries

Outline

- k-d Trees
- Tries

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)

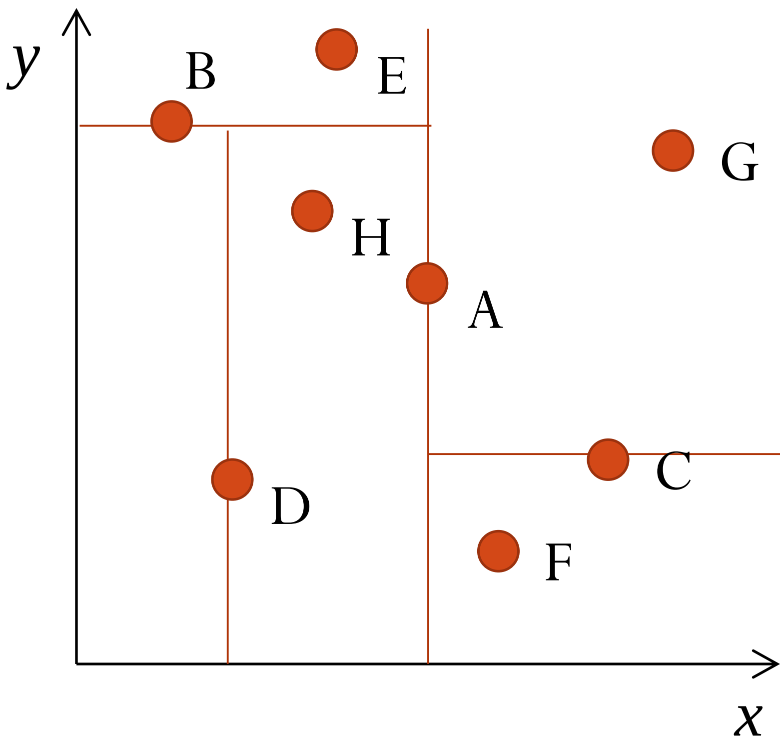
- Solution: k-d tree
  - $O(\log n)$ insert and search times

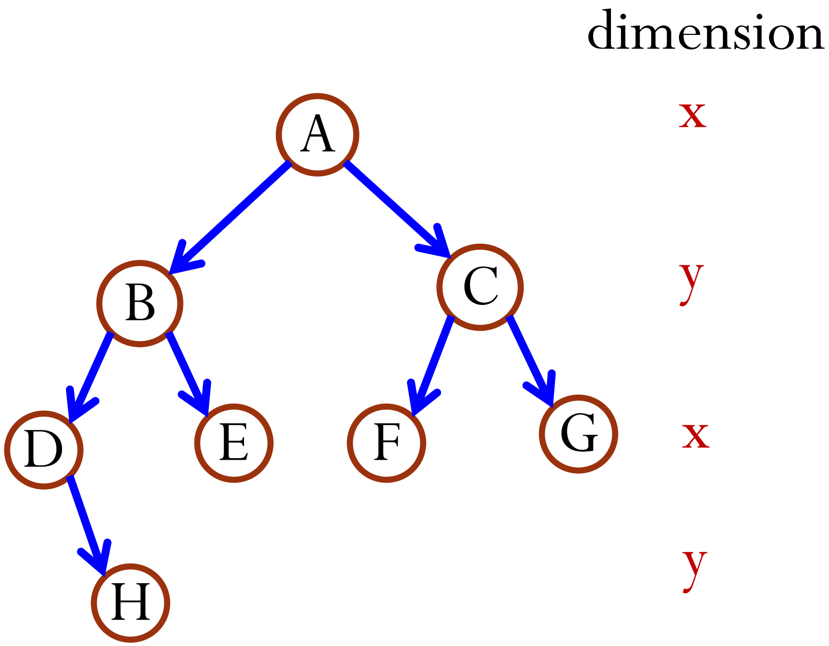
k-d Tree

- 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

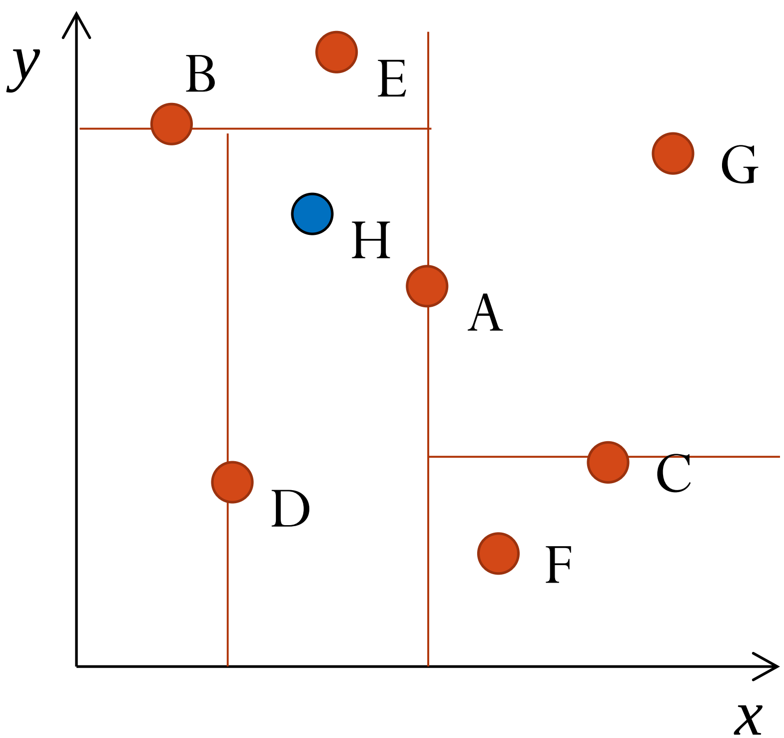
- If new item's key is equal to the root's key, return;
- If new item has a key smaller than that of root's along the dimension of the current level, recursive call on left subtree
