Faculty of Engineering and Tecnology

Computer Science Department

Trees_3

AVL Trees

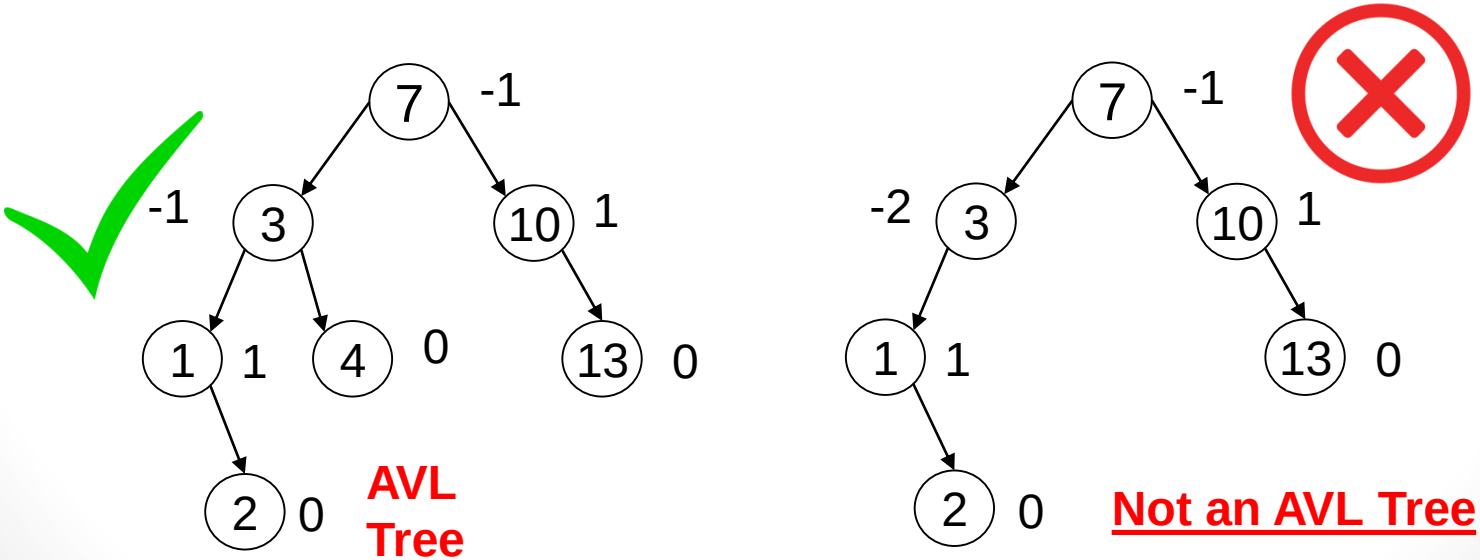
BIRZEIT UNIVERSITY

AVL Trees

- Introduction
- What is an AVL Tree?
- AVL Tree Implementation.
- Why AVL Trees?
- Rotations.

What is an AVL Tree?

- An AVL (Adel'son, Vel'skii, & Lands) tree is a binary search tree with a height balance property:
  - For each node v, the heights of the subtrees of v differ by at most 1.
- A subtree of an AVL tree is also an AVL tree.
- An AVL node can have a balance factor of -1, 0, or +1.



Why AVL Trees?

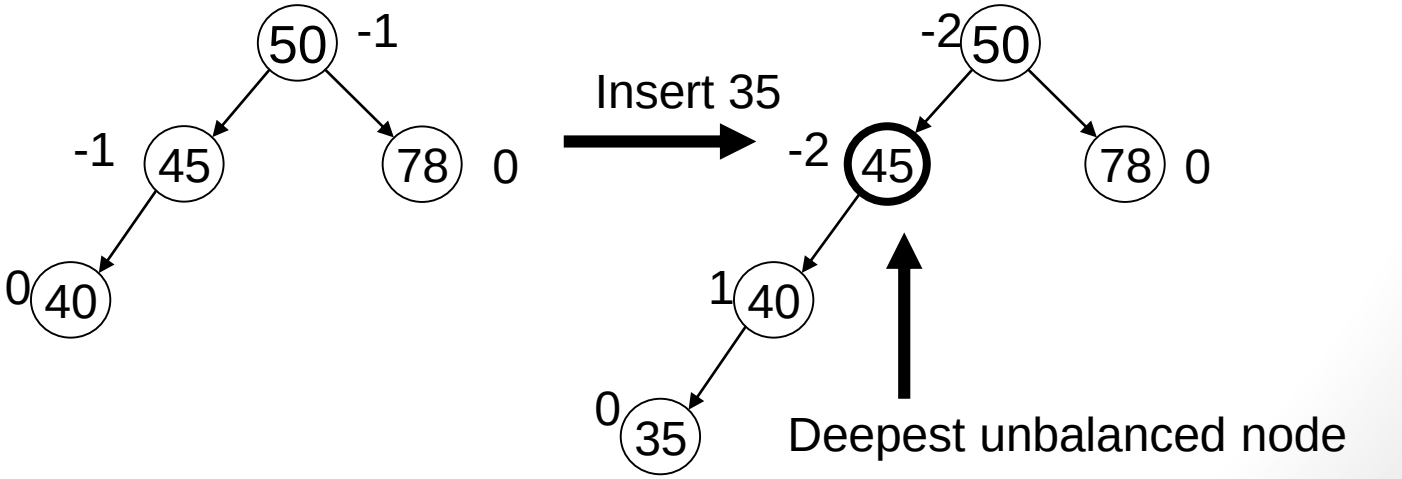
- Insertion or deletion in an ordinary Binary Search Tree can cause large imbalances.
- In the worst case searching an imbalanced Binary Search Tree is O(n).
- An AVL tree is rebalanced after each insertion or deletion.
  - The height-balance property ensures that the height of an AVL tree with n nodes is O(log n).
  - Searching, insertion, and deletion are all O(log n).

What is a Rotation?

- **A rotation is a process** of switching children and parents among two or three adjacent nodes to restore balance to a tree.

- **An insertion or deletion may cause an imbalance in an AVL tree.**

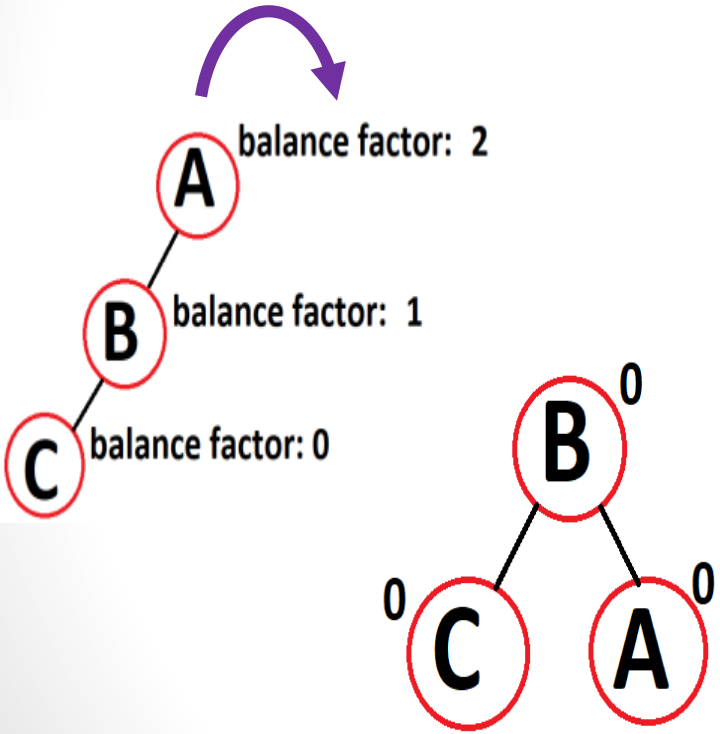
- The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to **-2 or +2** requires rotation to rebalance the tree.



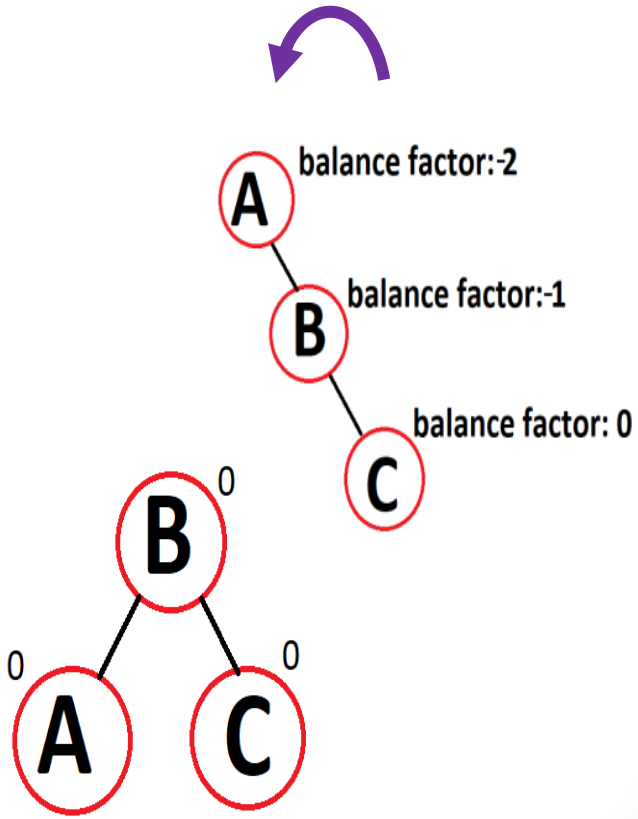
Single Rotation

- There are two kinds of single rotation:

Right Rotation.

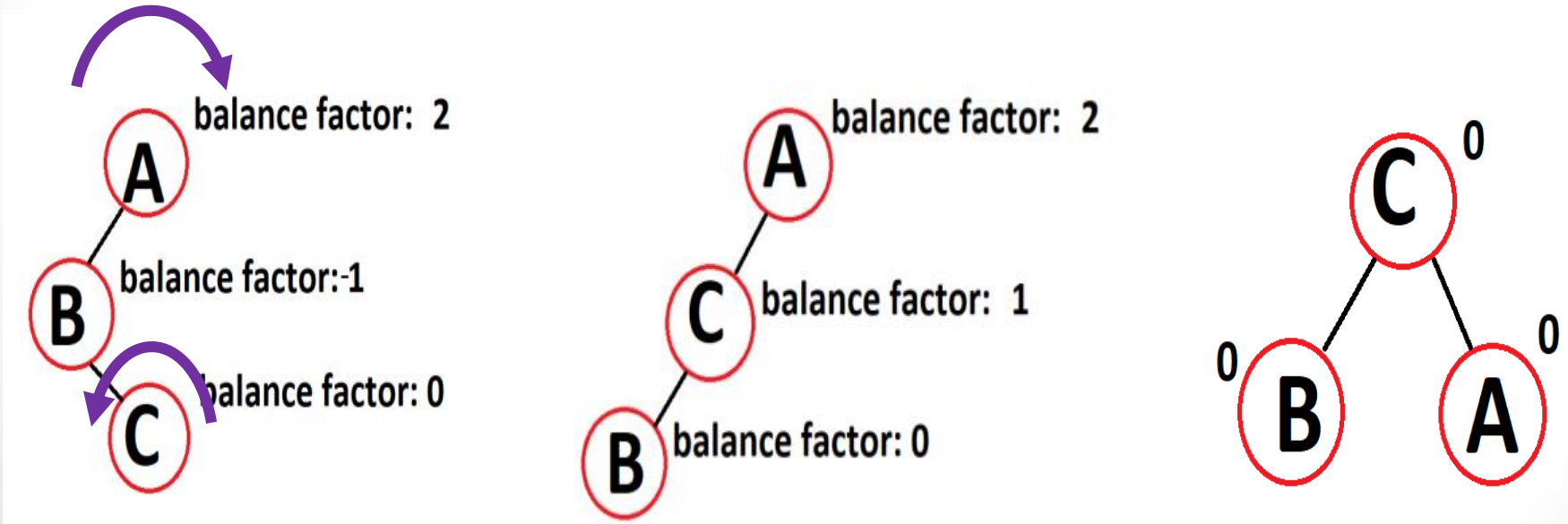


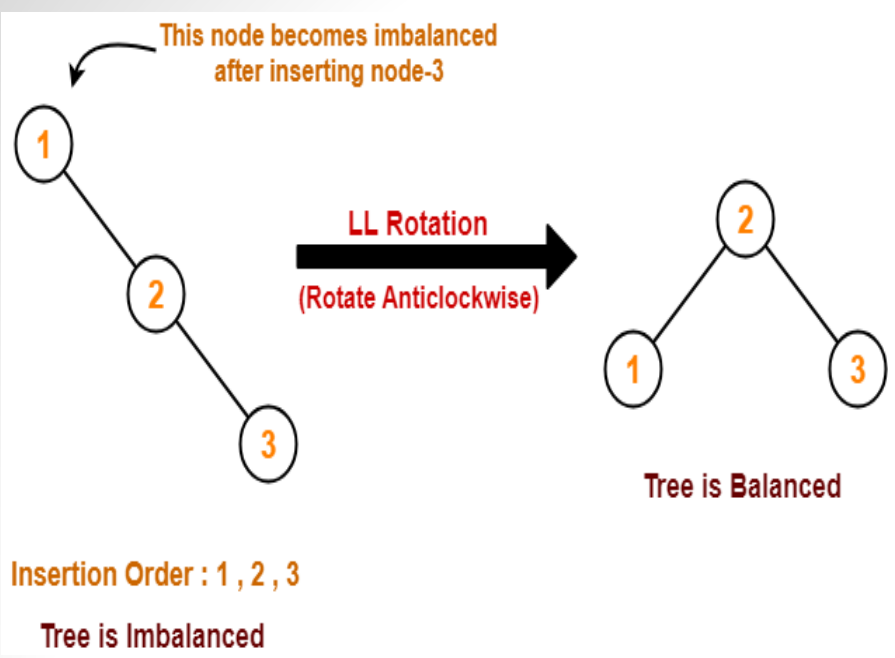
Left Rotation.



