Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Balanced Binary Search Trees

Outline

- Motivation
- AVL Trees
- Red-Black Trees

Red-Black Trees

- Height balanced trees like AVL trees
- Insertions, deletions and look-ups in O(log n) time
- If we have AVL trees, the why Red-Black Trees?
  - AVL trees are more rigidly balanced, thus they provide faster look-ups, but deletions and insertions are slow, relatively – more frequent

Red-Black Tree

- How is the tree kept balanced?
- During insertions and deletions, it is made sure that certain **Properties** of the tree are not violated.
- Properties:
  - The nodes are colored
  - Arrangement of these colors

Red-Black Trees

- **Colored Nodes:** nodes can be colored either red or black to satisfy the following conditions:

- **Red-Black Properties:**

1. **Root Property:** The root is black
2. **External Property:** Every external node is black
3. **Red Property:** The children of a red node are black
4. **Black Property:** Every path from the root-to-frontier contains exactly the same number of black internal nodes (**black depth**)

Red-Black Trees

- How do you color a node?
  - Just include a field “color” in the node class
  - or a boolean variable “isRed” or “isBlack”
- Why red and black colors?
  - arbitrary choice
  - you can use yellow and green

Note!

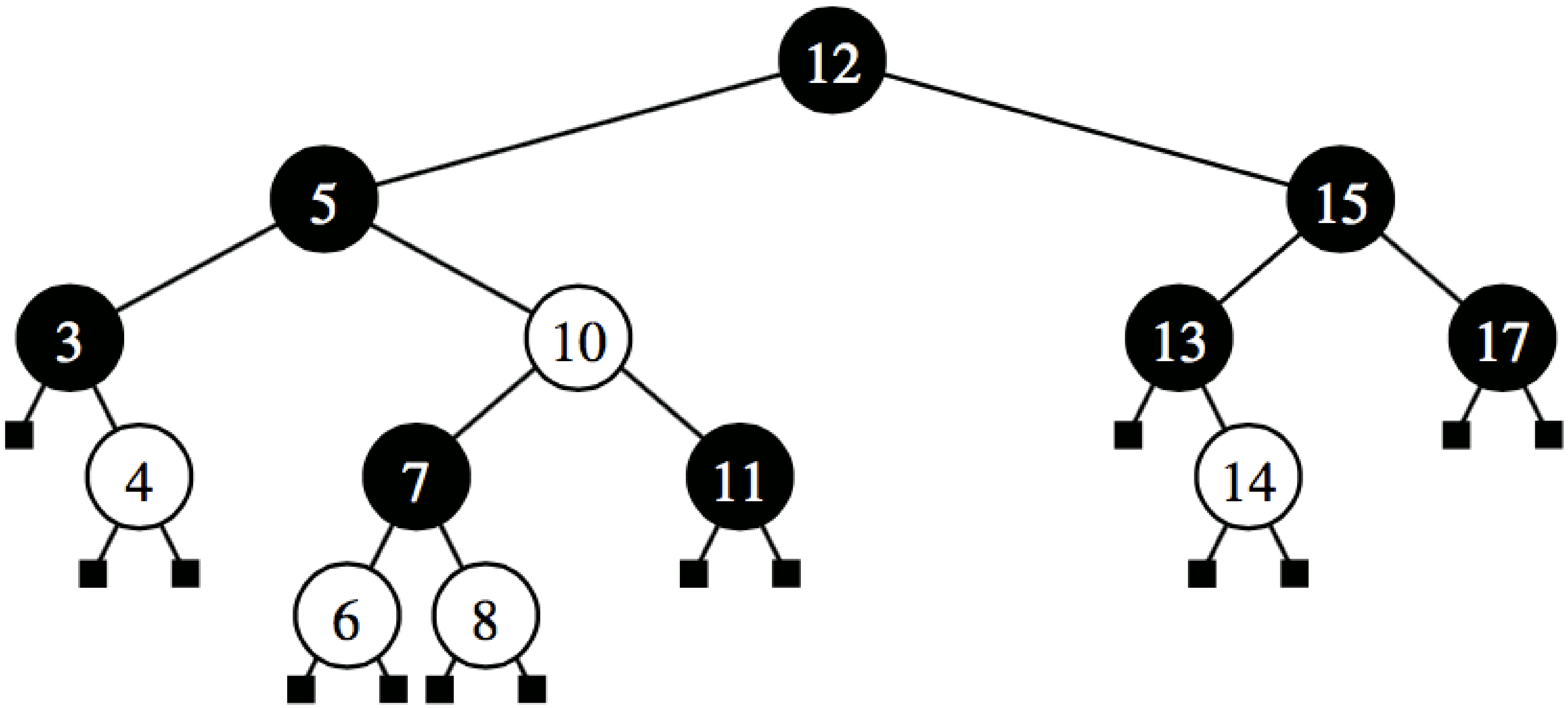
- For ease, I have taken some (not all) figures from the book (Data Structures & Algorithms in Java by Goodrich)

- In those figures, red nodes are colored as white nodes

- Please keep this in mind – Sorry for that!

- Also, you can see implementation in the book, here I will focus on concepts

Red-Black Tree



An example of a red-black tree, with "red" nodes drawn in white. The common black depth for this tree is 3.

Height of Red-Black Trees (I)

- **"Height of RB-Trees is O(log n)**

1. The black depth of the tree is $O(\log n_b)$, where n_b is the number of black nodes. To see this:

   a. imagine we removed all the red nodes- this would not change the black depth, but now the black depth of the leaves is the same as the height (no red nodes, see?).

   b. Thus we have a tree where all leaves have the same depth, or a complete tree.

   c. We know that complete trees have height $O(\log n)$.

   d. So, since the black depth of the original tree is equal to the height of the modified tree, and the height of the modified tree is $O(\log n_b)$, the black depth of the original tree is $O(\log n_b)$.

Height of Red-Black Trees (II)

2. In the worst case, that is the case with the tallest tree, there must be some long path from the root to a leaf.

3. Since the number of black nodes on that long path is limited to $O(\log n_b)$, the only way to make it longer is to have lots of red nodes.

4. Since red nodes cannot have red children, in the worst case, the number of nodes on that path must alternate red/black.

5. thus, that path can be only twice as long as the black depth of the tree.

6. Therefore, the worst case height of the tree is $O(2 \log n_b)$.

7. Therefore, the height of a red-black tree is $O(\log n)$.

Searching in RB Trees

- A red-black tree is a binary search tree
- Search operation in RB trees is performed exactly the same way as in binary search trees
- The color doesn’t matter
- O(log n)

Insertions in RB Trees

- Just as with AVL trees, perform the insertion by
  - first searching the tree until an external node is reached (if the key is not already in the tree)
  - then inserting the new node **x**
- Color of the new node **x**:
  - If this is the first entry, x is root, so color it **black**
  - otherwise, always, color it red

Insertions in RB Trees

- Why always color the new node as red if it is not the first entry? Why not color it black?

Insertions in RB Trees

- So the insertion **x**, as a new internal red node, may violate the **red property**

- That is, **y** (which is the parent of **x**) also has the red color

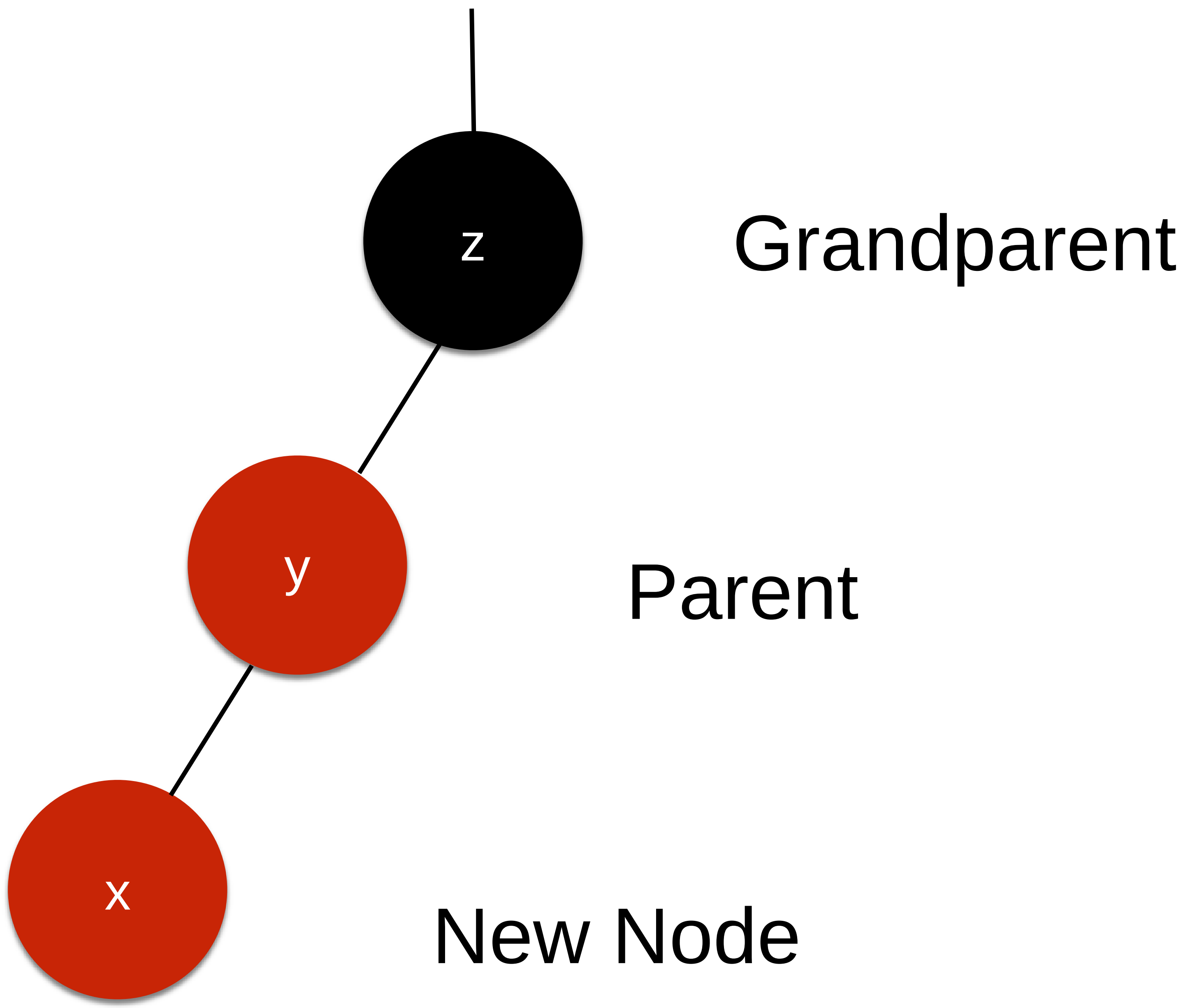
Double red problem

- Answer two questions about **y**:

  - Can **y** be the root of the tree?

