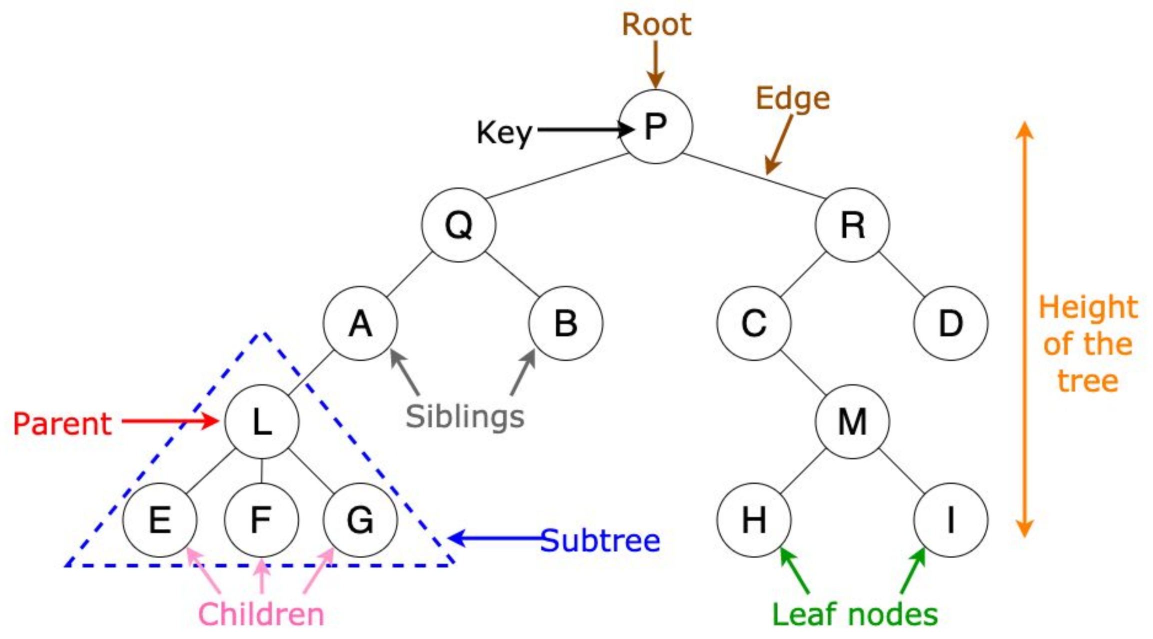
Tree Data Structure



Trees – A Nonlinear Approach to Data Organization

- Productivity experts often emphasize "nonlinear" thinking as a key to innovation and breakthroughs.
- In computing, trees are a primary example of a nonlinear structure, offering a powerful way to organize and access data more efficiently than linear structures, such as lists.

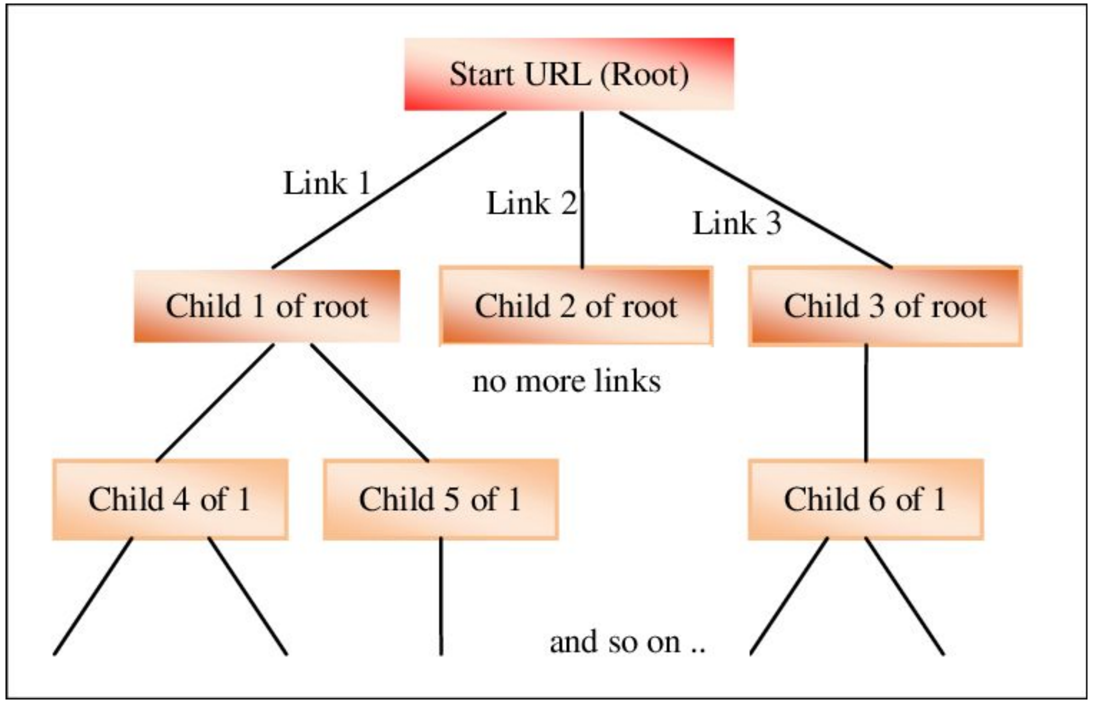
Trees – A Nonlinear Approach to Data Organization

Question: Imagine you’re looking for a book in a large library where all the books are arranged in a single row. How efficient would it be to find a specific book if you had to walk through every title one by one?

Ans: This linear arrangement would be slow and impractical. A **tree structure** would organize the library hierarchically: starting with broad categories (like Fiction and Non-Fiction), then genres (e.g., Mystery, History), and finally individual shelves for each genre. With this structure, you can go directly to the relevant section, saving time and making it easy to find specific books.

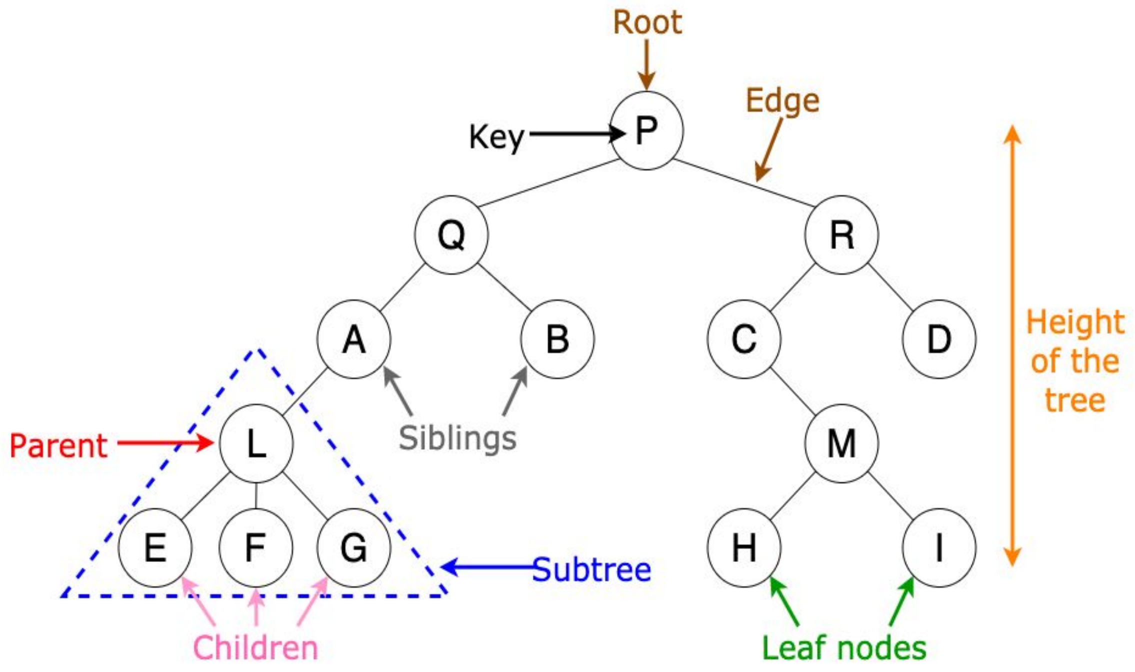
Why Trees Are Essential in Data Organization:

- Trees provide a **natural hierarchy** for organizing data, making them fundamental to systems like file directories, databases, websites, and GUIs.
- With trees, algorithms can be implemented more quickly and efficiently, allowing for faster performance in data-intensive applications.



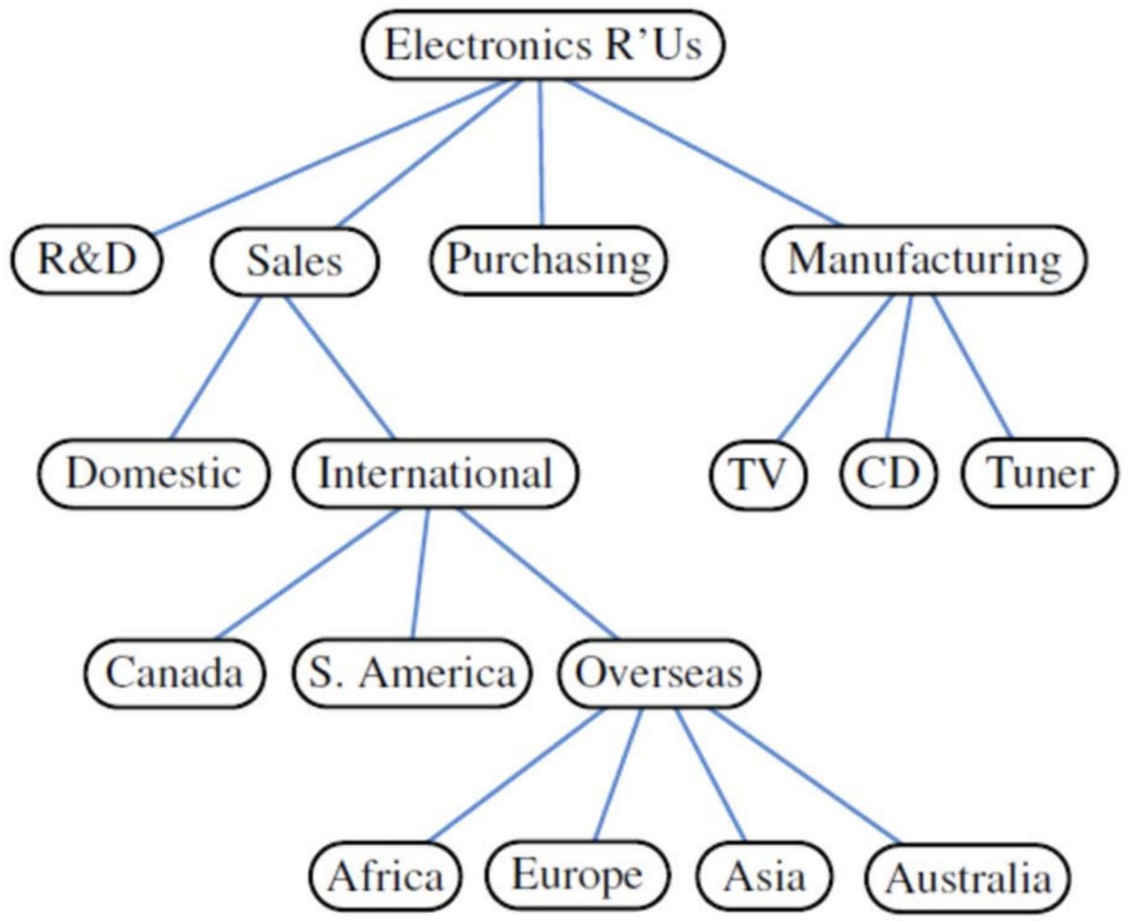
Understanding Nonlinearity in Trees:

- Trees are "nonlinear" because their structure is hierarchical rather than sequential.
    - In a tree, relationships aren't simply "before" or "after"; instead, they form a hierarchy where some nodes are **"above"** others (parents) and some are **"below"** (children).
    - This hierarchy closely resembles family trees, giving rise to terms like **parent, child, ancestor,** and **descendant** to define relationships.



Formal Defination of Tree

- Tree: an ADT that stores elements hierarchically
- Each element has
    - A parent element
    - Zero or more child elements
    - Root: top element



Formal Defination of Tree

Definition: A tree T is a set of nodes storing elements and a parent-child relation

If T is non-empty, it has a special node, called root of T, that has no parent

Every non-root node has a unique parent w; every node with parent w is a child of w

Basic Terms

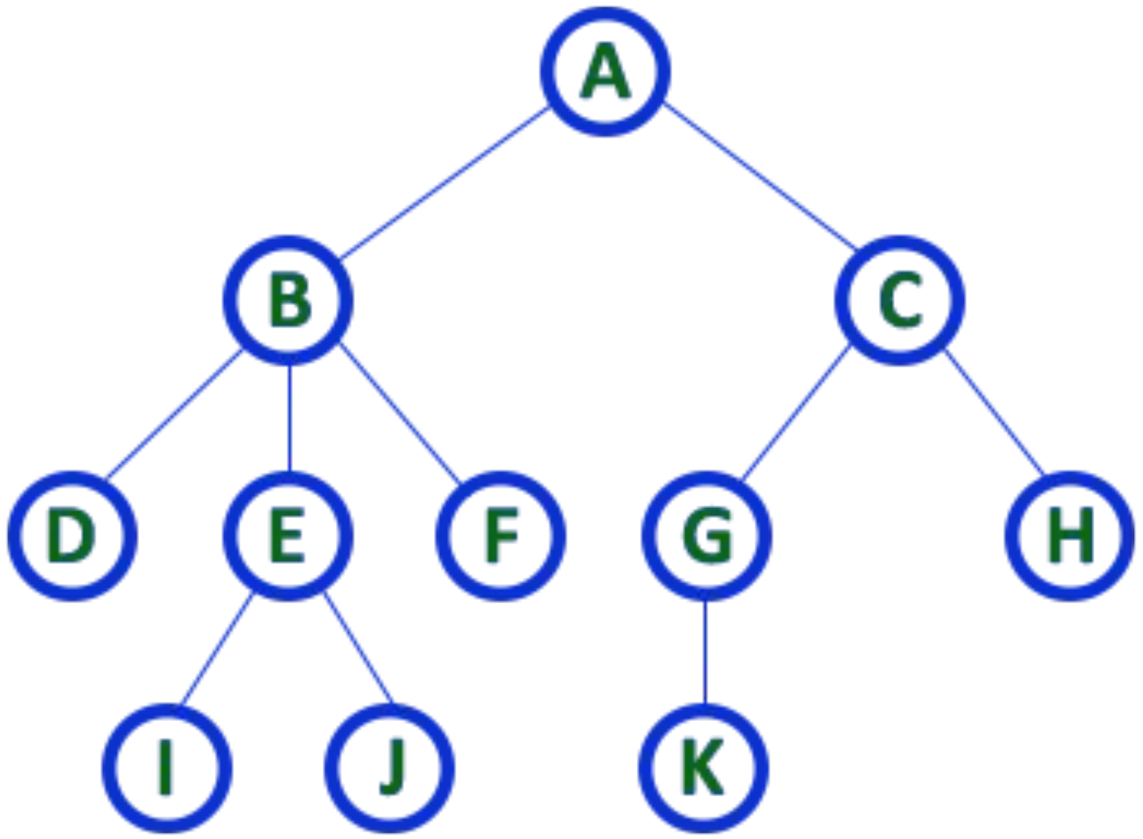
- Sibling: two nodes that are children of the sam e parent
- External node (leaves): a node v is external if i t has no children
- Internal node: a node v is internal if it has one or more children
- Ancestor: u is an ancestor of v if u = v or u is a n ancestor of the parent of v
- Descendant: v is a descendant of u if u is an a ncestor of v

Basic Terms

Subtree
of T rooted at v: the tree consisting of all the descendants of v in T (including v itself)

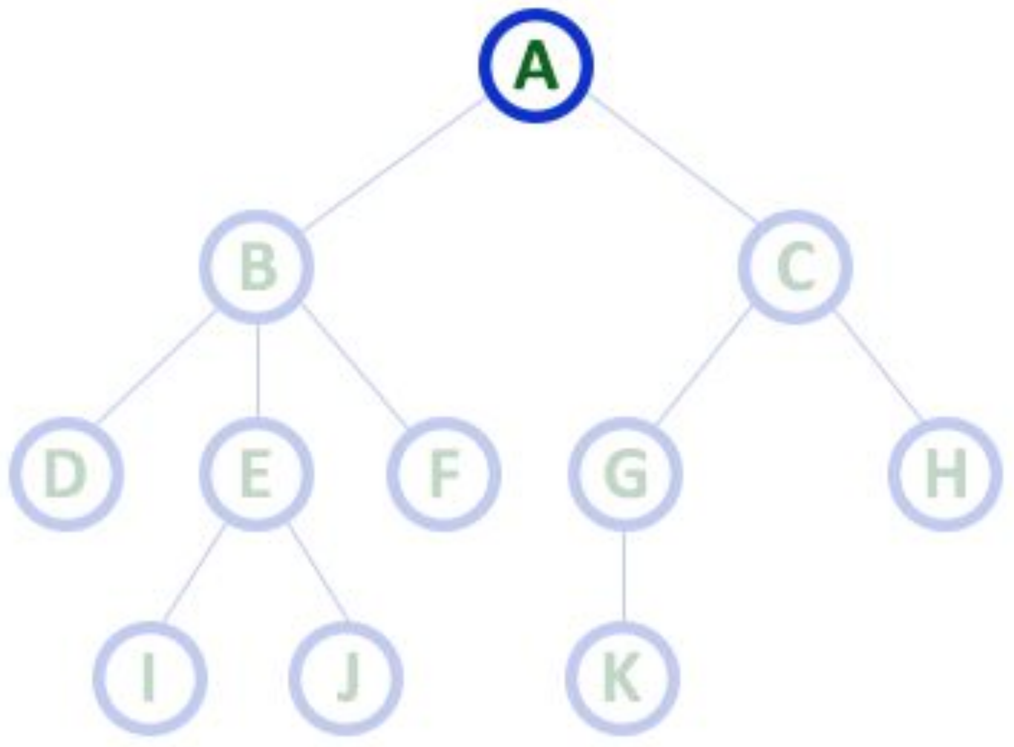
Edge: an edge of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa

Path: a path of T is a sequence of nodes such that any
two consecutive sequence form an edge



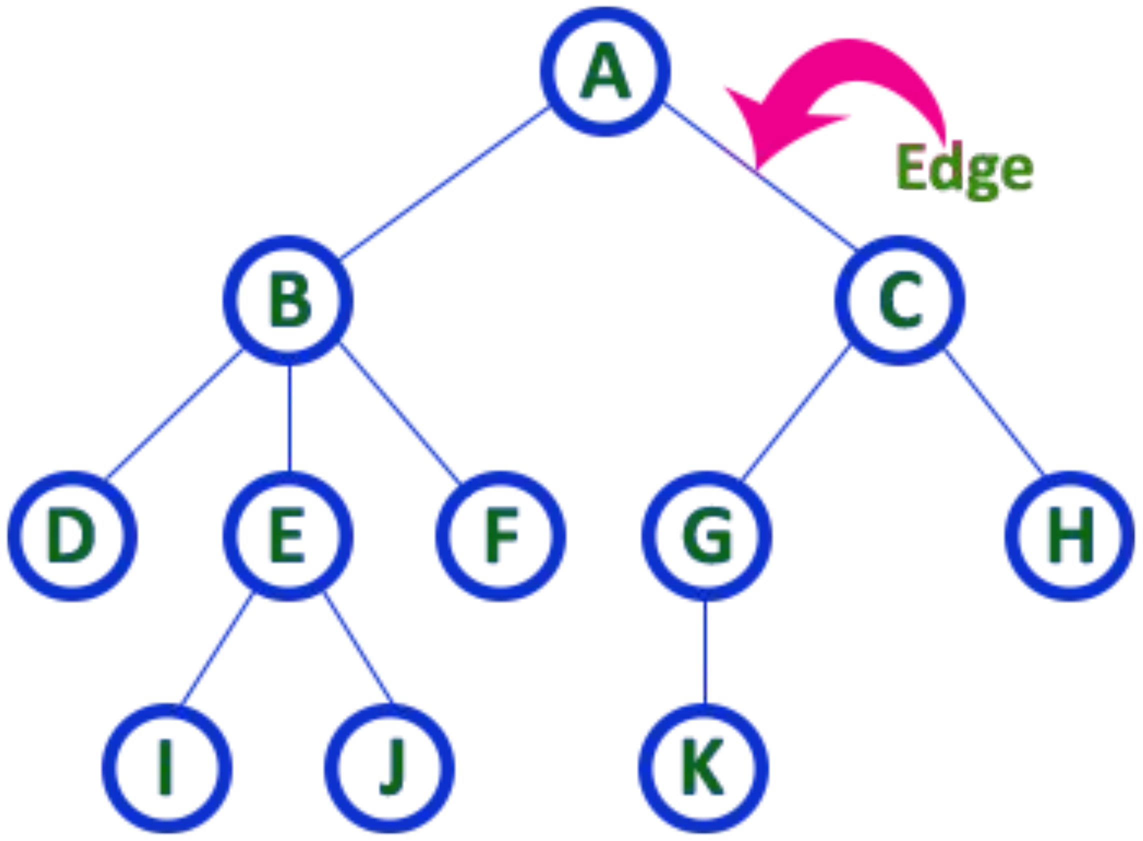
TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

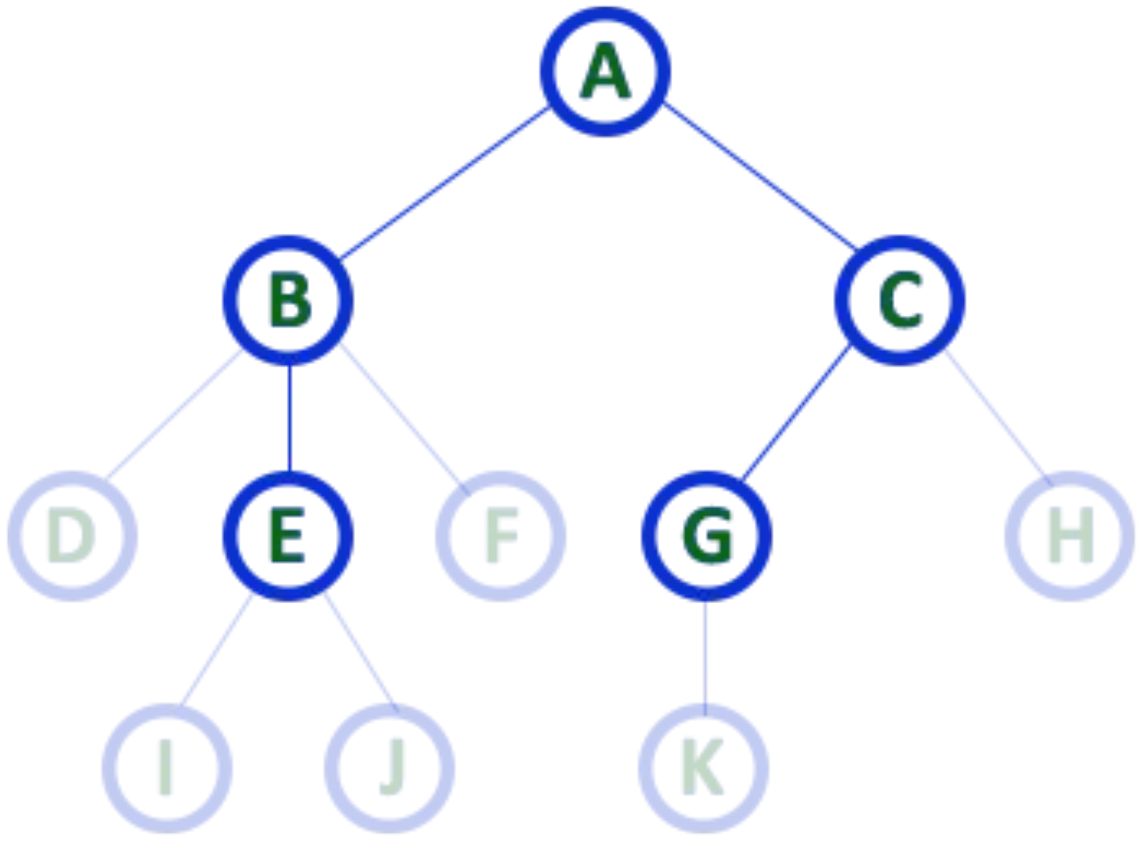


Here ‘A’ is the ‘root’ node

- In any tree the first node is called as ROOT node

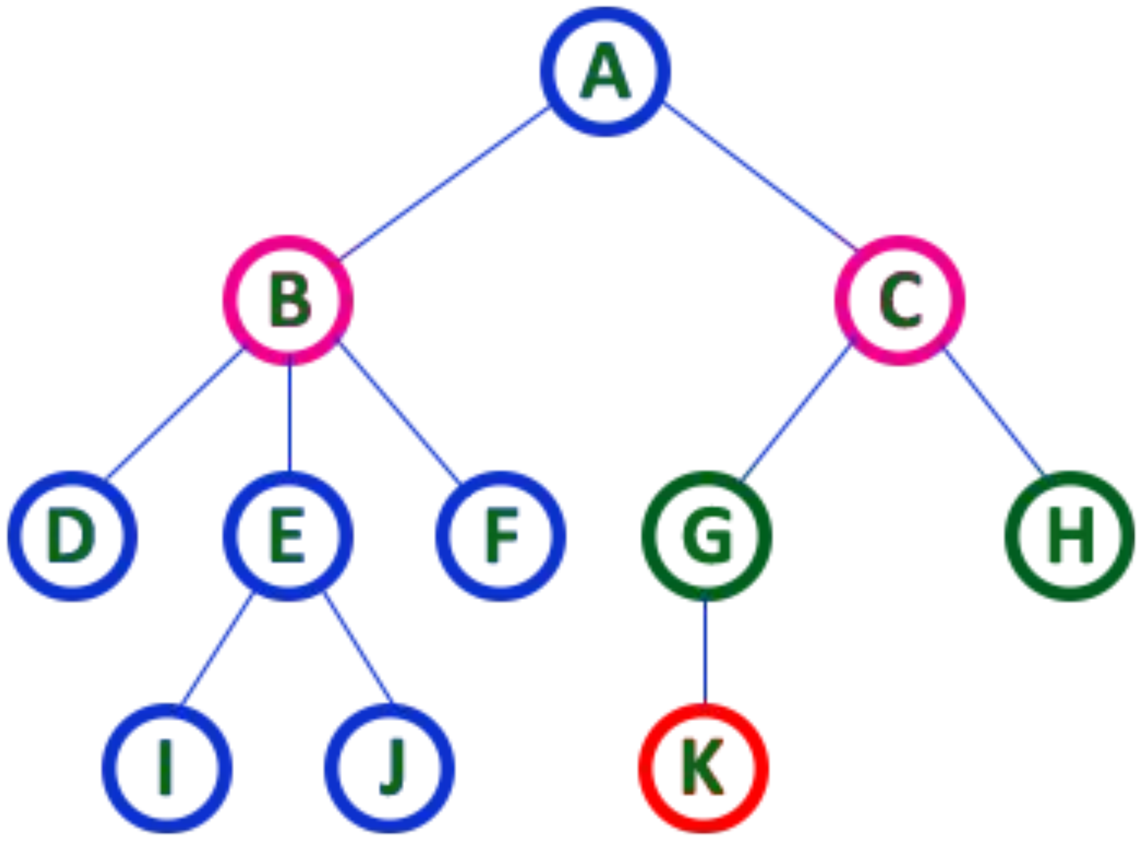


- In any tree, 'Edge' is a connecting link between two nodes.



Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called ‘Parent’
- A node which is predecessor of any other node is called ‘Parent’

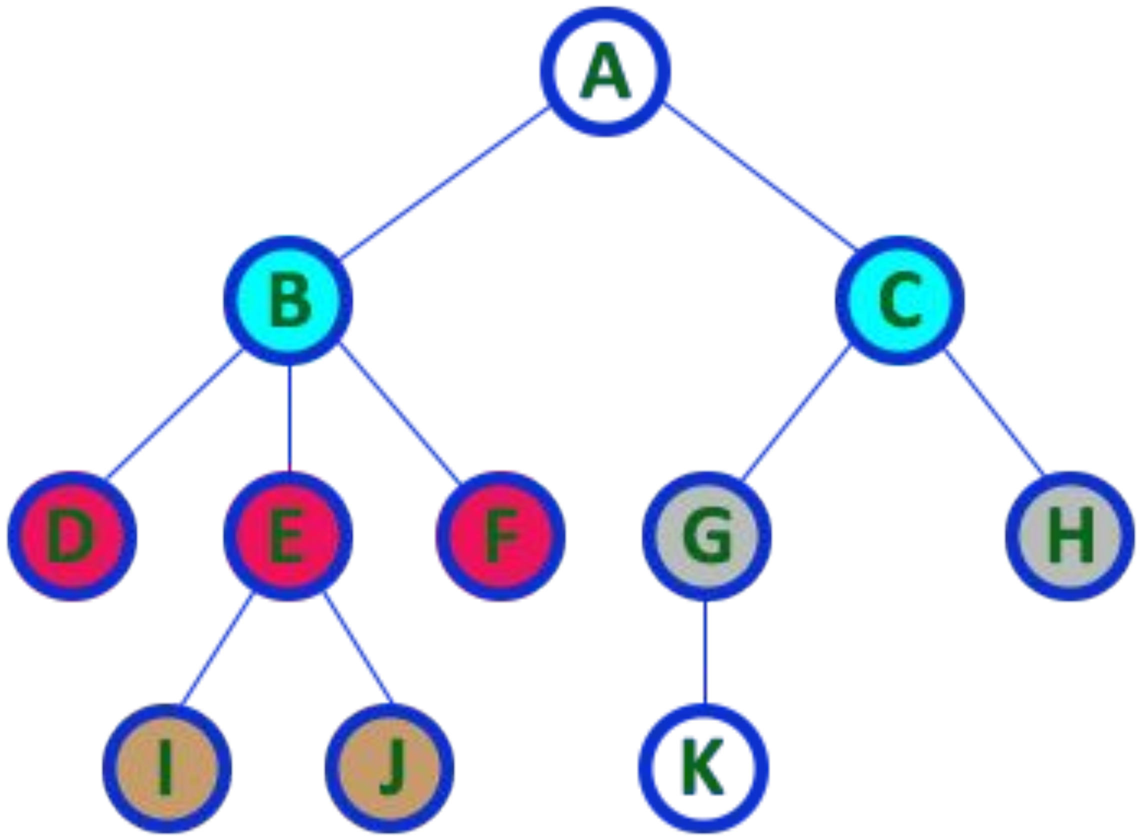


Here B & C are Children of A

Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node



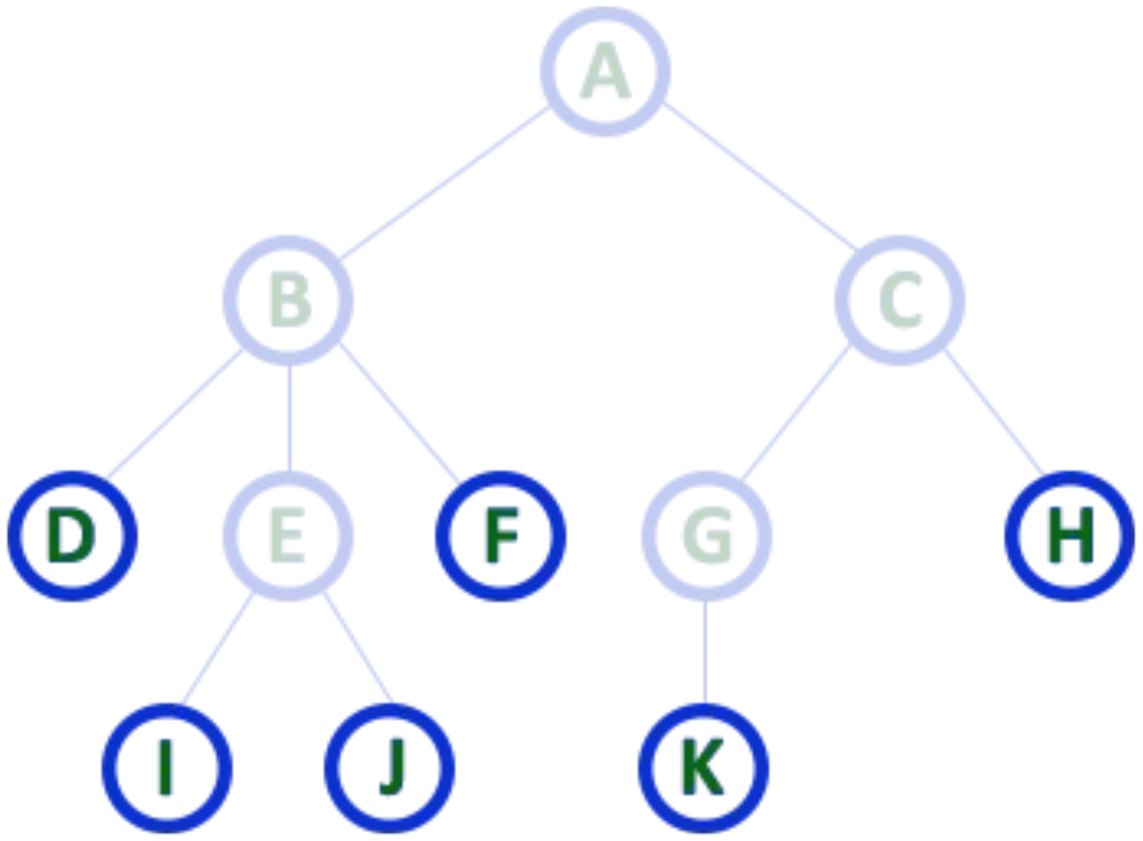
Here B & C are Siblings

Here D E & F are Siblings

Here G & H are Siblings

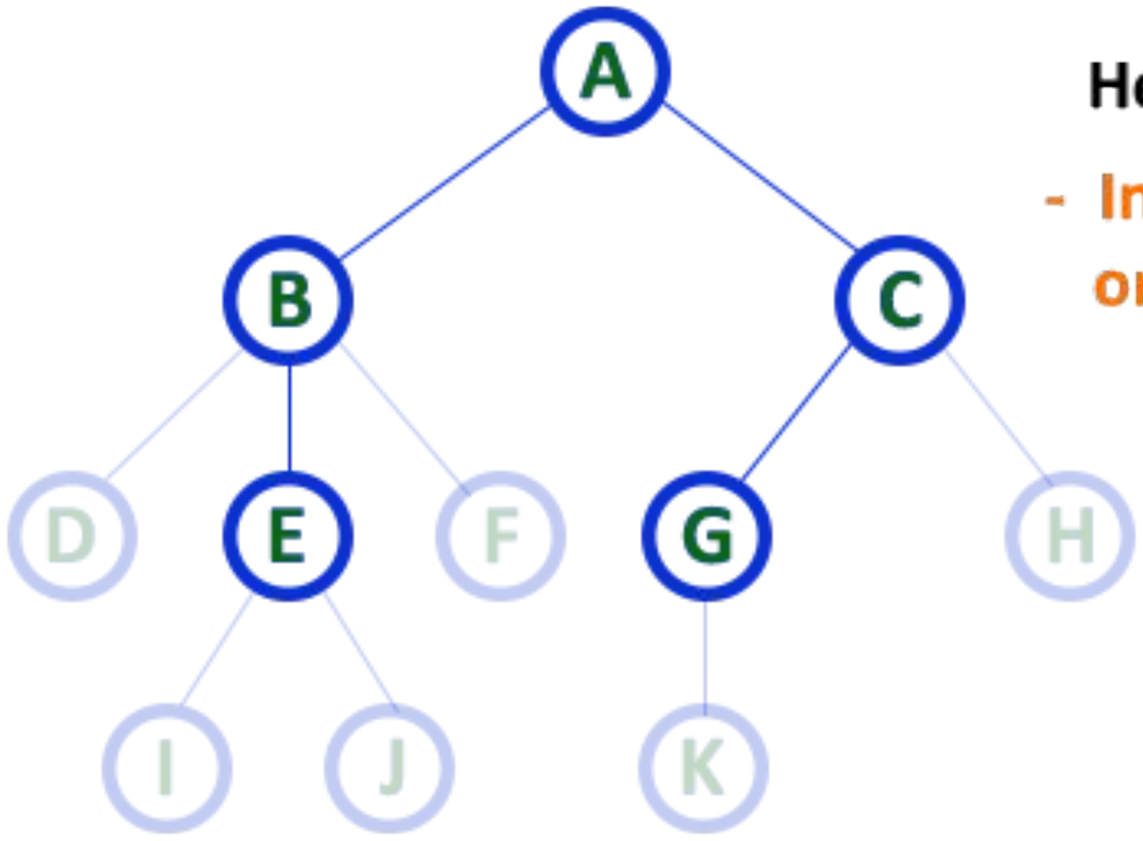
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'



Here D, I, J, F, K & H are Leaf nodes

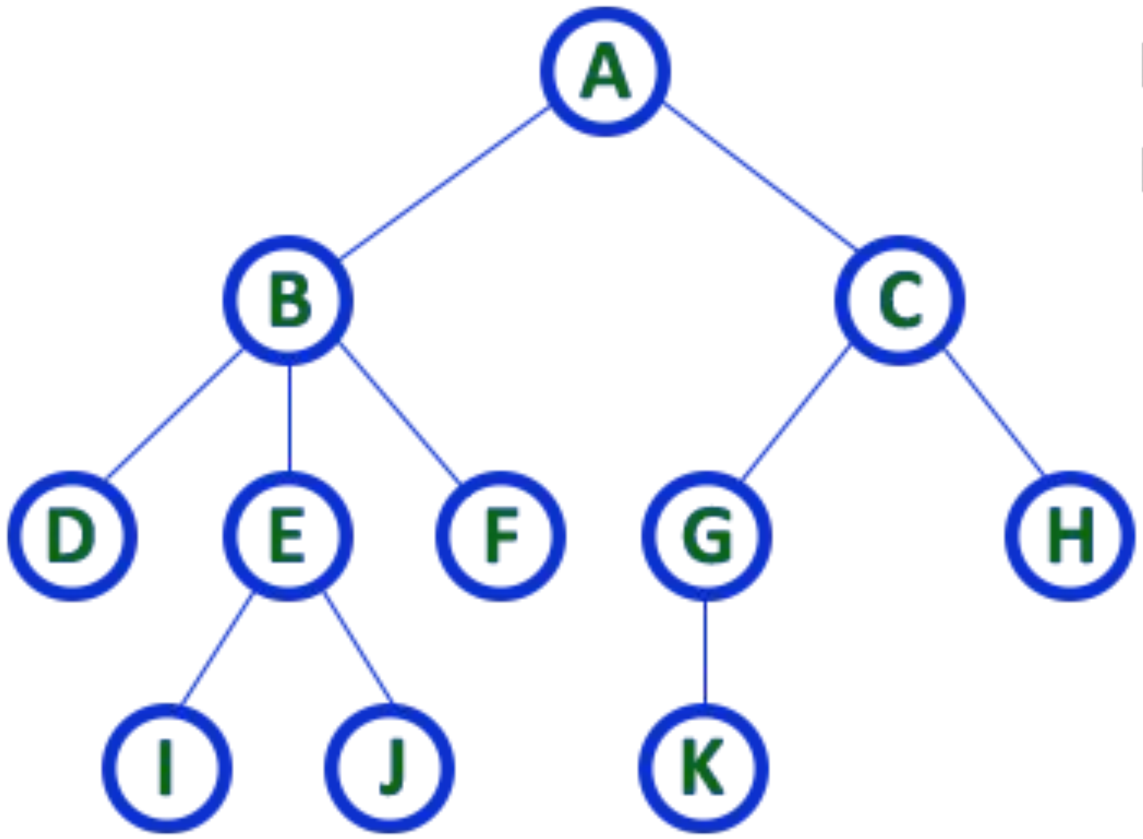
- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node



Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

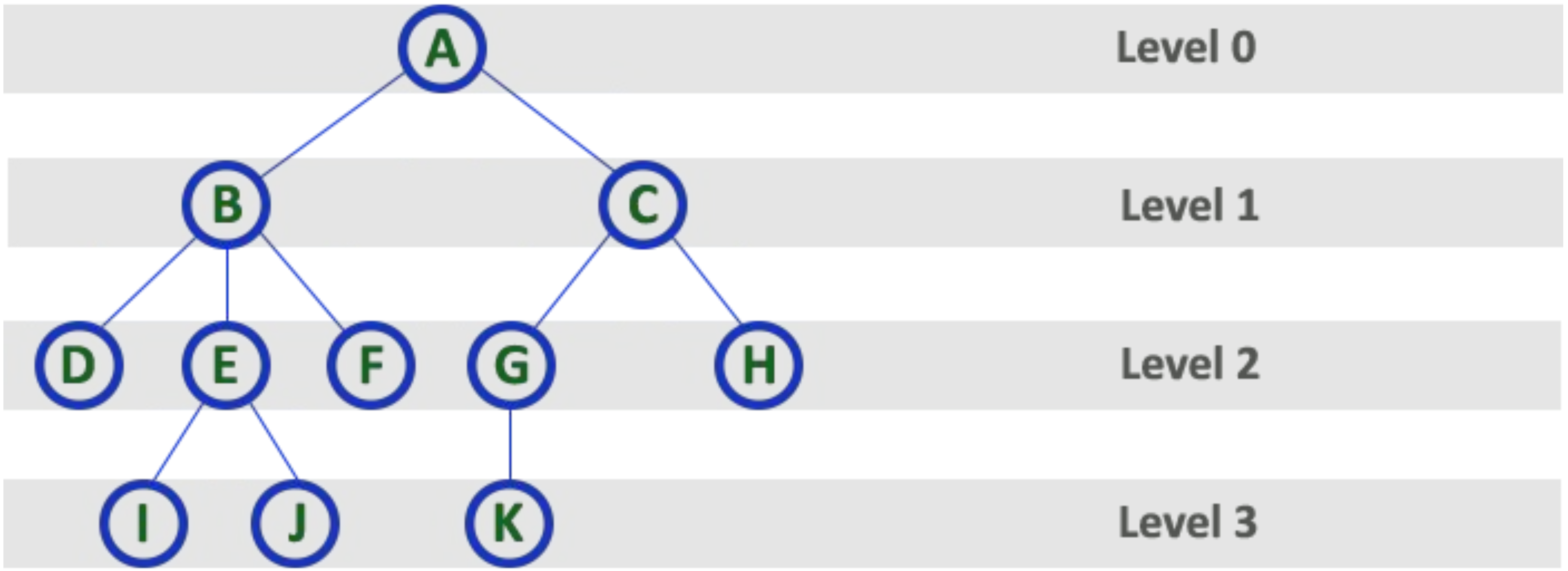


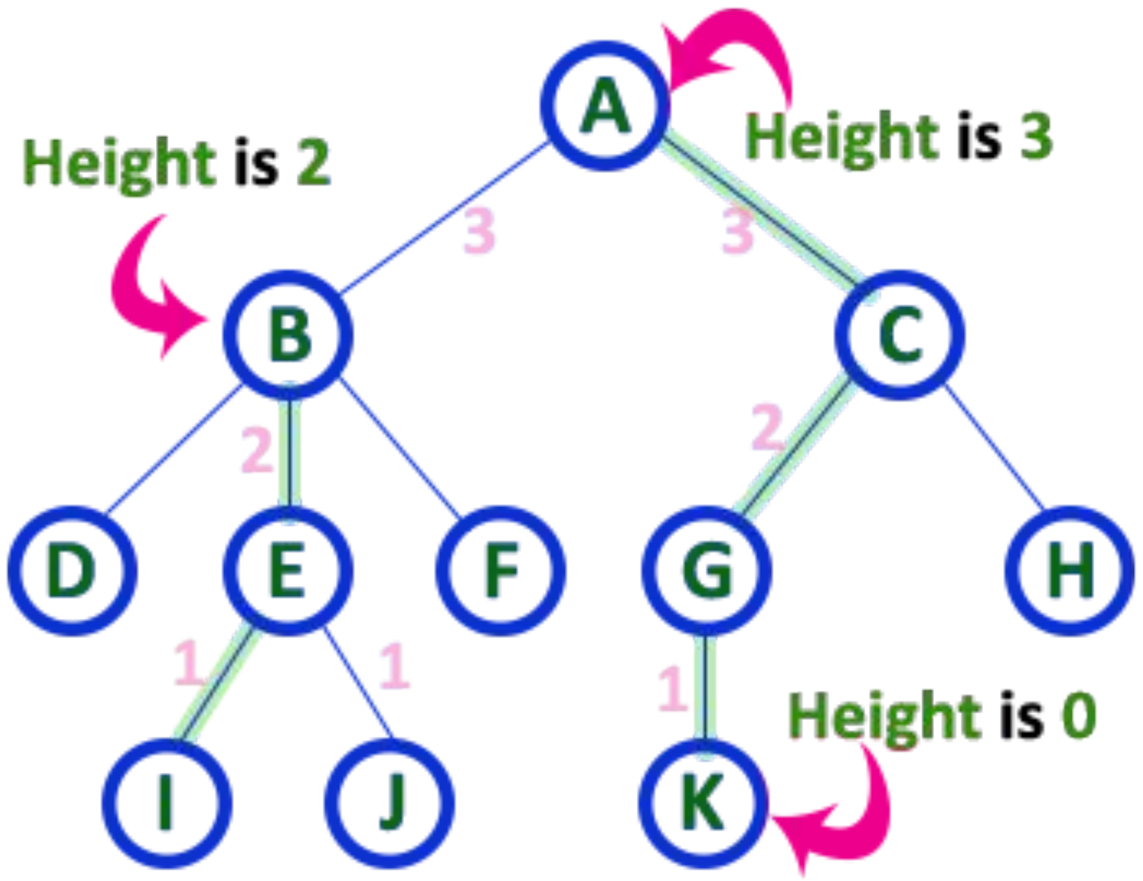
Here Degree of B is 3

Here Degree of A is 2

Here Degree of F is 0

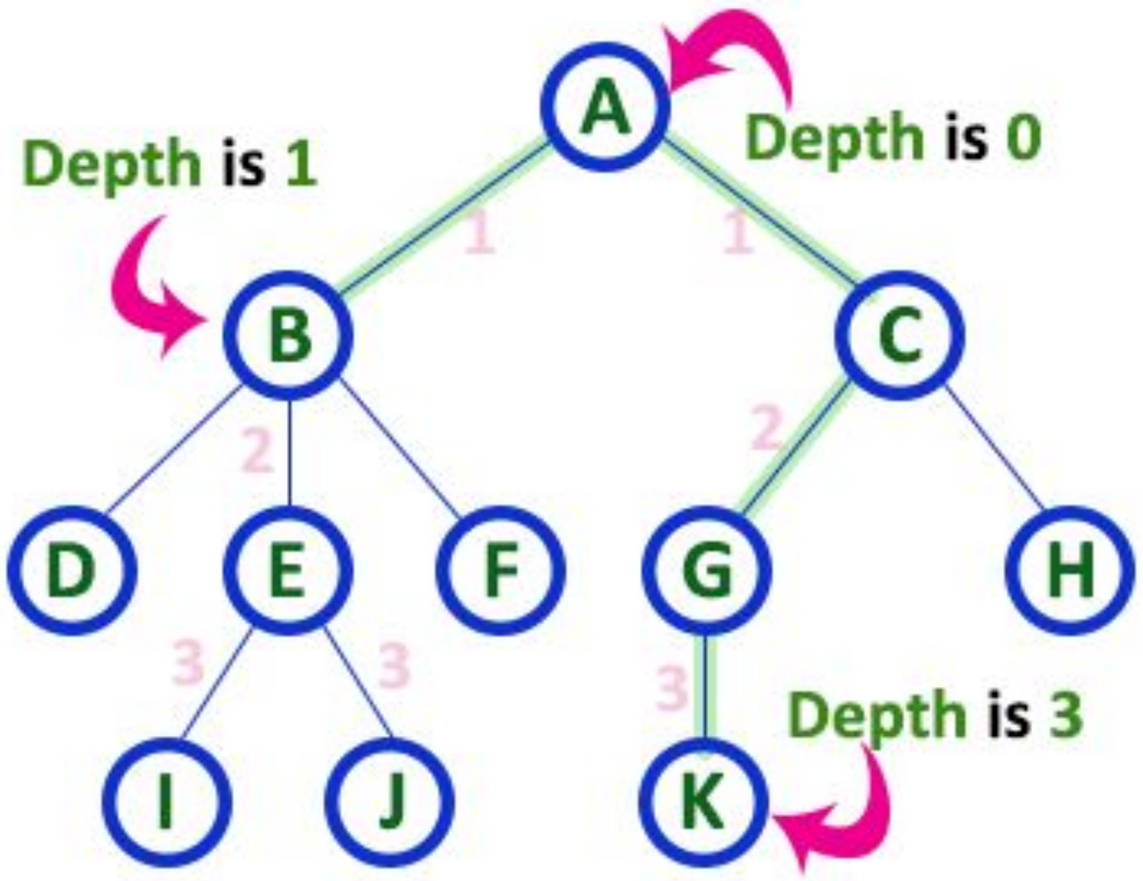
- In any tree, 'Degree' of a node is total number of children it has.


