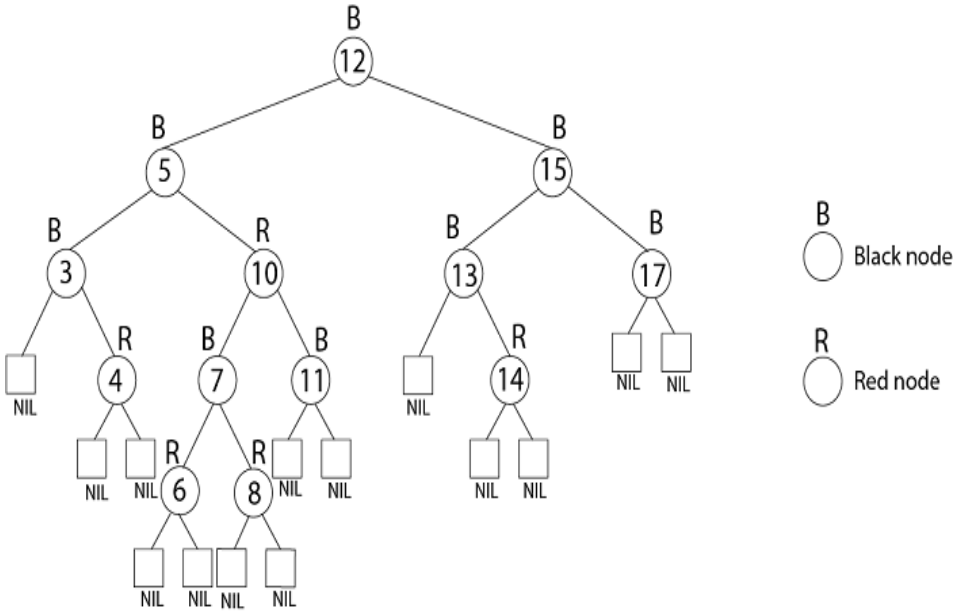
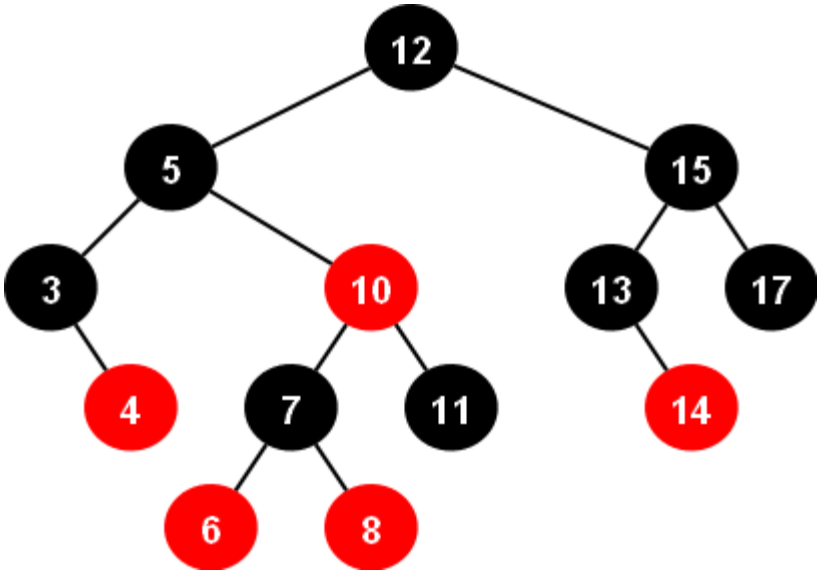
RED - BLACK TREE

- A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black).
- These colours are used to ensure that the tree remains balanced during insertions and deletions.
- A red-black tree has the following *red-black properties*:

1. Root node is **black**
2. Every node is either red or black.
3. Every leaf (NIL) is **black**. (This factor that every non-NIL node has two children.)
4. If a node is red, then both its **children are black**.(**BLACK- children rule**)
5. Every path from a node (including root) to any of its descendant NIL node has the same number of black nodes.(**Black-height rule**)