Double Rotation

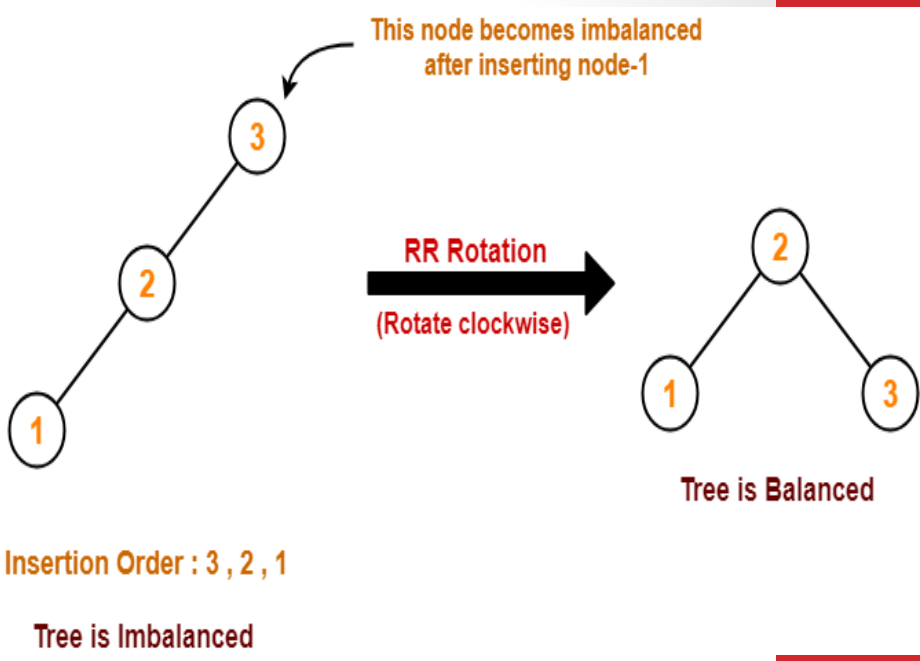
- A double **right-left :**rotation is a **right rotation** followed by a **left rotation.**
- A double **left-right :**rotation is a **left rotation** followed by a **right rotation**.





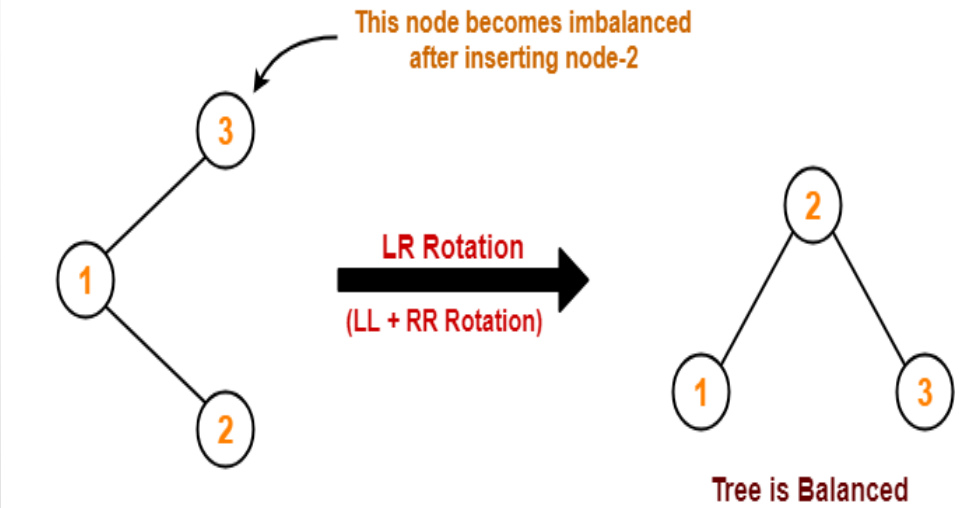
Insertion Order : 1 , 2 , 3

Tree is Imbalanced



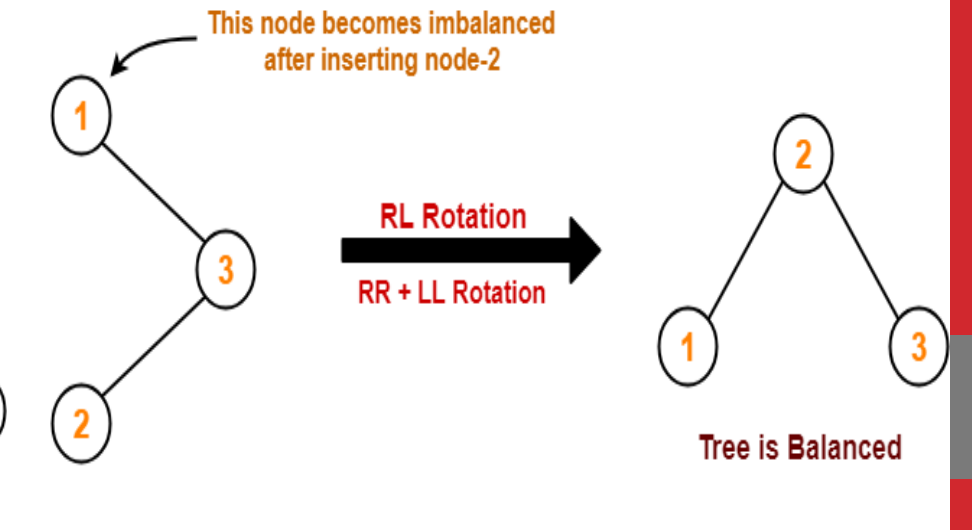
Insertion Order : 3 , 2 , 1

Tree is Imbalanced



Insertion Order : 3 , 1 , 2

Tree is Imbalanced

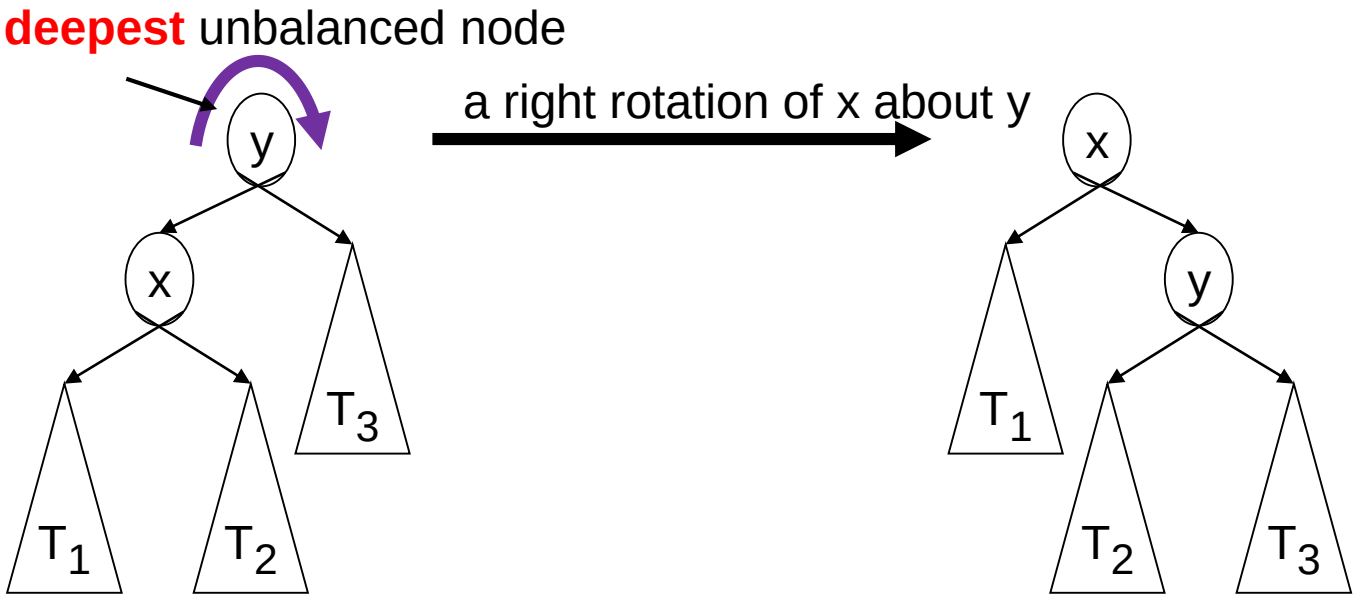


Insertion Order : 1 , 3 , 2

Tree is Imbalanced

Single Right Rotation

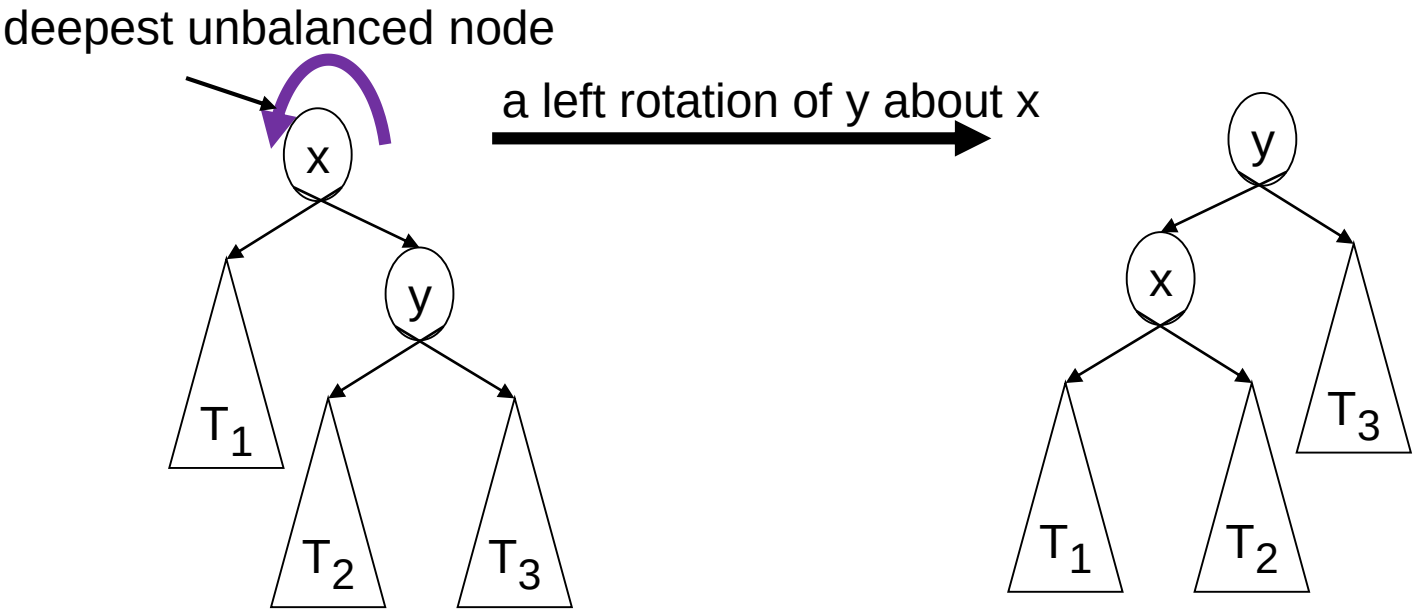
- Single right rotation:
  - The left child x of a node y becomes y's parent.
  - y becomes the right child of x.
  - The right child T_2 of x, **if any**, becomes the left child of y.



Note: The **pivot** of the rotation is the deepest unbalanced node

Single Left Rotation

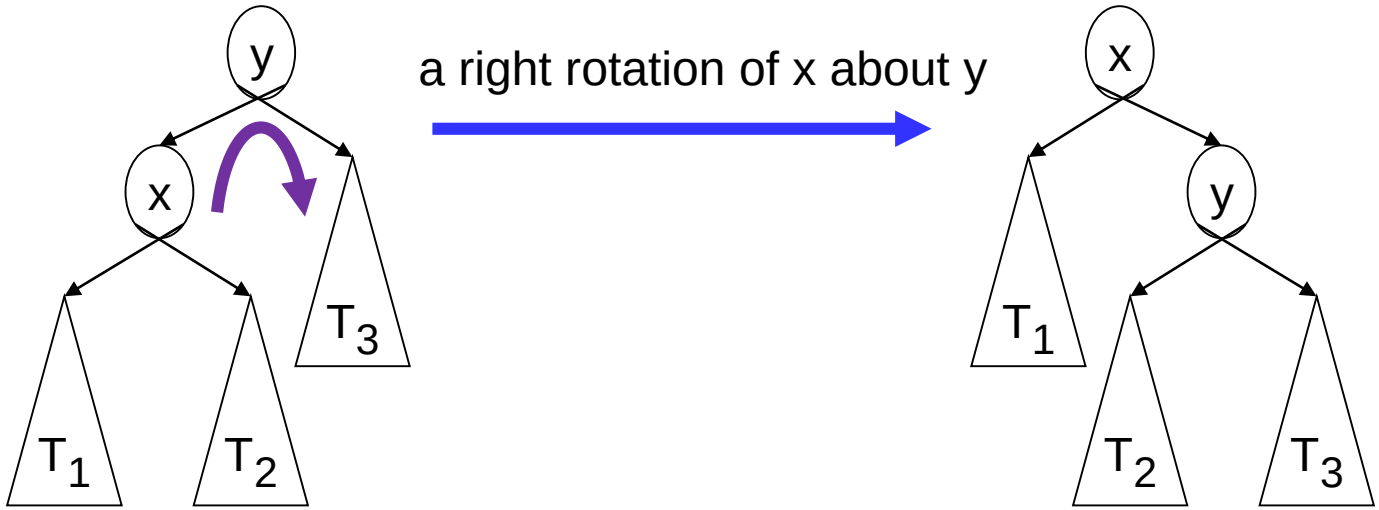
- Single left rotation:
  - The right child y of a node x becomes x's parent.
  - x becomes the left child of y.
  - The left child T_2 of y, **if any**, becomes the right child of x.



Note: The **pivot** of the rotation is the deepest unbalanced node

BST ordering property

- A rotation does not affect the ordering property of a BST.



