Data Structures and Algorithms

Tree

Prepared by:

Eng. Malek Al-Louzi

School of Computing and Informatics– Al Hussein Technical University

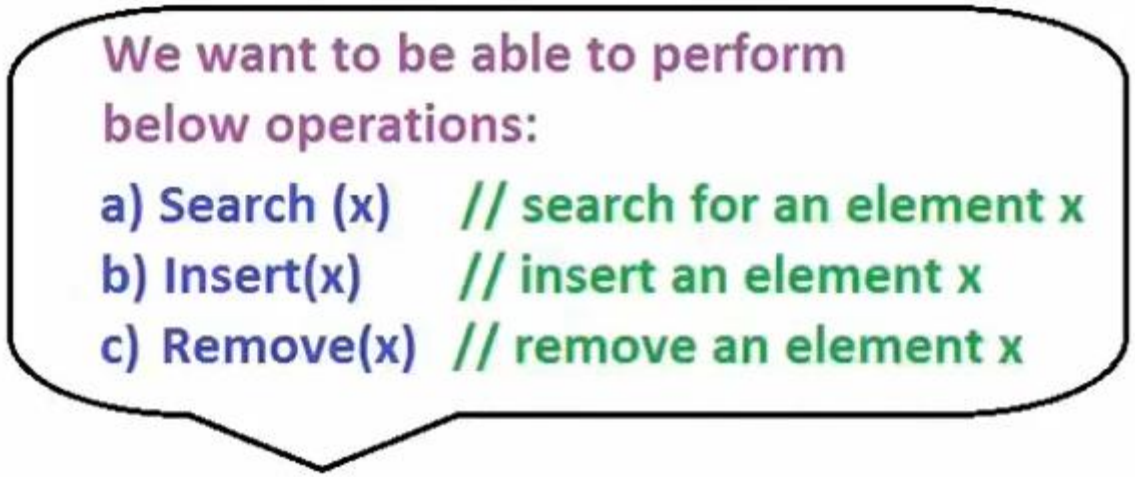
Fall 2021/2022

Outlines

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

Introduction to Trees

- BST is a special kind of binary tree.
- It is an efficient structure to organize data for quick search.
- Now, suppose you want to **store** a List in computer memory, and you want to perform the following:



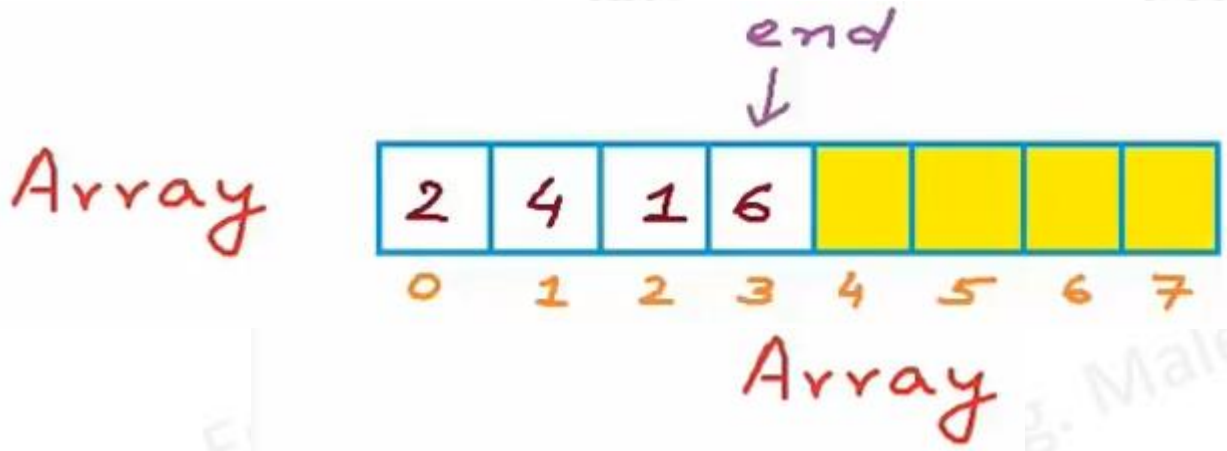
Introduction to Trees

- What data structure you will use?

- These are two well-known data structures in which we can store a collection.

Introduction to Trees

- For arrays:



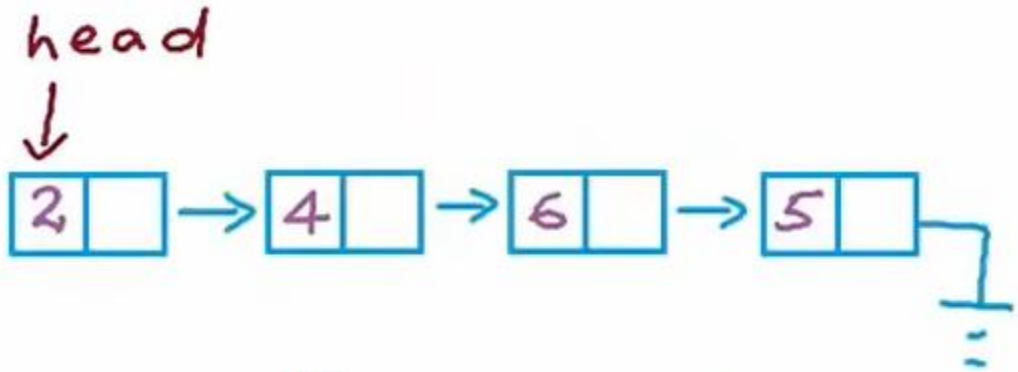
Search(x) O(n)

Insert(x) O(1)

Remove(x) O(n)

Introduction to Trees

- For linked list:



	Array	Linked List
Search(x)	O(n)	O(n)
Insert(x)	O(1)	O(1)
Remove(x)	O(n)	O(n)

Introduction to Trees

	Array (unsorted)	Linked List
Search(x)	O(n)	O(n)
Insert(x)	O(1)	O(1)
Remove(x)	O(n)	O(n)

Introduction to Trees

	Array (unsorted)	Linked List	Array (sorted)
Search(x)	O(n)	O(n)	O(log n)
Insert(x)	O(1)	O(1)	O(n)
Remove(x)	O(n)	O(n)	O(n)

Tree Definition as a Logical Model

- Can we use something better?
- If we used this data structure called BST.

	Array (unsorted)	Linked List	Array (sorted)	BST
Search(x)	O(n)	O(n)	O(log n)	O(log n)
Insert(x)	O(1)	O(1)	O(n)	O(log n)
Remove(x)	O(n)	O(n)	O(n)	O(log n)

Vocabularies that used with Tree data structure

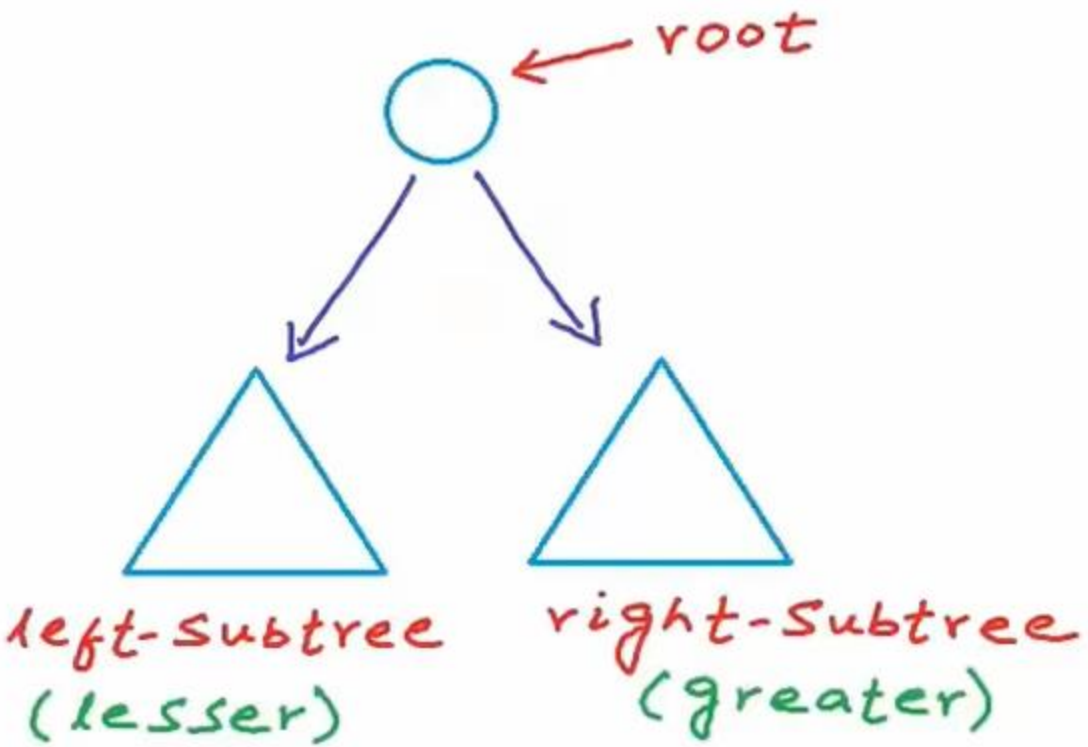
- The cost is O(n) in worst case, but we can avoid this by making sure that the tree is always balanced.
- We already talked about balanced binary tree
- BST is only a special kind of binary tree.

Vocabularies that used with Tree data structure

Binary Search Tree (BST)

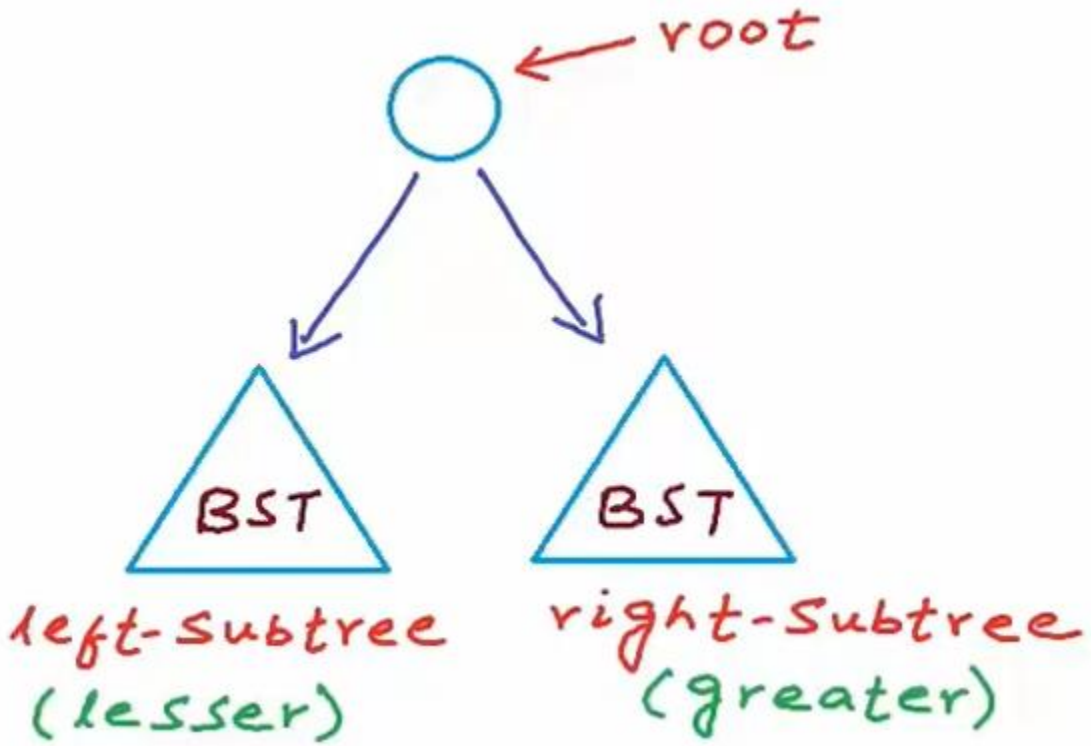
↳ a binary tree in which for each node, value of all the nodes in left subtree is lesser and value of all the nodes in right subtree is greater.

Vocabularies that used with Tree data structure



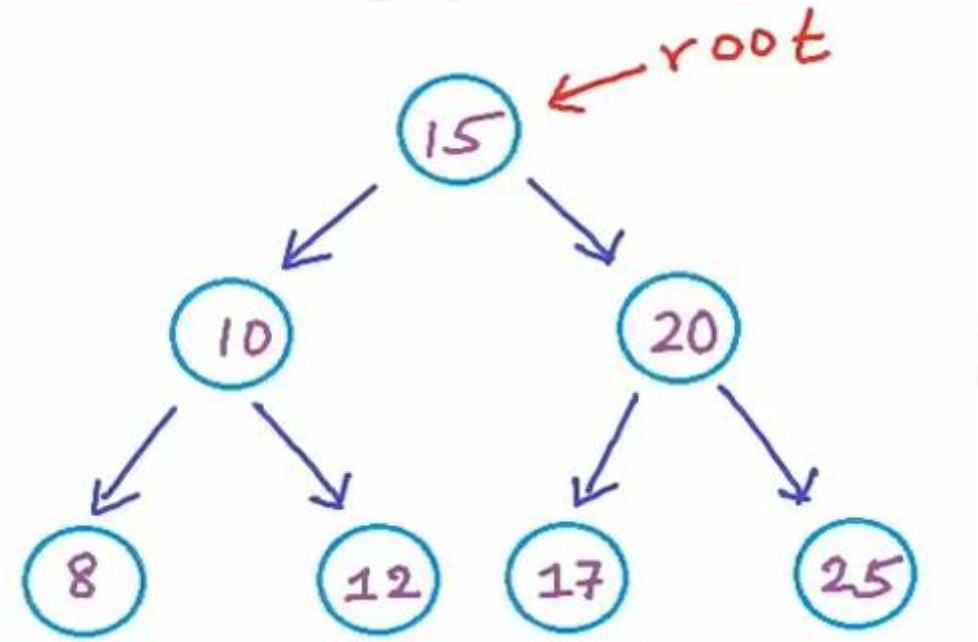
Vocabularies that used with Tree data structure

- This must be true for all the nodes



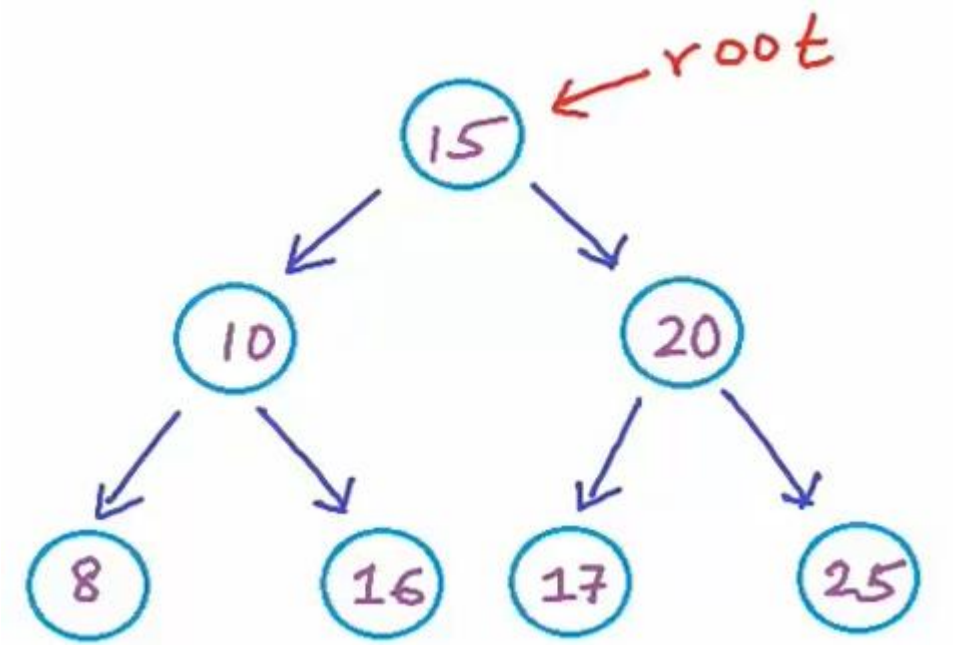
Vocabularies that used with Tree data structure

