TechnoCrats GSV

DSA Series Session 08

Binary Search Trees & avl

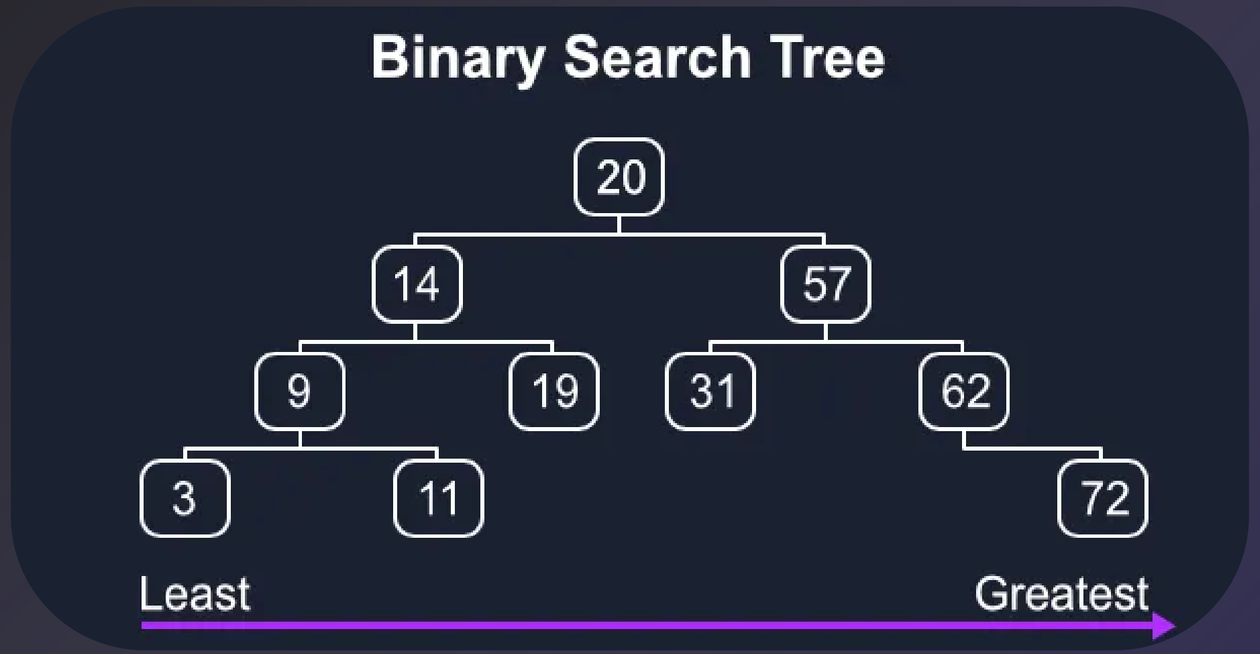
Session by
Ajitesh Channa

WHAT IS BST?

A Binary Search Tree is a binary tree in which each node follows a special order:

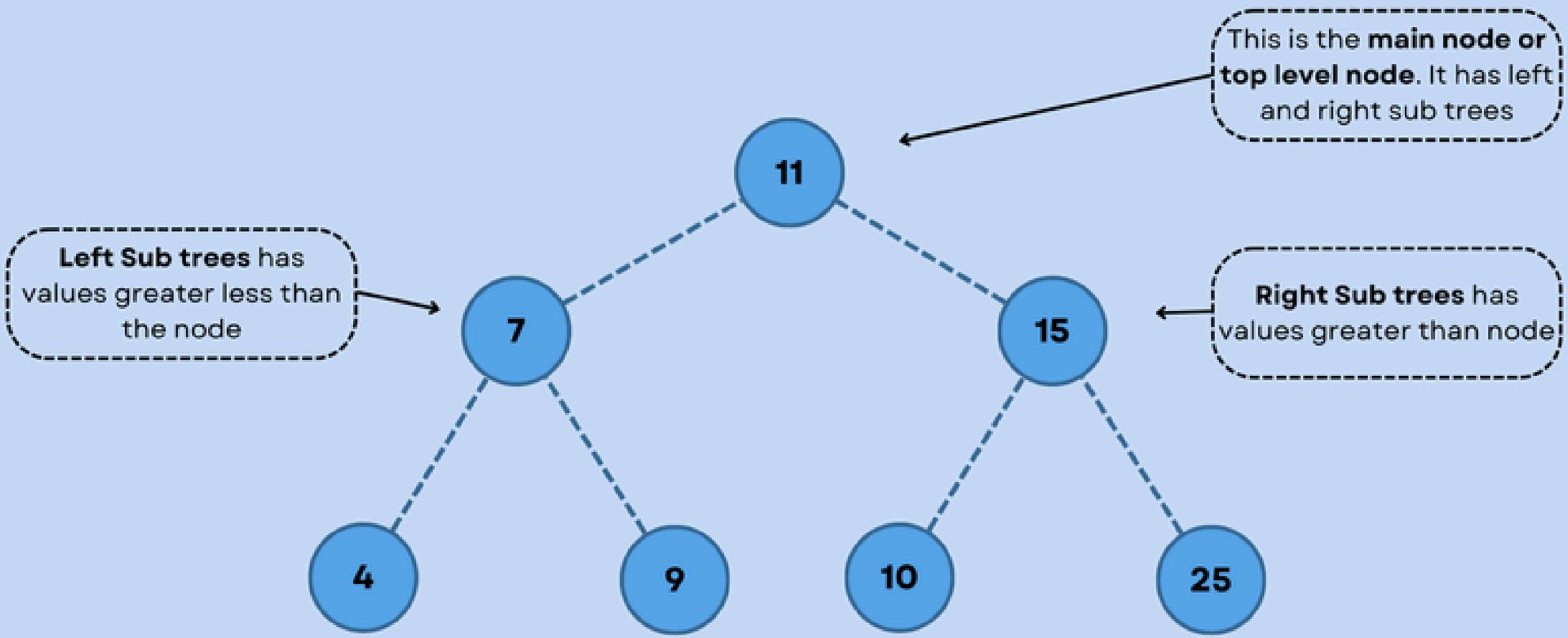
- Left subtree contains only nodes with values less than the node
- Right subtree contains only nodes with values greater than the node
- Both left and right subtrees are also BSTs

Sorted Structure – Left < Root < Right (always!)



TechnoCrats GSV

BINARY SEARCH TREE



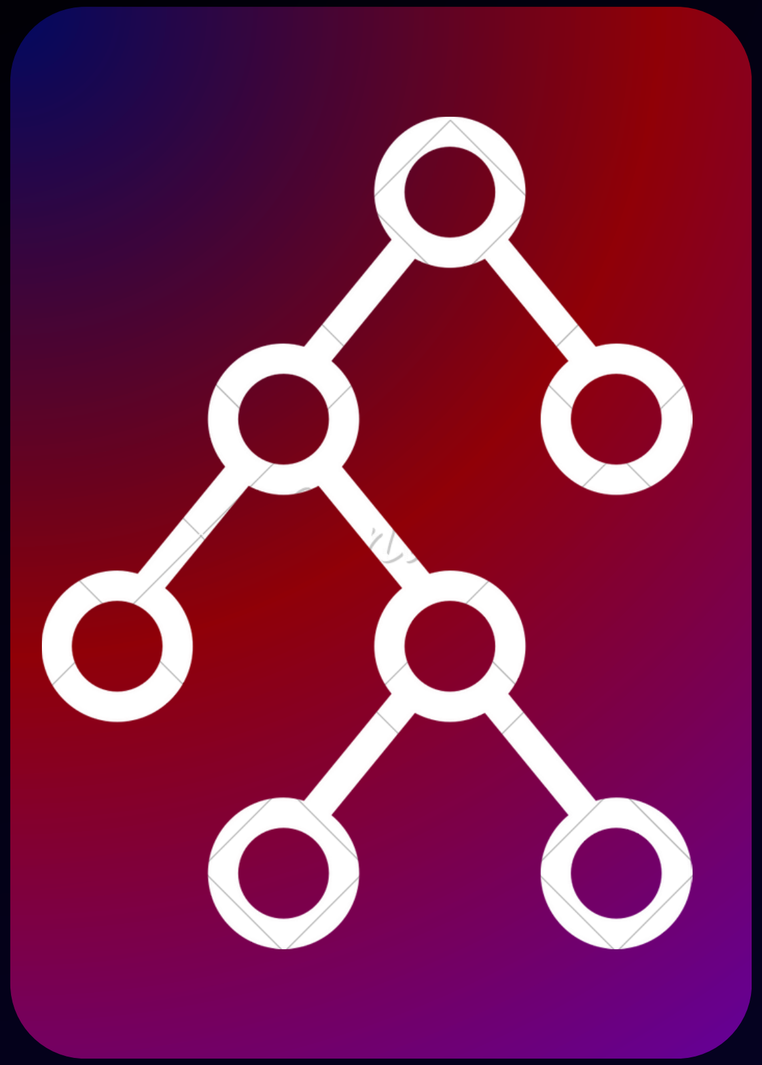
TechnoCrats GSV

EXAMPLE

Imagine you're shopping online for a phone between ₹5000 and ₹10,000 — you don't scroll through hundreds of options one by one, right?

Behind the scenes, products are often organized using Binary Search Tree logic (or a similar search-efficient structure), especially when it comes to price filters.

WHAT IS THE BENEFIT?



Search:

$O(\log n)$

TechnoCrats GSV

POINT TO BE NOTED!!!

INORDER TRAVERSAL OF A BINARY SEARCH TREE GIVES A SORTED LIST

BUILDING A BST
(INSERTION)

ALGORITHM

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

BUILDING A BST

LET'S CREATE ONE!