BST ordering property requirement:

$T_1 < x < y$

$x < T_2 < y$

$x < y < T_3$

Similar

BST ordering property requirement:

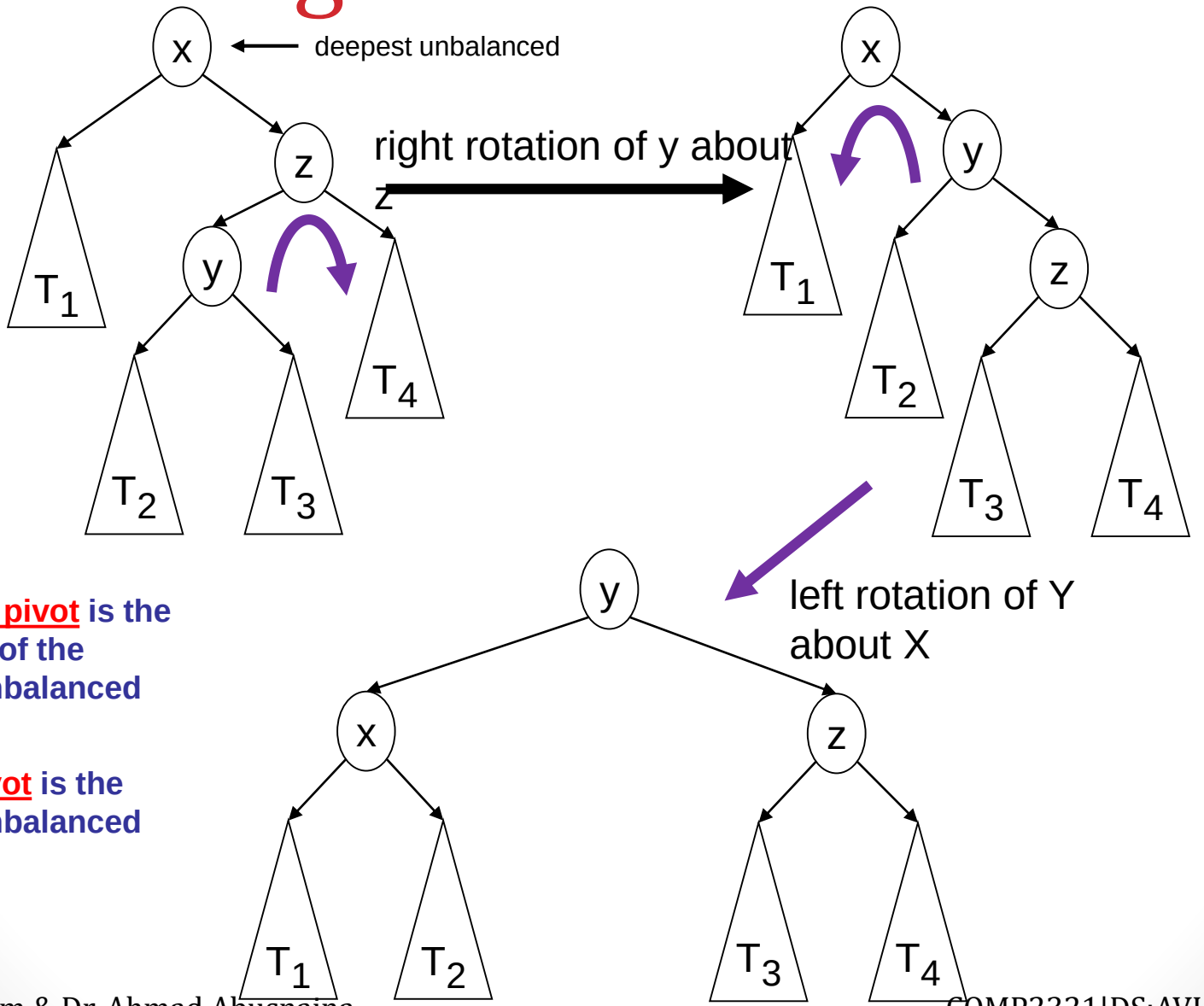
$T_1 < x < y$

$x < T_2 < y$

$x < y < T_3$

- Similarly for a left rotation.

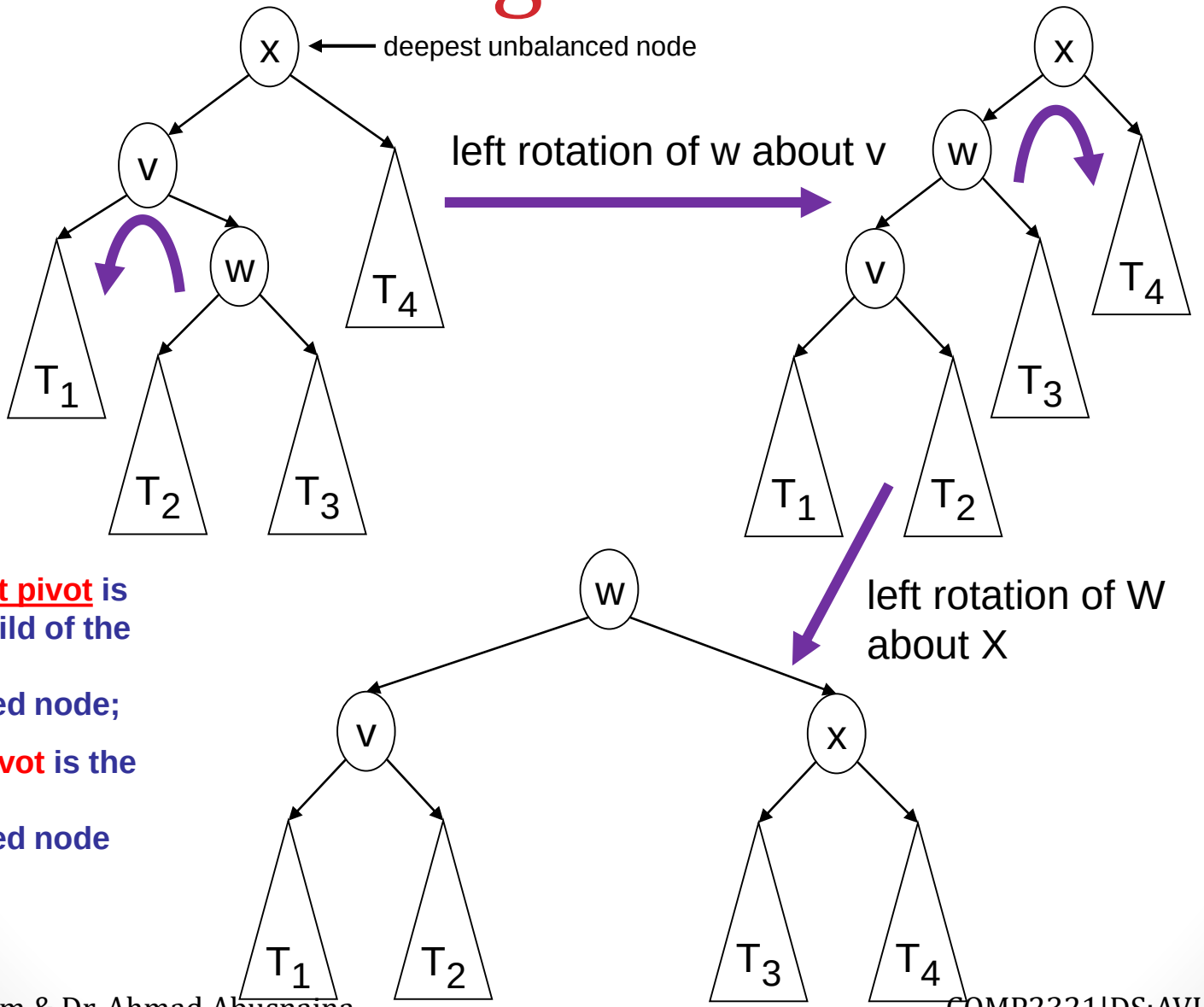
Double Right-Left Rotation



Note: First pivot is the right child of the deepest unbalanced node;

second pivot is the deepest unbalanced node

Double Left-Right Rotation



Note: First pivot is the left child of the deepest unbalanced node;

second pivot is the deepest unbalanced node

AVL Search Trees

- Inserting in an AVL tree
- Insertion implementation
- Deleting from an AVL tree

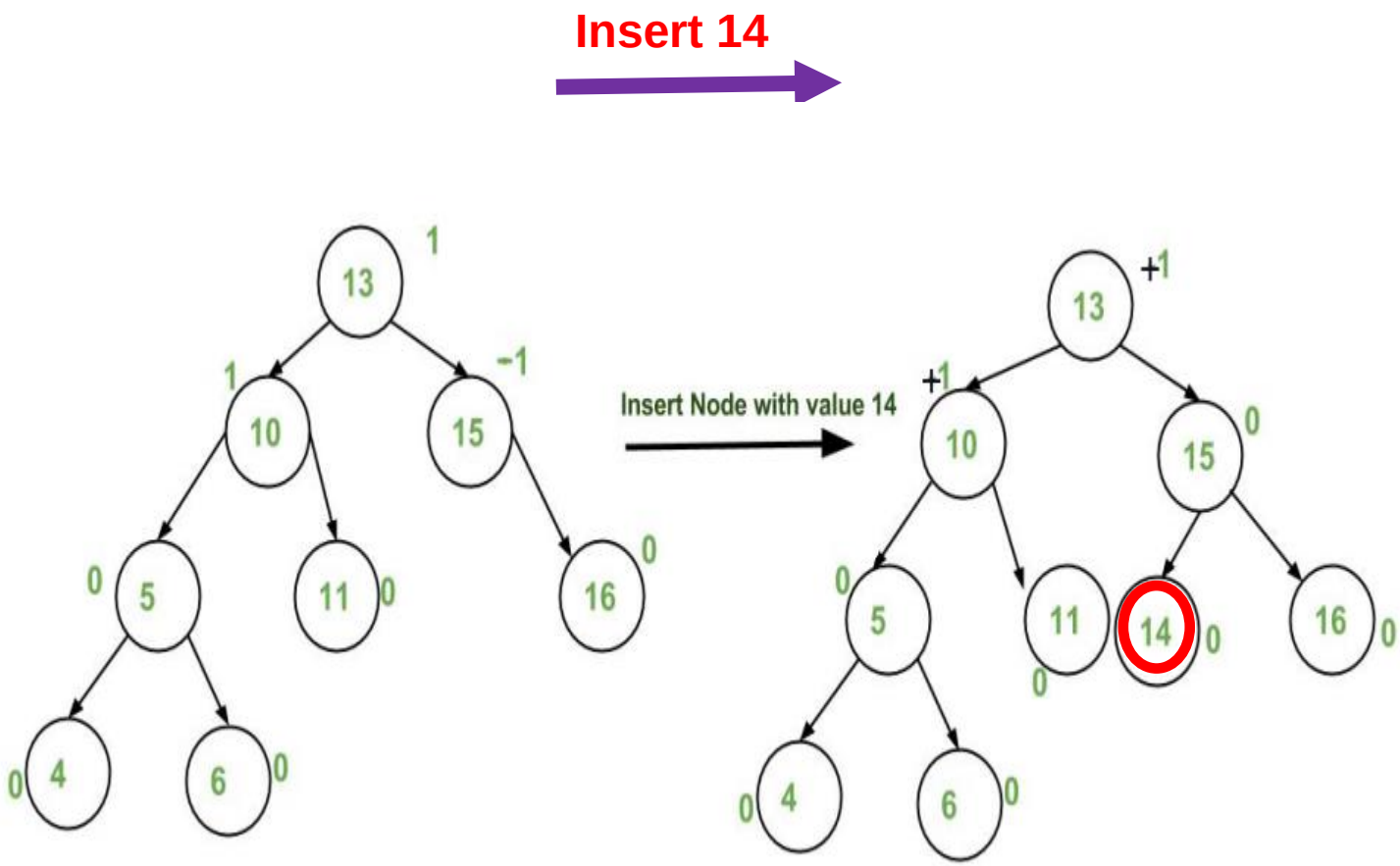
Insertion

- Insert using a BST insertion algorithm.
- Rebalance the tree if an imbalance occurs.
- An imbalance occurs if a node's balance factor changes from -1 to -2 or from+1 to +2.
- Rebalancing is done at the deepest unbalanced ancestor of the inserted node.

- **There are three insertion cases:**
  1. Insertion that does not cause an imbalance.
  2. Same side (**left-left** or **right-right**) insertion that causes an imbalance.
     - Requires a single rotation to rebalance.
  3. Opposite side **(left-right or right-left)** insertion that causes an imbalance.
     - Requires a double rotation to rebalance.

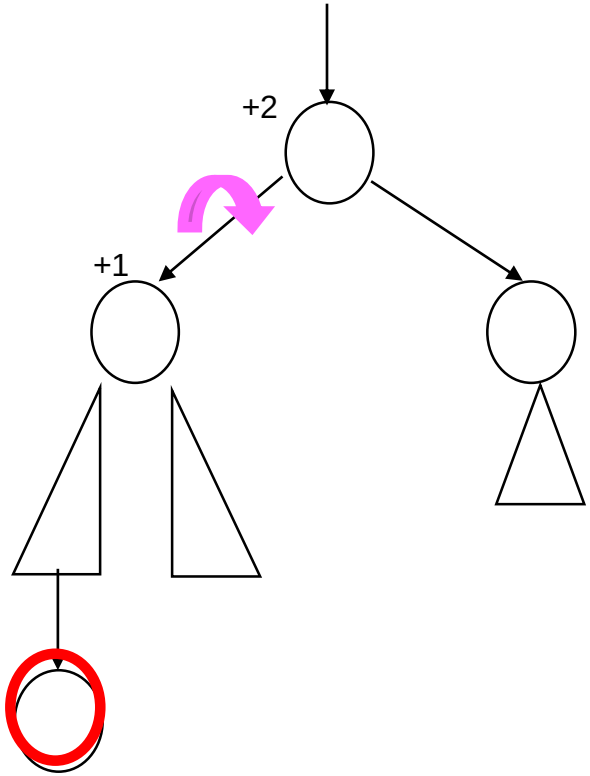
Insertion: case 1

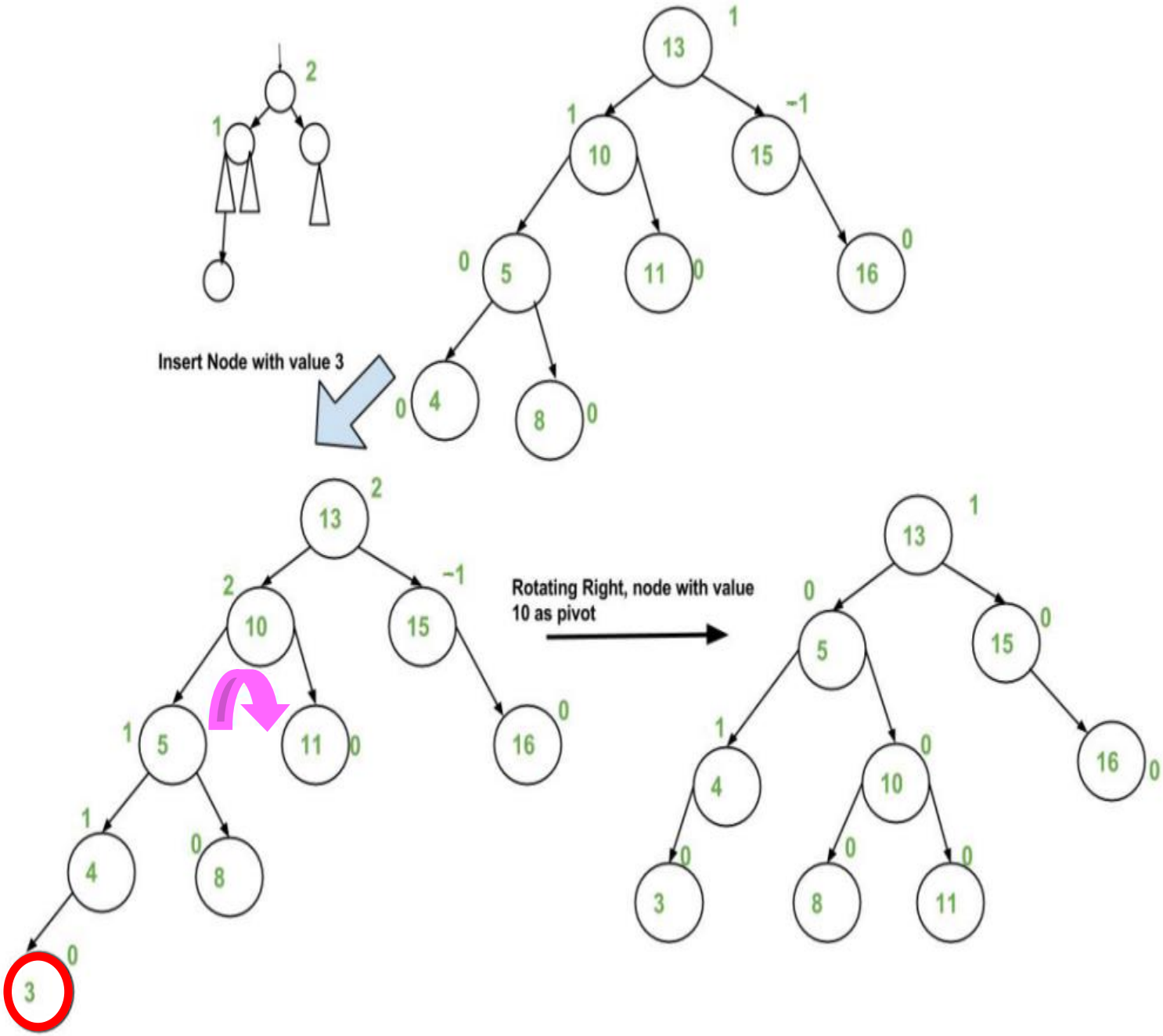
- Example: An insertion that does not cause an imbalance.



