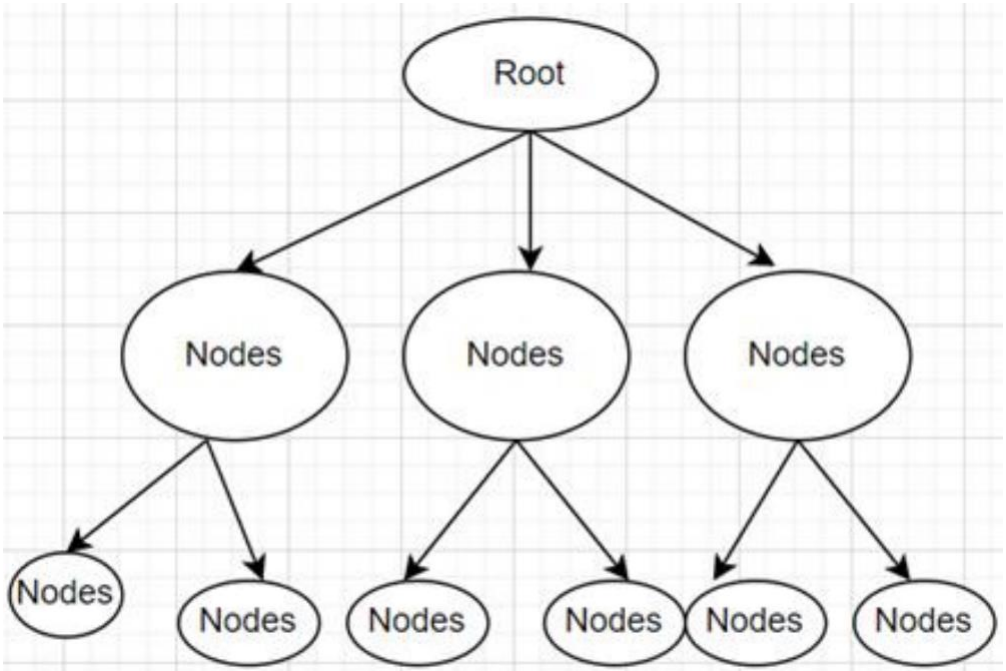
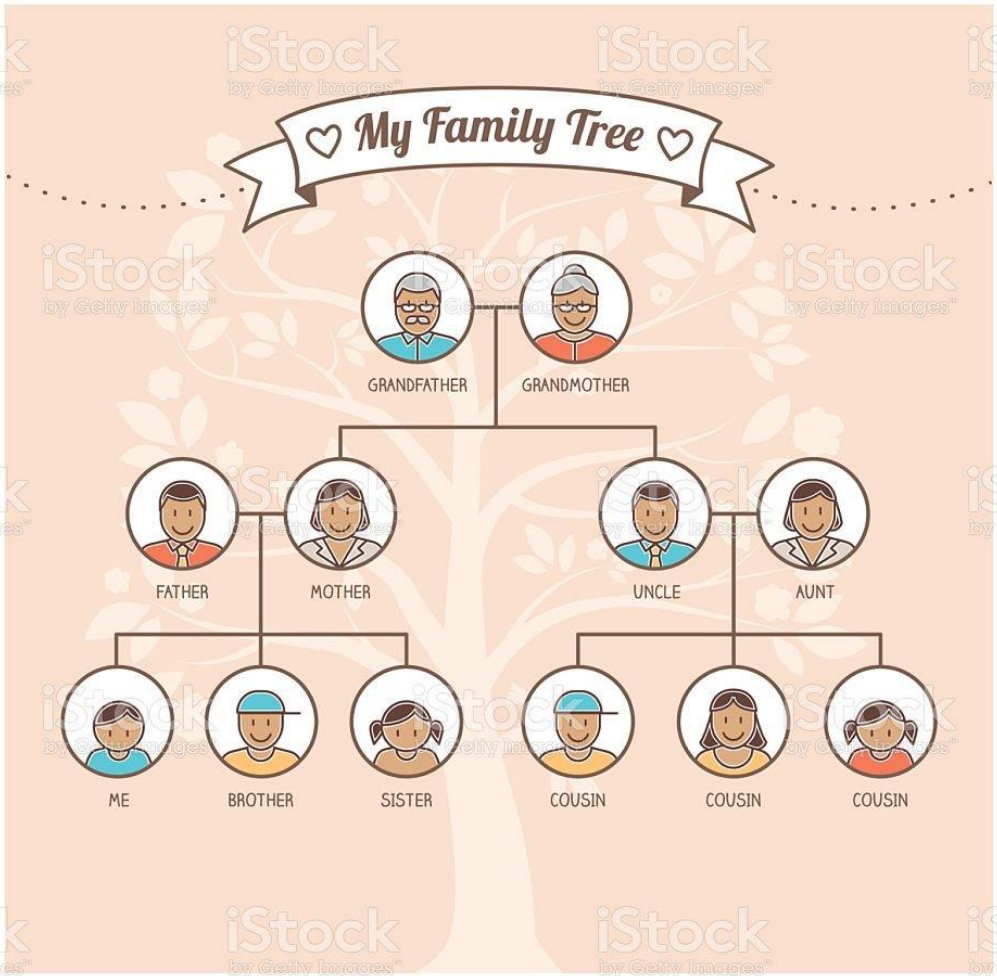
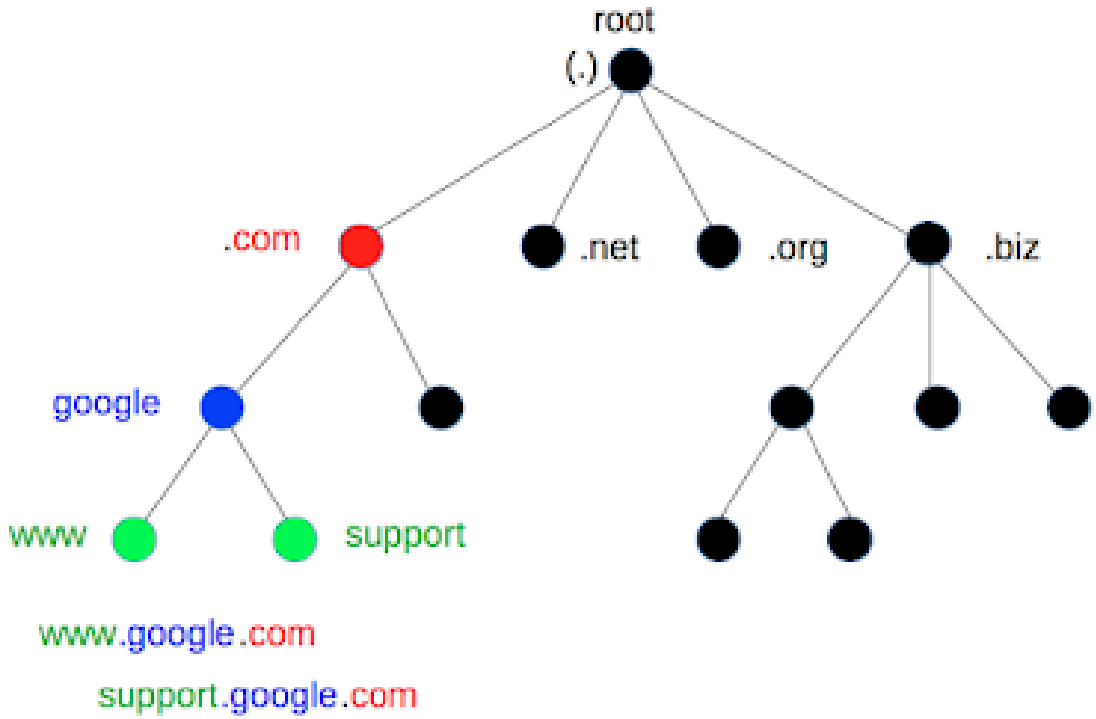
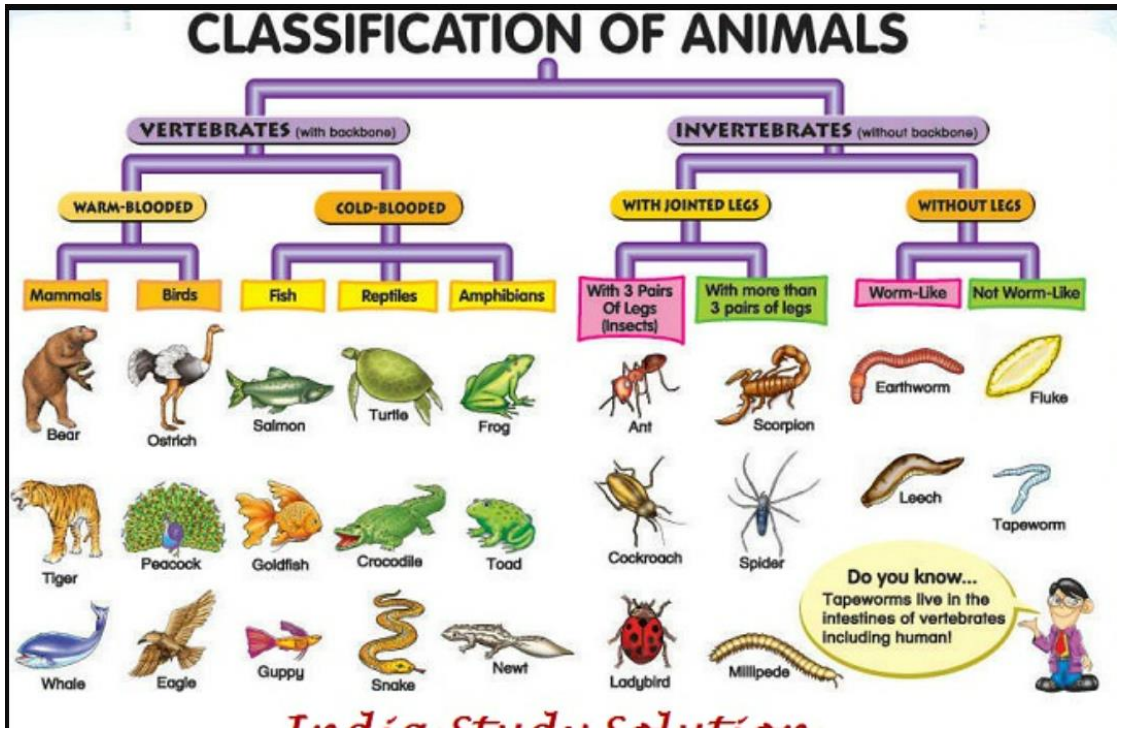
Trees

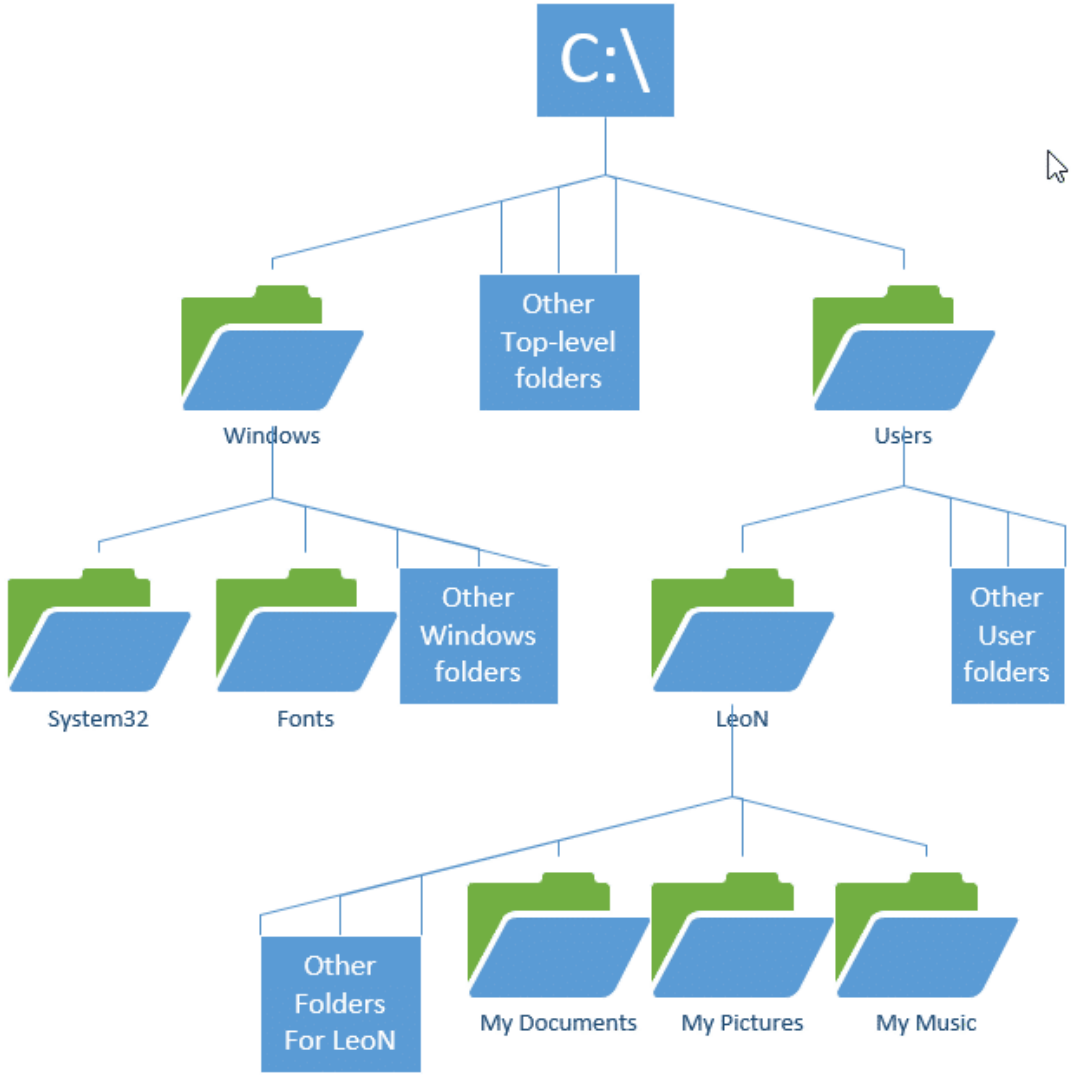
- *Non-Linear Data structure*
- *In hierarchical manner*
- *consists of nodes connected by edges.*
- *Tree looks like*




