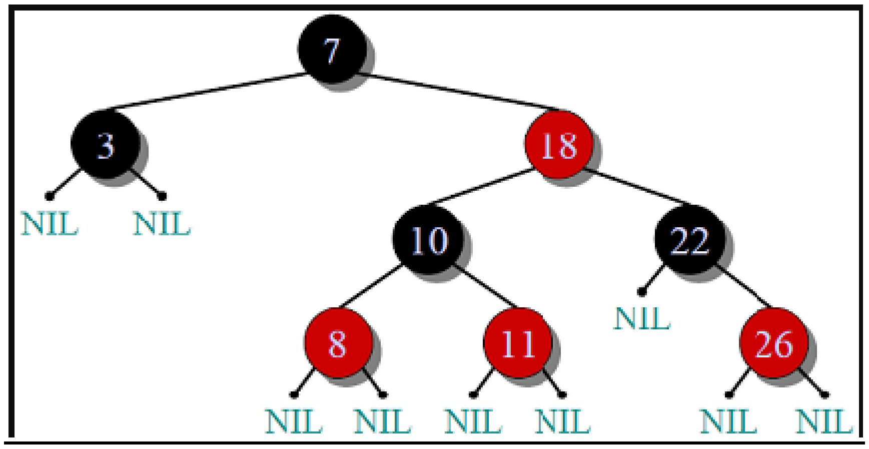
Unit-2

RED – BLACK TREE :

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

1. Every node has a color either red or black.
2. Root of tree is always black.
3. Every Leaf node is Null / Nil node and black.
4. Both children of red node are black.
5. Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.
6. No root to Leaf path contains two consecutive red nodes.
7. For each node x, all paths from x to descendent leaves contain the same number of black nodes (not counting x). This number is the black height of the node x , denoted by bh(x).



Need for Red Black tree:

- Most of the Binary Search Tree operations take **O(h)** time (where ‘**h**’ is the height of the tree) for example: Search , Max , Min , Insert , Delete etc.
- If Binary Search Tree becomes skewed then height of the tree will become equal to total number of nodes i.e. **‘n’** and complexity will increase to **O(n)**.

- So to make the complexity low we should balance the Binary Search Tree after each insertion and deletion operation. This will ensure the height **h** of tree as **log n** and complexity as **O(log n)**.
- So the height of a Red Black Tree is always **log n**.
- If frequent insertion and deletion are required then Red Black Tree give better performance than AVL Tree. If insertion and deletion are less frequent then AVL tree give good performance because AVL Trees are more balanced than Red Black Trees but they can cause more rotation and can increase time complexity.

<u>Properties :</u>

- In a red black tree of height 'h' has black height bh(x) >=h/2 from any node x.
- In a red black tree with 'n' nodes has height h <= 2log(n+1).

<u>Insertion in Red Black Tree :</u>

In Red-Black tree, we use two tools to do balancing.

- Recoloring
- Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithm has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

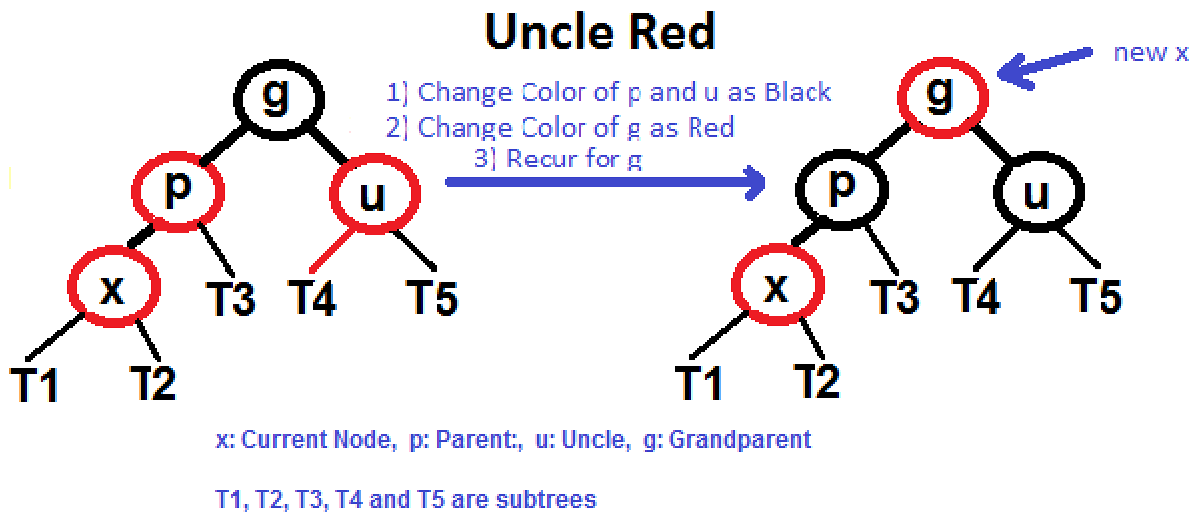
Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

