UNIT II - HIERARCHICAL DATA STRUCTURES

Binary Search Trees: Basics – Querying a Binary search tree – Insertion and Deletion- Red Black trees: Properties of Red-Black Trees – Rotations – Insertion – Deletion -B-Trees: Definition of B - trees – Basic operations on B-Trees – Deleting a key from a B-Tree- Heap – Heap Implementation – Disjoint Sets - Fibonacci Heaps: structure – Mergeable-heap operations- Decreasing a key and deleting a node-Bounding the maximum degree.

RED-BLACK TREES

Red-black trees are one of many search-tree schemes that are balanced to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

PROPERTIES OF RED-BLACK TREES

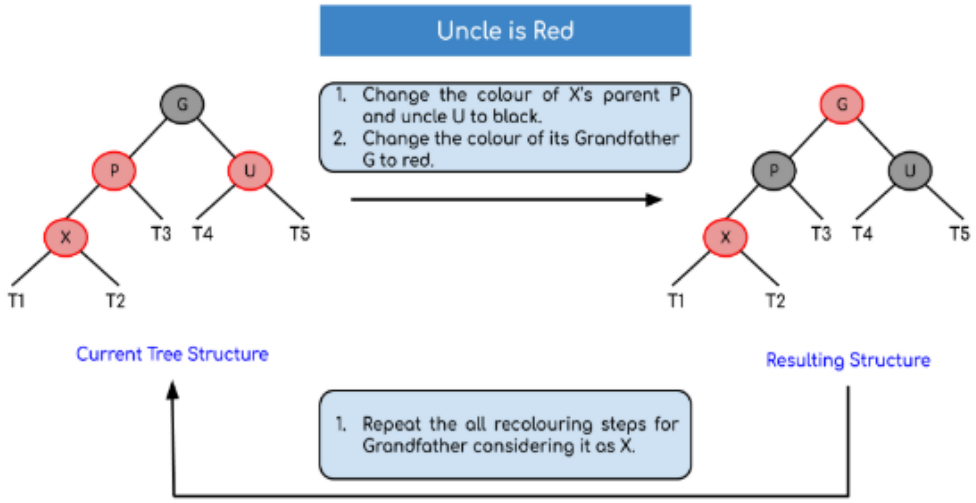
A red-black tree is a binary search tree with one extra bit of storage per node: its color either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, it ensures that no such path is more than twice as long as any other, making the tree balanced. The height of a red-black tree with n keys is at most $2\lg(n + 1)$, which is $O(\lg n)$. Each node of the tree contains the attributes color, key, left, right and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.

A red-black tree is a binary search tree that satisûes the following red-black properties:

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

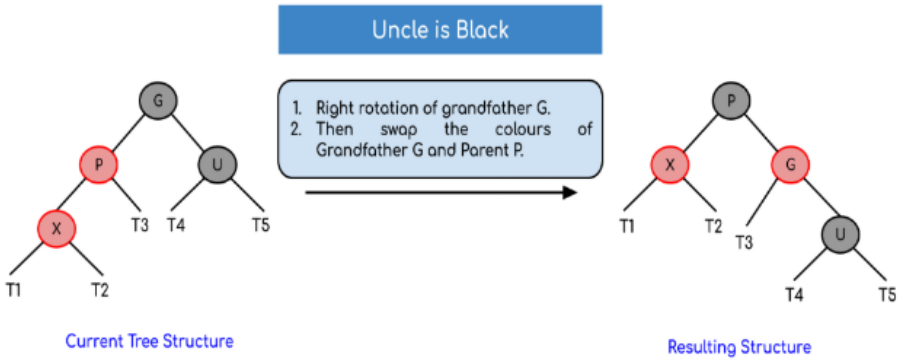
Insertion Logic

Insert the node similar to a binary tree and assign a red color to it. If the node is a root node then change its color to black. If it is not then check the color of the parent node. If its color is black then don't change the color but if it is red then check the color of the node's uncle. If the node's uncle is red then change the color of the node's parent and uncle to black and grandfather to red color. If grandfather is root then don't change grandfather to red color.