OPERATIONS

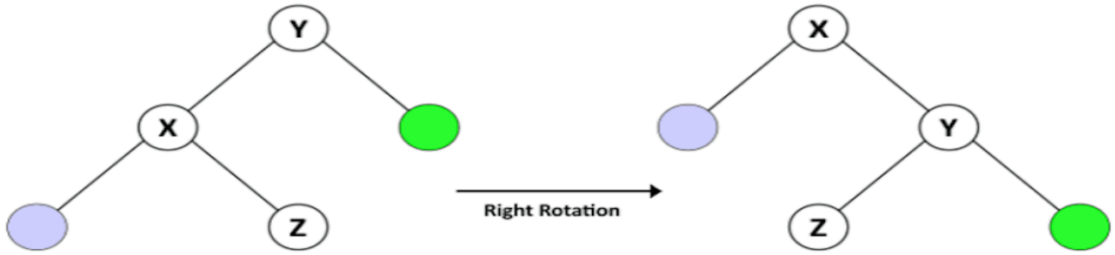
1. Insertion
2. Deletion

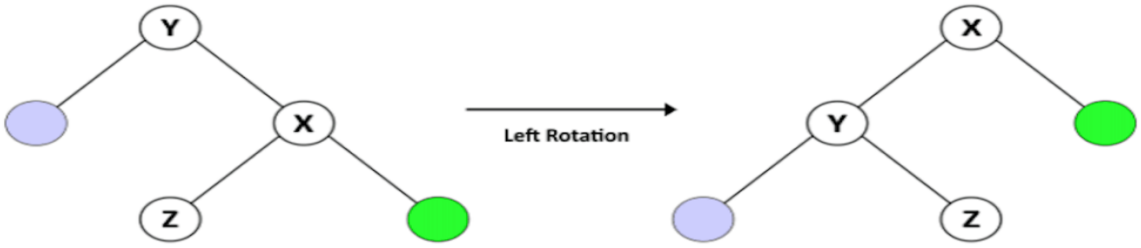
- The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree may violate the red-black properties.
- We used rotation as a tool to do balancing after insertion.
- In the Red-Black tree, we use two tools to do the balancing.
  - ✓ **Recoloring**
  - ✓ **Rotation**
- **Recolouring** is the **change in colour of the node** i.e. if it is **red then changes it to black and vice versa.**

Rotation

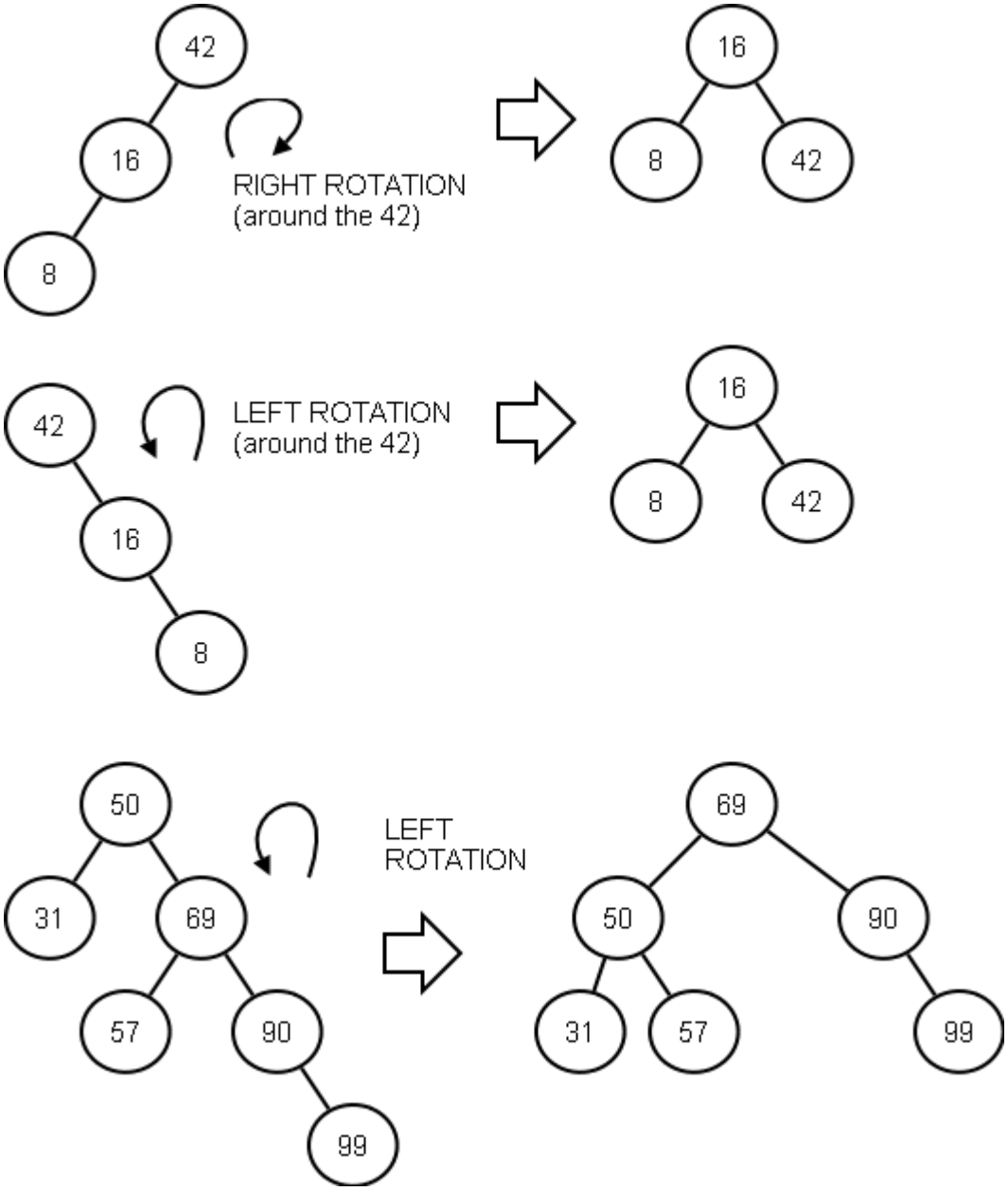
- Restructuring operations on red-black trees can generally be expressed more clearly in details of the rotation operation.
- If we start with a BST and only restructure using rotation, then we will still have a BST i.e. rotation do not break the BST-Property.

