Data Structures and Algorithms Design

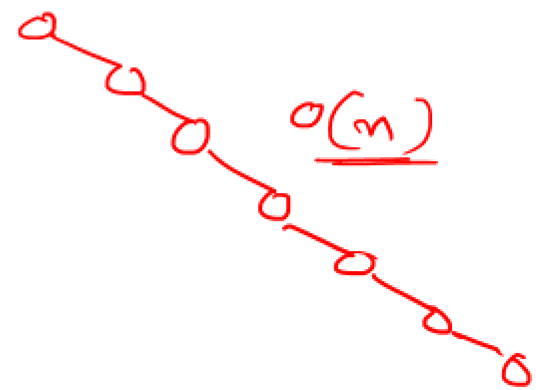
BITS Pilani
Hyderabad Campus

Febin.A.Vahab

AVL trees

1 2 3

- From previous lectures:
  - Binary search trees store linearly ordered data
  - Best case height: O(log(n))
  - Worst case height: O(n)

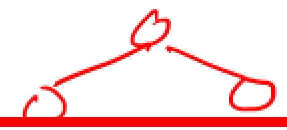


- Requirement:
  - Define and maintain a balance to ensure O(log(n)) operations

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



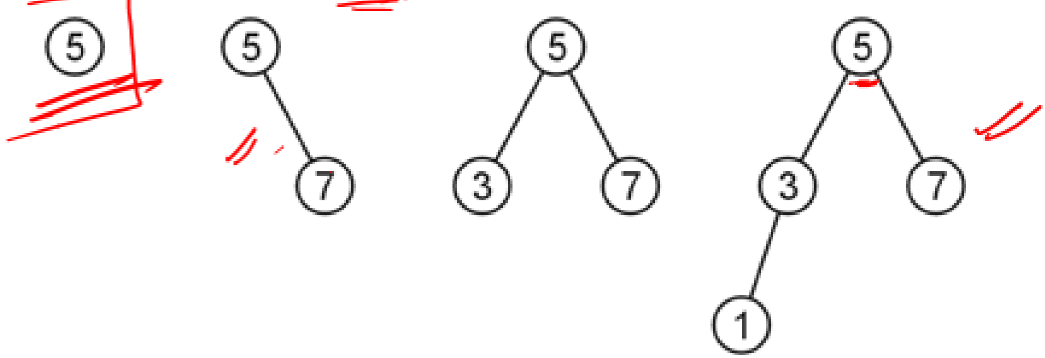
- AVL trees are balanced
- An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1
- This difference is called the **Balance Factor**.
- For an AVL tree |balance factor|<=1 for all the nodes.

AVL trees

1 0 -1

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

0-0=0 0-1=-1 1-1=0 2-1=1



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

AVL trees

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

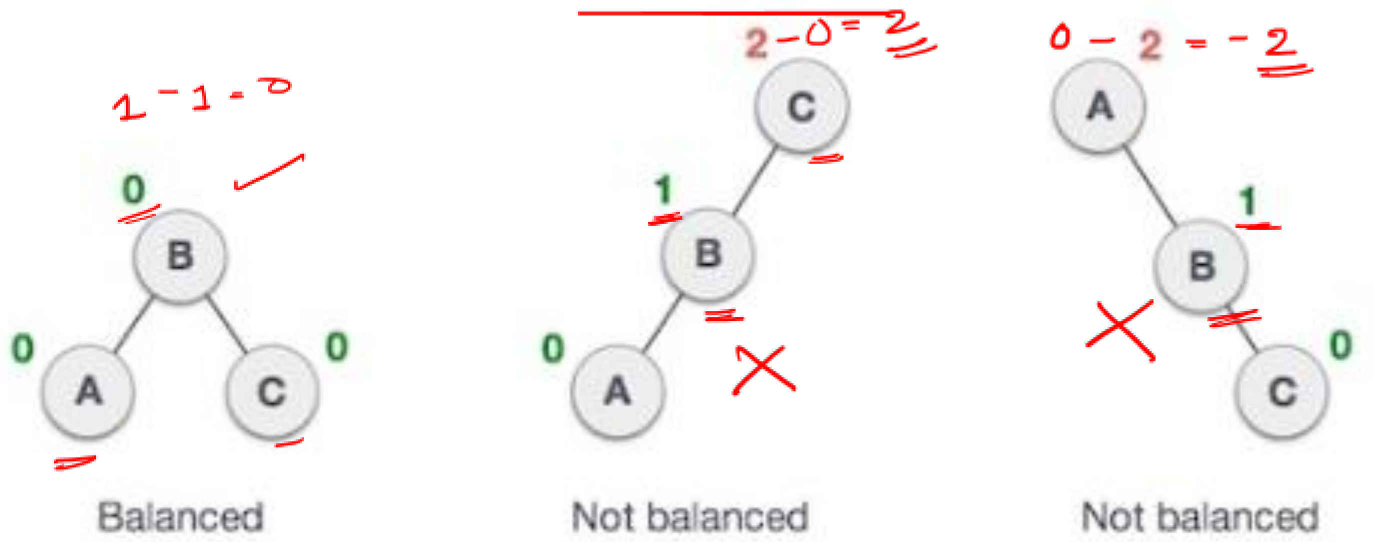
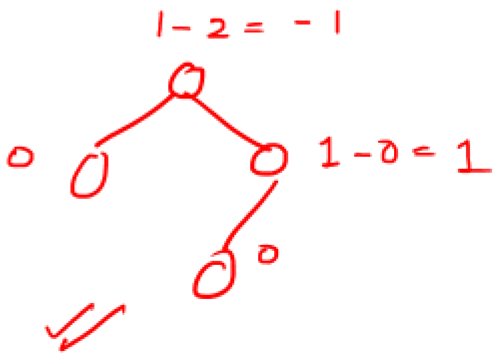
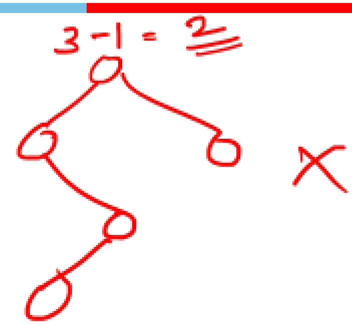
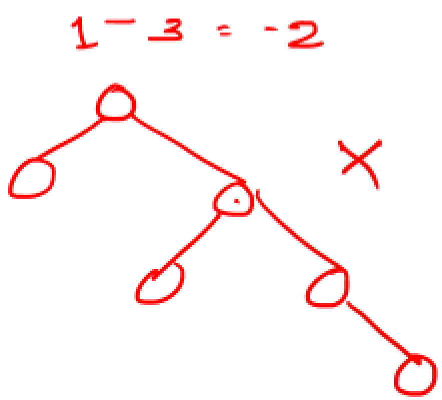


Image credit: Tutorials point

AVL Trees-Example







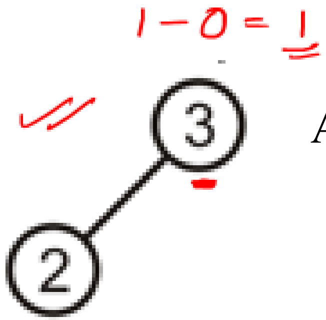
AVL trees-Rotations

- To balance itself, an AVL tree may perform the following four kinds of rotations −
  - Left rotation
  - Right rotation
  - Left-Right rotation
  - Right-Left rotation

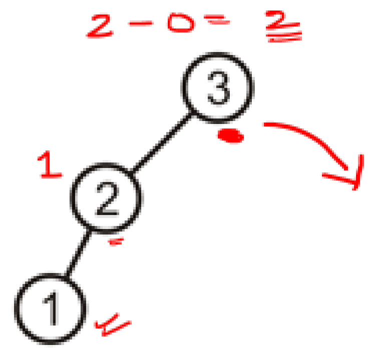
  - To have an unbalanced tree, we at least need a tree of height 2.

AVL trees-Rotations

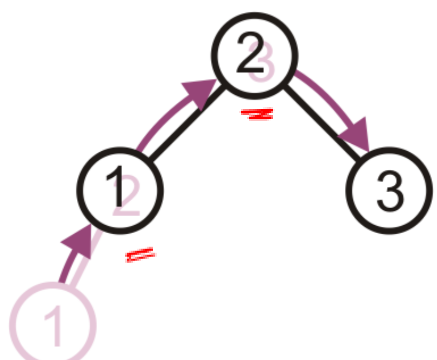
- **Right Rotation** Node is inserted in the left of left-subtree



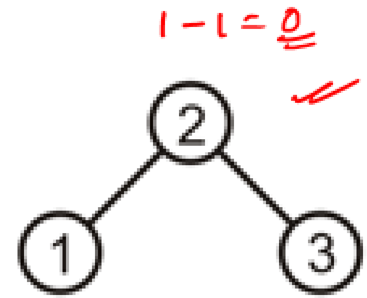
Add 1



The node 3 has become unbalanced

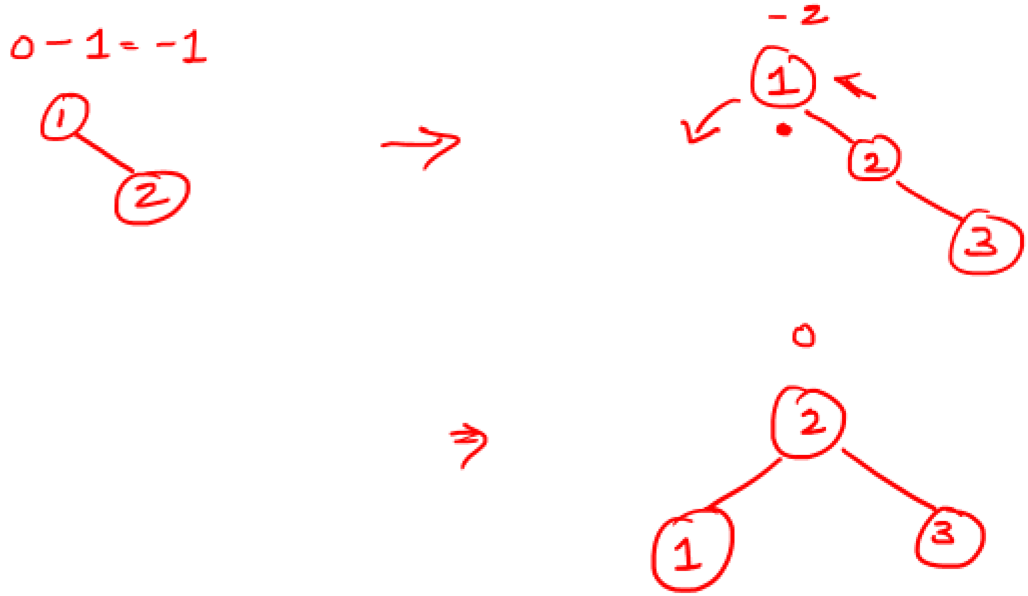


Promote 2 to the root, demote 3 to be 2's right child, and 1 remains the left child of 2



