



# Data Structures and Algorithms Design

**BITS Pilani**  
Hyderabad Campus

Febin.A.Vahab  
2019-20

# SESSION 9 -PLAN



Online Session(#)	List of Topic Title	Text/Ref Book/external resource
9	Binary Search Tree - Motivation with the task of Searching and Binary Search Algorithm, Properties of BST, Searching an element in BST  AVL Trees	T1: 3.1  T1:3.2

- From previous lectures:
  - Binary search trees store linearly ordered data
  - Best case height:  $O(\log(n))$
  - Worst case height:  $O(n)$
- Requirement:
  - Define and maintain a balance to ensure  $O(\log(n))$  operations

# AVL trees



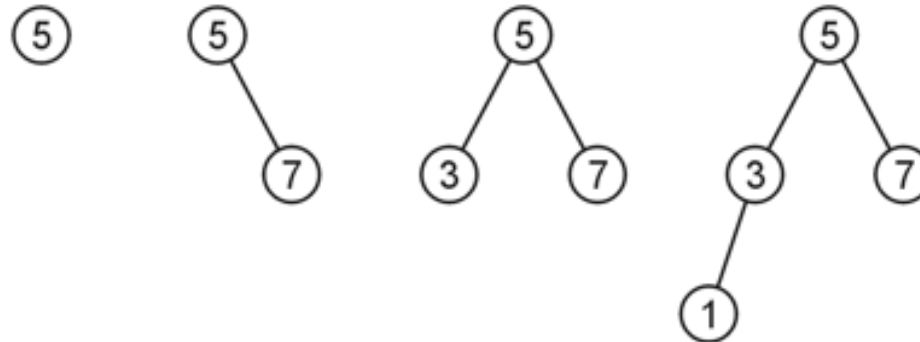
- The AVL tree is the first balanced binary search tree ever invented.
- It is named after its two inventors, [G.M. Adelson-Velskii](#) and [E.M. Landis](#), who published it in their 1962 paper "An algorithm for the organization of information."

- AVL trees are balanced
- An AVL Tree is a binary search tree such that for every internal node  $v$  of  $T$ , the heights of the children of  $v$  can differ by at most 1
- This difference is called the **Balance Factor**.
- For an AVL tree  $|\text{balance factor}| \leq 1$  for all the nodes.

# AVL trees



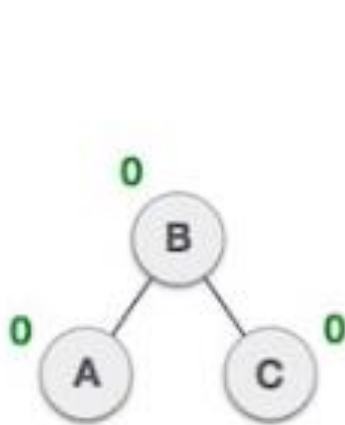
- **BalanceFactor=height(left-subtree)– height(right-subtree)**



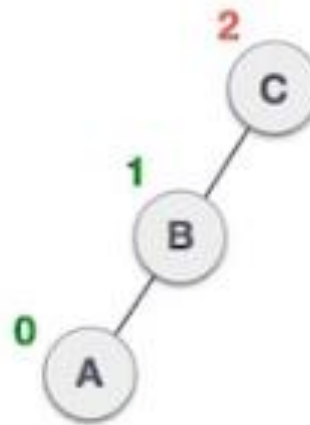
**AVL trees with 1 ,2,3,and 4 nodes**

# AVL trees

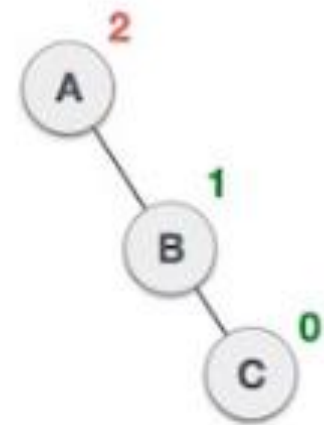
- **BalanceFactor=height(left-subtree)– height(right-subtree)**



