

# COMP2013

## Data Structures and Algorithms

### Lecture 9.

## Balanced Binary Search Tree

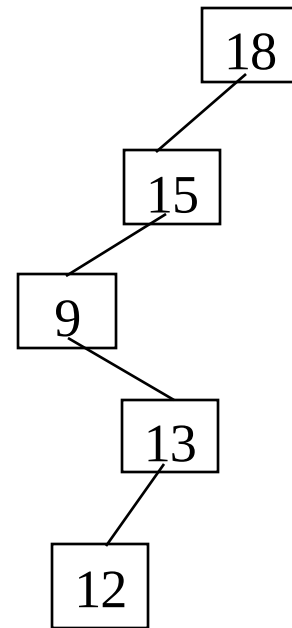
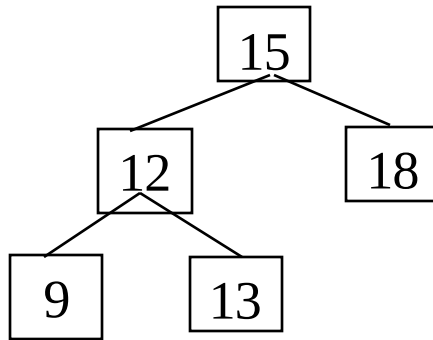
Dr. Jieming Shi

# Recap: Binary Search Tree



- **Binary search tree** is a binary tree with this property:
  - For any node  $x$  in the tree, for any node  $a$  in  $x$ 's left subtree,  $a.key \leq x.key$ , and for any node  $b$  in  $x$ 's right subtree,  $x.key \leq b.key$

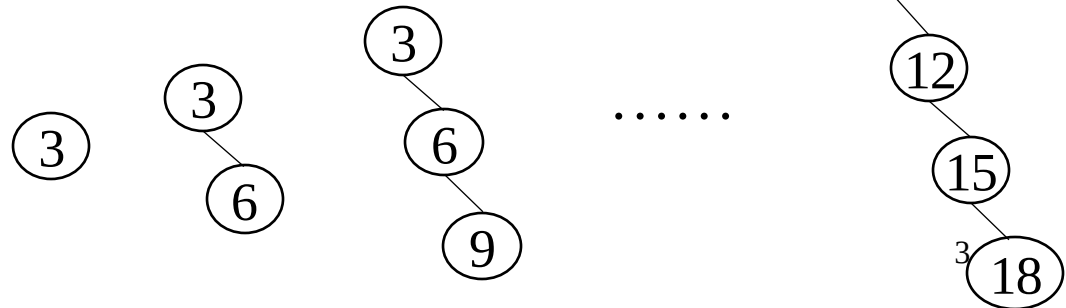
Search 13 in these BSTs,  
Which tree is faster?



# Unbalanced Tree: Example



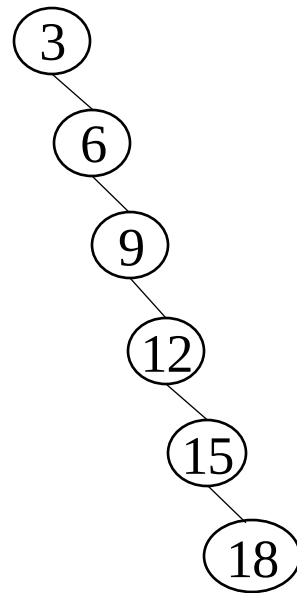
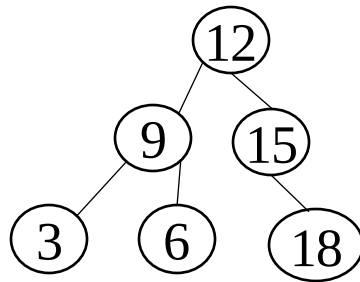
- Let's try to insert the keys 3, 6, 9, 12, 15, 18 (in this order) into a binary search tree .....
- **Problem: the tree is not “balanced”**
  - The right subtree is much taller than the left subtree
  - Tree height  $h$  can be as large as  $n-1$
  - High search time:  $O(h) = O(n)$   
where  $n$  is the number of keys (nodes in the tree)



# Balance the Tree?



- What is a ‘*balanced*’ tree?
  - E.g., for each node, its left and right subtrees have *similar* height
- How do we balance the tree while:
  - **Inserting** a key into the tree
  - **Deleting** a key from the tree
- Goal is to try to keep the height of tree in  $O(\log n)$  after insertions and deletions
- $n$  v.s.  $\log n$ :
  - 1000000
  - $\log_2 1000000 \approx 20$



# AVL Tree (Adelson-Velsky and Landis)

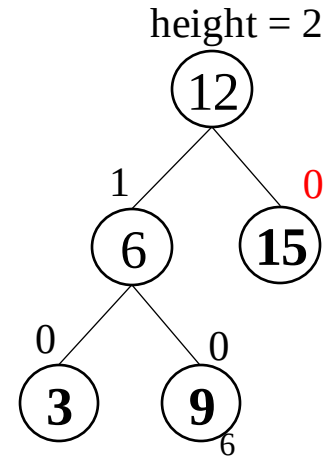
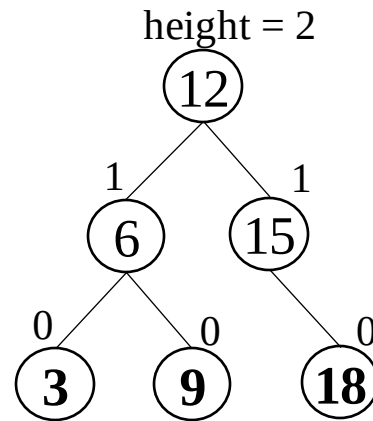


- A self-balancing BST that fixes and keeps balance after insertions and deletions.
- There are many other types of balanced BST, e.g., red-black trees in the textbook

# Terms for Binary Search Tree



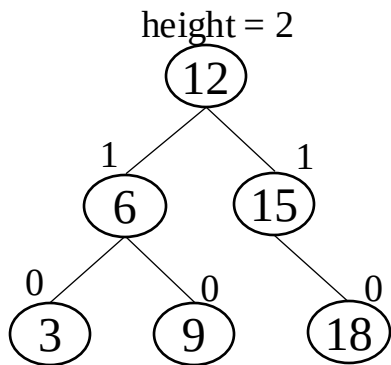
- A leaf node is a tree node without children
- The height of a node is the largest path length to any leaf node (from that node)
  - Each leaf node has the height 0
  - Let the height of a *NIL* node be  $-1$
- Example on binary search tree:
  - Leaf nodes are in bold in these trees
  - Heights of nodes are shown



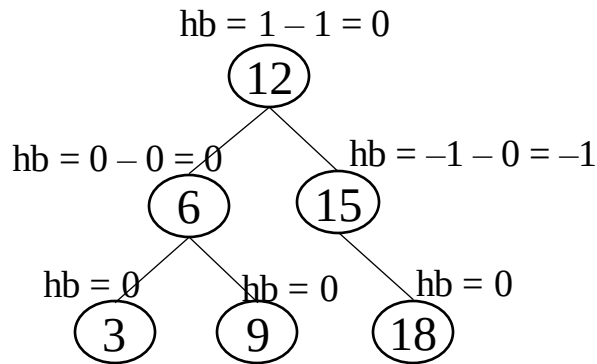
# AVL Tree



- AVL tree is a **height-balanced** *binary search tree*: for each node  $x$ , the heights of the left and right subtrees of  $x$  differ by *at most* 1.
- The height-balance ( $hb$ ) of a node  $x$  is:
  - $x$ 's left child height –  $x$ 's right child height
    - (NIL's height =  $-1$ )
- **AVL Tree Property**: each node has a height-balance either  $-1, 0, 1$   
(property is violated if some height-balance is  $<-1$  or  $>1$ )