A
B
C
D
E
F
G
H
I
J
K
Level 0
Level 1
Level 2
Level 3



Here Height of tree is 3

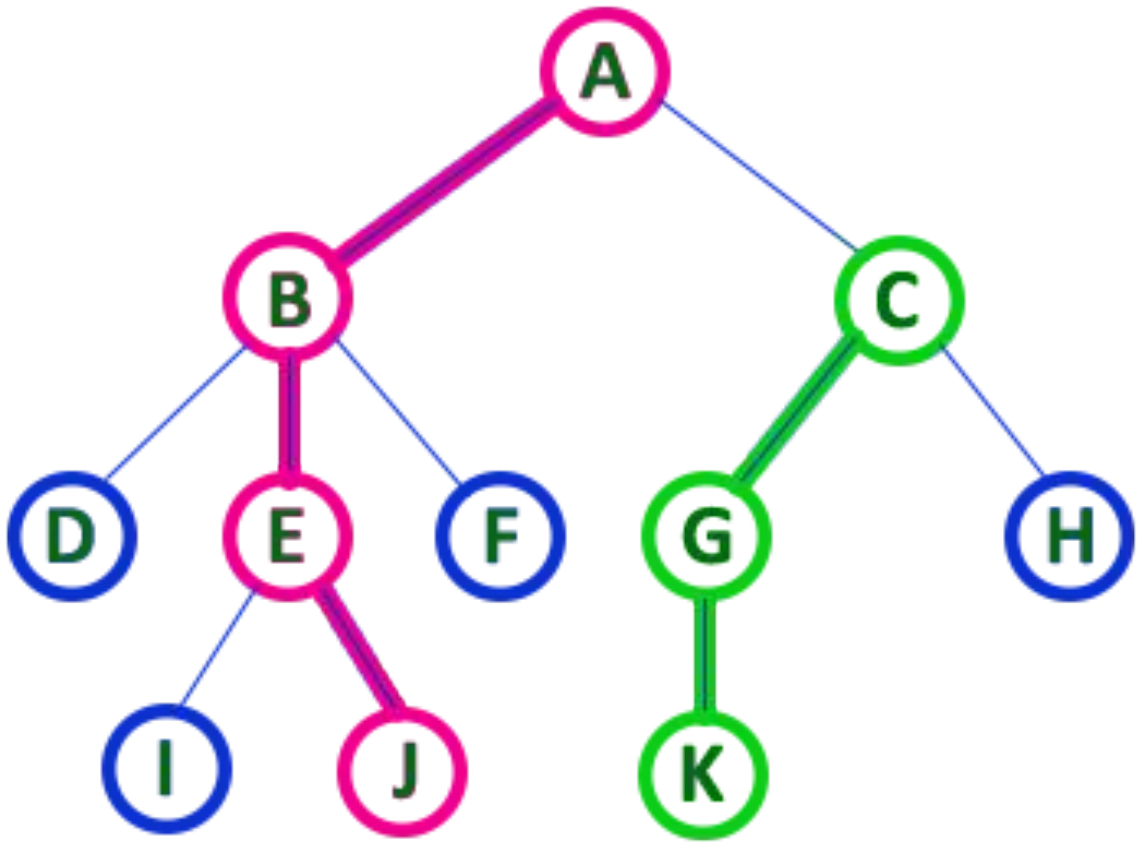
- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.



Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.

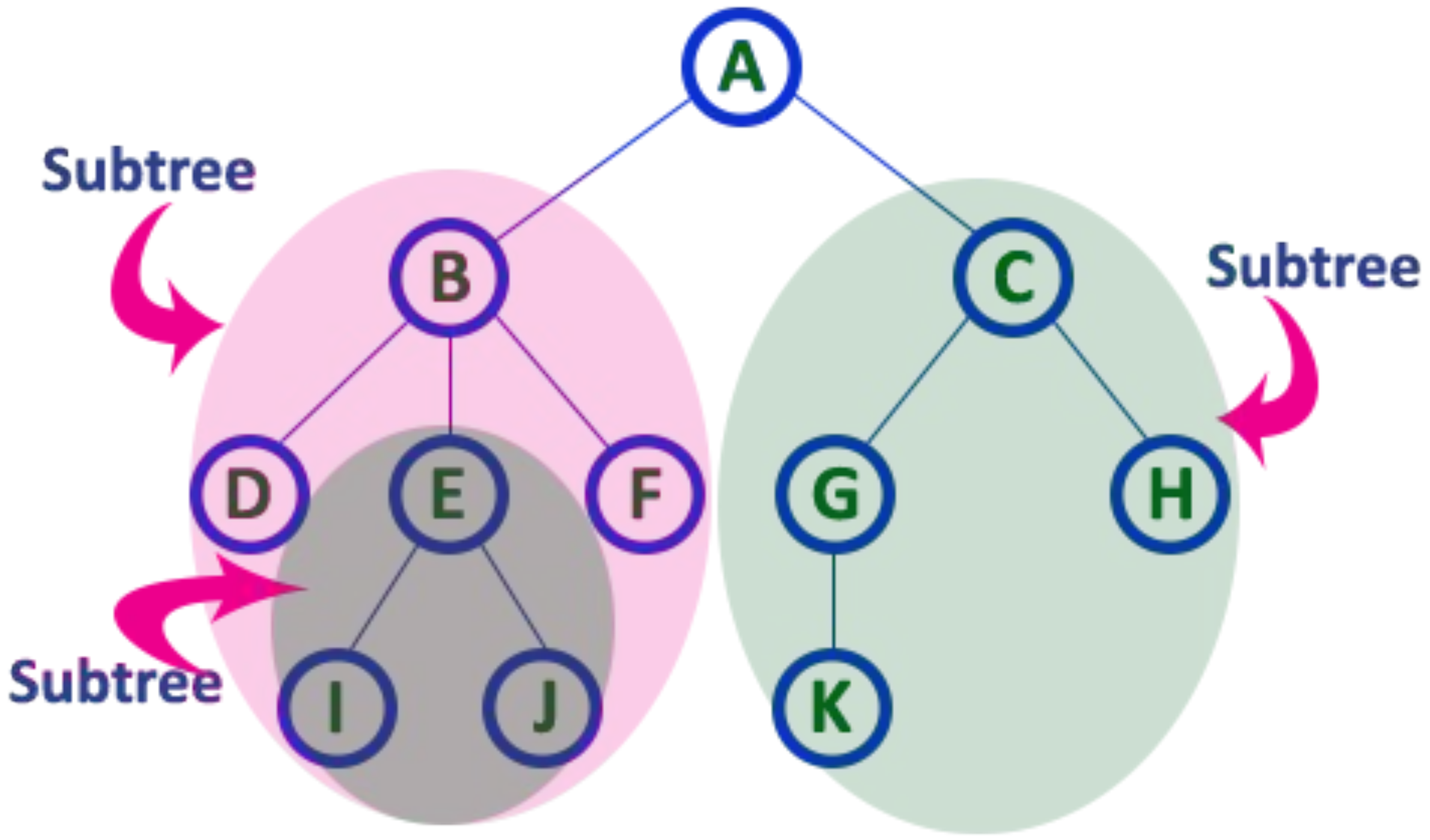
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.



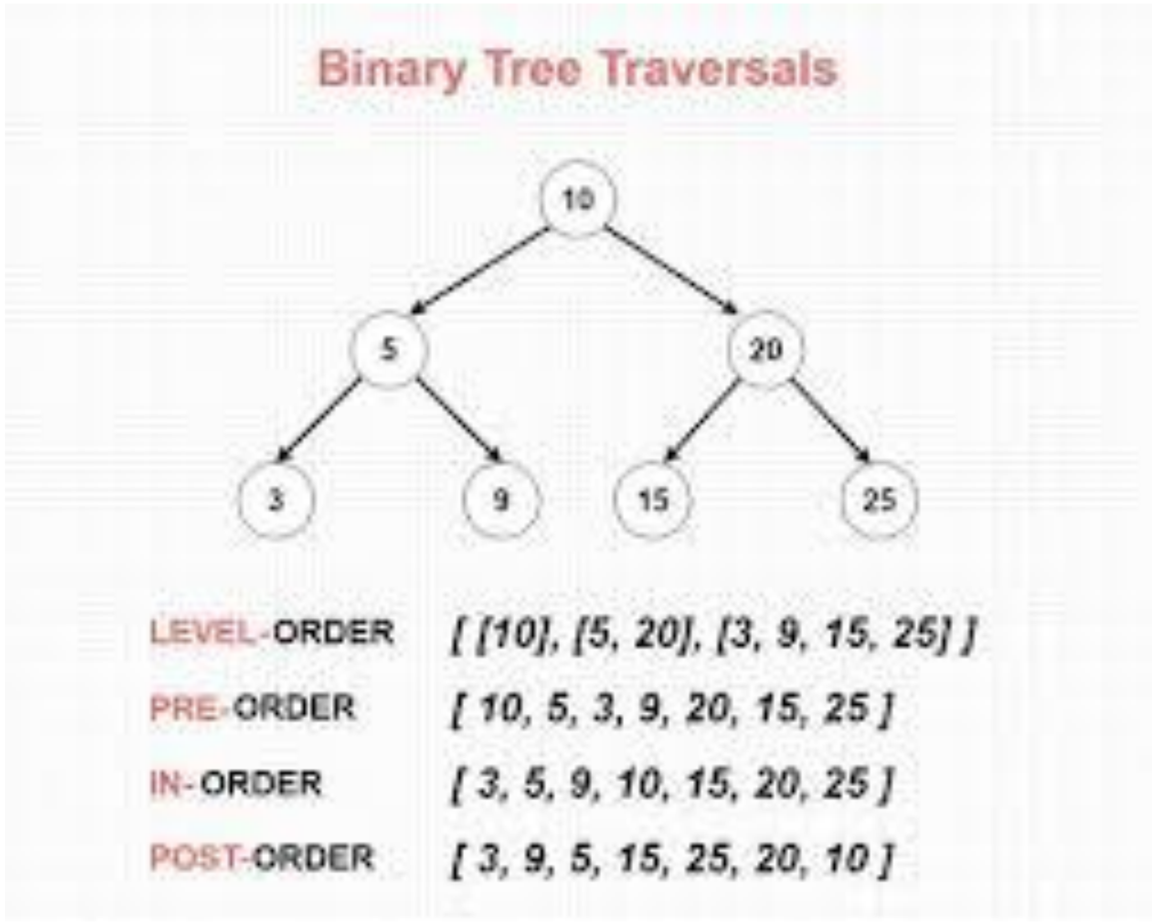
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K


A
Subtree
B
C
Subtree
D
E
F
G
H
Subtree
I
J
K

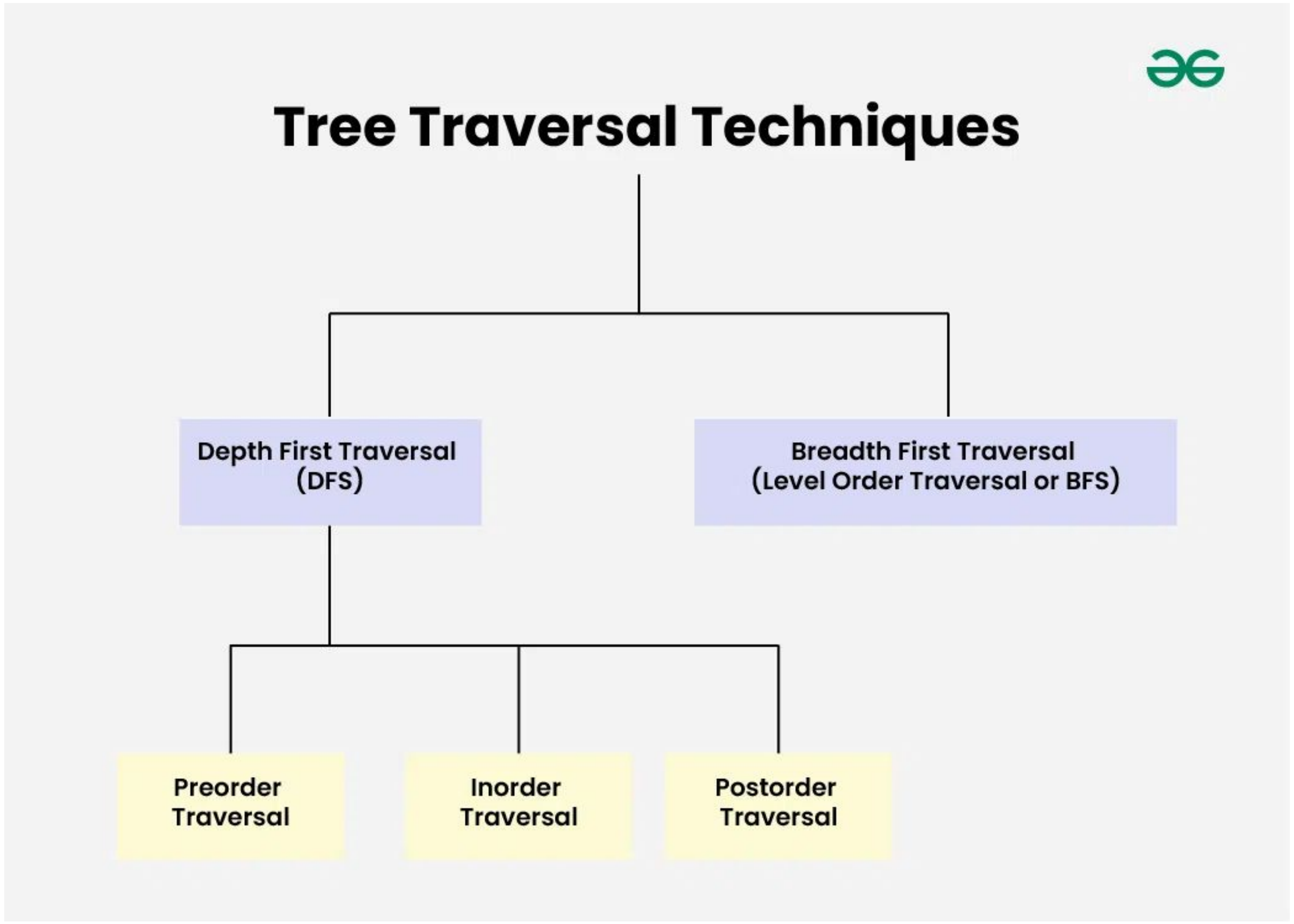
Tree Traversal



Tree Traversal

- Tree traversal refers to the process of visiting each node in a tree data structure in a specific order.

Tree Traversal Techniques



Tree Traversal Techniques
Depth First Traversal (DFS)
Breadth First Traversal (Level Order Traversal or BFS)
Preorder Traversal
Inorder Traversal
Postorder Traversal

Tree Traversal Techniques

Breadth-First Search (BFS)

- BFS explores all nodes at the current depth level before moving on to the next level.
- Use Queue DS

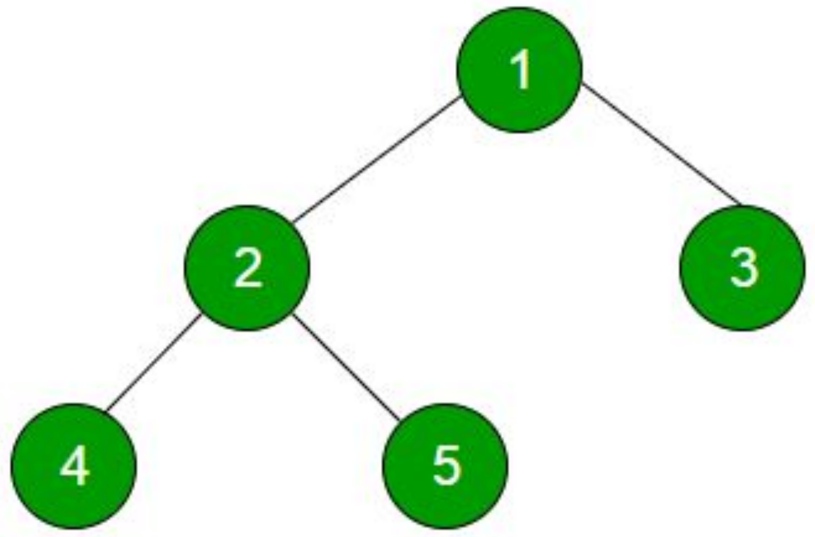
Depth-First Search (DFS)

- DFS explores as far as possible along each branch before backtracking.
- Use Stack Ds

Level Order Traversal (Breadth First Search or BFS)

- **Level Order Traversal** technique is defined as a method to traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level.

Output: 1 2 3 4 5

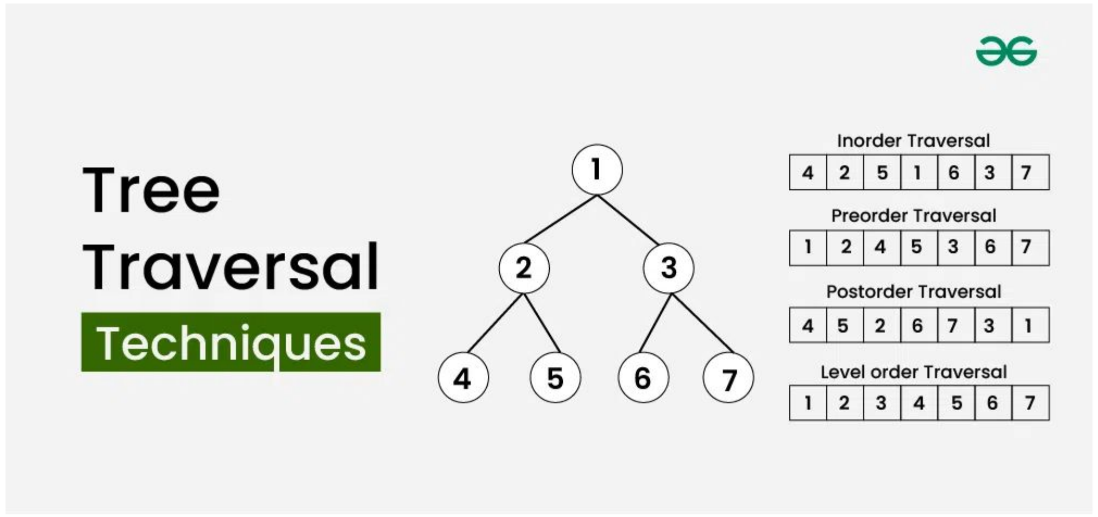


Depth First Search

In-Order Traversal
(Left, Root, Right)

Pre-Order Traversal
(Root, Left, Right)

Post-Order Traversal
(Left, Right, Root)



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

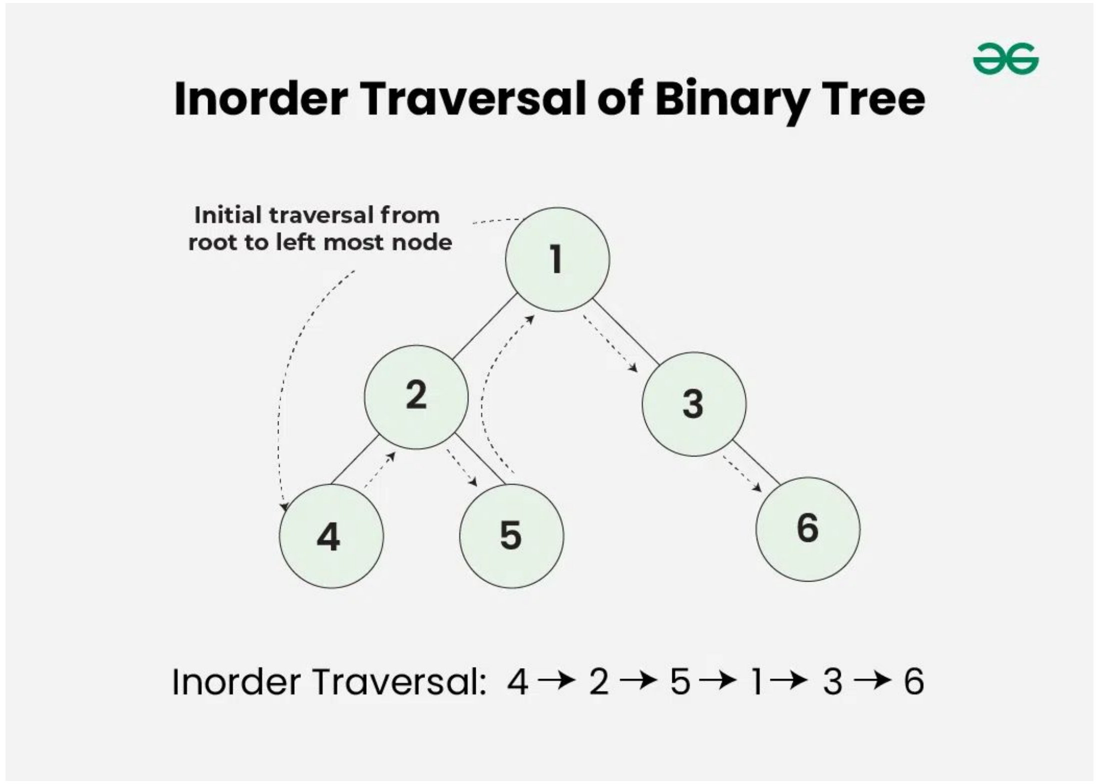
4	5	2	6	7	3	1

Level order Traversal

1	2	3	4	5	6	7

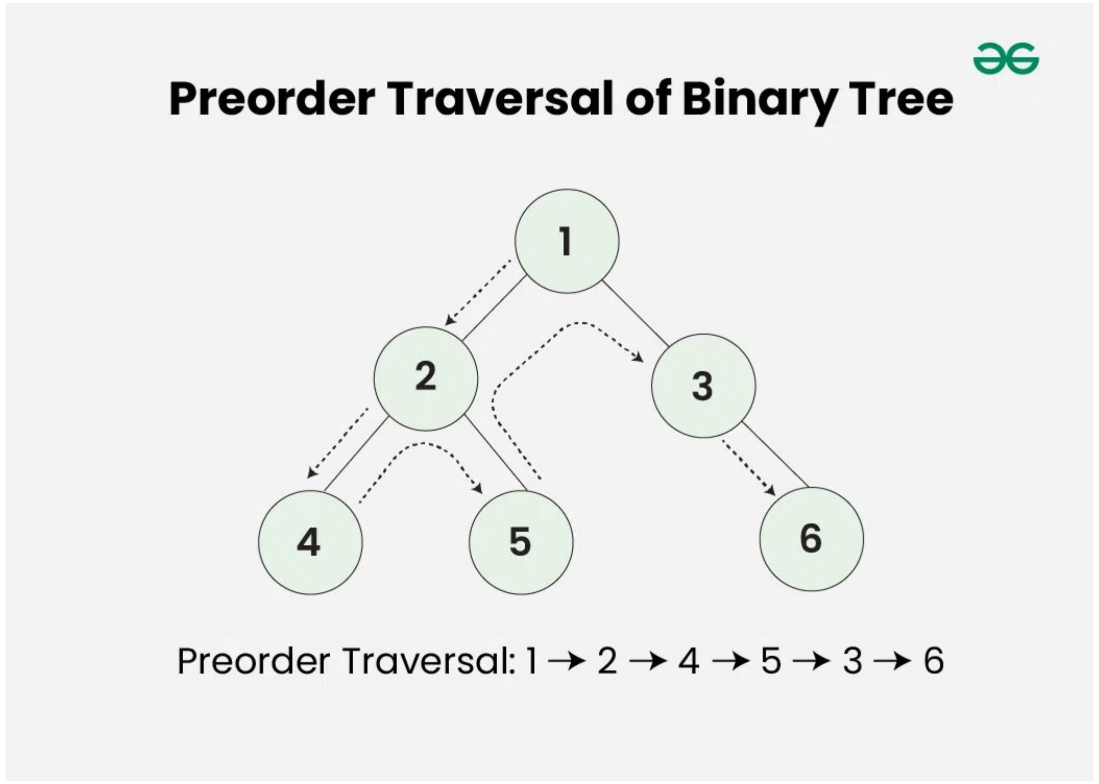
In-Order Traversal (Left, Root, Right)

- Inorder traversal visits the node in the order: **Left -> Root -> Right**


Inorder Traversal of Binary Tree
Initial traversal from root to left most node
1
2
3
4
5
6
Inorder Traversal: 4 → 2 → 5 → 1 → 3 → 6

