ALGORITHMS & COMPLEXITY

AVL TREE

Feb 16, 2022

Presented by:
Achyut Thapa (52)
Sabin Thapa (54)
Shreyam Pokharel (40)

TOPICS

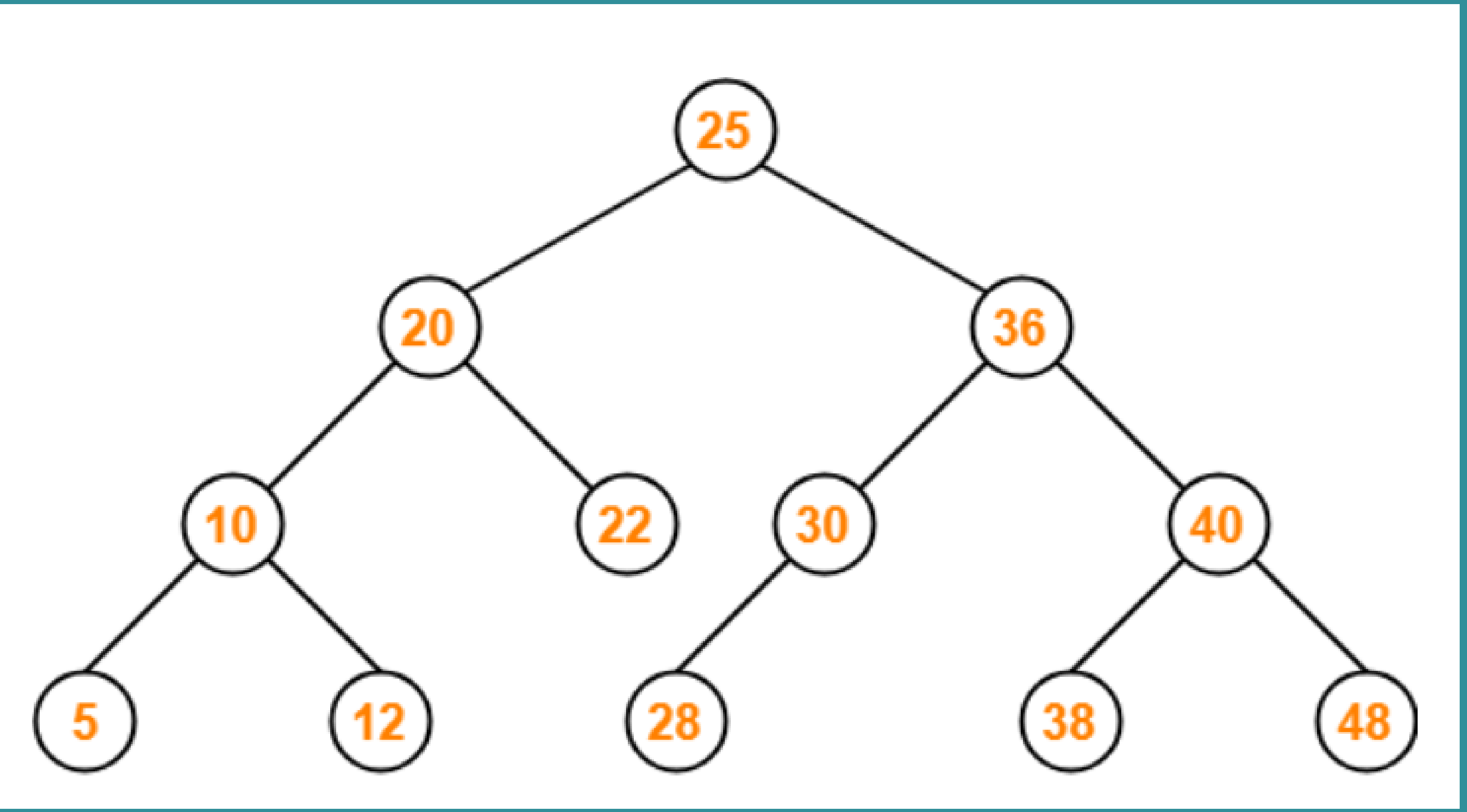
- Definitions
- Balance factor
- Rotations
- Insertion, deletion

BASIC CONCEPTS

- AVL Tree is the type of Binary Tree
  - Any node has at most 2 children
  - A binary tree may have zero children

BASIC CONCEPTS

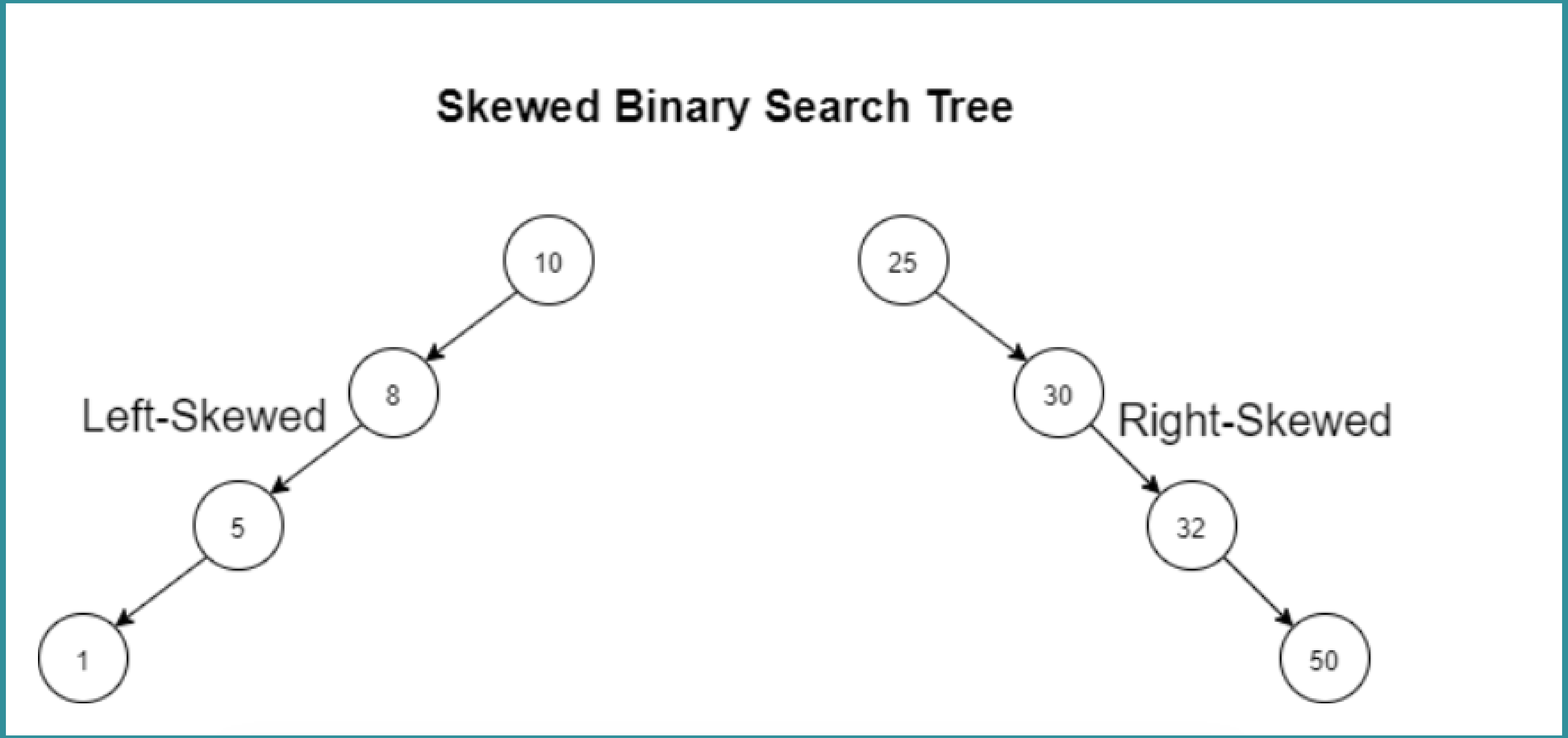
- AVL Tree is a variant of Binary Search Tree
  - All keys in left subtree are smaller than the key in the root.
  - All keys in right subtree are larger than the key in the root.
  - The left and right subtree are also Binary search tree.



PROBLEMS ON BST

- Height is not under control i.e. Its depends on insertion of elements
- The another problem is for a skewed binary search tree is that the worst case time complexity of search is O(n).

Examples related problems of BST



In order to search 1 in left skewed tree, time complexity will be O(n)

NEED TO RESOLVE THE PROBLEM

- There is a need to maintain the binary search tree to be of the balanced height, so that it is possible to obtained for the search option a time complexity of O(log N) in the worst case.
- So, the most popular balanced tree was introduced by:

1. Adelson
2. Velskii
3. Landis

DEFINITION OF AVL TREE

- AVL Tree is a self balancing binary search tree.
- AVL Tree is a Binary Tree in which the difference of height of right and left subtree of any nodes is less than or equal to 1.

BALANCE FACTOR

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

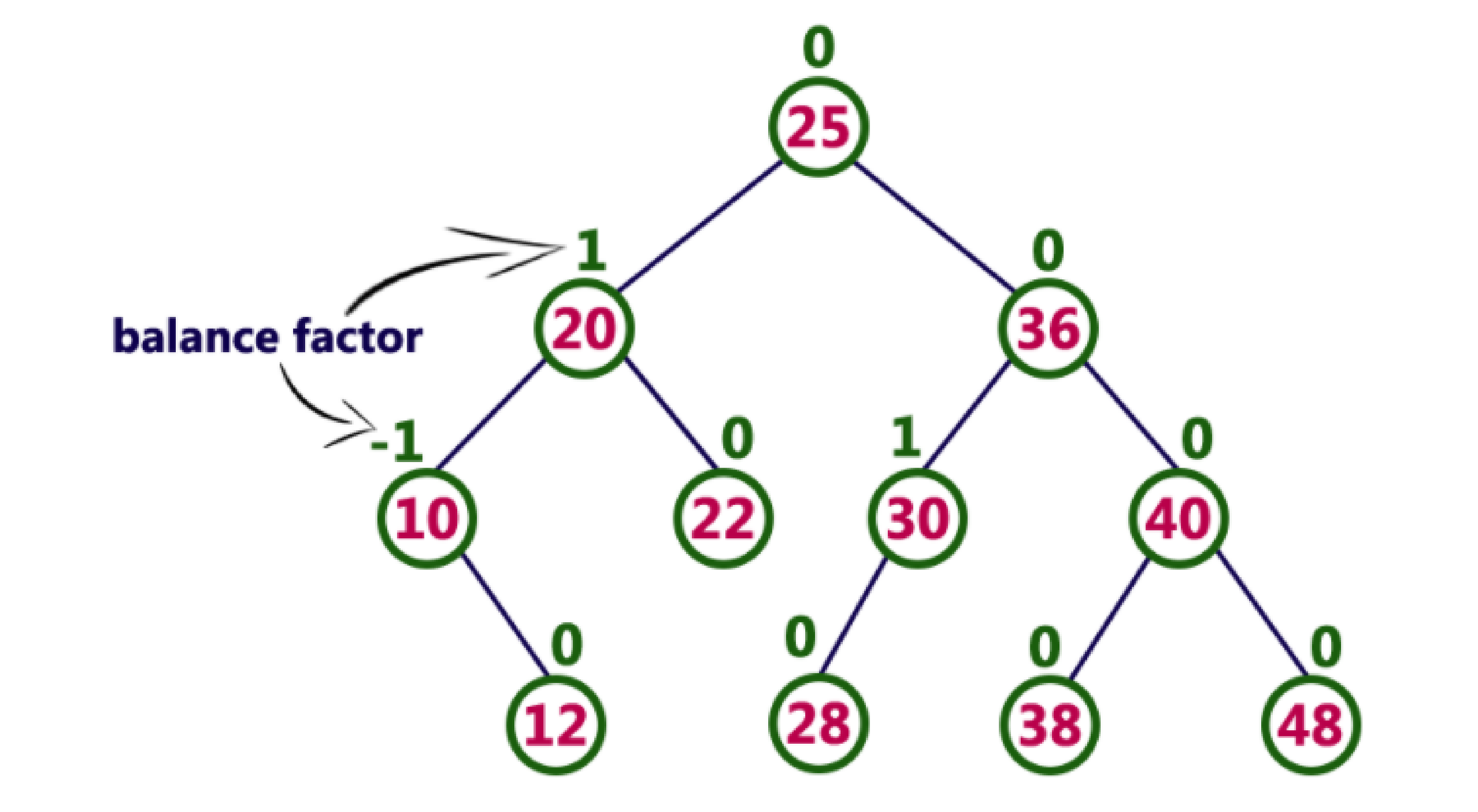
Balance Factor = Height of left subtree - Height of right subtree

