Red Black Tree

Dr. Tanvir Ahmed

Red Black Tree (RB Tree)

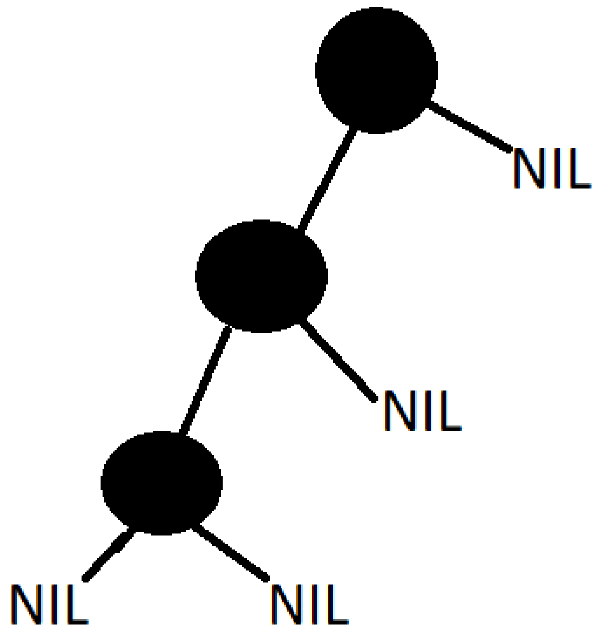
- This is another self balancing binary search tree.
- You already should know that a regular binary search tree can result in O(n) search operation
- As part of balancing, you have learned AVL tree, 2-4 tree, Splay tree, Treap etc.
- RB tree is another self balancing tree
    - Same run-time as AVL tree
    - Balancing condition is less restricted in RB tree compared to AVL tree
    - Example of RB tree is tree map and sorted map in java
- This is the final balanced binary tree structure we are going to study.
- Although it won’t seem like it at first, these trees are quite similar to 2-4 Trees.

Red Black Tree (RB Tree)

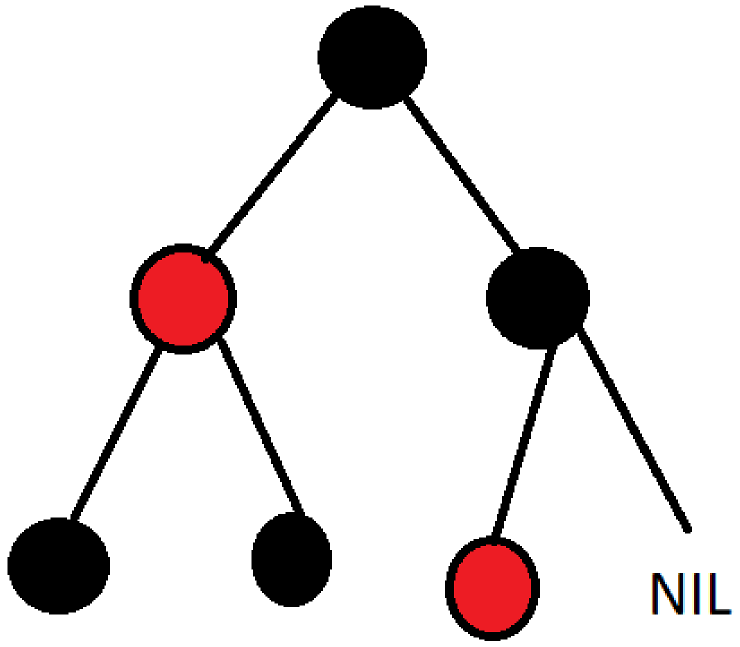
- Here are the rules for a valid RedBlack Tree:
- 1) Every node is red or black
- 2) Must be a valid Binary Search Tree, with each node colored red or black.
- 3) The root is colored **black.**
- 4) The children of a **red** node are **black**. (No Red-Red parent child relationship)
- 5) Every external node is **black**. (These are the "null nodes" that are the children of the leaf nodes.) (Sometimes they are also called leaf node and they are NIL and black color)
- 6) All the external black nodes have the same black depth. The black depth of a node is defined as the number of black colored ancestors minus one.
    - (another way: For each node, all paths from the node to descendant leaves contain the same number of black nodes.)

Examples

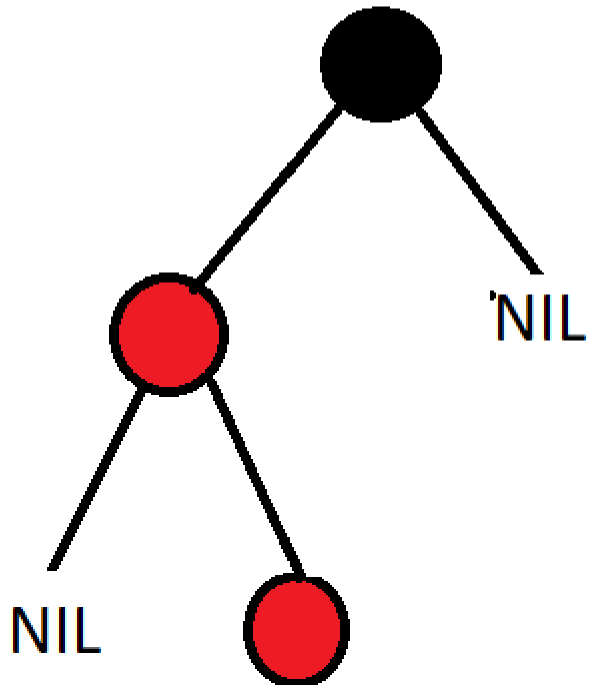
- Is it an RB tree?



- Is it an RB tree?

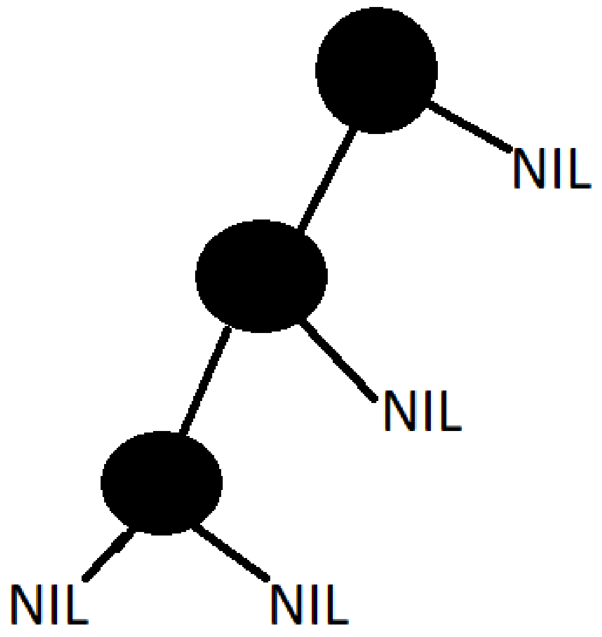


- Is it an RB tree?



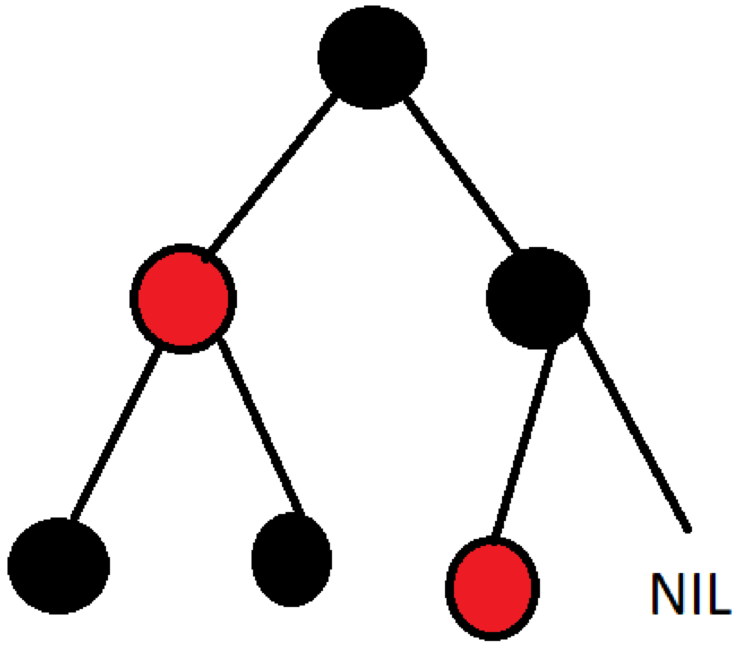
Examples

- Is it an RB tree?



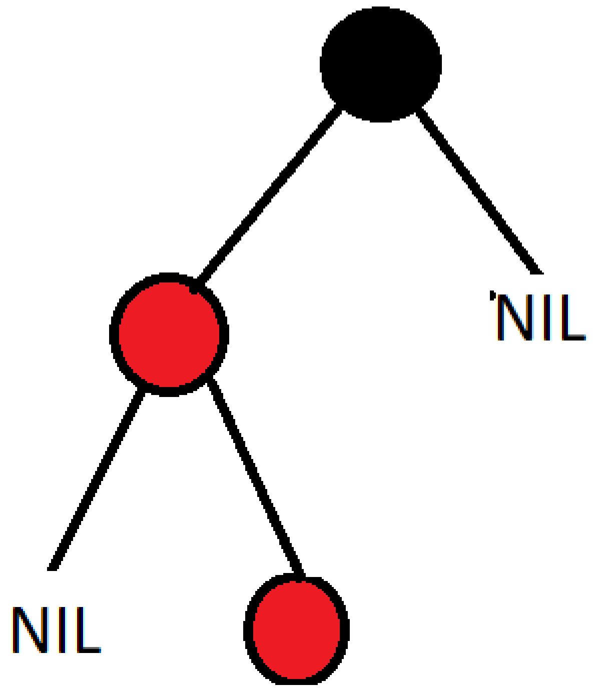
- No, it is violating property 6 from our rules (count the black nodes from root to leaf. Thy are not same)

- Is it an RB tree?



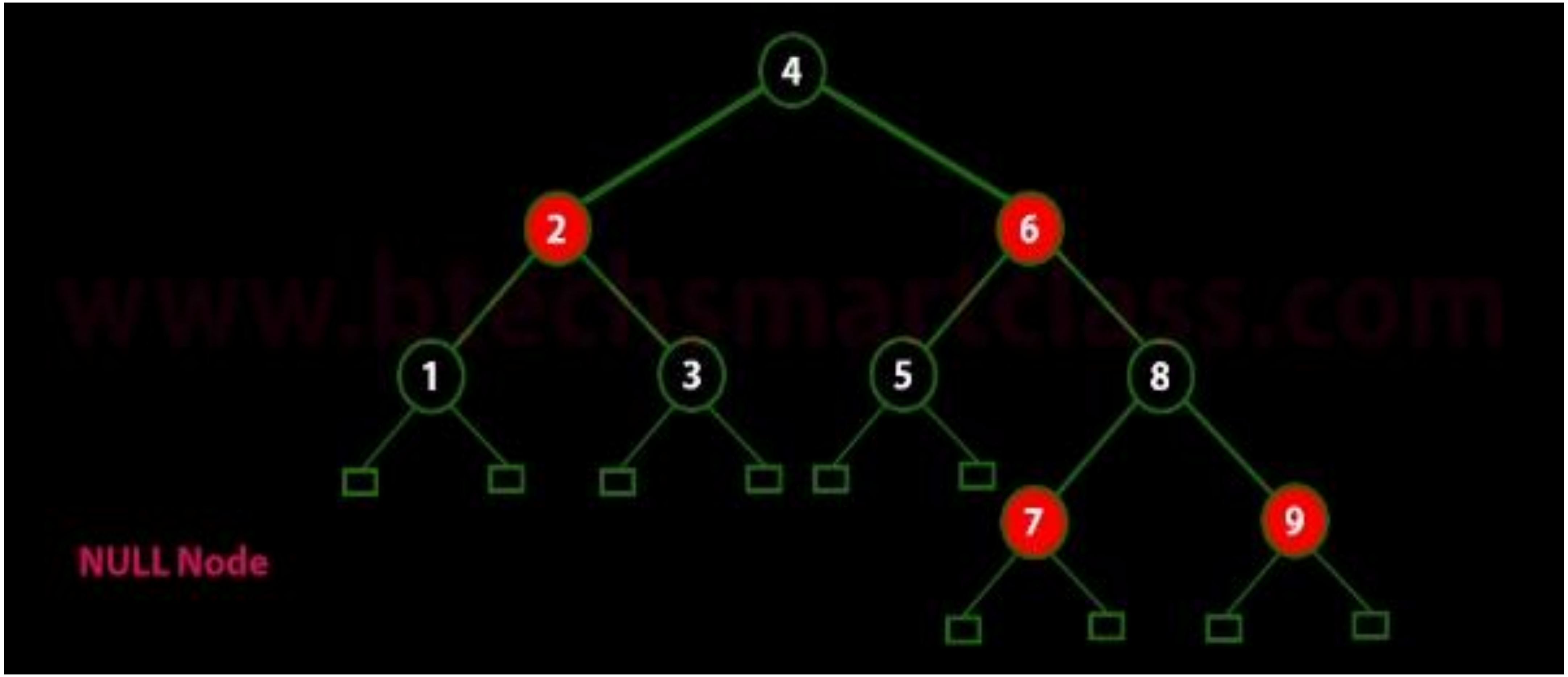
- Yes, fulfilled all the properties

- Is it an RB tree?



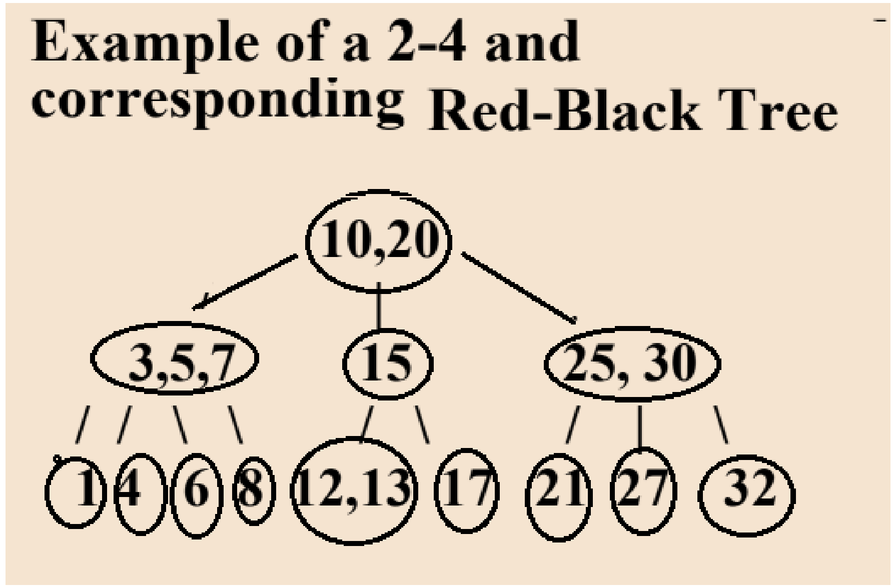
- No, it is violating property 4 (Red has a red child!)

RB tree example. Inserted numbers from 1 to 9

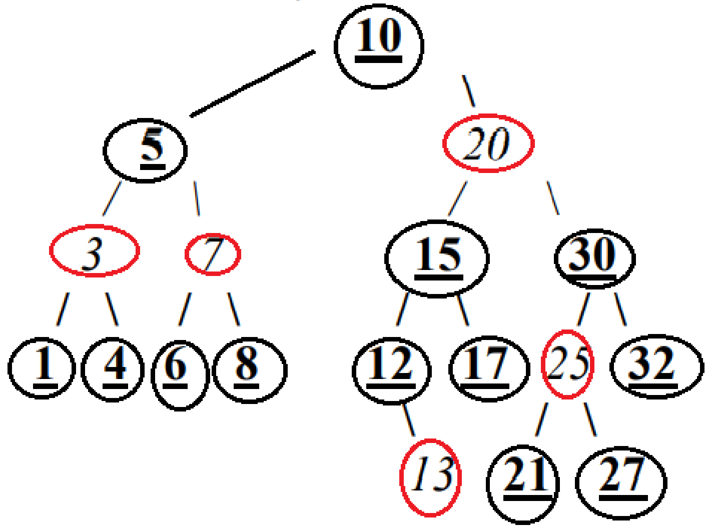


Converting a 2-4 tree into RB Tree

- Before we go on, let's go over a method to convert a 2-4 Tree into an equivalent Red-Black Tree:
  - Each 2-Node in a 2-4 Tree should be colored Black in the analogous Red-Black Tree.
  - Each 3-Node in a 2-4 Tree should be converted into two nodes in a Red-Black Tree, with the parent node being black and the child being red.
  - Each 4-Node in a 2-4 Tree should be converted into three nodes in a Red-Black Tree, with the parent node being black and the two children being red



Using the algorithm above, we have (black nodes in bold underline, red in italics):

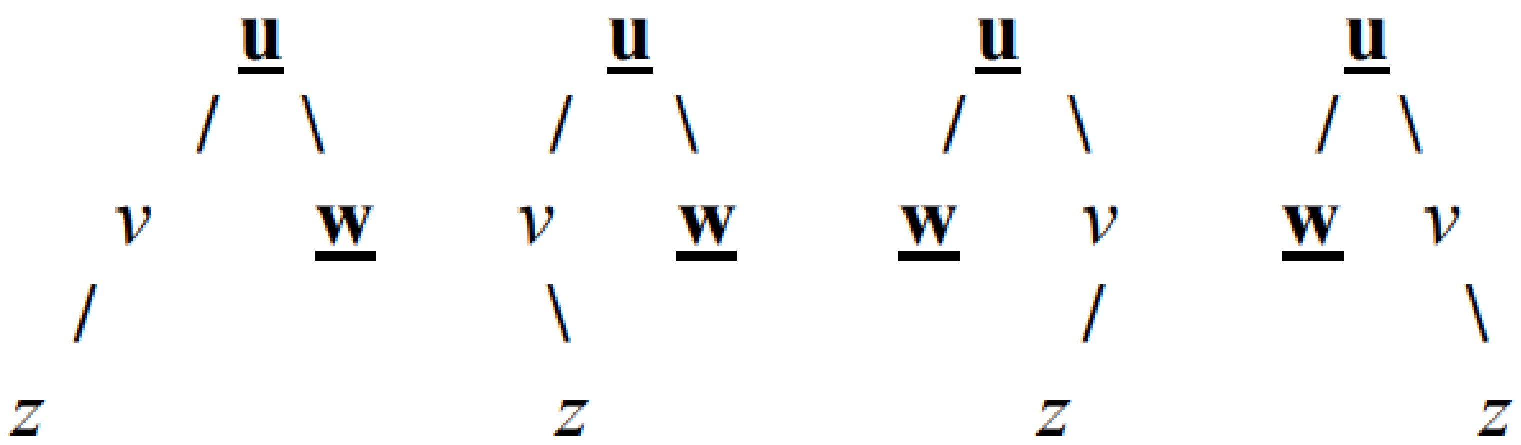


Insertion in RB Tree

- Initially, when we insert an element in a Red-Black Tree, we perform a **normal binary search tree insert** and place the node, coloring it red.
- The only exception to this is if the inserted node is the **root node.**
  - In this case, the **node is colored black.**
- Sometimes, this will cause no problem.
  - First of all, no black depths of external nodes change.
  - Next, as long as the inserted node's parent is colored black, there is no violation of the red nodes must have black children rule.
- Thus, we only have problems when the parent of the inserted node is red.
- Denote the inserted node as Z, the parent of the inserted node as **V,** and the grandparent of the inserted node as U.
- Finally, denote the **uncle of the inserted node as w.**
- We will break up our work to fix this situation into two cases:
  - 1) w is Black (Initially will be null node.)
  - 2) w is Red

