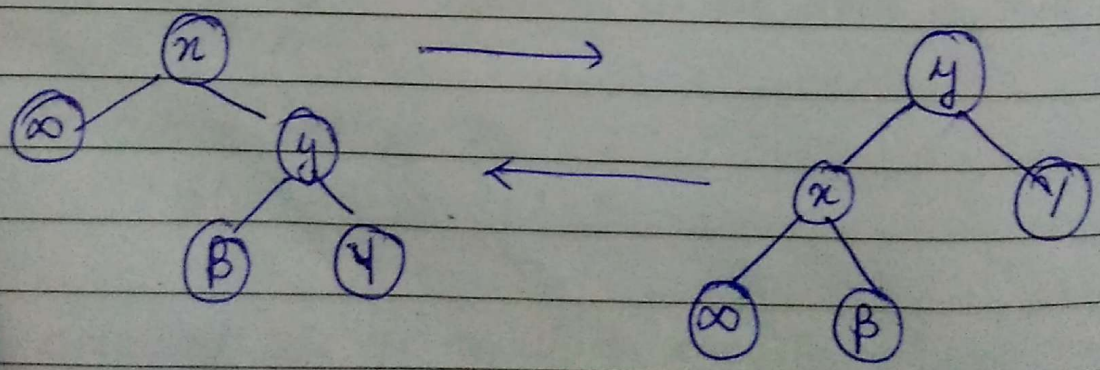
RED BLACK TREE

- A Red Black tree is a binary tree with one extra node i.e its color which can be either red or black.
Each node of the tree contains following field ~~of~~ color, key, left, right and parent. A BST is a red black tree if satisfy the following red black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black
4. If a node is red then both of its children should be black.
5. For each node all paths from the node to the descendent leaves contains the same no. of the black nodes.

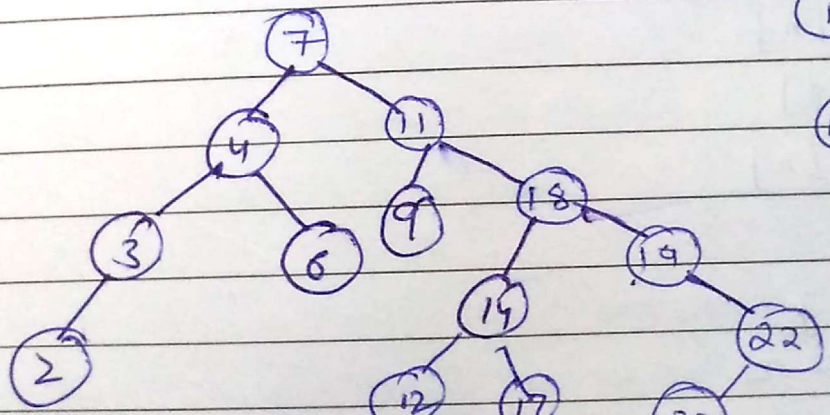
The operations like insert, delete when run on red black tree with n keys take order of log n time and may violate red black property.
To restore these properties we change color of some node or change the pointer structure.
Pointer structure is changed using rotation operation which helps in preserving BST property.
There are 2 kinds of rotation - left and right rotation.

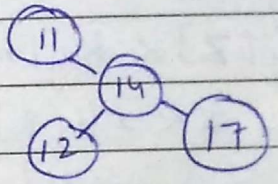
when we do a left rotation on node x be assume its right child is not nill node.

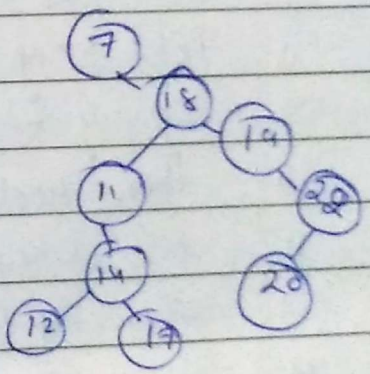


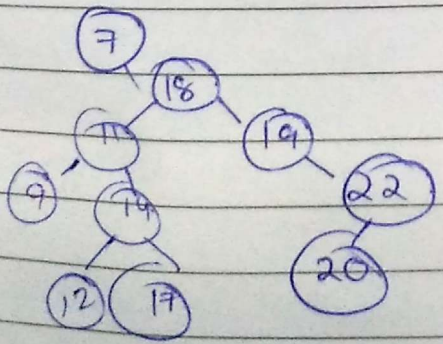
Left rotate (T, x)

