CS202 – Data Structures

LECTURE-11

Balanced Binary Search Trees (AVLs)

Rotations for restoring balance, AVL Operations

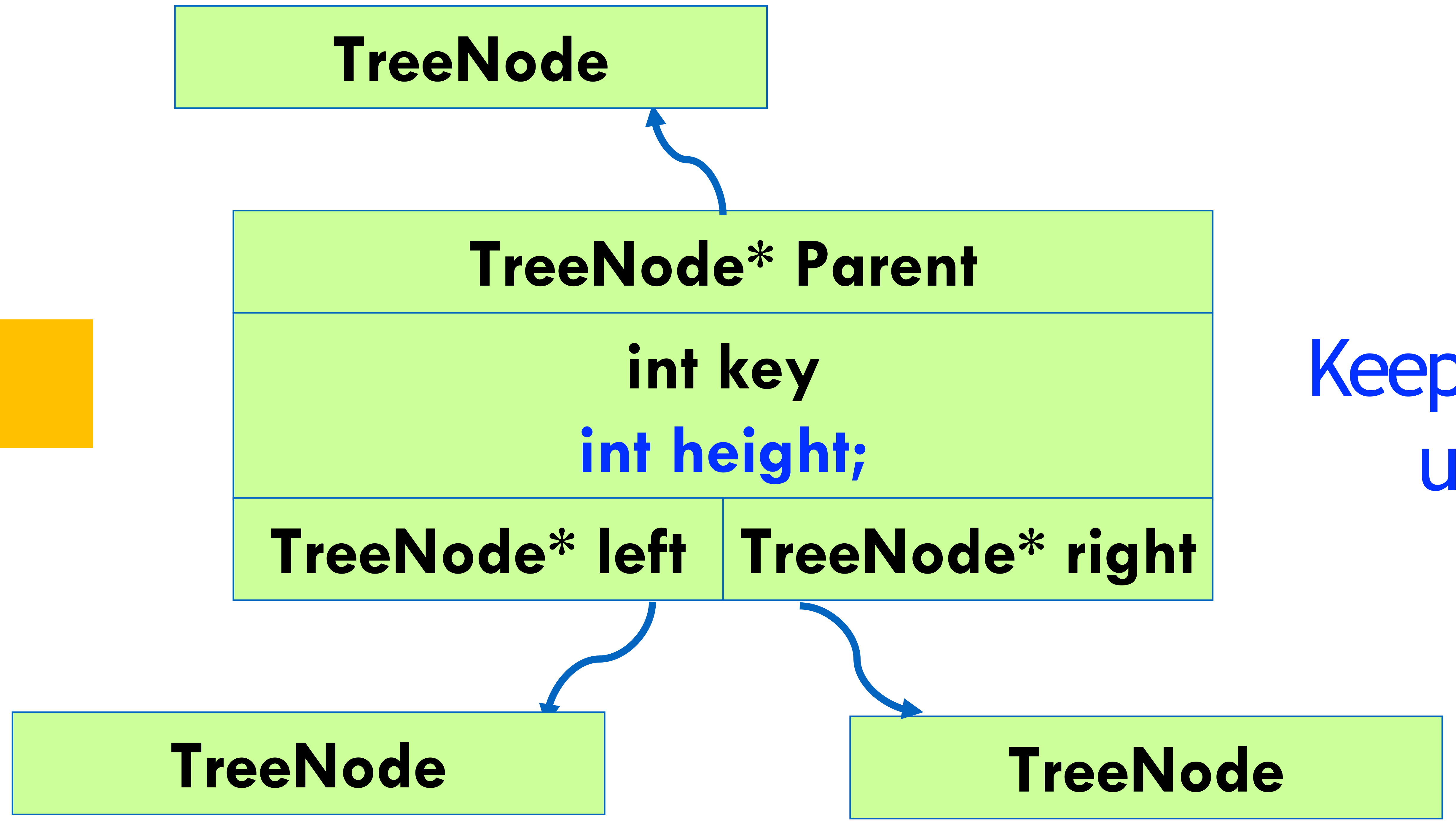
Dr. Maryam Abdul Ghafoor
Assistant Professor
Department of Computer Science, SBASSE

Agenda

- Rotations for Restoring Balance
- AVL Trees Operations
  - Search
  - Insertion
  - Deletion

Modified Binary Tree Structure

Binary Tree Node



Keep the height updated!

AVL Property

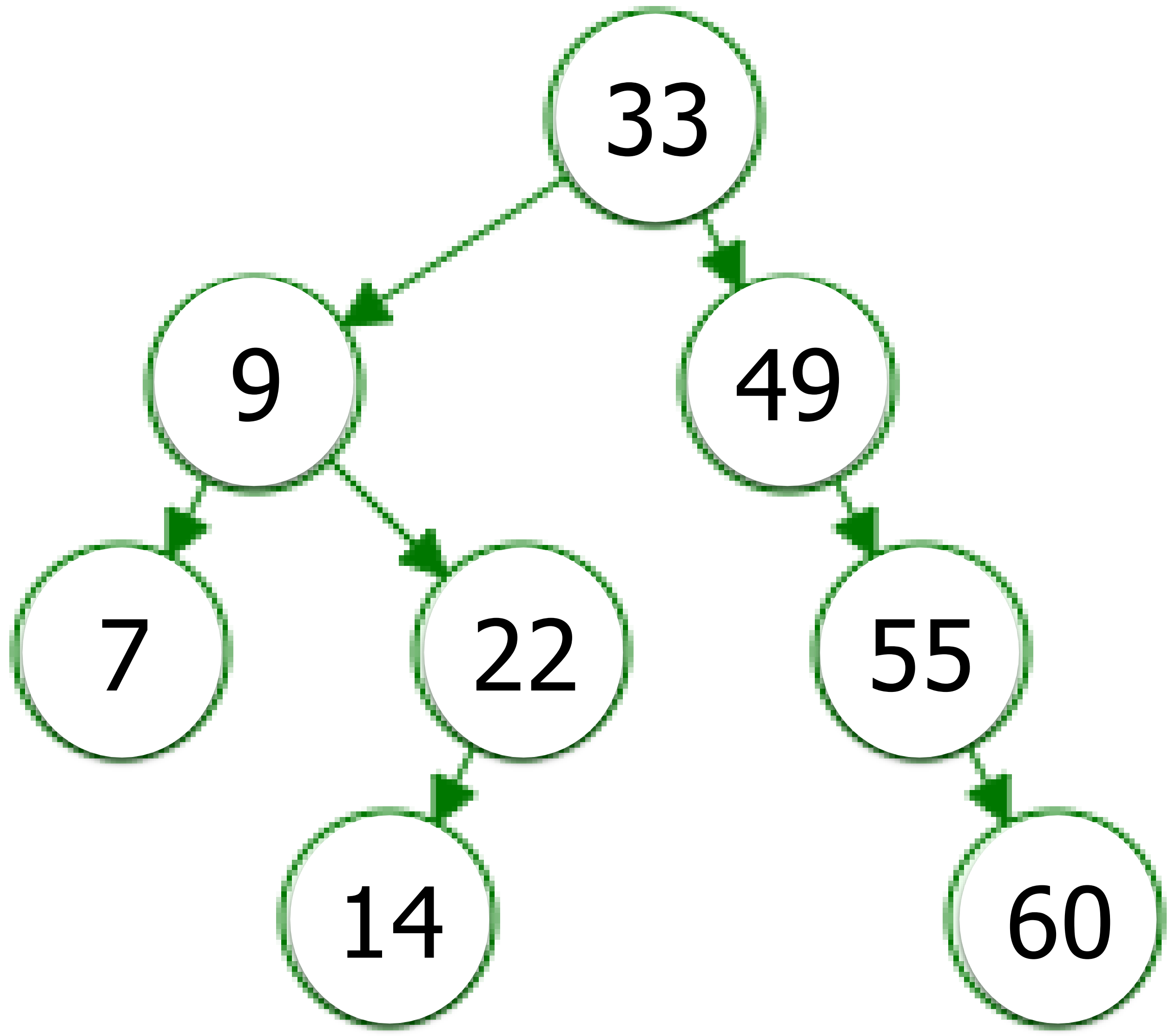
- AVLS are
  - BSTs that maintain height-balance property
  - Height Balance Property
    - For all nodes n,

$$| n \rightarrow left \rightarrow height - n \rightarrow right \rightarrow height | \leq 1$$