BALANCE FACTOR

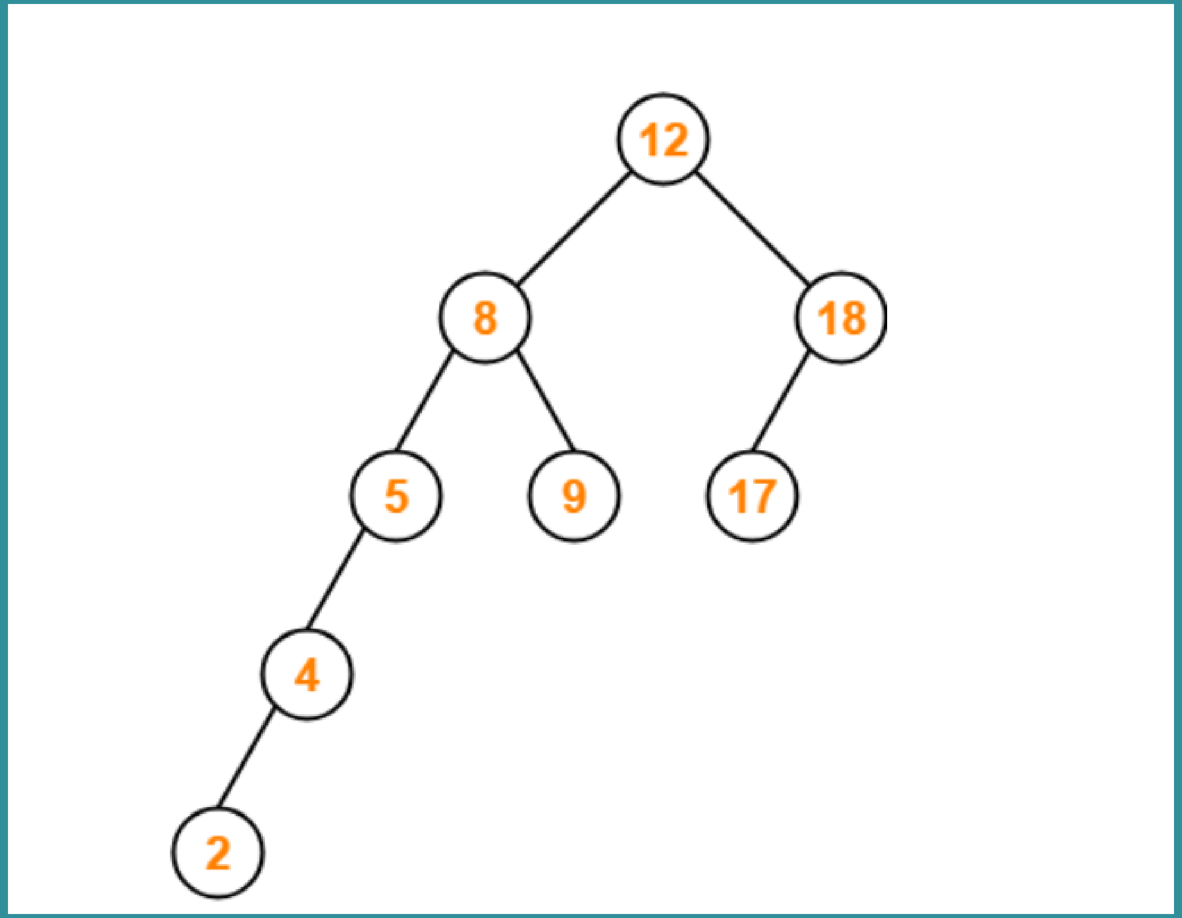
- The range of balanced factor is = {-1,0,1}
- If a Binary Search Tree obeys the balancing factor than it is called AVL Tree.
- In order to be AVL Tree, every node in a BST satisfy the balancing factor

- If Balancing factor = -1 -> Heavy Right AVL Tree
- If Balancing factor = 0 -> Balanced AVL Tree
- If Balancing factor = 1 -> Heavy Left AVL Tree

EXAMPLES



Example of AVL Tree

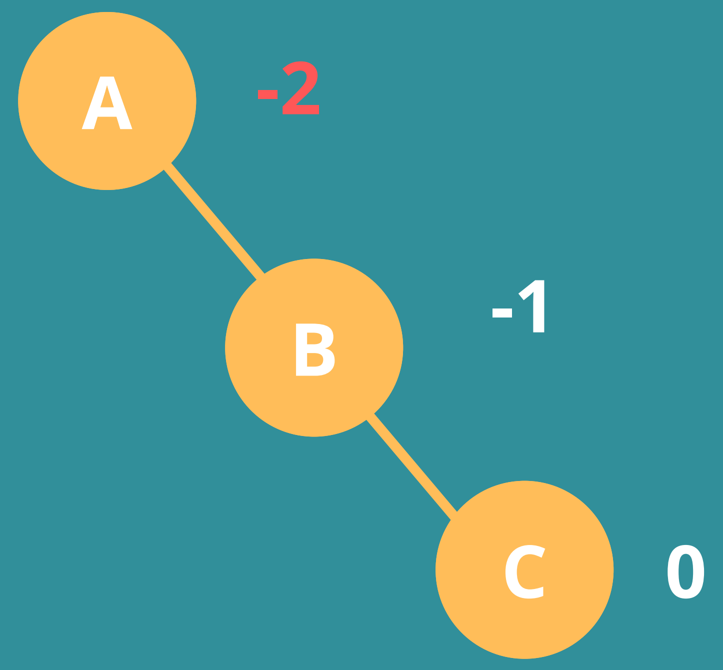


It is not AVL Tree

ROTATIONS

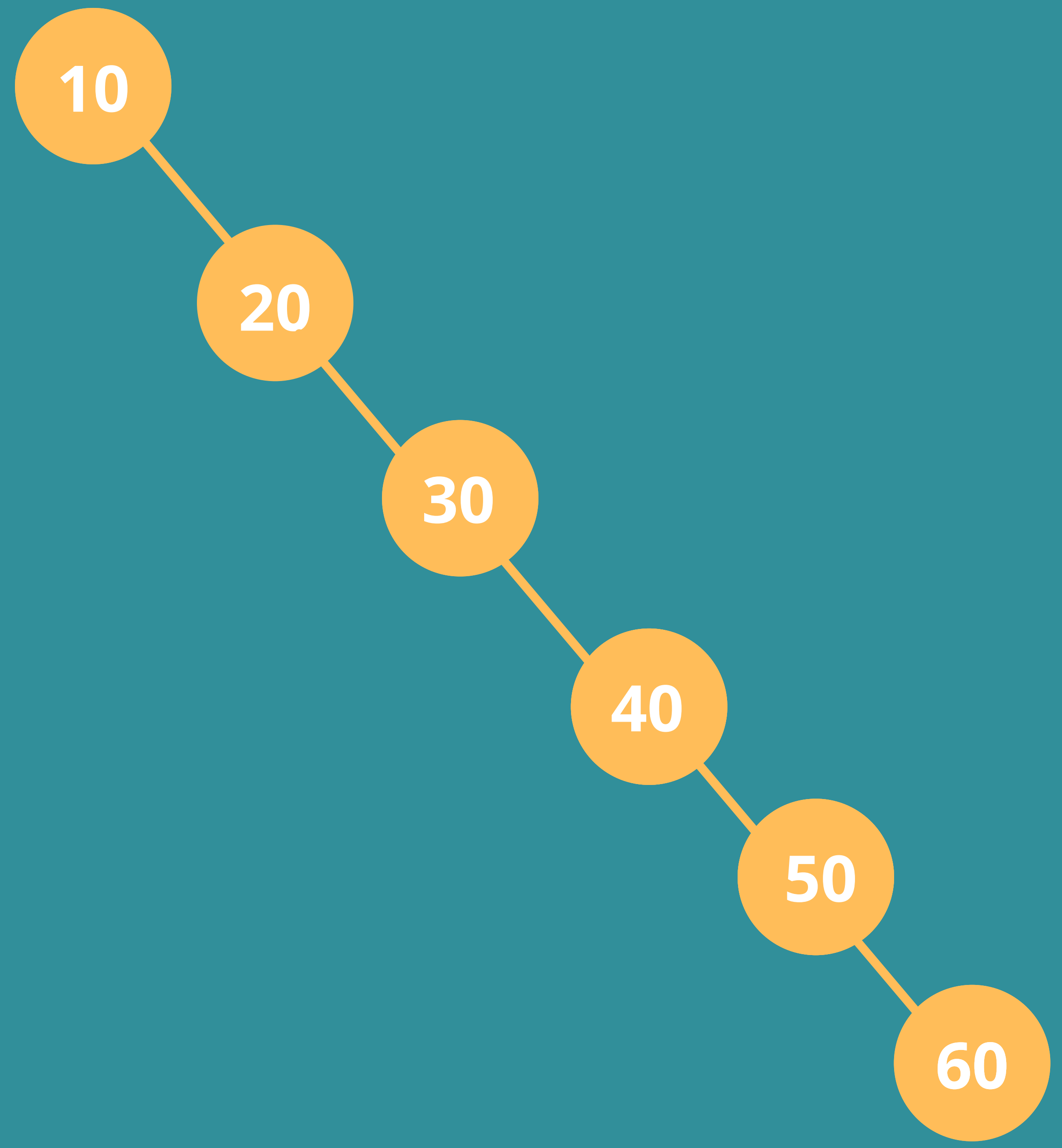
For a self balancing tree, rotations need to be performed in order for it to balance itself.

Balance Factor not in [-1,0,1] ? Perform rotation : Do nothing

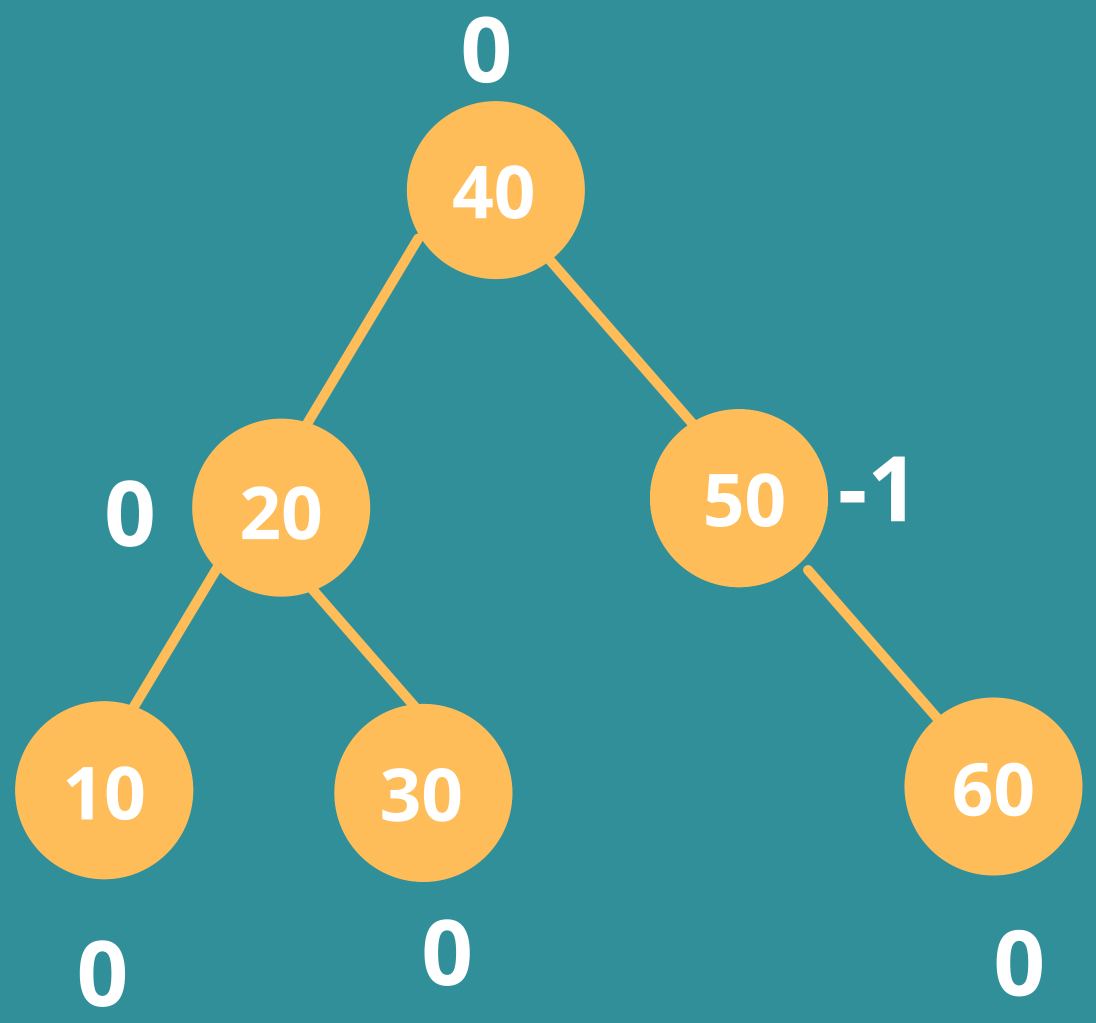


Unbalanced BST

ROTATIONS: WHY?



