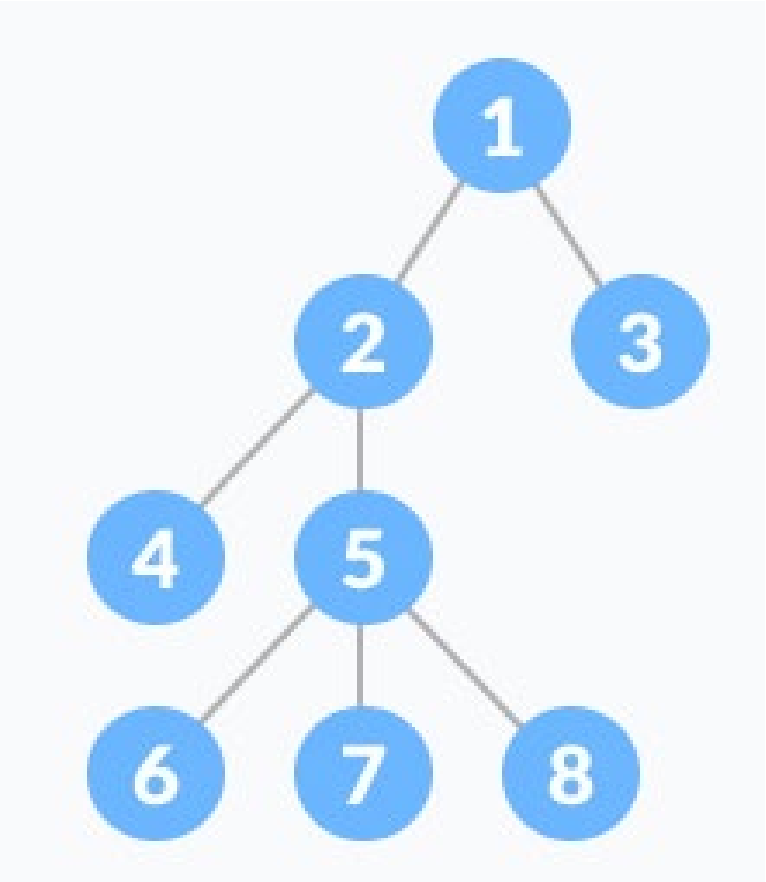
Unit 4: Trees

A.M. Joshi

Tree:

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

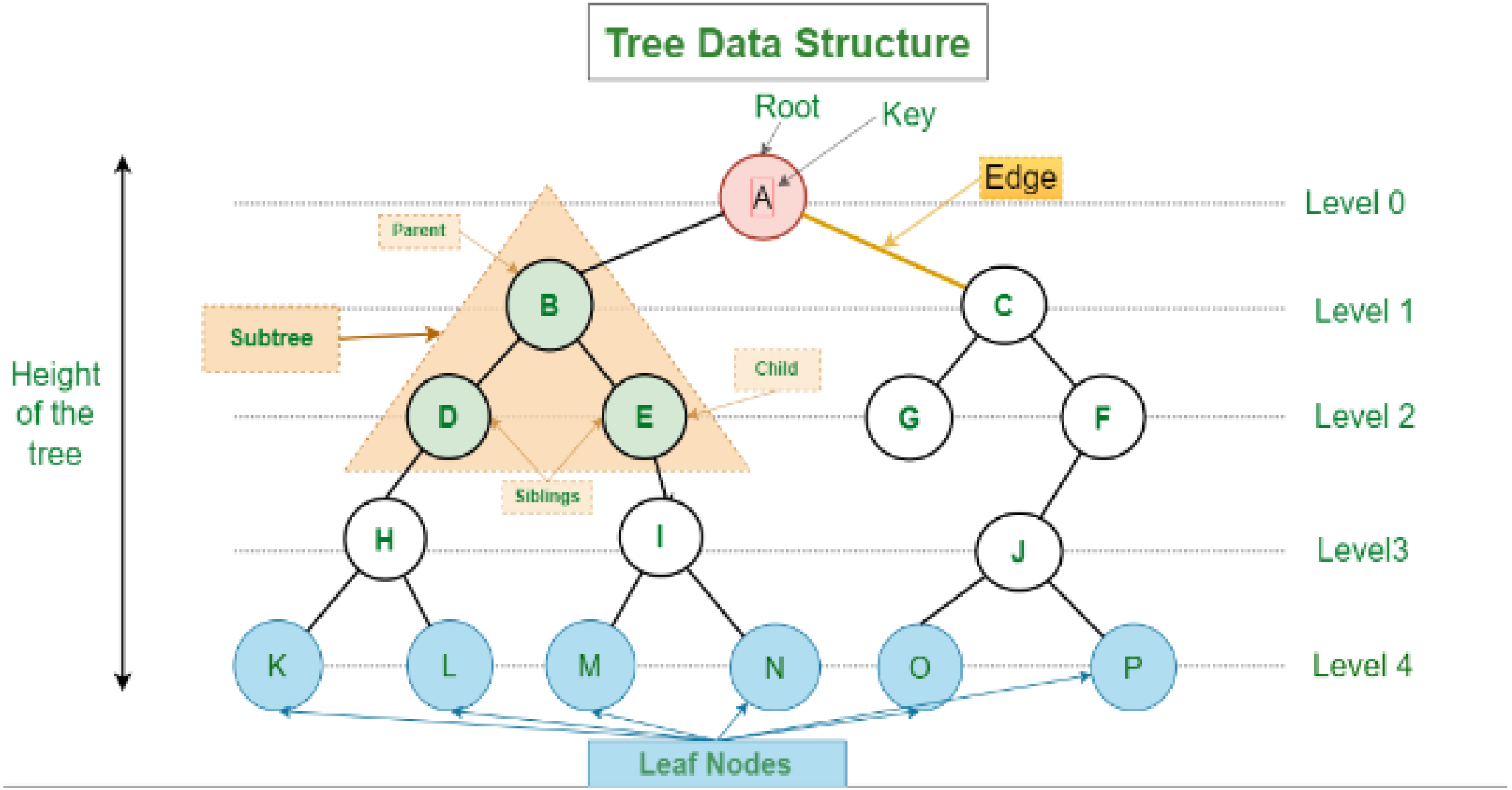


Why Tree Data Structure?

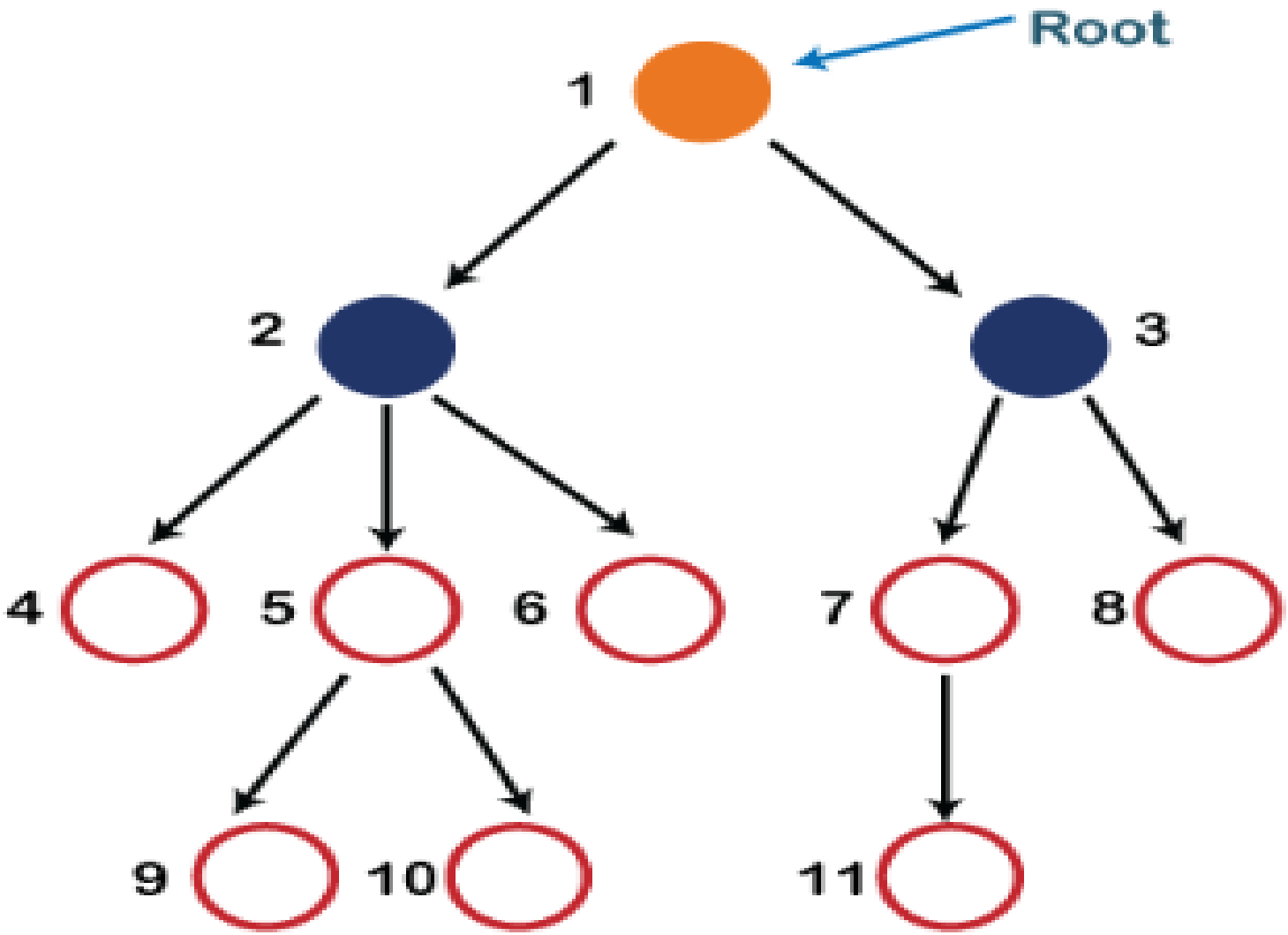
- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
- In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size.

- But, it is not acceptable in today's computational world.

- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.



Tree Terminologies


Root
1
2
3
4
5
6
7
8
9
10
11

Tree Terminologies

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is the root node of the tree. If a node is directly linked to some other node, it would be called a parent-child relationship.

- **Child node:** If the node is a descendant of any node, then the node is known as a child node.

- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.

- **Sibling:** The nodes that have the same parent are known as siblings.

Cont...

- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

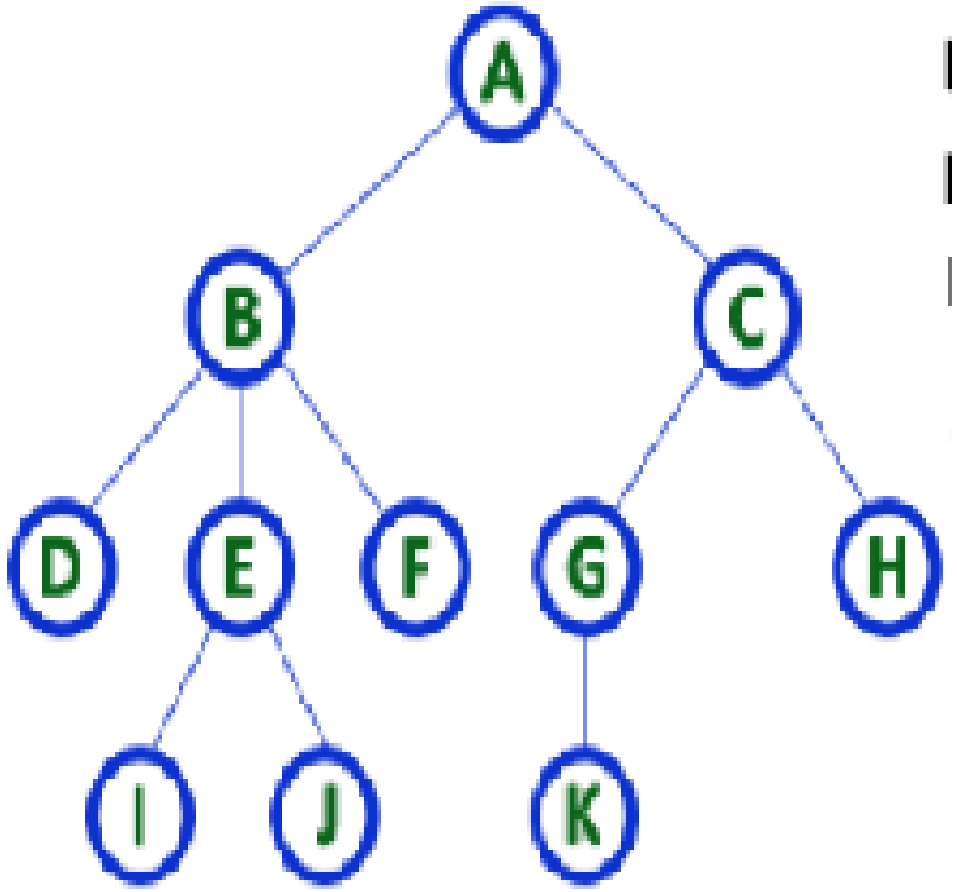
- **Internal nodes:-** A node has atleast one child node known as an internal

- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

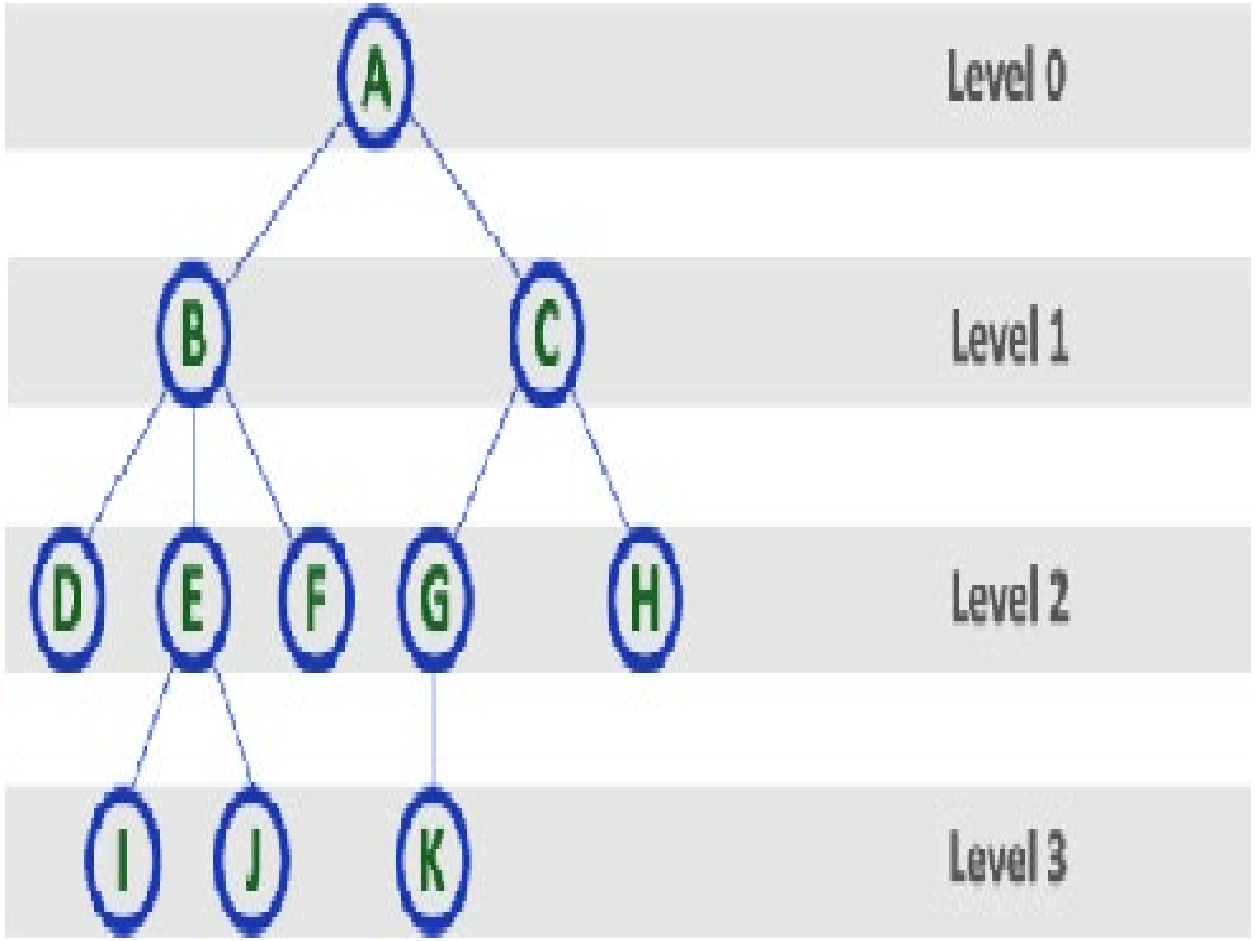
Cont...

- **Degree:-** In a tree data structure, the total number of children of a node is called as DEGREE of that Node.
- In simple words, the Degree of a node is total number of children it has.
- The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'.
- Here Degree of B is 3
- Degree of A is 2 and Degree of F is 0.
- In any tree Degree of node is, Total no. of children it has.



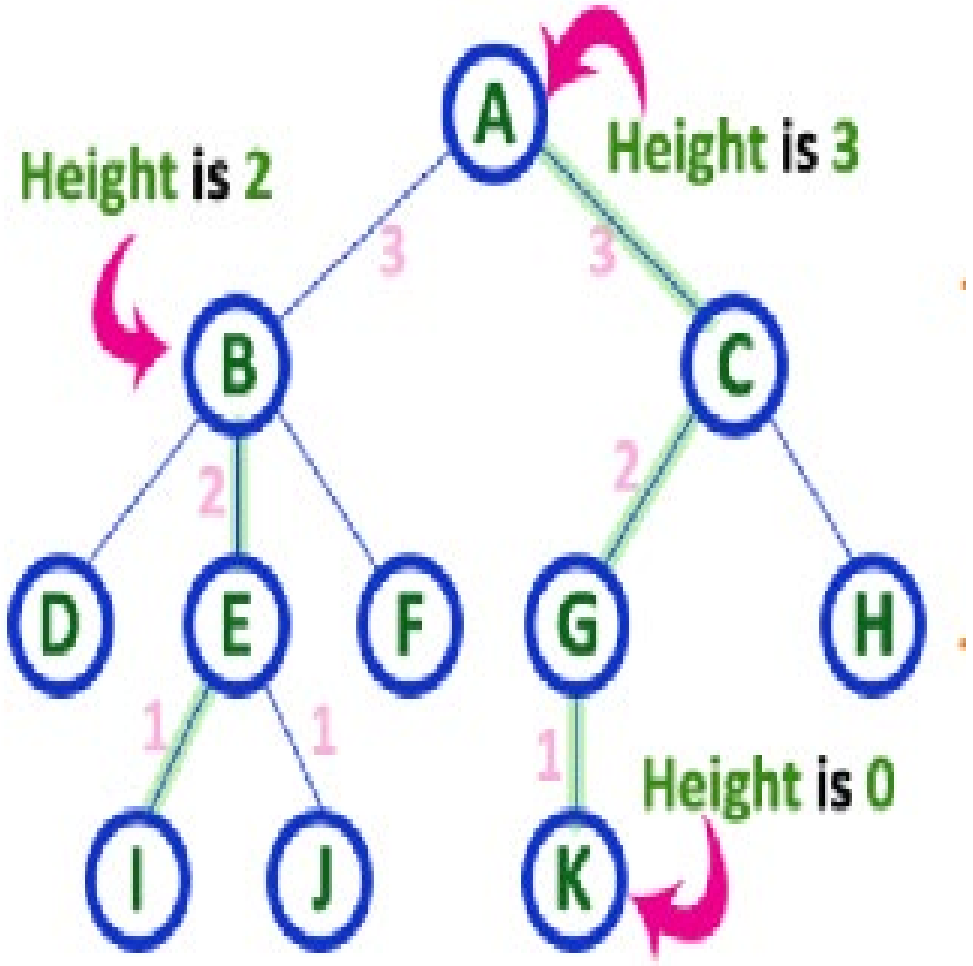
Cont...

- **Level:** In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...
- In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



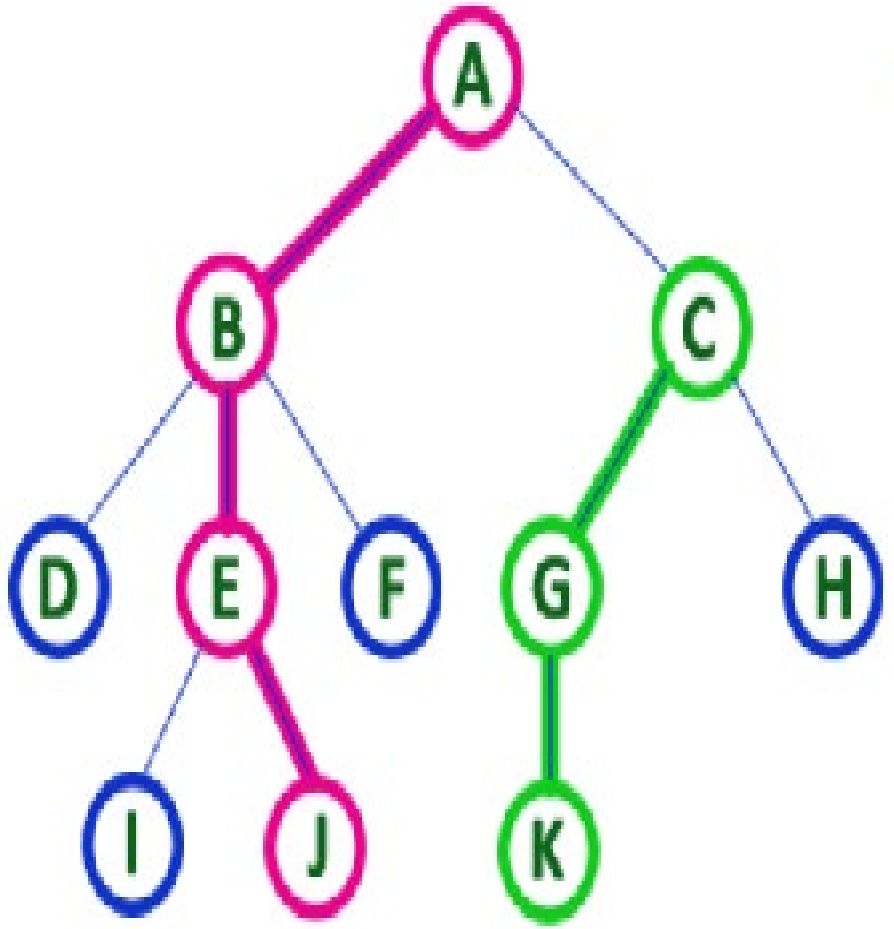
Cont...

- **Height:** In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node.
- In a tree, height of the root node is said to be height of the tree.
- In a tree, height of all leaf nodes is '0'.
- Here Height of Tree is 3
- In any tree 'Height of node' is total no. of edges from leaf to that node in longest path.
- In any tree 'Height of tree' is the height of the root node



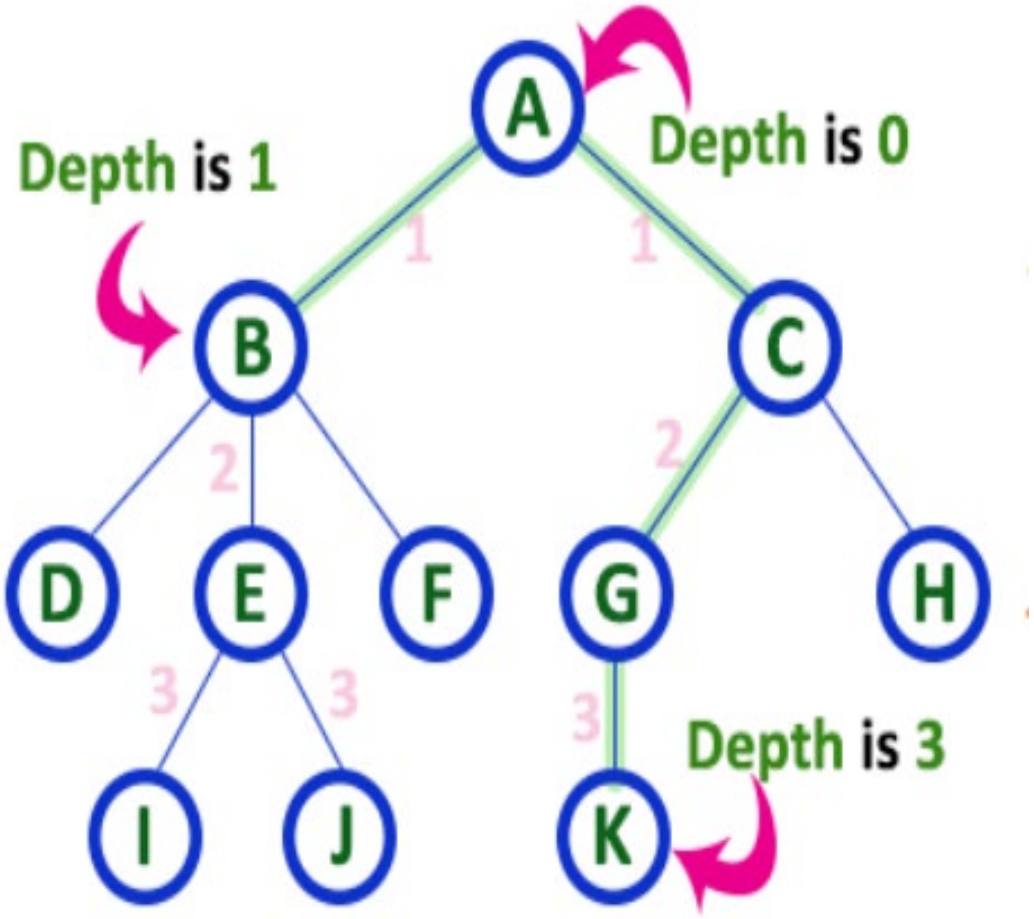
Cont...