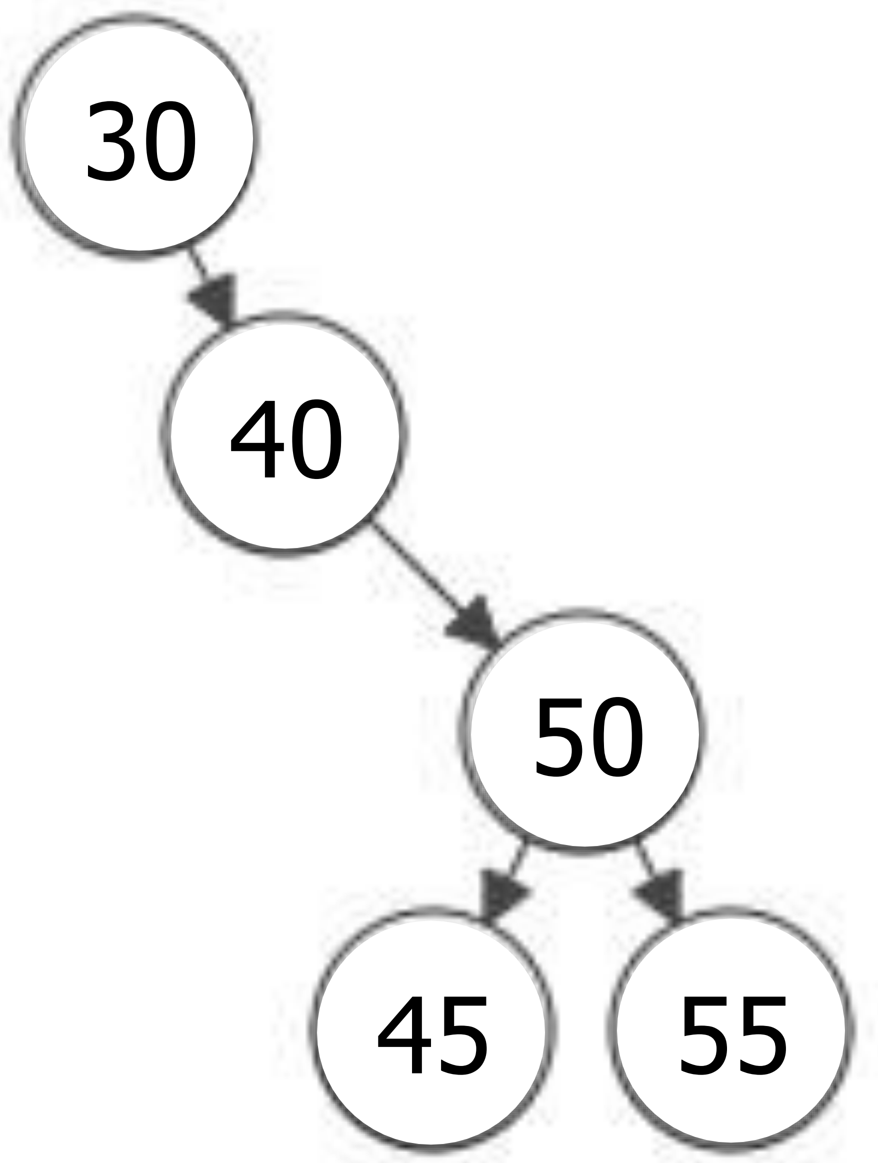
Which of the following are AVLs?

Poll

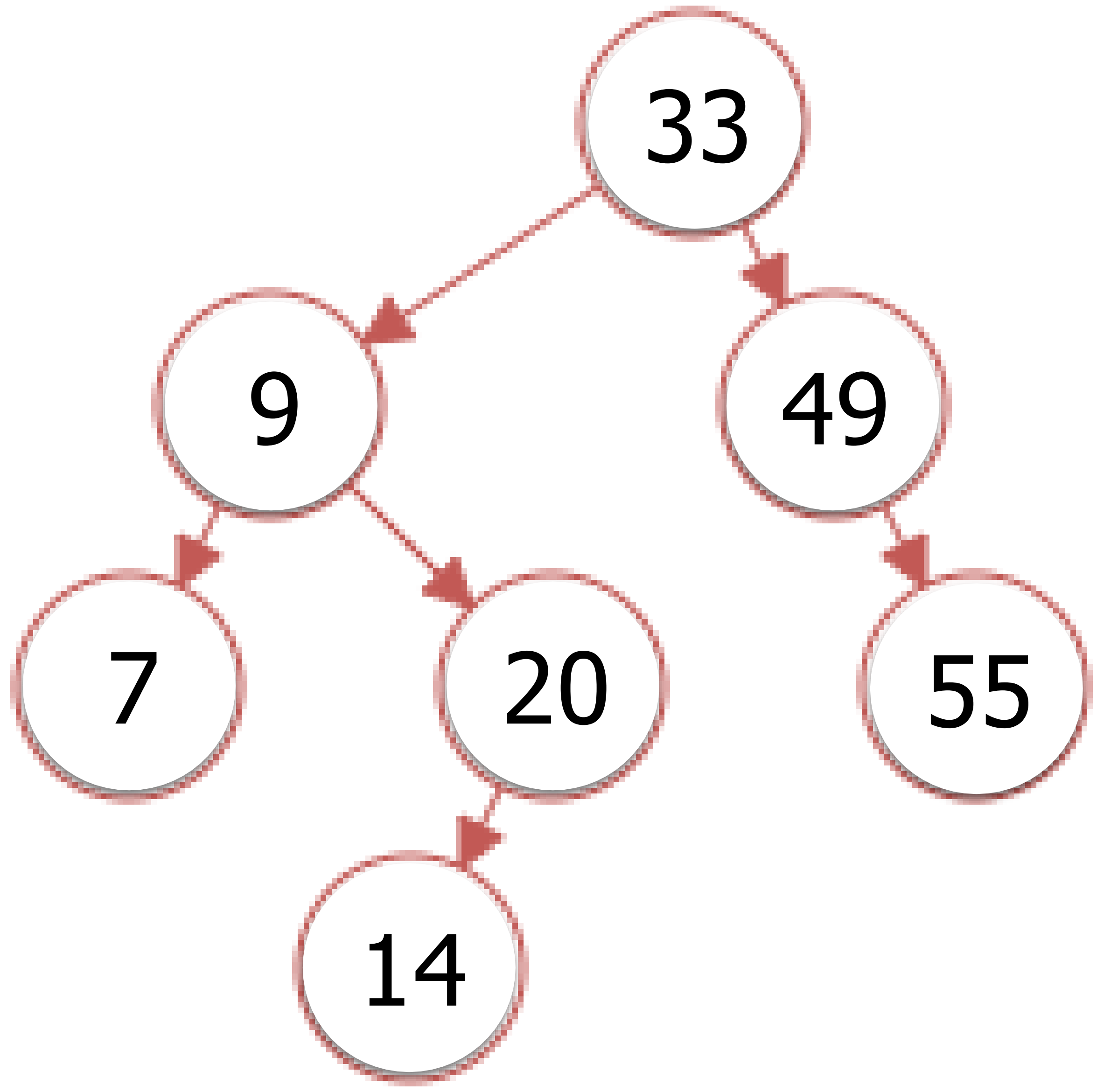
(A)



(B)



(C)



Operations in AVLS

- Search
- Insertion
- Deletion

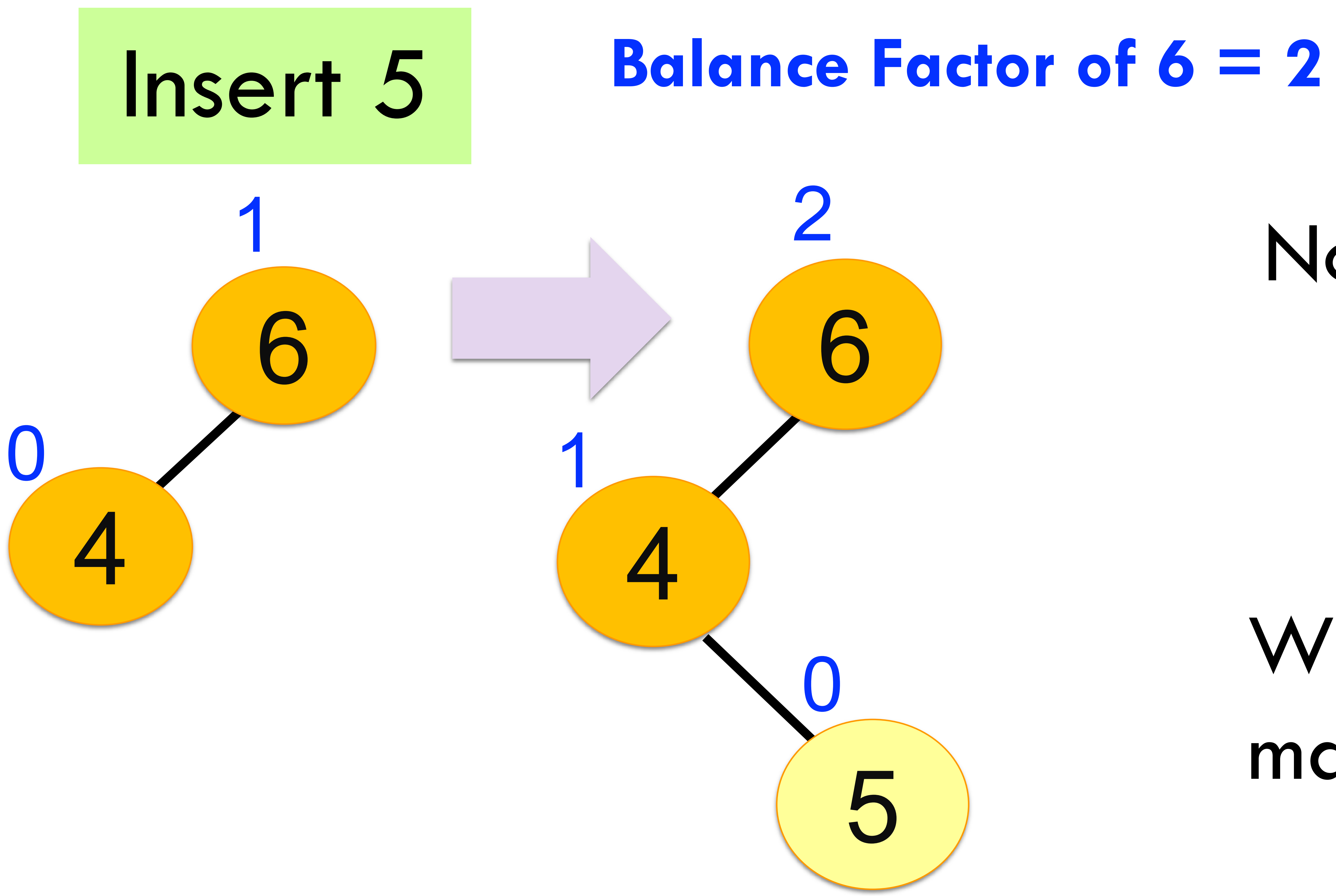
Search in AVL = Search in BST

```
Node* BSTSearch(Node* n, int key){ //Node is BSTNode, we are looking for key
   if( n→k == key)
      return k;

  else if( n→ k > key )
      return BSTSearch(n→left , key);

   else
      return BSTSearch(n→right , key);

}
```

Insertions in AVL

- Insert as in a simple BST
- Work your way up the tree, restoring AVL property
  - Suppose x is the lowest node **violating** the AVL property
    - Ask, "Is x right heavy or left heavy?" then ask, "Is x's child right heavy or left heavy?"
  - How to restore balance?
    - Single Rotations
    - Double Rotations

