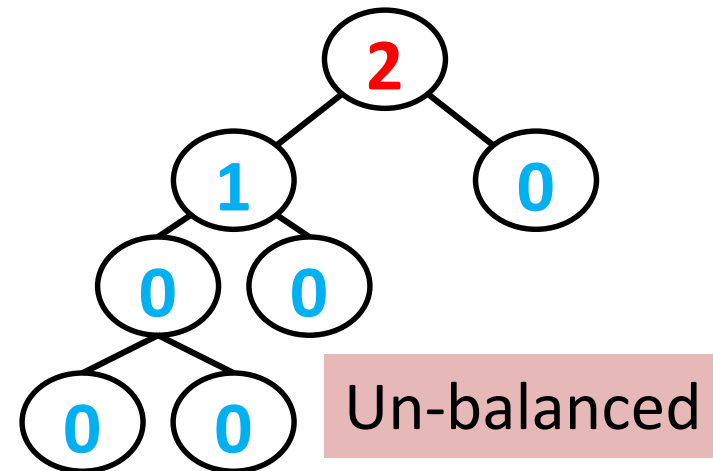
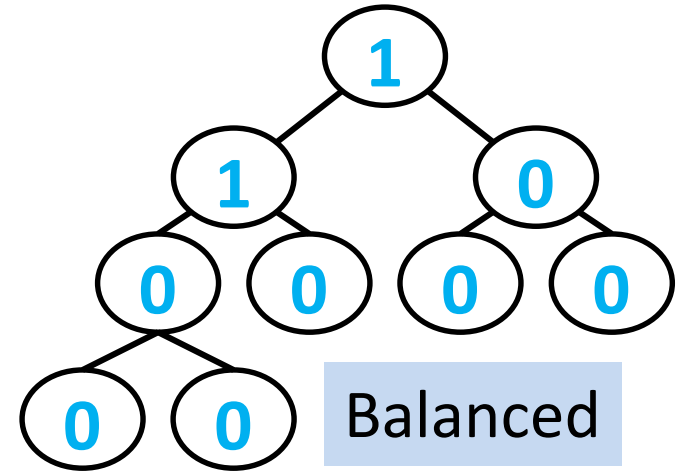


AVL Trees

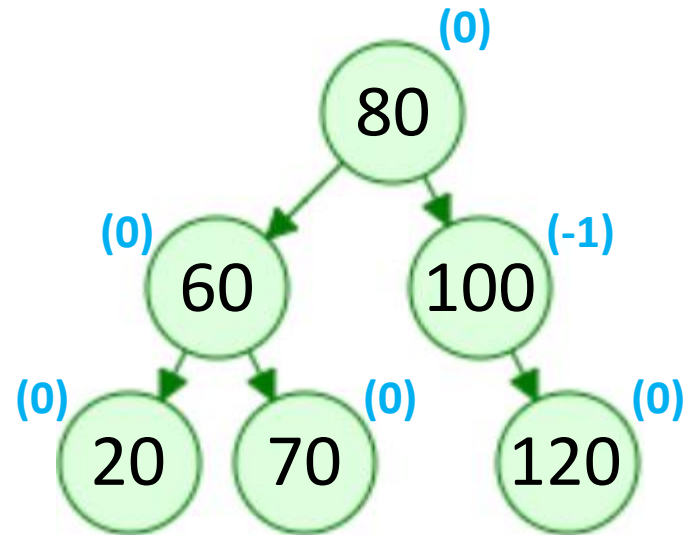
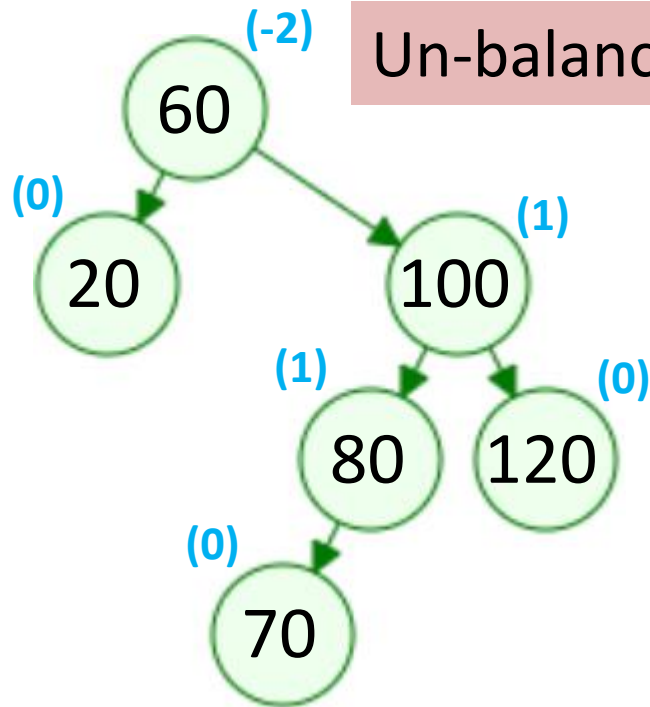
Introduction

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 by one.



Balanced BST???



Contd...

- Invented by Georgy Adelson-Velsky and Evgenii Landis 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

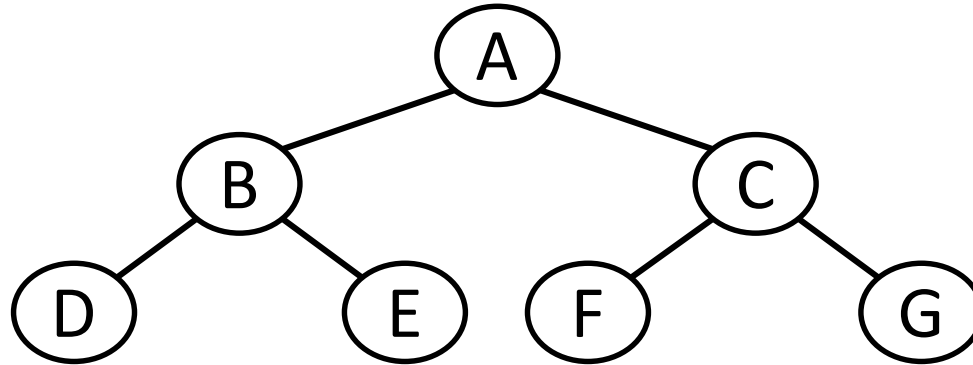
$$|\mathbf{HL} - \mathbf{HR}| \leq 1$$

- 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)

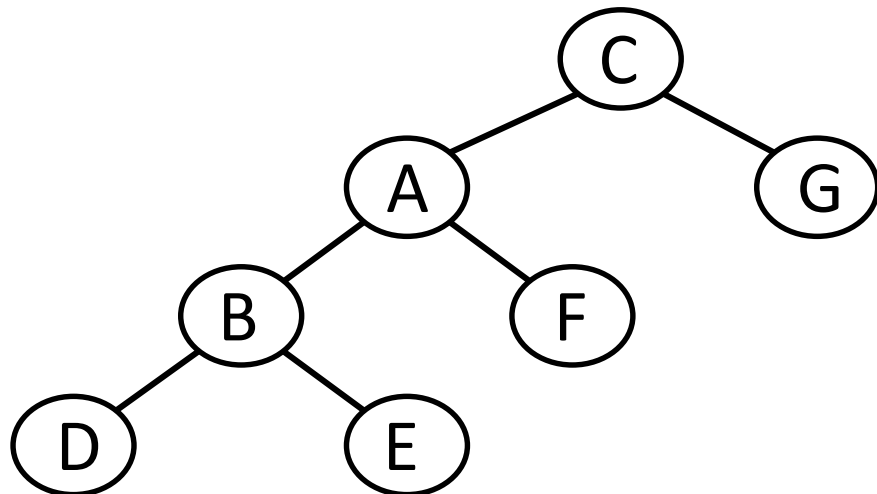
Operations on AVL Tree

- 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.

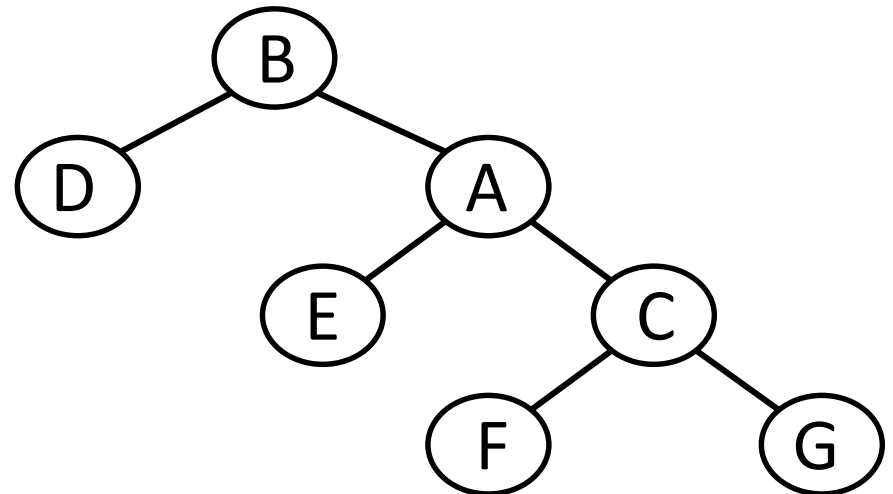
Rotation



- Left rotation



- Right rotation



Node Structure

- Four elements

- data <dataType>
- left <pointer to Node>
- right <pointer to Node>
- parent <pointer to Node>
- height <int>

// Height of a node

OR

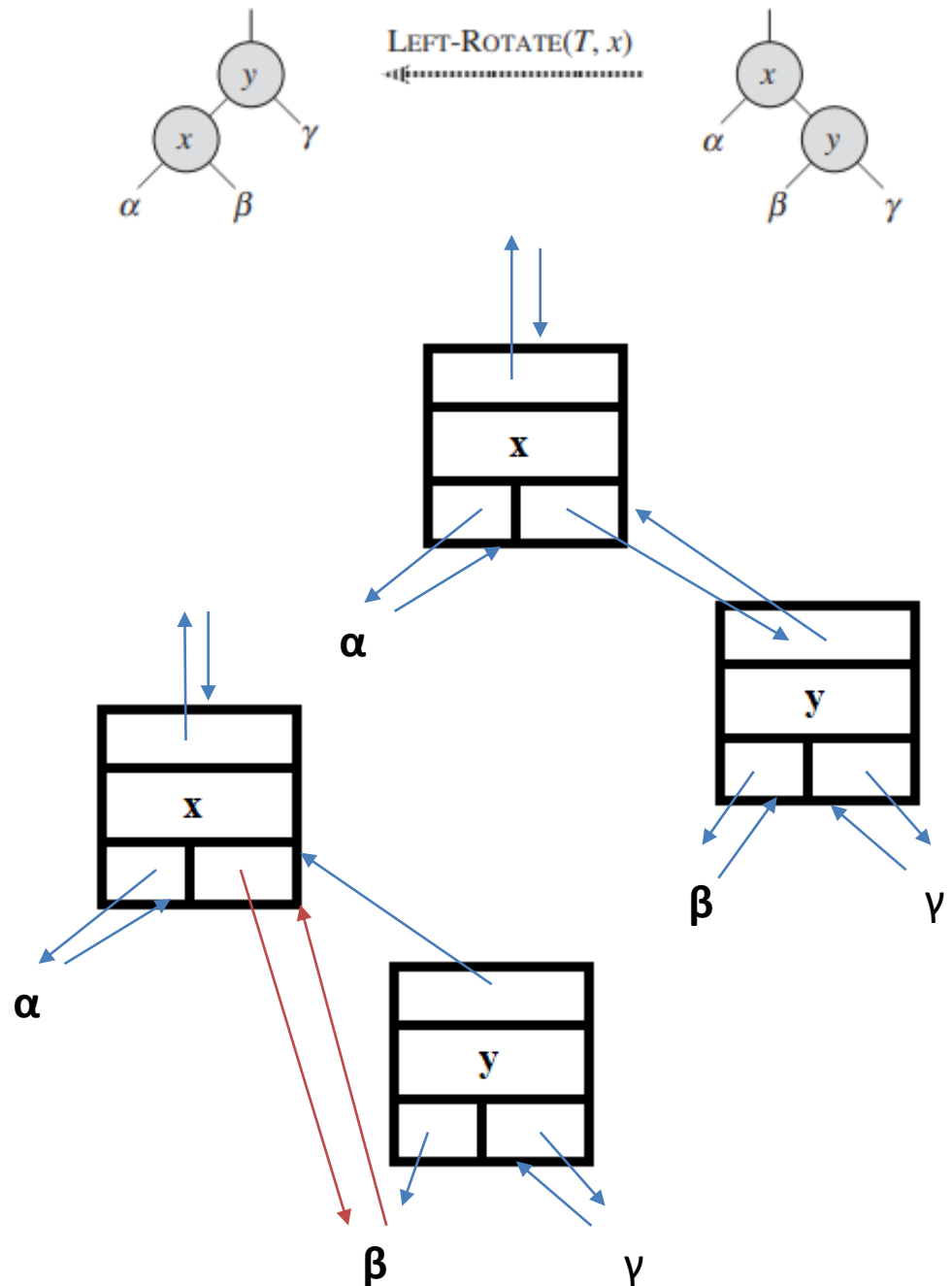
bal <LH (= 1), EH (= 0), RH (= -1)> // Balance factor

For a new node

- data = value
- left = right = NULL
- height = 1

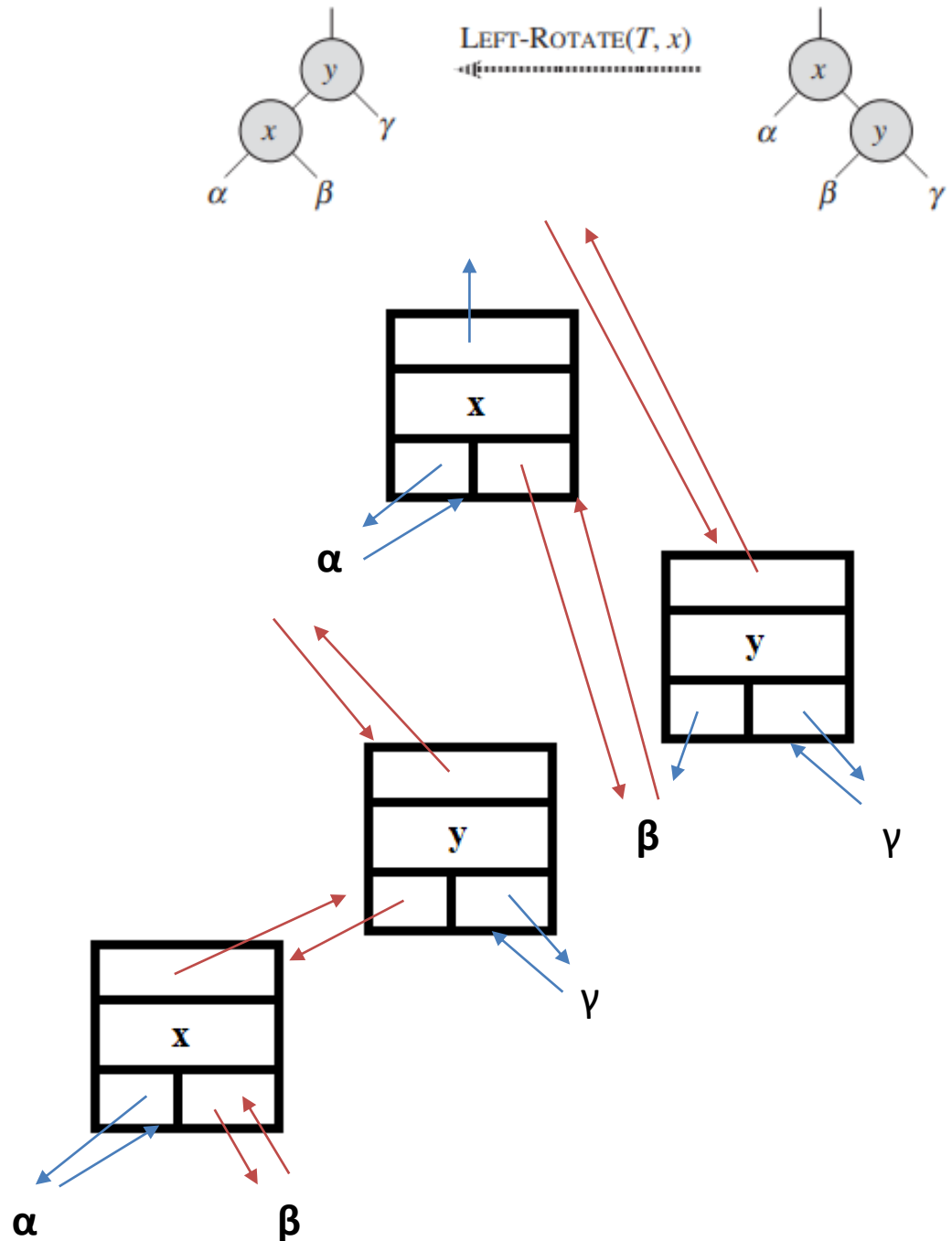
Left-Rotate(T, x)

1. $y = x.right$
2. $x.right = y.left$
3. if $y.left \neq T.nil$
4. $y.left.parent = x$
5. $y.parent = x.parent$
6. if $x.parent == T.nil$
7. $T.root = y$
8. elseif $x == x.parent.left$
9. $x.parent.left = y$
10. else $x.parent.right = y$
11. $y.left = x$
12. $x.parent = y$



Left-Rotate(T, x)

1. $y = x.right$
2. $x.right = y.left$
3. if $y.left \neq T.nil$
4. $y.left.parent = x$
5. $y.parent = x.parent$
6. if $x.parent == T.nil$
7. $T.root = y$
8. elseif $x == x.parent.left$
9. $x.parent.left = y$
10. else $x.parent.right = y$
11. $y.left = x$
12. $x.parent = y$



Left-Rotate(T, x)

1. $y = x.right$
2. $x.right = y.left$
3. if $y.left \neq T.nil$
4. $y.left.parent = x$
5. $y.parent = x.parent$
6. if $x.parent == T.nil$
7. $T.root = y$
8. elseif $x == x.parent.left$
9. $x.parent.left = y$
10. else $x.parent.right = y$
11. $y.left = x$
12. $x.parent = y$

Right-Rotate(T, x)

1. $y = x.left$
2. $x.left = y.right$
3. if $y.right \neq T.nil$
4. $y.right.parent = x$
5. $y.parent = x.parent$
6. if $x.parent == T.nil$
7. $T.root = y$
8. elseif $x == x.parent.right$
9. $x.parent.right = y$
10. else $x.parent.left = y$
11. $y.right = x$
12. $x.parent = y$

Node Structure (Without Parent)

- Four elements

- data <dataType>
- left <pointer to Node>
- right <pointer to Node>
- height <int>

// Height of a node

OR

bal <LH (= 1), EH (= 0), RH (= -1)> // Balance factor

For a new node

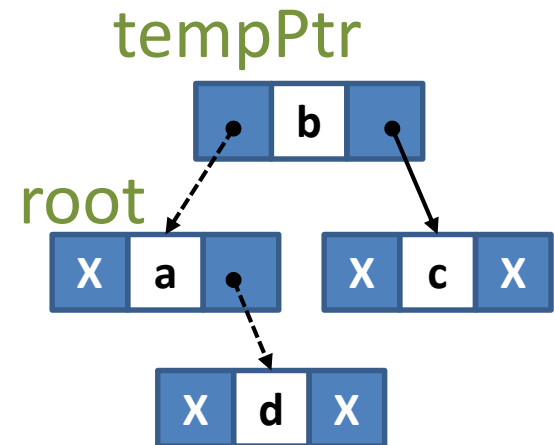
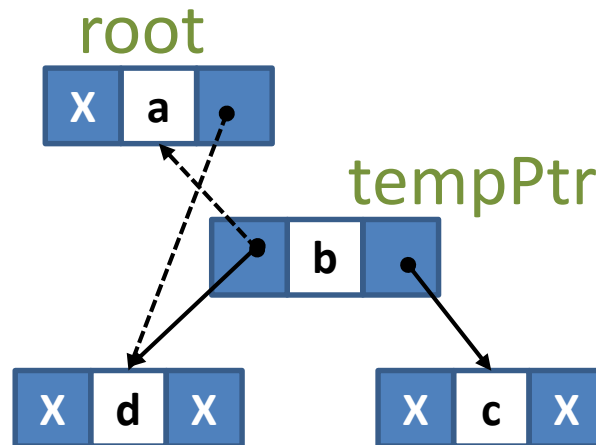
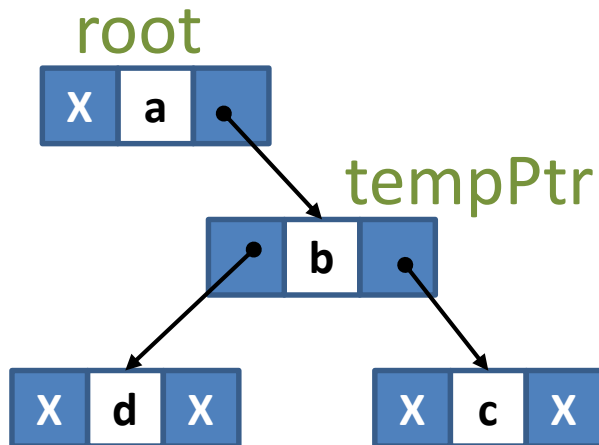
- data = value
- left = right = NULL
- height = 1

Left Rotation

Algorithm rotateLeft(root)

1. Exchange right subtree of root with left subtree of its right subtree.
2. Make its right subtree new root.

```
NODE* rotateLeft(NODE *root)
{
    NODE *tempPtr;
    tempPtr = root -> right;
    root->right = tempPtr -> left;
    tempPtr -> left = root;
    // Update height of root
    // Update height of tempPtr
    return tempPtr;
}
```

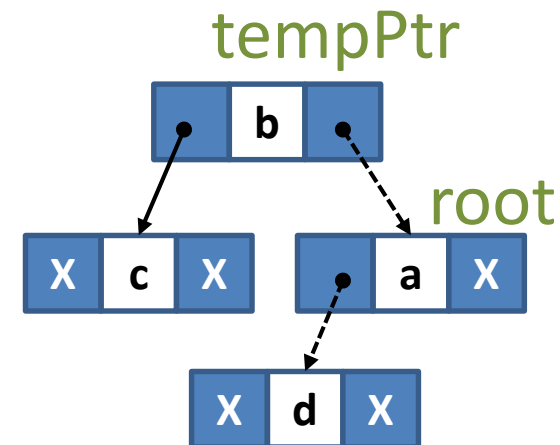
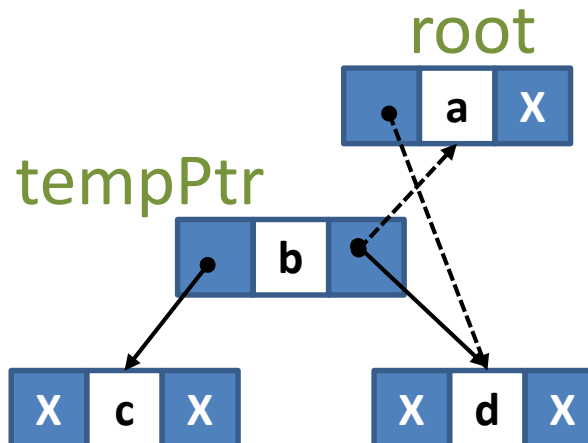
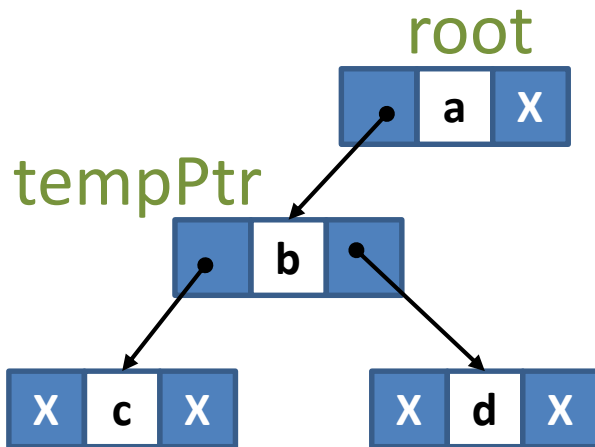


Right Rotation

Algorithm rotateRight(root)

1. Exchange left subtree of root with right subtree of its left subtree.
2. Make its left subtree new root.

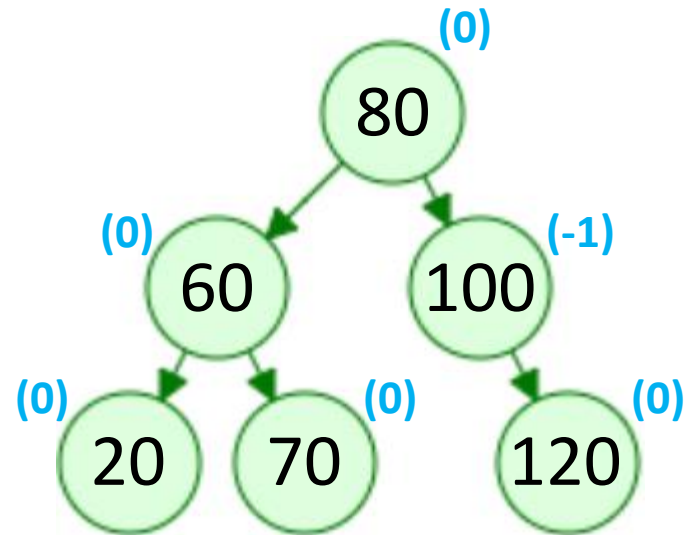
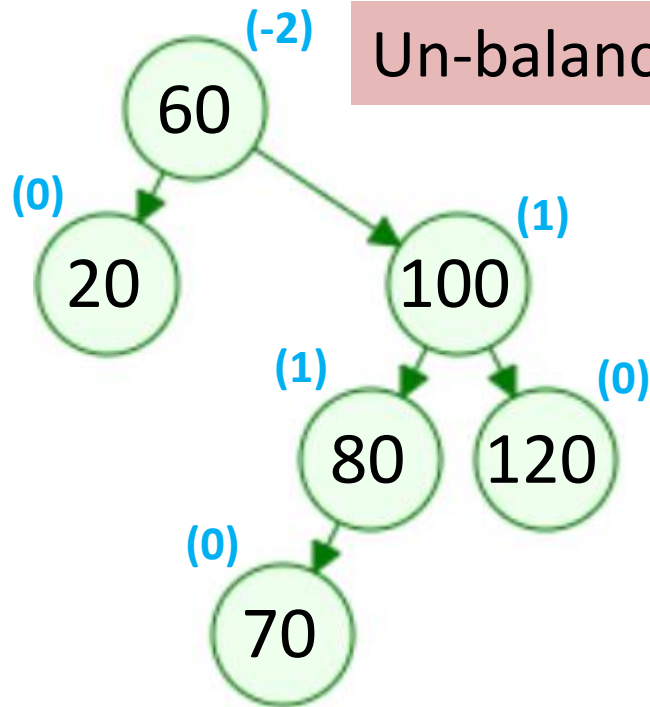
```
NODE* rotateRight(NODE *root)
{
    NODE *tempPtr;
    tempPtr = root -> left;
    root->left = tempPtr -> right;
    tempPtr -> right = root;
    // Update height of root
    // Update height of tempPtr
    return tempPtr;
}
```



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 .

Balanced BST???



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.

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

To ensure it's $2^{h/2}$, let's 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,

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

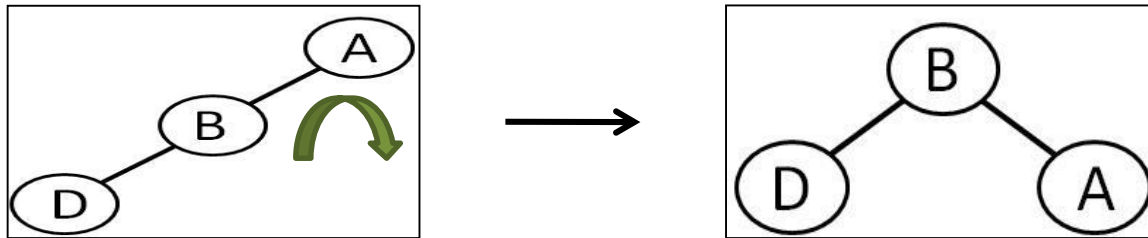
- Thus, in the worst-case AVL trees have height $h = O(\log n)$.
- This means that nicer/more balanced AVL trees will have the same bound on their height.

AVL Trees

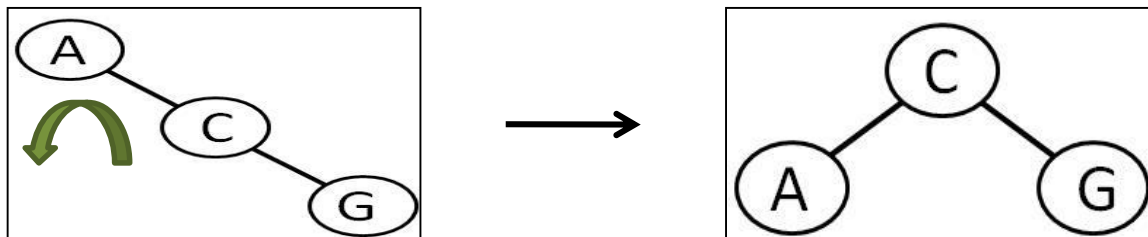
Insertion

Unbalanced Cases

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

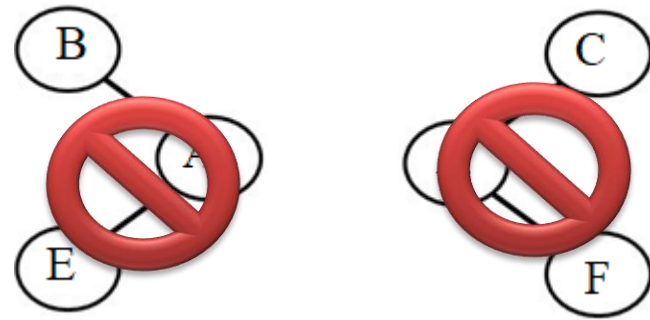


- **Right of Right:** insertion turned the right subtree of a right high AVL tree into a right high tree.