Balanced



Not balanced



Not balanced

Image credit: Tutorials point

# AVL Trees-Example

---





# AVL trees-Rotations

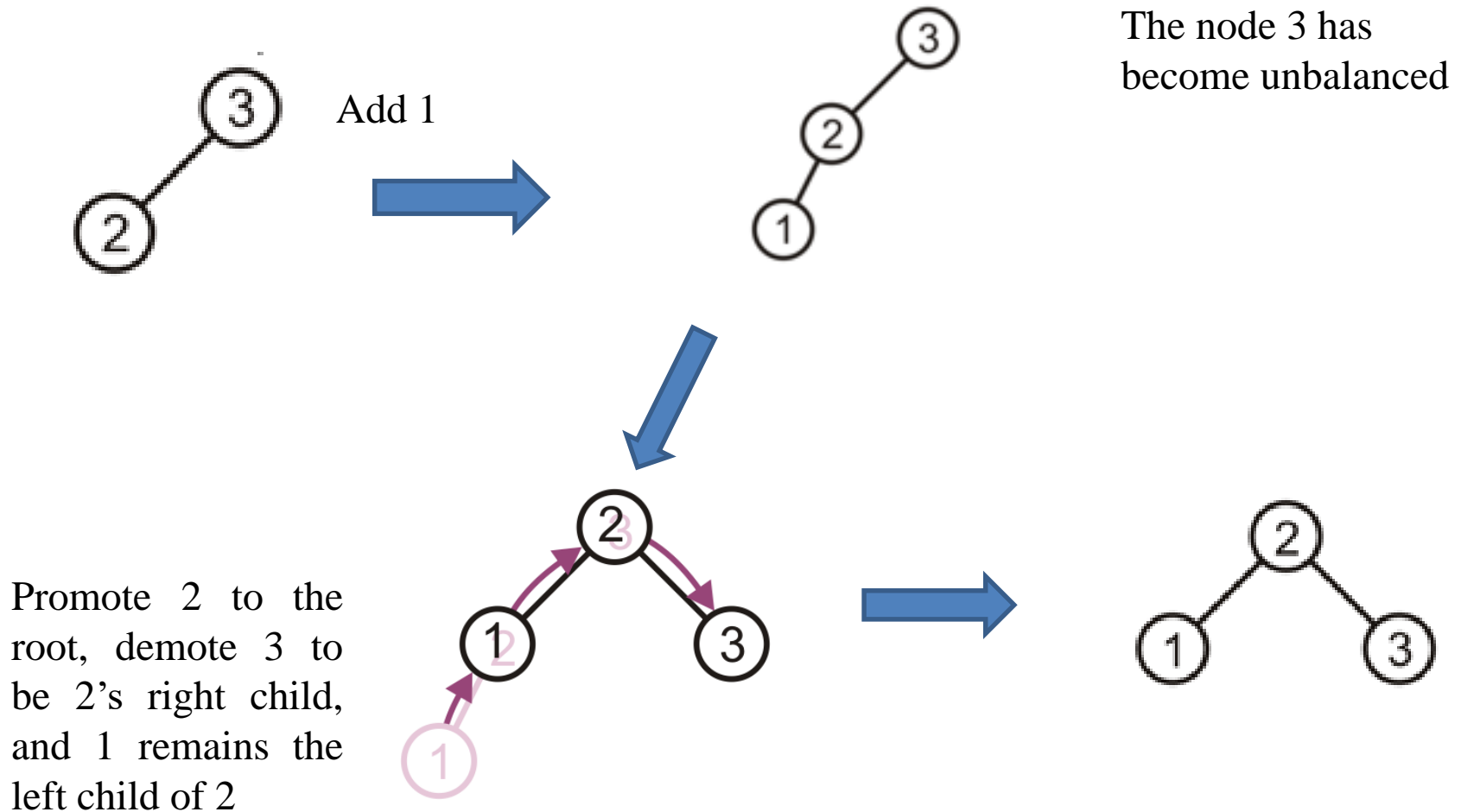


- To balance itself, an AVL tree may perform the following four kinds of rotations –
  - Left rotation
  - Right rotation
  - Left-Right rotation
  - Right-Left rotation
- To have an unbalanced tree, we at least need a tree of height 2.

# AVL trees-Rotations



- **Right Rotation** Node is inserted in the left of left-subtree



# AVL trees-Rotations

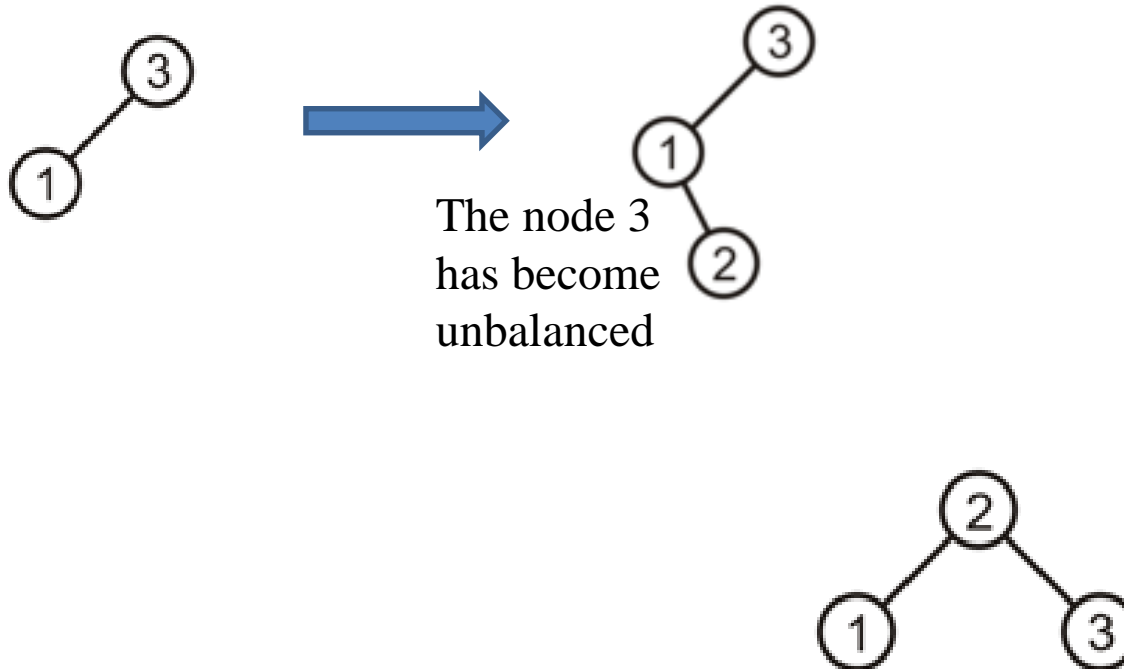


- **Left Rotation:** Node is inserted into right of right subtree.  
After inserting new node, tree becomes unbalanced