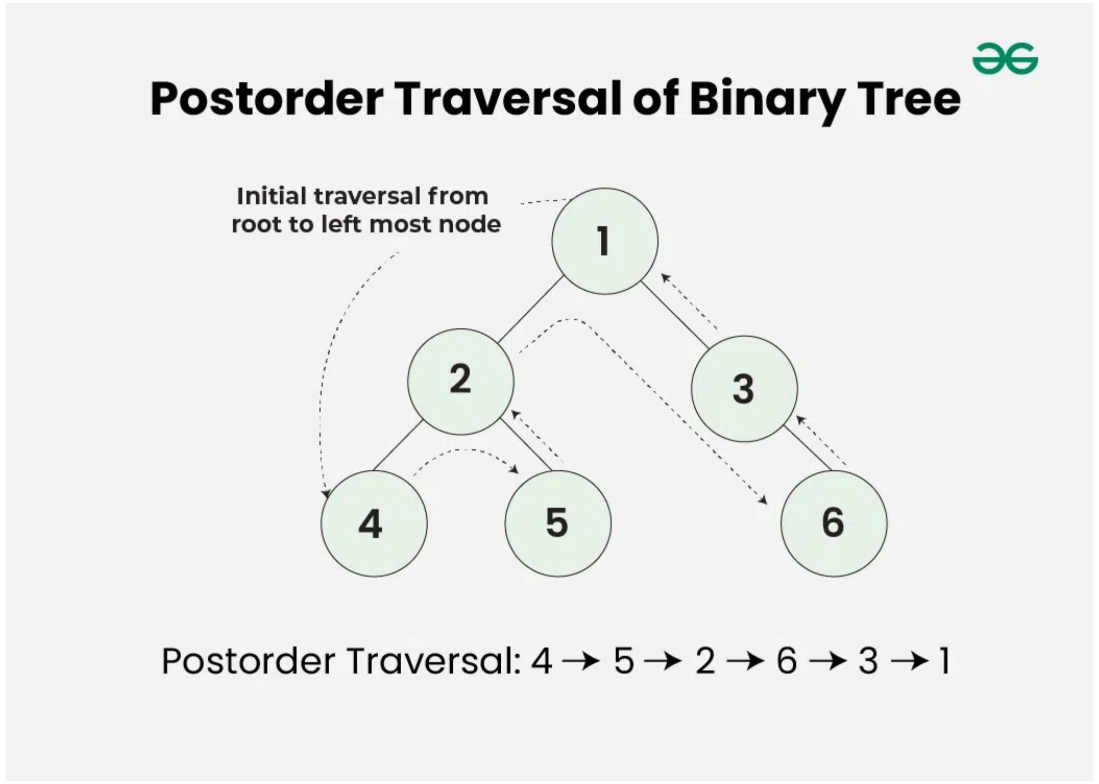
Pre-Order Traversal (Root, Left, Right)

- Preorder traversal visits the node in the order: **Root -> Left -> Right**


Preorder Traversal of Binary Tree
1
2
3
4
5
6
Preorder Traversal: 1 → 2 → 4 → 5 → 3 → 6

Post-Order Traversal (Left, Right, Root)

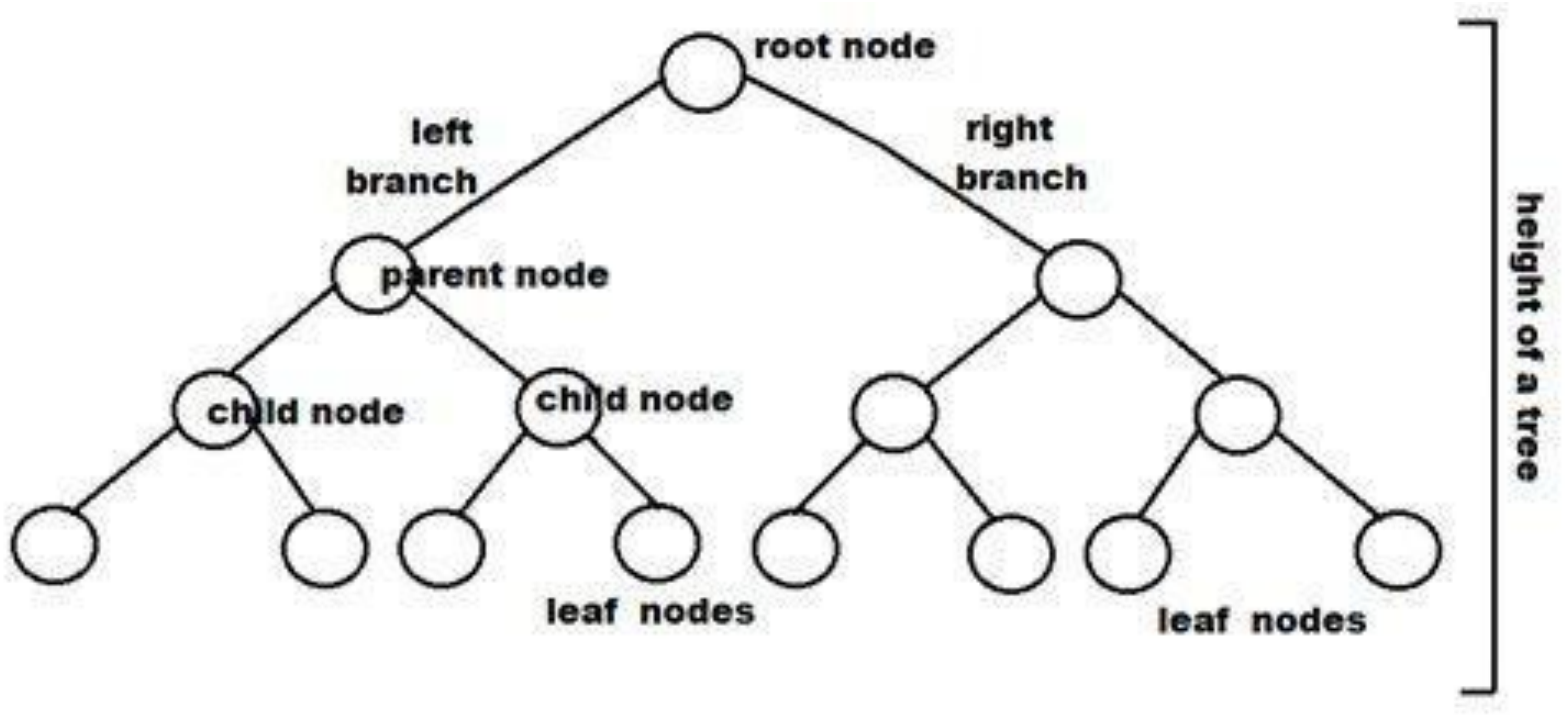
- Pos-torder traversal visits the node in the order: **Left -> Right -> Root**


Postorder Traversal of Binary Tree
Initial traversal from root to left most node
1
2
3
4
5
6
Postorder Traversal: 4 → 5 → 2 → 6 → 3 → 1

Binary Tree

Binary Tree

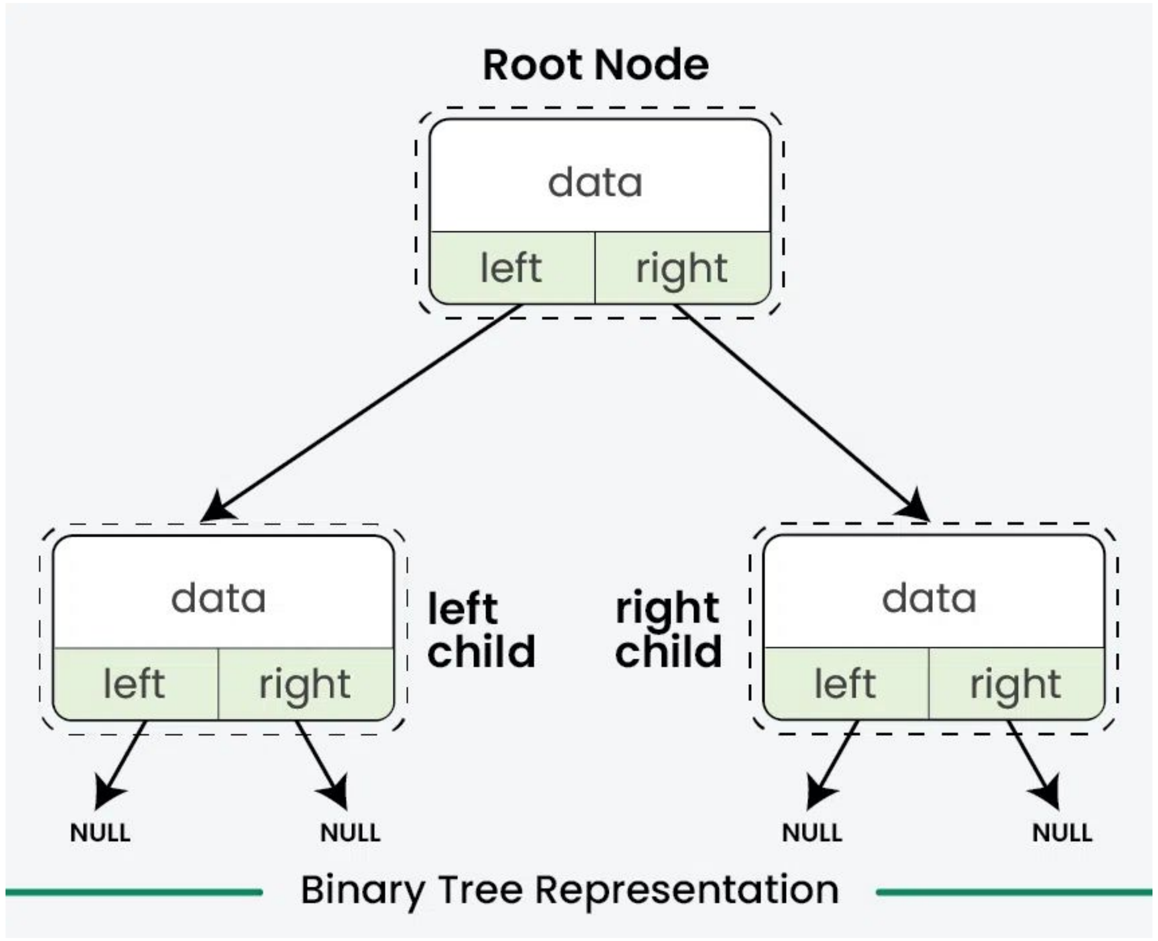
- A **binary tree** is a tree-type non-linear data structure with a maximum of two children for each parent.



Representation of Binary Tree

Each node in a Binary Tree has three parts:

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

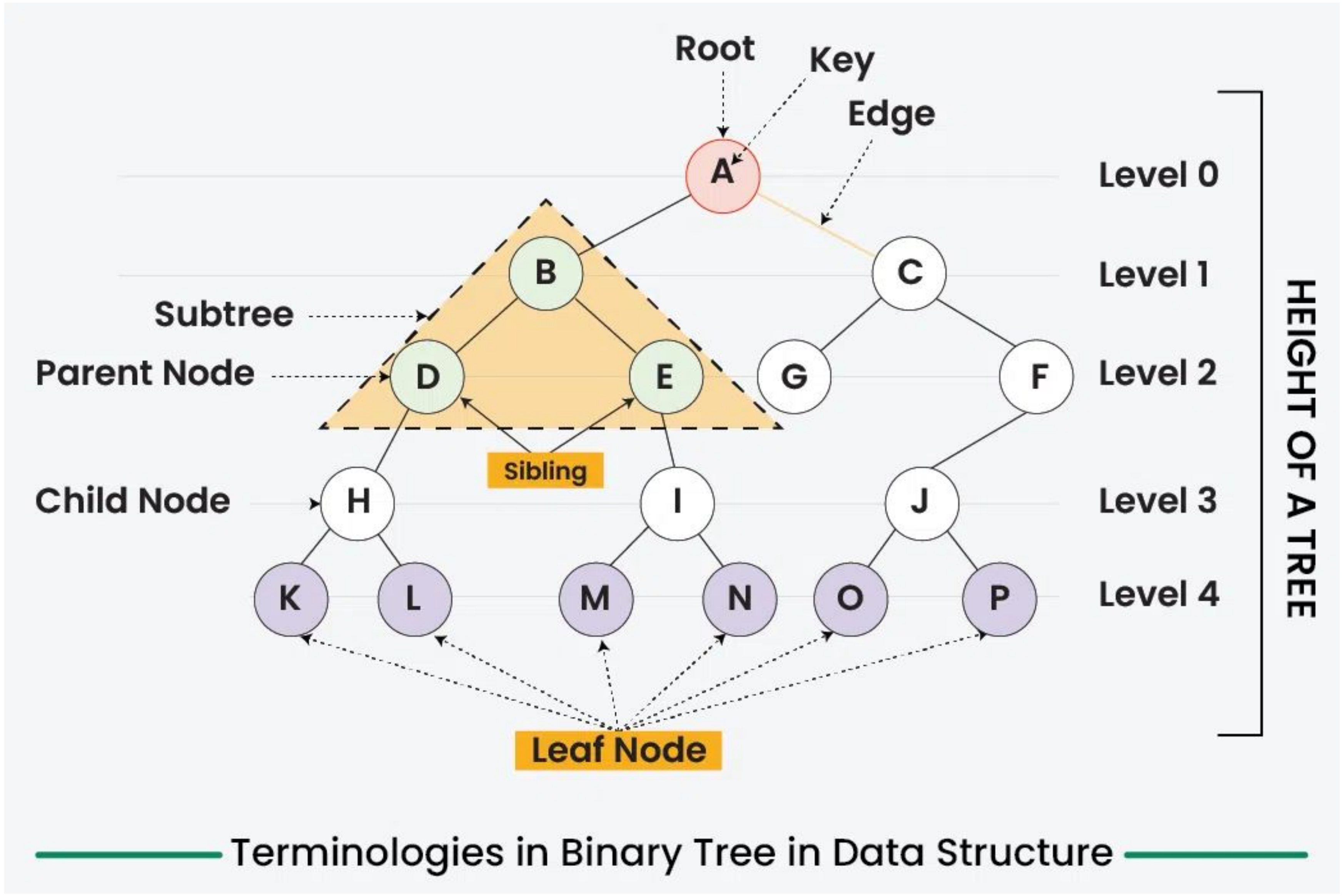


Create/Declare a Node of a Binary Tree

```
class Node {
int key;
Node left, right;

public Node(int item) {
 key = item;
left = right = null;
}
 }
```

Terminologies in Binary Tree



Terminologies in Binary Tree in Data Structure

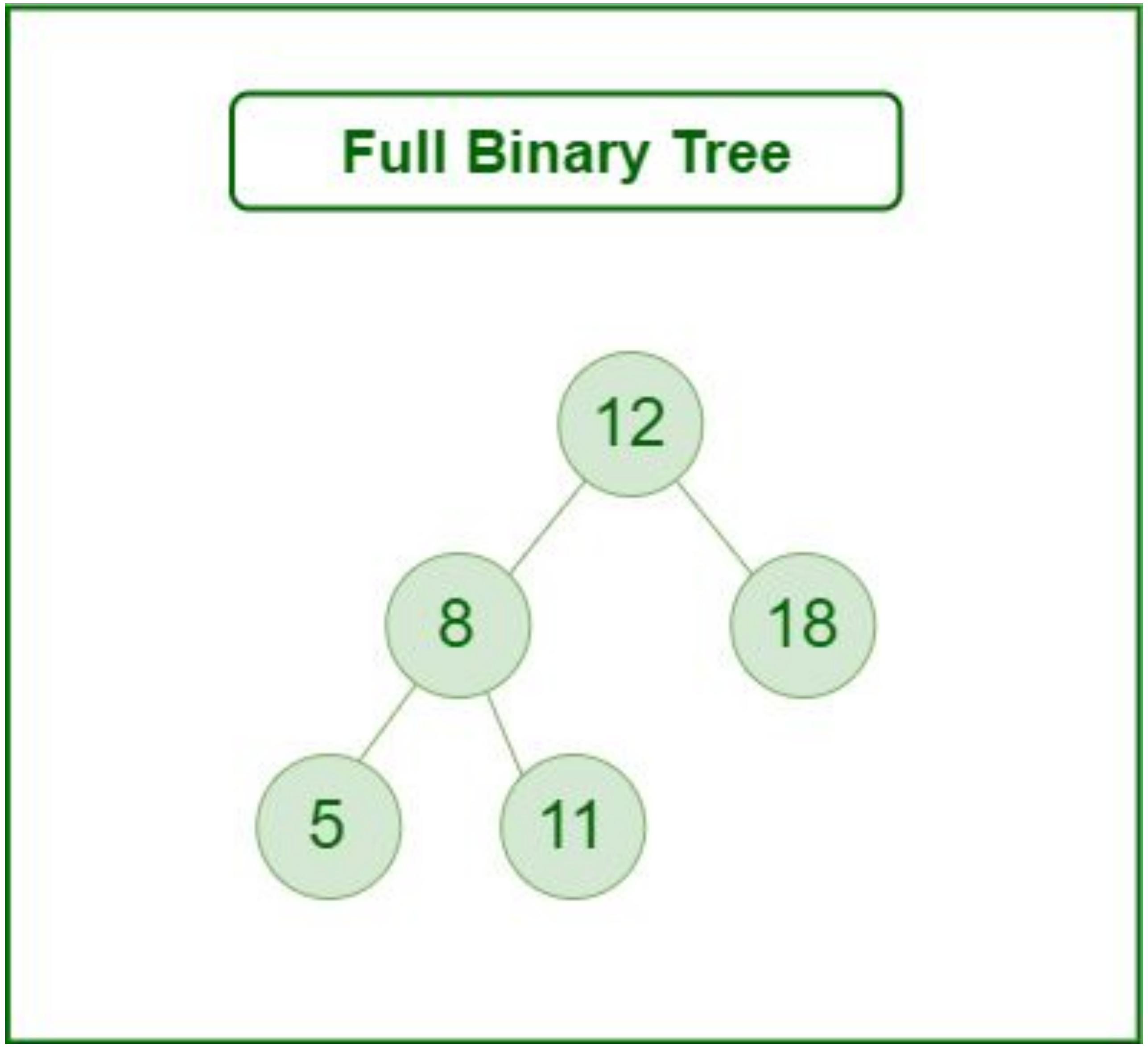
Why use binary trees

- Efficiency in Search Operations
  - Binary Search Tree (BST)
- Simplified Traversals
  - Binary trees have a **clear structure** with a left and right child for each node.
  - allows for efficient tree traversals (in-order, pre-order, post-order, and level-order)
- Memory Efficiency
  - In binary trees, each node typically contains two pointers to its children whereas general trees may require a more complex structure
- Balanced Tree Structures for Sorting
  - AVL and Red Black Tree for efficient **sorting algorithms**

Types of Binary Tree

Full Binary Tree

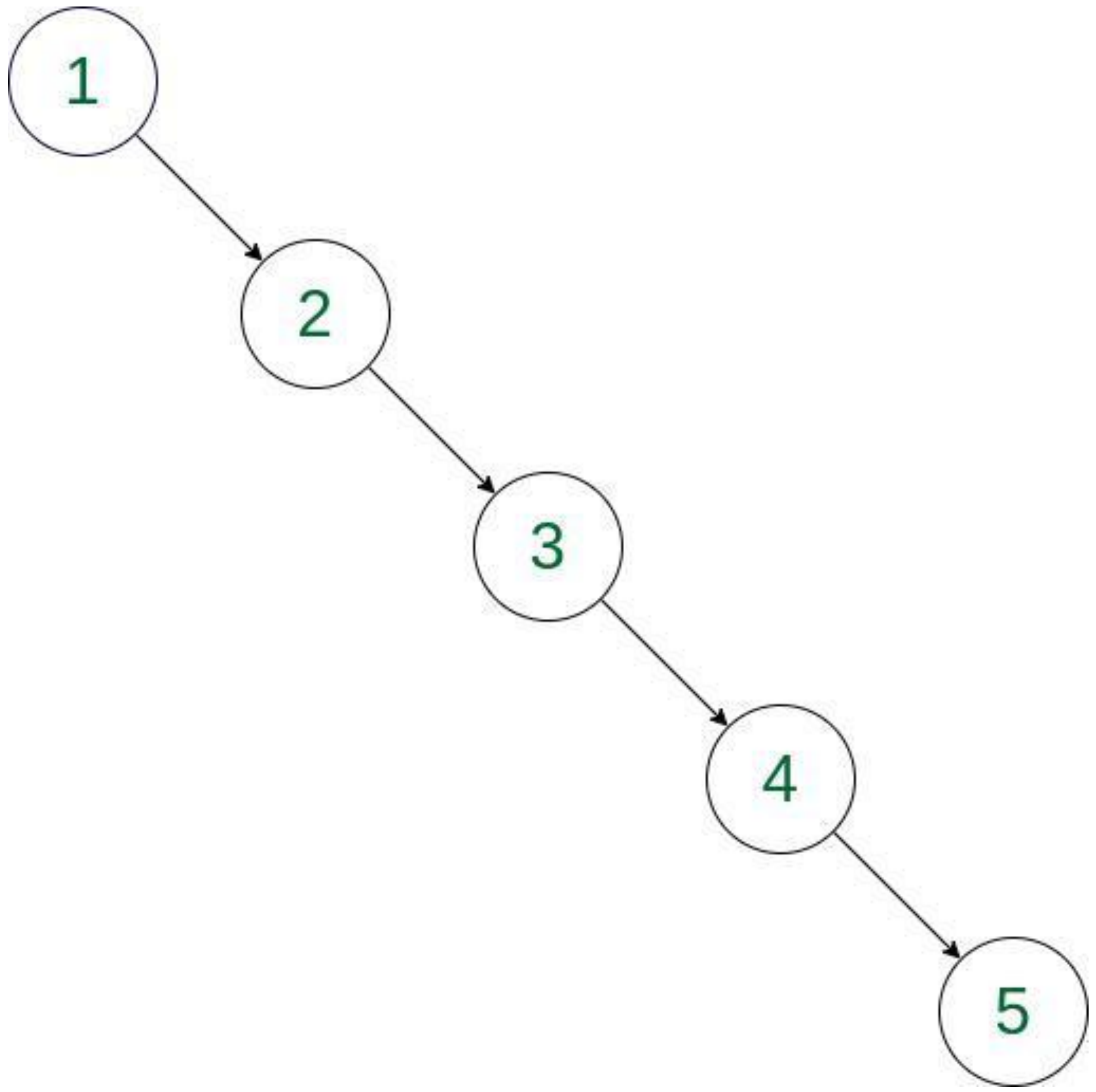
- *A full binary tree is a binary tree with either zero or two child nodes for each node.*



Types of Binary Tree

<u>Degenerate Binary Tree</u>

*Every **non-leaf node** has just **one** child in a binary tree known as a **Degenerate Binary tree**. The tree effectively transforms into a linked list as a result, with each node linking to its **single** child.*



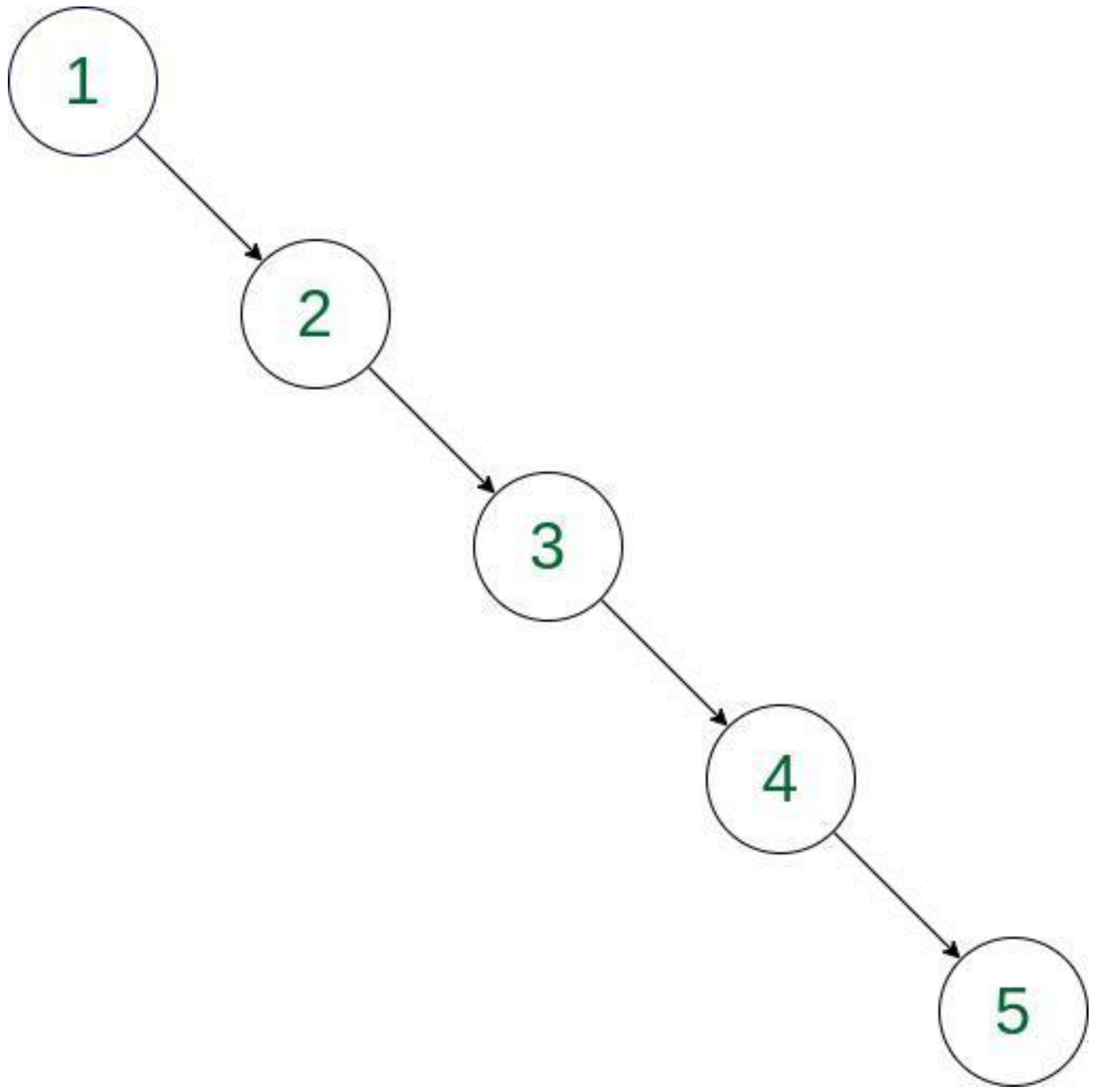
Types of Binary Tree

<u>Degenerate Binary Tree</u>

Degenerate Binary tree is of two types:

- **Left-skewed Tree:** If all the nodes in the degenerate tree have only a left child.

