

Balanced Tree

Lecture 3a

Introduction(Binary Tree)

The time required to search a binary tree varies between $O(n)$ and $O(\log_2 n)$

The structure of tree depends on the order in which the records are inserted.

Height of a Tree

The height of a node in a binary tree is the largest number of edges in a path from a leaf node to a target node.

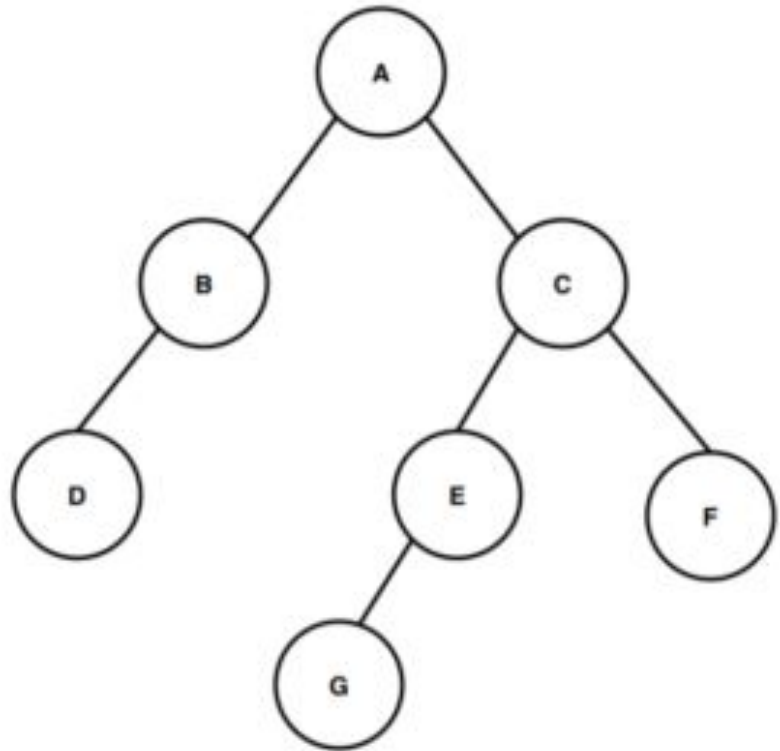
If the target node doesn't have any other nodes connected to it, the height of that node would be =0.

The height of a binary tree is the height of the root node in the whole binary tree.

In other words, the height of a binary tree is equal to the largest number of edges from the root to the most distant leaf node.

The depth of a node in a binary tree is the total number of edges from the root node to the target node.

Similarly, the depth of a binary tree is the total number of edges from the root node to the most distant leaf node.



First, we'll calculate the height of node C.

So, according to the definition, the height of node C is the largest number of edges in a path from the leaf node to node C.

We can see that there are two paths for node C: $C \rightarrow E \rightarrow G$, and $C \rightarrow F$.

The largest number of edges among these two paths would be 2 hence, the height of node C is 2.

Now we'll calculate the height of the binary tree. From the root, we can have three different paths leading to the leaf nodes:

$A \rightarrow C \rightarrow F$,
 $A \rightarrow B \rightarrow D$,
and $A \rightarrow C \rightarrow E \rightarrow G$.

Among these three paths, the path $A \rightarrow C \rightarrow E \rightarrow G$ contains the largest number of edges, which is 3.

Therefore, the height of the tree is 3.

Balanced Tree (AVL Tree)

The balance of a tree looks related to height of its left subtree and height of its right.

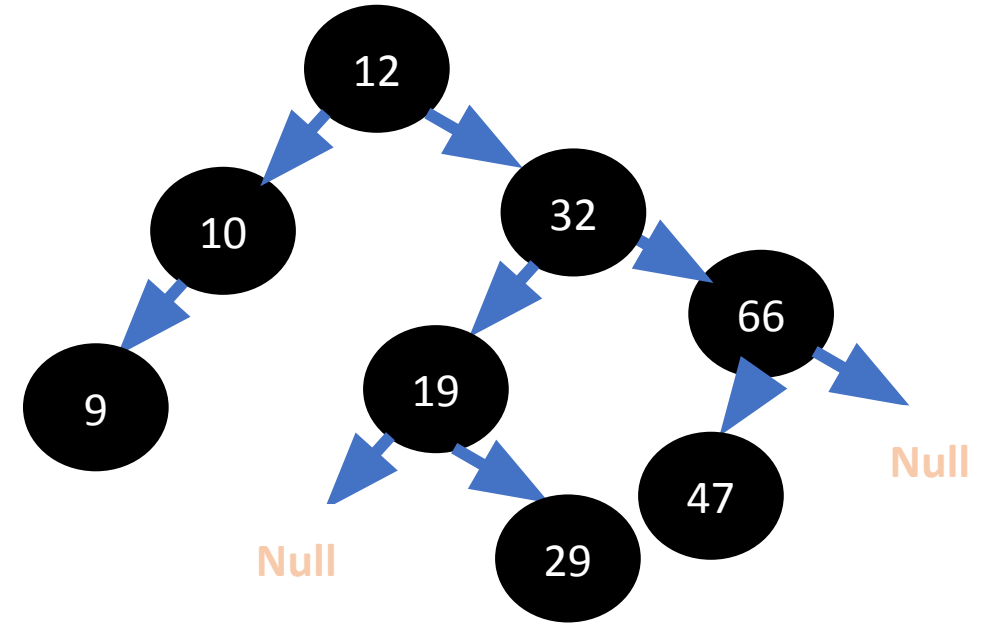
Height of a null tree is -1
and 0 if it contains single node.

Balance of a node = height of its left subtree - height of its right subtree

A balanced binary tree is a binary tree in which balance of every node is never more than 1 or always less than equal to one

This means balance of every node is either 0 or -1 or 1 in case of a balanced Tree

12 ,32,66,10,9,19,29,47



If the records are in some sorted order, then the resulting tree will have either left link NULL or right link NULL

But, if the records are inserted such that half the records are in left subtree and half of them are in right subtree ---a kind of balanced tree is created

Balance Factor

In an AVL tree, the balance factor of a node is a measure of the difference in height between the left and right subtrees of that node.

The balance factor is calculated as follows:

balance factor = height of left subtree - height of right subtree.

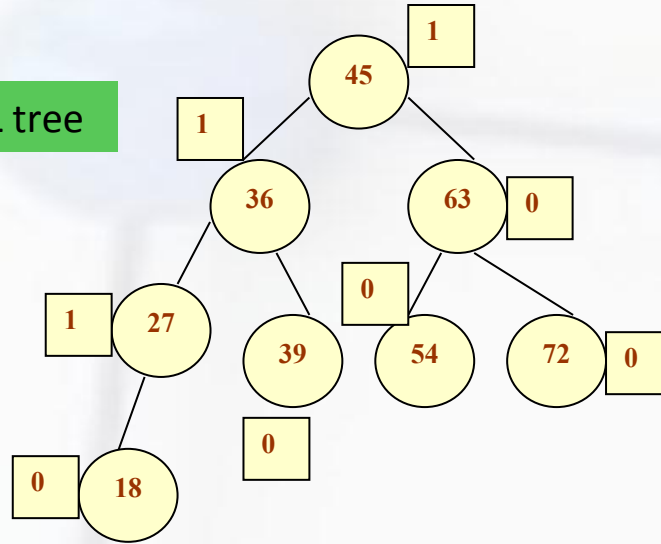
Balance factor = Height (left sub-tree) – Height (right sub-tree)

Balance factor

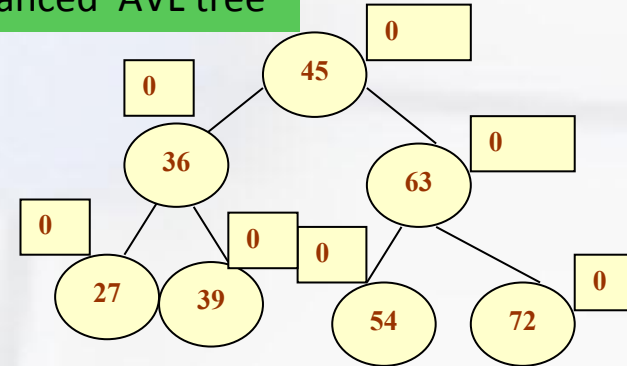
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called *Left-heavy tree*.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree is equal to the height of its right sub-tree.
- If the balance factor of a node is -1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called *Right-heavy tree*.
- A node is **unbalanced** if its balance factor is not -1, 0, or 1.