# AVL trees-Rotations



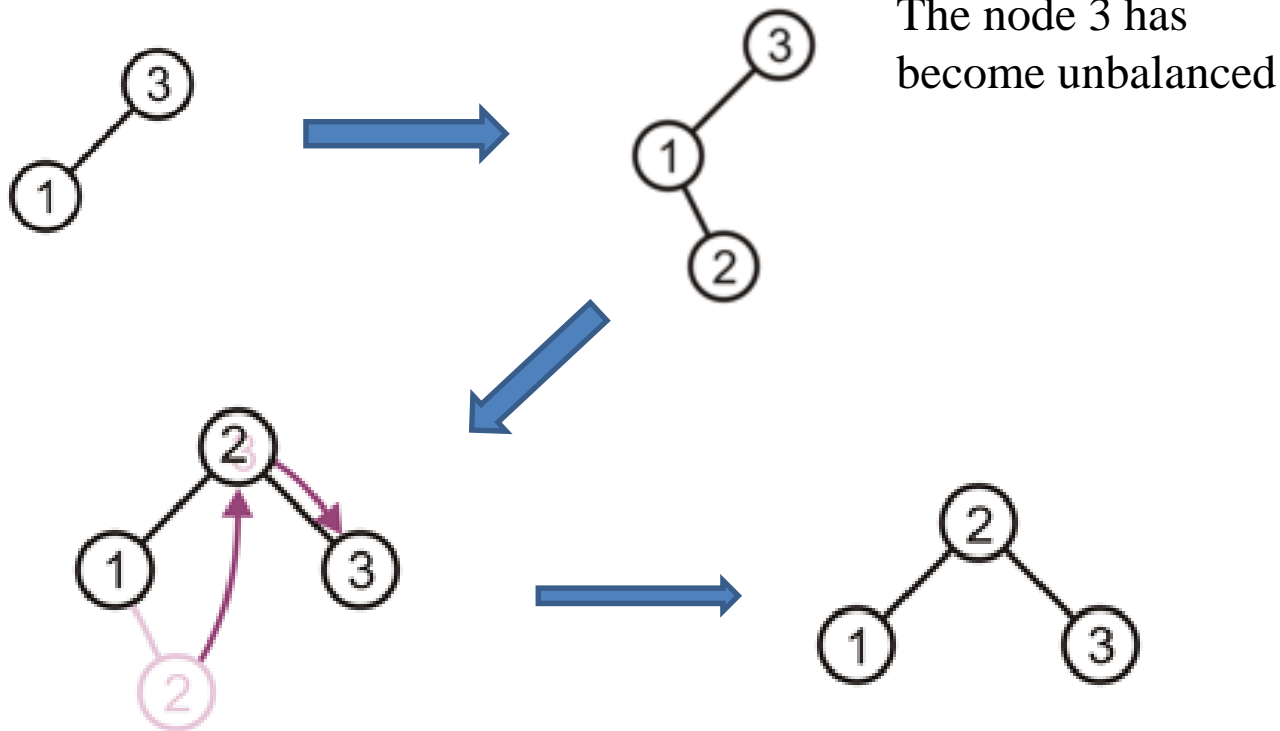
- **Left Right Rotation (Double Rotation):** Node is inserted in the right of left-subtree and makes the tree unbalanced.



# AVL trees-Rotations



- **Left Right Rotation**

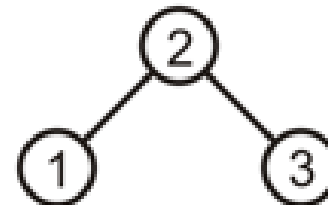


Promote 2 to the root, and assign  
1 and 3 to be its children

# AVL trees-Rotations



- **Right Left Rotation:** Node is inserted in the left of right subtree and make the tree unbalanced



# AVL Trees-General Case

## LL Imbalance

---



# AVL Trees-General Case

## LL Imbalance

---





# AVL Tree-General Case

## LR imbalance

---



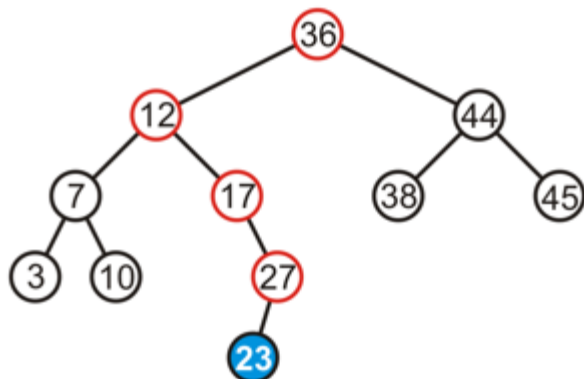
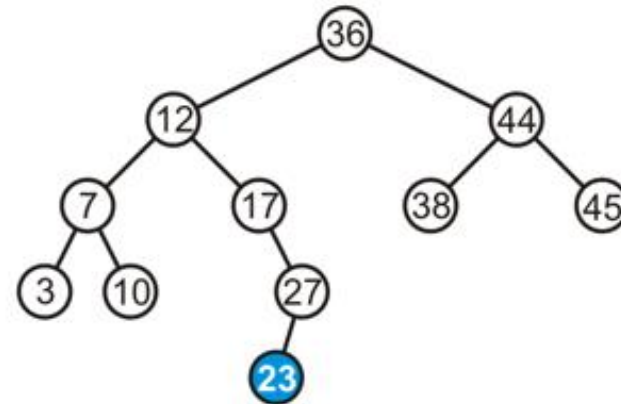
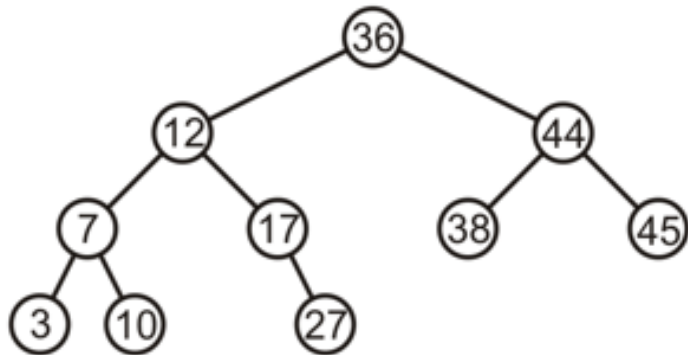
# AVL Tree-General Case

