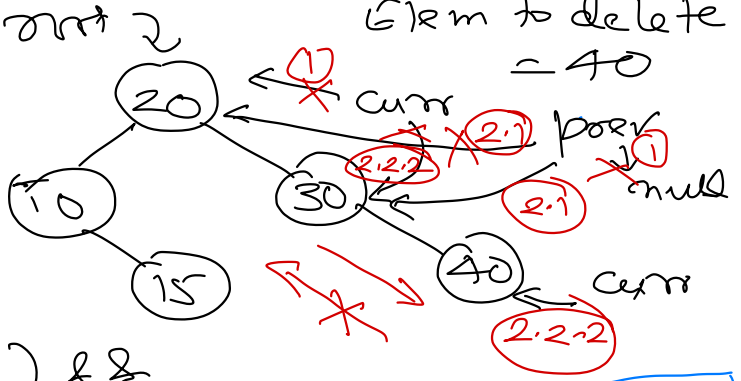
root → null

Empty BST

Delete (elem)

1. Set curr to root and prev to null.
2. while ((curr != null) && (curr.data != elem))
   - 2.1 Set prev to curr
   - 2.2 if (elem < curr.data)
     - 2.2.1 Set curr to curr's lchild.
   - else
     - 2.2.2 Set curr to curr's rchild.



In BST we can only delete a leaf node.

3. if curr == null ~~then STOP~~ loop while

4. if (curr is a leaf node) curr is not a leaf node

5. 5.1 if (curr.data < prev.data)

5.1.1 Set prev's lchild to null

else

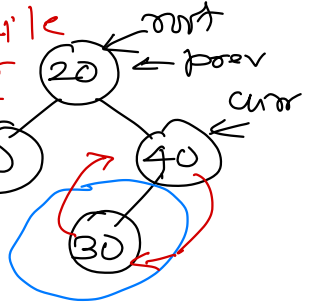
5.1.2 Set prev's rchild to null

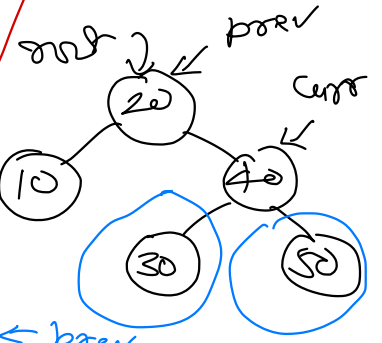
~~5.2 STOP~~

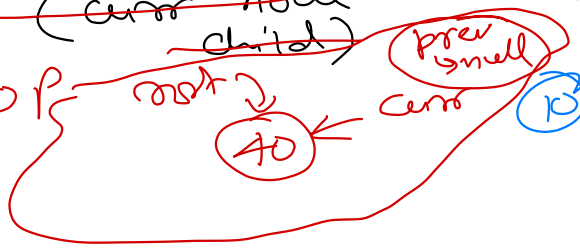
→ if curr == root ~~then~~ root = null

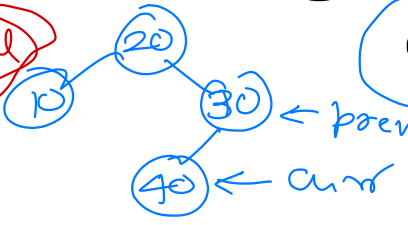
~~6. if (curr node has only one child)~~