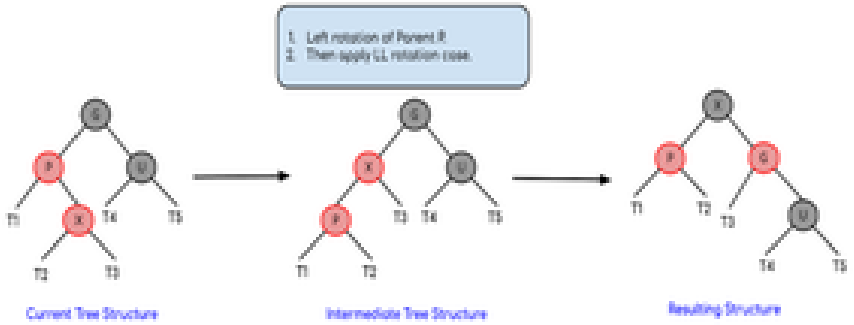


If the node's uncle is black, then there are four possible cases

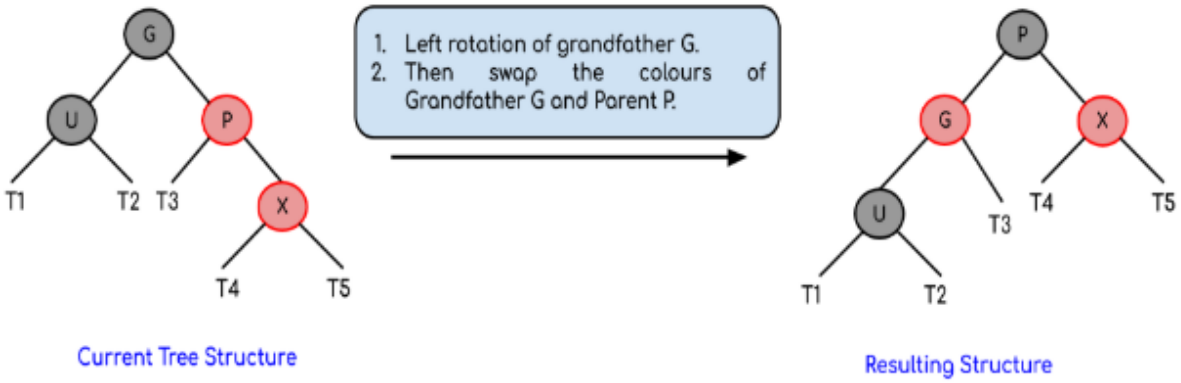
1. Left Left Rotation



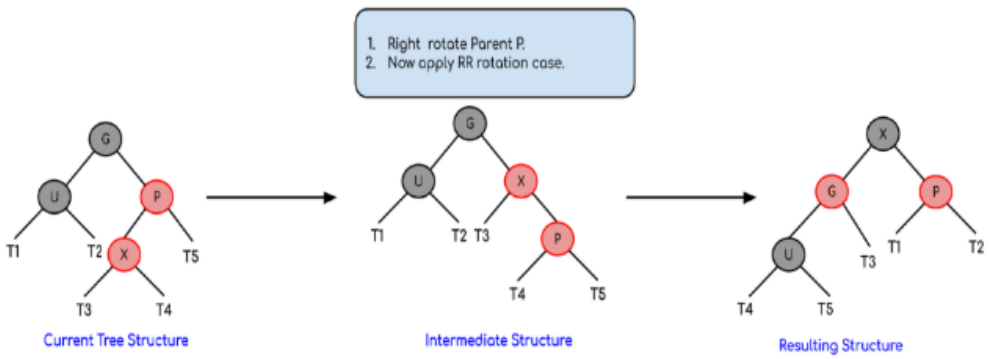
2. Left Right Rotation



3. Right Right Rotation



4. Right Left Rotation

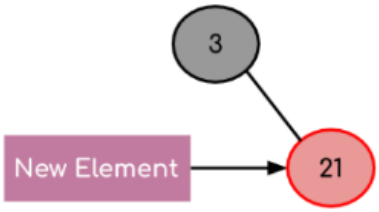


Creating a red-black tree with elements 3, 21, 32 and 15 in an empty tree.

Step 1: Inserting element 3 inside the tree.