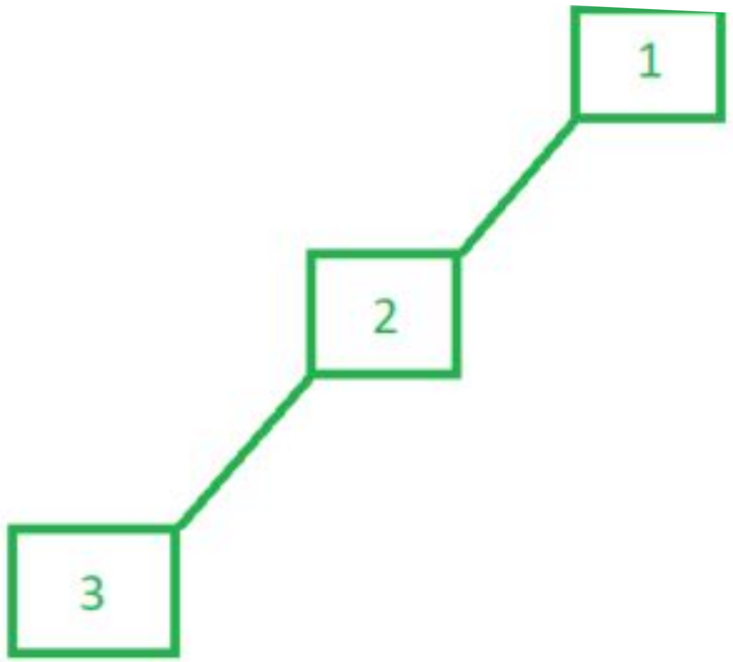
- **Right-skewed Tree:** If all the nodes in the degenerate tree have only a right child.

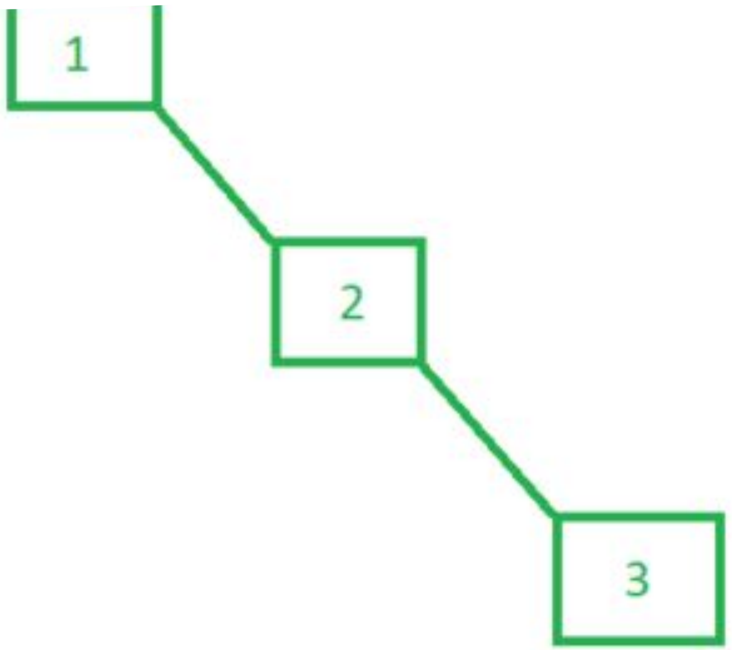


Types of Binary Tree

Skewed Binary Tree

A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child.

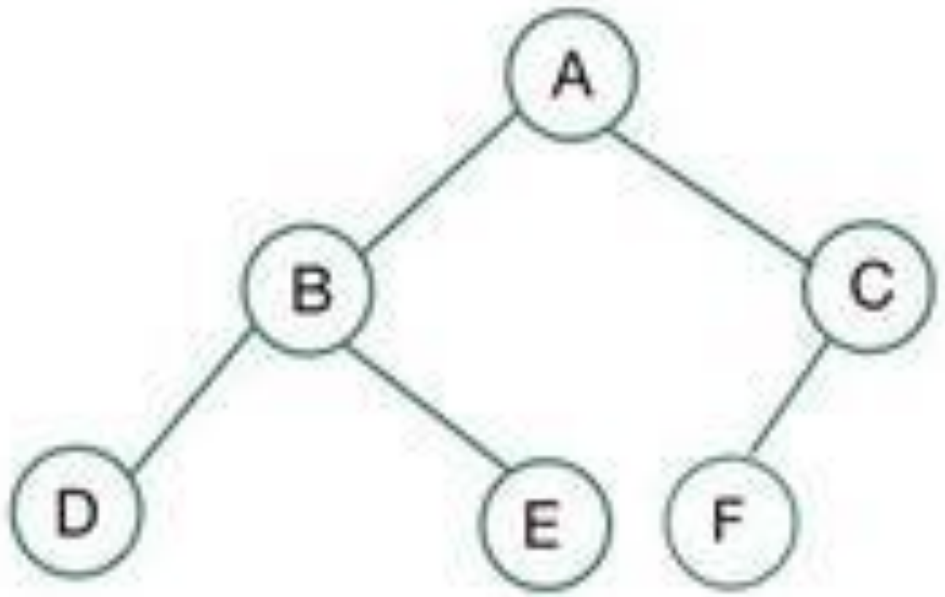




Types of Binary Tree

Complete Binary Tree

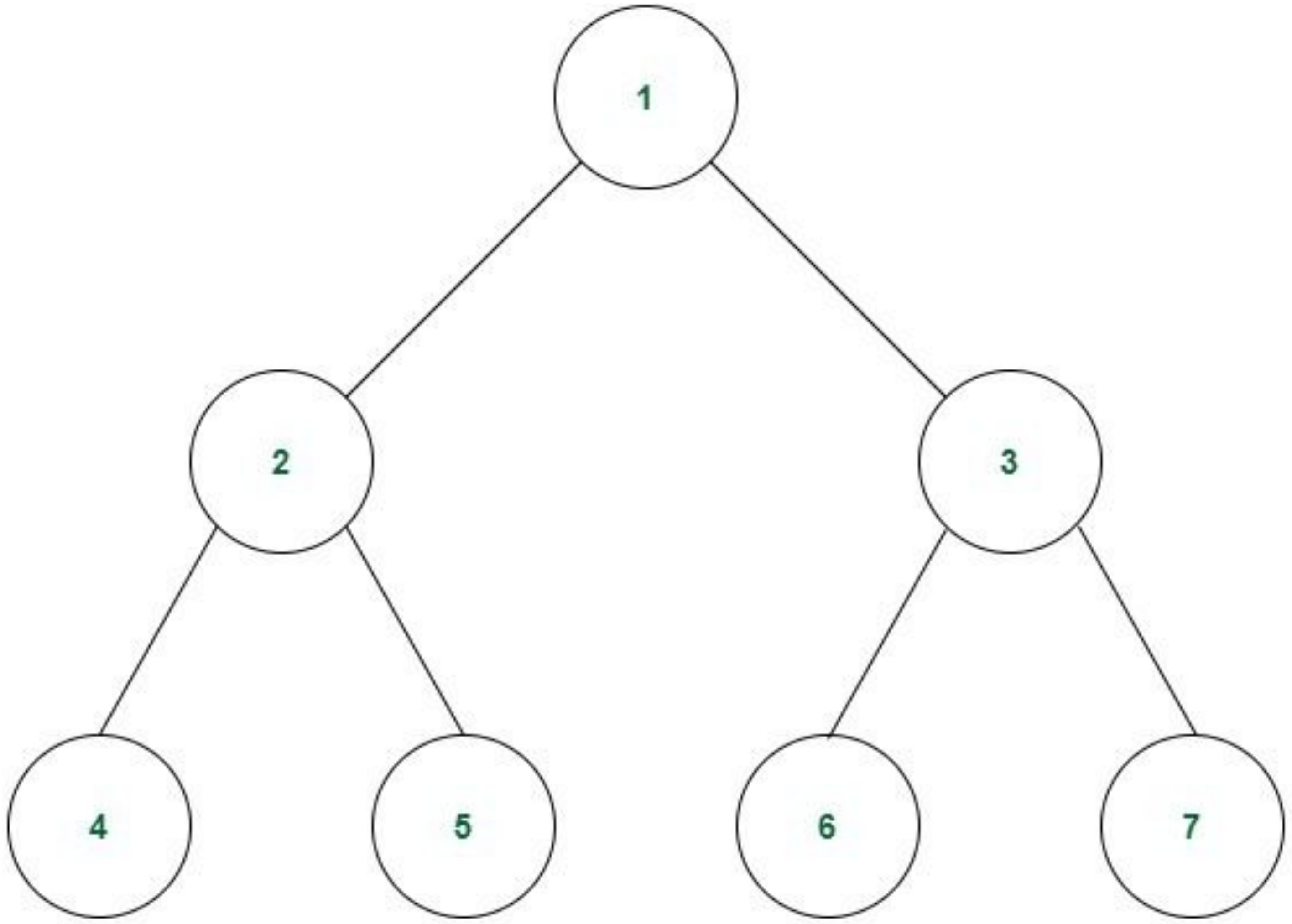
A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.



Types of Binary Tree

Perfect Binary Tree

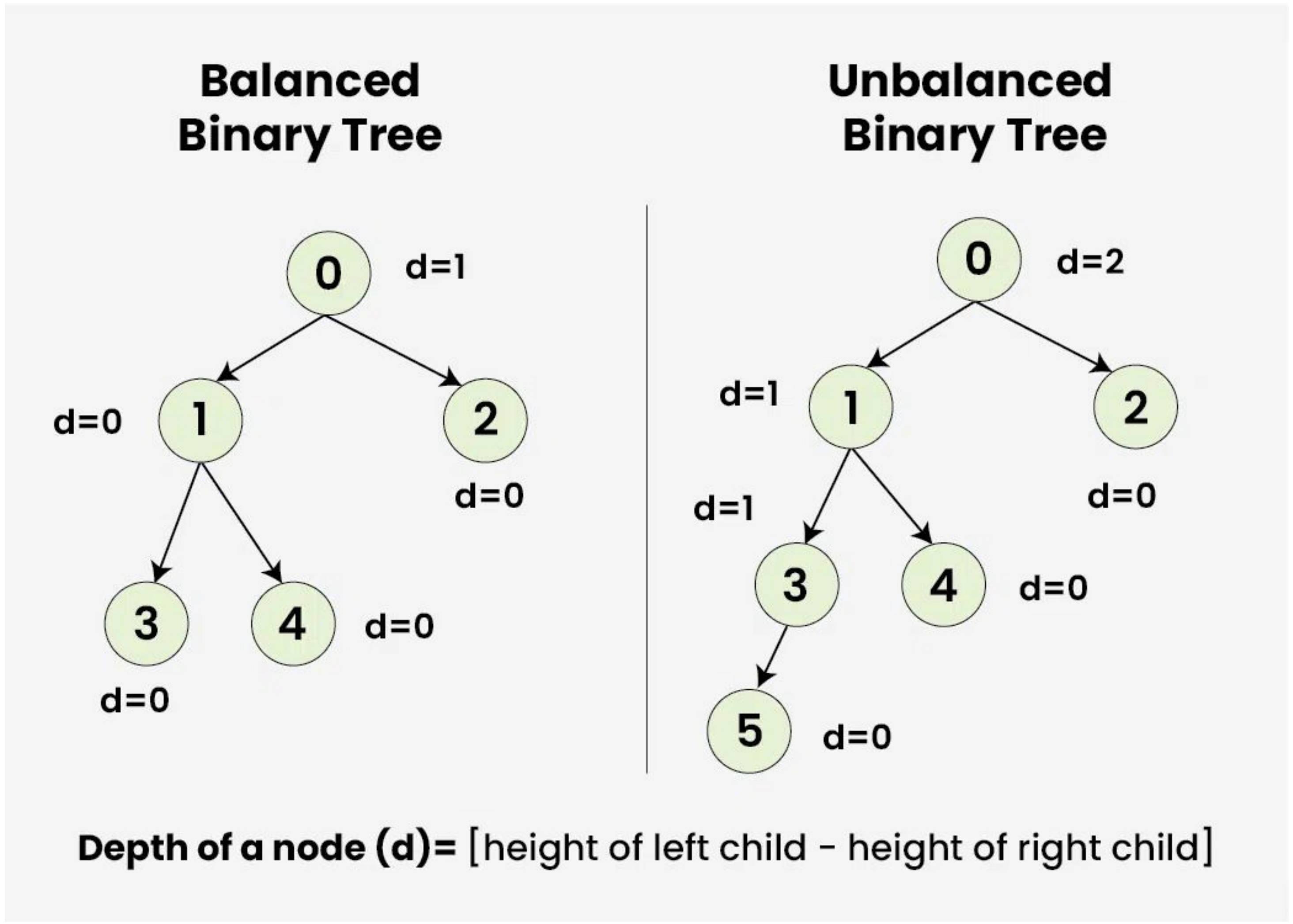
A **perfect binary tree** is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children.



Types of Binary Tree

Balanced Binary Tree

- It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Types of Binary Tree

Binary Search Tree

- A **Binary Search Tree (or BST)** is a data structure used in computer science for organizing and storing data in a sorted manner.
- Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child
- **left** child containing values less than the parent node and the **right** child containing values greater than the parent node.

Types of Binary Tree

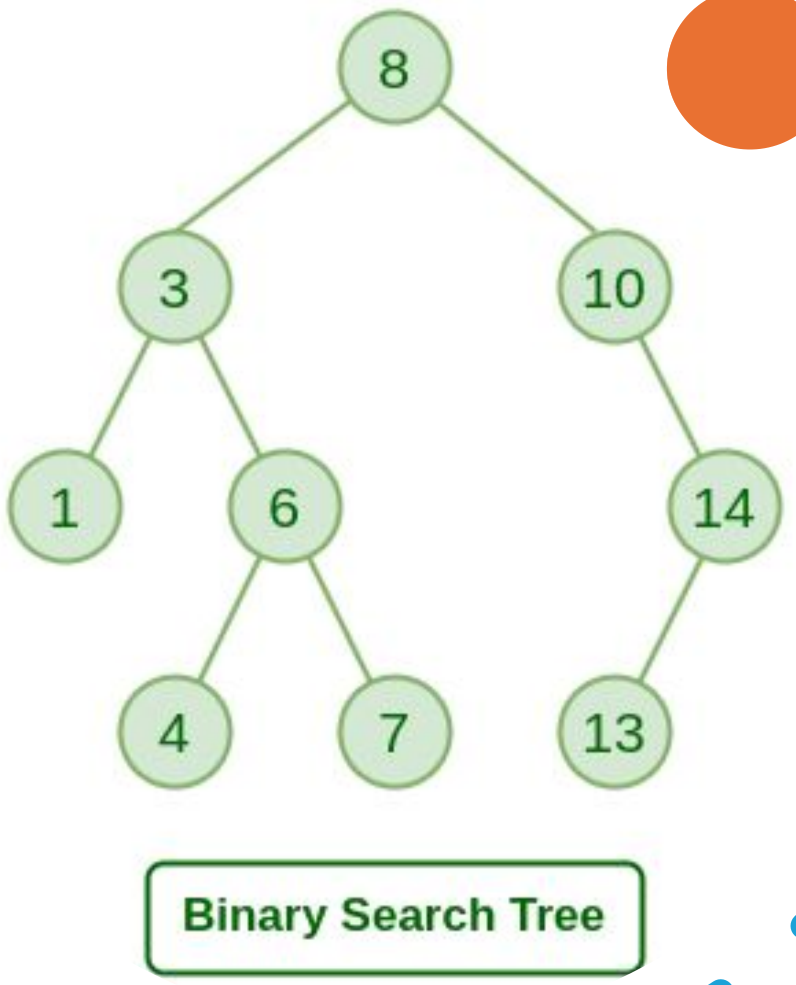
Binary Search Tree

- A **Binary Search Tree (or BST)** is a data structure used in computer science for organizing and storing data in a sorted manner.
- Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child
- **left** child containing values less than the parent node and the **right** child containing values greater than the parent node.

Types of Binary Tree

Binary Search Tree

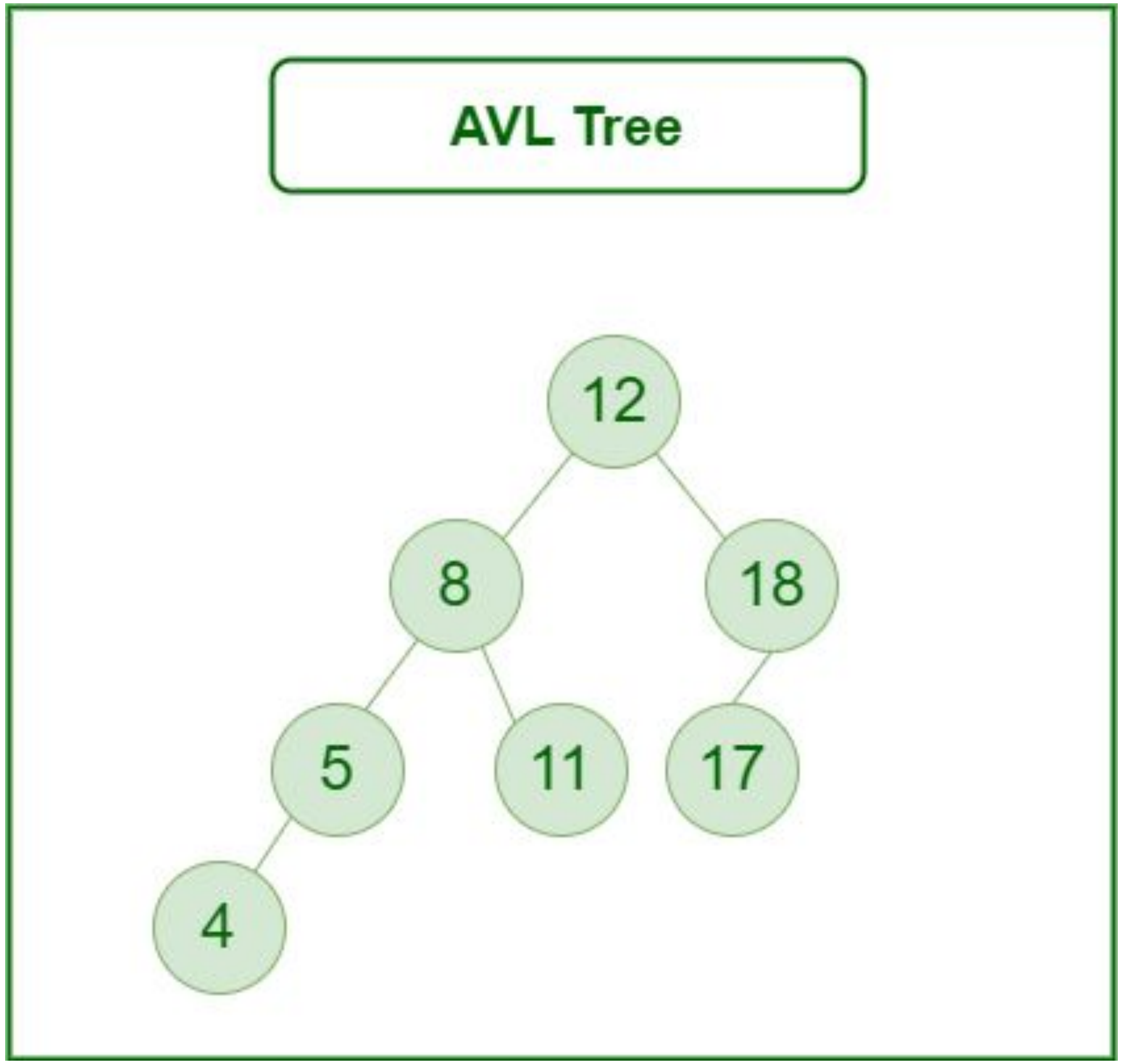
- This hierarchical structure allows for efficient **searching**, **insertion**, and **deletion** operations on the data stored in the tree.



Types of Binary Tree

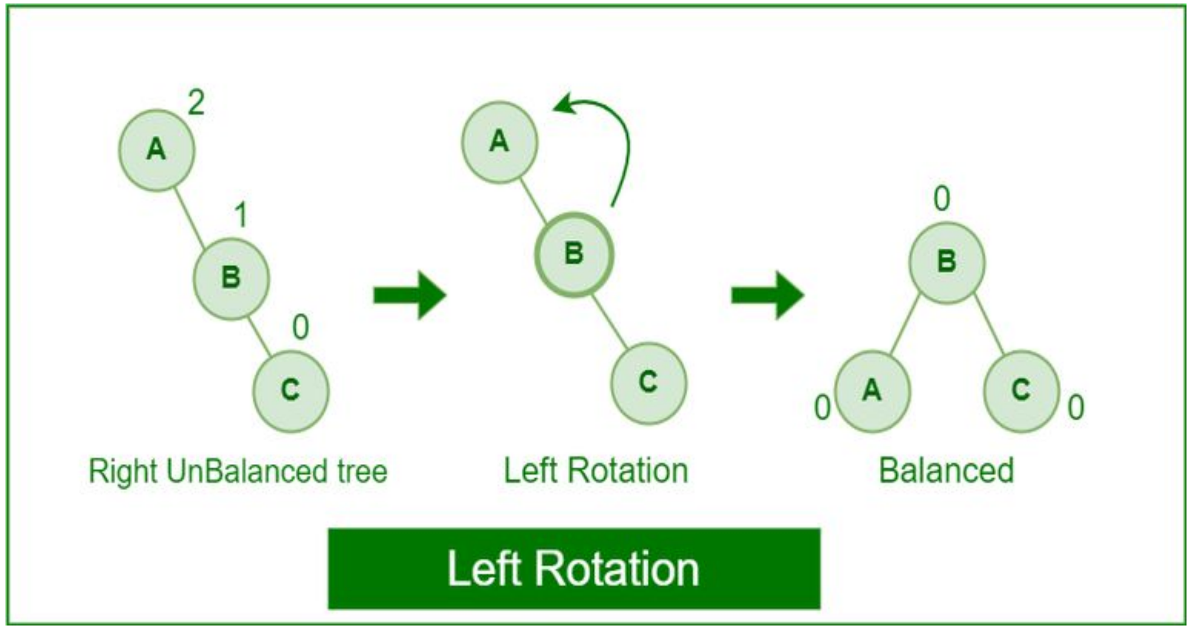
- *AVL Tree*

*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.*



Types of Binary Tree

- **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**



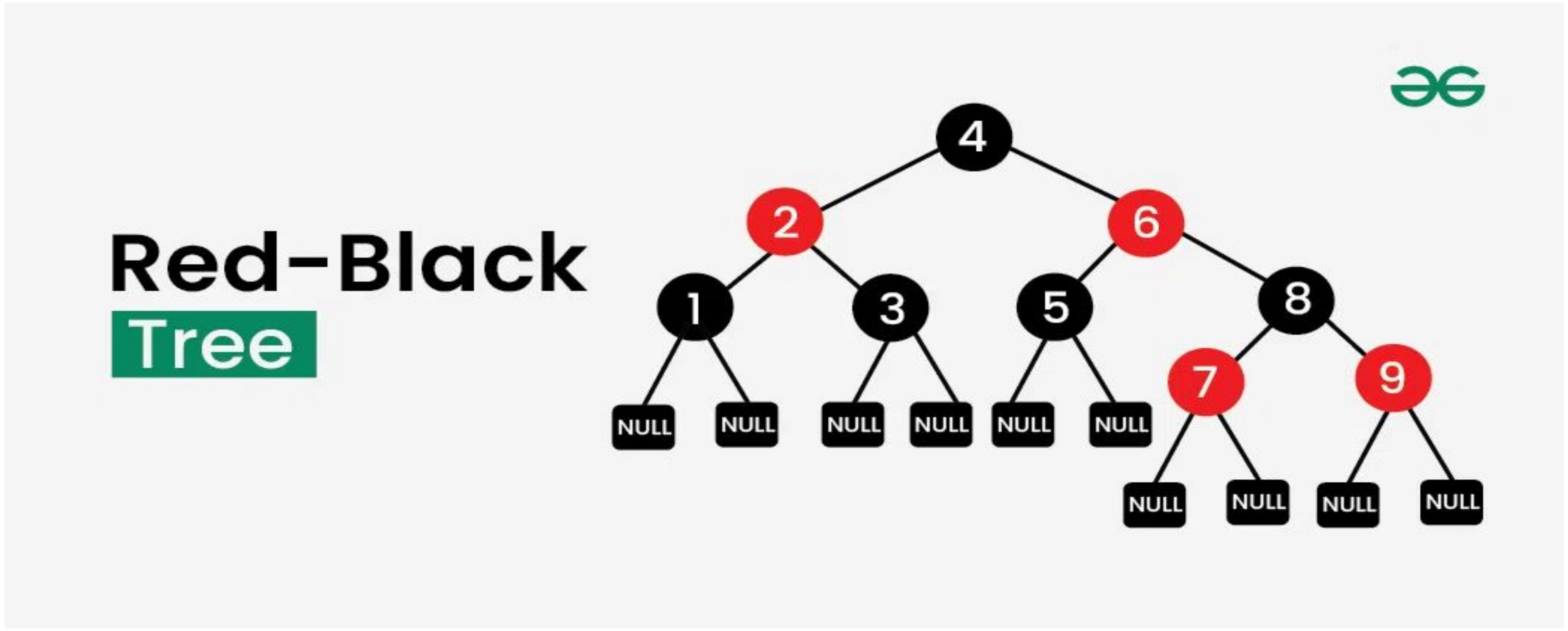
Types of Binary Tree

Red-Black Tree

- A **Red-Black Tree** is a self-balancing binary search tree where each node has an additional attribute: a color, which can be either **red** or **black**.
- The primary objective of these trees is to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation.