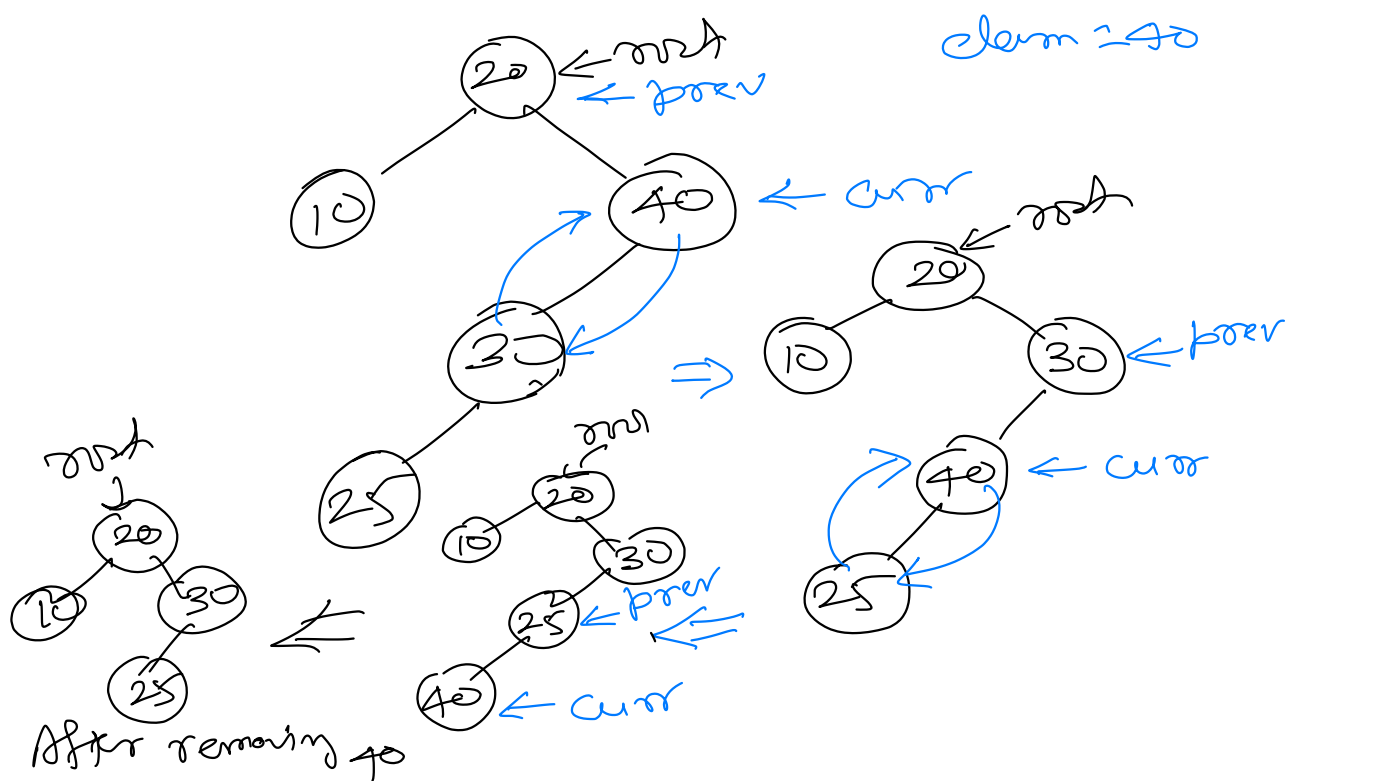
→ STOP





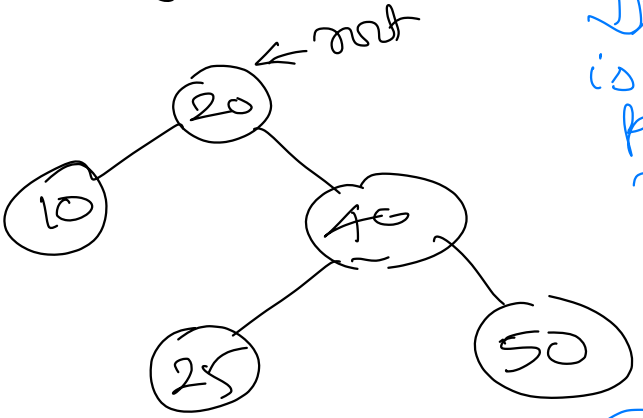




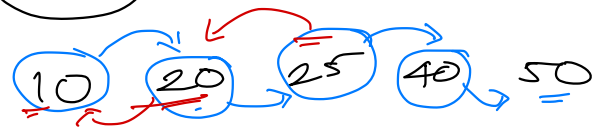

elem = 40
root
prev
curr
20
10
40
30
25
root
prev
curr
20
10
30
25
40
root
20
10
30
25
root
20
10
30
25
prev
curr
40
After removing 40

