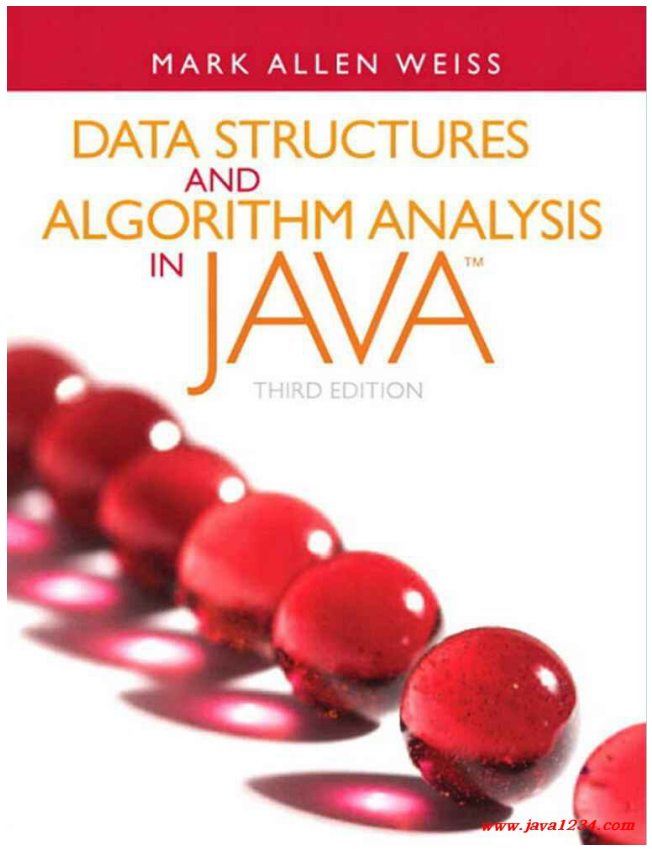
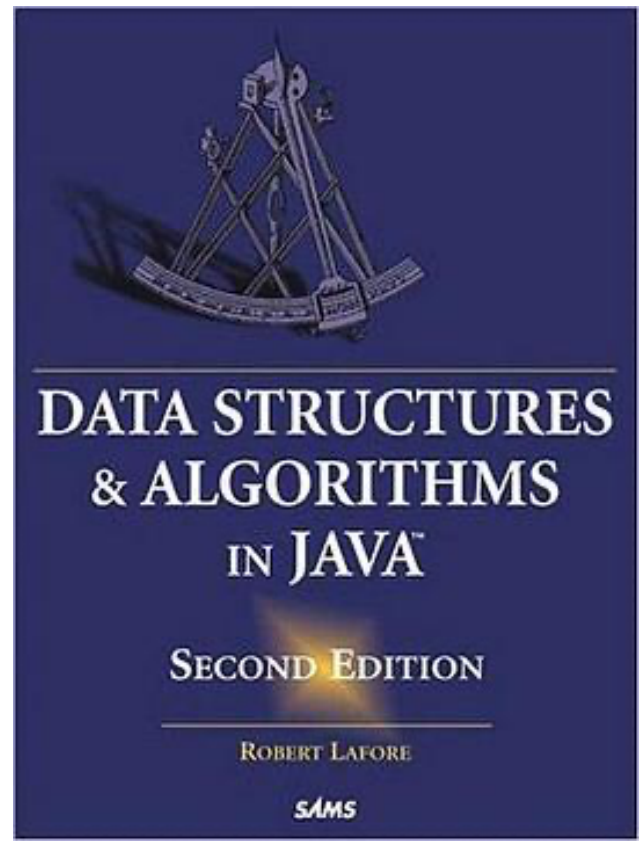
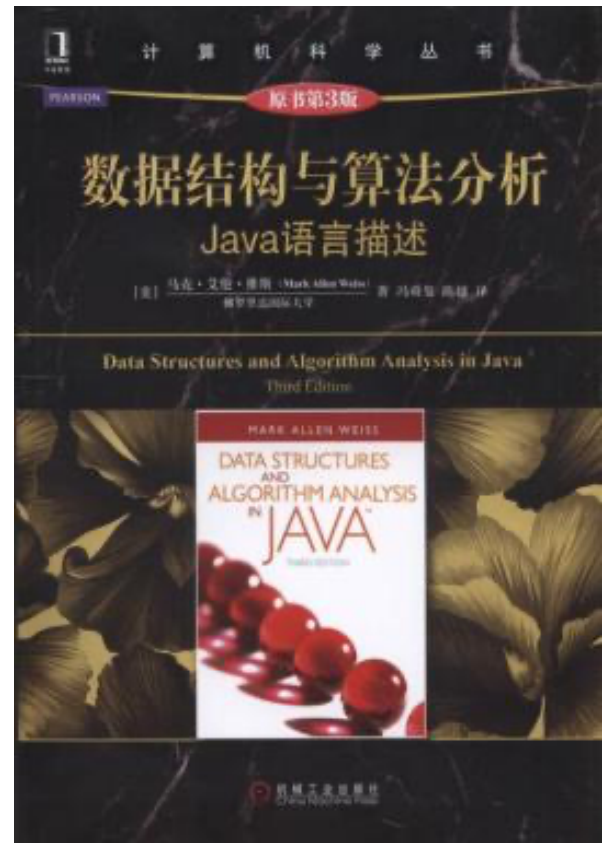
Topic 14 – Red Black Tree







- **Red Black Tree**
- Search
- Insertion and Deletion
    - Recoloring and Rotation
    - Insertion
    - Deletion

Why Red-Black Trees?

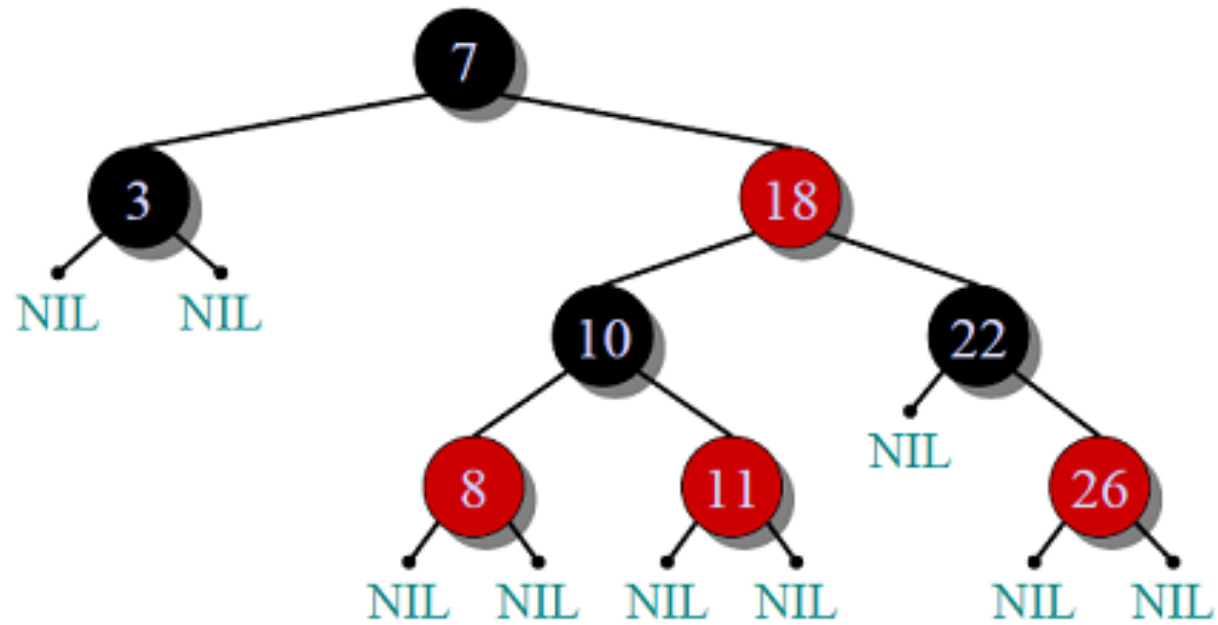
- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.
- The cost of these operations may become O(n) for a skewed Binary tree.
- If we make sure that height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations.
- The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Algorithm	Time Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

Red Black Tree

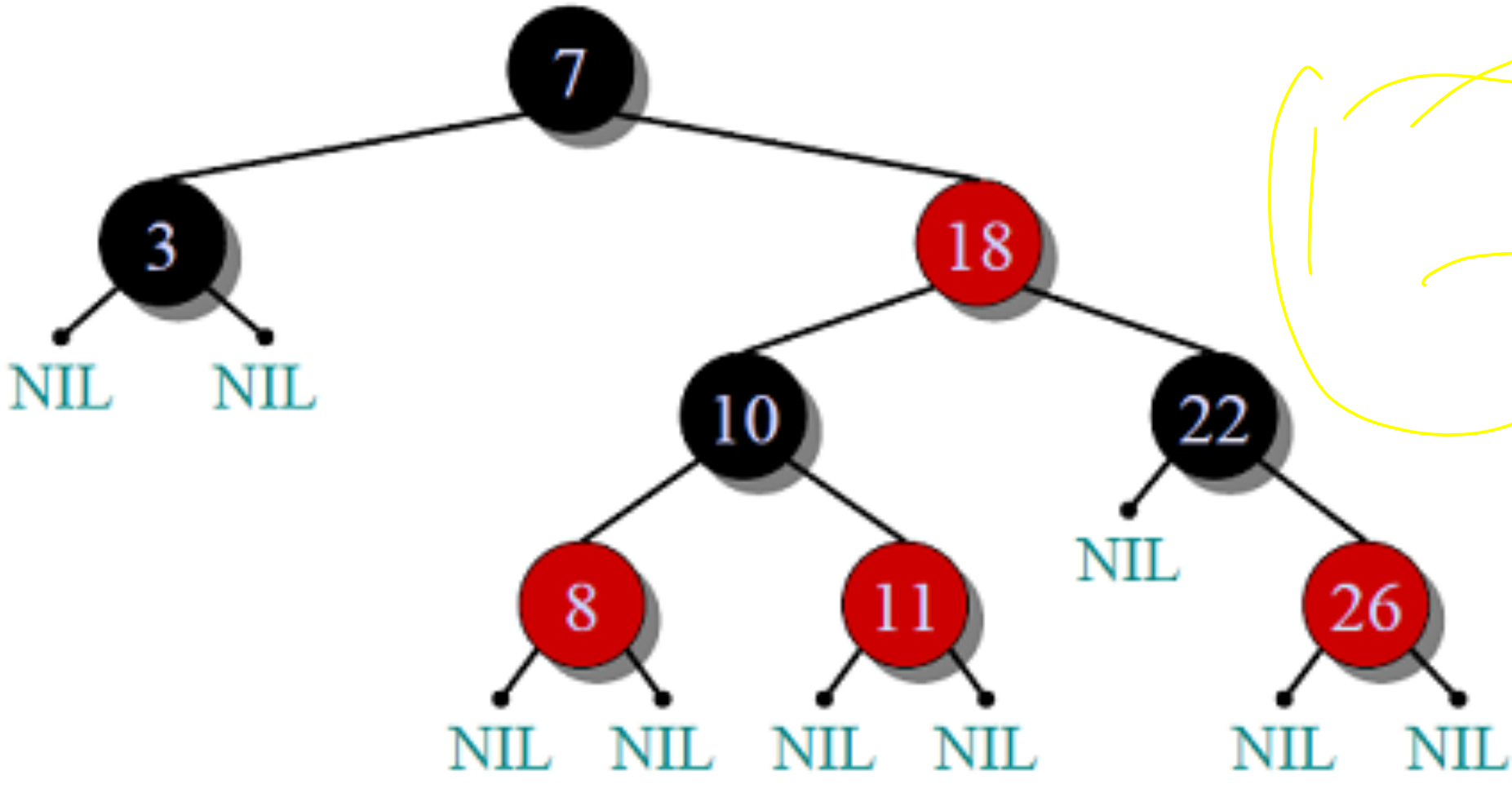
$h \leq 2 \log_2(n+1)$

- A red-black tree is a kind of **self-balancing binary search tree** where each node has an extra bit, and that bit is often interpreted as the colour (red or black).
- These colours are used to ensure that the tree remains balanced during insertions and deletions.
- Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.
- This tree was invented in 1972 by Rudolf Bayer.



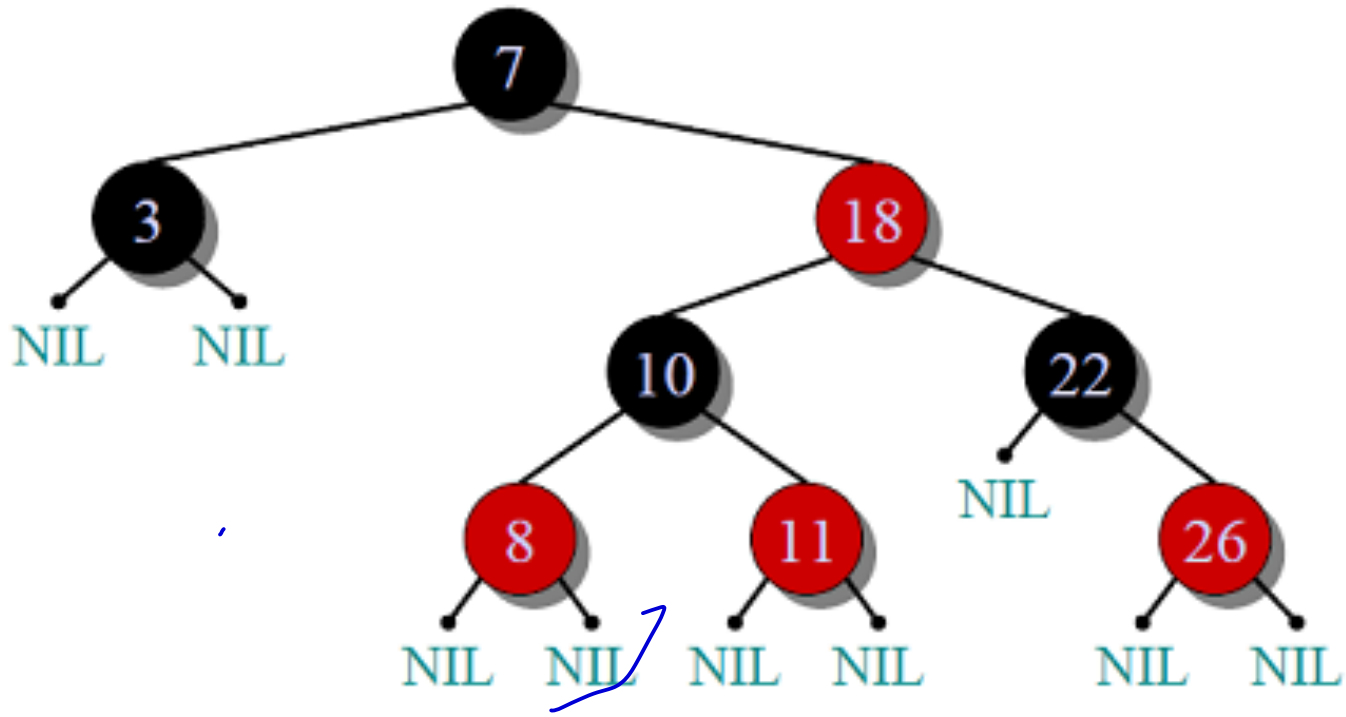
Red Black Tree

- It must be noted that as each node requires only 1 bit of space to store the colour information, these types of trees show identical memory footprint to the classic (uncoloured) binary search tree.



Red Black Tree

- **Rules That Every Red-Black Tree Follows:**
  1. Every node has a colour either red or black.
  2. The root of tree is always black.
  3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
  4. Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.



Comparison with AVL Tree

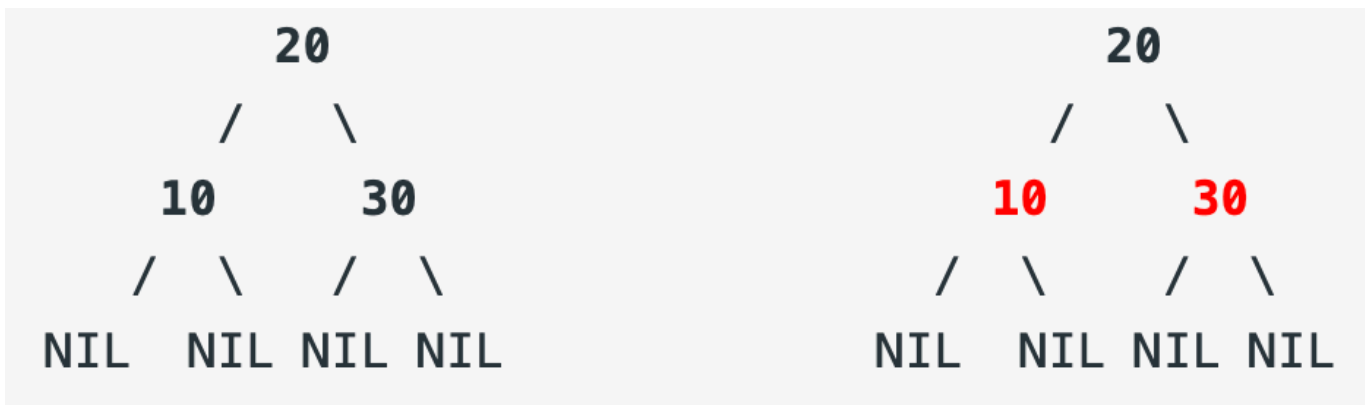
- The AVL trees are **more balanced** compared to Red-Black Trees, but they may cause **more rotations** during insertion and deletion.
- So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred.
- And if the insertions and deletions are less frequent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

AVL上的旋转动作太多了，不适合一些项目。

How does a Red-Black Tree ensure balance?

