

# Data Structures and Algorithms

## Tree



*Prepared by:*

**Eng. Malek Al-Louzi**

School of Computing and Informatics– Al Hussein Technical University

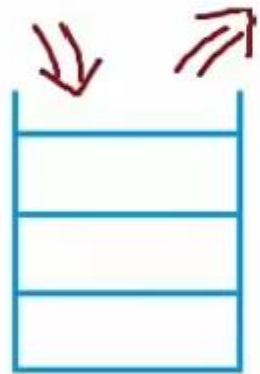
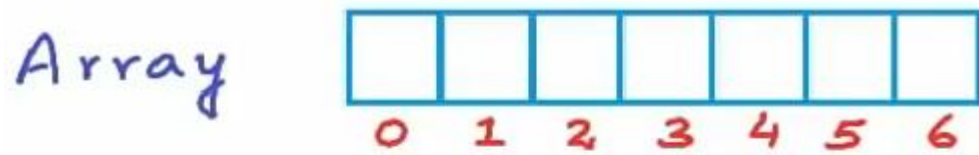
Spring 2021/2022

# Outlines

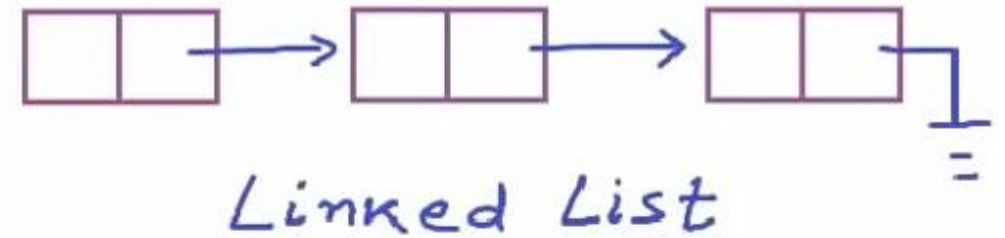
- Introduction to Trees
- Properties of Tree
- Tree Applications
- Binary Tree
- Binary Tree Implementation
- Binary Tree Traversal

# Introduction to Trees

- So far, we learned about Linear data structures:



Stack



Queue

# Introduction to Trees

- Linear data Structure:
  - Data is arranged in sequential manner.
  - There is a logical start and a logical end.
  - An element can have a next element and a previous element.
- Data structures are a ways to store and organize data in computers.
- For different kinds of data, we use different kinds of data structure.

# Introduction to Trees

- Our choice of data structure depends on number of factors:



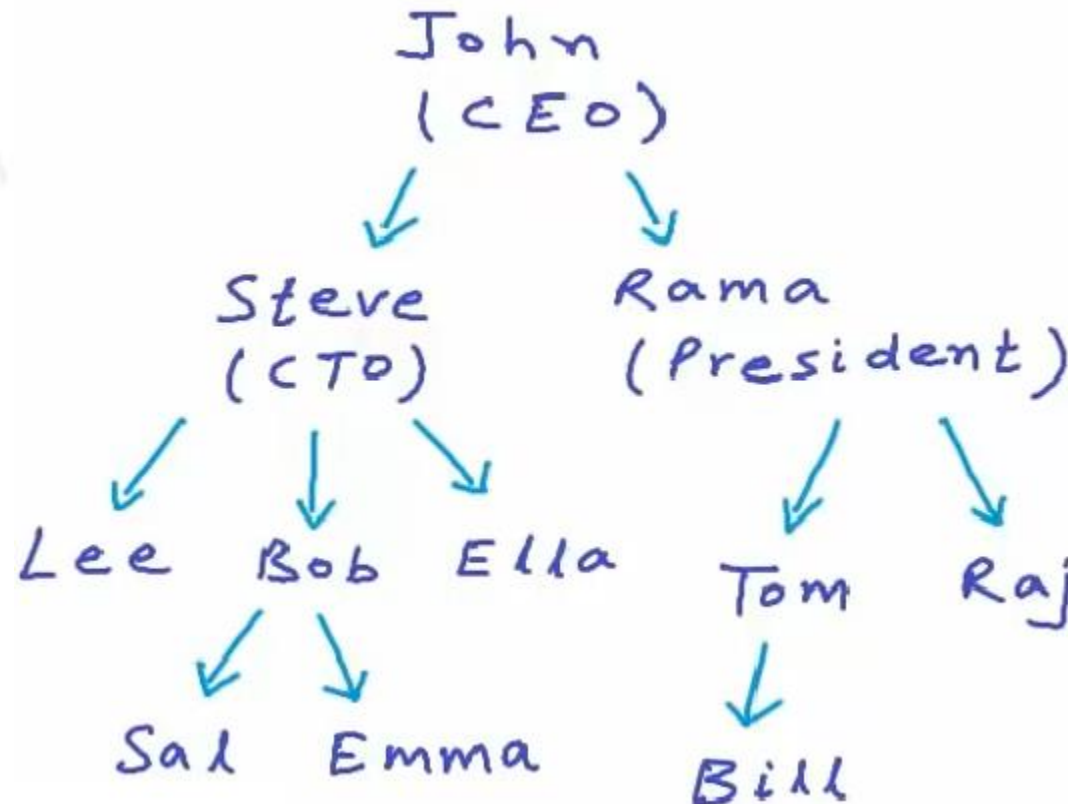
How should i decide which  
data structure to use?



- What needs to be stored?
- Cost of operations
- Memory usage
- Ease of implementation

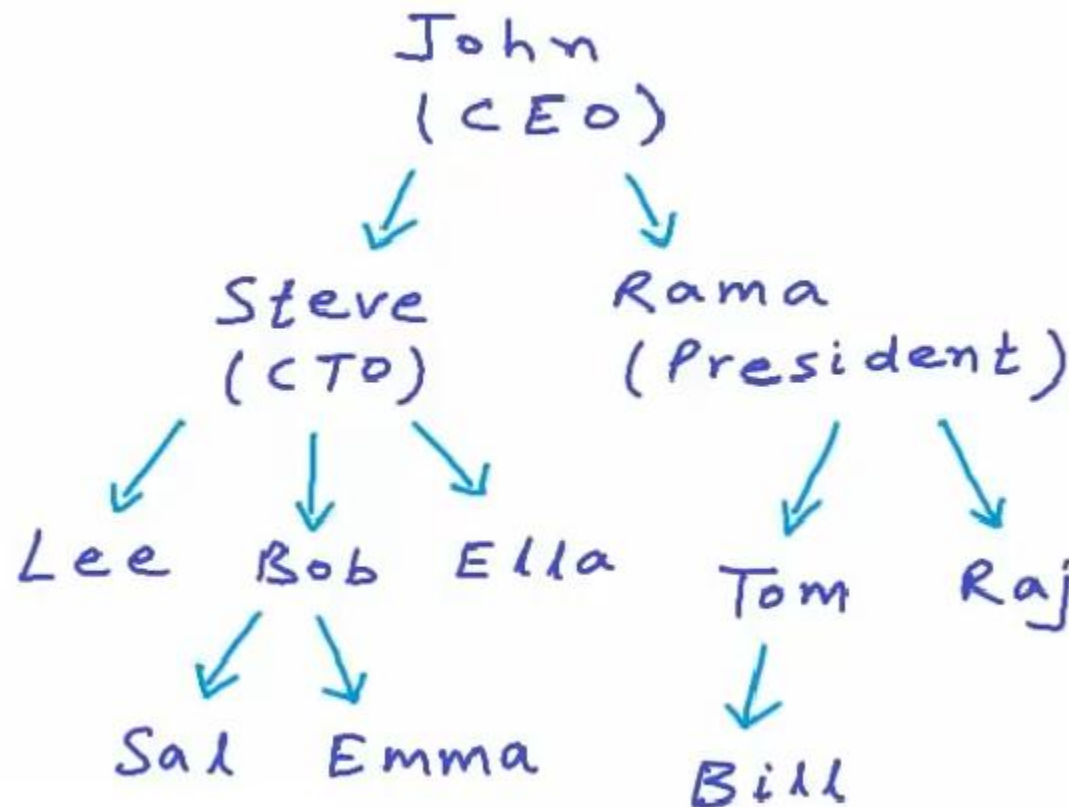
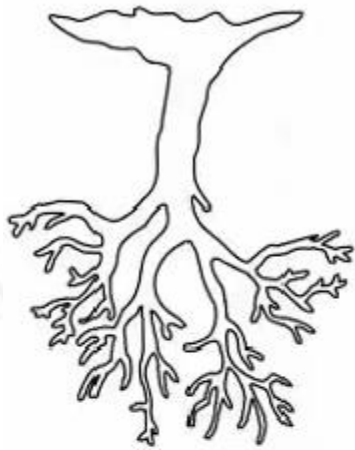
# Introduction to Trees

- Tree is a data structure used to represent hierarchical data.
- The following is an organizational hierarchy of a company.



# Introduction to Trees

- The previous organizational hierarchy is a Tree.
- You have to look at it upside down.



# Introduction to Trees

- The root is at top.
- Logical representation of a Tree data structure is always like this.
  - Root at top and branching downward direction.
- Tree is an efficient way to store and organize data that is naturally hierarchal.

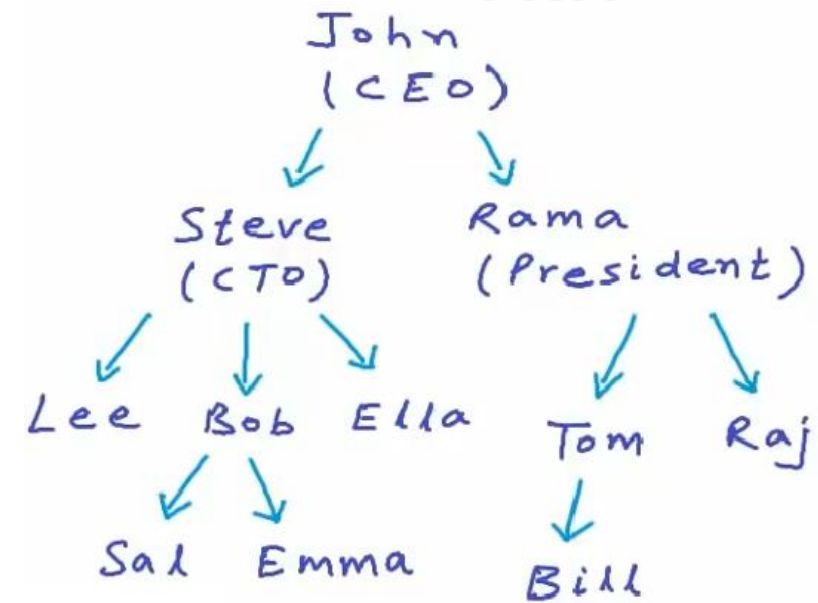
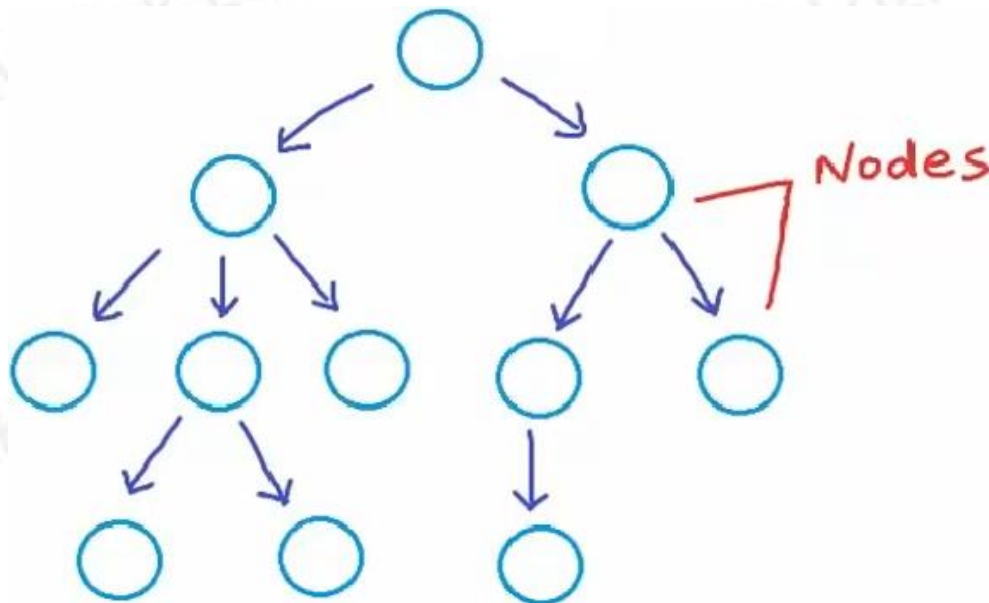


# Tree Definition as a Logical Model

- So, what is a Tree?

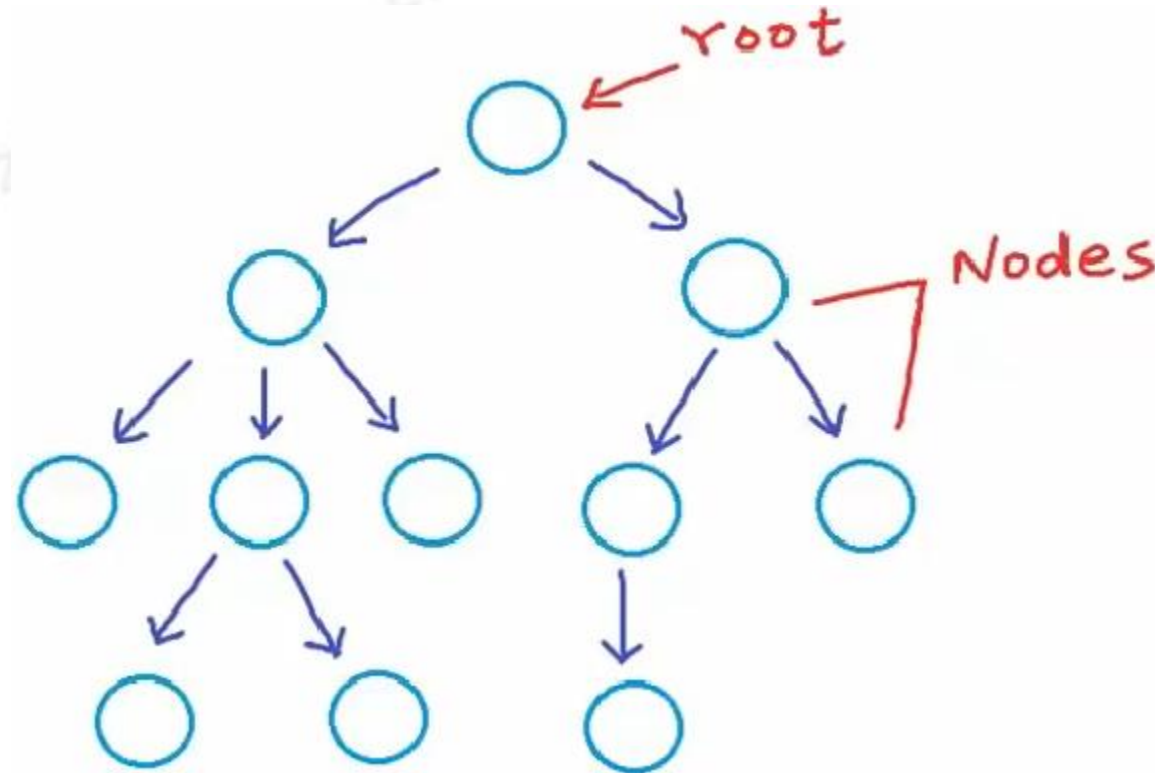
A collection of entities called nodes  
linked together to simulate a hierarchy.

- Tree is a nonlinear data structure; it is a hierarchal data structure.



# Tree Definition as a Logical Model

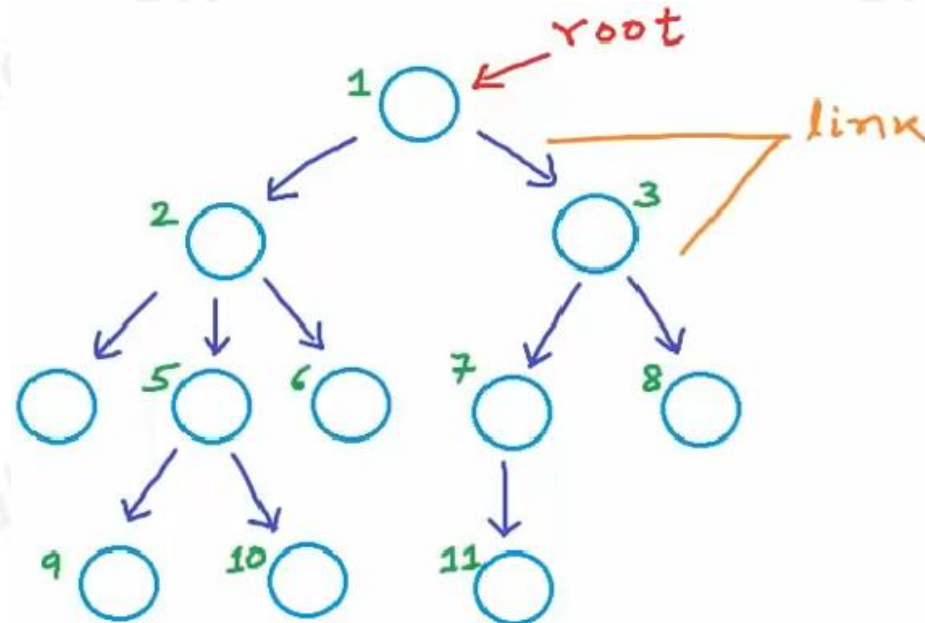
The topmost node called root of the tree.



# Tree Definition as a Logical Model

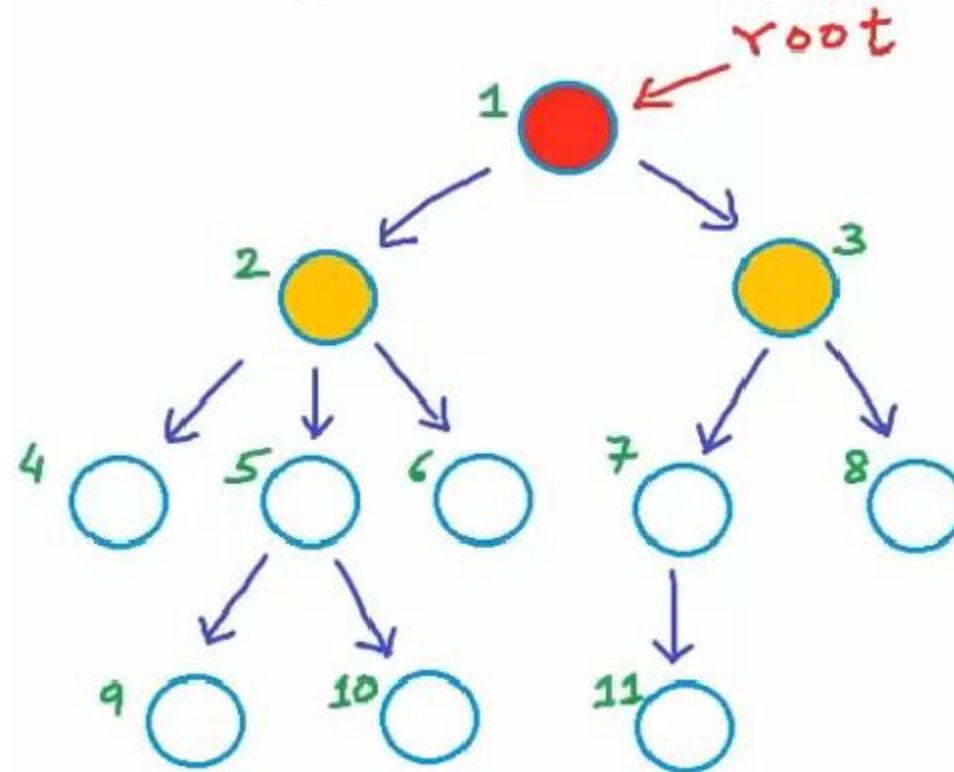
Each node will contain data of any type and may contain link or reference to other nodes that can be called its children.

- Each arrow is called a link.
- Links are **not** bidirectional.



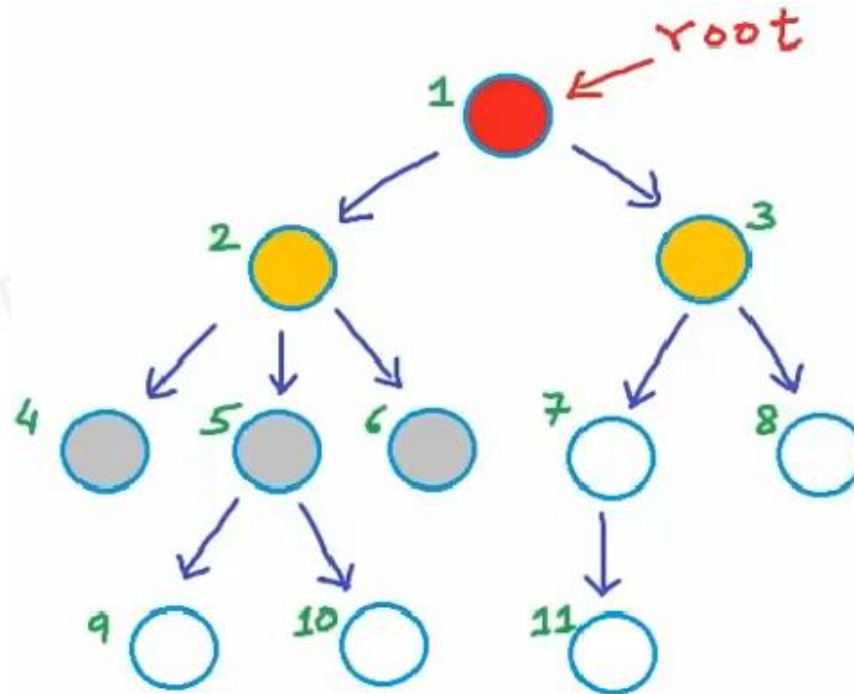
# Vocabularies that used with Tree data structure

- Node 2 and 3 are **children** of Node 1.
- Node 1 is **parent** of nodes 2 and 3.



