RED BLACK TREE

Mahip Adhikari(03), Aakriti Banjara(05), Jessica Thapa(55)

WHY RED BLACK TREE OVER BST AND AVL?

- Red Black Tree are Roughly height-Balanced tree.
- It guarantees for the time complexity O(log n) for all the operations.
- It only requires Max two rotations and recoloring.

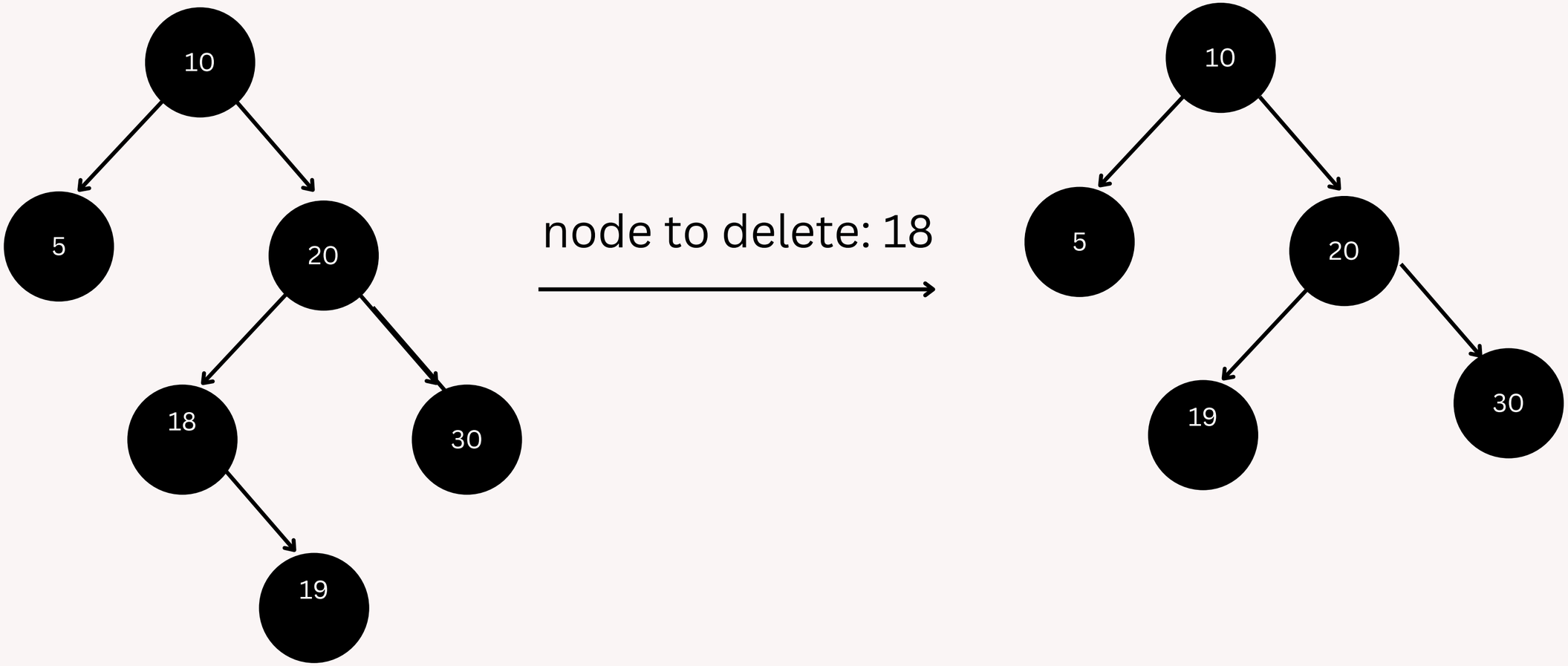
DELETION STEPS

Step 1: Perform the standard BST delete operation

- **To delete leaf node**
- **To delete node having single child**
- **To delete node having two child**

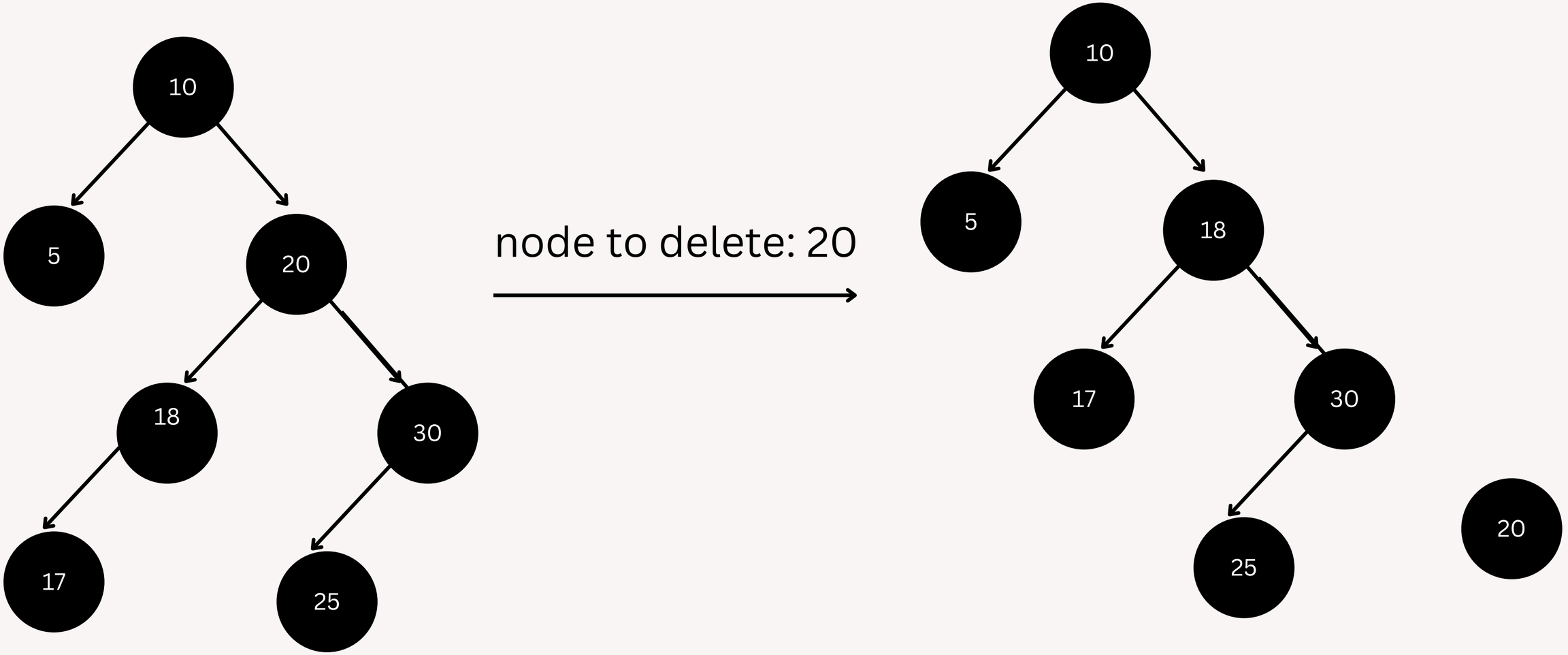
To delete leaf node

To delete node having single child



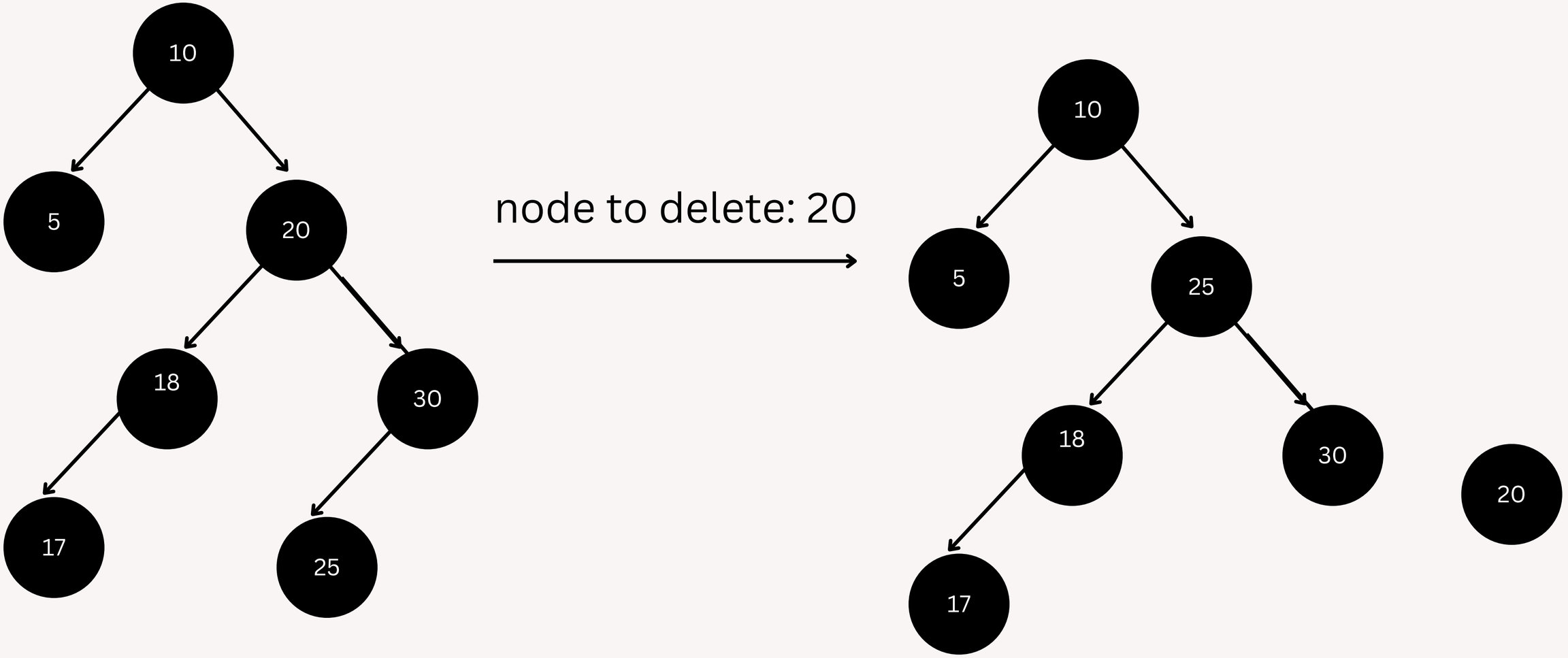
To delete node having two child

CASE A: In left subtree replace with most big element (Inorder predecessor)