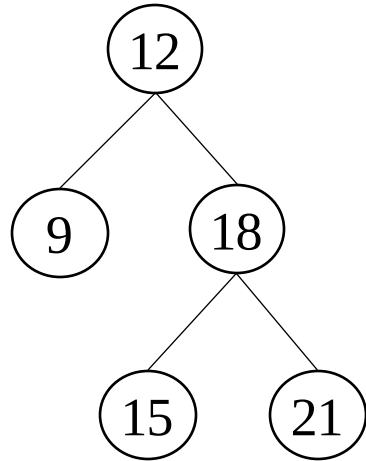
Heights of tree nodes



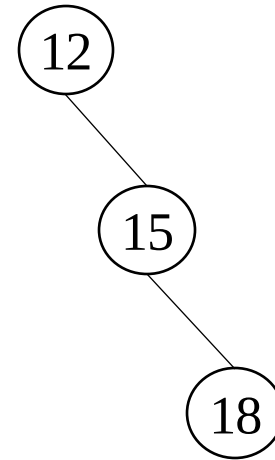
Height-balances of tree nodes



- Which tree satisfies the **AVL tree property**?
  - Hint: compute the height-balance of each node  
(first consider nodes at low levels, then nodes at high levels)



Tree A



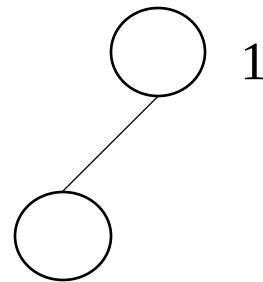
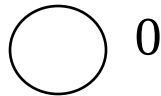
Tree B



# Height of AVL Tree



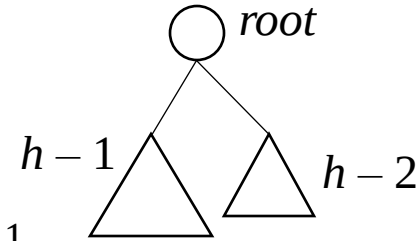
- What is the relationship between the AVL tree height  $h$  and the number of nodes  $n$ ?
- Let  $N(h)$  be the minimum number of nodes in an AVL tree with height  $h$
- We are going to find the relationship of  $n$  and  $h$  in an AVL tree.....
  - **We wish** to have something like  $h=O(\log n)$
- Base cases ( $h=0, h=1$ ):
  - $N(0) = 1$
  - $N(1) = 2$



# Height of AVL Tree



- For  $h > 1$ :
  - Let  $N(h)$  be the minimum number of nodes in an AVL tree with height  $h$
  - Consider the *left and right subtrees* of the root node
    - The subtrees are also AVL trees
  - Since the tree height is  $h$ , one of left or right subtrees must have the height  $h - 1$
  - By the AVL tree property, the other subtree cannot have a height smaller than  $h - 2$ .
    - The other subtree must have height  $h-2$ , considering  $N(h)$  being the minimum number of nodes
  - $N(h - 1)$  is the minimum number of nodes in an AVL tree of height  $h-1$
  - $N(h - 2)$  is the minimum number of nodes in an AVL tree of height  $h-2$
  - Together, we obtain the minimum number  $N(h) = N(h - 1) + N(h - 2) + 1$
  - $N(h-1) \geq N(h - 2)$  holds (Get this by *contradiction*).
  - $N(h) \geq 2N(h - 2) + 1$
  - $N(h) > 2N(h - 2) > 2 \cdot 2N(h - 4) > 2 \cdot 2 \cdot 2N(h - 6) > \dots > 2^{h/2}$
  - Take log with base 2:  $\log N(h) > h/2 \cdot \log 2 = h/2$
  - $h < 2 \log N(h)$
  - $h < 2 \log n$ , where  $n$  is the number of nodes in an AVL tree, and  $N(h) < n$
  - Worst-case AVL trees have height  $h = O(\log n)$ .



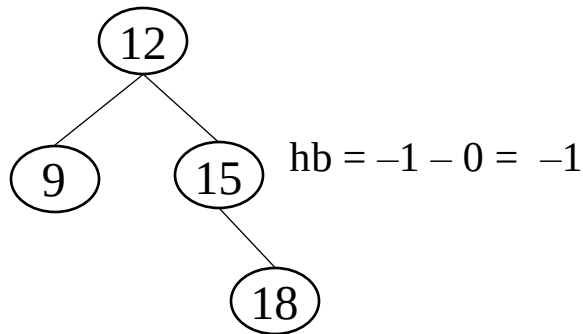
**Solve this recurrence by  
substitution, recursion tree, or  
solve it directly.**

- → AVL tree supports search and update operations in  $O(\log n)$  time

# Insertion in AVL tree

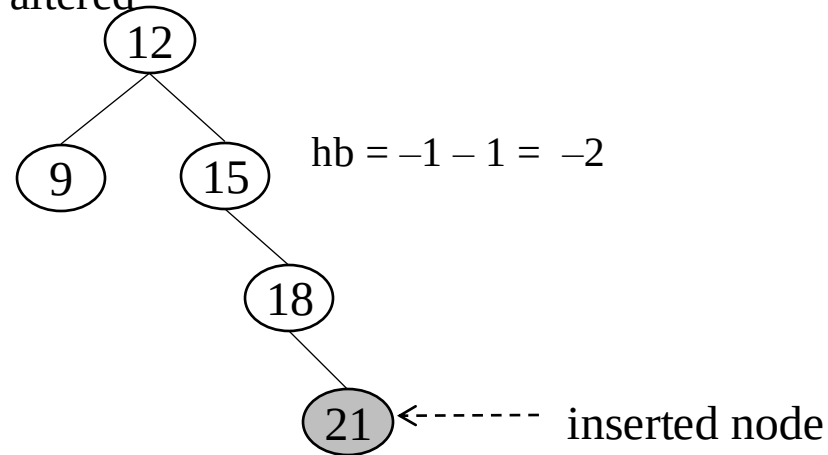


- When we insert a key (21) into an AVL tree, some node may have height-balance  $-2$  or  $2$  (BST insertion)
  - This violates the AVL tree property
- After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered
  - Because only those nodes have their subtrees altered
- How to fix the height imbalance?



e.g., insert 21

Tree before insertion (AVL yes)

