Red Black Trees

Dr A Vani Vathsala

Why Red-Black Trees?

Most of the **BST operations** (e.g., search, insert, delete.. etc) take **O(h)** time where h is the height of the BST.

The cost of these operations may become **O(n) for a skewed Binary tree**, where n is the number of nodes.

If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can **guarantee time complexity of O(Logn)** for all these operations.

The **height of a Red-Black tree is always O(Logn)** where n is the number of nodes in the tree.

Comparison with AVL Tree

AVL trees are more balanced compared to Red-Black Trees, but they **may cause more rotations** during insertion and deletion.

So if our application involves **many frequent insertions and deletions**, then **Red Black** trees should be preferred.

And if the insertions and deletions are less frequent and **search is a more frequent operation**, then **AVL tree should be preferred over Red-Black Tree.**

Red Black Trees

A Red-Black tree is a **self-balancing binary search tree** in which each node contains an extra bit for denoting the color of the node, either red or black. (0 for black, 1 for Red)

A red-black tree satisfies the following properties:

Red/Black Property: Every node is colored, either red or black.

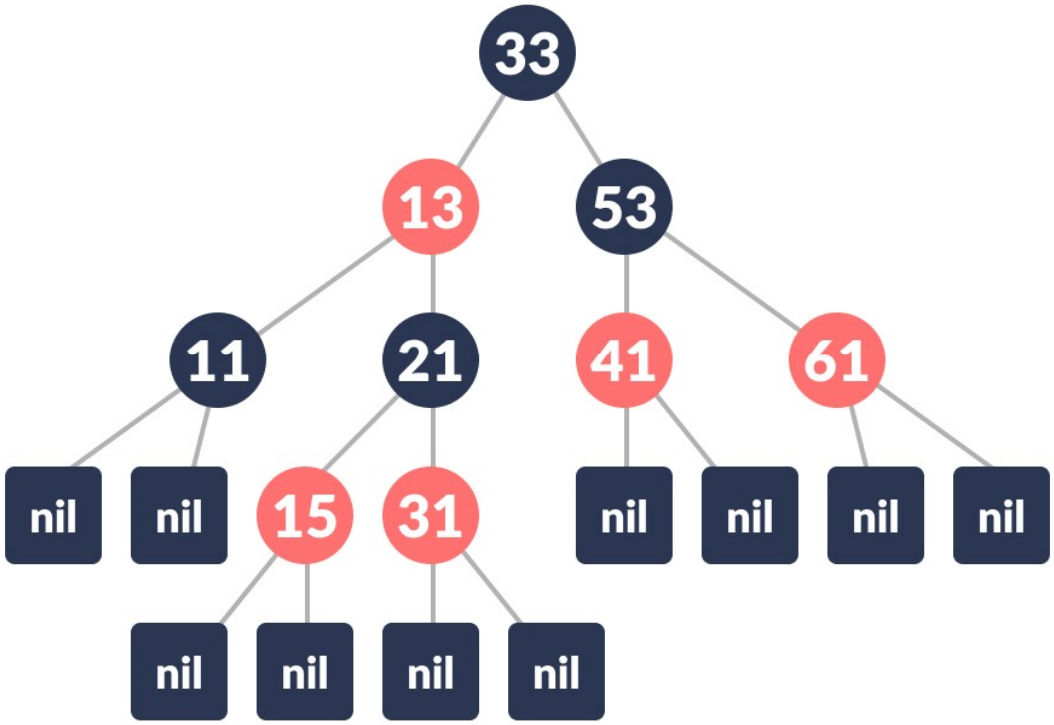
Root Property: The root is black.

Leaf Property: Every leaf (NIL Node) is black.

Red Property or Color Property: If a red node has children then, the children are always black nodes. (There cannot be two consecutive Red nodes on a path)

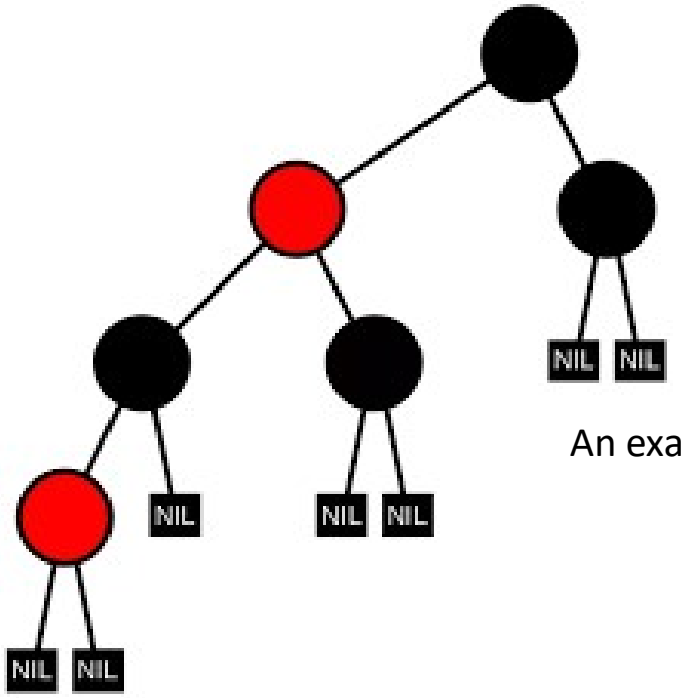
Depth Property: For each node, any simple path from this node to any of its descendant leaves has the same number of black nodes (also known as black-depth).

EXAMPLE OF A RED-BLACK TREE



Relation between RED-BLACK Trees and AVL Trees

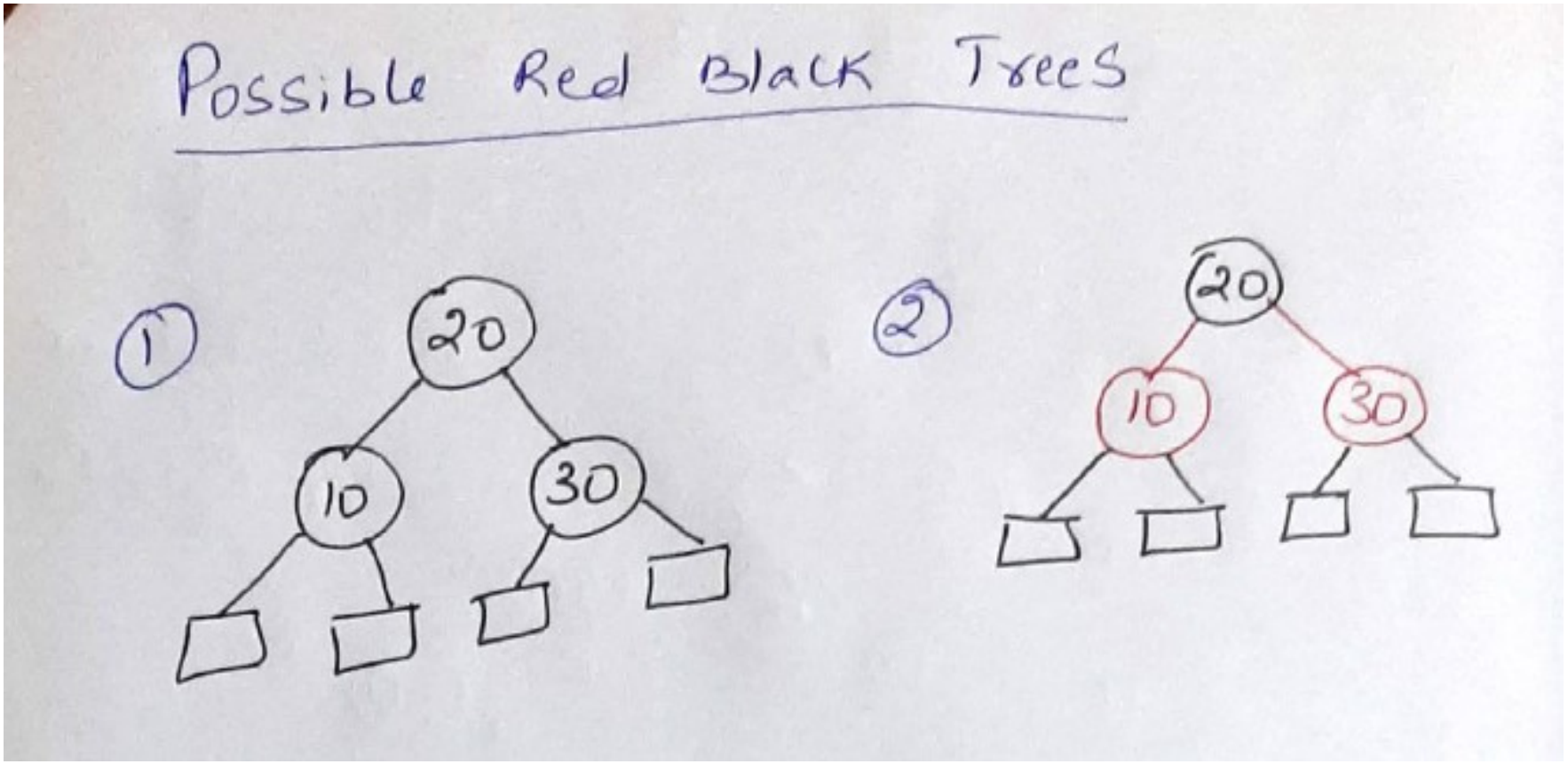
All AVL Trees are Red Black Trees. But not all Red Black Trees are AVL Trees.



An example of a Red Black Tree that is not an AVL Tree

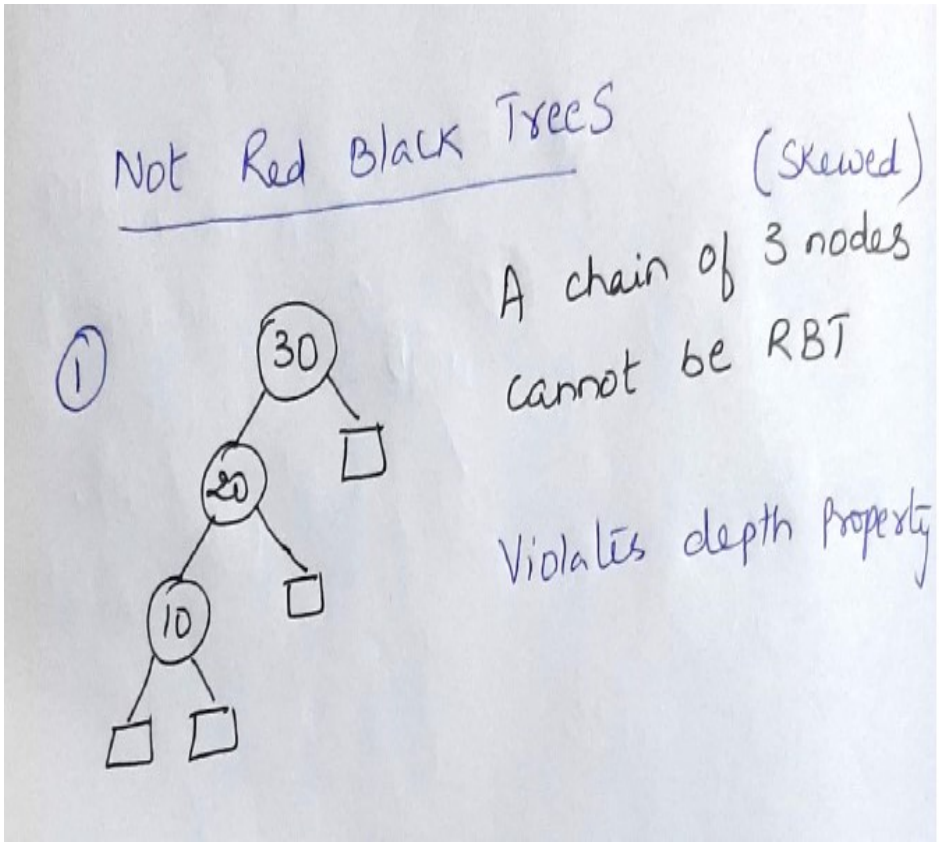
Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.

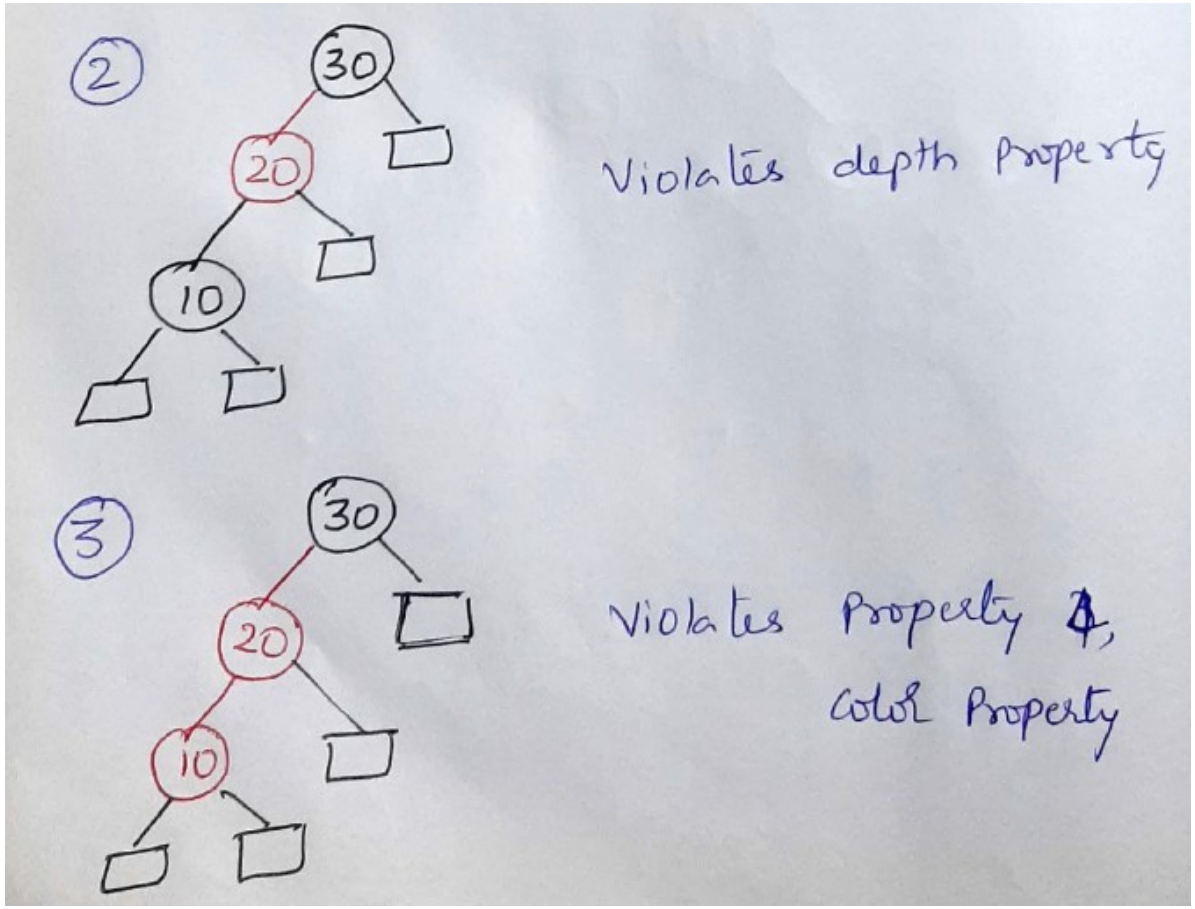
Possible Red Black Trees



Not Red Black Trees

A Skewed tree containing chain of 3 nodes cannot be a Red Black Tree





Insertion into RED BLACK Tree

The insertion operation in Red Black Tree is **similar to insertion operation in Binary Search Tree**.

But every node is **inserted with a color attached to it**.

After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform any of the following operations to make it Red Black Tree.

1. **Recolor**
2. **Rotation**
3. **Rotation followed by Recolor**

The **insertion operation** in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color **Red.**

Step 4 - If the parent of newNode is **Black** then exit from the operation.