To delete node having two child

CASE B: In right subtree replace with most small element (Inorder Sucessor)



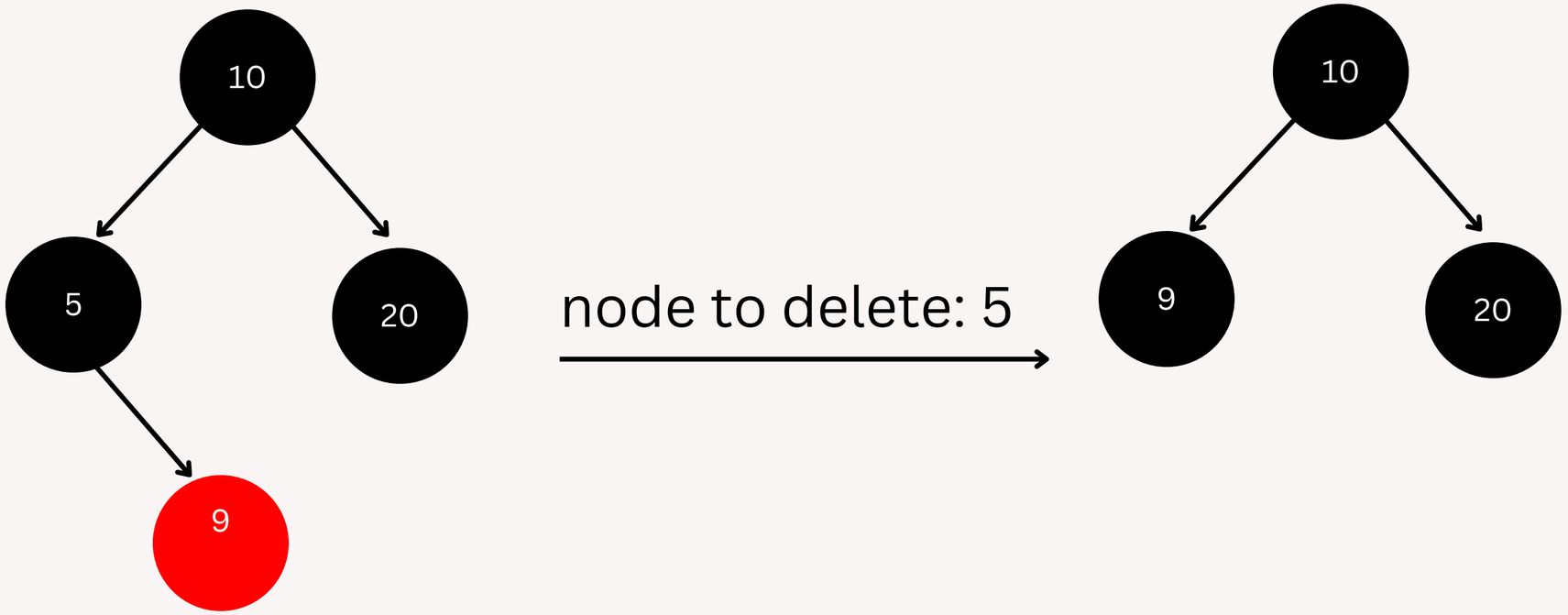
DELETION STEPS

Step 2: Three cases are arises when current node is replaced with another node, i.e. its successor node or predecessor node

- **Either A or B is RED**
- **Both A and B are RED**
- **Both A and B are BLACK**

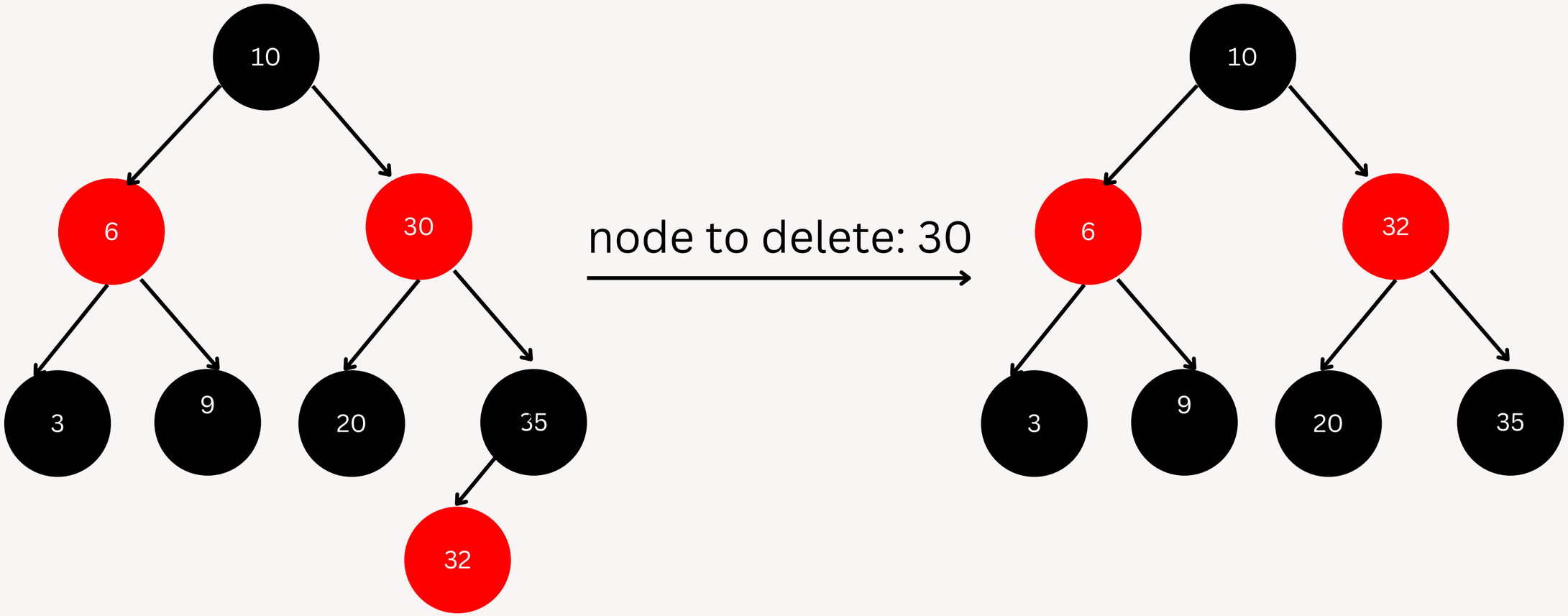
CASE-1 : EITHER A OR B IS RED

If either the node to be deleted (A) is red or the node that replace it (B) is red then, mark the replaced child as black.



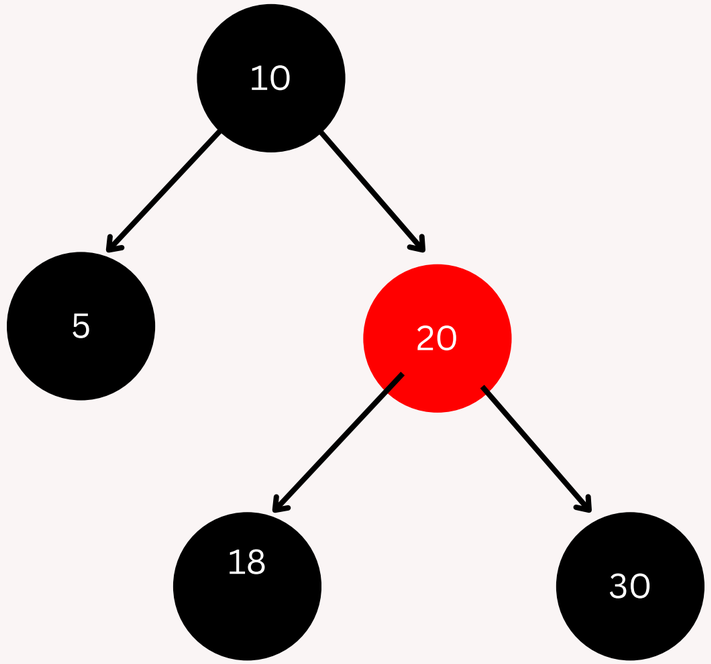
CASE-II: BOTH A AND B ARE RED

If both A and B are red, the deletion is same as Binary Search Tree.



