```
Tree_Insert (T, p)
{
    q = T→root;
    prev = NULL; // parent of q.
    while(q != NULL)
    {   prev = q;
        if ( p→key < q→key)
            q = q→left;
        else
            q = q→right;
    }
    p→parent = prev;
    if prev == NULL
        T→root = p;
    else if p→key < prev→key
        prev→left = p;
    else
        prev→right = p;
```

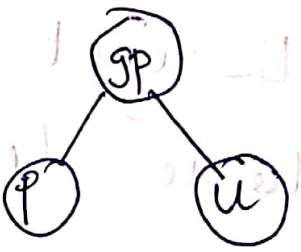
2

```
RB_Tree_fixup (T, p); // new node added
                      recoloring or restructuring of T
}
```

RB_Tree_fixup (T, x) ← node to be added / node to be adjusted ②

```
{
While (x ≠ Root and x→parent→color
                        == Red)
{
    p = x→parent;
    gp = p→parent;
    if gp→left == p then
        RB_left_correct (T, x, p, gp);
    else
        RB_Right_correct (T, x, p, gp);
}
x→color = black;
}
```

RB_left_correct (T, x, p, gp)



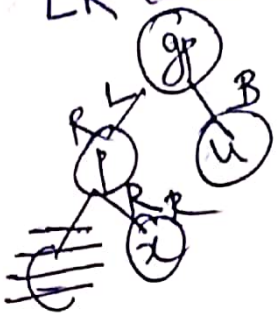

```
{
    u = gp→right;
    if u→color == red then
    {
        p→color = black;
        u→color = black;
        gp→color = red;
        x = gp;
    }
else // if u→color == black
```

case 1
uncle is red

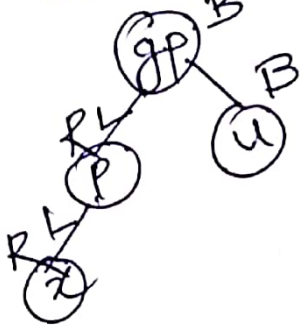
(3)

```
{   if (p->right == x) // x is Rightson of p.
    {
        rotateleft(p);
        x = x->left
    }
    else
    {
        rotateright(gp);
        gp->color = red;
        p->color = black;
    }
}
```

LR case



LL case



RB_right_correct is symmetric case where left ⇔ right.

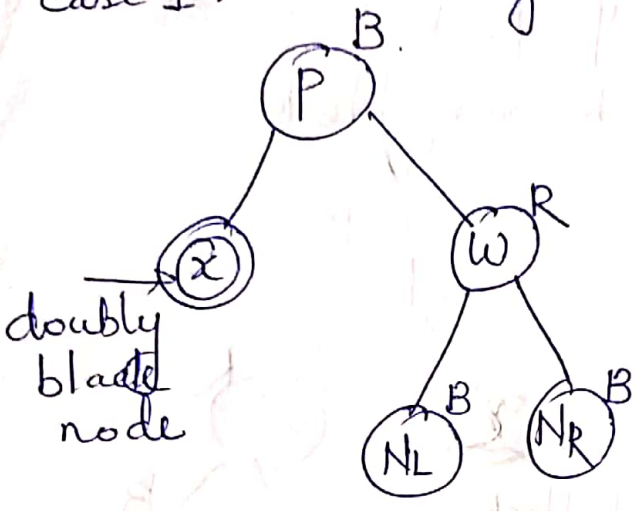
Let z → search a node containing desired data.

```
if (z->left == null or z->right == null)        // case 1 or case 2 of BST delete
    y = z;
else
{
    y = successor(z);
    z->data = y->data;
}
if (y->left == null) // In case 1 x is NULL
    x = y->right;
else
    x = y->left;
if (x != null) // if not case 1
    x->parent = y->parent;
if (y->parent == null)
    root = x;
else y->parent->left == y
    y->parent->left = x;
else
    y->parent->right = x;
```

```
if (y->color == black)
    RB_Delete_fixup(T, x);
free(y);
```

x is nonroot doubly black node
if it is a left child → 4 cases
if it is a right child → 4 cases (symmetric)