- **Path**
- In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes.
- Length of a Path is total number of nodes in that path.
- In below example the path A - B - E - J has length 4.
- In any tree ‘path’ is a sequence of nodes and edges between 2 nodes.
- Here path between A and J is : A-B-E-J
- Here path between C and K is : C-G-K



Cont...

- **Depth:-**
- In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node.
- In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree.
- In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.
- Here Depth of tree is 3.
- In any tree ‘Depth of node’is total no. of edges from root to that node.
- In any tree ‘Depth of tree’ is total no. of edges from root to leaf in a longest path.



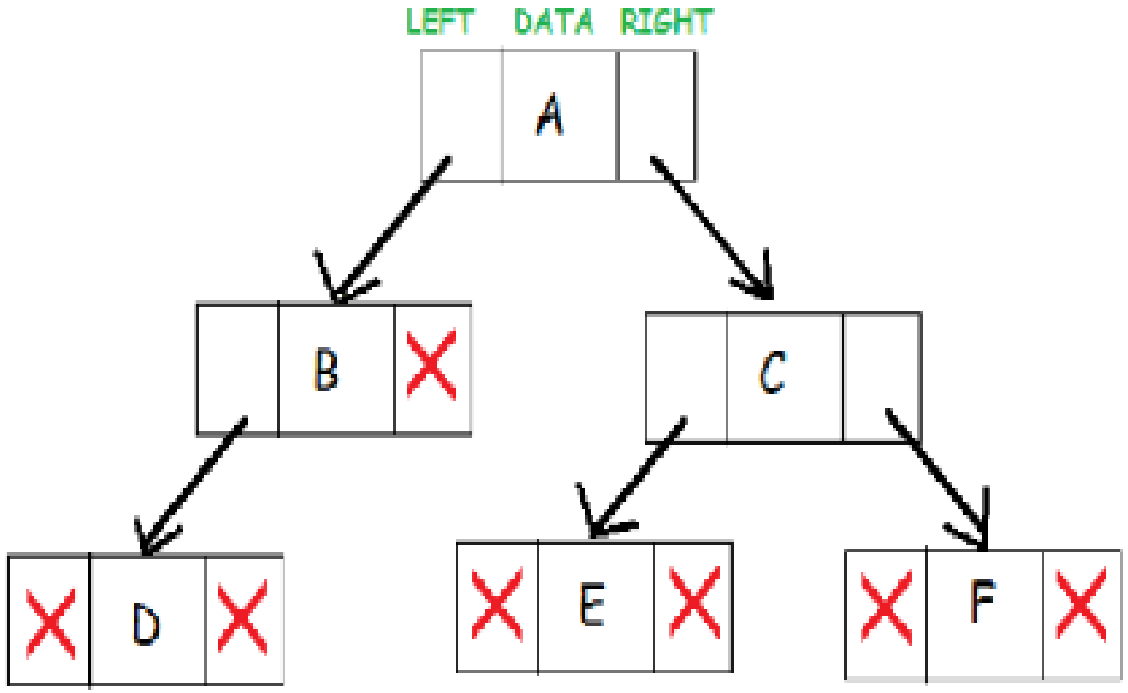
Binary Tree

- A binary tree is a tree data structure in which each node can have at most two children, which are referred to as the left child and the right child.
- The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves.
- A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.
- Binary trees have many applications in computer science, including data storage and retrieval, expression evaluation, network routing, and game AI.
- They can also be used to implement various algorithms such as searching, sorting, and graph algorithms.

Representation of Binary Tree:

- Each node in the tree contains the following:

- Data
- Pointer to the left child
- Pointer to the right child



In C, we can represent a tree node using structures.

```
struct node
 {
    int data;
    struct node* left;
    struct node* right;
};
```

Basic Operations On Binary Tree:

- Inserting an element.
- Removing an element.
- Searching for an element.
- Deletion for an element.
- Traversing an element. T
- here are four (mainly three) types of traversals in a binary tree which will be discussed ahead.

Binary Tree Traversals:

- Tree Traversal algorithms can be classified broadly into two categories:

1. Depth-First Search (DFS) Algorithms
2. Breadth-First Search (BFS) Algorithms

Tree Traversal using Depth-First Search (DFS) algorithm can be further classified into three categories:

- Preorder Traversal (current-left-right): Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.

- Inorder Traversal (left-current-right): Visit the current node after visiting all nodes inside the left subtree but before visiting any node within the right subtree. Here, the traversal is left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.

- Postorder Traversal (left-right-current): Visit the current node after visiting all the nodes of the left and right subtrees. Here, the traversal is left child – right child – root. It means that the left child has traversed first then the right child and finally its root node.

Tree Traversal using Breadth-First Search (BFS) algorithm can be further classified into one category:

- Level Order Traversal: Visit nodes level-by-level and left-to-right fashion at the same level. Here, the traversal is level-wise.
- It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

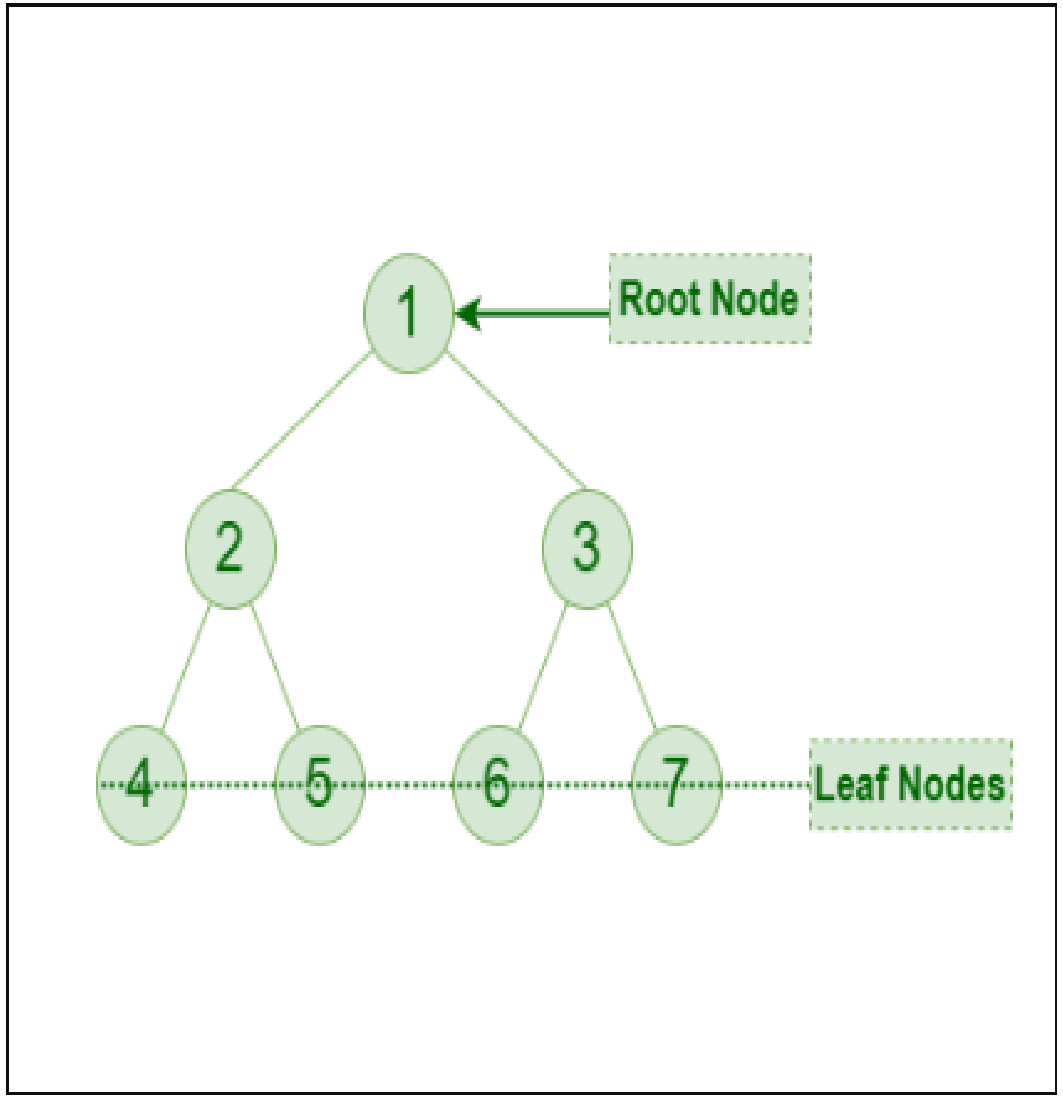
Example for all four traversal

Pre-order Traversal of the above tree:
(Root-Left-Right)
1-2-4-5-3-6-7

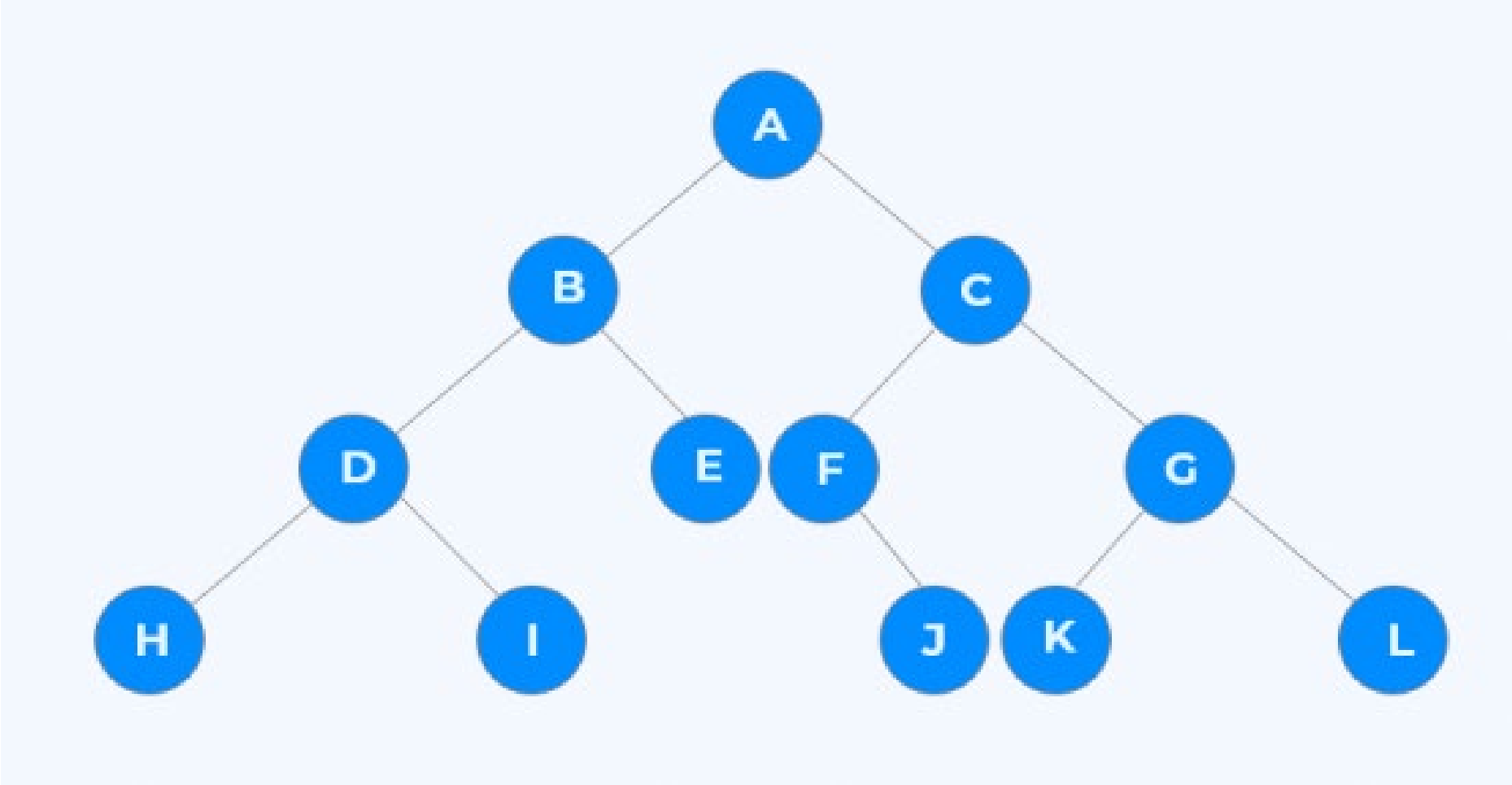
In-order Traversal of the above tree:
(Left-Root-Right)
4-2-5-1-6-3-7

