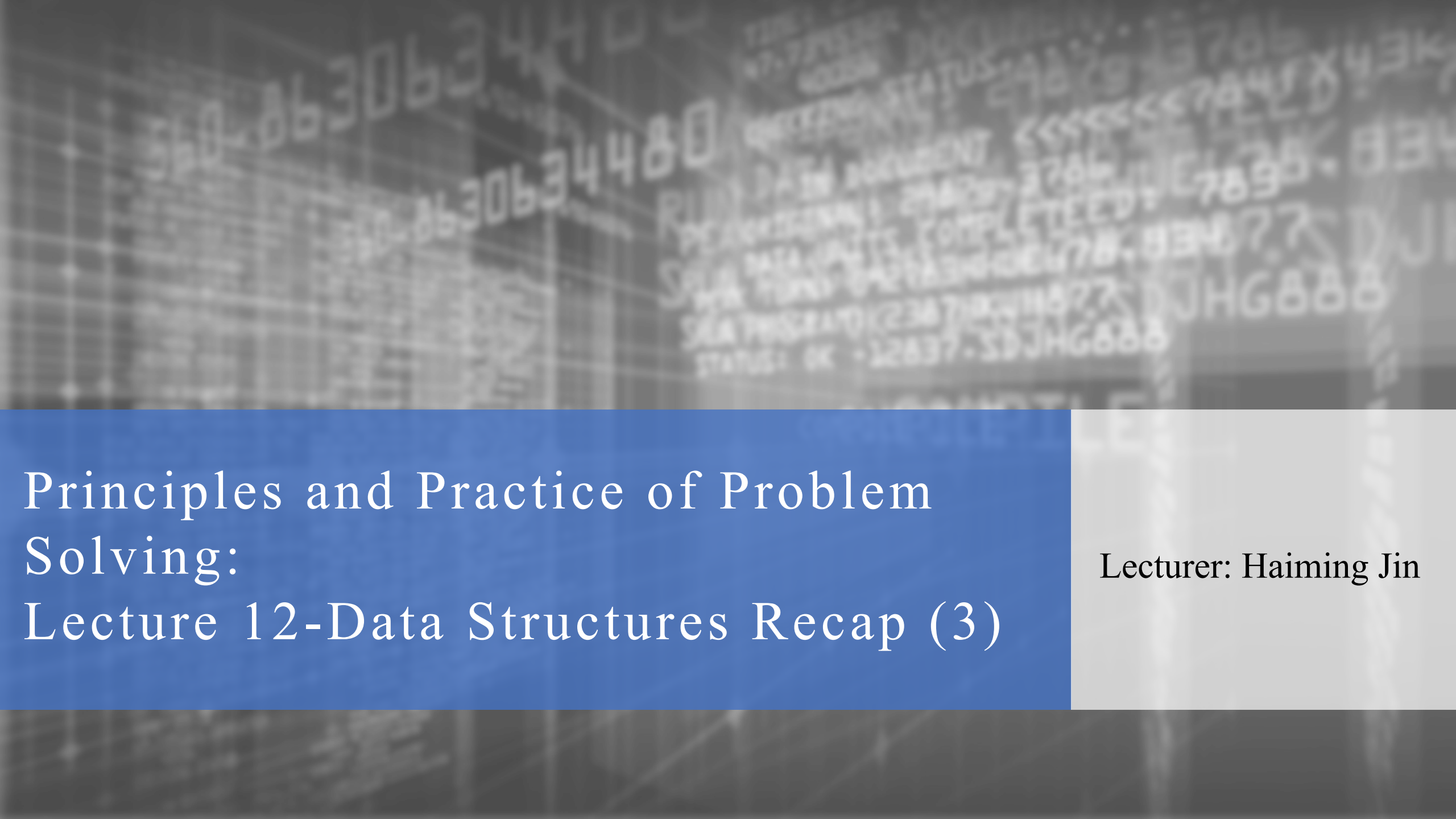


Principles and Practice of Problem Solving: Lecture 12-Data Structures Recap (3)

Lecturer: Haiming Jin

Outline

- k-d Tree
- Tries
- Red-Black 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)

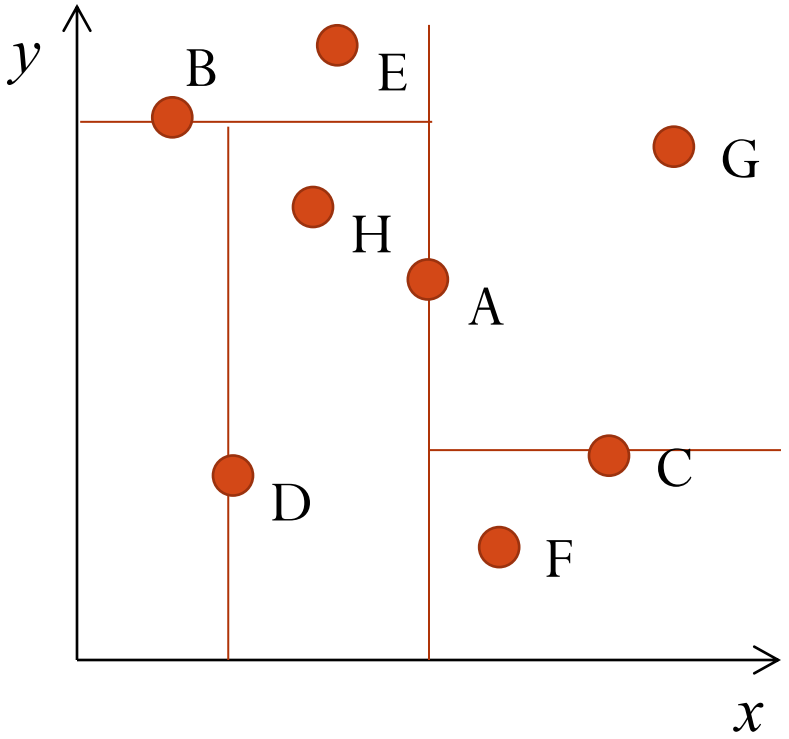
- Solution: k-d tree
  - $O(\log n)$ insert and search times

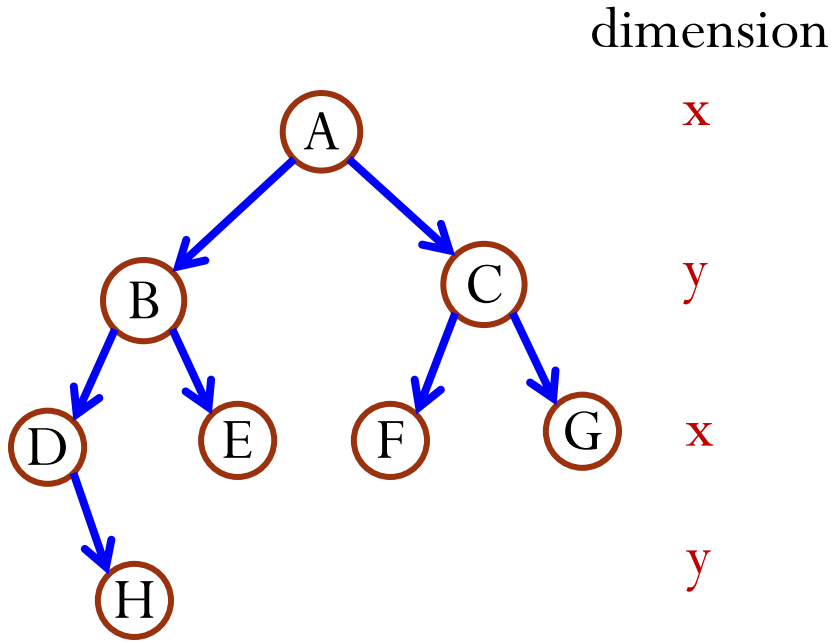
k-d Tree

- 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

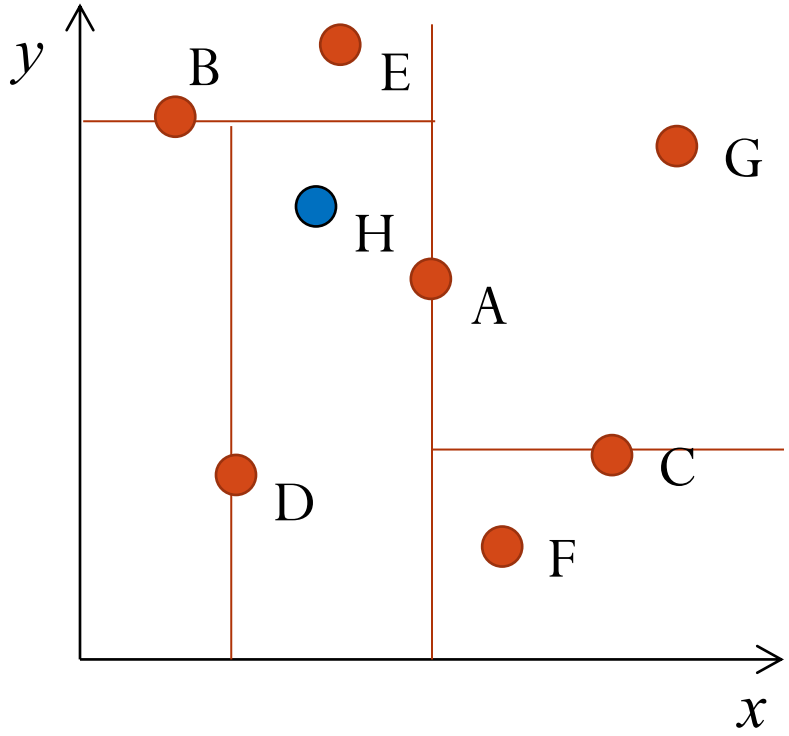
- If new item's key is equal to the root's key, return;
- If new item has a key smaller than that of root's along the dimension of the current level, recursive call on left subtree
- Else, recursive call on the right subtree
- In recursive call, cyclically increment the dimension

