Advanced Data Structures

(CS365)

Red Black Tree

Dr. Sourav Kanti Addya

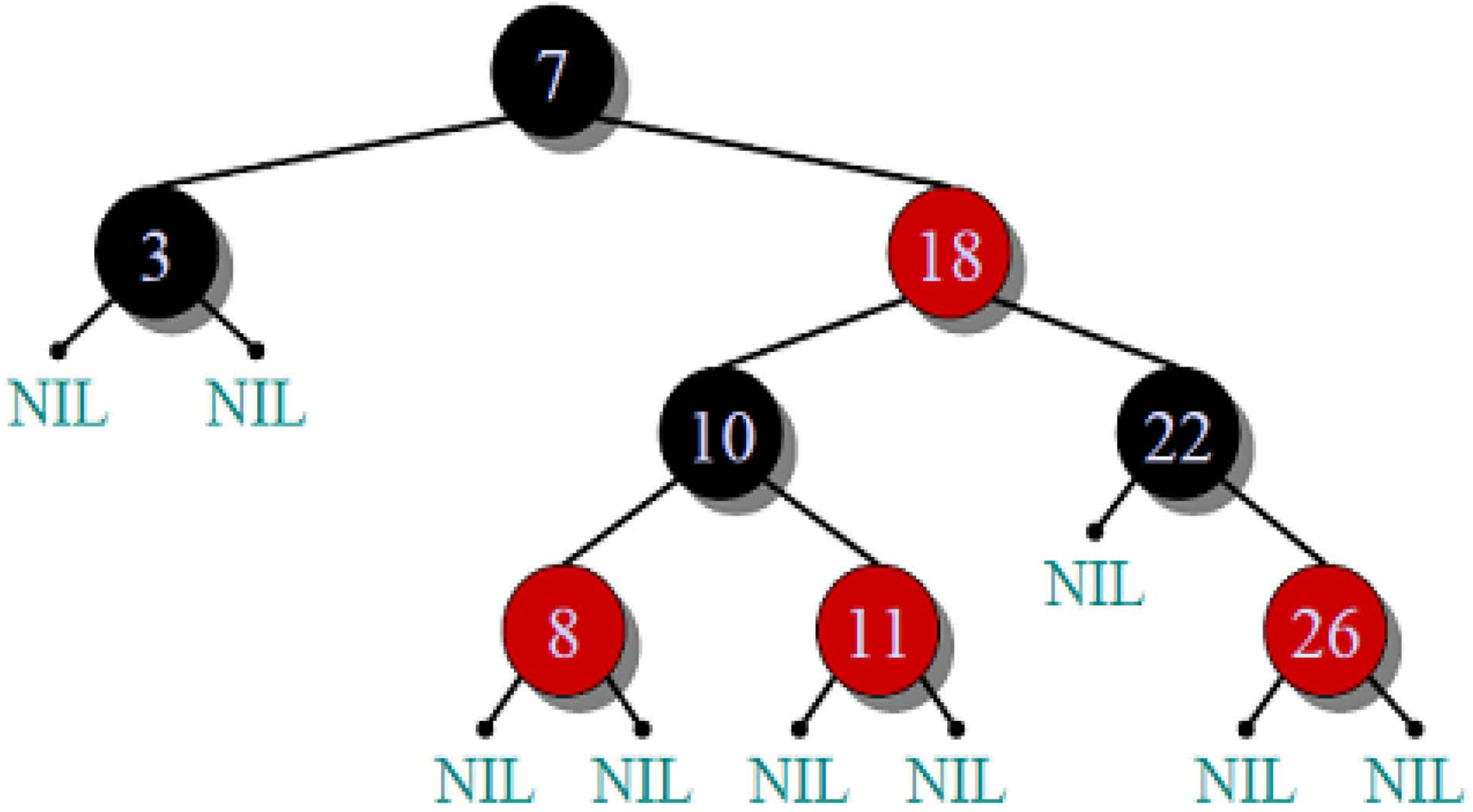
email: souravkaddya@nitk.edu.in

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL

Introduction

- Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.
  1. Every node has a color either red or black.
  2. Root of tree is always black.
  3. If a node is Red then Child can not be Red (Red Rule).
  4. Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes (Path rule).
  5. All NULL nodes are black.



Red-Black Tree (Example)

Why Red-Black Trees?

- Operations (e.g., search, max, min, insert, delete.. Etc.) take *O(h)* time where *h* is the height of the BST.
- The cost of these operations may become *O(n)* for a skewed Binary tree.
- The height of a Red-Black tree is always *O(log n)* where *n* is the number of nodes in the tree. Hence, the operations take *O(log n)* time.

Comparison with AVL Tree

- The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.
- If application involves many frequent insertions and deletions, then Red Black trees should be preferred.
- If insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Invalid RBTs

```
        30                      30                      30
       /  \                    /   \                   /   \
     20    NIL               20     NIL              20     NIL
    /  \                    /  \                    /  \
  10    NIL               10    NIL               10    NIL
 Violates                Violates                Violates
 Property 4.            Property 4              Property 3
```

Valid RBTs

```
          20                                        20
         /   \                                     /   \
      10       30                              10       30
     /  \     /  \                            /  \     /  \
  NIL   NIL NIL  NIL                       NIL   NIL NIL  NIL
```

Insertion in RBTs

- In Red-Black tree, we use two tools to do balancing.
    - **Recoloring.**
    - **Rotation.**

Steps

1. Let x be the newly inserted node. Perform standard BST insertion and make the color of newly inserted nodes as RED.
2. CASE-1: If x is root, change color of x as BLACK and Return.
3. CASE-2: If color of x's parent is BLACK then Return.
4. CASE-3: Do following if color of x's parent is Red

   Case 3a: If x's uncle is **RED** (Grand parent must have been black from property 4)