CASE-III

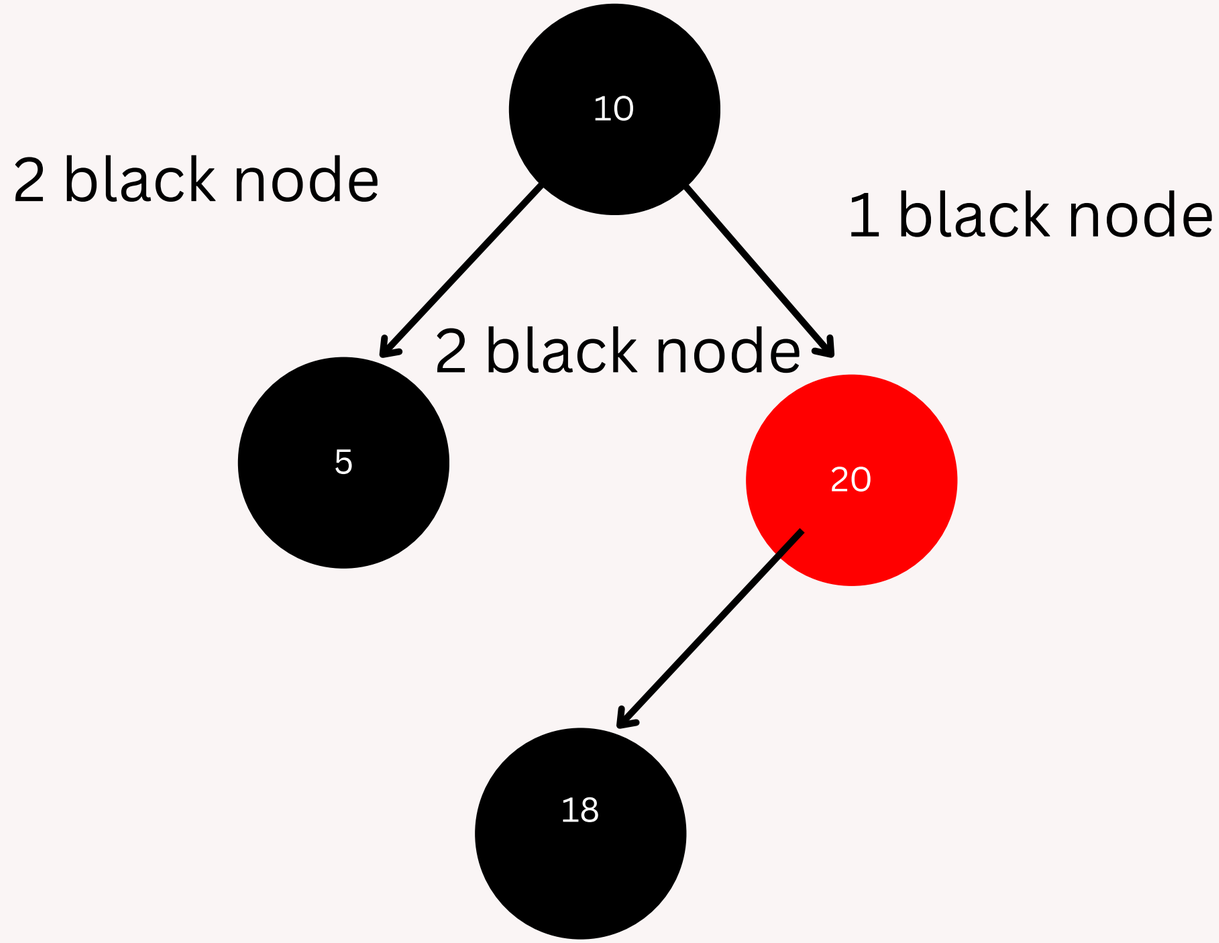
When both node are black: node to delete (A) and either predecessor or successor (B)



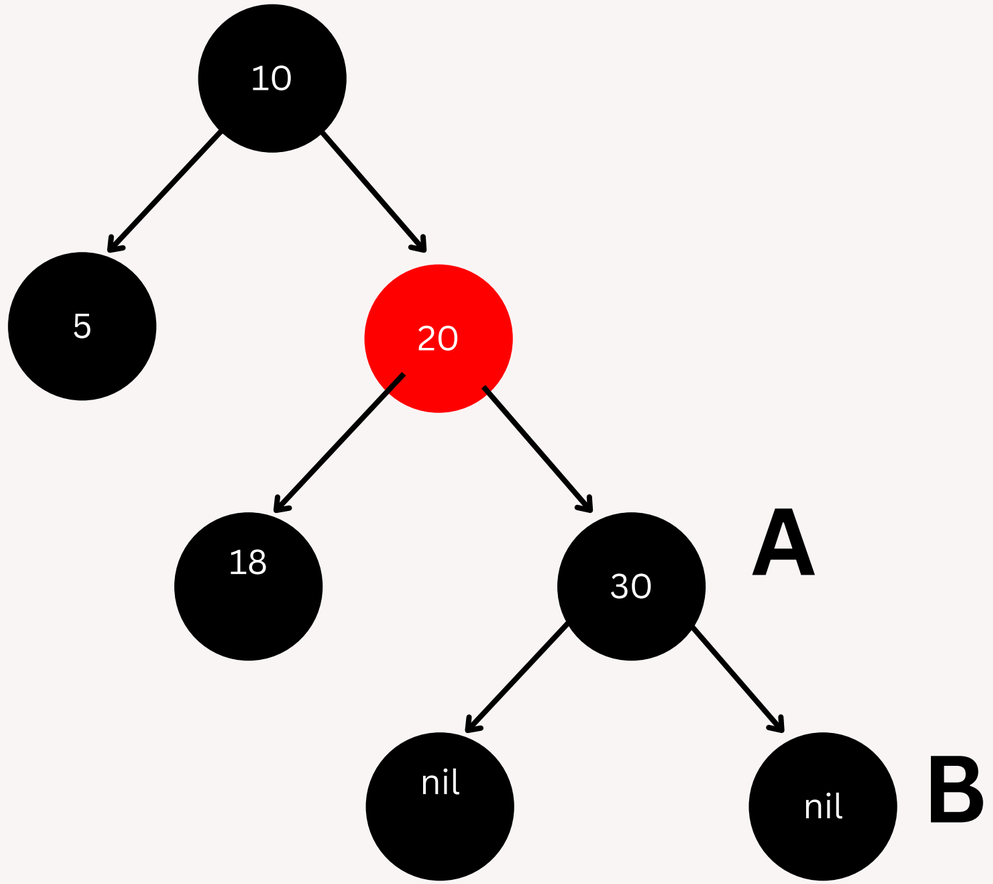
node to delete: 30

Is this red black tree?

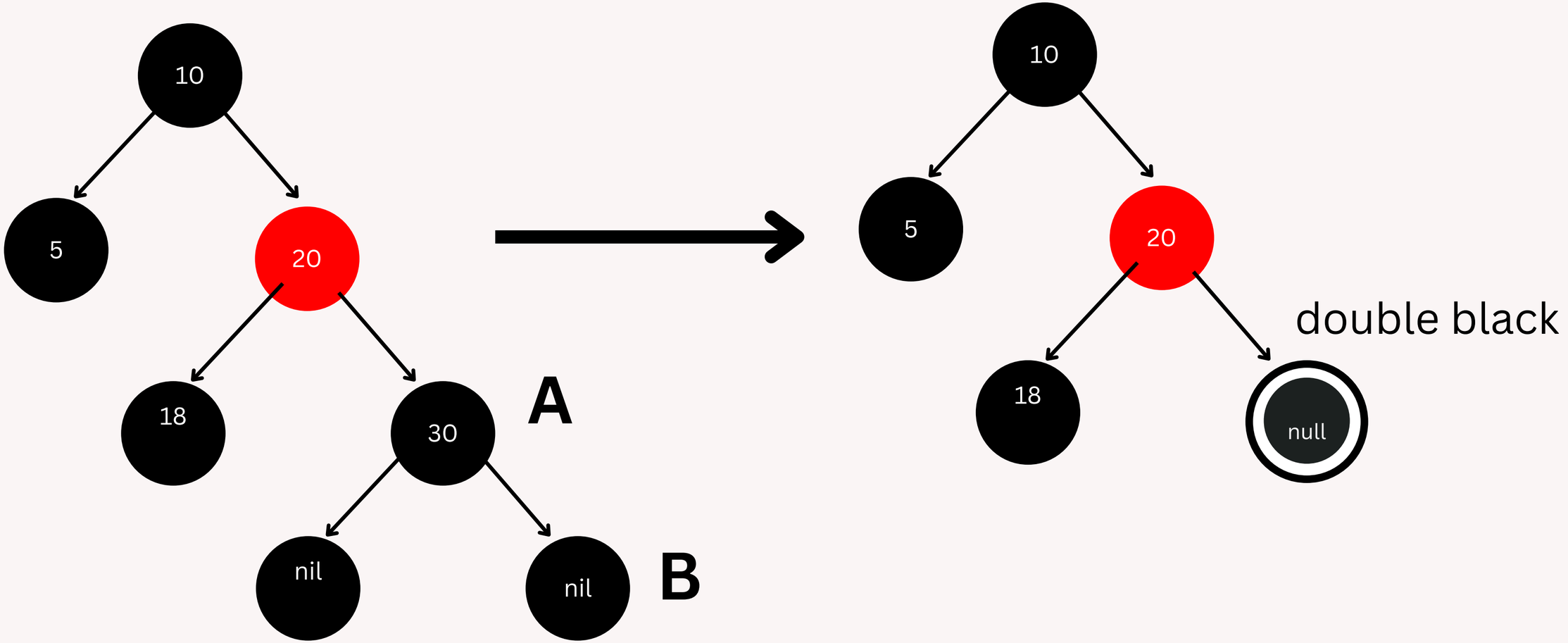
Deleting 30 directly would violate the property that total number of black node should be same in each path



can also be written as



Color B as double black. This will violate one of the properties of red black tree, so we need to convert this double black to single black.