1. y ← Right[x]
2. right[x] ← left[y]
3. P[left[y] ← x
4. P[y] ← P[x]
5. if P[x] = NIL[T]
6. then root[T] ← y
7. else if x = left[P[x]]
8. then left[P[x]] ← y
9. else right[P[x]] ← y
10. left[y] ← x
11. P[x] ← y









Insertion into a red black tree takes $O(\log n)$ time we insert node z into the tree T and then color it red. To preserve the properties of red black tree we use RB-INSERTFIXUP() procedure to re colour the nodes and perform rotations.
RBINSERT() procedure insert node z whose key field is assumed to have already filled in.

```
RB_Insert (T, z)
1   y ← NIL[T]
2   x ← root[T]
3   while x ≠ NIL[T]
4   do y ← x
5   if key[z] < key[x]
6   then  x ← left[x]
7   else  x ← right[x]
8   P[z] ← y
9   if  y = NIL[T]
10  then root[T] ← z
11  else if key[z] < key[y]
12  then left[y] ← z
13  else right[y] ← z
14  left[z] ← NIL[T]
15  right[z] ← NIL[T]
16  color[z] ← Red
17  RB_INSERT_FIXUP(T, z)
```

Page No.
Date : / /

(T, 4)

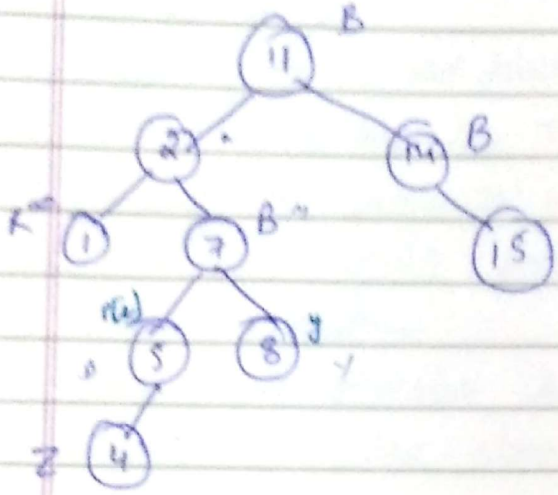
```
1  while color[P[z]] = Red
2  do if P[z] = left[P[P[z]]]
3     then y ← right[P[P[z]]]
4     if color[y] = RED
5     then color[P[z]] ← BLACK
6     color[y] ← BLACK.
7     color[P[P[z]]] ← RED
8        z ← P[P[z]]]
9     else if z = right[P[z]]
10       then z ← P[z]
11       left-Rotate(T, z)
12    else color[P[z]] ← BLACK.
13.   color[P[P[z]]] ← Red
14.      Right-Rotate(T, P[P[z]])
15.      else (Same as then clause)
16.   color[root[T]] ← BLACK
```

Which of the red black tree properties can be violated on the call to RB-INSERT-FIXUP?

Property 1 and 3 holds, both children of newly inserted red node are sentinal nil. property 5, which says that no. of black nodes should be same in every path from a given node also holds because node z replaces the black sentinal node and it is red in color with its children BLACK.

Property 2 is violated, if z is the root.

Property 4 is violated, if z's parent is red.



In this figure node z, is inserted which is red in color and its parent is also red. Thus it violates property 4 RB-INSERT-FIXUP() handles 3 invariance at the start of each iteration.

1. Node z is red
2. If P[z] is the root then ~~parent~~ P[z] is made black
3. If RB properties are violated then there is atmost one violation and it is either property 2 or 4.

If property 2 is violated, it occurs because z is root and is red in color. If property 4 is violated, it occurs because both ~~of~~ z and P[z] are red in color.
There are 2 possible outcomes of each iteration of the loop

1. pointer z moves up in the tree.
2. Some rotations are performed and the loop terminates.