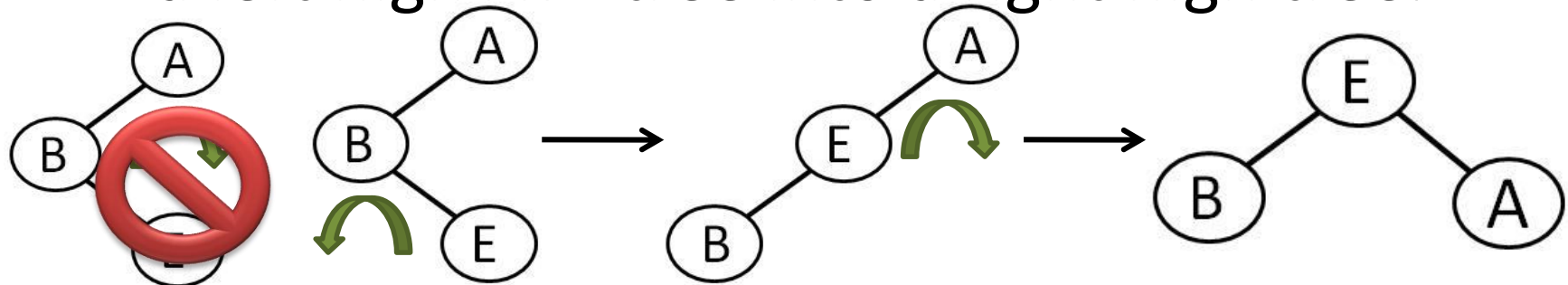


Unbalanced Cases

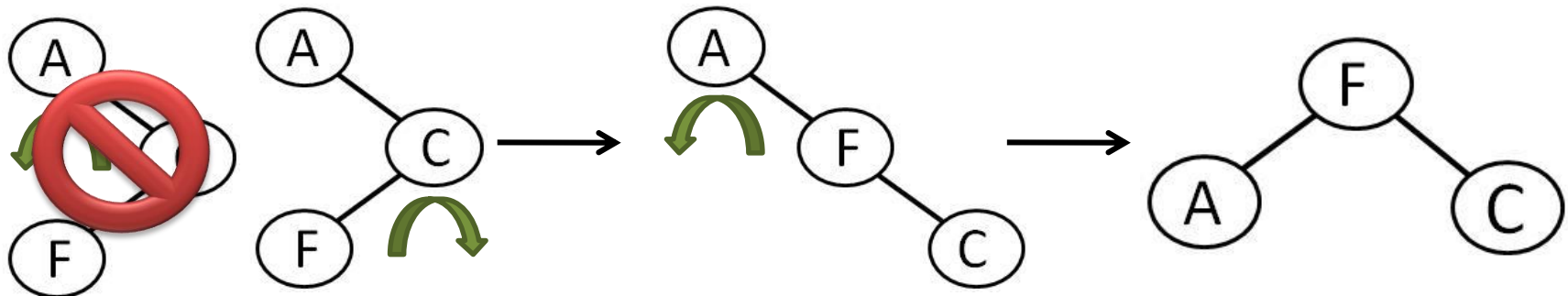
- Double rotation



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

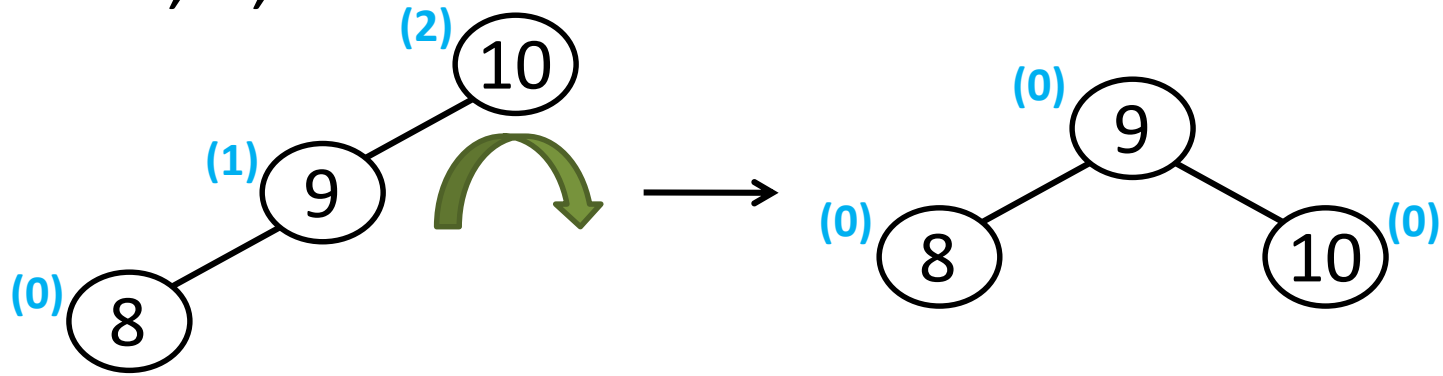


– **Left of Right:** insertion turned the right subtree of a right high AVL tree into a left high tree.

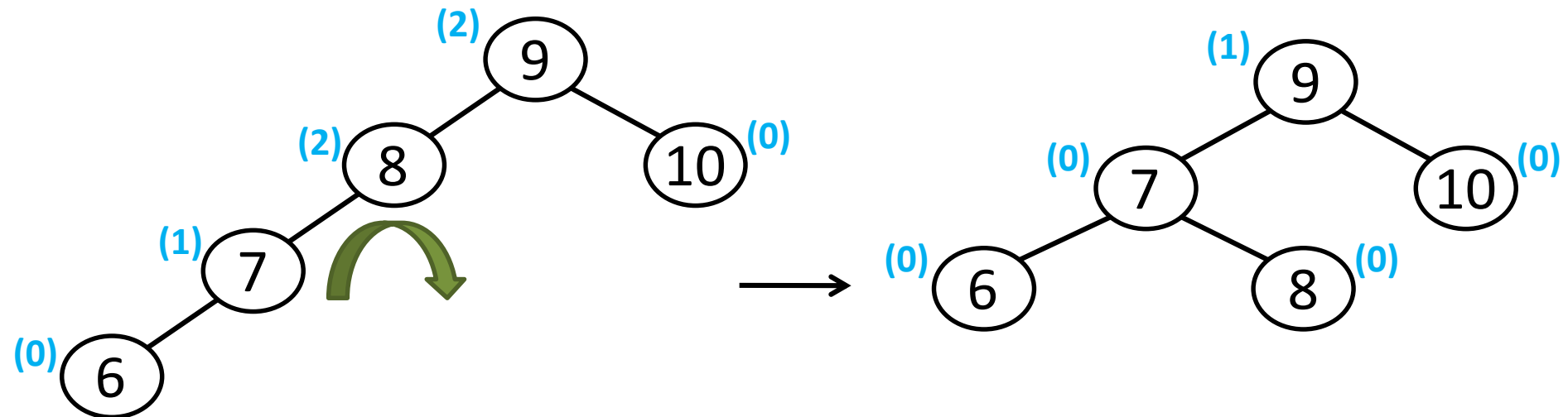


Example 1: Insert 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

- Insert 10, 9, 8

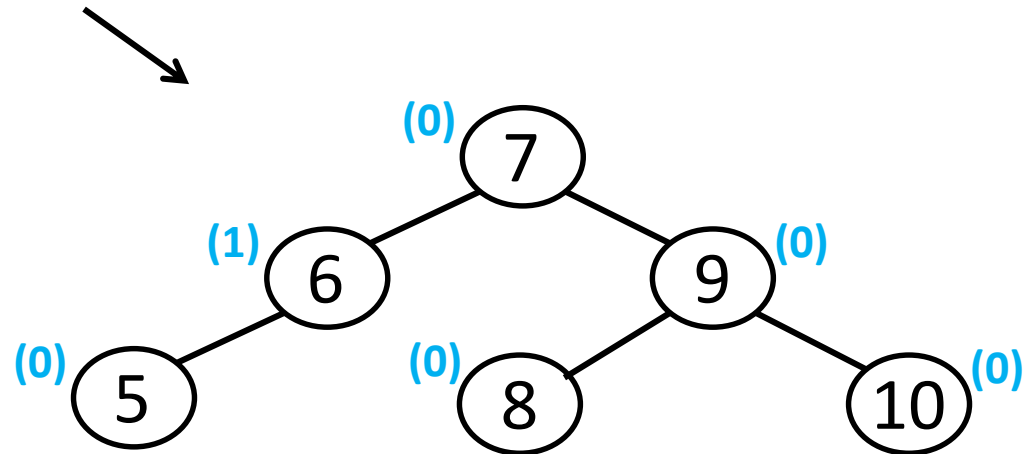
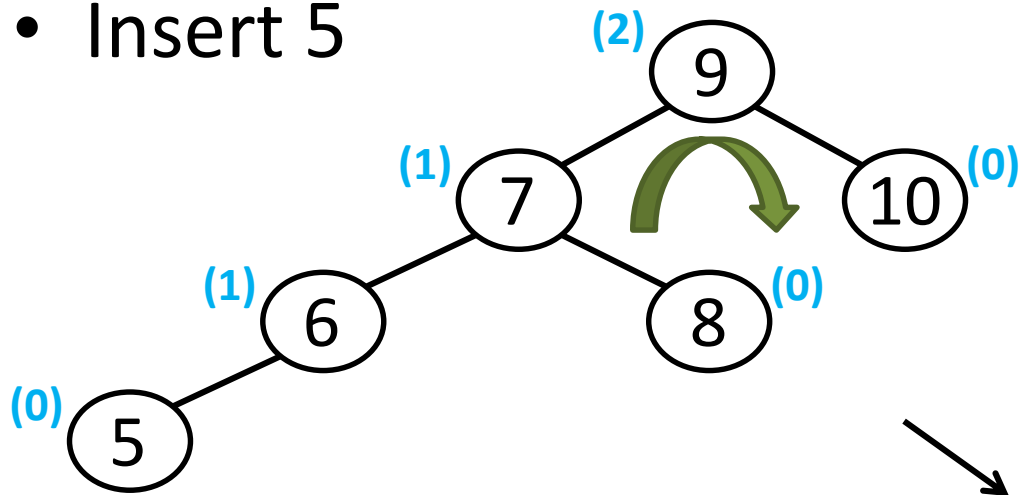


- Insert 7, 6



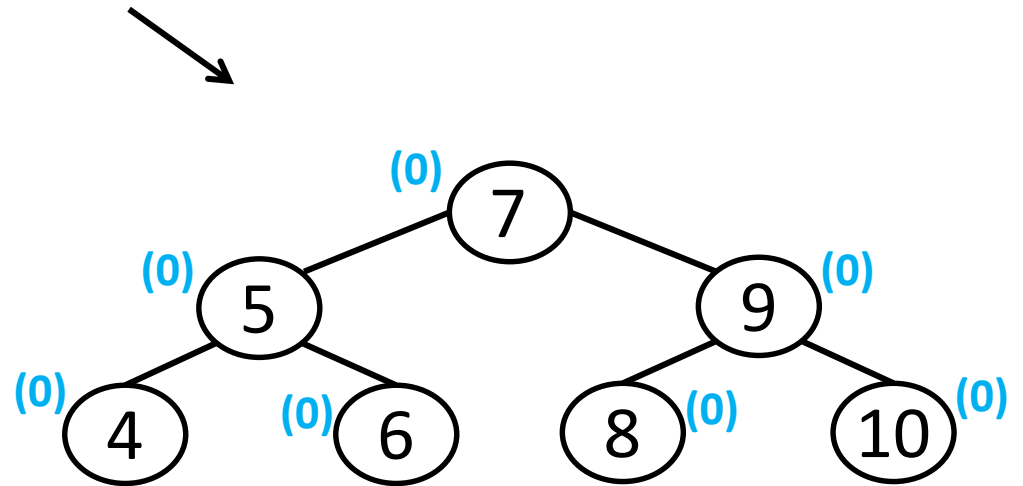
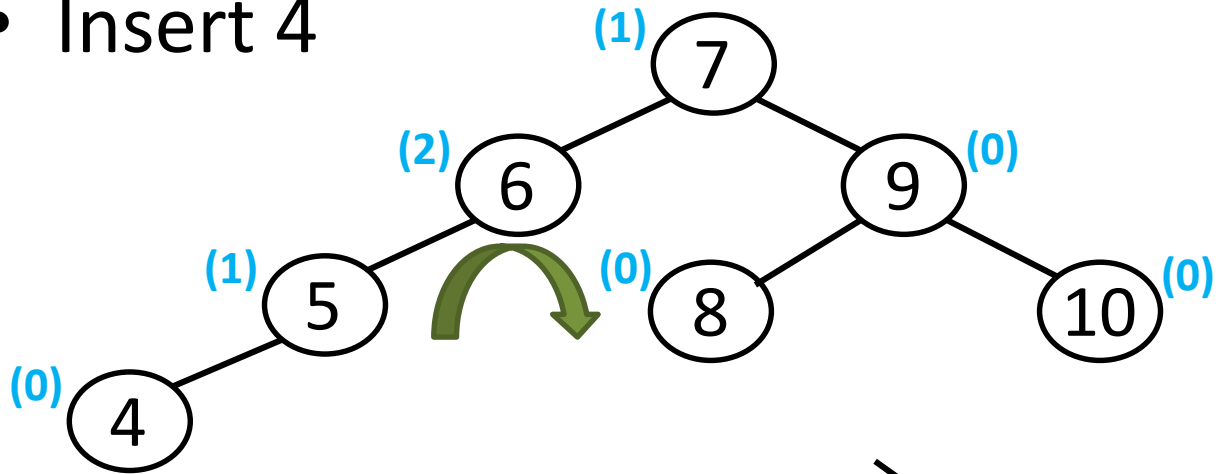
Contd...

- Insert 5



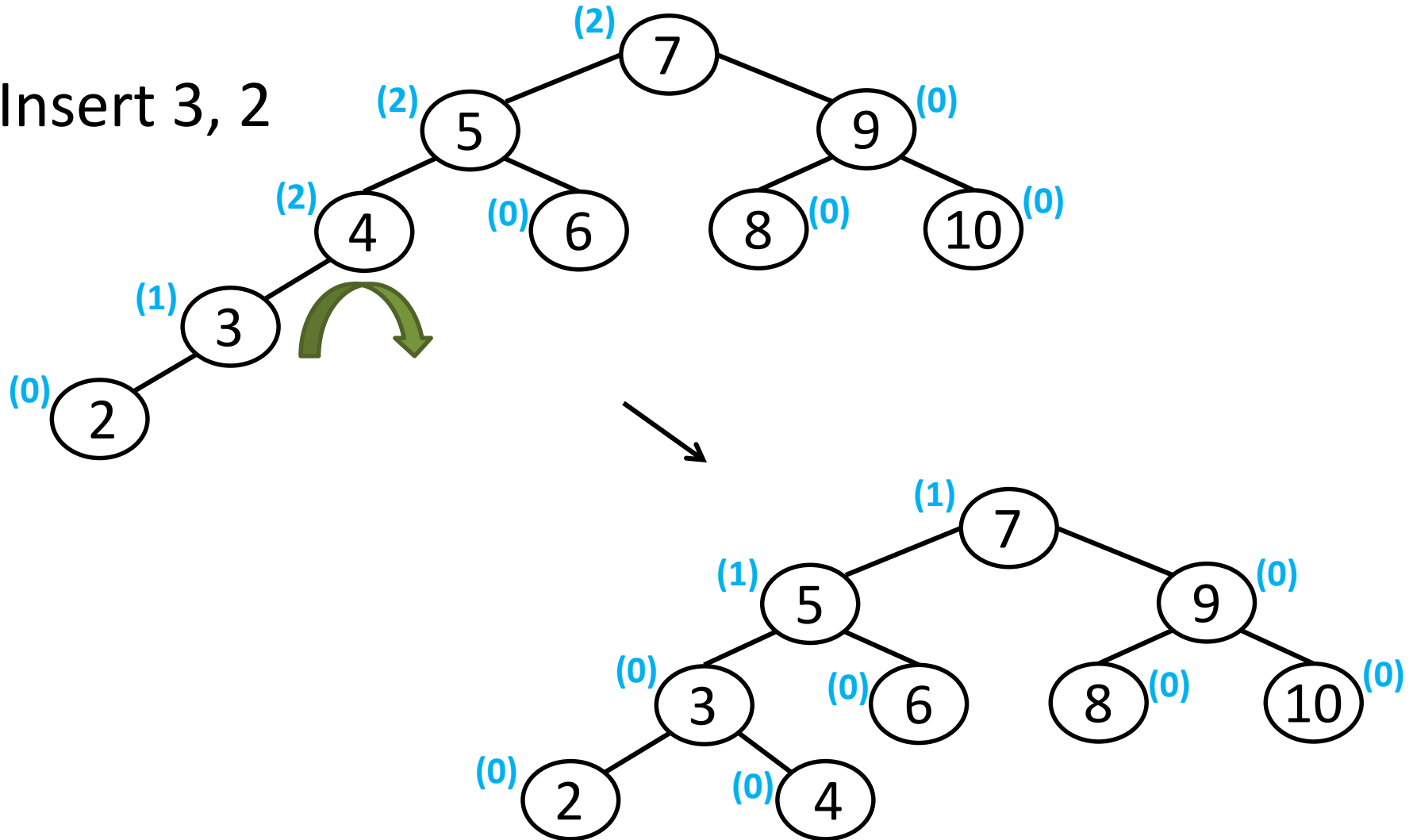
Contd...

- Insert 4



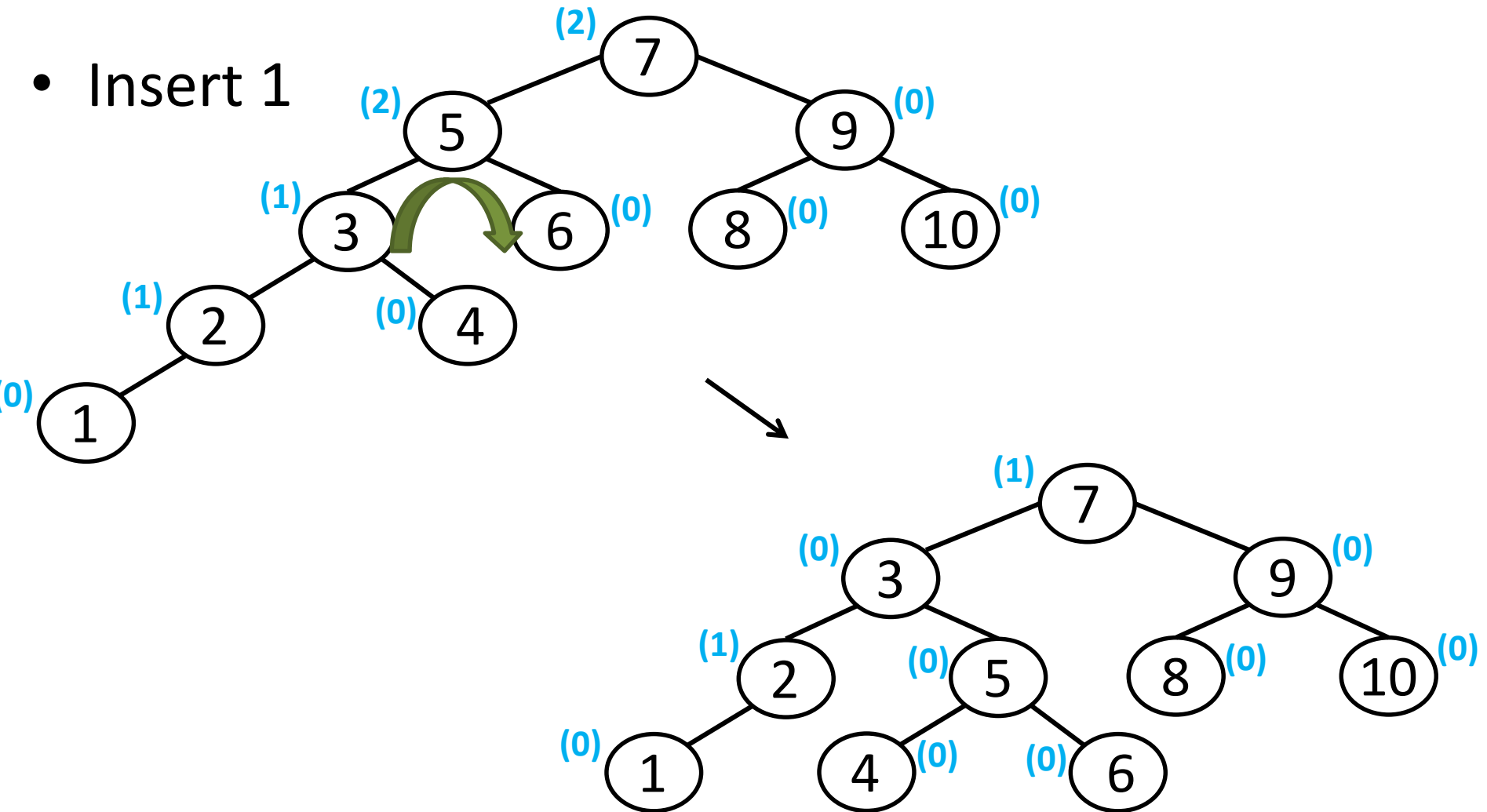
Contd...

- Insert 3, 2



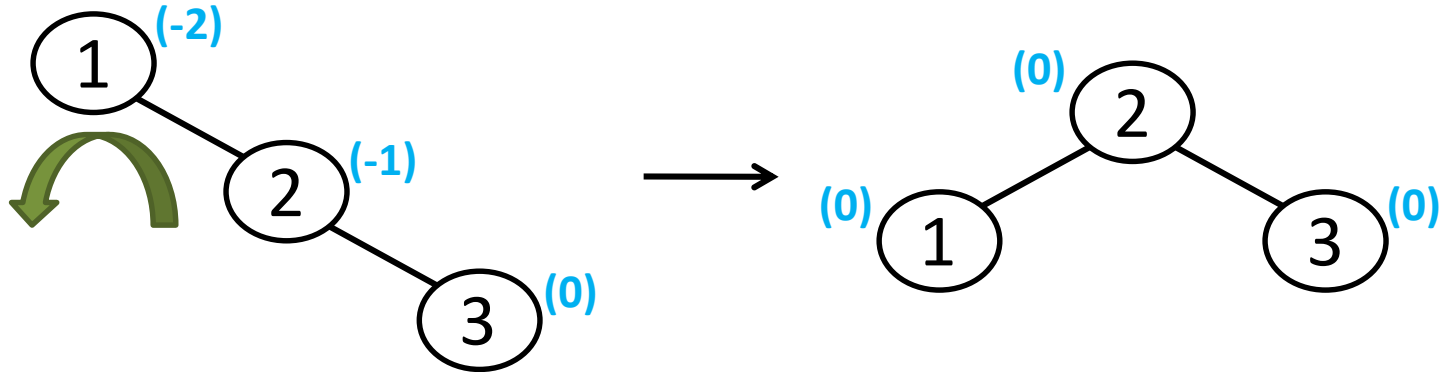
Contd...

- Insert 1

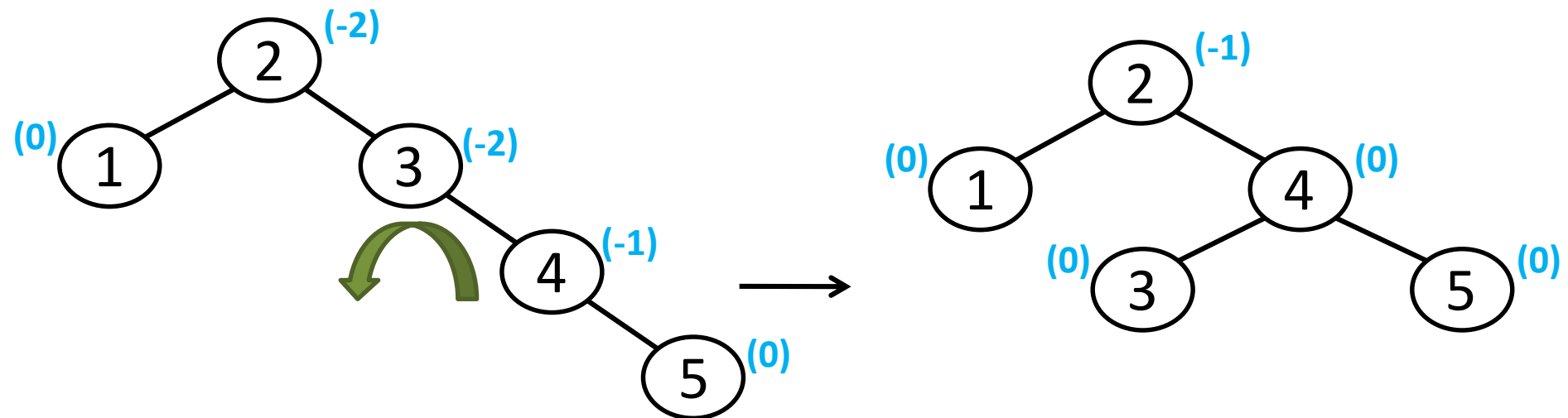


Example 2: Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

- Insert 1, 2, 3

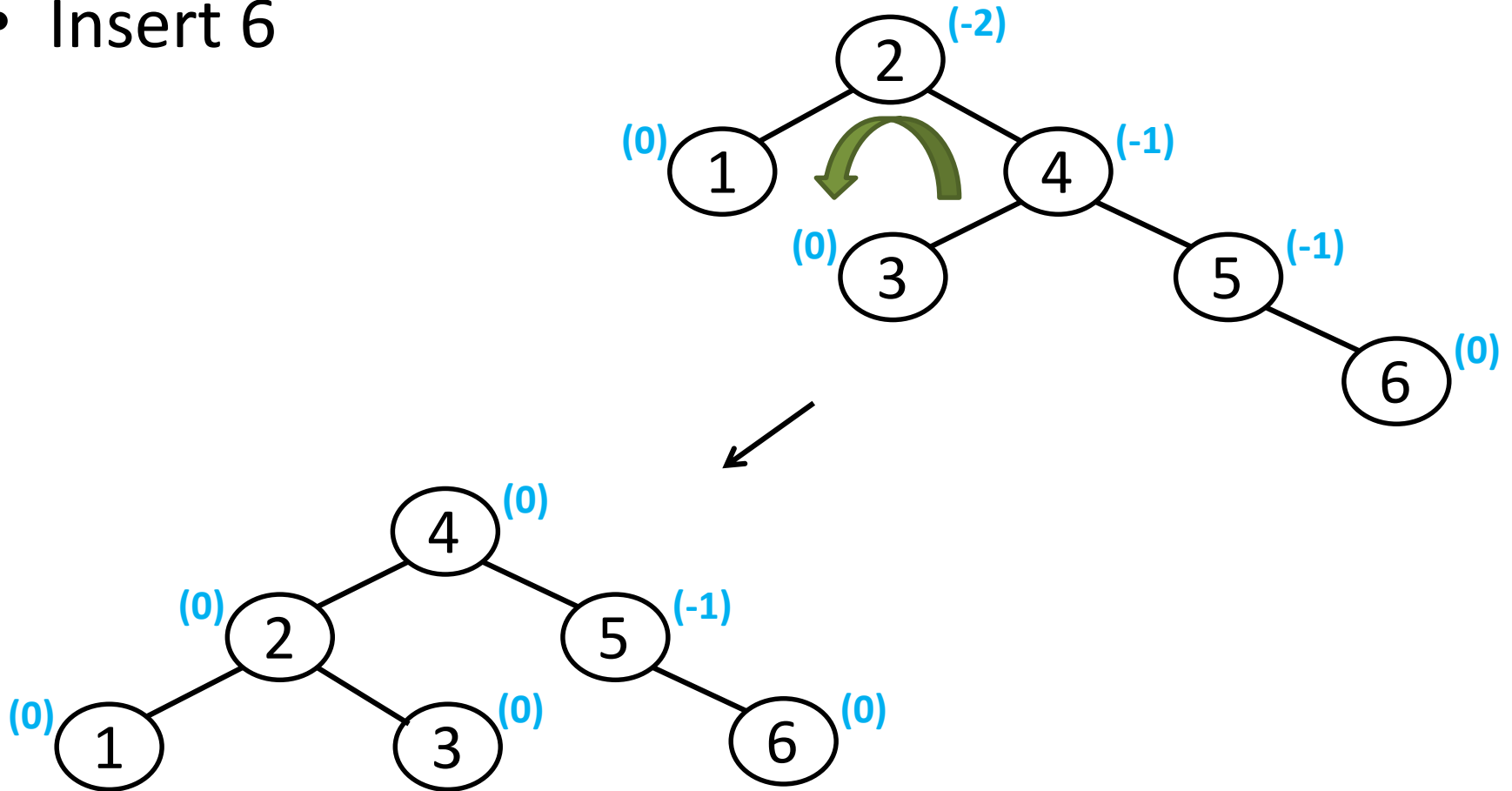


- Insert 4, 5



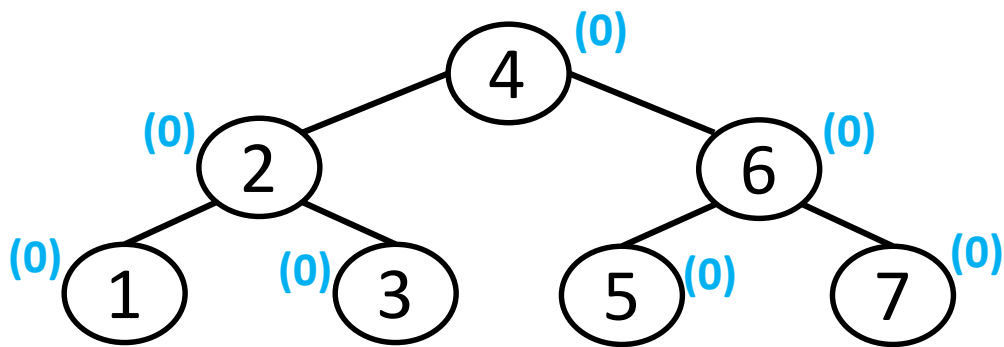
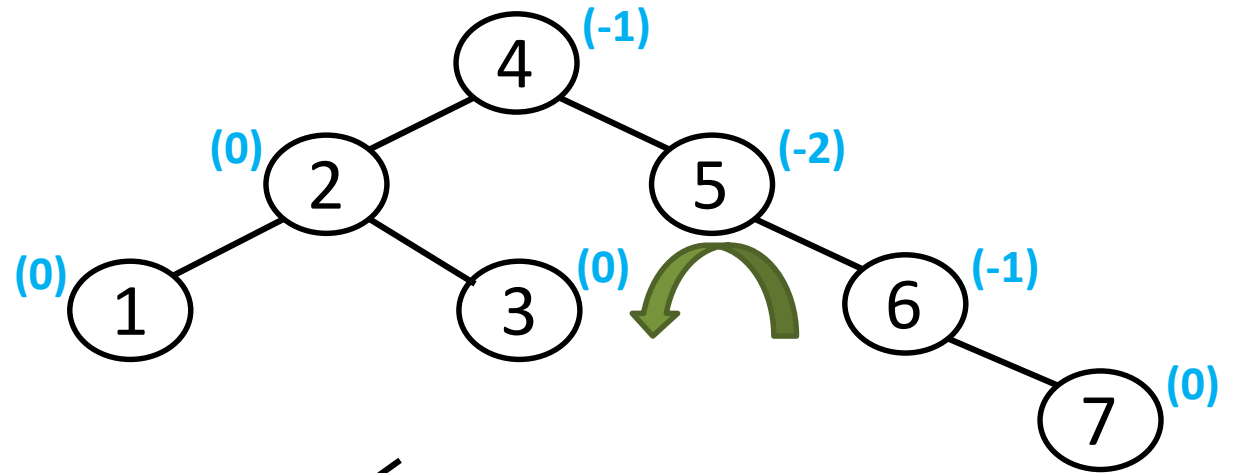
Contd...