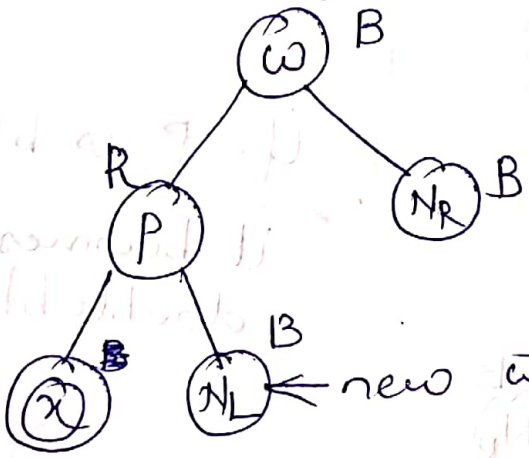
case 1: x sibling is R.



Soln

1) swap colors of p & w
2) perform left rotation at parent

So resulting tree

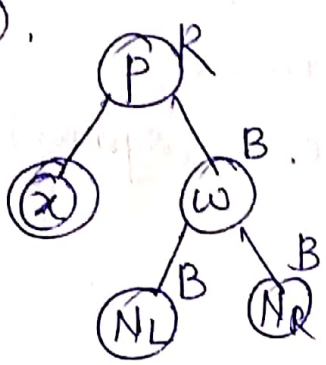


case 1 → doesn't give answer
its gets converted to
case 2, case 3, case 4
where sibling w is of black color.

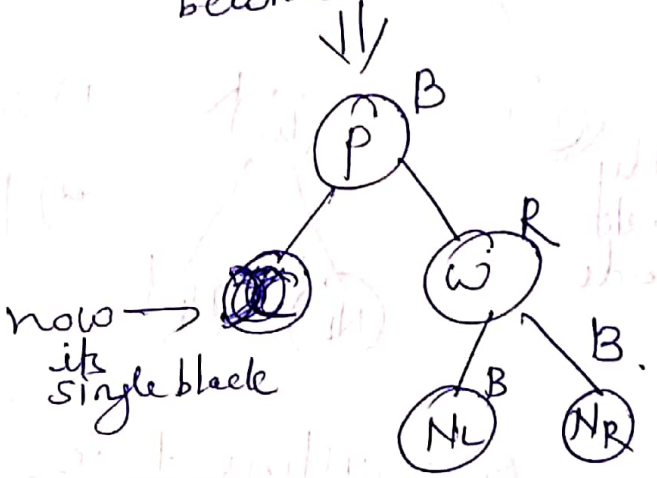
These cases are distinguished using by color's of w's children.

Case 2 : w black and N_L & N_R black

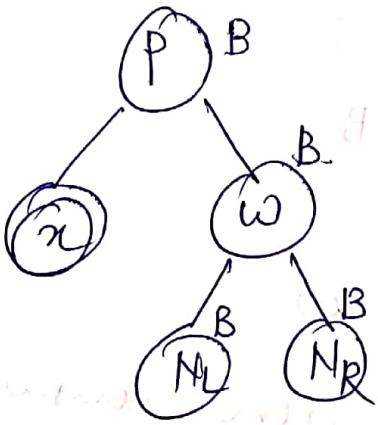
(a)



