Unit- 2

Topic - Introduction to RB Tree, Height of Red Black Tree

Red Black Tree, Insertion, Deletion

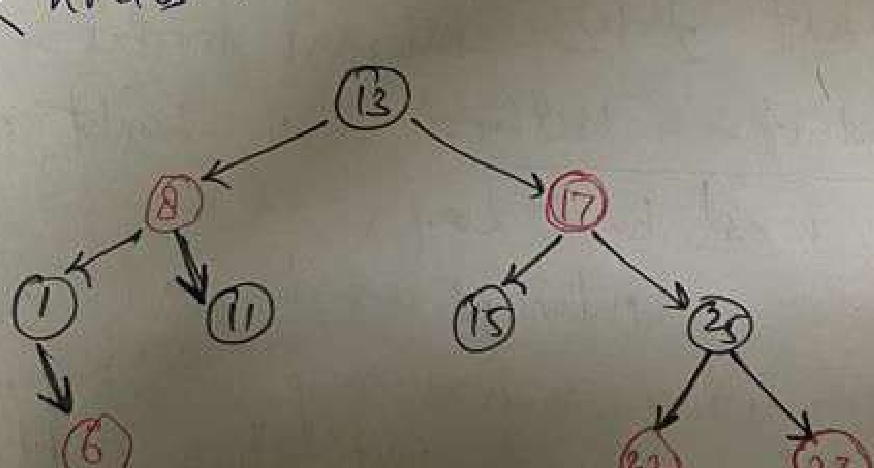
Red Black Tree RB Tree is a specialized tree noted for fast storage and retrieval of ordered information.

→ Nodes in Red Black Tree hold an extra bit called color representing "red" and "Black" which is used when re-organizing the tree to ensure that it is always balanced.

Properties of a Red Black Tree

1. Every node is either black or red
2. The root is always black
3. Every leaf is black
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendent leaves (NIL) contain same number of black nodes.

Example



RB Tree

Applications of Red Black Tree

① Efficient Searching & Sorting, RB Tree maintain a balanced structure, ensures that the height of tree remains logarithmic in relation to number of elements. It is ideal for searching & sorting applications.

② File System Implementation, RB Tree maintains directory structure efficiently, ensure fast file lookups and modifications.

③ Geospatial Applications, In GIS and mapping applications, RB Tree can help store and search for geospatial data efficiently.

Height of a Red Black Tree

→ A red black tree with n internal node has height at most $2\lg(n+1)$

→ bh(x) means black height of any node 'x'.

→ subtree rooted at any node x contains at least $2^{bh(x)}-1$ internal nodes.

→ Induction Method – if height of x is 0, then x must be a leaf,

internal nodes are $2^{bh(x)-1} = 2^0-1 = 0$

→ consider x has a positive height, and is internal node with two childrens.

Each child has a black height of either bh(x) or bh(x) − 1

→ Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has atleast $2^{bh(x)-1} - 1$ internal nodes.

→ Subtree rooted at x contains atleast

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 \quad \text{nodes}$$

$$= 2^{bh(x)} - 1$$

→ Let h be the height of the tree, According to property ④, atleast half the nodes on any simple path from root to a leaf not including the root must be black.

black height $(bh(x))$ must be atleast $h/2$, and thus

$$n \geq 2^{h/2} - 1$$

→ Moving 1, to left hand side (LHS), taking algorithm yields

$$n+1 \geq 2^{h/2}$$

$$\lg(n+1) \geq h/2 \log 2$$

$$\lg(n+1) \geq h/2$$

or, $\boxed{h \leq 2\lg(n+1)}$

→ As per this lemma each operation search, minimum, maximum, successor, predecessor runs in $O(\lg n)$ time on a red-black tree.

Rotations Search-tree operations Tree-Insert and tree-delete when run on a red-black tree with n keys take $O(\lg n)$ time. Because they modify tree, result may violate red-black properties ~~enumerated~~. To restore these properties, colors & pointers within nodes need to change.

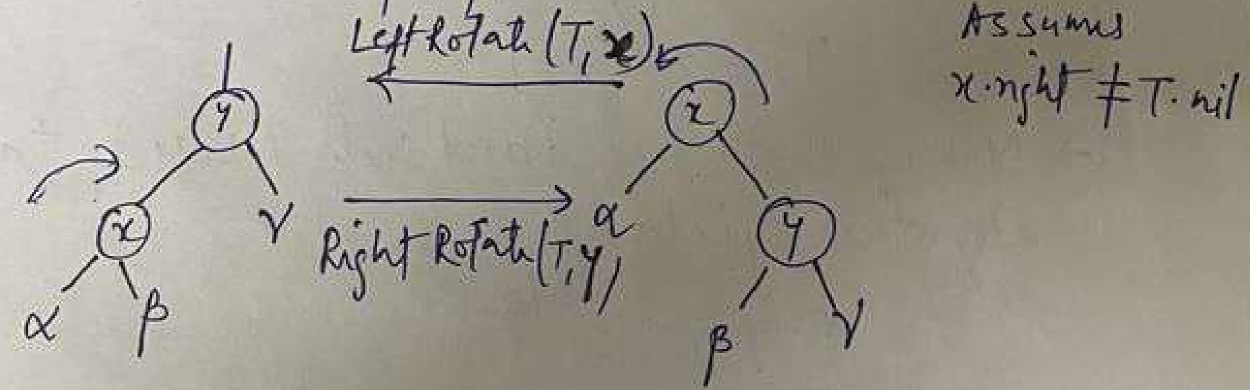
→ Pointer structure changes through rotation

→ Two types of rotations are there

- Left rotation
- Right rotation

Left Rotation

Performing left rotation on a node x, which transforms structure on the right side of figure to the structure of left.