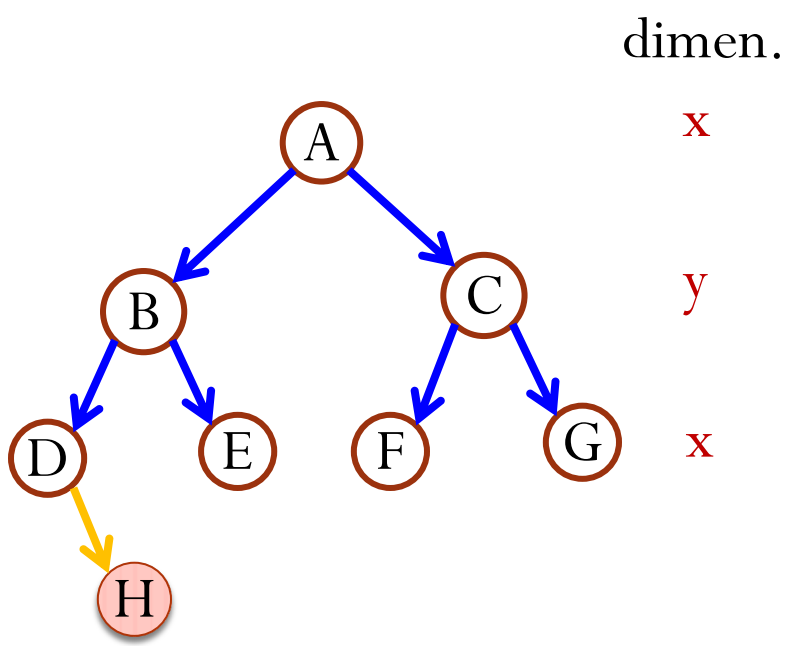
```
void insert(node *&root, Item item, int dim) {
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key == root->item.key) // equal in all
    return;                      // dimensions
  if(item.key[dim] < root->item.key[dim])
    insert(root->left, item, (dim+1)%numDim);
 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

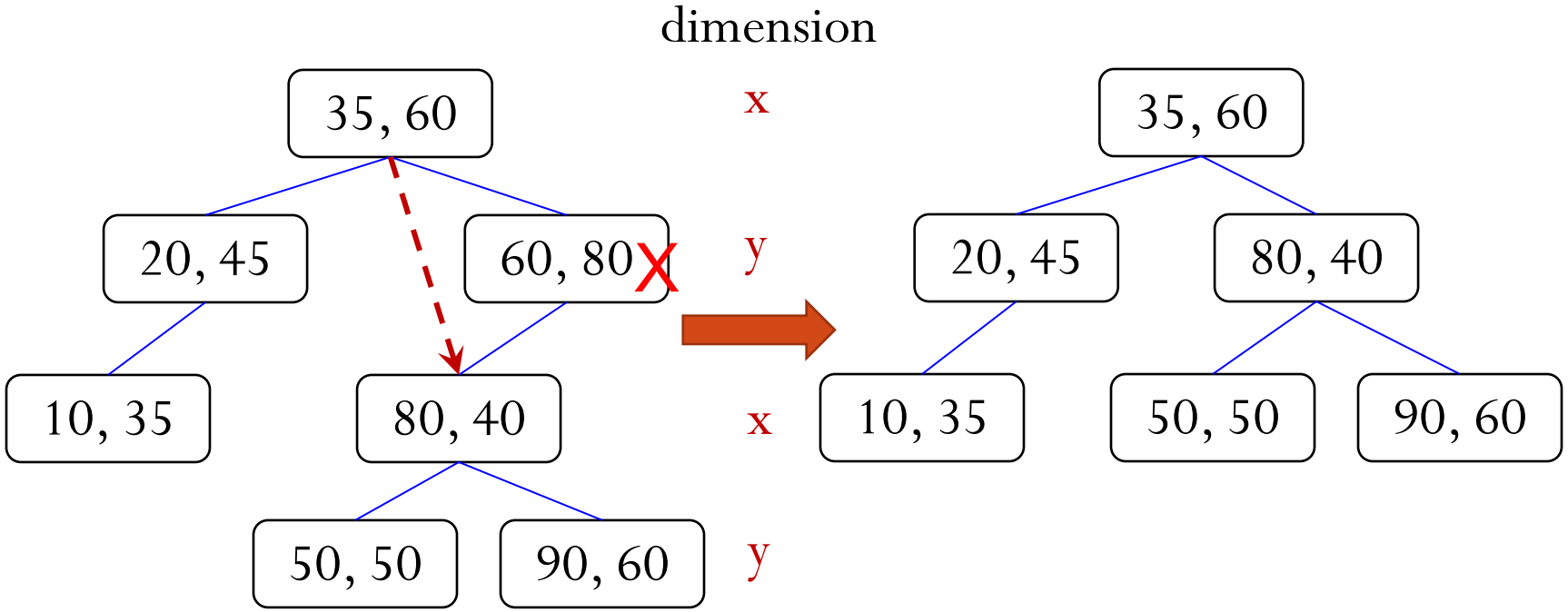
```
node *search(node *root, Key k, int dim) {
  if(root == NULL) return NULL;
  if(k == root->item.key)
    return root;
  if(k[dim] < root->item.key[dim])
    return search(root->left, k, (dim+1)%numDim);
  else
    return search(root->right, k, (dim+1)%numDim);
}
```

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)

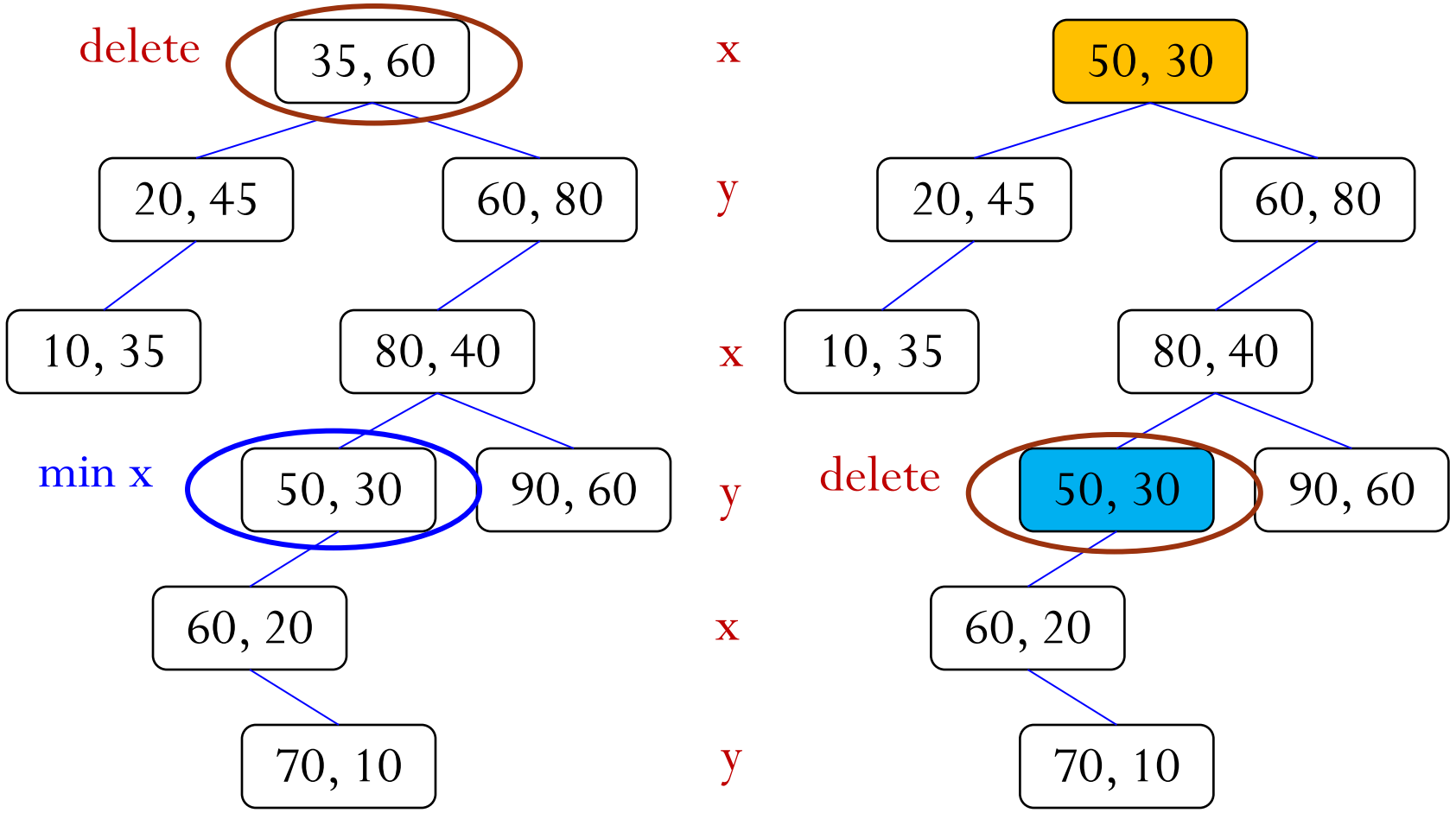
Answer: No!



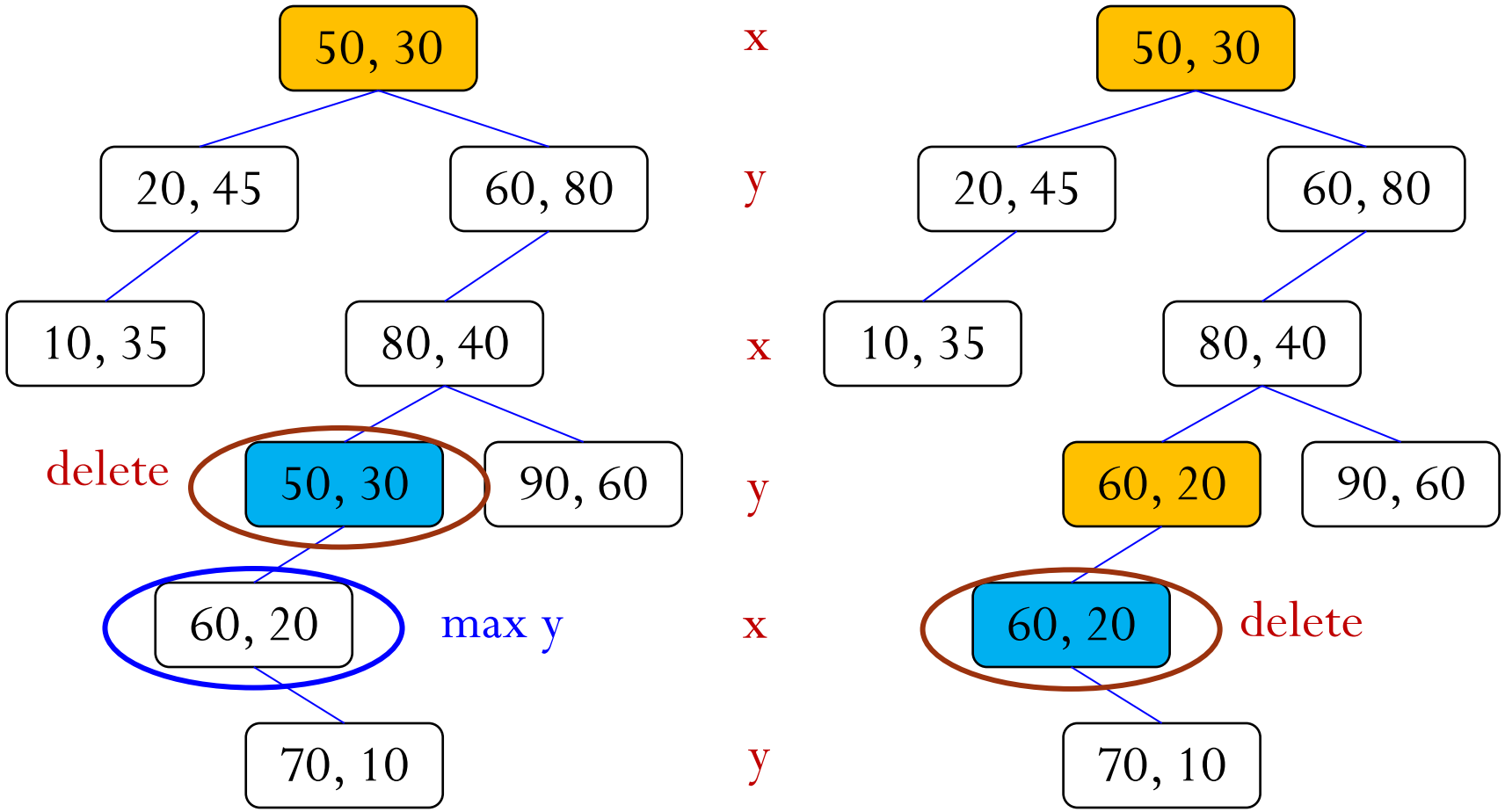
k-d Tree Removal of Non-leaf Node

- 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

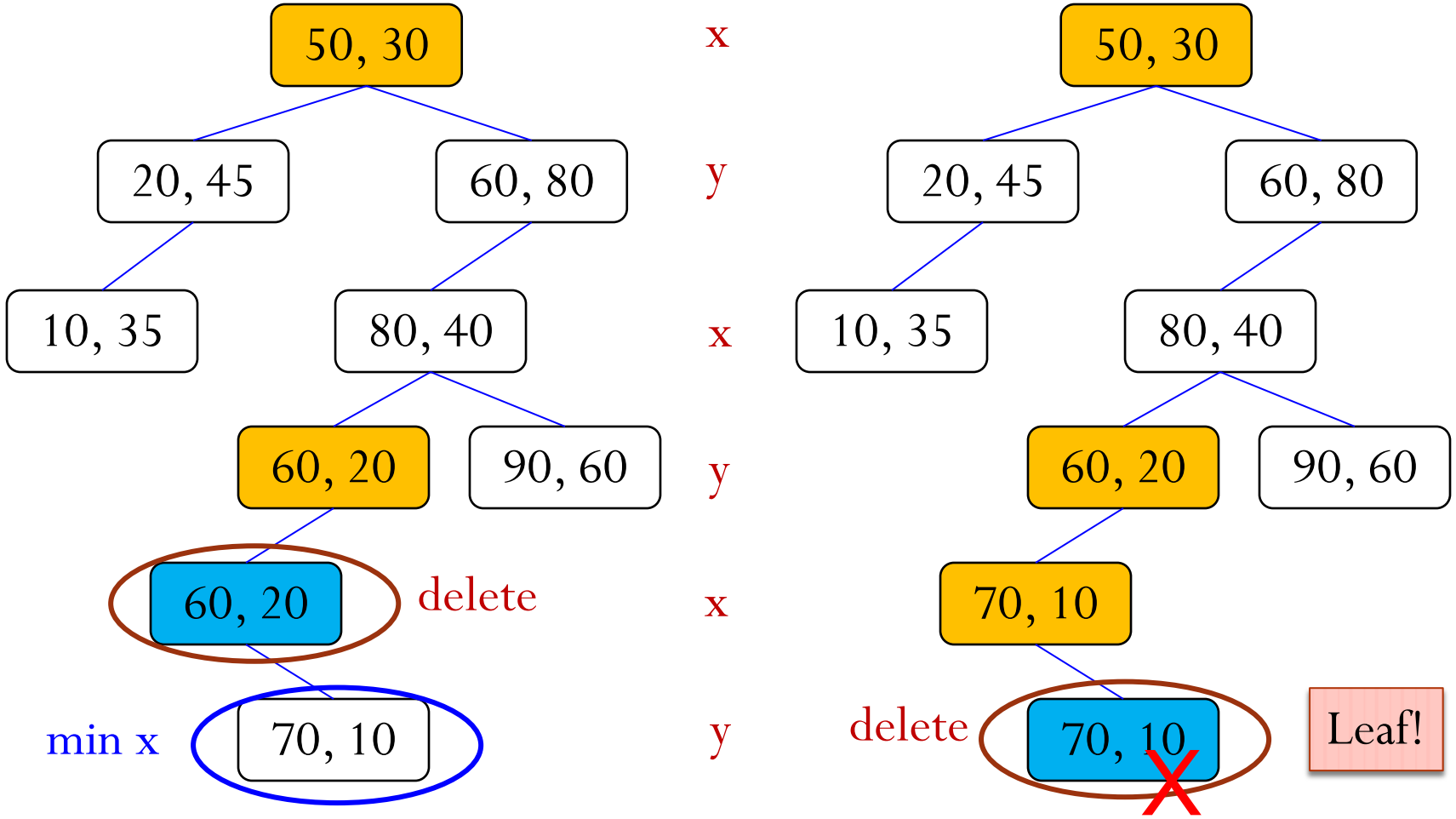
k-d Tree Removal Example



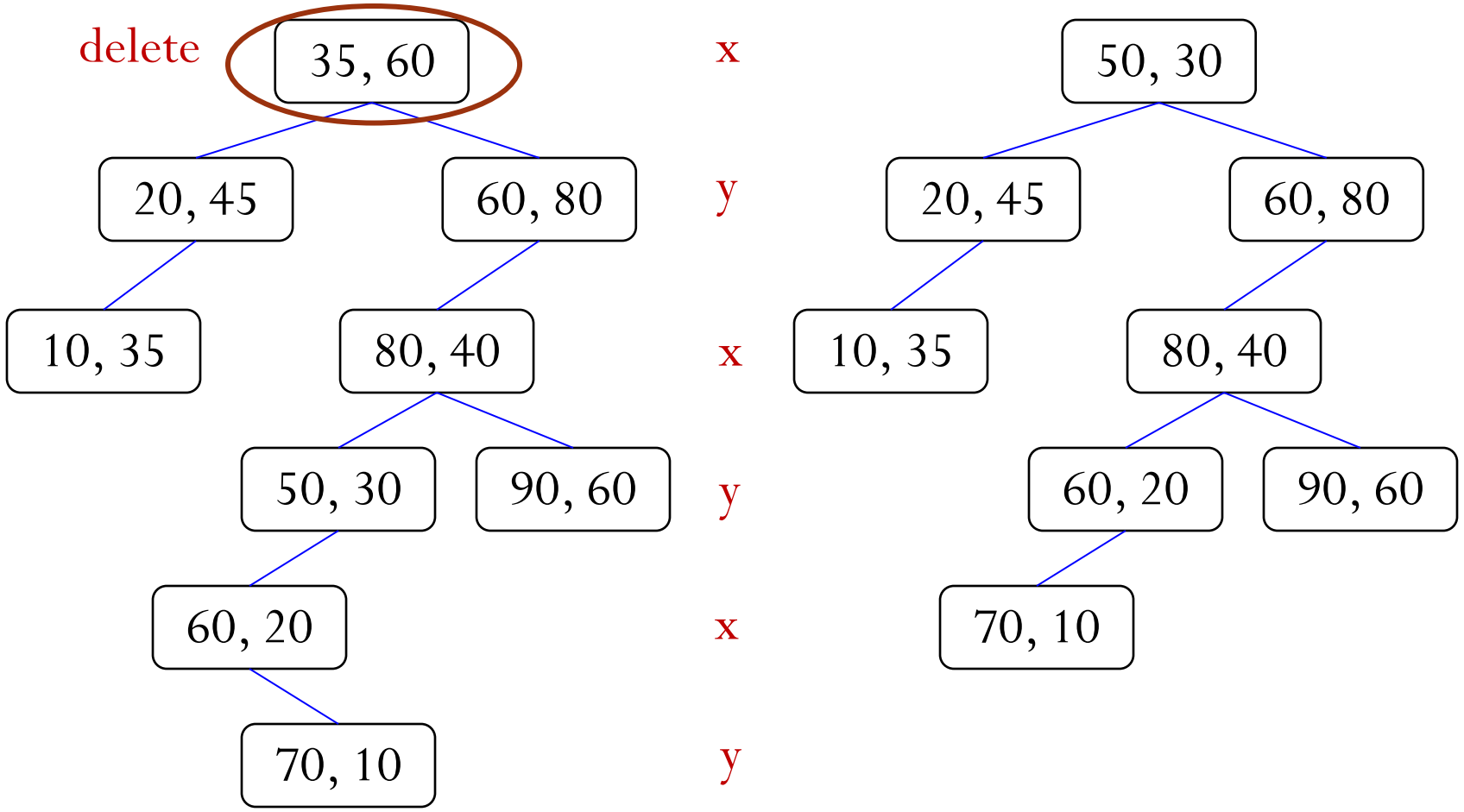
k-d Tree Removal Example



k-d Tree Removal Example

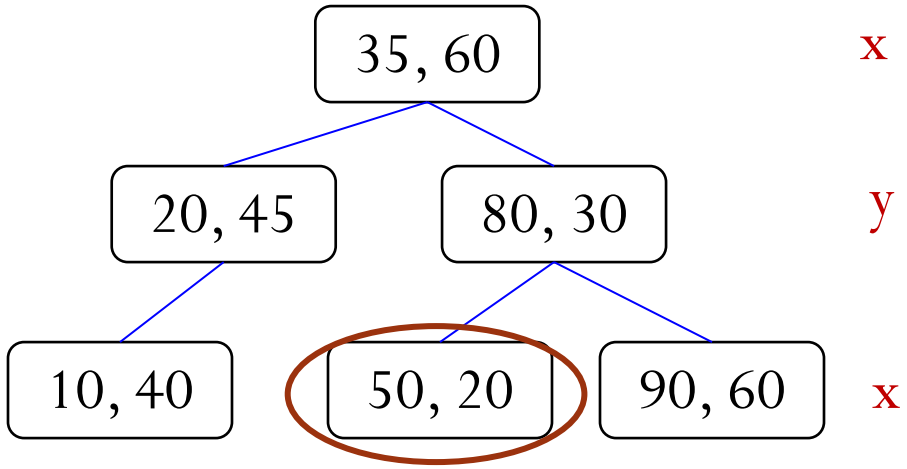


k-d Tree Removal Example: Summary



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 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)
```