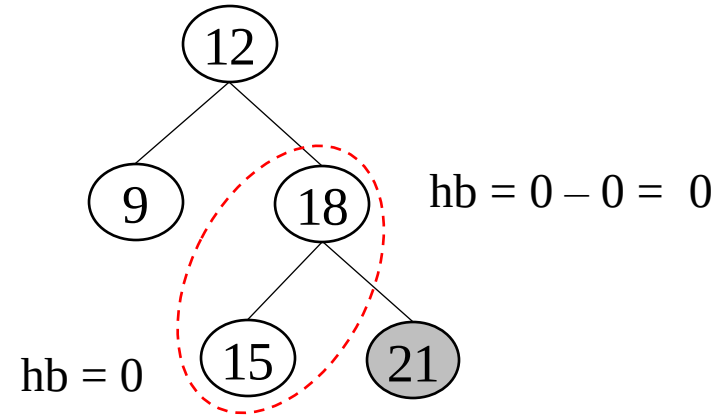
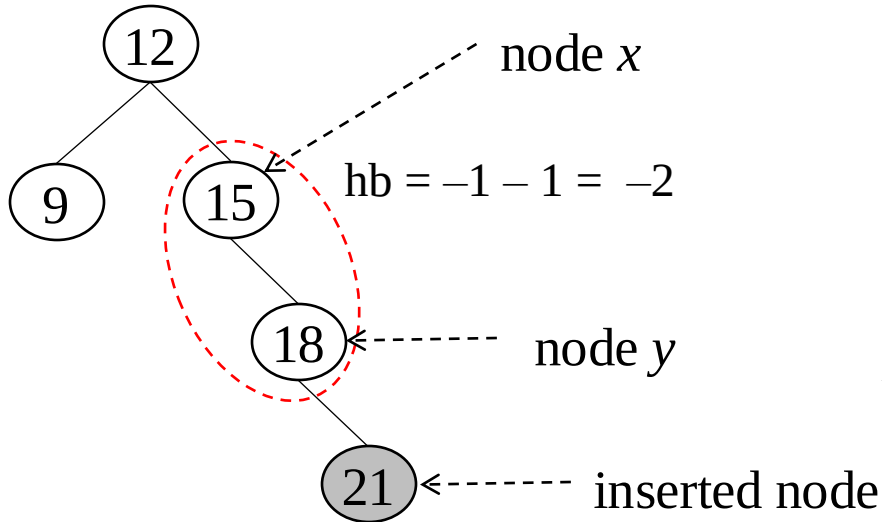


Tree after insertion (AVL no)

# Left Rotation



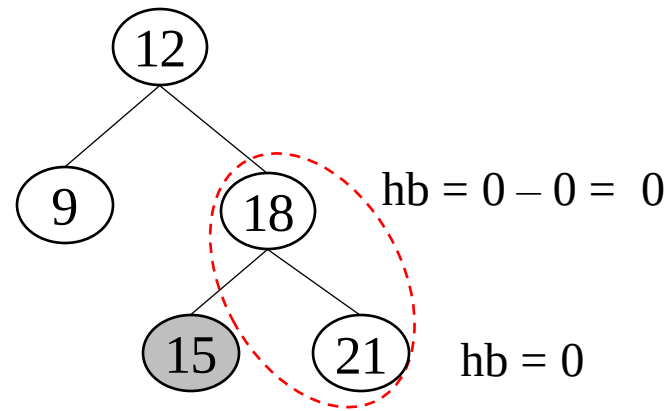
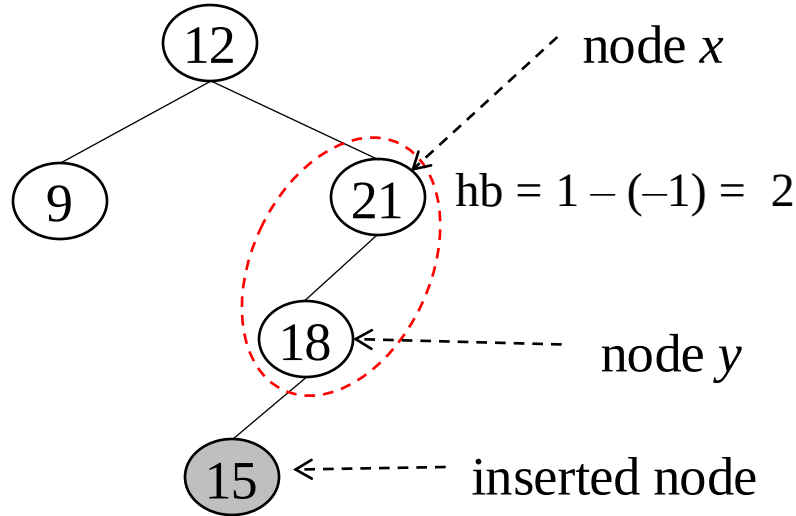
- The imbalance at 15
- Left rotation: rotate the nodes left to balance height; a swap between parent and right child
- BST tree property is maintained



# Right Rotation



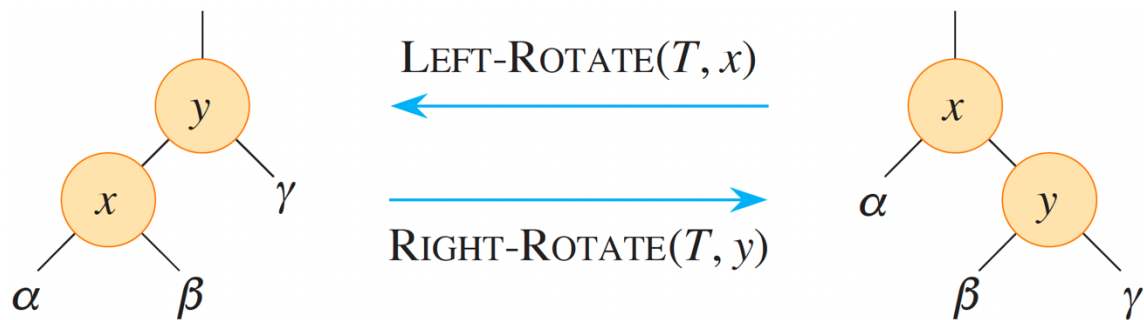
- The imbalance at 21 is caused by its higher left subtree over right subtree
- Right rotation: rotate the nodes right to balance height; a swap between parent and left child
- BST tree property is preserved



# Rotations



- The rotations finish by changing a constant number of pointers
- The rotations preserve the binary search tree property
- $\alpha \leq x \leq \beta \leq y \leq \gamma$



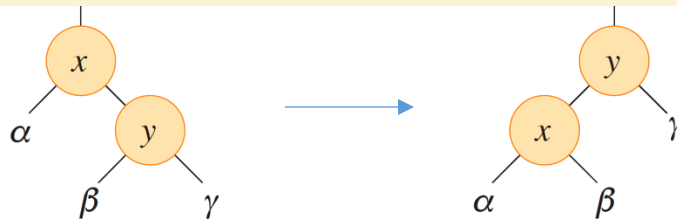
$\alpha, \beta, \gamma$  represent arbitrary subtrees

# Left Rotation



LEFT-ROTATE( $T, x$ )

```
1   $y = x.right$ 
2   $x.right = y.left$       // turn  $y$ 's left subtree into  $x$ 's right subtree
3  if  $y.left \neq T.nil$   // if  $y$ 's left subtree is not empty ...
4       $y.left.p = x$       // ... then  $x$  becomes the parent of the subtree's root
5   $y.p = x.p$             //  $x$ 's parent becomes  $y$ 's parent
6  if  $x.p == T.nil$       // if  $x$  was the root ...
7       $T.root = y$         // ... then  $y$  becomes the root
8  elseif  $x == x.p.left$  // otherwise, if  $x$  was a left child ...
9       $x.p.left = y$       // ... then  $y$  becomes a left child
10 else  $x.p.right = y$    // otherwise,  $x$  was a right child, and now  $y$  is
11  $y.left = x$            // make  $x$  become  $y$ 's left child
12  $x.p = y$ 
```