- Example:



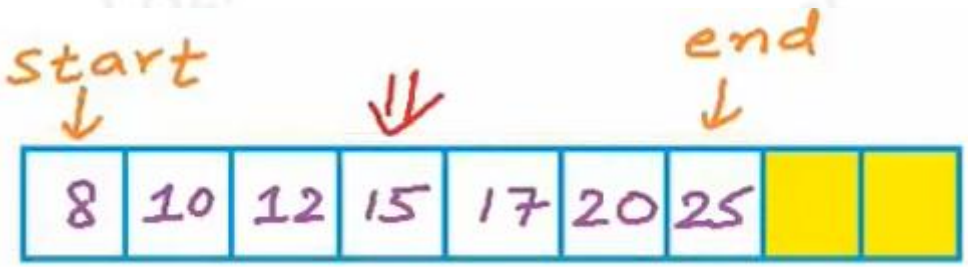
Vocabularies that used with Tree data structure

- Not BST:



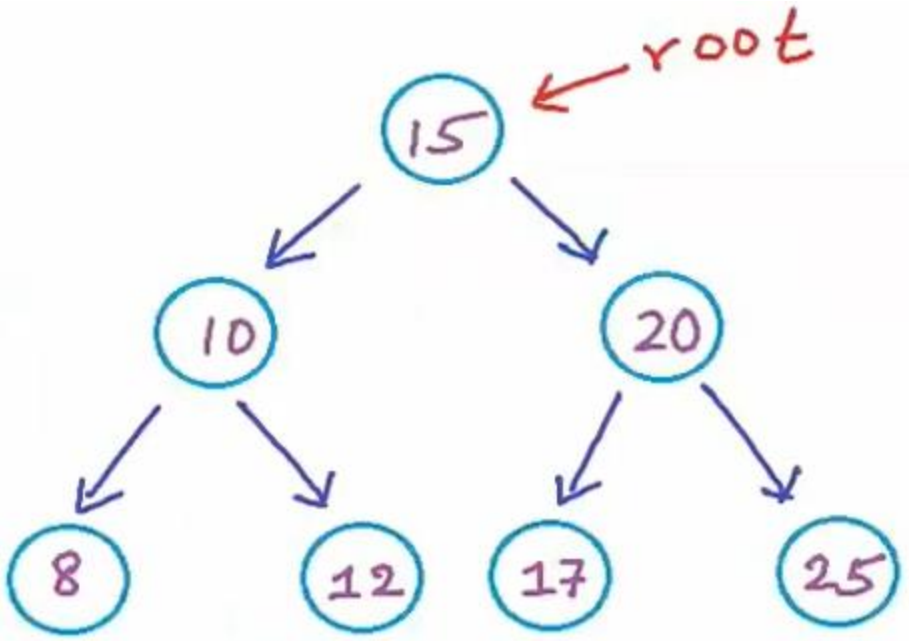
Properties of Tree

- As we were saying, search, insert and delete in O(logn), how?
- If the integers in an array binary search is, search 10:



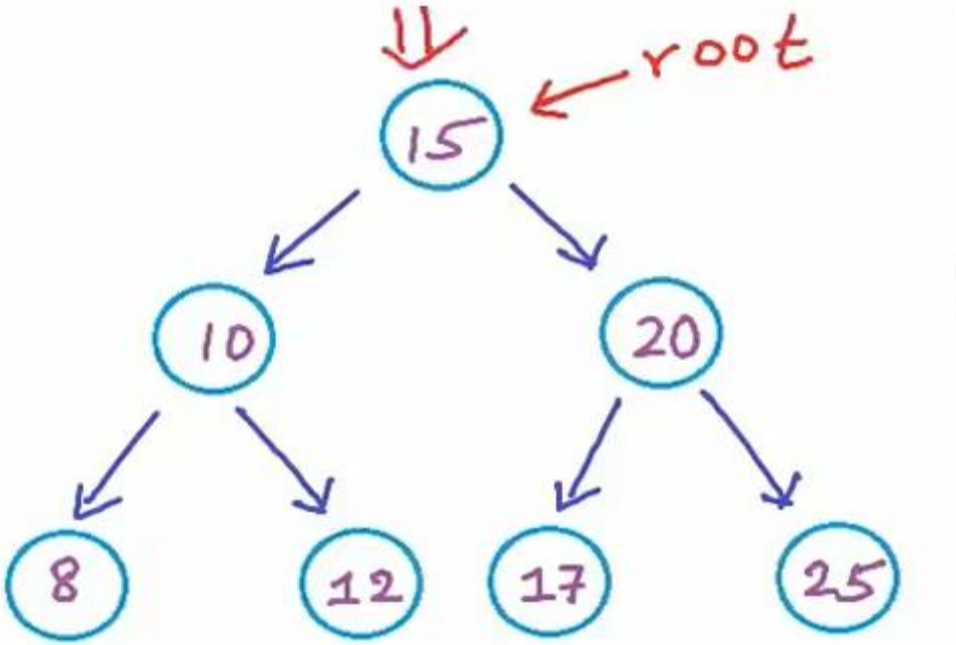
Properties of Tree

- Now, if we are using BST to store integers, **search** operation is very similar.
- We want to search 12:



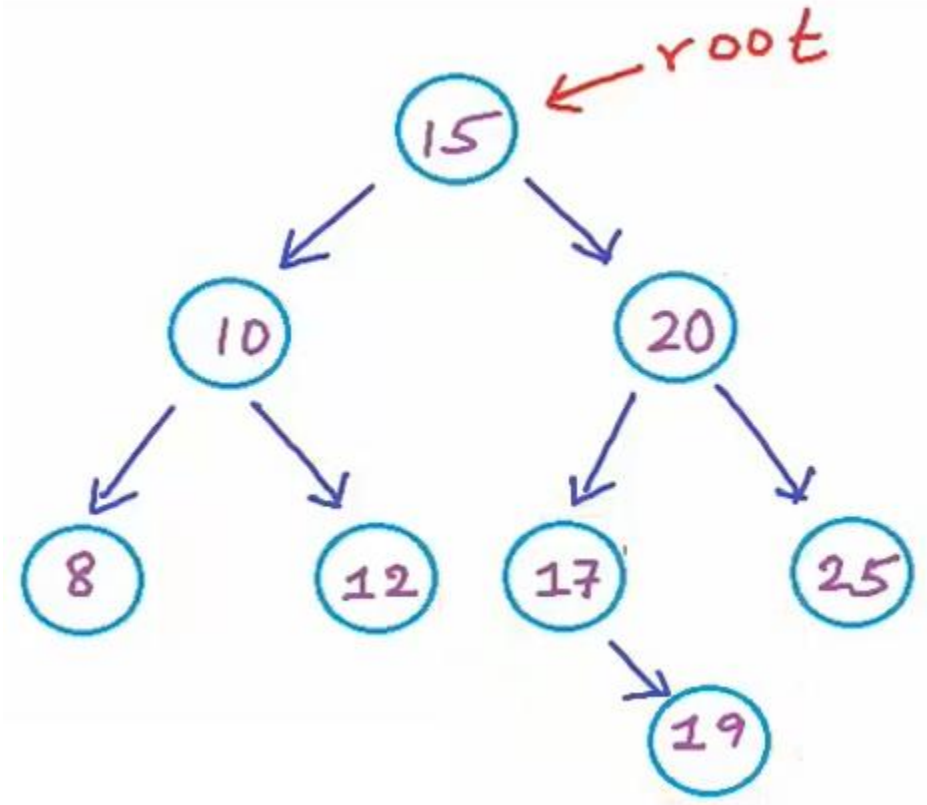
Properties of Tree

- To **insert**, first we need to find the position, we can find it in O(logn).
- Insert 19



Properties of Tree

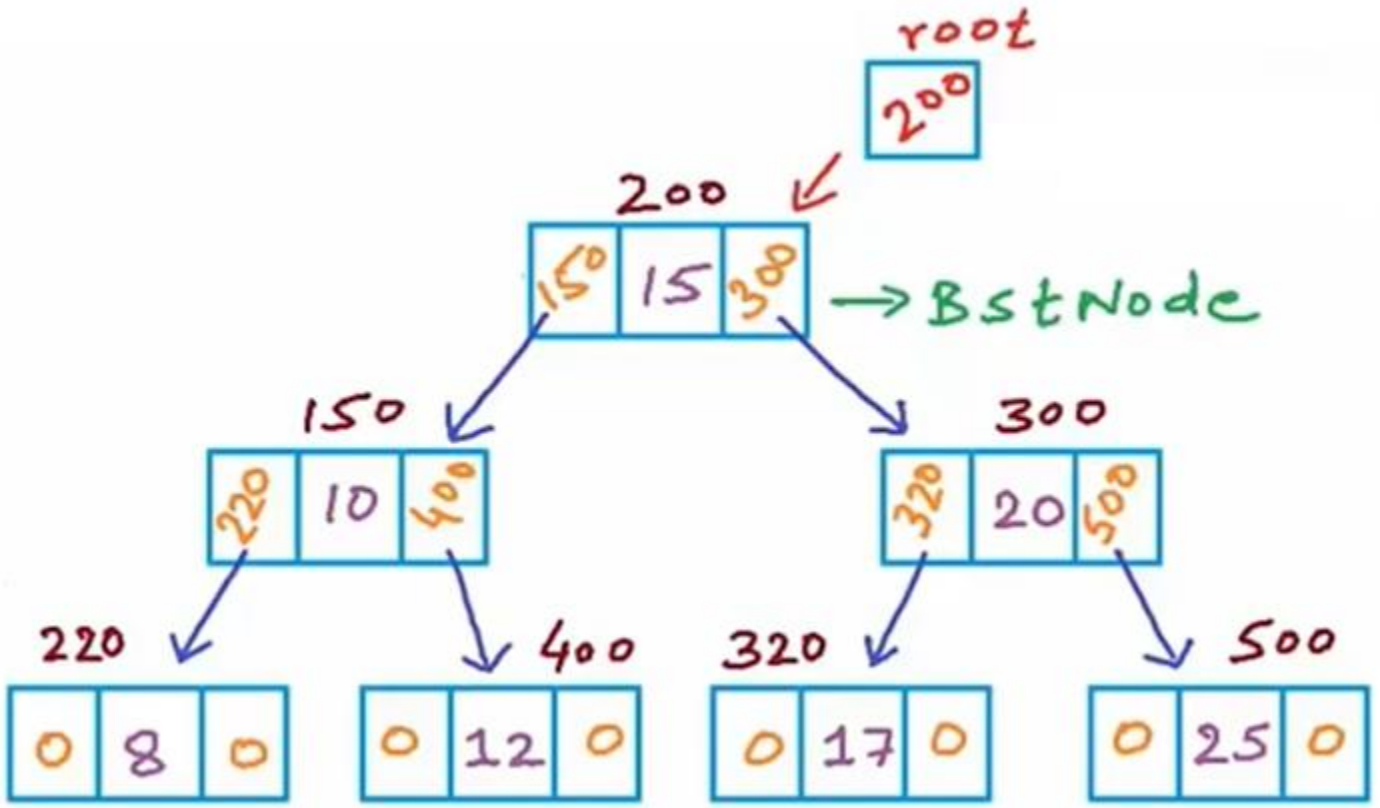
- Because we are using references, No shifting is needed like array



Properties of Tree

- To delete, we have to search the node, deleting will only adjusting some links.

Properties of Tree



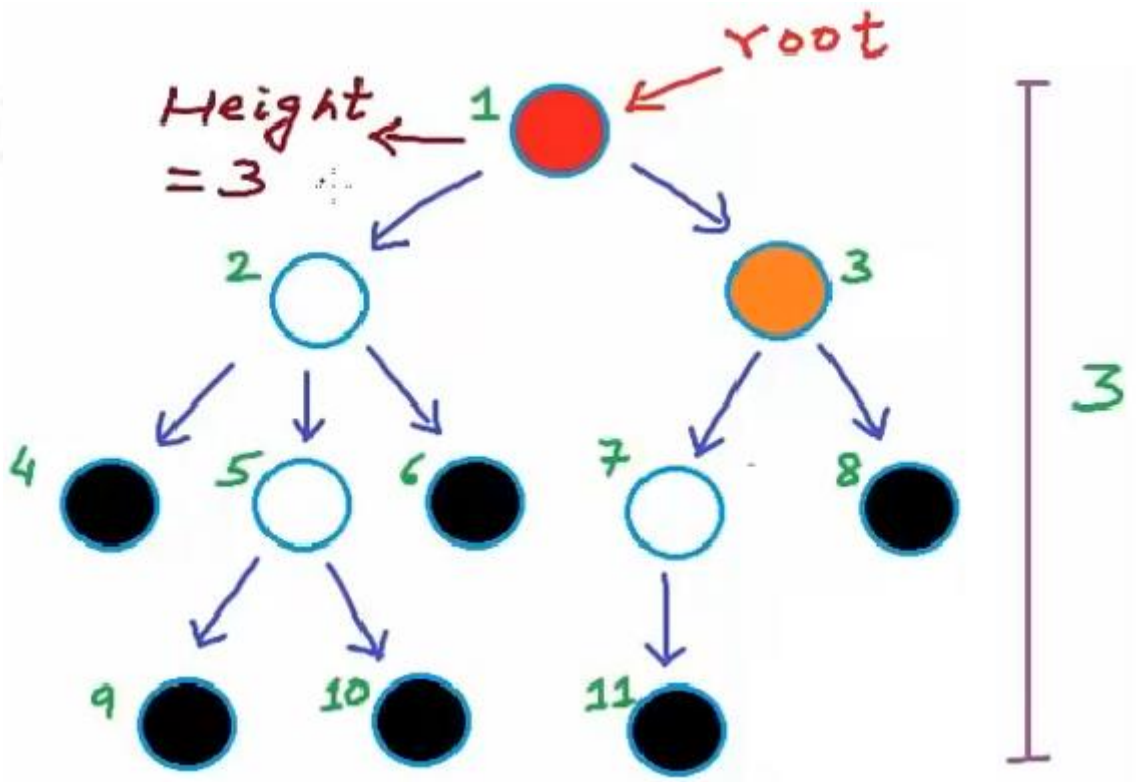
Properties of Tree

```
struct BstNode {
    int data;
    BstNode  left;
    BstNode  right;
};

BstNode  root;    // to store address of root node
```

Properties of Tree

- Heigh 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 =

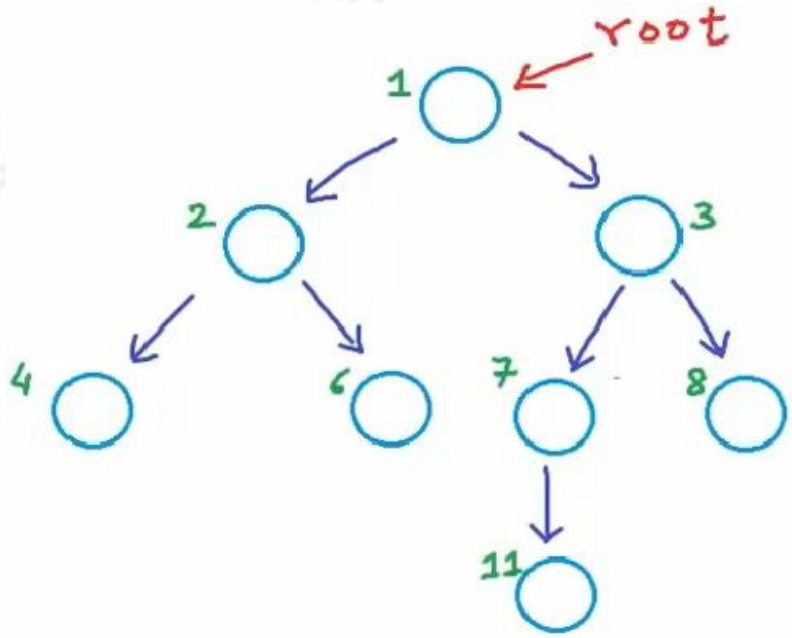
Height of root node.

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**.