## LR imbalance

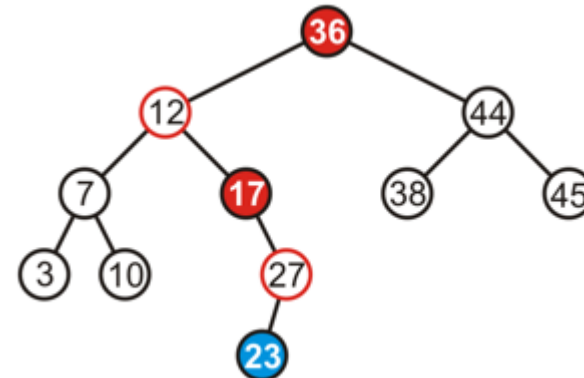
---



# AVL Insertion-Case 1

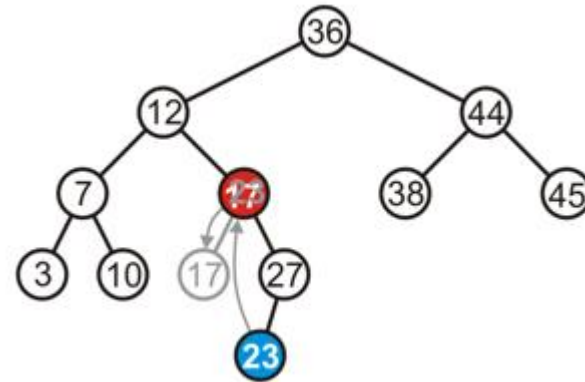
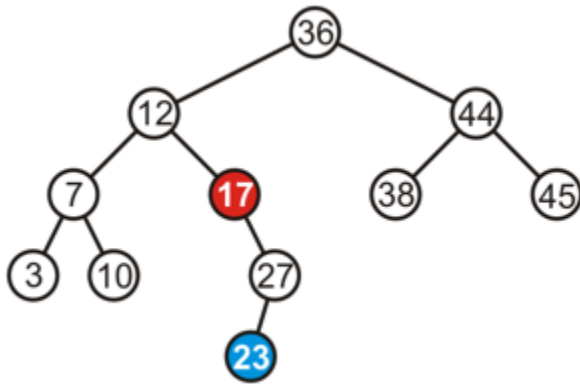


The heights of each of the sub-trees from here to the root are increased by one

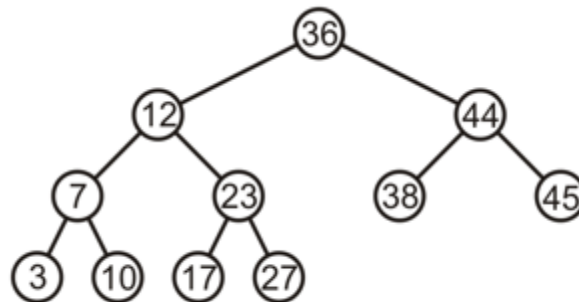


only two of the nodes are unbalanced: 17 and 36

# AVL Insertion



We only have to fix the imbalance at the lowest node

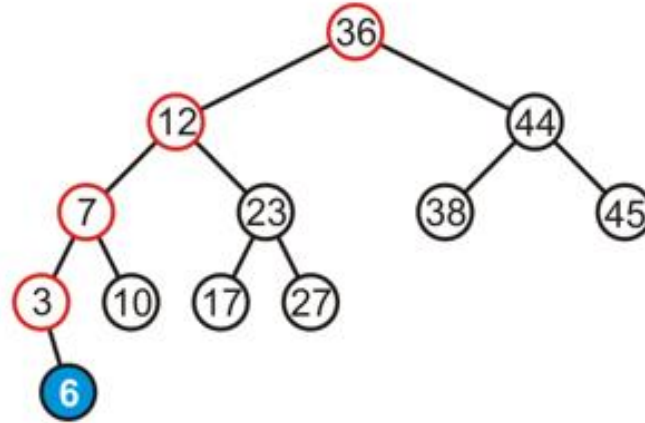
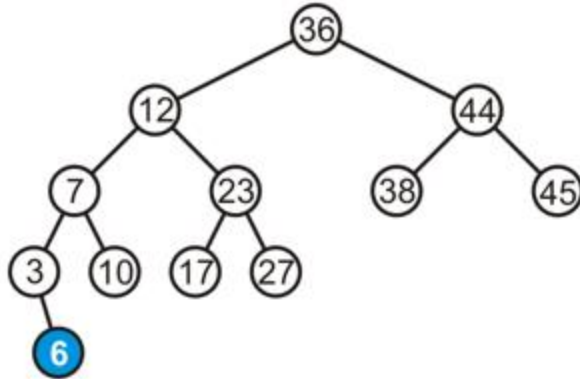


That node is no longer unbalanced. Incidentally, neither is the root. Now balanced again.

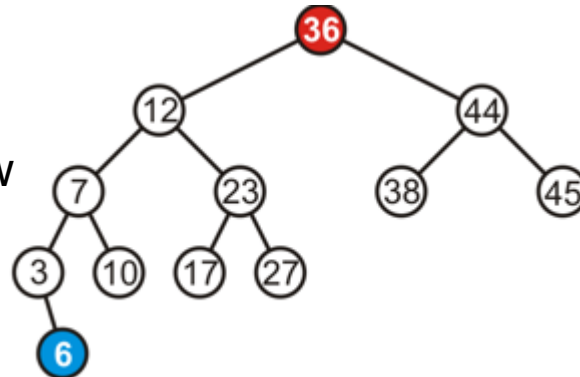
# AVL Insertion-Case 2

The height of each of the trees in the path back to the root are increased by one