  - Let **z** be the parent of **y (x's grandparent)**, what color is **z**?

Double Red Problem

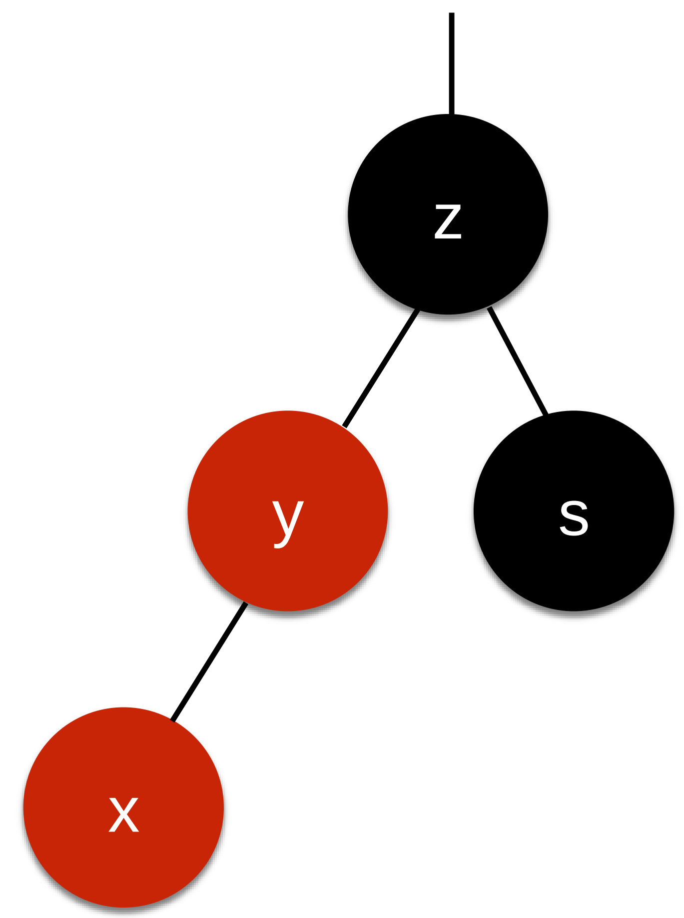


Insertions in RB Trees

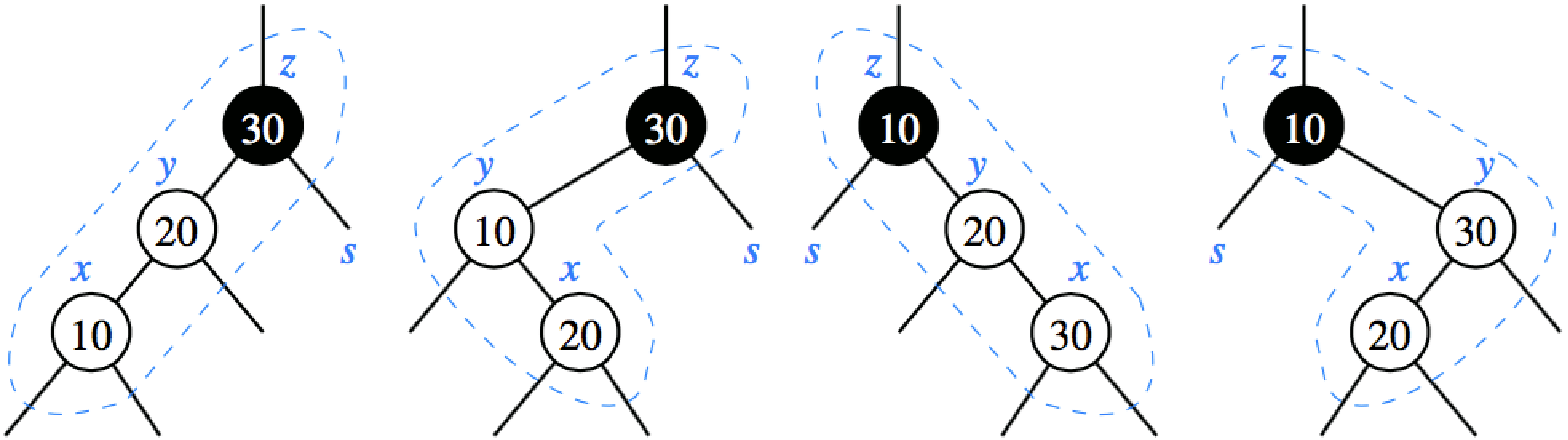
- Double Red Problem: there are two cases
- And they are solved with **restructuring** and **recoloring**, respectively

Insertions in RB Trees

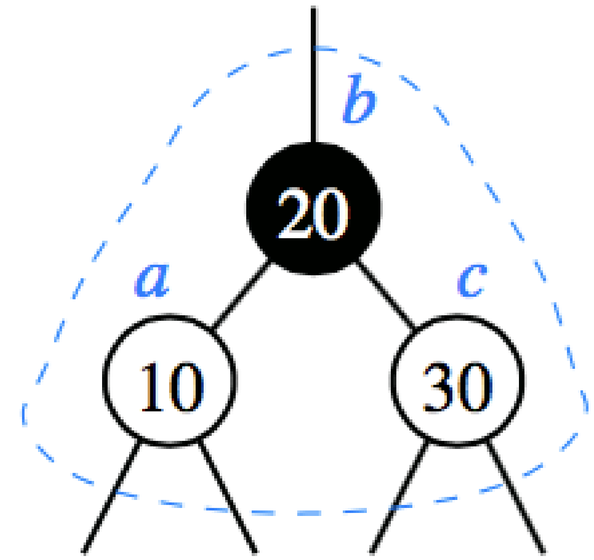
- **Case 1:** Let **s** be the uncle/aunt of x (the sibling of **y) and s** is black
    - Perform the **restructuring** using trinode restructuring algorithm for **x**, **y** and **z**



Case 1



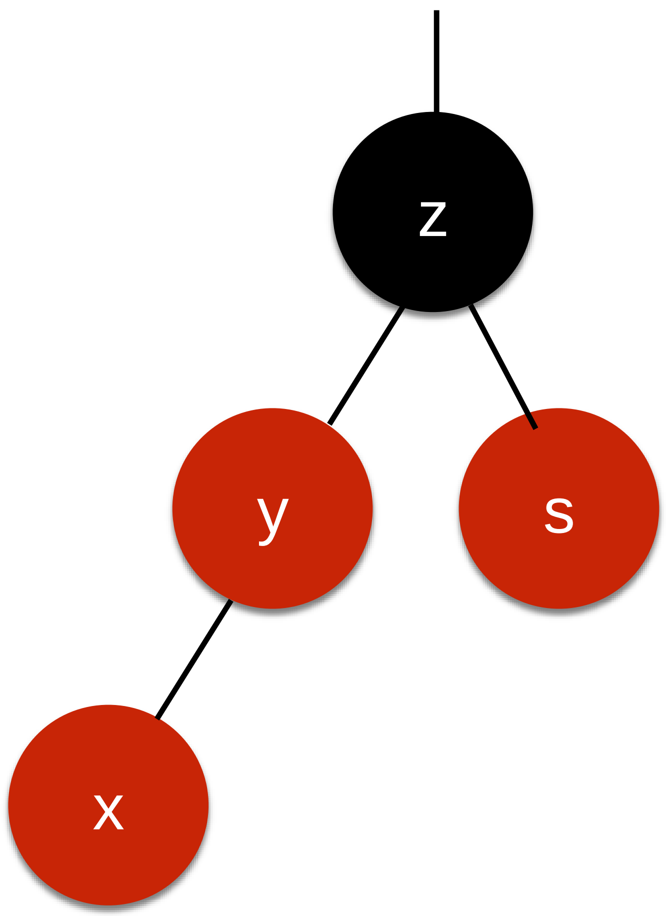
The four configurations for *x*, *y*, and *z* before restructuring



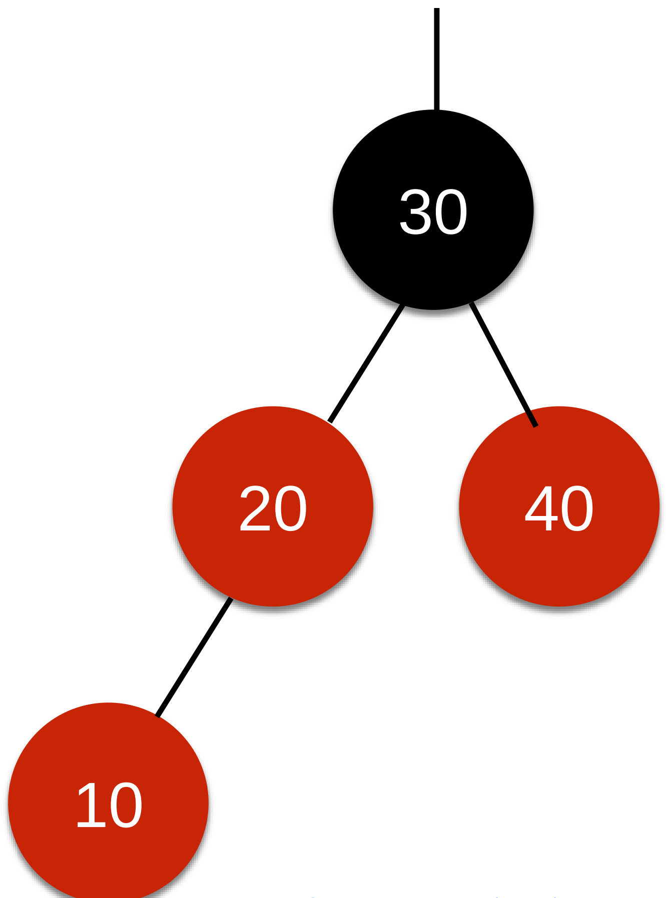
After Restructuring (a, b and c represent x, y and z respectively)