Examples of Balance Factor

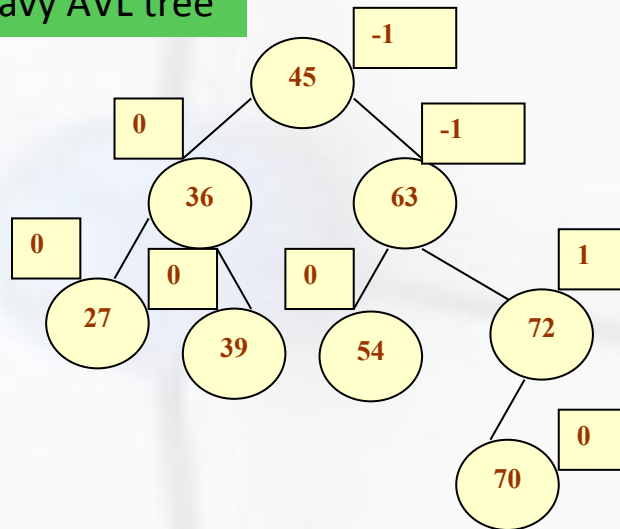
Left heavy AVL tree



Balanced AVL tree



Right heavy AVL tree



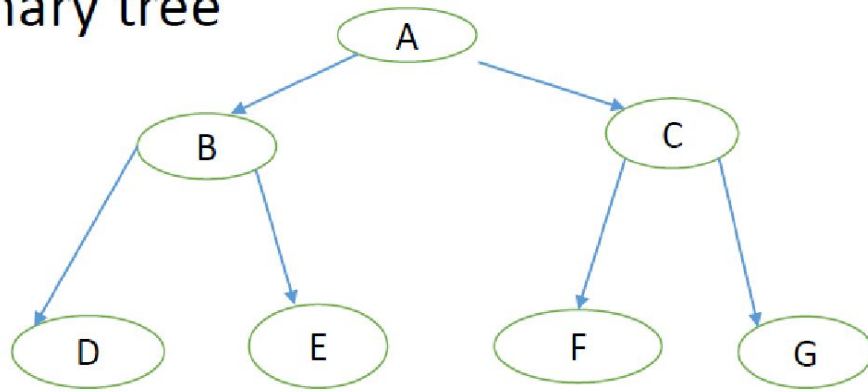
AVL Trees

- AVL tree is a **self-balancing binary search tree** in which the heights of the two sub-trees of a node may differ by at most one.
- AVL tree is a **height-balanced tree**
- Self-balancing : re-balance after BST inserting or deleting
- AVL is named by its inventor **Adelson-Velskii** and **Landis**
 - provided the height balance property and insertion and deletion operations that maintain the property in time complexity of $O(\log n)$

Building an AVL Tree

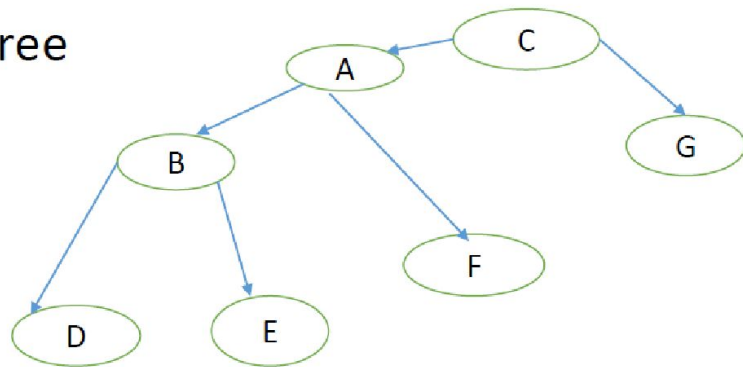
- AVL tree is same as BST but with a little difference that in AVL tree balance of every node is also stored.
- So, while building an AVL tree, it has to be checked that the balance of any node at any time is not more than 1.
- If the balance becomes more than 1 then right rotations and left rotations are performed so that the tree becomes balanced.
- What is rotation??

Complete binary tree



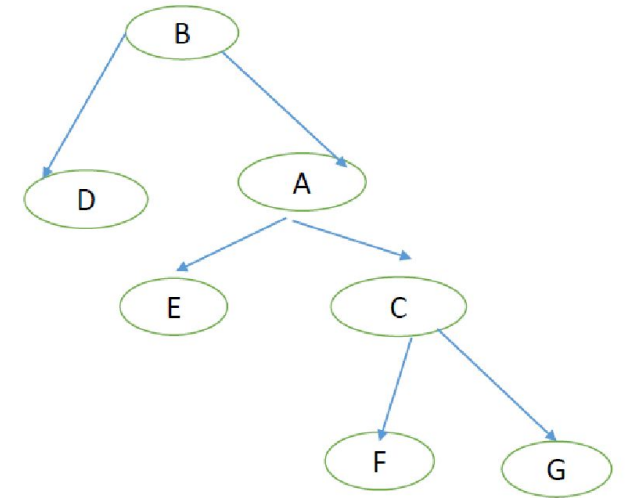
Left Rotation

Unbalanced tree



Right Rotation


Unbalanced tree





Building an AVL tree

While building an AVL tree, it is necessary to perform rotations to keep it balanced but the rotations shall be such that :

1. The inorder traversal of the transformed tree is same as of original tree
 2. The transformed tree is balanced i.e. balance of every node is not more than one.
- 

Height of Balanced Tree

Property: height of balanced tree of n nodes is $O(\log n)$.

Proof:

$N(h)$ denote the minimum number of AVL tree of height h , then

$$N(h) \leq n$$

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

We may assume that $N(h-1) > N(h-2)$,

$$\text{So, } N(h) > 1 + N(h-2) + N(h-2) = 1 + 2 \cdot N(h-2) > 2 \cdot N(h-2)$$

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

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

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

$$h < 2 \log N(h) \leq 2 \log n$$

Thus, these worst-case AVL trees have height $h = O(\log n)$.

Rotation algorithms

- **An algorithm to implement a left rotation at p is as follows:**

q = p->right;

temp = q->left;

q->left = p;

P->right = temp;

- **An algorithm to implement a right rotation at p is as follows:**

q = p->left;

temp = q->right;

q->right = p;

P->left = temp;

