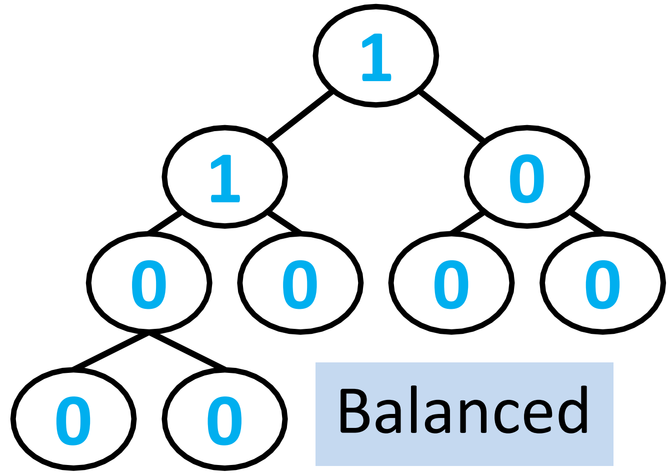
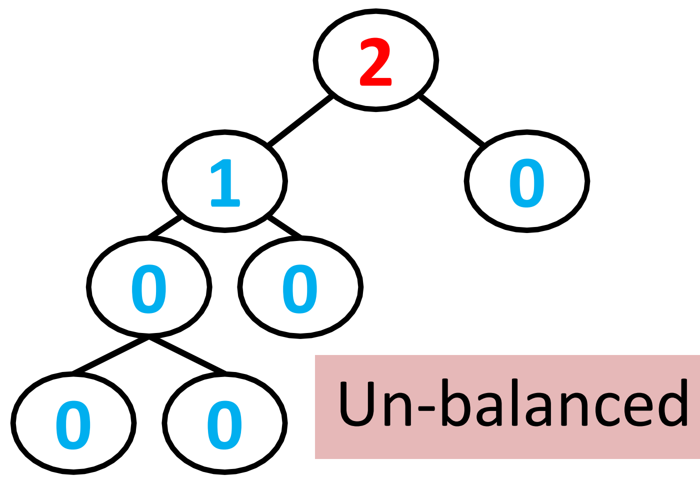
# AVL Trees

# Introduction

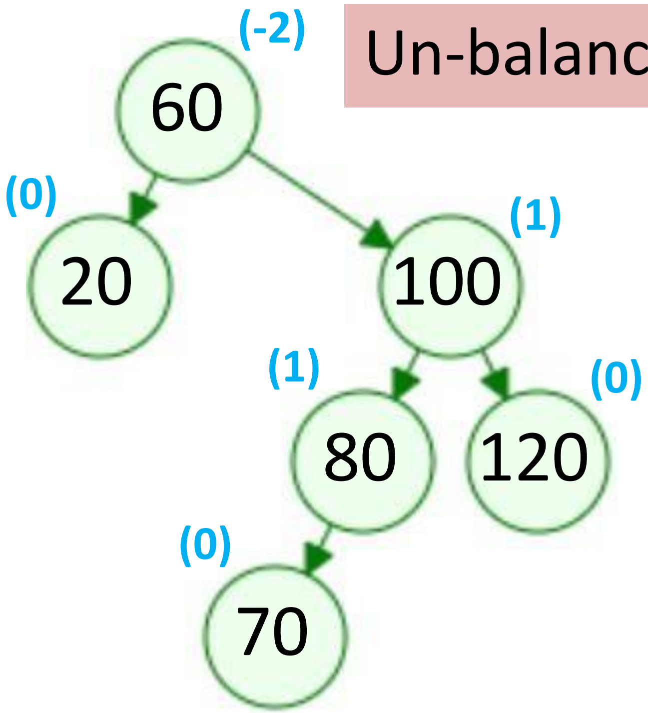
**Note:** Numbers within nodes represent height difference, i.e. **height of left sub-tree – height of right sub-tree**.

- In a Binary Search Tree with n nodes
  - Average case height is O(log n)
  - Worst case height is O(n)
- Thus it would be nice to be able to maintain a balanced tree during insertion.
  - A binary tree is said to be balanced if, for every node in the tree, the height of its two subtrees differs at most of one.

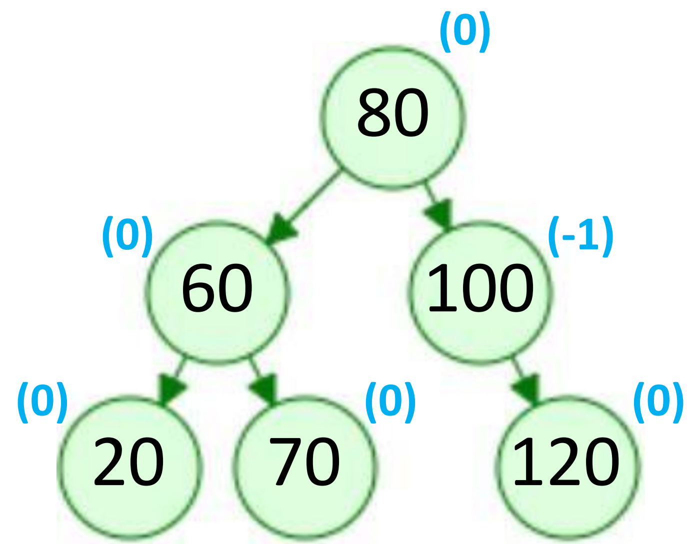


# Balanced BST???




Un-balanced BST




Balanced BST

# Contd…

- Invented by Georgy **A**delson-**V**elsky and Evgenii **L**andis in 1962.
- Height balanced binary search trees.
- Each node has a balance factor.
- Let **HL** and **HR** be the heights of left and right subtrees of any node, then

$$|\mathbf{HL} - \mathbf{HR}| <= 1$$

- Balance factor (**bal**) of a node **K** is **HL** – **HR**.
  - Left High (LH) = +1 (left sub-tree higher than right sub-tree)
  - Even High (EH) = 0 (left and right sub-trees have same height)
  - Right High (RH) = -1 (right sub-tree higher than left sub-tree)

# Height of AVL Trees

- **Guaranteed to be in the order of $\lg_2 n$ for a tree containing n nodes.**
- If an AVL tree has minimum number of nodes, then one of its subtrees is higher than the other by 1.
- Let, the left subtree is bigger than the right subtree, and
  - $N(h)$ = minimum number of nodes in an AVL tree of height $h$ rooted at $r$.
  - $N(h-1)$ = minimum number of nodes in the left subtree of $r$.
  - $N(h-2)$ = minimum number of nodes in the right subtree of $r$.

# Contd…

$N(h) = 1 + N(h-1) + N(h-2)$

As per assumption $N(h-1) > N(h-2)$, so

$N(h) > 1 + N(h-2) + N(h-2) = 1 + 2 \cdot N(h-2) > 2 \cdot N(h-2)$

That is,

$N(h) > 2 \cdot N(h-2)$

Knowing $N(0) = 1$, this recurrence can be solved.

$N(h) > 2 \cdot N(h-2) > 2 \cdot 2 \cdot N(h-4) > 2 \cdot 2 \cdot 2 \cdot N(h-6) > \cdots > 2^{h/2}$

To ensure it's $2^{h/2}$, lets check for a particular $h = 6$

$N(6) > 2 \cdot N(6-2) > 2 \cdot 2 \cdot N(4-2) > 2 \cdot 2 \cdot 2 \cdot N(2-2) > 2^3$

# Contd…

Thus,

$$N(h) > 2^{h/2}$$

Taking log,

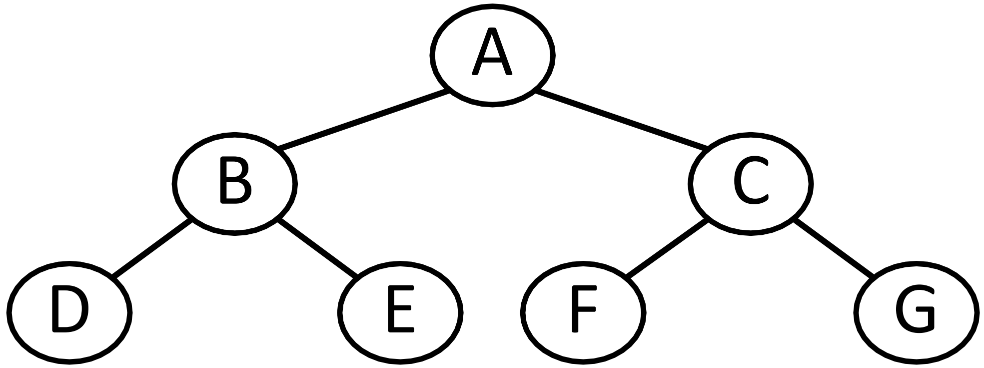
$$\log N(h) > \log 2^{h/2} \Leftrightarrow h < 2 \log N(h)$$

- Thus, in the worst-case AVL trees have height $h$ = O(log $n$).
- This means that nicer/more balanced AVL trees will have the same bound on their height.

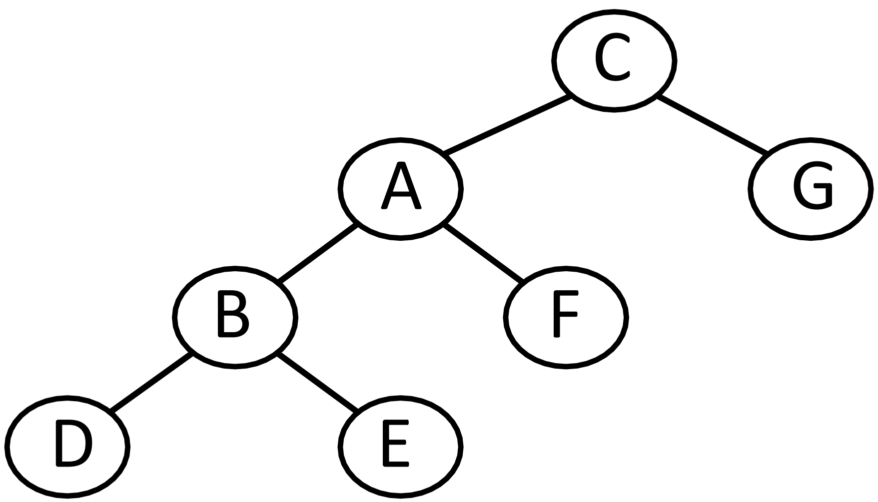
# Operations on AVL Tree

- Search
  - Similar as in the case of binary search trees, since both are organized according to the same criteria.
  - Complexity O(lg n).
- Insertion and Deletion
  - Similar as in the case of binary search trees. But after insertion or deletion of a node, the tree might have lost its AVL property (i.e. balance factor becomes greater than 1).
  - To maintain the AVL structure, further modifications (known as **ROTATIONS**) are required.

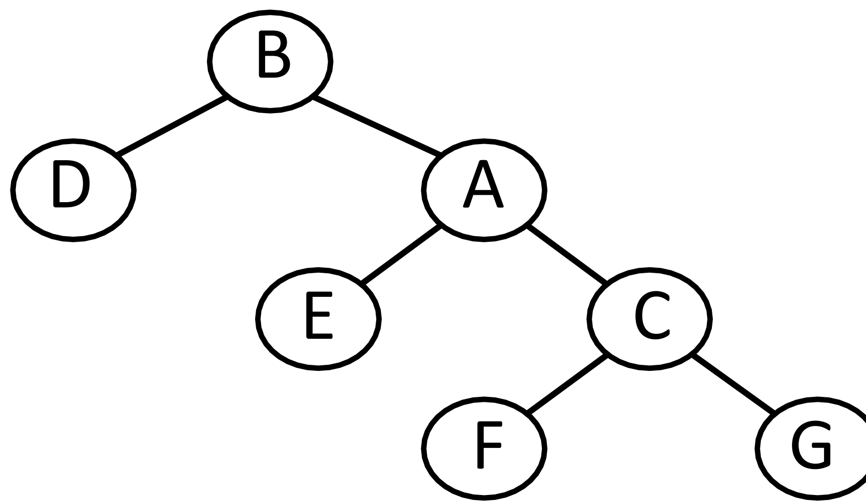
# Rotation




- Left rotation

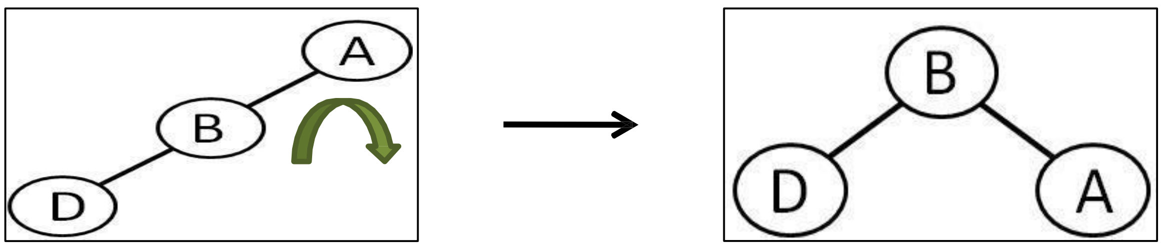



- Right rotation

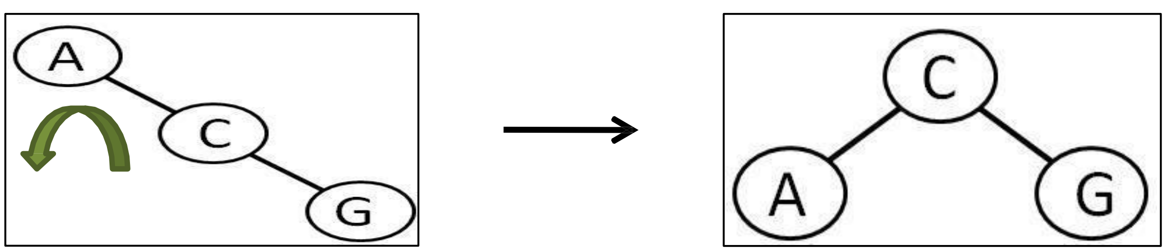


# Unbalanced Cases

- Single rotation
  - **Left of Left**: insertion turned the left subtree of a left high AVL tree into a left high tree.

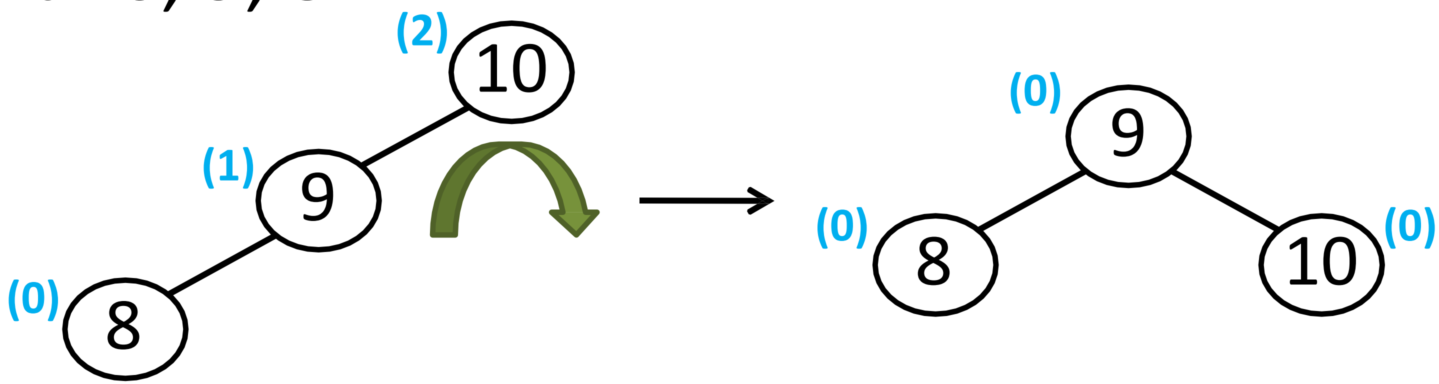



  - **Right of Right**: insertion turned the right subtree of a right high AVL tree into a right high tree.