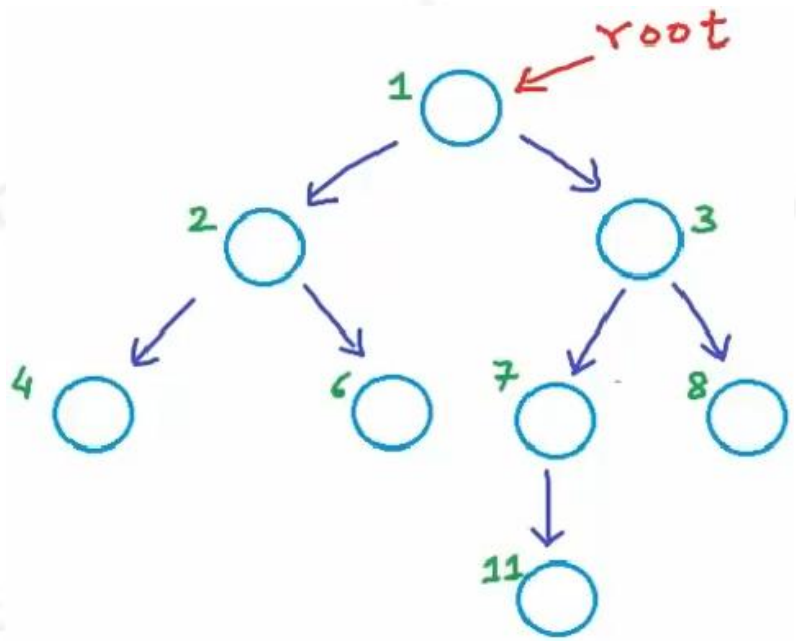


Binary Tree

Binary Tree

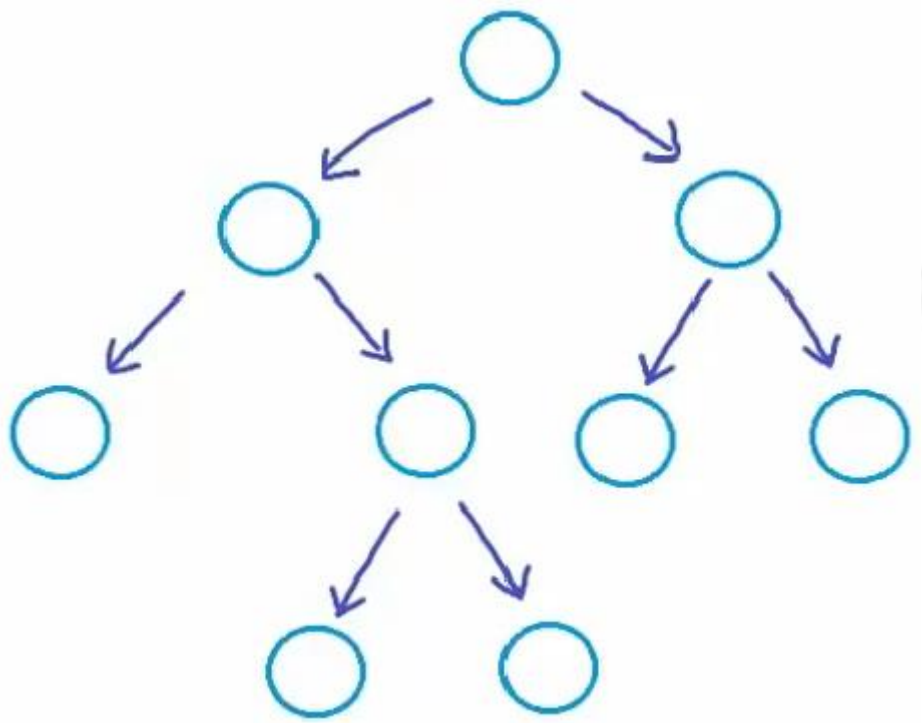
↓

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



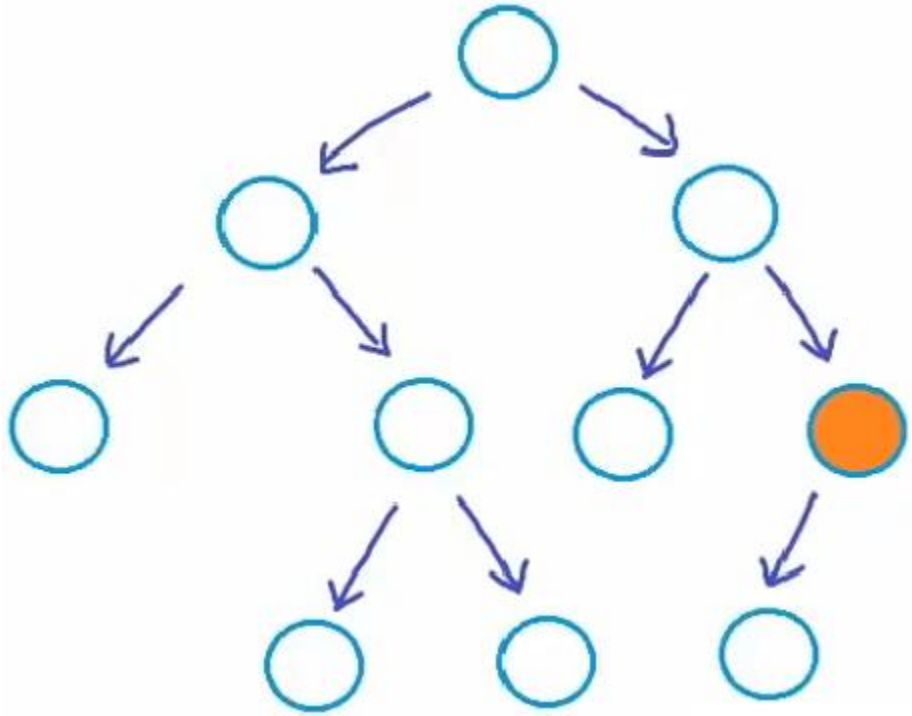
Binary Tree

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



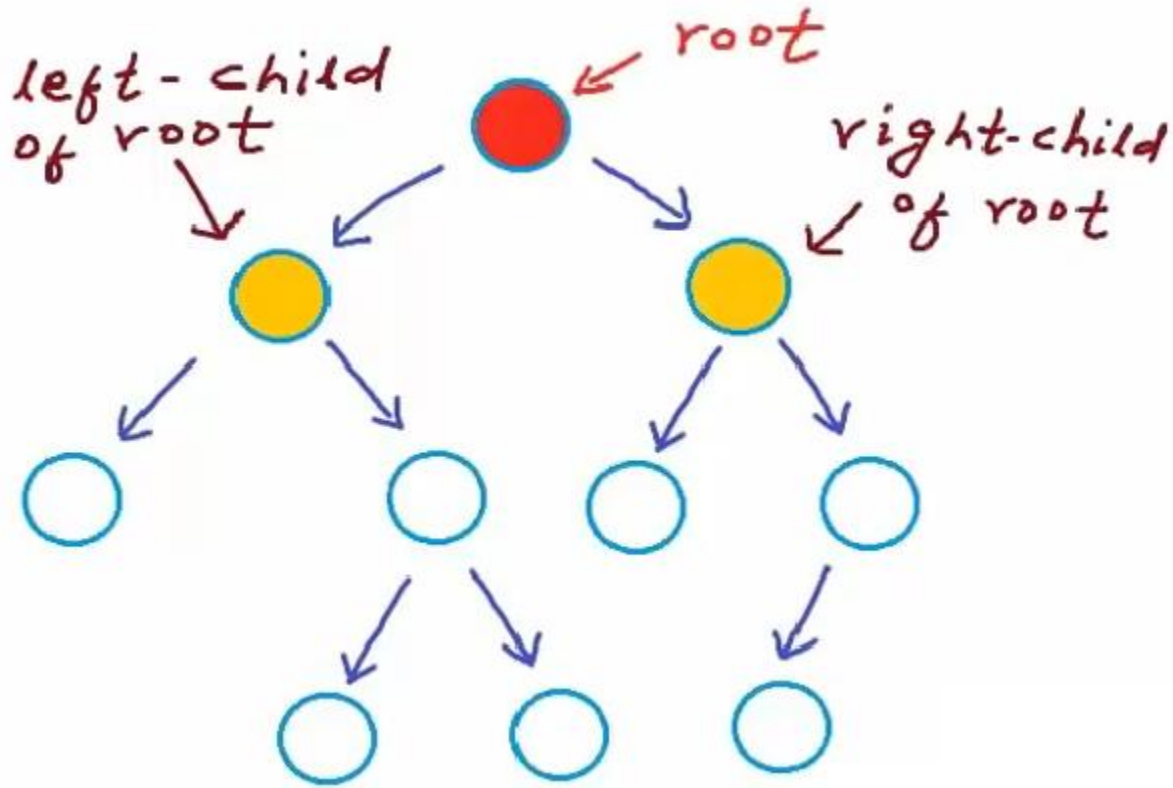
Binary Tree

- We could have a node with just One child.