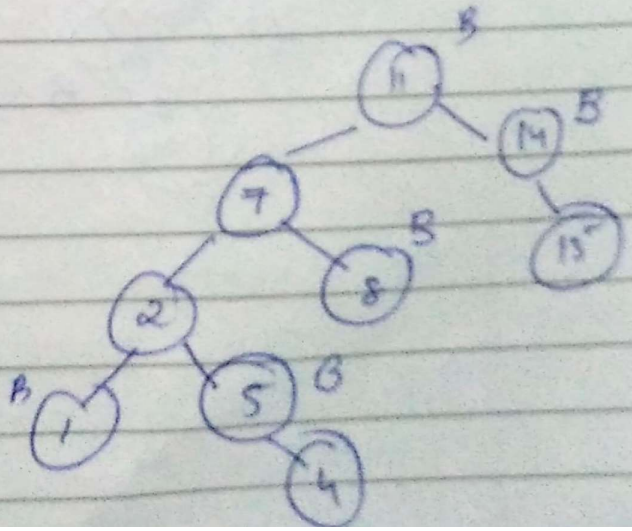
```
RB-Insert-Fixup(T,4)
while color[P[z]] = Red
    color[P[4]] // color of 5 is red.

do if P[4] = left[P[P[4]]] // 5 is left child of 7
   then y ← right[P[P[z]]]        y ← 8
```

3. if color [4] = RED
   color [8] = RED.
4. then color [P[z]] ← Black
   color [5] ← black.
5. color [4] ← black
   color [8] ← black.
   color [P[P[z]]] ← Red
   z ← 7

* line no 9 to 15 will not execute.

1. while color P[z] = Red.
   color [P[7]] // color of 2 is red.
2. do if P[7] = left [P[P[7]]] //
   2 = 11 | true
   4 ← 14
3. if color [14] = Red // false
9. z = right [P[z]]
   z = right [P[7]]
   2
   z = 7 // true
10. z ← P[z]
    z ← 2

left Rotate (T, 2)



while color [P[z] = Red

7 is red.

if P[z] = left [P[P[z]

7 = 7

11 true

y ← 14

9. z = right[P[z]]

= 2

2 = 7 // ~~false~~ true

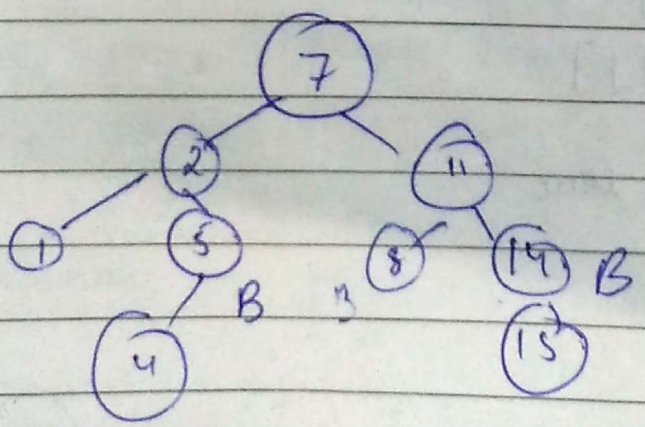
10.

---

color P[z] = Black

color P[P[z]] = Red

→ right rotate (T, P 11)



Page No.
Date : / /

deletion operation take $O(\log n)$ time. Two procedures are used RB_delete(T, z) to delete a node z.
RB-Delete-Fixup (T, z) to change the colors and perform rotations to restore RB properties.

```
RB-Delete (T, z)
1.  if left[z] = NIL[T] OR right[z]
2.  then y ← z
3.  else y ← Tree successor (z)
4.  if left[y] ≠ NIL[T]
5.  then x ← left[y]
6.  else x ← right[y]
7.  P[x] ← P[y]
8.  if P[y] = NIL[T]
9.  then root[T] ← x
10. else if y = left[P[y]]
11.    then left[P[y]] ← x
12. else right[P[y] ← x
13. if y ≠ z
14. then key[z] ← key[y]
15. copy y's satellite data into z
16. if color[y] = black
17. then RB-delete-fixup(T, x)
18. return y.
```

```
RB_Delete_Fixup (T, x)
1.  while x ≠ root [T] & color [x] = Black
2.  do if  x = left [P[x]]
3.  then w ← right [P[x]]
4.     if color [w] = Red
5.  then color [w] ← black.
6.    color [P[x]] ← Red
7.    left-Rotate (T, P[x])
8.    w ← right [P[x]]
9.    if color [left [w]] = Black and color [right [w]] = Black
10.   then  color [w] = red
11.          x ← P[x]
12.   else if color [right [w]] = black.
13.    then color [left [w]] = black.
14.   color [w] ← Red.
15.    Right_rotate (T, w)
16.   w ← Right [P[x]]
17.   else color [w] ← Color [P[x]]
18.   color [P[x]] ← Black
19.   color [right [w]] ← black
20.   left-Rotate (T, P[x])
21.     x ← Root [T]
22.  else (same as the clause with right & left exchanged)
23.   color [x] ← Black.
```