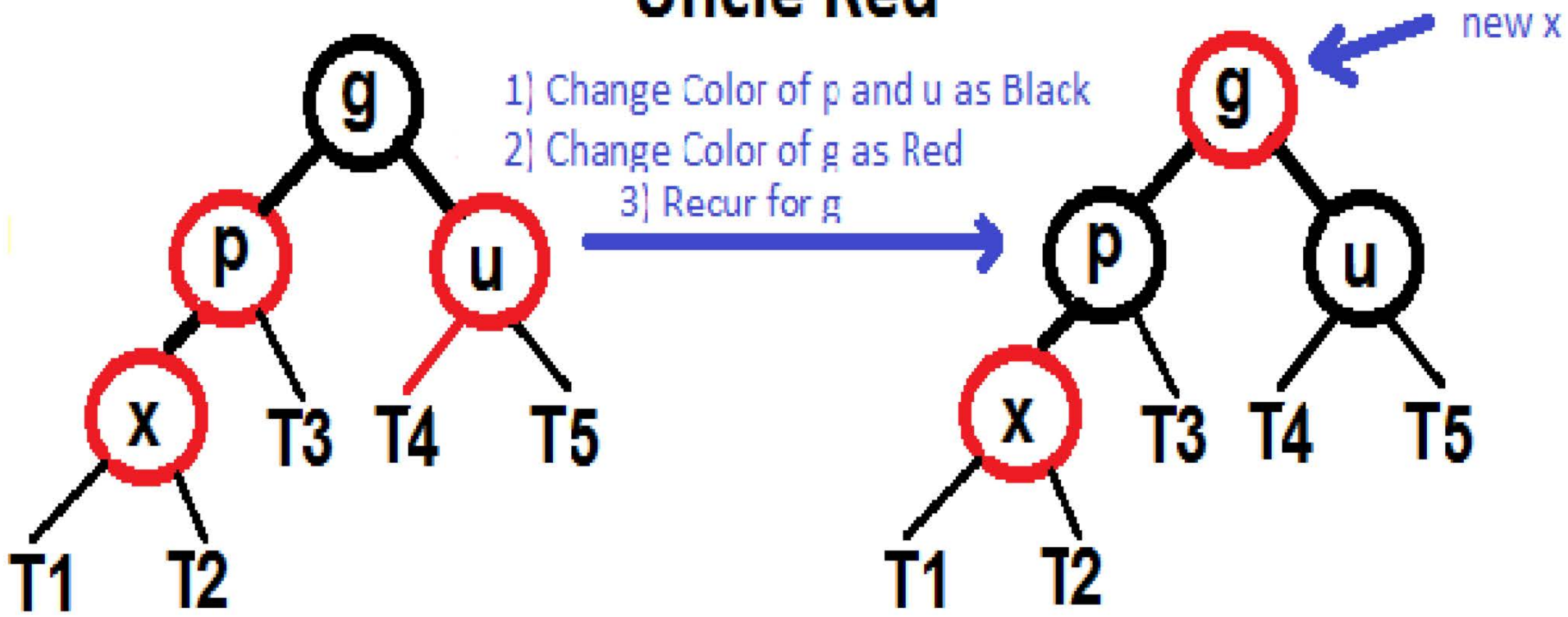
   1. Change color of parent and uncle as BLACK.
   2. Color the grand parent as RED.
   3. Change x = x's grandparent, repeat steps 2 to 4 for new x.

Steps (cont.)

Case 3b: If x's uncle is BLACK, then there can be four configurations for x, x's parent (p) and x's grandparent (g) - This is similar to AVL Tree.

1. Left Left Case (p is left child of g and x is left child of p).
2. Left Right Case (p is left child of g and x is right child of p).
3. Right Right Case.
4. Right Left Case.

Uncle Red



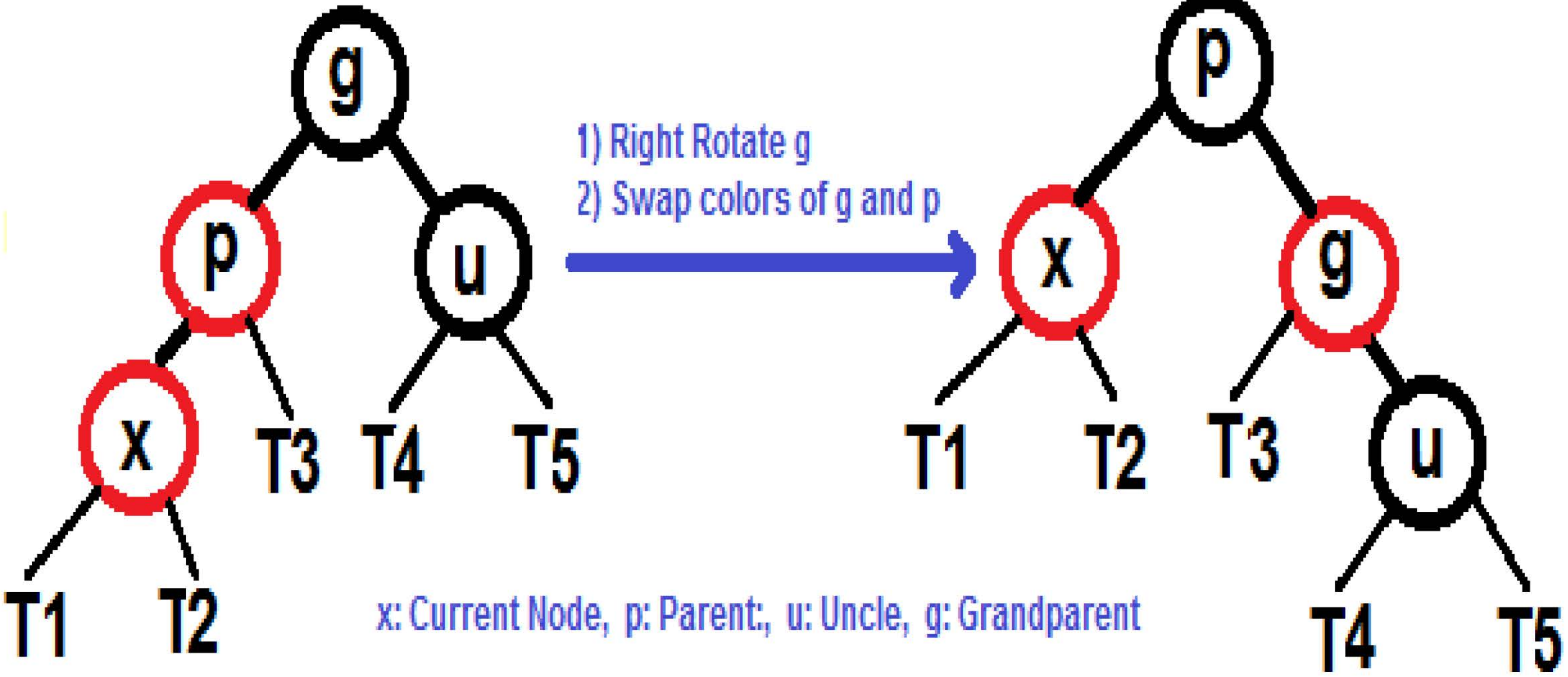
g
p
u
x
T1
T2
T3
T4
T5
1) Change Color of p and u as Black
2) Change Color of g as Red
3) Recur for g
new x

