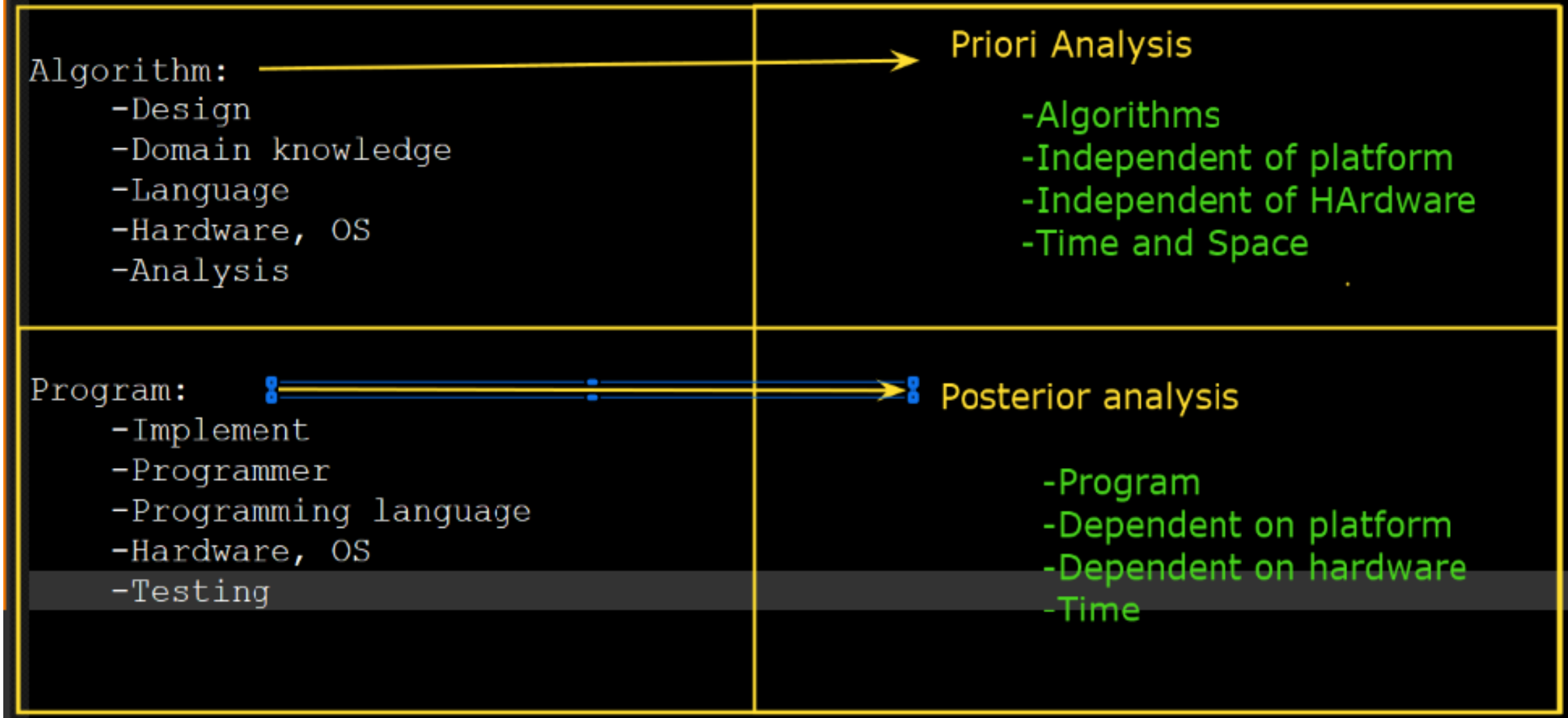
Data Structure

Sep23 : Day 5

Kiran Waghmare

CDAC Mumbai

Analysis of Algorithm:



Algorithm Complexity:-

Two main factors, which decide the efficiency of X.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

Asymptotic Notations

Asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

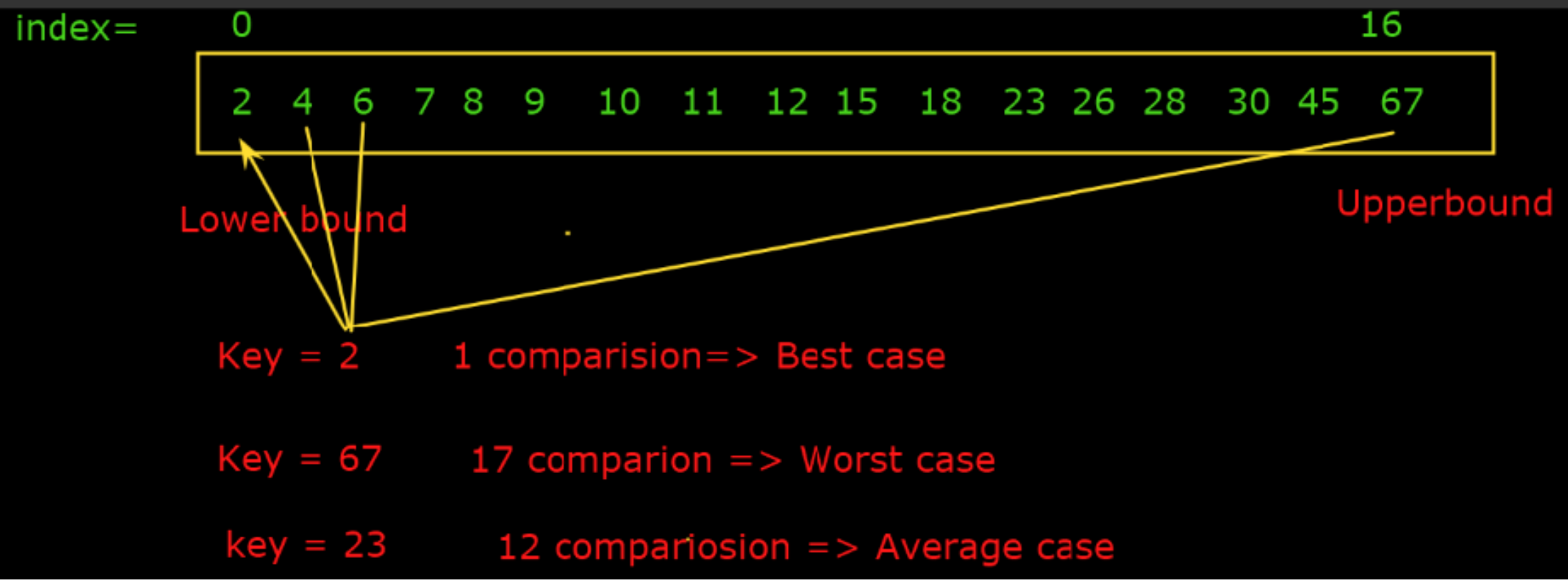
Usually, the time required by an algorithm falls under three types –

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

Asymptotic Notations:

-Asymptoic notation of an algorithm refers to defining the mathematical boundations run-time performance. Using asymptotic analysis, we can conclude, best case, average worst case scenario of the algorithm

1. Best case: Minimum time required for program execution.
2. Worst case: Maximum time required for program execution
3. Average case: Average time required for program execution.

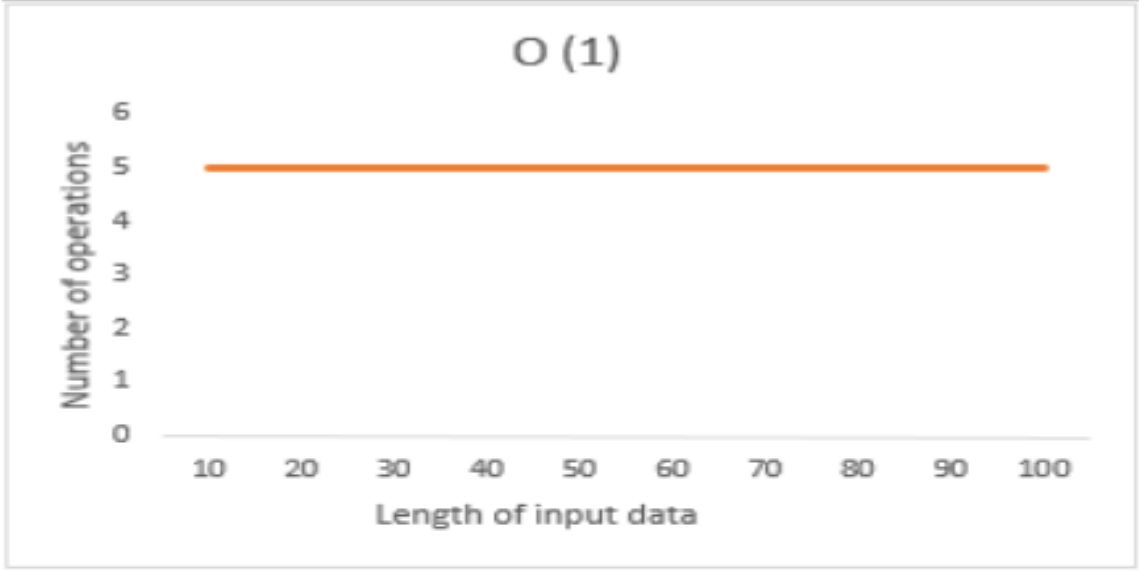


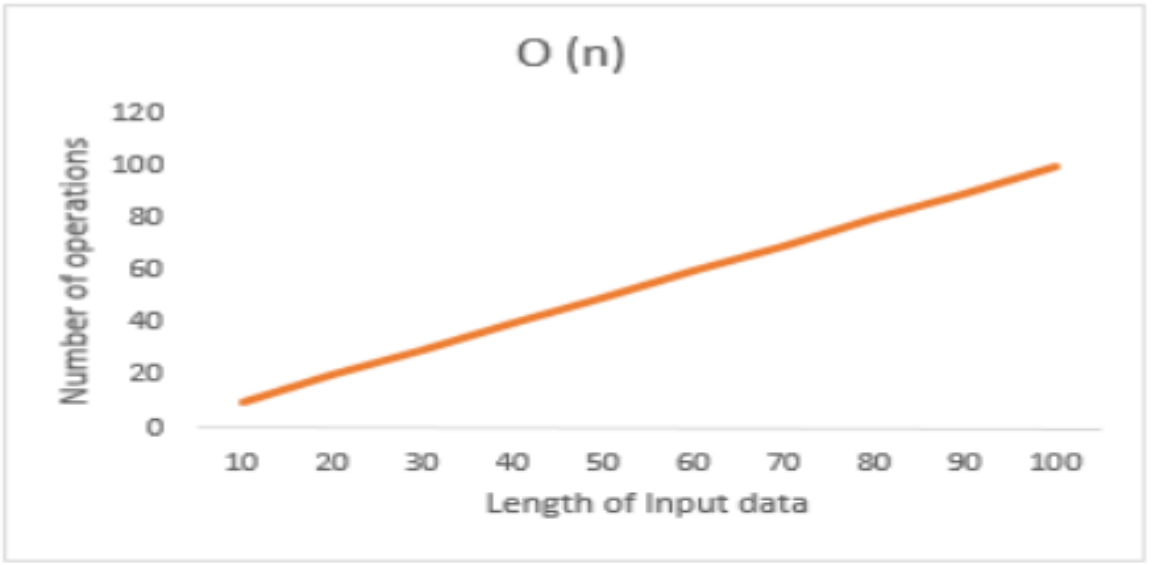
Asymptotic Notations

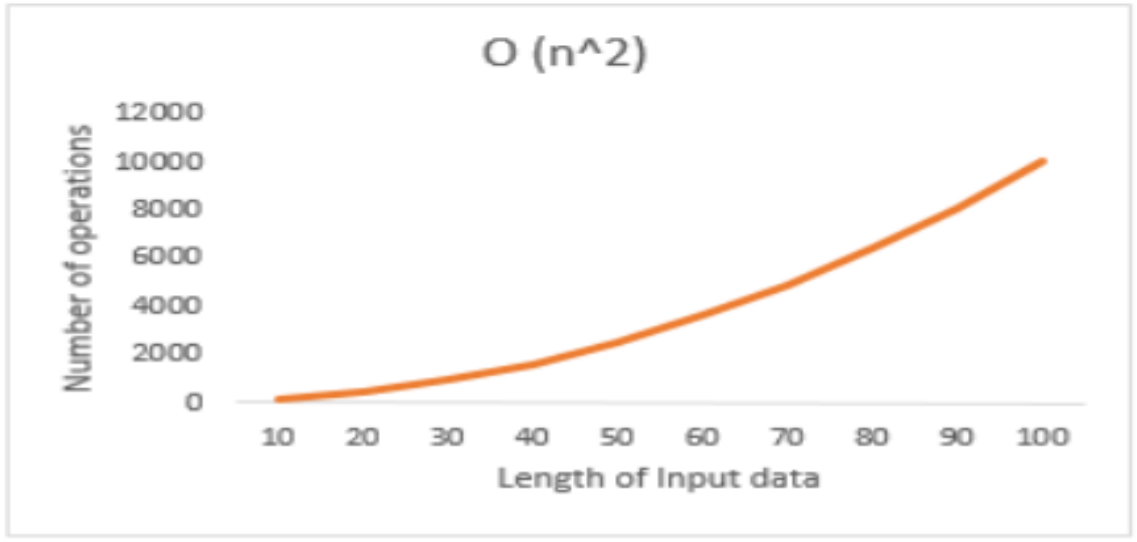
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