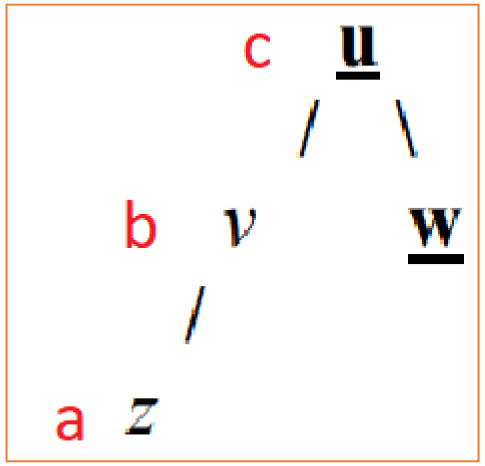
Insertion in RB Tree (Case 1)

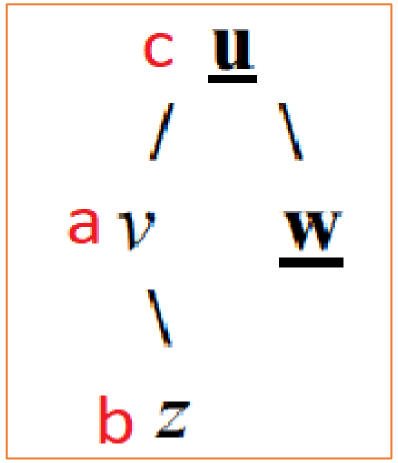
- **Case 1:** w is Black (Initially will be null node.)
- Using the same variable conventions as before, with w being a red node, recolor u, v, and w. In particular, make u red. But in doing this, we need to adhere to the rule that the children of red nodes are black. Thus, both v and w should get colored black. Luckily, this solves two problems: Now, the black depths of each of the external nodes below v and w is restored to its proper value, AND there is no double red occurrence in the subtrees rooted at v or w. ***was the issue? Z is red and parent v is also red)***
-

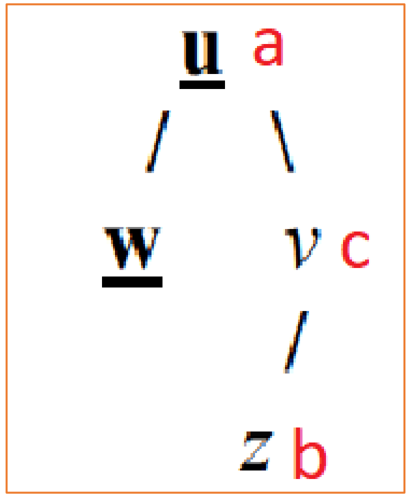


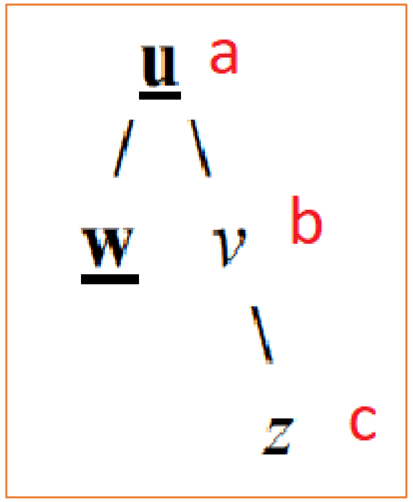
Insertion in RB Tree (Case-1)

- In each of these situations, you can relabel the nodes u, v, and z in their inorder order, a, b, and c, with a < b < c.

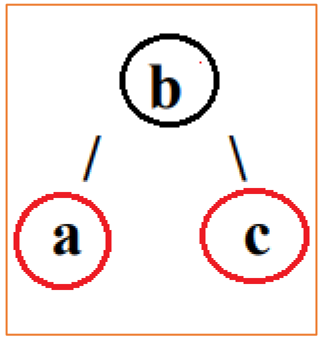






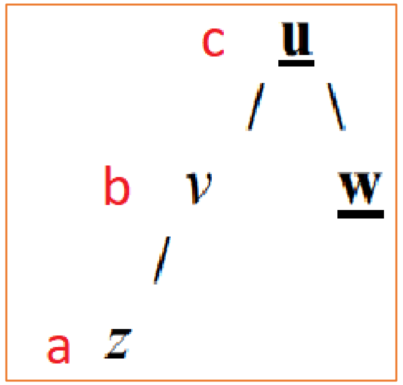


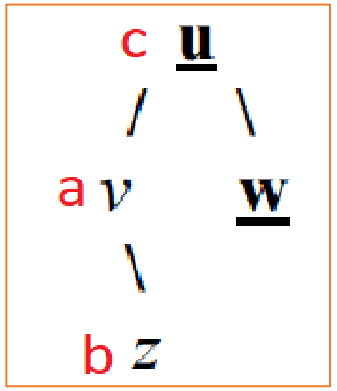
- In this case (for all four of these structures), the restructured tree will look as follows, with w inserted as a child of a or c, as appropriate:
  - **The node storing b is colored black while the nodes storing a and c are colored red.**

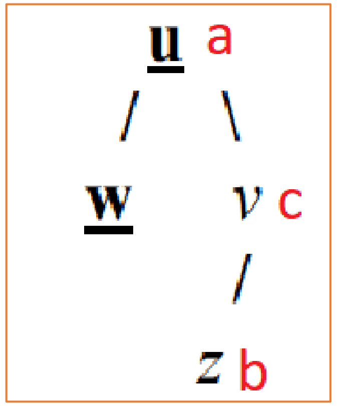


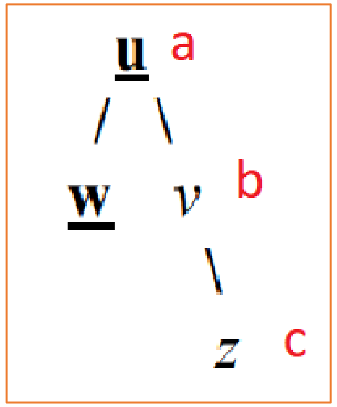
Insertion in RB Tree (Case-1)

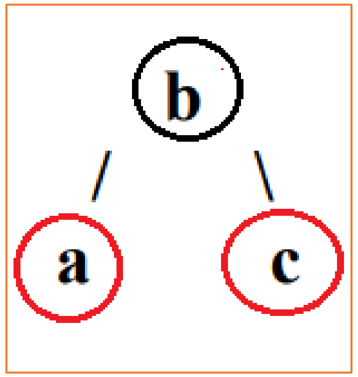
- This takes care of the double red problem.
- The node storing w will be one of the four subtrees that have either a or c as a parent.
- The reason this works is because previous to the restructuring, all four subtrees had an equal number of black nodes down any path.
- This number for each external node remains unchanged, as the "root" node of the group remains black, still adding to the black depth of each external node underneath it.











Insertion in RB Tree (Case 2)

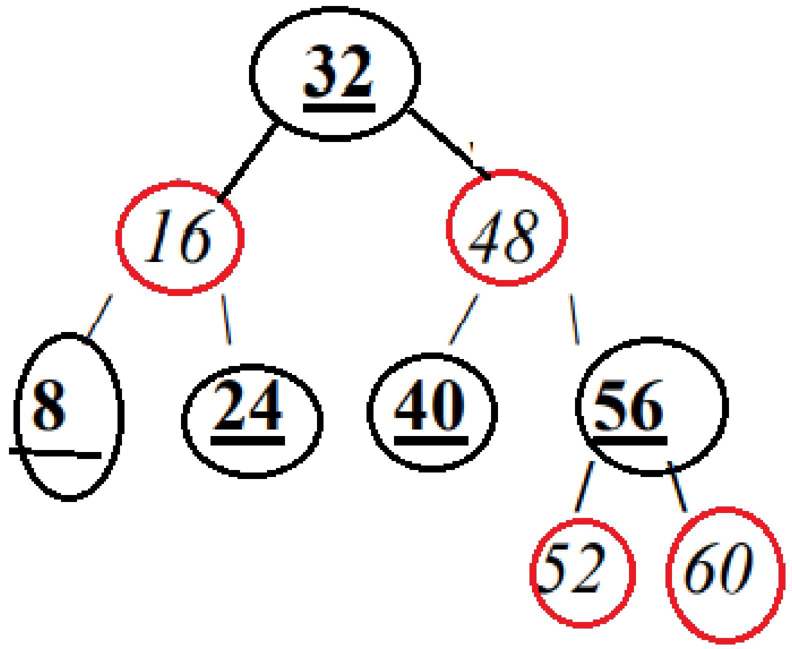
- **Case 2:** w is Red
- Using the same variable conventions as before (inserted node **Z**, the parent of the inserted node as **V**, and the grandparent of the inserted node as **U**. **uncle of the inserted node as w.),** with
- w being a red node, recolor u, v, and w.
    - In particular, **make u red.**
    - But in doing this, we need to adhere to the rule that the children of red nodes are black.
    - Thus, **both v and w should get colored black.**
    - Luckily, this solves two problems:
        - Now, the black depths of each of the external nodes below v and w is restored to its proper value,
        - AND there is no double red occurrence in the subtrees rooted at v or w.
    - But, the problem that is introduced is that u may change to become a double red node, since it was black previously, and could have had a red parent.

Insertion in RB Tree (Case 2)

- **Case 2:** w is Red
- Continuing the steps
- Now, we can deal with this issue at the node u, treating it as the node to restructure/recolor.
- As long as we continue to fall into case 2, we will just perform recolorings.
- If we ever fall into case 1, we will perform a restructuring, completing the insertion.
- (Note: if we ever follow this chain up to recoloring the root node red, we simply stop from doing that. The reason for this is that changing the color of the root node does NOT affect the relative black depth of any node in the tree.)

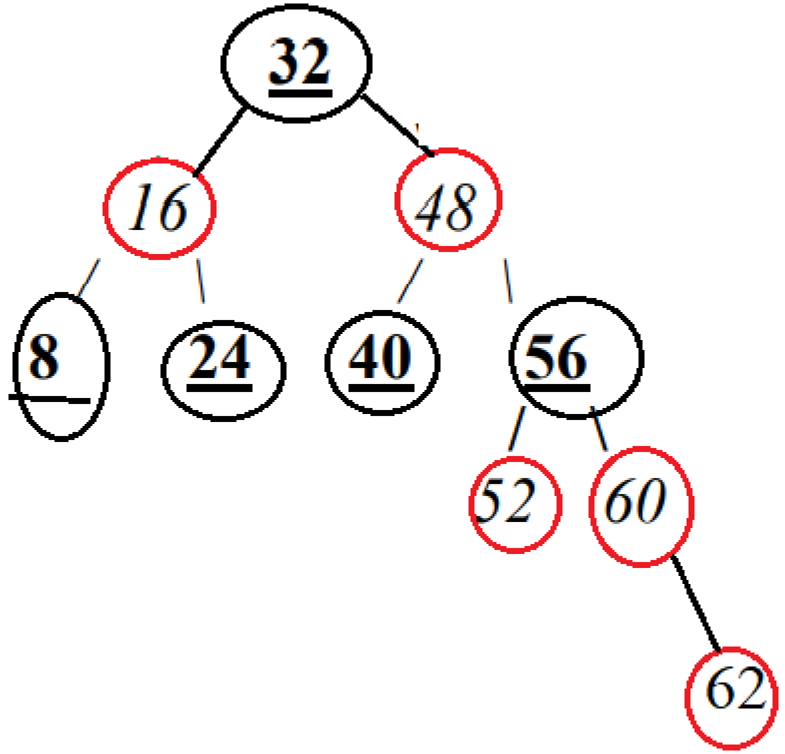
Case 2 Example

- Consider Inserting 62 into the following RB Tree



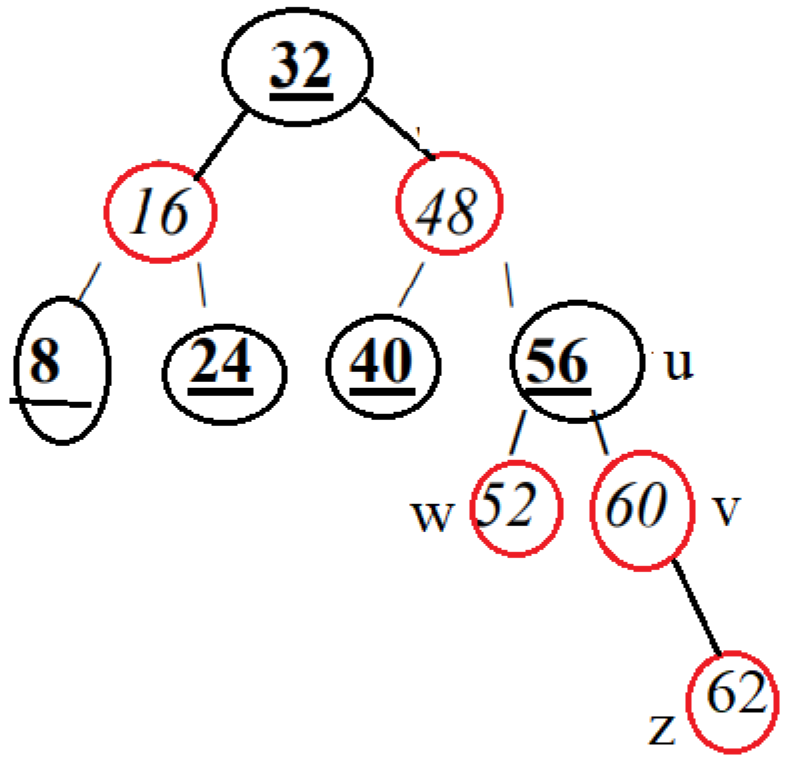
Case 2 Example

- According to the steps, insert 62 like a regular BST and color it to RED



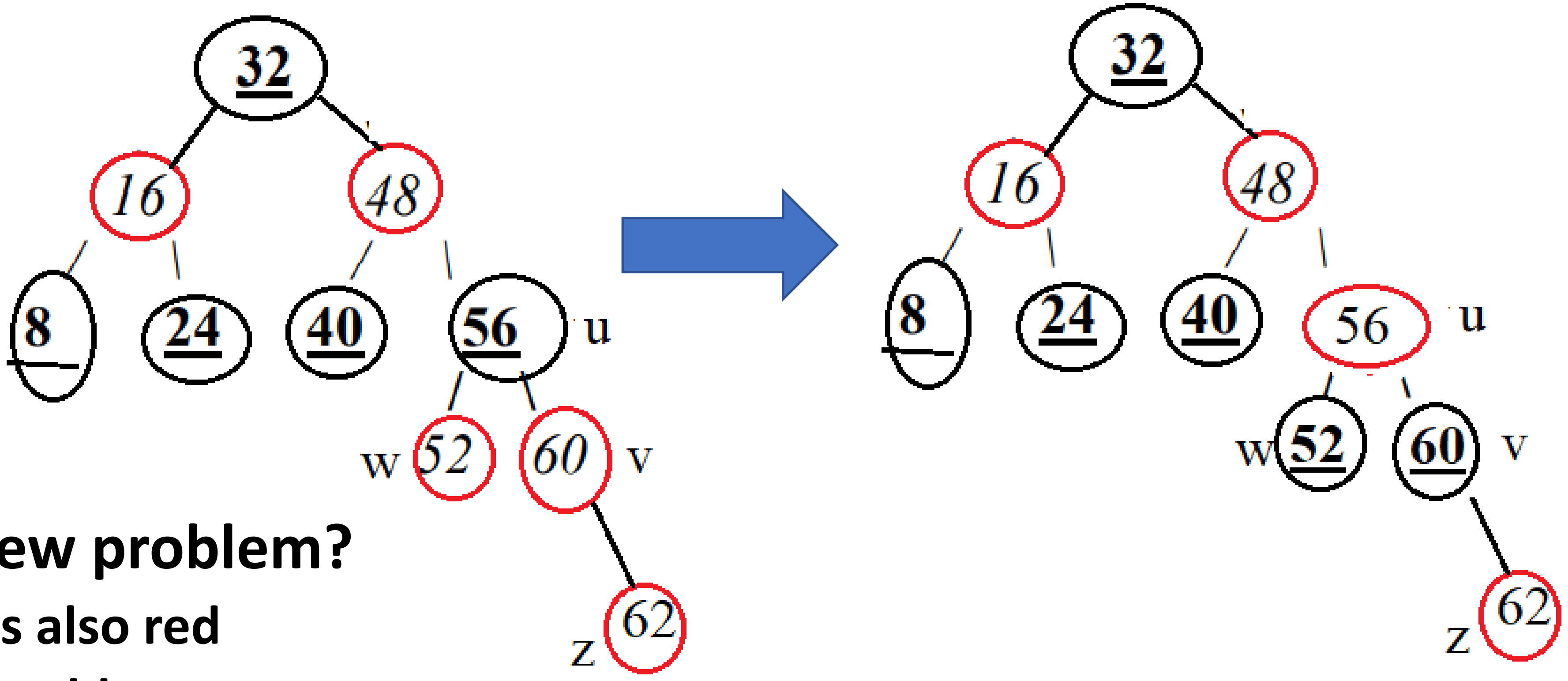
Case 2 Example

- Label them with z (new node), u (grant parent of new node), v (parent of new node), w (uncle of new node)
- **W is red, so it is in case 2**