- Else, recursive call on the right subtree
- In recursive call, cyclically increment the dimension

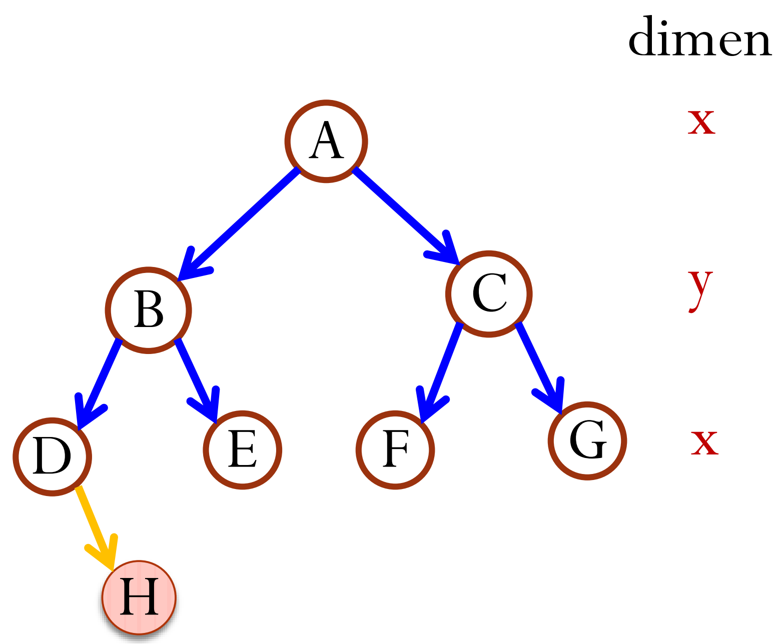
```
void insert(node *&root, Item item, int dim) {
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key == root->item.key) // equal in all
    return;                      // dimensions
  if(item.key[dim] < root->item.key[dim])
    insert(root->left, item, (dim+1)%numDim);