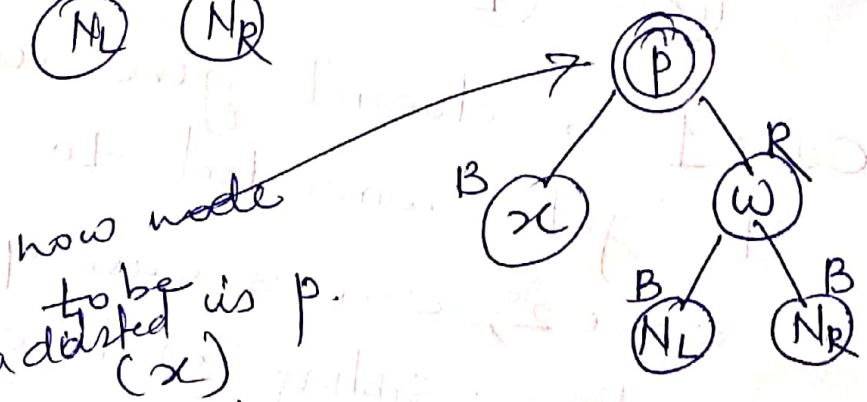
Give blackness of x & w to p. if p is Red, its becomes black.



(b)



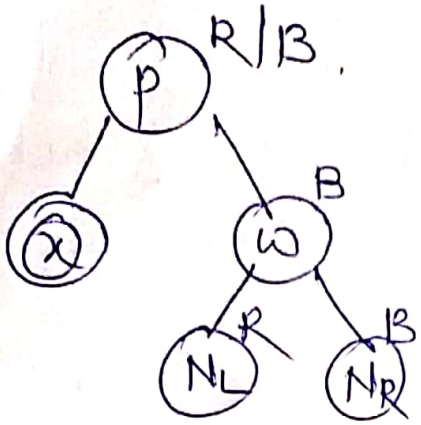
if p is black, it becomes double black.



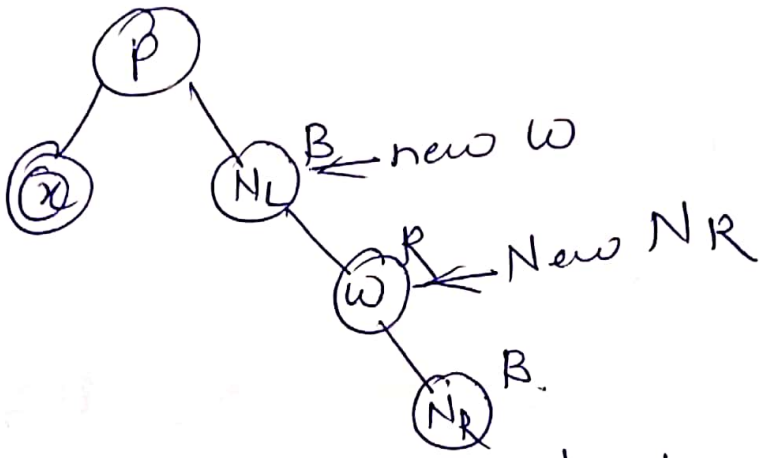
now node to be addressed is p. (x)

Violation of property moves one level up.

Case 3: two black, N_L is R & N_R B.