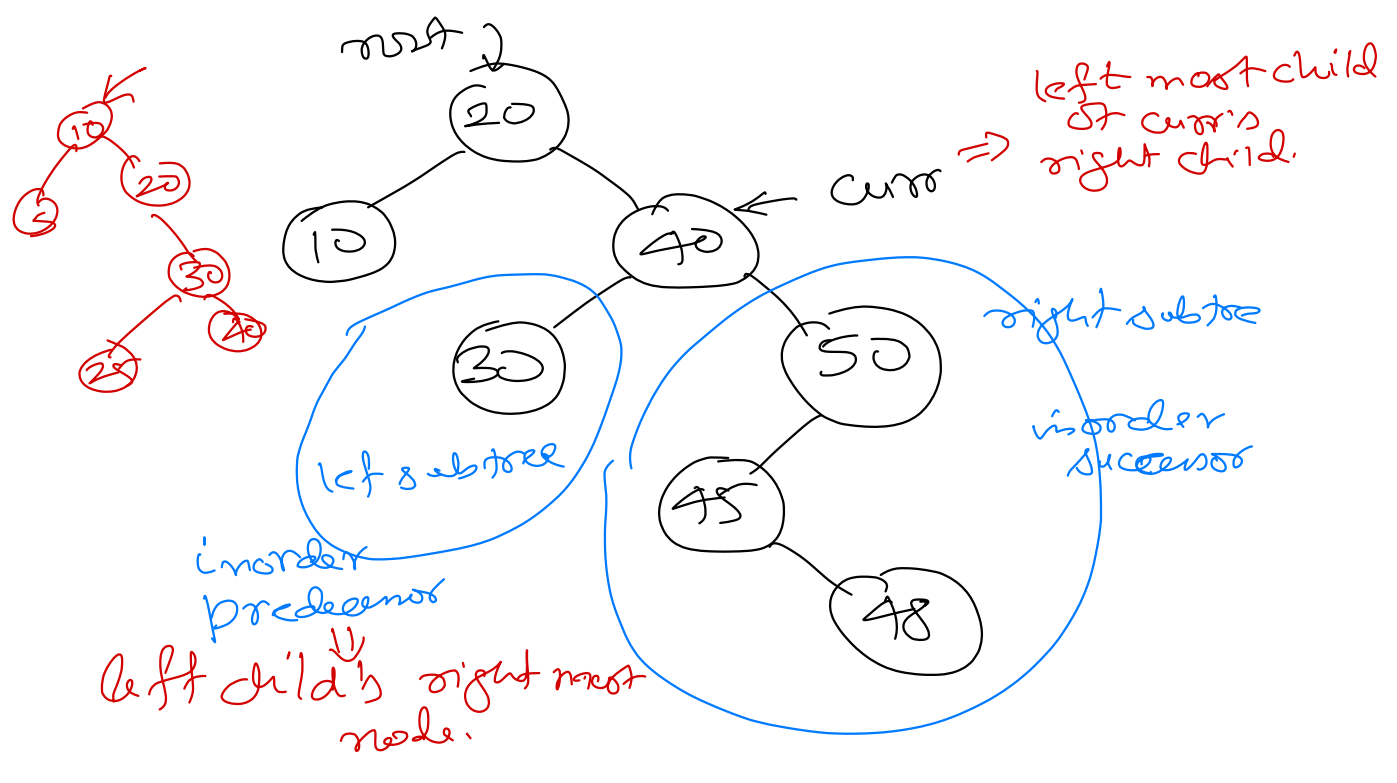
When node to be removed has two child. then
→ We find inorder successor node (is)
→ We swap value of "isNode" & "curr" node &
try to delete in "isNode" node.