Insertions in RB Trees

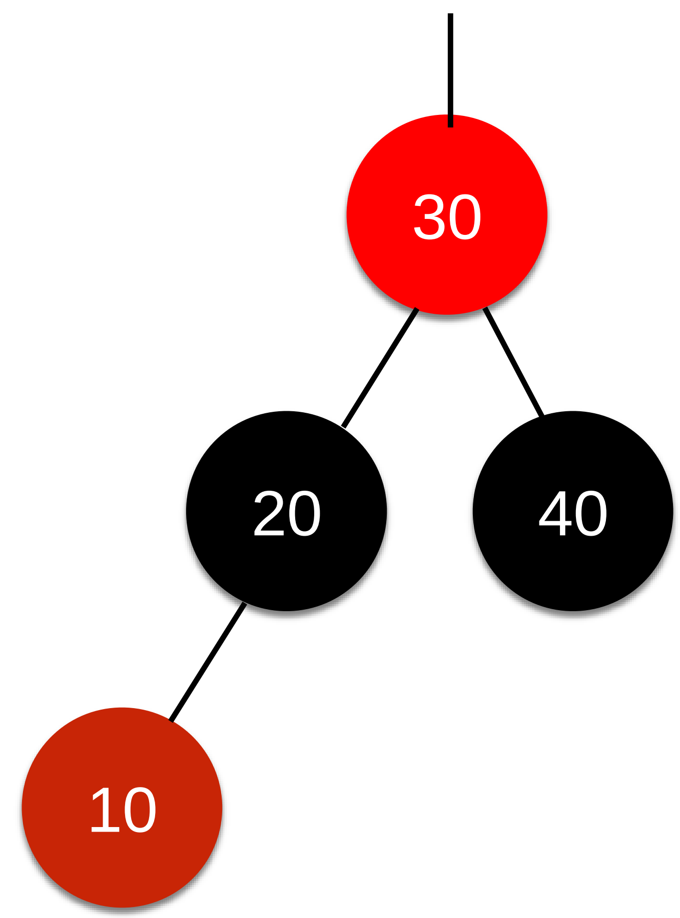
- **Case 2: s** is red
  - In this case, perform **recoloring of s, y and z**
  - color **y** and **s** <u>black</u>, and their parent **z** **red** (unless **z** is the root node)



Case 2



Before recoloring



After recoloring

Example

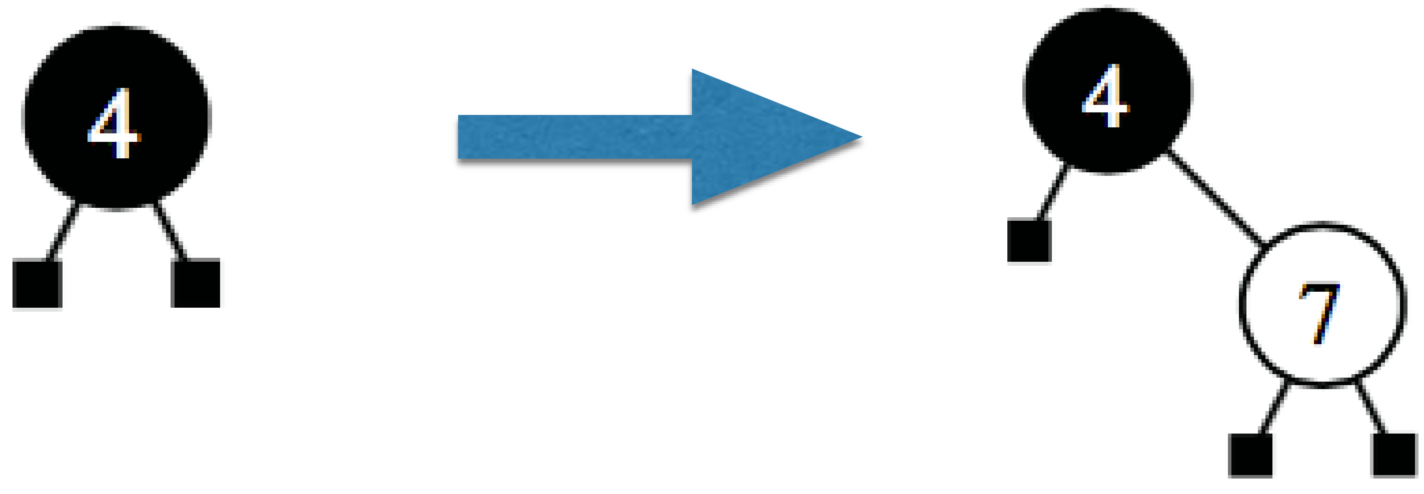
- Sequence of insertion operations in a red-black tree
- Sequence: 4, 7, 12, 15, 3, 5, 14, 18