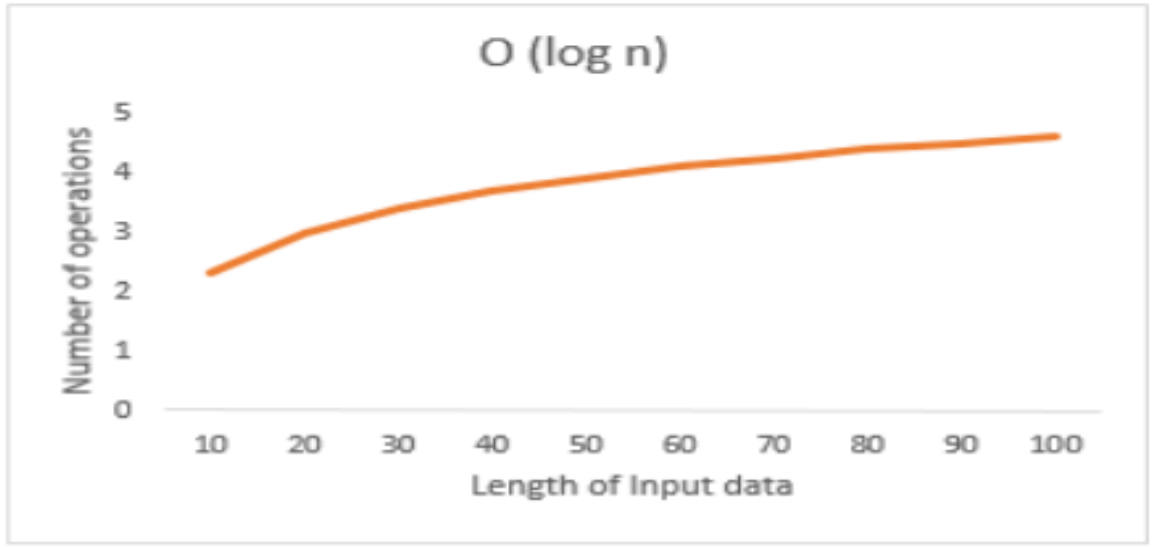
- **O Notation**
- **Ω Notation**
- **θ Notation**

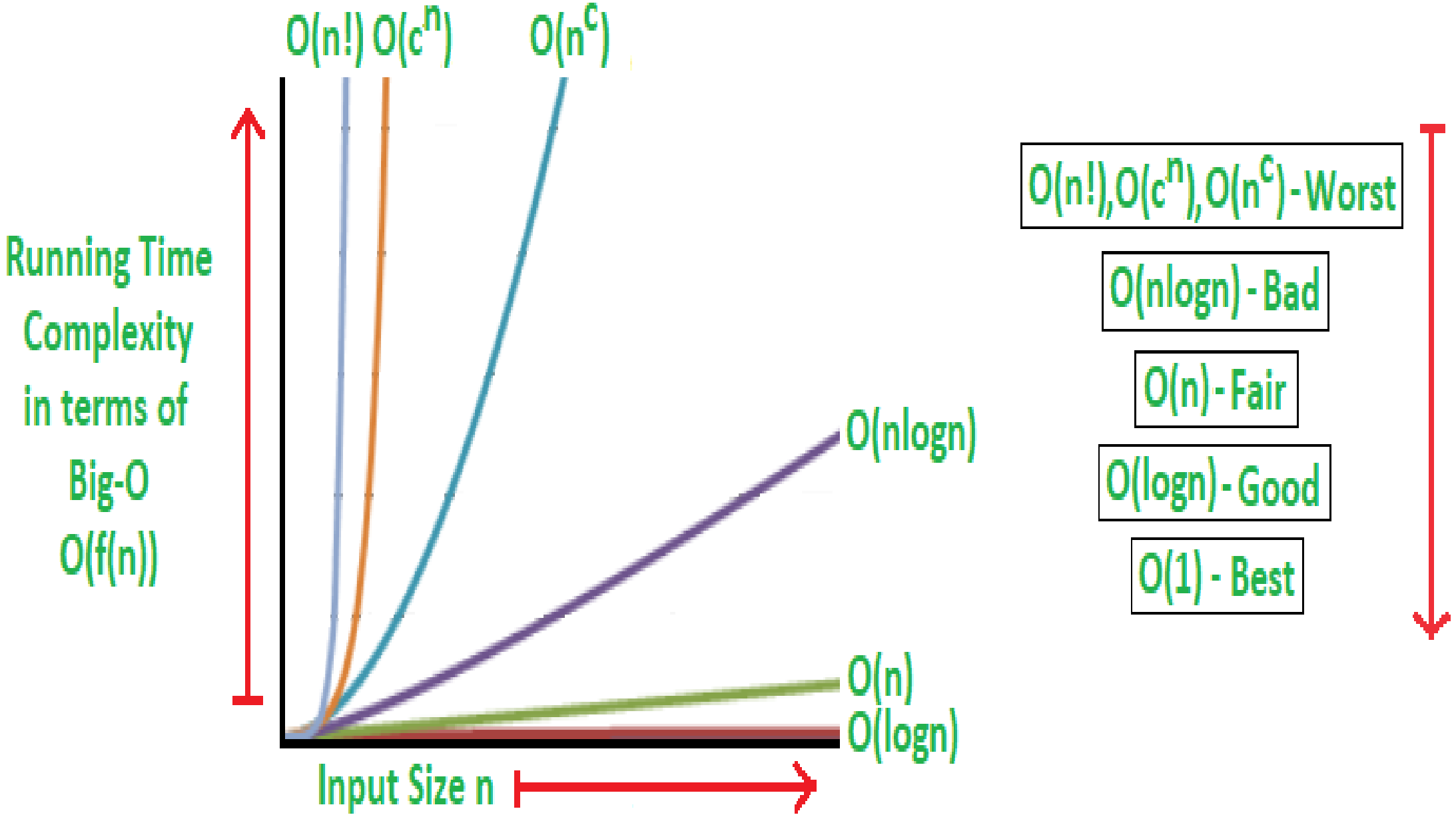
The order of growth for all time complexities are indicated in the graph below:










O(n!) O(c^n)
O(n^c)
Running Time Complexity in terms of Big-O O(f(n))
O(nlogn)
O(n)
O(logn)
Input Size n
O(n!),O(c^n),O(n^c) - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best

Commonly Used Functions and Their Comparison

1. **Constant Functions** - $f(n) = 1$ - Whatever is the input size n, these functions take a constant amount of time.
2. **Linear Functions** - $f(n) = n$ - These functions grow linearly with the input size n.
3. **Quadratic Functions** - $f(n) = n^2$ - These functions grow faster than the superlinear functions i.e., $n \log(n)$.
4. **Cubic Functions** - $f(n) = n^3$ - Faster growing than quadratic but slower than exponential.
5. **Logarithmic Functions** - $f(n) = \log(n)$ - These are slower growing than even linear functions.
6. **Superlinear Functions** - $f(n) = n \log(n)$ - Faster growing than linear but slower than quadratic.
7. **Exponential Functions** - $f(n) = c^n$ - Faster than all of the functions mentioned here except the factorial functions.
8. **Factorial Functions** - $f(n) = n!$ - Fastest growing than all these functions mentioned here.

Complexities of an Algorithm

The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n).

The complexity of an algorithm can be divided into two types.

The time complexity and the space complexity.

Time Complexity of an Algorithm

The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm.

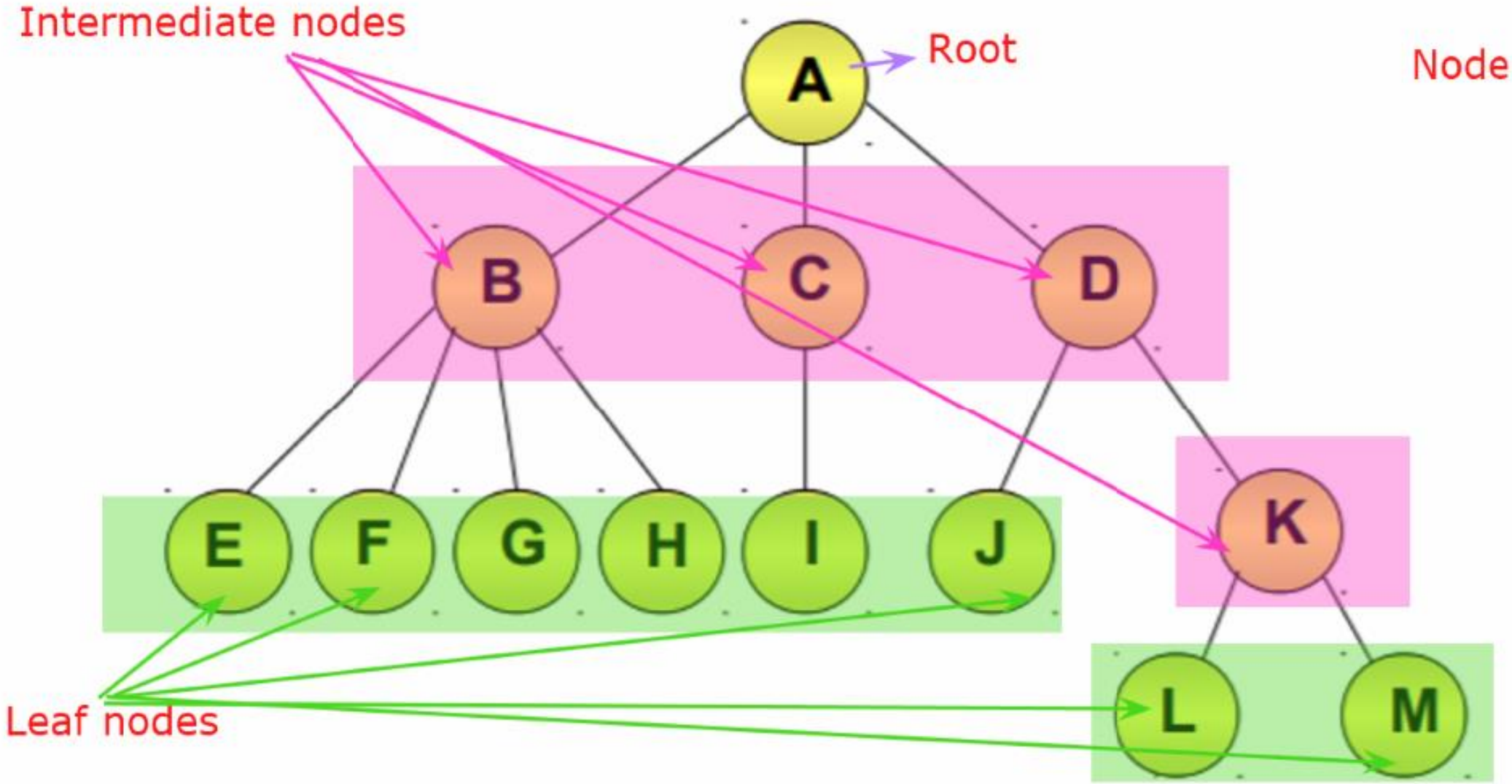
This calculation is totally independent of implementation and programming language.

Space Complexity of an Algorithm

Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm.

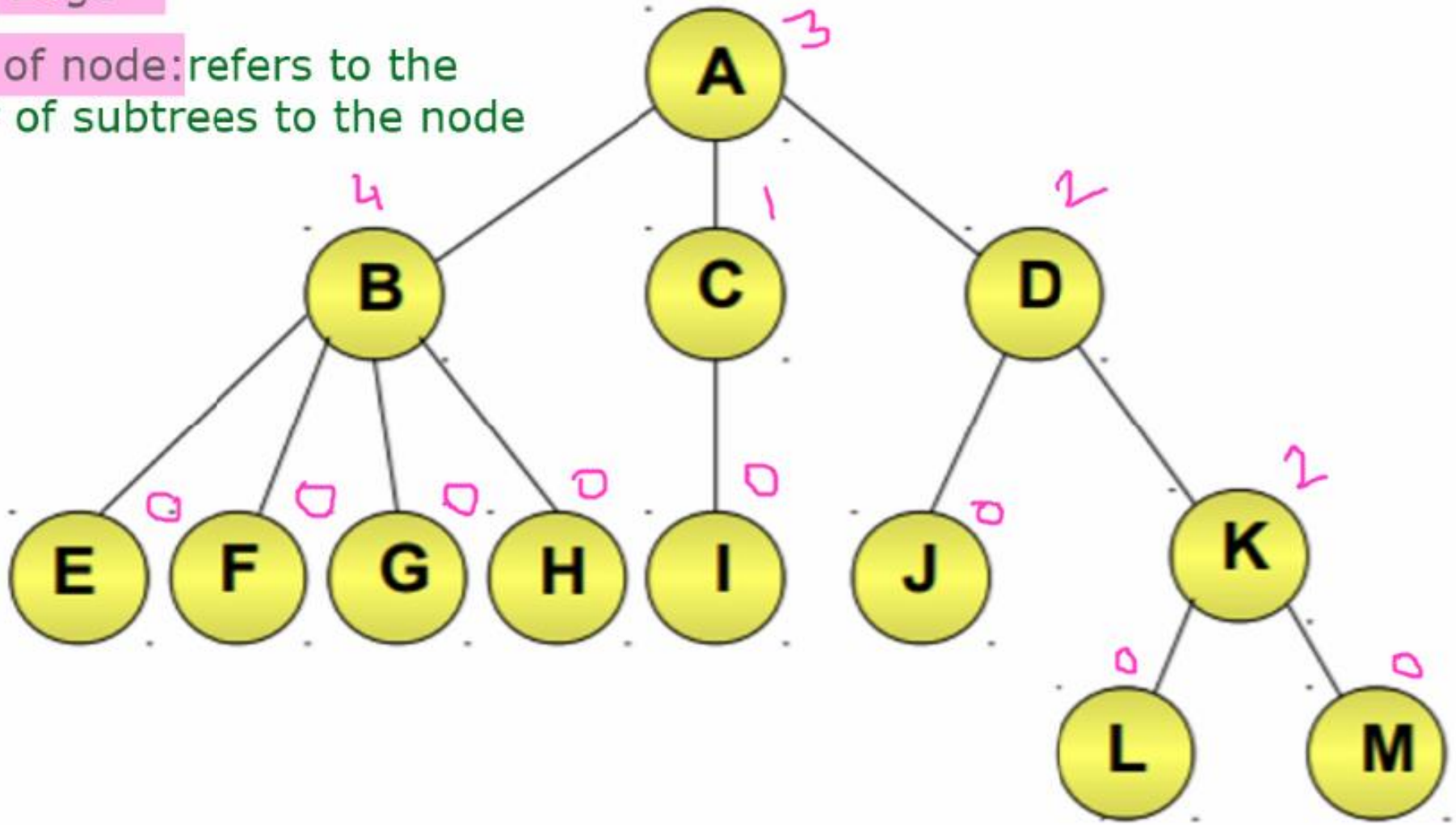
The memory space is generally considered as the primary memory.