RIGHT-ROTATE(T,X)

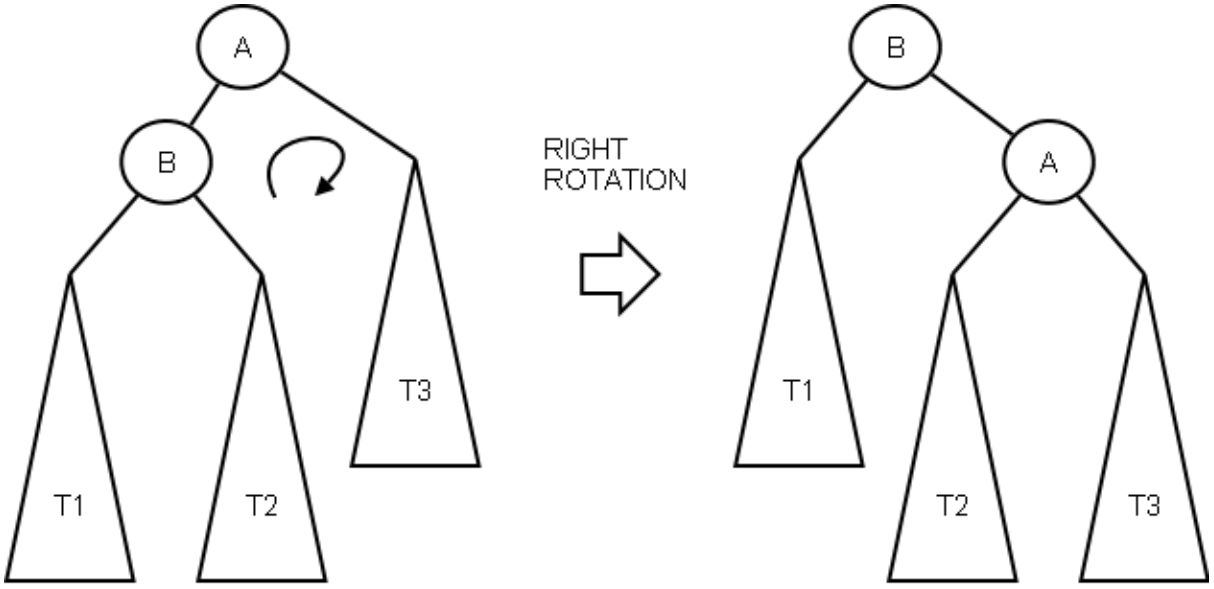

Y
X
Z
Right Rotation
X
Y
Z

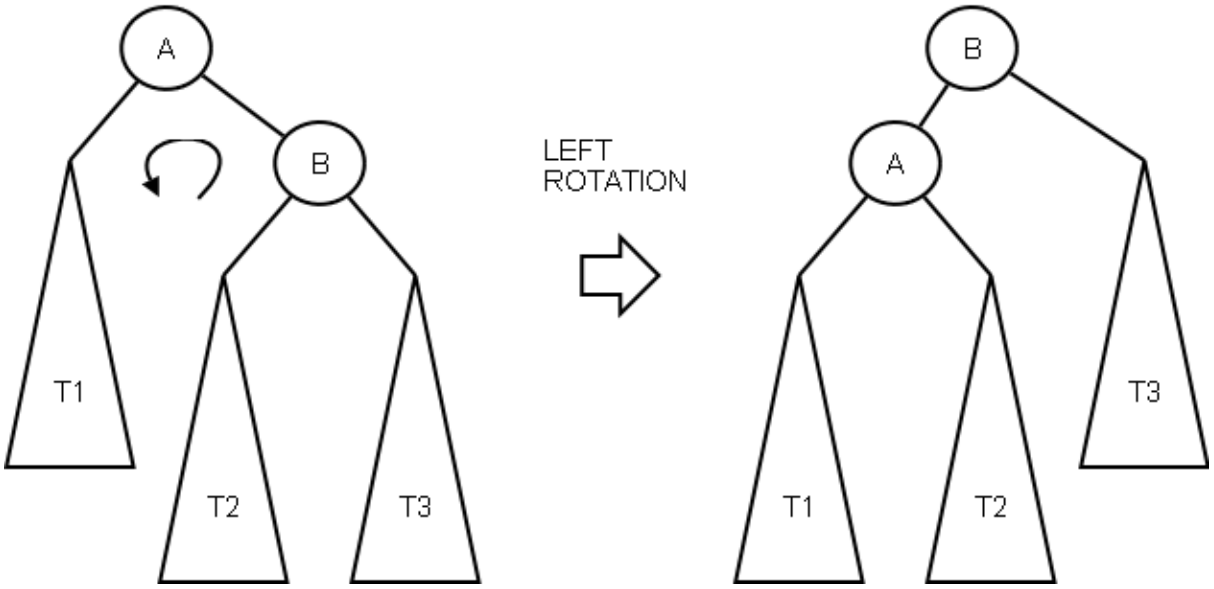

Y
X
Z
Left Rotation
X
Y
Z

EXAMPLE



In general


A
B
T1
T2
T3
RIGHT
ROTATION
B
A
T1
T2
T3


A
B
T1
T2
T3
LEFT
ROTATION
B
A
T1
T2
T3

RED-BLACK Tree-INSERTION

- In a **Red-Black** Tree, every new node must be inserted with the color RED.
- The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree.
- But it is inserted with a color property.
- 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 the following operation to make it Red Black Tree.