Inserting a Node in an AVL Tree

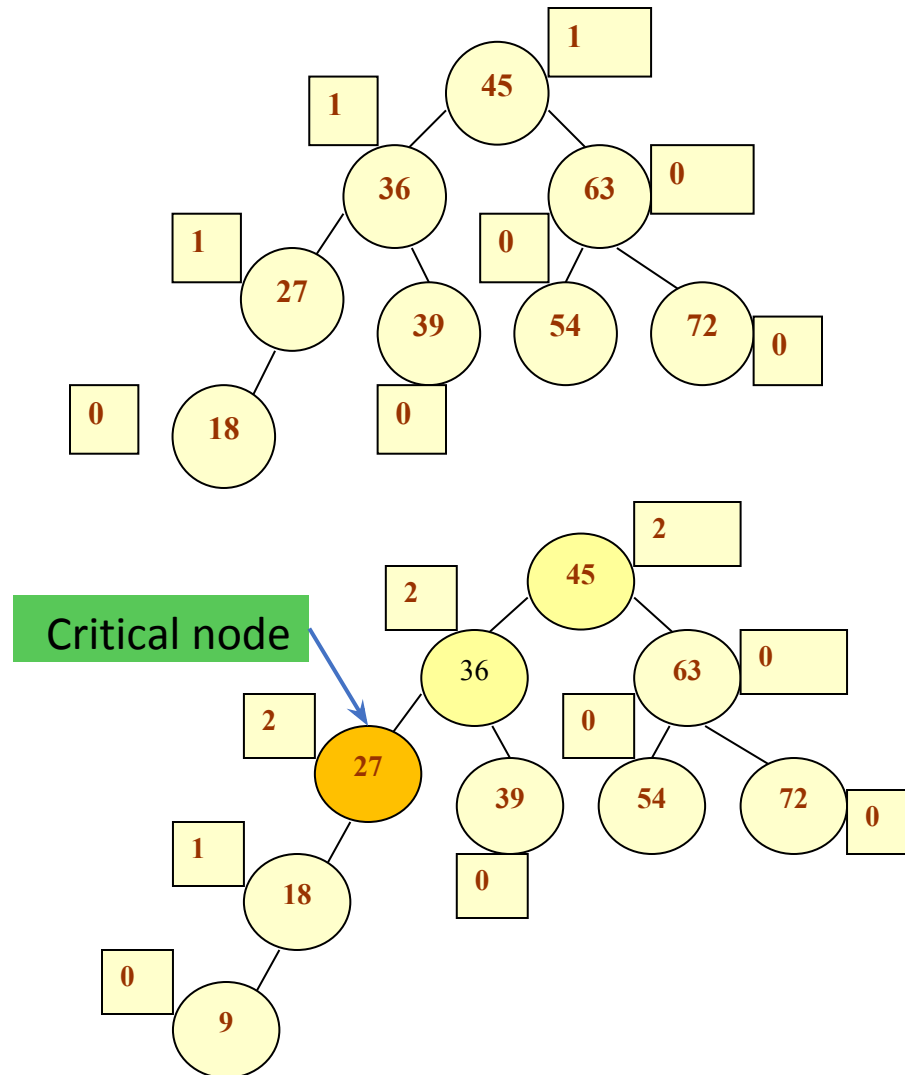
- Algorithm:
 - Step 1. insert new node as BST
 - Step 2: if not AVL tree, do re-balancing to derive an AVL tree
- A new node is inserted as the leaf node, so it will always have balance factor equal to zero.
- The nodes whose balance factors will change are those which lie on the path between the root of the tree and the newly inserted node.

Inserting a Node in an AVL Tree

- The possible changes which may take place in any node on the path are as follows:
 - Initially the node was either left or right heavy and after insertion has become balanced.
 - Initially the node was balanced and after insertion has become either left or right heavy.
 - Initially the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree thereby creating an unbalanced sub-tree. Such a node is said to be **a critical node**.
- **The critical node is the unbalanced node of lowest level**

Example of critical node

Example: Consider the AVL tree given below and insert 9 into it.

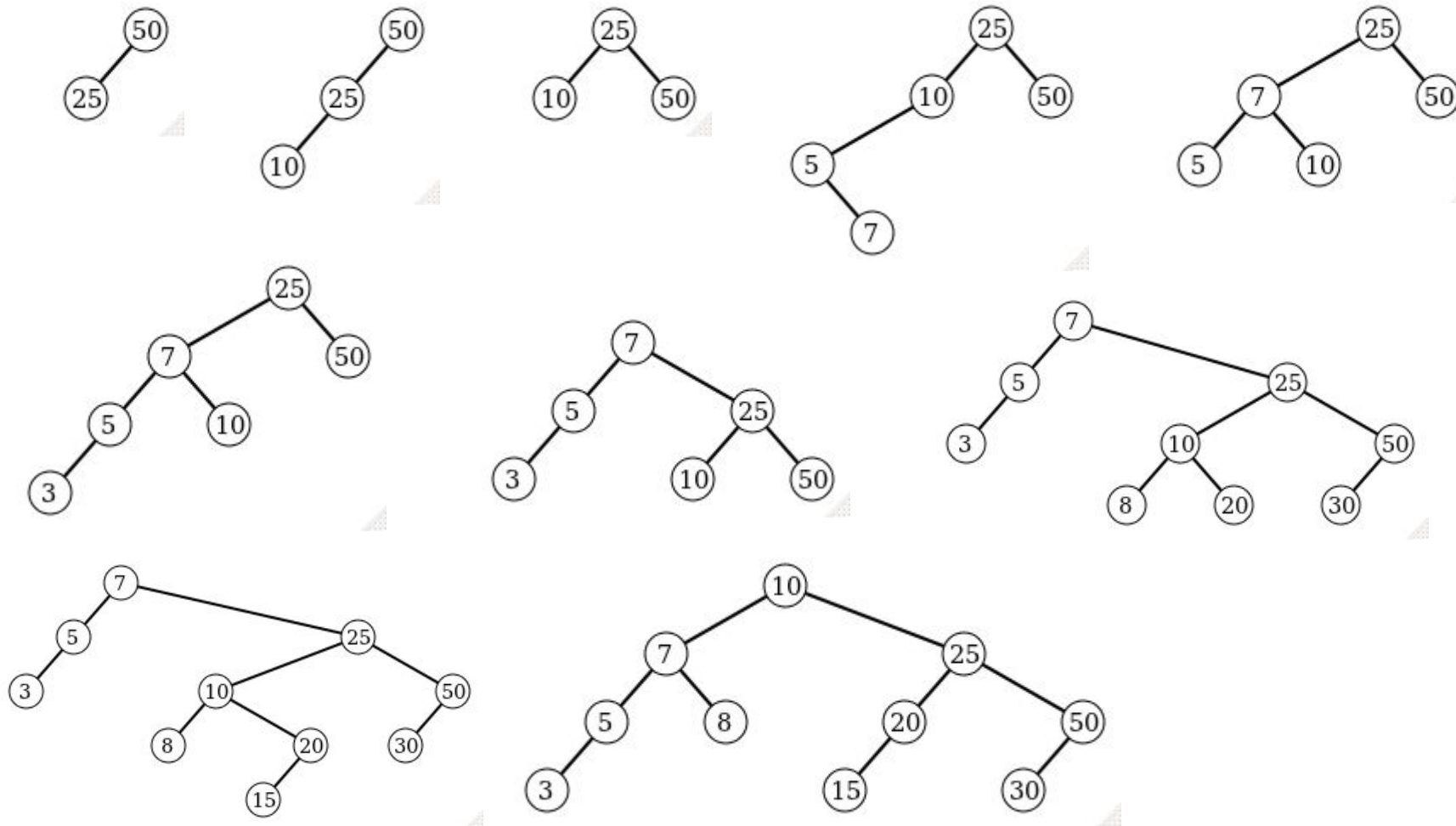


Example of AVL tree insert

Insert 50, 25, 10, 5, 7, 3, 30, 20, 8, 15 into AVL tree

Example of AVL tree insert

Insert 50, 25, 10, 5, 7, 3, 30, 20, 8, 15 into AVL tree



Cases of critical nodes

- Height of sub-tree where a new node is inserted can increase at most 1.
- The balance factor can increase by 1 or decrease by 1, so the balance factor of critical node is either 2 or -2.

Let x denote the critical node.

