Parul® University

Unit 6
Red Black Trees and AVL Trees

Amir Hussain

Cyber Security Trainer

Computer Science & Engineering

Topic

AVL tree Construction Operations on AVL

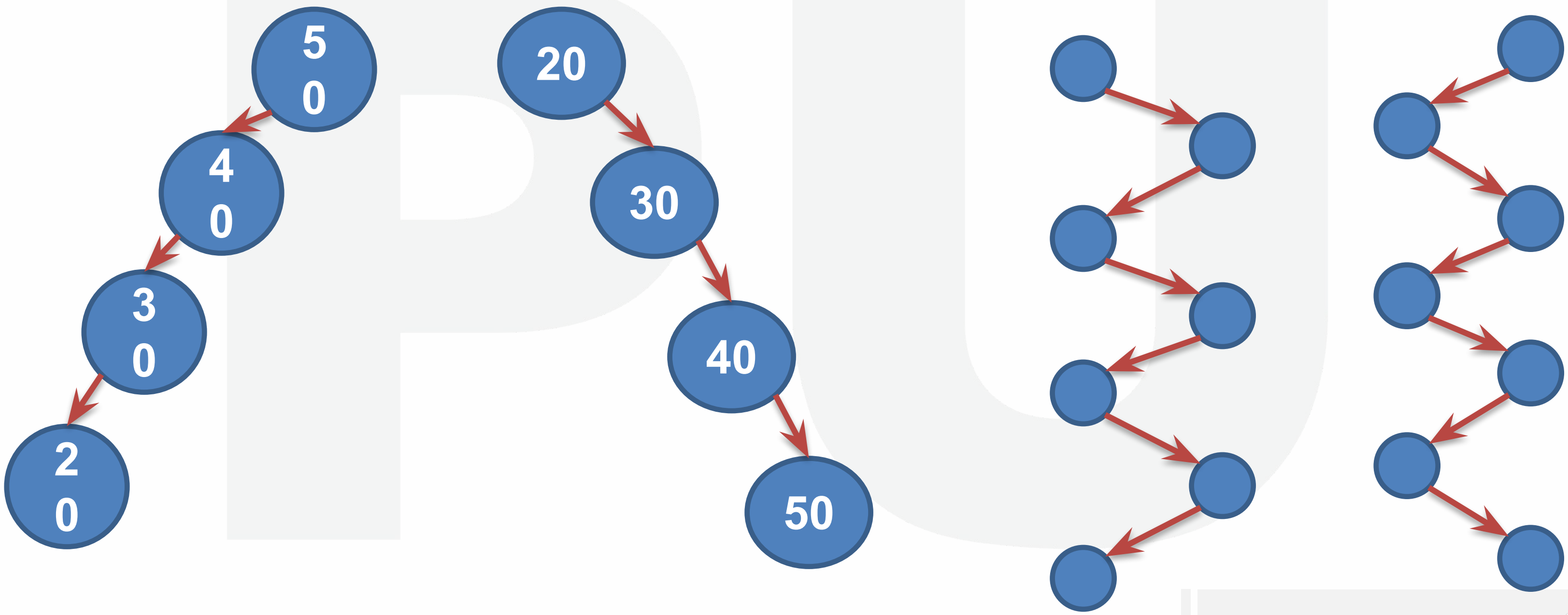
Introduction

- Binary Search Tree gives advantage of Fast Search, but sometimes in few cases we are not able to get this advantage. E.g. look into worst case BST

- Balanced binary trees are classified into two categories
  - Height Balanced Tree (AVL Tree)
  - Weight Balanced Tree

Balanced Tree

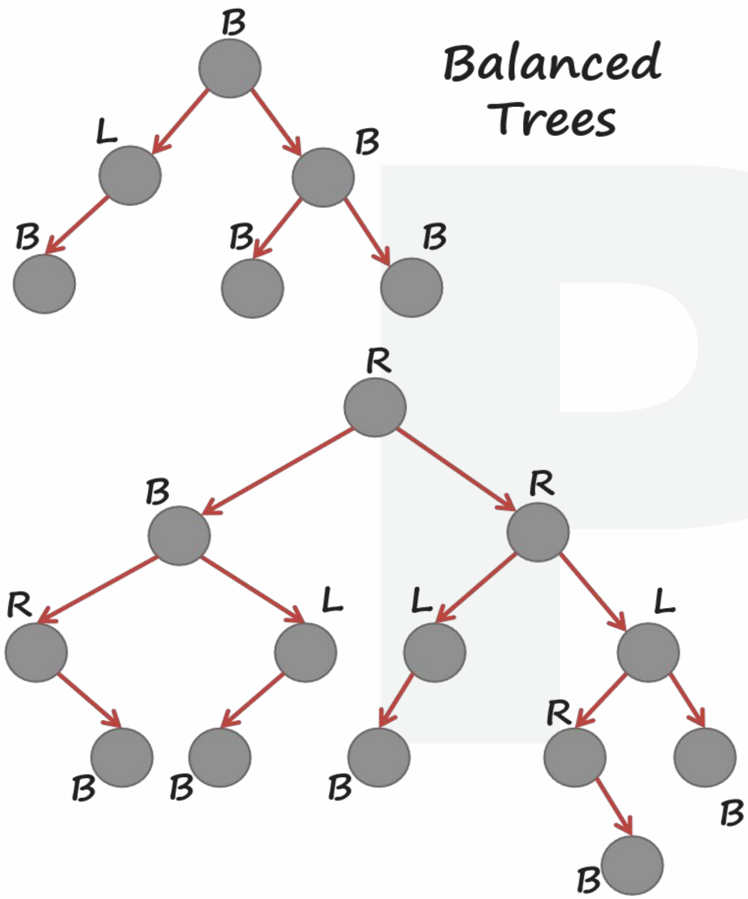
Worst search time cases for Binary Search Tree

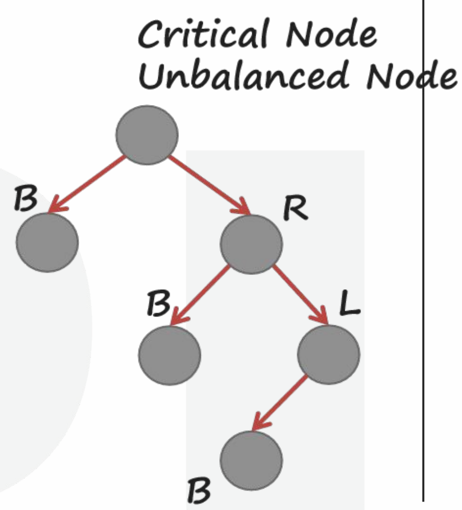


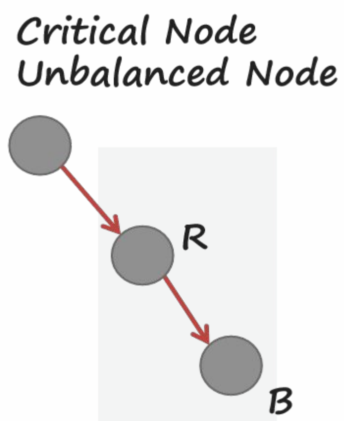
Height Balanced Tree (AVL Tree)

- A tree is called **AVL tree (Height Balanced Tree)**, if each node possessed one of the following properties
  - A **node** is called **left heavy**, if the **longest path in its left sub tree** is **one** longer than the **longest path of its right sub tree**
  - A **node** is called **right heavy**, if the **longest path in its right subtree** is **one** longer than **the longest path of its left sub tree**
  - A **node** is called **balanced**, if the longest path in **both the right and left sub-trees** are equal
- In height balanced tree, each node must be in one of these states
- If there exists a node in a tree where this is not true, then such a tree is called **Unbalanced**

Height Balanced Tree (AVL Tree)