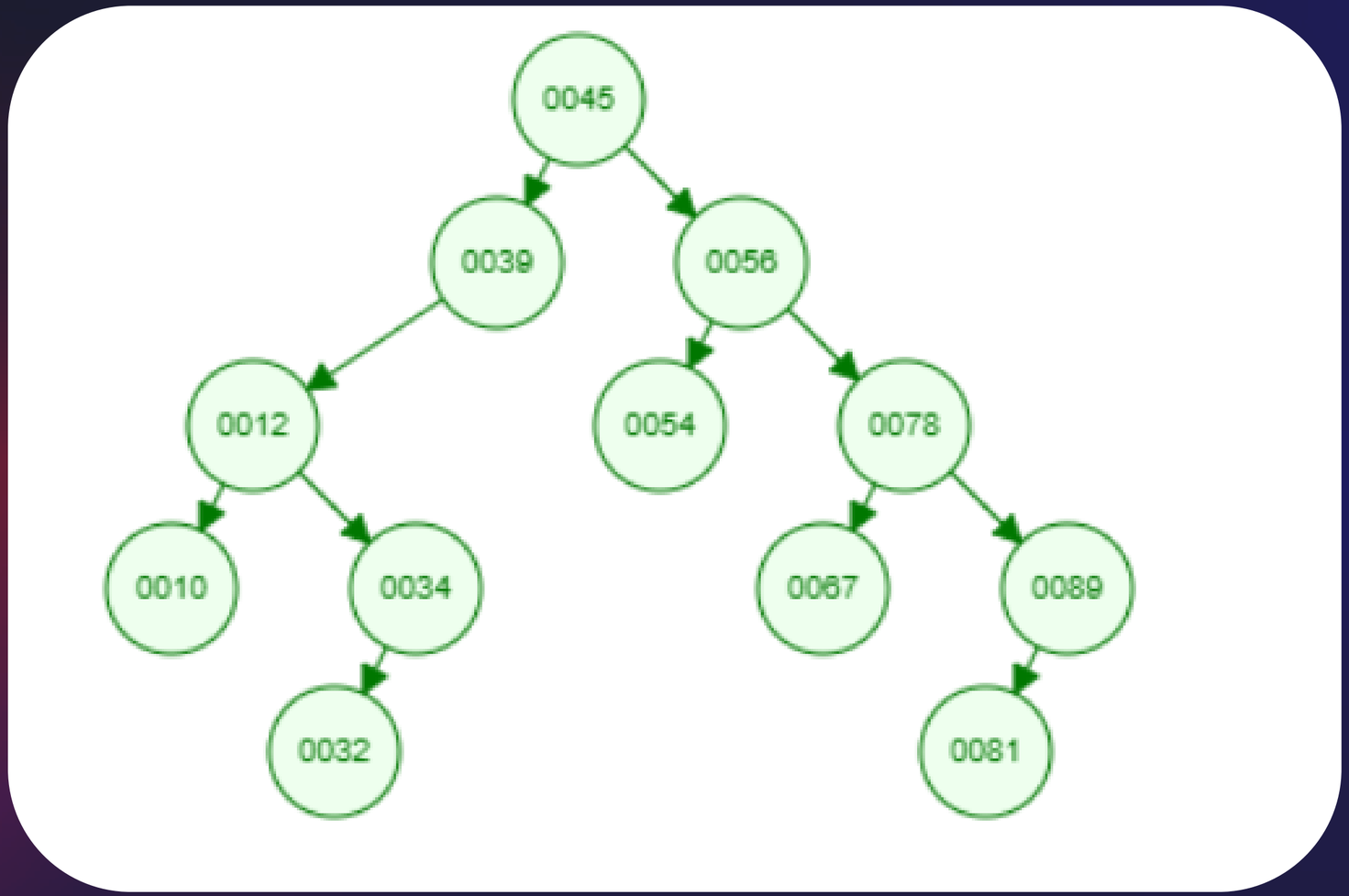
CREATE A BST WITH FOLLWING ELEMENTS

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

TechnoCrats GSV

CREATE A BST WITH FOLLWING ELEMENTS

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



SEARCHING IN A BINARY SEARCH TREE

The average running time of a search operation is O(log2n), as at every step, we eliminate half of the sub-tree from the search process.

Due to its efficiency in searching elements, binary search trees are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

ALGORITHM

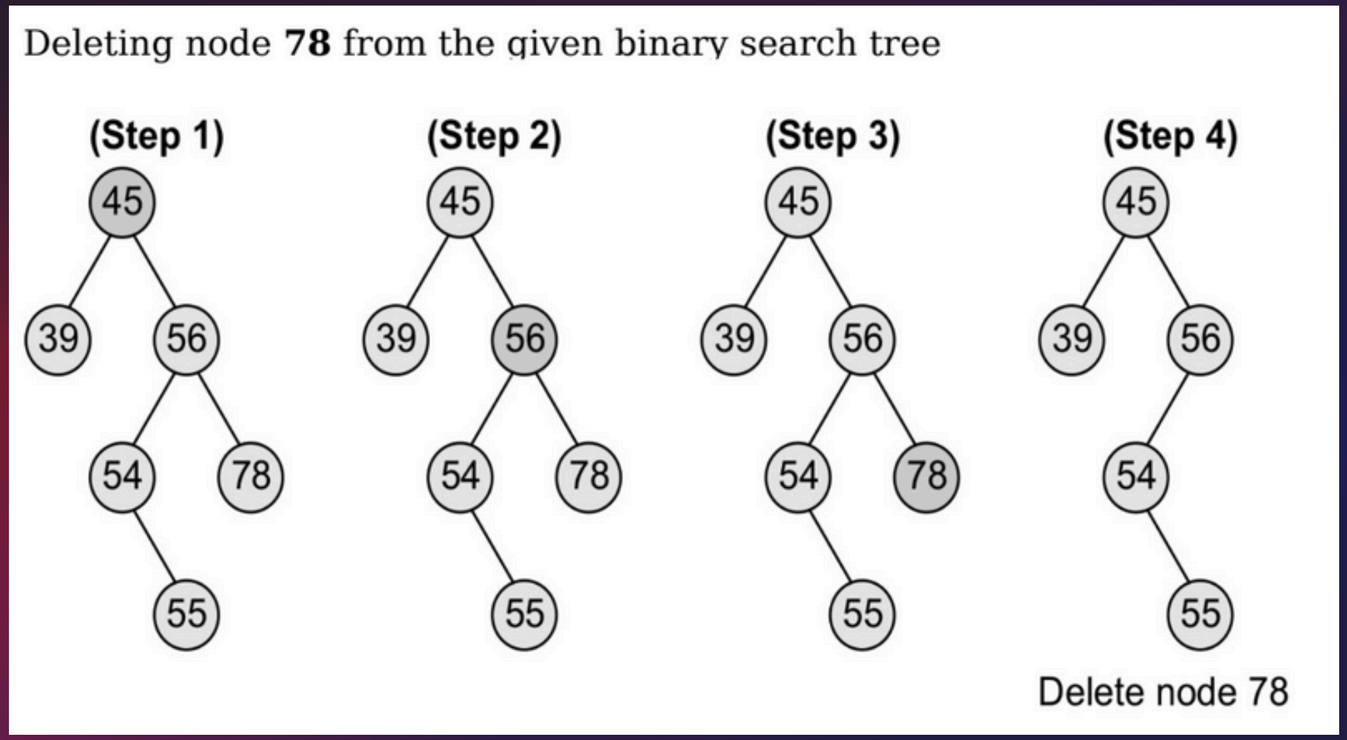
```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
           Return searchElement(TREE -> LEFT, VAL)
         ELSE
           Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

TechnoCrats GSV

LET’S SEARCH

DELETION
IN A BST

Case 1: Node is a Leaf Node (No Children)




