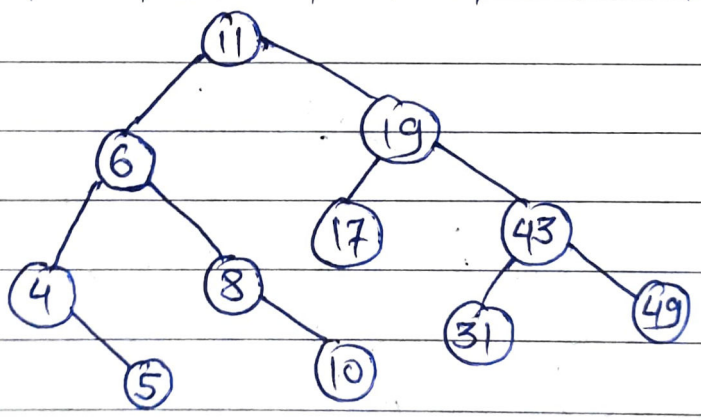
* Binary Search Tree - Insertion and Deletion Explained,

- Can have 0, 1, 2 children nodes.
- Left subtree elements should be lesser than root node and right subtree elements should be greater than root nodes.
- Insertion:

Ex. 1) Draw Binary search tree by inserting the following numbers from left to right.

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

→



- Deletion:

→ 0 children node.

→ 1 child

→ 2 children.

① If node we have to delete has 0 child, then simply delete that node by vanishing the link.

② If node we have to delete has 1 child, then replace the node with the child, and can be deleted.

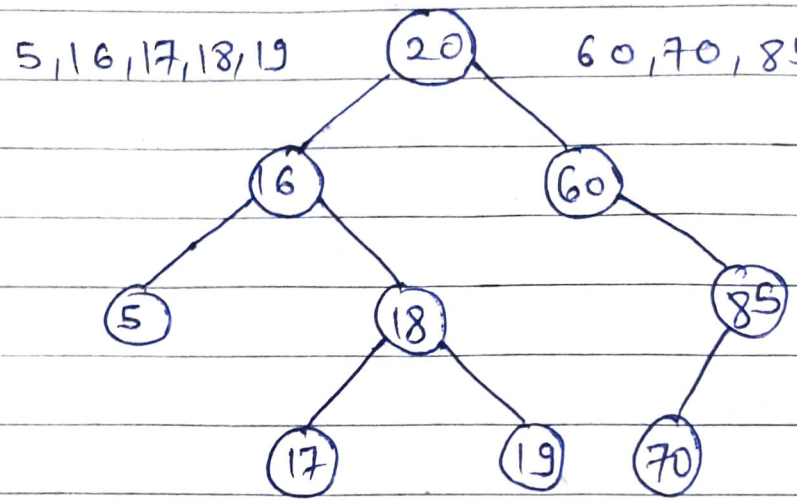
③ If node we have to delete has 2 children, then it might be replaced by either
(i) inorder predecessor i.e. largest element from left subtree of BST
(ii) inorder successor i.e. smallest element from right subtree of BST.

V-59

* Construction of BST when only preorder is given or only postorder is given:
- Preorder: 20, 16, 5, 18, 17, 19, 60, 85, 70 (Root L R)
Find BST, postorder and inorder.
→ Inorder traversal of a BST is always ascending order of elements.
⇒ Inorder would be:
5, 16, 17, 18, 19, 20, 60, 70, 85

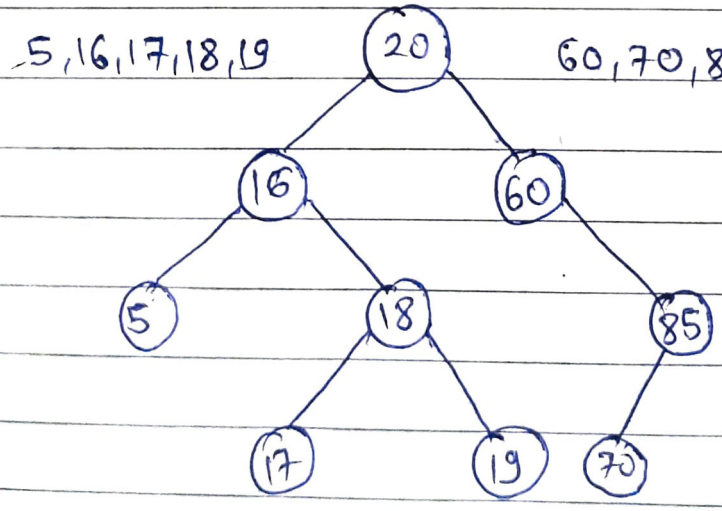
Now we know how to construct a tree using inorder and preorder.

Preorder: 20, 16, 5, 18, 17, 19, 60, 85, 70 (Root L R)
Inorder: 5, 16, 17, 18, 19, 20, 60, 70, 85 (L Root R)



V-60

* Construction of BST given postorder:
- Post order: 5, 17, 19, 18, 16, 70, 85, 60, 20 (L R Root)
  Inorder: 5, 16, 17, 18, 19, 20, 60, 70, 85 (L Root R)



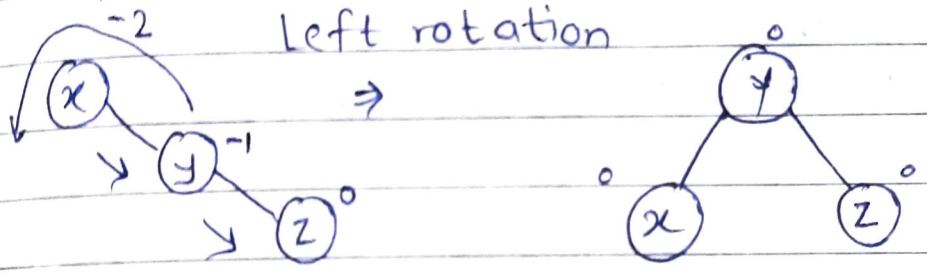
* AVL Tree - Insertion, Rotations (LL, RR, LR, RL):

- It is a BST.
(No duplicate elements are allowed)
- Balancing factor = |height of left subtree| - |height of right subtree|
= {-1, 0, 1}

- Types of rotations to balance AVL tree.

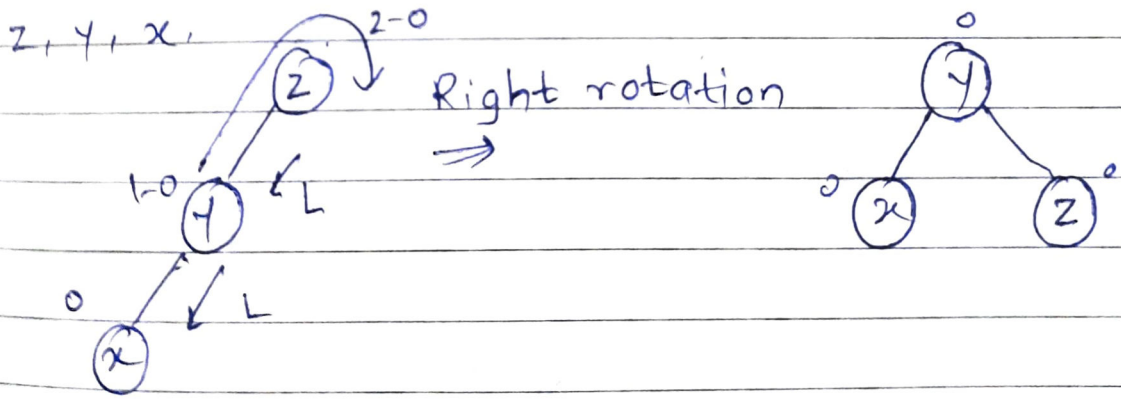
→ RR rotation.

x, y, z.



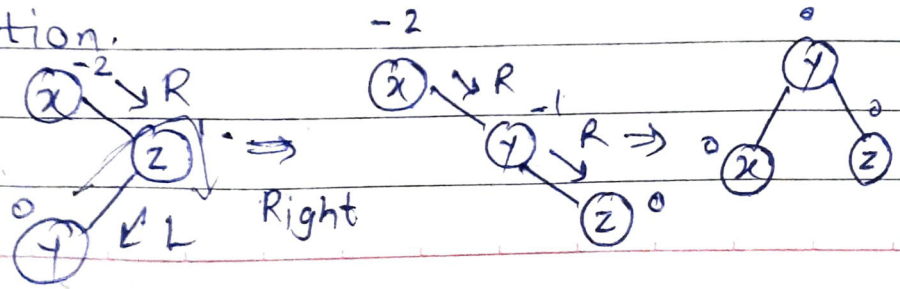
→ LL rotation.

z, y, x.



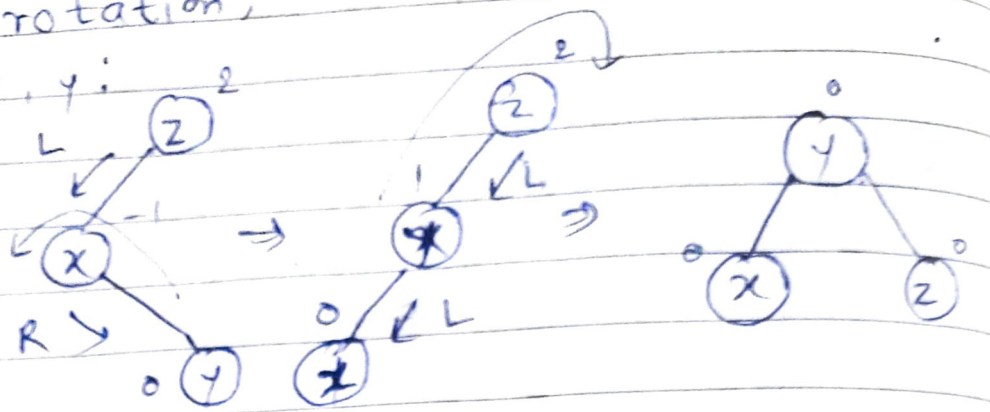
→ RL rotation.

x, z, y



⇒ LR rotation;

z, x, y;

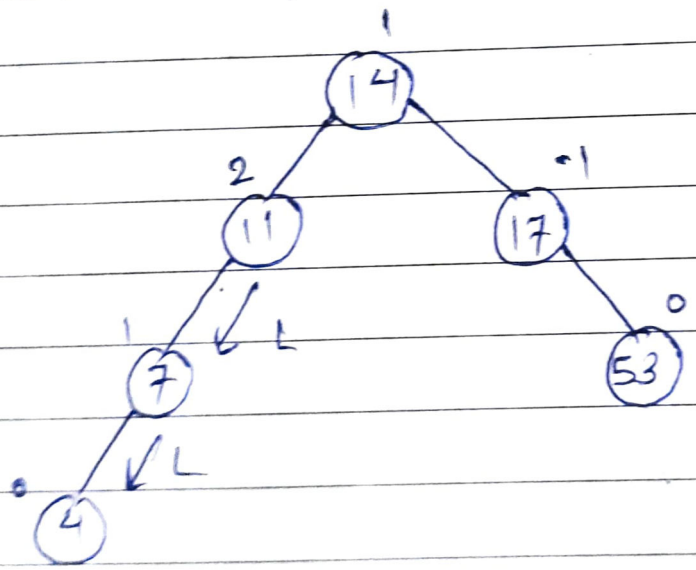


V-62

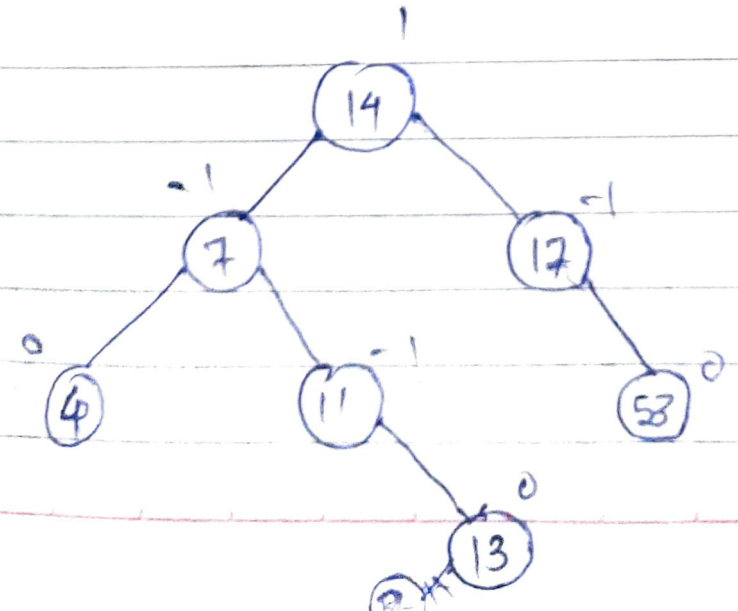
* AVL tree insertion

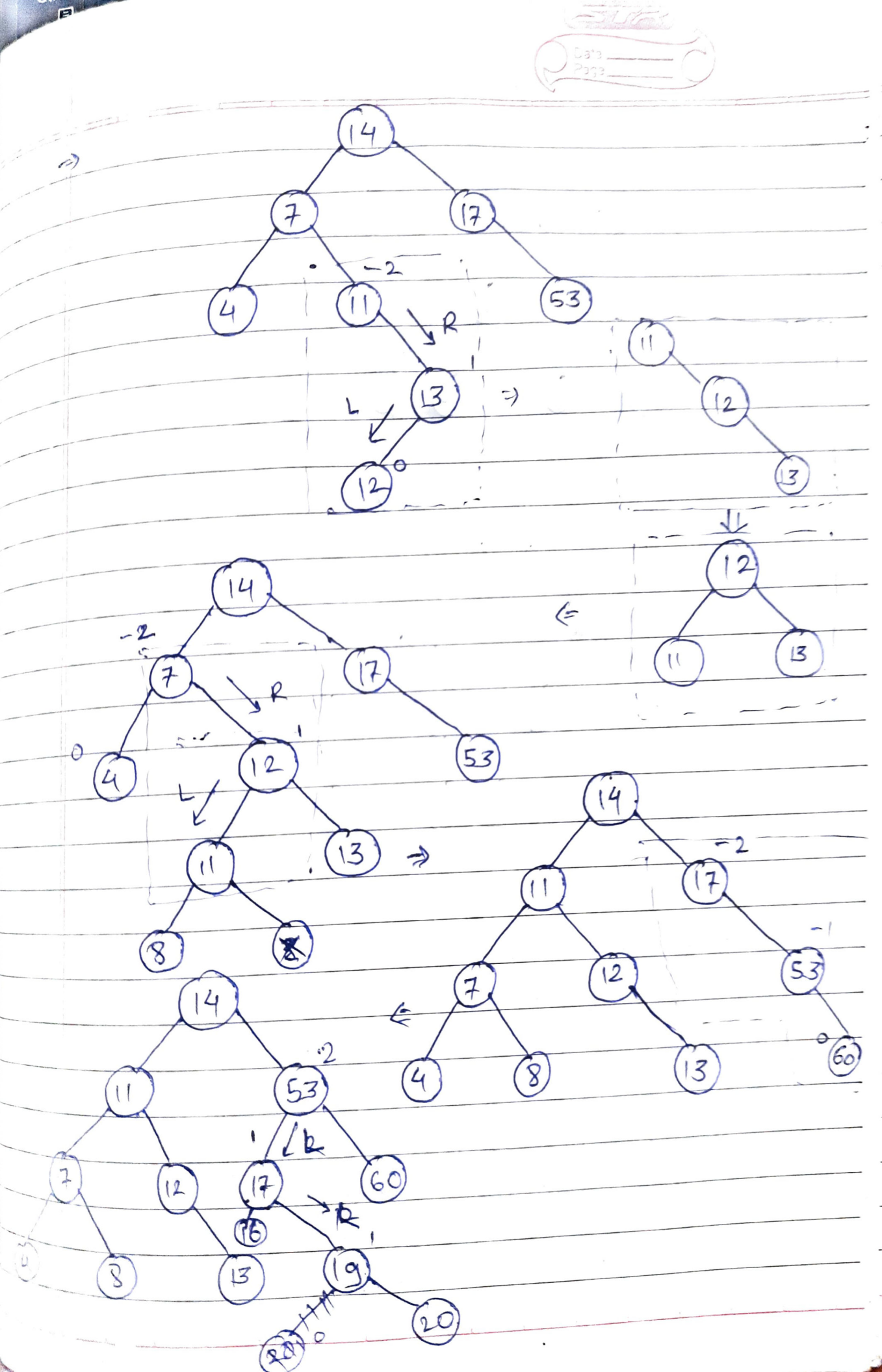
• Construct AVL tree by inserting the following data:

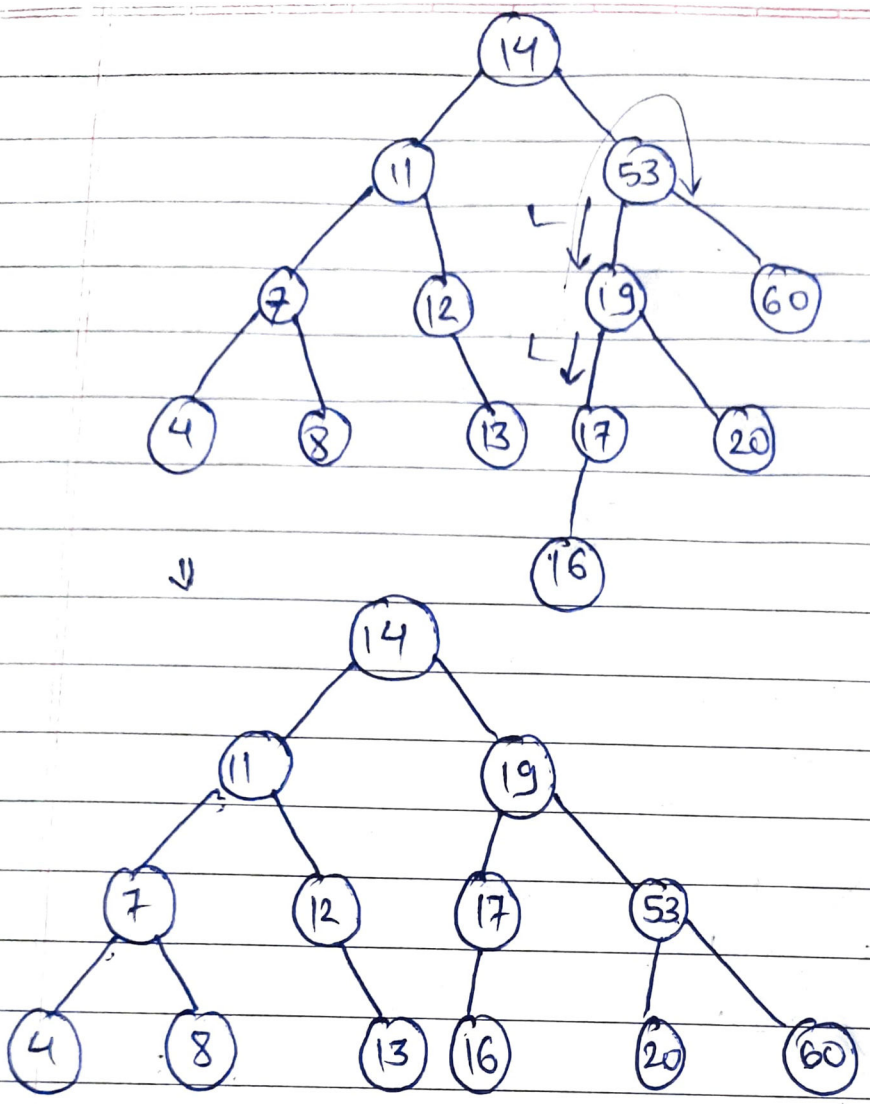
14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20.