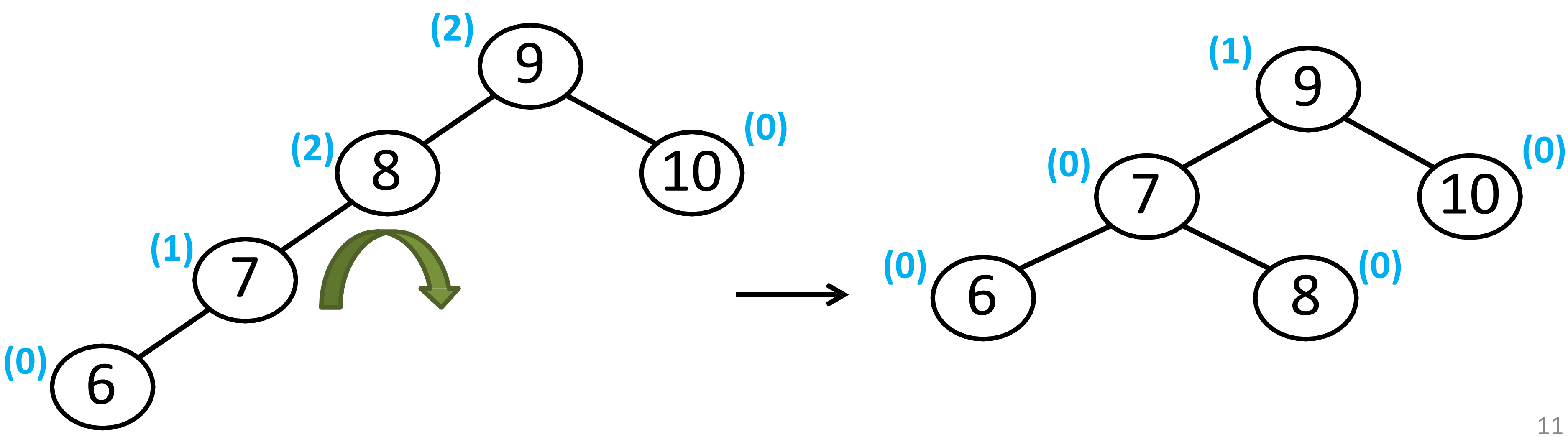


# Example 1: Insert 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

- Insert 10, 9, 8

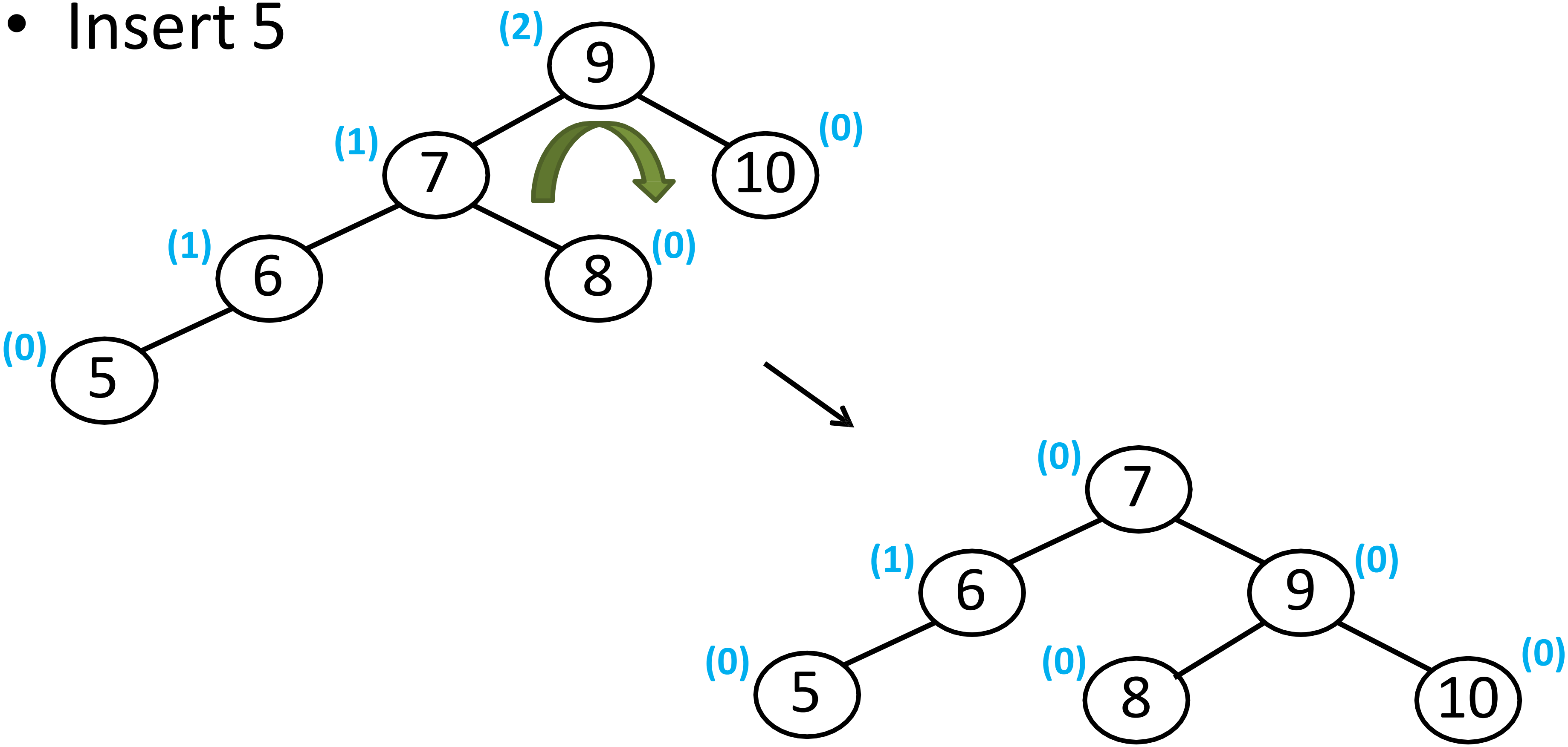



- Insert 7, 6



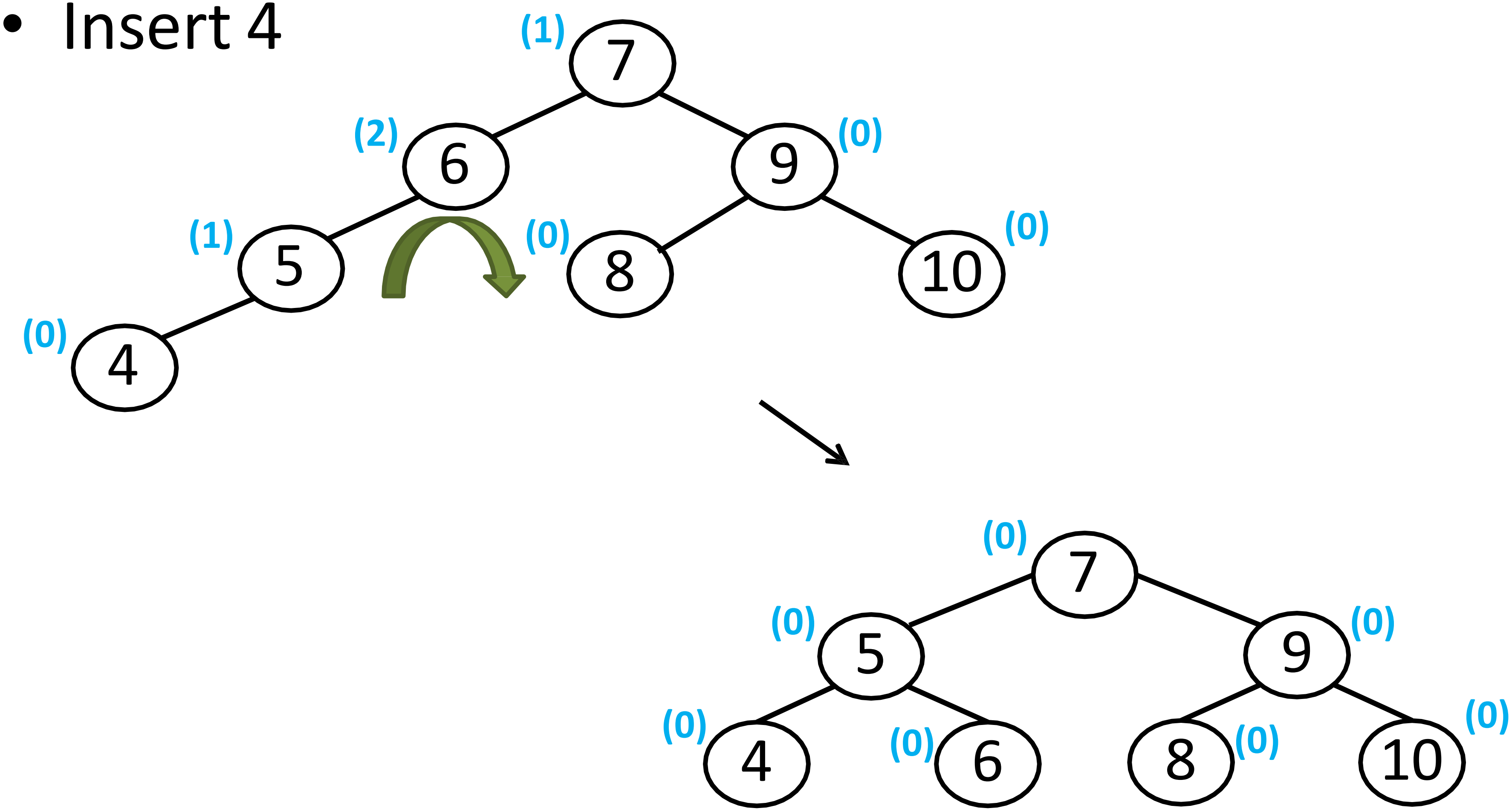
# Contd…

- Insert 5



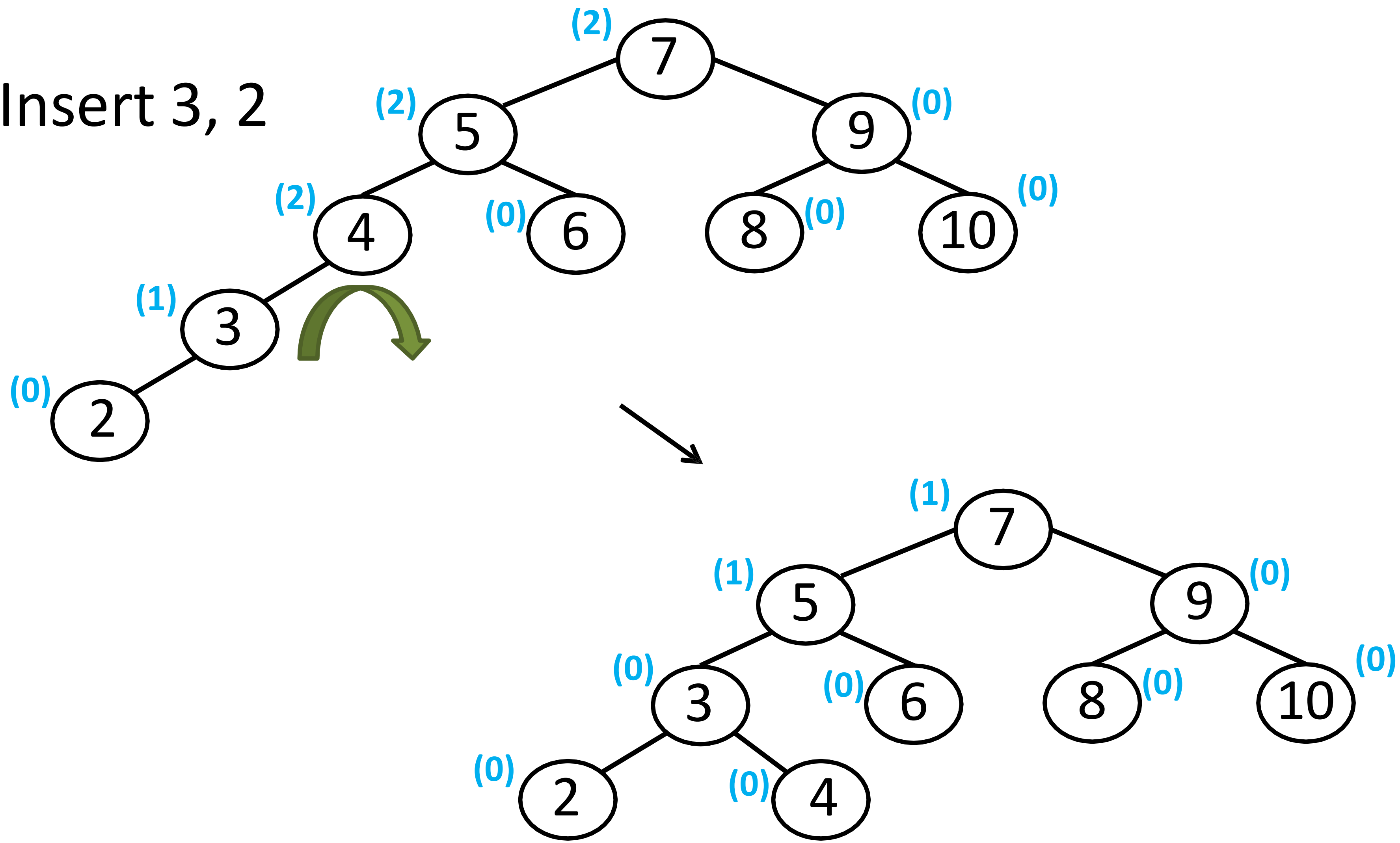
# Contd…

- Insert 4



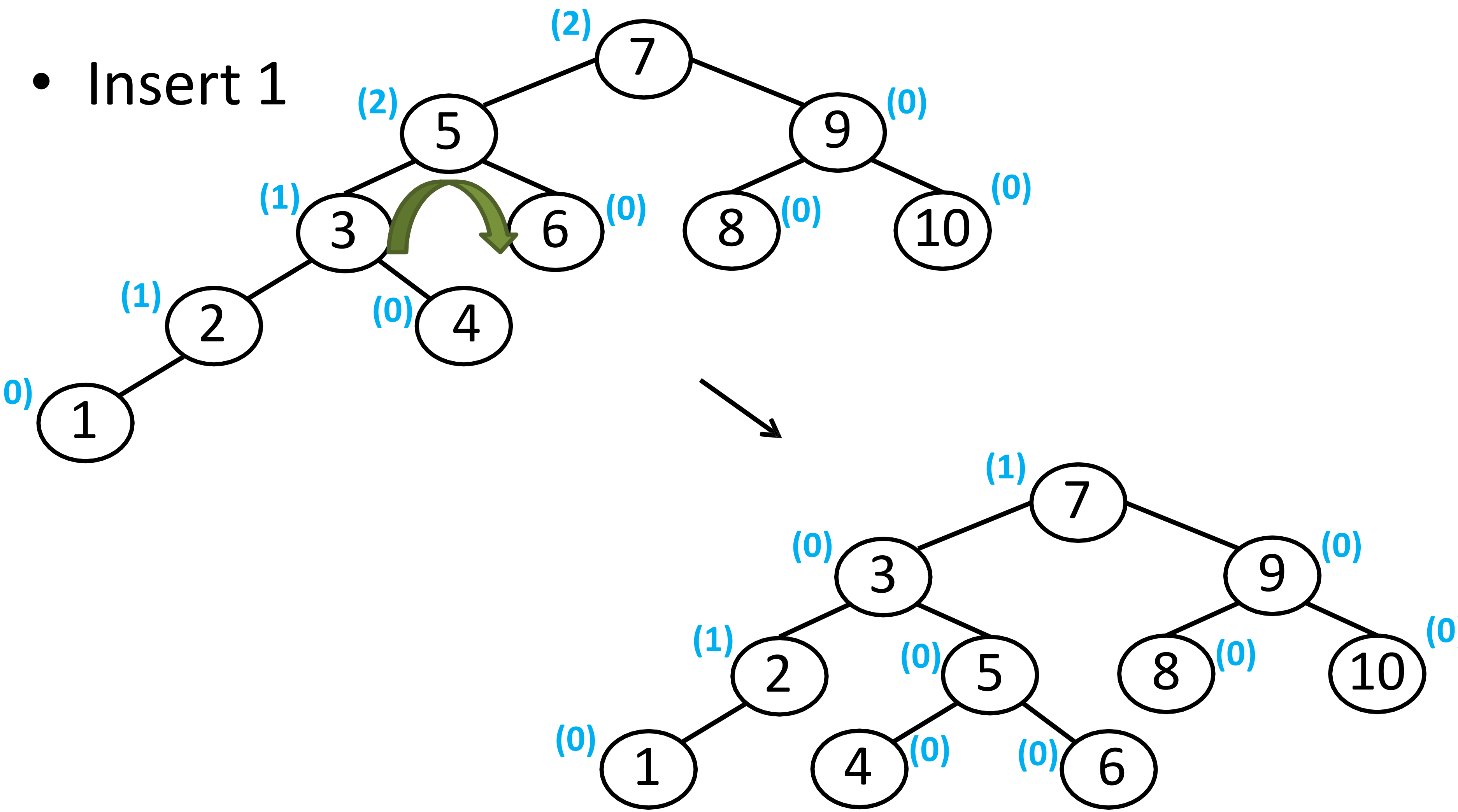
# Contd…

- Insert 3, 2



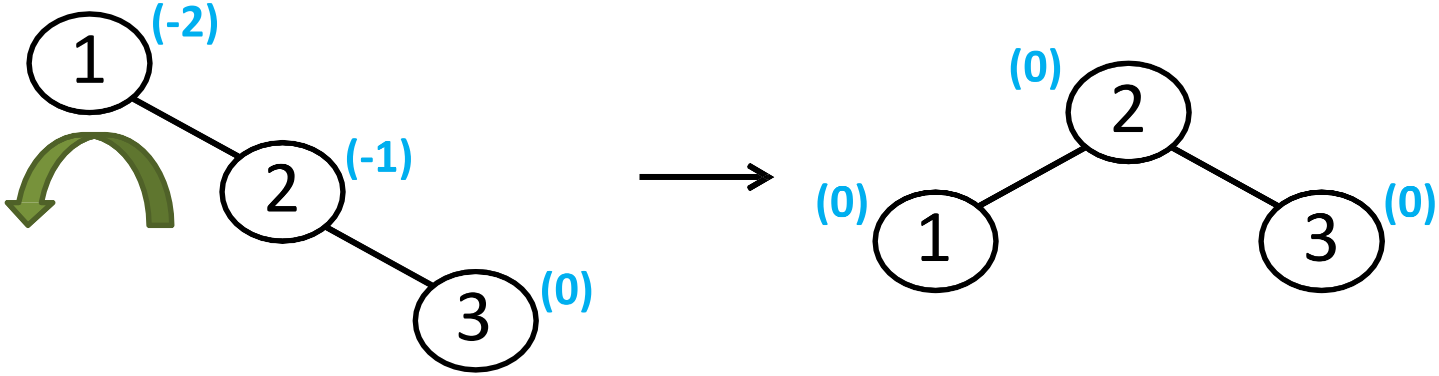
# Contd…

- Insert 1

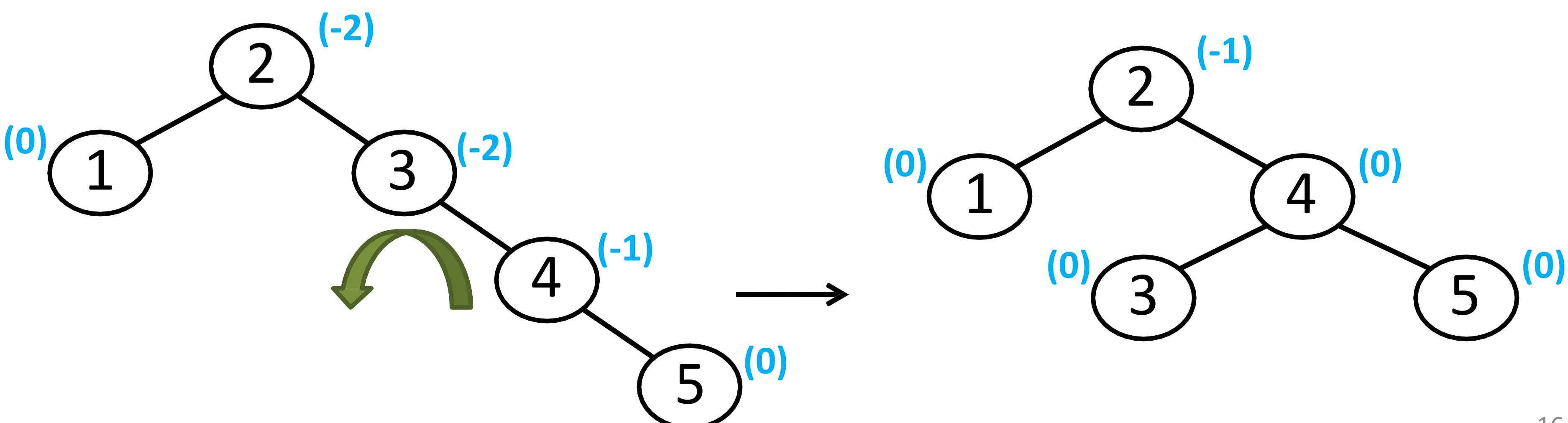


# Example 2: Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

- Insert 1, 2, 3

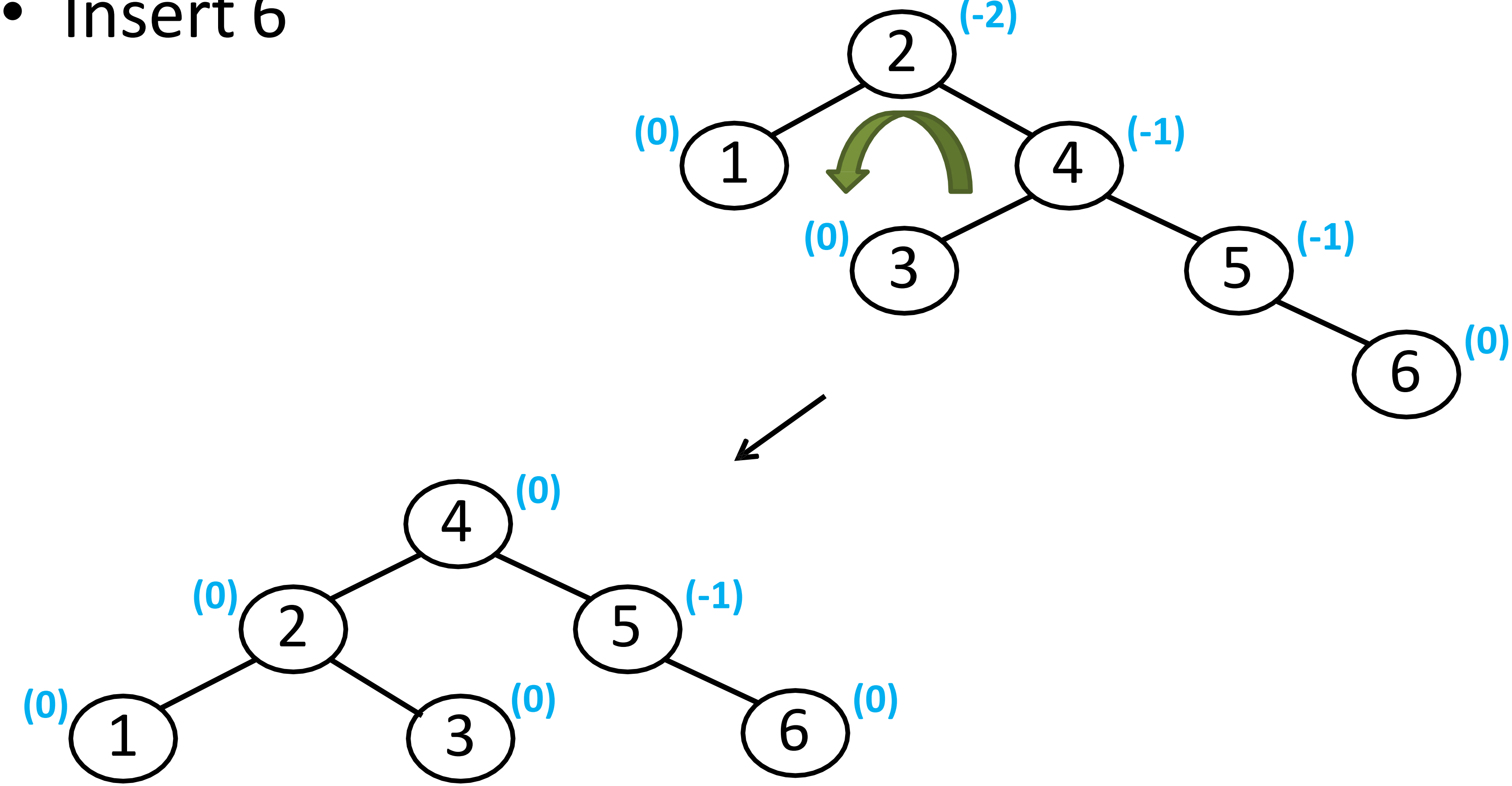