1. Perform standard BST insertion and make the color of newly inserted nodes as RED.
2. If x is root, change color of x as BLACK (Black height of complete tree increases by 1).

3. Do following if color of x's parent is not BLACK **and** x is not root.

3.1. **If x's uncle is RED** (Grand parent must have been black)
(i) Change color of parent and uncle as BLACK.
(ii) color of grand parent as RED.
(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.

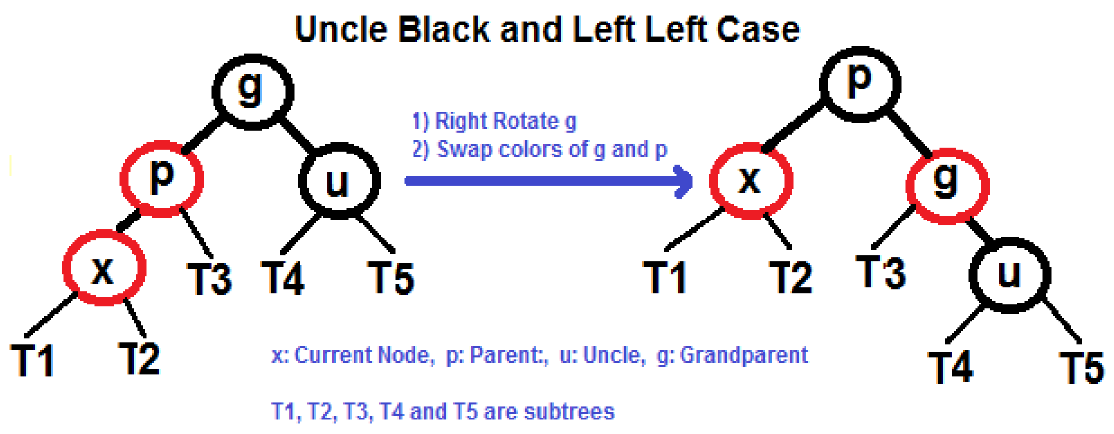

Uncle Red
1] Change Color of p and u as Black
2] Change Color of g as Red
3] Recur for g
new x
g
p
u
x
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

3.2. **If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to
i) Left Left Case (p is left child of g and x is left child of p)
ii) Left Right Case (p is left child of g and x is right child of p)
iii) Right Right Case (Mirror of case i)
iv) Right Left Case (Mirror of case ii)