Case 2 Example

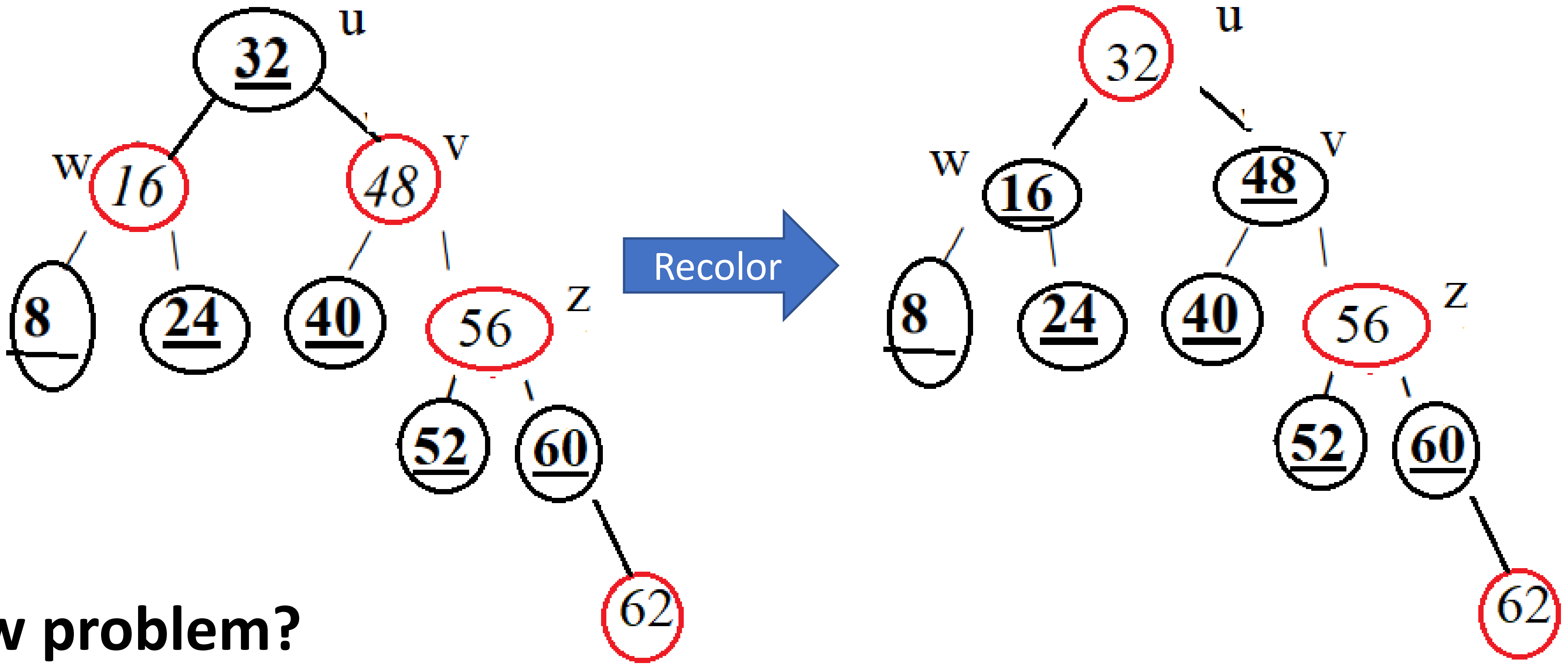
- **W is red, so it is in case 2**
- **Recolor:**
  - **u -> red,**
  - **V -> black**
  - **W -> black**



- **What is the new problem?**
  - **56’s parent is also red**
  - **Double red problem**

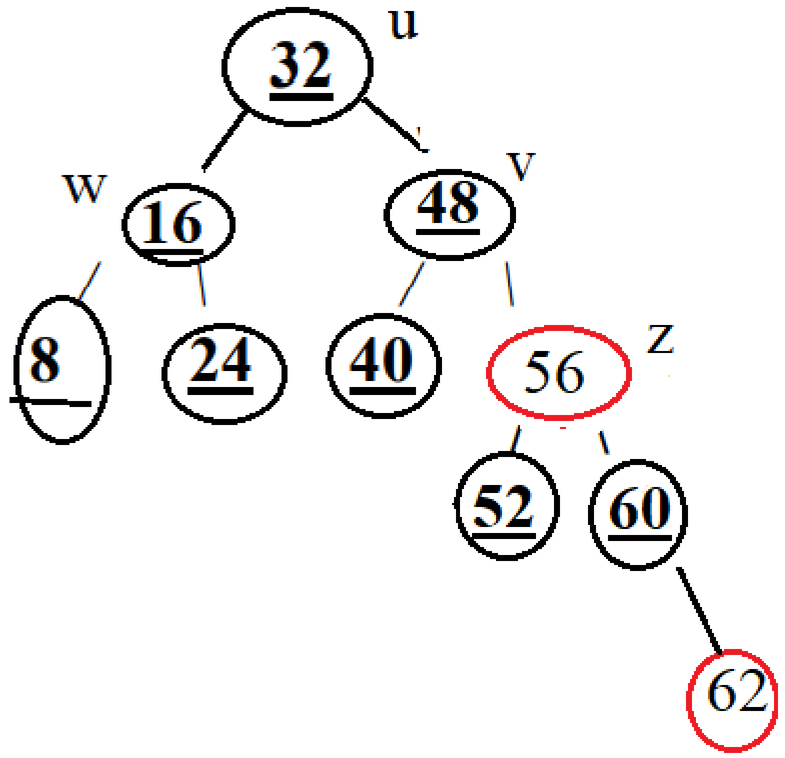
Case 2 Example

- **As we have a double red problem, we can perform another recoloring**
- **In that case, 56 will act as z, 48 ->v, 32->u, 16->w**
- **Recolor:**
  - **u -> red,**
  - **V -> black**
  - **W -> black**



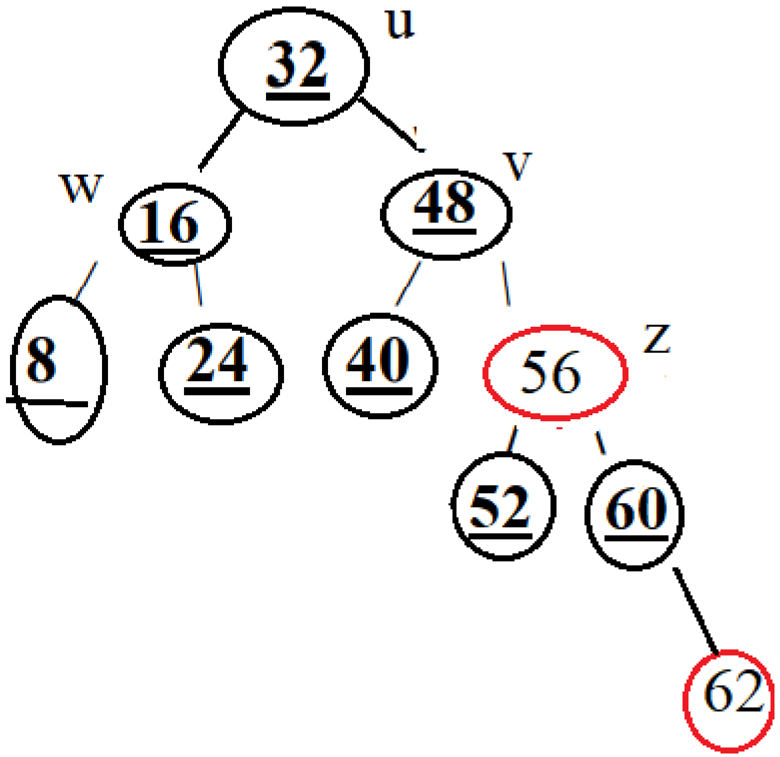
- **What is the new problem?**
  - **32 is the root and it is red!**
  - **So, we could actually simply have never changed its color to red**

Case 2 Example (final tree)



Is it an RB tree?

Relating the last example with 2-4 tree insertion



Note that, this example is using right biased approach, keeping more numbers on the right side while splitting (which is different than we have learned in our B tree notes)

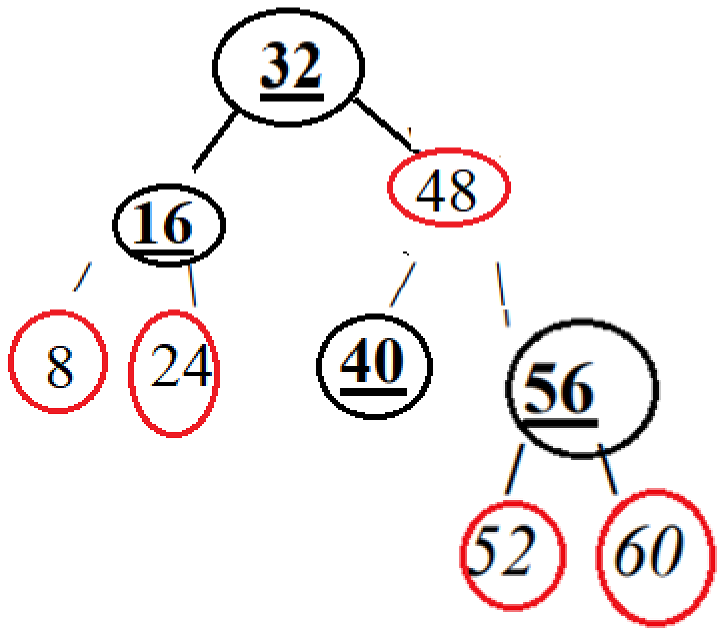
Consider the corresponding 2-4 Tree insertion:

```
          16, 32, 48
 /       /     \        \
8       24     40      52, 56, 60, 62

          16, 32, 48, 56
 /       /     \      \      \
8       24     40     52    60, 62

              32
          /          \
        16          48, 56
       /   \       /   |   \
      8    24     40   52   60, 62
```

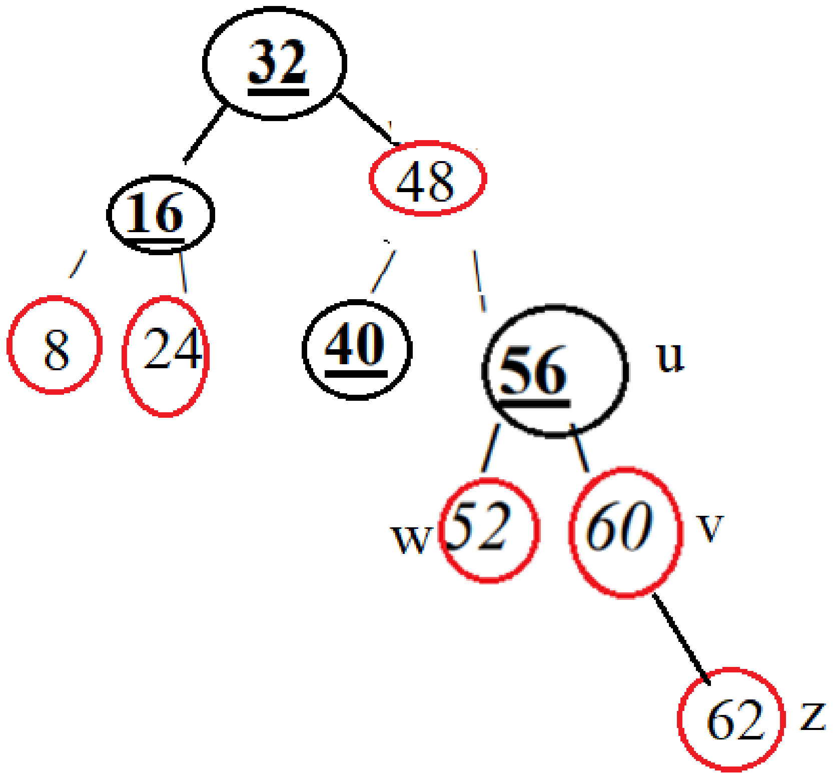
Which is the exact same tree you get when you convert the result Red-Black Tree into a 2-4 Tree. (The recolorings are equivalent to pushing a node up to a parent node in a 2-4 Tree.)

Case 1 Example

- Now consider the same example, except where 16 is black, and both 8 and 24 are red. We want to insert 62



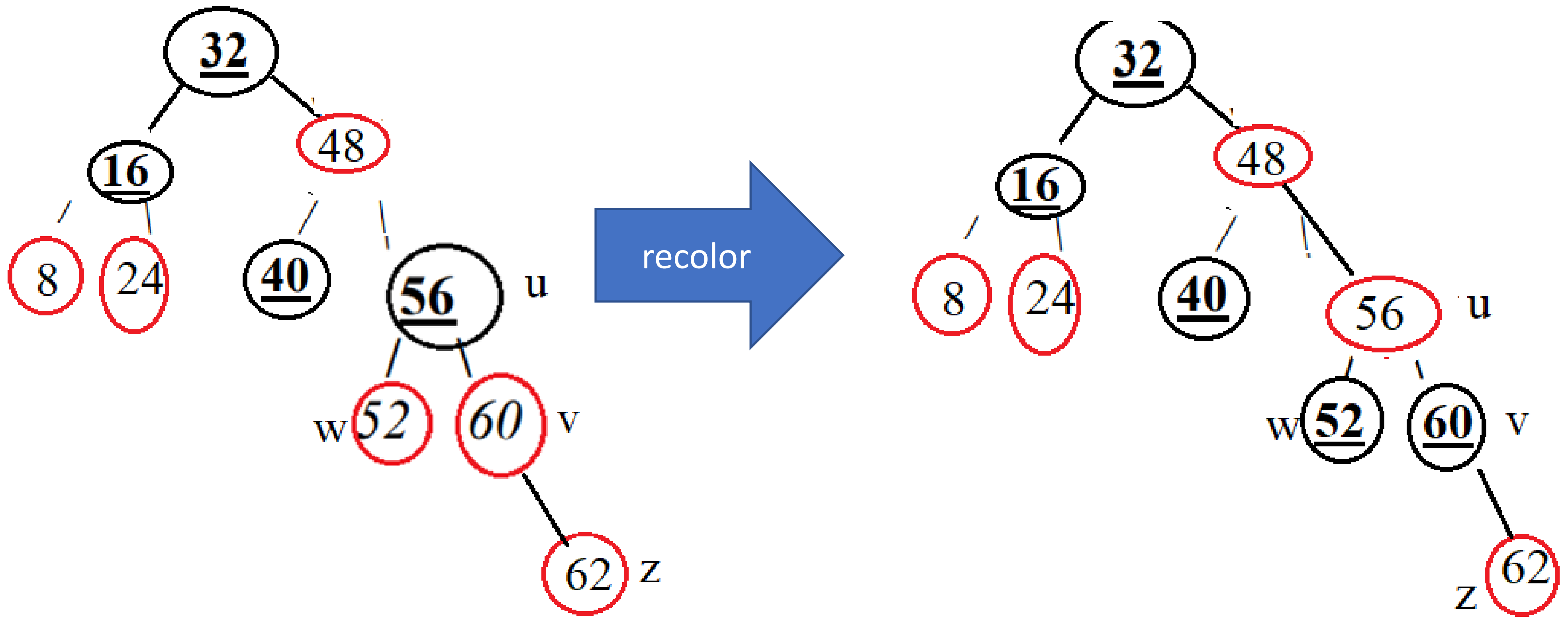
Insert 62, so far this is Case 2 (w is red)



- So far it is in case 2 as w is red. So, let's recolor

Case 1 Example

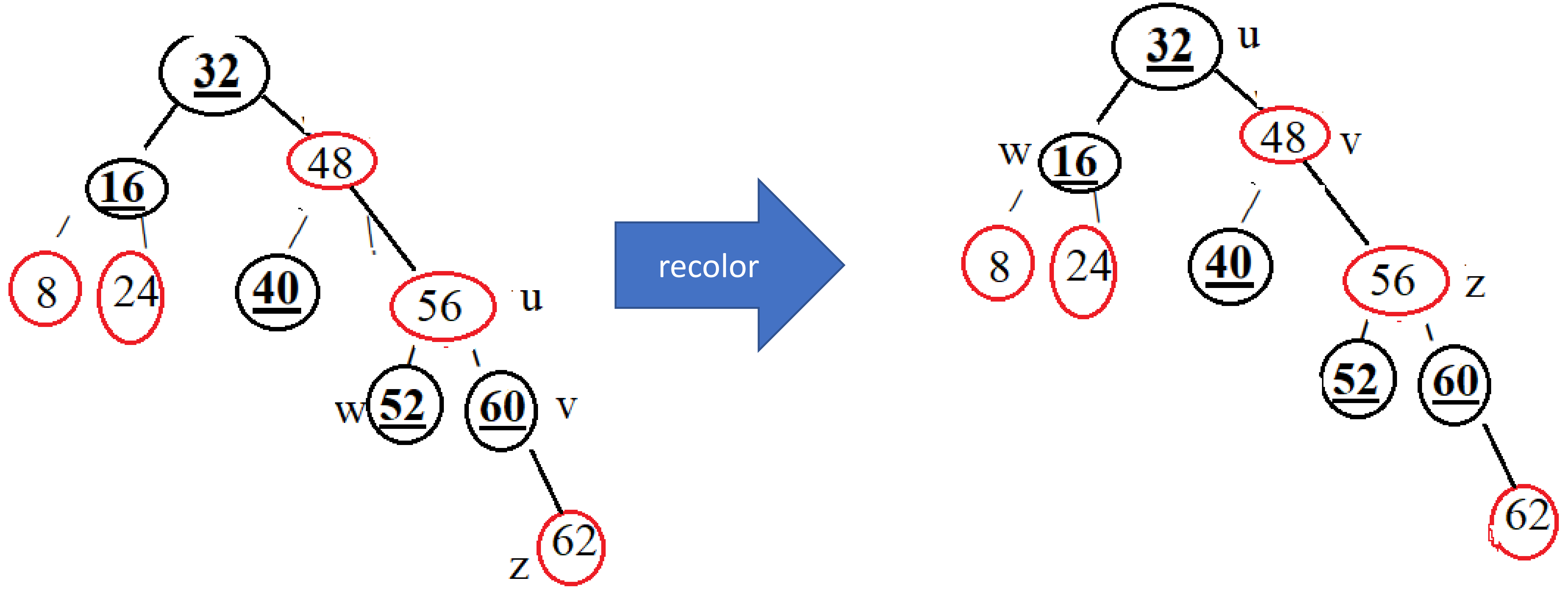
- So far it is in case 2 as w is red. So, let’s recolor



- Now, we have red-red problem (see, 56’s parent 48 is also red)
- Let us update our labels

Case 1 Example

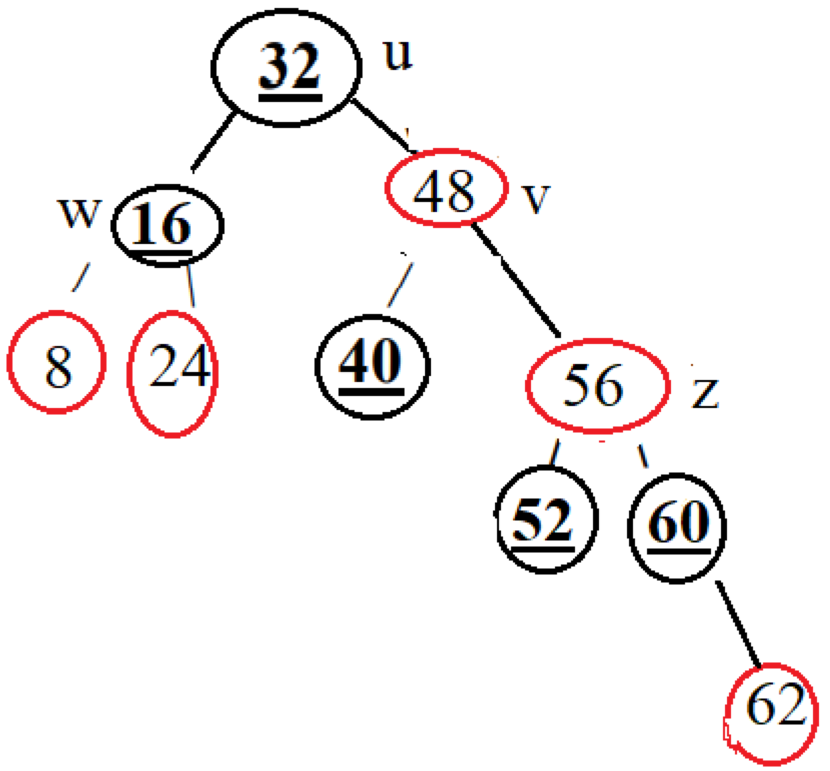
- So far it is in case 2 as w is red. So, let’s recolor