Post-order Traversal of the above tree:
(Left-Right-Root)
4-5-2-6-7-3-1

Level-order Traversal of the above tree:
1-2-3-4-5-6-7



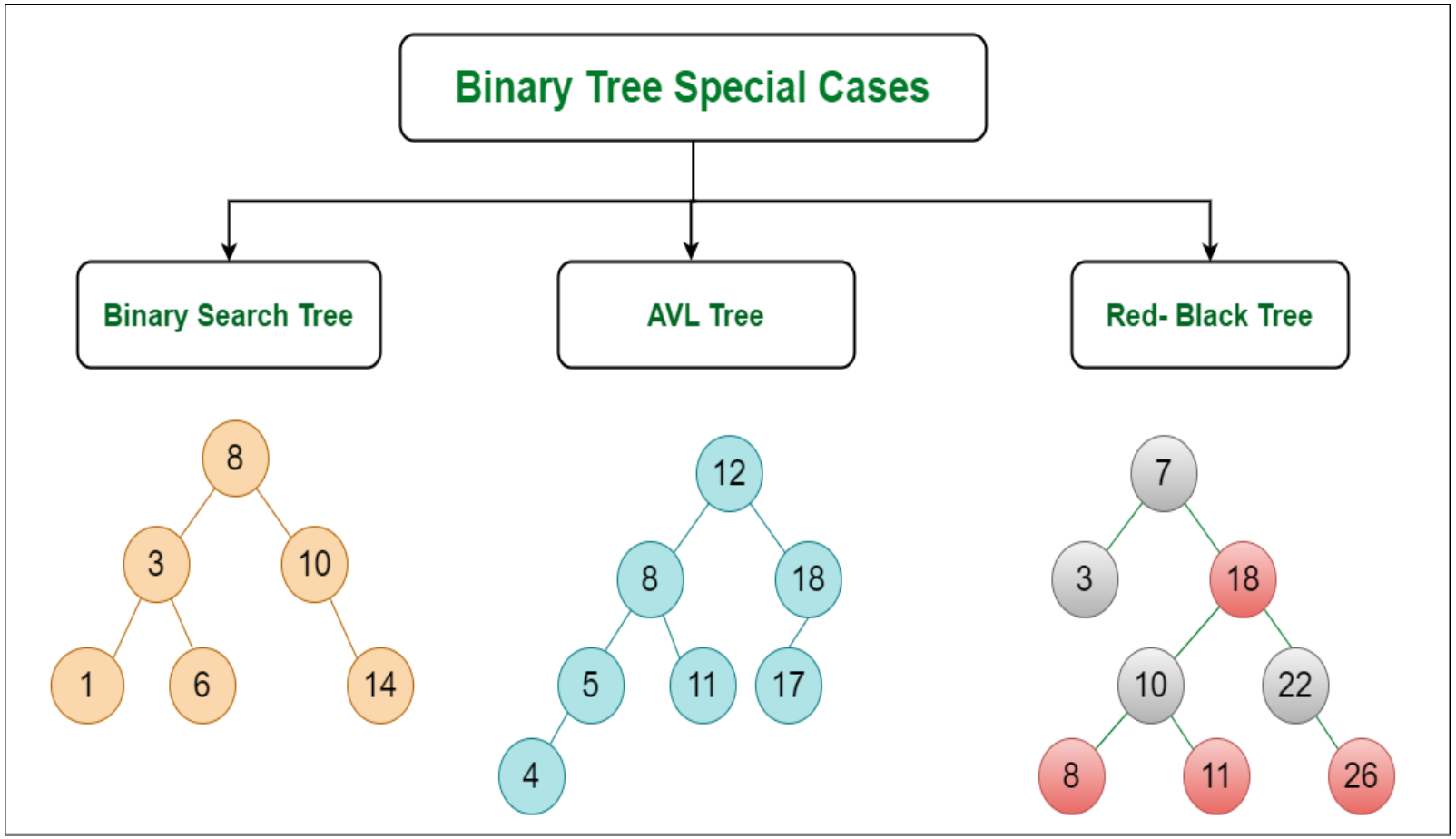
Solve



Some Special Types of Trees:

On the basis of node values, the Binary Tree can be classified into the following special types:

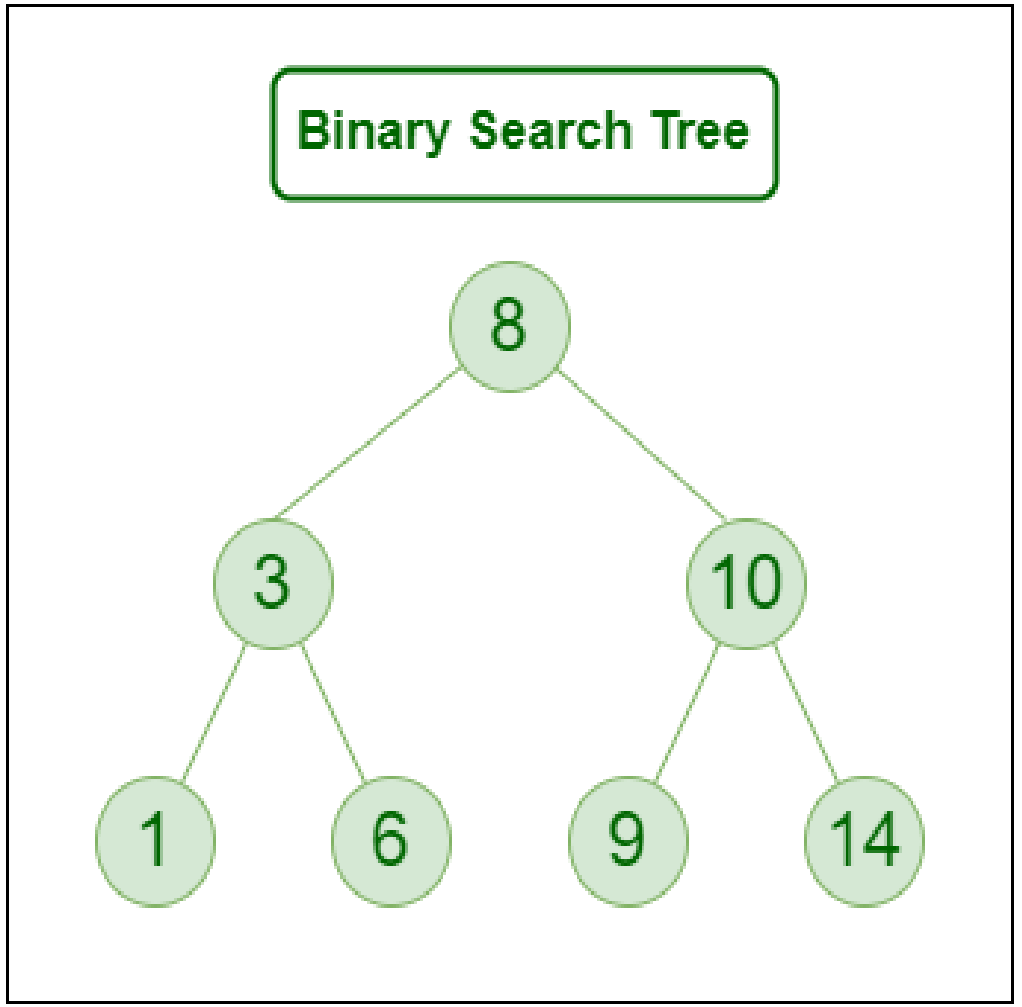
1. Binary Search Tree
2. AVL Tree
3. Red Black Tree
4. B Tree
5. B+ Tree
6. Segment Tree



1. Binary Search Tree

- Binary Search Tree is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Tree Implementation:

- Declaration of a Binary Tree :
- To declare the nodes of the binary tree, we will create a structure containing one data element and two pointers of the name left and right.

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

Create a New Node of the Binary Tree

- To create a new node of the binary tree, we will create a new function that will take the value and return the pointer to the new node created with that value.
- The new node which is being created will have the left and right pointers, pointing at null.

```
struct node* create(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->data = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}
```

Inserting the Left and/or the Right Child of a Node

- This function will take two inputs, first the value that needs to be inserted and second the node's pointer on which the left or the right child will be attached. To insert into the left side of the root node, we will simply call the create function with the value of the node as a parameter. So that a new node will be created, and as that function returns a pointer to the newly created node, we will assign that pointer to the left pointer of the root node that is being taken as input.
- Similarly, insertion on the right side will call the create function with the given value, and the returned pointer will be assigned to the write pointer of the root node.

```
// Insert on the left of the node.
struct node* insertLeft(struct node* root, int value) {
  root->left = create(value);
  return root->left;
}

// Insert on the right of the node.
struct node* insertRight(struct node* root, int value) {
  root->right = create(value);
  return root->right;
}
```

Traversal of Binary Tree

- Pre-Order :
- In this type of traversal, first, we will take the value of the current node(present node), then we will go to the left node, and after that, we will go to the right node.

```
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}
```

Cont...

- In-Order :
- In this type of traversal, first, we'll go to the left node, then we'll take the value of the current node, and then we will go to the right node.

```
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}
```

Cont...

- Post-Order :
- In this type of traversal, first, we'll go to the left node, then to the right node, and at last, we will take the current node's value.

```
void postorderTraversal(struct node* root)
{
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}
```

AVL

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

- 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.

- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Cont...

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

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

Imp point of AVL Tree are:

- It should be or is a BST.
- Height of Left Subtree - Height of Right Subtree = either {-1, 0, 1}
- This factor is known as Balance Factor, for each node the Balance factor would be either {-1, 0, 1}.
- Balance Factor (BF) should be check in every step.
- Duplicate element not allowed in BST.
- If Balance Factor is 2 or -2 then you should balance the tree.

AVL Rotations

- We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

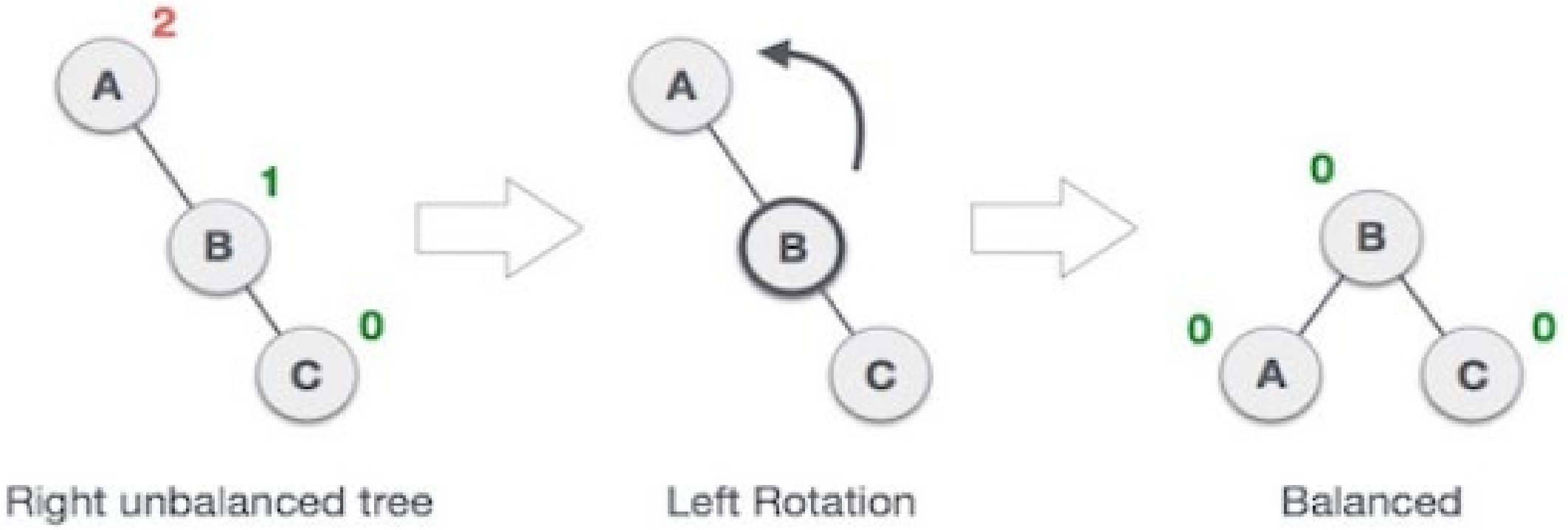
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

- Where node A is the node whose balance Factor is other than -1, 0, 1.

- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations.
- For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

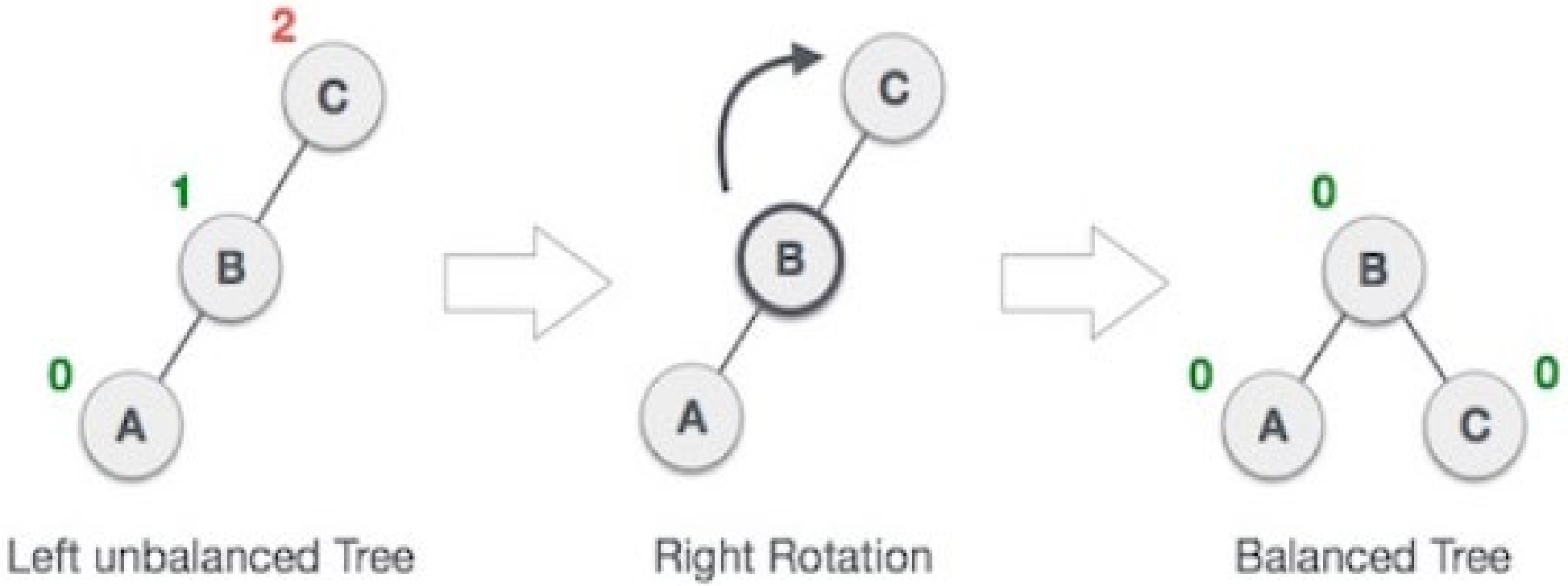
Left Rotation (LL Rotation)

- In left rotations, every node moves one position to left from the current position.



Right Rotation (RR Rotation)

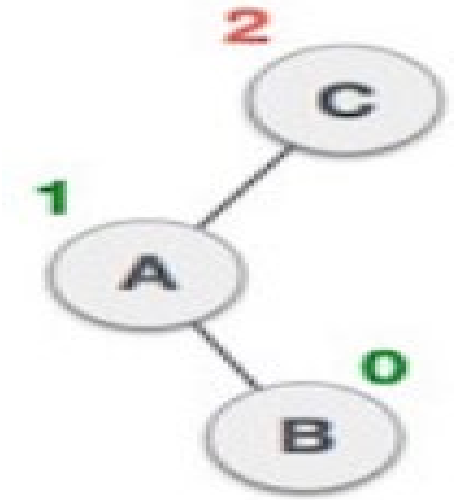
- In right rotations, every node moves one position to right from the current position.



3. LR Rotation

- Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.
- Let us understand each and every step very clearly:

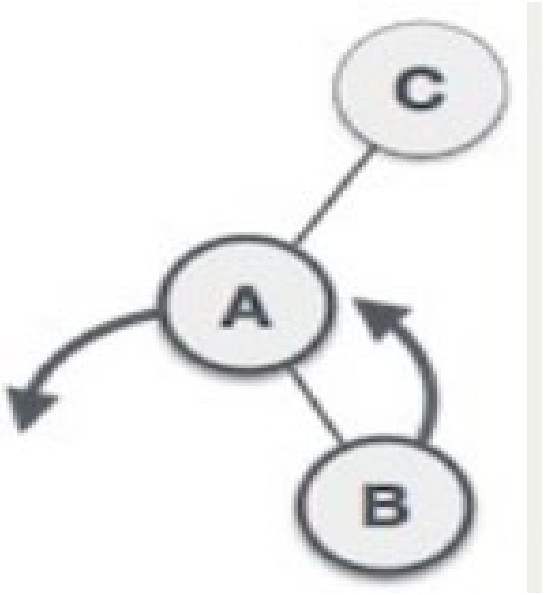
1. A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C



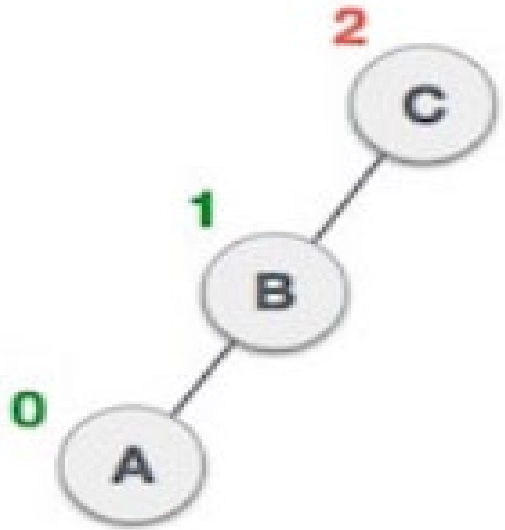
Cont...

2. As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first.

- By doing RR rotation, node A, has become the left subtree of B.

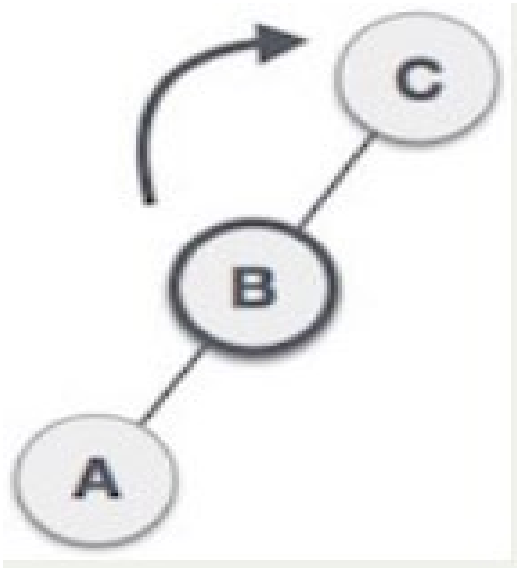


3. After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C

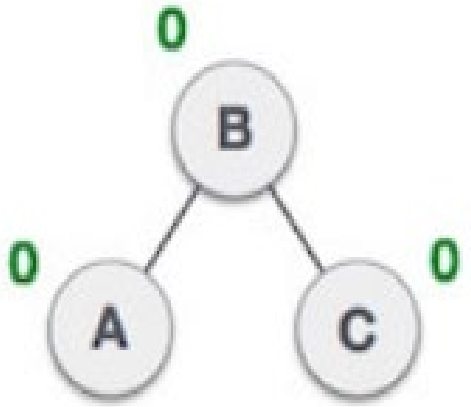


Cont...

4. Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B

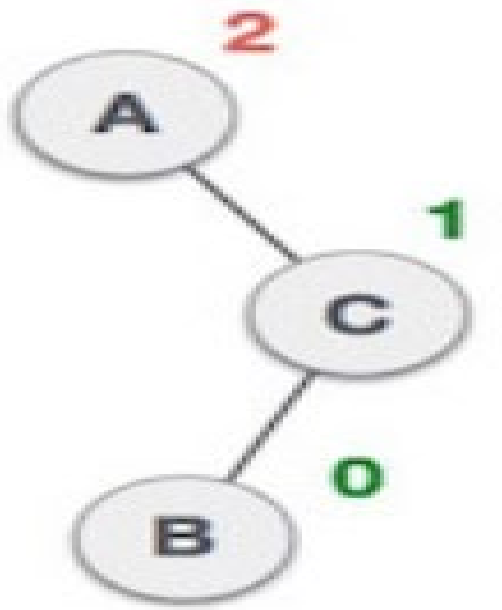


5. Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.



4. RL Rotation

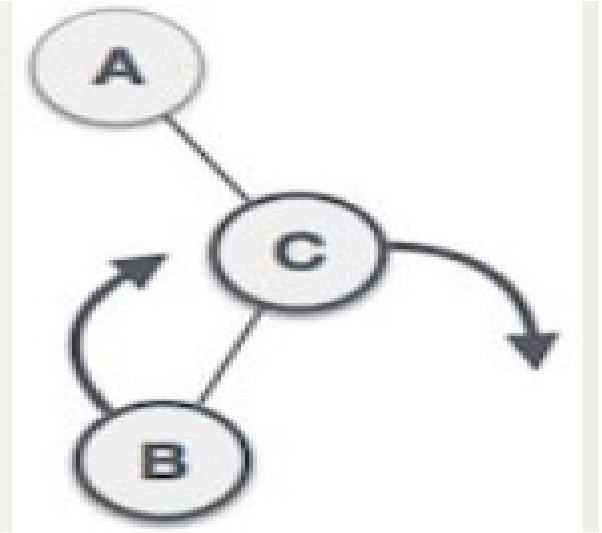
- Double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1. Lets take an Example
- A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2.
- This case is RL rotation where: Inserted node is in the left subtree of right subtree of A



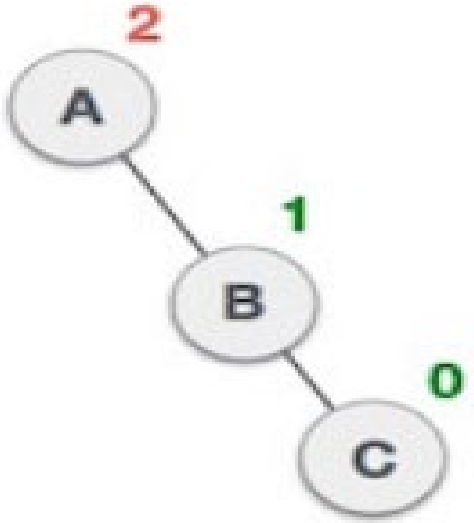
Cont...

1.As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first.

- By doing RR rotation, node C has become the right subtree of B.

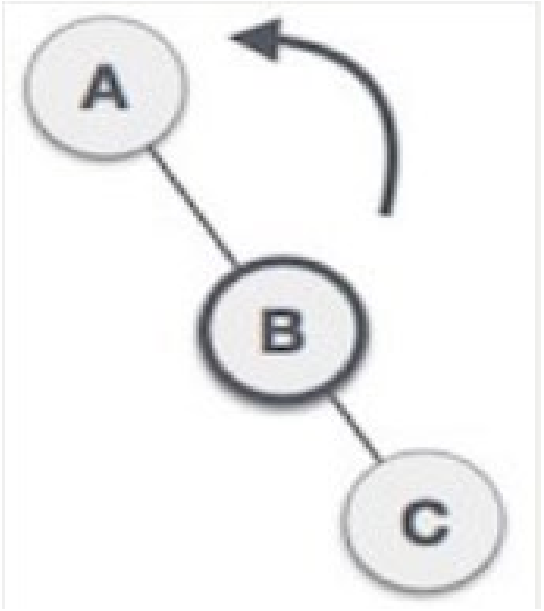


2. After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.

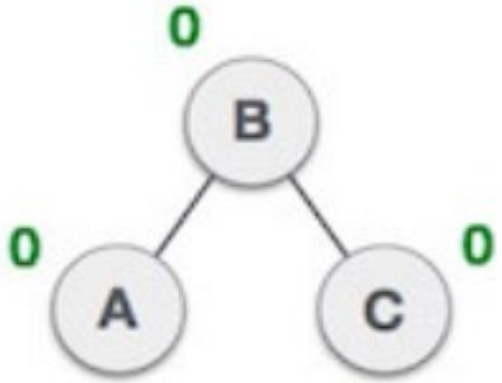


Cont...

3. Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.

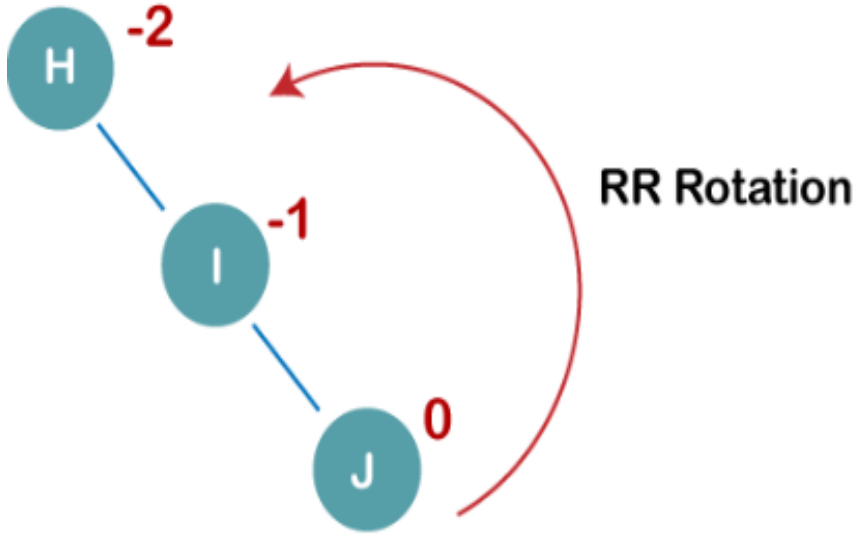


4. Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

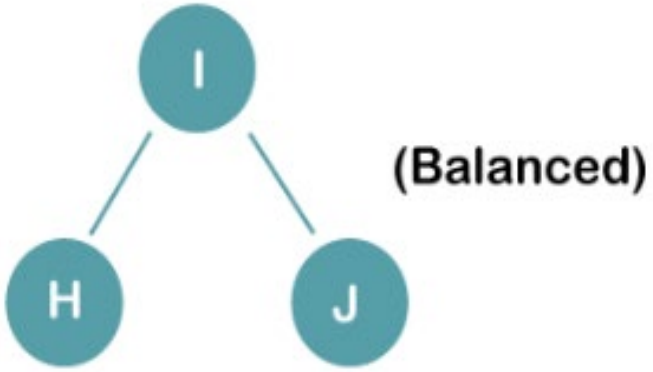


Q: Construct an AVL tree having the following elements H, I, J, B, A, E, C, F, D, G, K, L

- 1. Insert H, I, J:
- On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2.
- Since the BST is right-skewed, we will perform RR Rotation on node H.



- The resultant balance tree is:

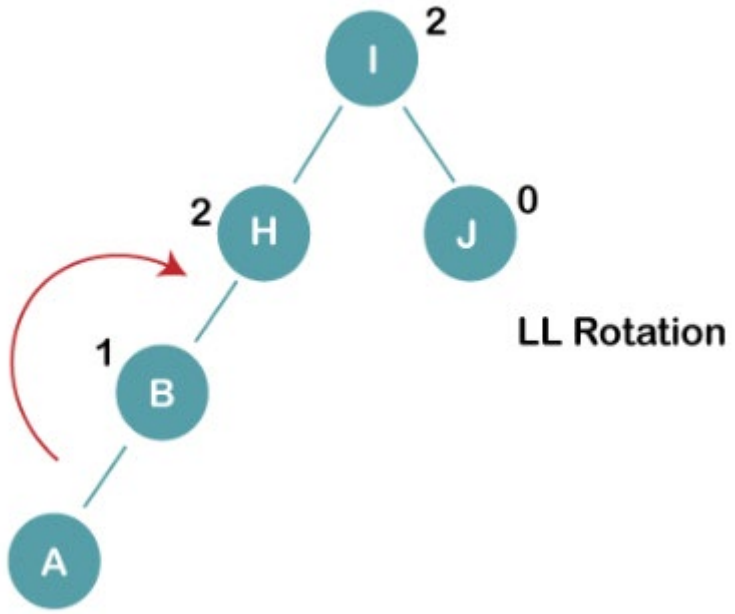


Cont...

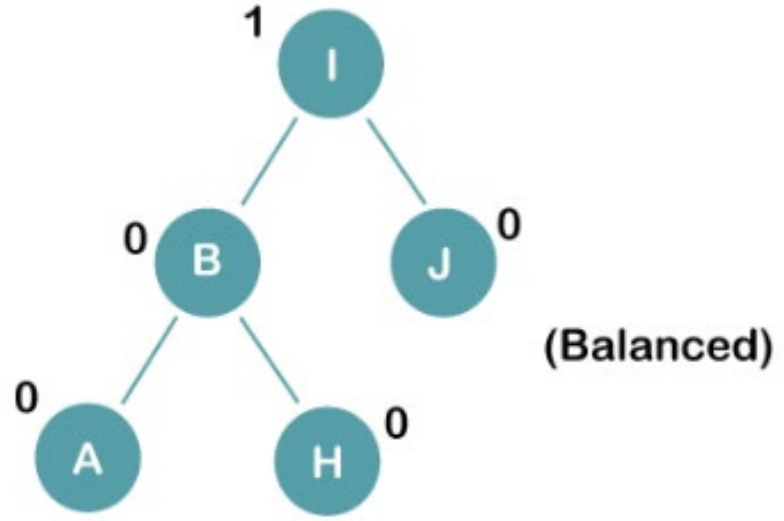
2. Insert B, A:

On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H.

Since the BST from H is left-skewed, we will perform LL Rotation on node H.



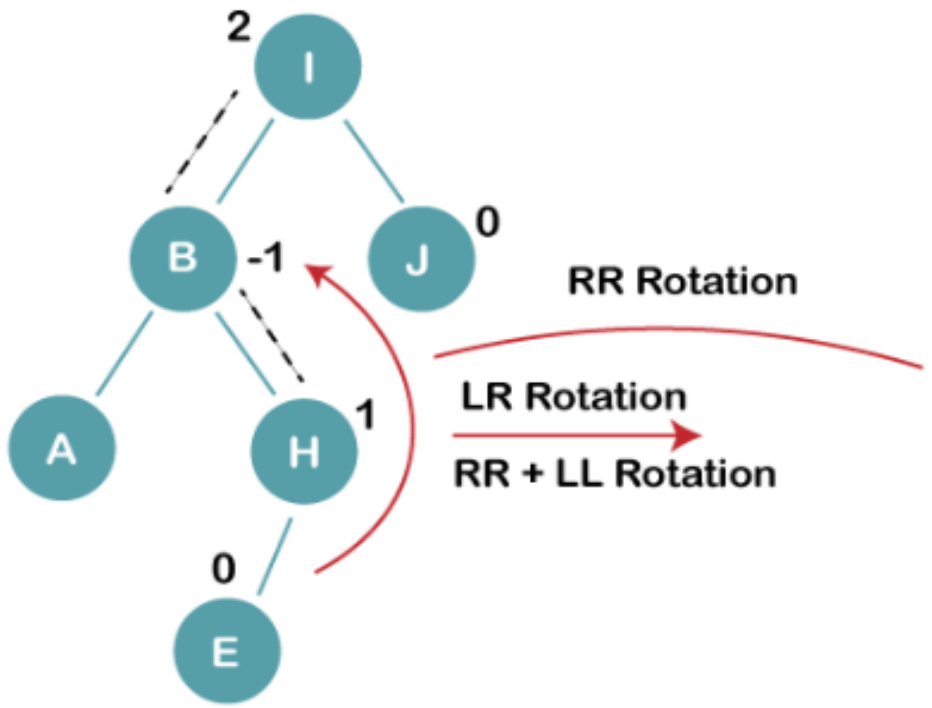
The resultant balance tree is:



Cont....

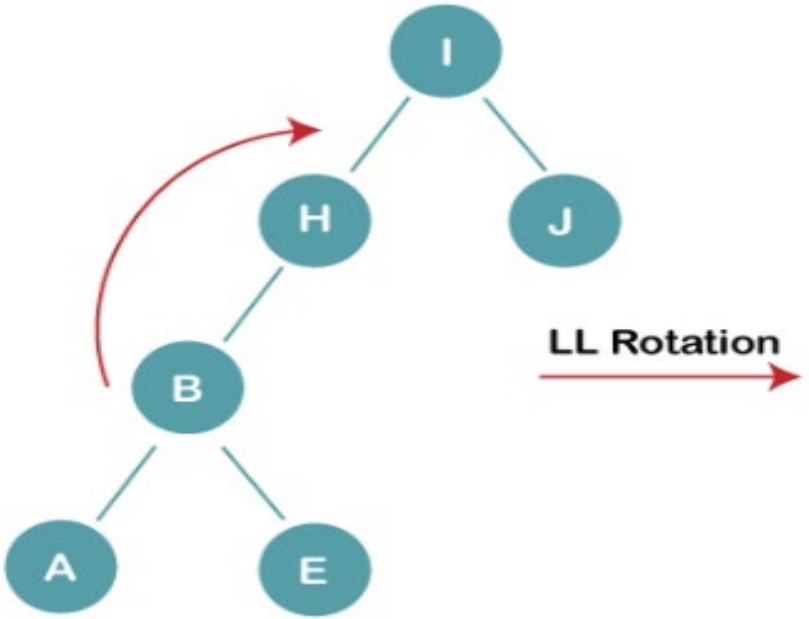
3. Insert E:

- On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation



3 a) We first perform RR rotation on node B

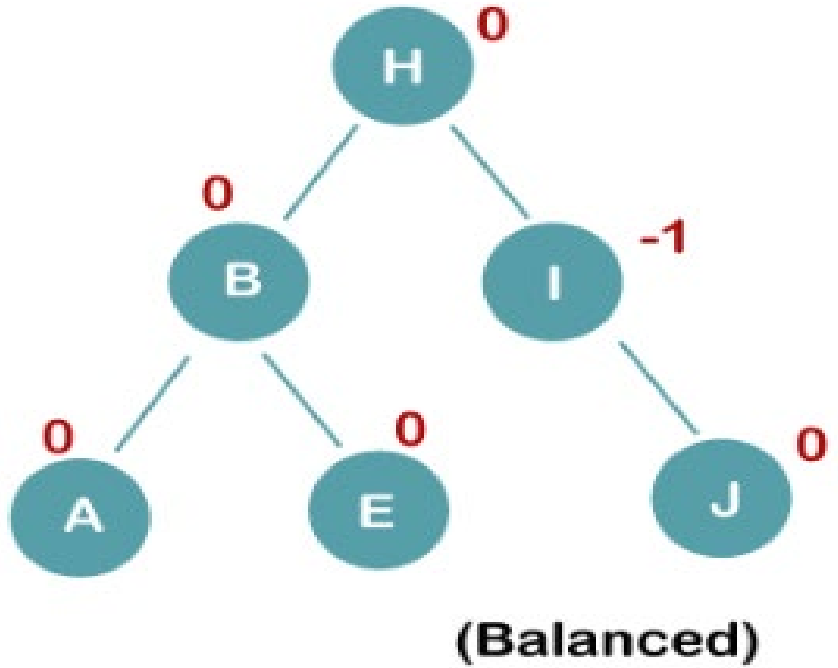
The resultant tree after RR rotation is:



Cont...