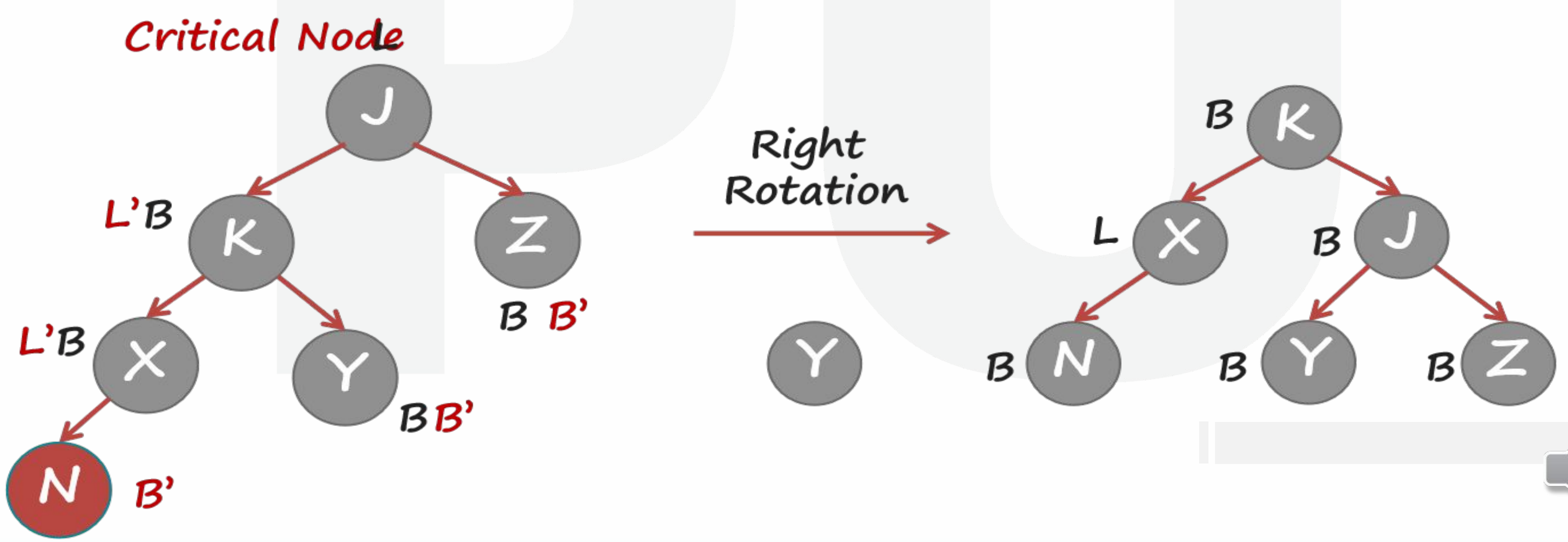




- Sometimes tree becomes unbalanced by inserting or deleting any node
- Then based on position of insertion, we need to rotate the unbalanced node
- **Rotation** is the **process** to **make tree balanced**

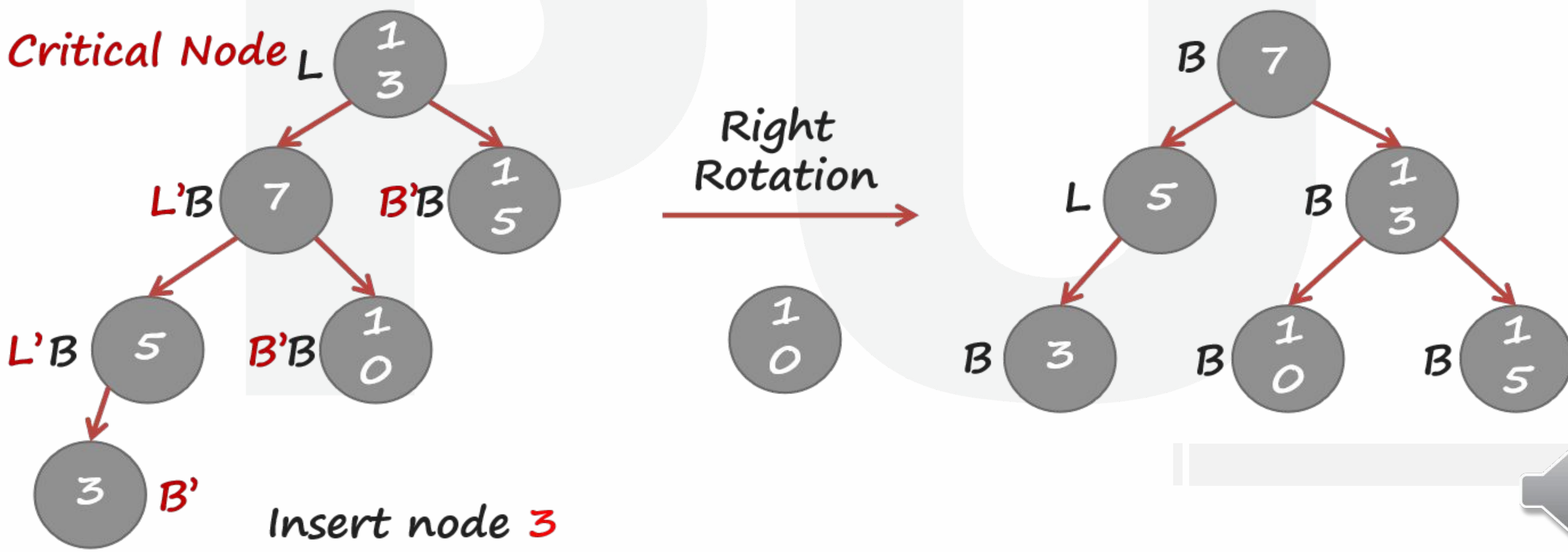
Right Rotation

- Detach left child's right sub-tree
- Consider left child to be the new parent
- Attach old parent onto right of new parent
- Attach old left child's old right sub-tree as left sub-tree of new right child



Right Rotation

a. **Detach** left child's right sub-tree
b. Consider **left child** to be the **new parent**
c. **Attach old parent** onto **right of new parent**
d. **Attach old left child's old right sub-tree** as **left sub-tree of new right child**



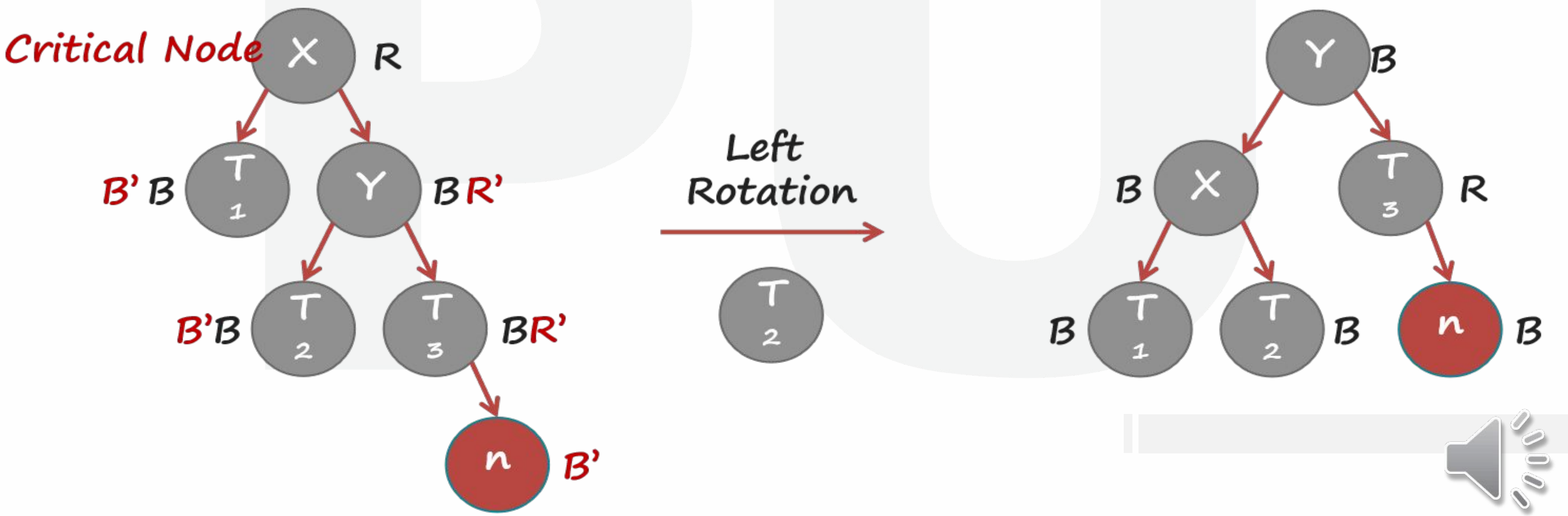
Left Rotation

a. **Detach** right child's leaf sub-tree

b. Consider **right child** to be **new parent**

c. **Attach old parent** onto **left of new parent**

d. **Attach old right child's old left sub-tree** as **right sub-tree of new left child**