AVL trees-Rotations

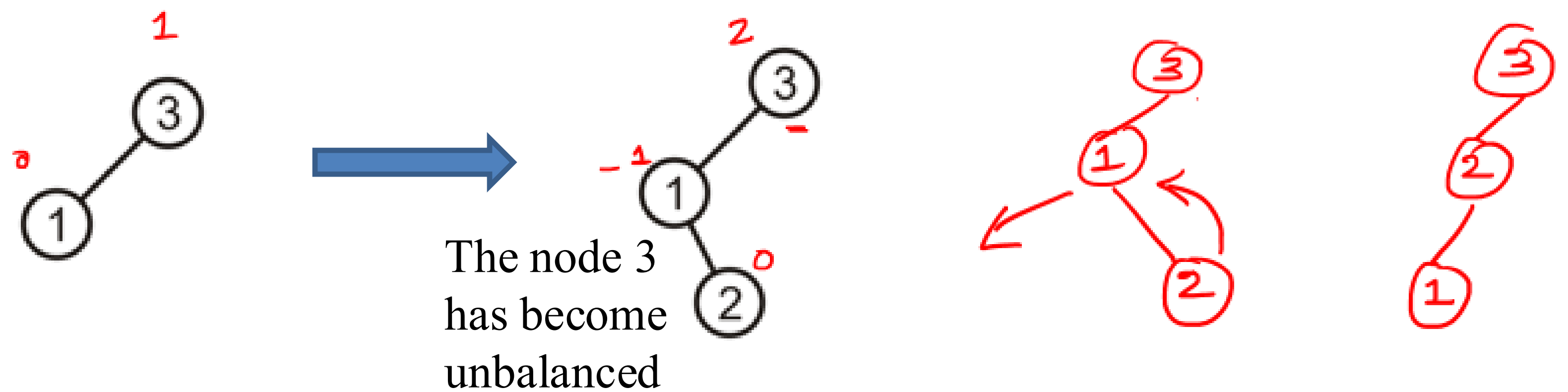
- **Left Rotation:** Node is inserted into right of right subtree. After inserting new node, tree becomes unbalanced

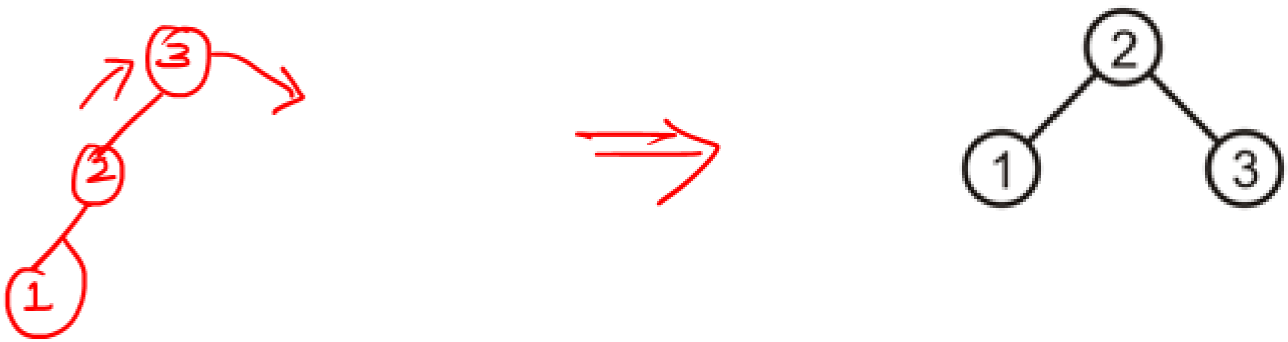


AVL trees-Rotations

LR unbalance

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

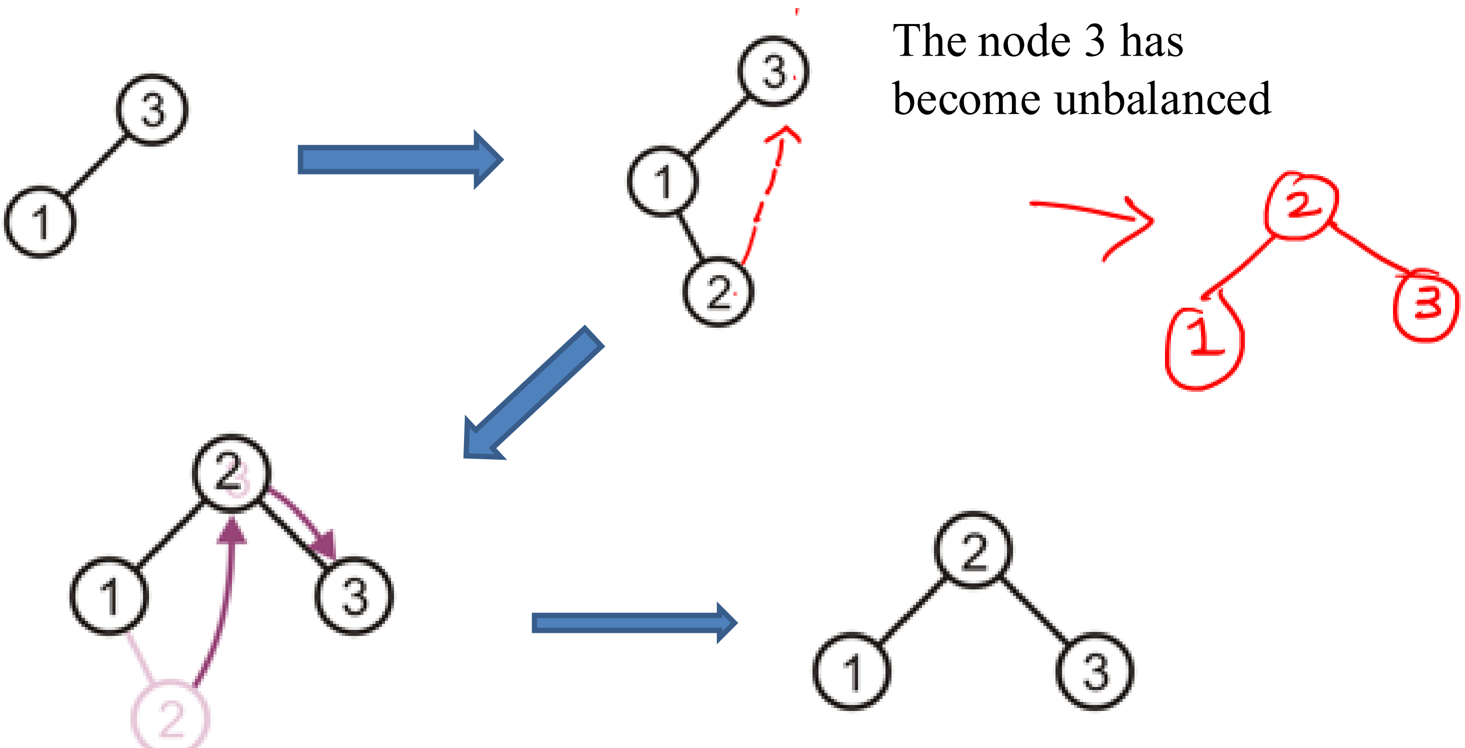




AVL trees-Rotations

- **Left Right Rotation**

The node 3 has become unbalanced

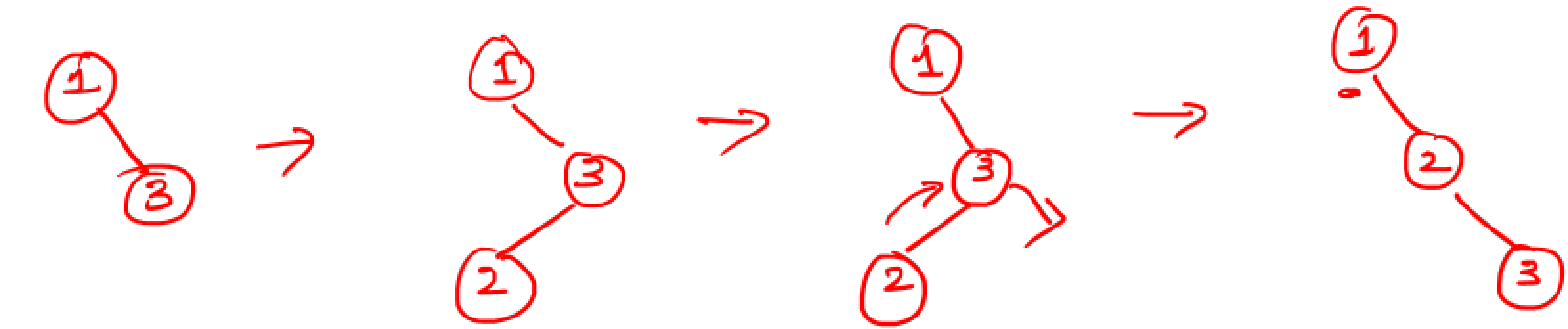


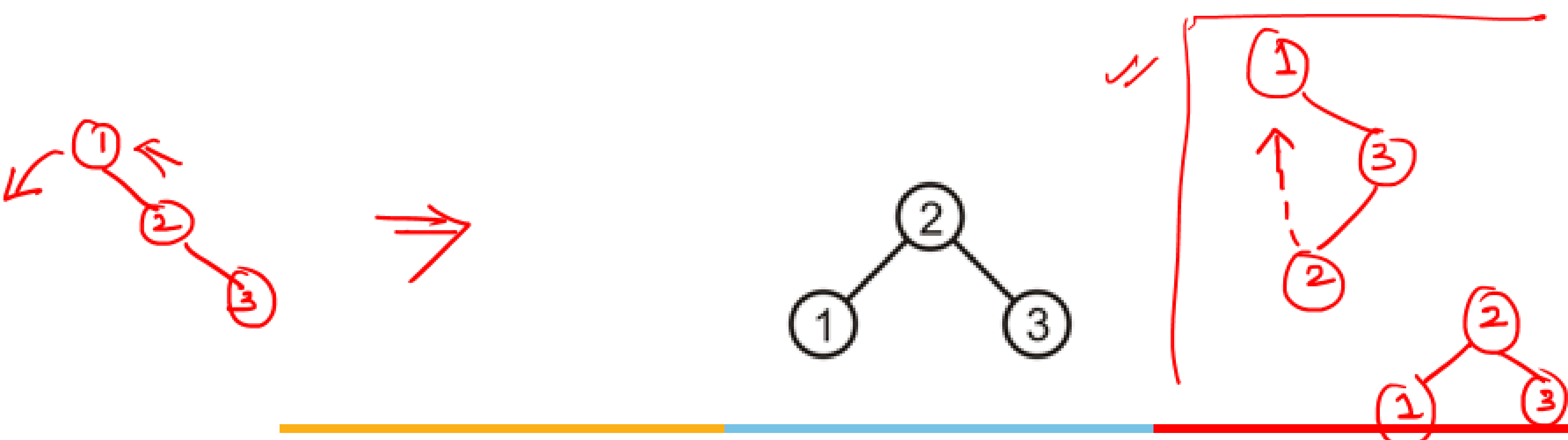
Promote 2 to the root, and assign 1 and 3 to be its children