Step 5 - If the parent of newNode is **Red** then check the color of parent node's sibling of newNode.

a) If it is colored **Red** then Recolor. Repeat the same until tree becomes Red Black Tree.

b) If it is colored **Black** or NULL then make suitable Rotation and Recolor it.

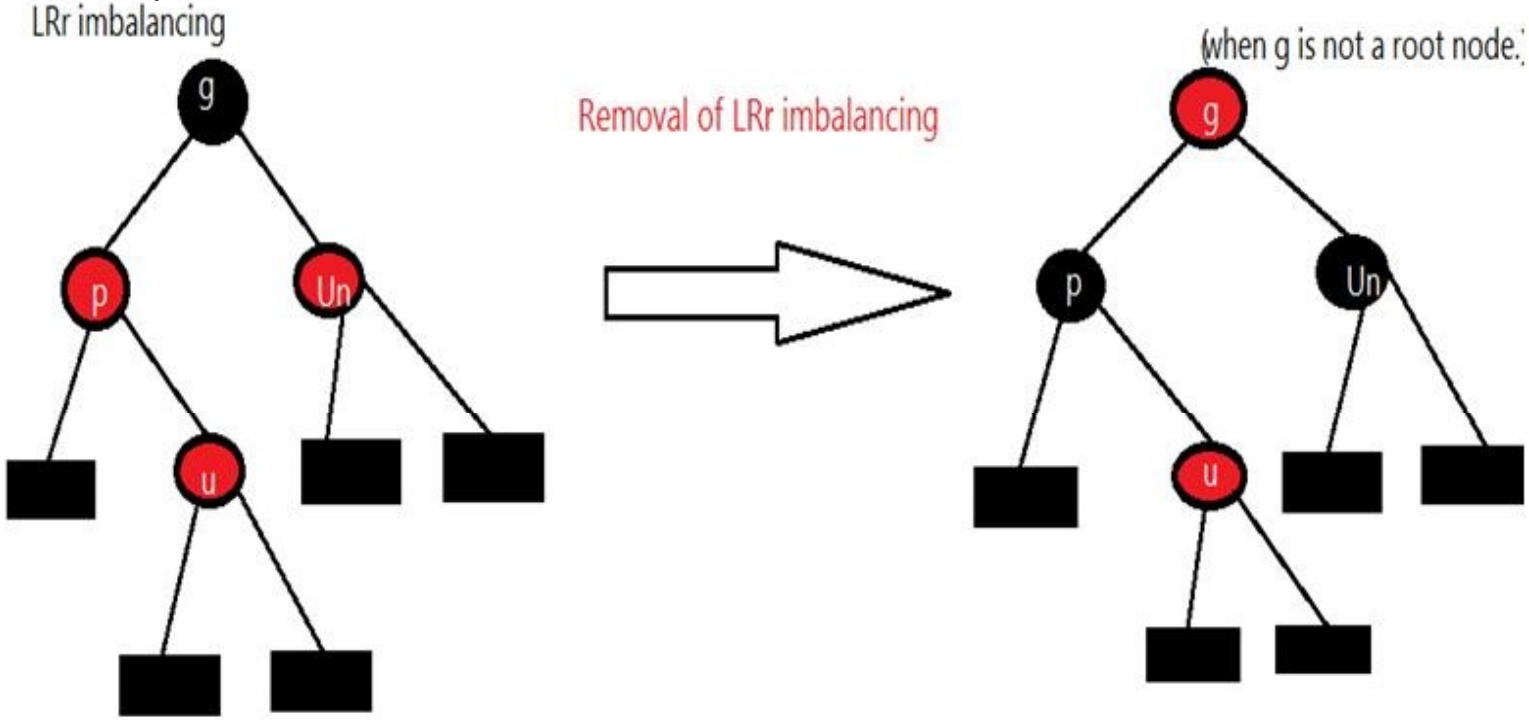
Balancing RED-Black Trees during Insertions

- **The following notation is used to name** the keys while defining cases of balancing
  - **u** is the **new**ly inserted node.
  - **p** is the **parent** node of **u**.
  - **g** is the **grandparent** node of **u**.
  - **Un** is the **uncle** node of **u**. (Child of g, sibling of p)

- Insertion cases can be broadly classified in to two cases:
  - **Case a):** Colour of **Un** node is **Red**. This case has 4 subcasses depending on positions of u,p, and g. No Rotations are required in this case, Only Recoloring has to be done.
    - LLr, LRr, RRr, RLr

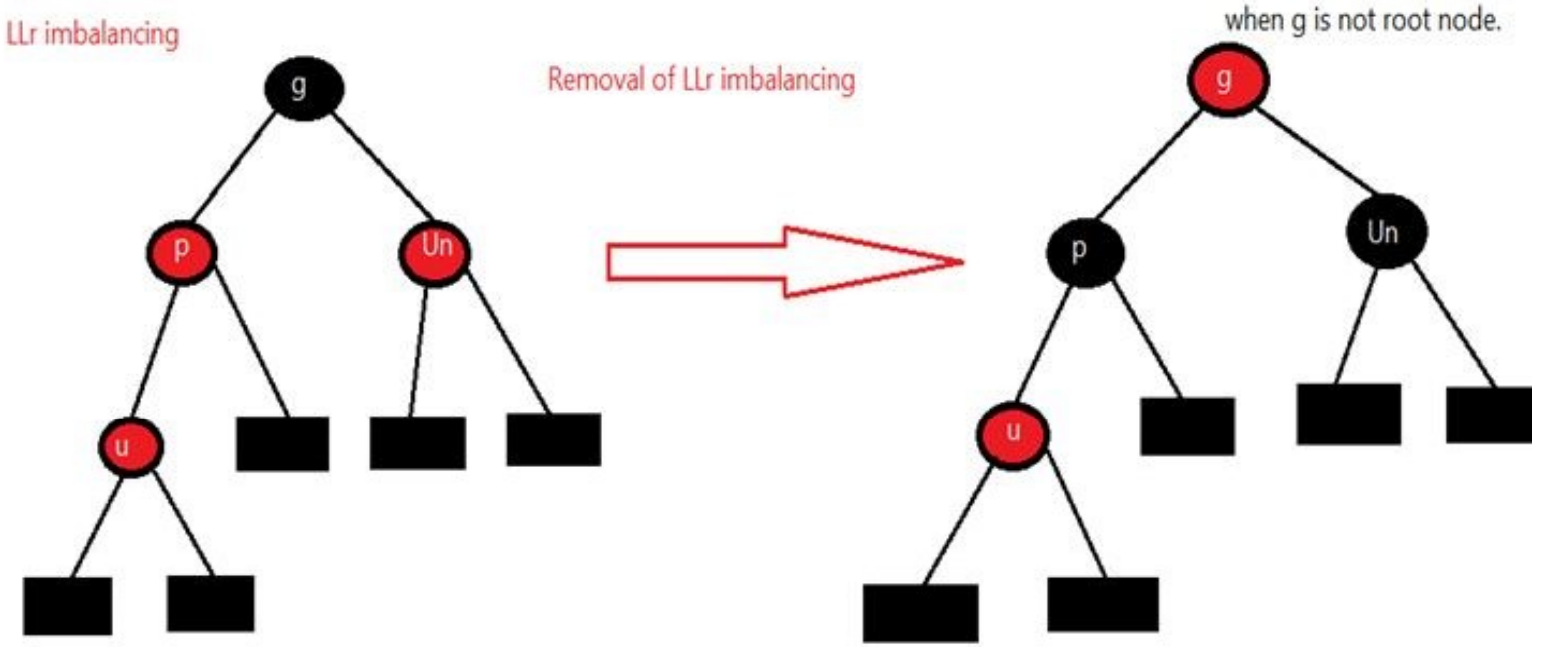
  - **Case b):** Colour of **Un** node is **Black or there is no Un node**. This case has 4 subcasses depending on positions of u,p, and g. Rotations are first done and then, recoloring has to be done if necessary.
    - LL, LR, RR, RL

1) **LRr imbalance :** In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at left and right positions with respect to grandparent **g** respectively. Therefore it is termed as Left Right imbalance**.** **r** in the case name represents **red** color for node Un.



Removal of LRr imbalance can be done by:
Change color of p from red to black.
Change color of Un from red to black.
Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.
Note: If given g is root node then there will be no changes in color of g.

2) LLr imbalance : In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at left and left positions with respect to grandparent **g** respectively. Therefore it is termed as Left Left imbalance**.**



Removal of LLr imbalance can be done by:

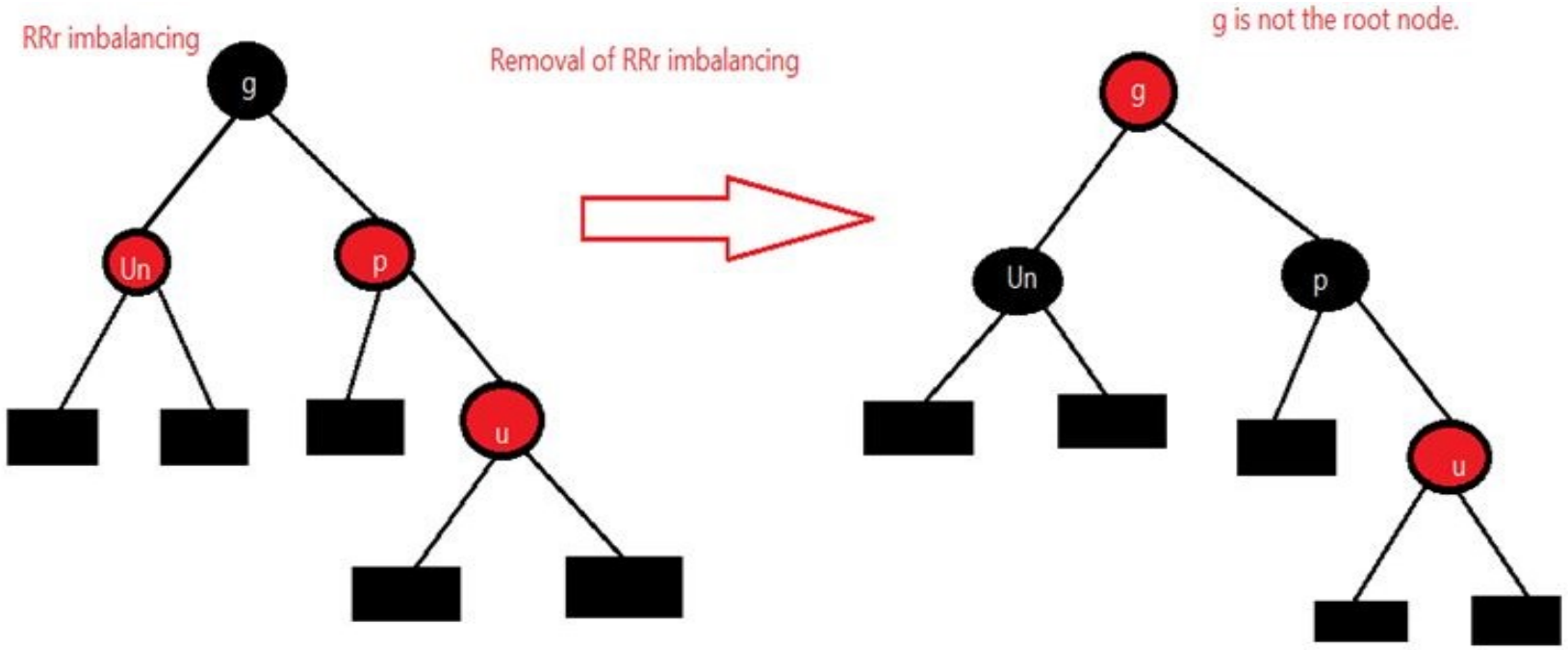
i.Change color of p from red to black.

ii.Change color of Un from red to black.

iii.Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.

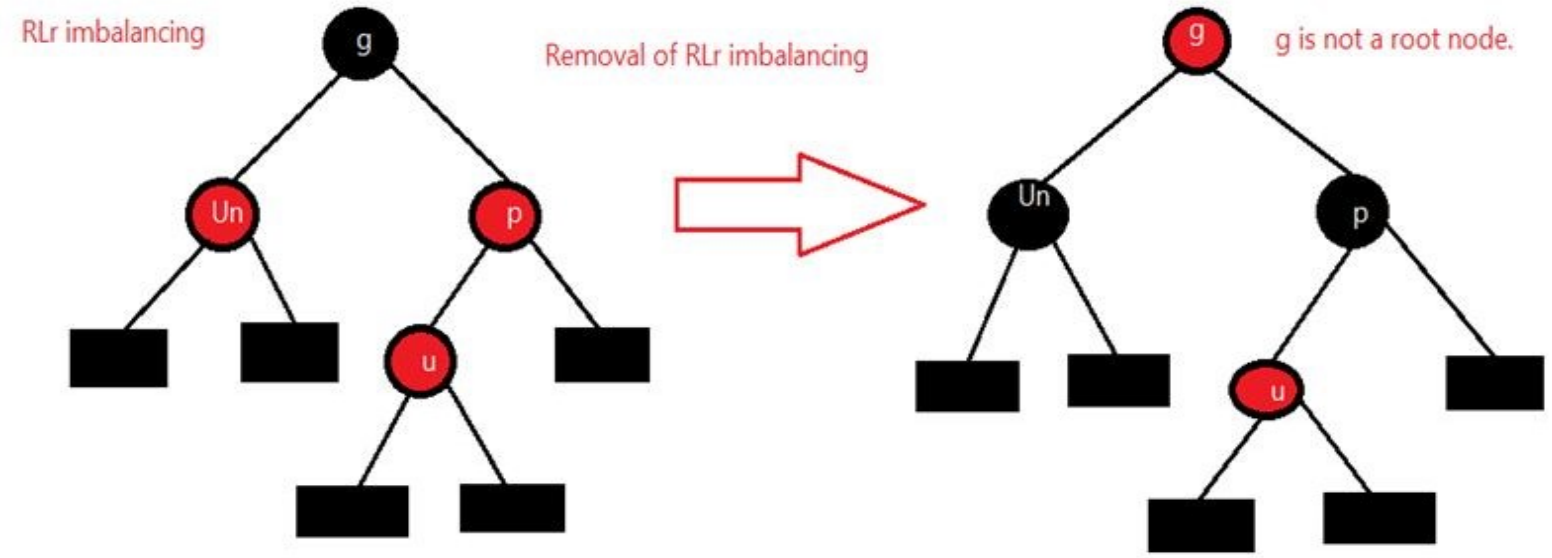
Note: If given g is root node then there will be no changes in color of g.

3) RRr imbalance : In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at right and right positions with respect to grandparent **g** respectively. Therefore it is termed as Right Right imbalance**.**



Removal of RRr imbalance can be done by:
Change color of p from red to black.
Change color of Un from red to black.
Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.
Note: If given g is root node then there will be no changes in color of g.