My Family Tree
GRANDFATHER
GRANDMOTHER
FATHER
MOTHER
UNCLE
AUNT
ME
BROTHER
SISTER
COUSIN
COUSIN
COUSIN


root
(.)
.com
.net
.org
.biz
google
www
support
www.google.com
support.google.com


CLASSIFICATION OF ANIMALS
VERTEBRATES (with backbone)
INVERTEBRATES (without backbone)
WARM-BLOODED
COLD-BLOODED
WITH JOINTED LEGS
WITHOUT LEGS
Mammals
Birds
Fish
Reptiles
Amphibians
With 3 Pairs Of Legs (Insects)
With more than 3 pairs of legs
Worm-Like
Not Worm-Like
Bear
Ostrich
Salmon
Turtle
Frog
Ant
Scorpion
Earthworm
Fluke
Tiger
Peacock
Goldfish
Crocodile
Toad
Cockroach
Spider
Leech
Tapeworm
Whale
Eagle
Guppy
Snake
Newt
Ladybird
Millipede
Do you know...
Tapeworms live in the intestines of vertebrates including human!

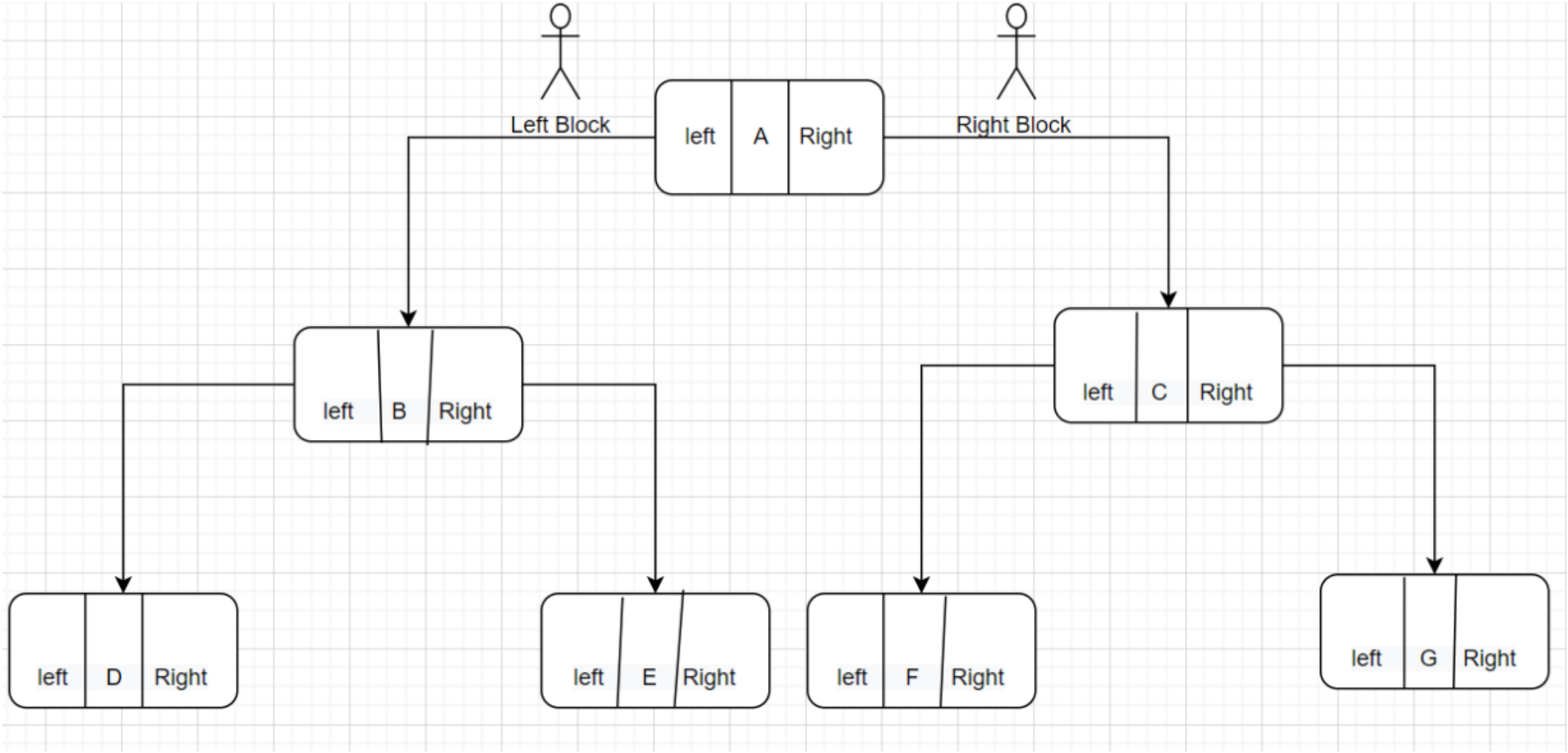

C:\
Windows
Other Top-level folders
Users
System32
Fonts
Other Windows folders
LeoN
Other User folders
Other Folders For LeoN
My Documents
My Pictures
My Music

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.*

- *In tree, we store the date in hierarchical manner which is non-linear, which provides*
    - *moderate access/search (quicker than Linked List and slower than arrays).*
    - *moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).*

Logical Representation of the Tree



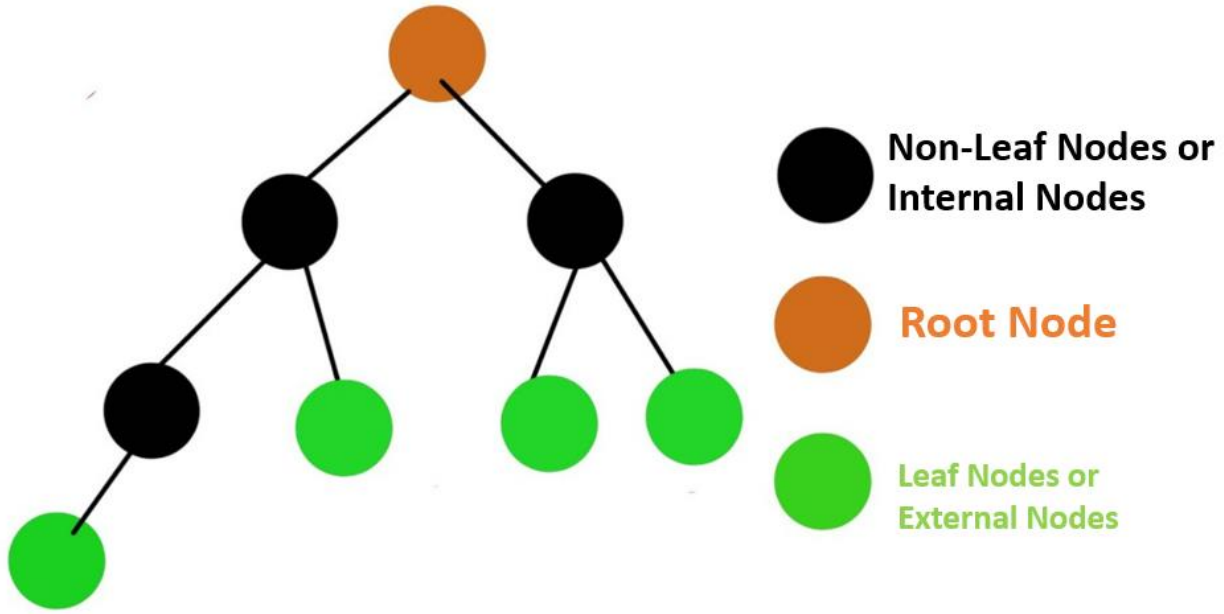
Tree Terminologies

Node

- *A node holds data and address, that creates the link to the other node.*

Root Node:

- *Starting Node, Topmost Node*
- *Only one node can be a root node in a Tree*
- *A Node without parent*



Leaf Node

- *Last Node, Bottom most Node*
- *A Node without children*
- *Also called as external nodes*

Non-Leaf Node

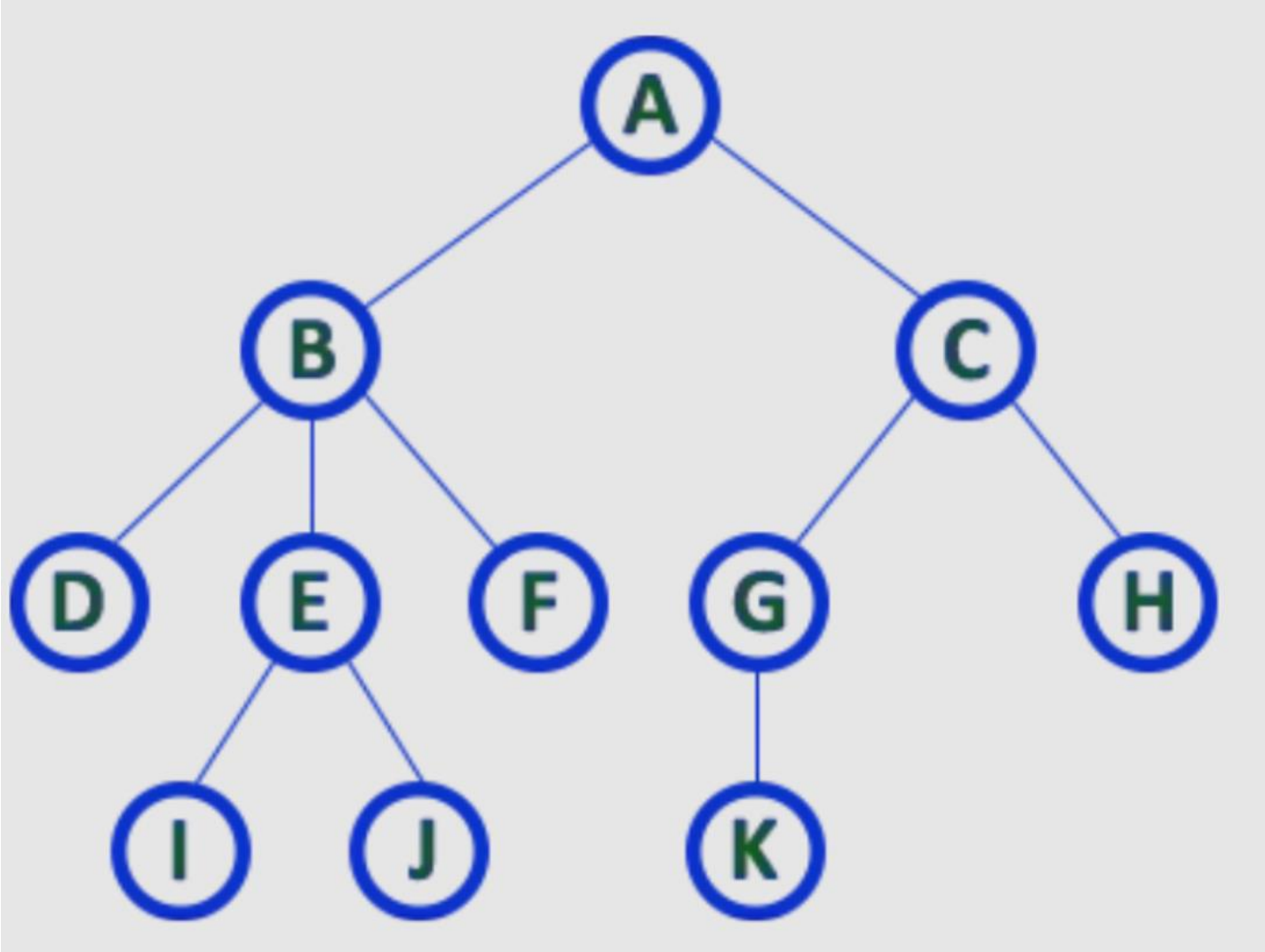
- *In between Nodes, Intermediate Nodes*
- *A node with at least one child and parent*
- *Also called as internal nodes (includes root node)*