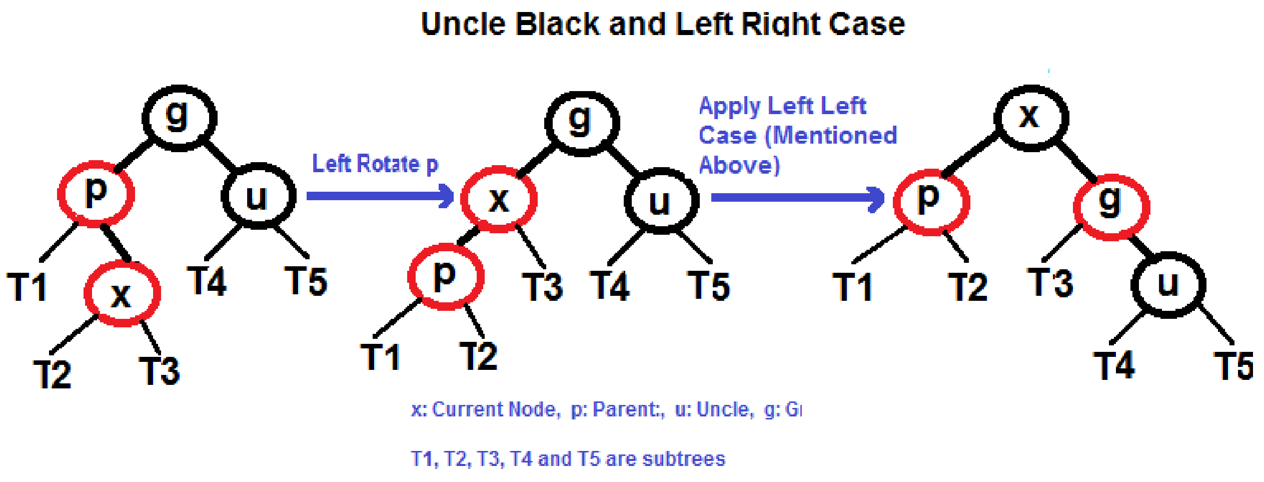
Following are operations to be performed in four subcases when uncle is BLACK.

All four cases when Uncle is BLACK

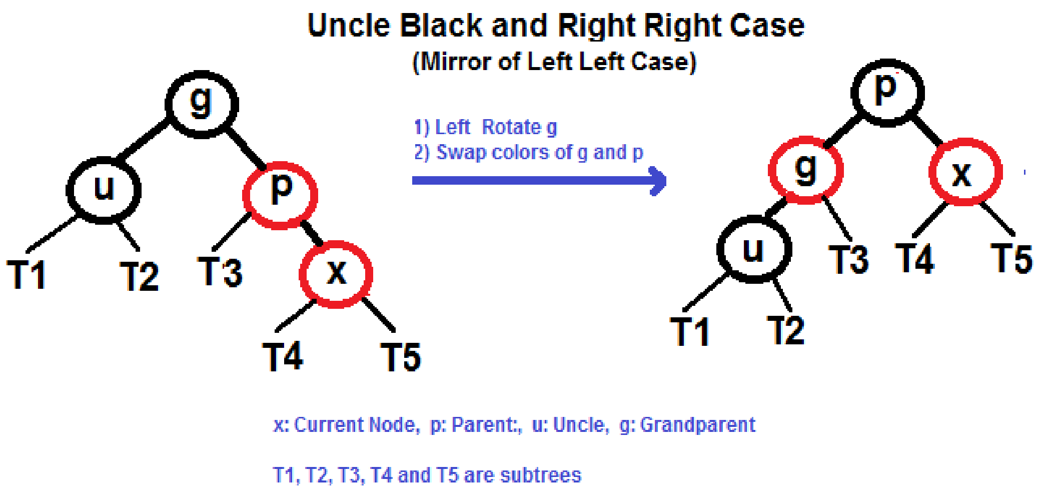
Left Left case (see g , p and x)


Uncle Black and Left Left Case
g
p
u
x
T1
T2
T3
T4
T5
1) Right Rotate g
2) Swap colors of g and p
p
x
g
u
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

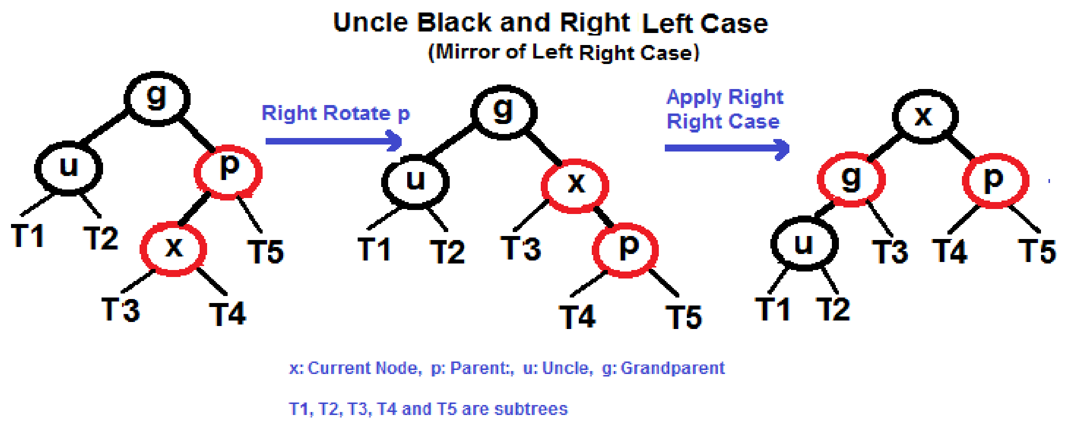
Left right case (See g, p and x)