Select Rotation based on Insertion Position

Case 1: Insertion into **Left sub-tree** of **nodes Left child**

Single Right Rotation

Case 2: Insertion into **Right sub-tree** of **node's Left child**

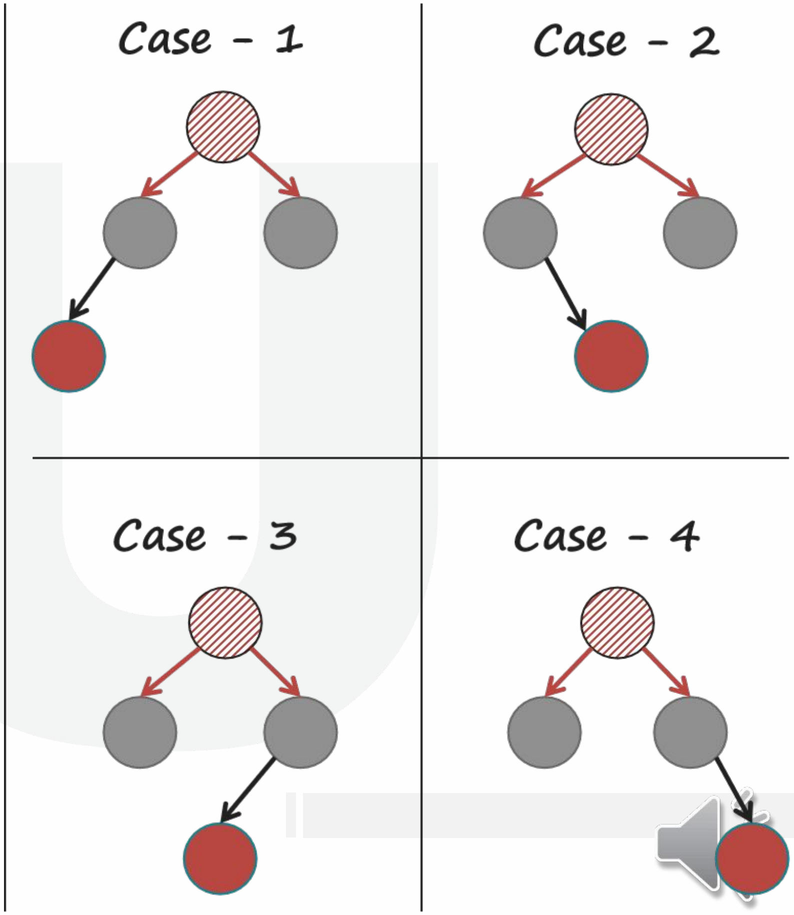
Left Right Rotation

Case 3: Insertion into **Left sub-tree** of **node's Right child**

Right Left Rotation

Case 4: Insertion into **Right sub-tree** of **node's Right child**

Single Left Rotation

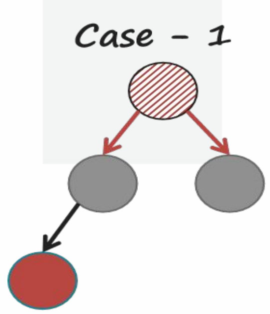


Insertion into Left sub-tree of nodes Left child

Case 1: If node becomes **unbalanced** after **insertion** of new node at **Left sub-tree** of nodes **Left child**, then we need to perform **Single Right Rotation** of **unbalanced node** to balance the node

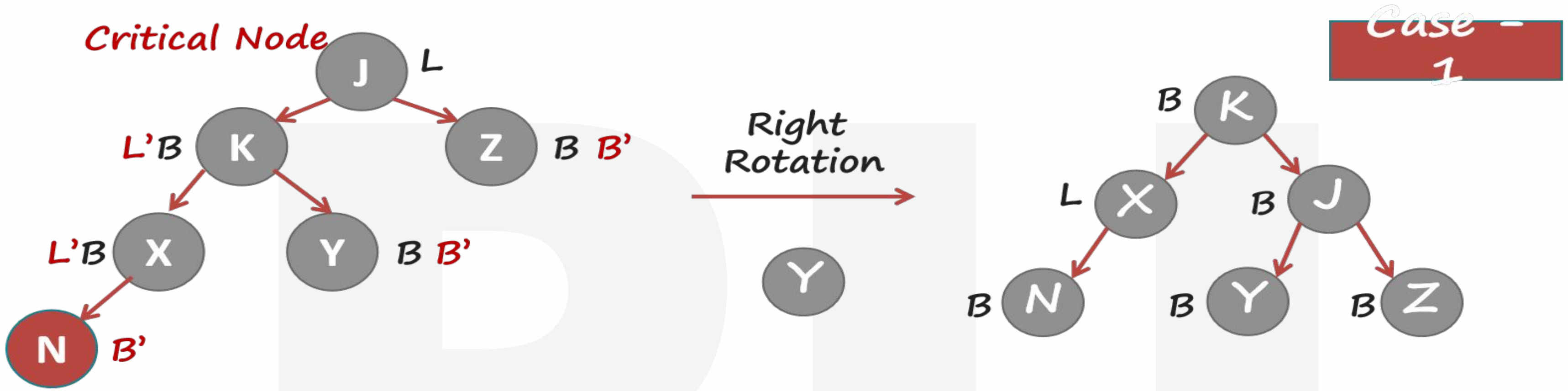
Right Rotation

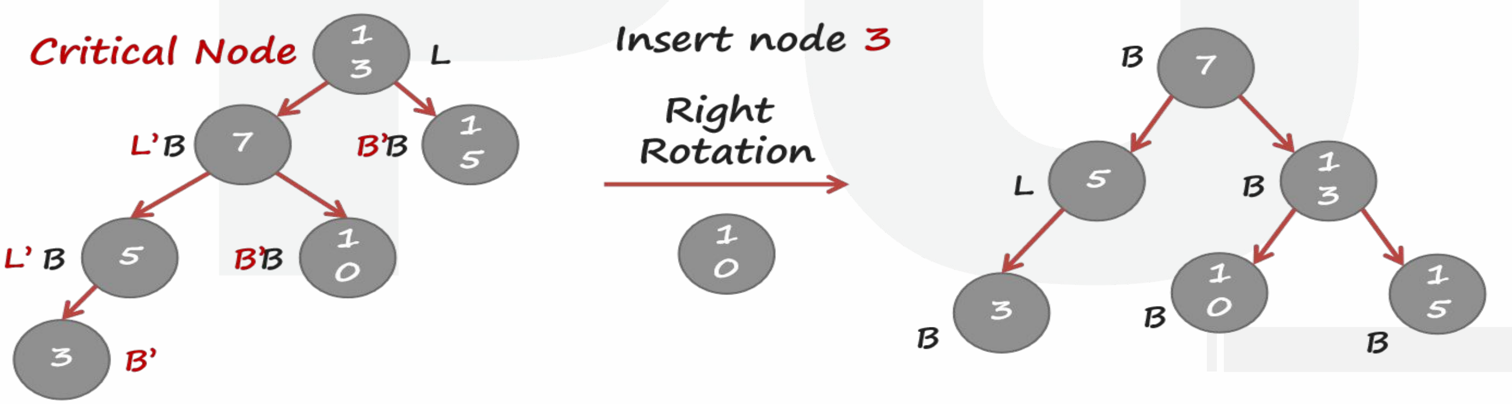
a. Detach leaf child's right sub-tree
b. Consider leaf child to be the new parent
c. Attach old parent onto right of new parent
d. Attach old leaf child's old right sub-tree as leaf sub-tree of new right child