- A simple example to understand balancing is, a chain of 3 nodes is not possible in the Red-Black tree.
- We can try any combination of colours and see all of them violate Red-Black tree property.
- A chain of 3 nodes is nodes is not possible in Red-Black Trees.
- Following are **NOT** Red-Black Tree

- Following are different possible Red-Black Trees with above 3 keys



Interesting points about Red-Black Tree:

1. ***Black Height of a Red-Black Tree :***
   - Black height is number of black nodes on a path from a node to a leaf. Leaf nodes are also counted black nodes. From above properties 3 and 4, we can derive, **a node of height h has black-height >= h/2.**
2. Height of a red-black tree with n nodes is h<= 2 $\log_2(n + 1)$.
3. **All leaves (NIL) are black.**
4. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

Black Height of a Red-Black Tree

- Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes.
- From the above properties 3 and 4, we can derive, **a Red-Black Tree of height h has black-height >= h/2**.

Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.

Height of a red-black tree with n nodes is h<= 2 $\log_2(n + 1)$

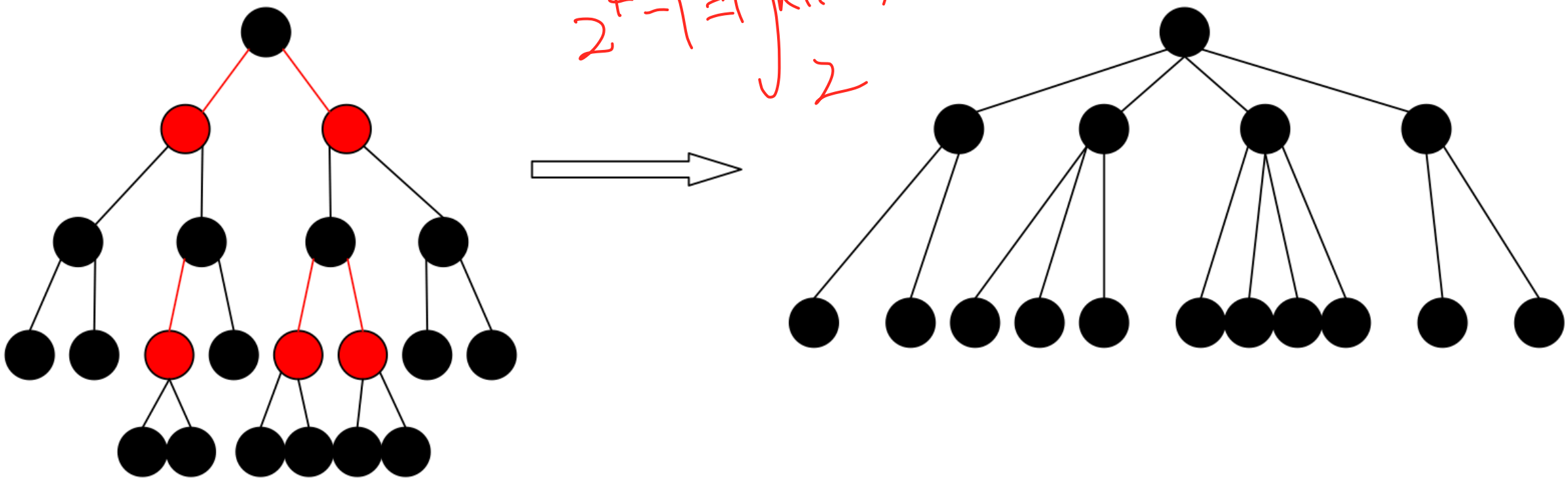
- This can be proved using following facts:
  1. For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then n >= 2k – 1 (Ex. If k is 3, then n is at least 7). This expression can also be written as k <= 2 $\log_2(n+1)$
  2. From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\log_2(n+1)$ black nodes.
  3. From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least ⌊ n/2 ⌋ where n is the total number of nodes.

Height of a red-black tree with n nodes is h<= 2 $\log_2(n + 1)$

- From above 2 points, we can conclude the fact that Red Black Tree with **n** nodes has height <= 2 $\log_2(n+1)$
- In this post, we introduced Red-Black trees and discussed how balance is ensured.
- The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on the Red-Black tree.

O(log n)

- Compact



- The height of compacted tree is O(log n)
- Since no two red nodes are connected, the height of the original tree is at most 2 log n = O(log n)

- Red Black Tree
- **Search**
- Insert and Delete
  - Recoloring and Rotation
  - Insertion
  - Deletion

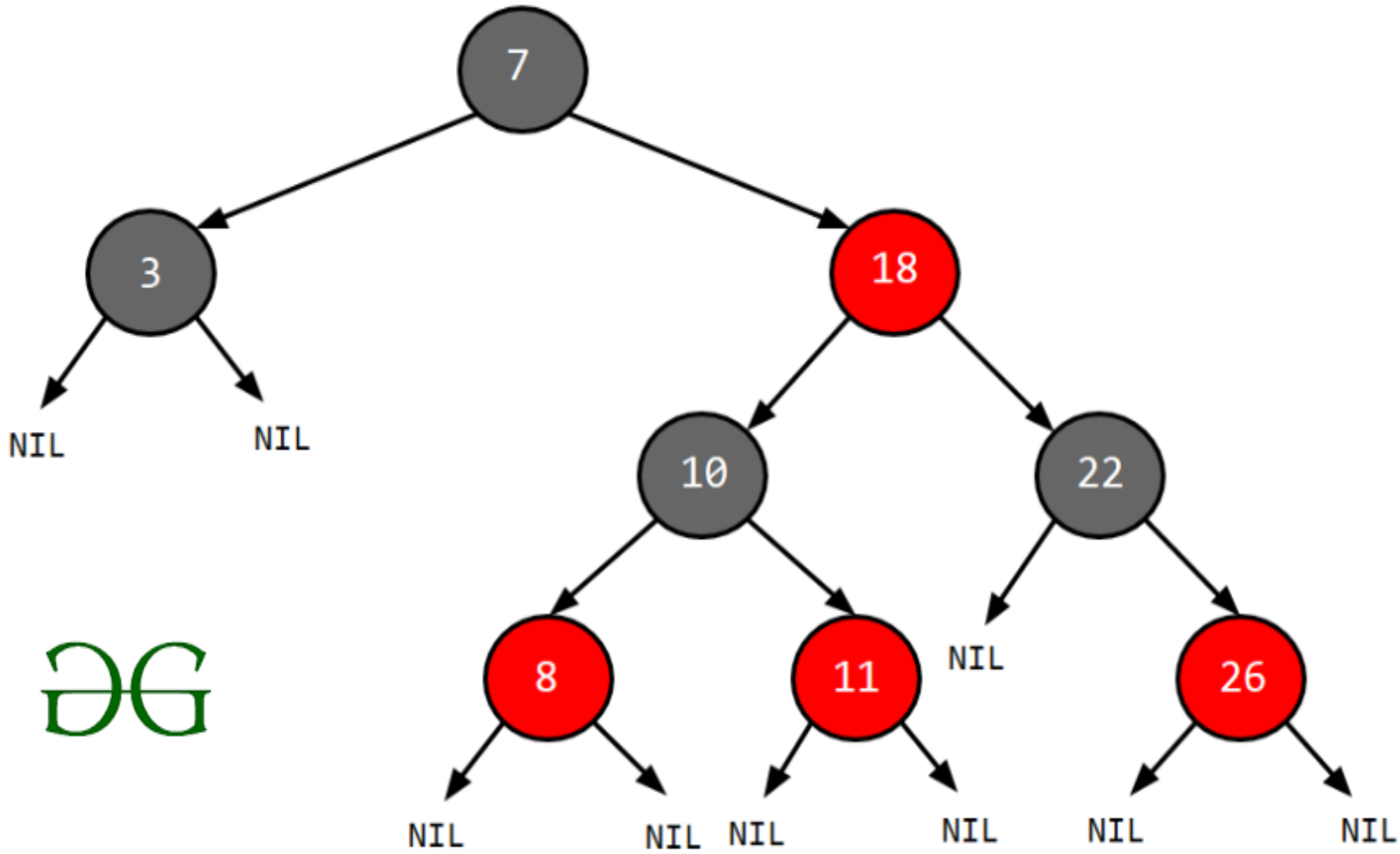
Search Operation in Red-black Tree

- As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

```
//Algorithm of Search
searchElement (tree, val)
Step 1:
If tree -> data = val OR tree = NULL
      Return tree
Else
      If val data
            Return searchElement (tree -> left, val)
      Else
            Return searchElement (tree -> right, val)
      [ End of if ]
[ End of if ]
Step 2: END
```

Example

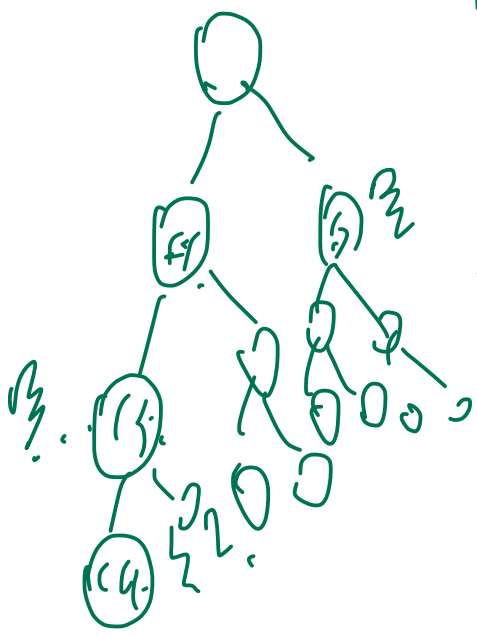
- **Searching 11 in the following red-black tree.**



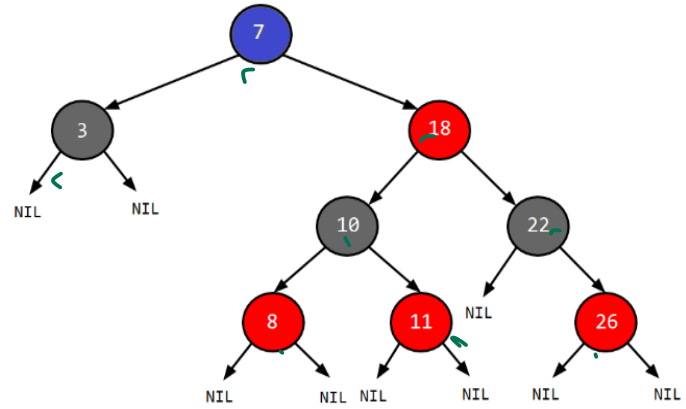
Example

- **Solution:**
  1. Start from the root.
  2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
  3. If the element to search is found anywhere, return true, else return false.

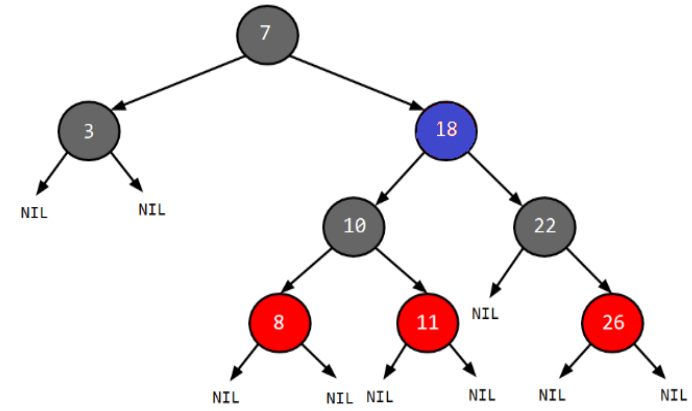
- Just follow the blue bubble.



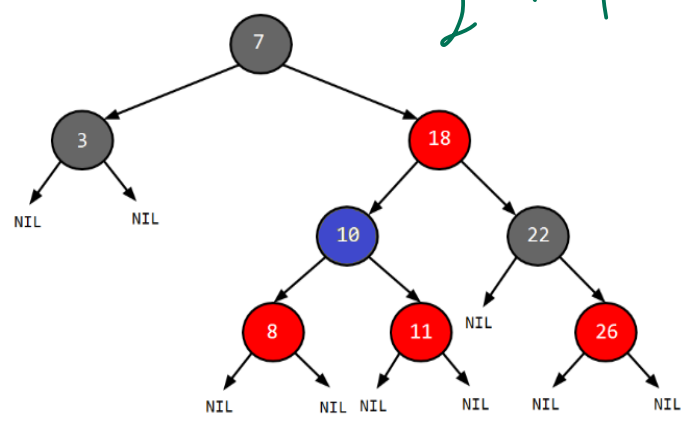
Step-1



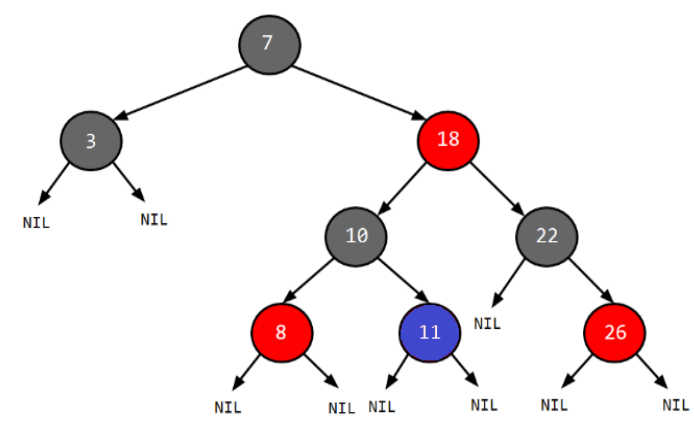
Step-2



Step-3



Step-4



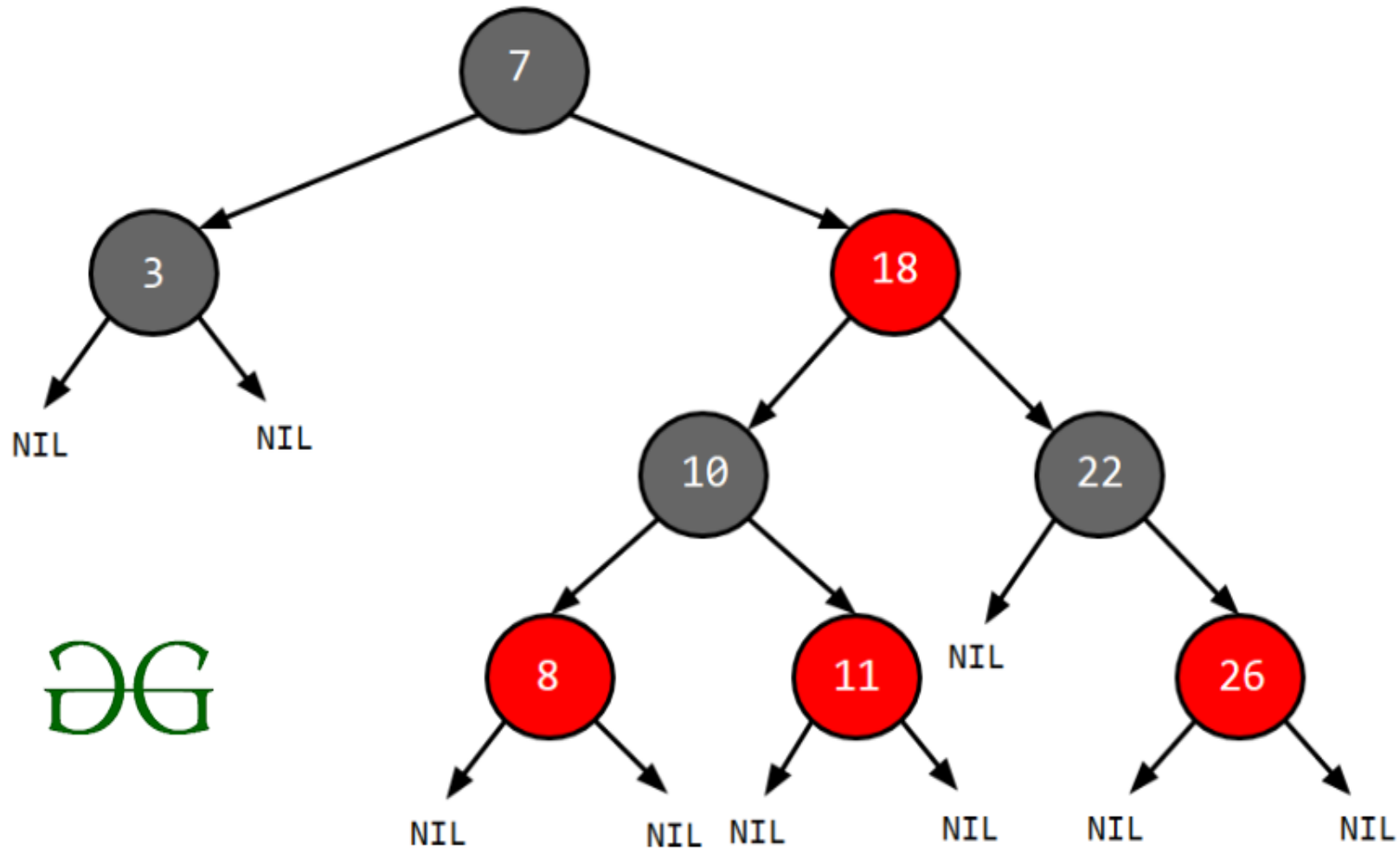
- In this post, we introduced Red-Black trees and discussed how balance is ensured.
- The hard part is to maintain balance when keys are added and removed.
- We have also seen how to search an element from the red-black tree.
- We will soon be discussing insertion and deletion operations in coming posts on the Red-Black tree.