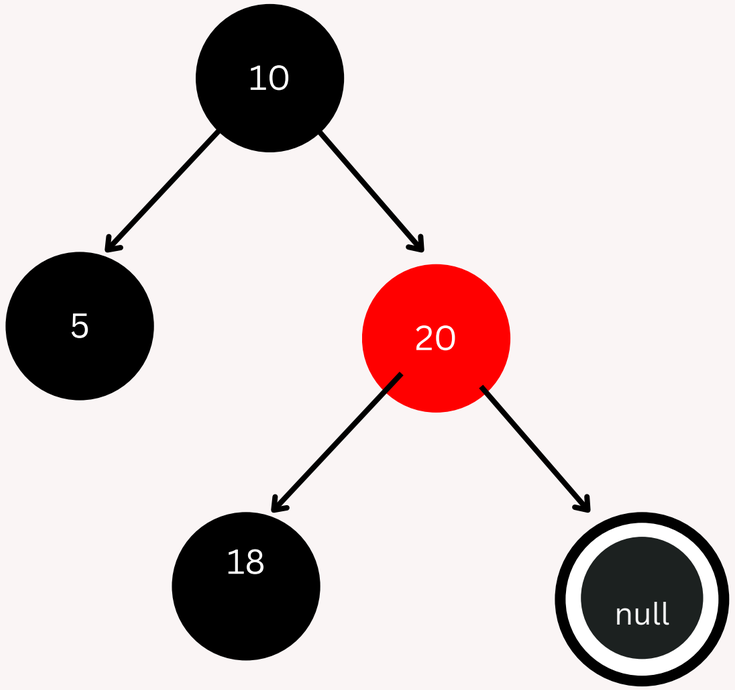


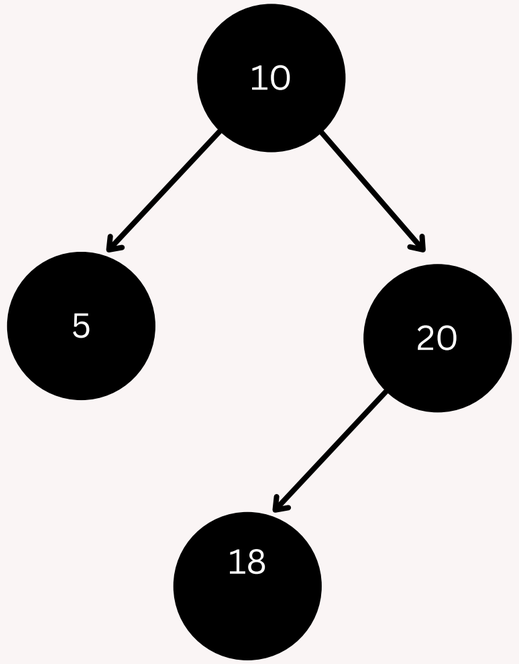
need to convert this double black as single black

sub case: III-I

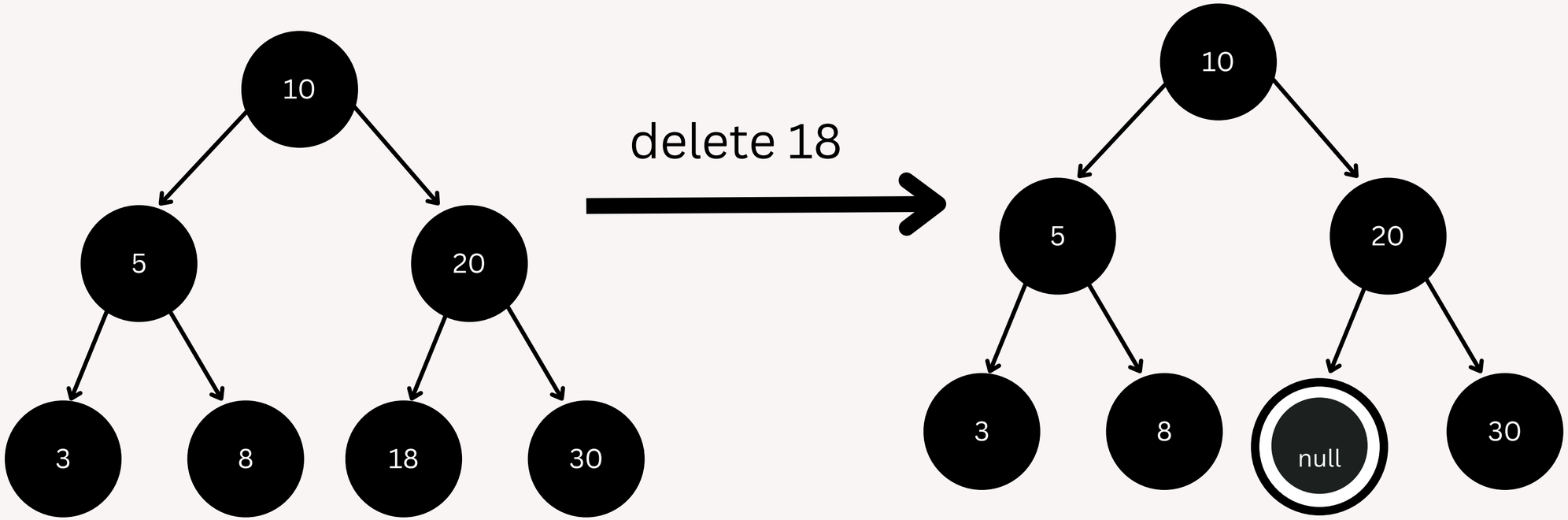
If DB sibling is black and both its children are black then:

1. **Remove DB**
2. **Add black to its parent**
    a. **if parent is black it becomes Double black**
    b. **if parent is red it becomes black**
3. **make sibling red**



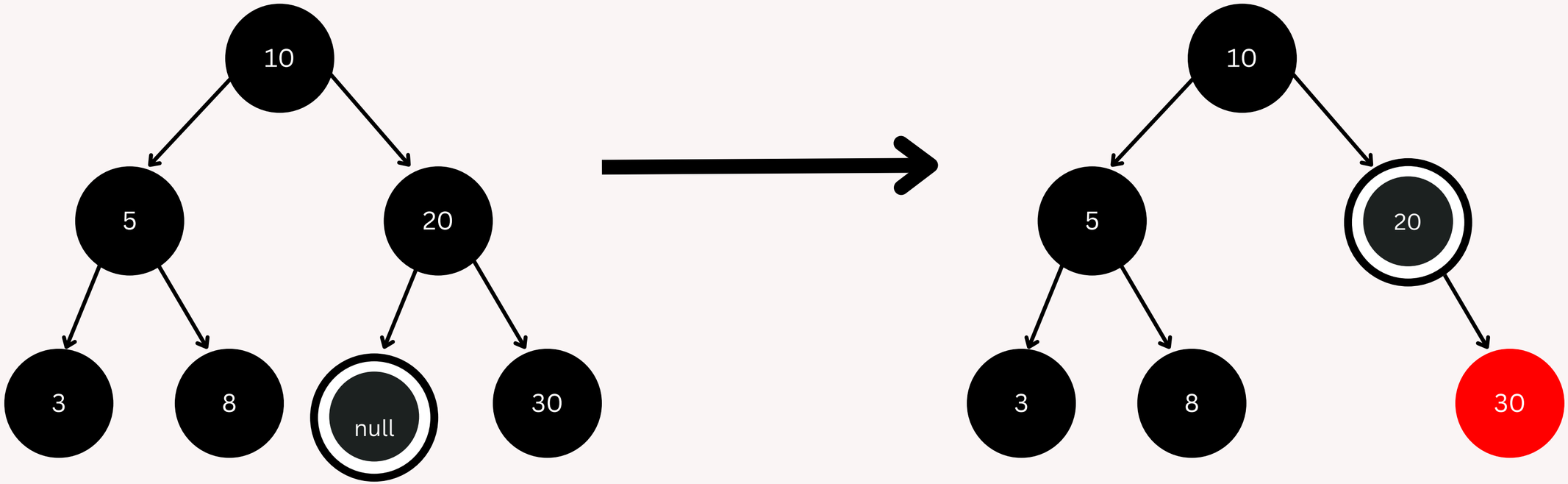


when DB parent is black and its sibling as well as its children are black



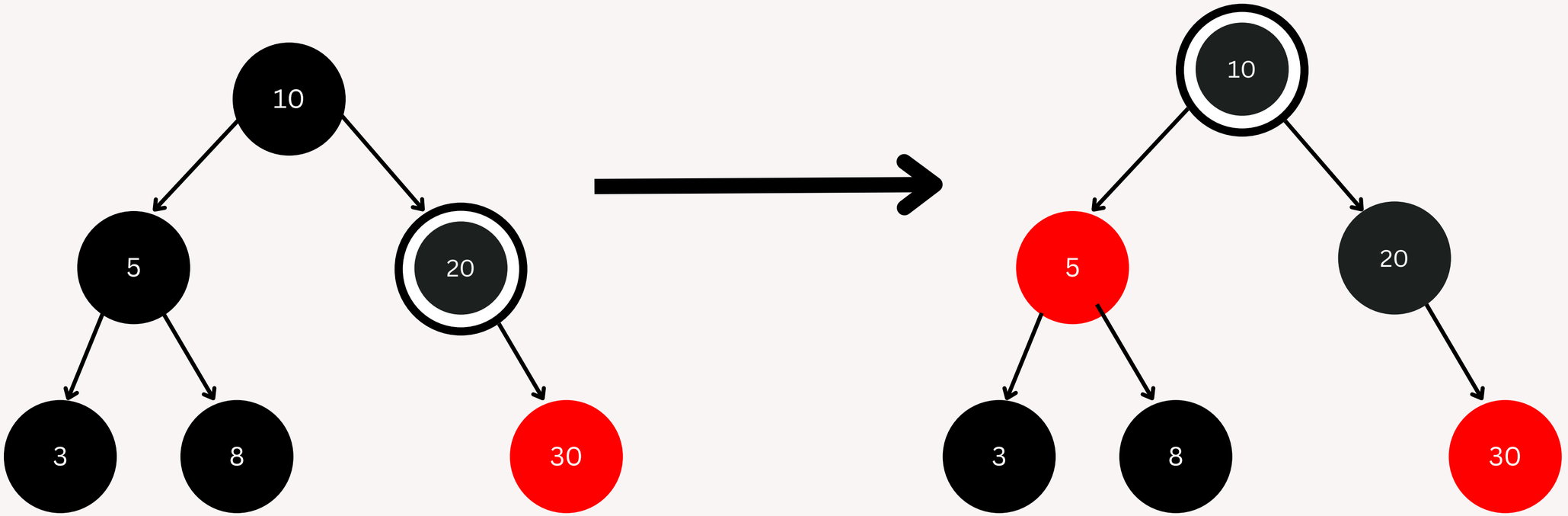
when DB parent is black and its sibling as well as its children are black