  1. Recolor

  2. Rotation

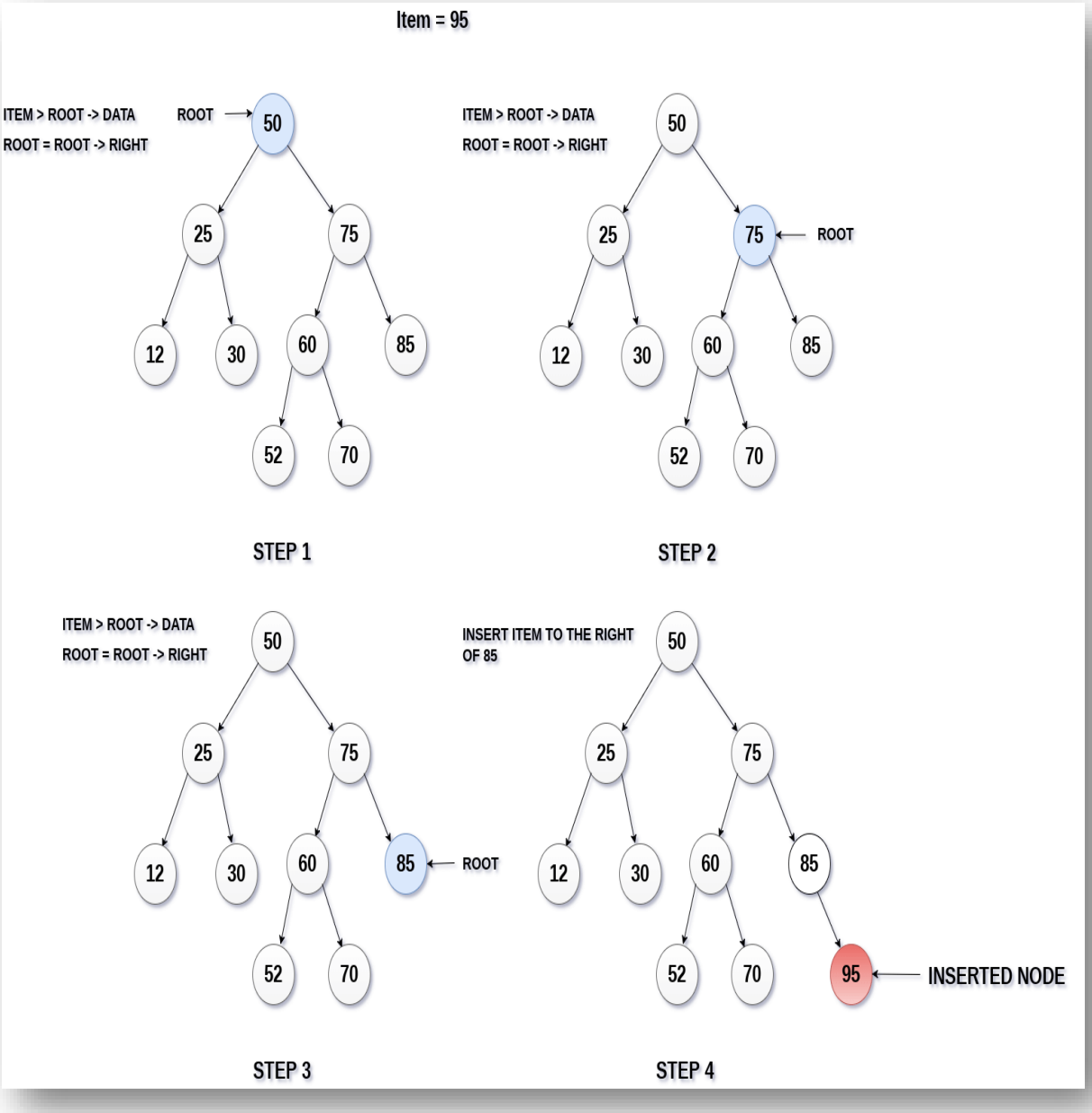
  3. Rotation followed by Recolor

Insertion of a key

A new key is always inserted at the leaf.

We start searching a key from the root until we hit a leaf node.

Once a leaf node is found, the new node is added as a child of the leaf node.

- 

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**.

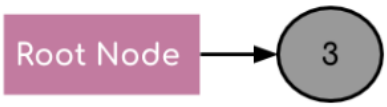