Trees


Intermediate nodes
A
Root
Node
B
C
D
E
F
G
H
I
J
K
L
M
Leaf nodes

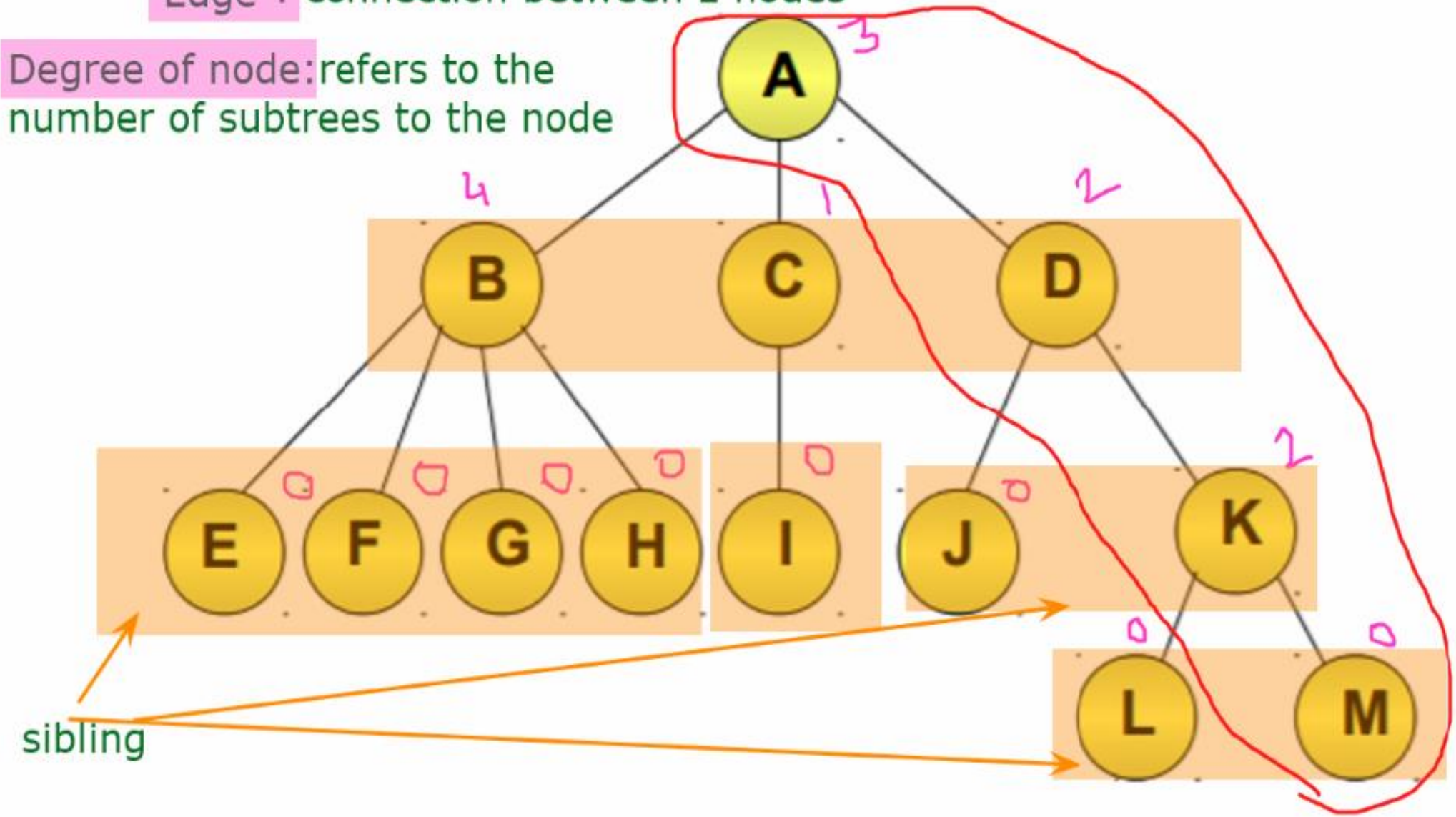
Edge : connection between 2 nodes

Degree of node: refers to the number of subtrees to the node



Edge : connection between 2 nodes

Degree of node: refers to the number of subtrees to the node



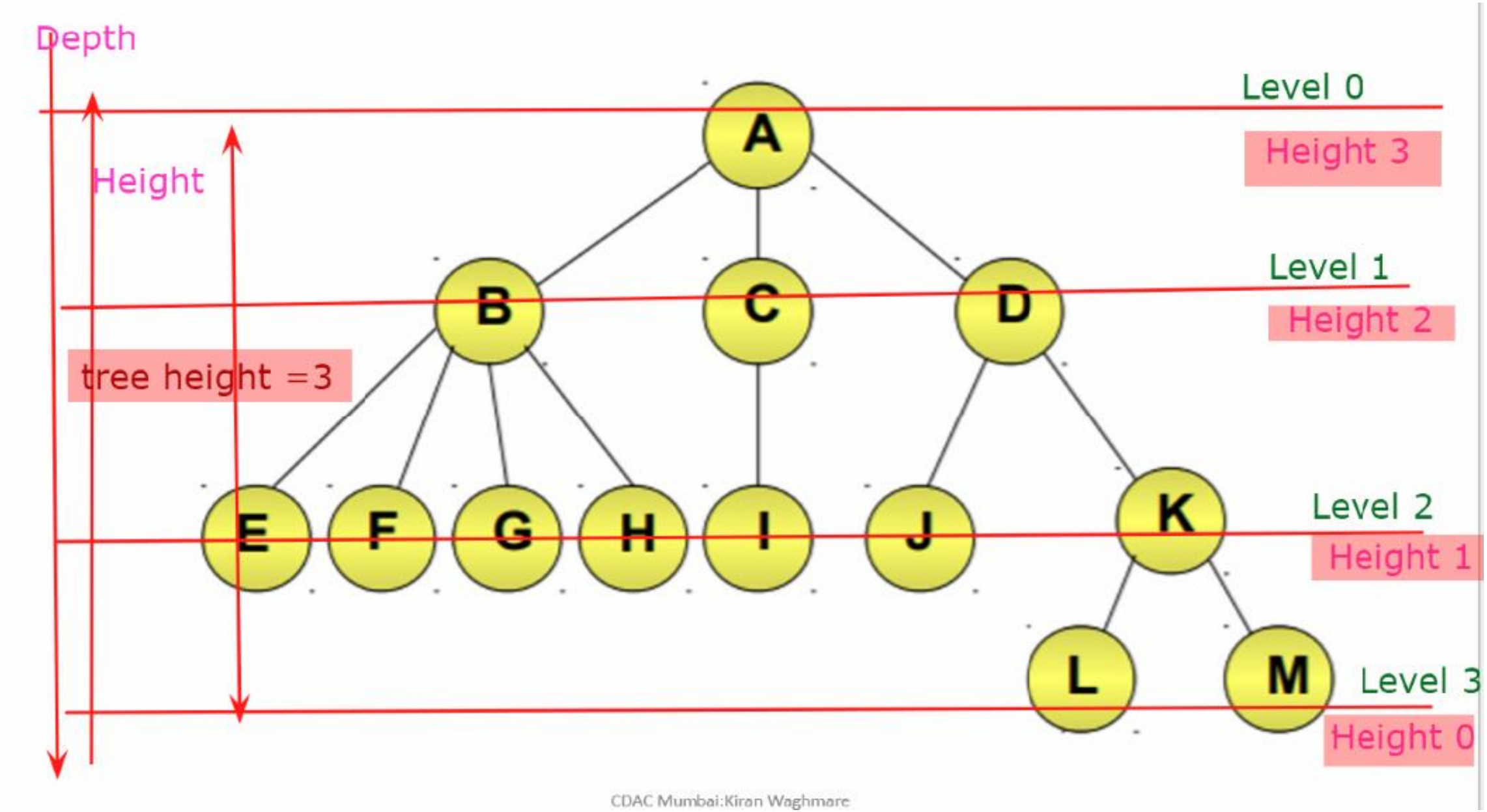
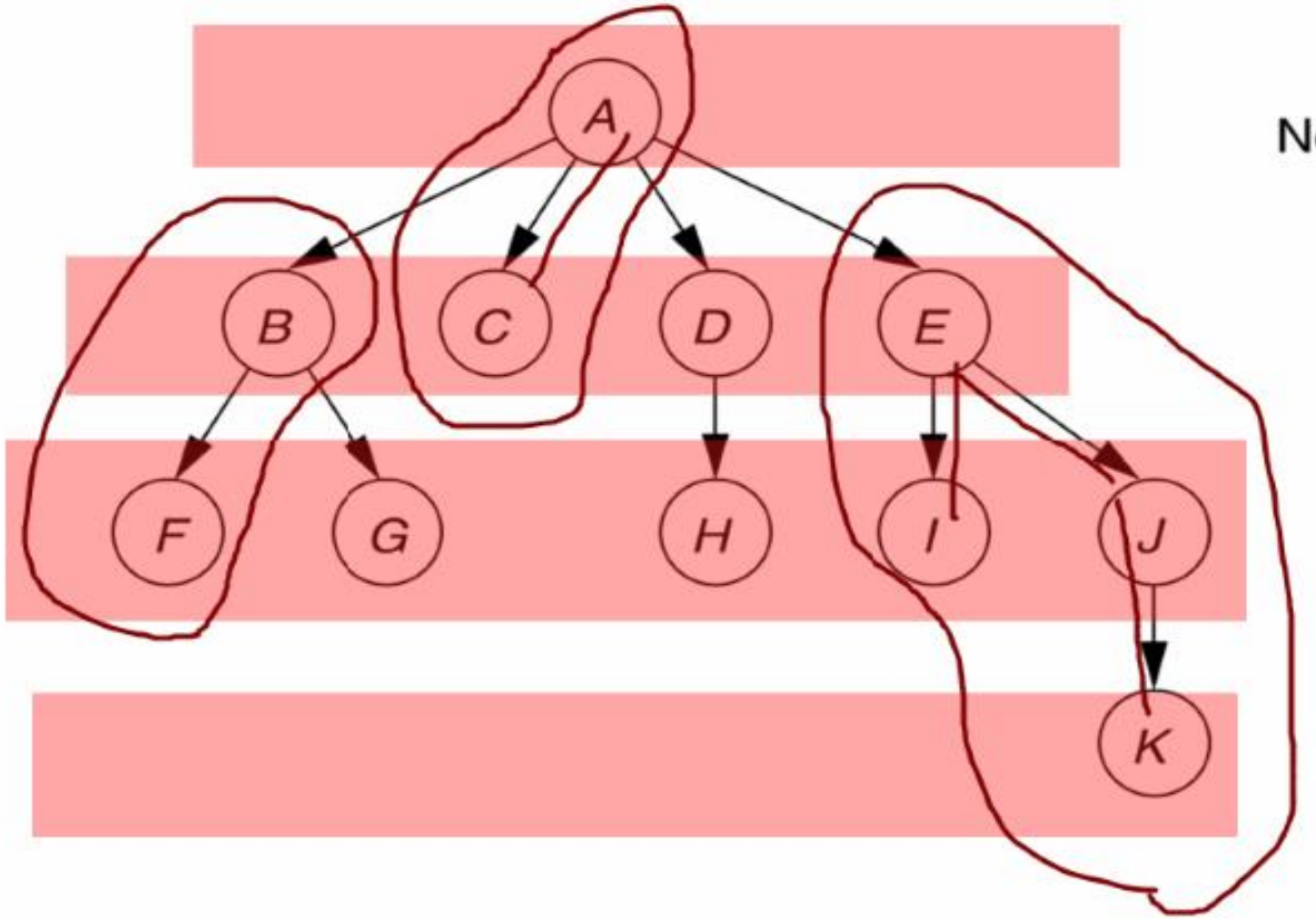
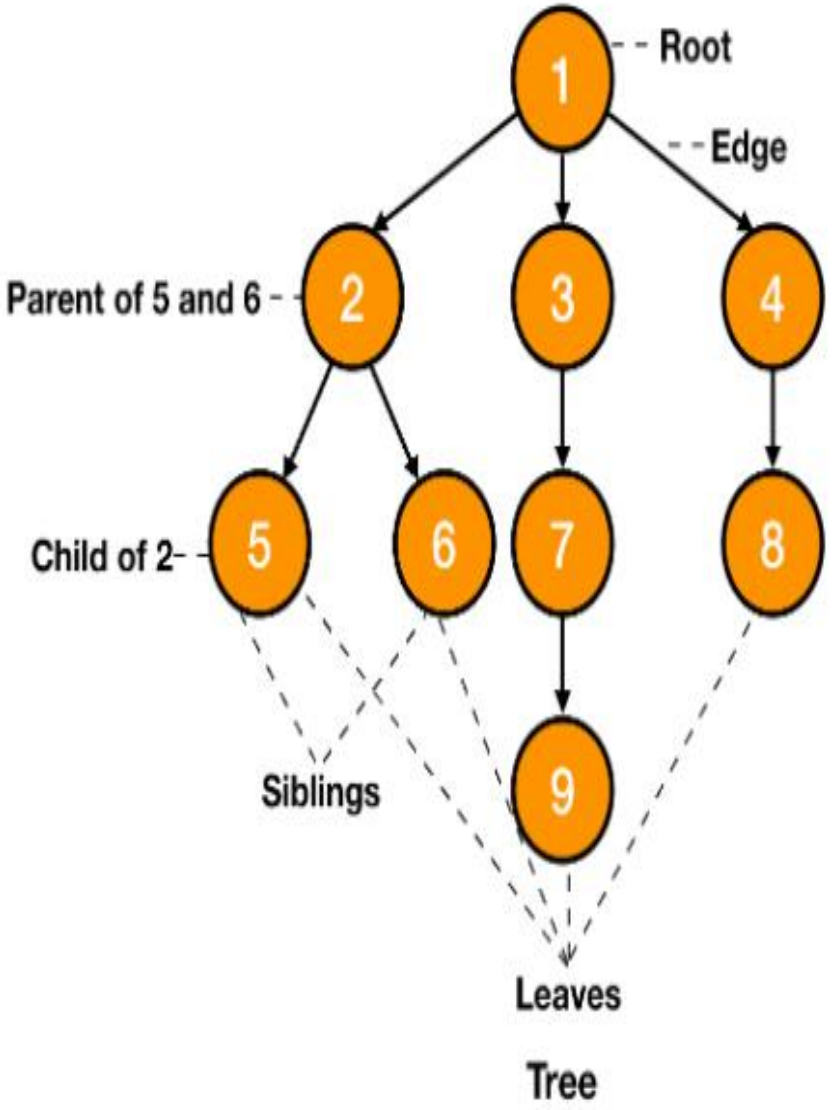
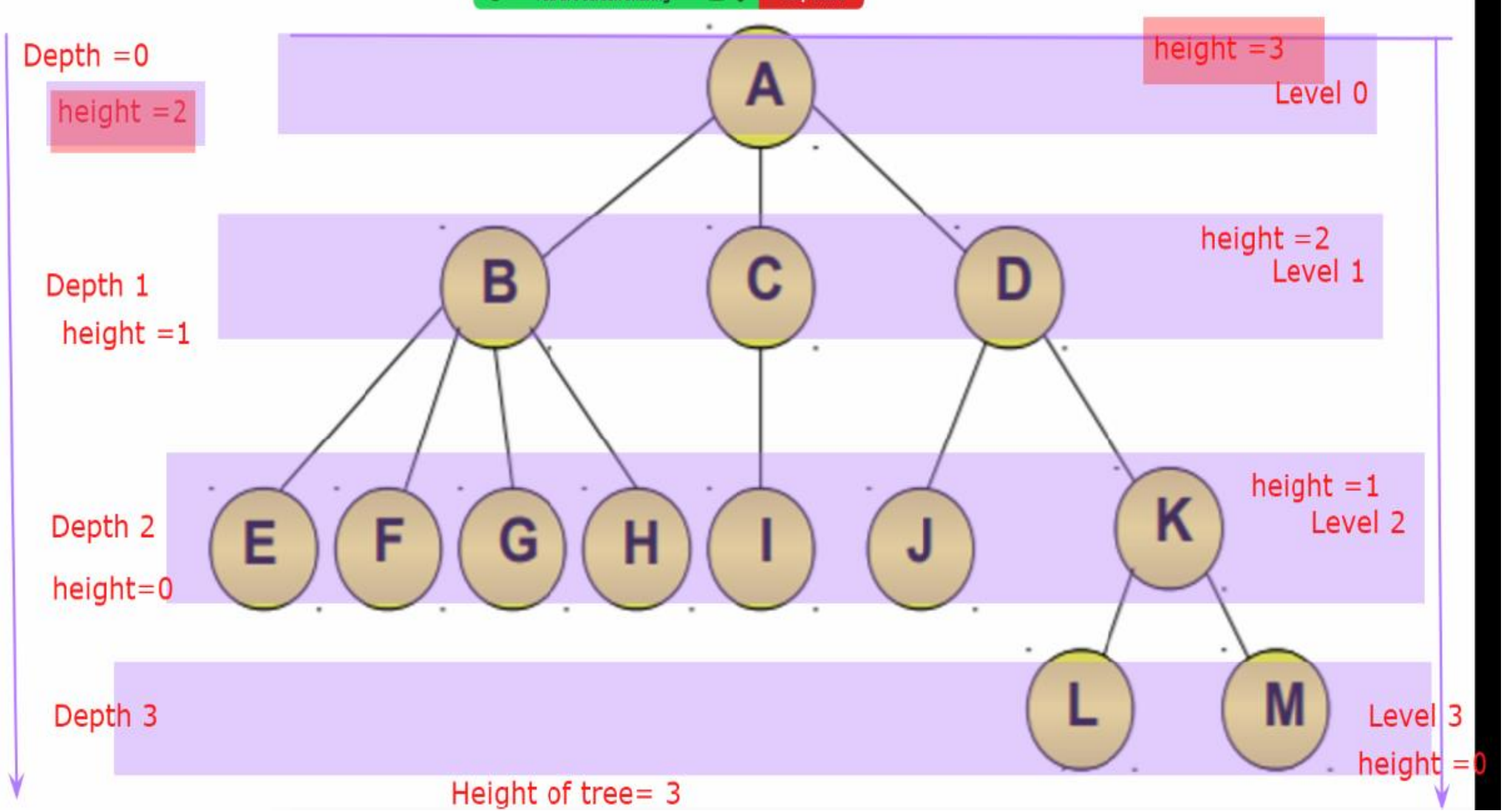