Single Right Rotation
of
unbalanced node

Insertion into Left sub-tree of nodes Left child



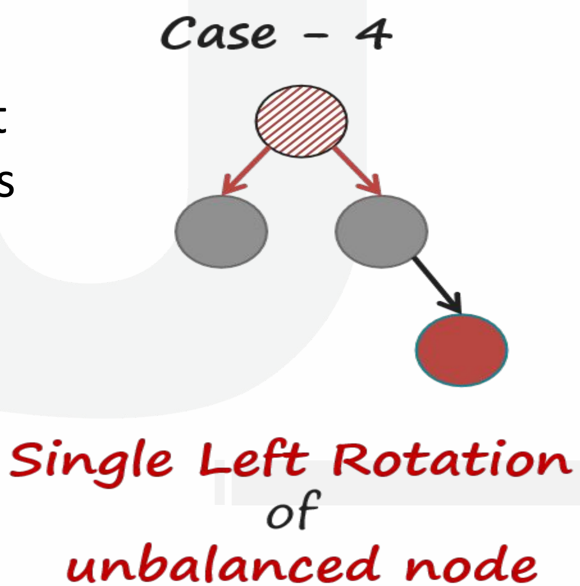


Insertion into Right sub-tree of node’s Right child

Case 4: If node becomes unbalanced after **insertion of new node** at **Right sub-tree** of **nodes Right child**, then we need to perform **Single Left Rotation** of **unbalance node** to balance the node

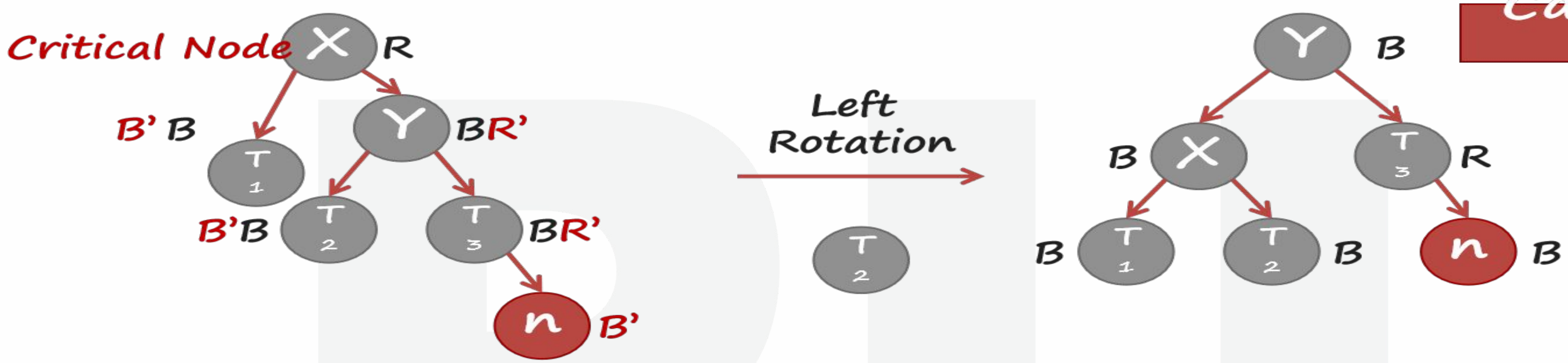
Left Rotation

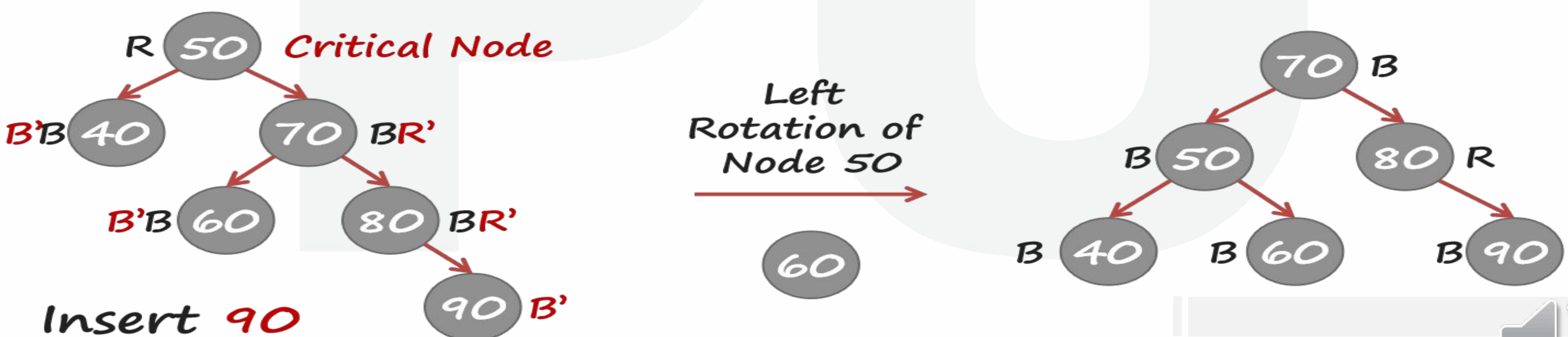
A. Detach right child’s leaf sub-tree
B. Consider right child to be new parent
C. Attach old parent onto left of new parent
D. Attach old right child’s old left sub-tree as right sub-tree of new left child



Insertion into Right sub-tree of node's Right child

Case – 4



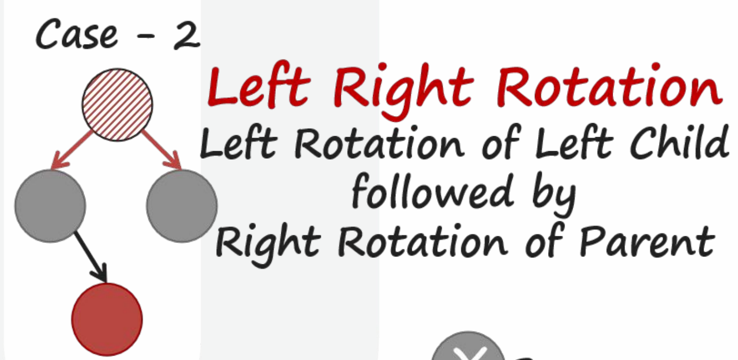


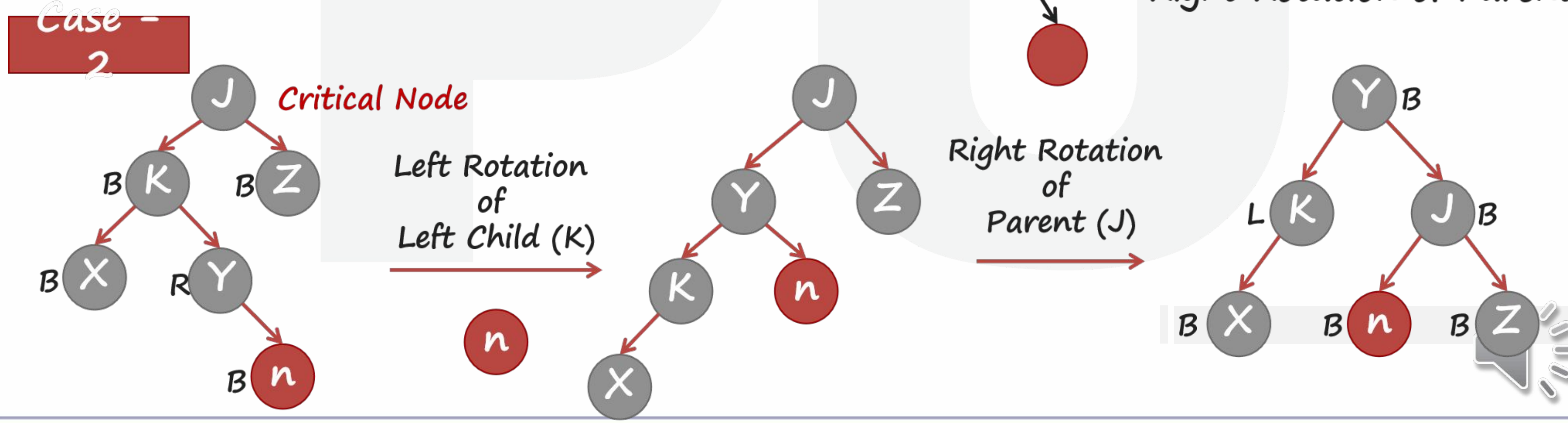
Insertion into Right sub-tree of node's Left child

Case 2: If node becomes unbalanced **after insertion** of **new node** at **Right sub-tree** of **node's Left child**, then we need to perform **Left Right Rotation** for **unbalanced node**.