x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

All four cases when Uncle is BLACK

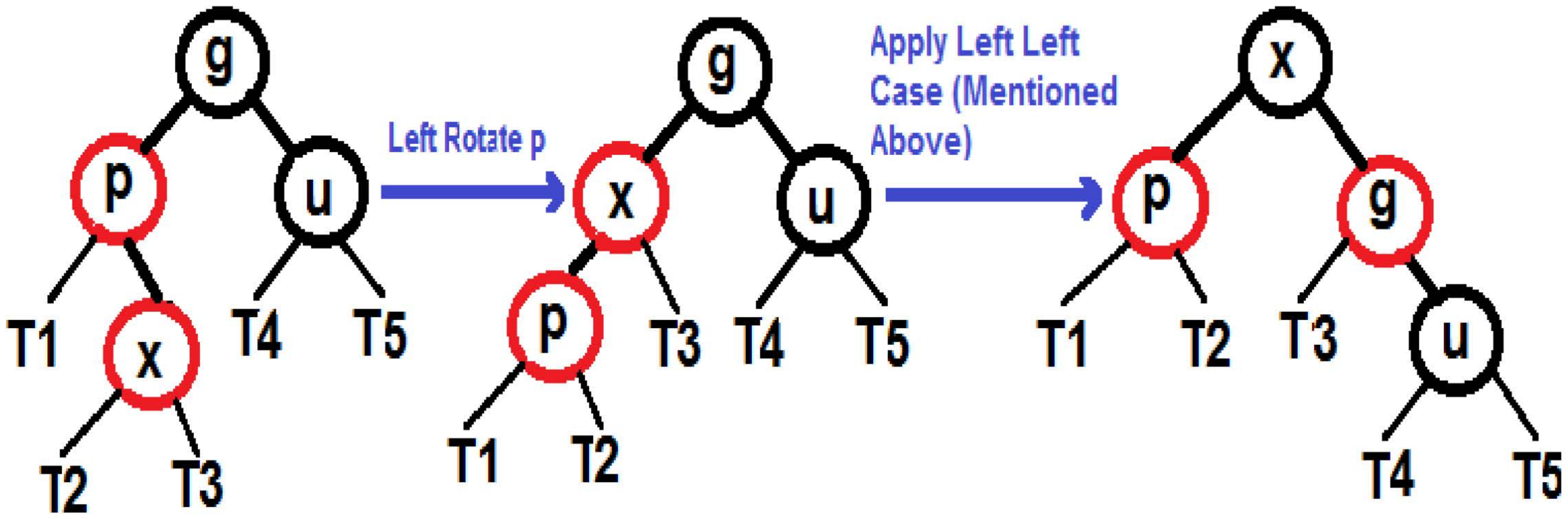
Uncle Black and Left Left Case



x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

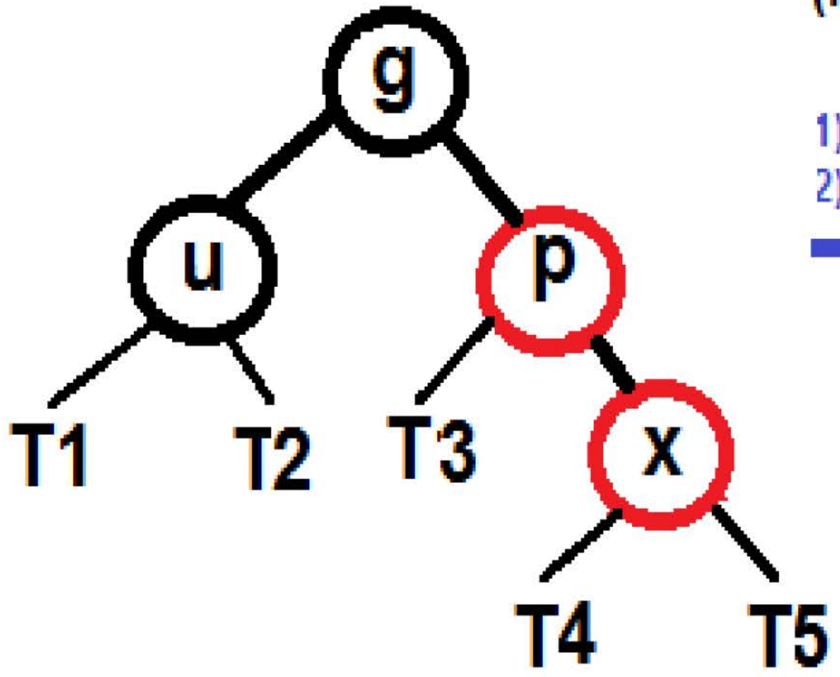
Uncle Black and Left Right Case



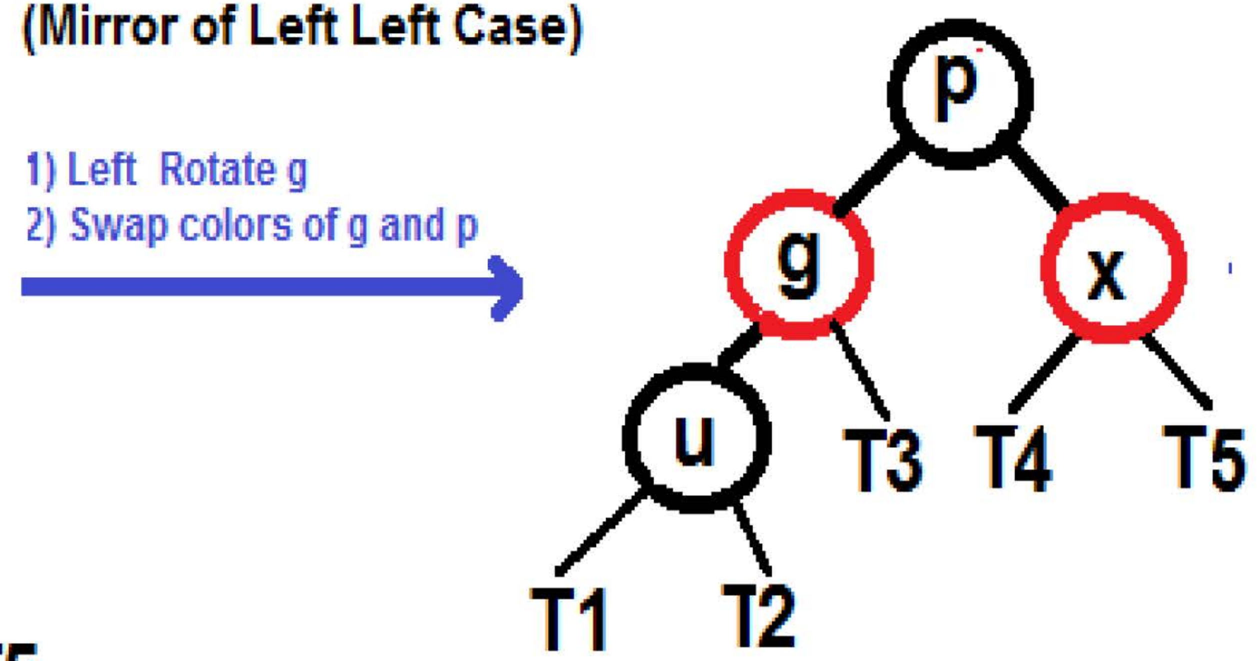
x: Current Node, p: Parent:, u: Uncle, g: Gr