$h \leq 2 \log(n+1)$

$\frac{1}{2} h - 1 \leq \log_2(n+1)$

Exercise:

- **1)** Is it possible to have all black nodes in a Red-Black tree?
- **2)** Draw a Red-Black Tree that is not an AVL tree structure-wise?



- Red Black Tree
- Search
- **Insertion and Deletion**
  - **Recoloring and Rotation**
  - Insertion
  - Deletion

$2^k - 1$

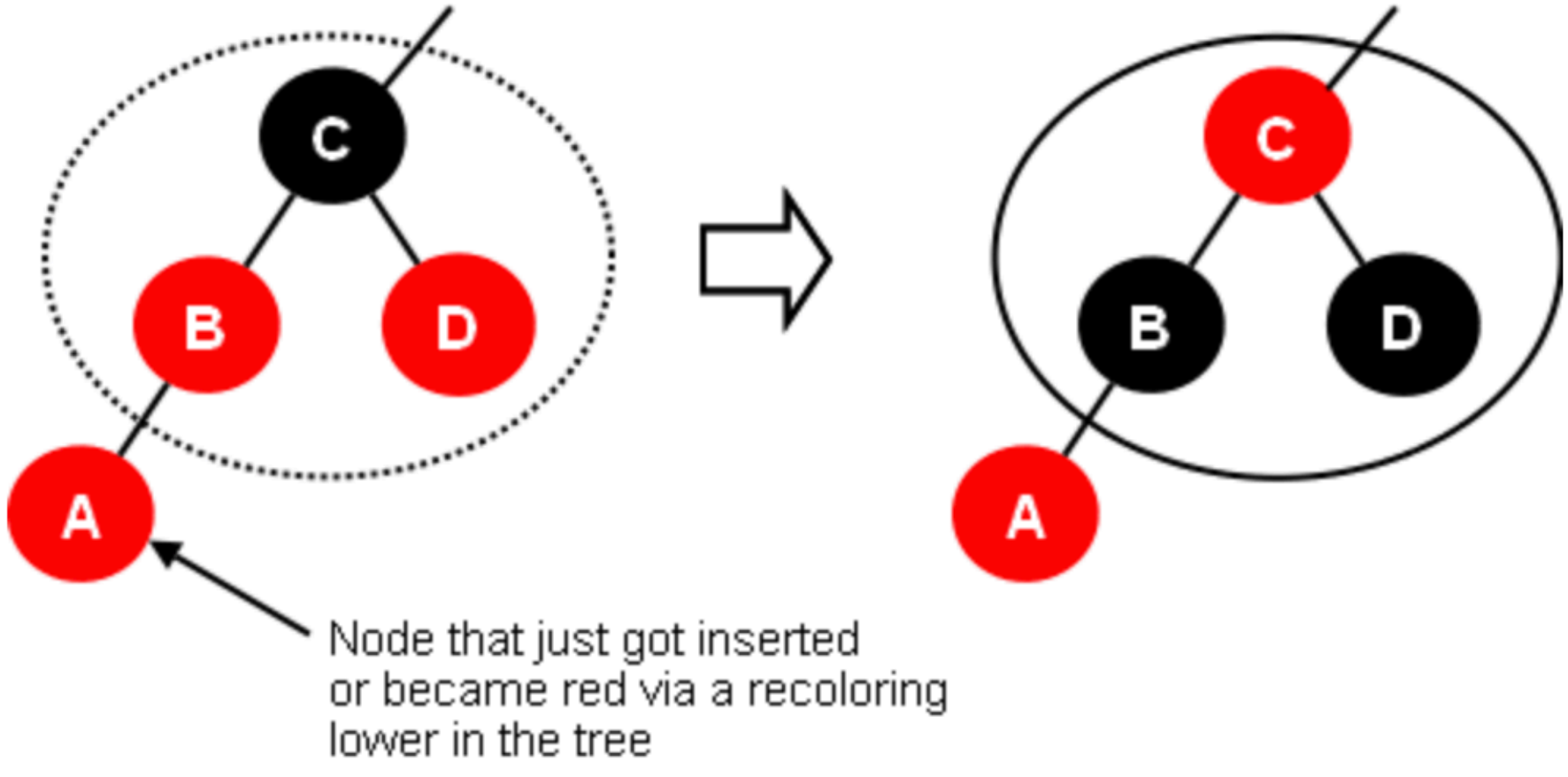
Balance

- In the Red-Black tree, we use two tools to do the balancing.
    1. Recoloring
    2. Rotation (Restructuring)
- Recolouring is the change in colour of the node i.e. if it is red then change it to black and vice versa.
    - It must be noted that the colour of the NULL node is always black.
- Moreover, we always **try recolouring first**, if recolouring doesn't work, then we go for rotation.

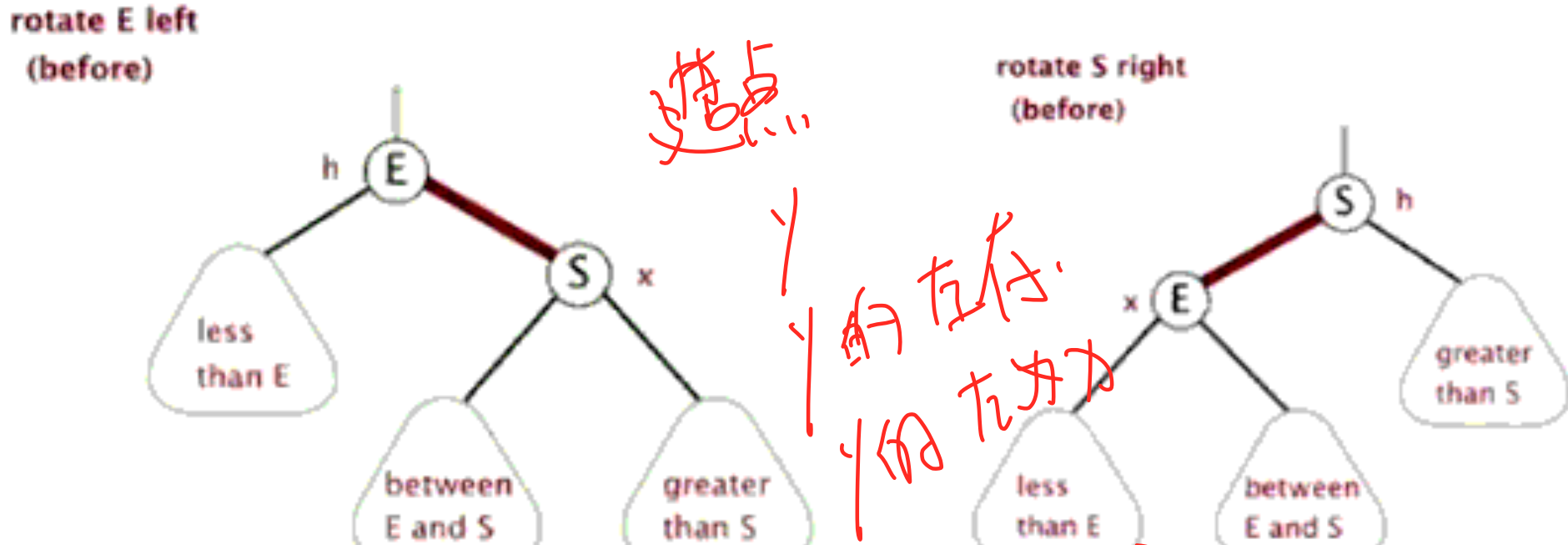
Recoloring

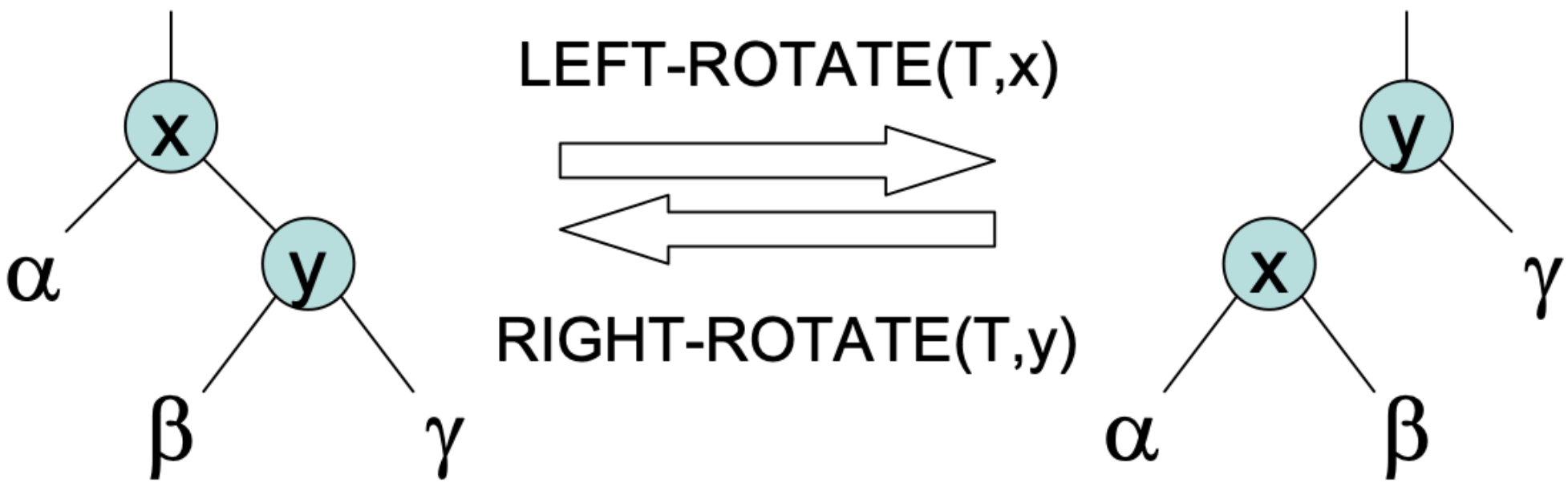
- Recolor whenever *the sibling of a red node's red parent is red*:

当红的父为空



Rotation (Restructuring)

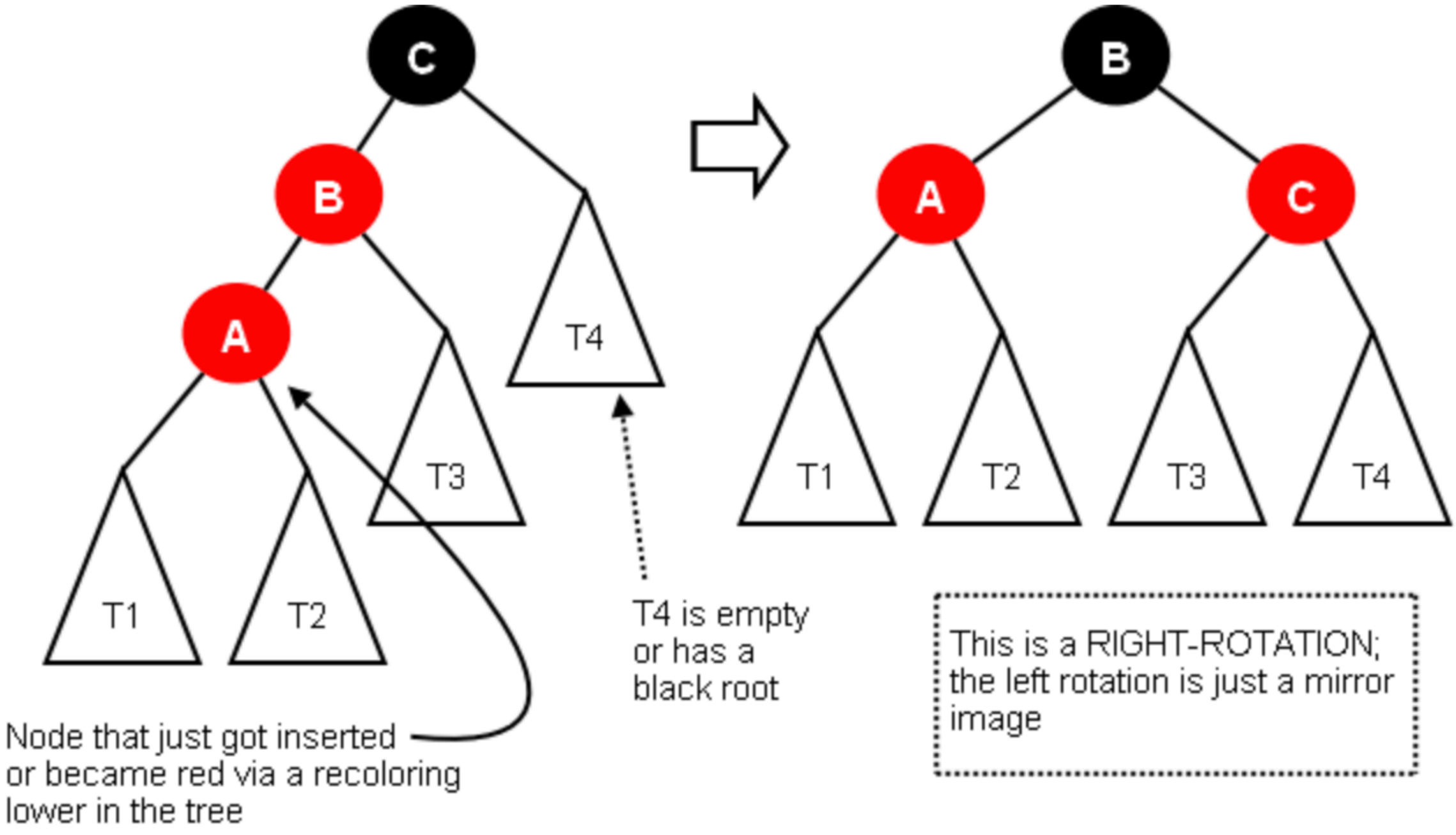




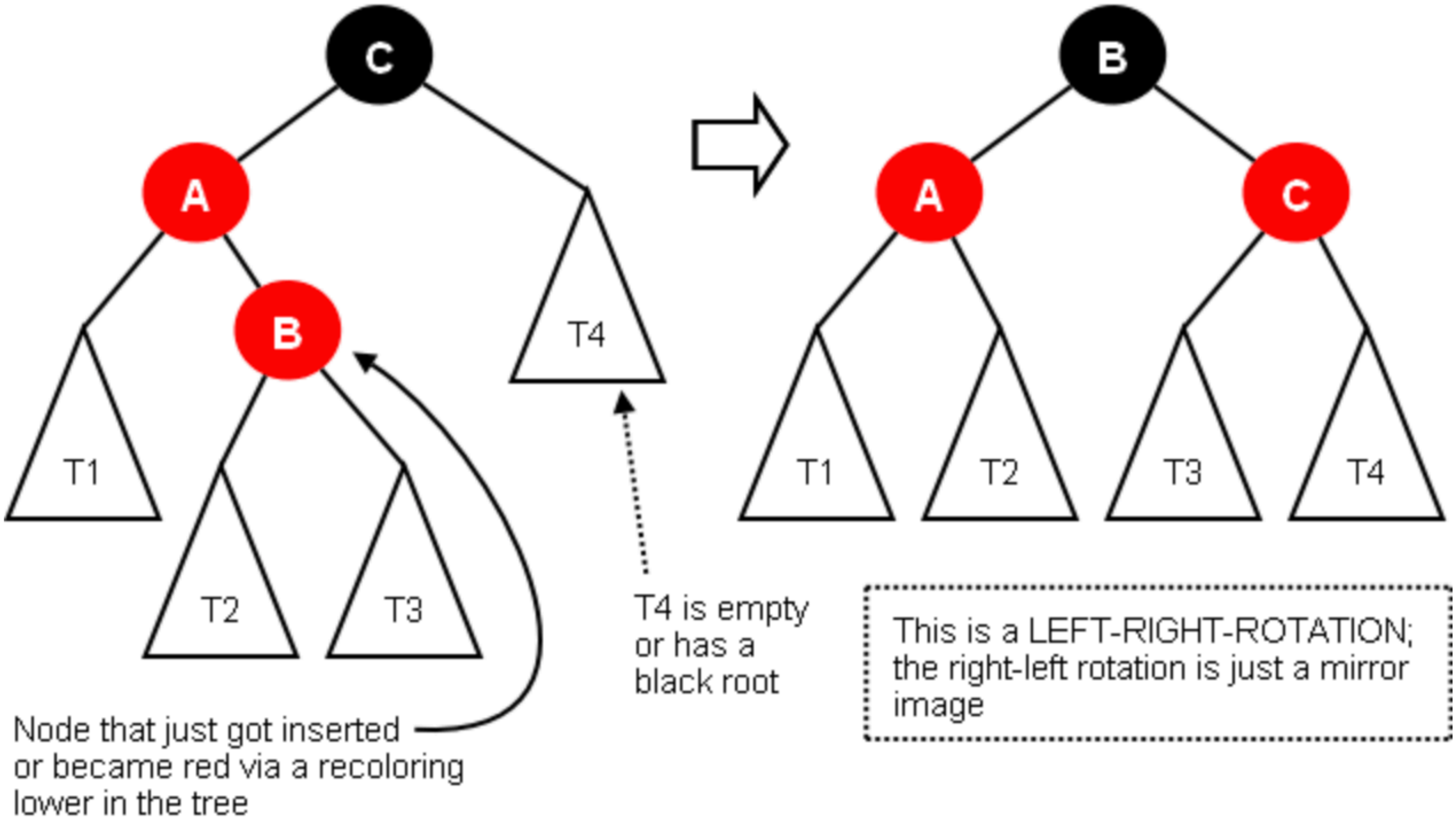
Rotation (Restructuring)

- Restructure whenever *the red child's red parent's sibling is black or null*. There are four cases:
  - Right
  - Left
  - Right-Left
  - Left-Right
- When you restructure, the root of the restructured subtree is colored black and its children are colored red.