a. making the parent double black
b. making the sibling red



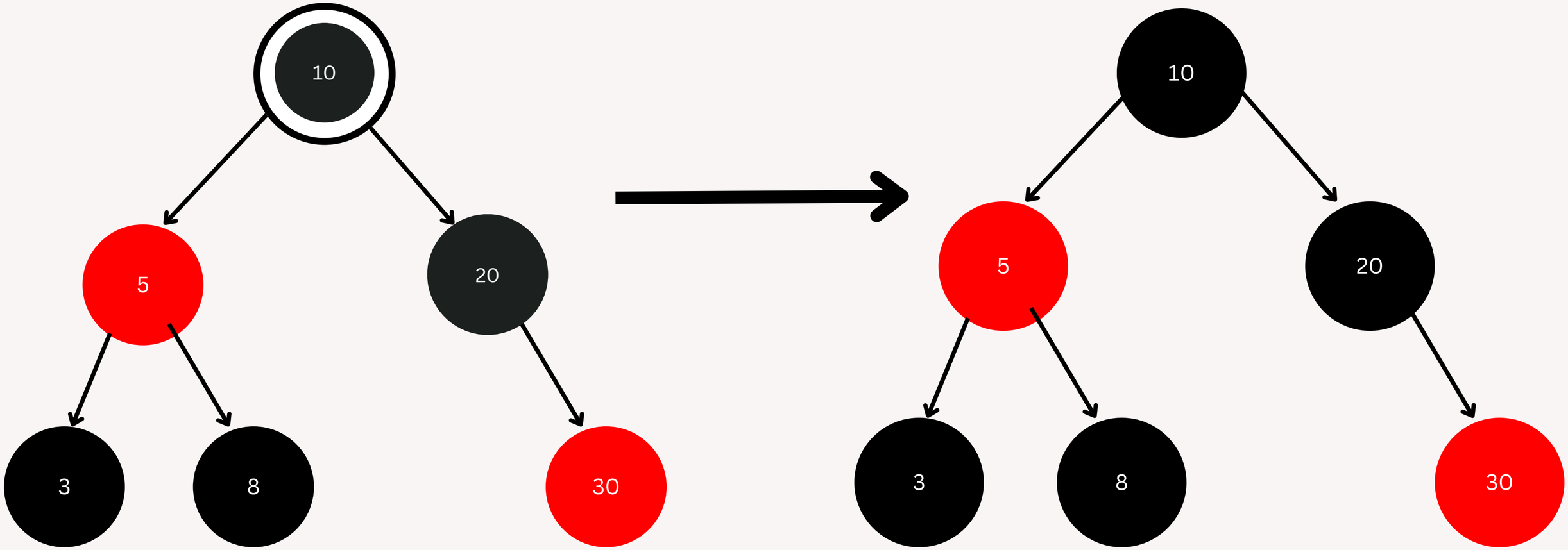
when DB parent is black and its sibling as well as its children are black

a. making the parent double black
b. making the sibling red



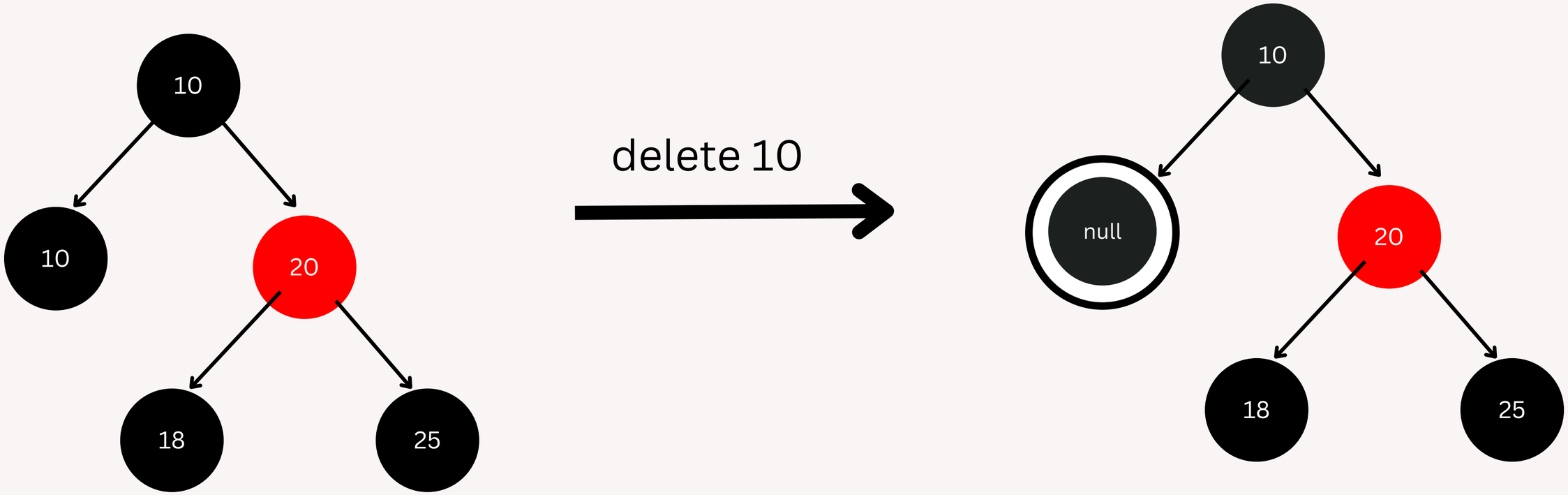
the root is double black?

if the root is double black then we can simply remove the double black from root



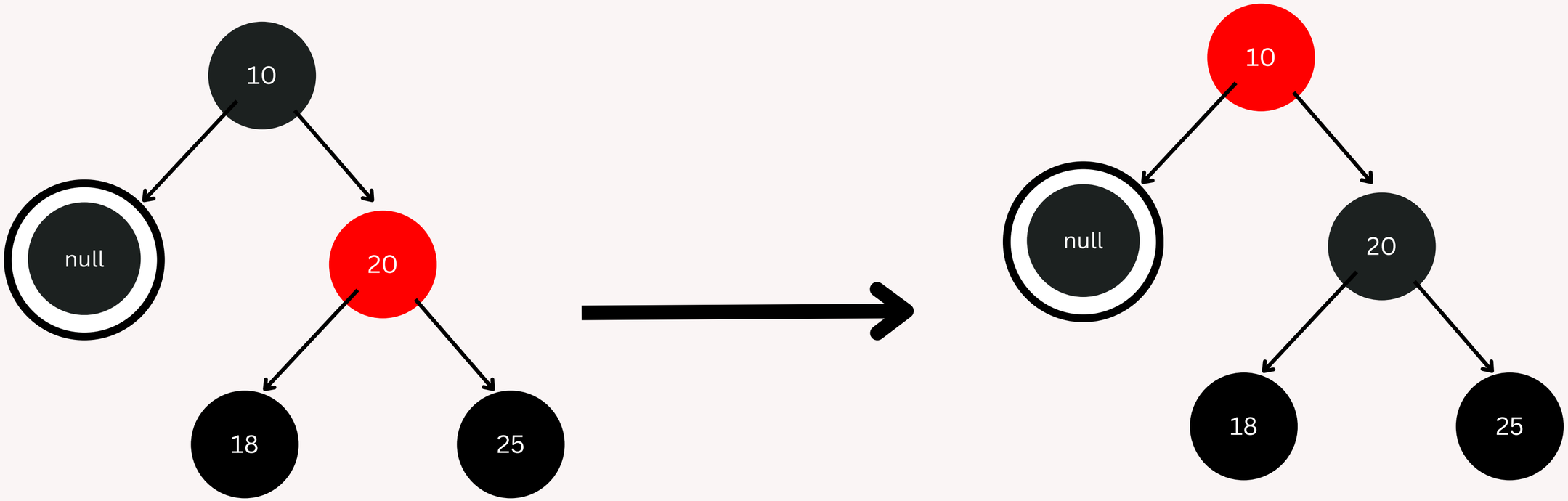
sub case: III-II

If DB sibling is red then:
a. swap color of parent and sibling.
b. rotate the Parent towards DB direction
c. re apply cases

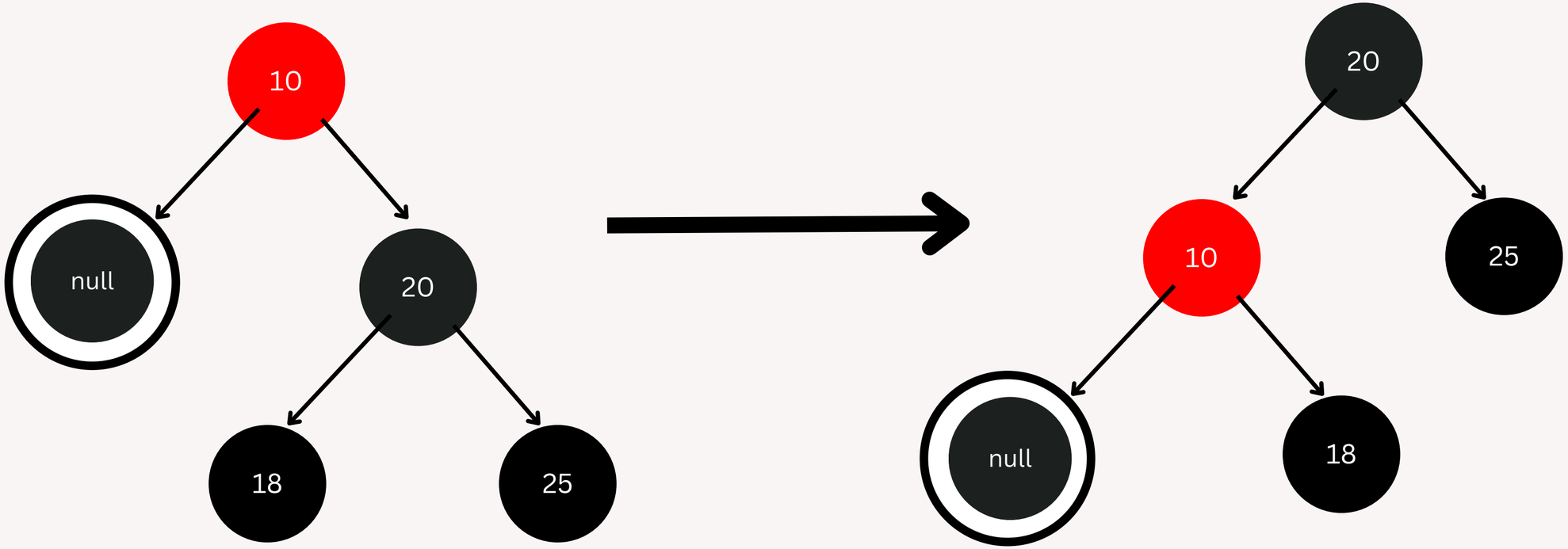


root cannot be red but this is an intermediate tree

swap color of parent and sibling.