T1, T2, T3, T4 and T5 are subtrees

Uncle Black and Right Right Case
(Mirror of Left Left Case)



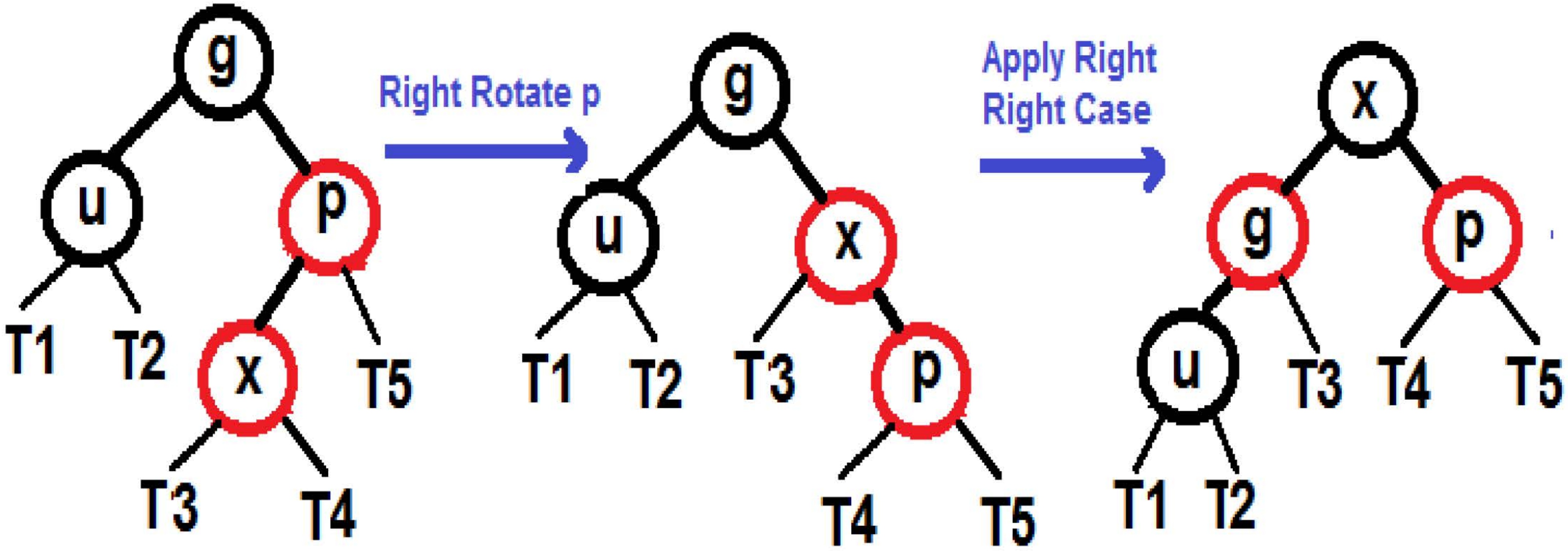
1) Left Rotate g
2) Swap colors of g and p



x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

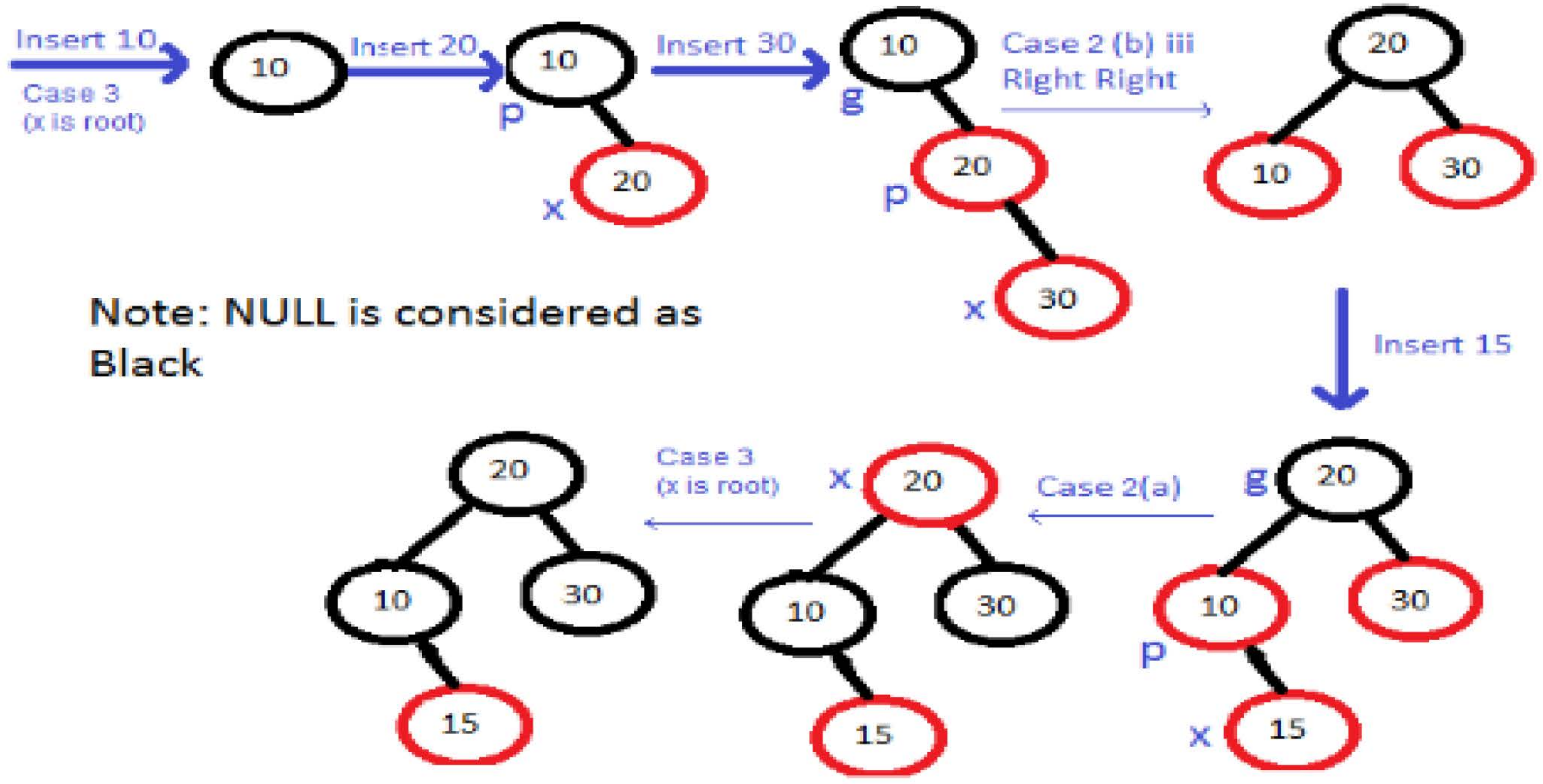
Uncle Black and Right Left Case
(Mirror of Left Right Case)