Rotation (Restructuring)

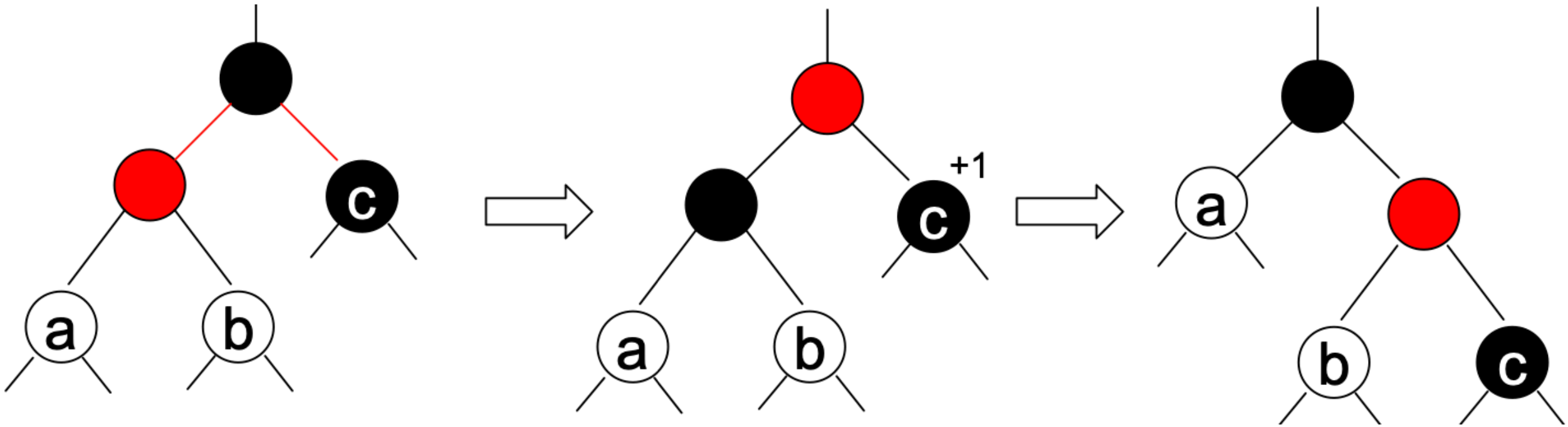


Rotation (Restructuring)



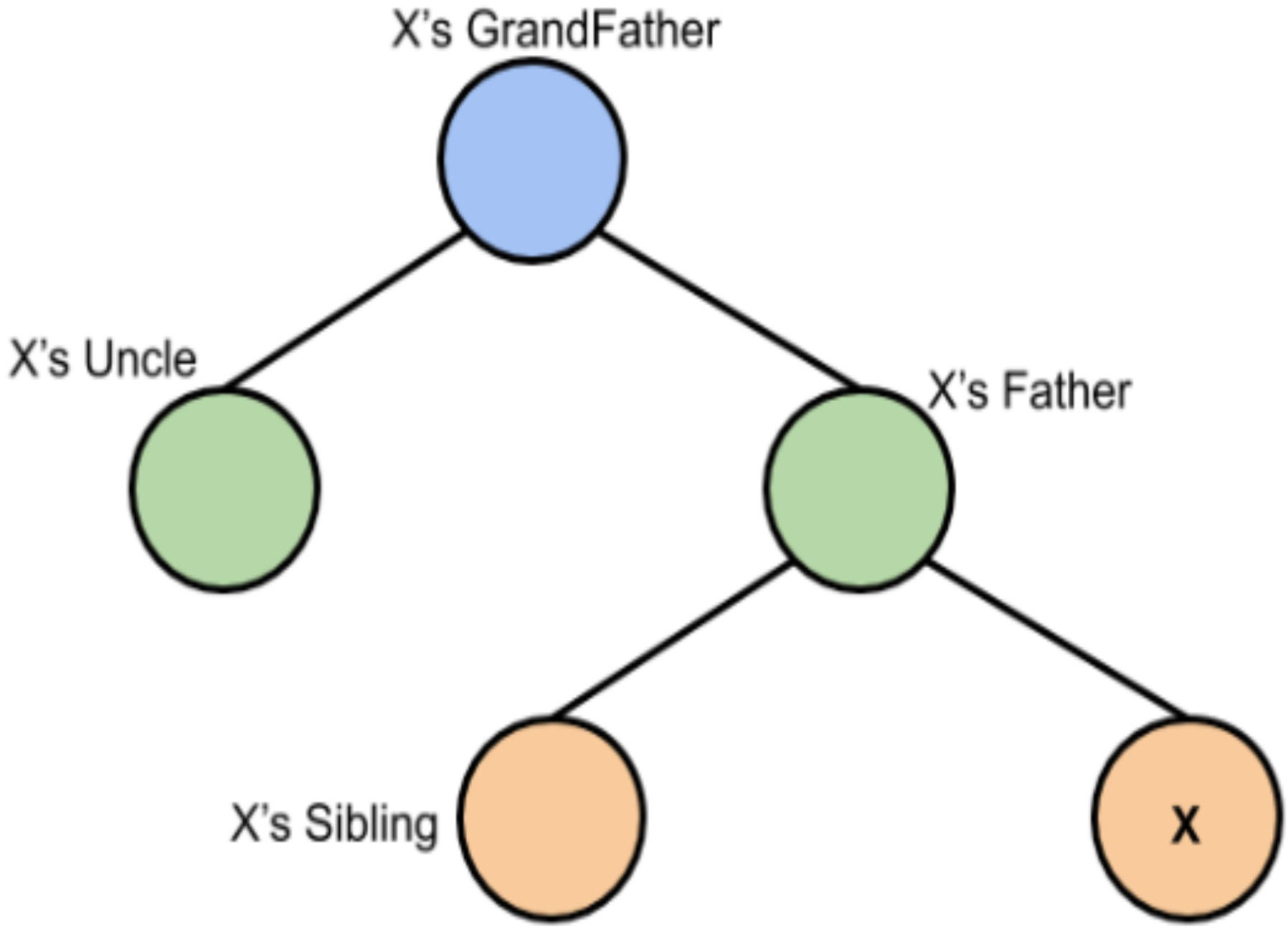
Common Problem

- A common problem and its solution are as following
  - ***double black***



- Red Black Tree
- Search
- Insertion and Deletion
    - Recoloring and Rotation
    - **Insertion**
    - Deletion

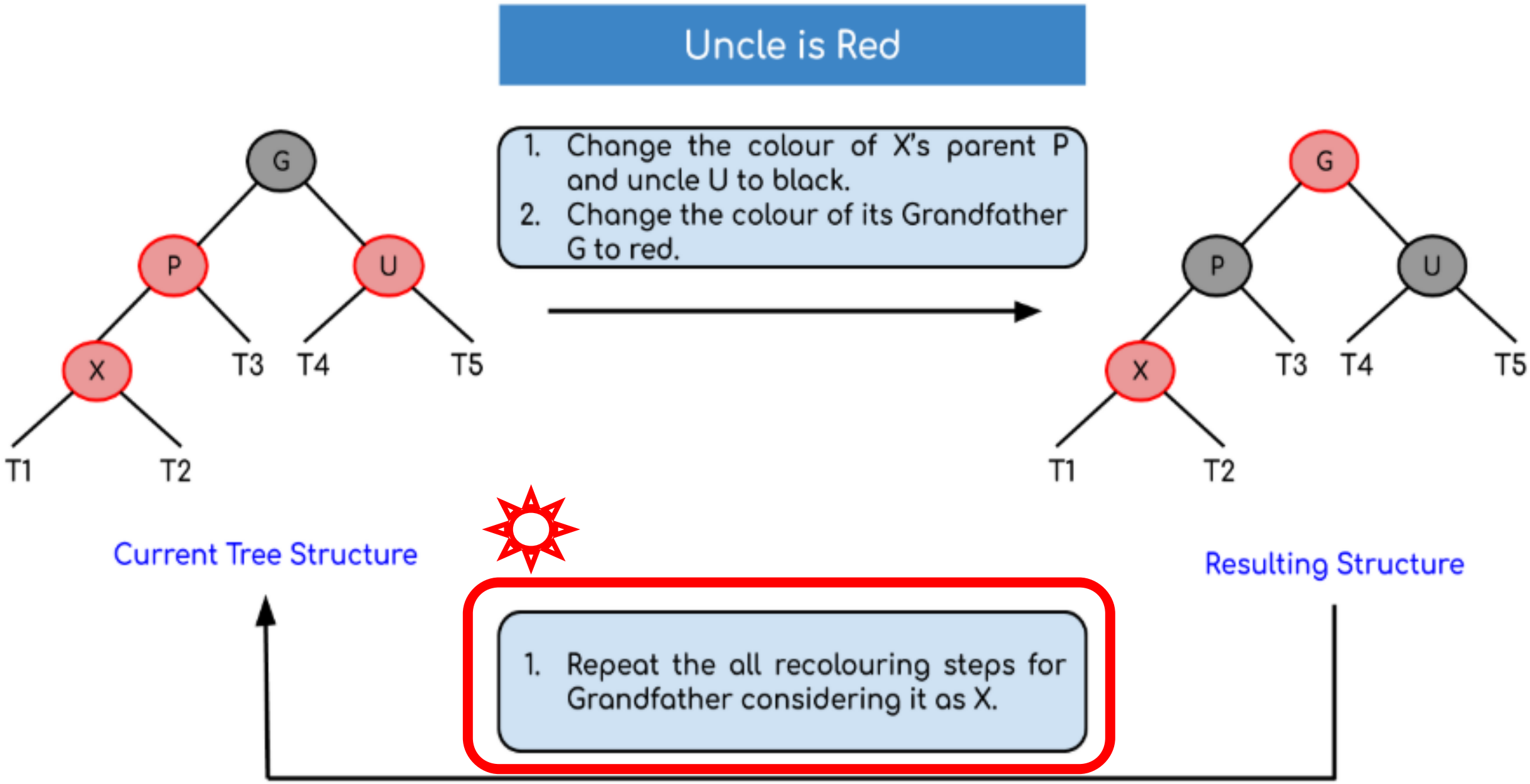
- **The representation we will be working with is:**



Insert

- First, you have to insert the node similarly to that in a binary tree and **assign a red colour** to it.
- If the node is a **root** node then change its colour to **black**
- ElseIf it does not then check the colour of the *parent node*.
  - If its colour (your parent node) is black then don't change the colour
  - ElseIf it is not i.e. **it is red** then **check the colour of the node's uncle**.
    - If **the node's uncle has a red colour** then change the colour of the node's parent and uncle to black and that of grandfather to red colour and repeat the same process for him (i.e. grandfather).

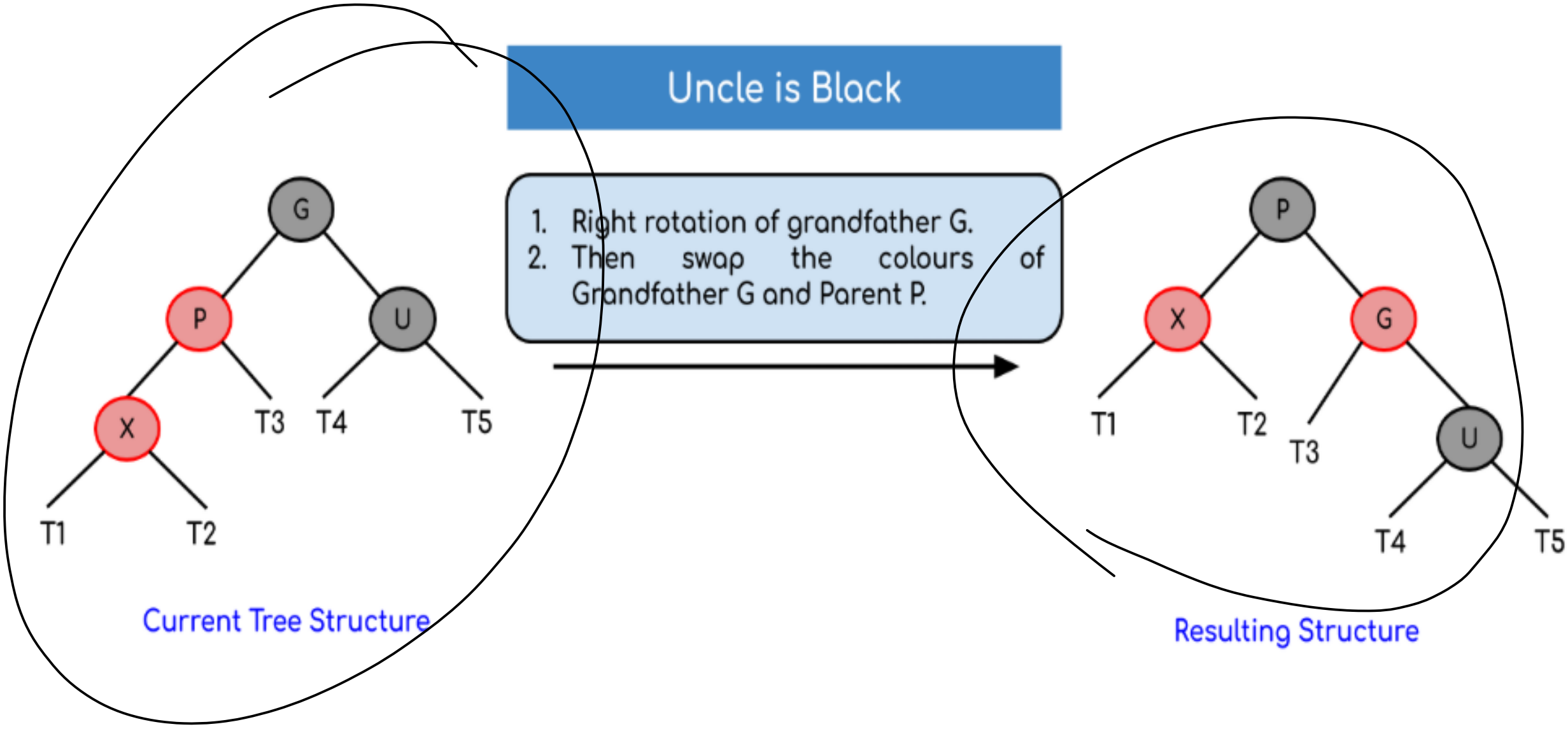
Case: Node x, P, and U are red



Insert (conti.)

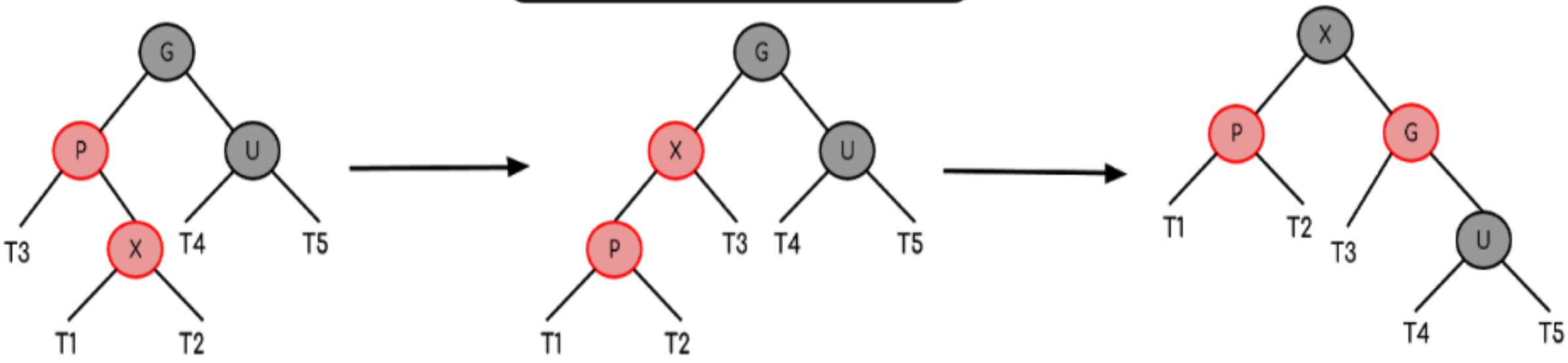
- ElseIf it is not i.e. it is red then **check the colour of the node's uncle**.
  - If the **node's uncle has a red colour** then change the colour of the node's parent and uncle to black and that of grandfather to red colour and repeat the same process for him (i.e. grandfather).
  - **ElseIf** the **node's uncle has black colour** then there are 4 possible cases:
    1. Left Left Case (LL rotation)
       - Note Left/Right means your position
    2. Left Right Case (LR rotation)
    3. Right Right Case (RR rotation)
    4. Right Left Case (RL rotation)

Left Left Case (LL rotation):



Left Right Case (LR rotation):