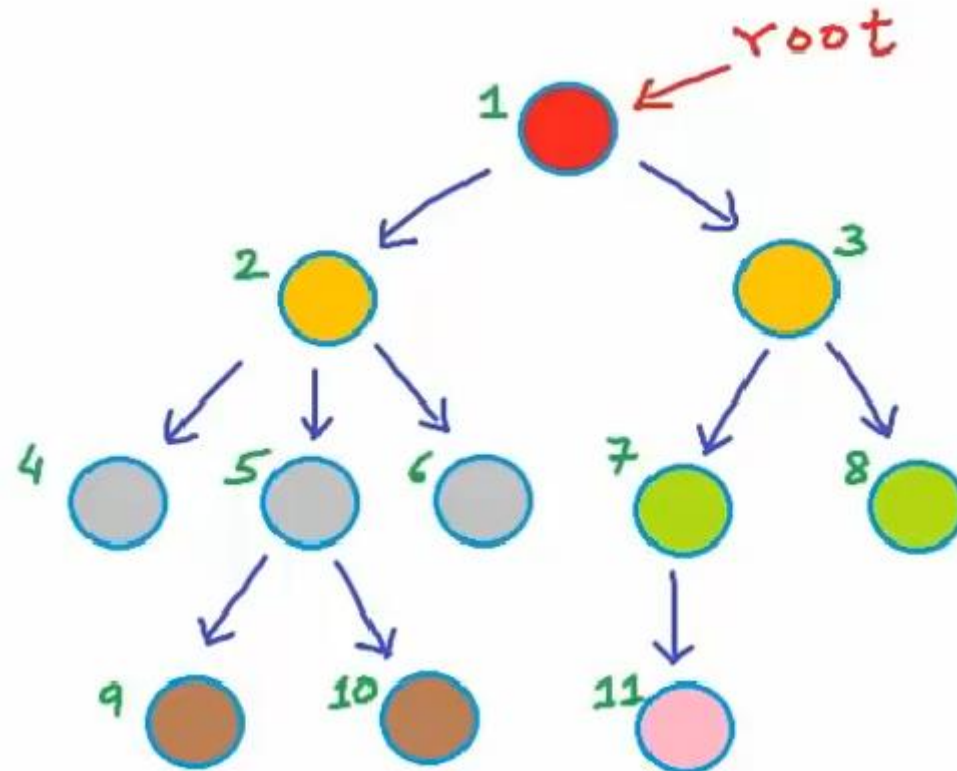
# Vocabularies that used with Tree data structure

- Node 4, 5 and 6 are children of Node 2.
- Node 2 is child of Node 1 and parent of nodes 4, 5 and 6.



# Vocabularies that used with Tree data structure

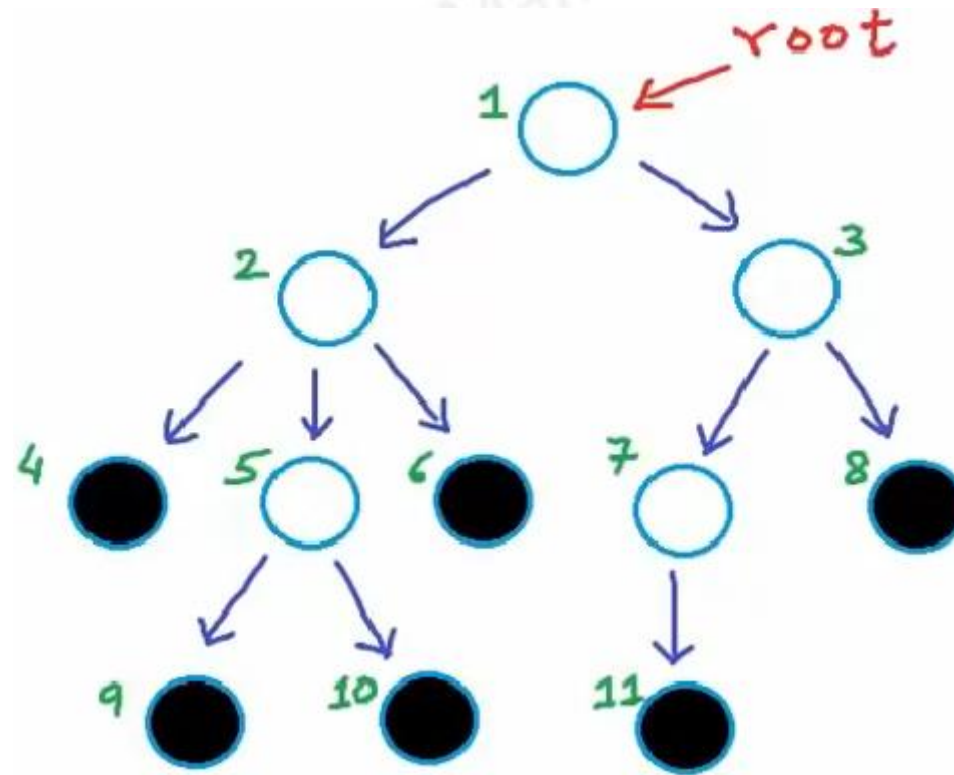
- Children of same parent are called **sibling**.
- **Root** is the only Node without parent.





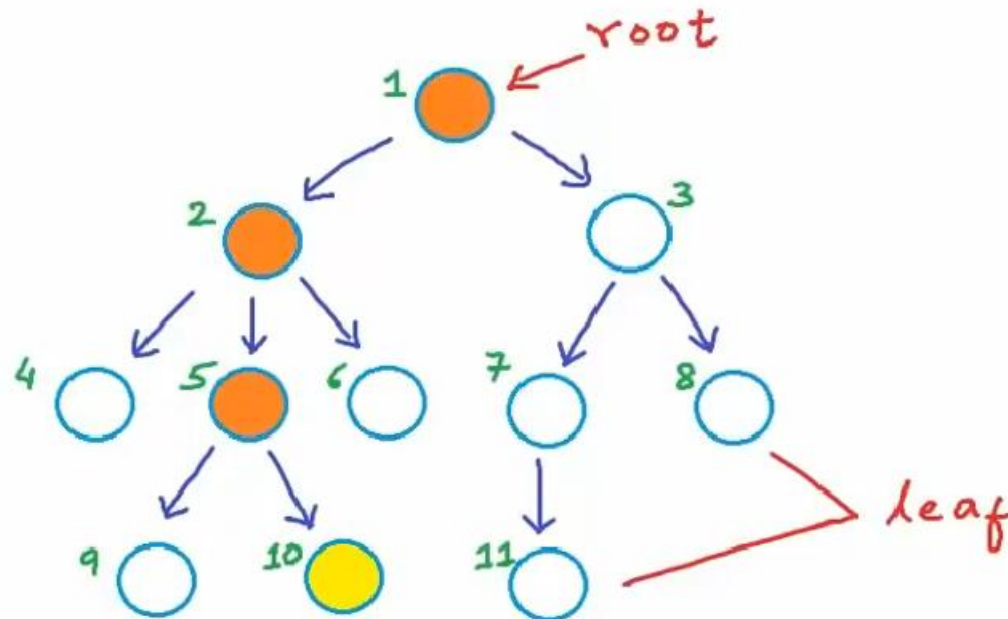
# Vocabularies that used with Tree data structure

- Any Node in the tree doesn't have a child is called **leaf Node**.



# Vocabularies that used with Tree data structure

- If we can go from node A to node B, then A is **ancestor** of B and B is **descendant** of A.
- In the following tree: 1, 2 and 5 are all ancestors of 10, and 10 is descendant of all these nodes.





# Vocabularies that used with Tree data structure

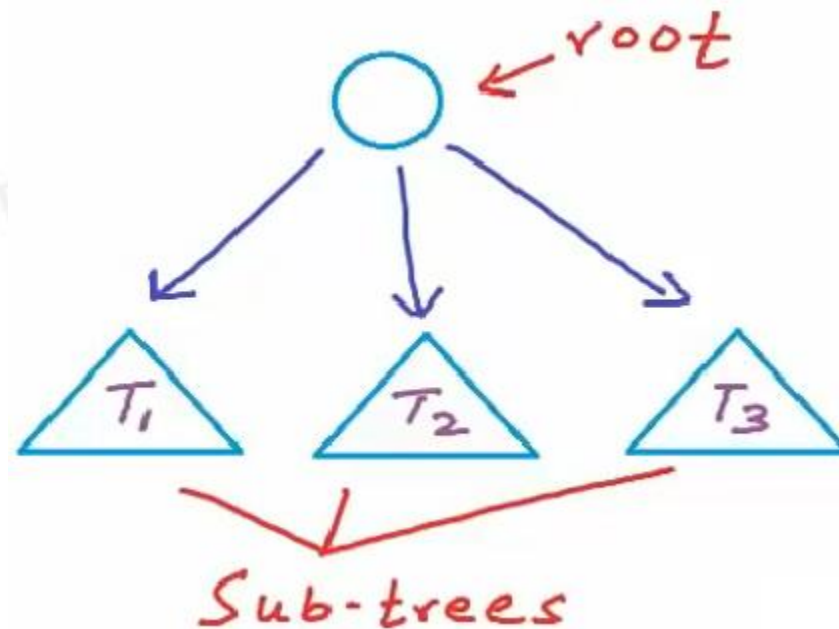
## ► Summary of the vocabularies:

root  
 children  
 Parent  
 Sibling → have same parent  
 leaf → has no child

If we can go from A to B  
 A is ancestor of B  
 B is descendent of A

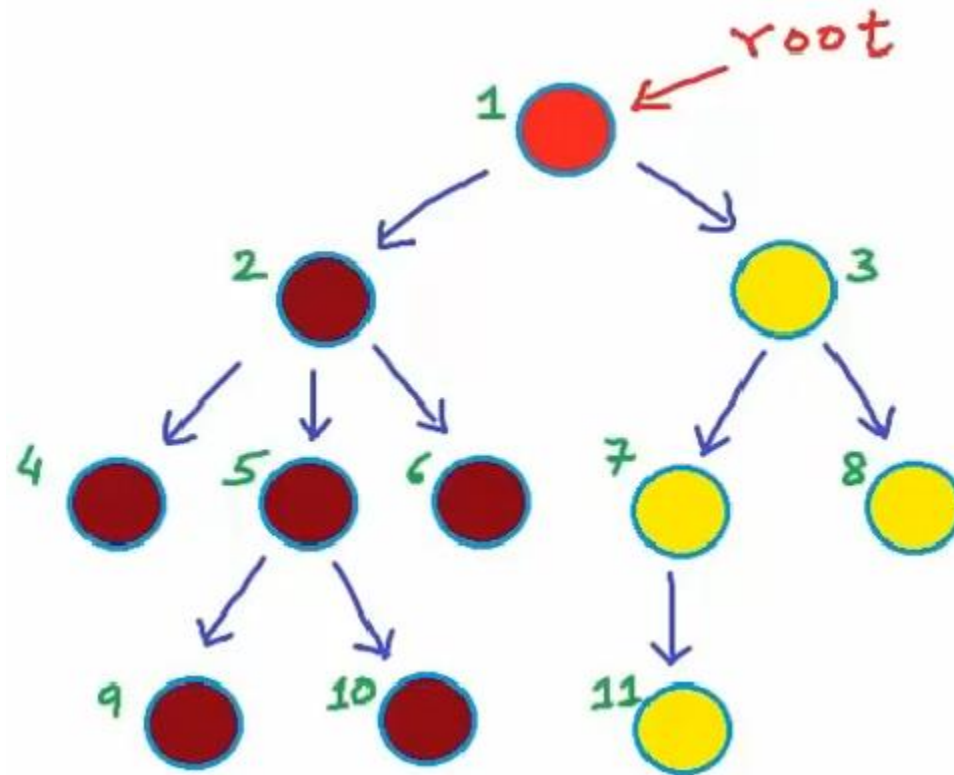
# Properties of Tree

- Tree can be called a **recursive** data structure.
  - Recursion is reducing something in a self similar manner.
  - Can be defined recursively as a structure that consists of a distinguished node called root and sub trees.



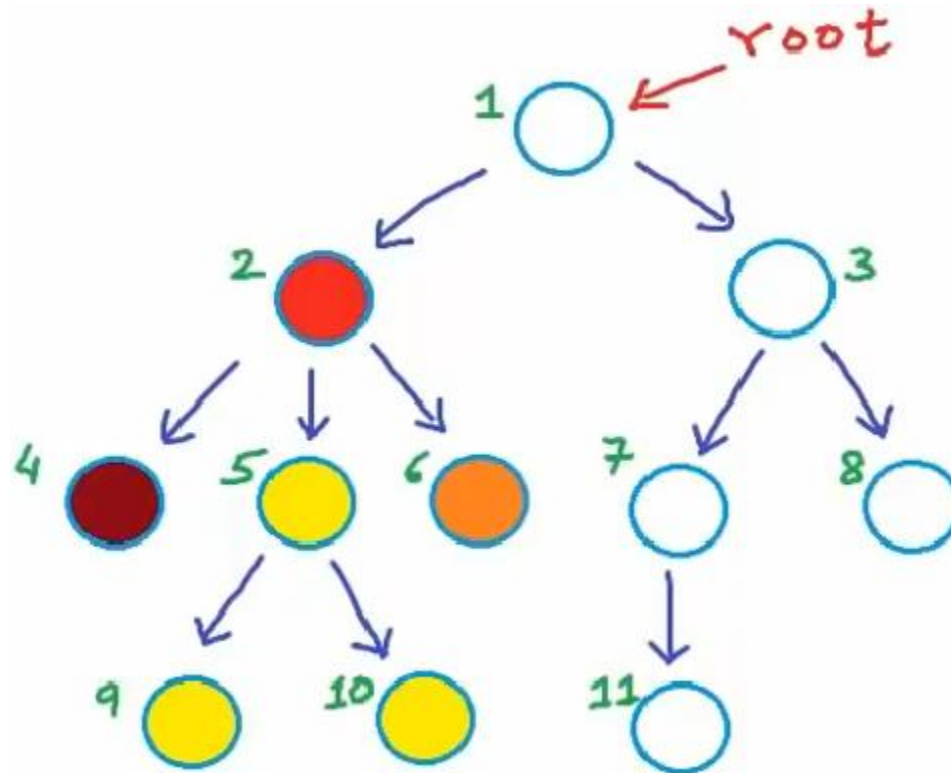
# Properties of Tree

- In the following Tree, we have Two sub-trees for the root node.



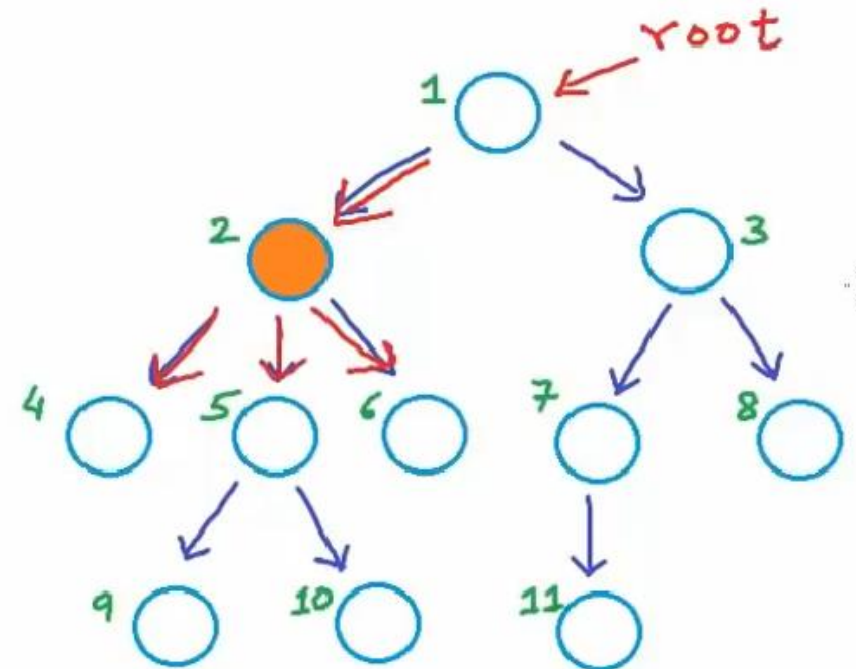
# Properties of Tree

- We can further split left sub tree, node 2 is a root node of this sub tree. It has Three sub trees.



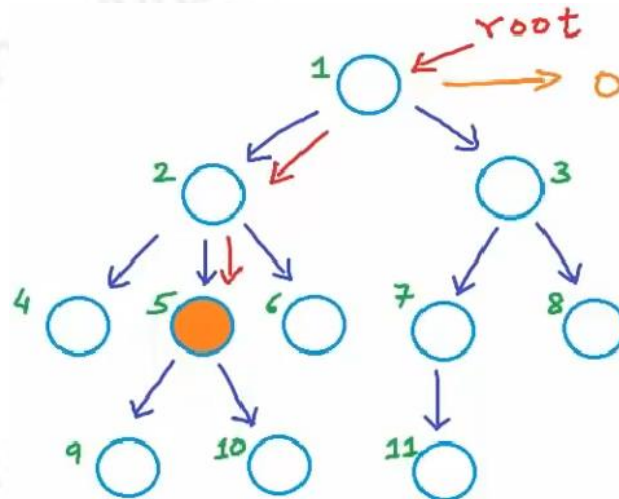
# Properties of Tree

- In a tree with  $N$  nodes, there will be exactly  $N-1$  links.
- All nodes except the root node have exactly One incoming link.
- In node 2, there is only One incoming link and Three outgoing links.
- One link for each parent child relationship.



# Properties of Tree

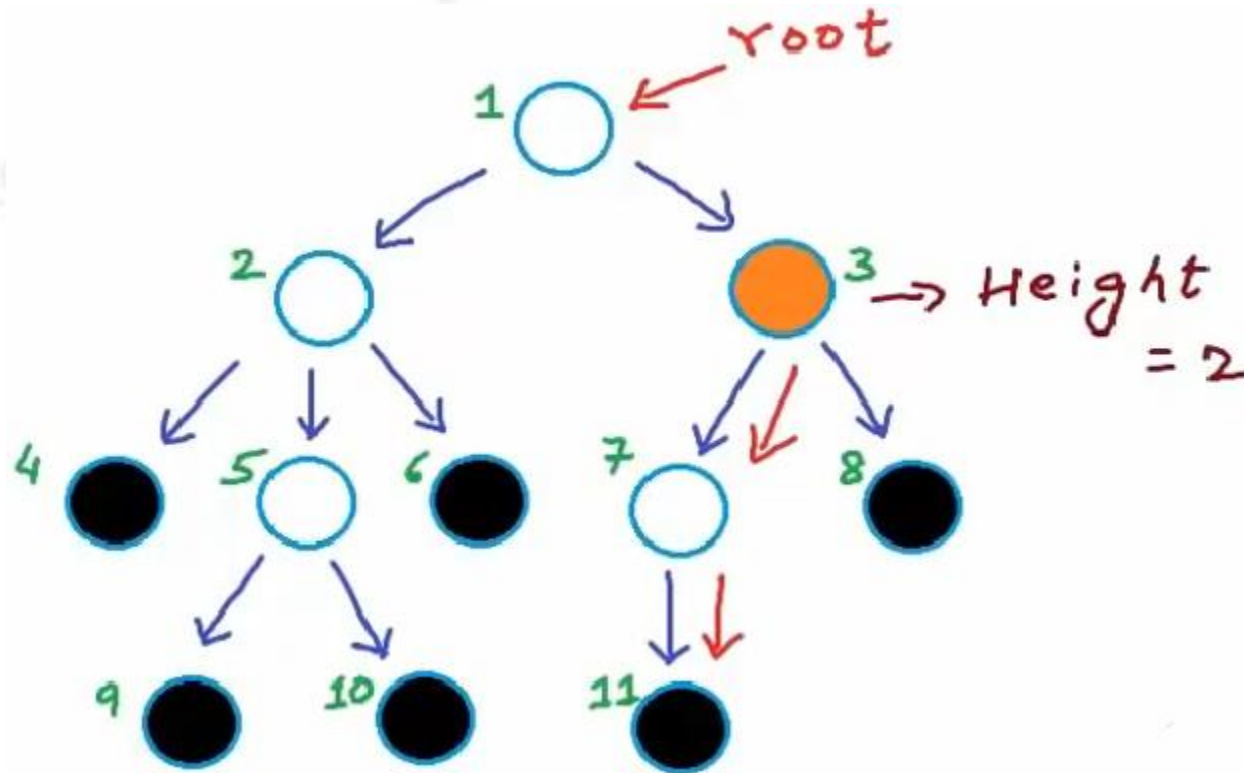
- Depth of node X is length of the path from root to node X.
- Each link in the path is One unit length.
- We can also say Depth of X is number of links in the path from root to X.
- In the following tree, depth of root node is 0 and depth of node Five is 2.