x: Current Node, p: Parent:, u: Uncle, g: Grandparent

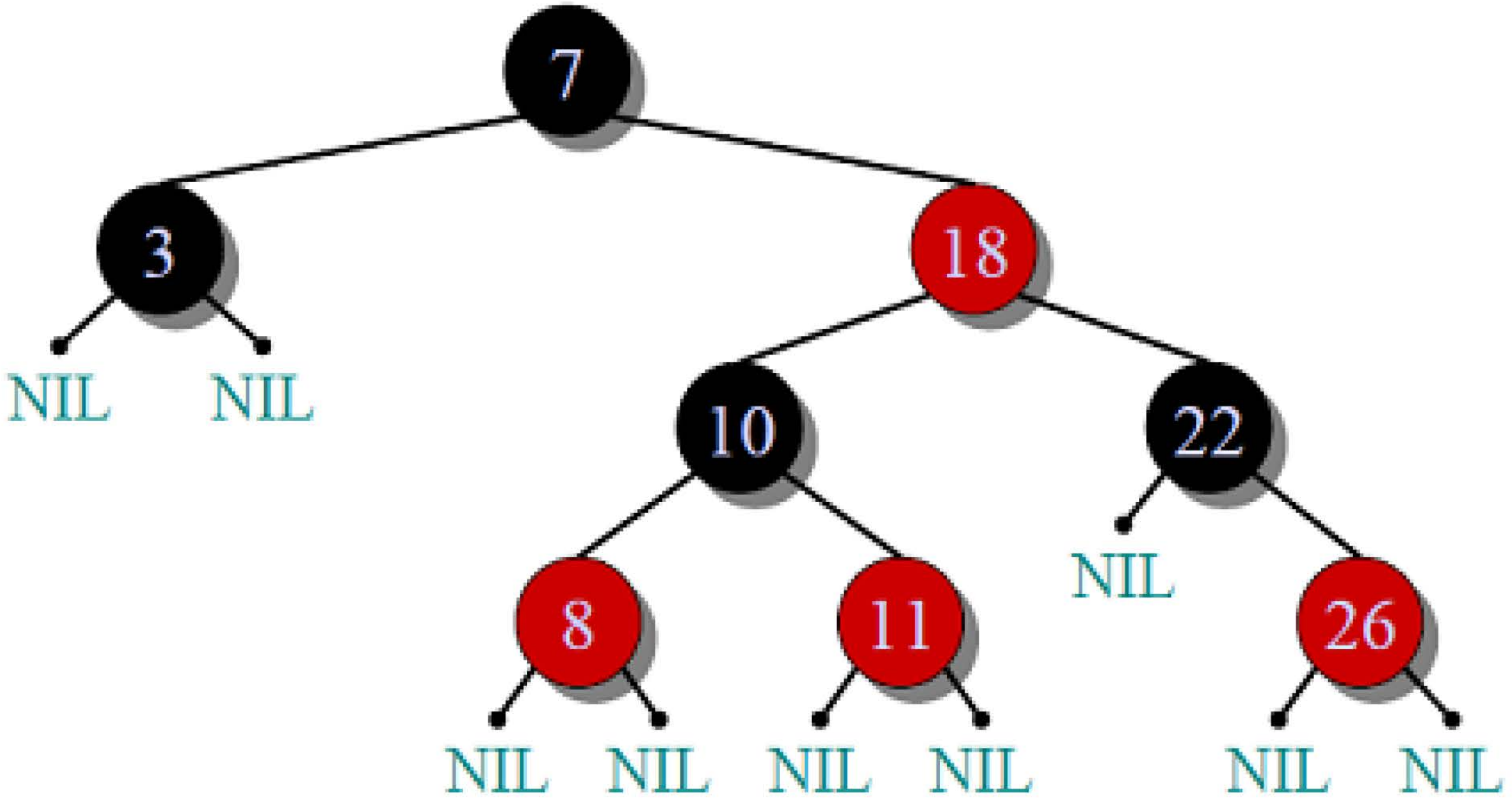
T1, T2, T3, T4 and T5 are subtrees

Insert 10, 20, 30 and 15 in an empty tree



Note: NULL is considered as Black

Insert 2, 16 and 13 in the given tree.



Insert 10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70.

Deletion in RBTs

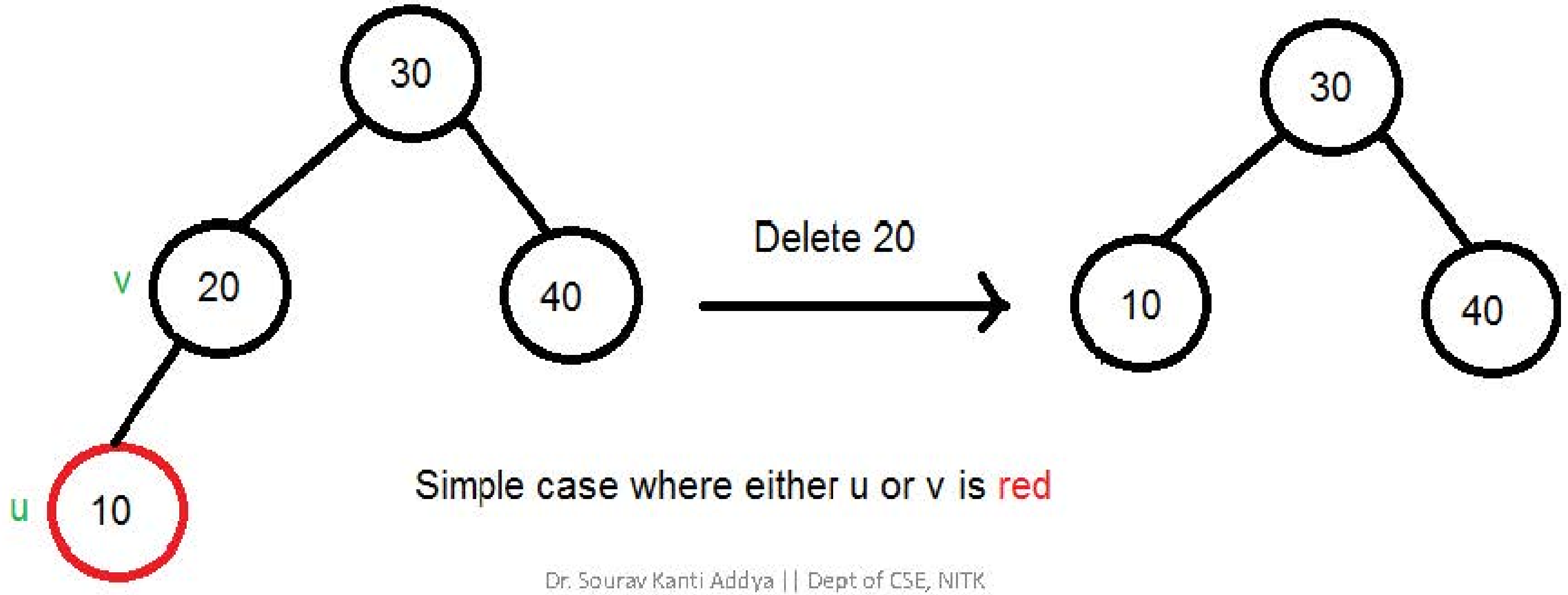
- In insert operation, we used to check the color of uncle to decide the appropriate case. In delete operation, ***we check color of sibling*** to decide the appropriate case.
- The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of **black height** in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.
- To understand deletion, notion of **double black** is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***.
- The main task now becomes to convert this double black to single black.