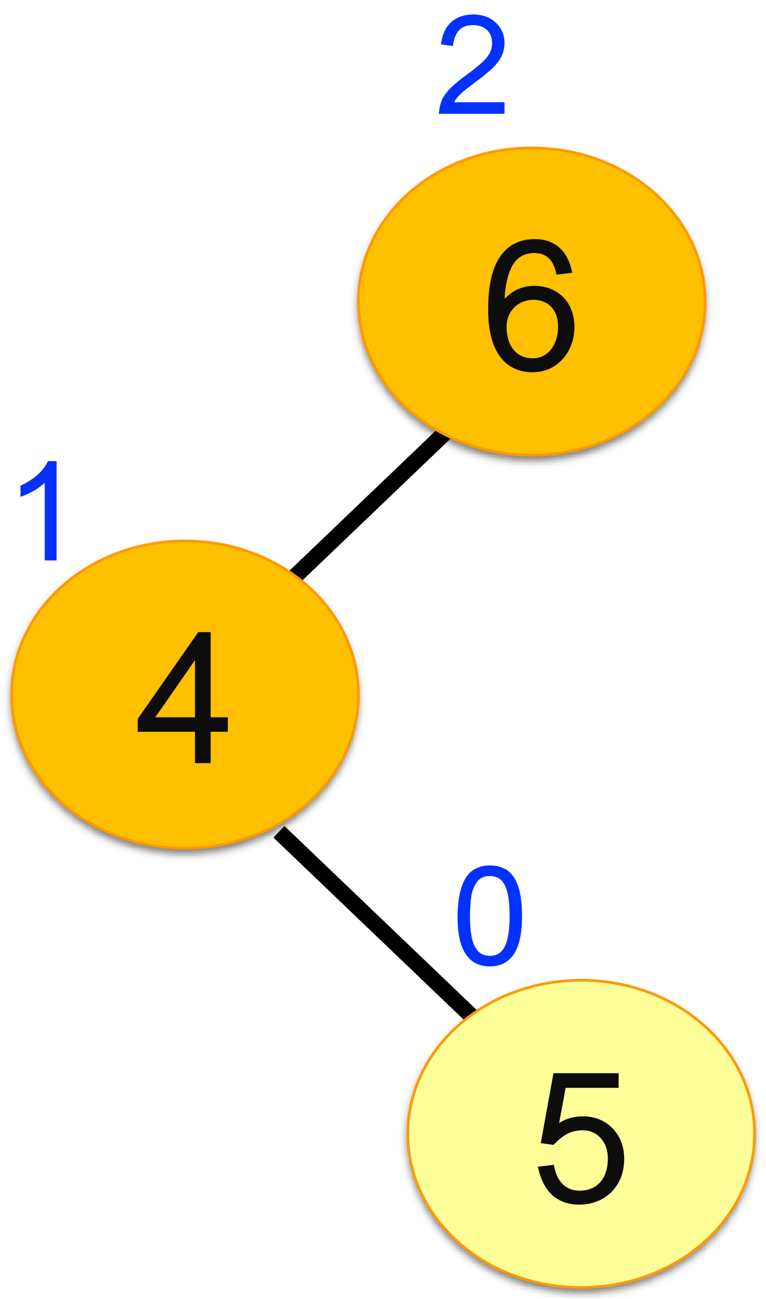
Restoring Balance



Node **6 violates** the **AVL property.**

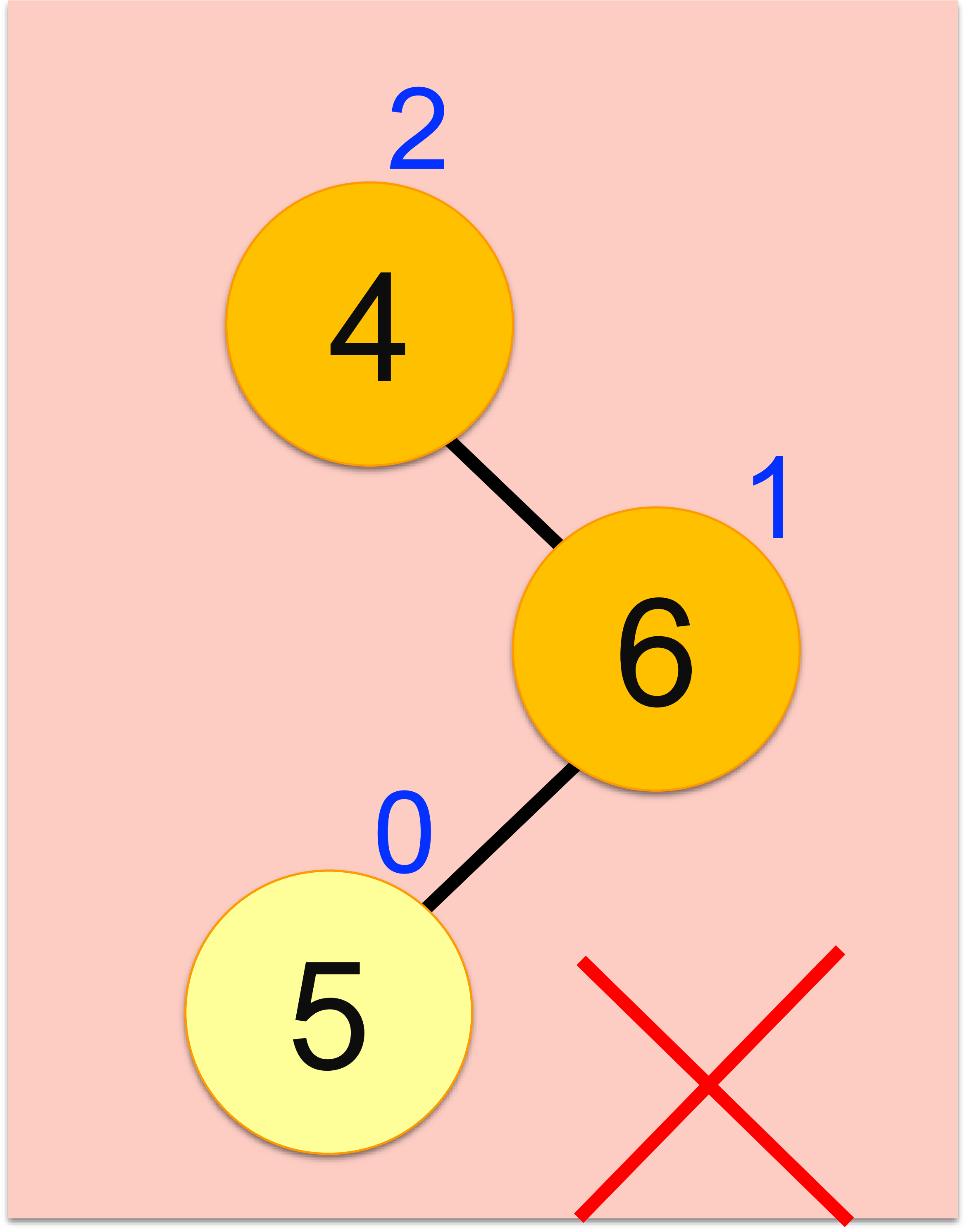
What are some possible options for making this tree balanced?