```
IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
    SET TREE = NULL
```

“It’s like the last tailender getting out in the match.
No one’s depending on him, no replacement needed, just walk back to the pavilion!”

DELETION

IN A BST

Case 2: Node has One Child

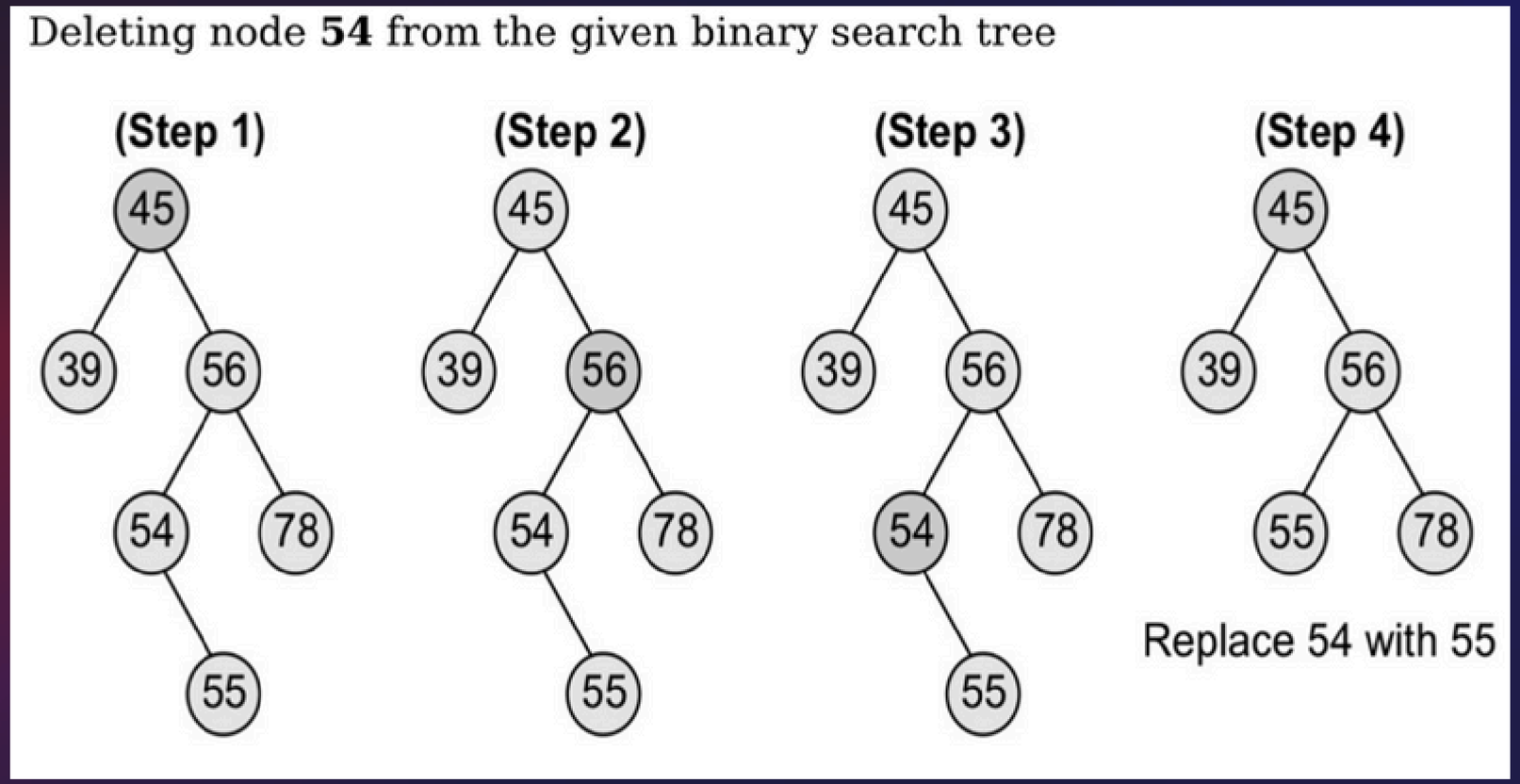
```
if root.left is NULL:
        root = root->right  # Only right child exists
elif root.right is None:
        root = root->left   # Only left child exists
```

In cricket, when a senior like MS Dhoni retires, the team doesn't panic — there's already a young player like Rishabh Pant being trained to replace him.
Same in BST — when a node with one child is deleted, the child steps in without any disruption!

DELETION

IN A BST

Case 2: Node has One Child



DELETION

IN A BST

Case 3: Node has Two Children

To handle this case, replace the node's value with its:

- in-order predecessor (largest value in the left sub-tree)

or

- in-order successor (smallest value in the right sub-tree).

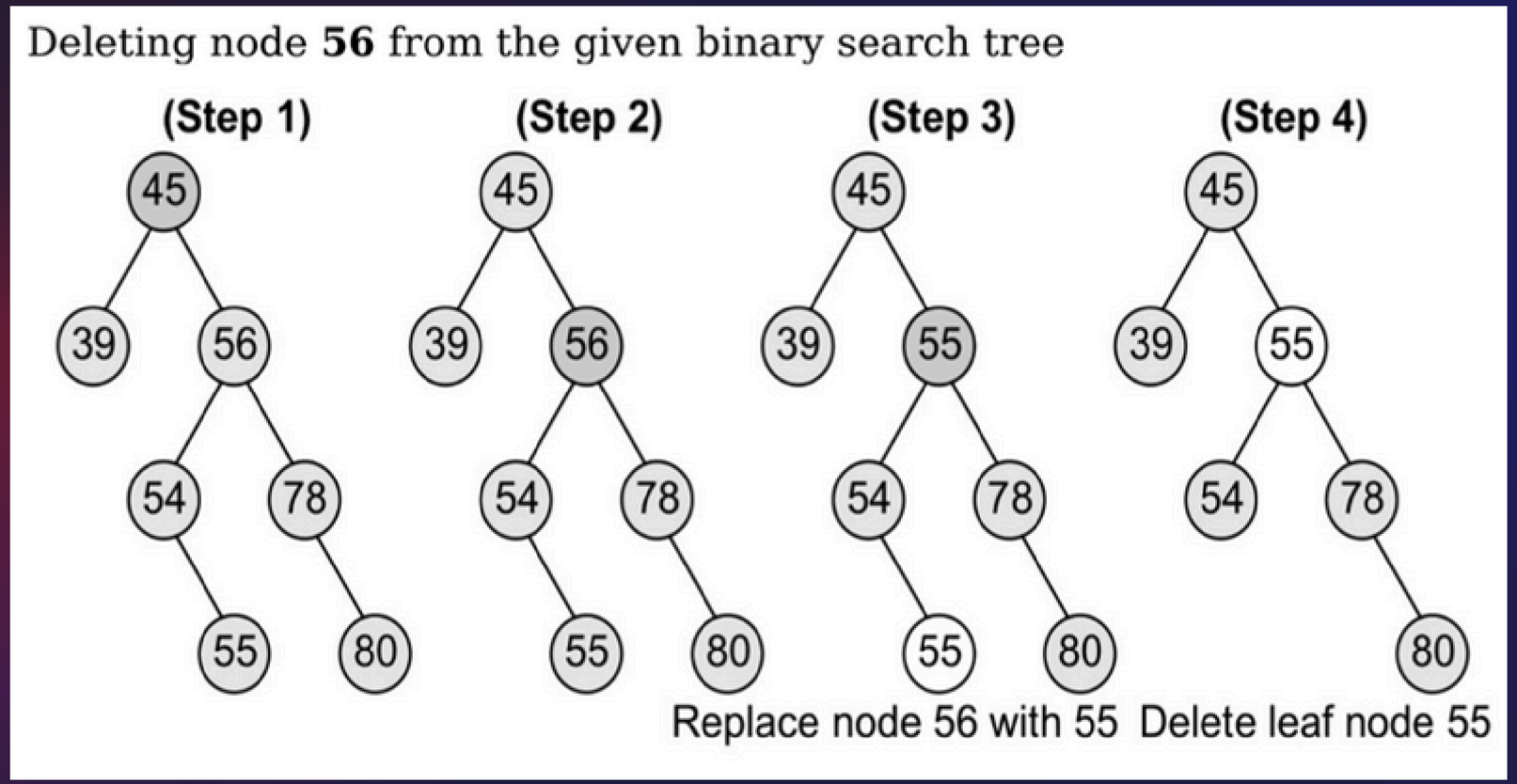
The in-order predecessor or the successor can then be deleted using any of the above cases.

Either Left-Max or Right-Min

DELETION

IN A BST

Case 3: Node has Two Children



DELETION

IN A BST

ALGORITHM

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
         IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
                SET TREE = TREE -> LEFT
          ELSE
                SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

TechnoCrats GSV

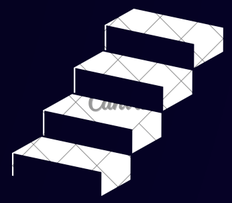
POINT TO BE NOTED!!!

TIME COMPLEXITY FOR DELETION:

O(Log(n))
Average Case

O(n)
Worst Case

HEIGHT OF BST

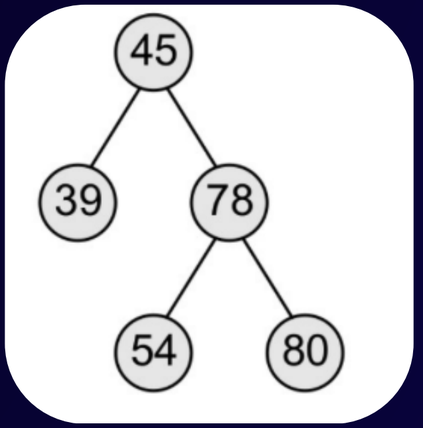


- Height of empty tree = -1 or 0 (based on definition)
- Height of a single node = 0

```
Height (TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
         SET LeftHeight = Height(TREE -> LEFT)
         SET RightHeight = Height(TREE -> RIGHT)
         IF LeftHeight > RightHeight
              Return LeftHeight + 1
         ELSE
              Return RightHeight + 1
         [END OF IF]
        [END OF IF]
Step 2: END
```

The height of a Binary Search Tree is the number of edges on the longest path from the root node to a leaf node.

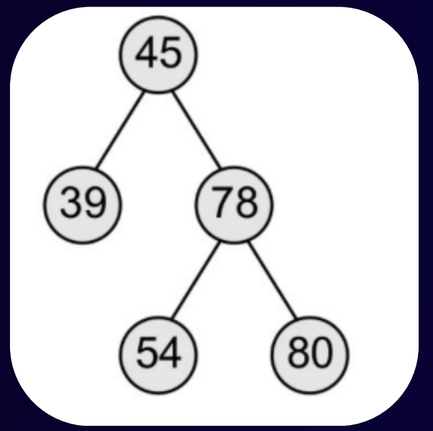
Height of Tree = Maximum(Height of Left Subtree, Height of Right Subtree) +1
= Max(1,2) + 1
= 2 + 1
= 3



NUMBER OF NODES

Number of nodes in a BST means total elements present in the tree — each data point counts as 1 node.