Deletion Steps

- Following are detailed steps for deletion.

  1. Perform **standard BST delete.** Let **v** be the node to be deleted and **u** be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

  2. **Simple Case: If either u or v is red,** we mark the replaced child as black (No change in black height.)

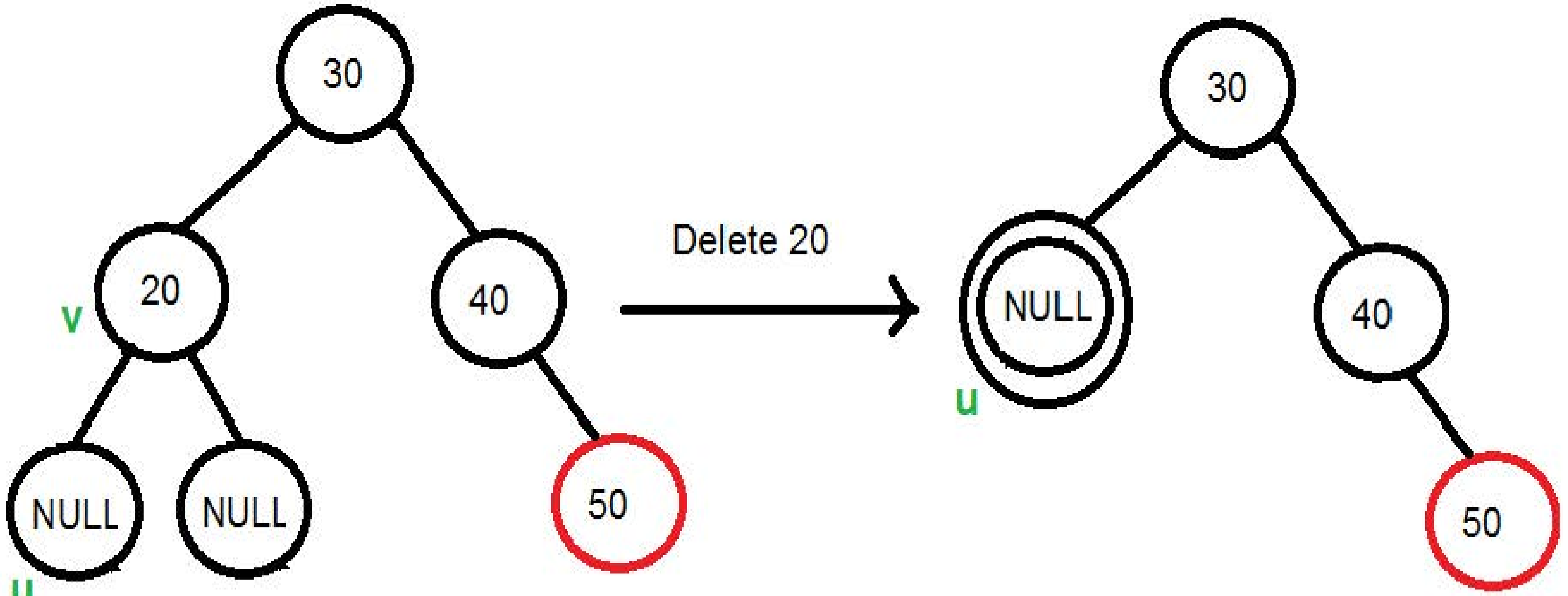


Simple case where either u or v is red

Deletion Steps (cont.)

3. **If Both u and v are Black.**

   3.1 Color u as double black



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.

Note that deletion is not done yet, this double black must become single black

Deletion Steps (cont.)

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

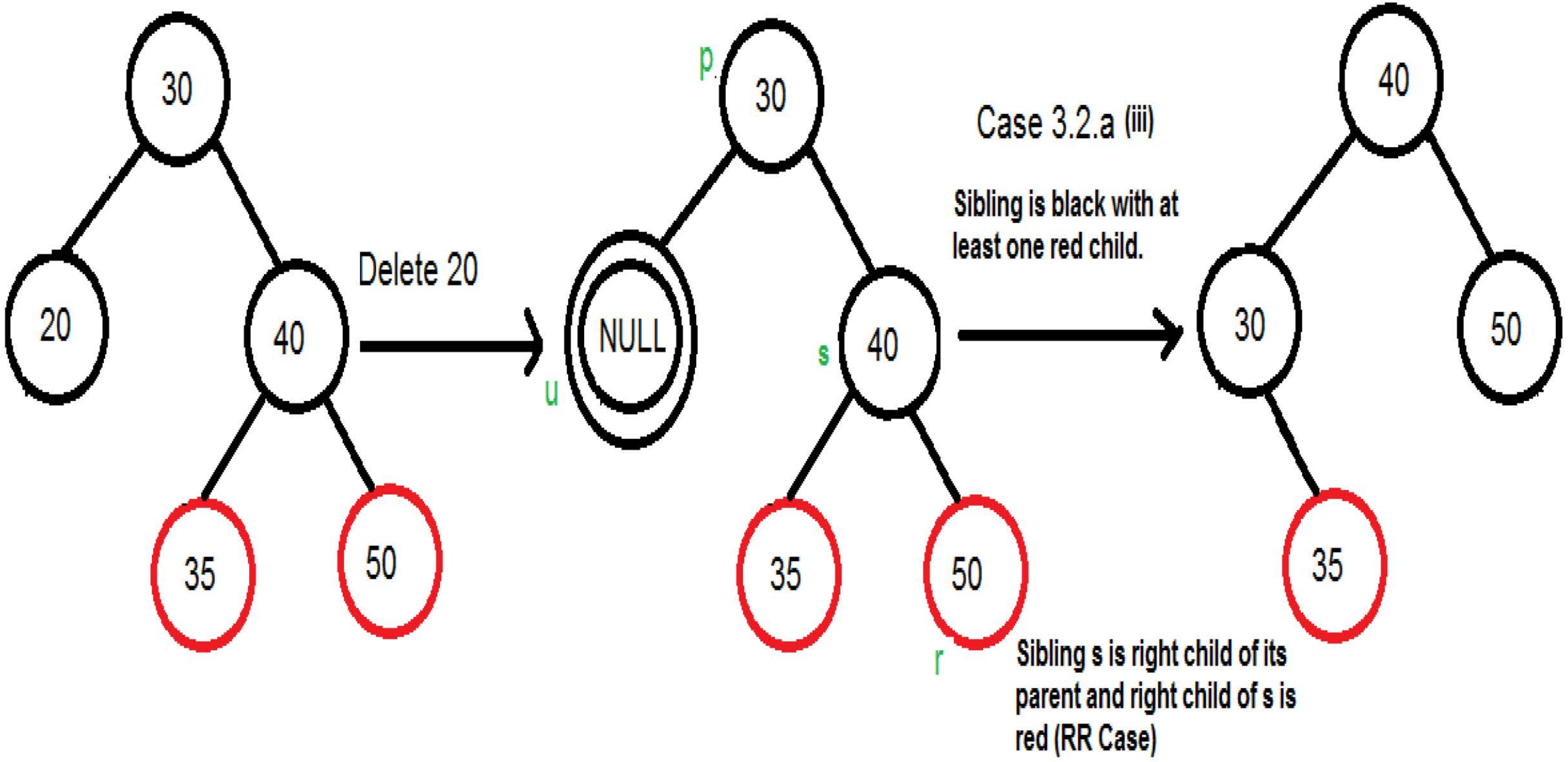
(a) **If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of **s** and **r**.