3b) We first perform LL rotation on the node I

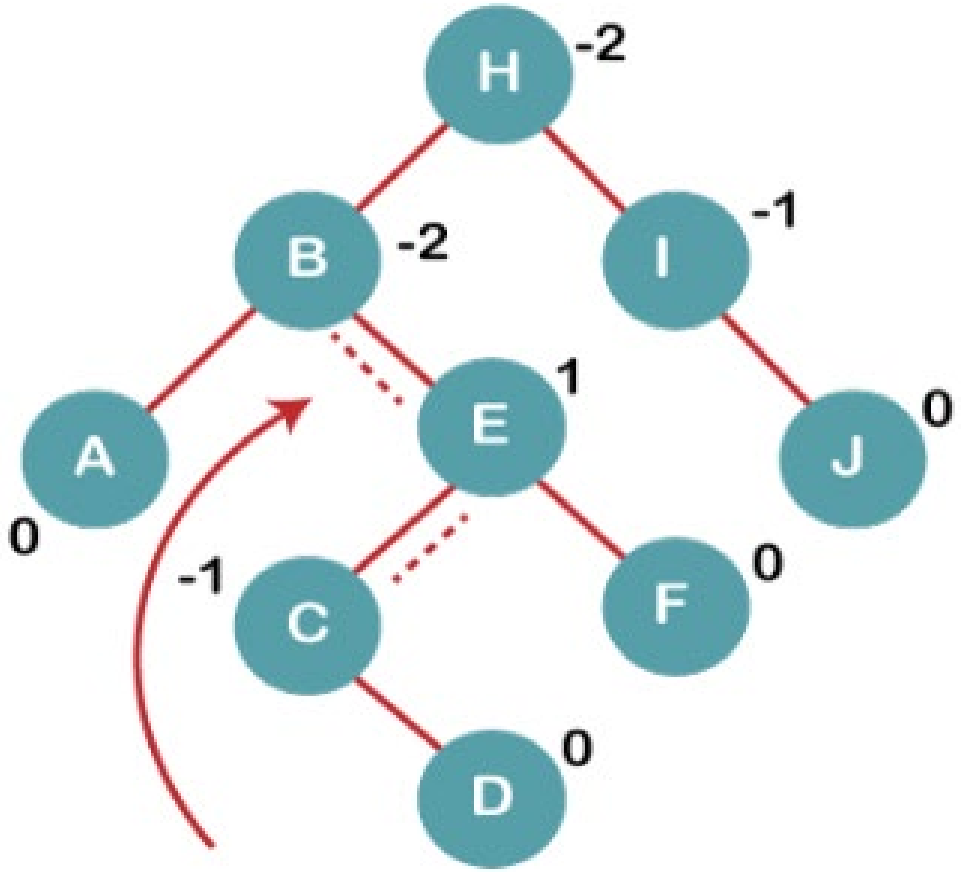
The resultant balanced tree after LL rotation is:



Cont...

- 4. Insert C, F, D

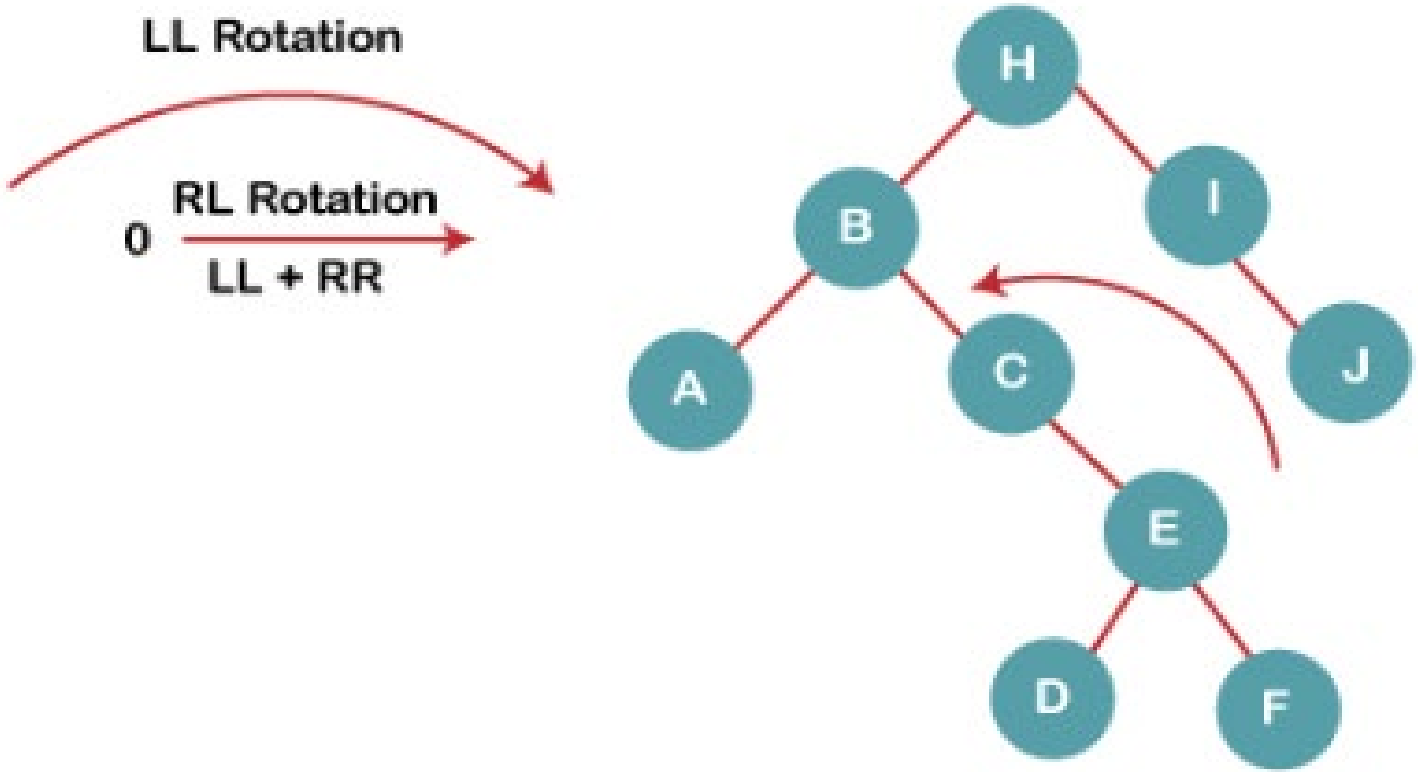
- On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.



Cont...

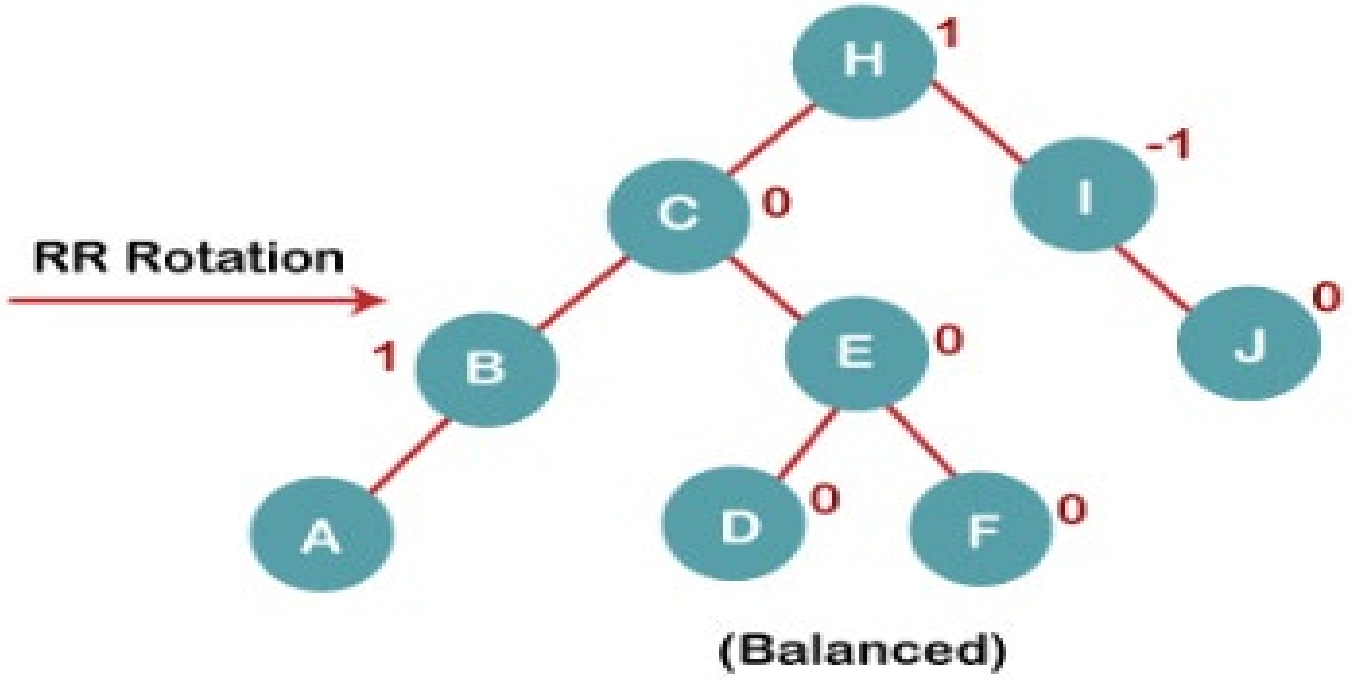
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:



Cont...

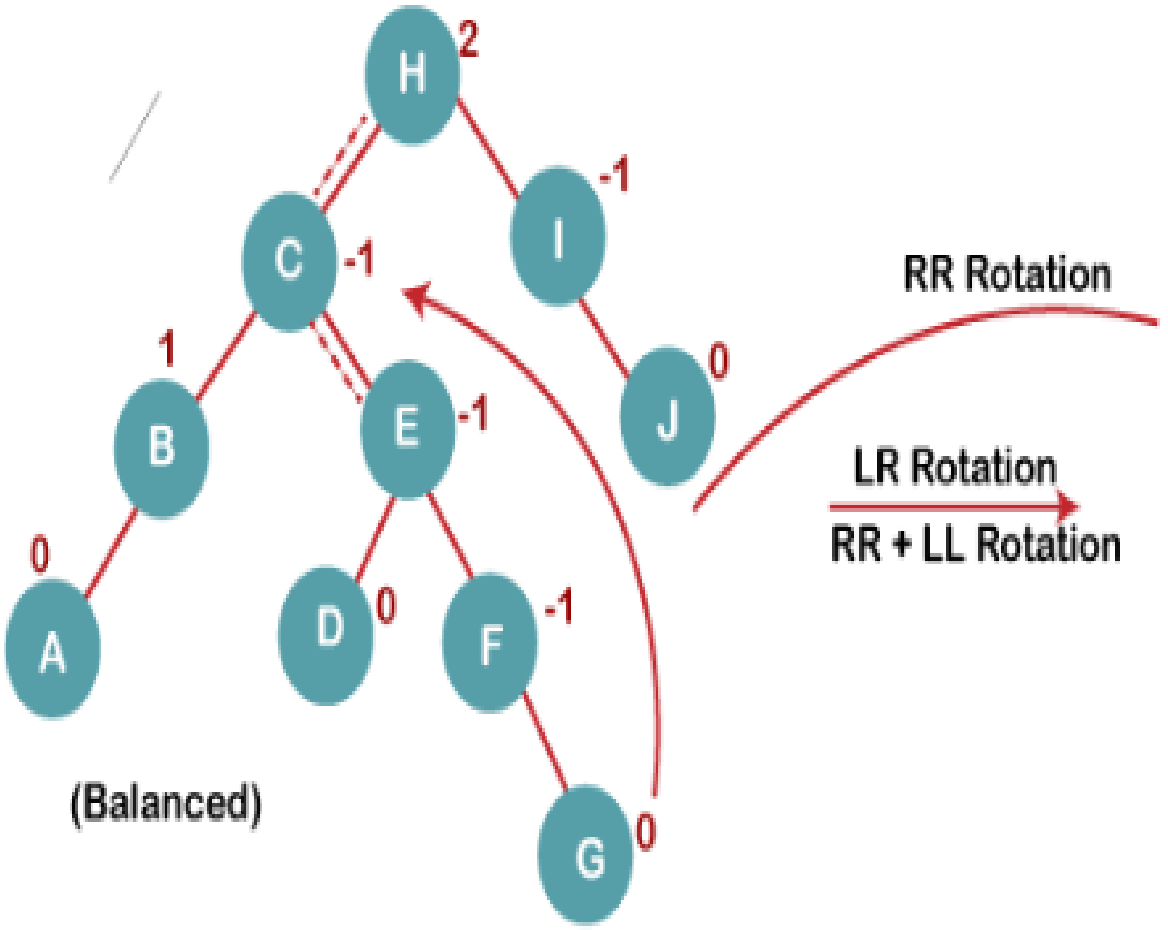
- 4b) We then perform RR rotation on node B
- The resultant balanced tree after RR rotation is:



Cont...