- Insert 6



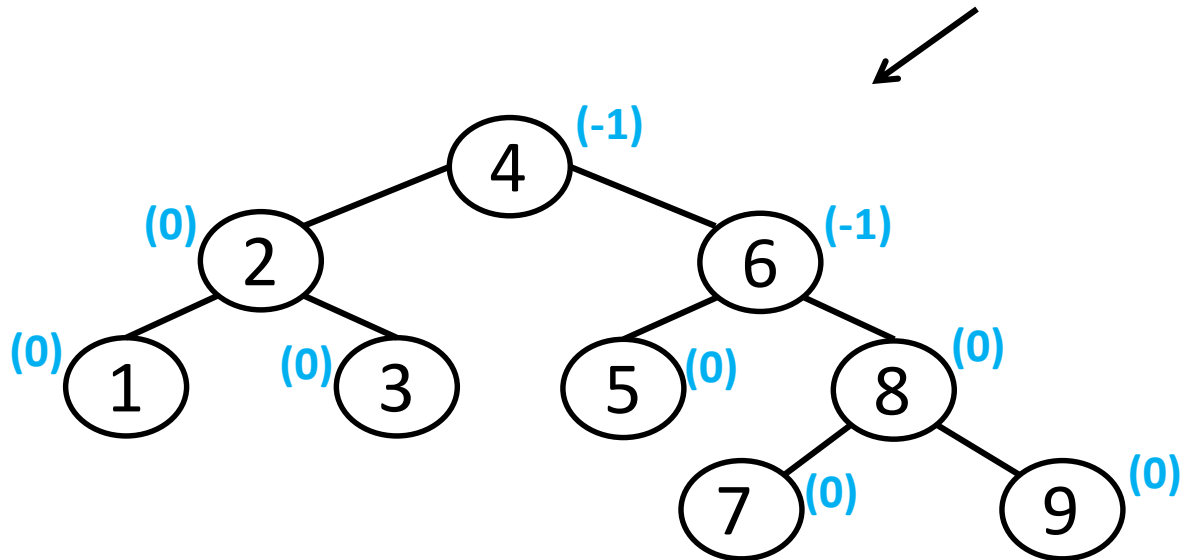
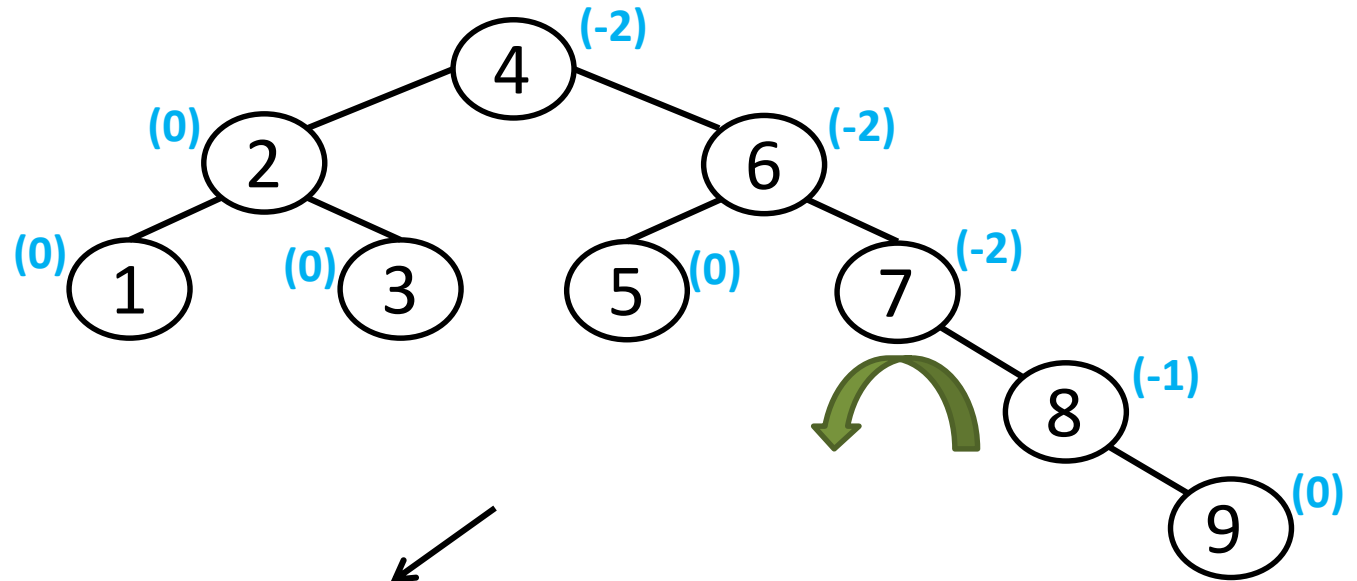
Contd...

- Insert 7



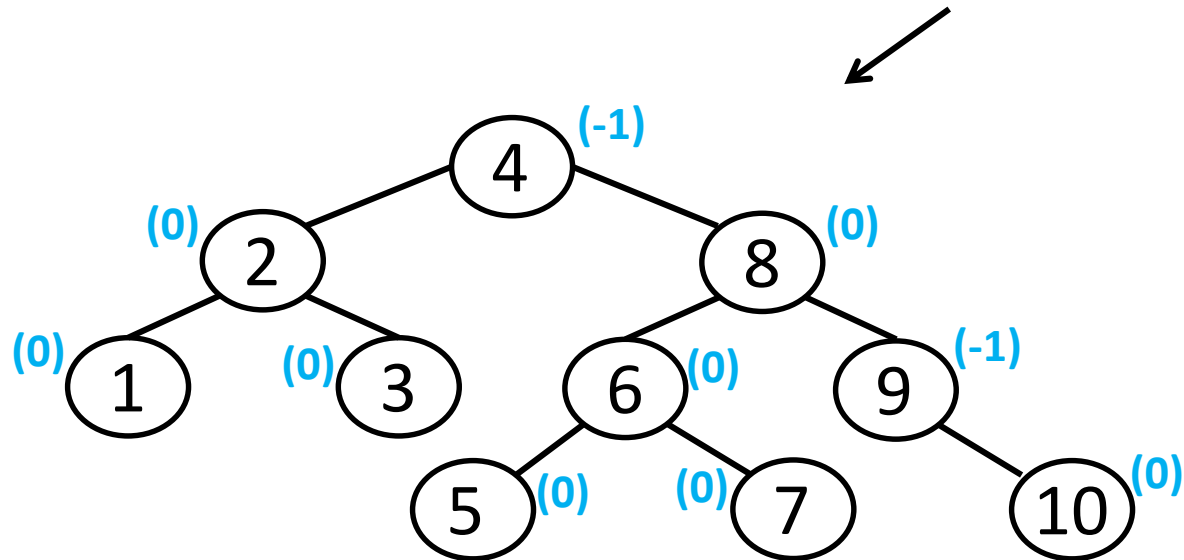
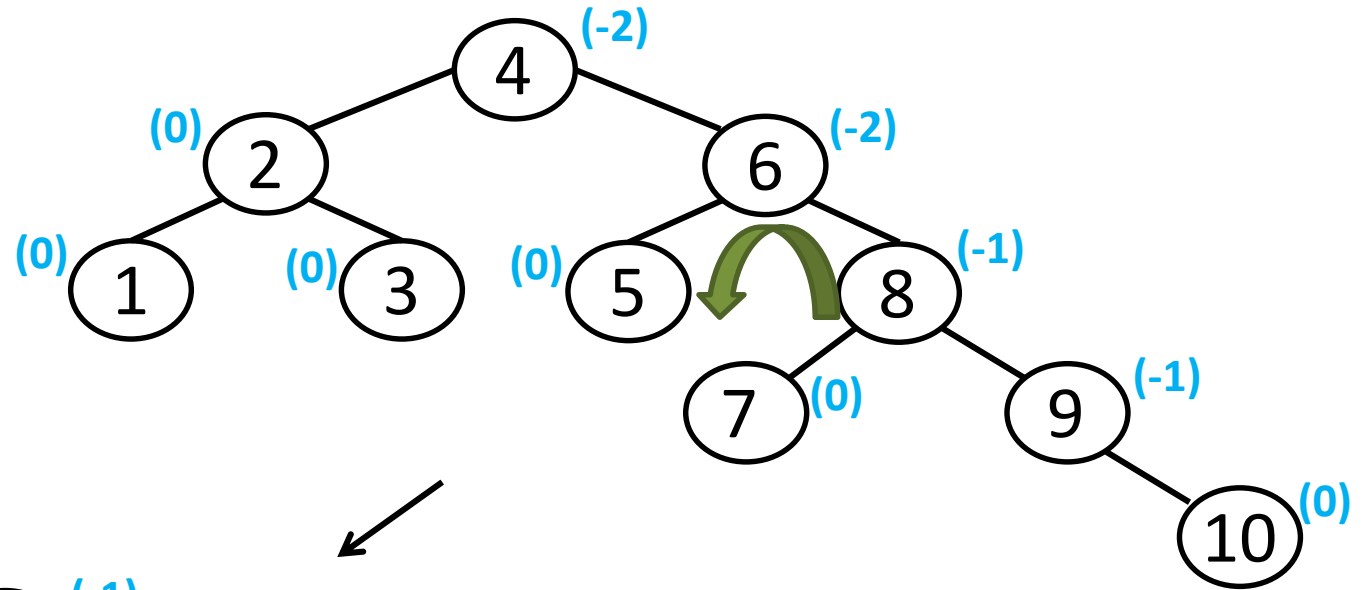
Contd...

- Insert 8, 9

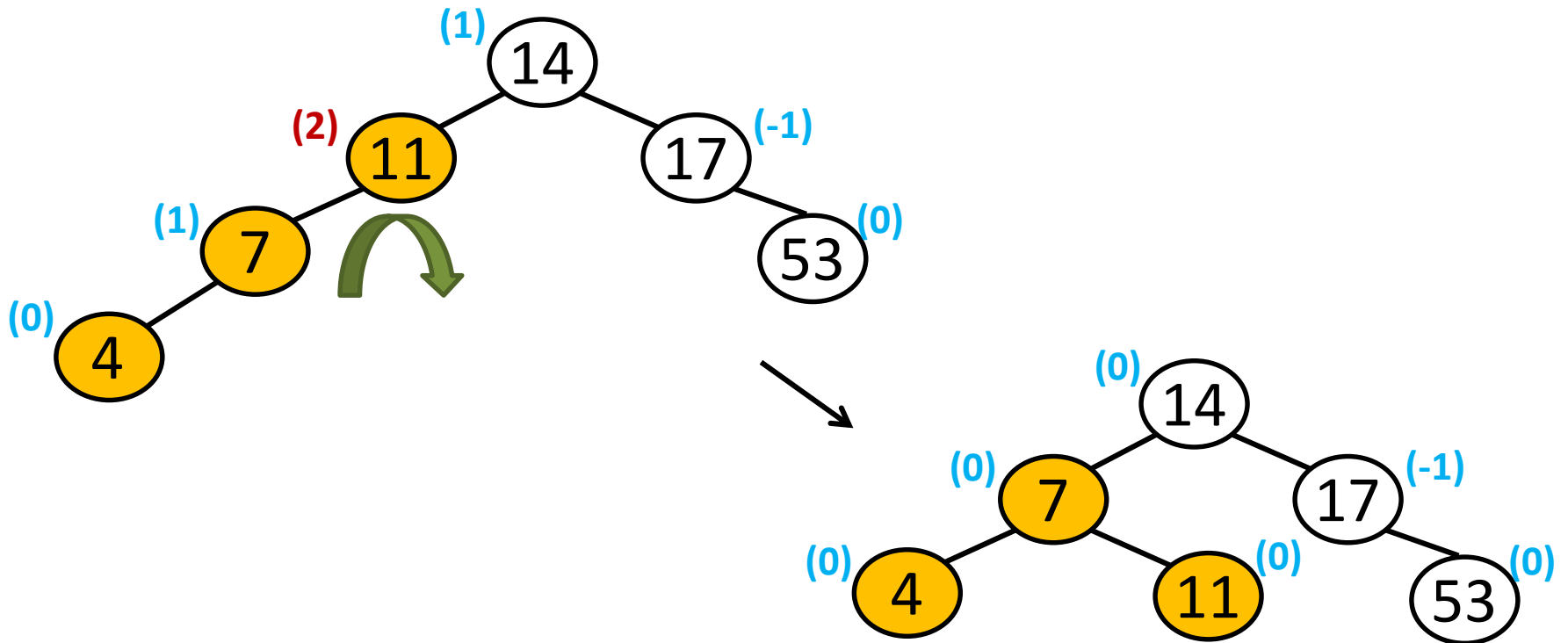


Contd...

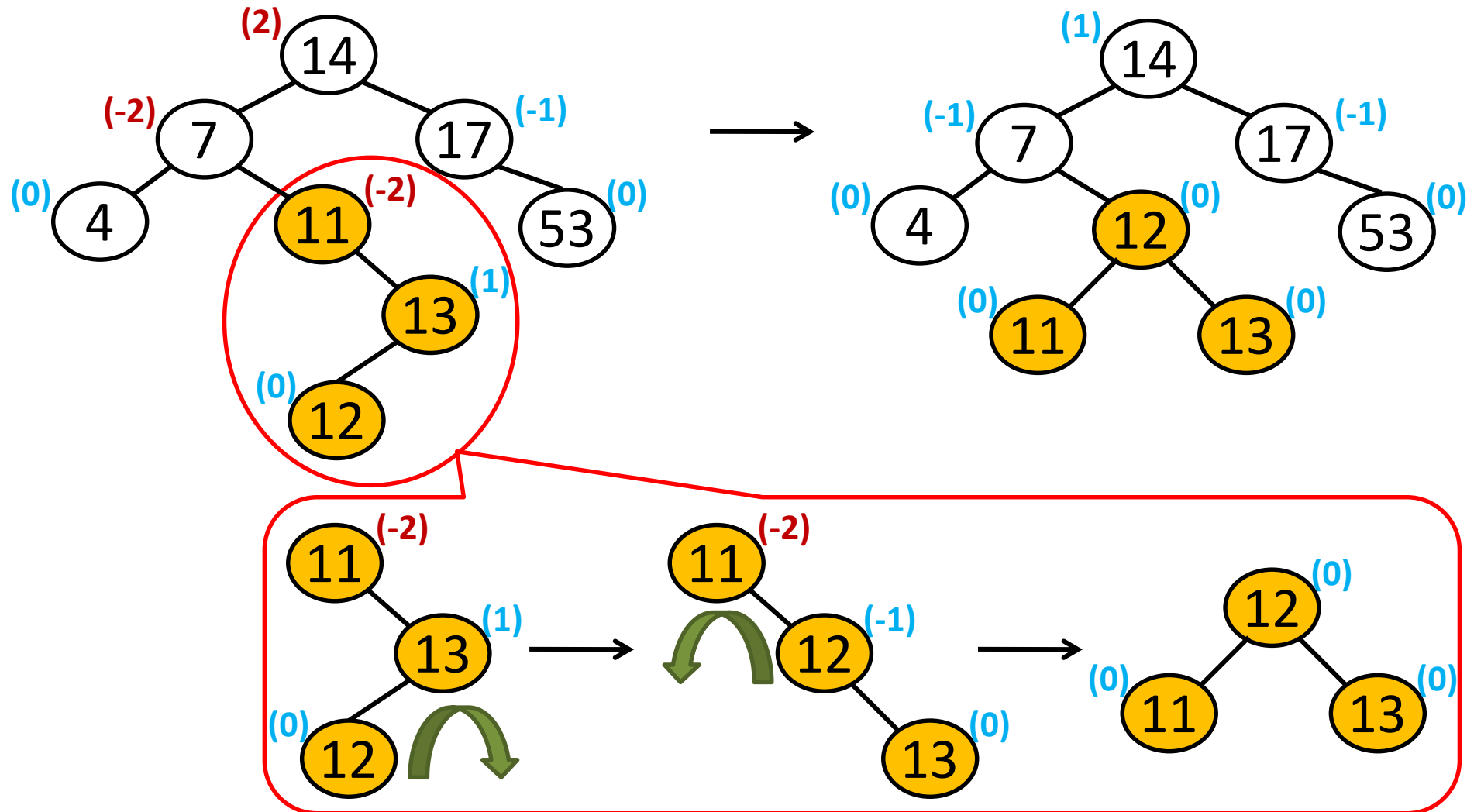
- Insert 10



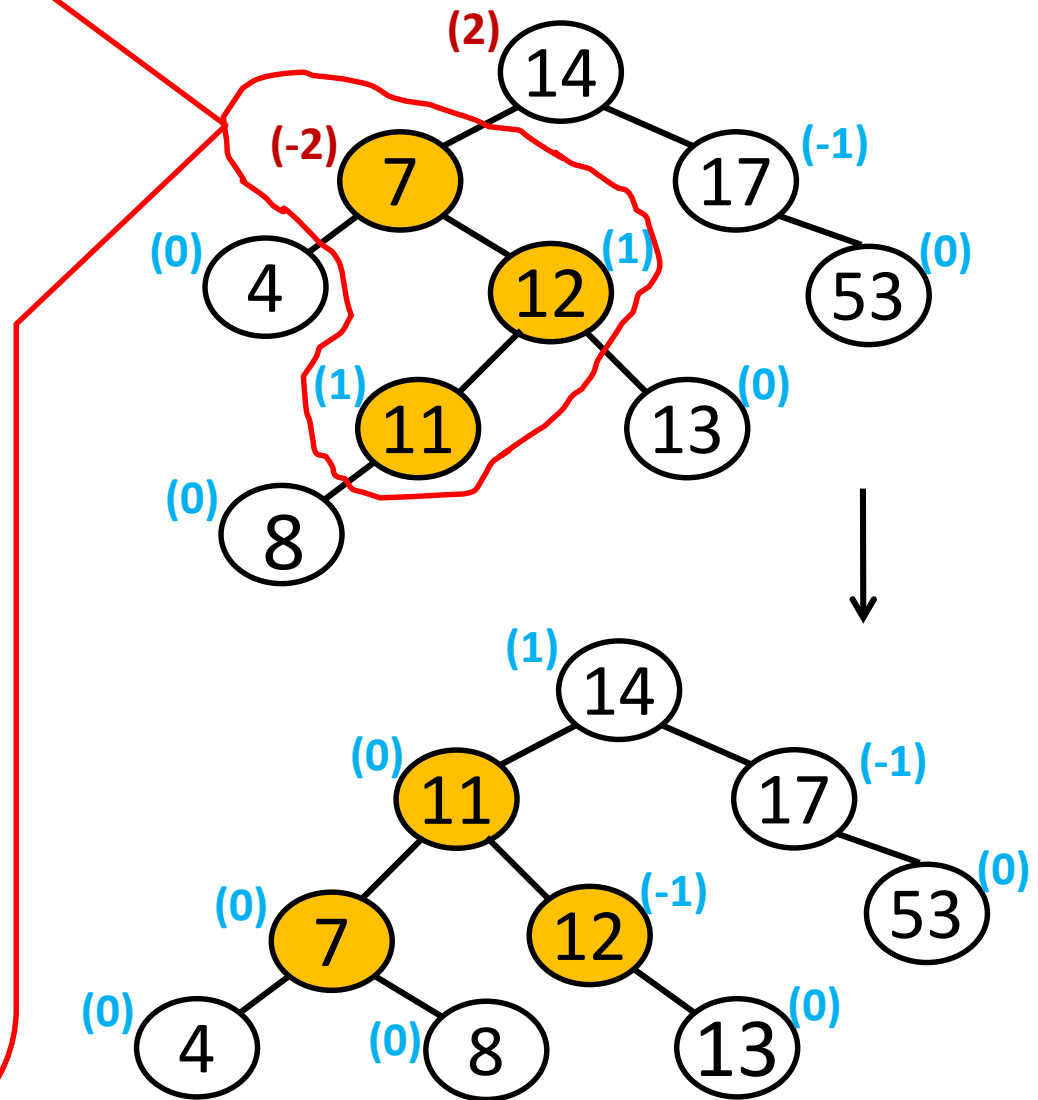
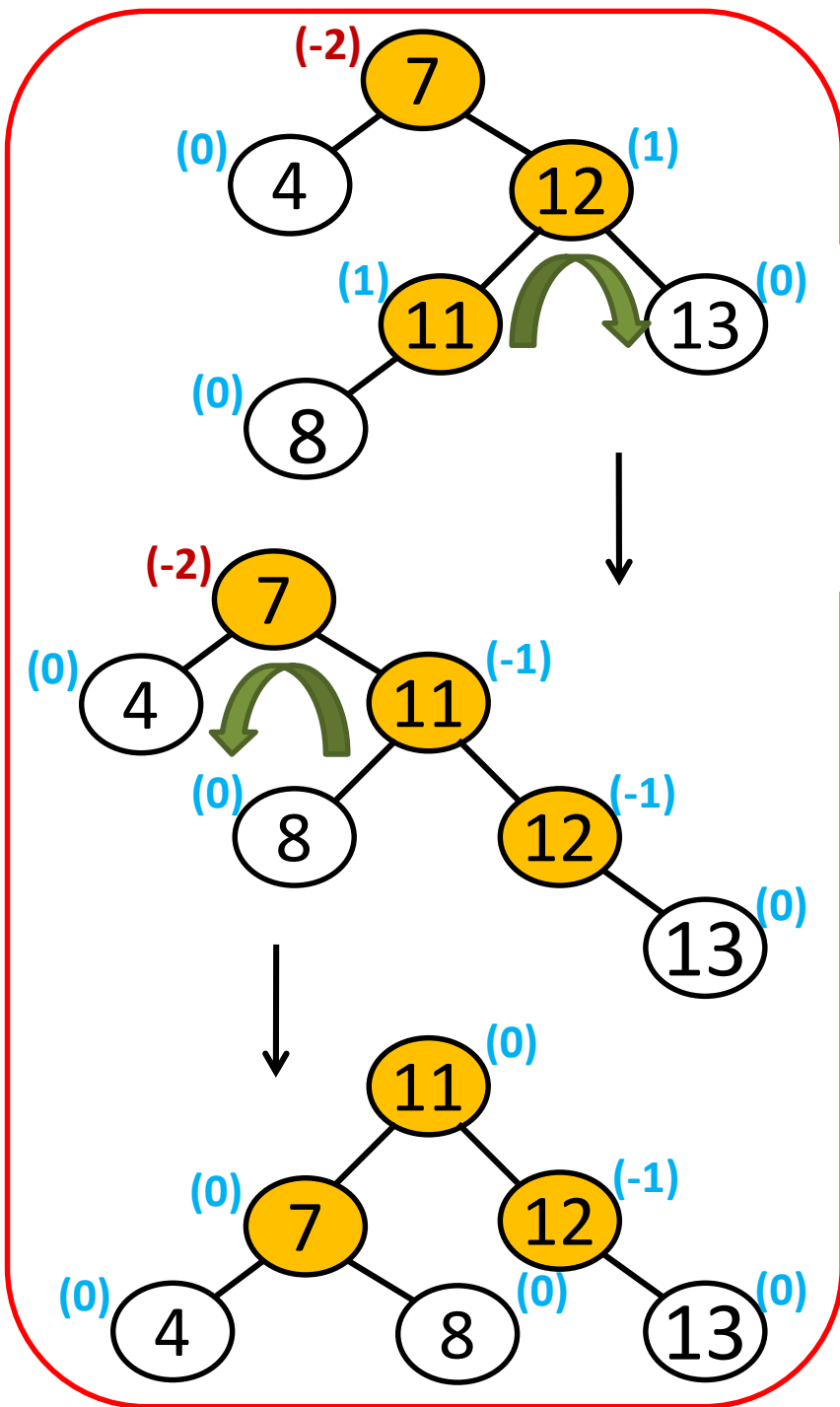
Example 1: Insert **14**, **17**, **11**, **7**, **53**, **4**, 13, 12, 8



Example 1: Insert **14**, **17**, **11**, **7**, **53**, **4**, **13**, **12**, 8

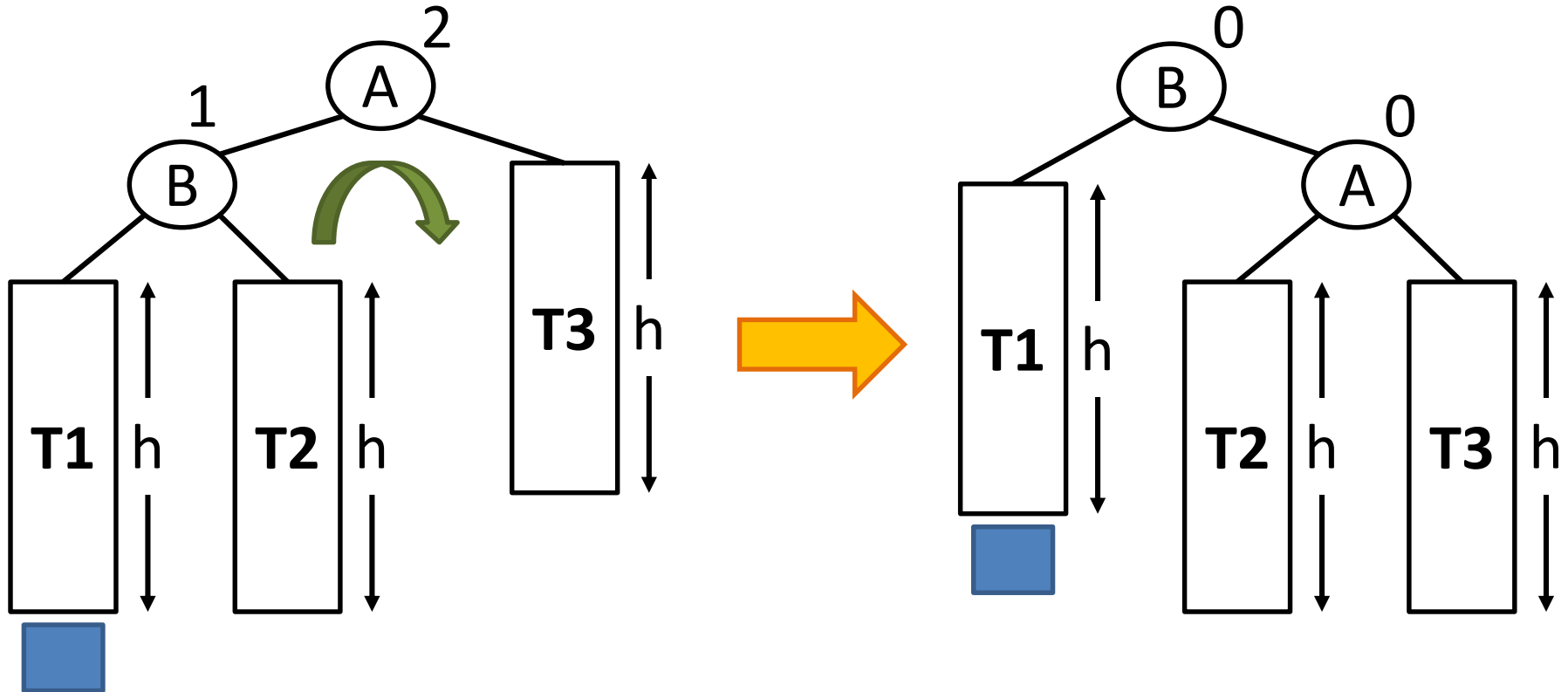


Example 1: Insert **14, 17, 11, 7, 53, 4, 13, 12, 8**



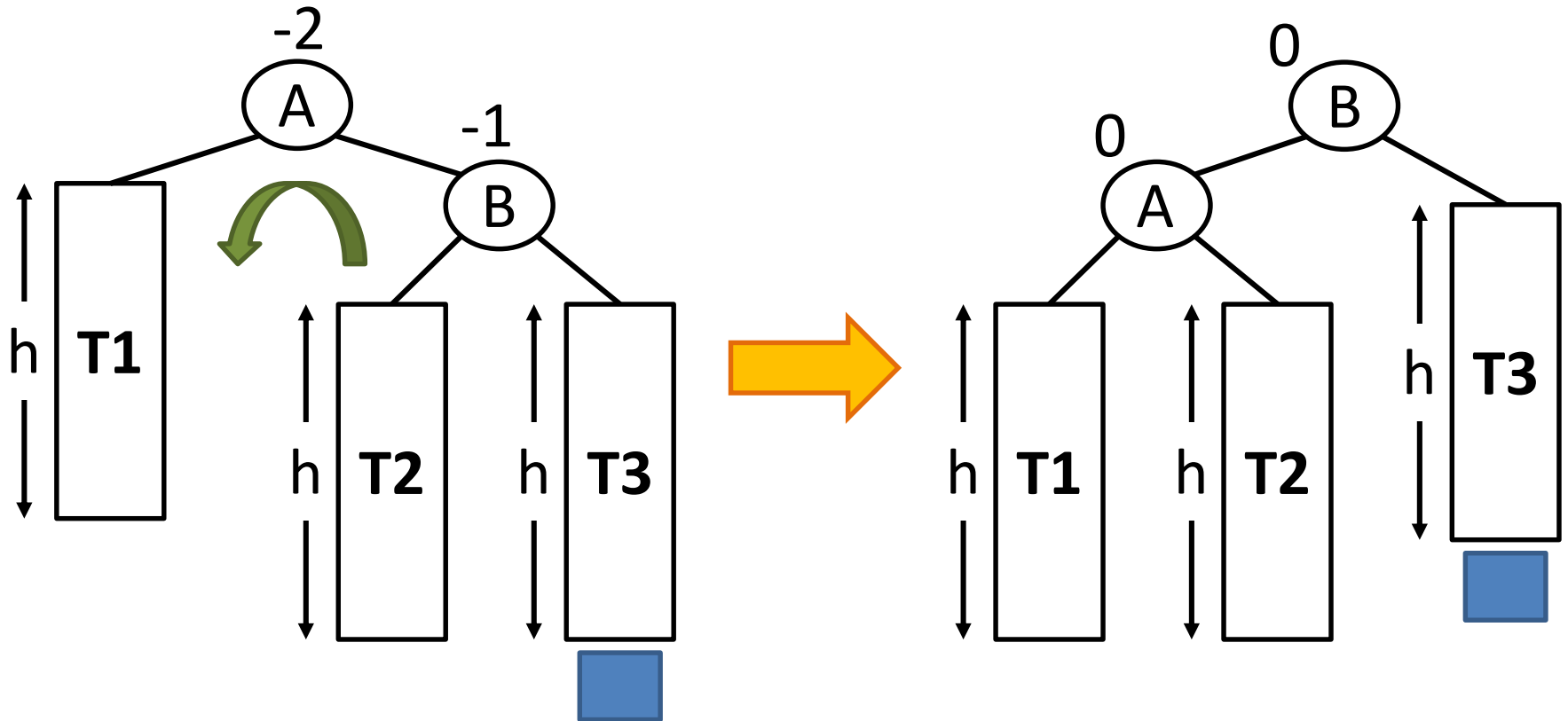
Case 1: Left of Left (Insertion)