4) RLr imbalance : In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at right and left positions with respect to grandparent **g** respectively. Therefore it is termed as Right Left imbalance**.**

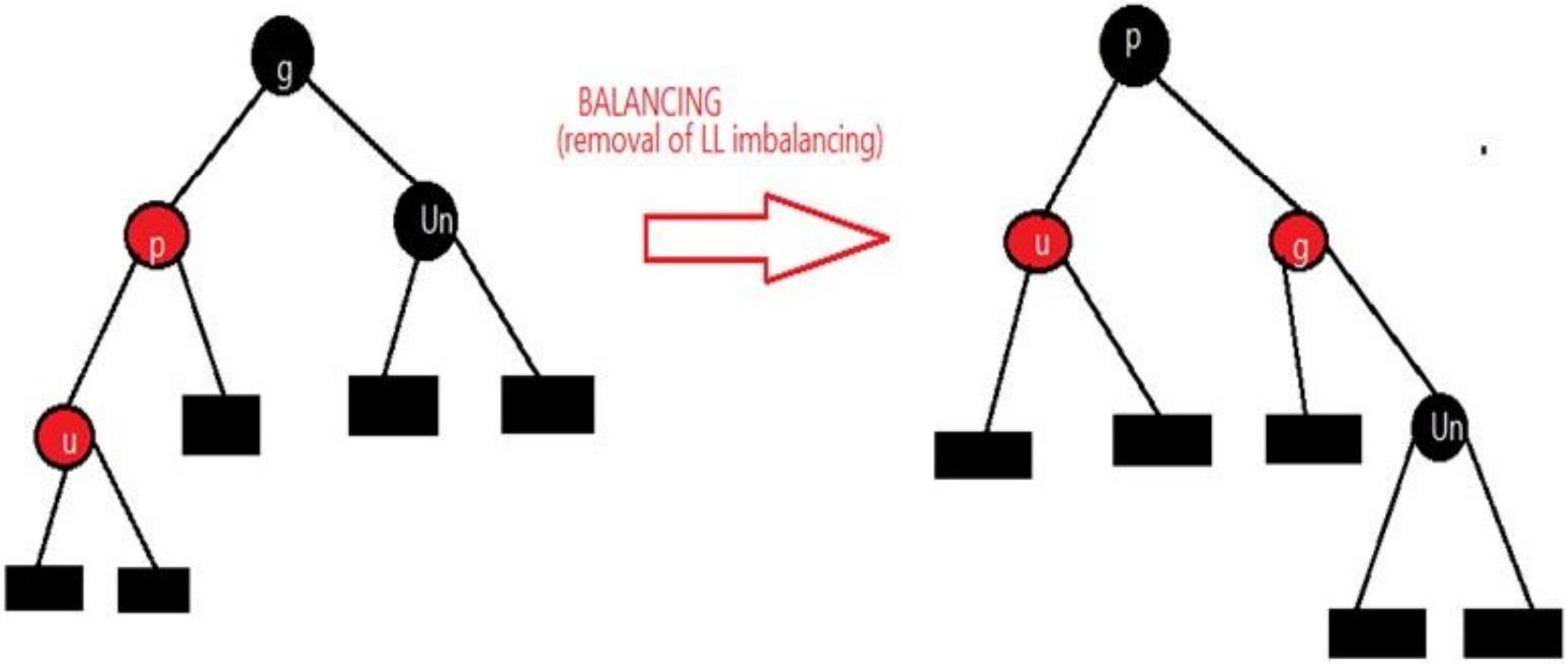


Removal of RLr imbalance can be done by:
Change color of p from red to black.
Change color of Un from red to black.
Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.
Note: If given g is root node then there will be no changes in color of g.

In the remaining four cases, color of node **Un is black or Un is not present.**

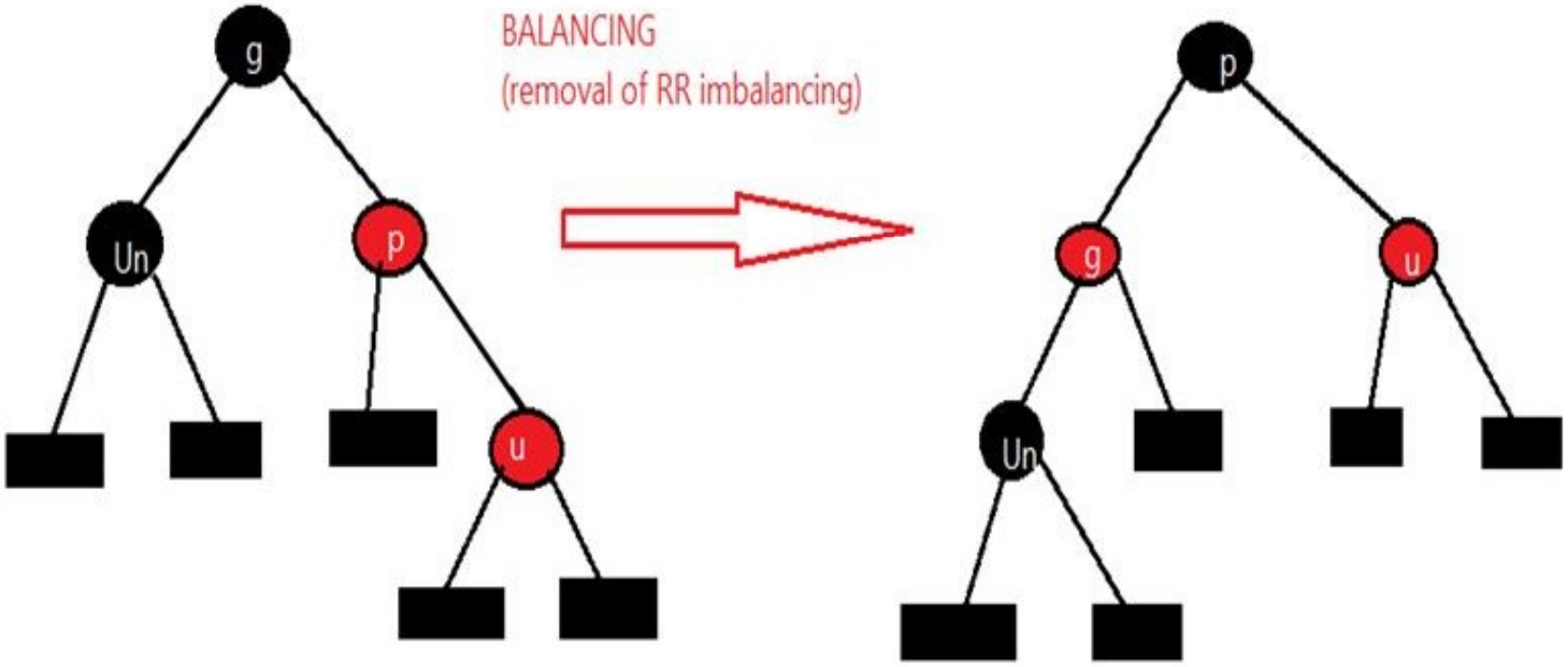
5) LL imbalance: It can be balanced by following two steps:

- Apply single rotation of p around g. (LL Rotation of AVL Tree)
- Recolor **p to black** and **g to red**.



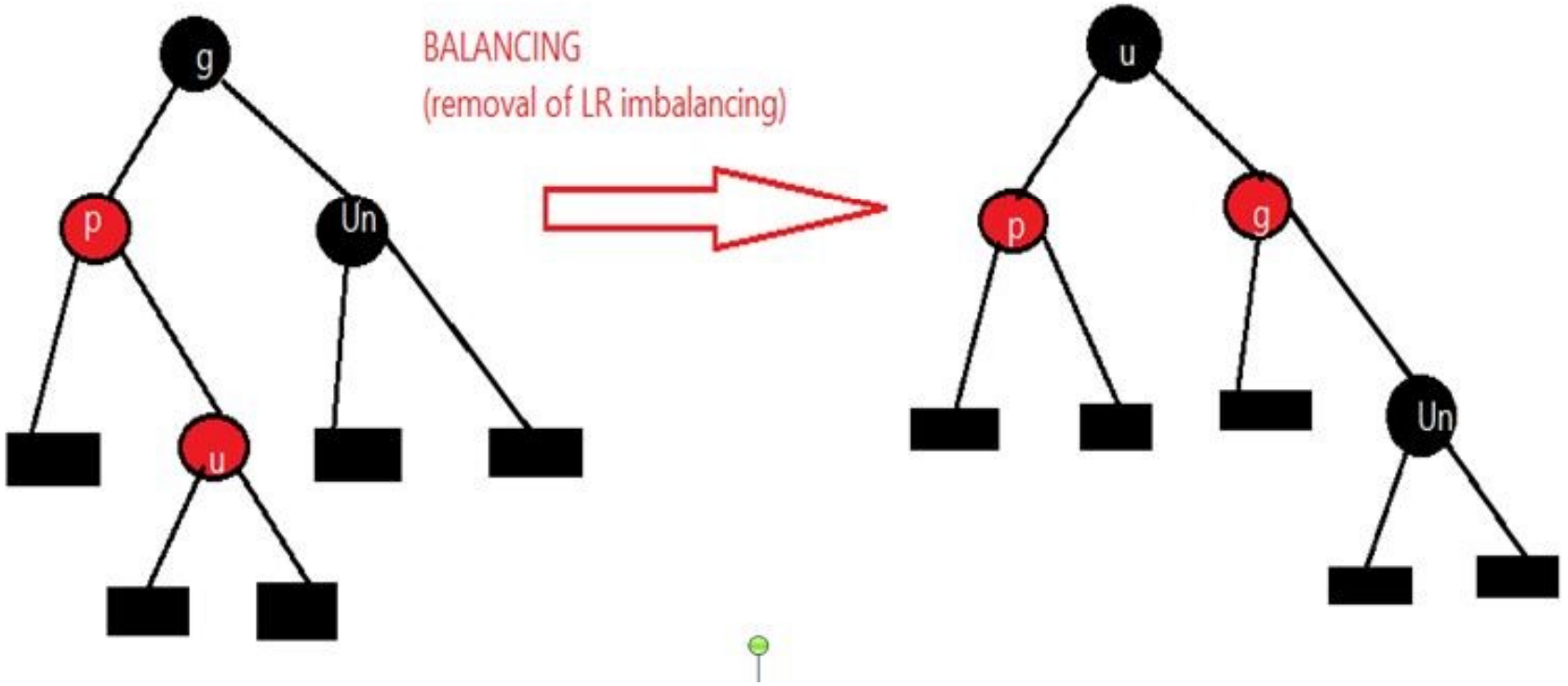
6) RR imbalance: It can be balanced by following two steps:

- Apply single rotation of p around g. (RR Rotation of AVL Tree)
- Recolor **p to black** and **g to red**.



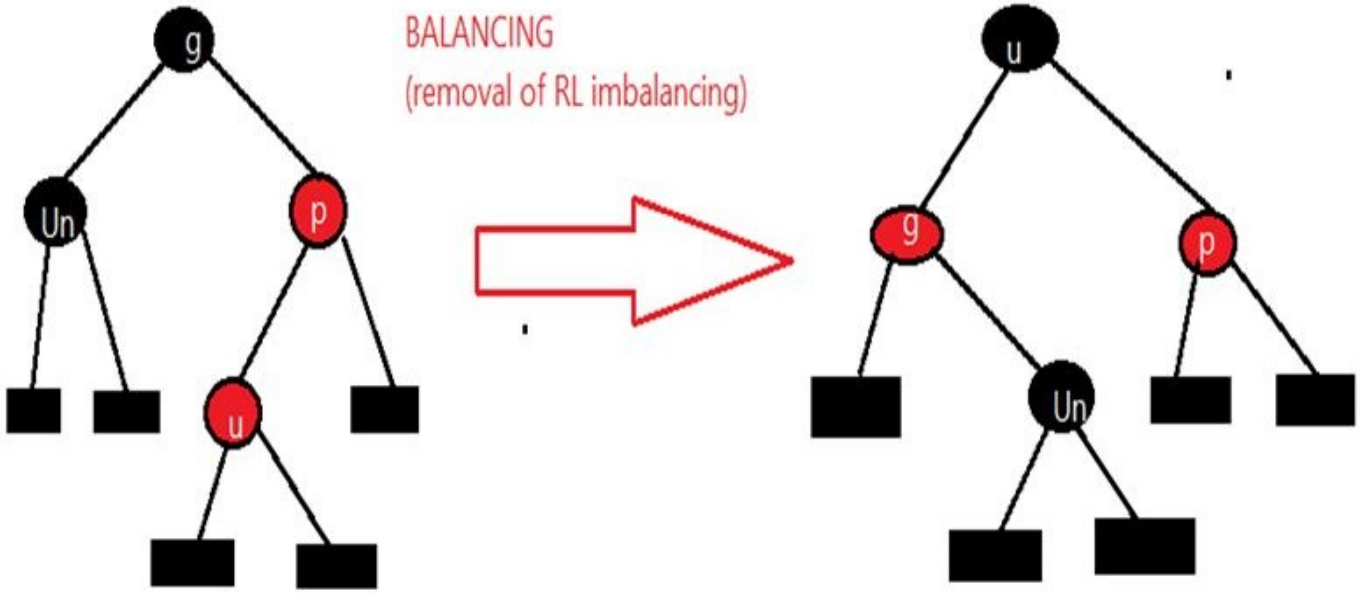
7) LR imbalance can be balanced by following steps:

- Apply double rotation of **u** around **p** followed by **u** around **g**. (LR Rotation of AVL Tree)
- Recolor **u to black** and **g to red.**



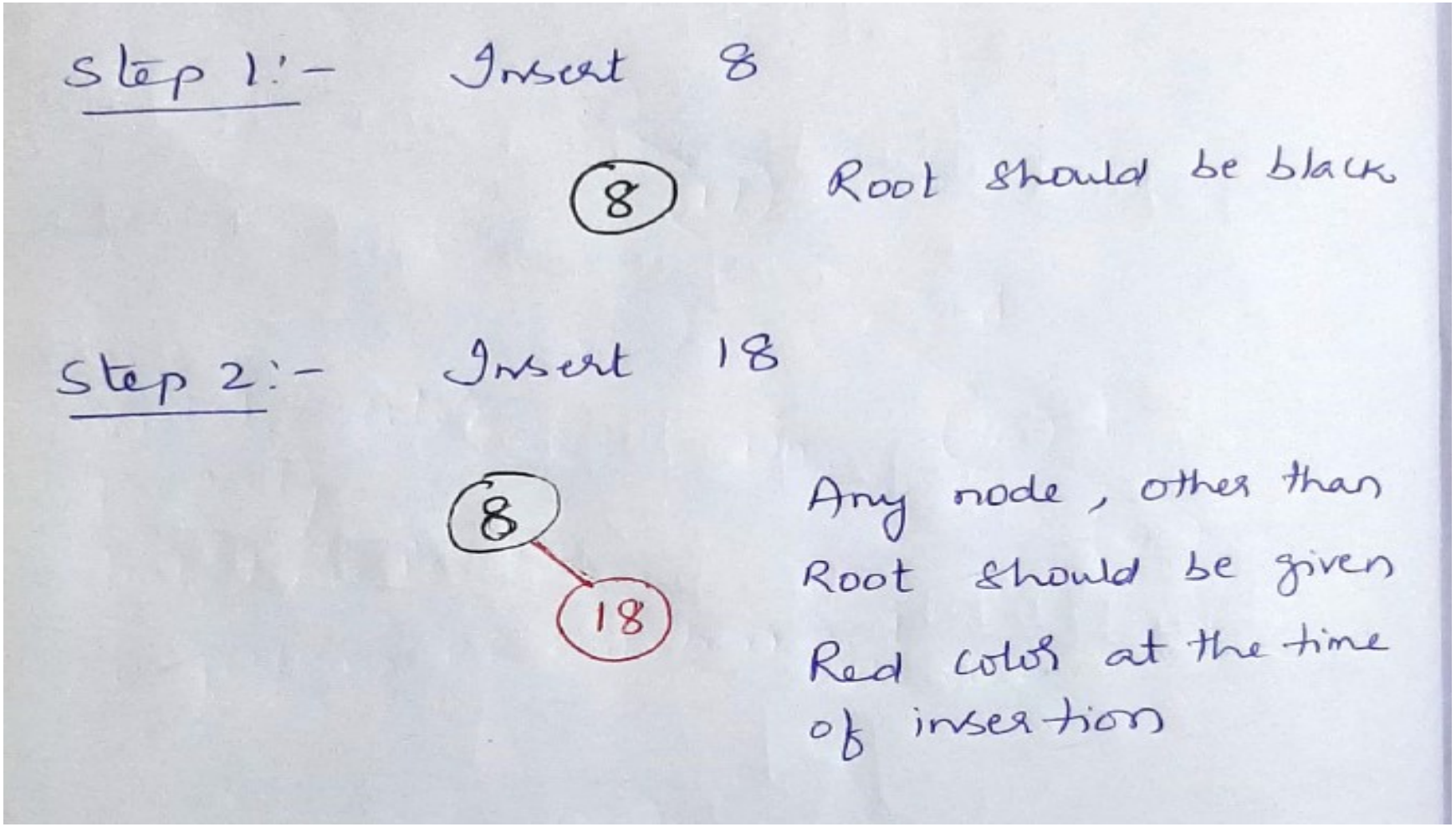
8) RL imbalance can be balanced by following steps:

- Apply double rotation of **u** around **p** followed by **u** around **g**. (RL Rotation of AVL Tree)
- Recolor **u to black** and **g to red.**



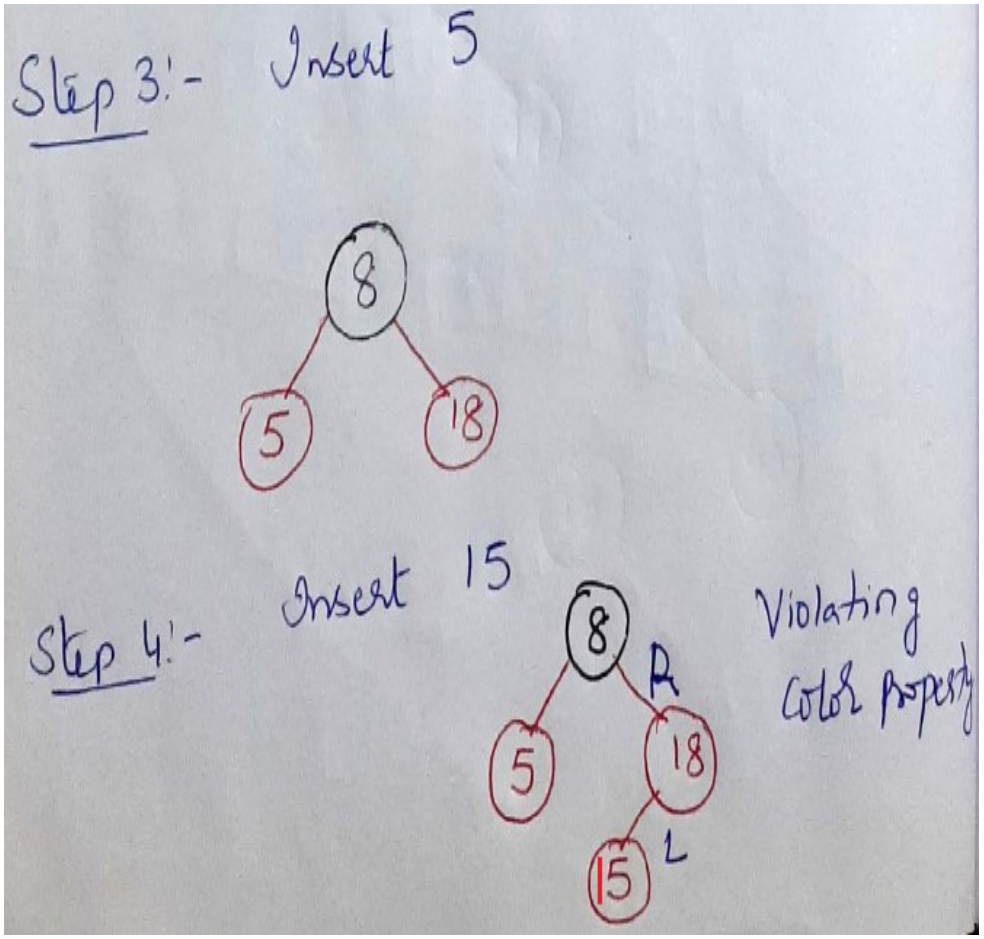
Red Black Tree Insertion Ex

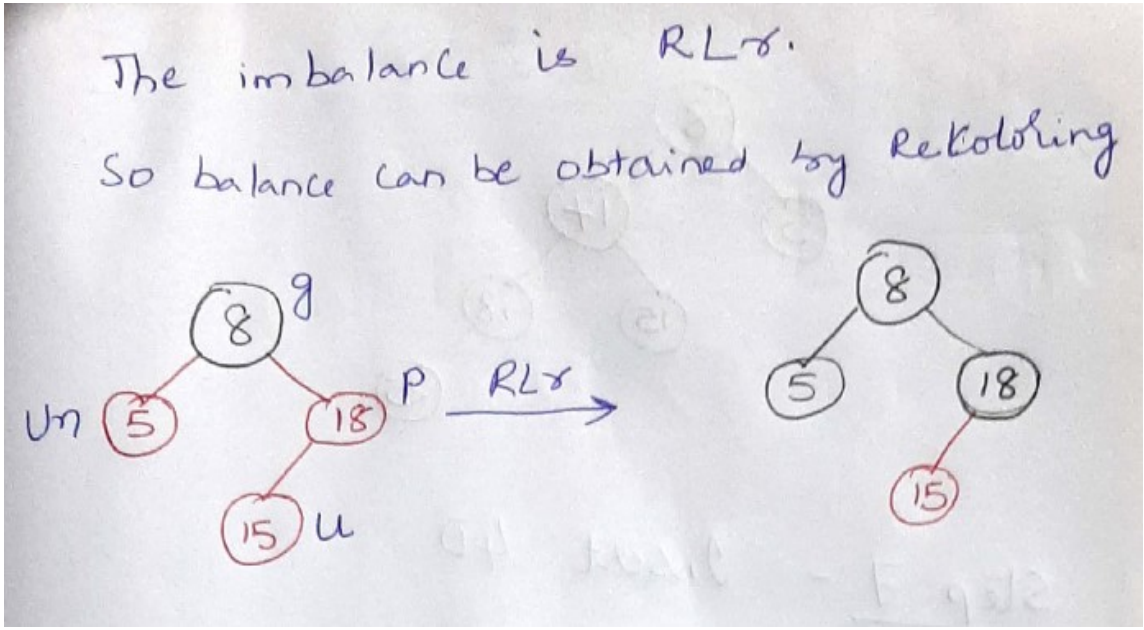
- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**



Red Black Tree Insertion Ex

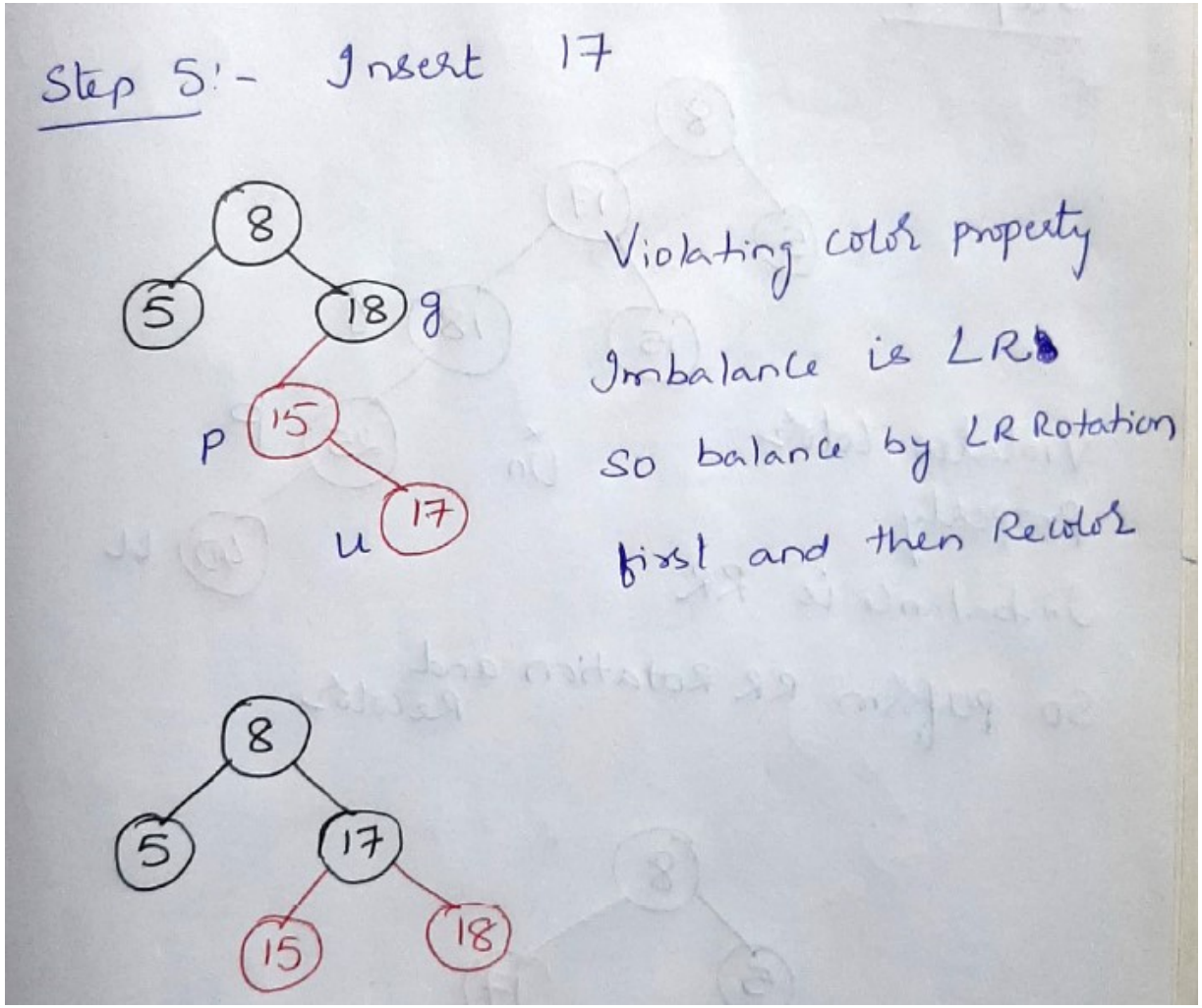
- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

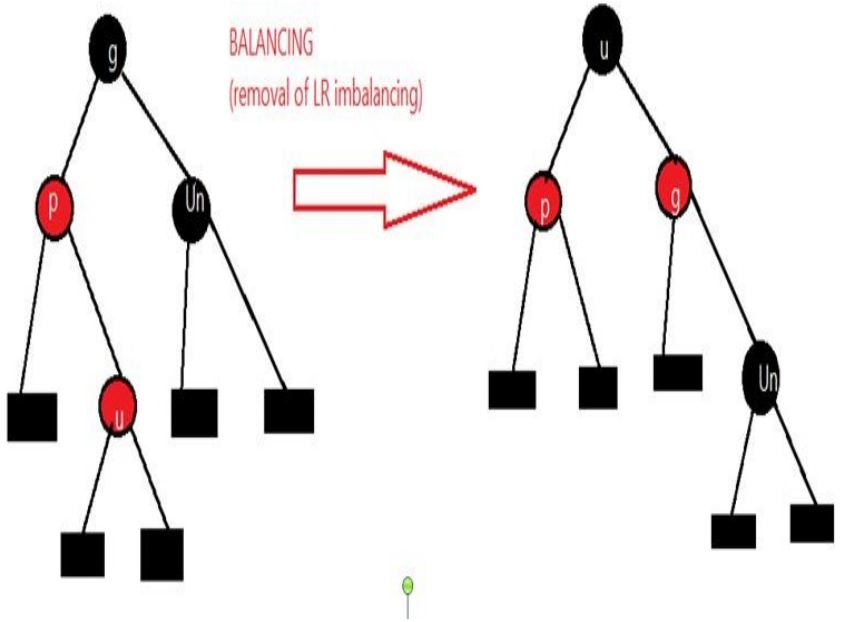




Red Black Tree Insertion Ex

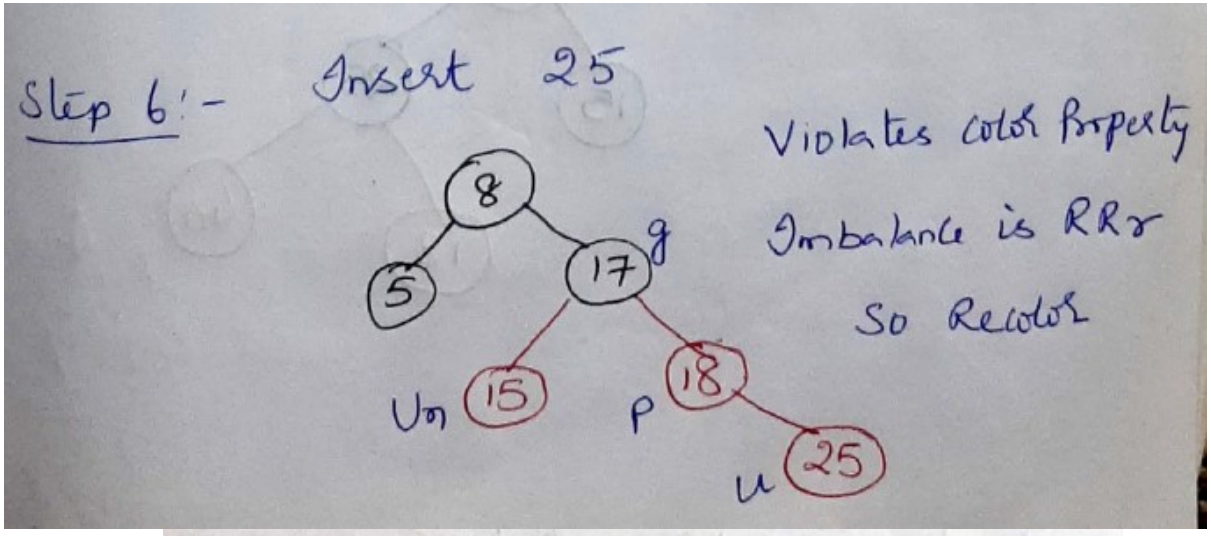
- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

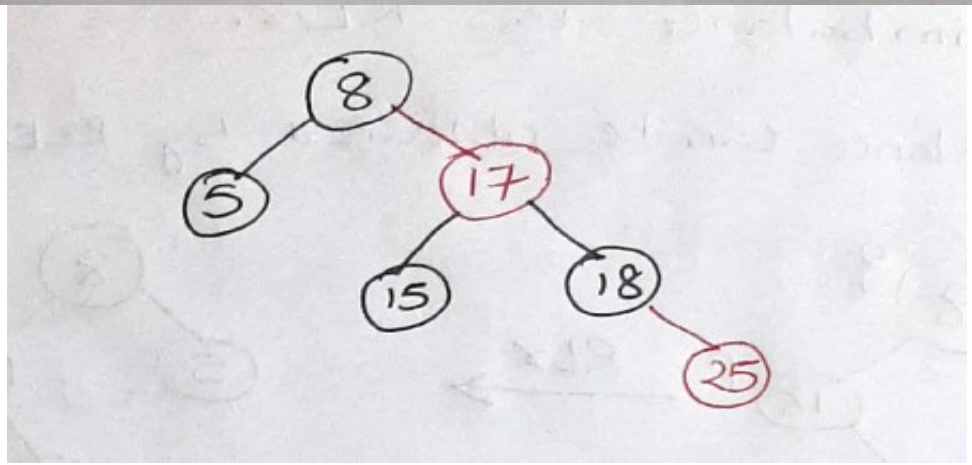




Red Black Tree Insertion Ex

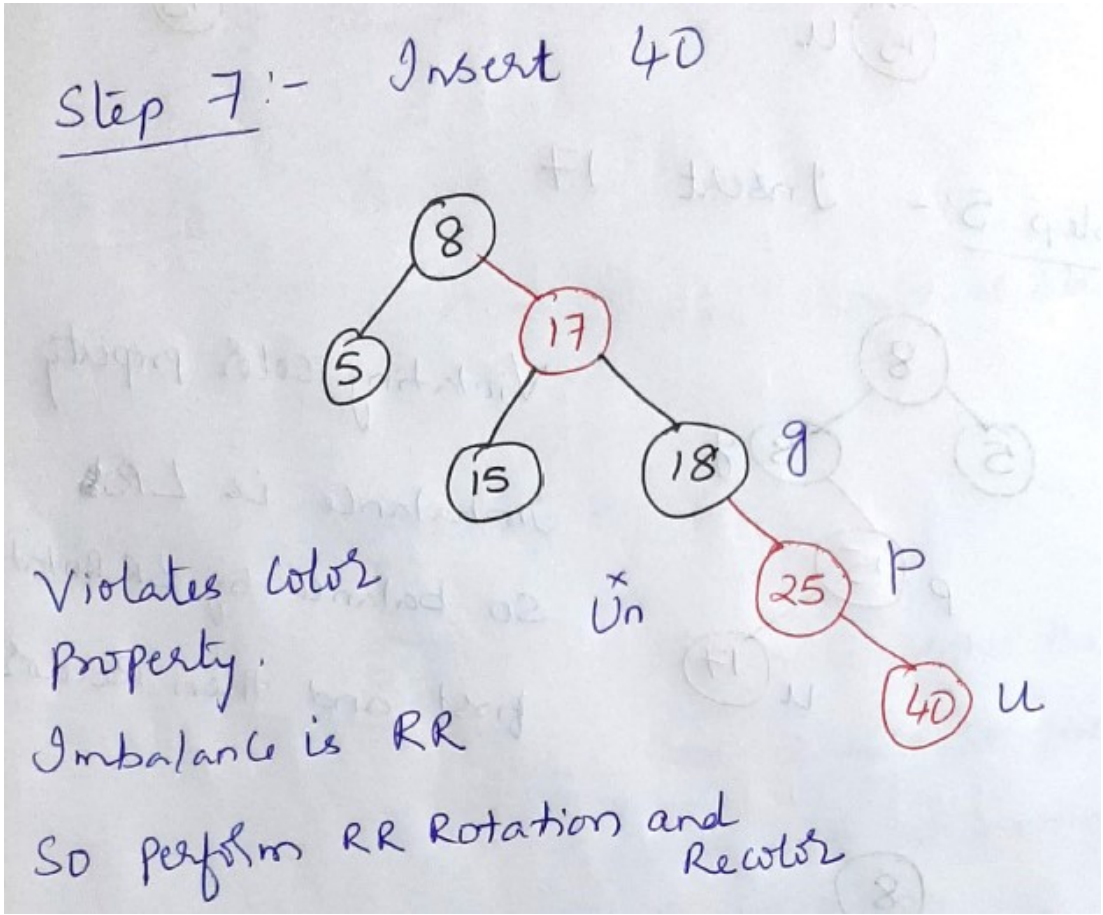
- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