Right Rotation

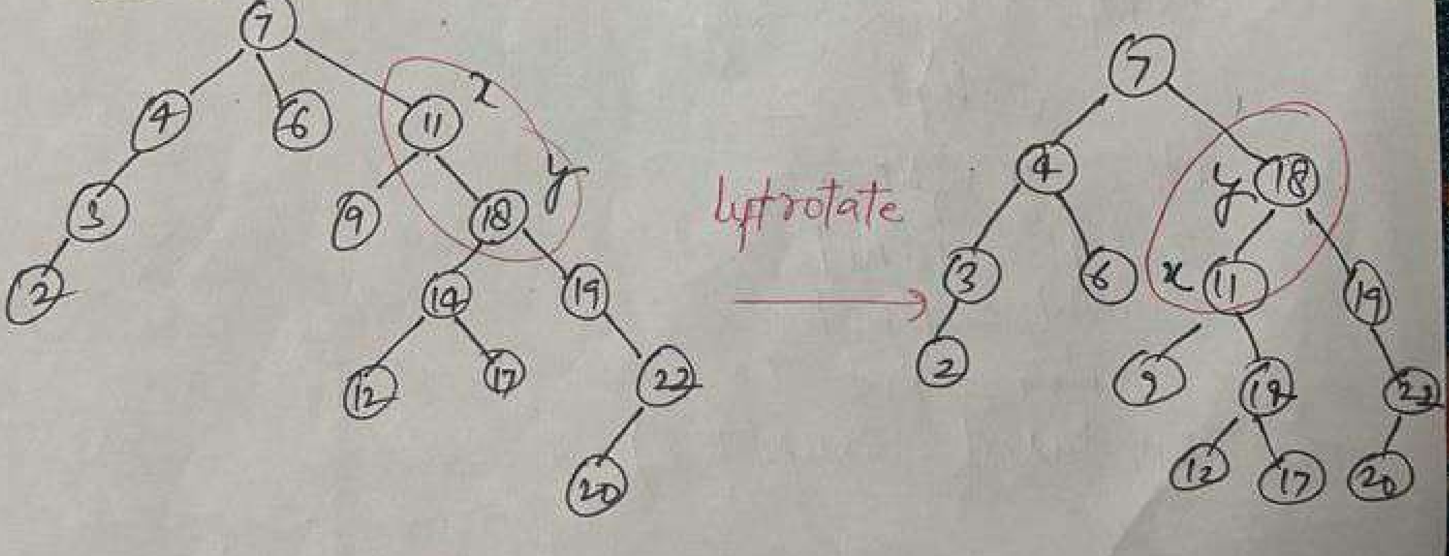
Performing right rotation on node y, Assign x as the parent y.

Algorithm LeftRotate (T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

Analysis Both Leftrotate and Right rotate run in O(1) time. Only pointers are changed by rotation and all other attributes in a node remain the same.

Example of Left Rotate



Insertion Procedure RB-Insert starts by inserting node z into the tree T as if it were an ordinary binary search tree, then it colors z red.

- To guarantee that the red-black properties are preserved an auxiliary procedure RB-Insert-Fixup recolors nodes and performs rotations.
- Call RB-Insert (T, z) inserts node z, whose key is assumed to have already been filled in, into red-black tree T.

RB-Insert (T, z)

```
x = T.root                  // node being compared with z
y = T.nil                   // y will be parent of z
while x ≠ T.nil             // descend until reaching the sentinel
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                     // found location, insert z with parent y
if y == T.nil
    T.root = z              // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil              // z's children are sentinel
z.right = T.nil
z.color = Red               // new nodes start out red
RB-Insert-Fixup (T, z)      // correct any violation of red-black properties
```

Lecture

Topic, R-B Tree Insertion, Deletion steps

Reference - Cormen

RB-Insert-Fixup(T, z)

lines 14-5 of RB-Insert set z.left and z.right to T.nil inorder to maintain proper tree structure. Third, line 16 colors z red. Fourth because coloring z red may cause a violation of one of the red-black properties, line 17 of RB-Insert calls RB-Insert-Fixup(T, z) in order to restore red-black properties.

RB-Insert-Fixup (T, z) Algorithm

z.p (parent)
z.p.p (grand parent)

```
while z.p.color == Red
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == Red
            z.p.color = Black        }
            y.color = Black          }
            z.p.p.color = Red        }  Case 1
            z = z.p.p                }
    else
        if z == z.p.right            }
            z = z.p                  }  Case 2
            Leftrotate(T, z)         }
        z.p.color = Black            }
        z.p.p.color = Red            }  Case 3
        Right-rotate(T, z.p.p)       }
```

```
16. else      // same as lines 3-15 but with right & left exchange.
17.     y = z.p.p.left
18.     if y.color == RED
19.         z.p.color = Black
20.         y.color = Black
21.         z.p.p.color = Red
22.         z = z.p.p
23.     else     // same as lines 3-15 but with right & left exchanged
24.         if z == z.p.left
25.             z = z.p
26.             Right-Rotate (T, z)
27.         z.p.color = Black
28.         z.p.p.color = Red
29.         Left rotate (T, z.p.p)
30. T.root.color = Black
```

How RB-Insert-Fixup Works?

Lines 3-15 deal with situation in which node z's parent z.p is left child of z's grandparent.

→ While loop maintains three part invariant at start of each iteration of the loop:—