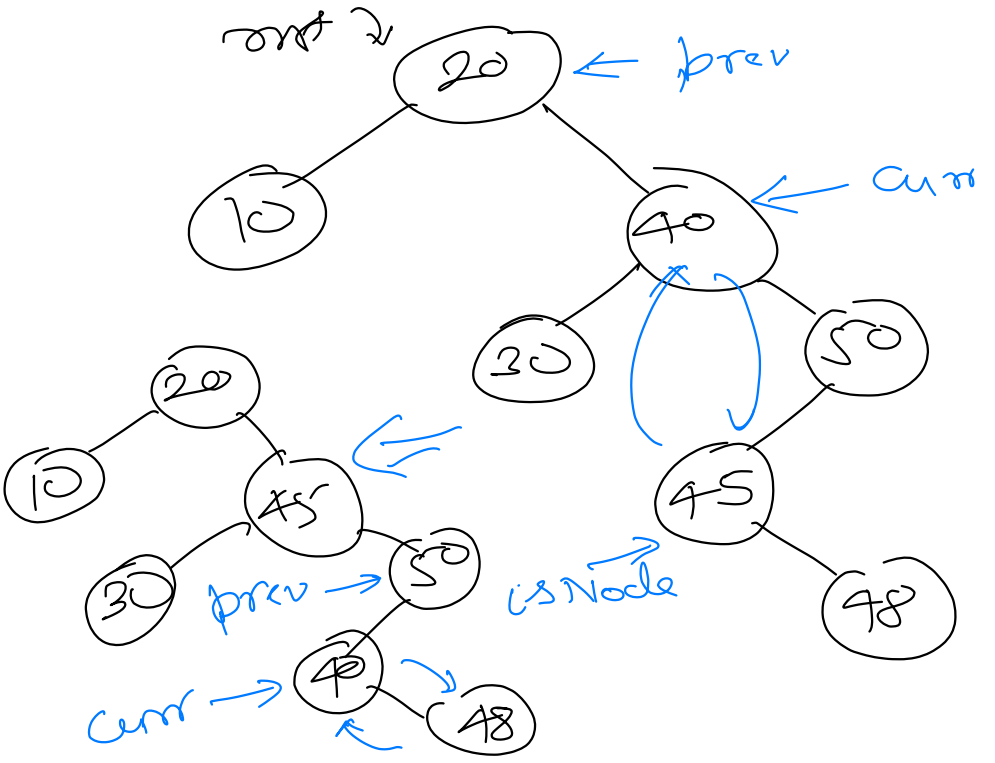
is the node that gets processed in inorder traversal, right after curr node.



Inorder traversal:




root
20
10
40
curr
30
50
45
48
left subtree
right subtree
inorder predecessor
left child's right most node.
inorder successor
left most child of curr's right child.
10
5
20
30
25
40

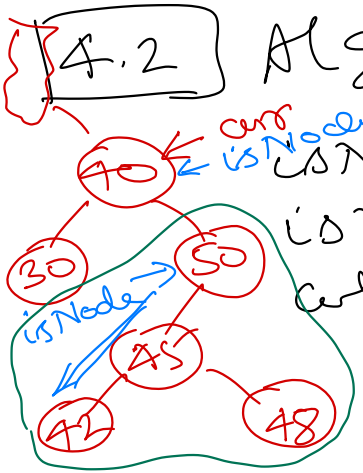

20
prev
10
40
curr
30
50
20
10
45
30
prev
50
40
48
curr
isNode
45
48

Algo for loop to be inserted between step3 & 5.