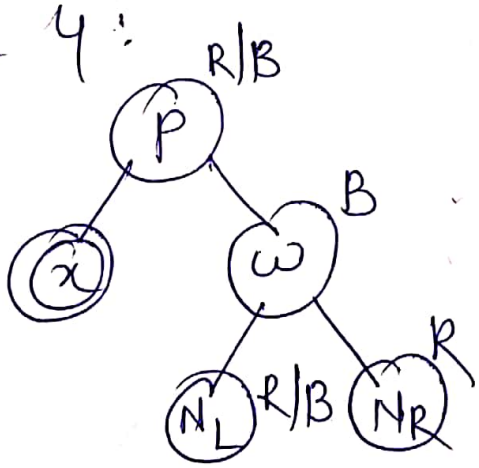


1) swap colors of N_L & w.
2) Right rotate at w

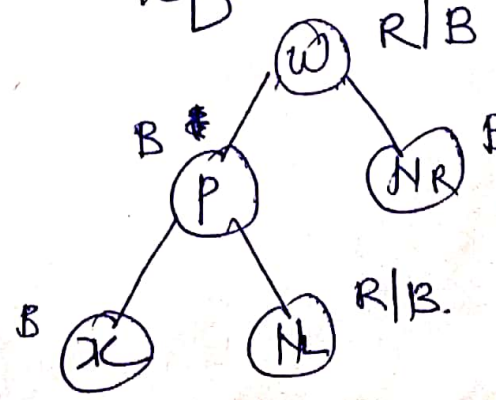


This is transformed to case 4.
where w is black, N_R is Red,
$N_L \rightarrow$ R/B.

- Case 4:

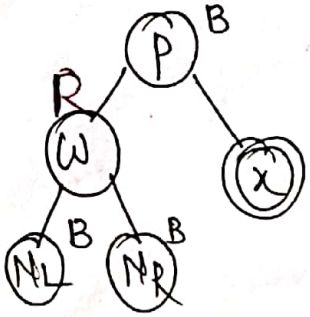


$N_R \rightarrow$ color = $w \rightarrow$ color
$w \rightarrow$ color = $p \rightarrow$ color
$p \rightarrow$ color = black
leftrotate at p

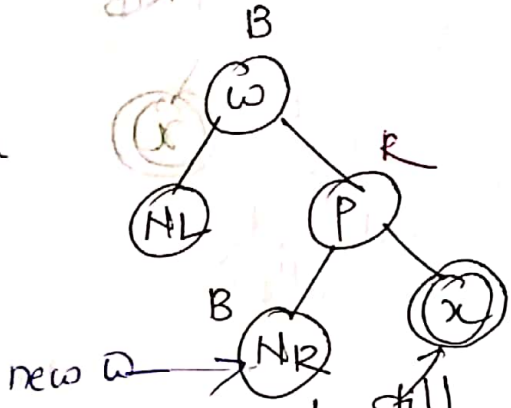


x is nonroot doubly black node & it is a right child.

case 1': x sibling w is R

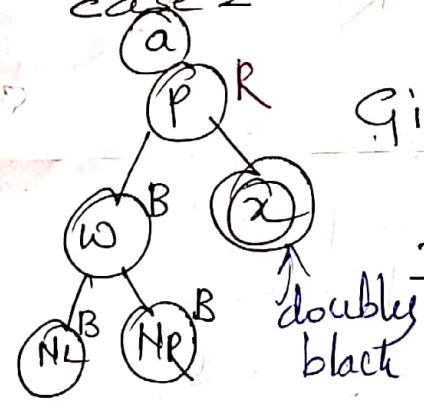


1) swap colors of p & w
2) Perform right rotation at parent.



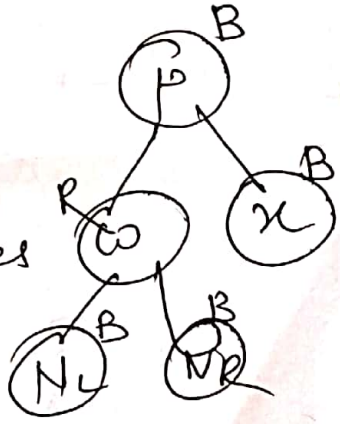
case 1' doesn't give answer, its get converted to case 2', case 3' or case 4' where sibling w is of black color.
These cases are distinguished using the colors of w's children.

case 2': w black, N_L & N_R black

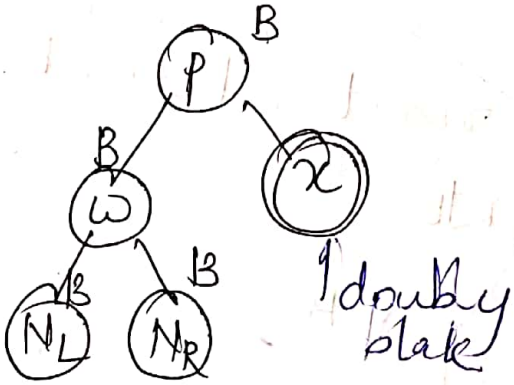


Give blackness of x & w to p.

