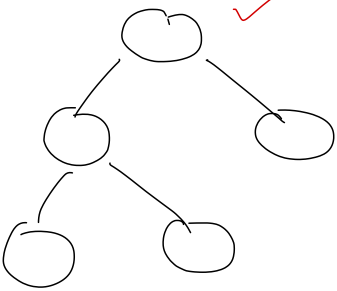
→ Complete Binary Tree

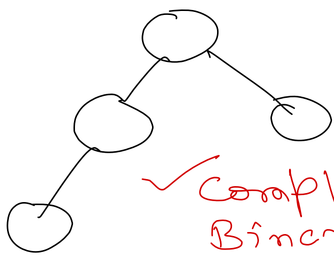
✓



✓Complete Binary Tree

✓Full Binary Tree

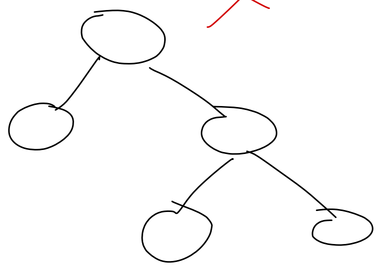
✓



✓Complete Binary Tree

✗ Full Binary Tree

✗



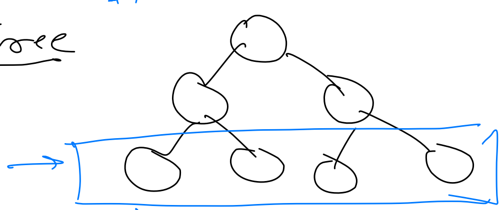
✗ Complete Binary Tree

✓ Full Binary Tree

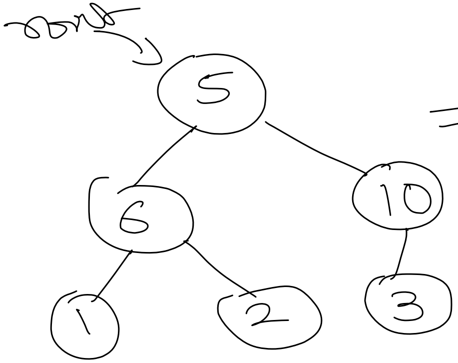
→ Full Binary Tree

→ Perfect Binary Tree

→ Complete + Full



All leaf nodes at same level.



$\Rightarrow$

1	2	3	4	5	6
5	6	10	1	2	3

i^{th} node

left child $2*i$

right child $2*i+1$

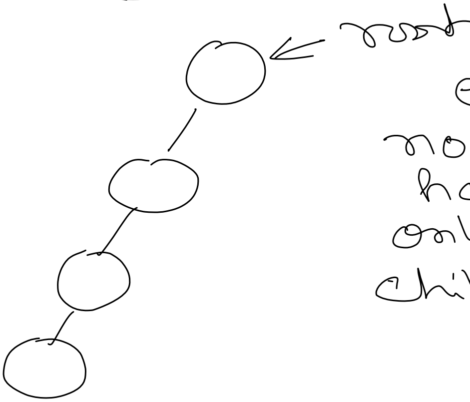
root → 1st element

2nd element   3rd element

4th ele   5th elem   6th elem

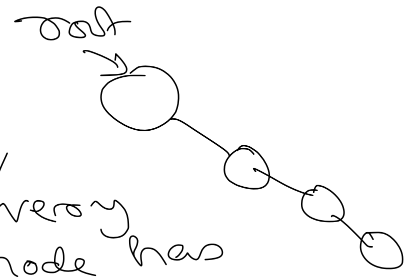
→ <u>Left Skewed</u>

every node has only left child.



→ <u>Right Skewed</u>

every node has only right child.



→ Skewed BST results in inefficient Search.

→ Balance BST

- In Order Traversal of BST
  ⇒ we get element in sorted order.

→ AVL Tree : Efficient Search

→ Red Black Tree : Efficient Insert & Delete

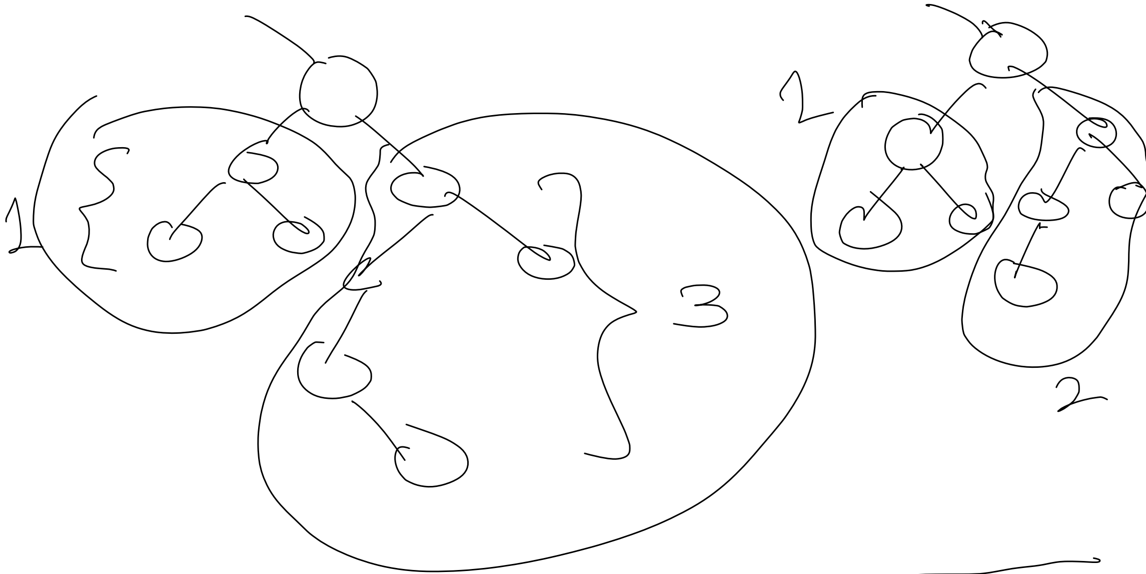
Uses Balance factor to determine if tree is unbalanced & if unbalanced uses Rotation to balance tree.

for a node
$BF = |h_L - h_R|$

h_L = height of left subtree
h_R = height of right subtree

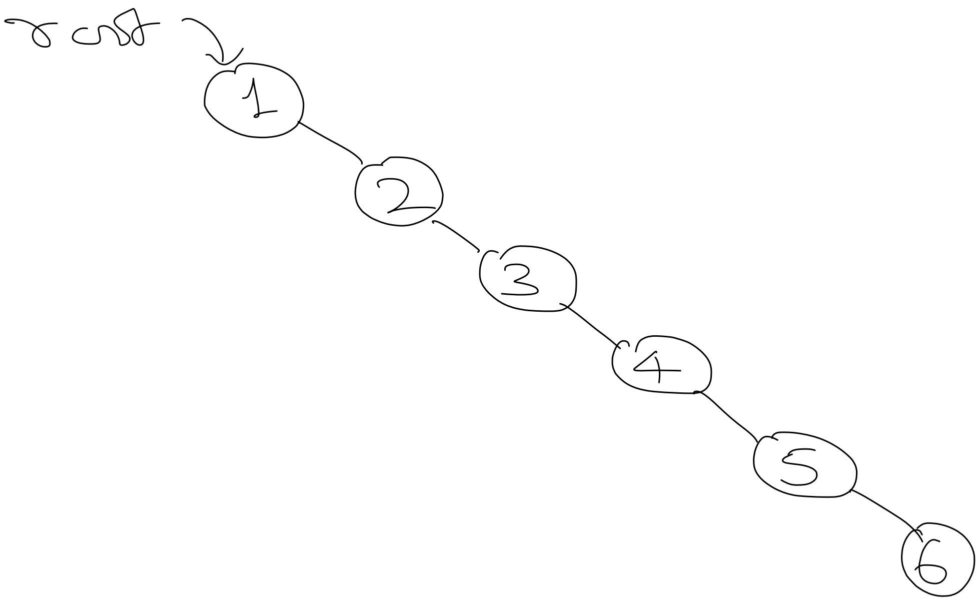
if BF > 1 then tree is unbalanced.

Uses colors to determine unbalanced tree.
if tree is unbalanced it first tries to re color the tree and then if required performs rotation.