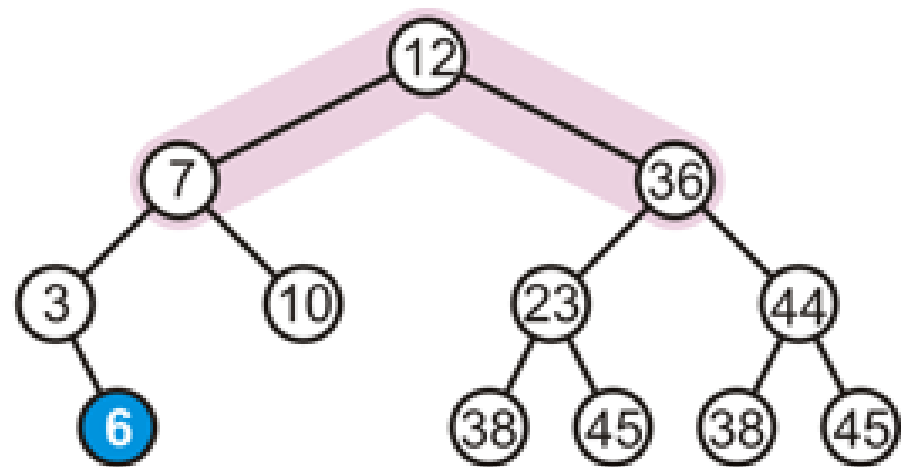
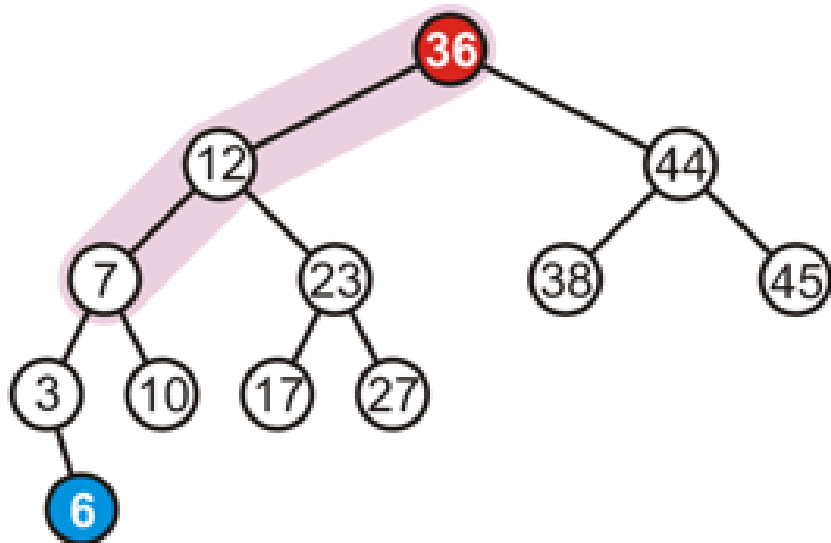
Consider adding 6



However, only the root node is now unbalanced



# AVL Insertion-Case 2



# AVL Tree-Creation



# Delete an element from AVL Trees



- We first do the normal BST deletion:
  - 0 children: just delete it
  - 1 child: delete it, connect child to parent
  - 2 children: put successor in your place
- Calculate Balance Factor again
- A is the critical node whose balance factor is disturbed upon deleting node x.
- If deleted node are from left subtree of A then It is called **Type L** delete otherwise it is called **Type R** delete