Search takes O(n) [WC]



balanced BST

Search takes O(lgn) [WC]

ROTATIONS - TYPES

1. **Left Rotation**
2. **Right Rotation**

Single Rotations

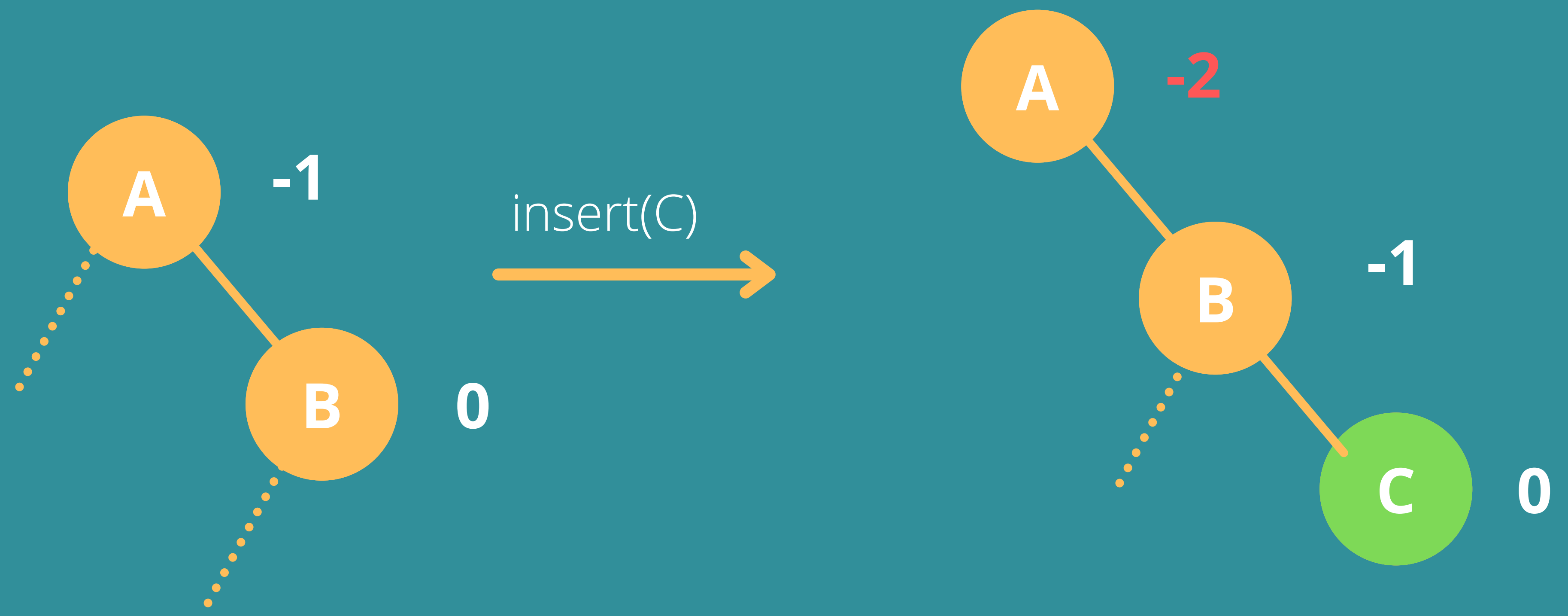
3. **Left Right (LR) Rotation**
4. **Right Left (RL) Rotation**

Double Rotations

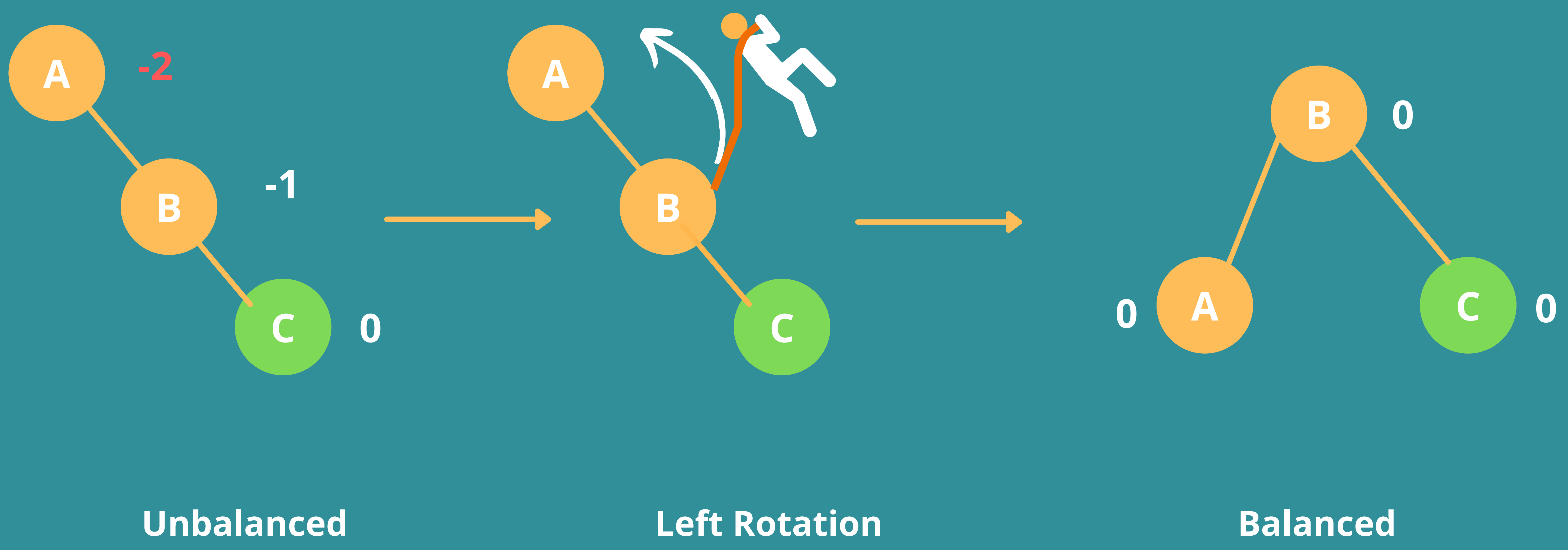
LEFT ROTATION

Performed when a tree becomes **right imbalanced.**

A tree is right imbalanced when a node is inserted into the right child of the right subtree of the node.

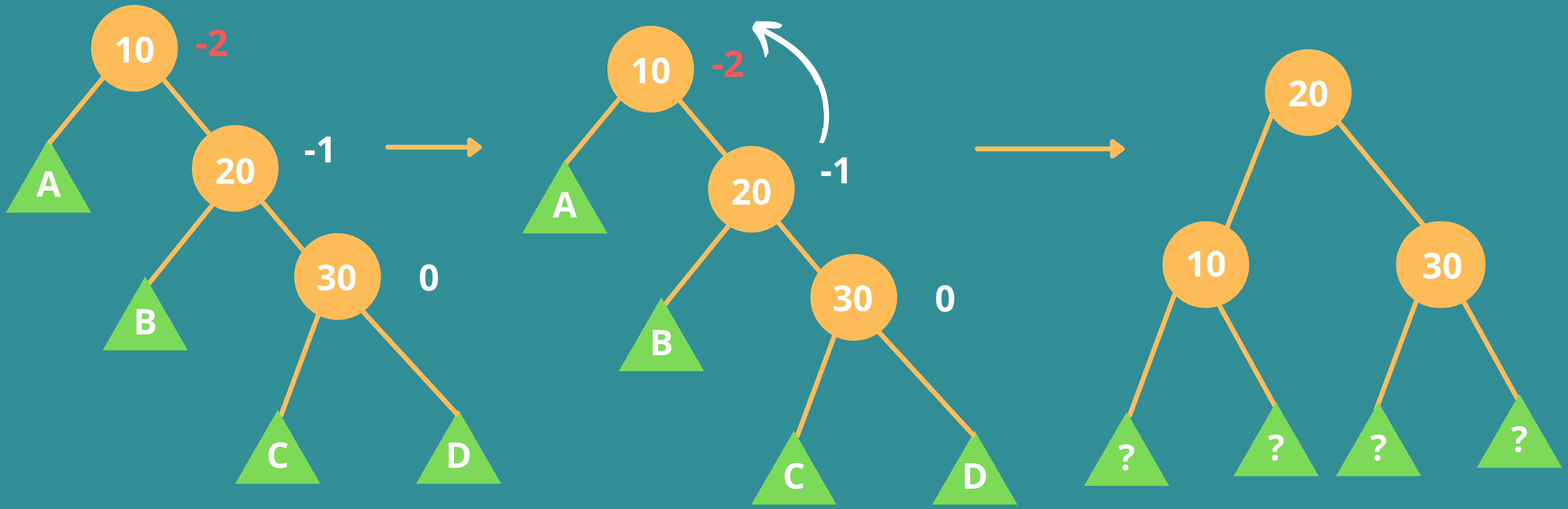


LEFT ROTATION



The tree is balanced! But are the properties of BST satisfied?

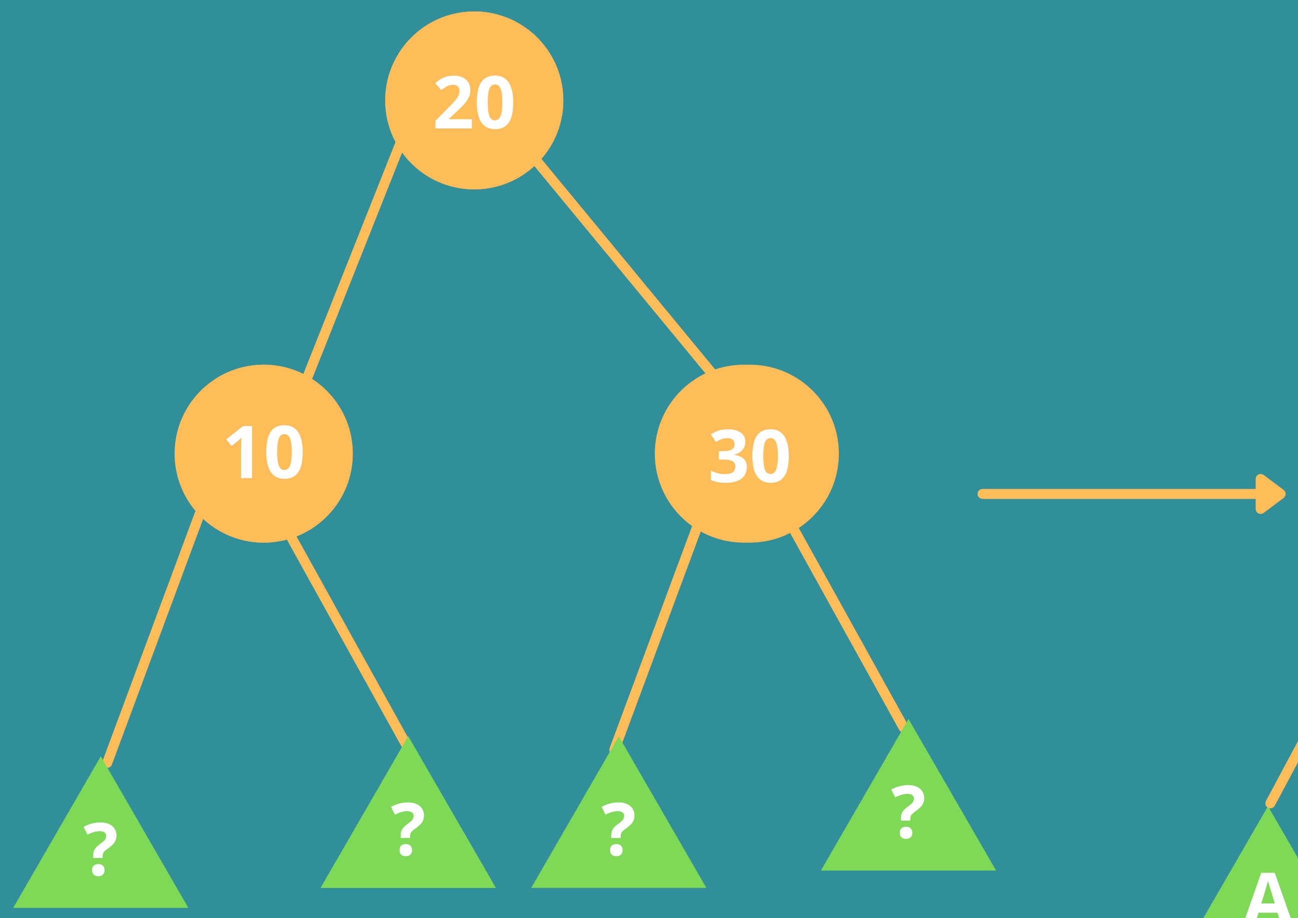
LEFT ROTATION

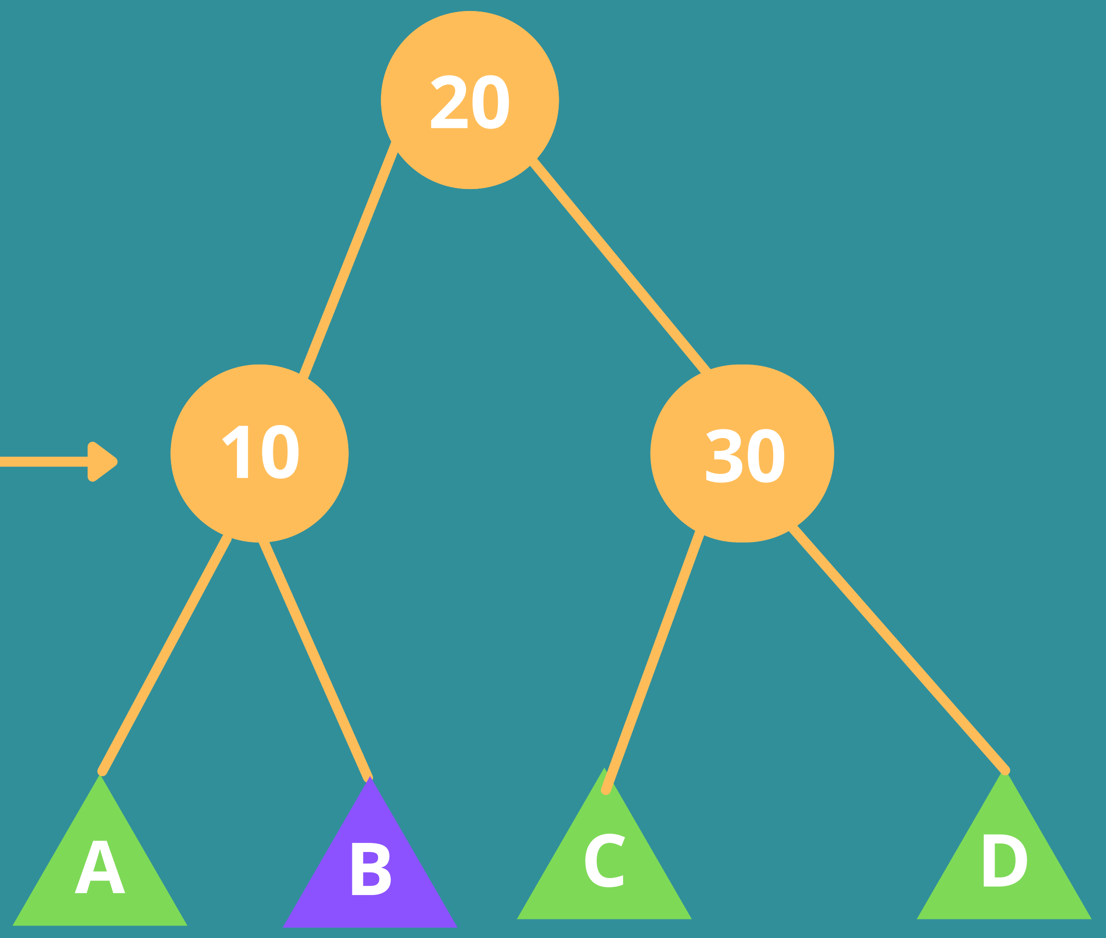


Where do the children go?