⇒




14
7
17
4
11
53
-2
R
13
L
12
11
12
13
14
-2
7
R
17
4
12
53
L
11
13
8
14
11
53
2
7
12
17
60
16
R
8
13
19
20
20
14
11
-2
17
7
12
-1
53
4
8
13
60



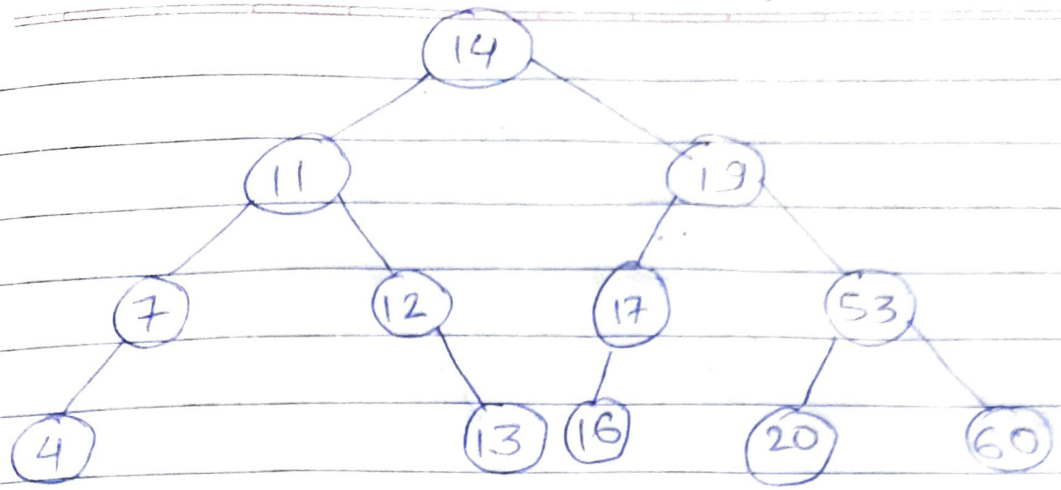
V-63

* Deletion in AVL Tree.

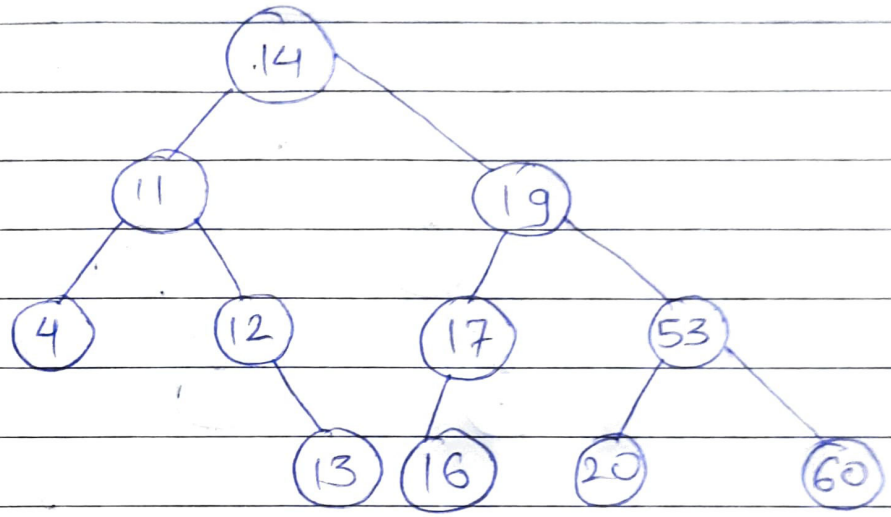
- We want to delete 8, 7, 11, 14, 17 from tree we created above.

- It follows rules of BST for deletion. the additional thing here is we have check balancing factor and balance tree if needed.

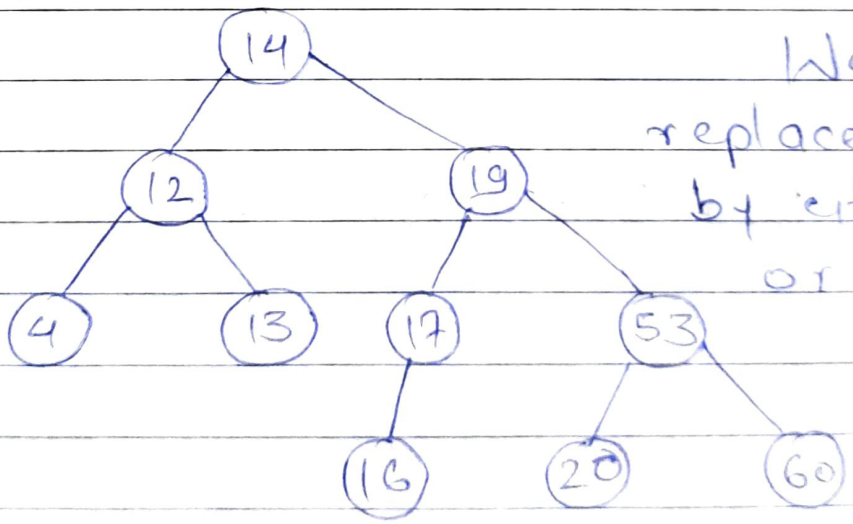
i) 8



ii) 7



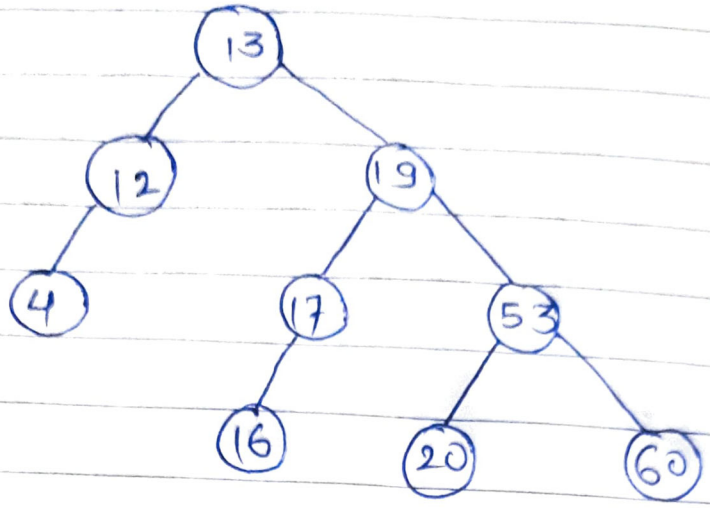
iii) 11



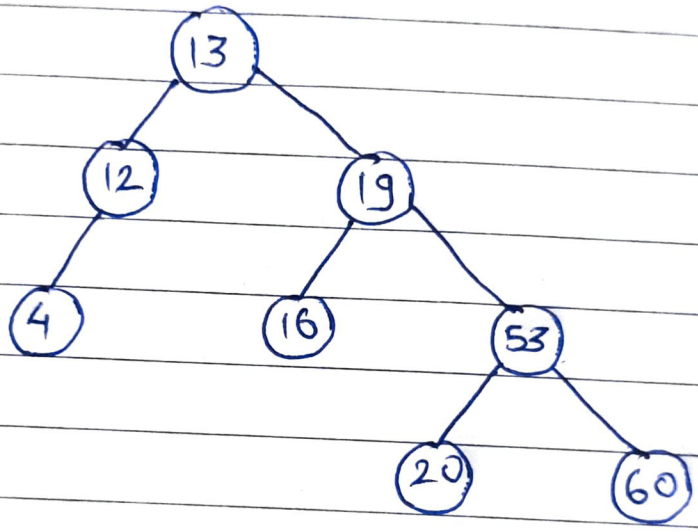
We can replace 11 by either 12 or 4.

If we replace 11 by 4, then we have to balance tree, but if we replace 11 by 12 tree would be already balanced.

iv) 14



v) 17



* Red-Black tree:

- AVL trees are highly height-balanced whereas Red-black trees are not that strictly balanced.

- Hence, AVL trees are more likely used for searching purpose whereas Red-Black trees are used for insertion or deletion

- More rotations are needed when any element is inserted or deleted for balancing in AVL trees compared to that of Red-Black tree, while Red-Black trees only need 2 rotations at max and recolouring; Hence, Data structure Red-Black tree is required.

- Properties of 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 it's children are Black.
  - → Every path from a node to any of it's descendant NIL node has same number of Black nodes.
  - → AVL trees are subset of Red-Black trees.

- Every perfect binary search tree that contains all the nodes black is also a red-black tree.

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

* Red-Black Tree Insertion:

• Rules:

① If tree is empty, create newnode as root node with color black.

② If tree is not empty, create newnode as leaf node with color red.

③ If parent of newnode is black then exit.

④ If parent of newnode is Red, then check the color of parent's sibling of newnode.

ⓐ If color is black or null then do suitable relation and recolor.

ⓑ If color is red then recolor and also check if parent's parent of newnode is not root node then recolour it and recheck.

• Also it should maintain properties of a ~~red~~ black tree:

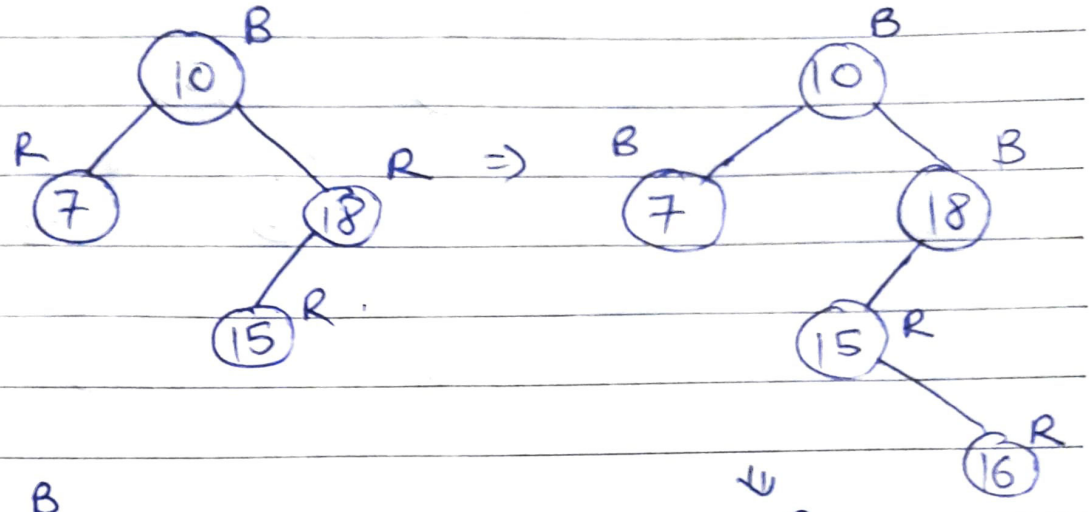
→ root = Black.

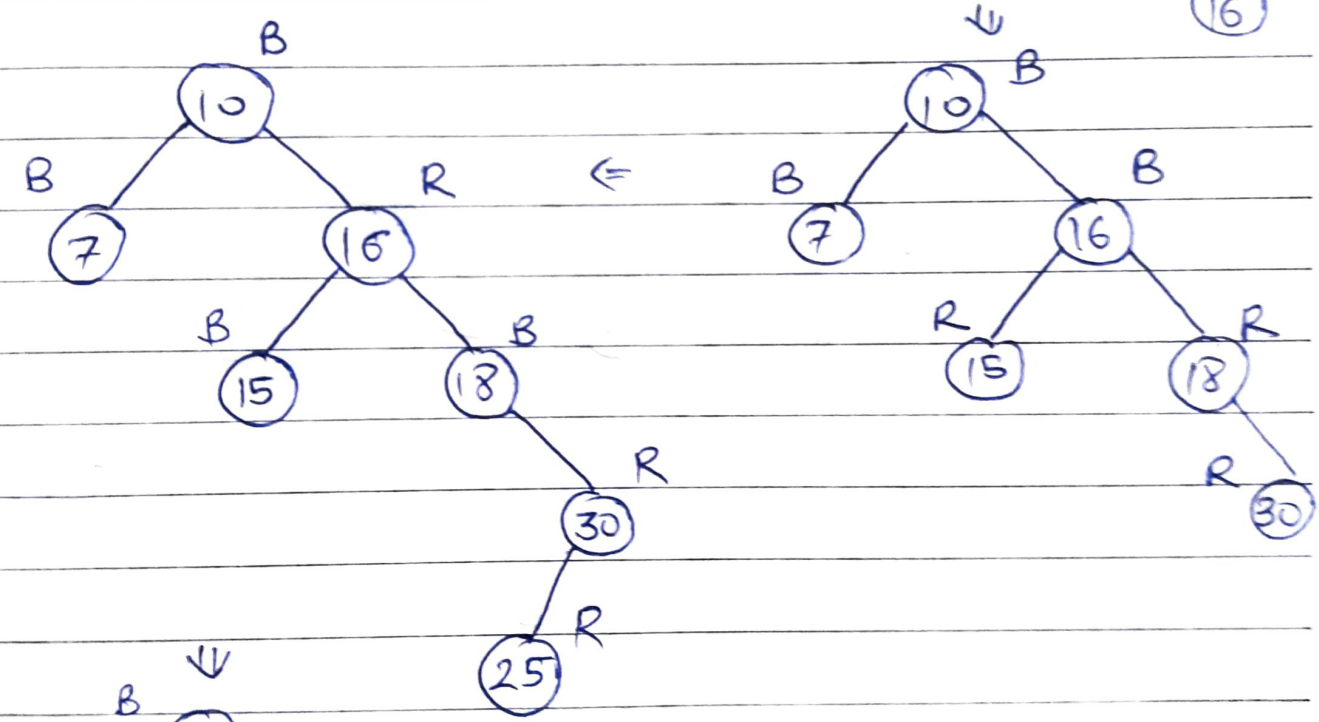
→ No two adjacent red nodes.

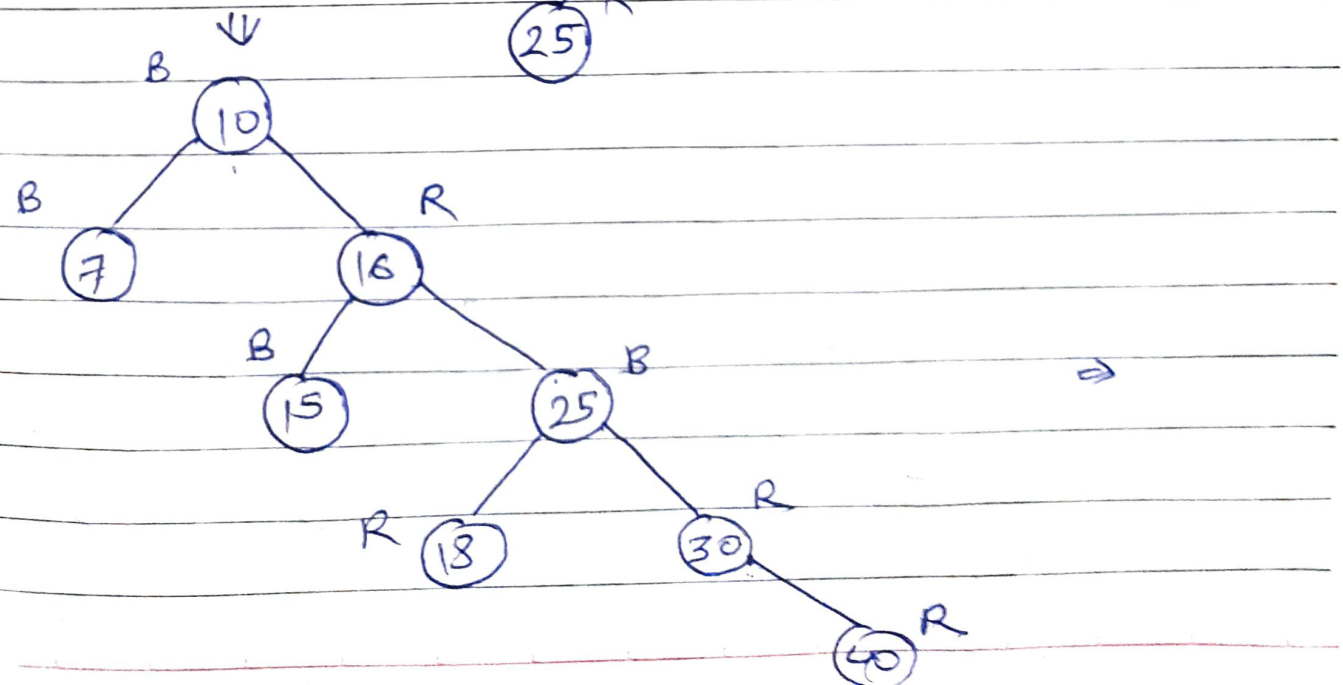
→ count number of black nodes in each path.

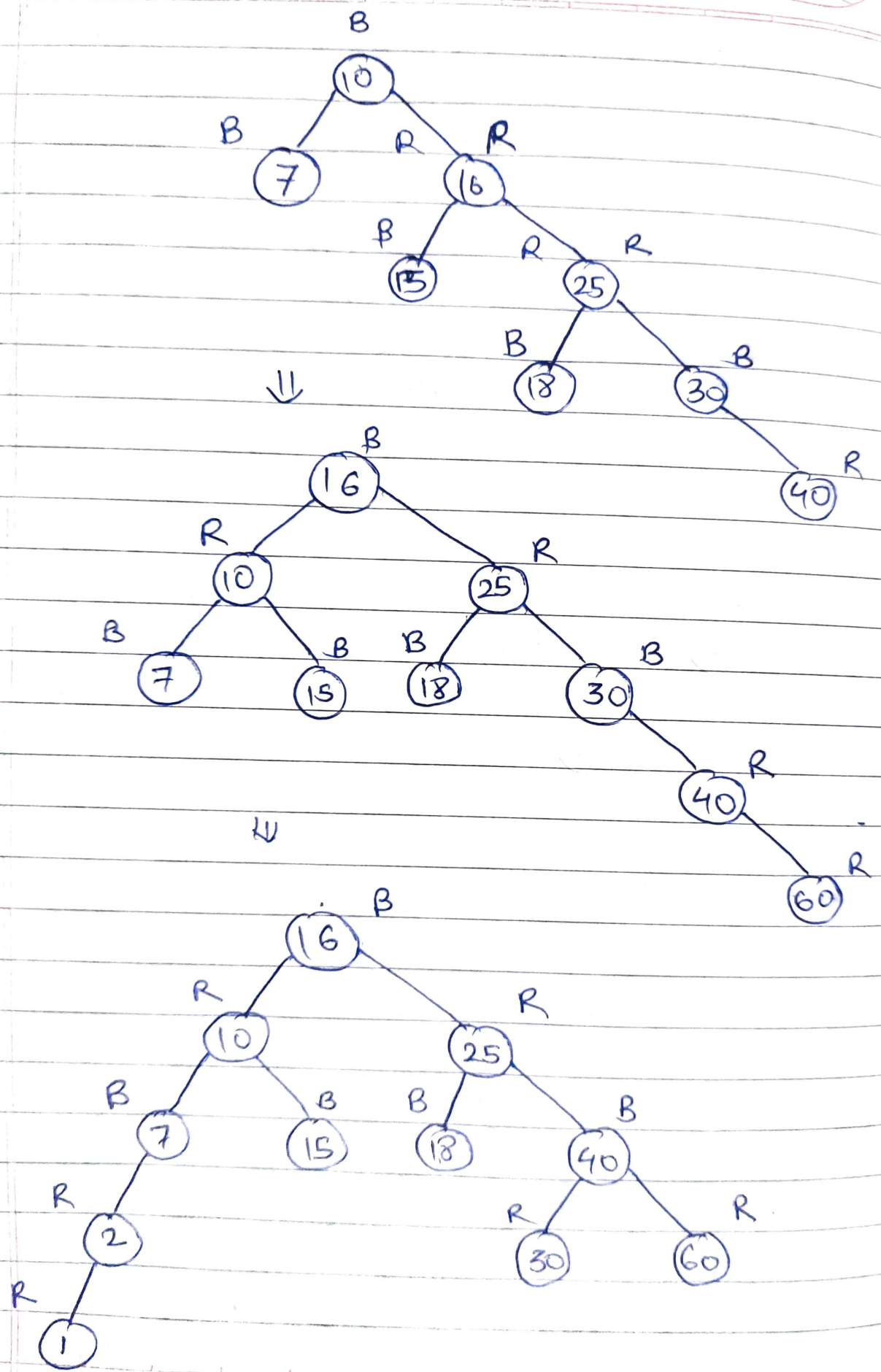
. Insert 10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70, to create a red black tree.