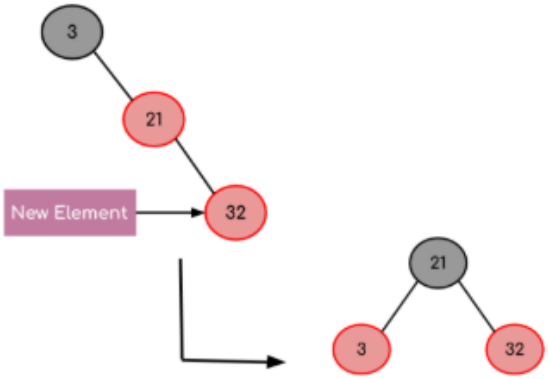
Since the first element is the root node, the color of the node is changed as black

Step 2: Inserting element 21 inside the tree.



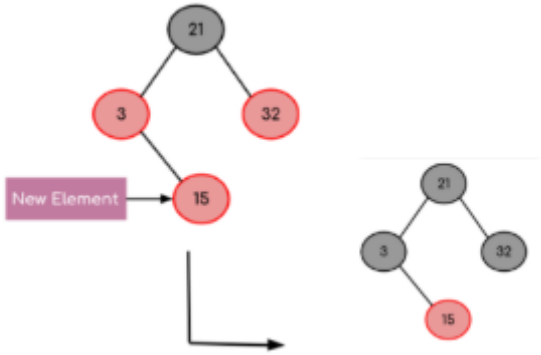
As Red Black tree is a Binary search tree, the new key 21 is checked with 3 and inserted in right and as a new node it is colored as red.

Step 3: Inserting element 32 inside the tree.



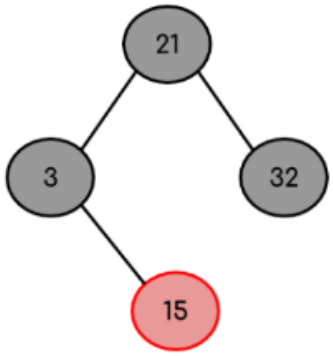
32 is inserted to the right of 21 and colored as red. Two red nodes are not possible so it needs a RR Rotation and we need to recolor to balance the tree. After rotating 21 becomes the root node which is red. But a root node cannot be red, So we need to recolor it as black. And 3 is the grandparent of 32 and hence recolor it as red.

Step 4: Inserting element 15 inside the tree.



15 is inserted to the right of 3 and colored as red. Two red nodes are not possible and as the uncle node is red, change the color of parent and uncle as black.

The Final tree structure is



DELETION

Deletion of a node in a red black tree takes O(lg n) time. Deleting a node from a red-black tree is more complicated than inserting a node. The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure. RB-DELETE AND RB-DELETE-FIXUP is used to perform deletion in Red-Black Tree.

RB-DELETE (T, z)

```
1 y = z
2 y-original-color = y.color
3 if z.left = = T.nil
4       x = z.right
5       RB-TRANSPLANT (T, z, z.right )        // replace z by its right child
6 elseif z.right = = T.nil
7       x = z.left
```

```
8       RB-TRANSPLANT (T, z, z.left )          // replace z by its left child
9 else y = TREE-MINIMUM(z.right )              // y is z's successor
10      y-original-color = y.color
11      x = y.right
12      if y ≠ z.right // is y farther down the tree?
13              RB-TRANSPLANT (T, y, y.right ) // replace y by its right child
14              y.right = z.right                    // z's right child becomes
15              y.right .p = y                       // y's right child
16      else x.p = y                          // in case x is T.nil
17      RB-TRANSPLANT (T, z, y)               // replace z by its successor y
18       y.left = z.left                      // and give z's left child to y,
19      y.left.p = y                          // which had no left child
20      y.color = z.color
21 if y-original-color = = BLACK              // if any red-black violations occurred,
22      RB-DELETE-FIXUP (T, x)                // correct them
```