- Insert 4, 5



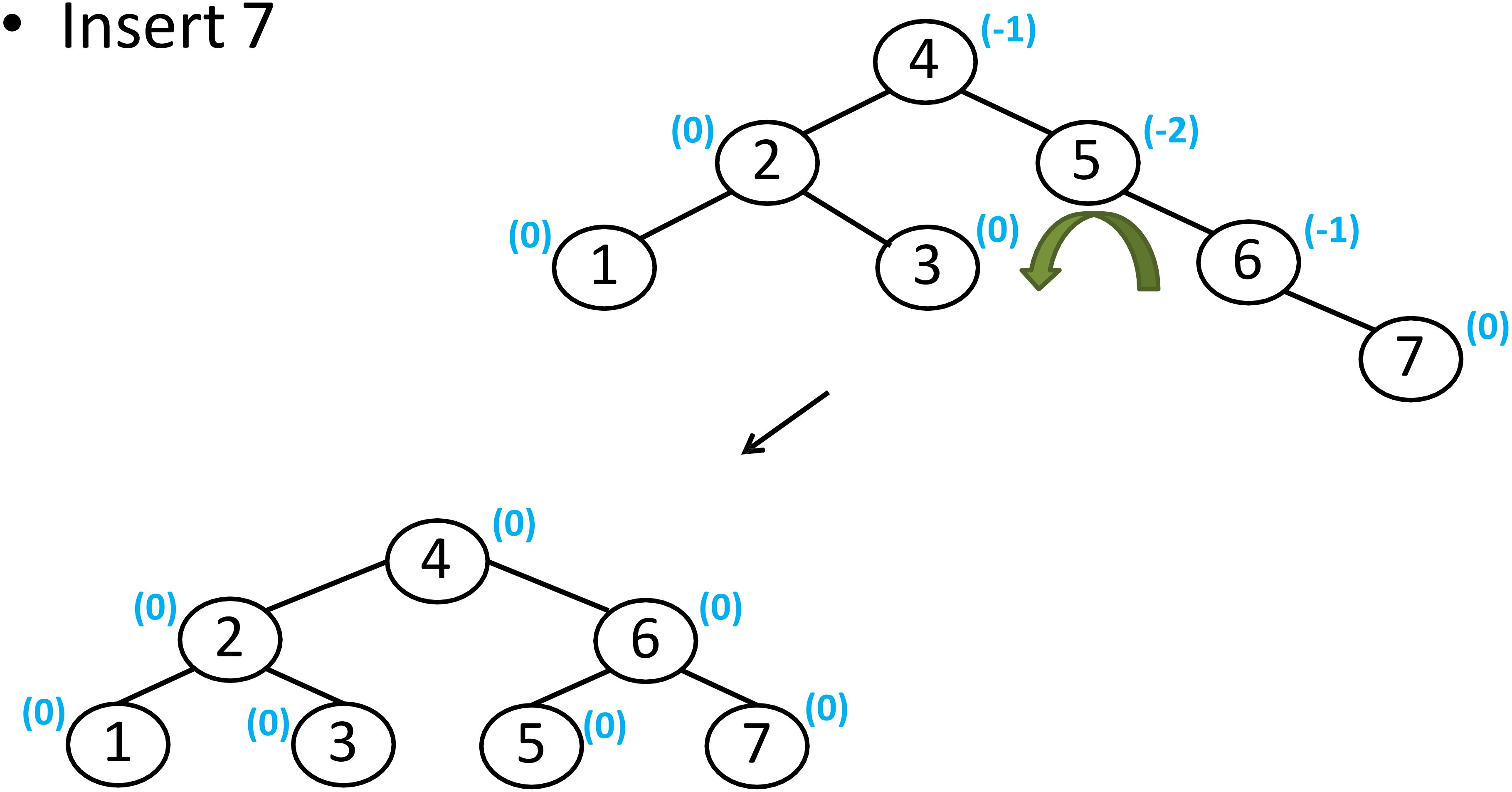
# Contd…

- Insert 6



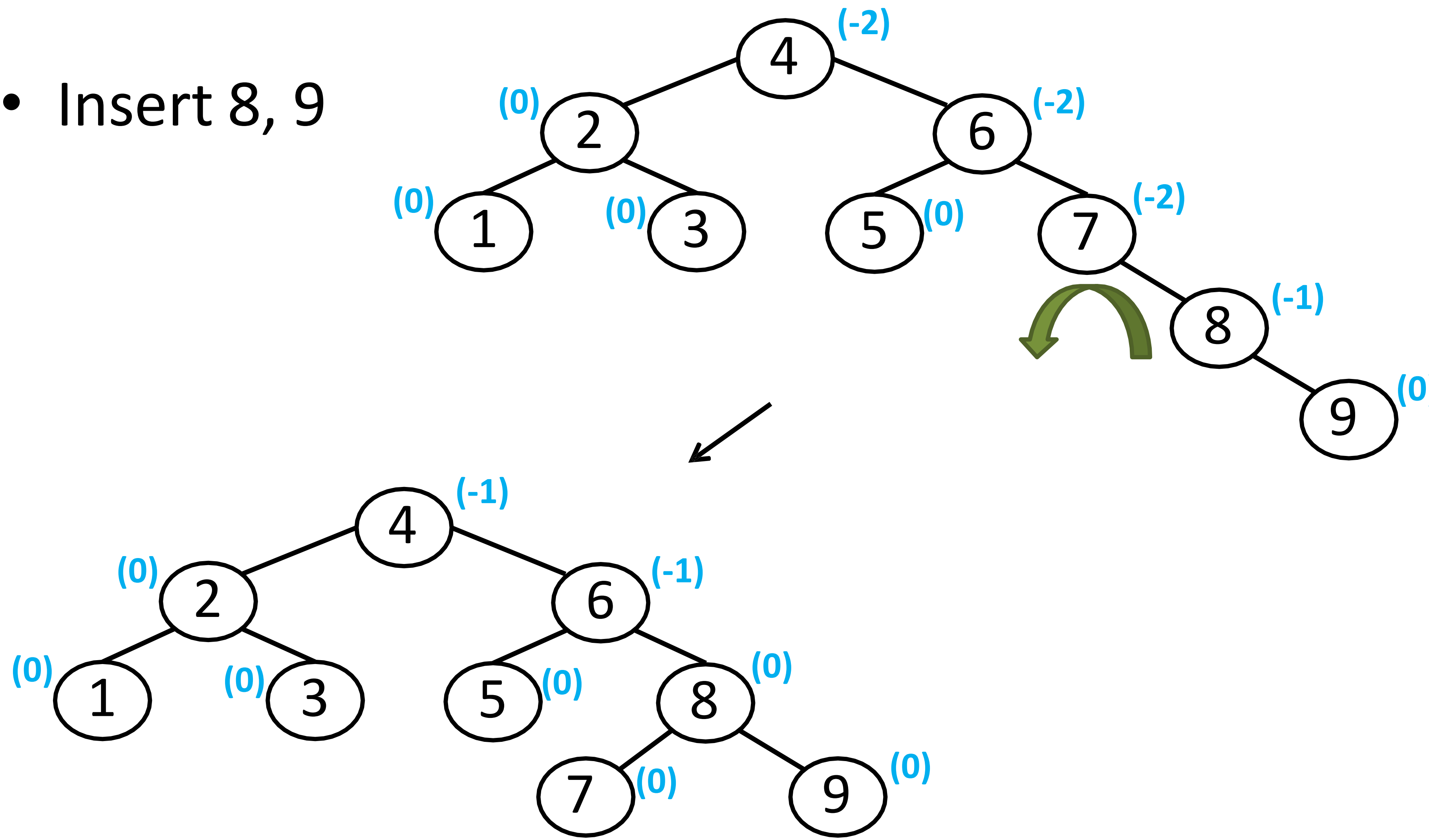
# Contd…

- Insert 7



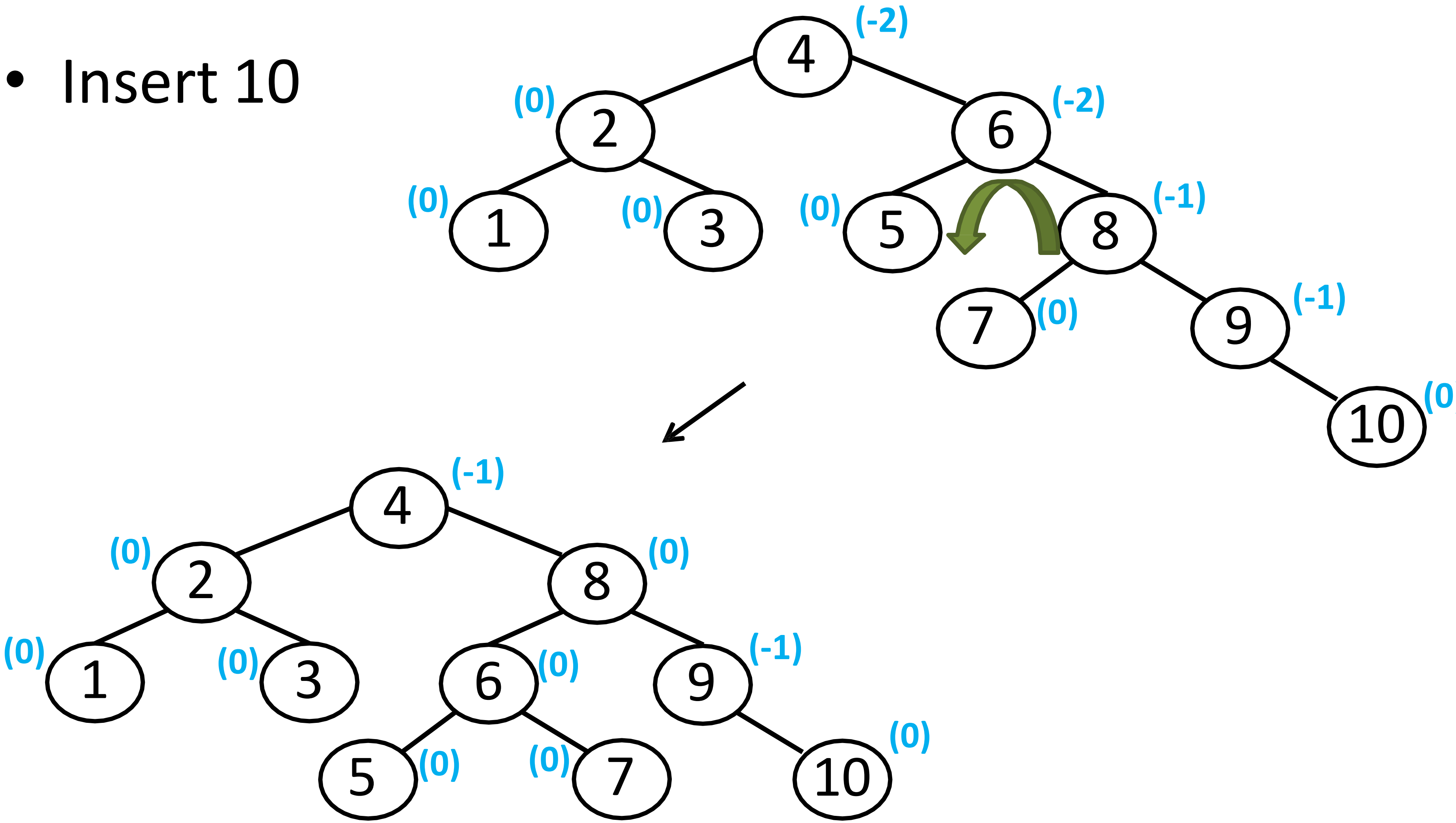
# Contd…

- Insert 8, 9



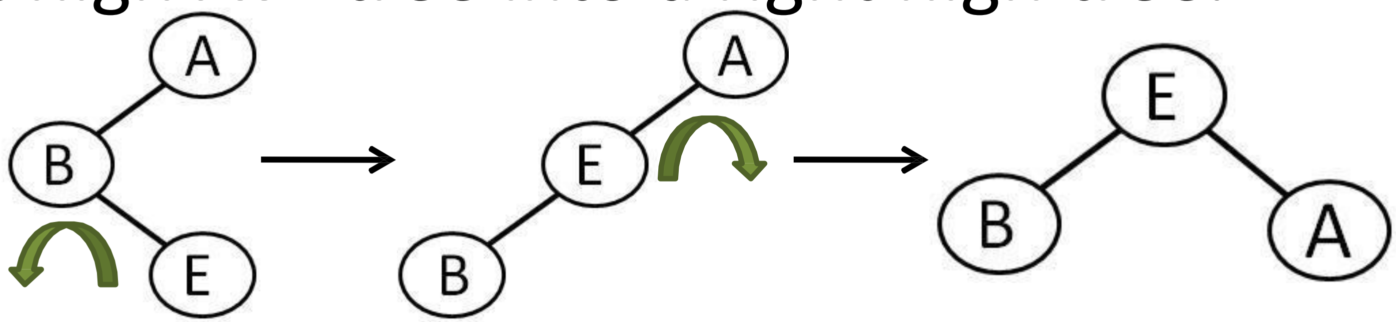
# Contd…

- Insert 10

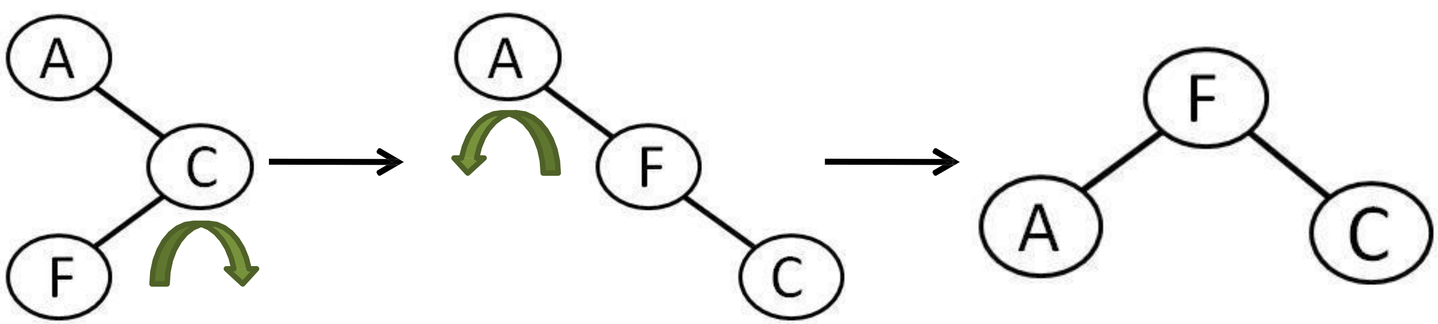


# Unbalanced Cases

- Double rotation
  - **Right of Left**: insertion turned the left subtree of a left high AVL tree into a right high tree.

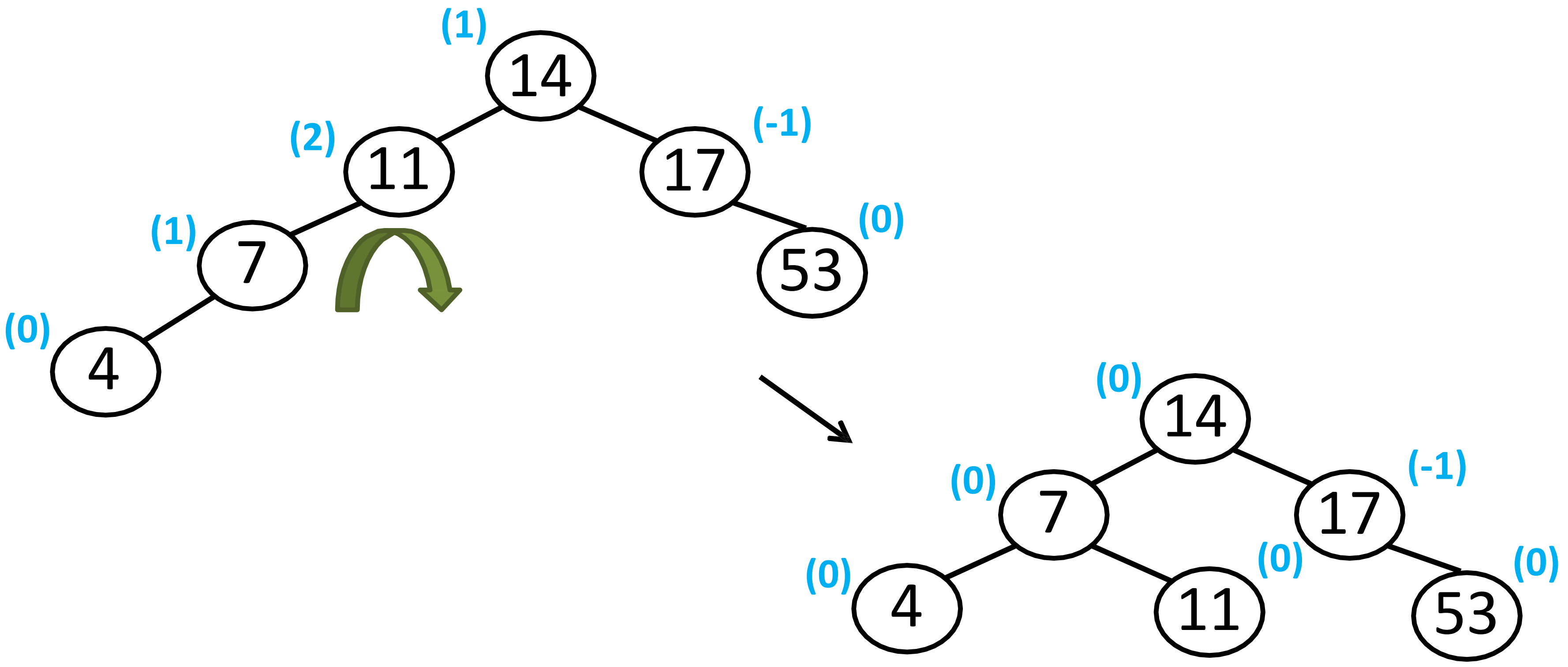



  - **Left of Right**: insertion turned the right subtree of a right high AVL tree into a left high tree.