1. Left rotation of Parent P.
2. Then apply LL rotation case.

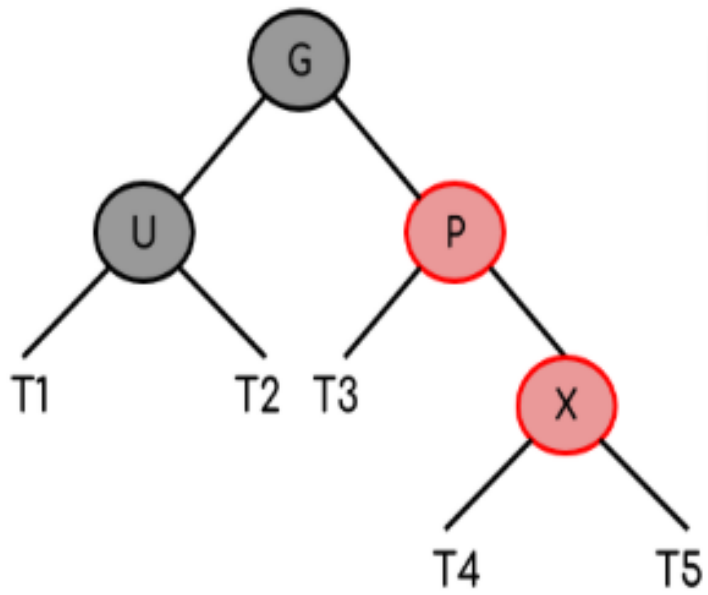


Current Tree Structure

Intermediate Tree Structure

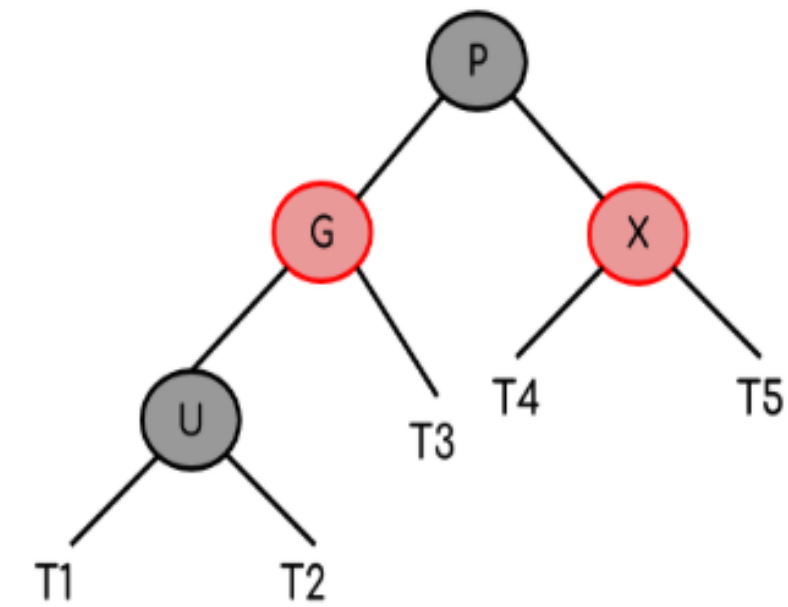
Resulting Structure

Right Right Case (RR rotation):



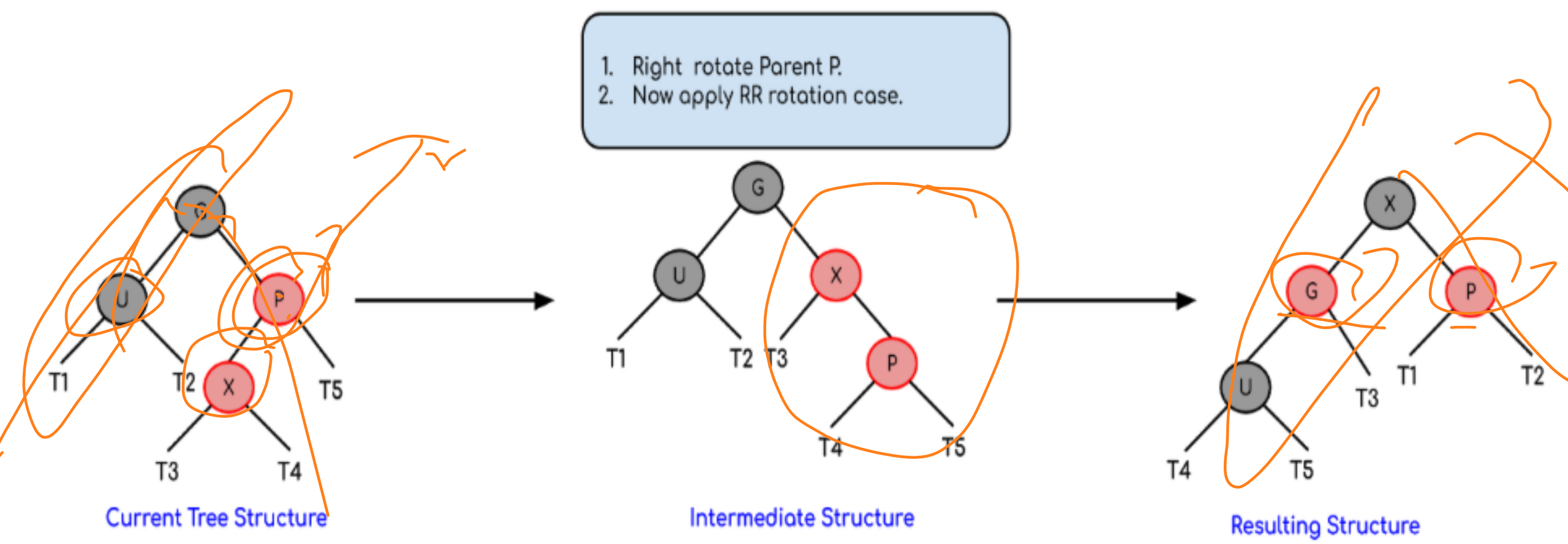
Current Tree Structure

1. Left rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.

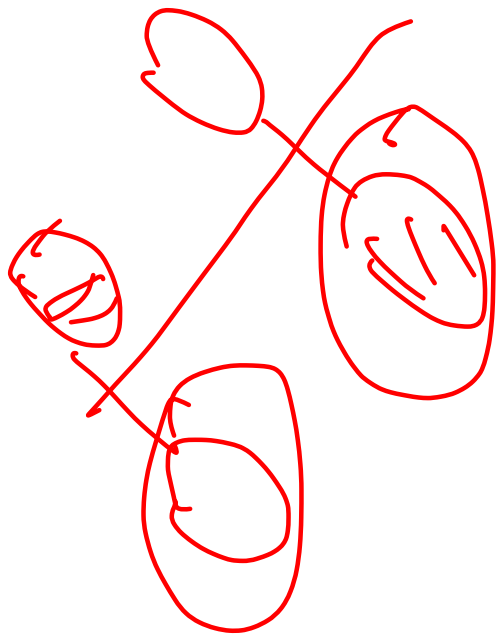


Resulting Structure

Right Left Case (RL rotation):



- Now, after these rotations, if the colours of the nodes are miss matching then recolour them.



Algorithm

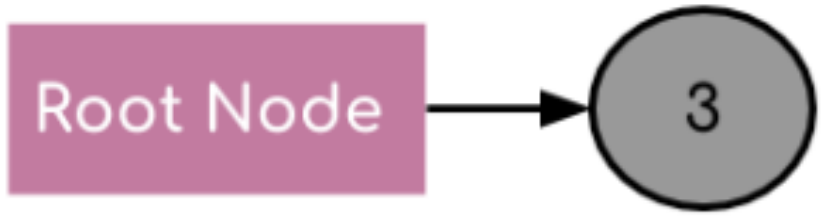
Let x be the newly inserted node.

1. Perform standard BST insertion and make the colour of newly inserted nodes as **RED**.
2. If x is the root, change the colour of x as **BLACK** (Black height of complete tree increases by 1).
3. Do the following if the color of x's parent is not BLACK **and** x is not the root.
    1. **a) If x's uncle is RED** (Grandparent must have been black)
        - **(i)** Change the colour of parent and uncle as BLACK.
        - **(ii)** Colour of a grandparent as RED.
        - **(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.**
    2. **b) If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**)
        - **(i)** Left Left Case (p is left child of g and x is left child of p)
        - **(ii)** Left Right Case (p is left child of g and x is the right child of p)
        - **(iii)** Right Right Case (Mirror of case i)
        - **(iv)** Right Left Case (Mirror of case ii)

Example 1

- Creating a red-black tree with elements 3, 21, 32 and 17 in an empty tree.
- **Solution:**

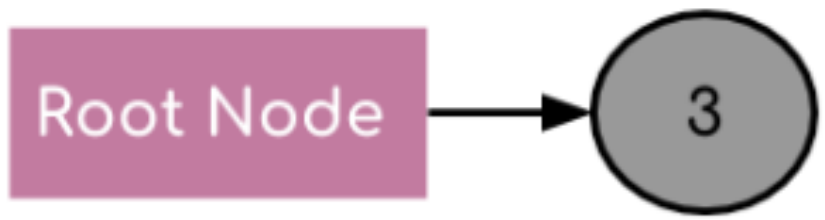
Step 1: Inserting element 3 inside the tree.



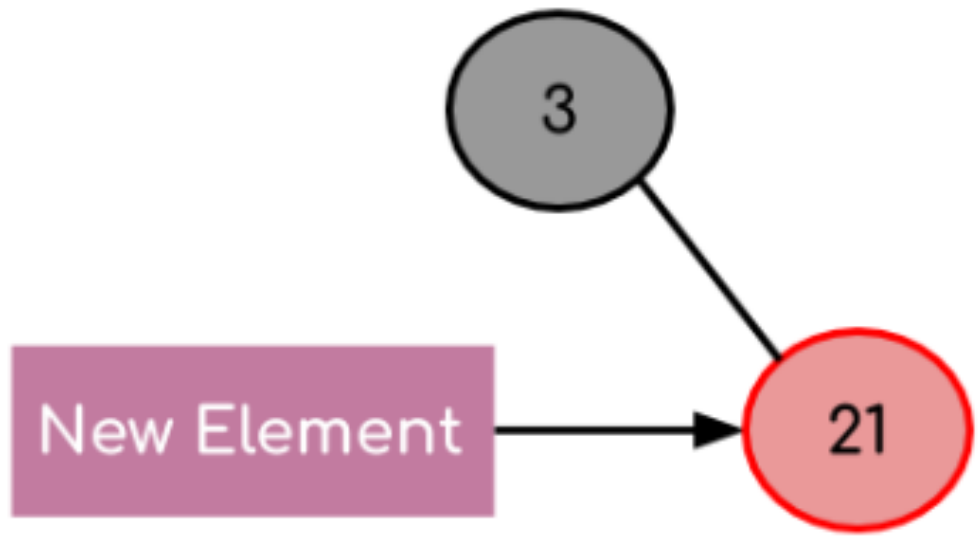
When the first element is inserted it is inserted as a root node and as root node has black colour so it acquires the colour black.

Example 1

Step 1: Inserting element 3 inside the tree.

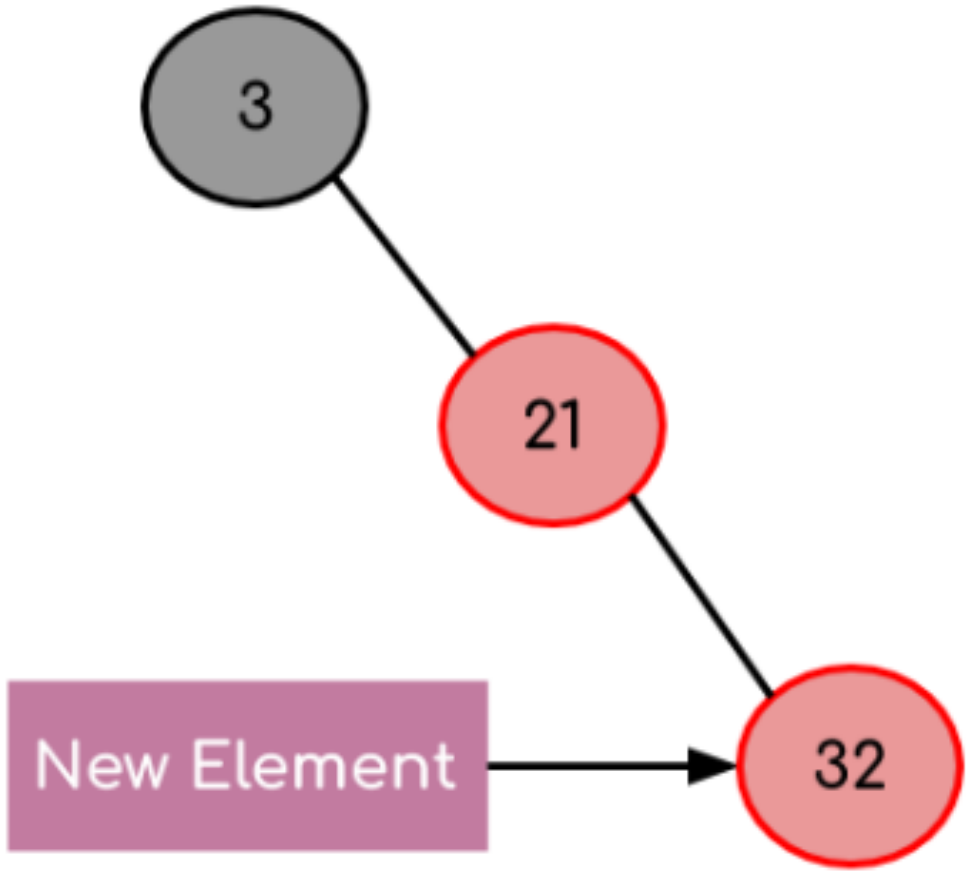


Step 2: Inserting element 21 inside the tree.

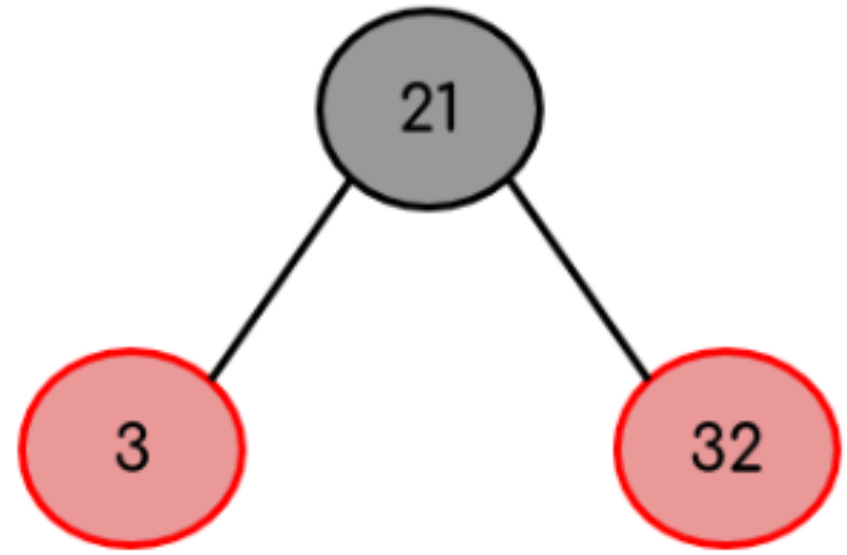


The new element is always inserted with a red colour and as 21 > 3 so it becomes the part of the right subtree of the root node.

Step 3: Inserting element 32 inside the tree.

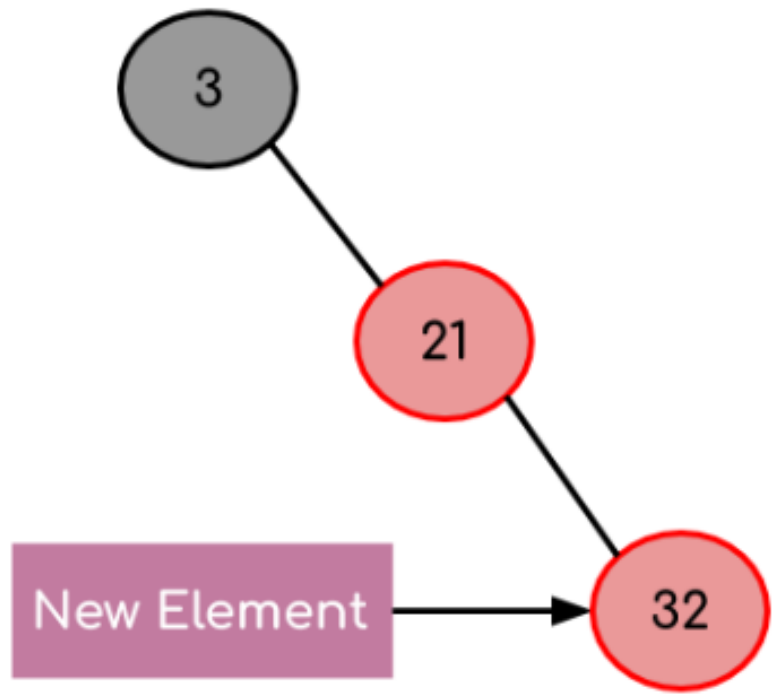


Here we see that as two red node are not possible and also we can see the conditions of RR rotation so it will follow RR rotation and recolouring to balance the tree.