Depth
Height
tree height =3
Level 0
Height 3
Level 1
Height 2
Level 2
Height 1
Level 3
Height 0
A
B
C
D
E
F
G
H
I
J
K
L
M
CDAC Mumbai:Kiran Waghmare

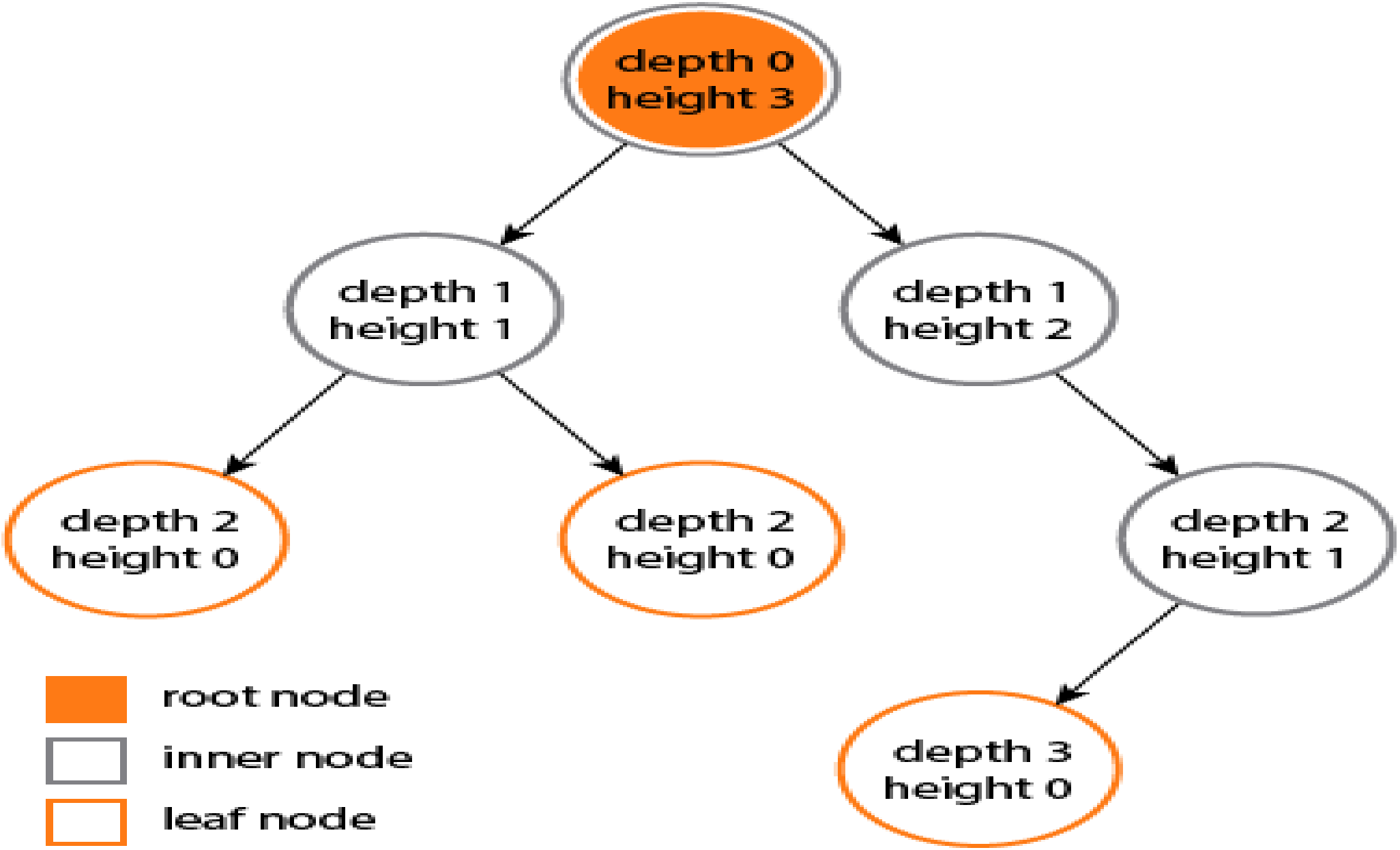
Figure 1
A tree, with height and depth information



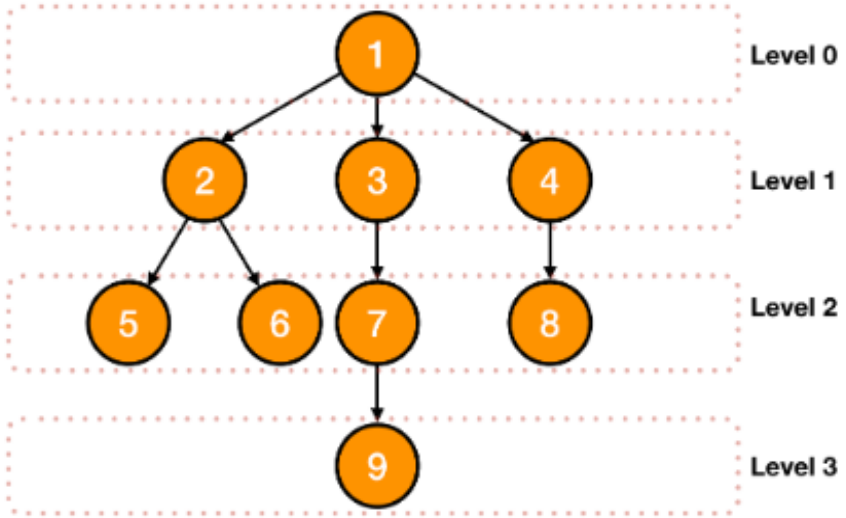
Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3


Root
Edge
Parent of 5 and 6
Child of 2
Siblings
Leaves
Tree
1
2
3
4
5
6
7
8
9

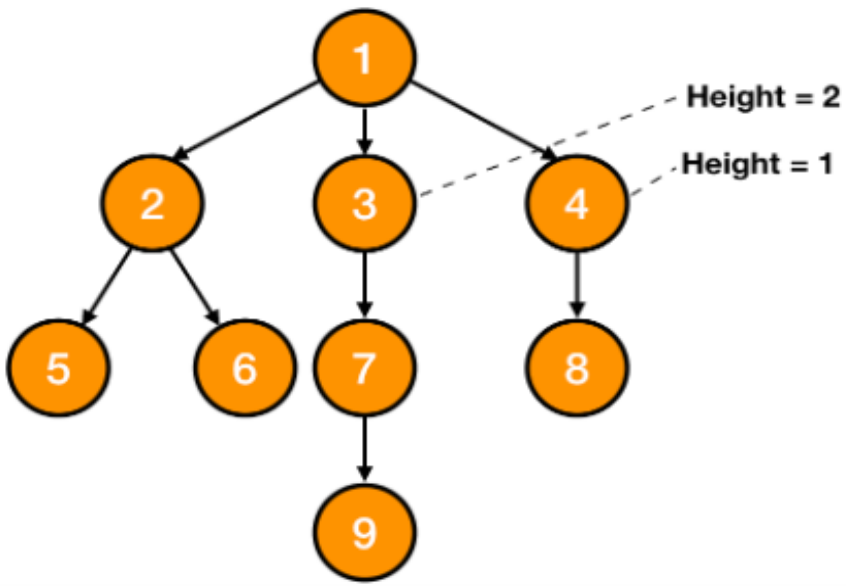

Depth =0
height =2
height =3
Level 0
A
B
C
D
Depth 1
height =1
height =2
Level 1
E
F
G
H
I
J
K
Depth 2
height=0
height =1
Level 2
L
M
Depth 3
Level 3
height =0
Height of tree= 3


depth 0
height 3
depth 1
height 1
depth 1
height 2
depth 2
height 0
depth 2
height 0
depth 2
height 1
depth 3
height 0
root node
inner node
leaf node

- **Level** → The root of a tree is at level 0 and the nodes whose parent is root are at level 1 and so on.



- **Height** → The height of a node is the number of nodes (excluding the node) on the longest path from the node to a leaf.



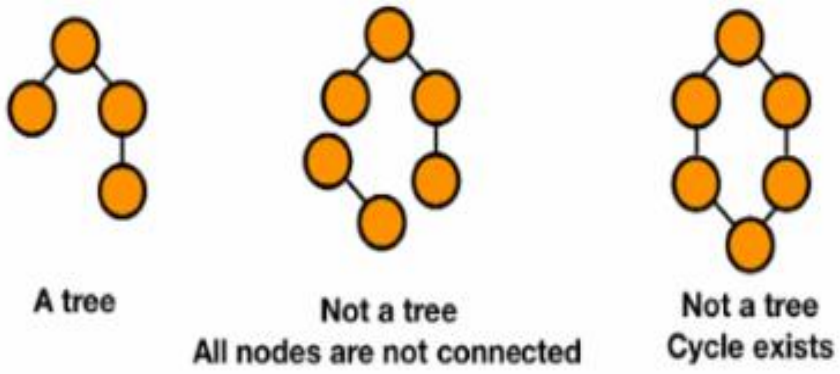
Properties of a Tree

A tree must have some properties so that we can differentiate from other data structures. So, let's look at the properties of a tree.

The numbers of nodes in a tree must be a finite and nonempty set.

There must exist a path to every node of a tree i.e., every node must be connected to some other node.