Uncle Black and Left Right Case
g
p
u
x
T1
T2
T3
T4
T5
Left Rotate p
g
x
u
p
T1
T2
T3
T4
T5
Apply Left Left Case (Mentioned Above)
x
p
g
u
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Gr
T1, T2, T3, T4 and T5 are subtrees

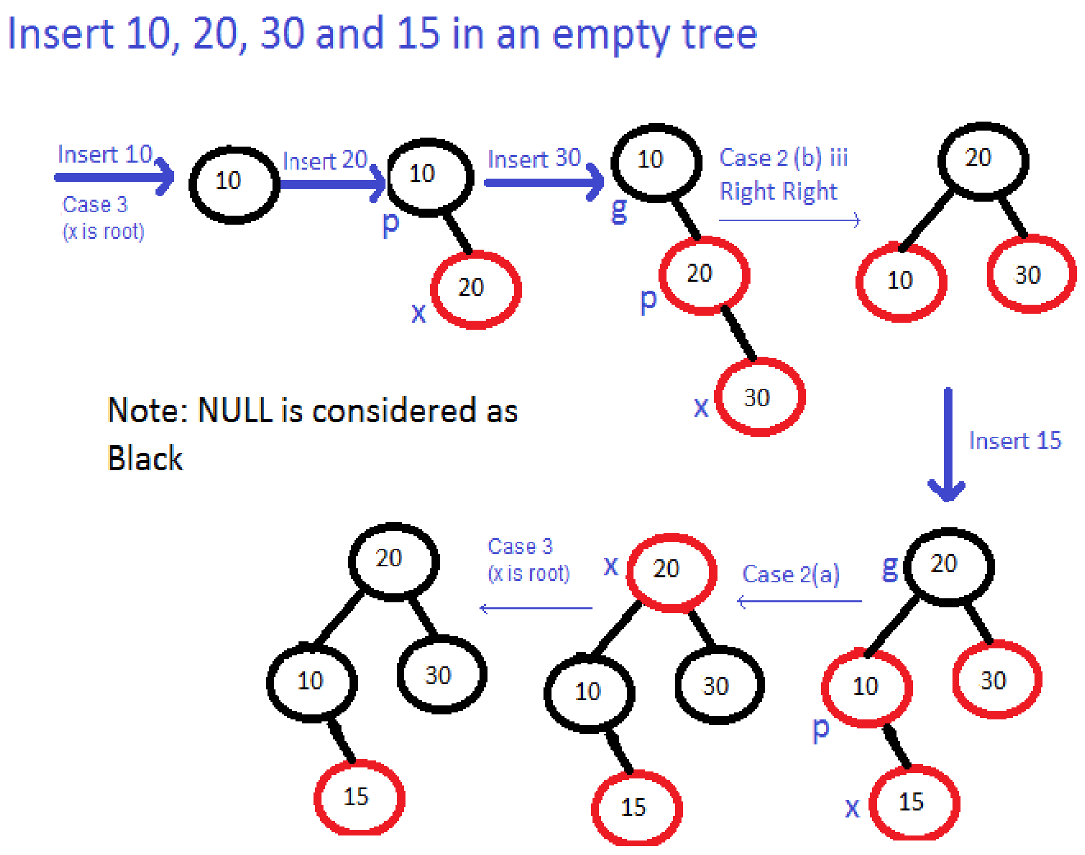
Right Right Case (See g, p and x)


Uncle Black and Right Right Case
(Mirror of Left Left Case)
g
u
p
x
T1
T2
T3
T4
T5
1) Left Rotate g
2) Swap colors of g and p
p
g
x
u
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

Right Left Case (See g, p and x)


Uncle Black and Right Left Case
(Mirror of Left Right Case)
Right Rotate p
Apply Right Right Case
g
u
p
x
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

Example of Insertion



Deletion in Red Black Tree:

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.