```
totalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
            Return totalNodes(TREE -> LEFT)
                + totalNodes(TREE -> RIGHT) + 1
        [END OF IF]
Step 2: END
```



Number of nodes = totalNodes(left sub–tree) + totalNodes(right sub–tree) + 1
= 1 + 3 + 1
= 4

TechnoCrats GSV

tREEs

avL tREEs

sELF-bALANCiNG

WHAT MAKES AN AVL TREE SPECIAL

At first glance, AVL Trees look just like your everyday Binary Search Trees (BSTs)...
But wait — there's a twist! 🌀

Each node in an AVL Tree carries an extra value called the **Balance Factor** – it's like the tree's internal compass for balance. 🧭

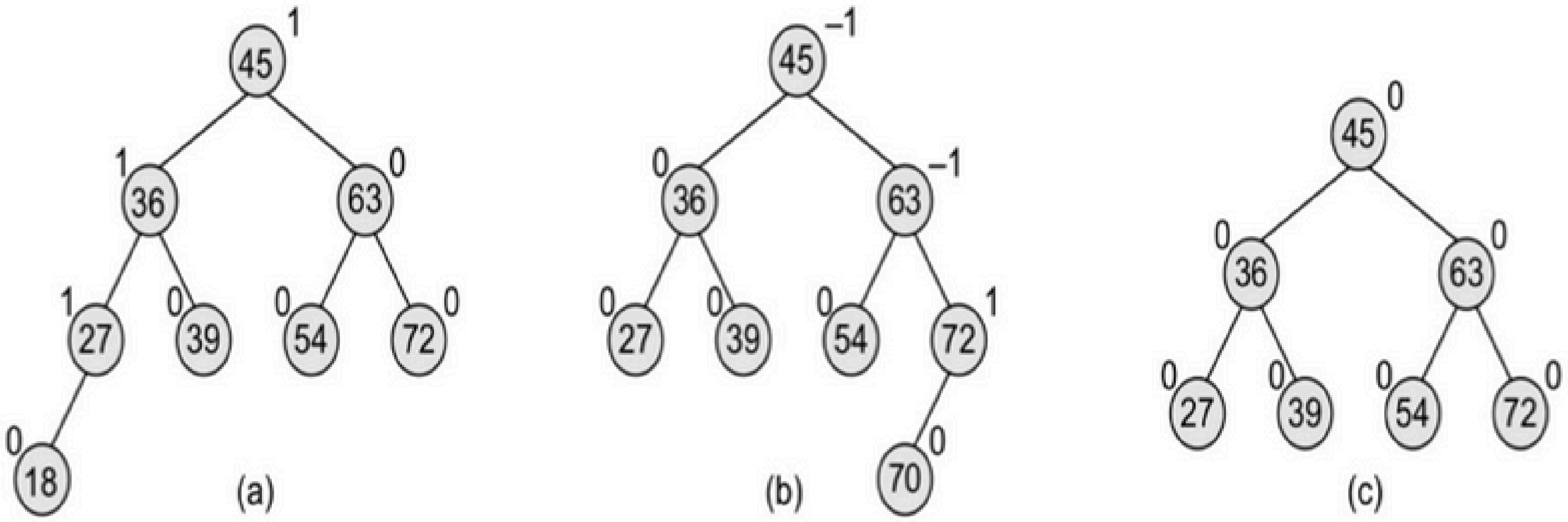
If the balance factor goes beyond this range... ⚠️
The node is unbalanced, and the tree needs to rebalance itself to maintain its structure.