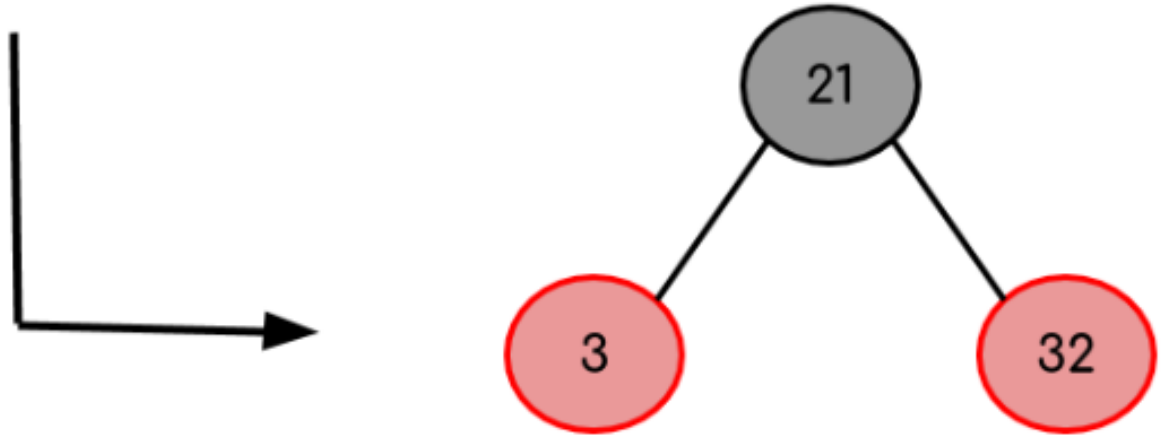


Example 1

Step 3: Inserting element 32 inside the tree.

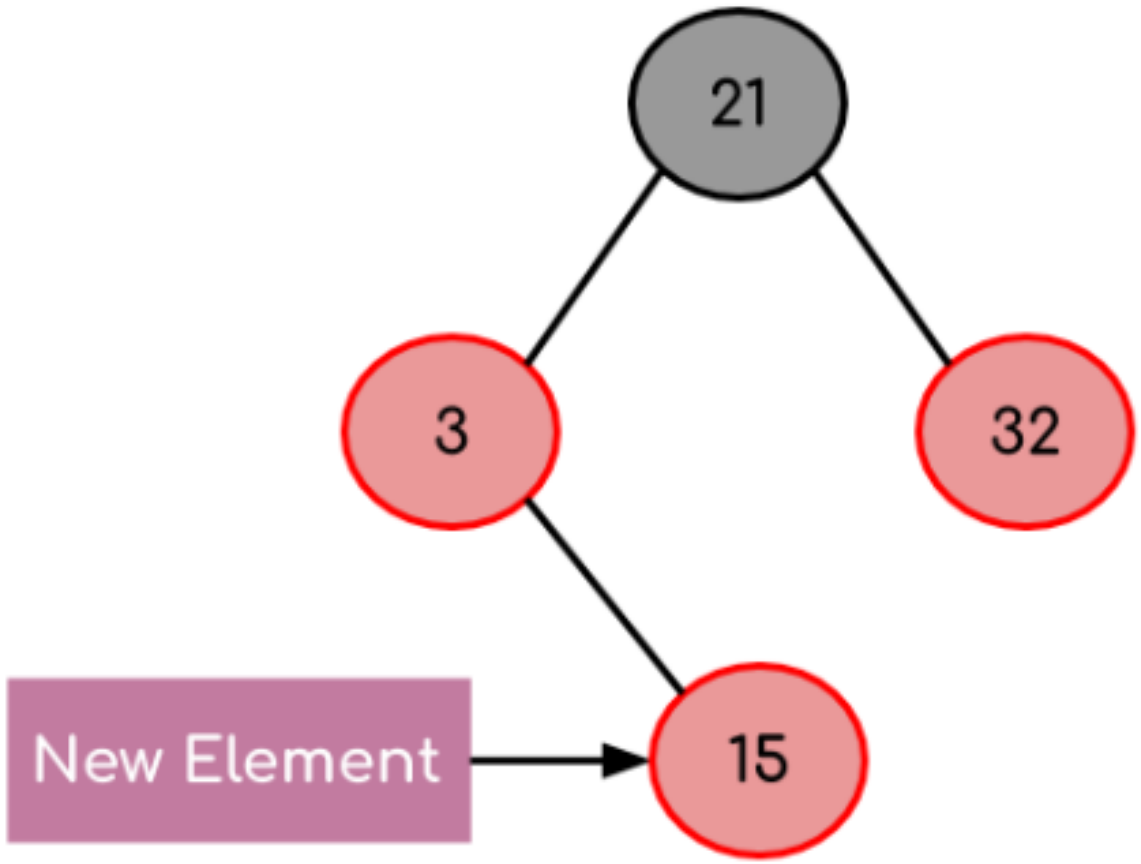


Here we see that as two red node are not possible and also we can see the conditions of RR rotation so it will follow RR rotation and recolouring to balance the tree.

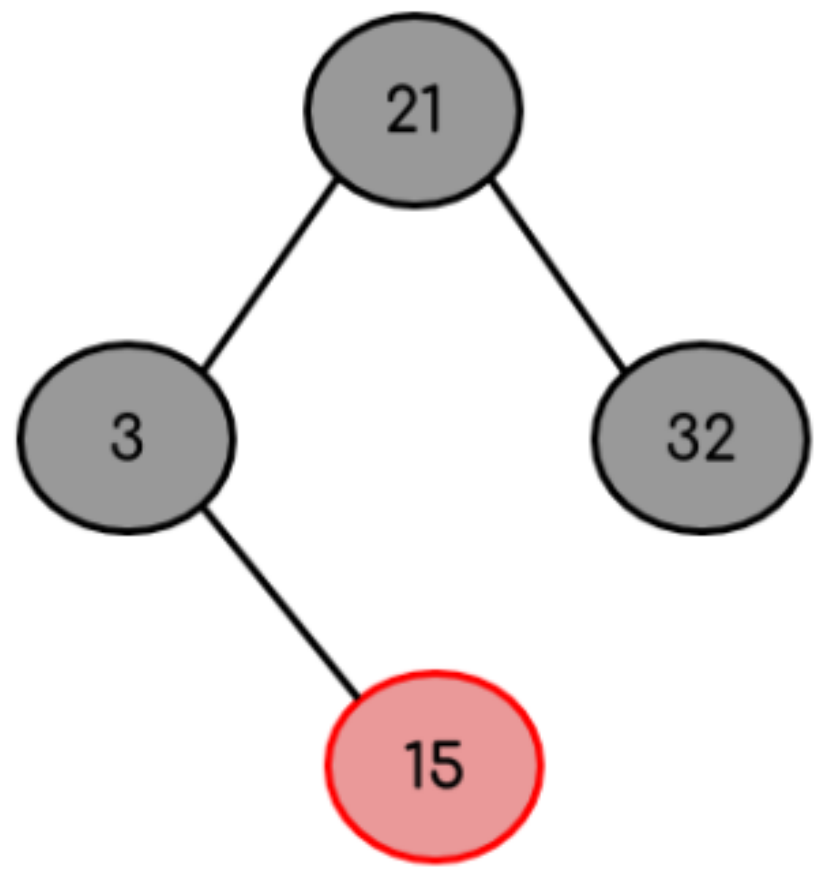


Now, as we insert 32 we see there is a red father-child pair which violates the Red-Black tree rule so we have to rotate it. Moreover, we see the conditions of RR rotation (considering the null node of the root node as black) so after rotation as the root node can't be Red so we have to perform recolouring in the tree resulting in the tree shown above.

Step 4: Inserting element 17 inside the tree.

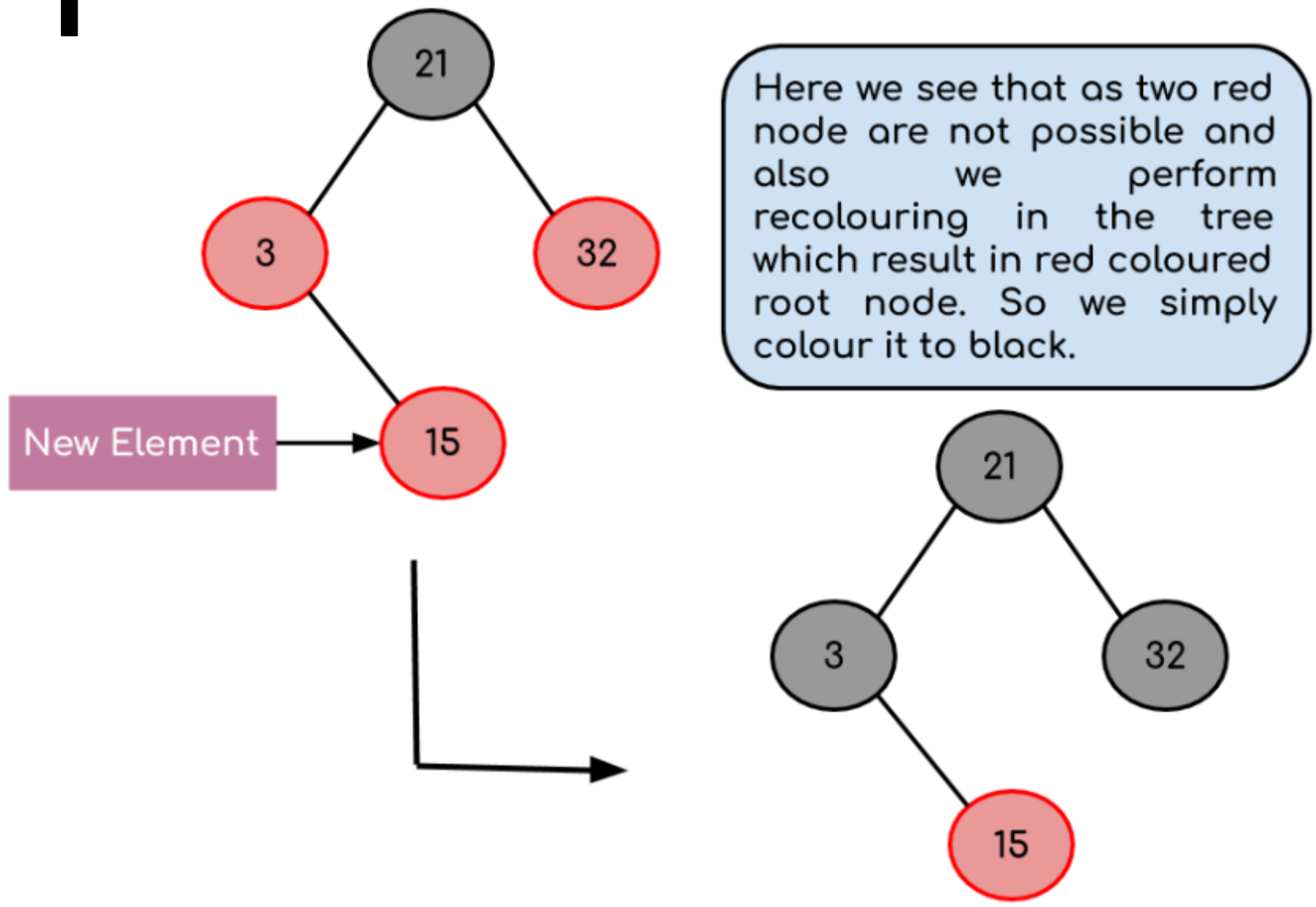


Here we see that as two red node are not possible and also we perform recolouring in the tree which result in red coloured root node. So we simply colour it to black.



Example 1

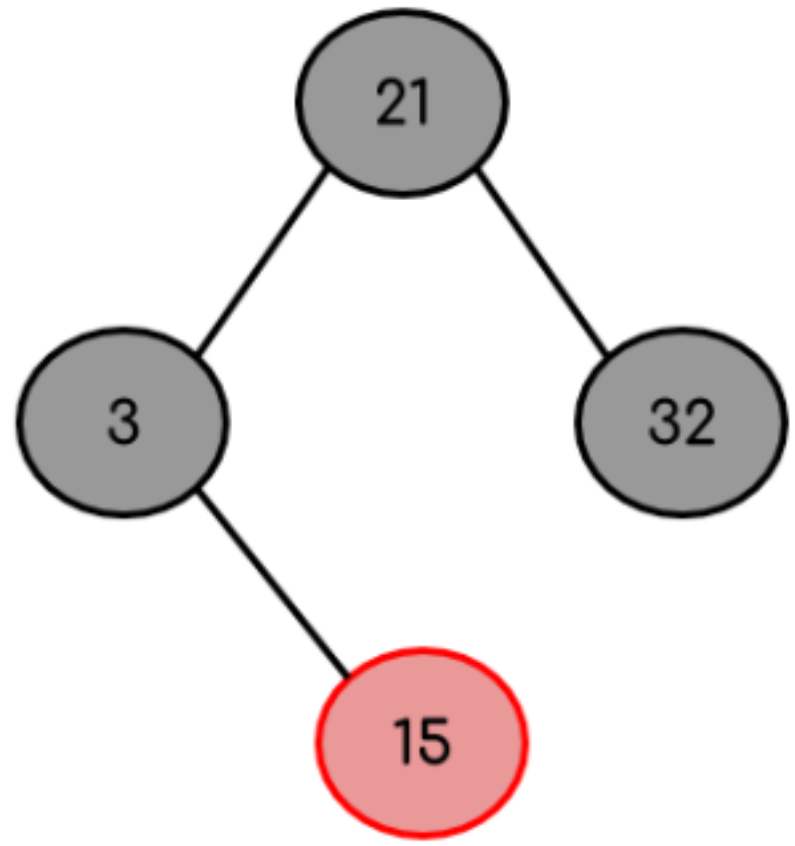
Step 4: Inserting element 17 inside the tree.



Now when we insert the new element the parent and the child node with red colour appear again and here we have to recolour them. We see that both the parent node and the uncle node have red colour so we simply change their colour to black and change the grandfather colour to red. Now, as a grandfather is the root node so we again change its colour to black resulting in the tree shown above.

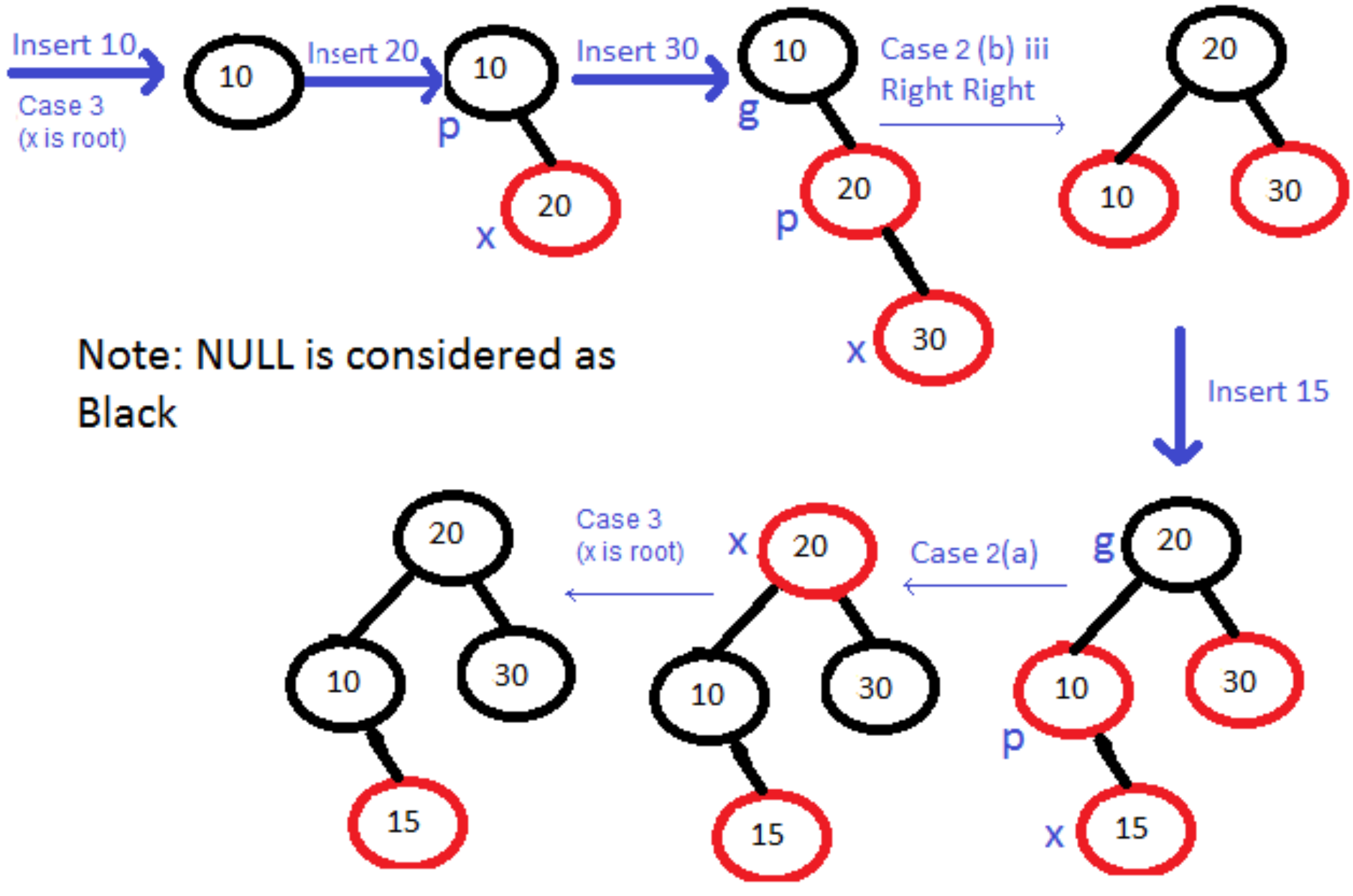
Example 1

- **Final Tree Structure:**



Example 2

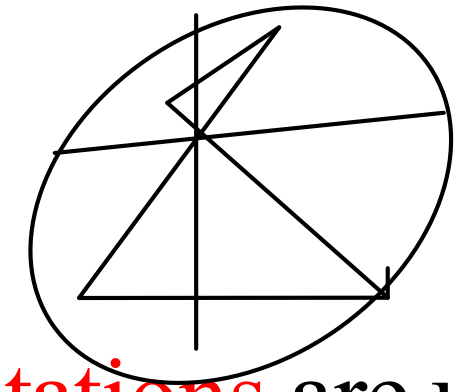
Insert 10, 20, 30 and 15 in an empty tree



Note: NULL is considered as Black

- Red Black Tree
- Search
- Insertion and Deletion
    - Recoloring and Rotation
    - Insertion
    - **Deletion**

Delete



- **Insertion Vs Deletion:**
  - Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.
  - In the insert operation, we check the color of the uncle to decide the appropriate case.
  - In the delete operation, ***we check the color of the 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.