RB-DELETE-FIXUP (T, x)

```
1 while x ≠ T.root and x.color == BLACK
2       if x == x.p.left // is x a left child?
3               w = x.p.right // w is x's sibling
4               if w.color == RED
5                       w.color = BLACK
 6                      x.p.color = RED
7                       LEFT-ROTATE(T, x.p)
8                       w = x.p.right
9               if w.left.color == BLACK and w.right.color == BLACK
10                      w.color D RED
11                      x = x.p
12              else
13                      if w.right.color == BLACK
14                              w.left.color = BLACK
15                              w.color = RED
16                              RIGHT-ROTATE(T, w)
17                              w = x.p.right
18                      w.color = x.p.color
```

```
19                  x.p.color = BLACK
20                  w.right.color = BLACK
21                  LEFT-ROTATE(T, x.p)
22                  x = T.root
23          else // same as lines 3-22, but with “right” and “left” exchanged
24                  w = x.p.left
25                  if w.color == RED
26                          w.color = BLACK
27                          x.p.color = RED
28                          RIGHT-ROTATE(T, x.p)
29                          w = x.p.left
30                  if w.right.color == BLACK and w.left.color == BLACK
31                          w.color = RED
32                          x = x.p
33                  else
34                          if w.left.color == BLACK
35                                  w.right.color = BLACK
36                                  w.color = RED
37                                  LEFT-ROTATE(T, w)
38                                  w = x.p.left
39                          w.color = x.p.color
40                          x.p.color = BLACK
41                          w.left.color = BLACK
42                          RIGHT-ROTATE(T, x.p)
43                          x = T.root
44 x.color = BLACK
```

Case 1: If the node to be deleted is red, then delete it.

Case 2: If the node to be deleted is double black, then check if it is root.

If it is root then remove double black and make it as black node

Case 3: If the double black node is not a root node, then check the sibling of double black node

If the double black node’s sibling and the children are black, then

a. Remove double black
b. Add black to parent
    i. If the parent is red, then recolor it as black.
    ii. If the parent is black then make it double black.

c. Make the sibling as red

Case 4: If double black node's sibling is red and children are black

Action 1: Swap the color of sibling and the parent of DB

Action 2: Do the rotation towards the double black and reapply the cases.

Case 5: If a double black sibling is black and the children are not black i.e). one is red and the red is near to double black

Action 1: Swap the color of sibling and sibling child.

Action 2: Do the rotation opposite to double black

Case 6: If a double black sibling is black and the children are not black i.e) one is red and the red is far away to double black.

Action 1: Swap the color of parent and sibling of double black

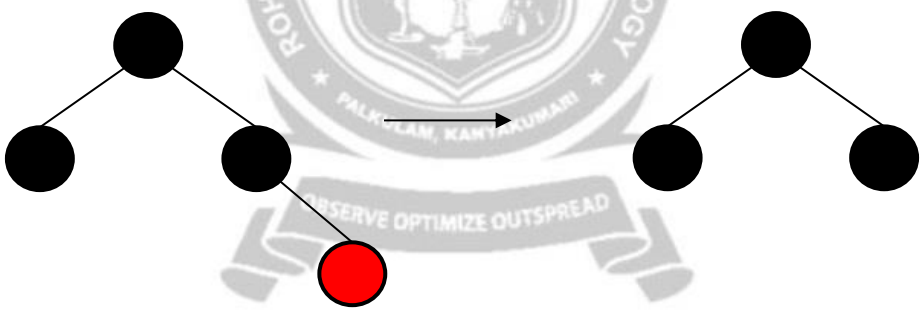
Action 2: Rotate towards the Double black

Action 3: Remove the double black and mark the far red child as black

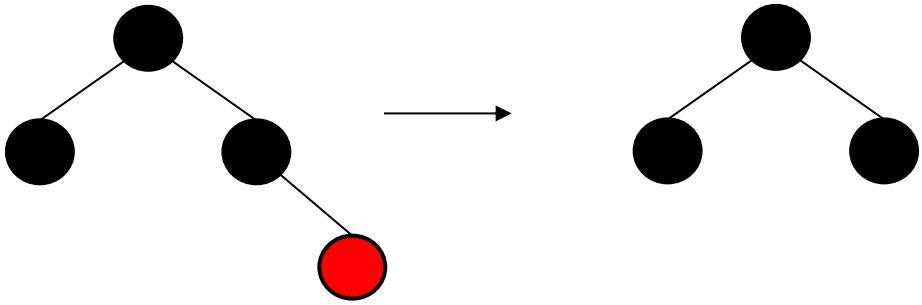
Example:

Case 1: If the node to be deleted is red, then delete it.

Delete 30: Node 30 is red and leaf node. So delete it.