a. Node z is red

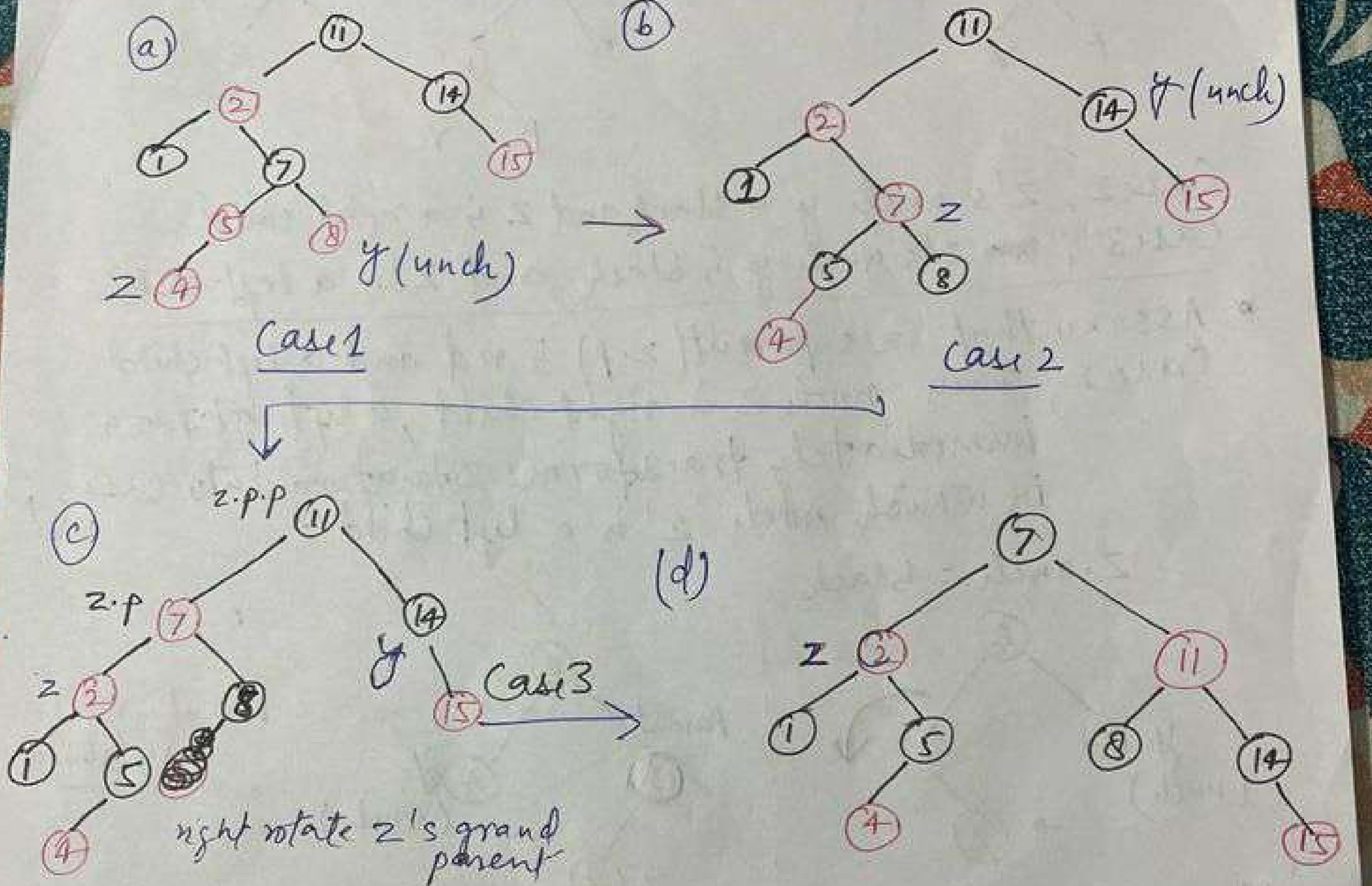
b. If z.p is root, then z.p is black

c. If tree violates any of red-black properties, it violates at most one, either property 2 or 4, but not both.

If tree violates property 2, it is because z is the root and is red.

If tree violates 4, it is because both z and z.p are red.

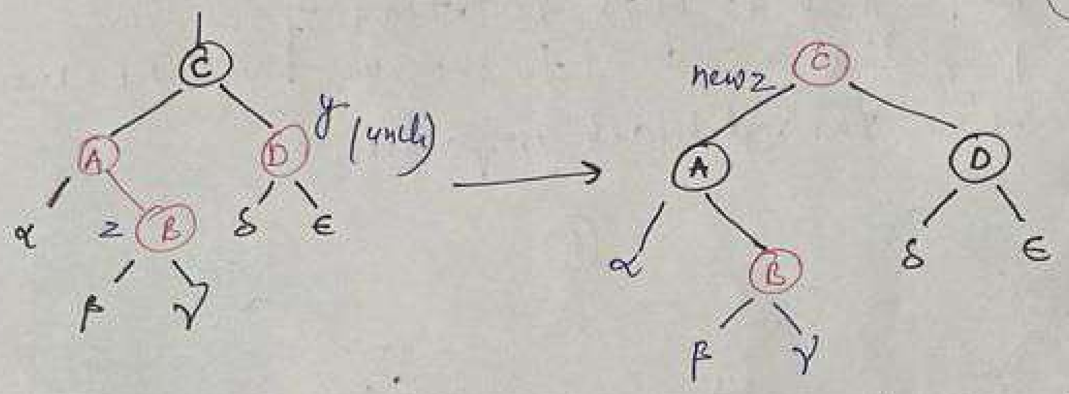
Part (c) deals with violation of red black properties is central to showing that RB-Insert-Fixup restores the red-black properties.



Case 1: z's uncle 'y' is Red

When Both z.p and y are red.
Because z's grandparent is black, its blackness can transfer down one level to both z.p & y, thereby fixing the problem of z and z.p both being red.

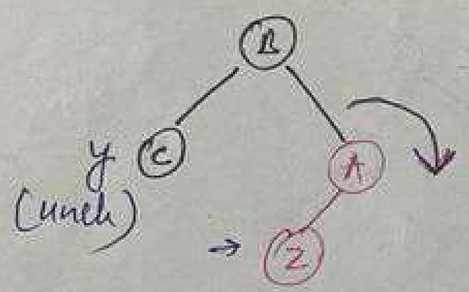
→ Thereby maintaining property (5)



Case 2: z's uncle y is black and z is a right child
Case 3: ~~..~~ z's uncle y is black and z is a left child

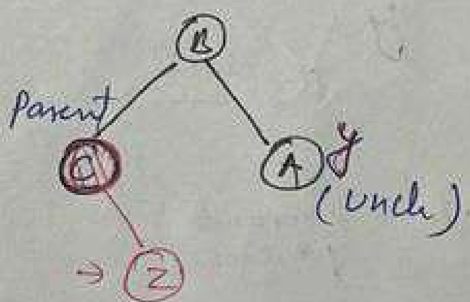
- Assume that z's parent (z.p) is red and a left child.
  Case 2, when node z is right child, a left rotation immediately transforms situation into case 3, in which node 'z' is a left child.

z.uncle = black



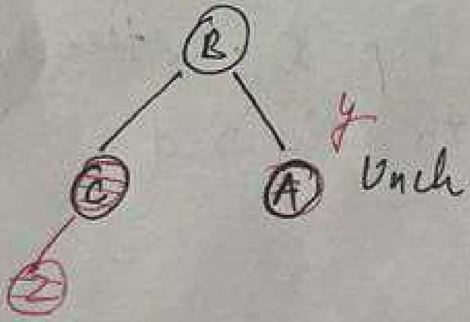
right rotate z's parent

Case ~~3~~ (RL)



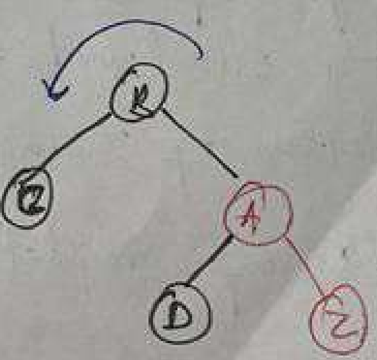
Parent is red & left child

Case 2 (LR)



Case 3 (LL)

Parent is right kind.



rotate grandparent & recolor

Case 2 (RR)

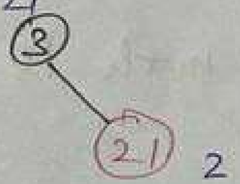
Case 0 when inserted node 'z' is root, simply color it black

Example Creating a Red-Black tree with element 3, 21, 32 and 15 in an empty tree.

Step 1

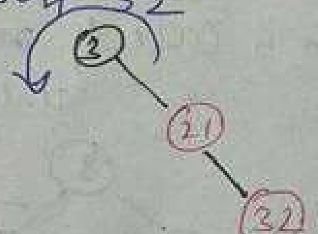
→ 3 root $\xrightarrow{\text{recolor}}$ 3