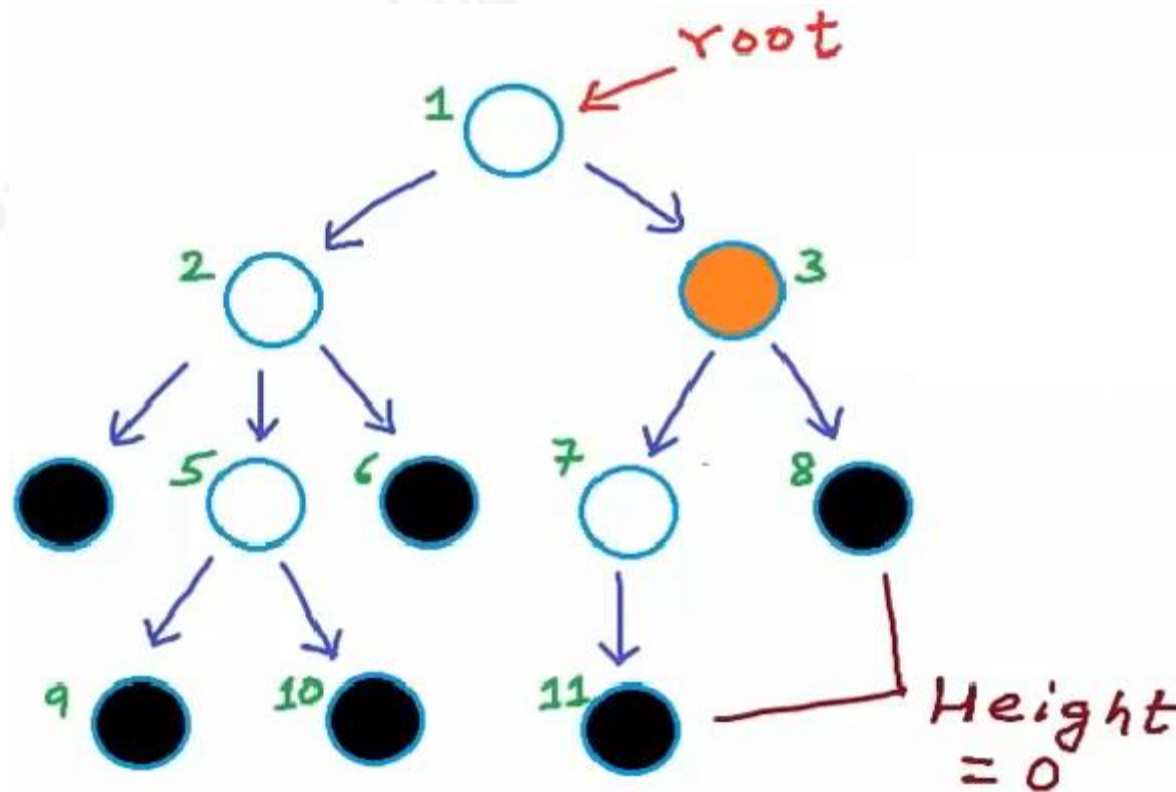
# Properties of Tree

- Height of Node X is number of links in the longest path from X to a leaf node.



# Properties of Tree

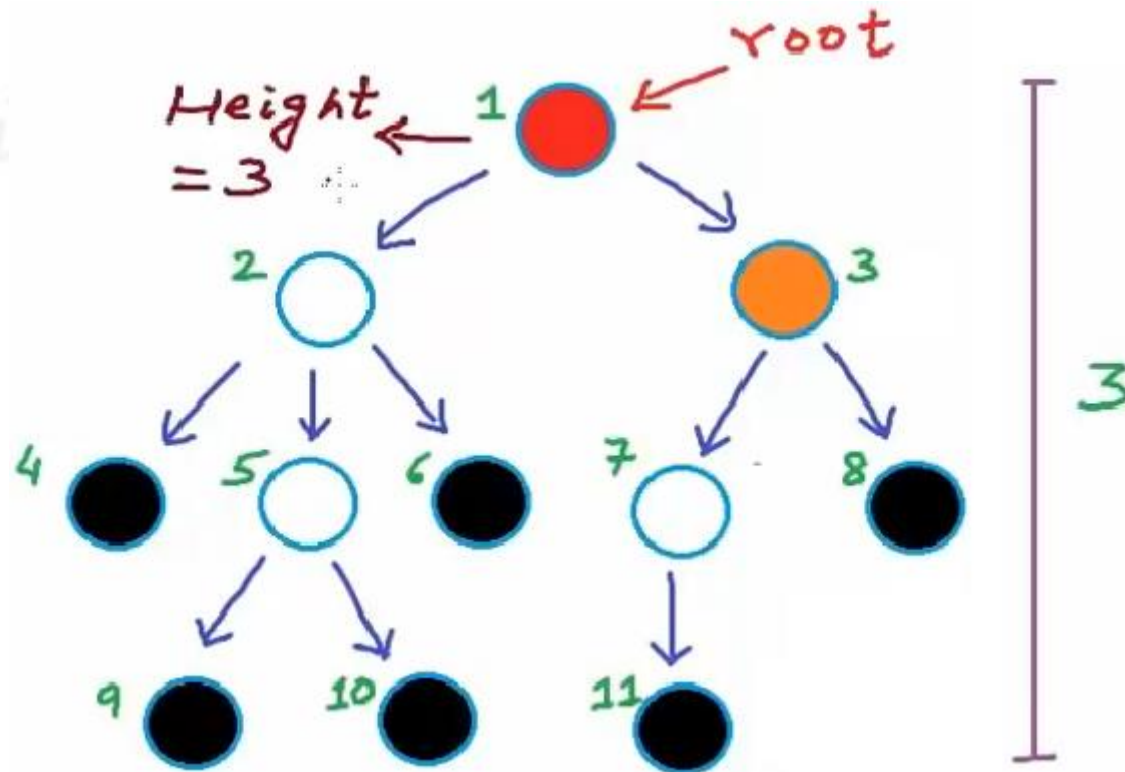
- Height of leaf Nodes is Zero.





# Properties of Tree

- Height of tree is defined as height of root node.
- Height of root Node in this tree.



# Properties of Tree

## ► Depth and Height summary:

Depth of  $x$  =

No. of edges in path  
from root to  $x$

Height of  $x$  =

No. of edges in longest  
path from  $x$  to a leaf

Height of tree =

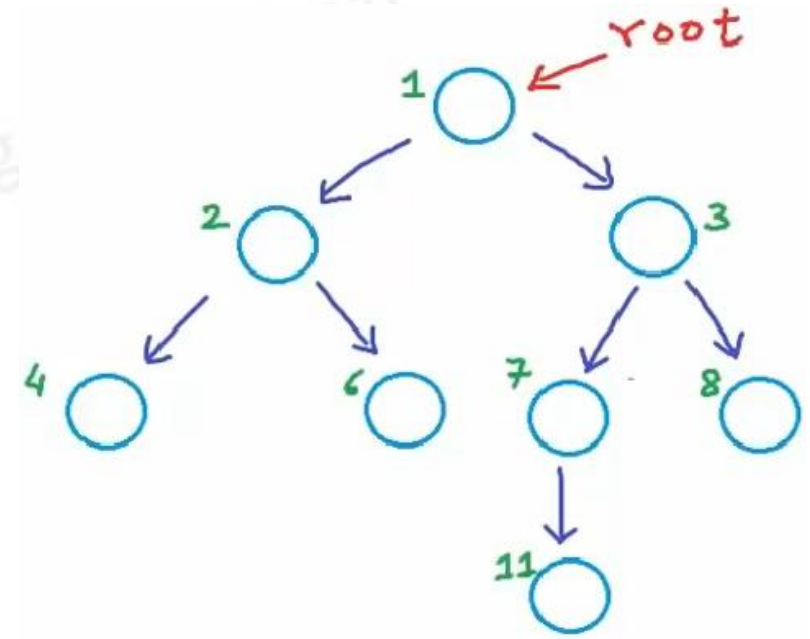
Height of root node.

# Tree Applications

- Storing naturally hierarchical data
  - Example: Files System.
- Organize data for quick search, insertion and deletion.
  - Example: Binary search Tree.

# Binary Tree

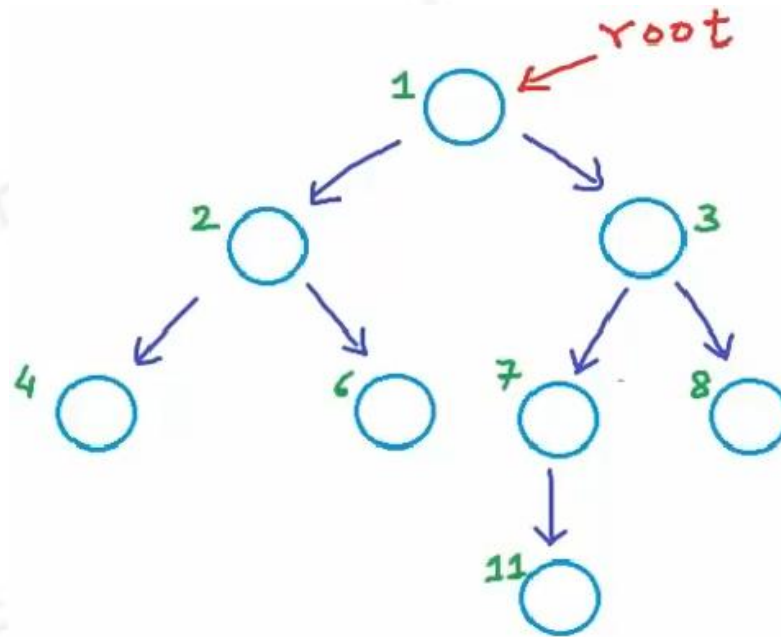
- Based on properties, trees are classified into different categories.
- Different kinds of trees are used in different scenarios.
- Simplest and most common kind is a tree with a property that any node can have at most Two children.
- This is Called a **Binary Tree**.



# Binary Tree

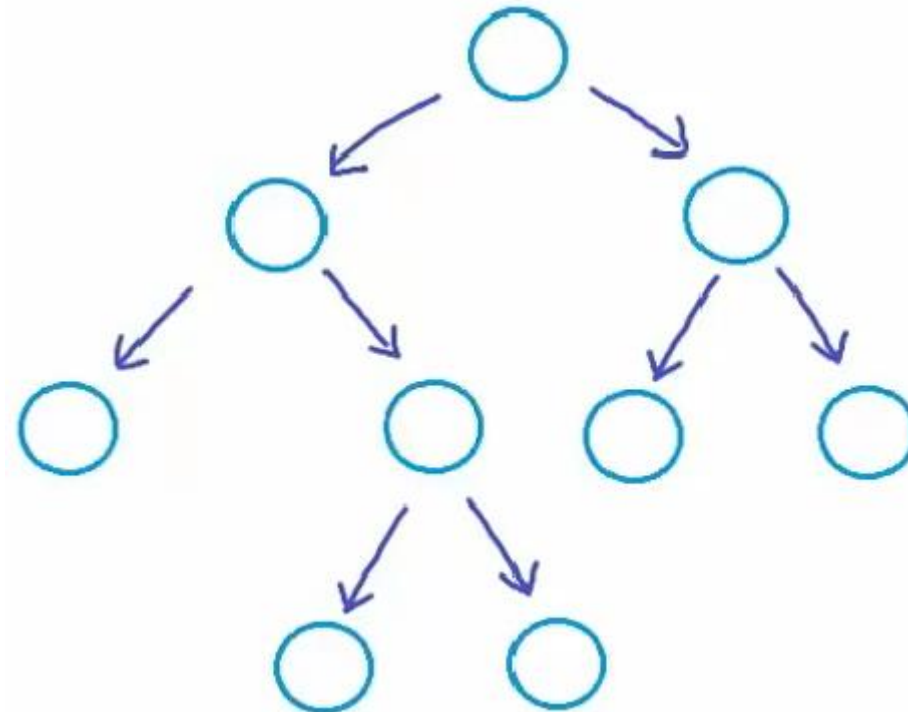
Binary Tree

↓  
a tree in which each  
node can have at most  
2 children



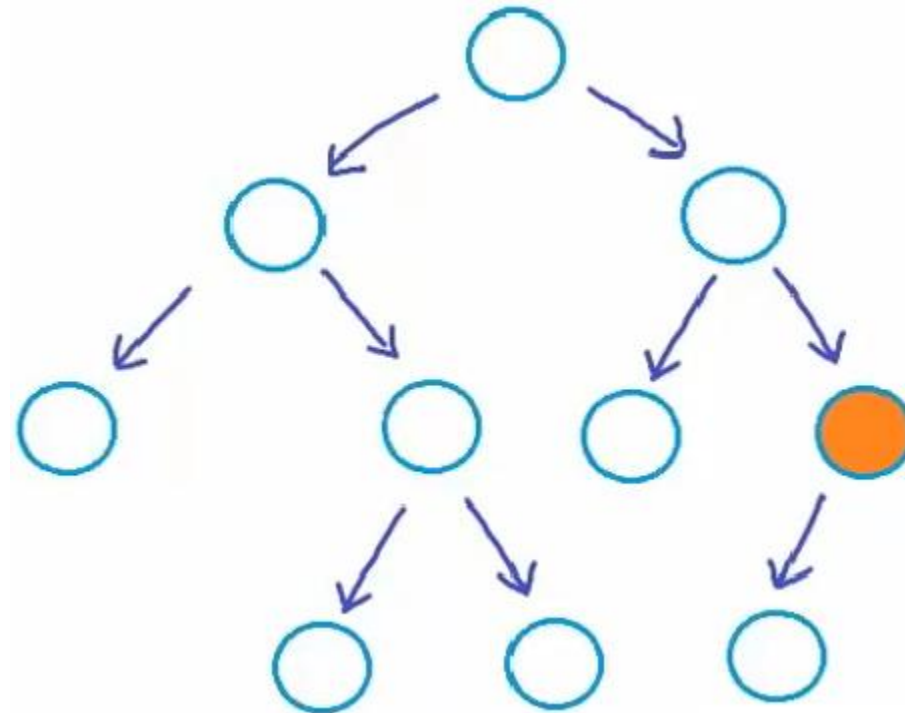
# Binary Tree

- In the following Tree, nodes have either Zero or Two children.



# Binary Tree

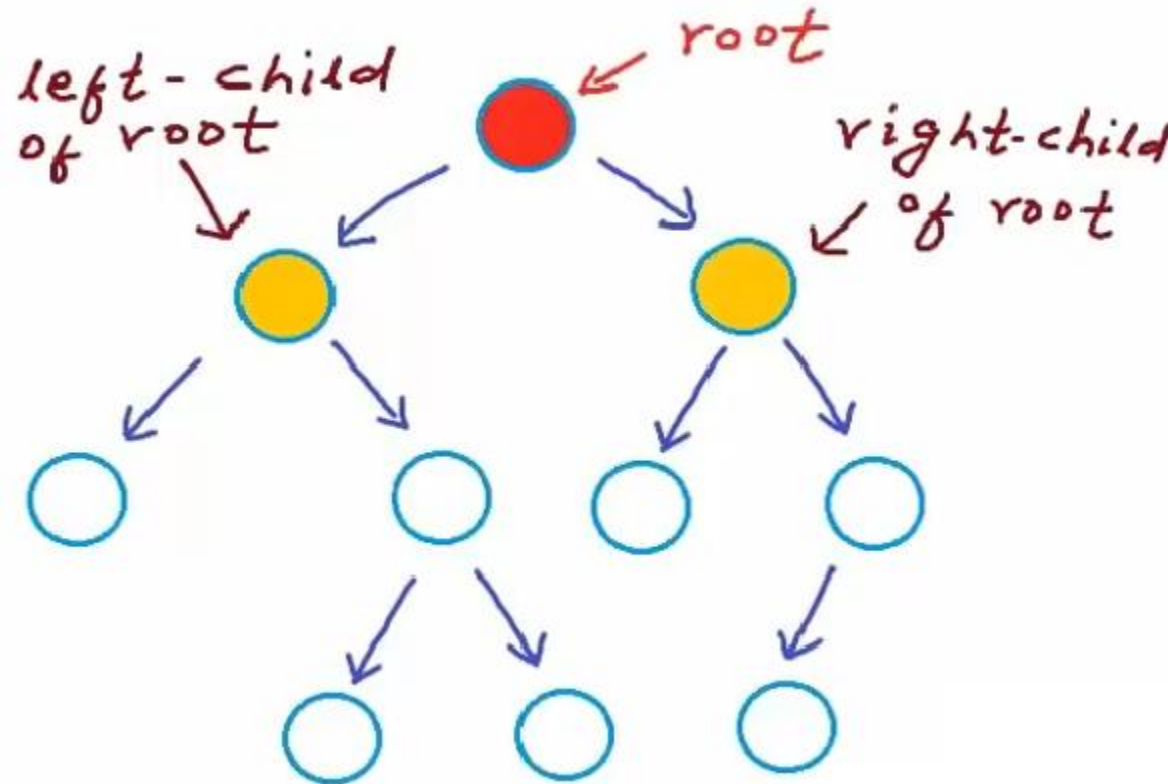
- We could have a node with just One child.





# Binary Tree

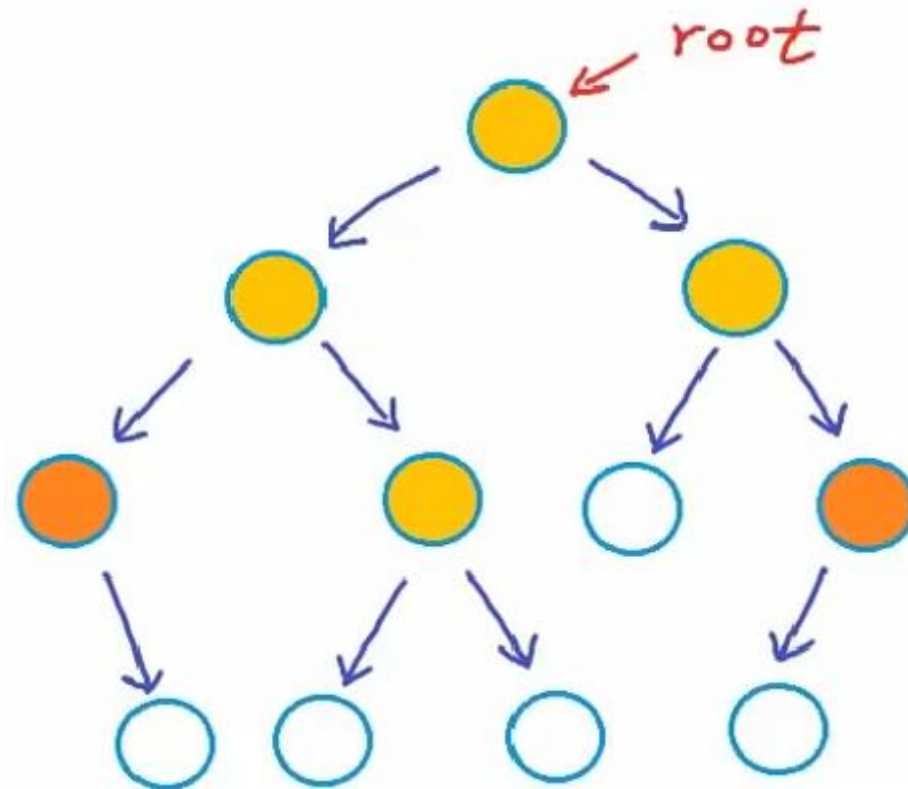
- Because each node can have at most Two children:
  - We call one of the children left child, and the other one right child.





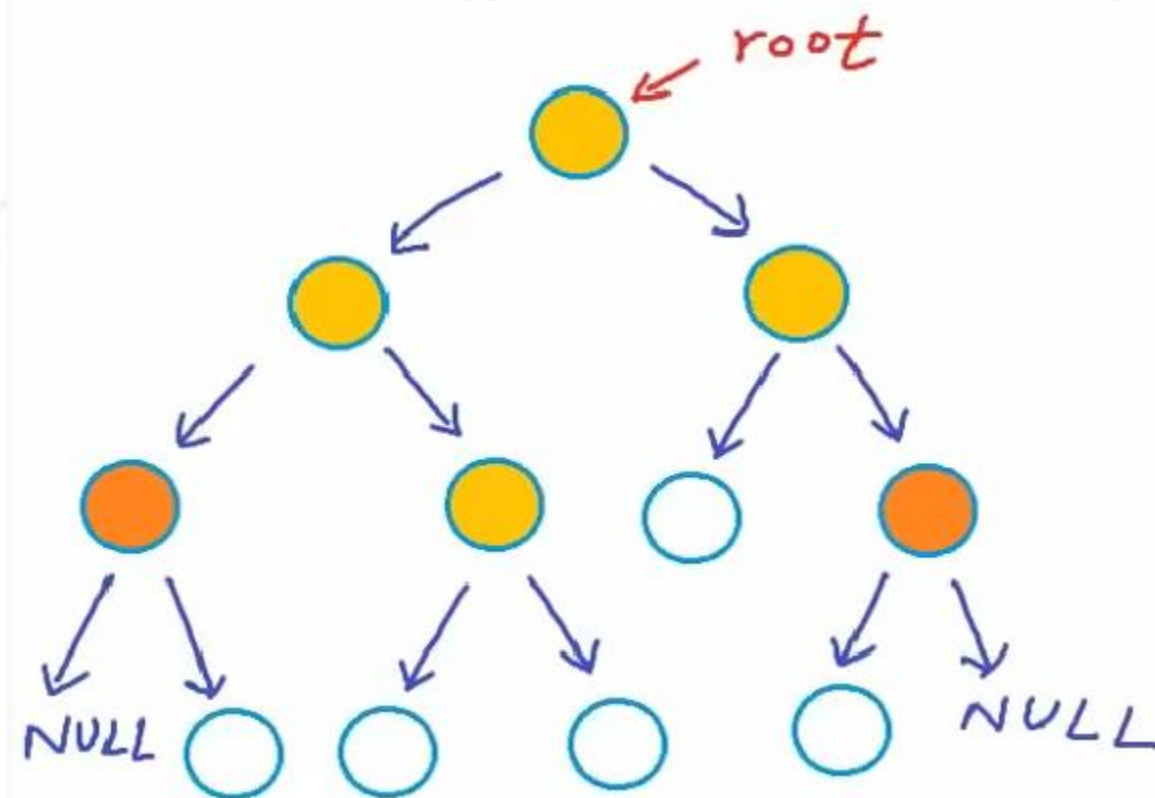
# Binary Tree

- A node may have both left and right child, or a node can have either left or right child.