BalFactor > 1 and *Key* < *Node* → *left* → *data*

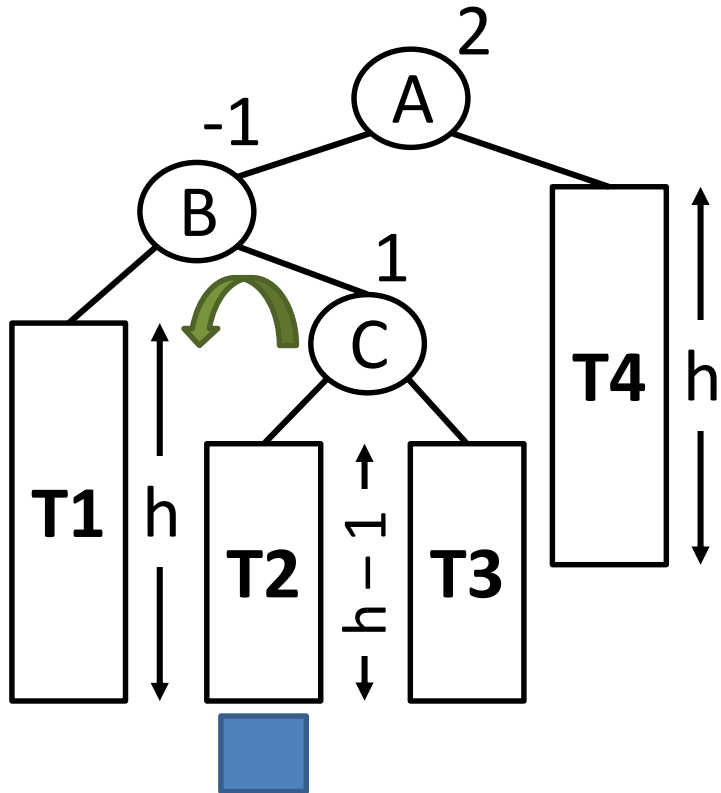


Case 2: Right of Right (Insertion)

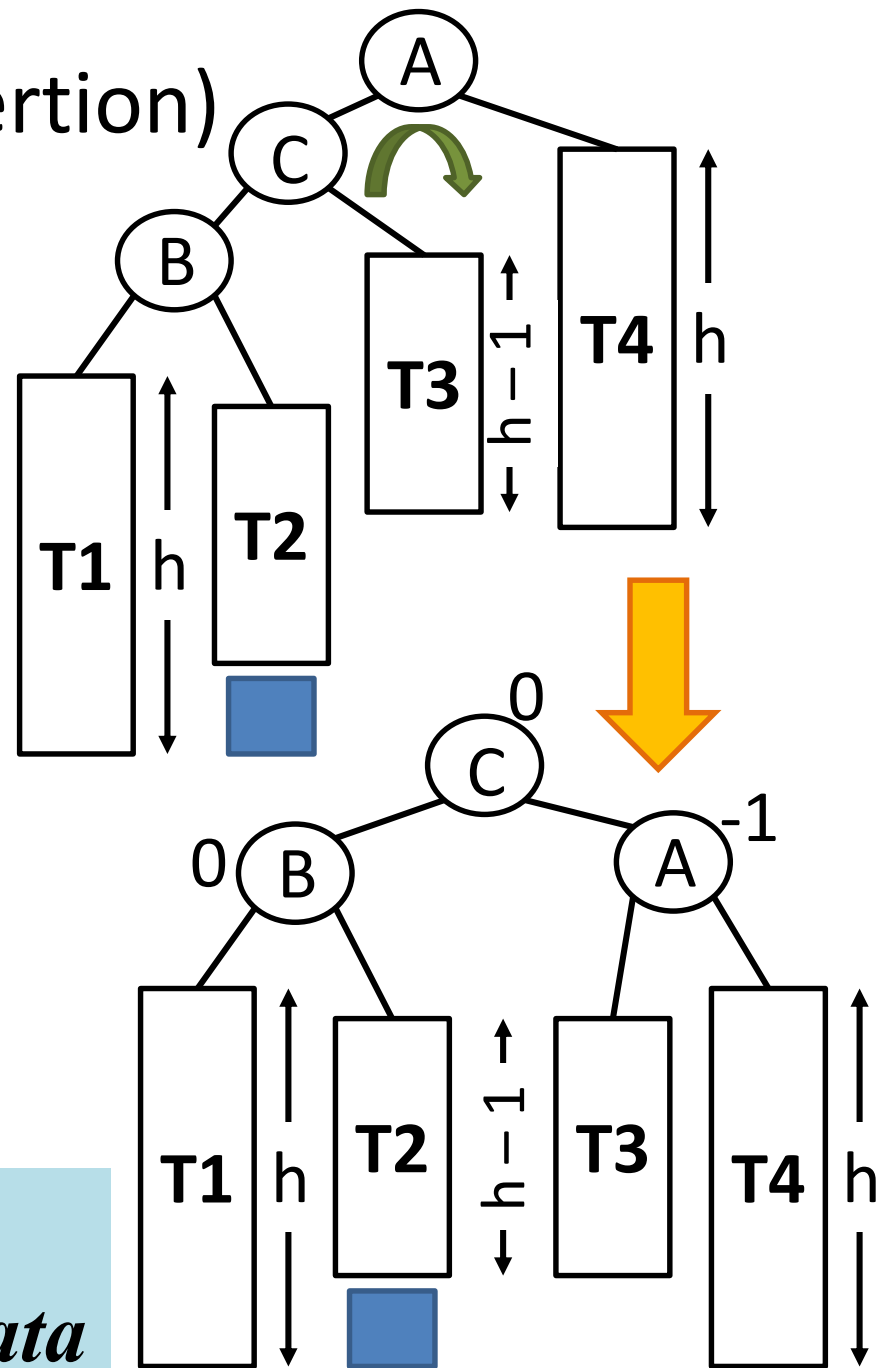
BalFactor < -1 and *Key* > *Node* → *right* → *data*



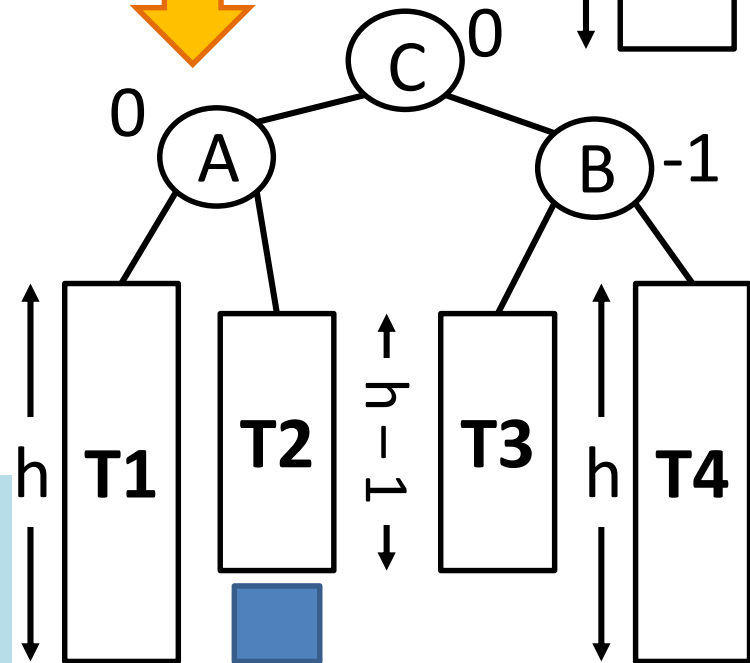
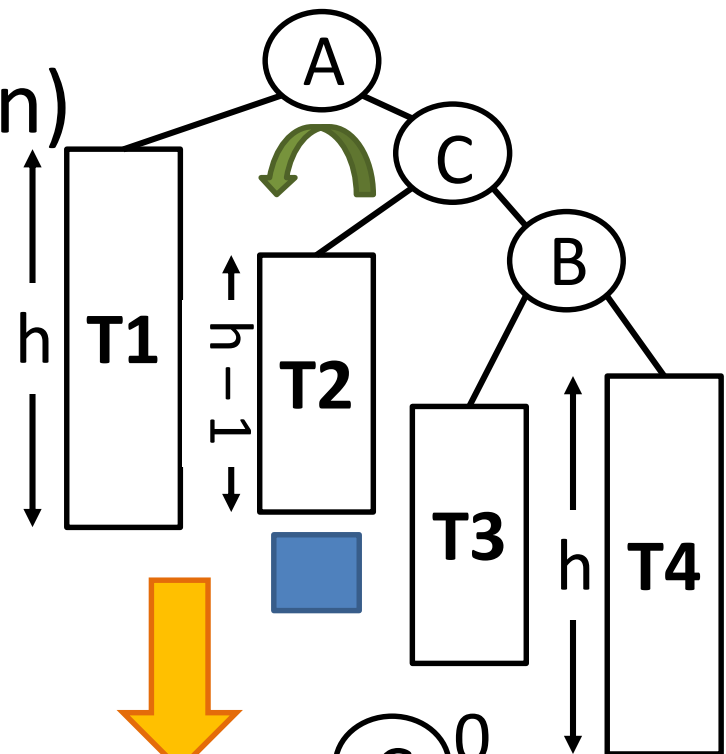
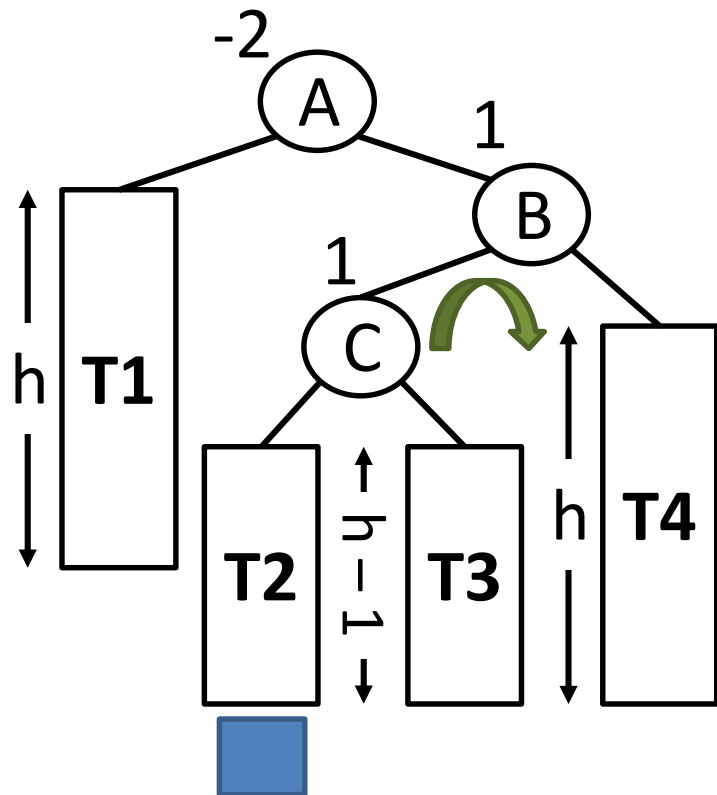
Case 3: Right of Left (Insertion)



***BalFactor* > 1 and**
Key* > *Node* → *left* → *data



Case 4: Left of Right (Insertion)



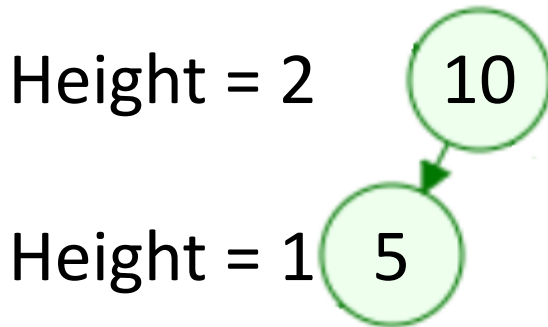
***BalFactor* < -1 and
Key < Node → right → data**

Insertion (recursion)

1. Insert the new-node with value *Key* using normal BST insertion.
2. Update height of the ancestor node, say *Node*.
3. Get the balance factor of this ancestor node, i.e. *Node*.
4. $BalFactor = (\text{height of } Node \rightarrow left - \text{height of } Node \rightarrow right)$.
5. If $BalFactor > 1$ and $Key < Node \rightarrow left \rightarrow data$
6. Return *rotateRight(Node)*. // Left of Left.

Example – 1

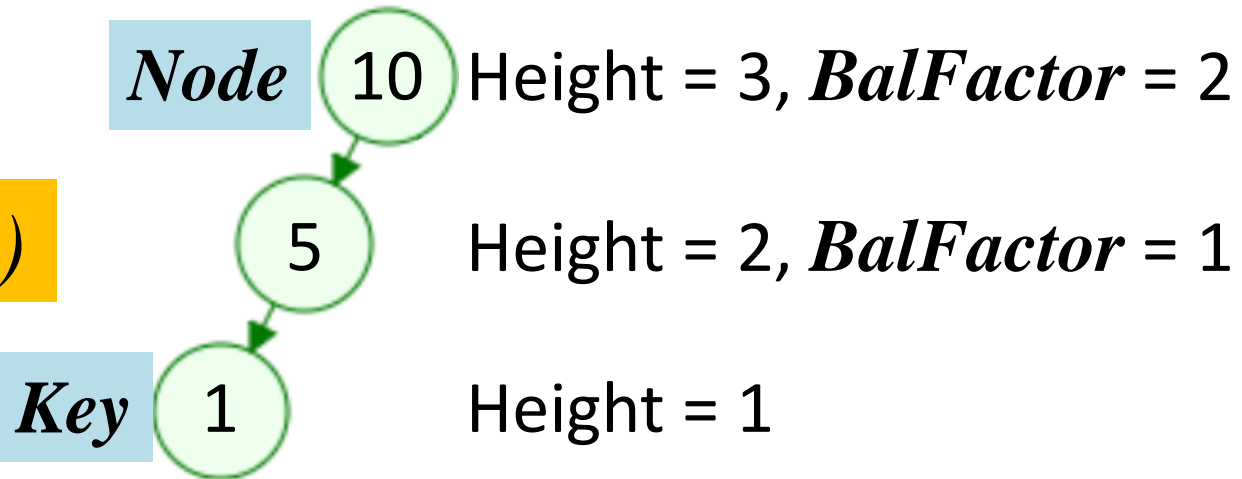
BalFactor > 1 and
Key < *Node* → *left* → *data*



Insert *Key* = 1

```
graph TD; A[Insert Key = 1] --> B[ ];
```

rotateRight(Node)

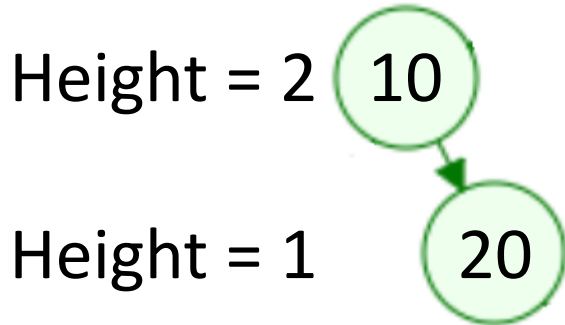


Contd...

7. If *BalFactor* < -1 and *Key* > *Node* → *right* → *data*
8. Return *rotateLeft(Node)*; // Right of Right.

Example – 2

BalFactor < -1 and
Key > *Node*→*right*→*data*

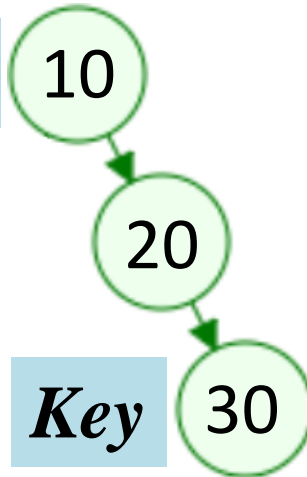


Insert *Key* = 30



rotateLeft(Node)

Node



Height = 3, *BalFactor* = -2

Height = 2, *BalFactor* = -1

Key

30

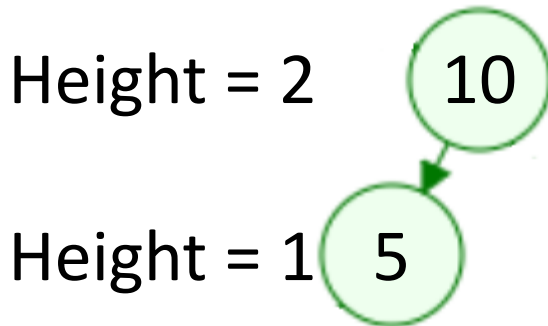
Height = 1

Contd...

7. If *BalFactor* < -1 and *Key* > *Node*→*right*→*data*
8. Return *rotateLeft(Node)*; // Right of Right.
9. If *BalFactor* > 1 and *Key* > *Node*→*left*→*data*
10. *Node*→*left* = *rotateLeft(Node*→*left)*;
11. Return *rotateRight(Node)*; // Right of Left.

Example – 3

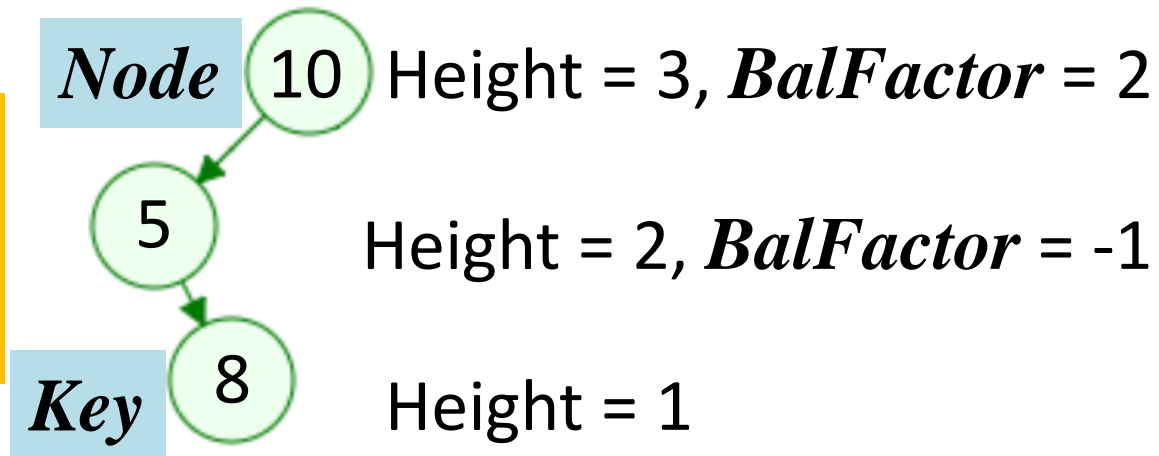
BalFactor > 1 and
Key > *Node*→*left*→*data*



Insert *Key* = 8

```
graph TD; A[Insert Key = 8] --> B[ ];
```

Node→*left* =
rotateLeft(Node→*left)*;
rotateRight(Node);

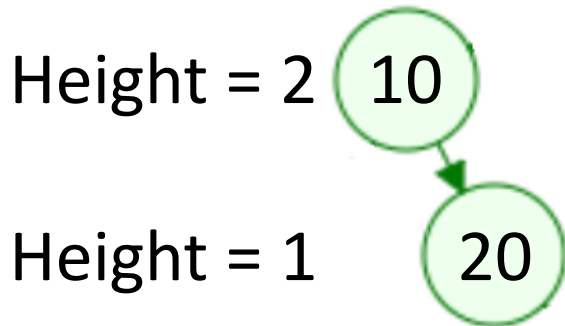


Contd...

7. If *BalFactor* < -1 and *Key* > *Node*→*right*→*data*
8. Return *rotateLeft(Node)*; // Right of Right.
9. If *BalFactor* > 1 and *Key* > *Node*→*left*→*data*
10. *Node*→*left* = *rotateLeft(Node*→*left)*;
11. Return *rotateRight(Node)*; // Right of Left.
12. If *BalFactor* < -1 and *Key* < *Node*→*right*→*data*
13. *Node*→*right* = *rotateRight(Node*→*right)*;
14. Return *rotateLeft(Node)*; // Left of Right.

Example – 4

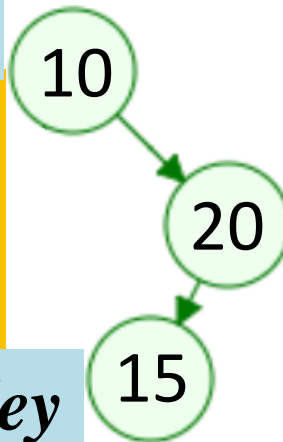
BalFactor < -1 and
Key < *Node*→*right*→*data*



Insert *Key* = 15



Node



Height = 3, *BalFactor* = -2

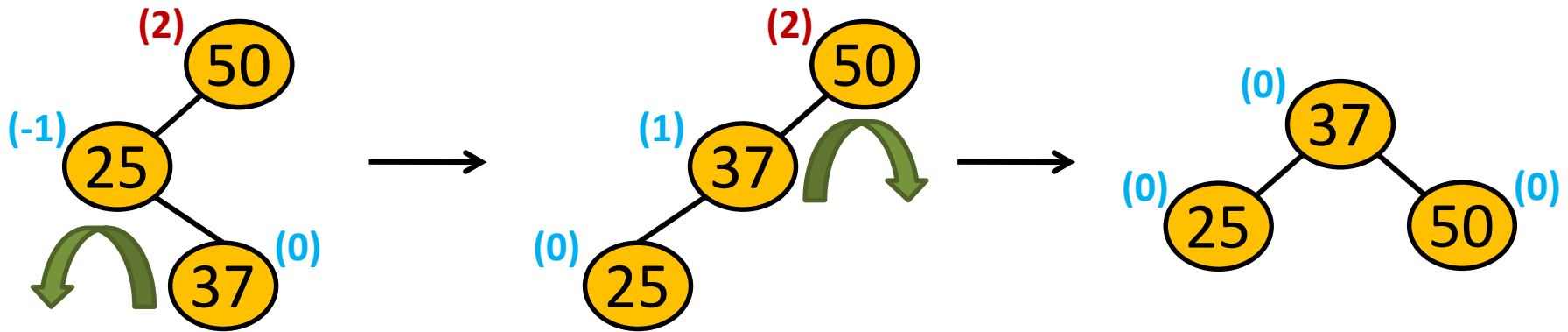
Height = 2, *BalFactor* = 1

Height = 1

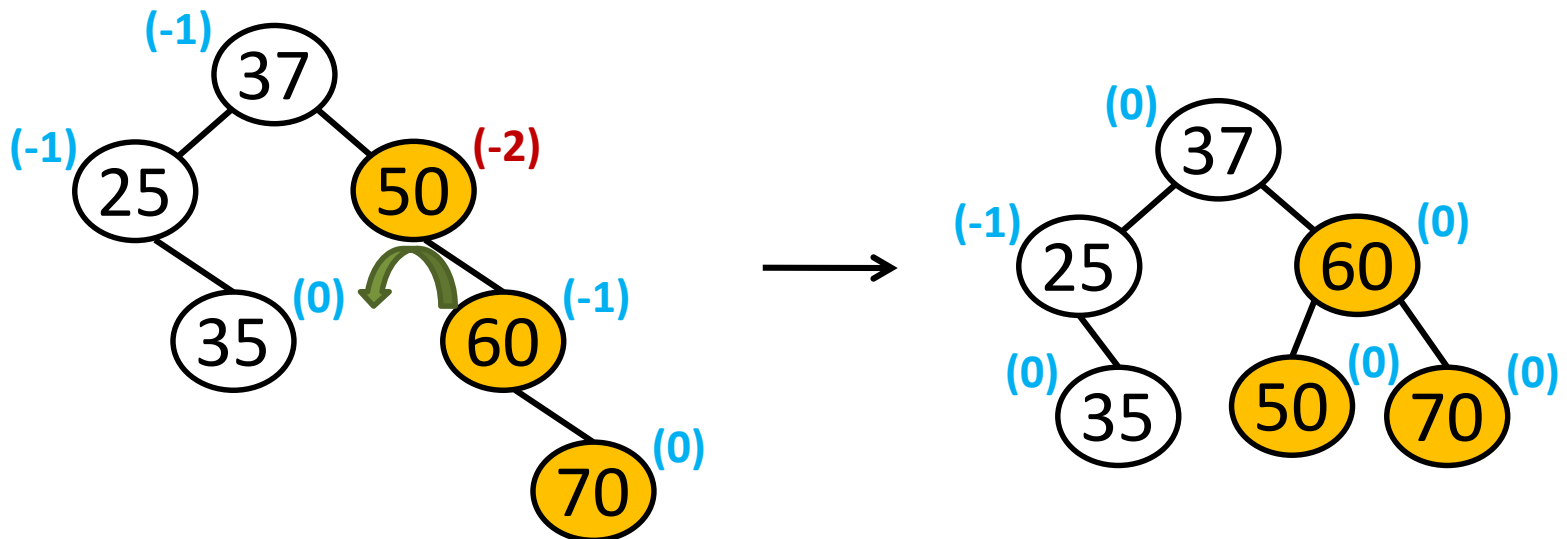
Node→*right* =
rotateRight(Node→*right*);
rotateLeft(Node);