There are four possible cases:

Case 1: $\text{balance-factor}(x) = 2$, $\text{balance-factor}(x \rightarrow \text{left}) \geq 0$,

Case 2: $\text{balance-factor}(x) = 2$, $\text{balance-factor}(x \rightarrow \text{left}) < 0$,

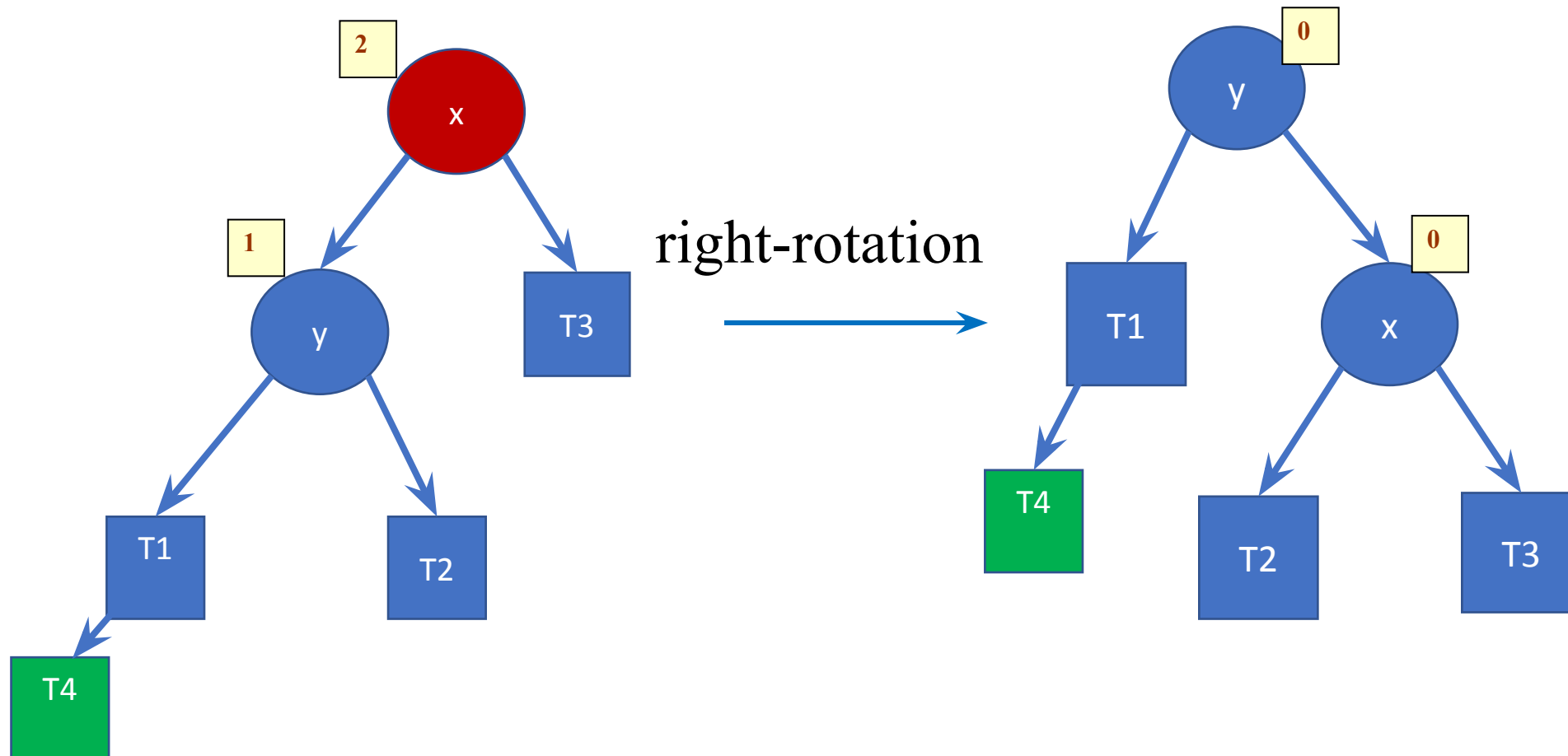
Case 3: $\text{balance-factor}(x) = -2$, $\text{balance-factor}(x \rightarrow \text{right}) \leq 0$,

Case 4: $\text{balance-factor}(x) = -2$, $\text{balance-factor}(x \rightarrow \text{right}) > 0$

Each case will different rotation operation for re-balancing

Case 1 : right-rotation to re-balance

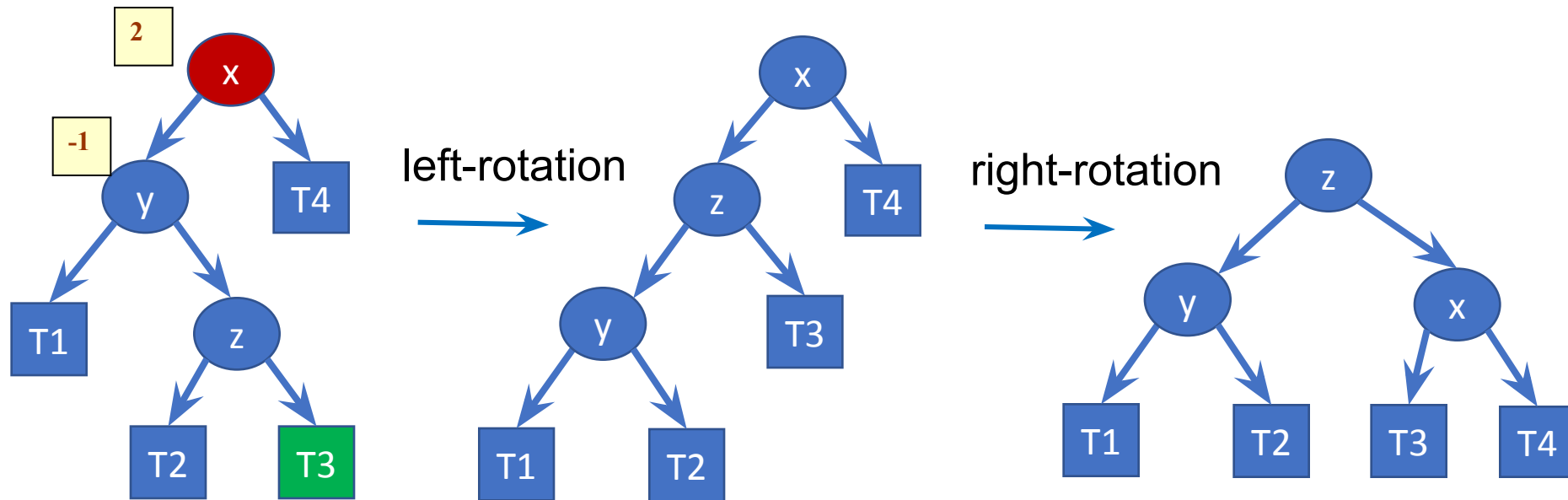
Case 1: $\text{balance-factor}(x) = 2$, $\text{balance-factor}(x \rightarrow \text{left}) \geq 0$
 $\text{right_rotation}(x)$ it return y as new top node



