hierarchical

Empty tree represented by NULL Pointer

Page No.
Date

Introduction to Trees:-

Datastructure - It is basically a way to organized the data in a way such that we can process the data efficiently.

Data str. -

Linear { arrays, linkedlist, stack, Queue }
→ one level (sequential)

Non-linear { Tree, Graph }
→ multiple level
contain some information

↑ Trees - collection of nodes, linked together to simulate a hierarchy.

Root - Top-most node
- have no parent node

Nodes - contain some information
contain link to another node.

Parent Node - immediate predecessor of any node.

child Node - immediate successor of any node

Grand Parent -

leaf node - The node have no child.
also called - external node.

non-leaf node - atleast one child.
also called - internal node.

Path - sequence of consecutive edges (link b/w to nodes) from source node to destination node.

Ancestor - any predecessor nodes on the path from root to that node.

Decendent - any successor nodes on the path from that node to leaf node.

subtree - contain a node of tree and all its decendents.

sibling - all children of same parent.

cousions - children of different parent.

Height of node may or may not be equal to depth of node.

Degree - no. of child of that node.

→ Degree of leaf = 0.

Degree of Tree - maximum degree of among all ~~any~~ nodes.

depth of node - length of path from root to that node.
OR.
= no. of edges, from root to that node.

→ depth of root node is zero.

Height of node - no. of edges in the longest path from that node to a leaf.

→ Height of tree = Height of Root node
= no. of edges in the longest path from Root to leaf.

Level of Node -
~~level~~ no. of edges from Root to the given node.

Level of tree = Height of tree = Depth of tree

level of node = depth of node ≠ Height of node

[Level of node = Depth of node] always.

[Level of tree = ~~level~~ Height of tree] always.

[Level of tree = Depth of tree] always.

If in a tree -

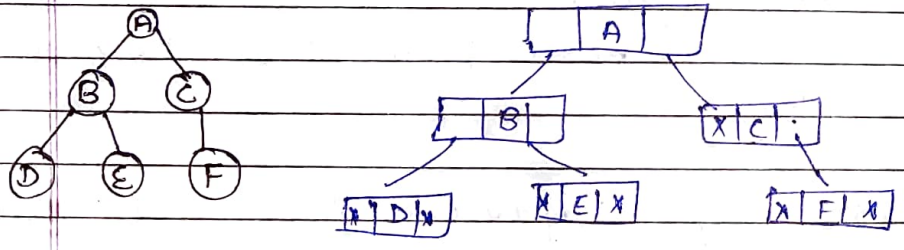
n = no. of nodes

then (n-1) = edges.

i.e. edges = (n-1)

Binary trees -

Each node have at most two children.




```
struct node
{
float / char / int data;
    struct node * left;
    struct node * right;
};
```

Binary tree & its types -

1. Max. no. of nodes at level i = 2^i
2. Max. no of nodes of height h = $2^{h+1}-1$
3. min. " " " " " " = h+1
4. Max. height (h) = n-1
5. Min. height = (h) = $\lceil \log_2(n+1) - 1 \rceil$

h = height — give max. no of nodes → min. height

n = no. of nodes. — give min. no of nodes → max. height

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1} \rightarrow \log_2(n+1) = \log_2 2^{h+1}$

$\log_2(n+1) = h+1$

$h = \lceil \log_2(n+1) - 1 \rceil$

$h = h+1$

$h = n-1$

Perfect binary tree is always Full and complete binary tree.

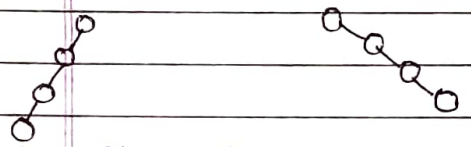
Types -

1. Full / Proper / strict -
   each node have either 0 or 2 children. $L = I + 1$ (L: leaf node, I: internal node)

2. complete Binary tree -
   all levels are completely filled (except possibly the last level) and last level has nodes as left as possible. Ex - Binary heap.
   nodes $\leftarrow n = 2L - 1$ (L: leaf node)

3. Perfect Binary tree - all internal nodes have 2 children and all leaves are at same level.
   height h, $n = 2^h - 1$ (n: node)
   Also called pathological tree

4. degenerated Binary tree - all the internal nodes have only one node.



Left skewed binary tree

Right skewed binary tree

Act as linked list

Height = no. of nodes.

	max height	min height	min nodes	max nodes
Binary tree	$n-1$	$\lceil \log_2(n+1) \rceil - 1$	$h+1$	$2^{h+1} - 1$
Full tree	$\frac{(n-1)}{2}$	$\lceil \log_2(n+1) \rceil - 1$	$2h+1$	$2^{h+1} - 1$
complete Binary tree	$\log n$	$\lceil \log_2(n+1) \rceil - 1$	2^h	$2^{h+1} - 1$

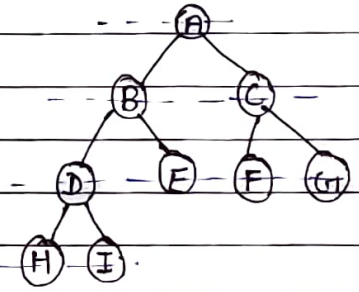
Binary tree implementation -

Array Representation of Binary tree -

Drawback - There is a lot of wastage of space.

L → R

Ex - I



CASE-I

A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

CASE-II

A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9

CASE-I if a node is at ith index -

left child = $[(2 * i) + 1]$

Right child = $[(2 * i) + 2]$

Parent = $\left\lfloor \frac{(i-1)}{2} \right\rfloor$

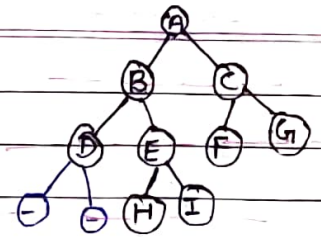
CASE-II if a node is at ith index -

left child = $(2 * i)$

Right child = $[(2 \times i) + 1]$

Parent = $\left\lfloor \frac{i}{2} \right\rfloor$

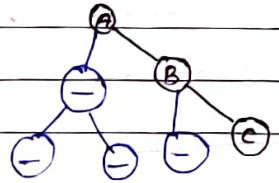
Ex-2



Firstly make a binary tree as a complete binary tree.