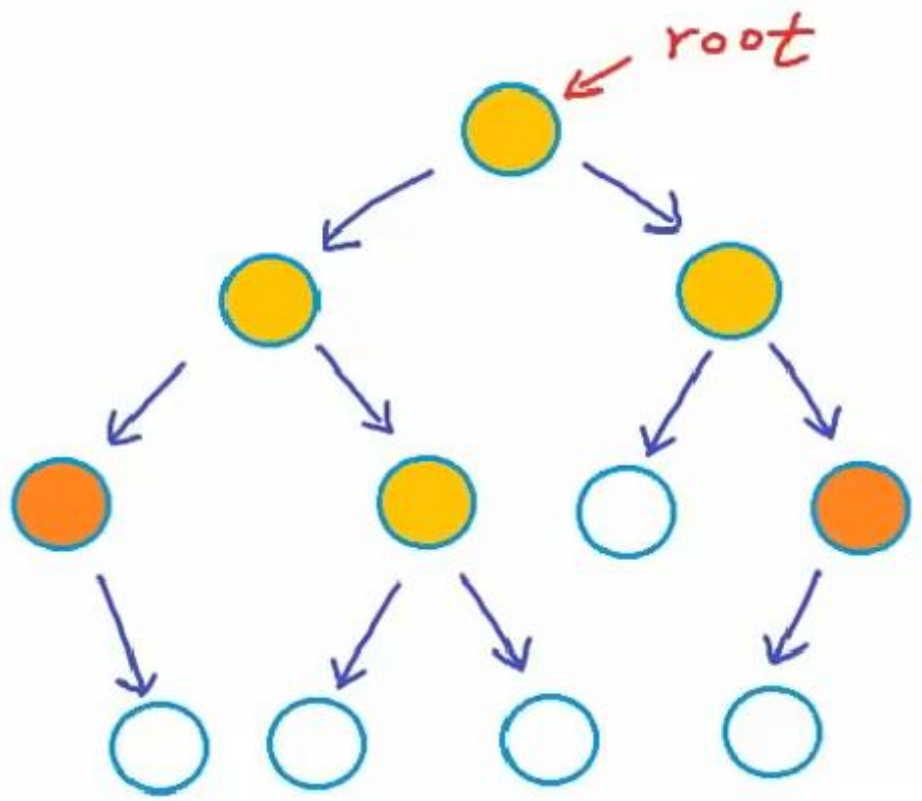
Binary Tree

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



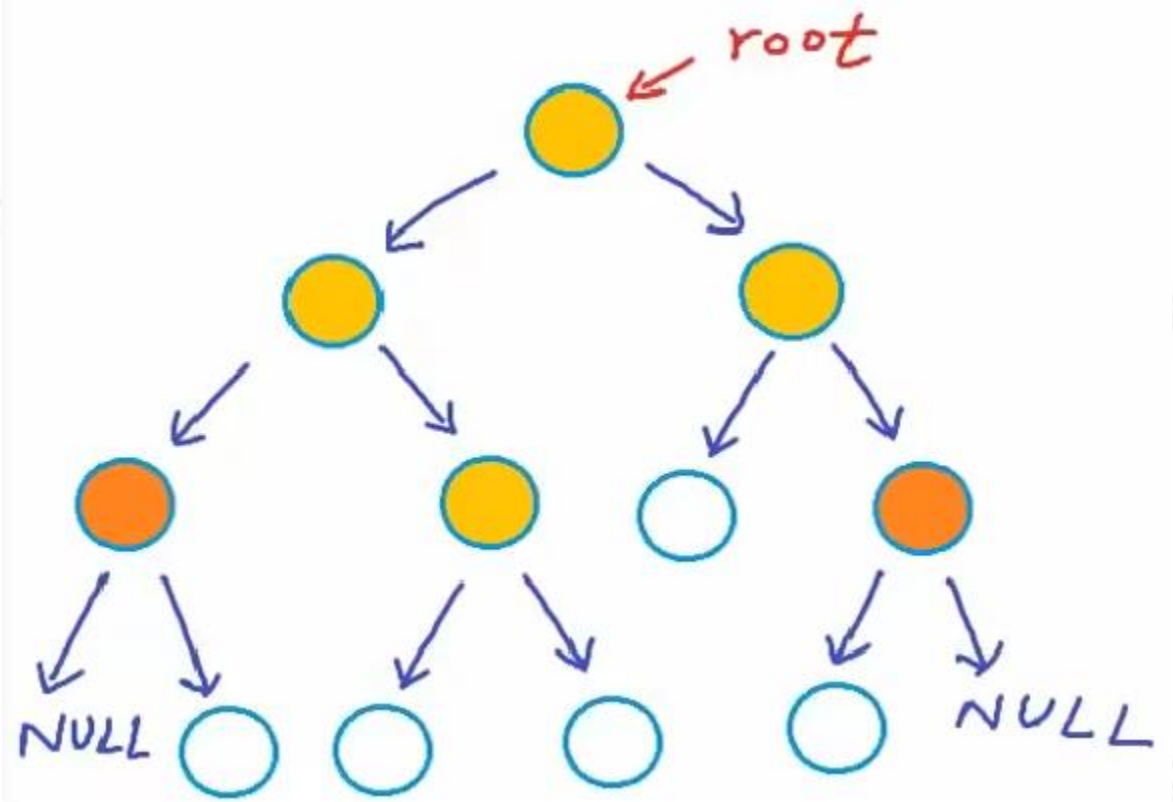
Binary Tree

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



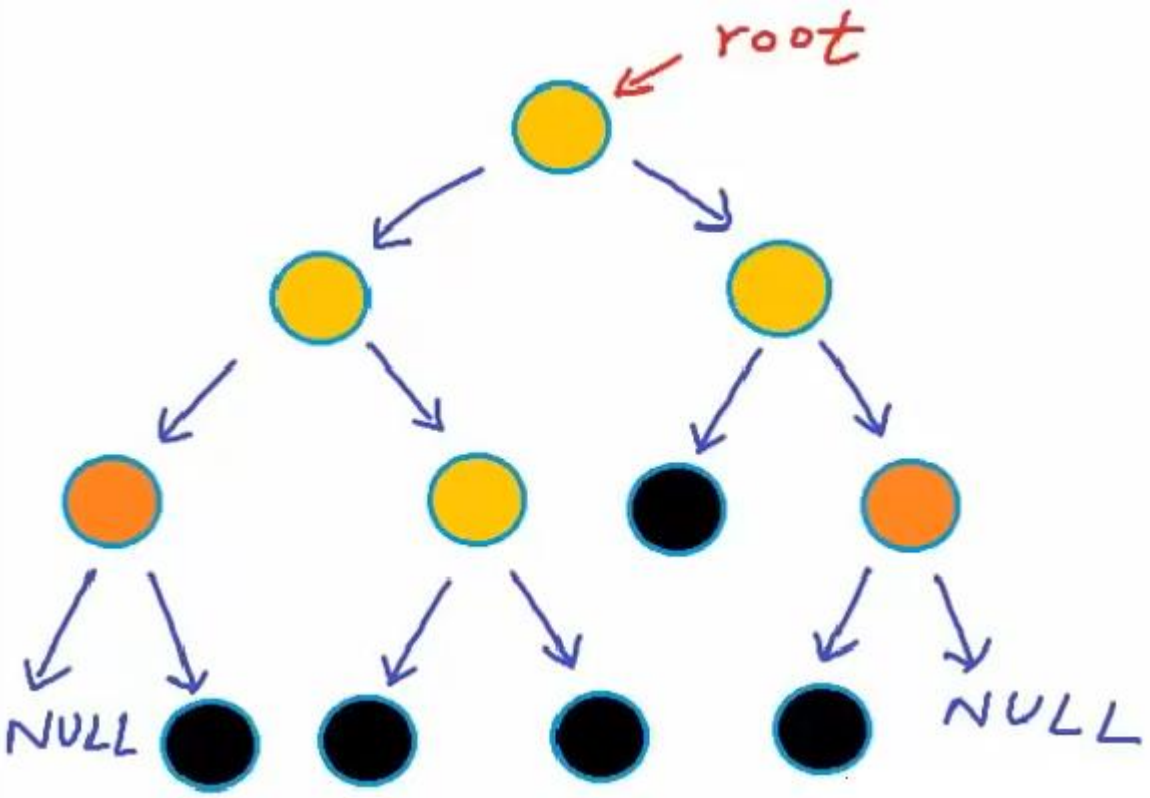
Binary Tree

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



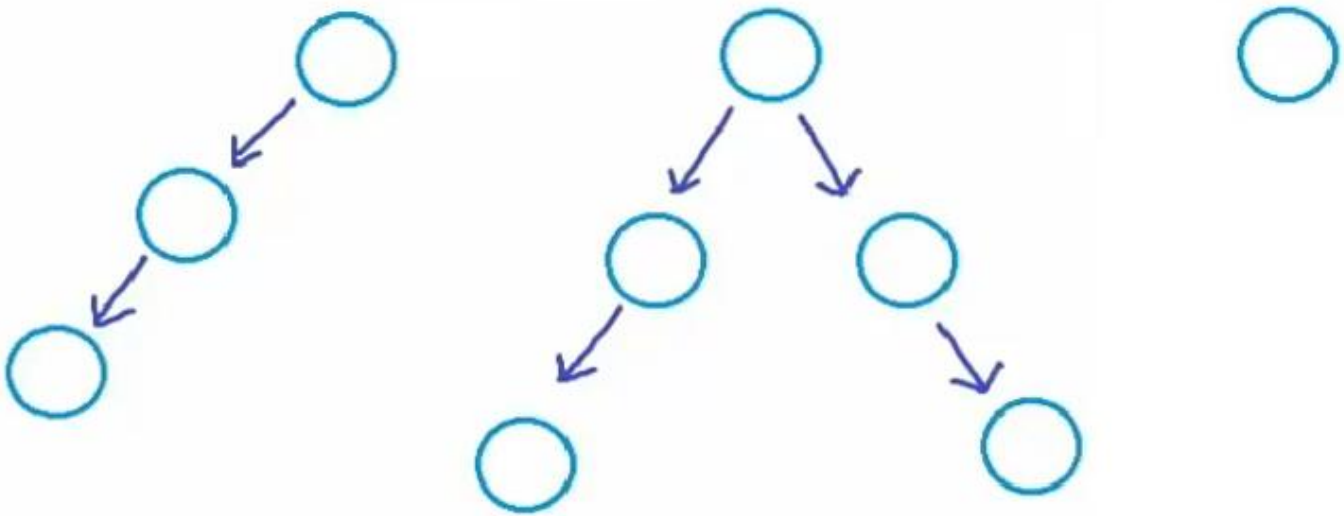
Binary Tree

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



Binary Tree

- The following three Trees are binary Trees:

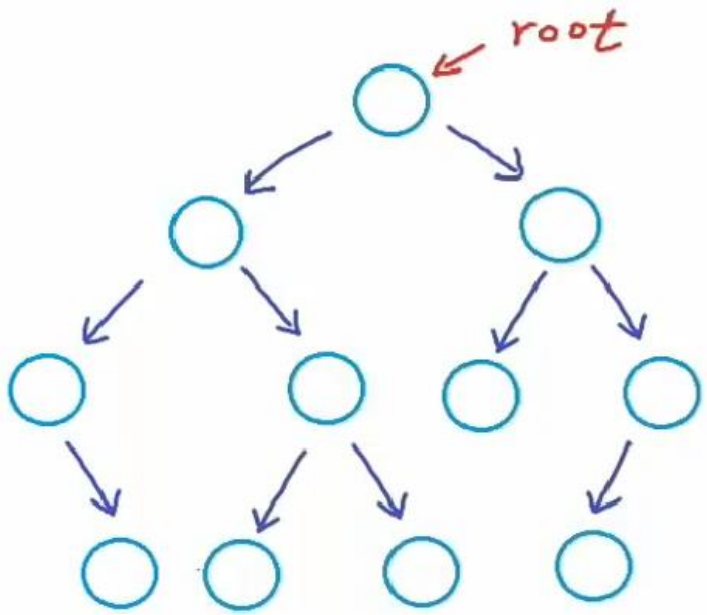


(1) (2) (3)

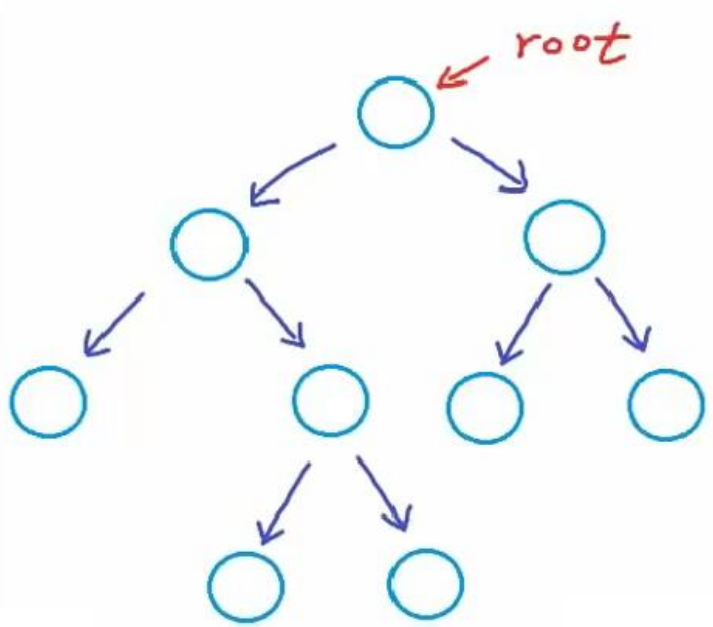
- Remember, the only condition is that a node cannot have more than Two children.

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.



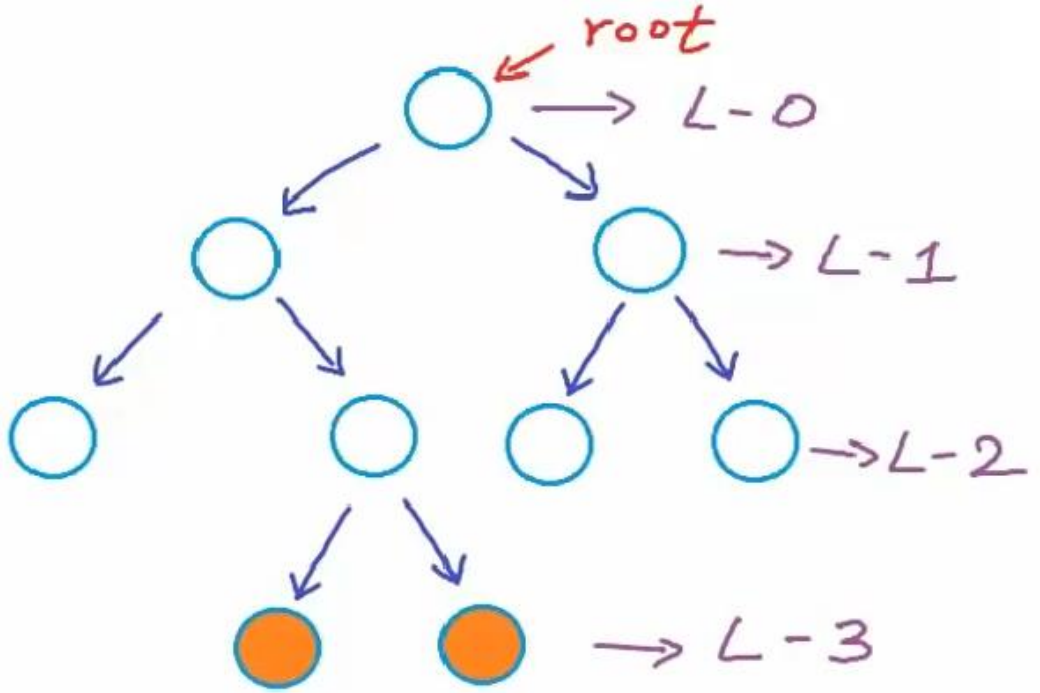
Not Strict Binary Tree



Strict Binary Tree

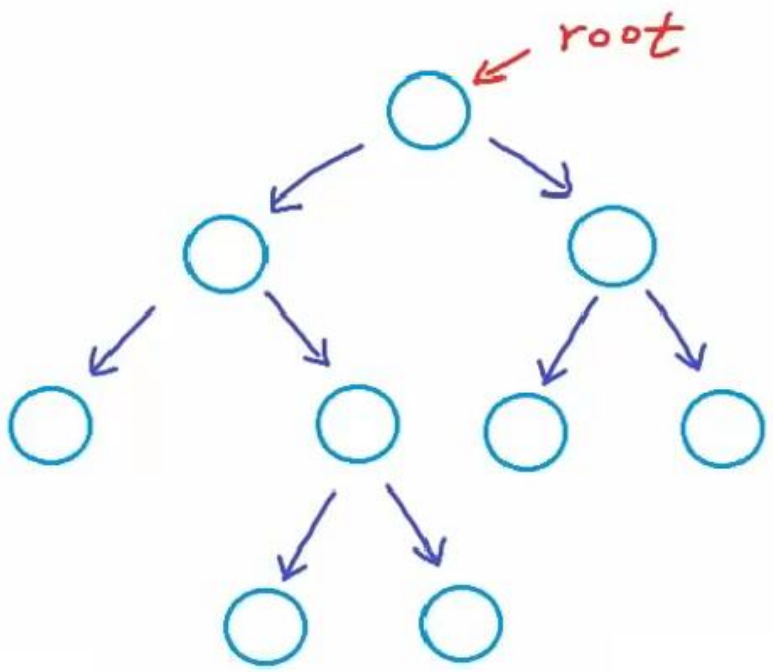
Binary Tree

- 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.

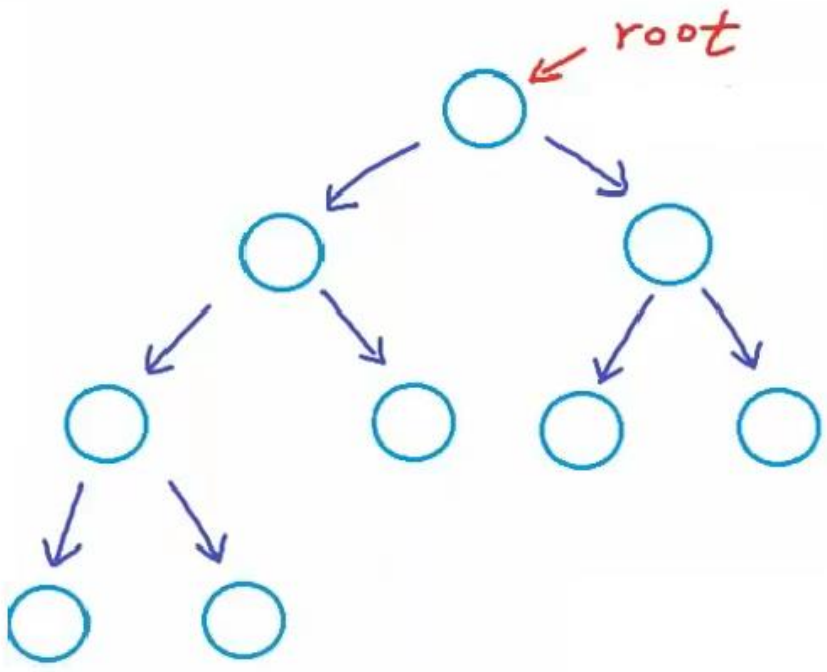


Binary Tree

- 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.



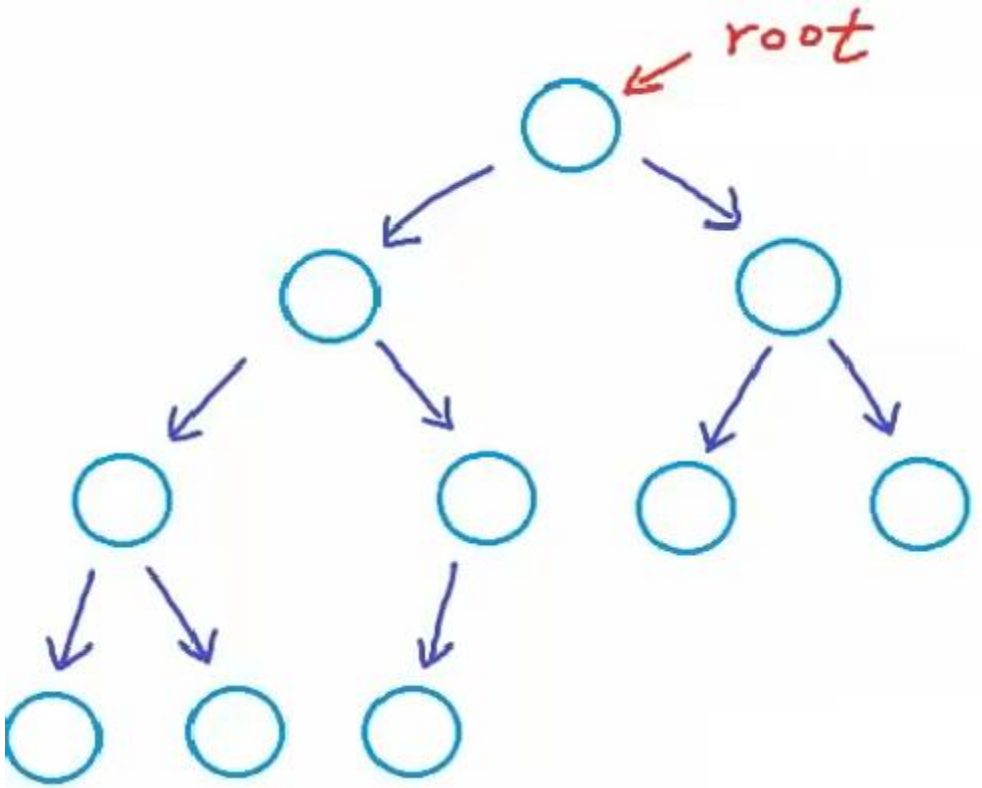
Not Complete Binary Tree



Complete Binary Tree

Binary Tree

- This is also a complete binary tree:



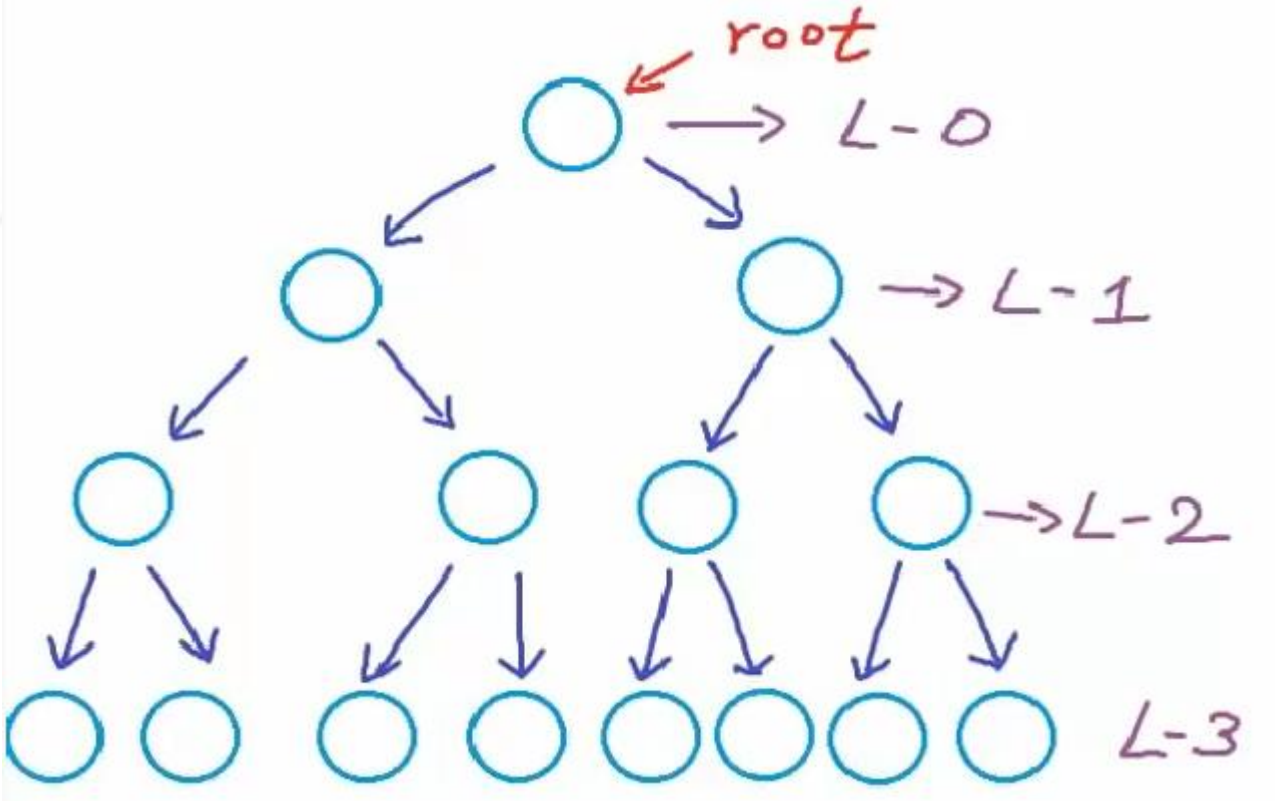
Binary Tree

- 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:

Maximum no. of nodes
in a binary tree with height h

$$= 2^0 + 2^1 + \ldots + 2^h$$

$$= 2^{h+1} - 1$$

$$= 2^{(\text{no. of levels})} - 1$$

- 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?

n = no. of nodes

$$n = 2^{h+1} - 1$$
$$\Rightarrow 2^{h+1} = (n+1)$$
$$\Rightarrow h = \log_2(n+1) - 1$$

- We can also calculate height as following:

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

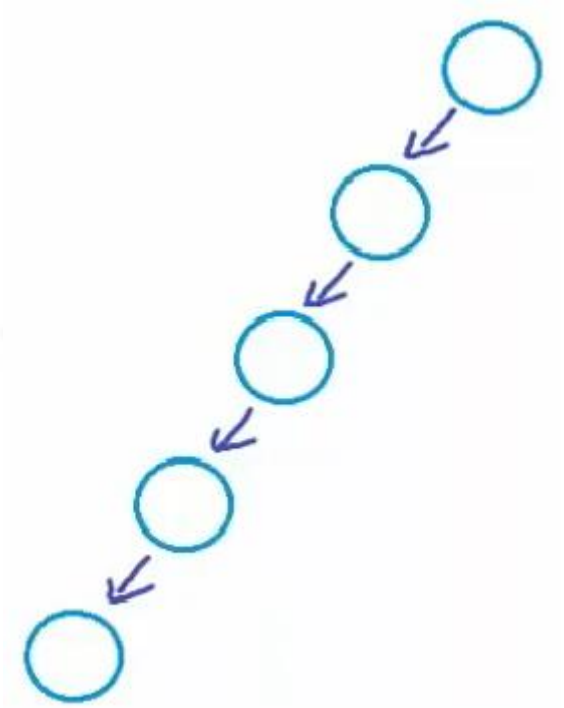
Binary Tree

- 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.

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 heigh possible is (**Dense tree**):

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

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

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

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 → no. of edges in longest path from root to a leaf

- Heigh of a tree with one node:

Height of tree with 1 node = 0

- Heigh of an empty tree:

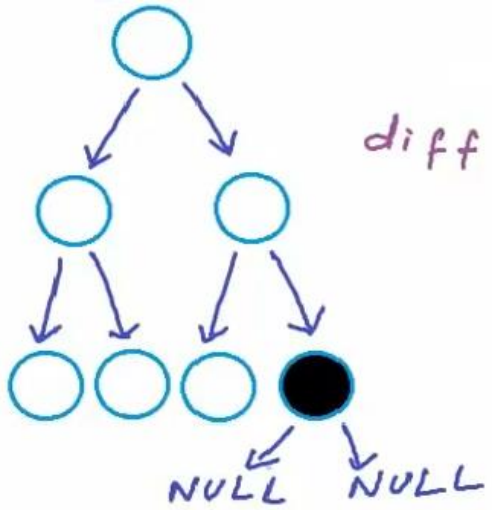
Height of an empty tree = -1

Binary Tree

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

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

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

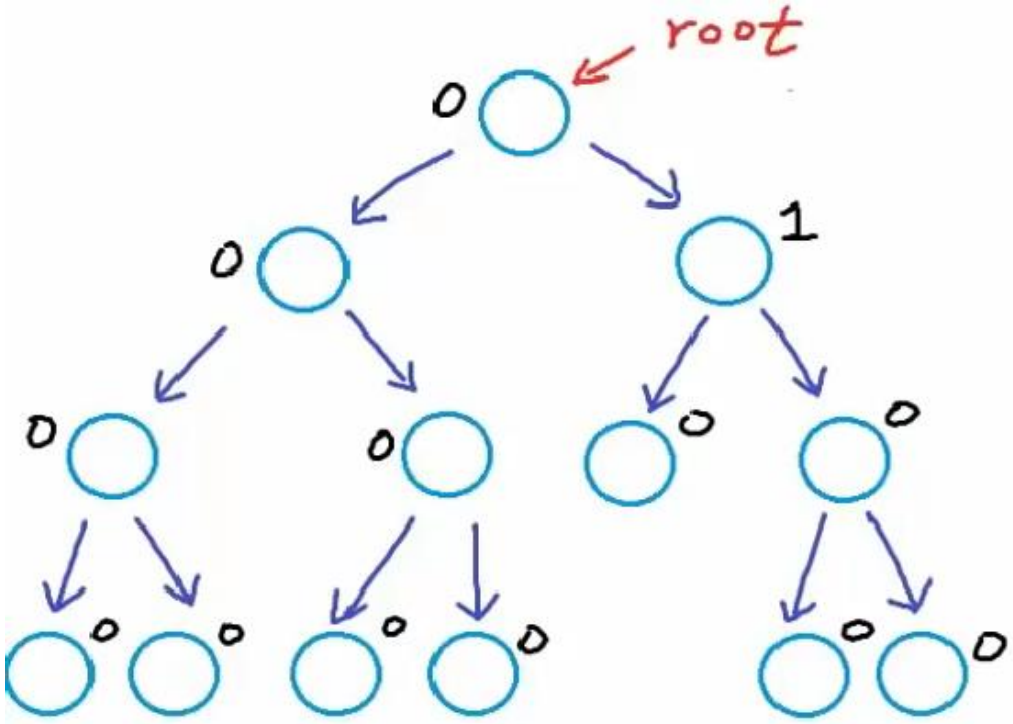


$$diff = |h_{left} - h_{right}|$$

$$|-1 - (-1)| = 0$$

Example 1

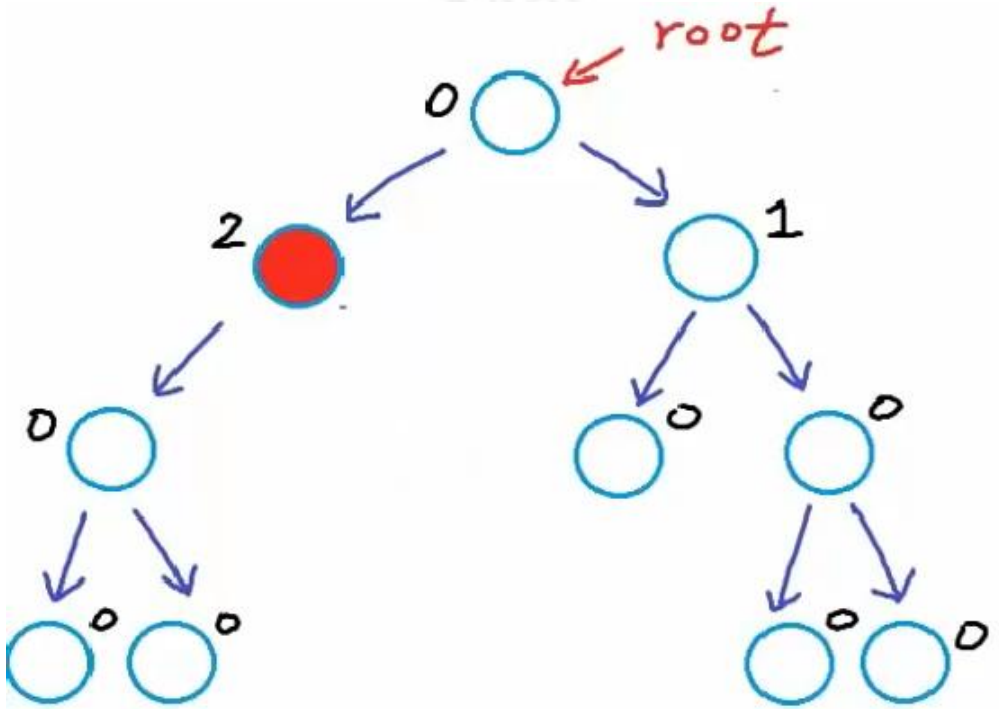
- The value of difference between heights of left and right subtrees of each node:



- 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:



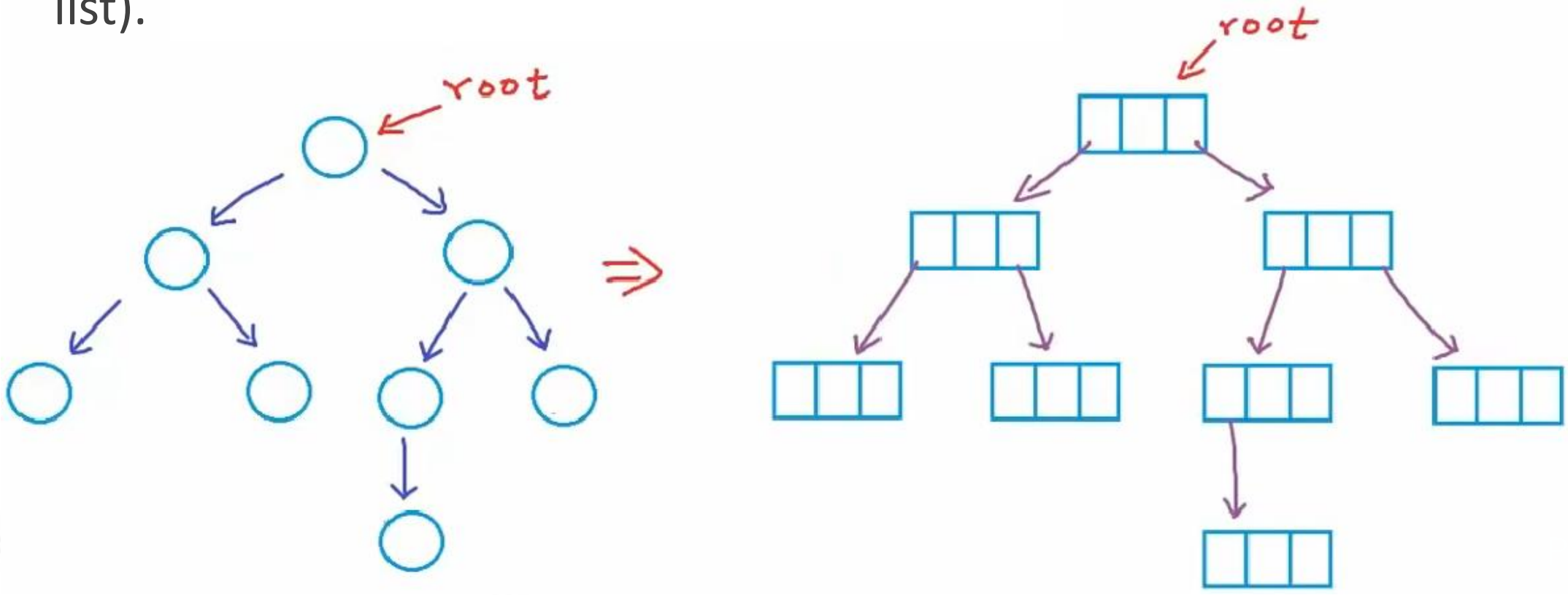
- Not balanced because one of the nodes has difference 2.

Binary Tree Implementation

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

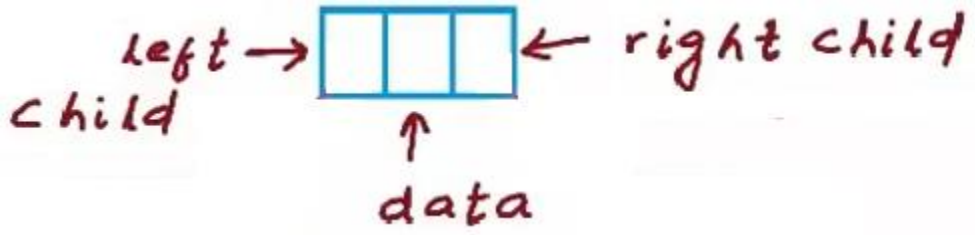
Binary Tree Implementation
Dynamically created nodes

- 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 filed is to store Data.
  - Left filed is to store a reference to the left child.
  - Right filed is to store a reference to the right child.