Delete 20: Node 20 is an internal node and has single child 30 which is red. Find the inorder successor of the right subtree. Inorder successor is 30. So Replace the value 30 in the place of 20 and delete leaf red node 30

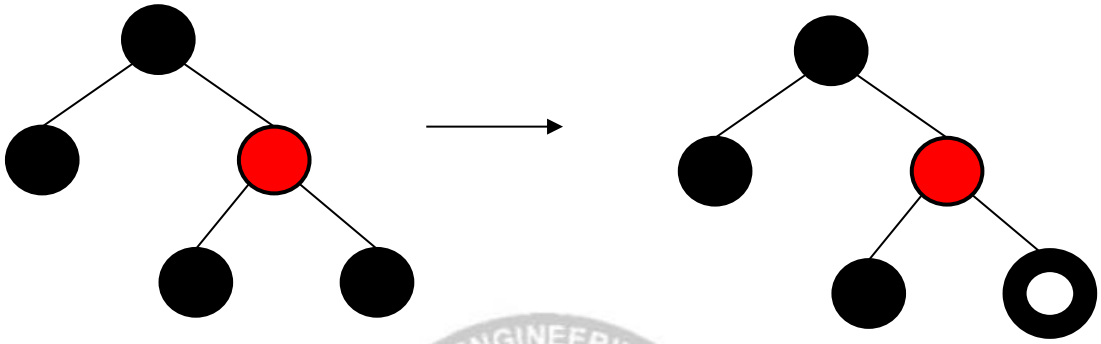


Case 2: If the node to be deleted is double black, then check if it is root.

Case 3: If the double black node is not a root node, then check the sibling of double black node

If the double black nodes sibling and the children are black

Delete 20

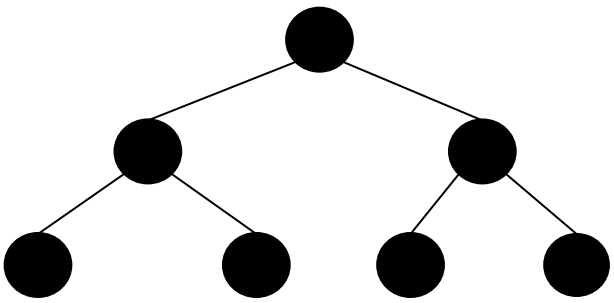


Double Black Children is black and sibling 15 is also black, then

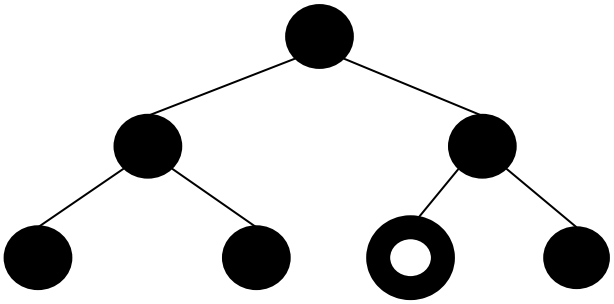
a. Remove double black
b. Add black to parent i.e) 30 is red which becomes black
c. Make the sibling as red i.e) 15 becomes red

If still double black exist after deleting then do the following:

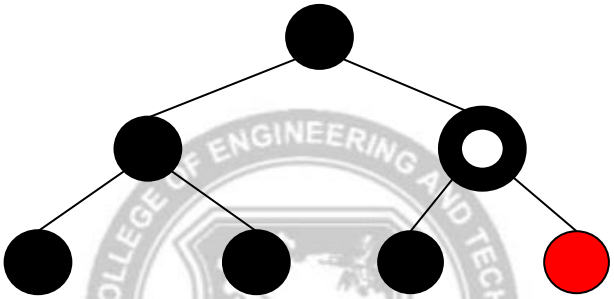
Delete 15



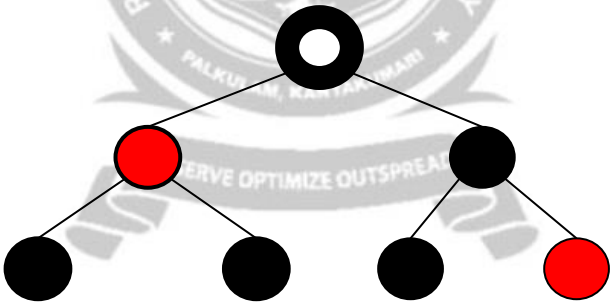
15 is Double Black and its Children are black and so it becomes Double Black