Example

4, 7, 12, 15, 3, 5, 14, 18

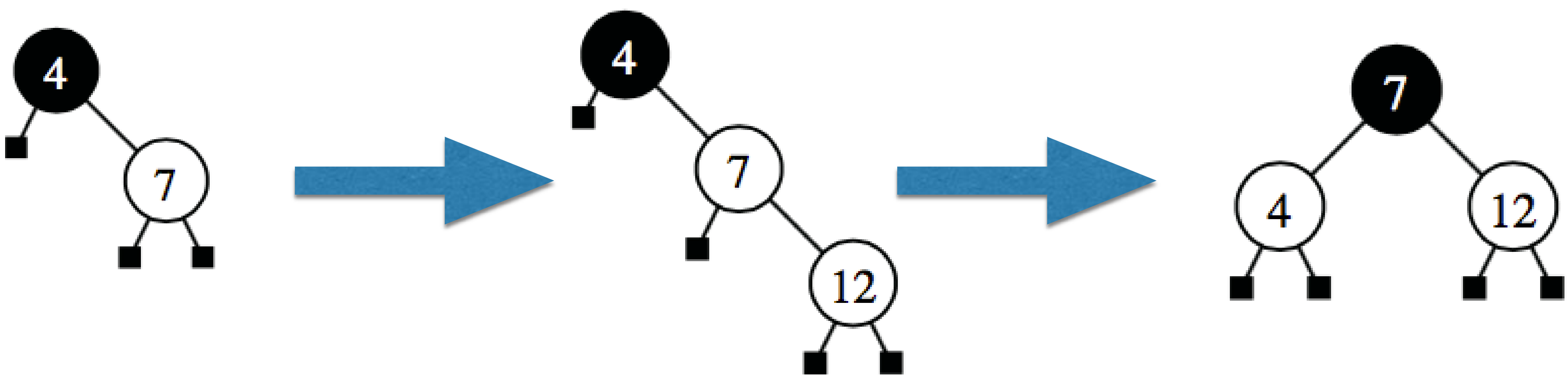
Example

4, **7**, 12, 15, 3, 5, 14, 18



Example

4, 7, **12**, 15, 3, 5, 14, 18

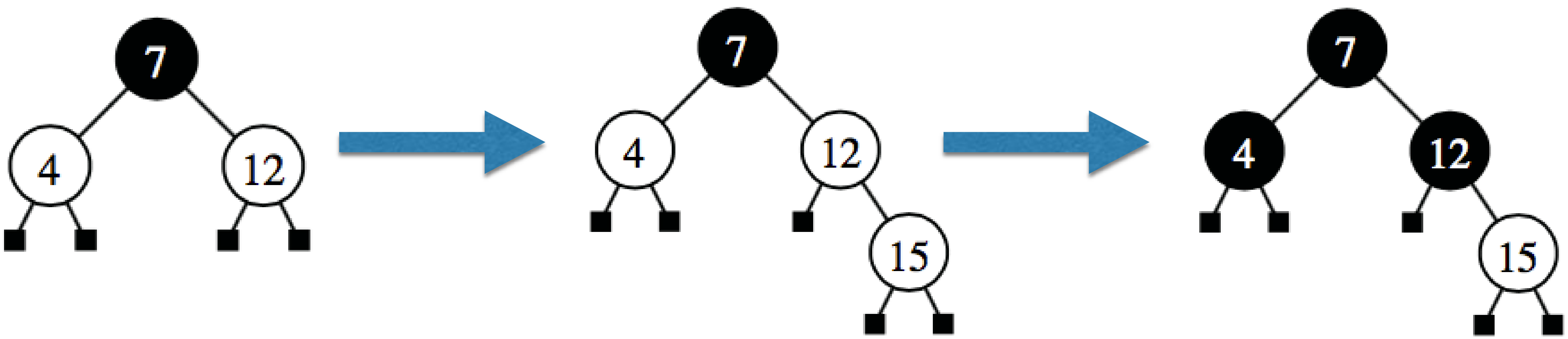


After Insertion

After Restructuring

Example

4, 7, 12, **15**, 3, 5, 14, 18

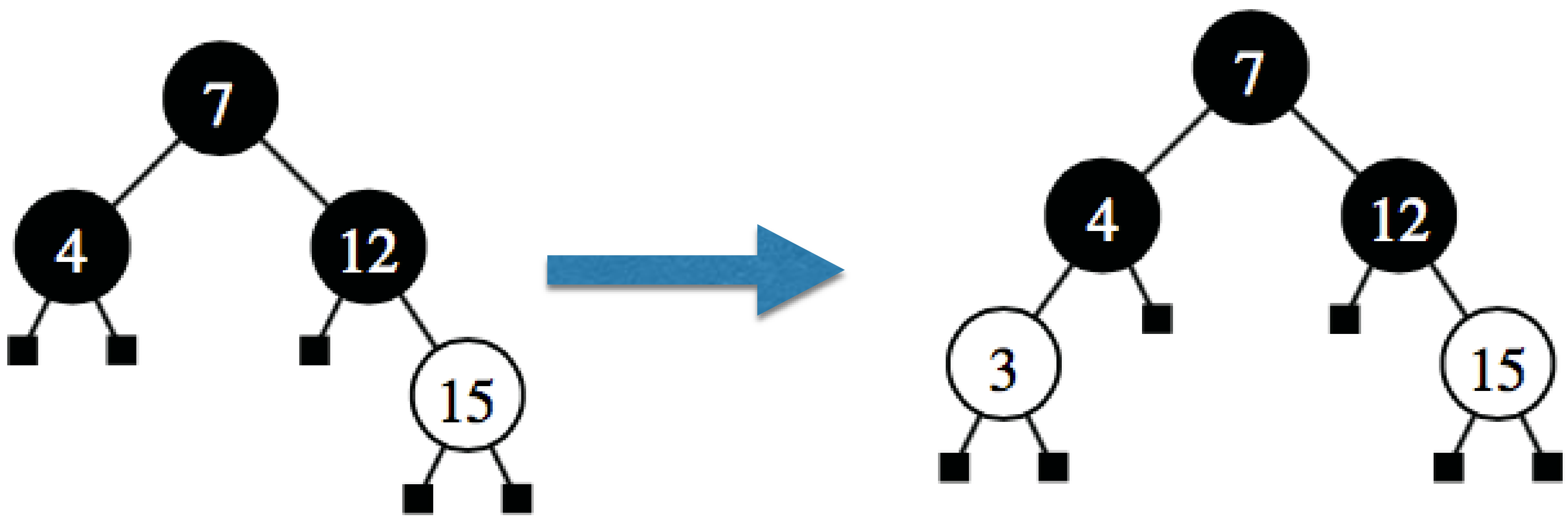


After Insertion

After Recoloring

Example

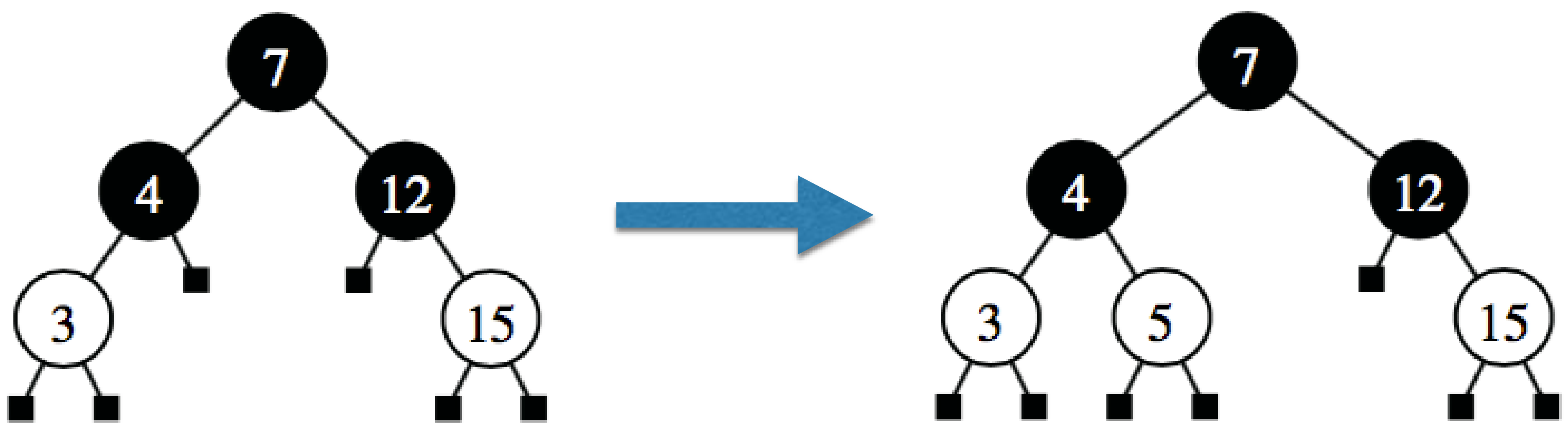
4, 7, 12, 15, **3**, 5, 14, 18



After Insertion

Example

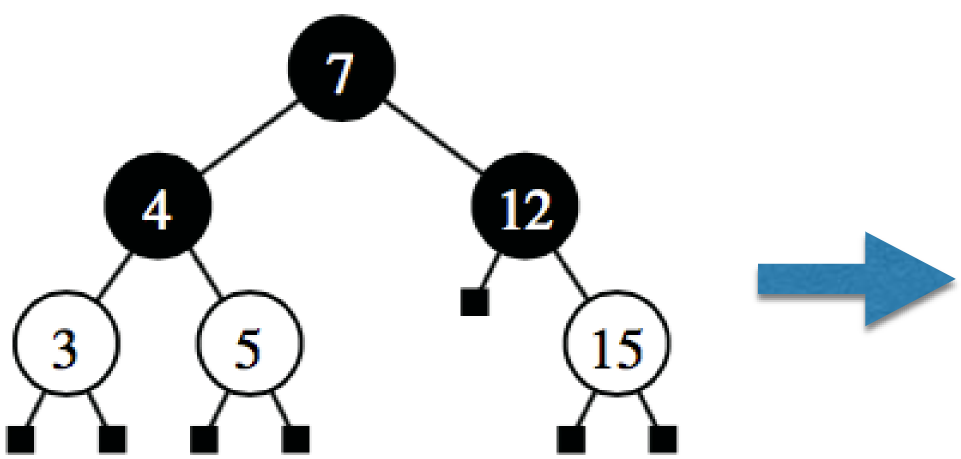
4, 7, 12, 15, 3, **5**, 14, 18

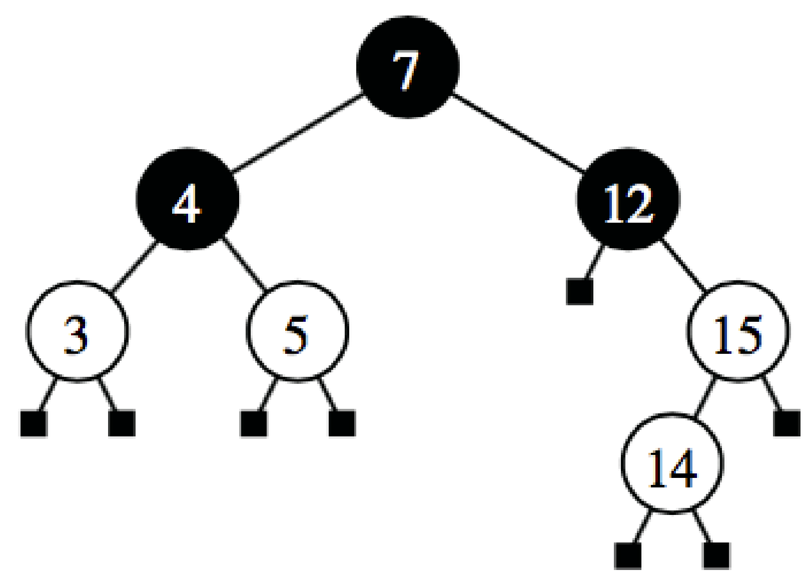


After Insertion

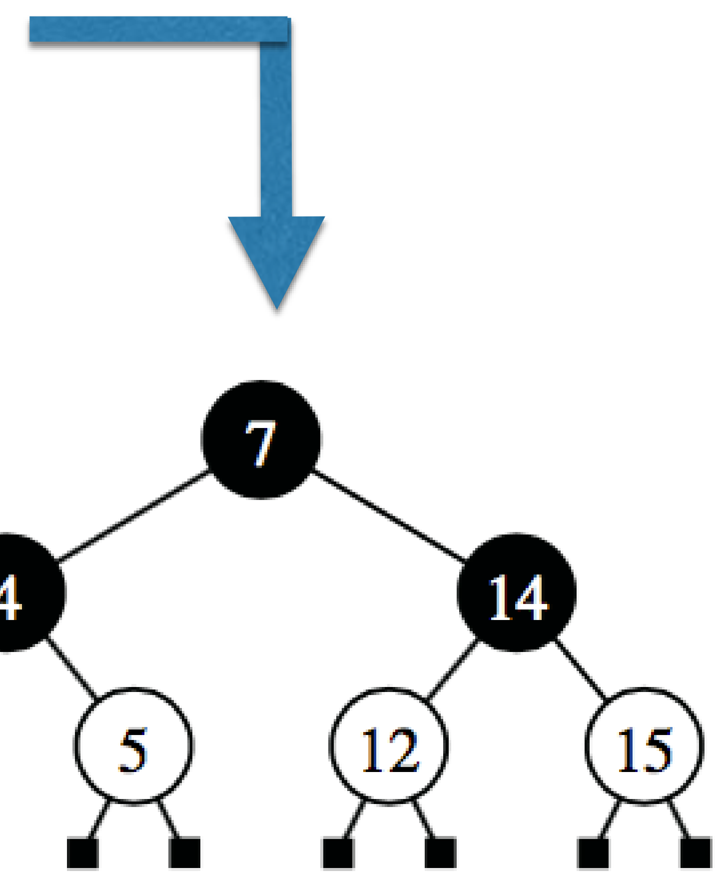
Example

4, 7, 12, 15, 3, 5, **14**, 18





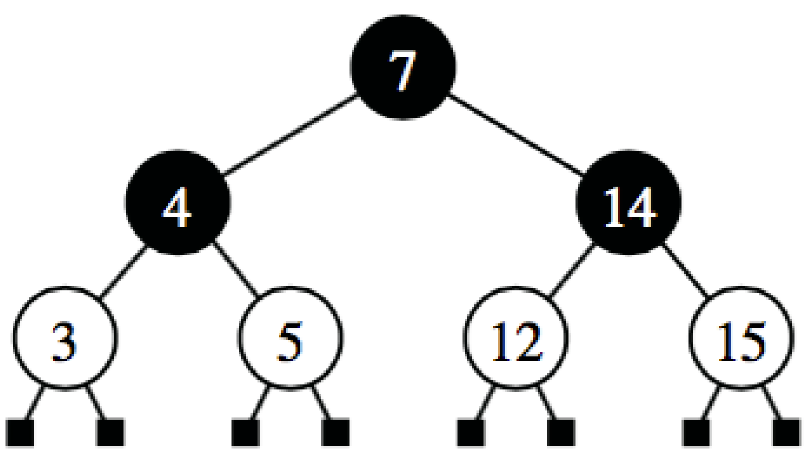
After Insertion

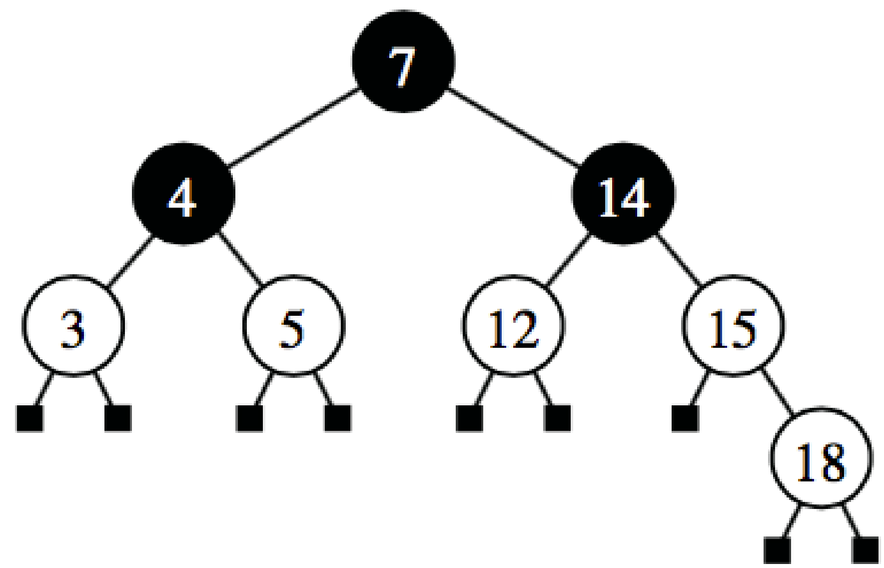


After Restructuring

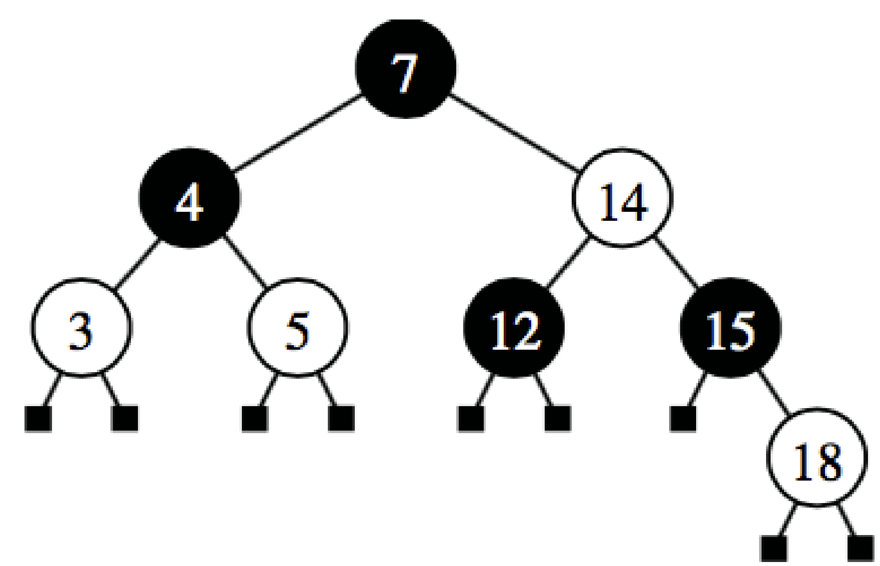
Example

4, 7, 12, 15, 3, 5, 14, **18**





After Insertion



After Recoloring

How long does it take?