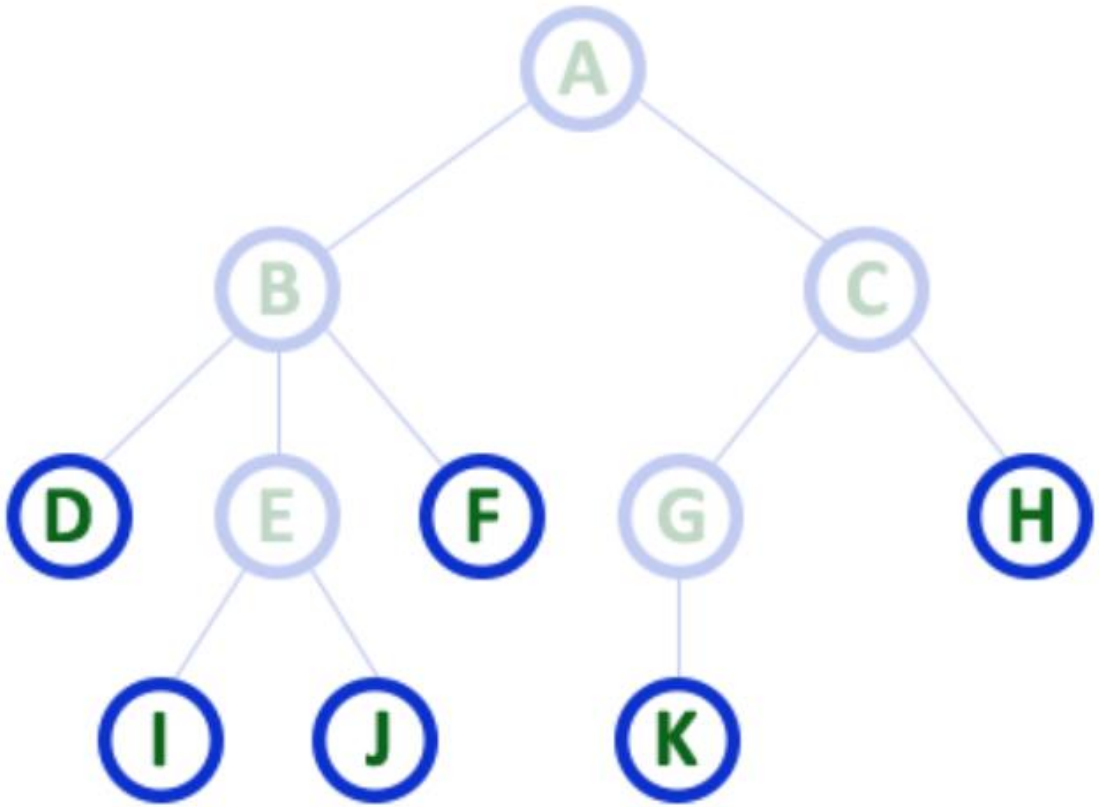
Edge

- *It is the link between any two nodes.*

In any tree with 'N' nodes there will be maximum of 'N-1' edges

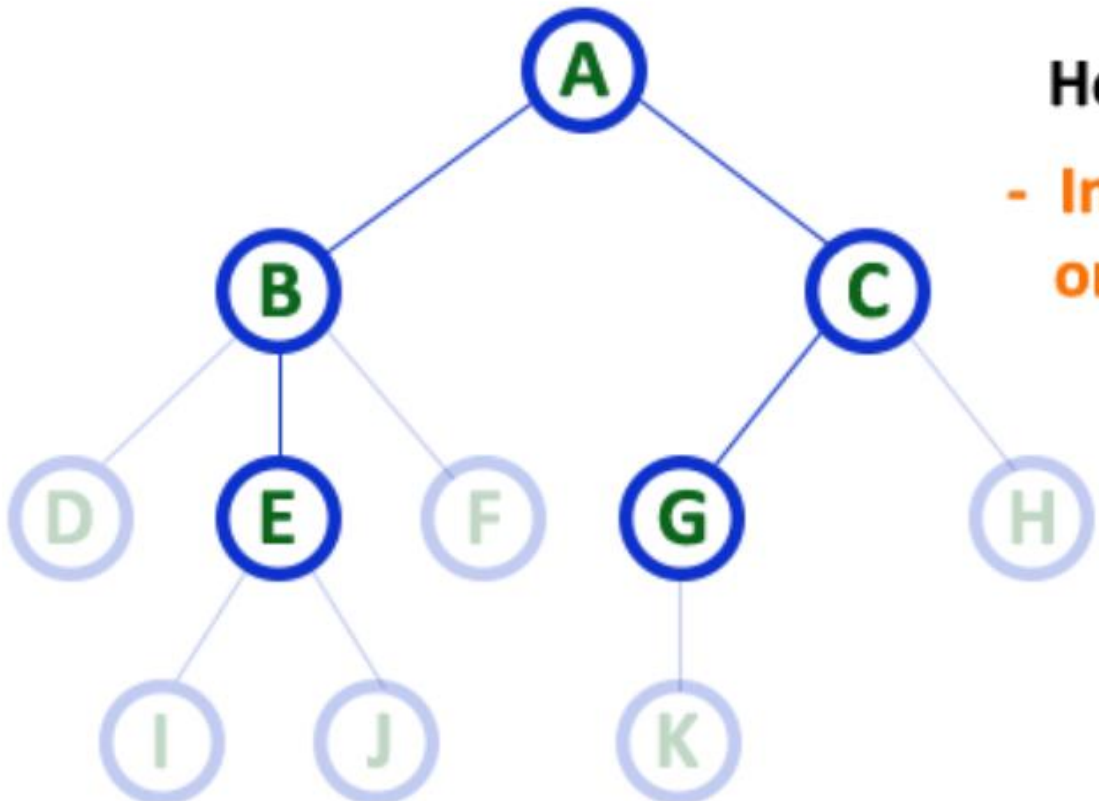
Example





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

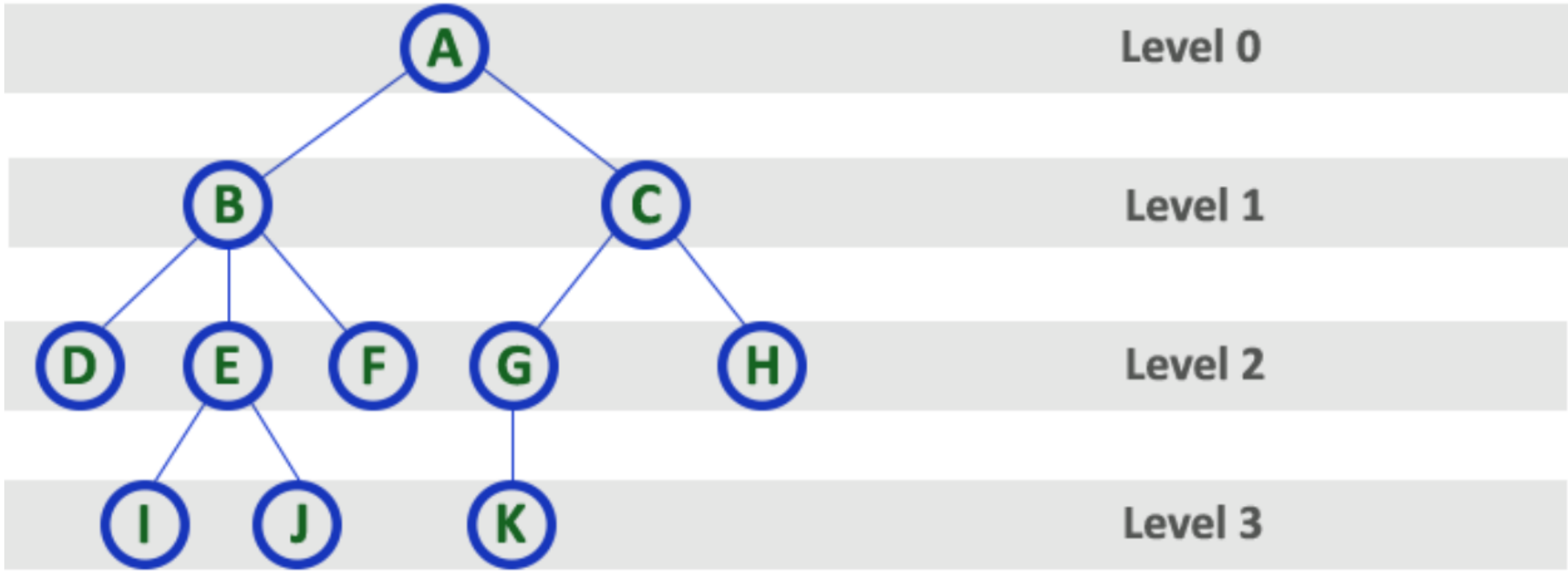
- **In any tree the node which does not have children is called 'Leaf'**
- **A node without successors is called a 'leaf' node**

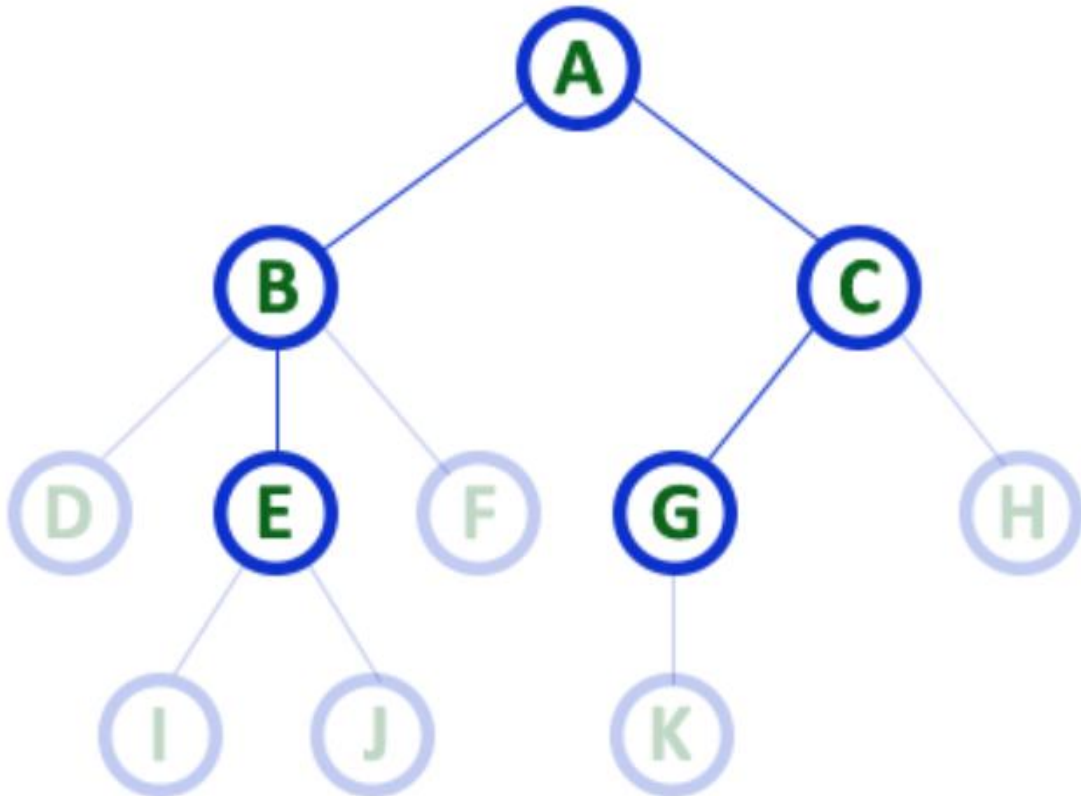


Here A, B, C, E & G are Internal nodes

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

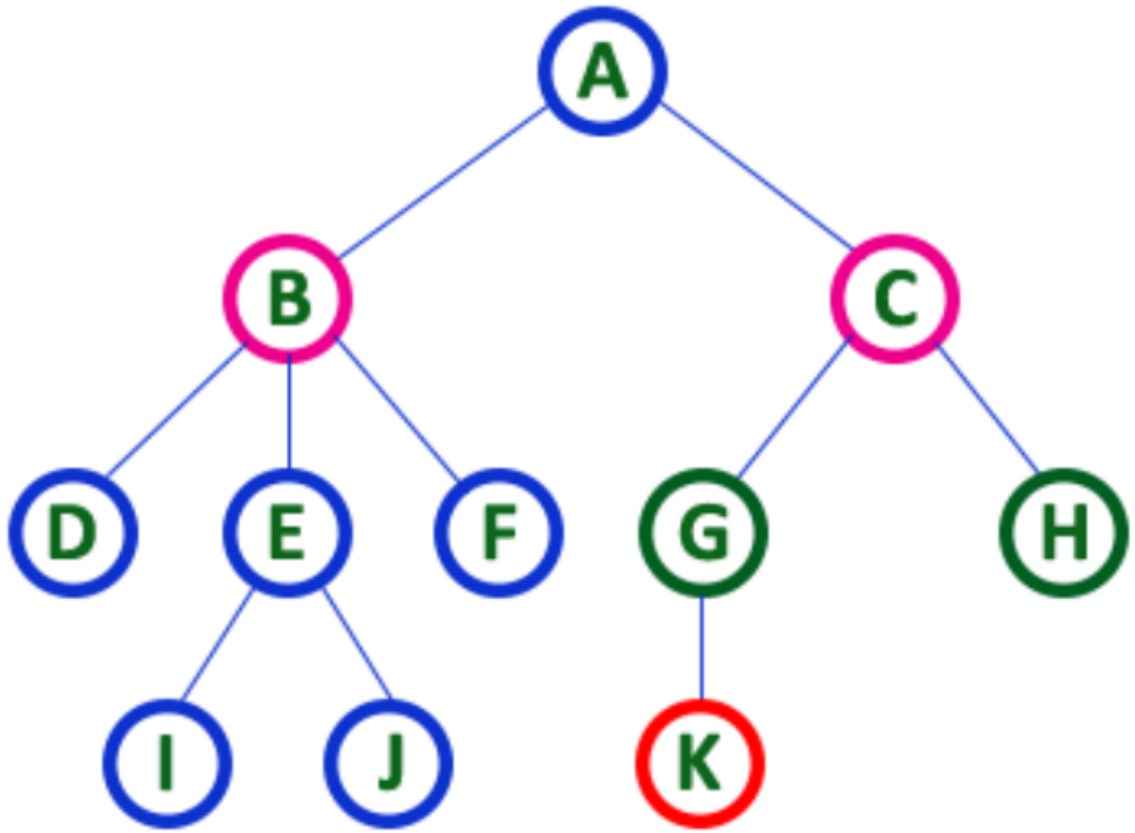
- **Every non-leaf node is called as 'Internal' node**