 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

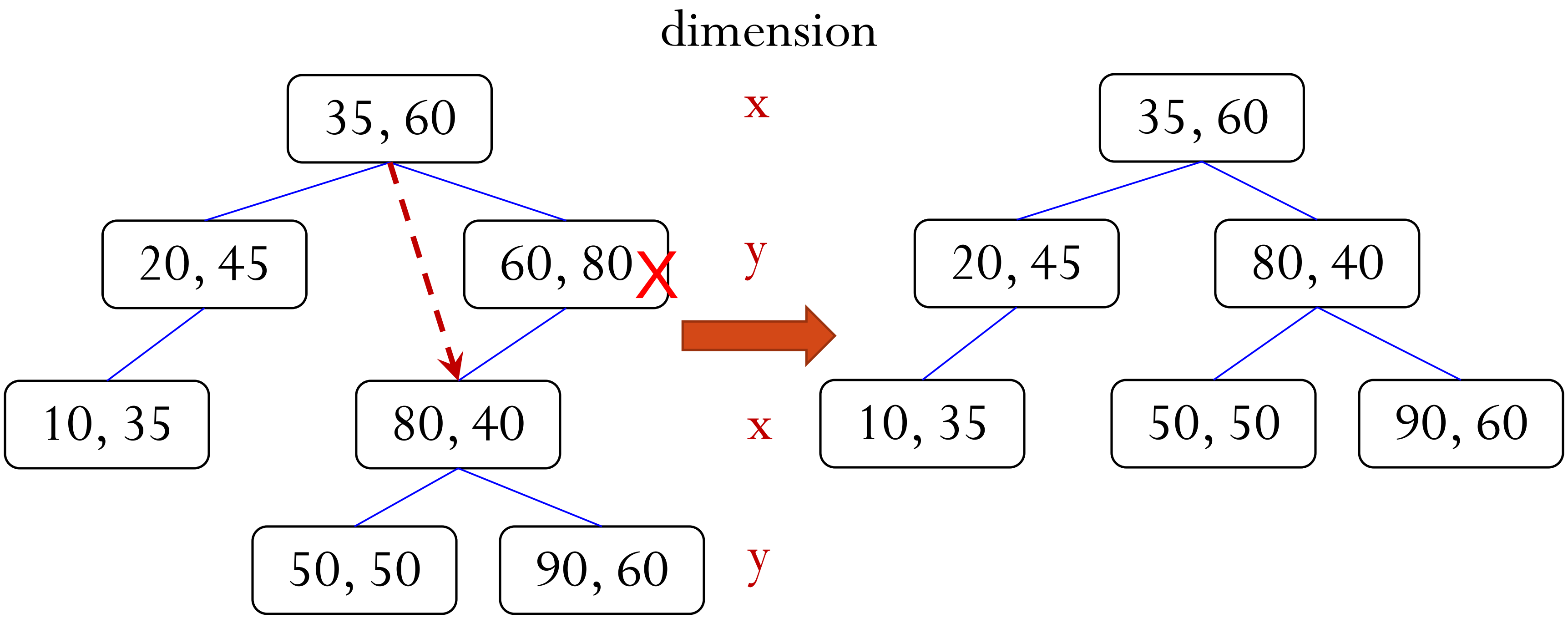
```
node *search(node *root, Key k, int dim) {
  if(root == NULL) return NULL;
  if(k == root->item.key)
    return root;
  if(k[dim] < root->item.key[dim])
    return search(root->left, k, (dim+1)%numDim);
  else
    return search(root->right, k, (dim+1)%numDim);
}
```

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)

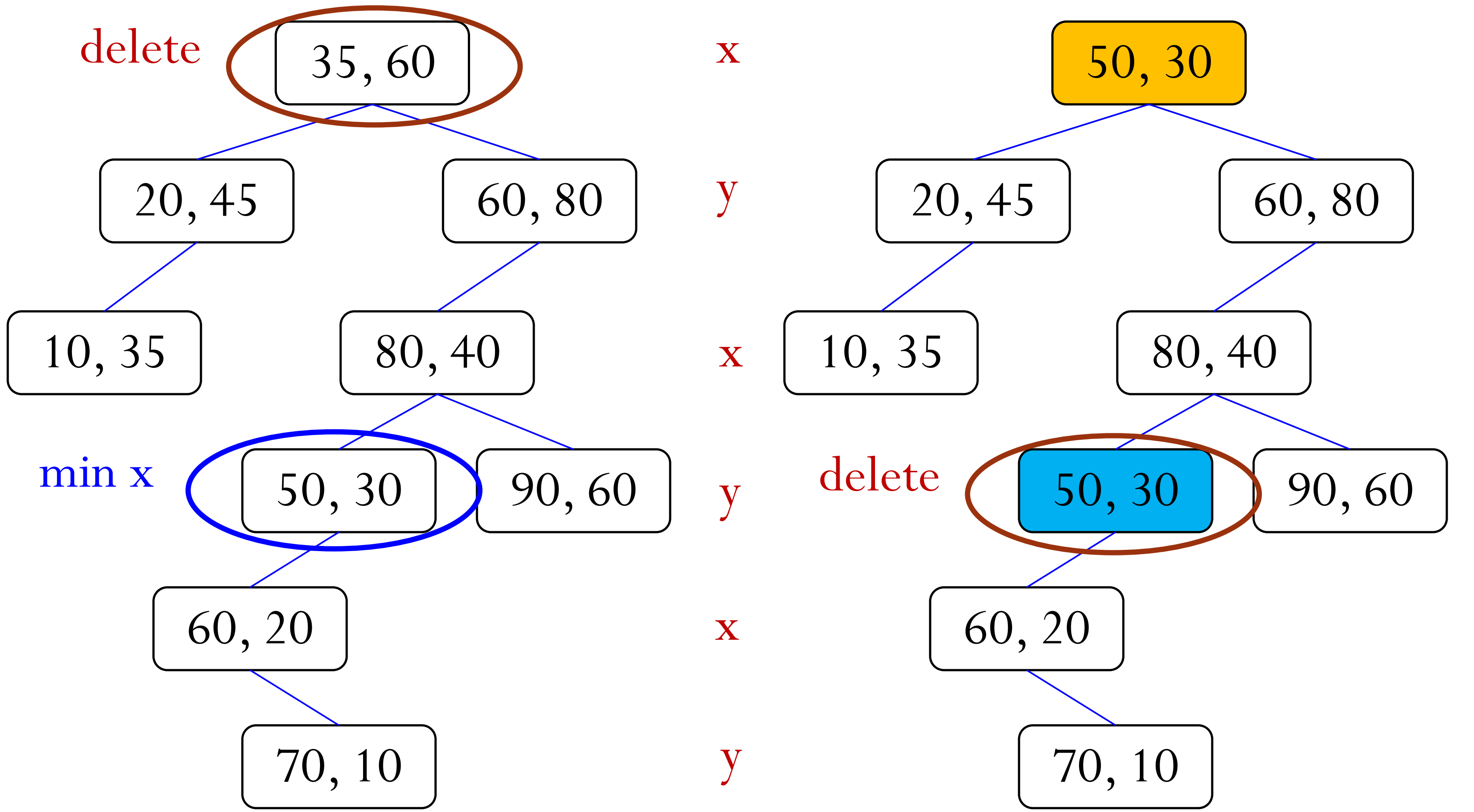
Answer: No!