AVL trees-Rotations

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

0 - 2 = -2

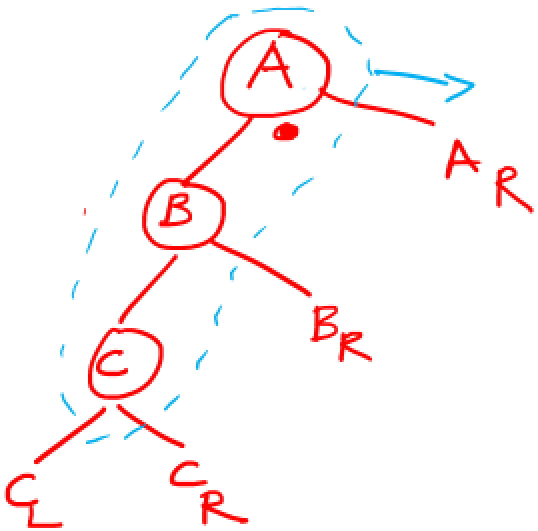


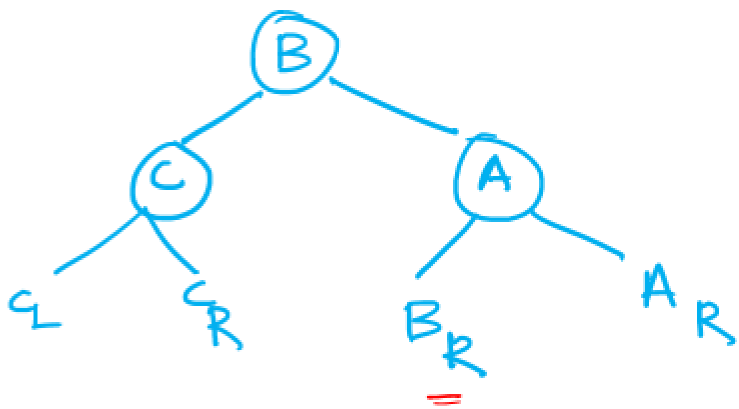


AVL Trees-General Case
LL Imbalance

Right Rotation

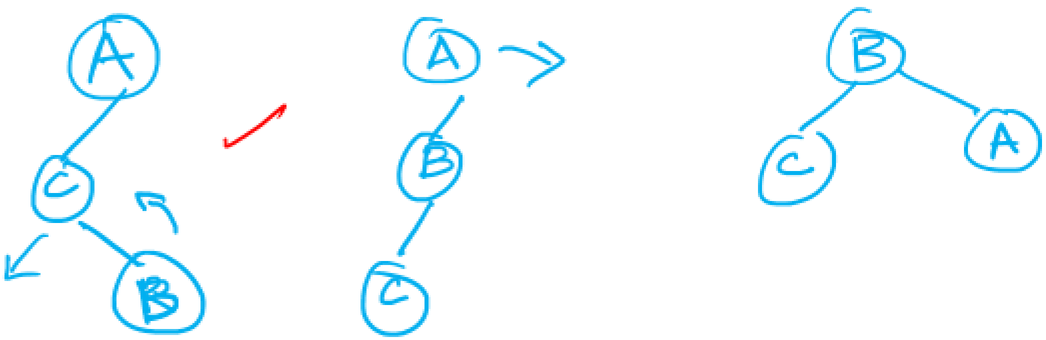
Imbalance caused by insertion of node on the left of the left subtree.

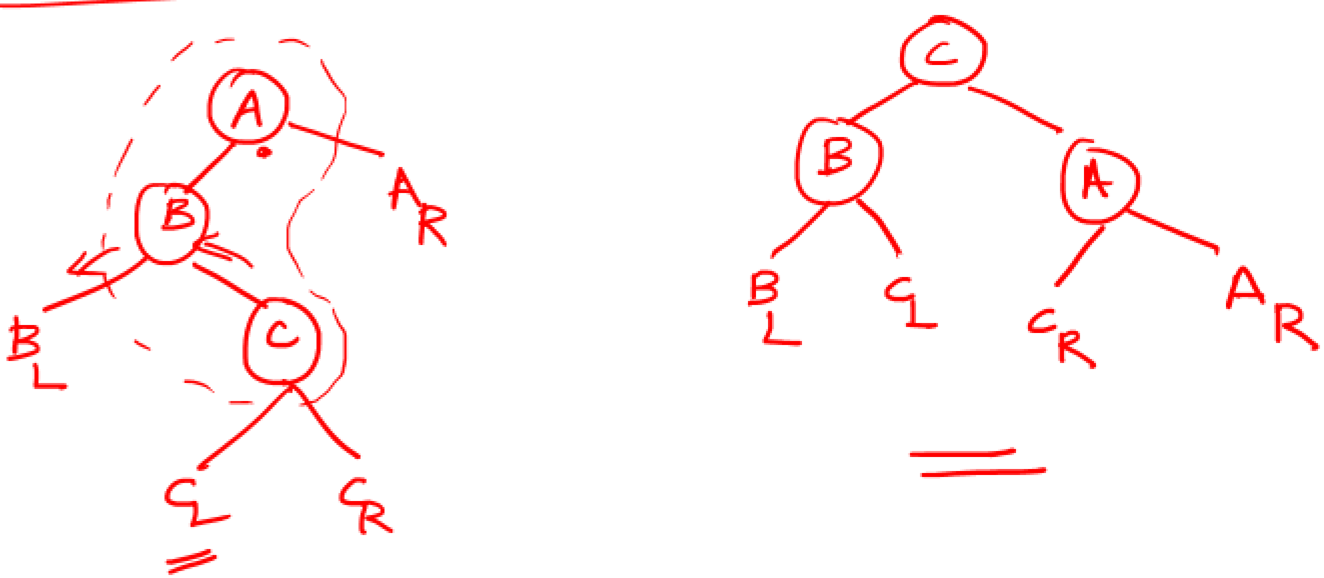




AVL Trees-General Case
LL Imbalance

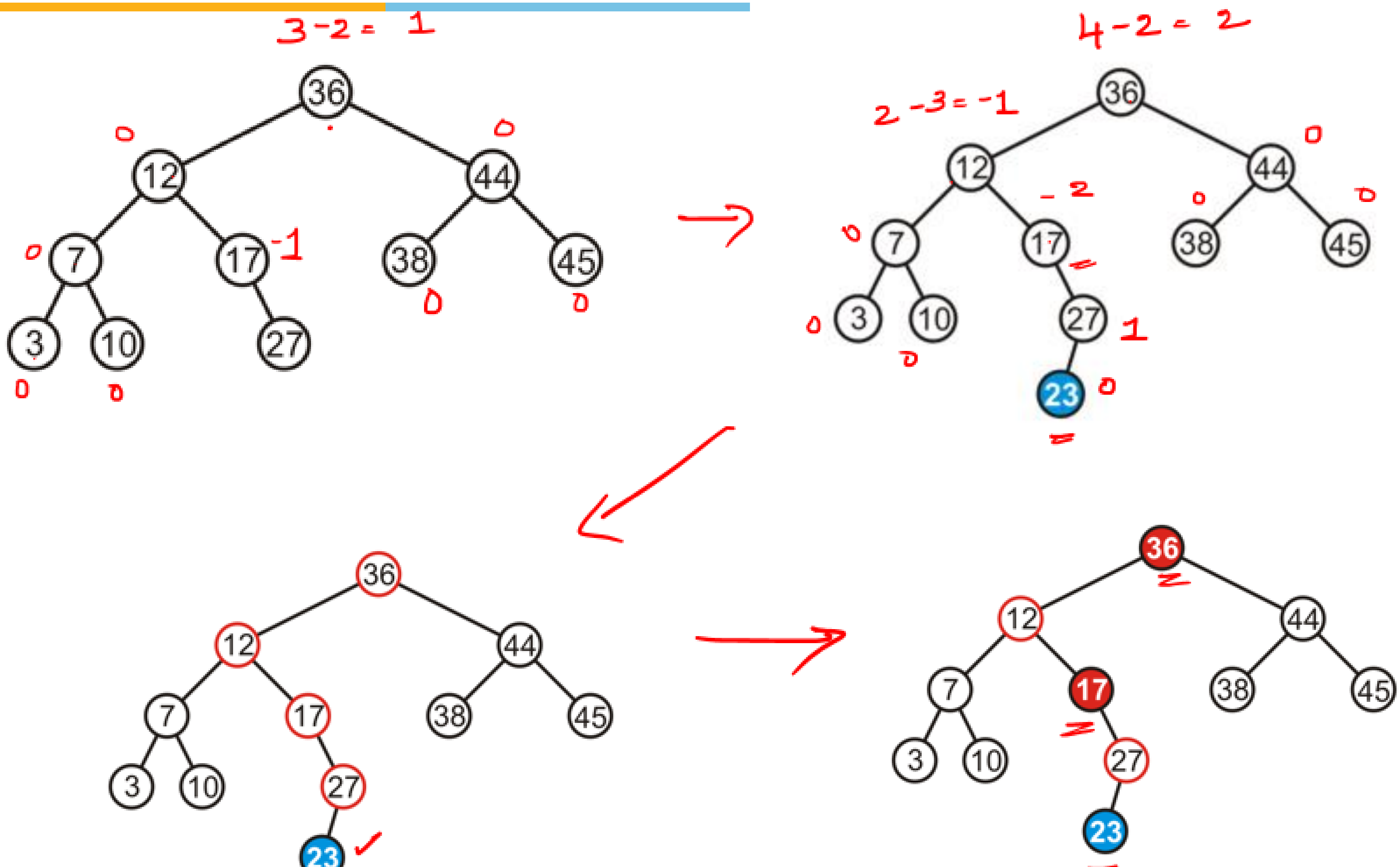
AVL Tree-General Case
LR imbalance = LR Rotation