A	B	C	D	E	F	G	–	–	H	I	I
0	1	2	3	4	5	6	7	8	9	10	

Ex-3

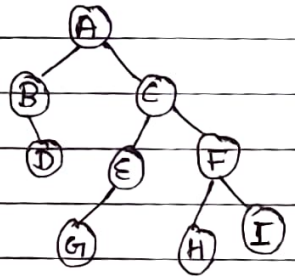


Right skewed -

A	–	B	–	–	–	C
0	1	2	3	4	5	6

Binary tree traversal -

Preorder - Root left Right
Inorder - left Root Right
Postorder - left Right Root



Preorder- A B D C E G F H I

Inorder- B D A G E C H F I

Postorder- D B G E H I F C A

TRYING is better than CRYING

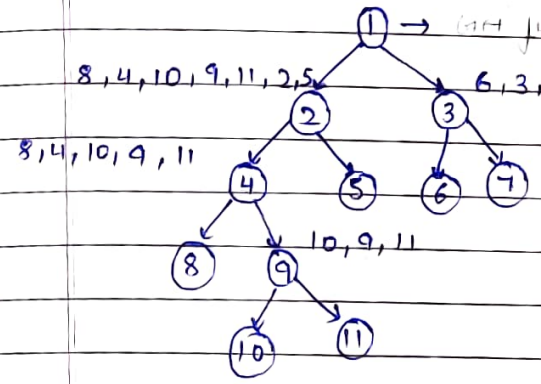
→ Binary tree traversal using c code

L ⟶ R

→ construct a Binary tree from Preorder and Inorder

Preorder- 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 (Root left Right)
L ⟶ R

In-order 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 (left Root Right)



L ← R

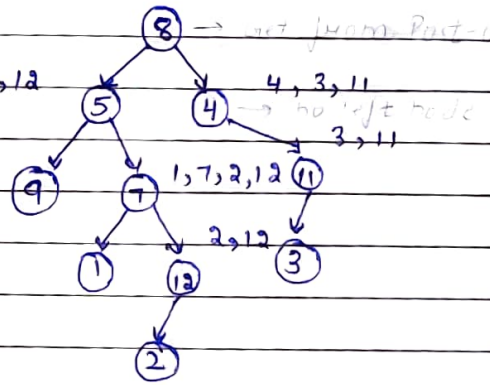
→ Construct Binary tree from Postorder and Inorder -

(left Right Root)

Post-order- 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

L ← R (left Root Right)

In-order- 9, 5, 1, 7, 2, 12, 8, 4, 3, 11



Construct Binary Tree from Preorder and Postorder traversal -

→ If only Preorder and Post order are given, you can't construct a unique binary tree.

→ You can construct a full unique binary tree from Preorder & Postorder

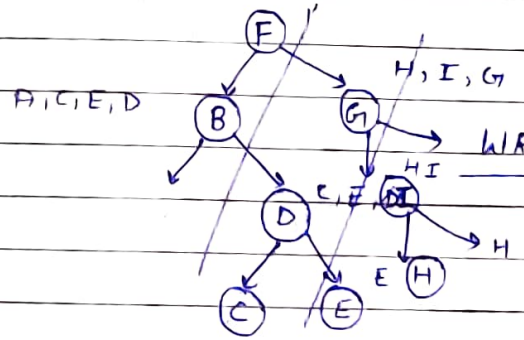
(Root left Right)

Preorder- F, B, A, D, C, E, G, I, H

L → R

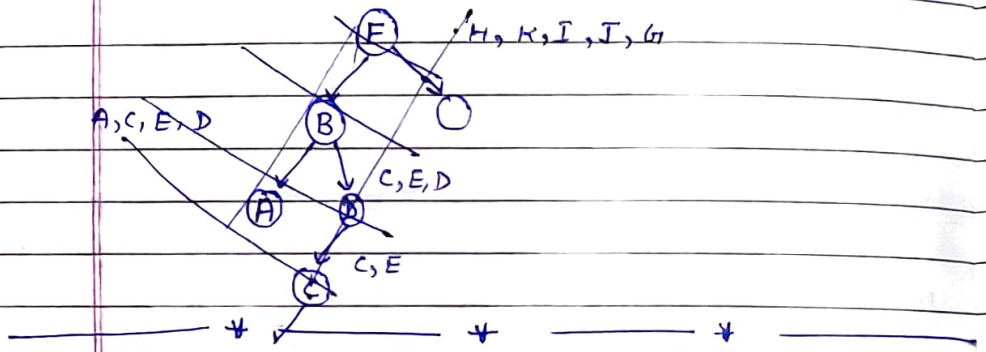
Postorder- A, C, E, D, B, H, I, G, F

(left Right Root)



Preorder- F B A D C E G I H K J (Root L R)

Postorder- A C E D B H K I J G E (L R Root)



Preorder- F B A D C E G I H

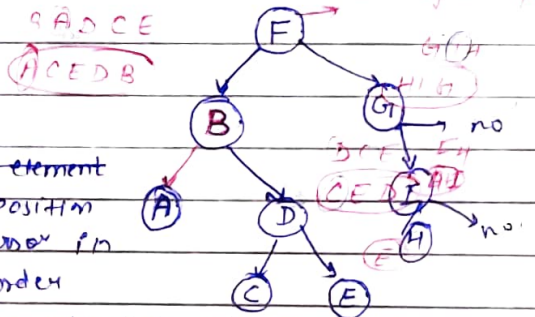
Postorder- A C E D B H I G F

(i) Write Root element ⟶ (1st of Pre / last of Post)