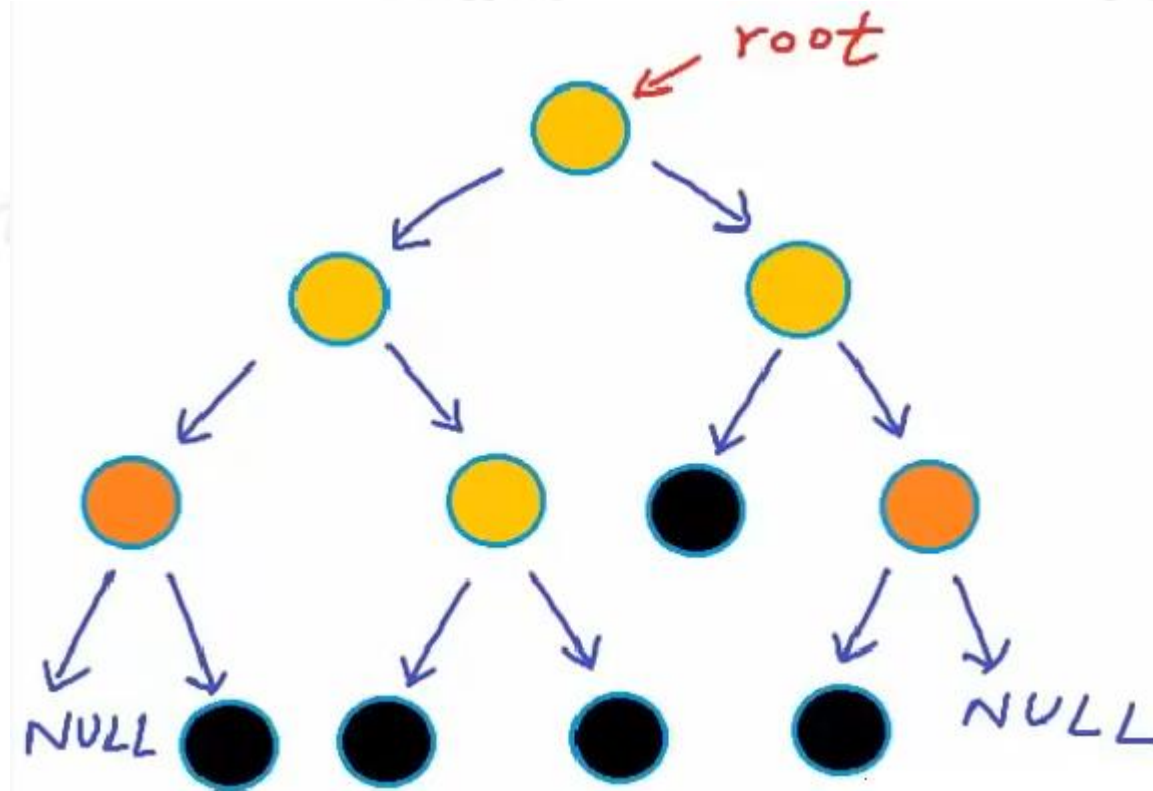
# Binary Tree

- In a program, we will set the reference to NULL if there is no child as shown below.



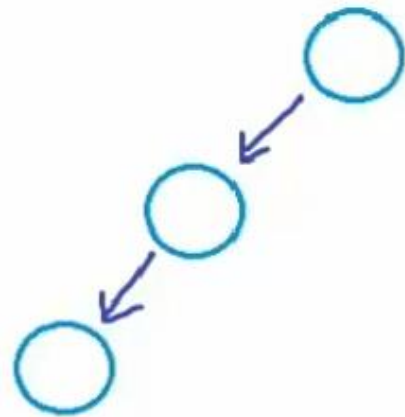
# Binary Tree

- In all leaf nodes, both left and right child are NULL.

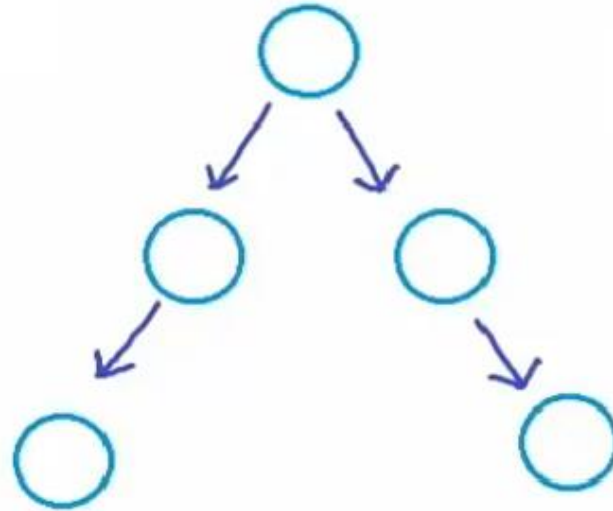


# Binary Tree

- The following three Trees are binary Trees:



(1)



(2)

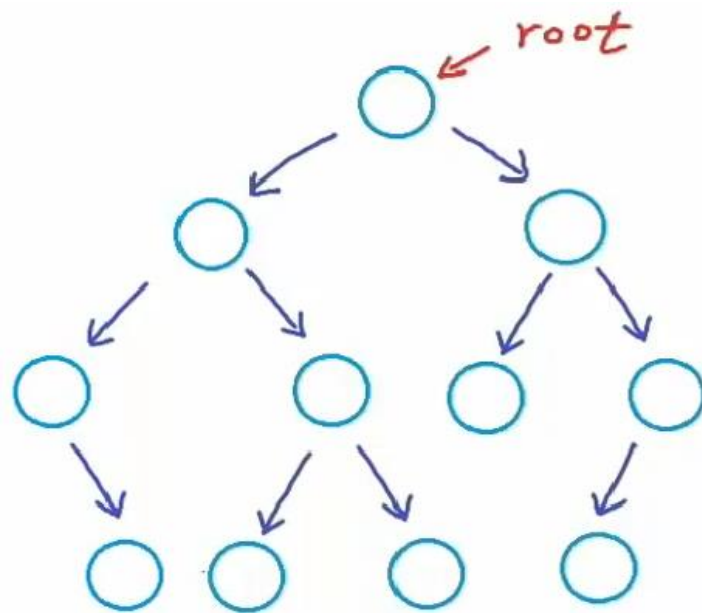


(3)

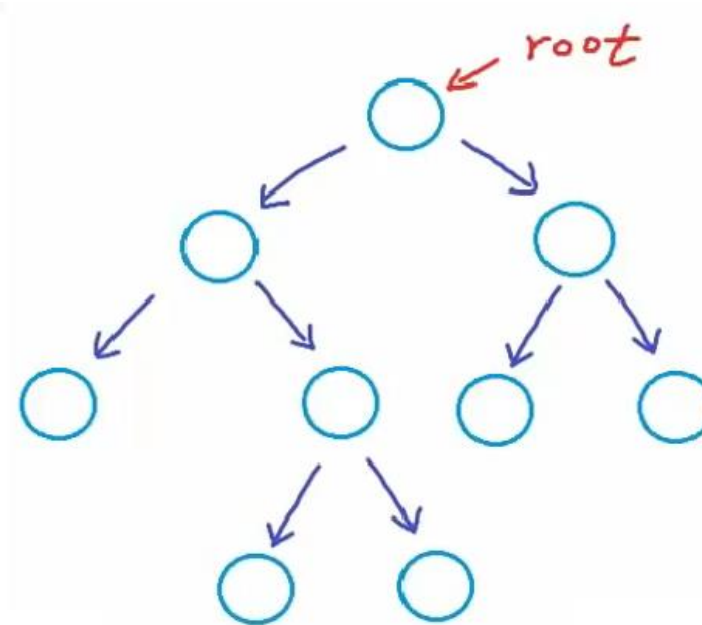
- Remember, the only condition is that a node cannot have more than Two children.

# Binary Tree

- Based on properties, we classify binary tree into different types.
- A binary tree can be called **Strict** or **Proper** binary tree if each node can have either Two or Zero children.



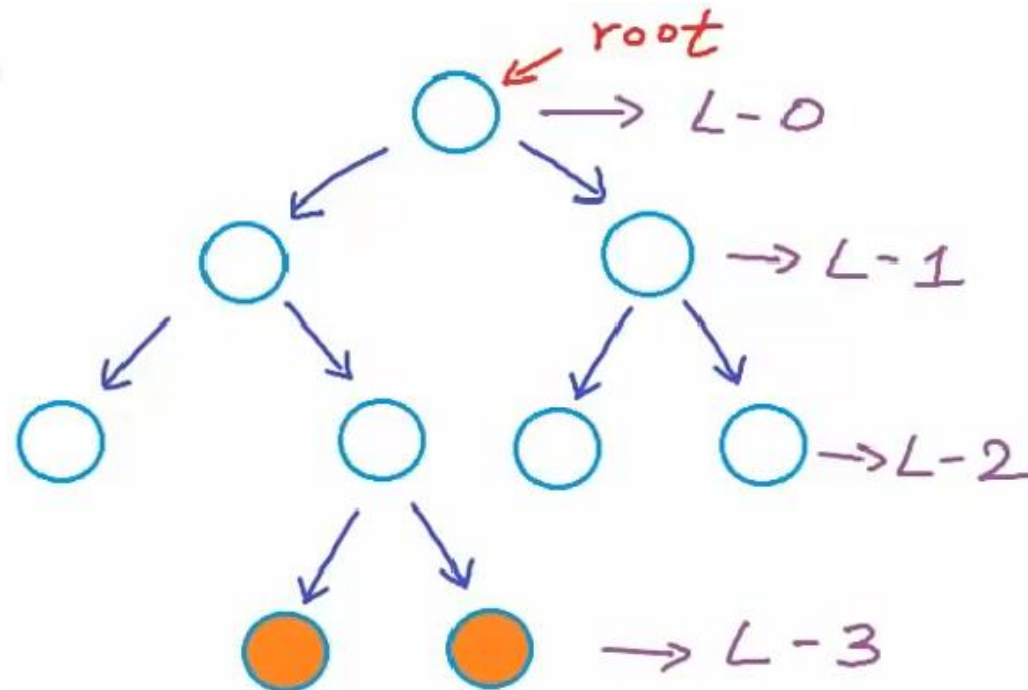
Not Strict Binary Tree



Strict Binary Tree

# Binary Tree

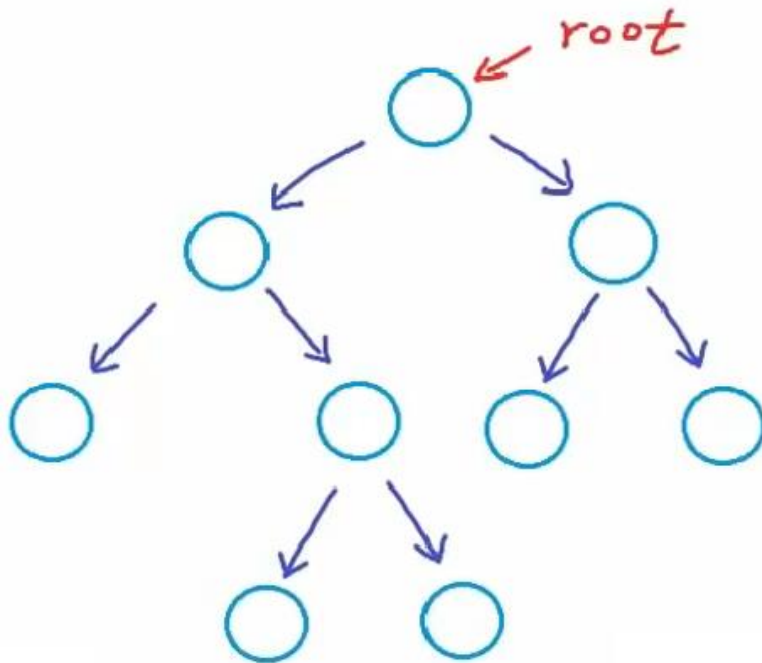
- Nodes at same depth can be called nodes at same level.
- Root node in a tree has depth 0.
- Depth of a node is defined as length of path from root to that node.



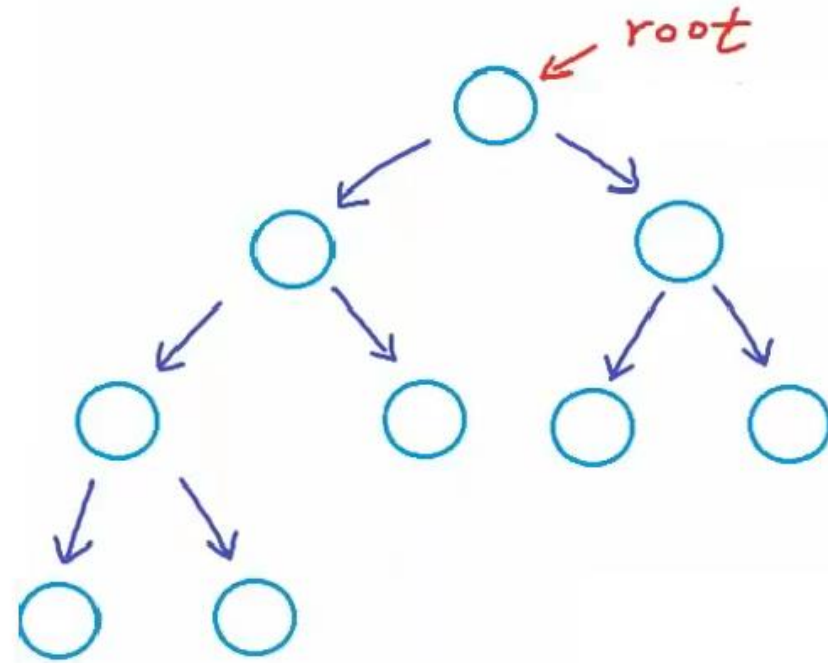


# Binary Tree

- A binary tree can be called **Complete** binary tree if all levels except the last level are completely filled, and all nodes are as left as possible.



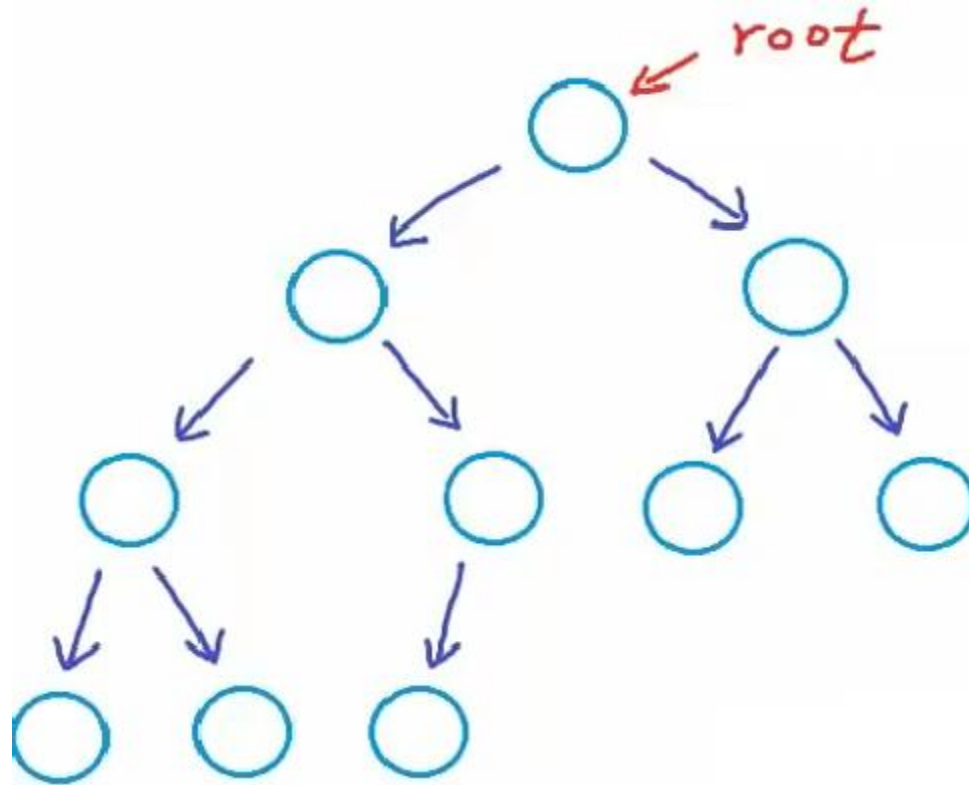
Not Complete Binary Tree



Complete Binary Tree

# Binary Tree

- This is also a complete binary tree:





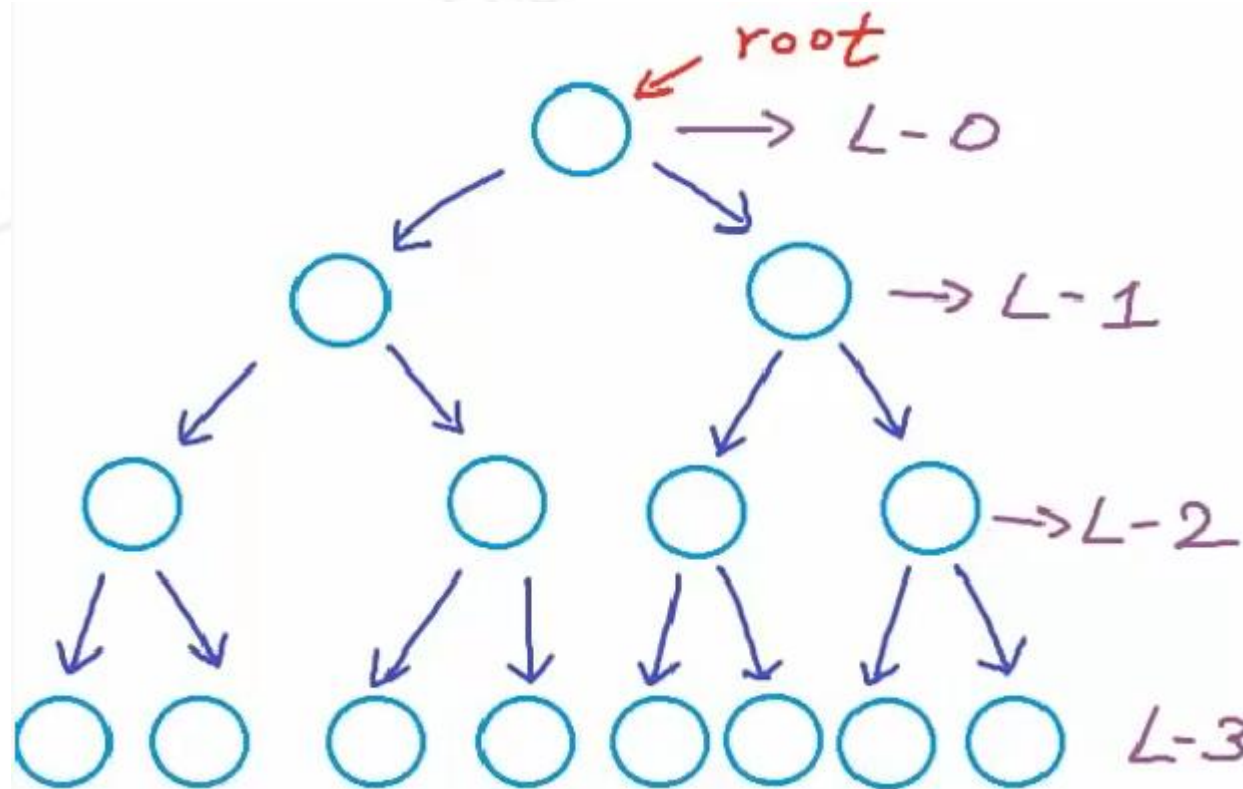
# Binary Tree

- Maximum depth of a tree is also equal to height of the tree.
- Maximum number of nodes at level (i):

Max no. of nodes at  
level  $i = 2^i$

# Binary Tree

- If all the levels are completely filled, such a binary tree can also be called a **perfect** binary tree.



# Binary Tree

- If **h** is the height of a binary tree, then **maximum** number of nodes is:

$$\begin{aligned} &\text{Maximum no. of nodes} \\ &\text{in a binary tree with height } h \\ &= 2^0 + 2^1 + \dots + 2^h \\ &= 2^{h+1} - 1 \\ &= 2^{(\text{no. of levels})} - 1 \end{aligned}$$

- Which equals to number of nodes in a perfect binary tree with the same height.