Step 4 - If the parent of newNode is Black then exit from the operation.

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

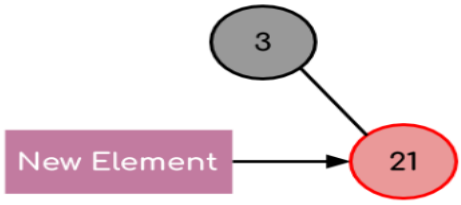
Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

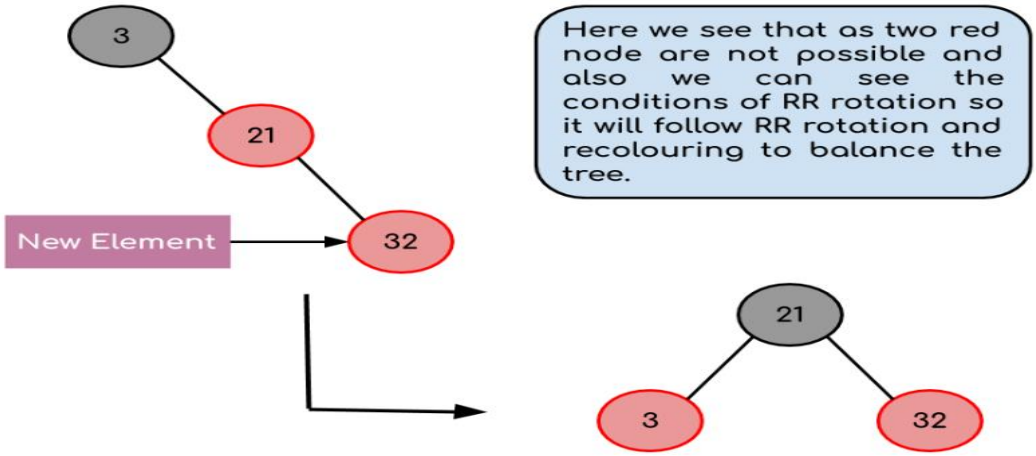
Step 1: Inserting element 3 inside the tree.



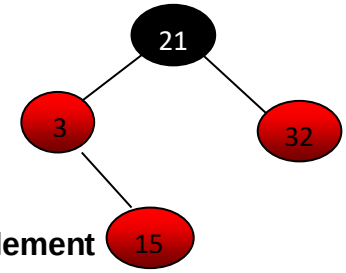
Step 2: Inserting element 21 inside the tree.



Step 3: Inserting element 32 inside the tree.