Binary Tree Implementation
Dynamically created nodes

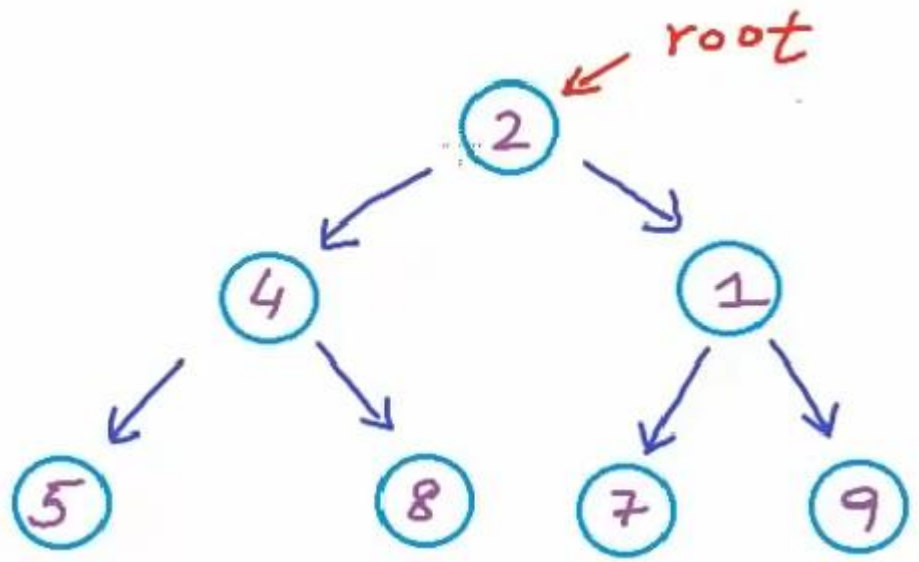
- The created nodes are linked to each other using references.
- For a binary tree of integers, we can define a node like this:

```
class Node{

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

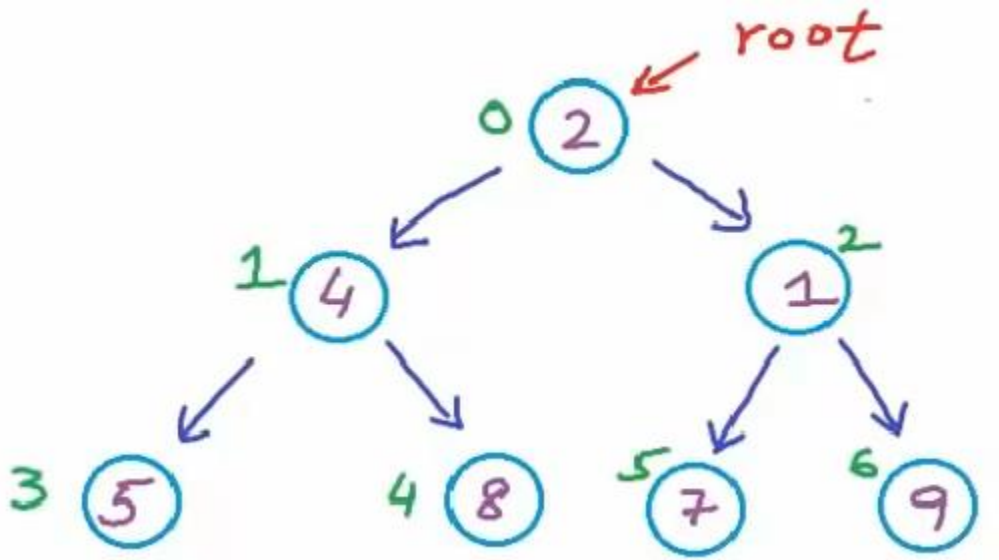
Binary Tree Implementation
Arrays

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



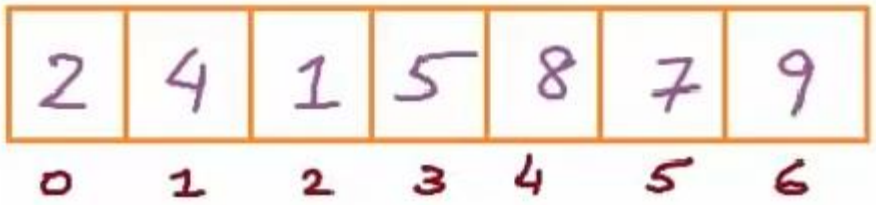
Binary Tree Implementation
Arrays

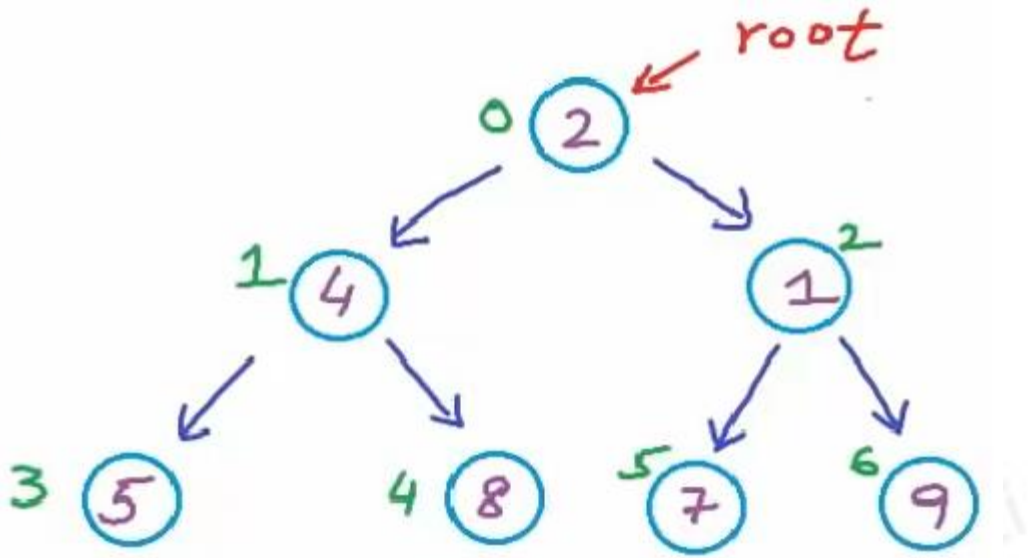
- We can number these nodes from 0 starting at root, going level by level from left to right:



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?

for node at index i,

$$\text{left-child-index} = 2i + 1$$

$$\text{right-child-index} = 2i + 2$$

$$\text{Parent-index} = \left\lfloor \frac{i-1}{2} \right\rfloor$$

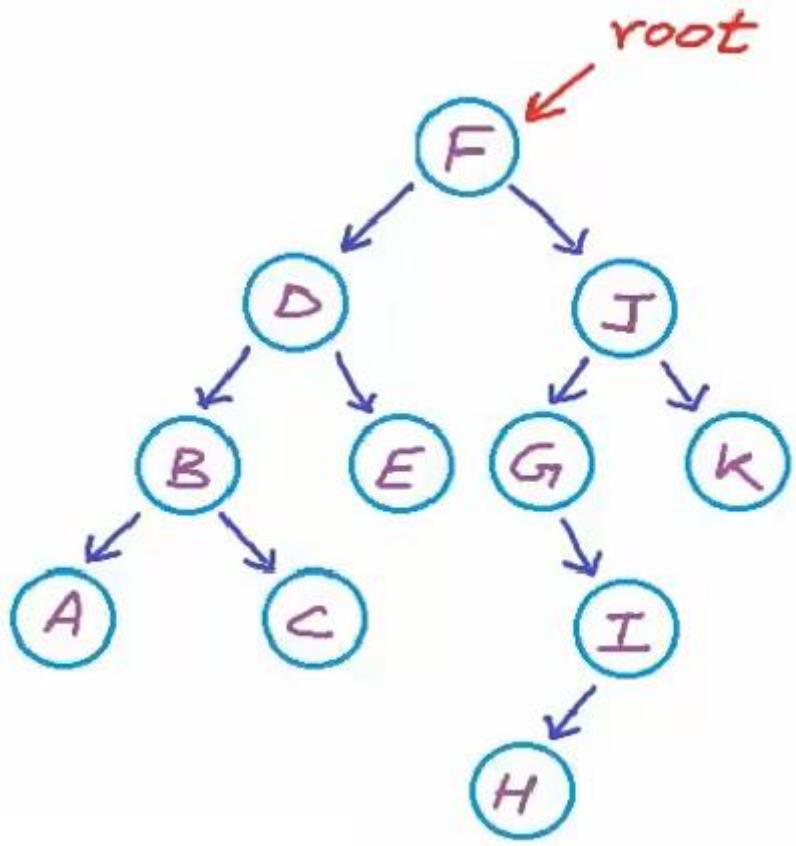
- **Remember, this is true only for a complete binary tree.**

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:

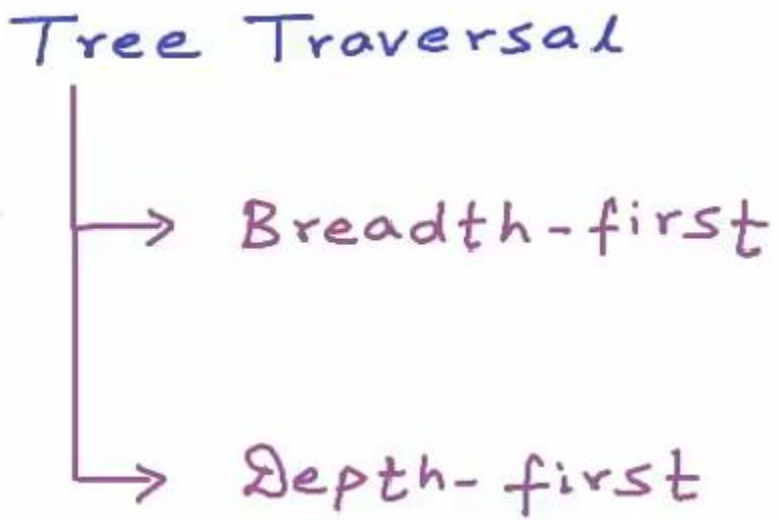
Tree Traversal
↳ process of visiting
each node in the tree
exactly once in some order.

- 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**.

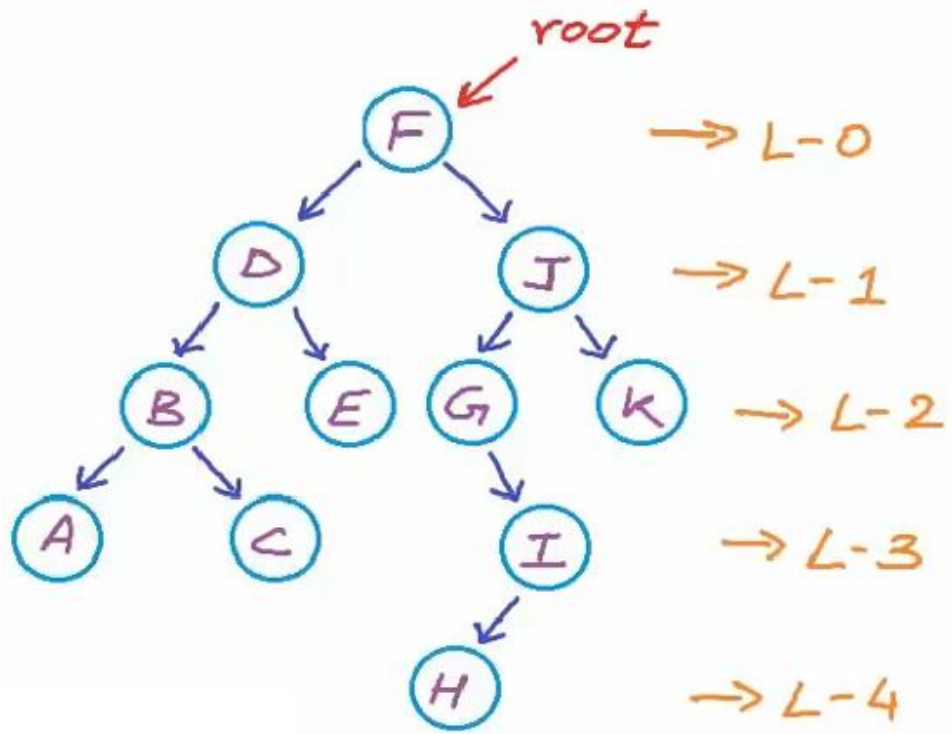


Binary Tree Traversal

- 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

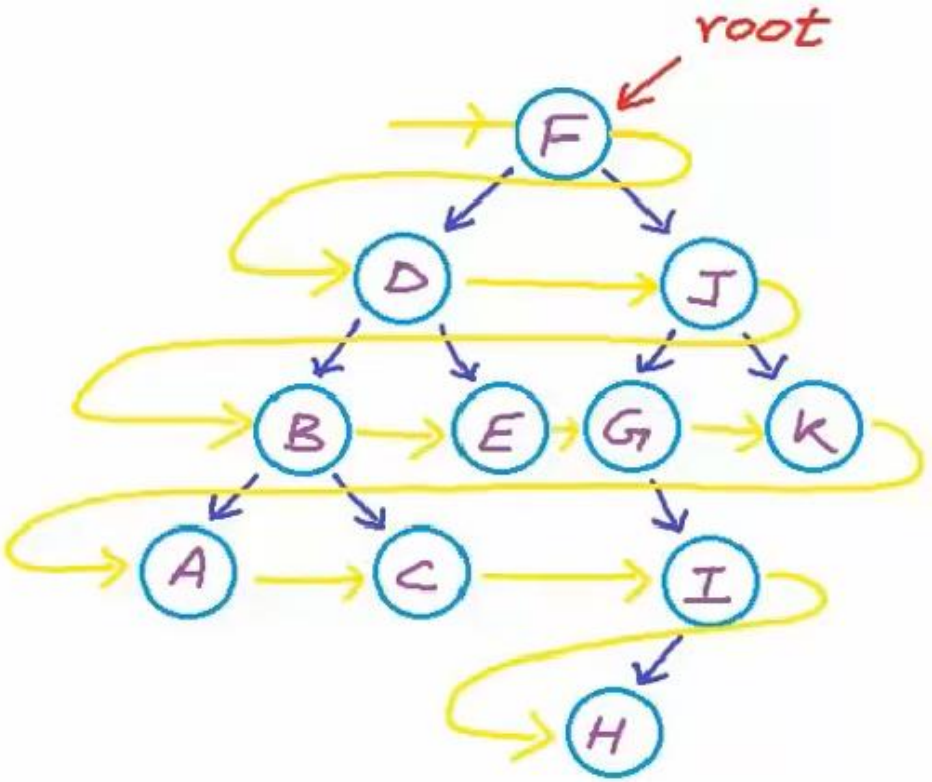