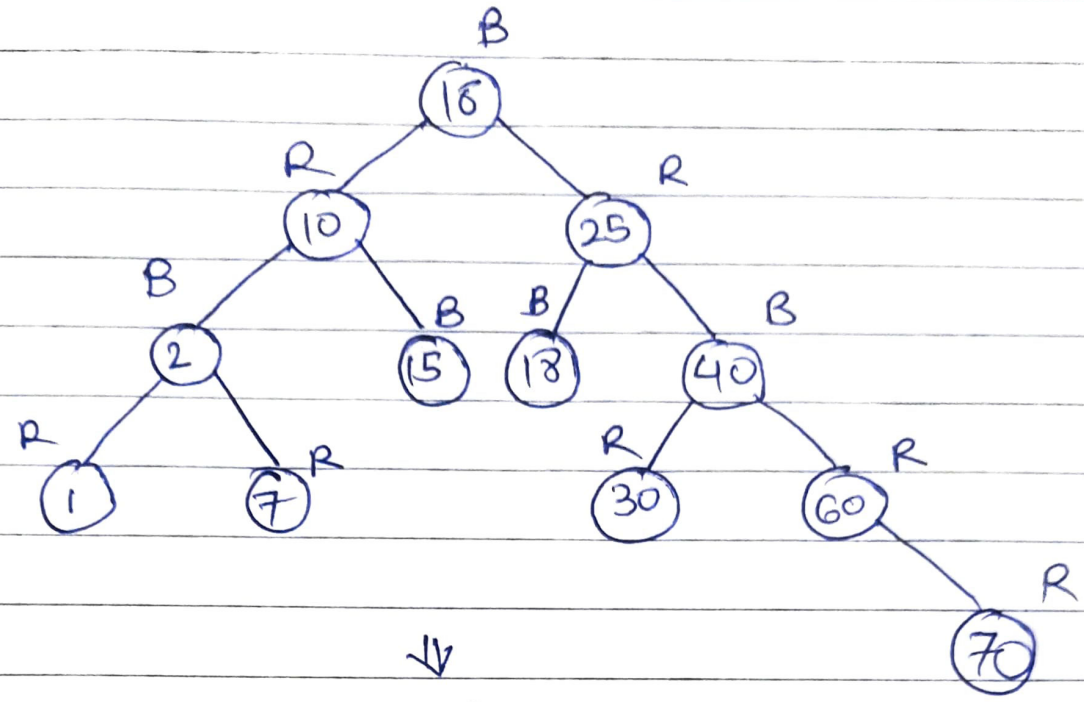
→

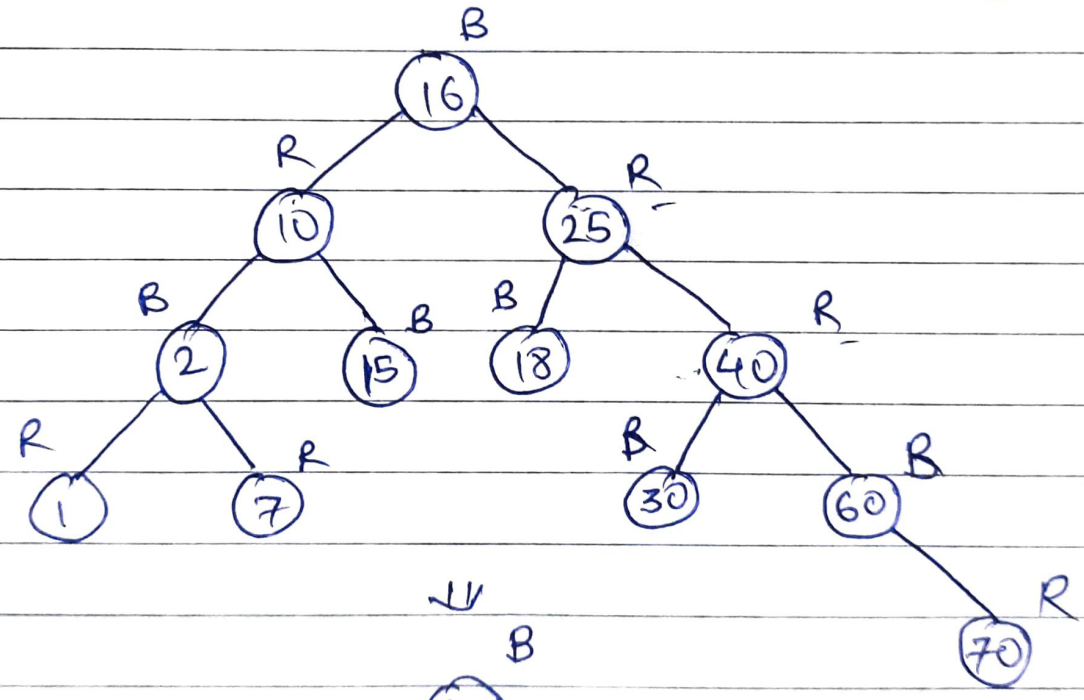


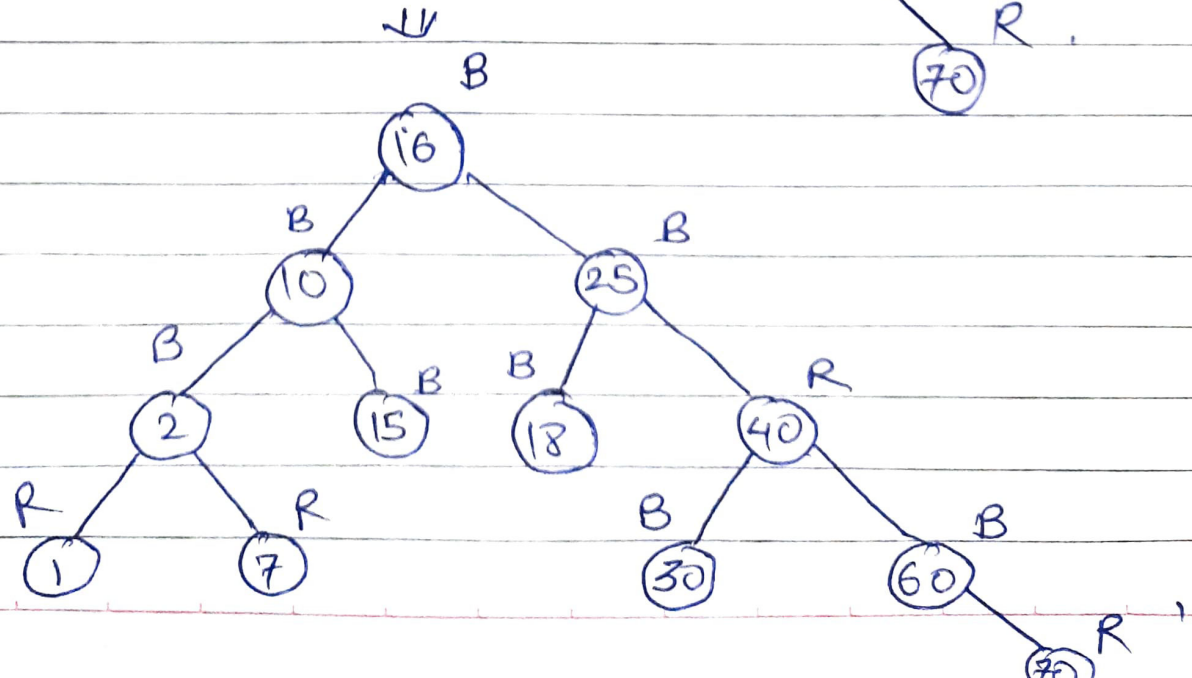





B
10
B
7
R
R
16
B
15
R
R
25
B
18
B
30
R
40
B
16
R
10
R
25
B
7
B
15
B
18
B
30
R
40
R
60
B
16
R
10
R
25
B
7
B
15
B
18
B
40
R
2
R
30
R
60
R
1


B
16
R
10
R
25
B
2
B
15
B
18
B
40
R
1
R
7
R
30
R
60
R
70


B
16
R
10
R
25
B
2
B
15
B
18
R
40
R
1
R
7
B
30
B
60
R
70


B
16
B
10
B
25
B
2
B
15
B
18
R
40
R
1
R
7
B
30
B
60
R

* Red-Black Tree Deletion:

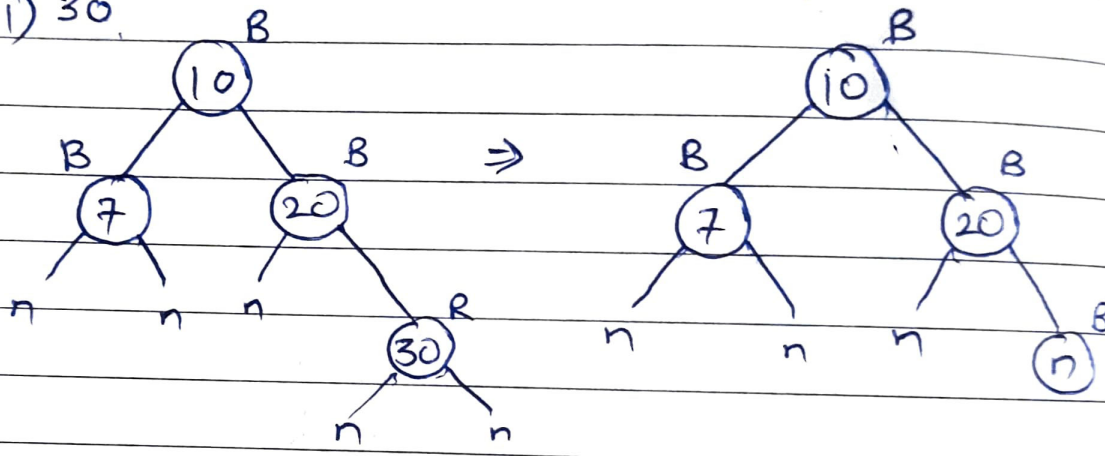
- Rules:

① Perform BST deletion.

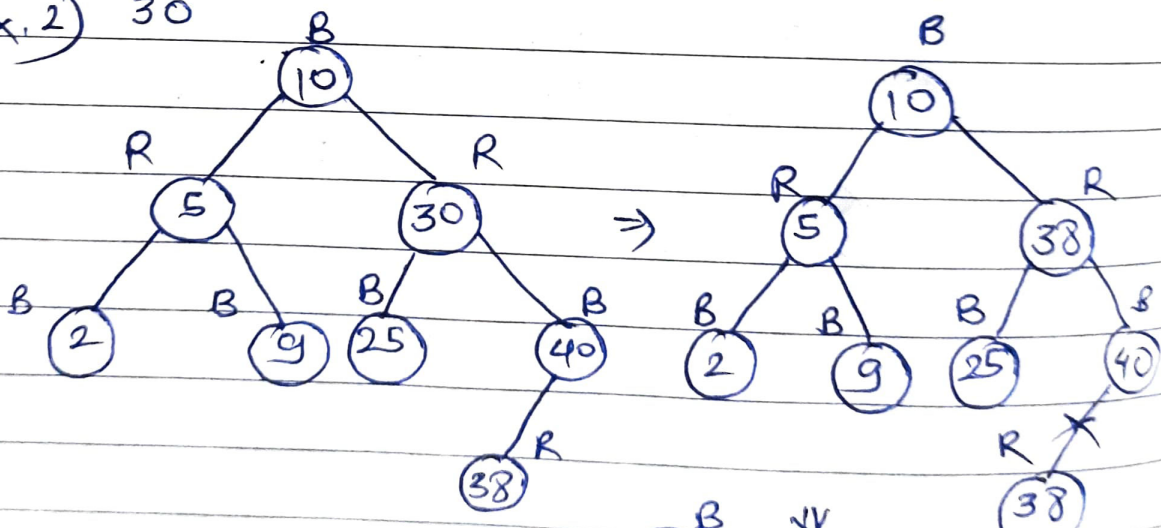
② Cases:

① If node to be deleted is red, just delete it (even if it is replaced).

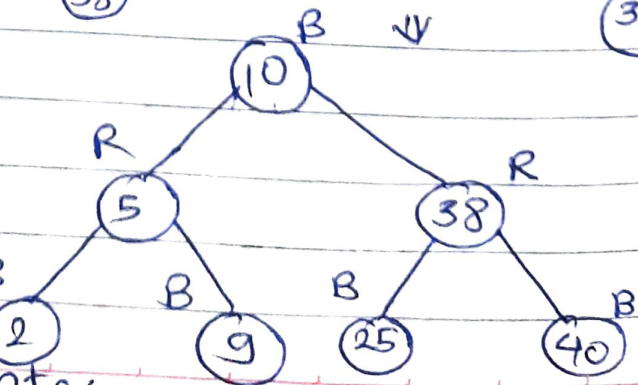
Ex.1) 30



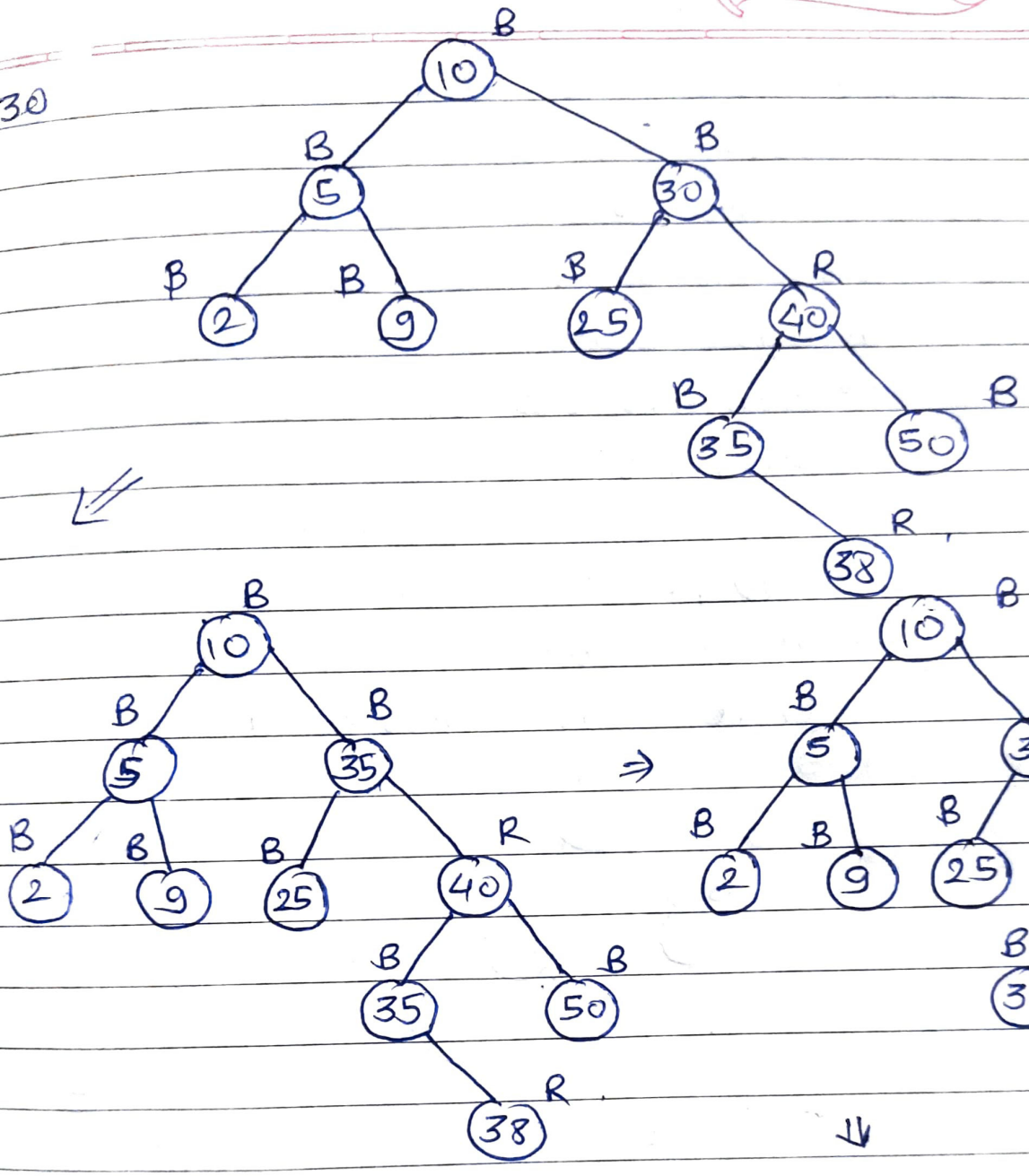
Ex.2) 30



Internal nodes always retain their original colour even after replacements.



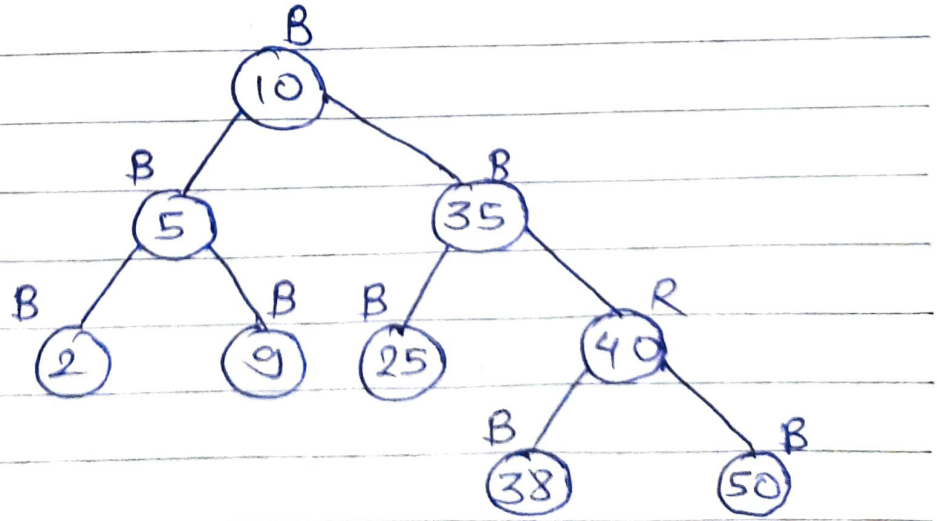
Ex 3) 30


B
10
B
5
B
30
B
2
B
9
B
25
R
40
B
35
B
50
R
38

B
10
B
5
B
35
B
2
B
9
B
25
R
40
B
35
B
50
R
38

⇒

B
10
B
5
B
35
B
2
B
9
B
25
R
40
B
38
B
50
R
38


B
10
B
5
B
35
B
2
B
9
B
25
R
40
B
38
B
50

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

③ If DB's sibling is black and both it's children are black. (it's children means children of sibling here)

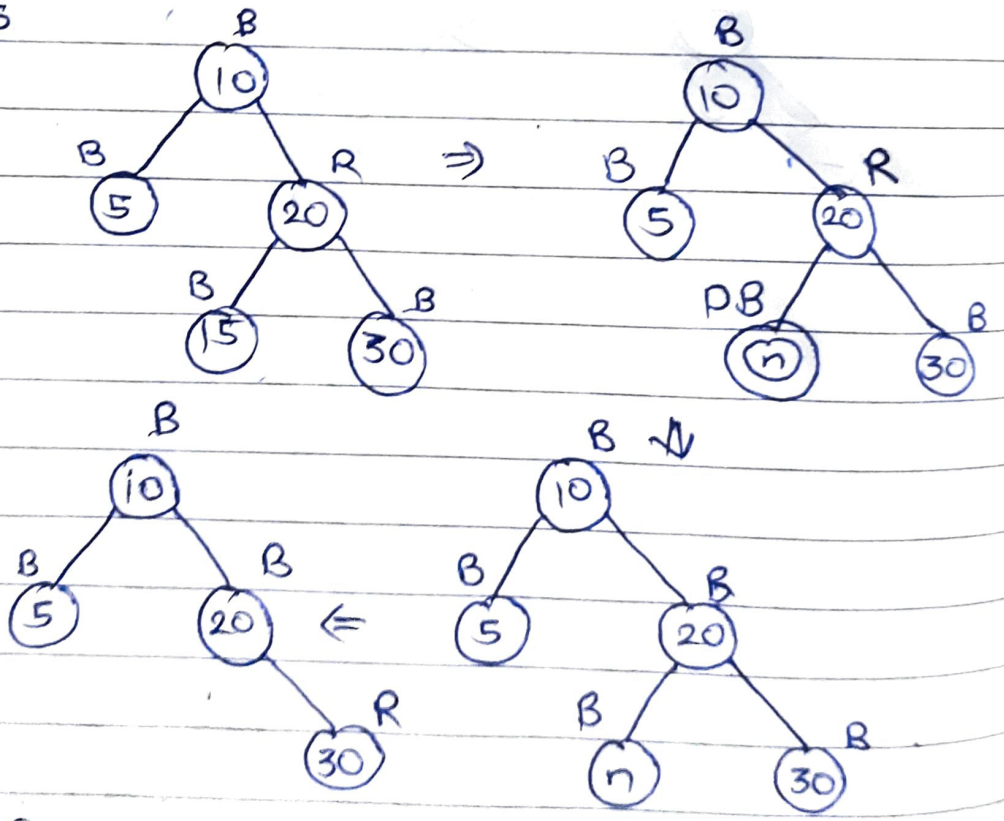
→ Remove DB and add black to it's parent (P)

ⓐ If P is Red, it becomes black.

ⓑ If P is Black, it becomes DB.