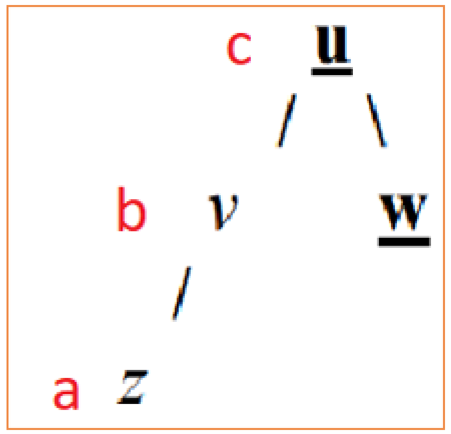


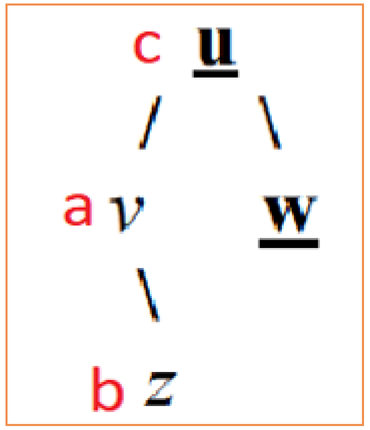
- Now, w is black!
- **So, this part is now in case 1**

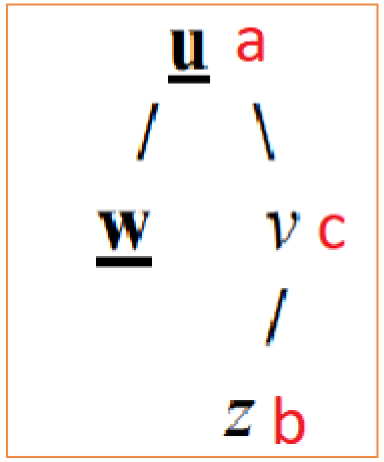
Case 1 Example

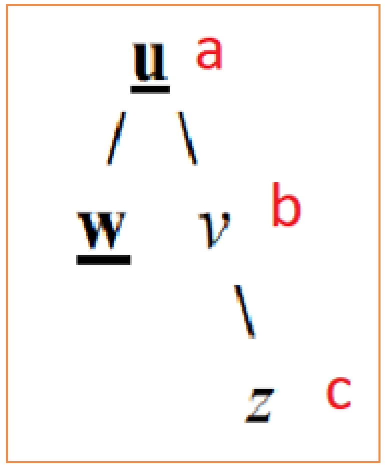
- Now, w is black!
- **So, this part is now in case 1**
- **So, we need to relabel u, v, and z with a, b, c based on a<b<c**





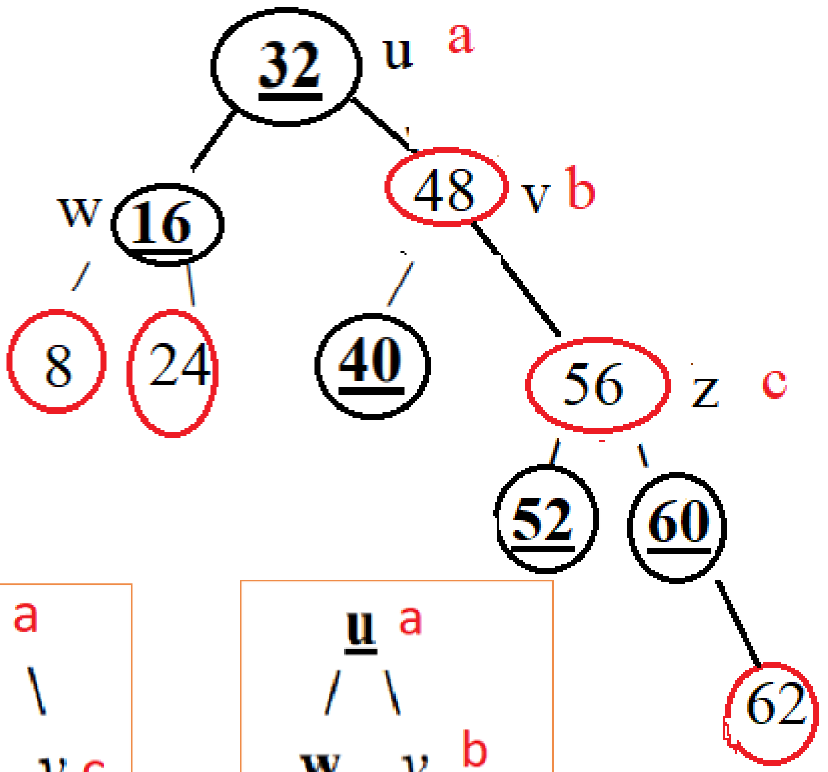


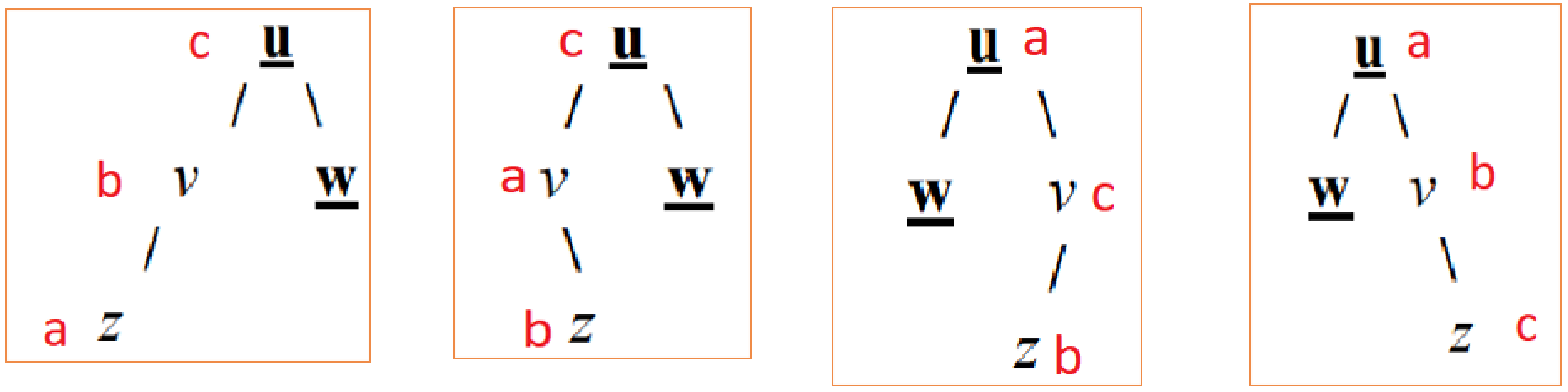


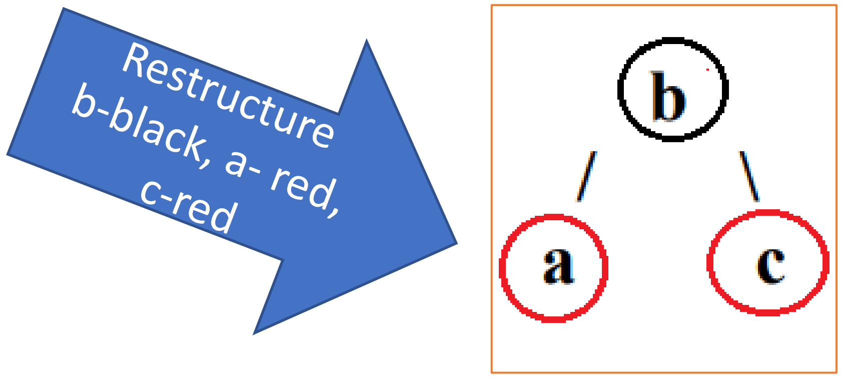


Case 1 Example

- Now, w is black!
- **So, this part is now in case 1**
- **So, we need to relabel u, v, and z with a, b, c based on a<b<c**



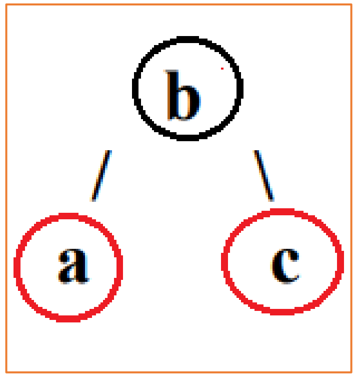




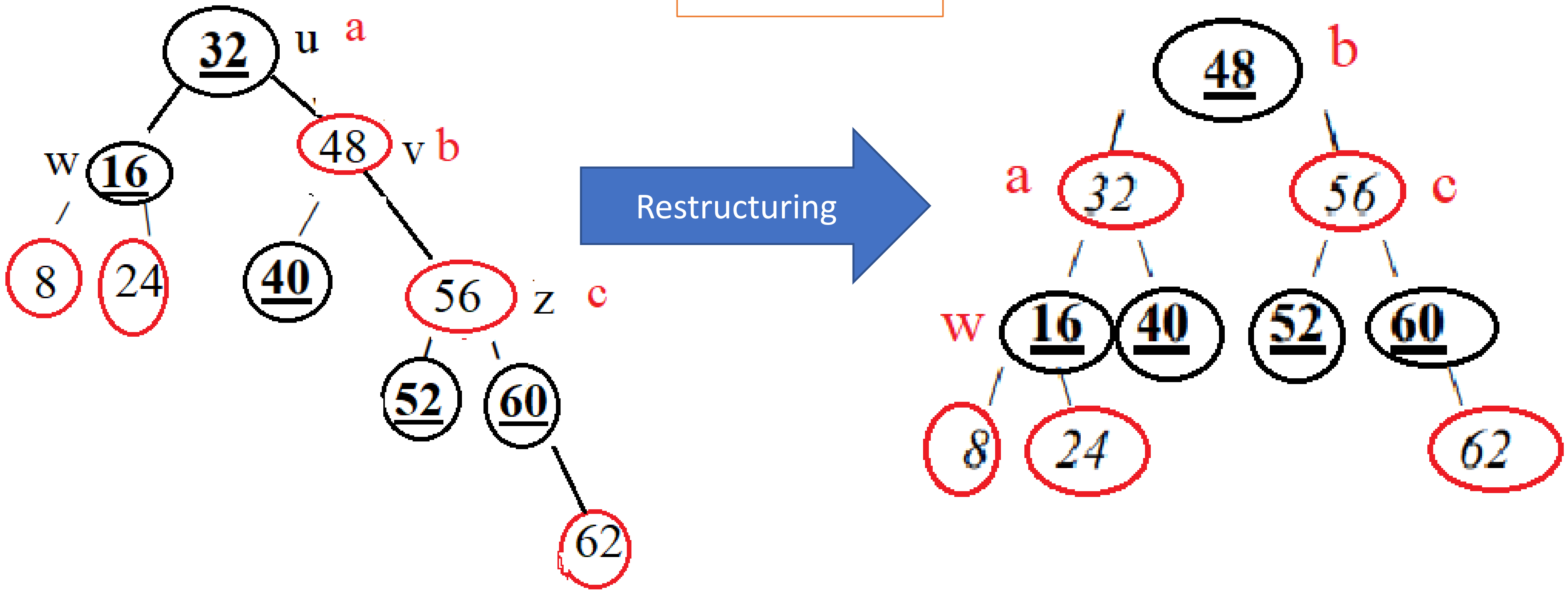
W will be the child of either a and c according to its value

Case 1 Example

- **After restructuring**

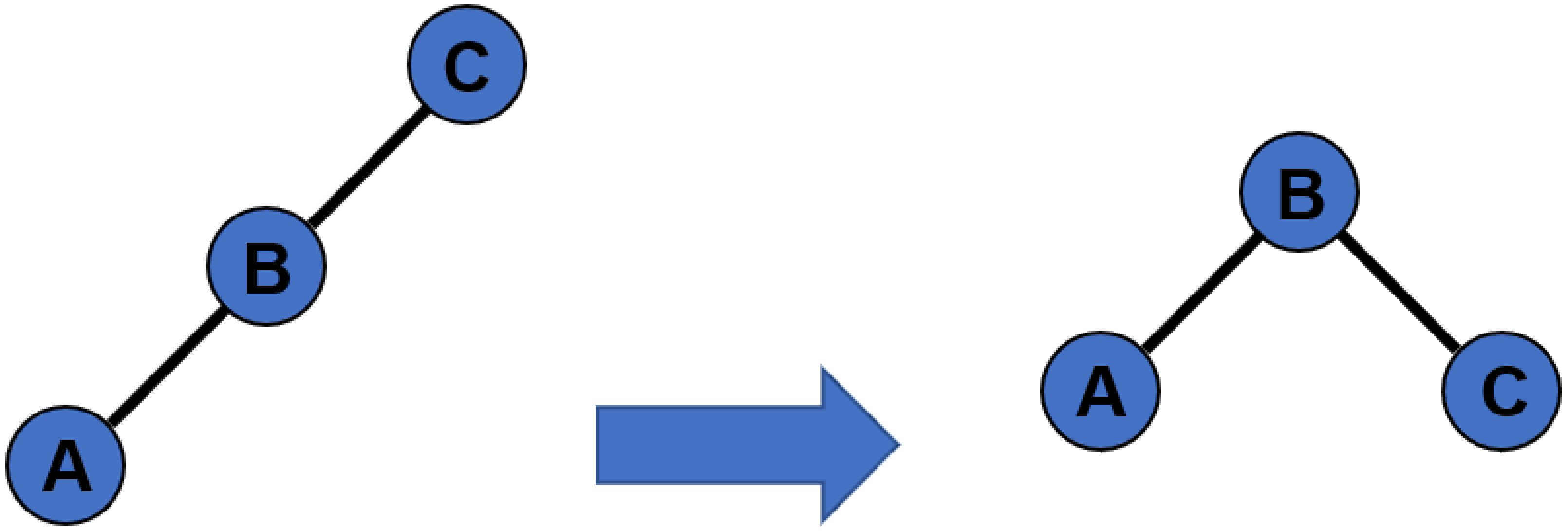


b is black, a is red, c is red, w will be the child of either a and c according to its value



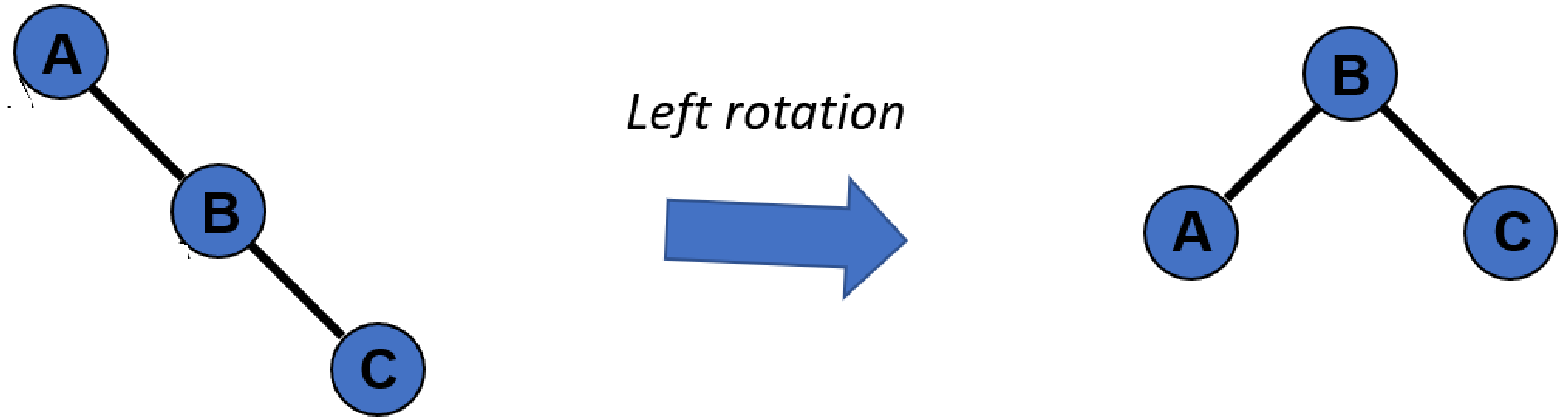
Are we doing any rotation?

- We are actually doing rotations (implicitly)
- For the following restructuring, we are doing one right rotation



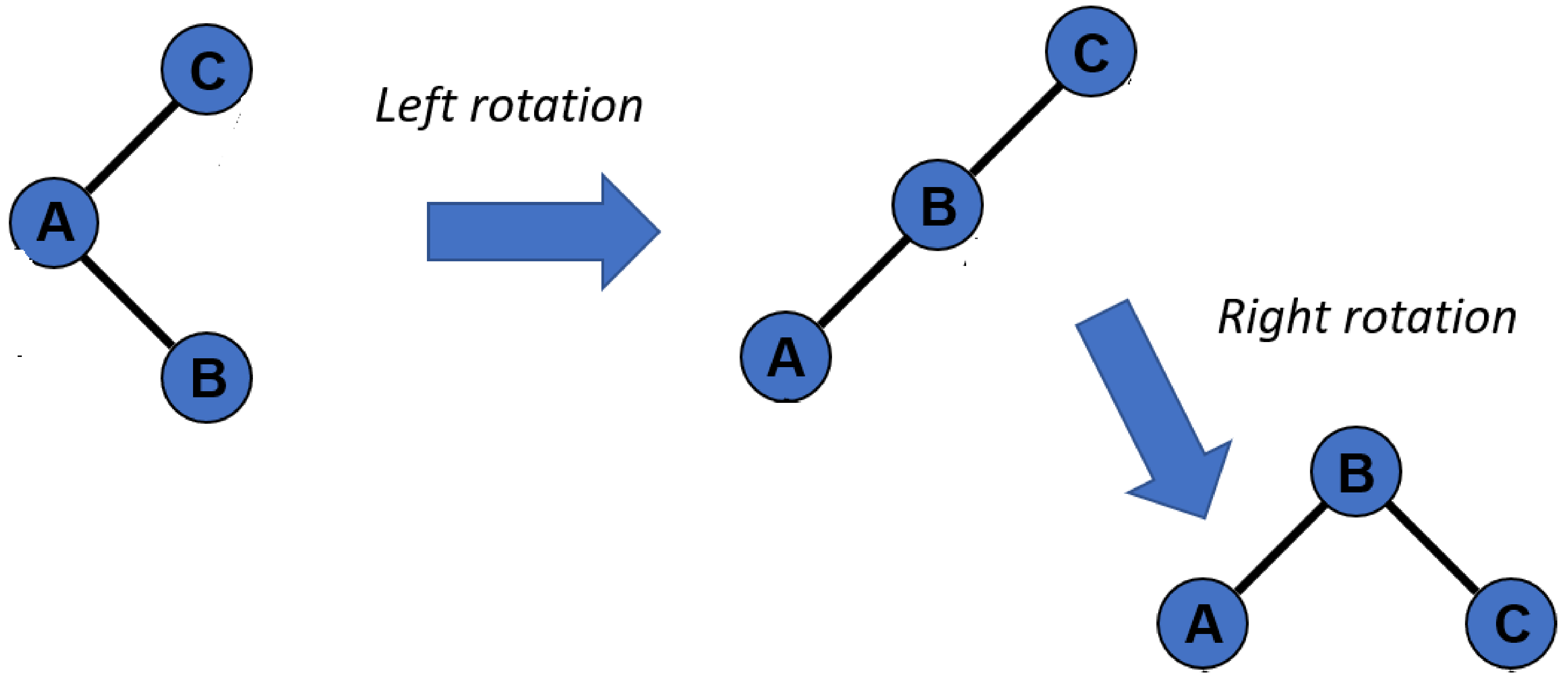
Are we doing any rotation?