Types of Binary Tree

Red-Black Tree



Types of Binary Tree

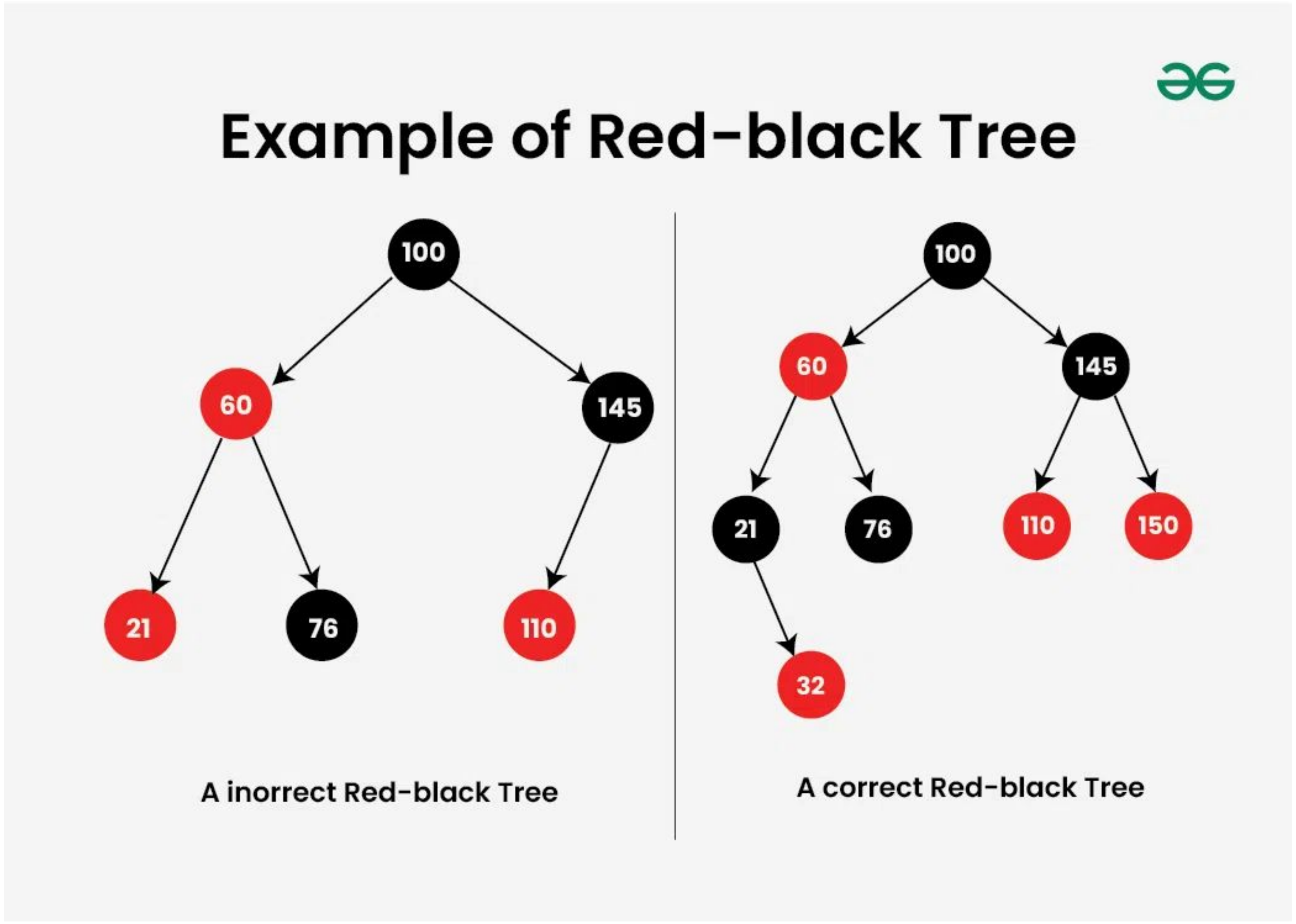
Red-Black Tree

Properties of Red-Black Trees: A Red-Black Tree have the following properties:

- **Node Color**: Each node is either red or **black**.
- **Root Property**: The root of the tree is always **black**.
- **Red Property**: Red nodes cannot have red children (no two consecutive red nodes on any path).
- **Black Property**: Every path from a node to its descendant null nodes (leaves) has the same number of **black** nodes.
- **Leaf Property**: All leaves (NIL nodes) are **black**.

Types of Binary Tree

Red Black Tree

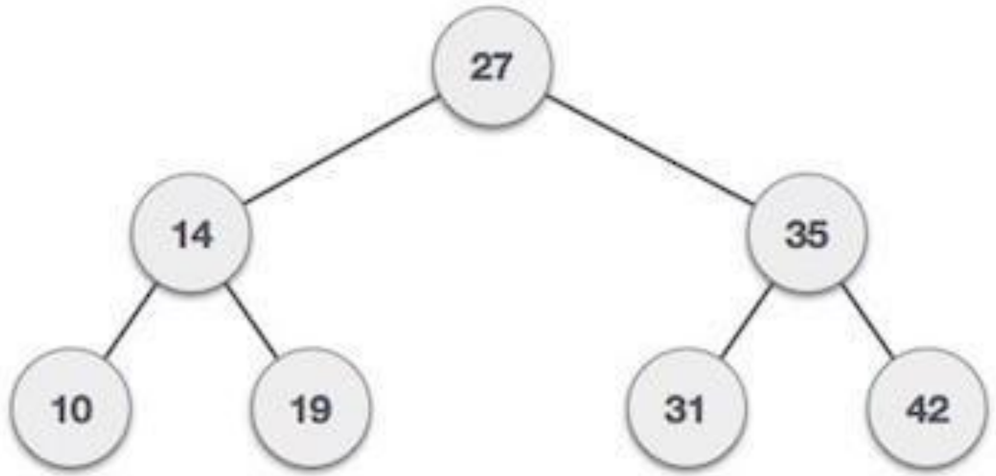


Types of Binary Tree

We will discuss the following Binary tree types in detail:

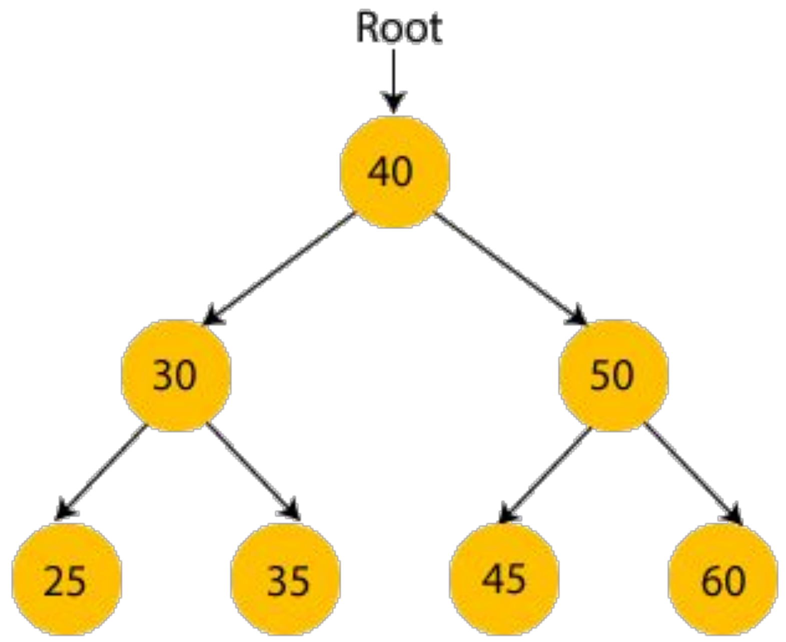
- Binary Search Tree
- AVL Tree
- Red Black Tree

Binary Search Tree



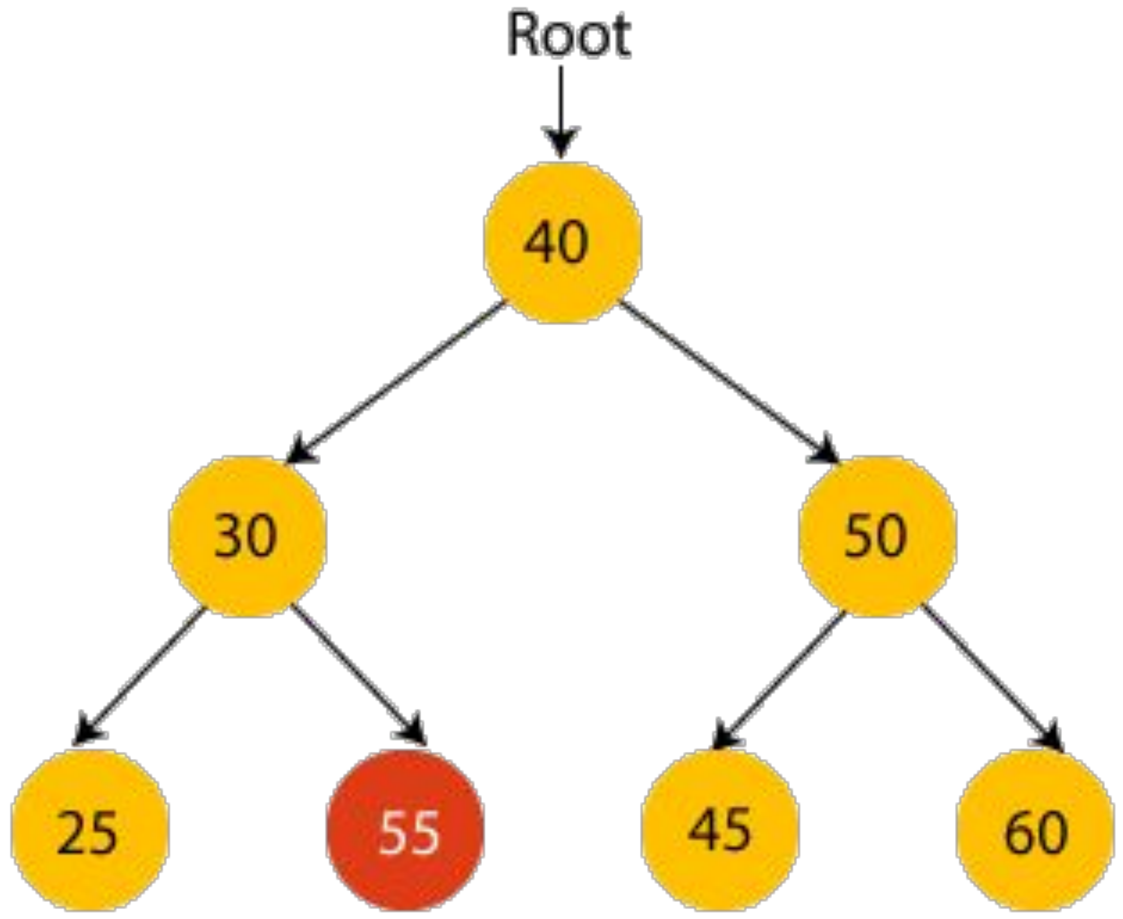
Binary Search Tree

- A binary search tree follows some order to arrange the elements.
- In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.
- This rule is applied recursively to the left and right subtrees of the root.



Binary Search Tree

- Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



Binary Search Tree

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

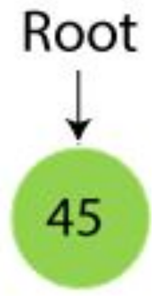
Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.

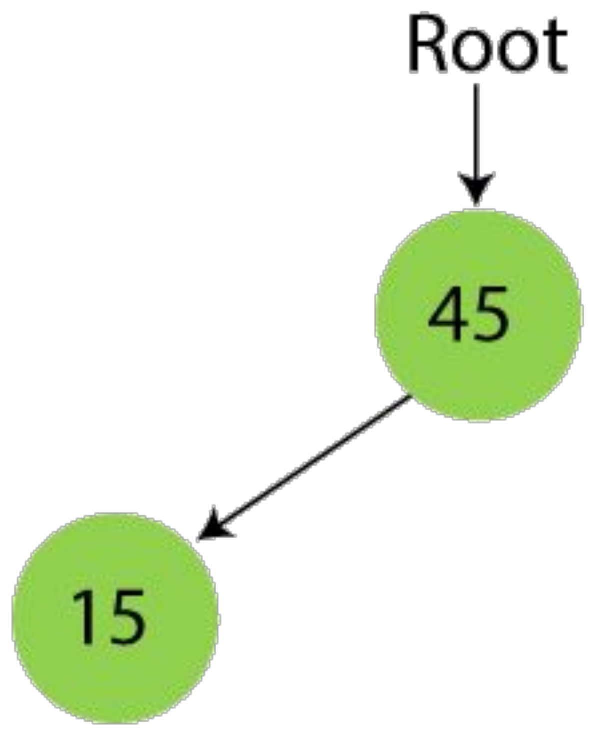
Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