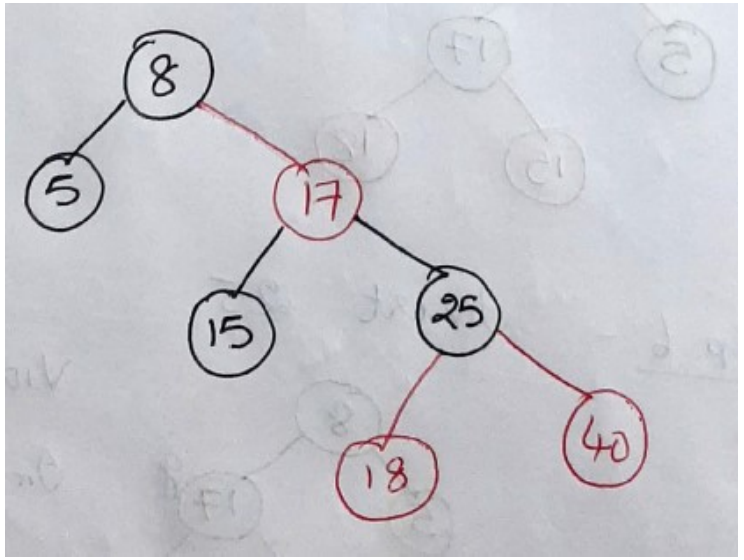




Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

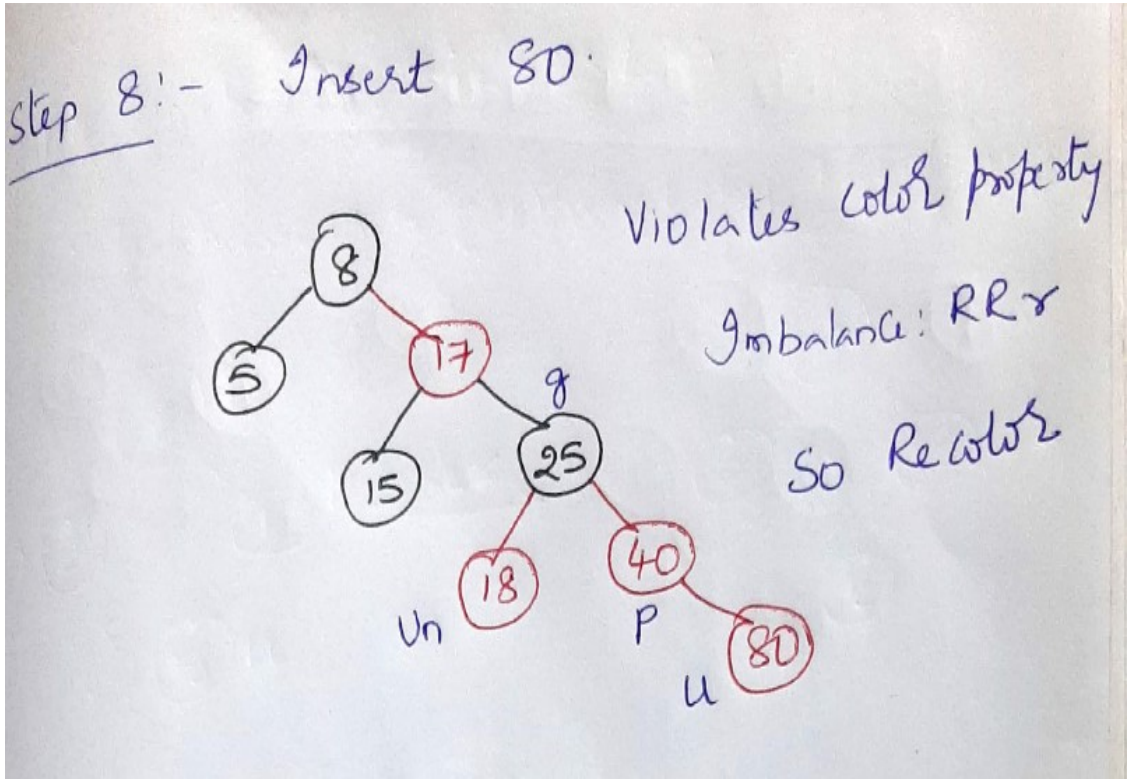


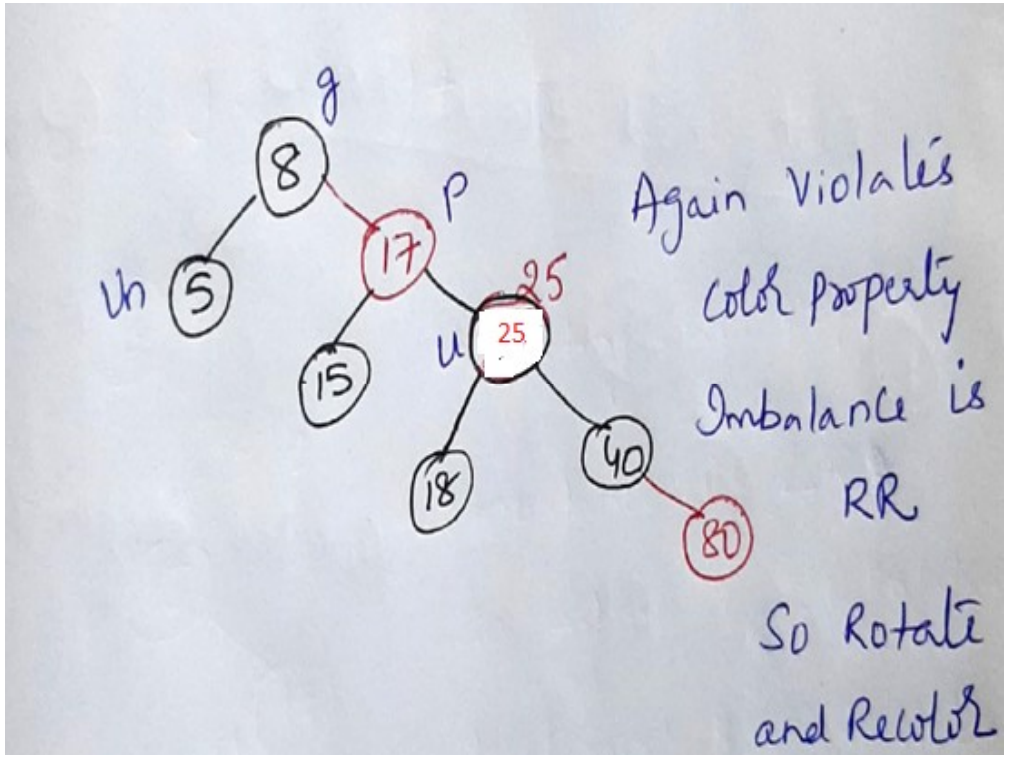


In RR and LL Rotations, remember , **p is made black** And **g is made red**.

Red Black Tree Insertion Ex

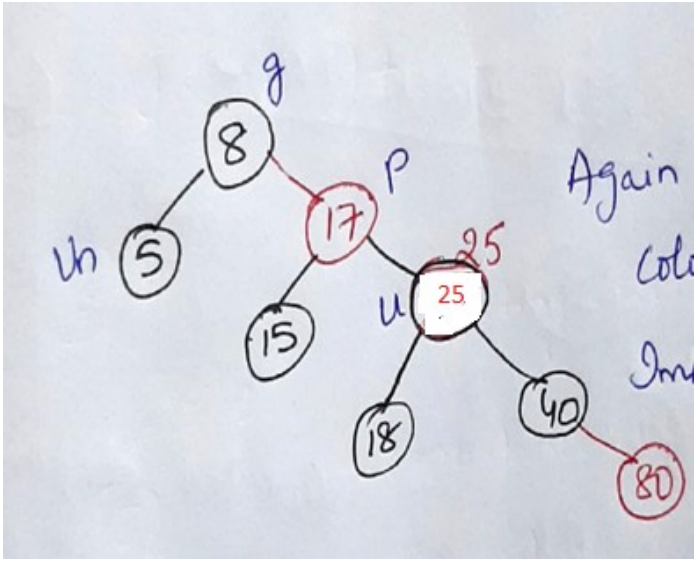
- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**





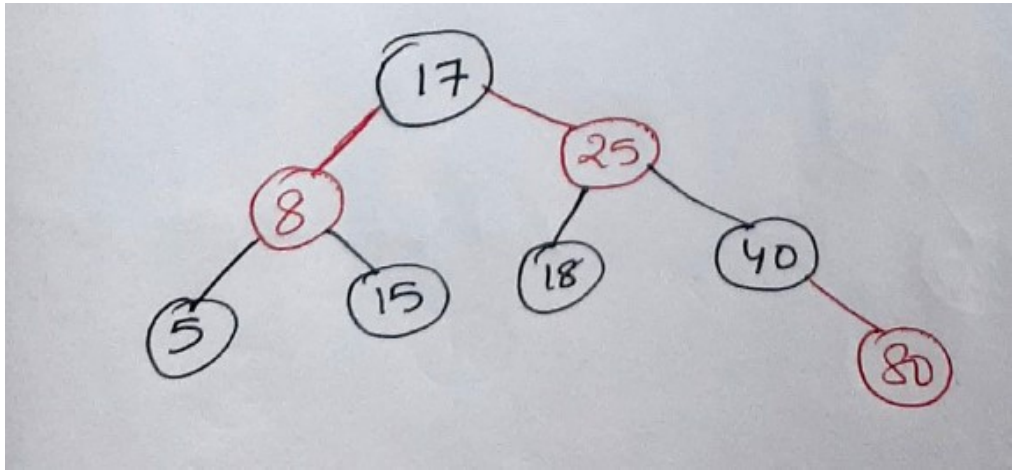
Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**



In RR and LL Rotations, remember , **p becomes the new root of the subtree and is made black, a**nd **g is made red**.

In RL and LR Rotations, remember , **u becomes the new root of the subtree and is made black, a**nd **g is made red**.



Exercise: Create a Red Black Tree With the following nos:
10,3,5,2,6,1,16,14,25,13,11

Red Black Tree Deletion

- Like Insertion, even in deletion recoloring and rotations are used to maintain the Red-Black properties.
- The main property **that violates after insertion is two consecutive reds**.
- In imbalances during insert operation, **we check color of uncle** to decide the appropriate case.
- In **delete**, the main violated property is, **change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.**
- In imbalances during delete operation, ***we check color of sibling*** to decide the appropriate case.

Red Black Tree Deletion

Following are detailed steps for deletion.

Step 1) Perform standard BST delete.

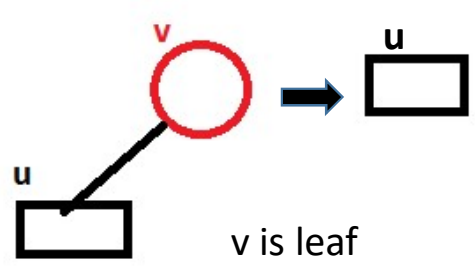
When we perform standard delete operation in BST, we always end up **deleting a node which is either leaf or has only one child** (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child).

So we only need to handle cases of deleting a node that is a leaf or has one child.

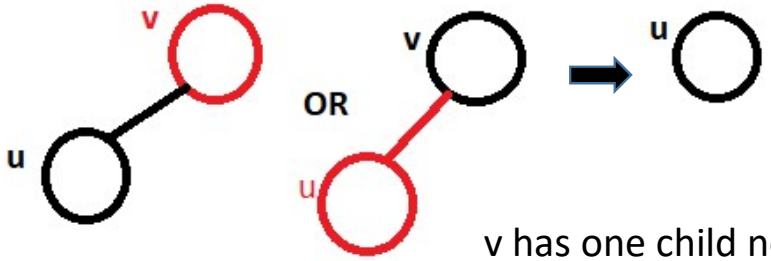
Let **v be the node to be deleted** and **u be the child that replaces v** (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

Step 2) Recolor or Rotate/Recolor

Case 1: Simple Case: If either v or u is red, we mark the replaced child as black (In this case, there will be no change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



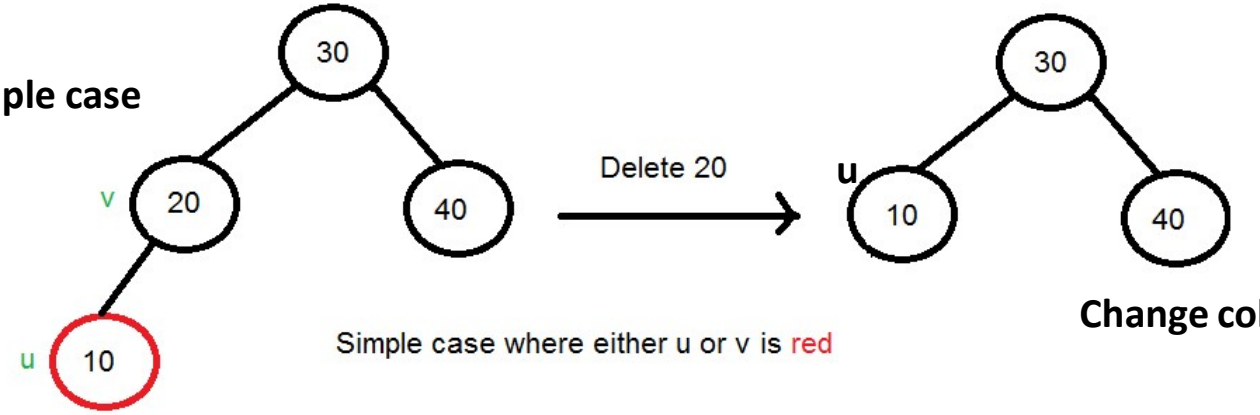
v is leaf



v has one child node

Red Black Tree Deletion

Ex: Simple case



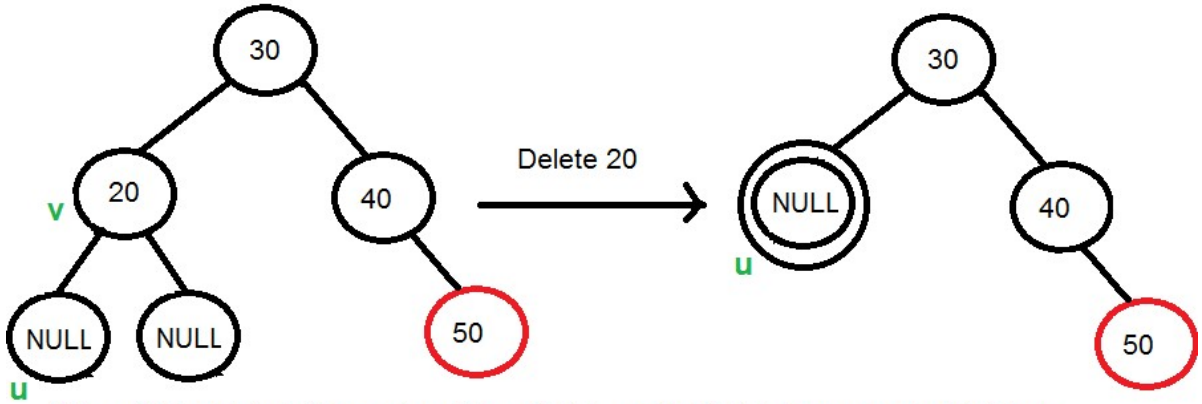
Simple case where either u or v is red

Change color of u from red to black.

