كلية الهندسة قسم المعلوماتية

بنى معطيات 1

Data Structure 1

ا. د. علي عمران سليمان

محاضرات الأسبوع العاشرة

أشجار

AVL

الفصل الثاني 2025-2026

محتوى الفصل

AVLTrees	
1-Introduction to the AVL Tree	
2- Why AVL Trees?.	
3-Operations on an AVL Tree	
4-Rotating the subtrees in an AVL Tree	
5-Insertion in an AVL Tree	
6-Deletion in an AVL Tree	
7-Searching in an AVL Tree	
8-Illustration of Insertion at AVL Tree	
9-Advantages and Disadvantages of AVL Tree	

References

- Deitel & Deitel, Java How to Program, Pearson; 10th Ed(2015)

https://www.geeksforgeeks.org ›

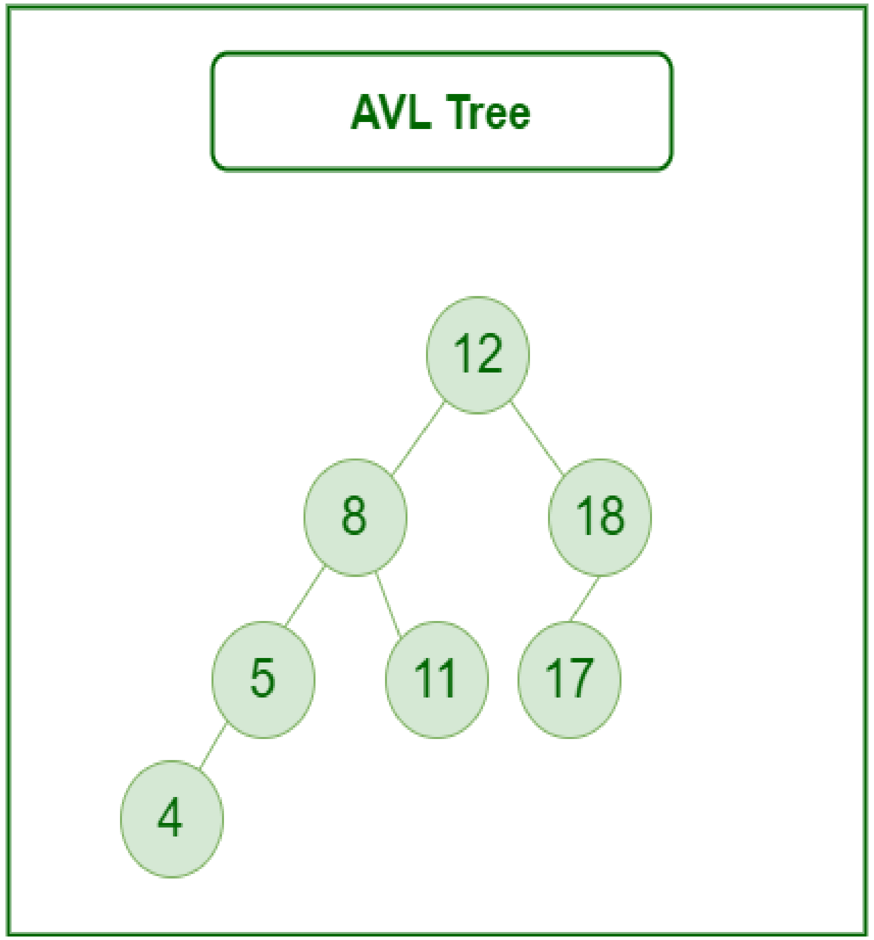
AVL tree 1

An AVL tree defined as a self-balancing [Binary Search Tree]() (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.

The AVL tree is named after its inventors, Georgy Adelson- Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.

AVL tree 2



Balance Factor(k) = height(left(k))-height(righ(k))

<u>BF(k) = {0 , 1, -1}</u>

<u>|BF(k)| <=1   is balanced AVL tree</u>

The tree on the left is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

Why AVL Trees?

*Most of the BST operations (e.g., **search**, **max**, **min**, **insert**, **delete**.. etc) take **O(h)** time where **h** is the height of the BST. The cost of these operations may become **O(n)** for a **skewed Binary tree**. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.*

Operations on an AVL Tree 1

Operations on an AVL Tree:

- Insertion
- Deletion
- Searching [It is similar to performing a search in BST]

Rotating the subtrees in an AVL Tree:

An AVL tree may rotate in one of the following four ways to keep itself balanced:

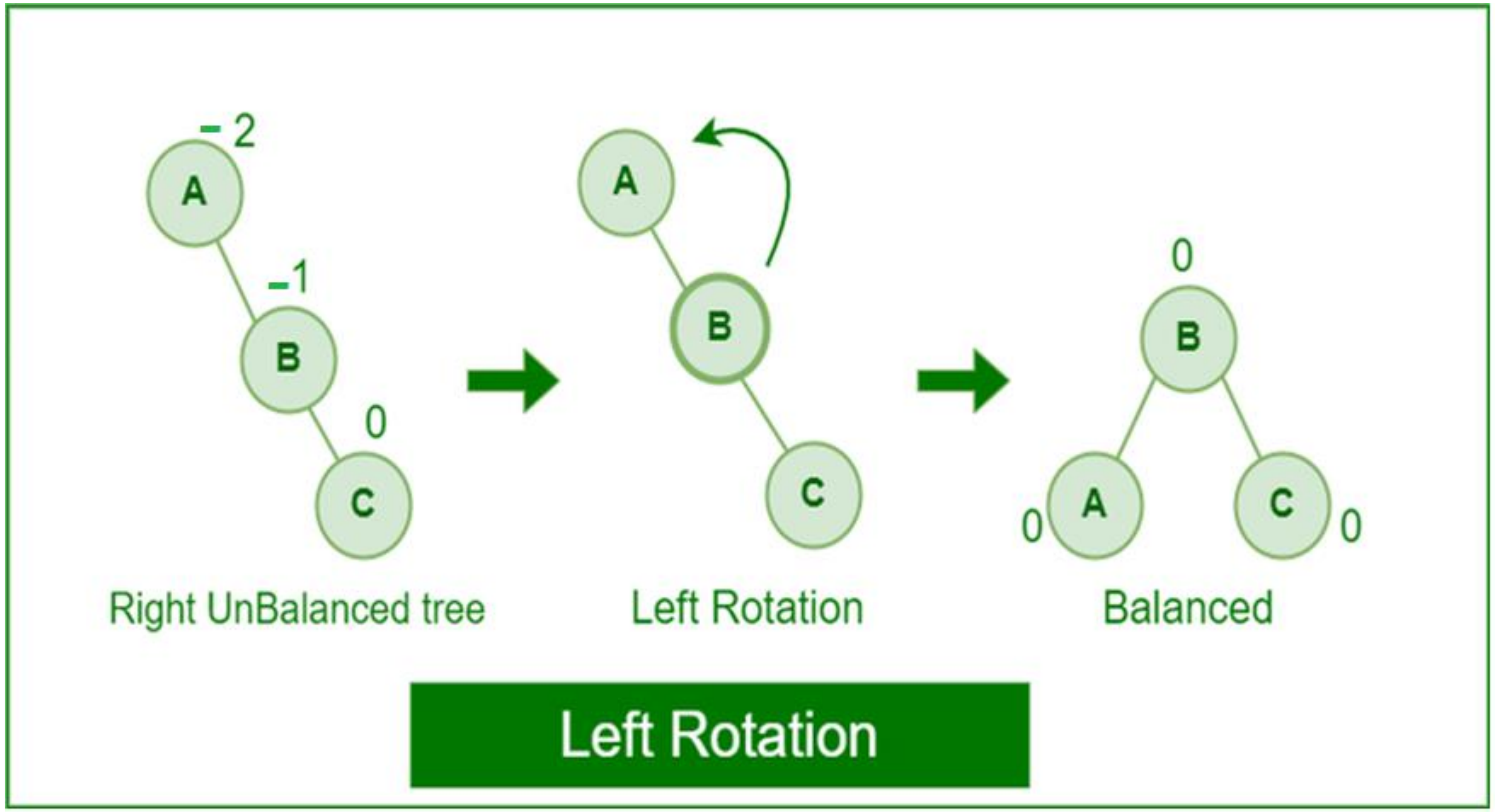
Left Rotation, Right Rotation, Left-Right Rotation, Right-Left Rotation>

Operations on an AVL Tree 2

Left Rotation:

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation

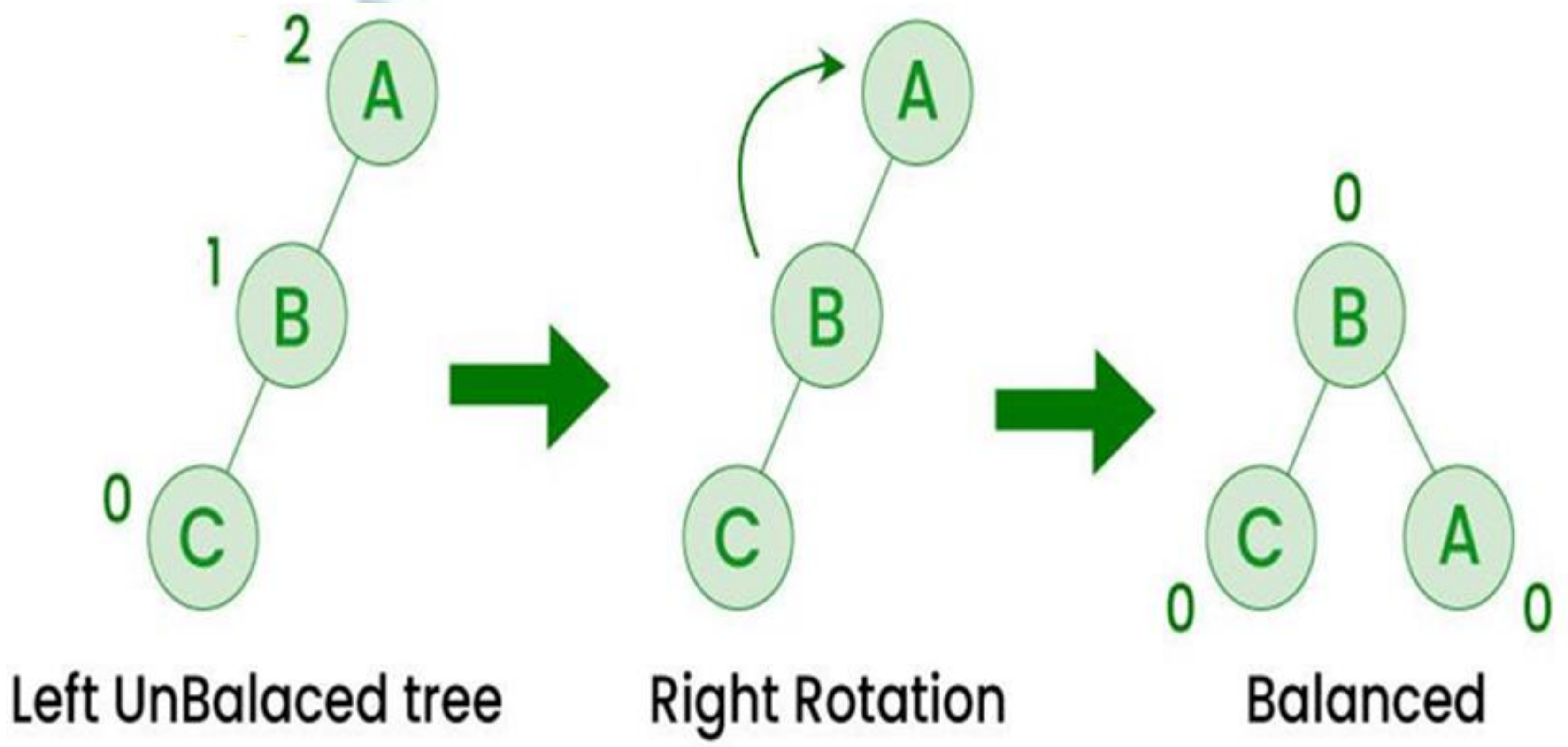
(anticlockwise rotation).



Operations on an AVL Tree 3

Right Rotation:

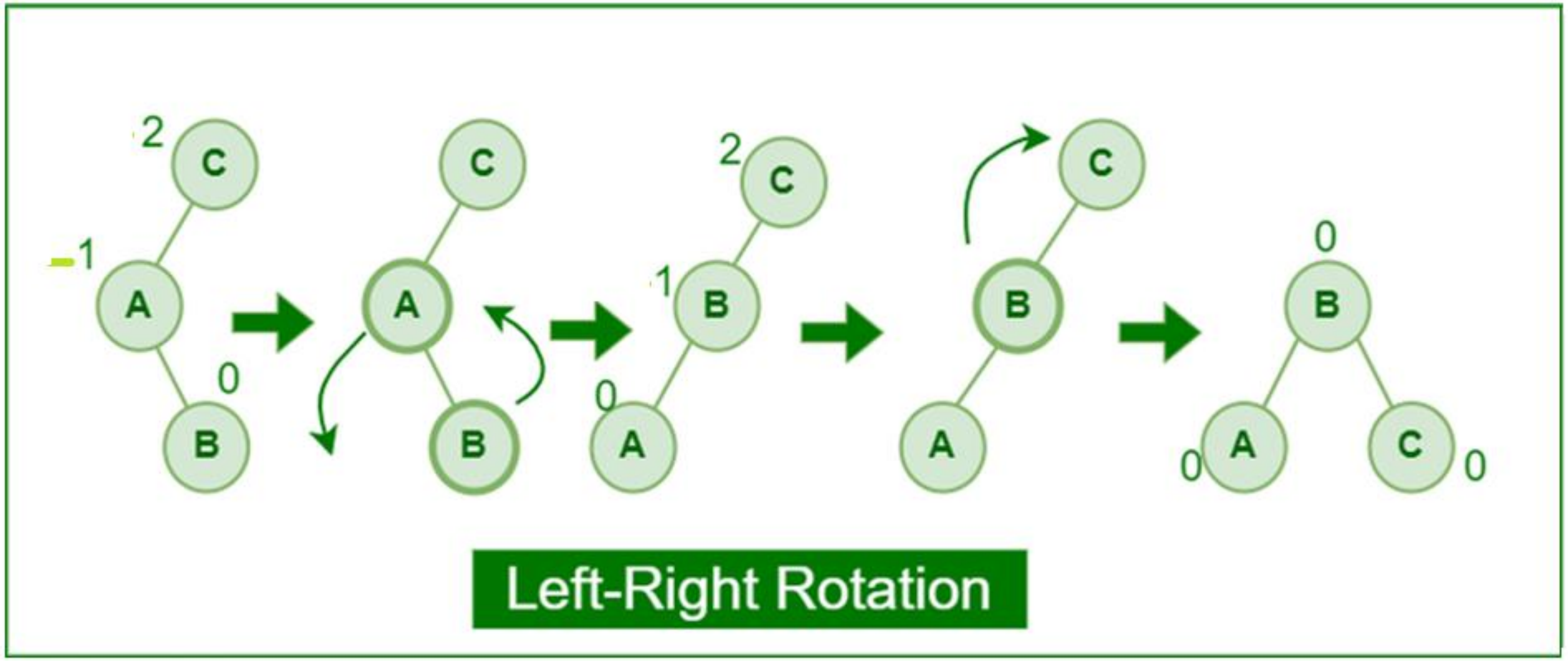
If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation (clockwise rotation).