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



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'**

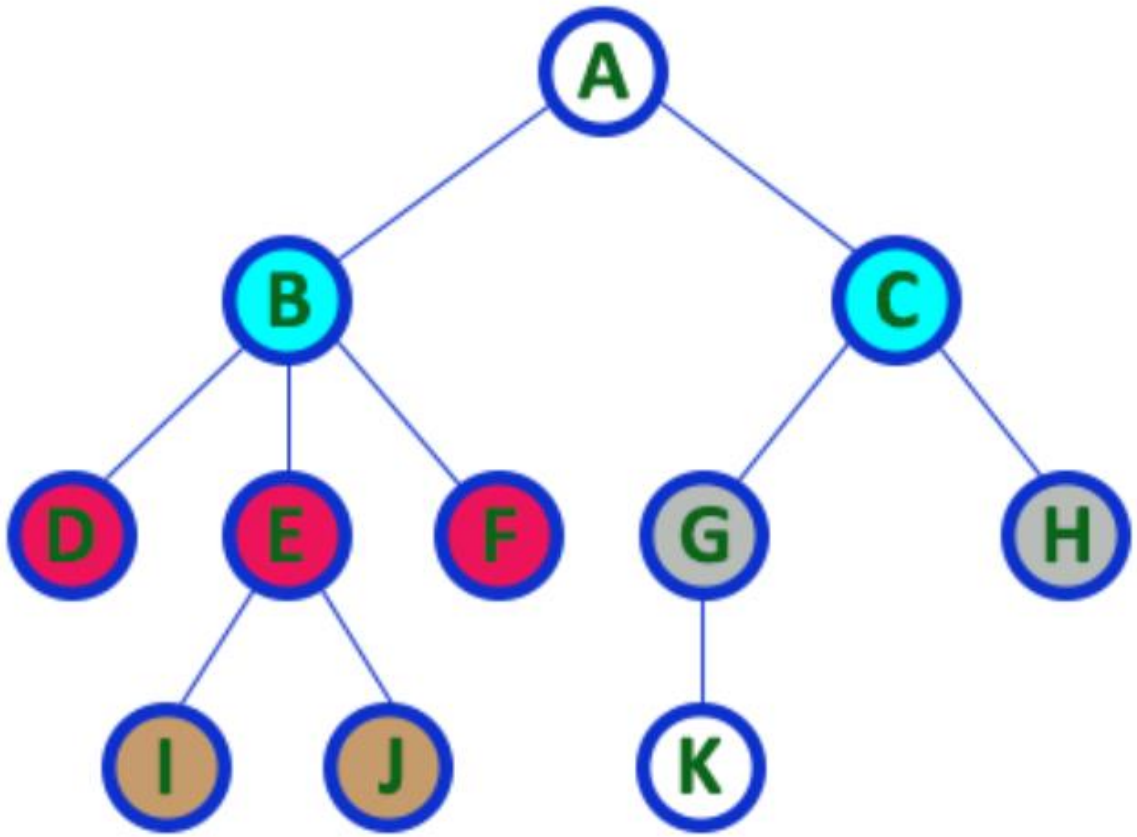


Here B & C are Children of A

Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node



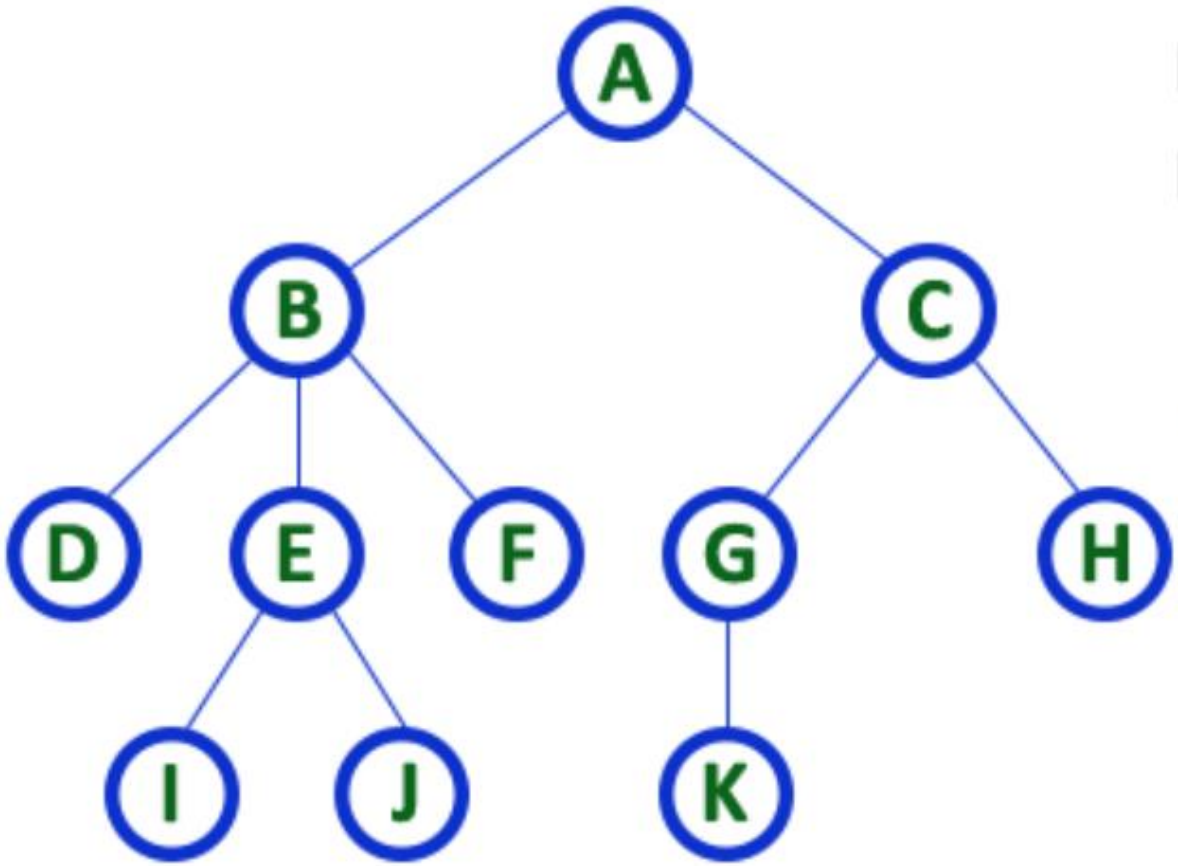
Here B & C are Siblings

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 Degree of B is 3

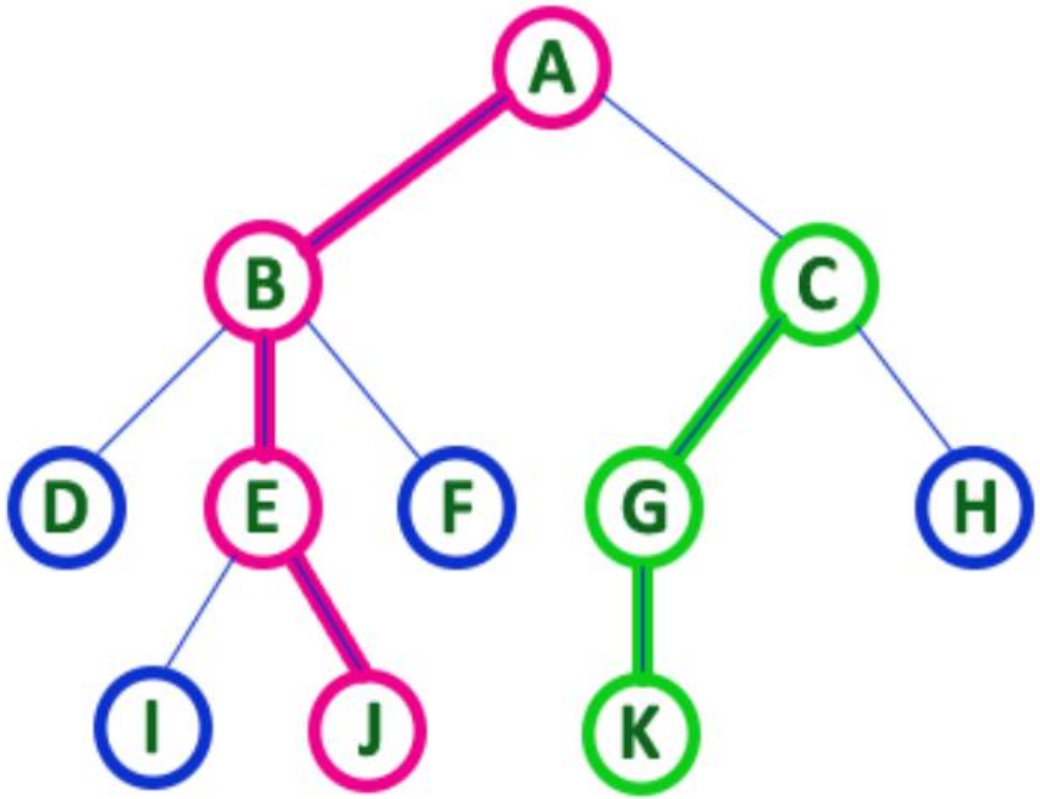
Here Degree of A is 2

Here Degree of F is 0

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

Path

- *The sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes.*
- *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