- In a tree, in breadth-first approach we would visit all the nodes at same depth or level before visiting the nodes at next level.
- Referring to the previous binary tree of characters, the levels of the nodes are:



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.

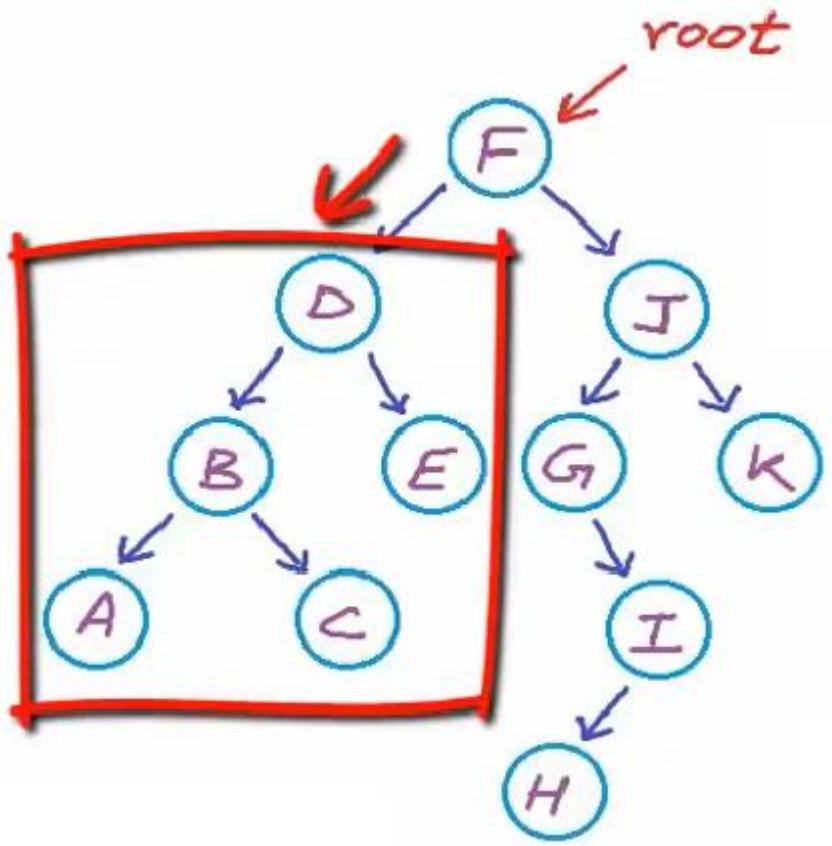
F, D, J, B, E, G, K, A, C, I, H



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

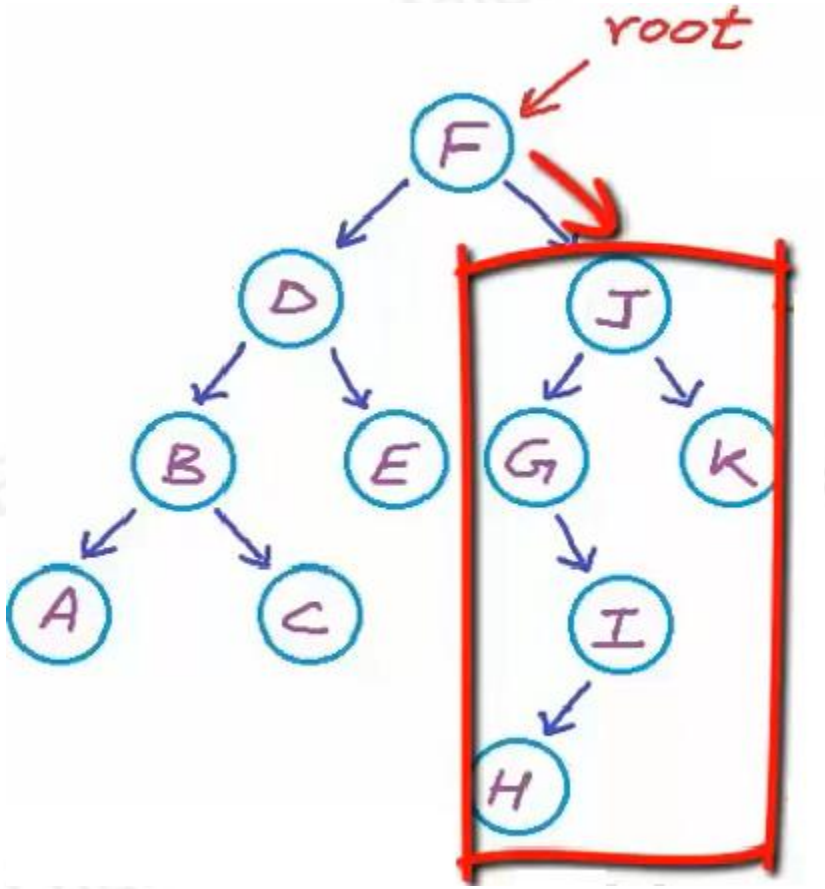
Binary Tree Traversal
Depth-First Approach

- 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).



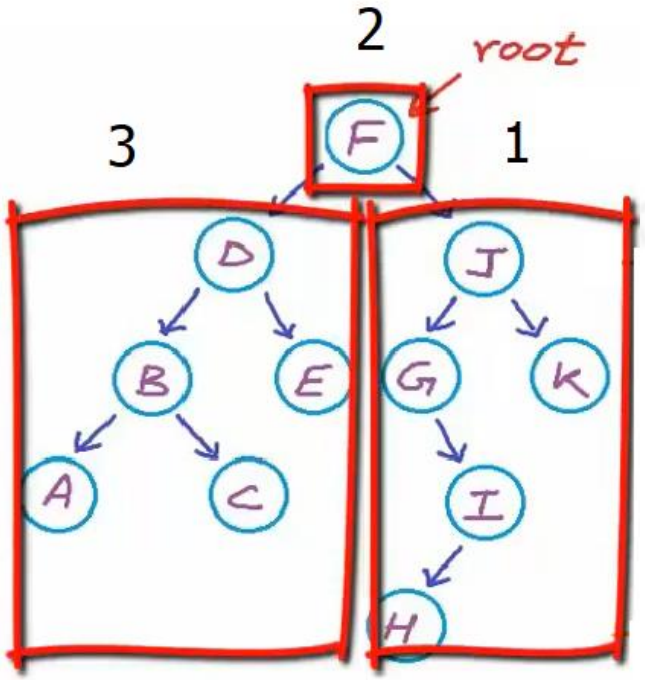
Binary Tree Traversal
Depth-First Approach

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



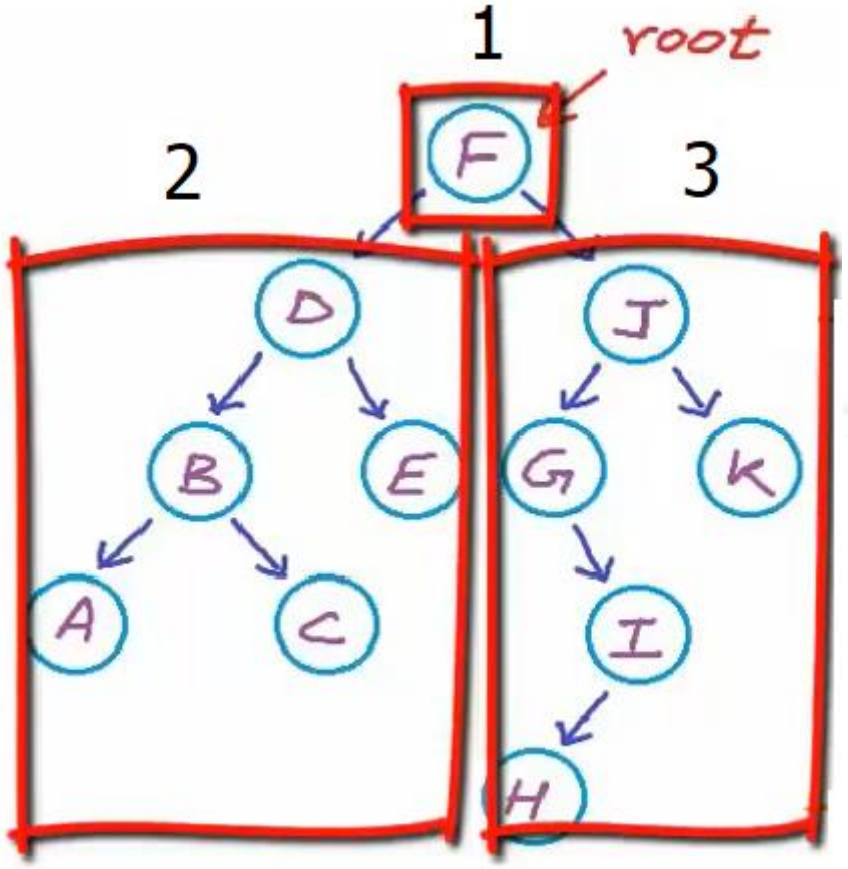
Binary Tree Traversal
Depth-First Approach

- 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.



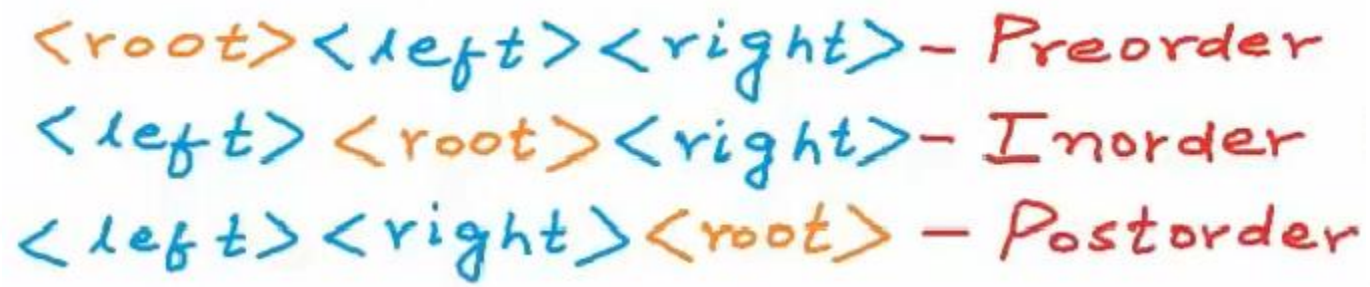
Binary Tree Traversal
Depth-First Approach

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



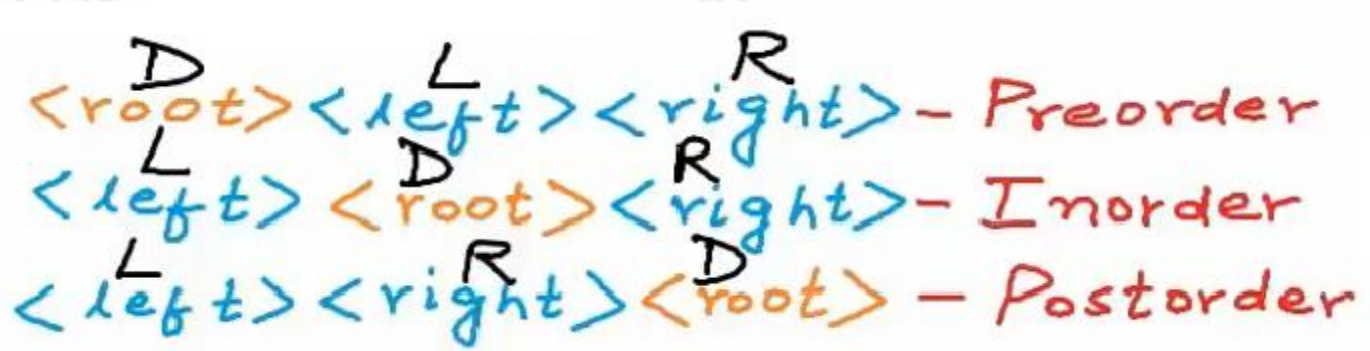
Binary Tree Traversal
Depth-First Approach

- 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:



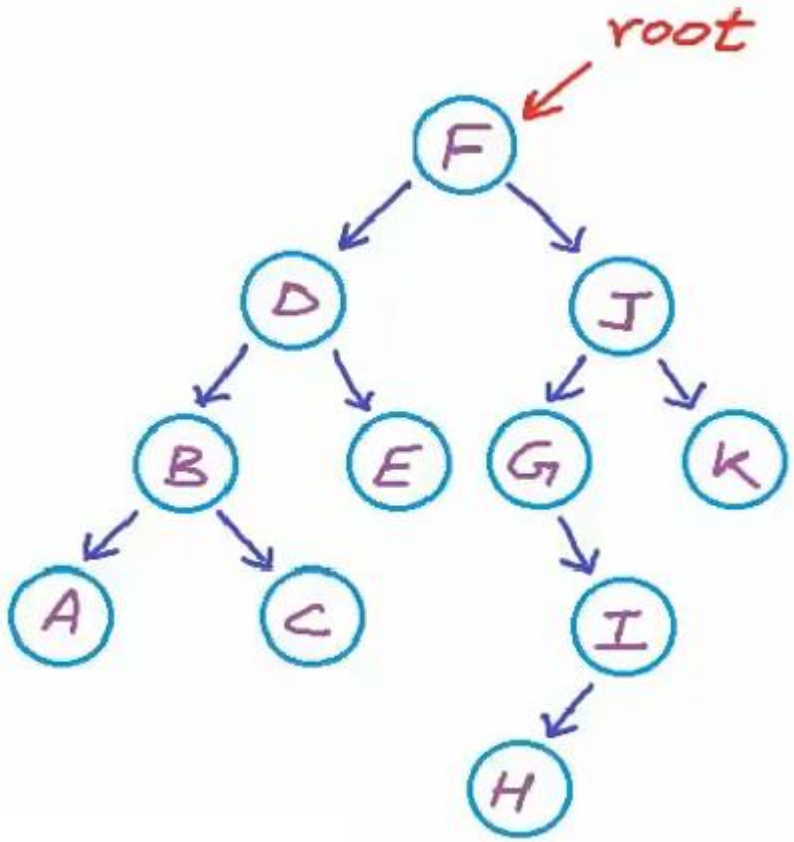
Binary Tree Traversal
Depth-First Approach

- 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).



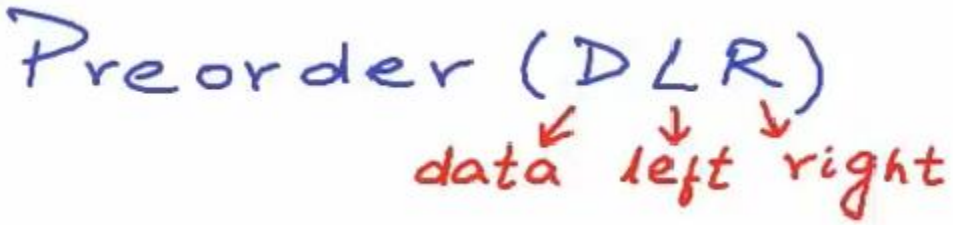
Binary Tree Traversal
Depth-First Approach

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

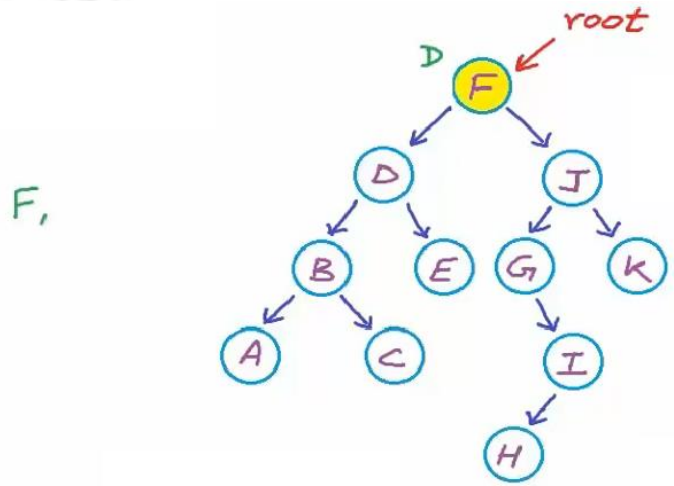


Binary Tree Traversal
Pre-Order Traversal