L-R-Rotation for Re-balancing

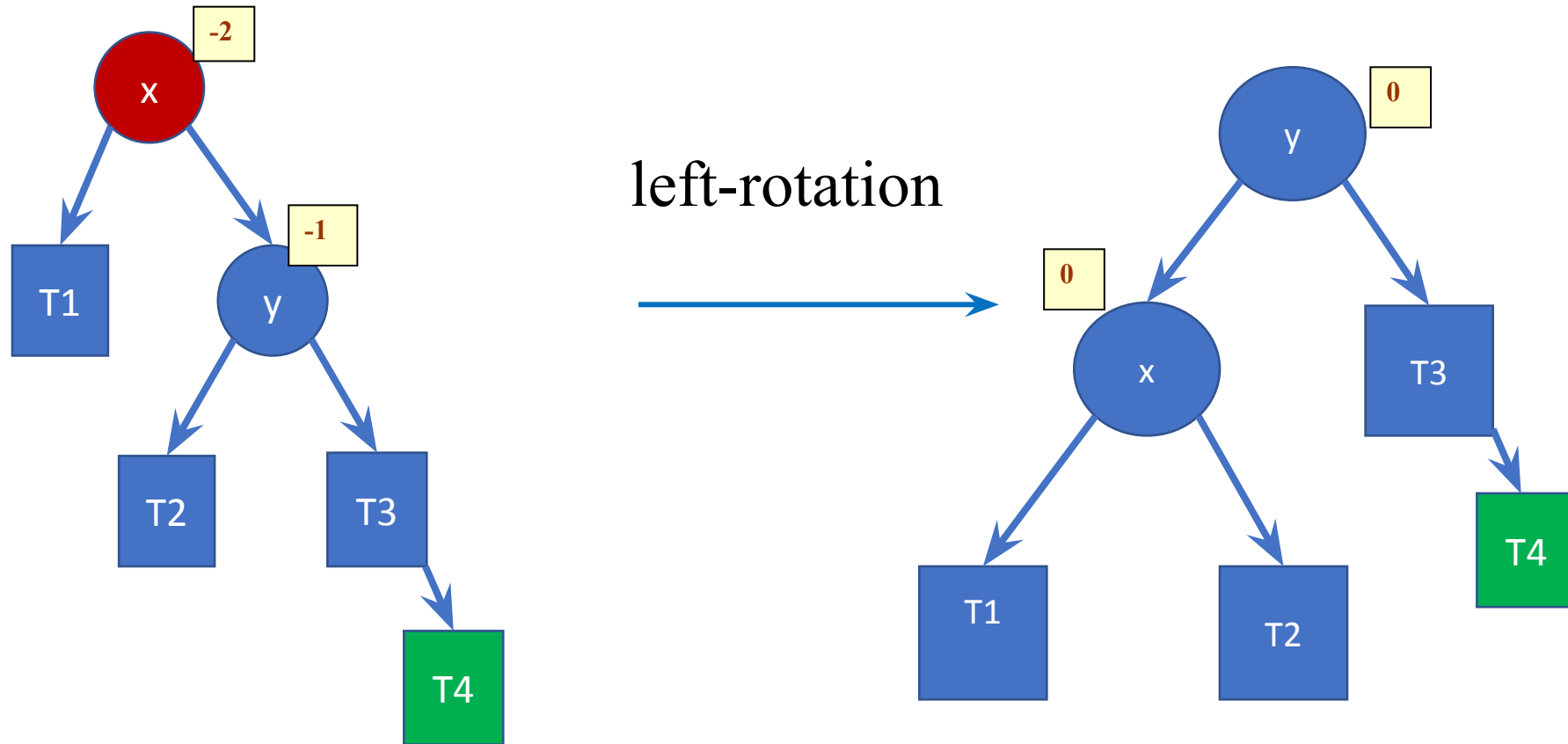
Case 2 $\text{balance-factor}(x) = 2$, $\text{balance-factor}(x \rightarrow \text{left}) < 0$

$x \rightarrow \text{left} = \text{left_rotate}(x \rightarrow \text{left}); \text{right_rotation}(x);$



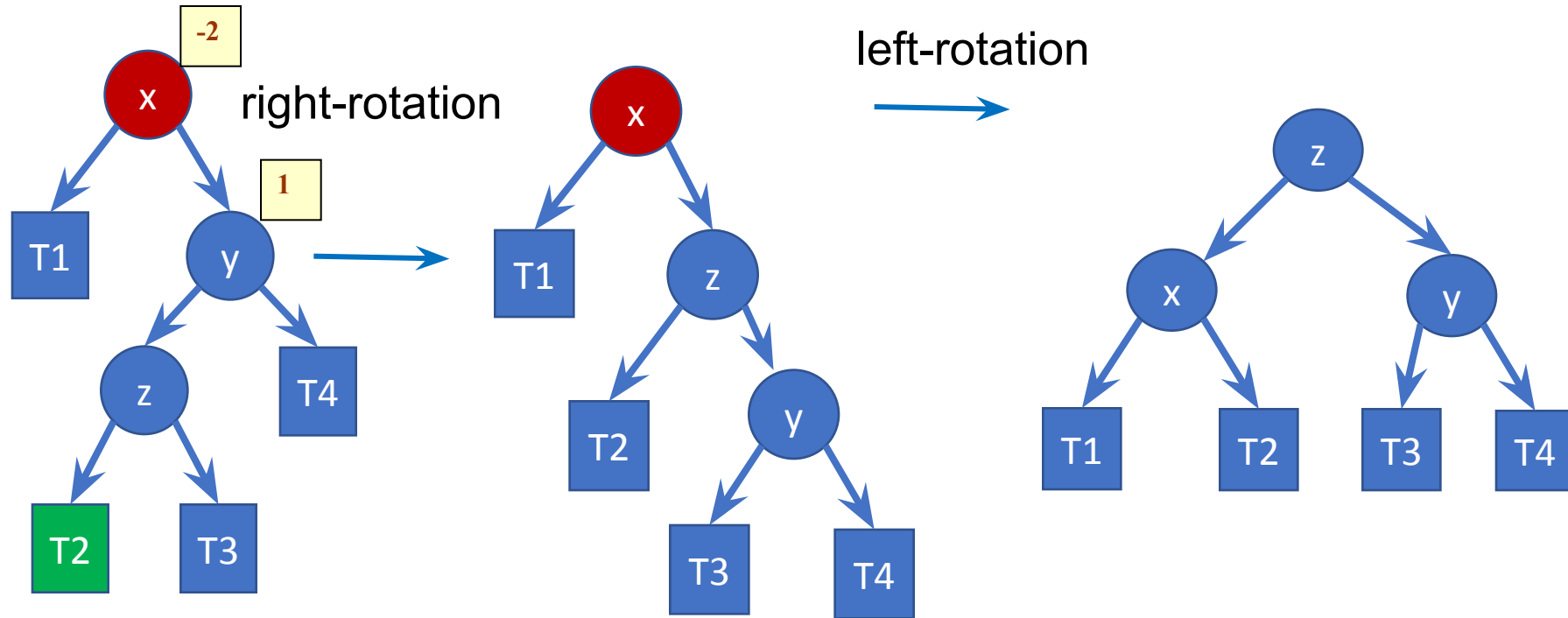
left-rotation to re-balance

Case 3: $\text{balance-factor}(x) = -2$, $\text{balance-factor}(x \rightarrow \text{right}) \leq 0$
`left_rotation(x);`



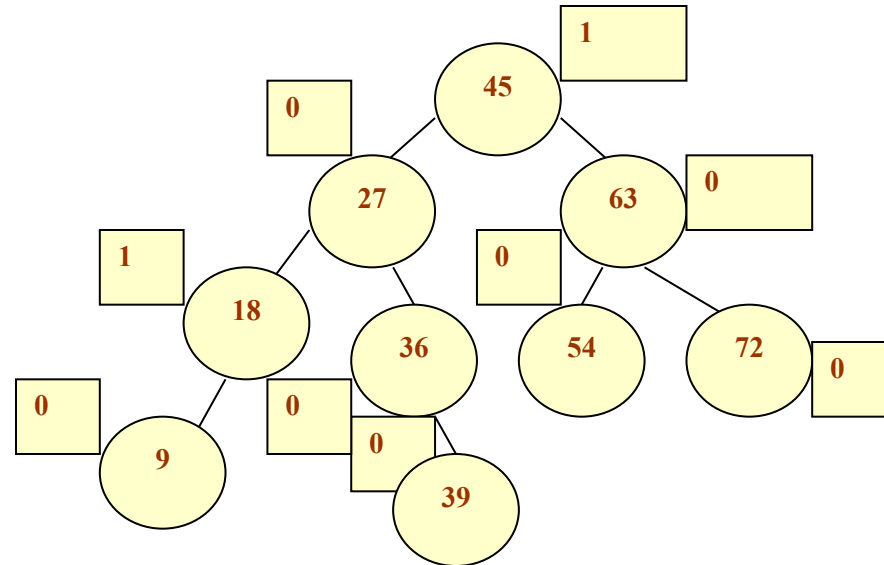
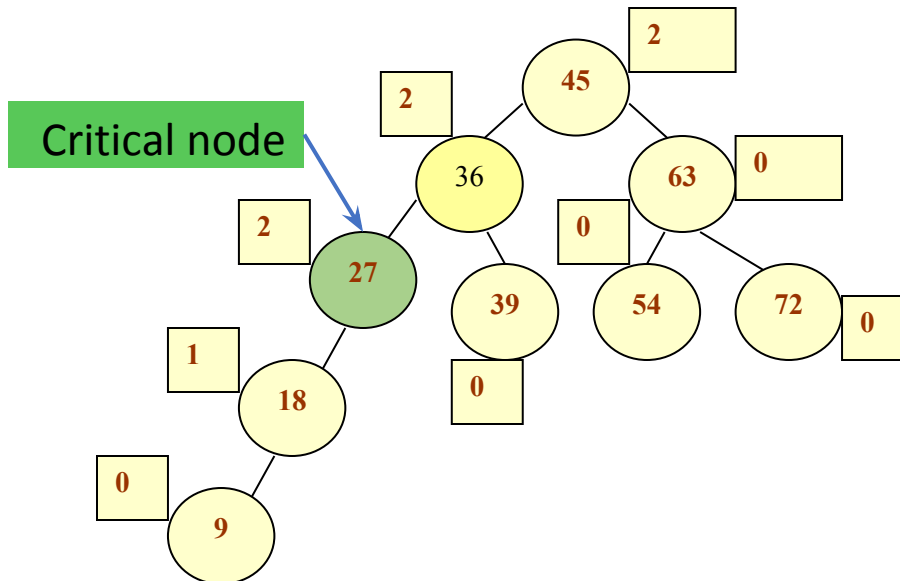
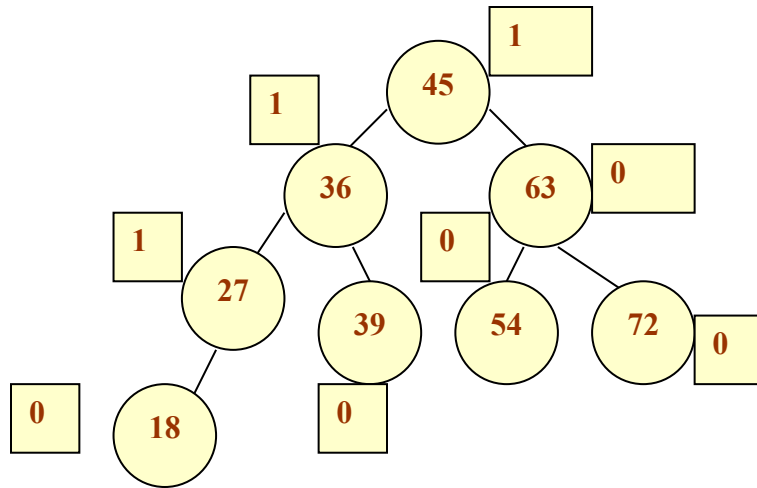
R-L-Rotation re-balancing