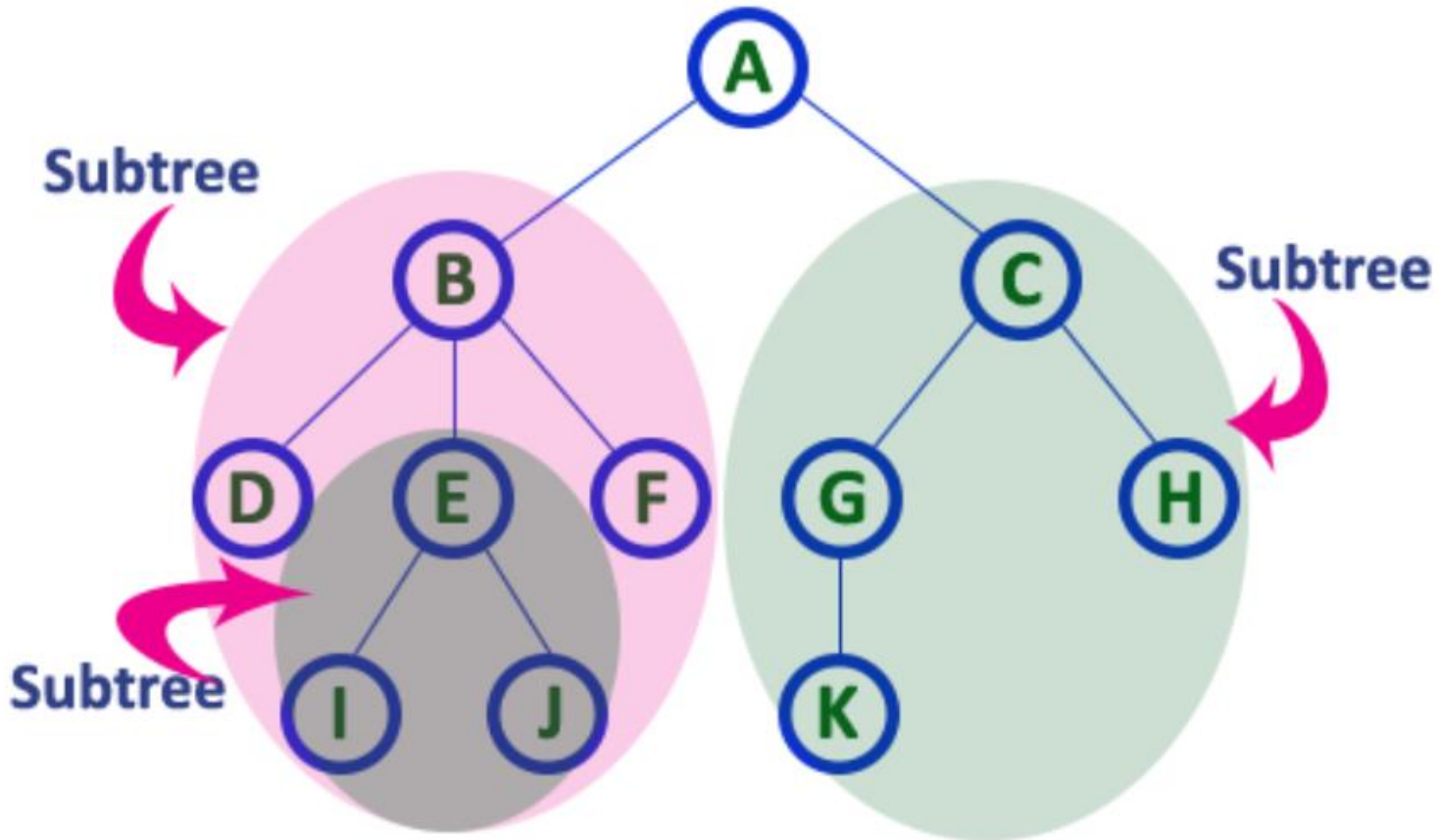
A - B - E - J

Here, 'Path' between C & K is

C - G - K

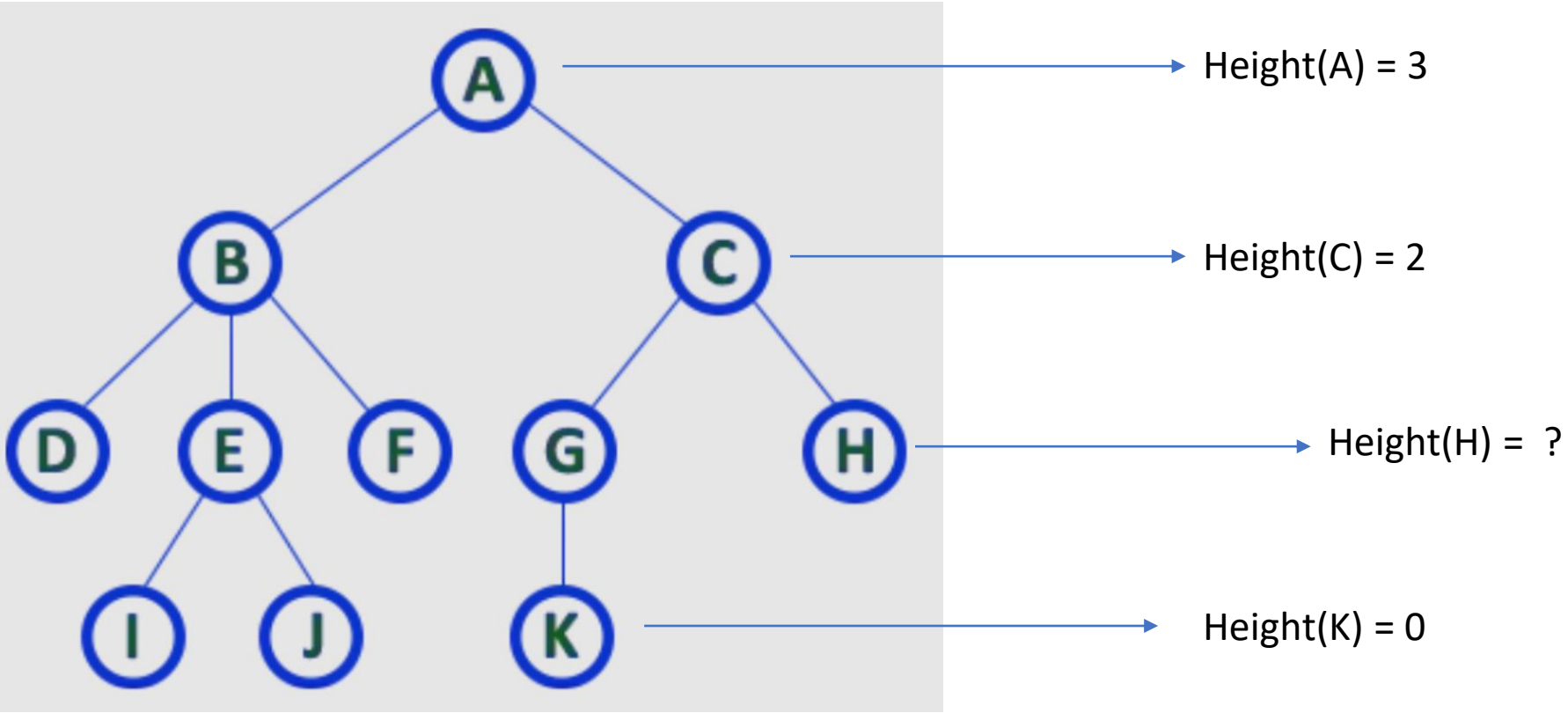
Sub Tree

- *Each child from a node forms a subtree recursively.*
- *Every child node will form a subtree on its parent node.*



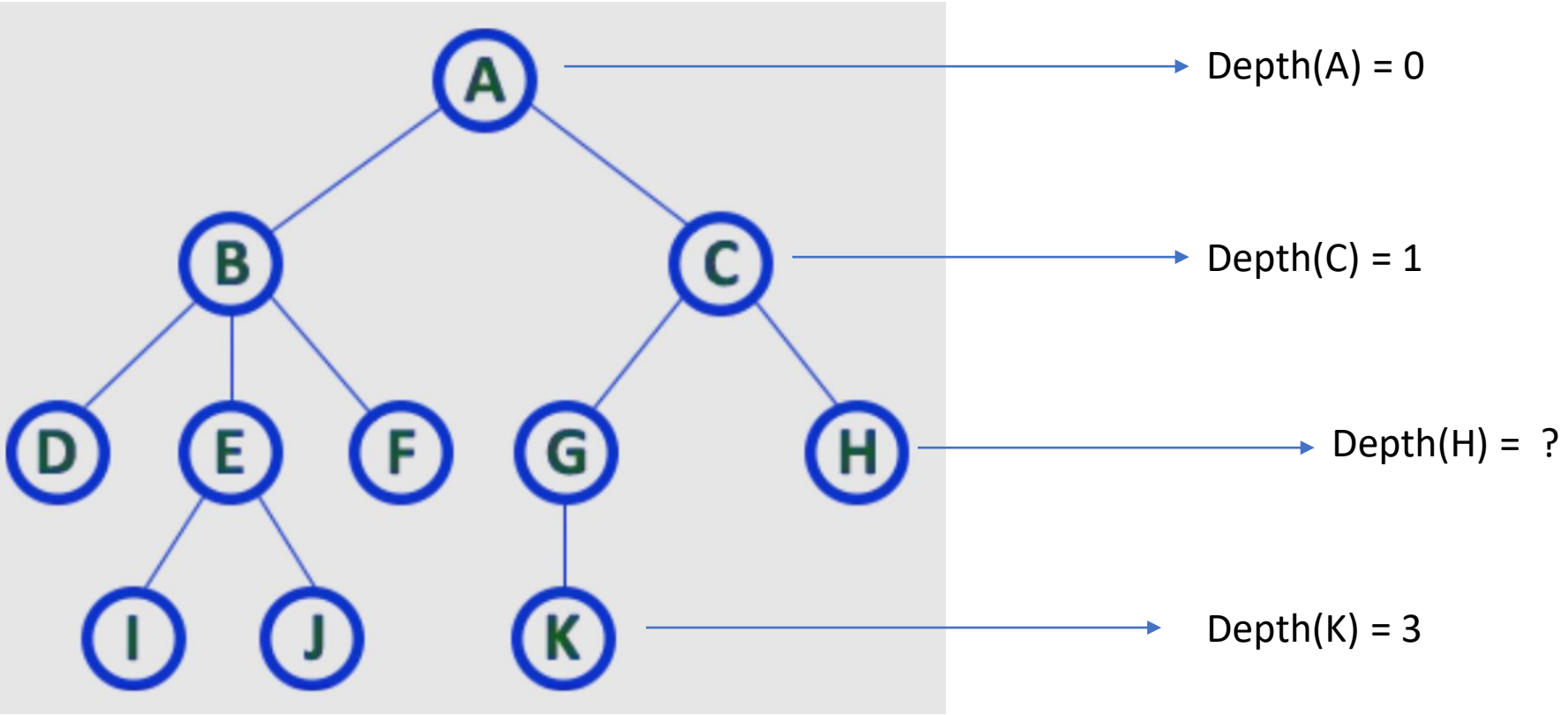
Height of the node

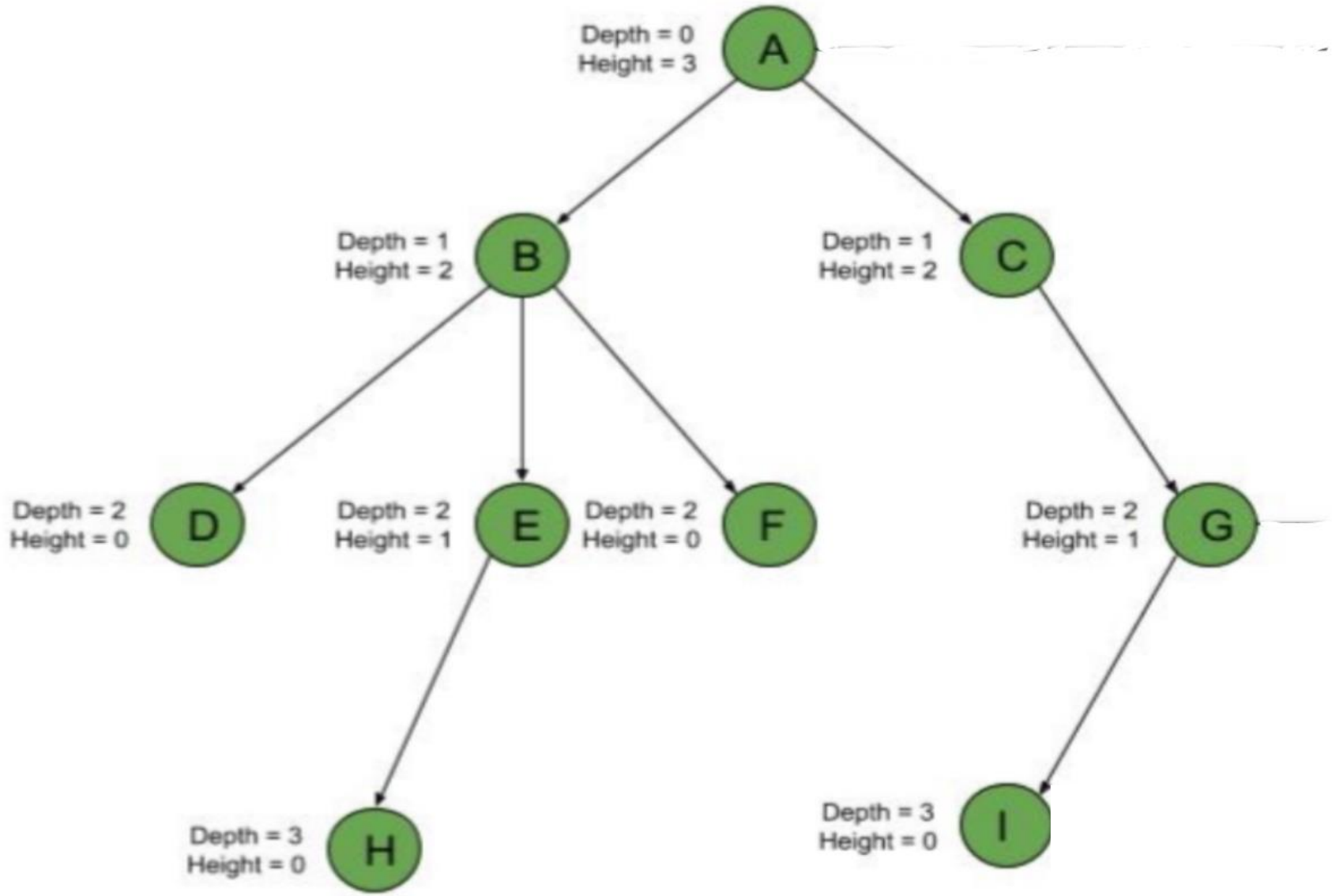
- *The height of the node is the* ***number of edges from the node to the deepest leaf node***
  - ***To the leaf!!***
  - ***Height of the leaf nodes are 0***