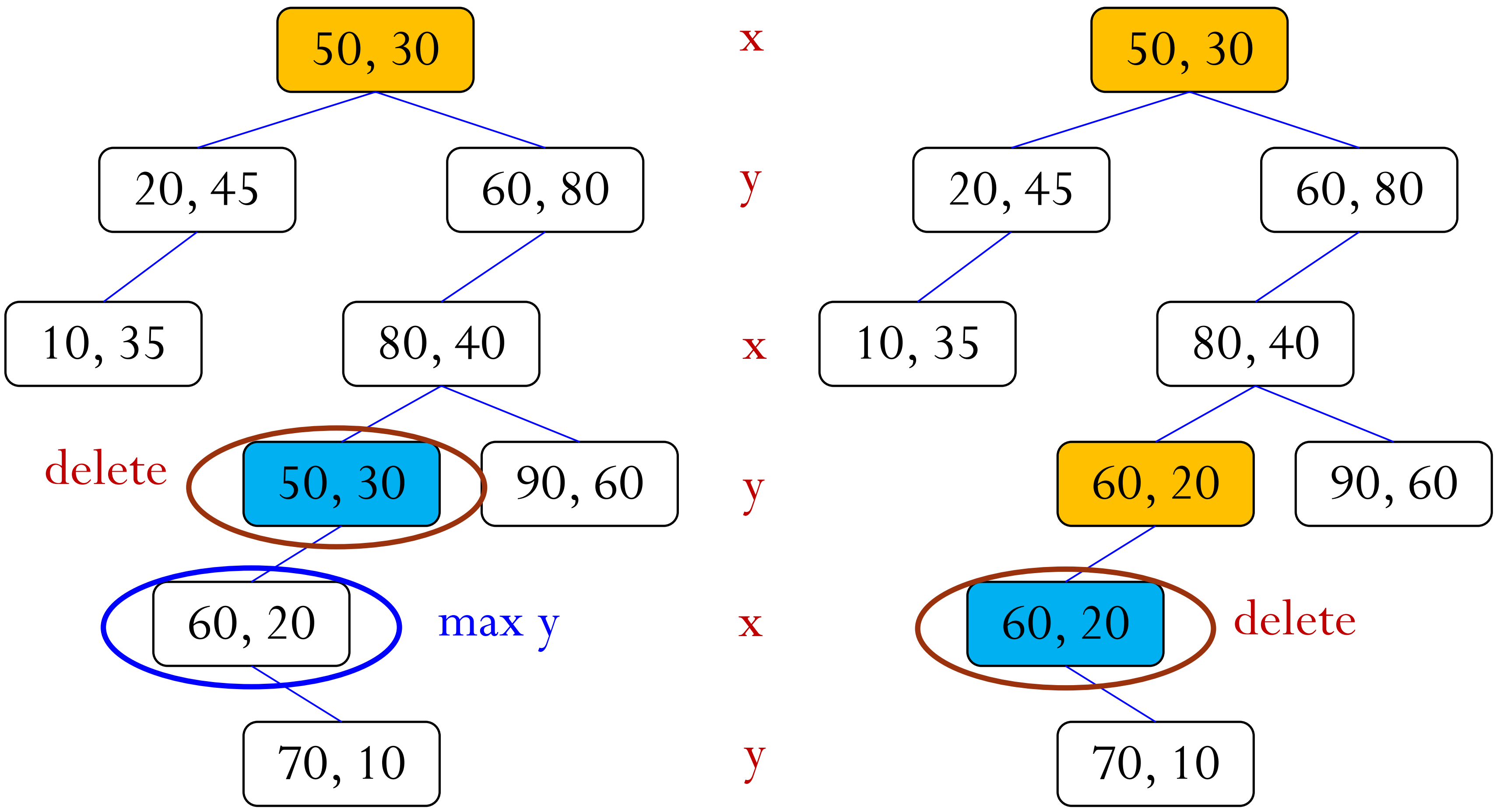
k-d Tree Removal of Non-leaf Node

- 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

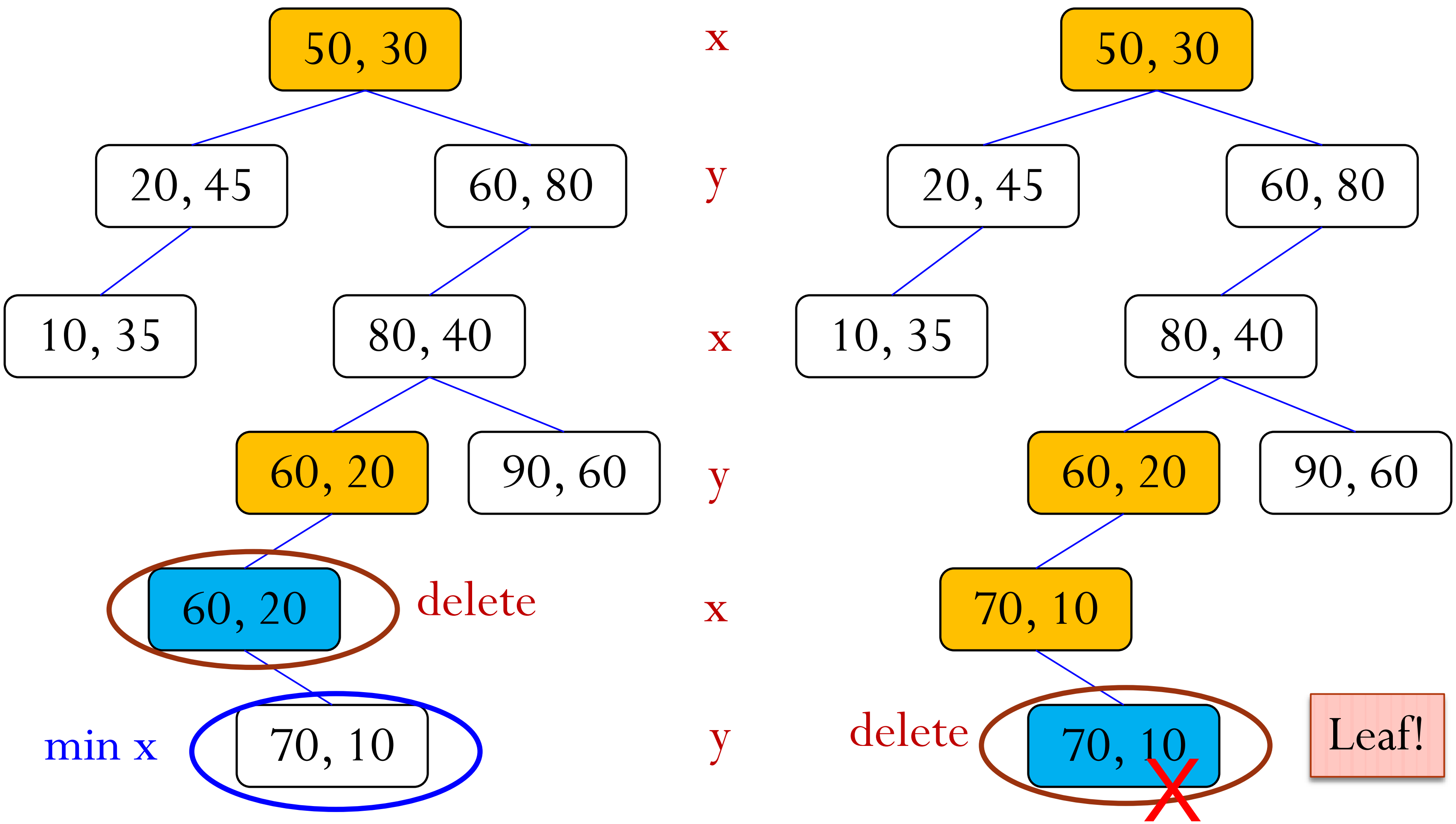
k-d Tree Removal Example



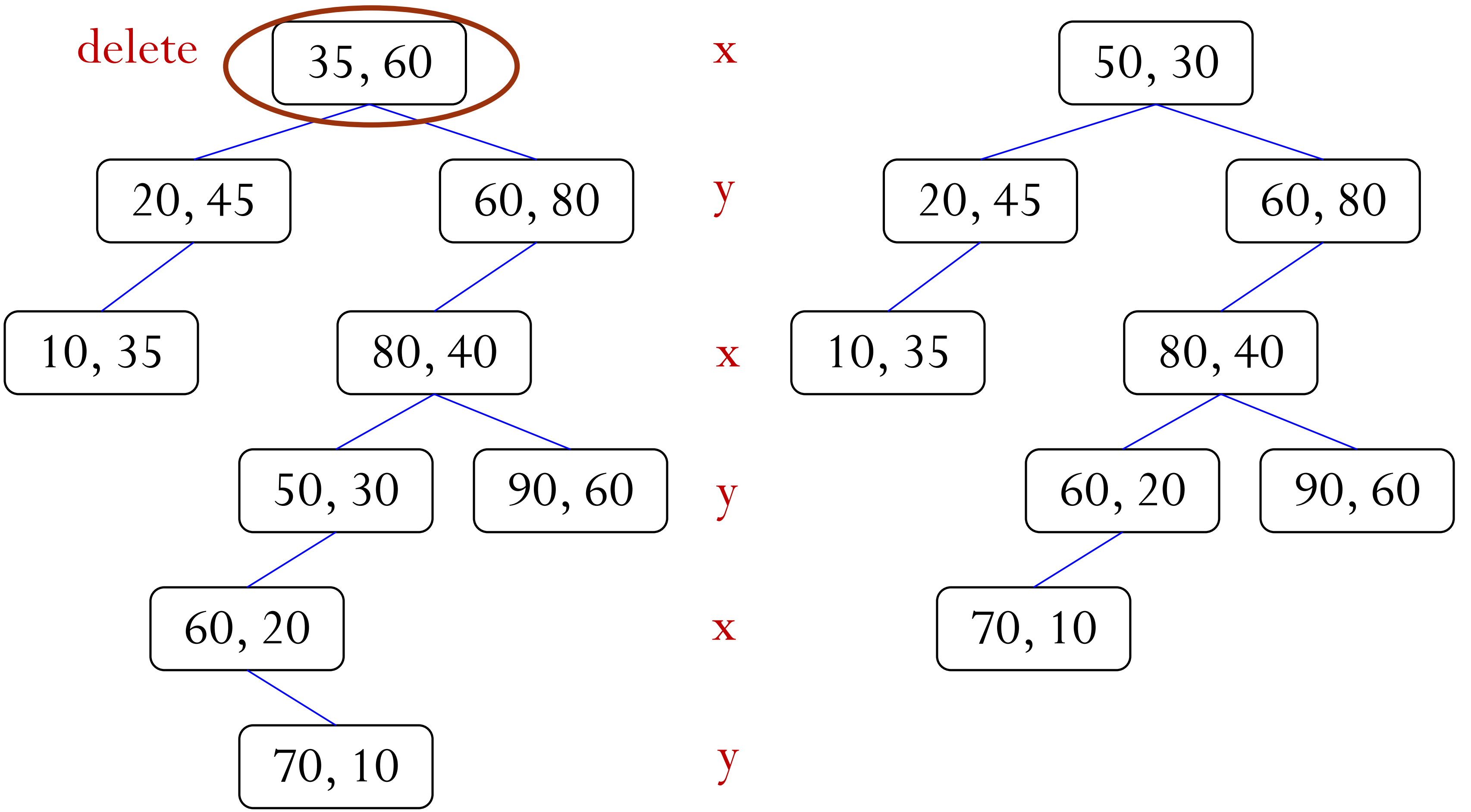
k-d Tree Removal Example



k-d Tree Removal Example

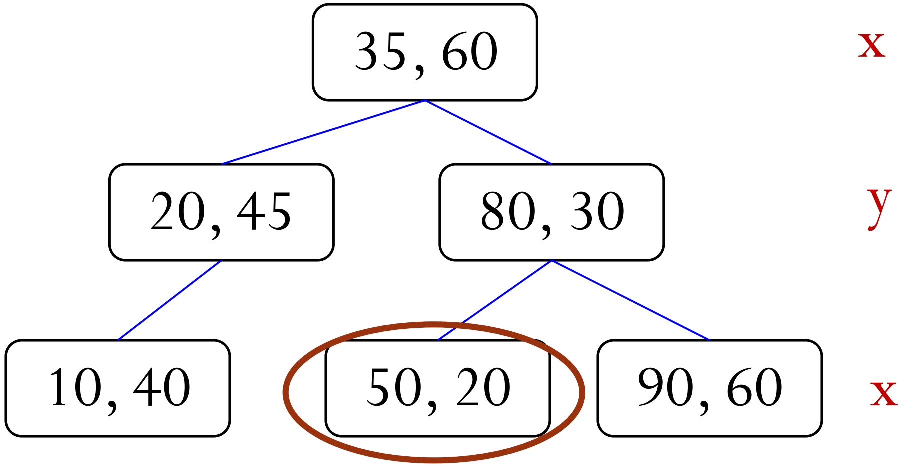


k-d Tree Removal Example: Summary