AVL Tree-General Case
LR imbalance

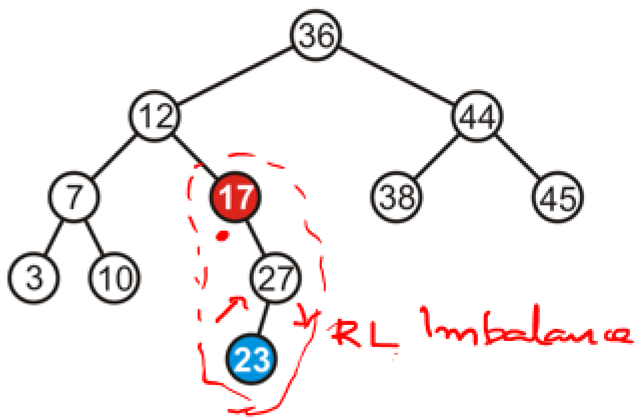
AVL Insertion-Case 1

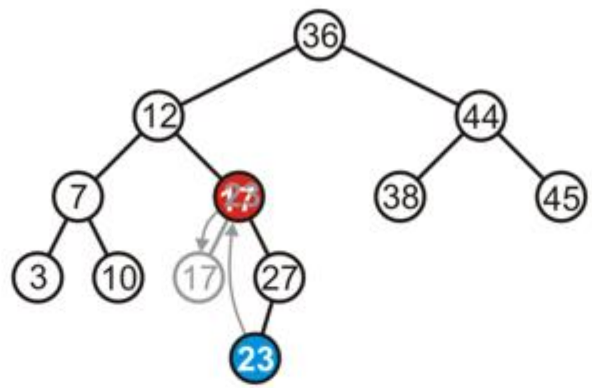


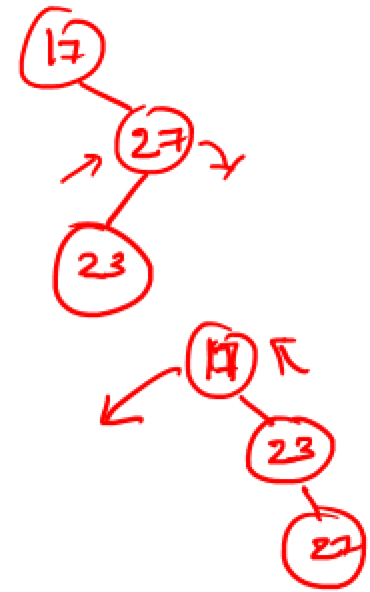
The heights of each of the sub-trees from here to the root are increased by one

only two of the nodes are unbalanced: 17 and 36

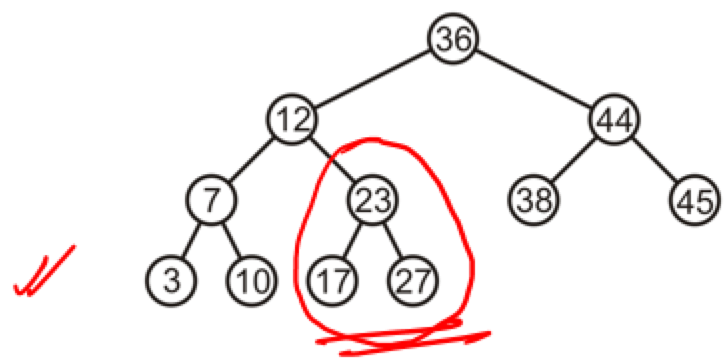
AVL Insertion







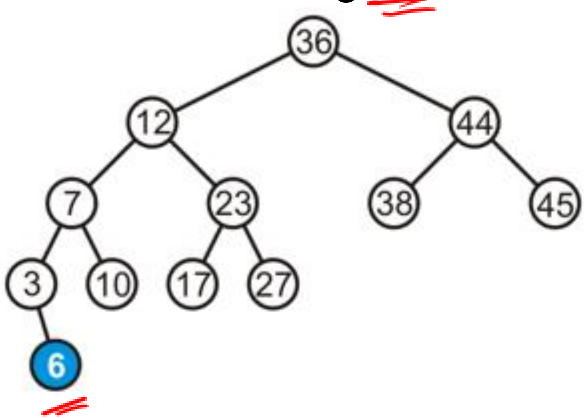
We only have to fix the imbalance at the lowest node



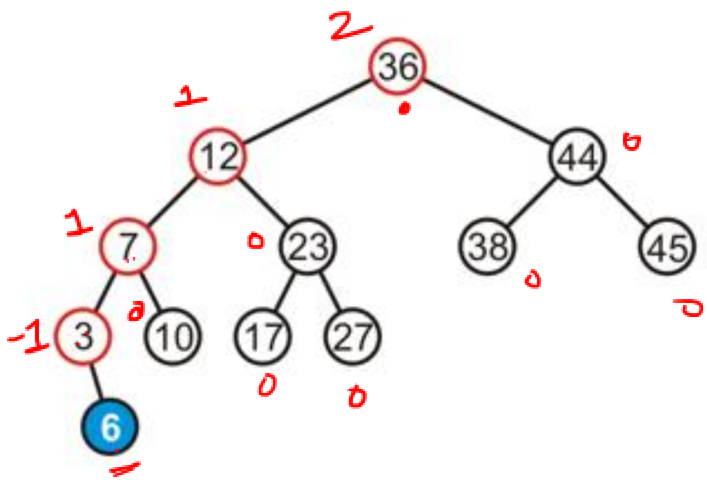
That node is no longer unbalanced. Incidentally, neither is the root .Now balanced again.

AVL Insertion-Case 2

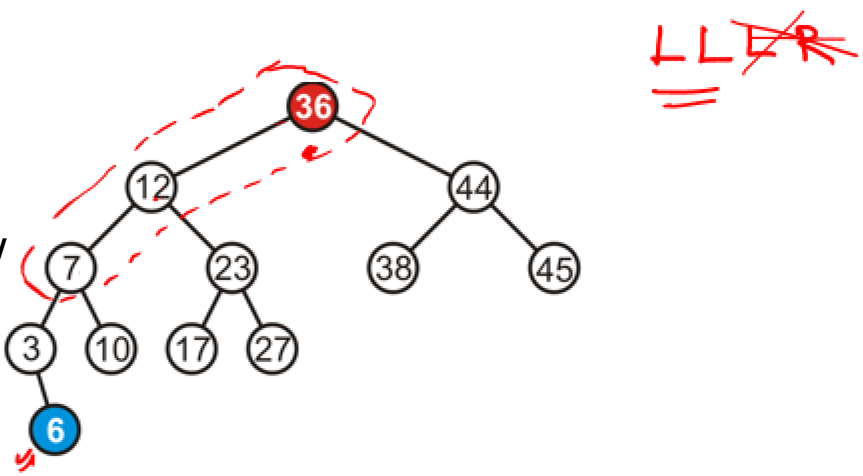
Consider adding 6



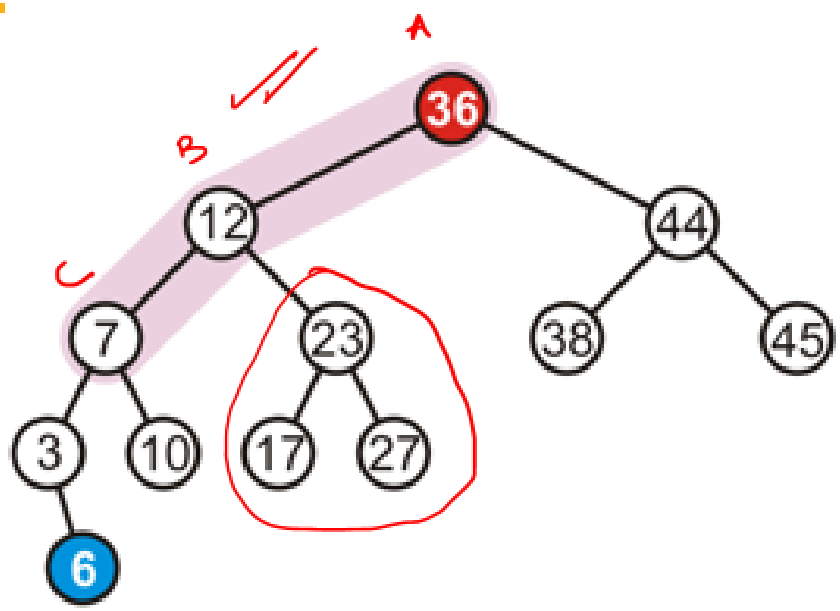
The height of each of the trees in the path back to the root are increased by one

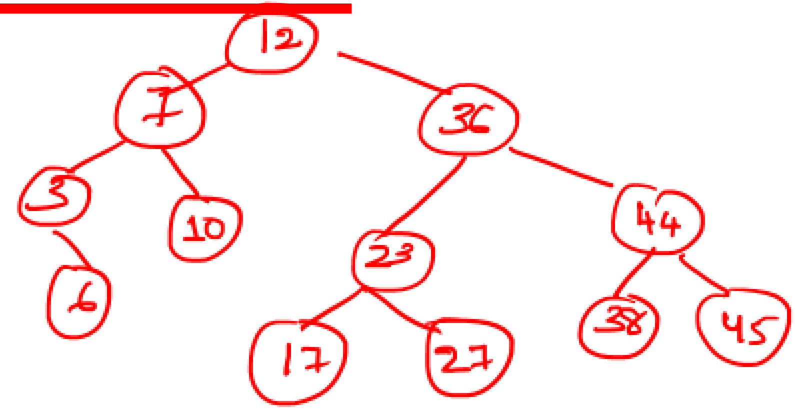


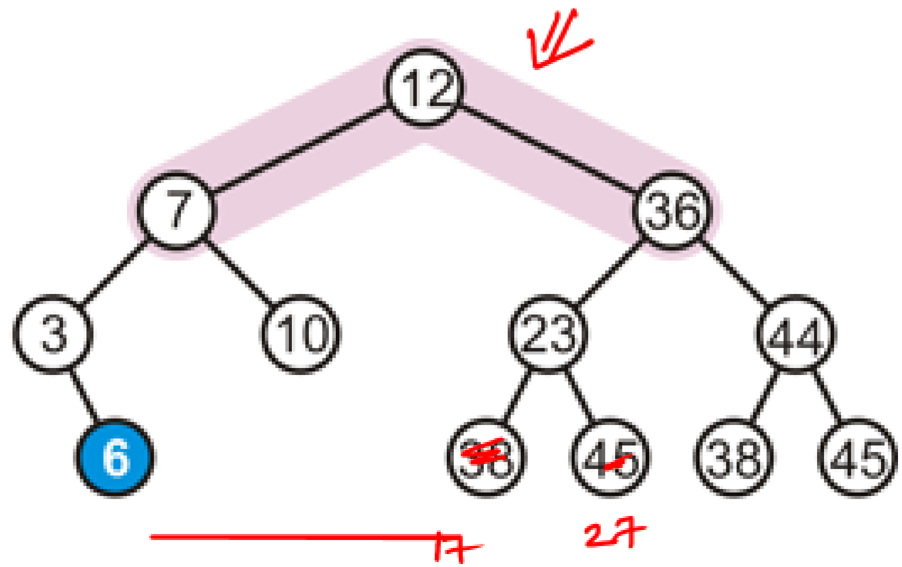
However, only the root node is now unbalanced