- Finding the place to insert – O(log n)
- Insertion – O(1)
- Fixing the double red problem – O(1)
- In worst case, the problem can cascade until the root – O(log n)
- Therefore **O(log n)**

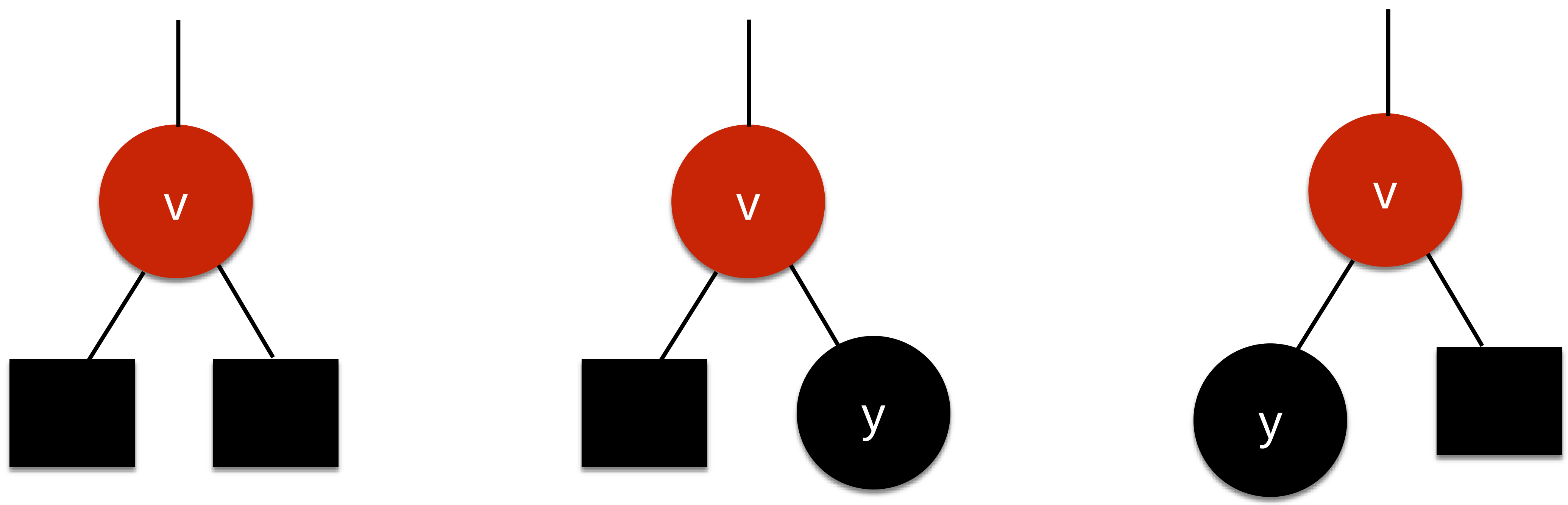
Deletions in RB Trees

- Initially proceed with deletion as for a binary search tree
- Two cases:

Deletions in RB Trees

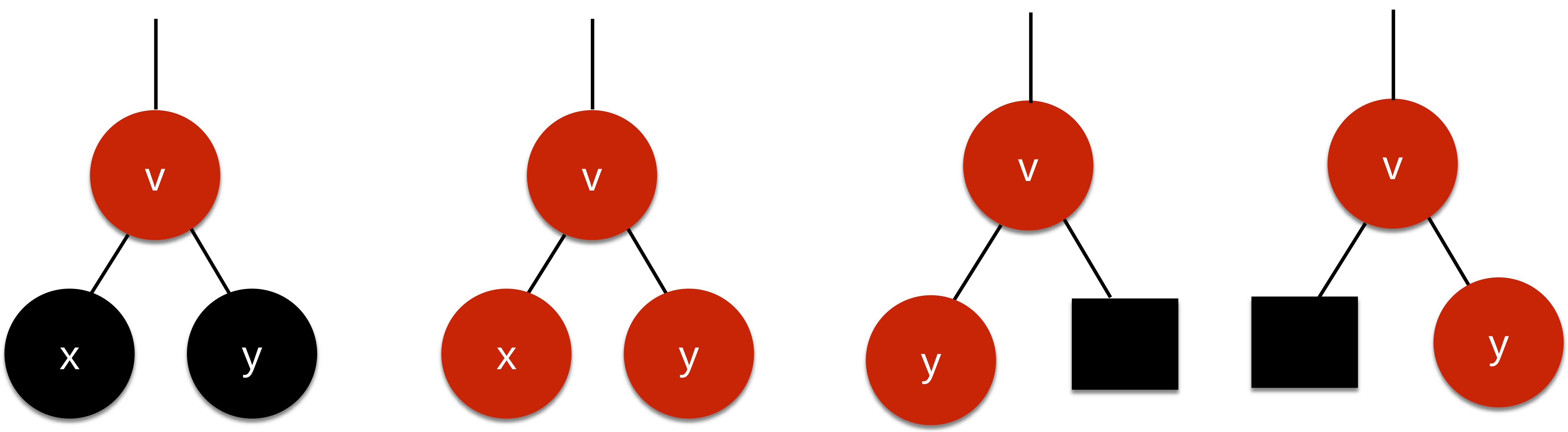
- Case 1: When the node to be removed is a red node (**v** in this case),

- Three scenarios (before the removal)



Deletions in RB Trees

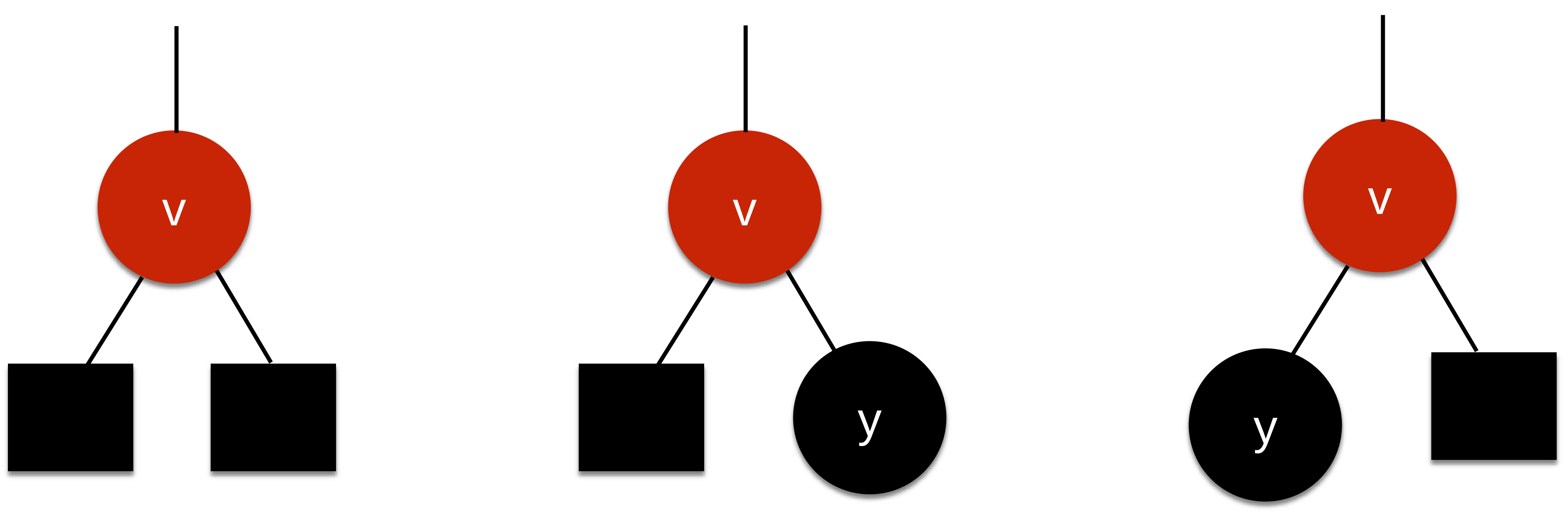
- what about these ones?



Deletions in RB Trees

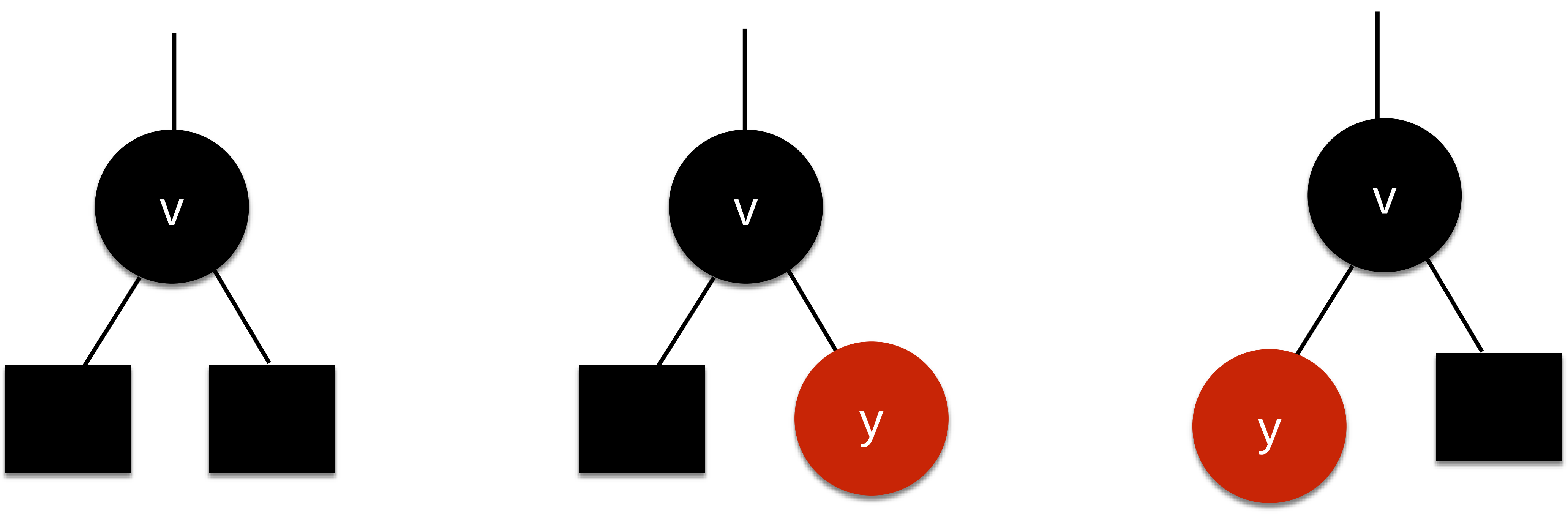
- Thus three scenarios, but what about the color of **v**'s parent?