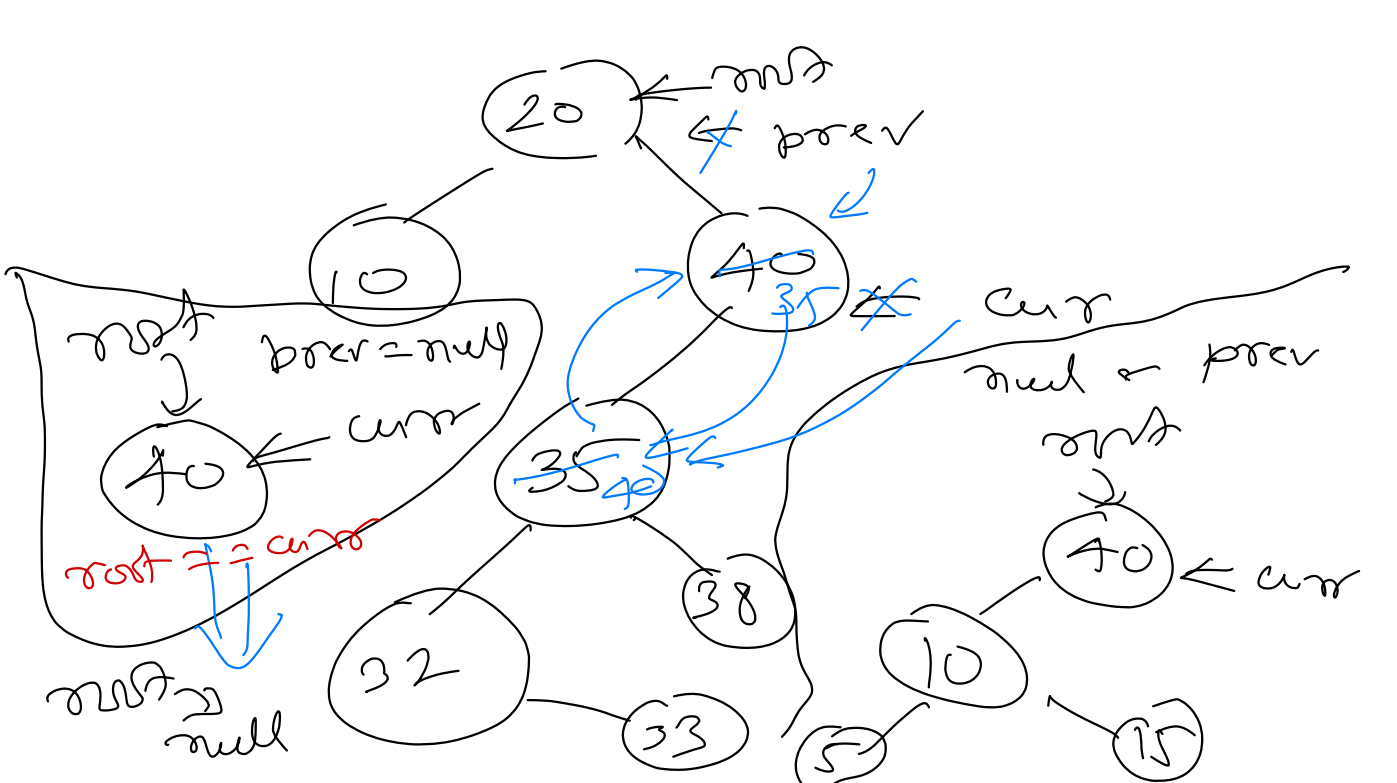
4. while (curr is not a leaf node)
   4.1 if (curr is having one child)
      4.1.1 Swap data of curr and its child.
      4.1.2 Set prev to curr.
      4.1.3 Set curr to its child.
      4.1.4 CONTINUE
   // curr is having two childs.
   4.2 Find inorder successor of curr, isNode.
   4.3 swap curr & isNode's data

4.4. Set curr to isNode.
4.5 Set prev to isNode's parent.
4.6 CONTINUE

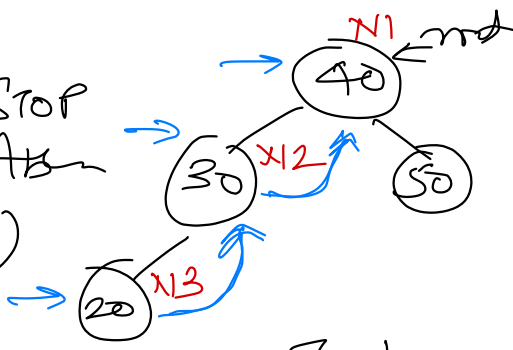
4.2 Algo ↑ find inorder successor of curr

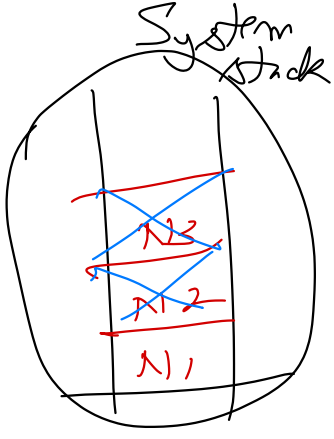



```
isNodeParent = curr
isNode = curr's rchild.
while (isNode has lchild){
    isNodeParent = isNode
    isNode = isNode's lchild.
```


20
root
prev
10
40
35
curr
root
prev=null
40
curr
35
40
root == curr
root
null
32
38
33
curr = prev
root
40
curr
10
5
15

```
inorder (node)
  if node is null then STOP
  if (node.lchild exist) then
      inorder(node.lchild)
  Process node
  if (node.rchild exists) then
      inorder(node.rchild
```

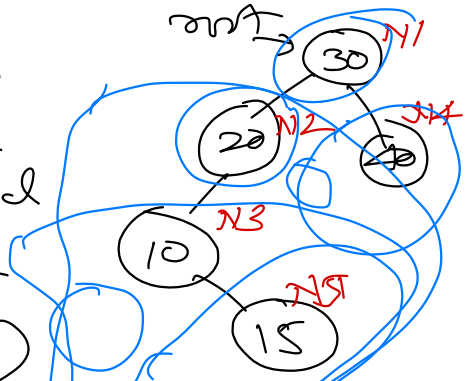


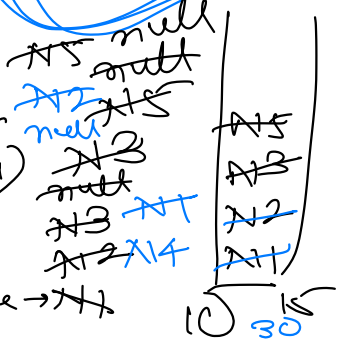


inorderIter(root)