# Binary Tree

- So, what will be the height of a perfect binary tree with  $n$  nodes?

$$\begin{aligned}n &= 2^{h+1} - 1 \rightarrow n = \text{no. of nodes} \\ \Rightarrow 2^{h+1} &= (n+1) \\ \Rightarrow h &= \log_2(n+1) - 1\end{aligned}$$

- We can also calculate height as following:

$$\begin{aligned}\text{Height of complete binary tree} \\ &= \lfloor \log_2 n \rfloor\end{aligned}$$

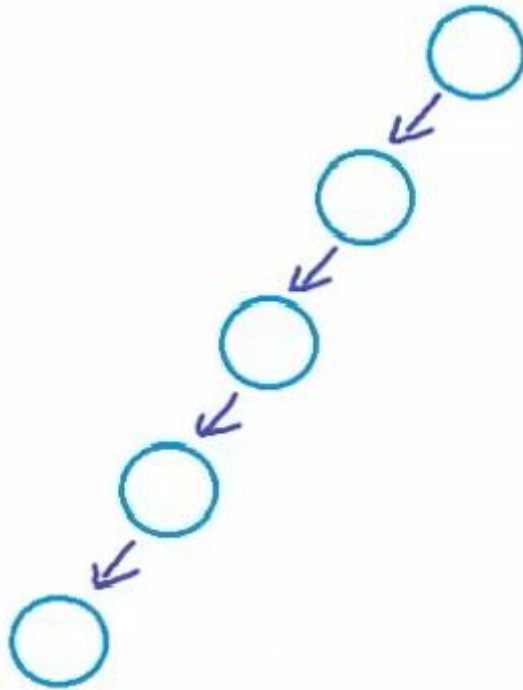
# Binary Tree

- Cost of a lot of operations on tree in terms of time depends on the height of tree.
- In such case, we want the height of the tree to be less.
- Height of a tree will be less if the tree will be **dense**, if the tree will be close to a perfect binary tree or a complete binary tree.
- Minimum height of a tree with  $n$  nodes, when the tree will be a complete binary tree.

$$\text{Height of complete binary tree} \\ = \lfloor \log_2 n \rfloor$$

# Binary Tree

- If we have a tree like the following, then the tree will have the maximum height:



# Binary Tree

► With  $n$  nodes:

► The minimum height possible is (**Dense tree**):

$$\text{Min-height} = \lfloor \log_2 n \rfloor$$

► The maximum height possible is (**Sparse tree**):

$$\text{Max-height} = n - 1$$



# Binary Tree

- We want to keep the height of a binary tree minimum possible.
- Commonly, we want to keep a binary tree **balanced**.

Balanced binary tree

↳ difference between  
height of left and right  
subtree for every node is  
not more than  $k$  (mostly 1)

# Binary Tree

- Height of a tree:

Height  $\rightarrow$  no. of edges in longest path from root to a leaf

- Height of a tree with one node:

Height of tree with 1 node = 0

- Height of an empty tree:

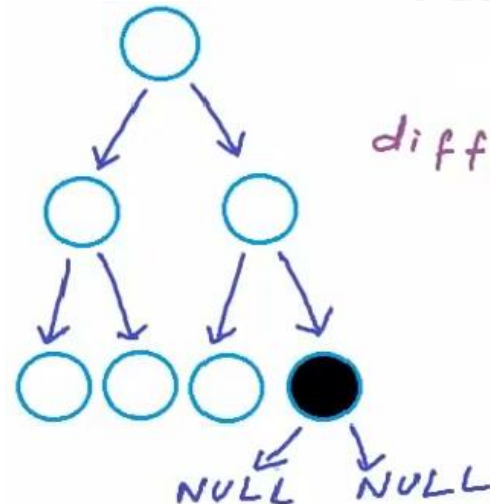
Height of an empty tree = -1

# Binary Tree

- The difference between heights of left and right subtrees of a node:

$$\text{diff} = |h_{\text{left}} - h_{\text{right}}|$$

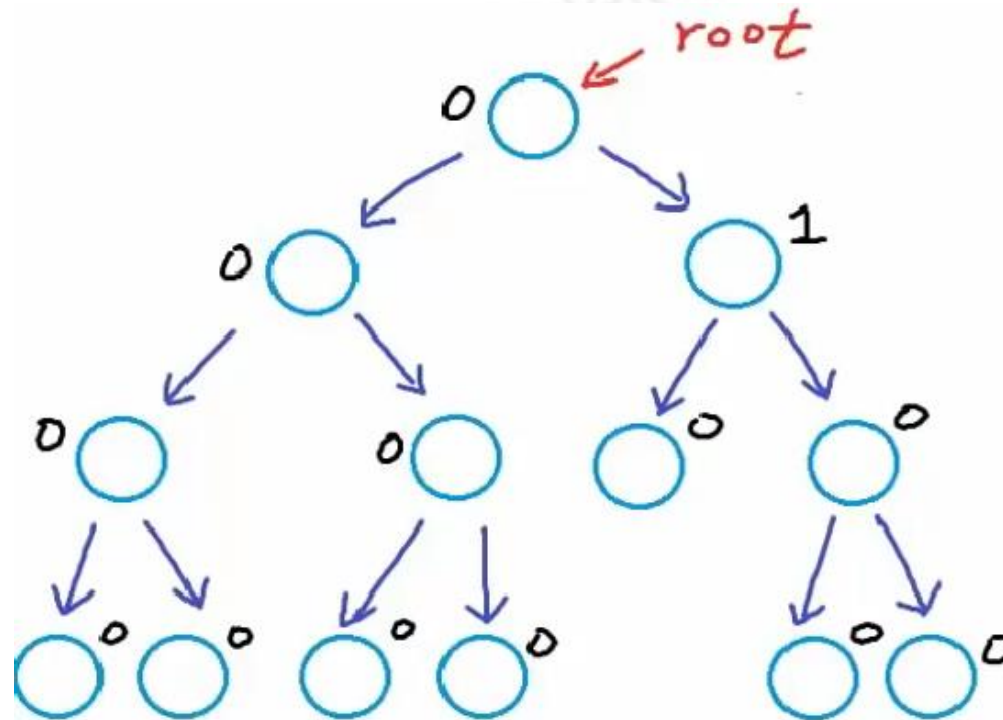
- Example: in the following leaf node both left and right subtrees are empty.



$$\begin{aligned} \text{diff} &= |h_{\text{left}} - h_{\text{right}}| \\ &= | \downarrow -1 - \downarrow (-1) | = 0 \end{aligned}$$

# Example 1

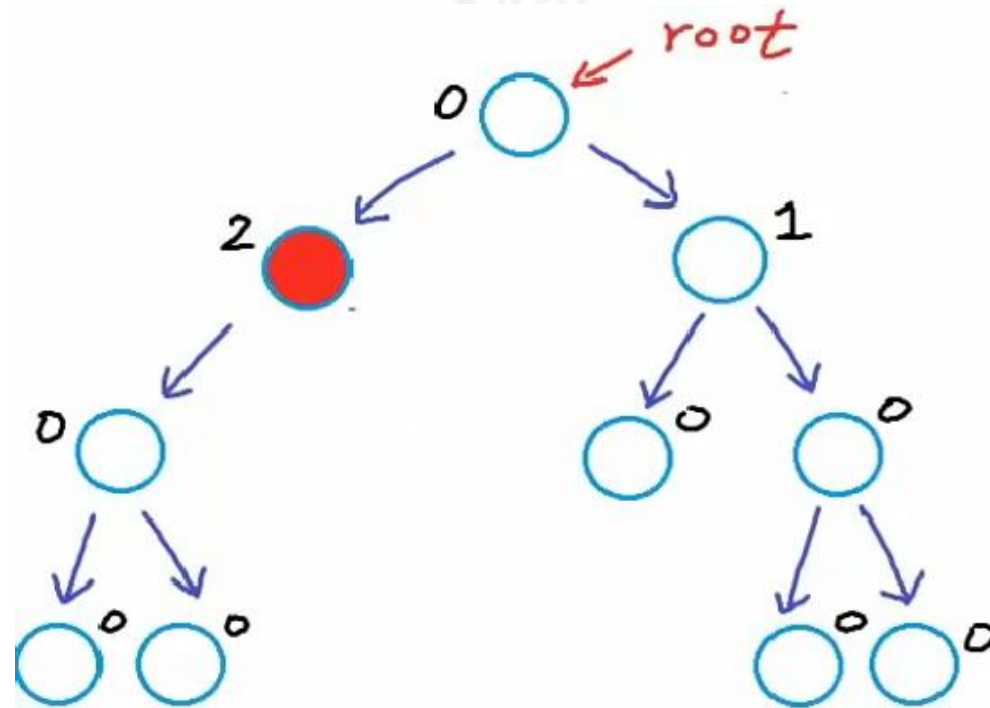
- The value of difference between heights of left and right subtrees of each node:



- Balanced binary tree because the maximum difference for any node is 1.

## Example 2

- The value of difference between heights of left and right subtrees of each node:



- Not balanced because one of the nodes has difference 2.

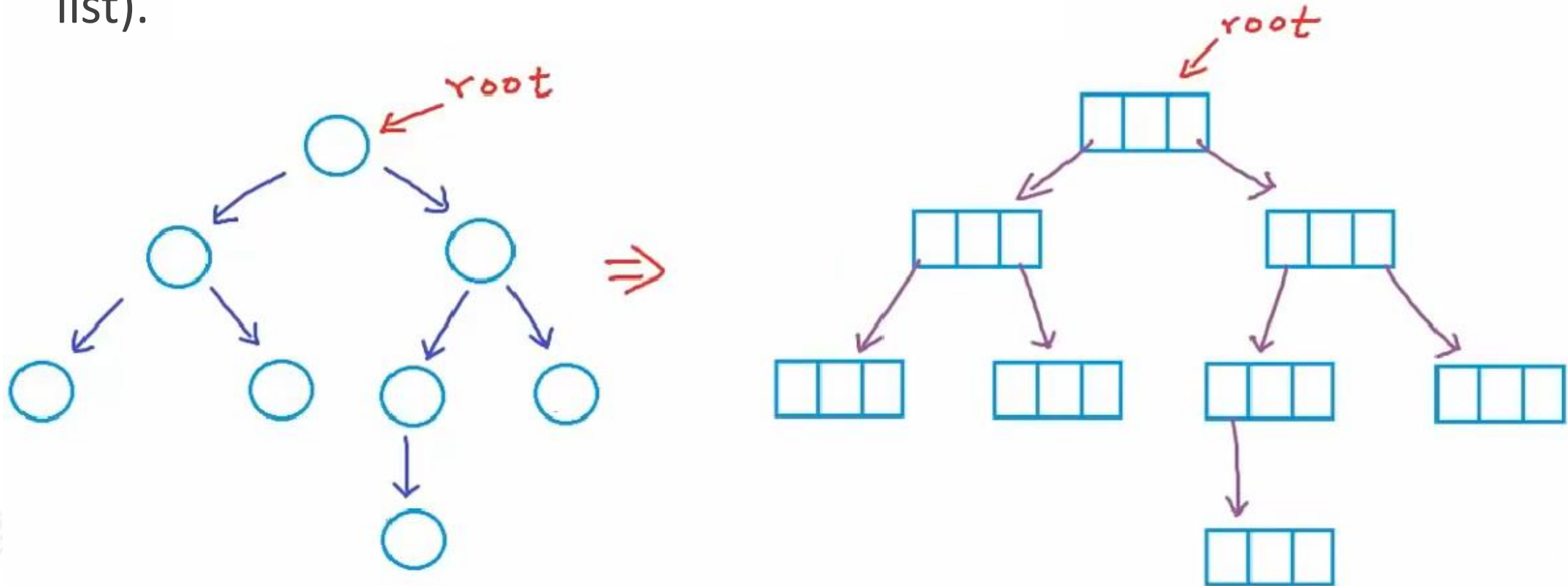
# Binary Tree Implementation

- We can implement binary tree using:
  - Dynamically created nodes.
  - Arrays (in some special cases).

# Binary Tree Implementation

## Dynamically created nodes

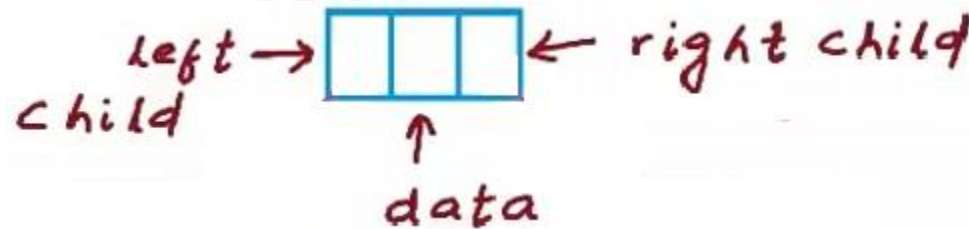
- The most common way of implementing tree.
- Dynamically created nodes linked using references (Same as linked list).





# Binary Tree Implementation

- Each Node has Three fields.
  - Middle field is to store Data.
  - Left field is to store a reference to the left child.
  - Right field is to store a reference to the right child.



# Binary Tree Implementation

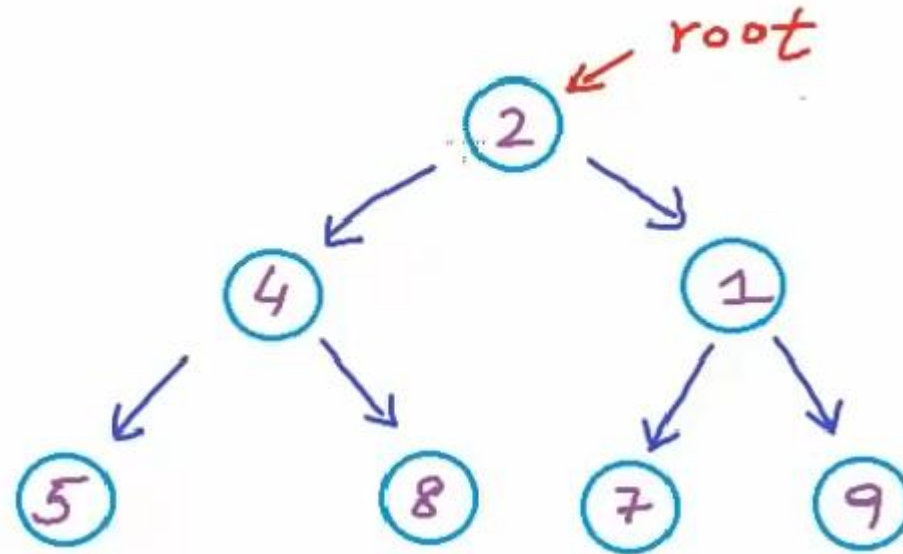
## Dynamically created nodes

- The created nodes are linked to each other using references.
- For a binary tree of integers, we can define a node like this:

```
class Node{  
    int data;  
    Node left;  
    Node right;  
}
```

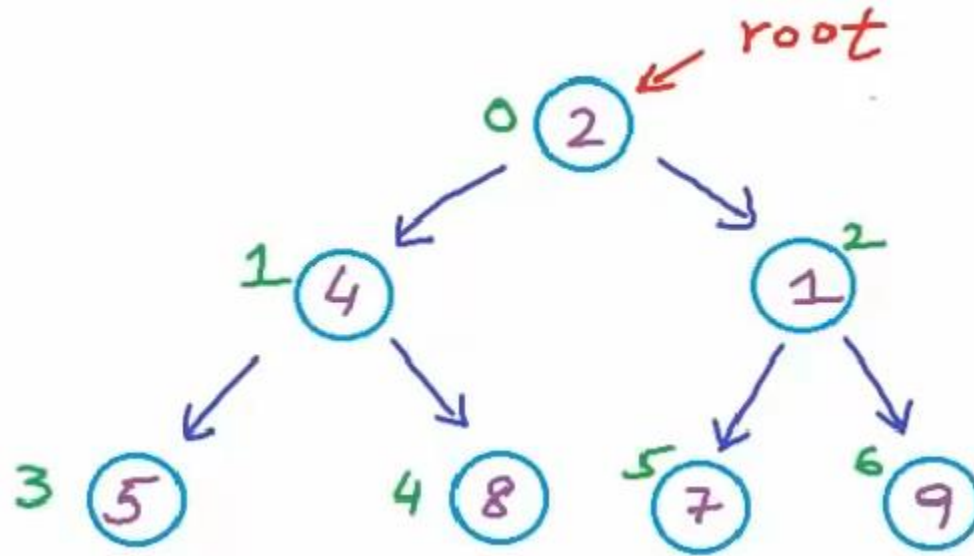
# Binary Tree Implementation Arrays

- Arrays are typically used for complete binary trees.
- The following is a complete binary tree:



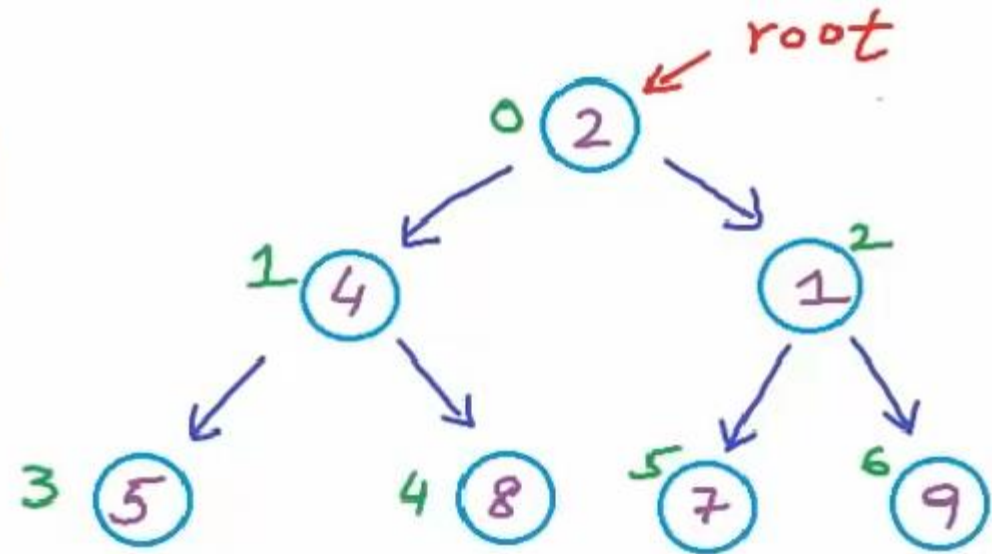
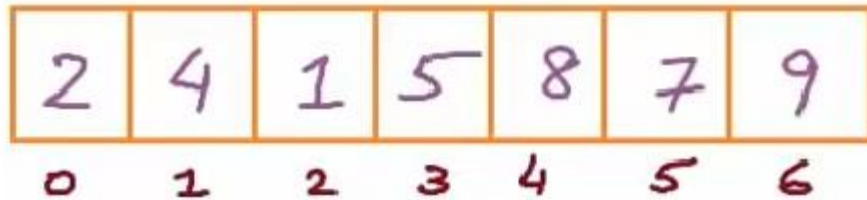
# Binary Tree Implementation Arrays

- We can number these nodes from 0 starting at root, going level by level from left to right:



# Binary Tree Implementation Arrays

- Now we can create an array of Seven integers.
- The numbers can be used as indices for these nodes.



# Binary Tree Implementation Arrays

- How will we store the information about the links?

$$\begin{aligned} \text{for node at index } i, \\ \text{left-child-index} &= 2i+1 \\ \text{right-child-index} &= 2i+2 \\ \text{Parent-index} &= \left\lfloor \frac{i-1}{2} \right\rfloor \end{aligned}$$

- Remember, this is true only for a complete binary tree.

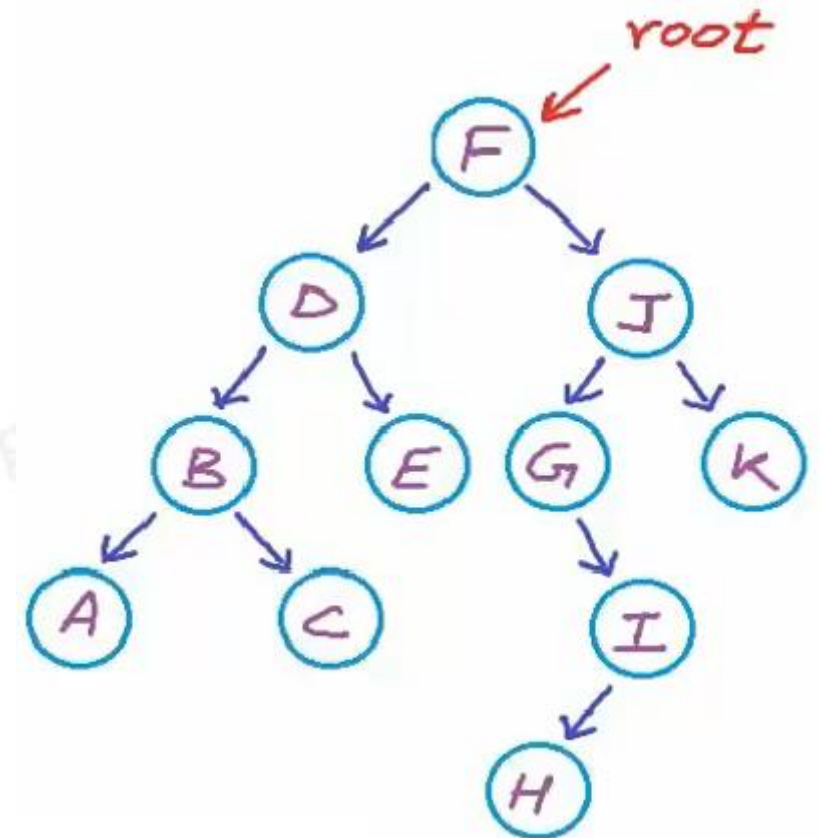
# Binary Tree Traversal

- When we are working with binary trees, we may often want to visit all nodes in the tree.
- Tree is not a linear data structure like array or linked list.
- In a linear data structure:
  - There would be a logical start and a logical end.
  - For each element, we would have only one next element.