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 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

k-d Tree Range Search

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

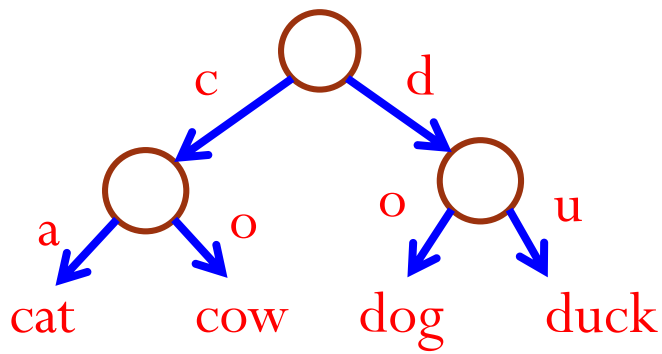
- Cycle through the dimensions as we go down the level
- **`searchRange[]`** holds two values (min, max) per dimension
  - Define a hyper-cube
  - min of dimension **`j`** at **`searchRange[2*j]`**, max at **`searchRange[2*j+1]`**
- **`treeRange[]`** holds lower bound and upper bound per dimension for 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

Outline

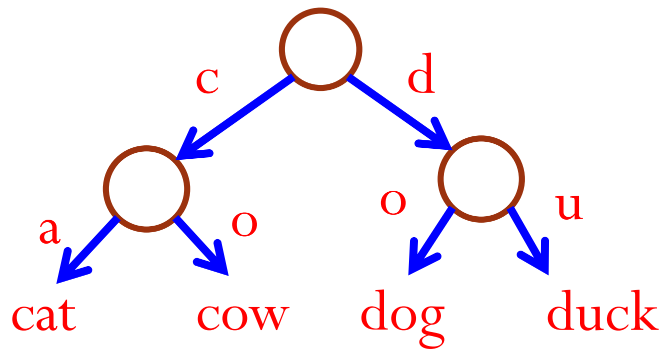
- k-d Trees
- Tries

Trie

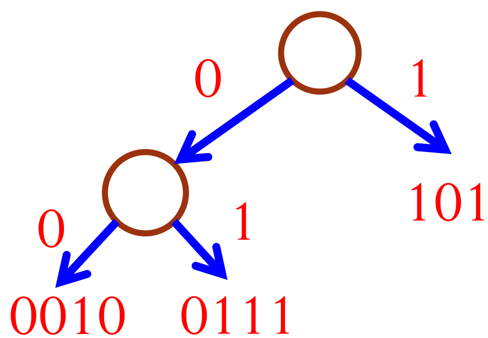
- The word "trie" comes from retrieval.
  - To distinguish with "tree", it is pronounced as "try".
- A trie is a tree that uses parts of the key, as opposed to the whole key, to perform search.