Add : 1 2 3 4 5 6

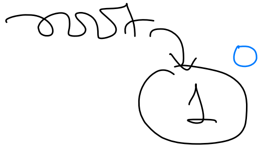
BST



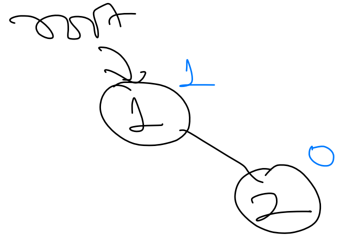
AVL

root
↓
empty

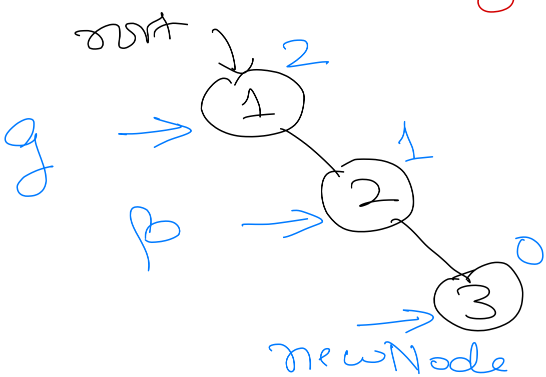
Insert(1)



Insert(2)



Insert(3)



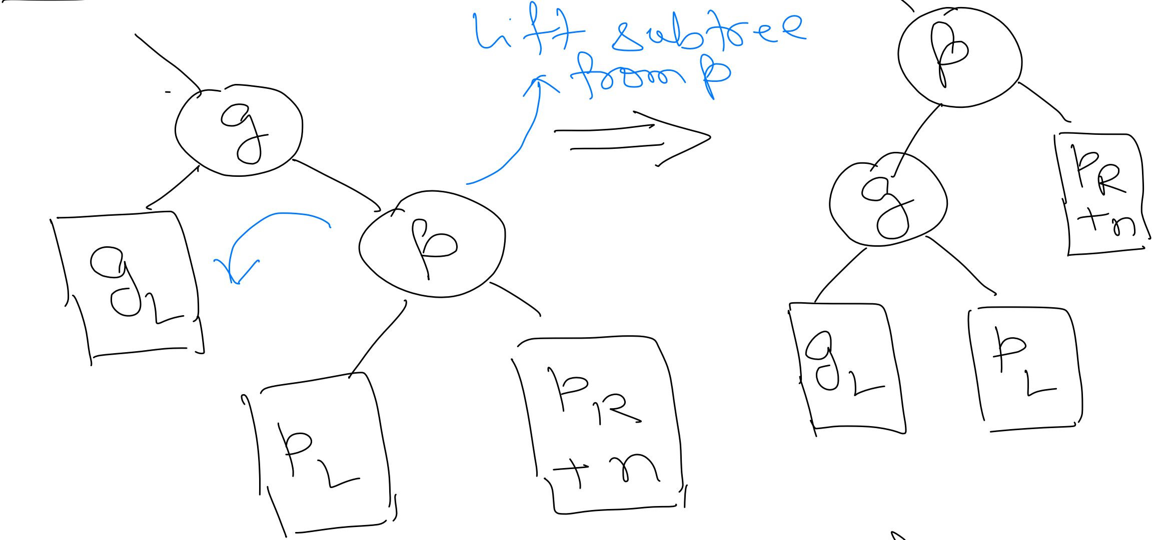
g ⇒ grand parent
⇓
nearest parent of new node with incorrect balance factor.

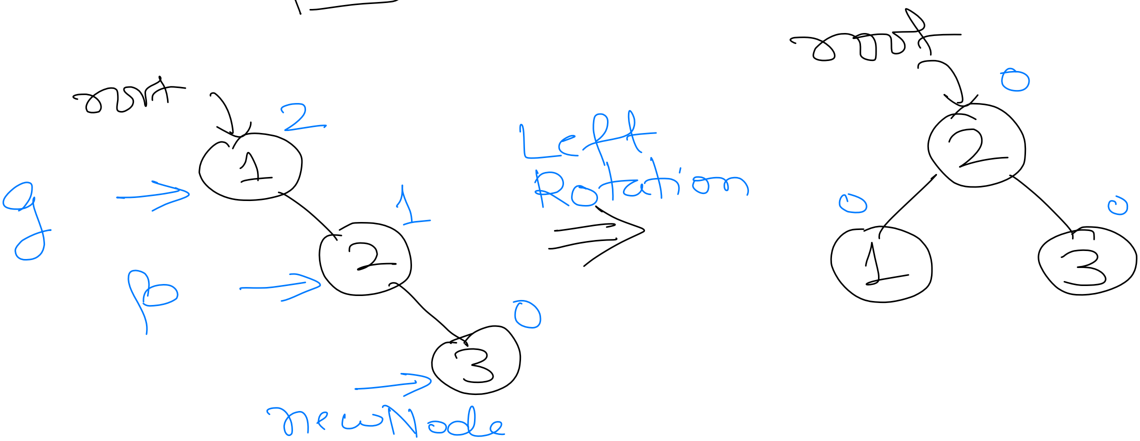
p ⇒ parent
↓
child of grand parent, in which subtree newNode was added.

R R
⇓
Left Rotation

To find type of rotation, we take two steps from g towards newNode.
We may reach newNode OR we may not, doesn't matter.

Left Rotation



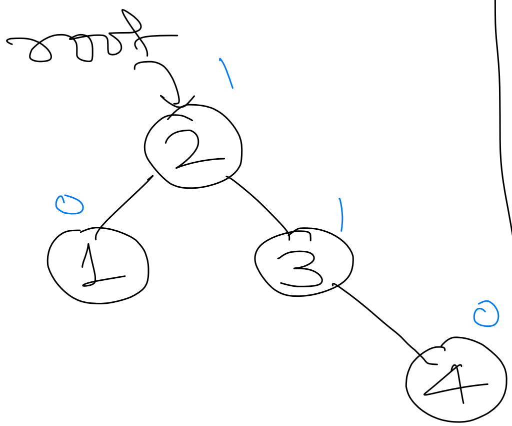


RR

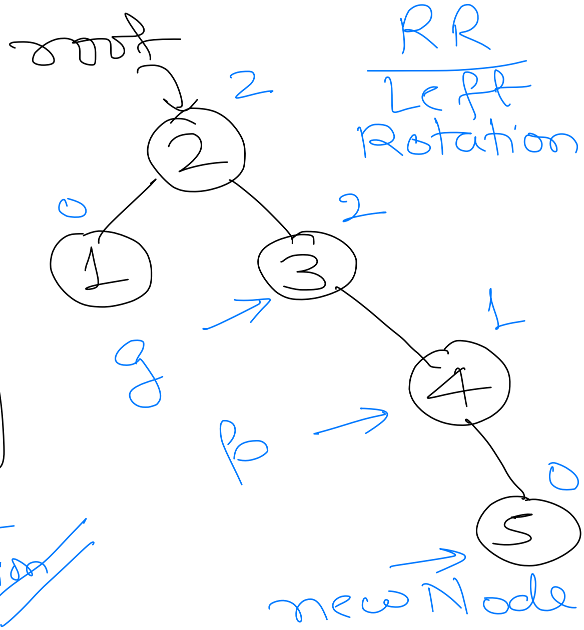
⇓

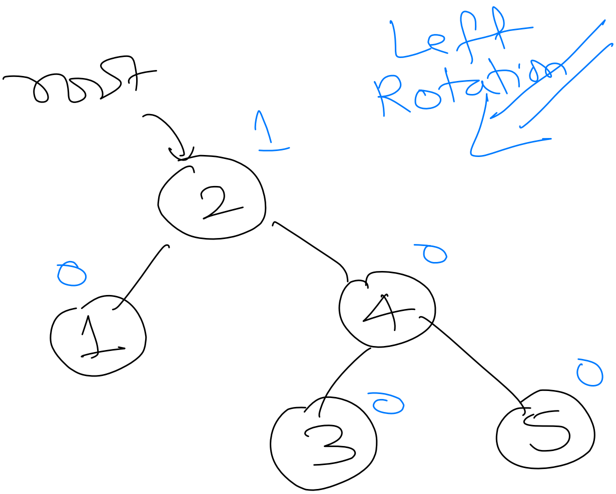
Left Rotation

Insert (4)

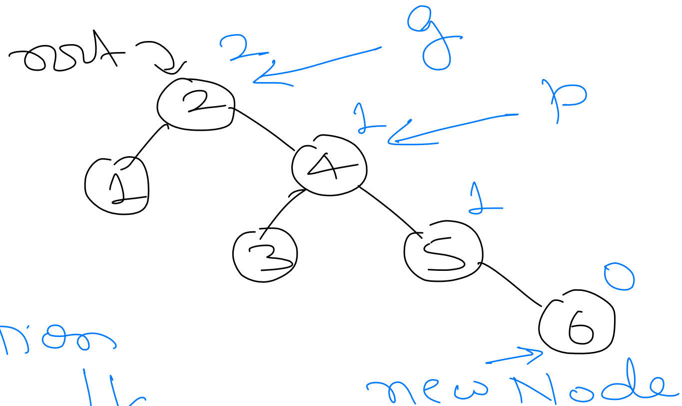


Insert (5)



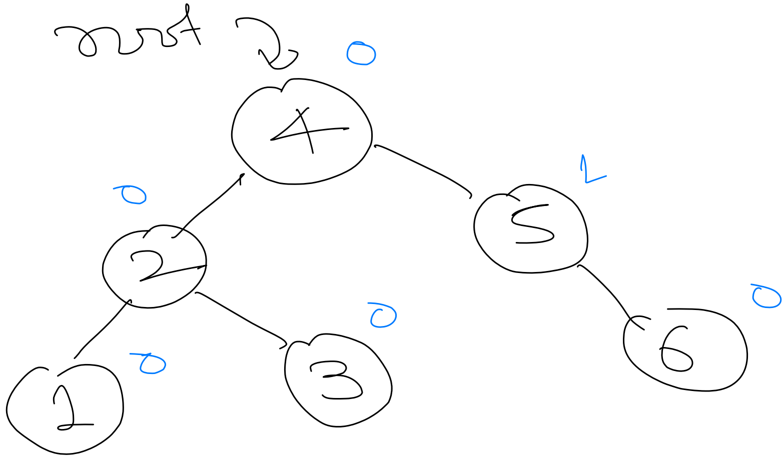


Insert (6)