```
Balance Factor = Height of Left Subtree - Height of Right Subtree
```

A node is "perfectly balanced" when its balance factor is:

- **-1 👉 slightly right-heavy**
- **0 👉 perfectly balanced**
- **+1 👉 slightly left-heavy**

SO... WHAT DO AVL TREES ACTUALLY LOOK LIKE ?



(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

INSERTION IN AVL

(INSERTING A NEW NODE IN AN AVL TREE)

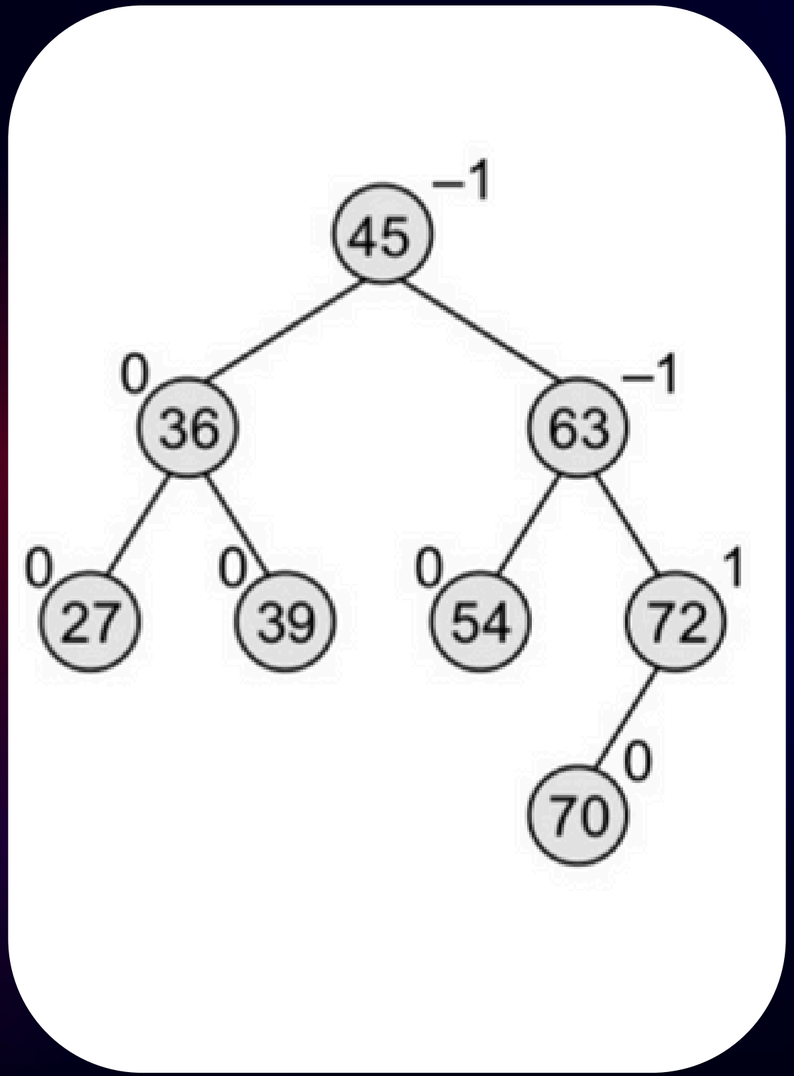
When you insert a new node in an AVL tree...

- **I**t always enters as a leaf 🍃 — and guess what?
- That leaf starts off perfectly balanced (Balance Factor = 0) ✅
- But the real drama begins on the way back up the tree... ⬆️🌳

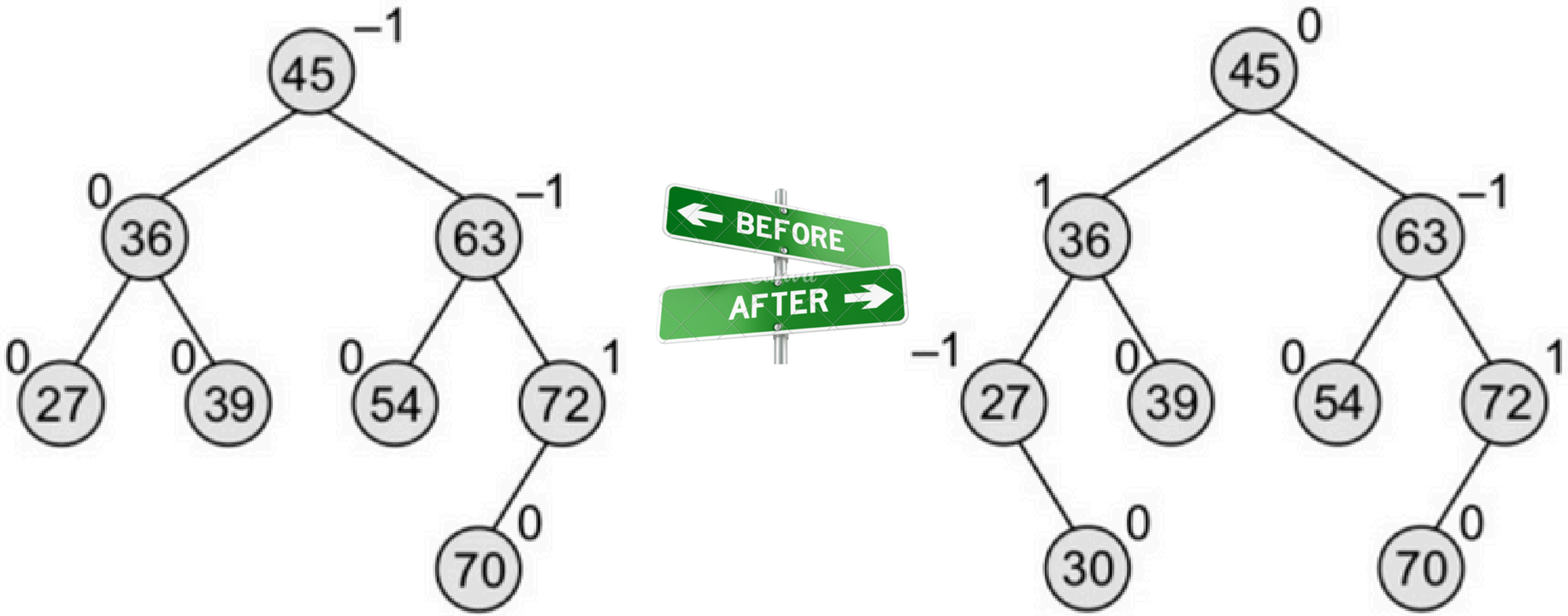
1. **The node was left- or right-heavy, but insertion restores balance!**
2. **The node was balanced before, but now becomes left- or right-heavy.**
3. **Oh no... Critical Situation! 🚨 The node was already heavy, and now you made it even heavier on the same side!**
   a. **Boom — now it's unbalanced and needs rebalancing.**
   b. **This node is called the Critical Node.**

🎯 What happens next?
The tree performs rotations like a ninja 🌀 to bring everything back into balance!

INSERT 30 IN THE FOLLWING TREE



INSERTION OF A NODE DOES NOT AFFECT THE BALANCE



ROTATION

🔁 The Four Rotational Remedies:

1. LL Rotation (Left-Left)
❎ The new node is added in the left child's left sub-tree.
🔧 Rotate right around the critical node.
Think: Heavy on the left-left side → fix with a right spin! 🌀

2. RR Rotation (Right-Right)
❎ The new node is added in the right child's right sub-tree.