Operations on an AVL Tree 4

Left-Right Rotation:

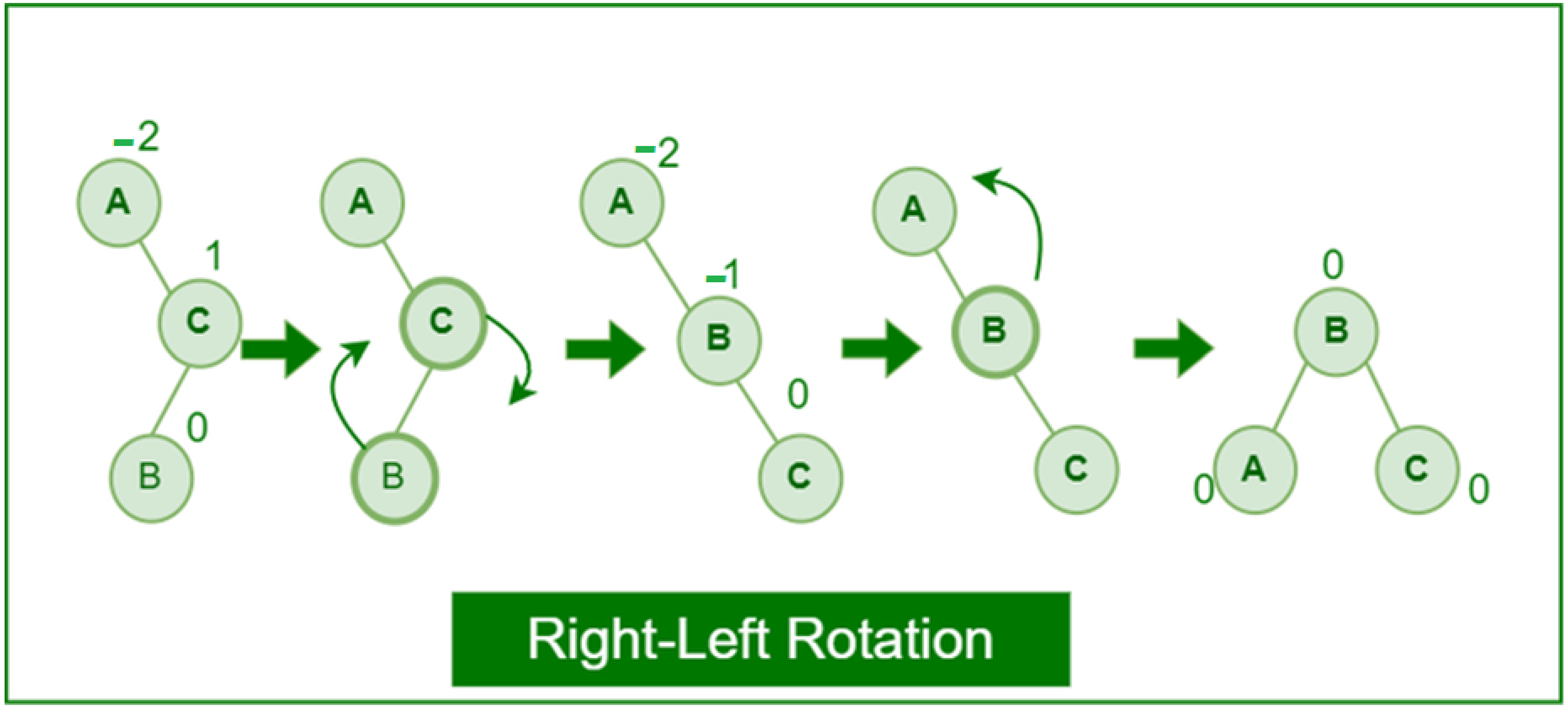
A left-right rotation is a combination in which first left rotation takes place, after that right rotation executes.



Operations on an AVL Tree 5

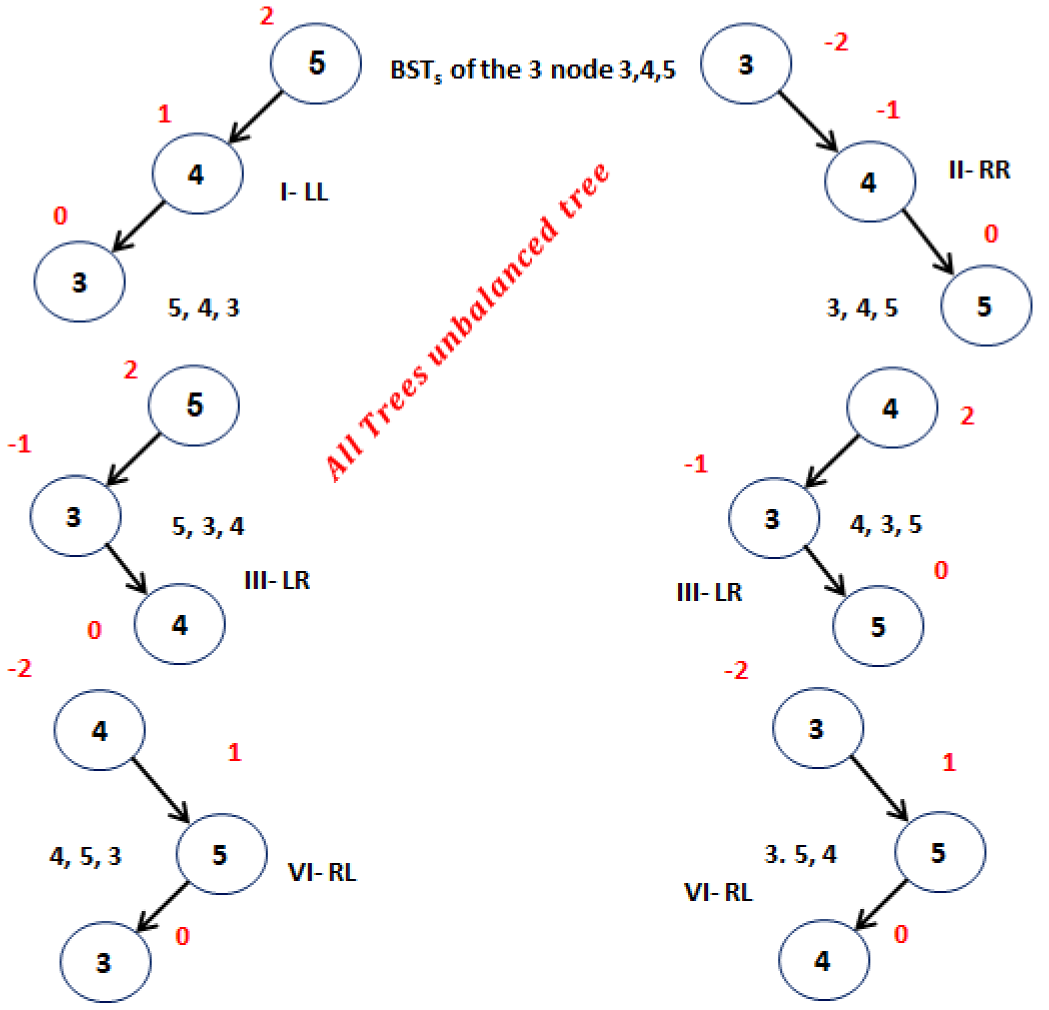
Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place, after that left rotation executes.



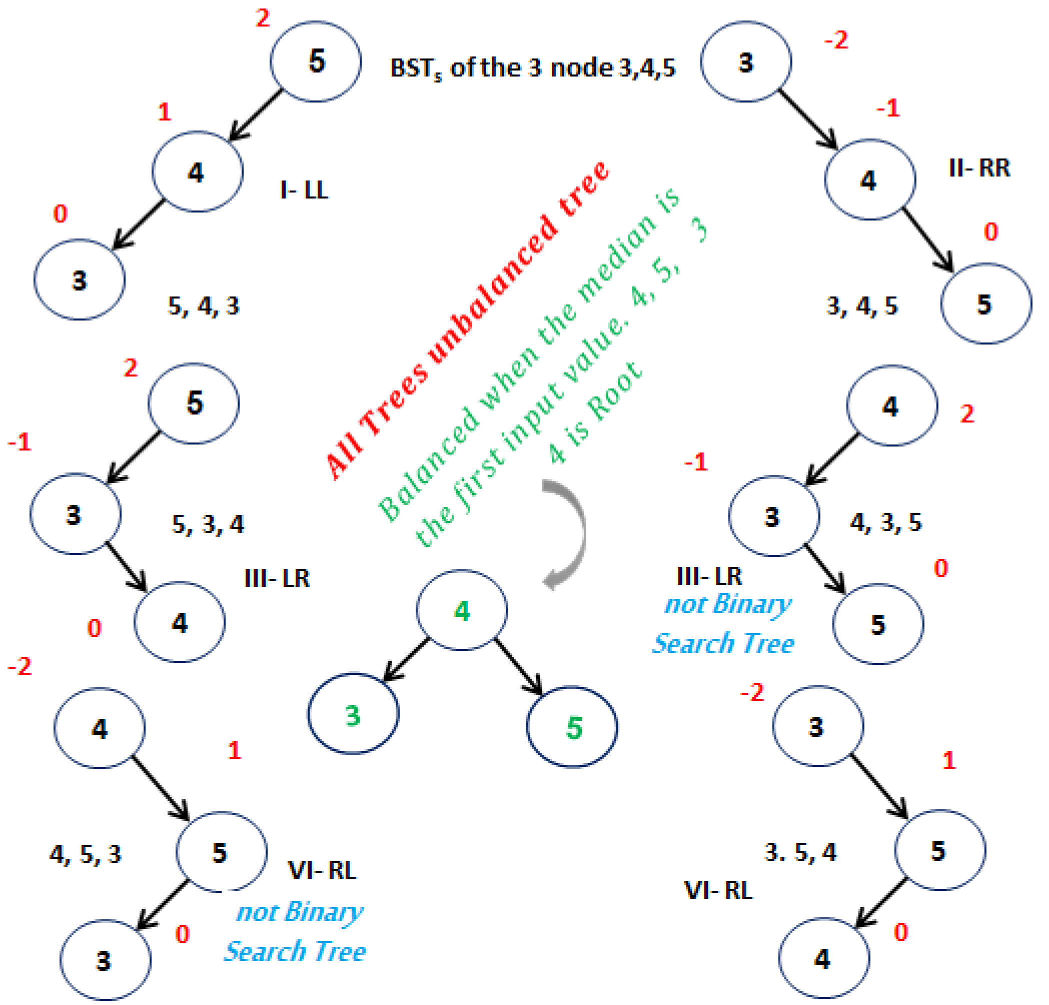
Operations on an AVL Tree 1

If we have three nodes (30, 40, 50), then the number of possible formations is 3! = 6



Operations on an AVL Tree 1

If we have three nodes (30, 40, 50), then the number of possible formations is 3! = 6

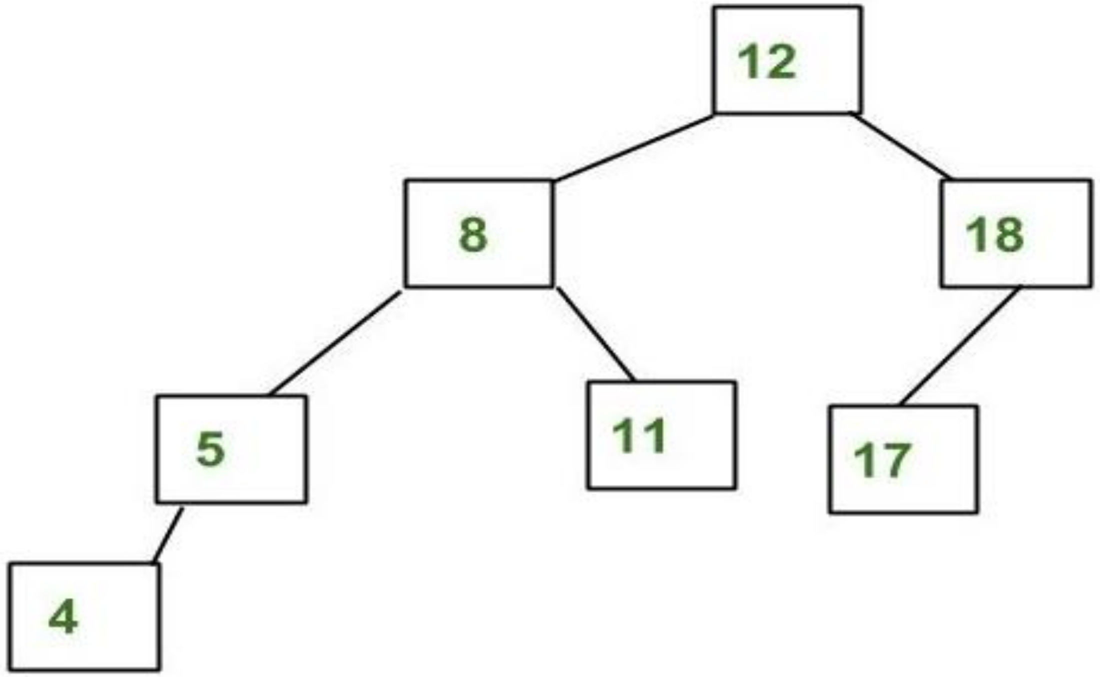


Insertion in an AVL Tree

AVL Tree:

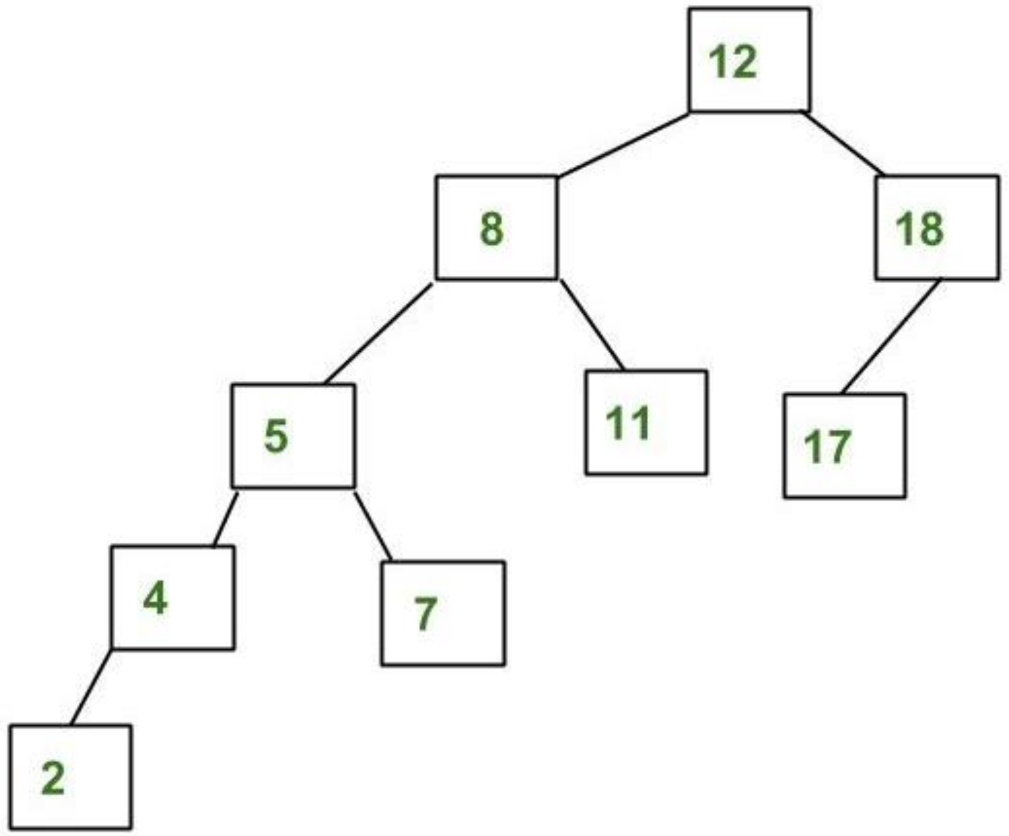
AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Insert a node with value 2 The tree is not an AVL tree



Insertion in an AVL Tree 2

The tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1(in absolute value).