(i.) Left Left case.

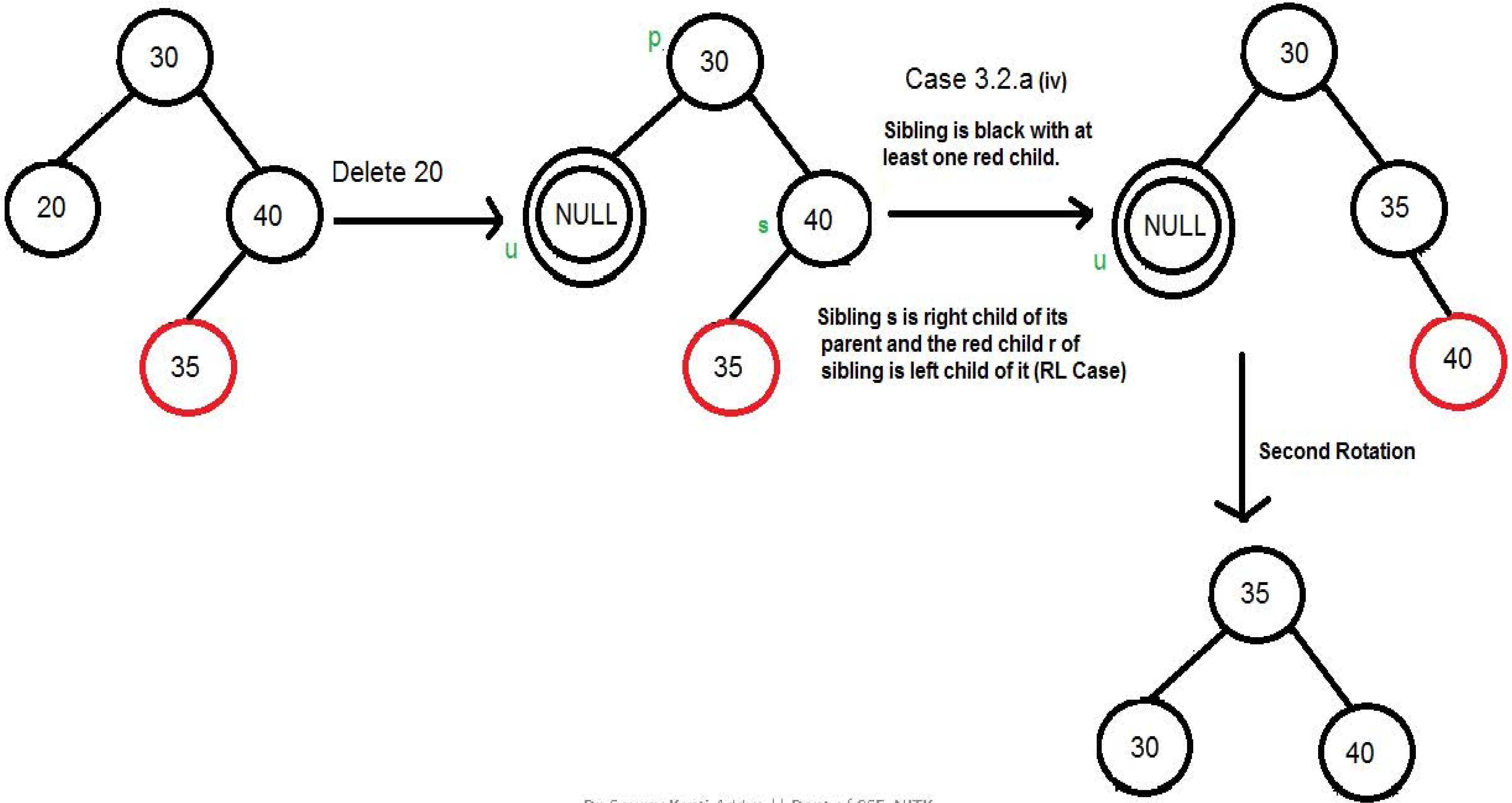
(ii.) Left Right case.

(iii.) Right Right case.

(iv.) Right Left case.



30
20
40
35
50
Delete 20
p
30
u
NULL
s
40
35
50
r
Case 3.2.a (iii)
Sibling is black with at least one red child.
40
30
50
35
Sibling s is right child of its parent and right child of s is red (RR Case)