(ii) check succesor of Preorder

(iii) ~~Put all element~~ check position of succesor in Postorder

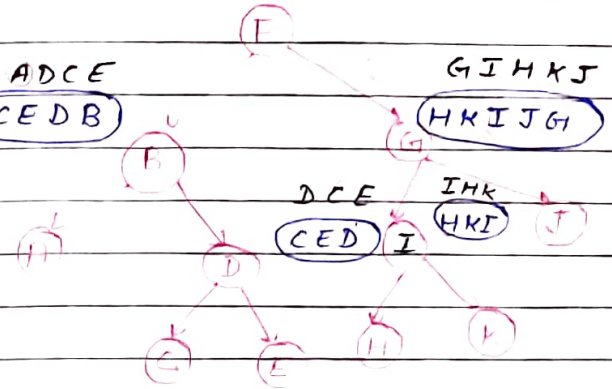
(iv) All the elements left hand side of succesor and succesor put in () node and remaining elements in another node.



sol.n-

Pre - B A D C E

Post - A C E D B

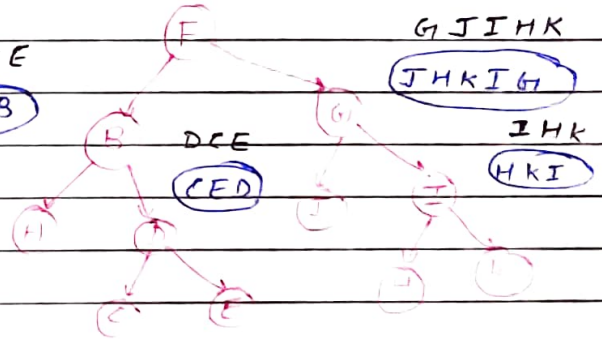


Preorder- F B A D C E G J I H K

Postorder- A C E D B J H K I G F

Pre - B A D C E

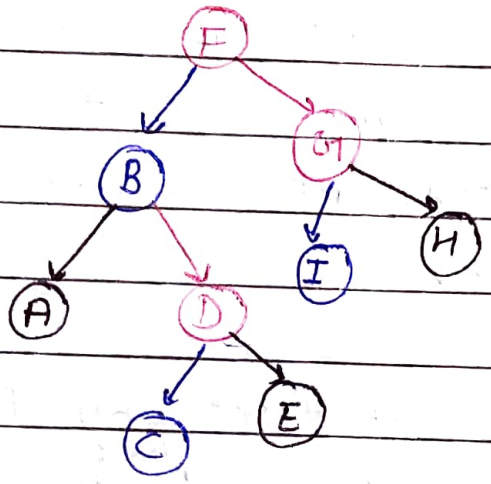
Post - A C E D B



Trick →

Preorder- F B A D C E G I H

Postorder- A C E D B H I G F



Inorder of Binary search tree is always in ascending order

(BST) (Binary search Tree)

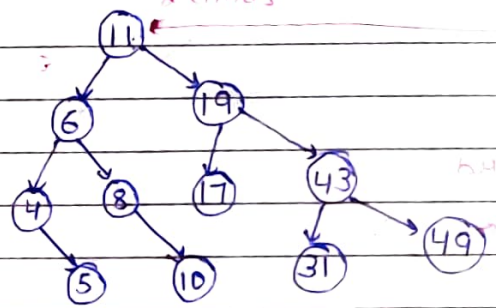
Duplicate elements are not allowed in BST, i.e we can't insert duplicate elements in the BST.

→ Atmost 2 childs. left

→ Kisi bhi node ka subtree me sabi child ki values lesser hogi node se and Right subtree k sabi child ki values greater hogi node se.

(I) Draw Binary search tree by inserting the following numbers ~~form~~ word left to Right.

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



(II) Insert a number (60) in above BST at its correct position.

(a) Insert a number (36) in above BST at its correct position.

(III) Deletion -

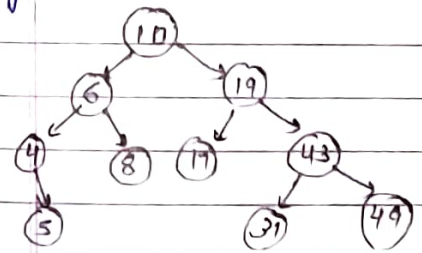
3 cases - 1. zero child
2. 1 child
3. 2 child

1. zero child -

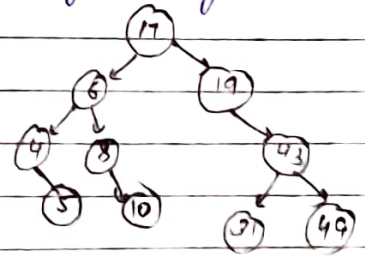
2. 1 child -

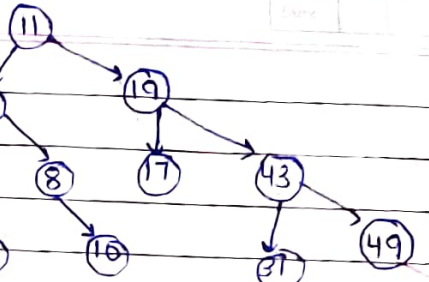
3. 2 child - 2 cases - (I) Inorder predecessor
(II) Inorder successor

Inorder predecessor -
→ largest node value from left subtree



Inorder successor -
→ smallest node value from Right subtree





construction of BST when only Preorder or Postorder is given.

(I) when only Preorder is given :-

(Root L R)

Preorder- 20, 16, 5, 18, 17, 19, 60, 85, 70.

(L → R)

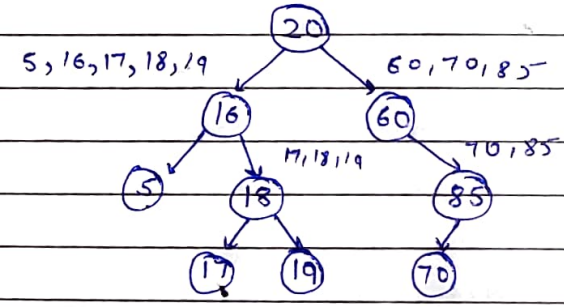
3 questions- Find Inorder traversal.
Find Postorder traversal.
const. BST.

Inorder traversal of BST is always in ascending order.

Inorder- 5, 16, 17, 18, 19, 20, 60, 70, 85

(L Root R)

BST-



(L R Root)

Post order- 5, 17, 19, 18, 16, 70, 85, 60, 20.