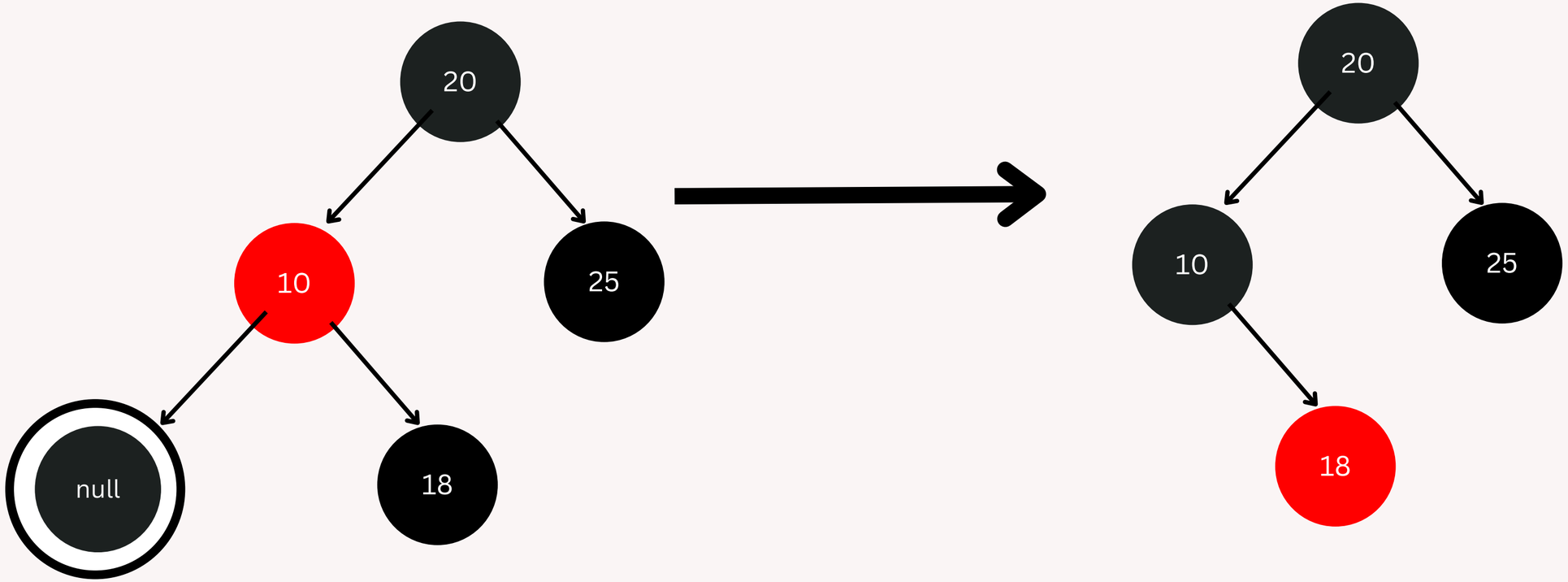


rotate the Parent towards DB direction i.e. in left direction for this case



now this is exactly like sub case II-III

change the parent color to black and then sibling to red



sub case: III-III

When Sibling is Black and any one sibling’s child is Red

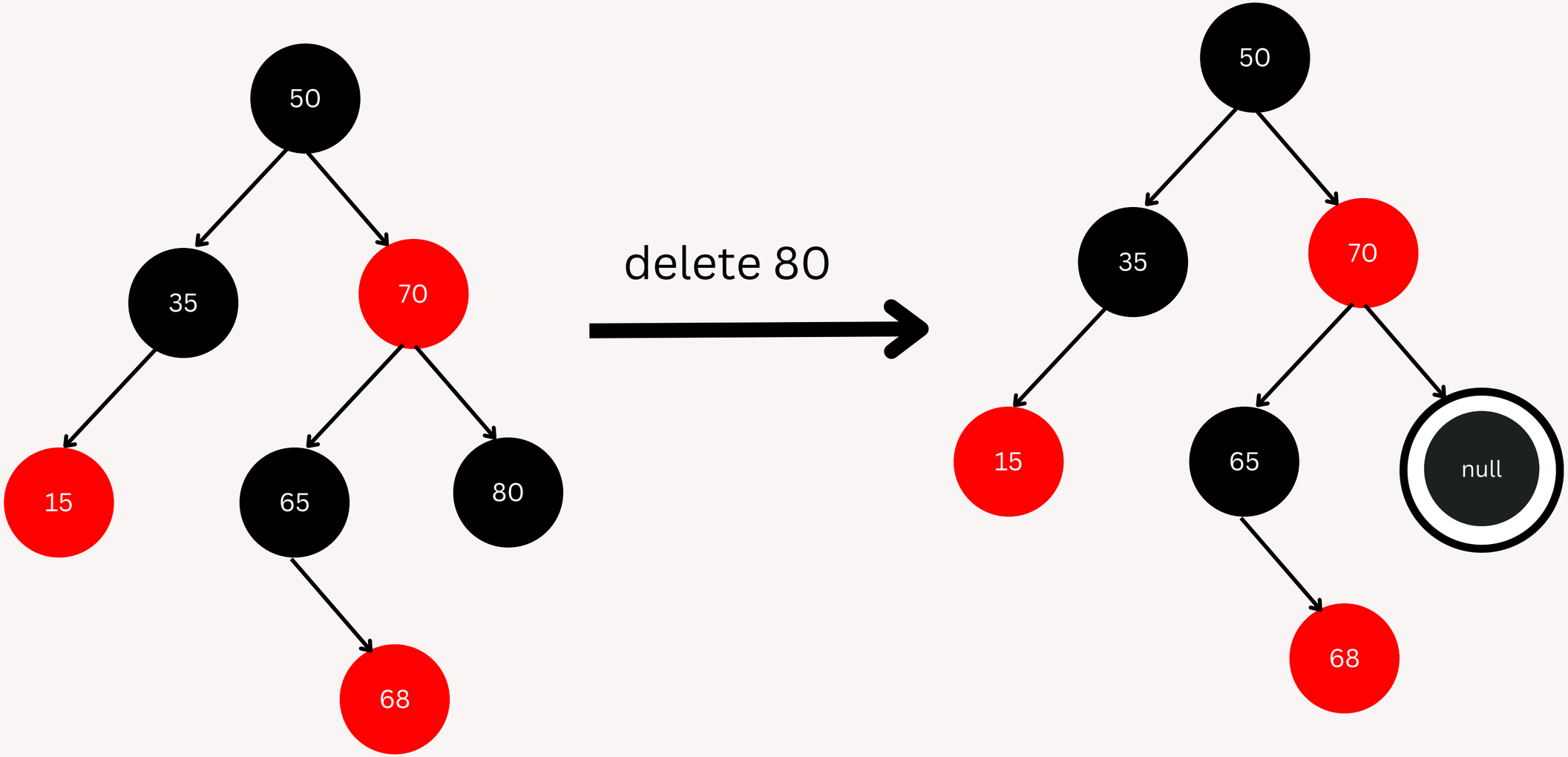
There are two cases that can appear:

a)When Sibling is Black and sibling’s near child is Red

b)When Sibling is Black and sibling’s far child is Red

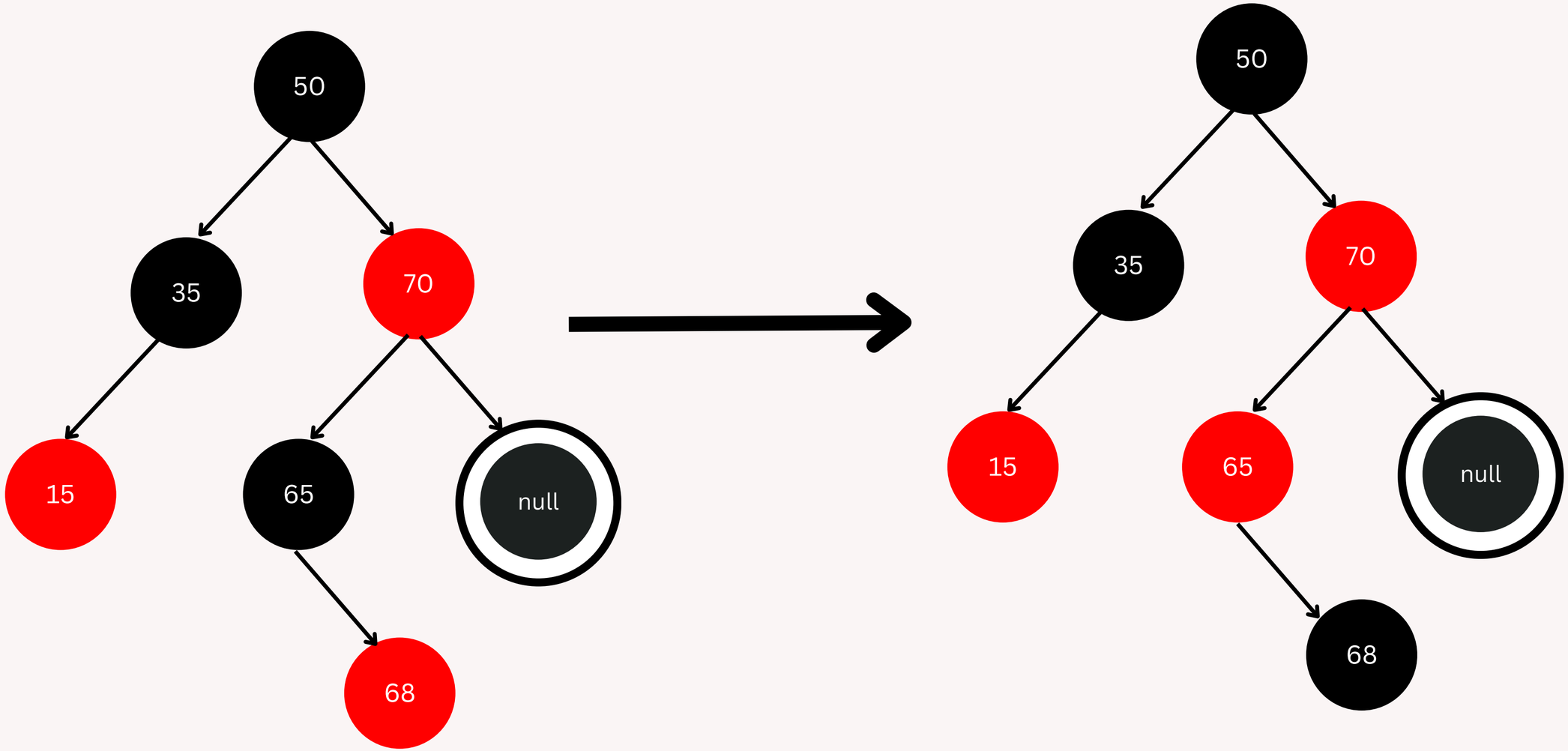
sub case: III-III

When Sibling is Black and sibling's near child is Red



sub case: III-III

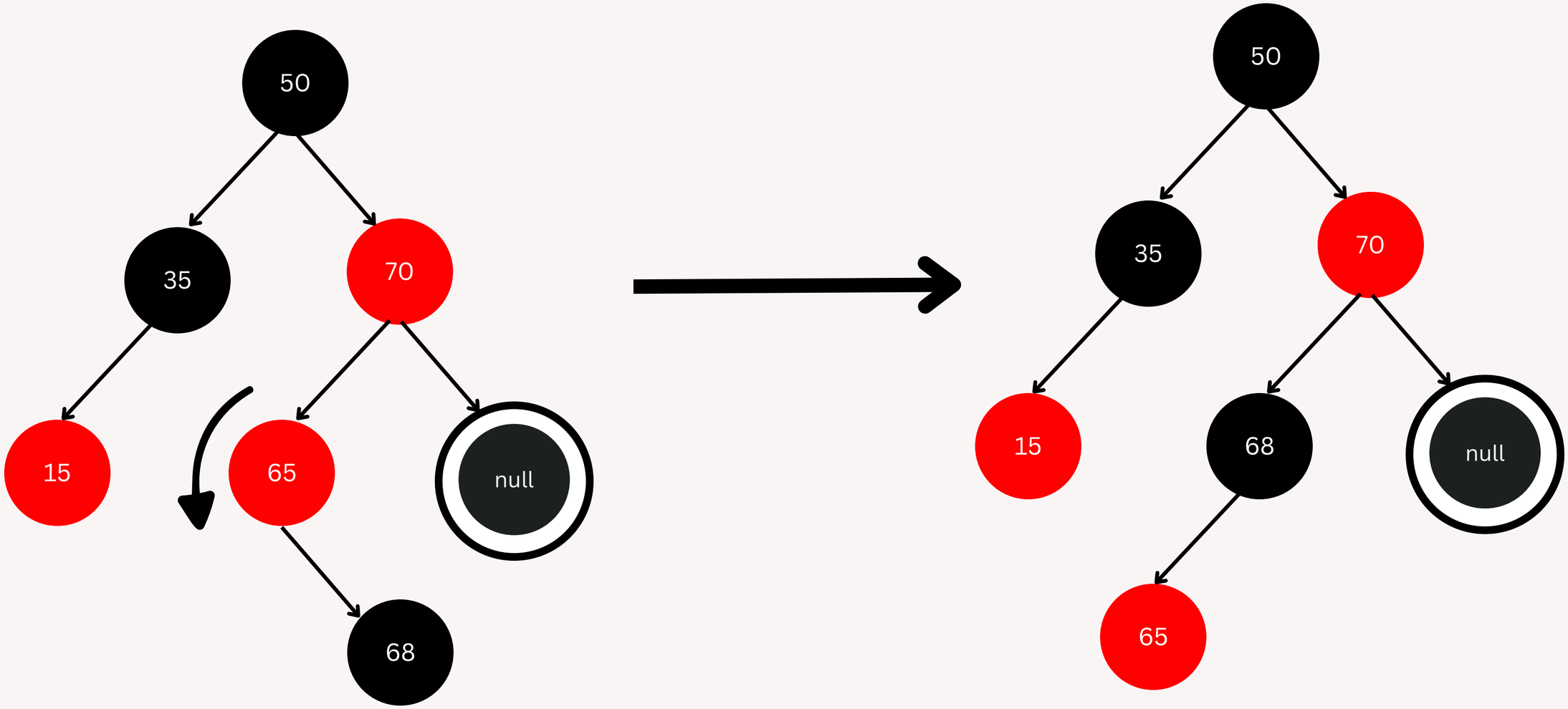
1) Swap colour of DB’s sibling and sibling’s child who is near to DB



