Binary Search Tree

Shoaib Rauf

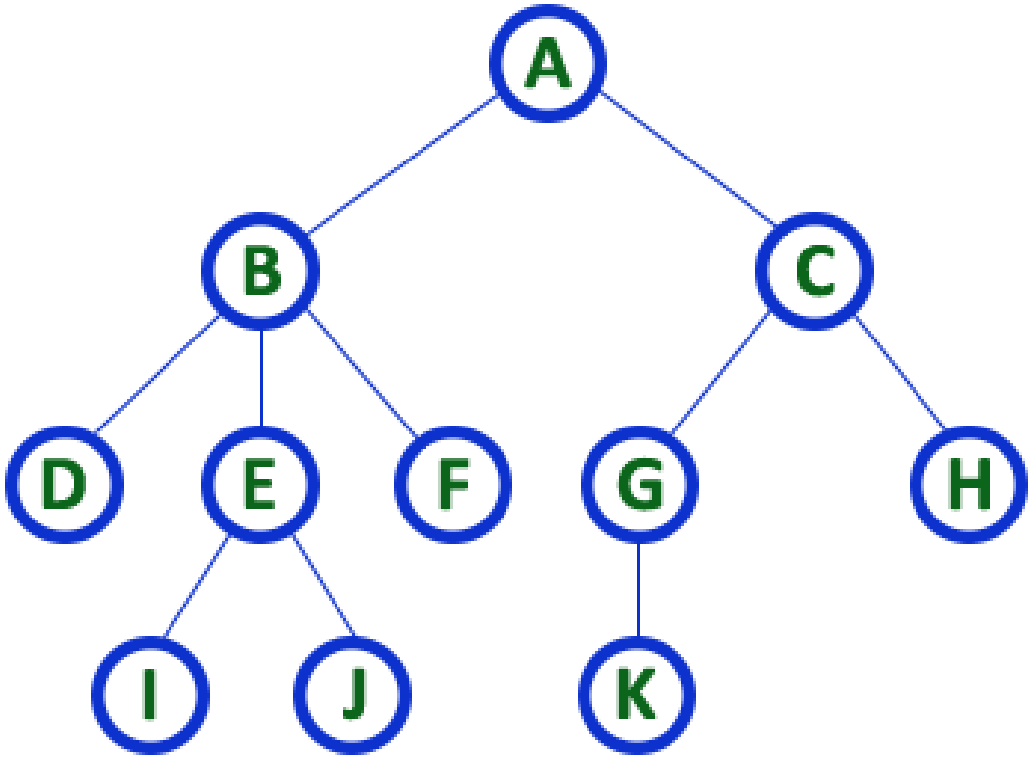
TREE Basics

- Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

- In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

- In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links

TREE Basics



TREE with 11 nodes and 10 edges

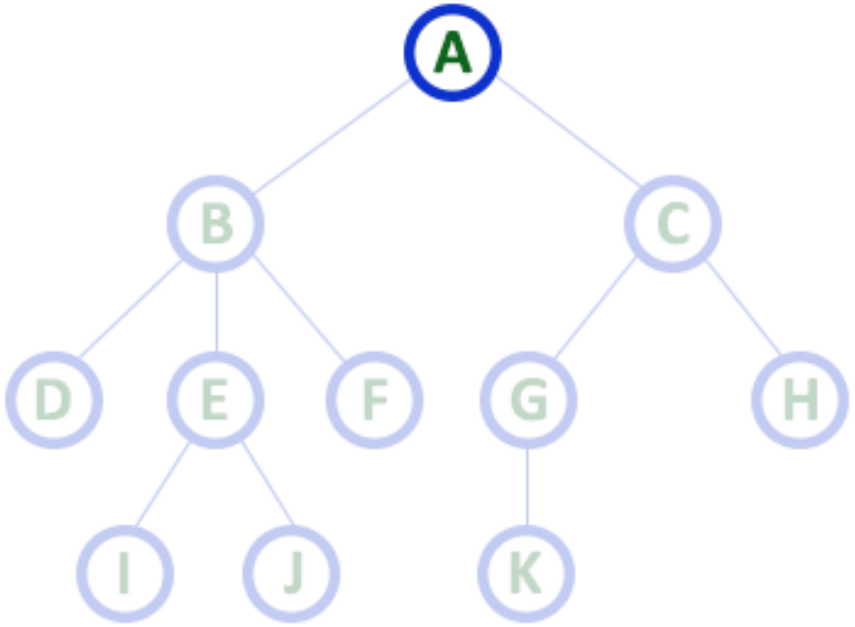
- **In any tree with 'N' nodes there will be maximum of 'N-1' edges**
- **In a tree every individual element is called as 'NODE'**

Terminologies

- Root
- Edge
- Parent
- Child
- Siblings
- Leaf
- Internal Node
- Degree
- Level
- Height
- Depth
- Path
- Sub-Tree

1. Root

- In a tree data structure, the first node is called as **Root Node**.
- Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

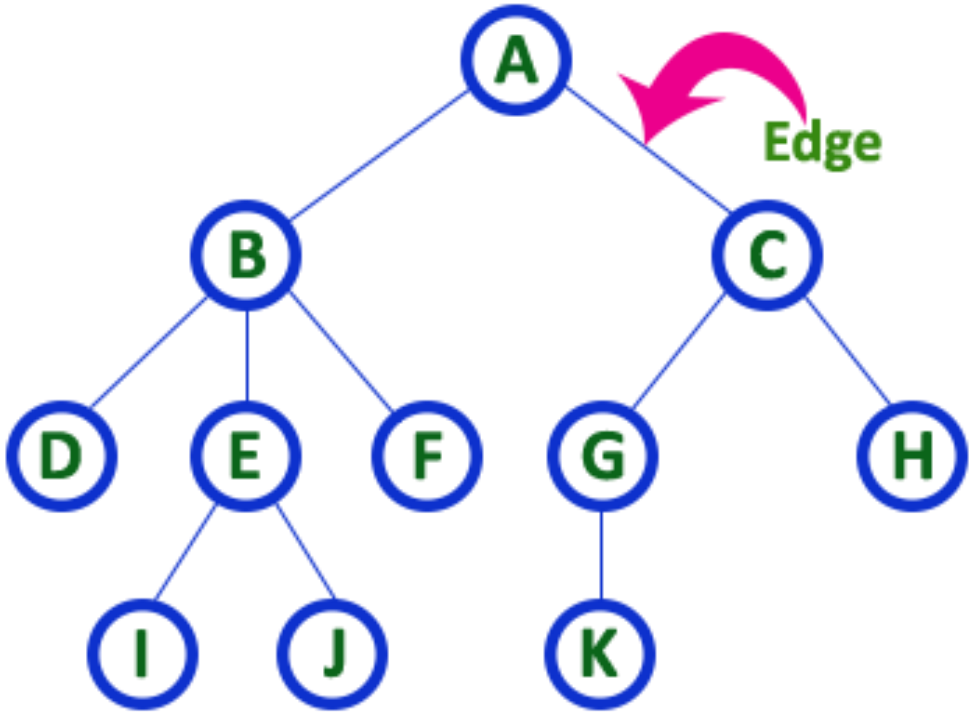


Here 'A' is the 'root' node

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

2. Edge

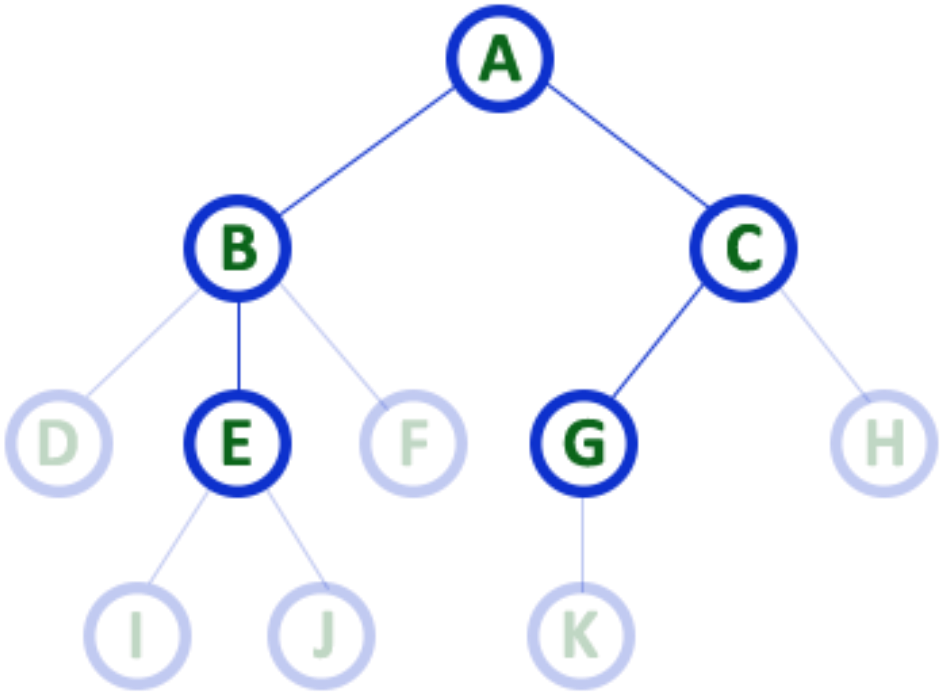
- In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



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

3. Parent

- In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**.
- In simple words, Parent node can also be defined as "**The node which has child / children**".