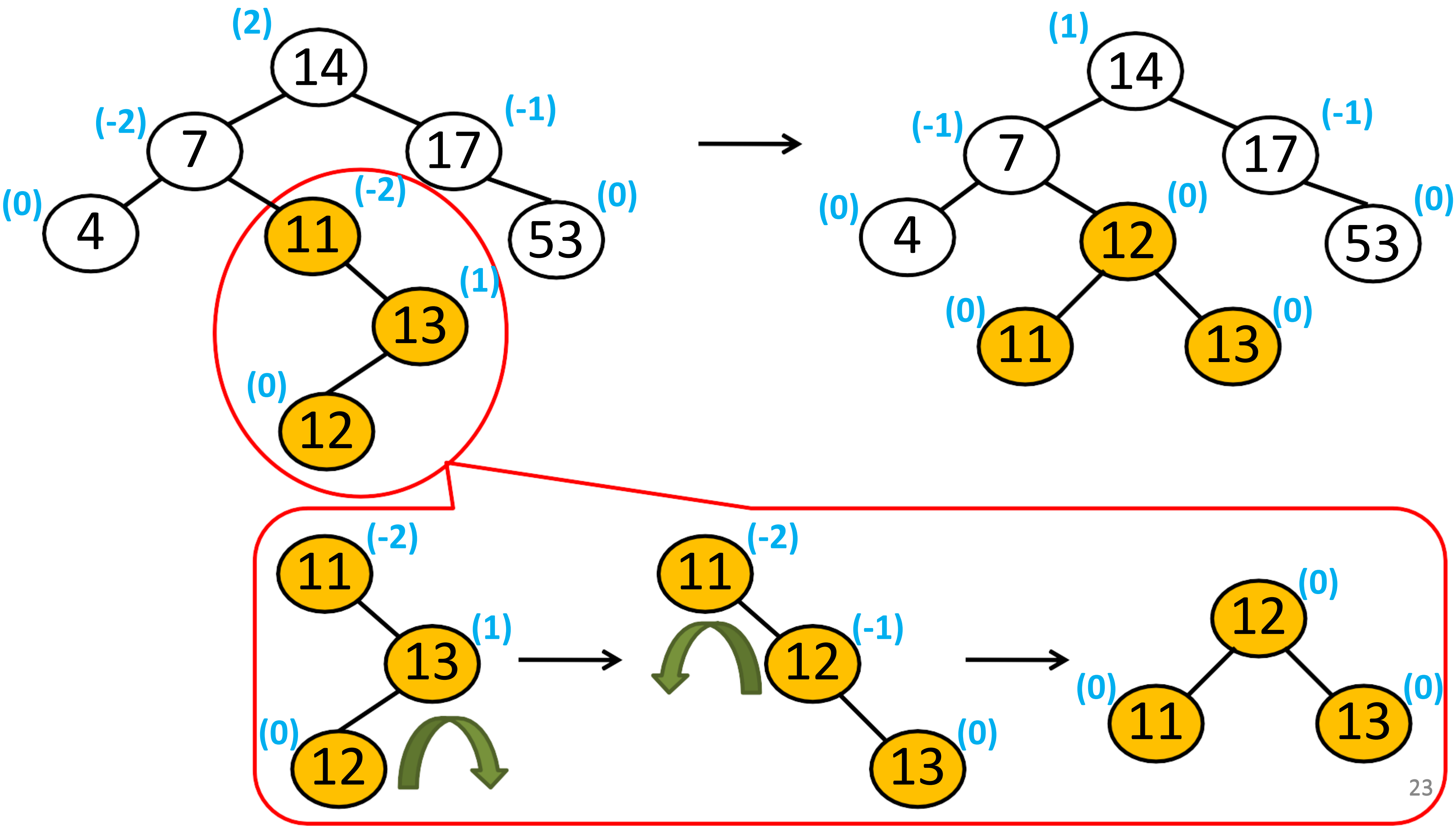
# Example 3: Insert 14, 17, 11, 7, 53, 4, 13, 12, 8

- Insert 14, 17, 11,7, 53, 4



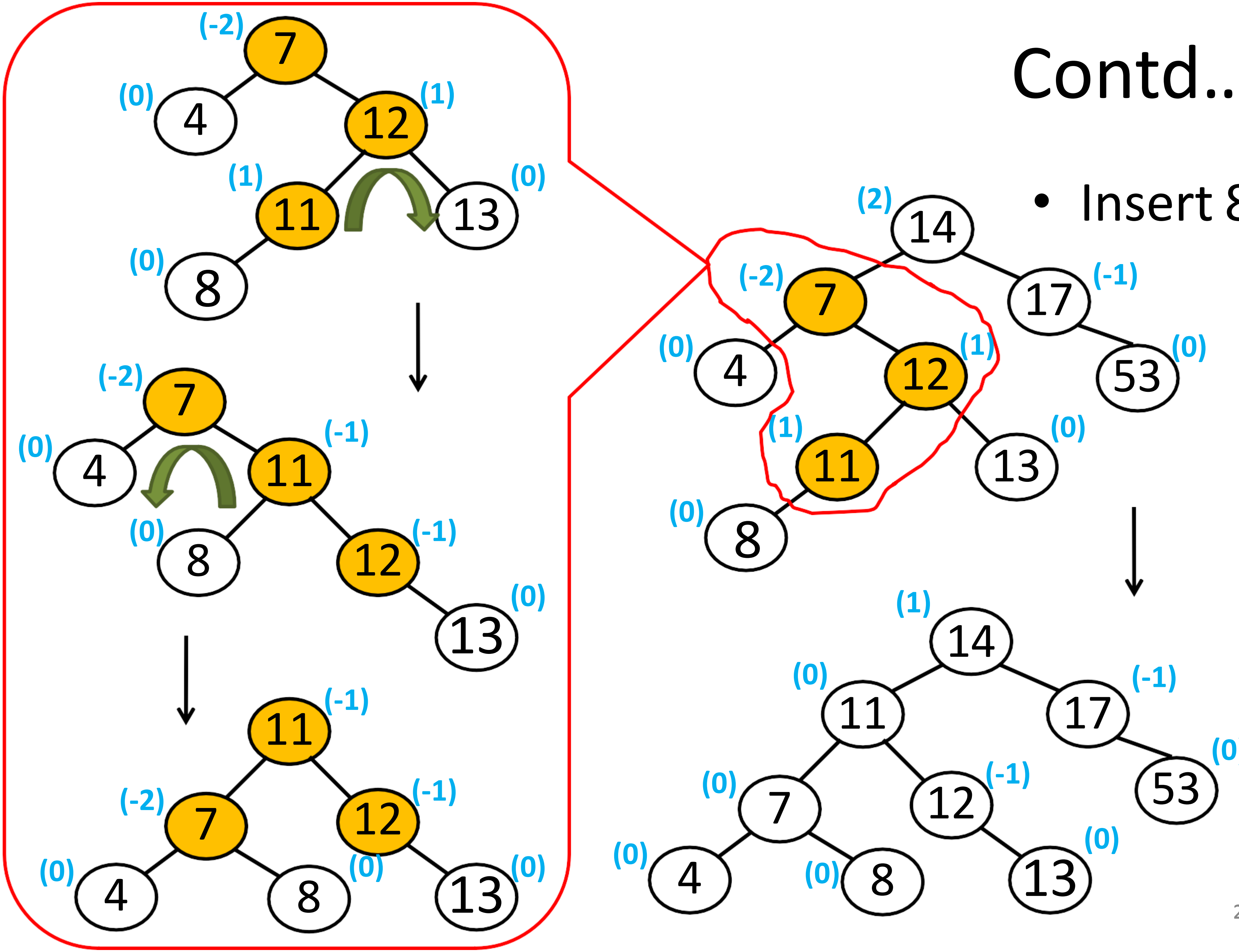
# Contd…

- Insert 13, 12



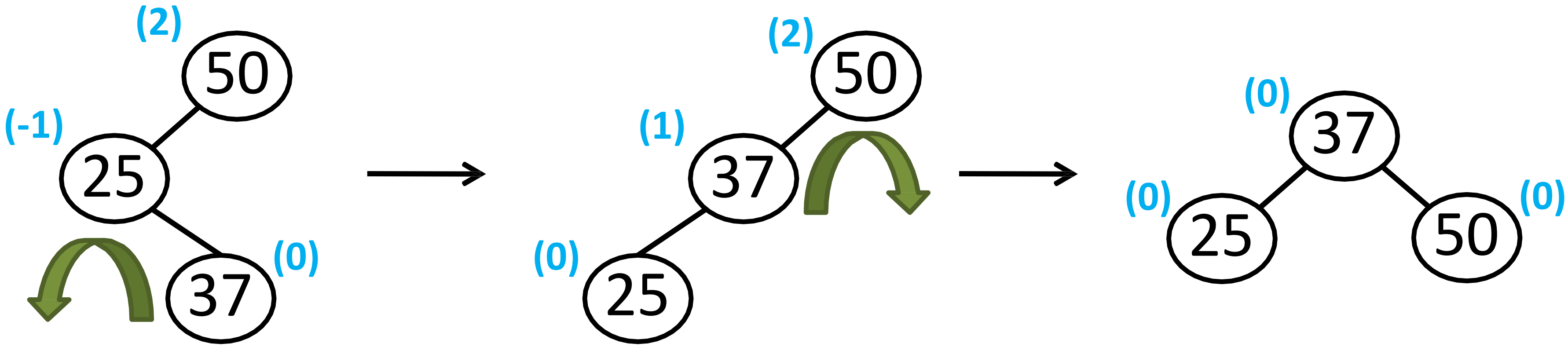
# Contd...