AVL Insertion-Case 2



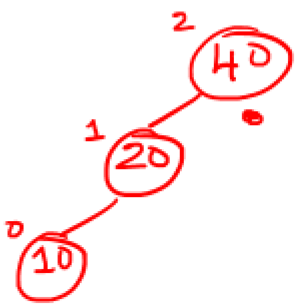




AVL Tree-Creation

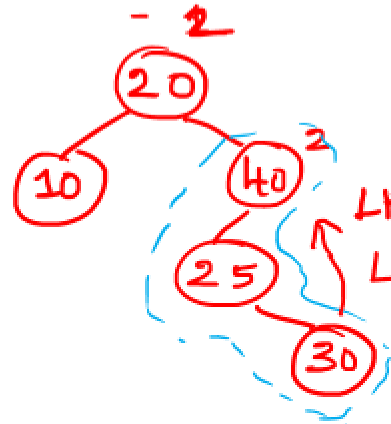
innovate achieve lead

40, 20, 10, 25, 30, 22, 50

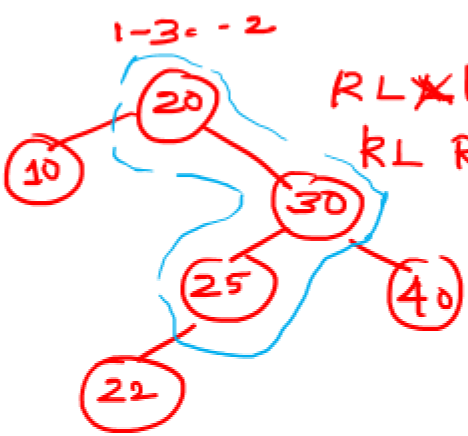


LL Imbalance
R Rotation

⇒

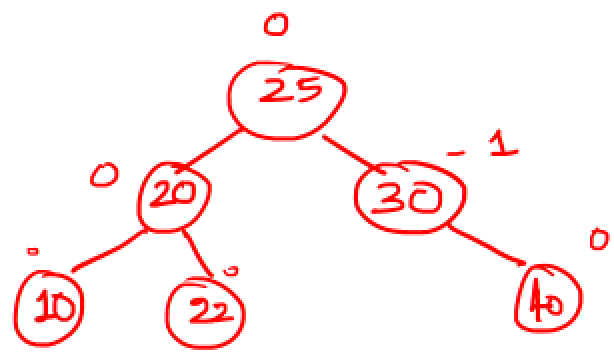


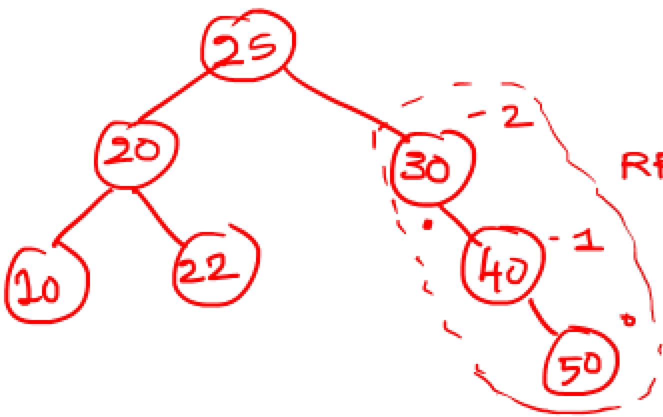
LR Imbalance
LR Rotation



RL Imbalance
RL Rotation

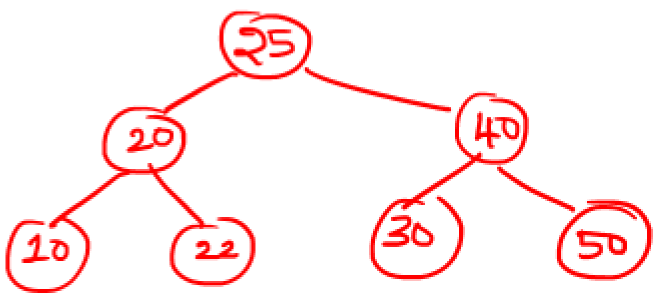
⇒



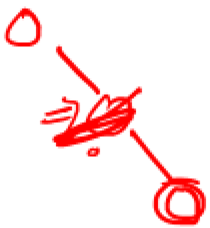


RR Imbalance

Left Rotation



Delete an element from AVL Trees



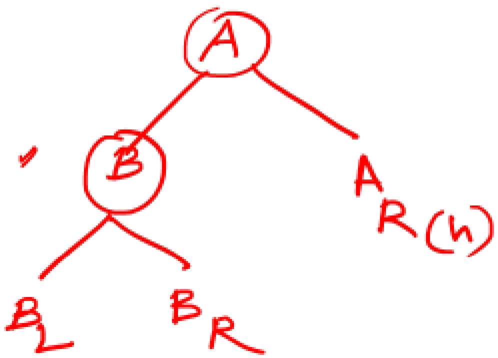
- We first do the normal BST deletion:
  - 0 children: just delete it
  - 1 child: delete it, connect child to parent
  - 2 children: put the inorder successor in node's place
- Calculate Balance Factor again
- A is the critical node whose balance factor is disturbed upon deleting node x.
- If deleted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

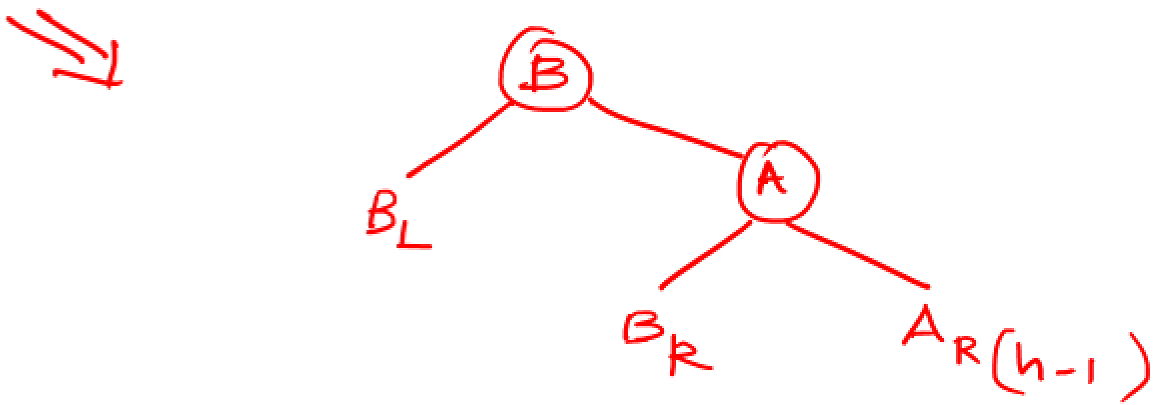
Delete an element from AVL Trees
R0 Rotation

bf of B = 0

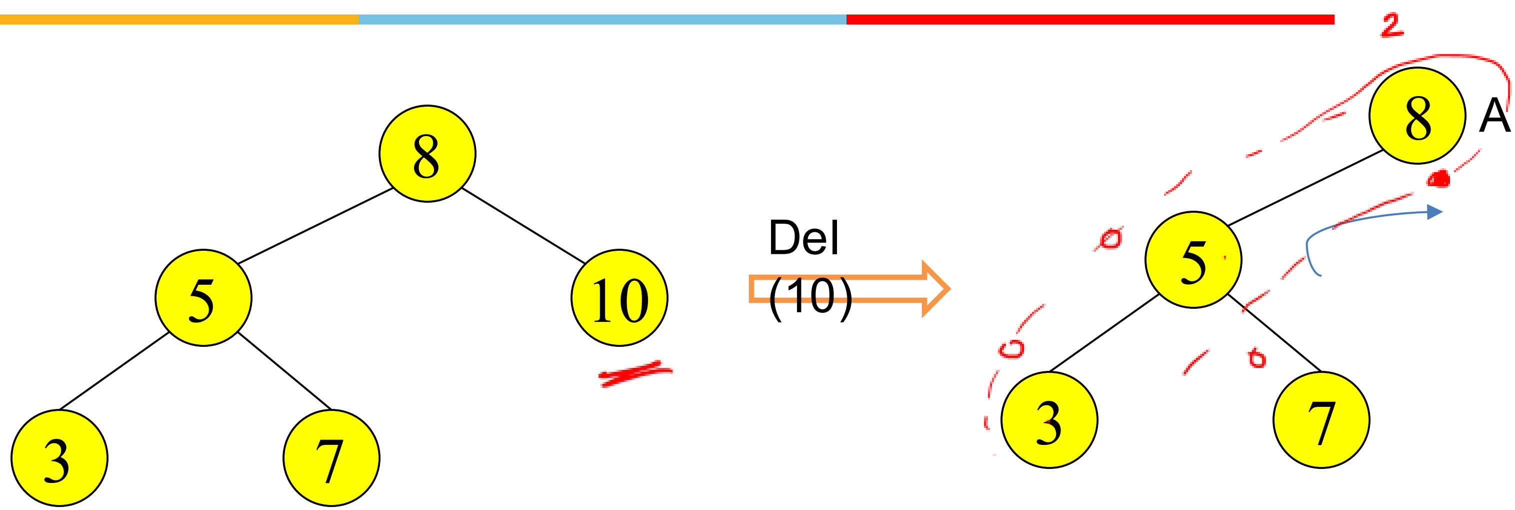
Delete a node from Right subtree of A

Perform a Right Rotation (R0)

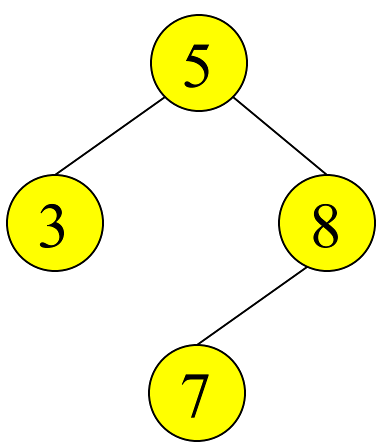




R0 Rotation



Apply right rotation on A

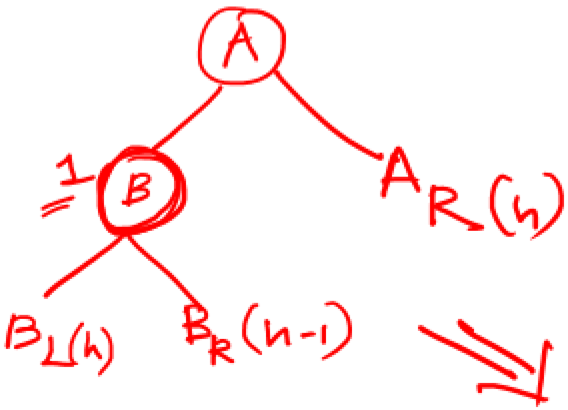


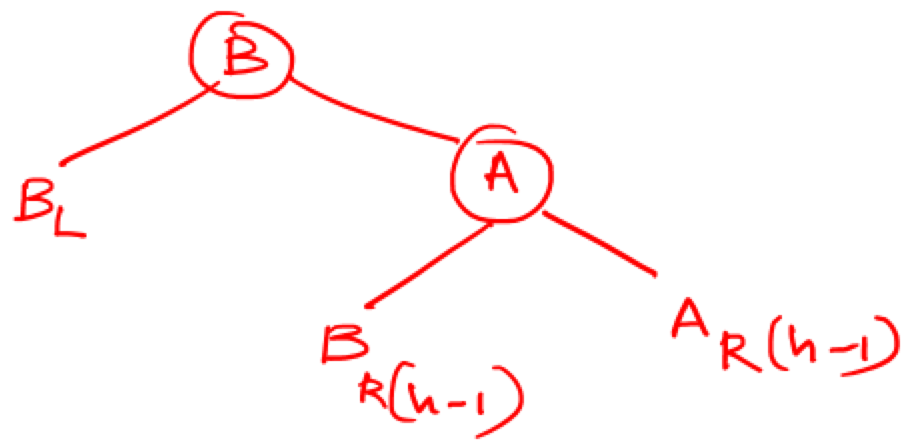
Delete an element from AVL Trees R1 Rotation