Insertion in an AVL Tree 3

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

Insertion in an AVL Tree 4

```
       y                              x
      / \   Right Rotation          /  \
    x     T3   - - - - - - - >    T1      y
   / \         < - - - - - - -           / \
 T1  T2      Left Rotation             T2  T3
```

Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.

Left Left Case

1. Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z
        / \
       y   T4
      / \
     x   T3
    / \
  T1   T2
```

Left Left Case

1. Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /     \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1    T2
```

Left Right Case

2. Left Right Case

```
         z
       /   \
     y       T4
   /   \
T1       x
       /   \
     T2      T3
```

Left Right Case

2. Left Right Case

```
     z                                z
    / \                             /   \
   y   T4  Left Rotate (y)         x    T4
  / \      - - - - - - - - ->     /  \
T1   x                           y    T3
    / \                         / \
  T2   T3                     T1   T2
```

Left Right Case

2. Left Right Case

```
     z                               z                            x
    / \                            /   \                         /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)     y      z
  / \      - - - - - - - - ->    /   \     - - - - - - - ->   / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

3. Right Right Case

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

Right Left Case

4. Right Left Case

```
     z
   /  \
T1     y
      /  \
     x    T4
   /  \
T2     T3
```

Right Left Case

4. Right Left Case

```
   z                            z
  / \                          / \
T1   y   Right Rotate (y)    T1    x
    / \  - - - - - - - - ->     /   \
   x   T4                      T2   y
  / \                              /   \
T2   T3                           T3   T4
```

Right Left Case

4. Right Left Case