There must not be any cycles in the tree. It means that the number of edges is one less than the number of nodes.



A tree

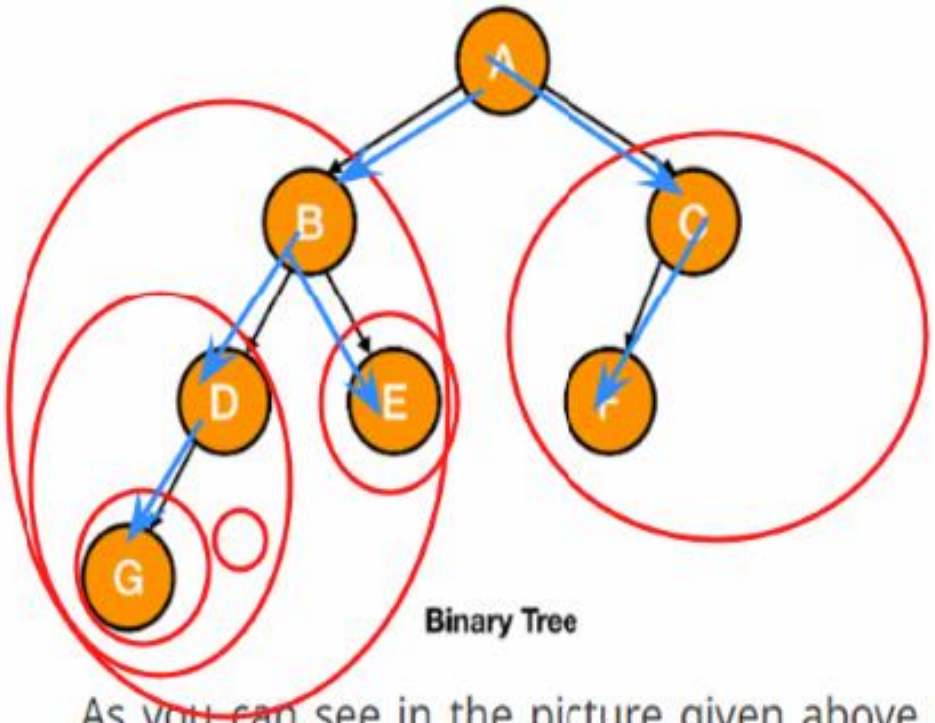
Not a tree
All nodes are not connected

Not a tree
Cycle exists

Binary Trees

no. of childrens = 0, 1, 2

A binary tree is a tree in which every node has at most two children.



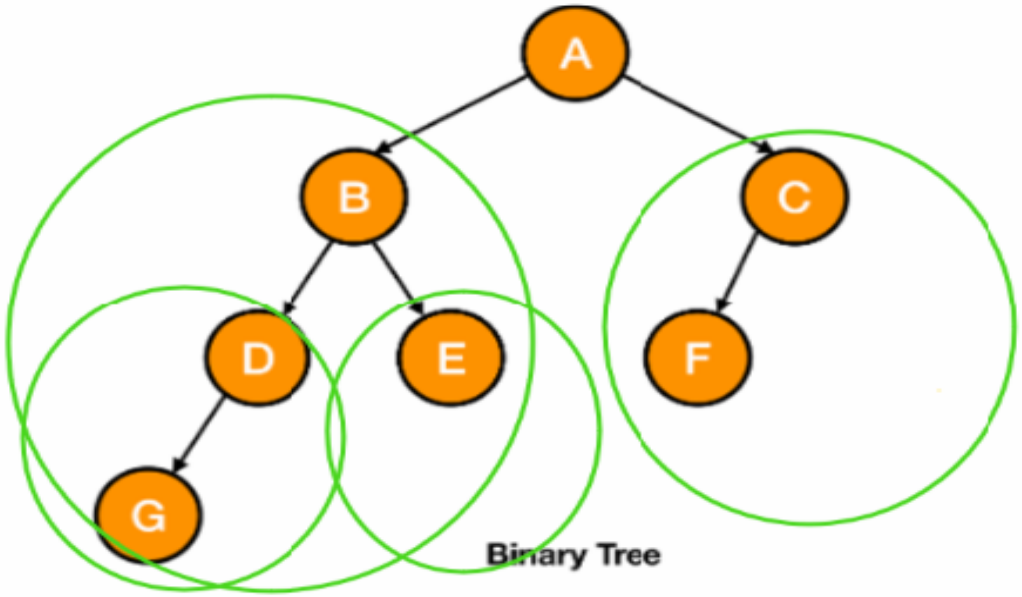
Binary Tree

As you can see in the picture given above, a node can have less than 2 children but not more than that.

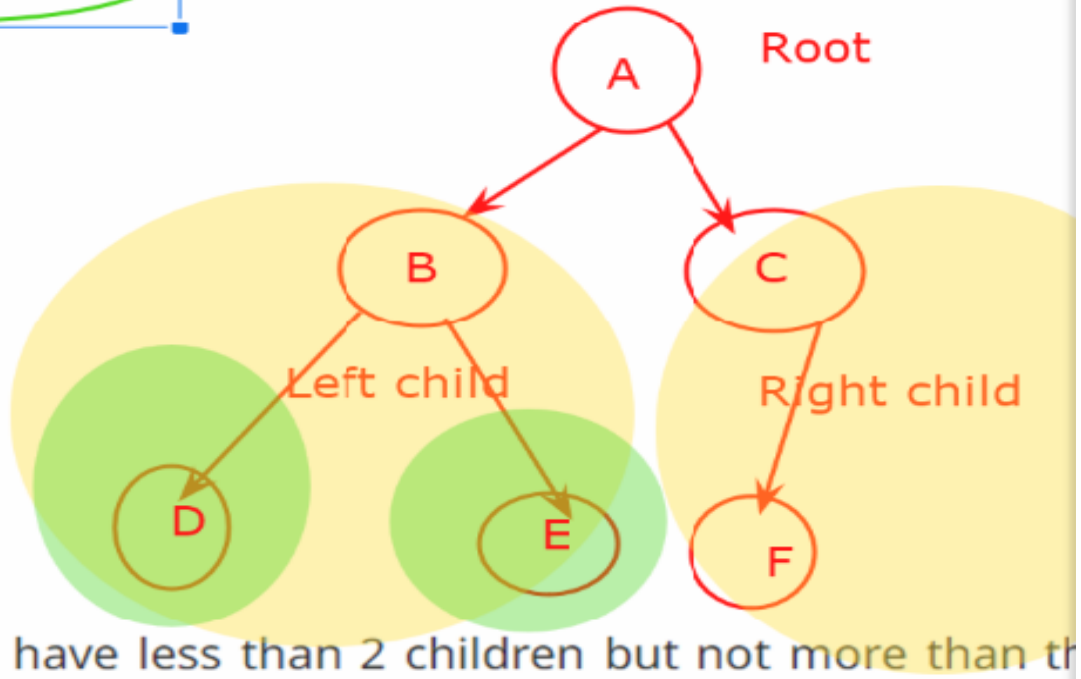
We can also classify a binary tree into different categories. Let's have a look at them:

Binary Trees

A binary tree is a tree in which every node has at most two children.



Binary Tree



As you can see in the picture given above, a node can have less than 2 children but not more than th

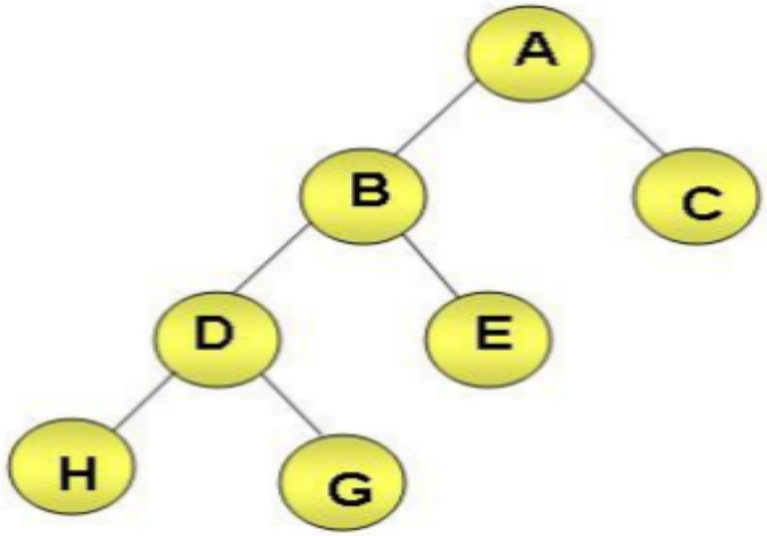
Left subtree Right Su

We can also classify a binary tree into different categories. Let's have a look at them:

CDAC Mumbai:Kiran Waghmare

Defining Binary Trees

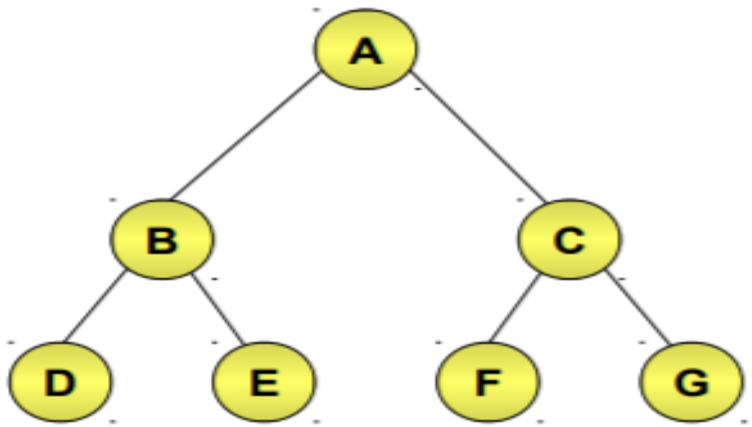
- Binary tree is a specific type of tree in which each node can have at most two children namely left child and right child.
- There are various types of binary trees:
  - Strictly binary tree
  - Full binary tree
  - Complete binary tree
- Strictly binary tree:
  - A binary tree in which every node, except for the leaf nodes, has non-empty left and right children.



Defining Binary Trees (Contd.)

- Full binary tree:
  - A binary tree of depth d that contains exactly $2^d - 1$ nodes.

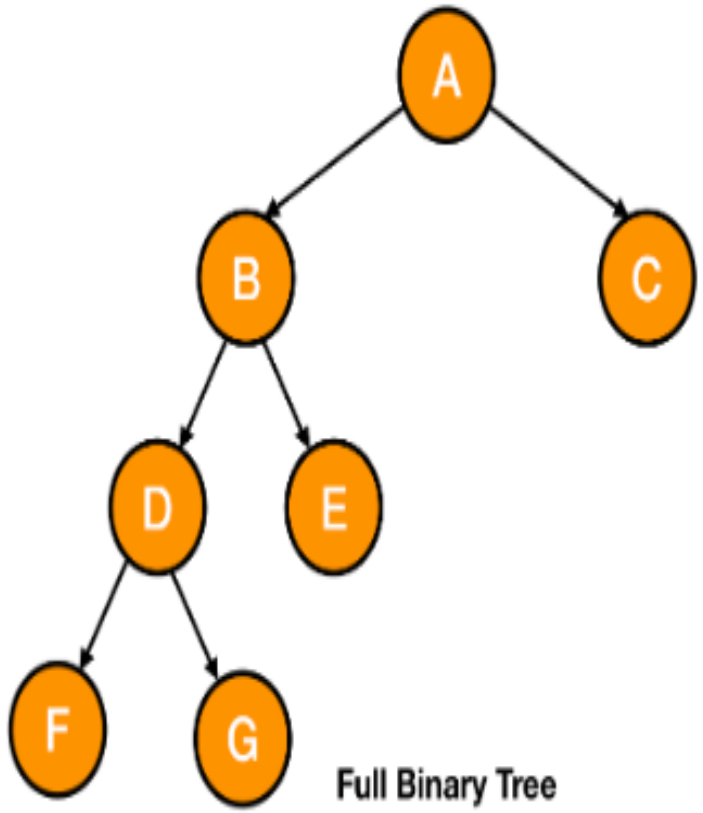
A Binary Tree is a full binary tree if every node has 0 or 2 children.



Depth = 3

Total number of nodes = $2^3 - 1 = 7$

Full Binary Tree → A binary tree in which every node has 2 children except the leaves is known as a full binary tree.



Full Binary Tree

Full binary tree

A full binary tree which is also called as proper binary tree or 2-tree is a tree in which all the node other than the leaves has exact two children.

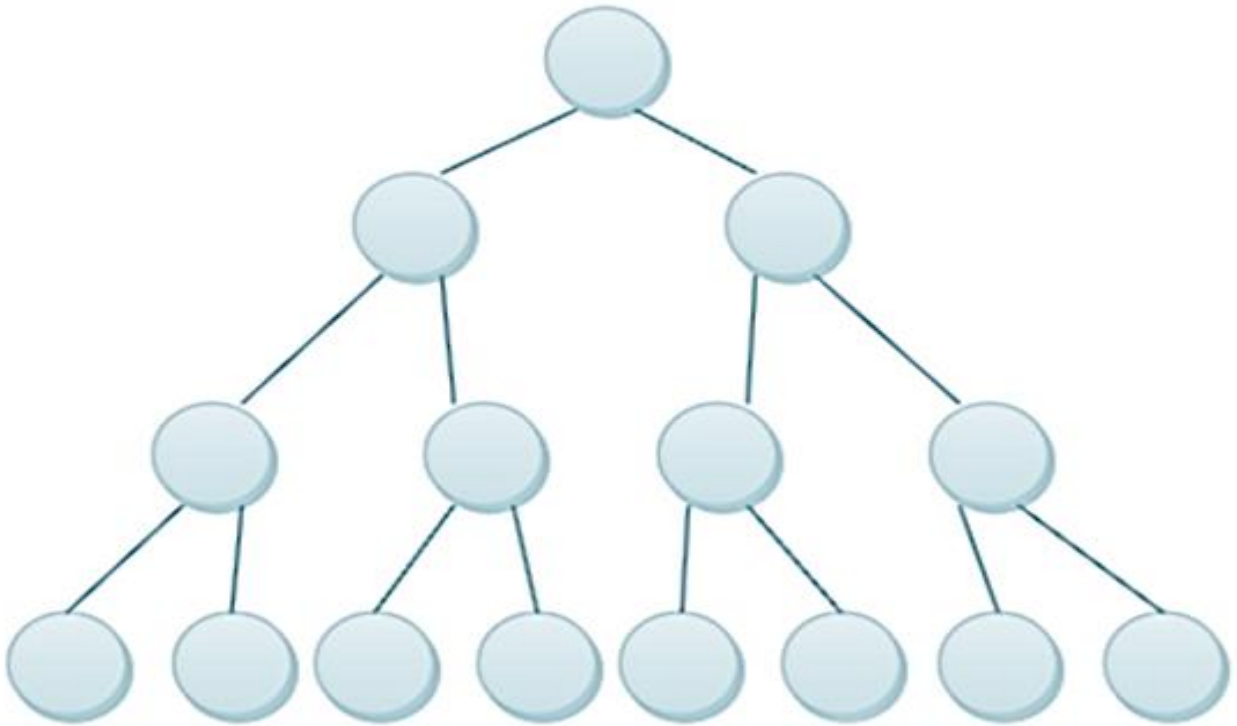


Figure: Full Binary tree

Complete binary tree

A complete binary tree is a binary tree in which at every level, except possibly the last, has to be filled and all nodes are as far left as possible.