🔧 Rotate left around the critical node.
Think: Heavy on the right-right side → fix with a left spin! 🔄

3. LR Rotation (Left-Right)
❎ The new node is added in the right sub-tree of the left child.
🔧 First left rotate, then right rotate (Double Rotation!)
Think: It's a zigzag on the left side — needs double twist! 🤸‍♂️

4. RL Rotation (Right-Left)
❎ The new node is added in the left sub-tree of the right child.
🔧 First right rotate, then left rotate (Double Rotation!)
Think: It's a zigzag on the right side — needs double twist! 🤹

ROTATION

🔁 The Four Rotational Remedies:

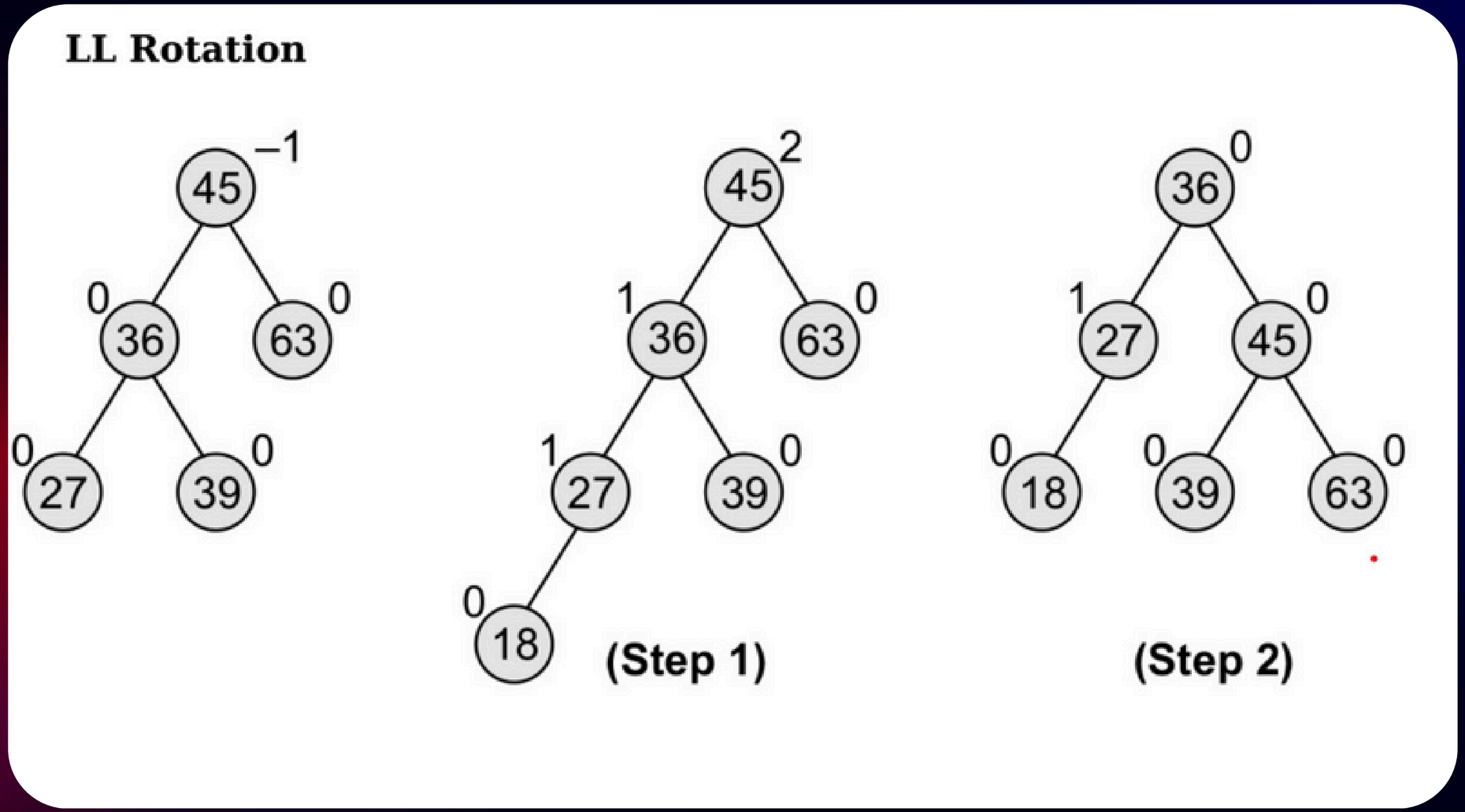
1. LL Rotation (Left-Left)
❎ The new node is added in the left child's left sub-tree.
🔧 Rotate right around the critical node.
Think: Heavy on the left-left side → fix with a right spin! 🌀

2. RR Rotation (Right-Right)
❎ The new node is added in the right child's right sub-tree.
🔧 Rotate left around the critical node.
Think: Heavy on the right-right side → fix with a left spin! 🔄

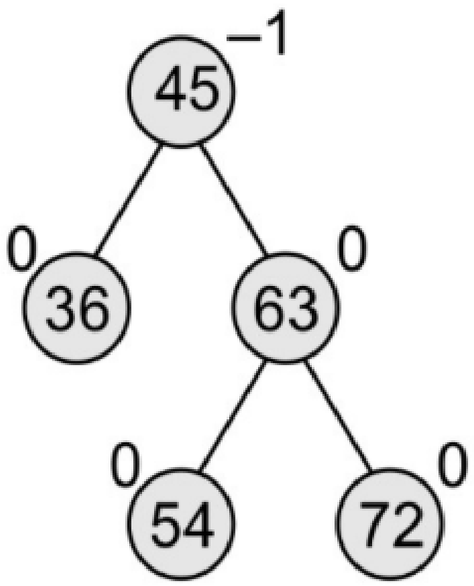
3. LR Rotation (Left-Right)
❎ The new node is added in the right sub-tree of the left child.
🔧 First left rotate, then right rotate (Double Rotation!)
Think: It's a zigzag on the left side — needs double twist! 🤸‍♂️

4. RL Rotation (Right-Left)
❎ The new node is added in the left sub-tree of the right child.
🔧 First right rotate, then left rotate (Double Rotation!)
Think: It's a zigzag on the right side — needs double twist! 🤹

LL ROTATION

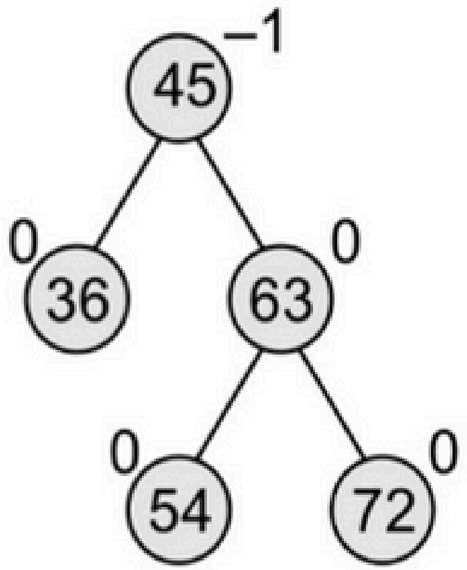

LL Rotation
45
−1
0
36
63
0
0
27
39
0
45
2
1
36
63
0
1
27
39
0
0
18