Insertion: case 2

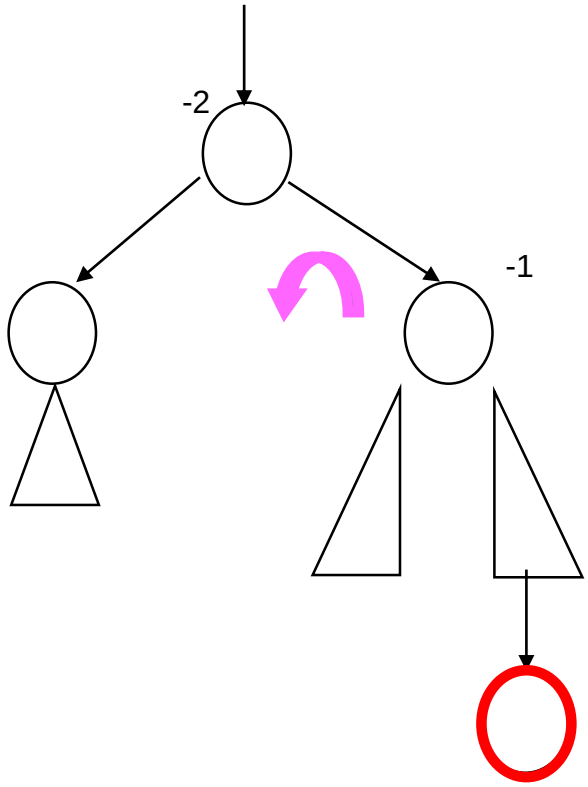
- **Case 2a**: The lowest node (with a balance factor of -2) had a taller left-subtree and the insertion was on the left-subtree of its left child.
- Requires single right rotation to rebalance.



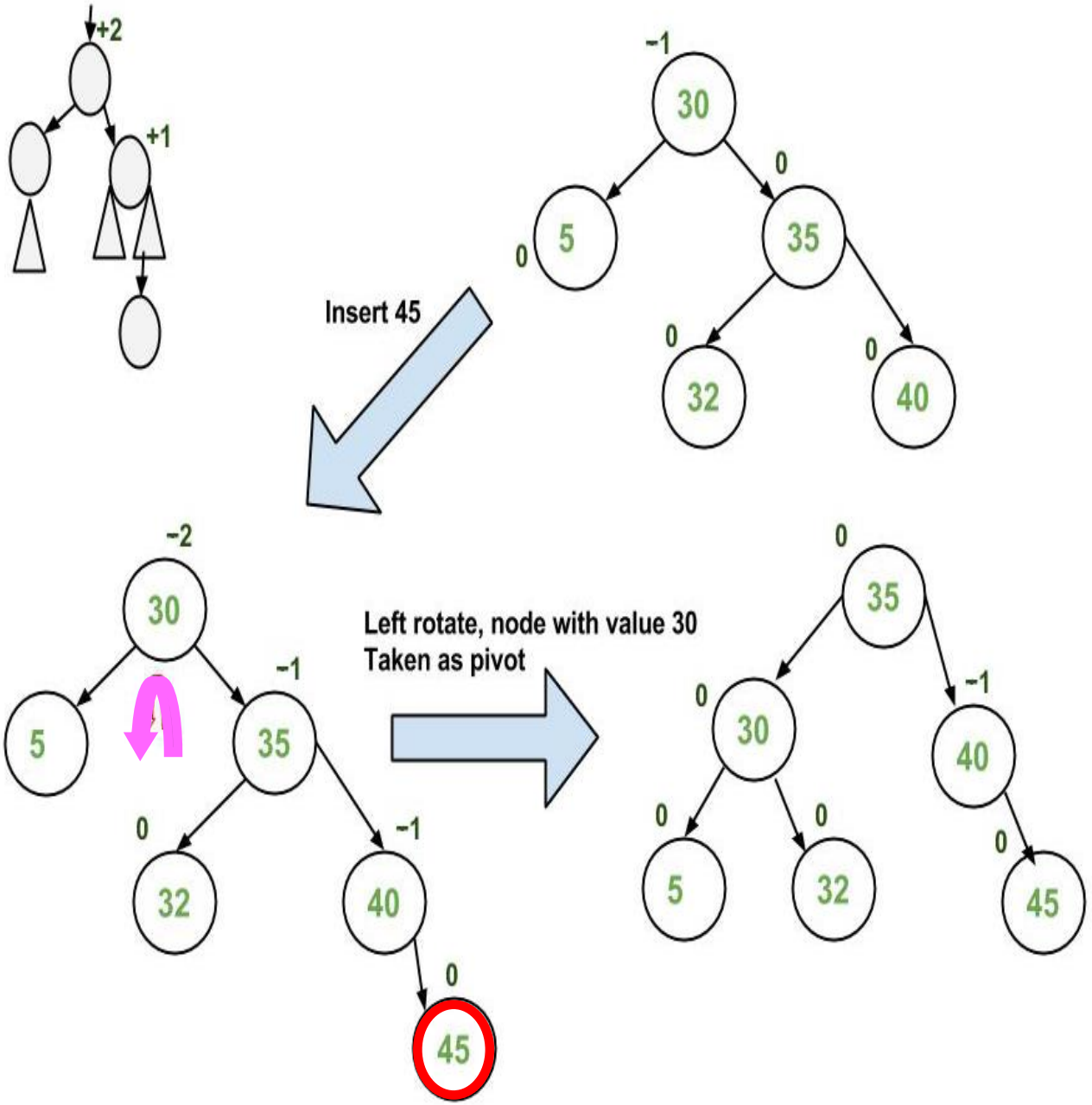

Insert Node with value 3
Rotating Right, node with value 10 as pivot

Insertion: case 2 (contd)

- Case 2b: The lowest node (with a balance factor of +2) had a taller right-subtree and the insertion was on the right-subtree of its right child.
- Requires single left rotation to rebalance.

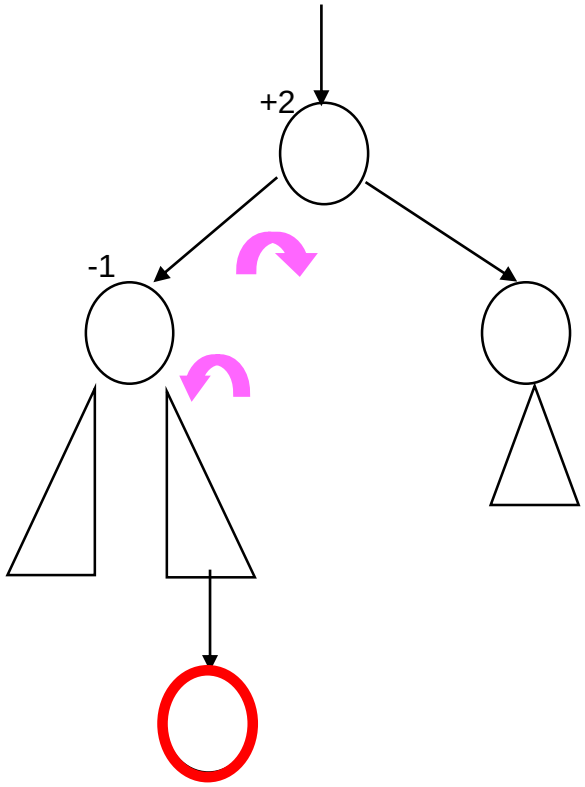


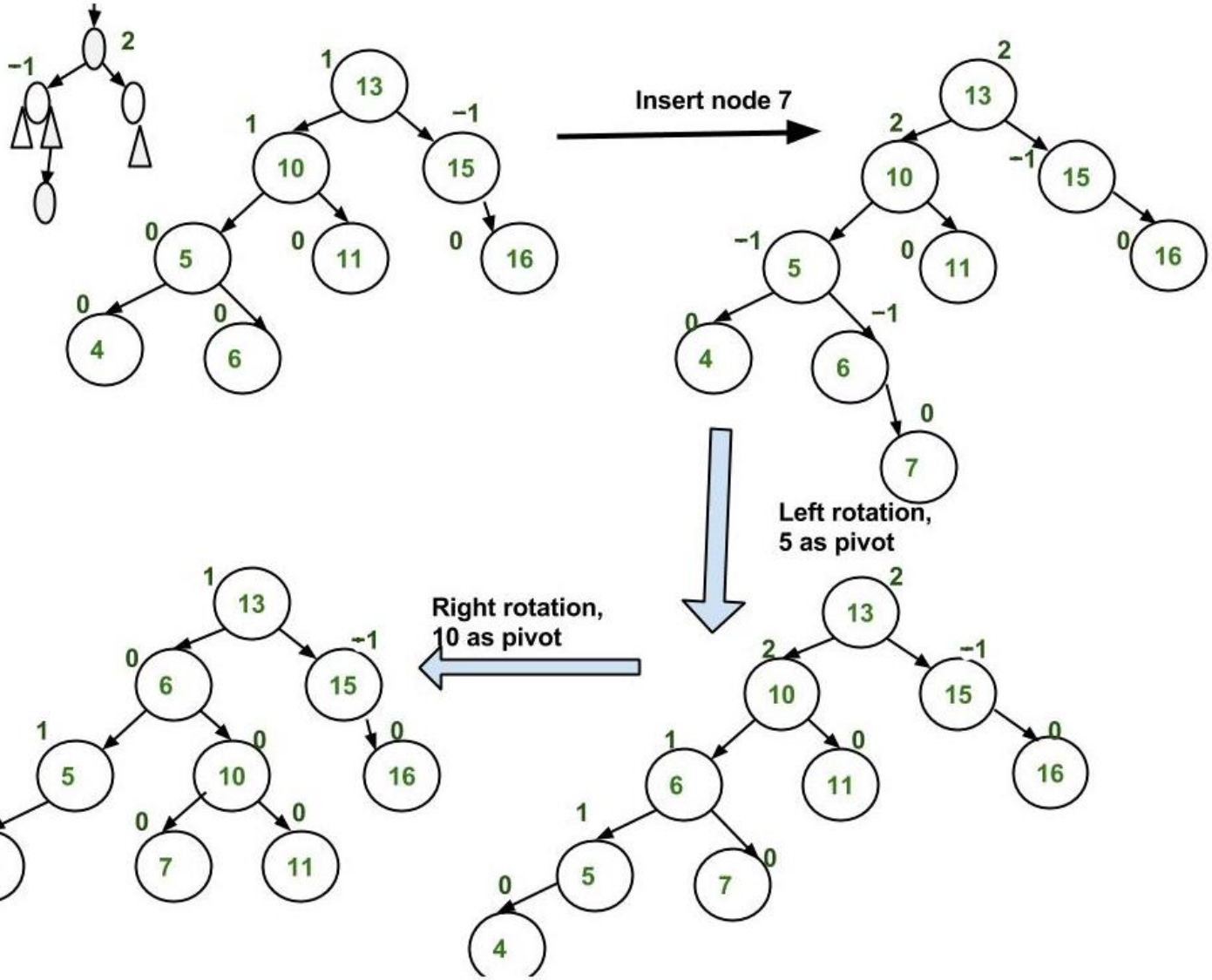
Example


+2
+1
Insert 45
-1
30
0
5
35
0
32
0
40
-2
30
5
-1
35
0
32
-1
40
0
45
Left rotate, node with value 30
Taken as pivot
0
35
0
30
-1
40
0
5
0
32
0
45

Insertion: case 3

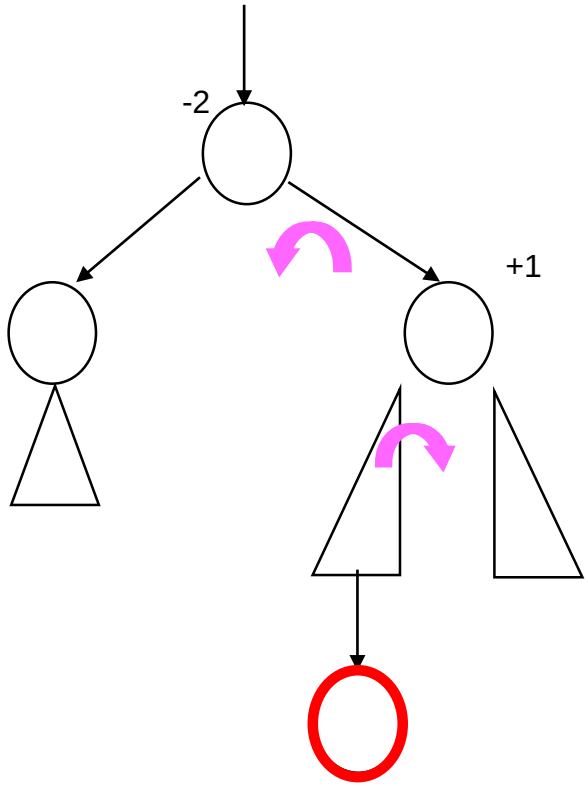
- Case 3a: The lowest node (with a balance factor of -2) had a taller left-subtree and the insertion was on the right-subtree of its left child.
- Requires a double left-right rotation to rebalance.