Case 2) If Both u and v are Black.

When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now is to **convert this double black to single black**. **Color u as double black.**

Note : If v is leaf, and u is NULL , color of NULL is considered as black. So the **deletion of a black leaf also causes a double black.**



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

Red Black Tree Deletion

Case 2 is further divided into following sub cases: **(In all these case Let sibling of node u be s.)**

2.1) If root is double black, just remove double black and keep it as black node.

In all the remaining cases, the current node **u is double black** and it is not root. The remaining cases are broadly classified into the following two cases:

2.2) Color of sibling s **is black:**

a) both children of s **are also black: Only Recolor**

b) at least one of sibling's children **is red. Has 2 subcases: Perform Rotation and Recolor in both the cases**

i) parent p of u **is black.**

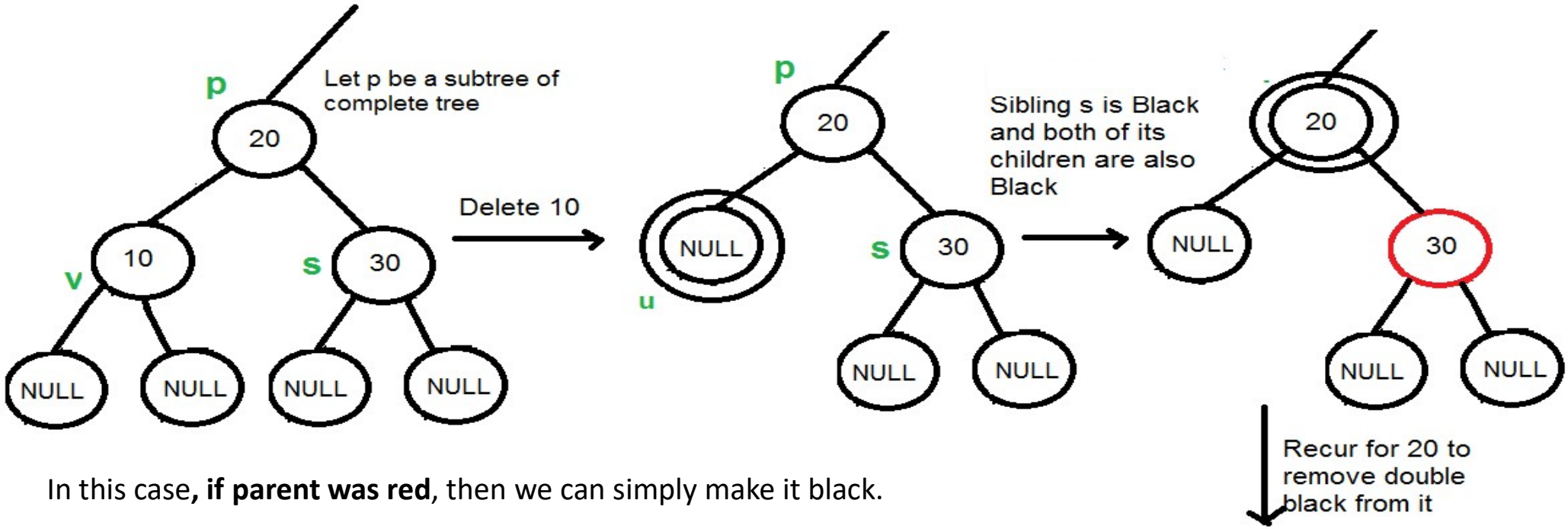
ii) parent p of **u is red.**

2.3) Color of sibling s is red: Perform Single Rotation and Recolor

Red Black Tree Deletion

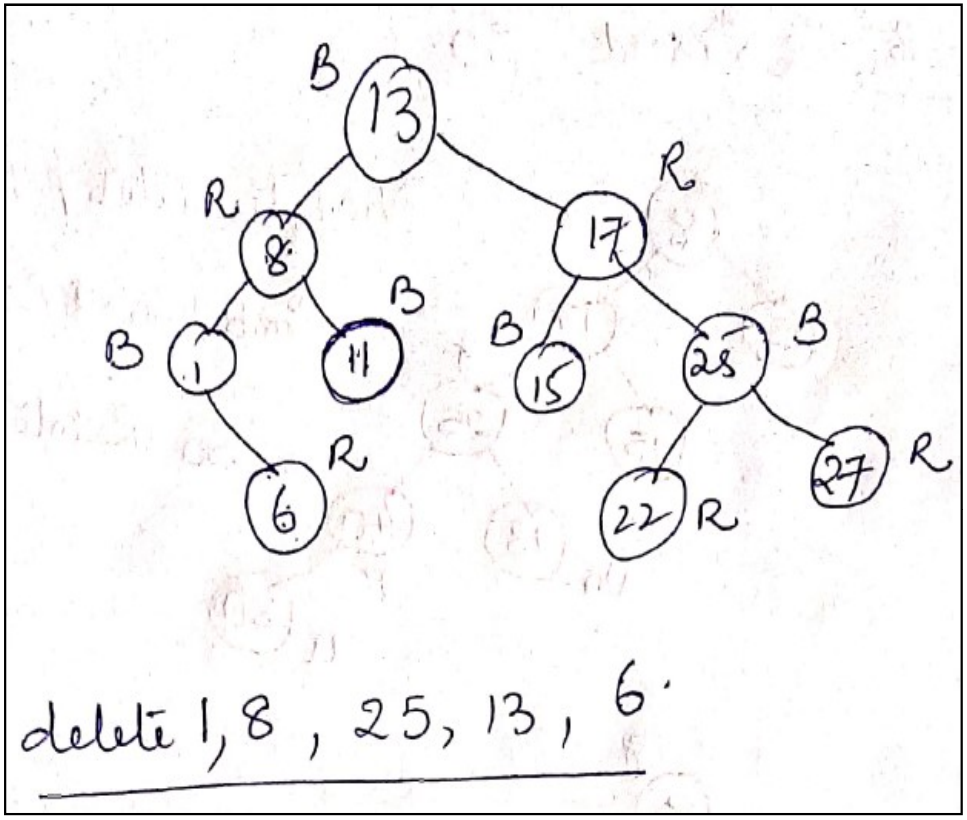
2.2) Color of s is black:

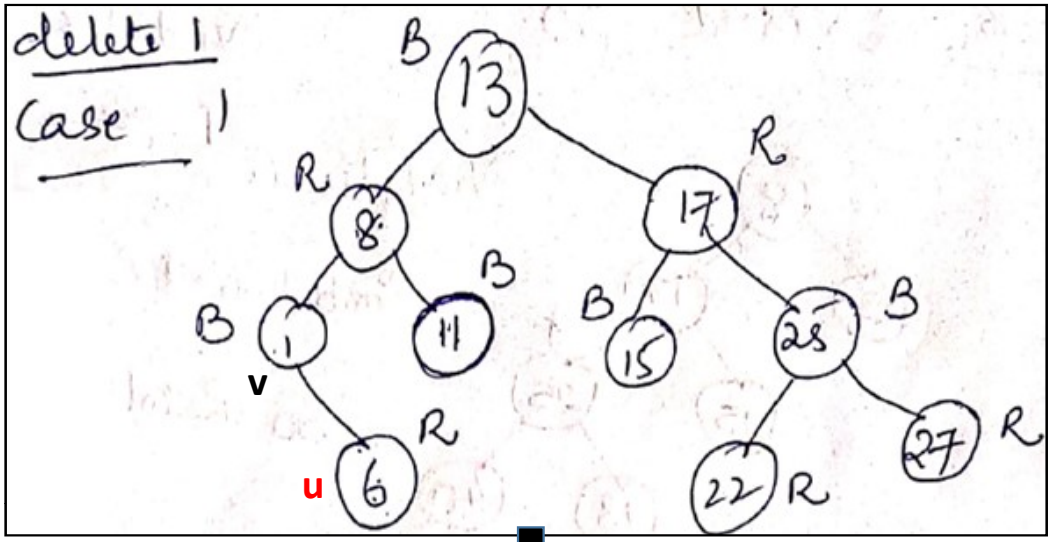
a) both children of s are also black: Pass one black color from u to p, and make s to red. So if p was red, it becomes black and we can exit. But if **p was black**, it becomes double black. So we need to apply other cases to balance the tree.

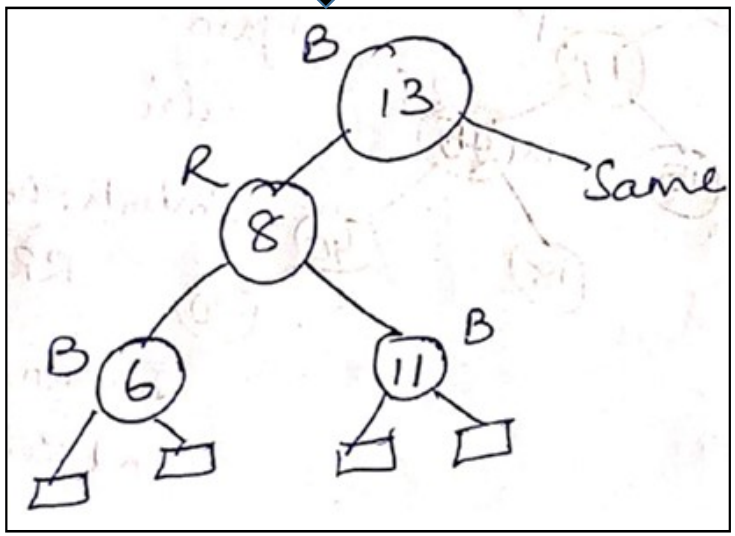


In this case**, if parent was red**, then we can simply make it black.

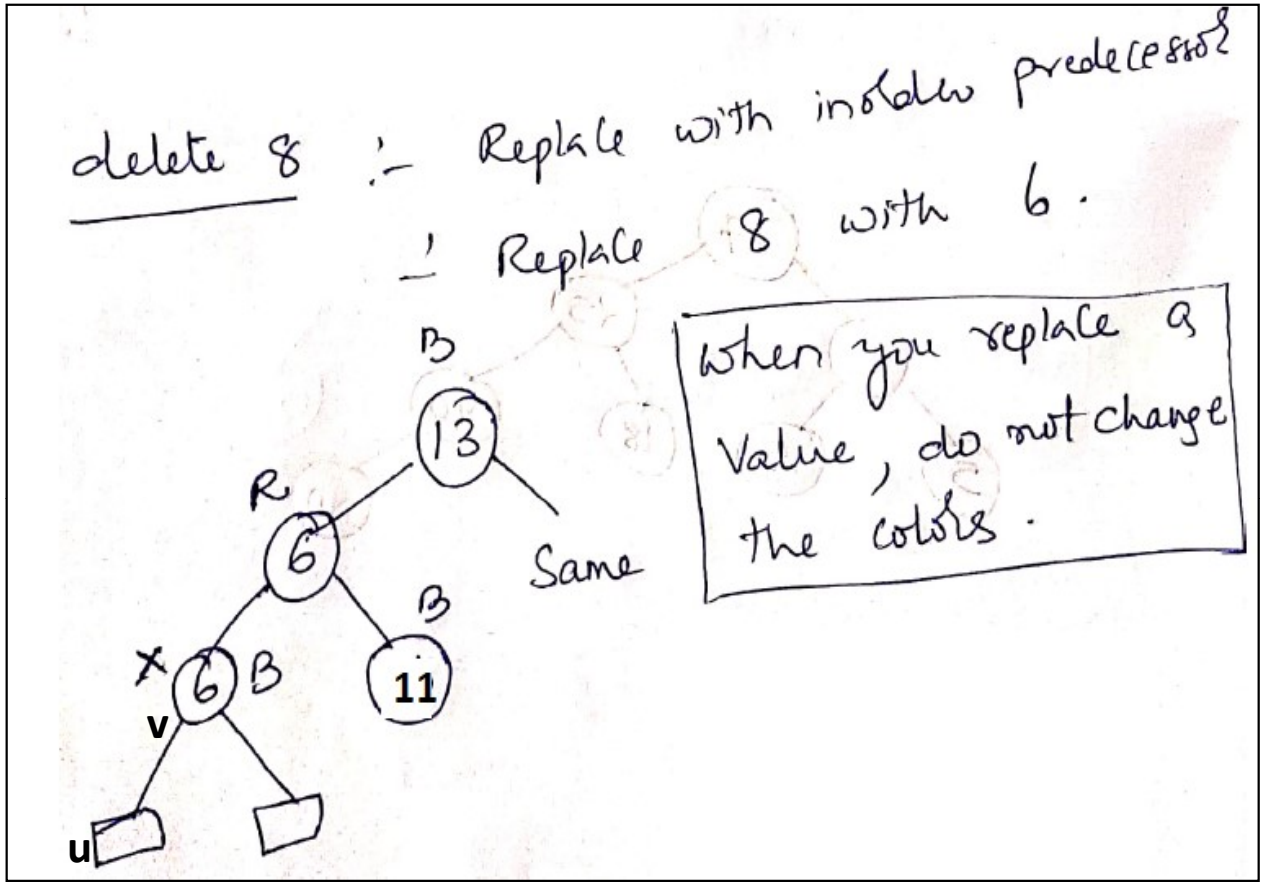
Red Black Tree Deletion : Example (delete 1,8,25,13,6)