```
    z                           z                            x
   / \                         / \                          /  \
T1   y   Right Rotate (y)    T1    x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->    /  \   - - - - - - - ->  / \    / \
   x   T4                     T2    y                 T1  T2  T3  T4
  / \                              /  \
T2   T3                          T3    T4
```

Illustration of Insertion at AVL Tree 1

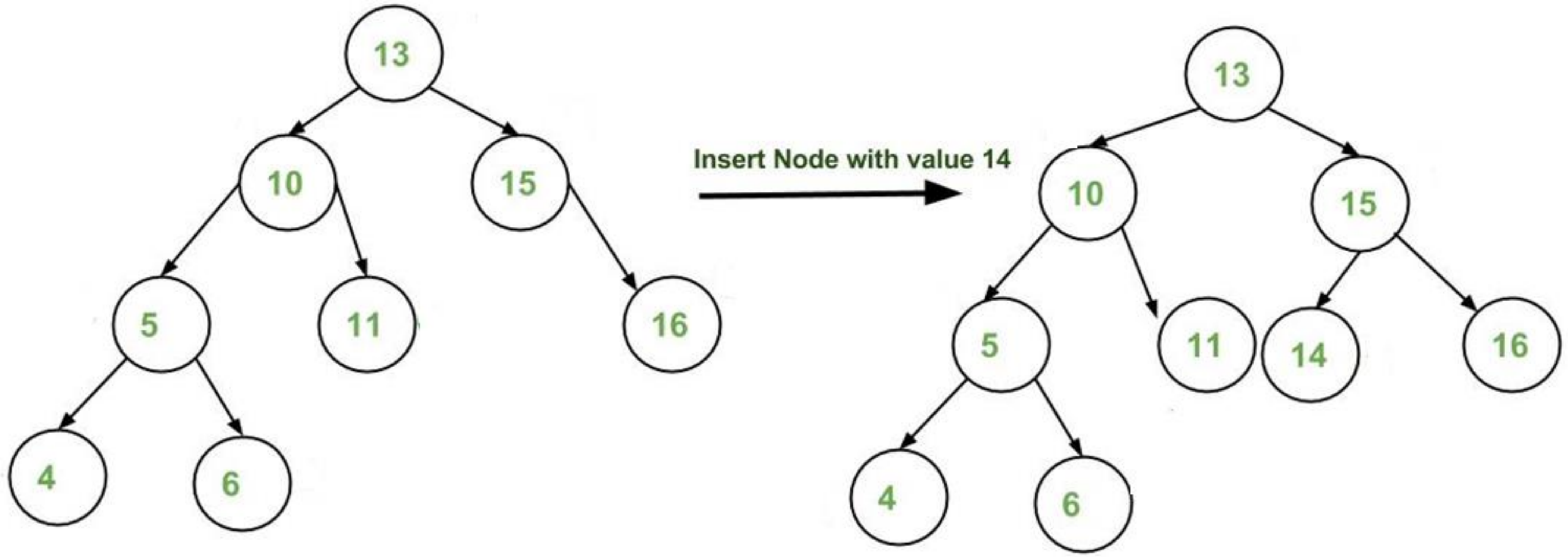


Illustration of Insertion at AVL Tree 1

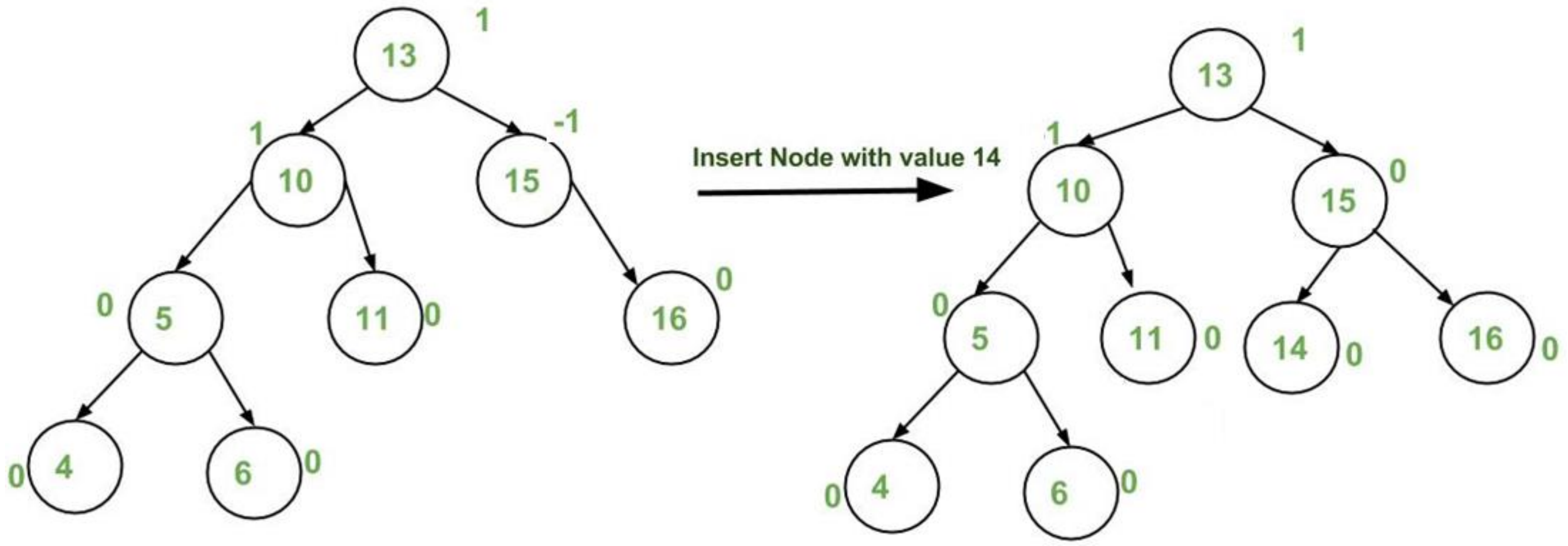


Illustration of Insertion at AVL Tree 2

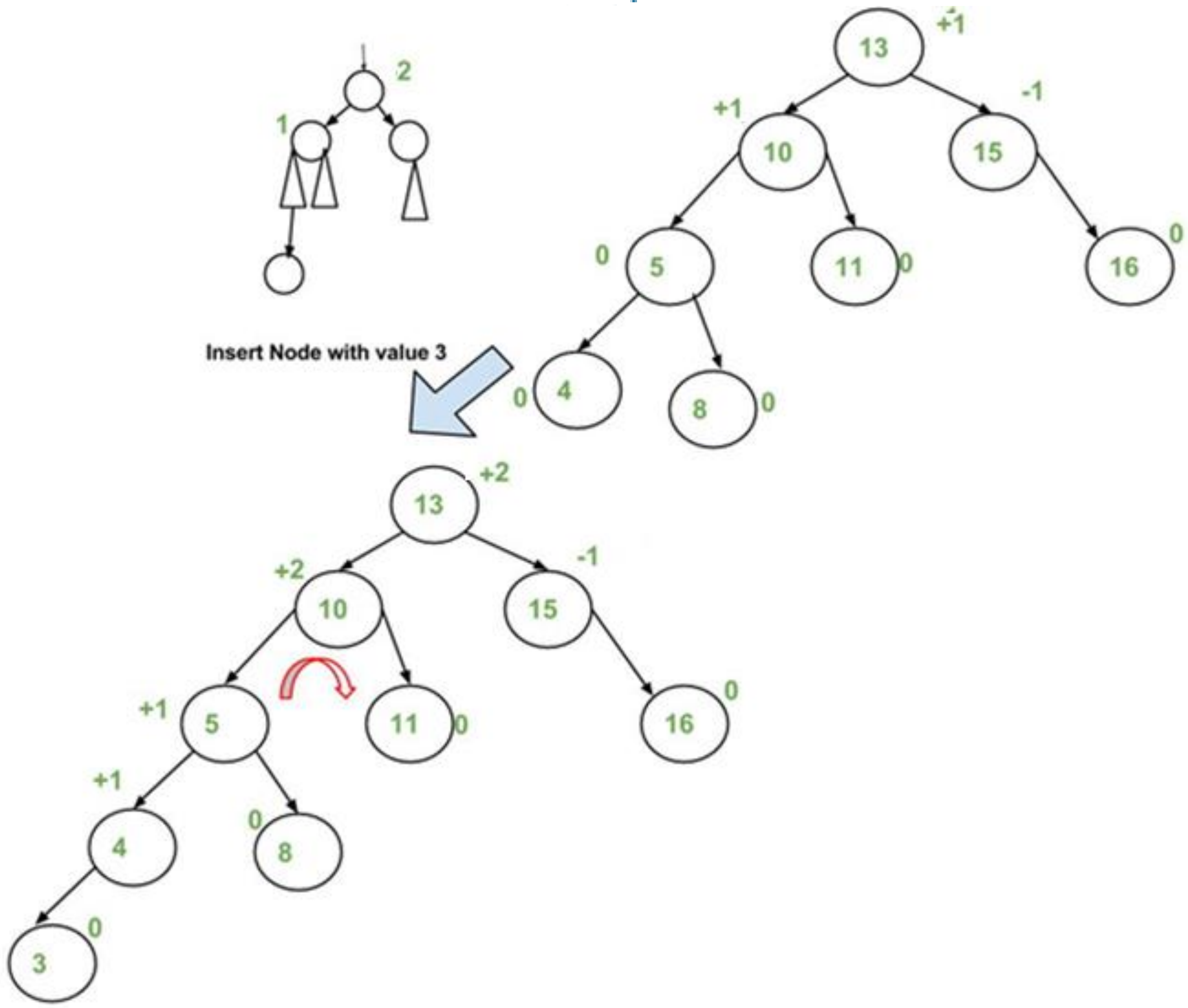


Illustration of Insertion at AVL Tree 2

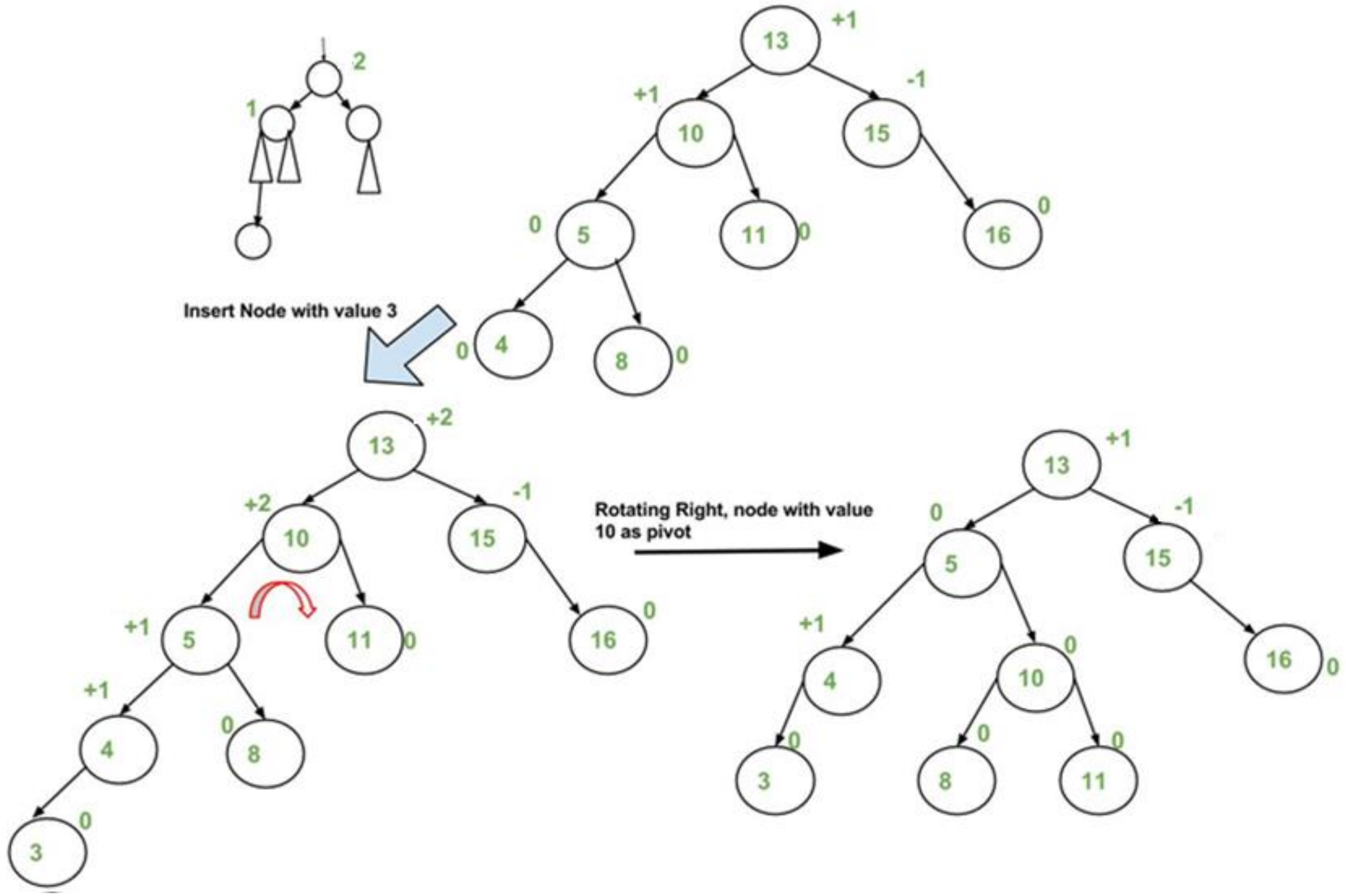


Illustration of Insertion at AVL Tree 3

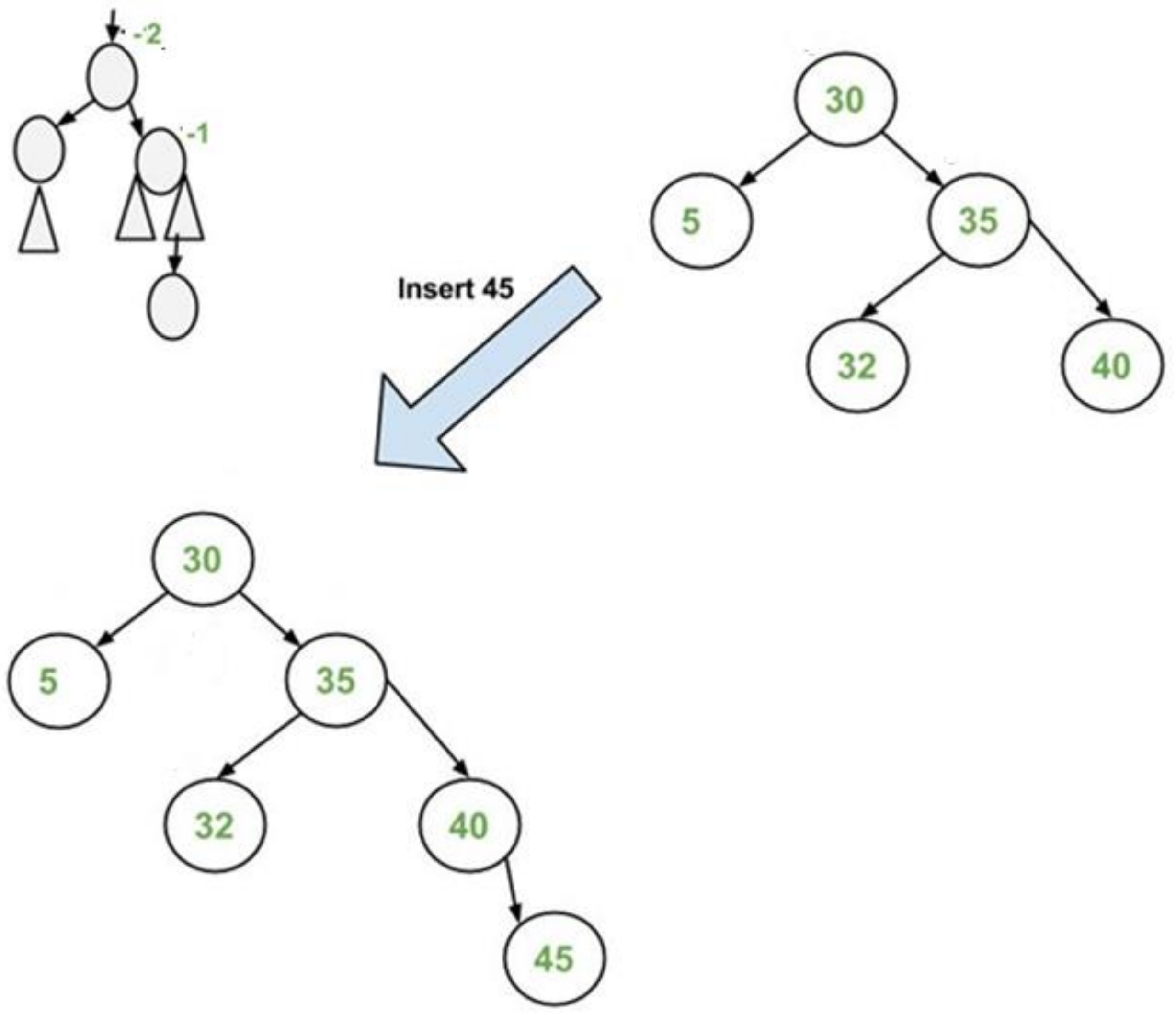


Illustration of Insertion at AVL Tree 3

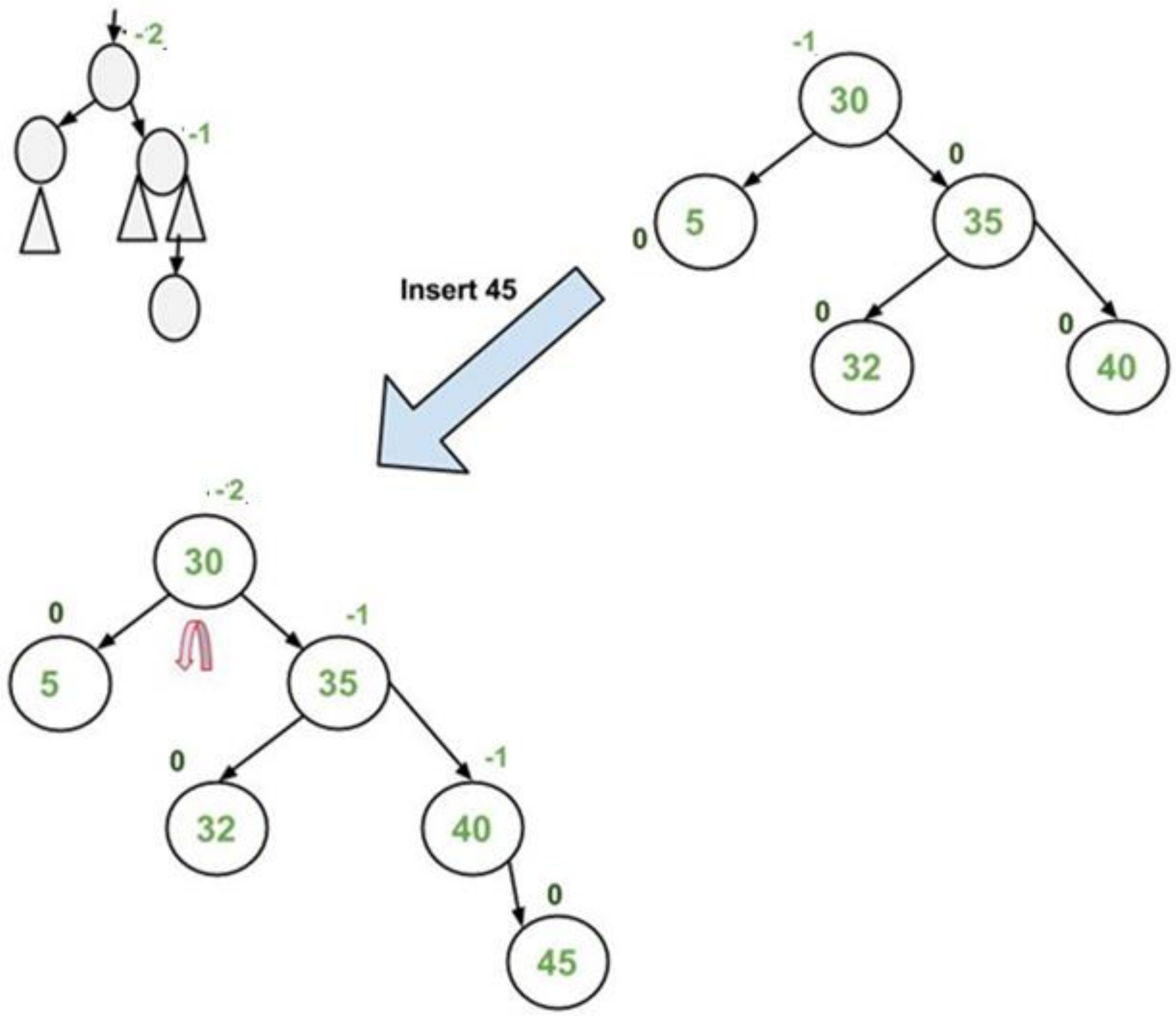


Illustration of Insertion at AVL Tree 3

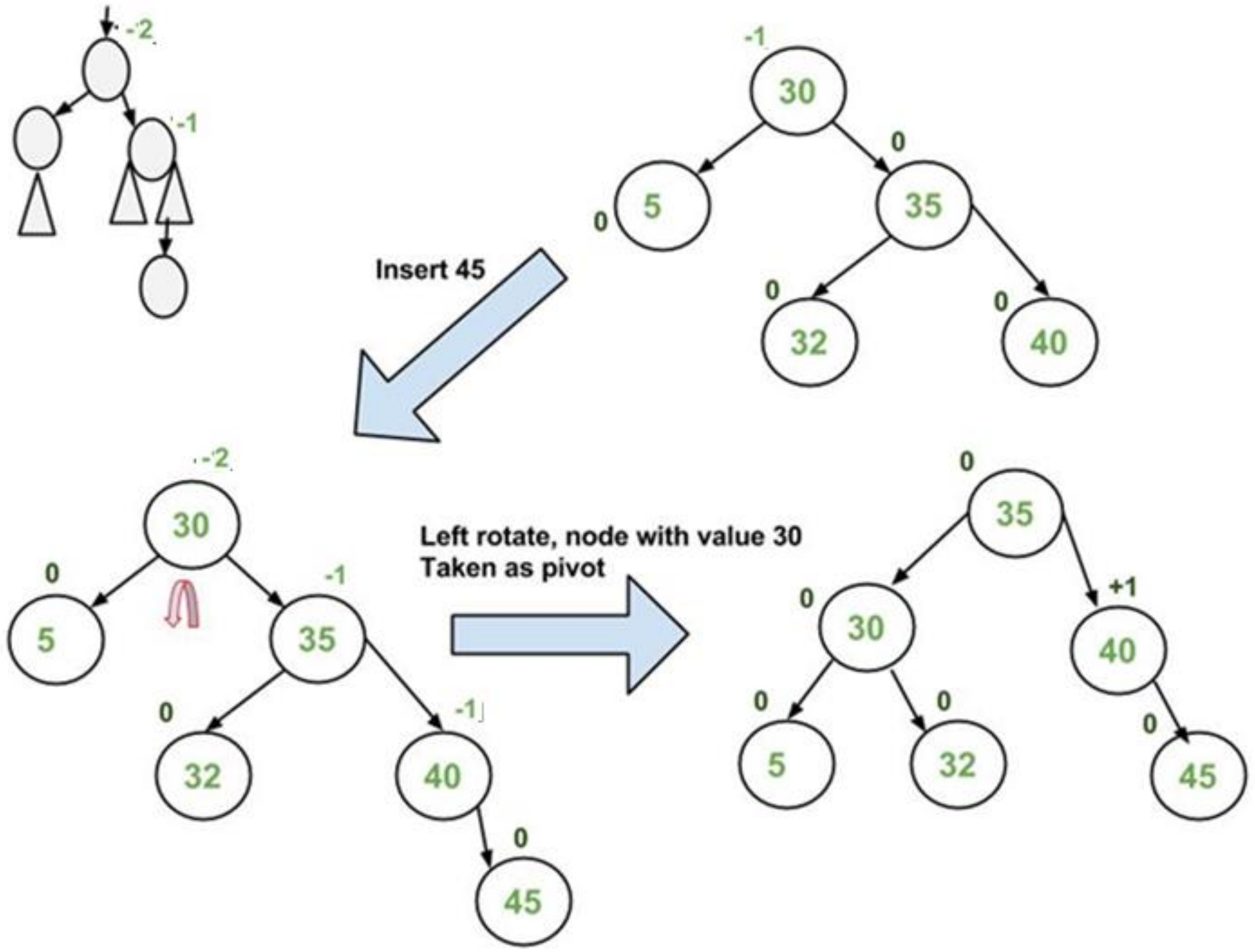


Illustration of Insertion at AVL Tree 4

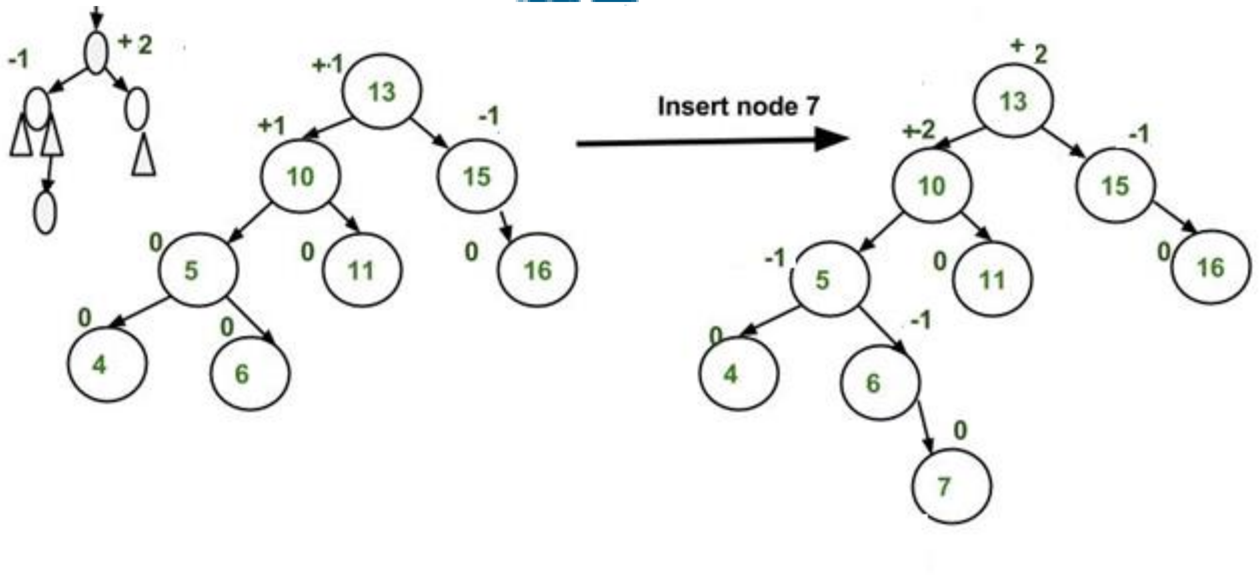


Illustration of Insertion at AVL Tree 4

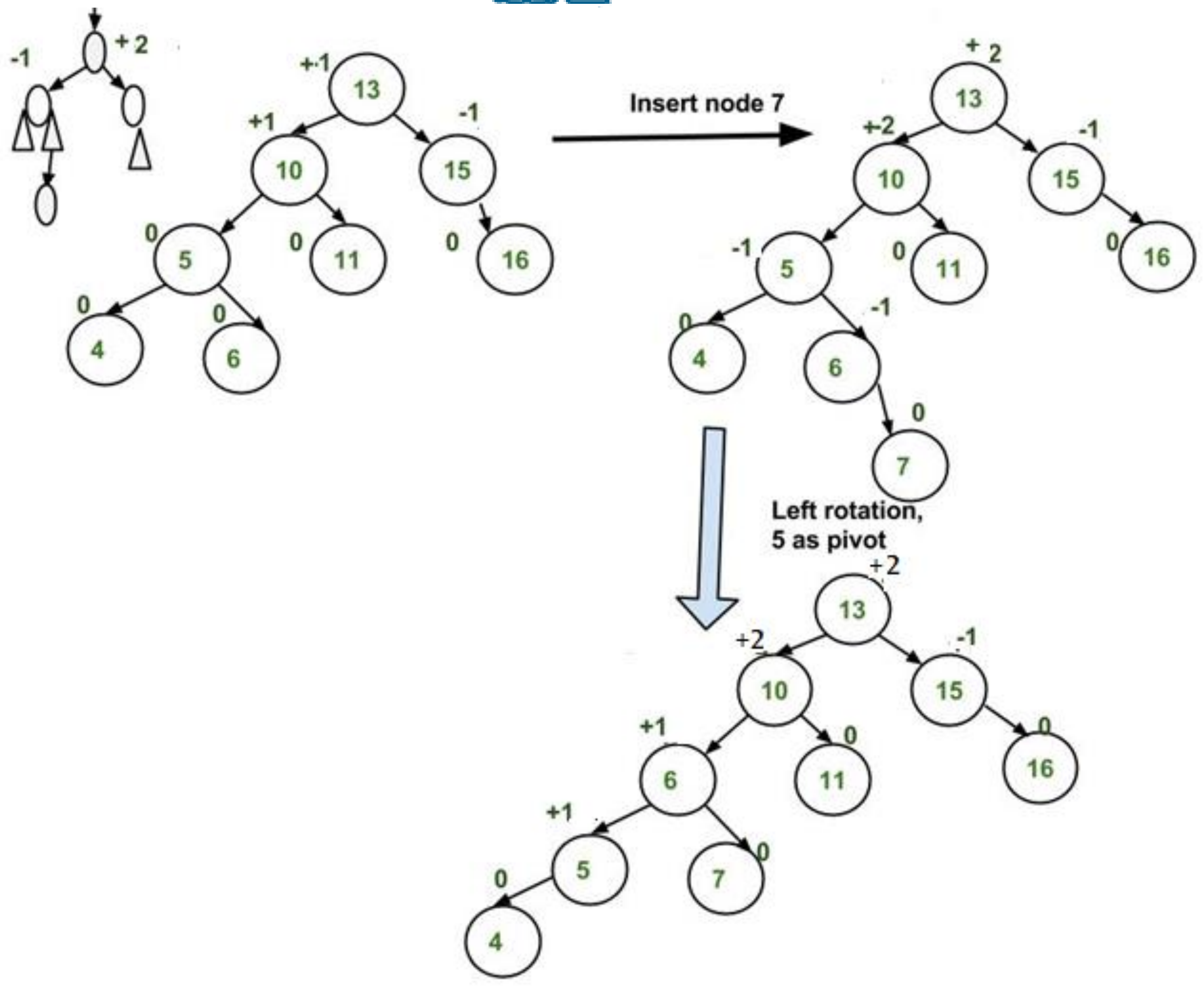