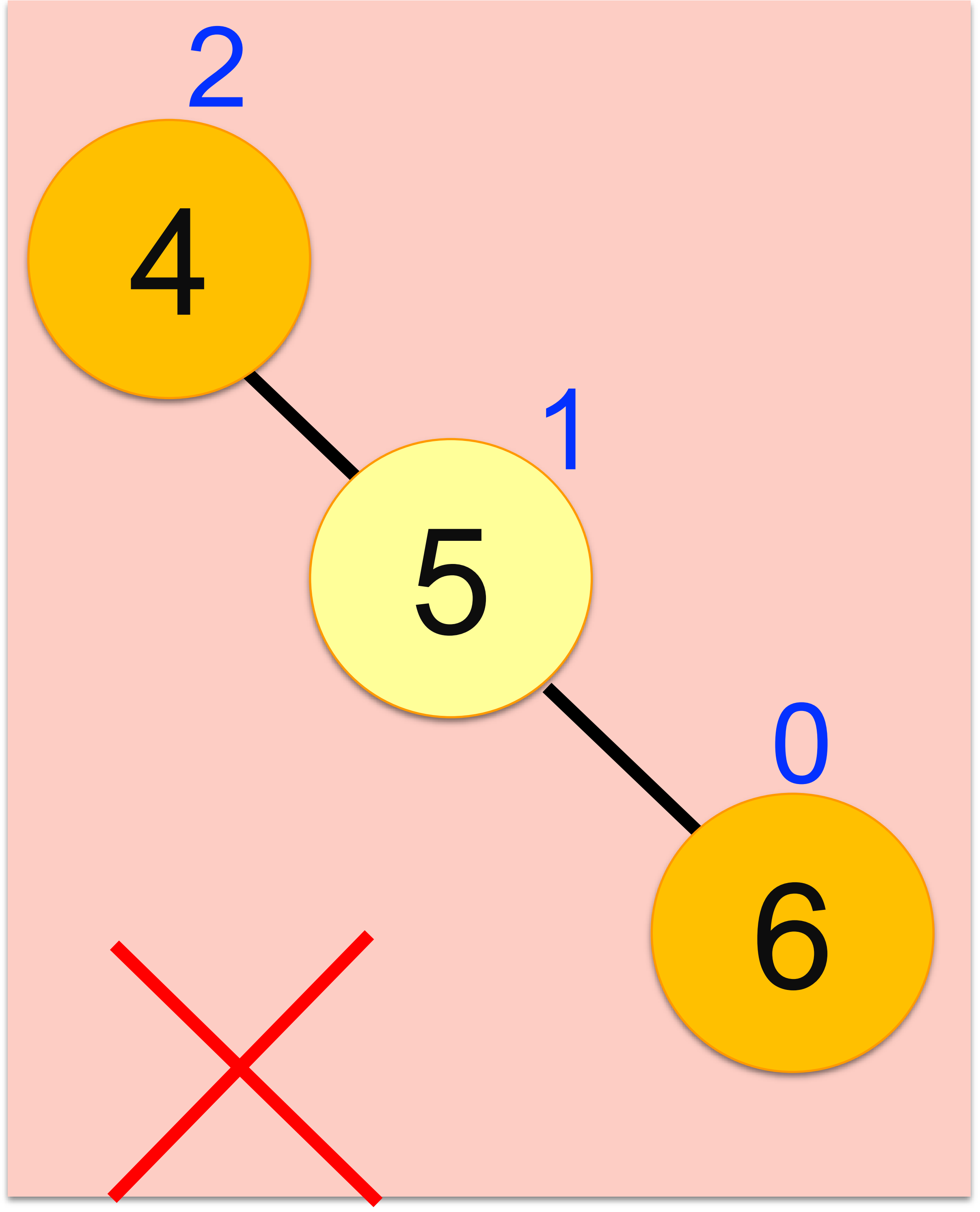
Restoring Balance

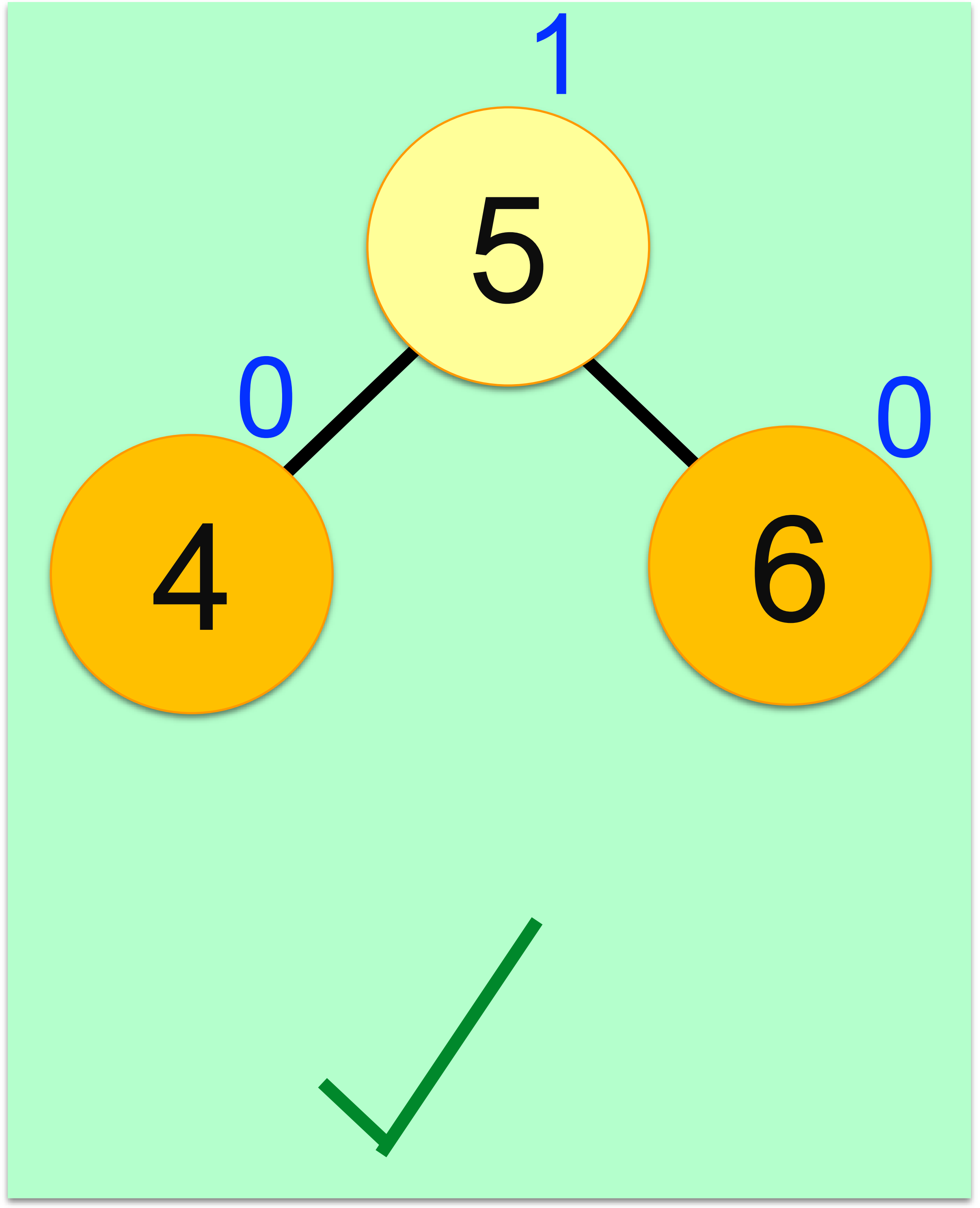


Balance Factor of 6 = 2

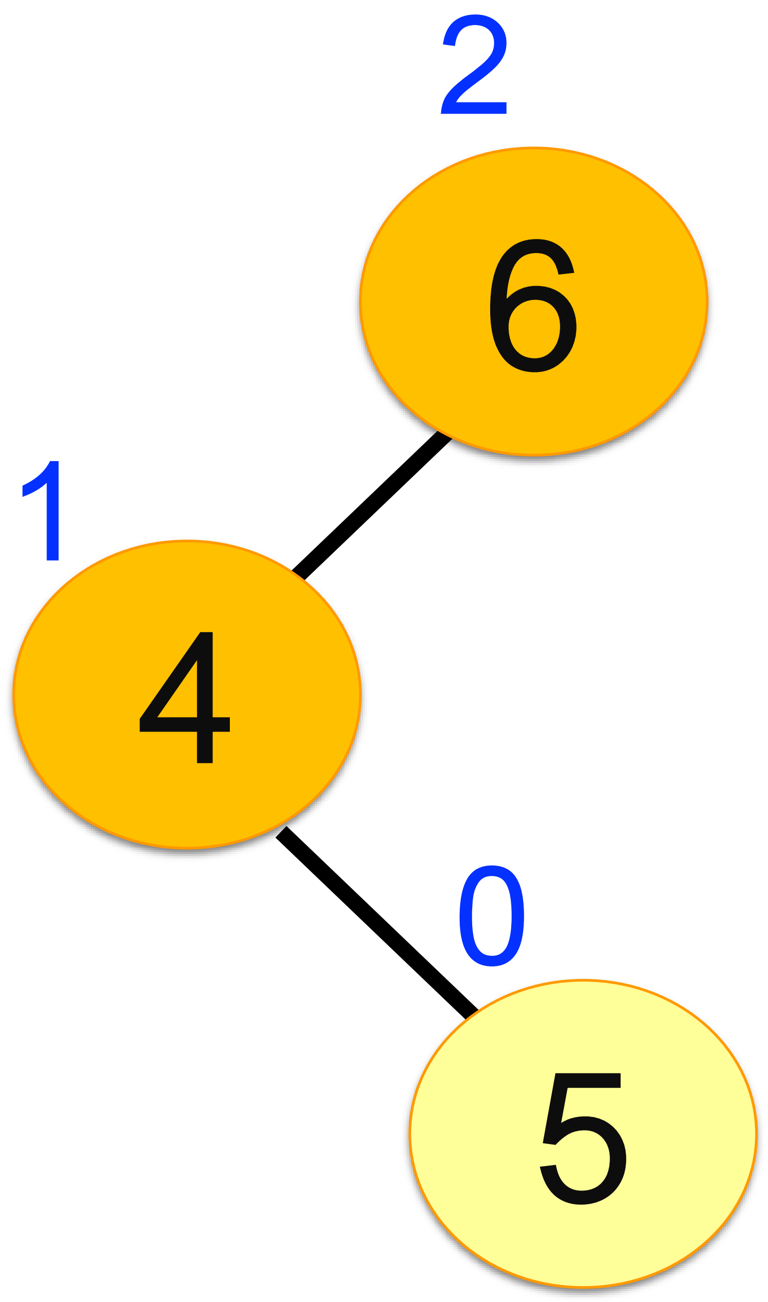
What are some possible options for making this tree balanced?



