Data Structures and Algorithms I SCS1308 - CS

Dr. Dinuni Fernando

Senior Lecturer

Lecture 13

UCSC

Learning outcomes

In this topic, we will cover:

- Functionalities of Red-black trees.
- Tree balancing.

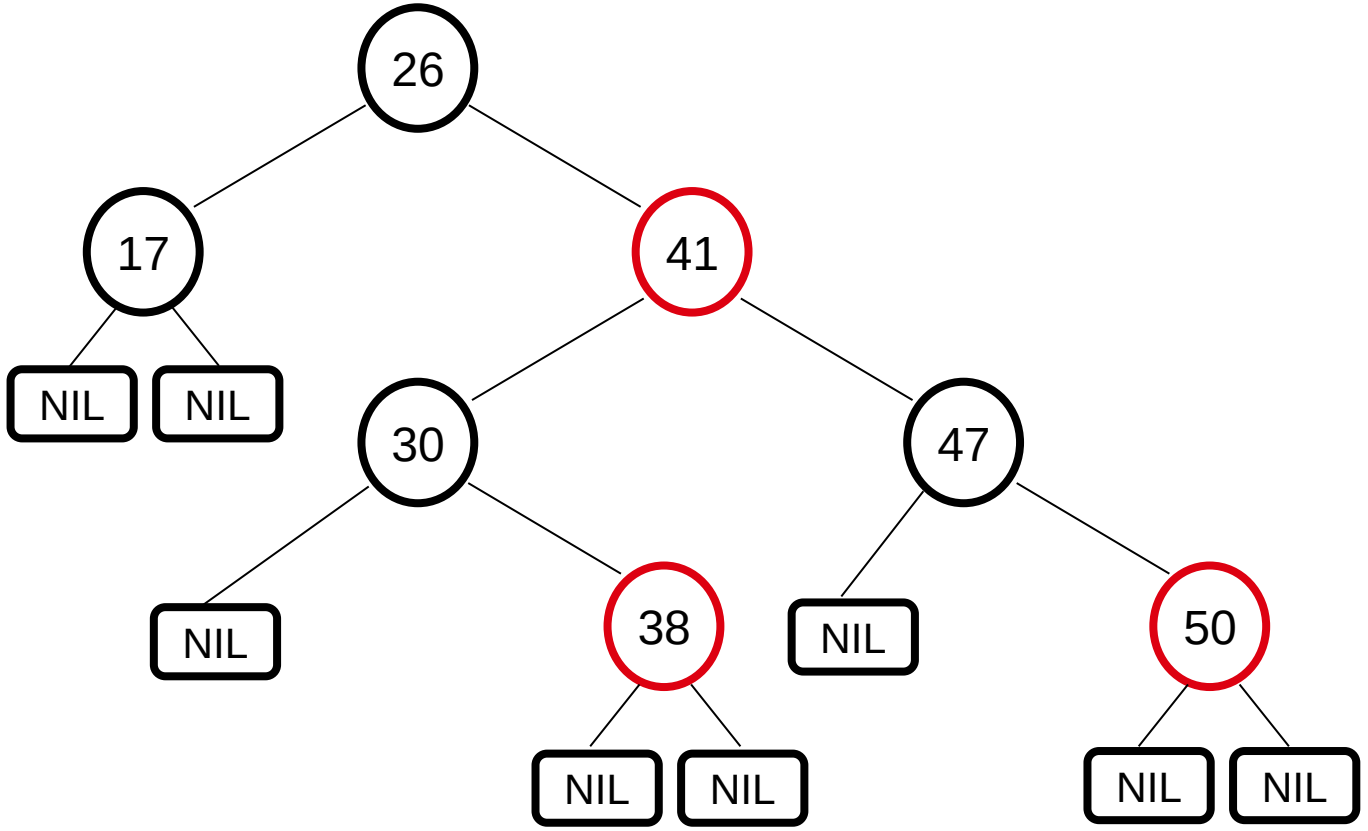
What is a Red-Black Tree

- Binary search tree with an additional attribute for its nodes: color which can be red or black.

- Constrains the way nodes can be colored on any path from the root to a leaf.

- Balanced tree → $O(\log N)$

Example : Red-Black Tree



- For convenience, we add NIL nodes and refer to them as the leaves of the tree.
  - Color[NIL] = BLACK

Definition : Red Black Tree

- A red-black tree is a binary search tree with an extra bit of storage per node.
- The extra bit represents the color of the node. It's either red or black.
- Each node contains the fields: color, key, left, right
- Any nil pointers are regarded as pointers to external nodes (leaves) and key bearing.

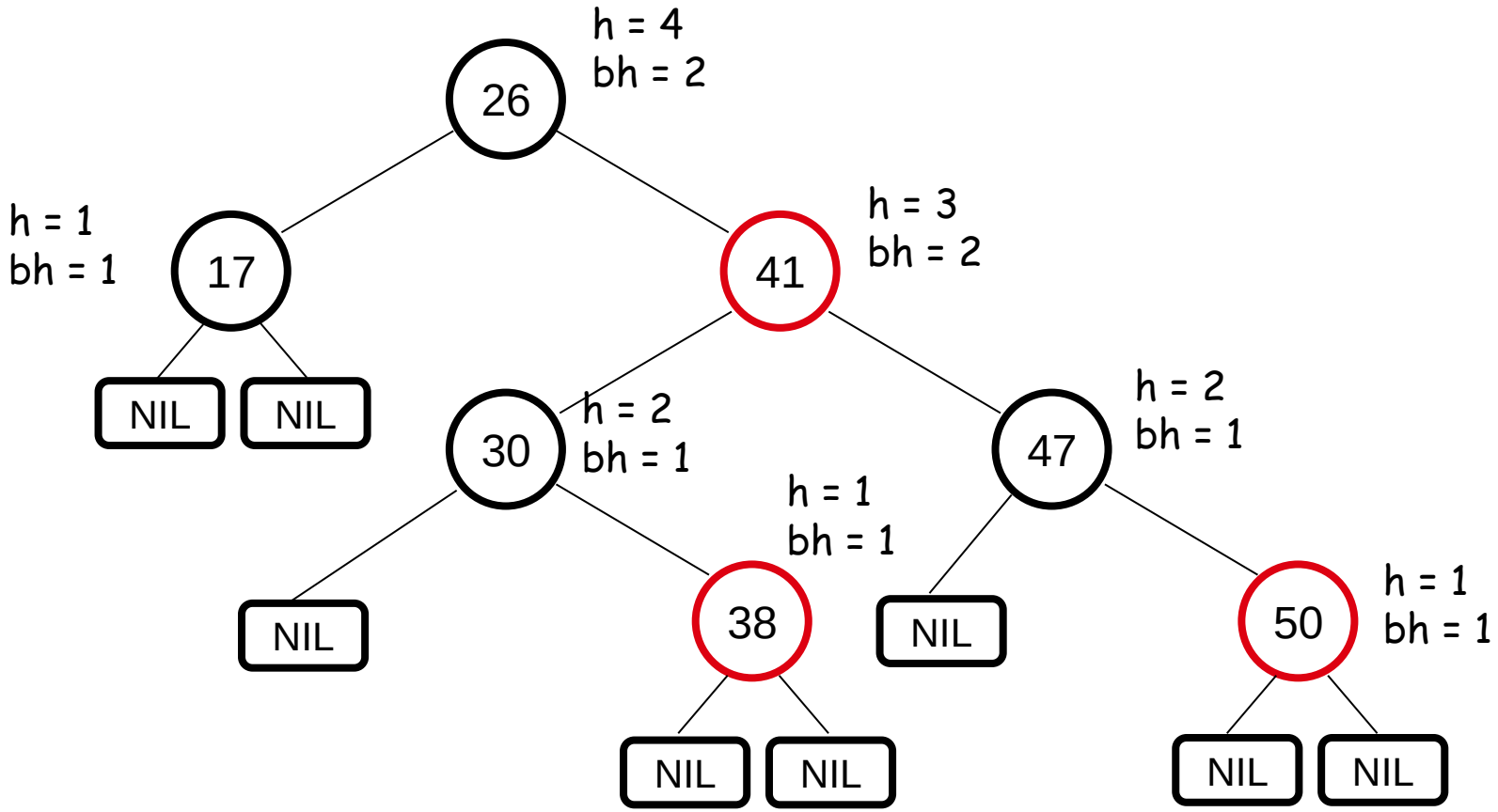
Properties

1. Every node is either red or black.
2. The root is black, newly added nodes are red.
3. Every leaf (nil) is black.
4. If a node is red then both of its children are black.
   - No two consecutive red nodes on a simple path from the root to a leaf
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Implication

- Red-black tree properties:
- It can be shown (with a proof by induction) that the tree has a height no more than 2 * log (n + 1).
- Thus, worst case lookUp, insert, delete are all – O(logn)

Definitions



- **Height of a node:** the number of edges in the **longest** path to a leaf.
- **Black-height** bh(x) of a node x: the number of black nodes (including NIL) on the path from x to a leaf, not counting x.

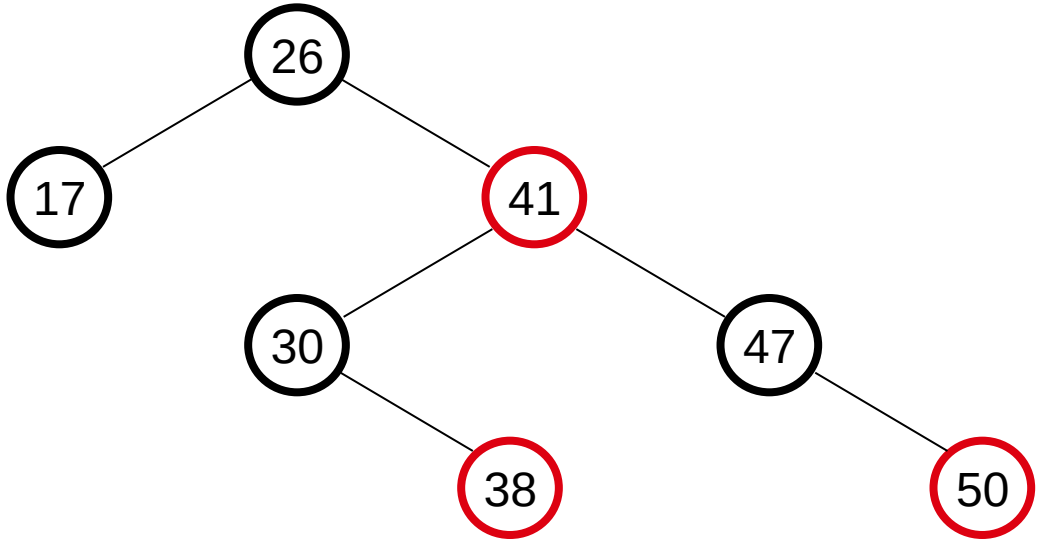
Operations on Red-Black Trees

- Non-modifying binary search tree operations (Retrieve, getnextItem) run in O(logn)
- What about Insert and delete ?
  - Still run on O(logn)
  - Need to guarantee that modified tree will be a valid red-black tree.