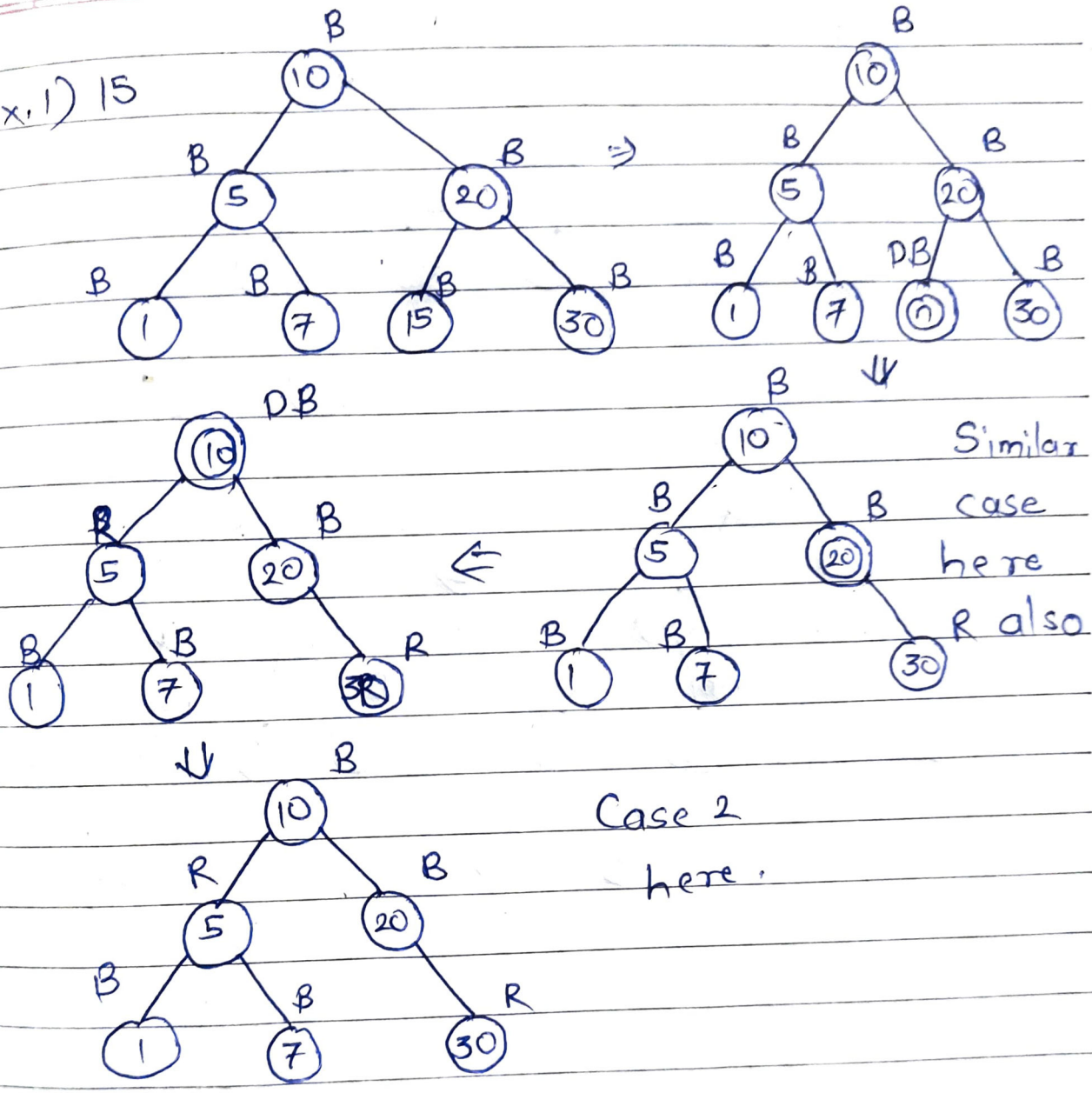
→ Make sibling of double black node (that was removed) Red.

Case ③ⓐ Ex.1): 15



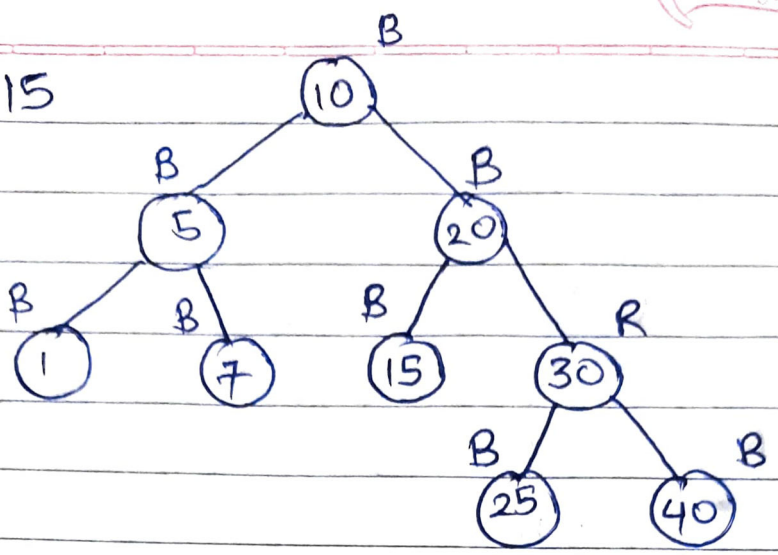
→ If still DB exists, apply other cases.

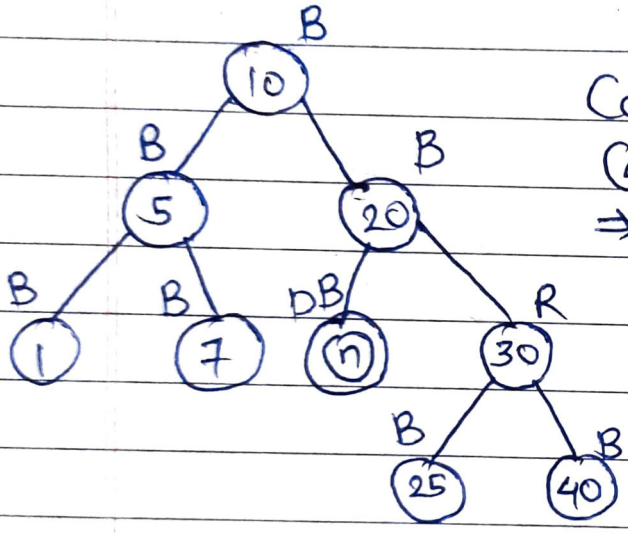
Case ③ⓑ Ex. 1) 15



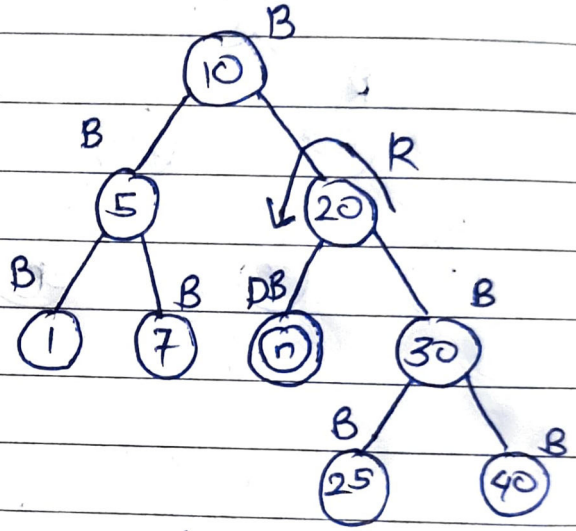
④ If DB's sibling is red
→ Swap colors of parent of DB and sibling of DB.
→ Rotate parent of DB in direction of DB.
→ Reapply cases.

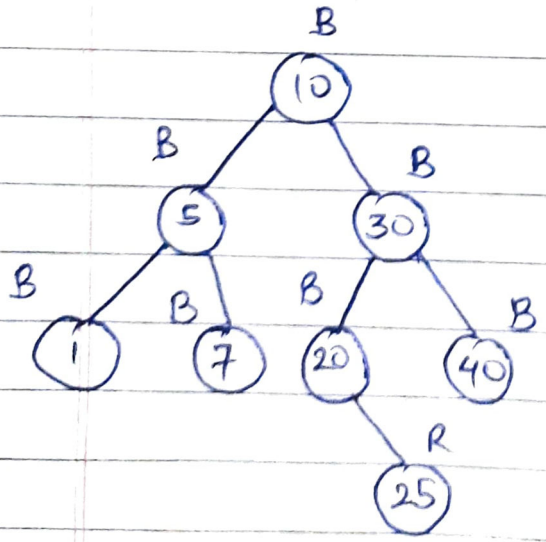
Ex.1) 15



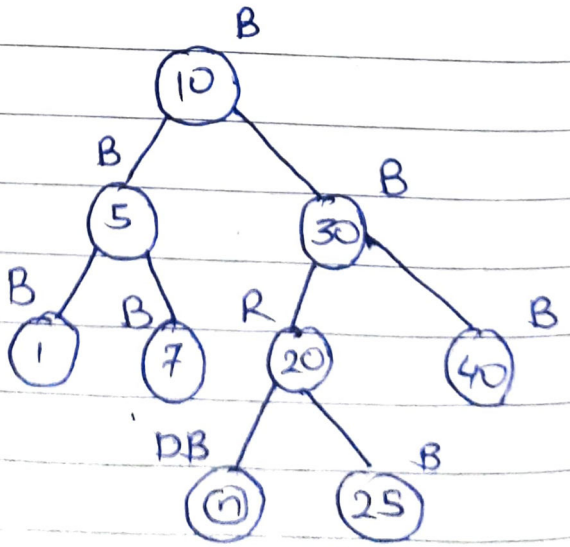


Case (4) ⇒





⇐



Case (3) (a)

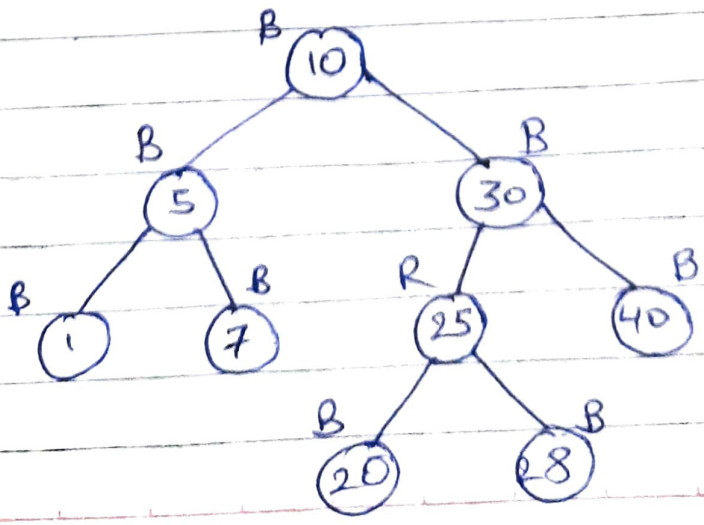
⑤ DB's sibling is black, child of DB's sibling who is far from DB is black, but child of DB's sibling who is near to DB is red.

→ Swap color of DB's sibling with child of that sibling near to DB
→ Rotate sibling of DB in opposite direction to DB (away from DB).
→ Apply case ⑥ (After case 5, we have to apply case ⑥ for sure).

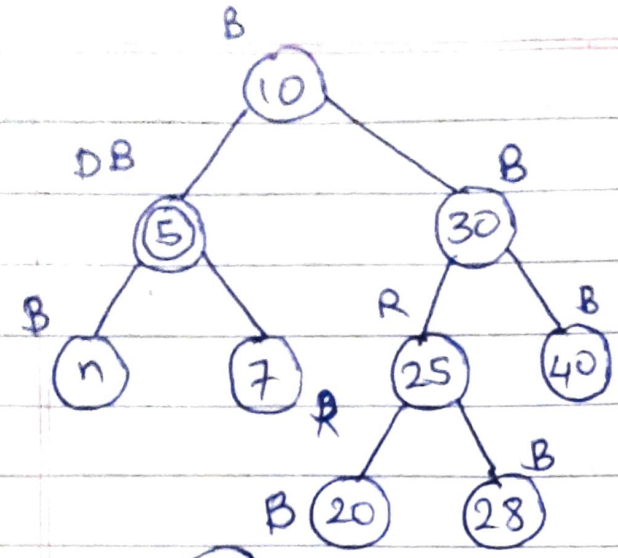
⑥ DB's sibling is black, child of DB's sibling who is far from DB is red and near child of DB's sibling is black.

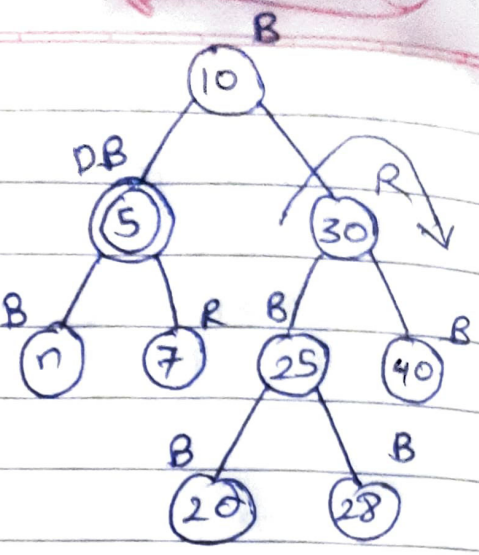
→ Swap color of Parent of DB with it's sibling of DB.
→ Rotate Parent of DB in direction of DB.
→ Remove DB.
→ Change color of red child to black.

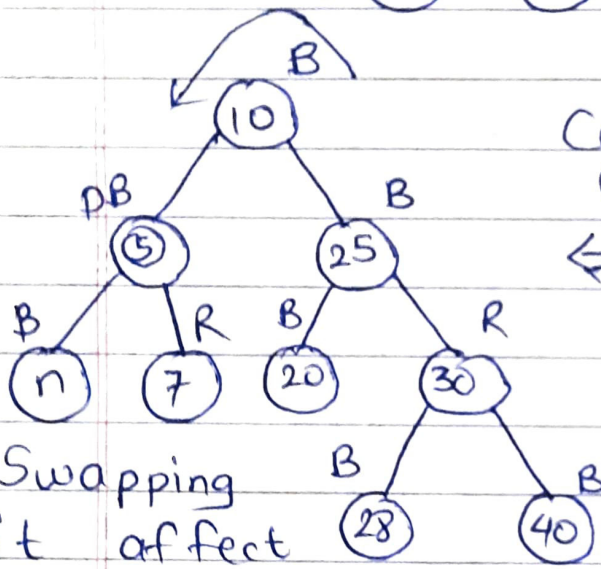
Ex.1) 1



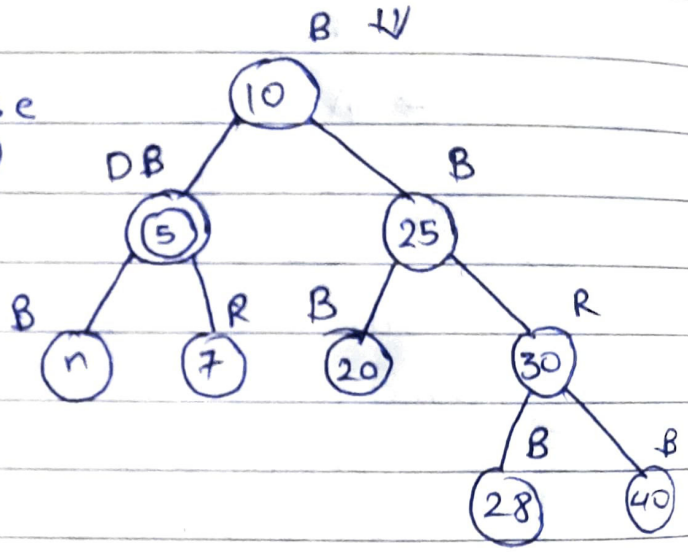
Case ③ ⑥.
⇒



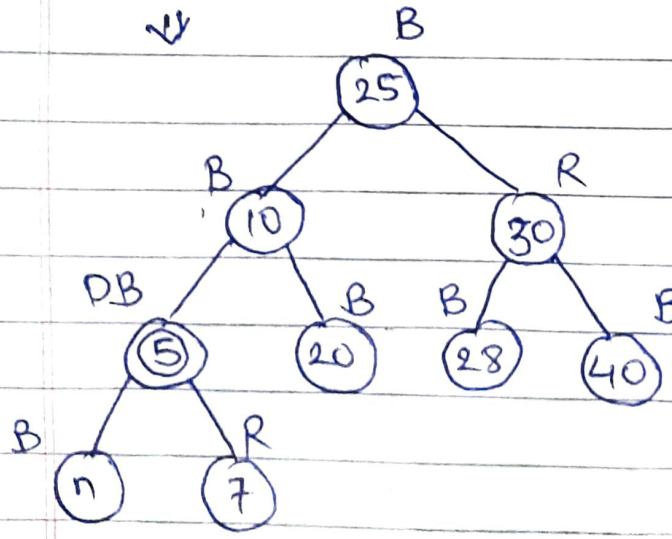


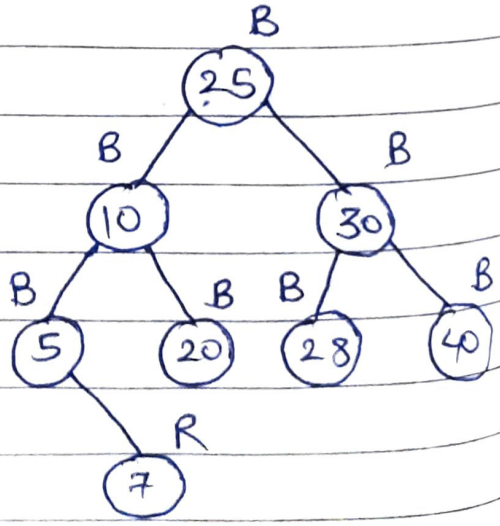


Case ⑥

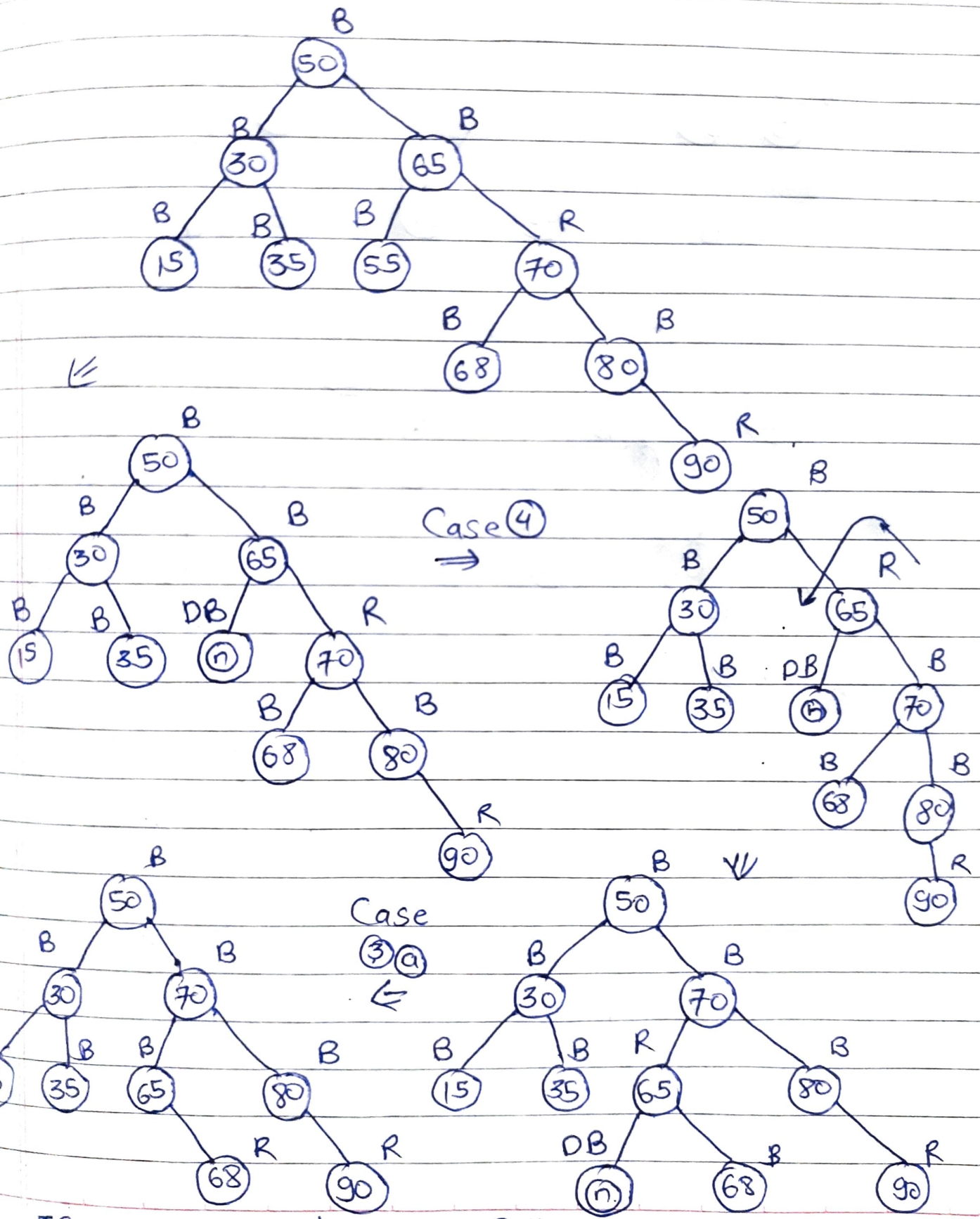


~~No~~ Swapping won't affect here as color of Parent of DB and sibling of DB is same i.e. black here.