Step 2 Insert 21

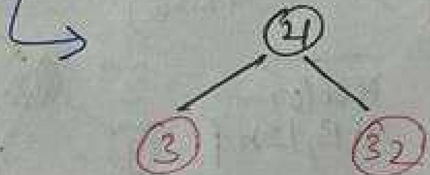


no violation

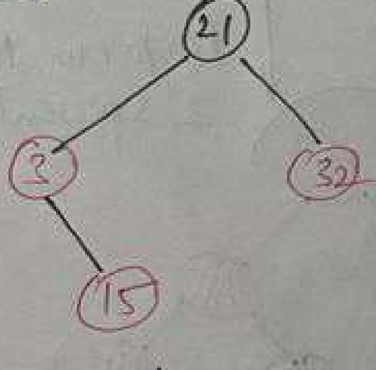
Step 3 Insert 32



violation of property (4), rotate grandparent & recolor

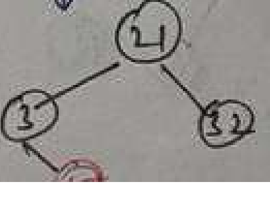


Step 4 Insert element 15

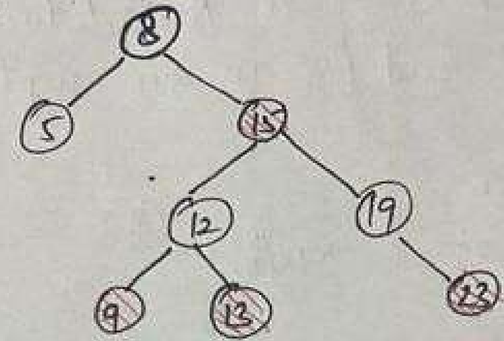


violation of property 4, & uncle is red (Case 1)
Recolor parent & uncle black & root black.

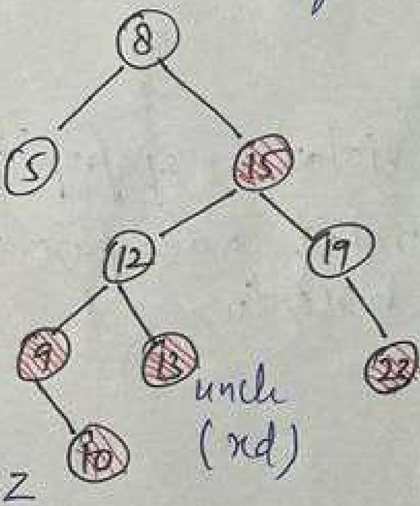
↓



Example Insert node '10' in shown Red-Black tree. Illustrate the process of insertion & readjusment.

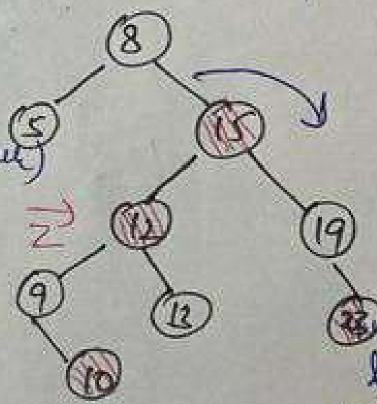


Solution Start by inserting node 10 in given RBTree, coloring it red.

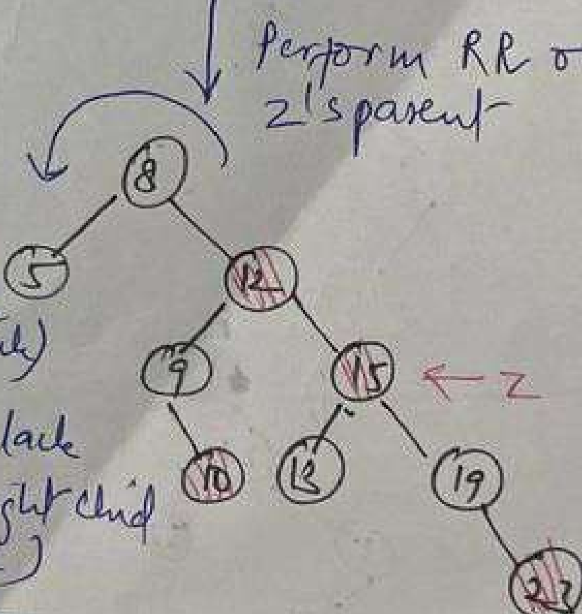


This is case 1, recolor parent, uncle, z moves up

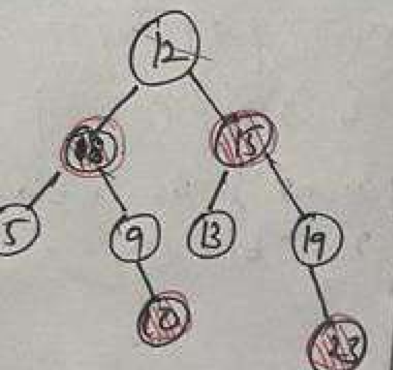
recolor 9&13, 12 red



Perform RR on z's parent

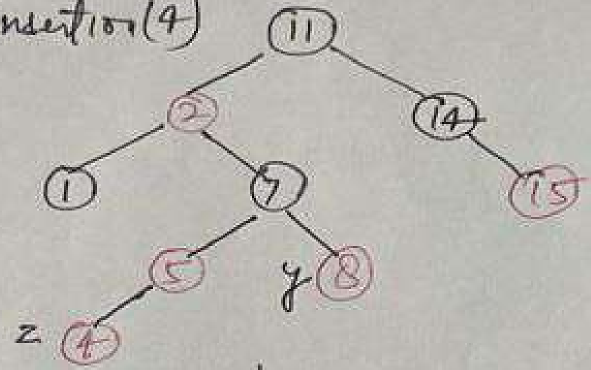


Perform rotation of Grandparent & recolor

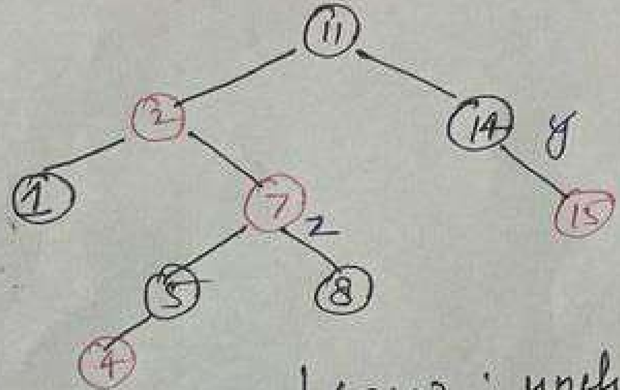


Final RBTree

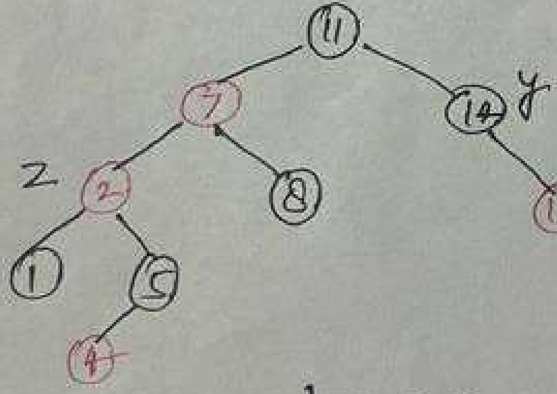
Example Insertion(4)



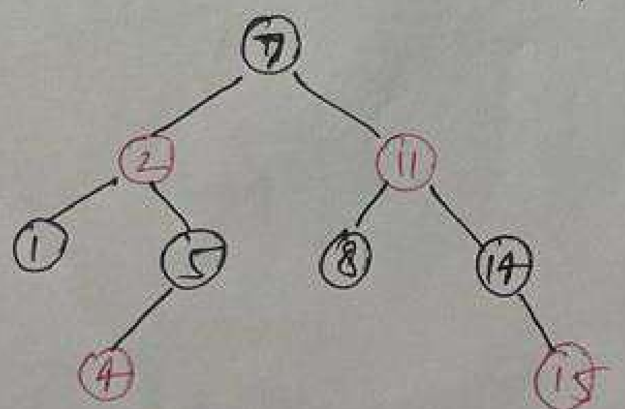
case 1 : uncle y is red.



case 2 : uncle y is black, z.p is right child



case 3 : uncle y is black, z.p is left child



Final RB Tree

Legal RB Tree

Lecture RBTree Deletion

Deletion: Deleting a node from RB Tree have three main steps/section.