Case 4. $\text{balance-factor}(x) = -2$, $\text{balance-factor}(x \rightarrow \text{right}) > 0$
 $x \rightarrow \text{right} = \text{right_rotate}(x \rightarrow \text{right}); \text{left_rotation}(x);$



Case 2: R-Rotations to Balance AVL Tree

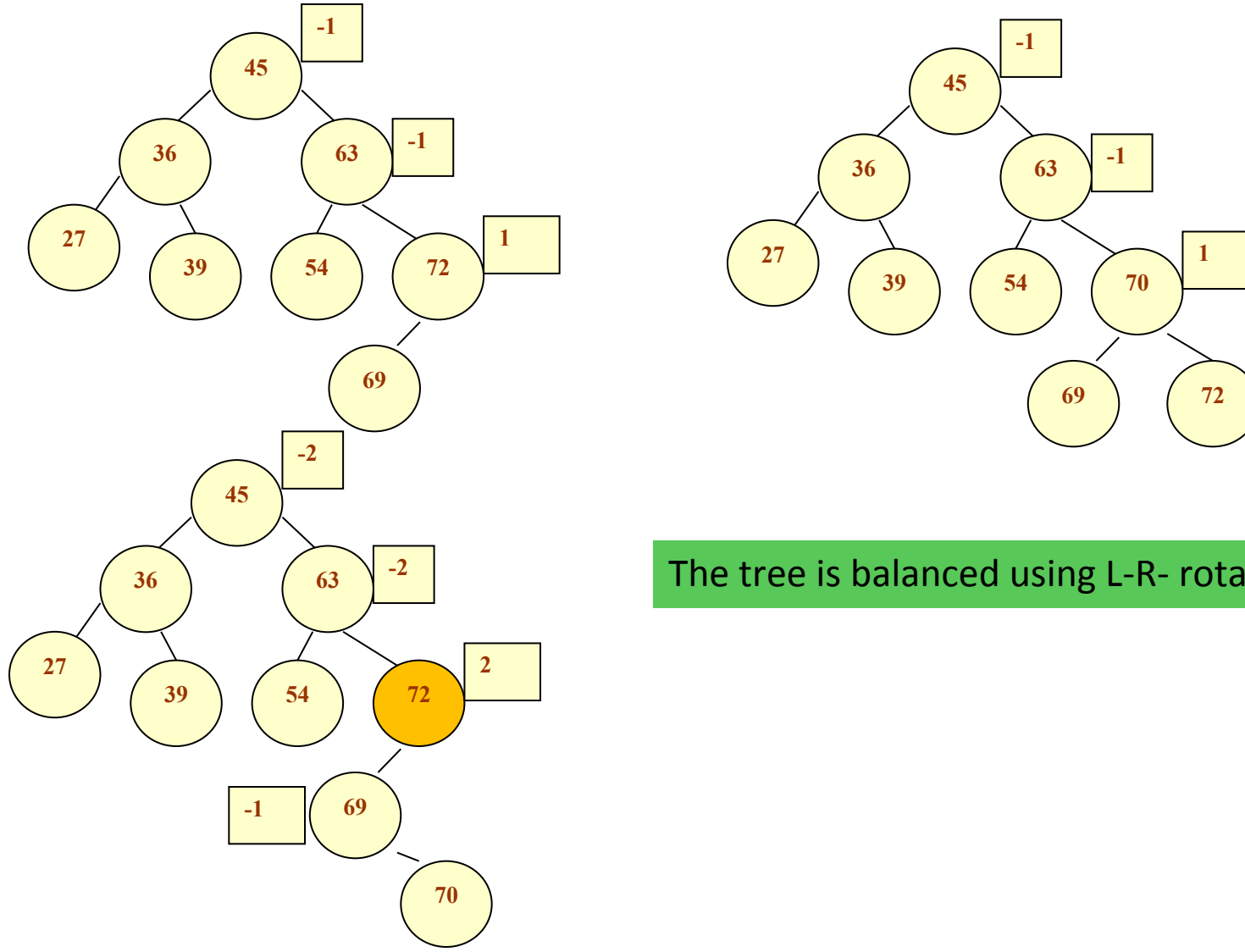
Example: Consider the AVL tree given below and insert 9 into it. LL case