# Binary Tree Traversal

- We have the following binary tree of characters.
- Tree traversal is not straight forward.
- For a tree, at anytime if we are pointing to a particular node then we can have more than one possible next node.
- We will learn the algorithms for tree traversal.



# Binary Tree Traversal

- Tree traversal can formally be defined as:

*Tree Traversal*

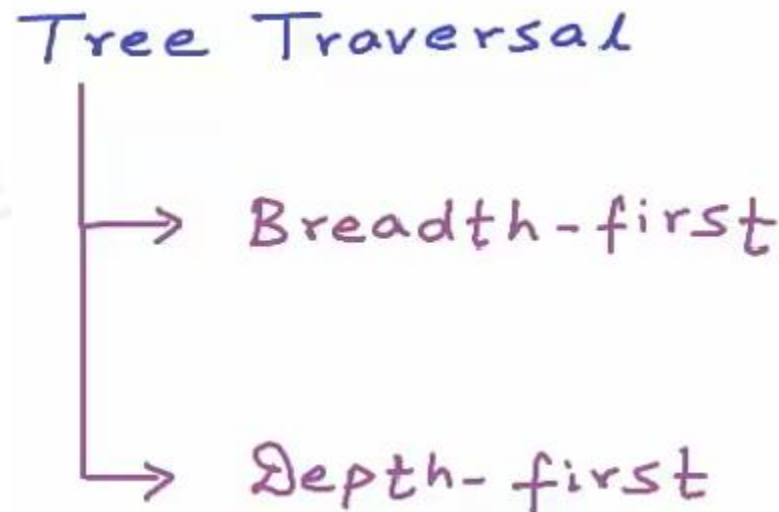
↳ process of visiting  
each node in the tree  
exactly once in some order.

- By visiting a node, we mean:

*Visit → Reading / Processing  
data in a node*

# Binary Tree Traversal

- Based on the order in which nodes are visited, tree traversal algorithms can be classified into Two categories.
- We can either go **Breadth-First**, or we can go **Depth-First**.



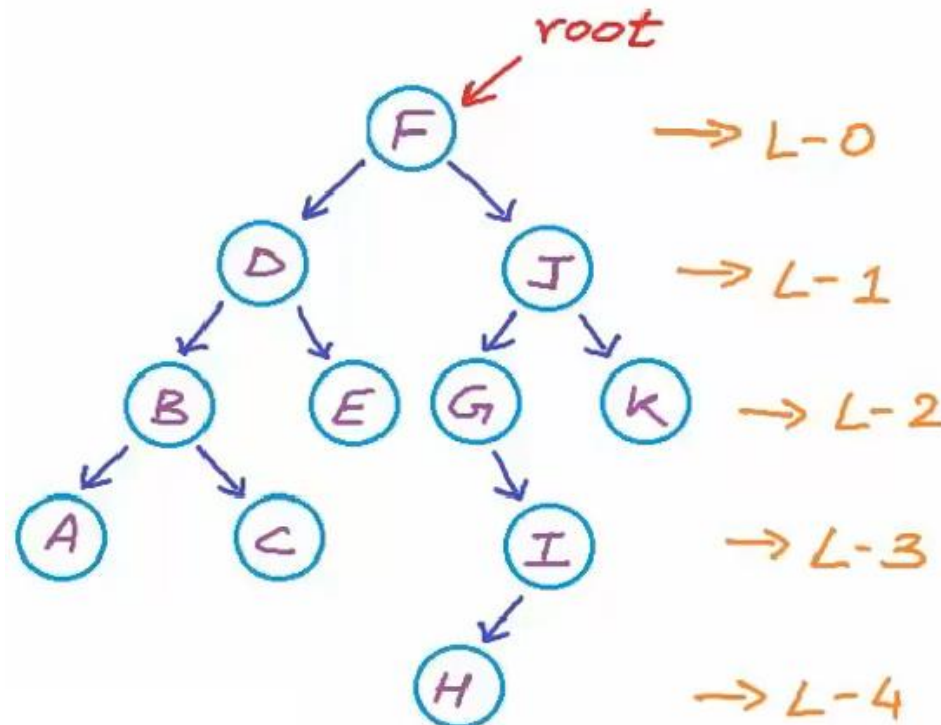
# Binary Tree Traversal

- Breadth-First traversal and Depth-First traversal are general techniques to traverse a graph.
- A graph is a data structure, we will learn it in the next session.
- For now, just know that tree is a special kind of graph.

# Binary Tree Traversal

## Breadth-First Approach

- In a tree, in breadth-first approach we would visit all the nodes at same depth or level before visiting the nodes at next level.
- Referring to the previous binary tree of characters, the levels of the nodes are:



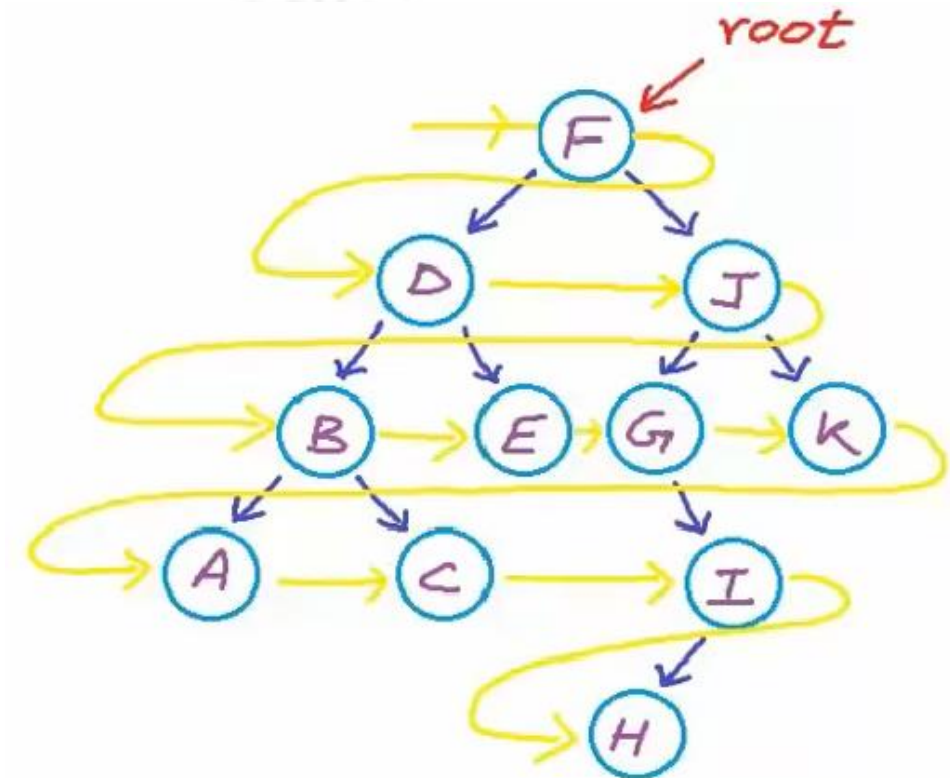
# Binary Tree Traversal

## Breadth-First Approach

- In breadth-first approach, we will start at level 0, then the next level visiting the nodes from left to right.

*F, D, J, B, E, G, K, A, C, I, H*

- This kind of breadth-first traversal in case of trees is called **Level Order Traversal**.

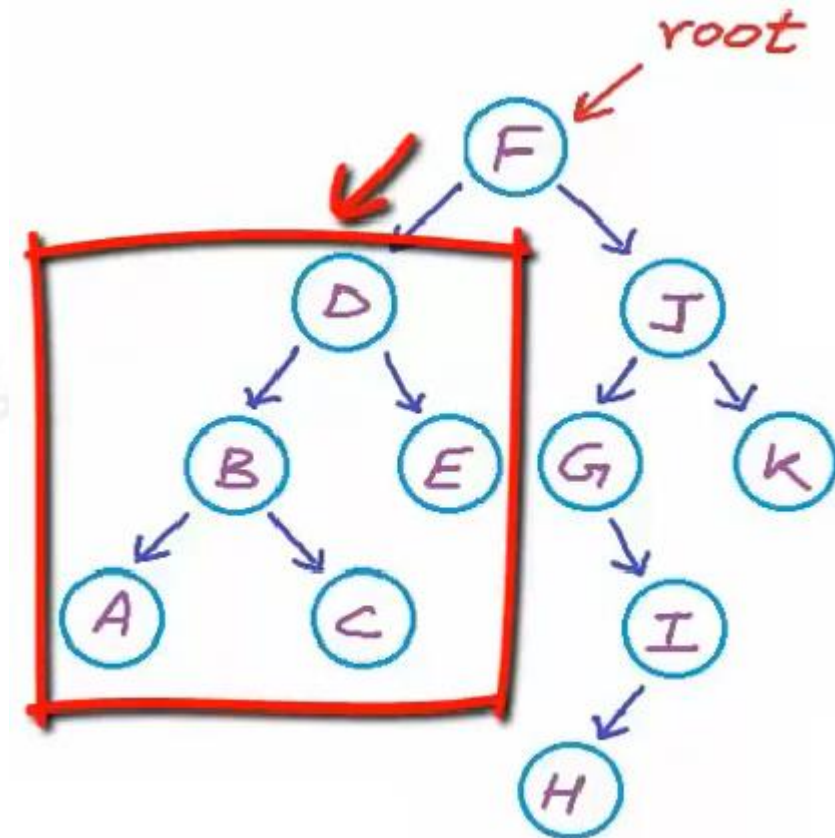




# Binary Tree Traversal

## Depth-First Approach

- In breadth-first approach, for any node we visit all its children before visiting any of its grandchildren.
- In Depth-first approach, if we would go to a child, we should complete the whole subtree of the child before going to the next child.
- In this tree, from the root (F) if we are going left (to D). then we should visit all the nodes in the left subtree before going to the right child (to J).

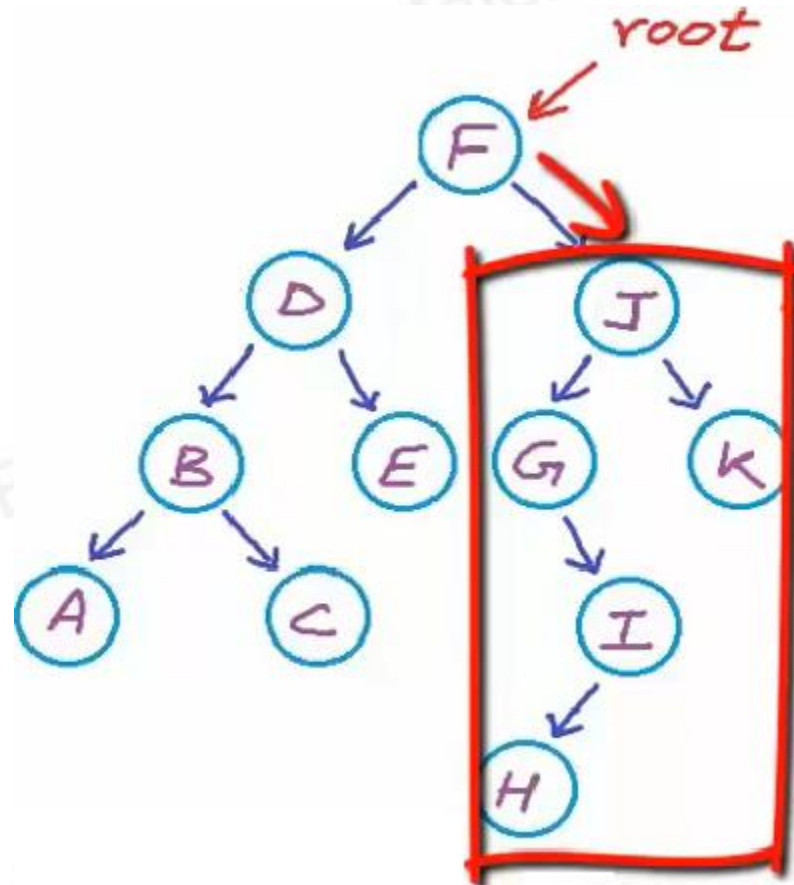




# Binary Tree Traversal

## Depth-First Approach

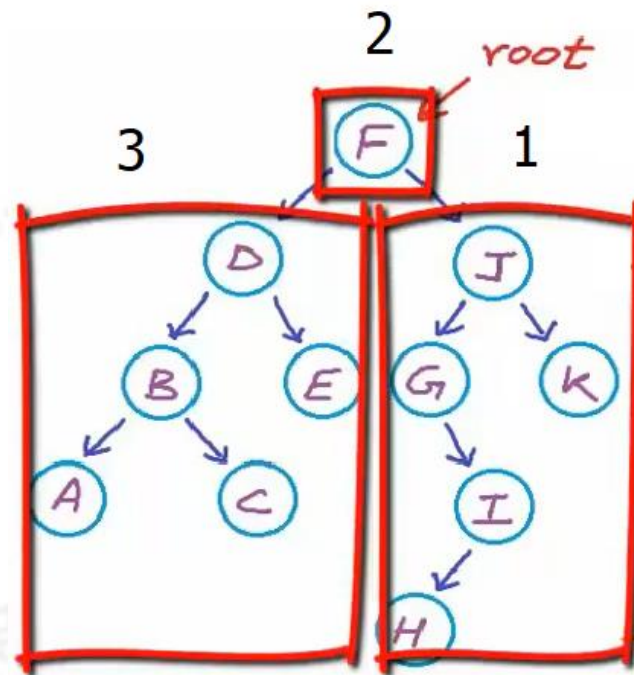
- Once again, when we will go to (J), we will visit the complete right subtree.



# Binary Tree Traversal

## Depth-First Approach

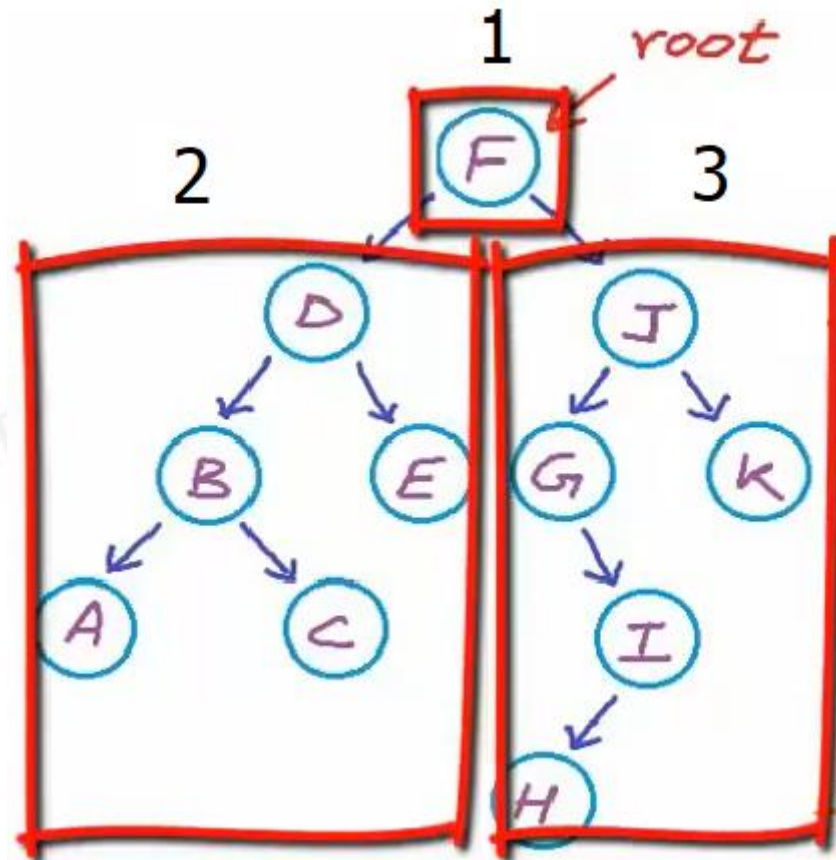
- In depth-first approach, the relative order of visiting the left subtree, the right subtree, and the root node can be different.
- For example, we can visit the right subtree, then the root and then the left subtree.



# Binary Tree Traversal

## Depth-First Approach

- Or we can first visit the root, then the left subtree and then the right subtree.



# Binary Tree Traversal

## Depth-First Approach

- So, the relative order can be different but the core idea in depth-first strategy is that visiting a child is visiting the complete subtree in that path.
- Based on the relative order of depth-first approach, there are Three popular strategies:

$\langle \text{root} \rangle \langle \text{left} \rangle \langle \text{right} \rangle$  - Preorder  
 $\langle \text{left} \rangle \langle \text{root} \rangle \langle \text{right} \rangle$  - Inorder  
 $\langle \text{left} \rangle \langle \text{right} \rangle \langle \text{root} \rangle$  - Postorder

# Binary Tree Traversal

## Depth-First Approach

- There is an easy way to remember the Three depth-first strategies.
- We can denote visiting a node with letter (D), going too left subtree as letter (L) and going to right subtree as letter (R).

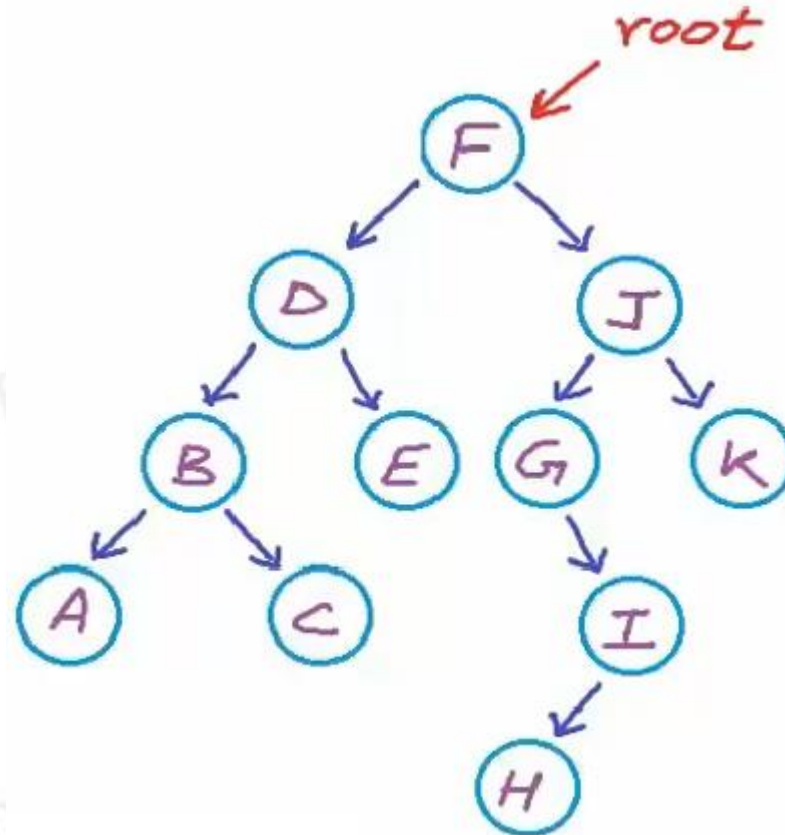
$\overset{D}{\langle \text{root} \rangle} \overset{L}{\langle \text{left} \rangle} \overset{R}{\langle \text{right} \rangle}$  - Preorder  
 $\overset{L}{\langle \text{left} \rangle} \overset{D}{\langle \text{root} \rangle} \overset{R}{\langle \text{right} \rangle}$  - Inorder  
 $\overset{L}{\langle \text{left} \rangle} \overset{R}{\langle \text{right} \rangle} \overset{D}{\langle \text{root} \rangle}$  - Postorder



# Binary Tree Traversal

## Depth-First Approach

- Let us now see what will be the pre-order, in-order and post-order traversal for the following tree.



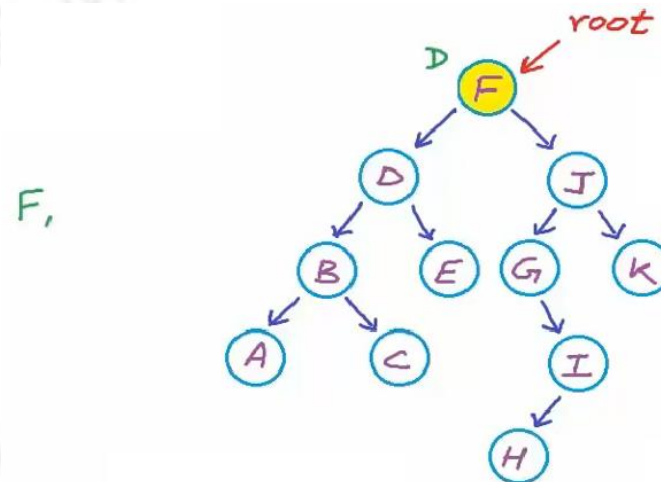
# Binary Tree Traversal

## Pre-Order Traversal

- We need to start at root node.

Preorder (DLR)  
 data ← left ← right

- For each node, we first need to visit the data.