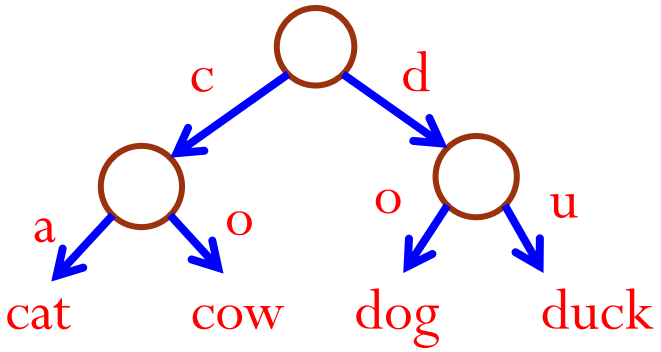
- Cycle through the dimensions as we go down the level
- **`searchRange[]`** holds two values (min, max) per dimension
  - Define a hyper-cube
  - min of dimension **`j`** at **`searchRange[2*j]`**, max at **`searchRange[2*j+1]`**
- **`treeRange[]`** holds lower bound and upper bound per dimension for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

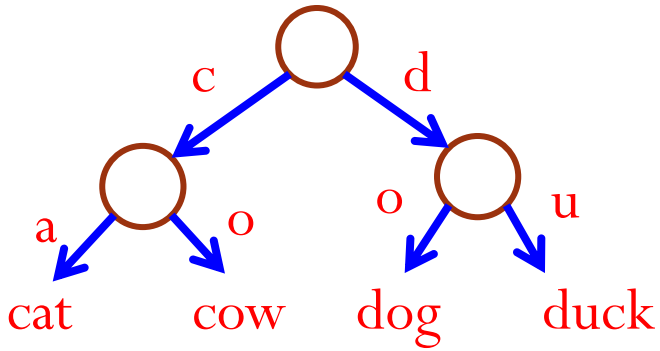
- k-d Tree
- Tries
- Red-Black Tree

Trie

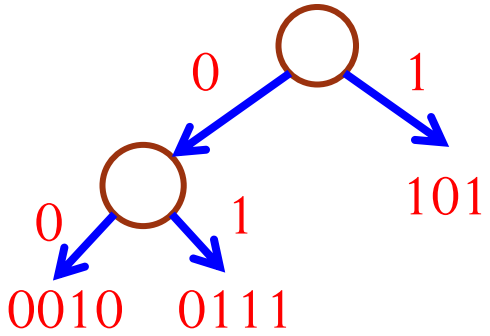
- The word "trie" comes from retrieval.
  - To distinguish with "tree", it is pronounced as "try".
- A trie is a tree that uses parts of the key, as opposed to the whole key, to perform search.
- Data records are only stored in **leaf** nodes. Internal nodes do not store records; they are "**branch**" points to direct the search process.



Trie

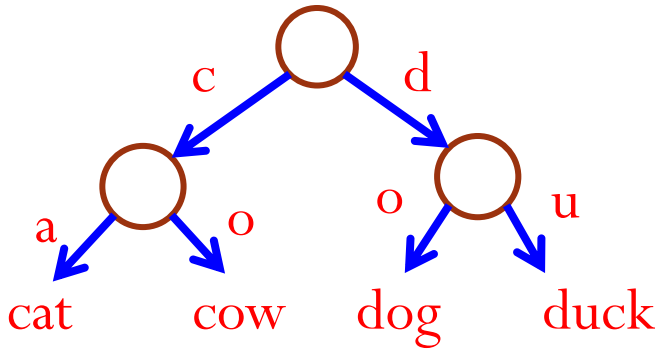


- Trie usually is used to store a set of strings from an alphabet.
  - The alphabet is in the general sense, not necessarily the English alphabet.
- For example, {0, 1} is an alphabet for binary codes {0010, 0111, 101}. We can store these three codes using a trie.



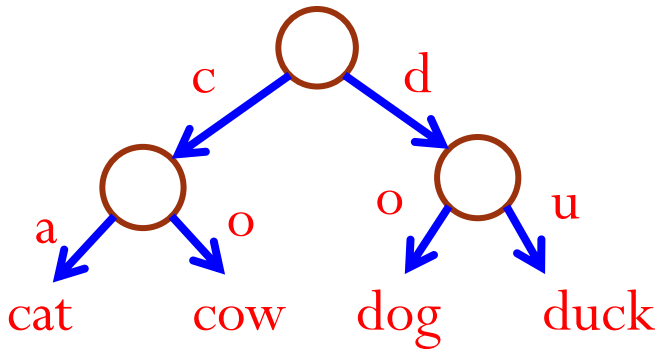
Trie

- 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

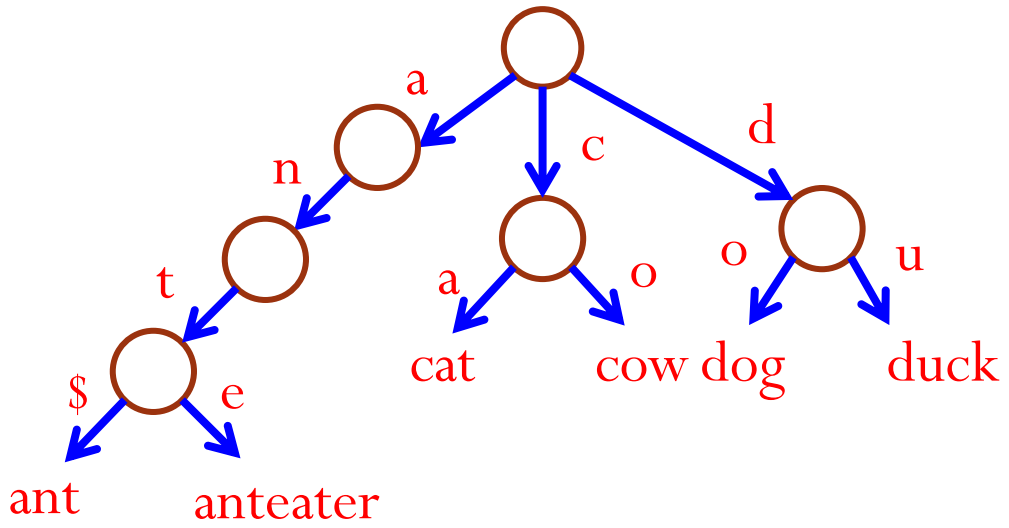
- The most significant symbol in a string determines the branch direction at the root.
- Each internal node is a "**branch**" point.
- As long as there is only one key in a branch, we do not need any further internal node below that branch; we can put the word directly as the leaf of that branch.



Trie

Implementation Issue

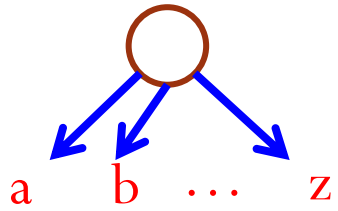
- Sometimes, a string in the set is exactly a **prefix** of another string.
  - For example, "ant" is a prefix of "anteater".
  - How can we make "ant" as a leaf in the trie?
- We add a symbol to the alphabet to indicate the end of a string. For example, use "$" to indicate the end.



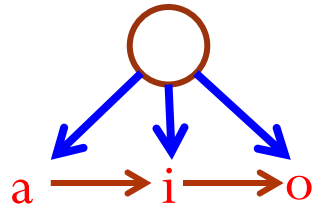
Trie

Implementation Issue

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



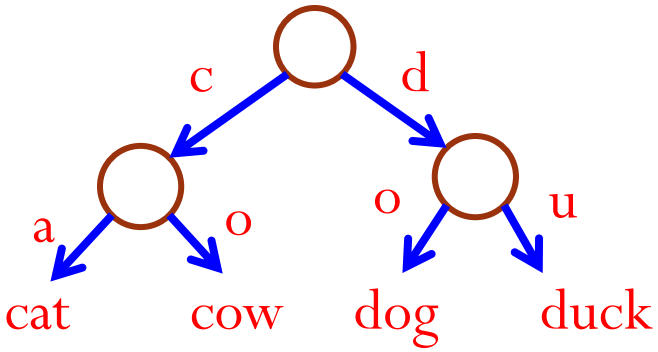
- However, most internal nodes have branches to only a small fraction of the possible symbols in the alphabet.
  - An alternate implementation is to store a linked list of pointers to the child nodes.



Trie

Search

- Follow the search path, starting from the root.
- When there is no branch, return false.
- When the search leads to a leaf, further compare with the key at the leaf.



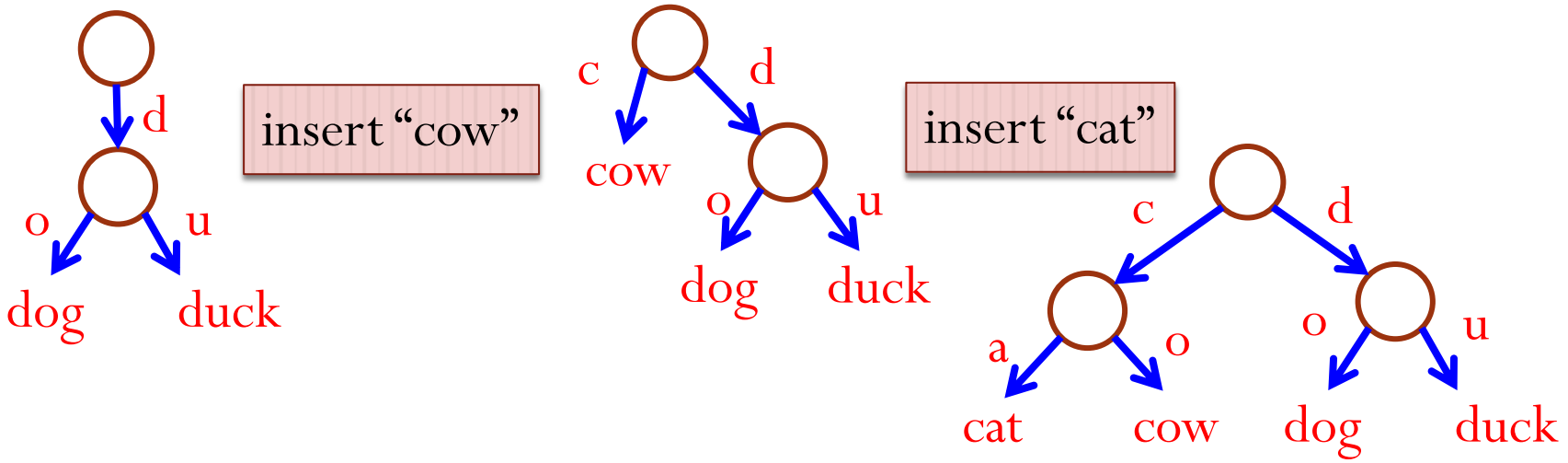
search “monkey”

search “cat”

Trie

Insertion

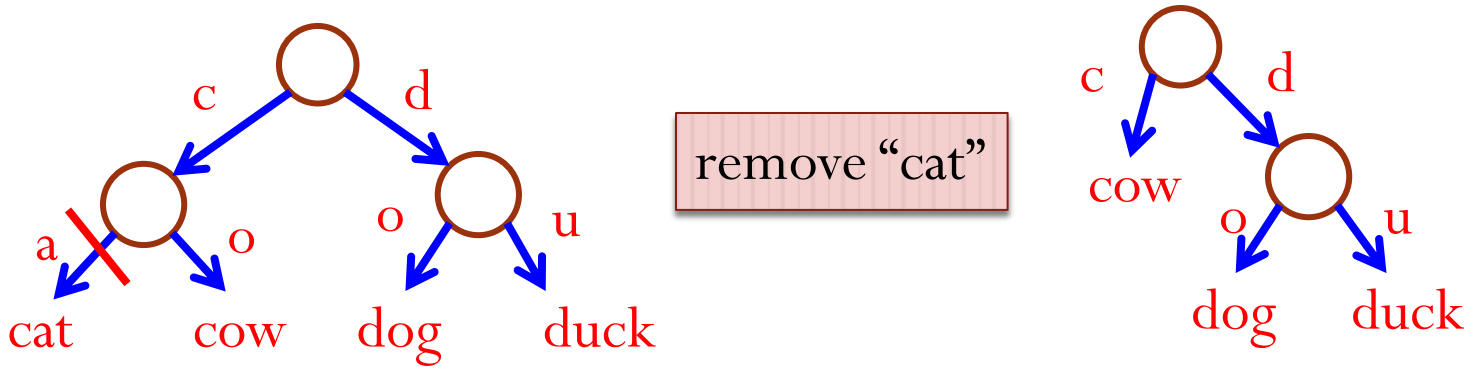
- Follow the search path, starting from the root.
- If a new branch is needed, add it.
- When the search leads to a leaf, a conflict occurs. We need to branch.
  - Use the next symbol in the key
  - The originally-unique word must be moved to lower level



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".

Outline

- k-d Tree
- Tries
- Red-Black Tree

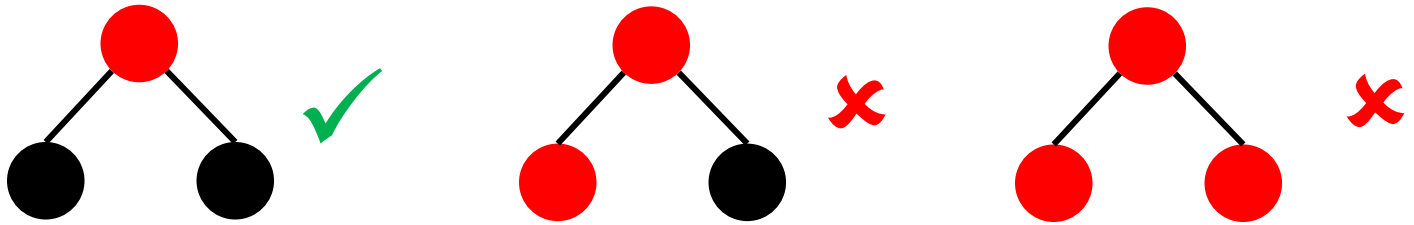
Red-Black Tree

- Binary search tree of height $h$can support any of the basic dynamic-set operations， such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE in $O(h)$ time.
- Thus, the set operations are fast if the height of the search tree is small.
- If its height is large, however, the set operations may run no faster than with a linked list.
- Red-black trees are one of many search-tree schemes that are "**balanced**" in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

Red-Black Tree

- A binary search tree. The data structure requires an extra one-bit color field in each node.
- Property

1. Every node is either red or black.
2. **Root rule**: The root is black.
3. **Red rule**: Red node can **only have** black children.
   - Can't have two consecutive red nodes on a path.

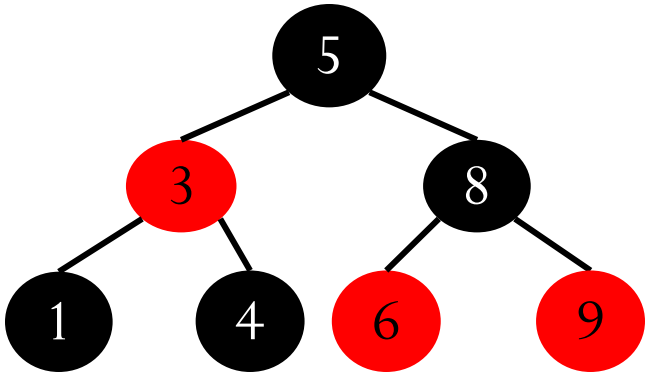


4. **Path rule**: **Every** path from a node x to NULL must have the **same number** of black nodes (including x itself).

Red-Black Tree Example

- Property

1. A binary search tree
2. Every node is either red or black.
3. **Root rule**: The root is black.
4. **Red rule**: Red node can **only have** black children.
5. **Path rule**: **Every** path from a node x to NULL must have the **same number** of black nodes (including x itself).