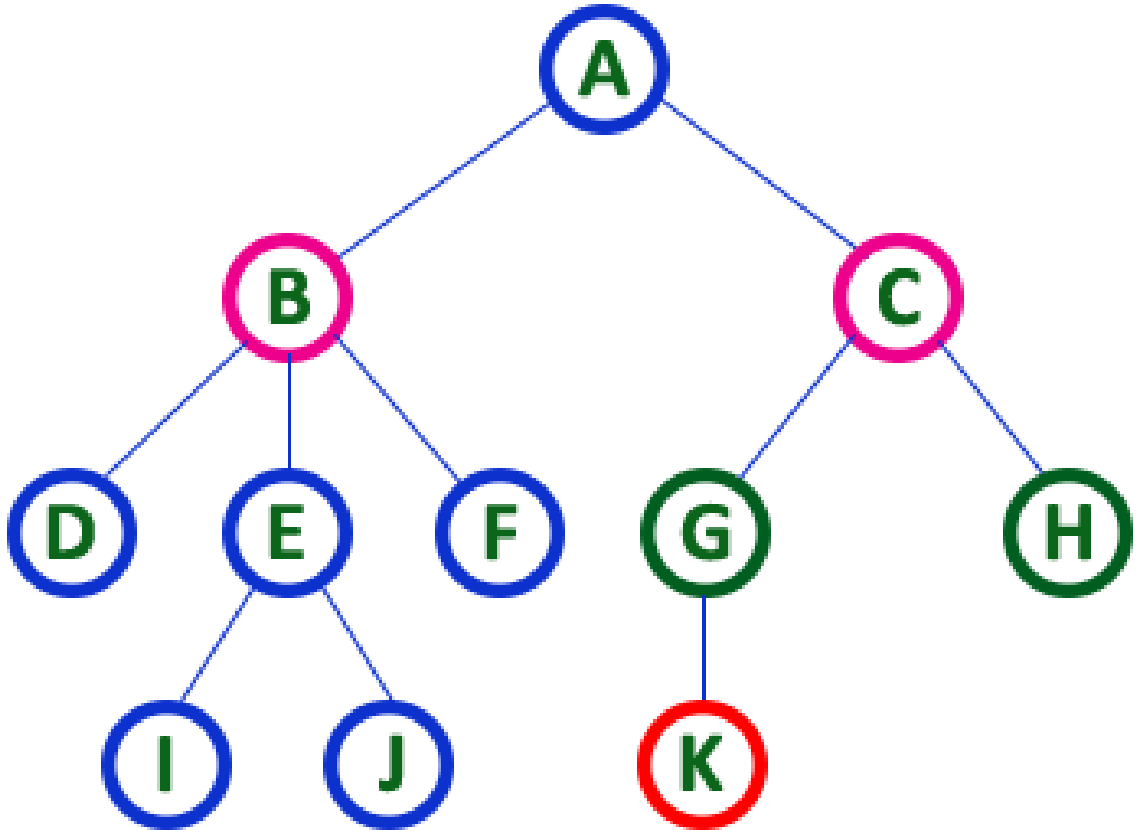


Here A, B, C, E & G are Parent nodes

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

4. Child

- In a tree data structure, the node which is descendant of any node is called as **CHILD Node**.



Here B & C are Children of A

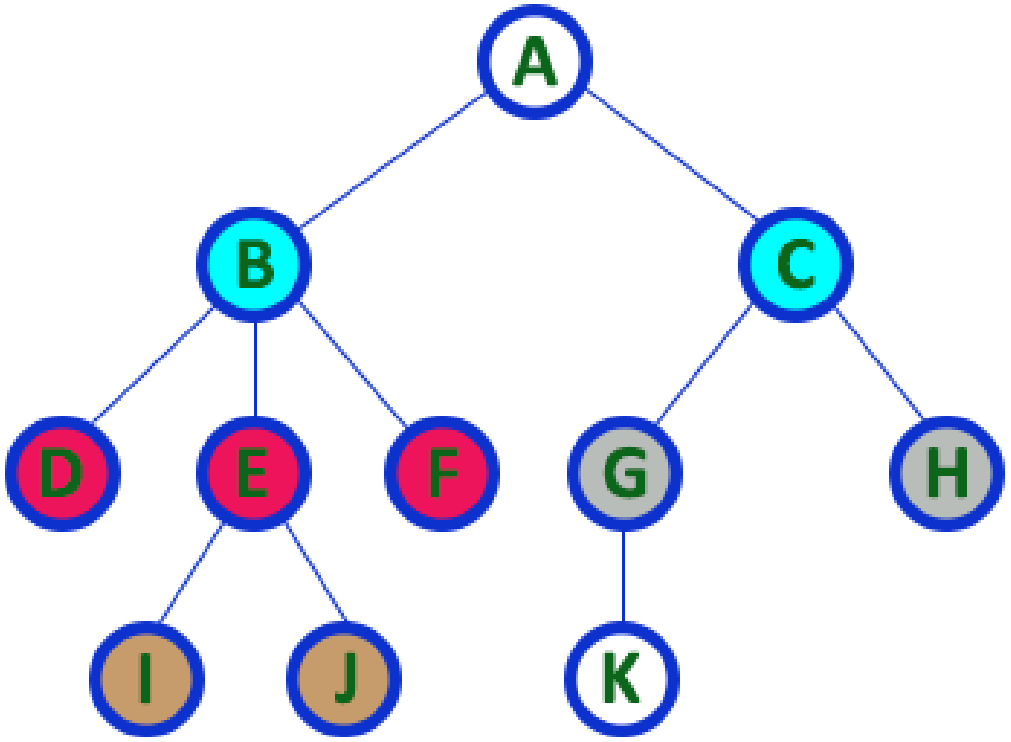
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