Left Right Rotation

Left Rotation of **Left Child** followed by
Right Rotation of **Parent**



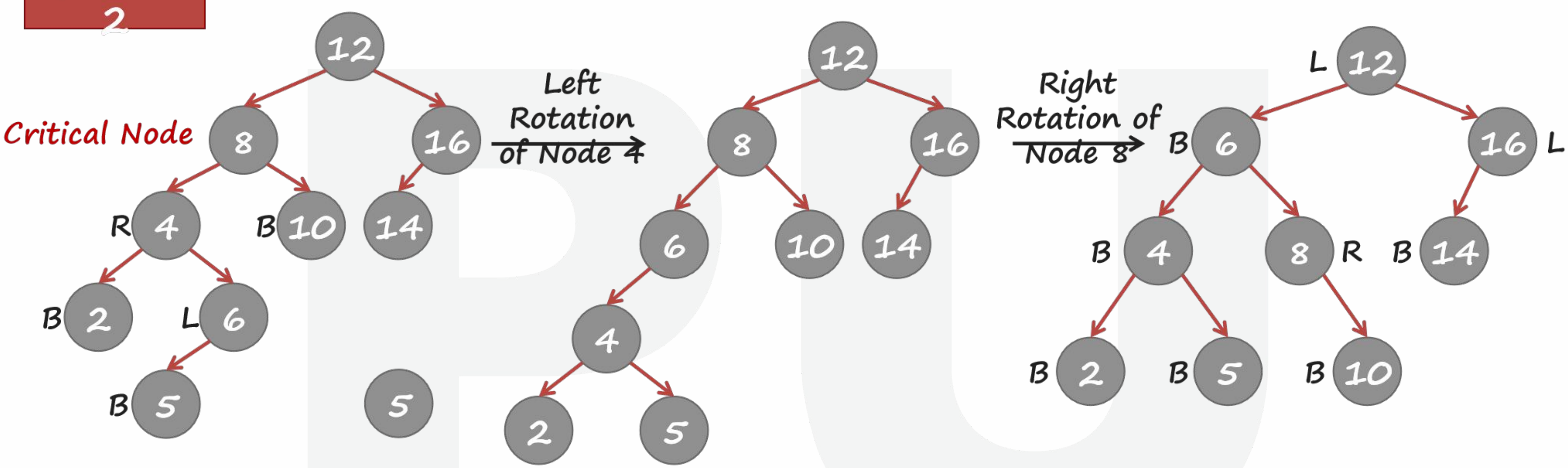


Parul® University

DIGITAL LEARNING CONTENT

Insertion into Right sub-tree of node’s Left child

Case - 2

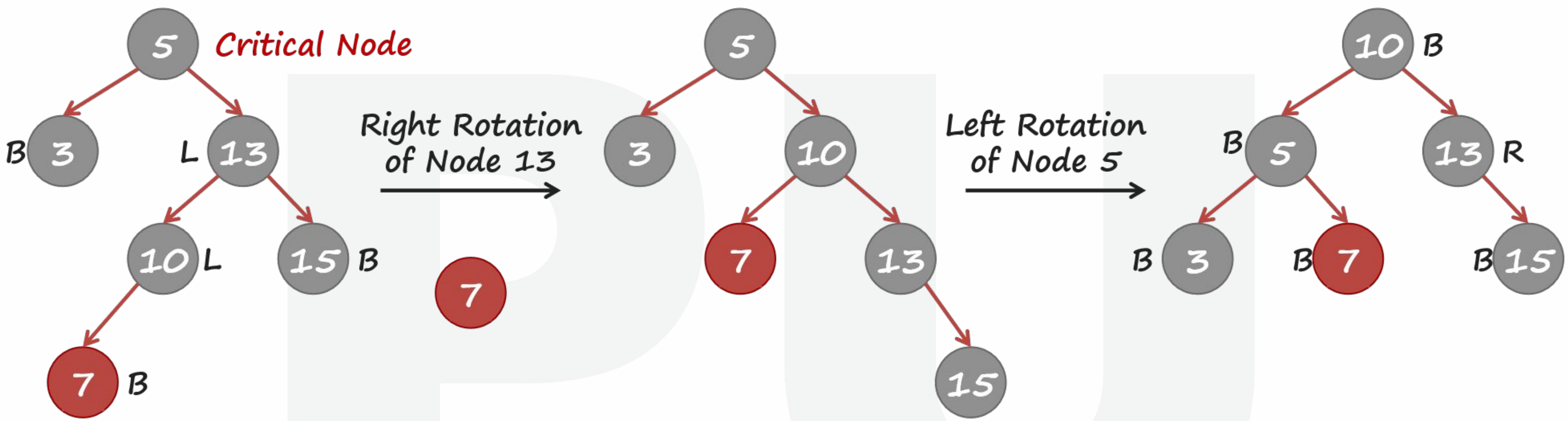


Left Right Rotation

Left Rotation of Left Child (4)
followed by
Right Rotation of Parent (8)