(II) only Post order is given- (Left Right Root)

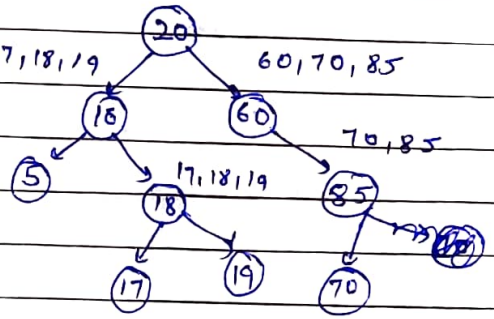
Postorder- 5, 17, 19, 18, 16, 70, 85, 60, 20

(L ← R)

Inorder- 5, 16, 17, 18, 19, 20, 60, 70, 85

(L Root R)

BST-

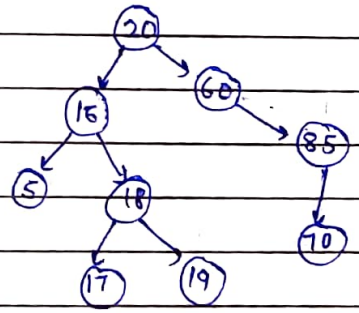


Preorder- 20, 16, 5, 18, 17, 19, 60, 85, 70.

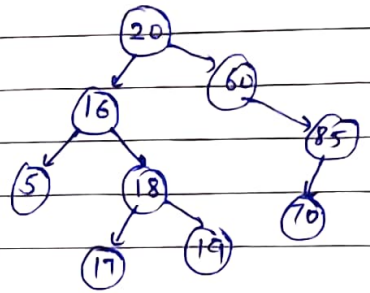
दो Bina Inorder Find करनो से भी BST बना सकते है।

(L ← R)

Postorder- 5, 17, 19, 18, 16, 70, 85, 60, 20.



Preorder - 20, 16, 5, 18, 17, 19, 60, 85, 70.



AVL Trees -

self-balanced tree
Height balanced tree

→ It is BST.

→ For each node -

$$|\text{height of left subtree} - \text{height of Right subtree}| = \{-1, 0, 1\}$$

This is known as **Balance Factor**-

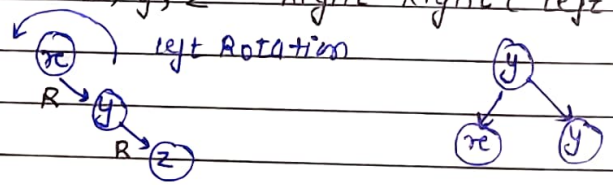
if the balance factor is not equal to $\{-1, 0, 1\}$ then we must have to balance the tree -

4 cases -

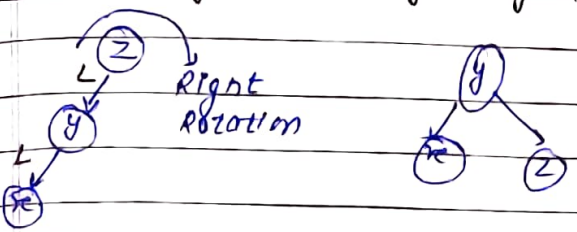
① Suppose we have two insert 3 elements - (x, y and z)

then -

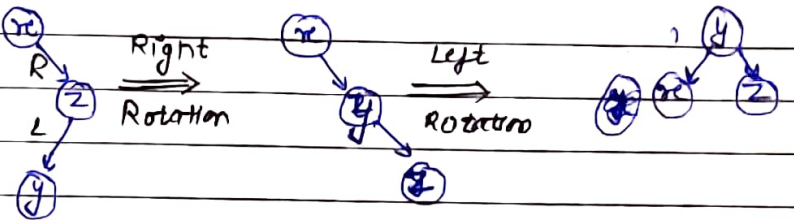
CASE-I - x, y, z Right Right (Left Rotation)



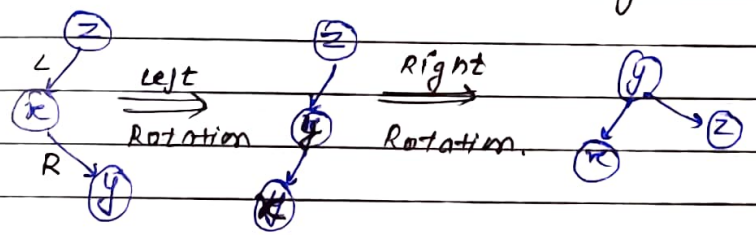
CASE-II z, y, x Left Left (Right Rotation)



CASE-III x, z, y Two Rotation [1. Right Rotation 2. Left Rotation]



CASE-IV z, x, y Two Rotation [1. Left Rotation 2. Right Rotation]



Trick x, y, z

(i) Find middle element of 3 elements
(ii) Put smaller element left side
(iii) Put largest element Right side