- In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes

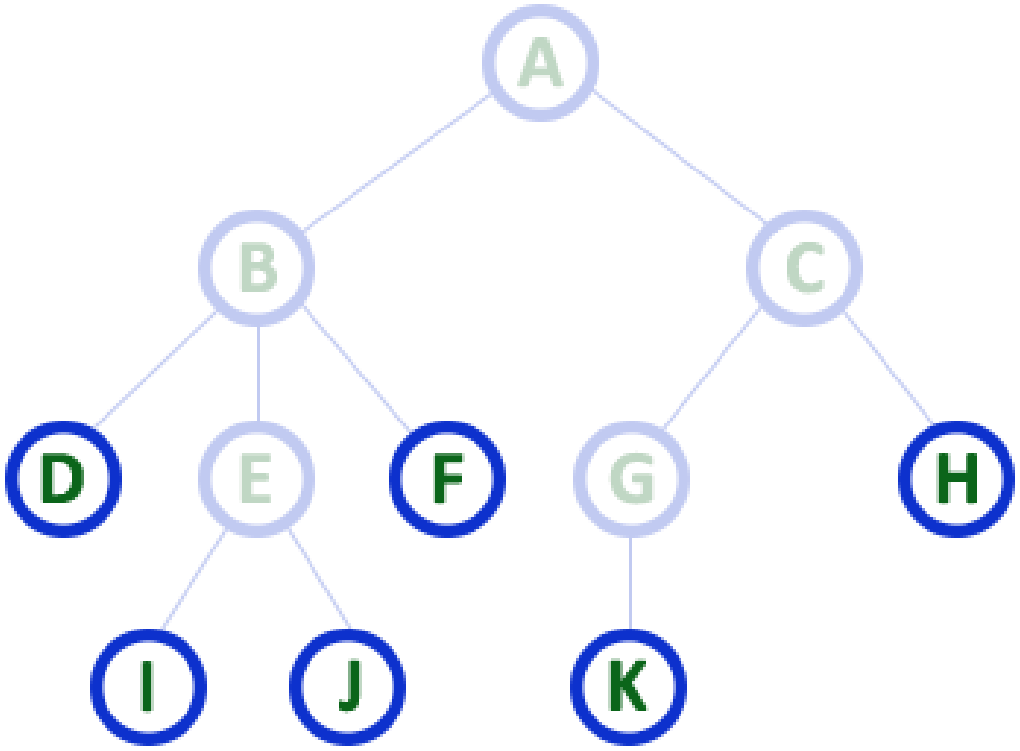


Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

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

6. Leaf

- In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

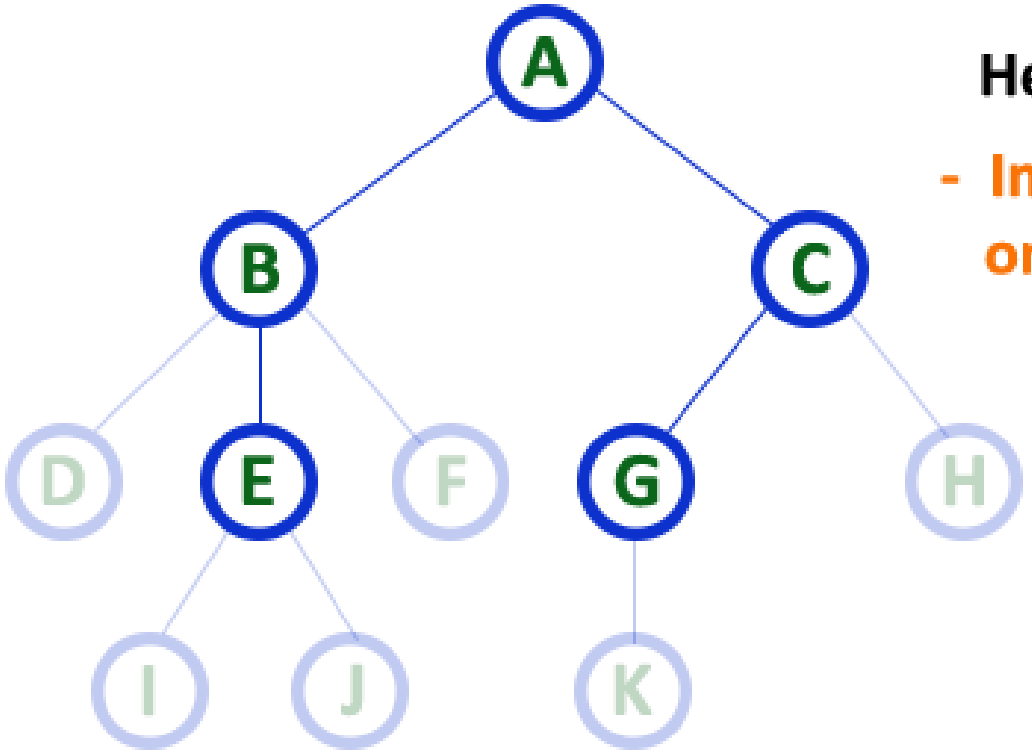


Here D, I, J, F, K & H are Leaf nodes

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

7. Internal Nodes

- In a tree data structure, the node which has at least one child is called as **INTERNAL Node**. In simple words, an internal node is a node with at least one child.



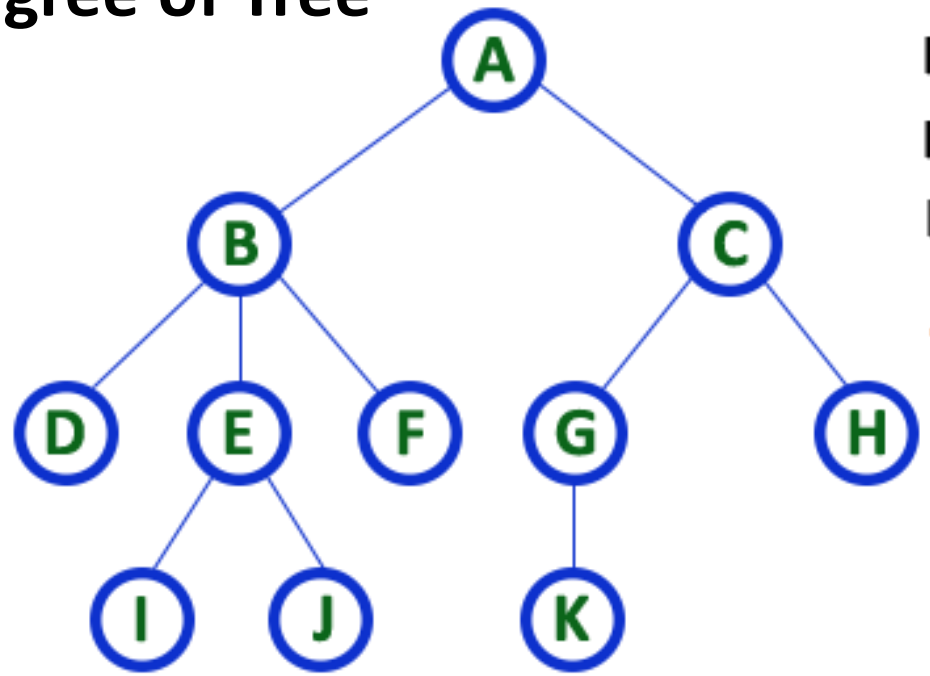
Here A, B, C, E & G are Internal nodes

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

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

8. 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

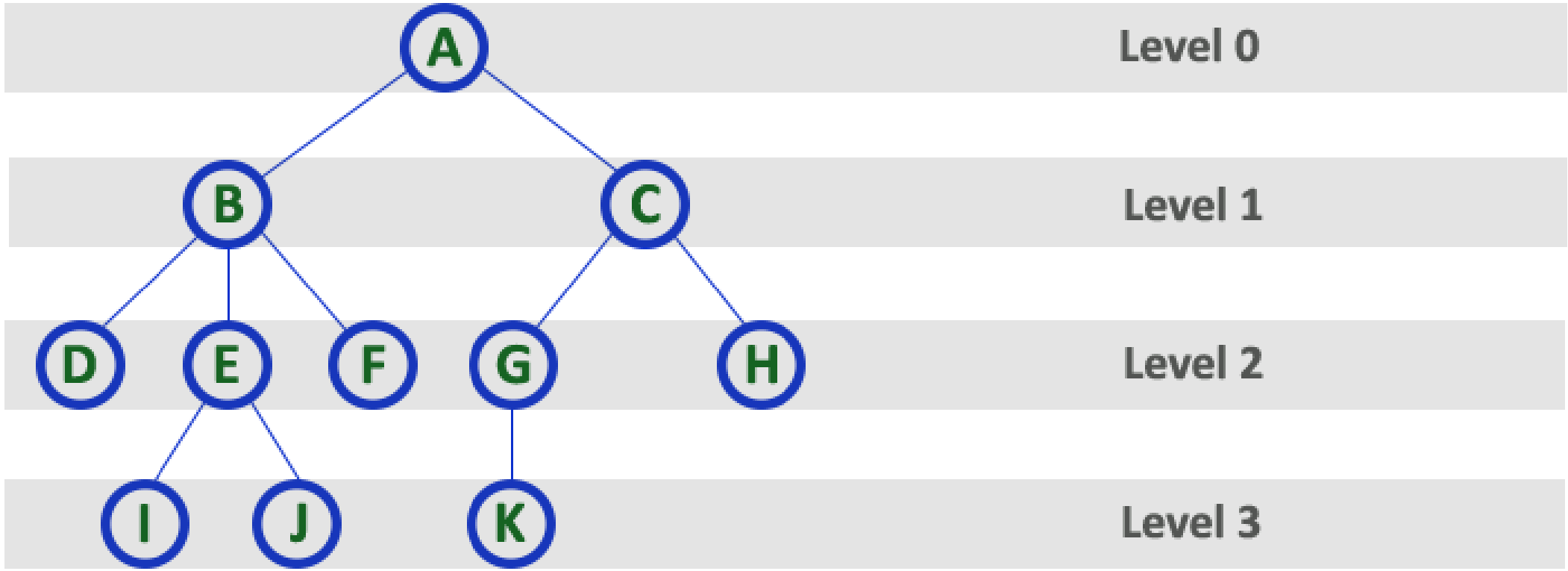
Here Degree of A is 2

Here Degree of F is 0

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

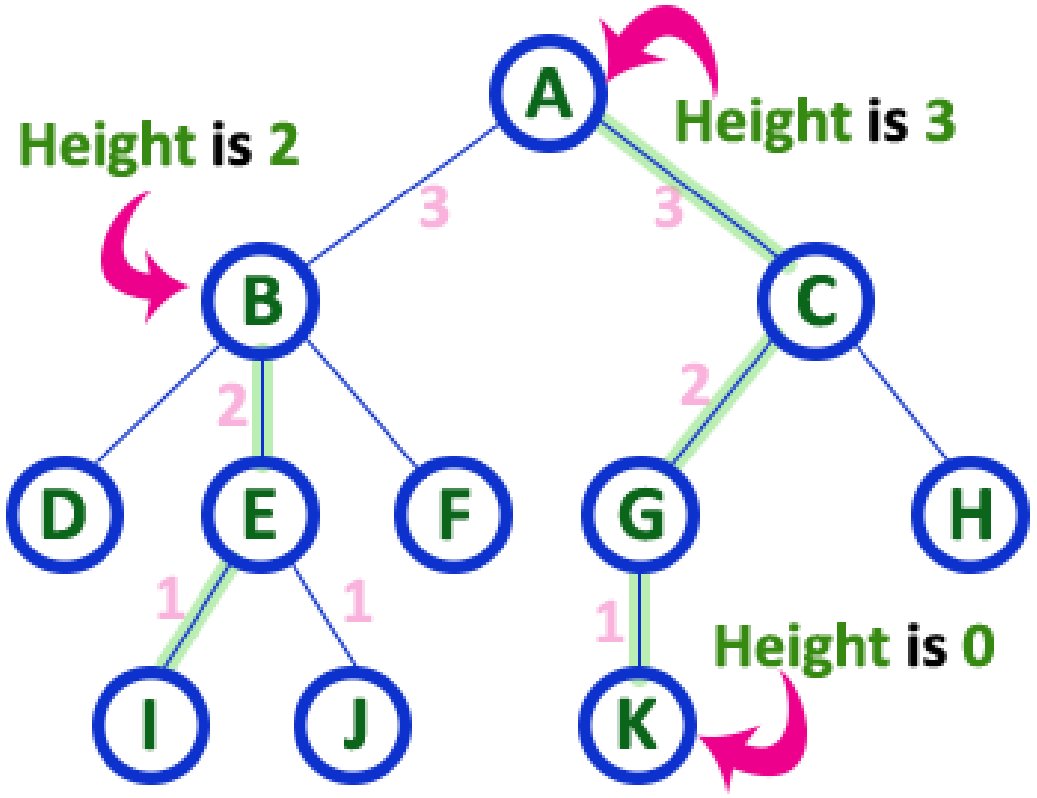
9. Level

- 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...