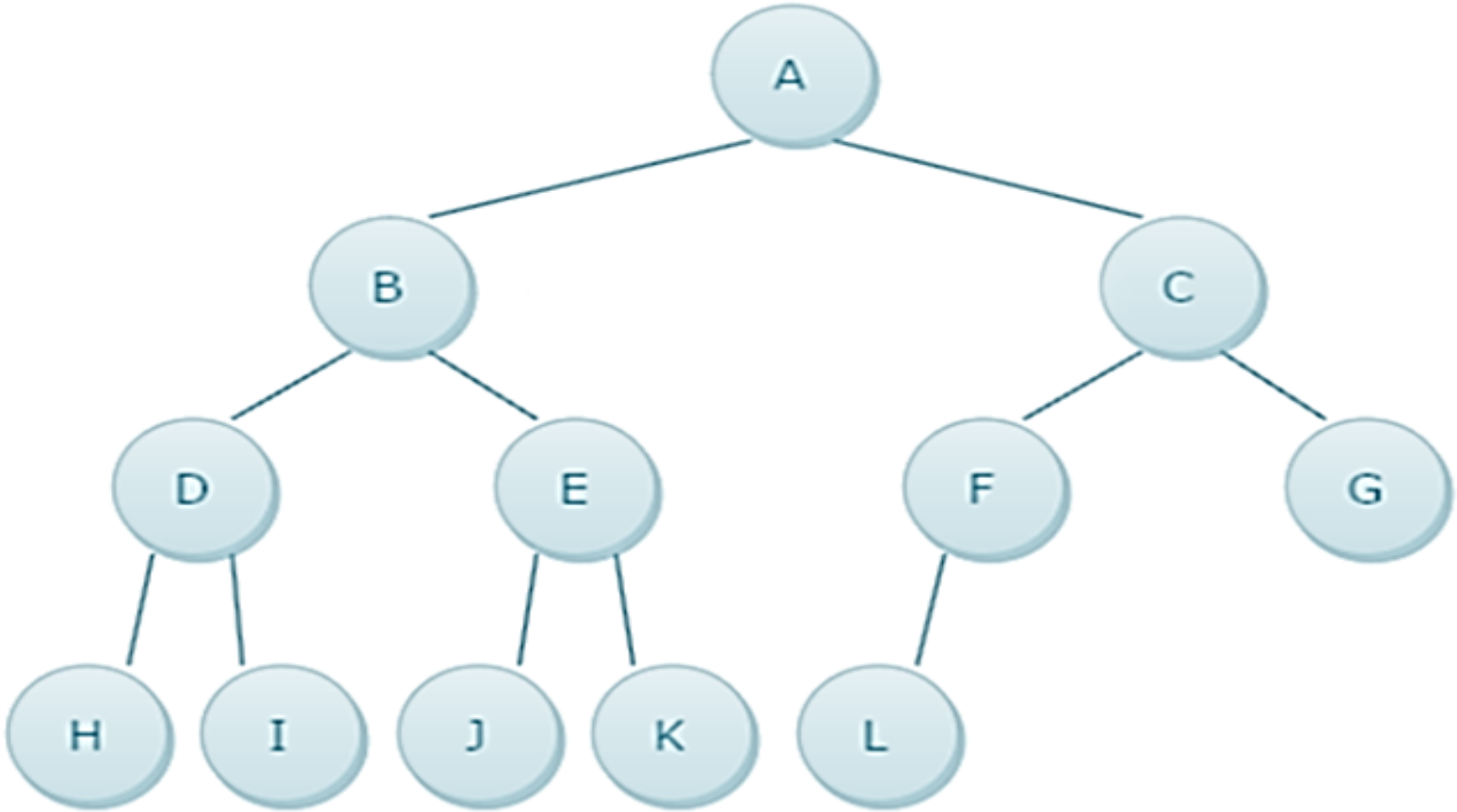
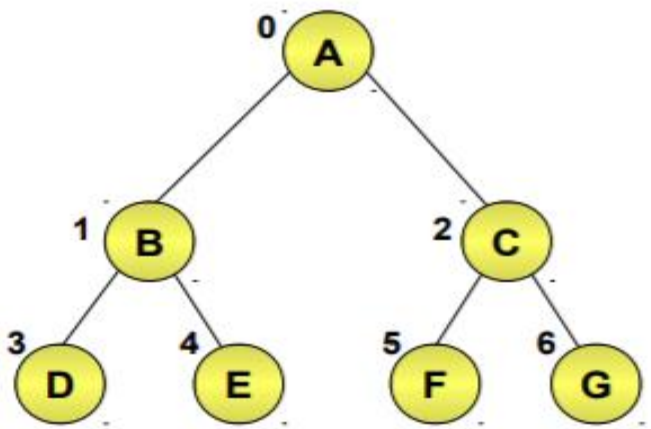


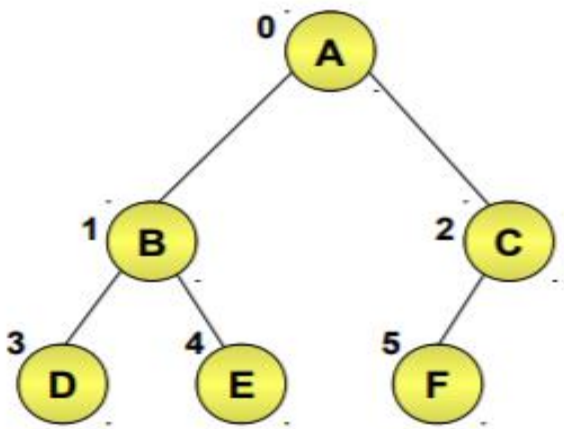
Figure: Complete Binary tree

Defining Binary Trees (Contd.)

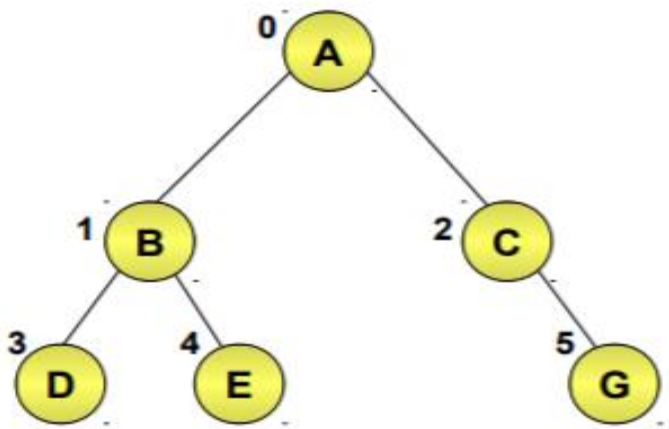
- Complete binary tree:
  - A binary tree with n nodes and depth d whose nodes correspond to the nodes numbered from 0 to n − 1 in the full binary tree of depth k.



Full Binary Tree

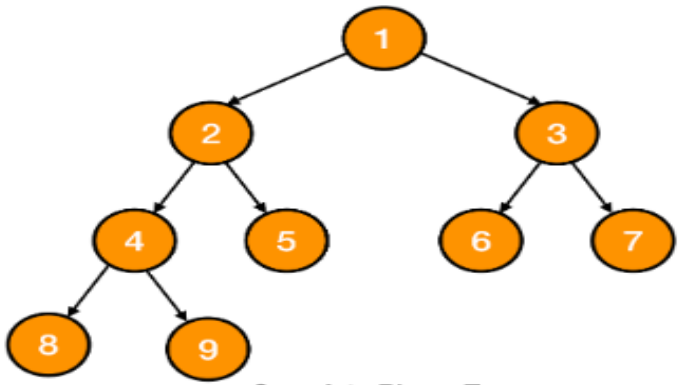


Complete Binary Tree



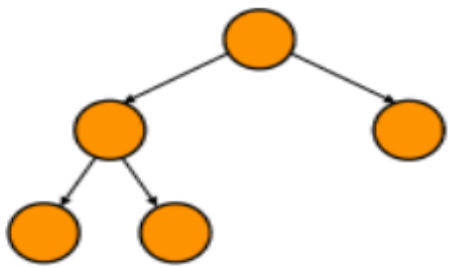
Incomplete Binary Tree

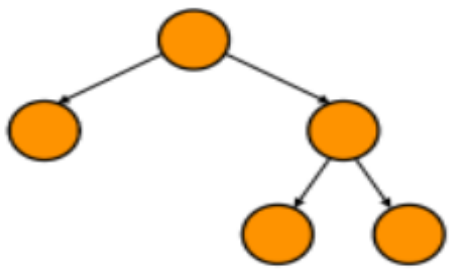
Complete Binary Tree → A binary tree which is completely filled with a possible exception at the bottom level i.e., the last level may not be completely filled and the bottom level is filled from left to right.

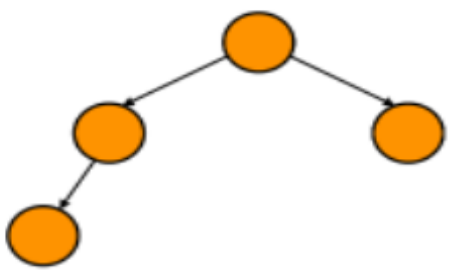


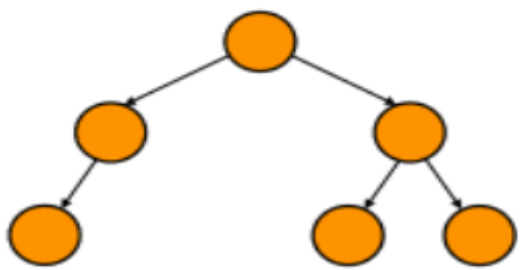
Complete Binary Tree

Let's look at this picture to understand the difference between a full and a complete binary tree.