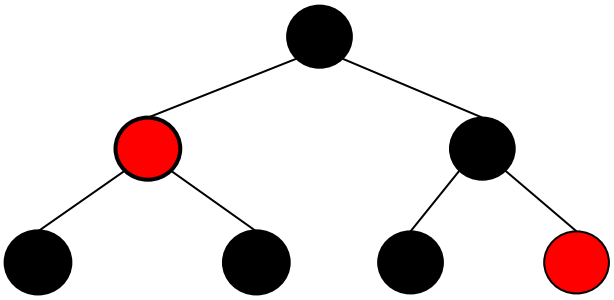
a. Remove double black
b. Add black to parent i.e) 20 is black which becomes double black
c. Make the sibling as red i.e) 30 becomes red

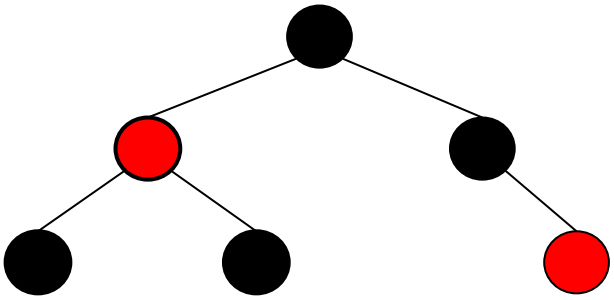


Still Double Black exists in node 20
Check sibling and it is black and so check the root and transfer double black to parent and make sibling red



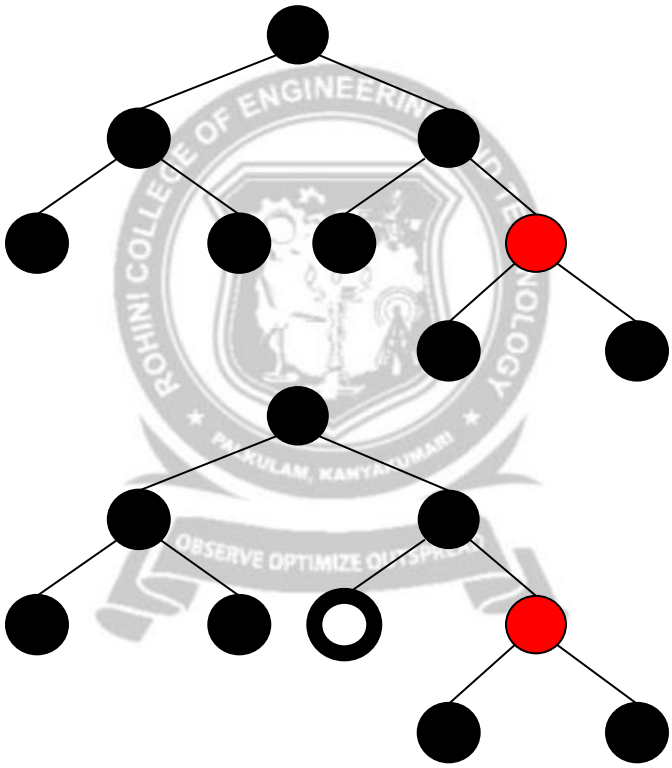
Still Double Black exists in node 10. But it is the root. So just remove it





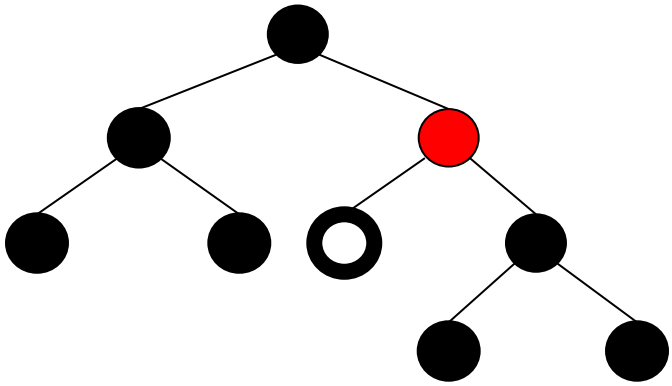
Case 4: If double black node's sibling is red