10. 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



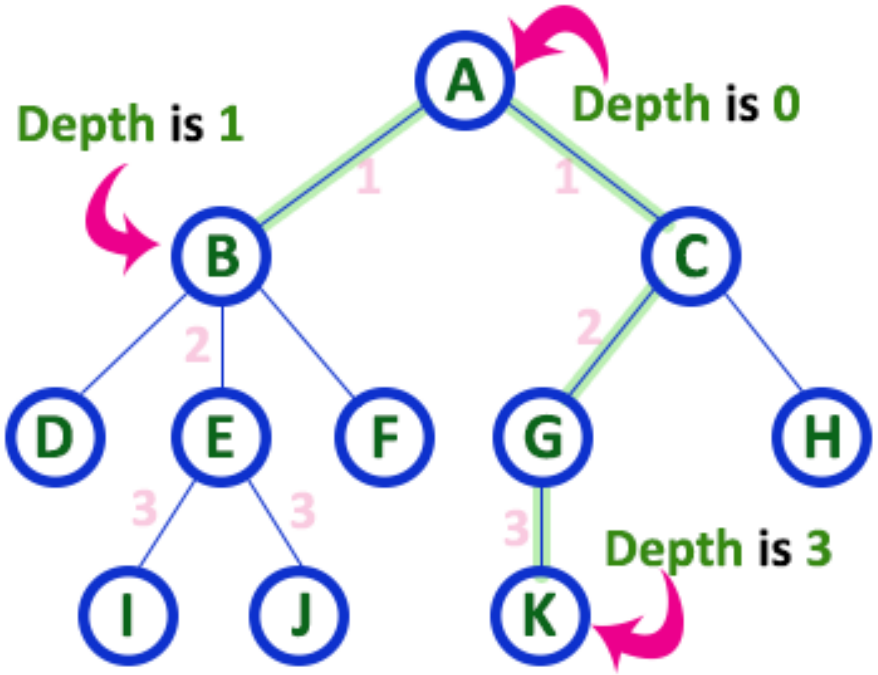
Here Height of tree is 3

- 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.

11. Depth

- 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'**



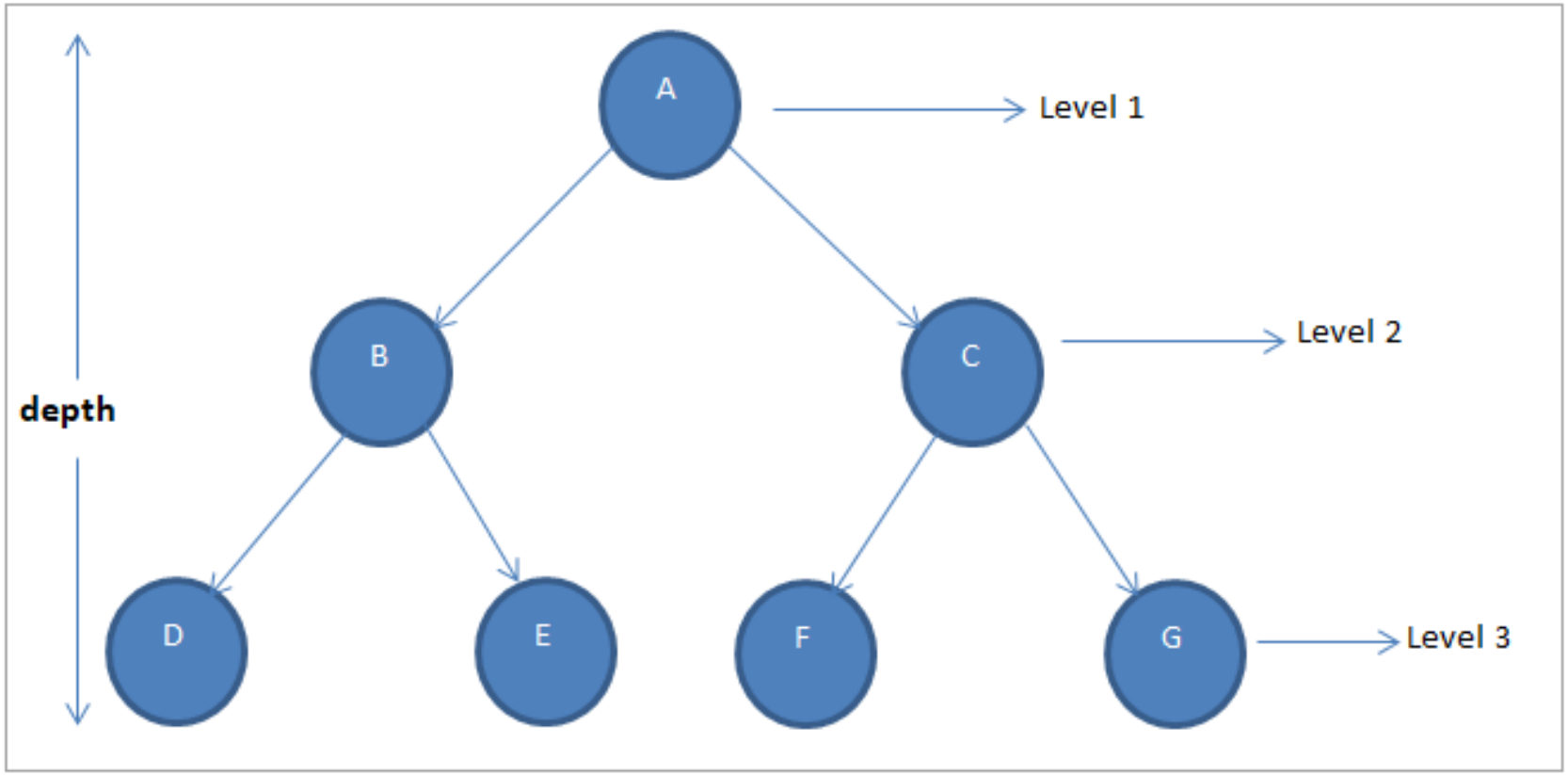
Here Depth of tree is 3

- In any tree, 'Depth of Node' is total 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.

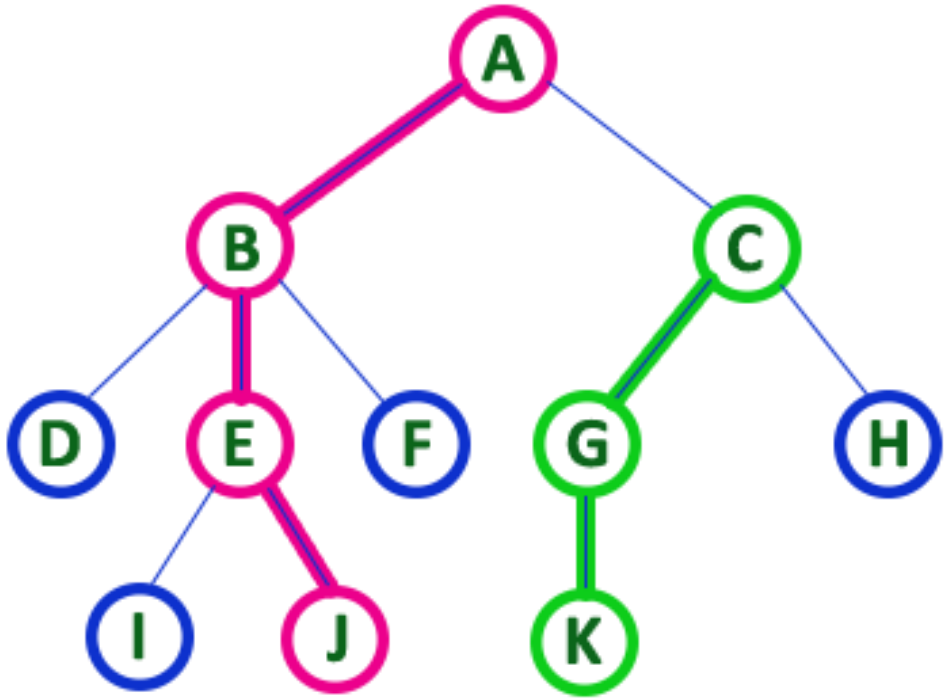
11. Depth & Height

- Given a binary tree of depth or height of h, the maximum number of nodes in a binary tree of height h = $2^h - 1$.
- Hence in a binary tree of height 3 (shown above), the maximum number of nodes = 2^3-1 = 7.



12. 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 two nodes.