(Step 1)
36
0
1
27
45
0
0
18
0
39
63
0
(Step 2)

RR ROTATION

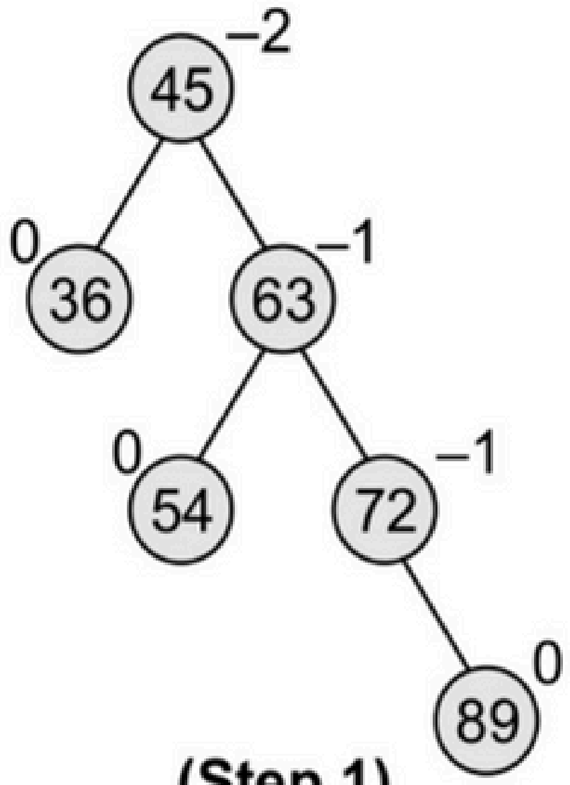


INSERT 89

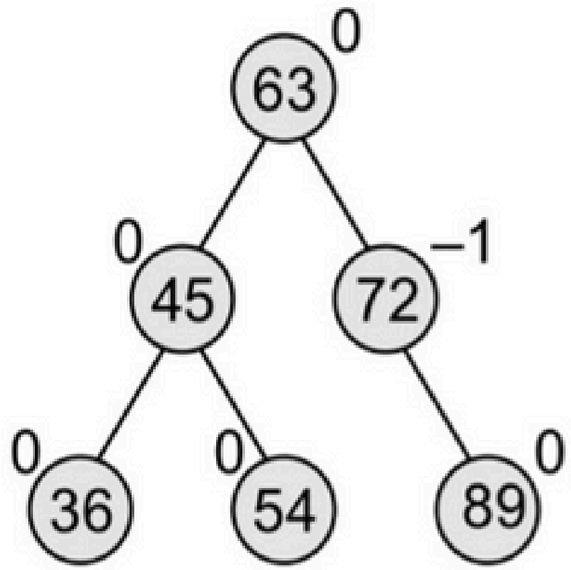
RR ROTATION



INSERT 89

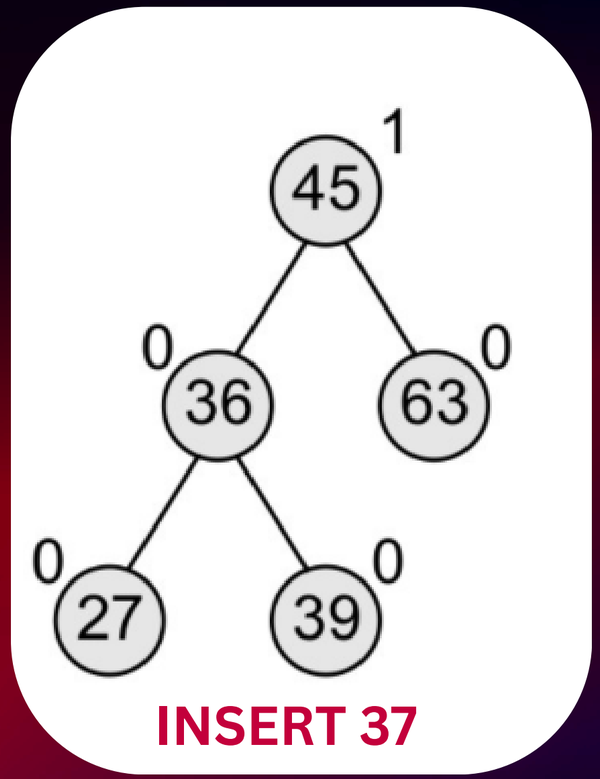


(Step 1)

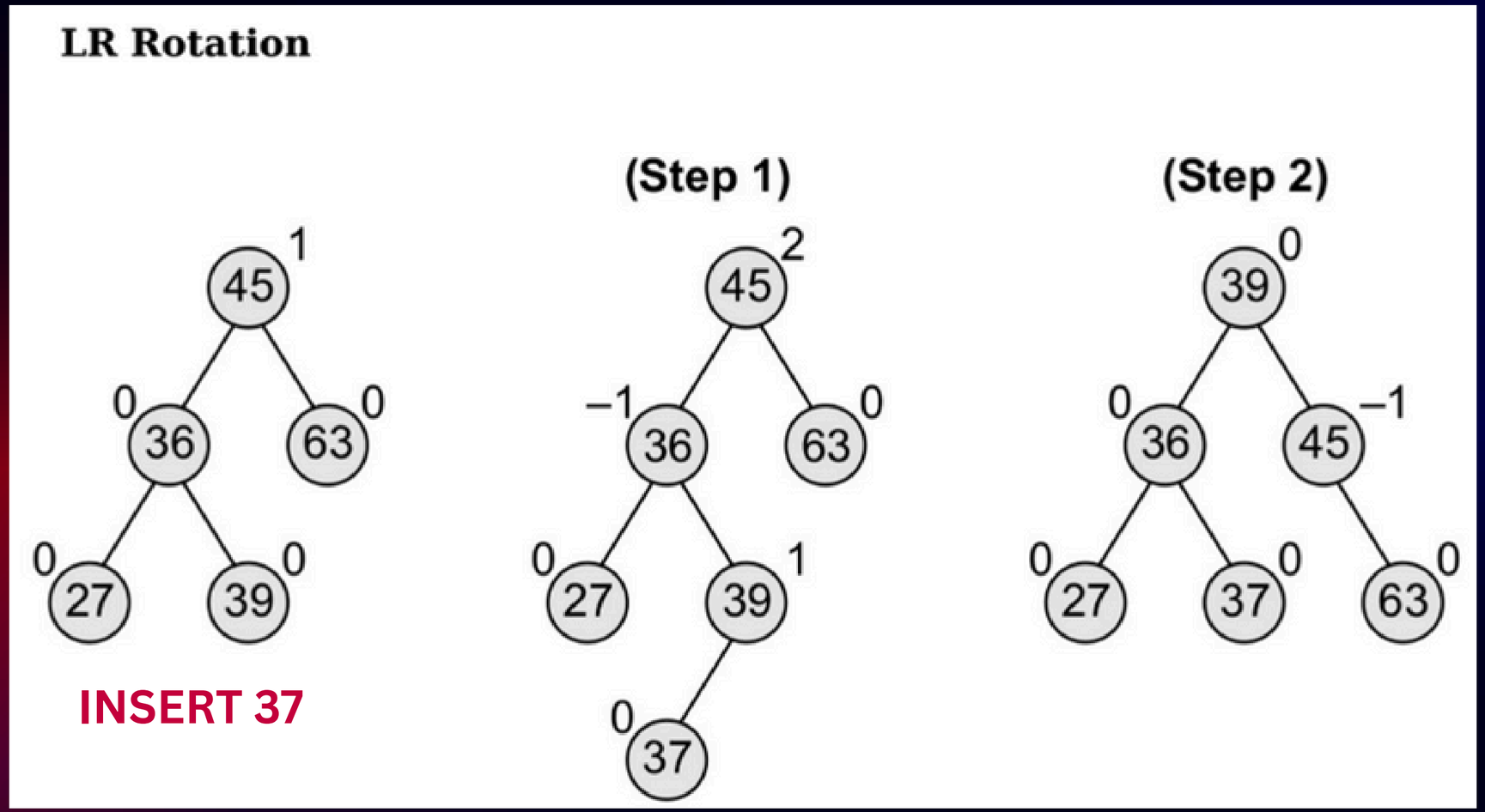


(Step 2)

LR ROTATION

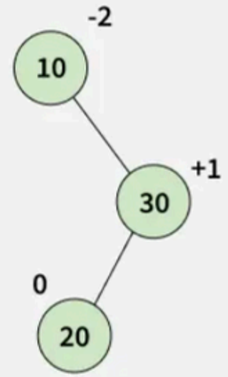


LR ROTATION



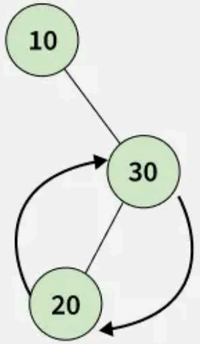
RL ROTATION

01 Step | Unbalanced tree



Node '10' has a balance factor of -2, which lies outside the acceptable range of -1 to +1.

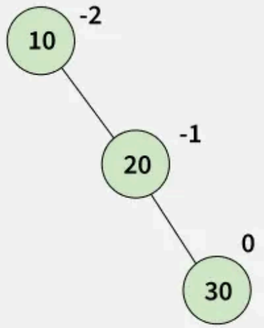
02 Step | Performing right rotation



To achieve balance in the tree, a right rotation is performed on nodes 30 and 20

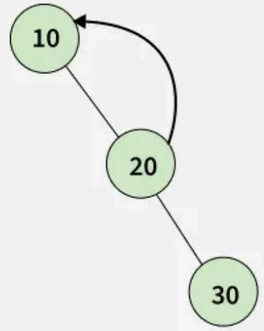
RL ROTATION

03 Step | Unbalanced tree



we need to perform left rotation on nodes 10 and 20 to balanced the tree

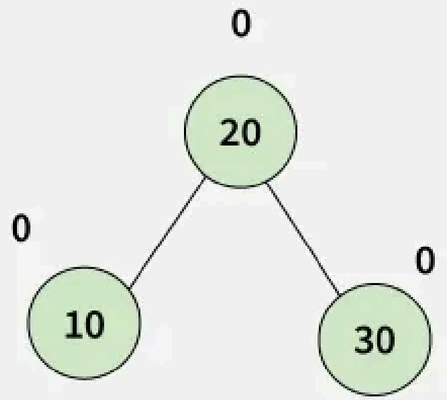
04 Step | Performing left rotation



To achieve balance in the tree, a left rotation is performed on nodes 10 and 20

RL ROTATION

05 Step | Balanced tree



The tree is balanced now, with all node balance factors within the valid range.

TIME FOR NET PRACTICE

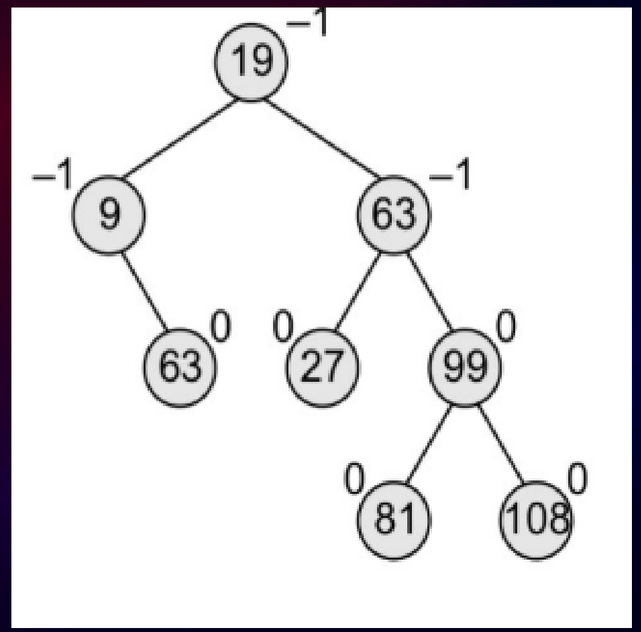
Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81

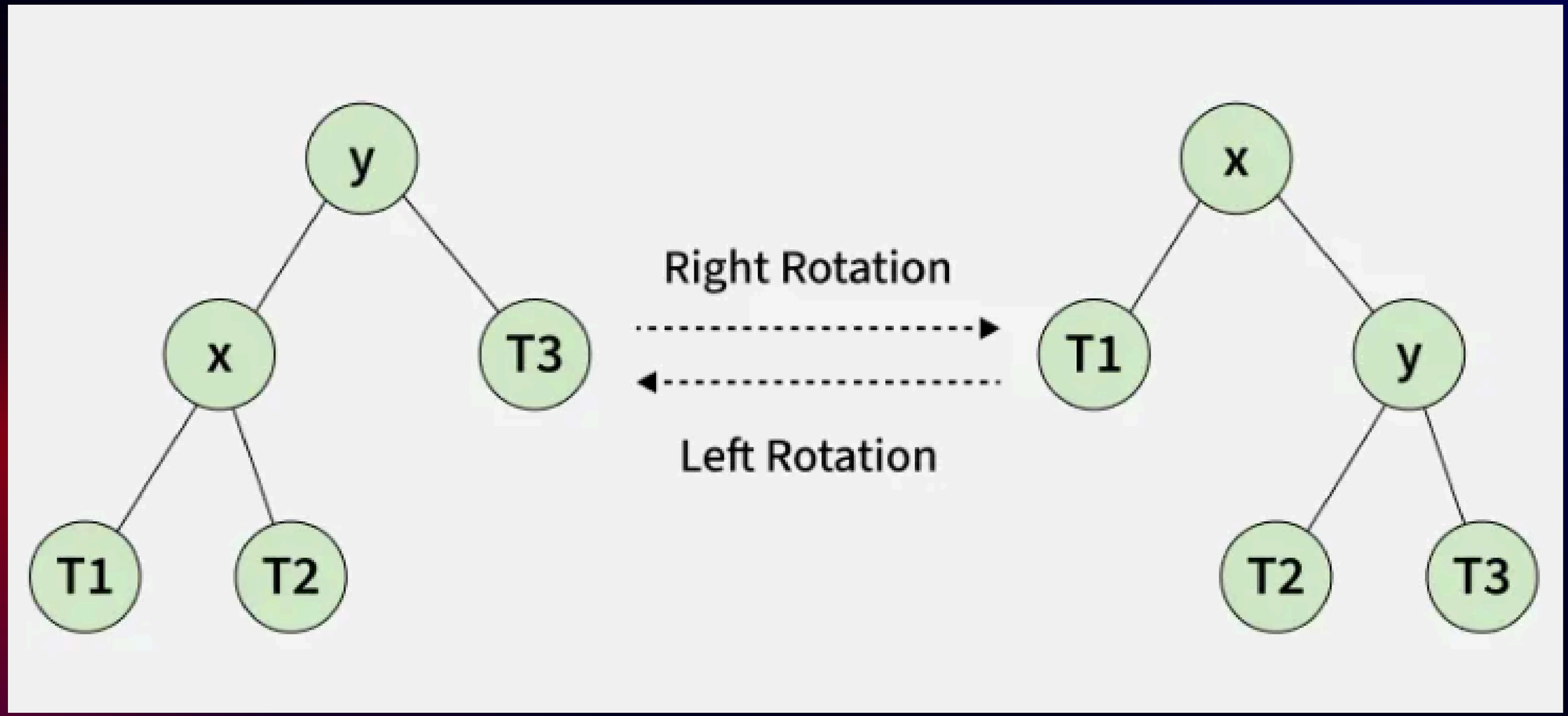
TIME FOR NET PRACTICE

Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81



POINT TO BE NOTED!!!



DELETION IN AVL

(DELETING A NODE IN AN AVL TREE)

🔍 **Step-by-Step Process:**

1. **Delete the node like a regular BST**
2. **After deletion, update the height of all affected ancestors**