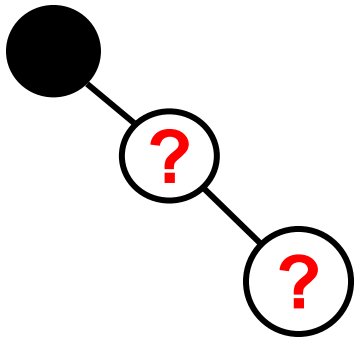


Counter Example

- Property

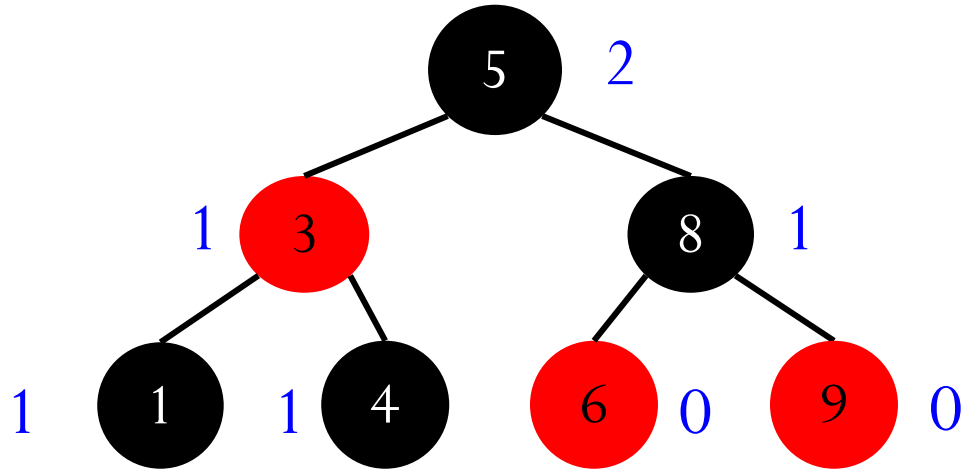
1. A binary search tree
2. Every node is either red or black.
3. **Root rule**: The root is black.
4. **Red rule**: Red node can **only have** black children.
5. **Path rule**: **Every** path from a node x to NULL must have the **same number** of black nodes (including x itself).

- **Claim**: a chain of length 3 cannot be a red-black tree



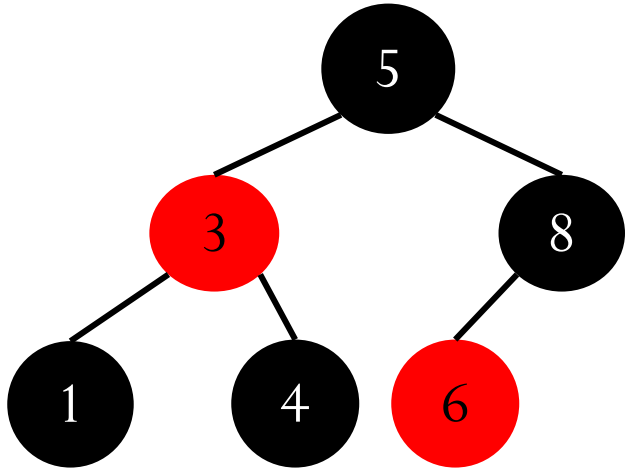
Black Height

- **Black height** of a node x is the number of black nodes on the path from x to NULL, **including** x itself.



Implication of the Rules

- If a **red** node has **at least one** child, it **must have two children** and they must be **black**.
  - Why?
    - A red node's child can only be black.
    - If has only one black child, then violate the **path rule**.
- If a black node has **only one** child, that child **must be** a **red leaf**.
  - Why?
    - Can't be black.
    - Must be a leaf.



Height Guarantee

- **Claim**: every red-black tree with n nodes has height $\leq 2\log_2(n+1)$.
- Proof:
  - In a binary tree with n nodes, there is a root-NULL path with **at most** $\log_2(n+1)$ nodes. (why?)
    - **Thus**: # black nodes on that path $\leq \log_2(n+1)$.
  - By **path rule**: every root-NULL path has $\leq \log_2(n+1)$ **black nodes**.
  - By **red rule**: every root-NULL path has $\leq 2\log_2(n+1)$ **total nodes**.

Q.E.D.

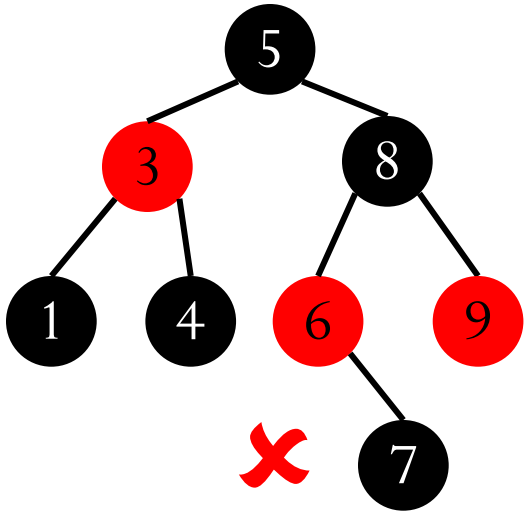
Operations on Red-Black Trees

- All **query operations** (e.g., search, min, max, succ, pred) work just like those on general BST.
  - They run in $O(\log n)$ time on a red-black trees with n nodes in the **worst case**.

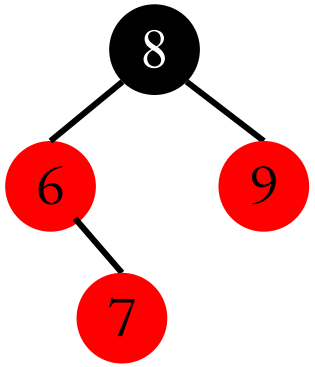
- The **modifying** operations "insertion" and "removal" must maintain the red-black tree properties.
  - They are complex.

Insertion

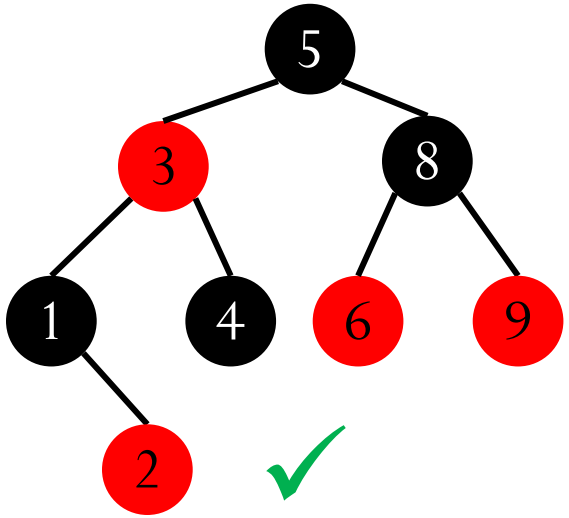
- New node is always a **leaf**.
  - However, it can't be **black**!
    - Otherwise, violate path rule.
  - Therefore the new leaf must be **red**.



- If parent is black, done (trivial case).
- If parent is red, violate the **red rule**!

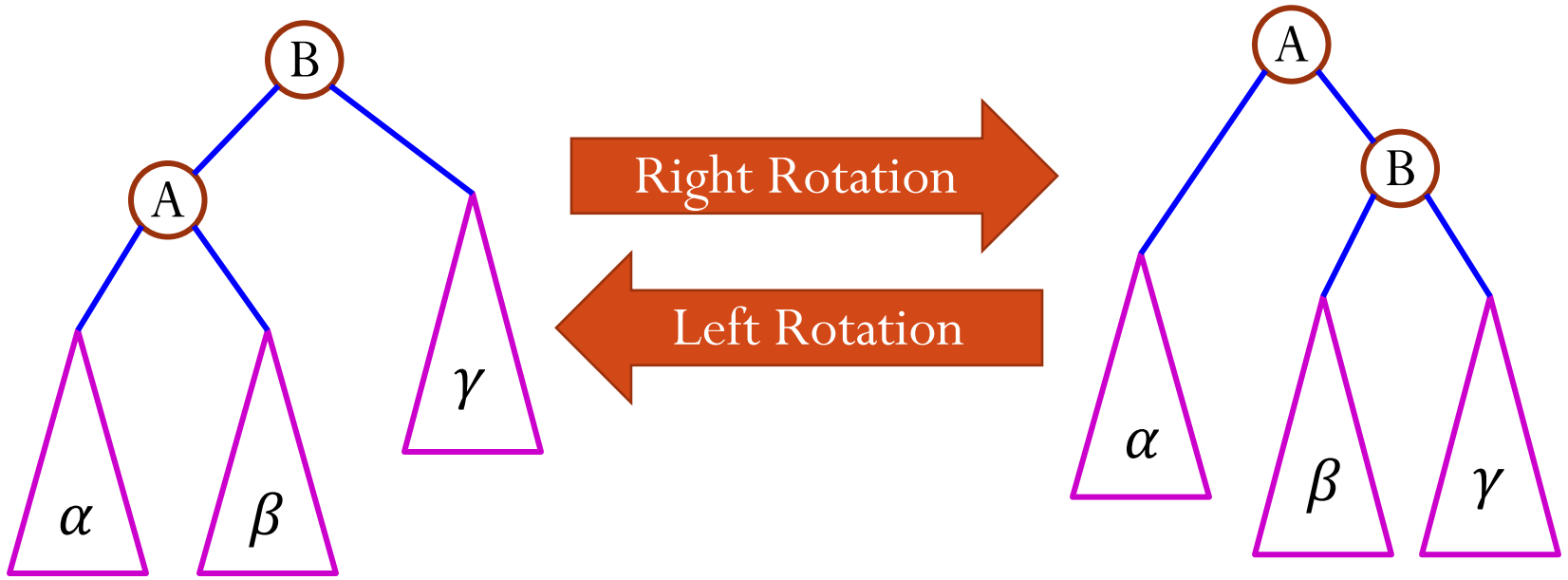


We have to do some work…

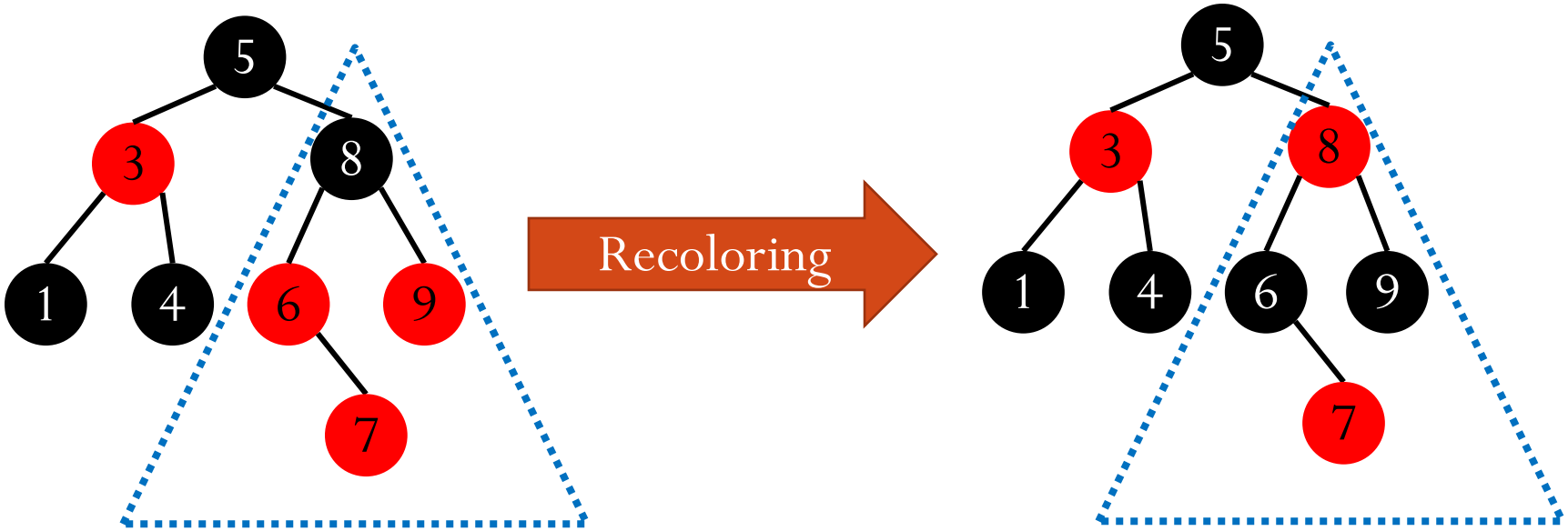


Modification: Rotation

- Maintain the binary search tree property.
- Can be done in $O(1)$ time.



Modification: Recoloring

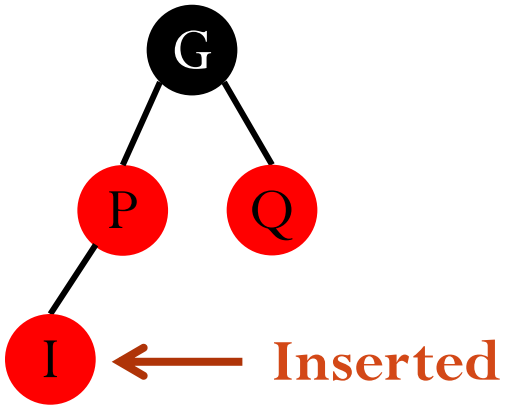


Insertion: Sketch

- Insert x as a **leaf**.
- Color x **red**.
  - Only **red rule** may be violated.
- Move the violation **up the tree** by recoloring/rotation.
  - At some point, the violation will be fixed.

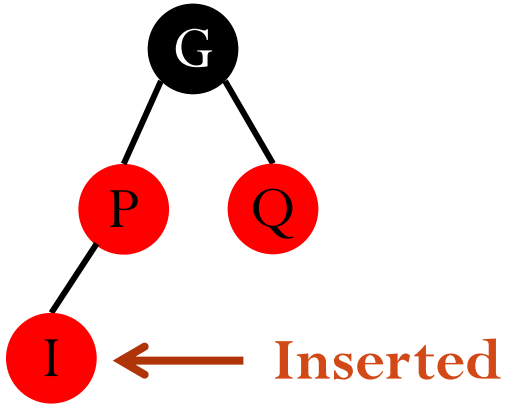
Violation at Leaf

- **Note**: only **red rule** may be violated by inserting a (red) node as a leaf.
- When violating, its **parent** is **red** and its **grandparent** is **black**.
- **Denote**: the inserted node as "I", its parent as "P", its grandparent as "G".



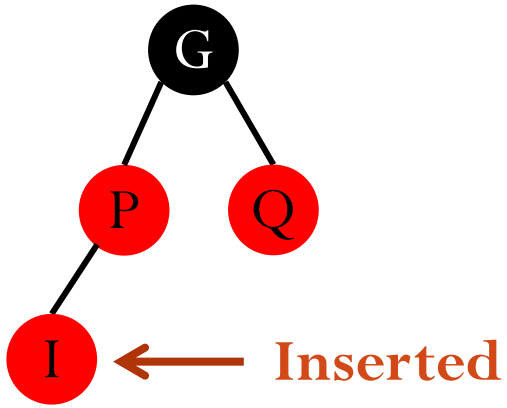
Violation at Leaf

- **Note**: only **red rule** may be violated by inserting a (red) node as a leaf.
- When violating, its **parent** is **red** and its **grandparent** is **black**.
- **Denote**: the inserted node as "I", its parent as "P", its grandparent as "G".
- **Claim**: in the old tree, "P" is a leaf, i.e., has no children.



Violation at Leaf

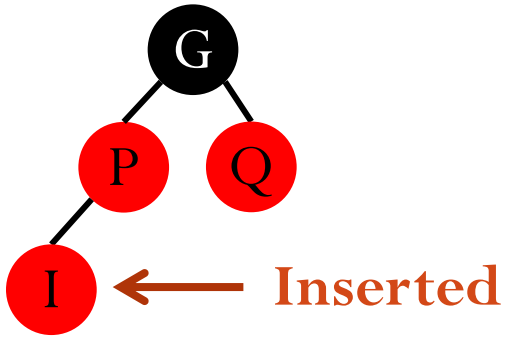
- **Assume**: the parent "P" is the **left child** of the grandparent "G".
  - The "right child" case is **symmetric**.
- **Denote**: the right child of the grandparent to be Q.
- **Claim**: Q is either a red leaf or a NULL.
  - Why?