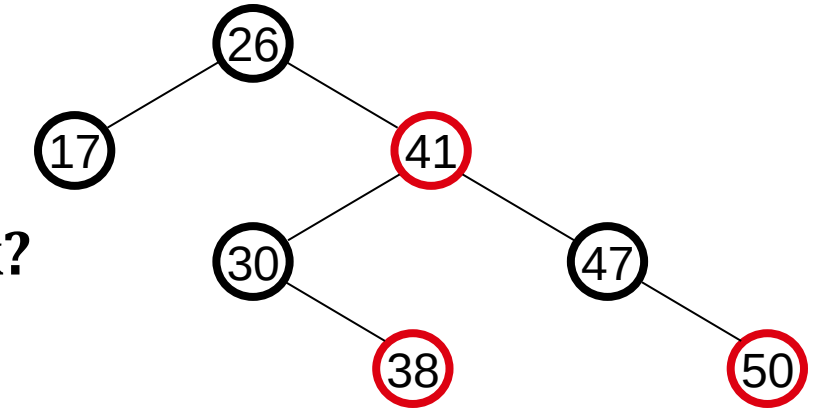
Insert a new node

What color to make the new node?

- Red? Let's insert 35!
  - Property 4 is violated: if a node is red, then both its children are black
- Black? Let's insert 14!
  - Property 5 is violated: all paths from a node to its leaves contain the same number of black nodes



Delete a node



What color was the node that was removed? **Black?**

1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

OK!

Not OK! If removing the root and the child that replaces it is **red**

OK!

Not OK! Could change the black heights of some nodes

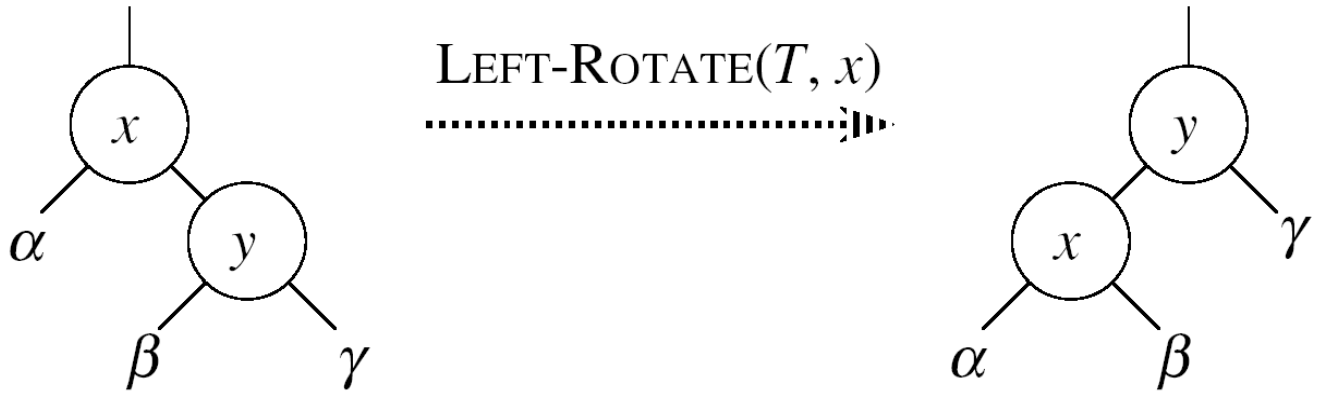
Not OK! Could create two red nodes in a row

Rotations

- Operations for re-structuring the tree after insert and delete operations
  - Together with some node re-coloring, they help restore the red-black-tree property
  - Change some of the pointer structure
  - **Preserve** the binary-search tree property
- Two types of rotations:
  - Left & right rotations

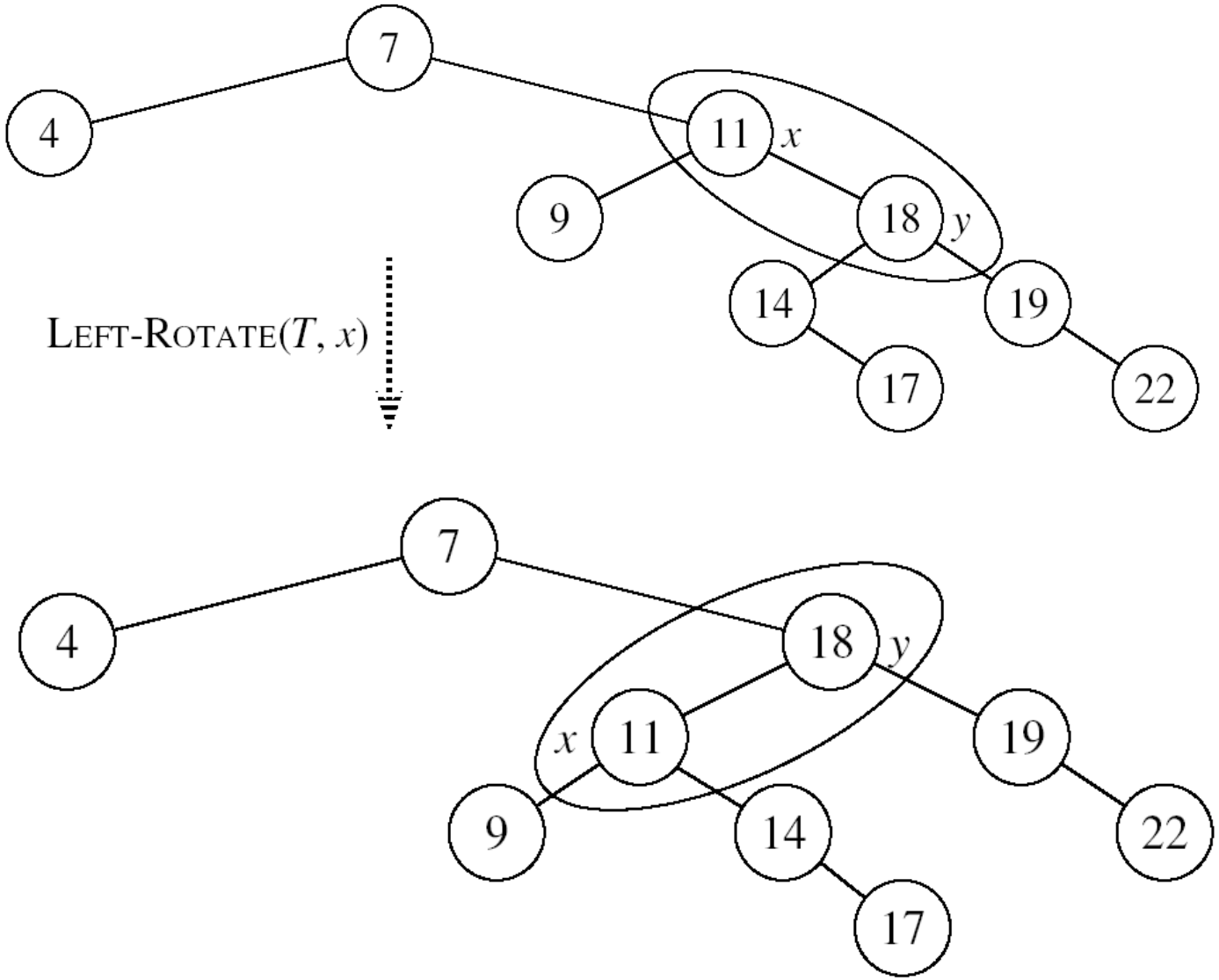
Left Rotation

- Assumptions for a left rotation on a node x:
  - The right child y of x is not NIL



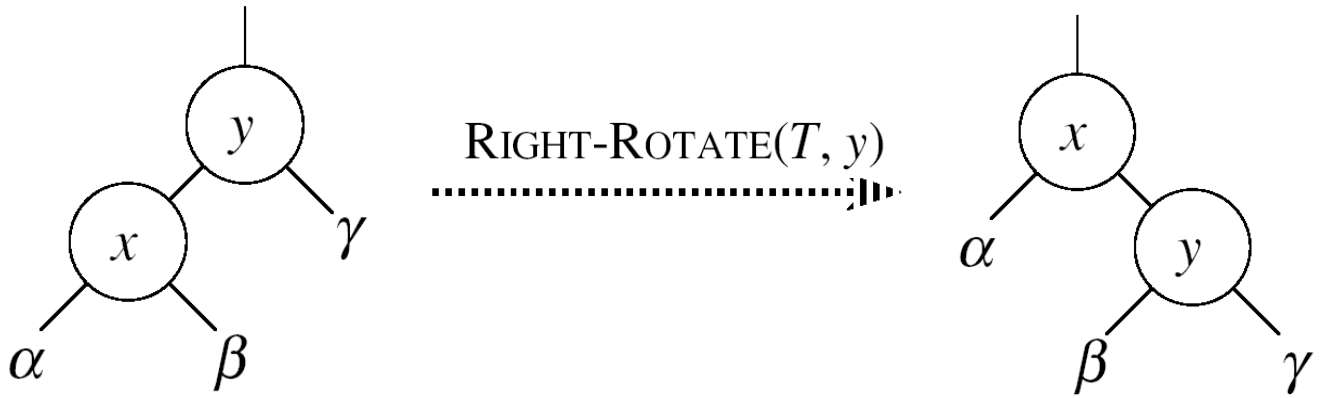
- Idea:
  - Pivots around the link from x to y
  - Makes y the new root of the subtree
  - x becomes y's left child
  - y's left child becomes x's right child

Example : Left-Rotate



Right Rotate

- Assumptions for a right rotation on a node x:
  - The left child x of y is not NIL



- Idea:
  - Pivots around the link from y to x
  - Makes x the new root of the subtree
  - y becomes x's right child
  - x's right child becomes y's left child

Insert a node

- Goal:
  - Insert a new node z into a red-black tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black tree properties
    - i.e., use RB-INSERT-FIXUP

RB Properties Affected by Insert

1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

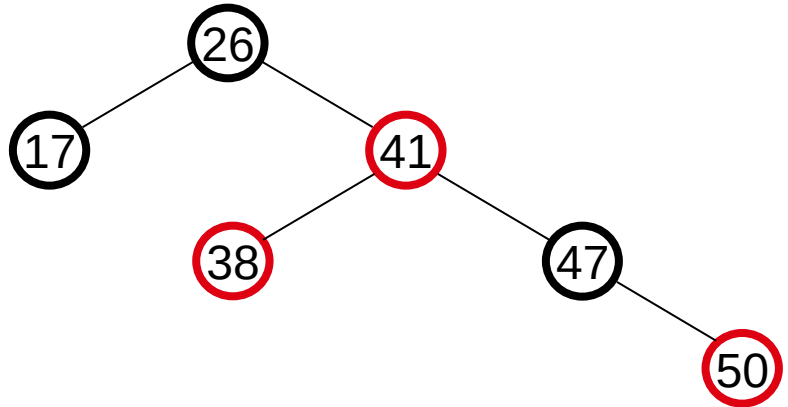
OK!