Here, 'Path' between A & J is

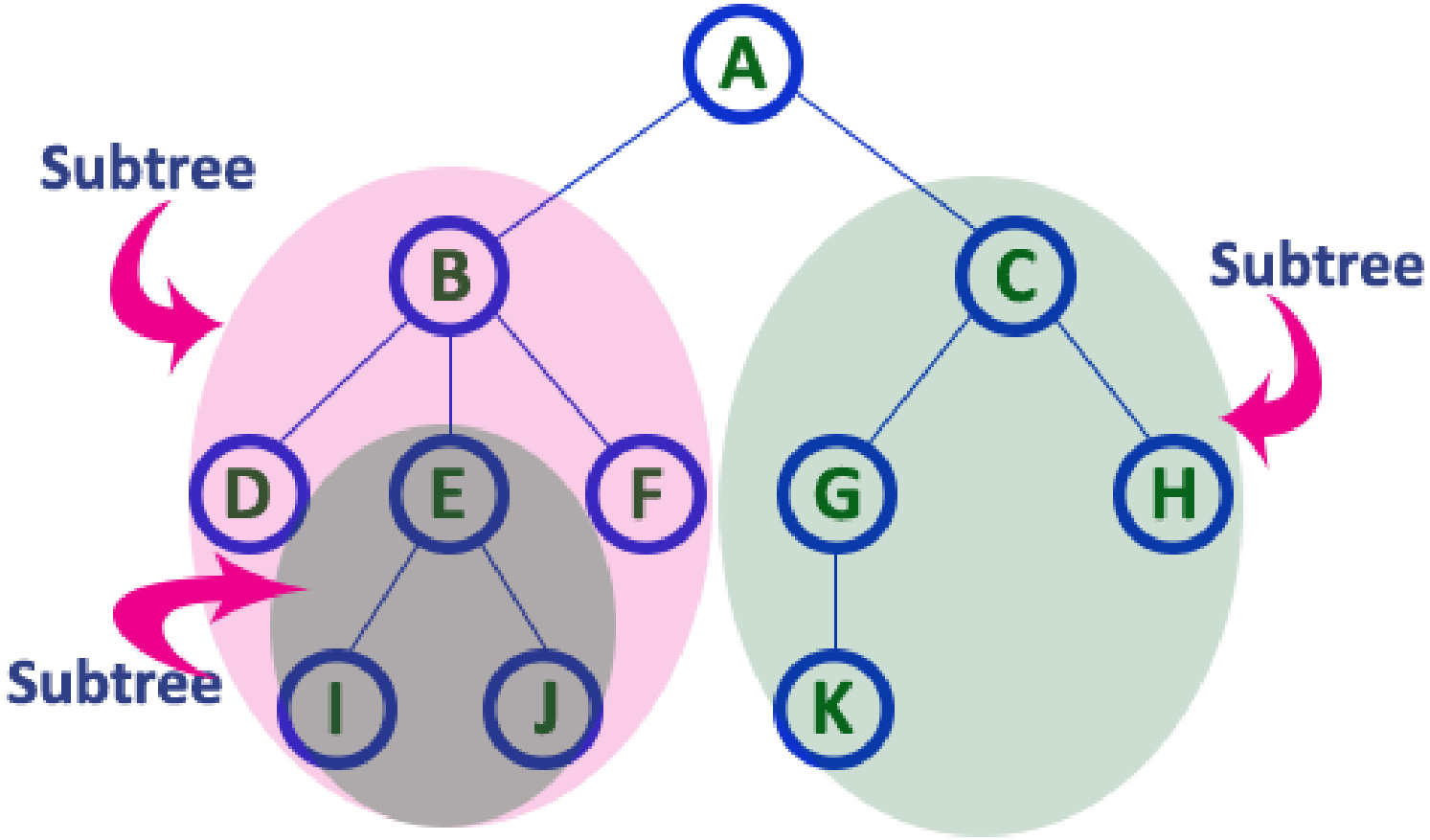
A - B - E - J

Here, 'Path' between C & K is

C - G - K

12. Sub Tree

- In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

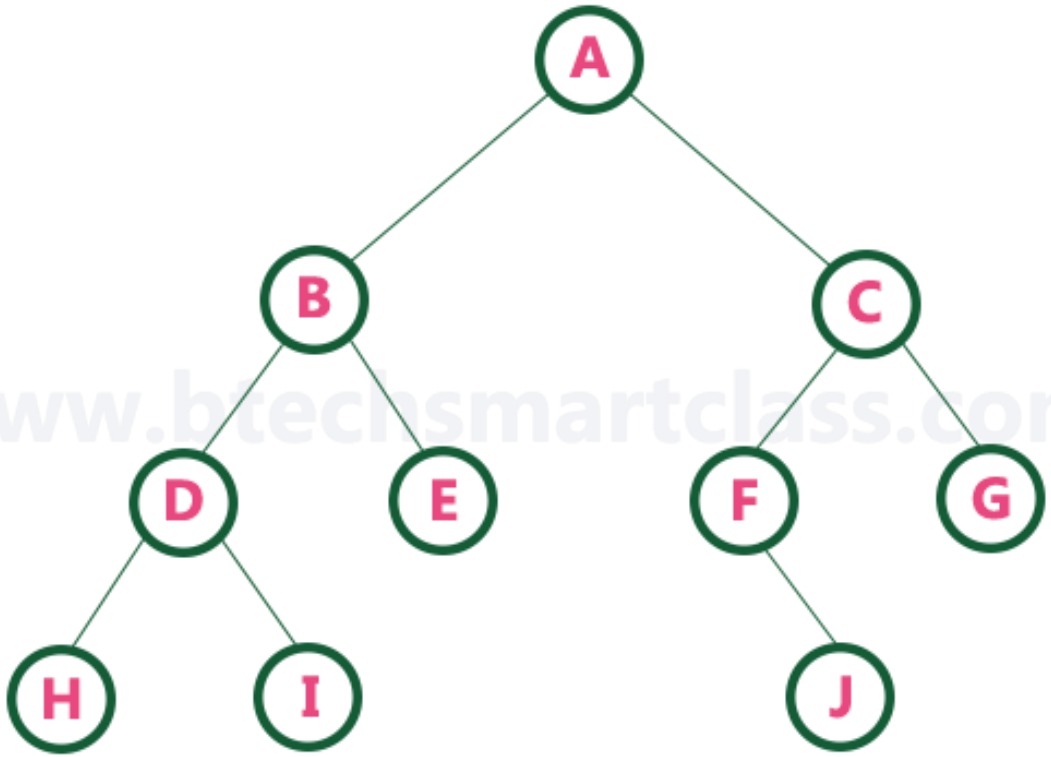


Binary Tree DataStructure

- A tree in which every node can have a maximum of two children is called Binary Tree.
- In a normal tree, every node can have any number of children.
- A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**.
- One is known as a left child and the other is known as right child.

Binary Tree DataStructure

- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.



Binary Tree: Complexity

- The BST is built on the idea of the binary search algorithm, which allows for fast lookup, insertion and removal of nodes.
- Insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree, O(log n)
- However, some times the worst case can happen, when the tree isn't balanced and the time complexity is O(n)

- **Worst-case performance: O(n)**
- **Best-case performance: O(1)**
- **Average performance: O(log n)**

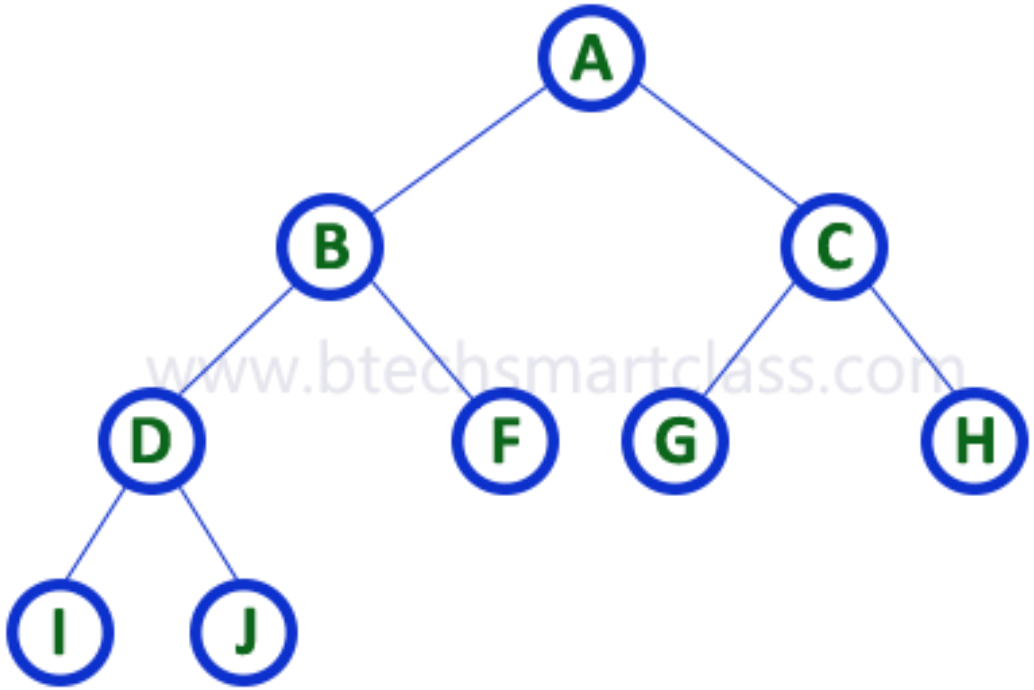
Types of Binary Tree(s):

Types of Binary tree are numbered below.