- Insert 8

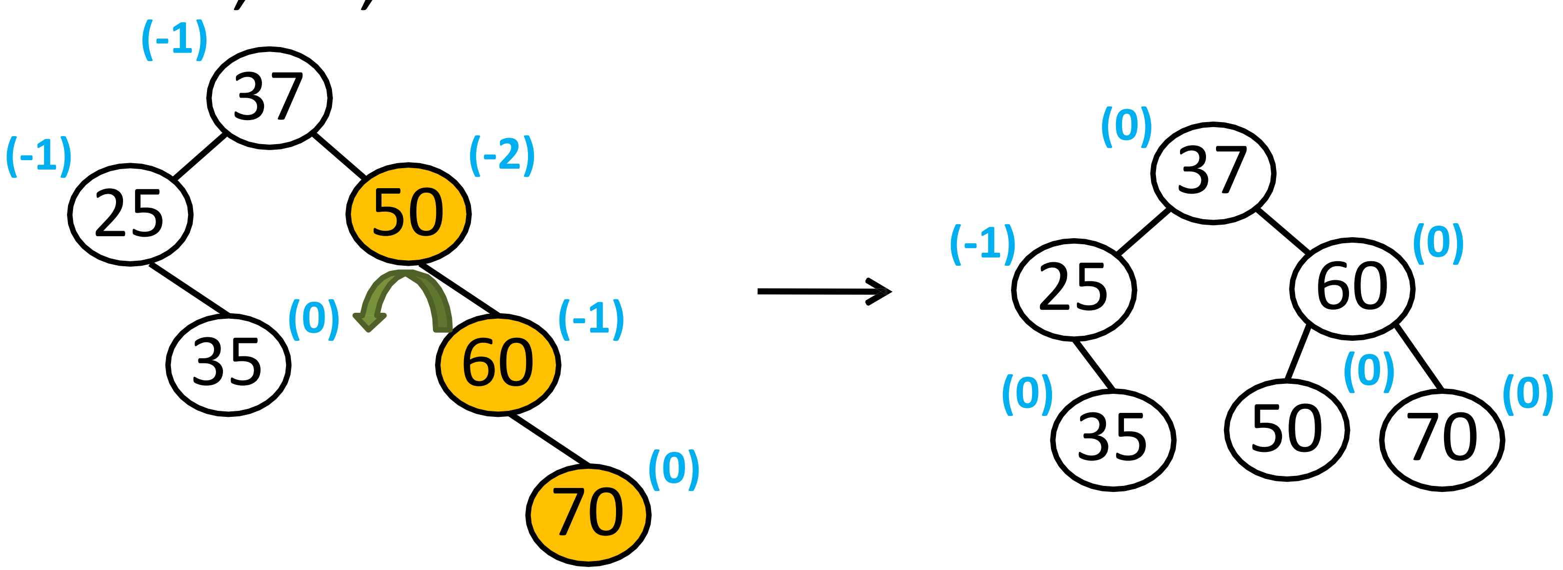


# Example 4: 50, 25, 37, 35, 60, 70, 30, 45, 34, 40, 55

- Insert 50, 25, 37

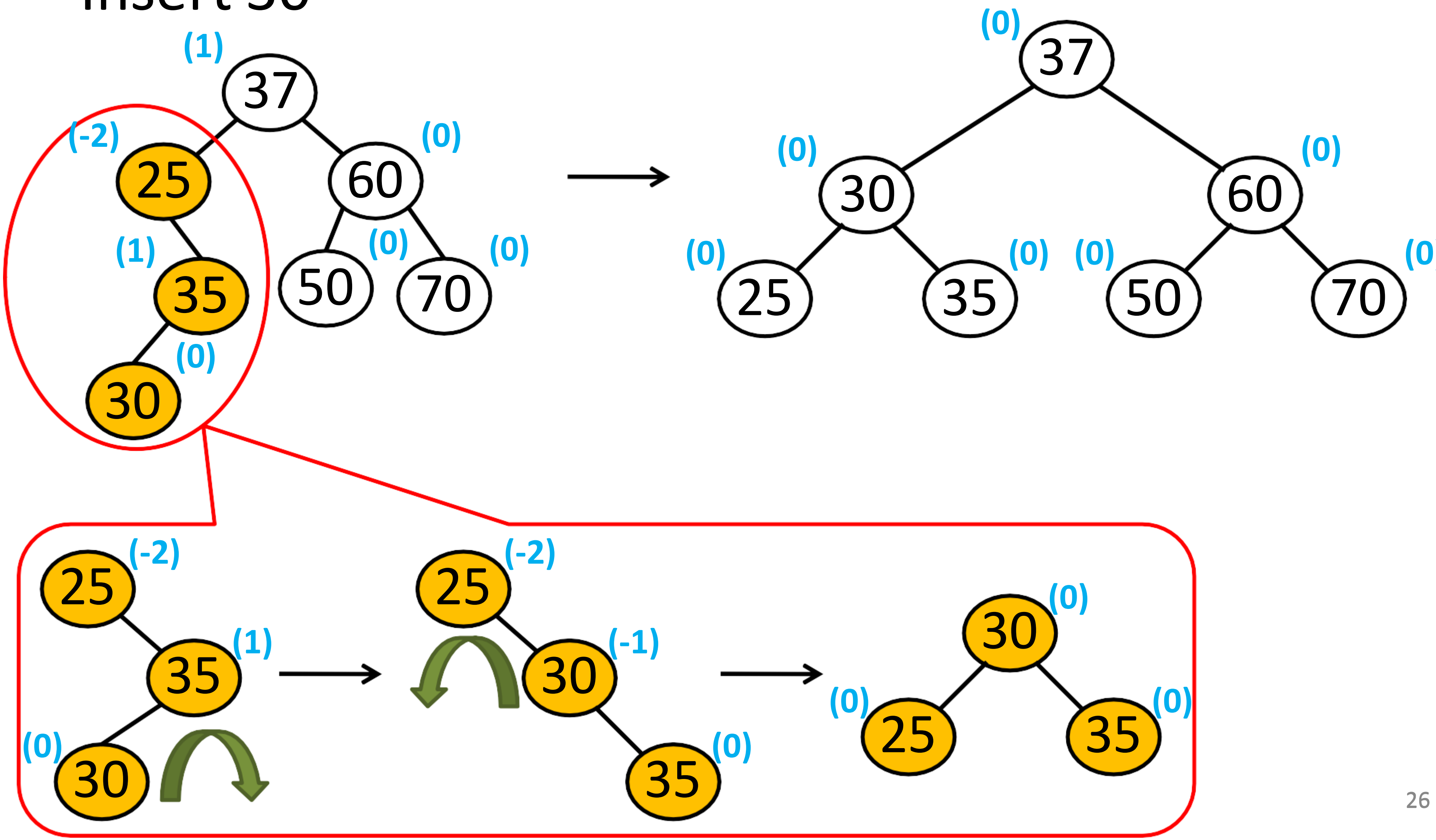



- Insert 35, 60, 70



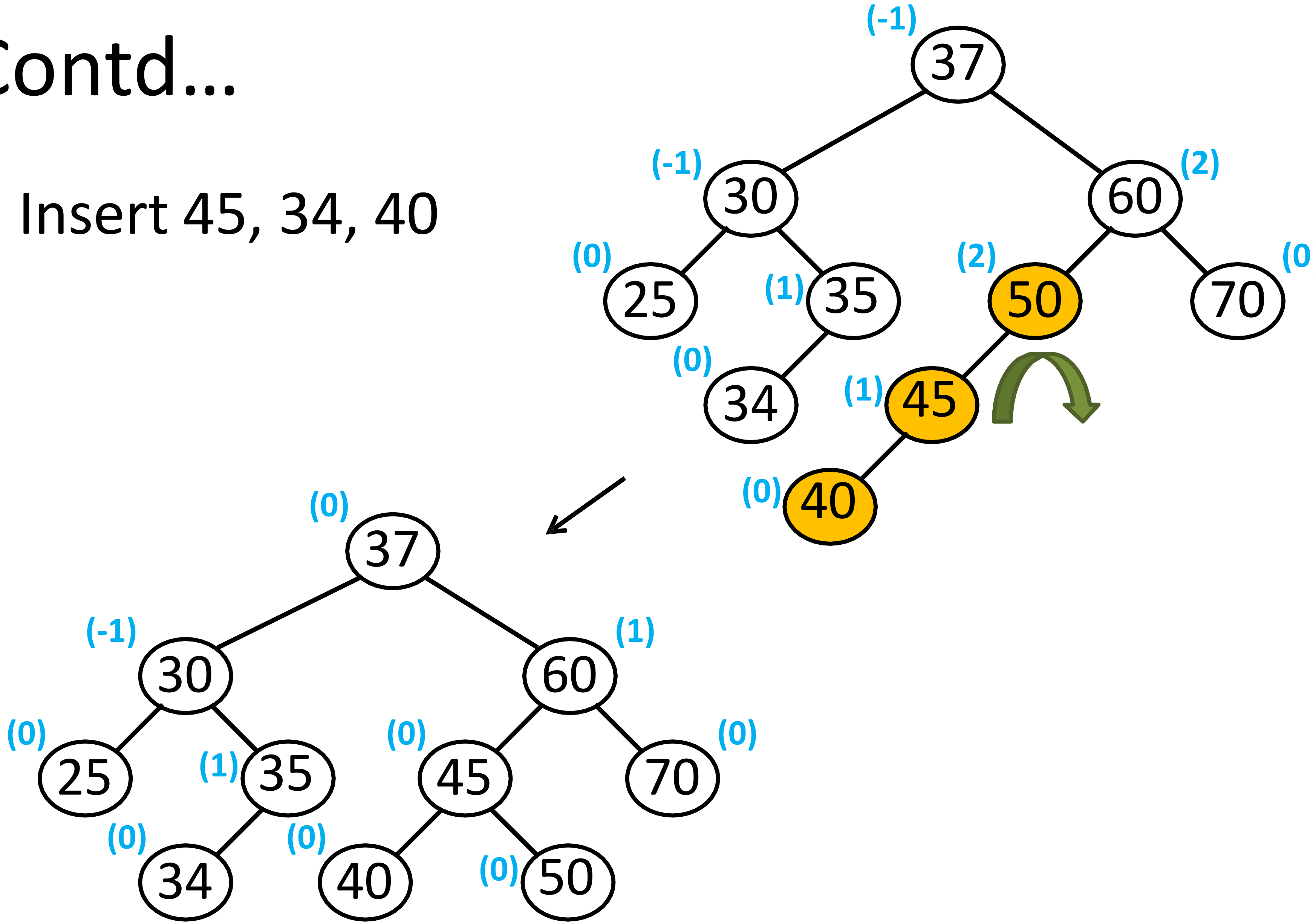
# Contd…

- Insert 30



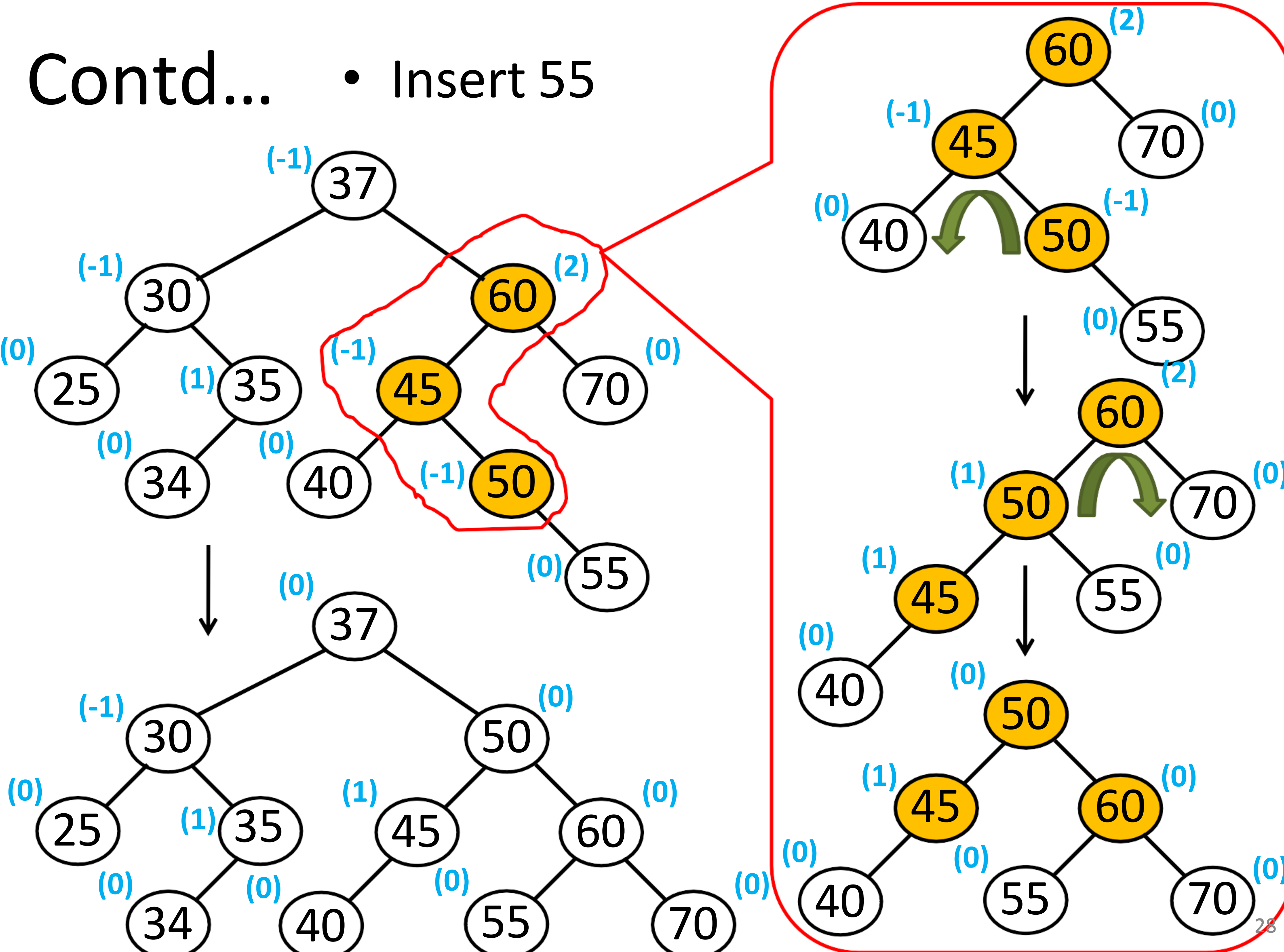
# Contd…

- Insert 45, 34, 40

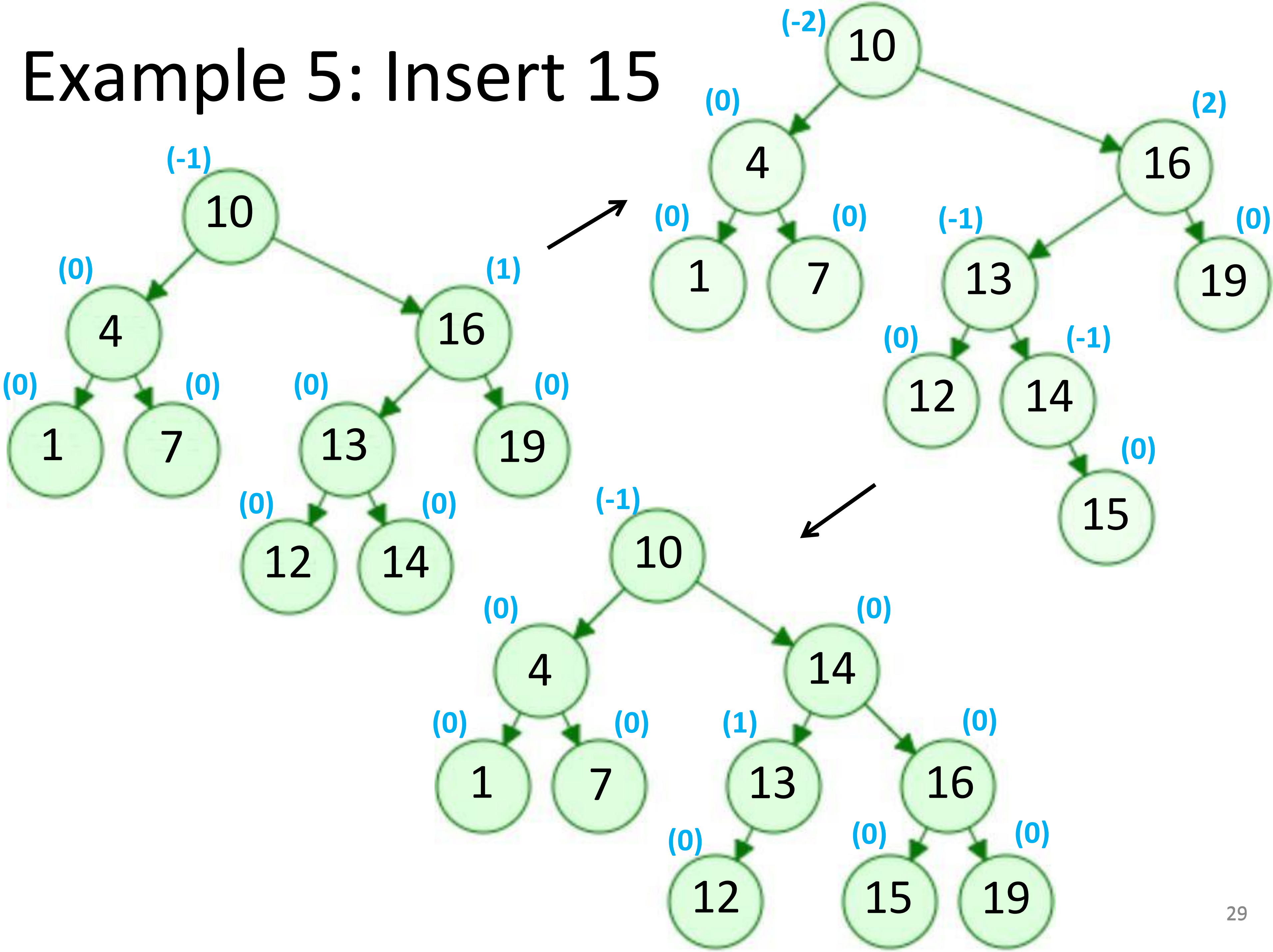


# Contd…

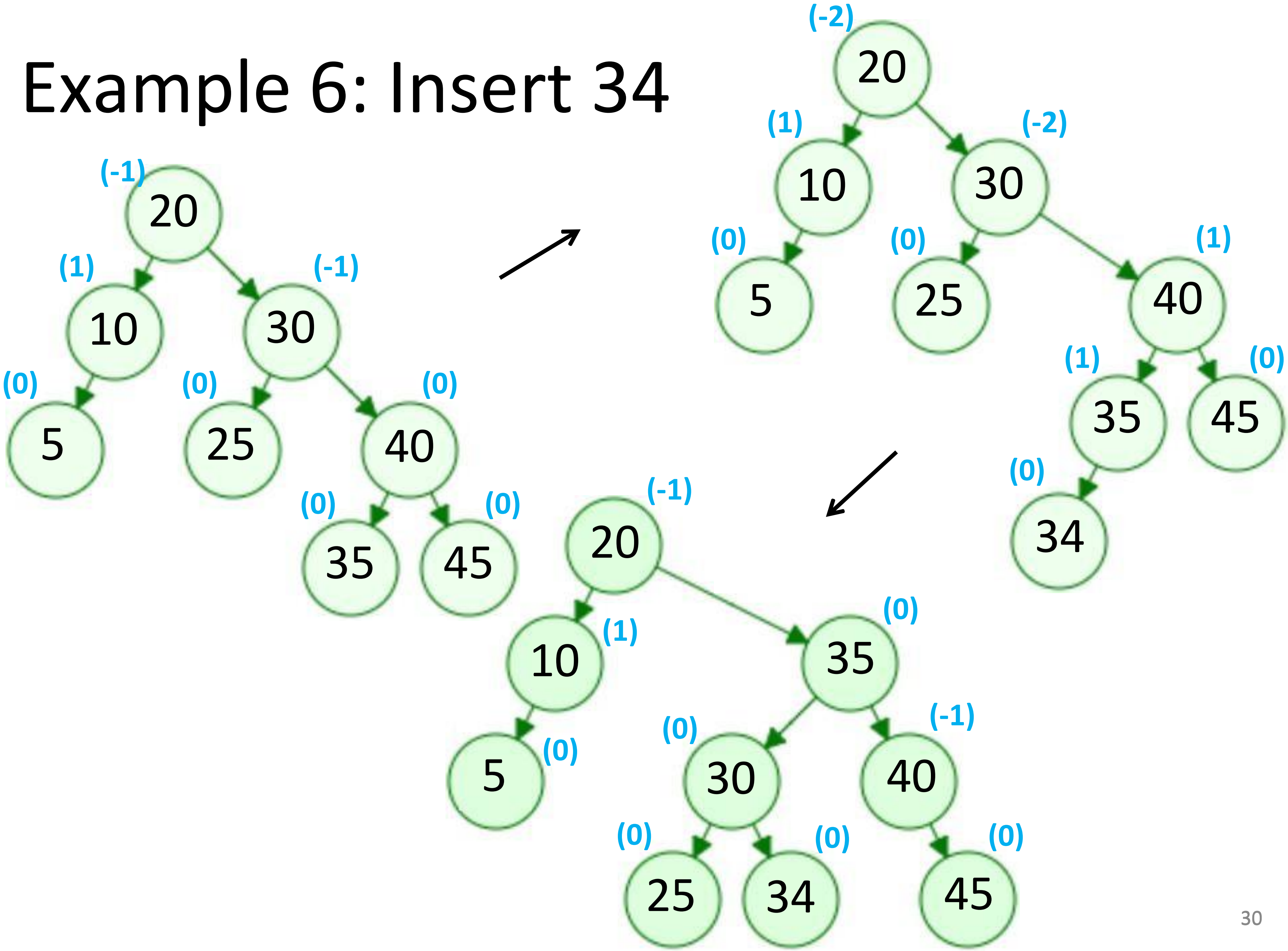
- Insert 55



# Example 5: Insert 15



# Example 6: Insert 34

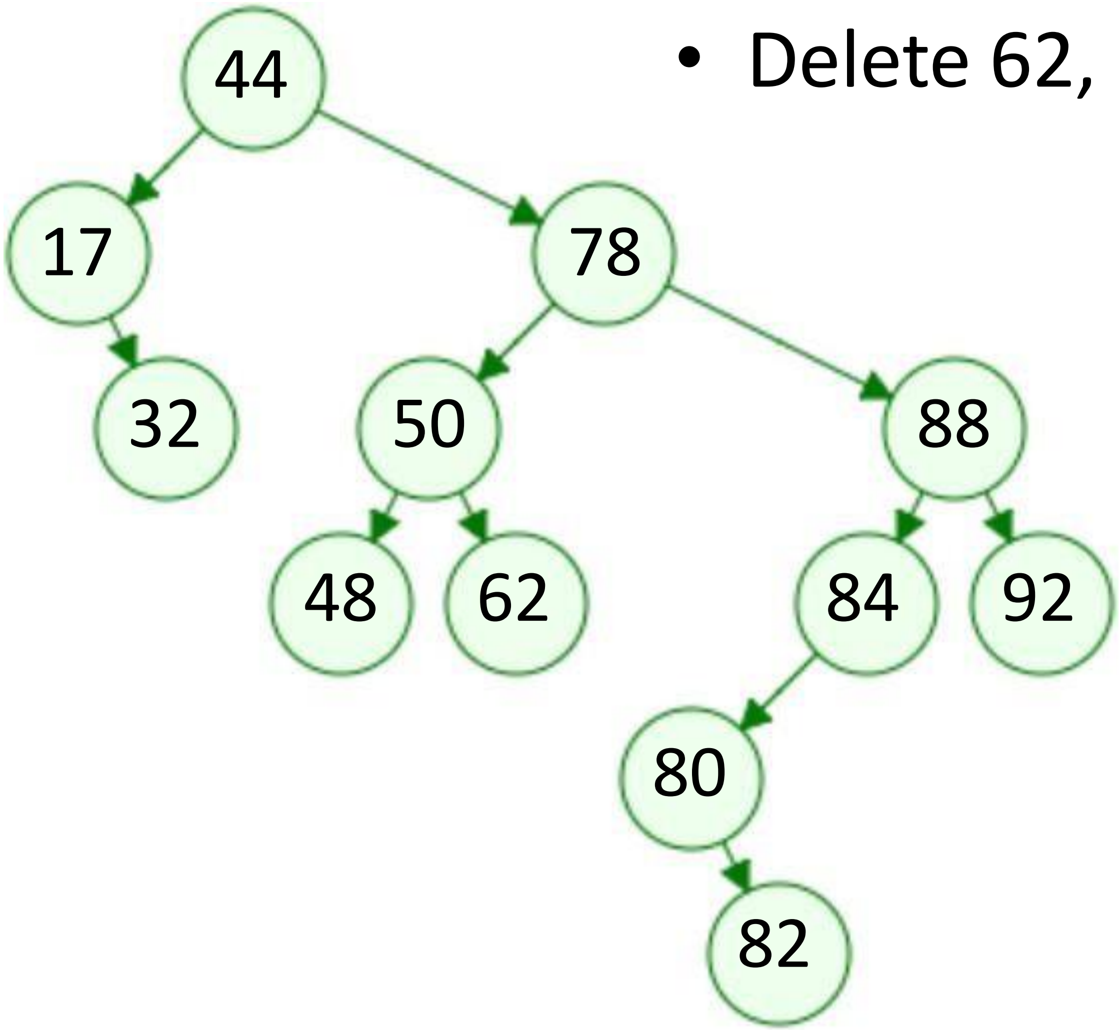


# BST Deletion

- Search for a node to remove.
- If the node is found, then there are three cases:

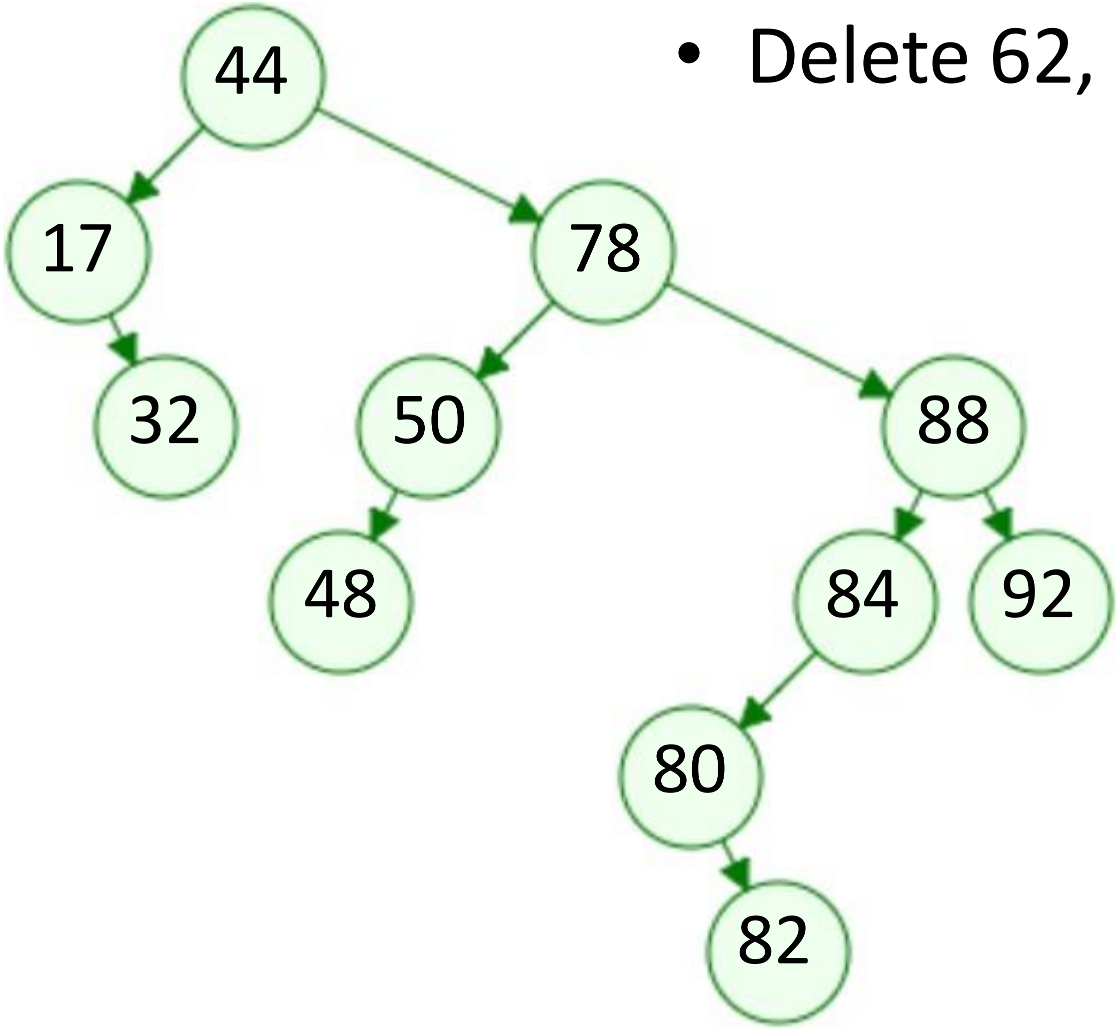
1. Node to be removed has no children.
   - Set corresponding link of the parent to NULL and dispose the node.
2. Node to be removed has one child.
   - Link single child (with it's subtree) directly to the parent of the removed node.
3. Node to be removed has two children.
   - Find inorder successor of the node.
   - Copy contents of the inorder successor to the node being removed.
   - Delete the inorder successor from the right subtree.