Illustration of Insertion at AVL Tree 4

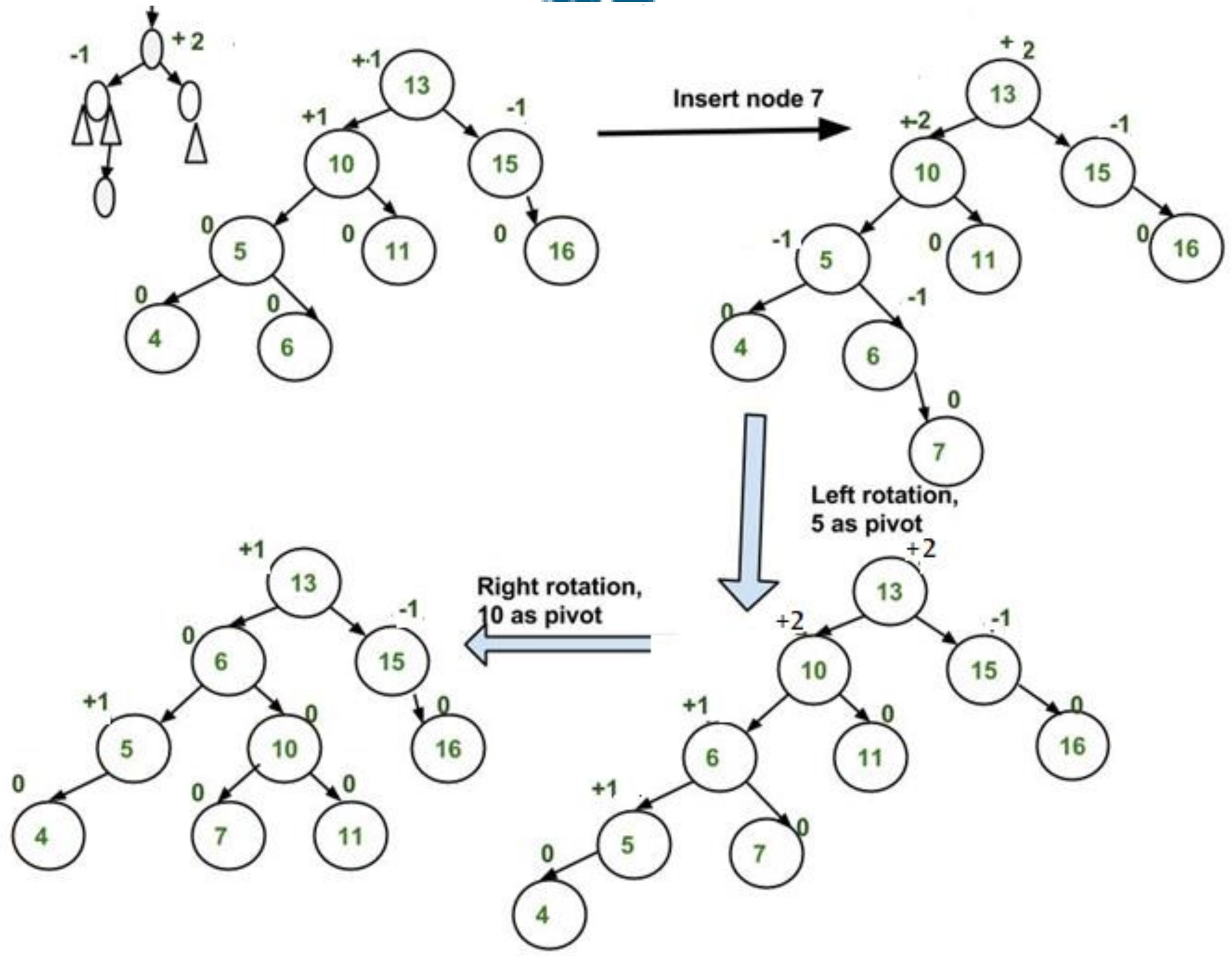


Illustration of Insertion at AVL Tree 5

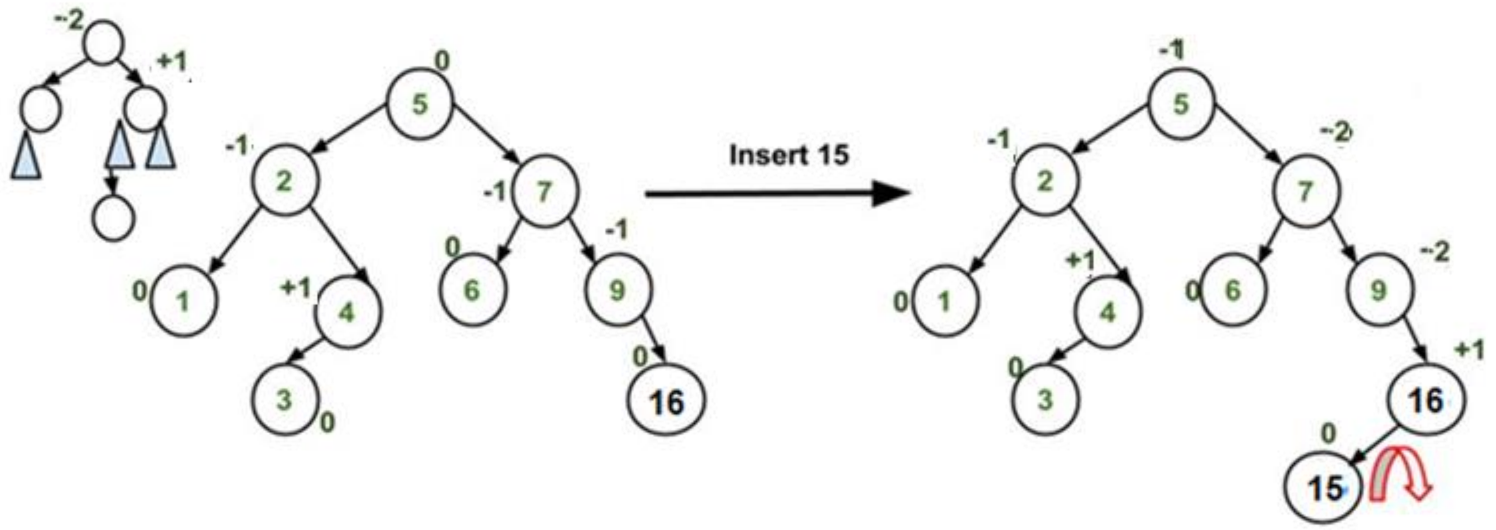


Illustration of Insertion at AVL Tree 5

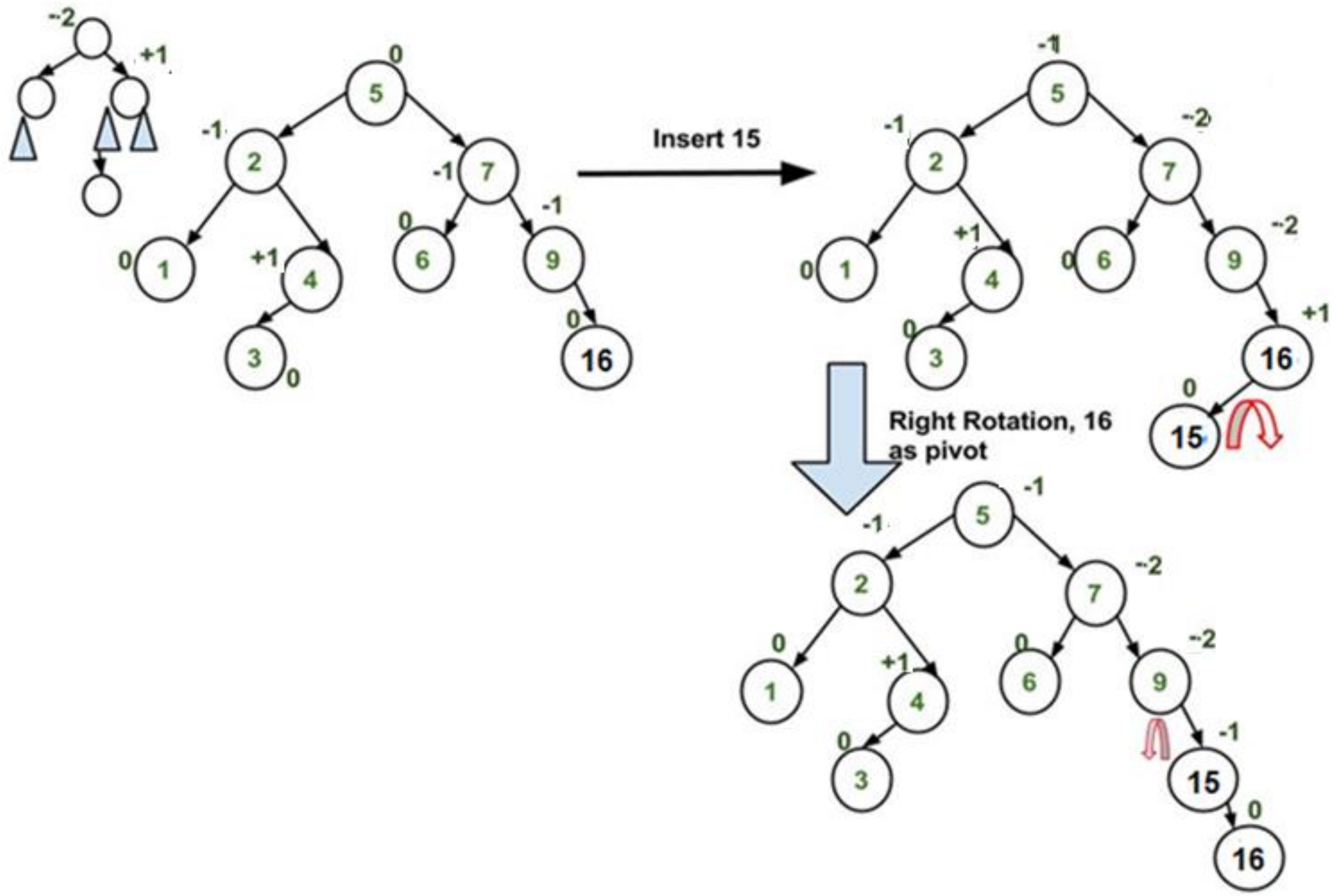
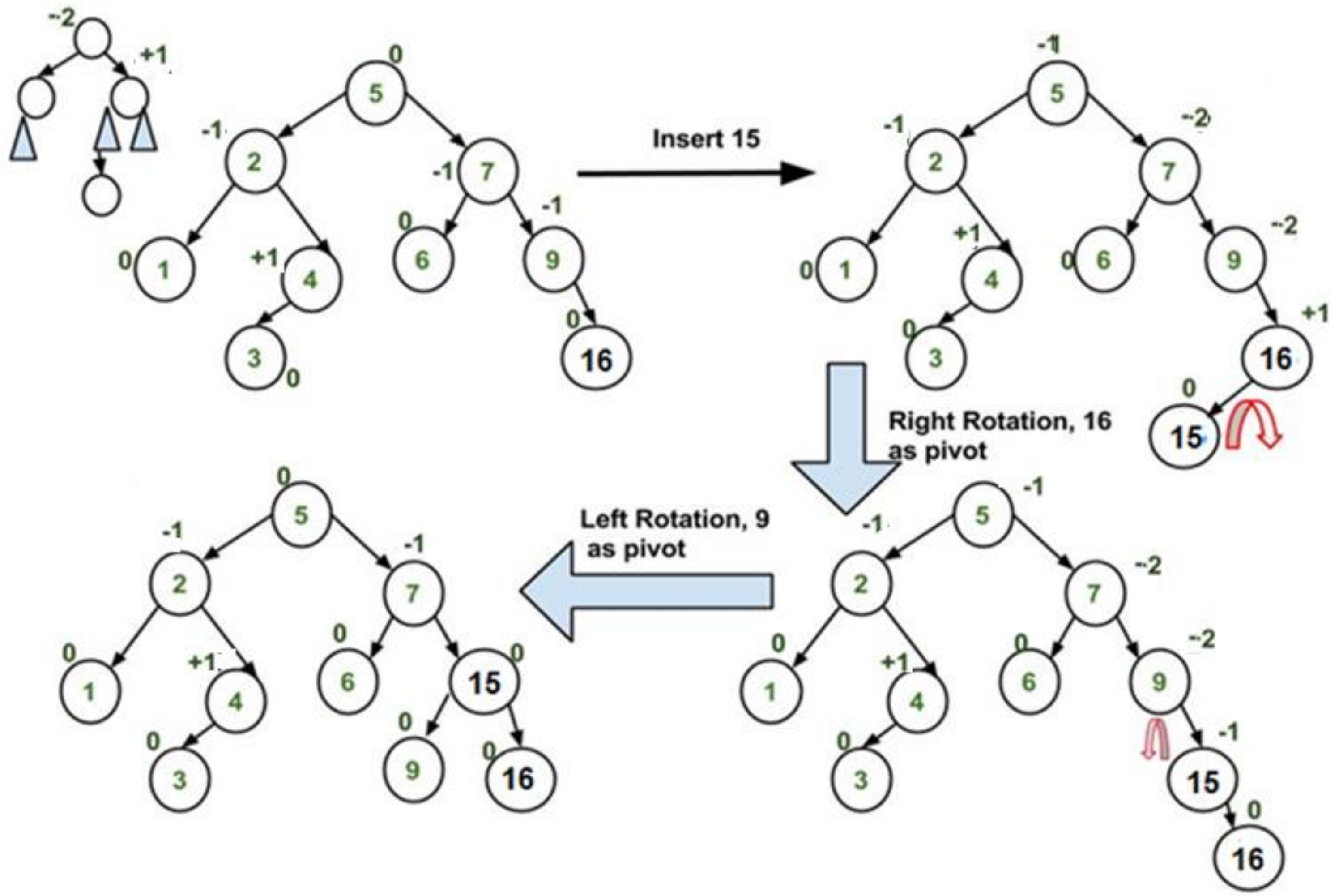


Illustration of Insertion at AVL Tree 5



Advantages of AVL Tree 7

Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed $\log_2(n)$, where, N is the total number of nodes in the tree.

Disadvantages of AVL Tree 8

Disadvantages of AVL Tree:

1. It is somewhat difficult to implement.
2. It has high constant factors for some of the operations.
3. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
4. Take more processing for balancing.
5. Less used compared to Red-Black trees (Because rotation is done when the difference is greater than double, which reduces rotation and increases height.).

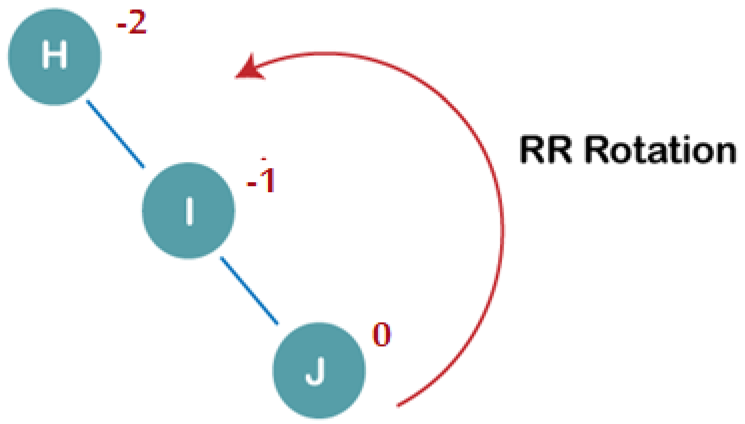
انتهت محاضرة الأسبوع 10

H,I,J,B,A,E,C,F,D,G,K,L

Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

Insert H, I, J

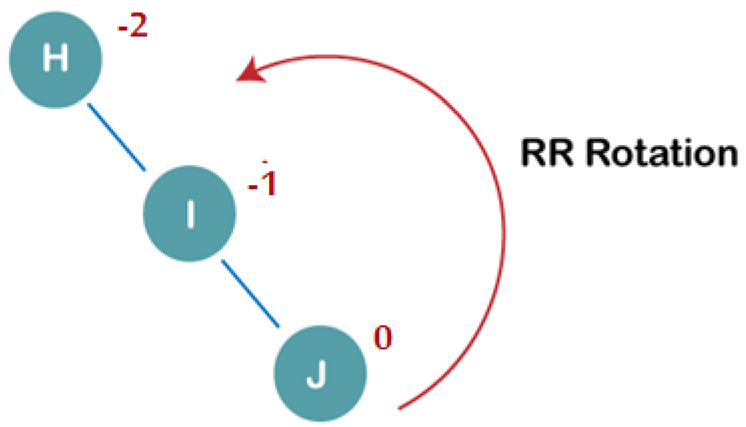


H,I,J,B,A,E,C,F,D,G,K,L

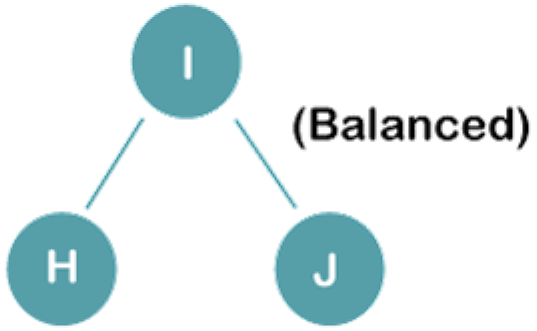
Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