1. Strict or Full Binary Tree
2. Complete or Perfect Binary Tree
3. Extended Binary Tree

1. Full Binary Tree/Strict Binary Tree:

- In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none.
- Strictly binary tree is also called as **Full Binary Tree**

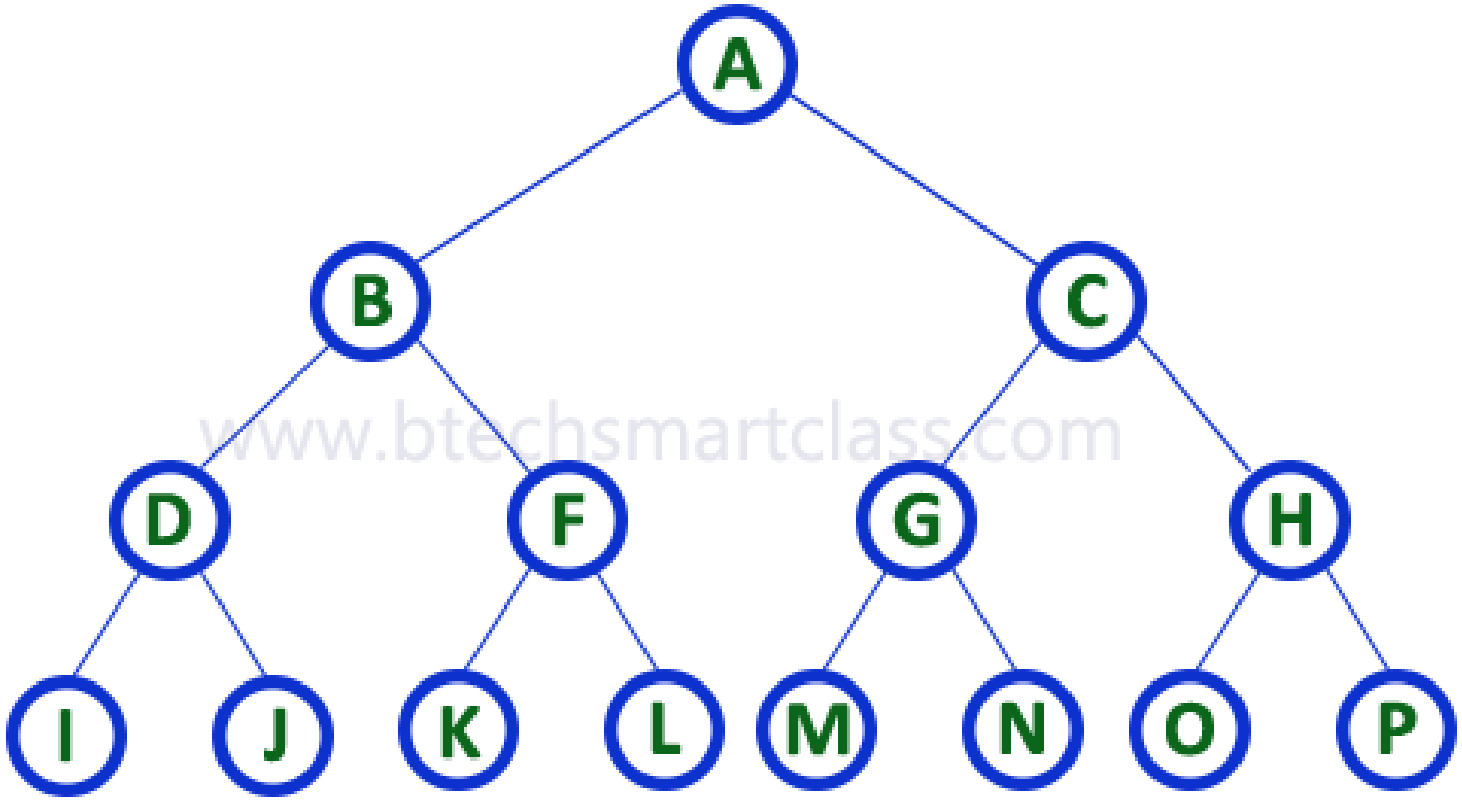


2. Complete or Perfect Binary Tree

- In a binary tree, every node can have a maximum of two children.
- But in strictly binary tree, every node **should have** exactly two children or none
- In complete binary tree all the nodes **must have** exactly two children and at every level of complete binary tree.
- Complete binary tree is also called as **Perfect Binary Tree**

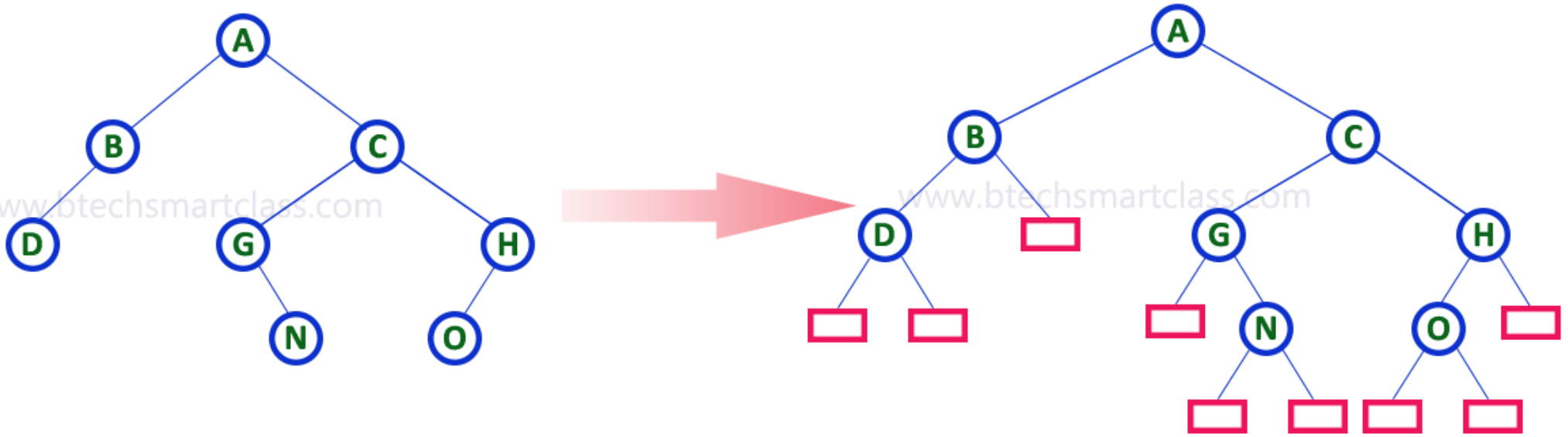
- For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

2. Complete or Perfect Binary Tree



3. Extended Binary Tree

- A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required

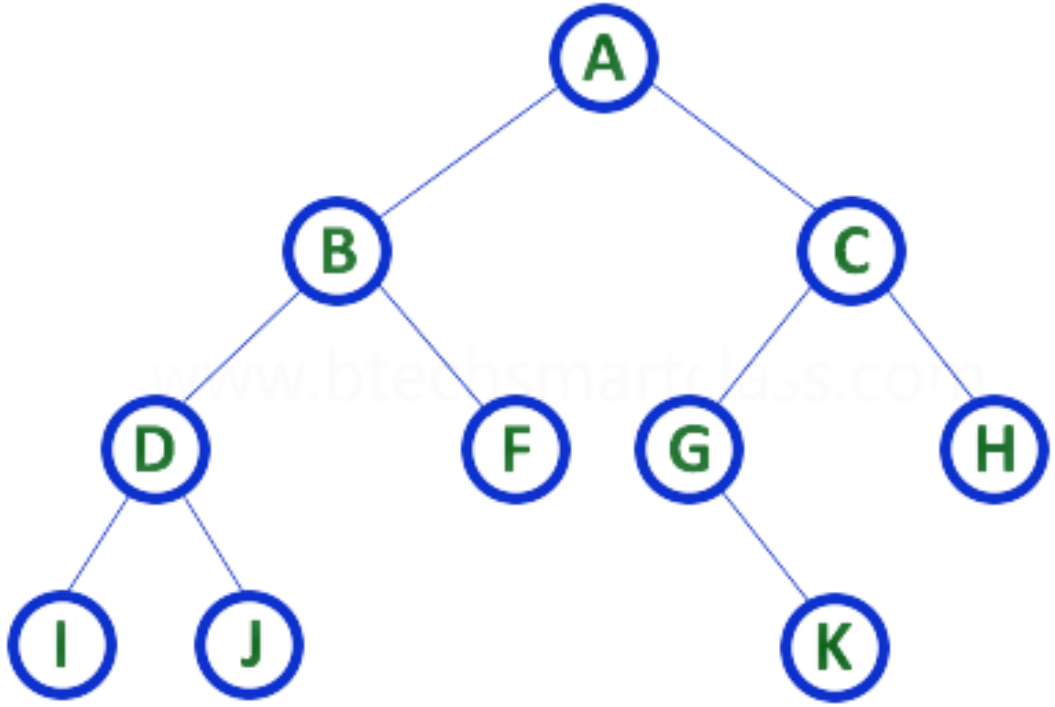


Implementation of Binary Tree

- Binary tree can be implemented through
  - Array
  - LinkedList

Implementation using Array

- In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.