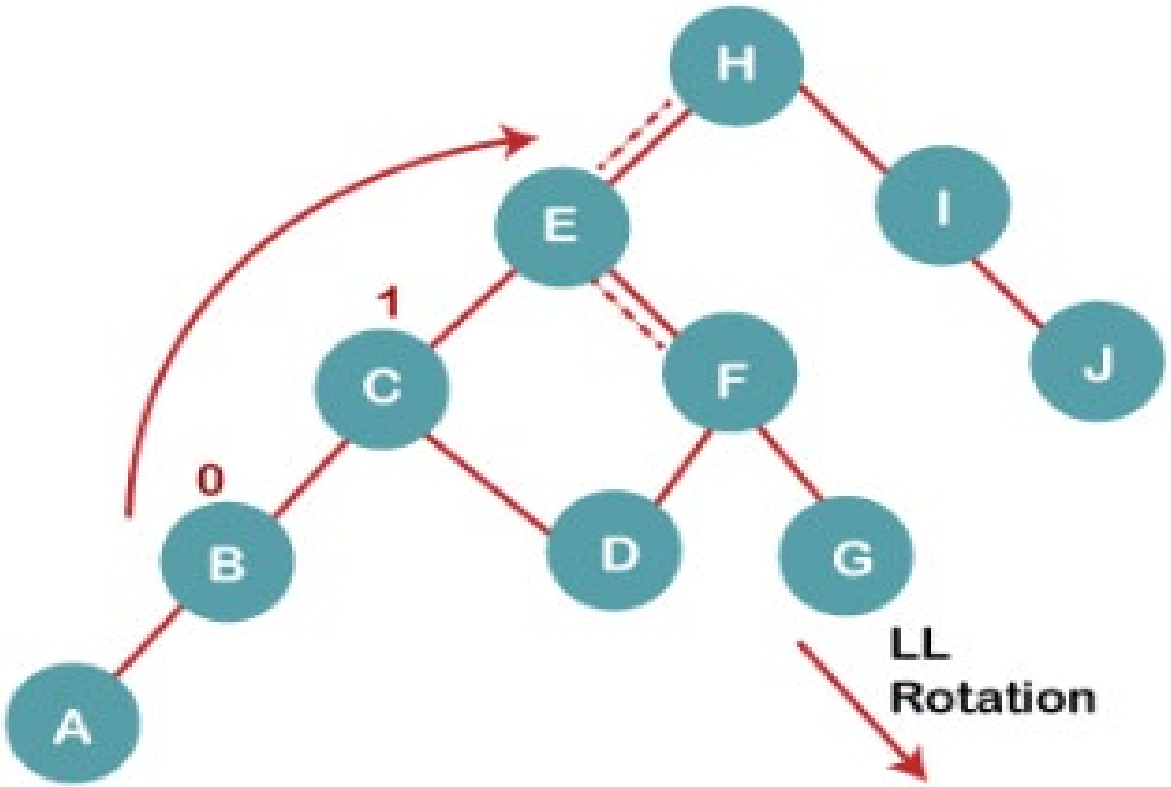
- 5. Insert G:
- On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

- 5 a) We first perform RR rotation on node C



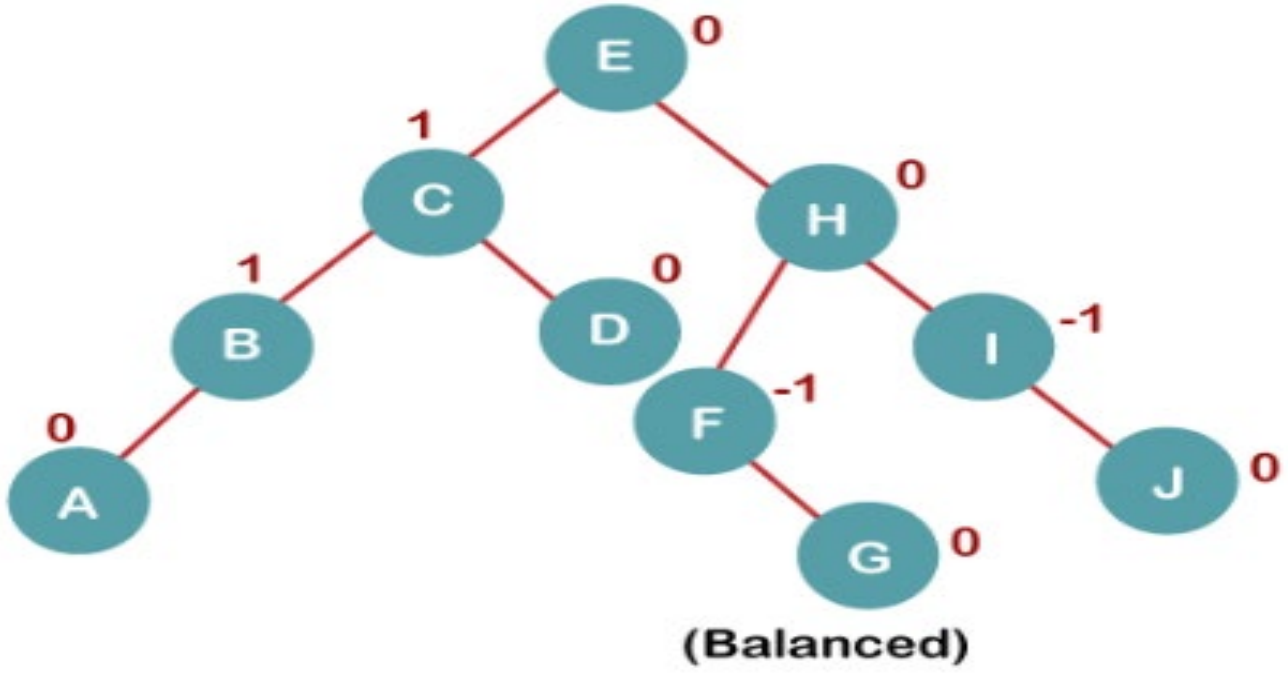
Cont...

- The resultant tree after RR rotation is:



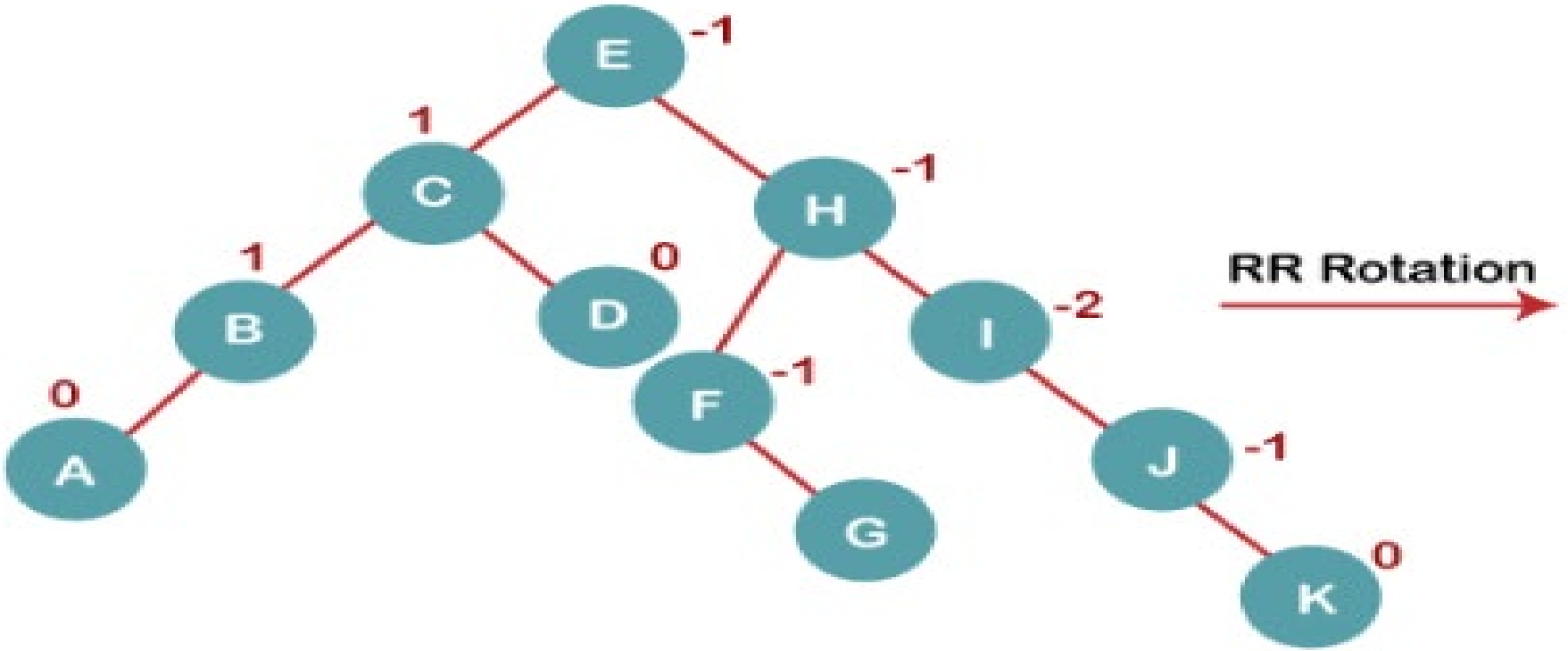
Cont...

- 5 b) We then perform LL rotation on node H
- The resultant balanced tree after LL rotation is:

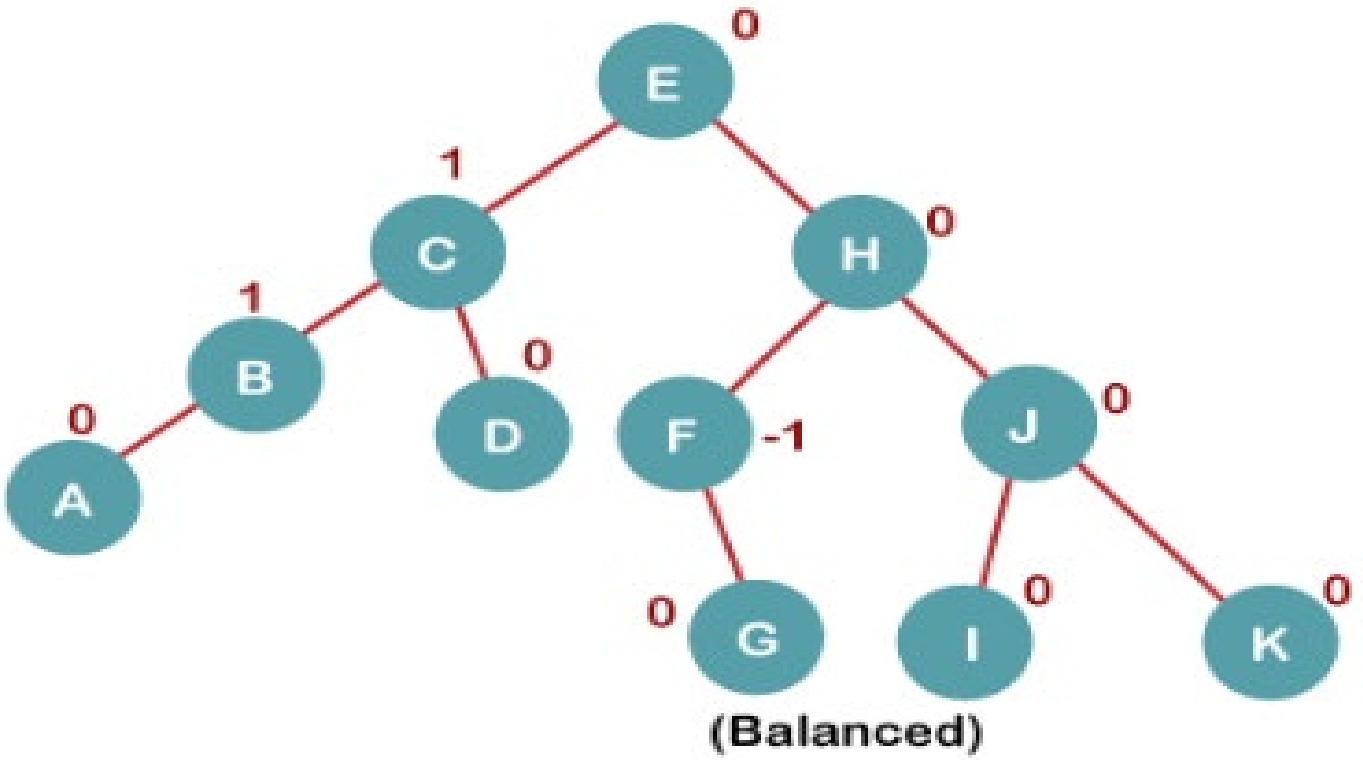


Cont...

- 6. Insert K
- On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.



The resultant balanced tree after RR rotation is:



(Balanced)

Cont...

7. Insert L:

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1.

Hence the tree is a Balanced AVL tree