The tree is balanced using R-rotation

Case 2: L-R-Rotation

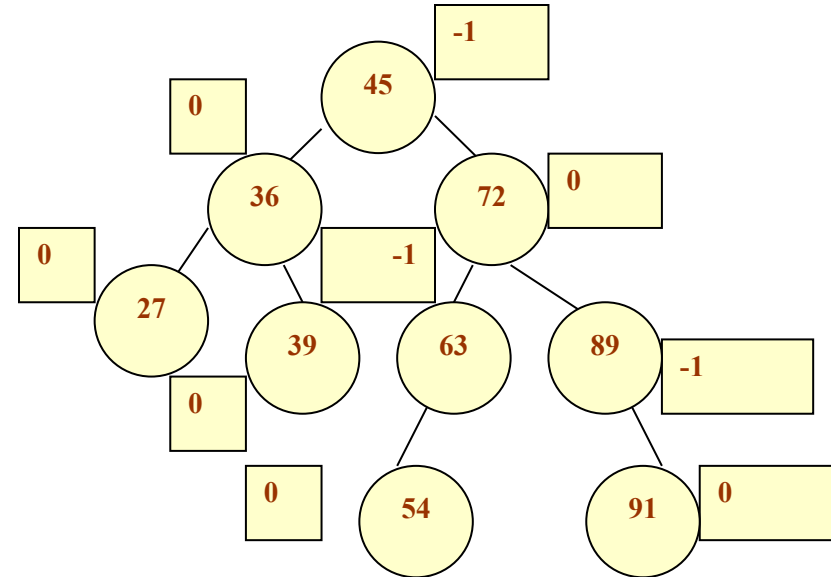
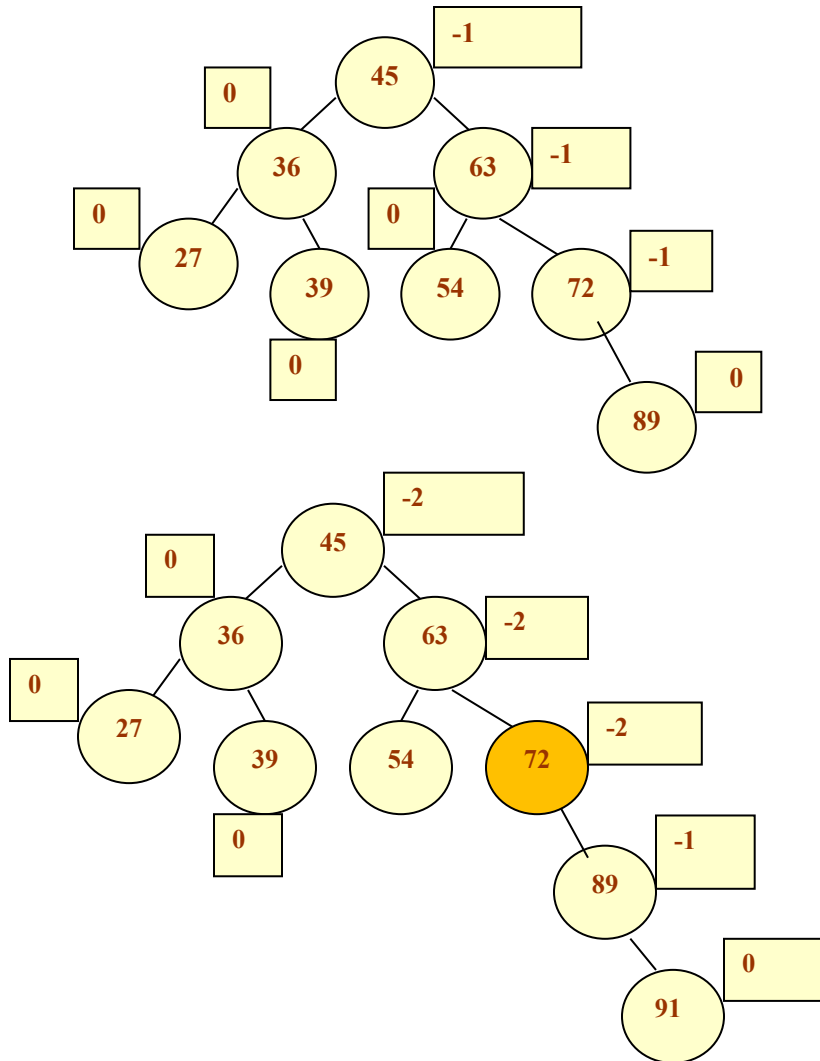
Example: Consider the AVL tree given below and insert 70 into it.



The tree is balanced using L-R- rotation

Case 3. L-Rotation to Balance

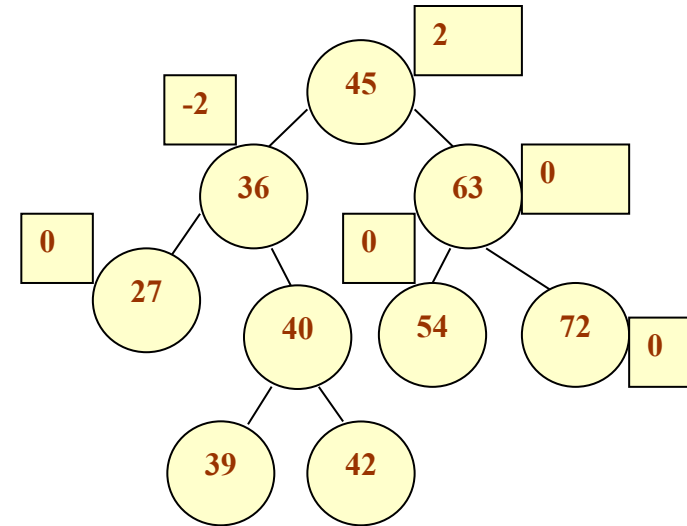
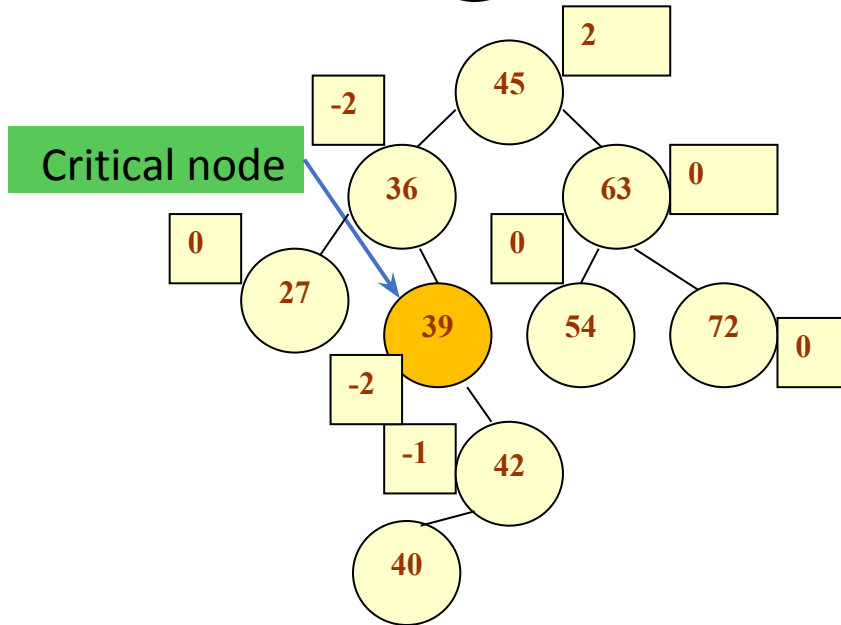
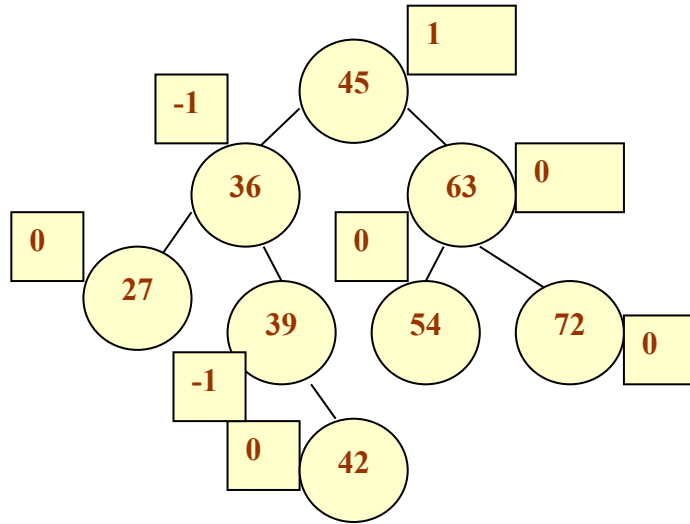
Example: Consider the AVL tree given below and insert 91 into it. RR-case



The tree is balanced using L-rotation

Case 4. R-L-Rotation to Balance

Example: Consider the AVL tree given below and insert 40 into it.



The tree is balanced using R-L rotation

Searching for a Node in an AVL Tree

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes $O(\log n)$ time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken.