If z is the root
⇒ not OK

OK!

If p(z) is red ⇒ not OK
z and p(z) are both red

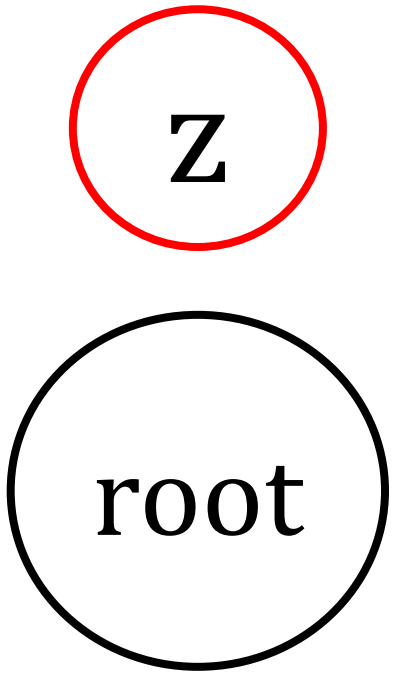
OK!



RB- FIXUP – Case 0

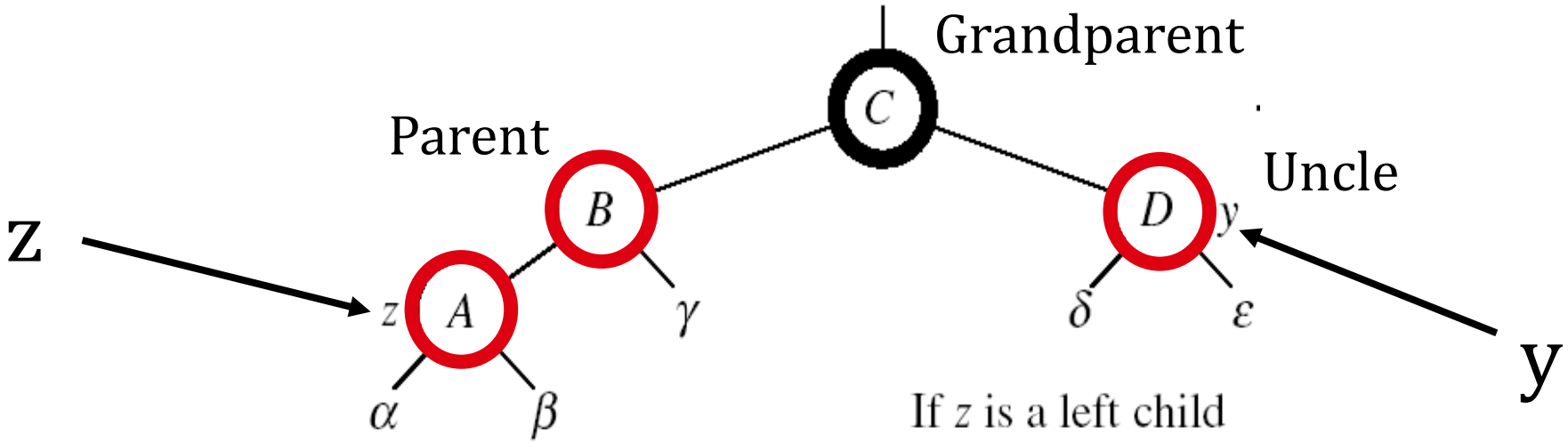
Case 0

- Insert node Red
  - If its root – change the colour to **black**



RB- FIXUP – Case 1

- z's "uncle" (y) is red (z could be either left or right child)



Solution: Recolor Parent, grand parent, and uncle

Case 1

Recolor Parent, grand parent, and uncle

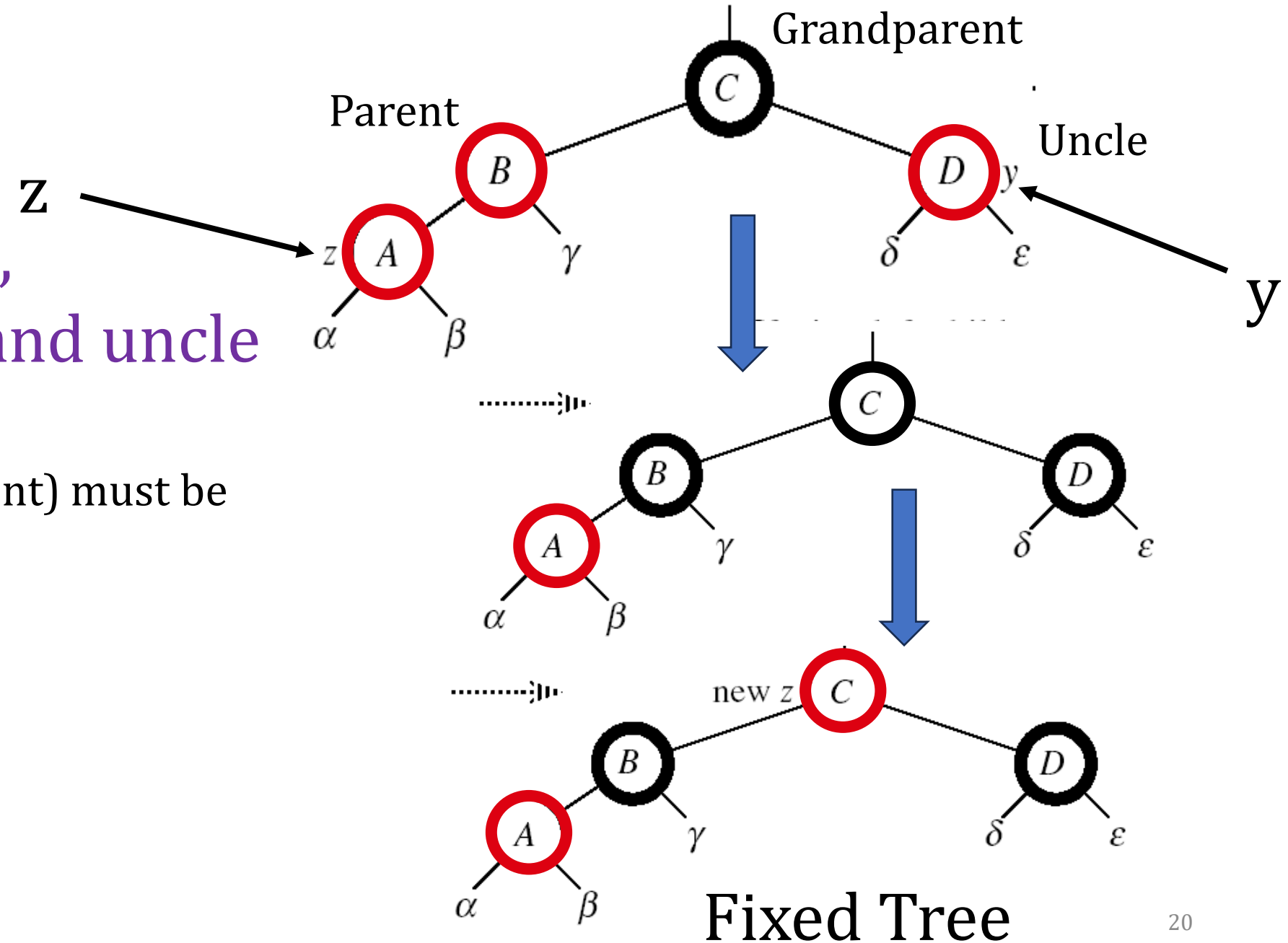
p[p[z]] (z's grandparent) must be black.

color p[z] ← **black**

color y ← **black**

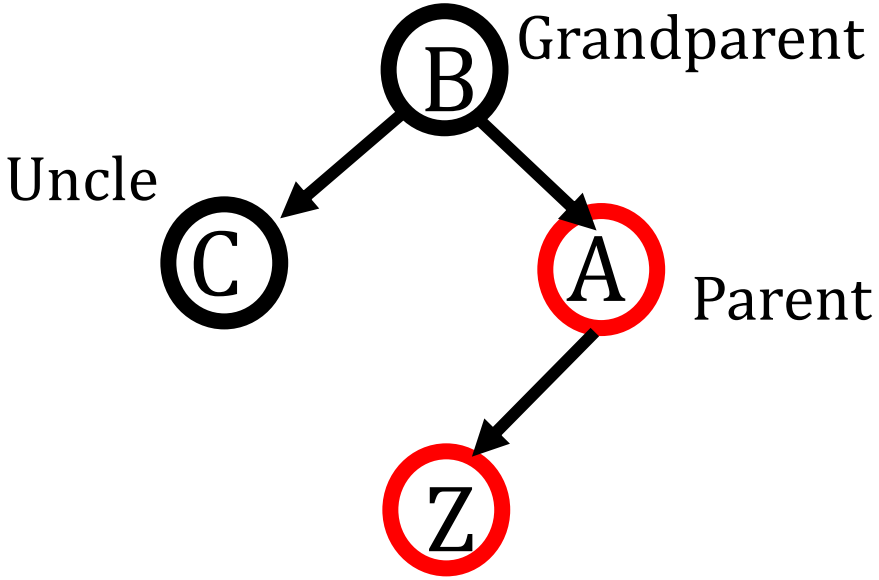
color p[p[z]] ← **red**

z = p[p[z]]