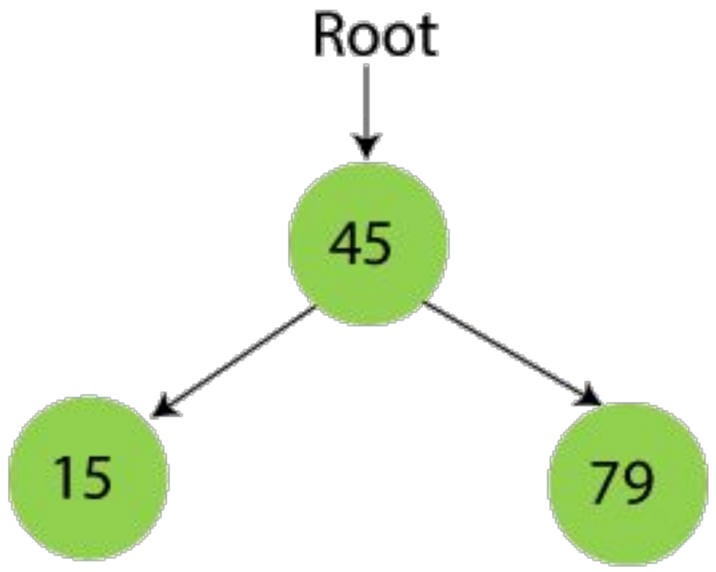
45, 15, 79, 90, 10, 55, 12, 20, 50



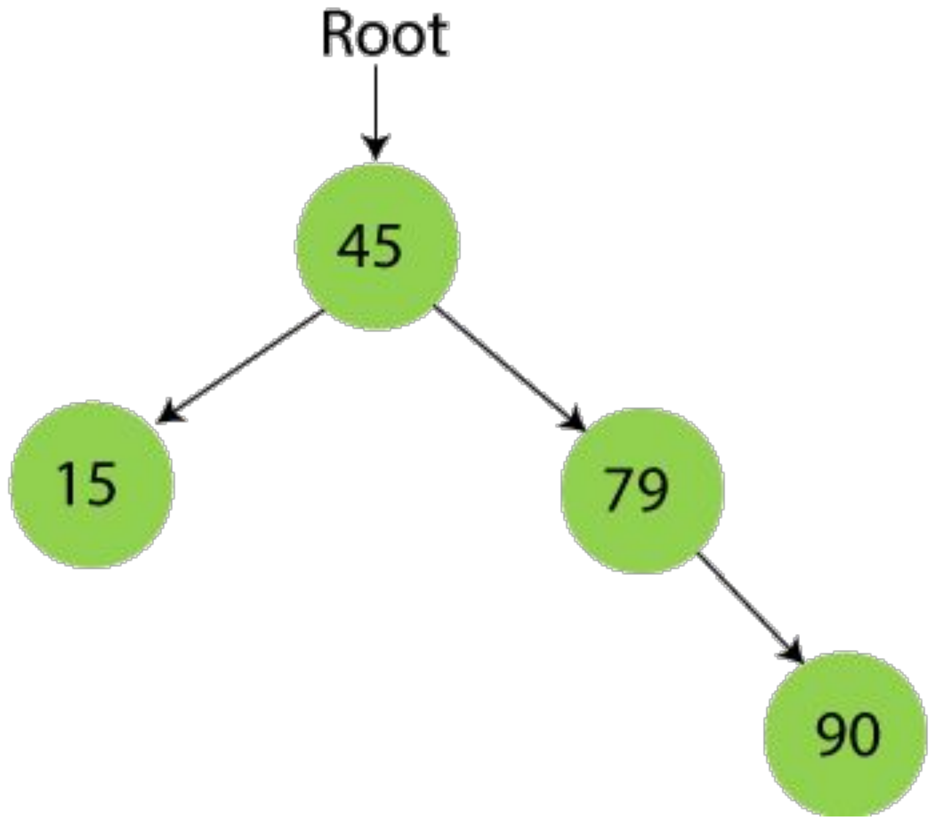
45, 15, 79, 90, 10, 55, 12, 20, 50



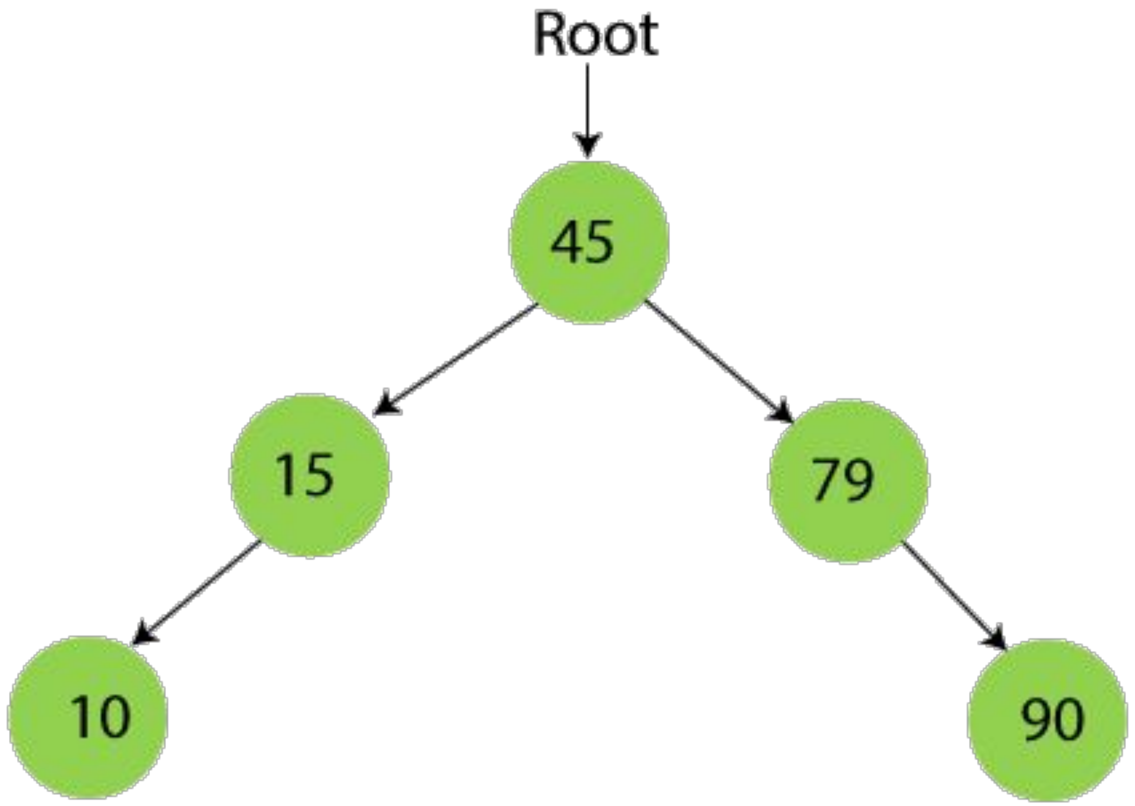
45, 15, 79, 90, 10, 55, 12, 20, 50



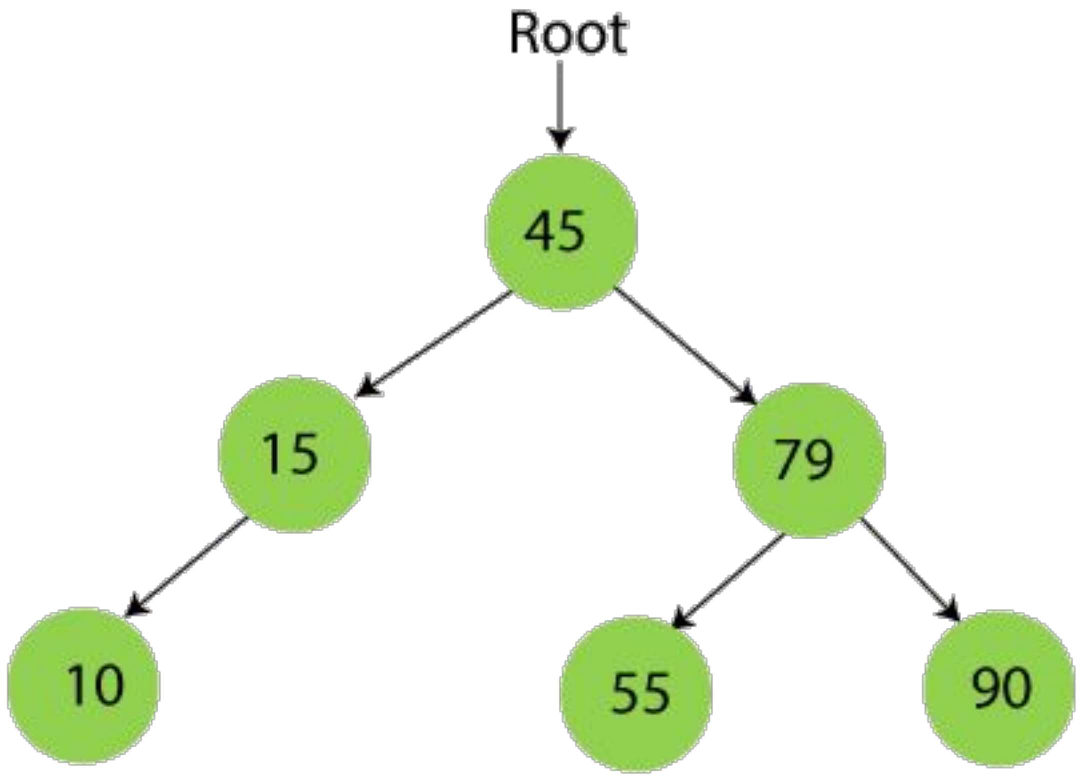
45, 15, 79, 90, 10, 55, 12, 20, 50



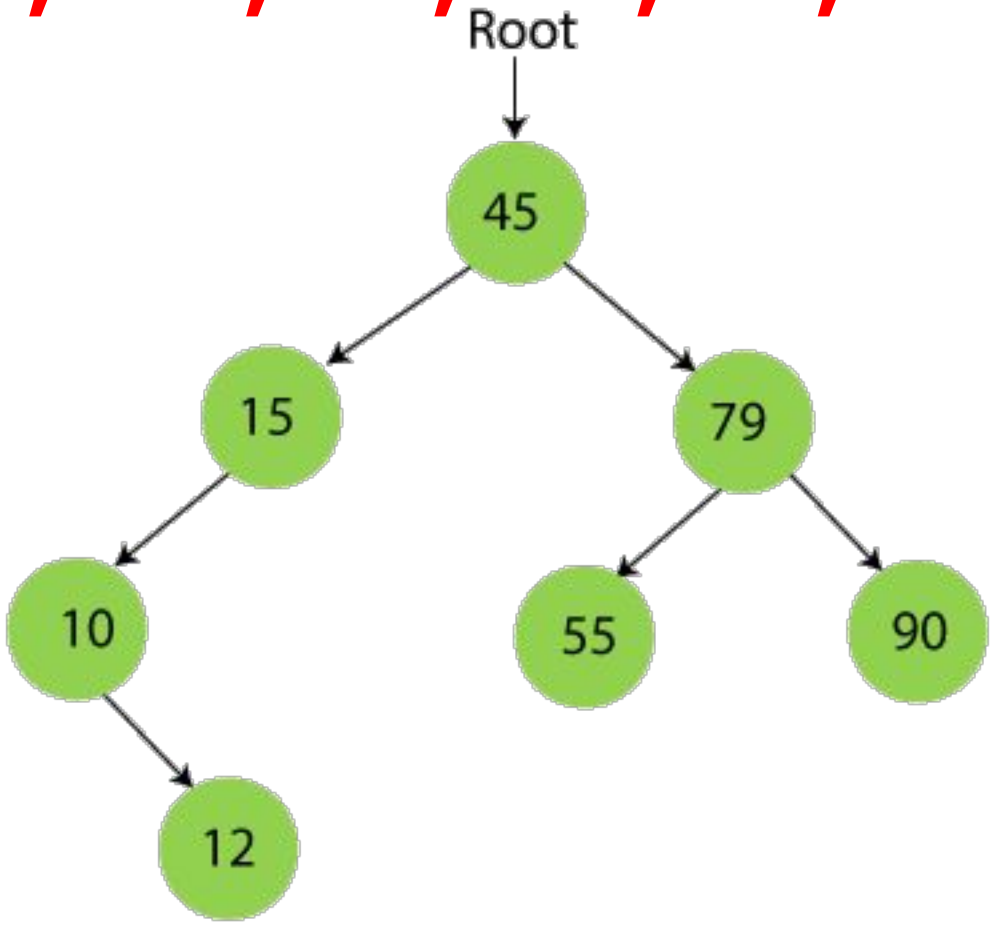
45, 15, 79, 90, 10, **55, 12, 20, 50**



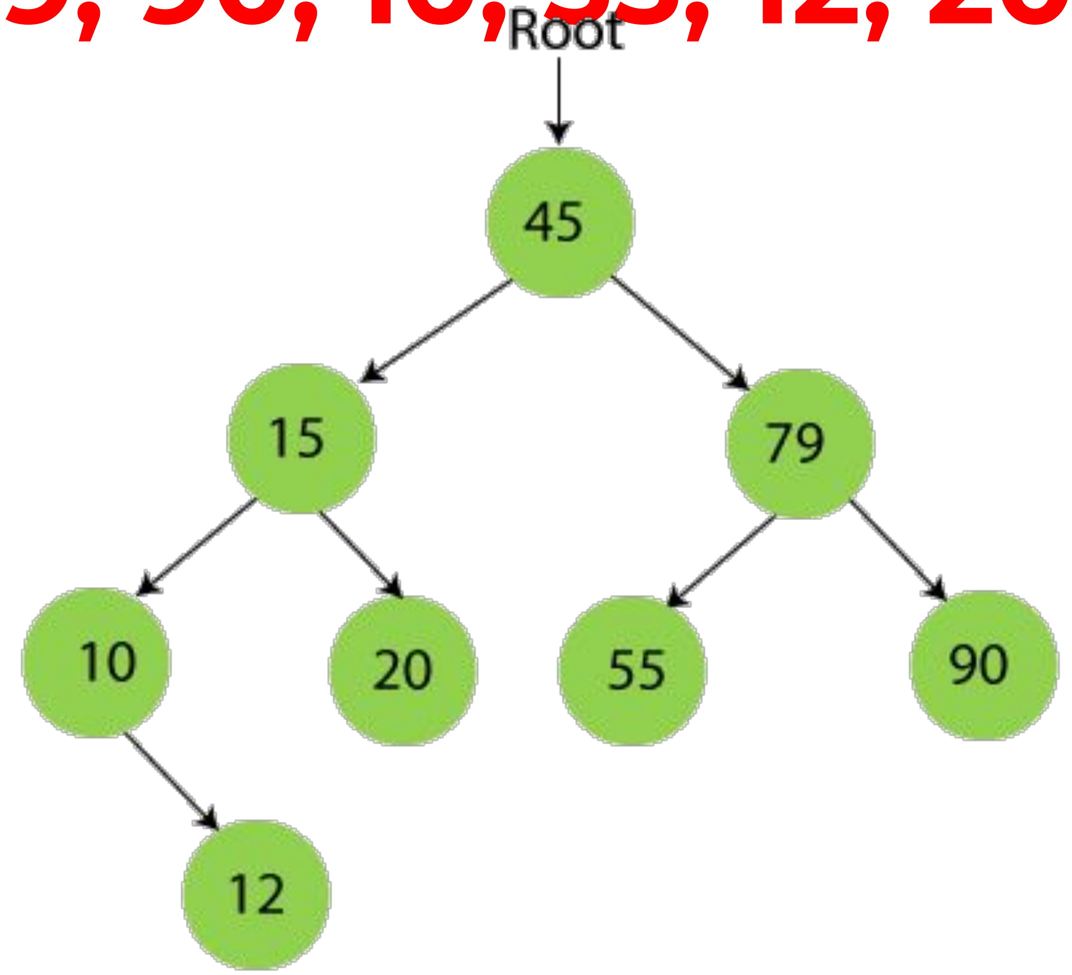
45, 15, 79, 90, 10, 55, 12, 20, 50



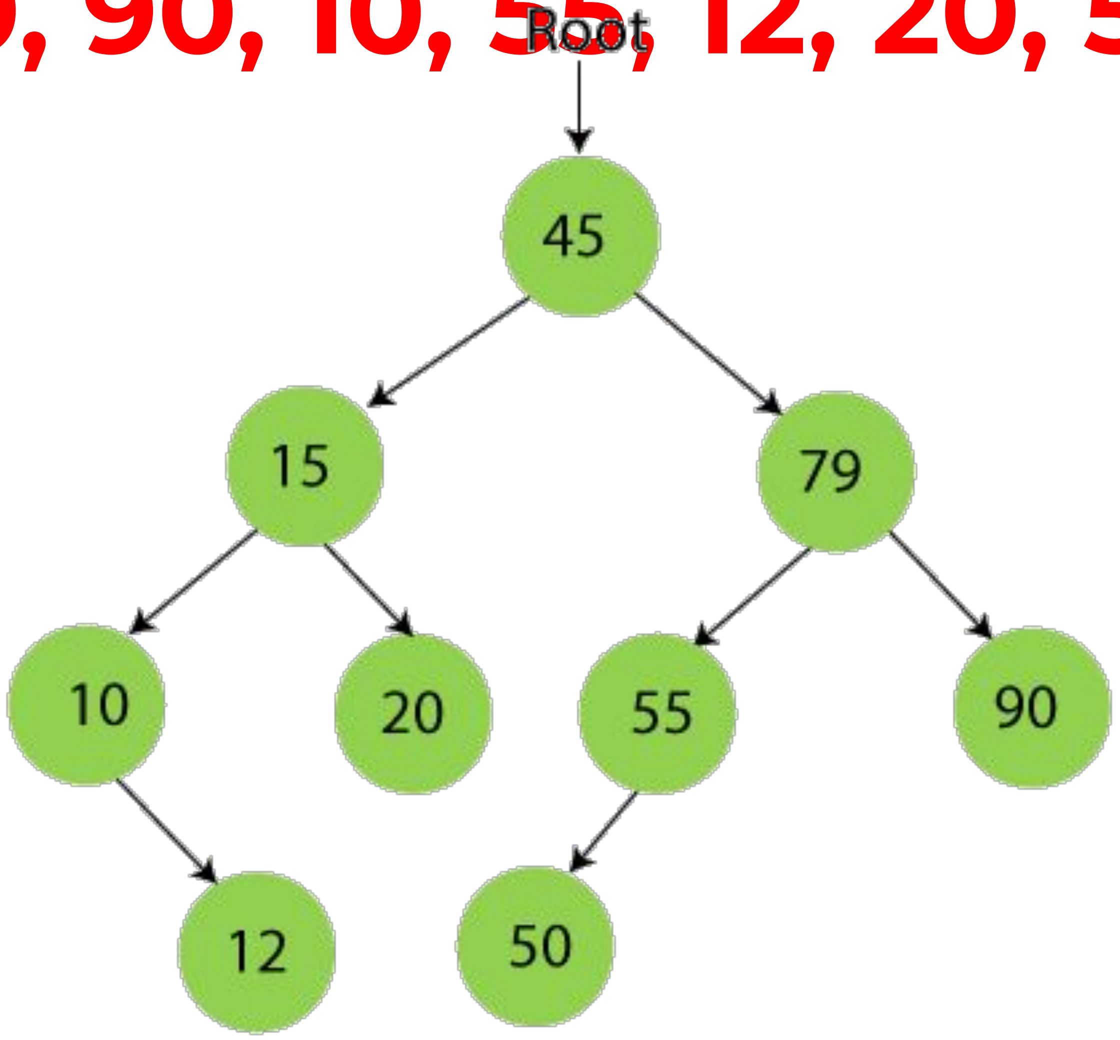
45, 15, 79, 90, 10, 55, 12, **20, 50**



45, 15, 79, 90, 10, 55, 12, 20, 50



45, 15, 79, 90, 10, 55, 12, 20, 50



Insertion Algorithms

```
Algorithm Insert(root, key)
    Input: root - root node of the binary search tree
           key - integer value to be inserted into the tree
    Begin
        // Step 1: If the tree is empty, create a new node and return it
        If root is null Then
            Return new Node(key)
        End If

        // Step 2: If the key already exists in the tree, return the root node
        If root.key == key Then
            Return root
        End If

        // Step 3: Recur down the tree
        If key < root.key Then
            root.left = Insert(root.left, key)
        Else
            root.right = Insert(root.right, key)
        End If

        // Step 4: Return the unchanged root node
        Return root
    End
```

Pseudocode

- Pseudocode is a way of representing an algorithm in a structured, plain-language format that’s easy to understand, regardless of programming language.

Searching in Binary search tree

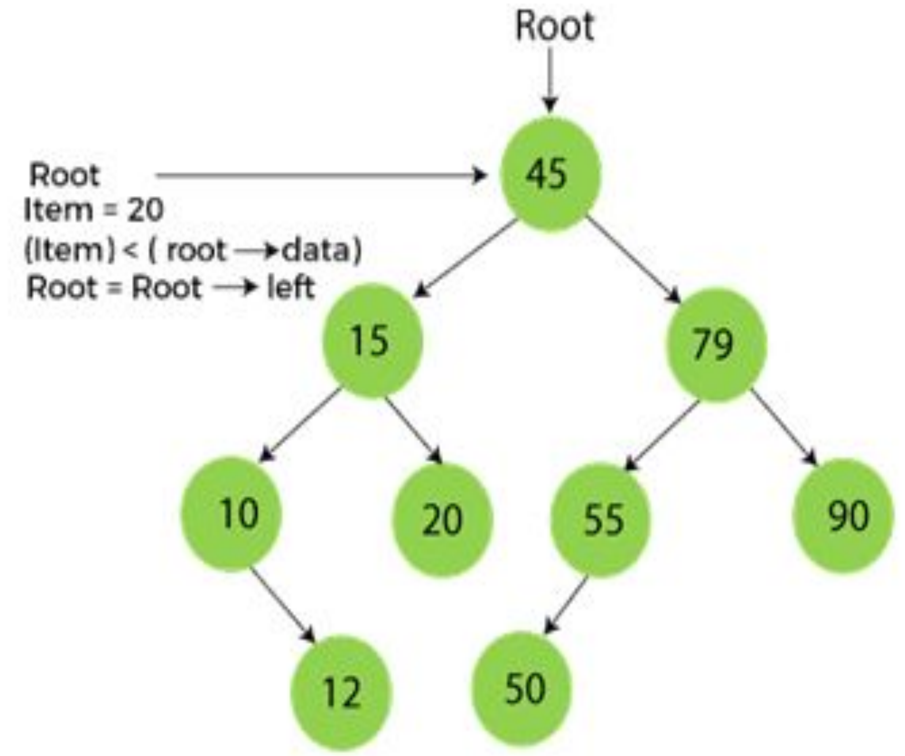
Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows –

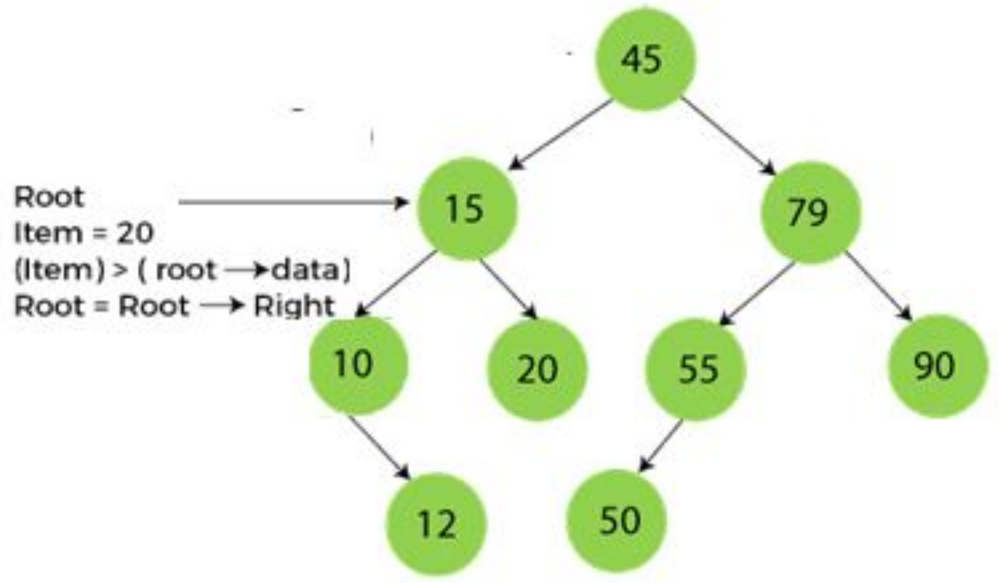
Searching in Binary search tree

- First, compare the element to be searched with the root element of the tree.
- If root is matched with the target element, then return the node's location.
- If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
- If it is larger than the root element, then move to the right subtree.
- Repeat the above procedure recursively until the match is found.
- If the element is not found or not present in the tree, then return NULL.

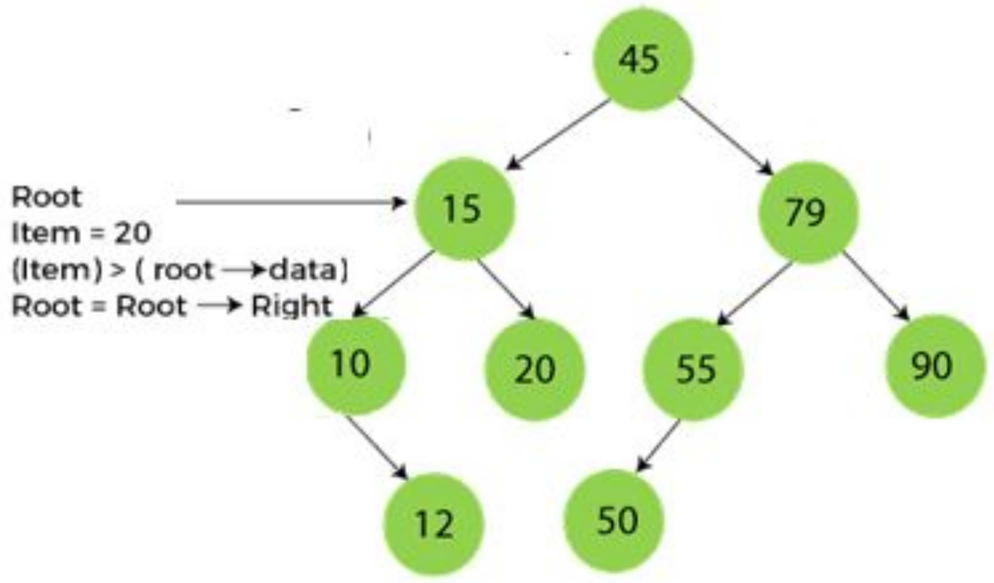
Searching in Binary search tree

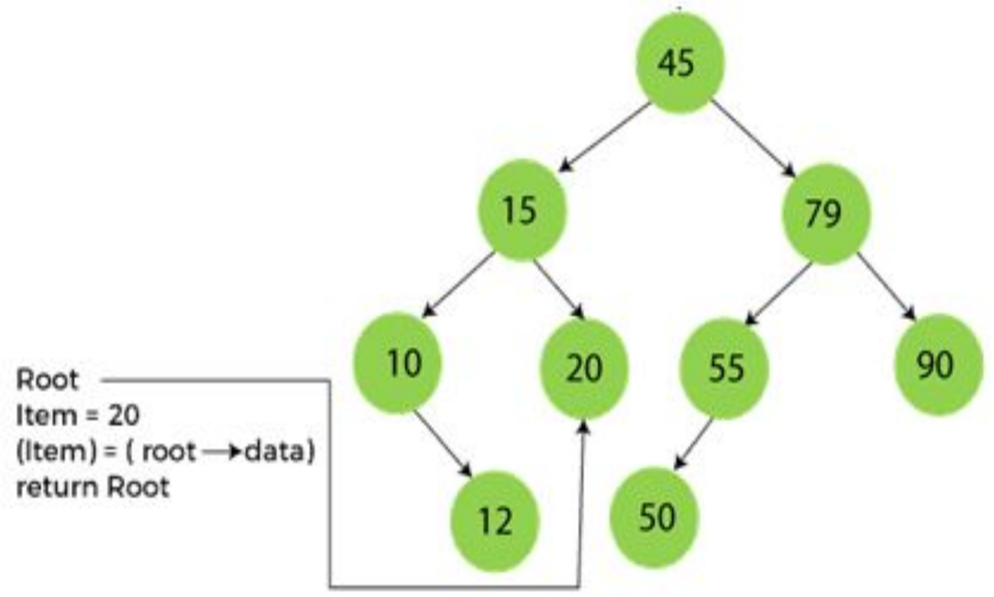
- We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.





Searching in Binary search Tree





Algorithm to search an element in Binary search tree

```
Search (root, item)
Step 1 - if (item = root → data) or (root = NULL)
return root
else if (item < root → data)
return Search(root → left, item)
else
return Search(root → right, item)
END if
Step 2 - END
```

Deletion in Binary Search Tree

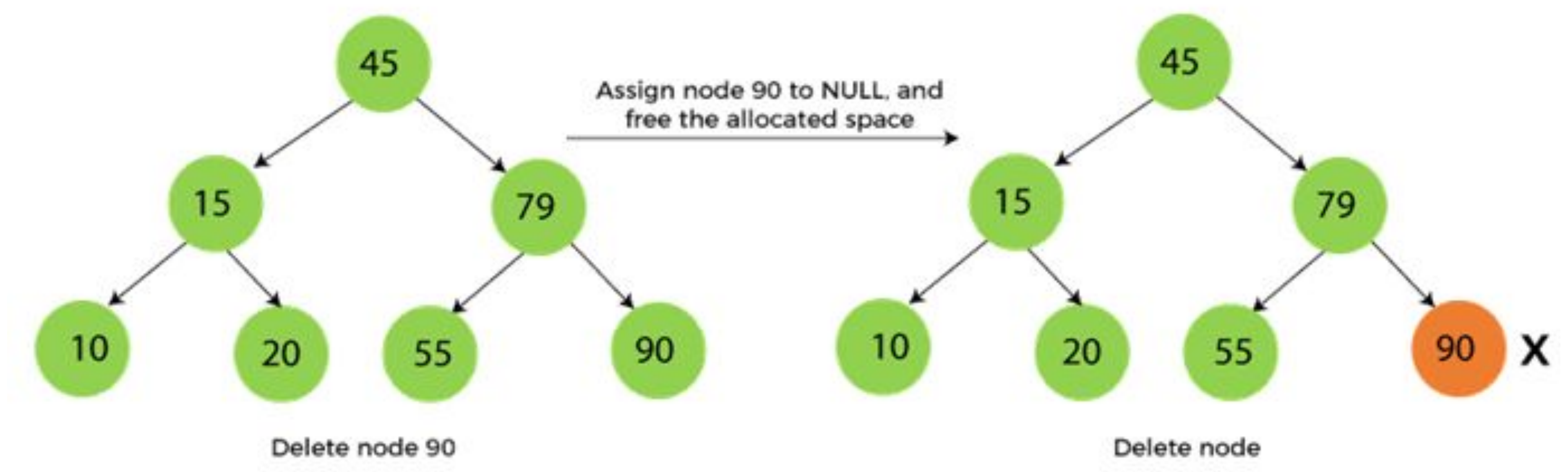
In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur –

- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

Deletion in Binary Search Tree

When the node to be deleted is the leaf node

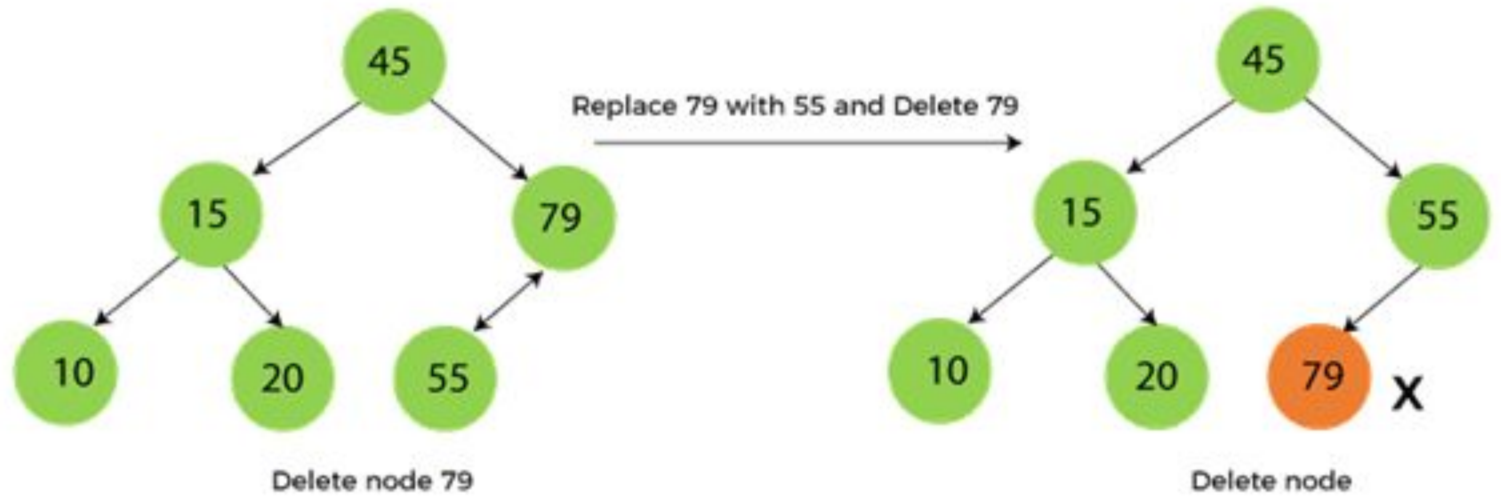
- It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.



Deletion in Binary Search Tree

When the node to be deleted has only one child

- In this case, we have to replace the target node with its child, and then delete the child node. suppose we have to delete the node 79,



Deletion in Binary Search Tree

When the node to be deleted has two children

- This case of deleting a node in BST is a bit complex among other two cases.
- In such a case, the steps to be followed are listed as follows –
  1. First, find the inorder successor of the node to be deleted.
  2. After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
  3. And at last, replace the node with NULL and free up the allocated space.