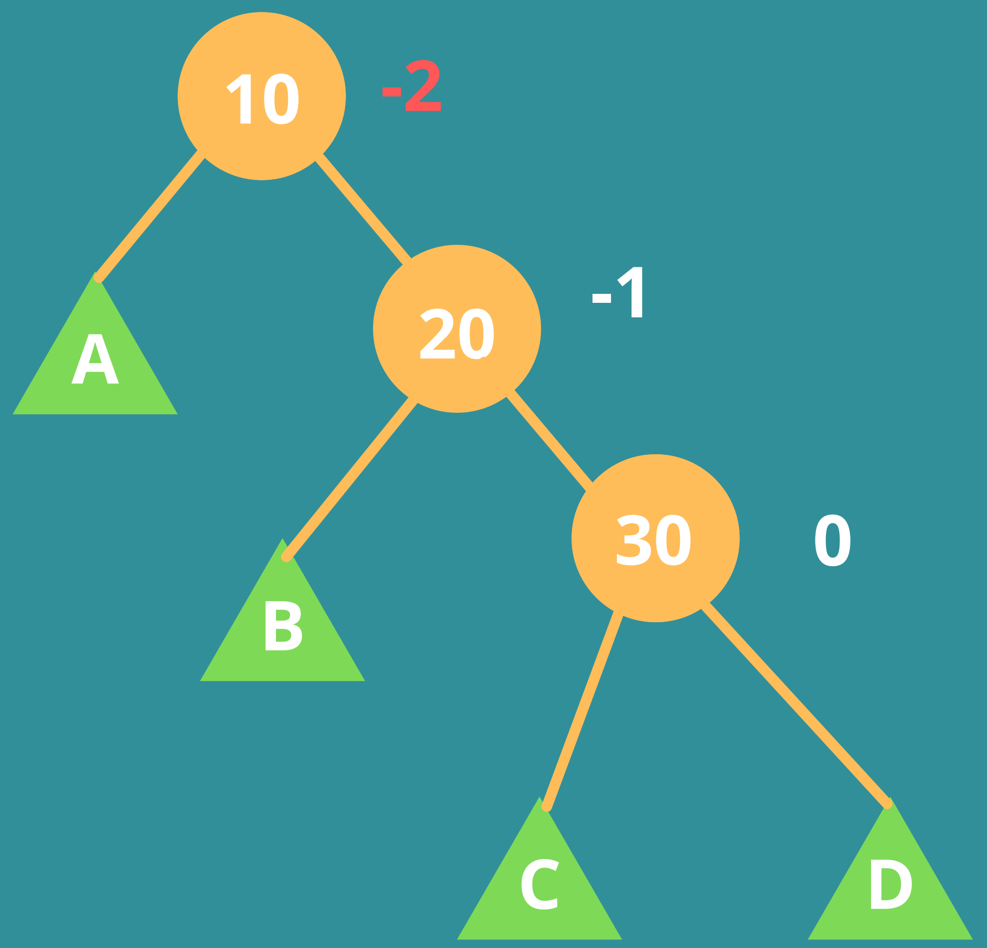
Ref: https://www.guru99.com/avl-tree.html

LEFT ROTATION





Final tree after Left Rotation



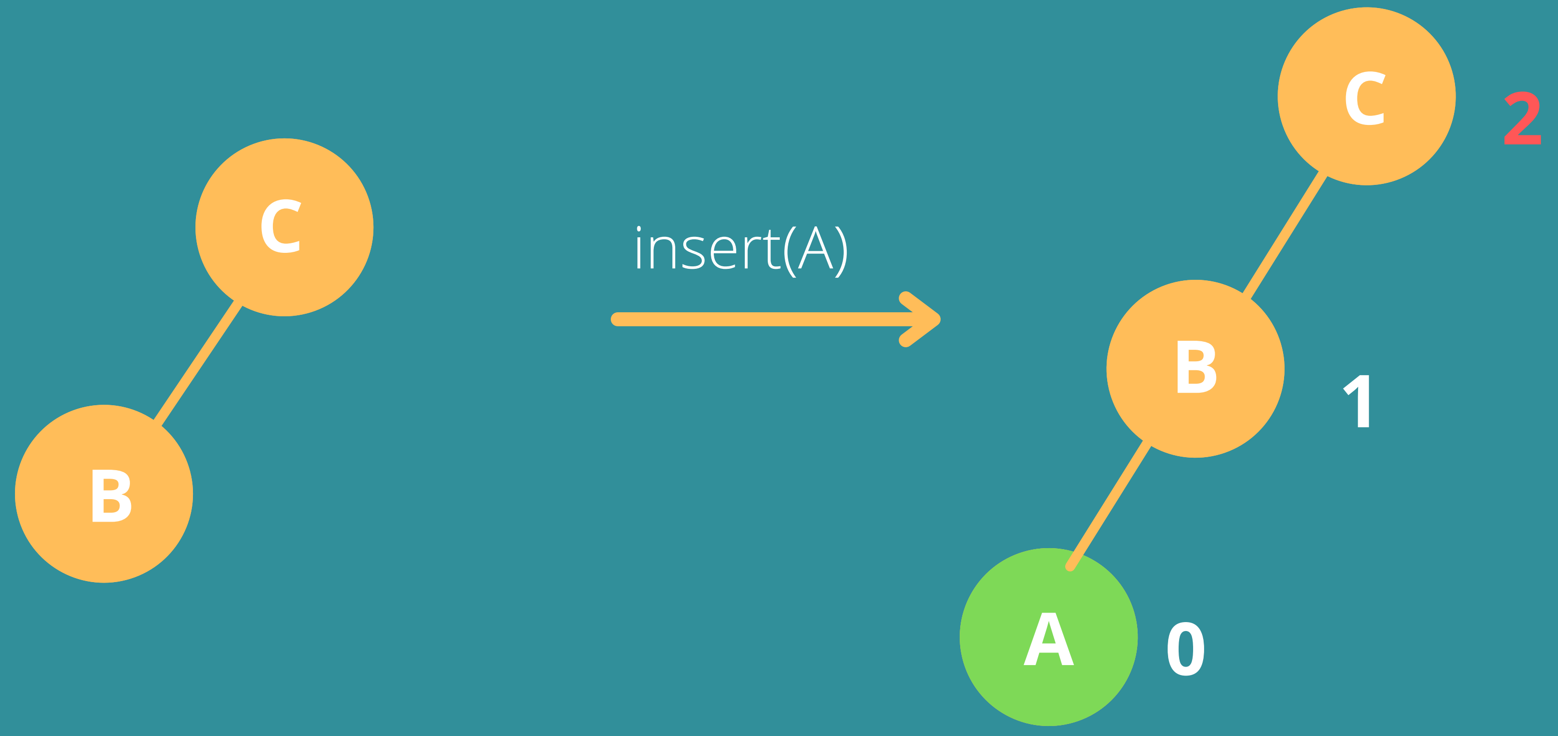
Orginal tree

What if node **A** was not in the tree?

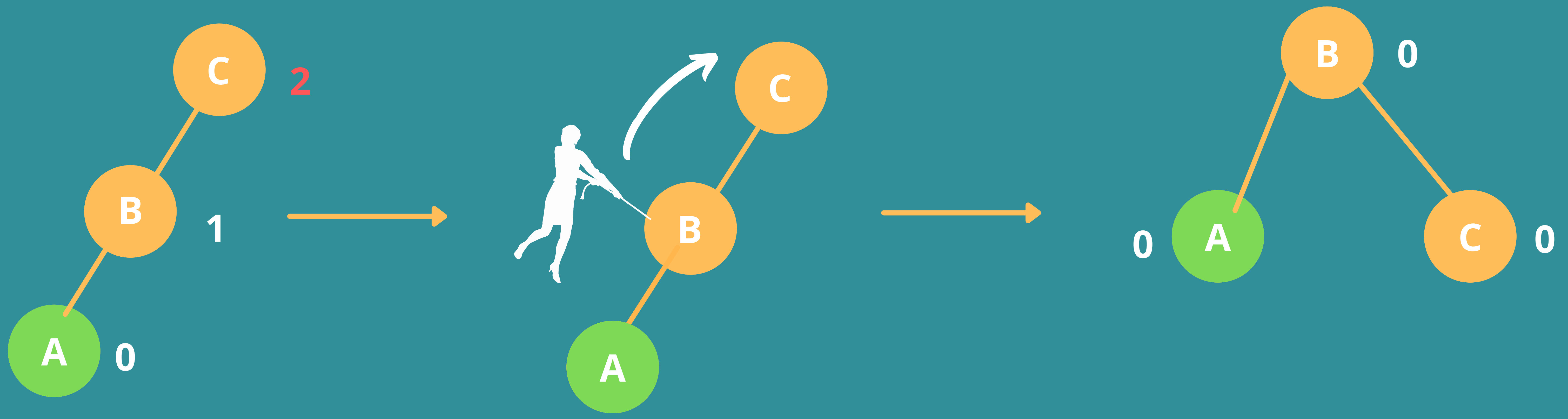
RIGHT ROTATION

Performed when a tree becomes **left imbalanced.**

A tree is left imbalanced when a node is inserted into the left child of the left subtree of the node.