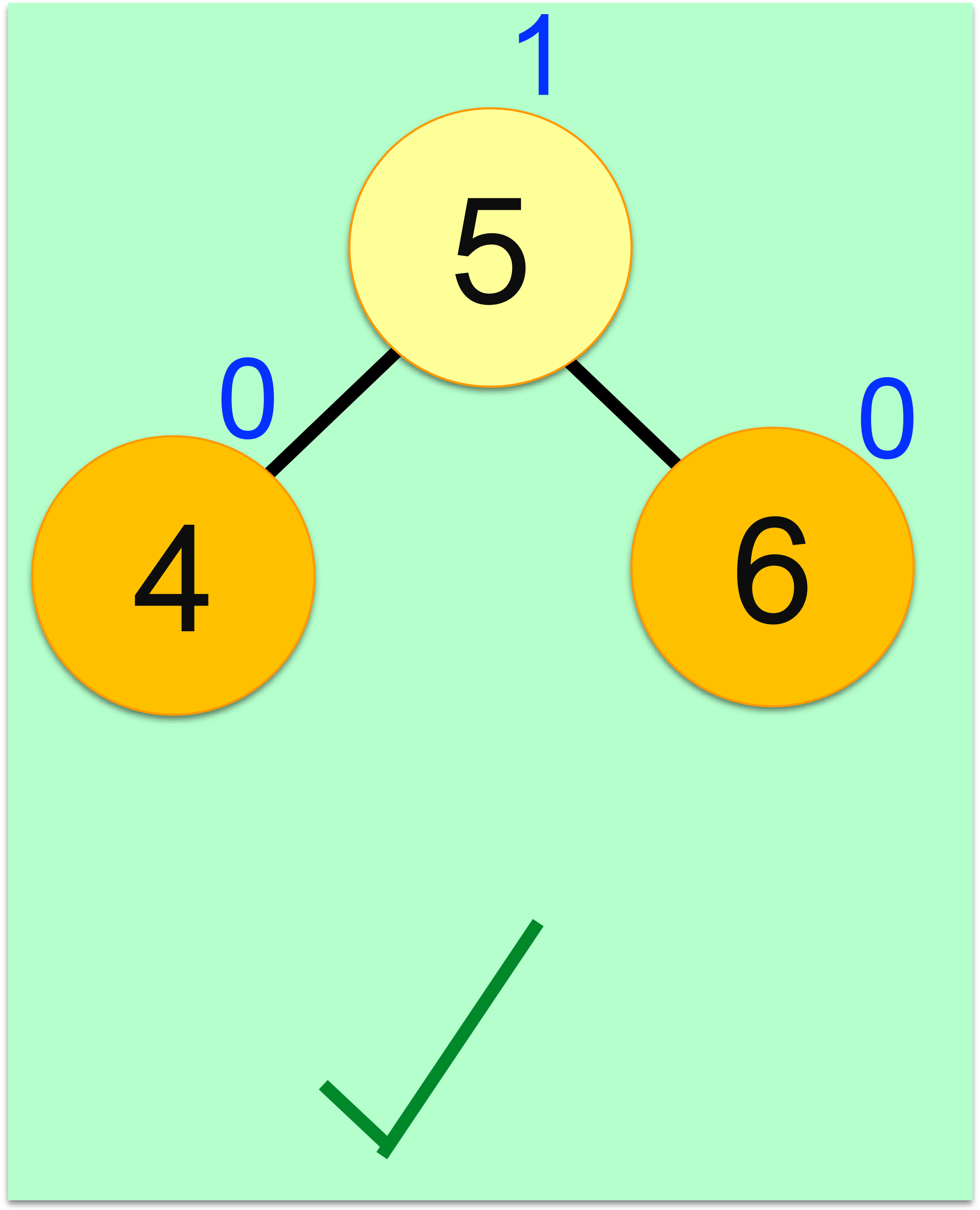




Restoring Balance



Balance Factor of 6 = 2



We need two steps

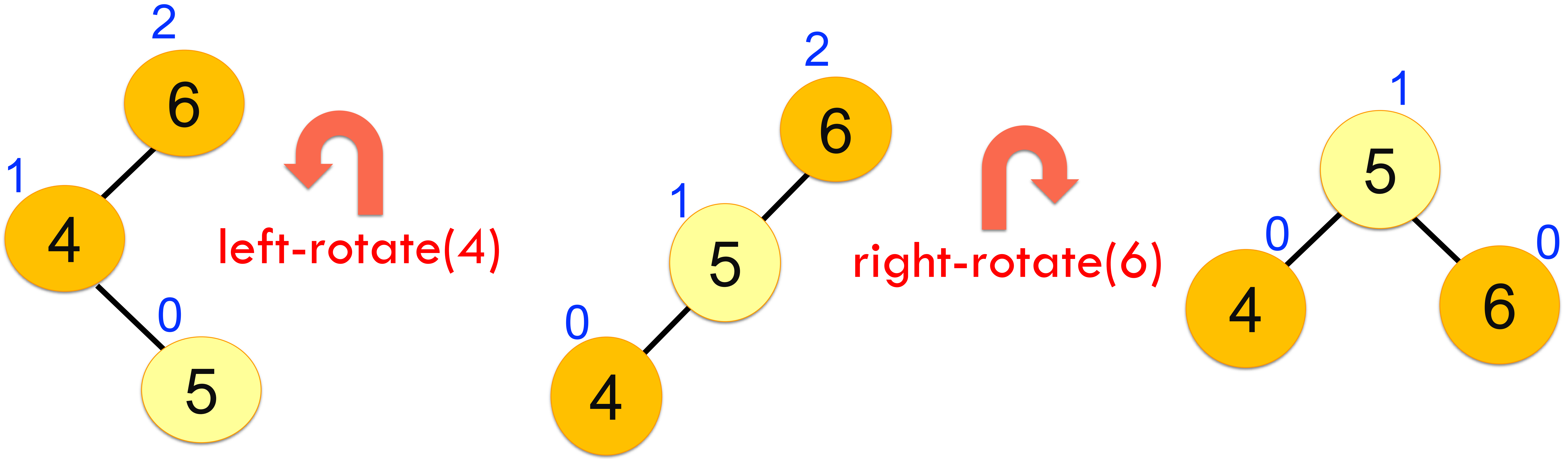
left-rotate(4)
right-rotate(6)

apply on the original tree

Restoring Balance

Node 6 violates the AVL property.

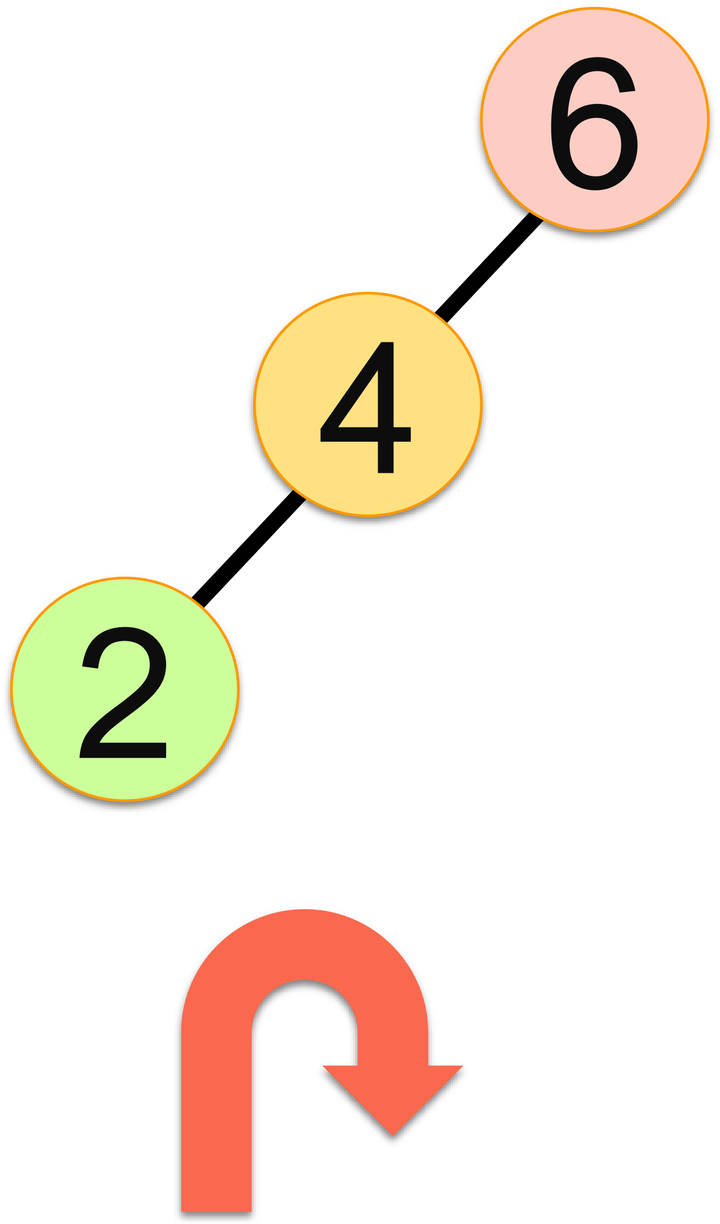
Balance Factor of 6 = 2



… but how do we get here sytematically?

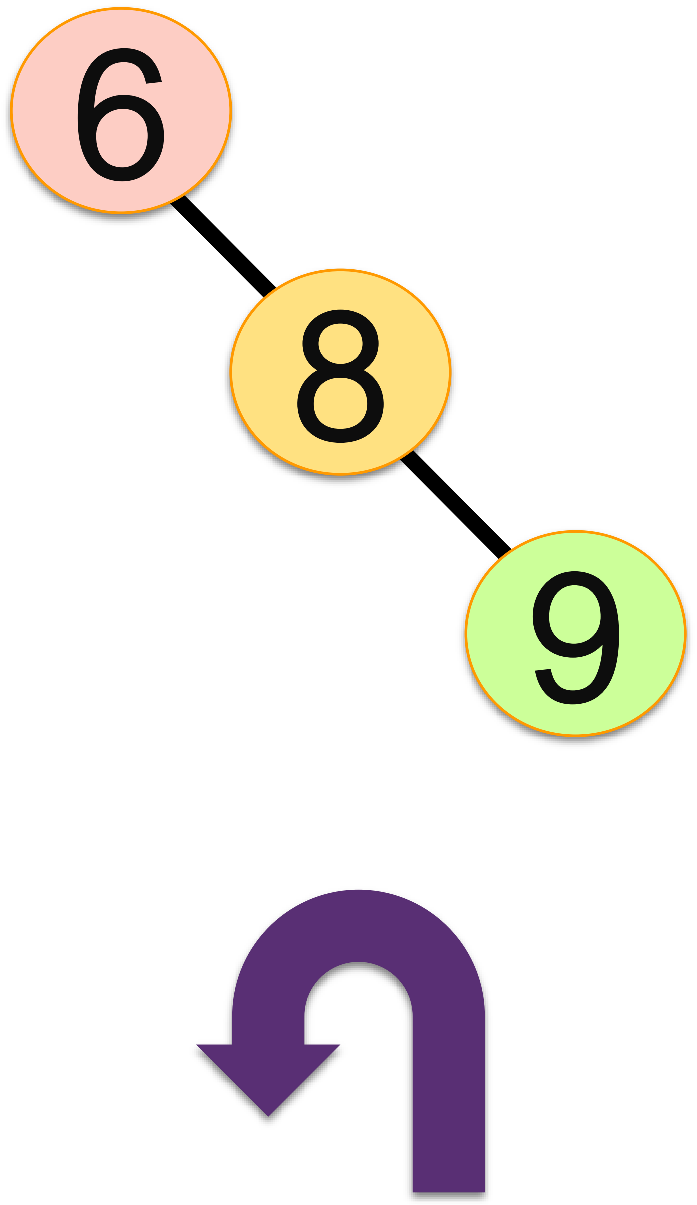
AVL Trees – Four Cases for Restoring Balance