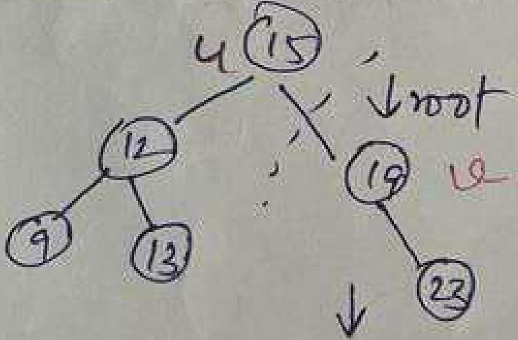
1) **Transplant** - helps us move subtrees within the tree, replaces subtree rooted at node u by subtree rooted at v.
2) **Delete** - deletes the nodes
3) **delete Fixup** - fixes any red-black violations.

Transplant (T, u, v) Algorithm

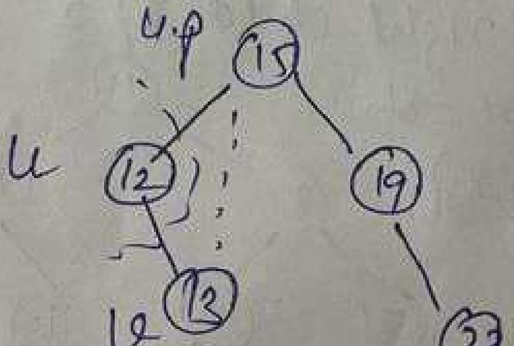
```
if u.p == T.nil           // u is root
    T.root = v
elseif u == u.p.left      // u is left child
    u.p.left = v          // replace u with v
else u.p.right = v        // u is right child
v.p = u.p
```

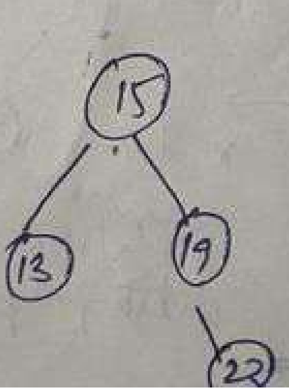
Example Show delete(15)

Case 1 u is root,

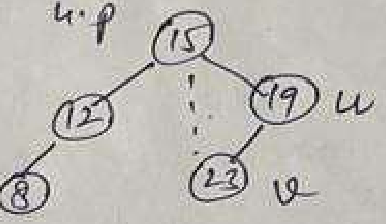


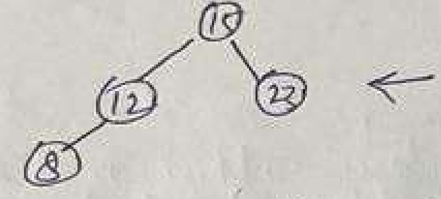
Case 2 u is left child
delete(12)





Case3: u is right child
Let us delete (19)



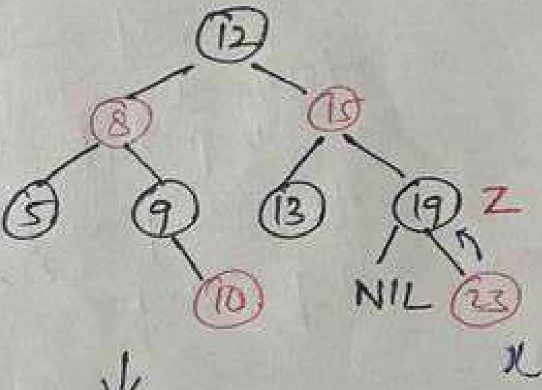


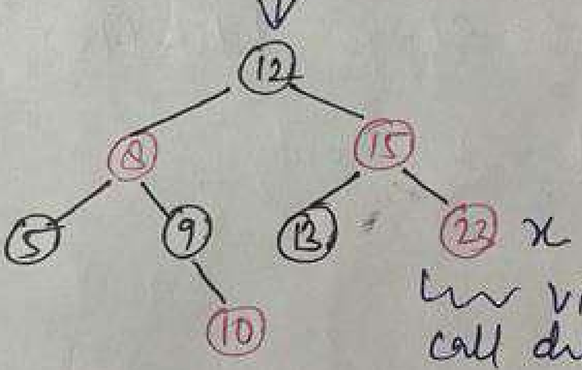
Delete function

1. Left child is NIL
2. right child is NIL
3. neither child is NIL

Case 1 delete(19)
left child is NIL

→ original color of z = black
→ transplant (T, z, x)
→ deletefixup (x)
→ set node 'x' to black

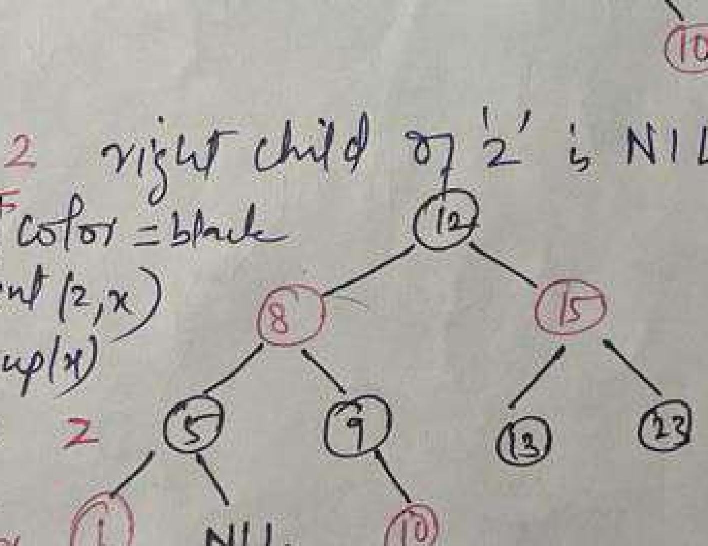


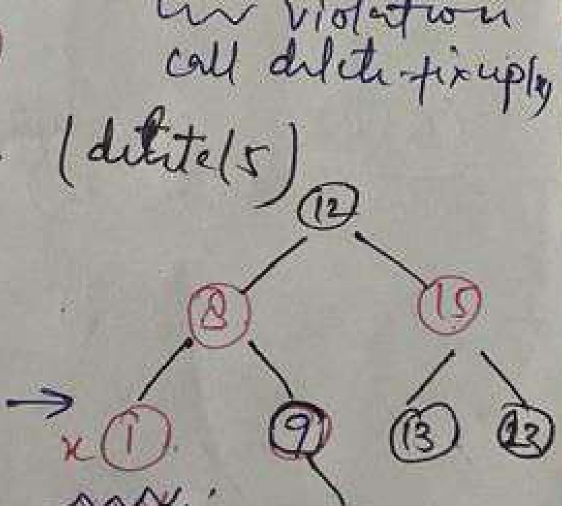


no violation
call delete-fixup(x)