Example 2: 50, 25, 37, 35, 60, 70, 30, 45, 34, 40, 55

- Insert 50, 25, 37

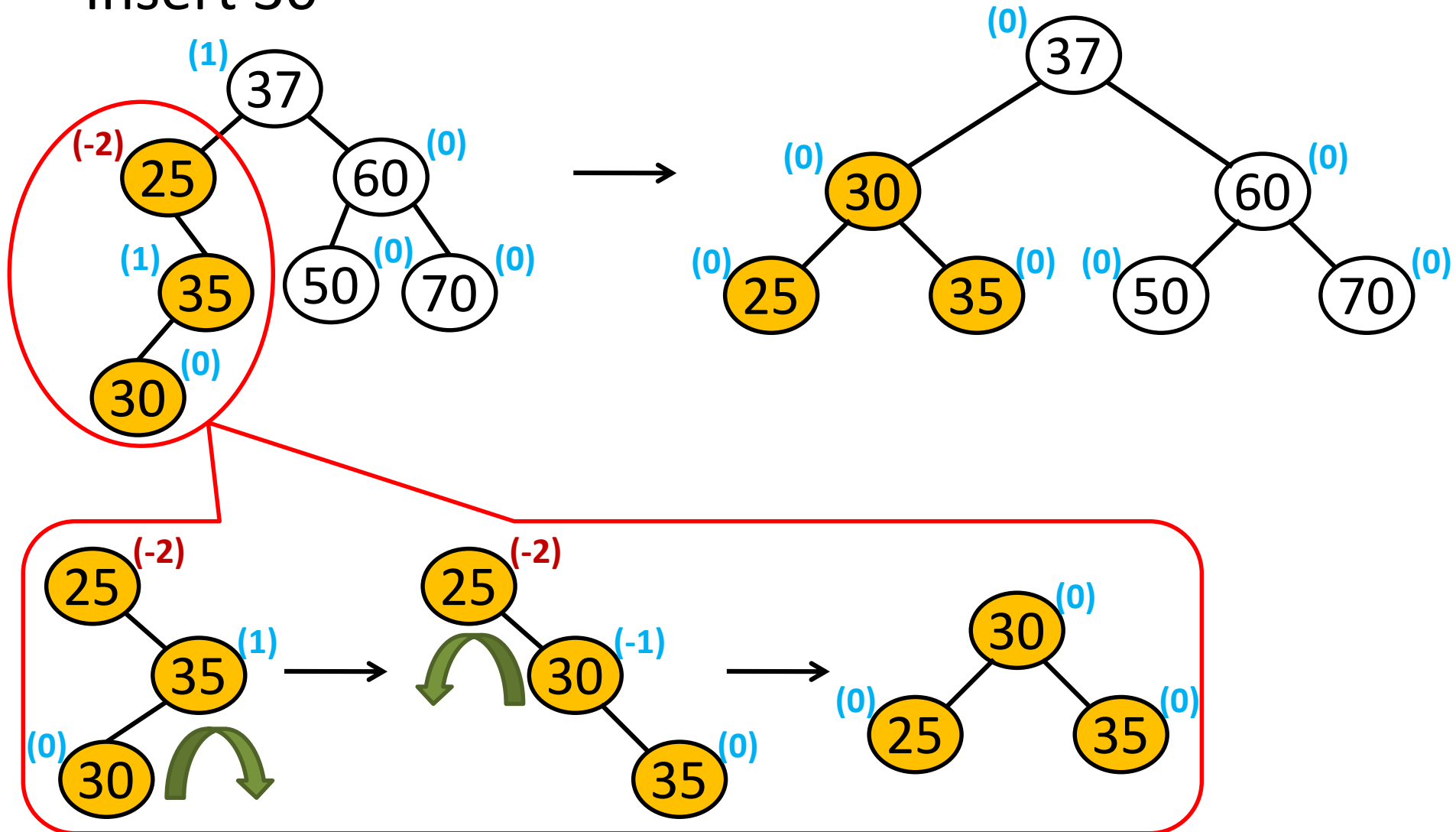


- Insert 35, 60, 70



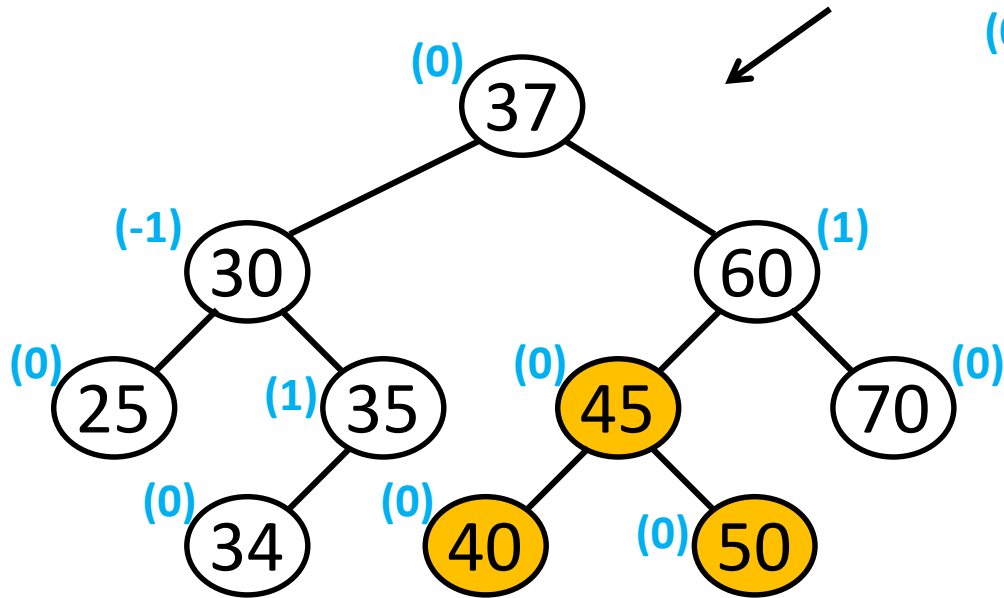
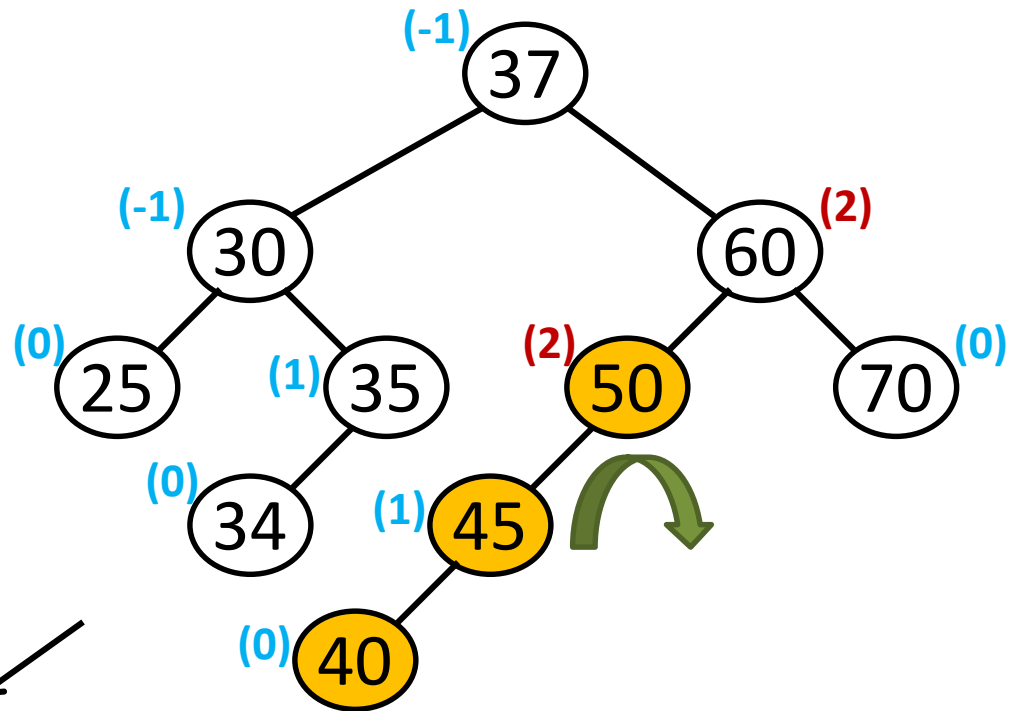
Contd...

- Insert 30

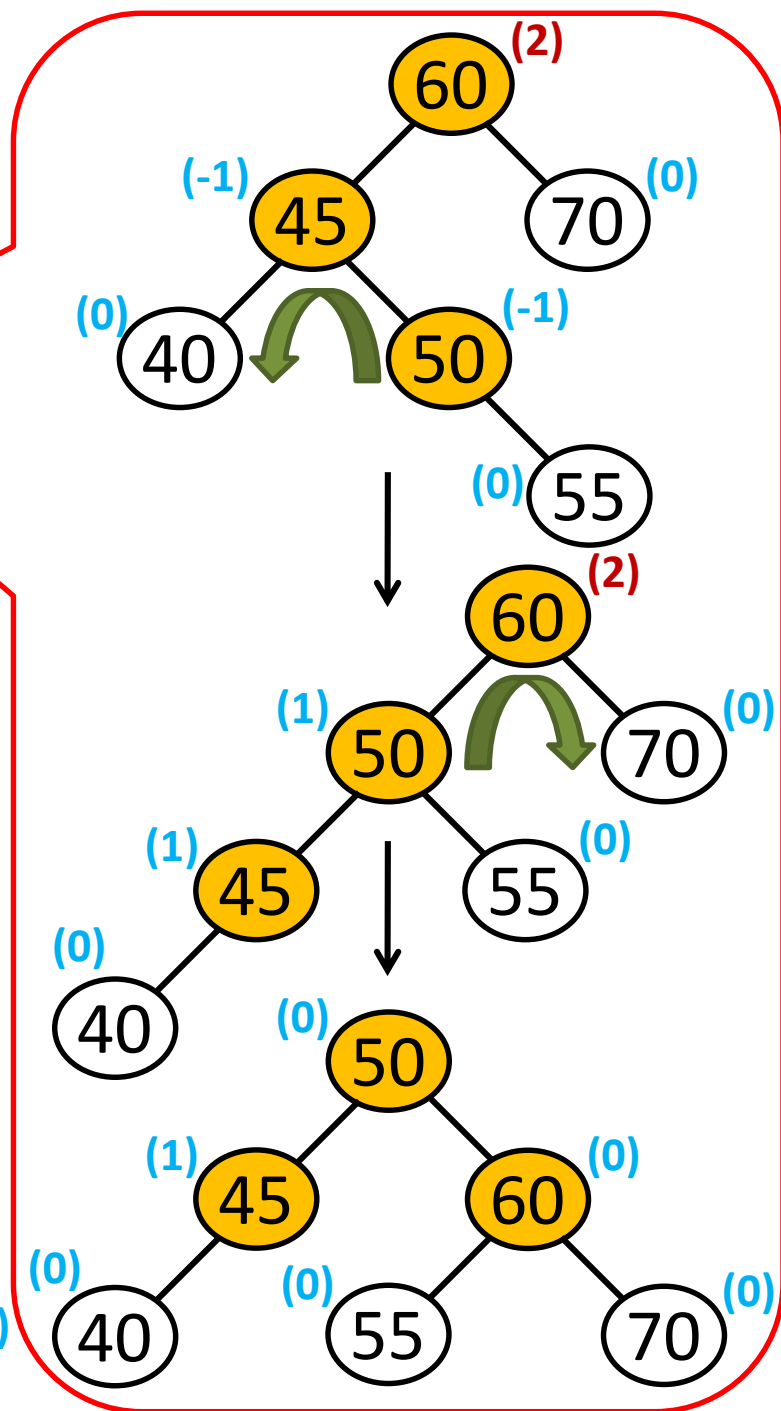
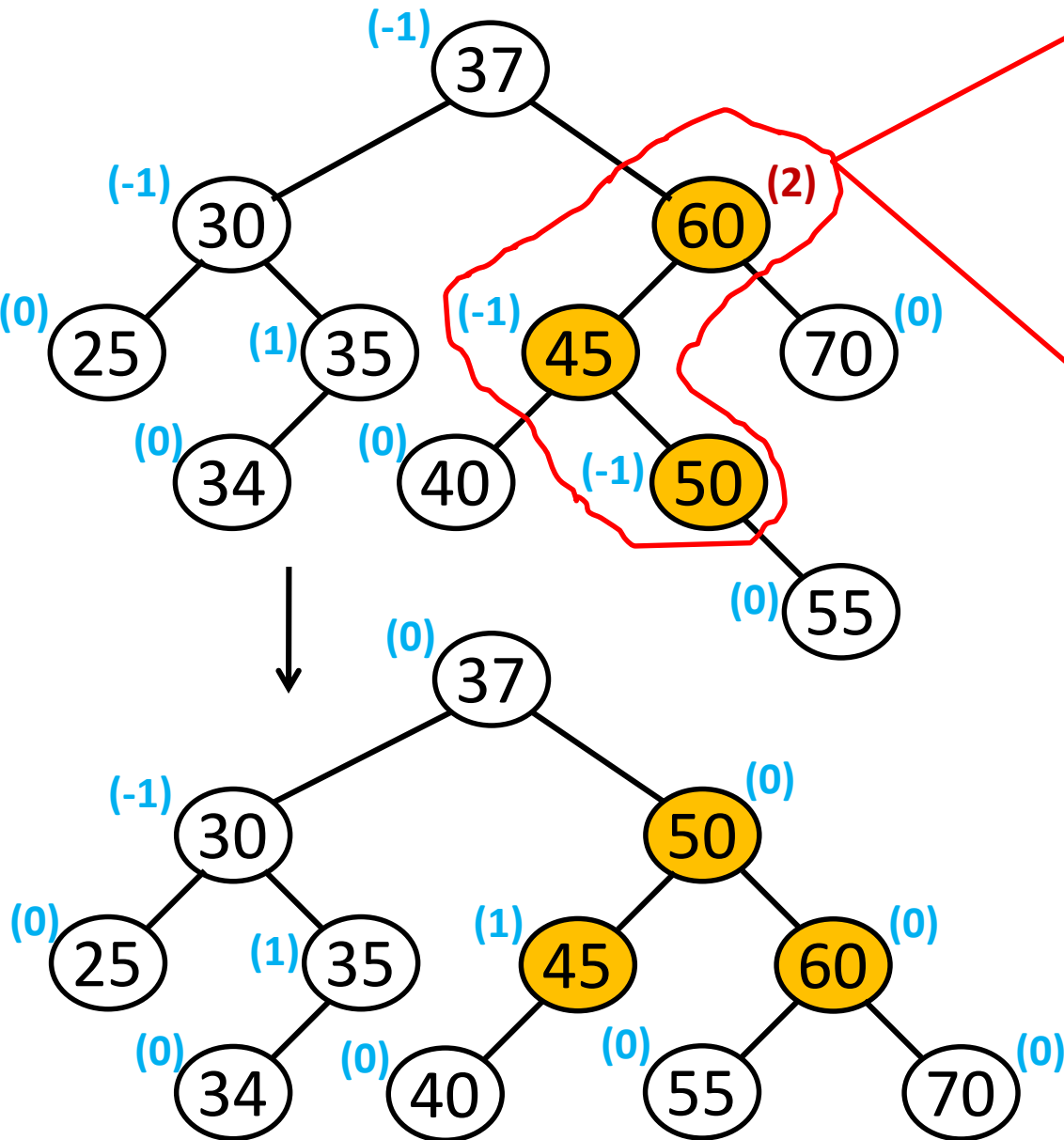


Contd...

- Insert 45, 34, 40



Contd... • Insert 55



AVL Trees

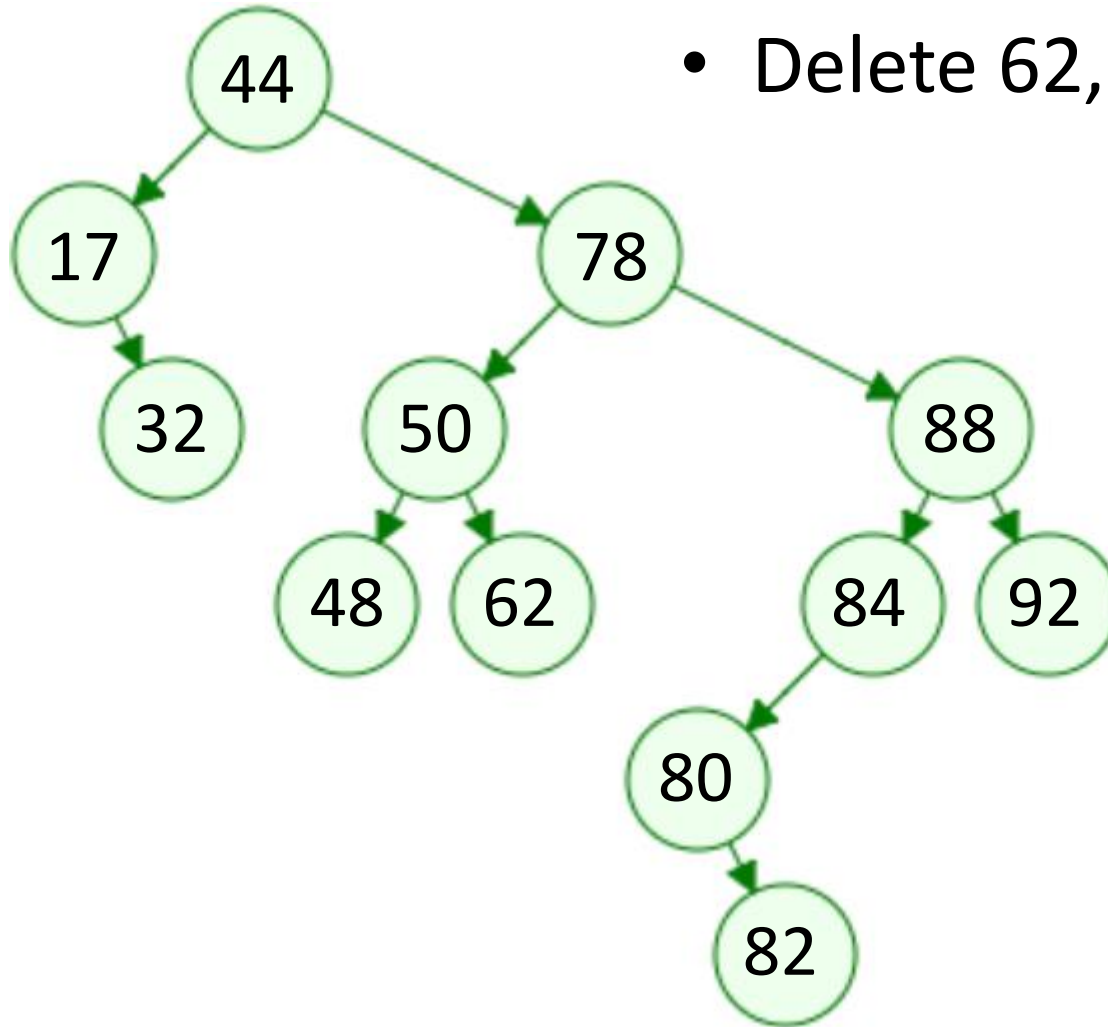
Deletion

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/predecessor of the node.
 - Copy contents of the inorder successor/predecessor to the node being removed.
 - Delete the inorder successor/predecessor from the right/left subtree.

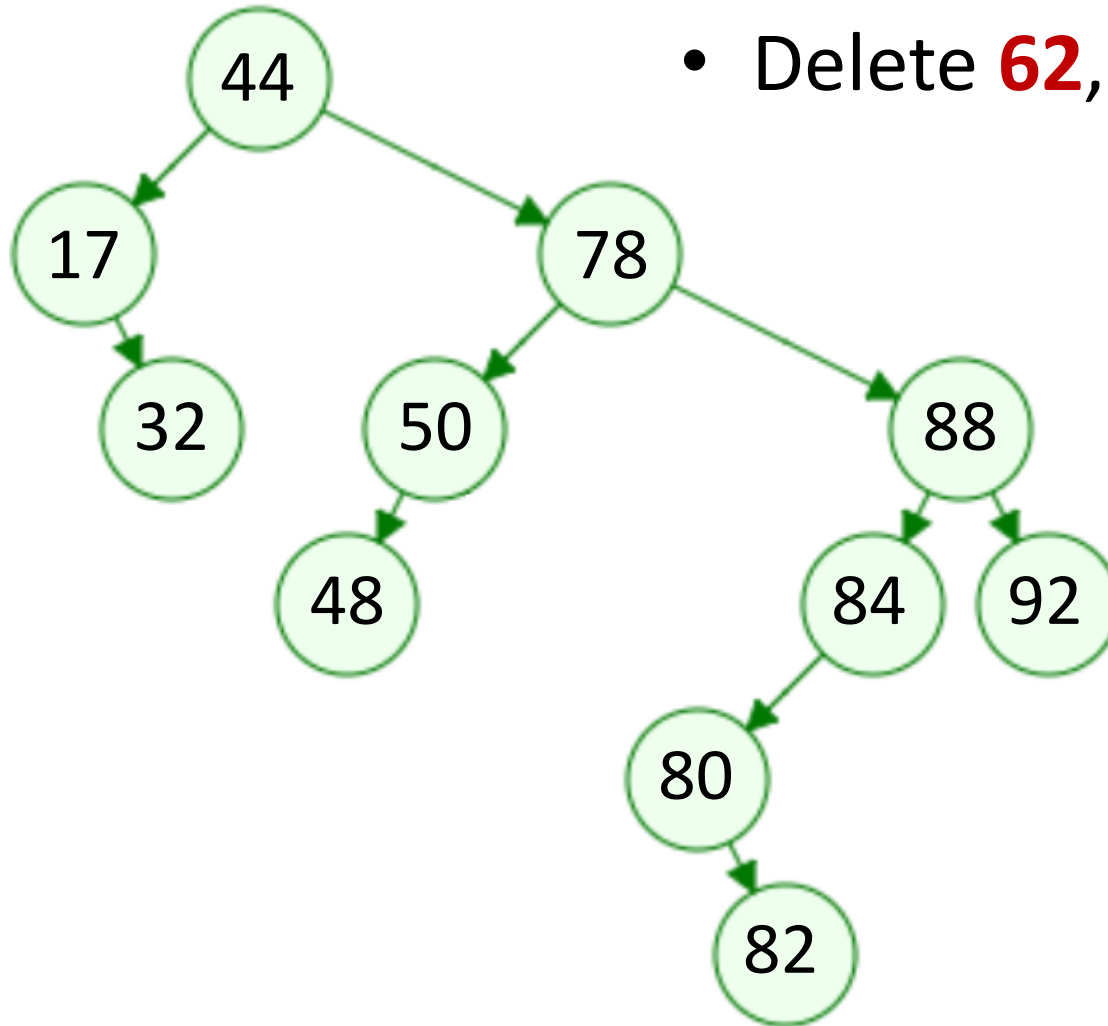
Example – BST Deletion

- Delete 62, 17, and 78



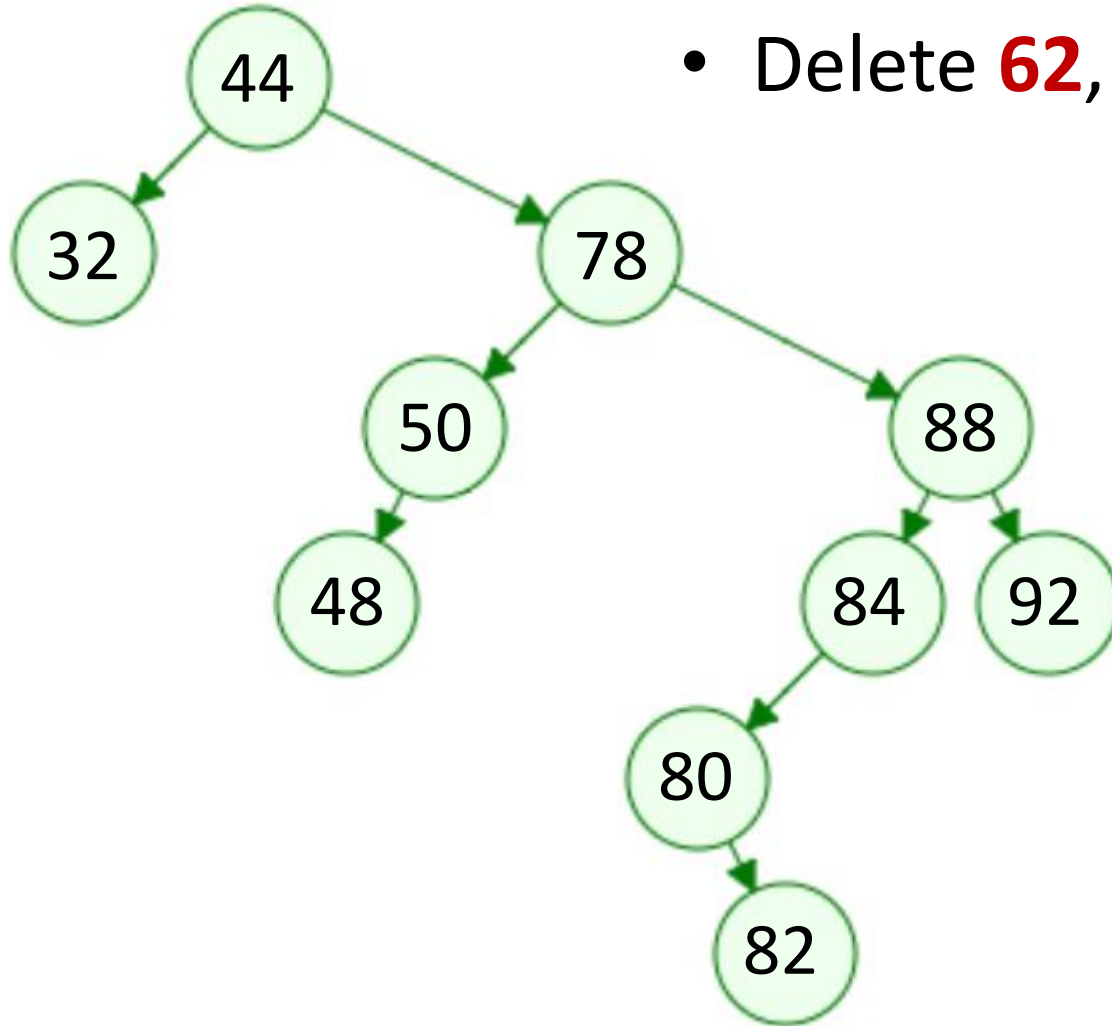
After deleting 62

- Delete **62**, 17, and 78



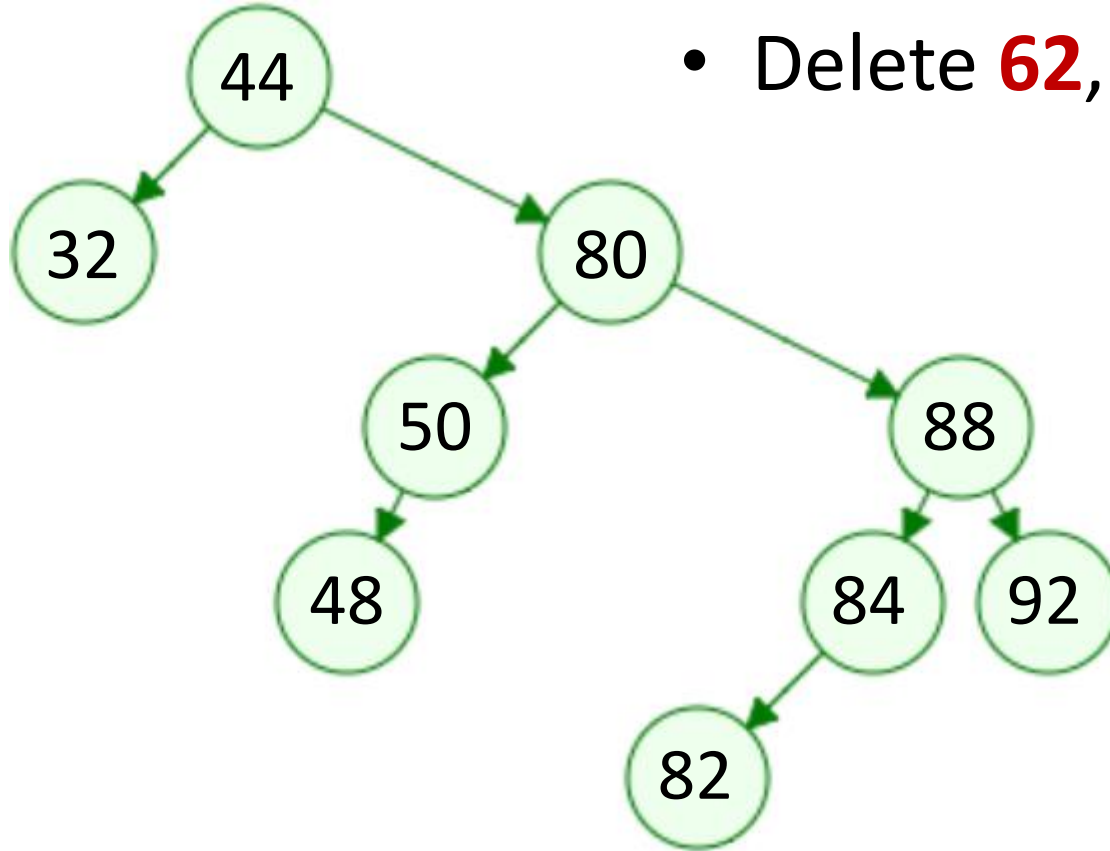
After deleting 17

- Delete **62**, **17**, and 78



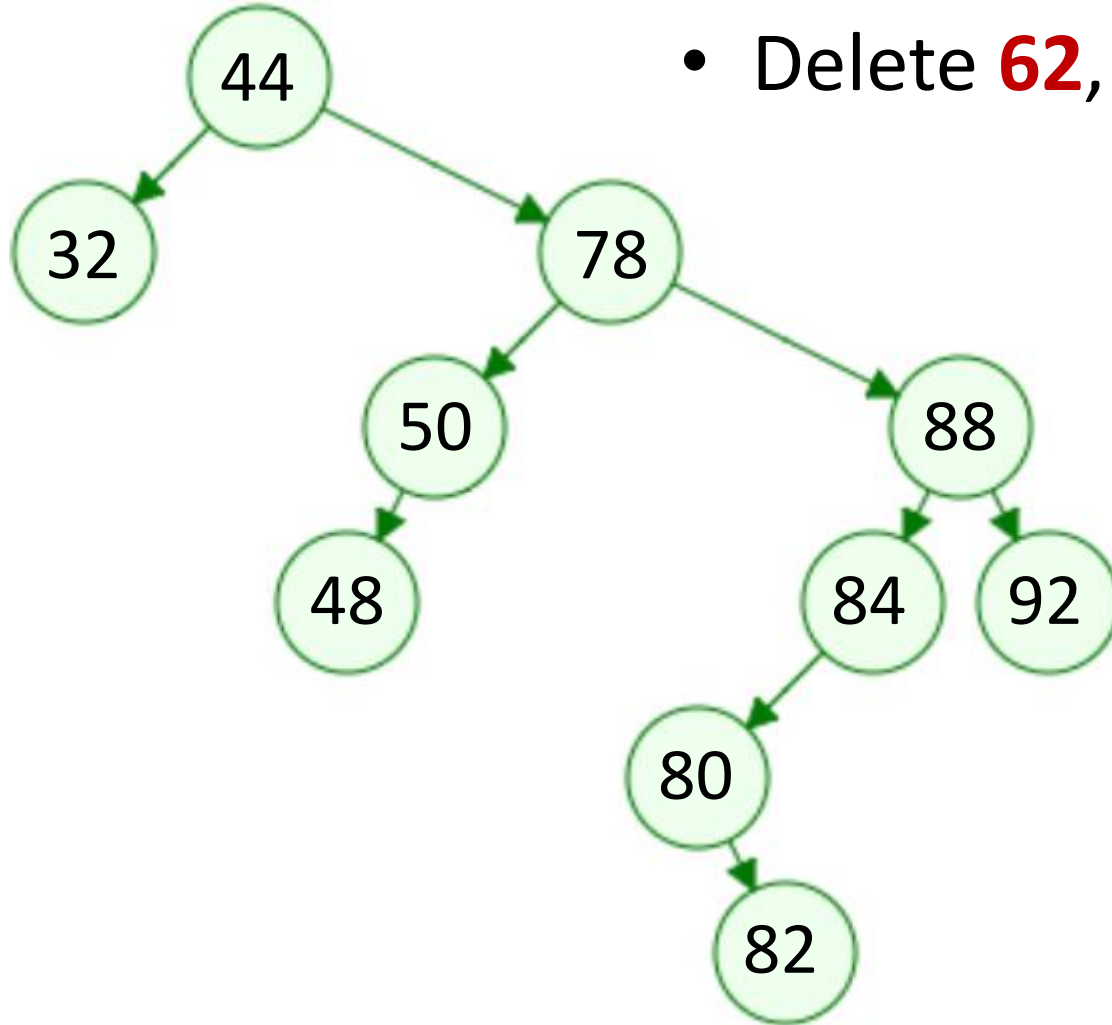
After deleting 78 (Using successor)