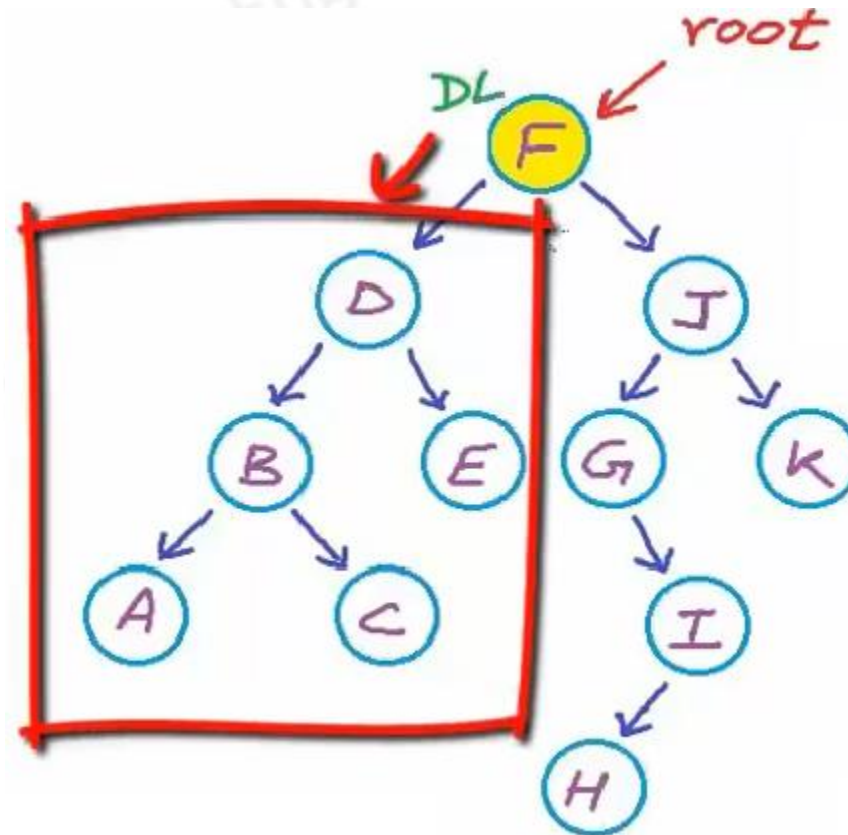




# Binary Tree Traversal

## Pre-Order Traversal

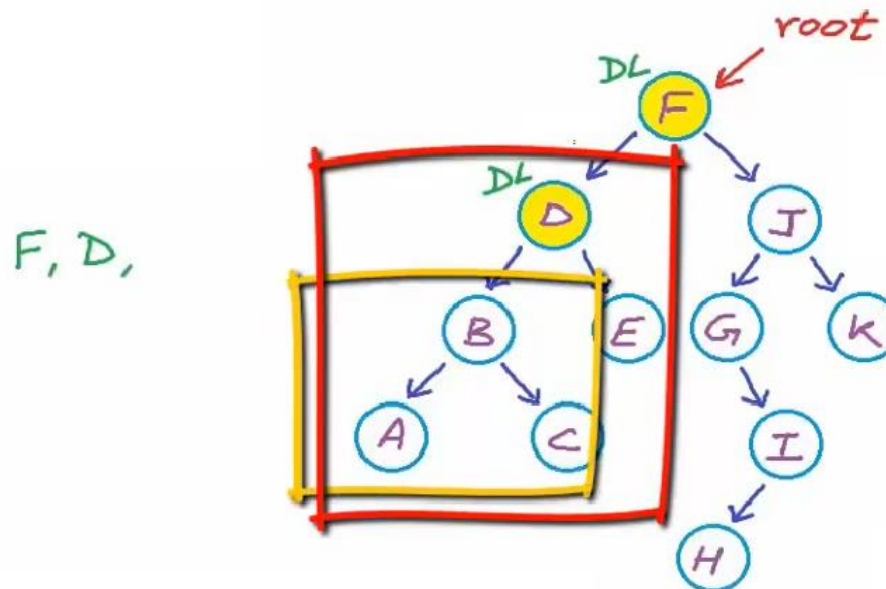
- Then we need to go left and finish the complete left subtree.
- Once all the nodes in left subtree are visited, only then I can go to the right subtree.



# Binary Tree Traversal

## Pre-Order Traversal

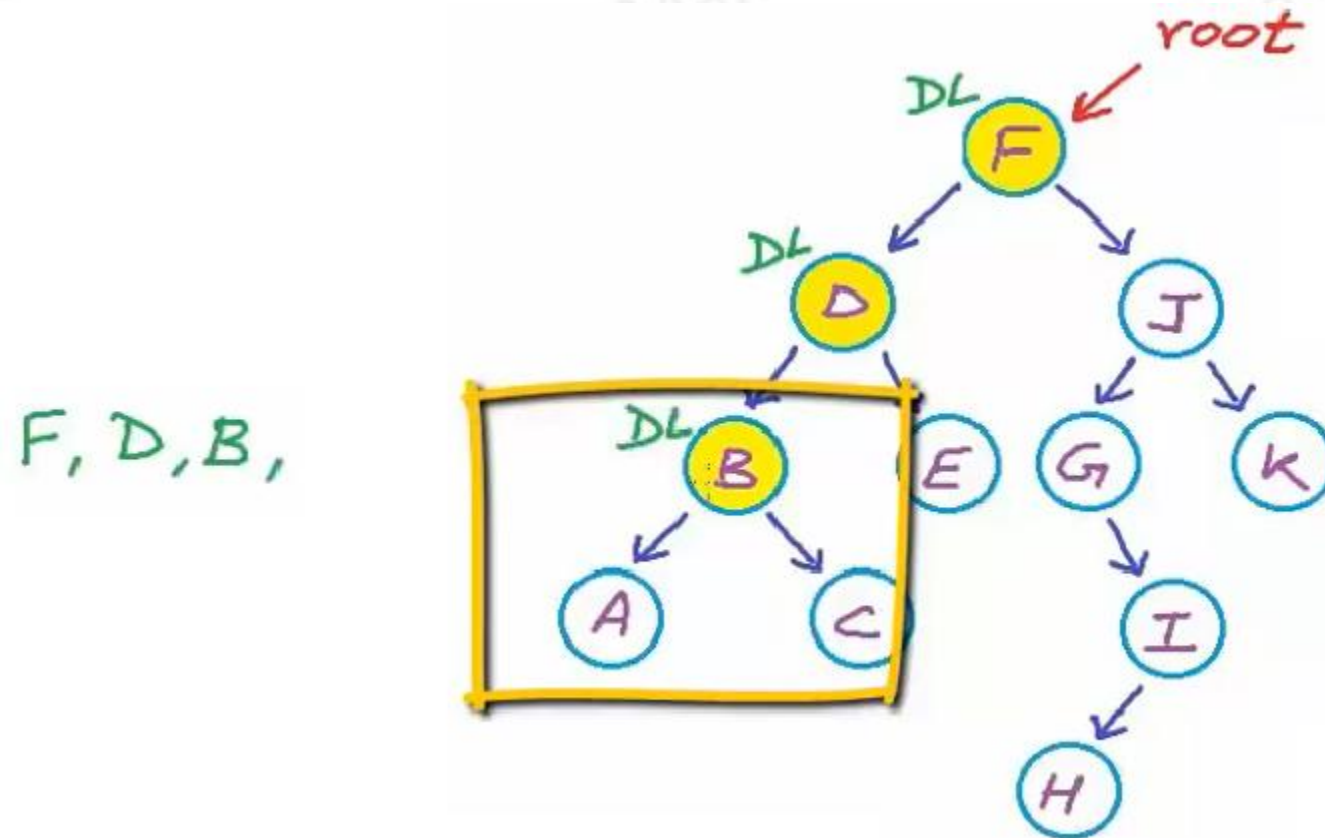
- The problem here is actually getting reduced in a self-similar (recursive) manner.
- Now we need to focus on the left subtree.
- We are at node D now, we need to visit the data, then we can go left.



# Binary Tree Traversal

## Pre-Order Traversal

- We are at node (B), we can visit the data and then go left.

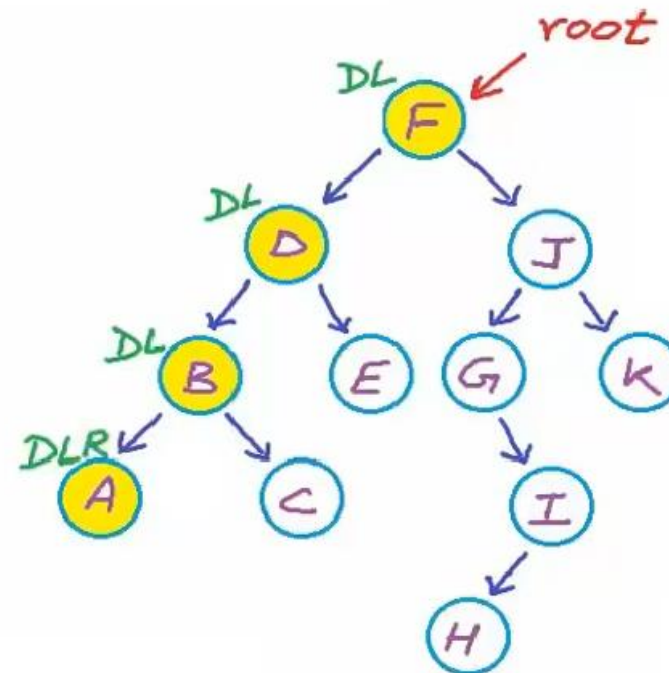


# Binary Tree Traversal

## Pre-Order Traversal

- We are at node (A), there is nothing in the left.
- We can say for node (A), left subtree is done.
- Now we can go right, but there is nothing at right as well, right is also done.

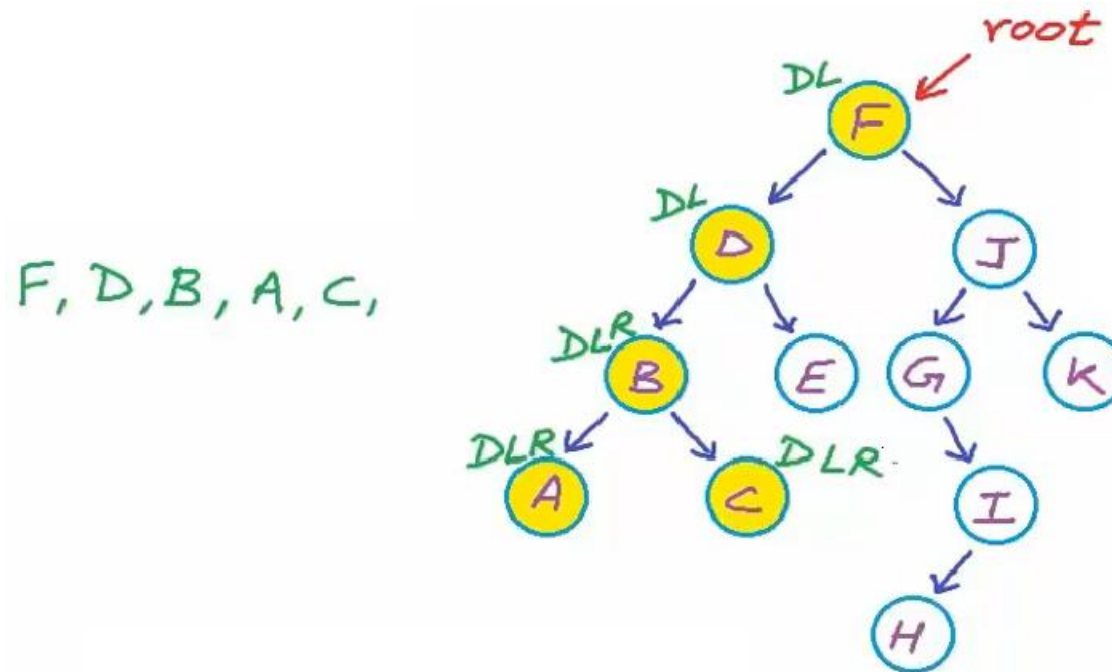
F, D, B, A,



# Binary Tree Traversal

## Pre-Order Traversal

- Now for node (B), left subtree is done so we can go right to node (C).
- We visit the data, then we can go left.
- Left and right of node (C) are null.

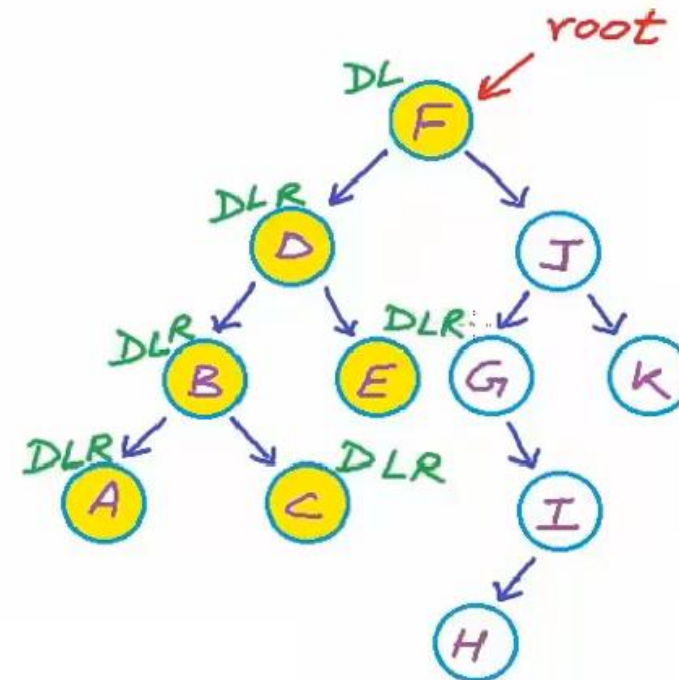


# Binary Tree Traversal

## Pre-Order Traversal

- Now for node (D), left subtree is done so we can go right to node (E).
- We visit the data, then we can go left.
- Left and right of node (E) are null.

F, D, B, A, C, E,

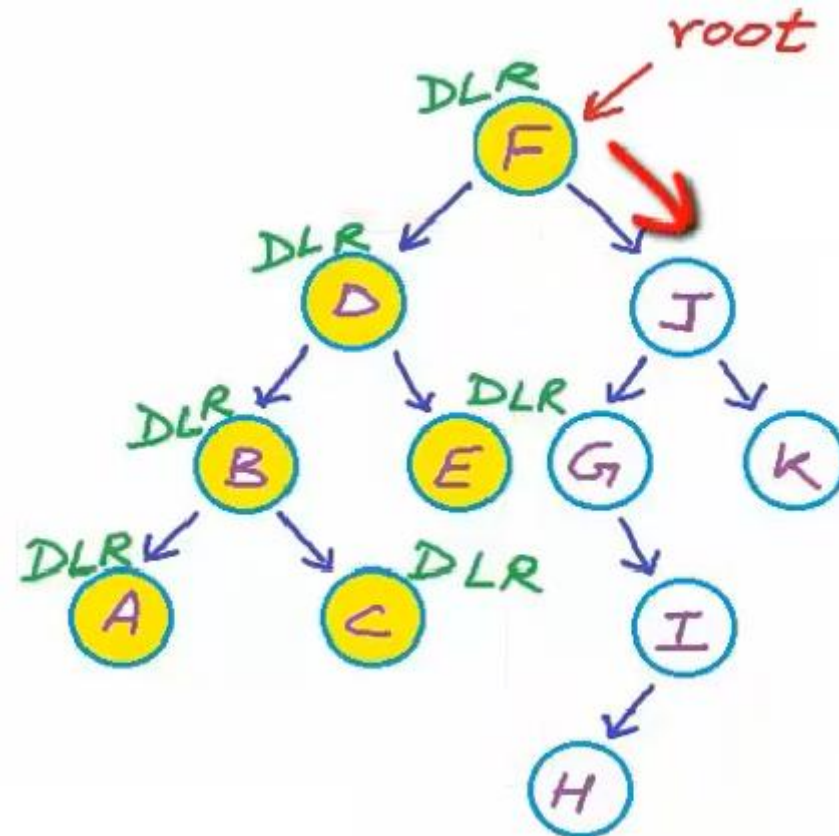




# Binary Tree Traversal

## Pre-Order Traversal

- At this stage, for node (F) the left subtree is visited so we can go right.



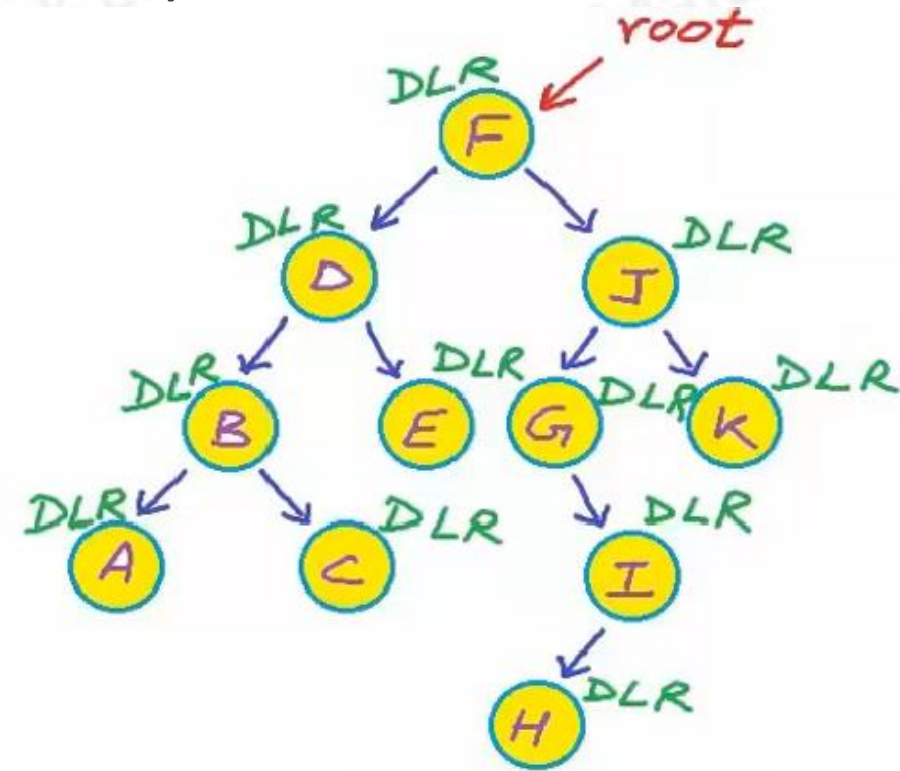


# Binary Tree Traversal

## Pre-Order Traversal

- Following the same strategy, the final output will be:

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



- This is how we can perform pre-order traversal manually.

# Binary Tree Traversal

## In-Order Traversal

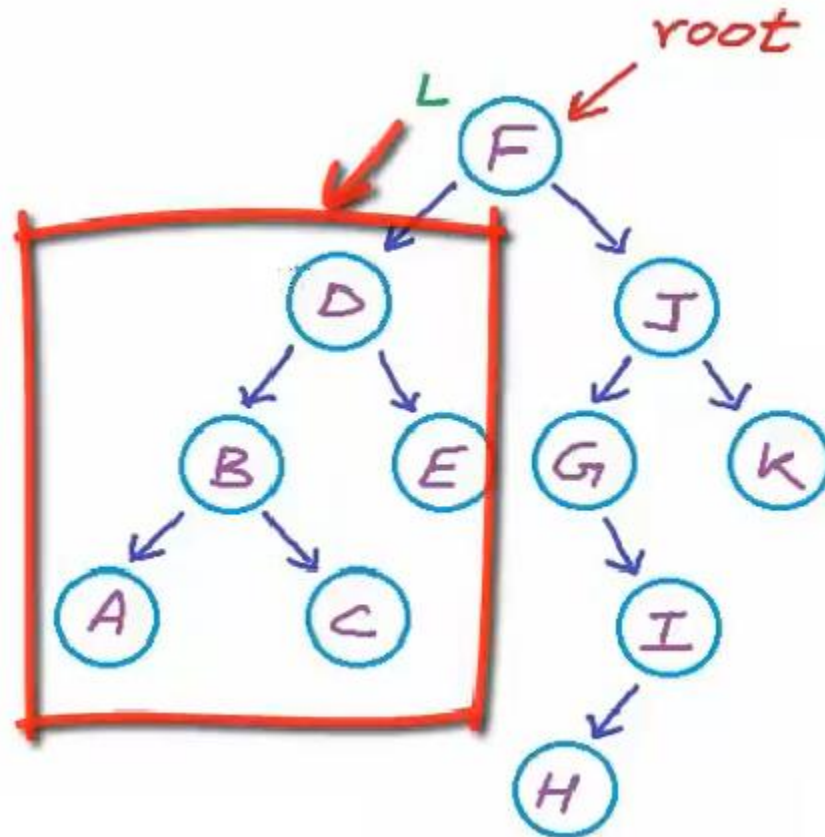
- In in-order traversal, we will first finish visiting the left subtree then visit the current node and then visit the right subtree.

*Inorder(LDR)*  
*left ↓ data ↓ right*

# Binary Tree Traversal

## In-Order Traversal

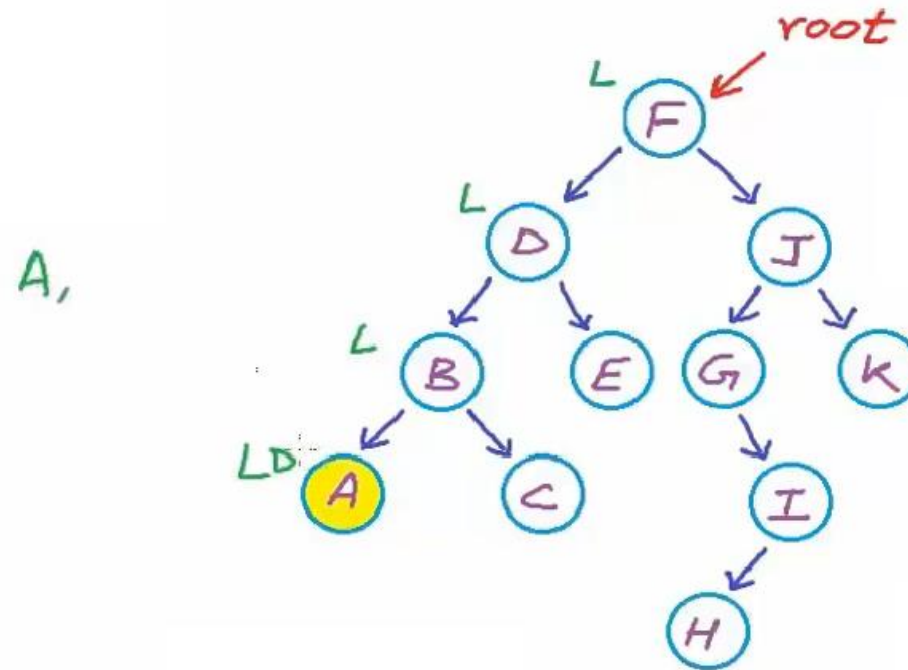
- Starting from the root node (F).
- We will first finish visiting the left subtree.