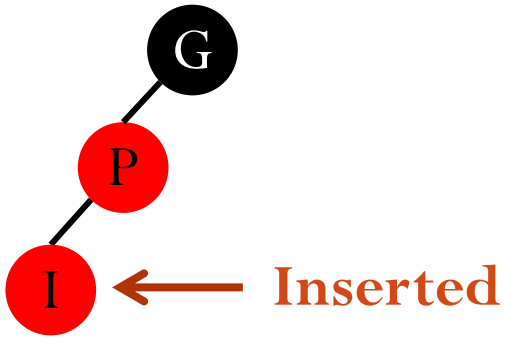
Violation at Leaf

- Three cases:

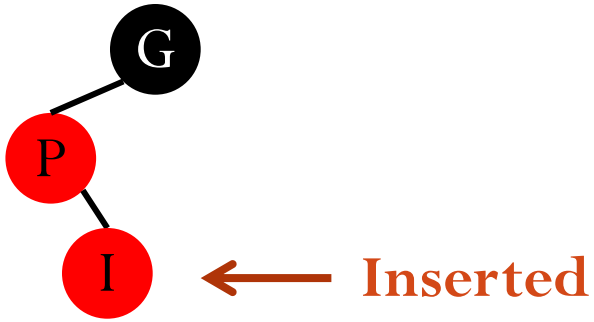
1. Q is a **red leaf**.



2. Q is empty; I is P's **left** child.



3. Q is empty; I is P's **right** child.



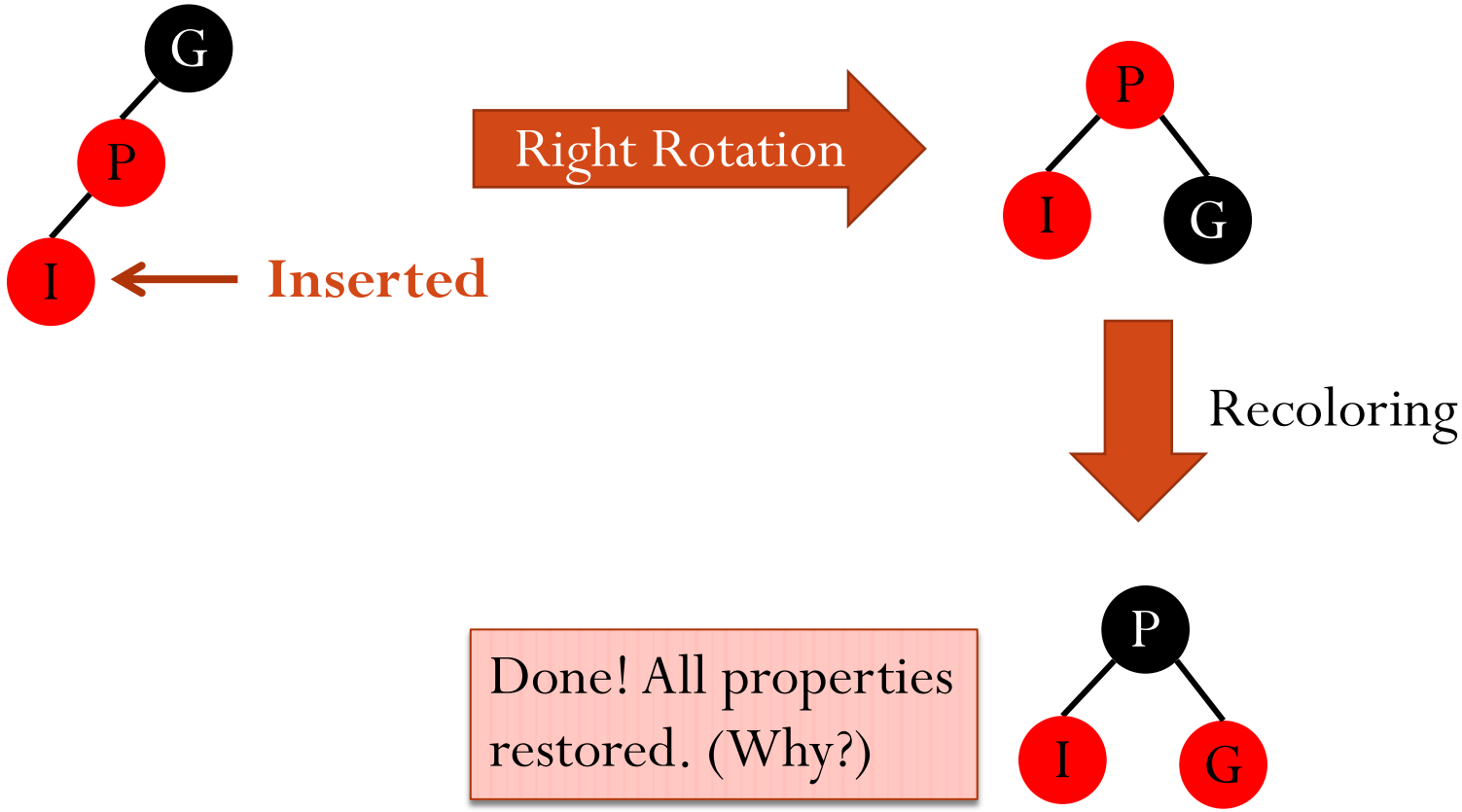
Violation at Leaf

- Case 1: Q is a **red leaf**.

May **recurse**, since G's parent may be red.

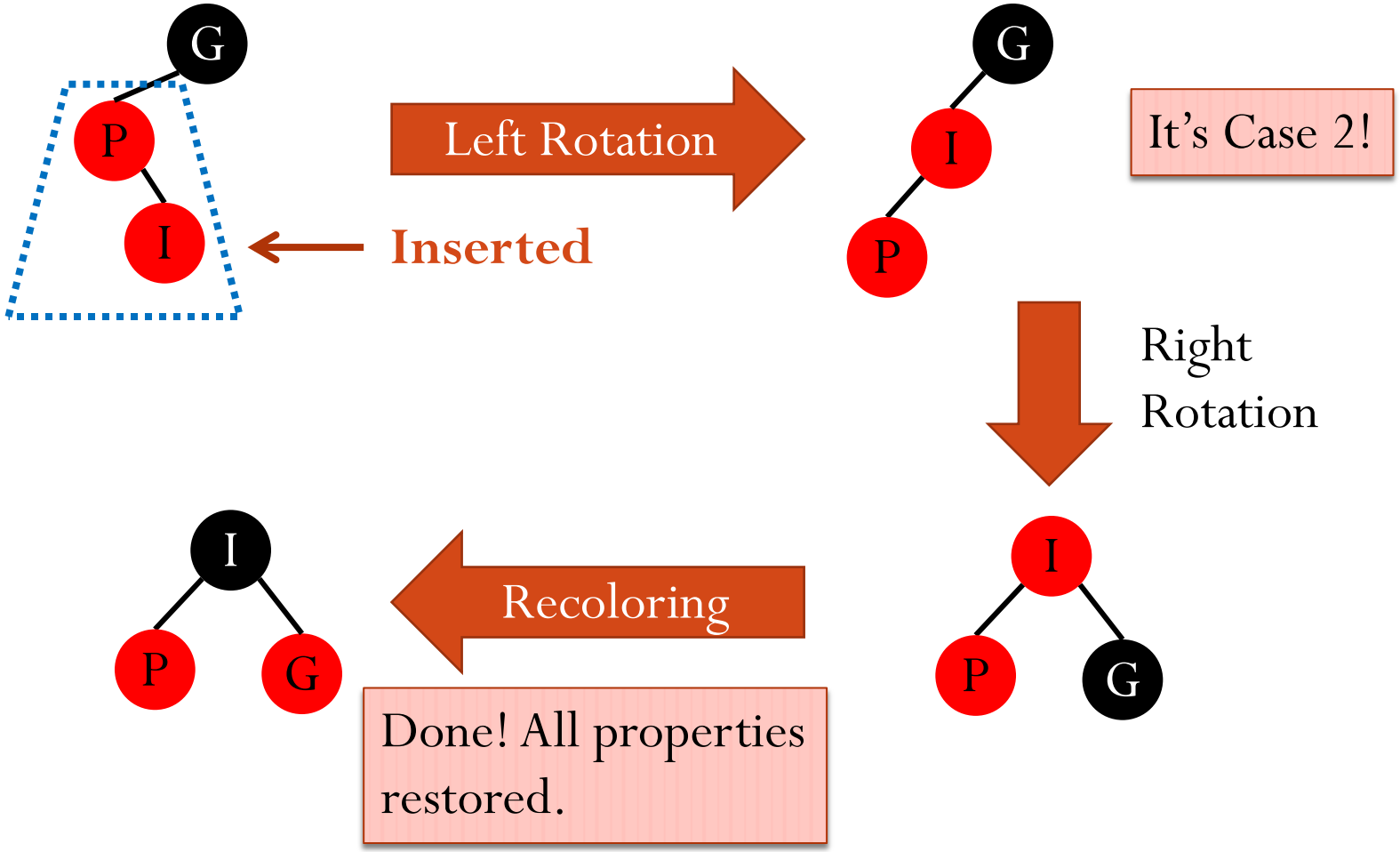
Violation at Leaf

- Case 2: Q is empty; I is P's **left** child.



Violation at Leaf

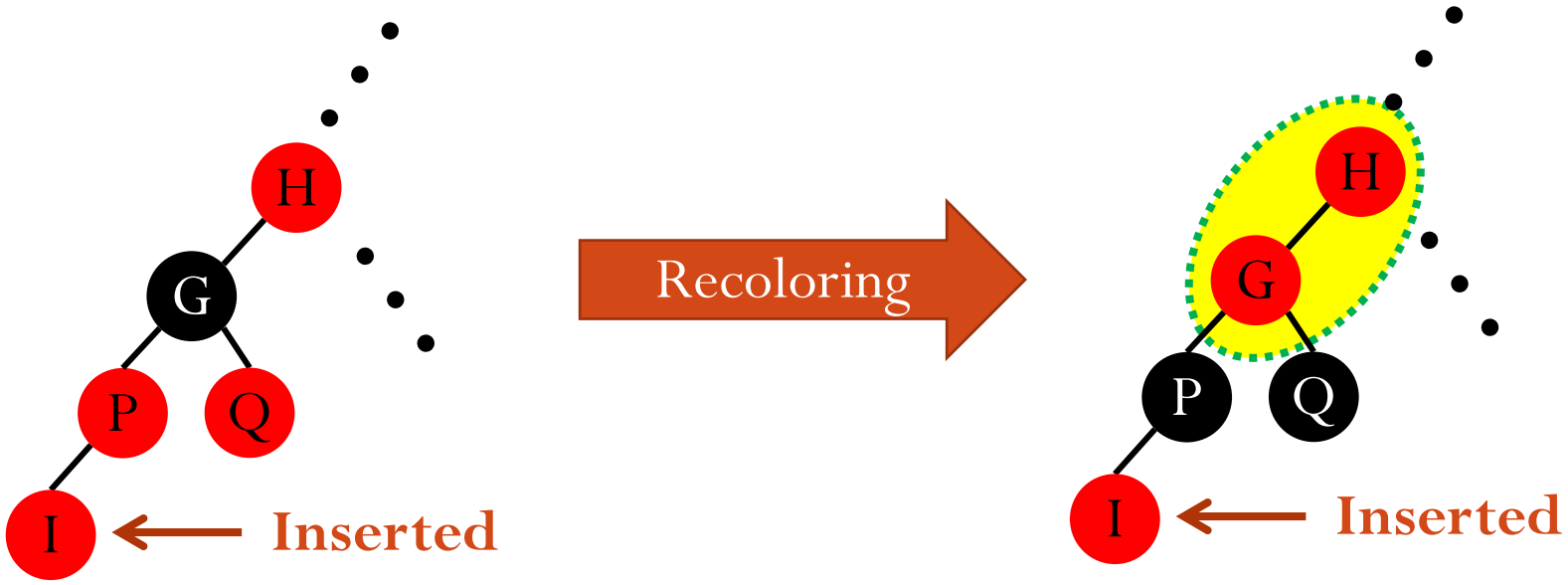
- Case 3: Q is empty; I is P's **right** child.



Violation at Leaf: Summary

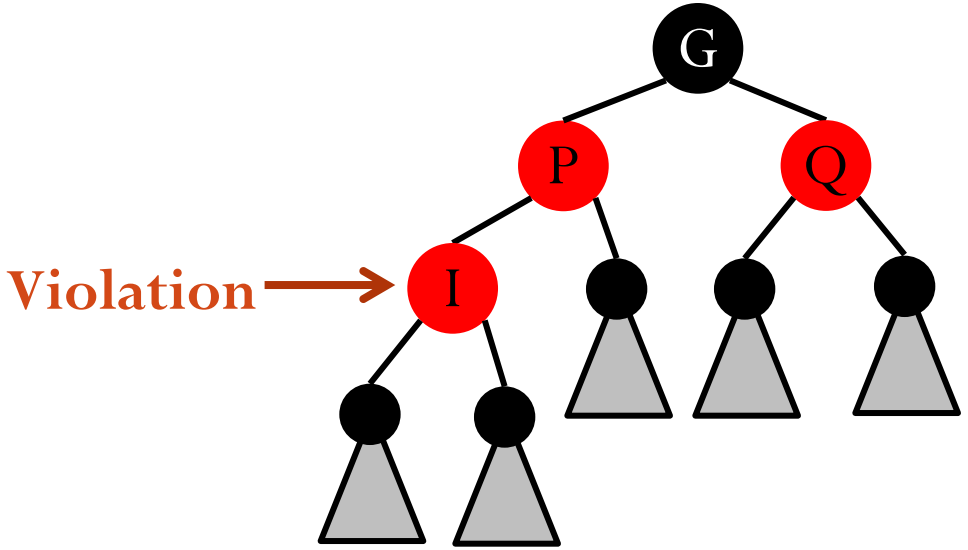
- For Case 2 (Q is empty; I is P's **left** child) and Case 3 (Q is empty; I is P's **right** child), **we're done**.

- For Case 1 (Q is a **red leaf**), we may recurse.
  - Violation of **red rule**.



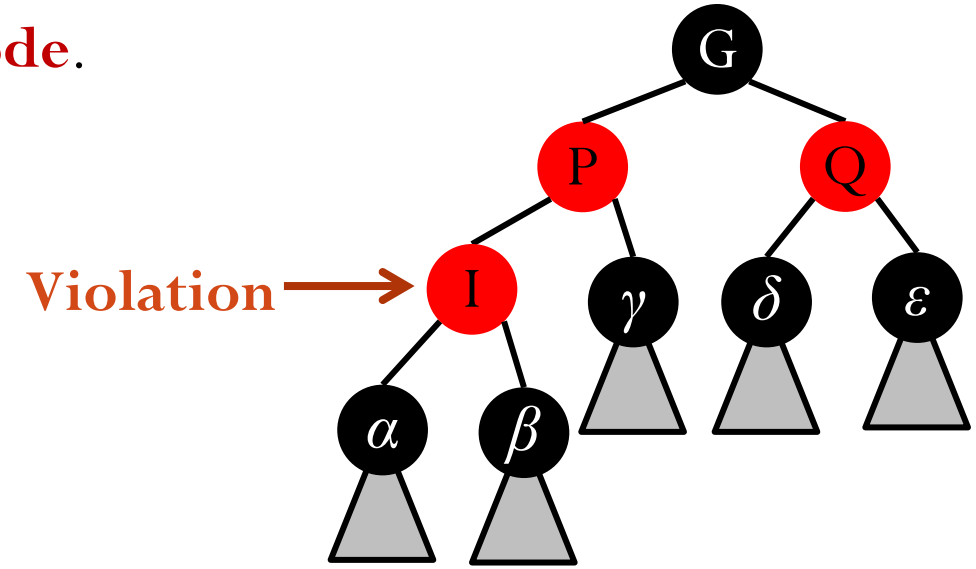
Violation at Internal Nodes

- Caused by **moving the violation up** the tree.
- When violating, its **parent** is **red** and its **grandparent** is **black**.
- **Assume**: the parent "P" is the **left child** of the grandparent "G". (The "right child" case is **symmetric**.)
- **Denote**: the right child of the grandparent to be Q.