Insertion into Left sub-tree of node's Right child

Case - 3

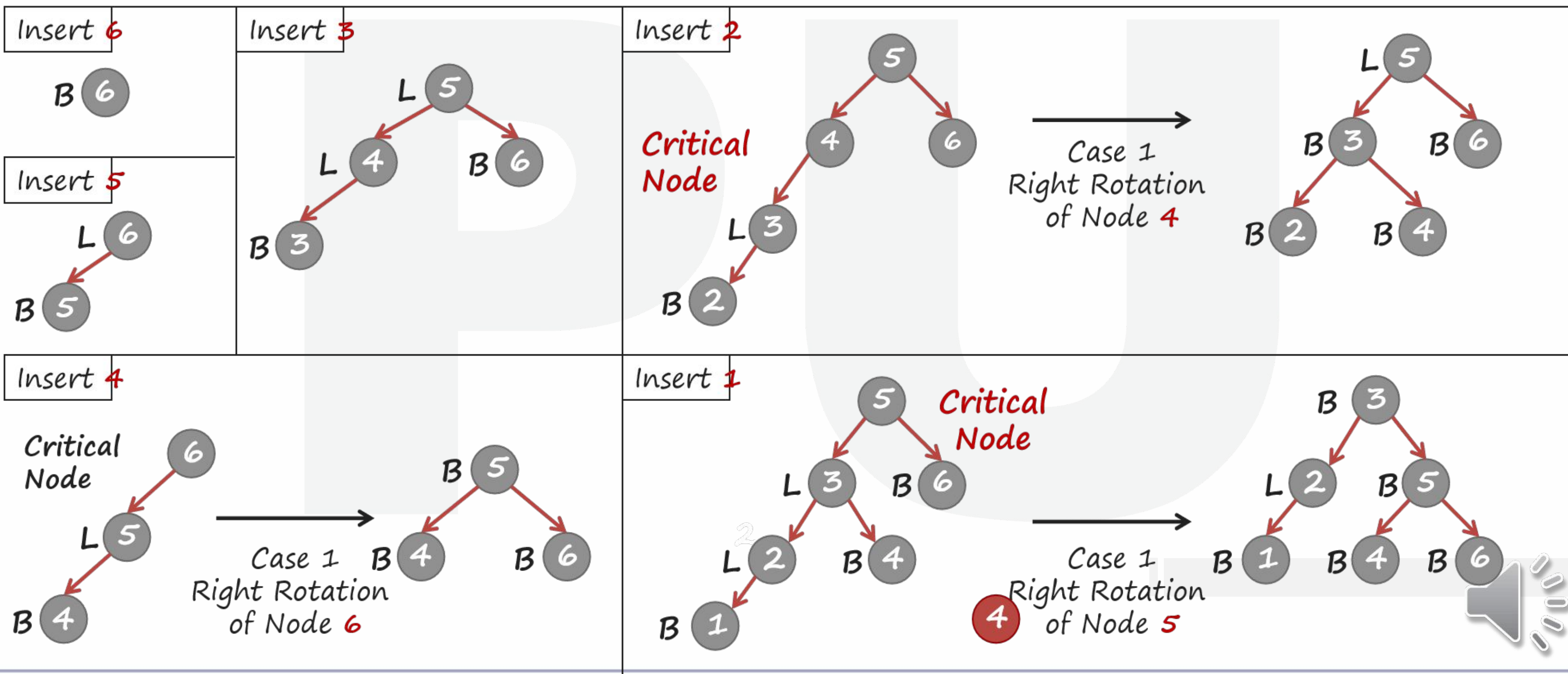


Right Left Rotation

Right Rotation of Right Child (13)
followed by
Left Rotation of Parent (5)

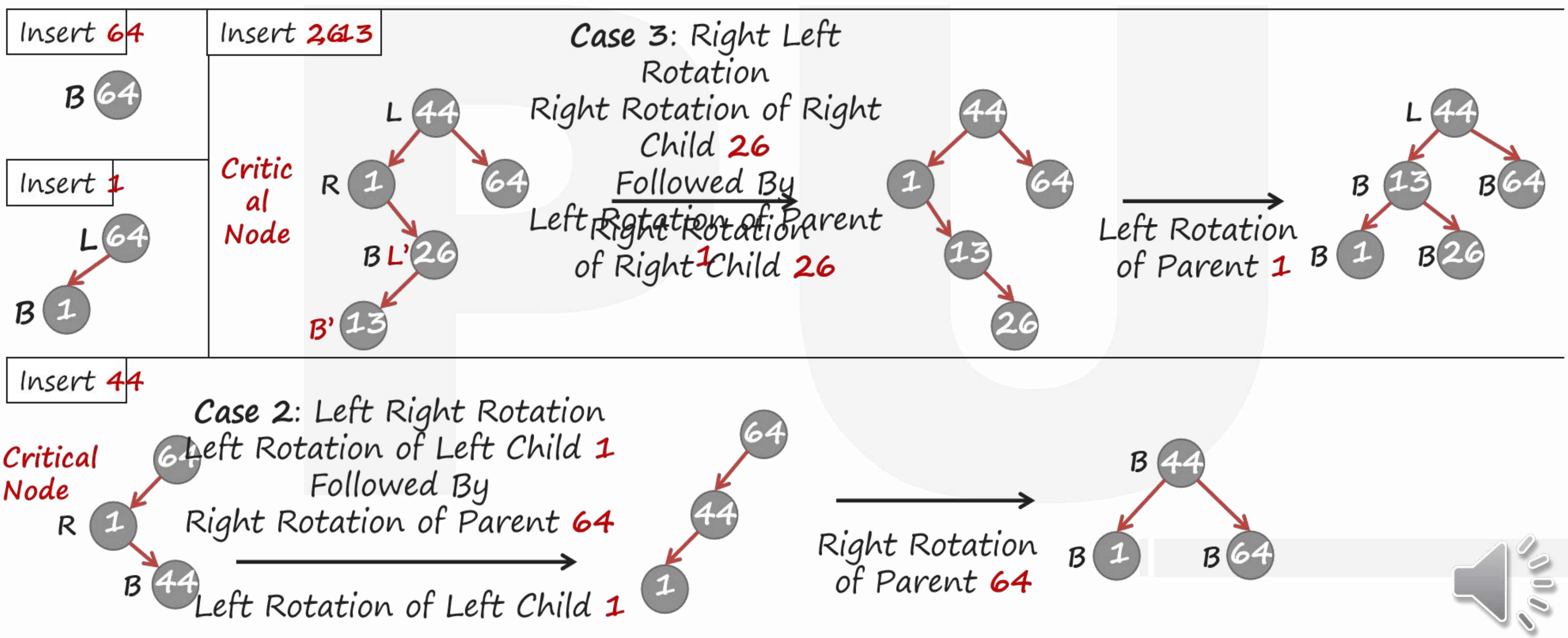
Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence 6, 5, 4, 3, 2, 1



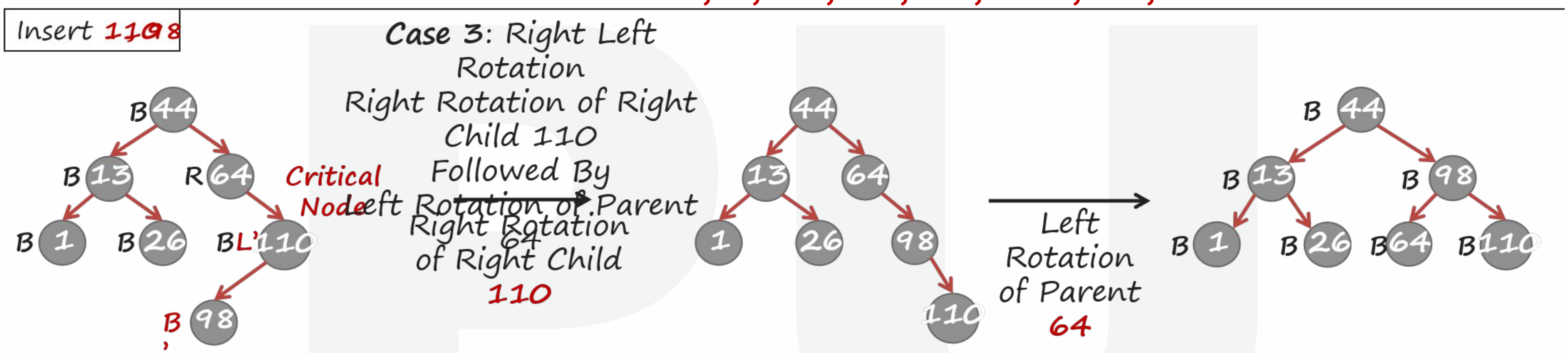
Construct AVL Search Tree

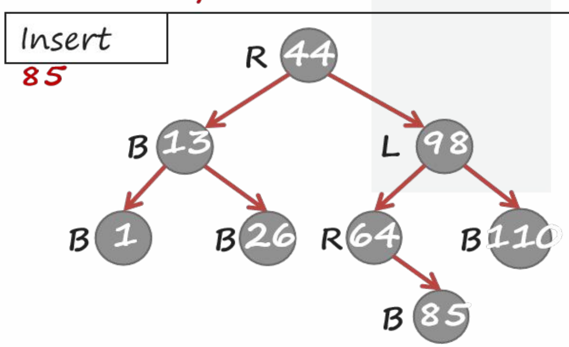
Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**



Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

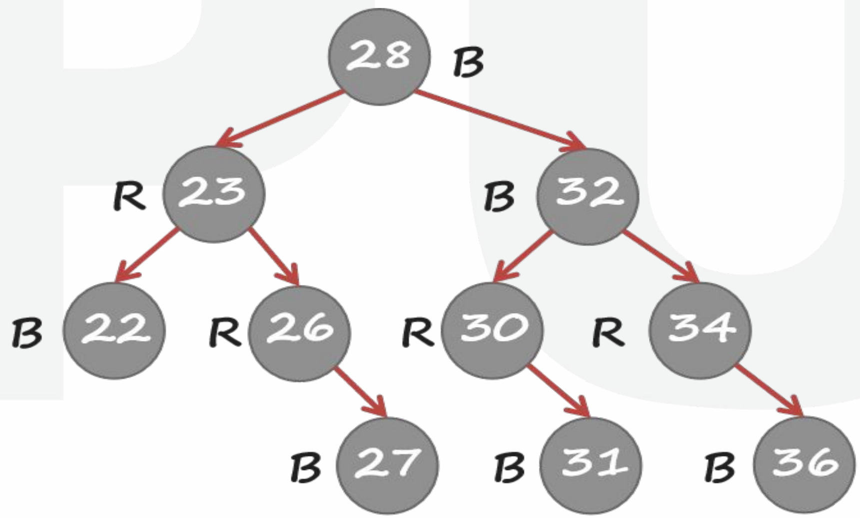




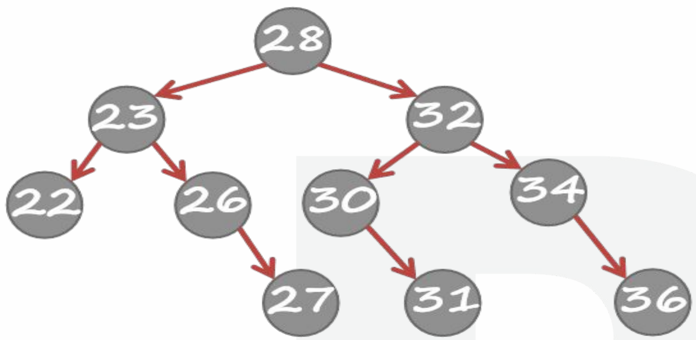
Deleting Node from AVL Tree

If element to be deleted **does not have empty right sub-tree**, then **element** is **replaced** with its **In-Order successor** and its **In-Order successor** is **deleted** instead

During **winding up phase**, we need to **revisit every node** on the **path** from the **point of deletion** up to the **root**, rebalance the tree if require