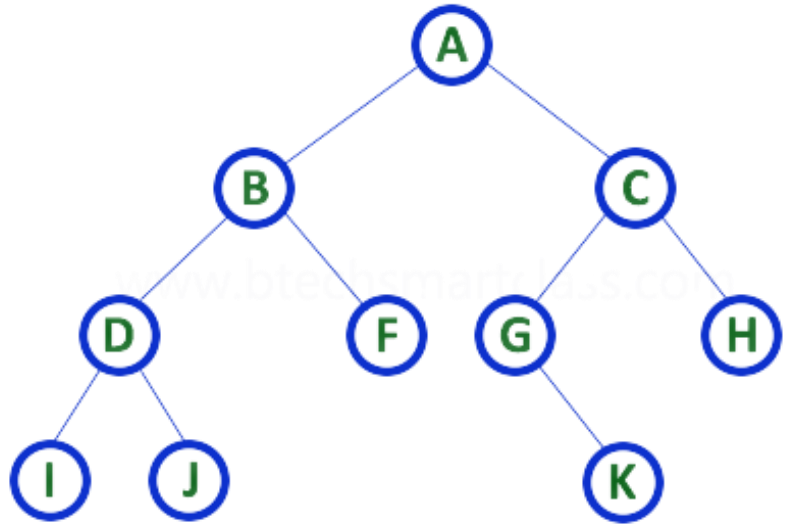
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

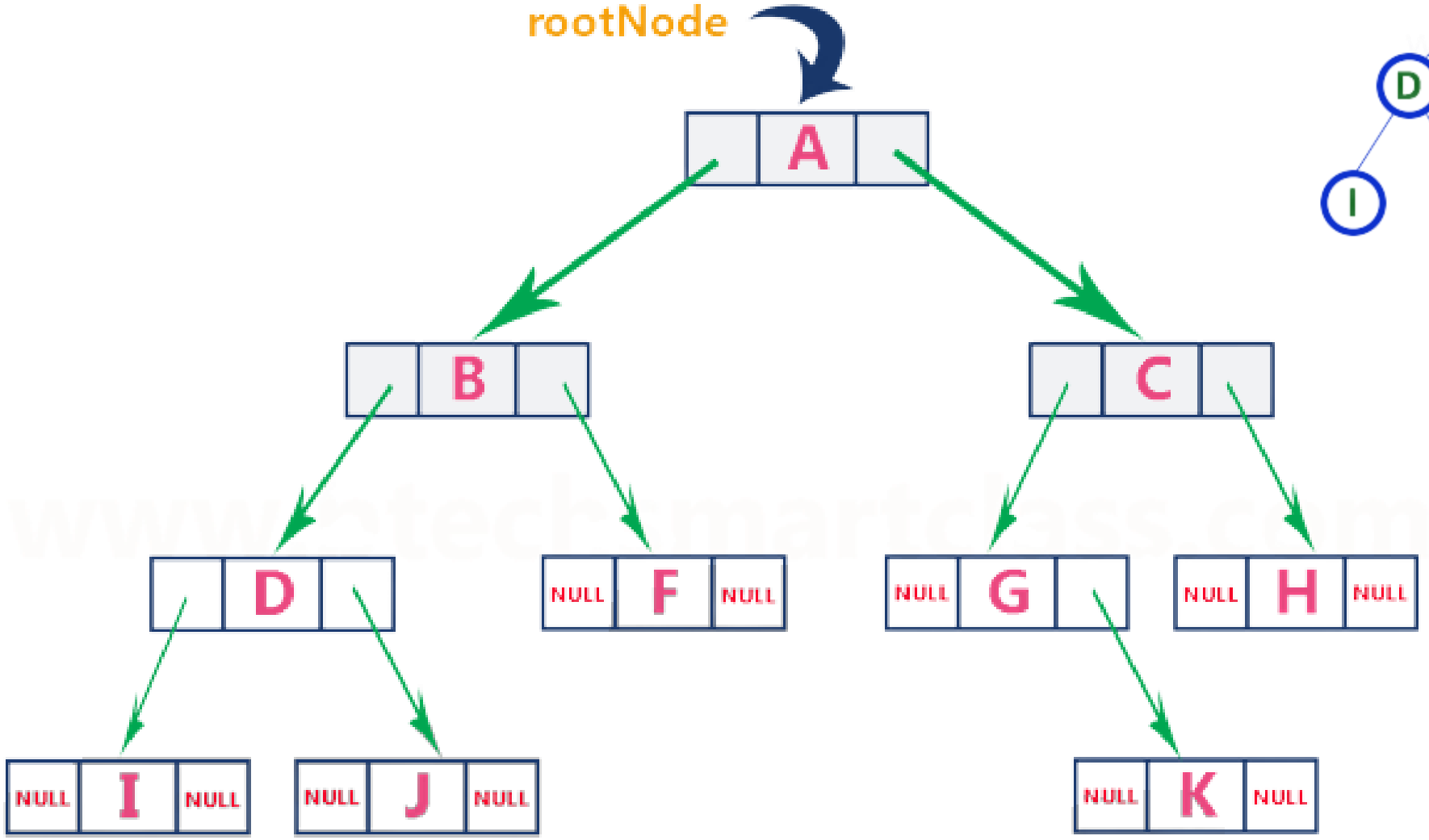
- To represent a binary tree of depth '**n**' using array representation, we need one dimensional array with a maximum size of **2n + 1**.

Implementation using LinkedList

- We use a double linked list to represent a binary tree.
- In a double linked list, every node consists of three fields.
- First field for storing left child address, second for storing actual data and third for storing right child address.

Implementation using LinkedList





Binary Tree Operation(s)

- Create: creates an empty tree.
- Insert: insert a node in the tree.
- Delete: deletes a node from the tree.
- Search: Searches for a node in the tree.

- In-order: in-order traversal of the tree.
- Preorder: pre-order traversal of the tree.
- Post-order: post-order traversal of the tree.

Insert Node in BST tree :

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Create a newNode with given value and set its **left** and **right** to **NULL**.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is **Empty**, then set **root** to **newNode**.

Step 4 - If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

Step 5 - If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.

Step 6- Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).

Step 7 - After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Node Insertion

```
#include<iostream>
#include<stdlib.h>
using namespace std;

//declaration for new bst node
struct bstnode
{
    int data;
    struct bstnode *left, *right;
};

// create a new BST node
struct bstnode *newNode(int key)
{
    struct bstnode *temp =  new struct bstnode();
    temp->data = key;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
/* insert a new node in BST with given key  */
struct bstnode* insert(struct bstnode* node, int key)
{
    //tree is empty; return a new node
    if (node == NULL) return newNode(key);

    //if tree is not empty find the proper place to insert new node
    if (key < node->data)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    //return the node pointer
    return node;
}
```

```
// main program
int main()
{
    struct bstnode *root = NULL;

    root = insert(root, 40);
    root = insert(root, 30);
    root = insert(root, 60);
    root = insert(root, 65);
    root = insert(root, 70);
```

Delete Node in a BST:

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Delete Node: Contd.

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1 - **Find** the node to be deleted using **search operation**

Step 2 - Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1 - **Find** the node to be deleted using **search operation**

Step 2 - If it has only one child then create a link between its parent node and child node.

Step 3 - Delete the node using **free** function and terminate the function.

Delete Node: Contd.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1 - **Find** the node to be deleted using **search operation**

Step 2 - If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.

Step 3 - **Swap** both **deleting node** and node which is found in the above step.

Step 4 - Then check whether deleting node came to **case 1** or **case 2** or else goto step 2

Step 5 - If it comes to **case 1**, then delete using case 1 logic.

Step 6- If it comes to **case 2**, then delete using case 2 logic.