- We are actually doing rotations **(implicitly)**
- For the following restructuring, we are doing one left rotation



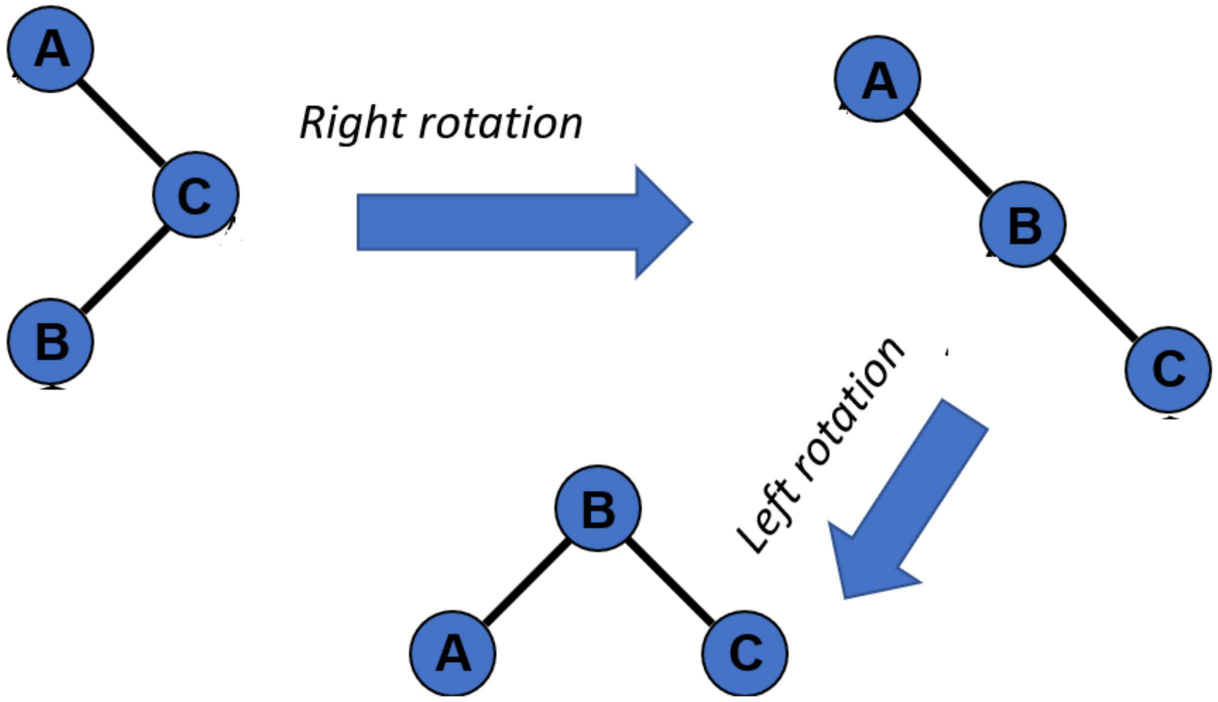
Are we doing any rotation?

- We are actually doing rotations (implicitly)
- For the following restructuring, we are doing 2 rotations



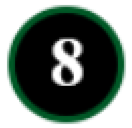
Are we doing any rotation?

- We are actually doing rotations (implicitly)
- For the following restructuring, we are doing 2 rotations



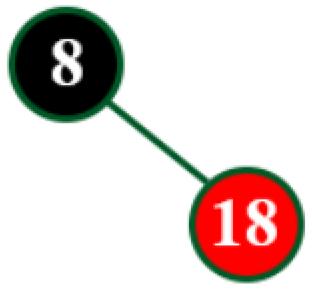
A complete Example

- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, 25, 40 & 80
- Key: 8
- The tree is empty. So, insert 8 in the root node with black color



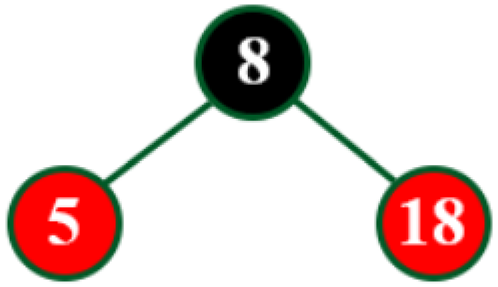
A complete Example

- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, 25, 40 & 80
- Key: 18
- Insert new node with the red color (As the tree is not empty)



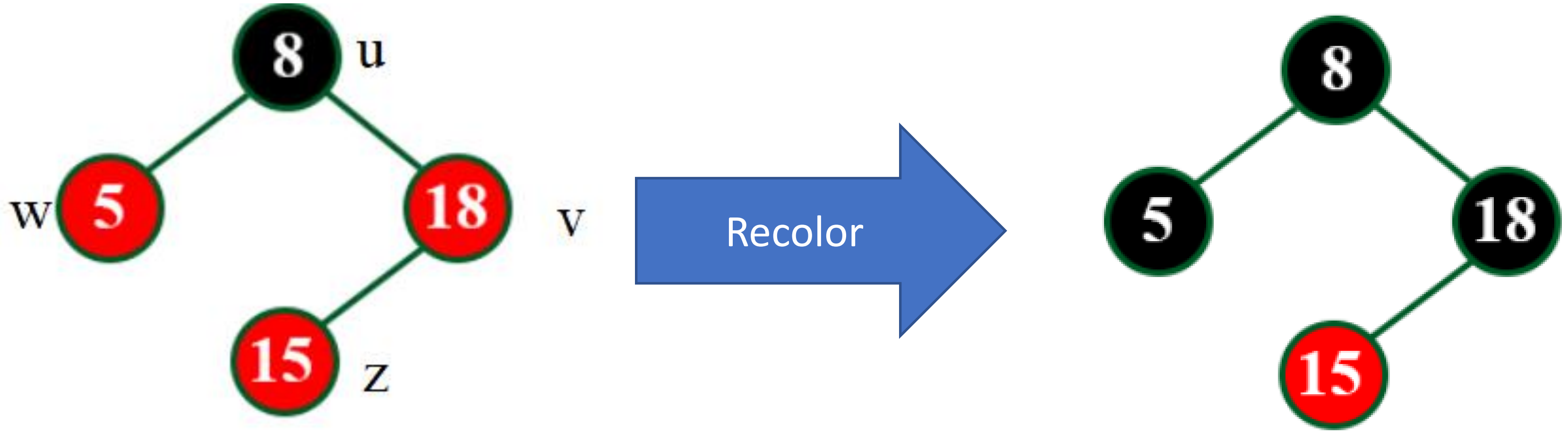
A complete Example

- Let us create red-black tree with the following keys:
- 8, 18, **5,** 15, 17, 25, 40 & 80
- Key**: 5**
- Insert new node with the red color (As the tree is not empty)



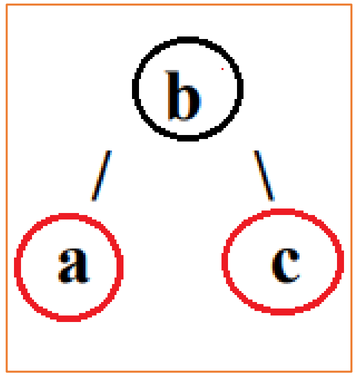
A complete Example

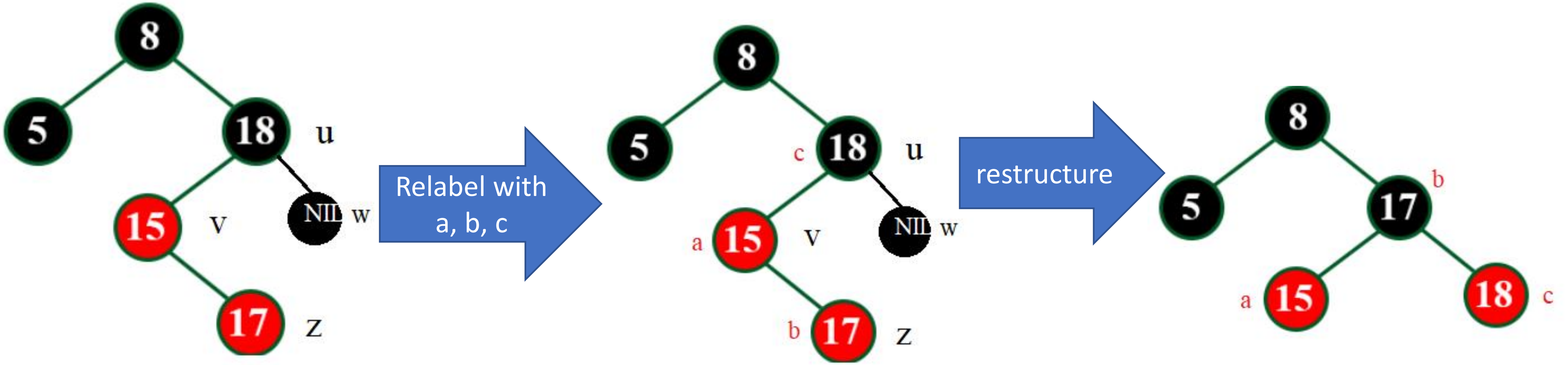
- Let us create red-black tree with the following keys:
- 8, 18, 5, **15**, 17, 25, 40 & 80
- Key: **15**
- Insert new node with the red color (As the tree is not empty). We have a red-red problem. Label them with z, v, u, w. The uncle (w) is red**. So, case 2**. So, recolor them. As 8 is root, it will remain black



A complete Example

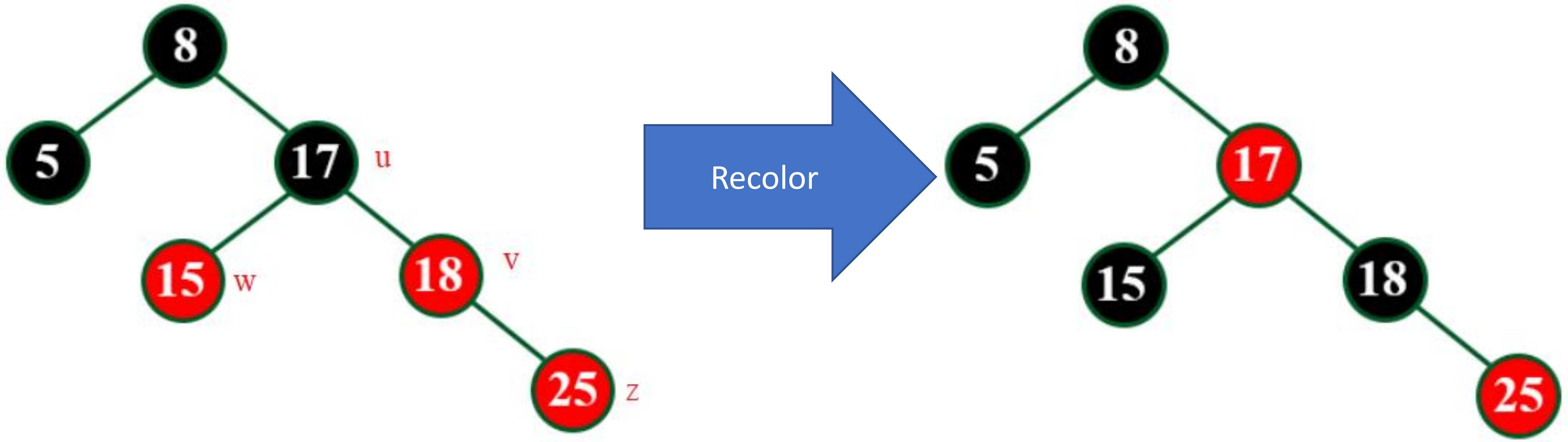
- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, 25, 40 & 80
- Key: 17
- Insert new node with the red color (As the tree is not empty).
- We have red-red problem. So, label z, v, u, and w. Uncle (w) is black (NILL nodes are black by default. **<u>It is in case 1</u>**. So, relabel them with a, b, c





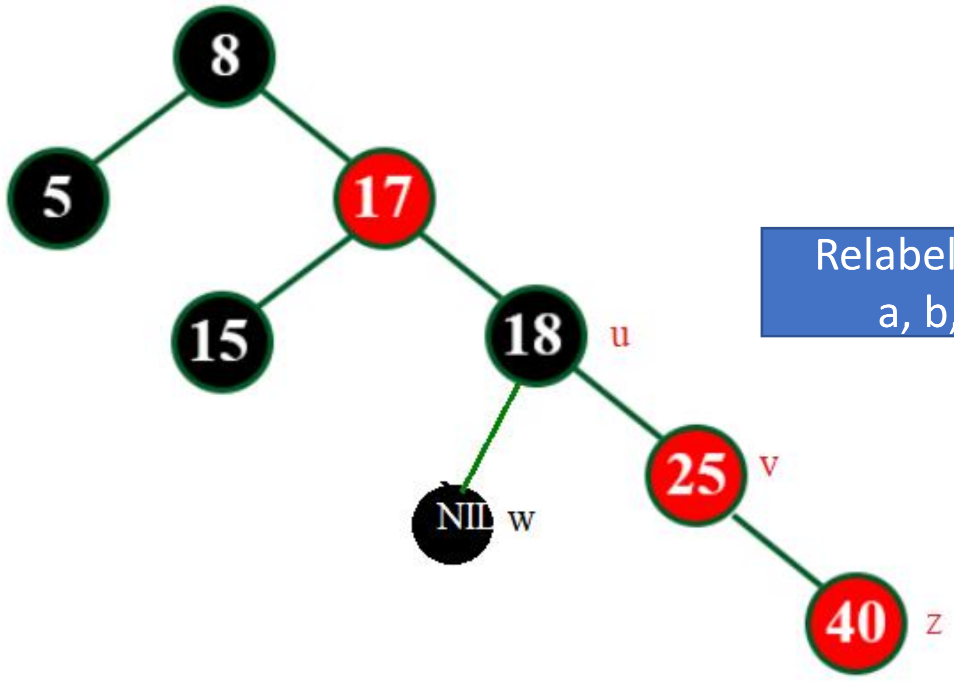
A complete Example

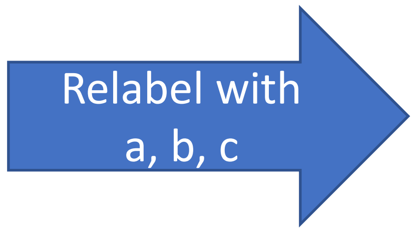
- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, **25**, 40 & 80
- Key: **25**
- Insert new node with the red color (As the tree is not empty). We have a red-red problem. Label them with z, v, u, w. The uncle (w) is red**. So, case 2**. So, recolor them. After recoloring, the properties are maintained, and no further action is needed

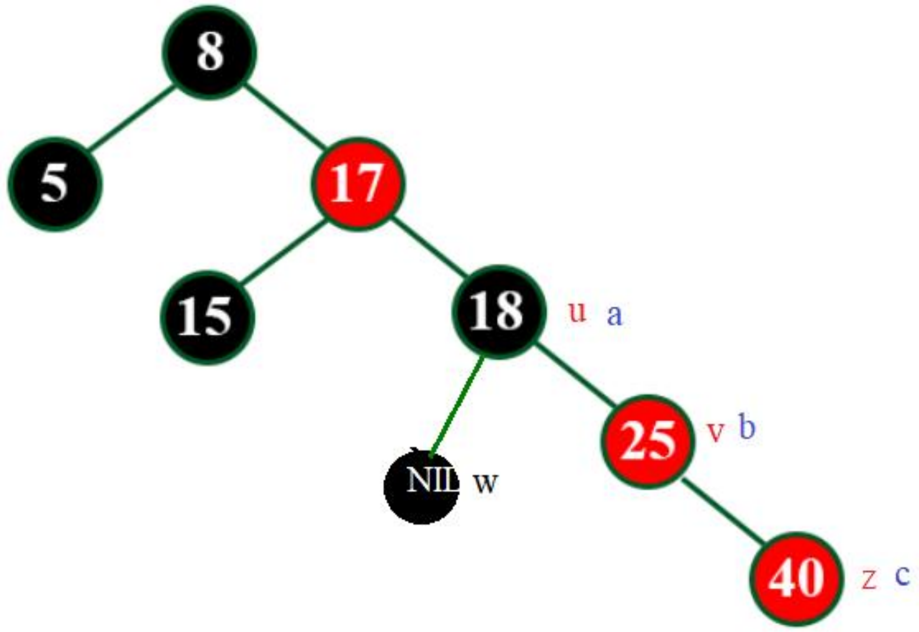


A complete Example

- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, 25, **40** & 80
- Key: 40 (Case 1), so relabel with a, b, c and restructure

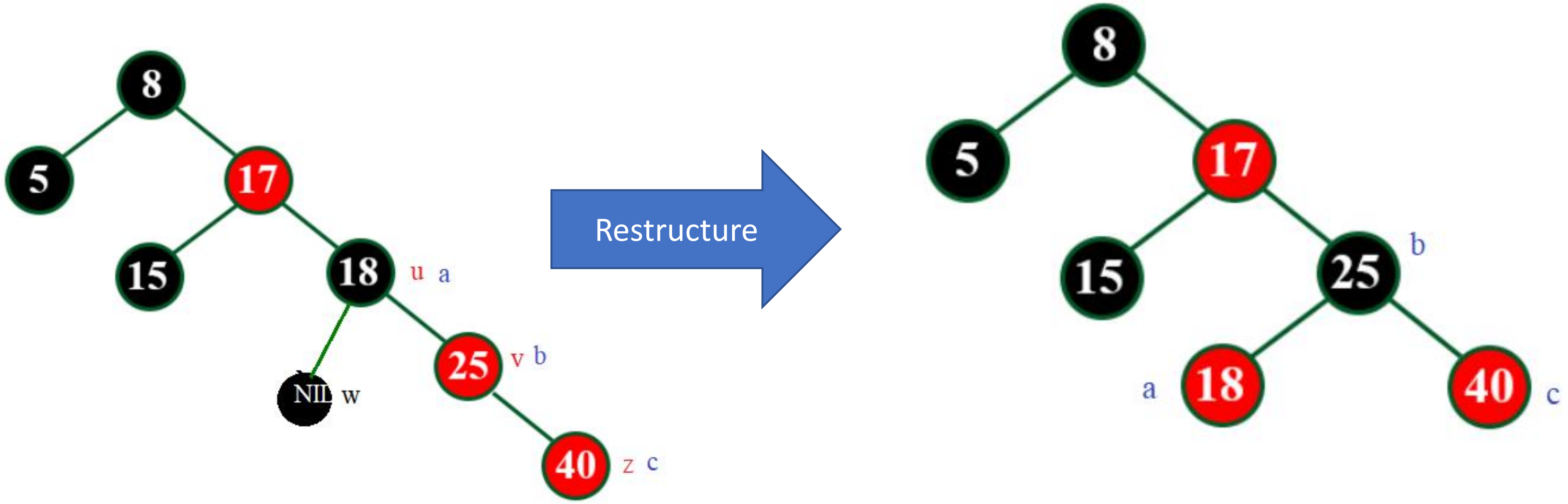






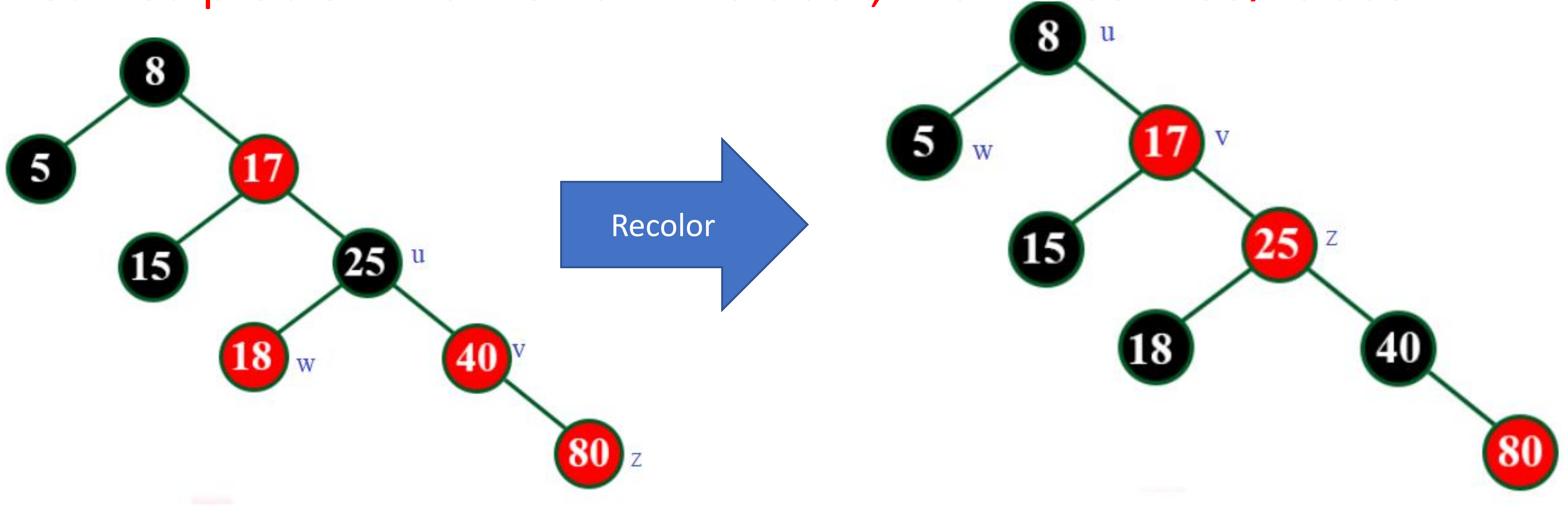
A complete Example

- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, 25, **40** & 80
- Key: 40 (continuing from last slide). After restructuring based on a, b,c, the tree is in RB tree and no further action is needed