Left-Left



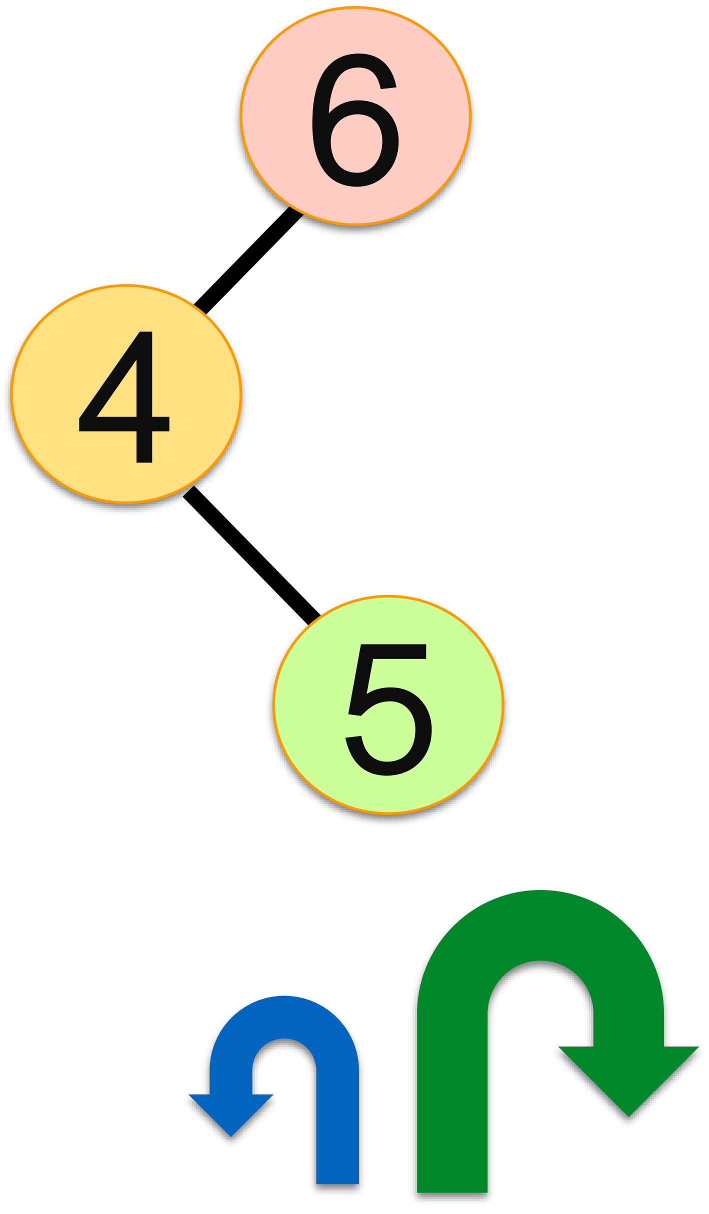
Right-rotate(6)

Right-Right



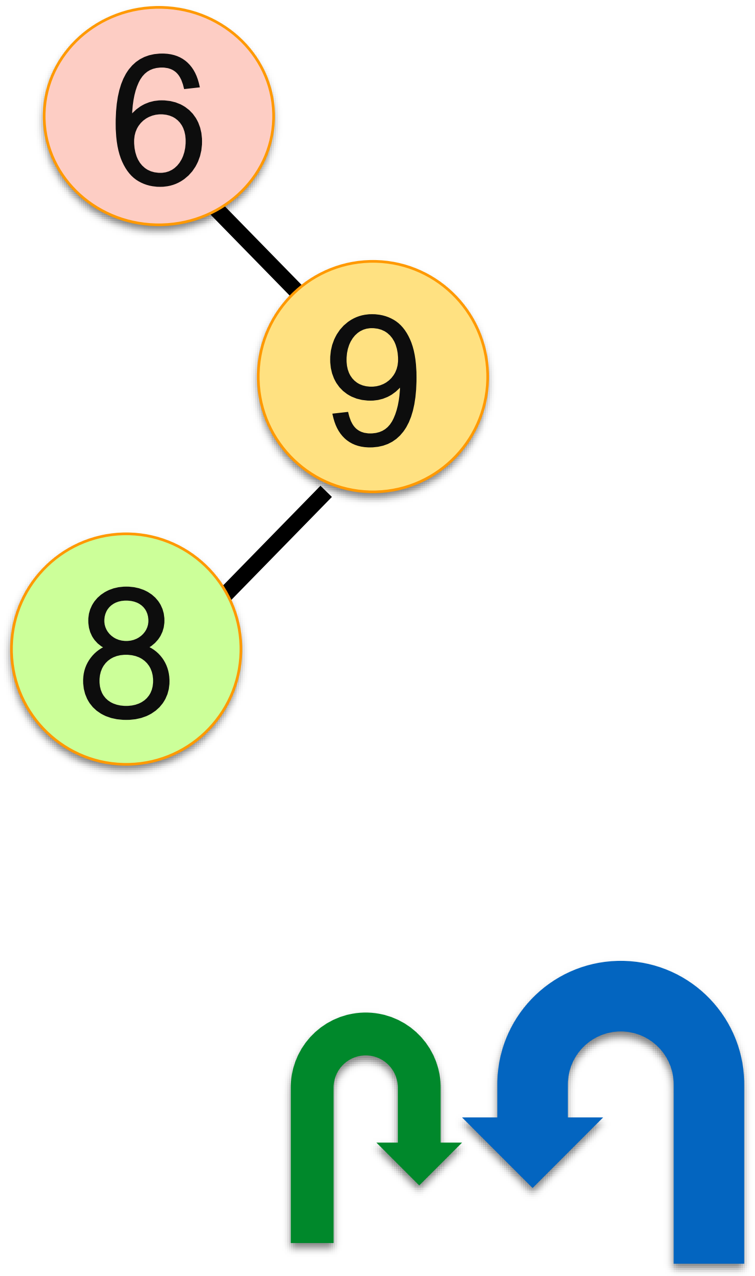
Left-rotate(6)

Left-Right



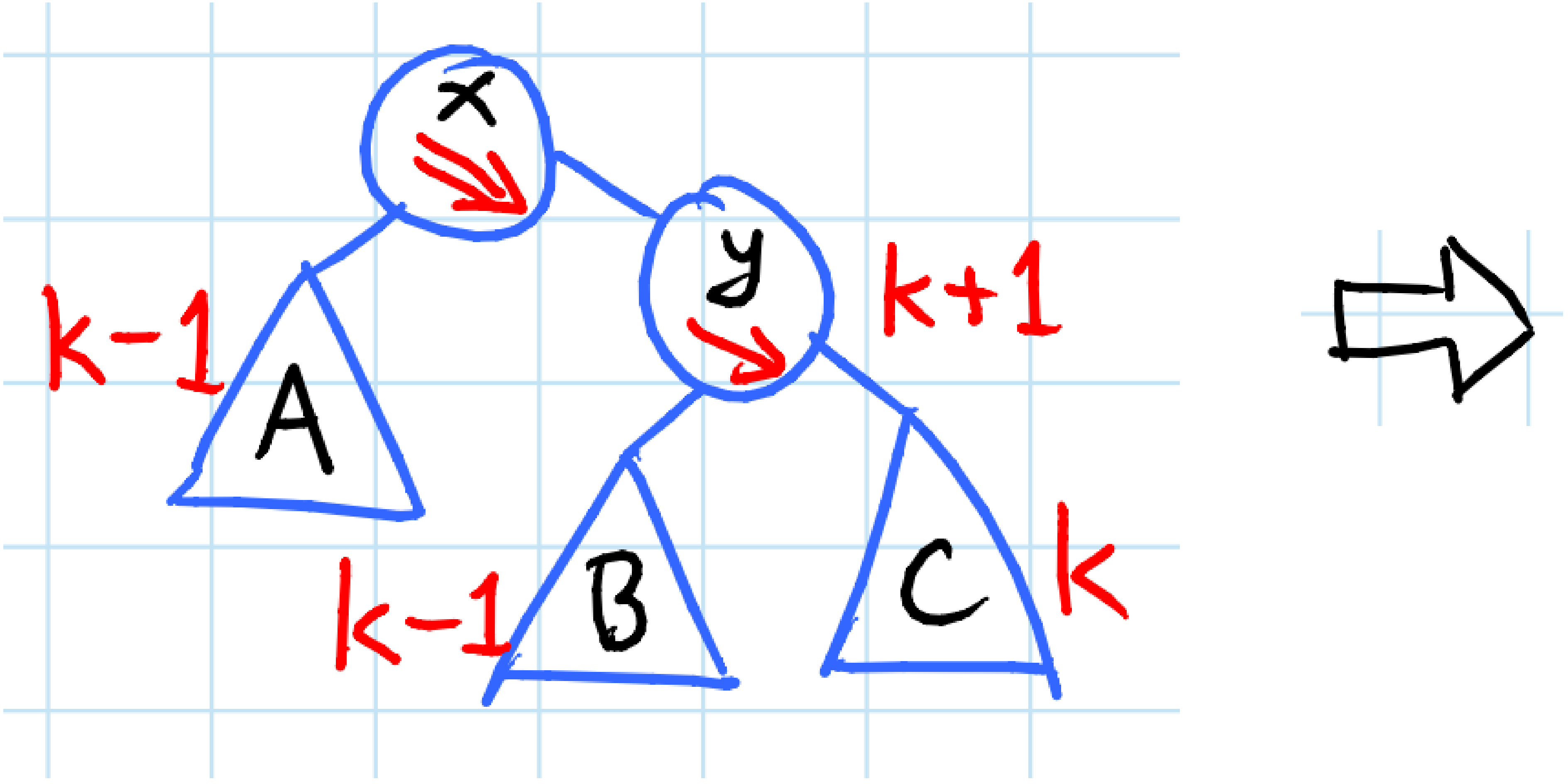
Left-rotate(4)
Right-rotate(6)

Right-Left



Right-rotate(9)
Left-rotate(6)

Generalizing (right-right case)



- Rotate left on x to restore balance

Hint: Recall that BSTs store keys in sorted order!