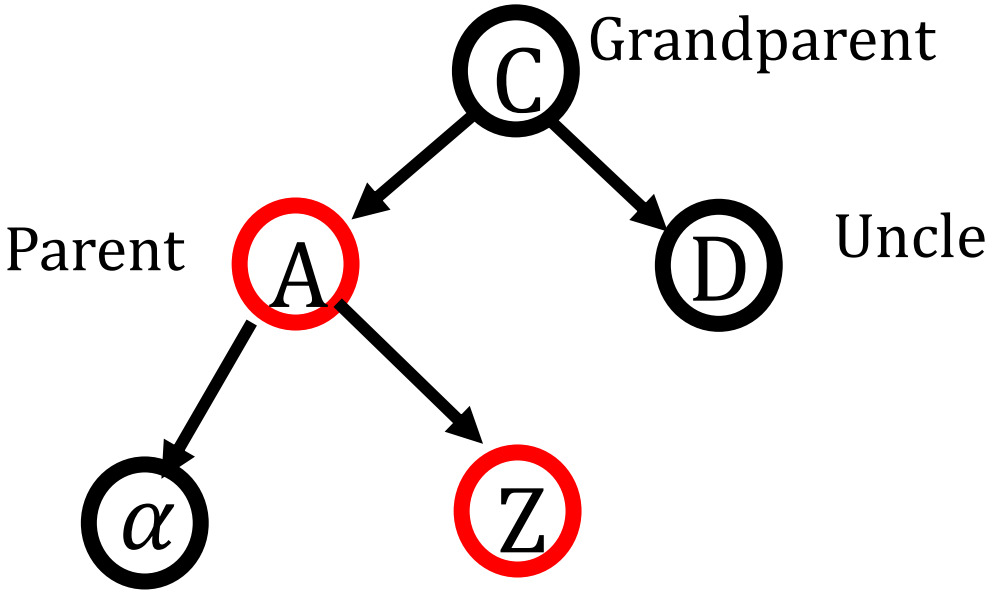


RB- FIXUP – Case 2

- **z**'s "uncle" (y) is **black**
- Z is in a triangle (parent, grand parent forms a triangle)



(i)

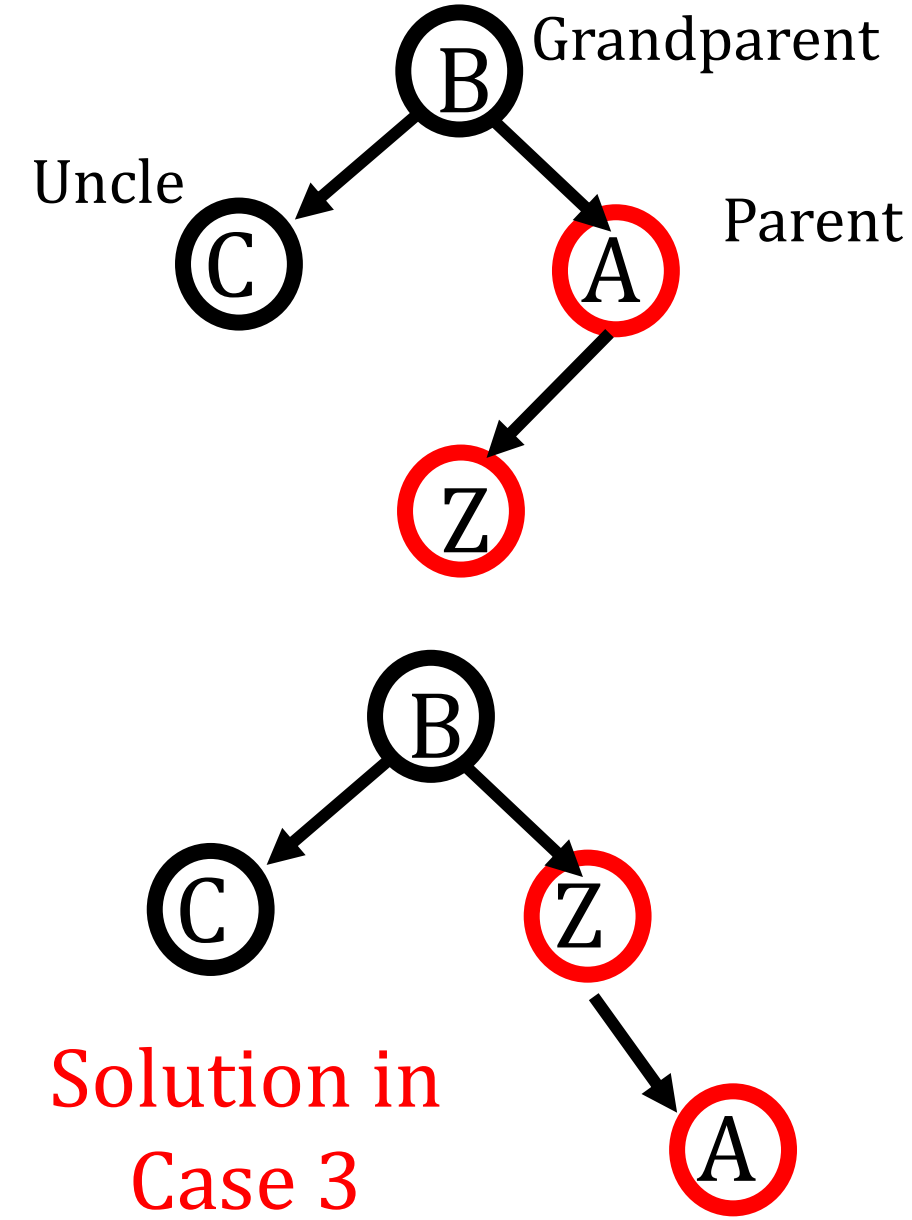


(ii)

RB- FIXUP – Case 2 –(i)

- **z**'s "uncle" (y) is **black**
- **z** is a left child

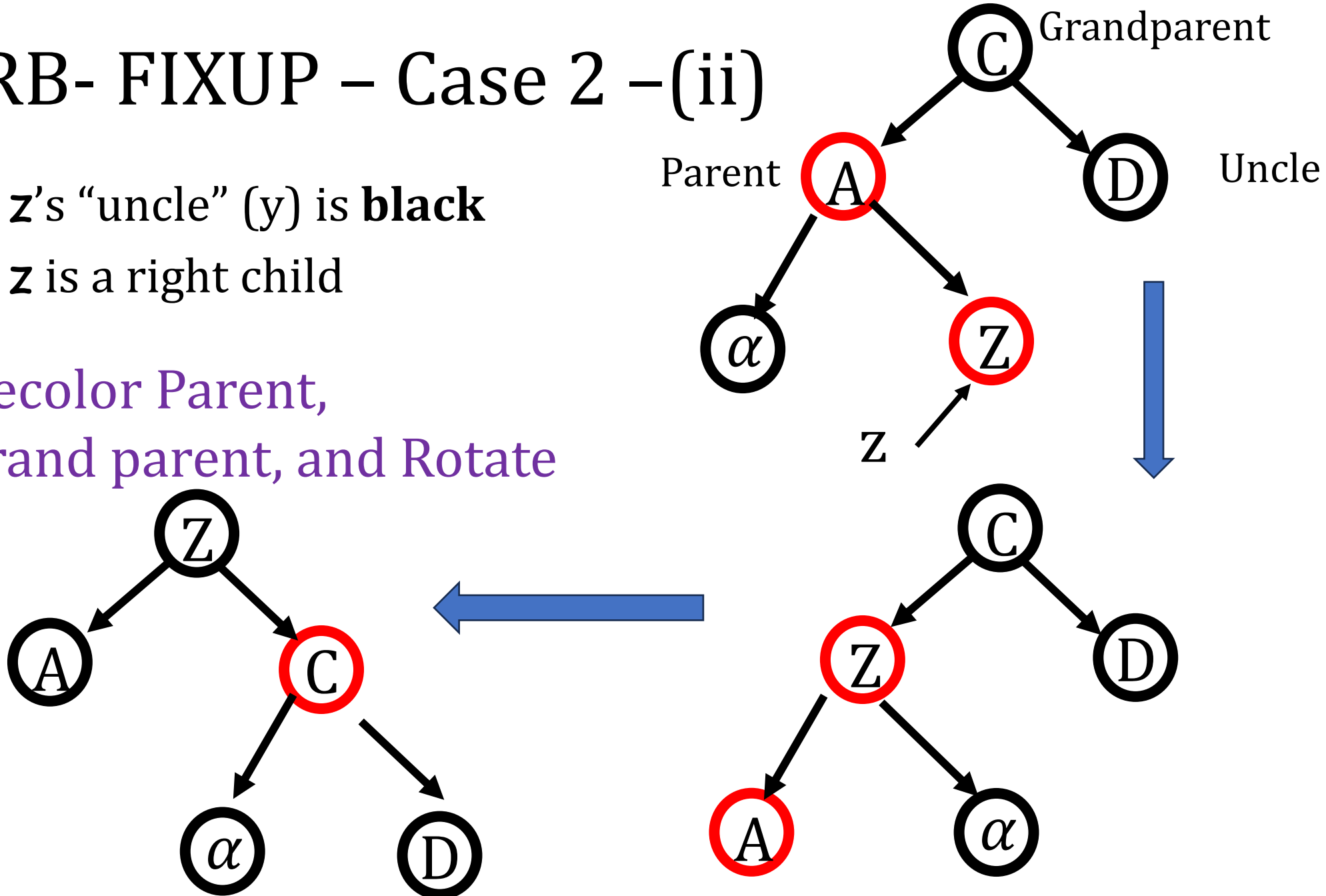
- Now Z is in a line



RB- FIXUP – Case 2 –(ii)

- **z**'s "uncle" (y) is **black**
- **z** is a right child

Recolor Parent,
grand parent, and Rotate



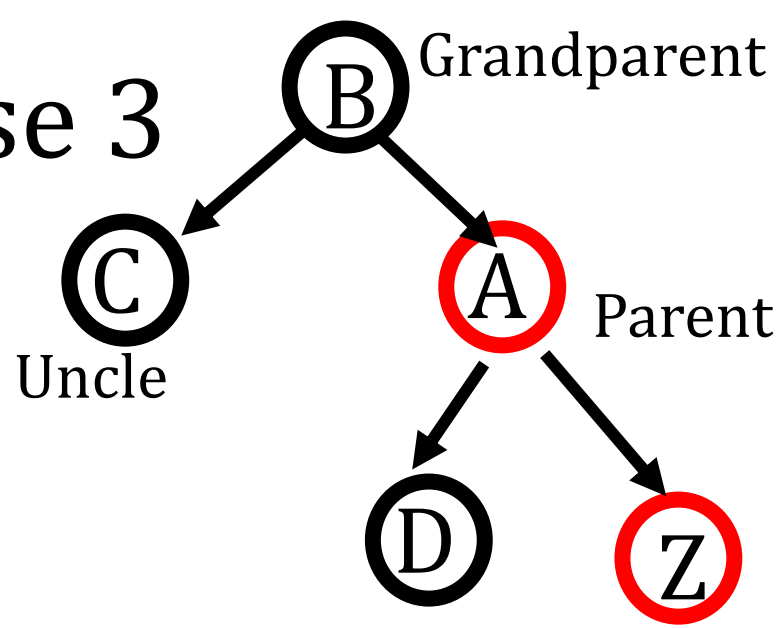
RB- FIXUP – Case 2

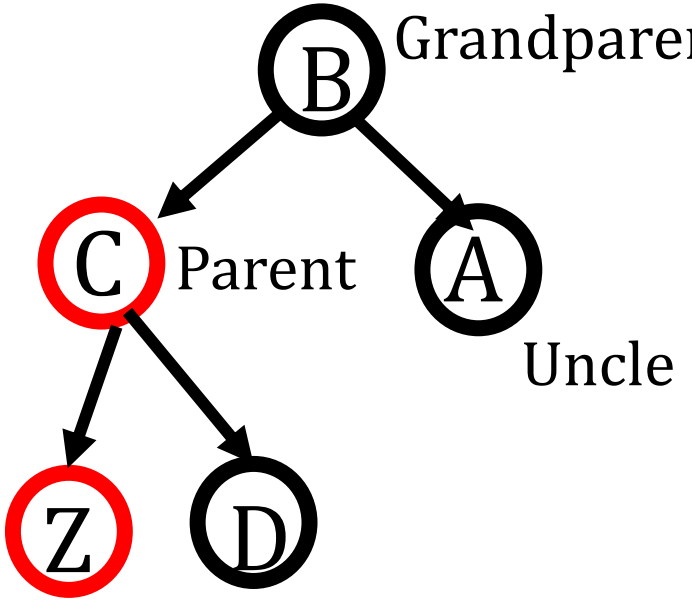
- z ← p[z]
- LEFT-ROTATE(T, z)