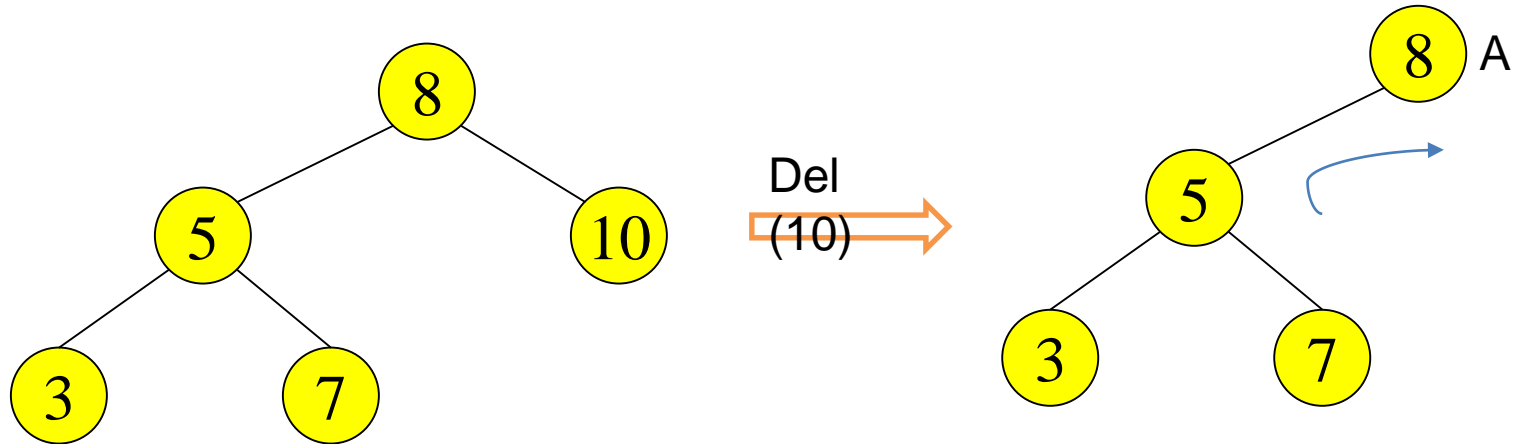
# Delete an element from AVL Trees

## R0 Rotation

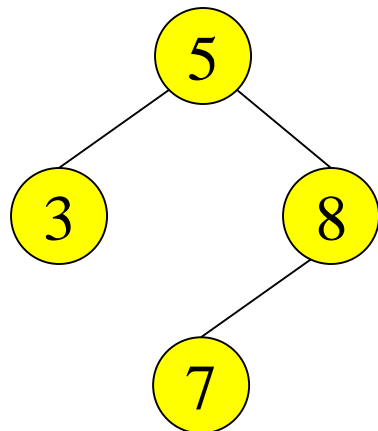
---



# R0 Rotation



Apply right rotation on A



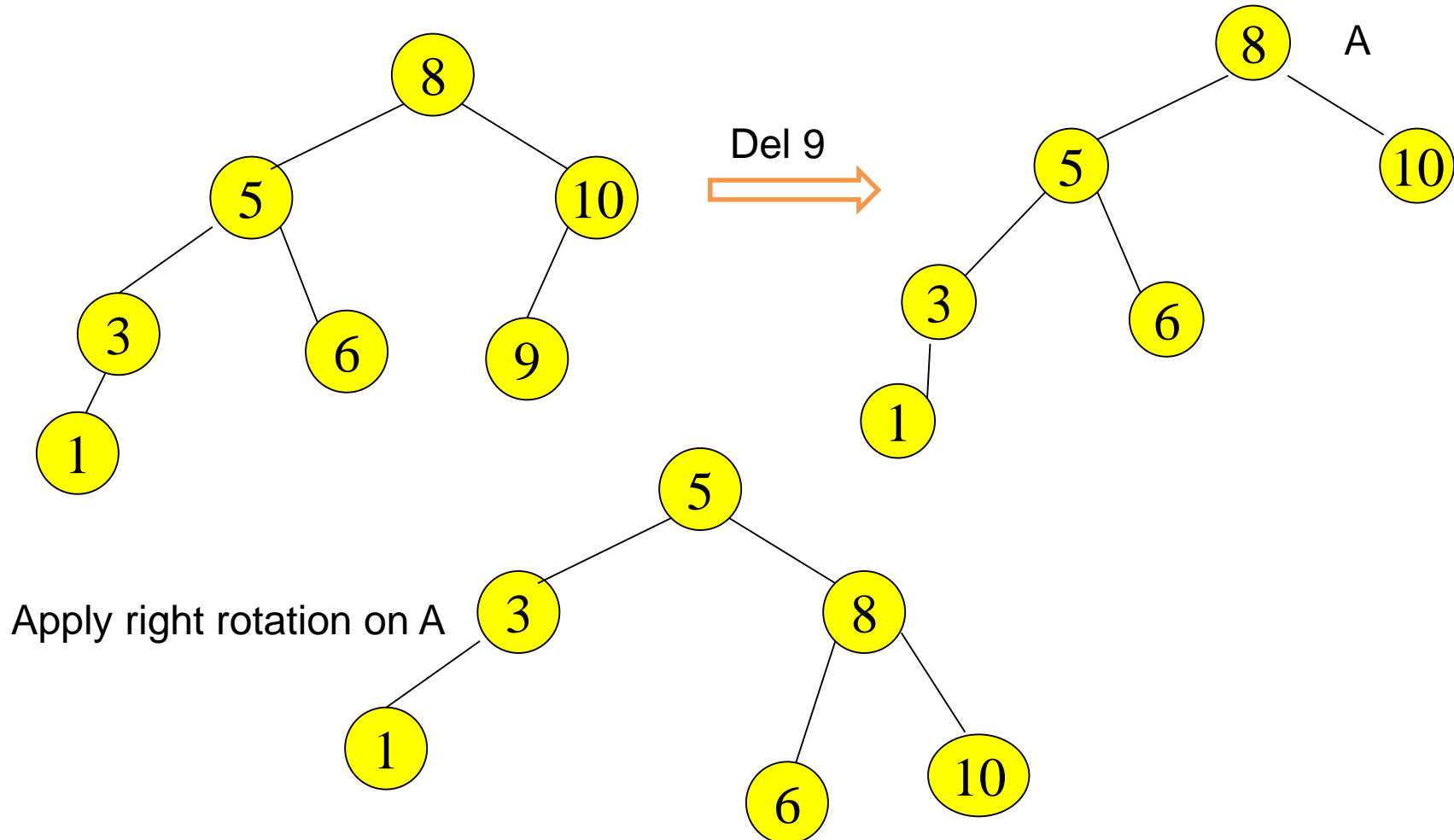
# Delete an element from AVL Trees

## R1 Rotation

---



# R1 Rotation



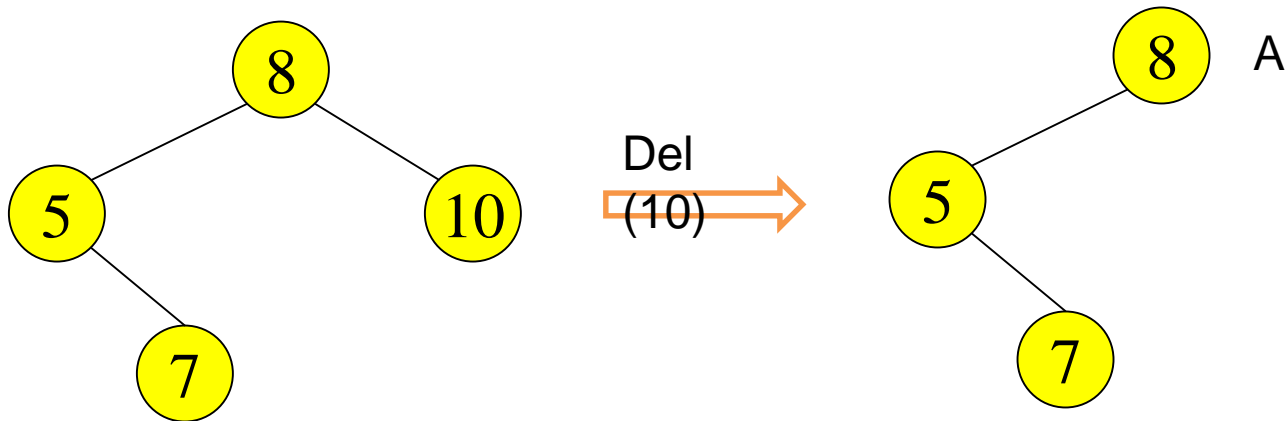