- Delete **62**, **17**, and **78**



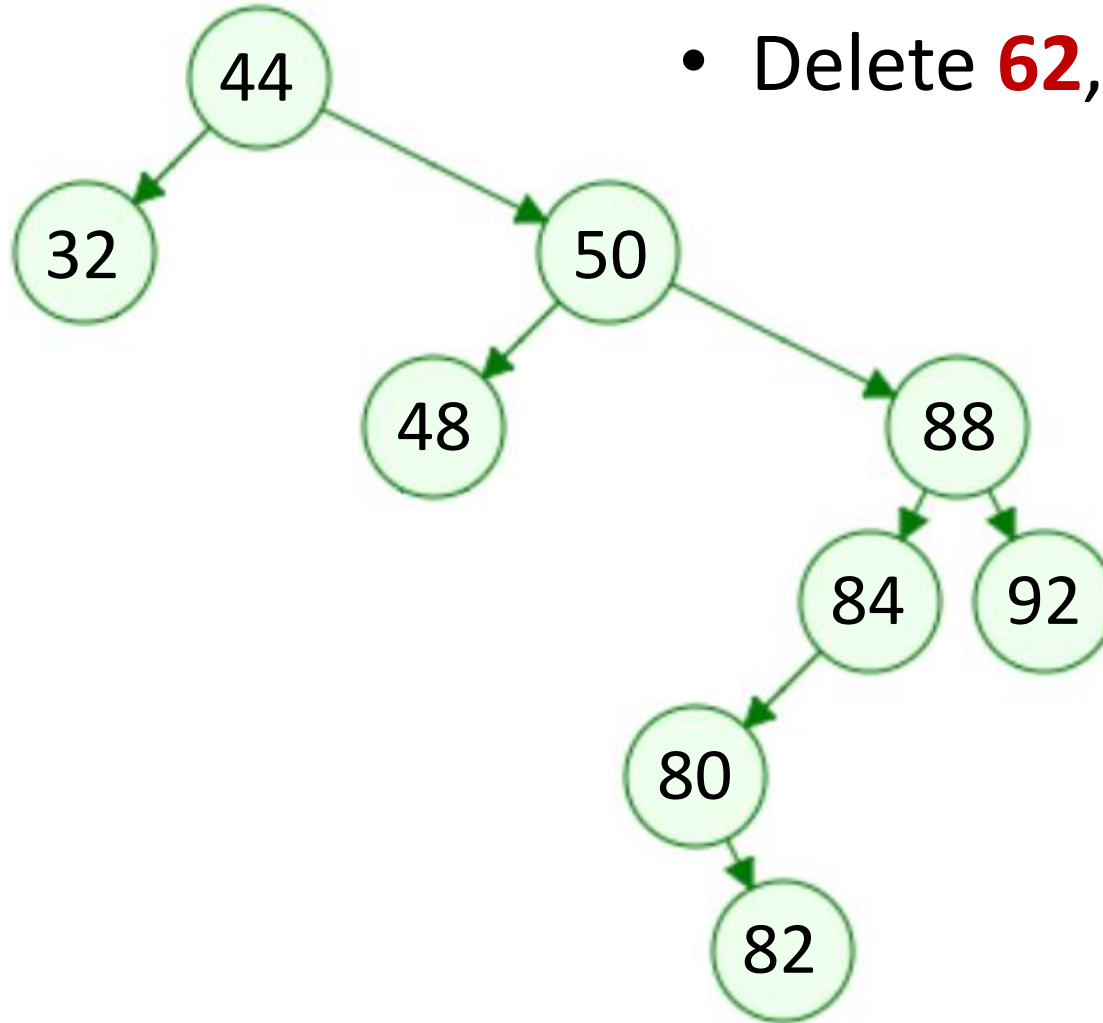
After deleting 17

- Delete **62**, **17**, and 78

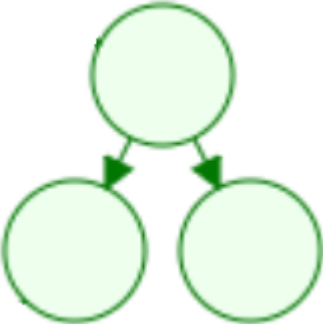
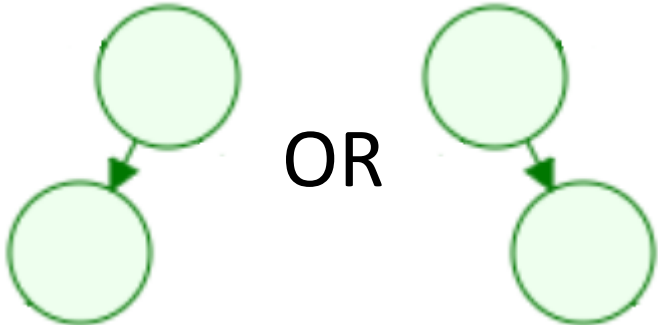


After deleting 78 (Using predecessor)

- Delete **62**, **17**, and **78**

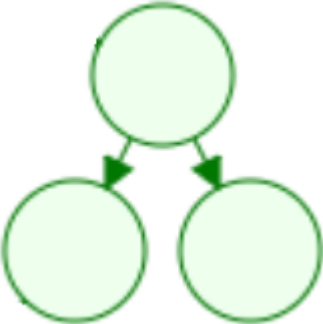
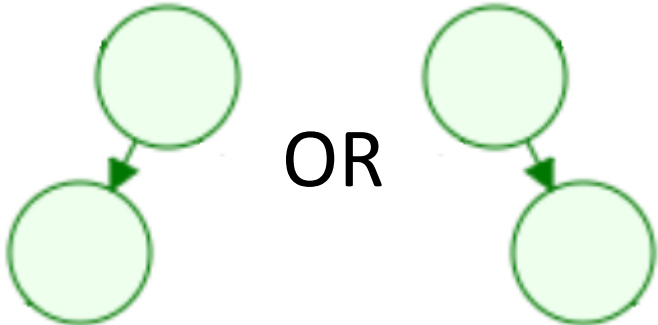
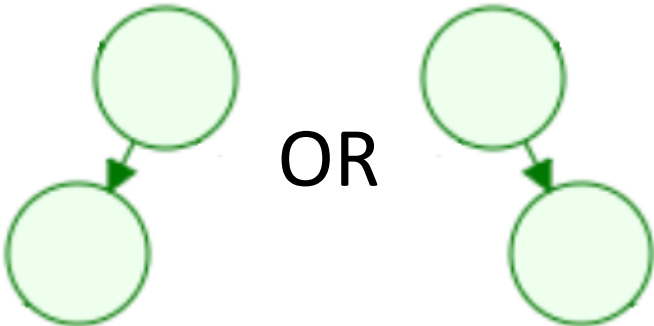



AVL Deletion

	Before deletion	After deletion
<ul style="list-style-type: none">Scenario 1	 <pre>graph TD; A(()) --> B(()); A --> C(());</pre>	 <pre>graph TD; A1(()) --> B1(()); A2(()) --> C2(());</pre>

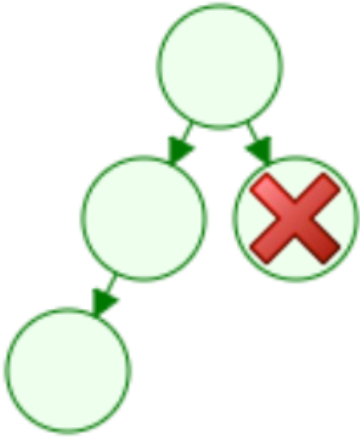
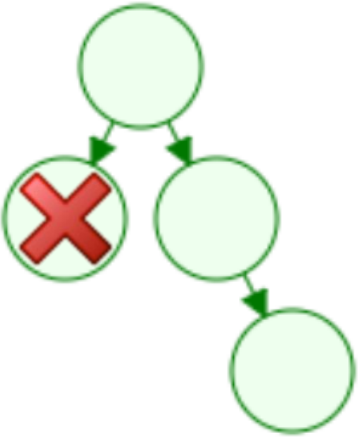
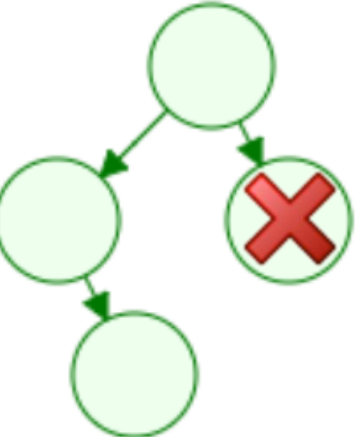
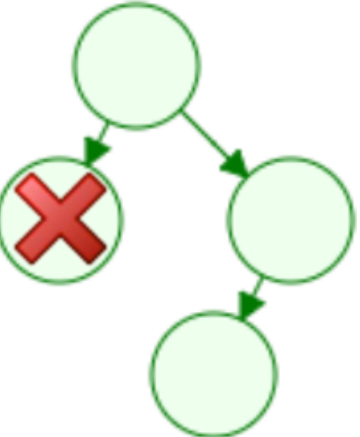
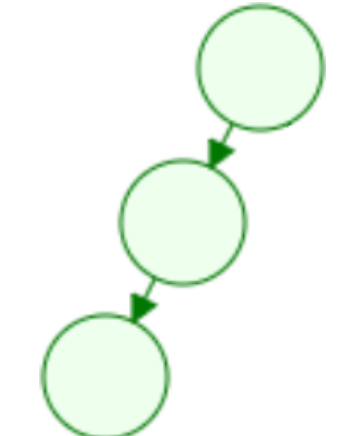
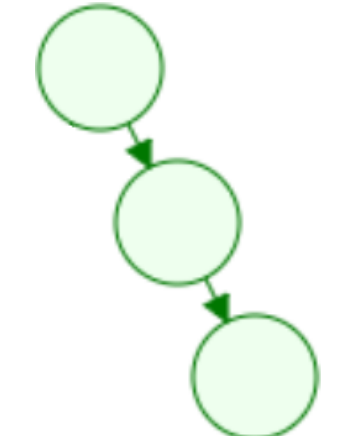
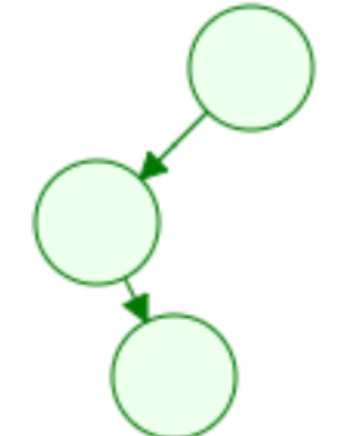
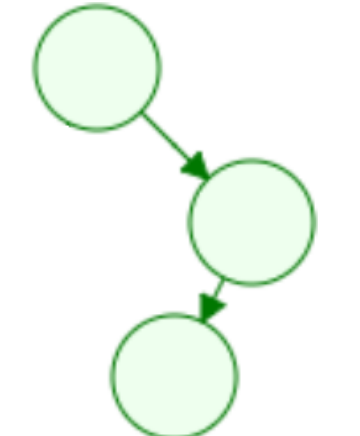
AVL Deletion

REQUIRES NO ROTATION

	Before deletion	After deletion
• Scenario 1	 <pre>graph TD; A(()) --> B(()); A --> C(());</pre>	 <pre>graph TD; A1(()) --> B1(()); A2(()) --> C2(());</pre>
• Scenario 2	 <pre>graph TD; A3(()) --> B3(()); A4(()) --> C4(());</pre>	 <pre>graph TD; D(());</pre>

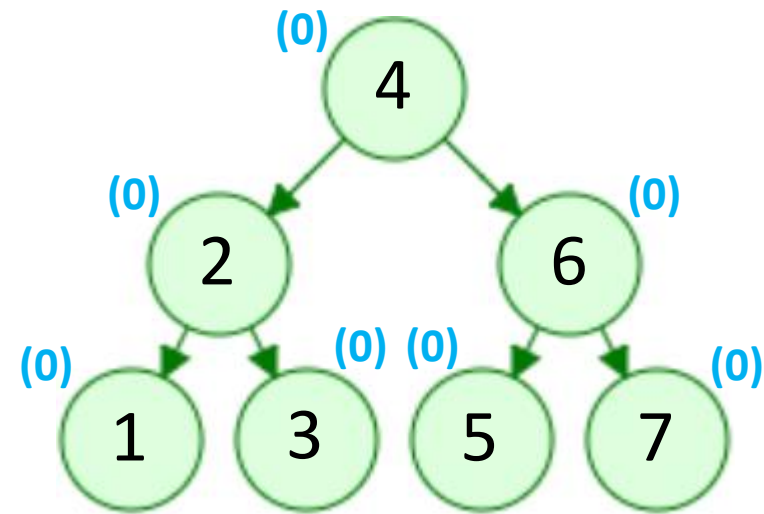
- Scenario 3

Contd...

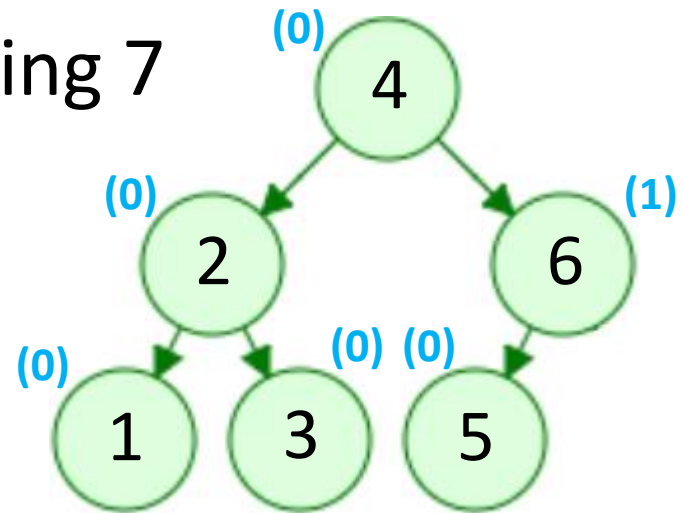
Before deletion				
After deletion				
Rotations	Right	Left	Left then Right	Right then Left

Example – 1

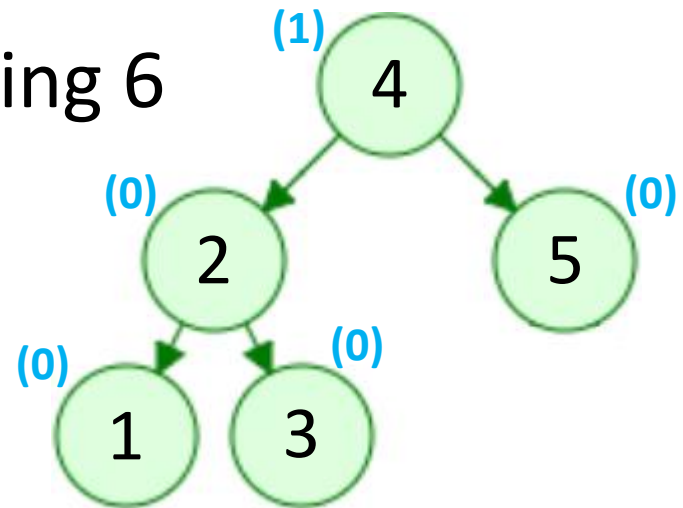
- Delete 7, 6, 5



After deleting 7

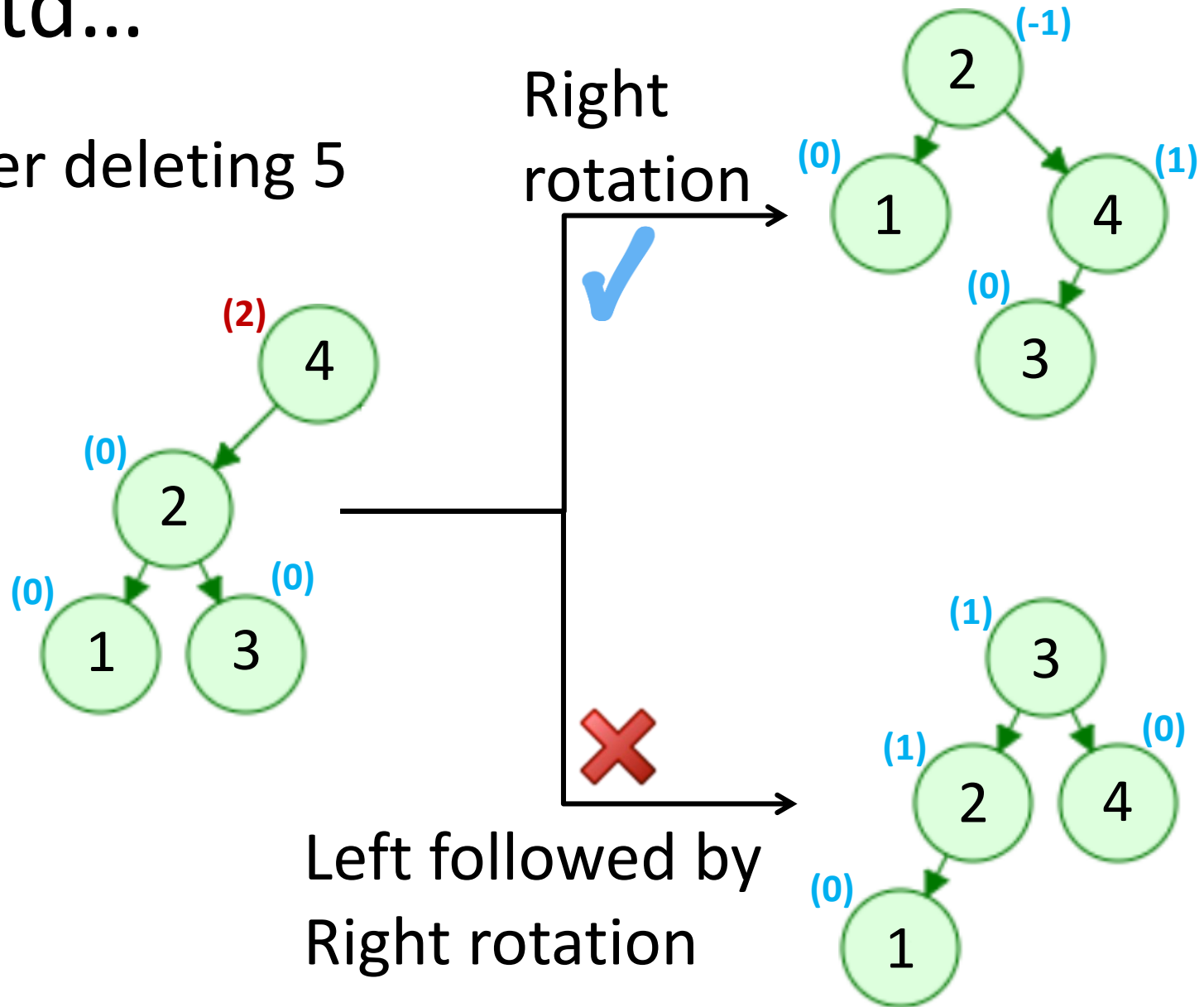


After deleting 6



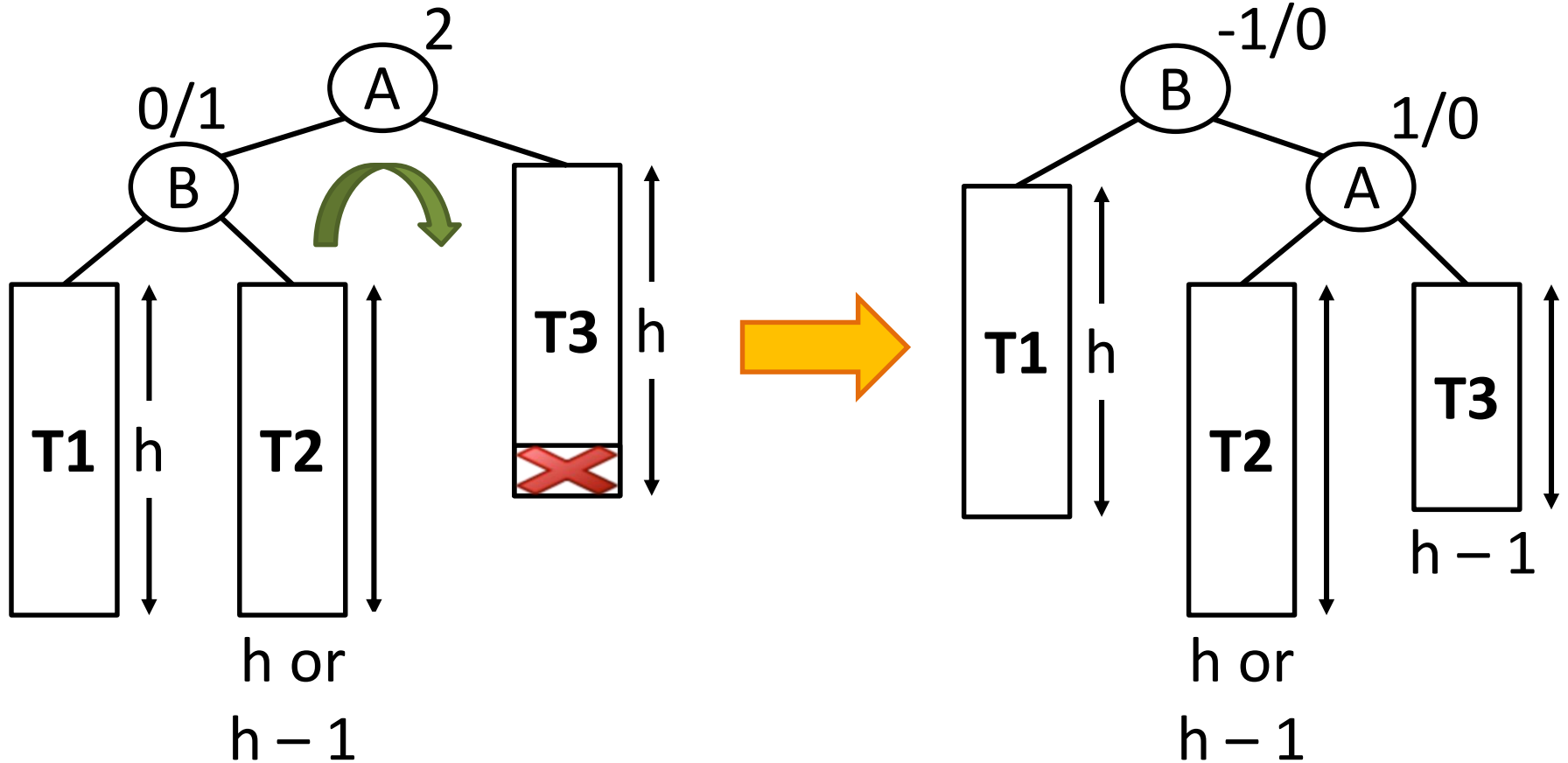
Contd...

- After deleting 5



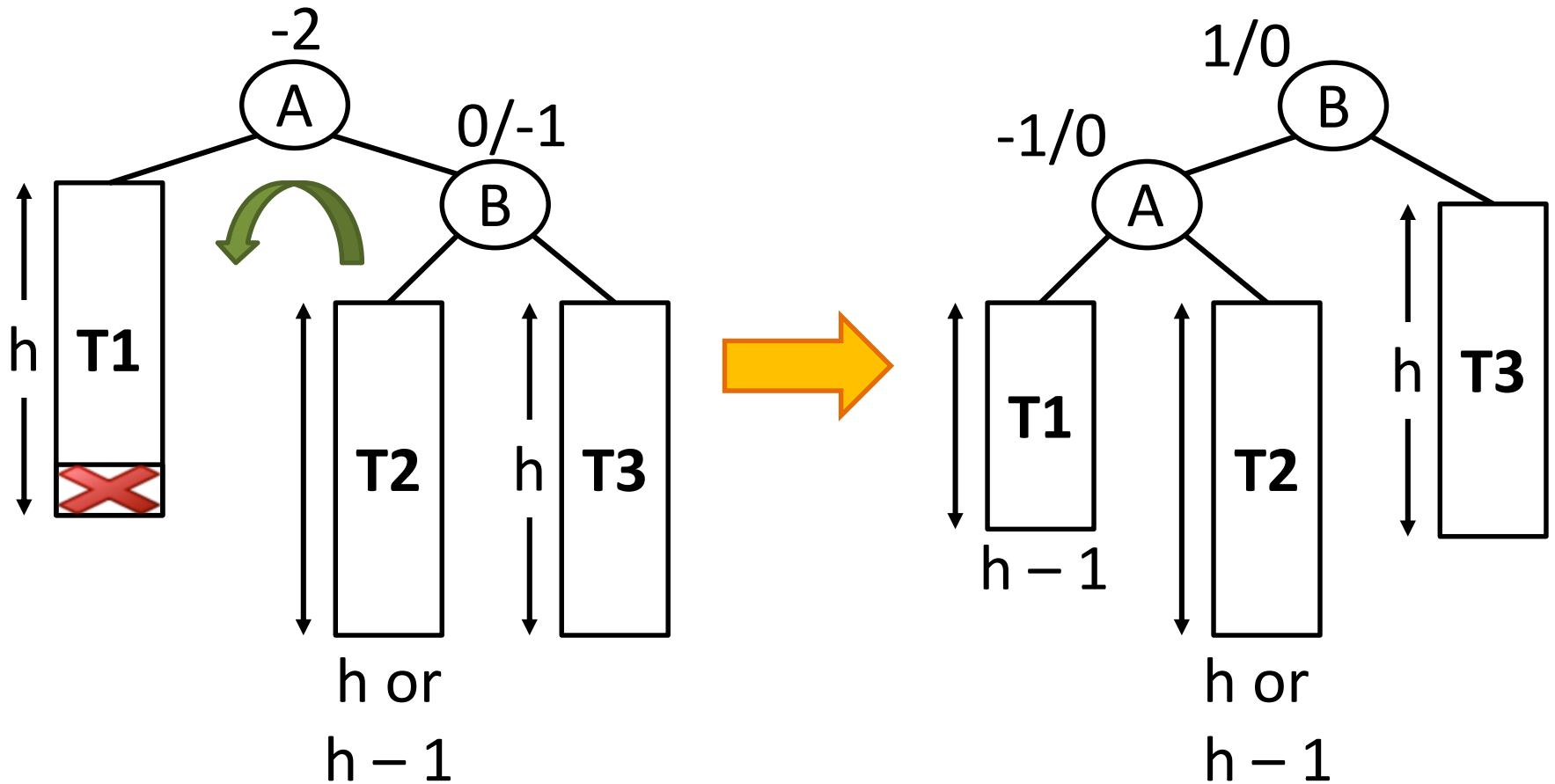
Case 1: Left of Left (Deletion)