→ now z is a left child, and both z and p[z] are red

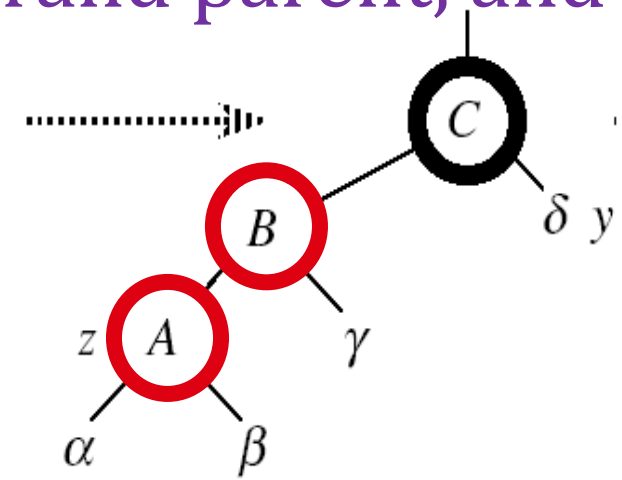
RB- FIXUP – Case 3

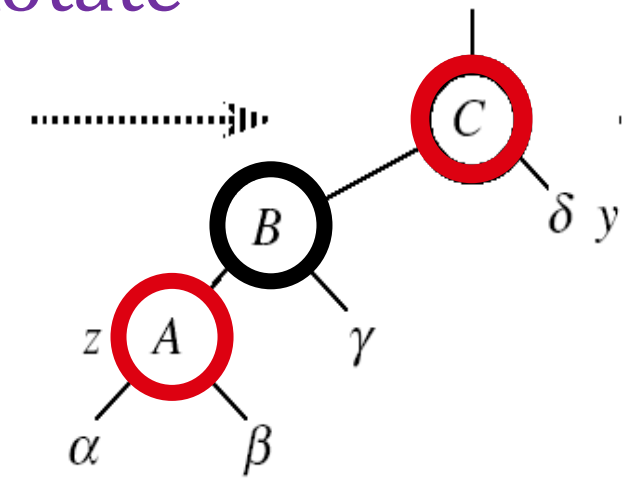
- **z**'s "uncle" (y) is **black**
- **z** is a left child
- Z is in a line

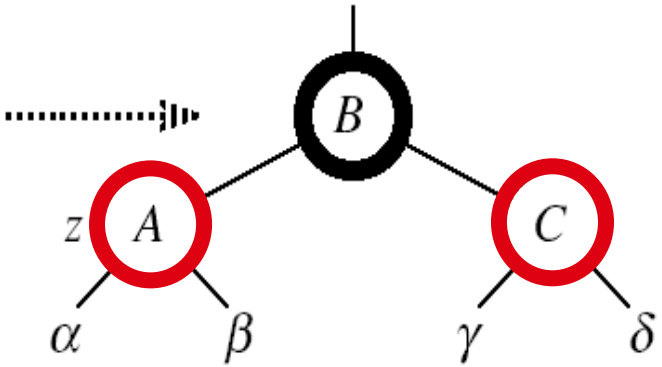




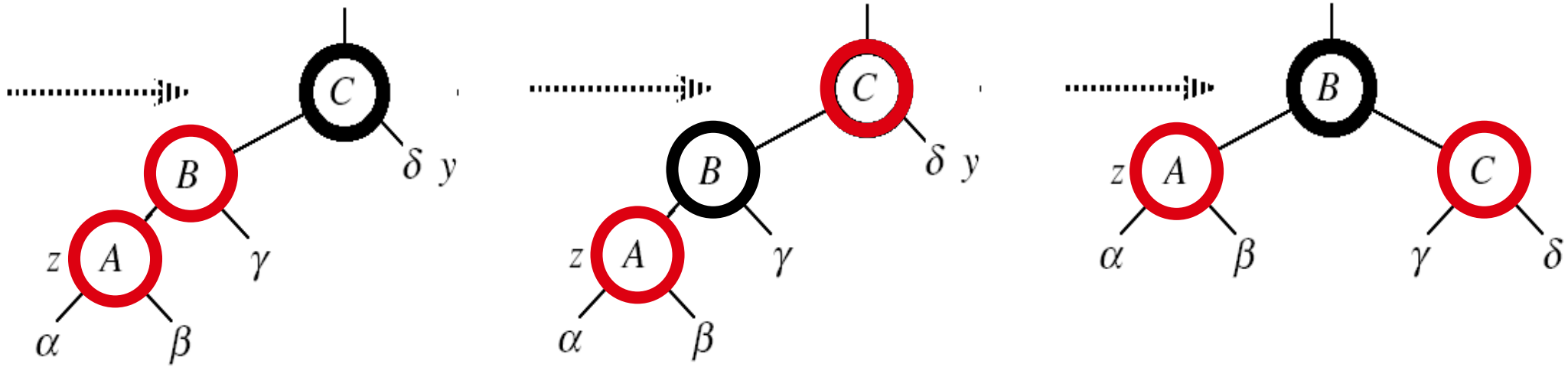
Recolor Parent,
grand parent, and Rotate