- You can write the pseudo code of Right-Rotate



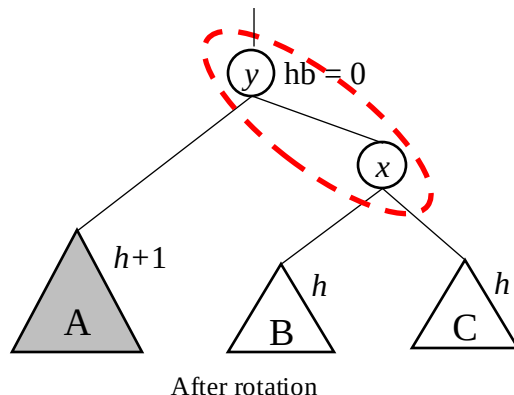
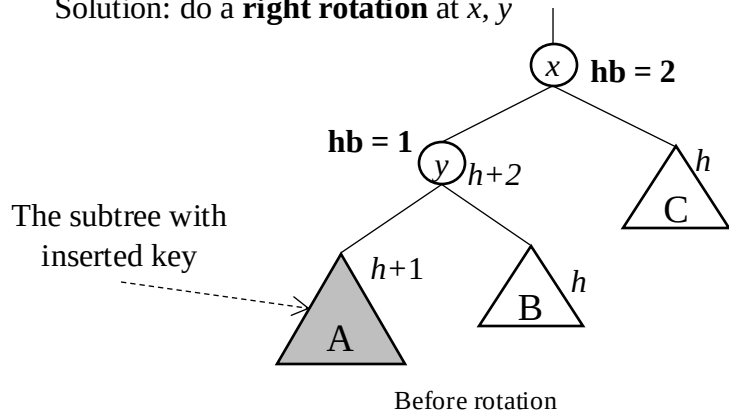


- After inserting a key  $k$ , let  $x$  be the *lowest node* violating AVL tree property
  - Case i (LL):  $k$  inserted into  $x$ 's left child's left subtree
  - Case ii (RR):  $k$  inserted into  $x$ 's right child's right subtree
  - Case iii (LR):  $k$  inserted into  $x$ 's left child's right subtree
  - Case iv (RL):  $k$  inserted into  $x$ 's right child's left subtree

# Case i: $k$ inserted into $x$ 's left child's left subtree



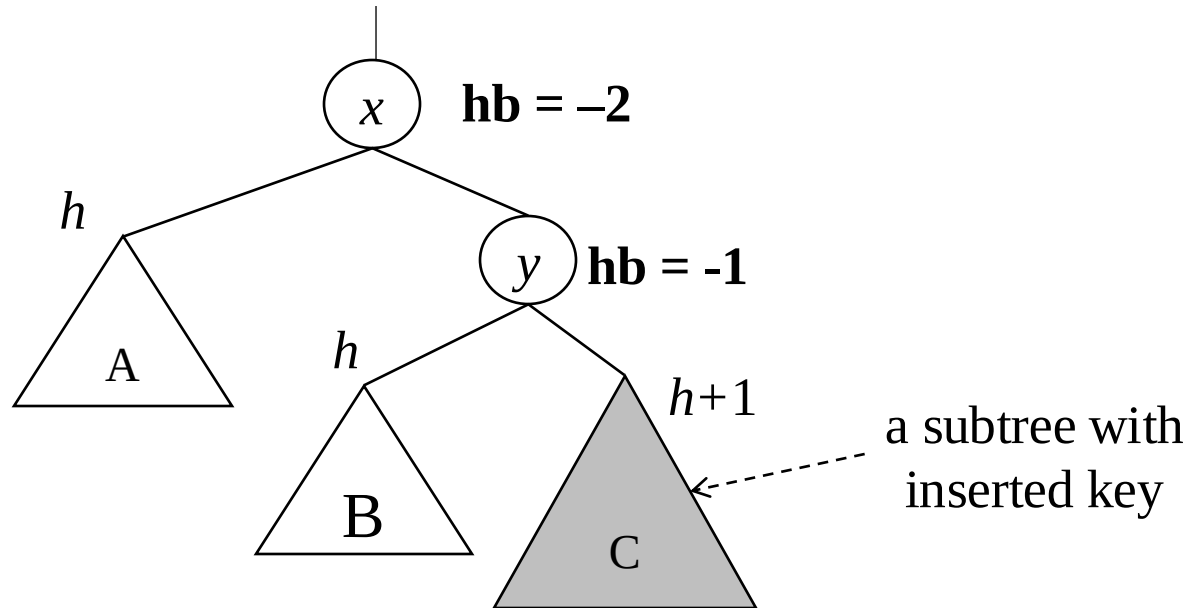
- After insertion, the AVL property is violated at node  $x$  ( $x$  is the *lowest* node breaking the property)
- Before insertion,  $x$  does not violate AVL property (before insertion,  $hb$  of  $x$  is one of 1,0,-1)
- The insertion increases the height of left subtree of  $x$  by 1 (leading to the violation), while the height of right subtree of  $x$  does not change
- Therefore, node  $x$  must have  $hb=2$  after insertion (before insertion,  $hb$  of  $x$  must be 1)
- Let  $h$  be the right subtree of  $x$
- After the insertion, the left subtree of  $x$  has height  $h+2$ , i.e.,  $y$  of the left subtree has height  $h+2$
- After the insertion, at least one of the subtrees of  $y$  must have height  $h+1$  (since  $y$  has height  $h+2$ )
- Can the right subtree of  $y$  have height  $h+1$ ?
  - NO. (If so, the height of  $y$  is already  $h+2$  before insertion, which already makes the tree imbalanced)
- After insertion, it must be the *left subtree* of  $y$  has height  $h+1$ .
- Can the height of the right subtree of  $y$  be  $h-1$ ?
  - NO. (If so, the lowest node breaking AVL property is  $y$ , not  $x$ )
- The right subtree of  $y$  must have height  $h$
- Solution: do a **right rotation** at  $x, y$



## Case ii: $k$ inserted into $x$ 's right child's right subtree



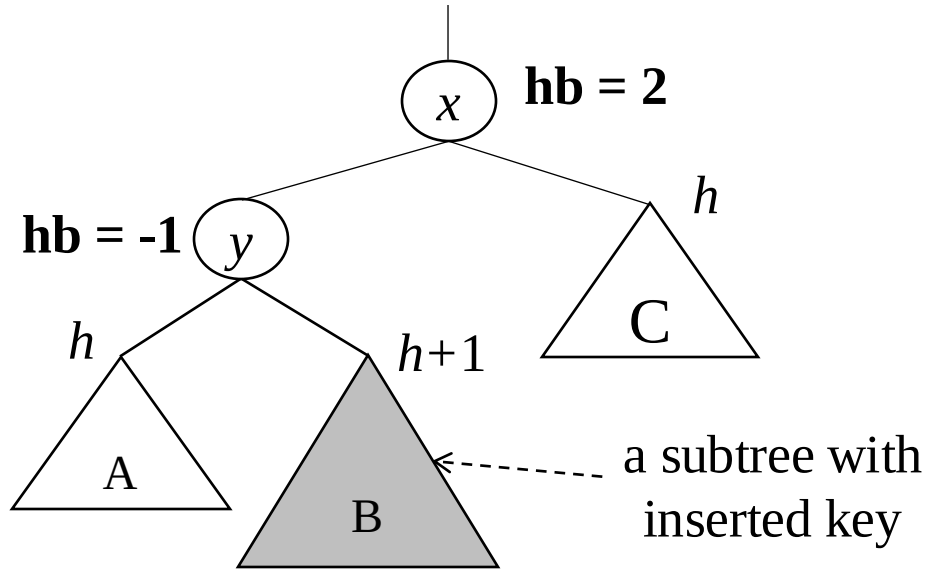
- The AVL property is violated at lowest node  $x$
- [Exercise] try to solve this problem