1. while (node is not null)
   - 1.1 Push node on stack
   - 1.2 Move node to its child

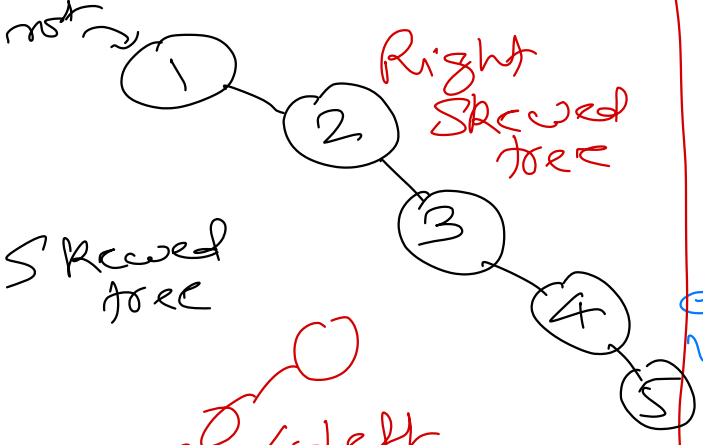
2. while (node is NULL) &&
   (stack is not empty)
   - 2.1 Pop node from stack
   - 2.2 Proces node.
   - 2.3 Set node to its rchild.
   - 2.4 while (node is not null)
     - 2.4.1 Push node on stack
     - 2.4.2 Move node to its child.





Problem with BST

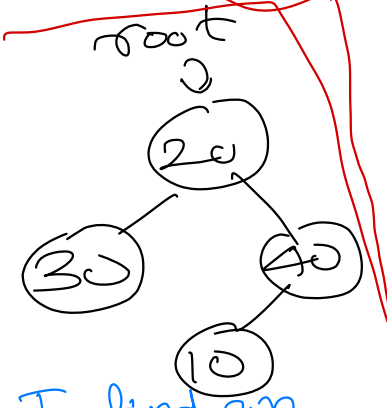
insert into BST
1 2 3 4 5



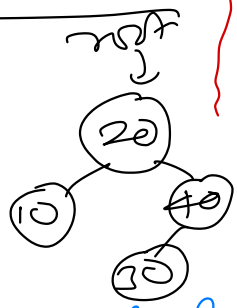
Right Skewed tree

Skewed tree

Adv of BST



To find an element, we visit every node in tree.



To find an element we need not visit every node

Balanced Trees → AVL Tree

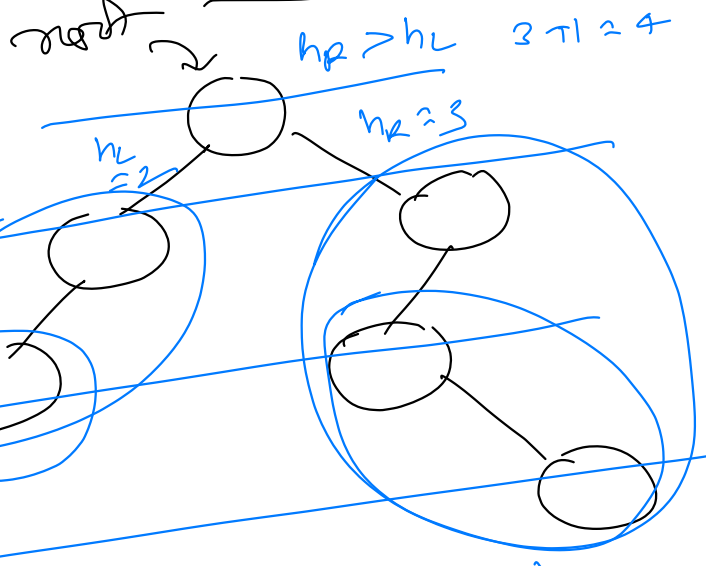
Height Balanced BST

Balanced Factor

Height of a tree

Every node in AVL Tree will have $BF \leq 1$

$BF = |h_L - h_R|$



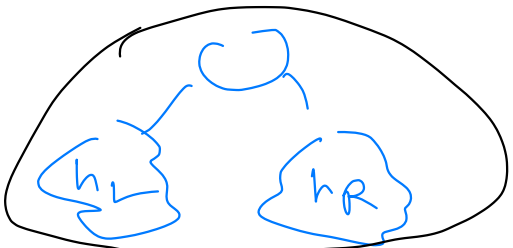
Find height of a tree

Height of a Tree (node)