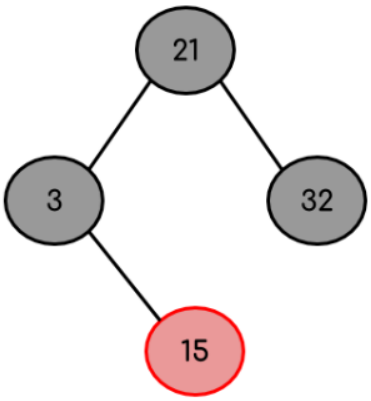
Step 4: Inserting element 15 inside the tree



New element

- As Two Red nodes are not possible.
- Check the color of parent's sibling
- It is red so perform recolor

Final Tree Structure:



RED-BLACK TREE -DELETION

- This operation removes a node from the tree.
- The main property that violates after insertion is two consecutive reds.
- 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.

- Deleting a node may or may not disrupt the red-black properties of a red-black tree.
- If this action violates the red-black properties, then a fixing algorithm is used to regain the red-black properties.

- Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.
- In insert operation, we check color of uncle to decide the appropriate case. In delete operation, we check color of sibling to decide the appropriate case.
- Deletion is fairly complex process. To understand deletion, notion of double black is used.
- When a black node is deleted and replaced by a black child, the child is marked as **double black**.
- The main task now becomes to convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

1) Perform standard BST delete.

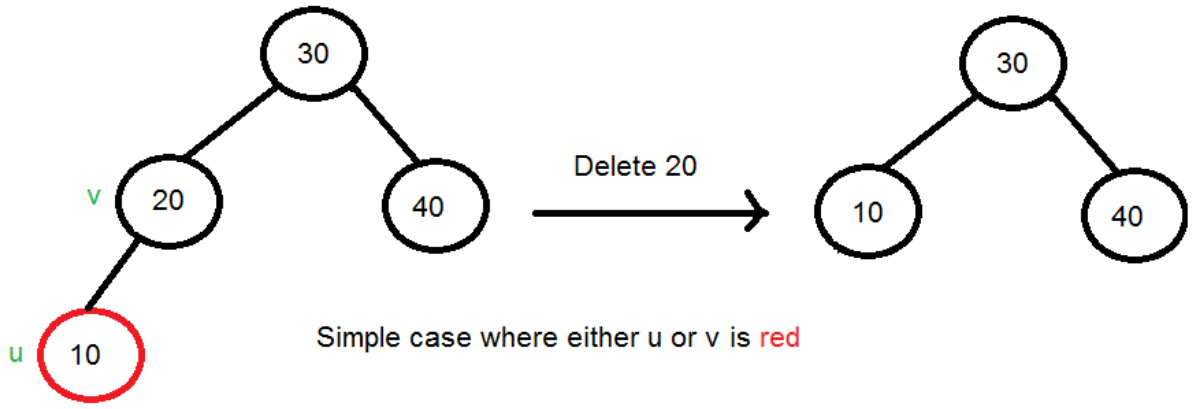
- When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal

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 where a node is 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).

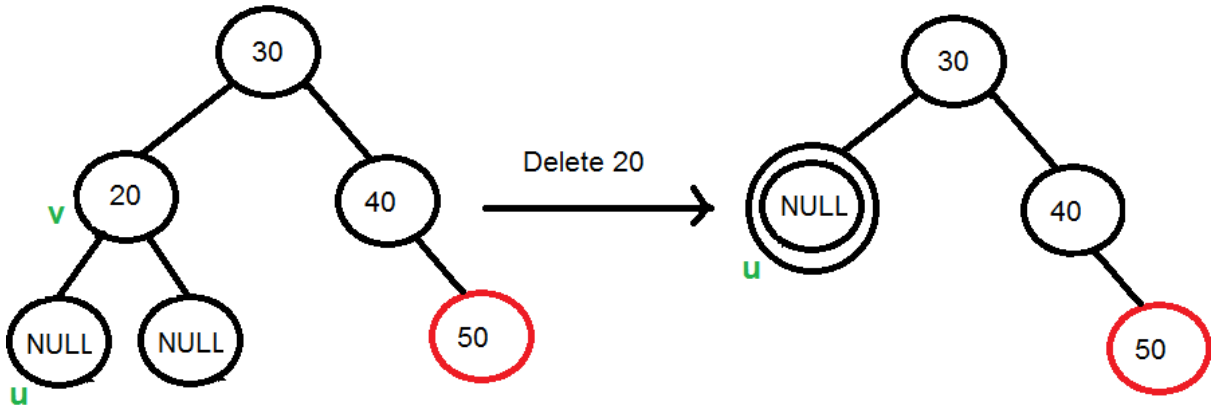
2) Simple Case: If either u or v is red

- We mark the replaced child as black (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.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that if v is leaf, then u is NULL and 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

3.2) Do following while the current node u is double black and it is not root. Let sibling of node be **s**.