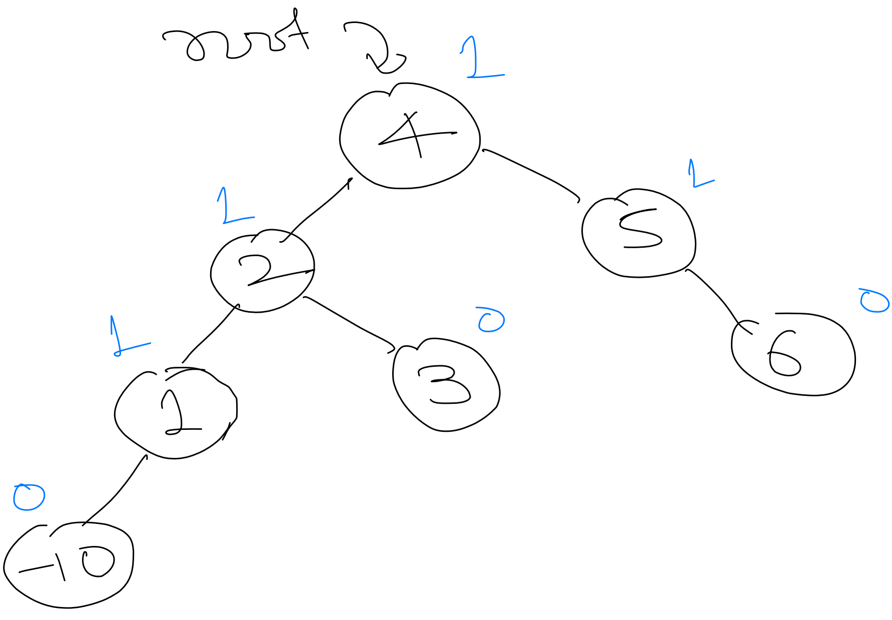


RR

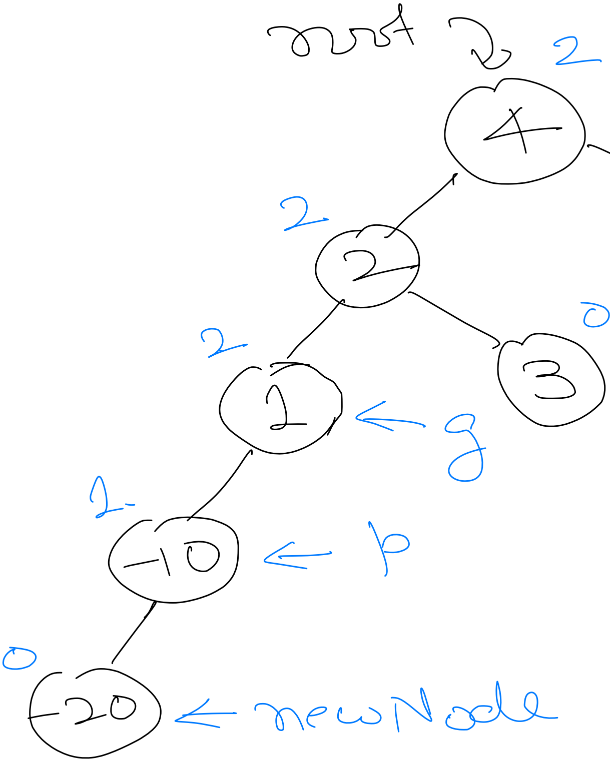
Left Rotation



Insert (-10)



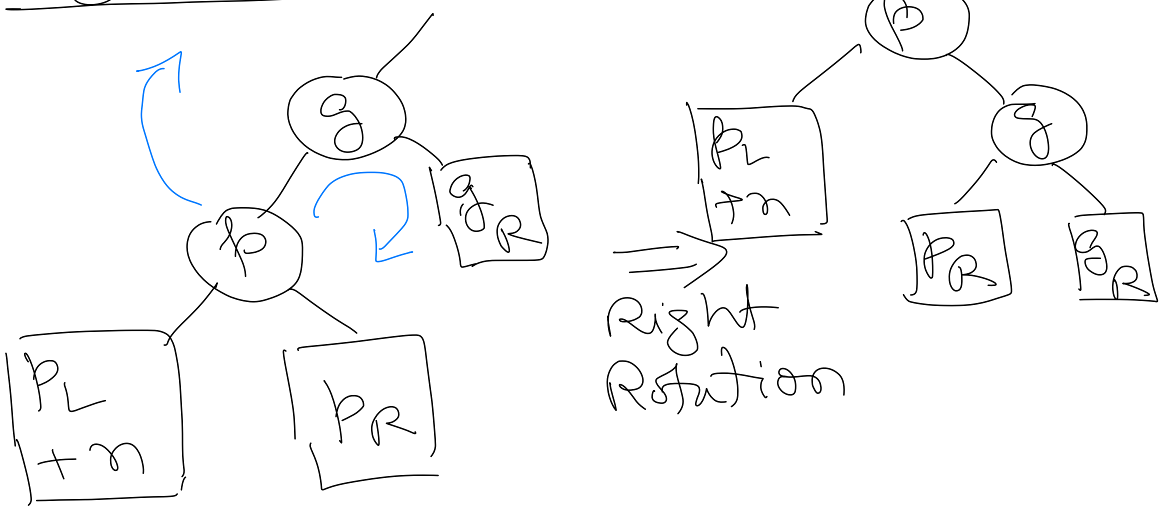
Insert (-20)

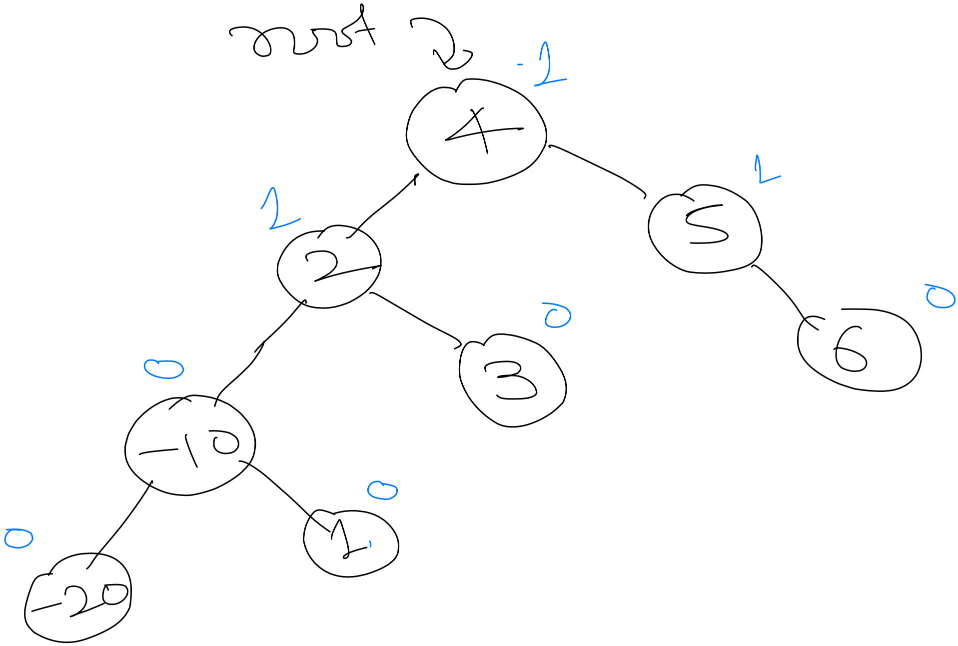


LL

Right Rotation

Right Rotation

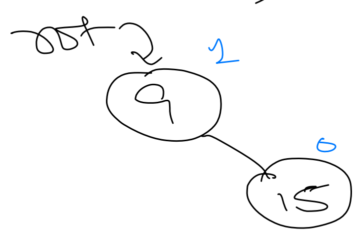




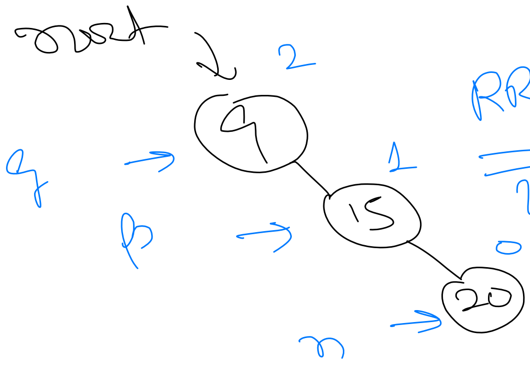
9 15 20 8 7 13 10

Insert (9)

Insert (15)

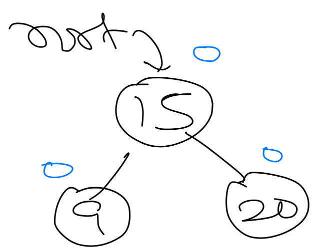


Insert (20)