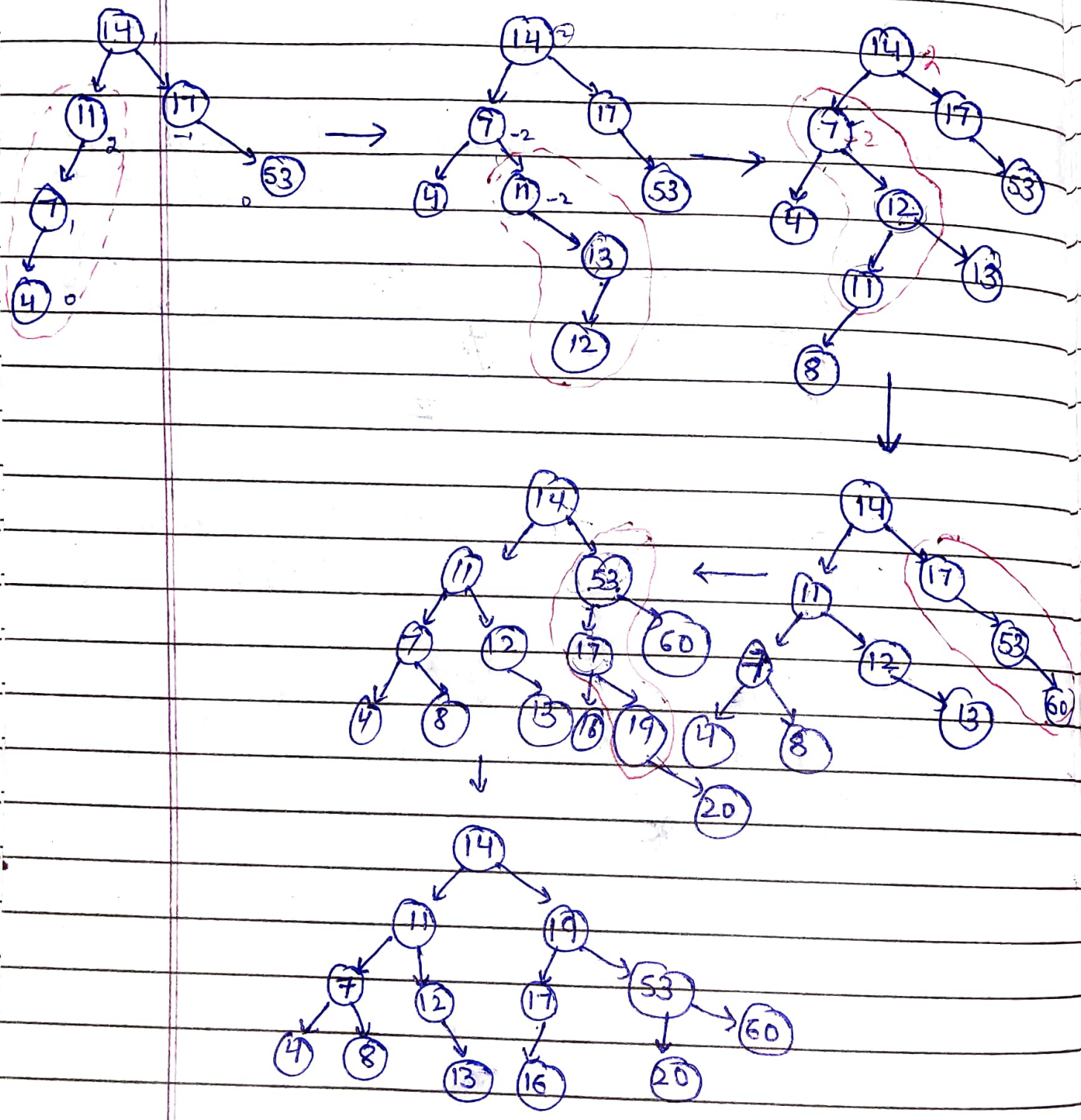
Insertion -

AVL

Page No.
Date

construct AVL tree by inserting the following data -

14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20



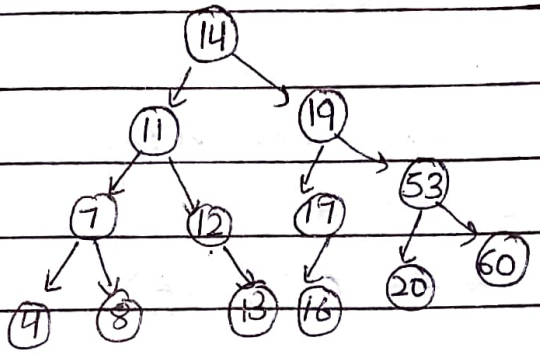
Page No.
Date

AVL- Delection-

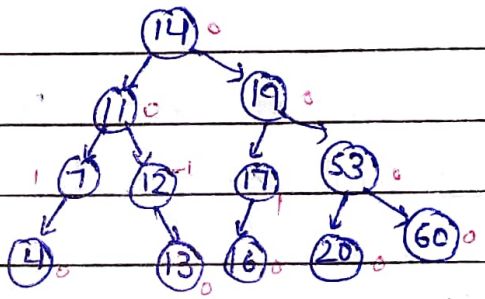
→ It is same as in BST.

→ check balanced factor of each node after each delection.

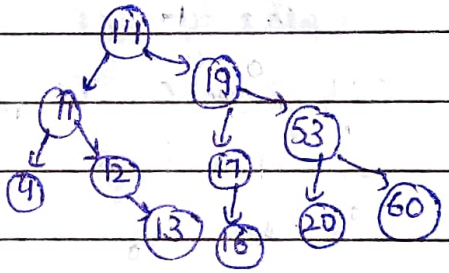
if ~~del~~ balanced factor not equal to {-1, 0, 1} then must be balance the tree.



(I) Delete -8

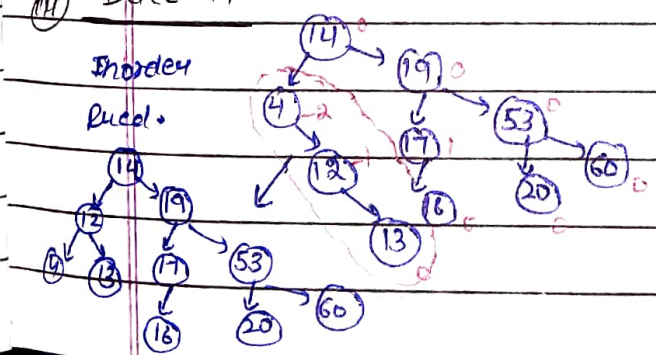


(II) Delete -7-

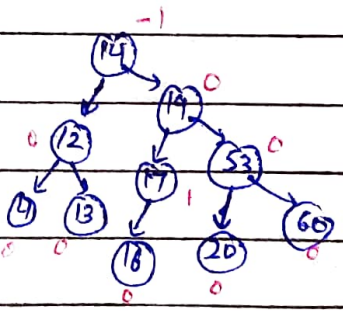


(III) Delete 11-

Inorder Pred.

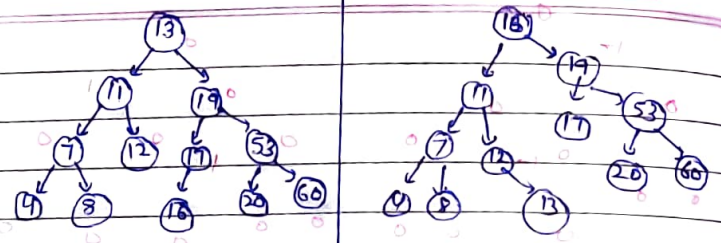


Inorder Succes.