Step 7 - Repeat the same process until the node is deleted from the tree.

```
//function to delete the node with given key and rearrange the root
struct bstnode* deleteNode(struct bstnode* root, int key)
{
    // empty tree
    if (root == NULL) return root;

    // search the tree and if key < root, go for lefmost tree
    if (key < root->data)
        root->left = deleteNode(root->left, key);

    // if key > root, go for rightmost tree
    else if (key > root->data)
        root->right = deleteNode(root->right, key);

    // key is same as root
    else
        {
        // node with only one child or no child
            if (root->left == NULL)
            {
                struct bstnode *temp = root->right;
                free(root);
                return temp;
            }
            else if (root->right == NULL)
            {
                struct bstnode *temp = root->left;
                free(root);
                return temp;
            }

        // node with both children; get successor and then delete the node
                struct bstnode* temp = minValueNode(root->right);

            // Copy the inorder successor's content to this node
                root->data = temp->data;

            // Delete the inorder successor
                root->right = deleteNode(root->right, temp->data);
            }
    return root;
}
```

```
// main program
int main()
{
    struct bstnode *root = NULL;

    root = insert(root, 40);
    root = insert(root, 30);
    root = insert(root, 60);
    root = insert(root, 65);
    root = insert(root, 70);

    cout<<"\nDelete node 40\n";
    root = deleteNode(root, 40);
```

Search Node in a BST:

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

```
void search(struct bstnode* root, int data)
{
    struct bstnode *current = root;

    if (current == NULL)
    {
        cout<<endl<<"No Binary Tree ";
    }
    else
    {
        while( (current->left != NULL) && (current->right != NULL) )
            {
                if (data < current->data)
                {
                    current = current->left;
                }
                else
                {
                    current = current->right;
                }
            }

            if(current->data == data)
            {
                cout<<endl<<"Value Found = "<<current->data;
            }
            else
            {
                cout<<endl<<"Value Not Found";
            }
    }
}
```

```
// main program
int main()
{
    struct bstnode *root = NULL;

    root = insert(root, 40);
    root = insert(root, 30);
    root = insert(root, 60);
    root = insert(root, 65);
    root = insert(root, 70);

    search(root,90);
```

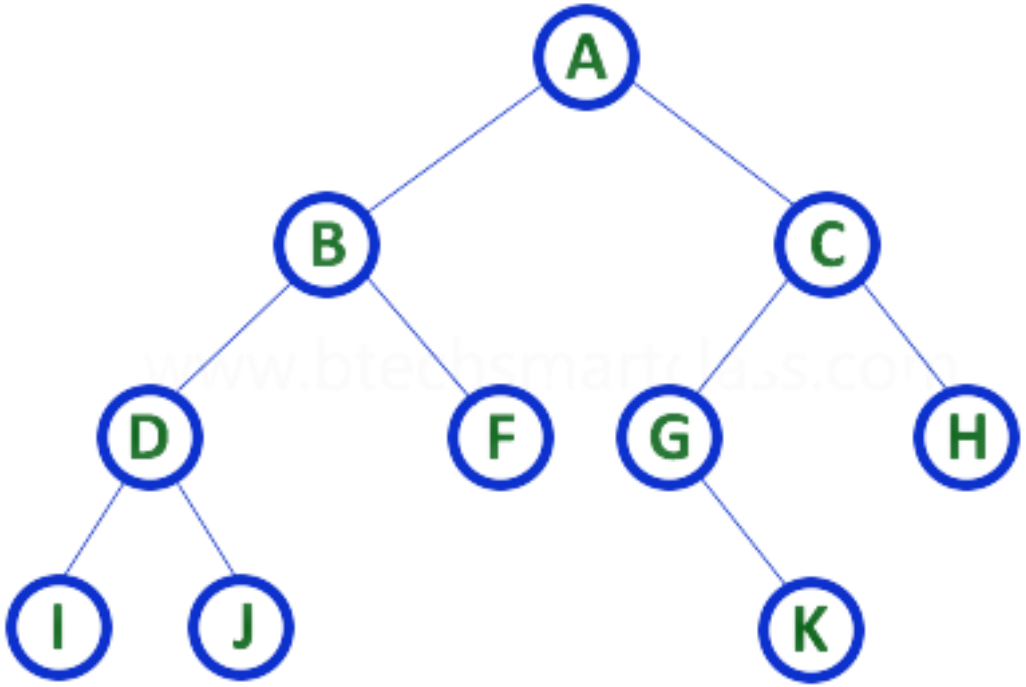
Tree traversal

- There are 3 kinds of traversals that are done typically over a binary search tree. All these traversals have a somewhat common way of going over the nodes of the tree.

- In-order
- Pre-order
- Post-order

In-Order Traversal (leftChild - root - rightChild)

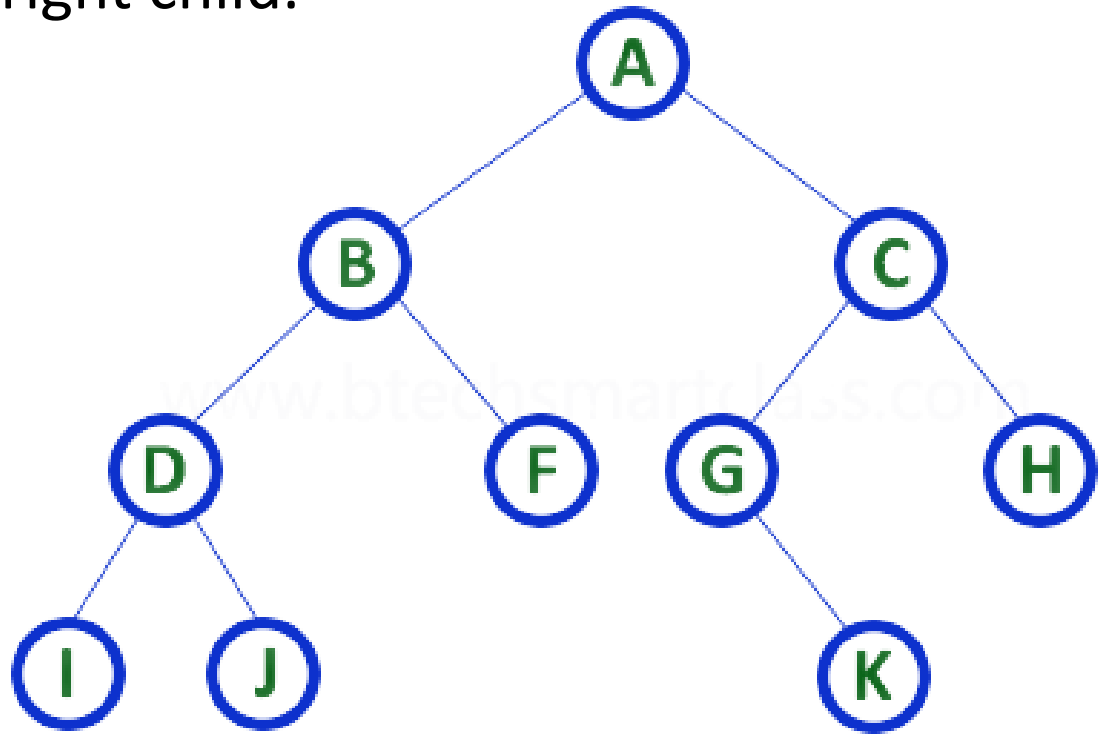
- In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node



- I - D - J - B - F - A - G - K - C - H

Pre-Order Traversal (root - leftChild - rightChild)

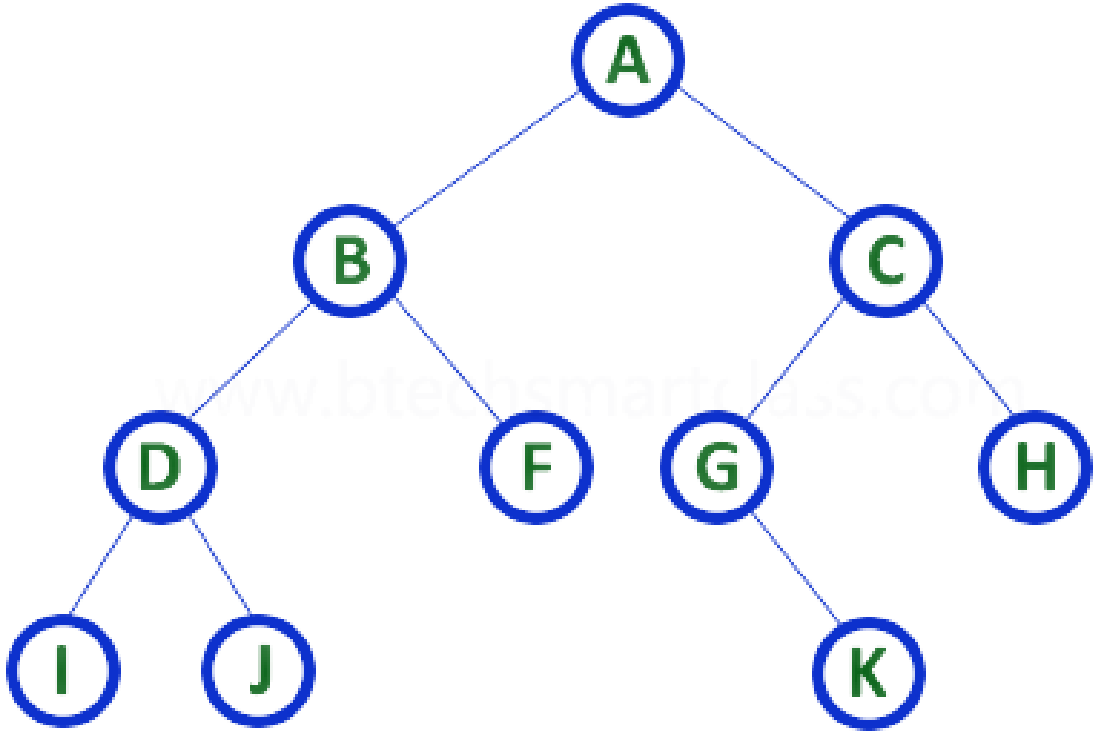
- In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child.



- A - B - D - I - J - F - C - G - K - H

Post-Order Traversal (leftChild - rightChild - root)

- In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child.



- I - J - D - F - B - K - G - H - C - A