- Thus, removal of **v**, and moving **y** up will not violate RB tree properties



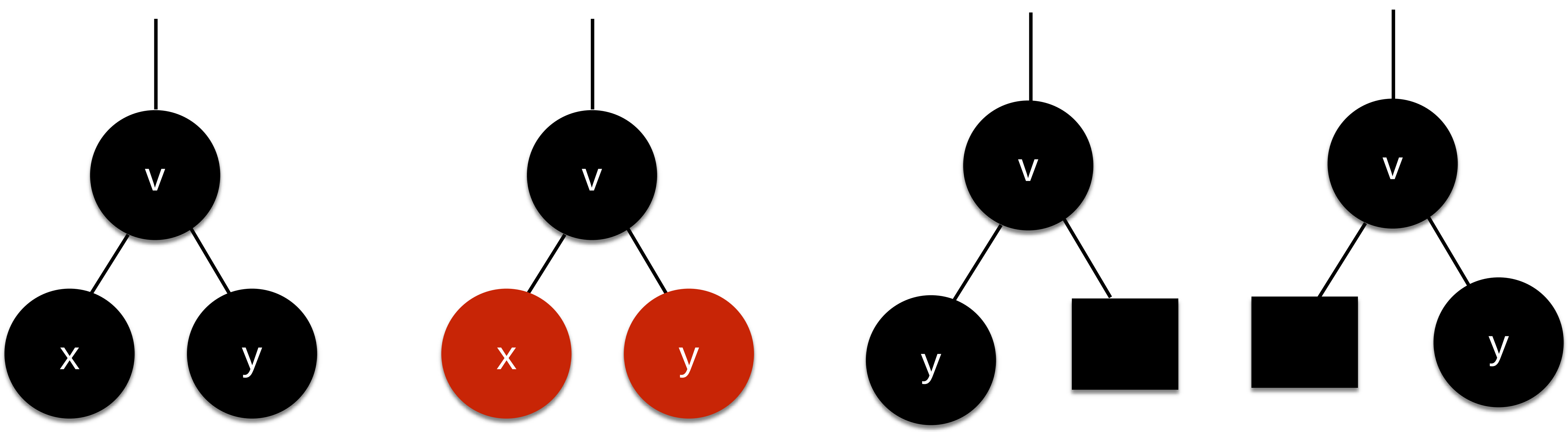
Deletions in RB Trees

- **Case 2:** The node to be removed is a black node
- Again Three scenarios before removal



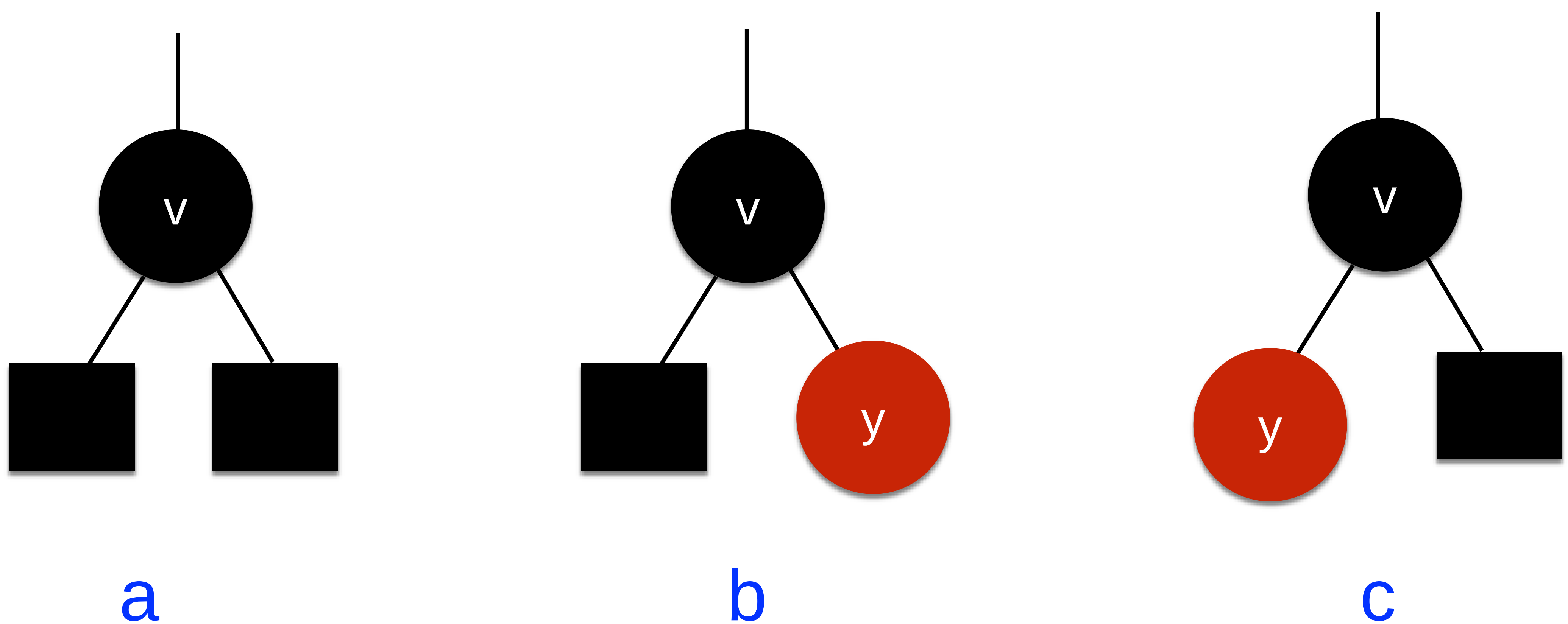
Deletions in RB Trees

- what about these ones?



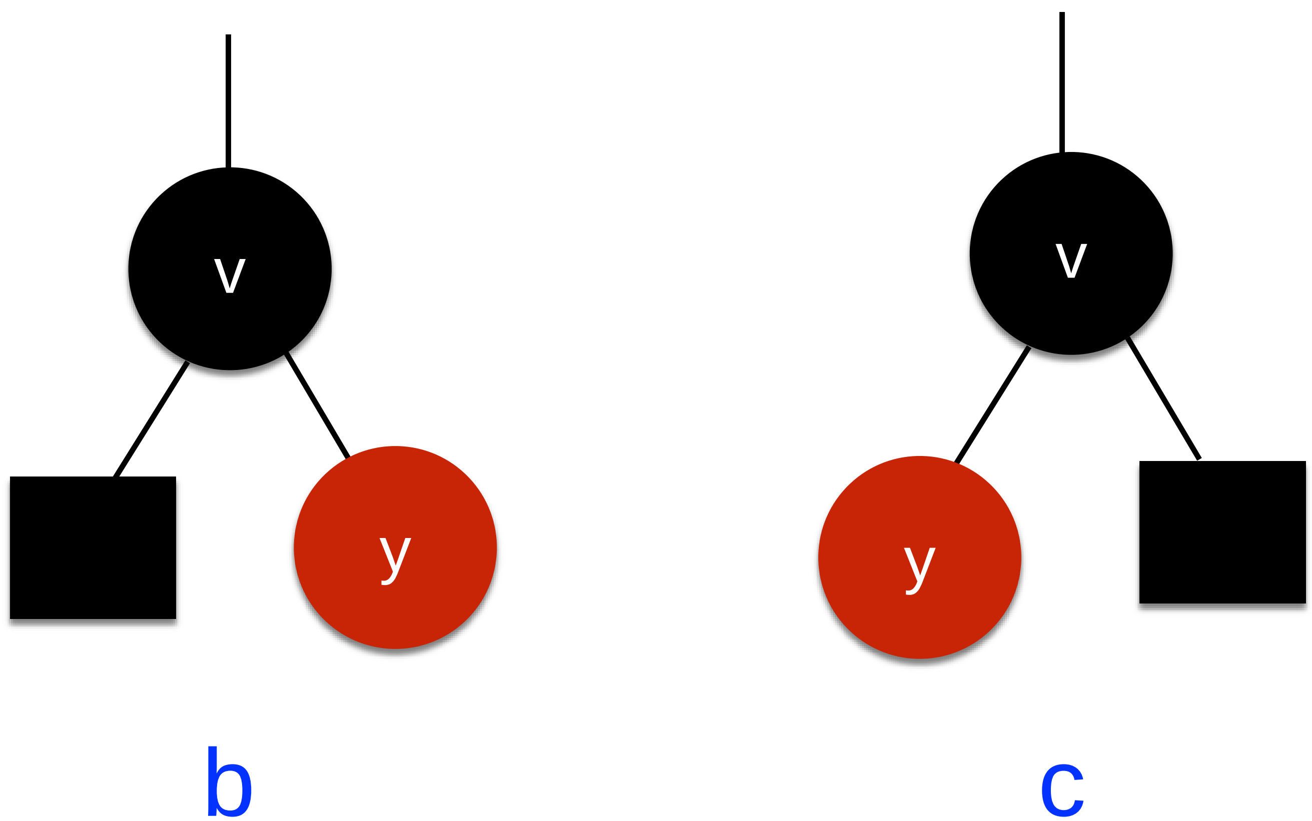
Deletions in RB Trees

- **Case 2:** The node to be removed is a black node
- So we have these three scenarios