Inorder Pre.

Delete 14 -



Ino. Succ.

BST -

Time complexity - (For searching

best case - O(1)

Average case - O(log₂n)

Worst case - O(n)

Red-Black ~~tree~~

AVL Time complexity - (For searching)

for all cases is O(log₂n)

AVL → Time complexity - (For searching)

for all cases is O(log₂n)

same in AVL & RB trees

In AVL, we have to do a lot of Rotation, but in RB trees we have to do only max 2 Rotations & Recoloring. This is advantage of Red Black over AVL.

11 September 2020

Introduction to Red-Black trees -

→ It is a BST.

→ Self balanced Binary search tree.

→ Each node contains an extra bit for denoting the color of the node (either Red or B)

Time complexity → AVL → O(log₂n)

→ strictly Height-balanced tree.
→ more balanced

Red Black tree → roughly height-balanced tree.
→ less balanced.

Time complexity → Red-Black tree → O(log₂n)

searching → AVL

Insertion / Deletion → Red-Black tree

→ It is a self-balancing BST.

→ Every node is either Black or Red

→ Root is always black.

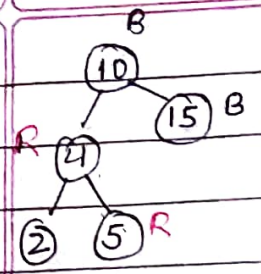
→ Every leaf which is NIL is Black.

→ If node is Red then its children are Black.

→ Every Path from a (Root) node to any of its descendent NIL node has same no. of Black nodes.

Page No.
Date

(iv)



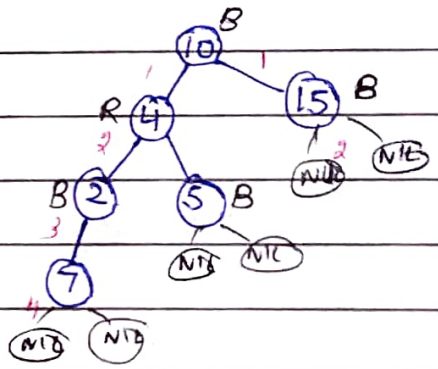
NOT ∵ Red ke Red child nahi ho sakthaj

→ The longest path from the root node is no more than the twice of shortest path

OR

→ The path b/w root and furthest leaf node is no more than twice the path b/w root and nearest leaf node.

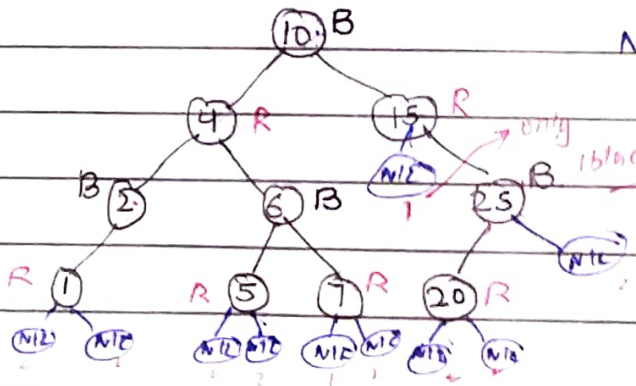
Ex →



4 is not greater than 2 (2)

Hence, it follows above property.

(v)



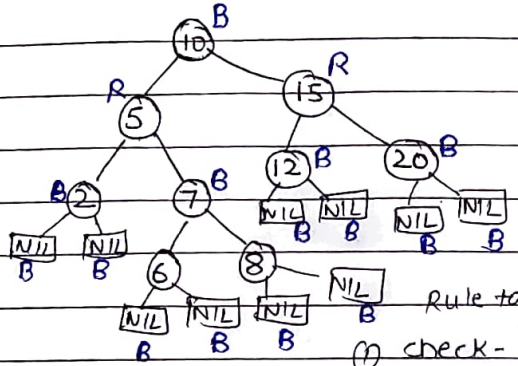
NOT ∵ No. of black node are not equal.

(by coloring the nodes acc. to rules)

→ AVL tree is subset of R-B tree.

→ R-B tree are not AVL trees.

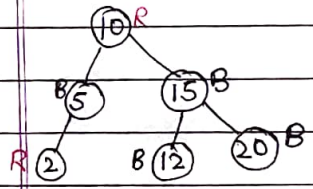
Ex →



Rule to check it is R-B or not

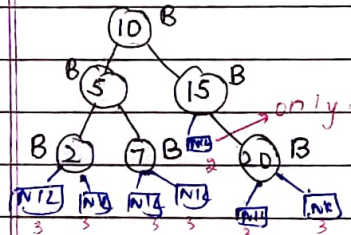
(i) check - it is BST or not
(ii) check the rules of RB tree.

(i)



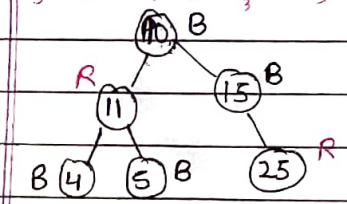
NO [∵ Root → Red]

(ii)



NO [∵ NO. of black nodes are not equal].

(iii)



NO [∵ It is not BST]

↔ Move back to 2 pages

Note → Every Perfect Binary tree contains all Black nodes must be a Red-Black tree.

Insertion in RB Tree

10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70.

(i) If tree is empty → create newnode as root node with color black.

(ii) If tree is not empty, create newnode as leaf node with color Red.

(iii) If Parent of newnode is black then exit.