Delete 15

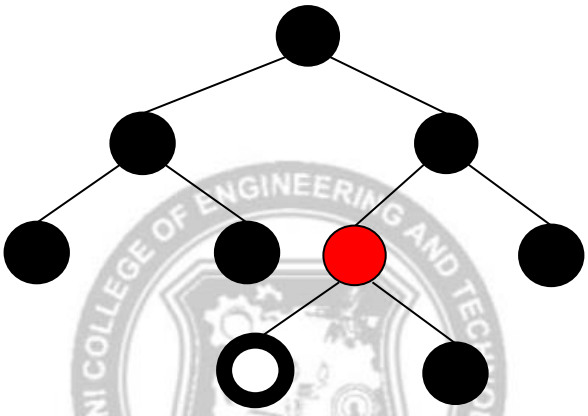


Double black node's sibling is red

Action 1: Swap the color of sibling 30 and the parent 20 of Double Black

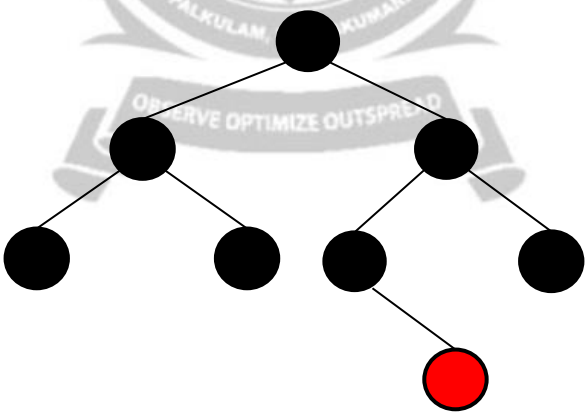


Action 2: Do the rotation towards the double black and reapply the cases.



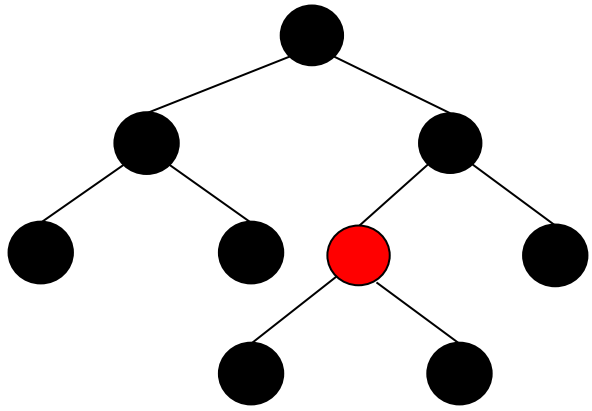
Reapply the cases since there is double black and it follow the case 3

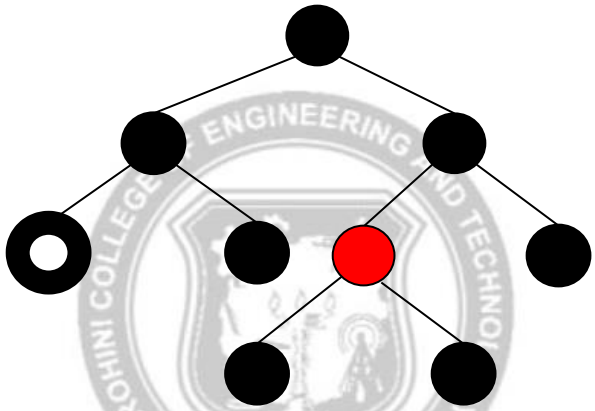
- Move the double black to parent 20 which is red and it becomes black
- Make the sibling 25 as red and remove the node to be deleted



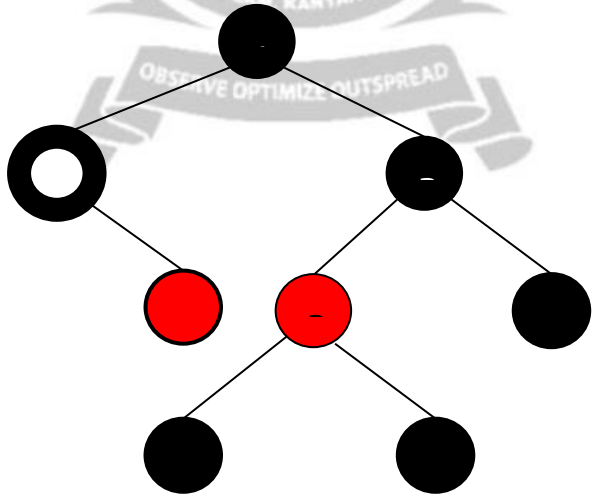
Case 5: If a double black sibling is black and the children are not black i.e). one is red and the red is near to double black