sub case: III-III

2)Rotate Sibling in Opposite direction to DB

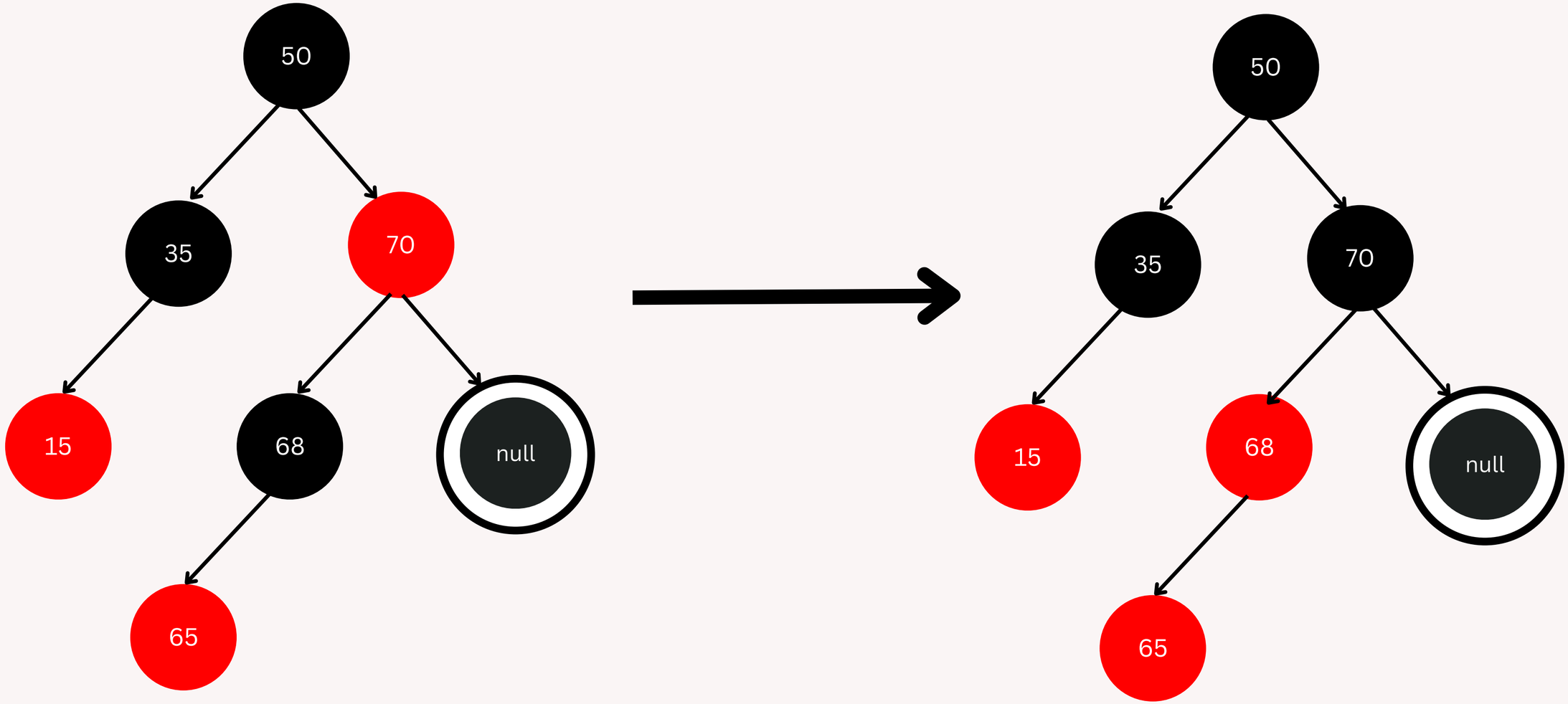
3)Apply Case b



sub case: III-III

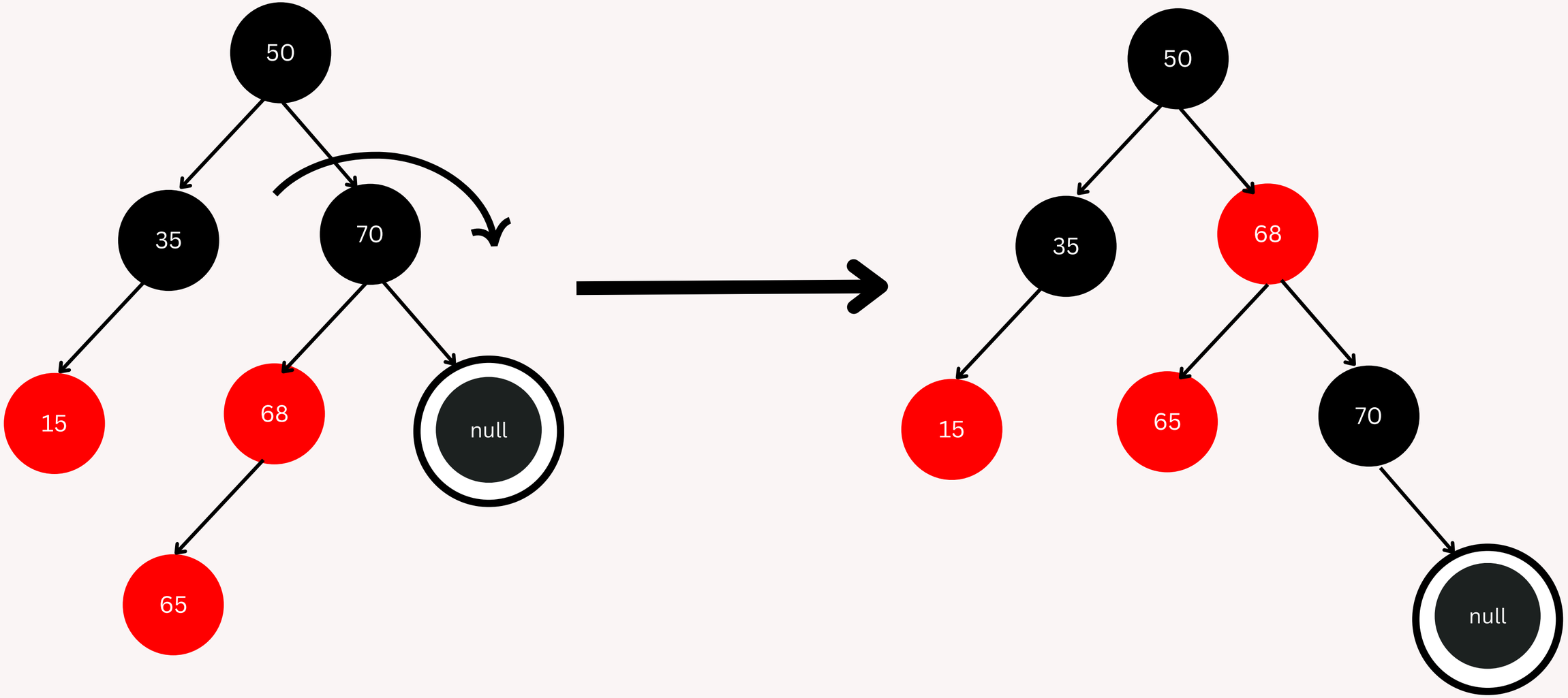
b)When Sibling is Black and sibling’s far child is Red

1) Swap the colour of parent and sibling



sub case: III-III

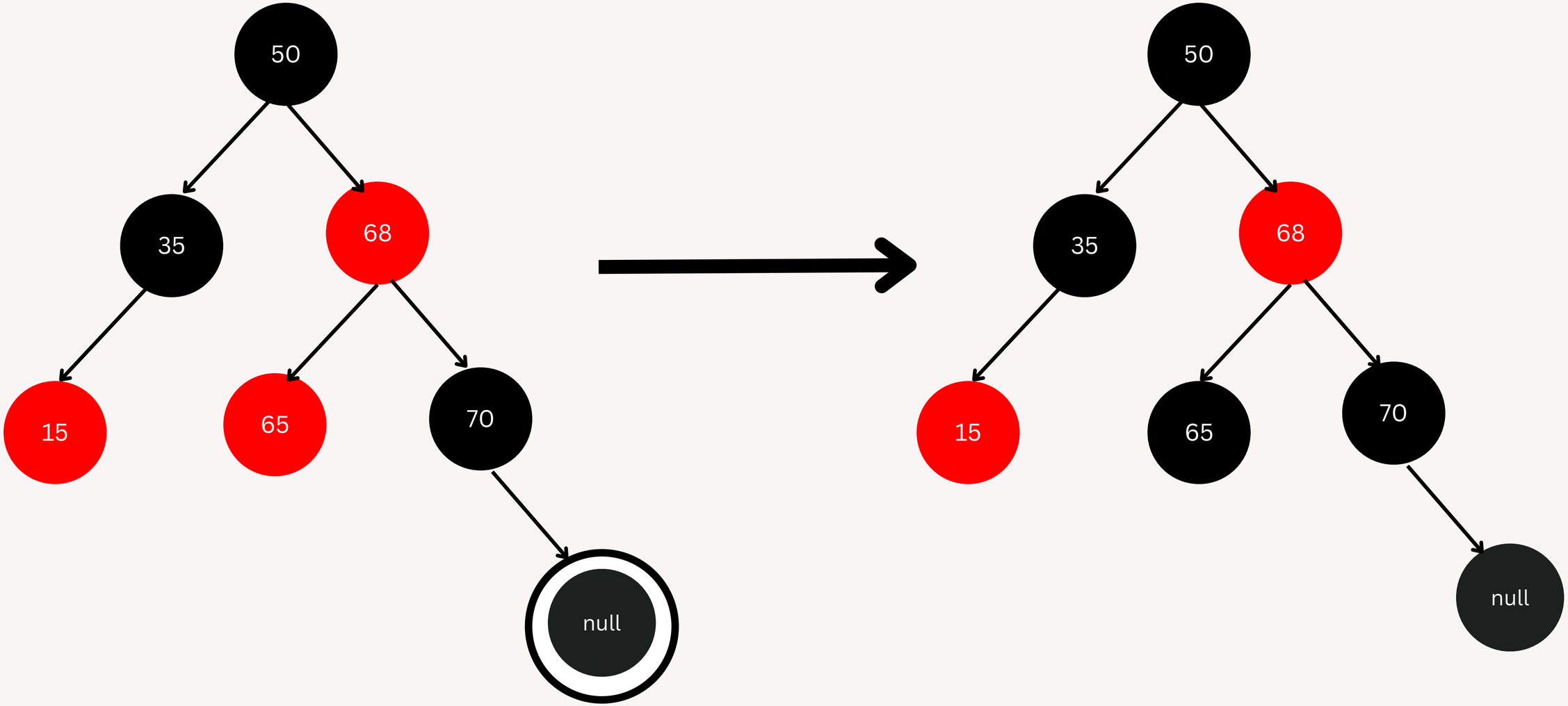
2)Rotate Parent in direction to DB



sub case: III-III

3)Remove DB

4)Change the color of red child to black



Code

```python
class Node:
    def __init__(self, data, color='red'):
        self.data = data
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    def __init__(self):
        self.TNULL = Node(0, color='black')
        self.root = self.TNULL

    def delete_node(self, data):
        node = self._find_node(self.root, data)
        if node == self.TNULL:
            return

        self._delete_node_helper(node)

    def _delete_node_helper(self, node):
        y = node
        y_original_color = y.color
        if node.left == self.TNULL:
            x = node.right
            self._rb_transplant(node, node.right)
        elif node.right == self.TNULL:
            x = node.left
            self._rb_transplant(node, node.left)
        else:
            y = self._minimum(node.right)
            y_original_color = y.color
            x = y.right
            if y.parent == node:
                x.parent = y
            else:
                self._rb_transplant(y, y.right)
                y.right = node.right
                y.right.parent = y
        self._rb_transplant(node, y)
        y.left = node.left
        y.left.parent = y
        y.color = node.color

    if y_original_color == 'black':
        self._fix_delete(x)

def _rb_transplant(self, u, v):
    if u.parent == None:
        self.root = v
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    v.parent = u.parent

def _fix_delete(self, x):
    while x != self.root and x.color == 'black':
        if x == x.parent.left:
            s = x.parent.right
            if s.color == 'red':
                s.color = 'black'
                x.parent.color = 'red'
                self._left_rotate(x.parent)
                s = x.parent.right

            if s.left.color == 'black' and s.right.color == 'black':
                s.color = 'red'
                x = x.parent
            else:
                if s.right.color == 'black':
                    s.left.color = 'black'
                    s.color = 'red'
                    self._right_rotate(s)
                    s = x.parent.right
```

Code

```
                s.color = x.parent.color
                x.parent.color = 'black'
                s.right.color = 'black'
                self._left_rotate(x.parent)
                x = self.root
            else:
                s = x.parent.left
                if s.color == 'red':
                    s.color = 'black'
                    x.parent.color = 'red'
                    self._right_rotate(x.parent)
                    s = x.parent.left

                if s.left.color == 'black' and s.right.color == 'black':
                    s.color = 'red'
                    x = x.parent
                else:
                    if s.left.color == 'black':
                        s.right.color = 'black'
                        s.color = 'red'
                        self._left_rotate(s)
                        s = x.parent.left

                    s.color = x.parent.color
                    x.parent.color = 'black'
                    s.left.color = 'black'
                    self._right_rotate(x.parent)
                    x = self.root
        x.color = 'black'

    def _left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.TNULL:
            y.left.parent = x
```

```
        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def _right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.TNULL:
            y.right.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def _minimum(self, node):
        while node.left != self.TNULL:
            node = node.left
        return node

    def _find_node(self, node, key):
        while node != self.TNULL and key != node.data:
            if key < node.data:
                node = node.left
            else:
                node = node.right
        return node
```

Thank You

Any Questions?