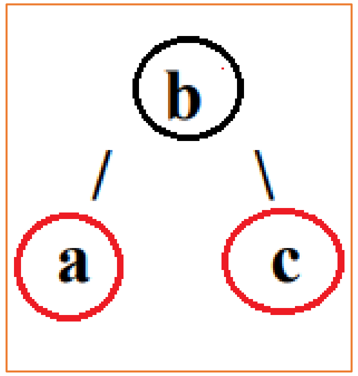


A complete Example

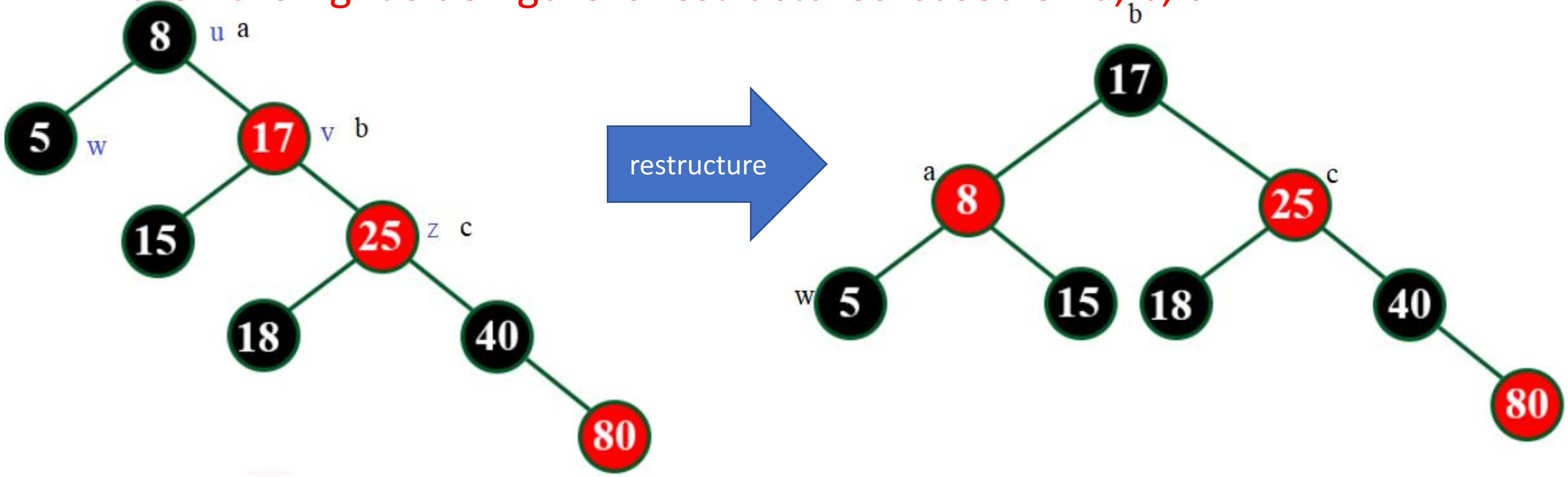
- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, 25, 40 & 80
- Key: 80 (Case 2. So, recolor). However, after re-coloring, we still have Red-Red problem. As the new w is black, it is in case 1. So, relabel.



A complete Example



- Let us create red-black tree with the following keys:
- 8, 18, 5, 15, 17, 25, 40 & 80
- Key: 80 continue…. The left side figure was relabeled with a, b, c and then the right side figure is restructured based on a, b, c

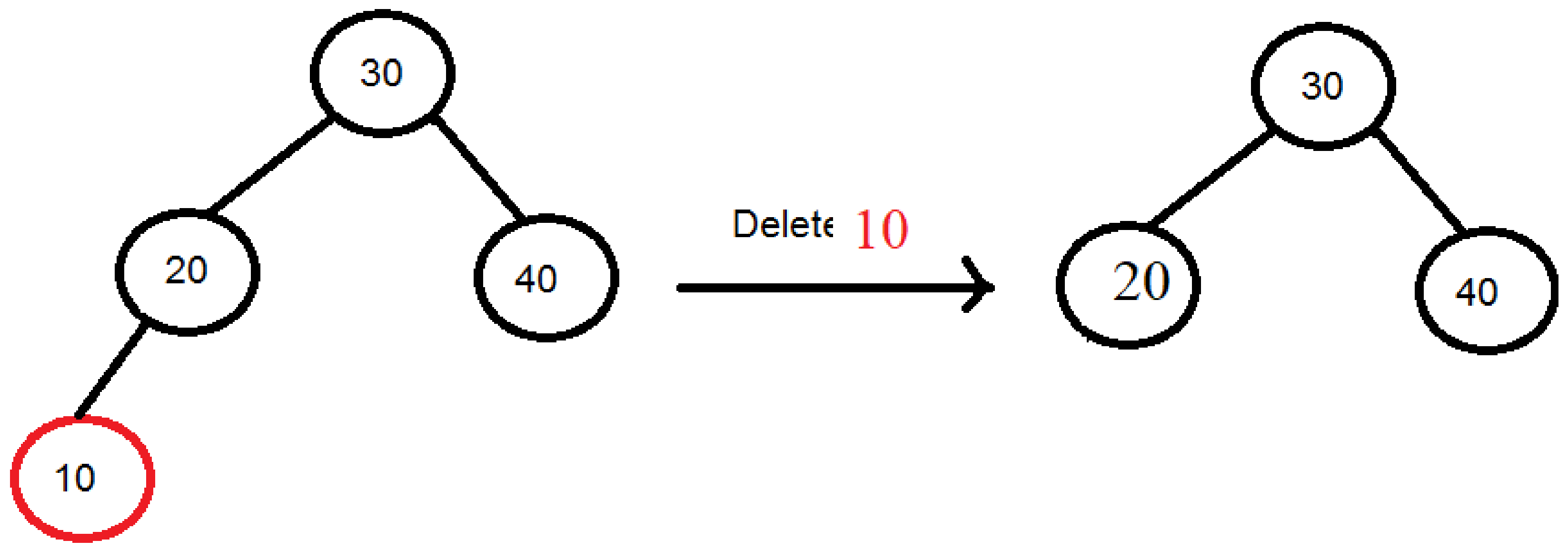


Deletion in an RB Tree

- Initially, we will delete a node just like we delete a node in a normal binary search tree.
- Here are the cases we will look at:
- 1) Red leaf node
- 2) **Black** node, with one red child node
- 3) **Black** leaf node
- Before we discuss how to deal with "double black" nodes, let's real quickly justify why the cases above are the only cases we will deal with. First off, we will only delete nodes with 0 or 1 child (in BST a node with 2 children is simplified to delete a node with 0 or 1 child). Neither colored node can have one black child. If it did, the black height of the node's null child would not be proper. Further more, a red node can NOT have a red child. These observations narrow the cases to the situations listed above.

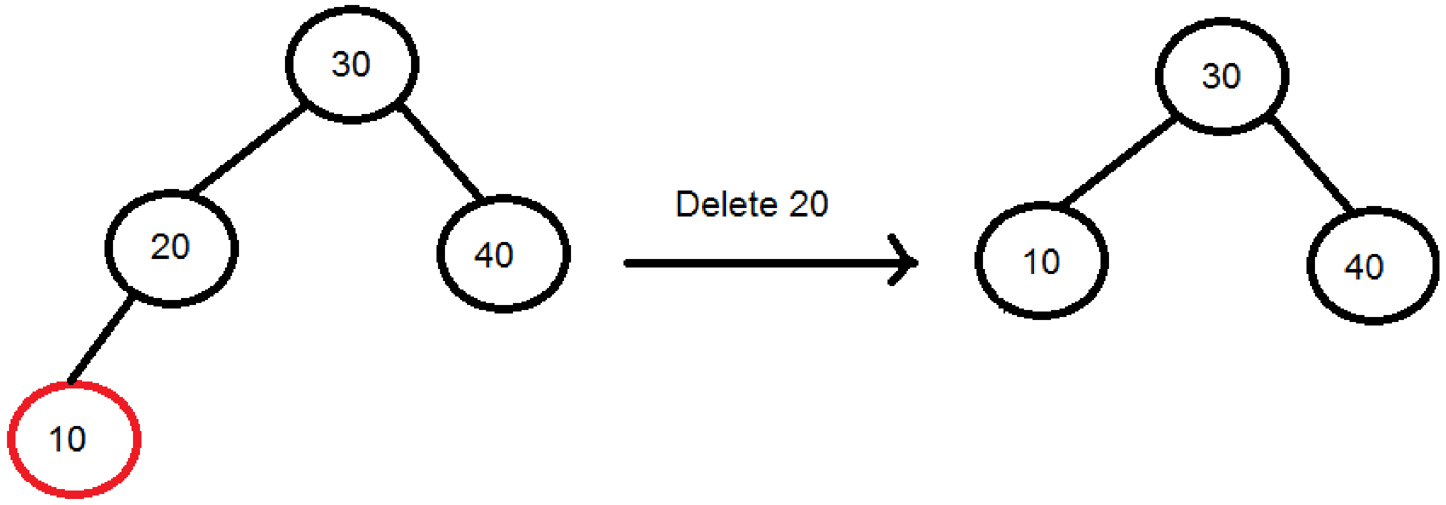
Deletion in an RB Tree

- 1)Deleting Red leaf node:
- In the first case, the normal binary search tree delete is sufficient.
- Removing a red node does not change the black depths of any node, nor create a red child for any red node.



Deletion in an RB Tree

- 2) **Black node, with one red child node**
- In the second case, after we complete the binary search tree delete, we must simply recolor the child node of the deleted node to black.
- Changing this color adds one to the black depths of each node in the subtree of the deleted node, restoring the equality of the black depths of all external nodes.
- Also, changing a node to black does not violate any of the other RedBlack Tree specifications.



Deletion in an RB Tree

- **3) Black leaf node**
- Neither of the strategies mentioned above are adequate for dealing with the third case.
- Instead, when we patch in the child of the deleted node in this case, in order to temporarily preserve the black depth property, we will color this child node a fictitious **"double black"** color.
- (Remember, in red black tree all the external NIL nodes are also black). So, in this case, that NILL node will be considered as **fictitious double black node**

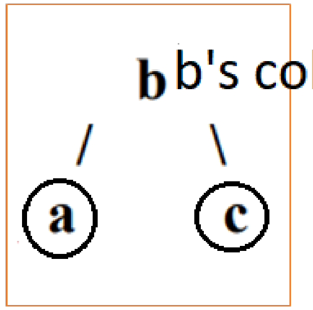
- The remaining cases with this **"double black"** node can be categorized as follows:
- 3.1) The sibling of the "double black" node is black and has a red child.
- 3.2) The sibling of the "double black" node is black and both children are black.
- 3.3) The sibling of the "double black" node is red.
- (Note that initially, even though the double black node is a null node, after starting the recoloring/restructuring process, we may create a double black node that is NOT null.)

Deletion in an RB Tree: Dealing with case 3

- **Case 3) Black leaf node**
- To deal with each of these situations, let's first set up names for all of the important nodes:
- 1) Let the child of the deleted node, which is colored "double black" be ***r.***
- 2) Let ***y*** be the sibling of ***r.***
- 3) Let ***z*** be a child of ***y***, in each case, the specific child will be designated.
- 4) Let ***x*** be the parent of ***y.***

Deletion in an RB Tree: Dealing with case 3.1

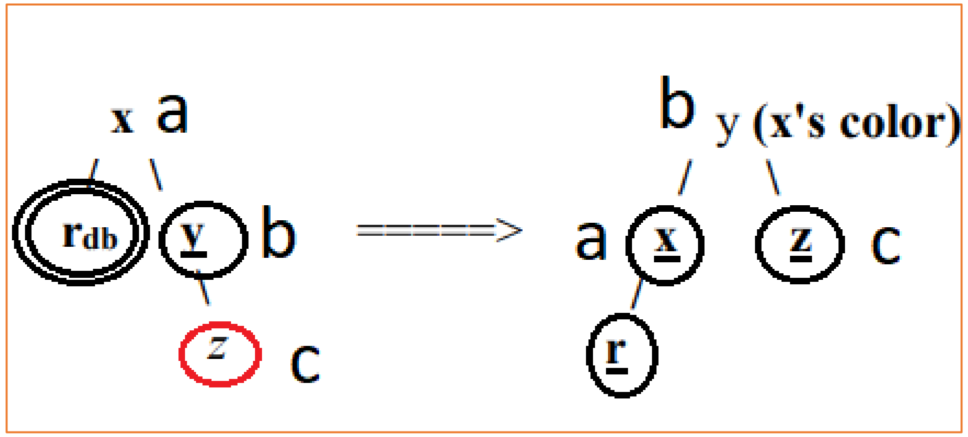
- **Case 3.1: y is black and has a red child z. [remember, y is the sibling of r-the double black node]**
- Take the nodes, x, y, and z and relabel them a, b, and c, in their inorder ordering. (like we did during insertion steps)
- Place b where x used to be, and then have a and c be the left and right children of x, respectively.
- Color a and c black, and color b whatever color x USED to be.
  - This eliminates the "double black" problem, so we can stop here

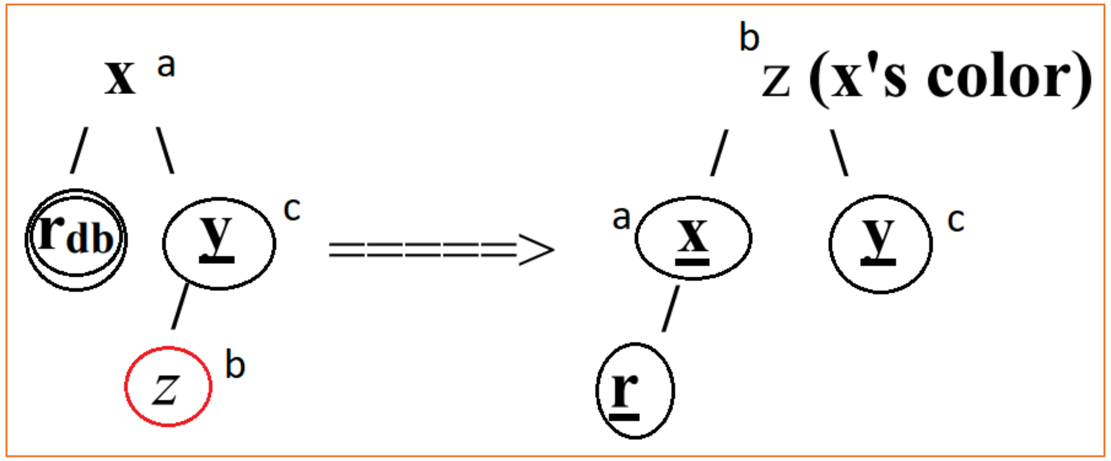


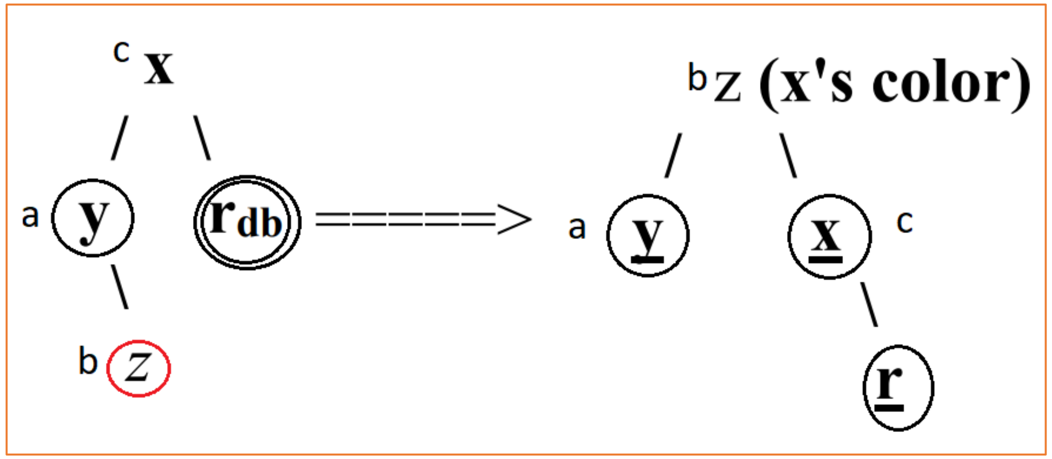
b's color depend on x

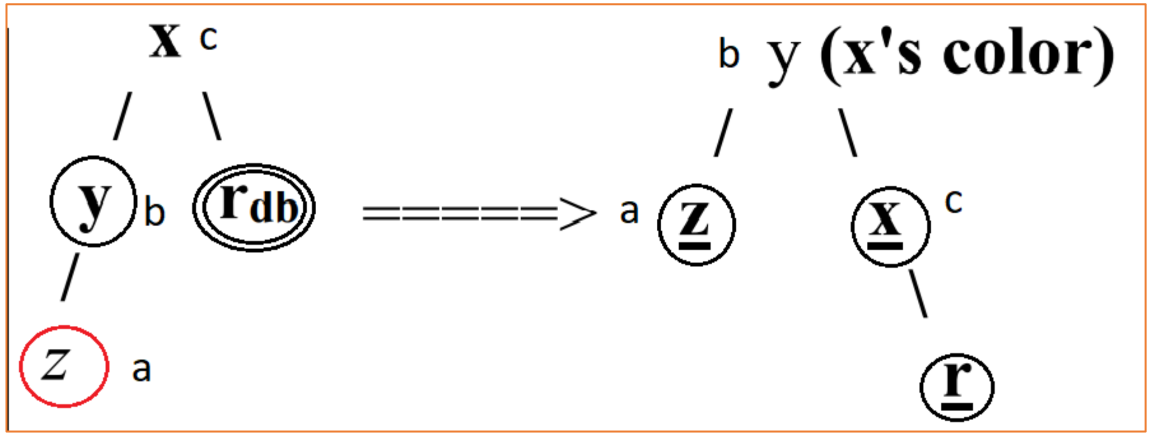
Deletion in an RB Tree: Dealing with case 3.1