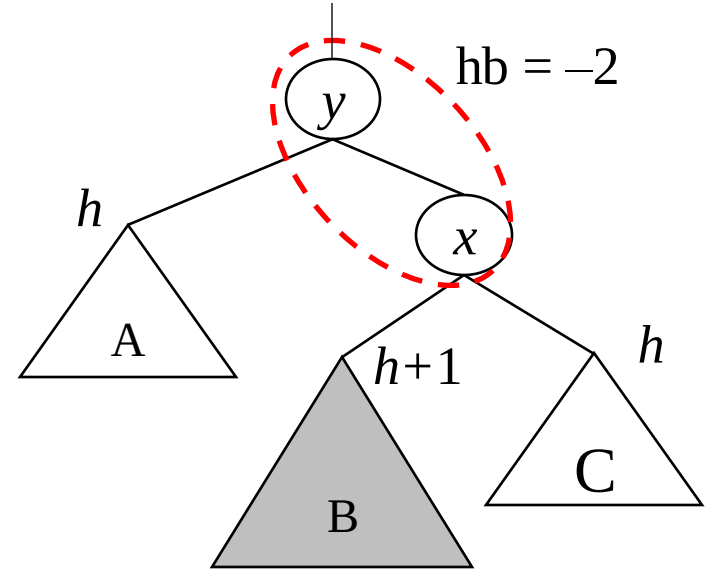
# Case iii: $k$ inserted into $x$ 's left child's right subtree



- The AVL property is violated at lowest node  $x$
- Can we solve this by a right rotation?
  - **NO!** Node  $y$  will violate the property



Before rotation

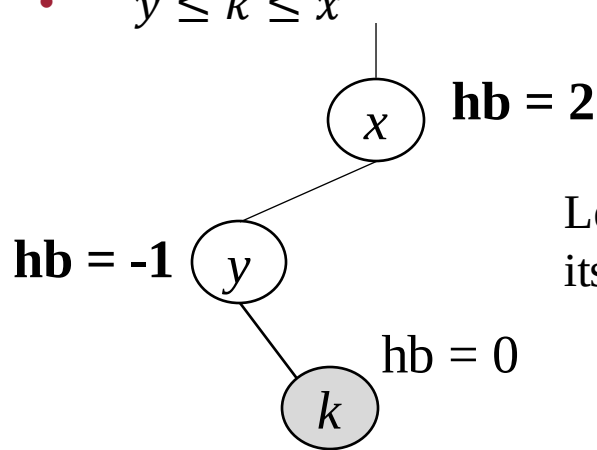


After rotation

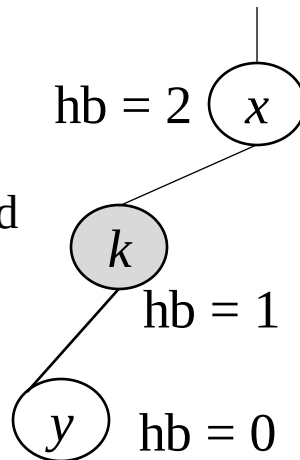
# Case iii: $k$ inserted into $x$ 's left child's right subtree



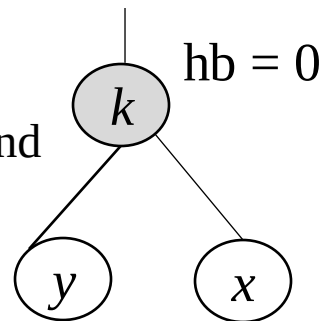
- **Double Rotation**
- Special subcase: when the right subtree of  $x$  is empty, and the subtrees of  $y$  are empty before insertion
- $y \leq k \leq x$



Left rotate  $y$  and  
its right child  $k$



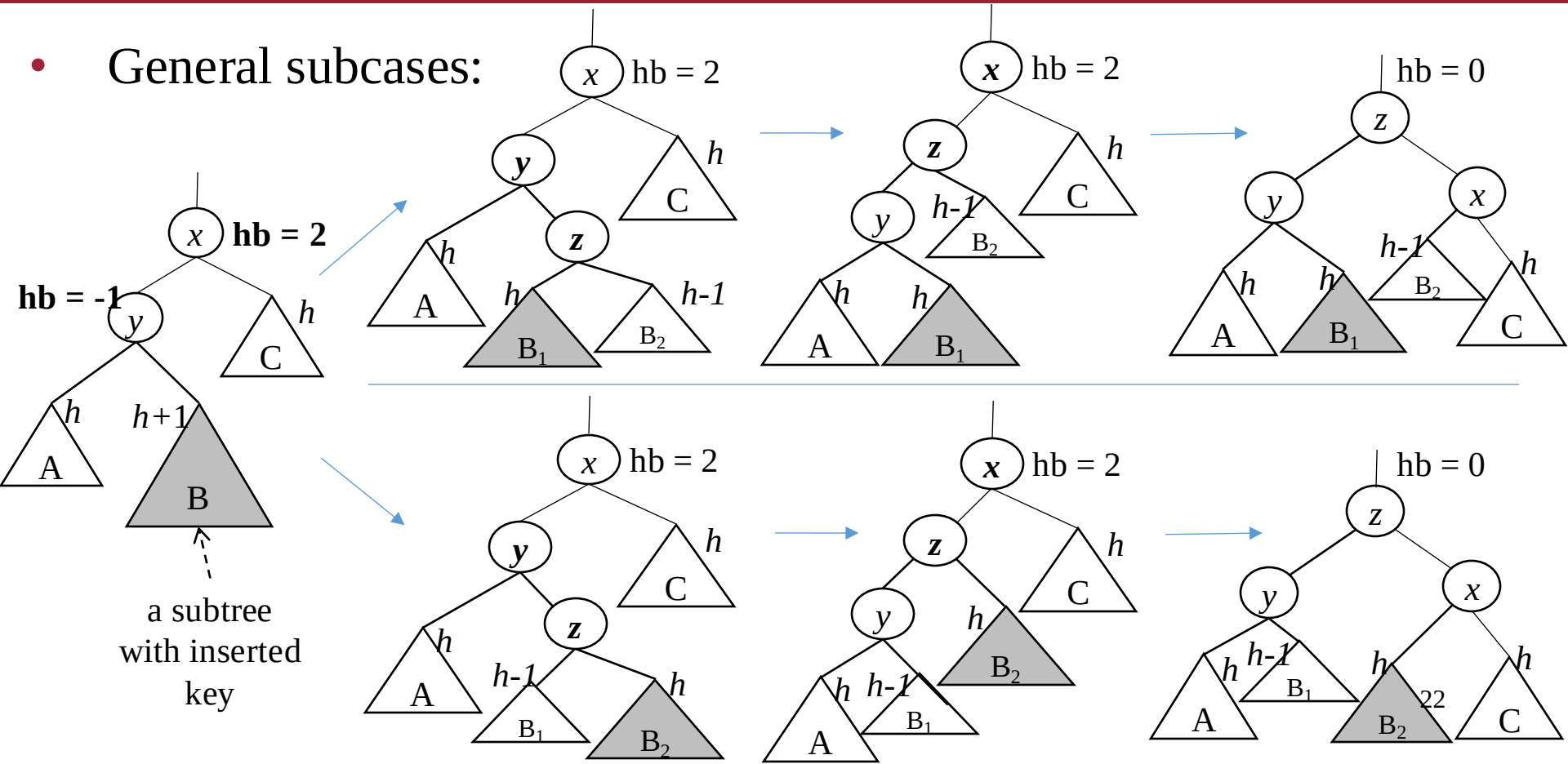
Right rotate  $x$  and  
its left child  $k$