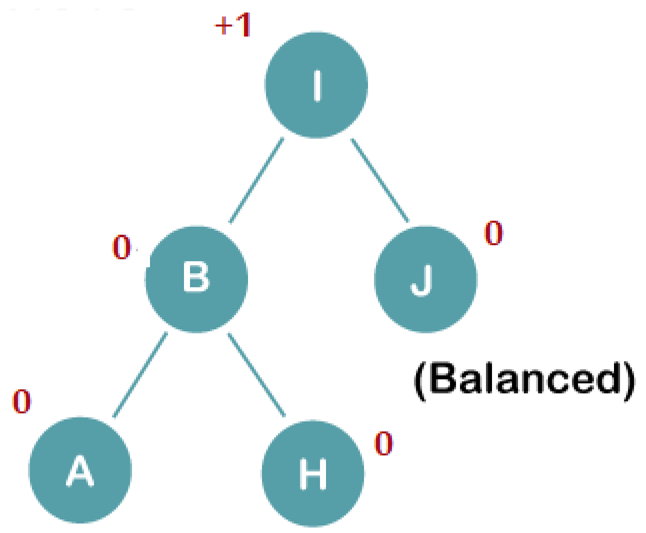
Insert H, I, J



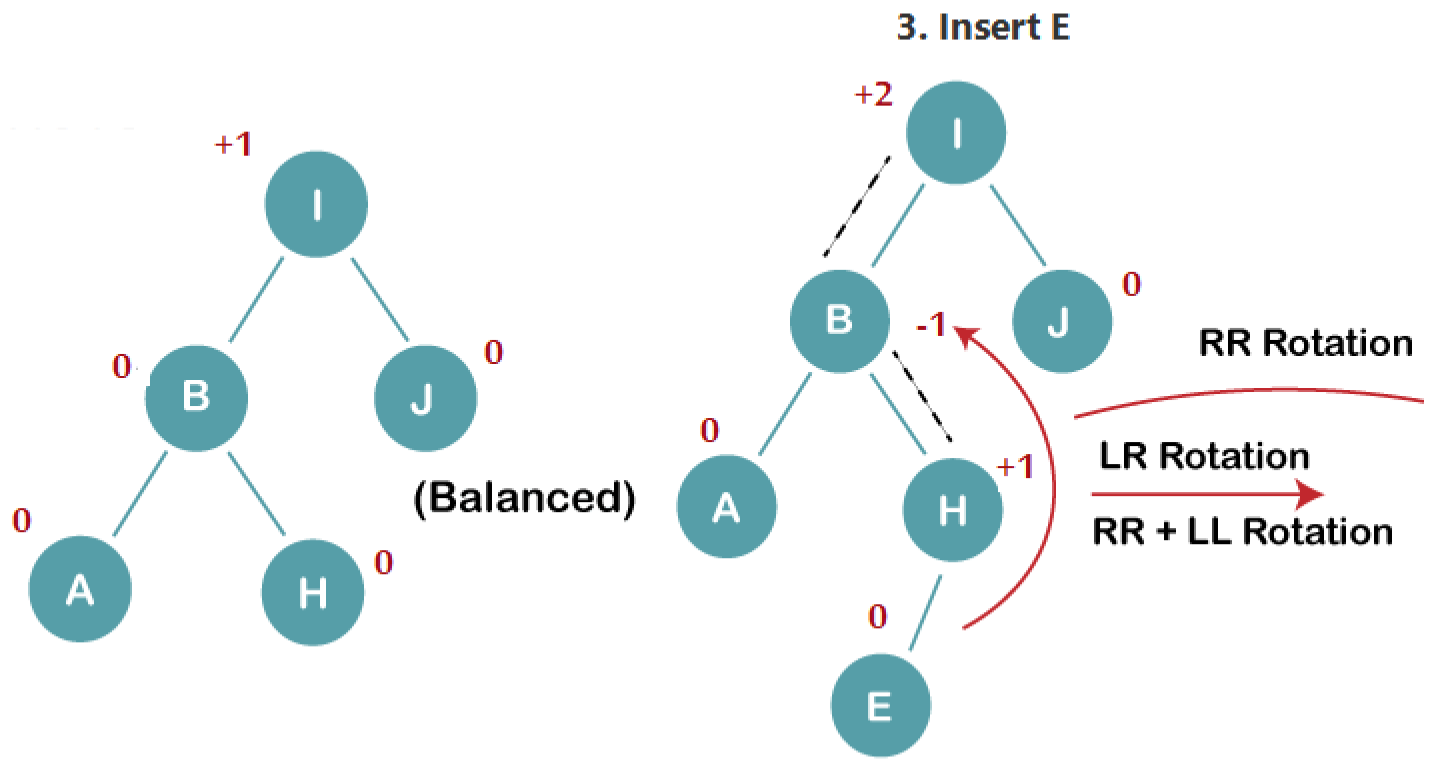
The resultant balance tree is:



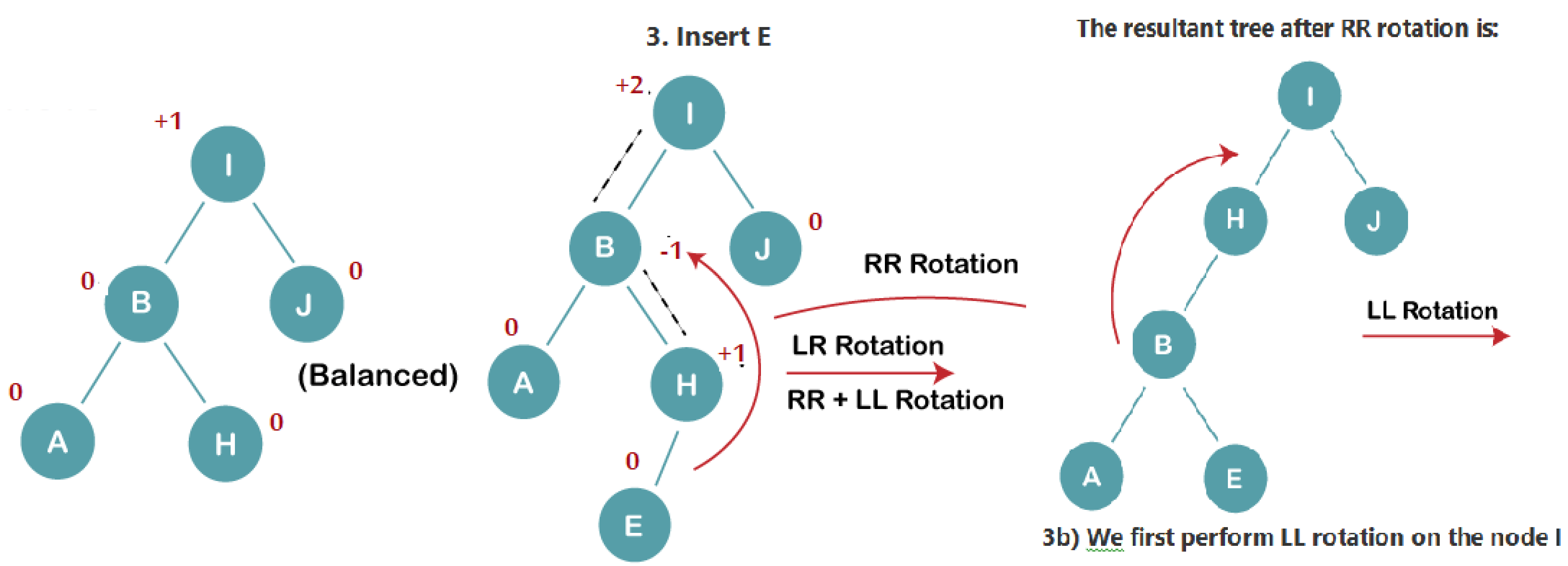
H,I,J,B,A,E,C,F,D,G,K,L



H,I,J,B,A,E,C,F,D,G,K,L

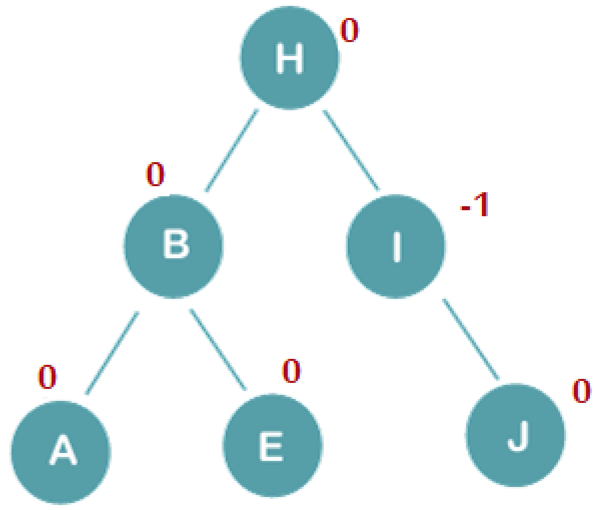


H,I,J,B,A,E,C,F,D,G,K,L



H,I,J,B,A,E,C,F,D,G,K,L

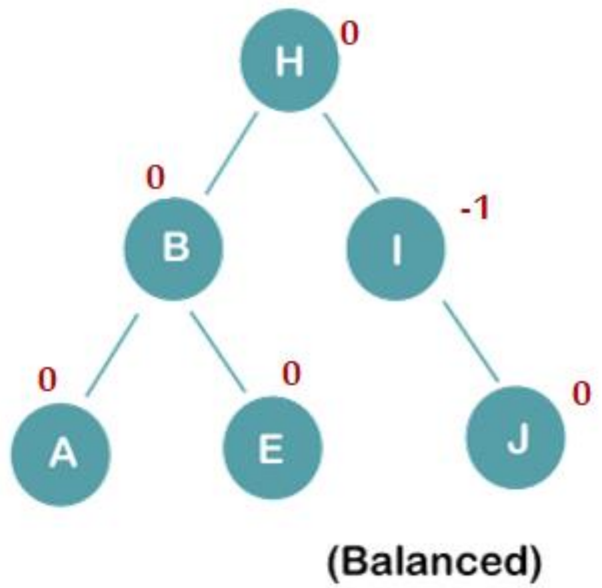
The resultant balanced tree after LL rotation is:



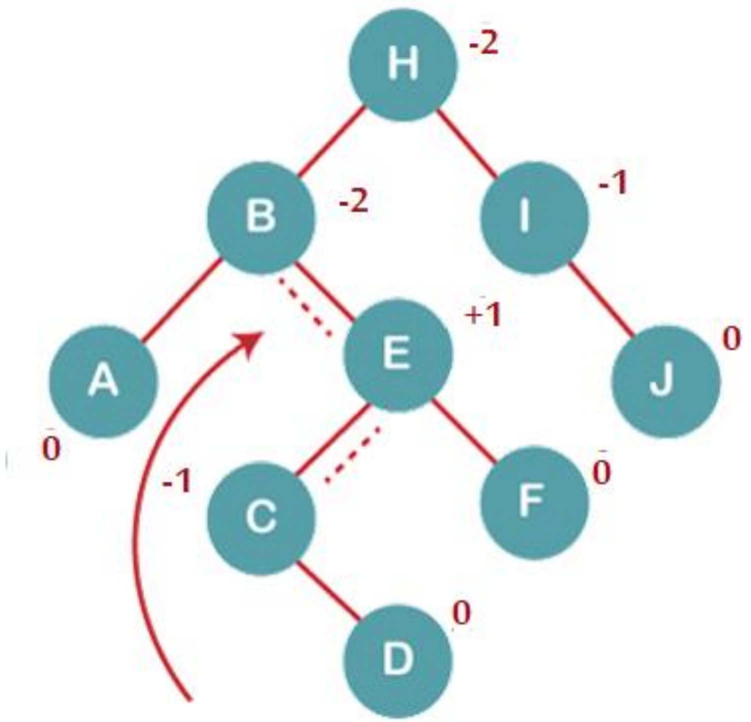
(Balanced)

H,I,J,B,A,E,C,F,D,G,K,L

The resultant balanced tree after LL rotation is:

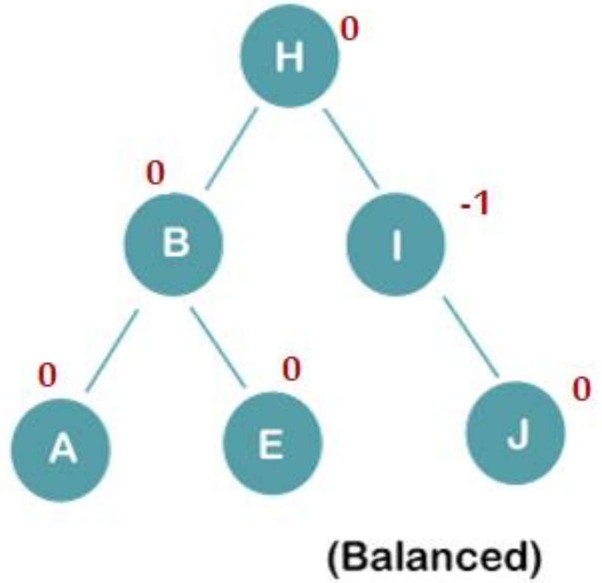


4. Insert C, F, D

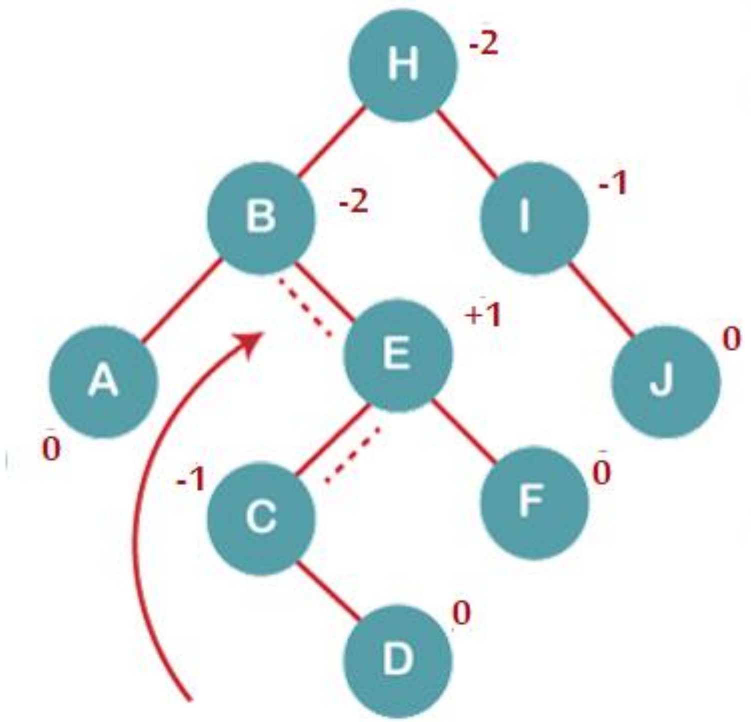


H,I,J,B,A,E,C,F,D,G,K,L

The resultant balanced tree after LL rotation is:

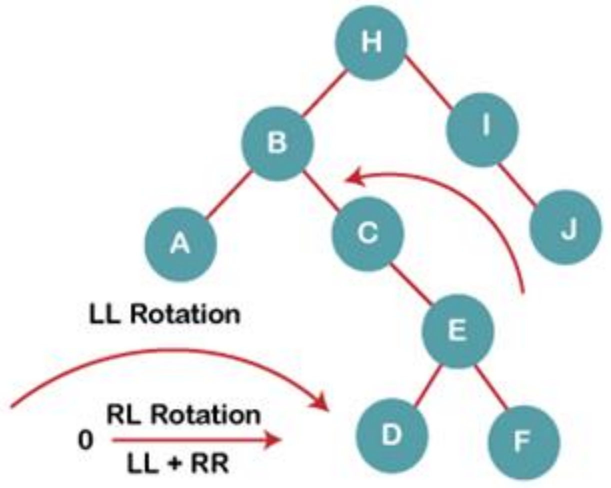


4. Insert C, F, D



4a) We first perform LL rotation on node E

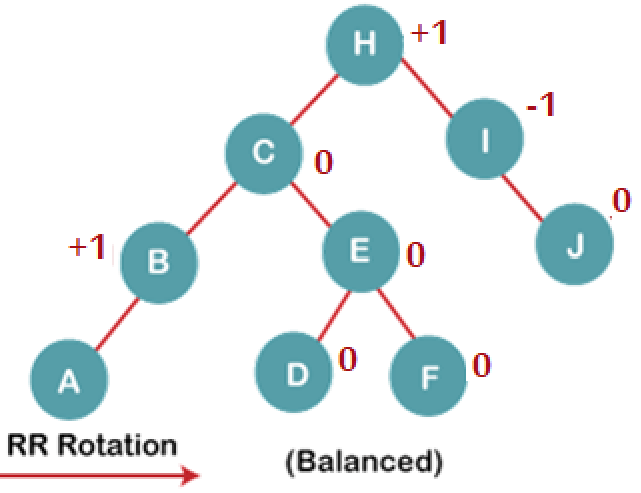
The resultant tree after LL rotation is:



H,I,J,B,A,E,C,F,D,G,K,L

4b) We then perform RR rotation on node B

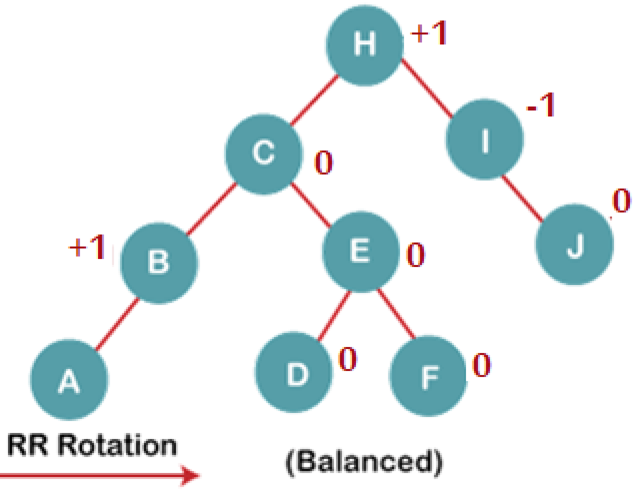
The resultant balanced tree after RR rotation is:



H,I,J,B,A,E,C,F,D,G,K,L

4b) We then perform RR rotation on node B

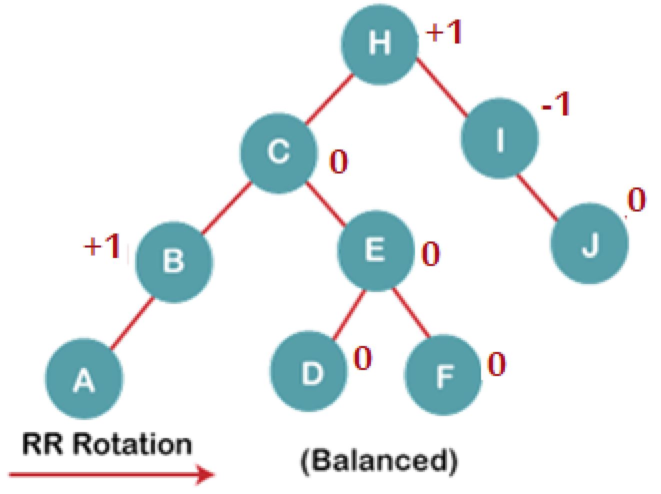