- **<u>Case 3.1: y is black and has a red child z.</u>**
- **Similar to insertion, we can have the four different cases of labeling a, b, c**









The above steps eliminates the double black node, but maintain the "black depth" of each external node in the tree

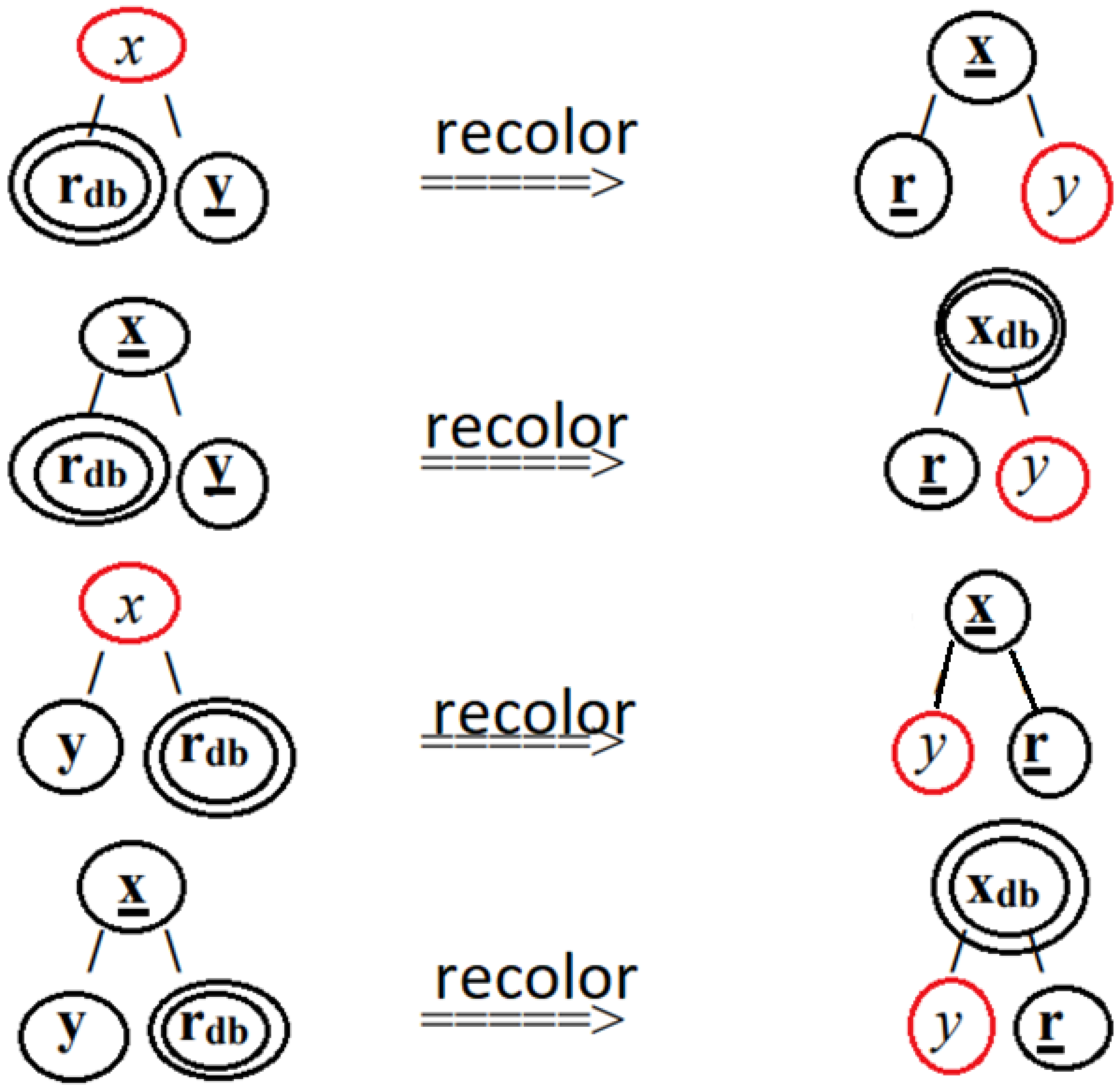
Deletion in an RB Tree: Dealing with case 3.2

- **Case 3.2:** y is black and both of its children are as well.
- We deal with this case by just recoloring, instead of making any structural changes to the tree.
- In particular, we will color r black, (changing it from "double black")
- then color y red.
- To compensate for this, we must change x from red to black.
- BUT, if x was black, we make x “double black” and process further

Deletion in an RB Tree: Dealing with case 3.2

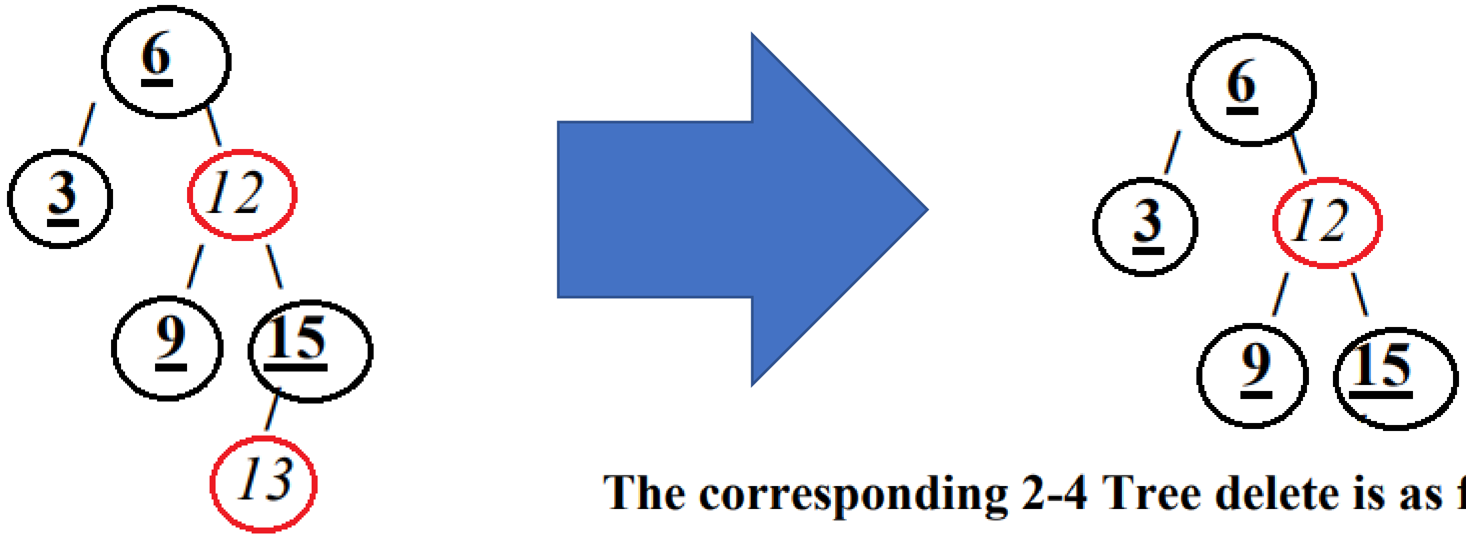
<u>**Case 3.2:**</u> y is black and both of its children are as well.

-Note that recoloring a **double black** means making it black.



Example deletion from an RB Tree

- Delete 13
- Case 1: Red leaf node
- Simply remove the node like a BST. No more action

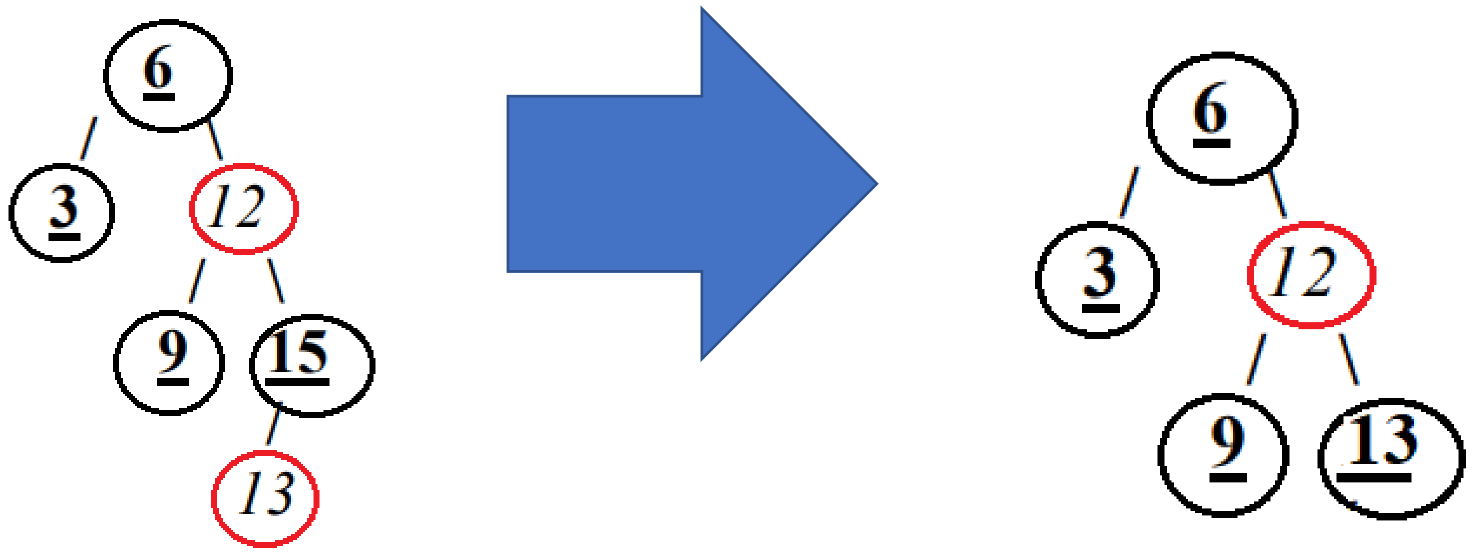


The corresponding 2-4 Tree delete is as follows:

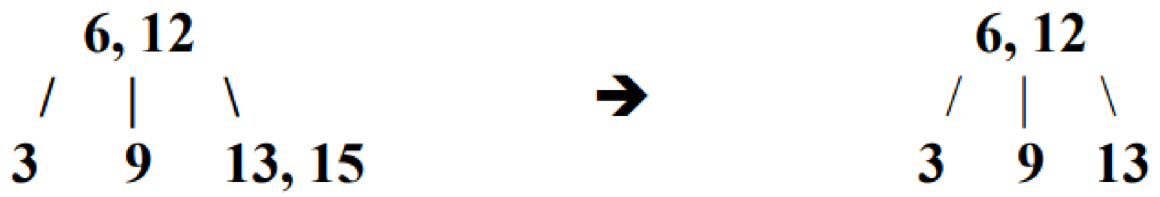
```
      6, 12                        6, 12
    /   |    \                   /   |   \
   3    9    13, 15     ➔       3    9    15
```

Example deletion from an RB Tree

- Delete 15
- Case 2: **Black** node, with one red child node
- Remove the node like a BST. And then recolor the child node of the deleted node

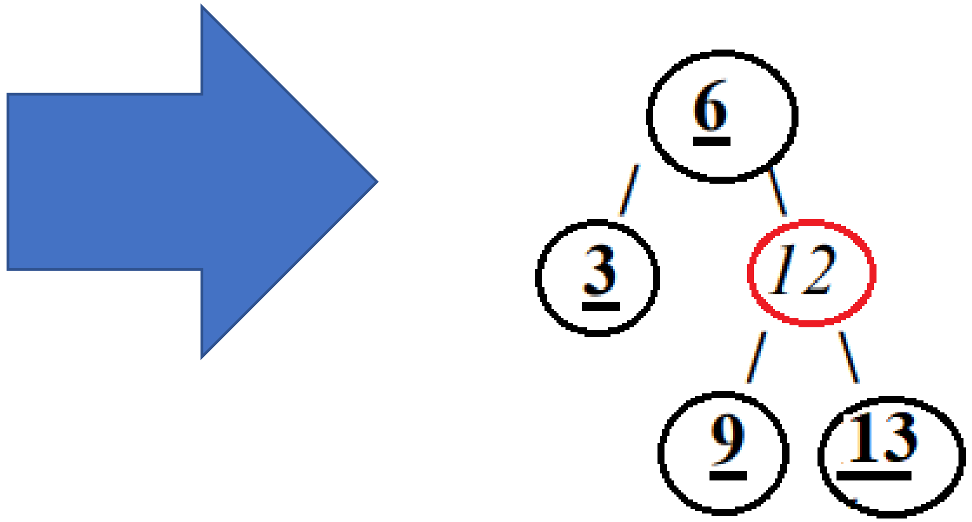


The corresponding 2-4 Tree delete is as follows:



Example deletion from an RB Tree

- Delete 15
- Case 2: **Black** node, with one red child node
- Remove the node like a BST. And then recolor the child node of the deleted node

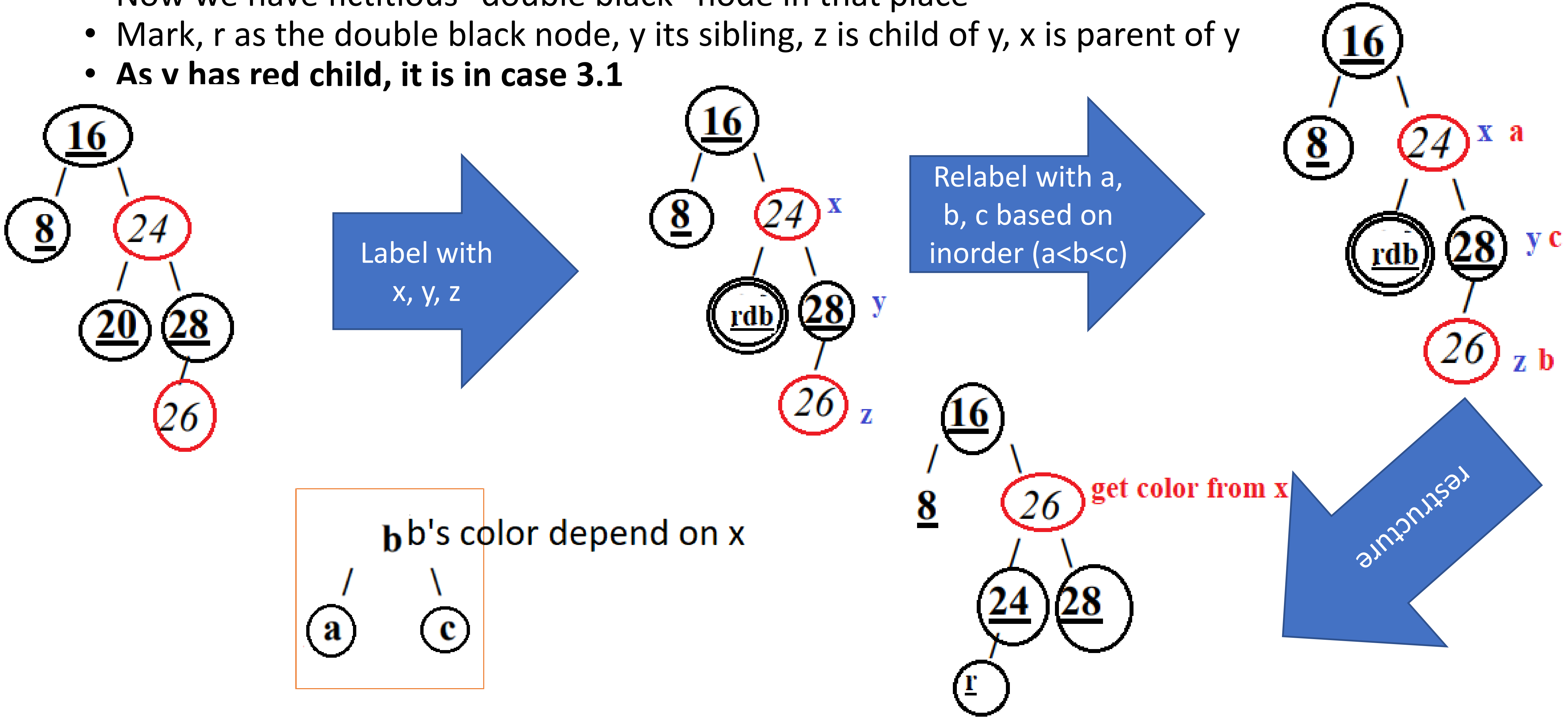


The corresponding 2-4 Tree delete is as follows:

```
   6, 12                        6, 12
 /   |   \          ➔         /   |   \
3    9    13, 15             3    9    13
```

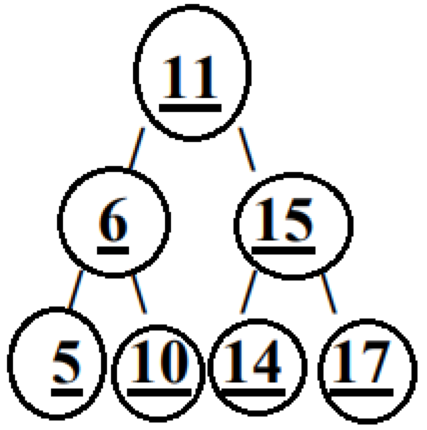
Example deletion from an RB Tree

- Delete **20**
- Case 3: **Black** leaf node
  - Now we have fictitious “double black” node in that place
  - Mark, r as the double black node, y its sibling, z is child of y, x is parent of y
  - **As y has red child, it is in case 3.1**