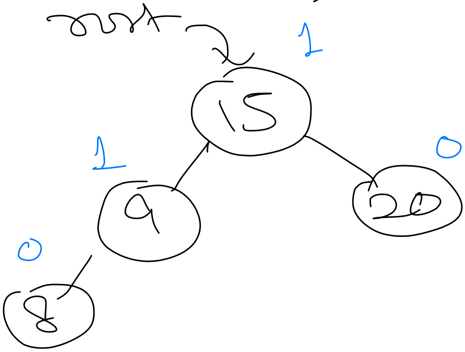


RR

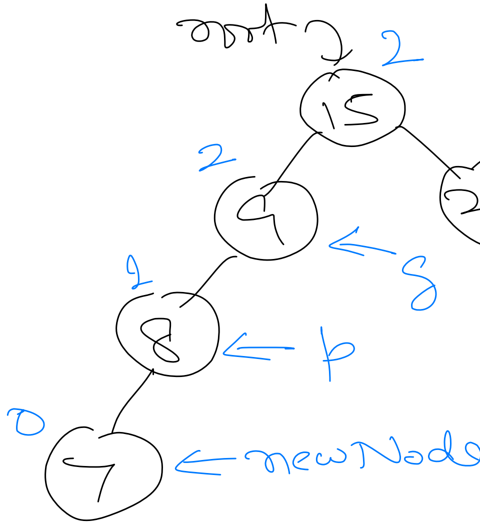
Left Rotation



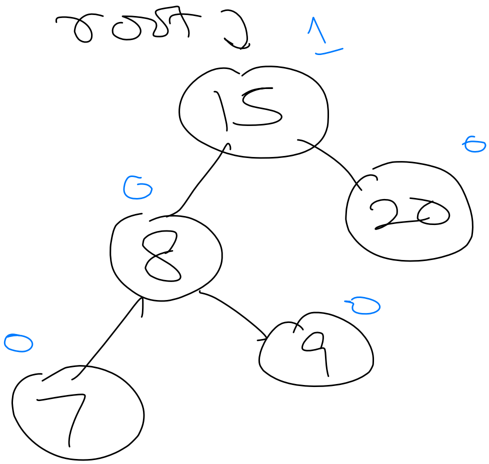
Insert (8)



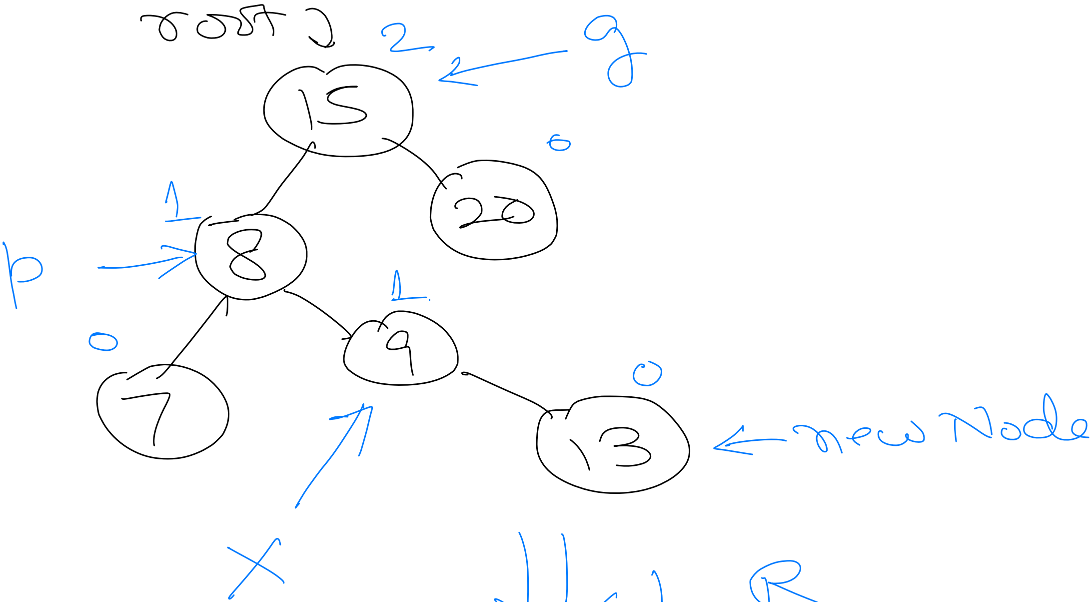
Insert (7)



Right Rotation ⇓ LL



Insert (13)

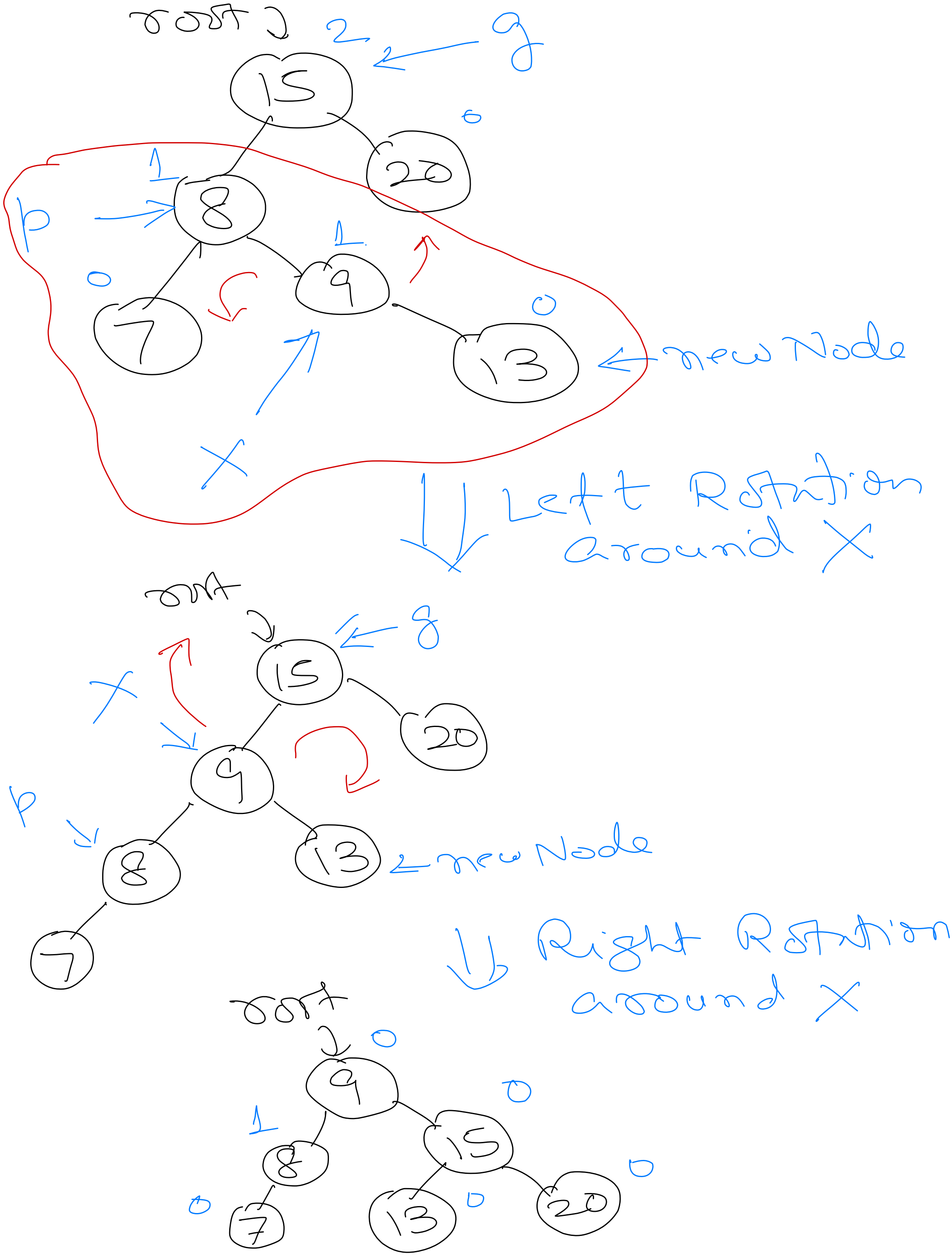


LR

① Left Rotation
② Right Rotation

around node X

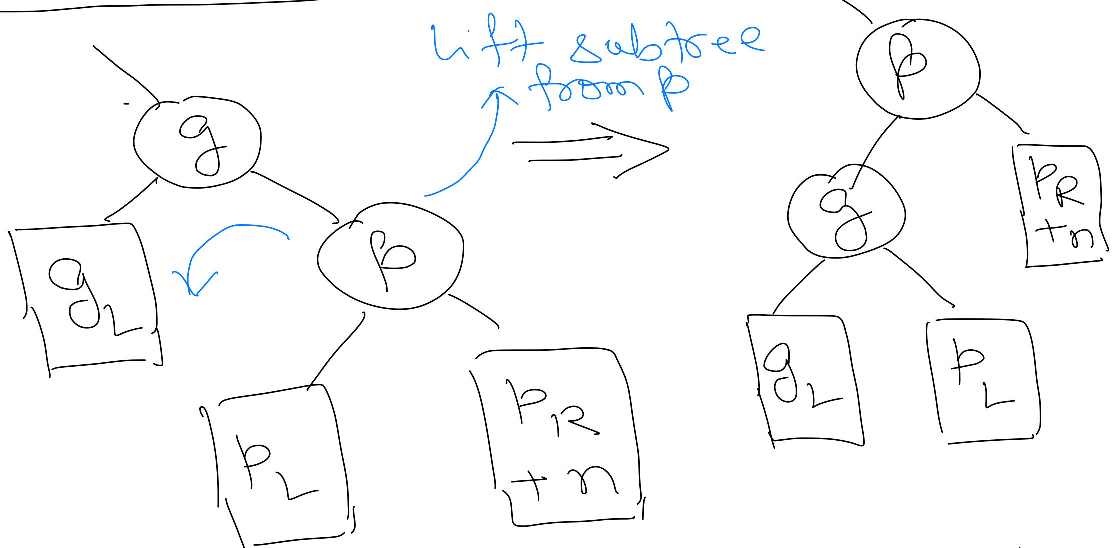
reached from g by taking two steps LR


root
g
p
x
new Node
Left Rotation around X
root
g
x
p
new Node
Right Rotation around X
root

RL ⇒ Right Rotation } around X
Left Rotation }

X ⇒ take two steps from g.
⇓ ?
RL

Left Rotation : Algo



→ Swap values of g & p nodes.
→ Set child nodes of g & p.