Depth of the node

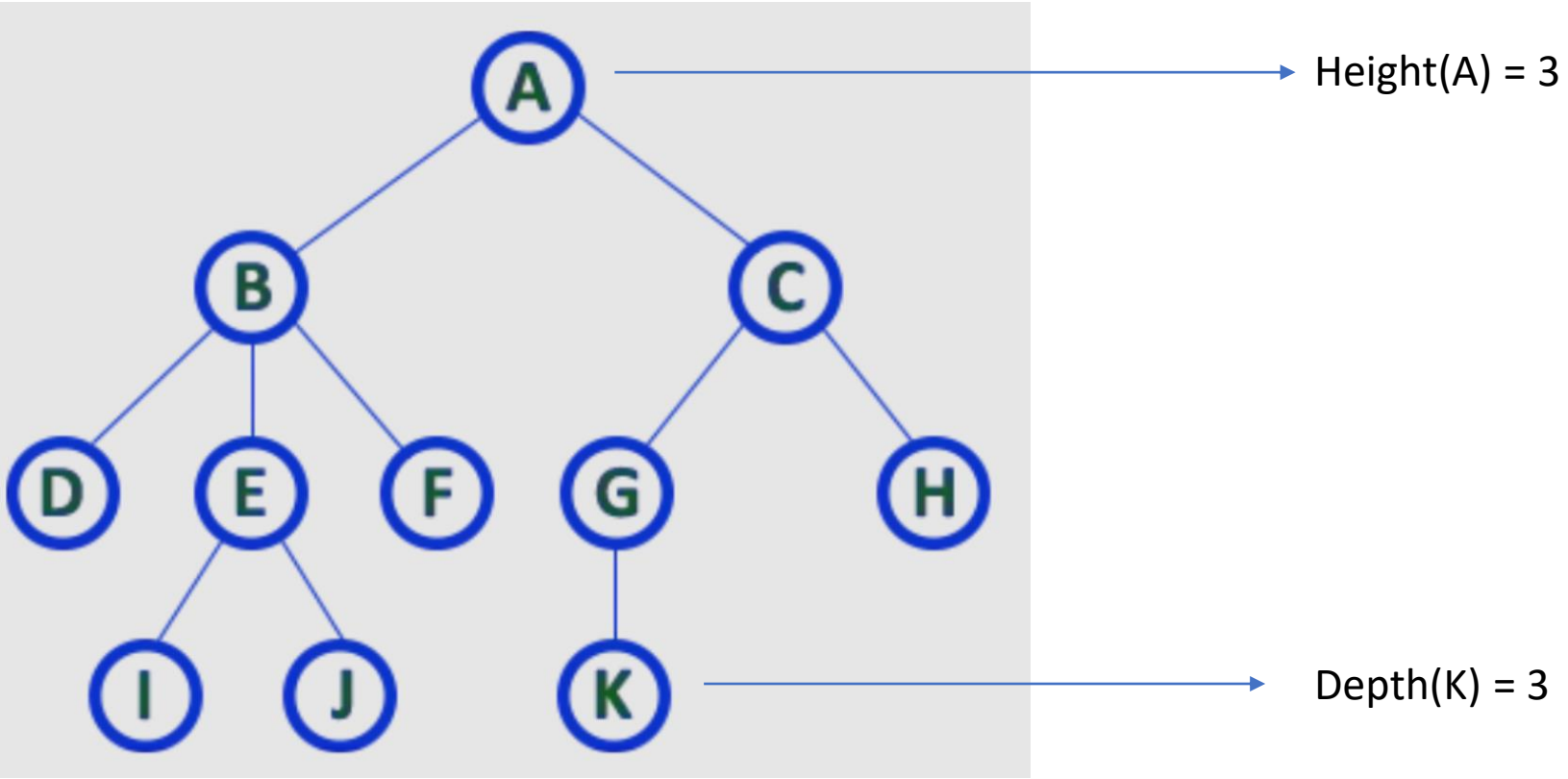
- *The depth of the node is the* ***number of edges from the particular node to the root node***
  - ***To the root!!***
  - ***Depth of the root node is 0***




Depth = 0
Height = 3
A
Depth = 1
Height = 2
B
Depth = 1
Height = 2
C
Depth = 2
Height = 0
D
Depth = 2
Height = 1
E
Depth = 2
Height = 0
F
Depth = 2
Height = 1
G
Depth = 3
Height = 0
H
Depth = 3
Height = 0
I

Height of the Tree *– The height of a tree is a height of the root.*

Depth of the Tree - *The depth of tree is the depth of the deepest node (leaf node).*



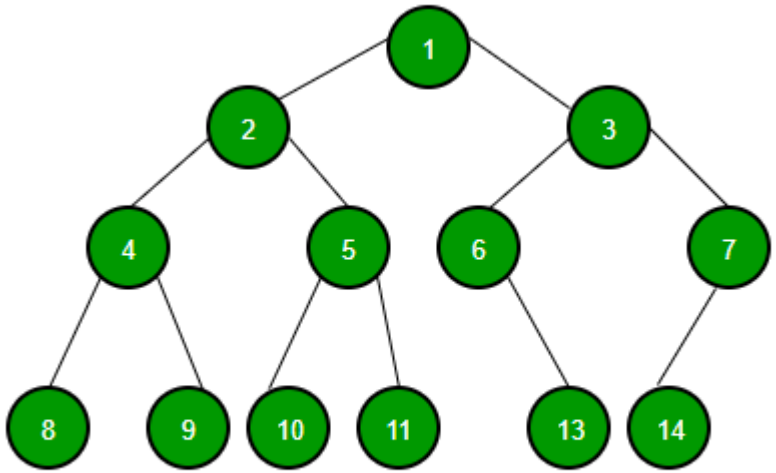
Is height and Depth of a tree equal??

- **Yes, height and depth of a tree is equal...**
- but height and depth of a node is not equal because...
- the height is calculated by traversing from the given node to the deepest possible leaf.
- depth is calculated from traversal from root to the given node.....

Binary Tree

Binary Tree can have 2 children for each element.

- Right Child
- Left Child

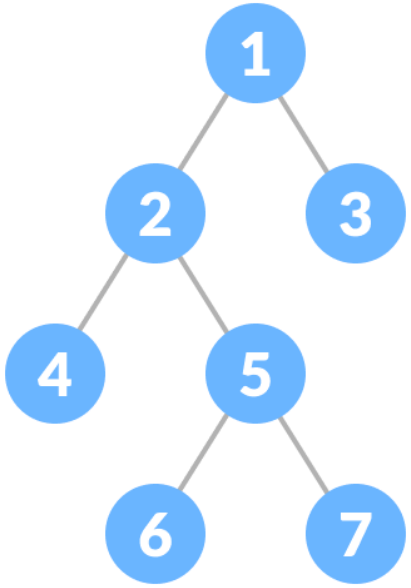


Types of Binary Tree

1. Full Binary Tree
2. Perfect Binary Tree
3. Complete Binary Tree
4. Degenerate or Pathological Tree
5. Balanced Binary Tree
6. Skewed Binary Tree

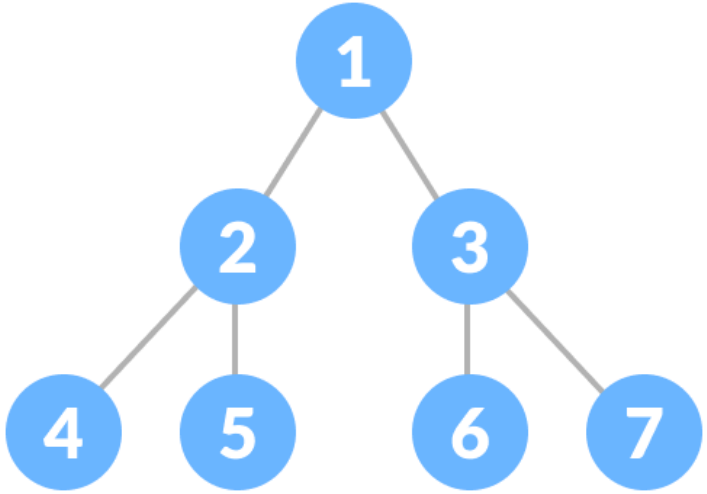
Full Binary Tree

- Every parent node/internal node has either two or no children



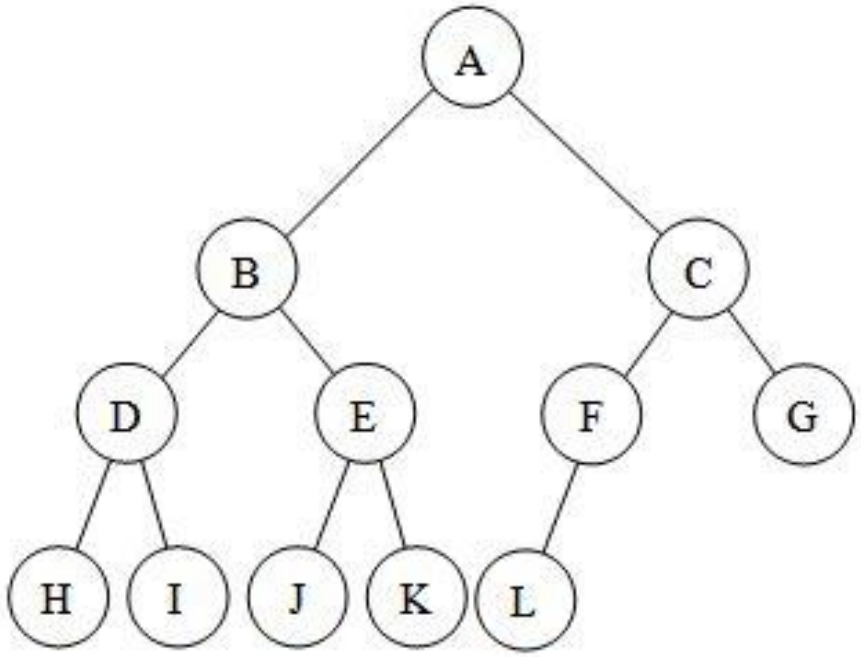
Perfect Binary Tree

- Every internal node has exactly two child nodes and all the leaf nodes are at the same level.



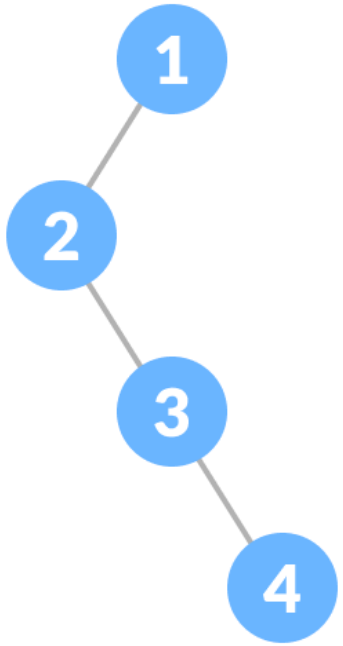
Complete Binary Tree

- Completely filled at every level possible except at last.
- The filling of the leaf node must start from the leftmost side.
- It is not mandatory that the last leaf node must have the right sibling.



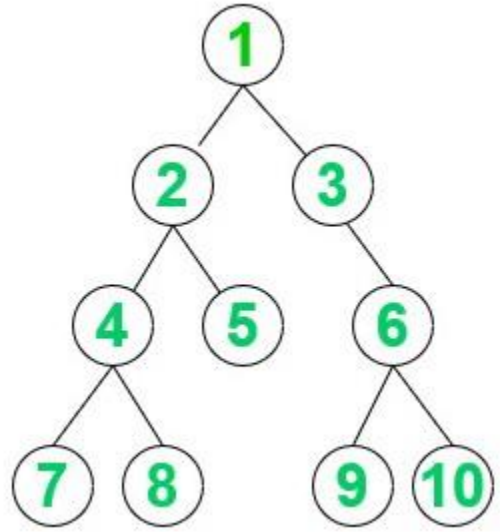
Degenerate Binary Tree

- Tree having a single child either left or right.