RIGHT ROTATION

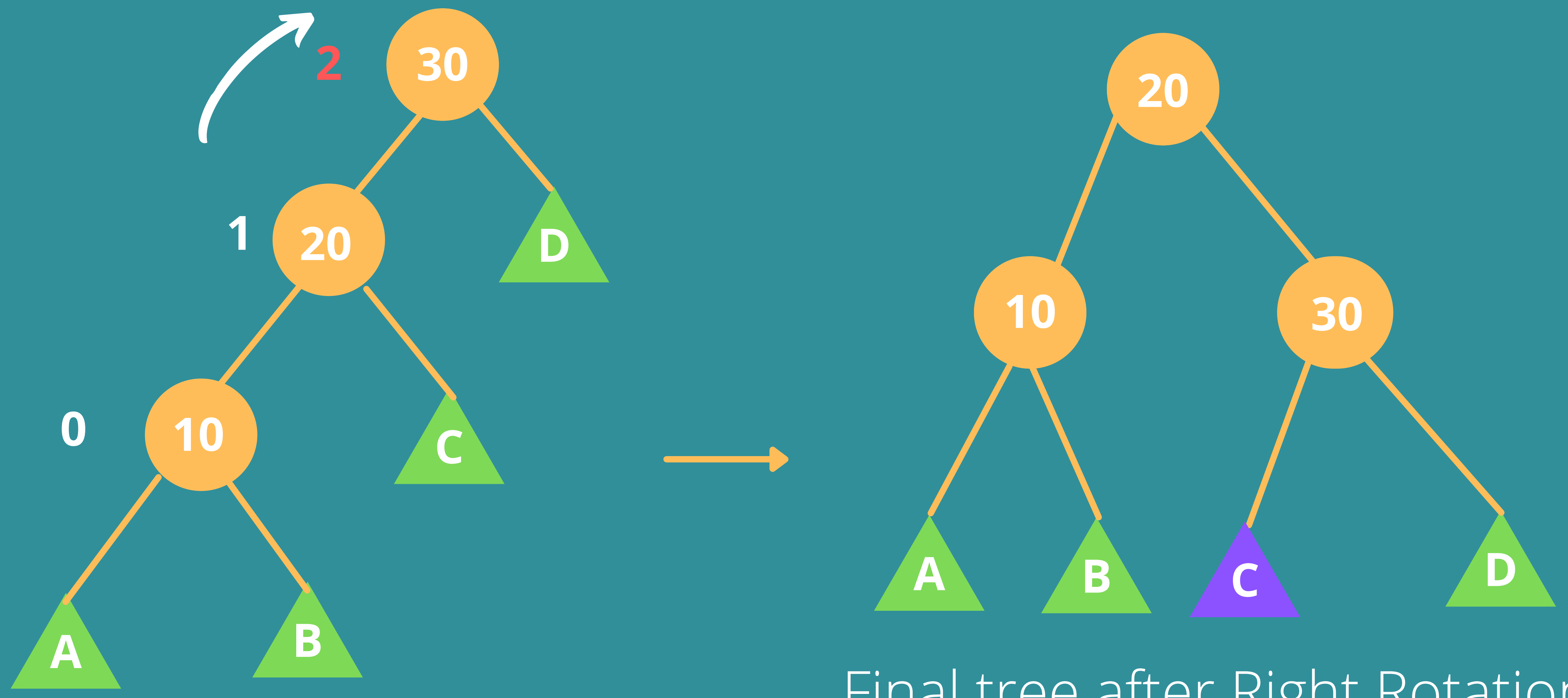


Unbalanced

Right Rotation

Balanced

RIGHT ROTATION

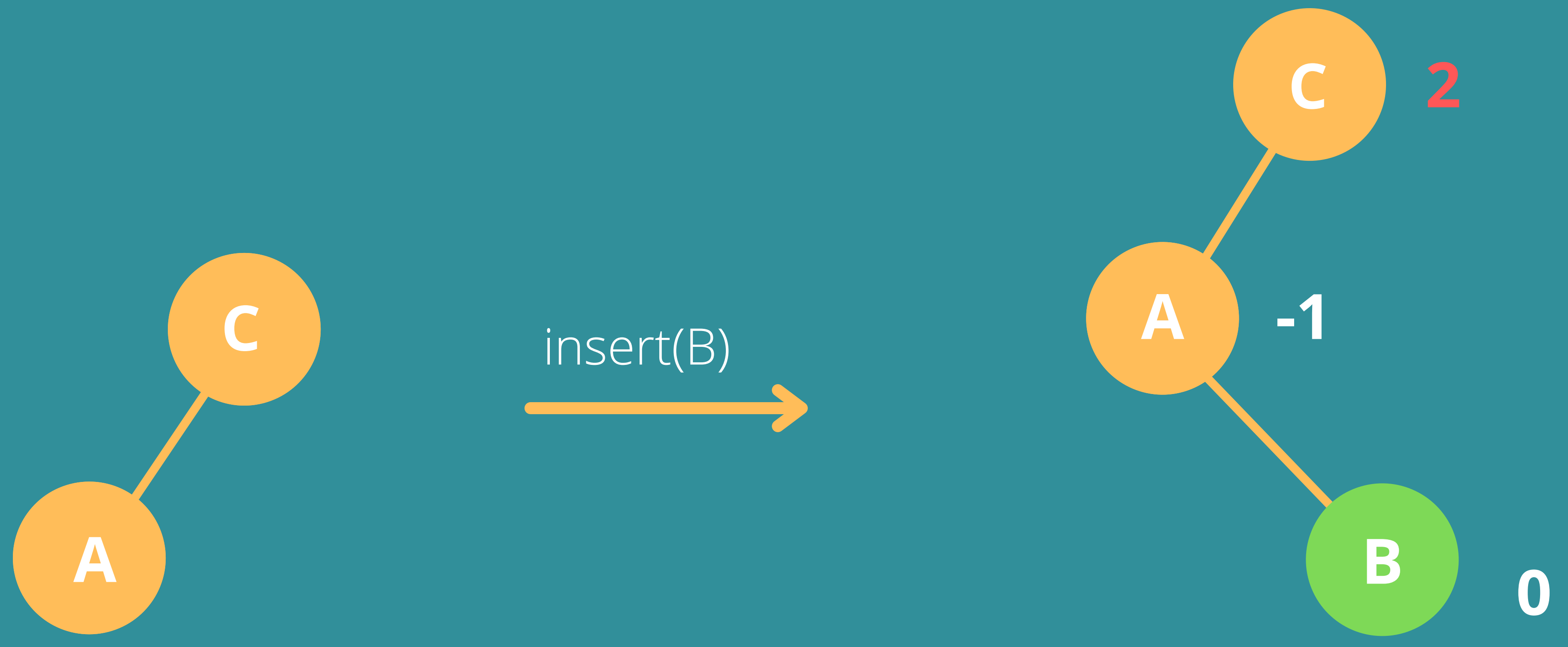


Final tree after Right Rotation

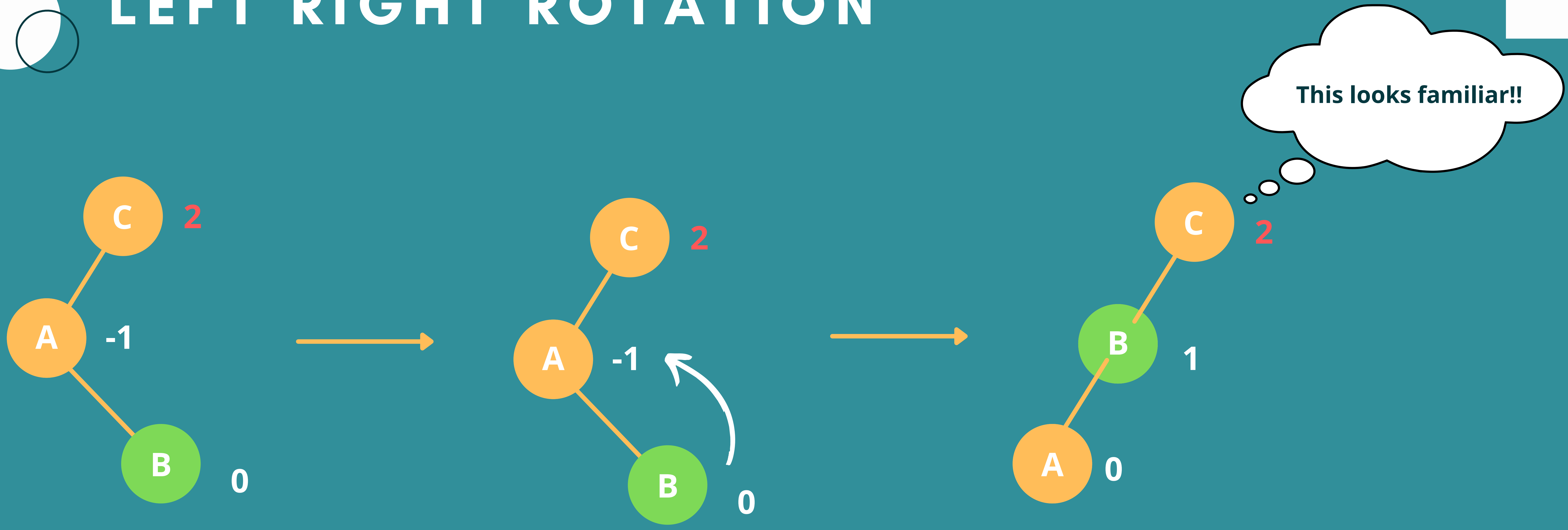
LEFT RIGHT ROTATION

Combination of left rotation followed by right rotation.

Applicable when a node is inserted into the right subtree of the left subtree.



LEFT RIGHT ROTATION

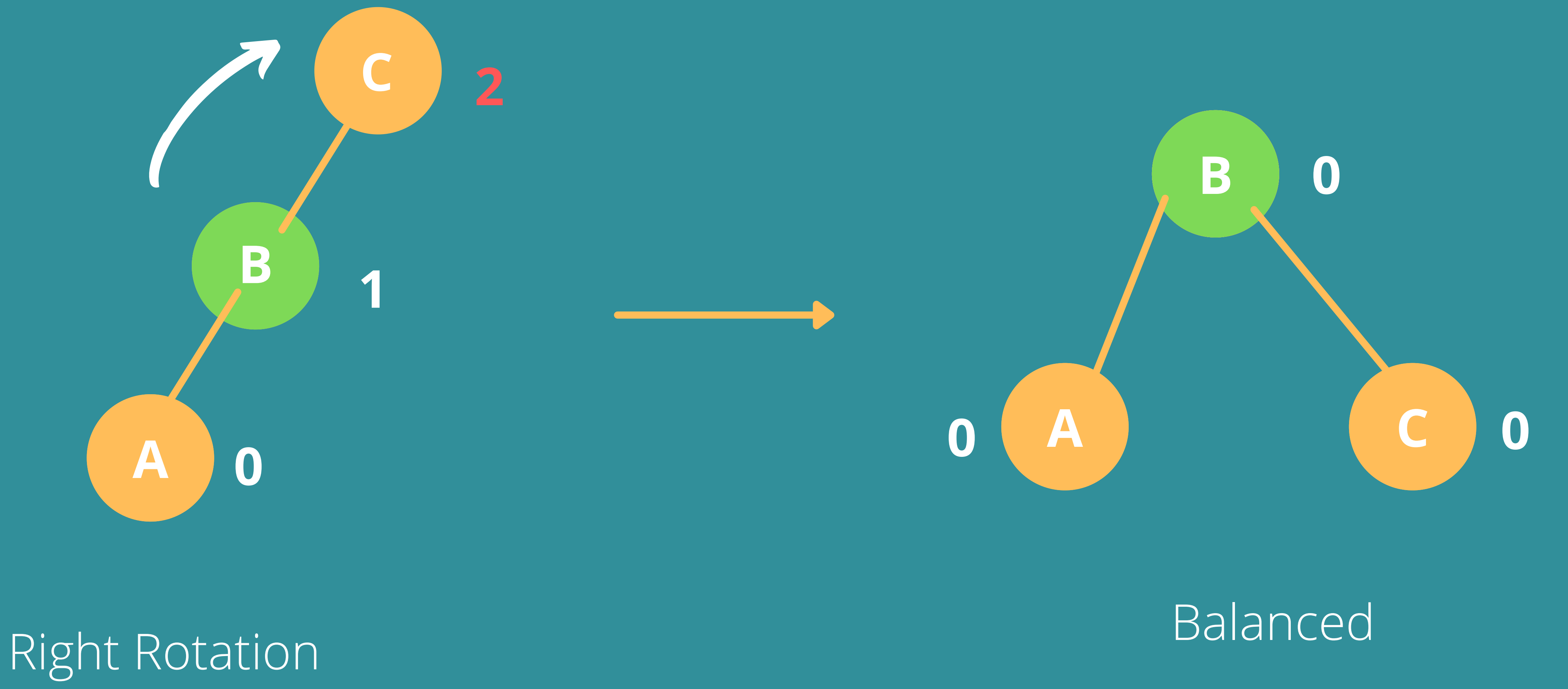


Left Rotation

Still unbalanced

Step 1: Perform a Left Rotation at node B

LEFT RIGHT ROTATION

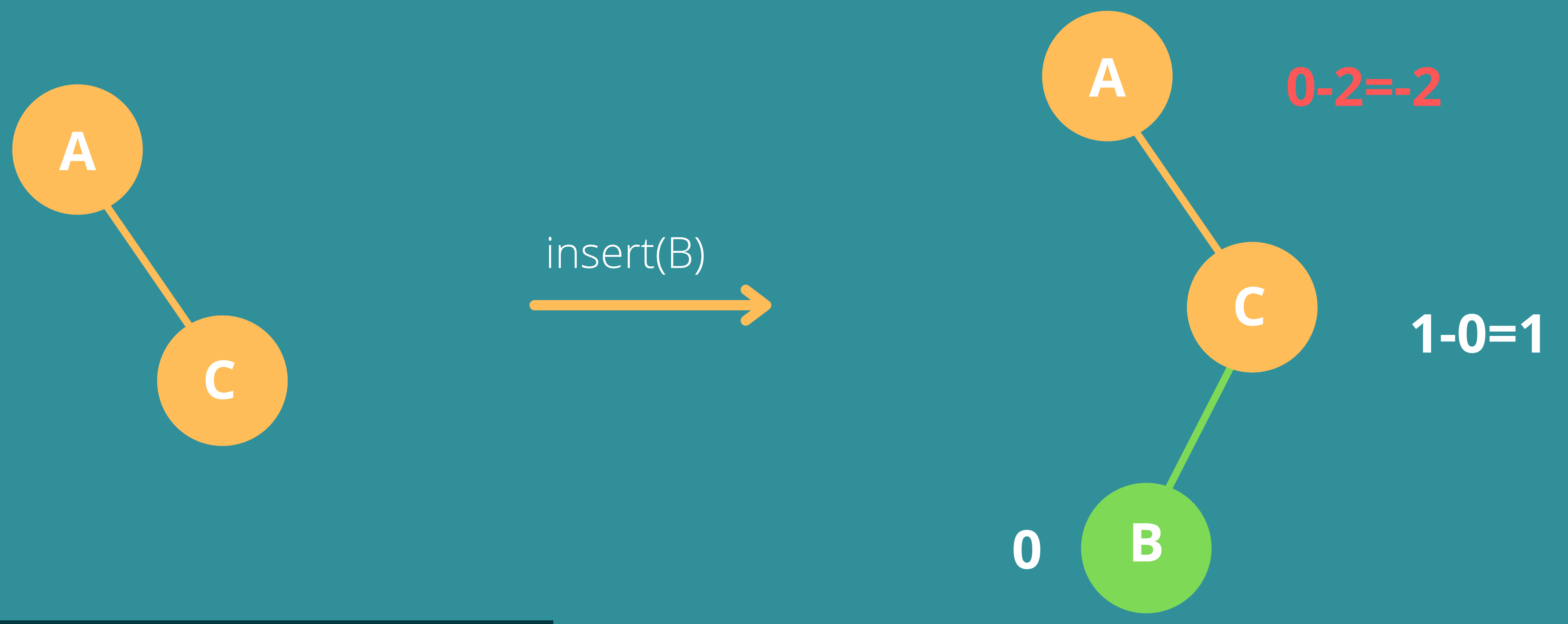


Step 2: Perform a Right Rotation at node B

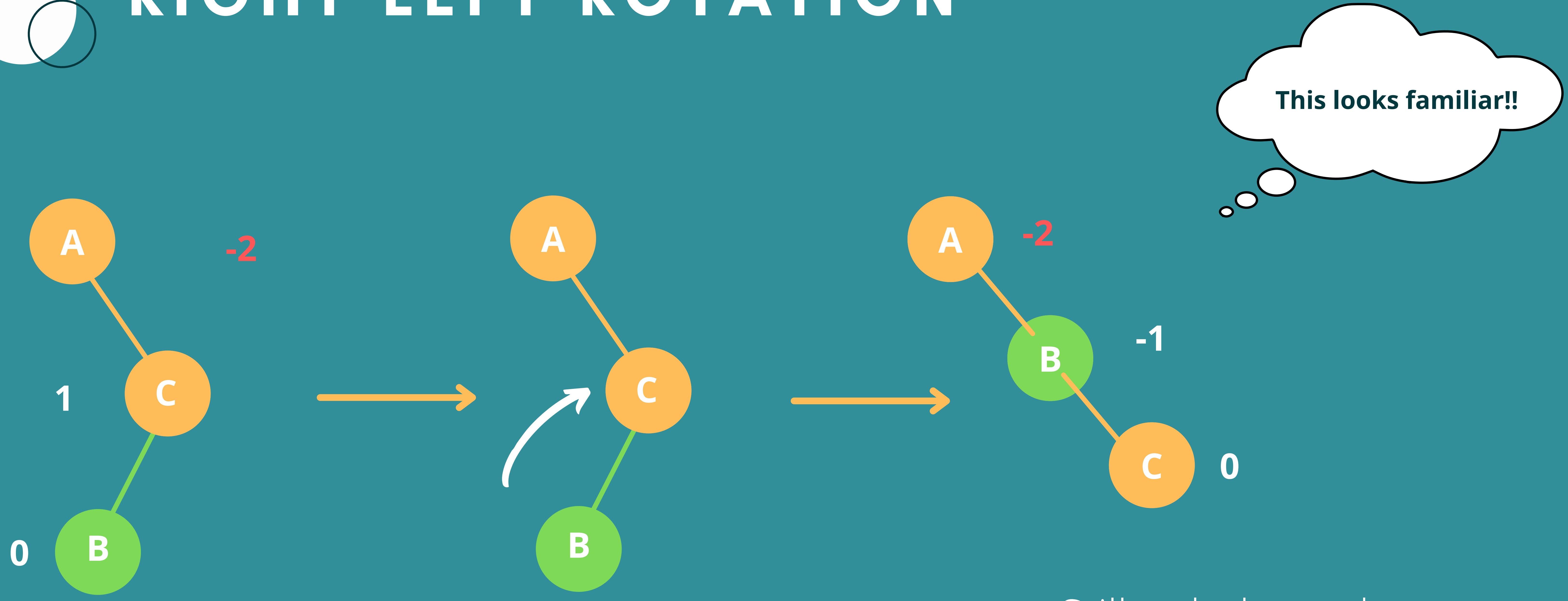
RIGHT LEFT ROTATION

Combination of right rotation followed by left rotation.