Red-Black Tree

→ Every node will be either Red or Black.

→ Root node must be black.

→ Parent and child nodes both can't be red.

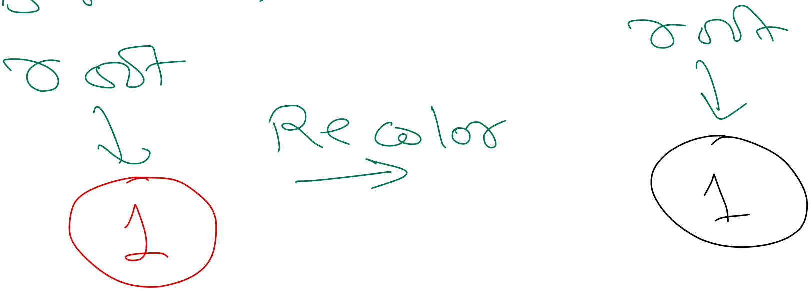
→ Both childrens of a red node must be black.

→ All child of leaf/empty nodes are black.

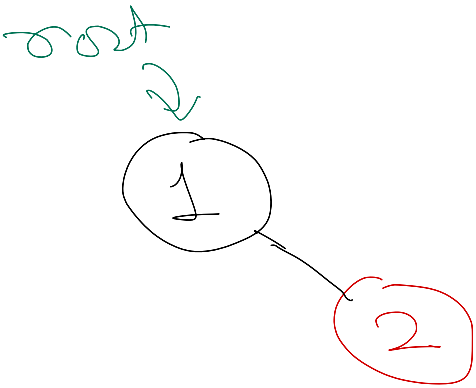
→ Every path from a node to its leaf node has same number of black nodes.

→ New node inserted in red-black tree is always red.

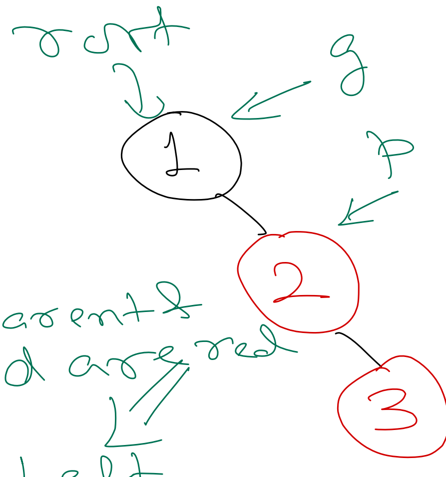
Insert (1)



Insert (2)

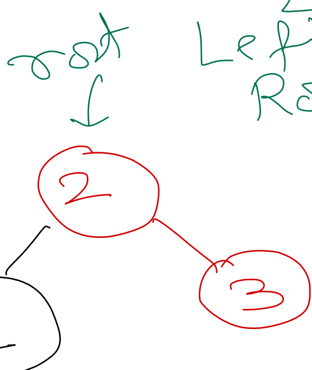


Insert (3)



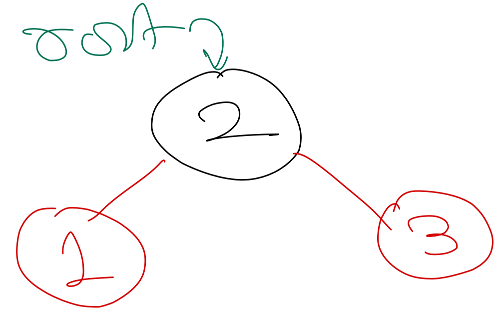
Parent & Child are red

Left Rotation

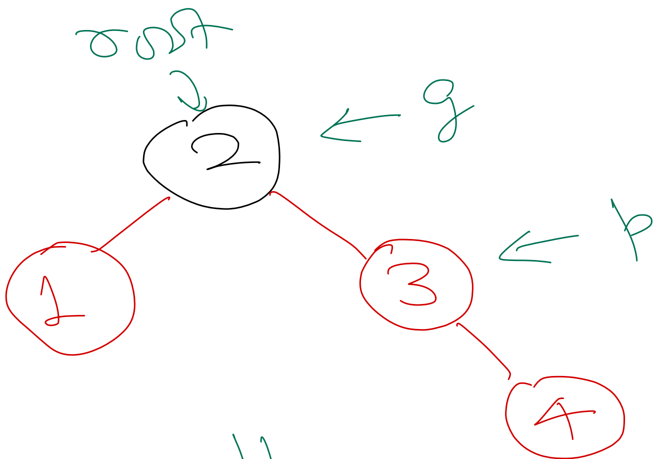


Swap color of parent and grand parent

⇓



Insert (4)



⇓

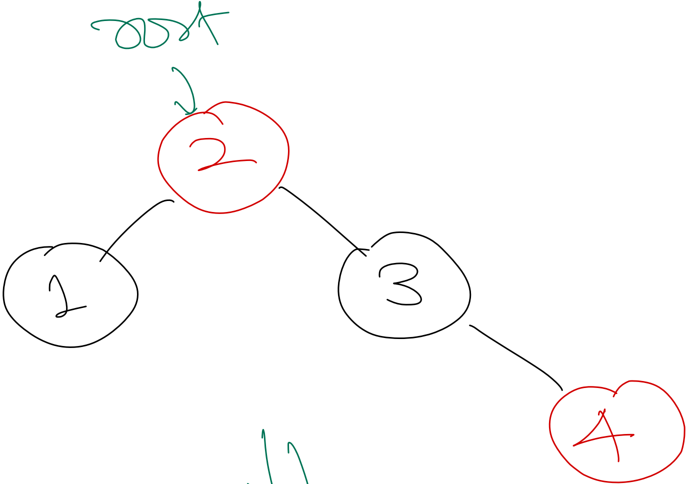
Parent(3) & child(4) are red
But uncle(1) & child is red

⇓

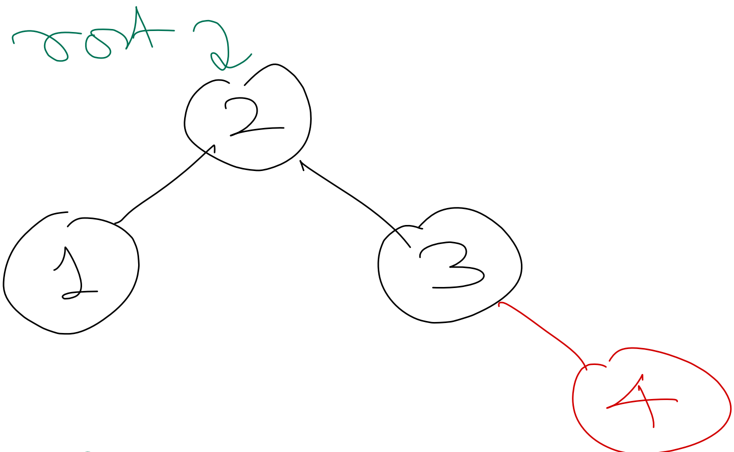
Recoloring

⇓

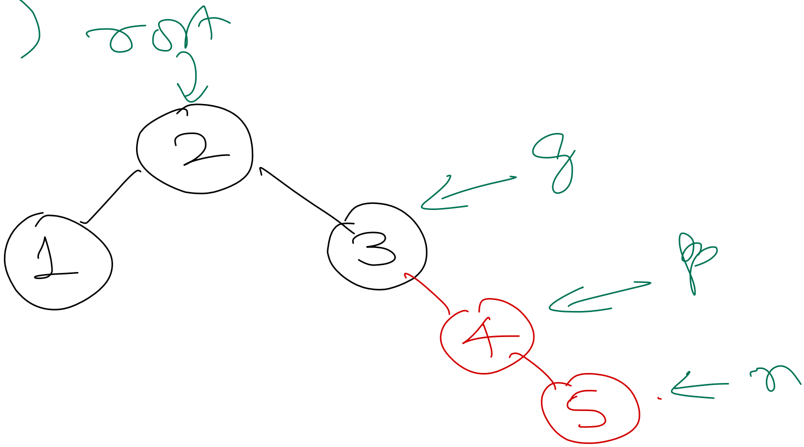
Push blackness down from grand parent



Make root as black

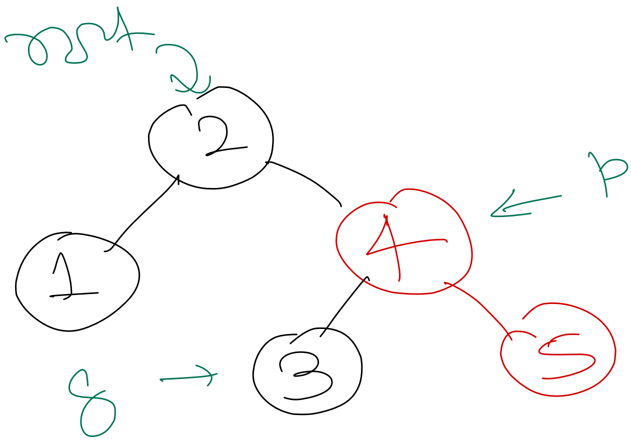


Insert(5)