# Delete an element from AVL Trees

## R -1 Rotation

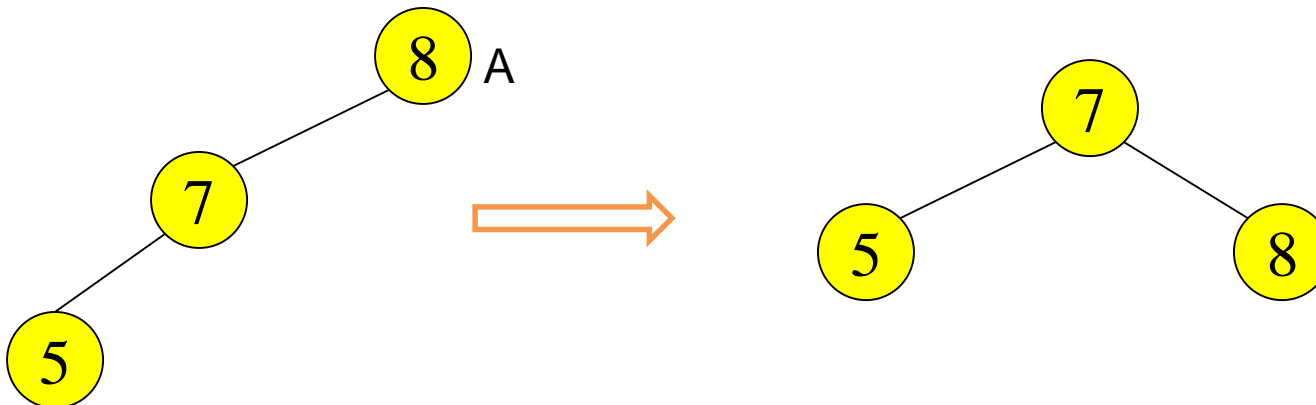
---



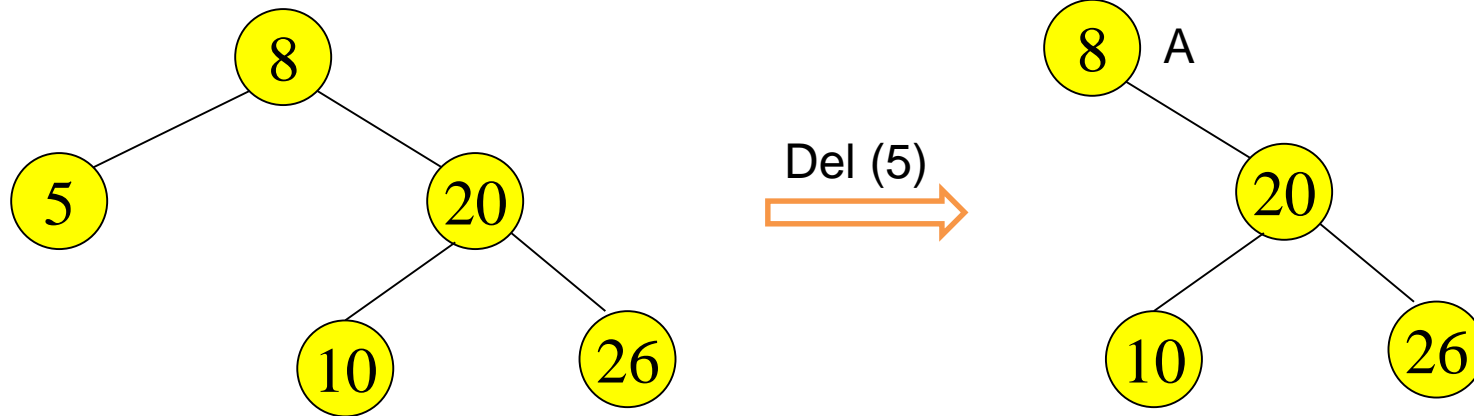
# Type R-1 Rotation



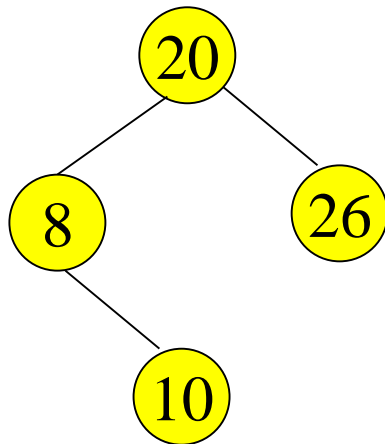
Apply left Rotation on left child of node A and Then right rotation on node A