Deletions in RB Trees

- **Case 2:** The node to be removed is a <u>black</u> node

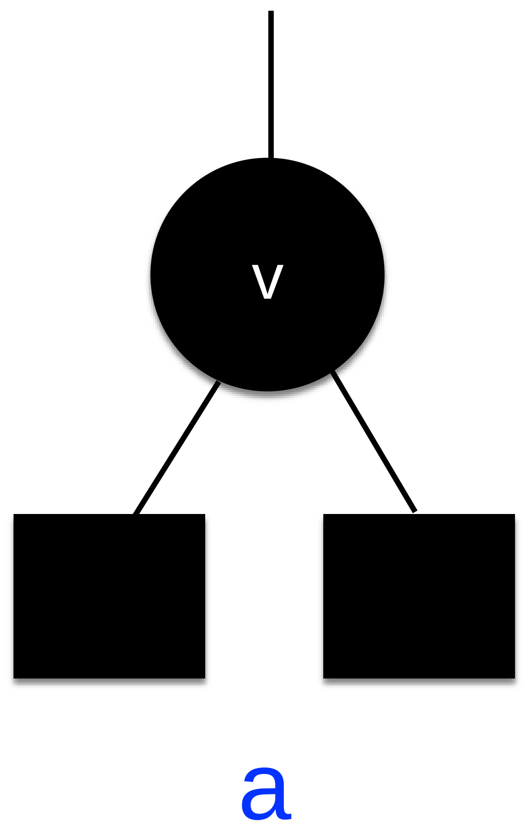


- Remove **v**, move **y** up, and change **y**'s color to <u>black</u>

Deletions in RB Trees

- **Case 2:** The node to be removed is a <u>black</u> node

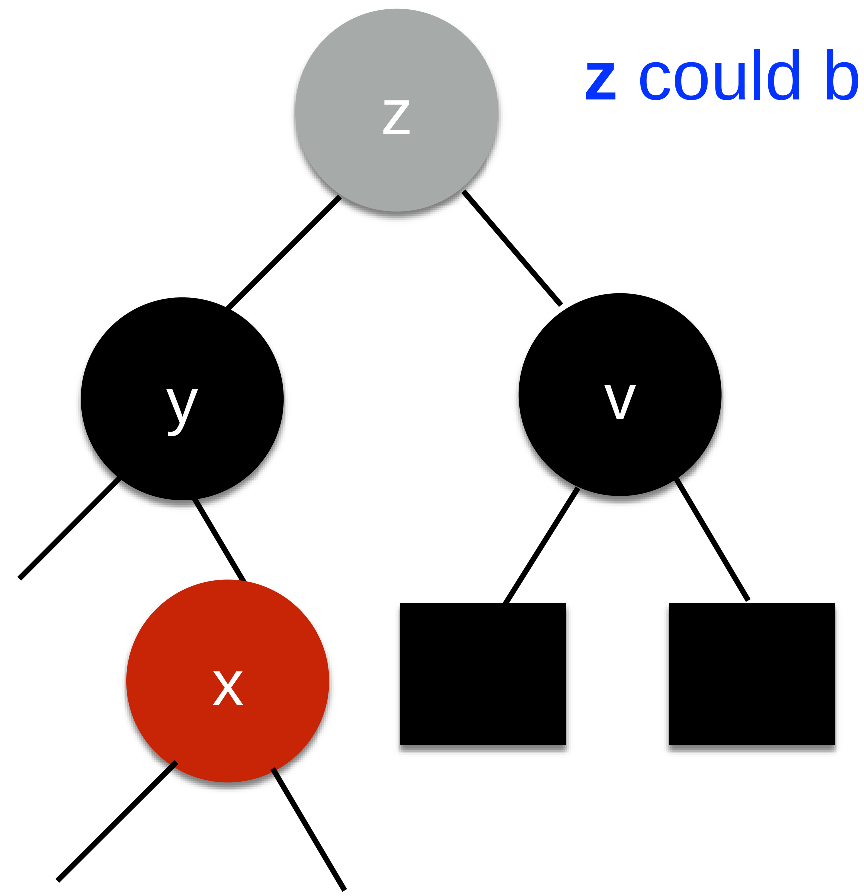
- Deleting **v** will result in violating the depth property of RB Trees



The most difficult case!

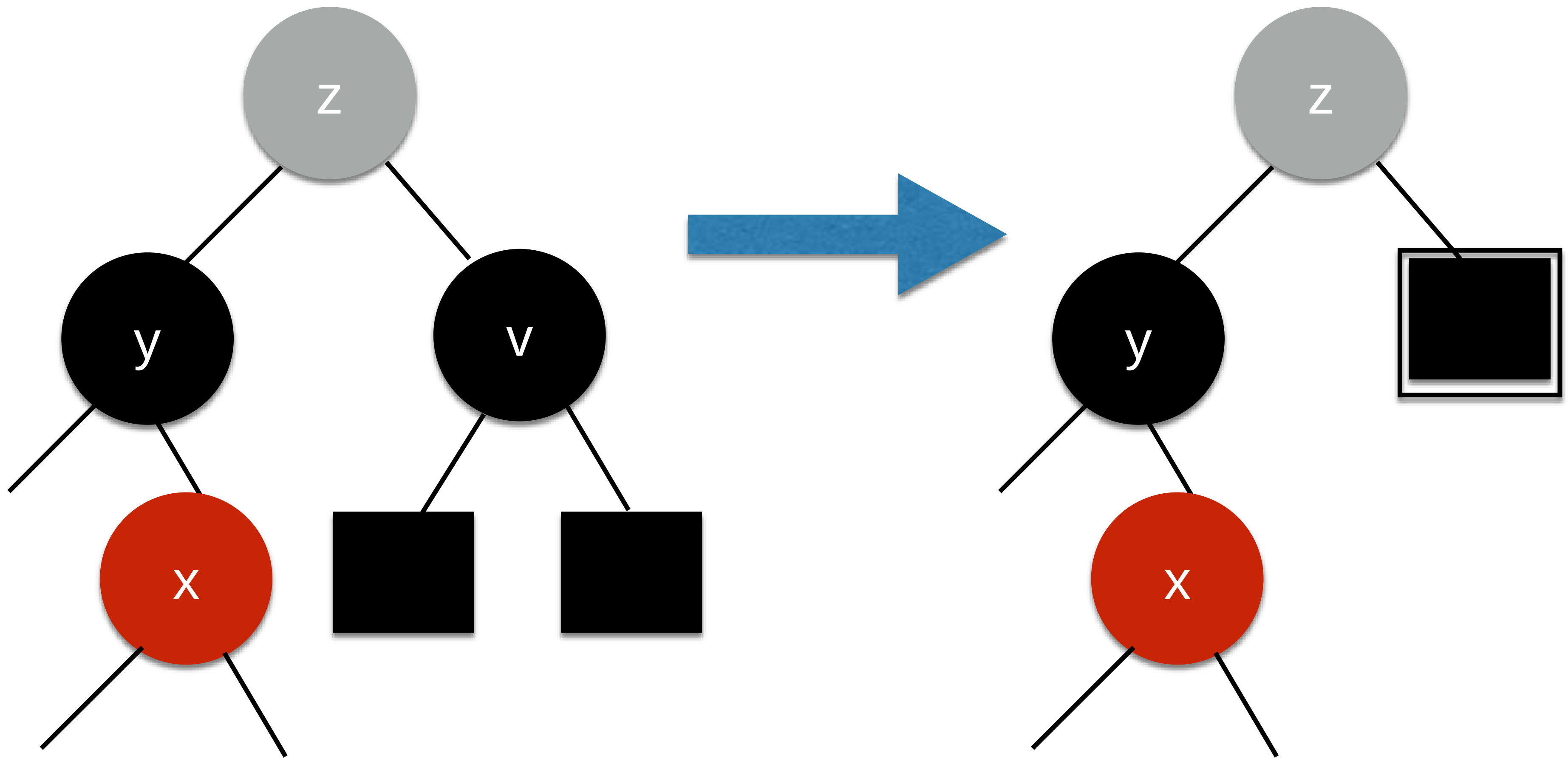
Deletions in RB Trees

- Let's make the picture a little bigger to bring more characters
- Remember, **v** is to be removed
- **x** could also be the left child



Case 2.a.1: **y** is black and <u>has a red child</u>

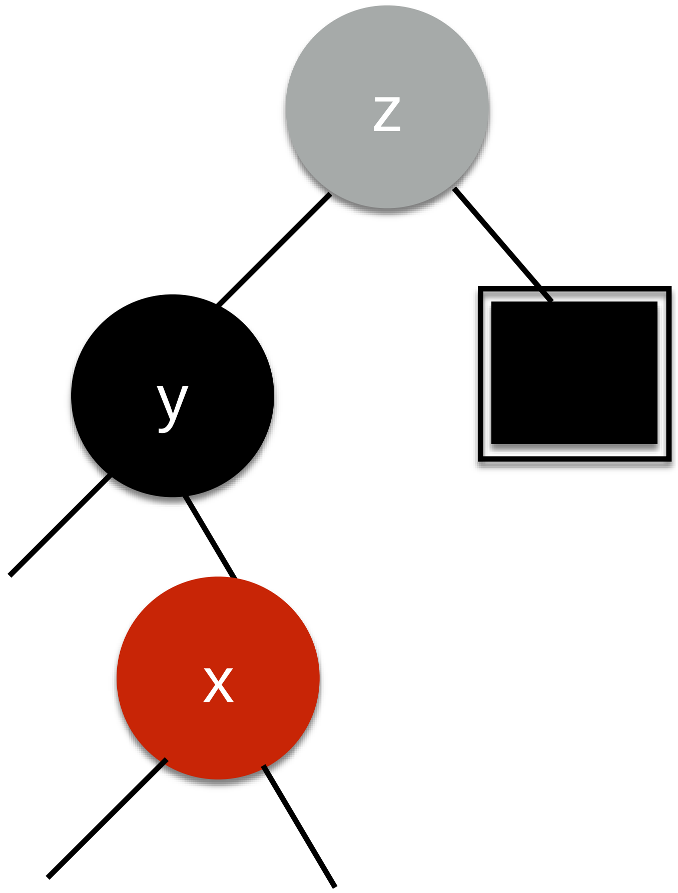
Deletions in RB Trees



- Remove **v,** color the leaf **double-black**
- double black is needed to preserve the depth property