# Example – BST Deletion



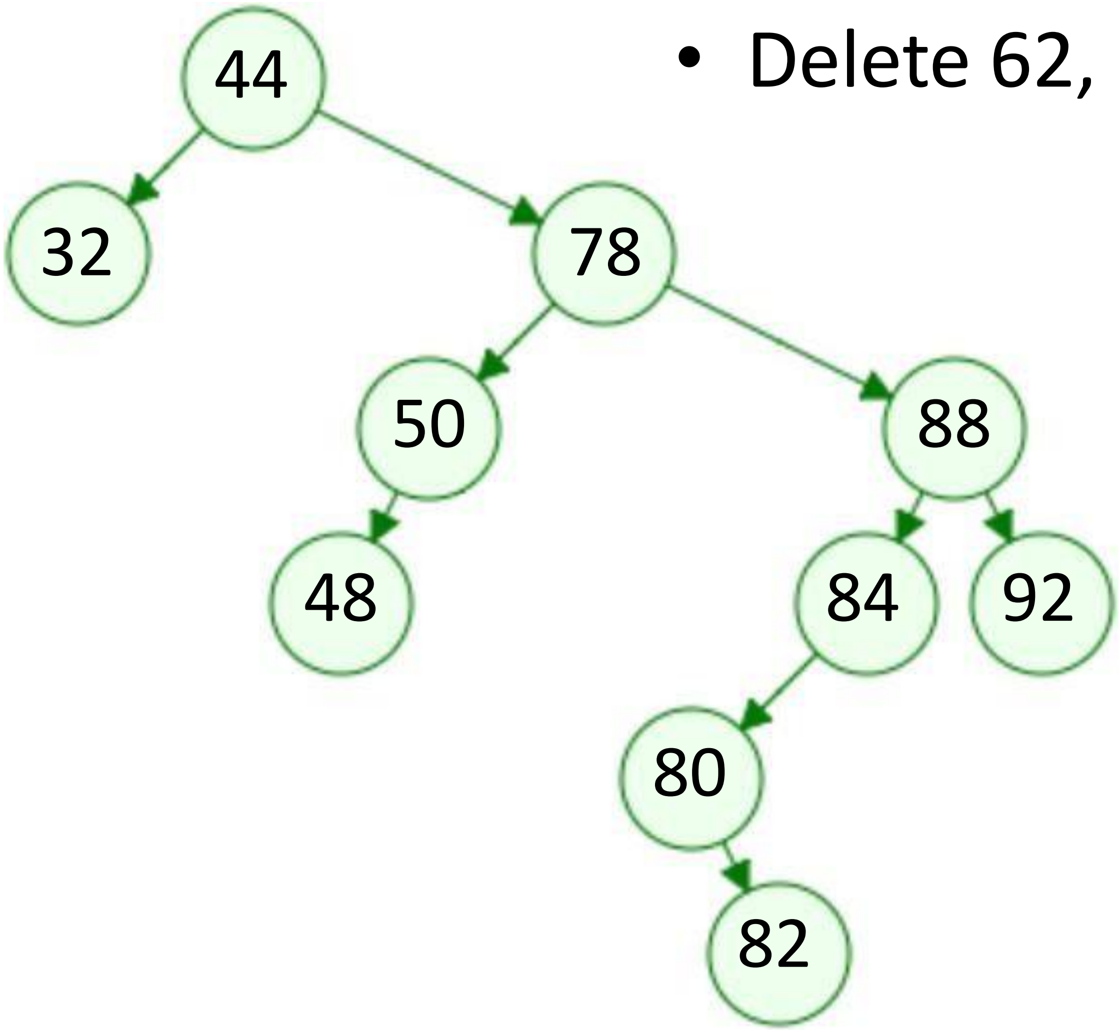

- Delete 62, 17, and 78

# After deleting 62



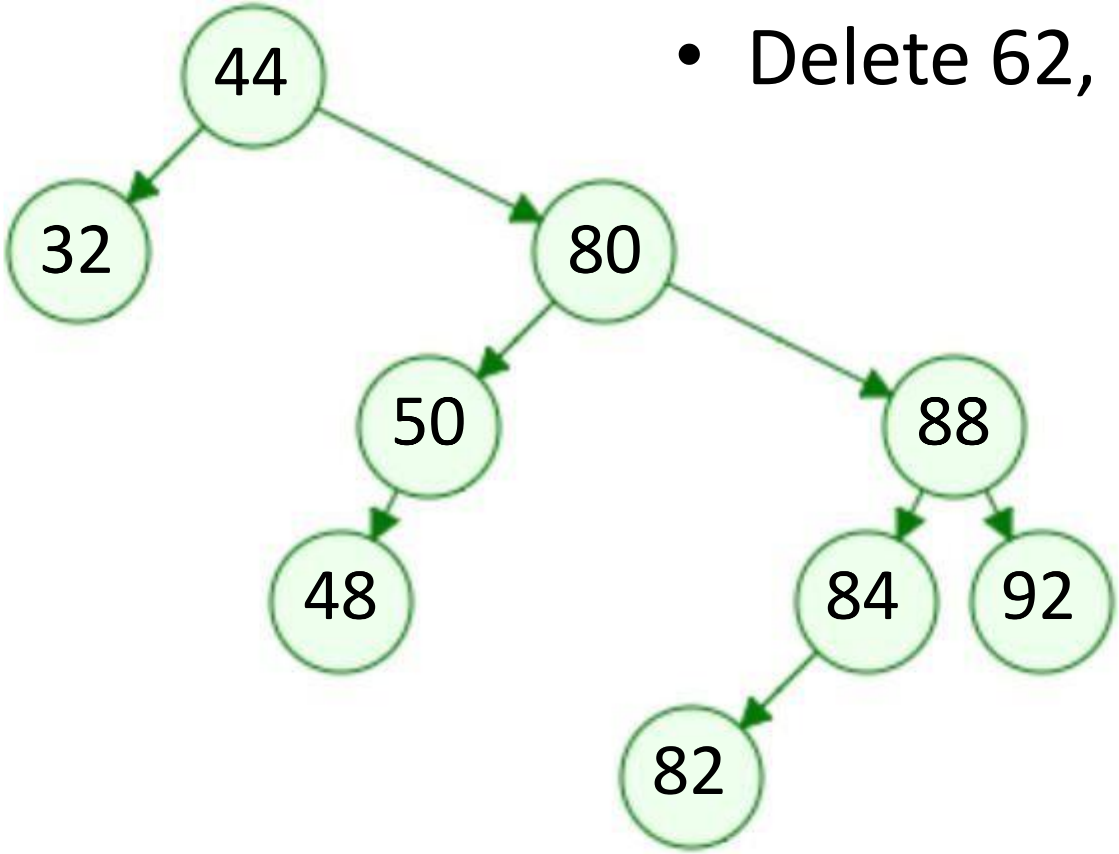

- Delete 62, 17, and 78

# After deleting 17




- Delete 62, 17, and 78

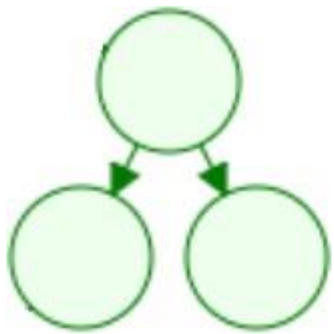
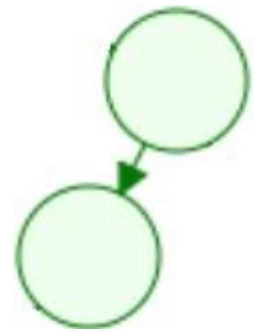
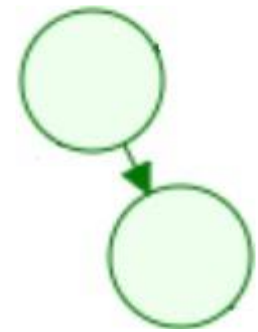
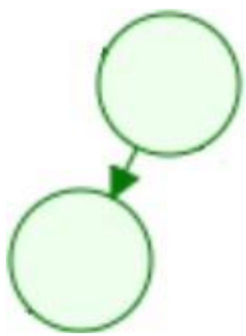
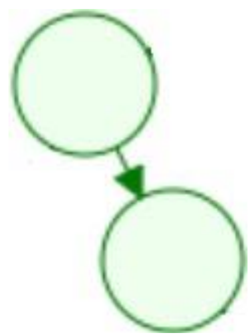
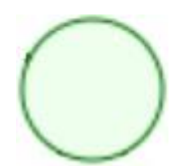
# After deleting 78




- Delete 62, 17, and 78

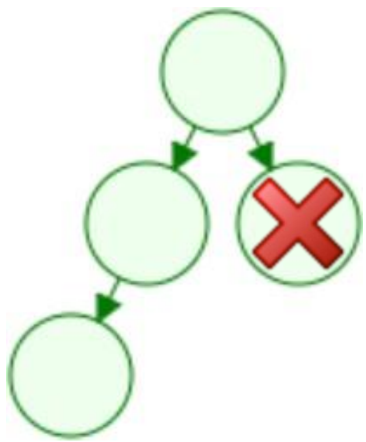
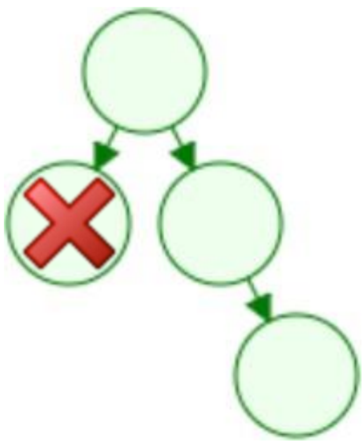
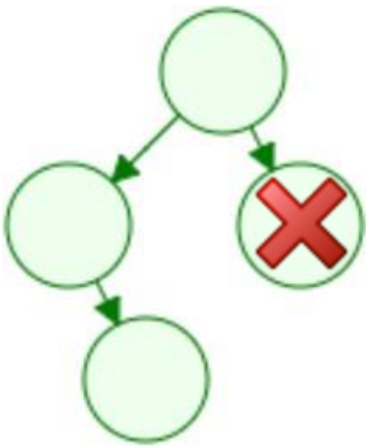
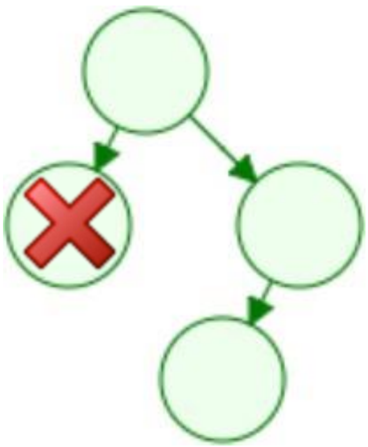
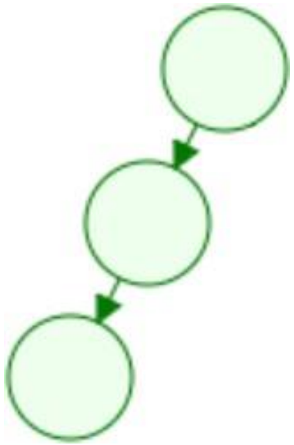
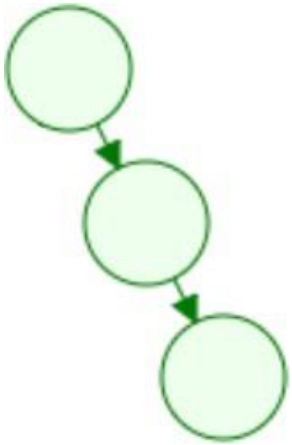
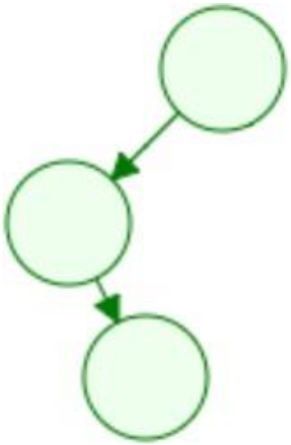
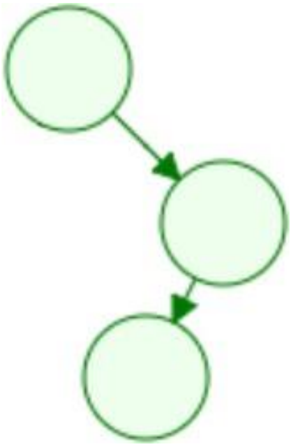
# AVL Deletion

**REQUIRES NO ROTATION**

| | Before deletion | After deletion |
|---|---|---|
| • Case 1 |  |  OR  |
| • Case 2 |  OR  |  |

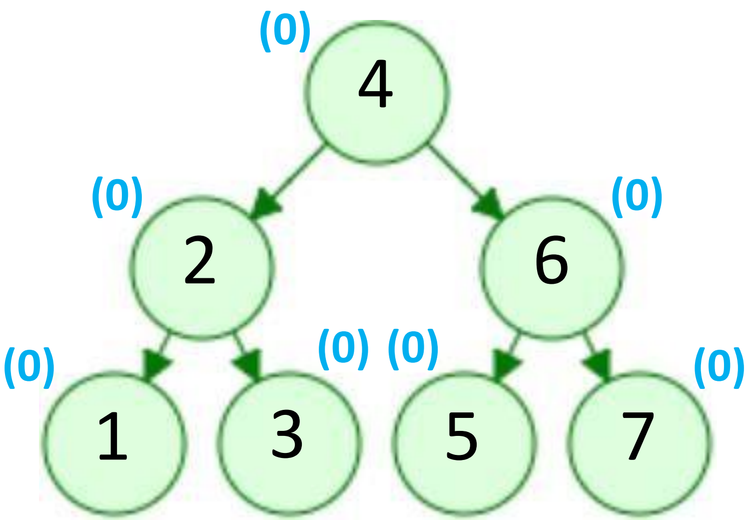
# Contd…

- Case 3

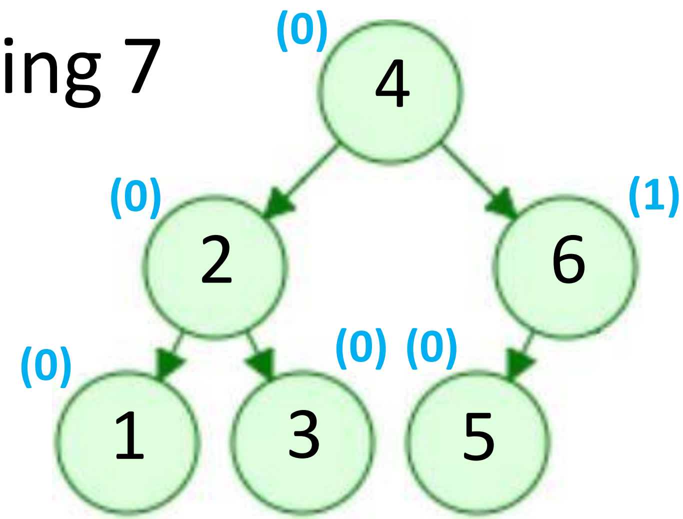
| | | | | |
|---|---|---|---|---|
| Before deletion |  |  |  |  |
| After deletion |  |  |  |  |
| **Rotations** | **Right** | **Left** | **Left then Right** | **Right then Left** |