Case 2 right child of 'z' is NIL (delete(5))

- original color = black
- transplant (z, x)
- deletefixup(x)
- recolor x to black

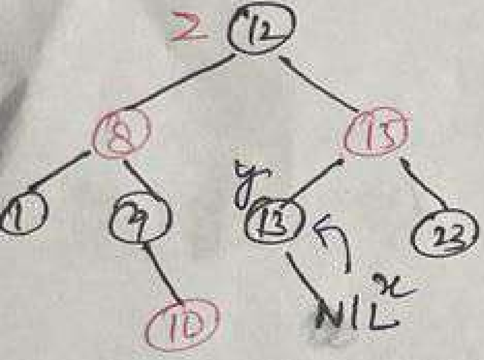




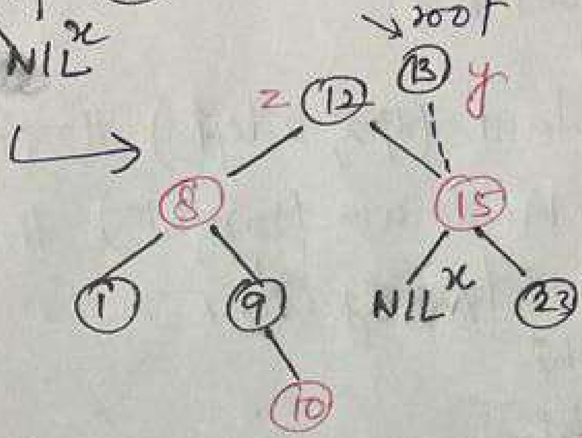
violation
delete-fixup(x)

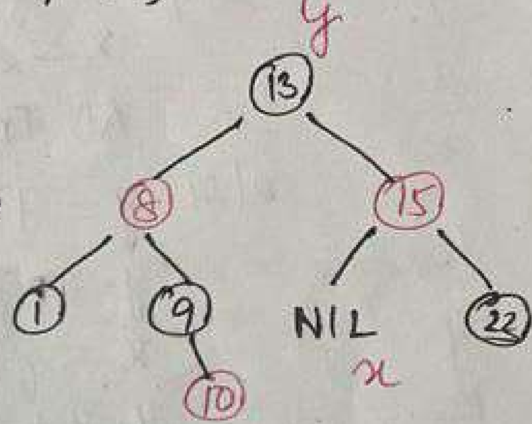
Case 3 Neither child NIL

delete(12)

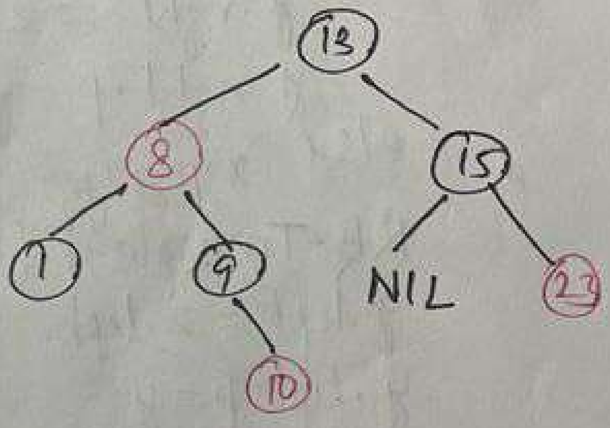


1) find minimum, 13 call it y
(successor)
2) transplant (y, x)
3) transplant (z, y)
4) call deletefixup (x)





deletefixup(x)

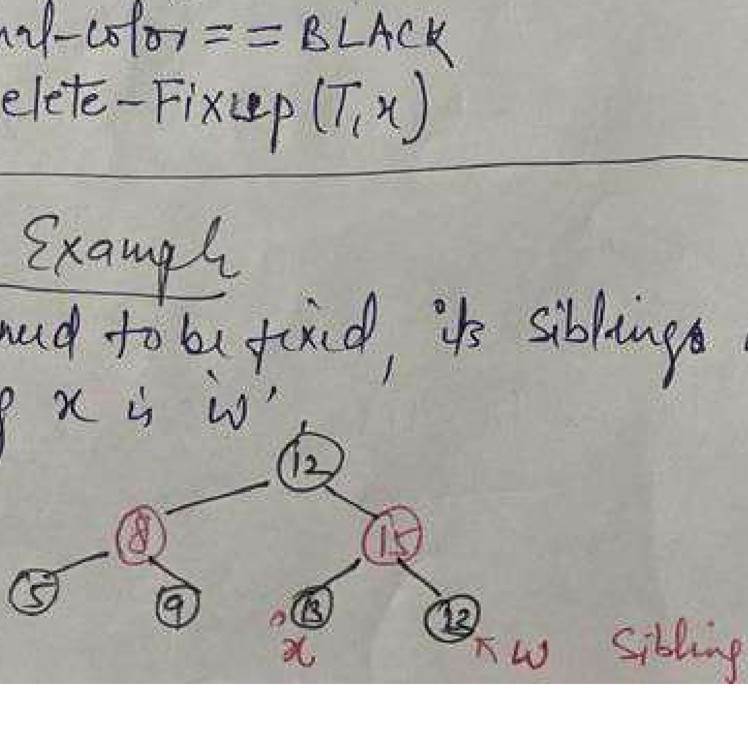


Algorithm RB-Delete(T, z)

```
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-Transplant(T, z, z.right)     // replace z by its right child
elseif z.right == T.nil
    x = z.left
    RB-Transplant(T, z, z.left)      // replace z by its left child
else y = Tree-Minimum(z.right)       // y is z's successor
    y-original-color = y.color
    x = y.right
    if y ≠ z.right
        RB-Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
    else x.p = y
    RB-Transplant(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-Delete-Fixup(T, x)
```