- Data records are only stored in **leaf** nodes. Internal nodes serve as **placeholders** to direct the search process.



Trie

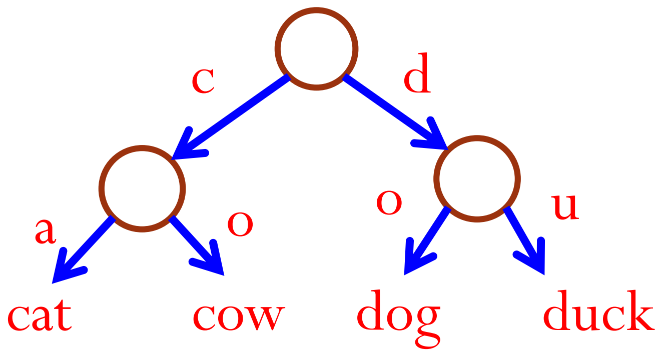


- Trie usually is used to store a set of strings from an alphabet.
  - The alphabet is in the general sense, not necessarily the English alphabet.
- For example, {0, 1} is an alphabet for binary codes {0010, 0111, 101}. We can store these three codes using a trie.



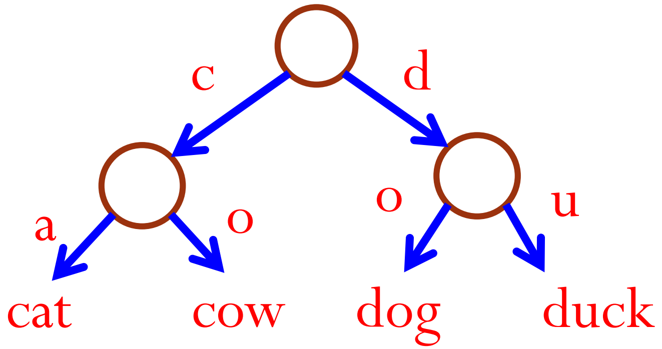
Trie

- Each edge of the trie is labeled with symbols from the alphabet.
- Labels of edges on the path from the root to any leaf in the trie forms a **prefix** of a string in that leaf.
  - Trie is also called **prefix-tree**.



Trie

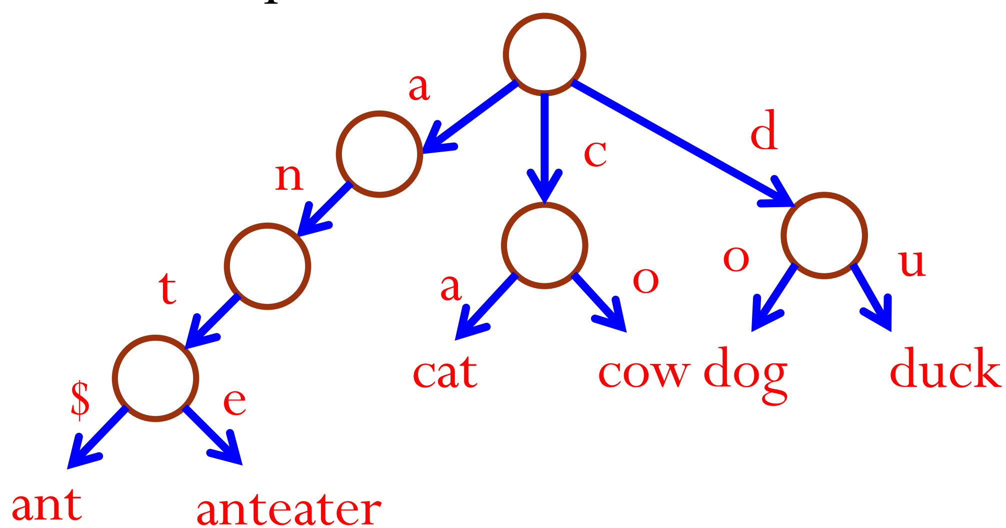
- The most significant symbol in a string determines the branch direction at the root.
- Each internal node is a "**branch**" point.
- As long as there is only one key in a branch, we do not need any further internal node below that branch; we can put the word directly as the leaf of that branch.



Trie

Implementation Issue

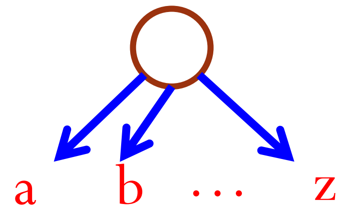
- Sometimes, a string in the set is exactly a **prefix** of another string.
  - For example, "ant" is a prefix of "anteater".
  - How can we make "ant" as a leaf in the trie?
- We add a symbol to the alphabet to indicate the end of a string. For example, use "$" to indicate the end.



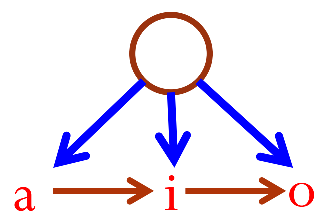
Trie

Implementation Issue

- We can keep an array of pointers in a node, which corresponds to **all** possible symbols in the alphabet.