Delete 1



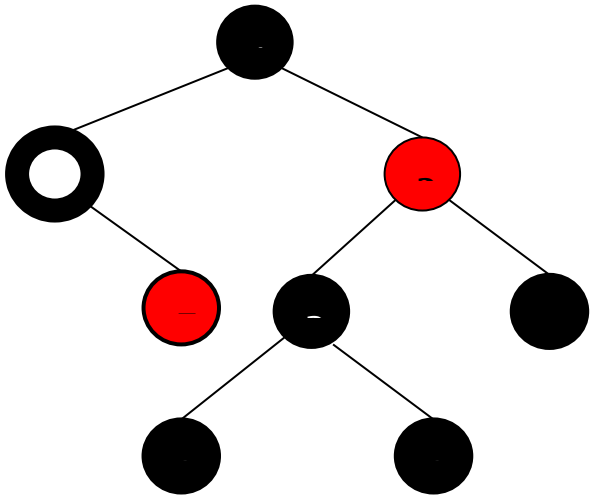


Double black Sibling 7 is black. So Move double black to parent 5 and make the sibling 7 as red and remove nil node

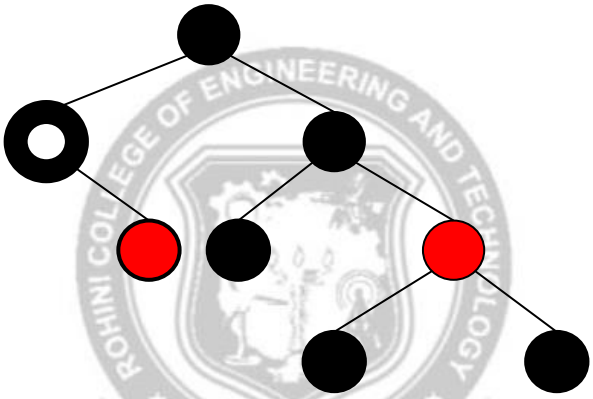


Still Double black exists. Now the double black node is 5 and the sibling of node 5 is black and all children are not black. One of its children is red and it is near to Double black.

Action 1: Swap the color of sibling 30 and sibling child 25.



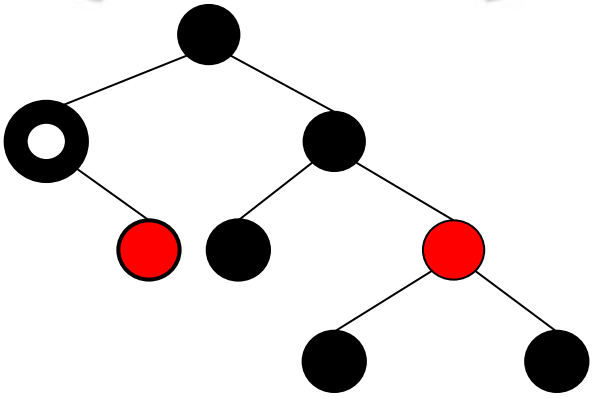
Action 2: Do the rotation opposite to double black



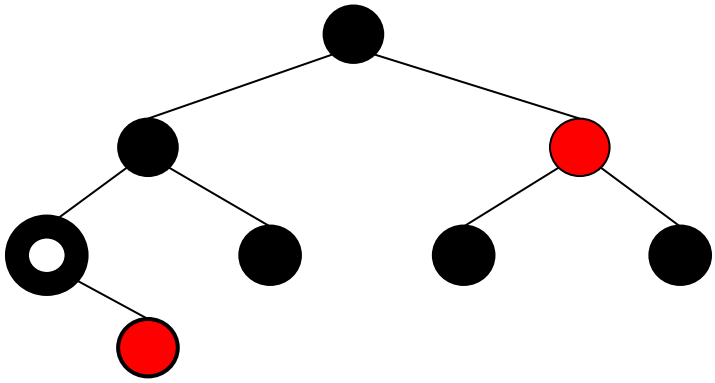
This follows **Case 6. Since the double black node's sibling is black and one of its children is red and it is far away from the double black node.**

Action 1: Swap the color of parent and sibling of double black

Since the parent of double black 10 is black and the sibling 25 also black, it doesn't make any change



Action 2: Rotate towards the Double black



Action 3: Remove the double black and mark the far red child as black