RB Tree Fixes Example

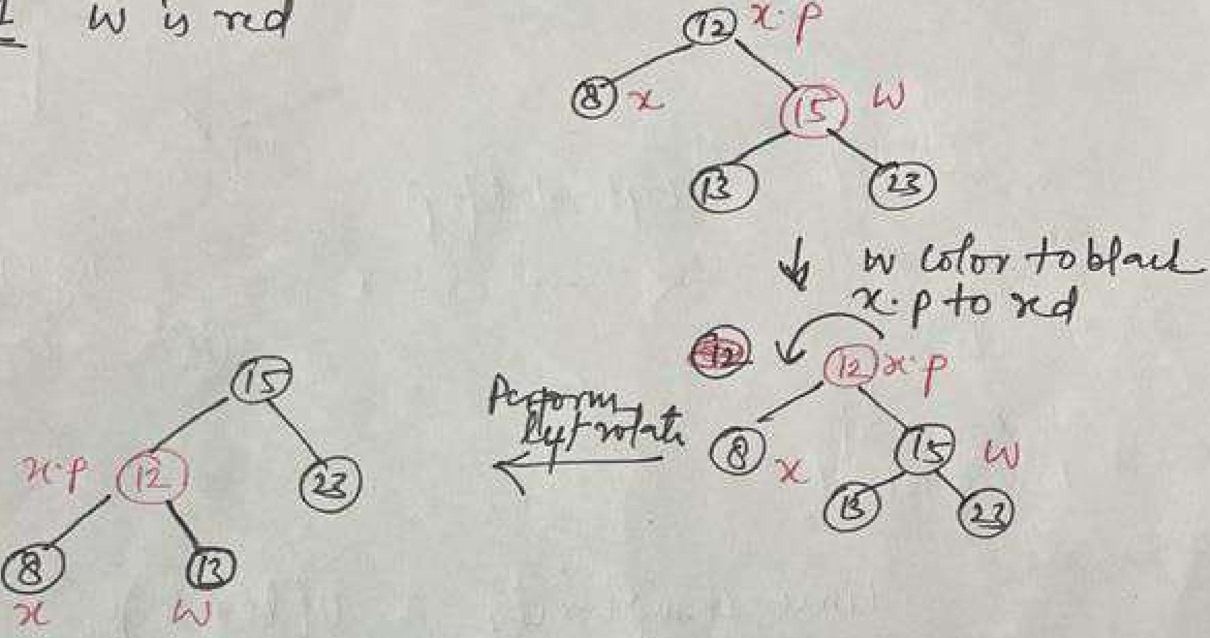
If a node 'x' need to be fixed, its siblings is considered. Consider sibling x is 'w'



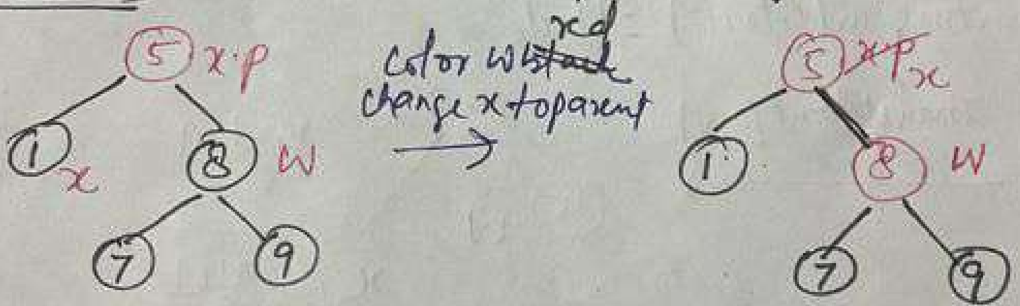
Four types of fixes are:—

1. w is red
2. w is black, and w.left & w.right are black
3. w is black and w.left is red and w.right is black
4. w is black and w.right is red