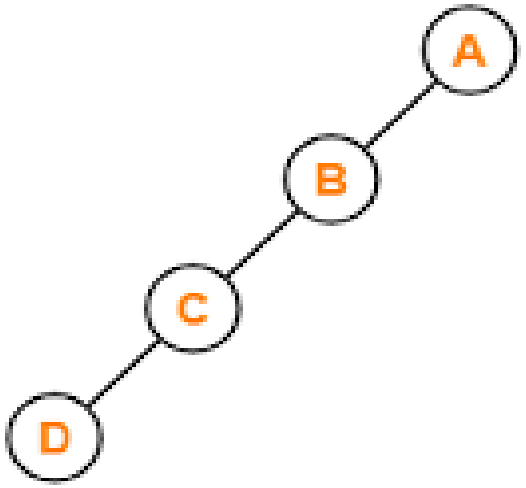
Balanced Binary Tree

- Balance factor = height of left sub tree – height of right sub tree
- 1, 0, -1 -> Tree is Balanced

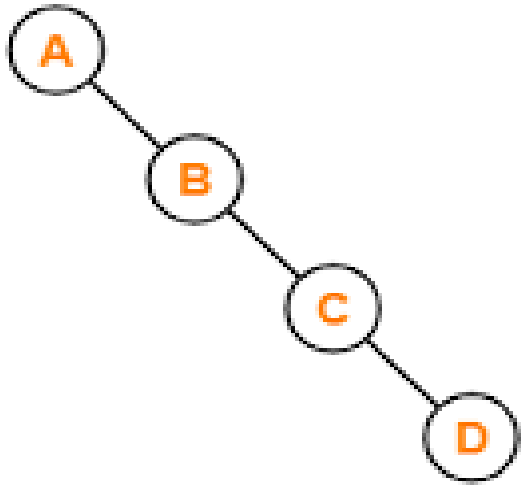


Skewed Binary Tree

- Tree is either dominated by the left nodes or the right nodes.
- Types:
  - Right Skewed Binary Tree
  - Left Skewed Binary Tree



Left Skewed Binary Tree

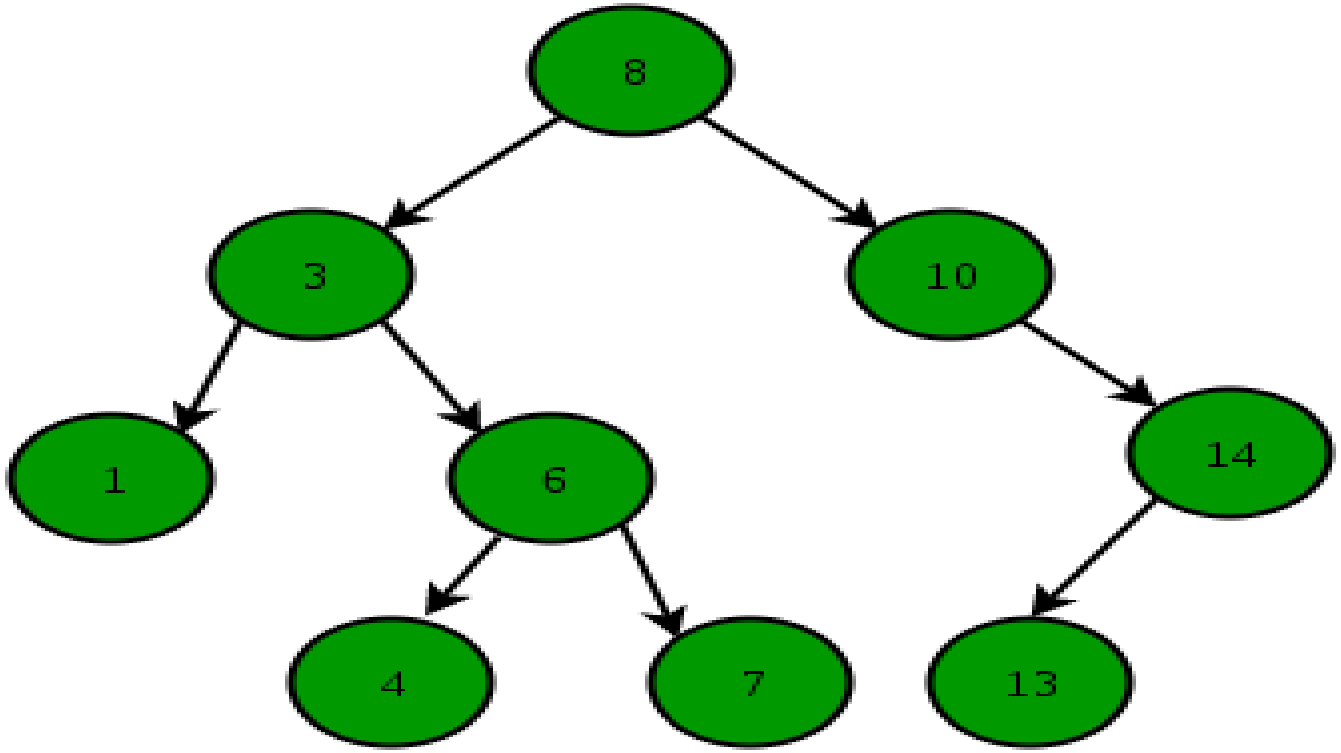


Right Skewed Binary Tree

Binary Search Tree

Binary Search Tree can have 2 children for each element.

- 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.



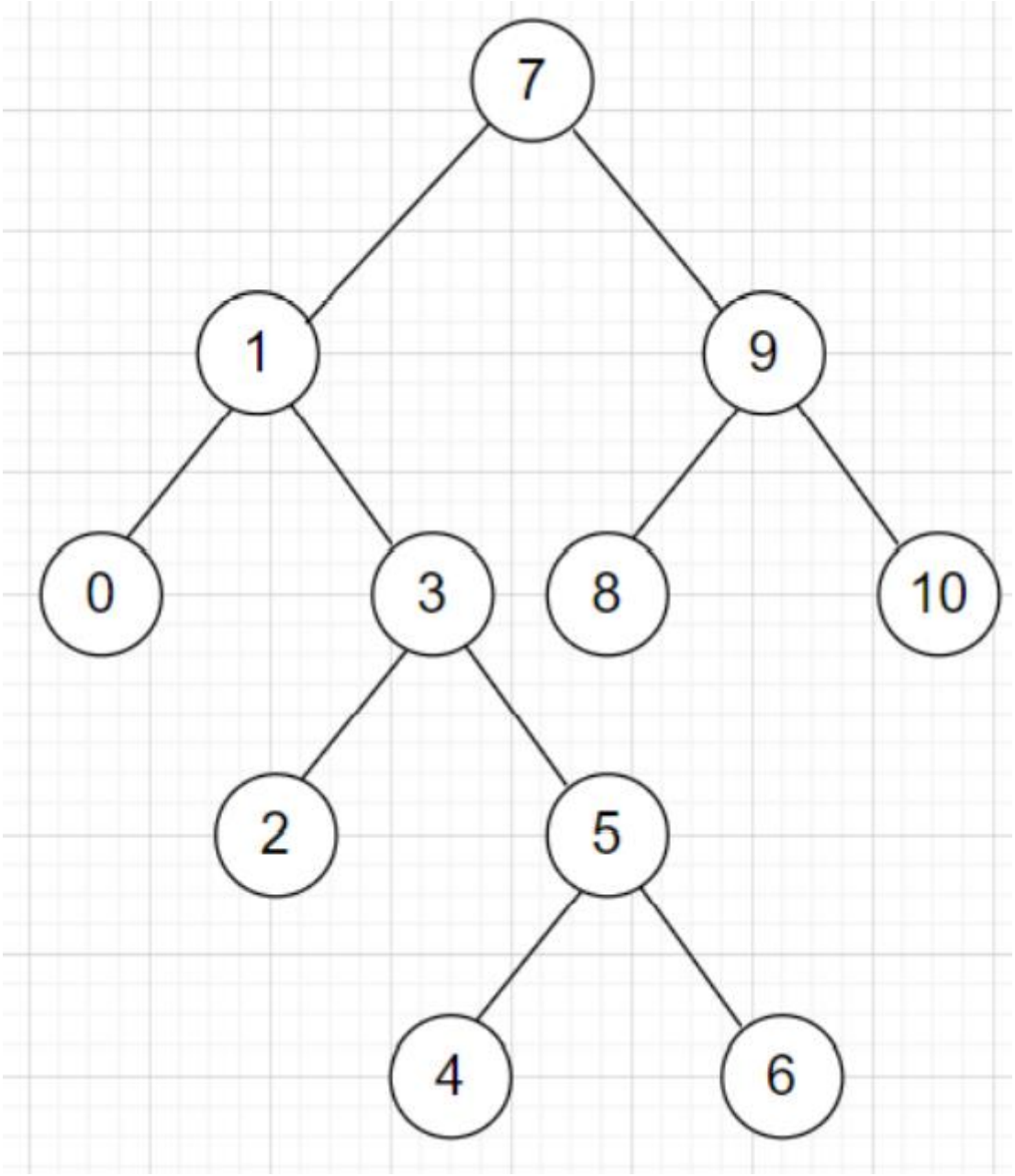
Basic operations on a BST

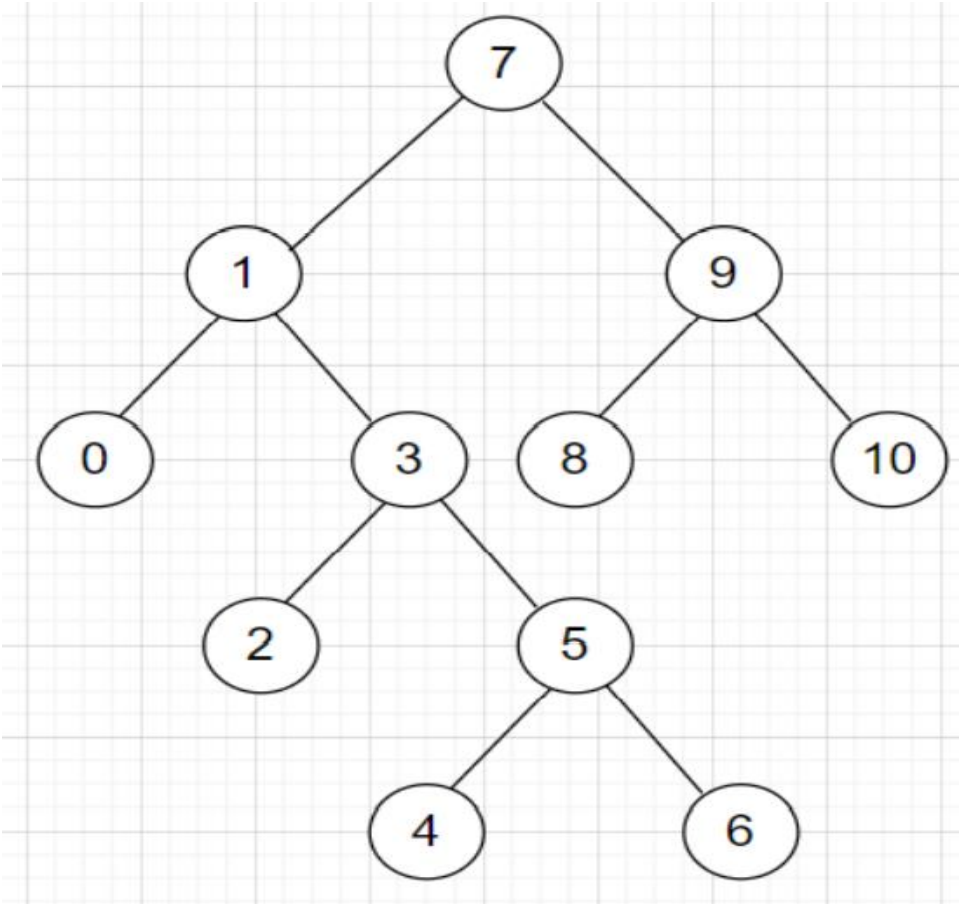
- Create: creates an empty tree.
- Insert: insert a node in the tree.
- Search: Searches for a node in the tree.
- Delete: deletes a node from the tree.
- Inorder: in-order traversal of the tree.
- Preorder: pre-order traversal of the tree.
- Postorder: post-order traversal of the tree.

Let’s do insertion in BST

Tree Traversal

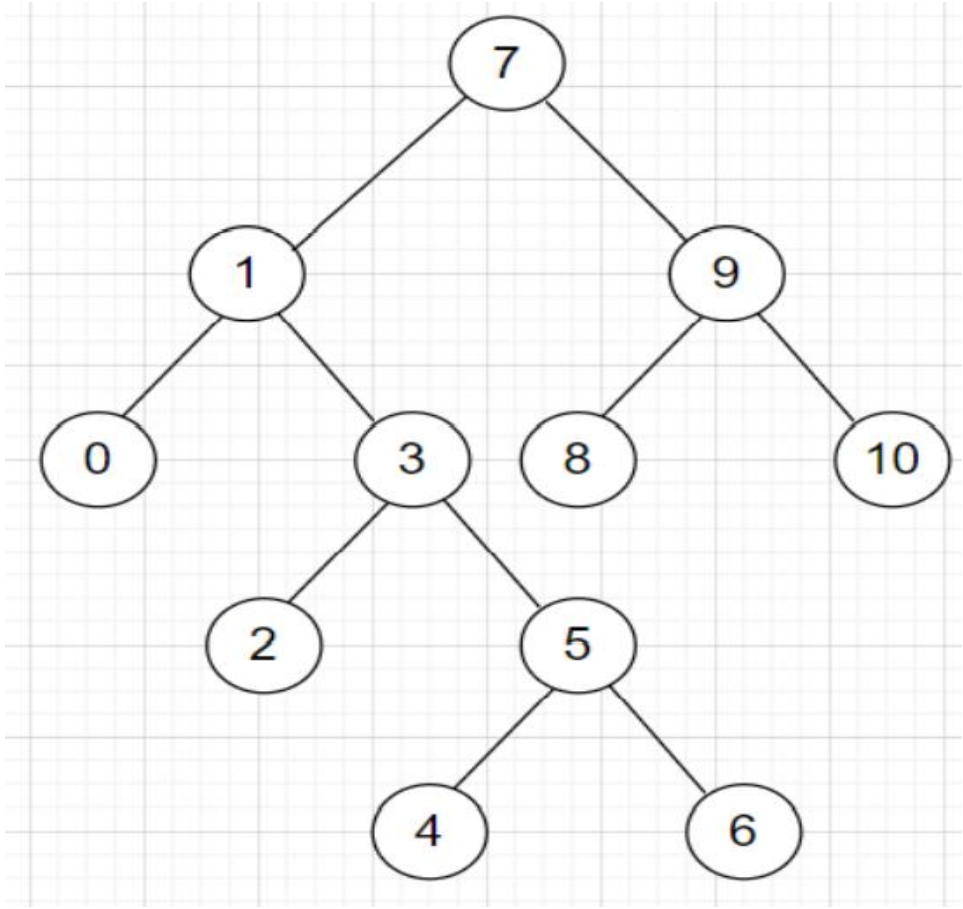
1. *Pre-order Traversal*
2. *Post – order Traversal*
3. *In-order Traversal*