1. If node is empty then return 0
2. if node is leaf node then return 1
3. h_L = height of a Tree (node.lchild)
4. h_R = height of a Tree (node.rchild)
5. return 1 + max(h_L, h_R)

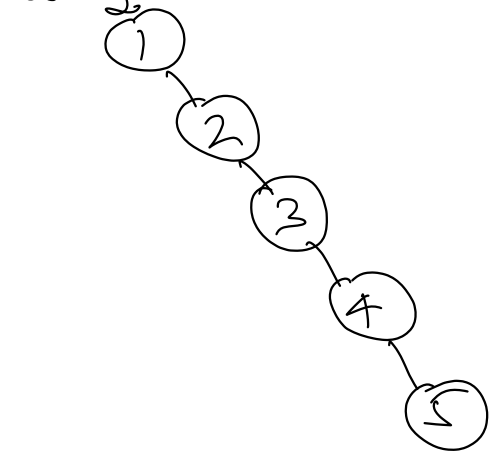
root == null
height = 0

root != null
height = 1



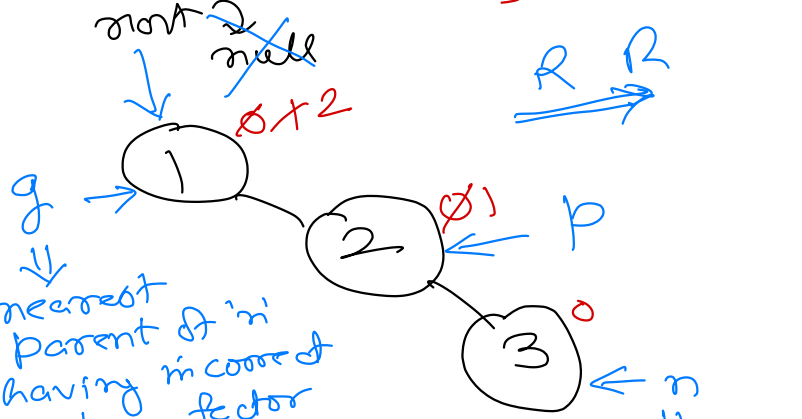
insert → 1 2 3 4 5

BST



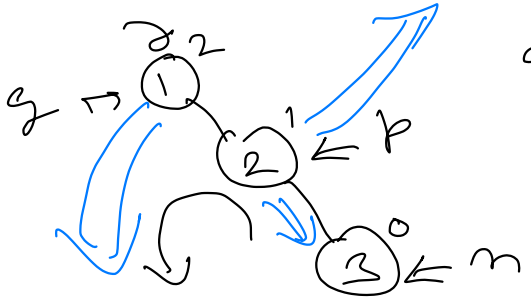
AVL Tree

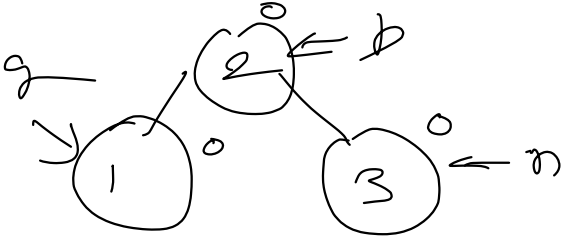
balance factor