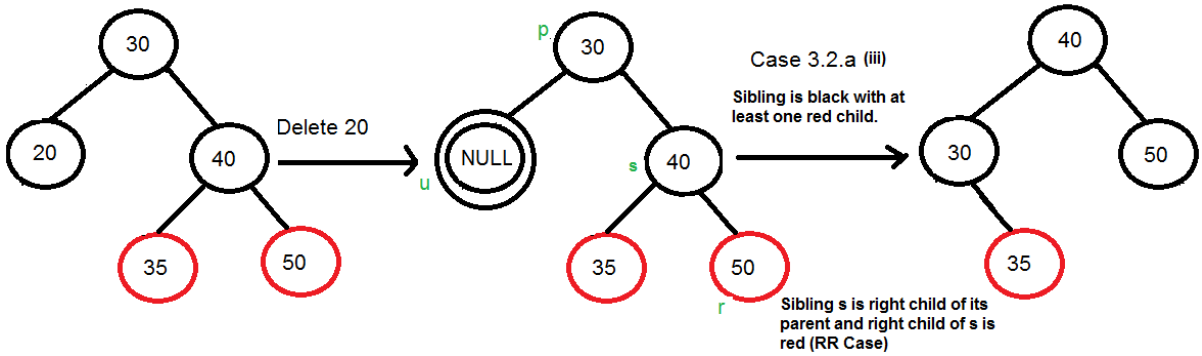
(a): If sibling s is black and at least one of sibling's children is red, perform rotation(s).

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

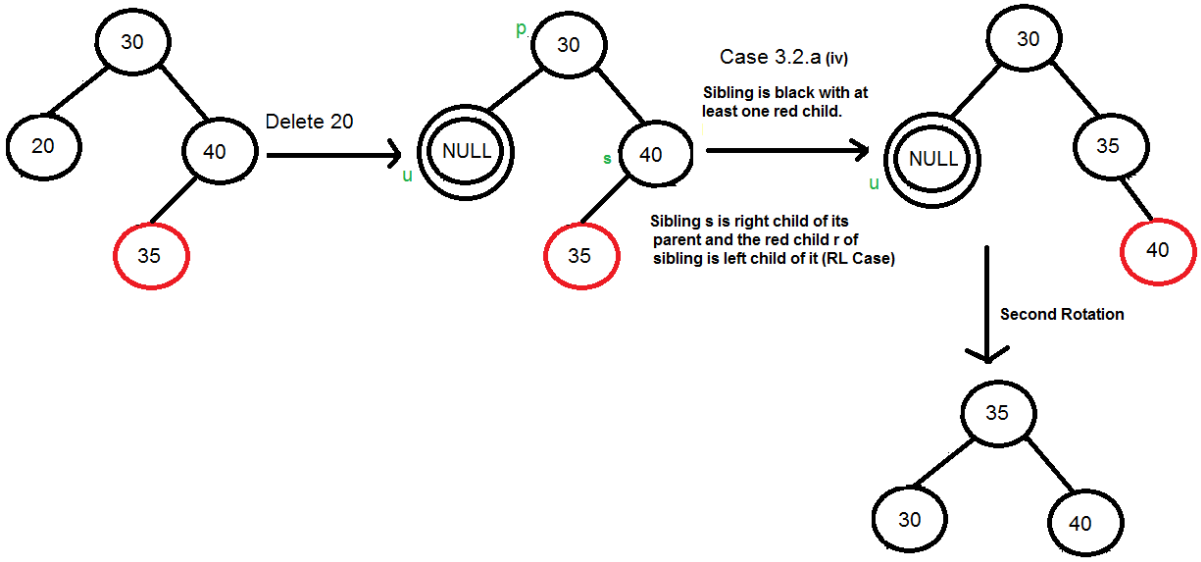
(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