A < x < B < y < C

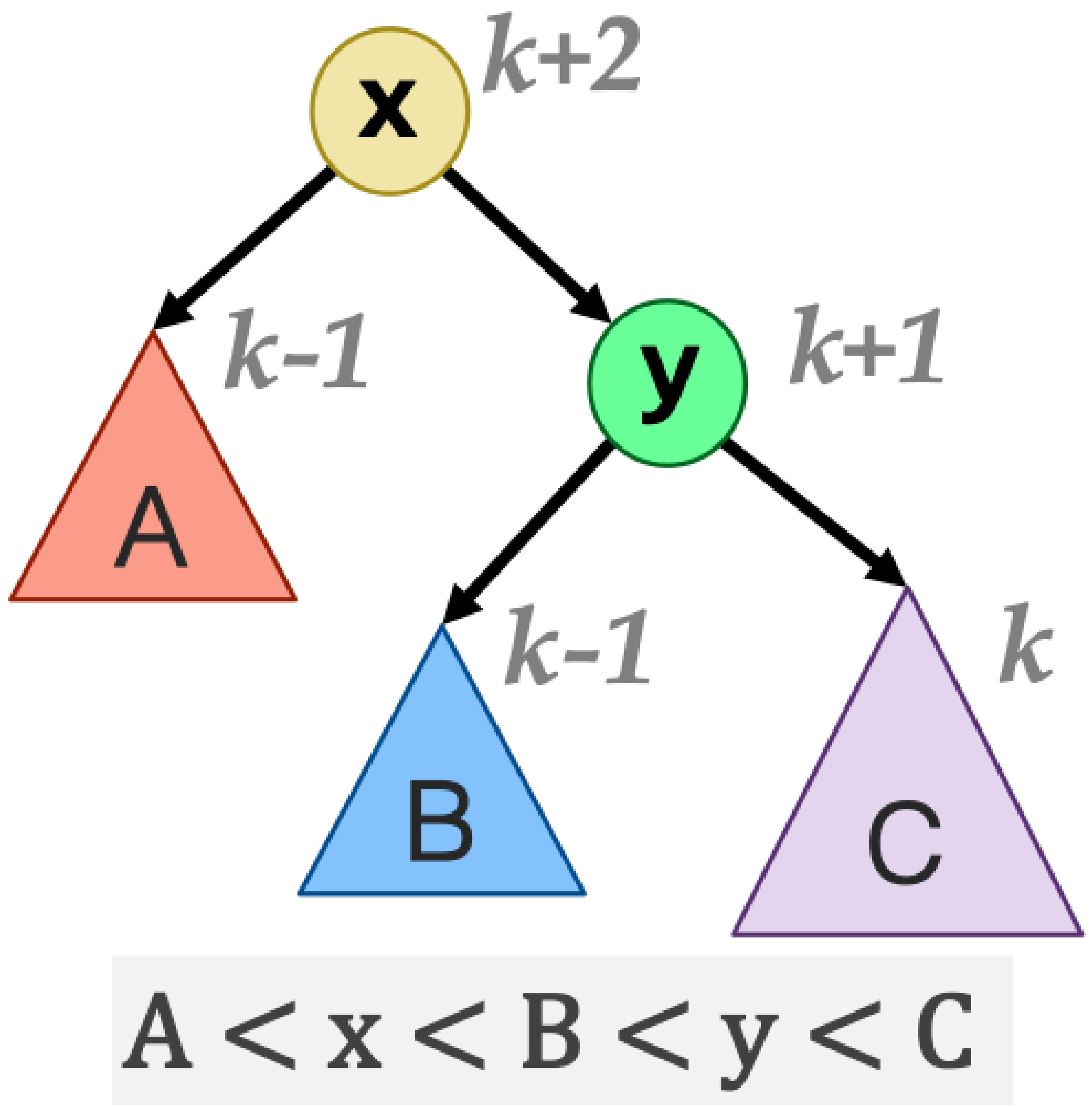
Verify using an inorder traversal on the resulting BST

left-left case is symmetric

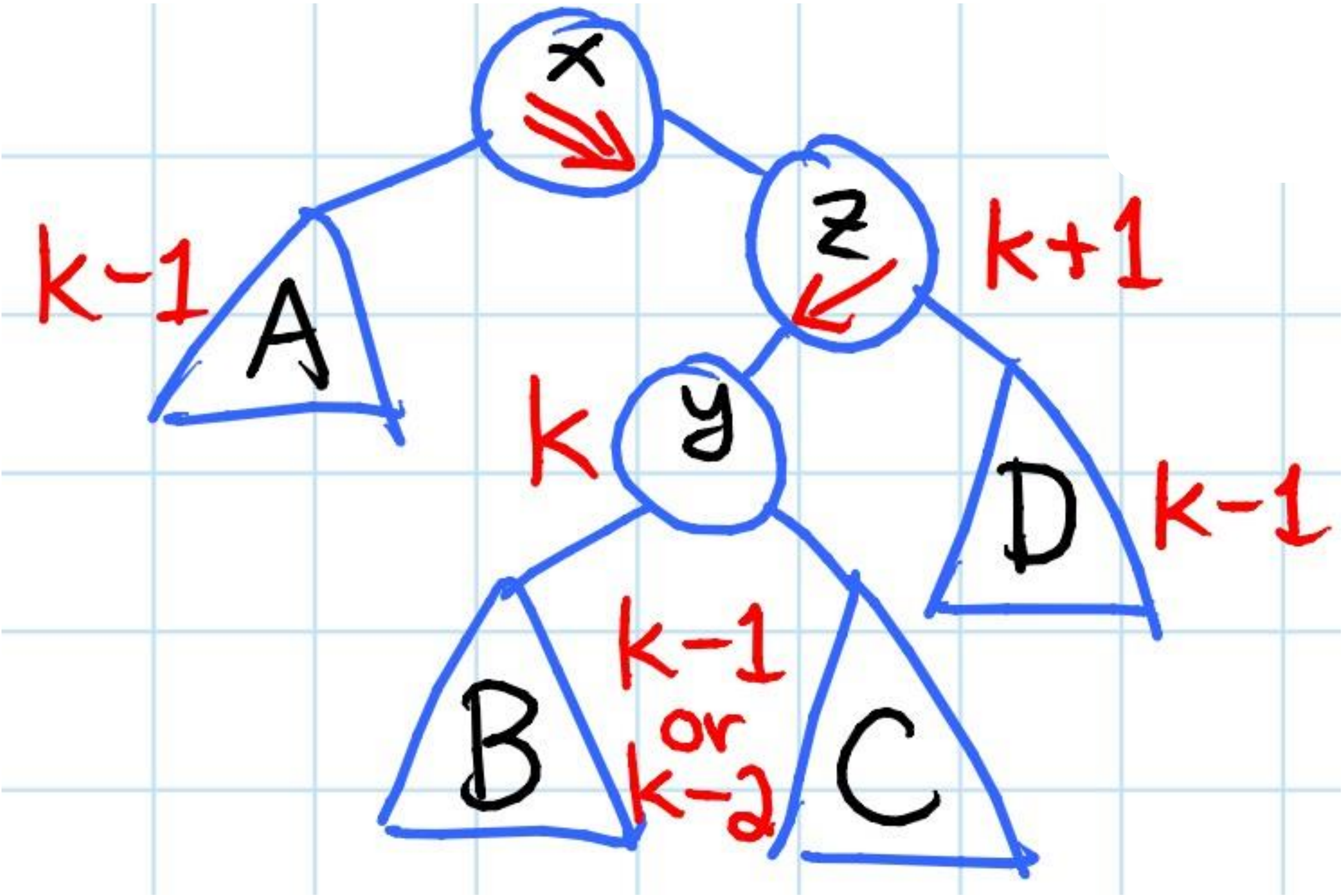
How to Perform Rotation?

Time Complexity – O(1)

```
Node* rotate(node* x){
    Node*y = x → right;
    Node* B = y → left;

   // Perform rotation
     y→left = x;
     X→right = B;

    // Update heights
    X→height = max(height(x→left), height(x  →right)) + 1;
    y→height = max(height(y→left), height(y→  right)) + 1;


  return y;
}
```



Generalizing (right-left case)



- Rotate right on Z
- Rotate left on x

A <x <B <y <C <z <D

left-right case is symmetric

Insertion in AVL

1. Insert new node n in a tree using BST insertion algorithm
2. Update height of n
3. Compute balance factor of n and check for imbalance
   - If tree is unbalanced, perform rotations
   - There are Four possible cases
     - Left-left case → rightRotate(n)
     - Right-right case → leftRorate(n)
     - Right-left case → rightRotate(n→right) then leftRotate(n)
     - Left-Right case → leftRotate(n→left) then rightRotate(n)

Insertion in AVL

```
Node* BSTInsert(Node* n, int key){ // key = new value to be inserted in tree

    if( n == null)
        //create and return new node
  else if( n→ k > key )
      n→left = BSTInsert(n→left , key); //recurse left
  else
        n→right = BSTInsert(n→right , key); // recurse right

    updateHeight(n);
    int bf = getBalanceFactor(n);
    if(bf > 2 && FindBalanceFactor(n→left) > 0 )// 1. Left-left case
         RotateRight(n);
     //Similarly add checks for remaining three cases here
     // 2. Right-Right case, 3. Left-Right case, 4. Right-Left case

  return n;
}
```