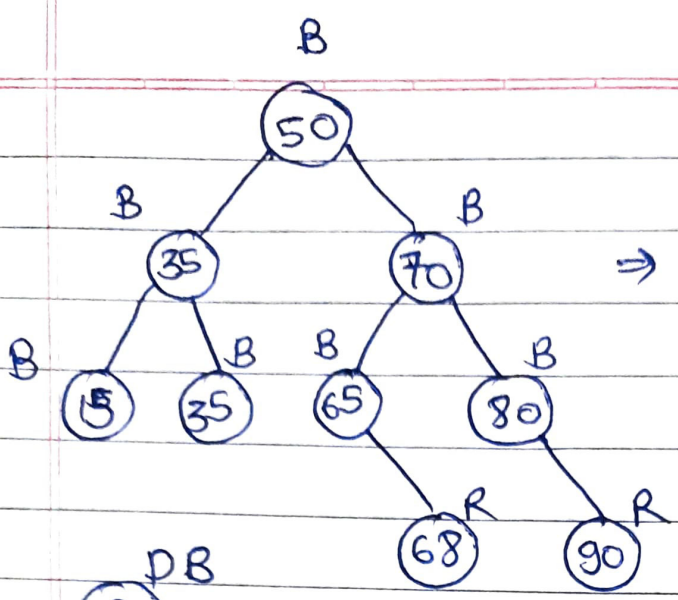




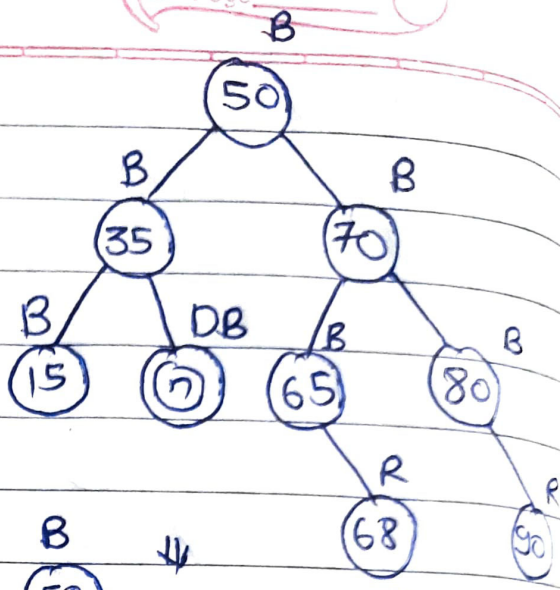
5) Delete 55, 30, 90, 80, 50, 35, 15, 65, 68 from the tree below.



55 is deleted, successfully.

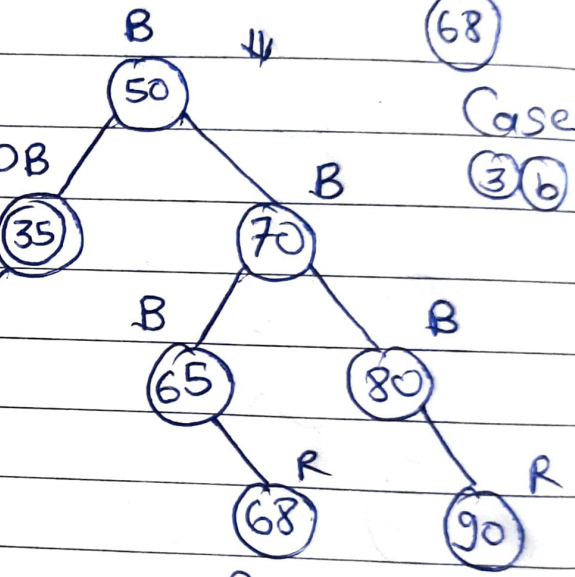


⇒

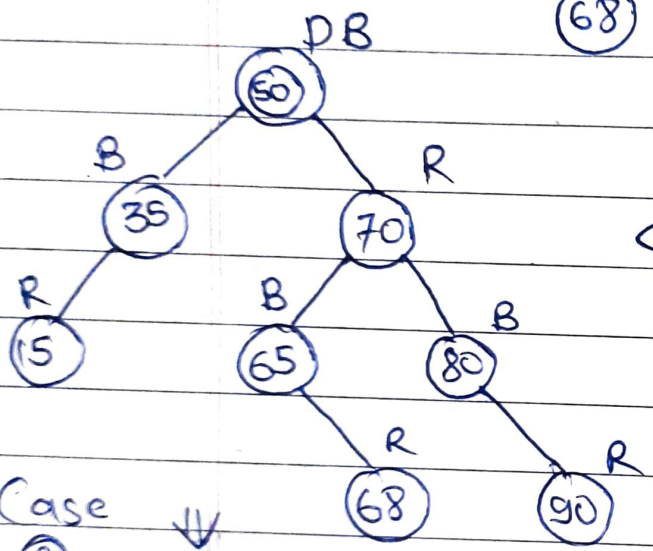


Case 3b

⇓

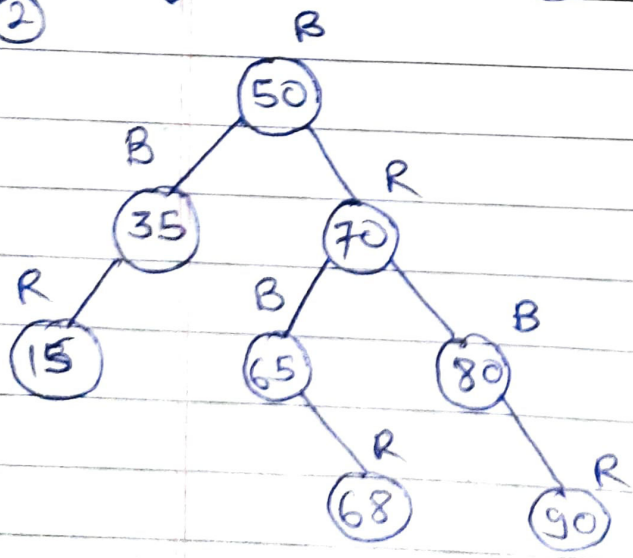


⇐



Case 2

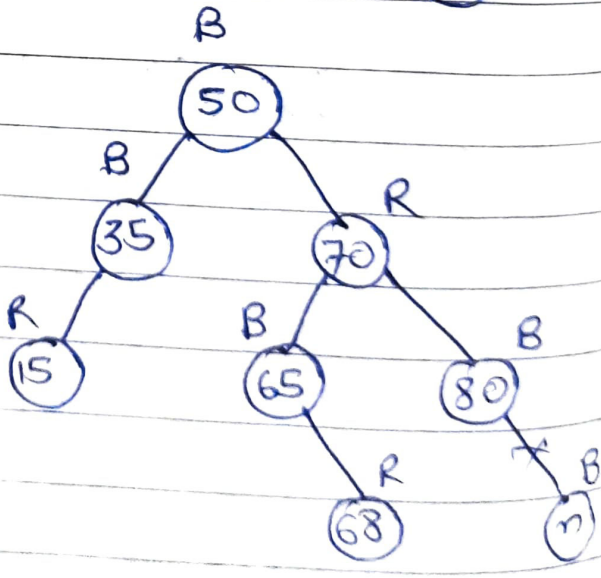
⇓



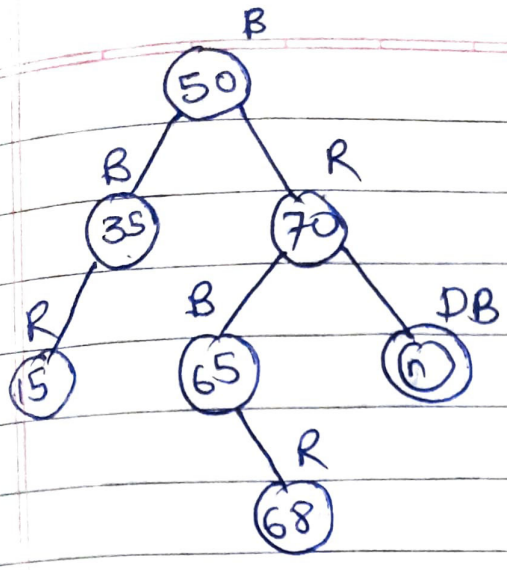
30 is deleted, successfully.

Case 1

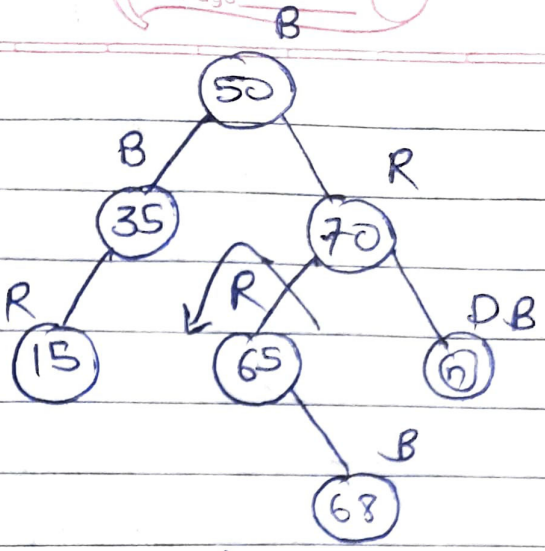
⇒

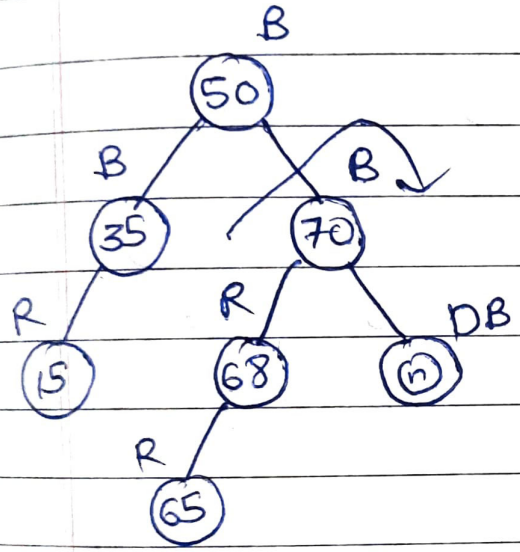


90 is deleted, successfully.

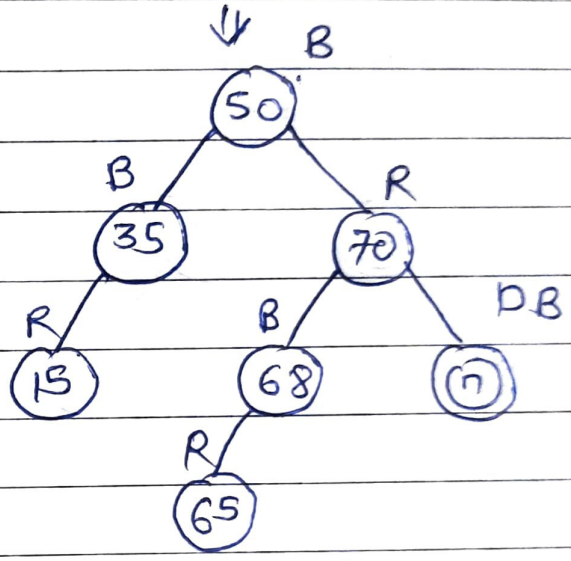


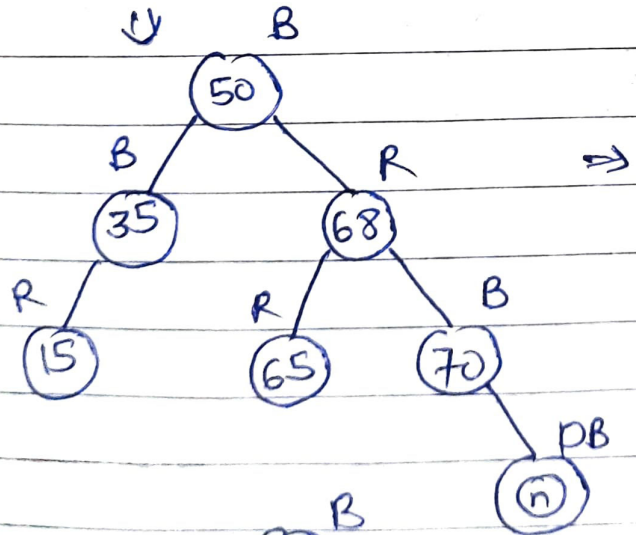
Case (5) ⇒



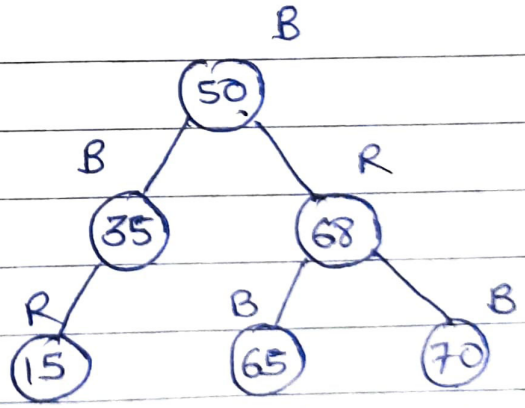


Case (6) ⇐





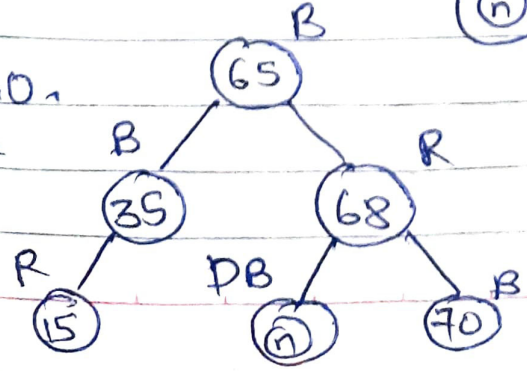
⇒



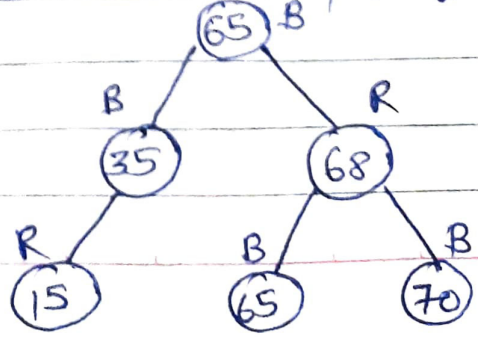
80 is deleted, successfully.

P.T.O.

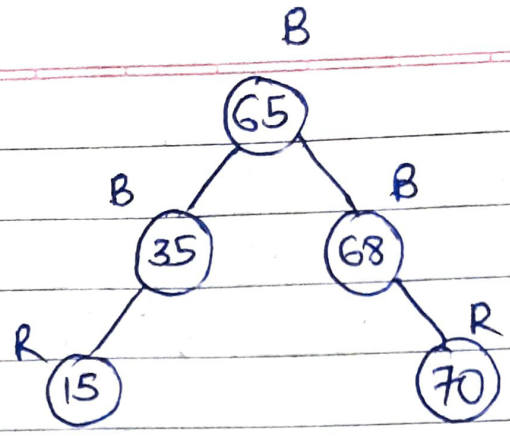
Case (3)(a) ⇐



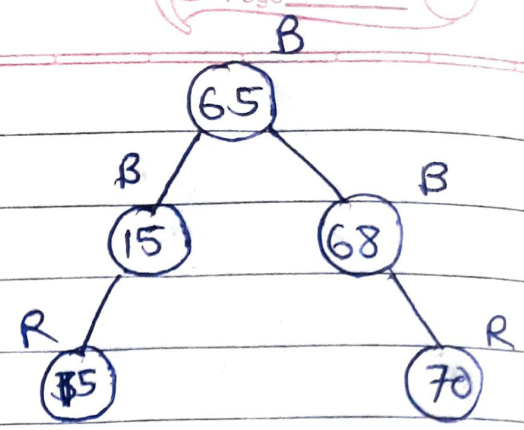
⇐



Case ⇒ ③ (a)

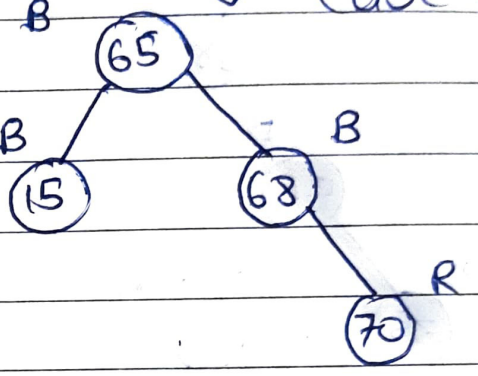


⇒



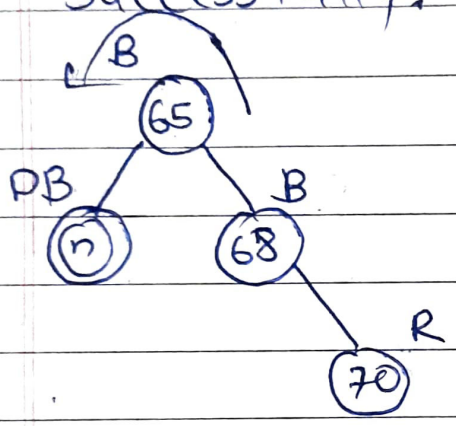
50 is deleted, successfully.

⇓ Case ①



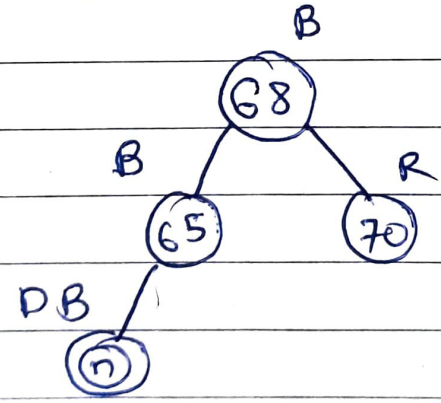
35 is deleted, successfully.

⇐ ~~Case ⑥~~

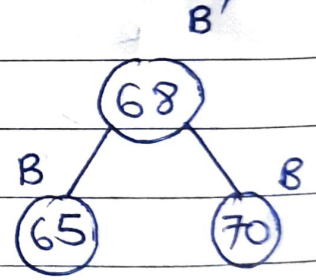


Case ⑥ ⇓

No change even after swapping colors of DB's parent & sibling ⇒

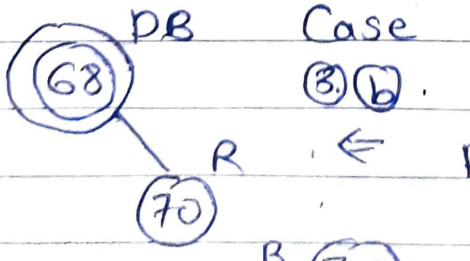


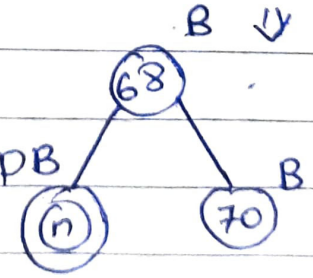
⇒



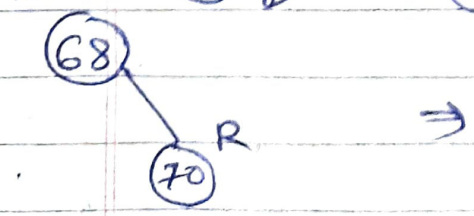
15 is deleted, successfully.