Complete and Full

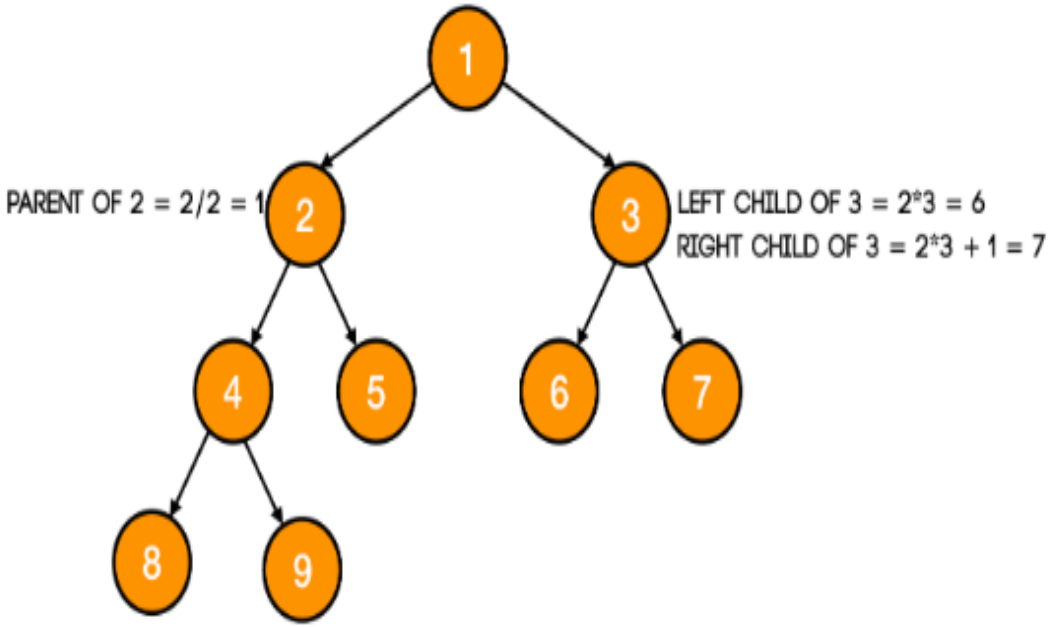

Full but NOT Complete

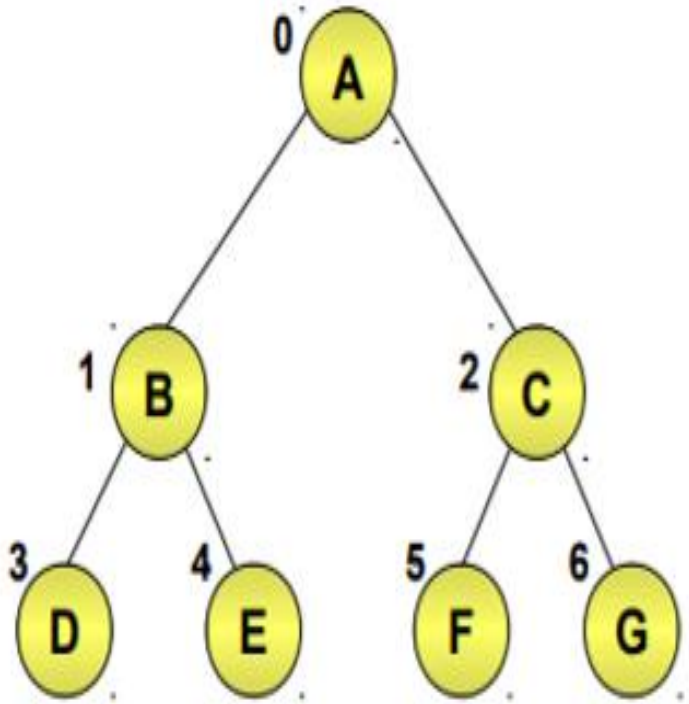

Complete but NOT Full


Neither Complete nor Full

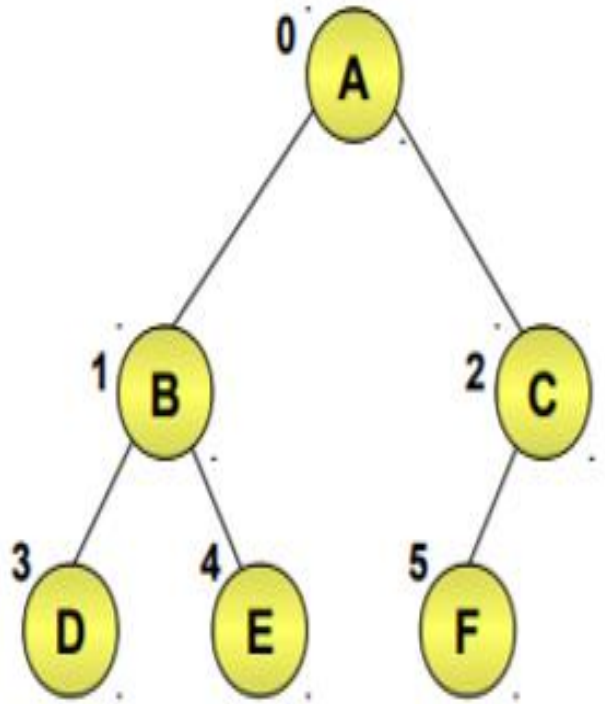
A complete binary tree also holds some important properties. So, let's look at them.

- The **parent of node i** is $\left\lfloor \frac{i}{2} \right\rfloor$. For example, the parent of node 4 is 2 and the parent of node 5 is also 2.
- The **left child of node i** is $2i$.
- The **right child of node i** is $2i + 1$

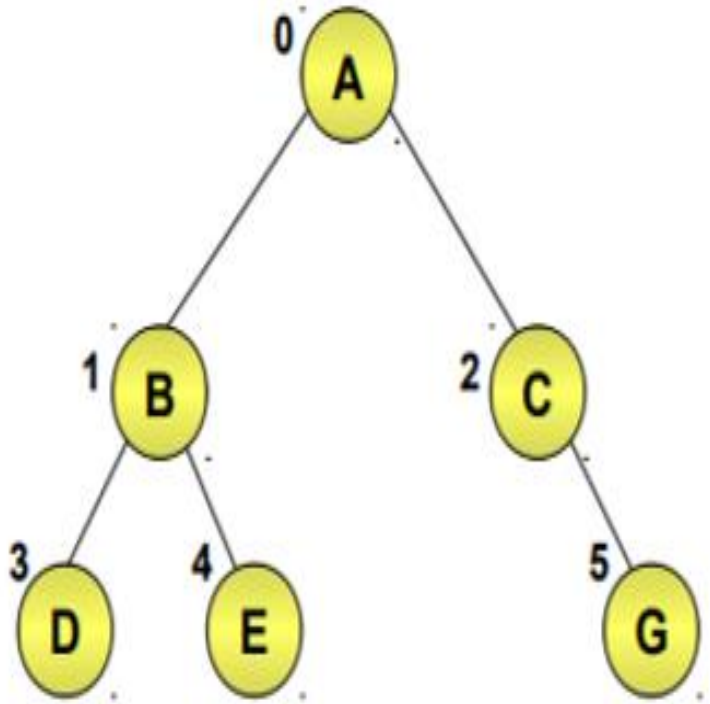




Full Binary Tree



Complete Binary Tree

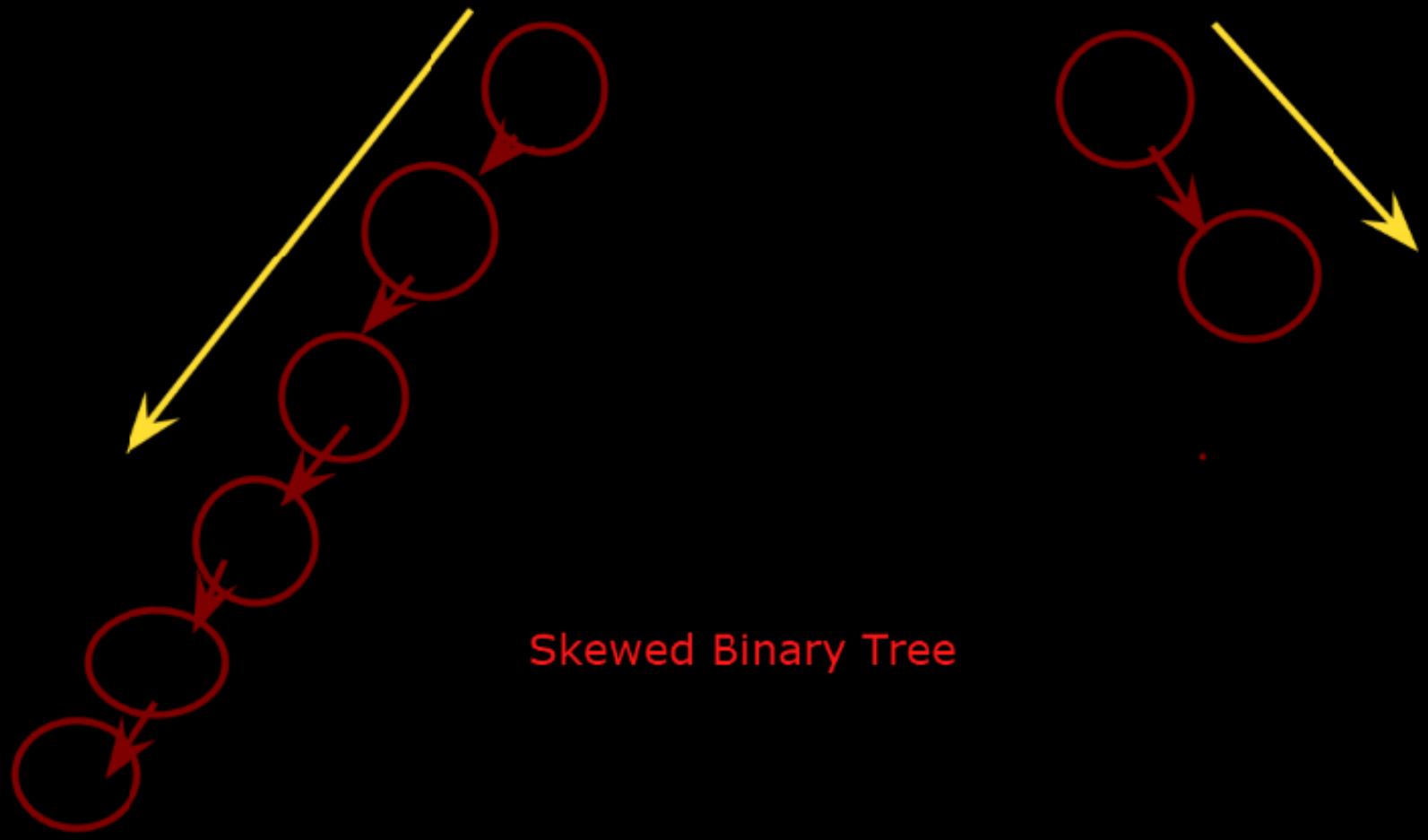


Incomplete Binary Tree

Binary Tree:

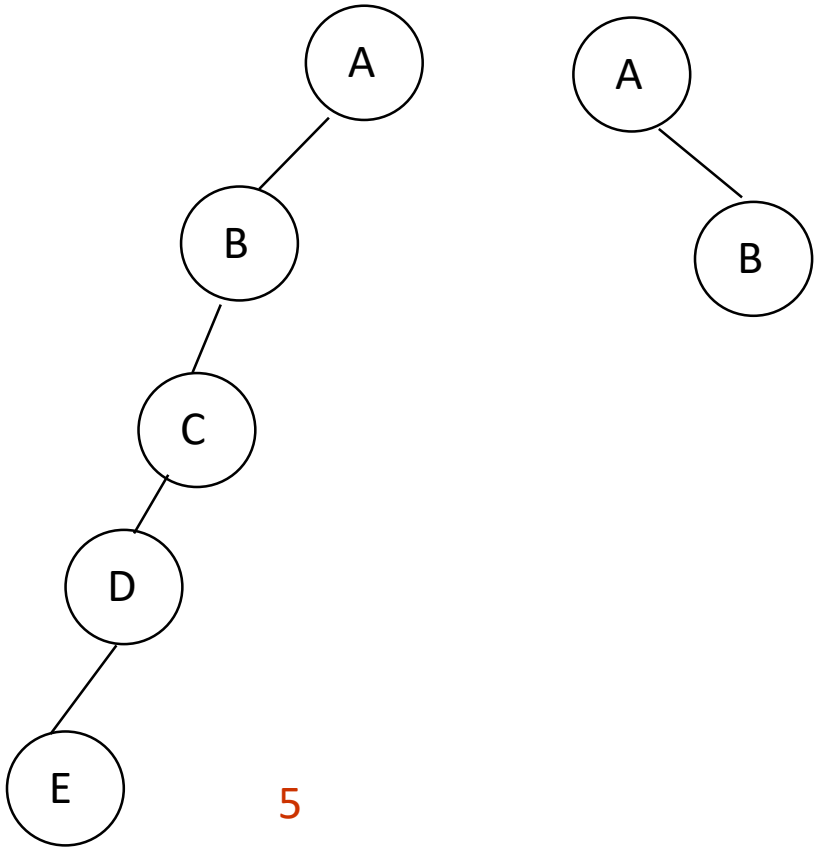
A binary tree is a tree in which every node has at most two children.

0,1,2,2

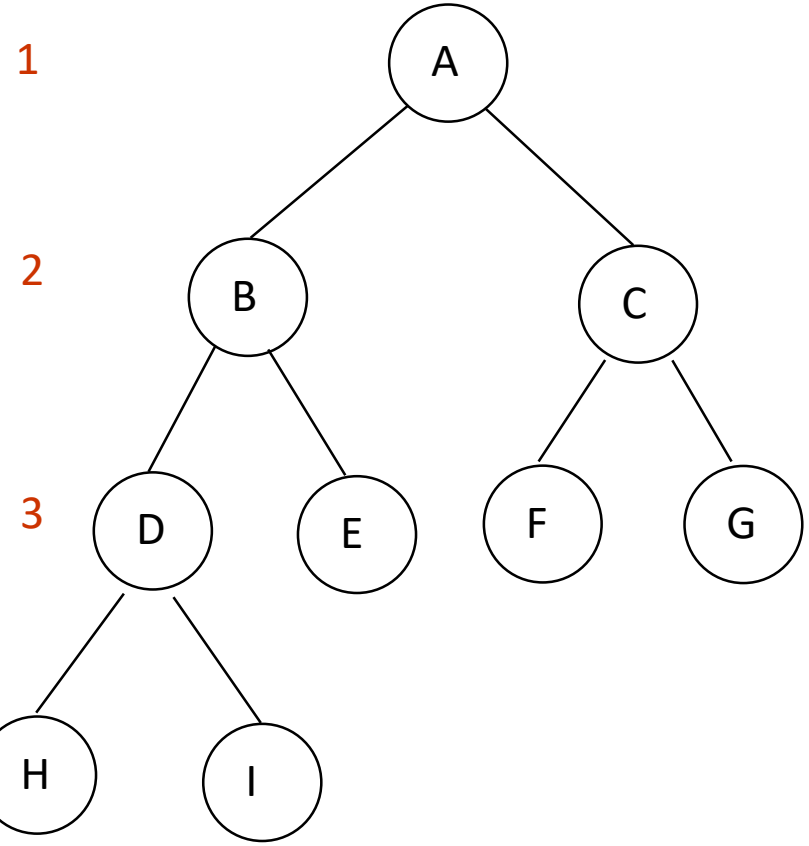


Examples of the Binary Tree

Skewed Binary Tree



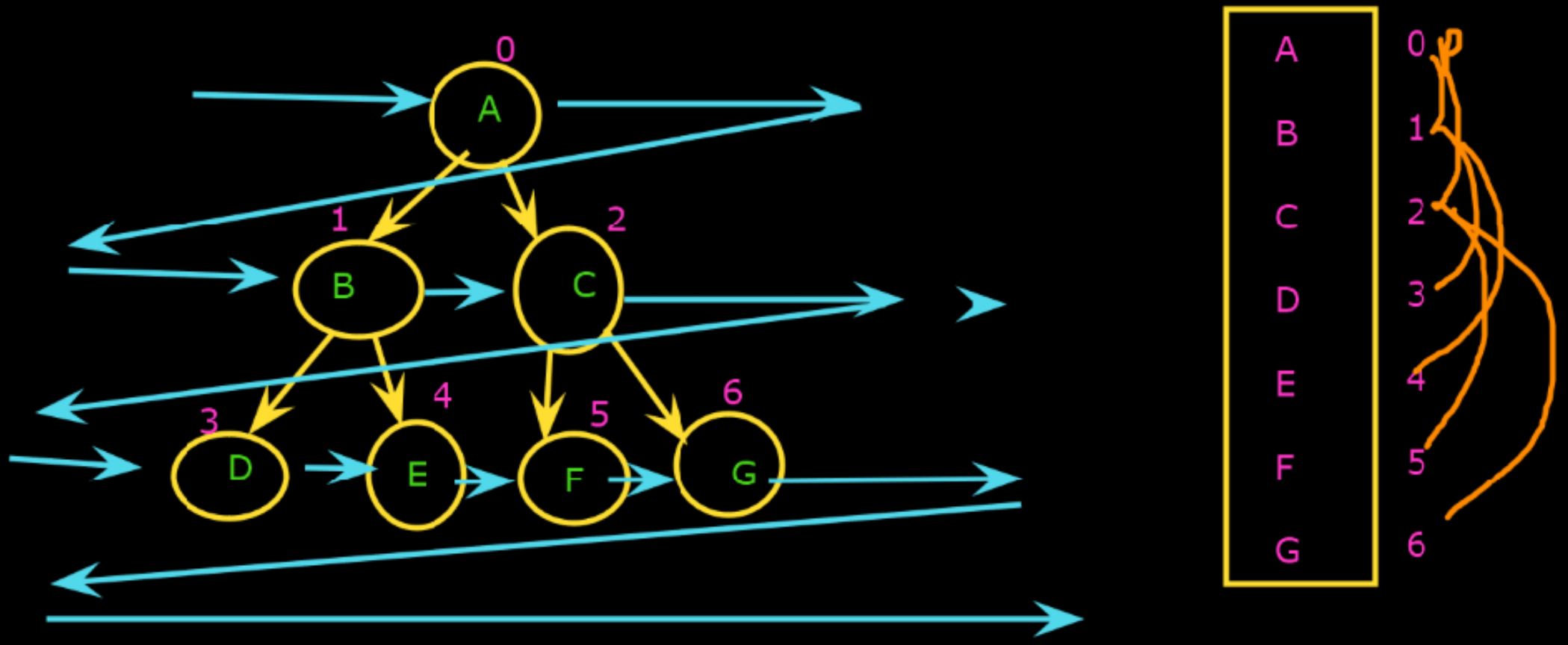
Complete Binary Tree



Binary Tree:

A binary tree is a tree in which every node has at most two children.

0,1,2,2

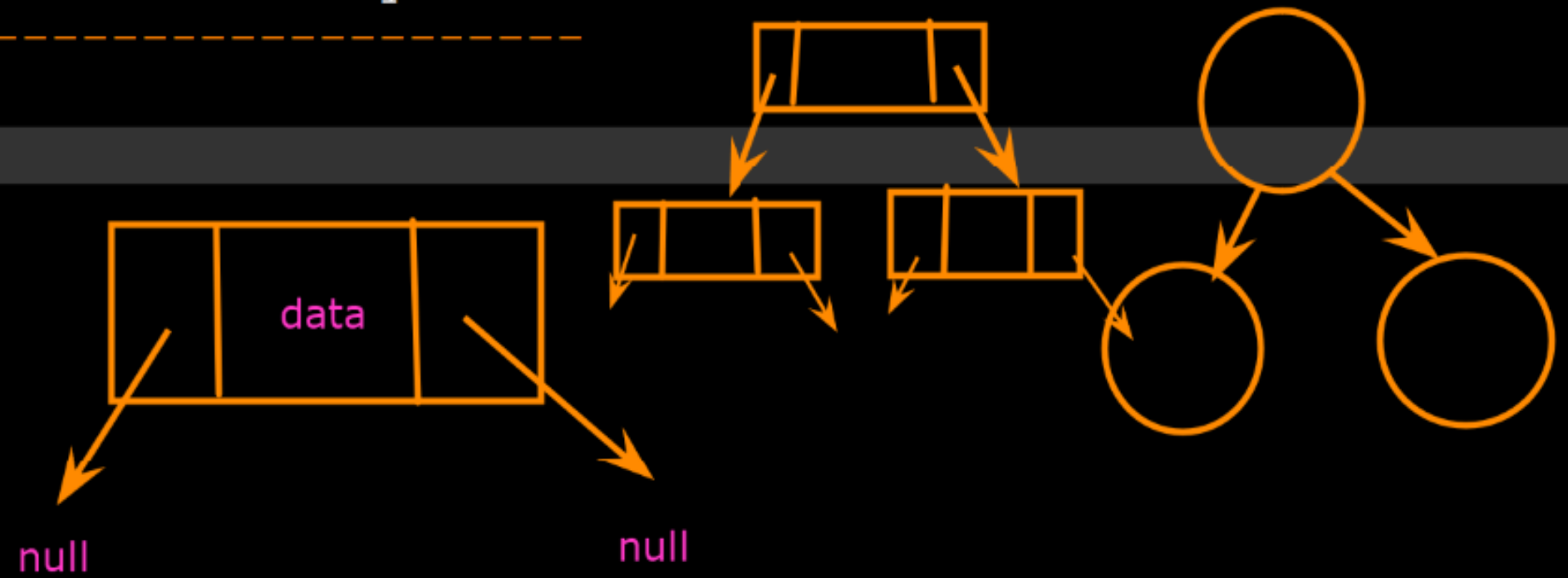


Array Representation of Binary Tree

Binary Tree:

A binary tree is a tree in which every node has at most two children.
0,1,2,2

Node structure for Binary Tree:



OPERATIONS ON TREES

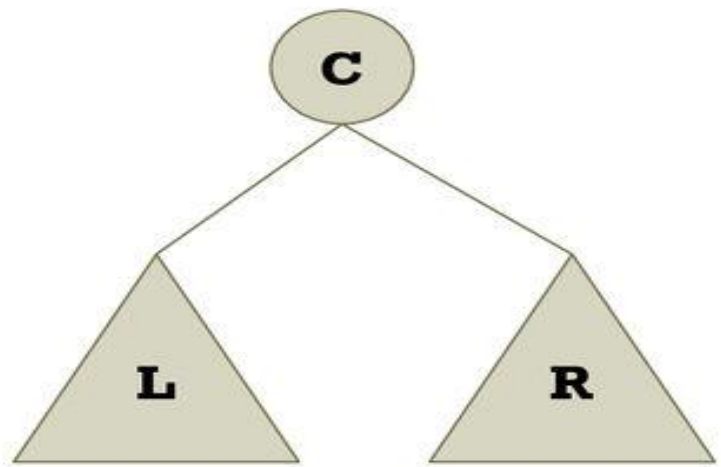
Traversing a Binary Tree

1)TRAVERSING

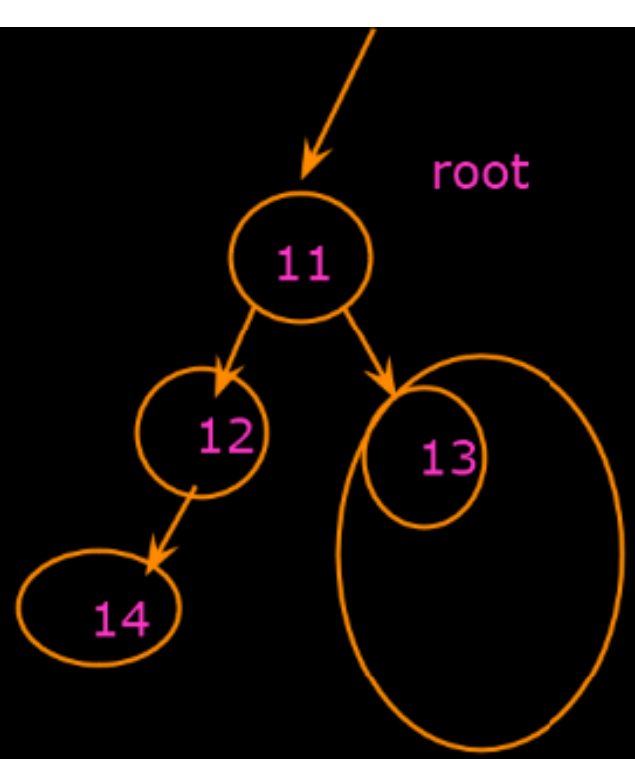
- You can implement various operations on a binary tree.
- A common operation on a binary tree is traversal.
- Traversal refers to the process of visiting all the nodes of a binary tree once.
- There are three ways for traversing a binary tree:
  - Inorder traversal
  - Preorder traversal
  - Postorder traversal

Traversal of Binary Tree

- Traversal methods
  - **Inorder traversal: LCR**
    - Visiting a left subtree, a root node, and a right subtree
  - **Preorder traversal: CLR**
    - Visiting the root node node before subtrees
  - **Postorder traversal: LRC**
    - Visiting subtrees before visiting the root node
  - **Level order traversal**

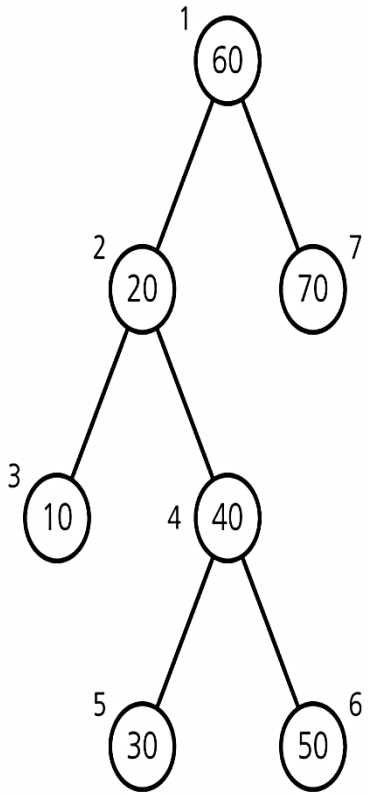


```
    System.out.println(root.data+ " ");
    printInorder(root.right);

    }

    void printPreorder(Node root)
{
    if(root == null)
        return;

    System.out.println(root.data+ " ")
    printPreorder(root.left);
    printPreorder(root.right;

}
```

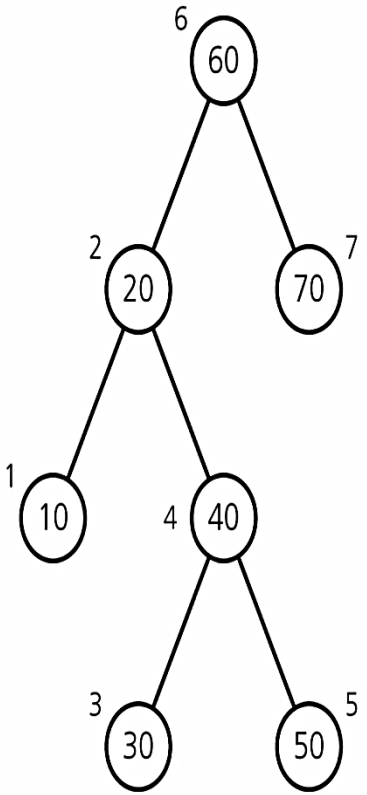



```
void printPostorder(Node root)
{
    if(root == null)
        return;
```

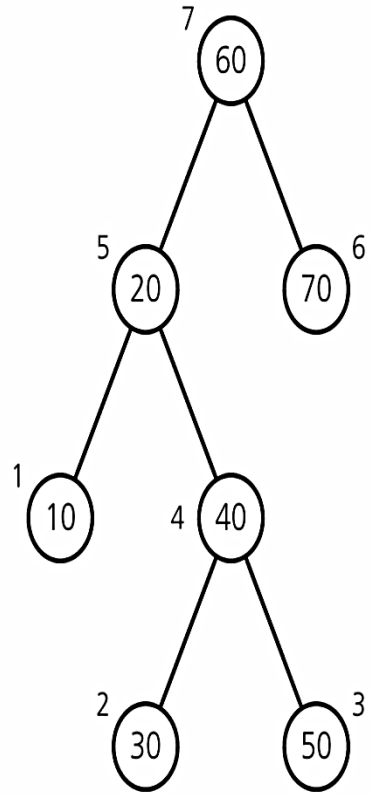
Binary Tree Traversals



(a) Preorder: 60, 20, 10, 40, 30, 50, 70



(b) Inorder: 10, 20, 30, 40, 50, 60, 70



(c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

InOrder(root) visits nodes in the following order:

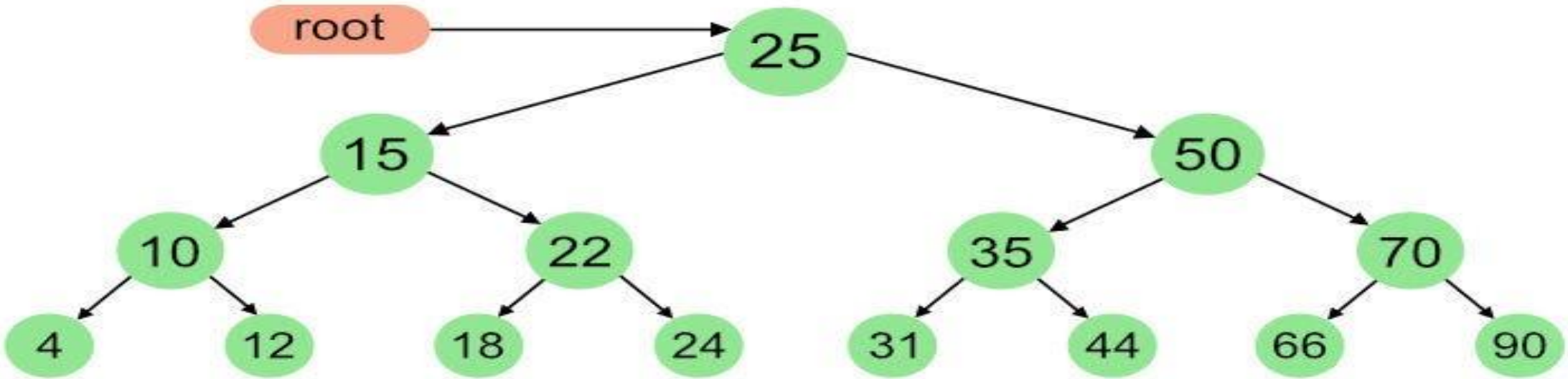
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Thanks