Deleting a Node from an AVL Tree

- Deleting algorithm

- Step 1. delete node as BST

- Step 2. if not height balanced, rebalance by rotation

If the resulting BST of step 1 is not height-balanced, find the critical node. There are four possible cases similar to the inserting. Then do corresponding rotation by the case

Rotate to rebalance

Case 1: $\text{balance-factor}(x) = 2$, $\text{balance-factor}(x \rightarrow \text{left}) \geq 0$,
 $\text{right_rotation}(x);$

Case 2: $\text{balance-factor}(x) = 2$, $\text{balance-factor}(x \rightarrow \text{left}) < 0$,
 $x \rightarrow \text{left} = \text{left_rotate}(x \rightarrow \text{left}); \text{right_rotation}(x);$

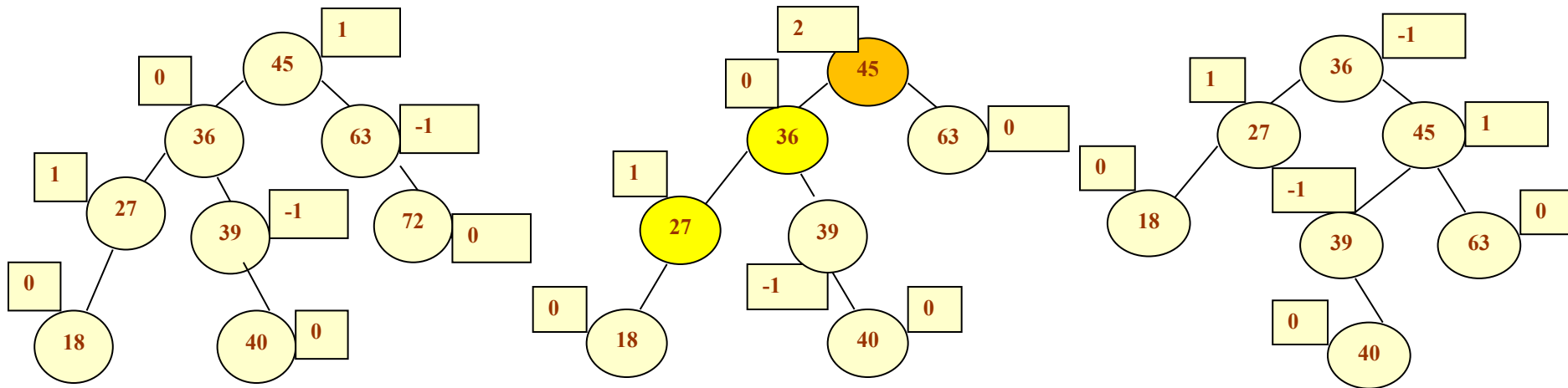
Case 3: $\text{balance-factor}(x) = -2$, $\text{balance-factor}(x \rightarrow \text{right}) \leq 0$,
 $\text{left_rotation}(x);$

Case 4: $\text{balance-factor}(x) = -2$, $\text{balance-factor}(x \rightarrow \text{right}) > 0$
 $x \rightarrow \text{right} = \text{right_rotate}(x \rightarrow \text{right}); \text{left_rotation}(x);$

Deleting a Node from an AVL Tree

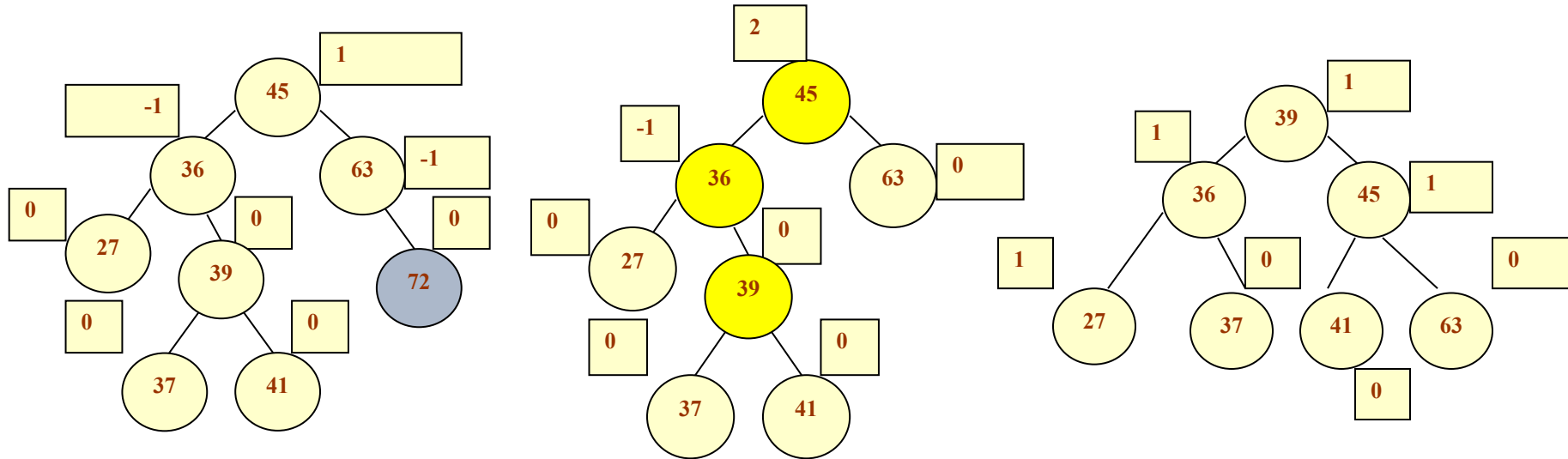
Example: Consider the AVL tree given below and delete 72 from it.

Case 1. R-rotation



Deleting a Node from an AVL Tree

- Consider the AVL tree given below and delete 72 from it.
- Case 2. L-R-rotation



AVL implementation node design

```
struct Node
{
    int key;
    int height; // or using balance factor
    int data;    // application data associated with key
    struct Node *left, *right;
};
```

AVL right-rotate

```
node *right_rotate(node *y) {  
    node *x = y->left;  
    node *T2 = x->right;  
    // perform rotation  
    x->right = y;  
    y->left = T2;  
    // Update heights  
    y->height = max(height(y->left), height(y->right))+1;  
    x->height = max(height(x->left), height(x->right))+1;  
    return x;  
}
```

Diagram illustrating the right rotation process:

Initial State (Left):

```
      y  
     /\n    x  T3  
   /\  \n  T1 T2
```

Right Rotation: The node x becomes the new root, and y becomes its right child. The subtree rooted at T3 is moved to be the right child of x.

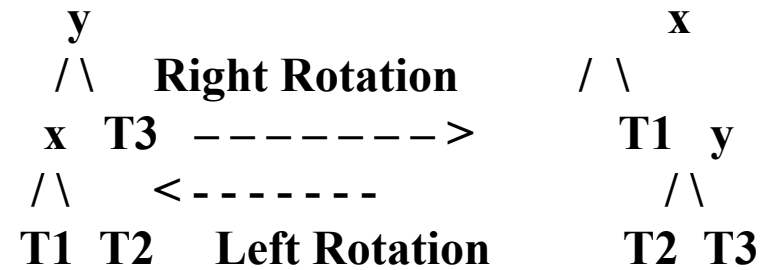
Final State (Right):

```
      x  
     /\n    T1 y  
   /\  \n  T2 T3
```

Left Rotation: The node y becomes the new root, and x becomes its left child. The subtree rooted at T2 is moved to be the left child of y.

AVL left-rotate

```
node *left_rotate(node *x)
{
    node *y = x->right;
    node *T2 = y->left;
    // perform rotation
    y->left = x;
    x->right = T2;
    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
    return y;
}
```



AVL insert

```
node* insert(node* root, int key) {  
    if (root == NULL) return(new_node(key));  
    if (key < root->key)  
        root->left = insert(root->left, key);  
    else if (key > root->key)  
        root->right = insert(root->right, key);  
    else return root;  
  
    root->height = 1 + max(height(root->left), height(root->right));  
    int balance = balance_factor(root);  
  
    // do rotation by cases
```

AVL implementation insert

```
if (balance > 1 && balance_factor(root->left) >=0 )  
    return right_rotate(root);  
if (balance < -1 && balance_factor(root->left) <=0 )  
    return left_rotate(root);  
if (balance > 1 && balance_factor(root->left) < 0) {  
    root->left = left_rotate(root->left);  
    return right_rotate(root);  
}  
if (balance < -1 && balance_factor(root->left) > 0) {  
    root->right = right_rotate(root->right);  
    return left_rotate(root);  
}  
return root; }
```

**Thank
You**