The resultant balanced tree after RR rotation is:



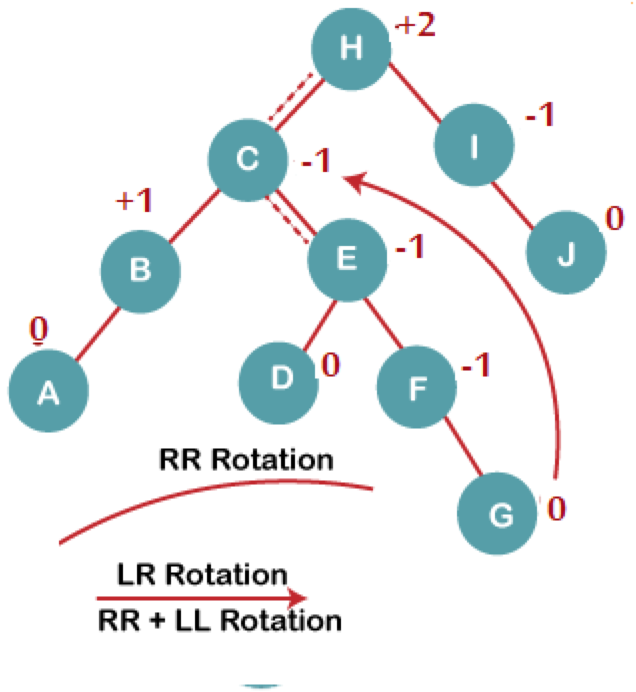
H,I,J,B,A,E,C,F,D,G,K,L

4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



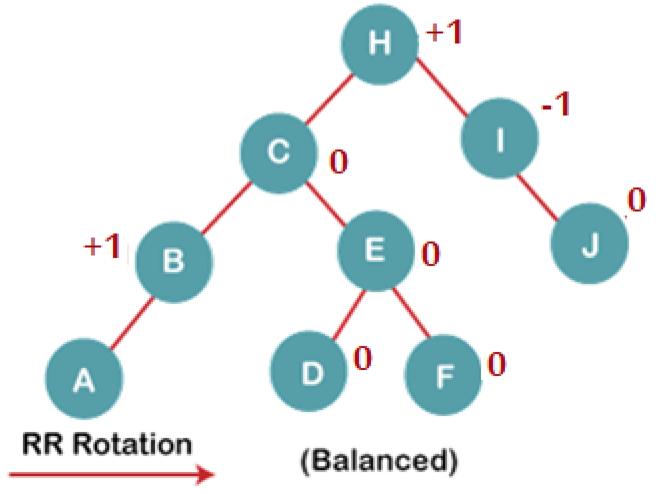
5. Insert G



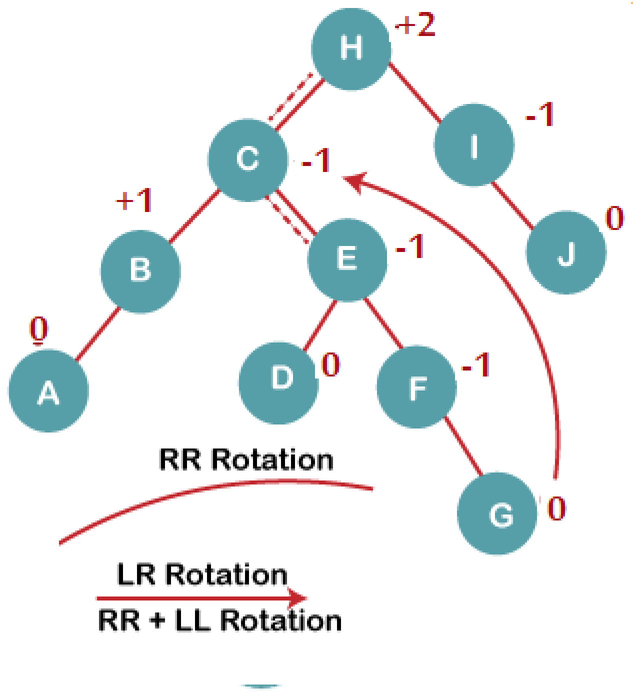
H,I,J,B,A,E,C,F,D,G,K,L

4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:

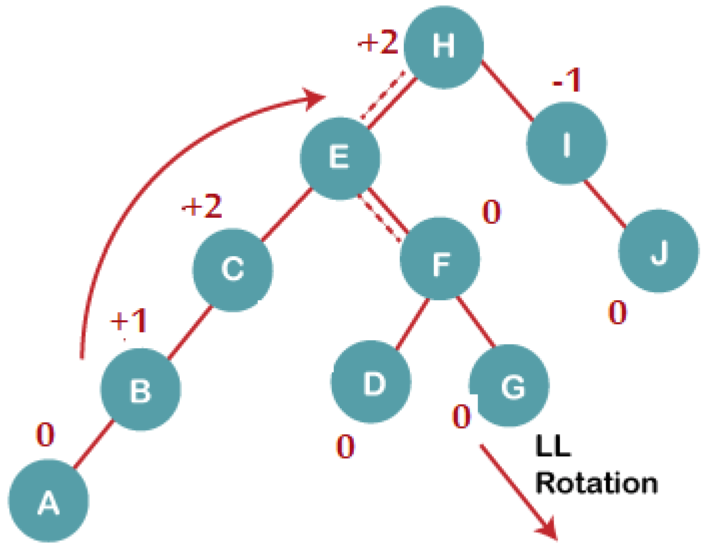


5. Insert G



5 a) We first perform RR rotation on node C

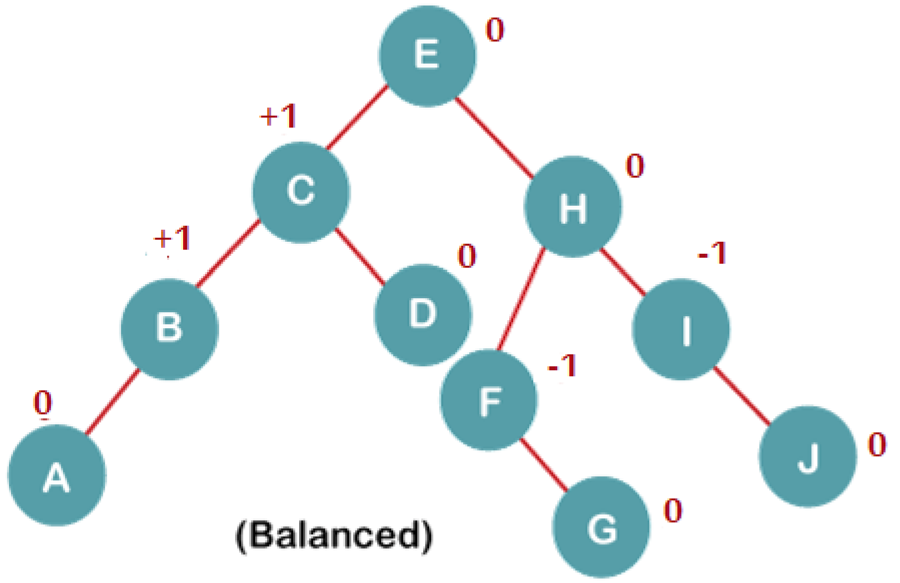
The resultant tree after RR rotation is:



H,I,J,B,A,E,C,F,D,G,K,L

5 b) We then perform LL rotation on node H

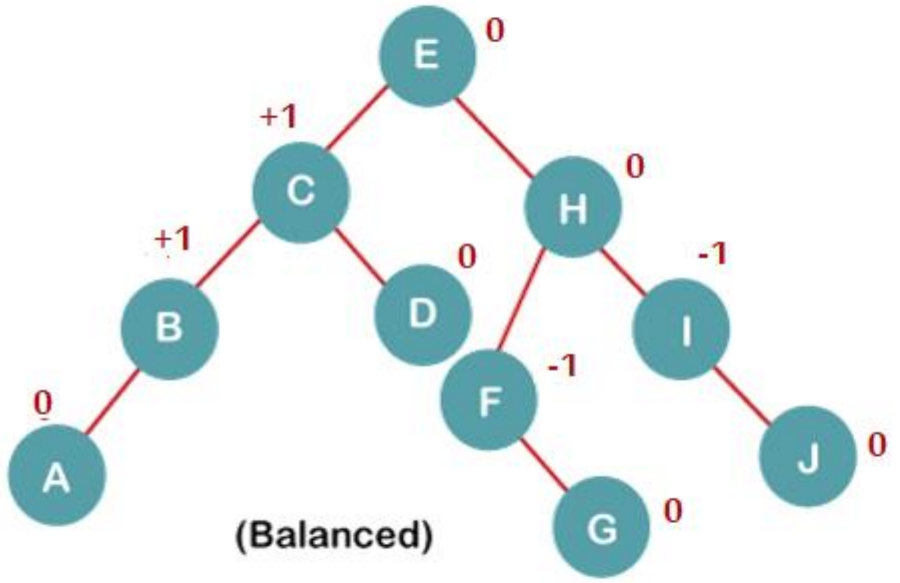
The resultant balanced tree after LL rotation is:



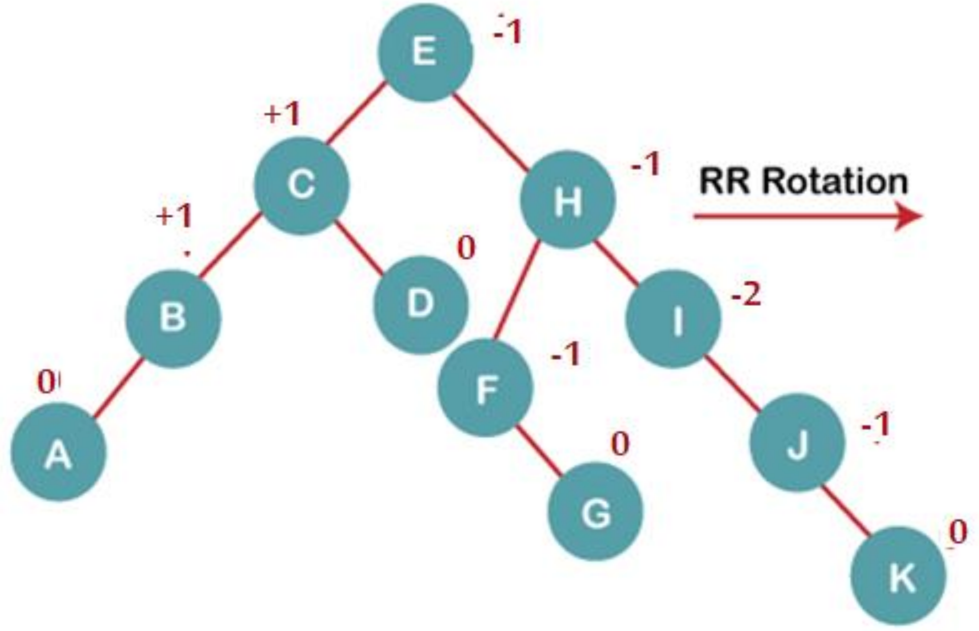
H,I,J,B,A,E,C,F,D,G,K,L

5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

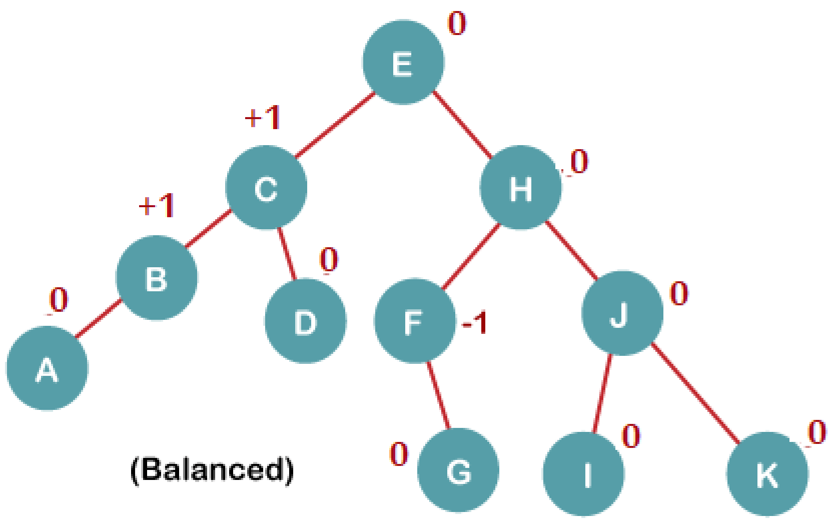


6. Insert K



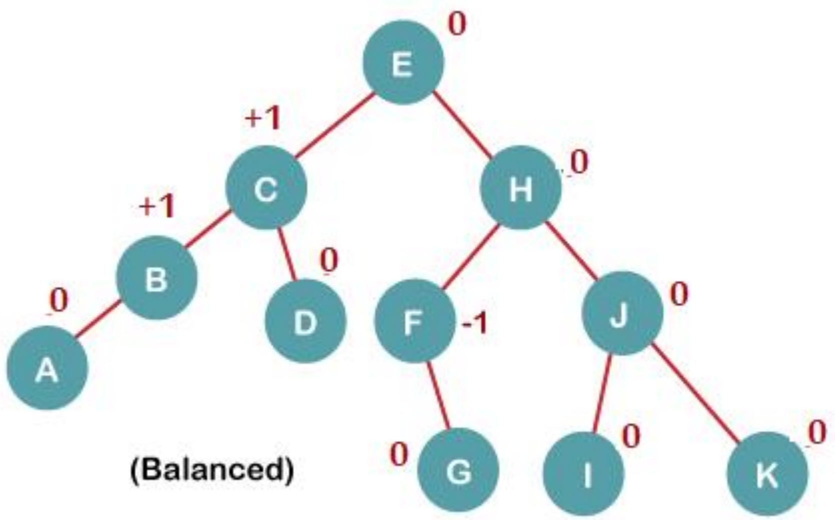
H,I,J,B,A,E,C,F,D,G,K,L

The resultant balanced tree after RR rotation is:



H,I,J,B,A,E,C,F,D,G,K,L

The resultant balanced tree after RR rotation is:



7. Insert L

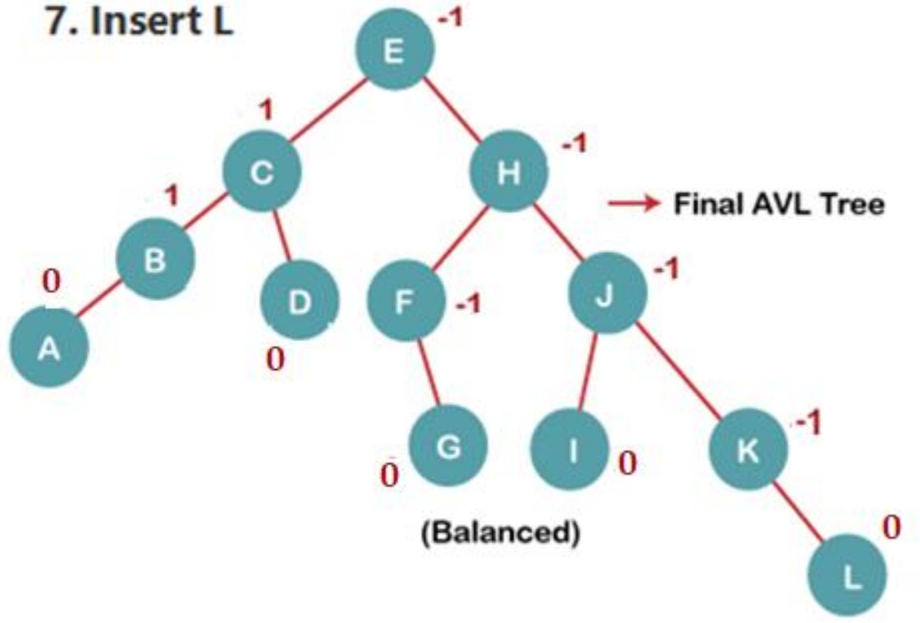


Illustration of Insertion at AVL Tree 5