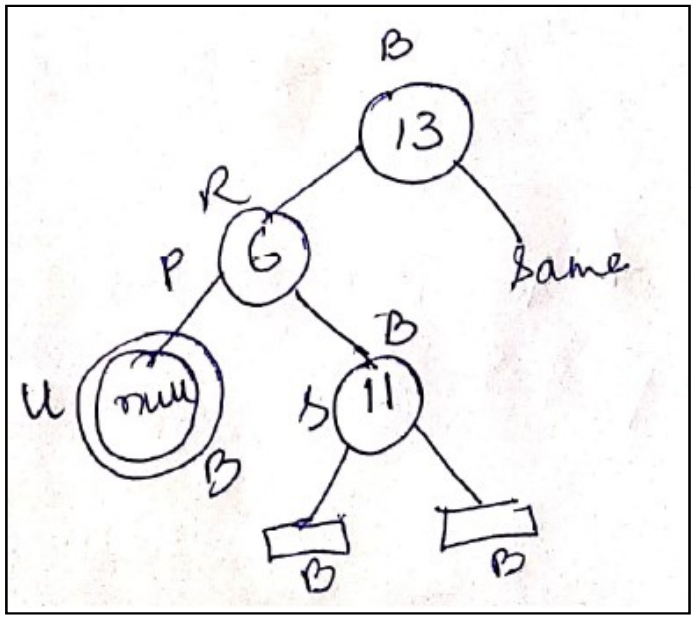






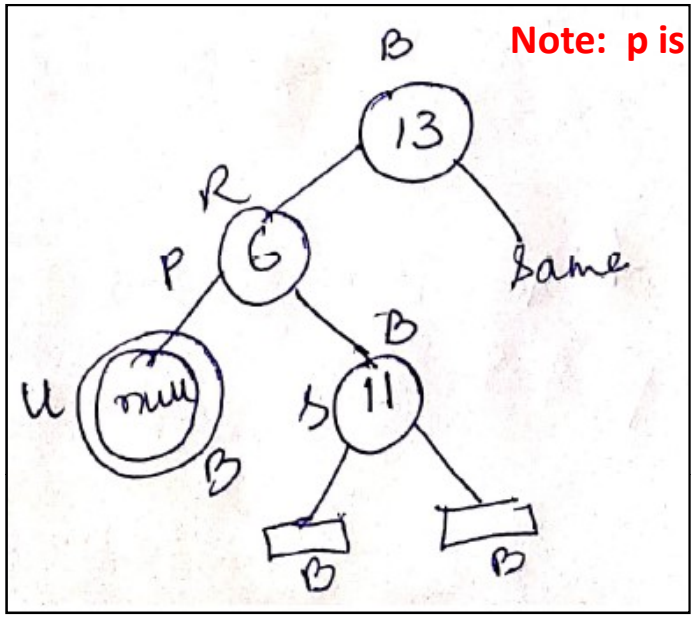
Red Black Tree Deletion : Example (delete 1,8,25,13,6)



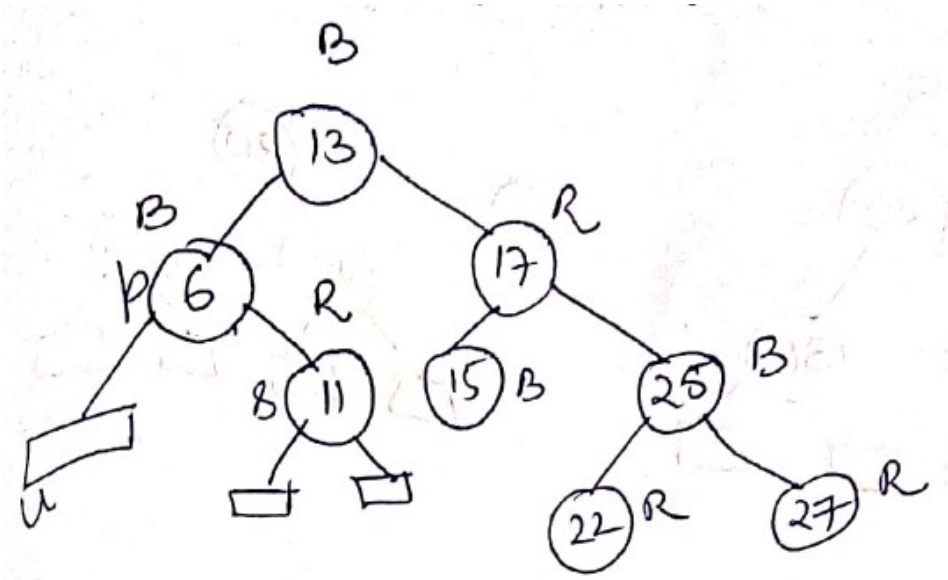


Case : 2.2 a) : Sibling is black and both its children are also black

Red Black Tree Deletion : Example (delete 1,8,25,13,6)

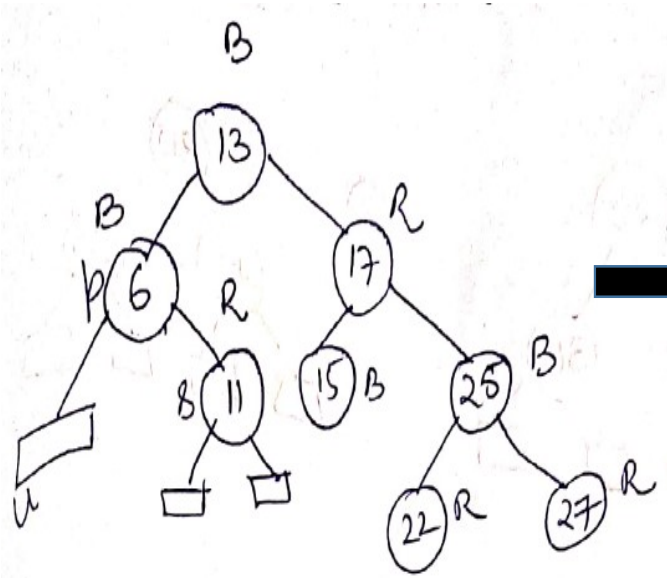


Note: p is red



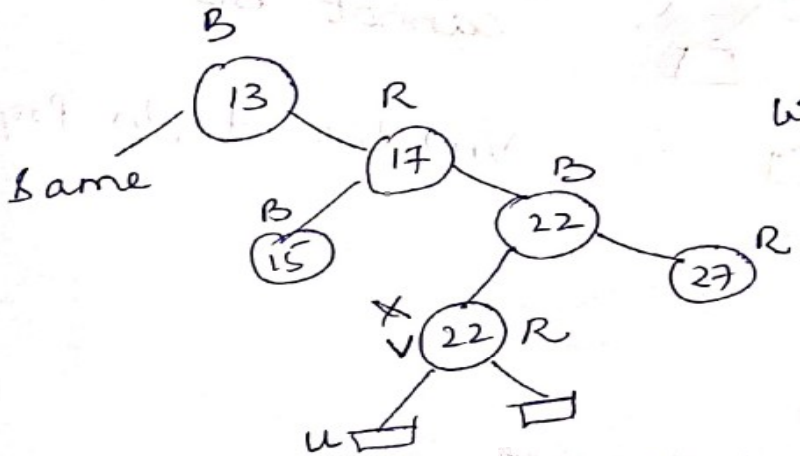
Case : 2.2 a) : Sibling is black and both its children are also black : Only Recolor

Red Black Tree Deletion :
Example (delete 1,8,25,13,6)



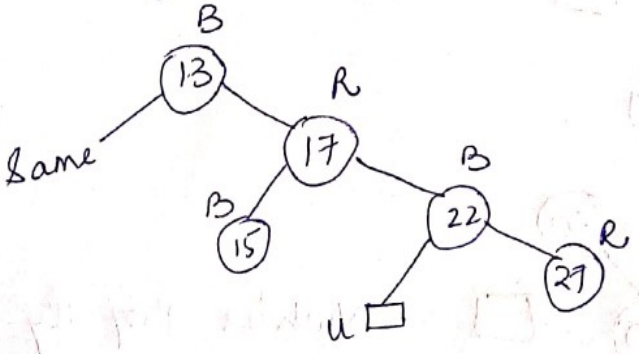
delete 25 :- Replace with inOrder predecessor

Replace 25 with 22



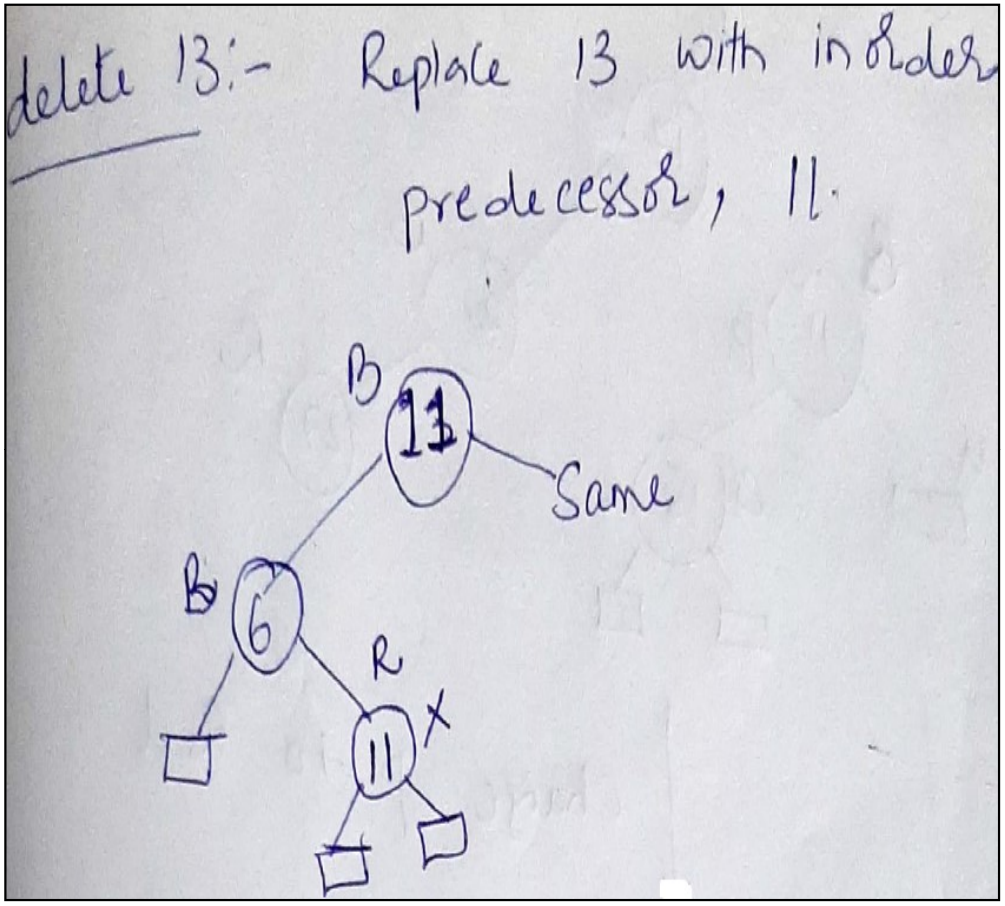
When deleting a Red node, simply delete it

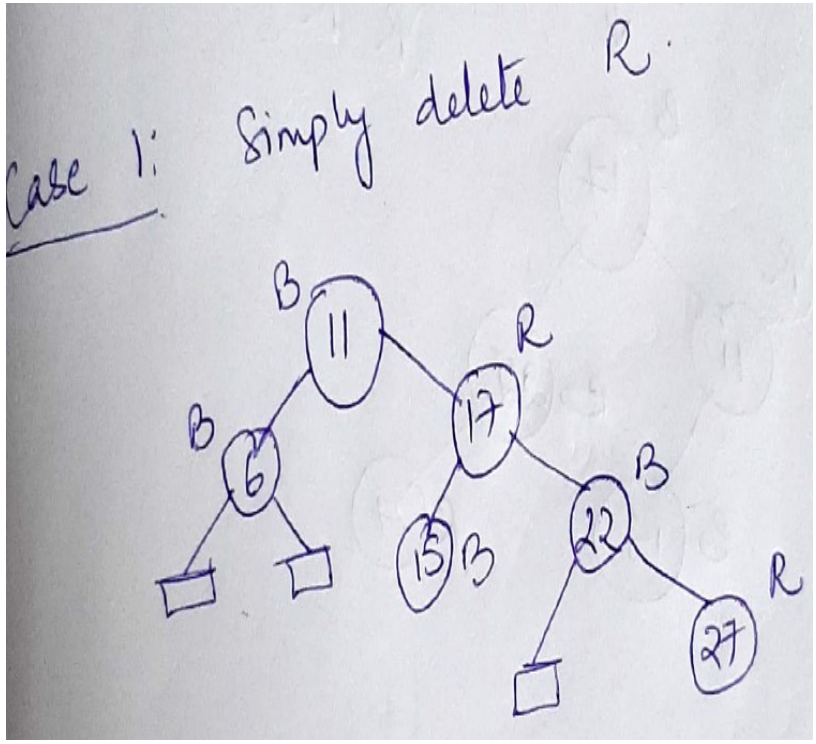
Case 1:-



Red Black Tree Deletion :

Example (delete 1,8,25,13,6)





Red Black Tree Deletion

2.2 b) Current node u is double black and it is not root. **Sibling s is black and at least one of sibling's children is red.**

This case is further divided into two sub cases based on whether **p is black or p is red.**

i) p is black: Perform rotation and recolor.

Let the **red** child of **s** be **r**. This case can be divided in four sub cases depending upon positions of **s and r**

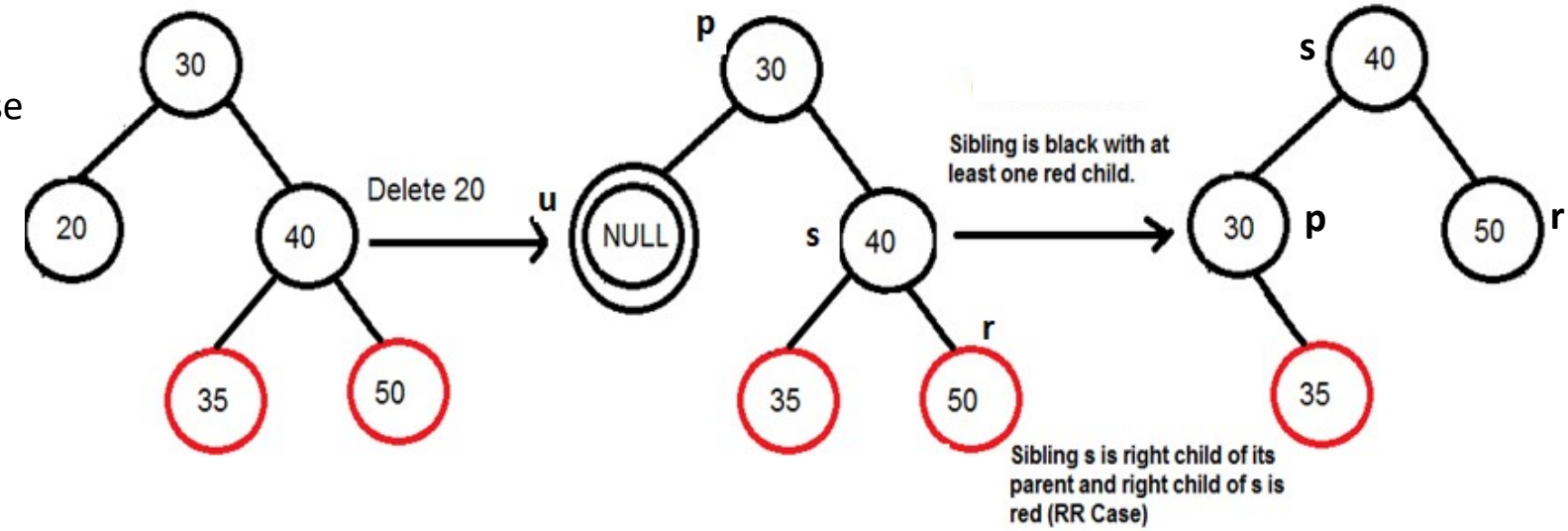
(i) LLbbr Case (s is left child of its parent and r is left child of s).

(ii)LRbbr Case(s is left child of its parent and r is right child).