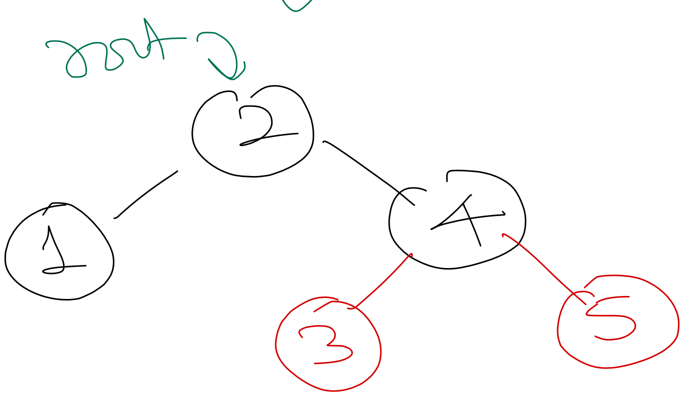
Parent (4) & child (5) are red.
Uncle of child (5) do not exist ⇒ black.

⇓ Left rotation

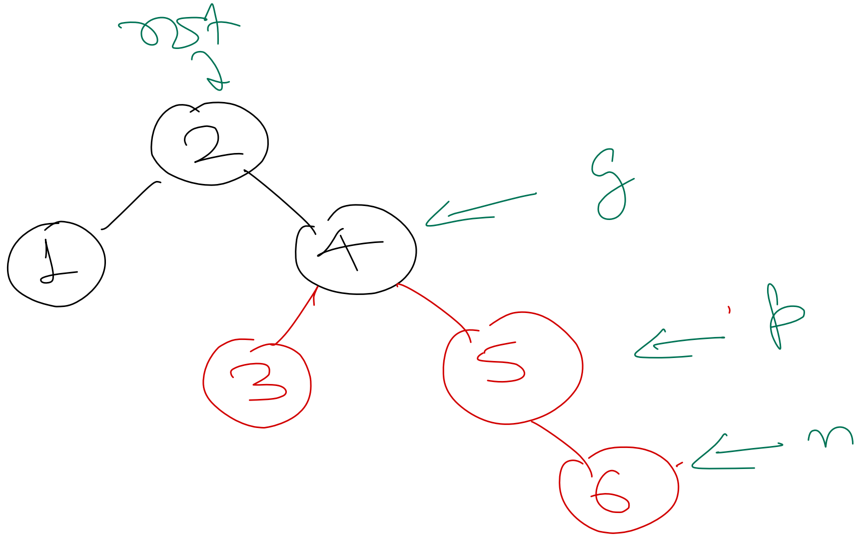


Swap color of p & g

⇓

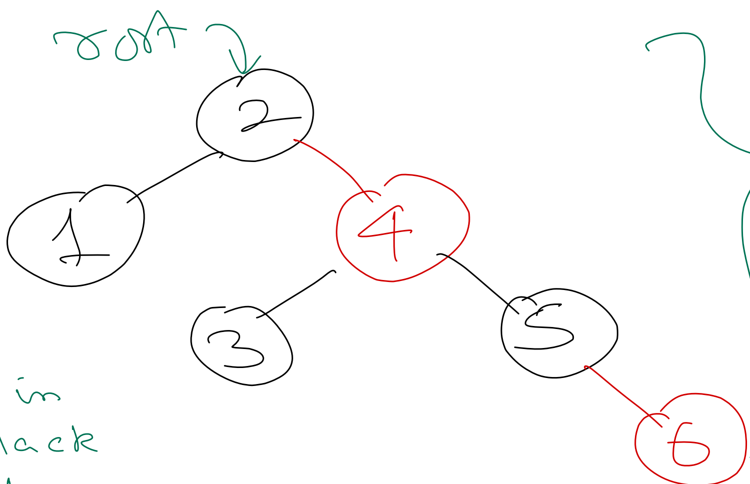


Insert (6)



Parent (5) & child (6) are red.
Uncle (3) of child (6) is red.

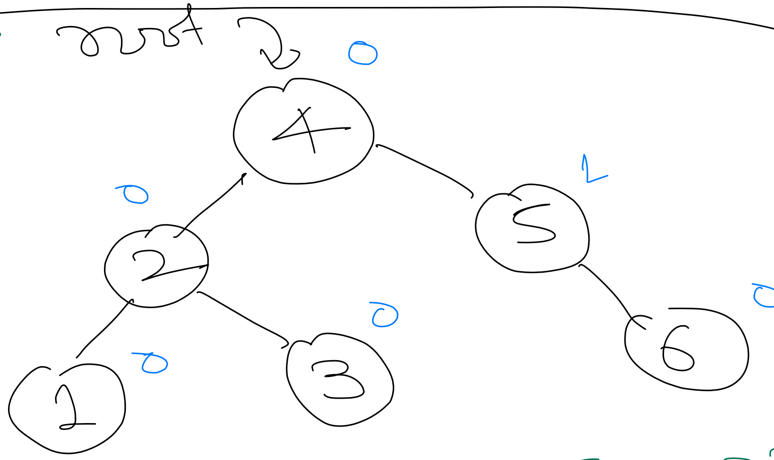
⇓ Recolor
Push blackness down from grandparent