# Case iii: $k$ inserted into $x$ 's left child's right subtree



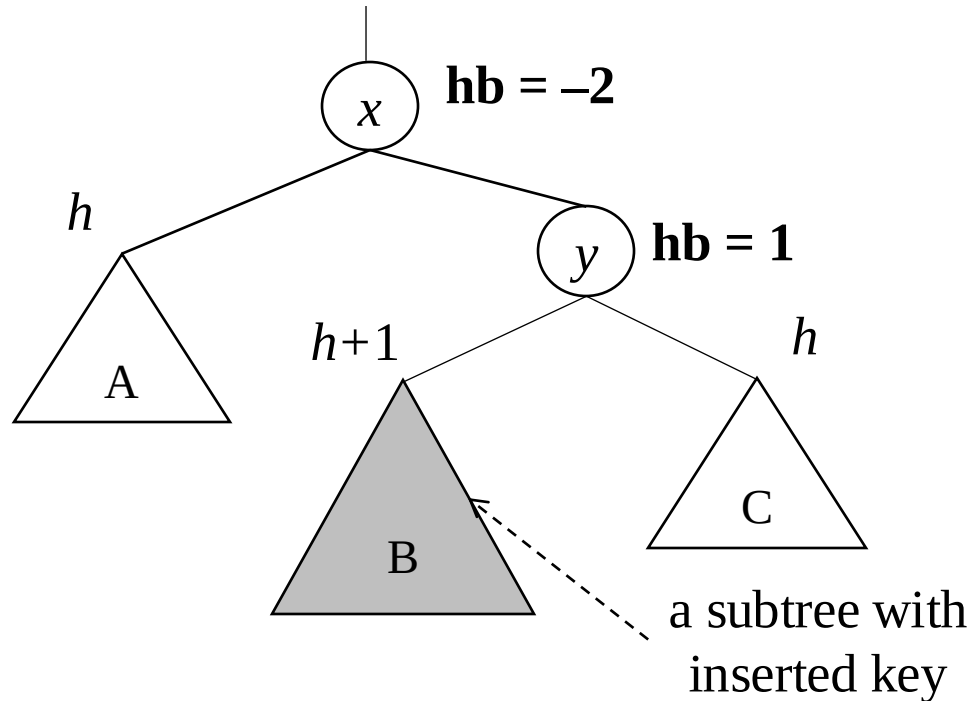
- General subcases:



# Case iv: $k$ inserted into $x$ 's right child's left subtree



- The AVL property is violated at node  $x$
- [Exercise] try to solve this problem





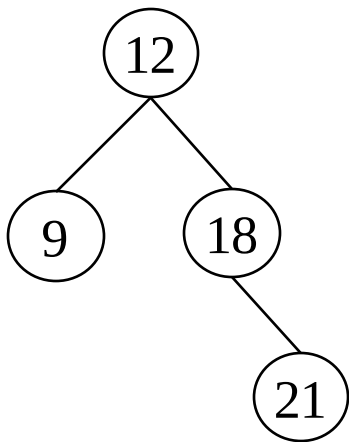
- After inserting key  $k$ , let  $x$  be the *lowest node* violating AVL tree property
  - Case i:  $k$  inserted into  $x$ 's left child  $y$ 's left subtree ( $hb$  of  $x$  is **2**,  $hb$  of  $y$  is **1**)
    - Right rotation
  - Case ii:  $k$  inserted into  $x$ 's right child  $y$ 's right subtree ( $hb$  of  $x$  is **-2**,  $hb$  of  $y$  is **-1**)
    - Left rotation
  - Case iii:  $k$  inserted into  $x$ 's left child  $y$ 's right subtree ( $hb$  of  $x$  is **2**,  $hb$  of  $y$  is **-1**)
    - Double rotation: Left rotation and then right rotation
  - Case iv:  $k$  inserted into  $x$ 's right child  $y$ 's left subtree ( $hb$  of  $x$  is **-2**,  $hb$  of  $y$  is **1**)
    - Double rotation: right rotation and then left rotation
- For each insertion, at most two rotations are needed to restore the height balance of the entire tree.



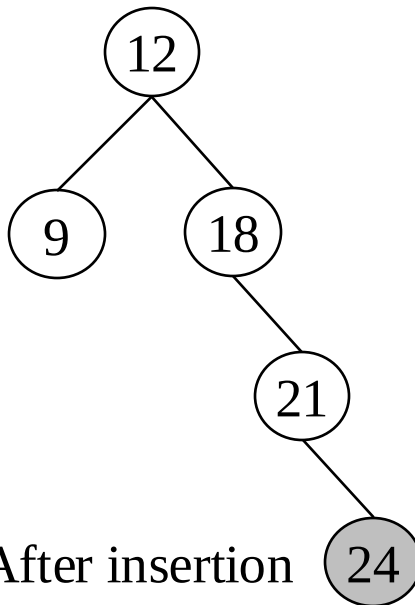
# Insertion: Example 1



- Insert '24' into the tree
  - Node '18' has a height-balance  $-2$
  - Do a left rotation at '18', '21'

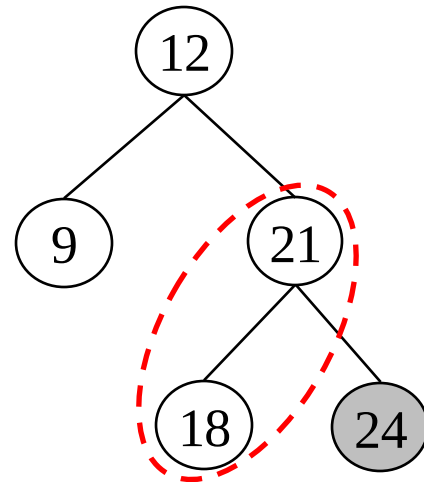


Before insertion



After insertion

Which rotation case?

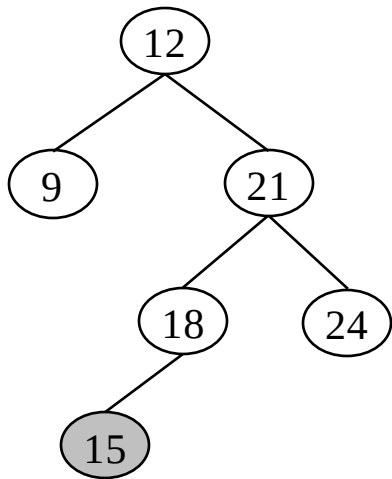


After left rotation<sup>25</sup>

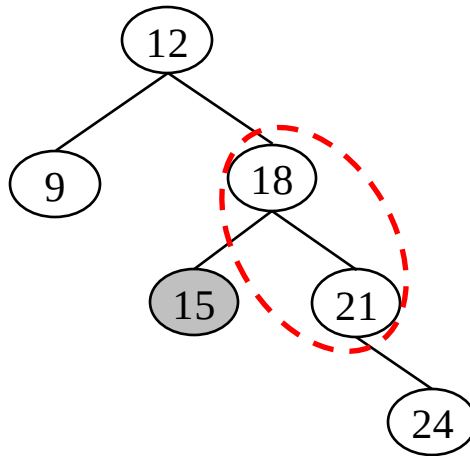
# Insertion: Example 2



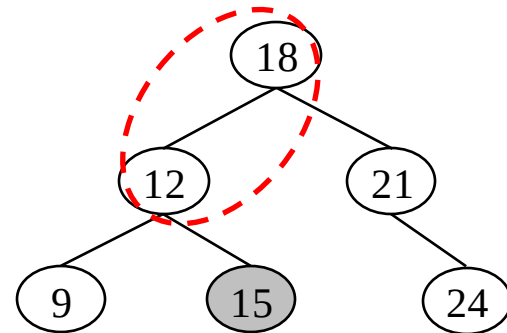
- Insert '15' into the tree
  - Node '12' has a height-balance  $-2$
  - Do a double rotation:  
right rotate '21', '18'; left rotate '12', '18'