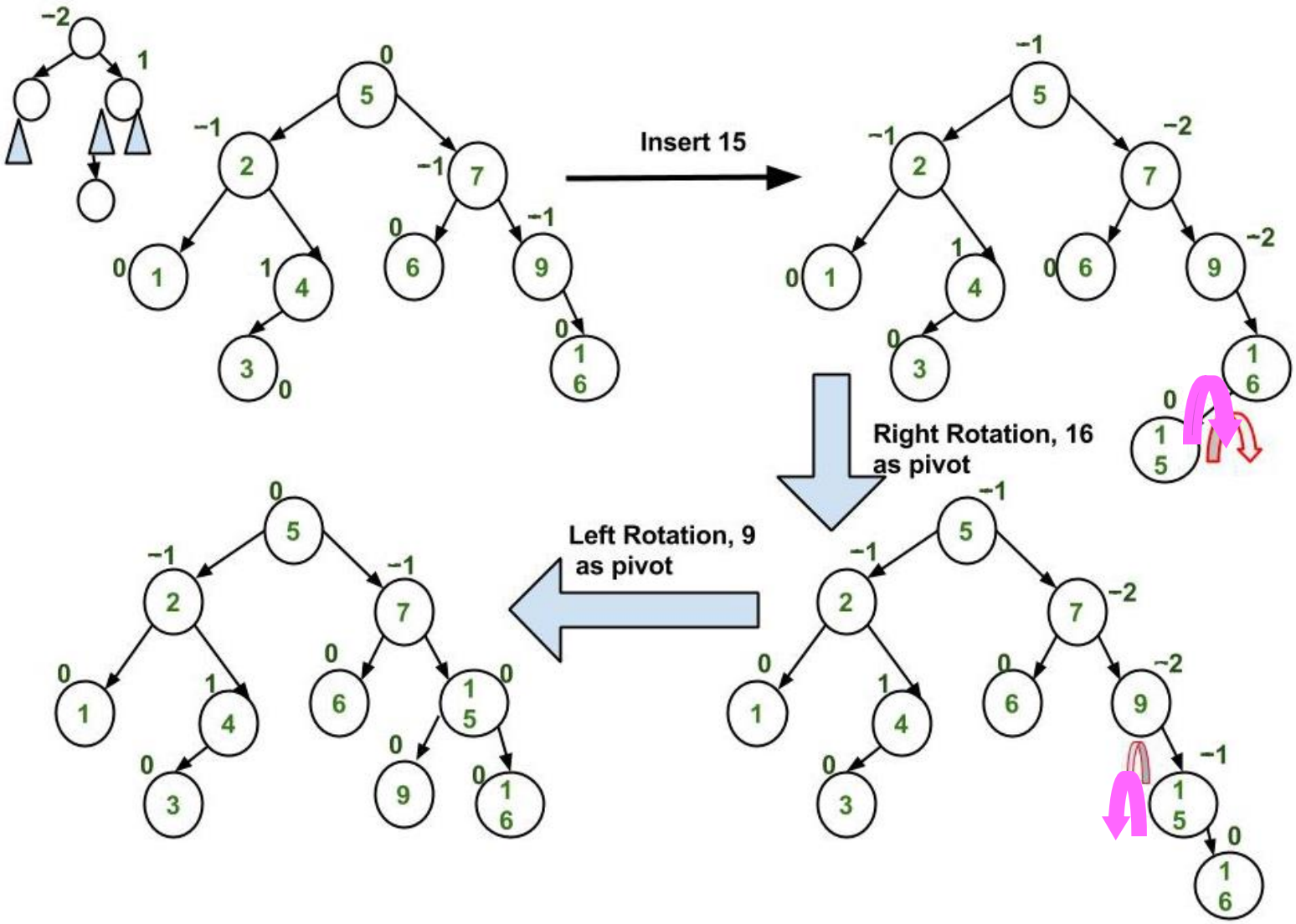

Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot

Insertion: case 3 (contd)

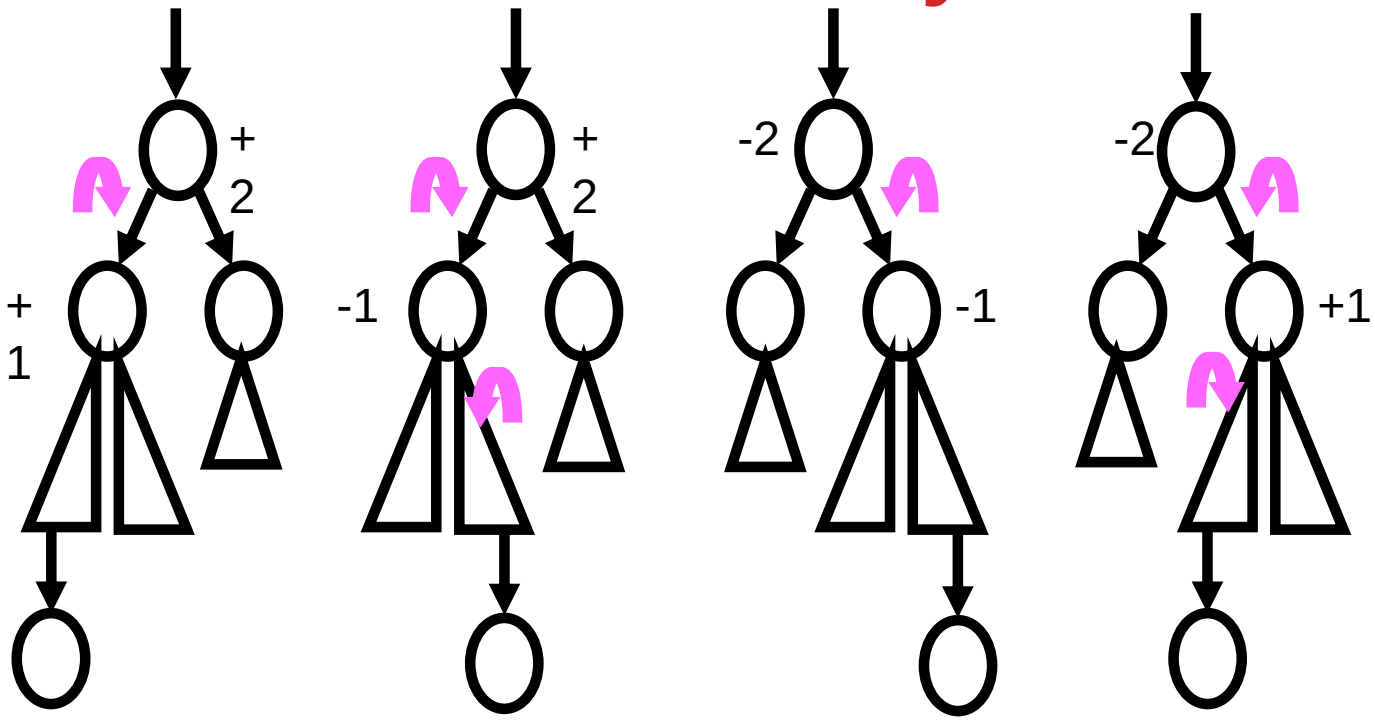
- Case 3b: The lowest node (with a balance factor of +2) had a taller right-subtree and the insertion was on the left-subtree of its right child.
- Requires a double right-left rotation to rebalance.



Example


Insert 15
Right Rotation, 16 as pivot
Left Rotation, 9 as pivot

AVL Rotation Summary



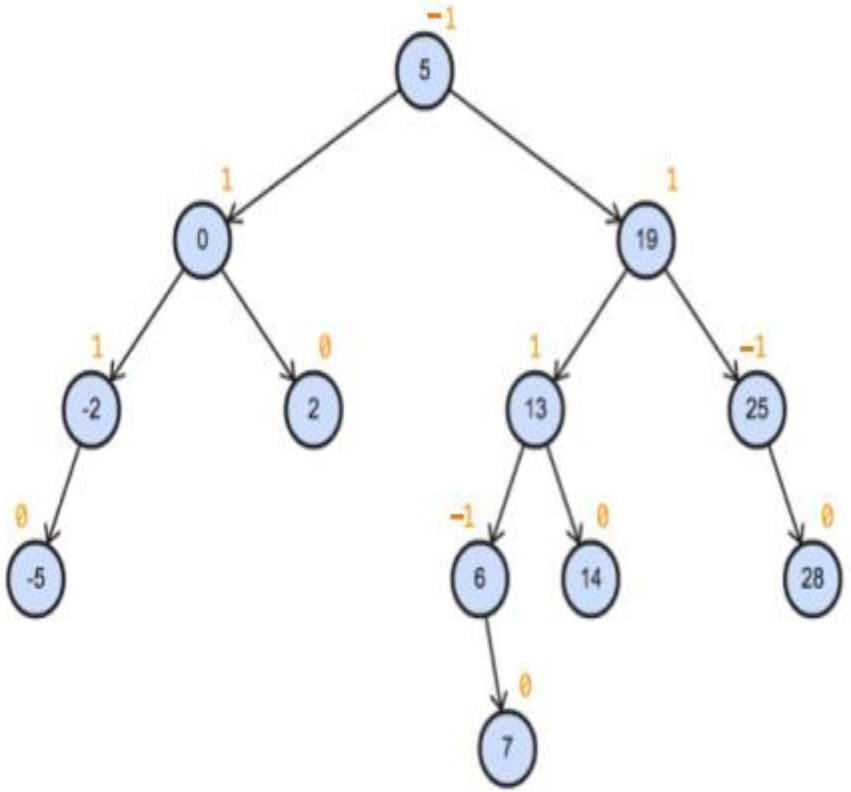
Single right rotation

Double left-right rotation

Single left rotation

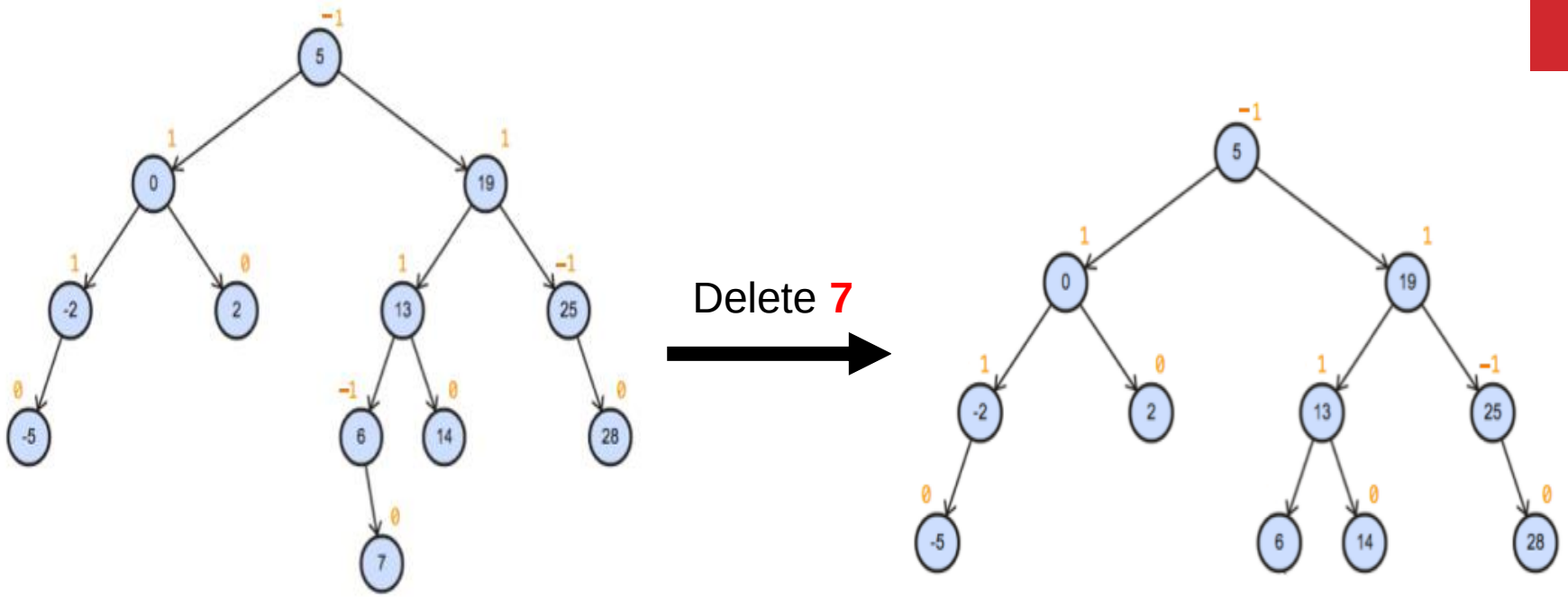
Double right-left rotation

Exercise: Insert into an initially empty AVL tree each of the following keys, in the order in which they appear in the sequence: **0, 25, 19, 5, -2, 28, 13, -5, 2, 6, 14, 7.**

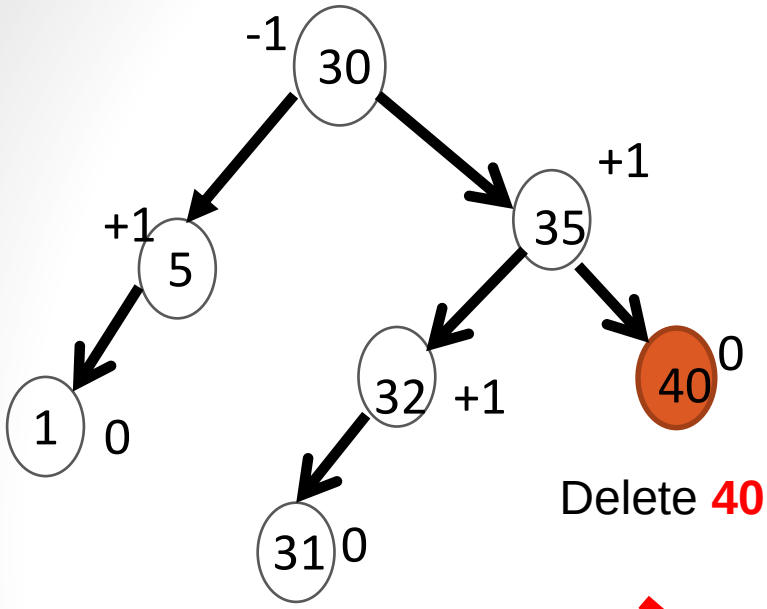


Deletion

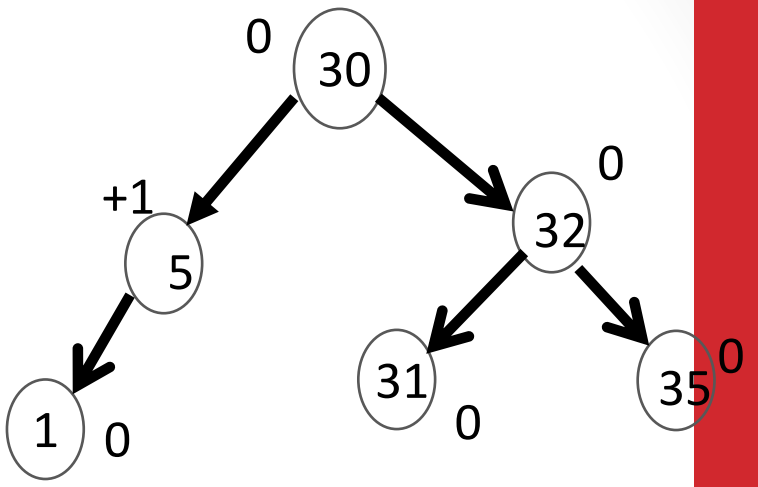
- Delete by a BST deletion by copying algorithm.
- Rebalance the tree if an imbalance occurs.
- There are three deletion cases:
  1. **Deletion that does not cause an imbalance.**
  2. **Deletion that requires a single rotation to rebalance.**
  3. **Deletion that requires two or more rotations to rebalance.**
- Deletion case 1 example:



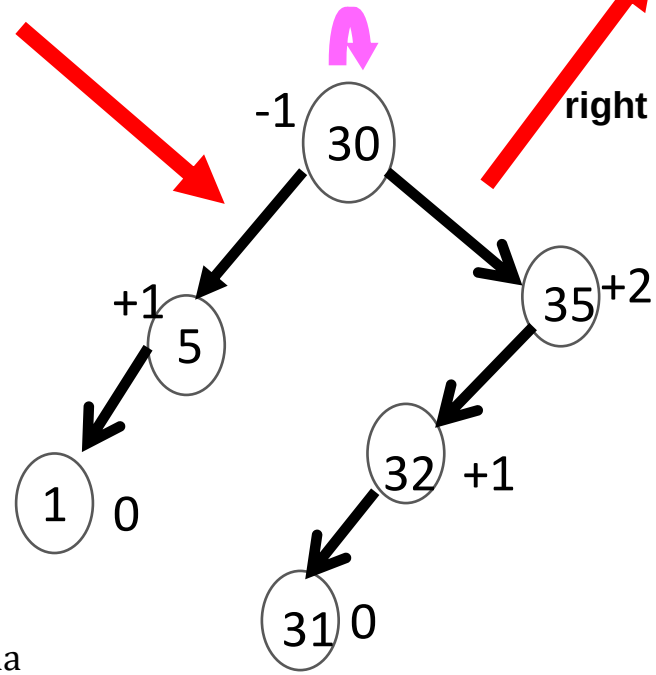
Deletion: case 2 examples



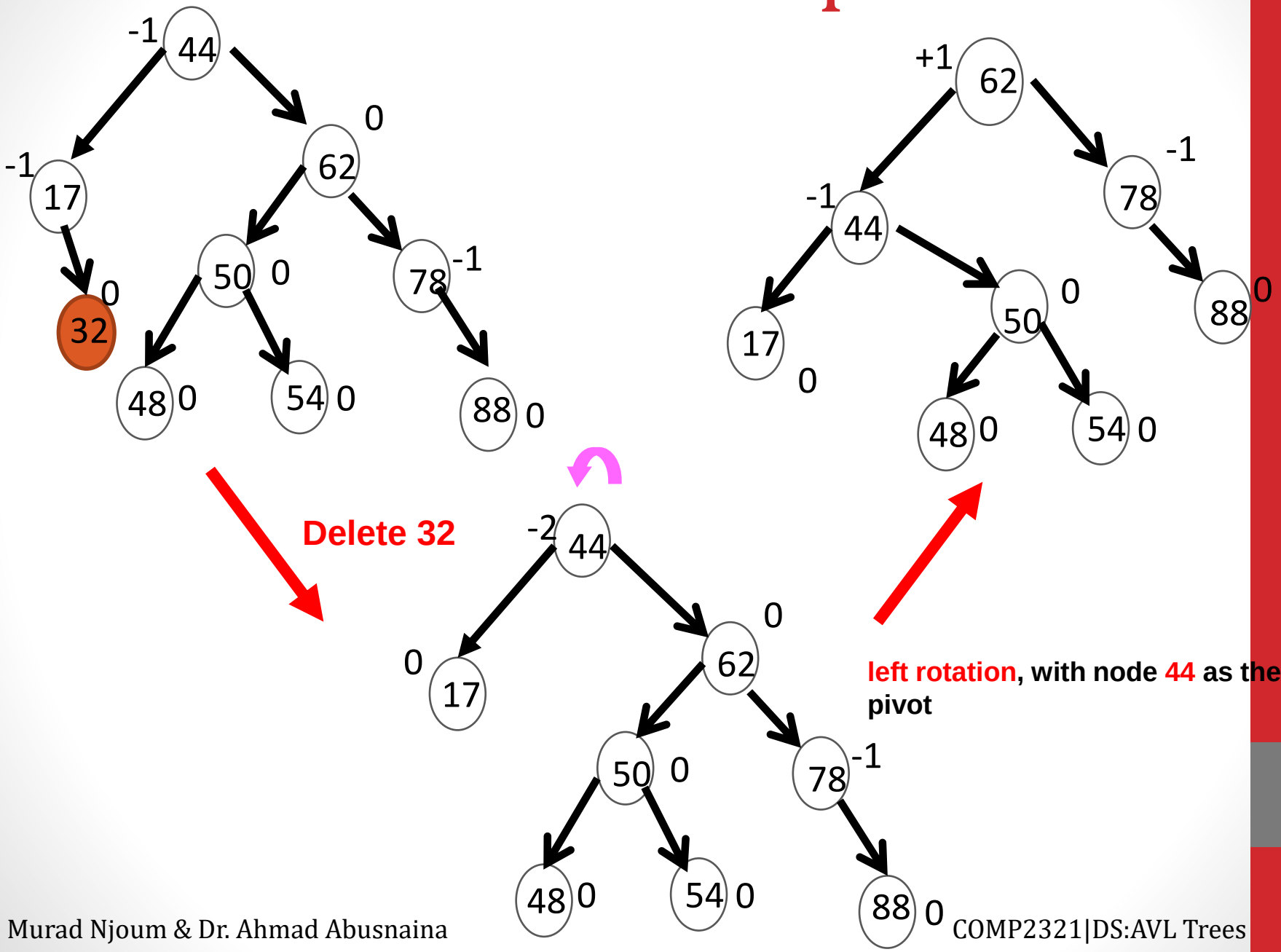
Delete **40**



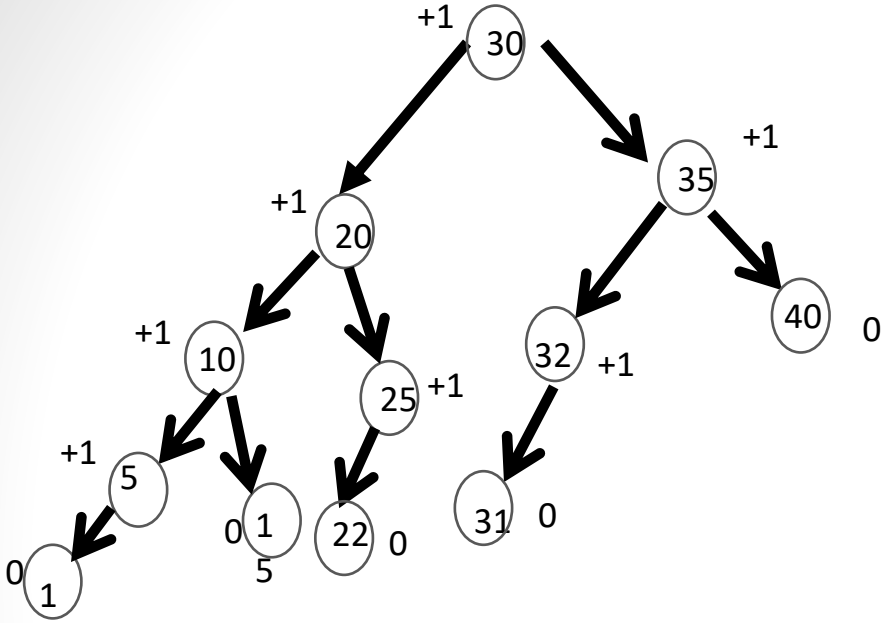
right rotation, with node 35 as the pivot