Red-black Tree after inserting 1 2 3 4 5 6

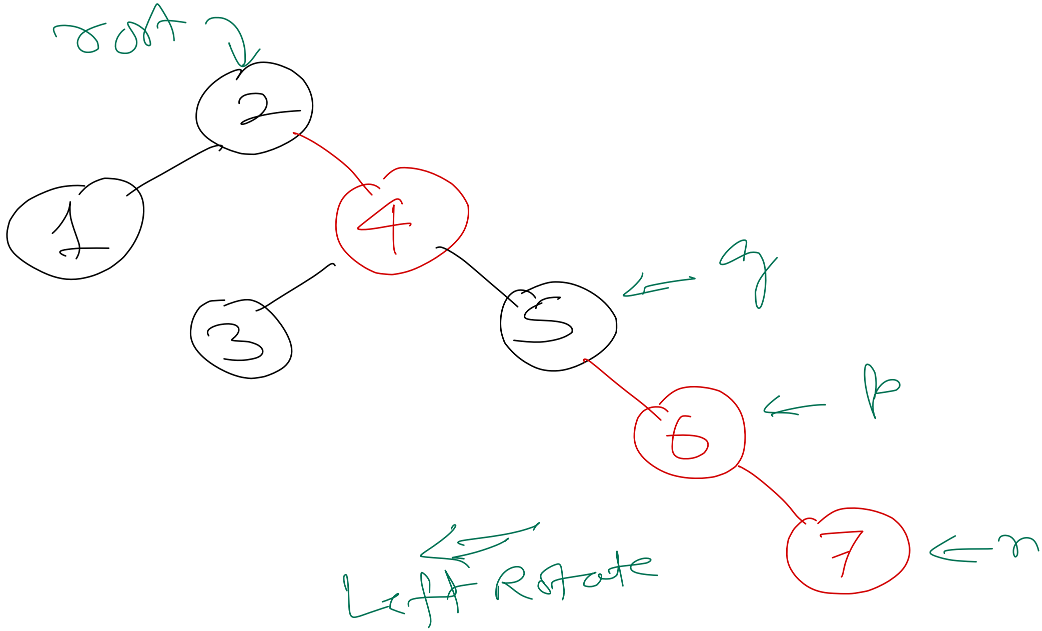
Levels in Red-black tree = 4

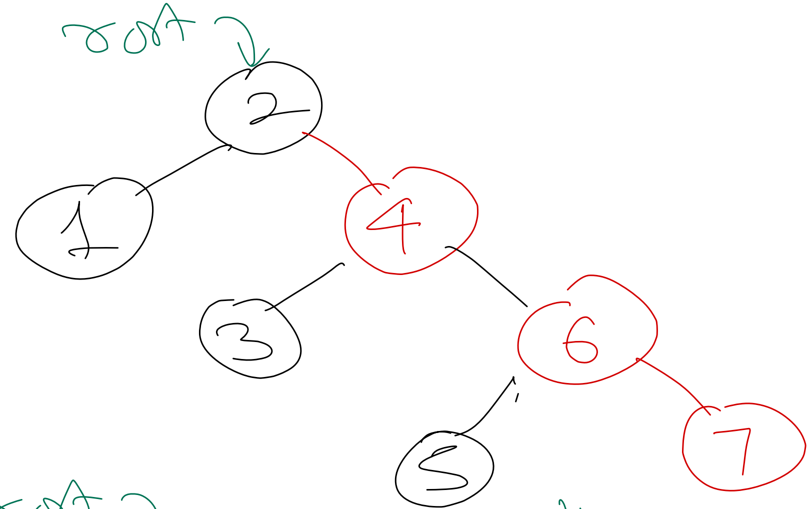
AVL Tree after inserting 1 2 3 4 5 6



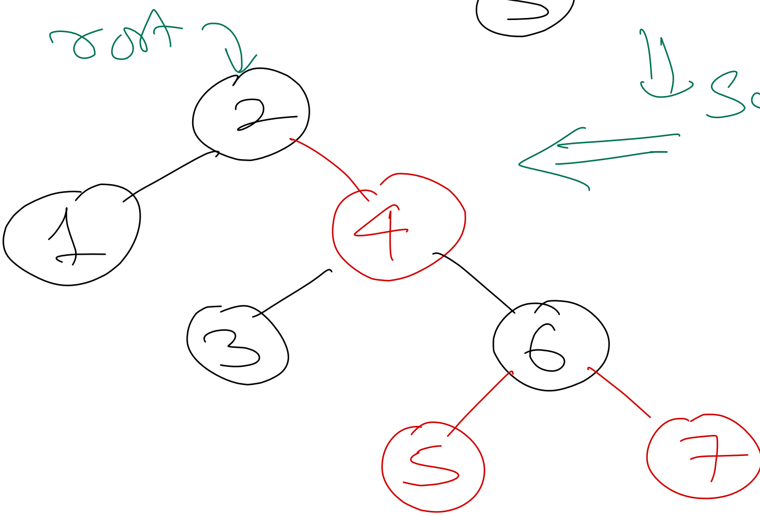
Levels in AVL Tree = 3

Insert (7)

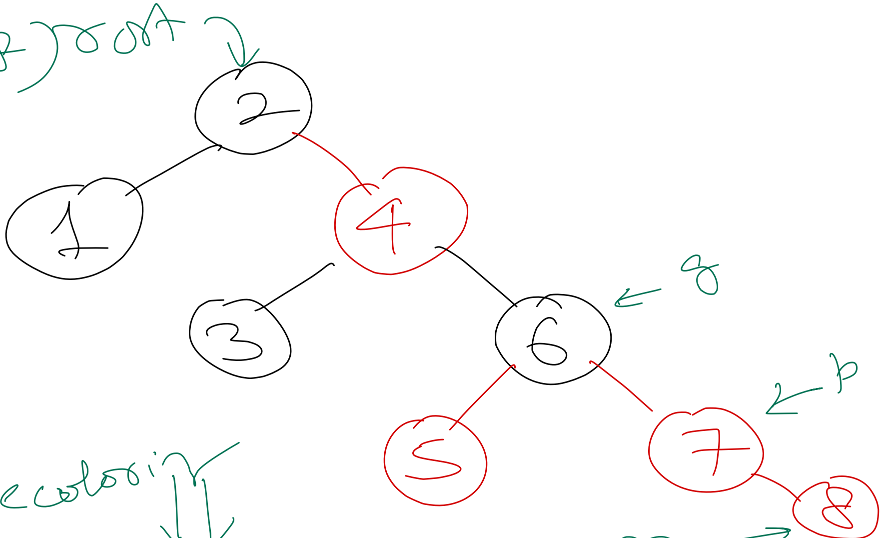




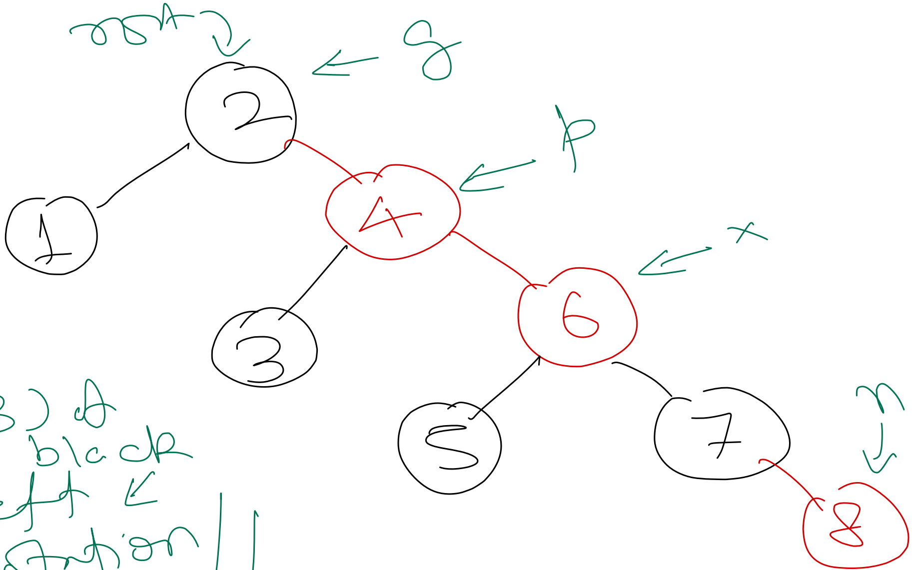
Swap p & g color



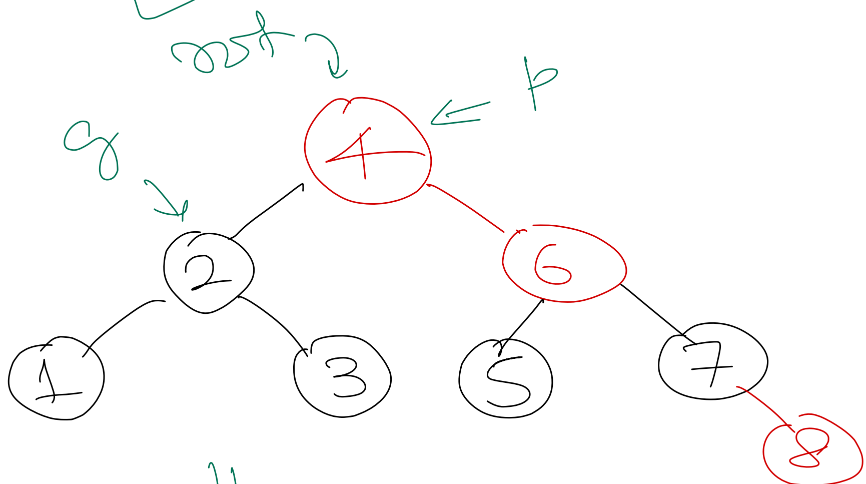
Insert(8)



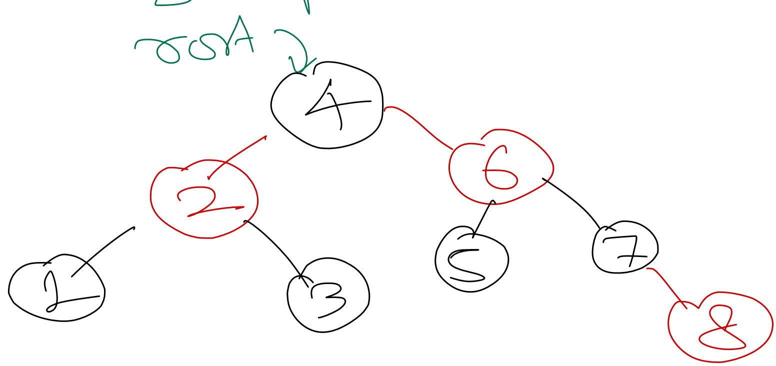
Recoloring



Uncle (3) of X(6) is black
Left Rotation



Swap color of p & g



M-way Search Tree

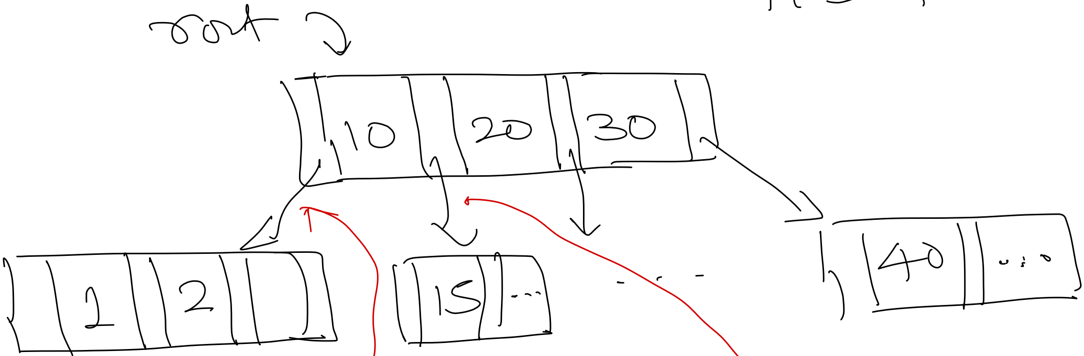
Multi-way

⇓

Each node stores multiple keys.

M-way tree of order M will have M childrens & (M-1) keys.

M = 4



values stored in subtree will be less than 10.

values stored in subtree will be greater than 10 but less than 20.