# Example – 1

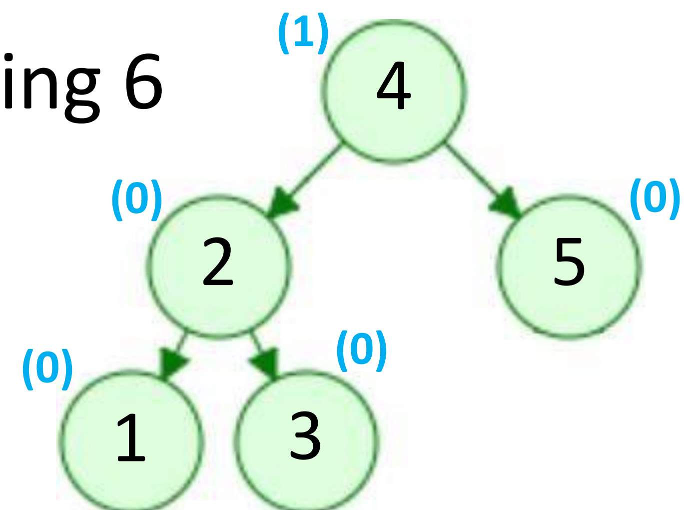
- Delete 7, 6, 5




After deleting 7

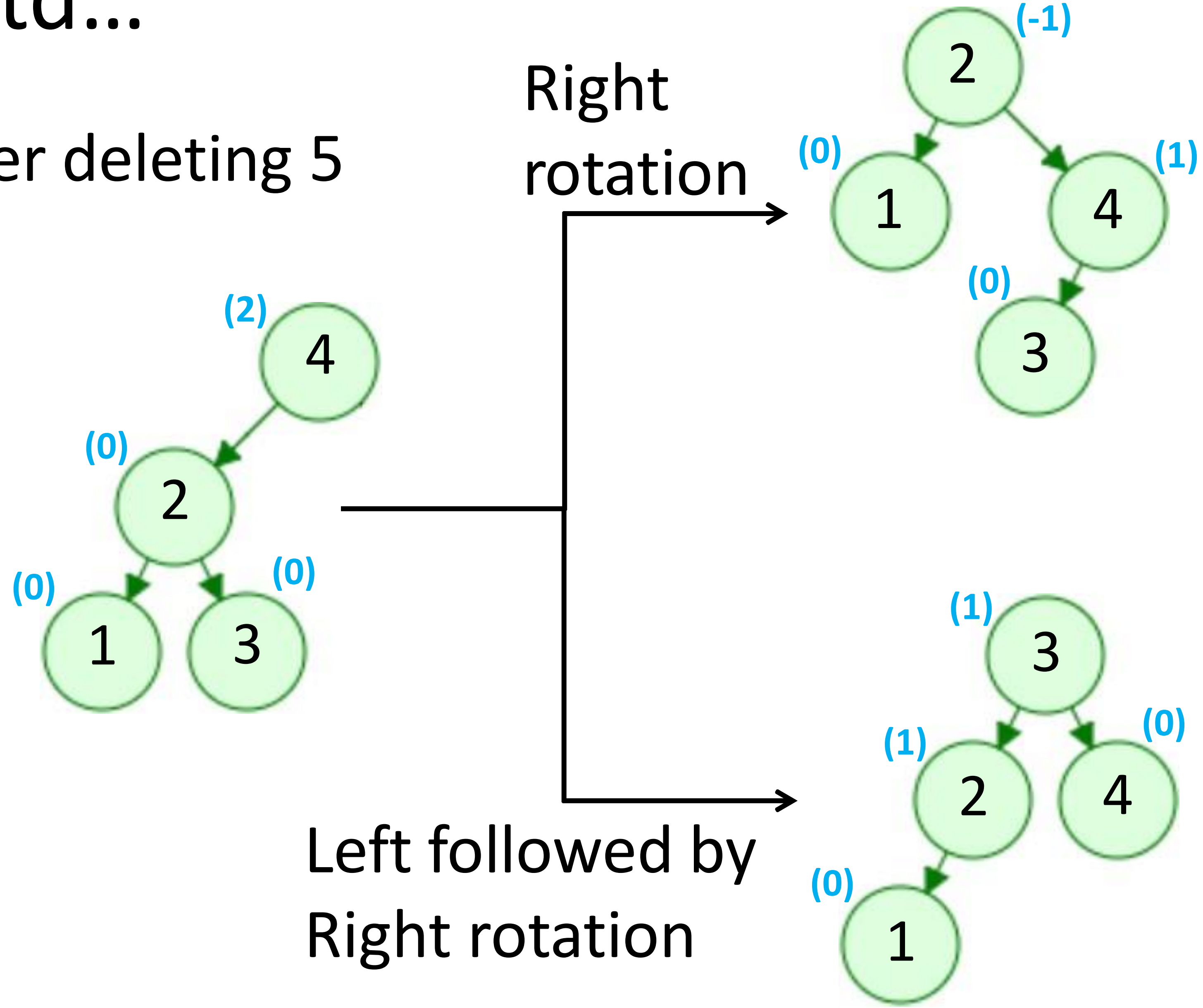



After deleting 6



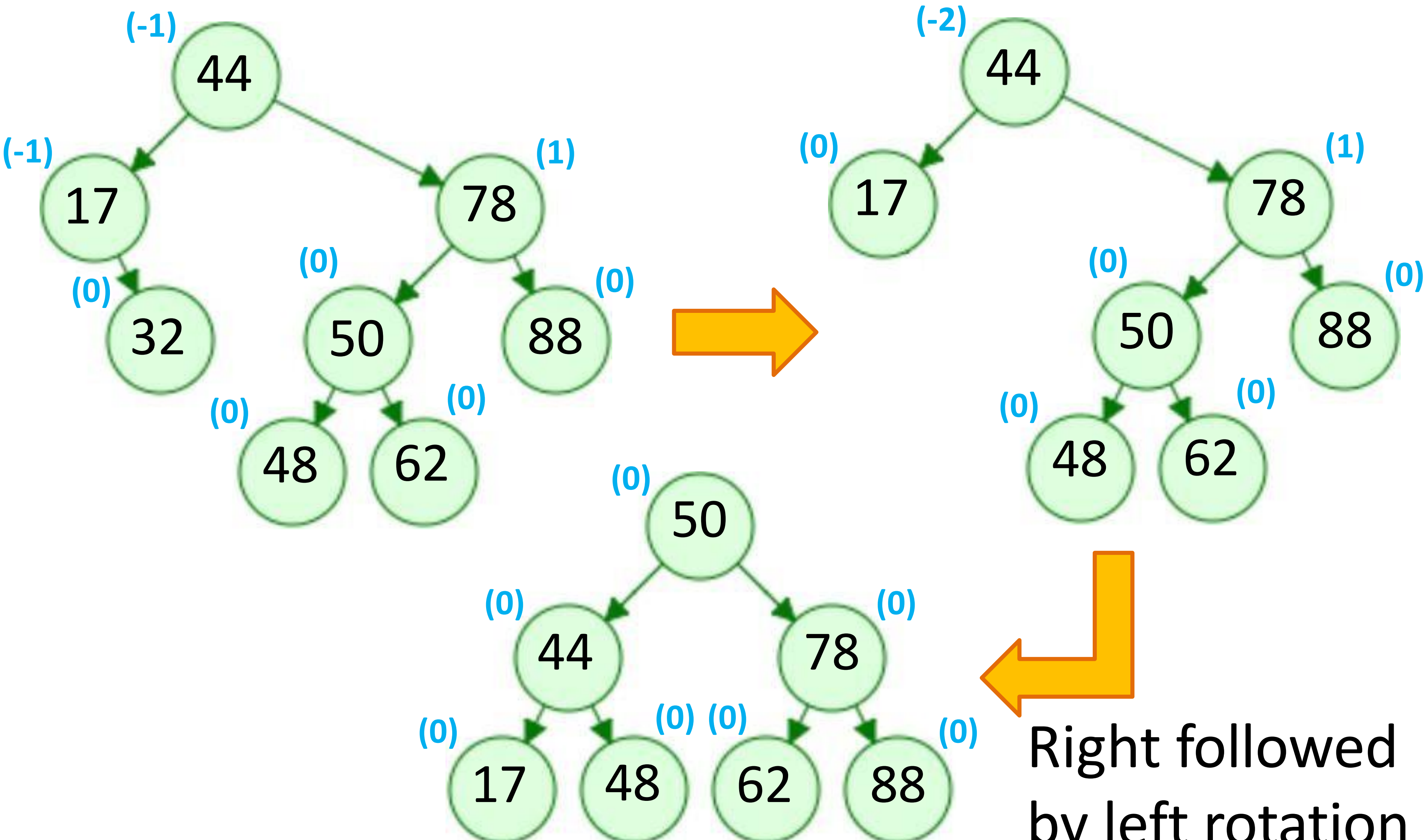
# Contd…

- After deleting 5



# Example – 2

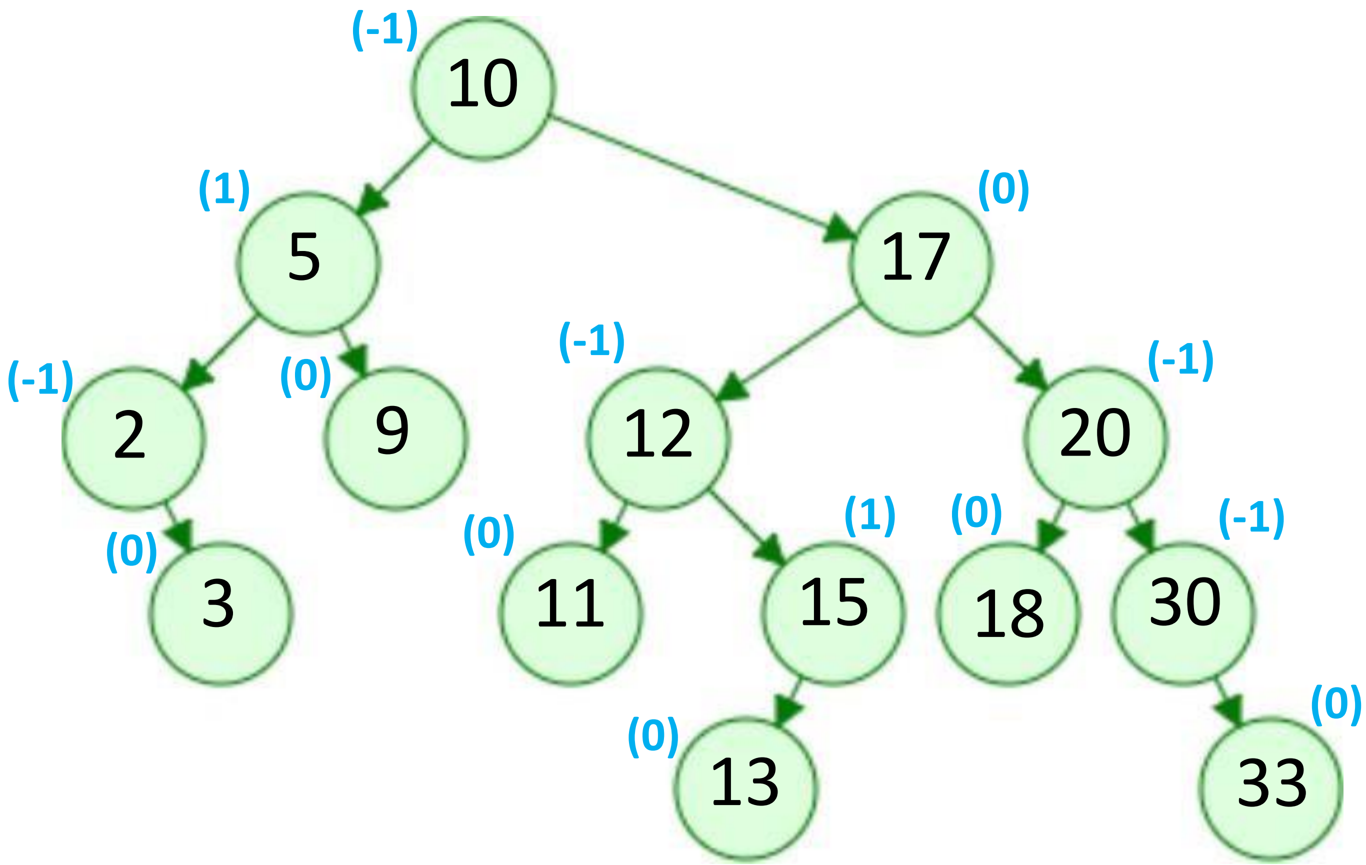
# Delete 32



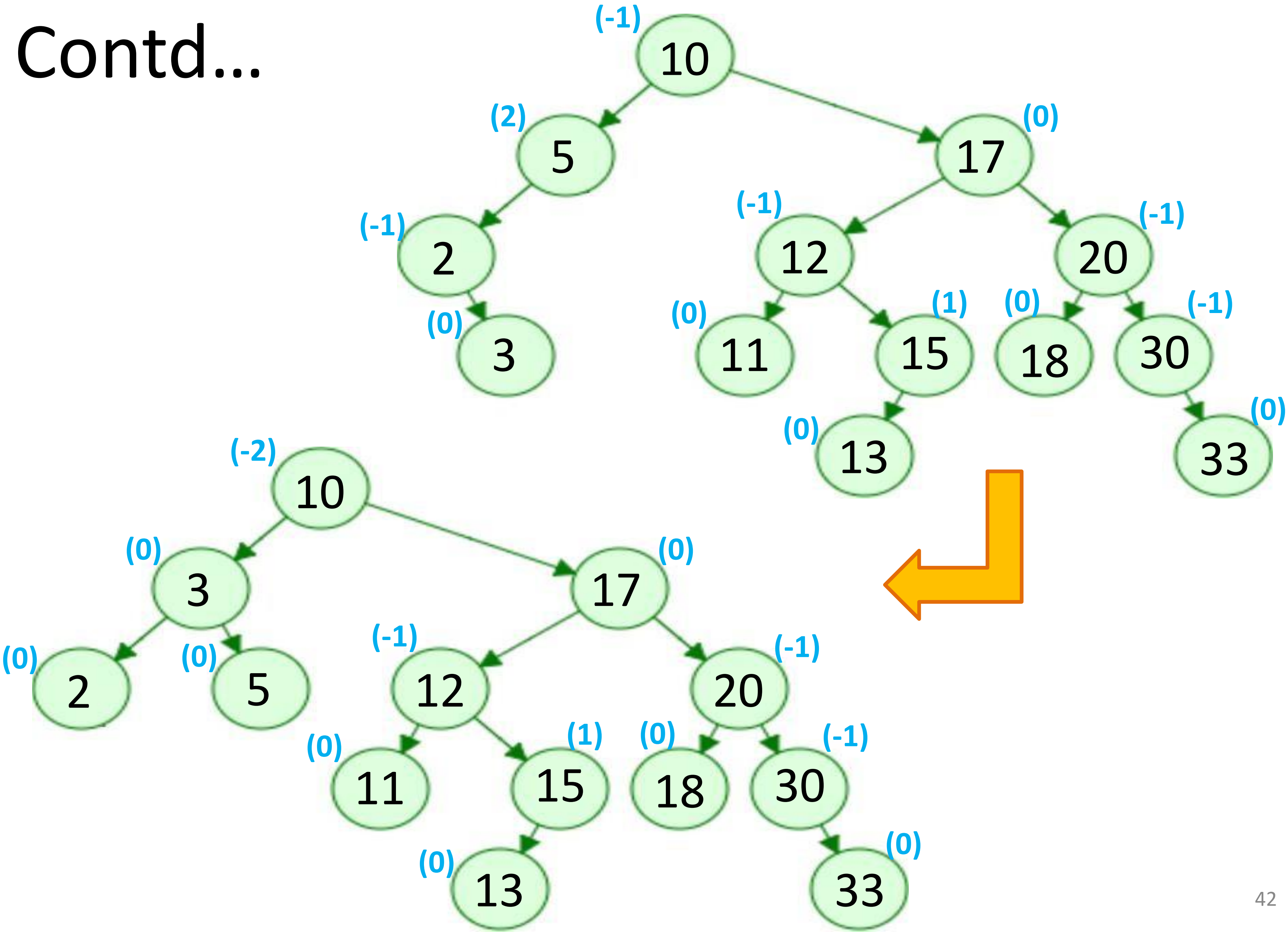

Right followed by left rotation

# Example – 3

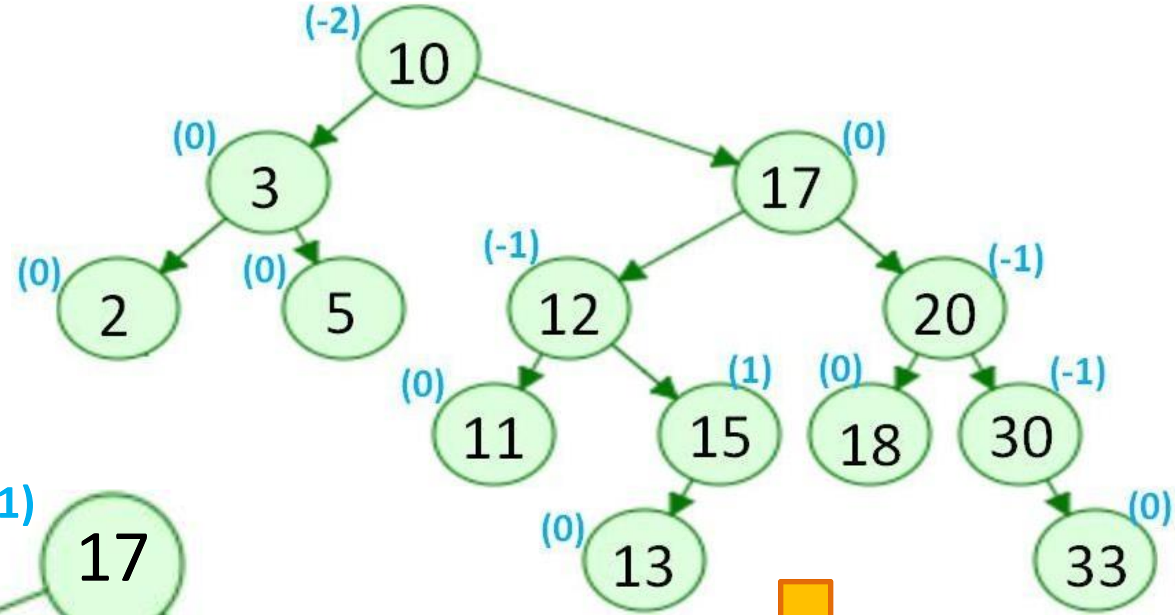
- Delete 9



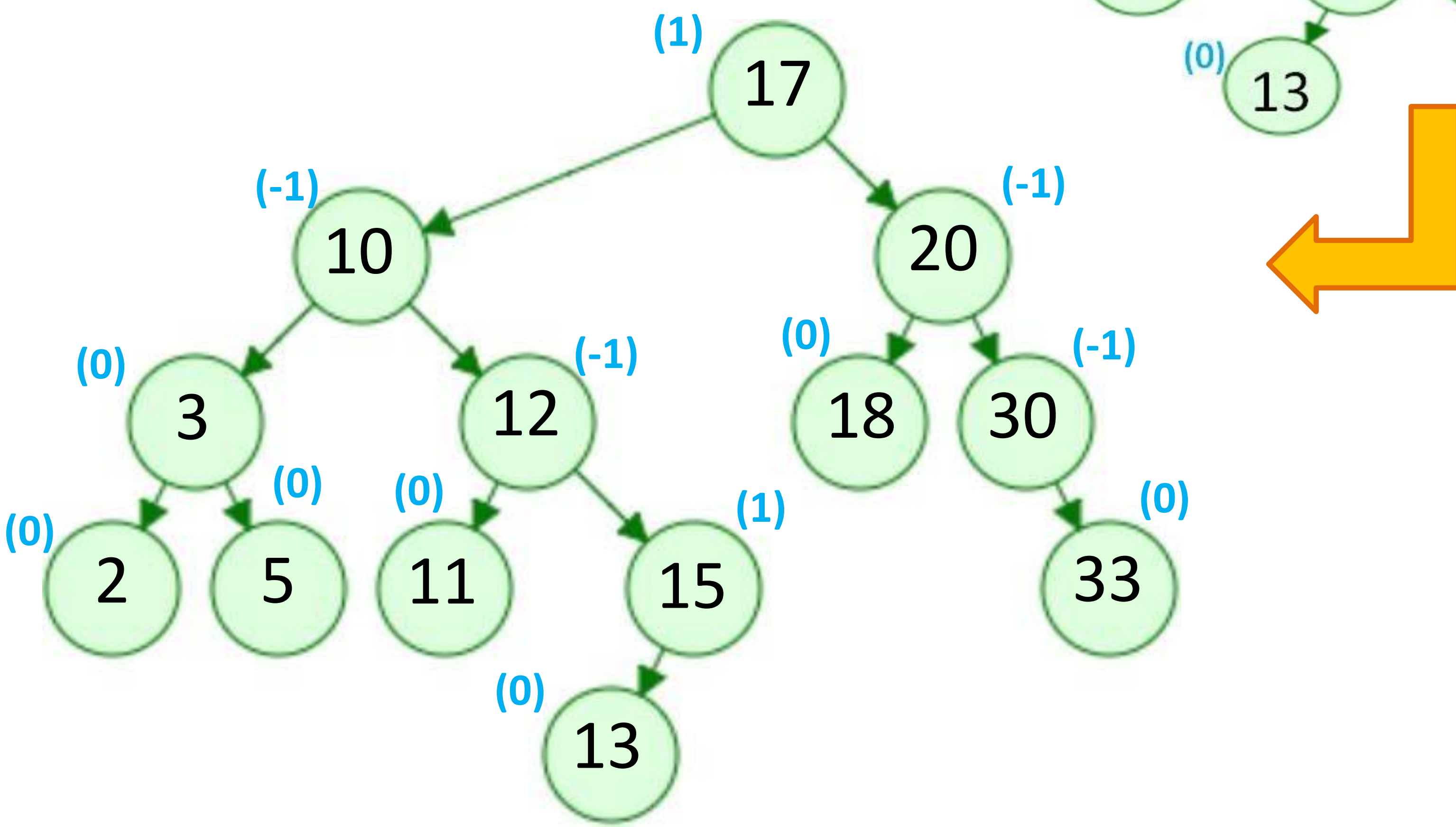
# Contd...