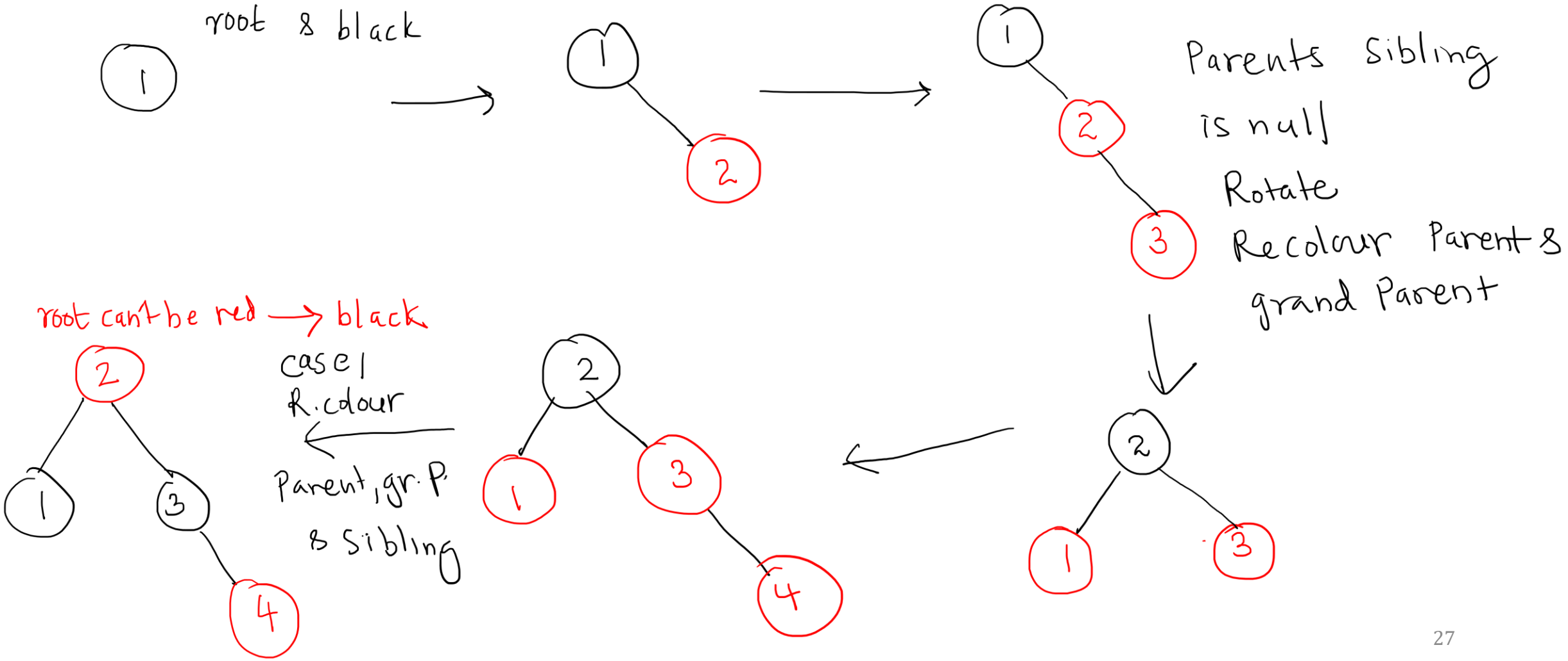


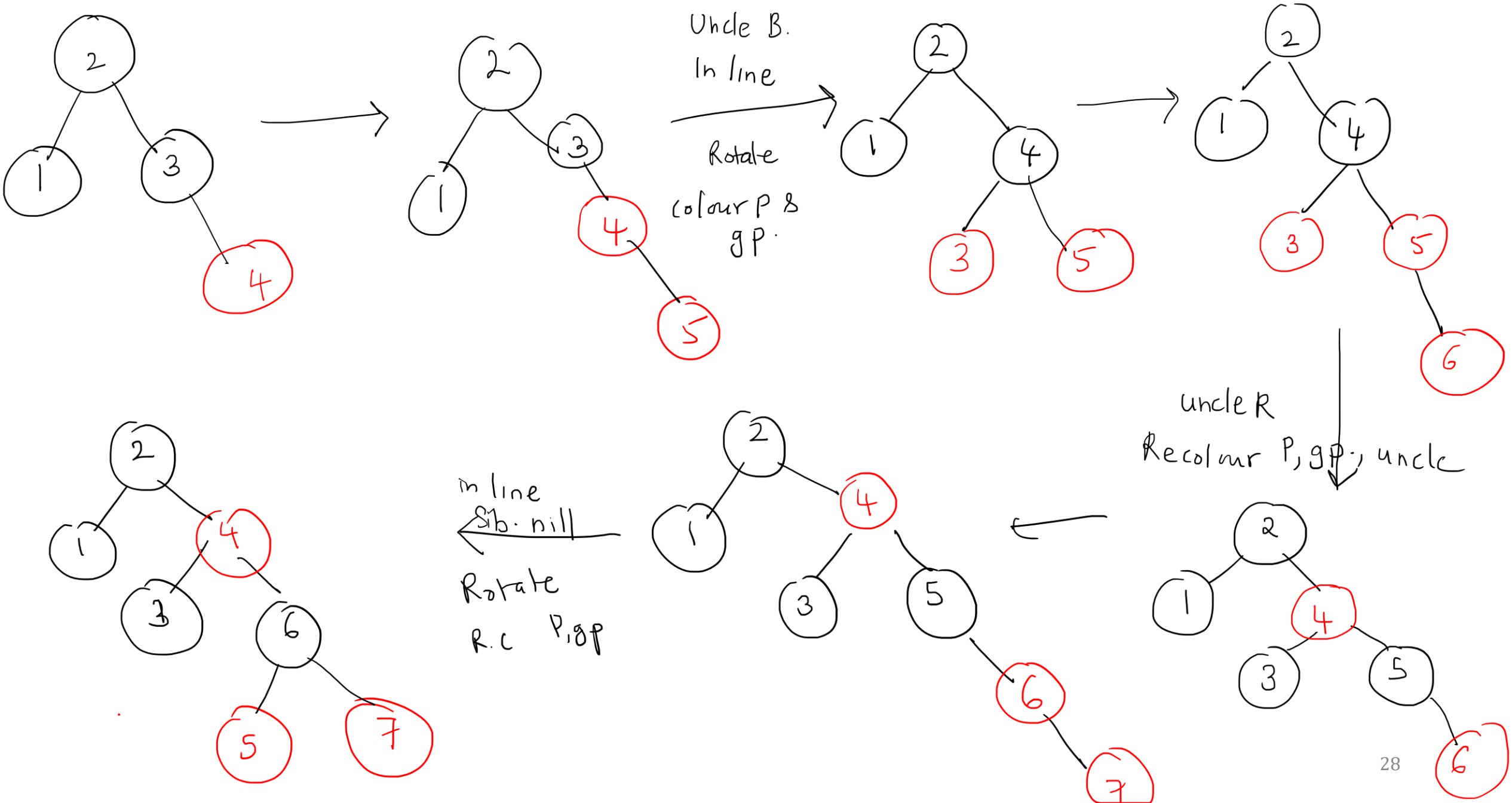
RB- FIXUP – Case 3



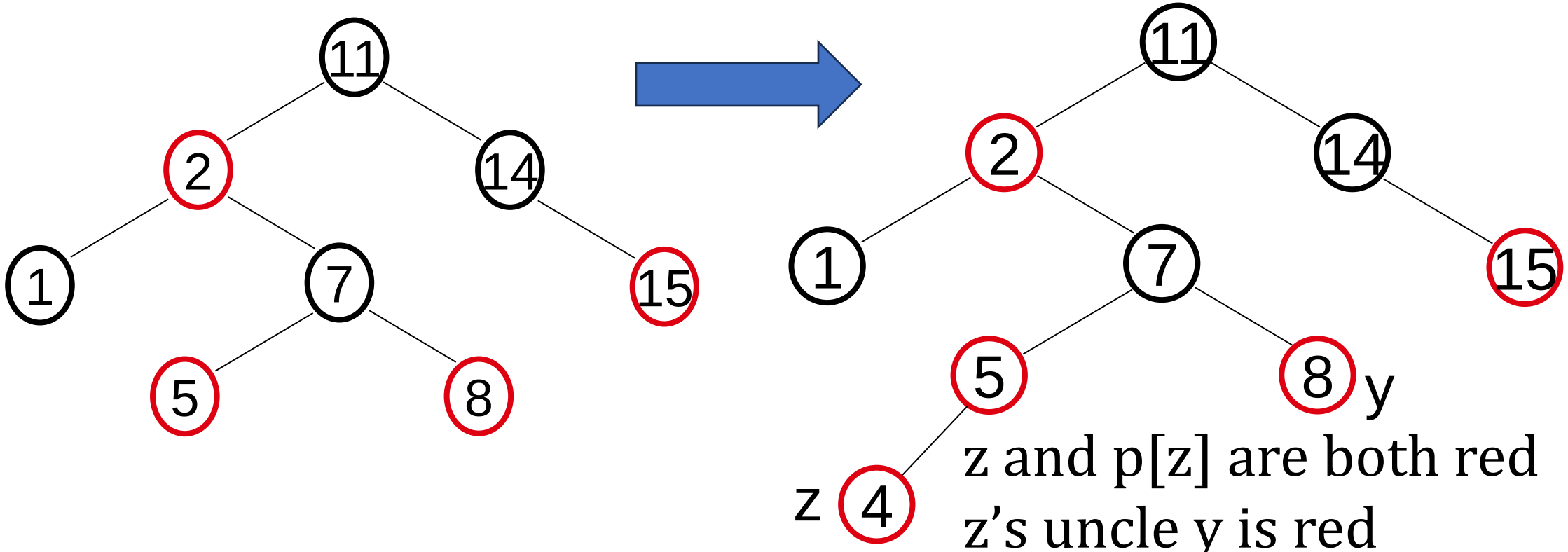
- color p[z] ← black
- color p[p[z]] ← red
- RIGHT-ROTATE(T, p[p[z]])
- No longer have 2 reds in a row
- p[z] is now black

Example Insert 1-7 nodes to RB tree

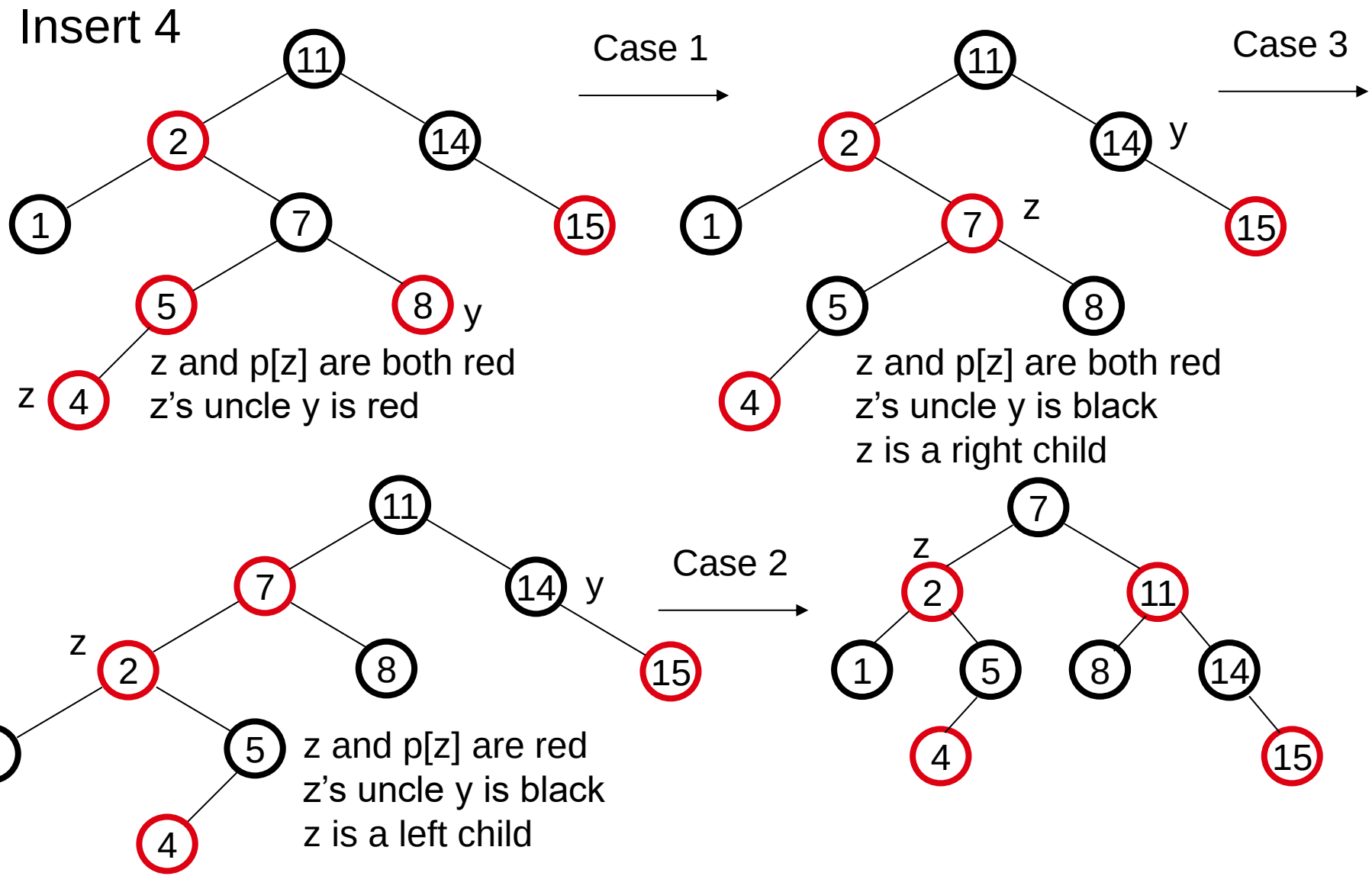



Uncle B.
In line
Rotate
colour P & gP.
uncle R
Recolour P, gP; uncle
In line
Sib. nill
Rotate
R.C P, gP

Homework : insert node 4



Example : insert node 4



Red Black Tree Deletion : Tricky

Upon deletion : If deleting node is

- Red node – no properties break
- **Black** node – end up with **Double Black** problem – missing black node along some path violates black-height property.

- How to fix ?
  - Color of the sibling of the double black node
  - Color of the sibling's children

Deletion -Algorithm

- Delete the node like in a regular Binary Search Tree (BST).
- Fix any violations of the Red-Black Tree properties using rotations and recoloring.

- Steps:
  - Perform standard BST deletion.
  - Fix any Red-Black Tree property violations:
    - Case 1: The sibling is red (recoloring and rotations).
    - Case 2: The sibling is black with black children (recoloring).
    - Case 3: The sibling is black with one red child (rotations and recoloring).