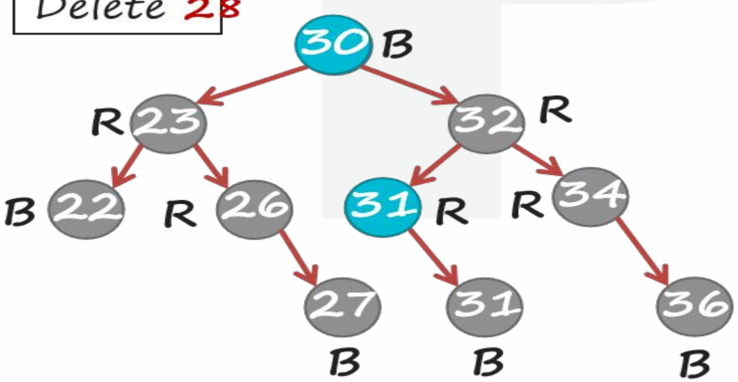
Deleting Node from AVL Tree



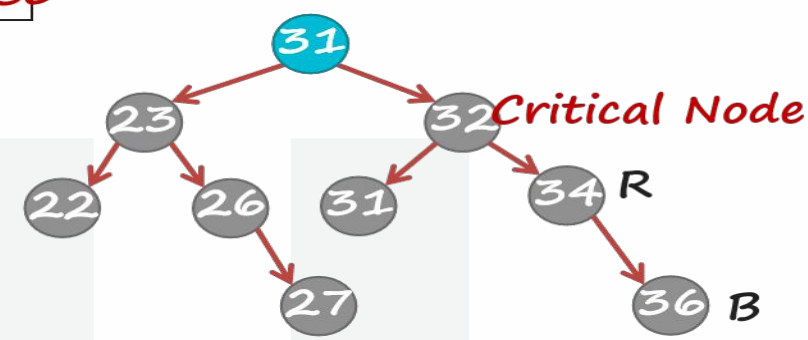
In-Order Traversal

22, 23, 26, 27, 28, 30, 31, 32, 34, 36

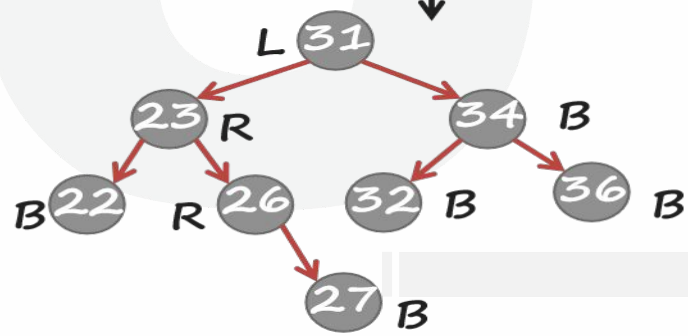
Delete 28



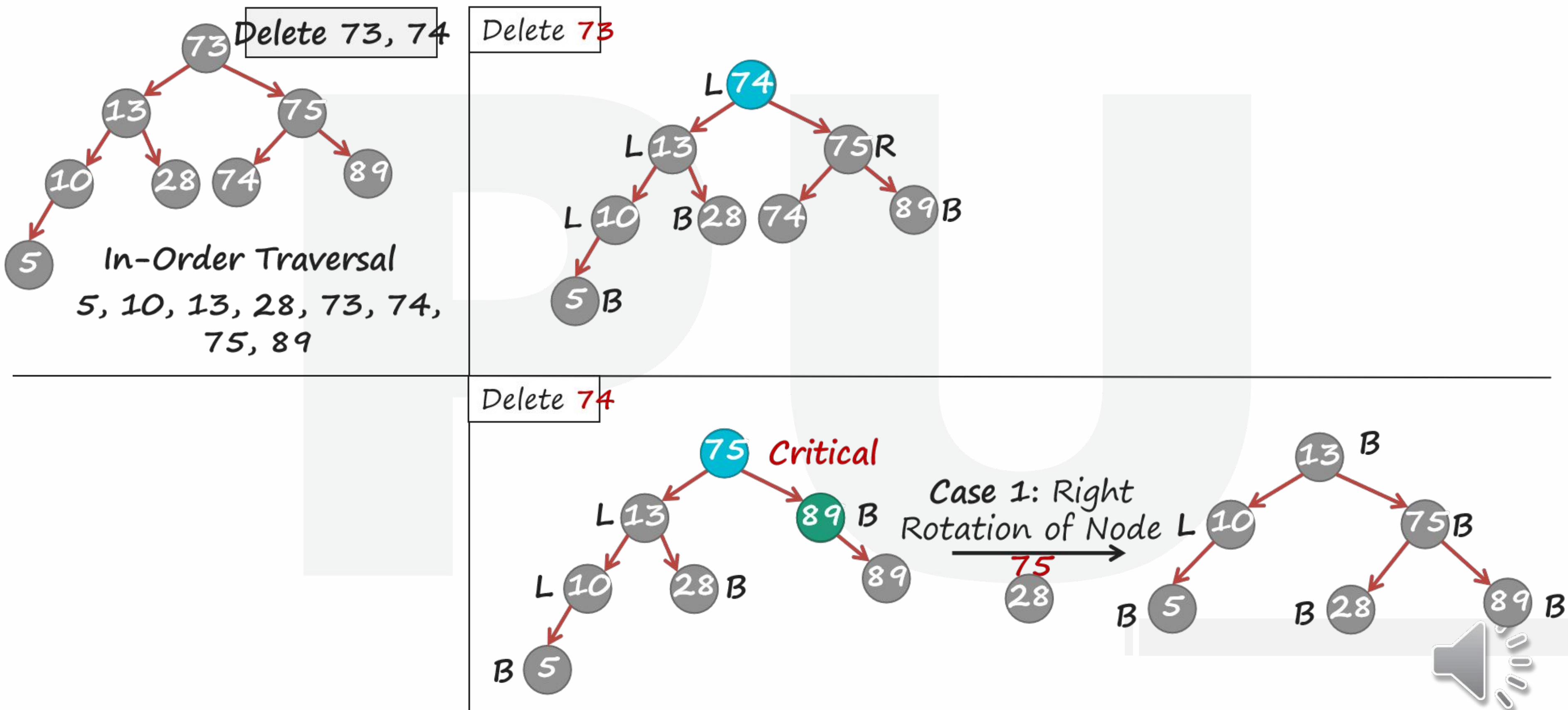
Delete 30



Case 4:
Left Rotation of
Node 32



Deleting Node from AVL Tree



Topic

Red Black Trees

Red Black Trees

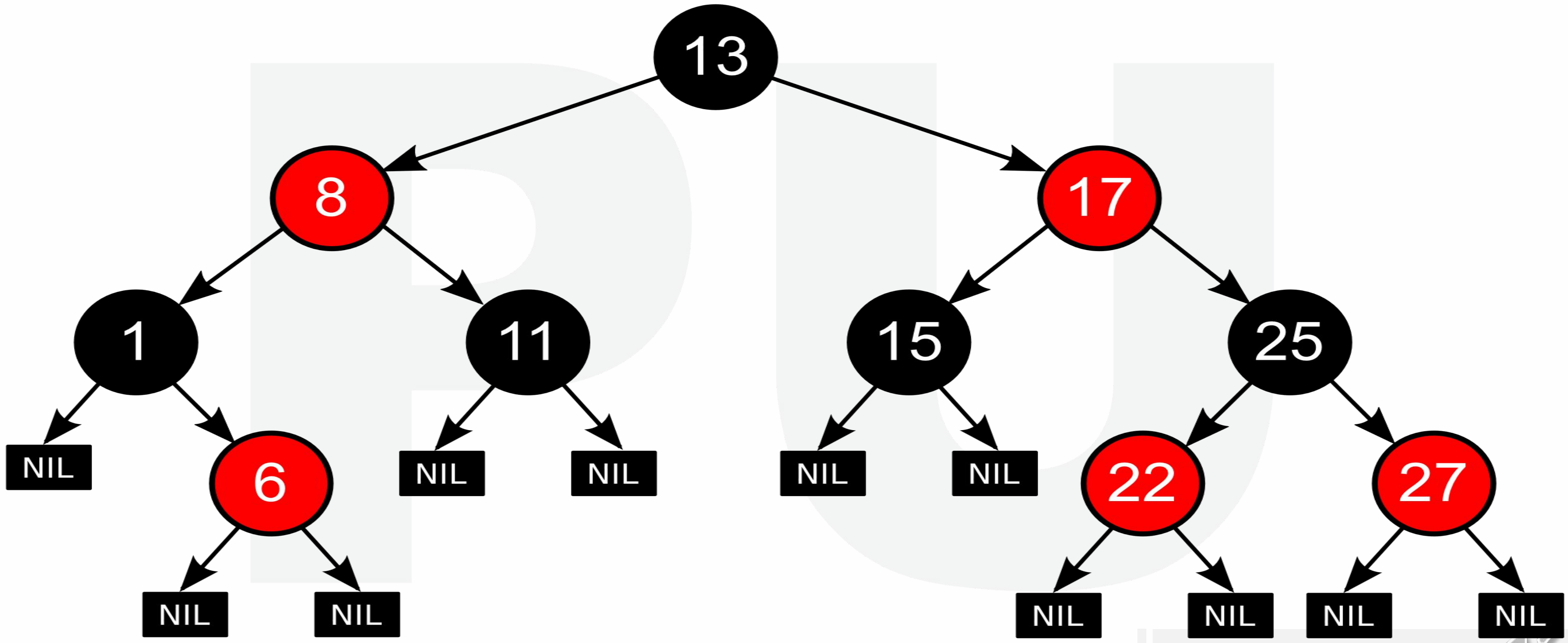
A Red-Black Tree is a self-balancing binary search tree where each node has an additional attribute: a color, which can be either red or black.

The primary objective of these trees is to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation.

Properties of Red Black Trees

1. Node Color: Each node is either red or black.
2. Root Property: The root of the tree is always black.
3. Red Property: Red nodes cannot have red children (no two consecutive red nodes on any path).
4. Black Property: Every path from a node to its descendant null nodes (leaves) has the same number of black nodes.
5. Leaf Property: All leaves (NIL nodes) are black.

Red Black Trees



Insertion Operation on Red Black Trees

Inserting a new node in a Red-Black Tree involves a two-step process: performing a standard binary search tree (BST) insertion, followed by fixing any violations of Red-Black properties.

BST Insert: Insert the new node like in a standard BST.
Fix Violations:
    If the parent of the new node is black, no properties are violated.
    If the parent is red, the tree might violate the Red Property, requiring fixes.

Insertion Operation on Red Black Trees

Fixing Violations During Insertion

After inserting the new node as a red node, we might encounter several cases depending on the colors of the node's parent and uncle (the sibling of the parent)

Case 1: Uncle is Red:

Recolor the parent and uncle to black, and the grandparent to red. Then move up the tree to check for further violations.

Case 2: Uncle is Black:

Sub-case 2.1: Node is a right child: Perform a left rotation on the parent.

Sub-case 2.2: Node is a left child: Perform a right rotation on the grandparent and recolor appropriately.

Searching

Searching for a node in a Red-Black Tree is similar to searching in a standard Binary Search Tree (BST).

Start at the Root: Begin the search at the root node.

Traverse the Tree:

- If the target value is equal to the current node's value, the node is found.
- If the target value is less than the current node's value, move to the left child.
- If the target value is greater than the current node's value, move to the right child.

Repeat: Continue this process until the target value is found or a NIL node is reached .

Deletion

Deleting a node from a Red-Black Tree also involves a two-step process:

BST Deletion: Remove the node using standard BST rules.

Fix Double Black:
If a black node is deleted, a “double black” condition might arise, which requires specific fixes.

Deletion

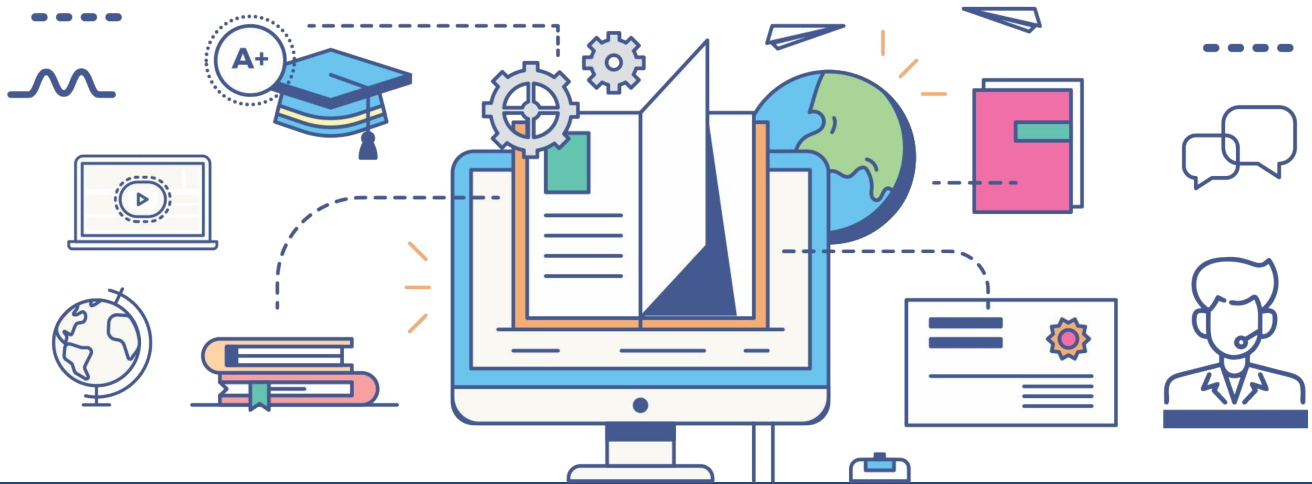
When a black node is deleted, we handle the double black issue based on the sibling's color and the colors of its children:

Case 1: Sibling is Red: Rotate the parent and recolor the sibling and parent.

Case 2: Sibling is Black:

- **Sub-case 2.1: Sibling's children are black:** Recolor the sibling and propagate the double black upwards.
- **Sub-case 2.2: At least one of the sibling's children is red:**
  - **If the sibling's far child is red:** Perform a rotation on the parent and sibling, and recolor appropriately.
  - **If the sibling's near child is red:** Rotate the sibling and its child, then handle as above.

DIGITAL LEARNING CONTENT



Parul® University

www.paruluniversity.ac.in