(iii)RRbbr Case (s is right child of its parent and r is right child of)

(iv)RLbbr Case (s is right child of its parent and r is left child of s)

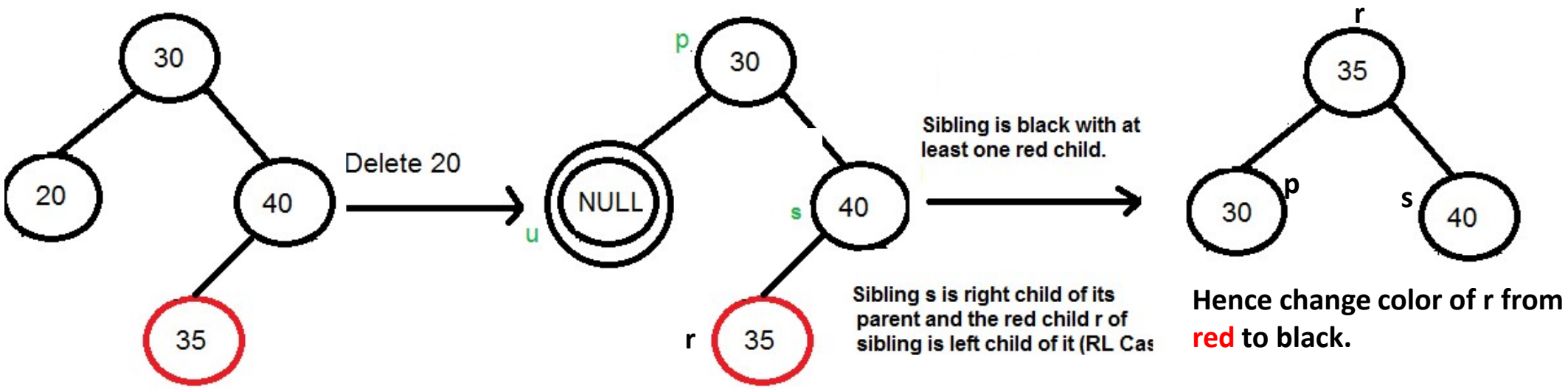
Ex for RRbbr Case



Change color of r from red to black.

Red Black Tree Deletion

Ex for RLbbr Case : Perform rotation.



Hence change color of r from red to black.

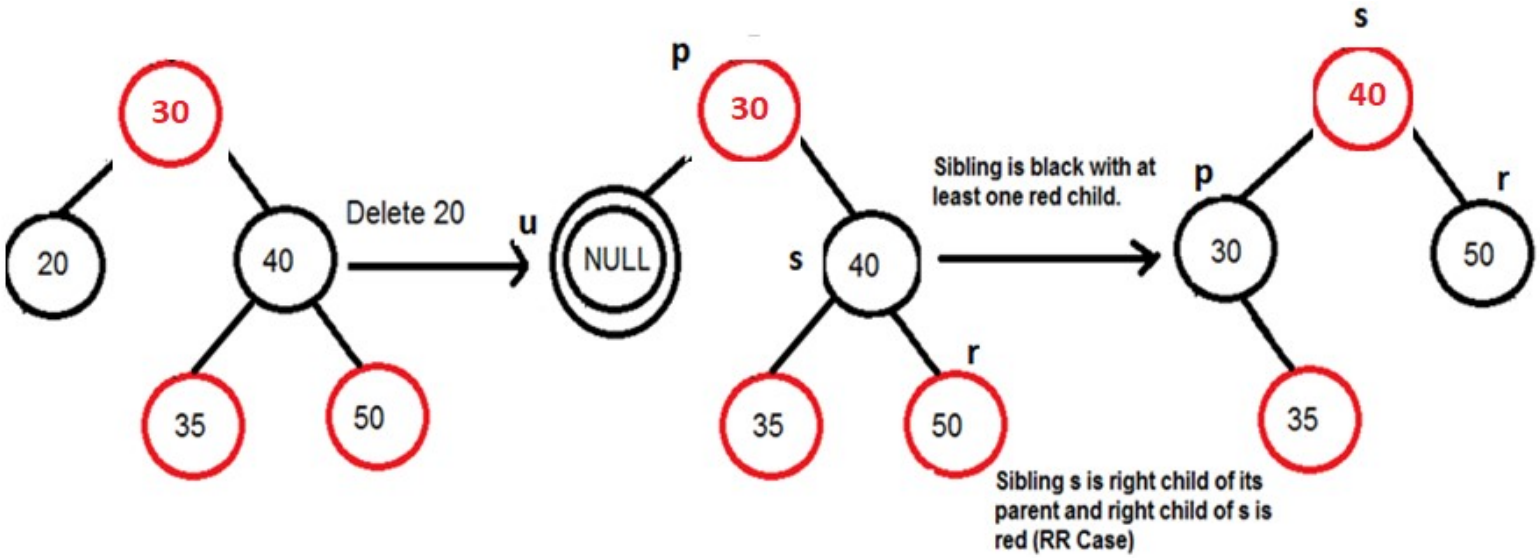
Red Black Tree Deletion

2.2 b) ii) Current node u is double black and it is not root. **Sibling s is black and at least one of sibling's children is red. and p is red: Perform rotation and recolor.**

Let the **red** child of **s** be **r**. This case can be divided in four sub cases depending upon positions of **s and r**

(i) LLrbr Case (s is left child of its parent and r is left child of s).
(ii)LRrbr Case(s is left child of its parent and r is right child).
(iii)RRrbr Case (s is right child of its parent and r is right child of)
(iv)RLrbr Case (s is right child of its parent and r is left child of s)

Ex for RRrbr Case

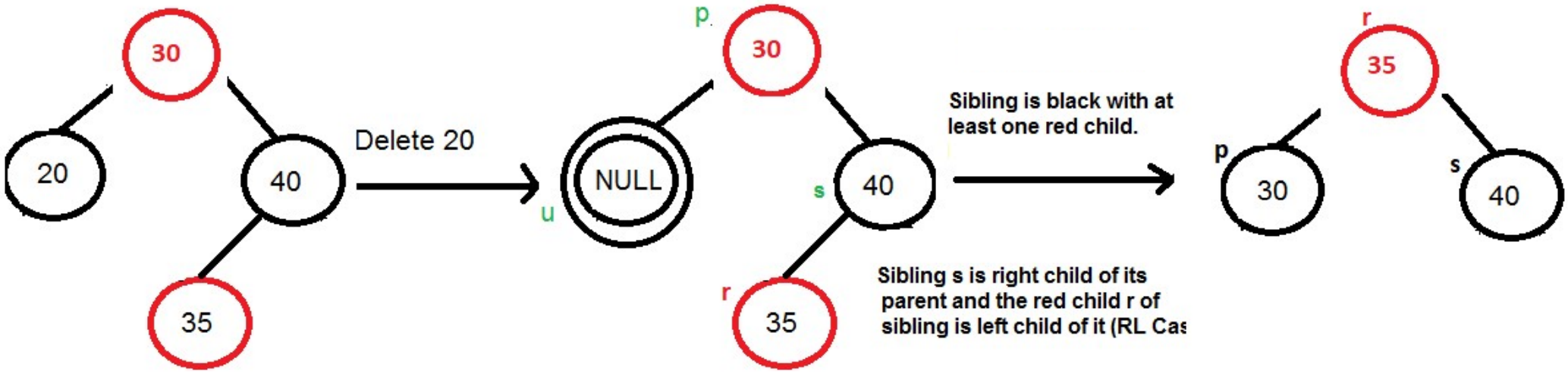


Remember we are deleting from a valid RBT, so parent of **p** should have been black in color, we will not be having two consecutive red nodes even before rotation and after rotation.

Change color of p from red to black.
Change color of s from black to red
Change color of r from red to black

Red Black Tree Deletion

Ex for RLrbr Case : **Perform rotation and recolor.**



Hence change color of p from red to black.

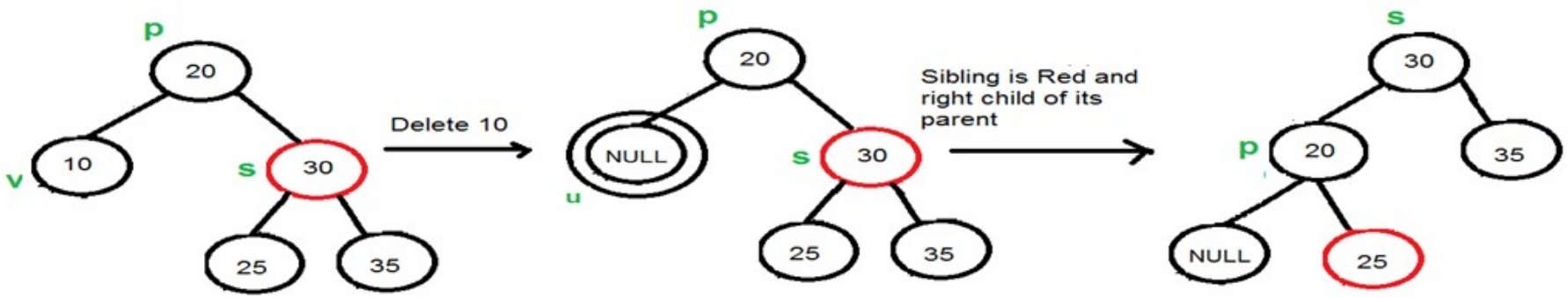
Red Black Tree Deletion

Case 2.3) **Sibling s is red:** **perform a rotation**, recolor

This case can be divided in two subcases.

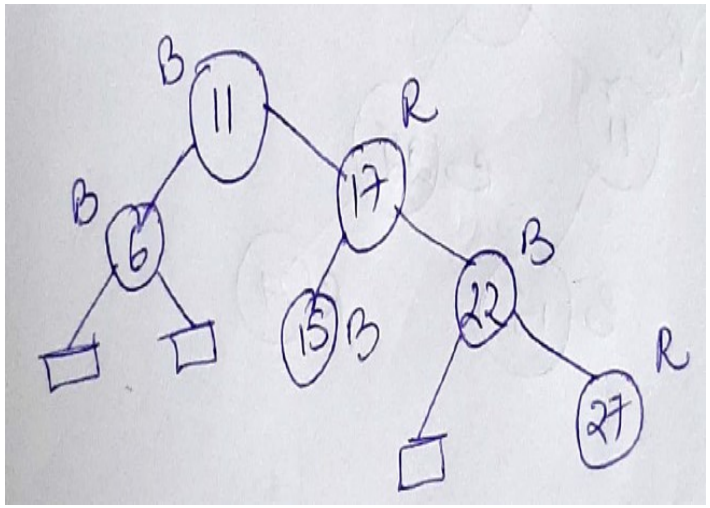
(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

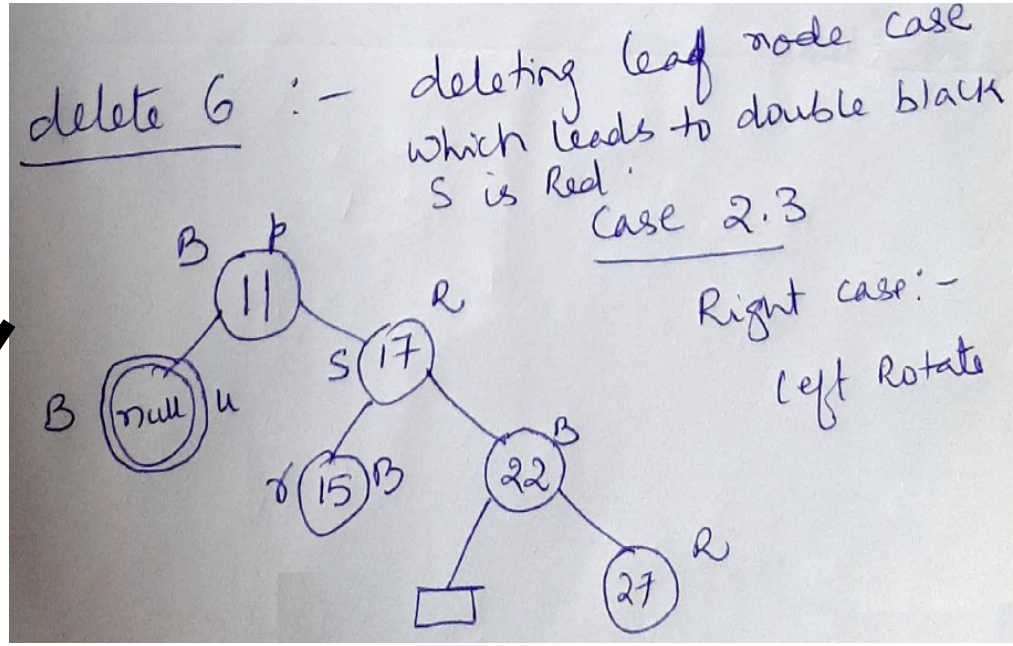
(ii) Right Case (s is right child of its parent). We left rotate the parent p.

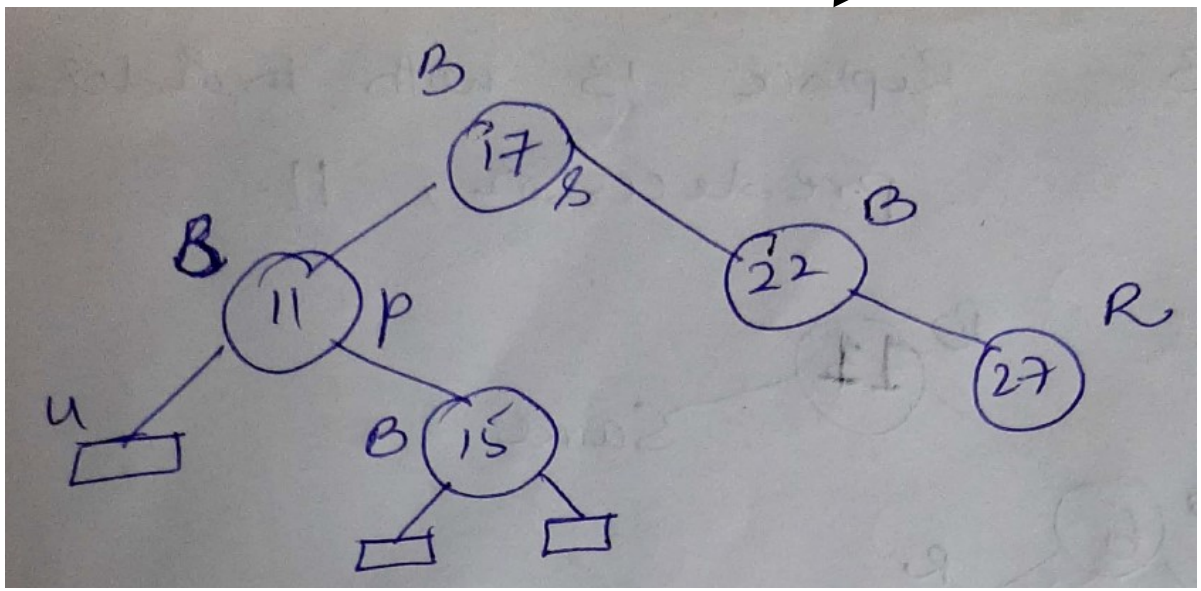


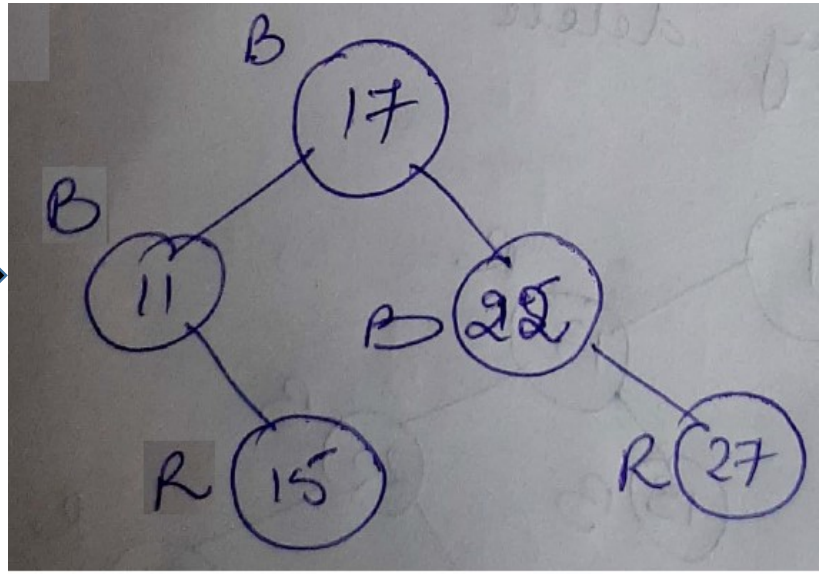
Recolor **s to black and p's** **new child to red**

Red Black Tree Deletion : Example
(delete 1,8,25,13,6)









Exercise: Red Black Tree Deletion : (delete 35, 40, 11, 20, 5, 12)