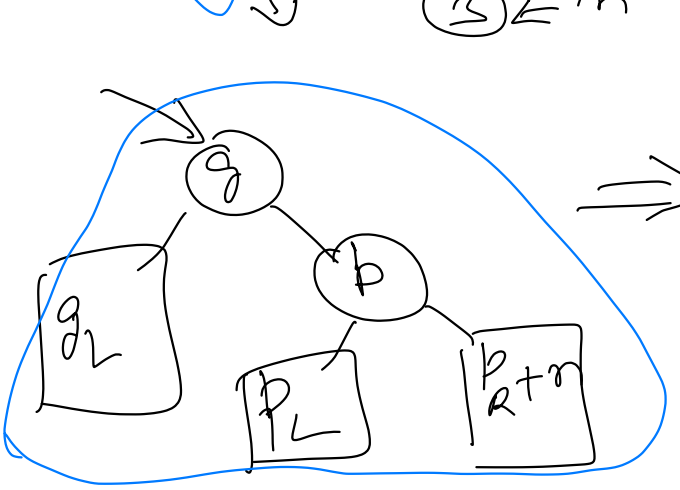


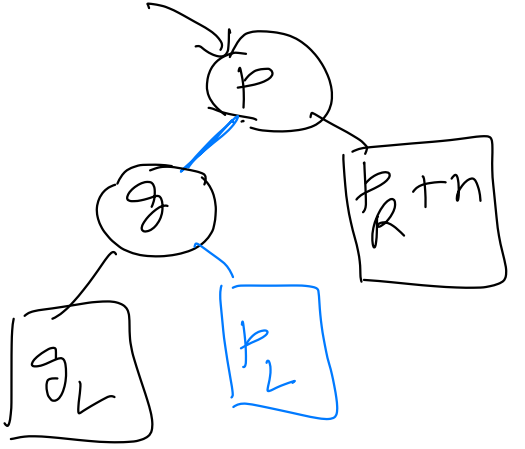
when AVL Tree is not balanced then we perform rotation.

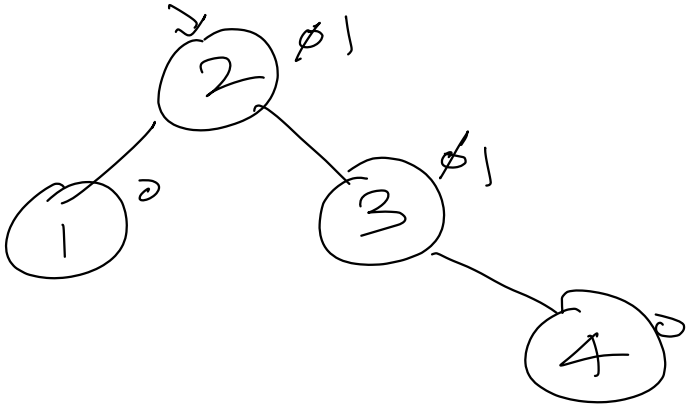
Left Rotation
Right Rotation

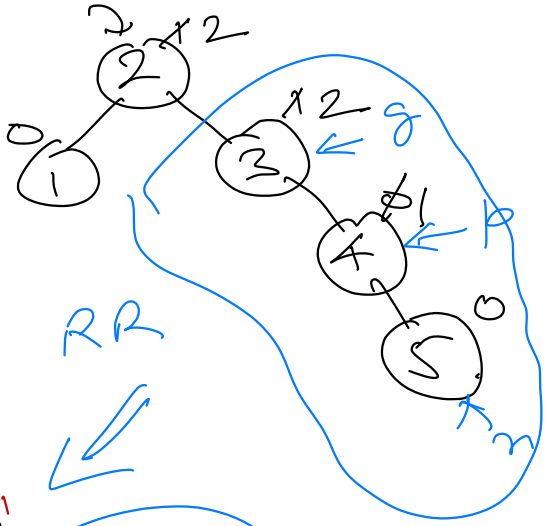

g
1
2
2
p
3
0
n

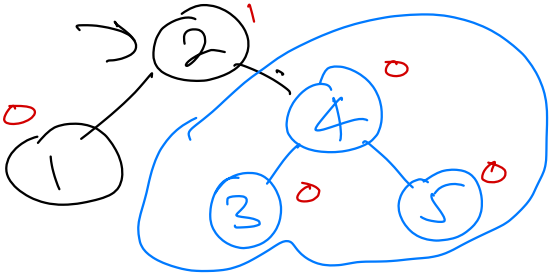

g
2
0
p
1
0
3
0
n


g
p
g_L
p_L
p_{R+n}


p
g
p_{R+n}
g_L
p_L


2
1
3
4
0
1
0


2
1
2
3
2
4
1
5
0
g
p
n
RR


2
1
0
4
0
3
0
5
0