Types of Delete_fixes

- Suppose after deletion, violated node is x, its sibling is w

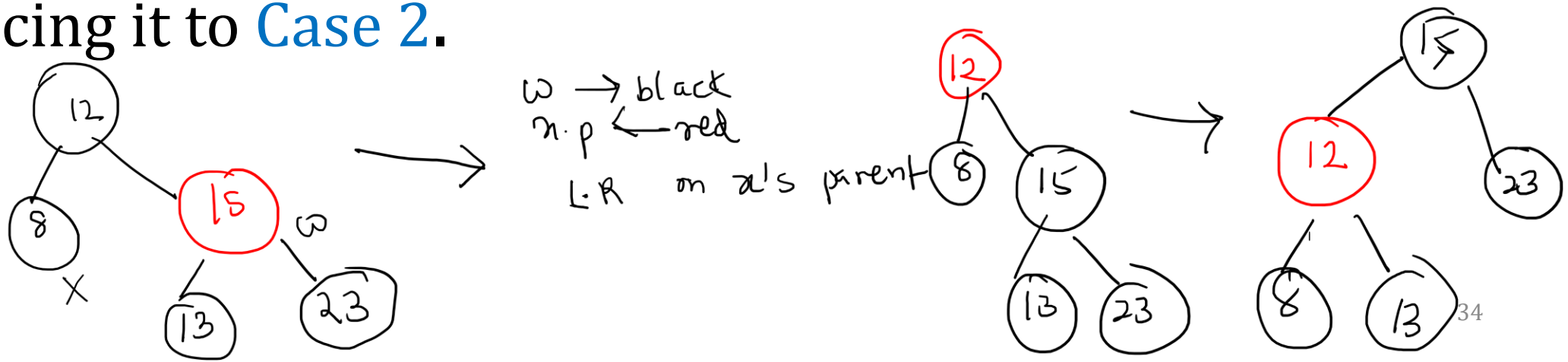
1. W is red
2. W is black, w.left and w.right is black
3. W is black, w.left is red and w.right is black
4. w. is black, w.right and w.left is red

Fixing violations during deletion

(w)

Case 1: Sibling is Red

- Rotate the **parent** towards the double black.
- Swap colors between the **parent** and **sibling**.
- This transforms the tree so the sibling becomes **black**, and now the double black node has a **black sibling**, reducing it to Case 2.



Fixing violations during deletion : Case 2: Sibling is Black

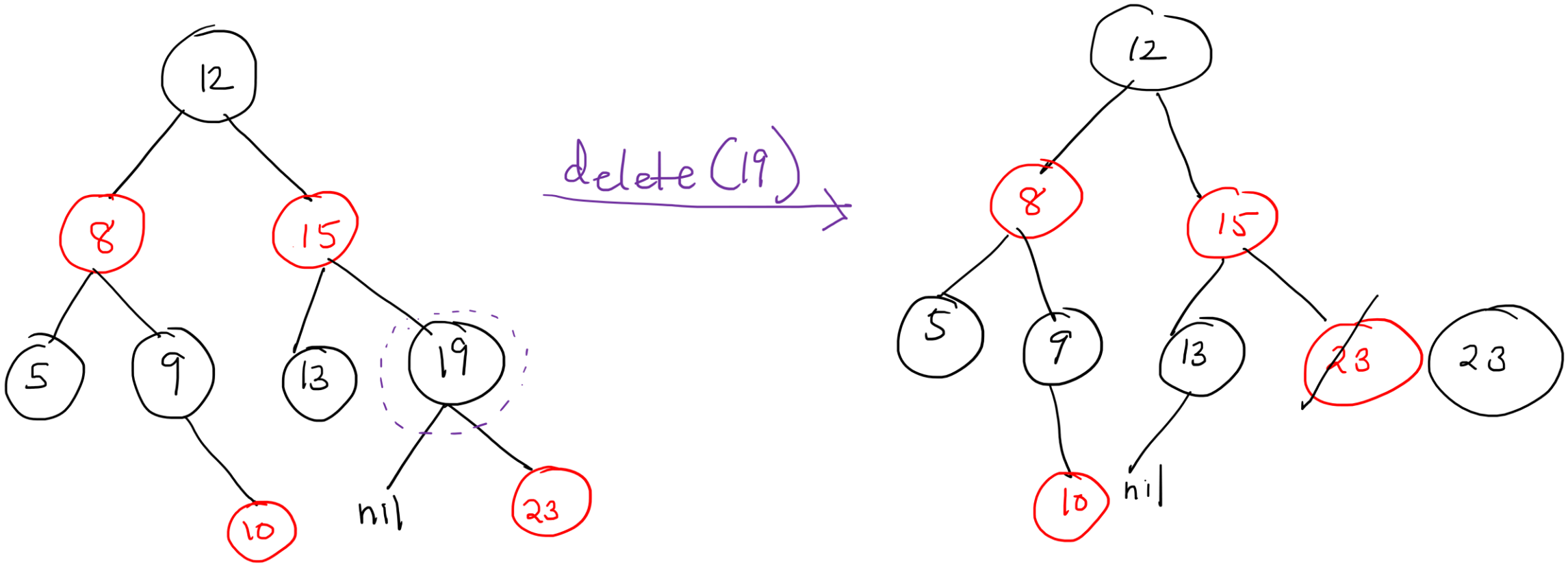
Sibling's children are black

- Recolor the sibling red.
- Move the double black issue up to the parent because we've decreased the number of black nodes below the sibling and parent.
- If the parent was red, recolor it black, and the process ends.
- If the parent was black, it becomes double black, and the process repeats upwards.

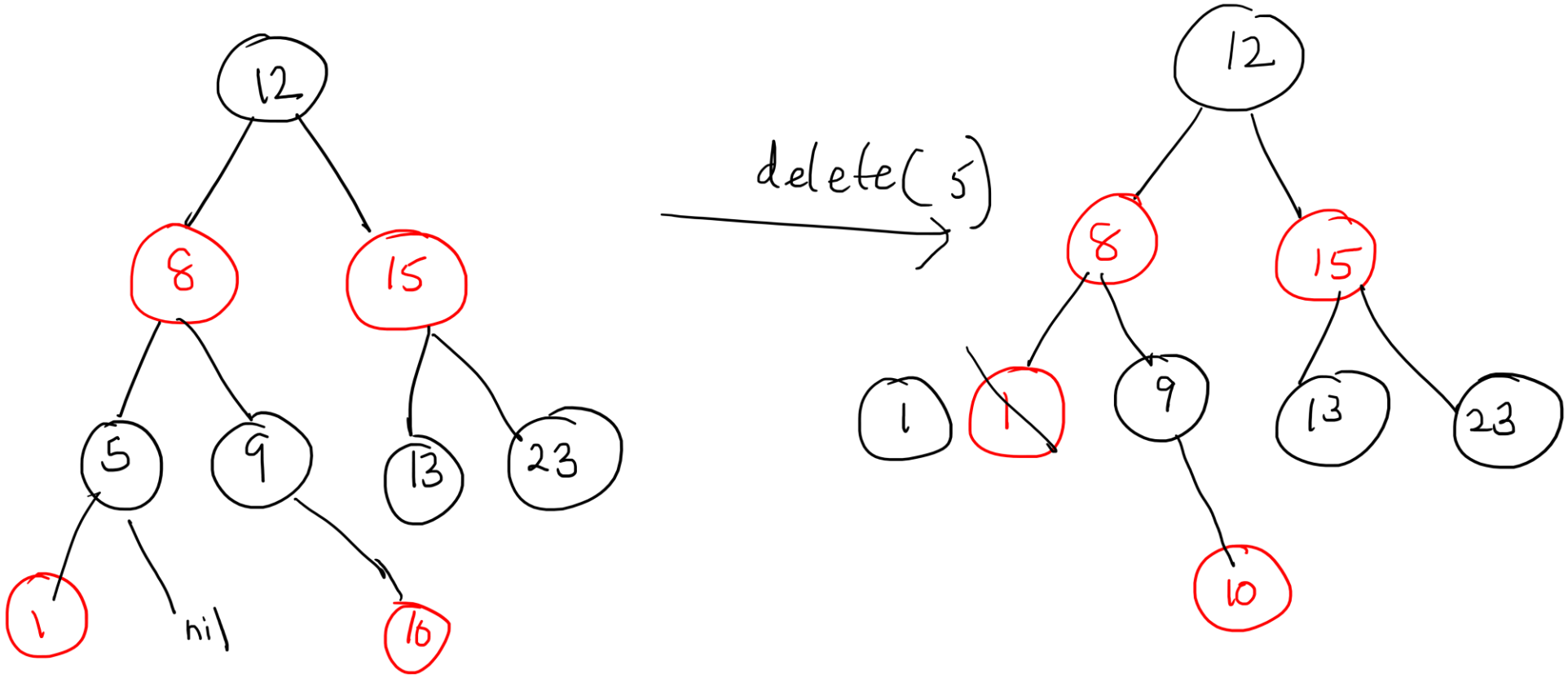
Sibling has at least one red child

- Red children allow us to **borrow** a black from the sibling's subtree through rotations and recoloring.
- If the far child (relative to the double black) is red:
  - Rotate around the parent.
  - Swap colors appropriately.
  - The double black is fixed immediately.
- If the near child is red:
  - First, rotate around the sibling to make the near child into the far child.
  - Then, apply the far child case.