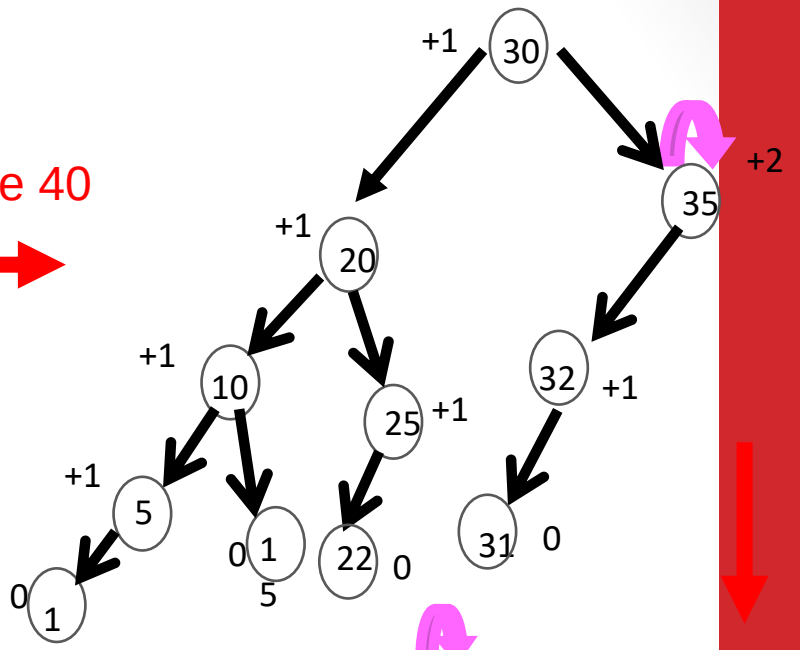
Deletion: case 2 examples



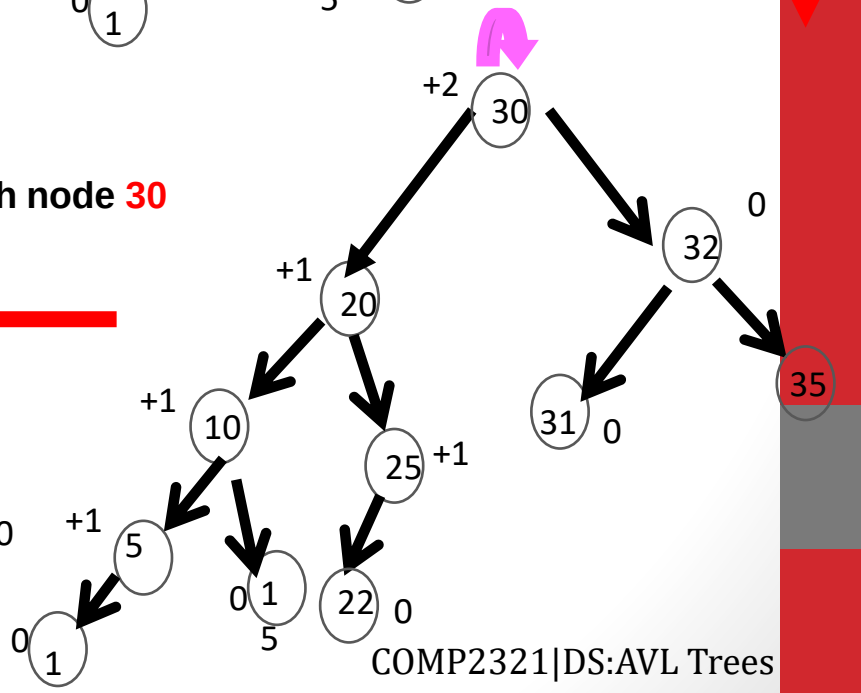
Deletion: case 3



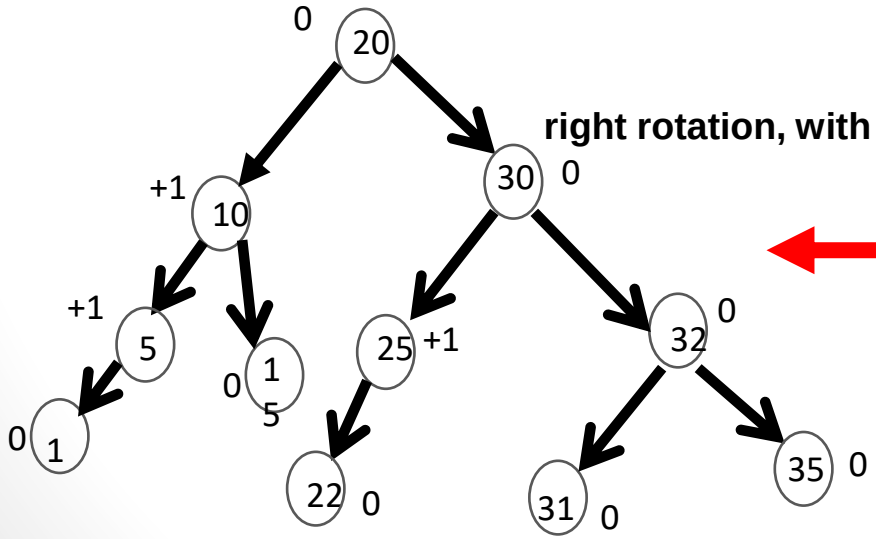
Delete 40



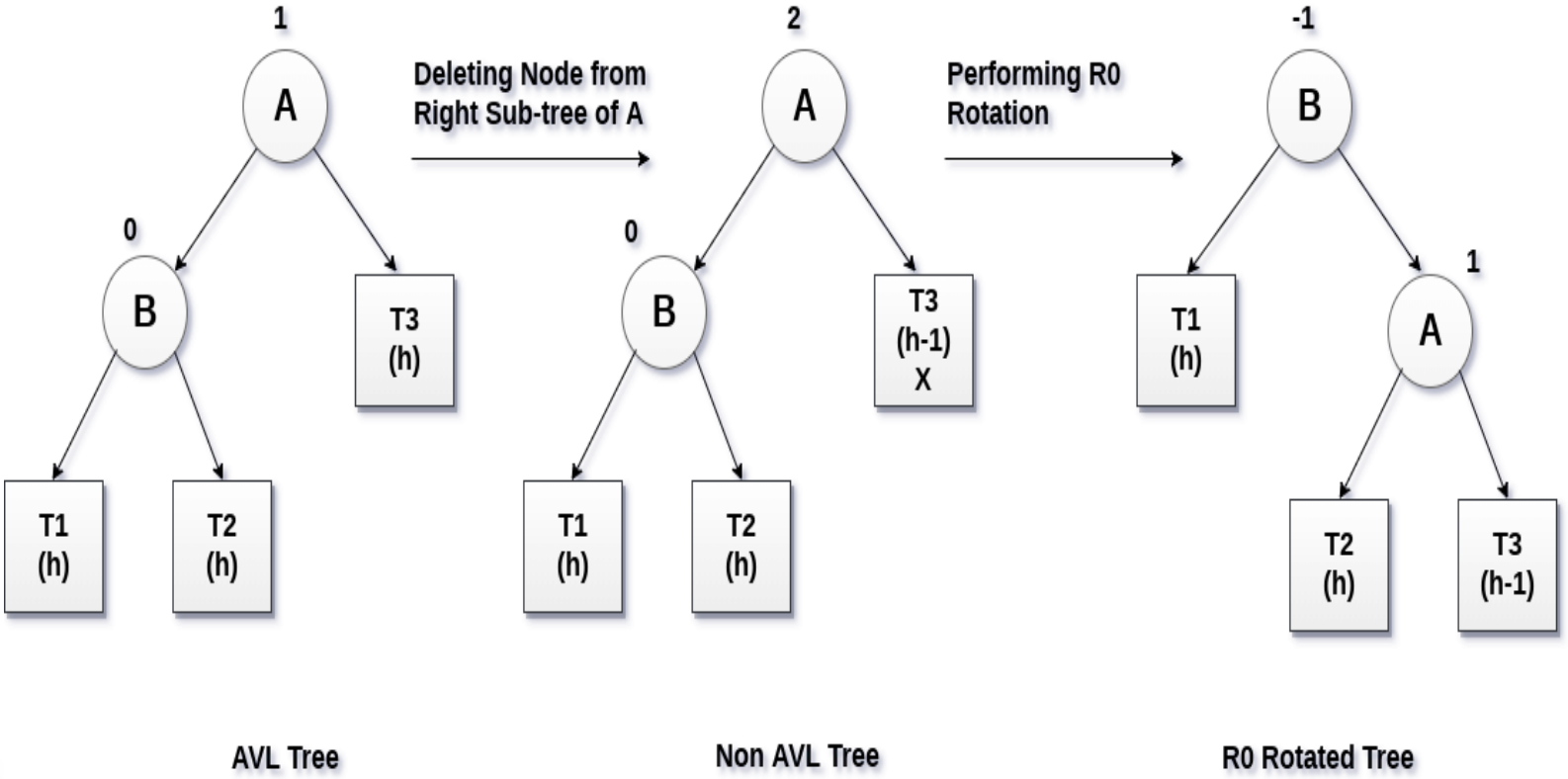
right rotation, with node 35



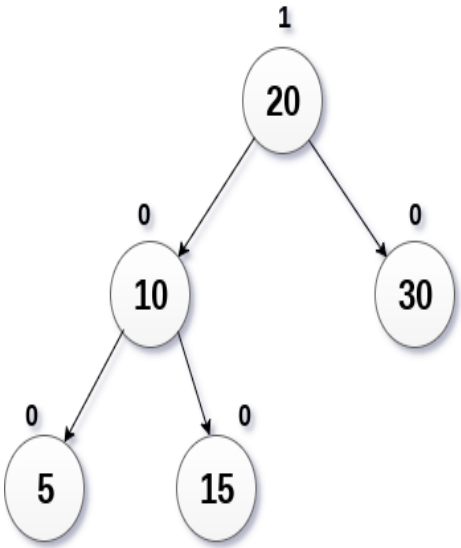
right rotation, with node 30

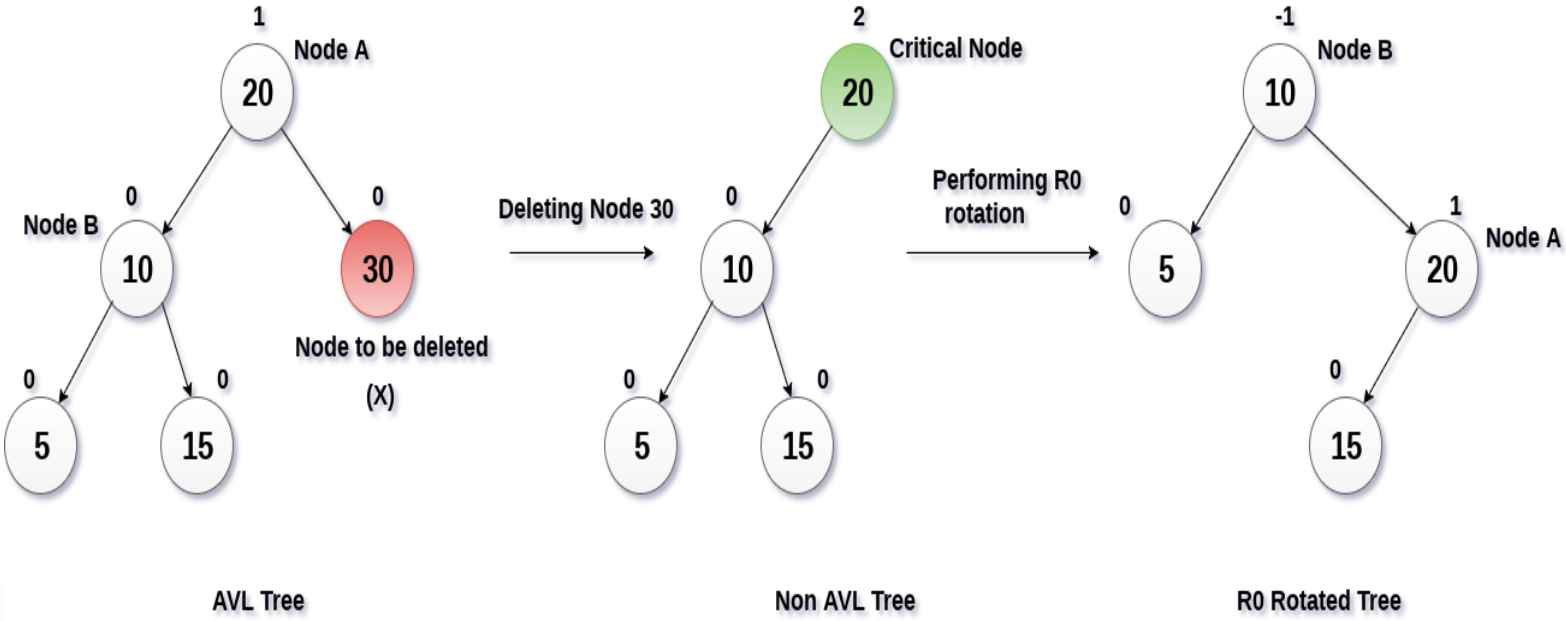


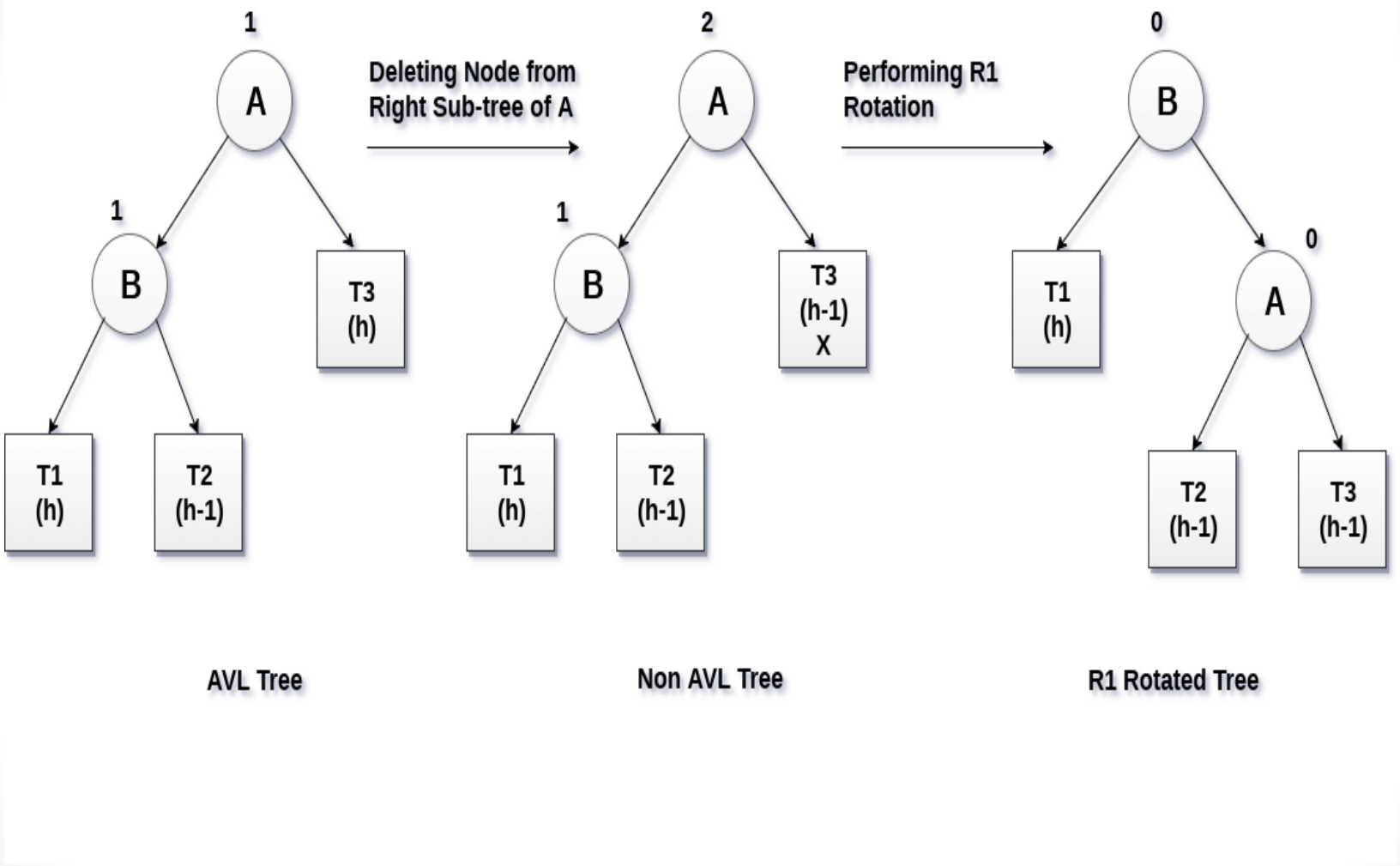
Deletion- In Depth- More Examples



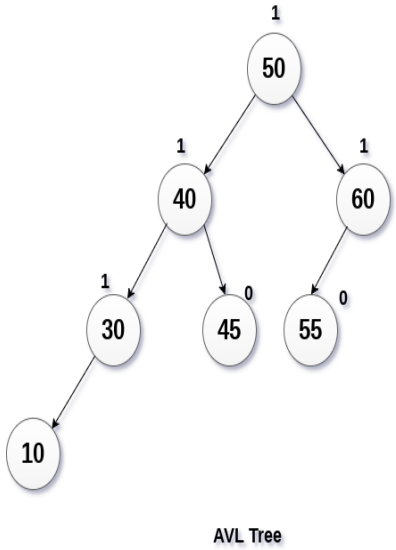
Example 1


1
20
0
10
0
30
0
5
0
15

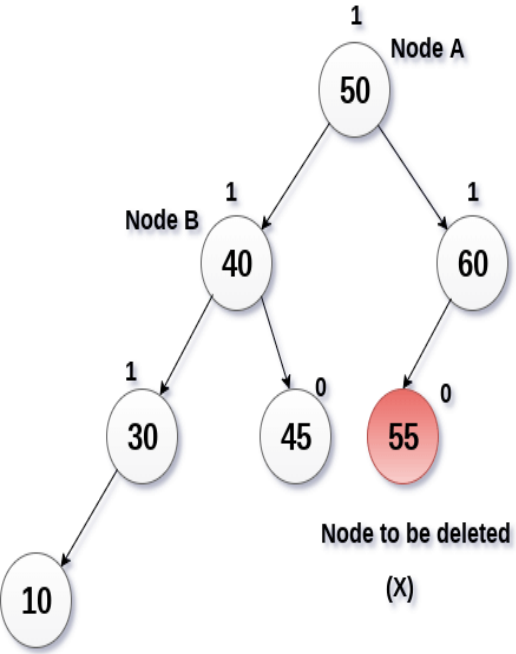

1
Node A
20
0
Node B
10
0
30
Node to be deleted
(X)
0
5
0
15
AVL Tree
Deleting Node 30
2
Critical Node
20
0
10
0
5
0
15
Non AVL Tree
Performing R0
rotation
-1
Node B
10
0
5
1
Node A
20
0
15
R0 Rotated Tree


1
A
1
B
T3
(h)
T1
(h)
T2
(h-1)
AVL Tree
Deleting Node from
Right Sub-tree of A
2
A
1
B
T3
(h-1)
X
T1
(h)
T2
(h-1)
Non AVL Tree
Performing R1
Rotation
0
B
T1
(h)
0
A
T2
(h-1)
T3
(h-1)
R1 Rotated Tree