B - Tree

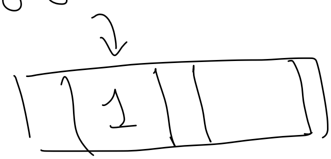
- Root has at least two subtrees unless its the only node.
- New element is added in a leaf node.
- Intermediate node must be $\frac{1}{2}$ full.

B - Tree of order = 3

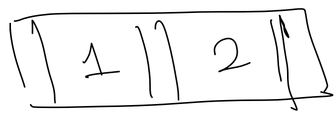
root → empty

Insert (1)

root



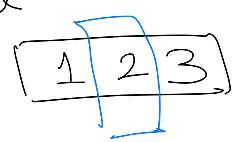
Insert (2)



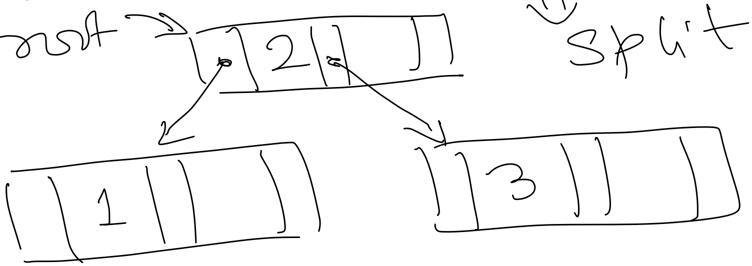
Insert (3) ⇒ Node is full

⇓

Split

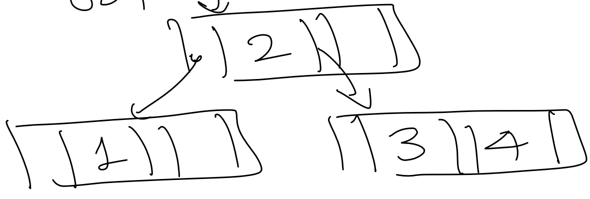


root ⇒

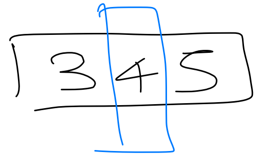


Insert (4)

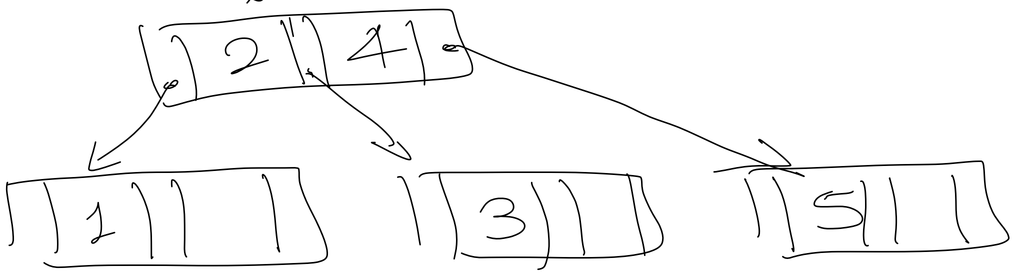
root



Insert (5) => node full
⇓
split

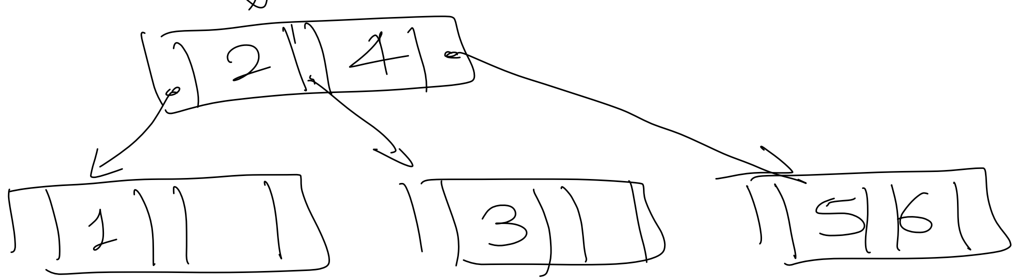


root

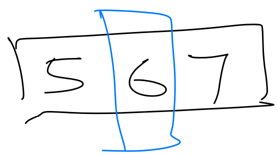


Insert (6)

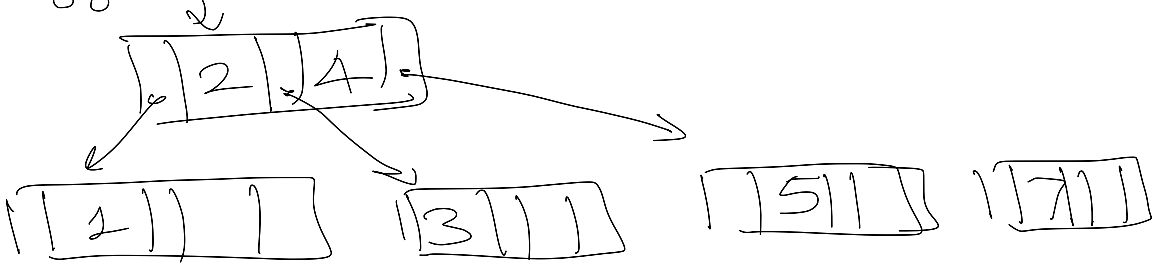
root



Insert (7) => leaf node is full
⇓
Split leaf



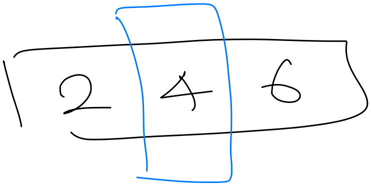
root

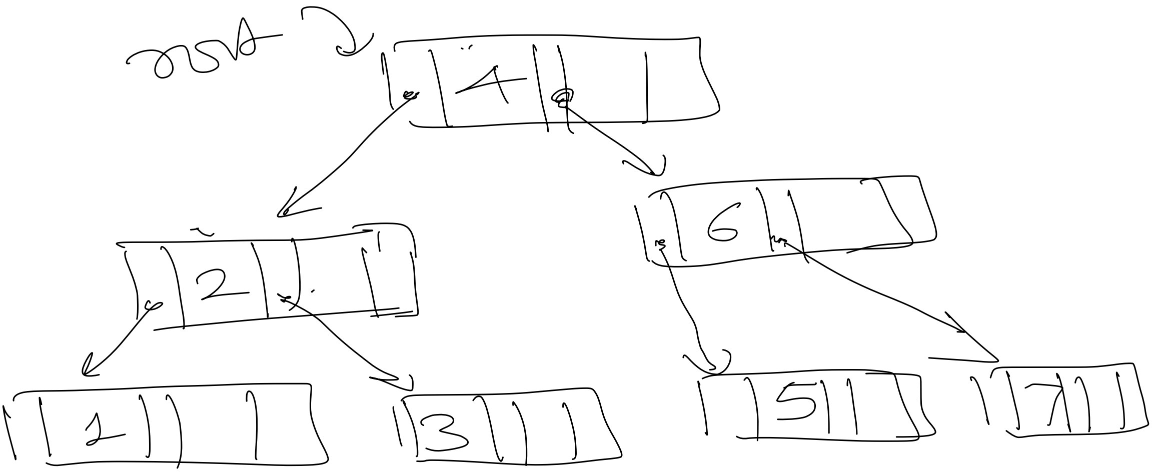


Move 6 to parent ⇒ root also full

⇓

Split root





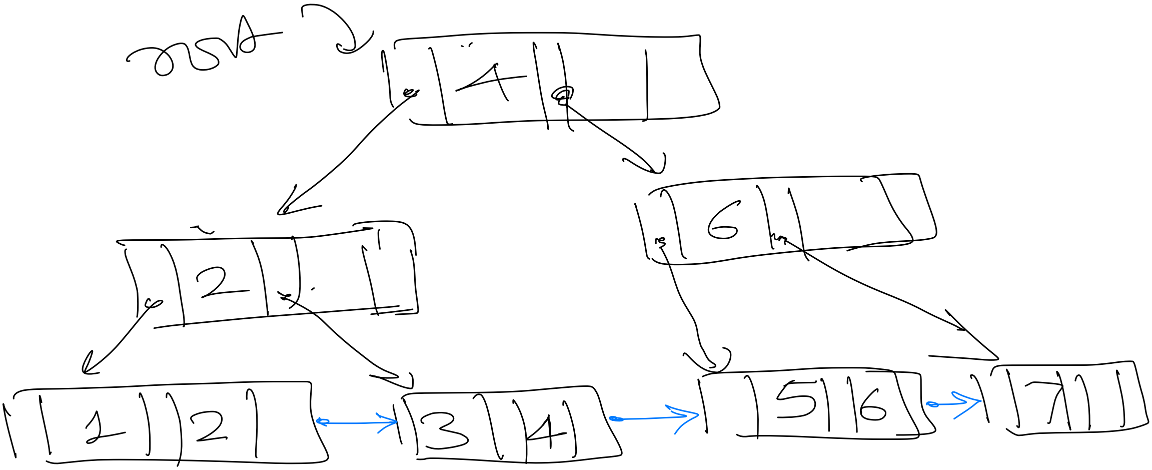
B+ - Tree

↳ During split, middle element is not moved, but copied to parent.

⇓

Keys can be duplicated in B+ Tree

↳ All leaf nodes are linked together.



→ All keys stored in B+ Tree are present in leaf nodes.

B* - Tree

↳ Each intermediate node should b 2/3 full.