***BalFactor* > 1 and *balance(Node*→*left*) >= 0**

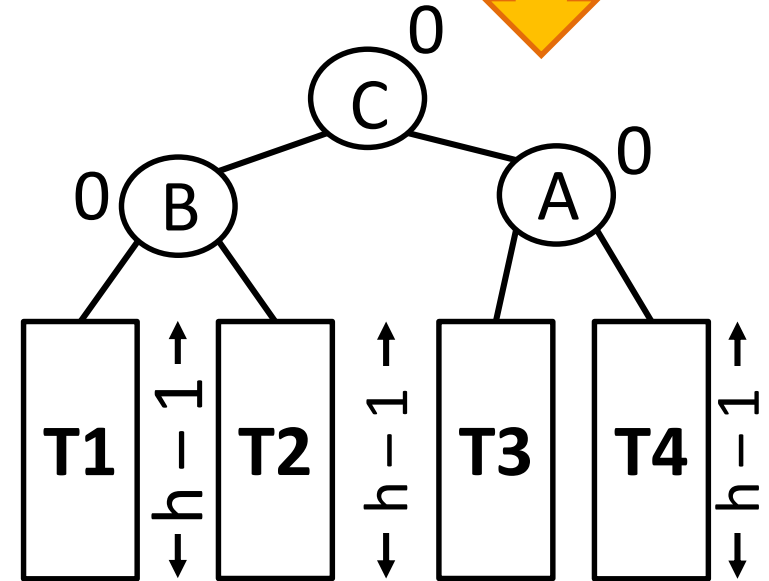
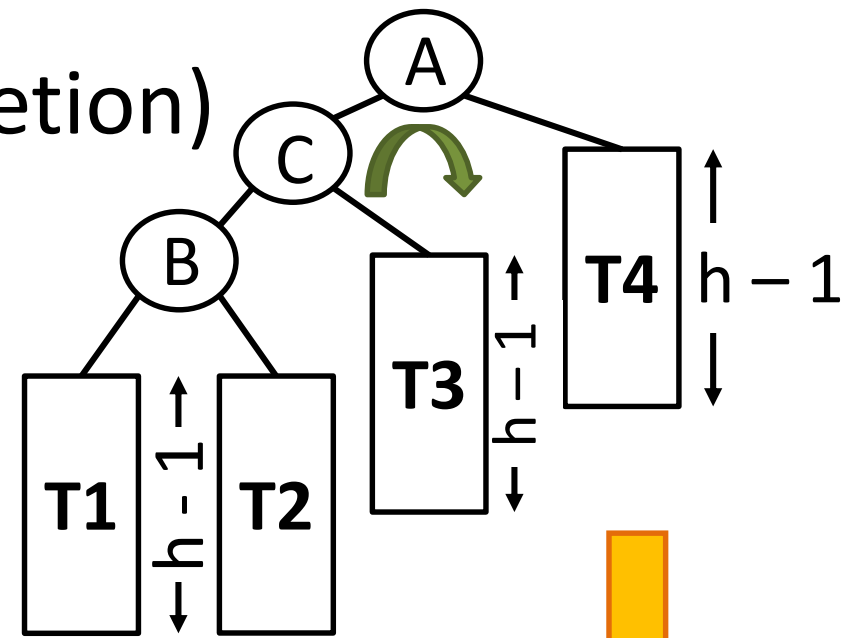
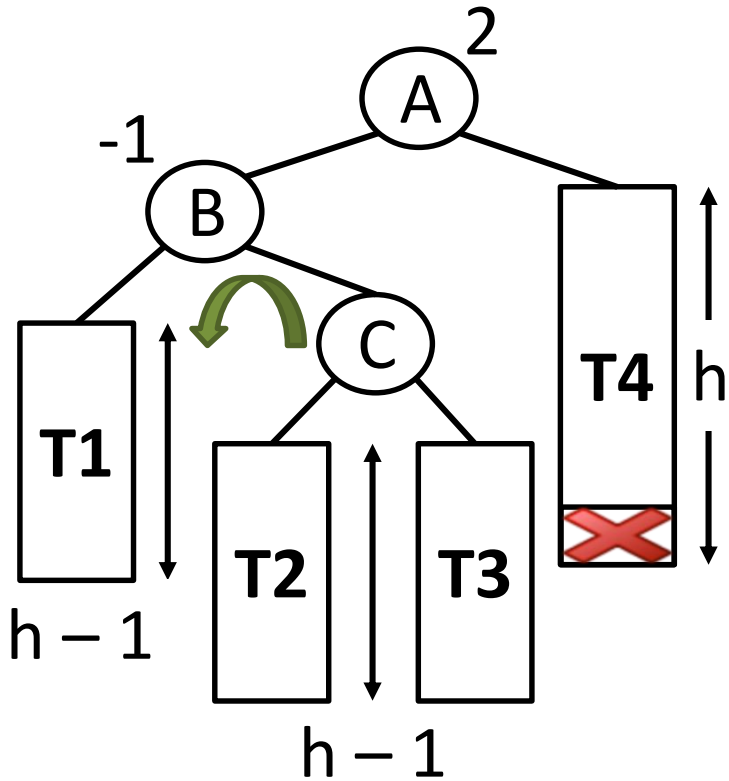


Case 2: Right of Right (Deletion)

BalFactor < -1 and *balance(Node*→*right)* ≤ 0

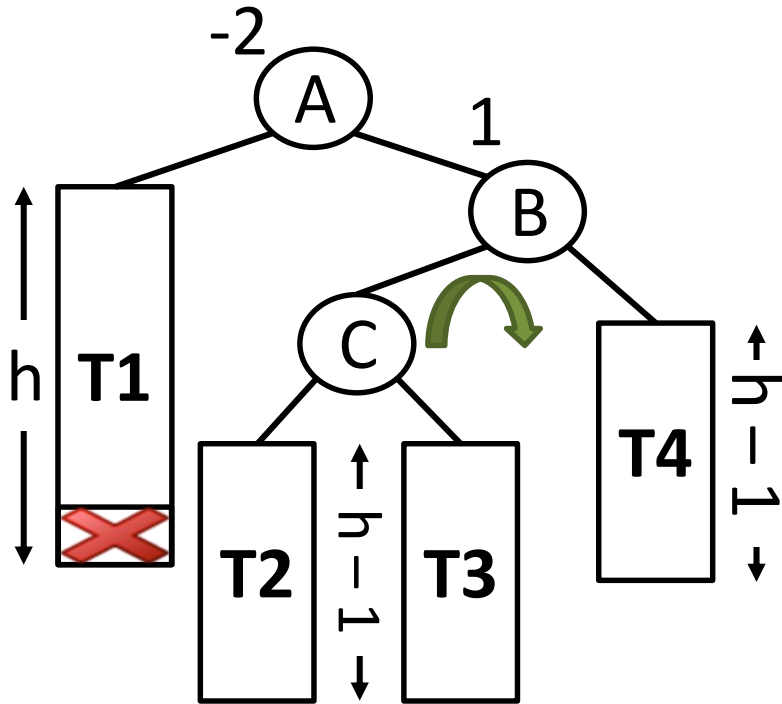


Case 3: Right of Left (Deletion)

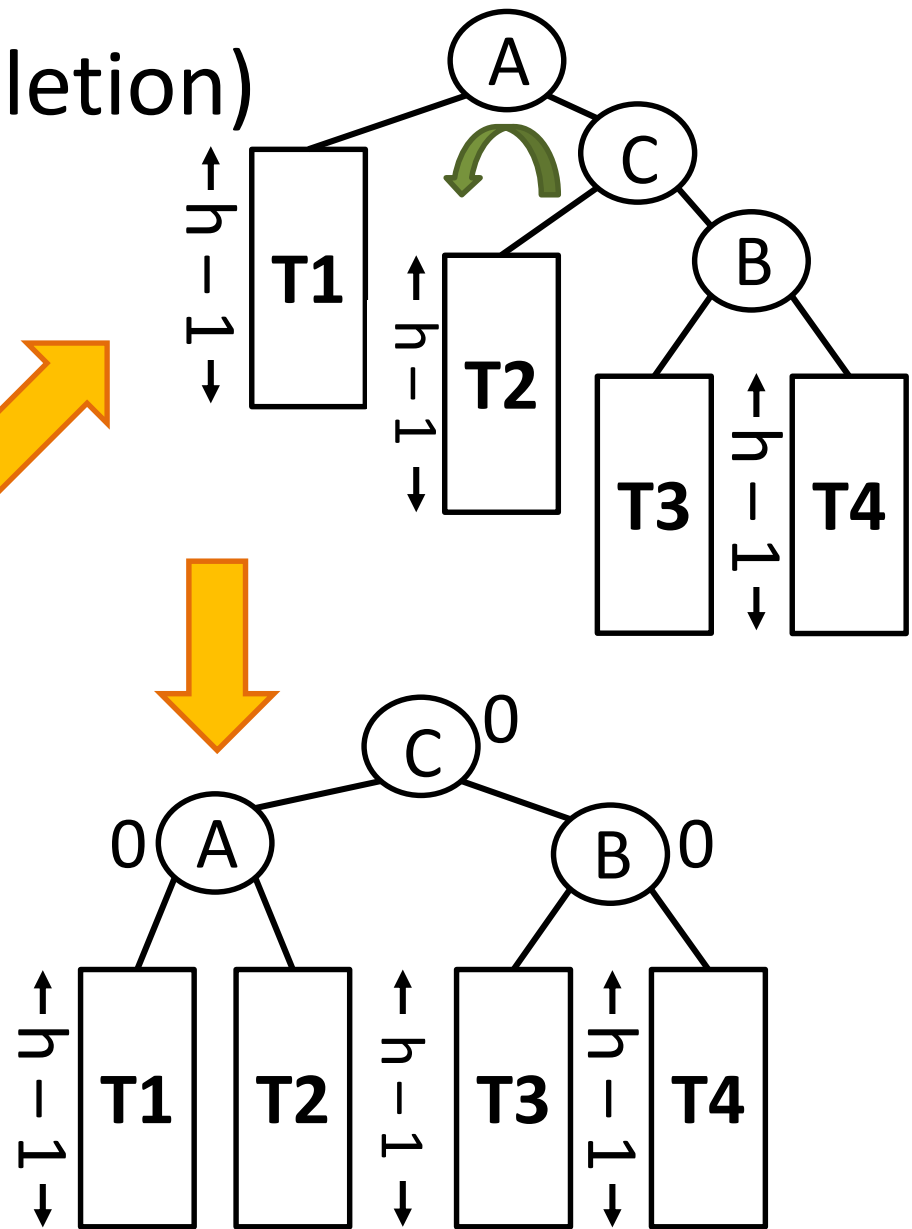


BalFactor > 1 and
balance(Node → *left*) < 0

Case 4: Left of Right (Deletion)



BalFactor < -1 and
balance(Node → right) > 0

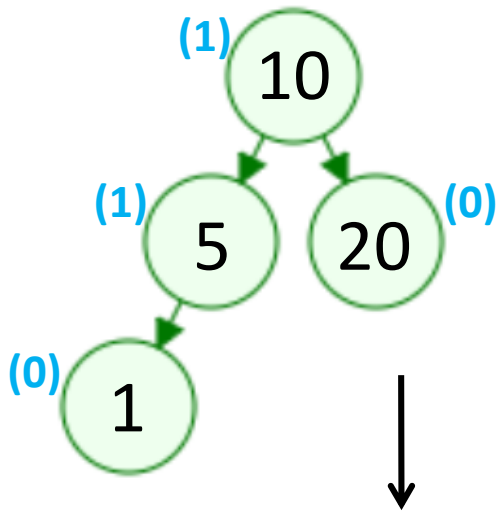


Deletion (recursion)

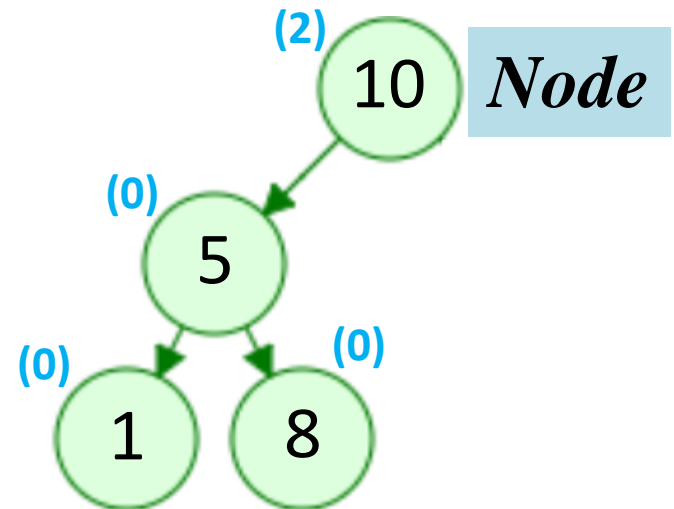
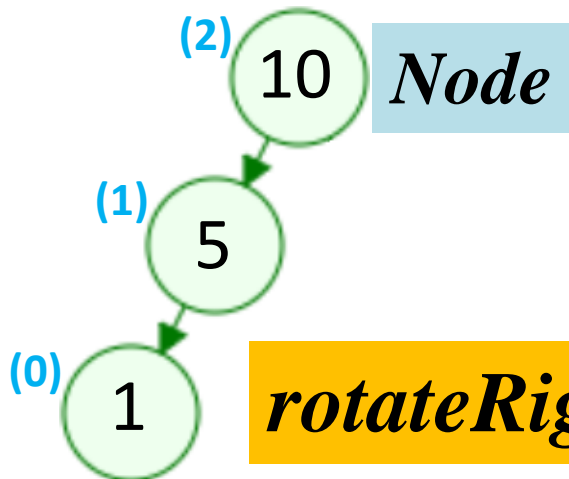
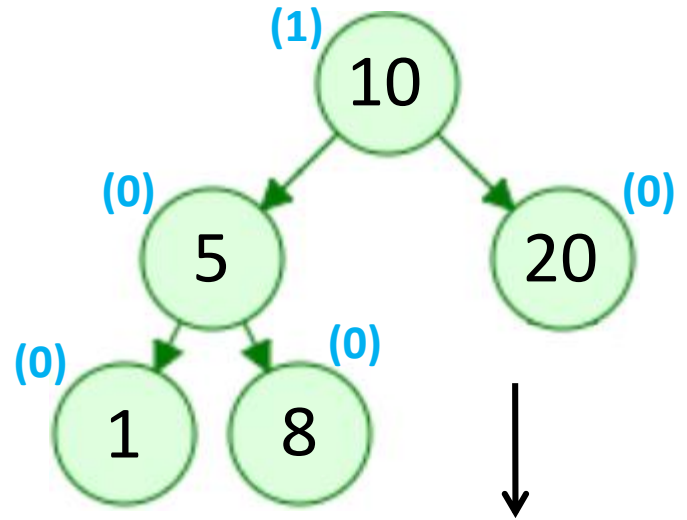
1. Delete the node with value *Key* using normal BST deletion.
2. Update height of the ancestor node, say *Node*.
3. Get the balance factor of this ancestor node, i.e. *Node*.
4. $BalFactor = (\text{height of } Node \rightarrow left - \text{height of } Node \rightarrow right).$
5. If $BalFactor > 1$ and $balance(Node \rightarrow left) \geq 0$
6. Return *rotateRight(Node)*. // Left of Left.

Example – 2

BalFactor > 1 and
balance(Node → *left*) >= 0



Delete
Key = 20

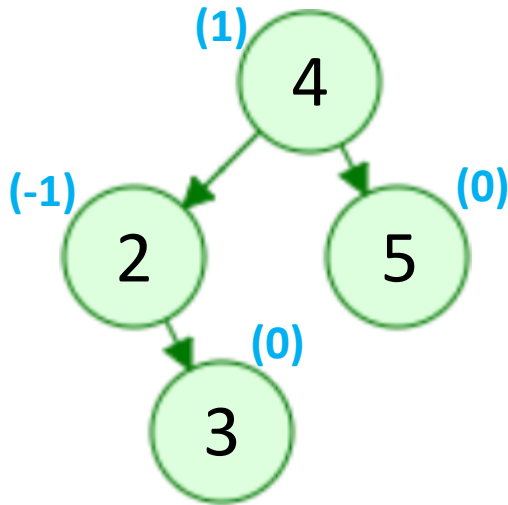


Contd...

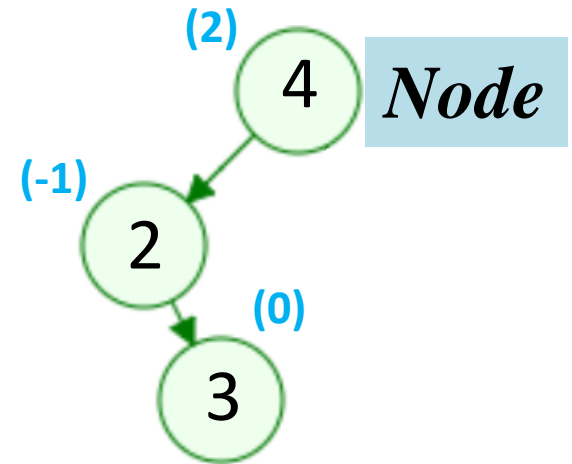
7. If *BalFactor* > 1 and *balance(Node*→*left*) < 0
8. *Node*→*left* = *rotateLeft*(*Node*→*left*);
9. Return *rotateRight*(*Node*); // Right of Left.

Example – 3

BalFactor > 1 and
balance(Node→*left)* < 0



Delete *Key* = 5
→



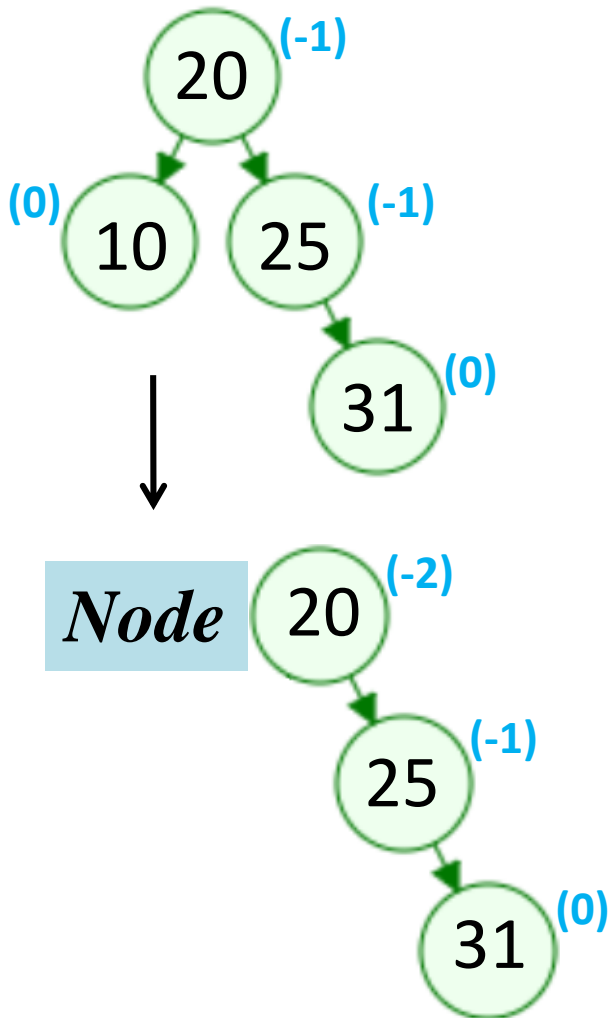
Node→*left* = *rotateLeft(Node*→*left)*;
rotateRight(Node);

Contd...

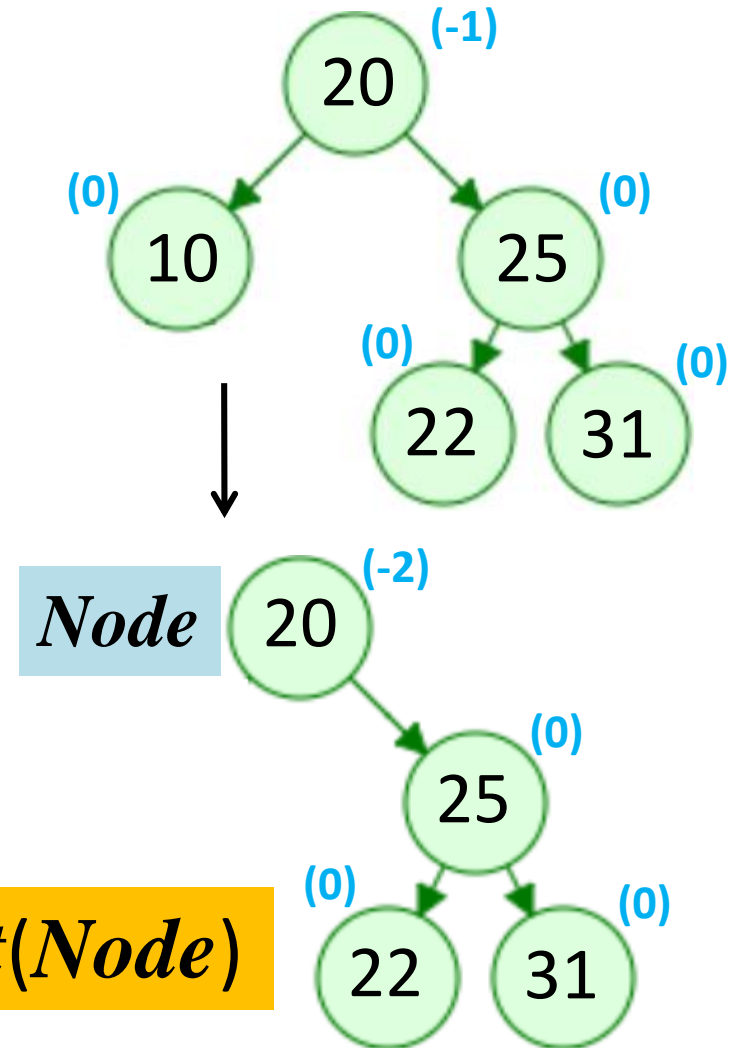
7. If *BalFactor* > 1 and *balance(Node*→*left*) < 0
8. *Node*→*left* = *rotateLeft*(*Node*→*left*);
9. Return *rotateRight*(*Node*); // Right of Left.
10. If *BalFactor* < -1 and *balance(Node*→*right*) <= 0
11. Return *rotateLeft*(*Node*); // Right of Right.

Example – 4

BalFactor < -1 and
balance(Node→*right)* <= 0



Delete
Key = 10



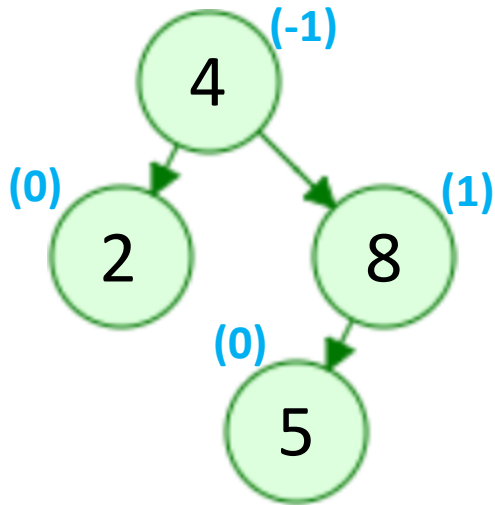
rotateLeft(Node)

Contd...

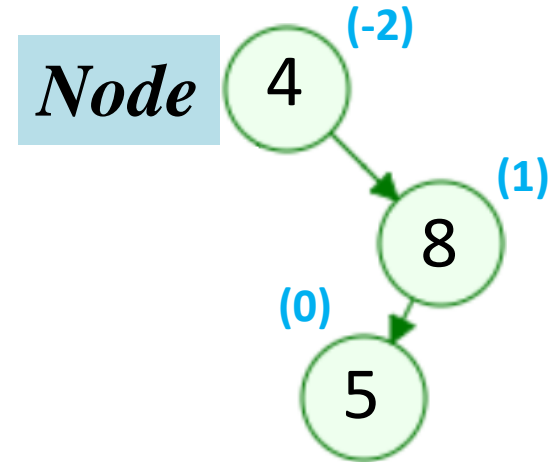
7. If *BalFactor* > 1 and *balance(Node→left)* < 0
8. *Node→left* = *rotateLeft(Node→left)*;
9. Return *rotateRight(Node)*; // Right of Left.
10. If *BalFactor* < -1 and *balance(Node→right)* <= 0
11. Return *rotateLeft(Node)*; // Right of Right.
12. If *BalFactor* < -1 and *balance(Node→right)* > 0
13. *Node→right* = *rotateRight(Node→right)*;
14. Return *rotateLeft(Node)*; // Left of Right.

Example – 5

BalFactor < -1 and
balance(Node→*right)* > 0



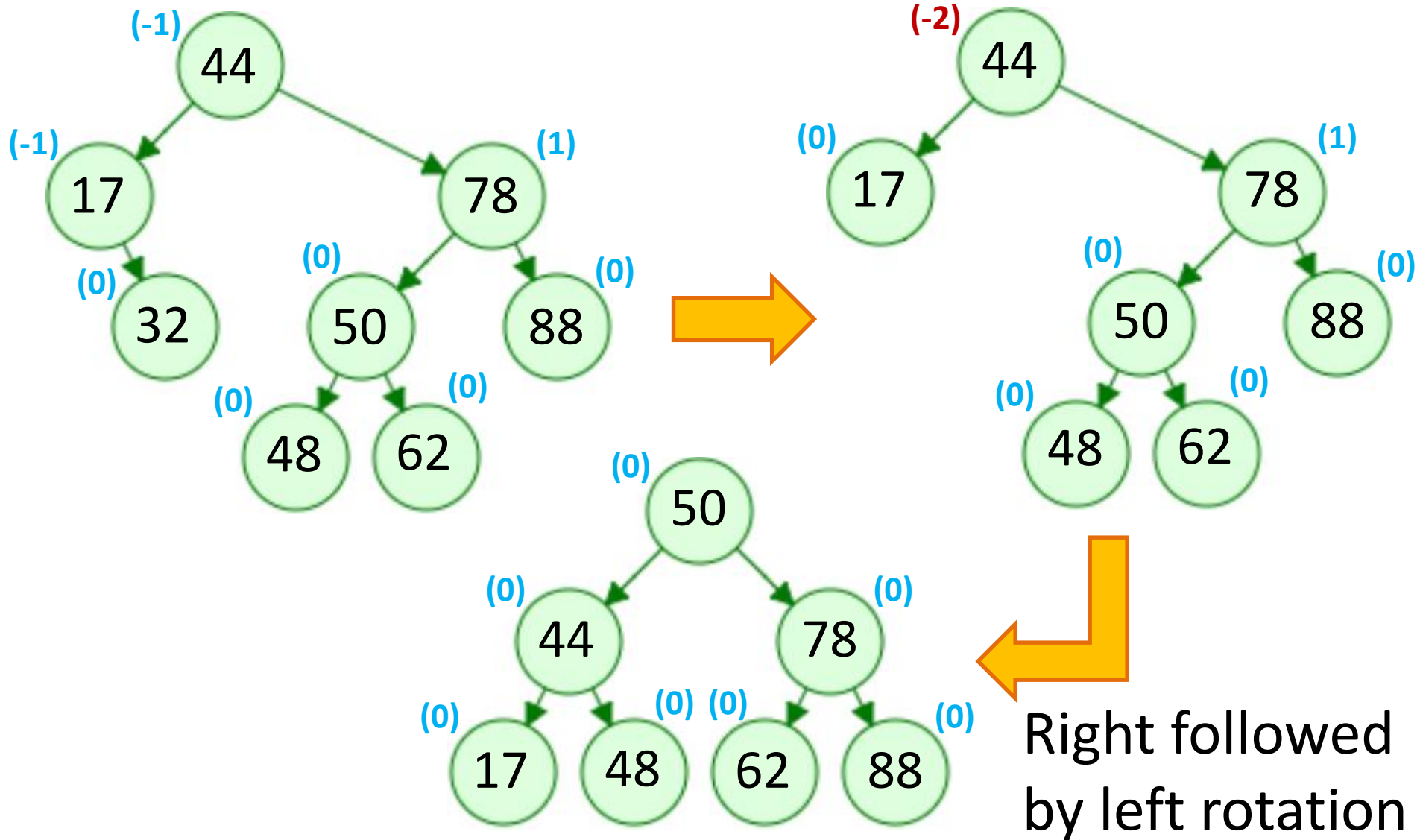
Delete *Key* = 2
→



Node→*right* = *rotateRight(Node*→*right)*;
rotateLeft(Node);