Deletions in RB Trees

- So the depth property is preserved
- but we must get rid of double black

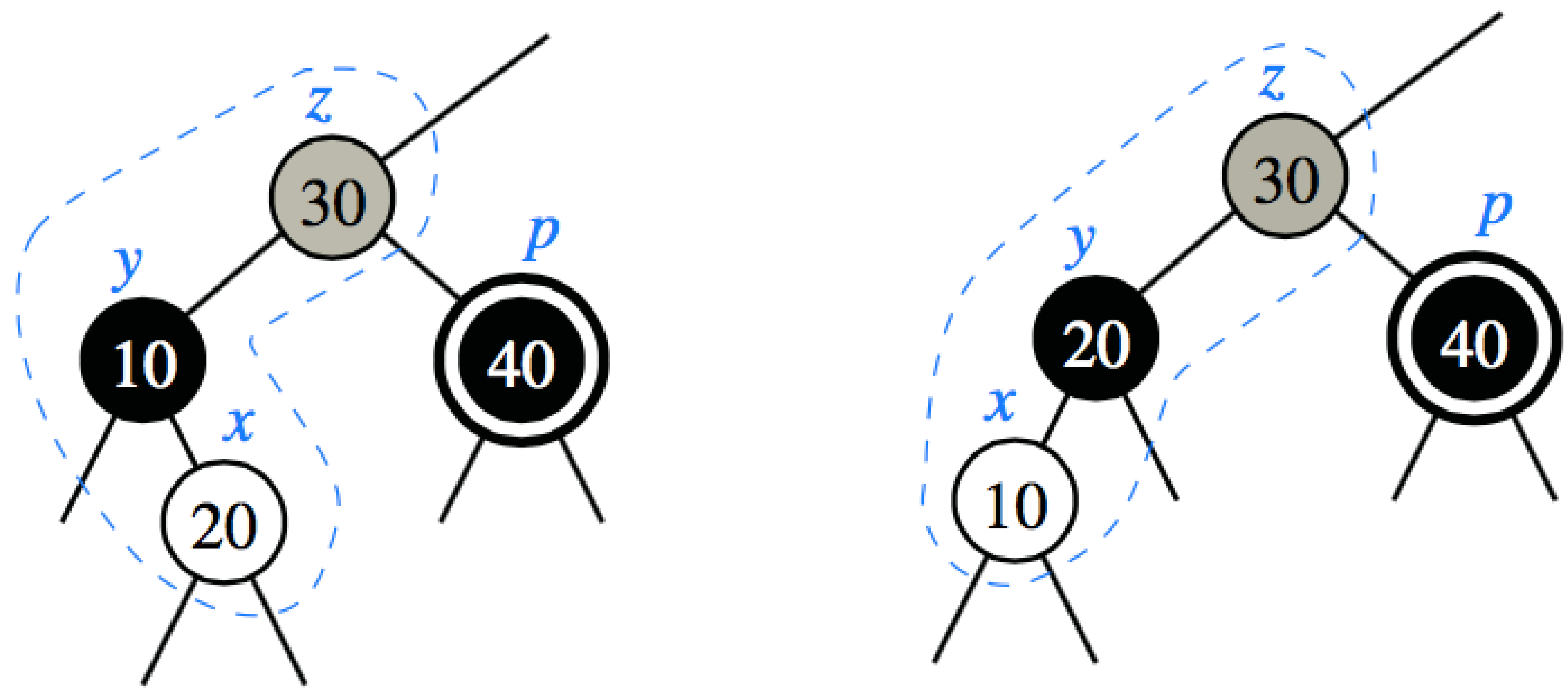


Getting rid of Double Black

- Let’s consider a double black at an arbitrary position **p**
- Three cases.

Getting rid of Double Black

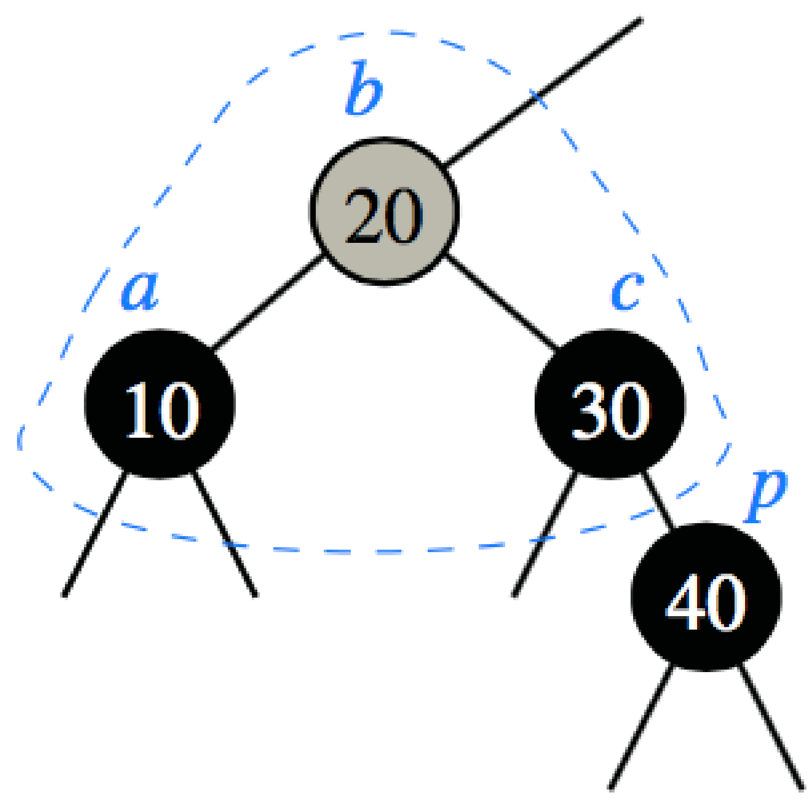
- **Case 2.a.1:** Sibling of **p** is black and has a red child



Restructure using **x**, **y** and **z**

Getting rid of Double Black

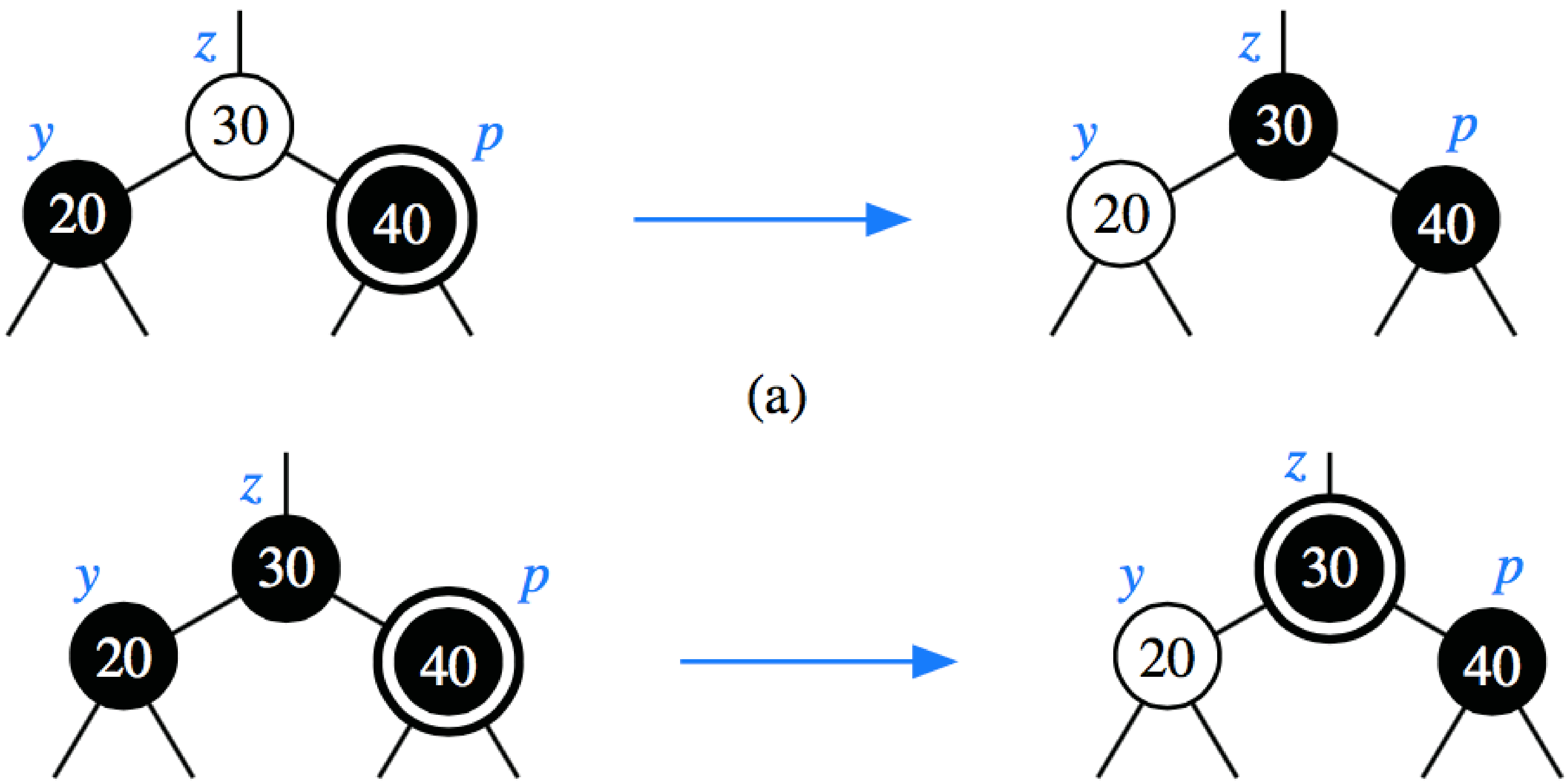
- **Case 2.a.1:** Sibling of **p** is black and has a red child



After Restructuring (where a, b and c represent x, y and z respectively)

Getting rid of Double Black

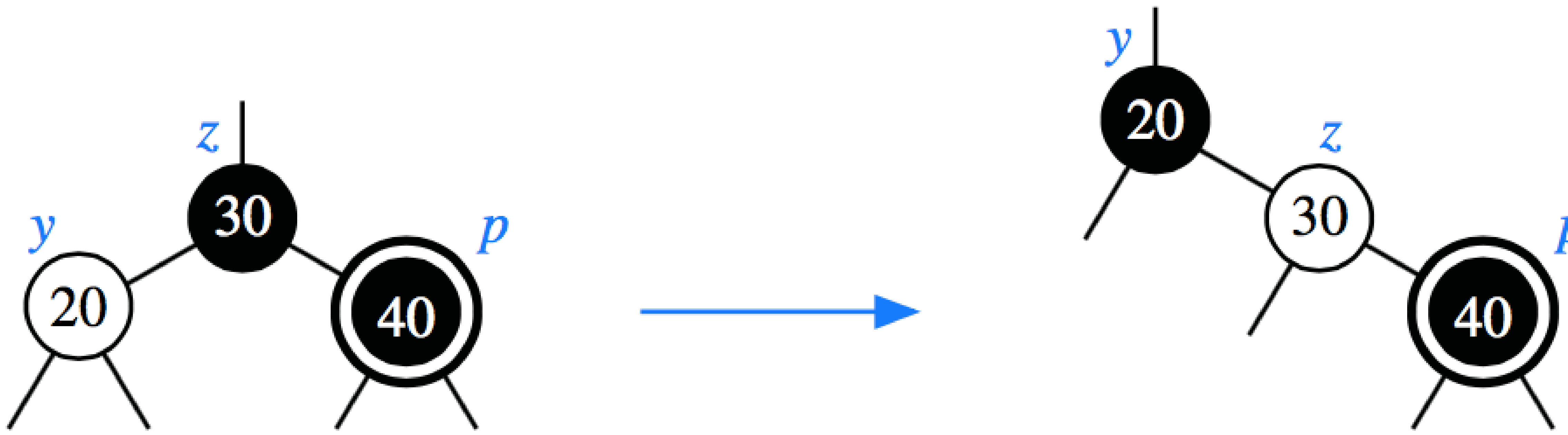
- **Case 2.a.2:** Sibling of **p** is black, whose both children are also **black**



Solved using recoloring

Getting rid of Double Black

- **Case 2.a.3:** Sibling of **p** is red



Solved using rotation and recoloring

After this the new sibling of p will be black and so …

How long does it take?

- Finding the node to delete – O(log n)
- Swapping and deletion – O(1)
- Each individual fix (rotation, restructuring and recoloring) – O(1)
- In worst case, double black problem can cascade until the root – O(log n)
- Therefore **O(log n)**