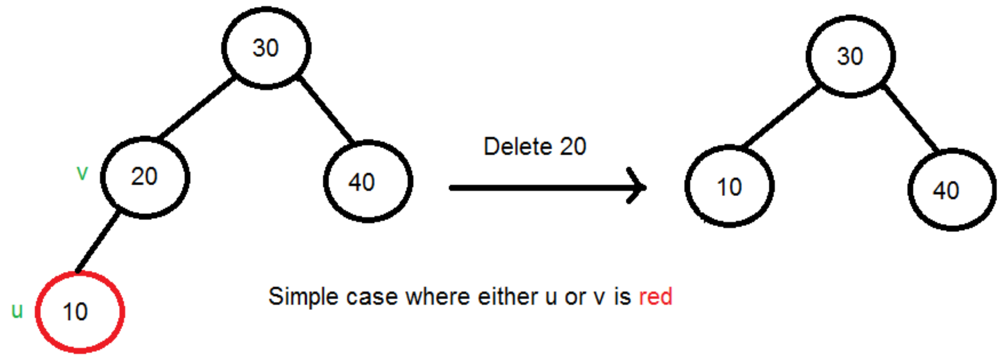
The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

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.

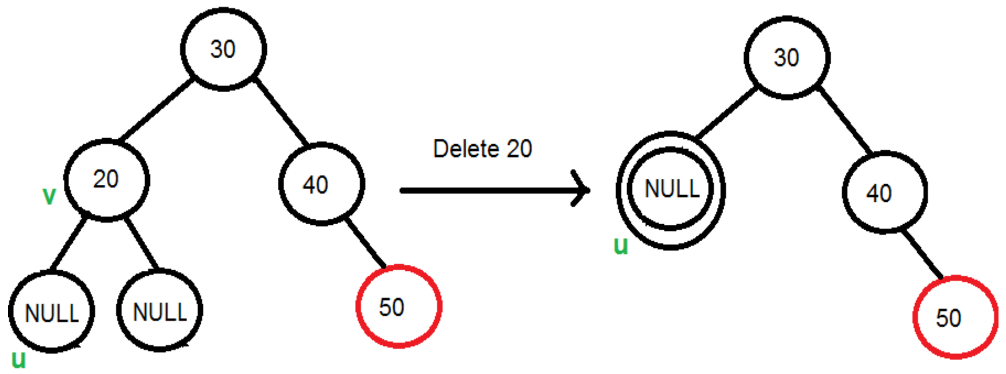
Deletion Steps

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

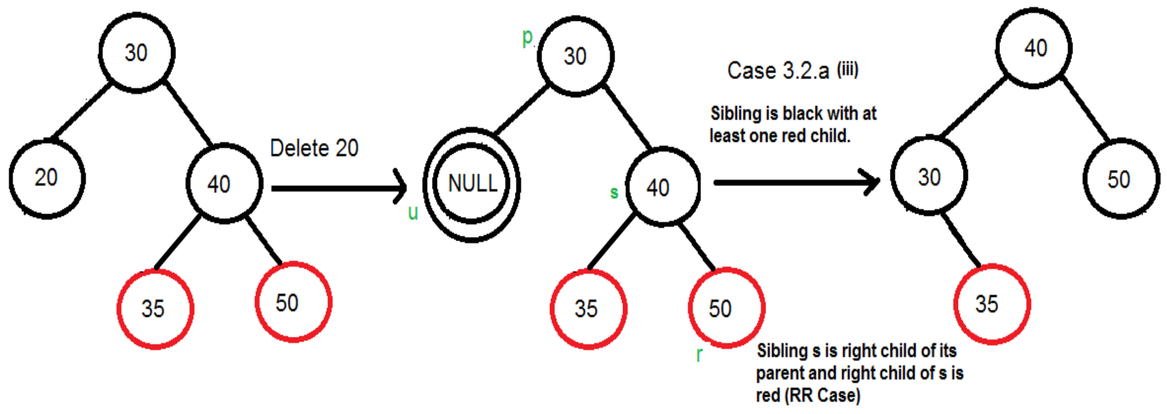
3.2. Do following while the current node u is double black and it is not root. Let sibling of node be **s**.