```
// https://www.geeksforgeeks.org/dsa/deletion-in-an-avl-tree/C++ program
#include<iostream>
using namespace std;

// An AVL tree node
class Node {   public:    int key;     Node *left; Node *right;     int height; };

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(Node *N) {   if (N == NULL)          return 0;    return N->height; }

// A utility function to get maximum of two integers
int max(int a, int b) {    return (a > b)? a : b; }

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
Node* newNode(int key) {   Node* node = new Node();    node->key = key;
    node->left = NULL;    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node); }
```

Illustration of Insertion at AVL Tree 5

```
// A utility function to right rotate subtree rooted with y See the diagram given above.
    Node *rightRotate(Node *y) {   Node *x = y->left;     Node *T2 = x->right;

    x->right = y;     y->left = T2; // Perform rotation

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
     return x; // Return new root
}

     // A utility function to left rotate subtree rooted with x
     // See the diagram given above.
    Node *leftRotate(Node *x) { Node *y = x->right;   Node *T2 = y->left;

    y->left = x;    x->right = T2; // Perform rotation

    // Update heights
    x->height = max(height(x->left),height(x->right)) + 1;
    y->height = max(height(y->left),height(y->right)) + 1;
    return y; // Return new root
}
```

Illustration of Insertion at AVL Tree 5

```
// Get Balance factor of node N
int getBalance(Node *N) { if (N == NULL) return 0;
       return height(N->left) - height(N->right); }

Node* insert(Node* node, int key) {    /* 1. Perform the normal BST rotation */
    if (node == NULL)       return(newNode(key));
    if (key < node->key) node->left = insert(node->left, key);
    else if (key > node->key)node->right = insert(node->right, key);
    else    return node; // Equal keys not allowed

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left), height(node->right));

    /* 3. Get the balance factor of this ancestor node to check whether
        this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases Left Left Case
    if (balance > 1 && key < node->left->key)return rightRotate(node);

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

Illustration of Insertion at AVL Tree 5

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

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

    /* return the (unchanged) node pointer */
    return node; }

/* Given a non-empty binary search tree, return the node with minimum key value
found in that tree. Note that the entire tree does not need to be searched. */
Node * minValueNode(Node* node) {     Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL) current = current->left;     return current;
}
```

Illustration of Insertion at AVL Tree 5

```
// Recursive function to delete a node  with given key from subtree with
// given root. It returns root of the modified subtree.
Node* deleteNode(Node* root, int key) {   // STEP 1: PERFORM STANDARD BST DELETE
     if (root == NULL)        return root;

     // If the key to be del is smaller  than the root's key, then it lies in left subtree
     if ( key < root->key ) root->left = deleteNode(root->left, key);

     // If the key to be del is greater than the root's key, then it lies in right subtree
     else if( key > root->key ) root->right = deleteNode(root->right, key);

     // if key is same as root's key, then This is the node to be deleted
     else     {        // node with only one child or no child
       if( (root->left == NULL) || (root->right == NULL) )
         {  Node *temp = root->left ? root->left :root->right;

         if (temp == NULL) {  temp = root; root = NULL; } // No child case
              else // One child case
              *root = *temp; // Copy the contents of the non-empty child
              free(temp);    }
```

Illustration of Insertion at AVL Tree 5

```
else  {   // node with two children: Get the inorder
           Node* temp = minValueNode(root->right); // successor (smallest in the right subtree)
            root->key = temp->key; // Copy the inorder successor's data to this node
            root->right = deleteNode(root->right, temp->key); // Delete the inorder successor
            }
       }
            if (root == NULL)     return root; // If the tree had only one node then return
       // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
       root->height = 1 + max(height(root->left), height(root->right));
       // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether this
       // node became unbalanced)
       int balance = getBalance(root);

       // If this node becomes unbalanced, then there are 4 cases     // Left Left Case
       if (balance > 1 && getBalance(root->left) >= 0)          return rightRotate(root);
```

Illustration of Insertion at AVL Tree 5

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

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

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

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(Node *root) {   if(root != NULL)
    {cout << root->key << " "; preOrder(root->left);preOrder(root->right);   }
}
```

Illustration of Insertion at AVL Tree 5

```
// Driver Code
int main()
{ Node *root = NULL;
    /* Constructing tree given in the above figure */
    root = insert(root, 9);     root = insert(root, 5);     root = insert(root, 10);
    root = insert(root, 0);     root = insert(root, 6);     root = insert(root, 11);
    root = insert(root, -1);    root = insert(root, 1);     root = insert(root, 2);

    /* The constructed AVL Tree would be
         9
        / \
       1  10
      / \   \
     0   5  11
    /   / \
  -1   2   6
    */
```

Illustration of Insertion at AVL Tree 5

```
cout << "Preorder traversal of the constructed AVL tree is \n";
    preOrder(root);

    root = deleteNode(root, 10);
     /* The AVL Tree after deletion of 10
          1
         / \
        0   9
       /    / \
    -1     5   11
          / \
         2   6
    */

    cout << "\nPreorder traversal after deletion of 10 \n";
    preOrder(root);
   system("pause");  return 0;
}
```