Deletion in Binary Search Tree

In-order successor

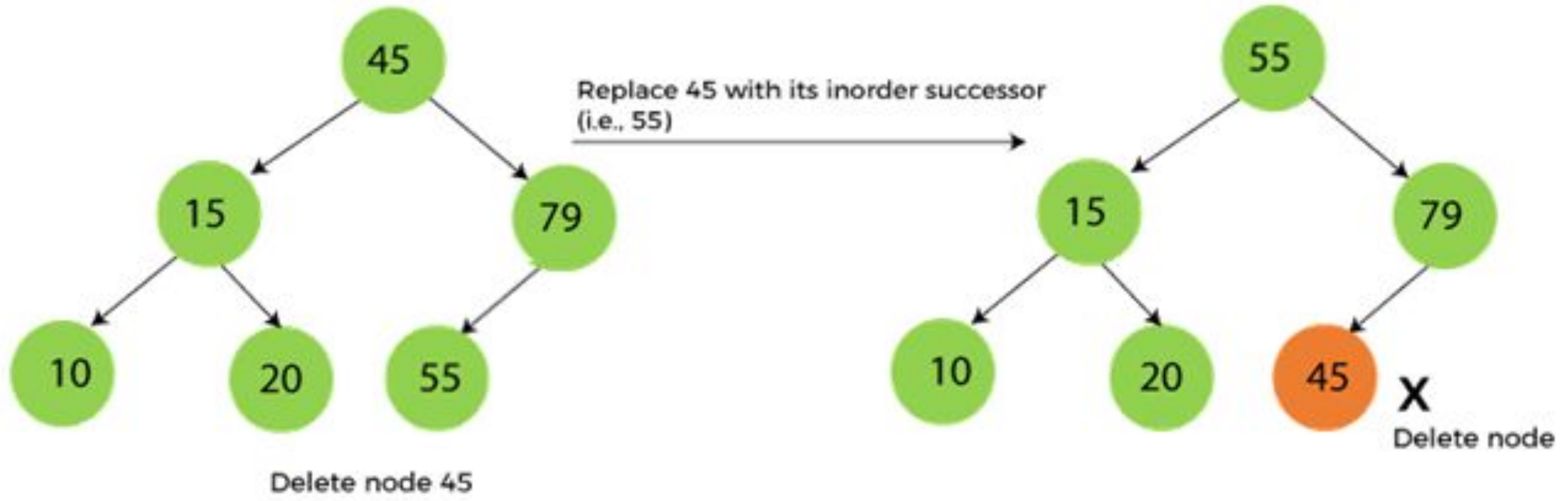
In a binary search tree (BST), the **inorder successor** of a node is the node that would come directly after it in an **inorder traversal** (left-root-right traversal) of the tree.

In simple terms:

1. **Definition**: The in-order successor of a node is the smallest node that is larger than the given node.

If the node has a **right subtree**, the in-order successor is the **leftmost node in the right subtree**.

Deletion in Binary Search Tree

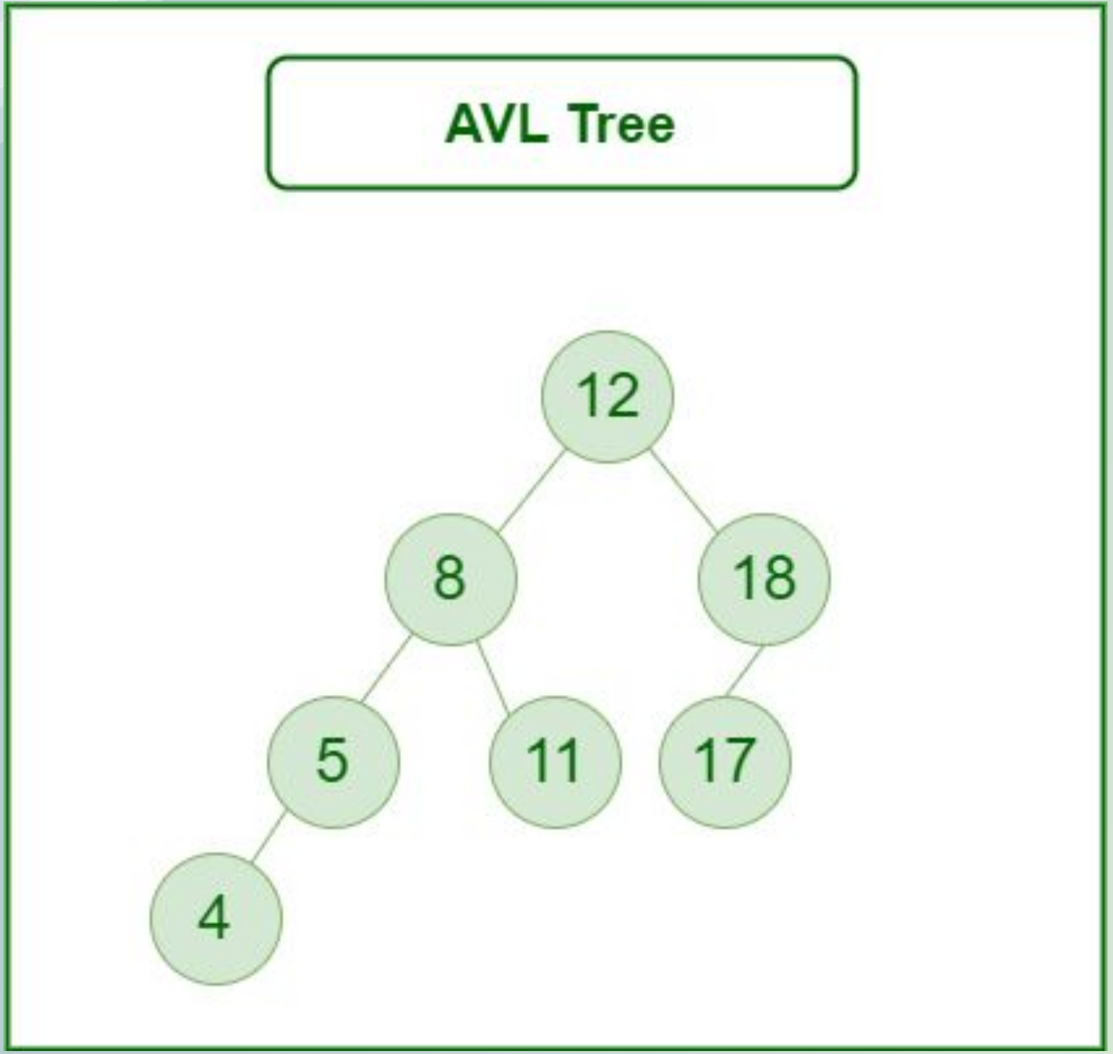


Algorithm

Time Complexity

Operations	Best Case	Average Case	Worst Case
Insertion	O(logn)	O(logn)	O(n)
Deletion	O(logn)	O(logn)	O(n)
Search	O(logn)	O(logn)	O(n)

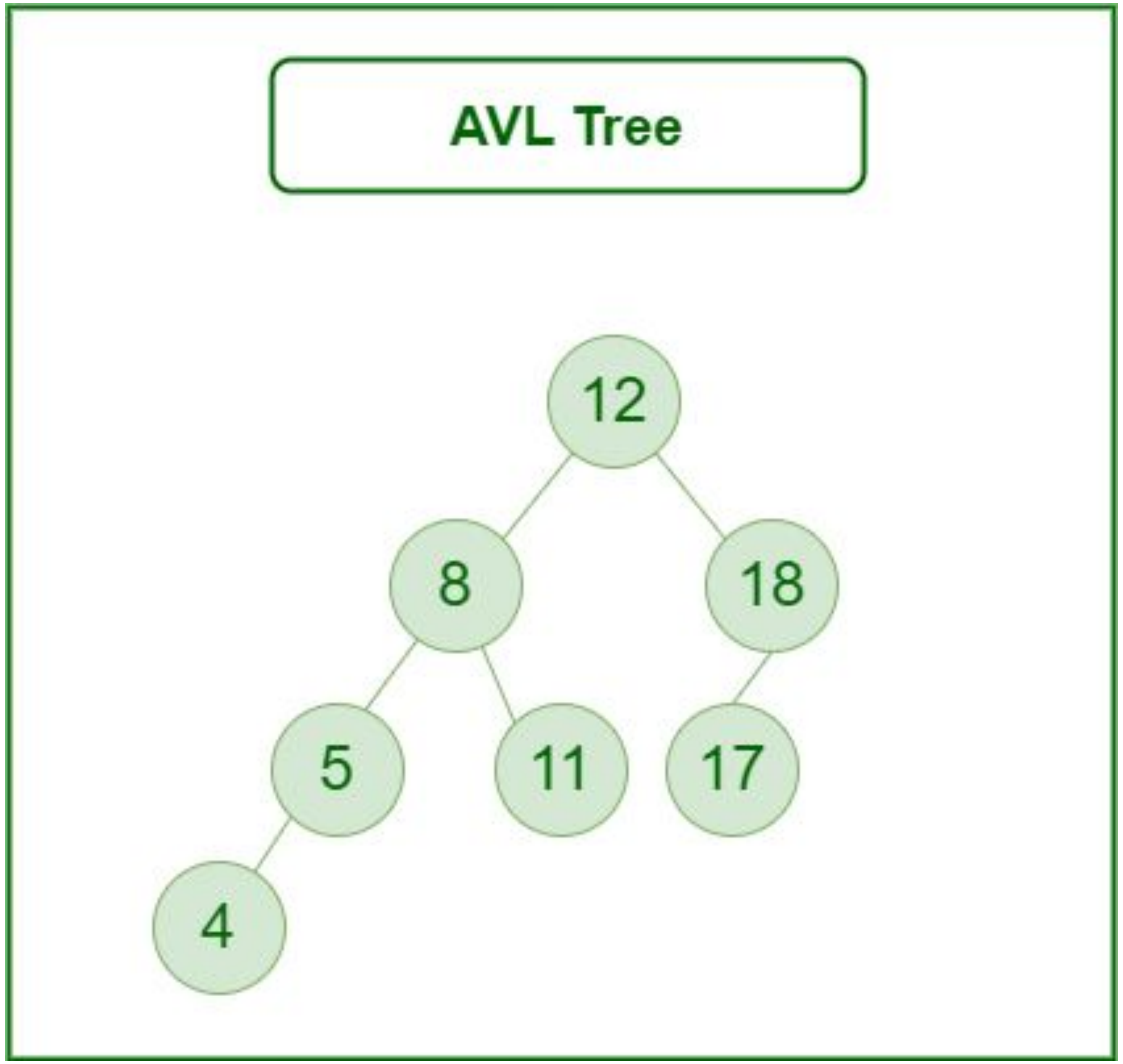
AVL Tree



AVL Tree

- *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 : Example



Operations on an AVL Tree:

Insertion

Deletion

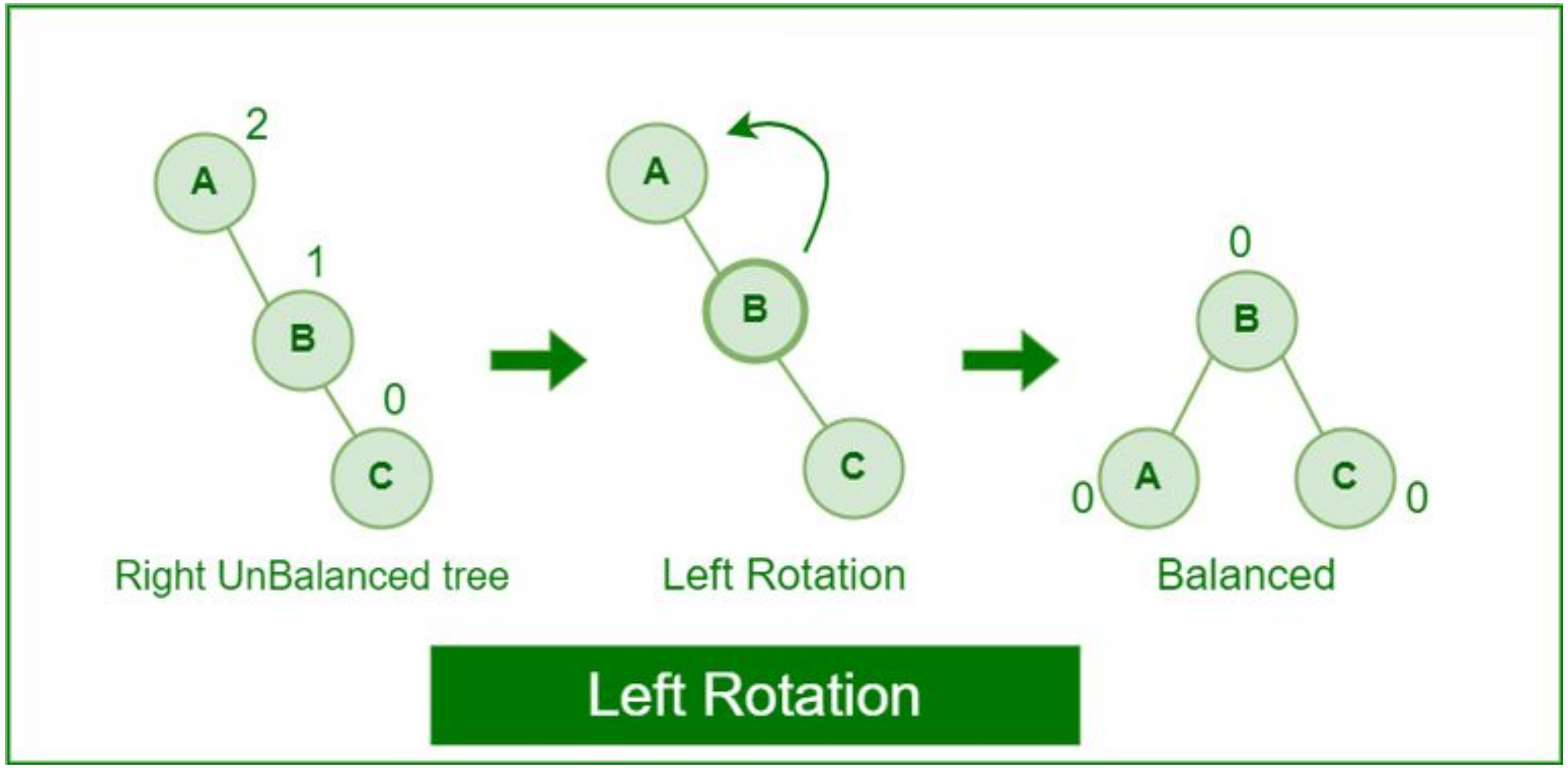
Searching [It is similar to performing a search in BST]

Rotating the subtrees in an AVL Tree:

- An AVL tree may rotate in one of the following four ways to keep itself balanced:
  1. Left Rotation
  2. Right Rotation
  3. Left Right Rotation
  4. Right Left Rotation

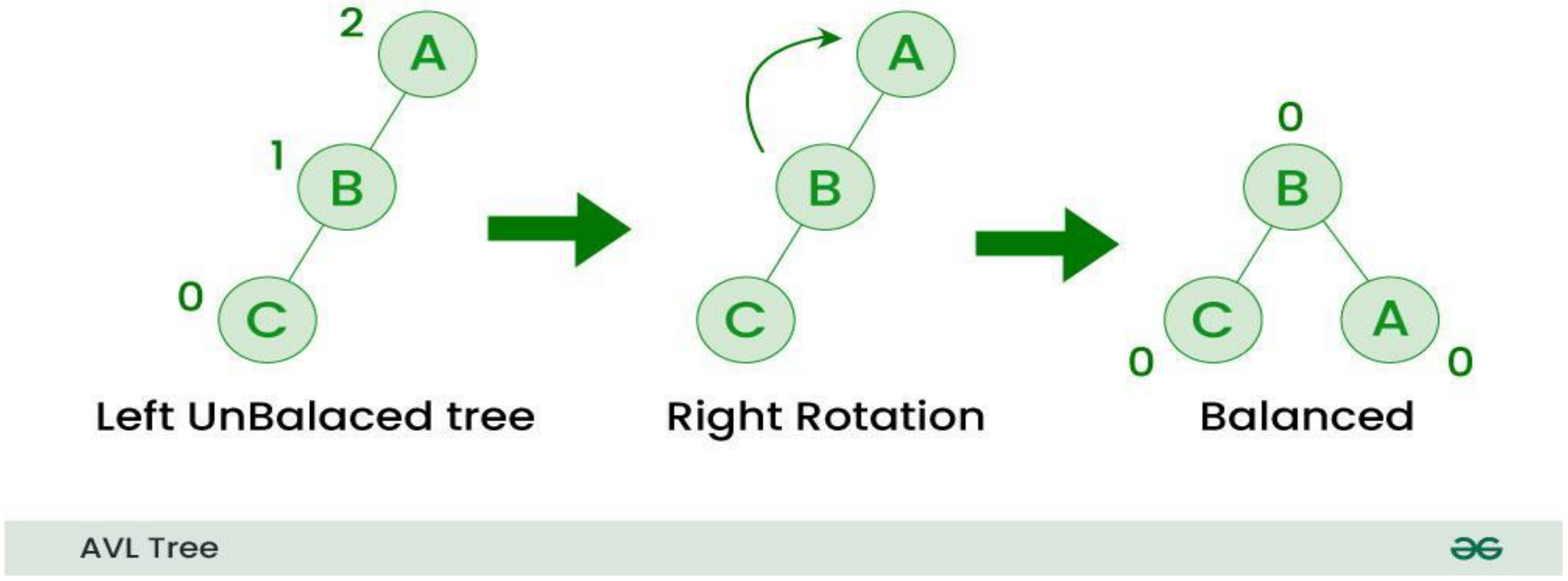
AVL Tree: Left Rotation

- 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.



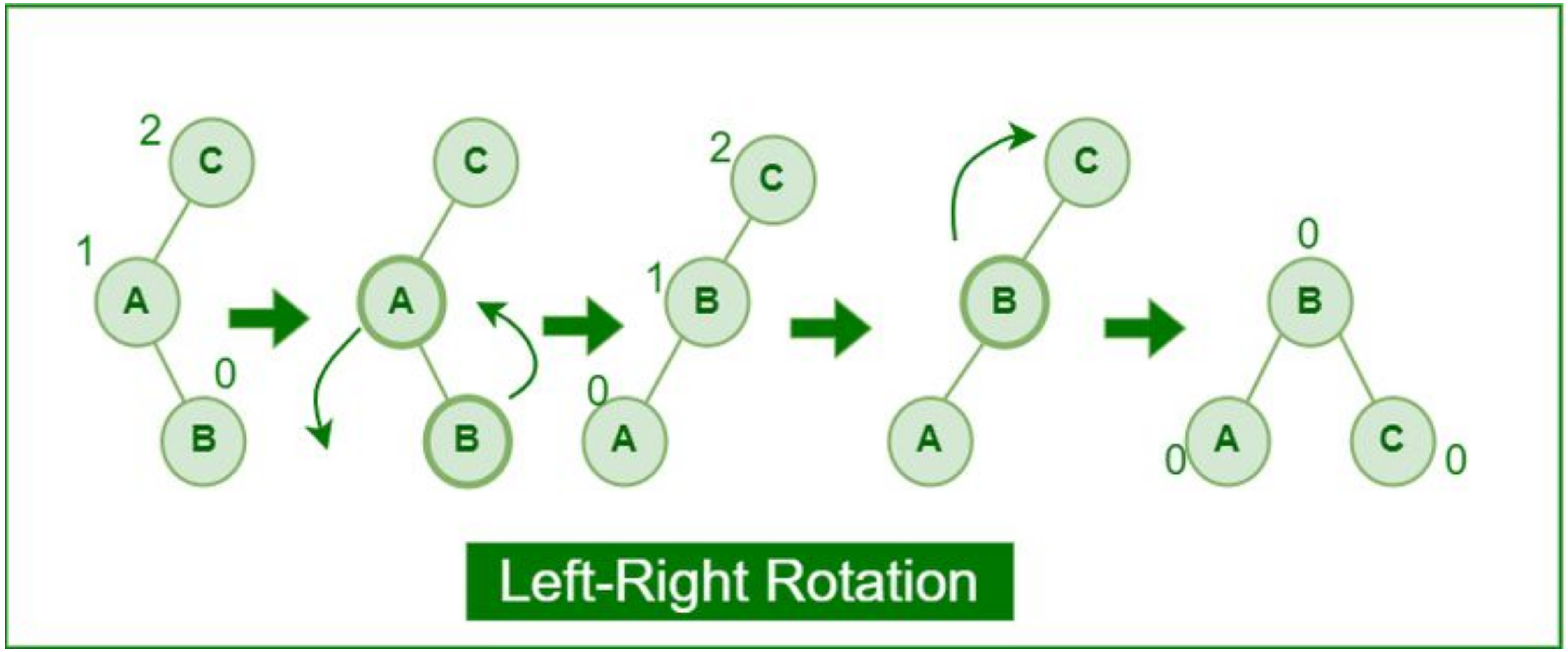
AVL Tree: Right Rotation

- 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.



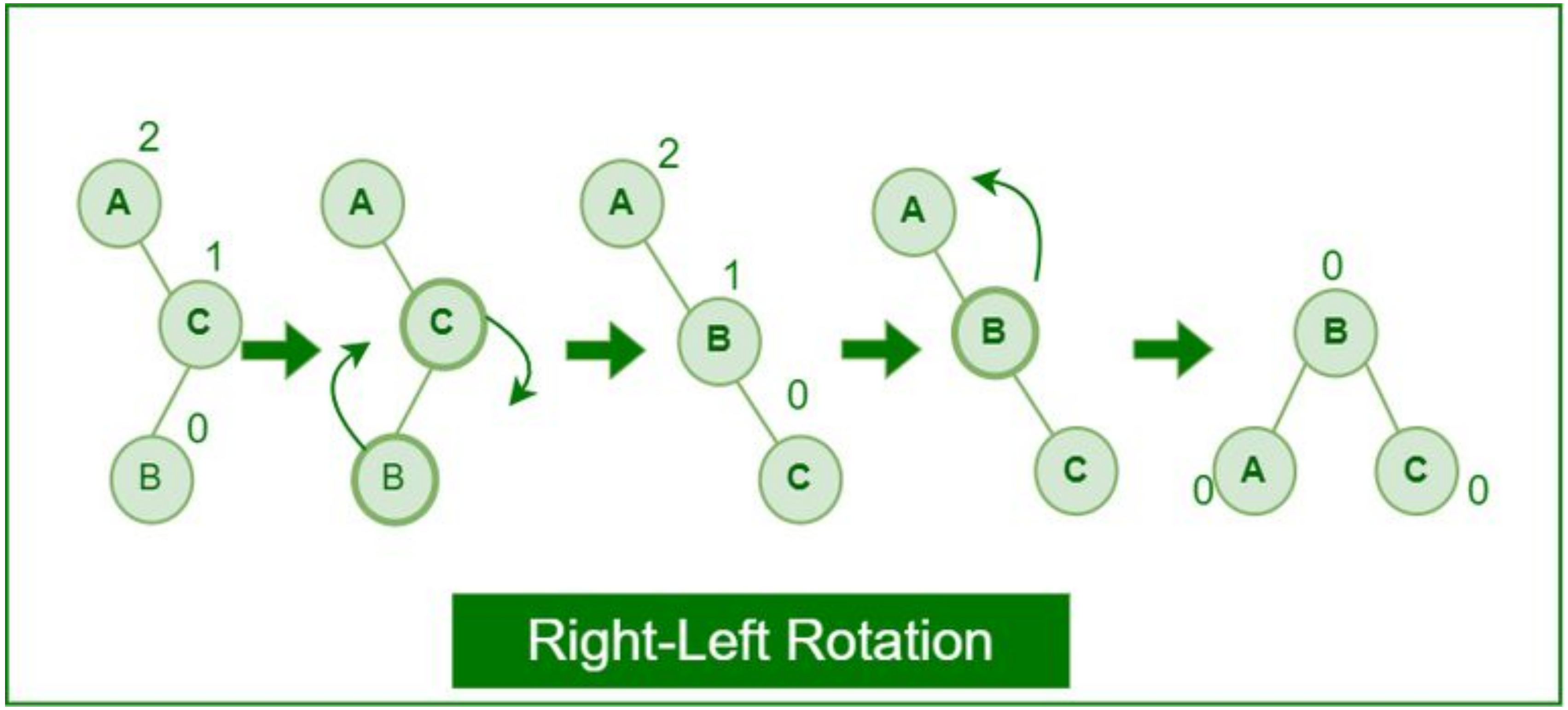
AVL Tree: Left-Right Rotation

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



AVL Tree: Right-Left-Rotation

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



Advantages of AVL Tree:

AVL trees can self-balance themselves and therefore provides time complexity as O(Log n) for search, insert and delete.

It is a BST only (with balancing), so items can be traversed in sorted order.

Since the balancing rules are strict compared to Red Black Tree, AVL trees in general have relatively less height and hence the search is faster.

AVL tree is relatively less complex to understand and implement compared to Red Black Trees.

Disadvantages of AVL Tree:

It is difficult to implement compared to normal BST and easier compared to Red Black

Less used compared to Red-Black trees.

Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.

Red-Black Tree