3. **Check the balance factor of each node**
4. **If any node is unbalanced, apply rotation(s) to rebalance**

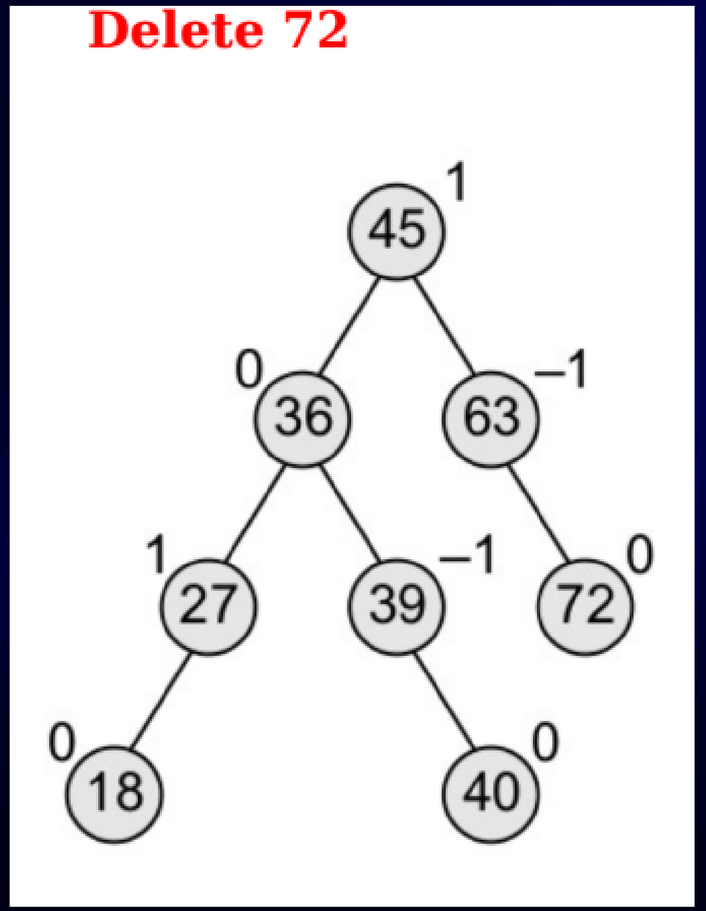
🎯 **What happens next?**
The tree performs rotations like a ninja 🌀 to bring everything back into balance!

R0 ROTATION

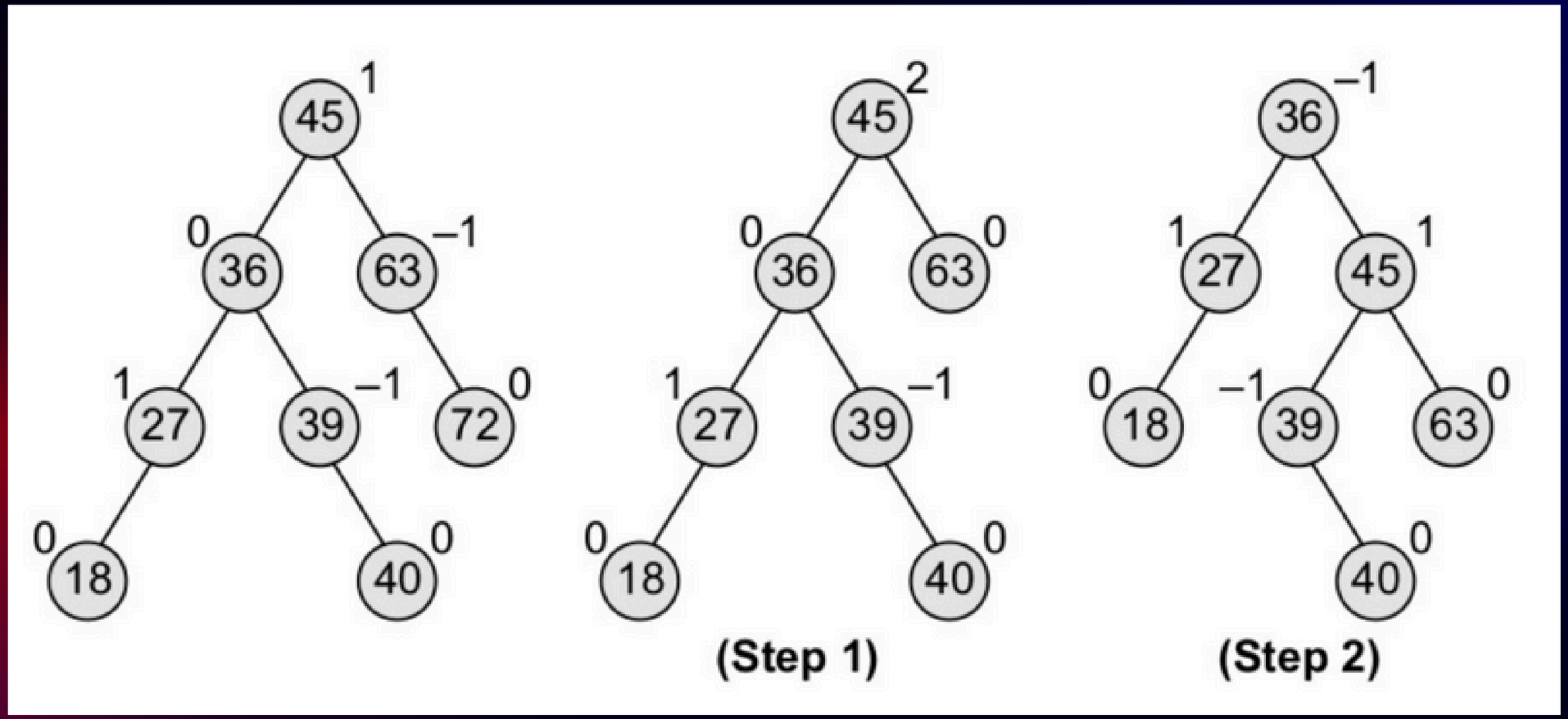
R0 – Single Rotation (Simple Rotate)
When to Use:

- **LL Case → Right Rotate**
- **RR Case → Left Rotate**

🧠 “R0 is like a quick fix. Tree says: ‘I’m a bit off-balance, just twist me once!’”



R0 ROTATION


45
1
0
36
63
−1
1
27
39
−1
72
0
0
18
40
0
(Step 1)
45
2
0
36
63
0
1
27
39
−1
0
18
40
0
(Step 2)
36
−1
1
27
45
1
0
18
−1
39
63
0
40
0

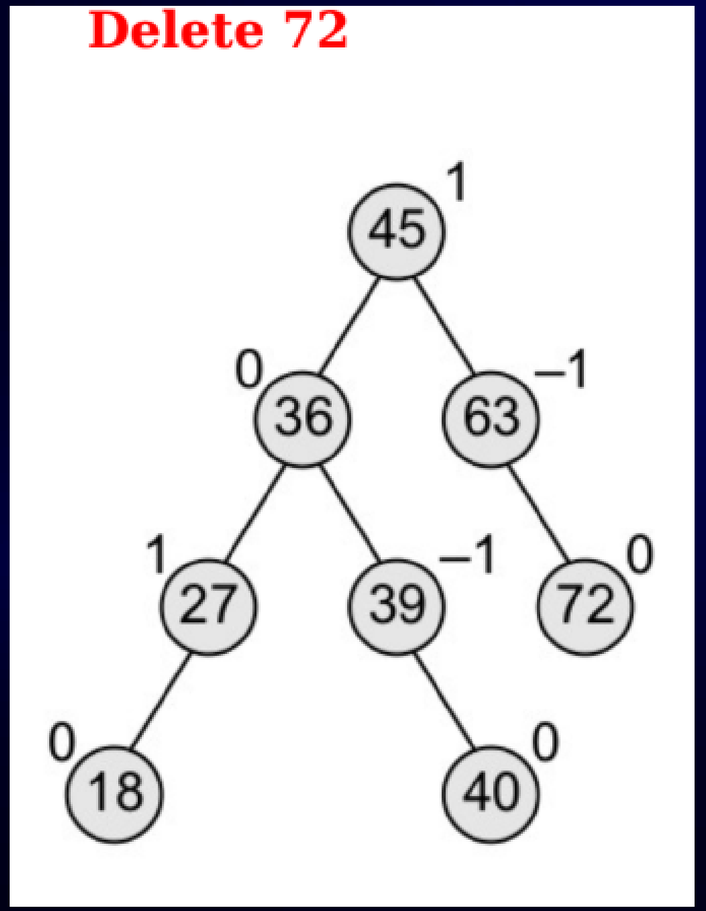
R1 ROTATION

R1 – Double Rotation (Two-step Rotate)

When to Use:

- **LR Case → Left at left child, then Right**
- **RL Case → Right at right child, then Left**

🧠 **"R1 is like untangling twisted earphones — takes two steps to fix!"**

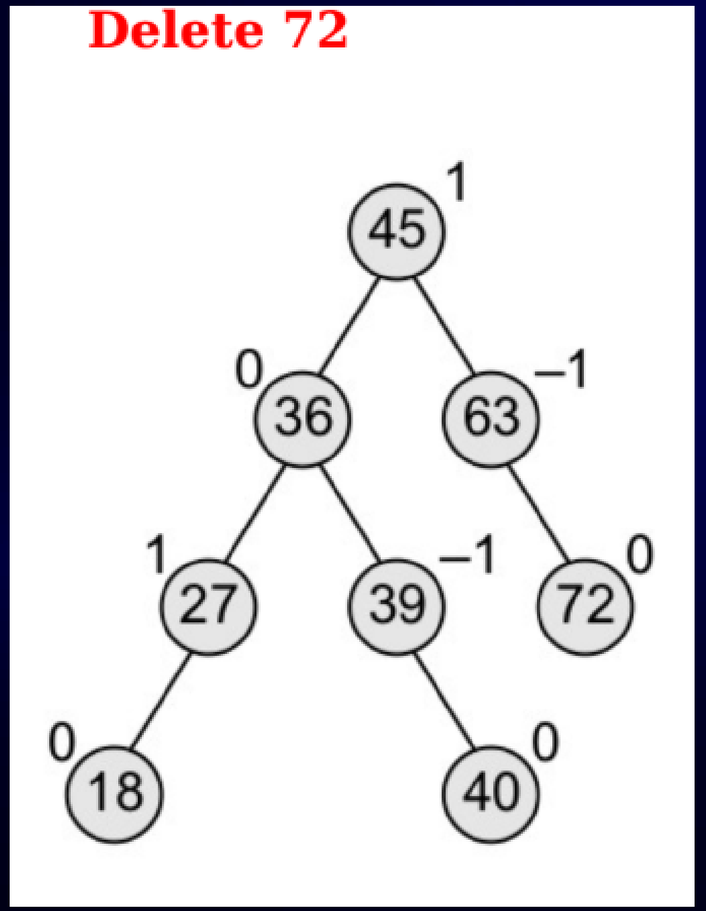


R1 ROTATION

R1 – Double Rotation (Two-step Rotate)
When to Use:

- **LR Case → Left at left child, then Right**
- **RL Case → Right at right child, then Left**

🧠 **"R1 is like untangling twisted earphones — takes two steps to fix!"**

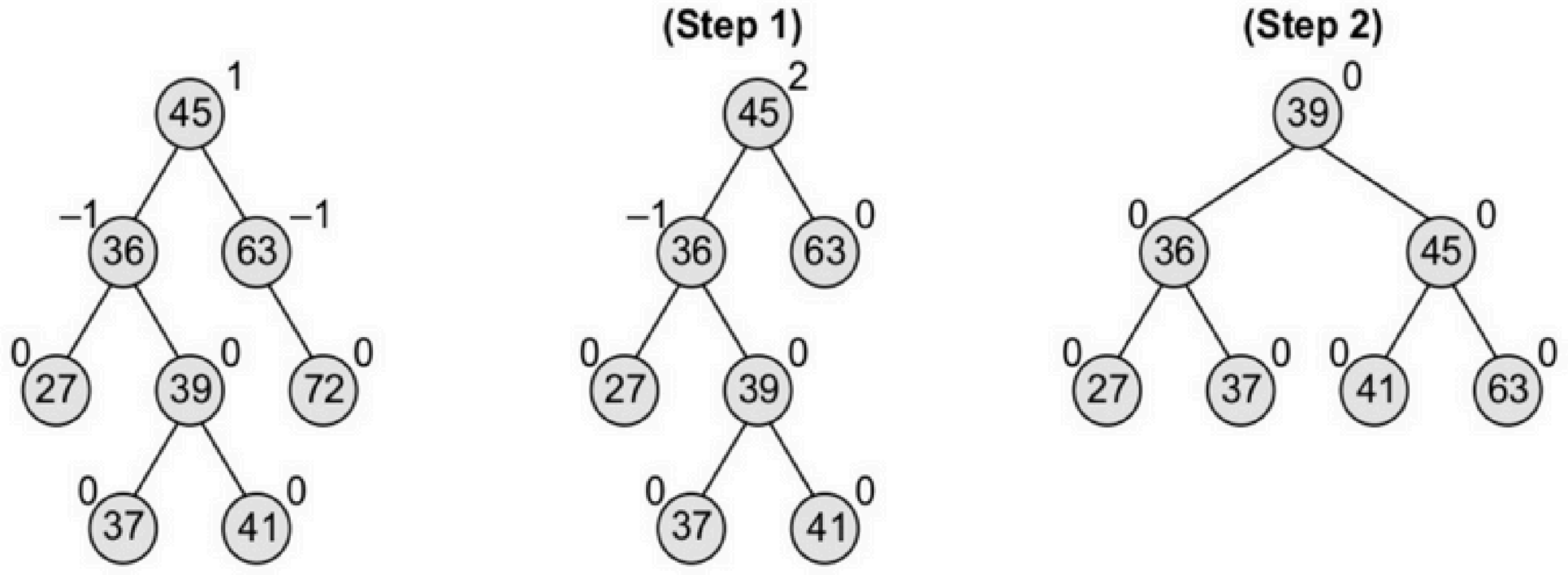


Why R1 ?

When the node is left-heavy, but its left child is right-heavy, we can't fix it with a single rotation — we need double (R1)

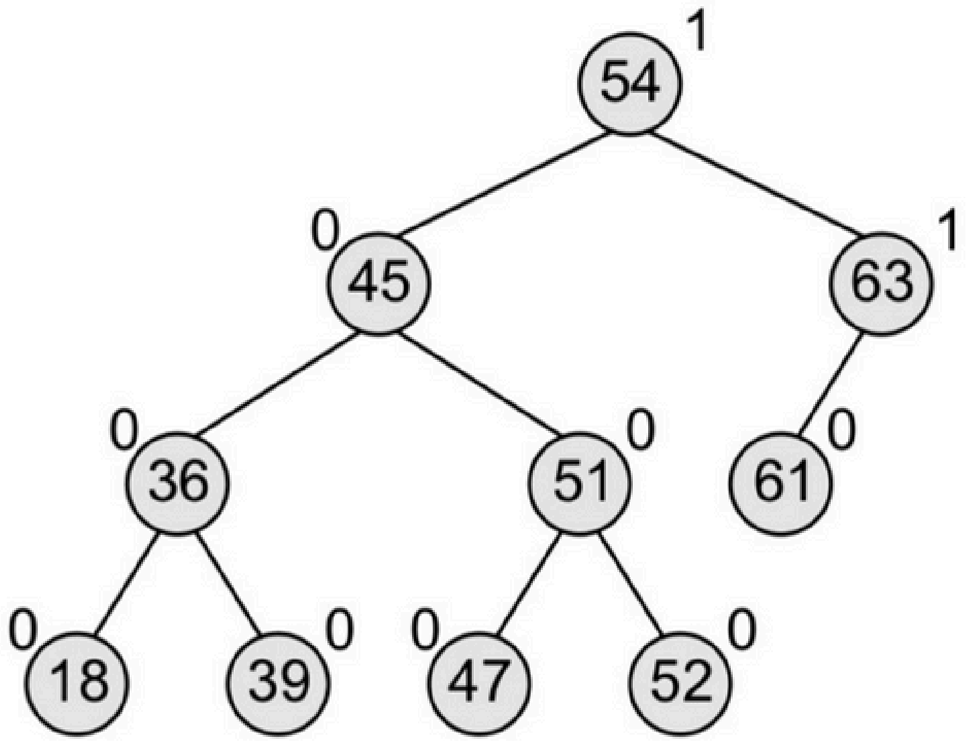
R1 ROTATION

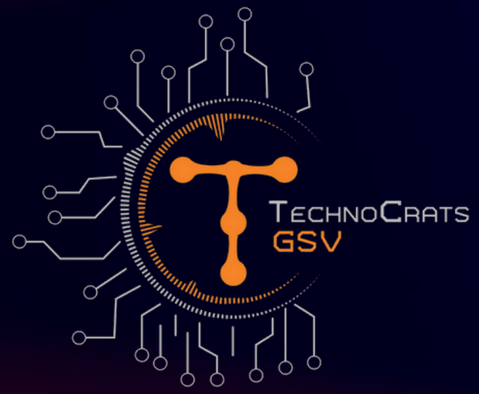
Delete 72



TIME FOR NET PRACTICE

Delete nodes 52, 36, and 61 from the AVL tree given.





Thank You