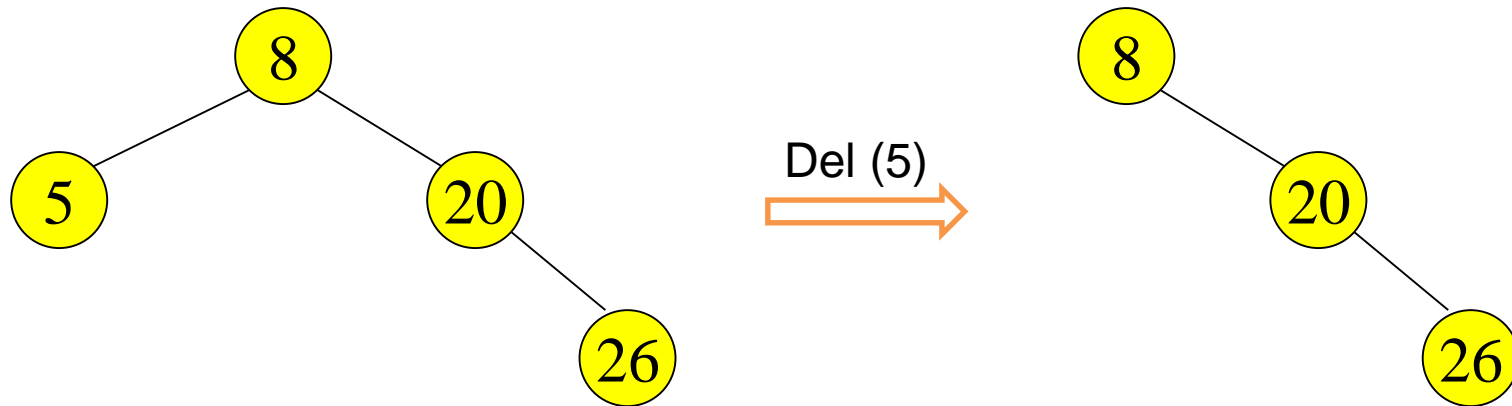
# Type L



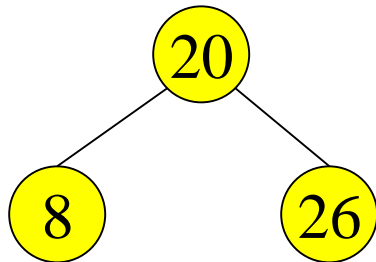
Since it L(0) type apply Left rotation on A



# Type L

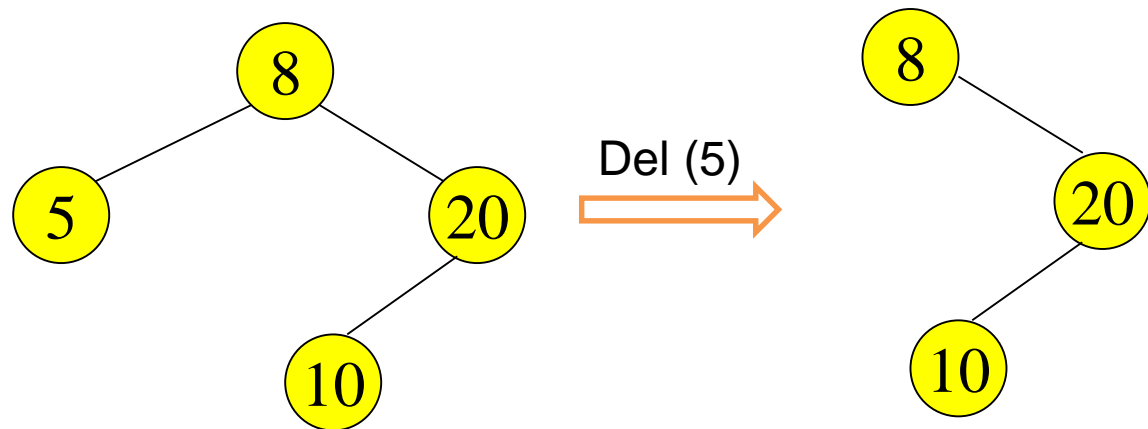


Since it L(-1) = case apply Left rotation on A

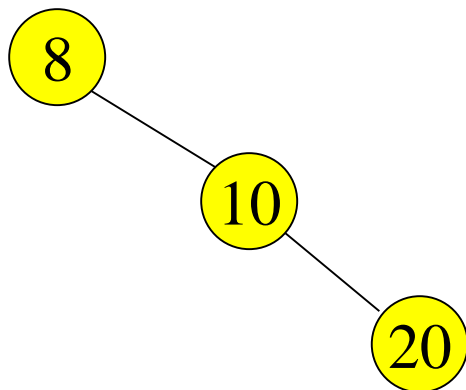




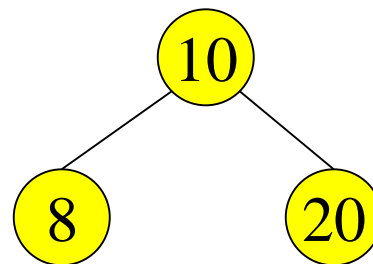
# Type L



Since it L(1), In  $L_1$  case we have to solve in two steps,  
Step1: Right Rotation at right child of 'A'



Step2: Left rotation at node A



# AVL trees-Deletions



- Removing a node from an AVL tree may cause more than one AVL imbalance
- Like insert, remove must check after it has been successfully called on a child to see if it caused an imbalance
- Unfortunately, it may cause  $O(h)$  imbalances that must be corrected
- Insertions will only cause one imbalance that must be fixed
- But in removal, a single trinode restructuring may not restore the height-balance property globally
- So, after rebalancing, we continue walking up  $T$  looking for unbalanced nodes.
- If we find another, we perform a restructure operation to restore its balance, and continue marching up  $T$  looking for more, all the way to the root

# AVL trees-Deletions



# AVL trees-Deletions

---



# AVL trees-Deletions



# AVL trees-Deletions



# AVL trees-Applications



- AVL trees are applied in the following situations:
  - There are few insertion and deletion operations
  - Short search time is needed

# AVL Trees-Summary



- AVL balance is defined by ensuring the difference in heights is 0 or 1
- Insertions and Removals are like binary search trees
- Each insertion requires at least one correction to maintain AVL balance
- Removals may require  $O(h)$  corrections
- These corrections require  $Q(1)$  time
- Height of the AVL tree is  $O(\log(n))$
- $\therefore$  all  $O(h)$  operations are  $O(\log(n))$





# THANK YOU!!!

**BITS Pilani**  
Hyderabad Campus