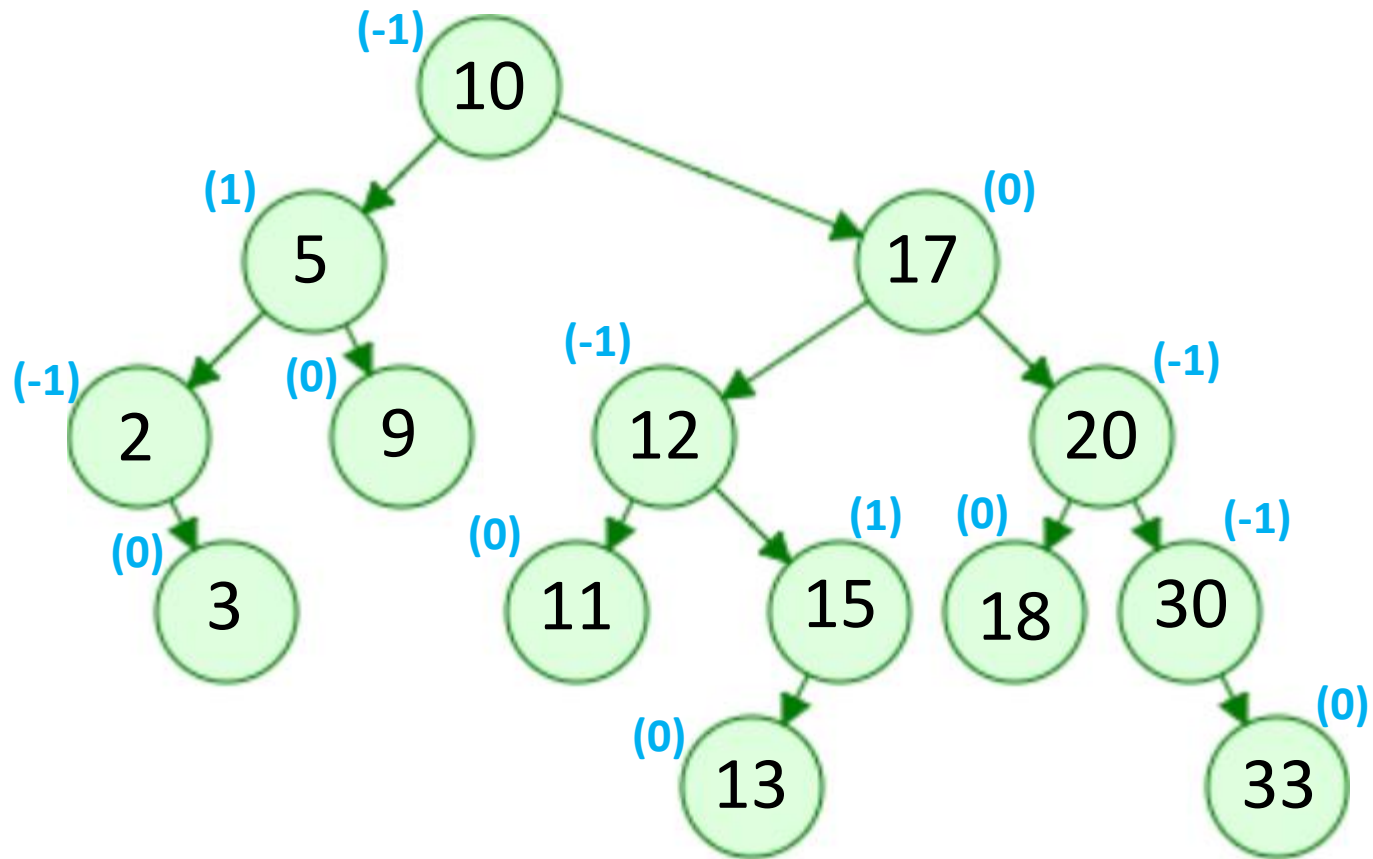
Example – 6

Delete 32

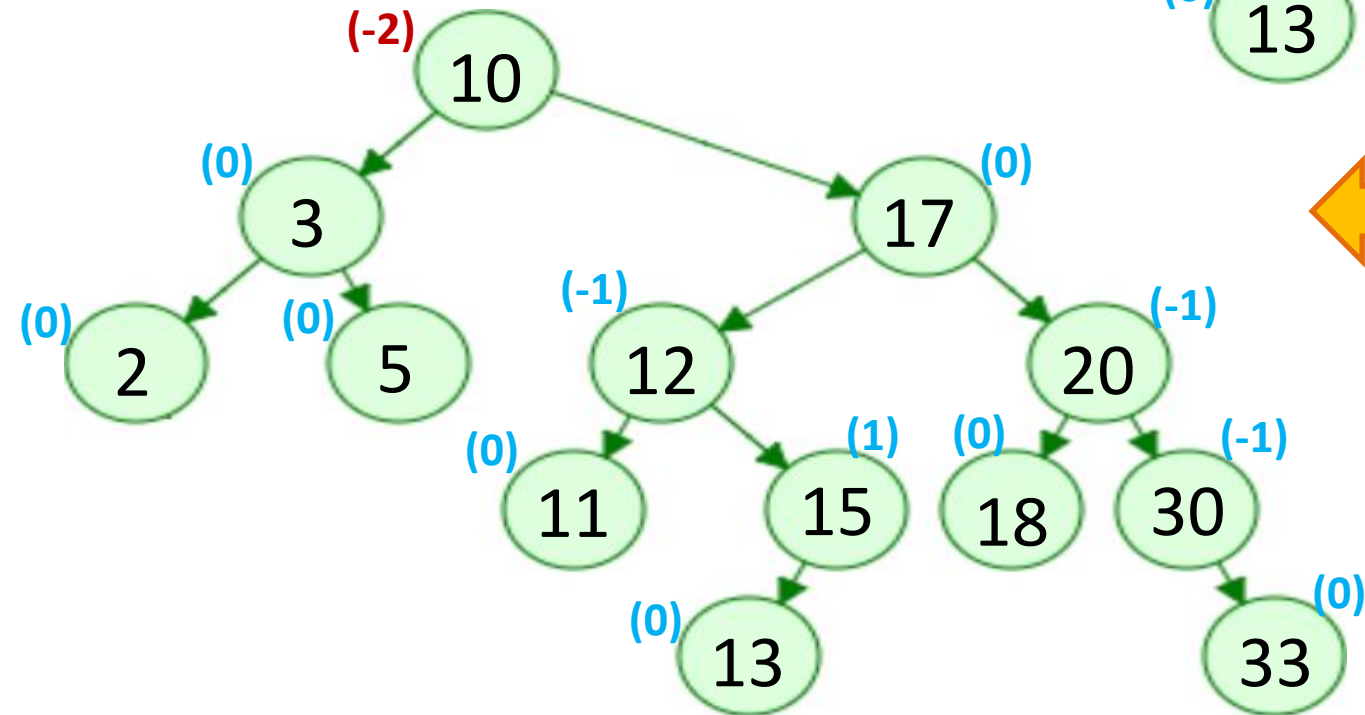
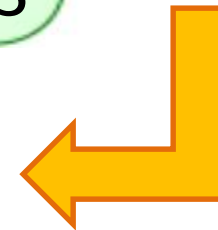
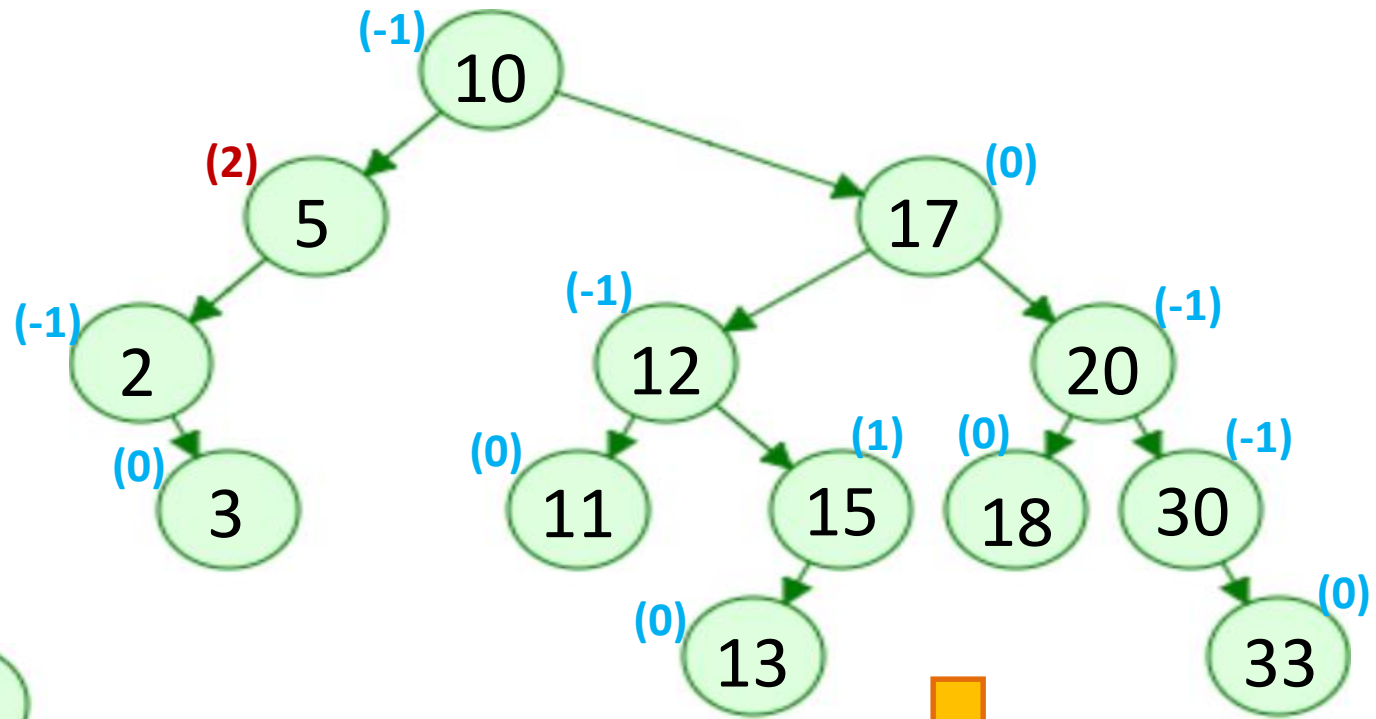


Example – 7

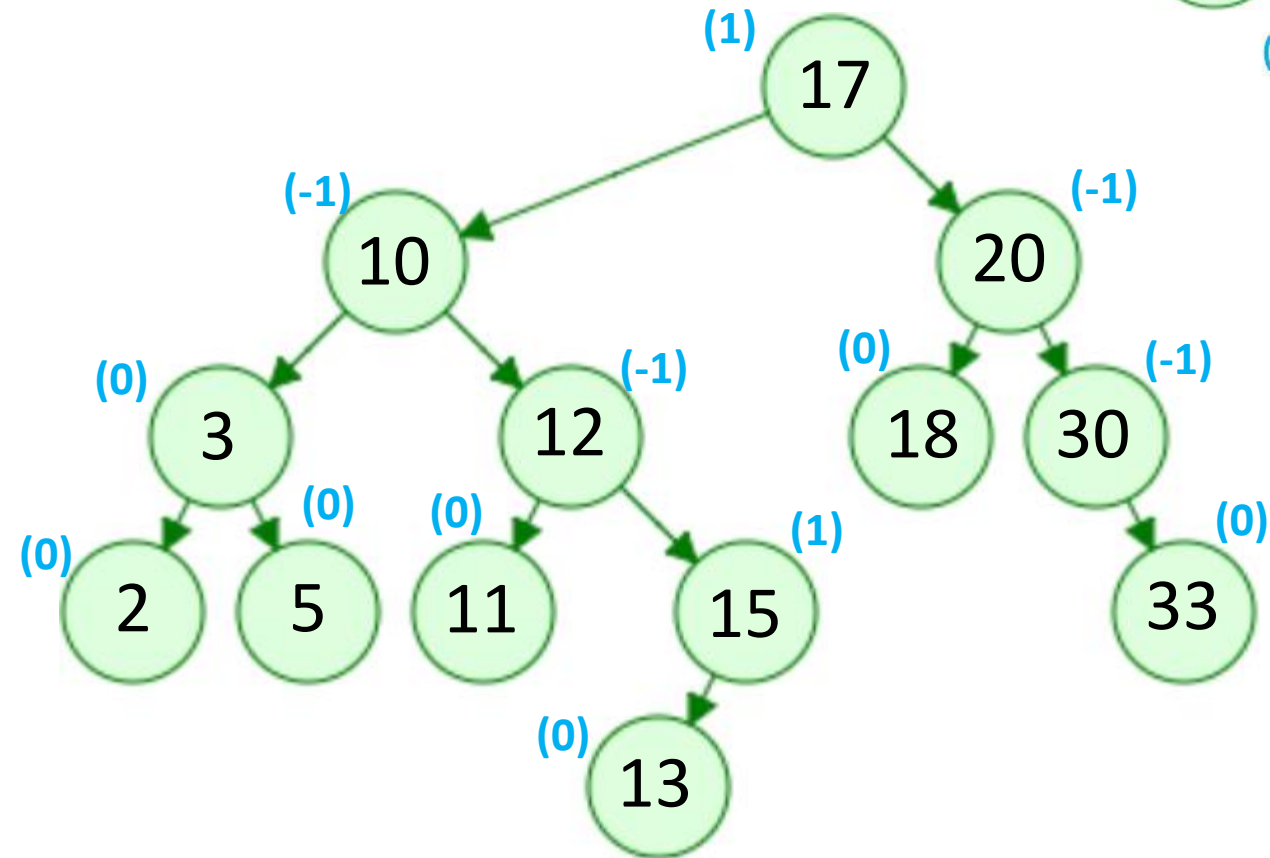
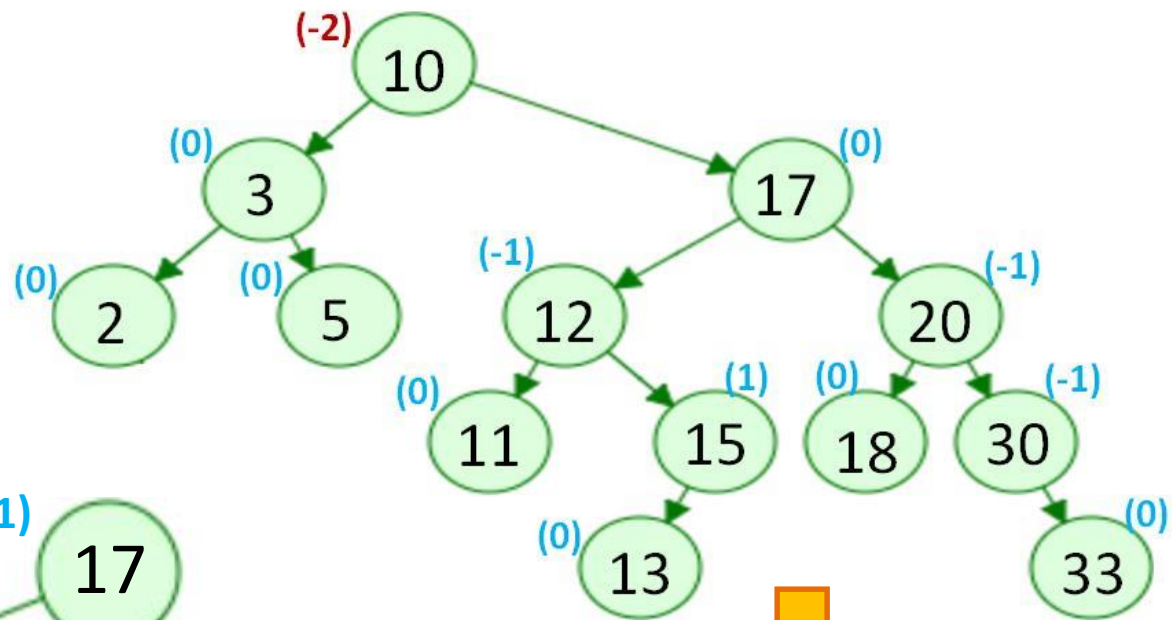
Delete 9



Contd...



Contd...



AVL

Question 1

1. Insert 10, 20, 15, 25, 30, 16, 18, 19.
2. Delete 30.

Question 2

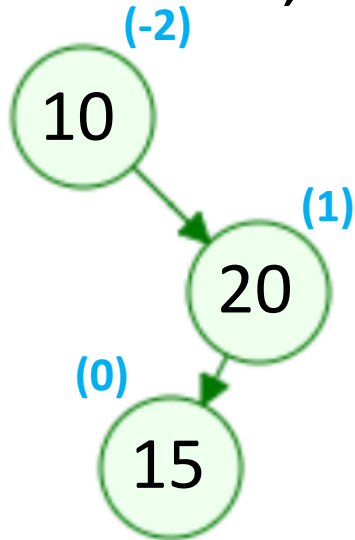
1. Insert 3, 11, 2, 12, 8, 10, 4, 9, 5, 1, 7, 6.
2. Delete 12.

Question 3

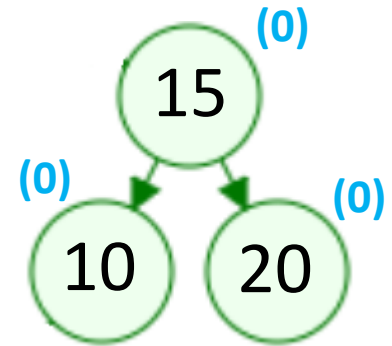
1. Insert 50, 80, 25, 30, 70, 75, 77, 65, 60, 64, 63, 55.
2. Delete 70.

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

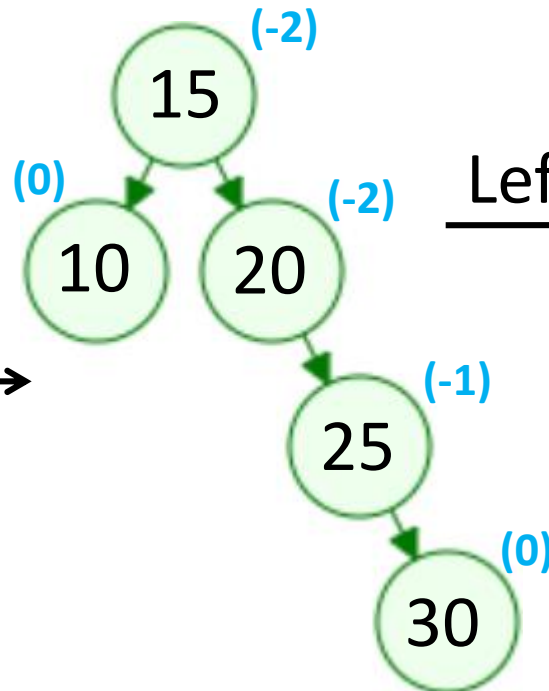
- Insert 10, 20, 15



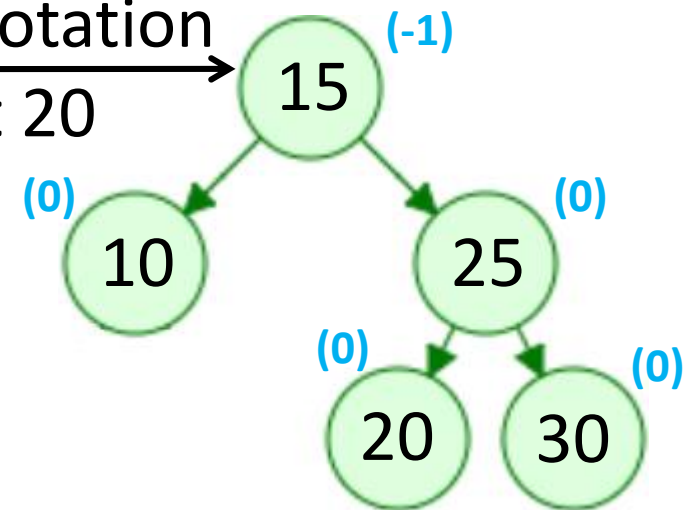
Right Rotation at 20
Left Rotation at 10



- Insert 25, 30

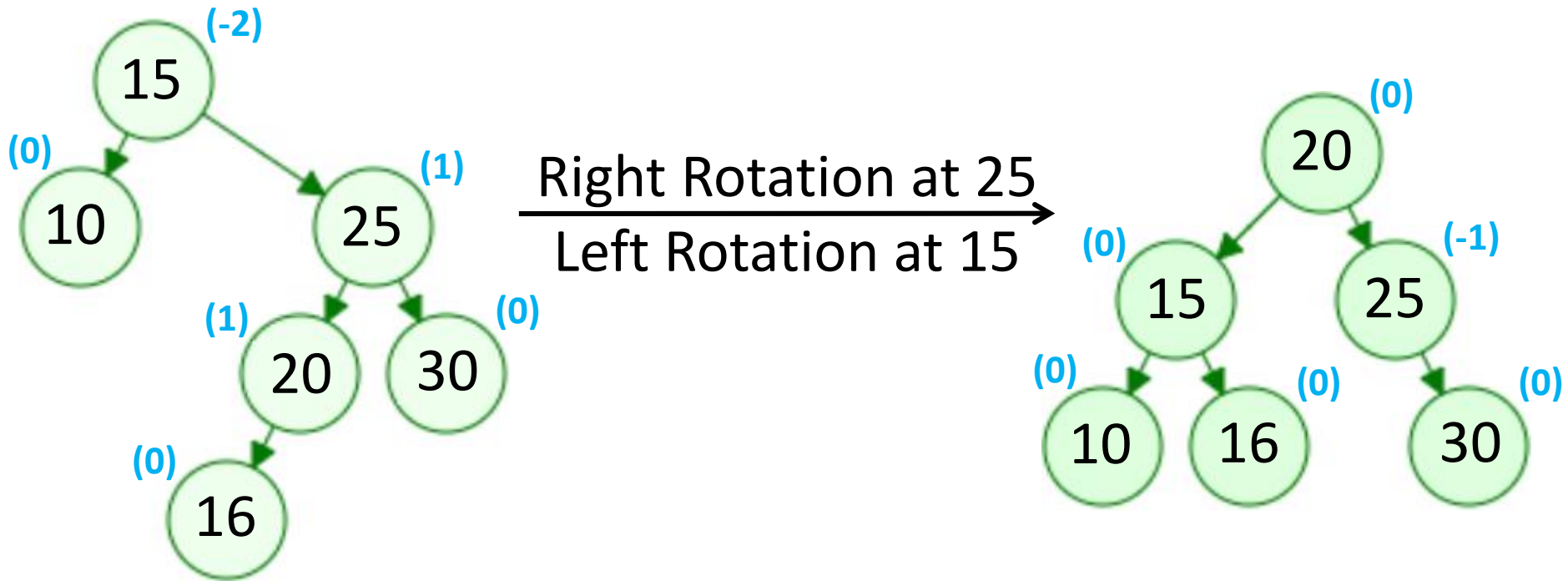


Left Rotation
at 20



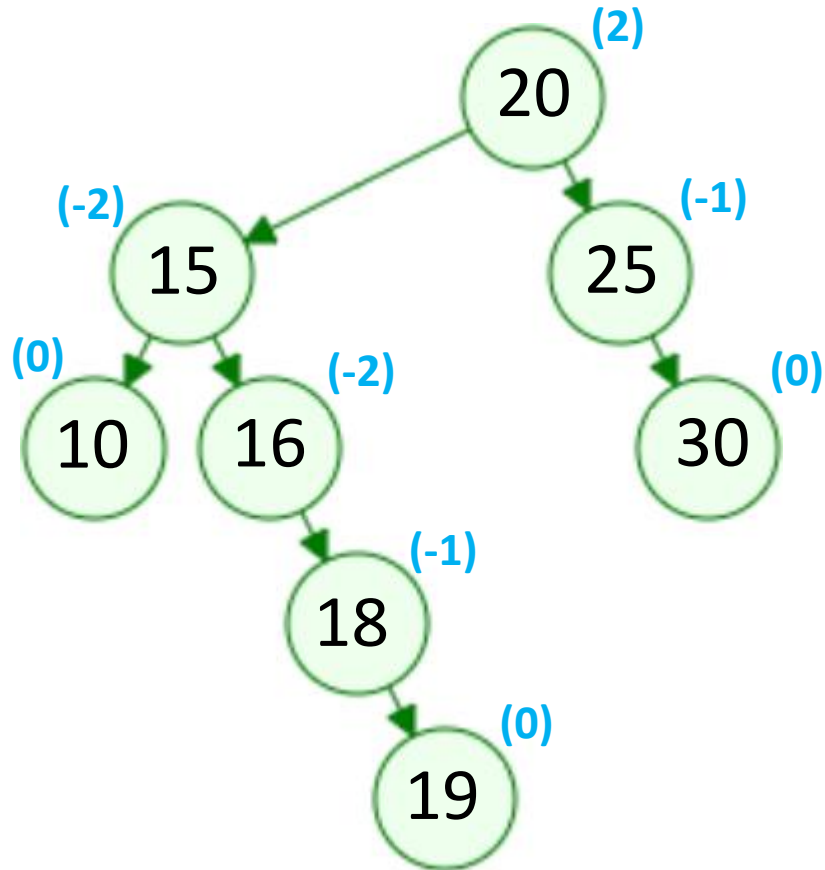
Contd...

- Insert 16

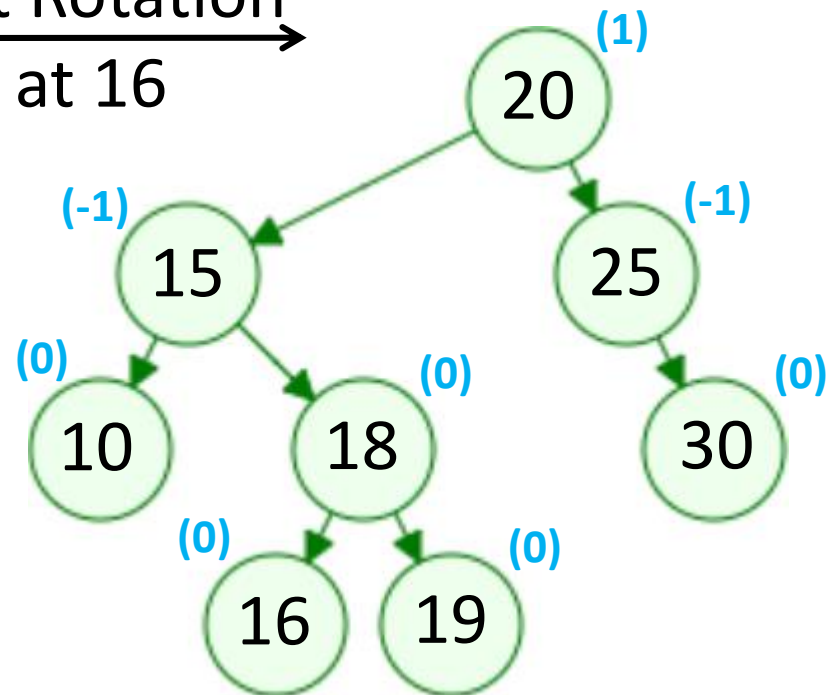


Contd...

- Insert 18, 19

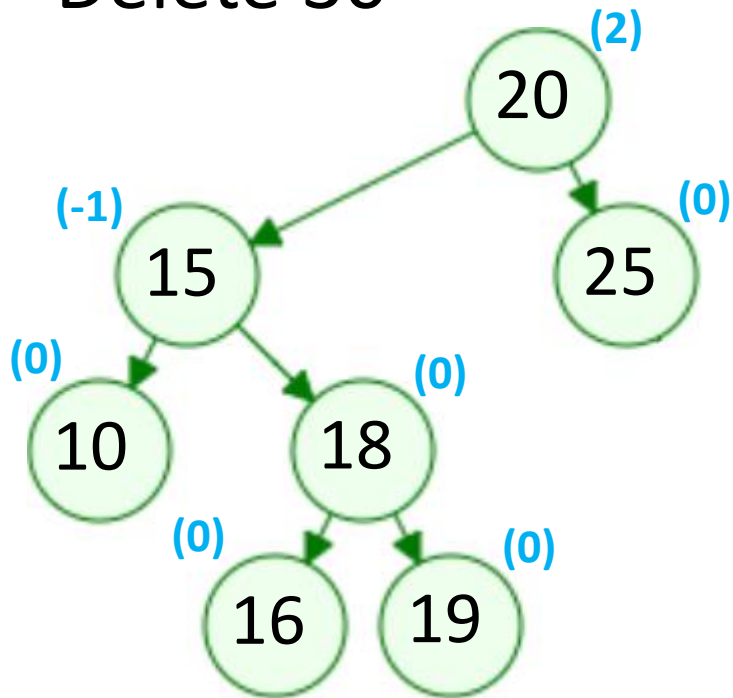


Left Rotation
at 16

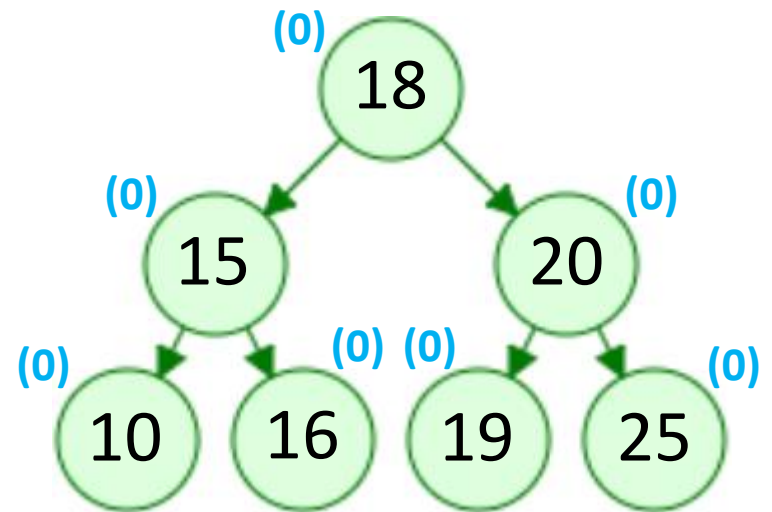


Contd...

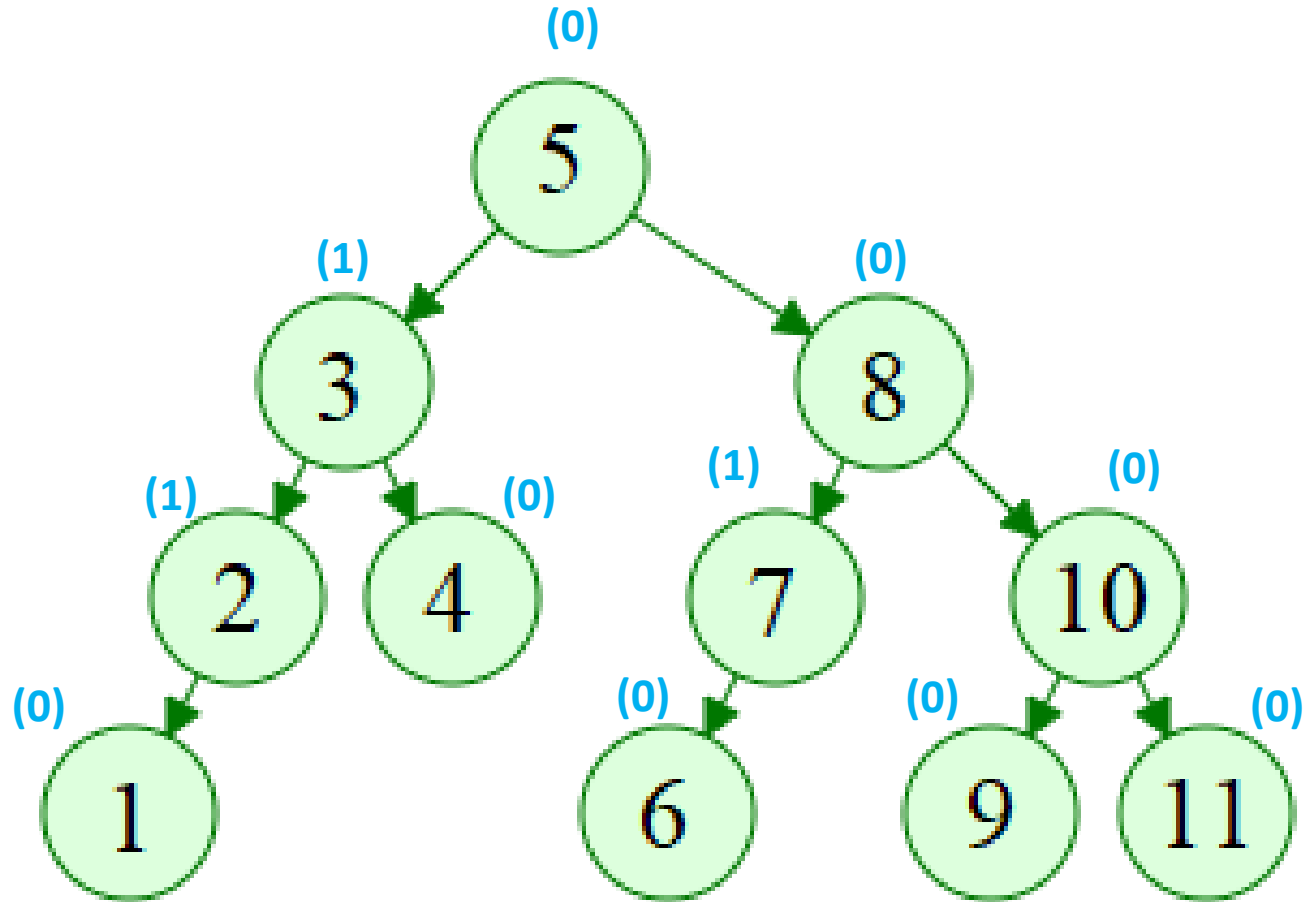
- Delete 30



Left Rotation at 15
Right Rotation at 20



Create AVL: 3, 11, 2, 12, 8, 10, 4, 9, 5, 1, 7, 6.
Delete 12.



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.

Thank You