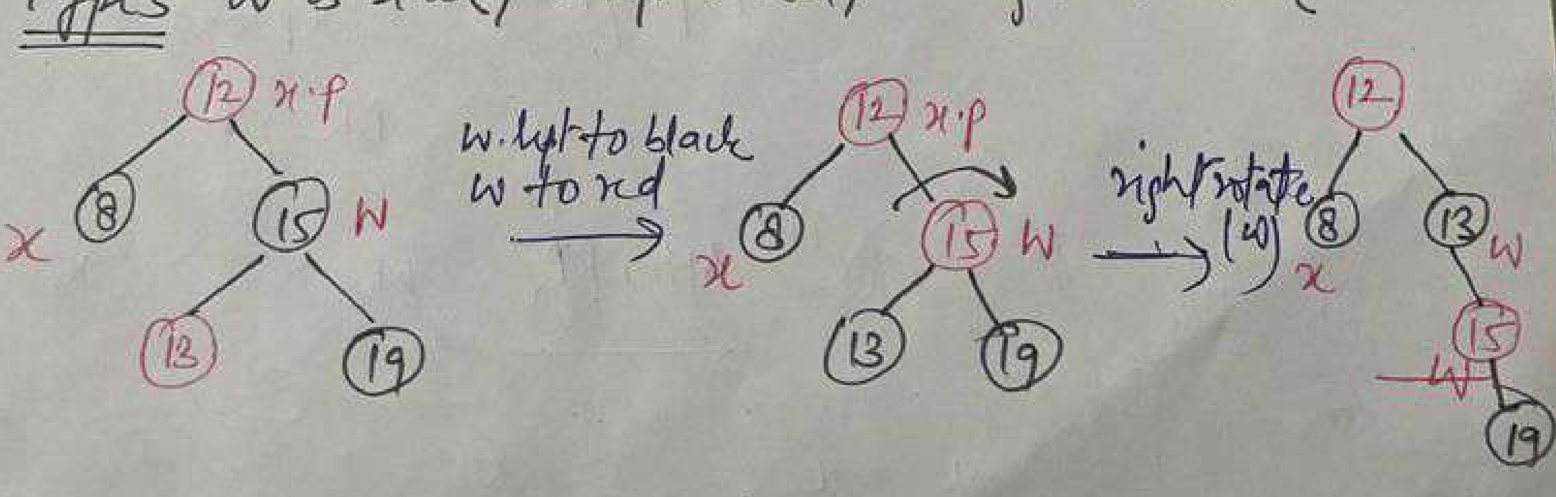
Type 1 w is red



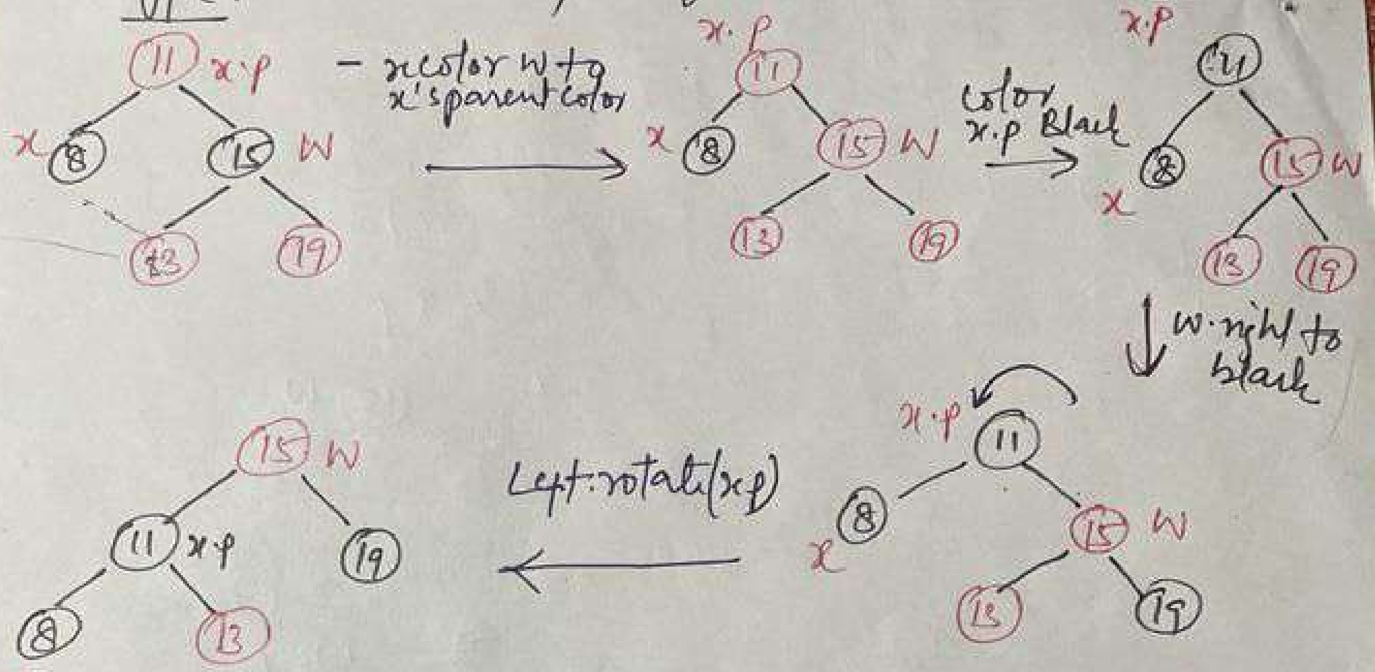
Type 2 when w, w.left, w.right is black



Type 3 w is black, w.left is Red, w.right is black



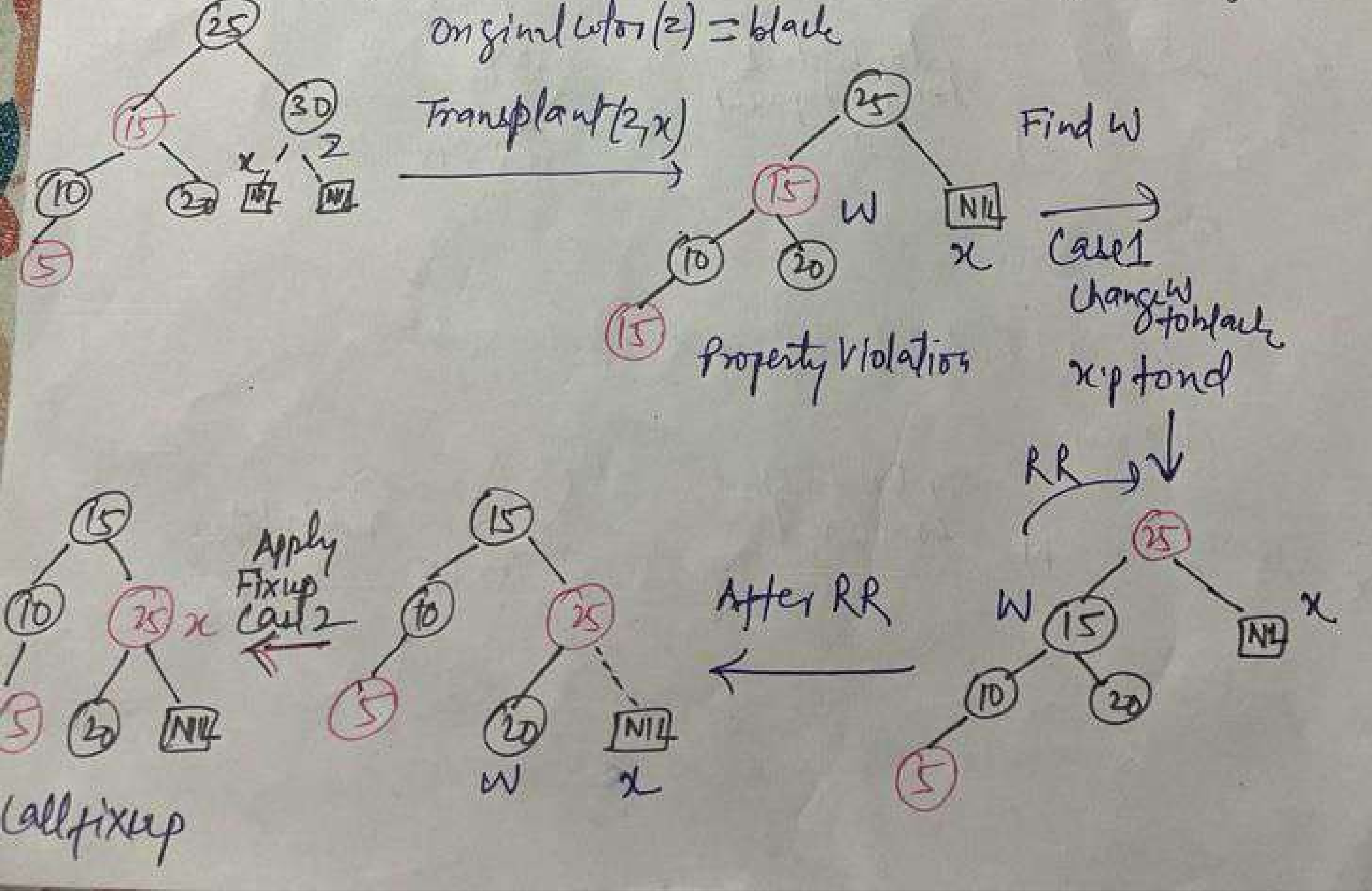
Type 4 W is black, W.right is Red



set x = root
set x = Black

Analysis Time complexity is $O(\log n)$

Example Perform delete(30) operation on following RB Tree.



Algorithm RB-DELETE-FIXUP(T, x)

```
while x ≠ T.root and x.color == Black
    if x == x.p.left
        w = x.p.right
        if w.color == red
            w.color = Black
            x.p.color = Red                        Case 1
            Left rotate (T, x.p)
            w = x.p.right
        if w.left.color == Black & w.right.color == Black
            w.color = Red
            x = x.p                                Case 2
        else
            if w.right.color == Black
                w.left.color = Black
                w.color = Red
                Right-Rotate (T, w)                Case 3
                w = x.p.right
            w.color = x.p.color
            x.p.color = Black
            w.right.color = Black                  Case 4
            Left-rotate (T, x.p)
            x = T.root
```

Remaining Part with right & left exchanged.