3.2.1. 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 subcases depending upon positions of s and r.

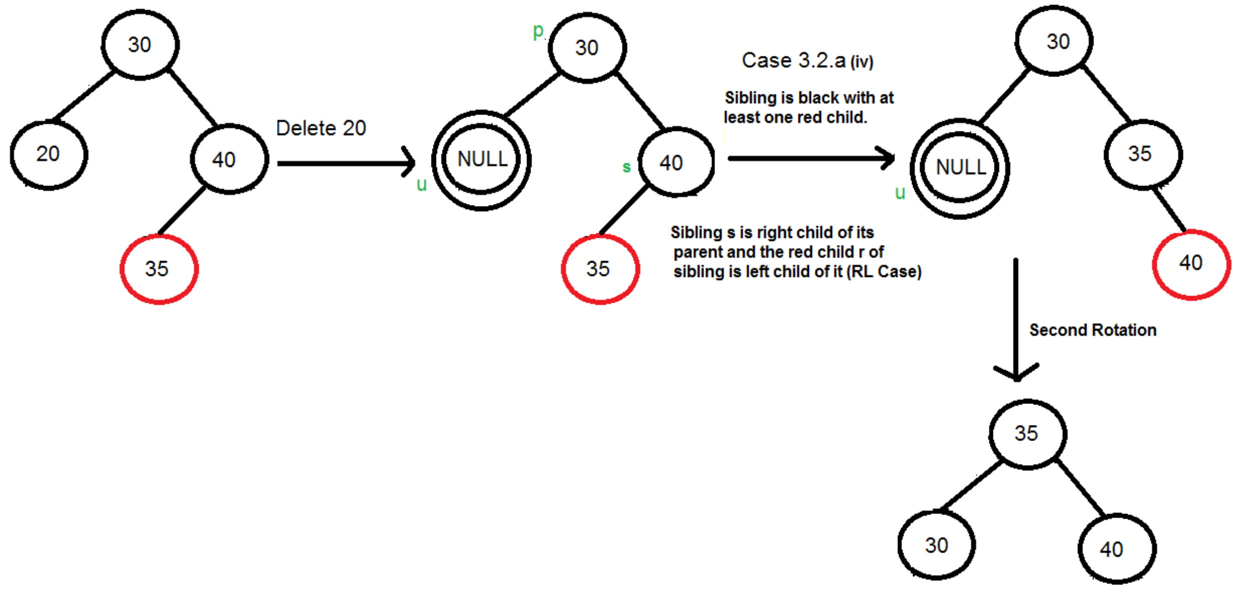
3.2.1.1. Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

3.2.1.2. 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.

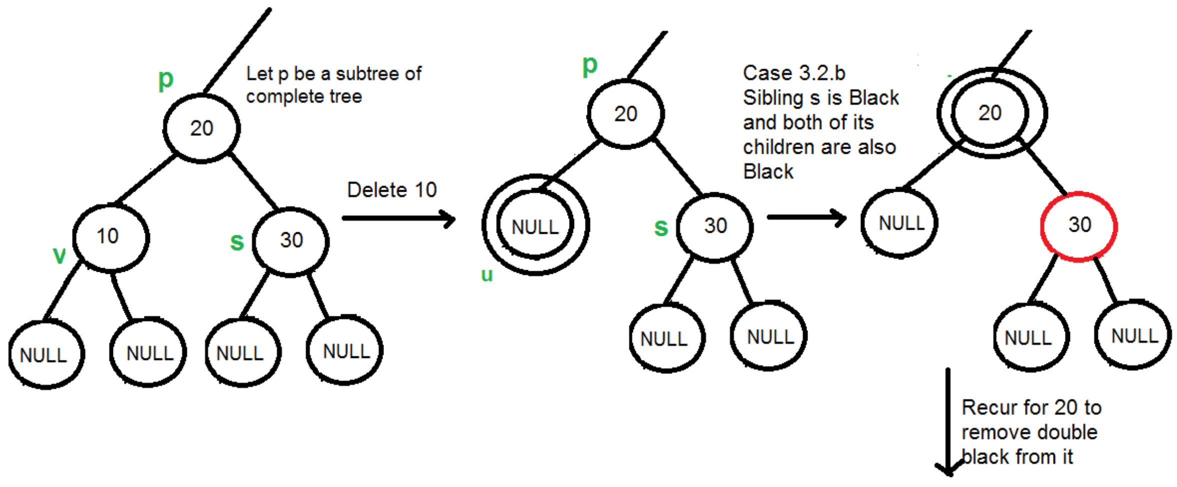
3.2.1.3. Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)