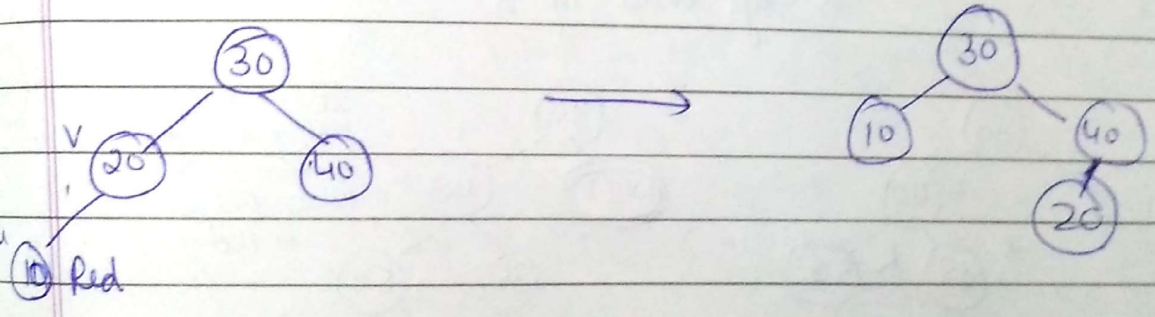
To understand notion of double black is used when a black node is deleted and replaced by a right child and child is marked by double black. The main task is to convert this double black into the single black.

Page No.
Date : / /

Following are the detailed steps for deletion.

Perform standard BST delete. When we perform delete operation we always end up deleting a node which is either a leaf or as only one child (for an internal node we copy the successor and then recursively call delete) 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 (u is null when v is a leaf) and color of null is considered as black.

If u or v is red we mark the replaced child as black (no change in black height). Both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in RB tree.



If both u and v are black.

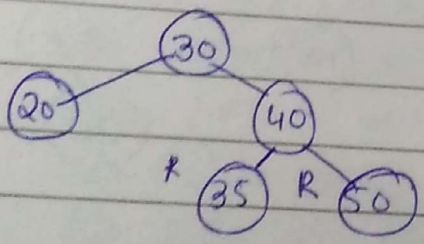
3.1 Color u as double black. Now our ~~over~~ task is to reduce double black to single. Note that v is ~~to~~ leaf u is null and colour of null is considered as black so deletion of black leaf also causes double black.

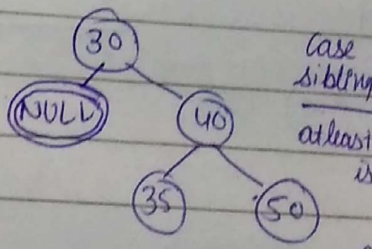
3.2 Do the following while the current node u is double black or it is not root. Let sibling of node be 's'.

a. If sibling 's' is black and atleast one of its children is red perform rotation. Let the red child of s be r.

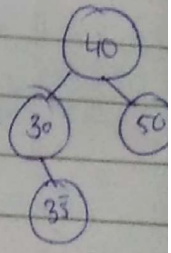
This case can be divided into four sub cases depending upon position of s and x.

- **Left Left case** - 's' is left child of its parent and x is left child of s or both children of s are red. This is mirror image of right right case.
- Left right case - 's' is left child of its parent and x is right child. This is mirror image of right left case.
- Right left case - s is the right child of its parent and x is the right child or both children of s are red.
- Right right case - s is the right child of its parent and x is left child of s.





Case 3.2 a(iii)
Sibling is black
atleast one child is red



RR Case

(b) If sibling is red perform a rotation to move old sibling up. Recolour the old sibling and parent, the new sibling is always black this mainly converts the tree to black sibling case by rotation and leads to the previous case. This can be divided into two cases

- s is the left child of its parent. This mirror of right right case. We perform the right rotation on parent p.

Page No.
Date : / /

s. is the right child of its parent. we left rotate the parent p