Violation at Internal Nodes

- Three Cases:
  1. Q is a **red node**.



- **Claim**:
  - α, β, γ, δ, ε are trees with **black root**.
  - α, β, γ, δ, ε have the **same black height**.

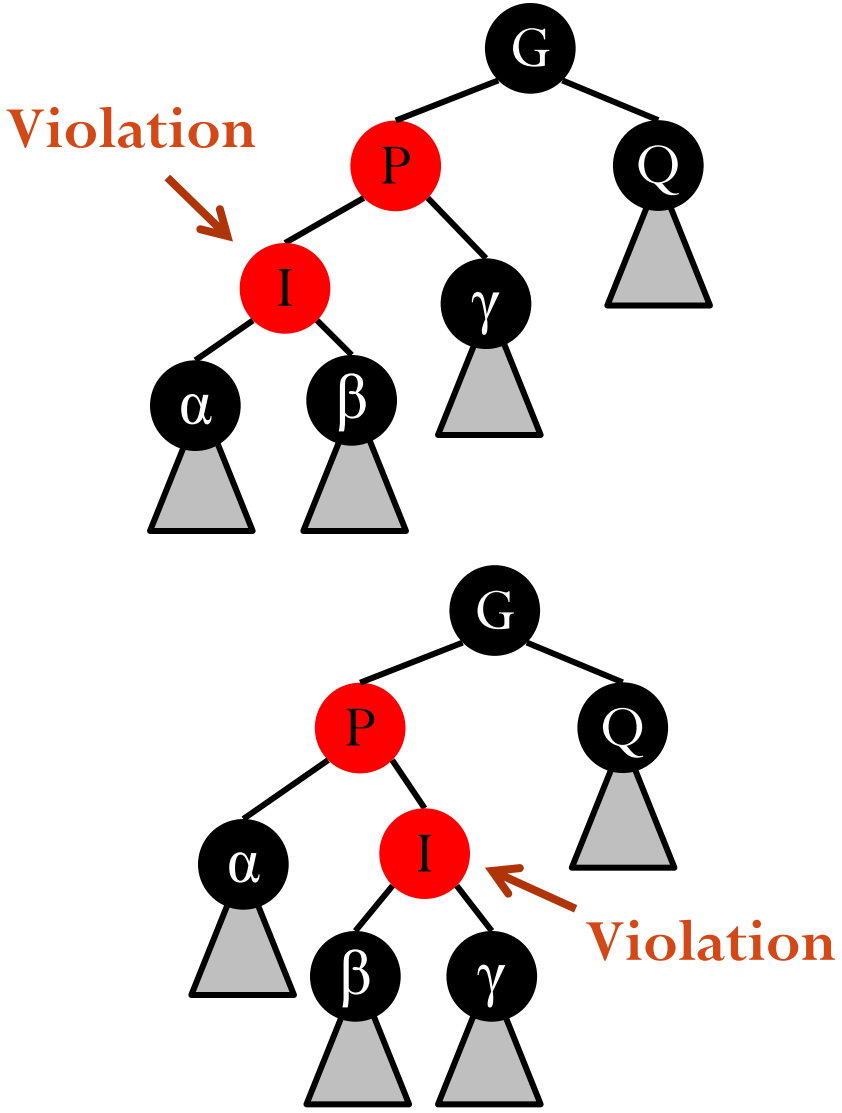
Violation at Internal Nodes

- Three Cases:

  2. Q is a **black node**; I is P's **left** child.

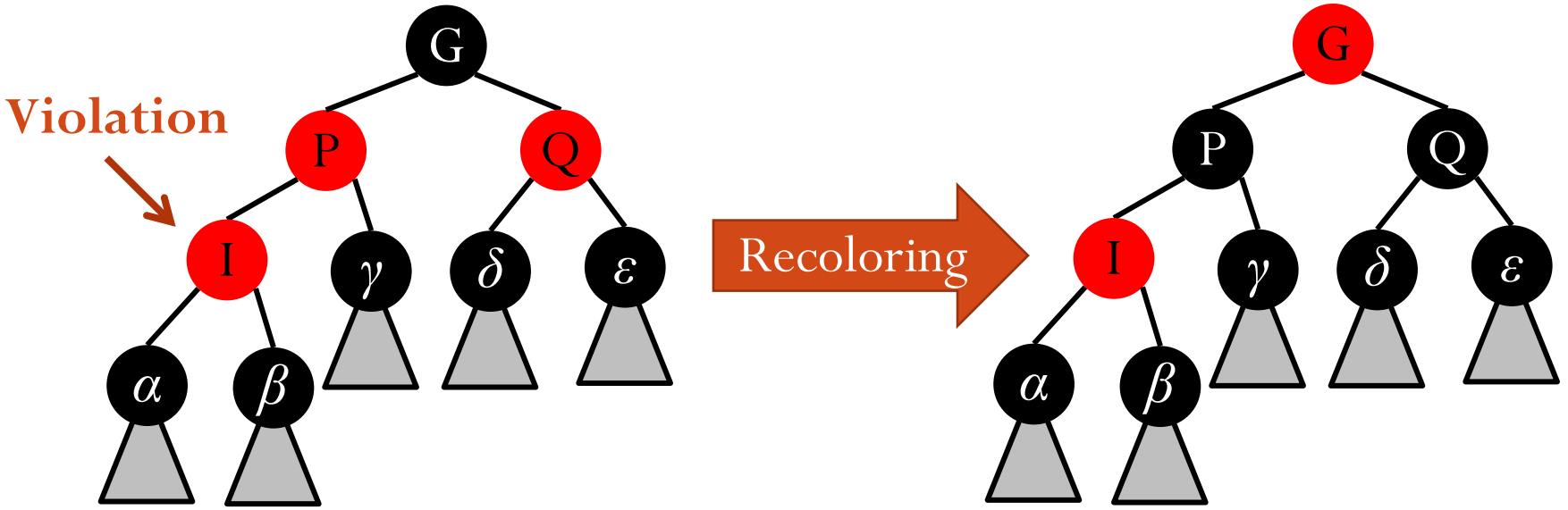
  3. Q is a **black node**; I is P's **right** child.

- **Claim** for Case 2 and 3:
  - α, β, γ, Q are trees with **black root**.
  - α, β, γ, Q have the **same black height**.



Violation at Internal Nodes

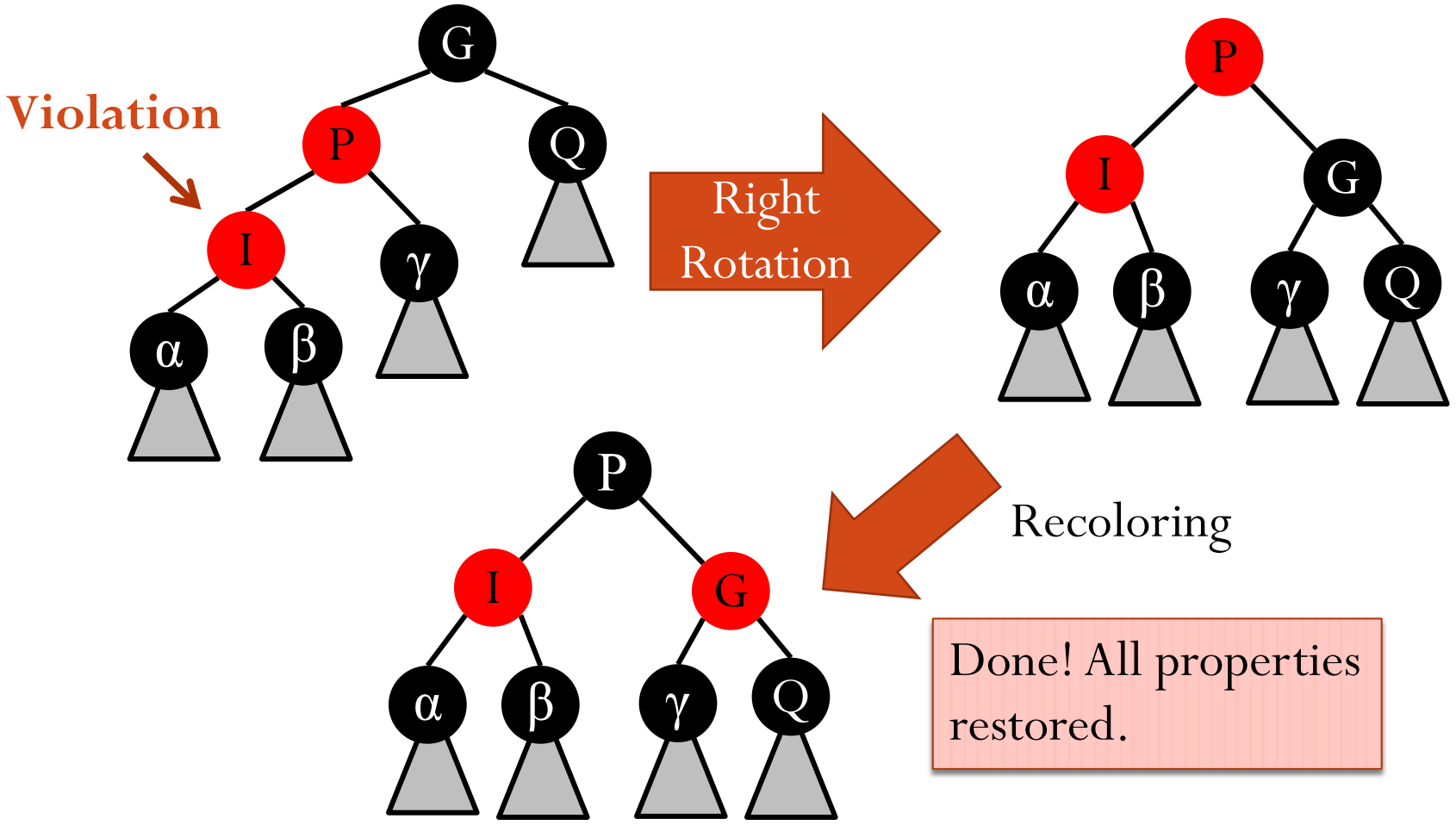
- Case 1: Q is a **red node**.



May **recurse**, since G's parent may be red.

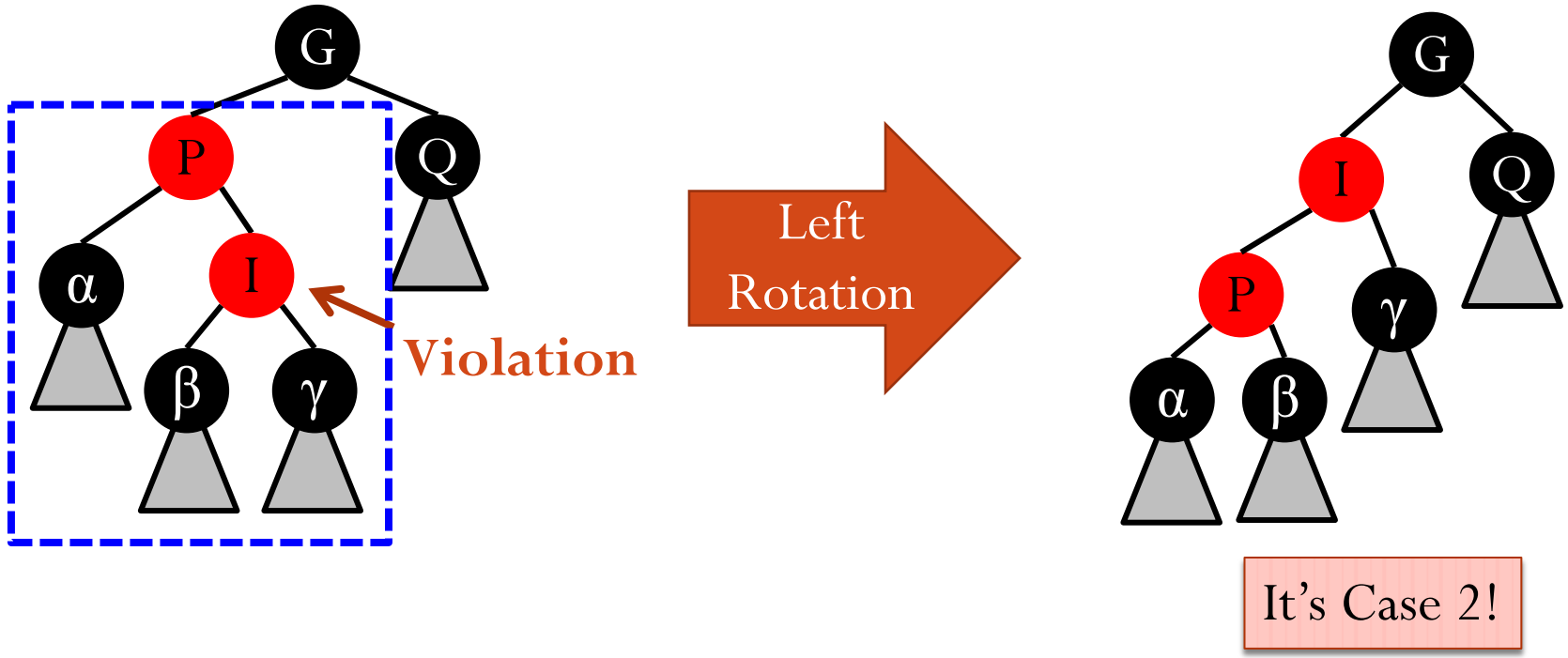
Violation at Internal Nodes

- Case 2: Q is a **black node**; I is P's **left** child.

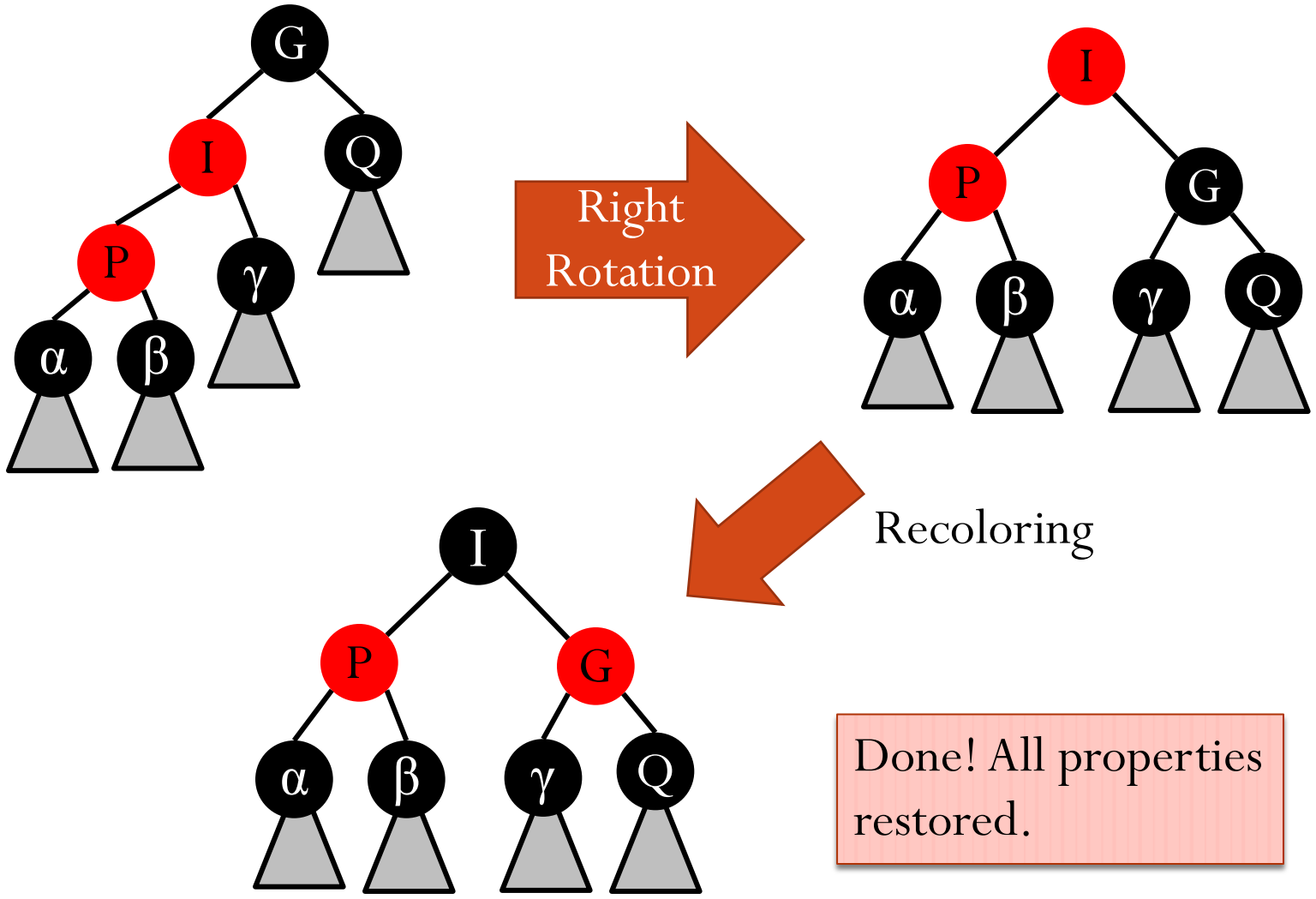


Violation at Internal Nodes

- Case 3: Q is a **black node**; I is P's **right** child.