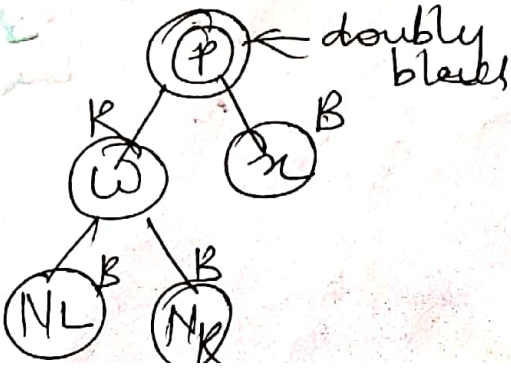
If p is Red, it becomes ~~Black~~, stop



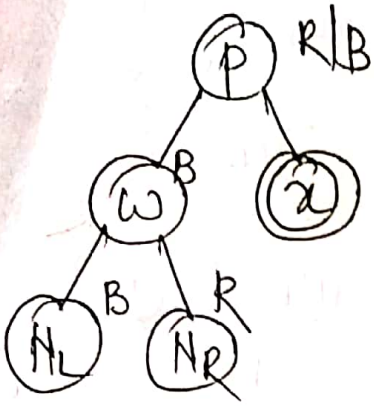
(b)



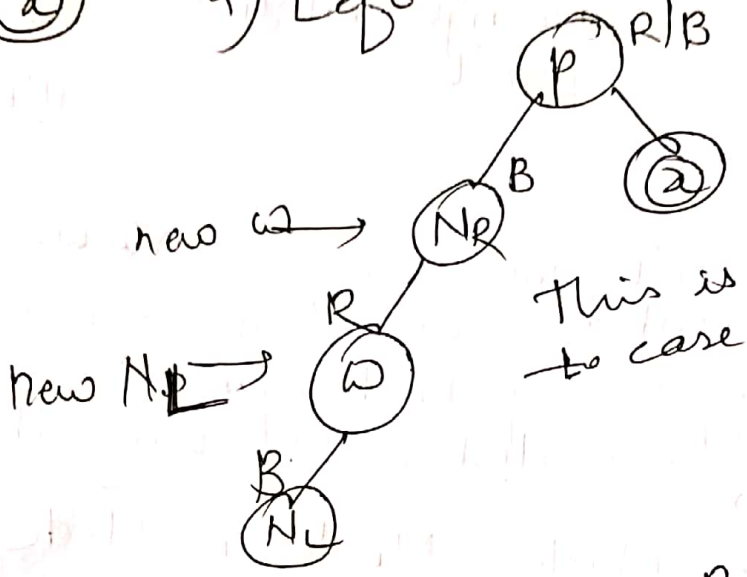
If p is black
p becomes double black
Violation of property moves one level up



Case 3': w is black, N_L is B & N_R is R

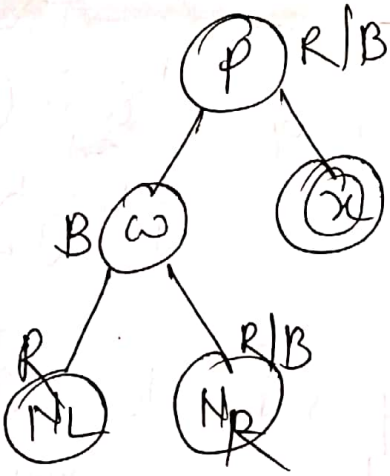


1) Swap colors of N_R & w.
2) Left rotate at w.



This is transformed to case 4.

case 4' → w→B, N_L→R, N_R→R/B.



N_L→color = w→color
w→color = p→color
p→color = black
x becomes singly black
right rotate at p