After insertion



After right rotation



After left rotation

# AVL Tree Deletion Algorithm

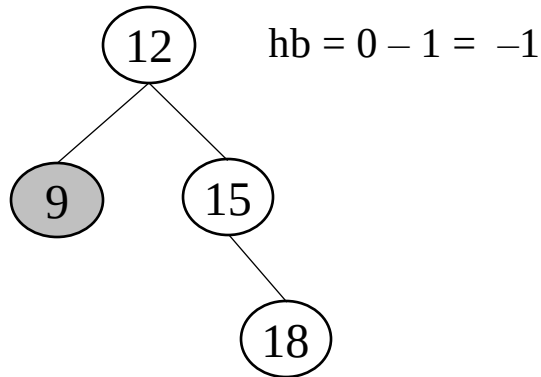


- We consider three cases when deleting node  $x$ :
  - If  $x$  has no child, remove it
  - If  $x$  has one child, replace it by the child
  - If  $x$  has two child, replace it by the successor of  $x$   
(see lecture #8 for details)
- After deleting a node  $x$ :
  - Trace the path from the lowest node with child changed to root
    - Need to find this node
  - For **each node along path**, restore the height balance if needed by doing single or double rotations
    - **$O(h)$  rotations**, while insertion only needs 1-2 rotation

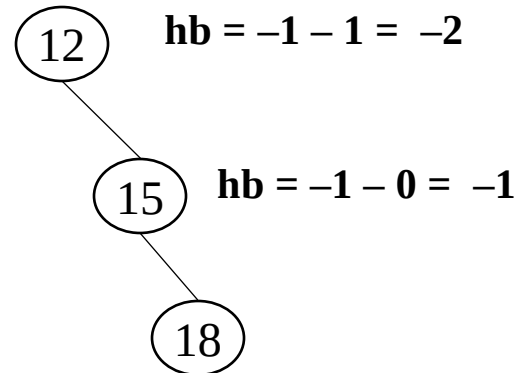
# Deletion in AVL Tree



- When we delete a key (9) from an AVL tree, some node may have height-balance  $-2$  or  $2$ 
  - This violates the AVL tree property
- How do we solve this problem?
  - Solution: do rotation (as we learnt before)



Tree before deletion  
(AVL yes)



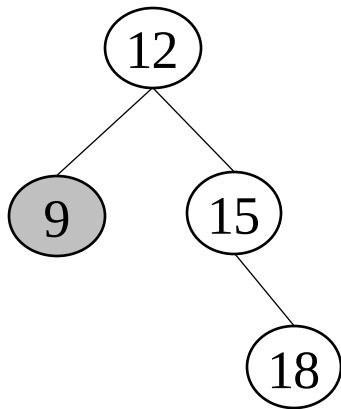
Tree after deletion  
(AVL no)

# Deletion: Example 1

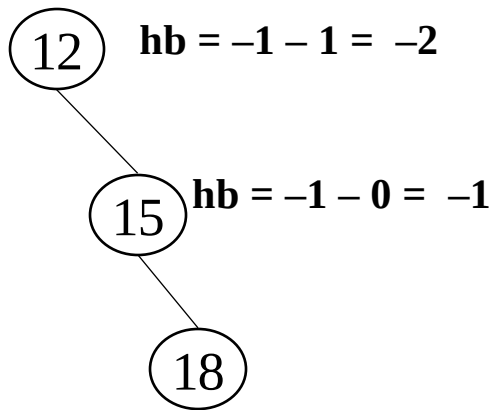


- Delete '9' from the tree
  - '9' has no children; just delete it
  - Node '12' has a height-balance  $-2$
  - Do a left rotation at '12', '15'

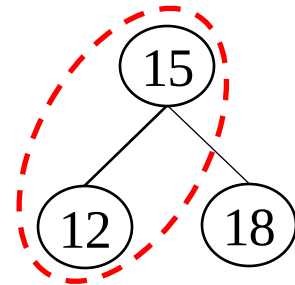
Which rotation case?



Before deletion



After deletion



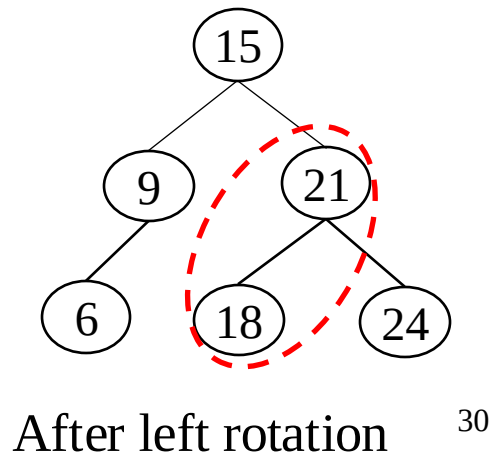
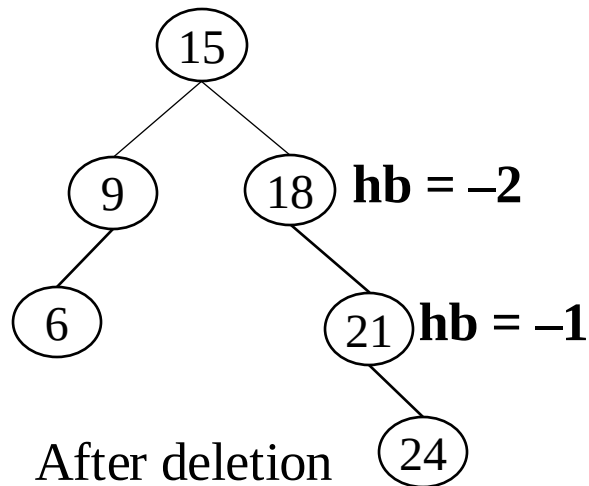
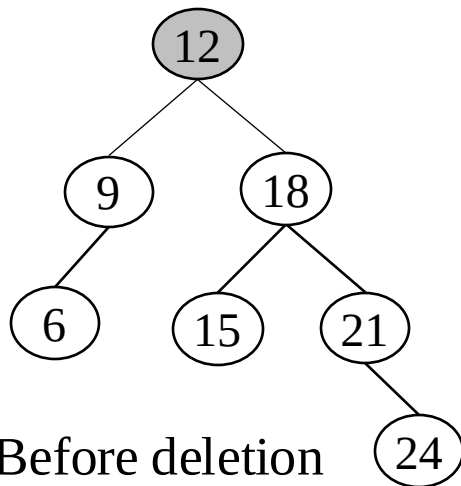
After left rotation

# Deletion: Example 2



- Delete '12' from the tree
  - Replace '12' by its successor '15', then delete old '15'
  - Node '18' is the lowest node with child changed
  - Node '18' has a height-balance  $-2$
  - Do a left rotation at '18', '21'
- After deletion, starting from **18** check the hb of ancestors

Which rotation case?