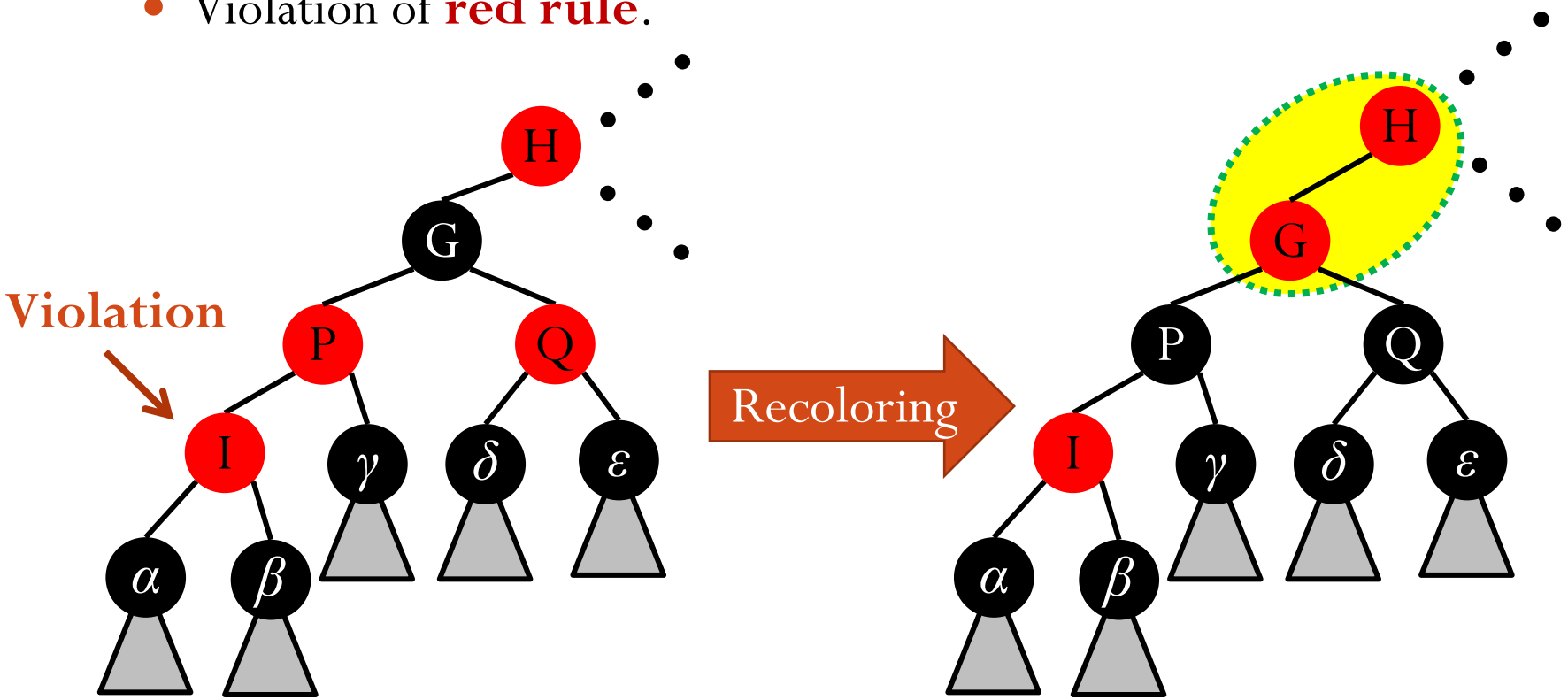


Violation at Internal Nodes: Case 3 (cont.)



Violation at Internal Nodes: Summary

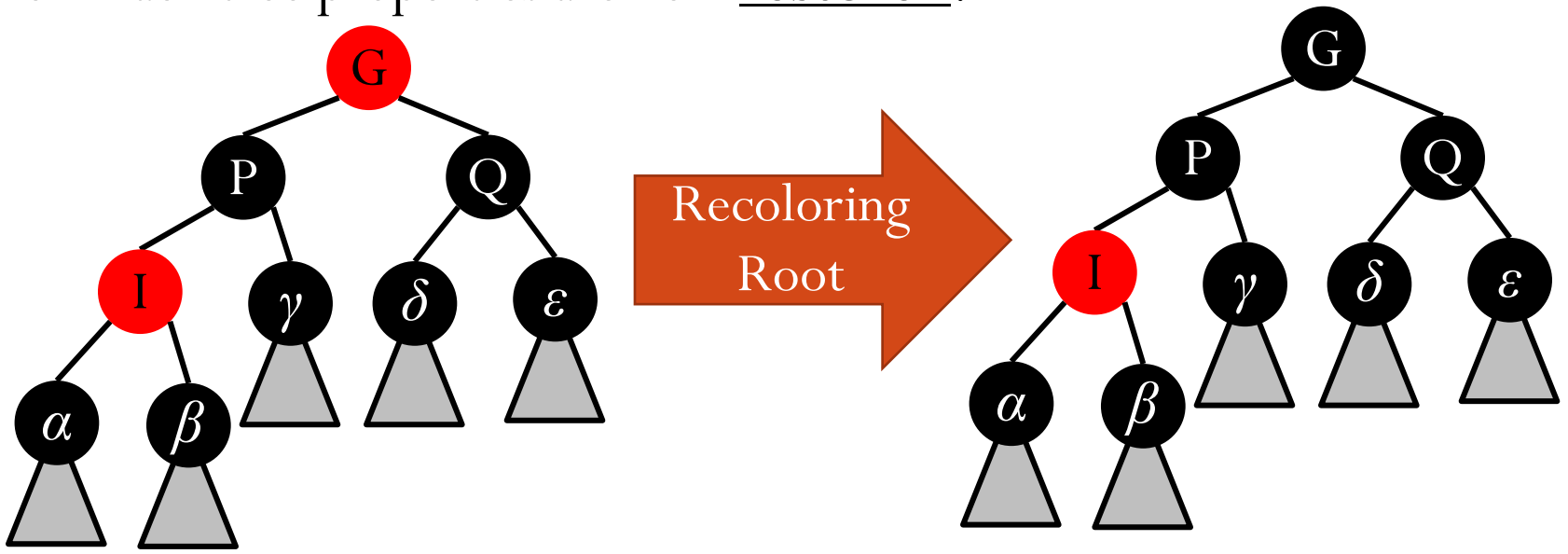
- For Case 2 (Q is a **black node**; I is P's **left** child) and Case 3 (Q is a **black node**; I is P's **right** child), **we're done**.
- For Case 1 (Q is a **red node**), we may recurse.
  - Violation of **red rule**.



Final Step: Violation Fix at the Root

- By **moving the violation up** the tree …
  - … the root may become **red**.

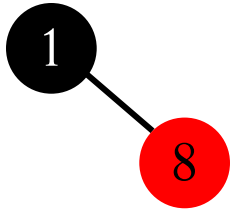
- Final step: set root to be **black**.
  - All red-black tree properties are now **restored**.



Example

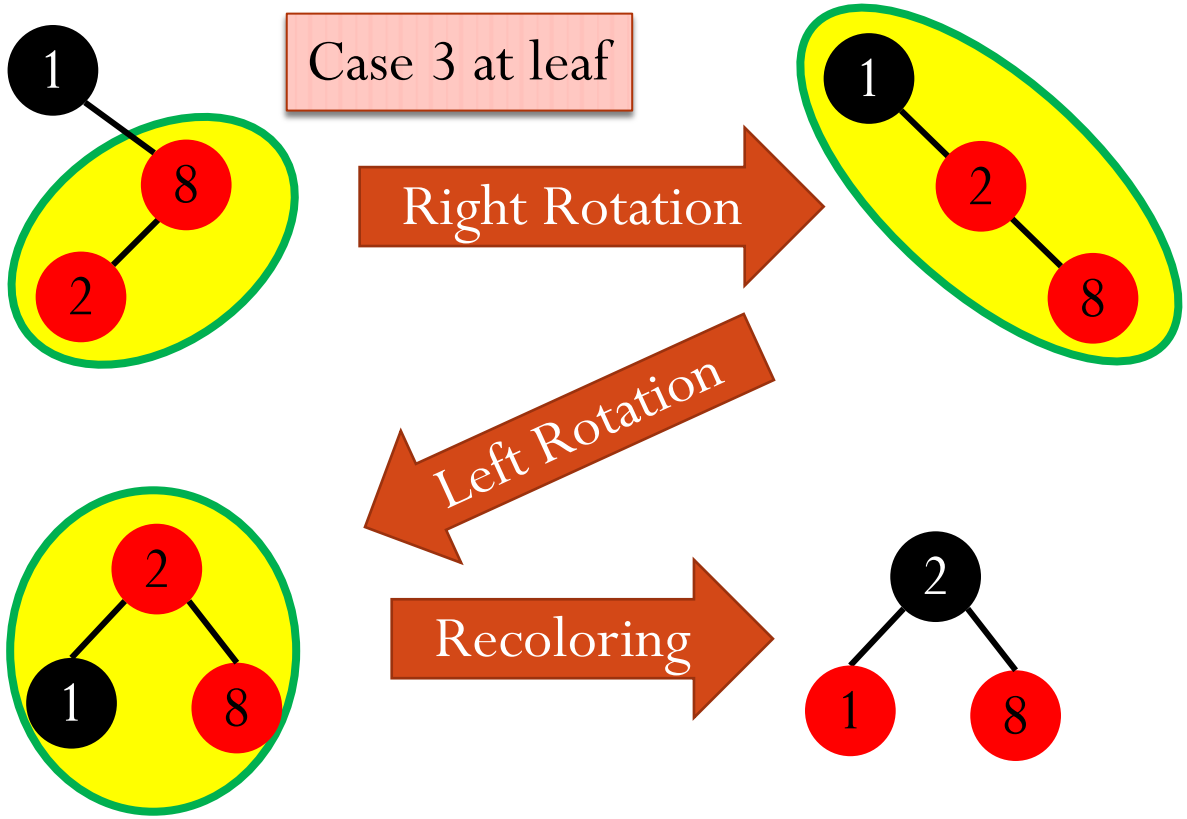
- Insert 1

- Insert 8



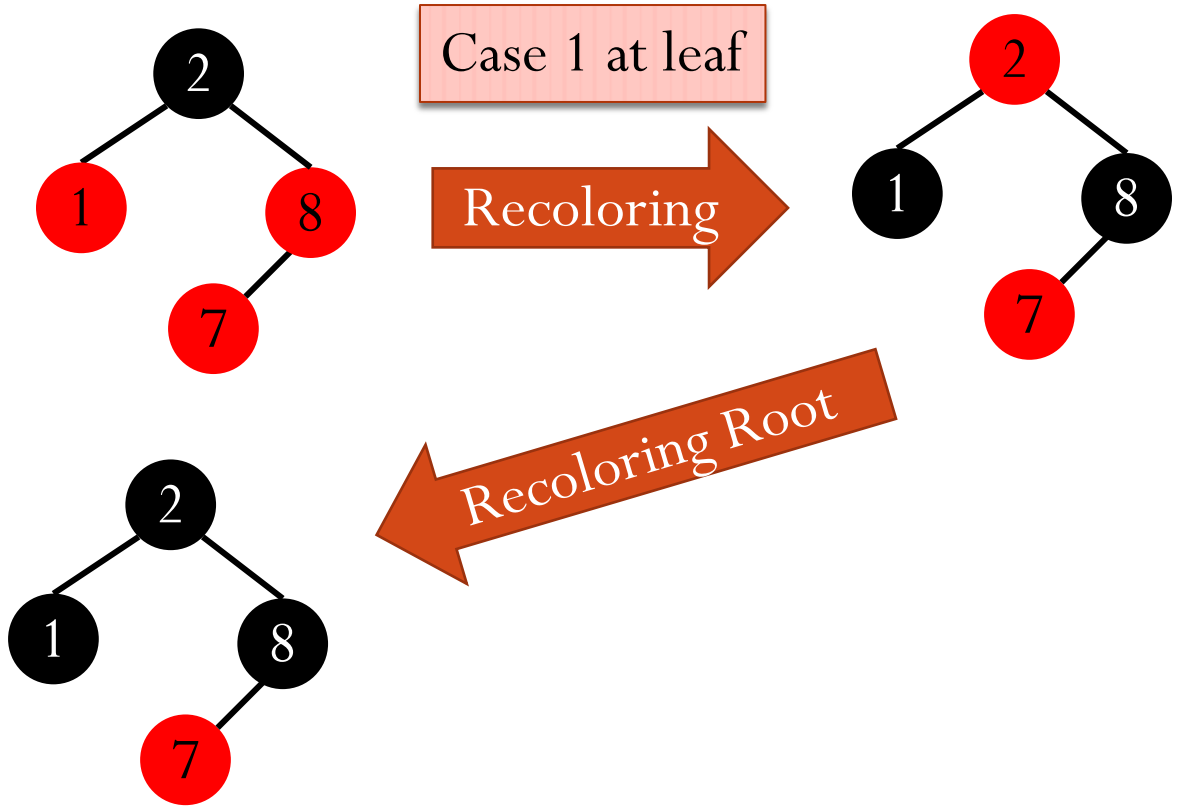
Example (cont.)