(iv) If Parent of newnode is Red, then check the color of parent's sibling of newnode.
↓ Uncle

(a) If color is black or null then do suitable rotation & recolor.
(recolor other 2 nodes except root of 2 nodes which become middle node)

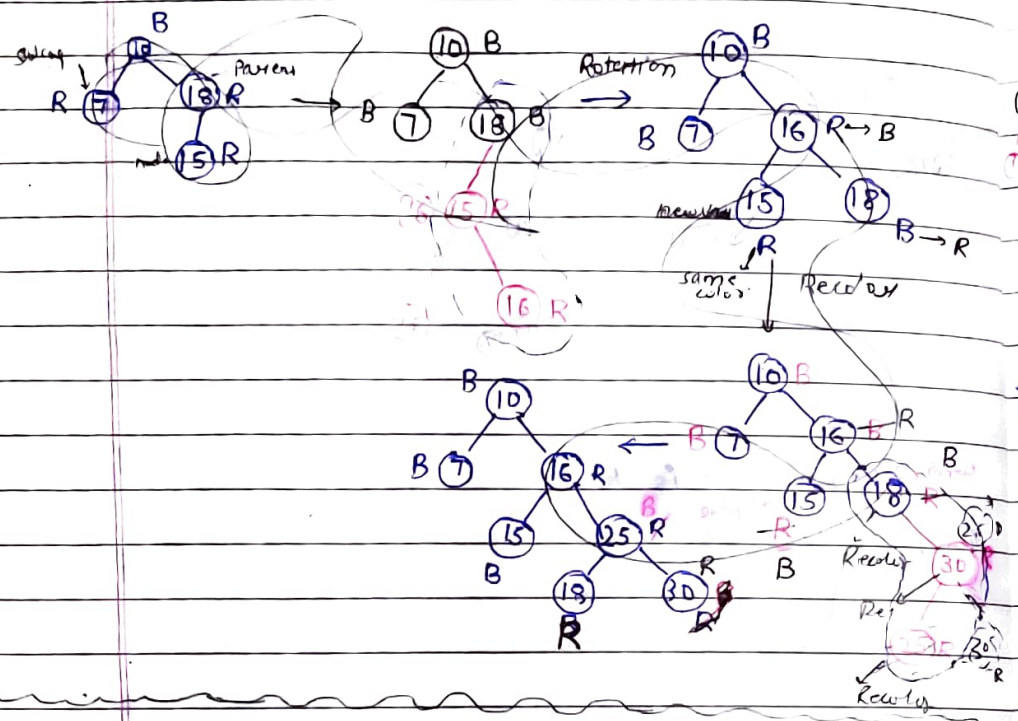
(b) If color is Red, then recolor & also, recolor both Parent's sibling.
check if parent's Parent of newnode is not root node then recolor it & recheck.

Every RB tree must be BST.
But
every BST need not to be RB tree.

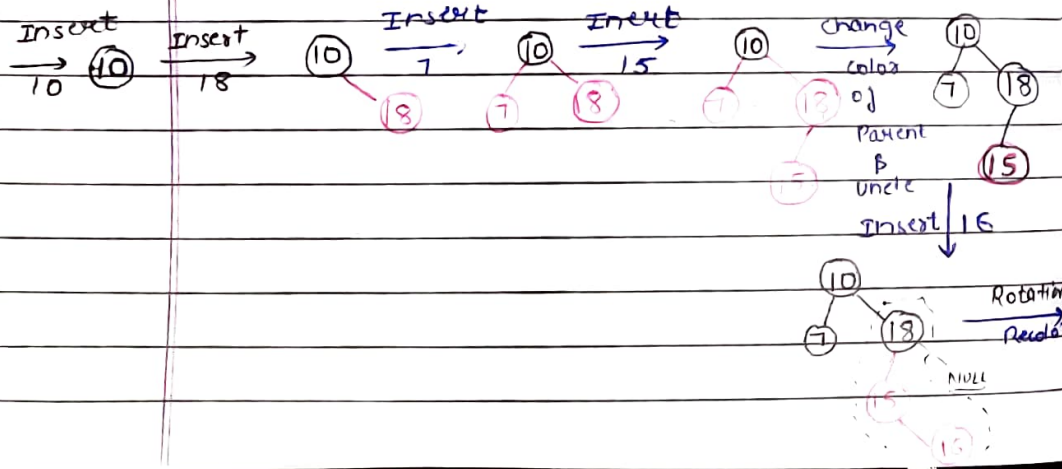
Page No.
Date

Insertion -

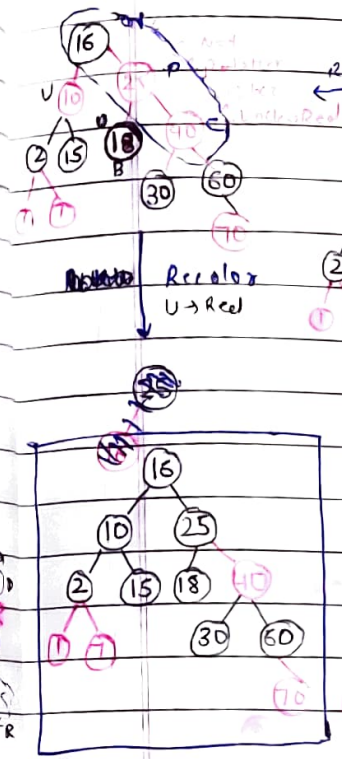
Ex- 10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70