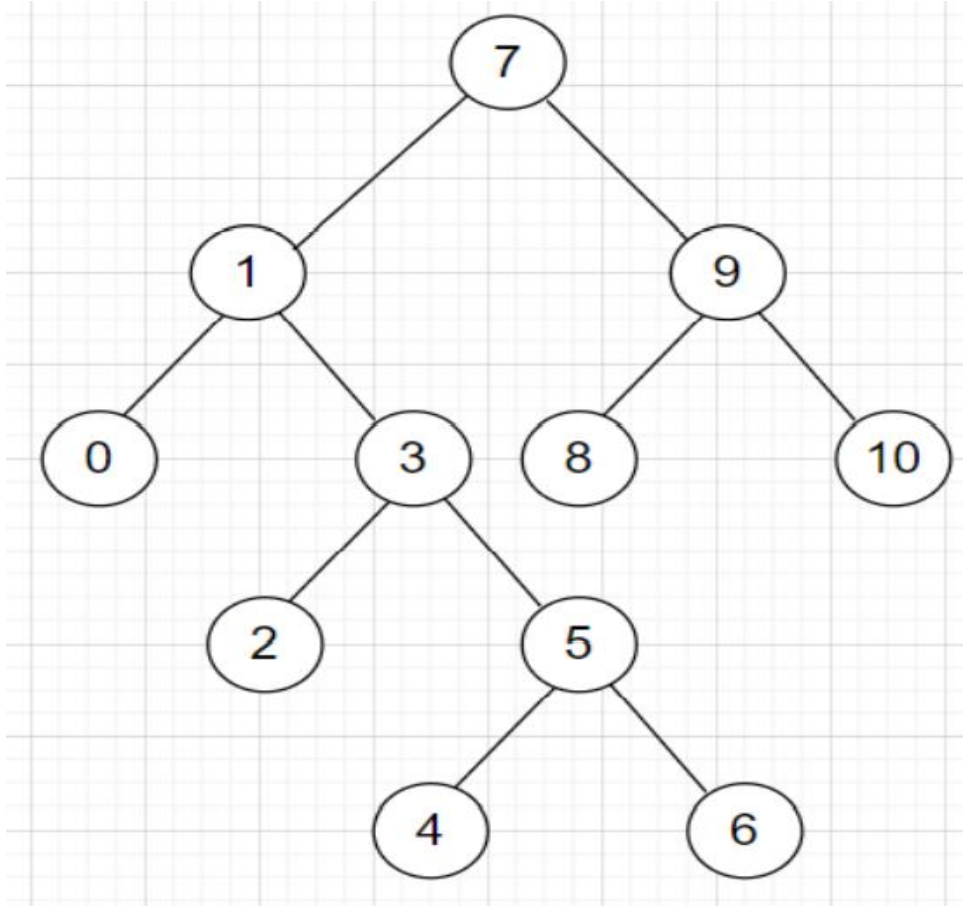
Pre-order traversal:

- Steps:
  1. Visit the root
  2. Visit all the nodes on the left side
  3. Visit all the nodes on the Right side
- Summary: Begins at the root (7), ends at the right-most node (10)
- Traversal sequence: 7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10



Post-order traversal:

- Steps:
  1. Visit all the nodes on the left side
  2. Visit all the nodes on the Right side
  3. Visit the root
- Summary: Begins with the left-most node (0), ends with the root (7)
- Traversal sequence: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9,7



In-order traversal:

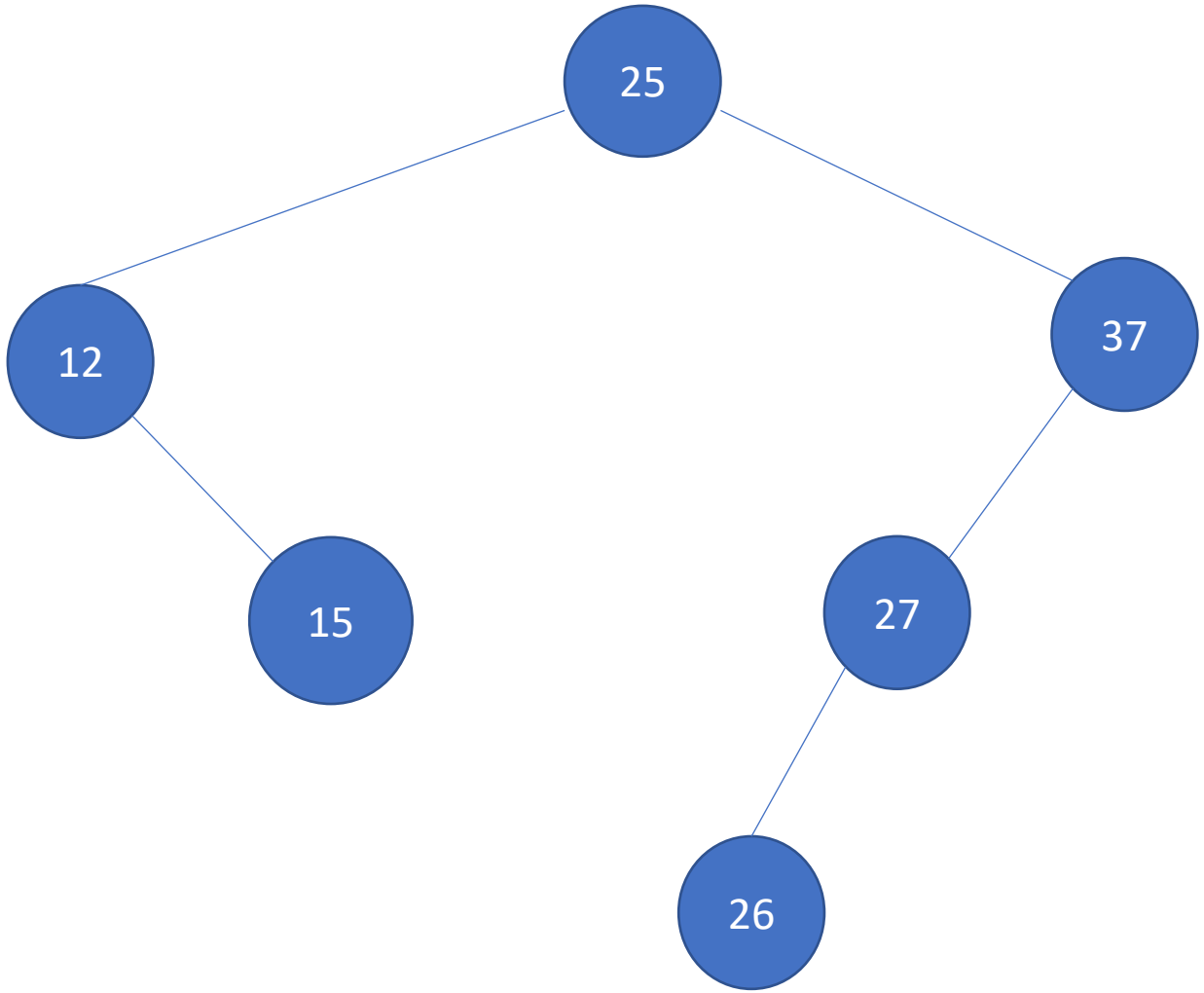
- Steps:
  1. Visit all the nodes on the left side
  2. Visit the root
  3. Visit all the nodes on the Right side
- Summary: Begins at the left-most node (0), ends at the rightmost node (10)
- Traversal Sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

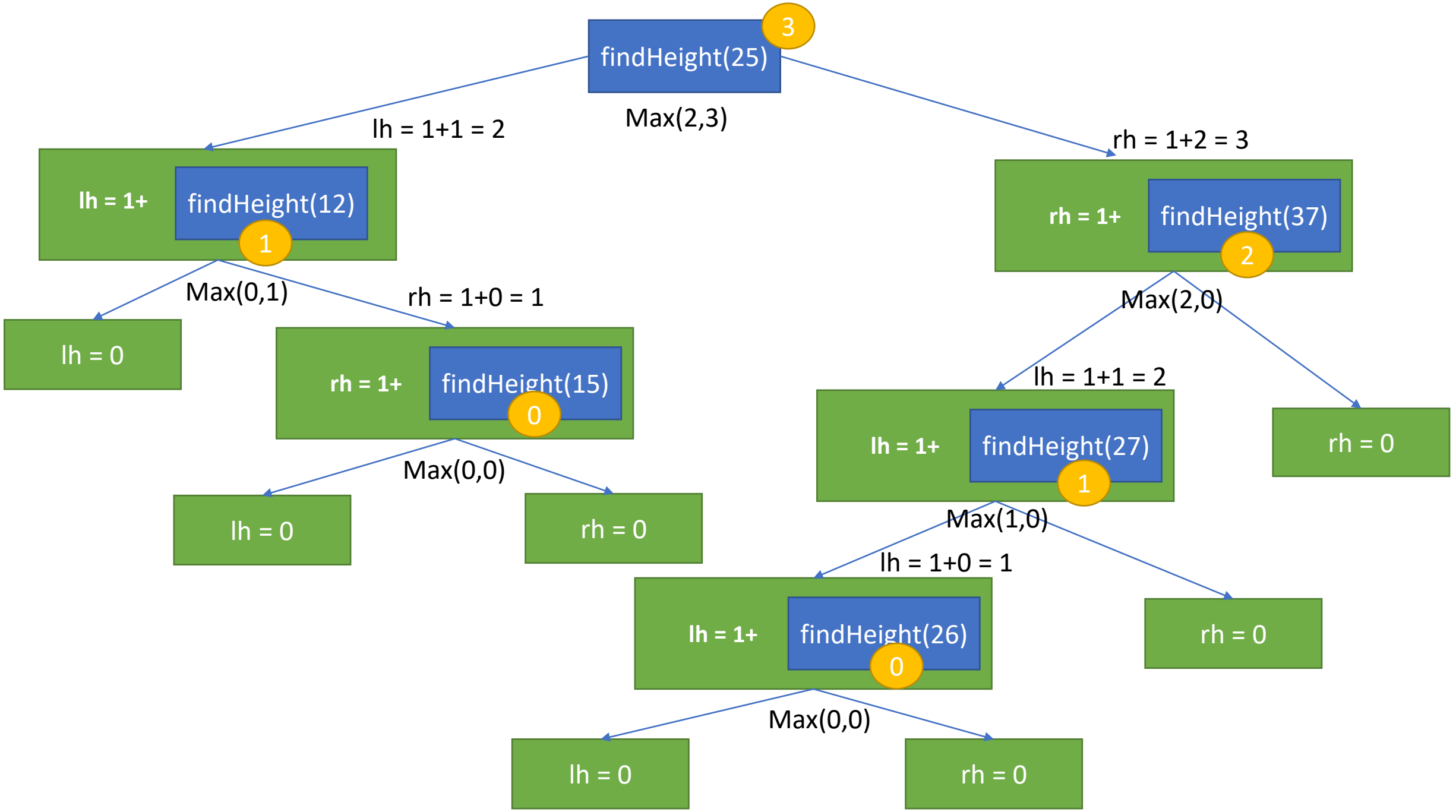
When to use Pre-Order, In-order or Post-Order?

- The traversal strategy the programmer selects depends on the specific needs of the algorithm being designed. The goal is speed, so pick the strategy that brings you the nodes you require the fastest.
- **pre-order:**
    - If you know you need to explore the roots before inspecting any leaves, you pick pre-order because you will encounter all the roots before all of the leaves.
    - Used to create a copy of a tree. For example, if you want to create a replica of a tree, put the nodes in an array with a pre-order traversal. Then perform an Insert operation on a new tree for each value in the array. You will end up with a copy of your original tree.
- **post-order:**
    - If you know you need to explore all the leaves before any nodes, you select post-order because you don't waste any time inspecting roots in search for leaves.
    - Used to delete a tree from leaf to root
- **in-order:**
    - If you know that the tree has an inherent sequence in the nodes, and you want to flatten the tree back into its original sequence, than an in-order traversal should be used. The tree would be flattened in the same way it was created. A pre-order or post-order traversal might not unwind the tree back into the sequence which was used to create it.
    - Used to get the values of the nodes in non-decreasing (sorted/ascending) order in a BST.

Let’s Code for finding the Height/Depth of the Tree

Tree:




findHeight(25)
3
lh = 1+1 = 2
Max(2,3)
rh = 1+2 = 3
lh = 1+
findHeight(12)
1
rh = 1+
findHeight(37)
2
Max(0,1)
rh = 1+0 = 1
Max(2,0)
lh = 0
rh = 1+
findHeight(15)
0
lh = 1+1 = 2
lh = 1+
findHeight(27)
1
rh = 0
Max(0,0)
Max(1,0)
lh = 0
rh = 0
lh = 1+0 = 1
lh = 1+
findHeight(26)
0
rh = 0
Max(0,0)
lh = 0
rh = 0

AVL Tree

- AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
- AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.
- Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)