bf of B = 1

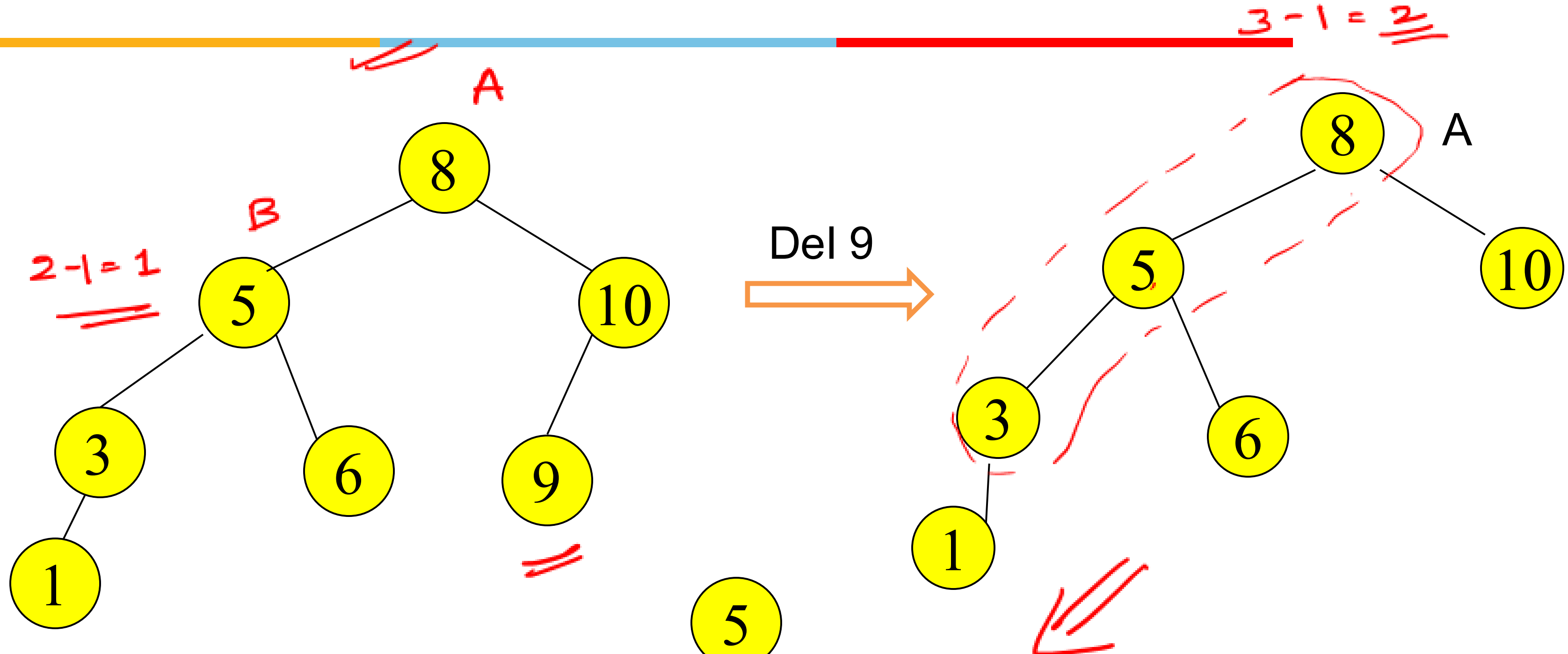
Delete a node from right subtree of A

Perform Right Rotation (R1 Rotation)

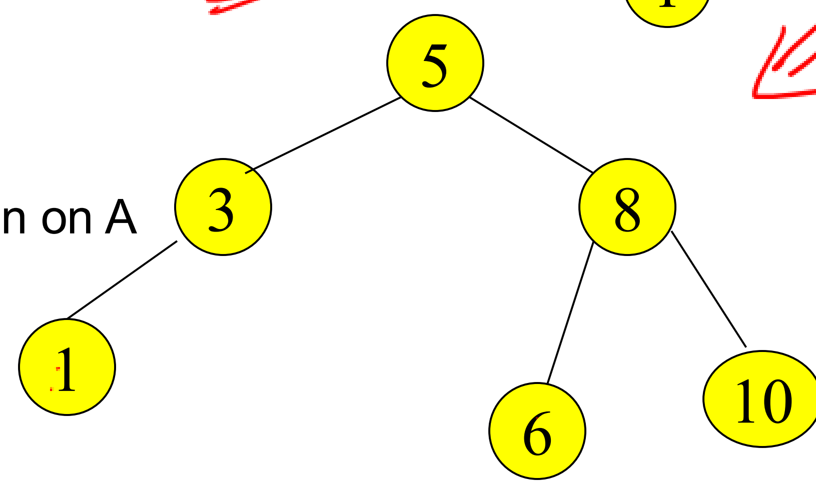




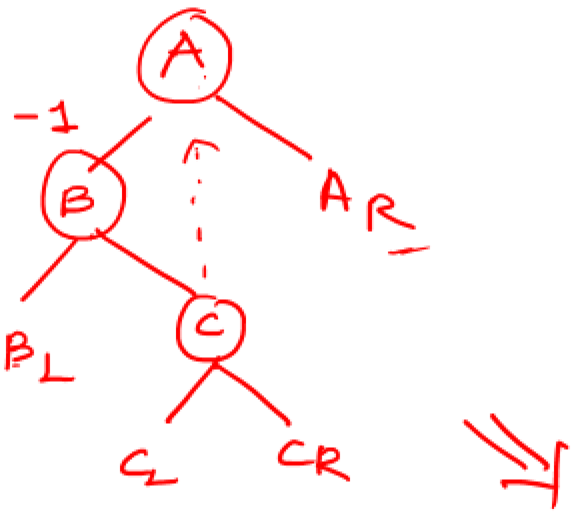
R1 Rotation



Apply right rotation on A



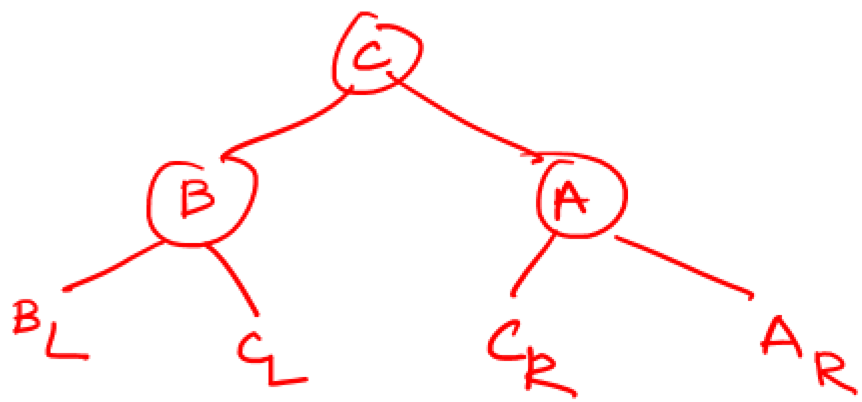
Delete an element from AVL Trees
R -1 Rotation



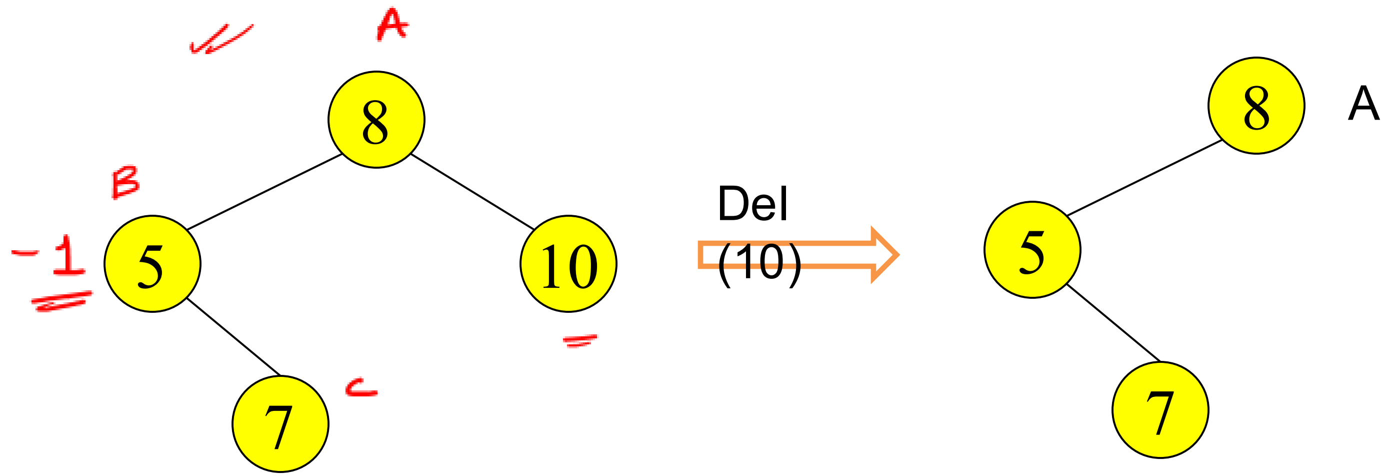
bf of B = -1

Delete node from right subtree of A

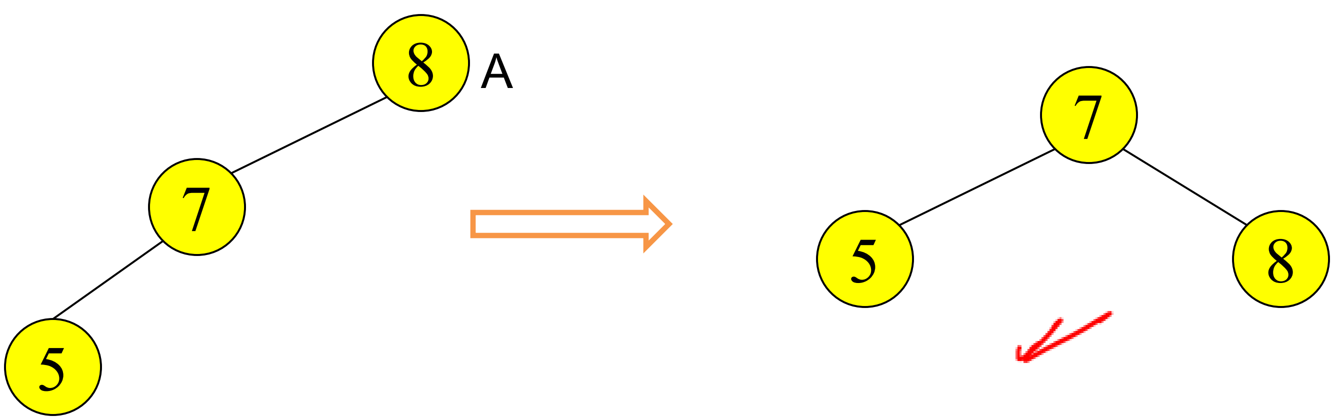
Perform Right Rotation (R-1 Rotation)