Case ③ (b) ⇐

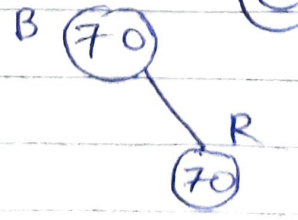




Case ② ⇙



⇒



⇒ Case ①

65 is deleted successfully.

* Splay Trees:
- Self adjusted BST
- Operations: search, insert, delete, splaying.
- Roughly balanced trees.
- Splaying means rearranging the tree so that the element on which we are going to perform the operation now becomes root of the tree.

- Rotations:

→ Zig rotations:

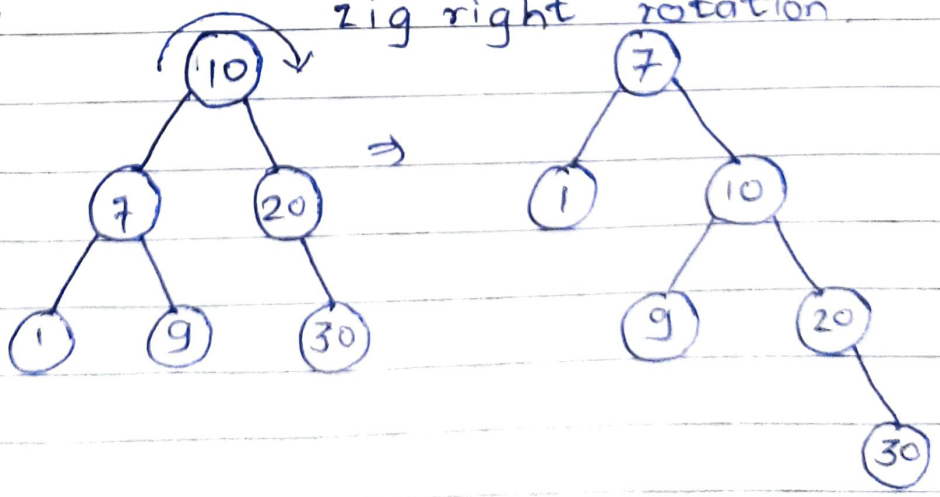
Case ①: Search item is root node.

→ Return the root node, no need to rotate.

Case ②: Search item is child of root node

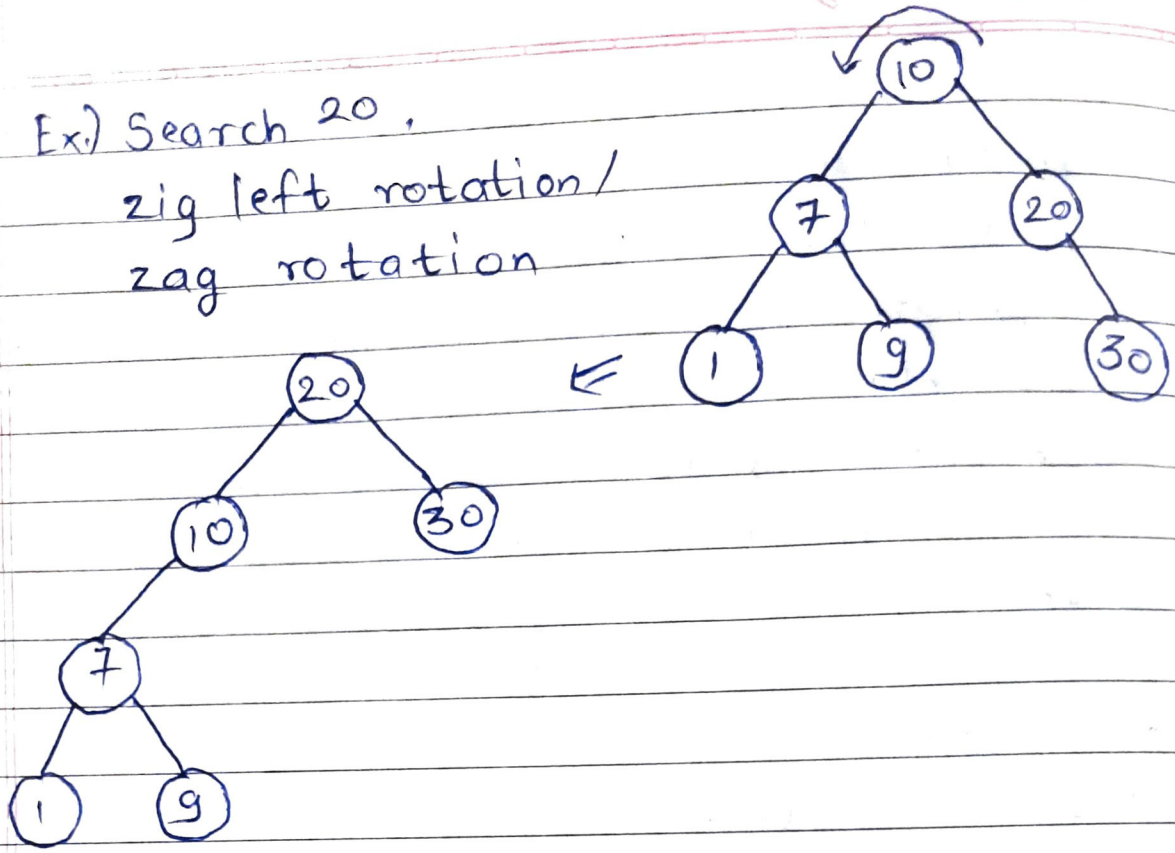
→ Left child (zig right) (zig) } rotation
→ Right child (zig left) (zag) }

Ex) Search 7.



Ex.) Search 20.

zig left rotation/
zag rotation

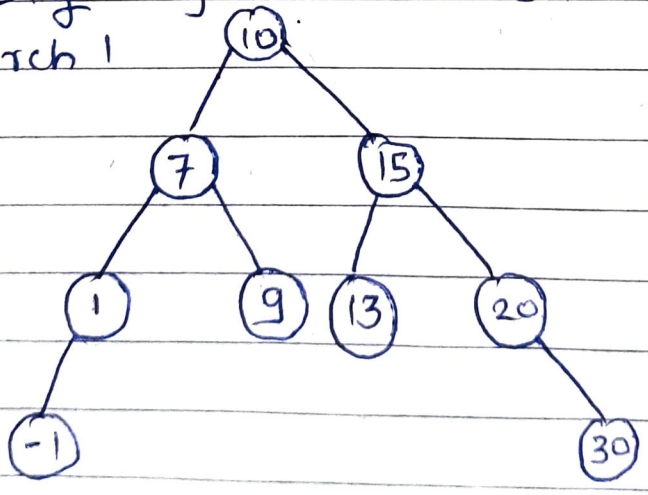


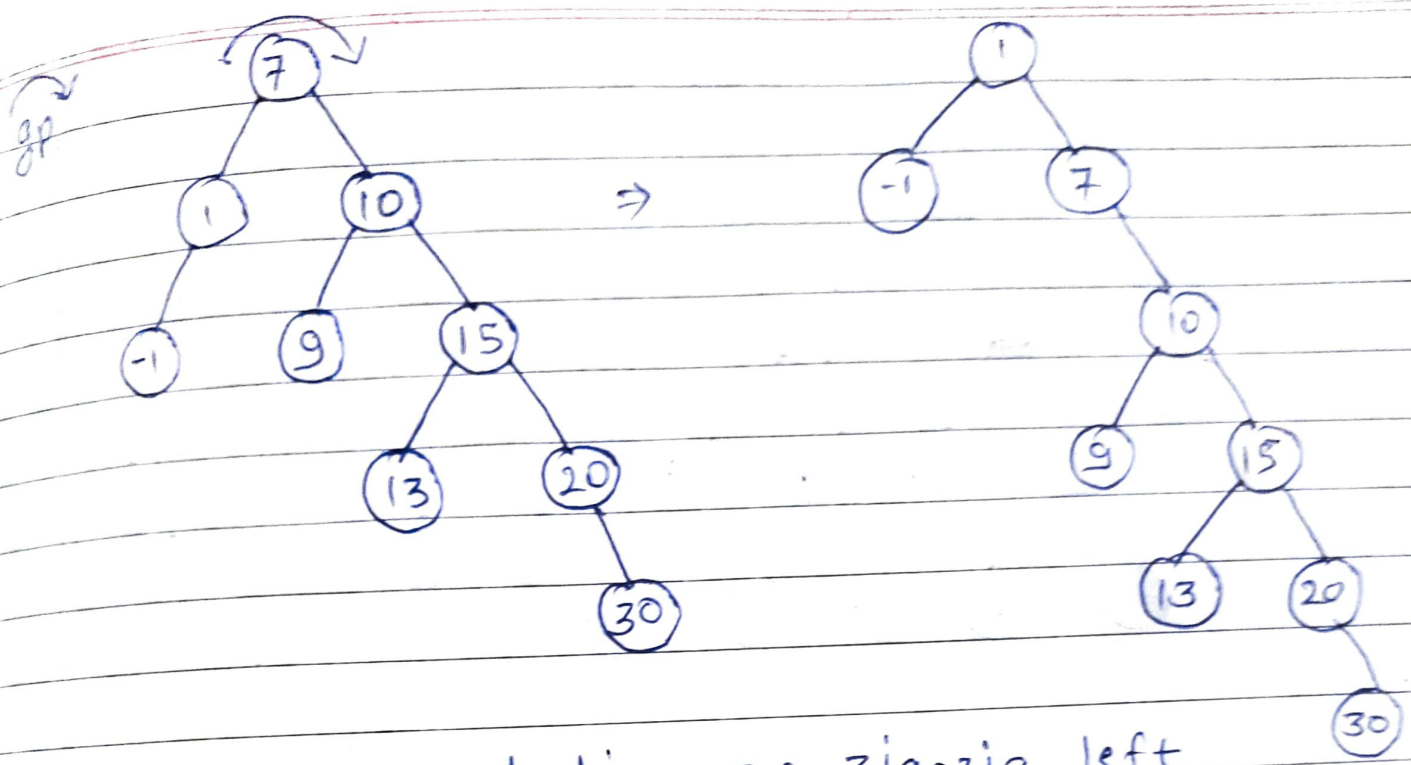
Case ③: Search item is grand child of root node.

→ ① Zig-Zig rotation:

① ⓐ Zig-Zig right rotation

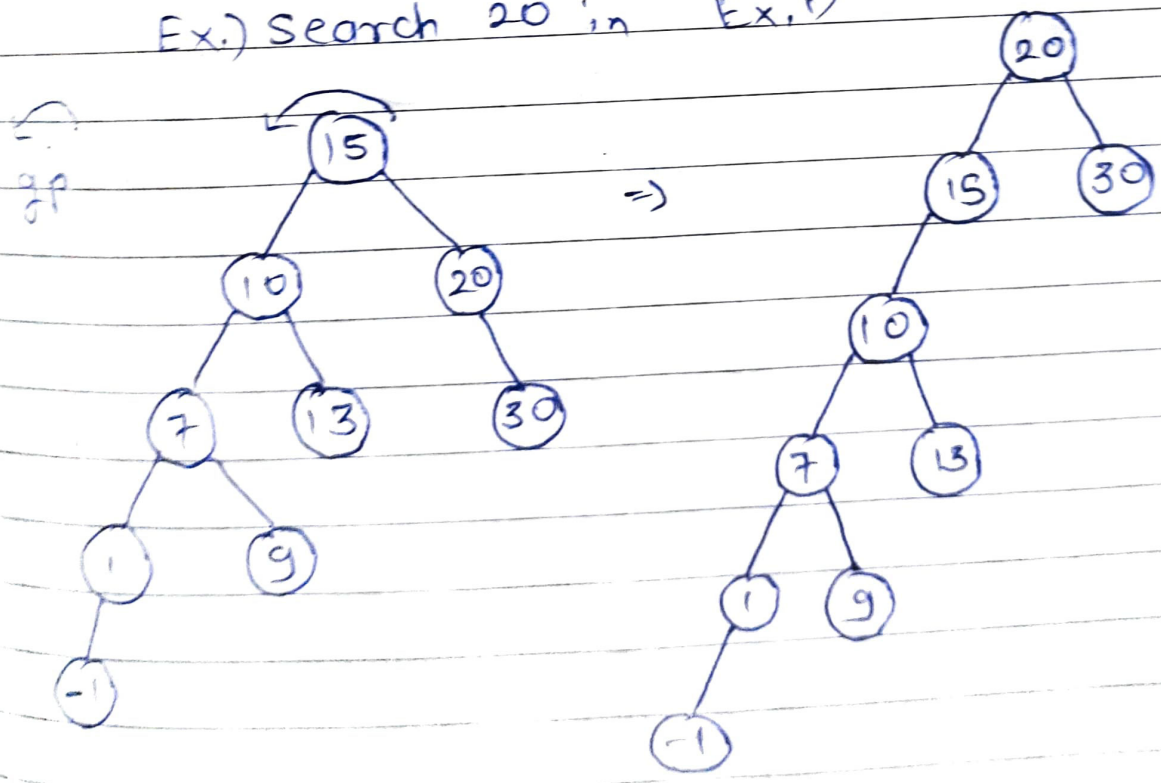
Ex.) Search 1





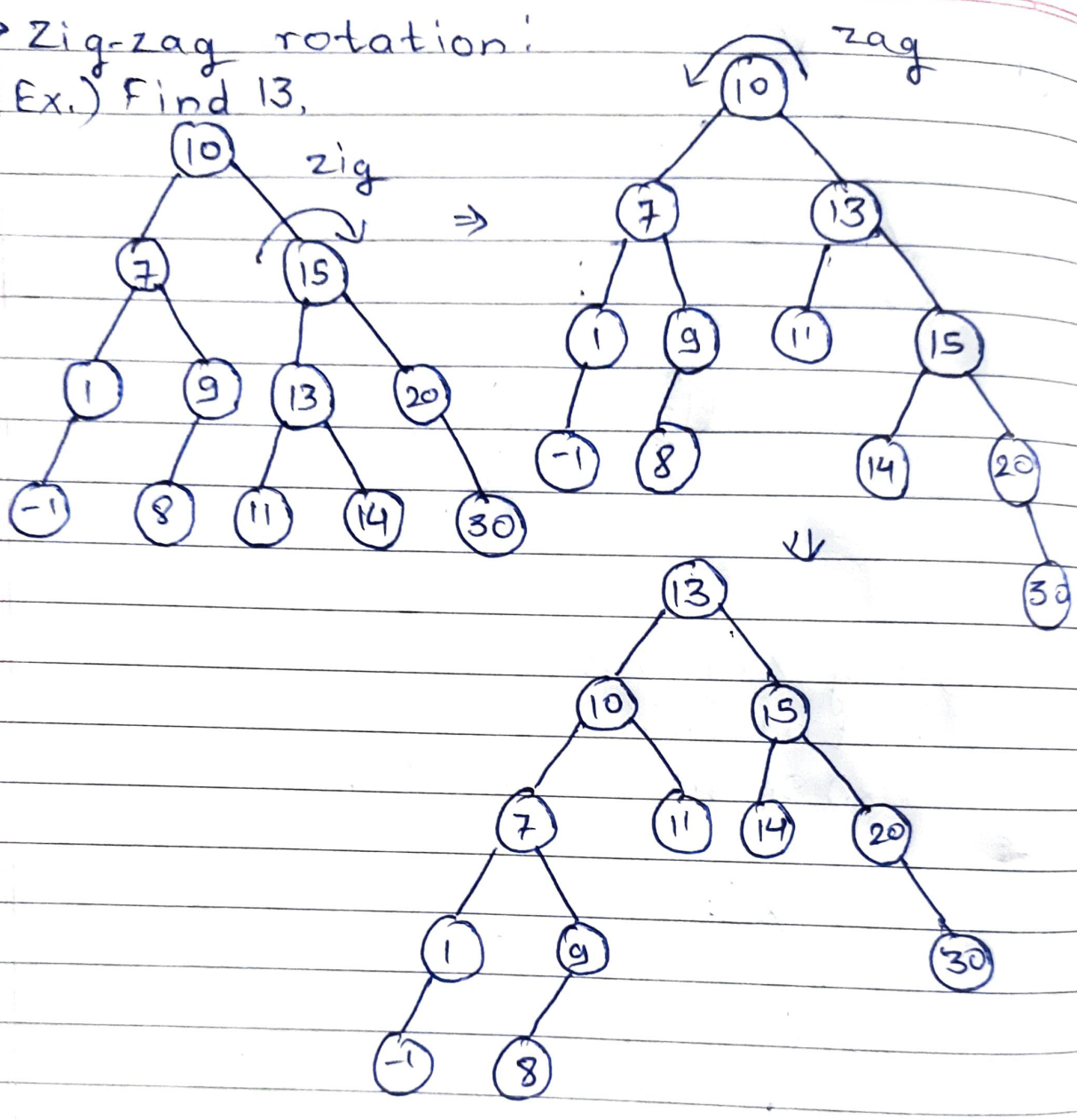
(3) (b) Zag-Zag rotation or zig-zig left rotation:

Ex.) Search 20 in Ex.1)



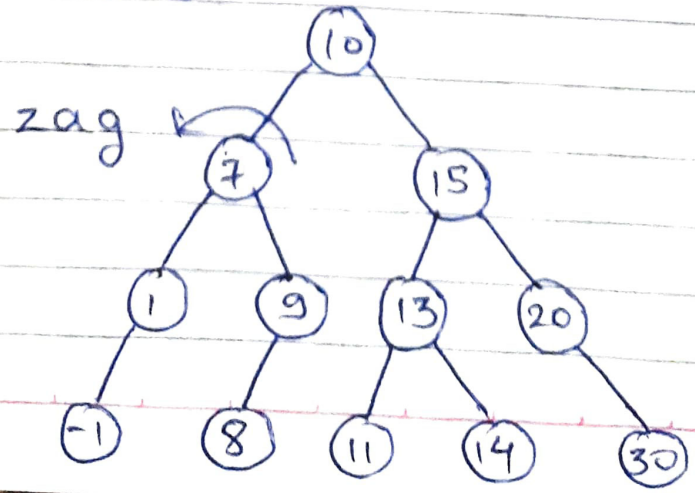
③→ Zig-zag rotation:

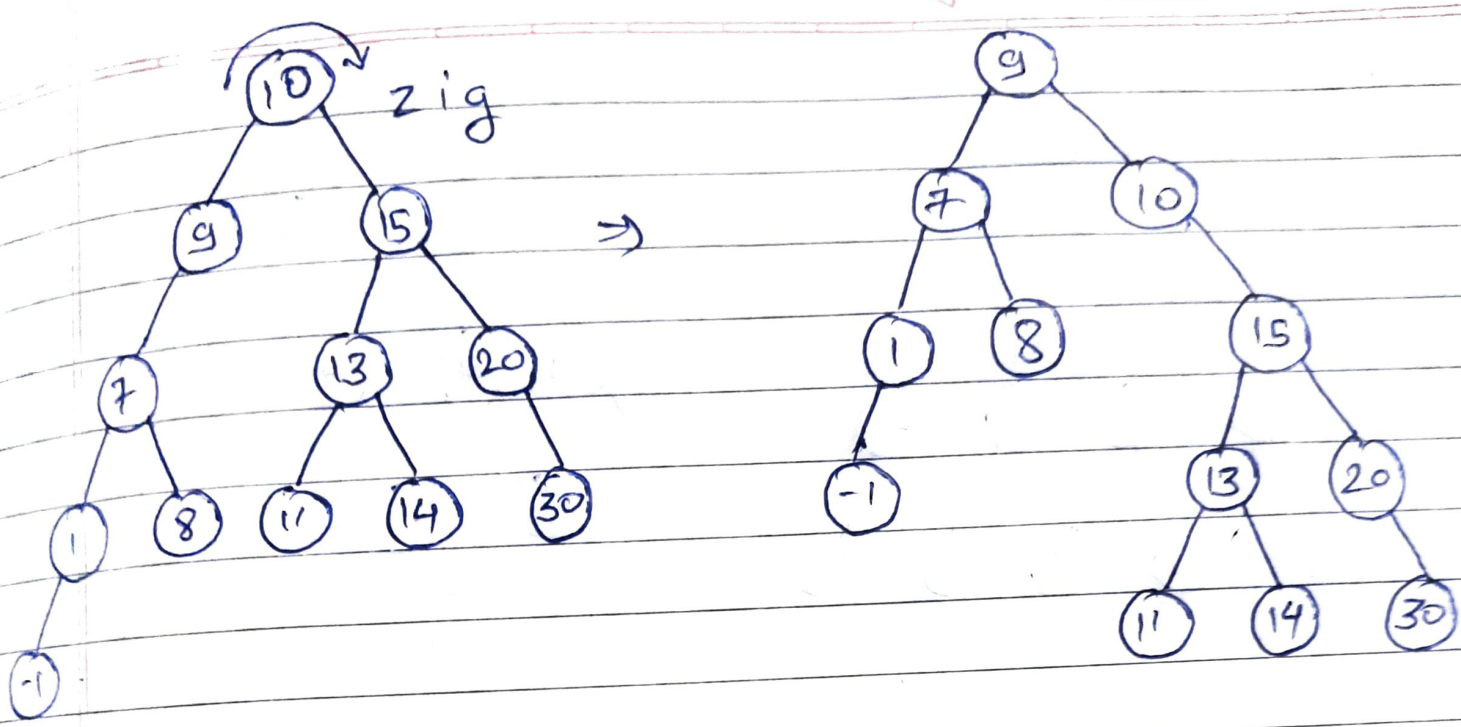
Ex.) Find 13.