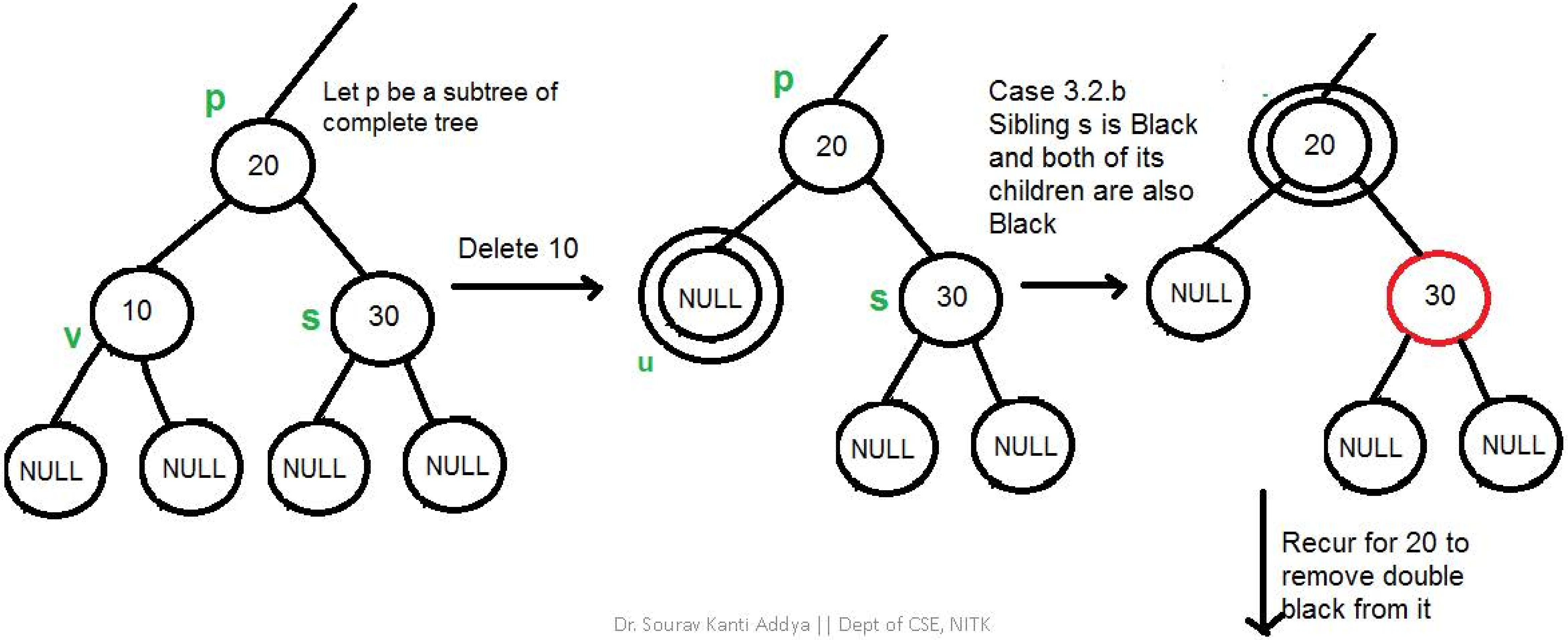
Case 3.2.a (iv)

Sibling is black with at least one red child.

Sibling s is right child of its parent and the red child r of sibling is left child of it (RL Case)

Deletion Steps (cont.)

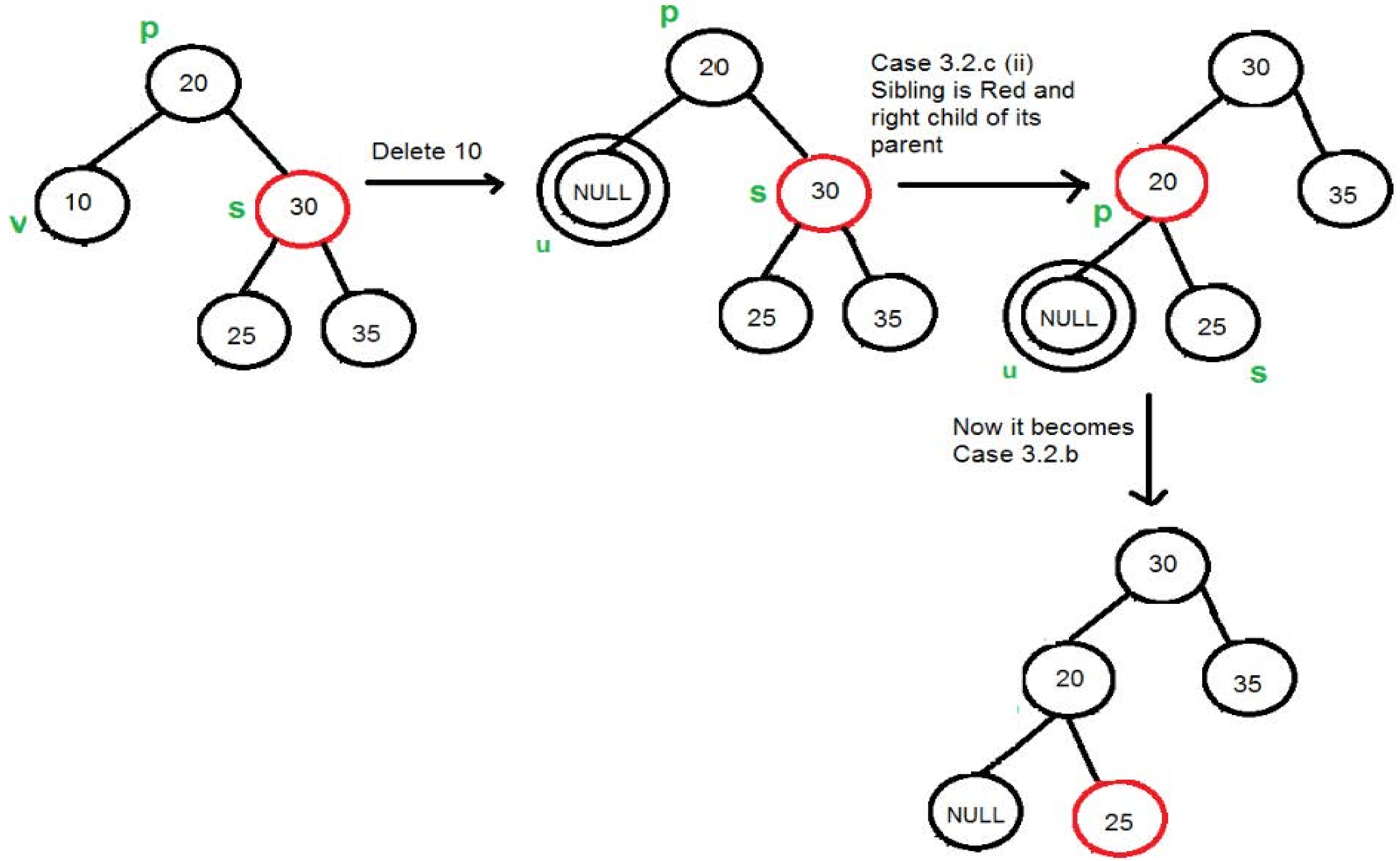
3.2 (b) **If sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black.


p
Let p be a subtree of complete tree
20
v
10
s
30
NULL
NULL
NULL
NULL
Delete 10
p
20
NULL
u
s
30
NULL
NULL
Case 3.2.b Sibling s is Black and both of its children are also Black
20
NULL
30
NULL
NULL
Recur for 20 to remove double black from it

3.2 (c) If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. This case can be divided in two subcases.

(i.) Left Case

(ii.) Right Case.


p
20
v
10
s
30
25
35
Delete 10
p
20
NULL
u
s
30
25
35
Case 3.2.c (ii)
Sibling is Red and
right child of its
parent
30
20
p
35
NULL
u
25
s
Now it becomes
Case 3.2.b
30
20
35
NULL
25