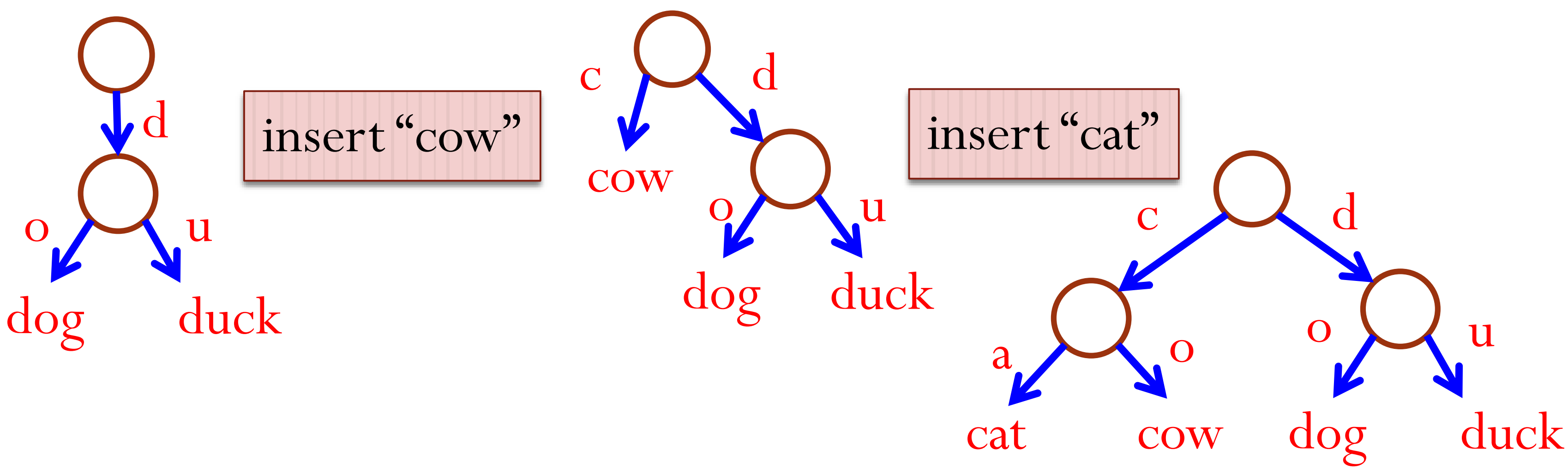
- However, most internal nodes have branches to only a small fraction of the possible symbols in the alphabet.
  - An alternate implementation is to store a linked list of pointers to the child nodes.



Trie

Insertion

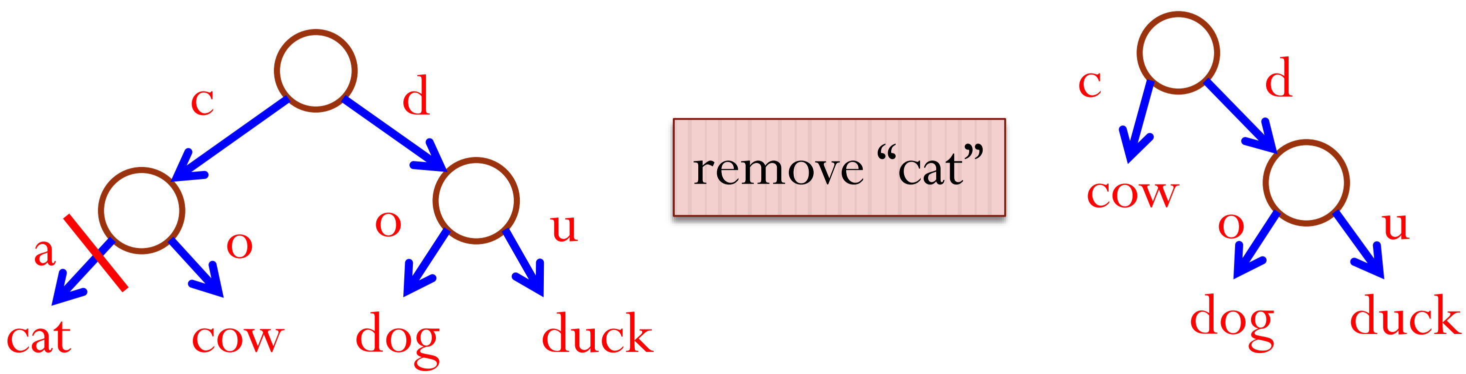
- Follow the search path, starting from the root.
- If a new branch is needed, add it.
- When the search leads to a leaf, a conflict occurs. We need to branch.
  - Use the next symbol in the key
  - The originally-unique word must be moved to lower level



Trie

Removal

- The key to be removed is always at the leaf.
- After deleting the key, if the parent of that key now has only one child C, remove the parent node and move key C one level up.
  - If key C is the only child of its new parent, repeat the above procedure again.



Time Complexity of Trie

- In the worst case, inserting or finding a key that consists of k symbols is $O(k)$.
  - This does not depend on the number of keys N.
  - Comparison: stroring 32 integers in the range [0, 127] using a trie versus using a BST. What are heights in the **worst case**?
    - BST: 32; Trie: 7
- Sometimes we can access records even **faster**.
  - A key is stored at the depth which is enough to distinguish it with others.
  - For example, in the previous example, we can find the word "duck" with just "du".