# Binary Tree Traversal

## In-Order Traversal

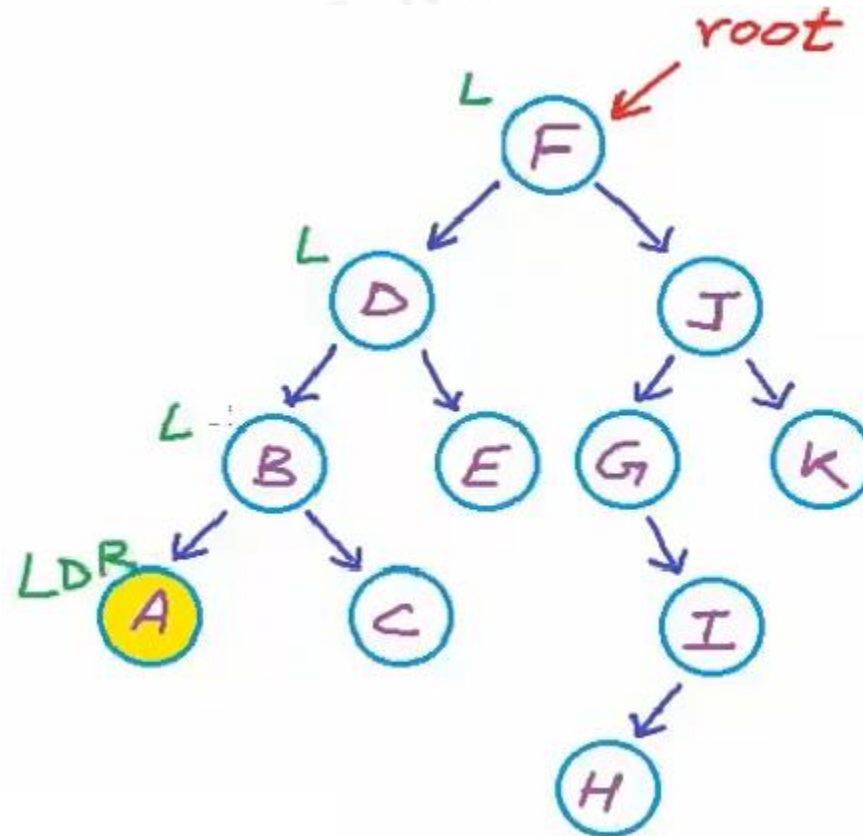
- Again, for node (D) we will first go left to (B).
- From (B) to (A).
- Now, for (A) there is nothing in left, we can say left is done so we can visit the data.



# Binary Tree Traversal

## In-Order Traversal

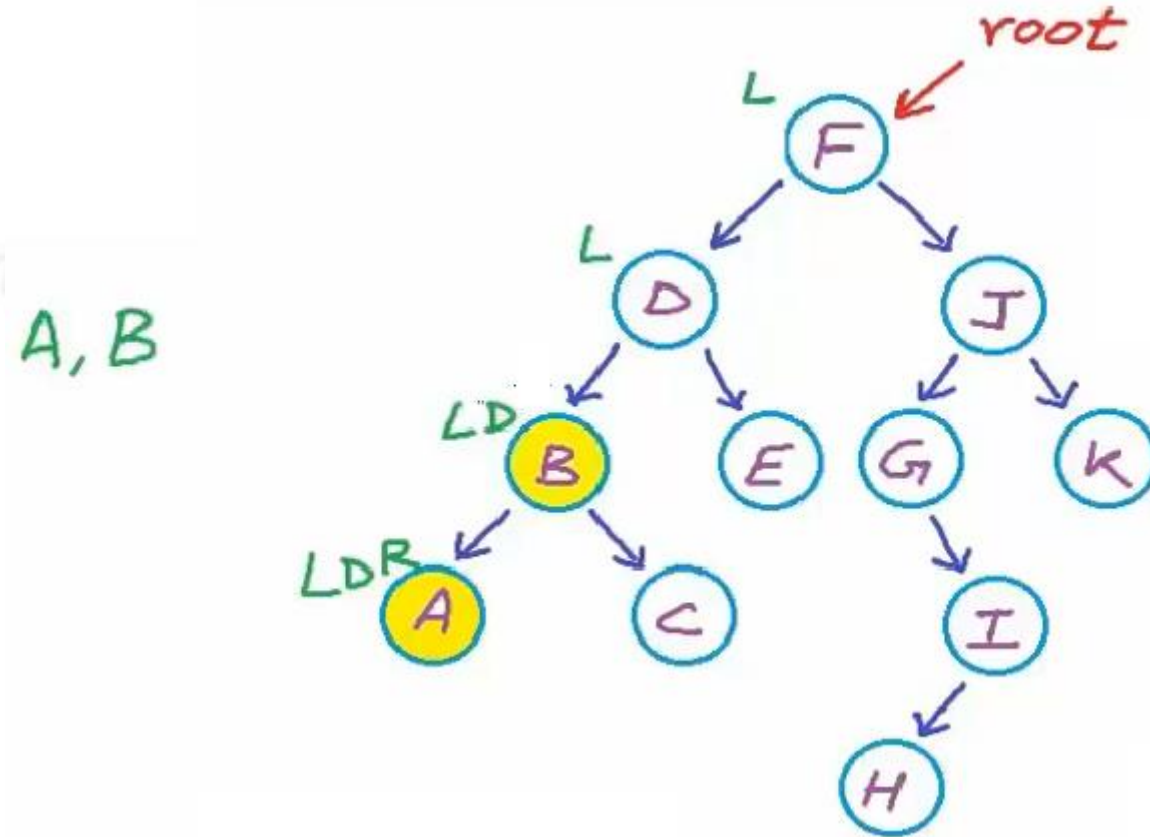
- Now we can go node's (A) right but there is nothing to the right, we can say right is done.



# Binary Tree Traversal

## In-Order Traversal

- For (B), left subtree is done so we can visit the data.

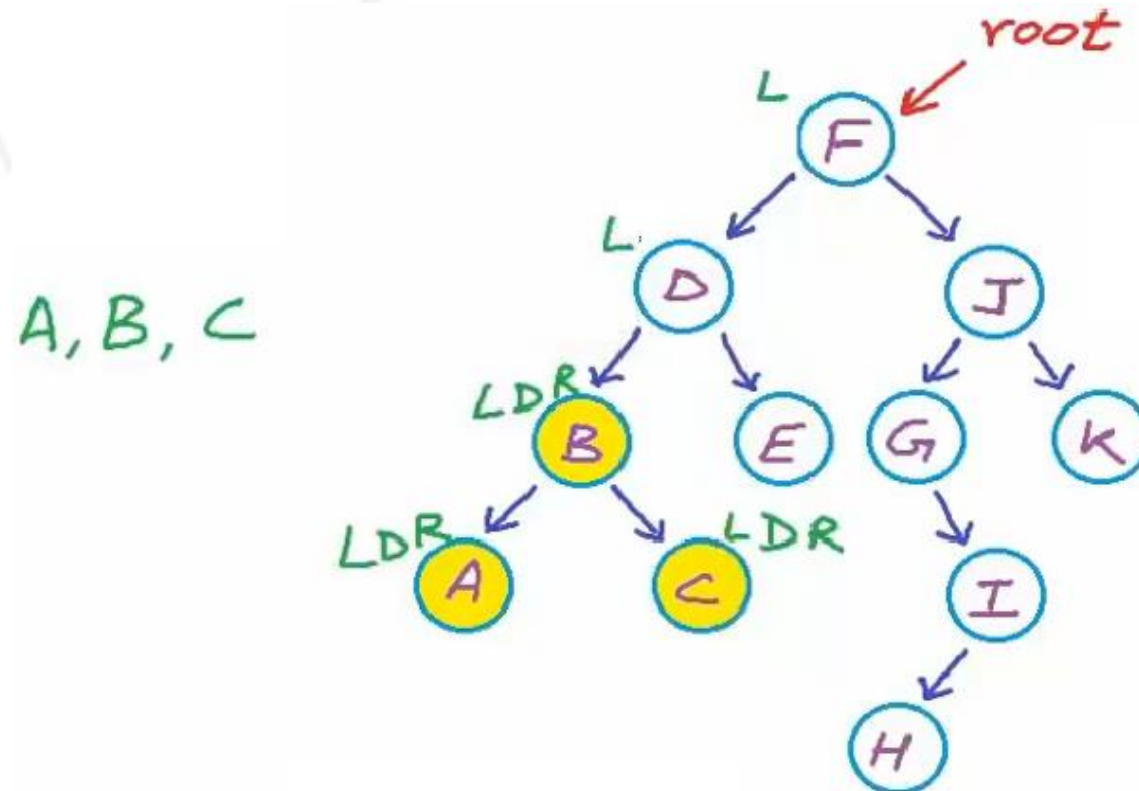




# Binary Tree Traversal

## In-Order Traversal

- For (B), we can go right to (C).
- There is nothing in left so we can visit the data and there is nothing in right as well.



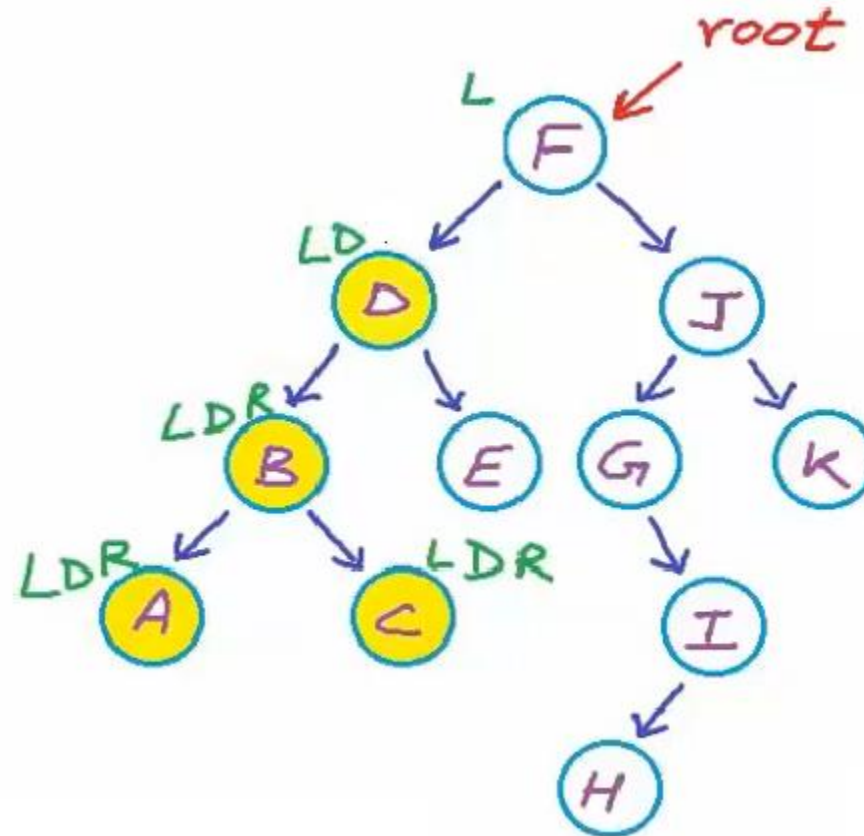


# Binary Tree Traversal

## In-Order Traversal

- No left of node (D) is completely done so we can visit it.

A, B, C, D,

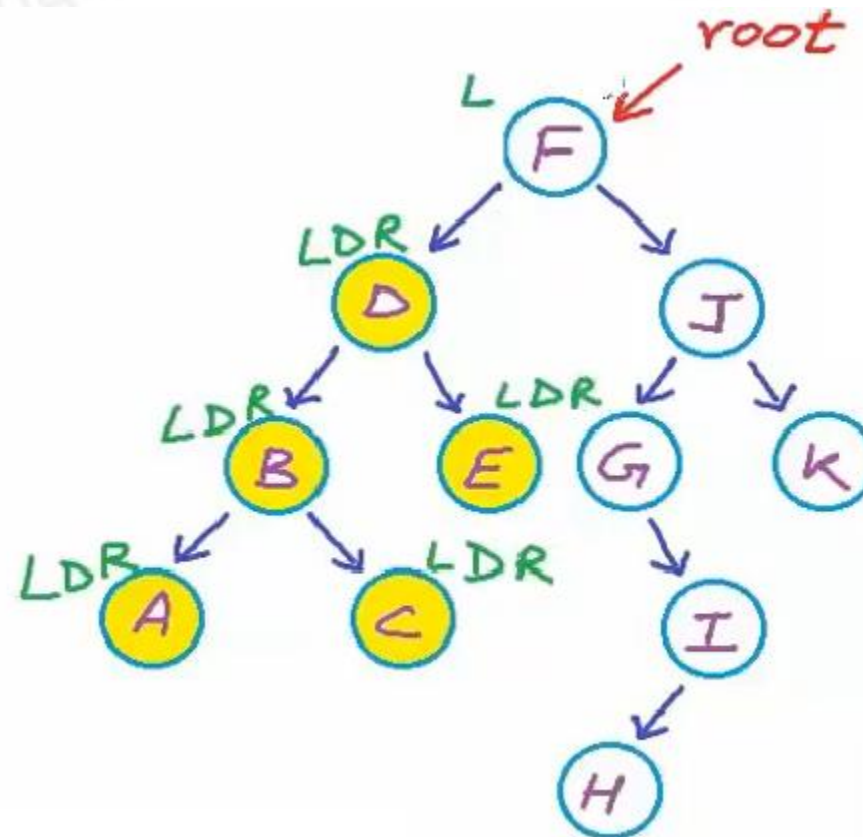


# Binary Tree Traversal

## In-Order Traversal

- Now we can go to node's (D) right to (E).
- For (E), left and right are null.

A, B, C, D, E,

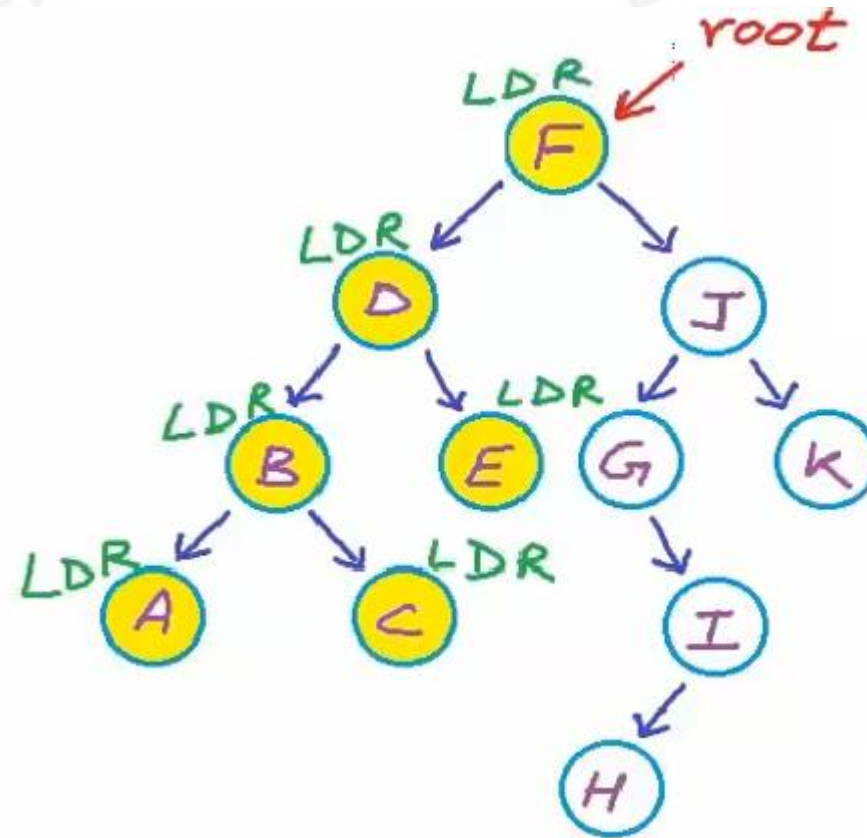


# Binary Tree Traversal

## In-Order Traversal

- At this stage left subtree of (F) is done so we can visit the data of (F) and then go to the right of it.

A, B, C, D, E, F,

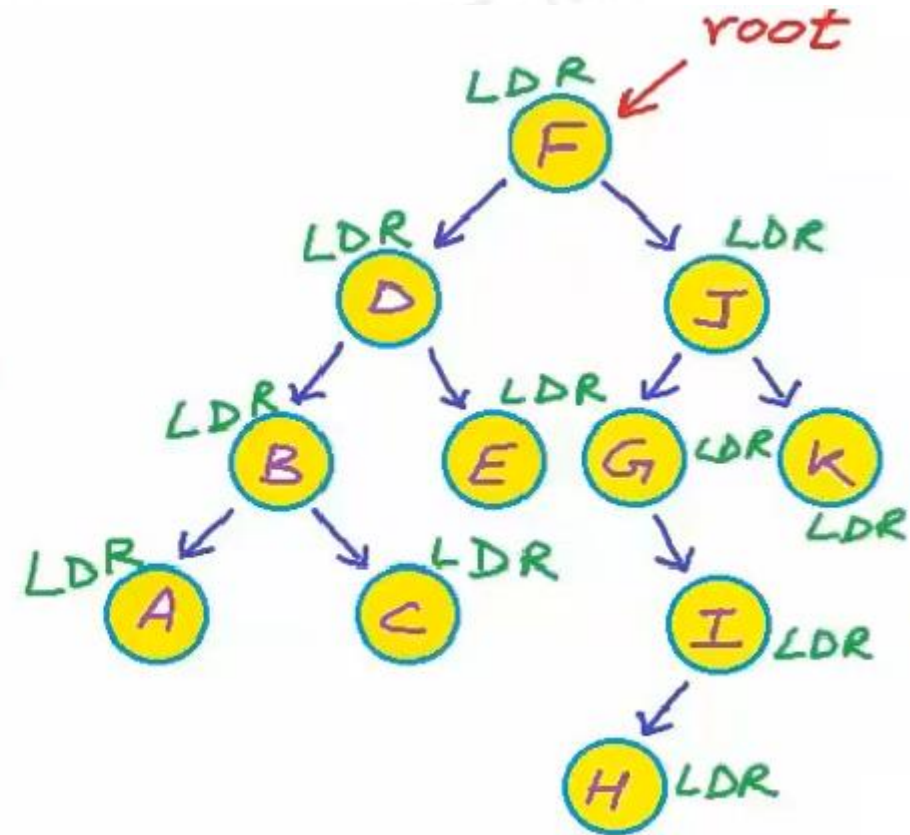


# Binary Tree Traversal

## In-Order Traversal

- If we go on like this, this finally will be our in-order traversal.

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

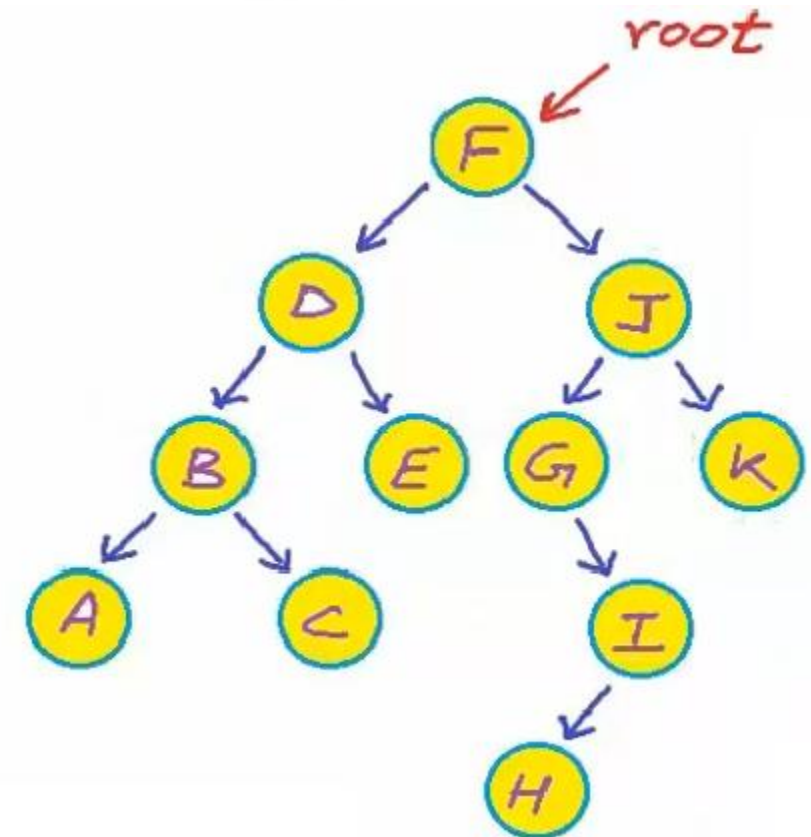


# Binary Tree Traversal

## Post-Order Traversal

- Now we should figure out the post-order traversal.
- This is what we will get for post-order traversal.

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



**Any Questions???...**