Applicable when a node is inserted into the left subtree of the right subtree.



RIGHT LEFT ROTATION

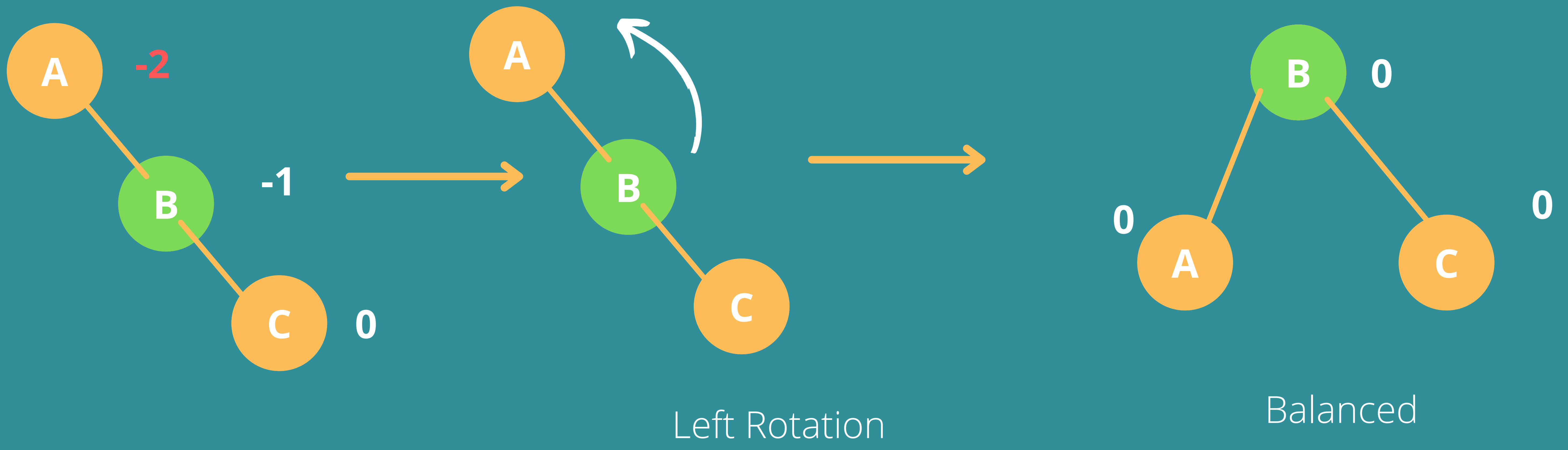


Right rotation

Still unbalanced

Step 1: Perform a Right Rotation at node B

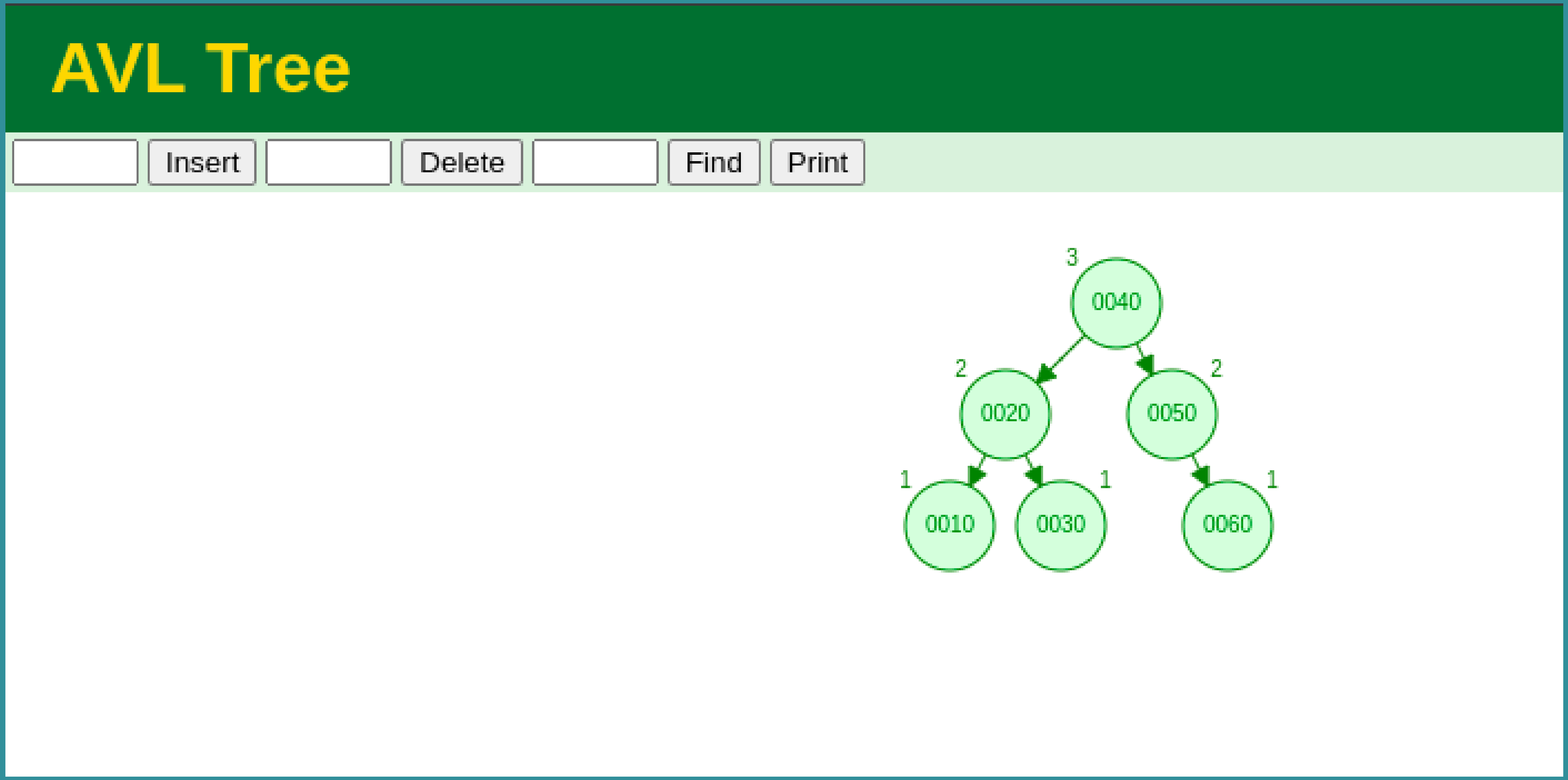
RIGHT LEFT ROTATION



Step 2: Perform a Left Rotation at node B

AVL TREE VISUALIZATION

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html



INSERTION OF A NODE IN AN AVL TREE

- The insertion of a node in an AVL tree is similar to that of a Binary Search Tree (BST).
- After the insertion, we just modify the tree to maintain its balance factor of its node (if required).
- The worst case time complexity for insertion in an AVL tree is O(lg n).

INSERTION: ALGORITHM(BST)

```
InsertionBST(root, newNode)
  1. if root == NULL
  2.     root = newNode
  3.     return root
  4. endif
  5. if newNode→key < root→key
  6.     return InsertionBST(root→left, newNode)
  7. else
  8.     return InsertionBST(root→right, newNode)
  9. endif
```

INSERTION: IMPLEMENTATION

1. Perform the normal BST insertion.
2. Update and check the balance factor of each node starting from the inserted node up till the root.
   - If the **balance factor > 1**, then the tree has imbalance. Since the balance factor is positive, we are either in **Left-Left** case or **Left-Right** case. Check which of the two cases we are in by comparing the newly inserted key with the key in left subtree root.
   - If the **balance factor < -1**, then the tree has imbalance. SInce the balance factor is negative, we are either in **Right-Right** case or **Right-Left** case. Check which of the two cases we are in by comparing the newly inserted key with the key in right subtree root.
3. Maintain the balance factor of the nodes (if not balanced) starting from the inserted node up till the root by performing the appropriate rotations.

INSERTION OF A NODE IN AN AVL TREE

Let us consider an example with data in the following order:

1, 3, 6, 4, 5

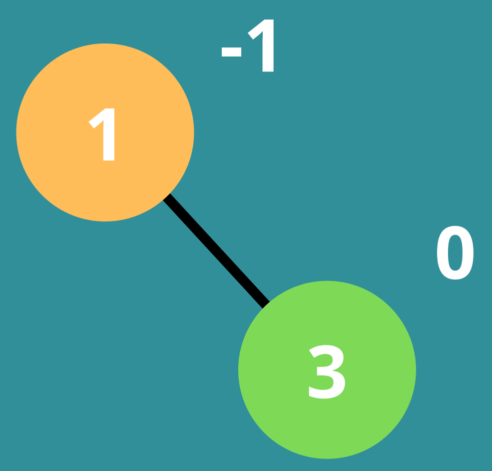
INSERTION OF A NODE IN AN AVL TREE

Data: 1, 3, 6, 4, 5

First the tree is empty, so we insert 1 as a root node.

(Balanced)

Now, we insert 3 to the right of 1.

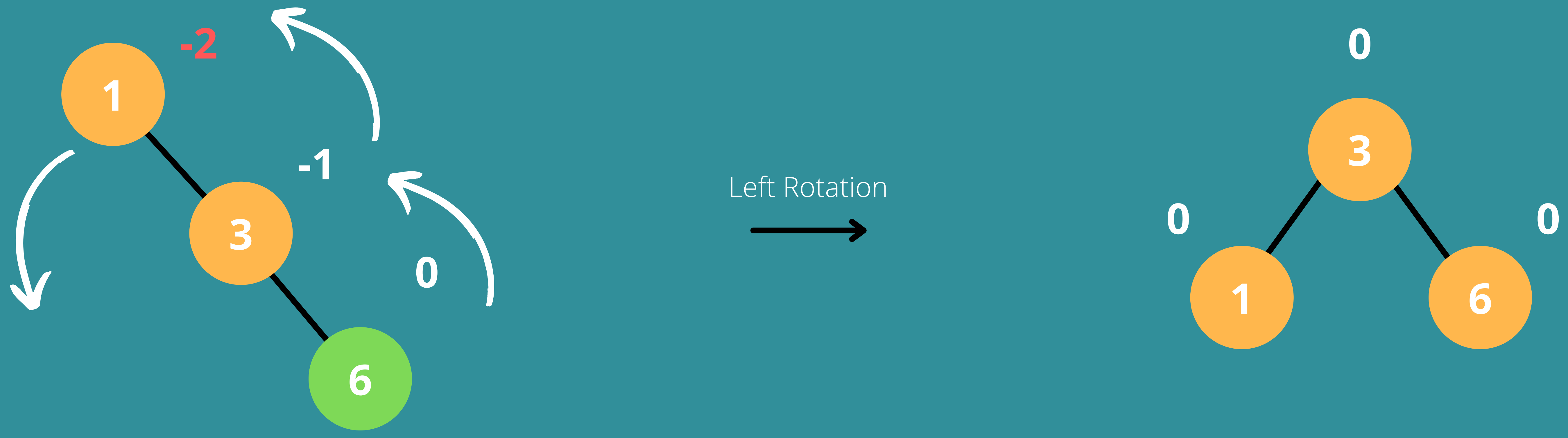


(Balanced)

INSERTION OF A NODE IN AN AVL TREE

Data: 1, 3, 6, 4, 5

Since the tree is balanced, we now add the next node i.e. 6 to the tree.



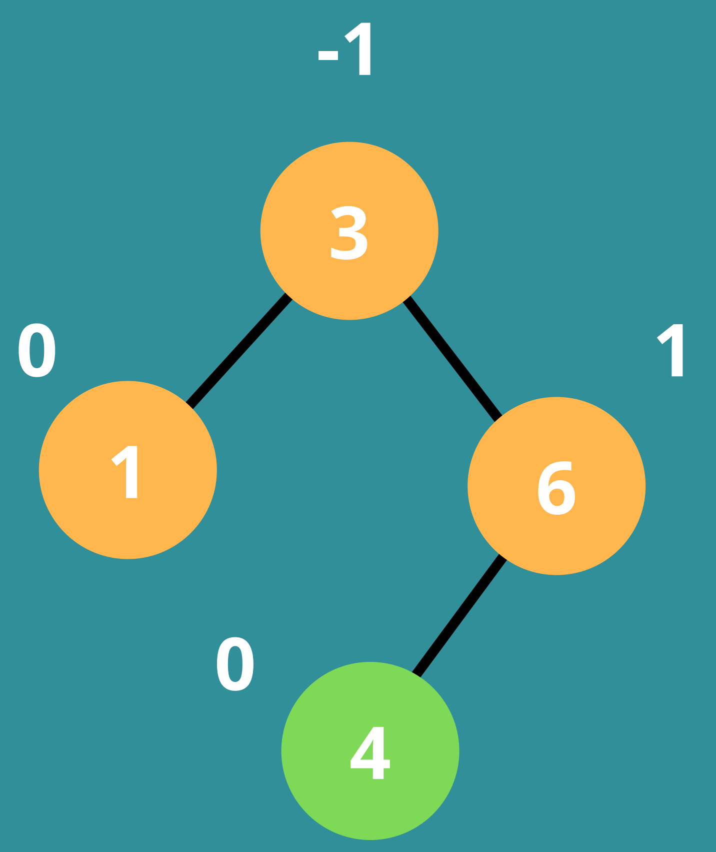
(Right Imbalance)

(Balanced)

INSERTION OF A NODE IN AN AVL TREE

Data: 1, 3, 6, 4, 5

Since the tree is balanced, we now add the next node i.e. 4 to the tree.