④→ zag-zig rotation:

Ex.) Search 9.



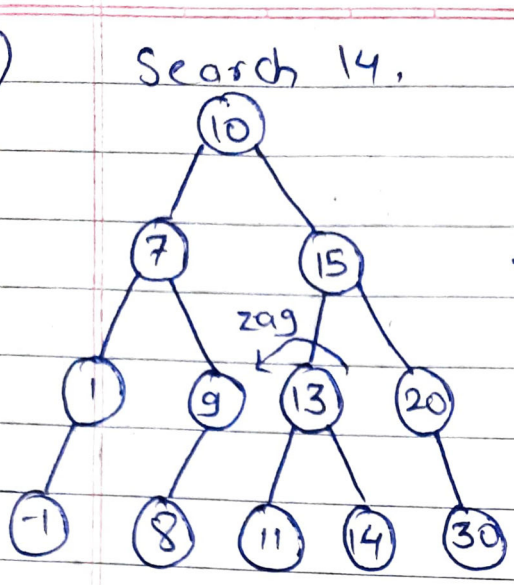


- Advantages of Splay trees:
  - → No extra info stored.
  - → Fastest type of BST for memory.
  - → Easy to implement.
  - → Better performance because frequently accessed nodes will move nearer to root.

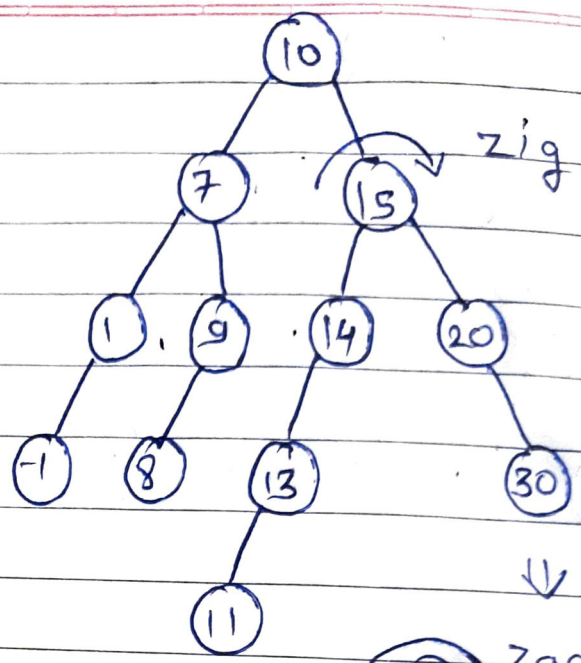
- Disadvantage of Splay tree:
  - → Height of Splay tree can be linear.
  - → Performs unnecessary splay tr when a node is only being read.

- Application of Splay tree:
  - → Used in caches as most frequently used data is stored in caches.
  - → Used in a network router.

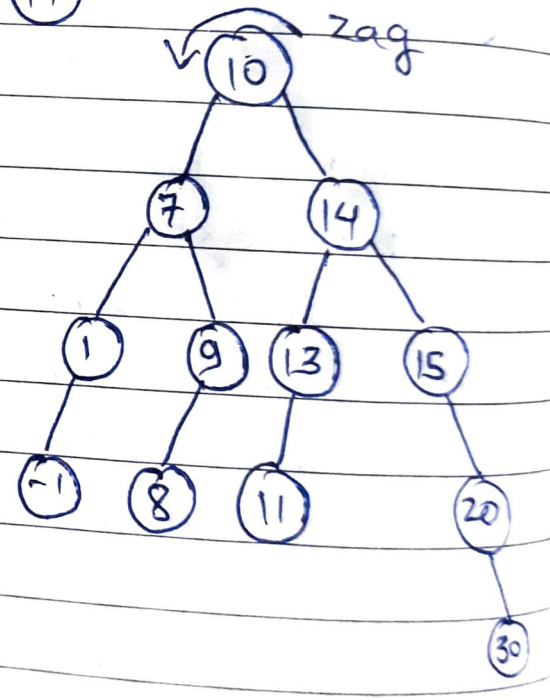
Q.) Search 14.


10
7
15
zag
1
9
13
20
-1
8
11
14
30

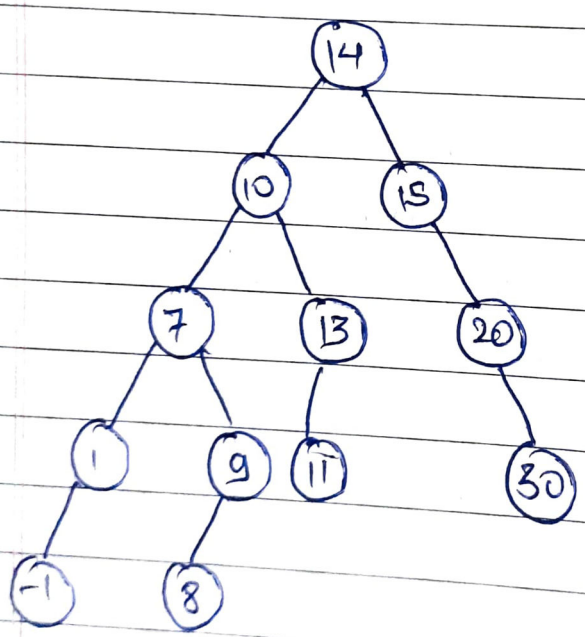
⇒


10
zig
7
15
1
9
14
20
-1
8
13
30
11

⇓


zag
10
7
14
1
9
13
15
-1
8
11
20
30

⇐

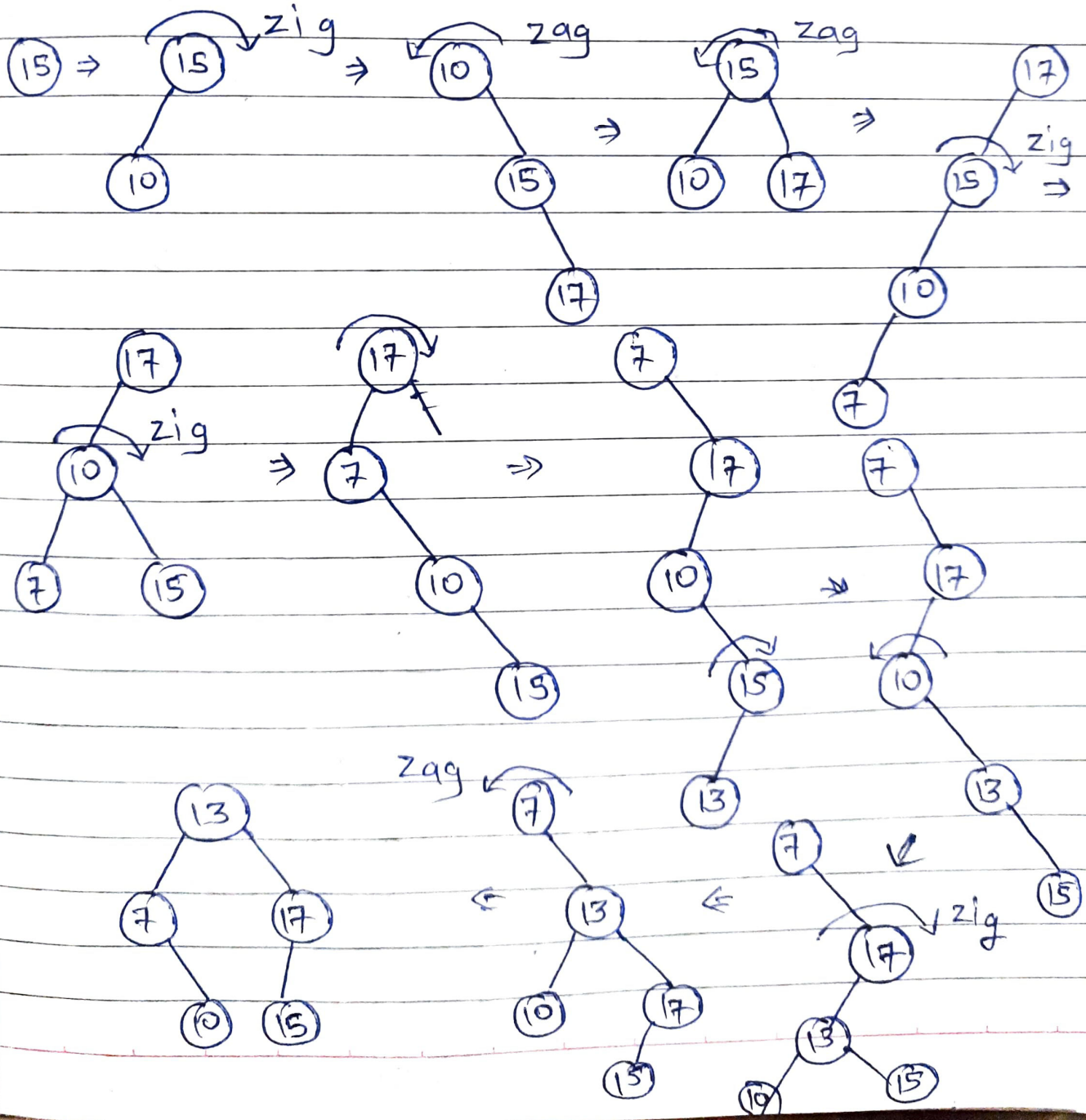

14
10
15
7
13
20
1
9
11
30
-1
8

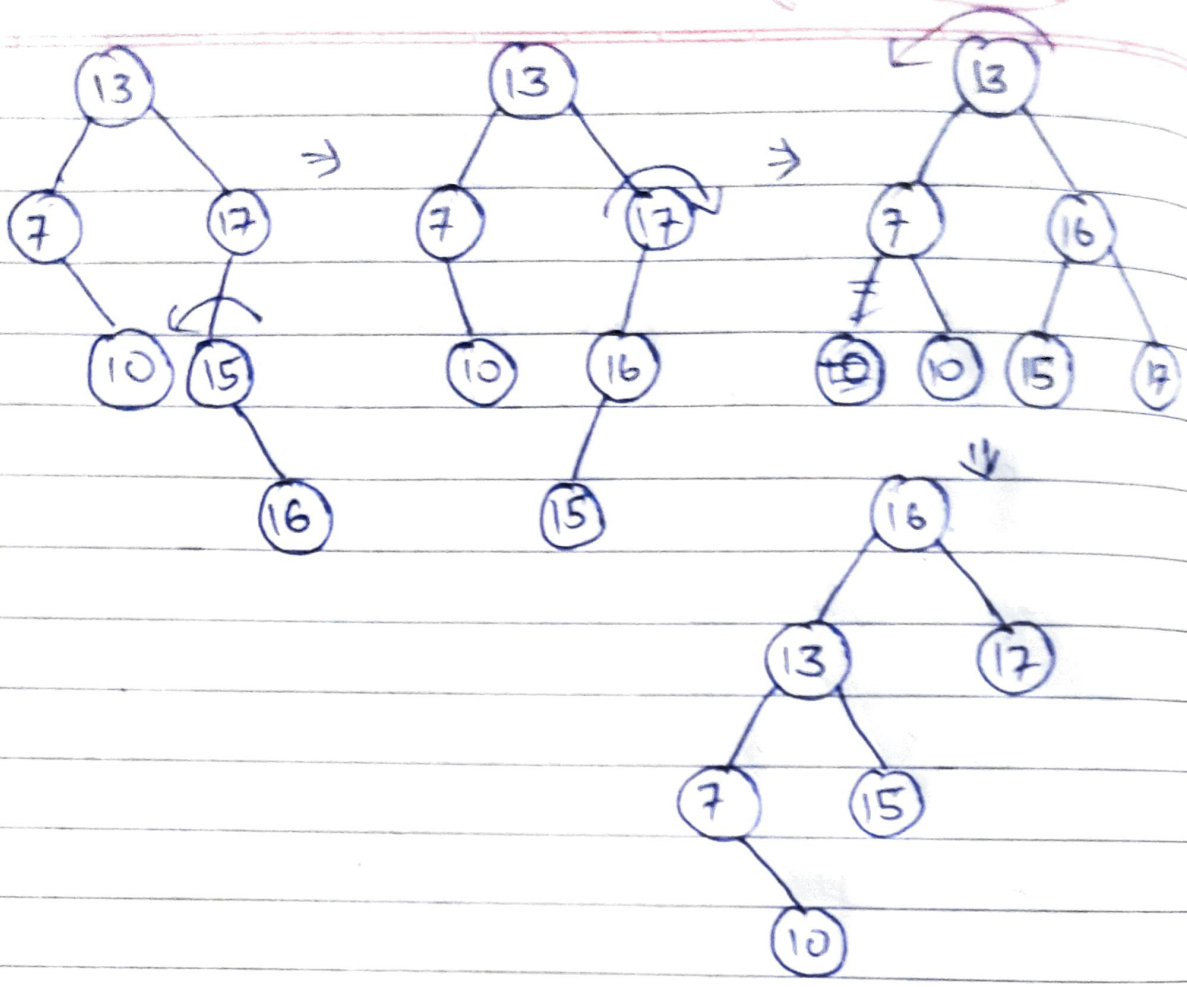
* Insertion in Splay Tree :

- After inserting a node according to the rules of insertion in BST, we have to do splaying that is the recently inserted node should be the root of tree.

Ex.) 15, 10, 17, 7, 13, 16

→



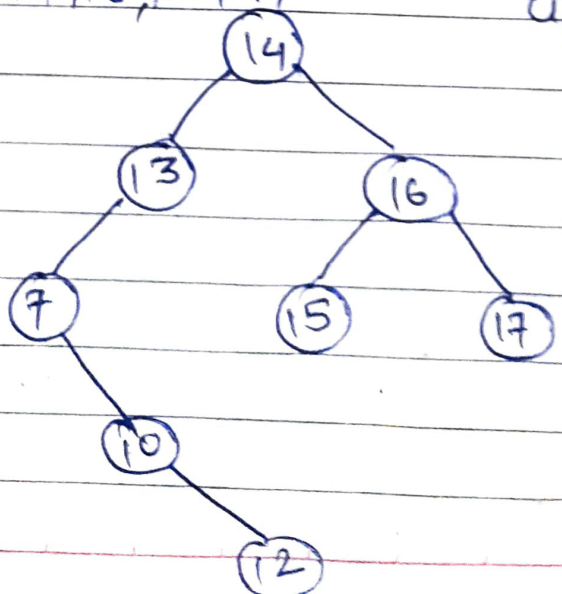


- Algorithm skipped.

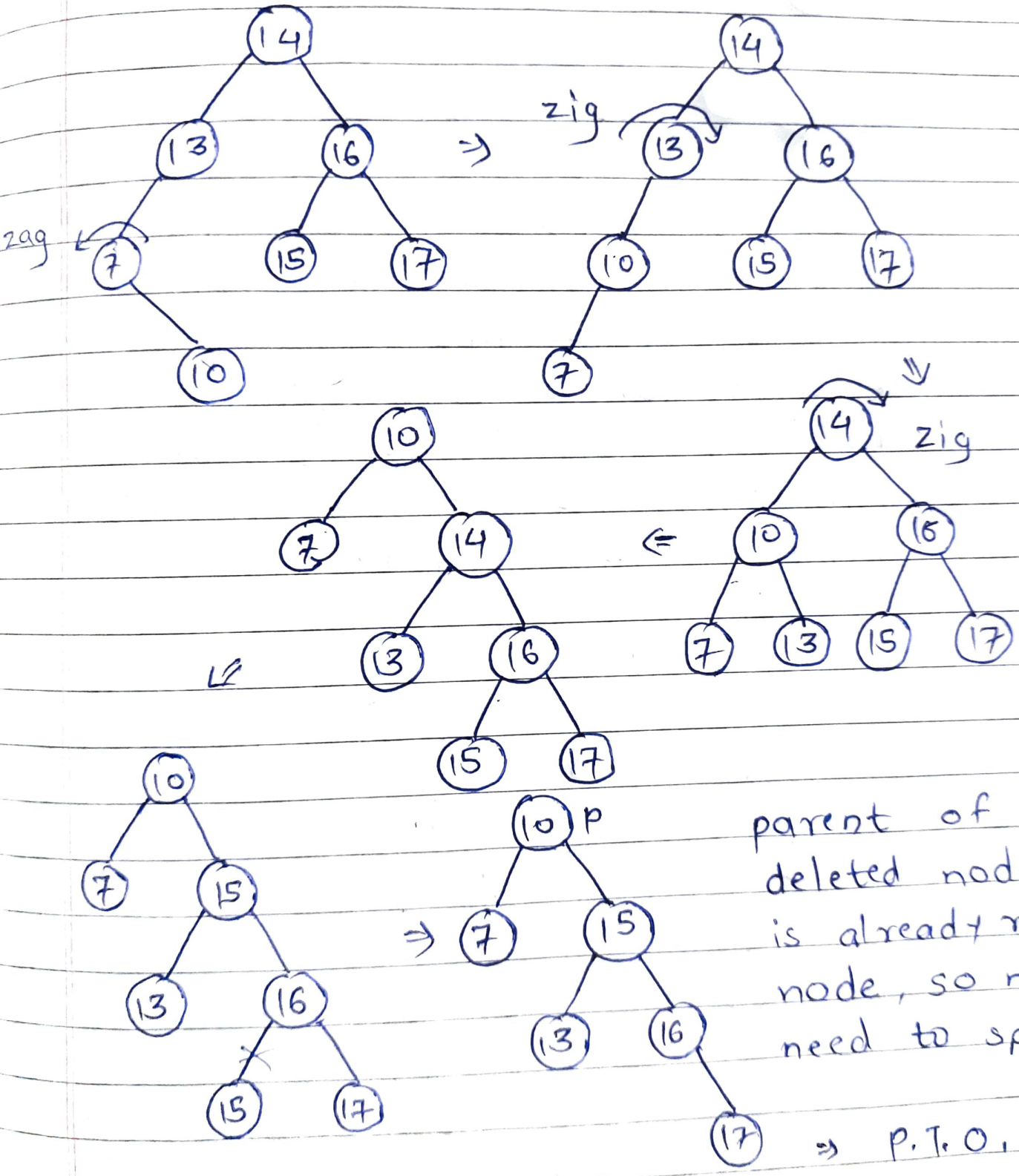
V-69.

* Deletion in splay trees: (Using bottom-up splaying).