30
20
40
35
50
Delete 20
p
30
NULL
u
s
40
35
50
r
Case 3.2.a (iii)
Sibling is black with at least one red child.
40
30
50
35
Sibling s is right child of its parent and right child of s is red (RR Case)

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


30
20
40
35
Delete 20
p
30
NULL
u
s
40
35
Case 3.2.a (iv)
Sibling is black with at least one red child.
Sibling s is right child of its parent and the red child r of sibling is left child of it (RL Case)
30
NULL
u
35
40
Second Rotation
35
30
40

3.2.2. **If sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black.

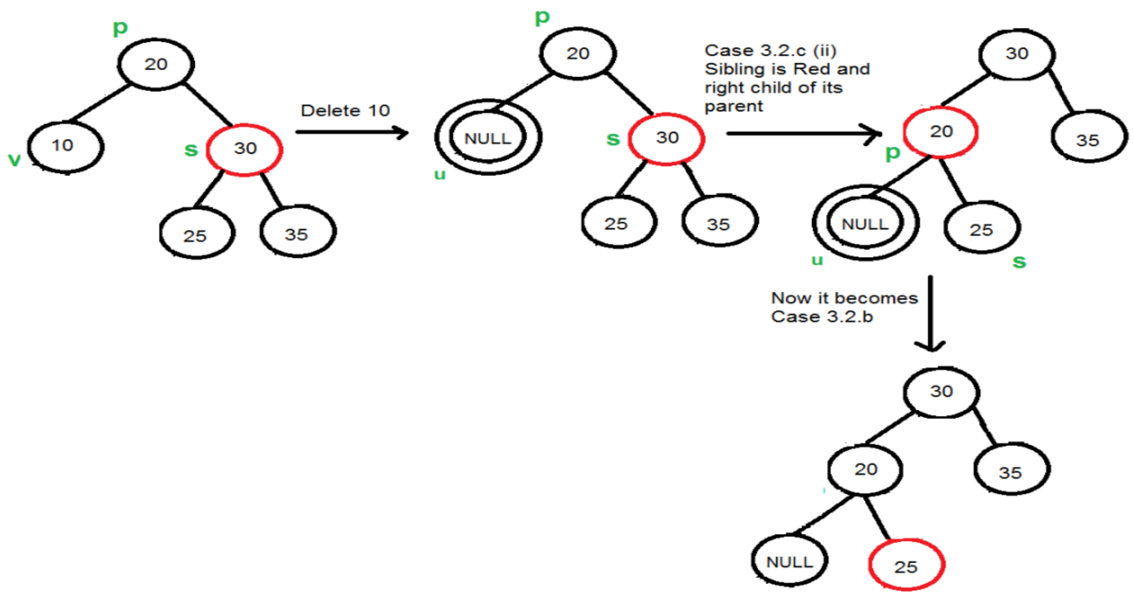

p
Let p be a subtree of complete tree
20
v
10
s
30
NULL
NULL
NULL
NULL
Delete 10
p
20
NULL
u
s
30
NULL
NULL
Case 3.2.b Sibling s is Black and both of its children are also Black
20
NULL
30
NULL
NULL
Recur for 20 to remove double black from it

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)

3.2.3. **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 subcases.

3.2.3.1. **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.

3.2.3.2. **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)

Time complexity for insertion in red black tree is O(logn) and deletion is also O(logn).