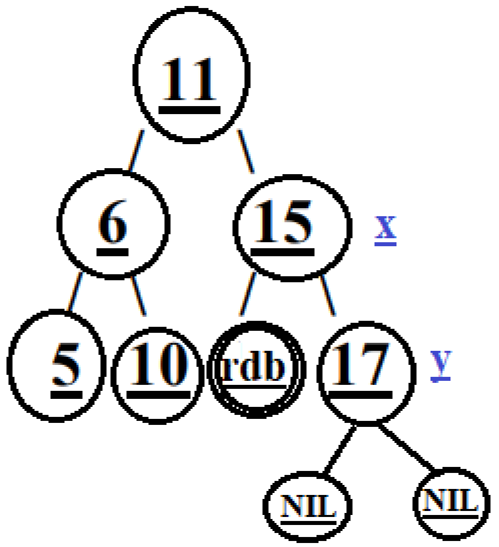


Example deletion from an RB Tree

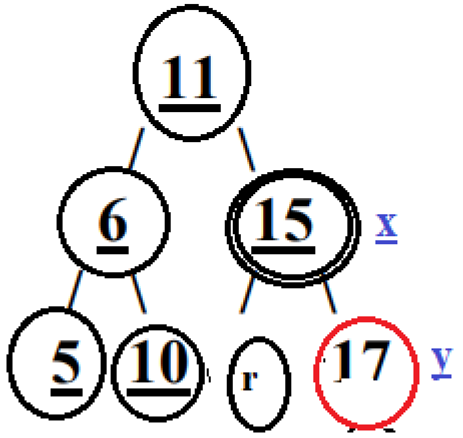
- Delete 14
- Case 3: **Black** leaf node
  - Now we have fictitious “double black” node in that place
  - Mark, r as the double black node, y its sibling, z is child of y (not necessary), x is parent of y
  - As y has 2 black children (case 3.2)



Label with x, y, z



color (but 15 is now double black)

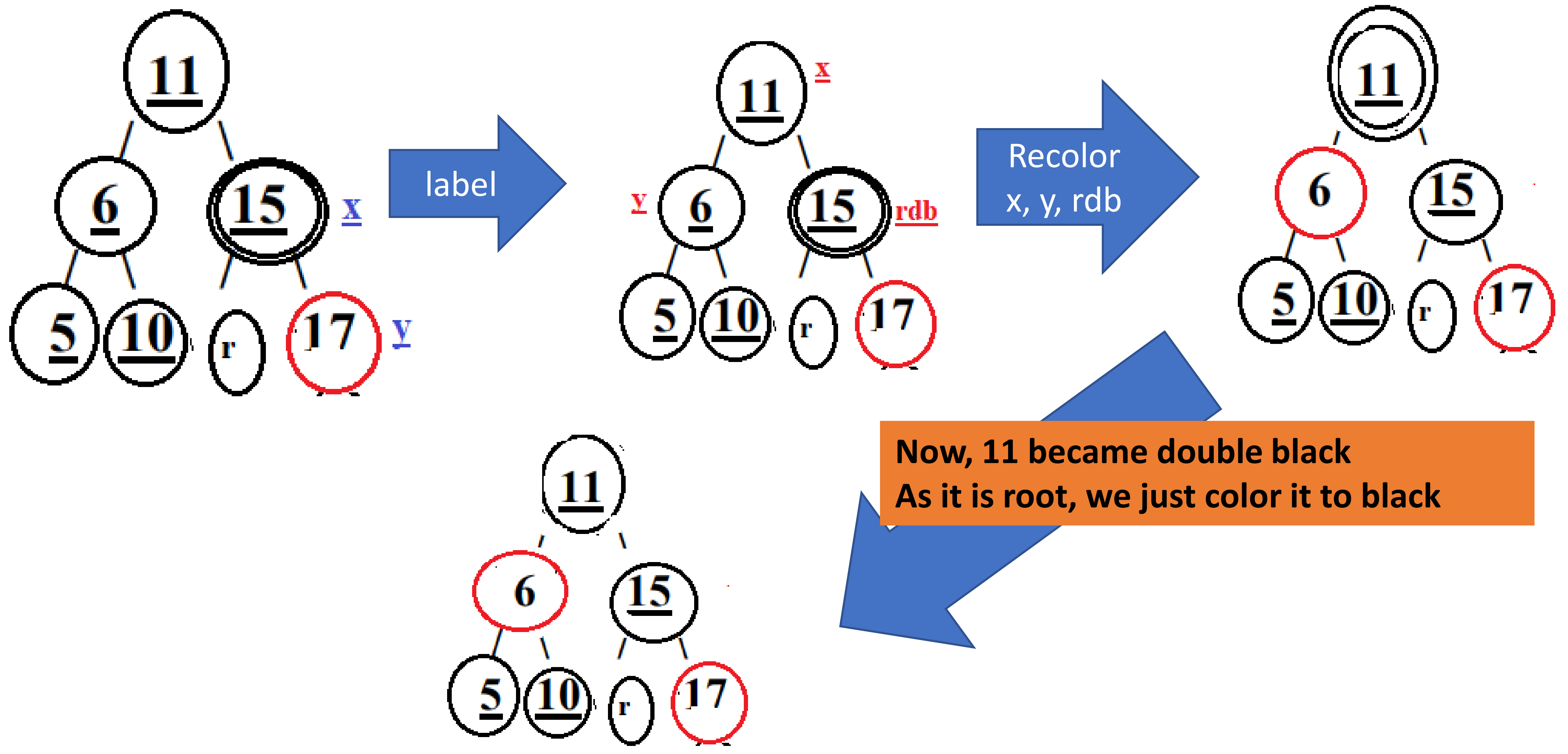


Now, 15 is a double black node, its sibling has two black children, so it is again case 2

Example deletion from an RB Tree

- Delete 14 **(we are still processing)**

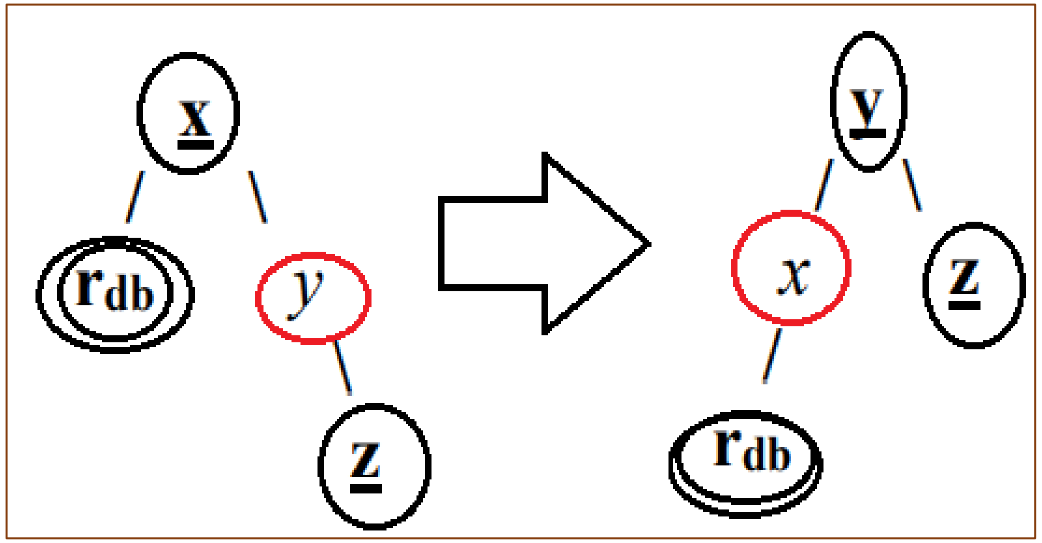
Now, 15 is a double black node, its sibling has two black children, so it is again case 2
- Label, x, y, z (Z not needed for 3.2)

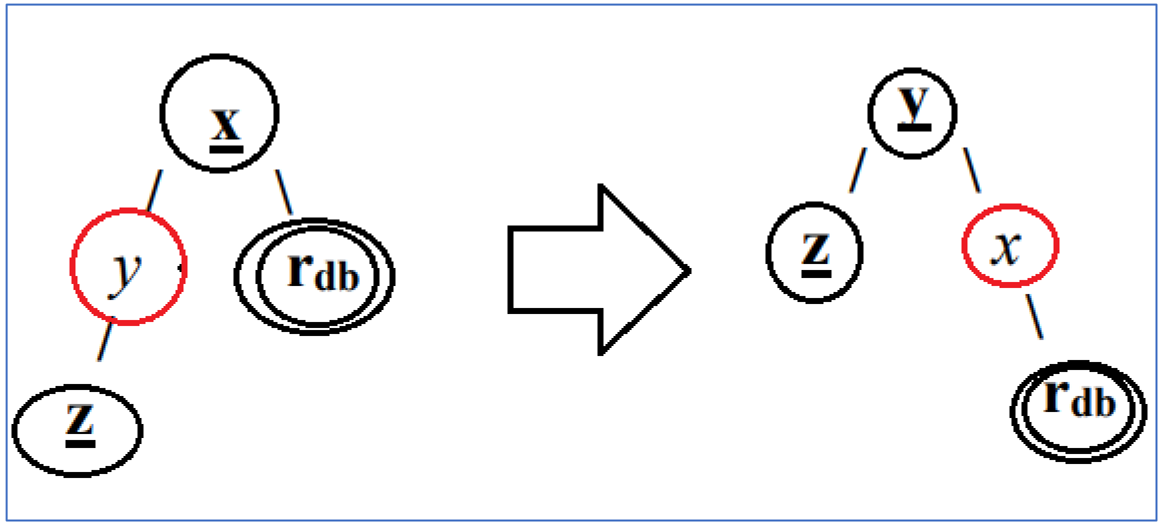


Now, 11 became double black
As it is root, we just color it to black

Deletion in an RB Tree: Dealing with case 3.3

- This is the last case where we have to deal with a **“double black”** node r.
- In this situation, y, the sibling of r, is red.
- If y is a right child of x (where x is r's parent), let z be the right child of y.
- Otherwise, let z be the left child of y.
- (Note that both x and z must be black. (as y is read)
- Perform a restructuring on the node z, placing y where x used to be:



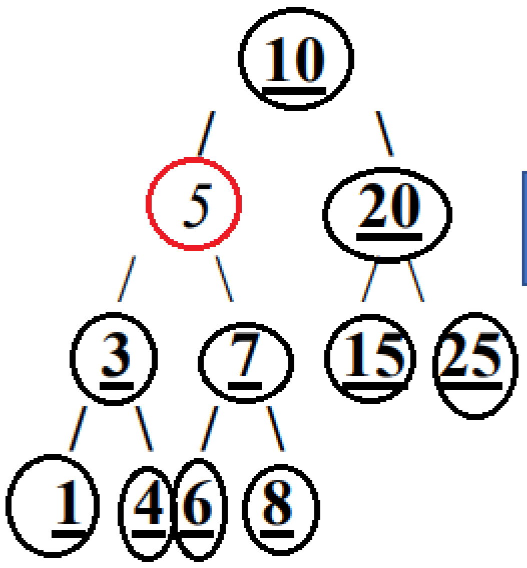


Deletion in an RB Tree: Dealing with case 3.3

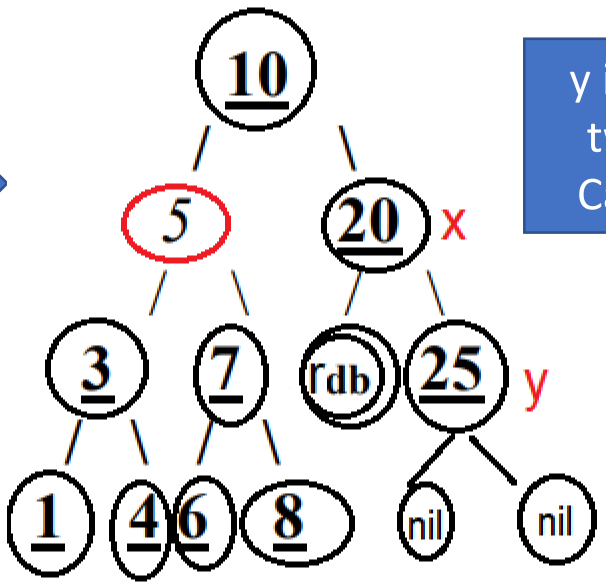
- In the first picture of the last slide, since y's right child in the original picture MUST BE black, x's new left child MUST BE black as well. This puts us in either case 3.1 or case 3.2 to deal with the "double black" node.
- In the second picture of the last slide, we have that y's original left child must have been black, thus, in the restructured picture, x's new right child must be black as well, putting us in either case 3.1 or case 3.2 as desired.

Example

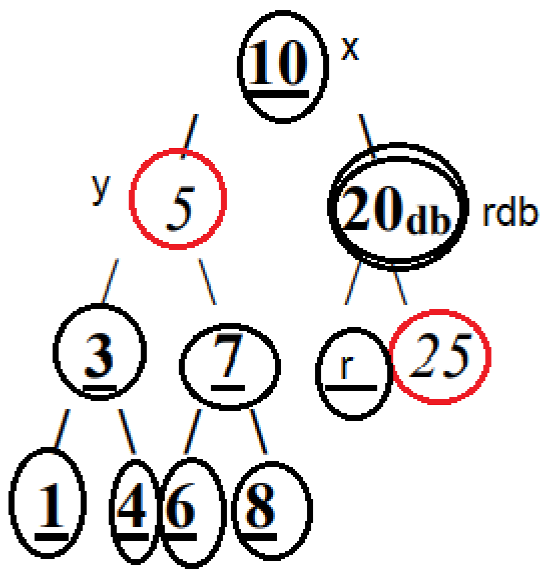
- Consider deleting 15 from the Red-Black Tree below:



Delete 15



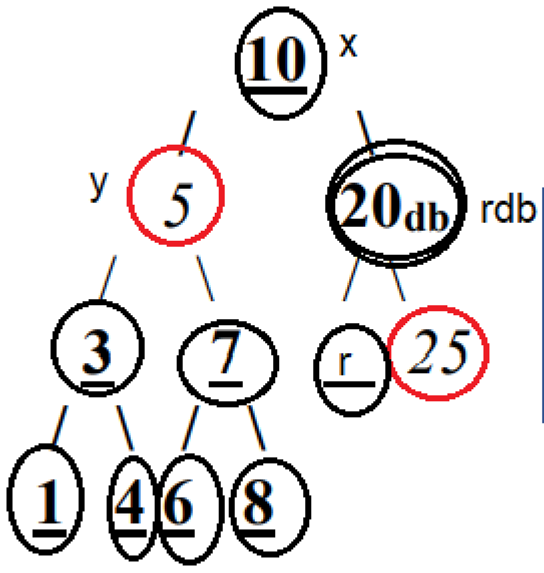
y is black and has two black child. Case3.2. Recolor



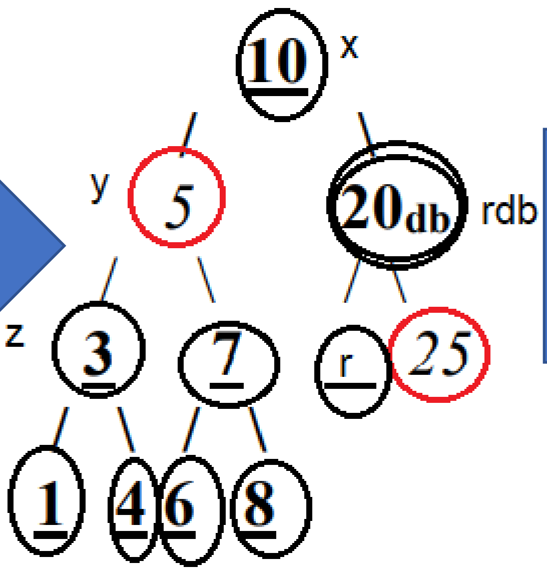
After recoloring, 20 is now "double black", its sibling y is red.
So, case 3.3
Let's solve it in the next slide

Example

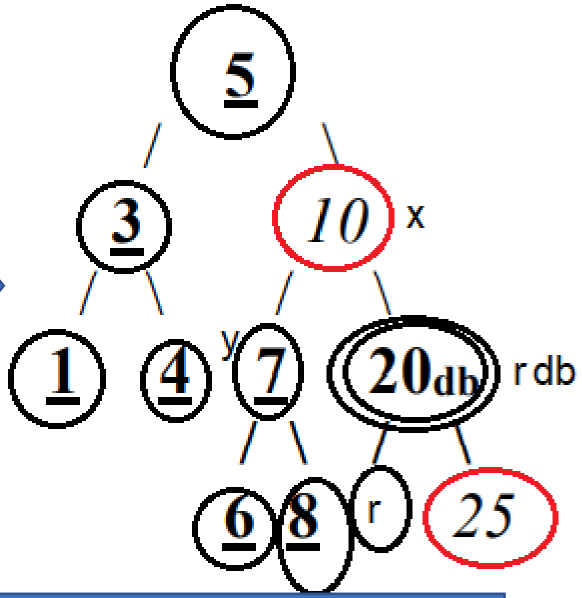
- Continuing the process of deleting 15 from the last slide



Case 3.3
Y is 5's left child, take 3 as z



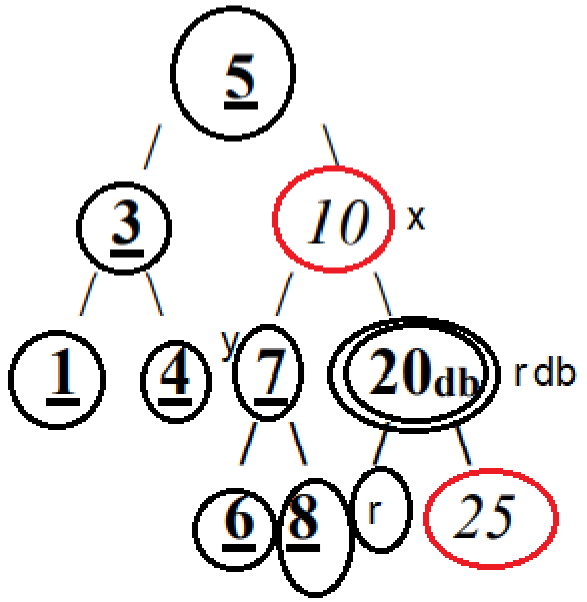
Restructure on z, place y to the place of x



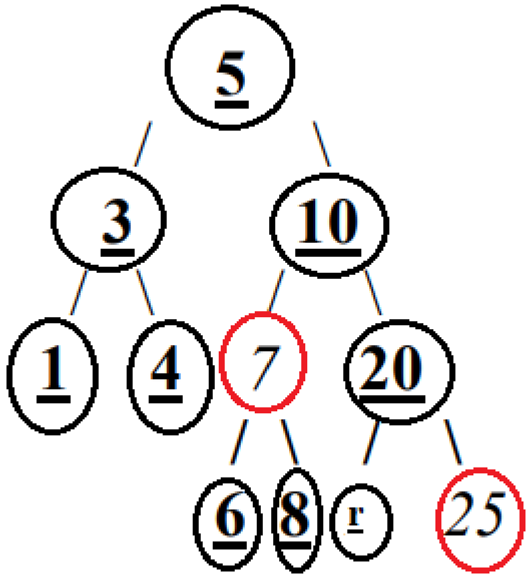
Now we are in case 3.2 as r's sibling 7 is black and has 2 black children
Let's continue next slide

Example

- Continuing the process of deleting 15 from the last slide



Recolor x, y, and rdb as it is in case 3.2



Summarizing Red-Black Tree Delete

- We have a couple simple cases to deal with, which we can do without any extra work.
- The rest of the cases result in a "double black" colored node. The goal is to deal with the double black(DB) node to get rid of this property. Here is the outline of these cases:
- Case 3.1: DB node has black sibling with at least one red child.
  - This fixes a the problem structurally. No extra work is required after this case completes.
- Case 3.2: DB node has black sibling with two black children.
  - This uses a recoloring and no structural change. It may solve the problem, but may ALSO propagate the DB node to the parent of the current DB node.
- Case 3.3: DB Node has red sibling
  - A structural change here puts you in case 3.1 or case 3.2. At this point, a single application of either case is sufficient.