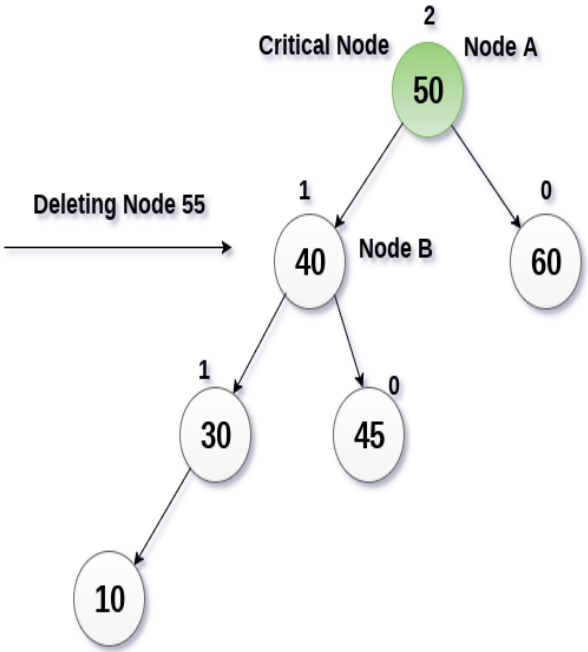
Example 2



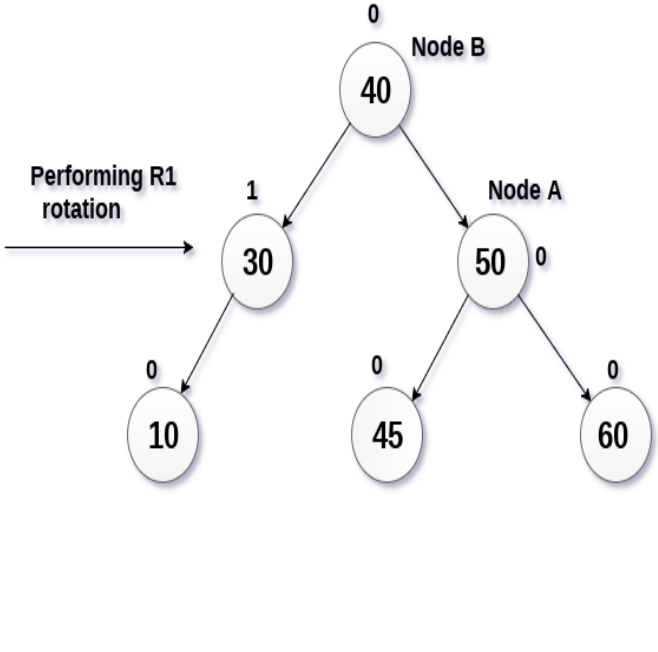
AVL Tree



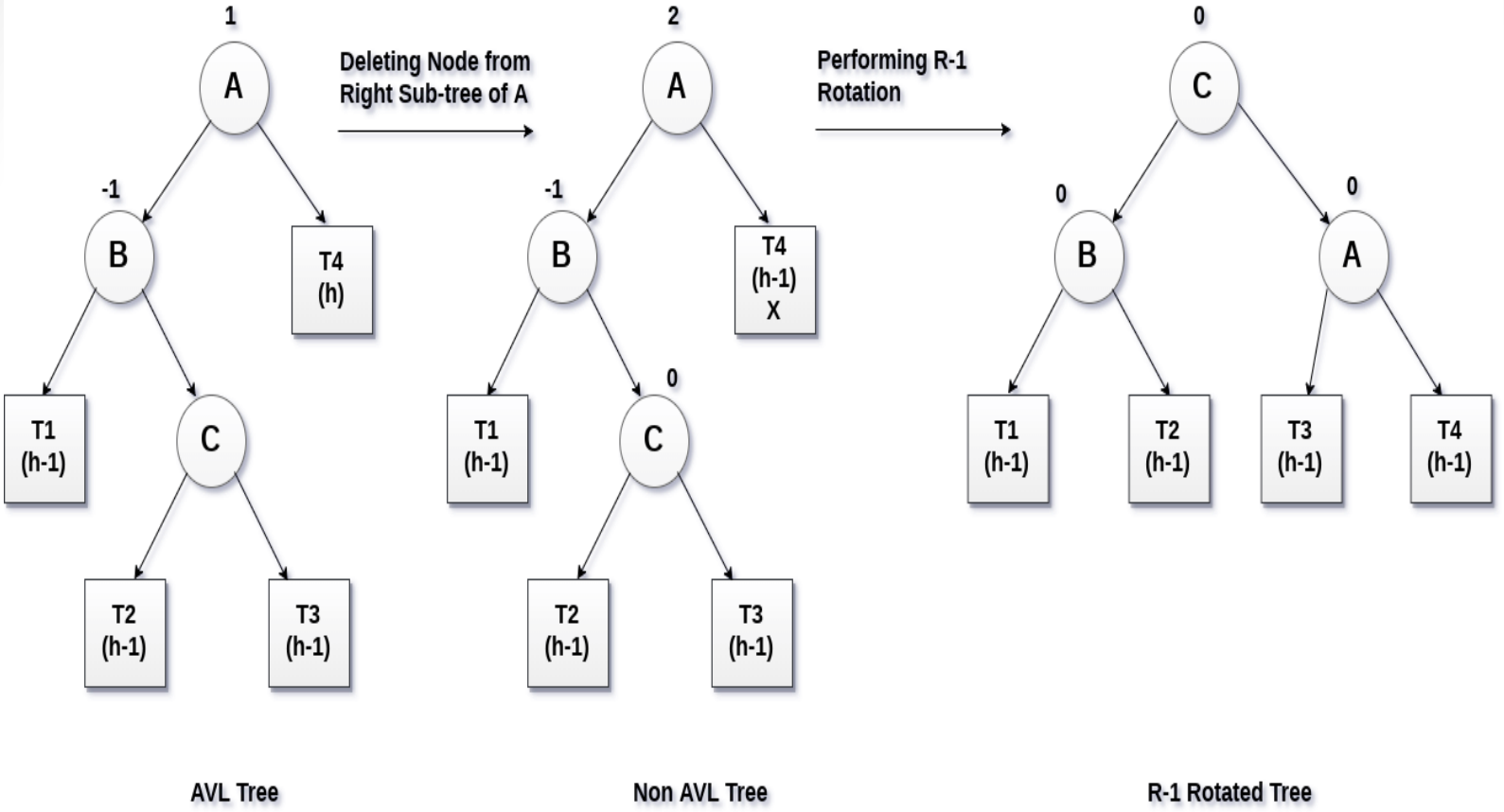
AVL Tree



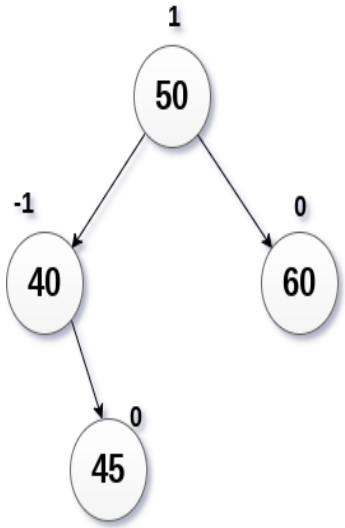
Non AVL Tree

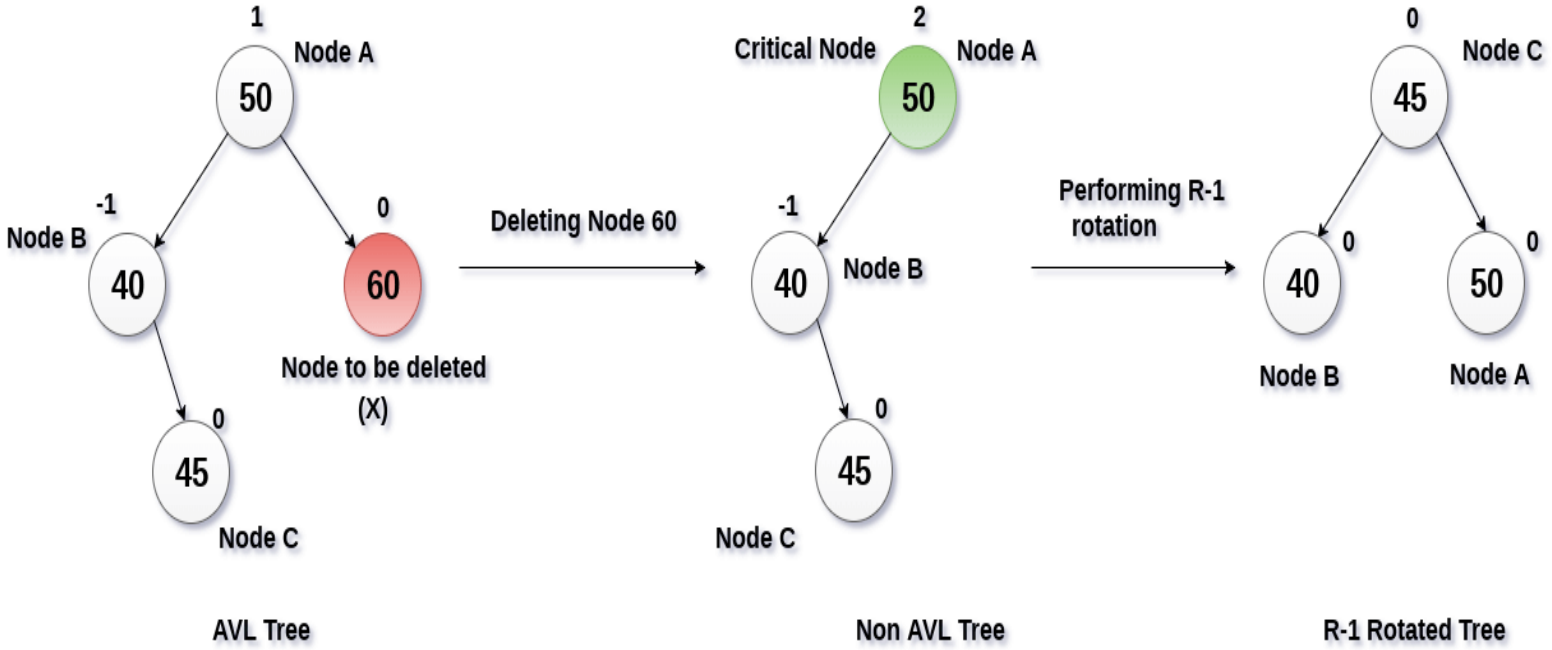


R1 Rotated Tree


1
A
-1
B
T4
(h)
T1
(h-1)
C
T2
(h-1)
T3
(h-1)
Deleting Node from
Right Sub-tree of A
2
A
-1
B
T4
(h-1)
X
T1
(h-1)
0
C
T2
(h-1)
T3
(h-1)
Performing R-1
Rotation
0
C
0
B
0
A
T1
(h-1)
T2
(h-1)
T3
(h-1)
T4
(h-1)
AVL Tree
Non AVL Tree
R-1 Rotated Tree

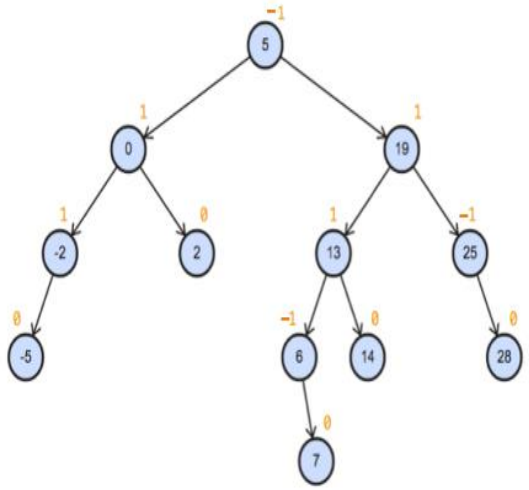
Example 3

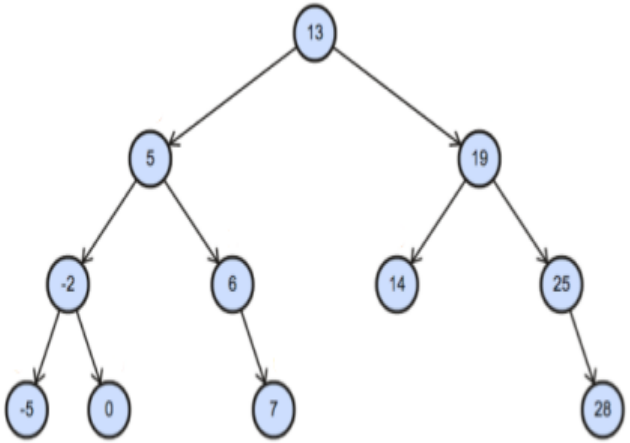

1
50
-1
0
40
60
0
45


1
Node A
50
-1
Node B
0
40
60
Node to be deleted
(X)
0
45
Node C
AVL Tree
Deleting Node 60
2
Critical Node
Node A
50
-1
40
Node B
0
45
Node C
Non AVL Tree
Performing R-1
rotation
0
Node C
45
0
40
0
50
Node B
Node A
R-1 Rotated Tree

Exercise (Previous Built AVL-Tree) :

A- Delete node 2





B- Delete root

C- Delete node 7, then 2 (Try it at home)

```
struct Node
{   int key;
    struct Node *left;
    struct Node *right;
    int height;
};
int max(int a, int b);
int height(struct Node *N)
{    if (N == NULL)
        return 0;
    return N->height;}
 int max(int a, int b)
{
    return (a > b)? a : b;
}
```

```
struct Node * minValueNode(struct Node*
node)
{
    struct Node* current = node;

    while (current->left != NULL)
        current = current->left;

    return current;
}
```

```
int getBalanceFactor(struct Node
*N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N-
>right);
}
```

```
struct Node* newNode(int key)
{
    struct Node* node = (struct
Node*)
            malloc(sizeof(struct
Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 0;
    return(node);
}
```

```
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y-
>right))+1;
    x->height = max(height(x->left), height(x-
>right))+1;

    return x;
}
```

```
struct Node *leftRotate(struct Node
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), hei
    y->height = max(height(y->left), hei

    return y;
}
```

```
struct Node* insertNode(struct
Node* node, int key)
{
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insertNode(node-
>left, key);
    else if (key > node->key)
        node->right =
insertNode(node->right, key);
    else
        return node;

    node->height = 1 +
max(height(node->left),
                    height(node-
>right));

    int balance =
getBalanceFactor(node);
```

```
// Right Right Case
    if (balance < -1 && key > node->
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->l
    {
        node->left = leftRotate(node->
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->
    {
        node->right = rightRotate(nod
        return leftRotate(node);
    }
    return node;
}
```

```
truct Node* deleteNode(struct Node*
root, int key)
{
    if (root == NULL)
        return root;

    if ( key < root->key )
        root->left = deleteNode(root->left,
key);
    else if( key > root->key )
        root->right = deleteNode(root-
>right, key);
    else
    {
        if( (root->left == NULL) || (root-
>right == NULL) )
        {
            struct Node *temp = root->left ?
root->left :
                                root->right;
            if (temp == NULL)
            {
                temp = root;
                root = NULL;   }
            else
                *root = *temp;
```

```
else
        {    struct Node* temp = minValueNode(roo
            root->key = temp->key;
            root->right = deleteNode(root->right, temp
        }
    }

    if (root == NULL)
        return root;
// STEP 2: UPDATE HEIGHT OF THE CURRENT
    root->height = max(height(root->left), height(ro

    // STEP 3: GET THE BALANCE FACTOR OF T
    // this node became unbalanced)
    int balance = getBalanceFactor(root);

    // If this node becomes unbalanced, then there
    // Left Left Case
```

```
if (balance > 1 && getBalanceFactor(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalanceFactor(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalanceFactor(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalanceFactor(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
     return root;
}
```

Exercise

- Rewrite the above codes for delete nodes from tree.

- Insert the following Number in AVL tree

  {20,50,30,15,3,45,17,25,12,11,7,19,14,2}

  Then Delete Number {45,20,15,25}

  Show your works after each step (Check Balance)

THANK YOU