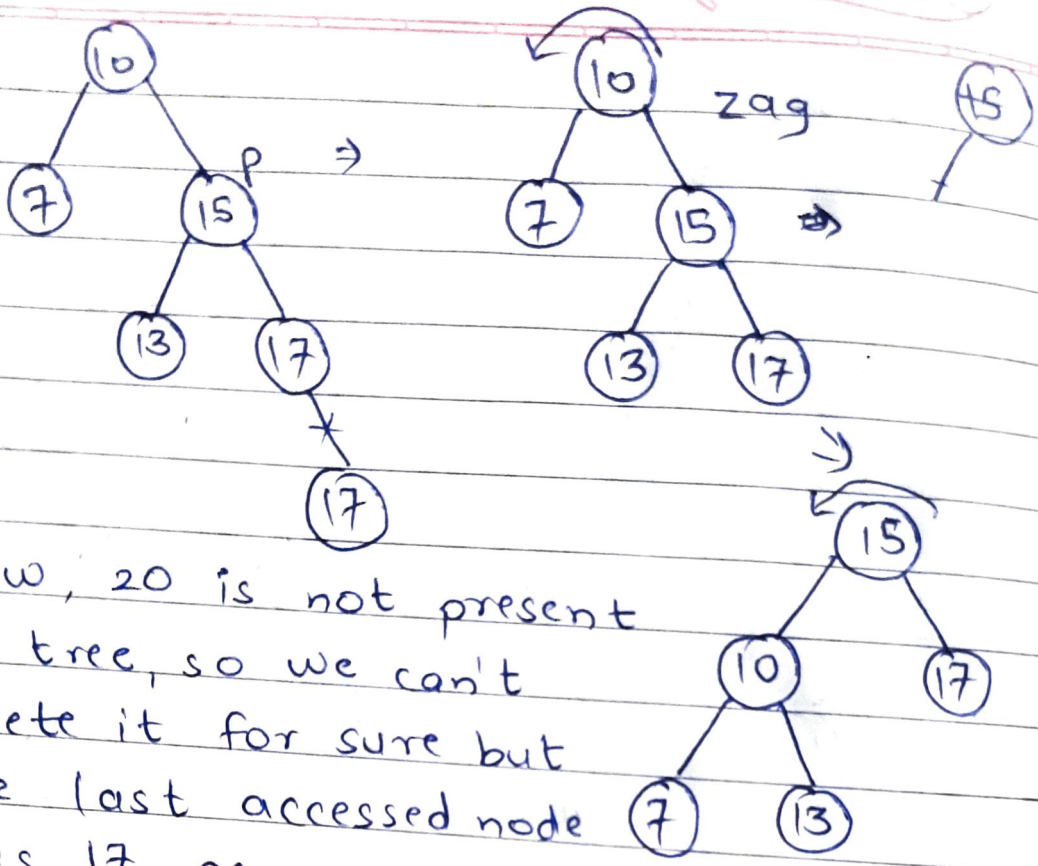
Ex.) Delete 12, 14, 16, 20, 17



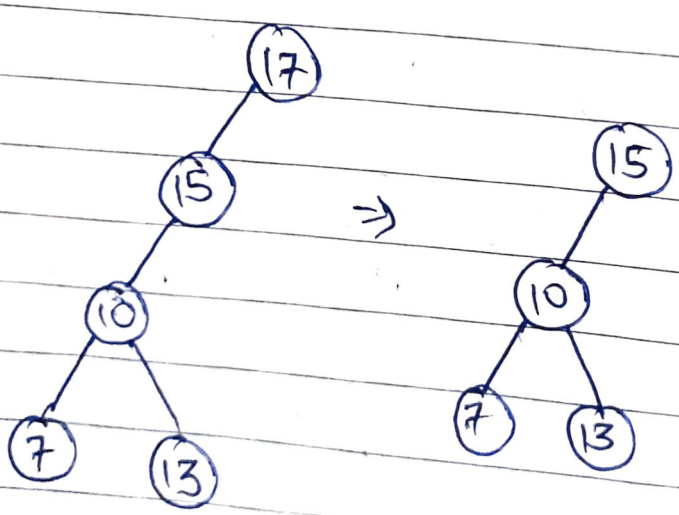
- Perform BST deletion, splay the parent of deleted node.



⇒ P.T.O.



Now, 20 is not present in tree, so we can't delete it for sure but the last accessed node was 17 as we searched for 20 in tree so 17 will be splayed.



- There are two ways of de splaying after deletion,
  (i) Top-down splaying
  (ii) Bottom-up slaying.

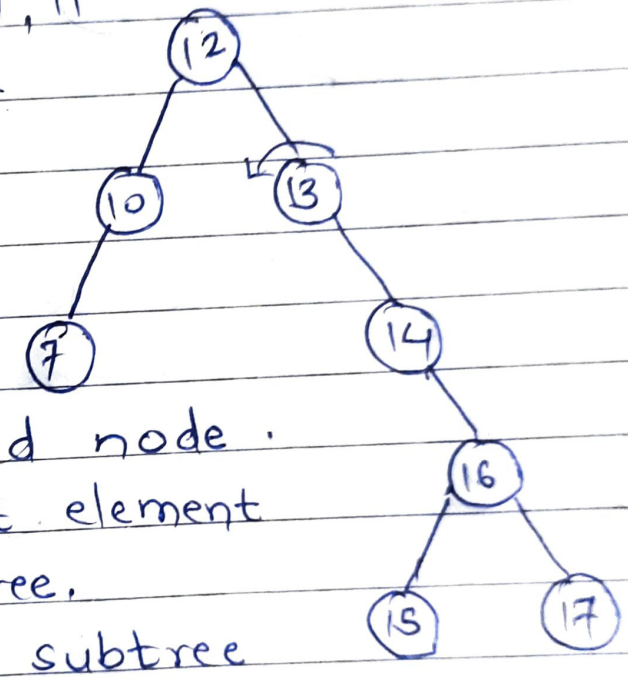
Above we learned deletion followed by "bottom-up splaying".

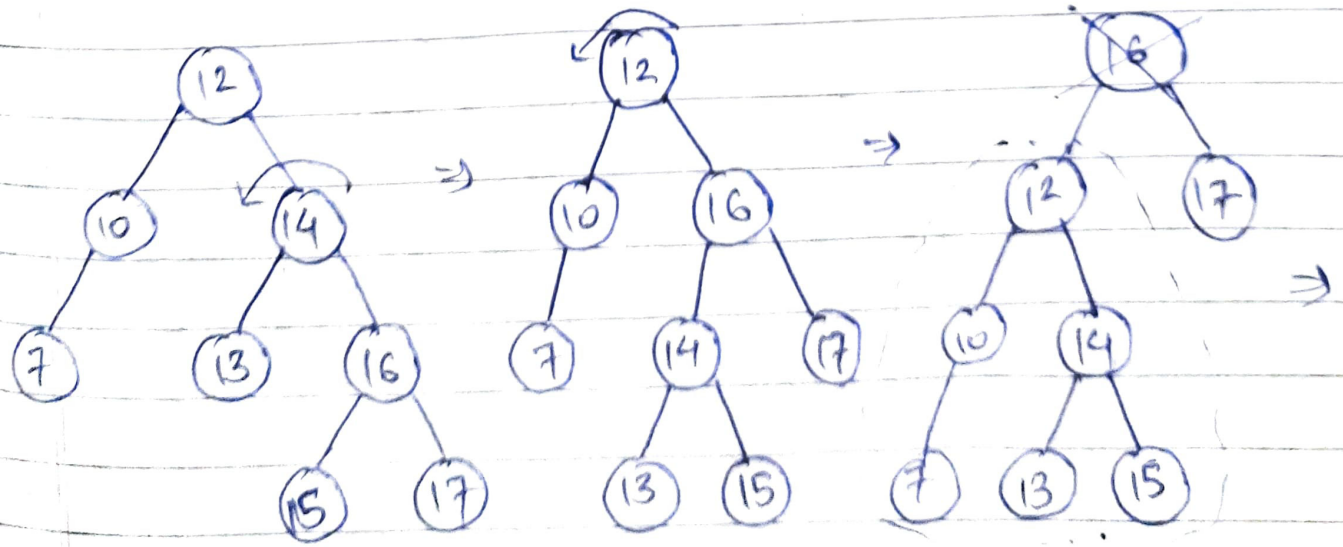
V-70

* Deletion in splay tree (using top-down splaying):

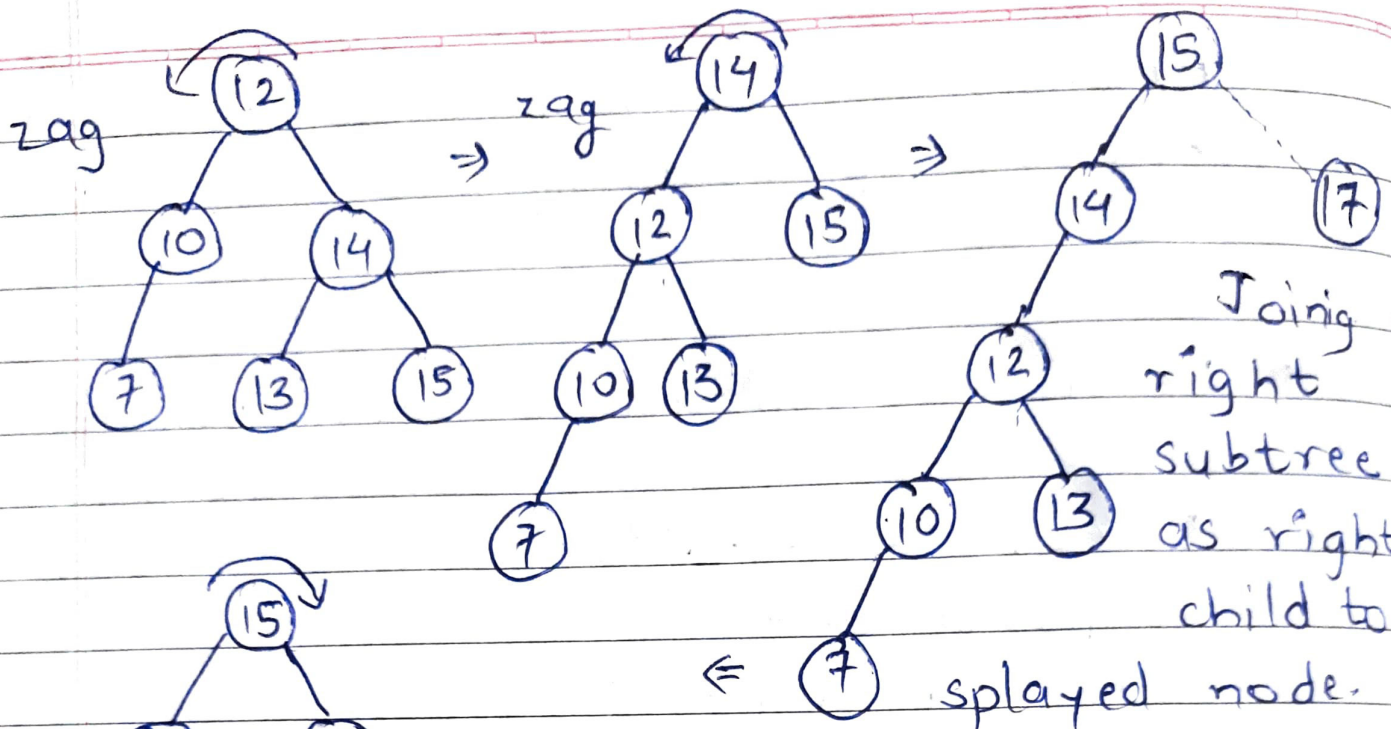
Ex:) Delete 16, 12, 7, 17, 11

- Steps to delete a node using top-down splaying.
  - → Splay the node to be deleted.
  - → Delete the splayed node.
  - → Splay the largest element in the left subtree.
  - → Join the right subtree to the splayed node as a right child.



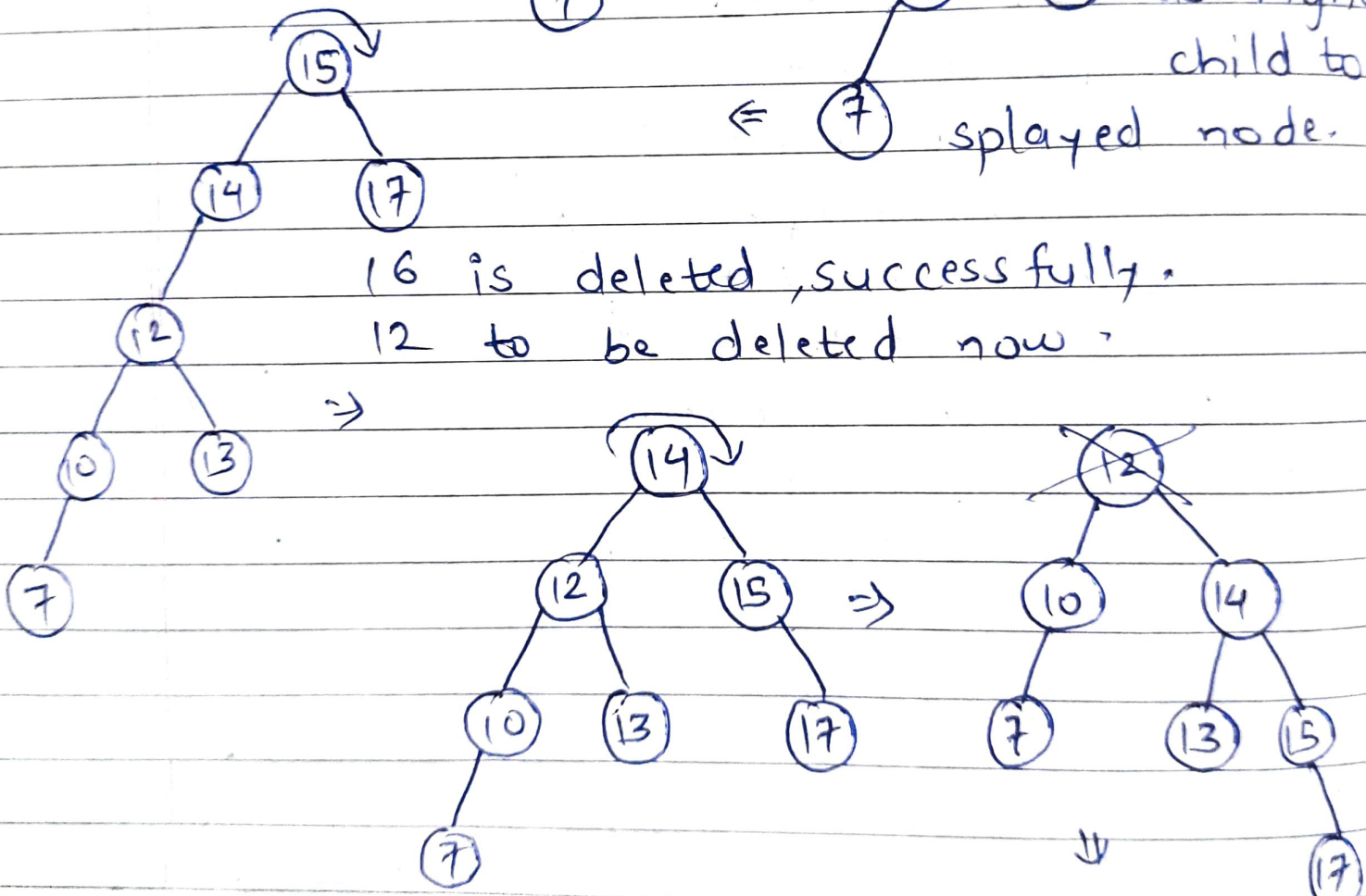


Now, we have to splay largest element from left subtree.

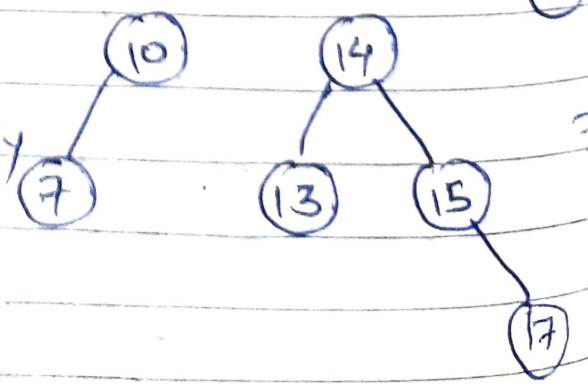


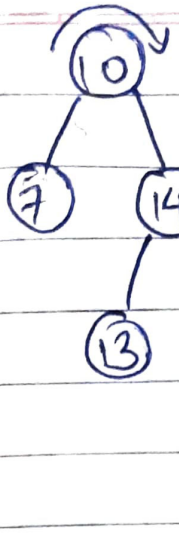
16 is deleted, successfully.

12 to be deleted now.

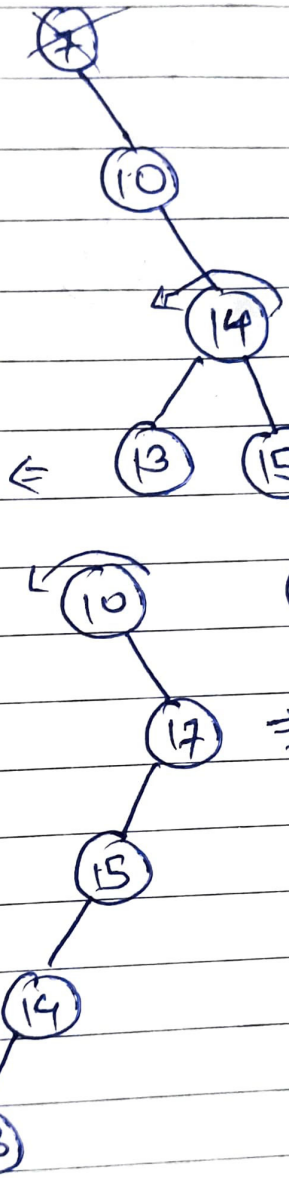


Largest element in left subtree is already root node, so no need to splay.





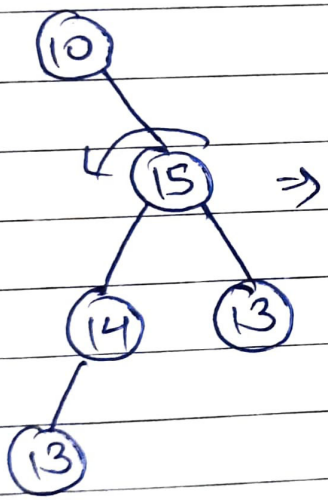
12 is deleted, successfully.
7 is to be deleted,

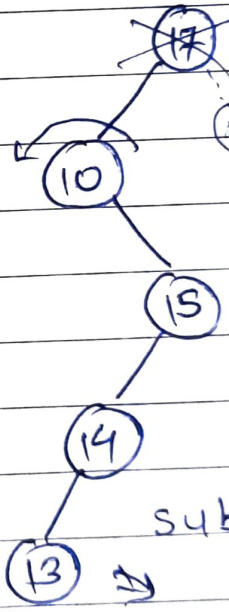


No left subtree is there so the right subtree becomes the tree now,

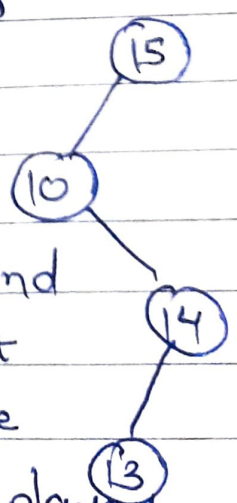
7 is deleted, successfully.

17 is to be deleted now,

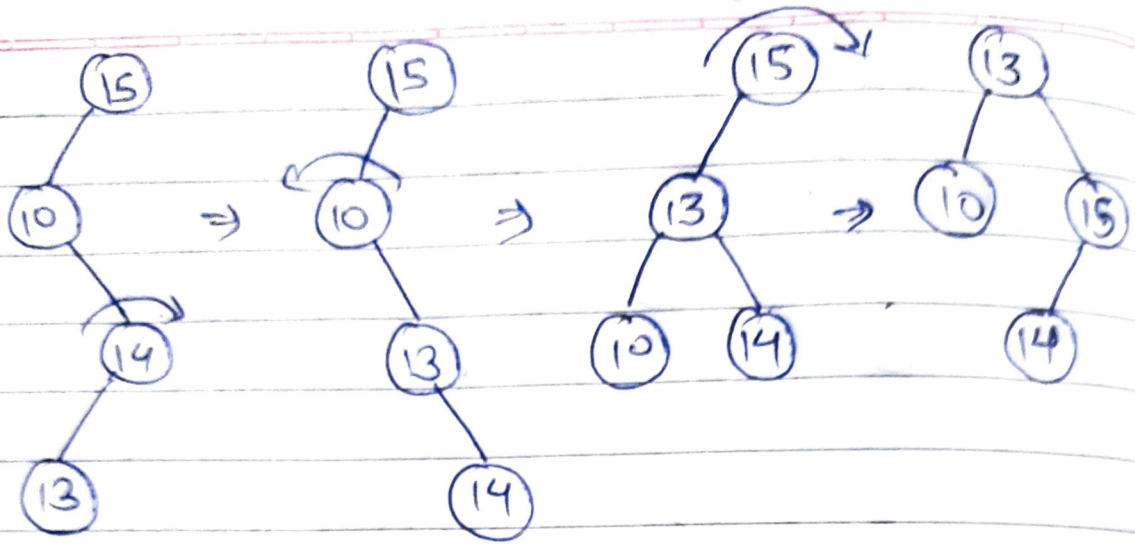




Splay largest element from left subtree now



17 is deleted, successfully,
Now, 11 is not present in the tree, so we can't splay it and delete, but the last element accessed in searching 11 will be splayed, so here 13 will be splayed.


15
10
14
13
15
10
13
14
15
13
10
14
13
10
15
14