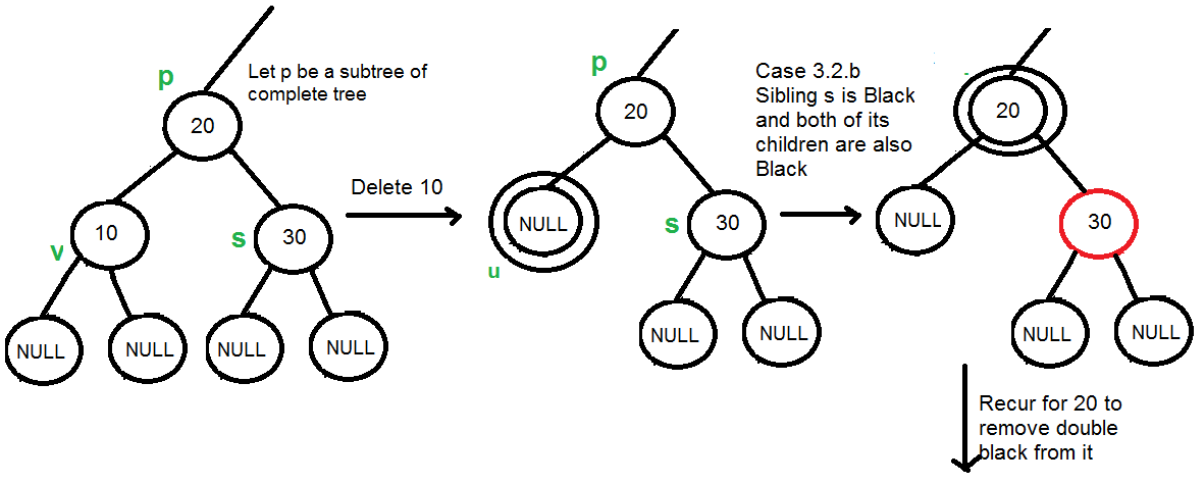
(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



(IV) **Right Left Case** (s is right child of its parent and r is left child of s)



(b): If sibling is black and it's both children are black, perform recoloring, and recur for the parent if parent is black.



- In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent.

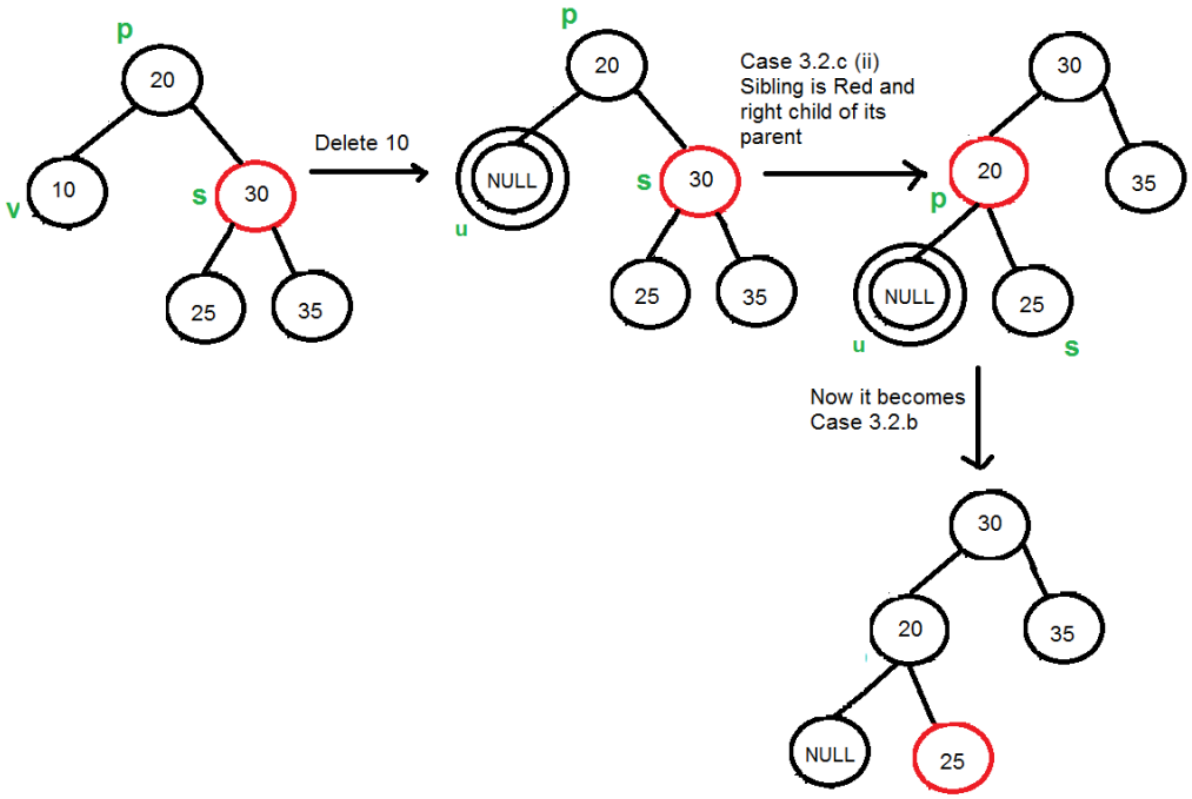
- The new sibling is always black (See the below diagram).
- This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two sub cases.

(i) Left Case (s is left child of its parent).

- This is mirror of 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.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).