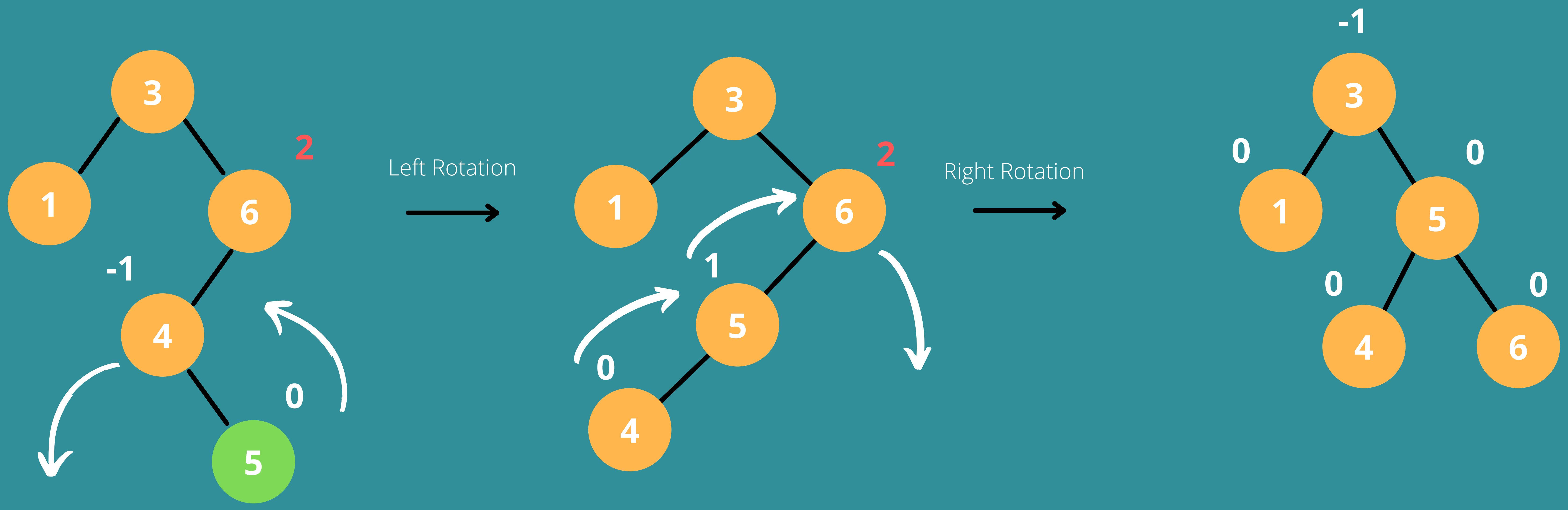


(Balanced)

INSERTION OF A NODE IN AN AVL TREE

Data: 1, 3, 6, 4, 5

Since the tree is balanced, we now add the next node i.e. 5 to the tree.



(Left-Right case)

(Left imbalance)

(Balanced)

DELETION OF A NODE FROM AN AVL TREE

- The deletion of a node in an AVL tree is similar to that of a Binary Search Tree (BST).
- After the deletion, we just modify the tree to maintain the balance factor of its nodes (if required).
- The worst case time complexity for deletion in an AVL tree is O(lg n).

DELETION: ALGORITHM(BST)

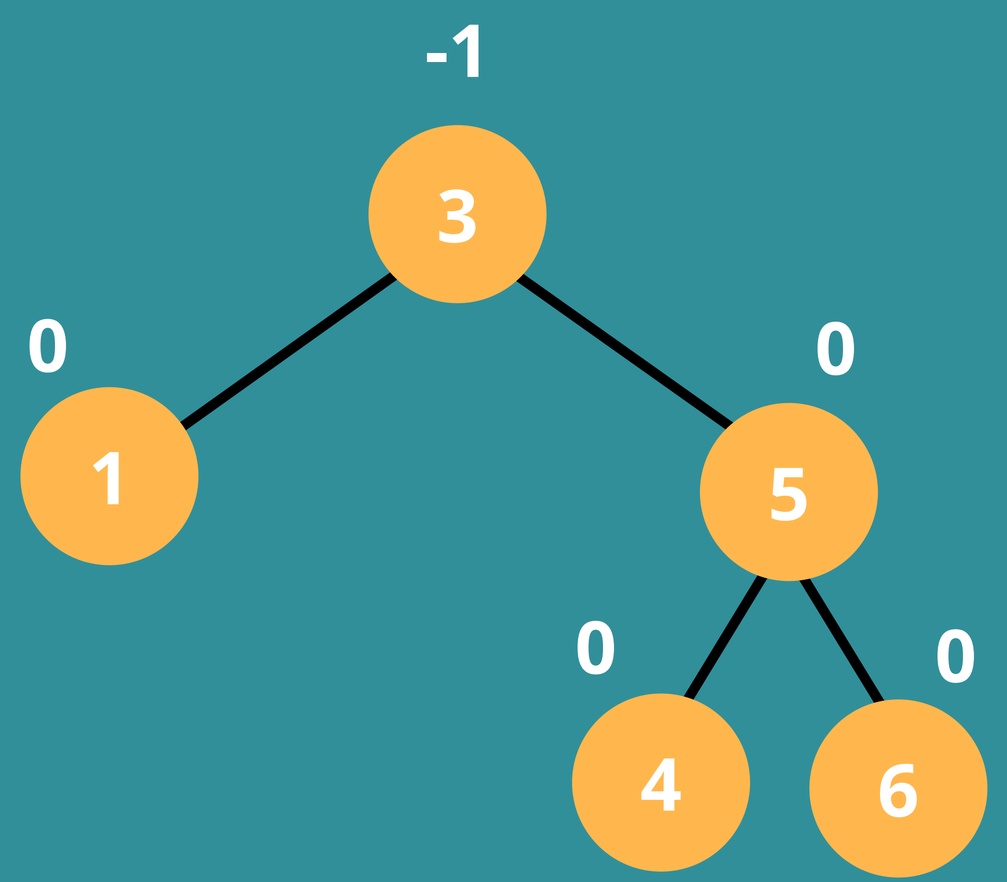
```
DeletionBST(root, dltKey)
if root == NULL
    return false
endif
if dltKey < root→key
    return deleteBST(root→left, dltKey)
elseif dltKey > root→key
    return deleteBST(root→right, dltKey)
else
    if root→left == NULL
        root = root→right
        return true
    elseif root→right == NULL
        root = root→left
        return true
    else
        nodeToDelete = root
        largest = the largest node in left subtree
        nodeToDelete→key = largest→key
        return deleteBST(nodeToDelete→left, largest→key)
    endif
endif
```

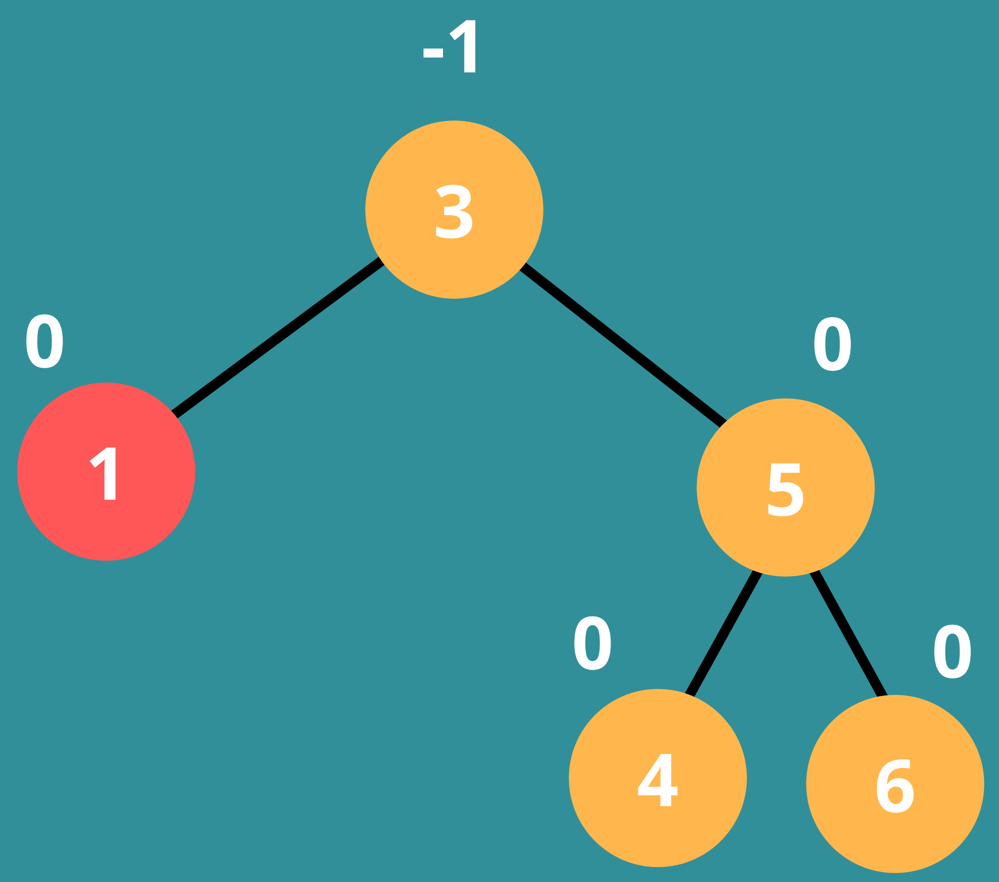
DELETION: IMPLEMENTATION

1. Perform the normal BST deletion.
2. Update and check the balance factor of each node starting from the parent node of the recently deleted node (not the key) up till the root.
   - If the **balance factor > 1**, then the tree has imbalance. Since the balance factor is positive, we are either in **Left-Left** case or **Left-Right** case. To check whether it is Left-Left case or Left-Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left-Left case, else Left-Right case.
   - If the **balance factor < -1**, then the tree has imbalance. SInce the balance factor is negative, we are either in **Right-Right** case or **Right-Left** case. To check whether it is Right-Right case or Right-Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right-Right case, else Right-Left case.
3. Maintain the balance factor of the nodes (if not balanced) starting from the parent node of the recently deleted node up till the root by performing the appropriate rotations.

DELETION OF A NODE FROM AN AVL TREE

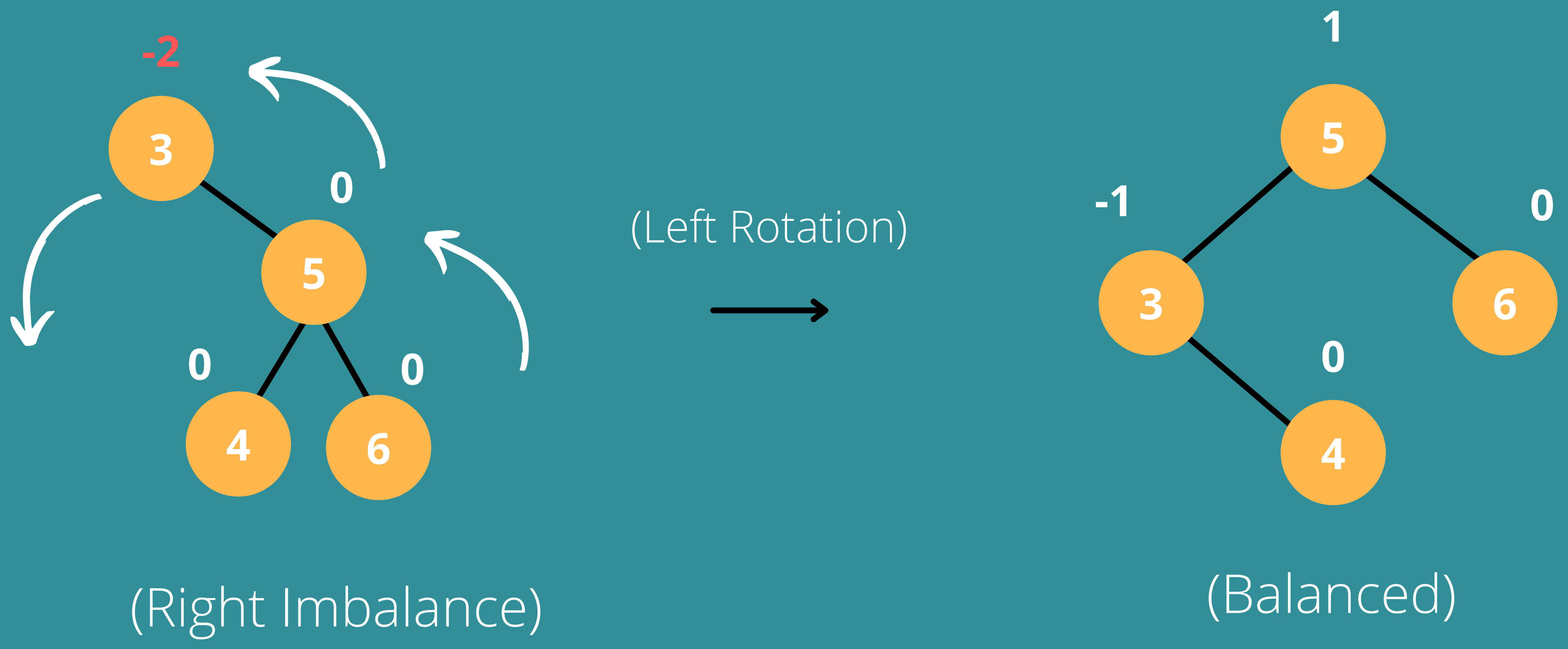
Let us consider an AVL tree:





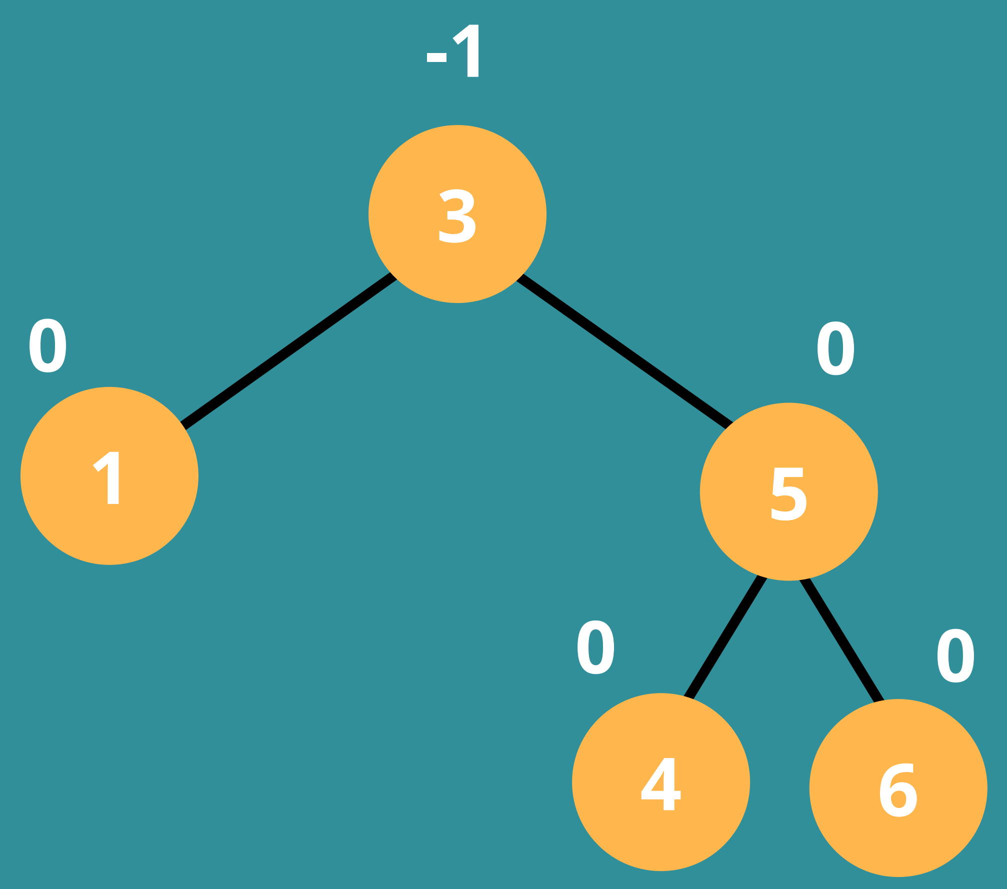
Suppose we need to delete the key 1 from the tree.

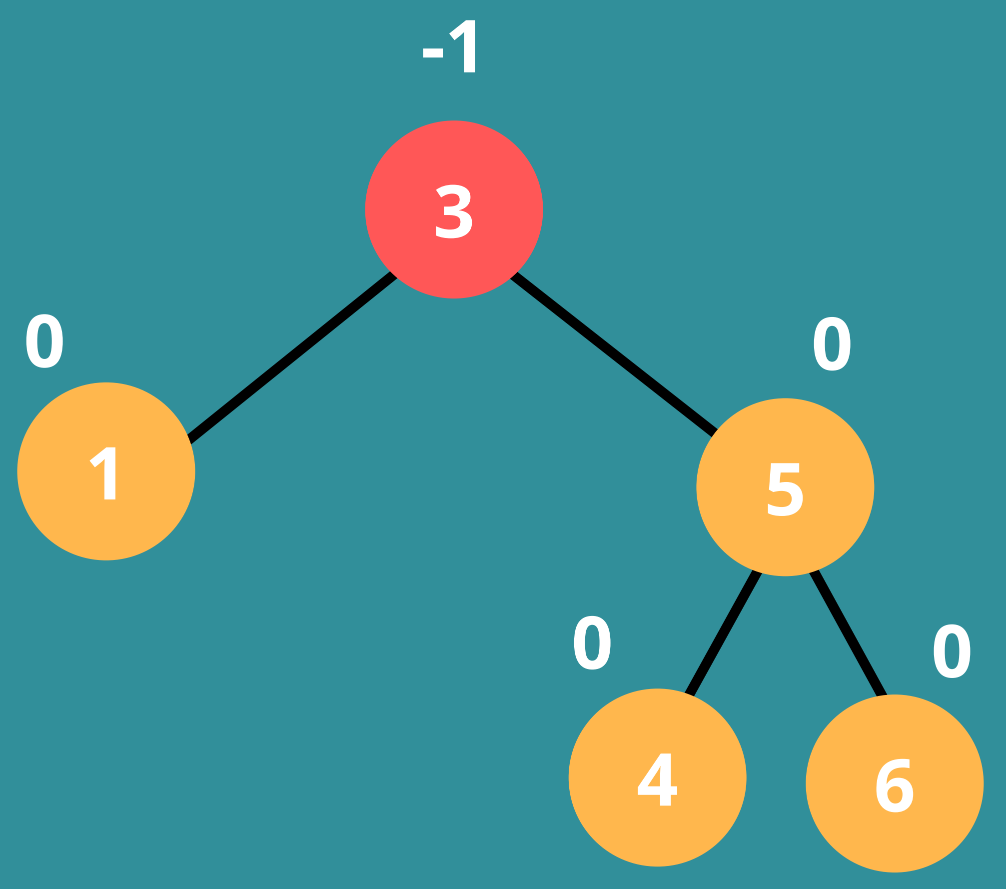
DELETION OF A NODE FROM AN AVL TREE



DELETION OF A NODE FROM AN AVL TREE

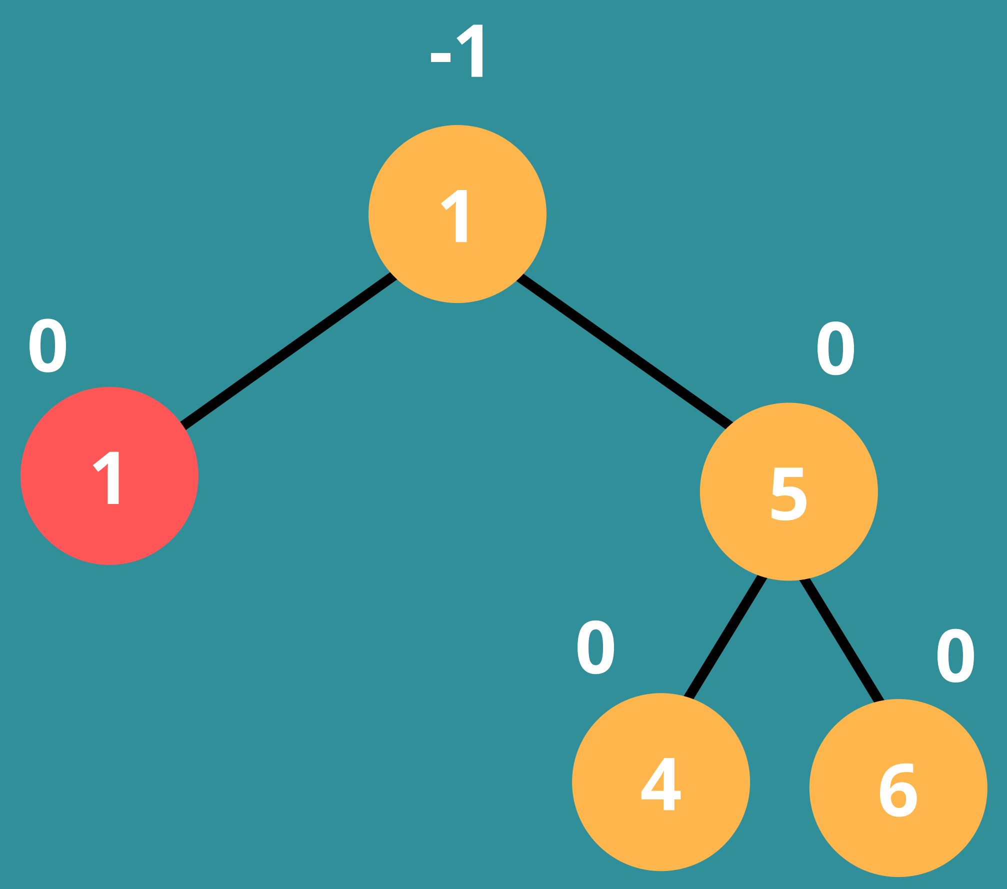
Let us again consider the same original AVL tree:



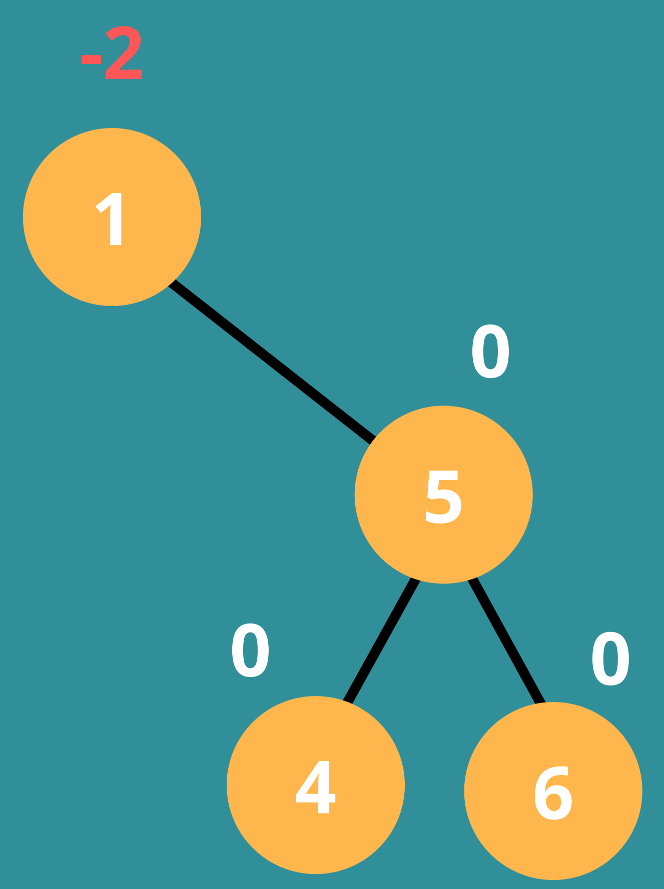


Suppose we need to delete the key 3 from the tree.

DELETION OF A NODE FROM AN AVL TREE

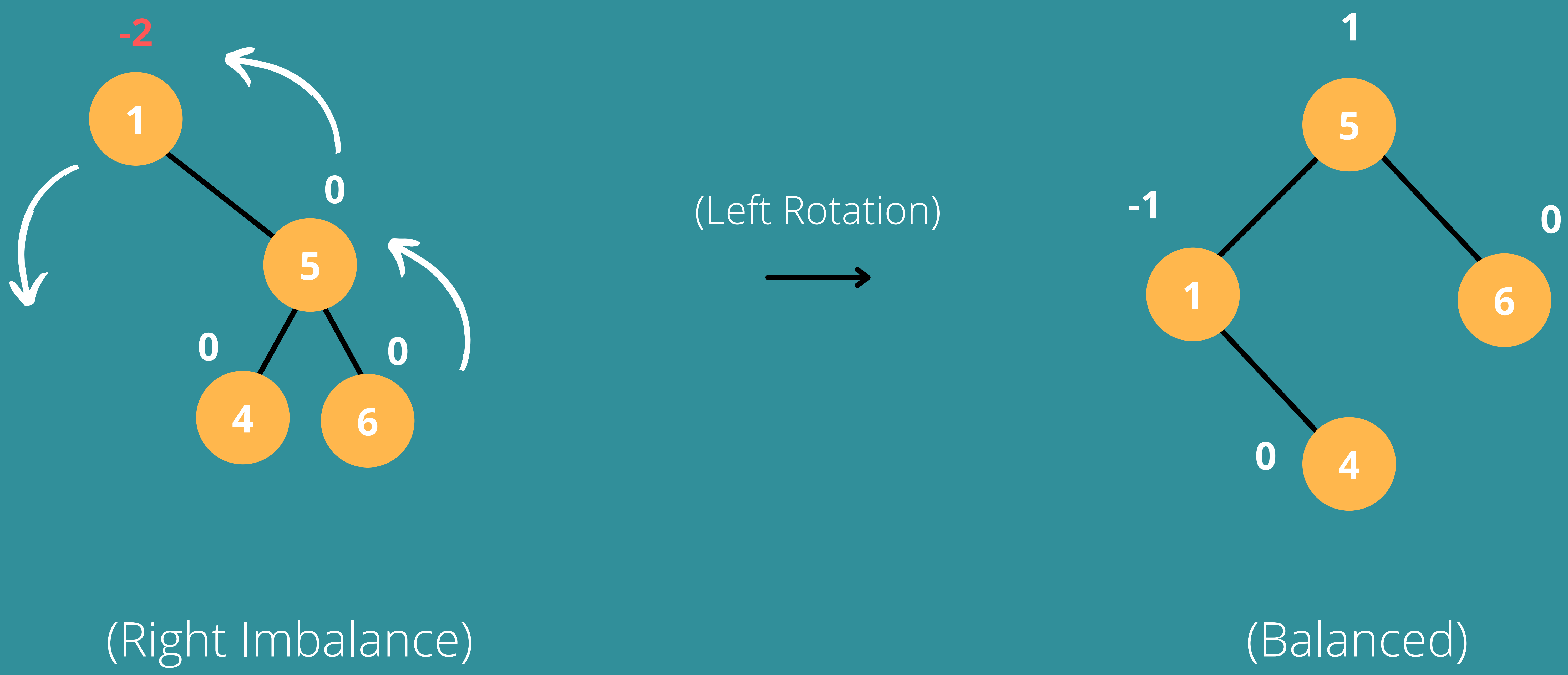


Replace by 1 (Can be replaced by 4 as well)



(Right Imbalance)

DELETION OF A NODE FROM AN AVL TREE



THANK YOU