- Insert 2



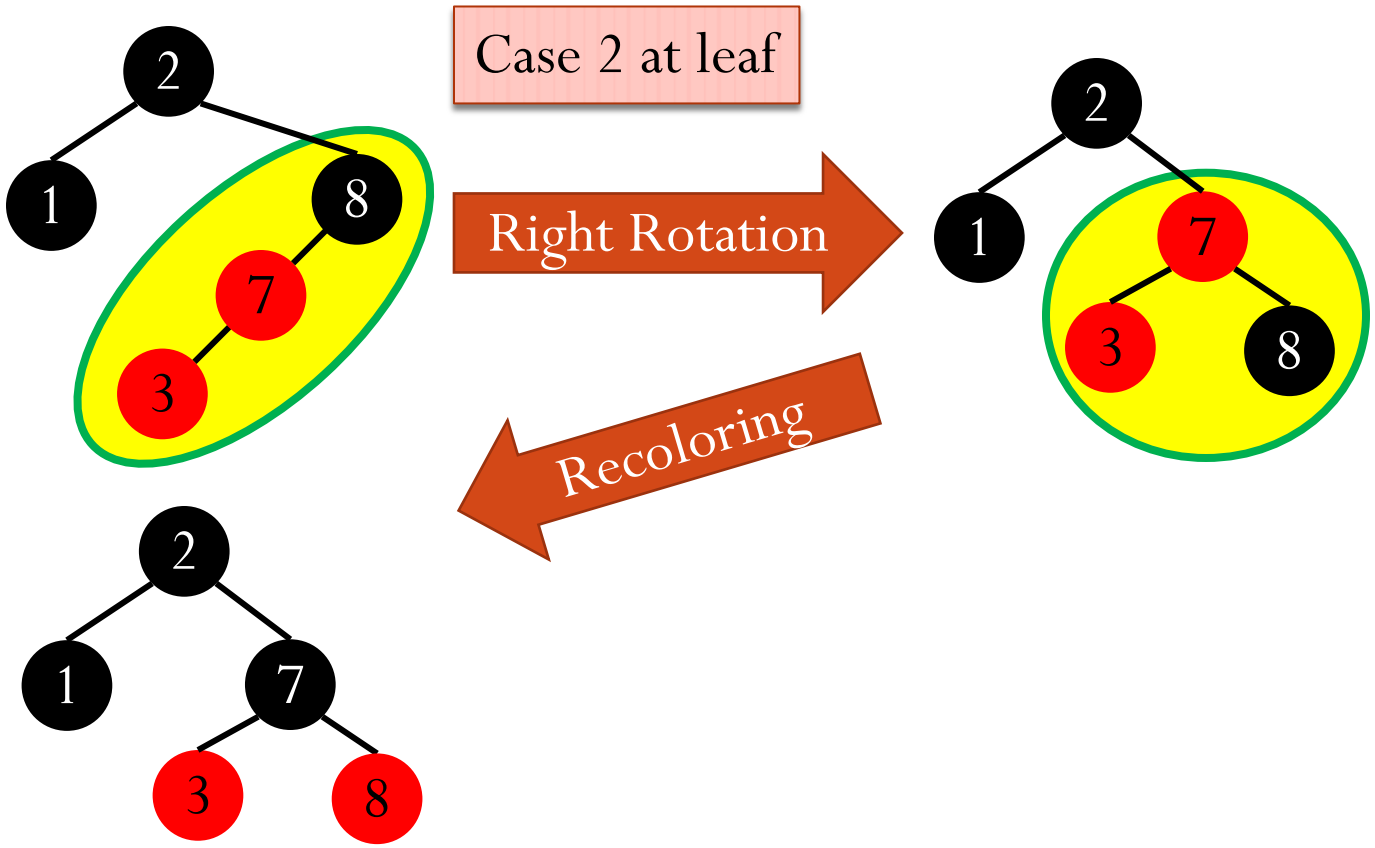
Example (cont.)

- Insert 7



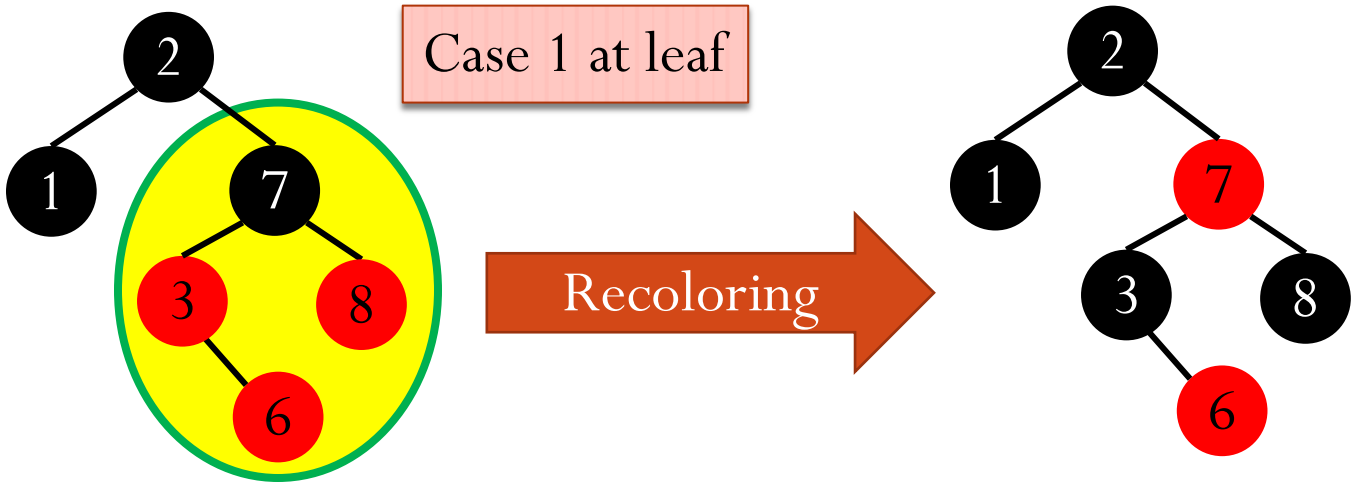
Example (cont.)

- Insert 3



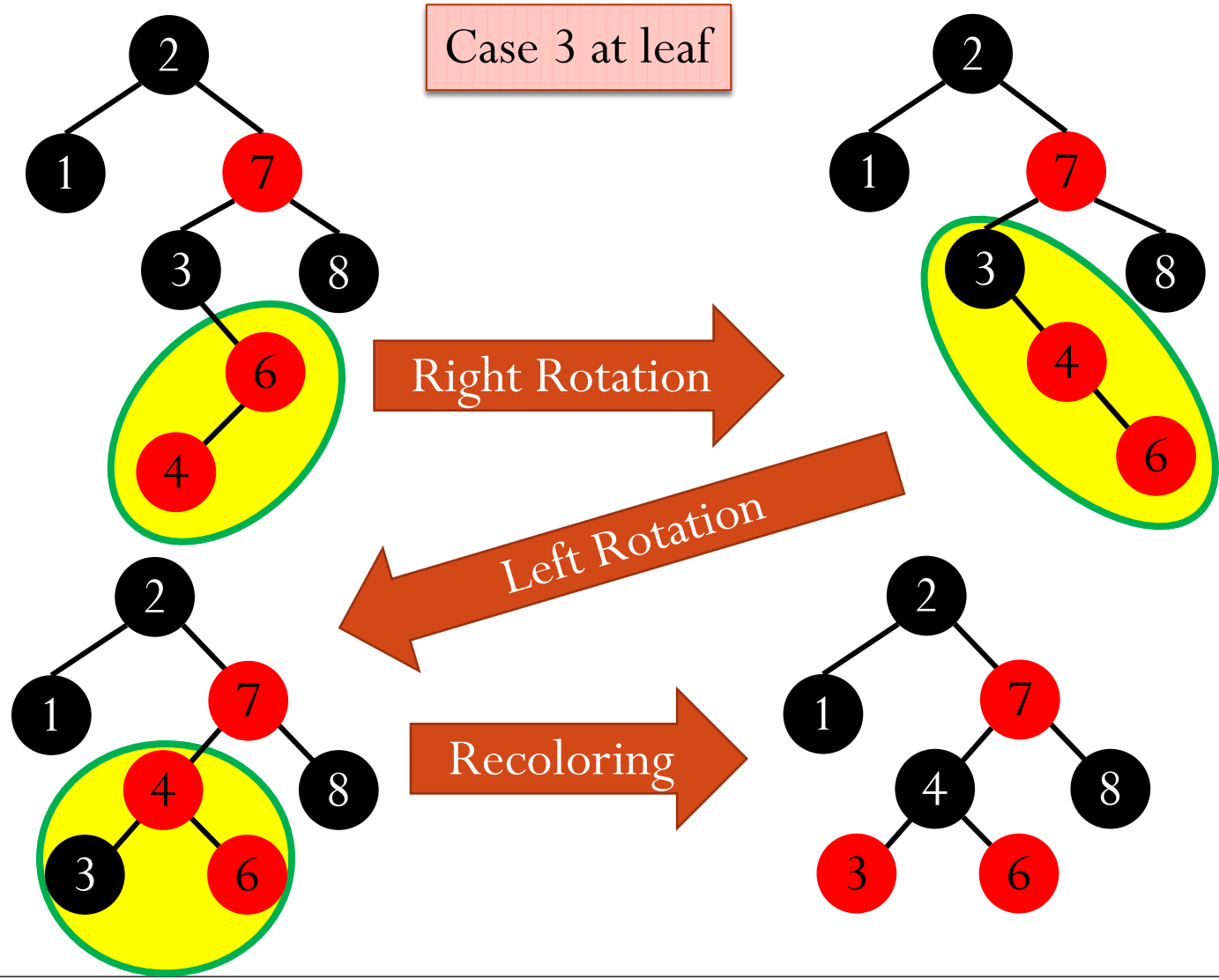
Example (cont.)

- Insert 6



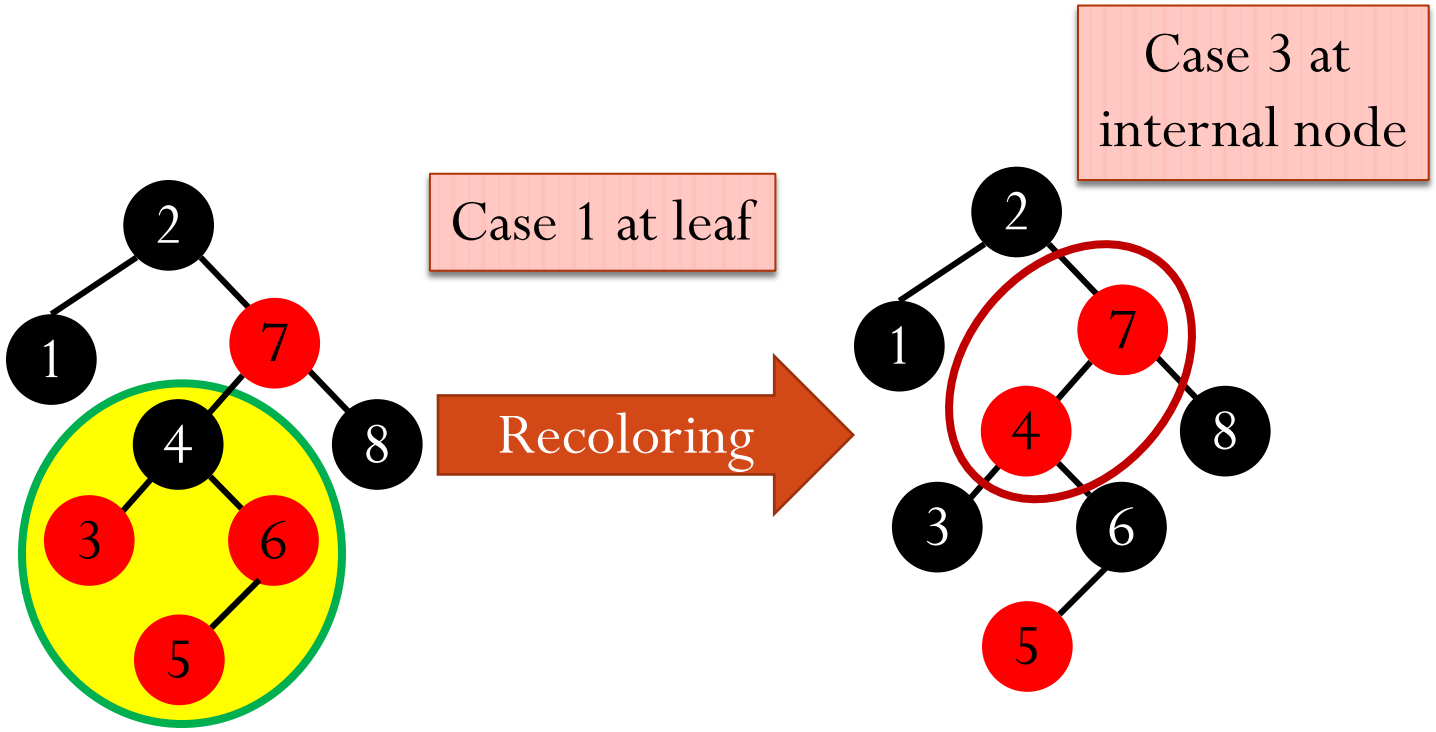
Example (cont.)

- Insert 4

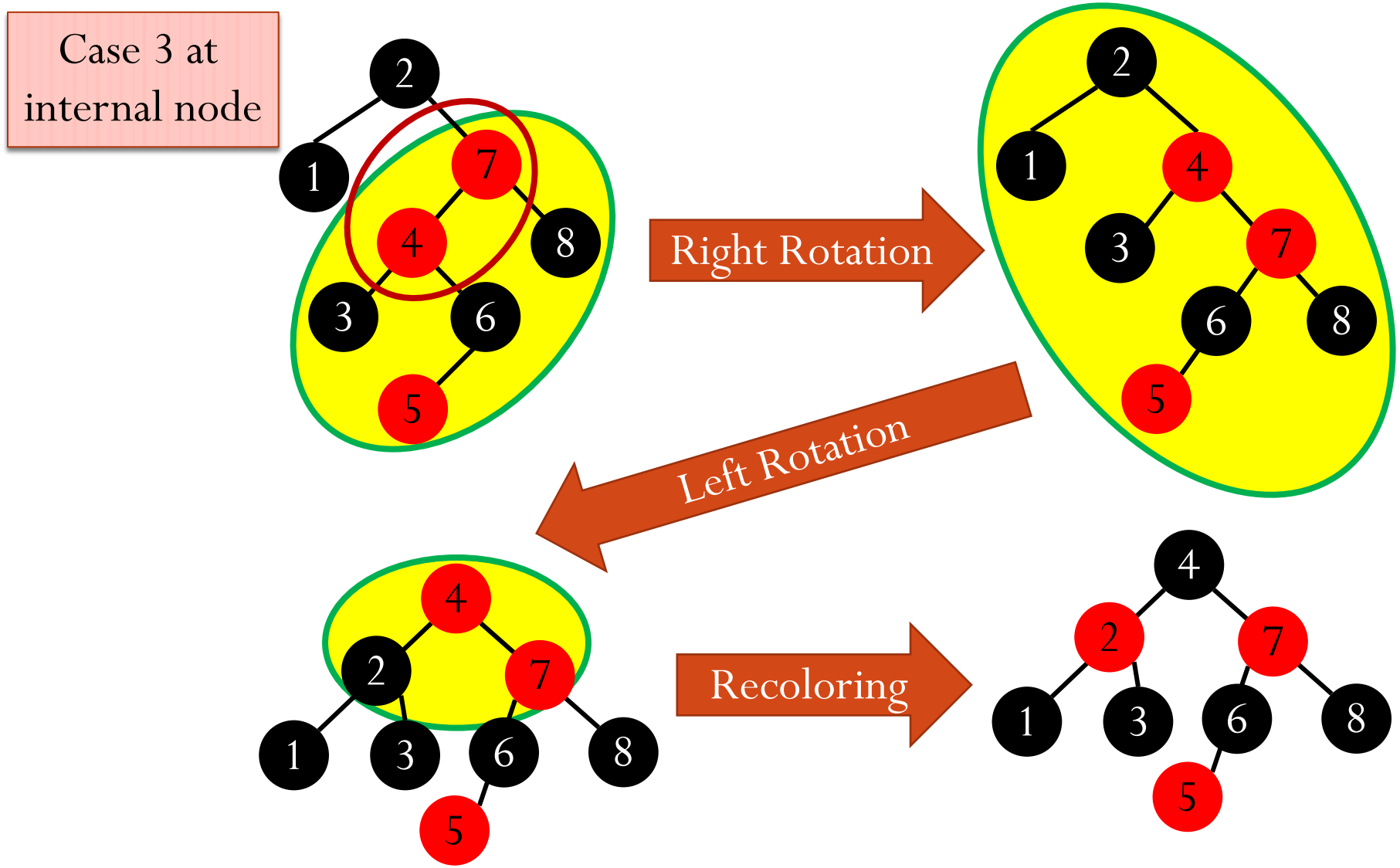


Example (cont.)

- Insert 5



Example (cont.)



Runtime Complexity

- Number of rotations required
  - For case 1, only need to recolor, **no** rotation.
  - For case 2 or 3, perform 1 or 2 rotations and terminate.
  - **Thus**: # rotations $= O(1)$.

- Number of recoloring required
  - Worst case: $O(\log n)$

- Runtime complexity is $O(\log n)$.