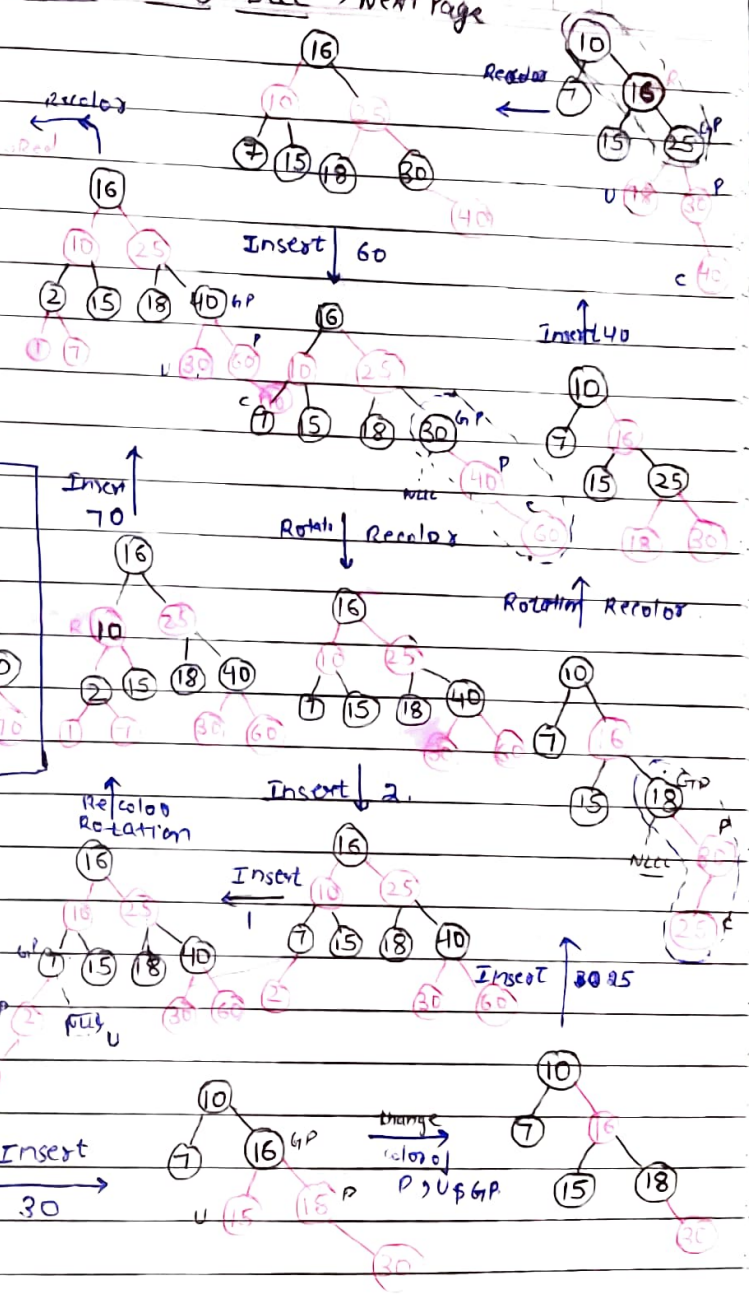


Ex- 10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70



→ Delection - R-B Tree → Next Page





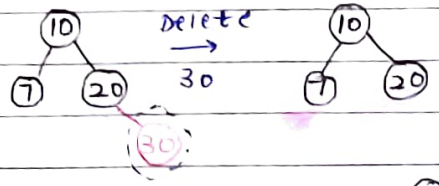
DELECTION – RB Tree –

step-1 Perform BST delection.

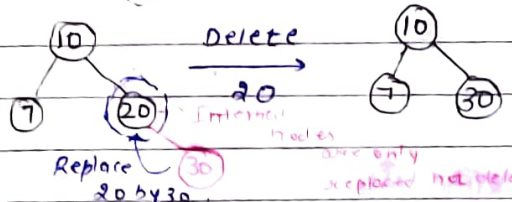
step-2

case-I – If node to be deleted is Red just delete it.

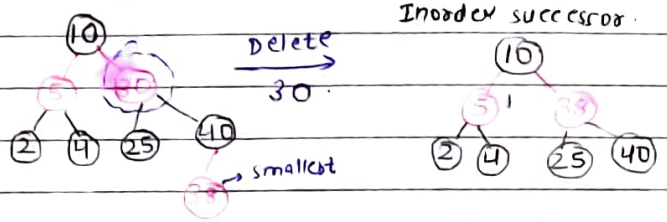
(i)



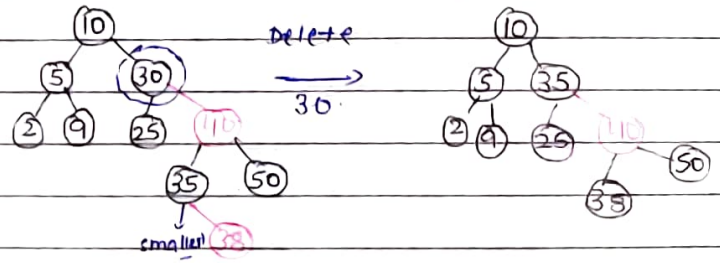
(ii)



(iii)



(iv)



case-2 If root is DB (double black), just remove DB.

case-3 If DB's sibling is black & both its children are black. → Remove DB

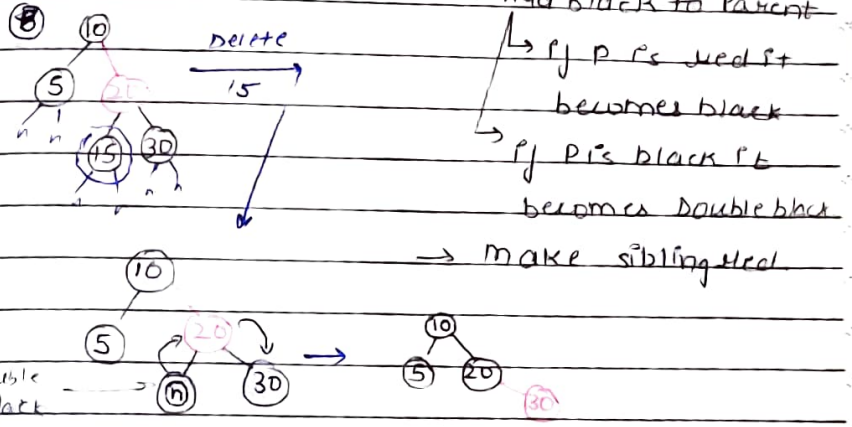
→ Add black to Parent

↳ If P is red it becomes black

↳ If P is black it becomes Double black

→ make sibling Red

(i)



→ If still DB exits, apply these cases.

CASE-4 If DB's sibling is red.

→ swap colour of parent & its sibling

→ Rotate parent in DB direction.

→ Reapply cases.

CASE-5 – DB's sibling is black, sibling's child who is far from DB is black, but near child to DB is red.

→ swap colour of DB's sibling and sibling's child who is near to DB.

→ Rotate sibling in opposite direction to DB.

→ apply case 6.

near → black

Case-6 DB's Sibling is black, far child is red.

~~→ swap color of DB's sibling & SI~~

→ swap color of Parent & Sibling

→ Rotate parent in DB's direction

→ remove DB

→ change color of red child to black

→ 55, 30, 90, 80, 50, 35, 15, 65, 68