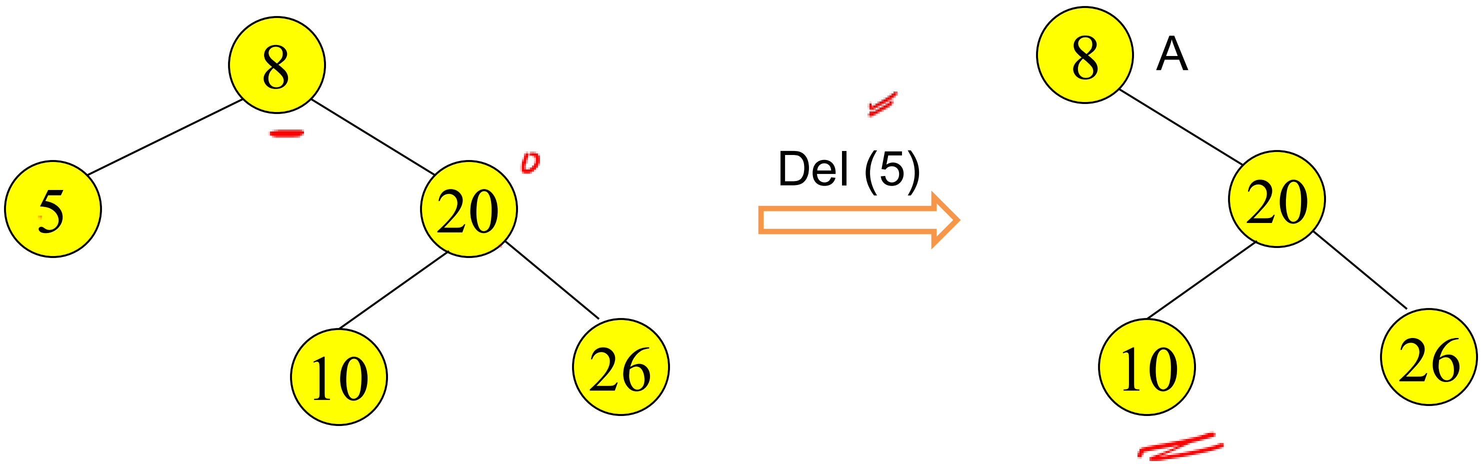
Type R-1 Rotation



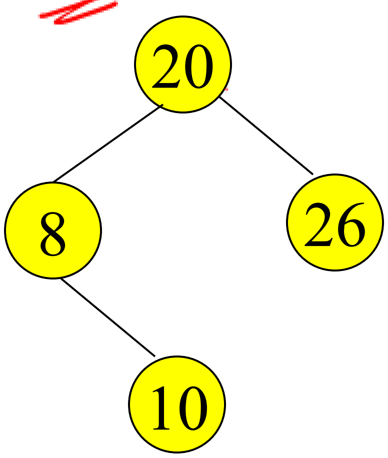
Apply left Rotation on left child of node A and Then right rotation on node A



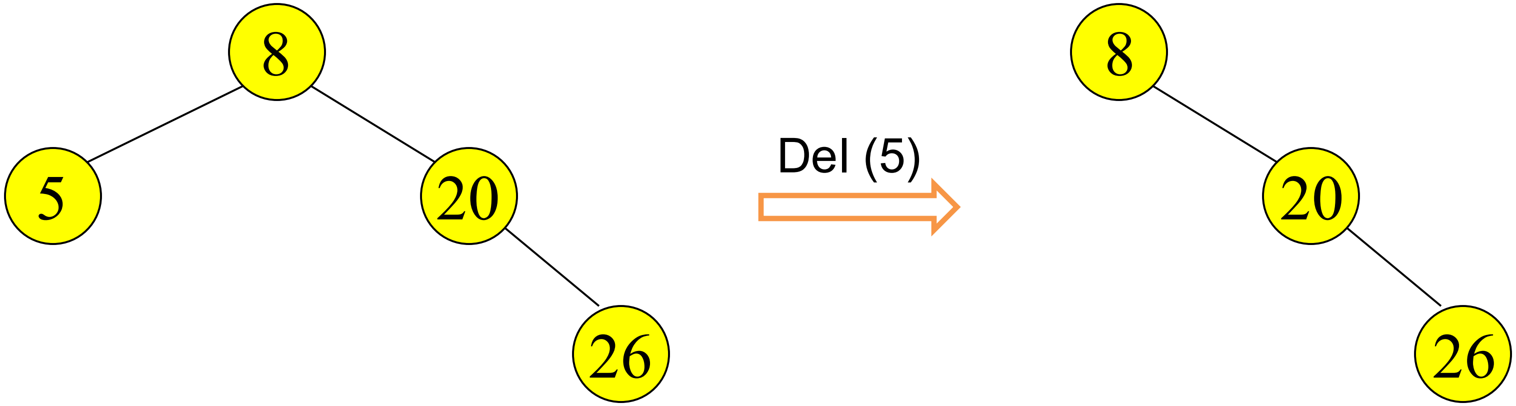
Type L



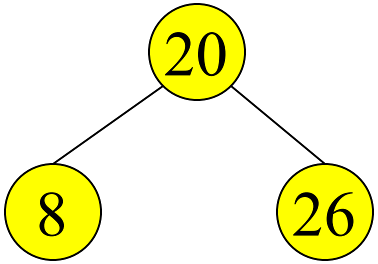
Since it L(0) type apply Left rotation on A