Delete

- To understand deletion, the 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**.

双黑

删除黑的补一个

此时，

这个儿子

是双份黑，

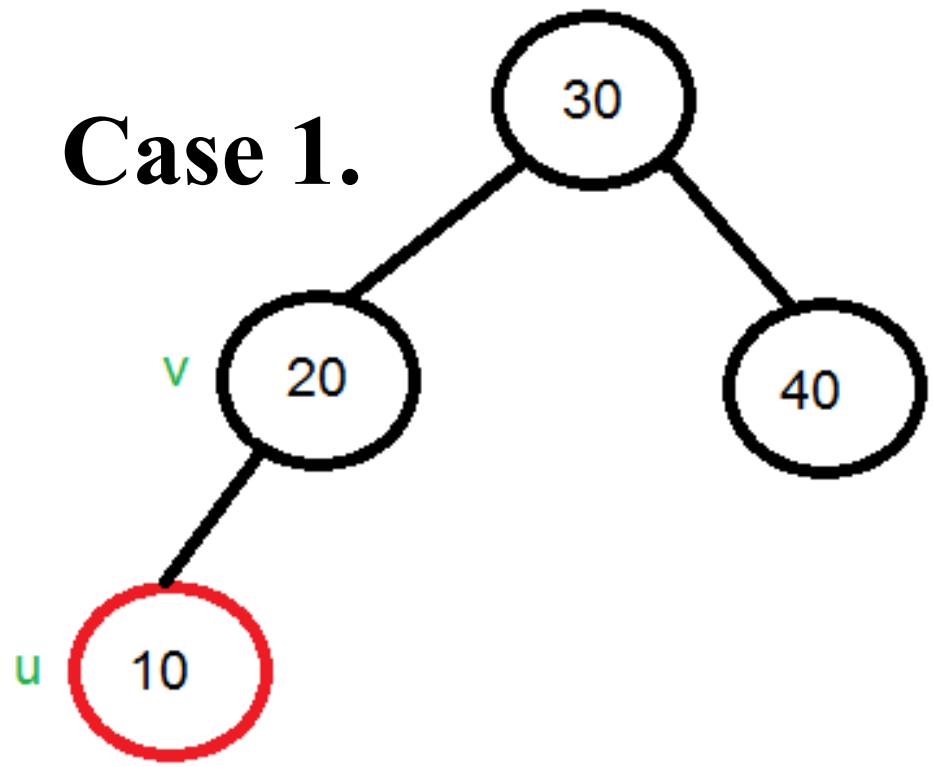
Recall the deletion on standard BST

- Perform standard BST delete.
- When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child).
- So we only need to handle cases where a node is leaf or has one child.

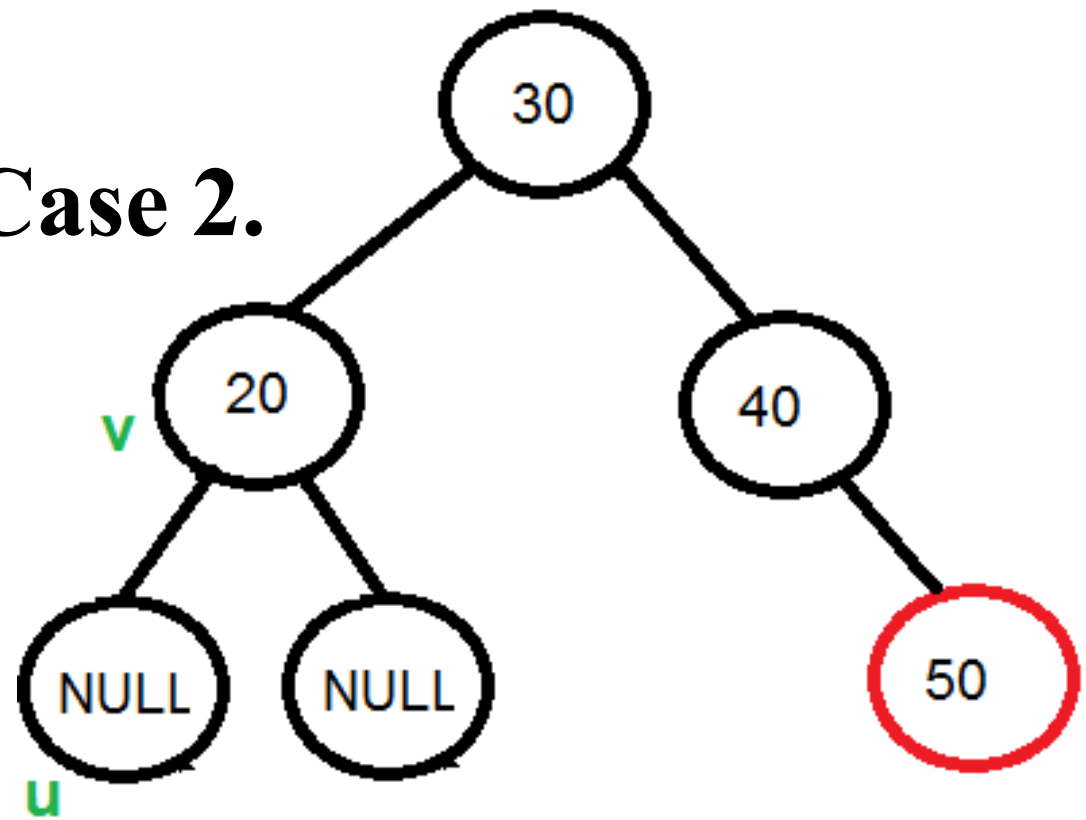
Delete

- 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**)
- **Case 1. either u or v is red.**
- **Case 2. both u and v are Black.**
- **Case 3.** If u is root, make it single black and return (Black height of complete tree reduces by 1).

Case 1.



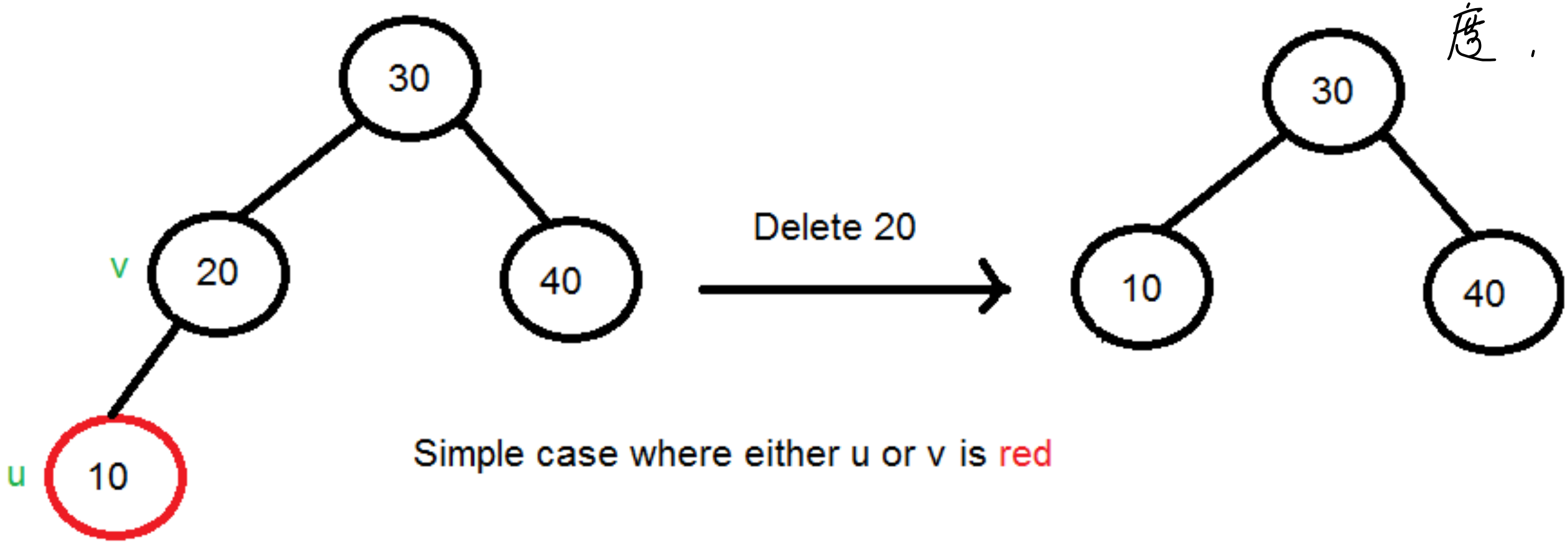
Case 2.



Delete ⇒看兄弟点

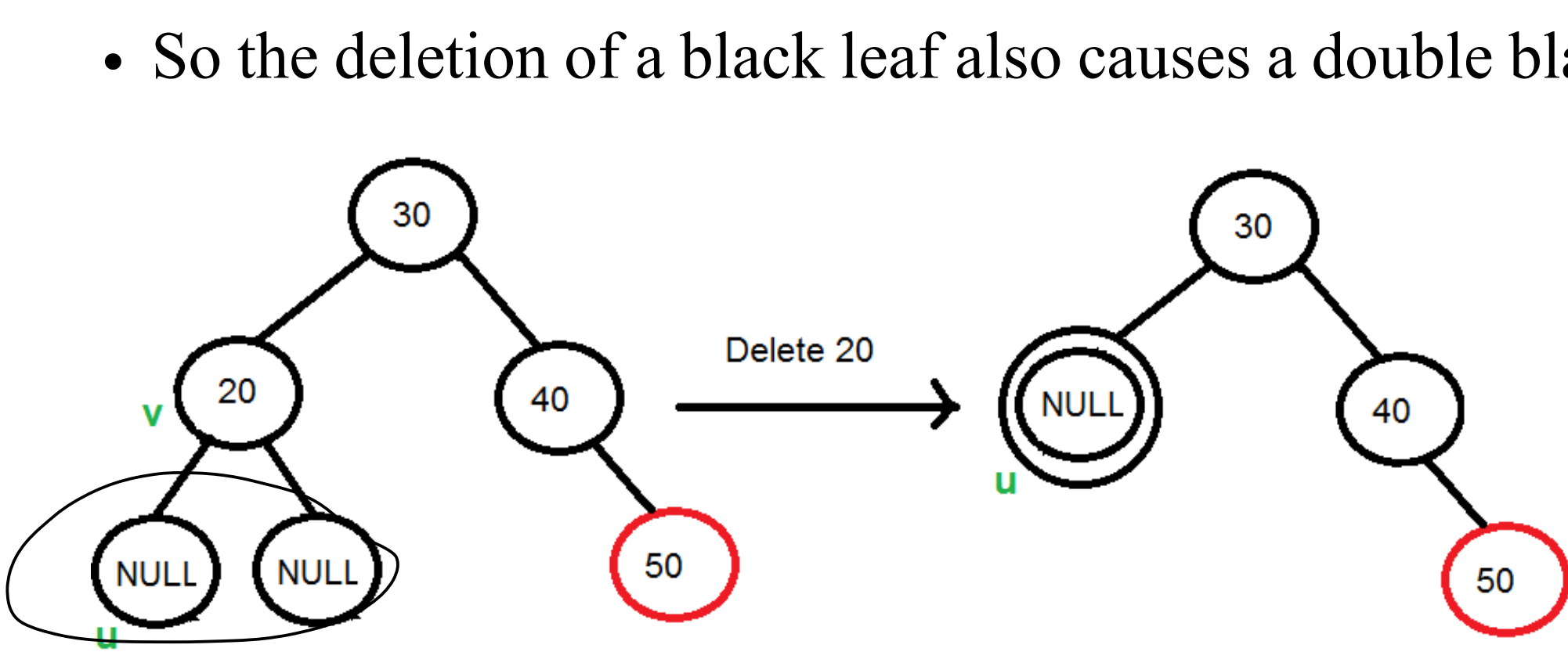
- **Case 1. either u or v is red.**
  - We mark the **replaced child** as black (No change in black height).
    - Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.

无论是 u黑 v红 还是 u黑u红，都把 child u 改为黑色即可，此时不改变黑高度.



- **Case 2. both u and v are Black.**
  - Color u as **double black**.
  - Now our task reduces to convert this double black to single black.
    - Note that if v is leaf, then u is NULL and color of NULL is considered black.
    - So the deletion of a black leaf also causes a double black.

u, v 都黑，v是叶子也不要啊！

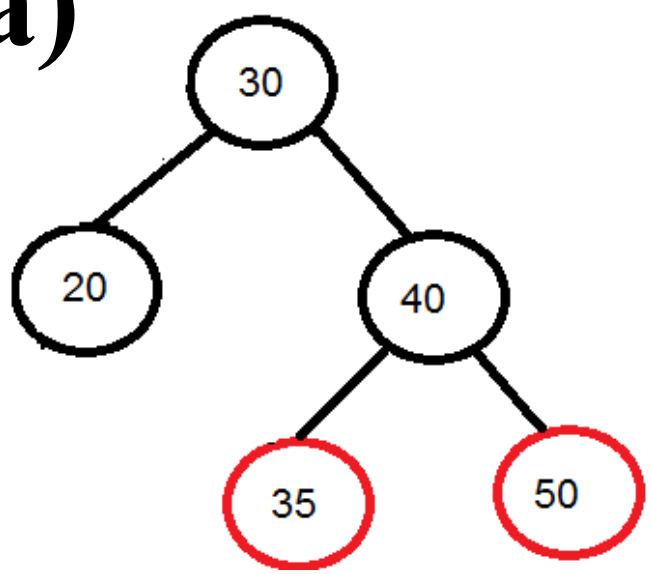


When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.

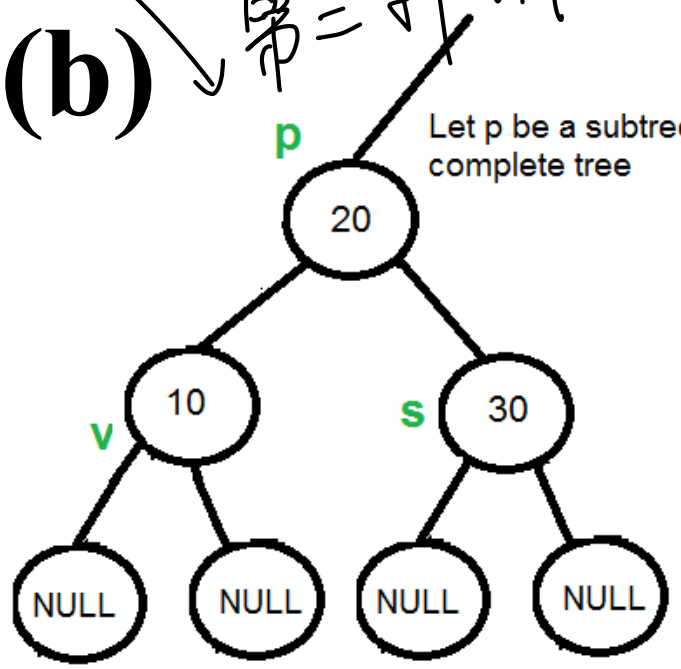
Note that deletion is not done yet, this double black must become single black

- **Case 2. both u and v are Black.**
  - Do following while the current node u is double black, and it is not the root. Let sibling of node be **s**.
    - **(a): If sibling s is black and at least one of sibling's children is red**
    - **(b): If sibling is black and its both children are black**
    - **(c): If sibling is red**

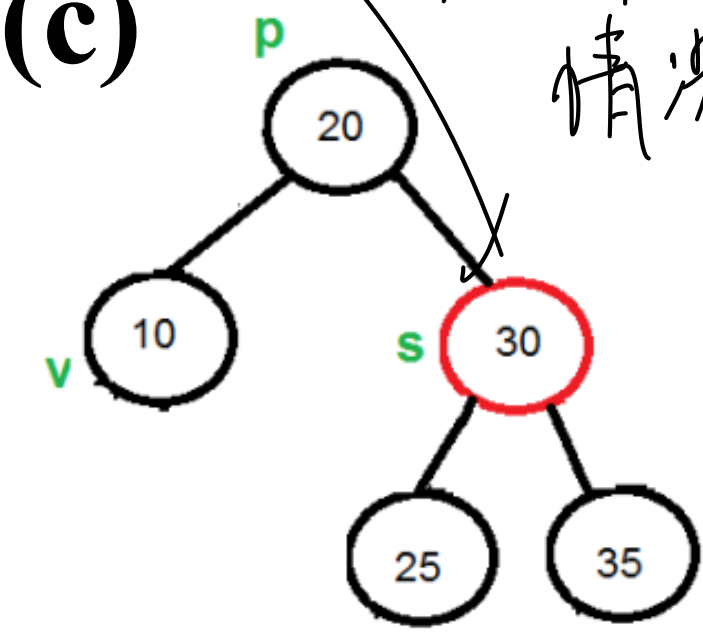
(a)



(b)