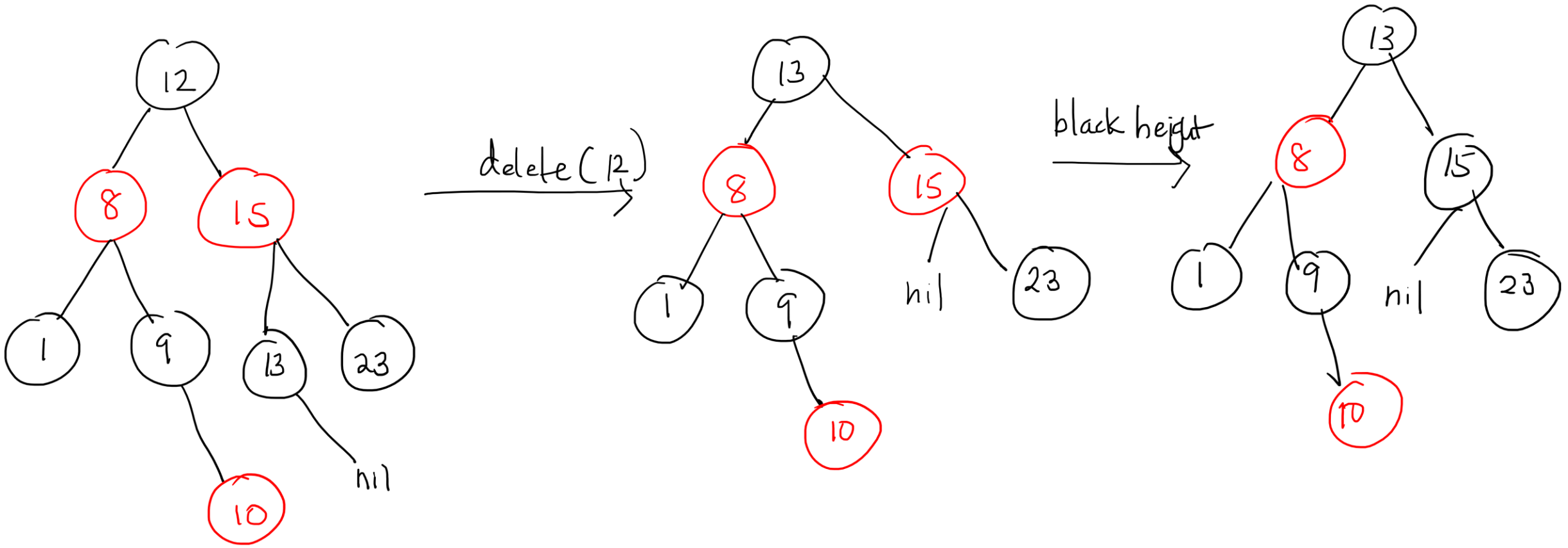
Scenario : 1 : left child is NIL



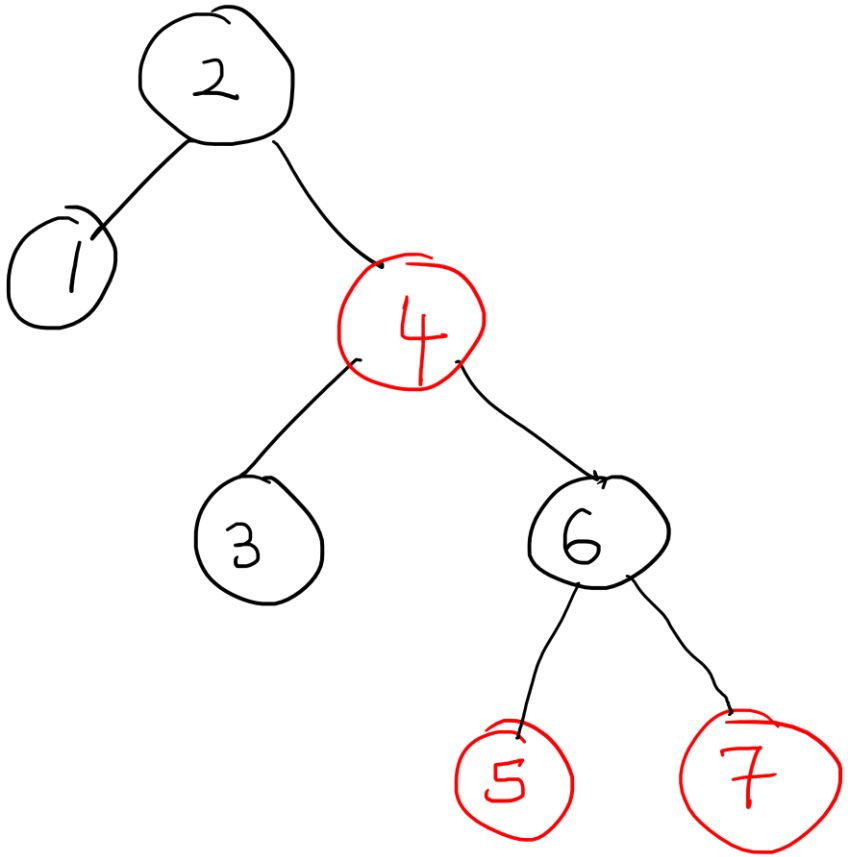
Scenario 2 : right child is NIL



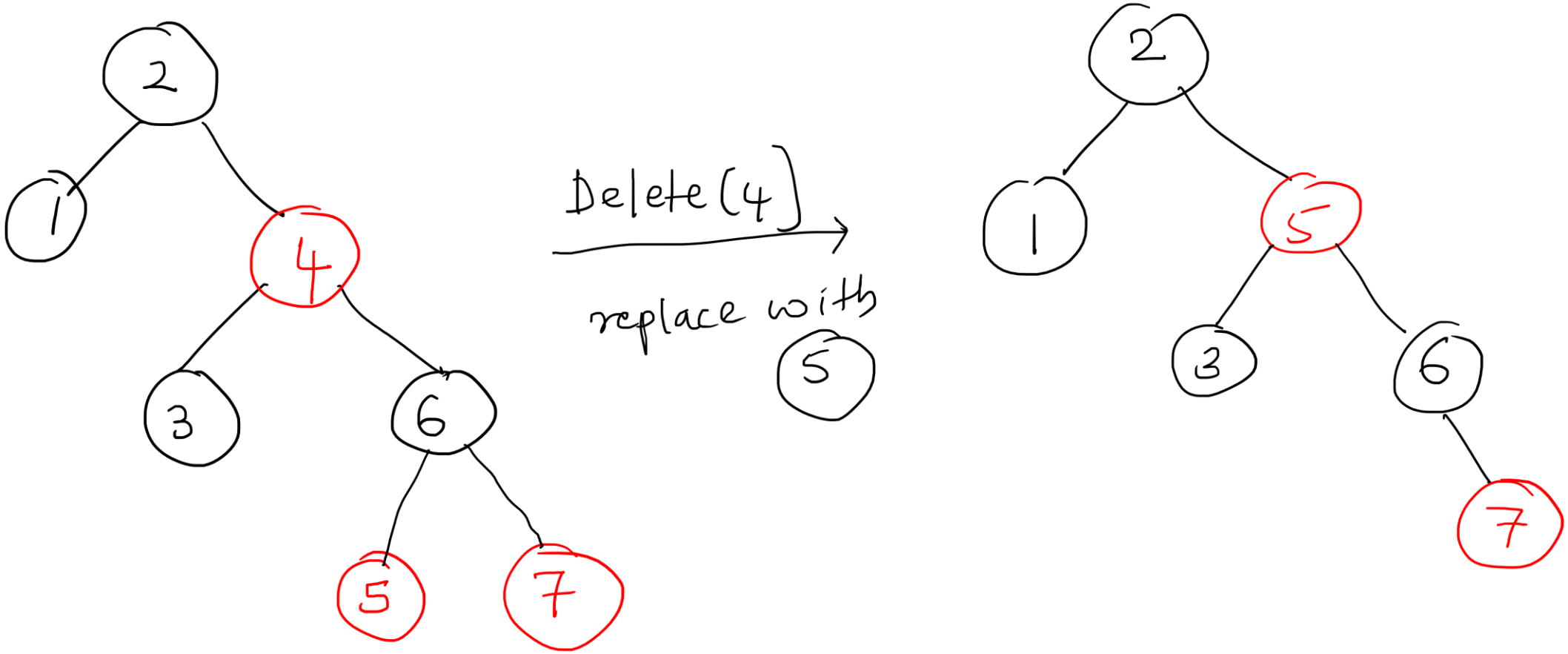
Scenario 3 : neither child is NIL



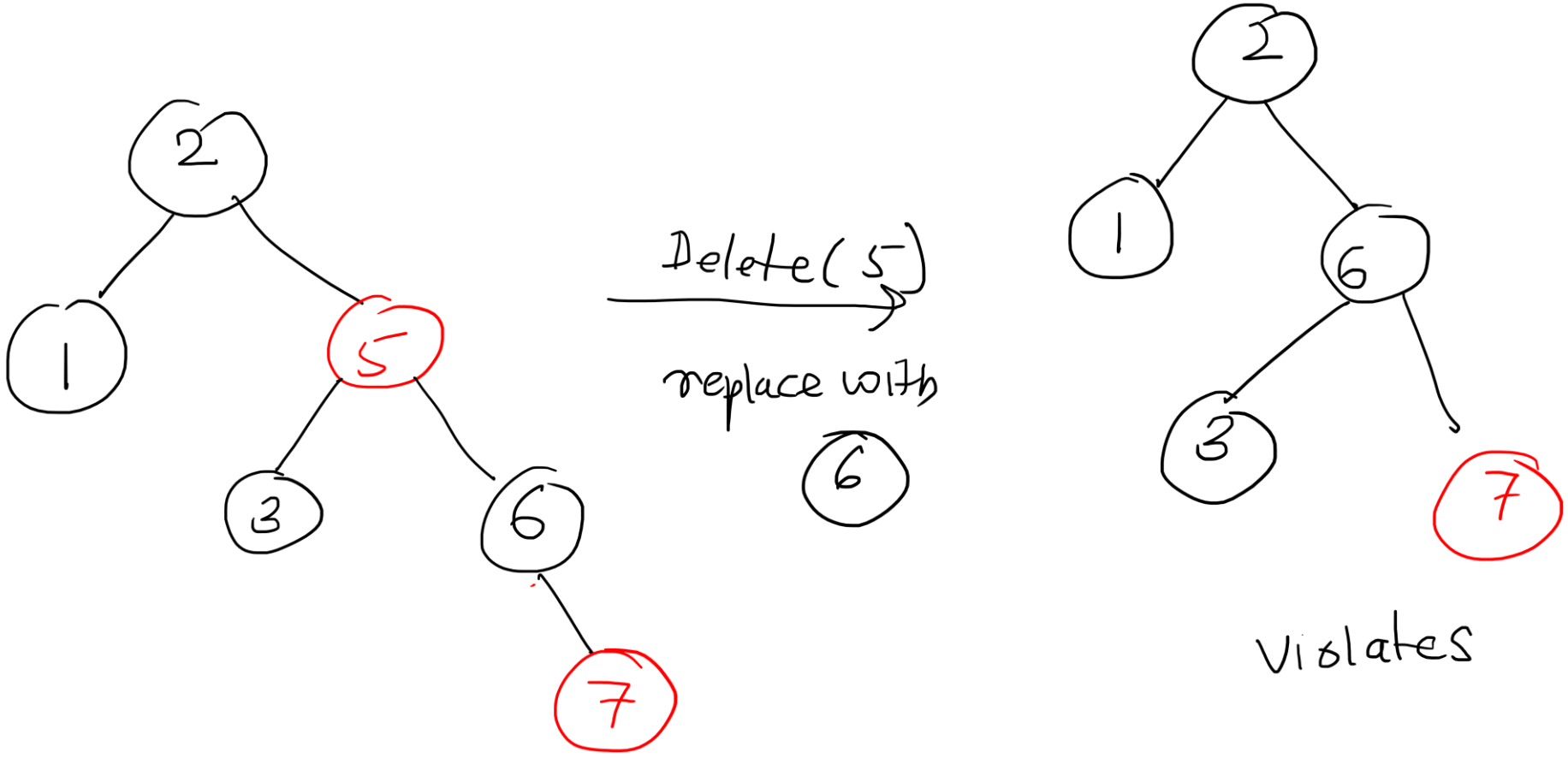
Delete Nodes 4,5,6



Delete Node 4



Delete Node 5



Thank you