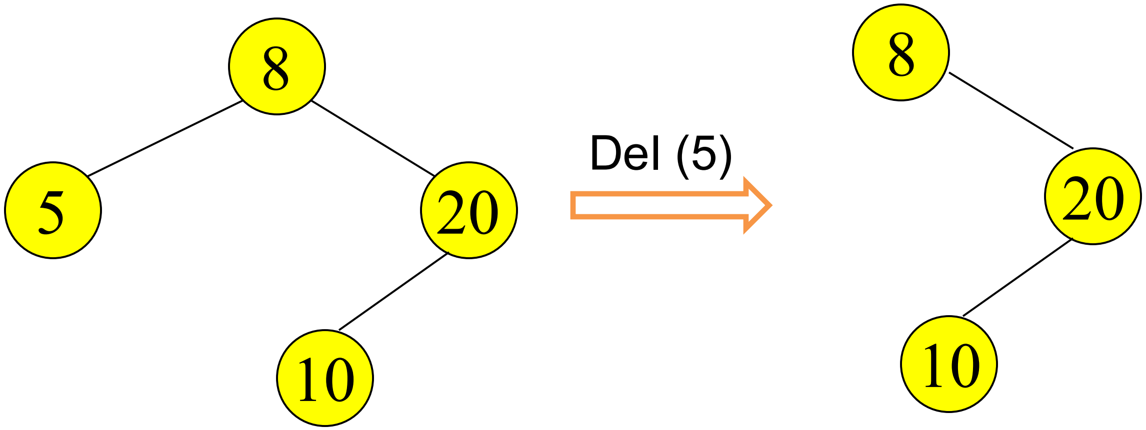
Type L



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



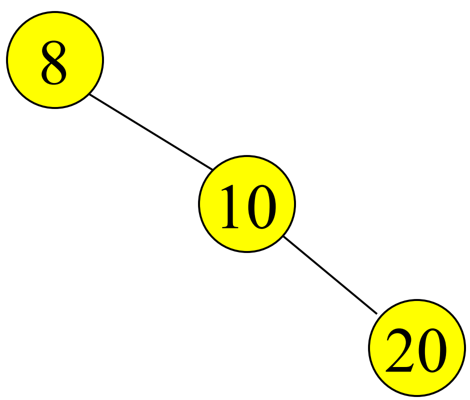
Type L



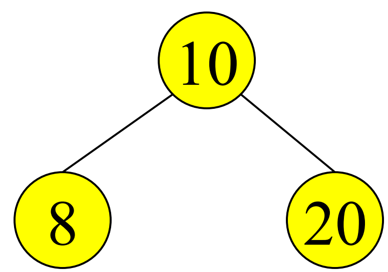
RL

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



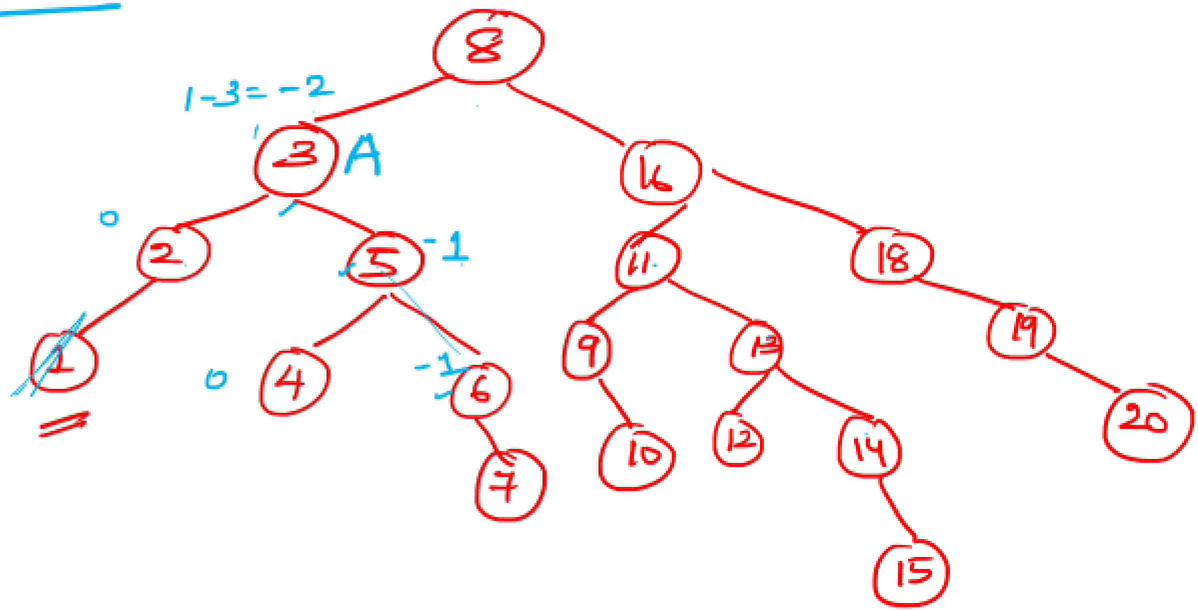
AVL trees-Deletions

- 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

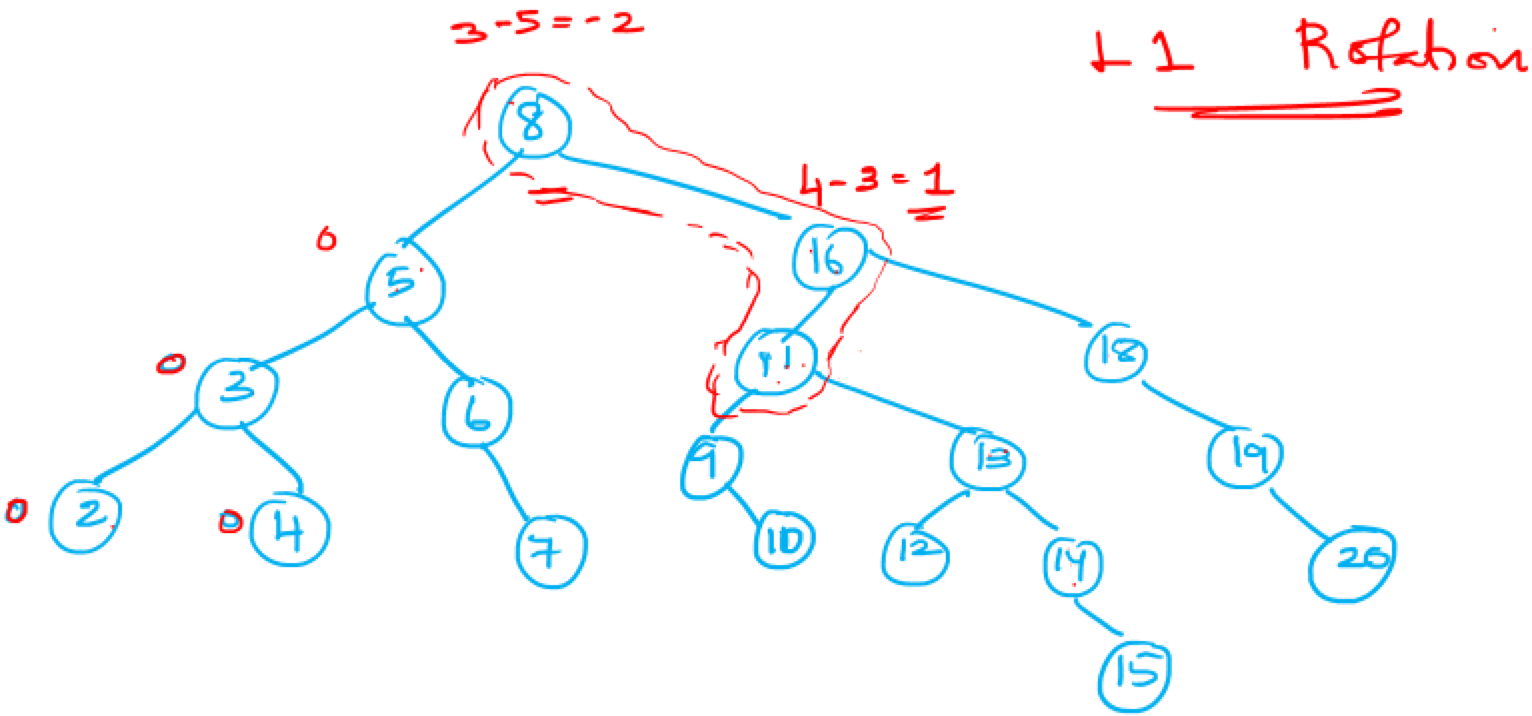
AVL trees-Deletions

L-1 Rotation

Delete 1

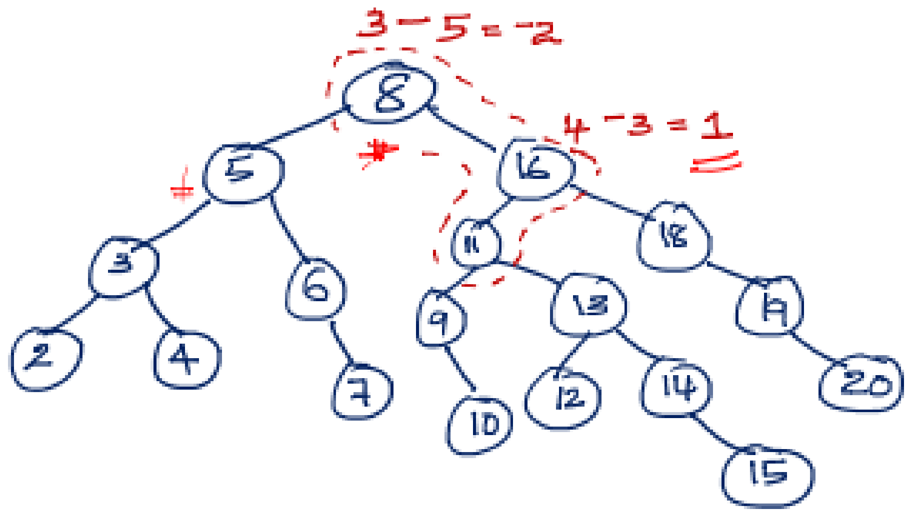


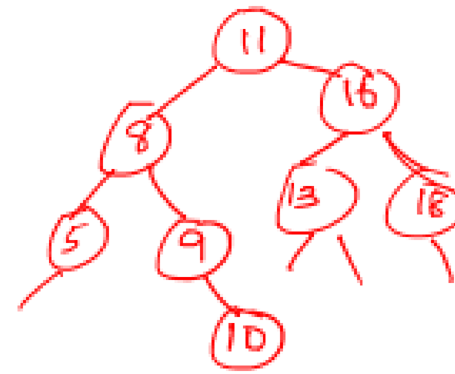
AVL trees-Deletions



AVL trees-Deletions

L-Type L(1)





Now Node 5 is balanced. But node 8 is now unbalanced.

So perform a left rotation. (L 1 Rotation)

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))

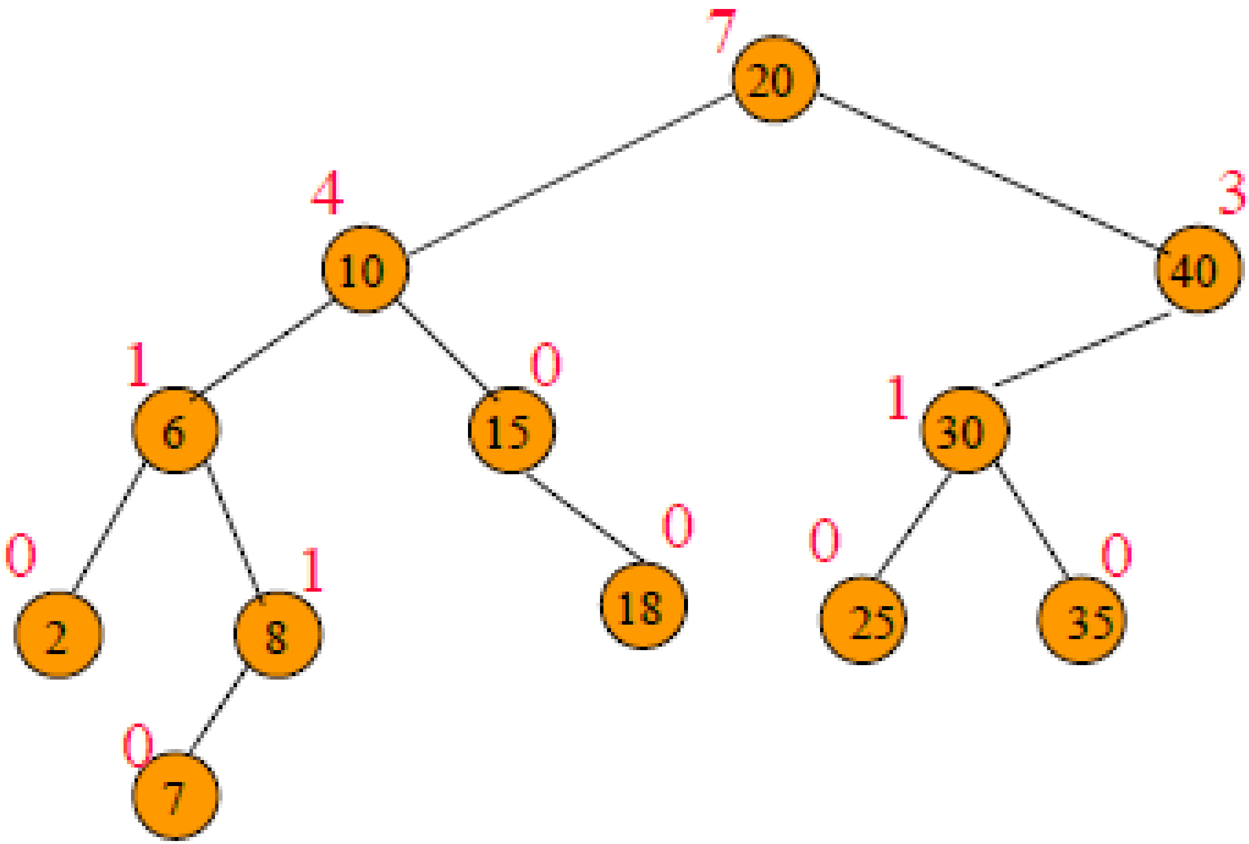
Find k-th smallest element in BST

- The idea is to maintain rank of each node.
- We can keep track of elements in a subtree of any node while building the tree.
- Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Rank

- Rank of an element is its position in inorder traversal (inorder = ascending key order).
- [2,6,7,8,10,15,18,20,25,30,35,40]
- rank(2) = 0
- rank(15) = 5
- rank(20) = 7
- **leftSize(x) = rank(x) with respect to elements in subtree rooted at x**

Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

Find k-th smallest element in BST

- Assume that the root is having N nodes in its left subtree.
  - If K = N + 1, root is K-th node.
  - If K > N, we continue our search in the right subtree for the (K – (N + 1))-th smallest element.
  - Else we will continue our search (recursion) for the Kth smallest element in the left subtree of root.
  - Note that we need the count of elements in left subtree only.
- Time complexity: O(h) where h is height of tree.

Find k-th smallest element in BST

1. *start*
2. ***if K = root.leftElements + 1***
   1. *root node is the K th node.*
   2. *goto stop*
3. ***else if K > root.leftElements***
   1. *K = K - (root.leftElements + 1)*
   2. *root = root.right*
   3. *goto start*
4. ***else***
   1. *root = root.left*
   2. *goto start*
5. *stop*

innovate achieve lead

THANK YOU!!!

BITS Pilani
Hyderabad Campus