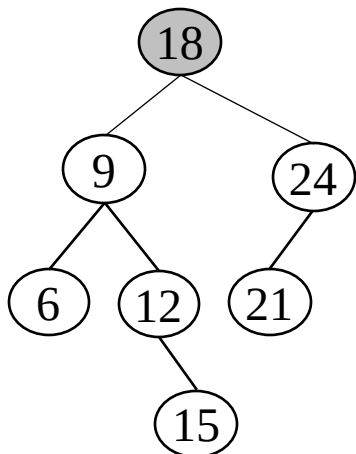


# Deletion: Example 3

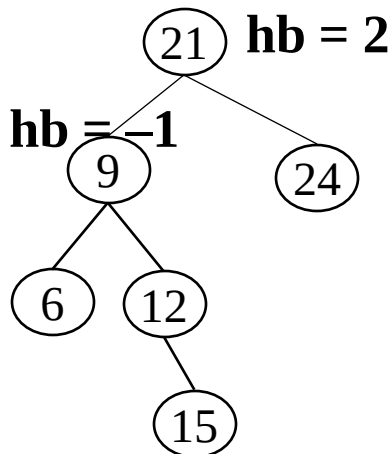


- Delete '18' from the tree
  - Replace '18' by its successor '21', then delete old '18'
  - Node '24' is the lowest node with child changed, starting from '24', we check it and its ancestors
  - Node '21' has a height-balance 2
  - Do a double rotation :
    - left rotate '9', '12';
    - right rotate '21', '12'

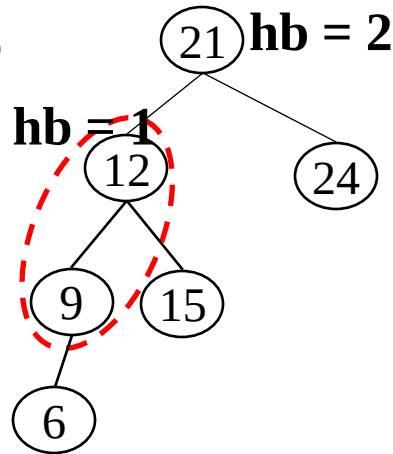
Which rotation case?



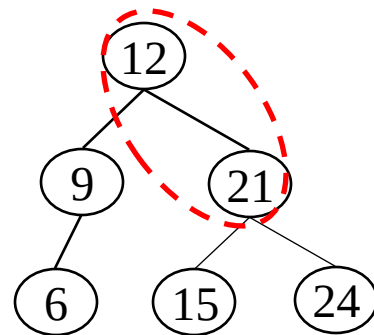
Before deletion



After deletion



After left rotation

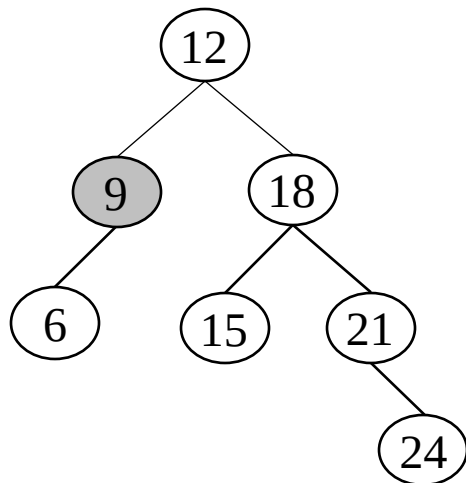


After right rotation

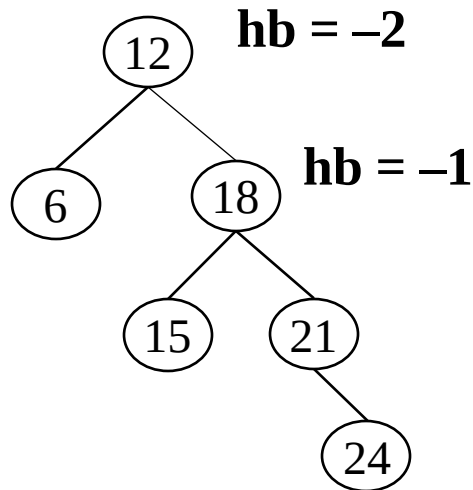
# Deletion: Example 4



- Delete '9' from the tree
  - Node '12' has a height-balance  $-2$
  - Do a left rotation at '12', '18'

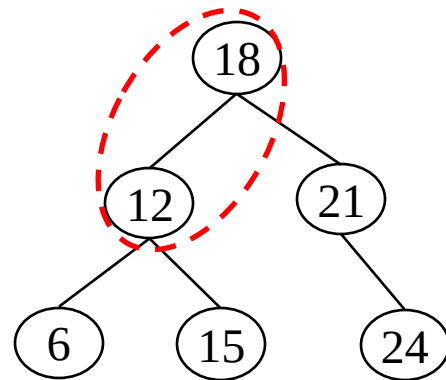


Before deletion



After deletion

Which rotation case?



After left rotation

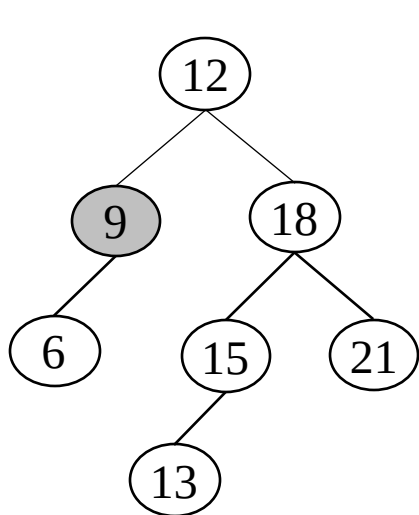


# Deletion: Example 5

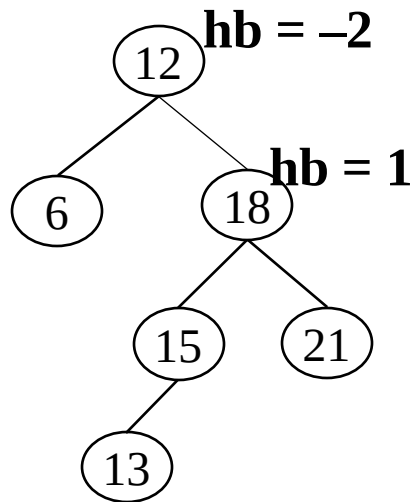


- Delete '9' from the tree
  - Node '12' has a height-balance  $-2$
  - Do a double rotation:  
right rotate '18', '15'; left rotate '12', '15'

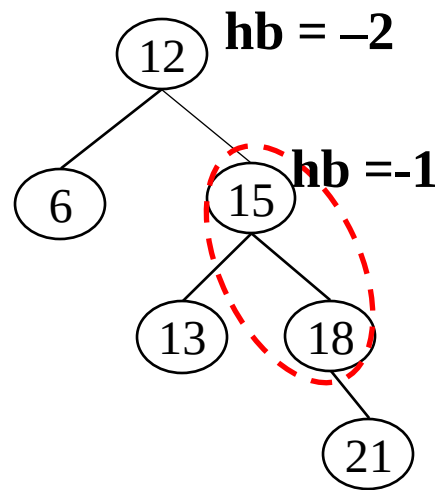
Which rotation case?



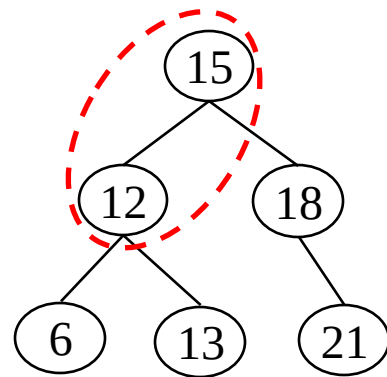
Before deletion



After deletion



After right rotation



After left rotation



- AVL tree property
- Single, double rotations in AVL tree
- Insertion, deletion in AVL tree