Helper Functions

```
void UpdateHeight(Node* n){
//sets the height of the node (n) to the 1+max(height of its children)
//code here


}


int getBalanceFactor(Node* n){
// returns the balance factor of node, i.e., difference between the heights of left-subtree and right-subtree
//code here


}
```

Helper Functions – Time Complexities

```
void UpdateHeight(Node* n){  O(1)
//sets the height of the node (n) to the 1+max(height of its children)
//code here


}


int getBalanceFactor(Node* n){  O(1)
// returns the balance factor of node, i.e., difference between the heights of left-subtree and right-subtree
//code here


}
```

Insertion in AVL – Time Complexity

1. New node insertion   O(log n)
2. Update height   O(1)
3. Compute balance factor O(1)
4. Perform rotations O(1)

Deletion in AVL

1. Delete node n from the tree using BST deletion algorithm
2. Update height of n
3. Compute balance factor of n and check for imbalance
   - If tree is unbalanced, perform rotations
   - There are Four possible cases
     - Left-left case → rightRotate(n)
     - Right-right case → leftRorate(n)
     - Right-left case → rightRotate(n→right) then leftRotate(n)
     - Left-Right case → leftRotate(n→left) then rightRotate(n)

Deletion in AVL

```
Node* BSTDelete(Node* n, int key){ // key = value to be removed from tree
   //Find node to be deleted
   if( n→ k > key )
        n→left = BSTDelete(n→left , key); //recurse left
   else if ( n→ k < key )
       n→right = BSTDelete(n→right , key); // recurse right
   else // key found, BSTDelete logic here
   // 1.Leaf node, 2. node with one child, 3. Node with two children

    updateHeight(n);
    int bf = getBalanceFactor(n);
    if(bf > 2 && FindBalanceFactor(n→left) > 0 )// 1. Left-left case
          RotateRight(n);
     //Similarly add checks for remaining three cases here
     // 2. Right-Right case, 3. Left-Right case, 4. Right-Left case

  return n;
}
```

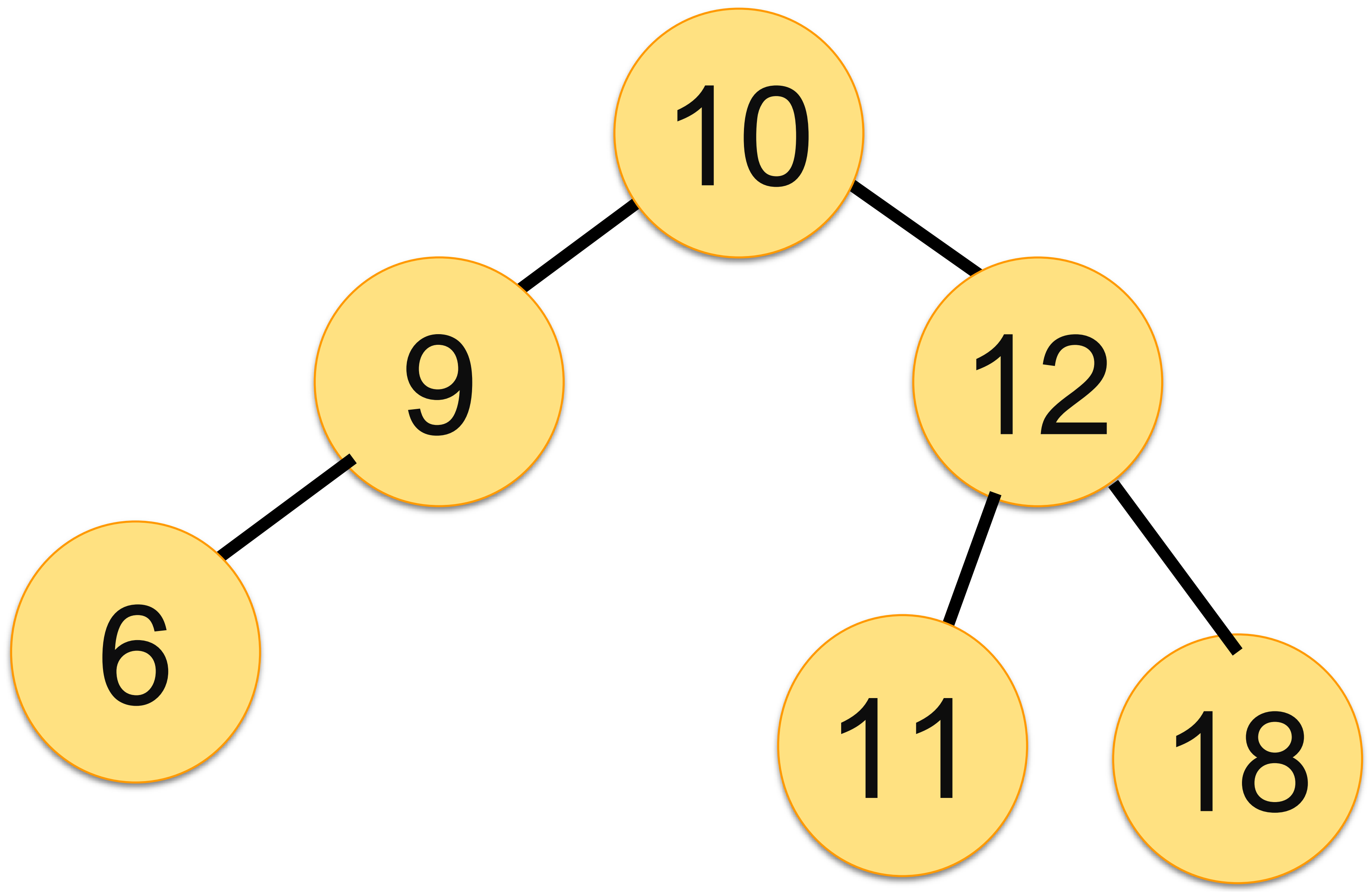
Deletion in AVL – Time Complexity

1. Delete Node O(log n)
2. Update height O(1)
3. Compute balance factor O(1)
4. Perform rotations O(1)

AVL Time Complexities

- Search O(log n)
- Insertion O(log n)
- Deletion O(log n)

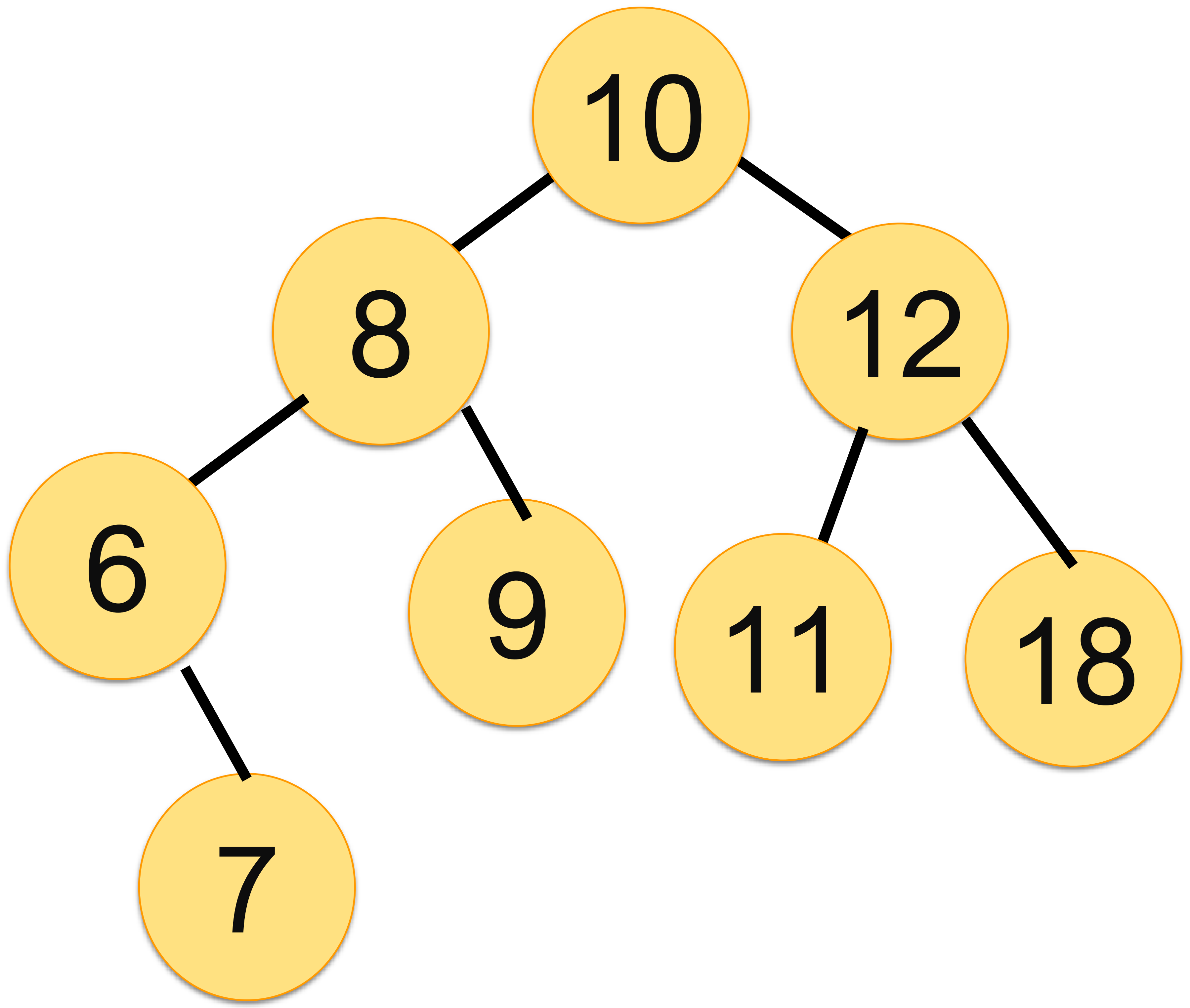
Insert Examples



Insert ‘8’

Insert ‘7’

Delete



Delete ‘ 18’

Delete ‘10’

Questions