(-1)
10
(2)
5
(0)
17
(-1)
2
(-1)
12
(-1)
20
(0)
3
(0)
11
(1)
15
(0)
18
(-1)
30
(0)
13
(0)
33
(-2)
10
(0)
3
(0)
17
(0)
2
(0)
5
(-1)
12
(-1)
20
(0)
11
(1)
15
(0)
18
(-1)
30
(0)
13
(0)
33

# Contd…



(-2) 10
(0) 3
(0) 17
(0) 2
(0) 5
(-1) 12
(-1) 20
(0) 11
(1) 15
(0) 18
(-1) 30
(0) 13
(0) 33

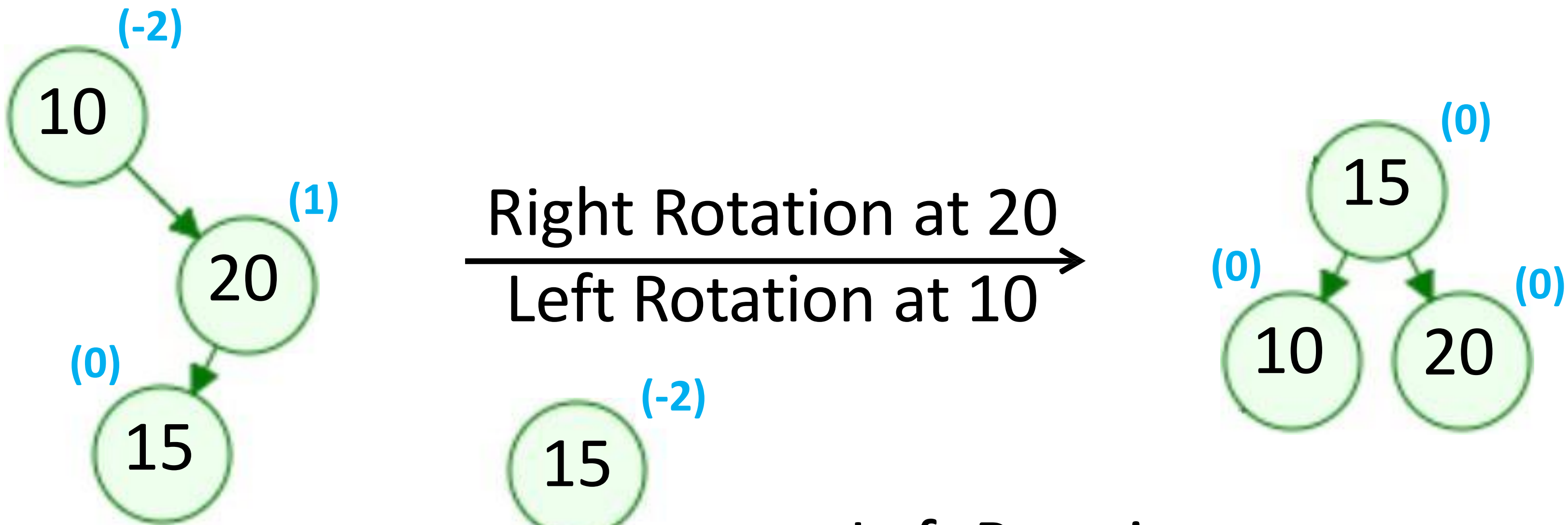



(1) 17
(-1) 10
(-1) 20
(0) 3
(-1) 12
(0) 18
(-1) 30
(0) 2
(0) 5
(0) 11
(1) 15
(0) 33
(0) 13

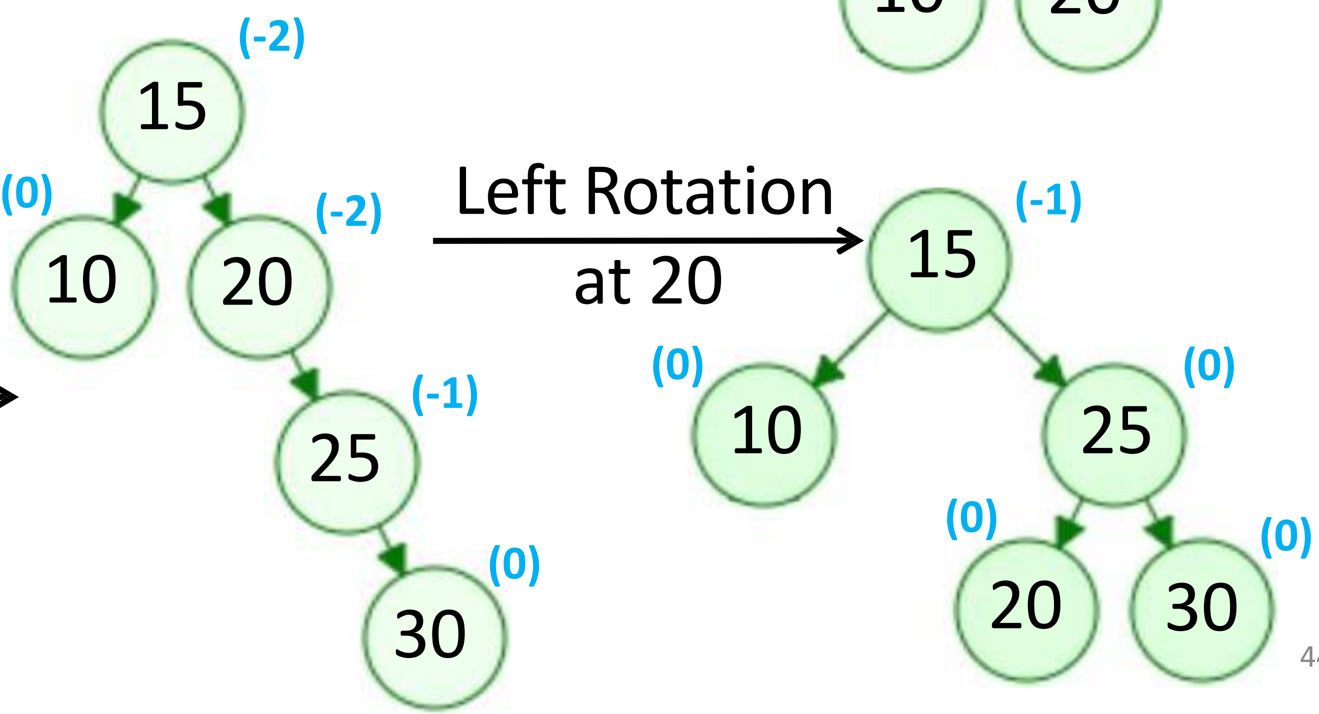
# Create AVL: 10, 20, 15, 25, 30, 16, 18, 19.

# Delete 30.

- Insert 10, 20, 15

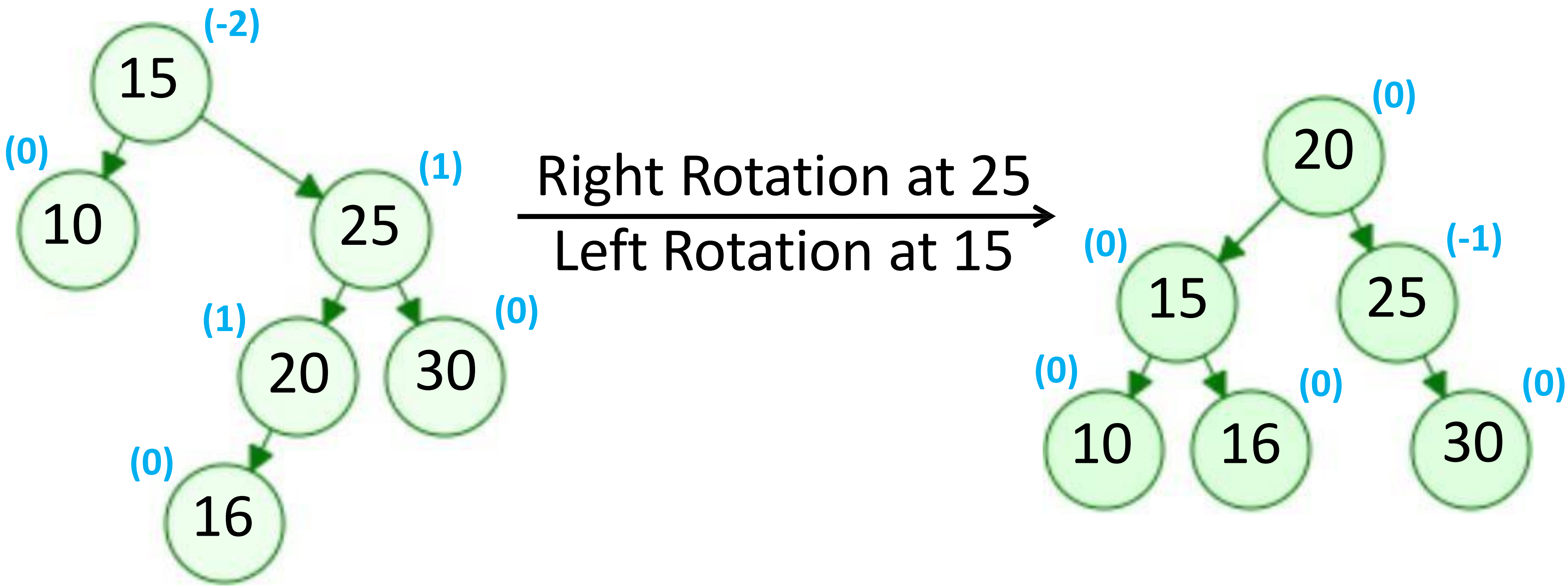



- Insert 25, 30



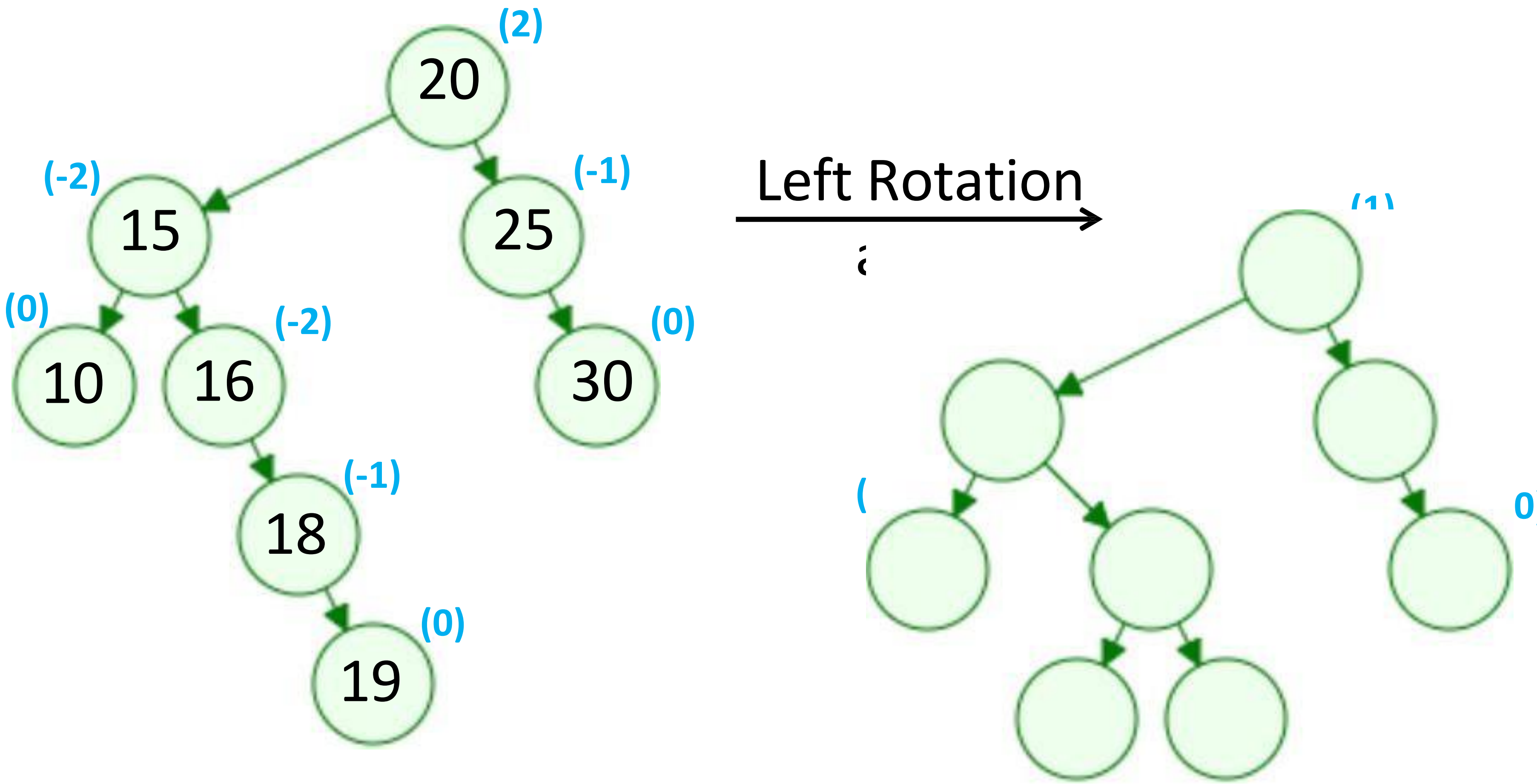
# Contd…

- Insert 16



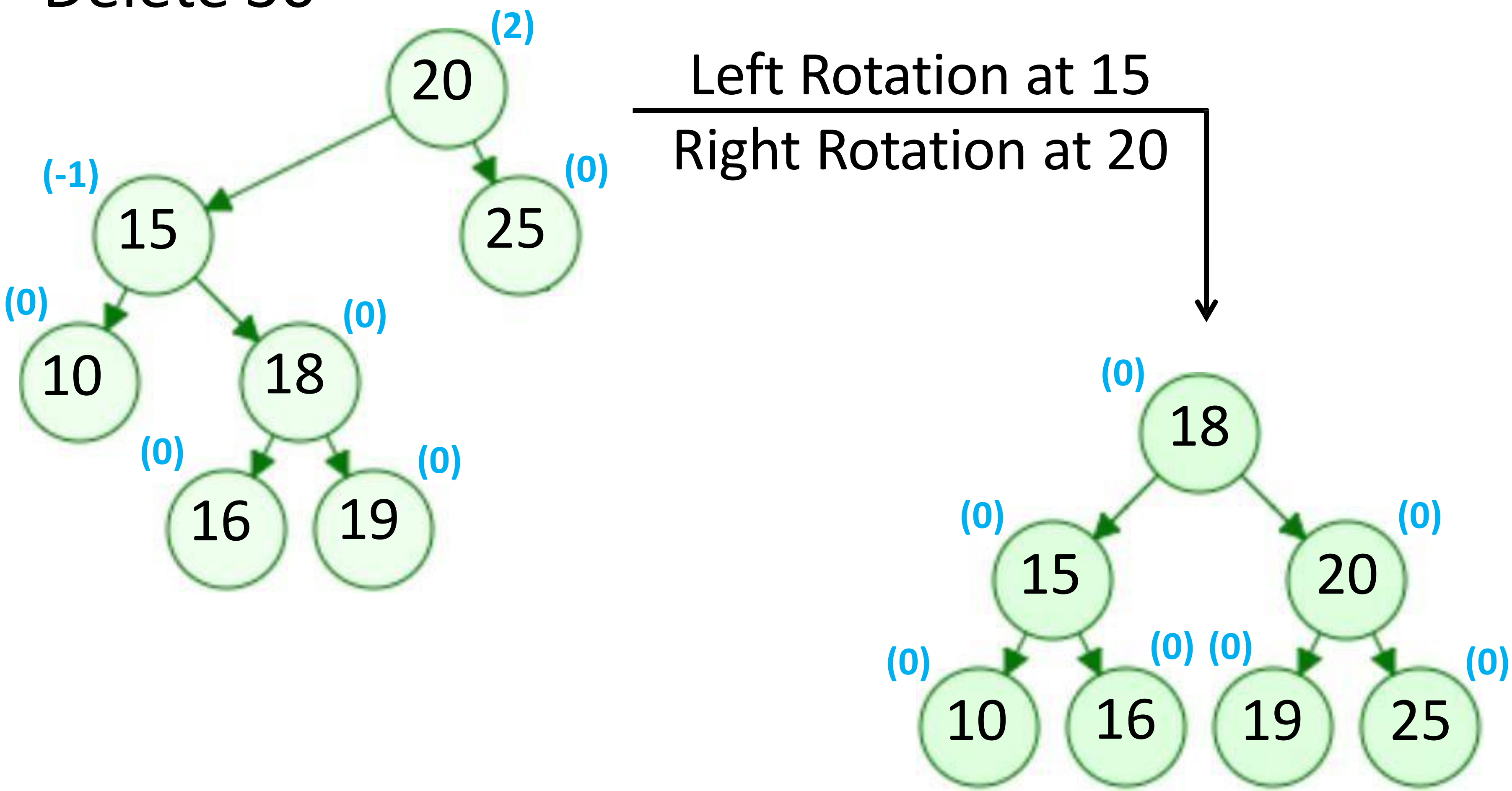
# Contd…

- Insert 18, 19



# Contd…

- Delete 30



# Summary

- AVL trees are always balanced thus worst-case complexity of all operations (search, insert, and delete) is O(log n).
- Rotations performed for height balancing are constant time operations, but takes a little time.
- Difficult to program & debug.
- Needs space to store either a height or a balance factor.
- Suitable for applications where search or look-up is the most frequent operations as compared to insertion or deletion.