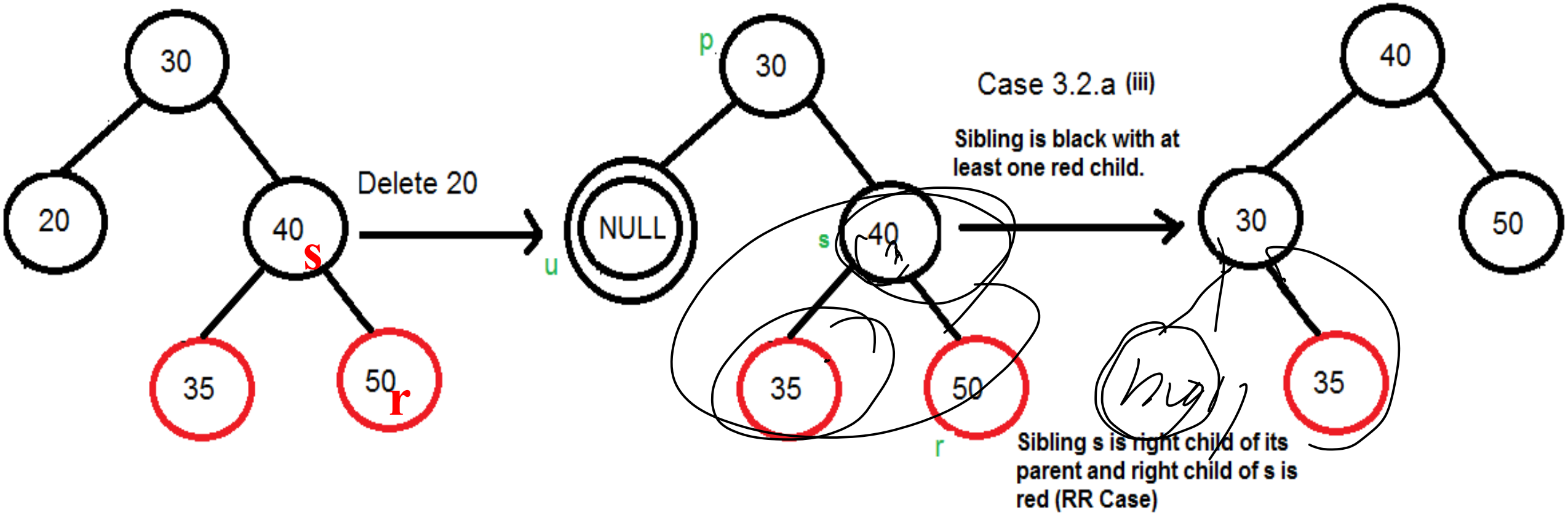


(c)

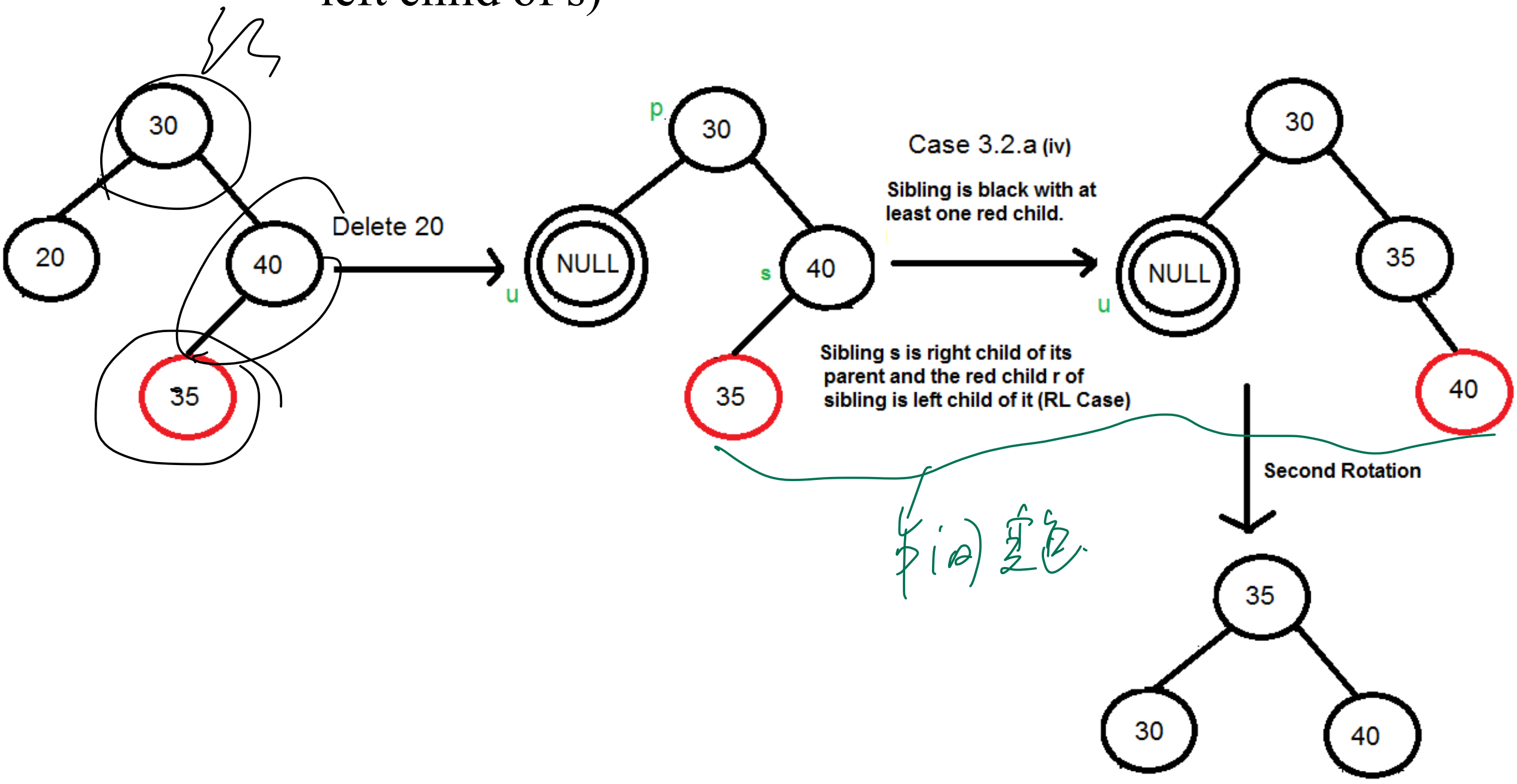


- **(a): If sibling s is black and at least one of sibling's children is red**, perform **rotation(s)**.
  - Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.
  - **(i)** Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.
  - **(ii)** Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.
  - **(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)
  - **(iv)** Right Left Case (s is right child of its parent and r is left child of s)

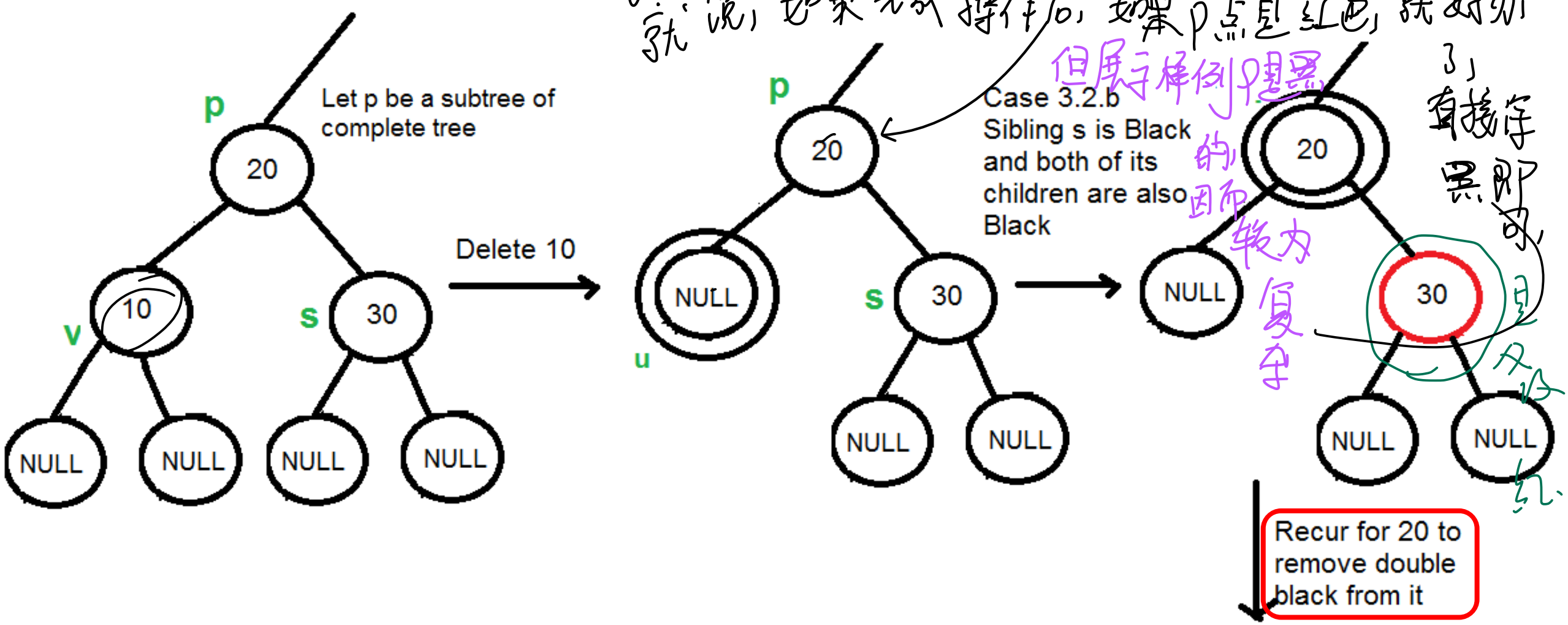
- **(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



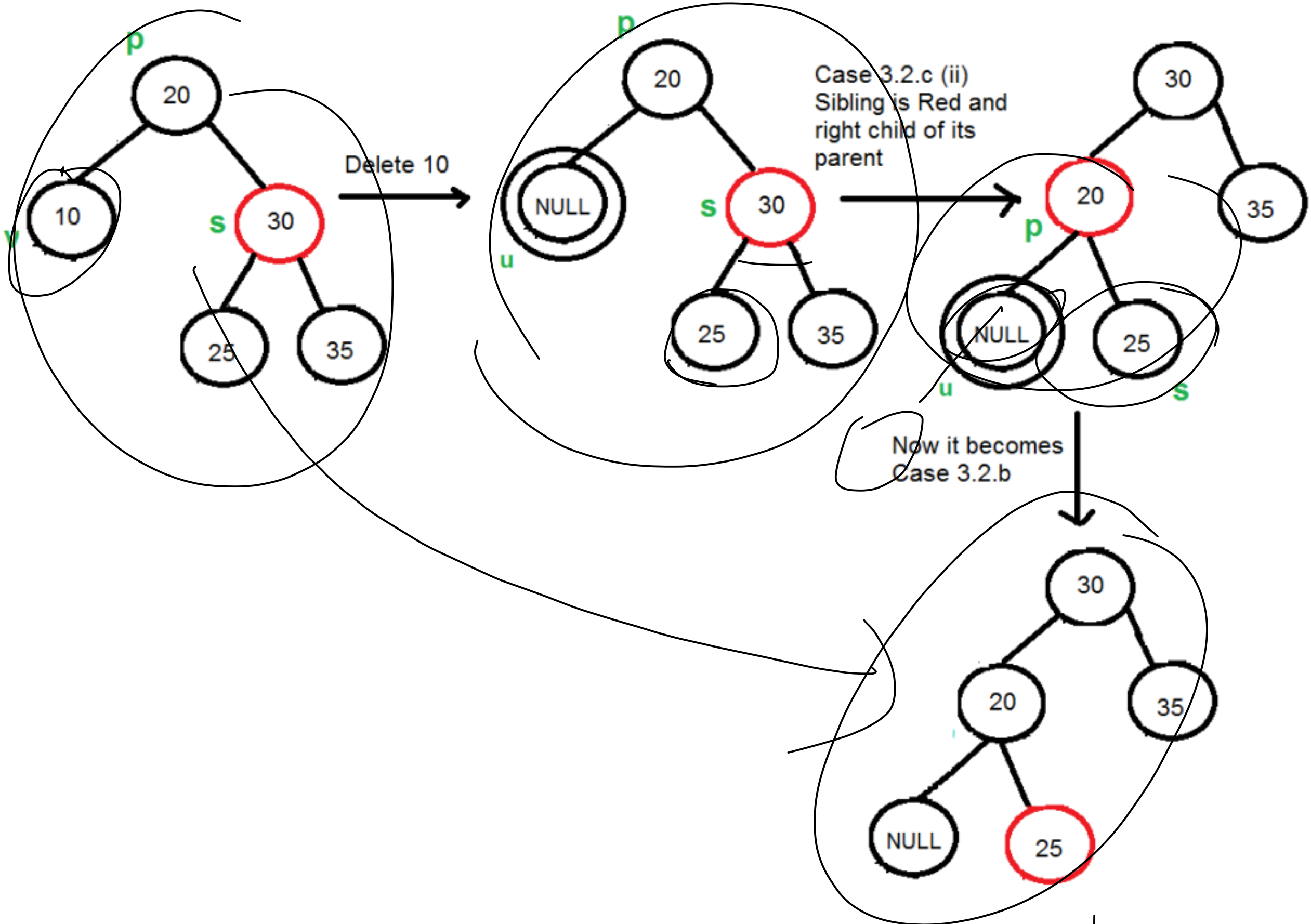
- **(iv)** Right Left Case (s is right child of its parent and r is left child of s)

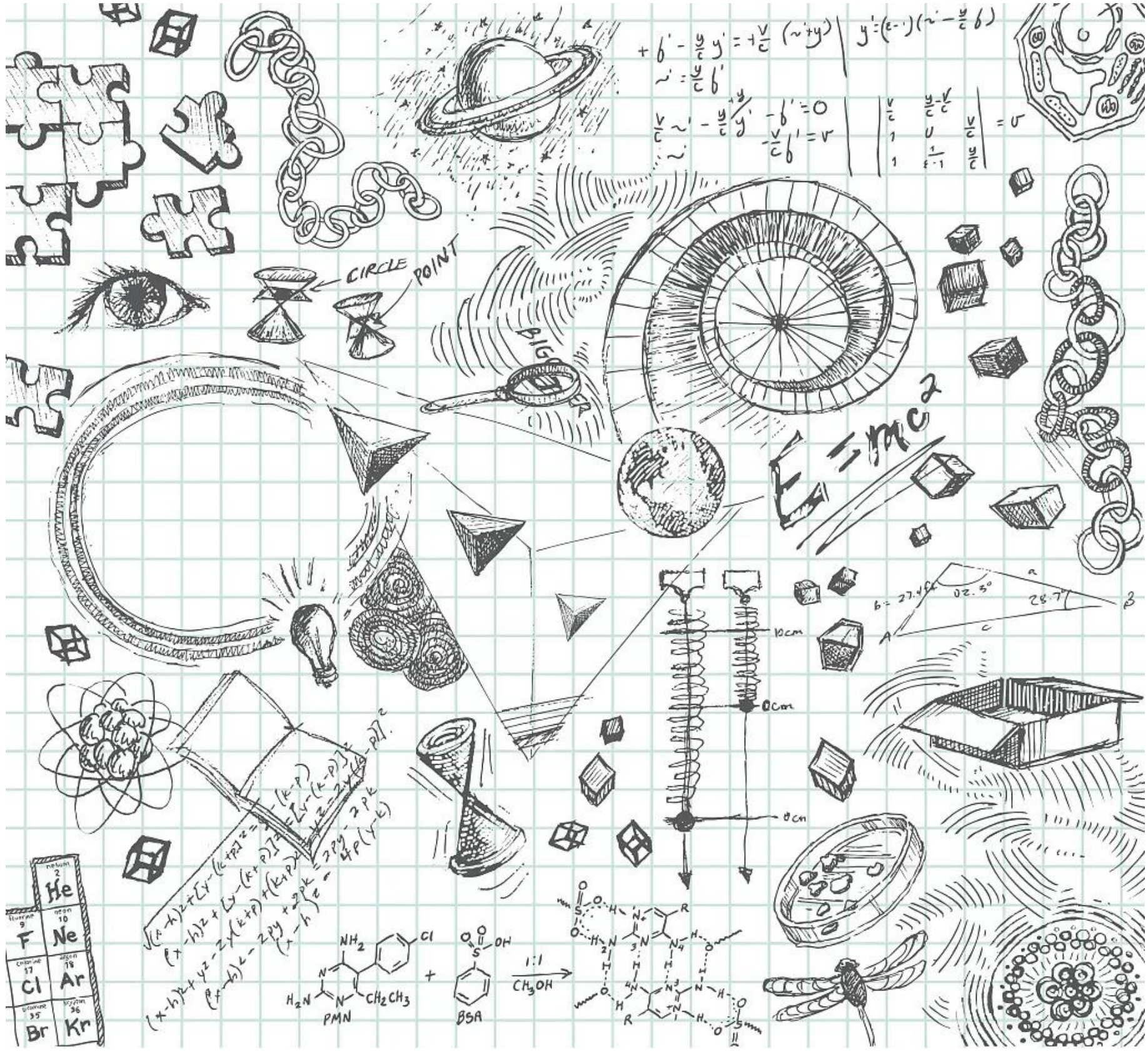


- **(b): f sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black.
  - Note if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)



- **(c): If sibling is red**, perform a rotation to move old sibling up, recolor the old sibling and parent.
  - The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.
    - **(i)** Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.
    - **(ii)** Right Case (s is right child of its parent). We left rotate the parent p.


p
20
10
y
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
p
20
35
NULL
u
25
s
Now it becomes
Case 3.2.b
30
20
35
NULL
25


CIRCLE
POINT
BIG
E=mc2
10cm
0cm
8cm
PMN
BSA
1:1
CH3OH