- We need to start at root node.

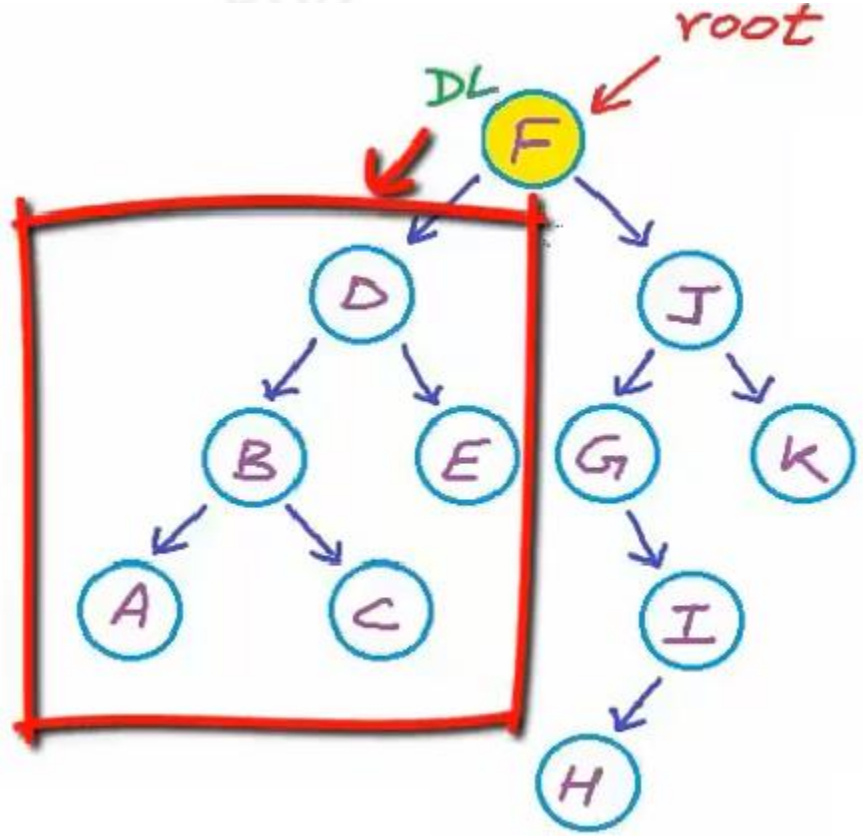


- For each node, we first need to visit the data.



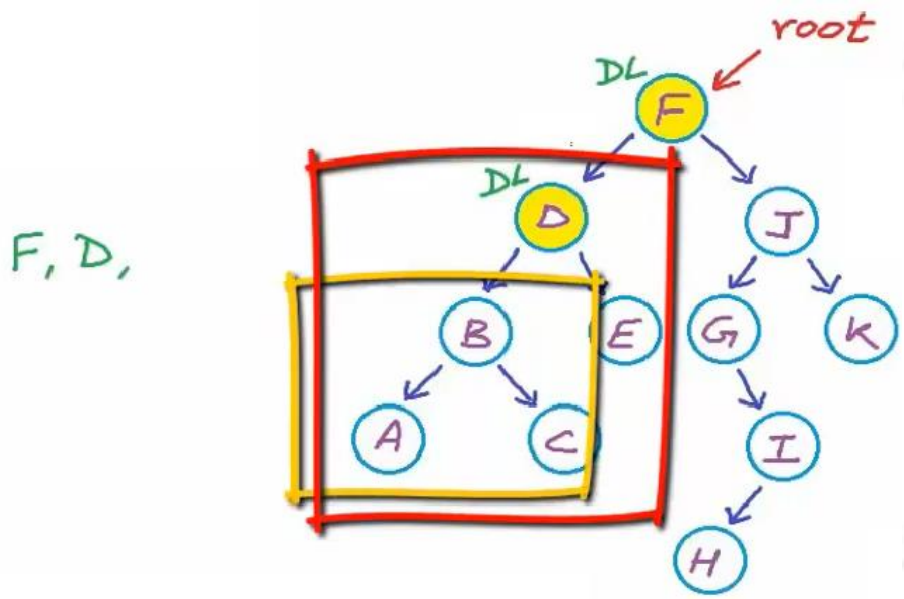
Binary Tree Traversal
Pre-Order Traversal

- 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.



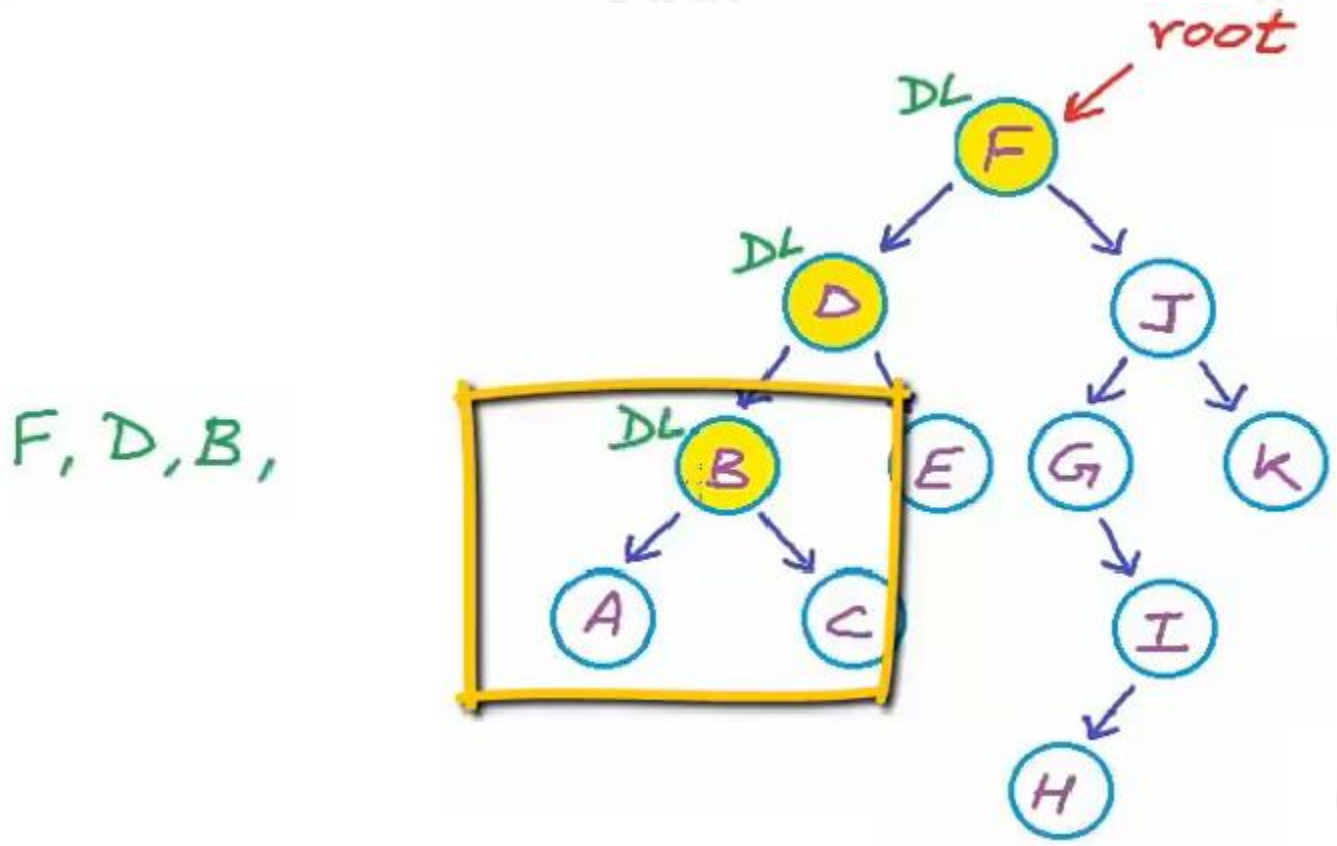
Binary Tree Traversal
Pre-Order Traversal

- 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

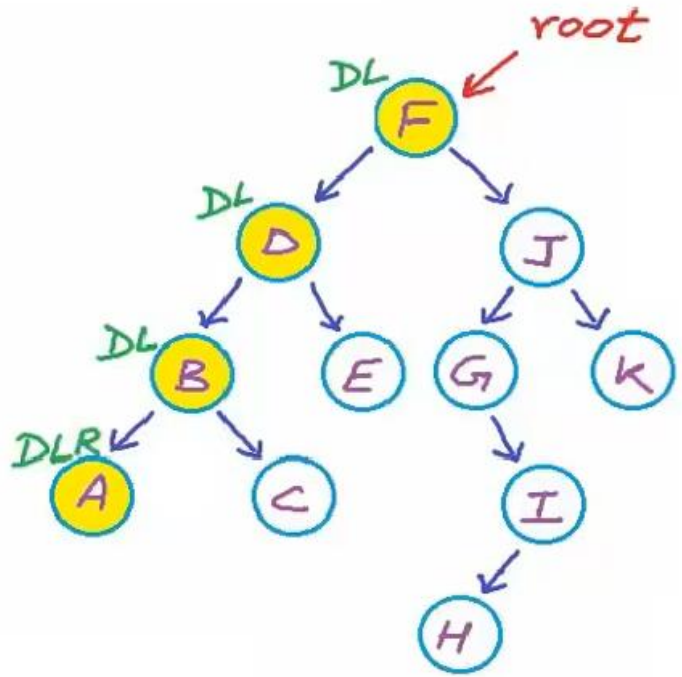
- We are at node (B), we can visit the data and then go left.



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.

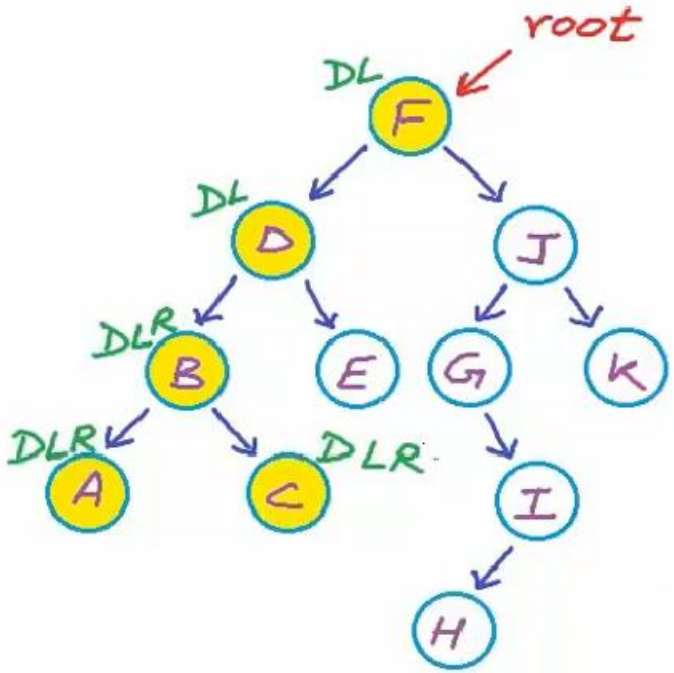
F, D, B, A,



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.

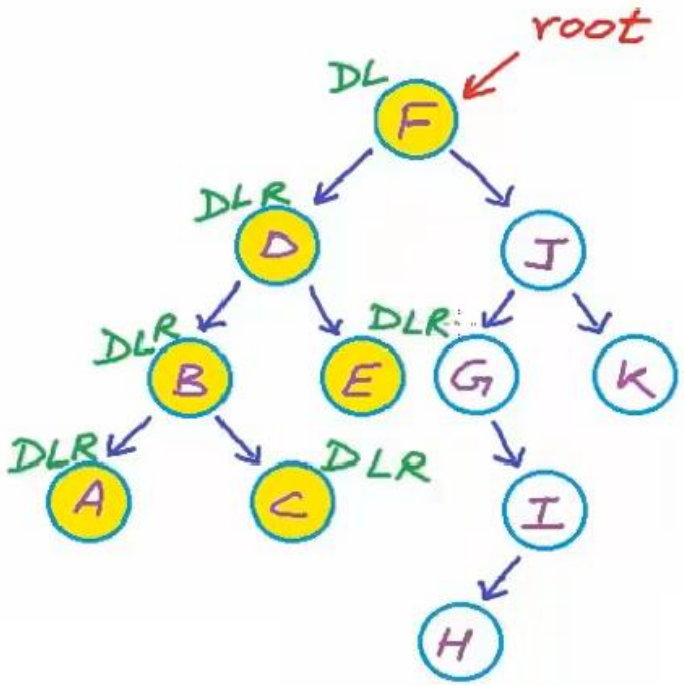
F, D, B, A, C,



Binary Tree Traversal
Pre-Order Traversal

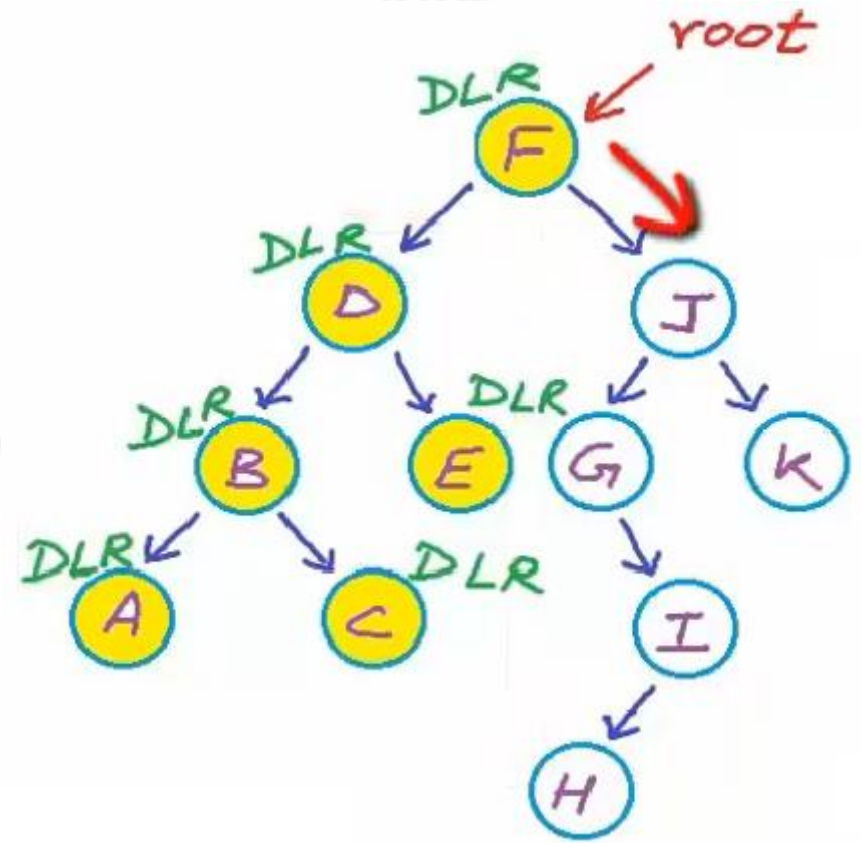
- Now for node (D), left subtree is done so we can go right to node (E).
- We visit the data, then we can go left.
- Left and right of node (E) are null.

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



Binary Tree Traversal
Pre-Order Traversal

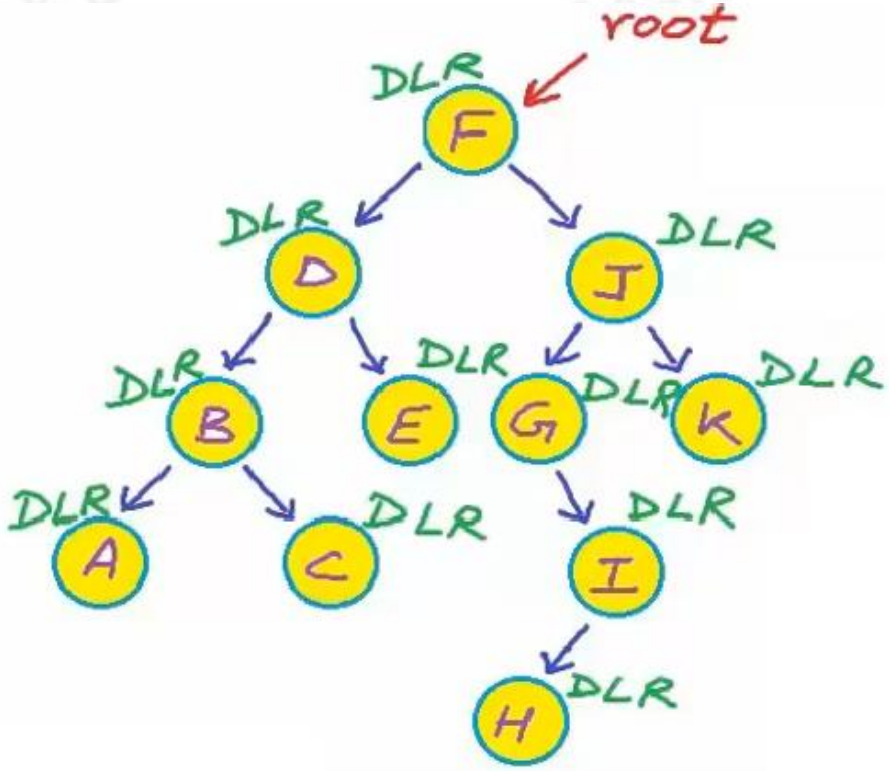
- At this stage, for node (F) the left subtree is visited so we can go right.



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



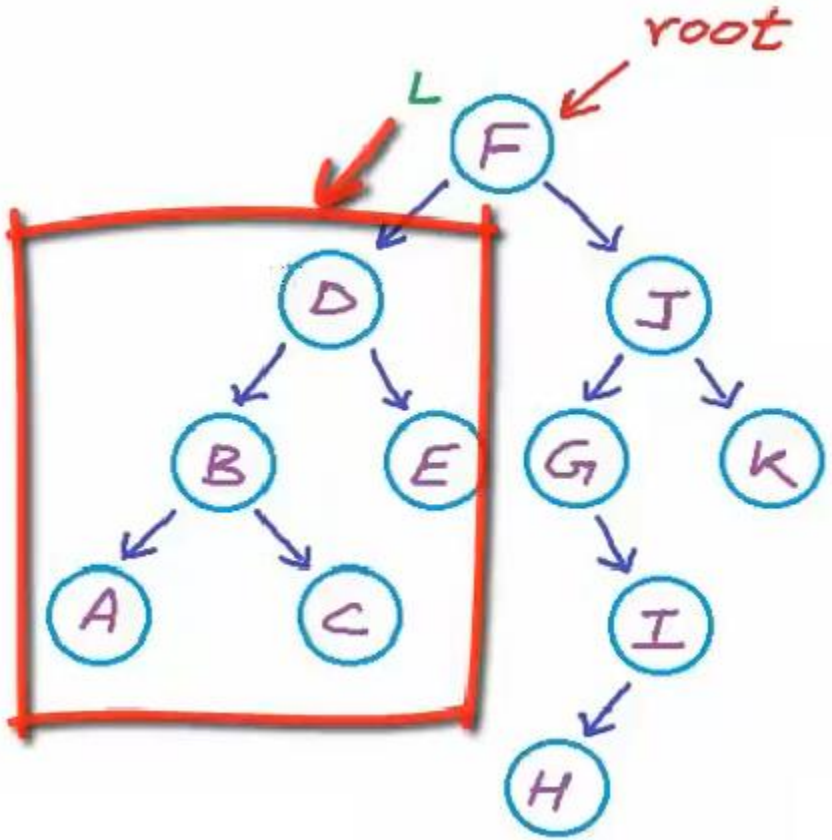
- This is how we can perform pre-order traversal manually.

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.

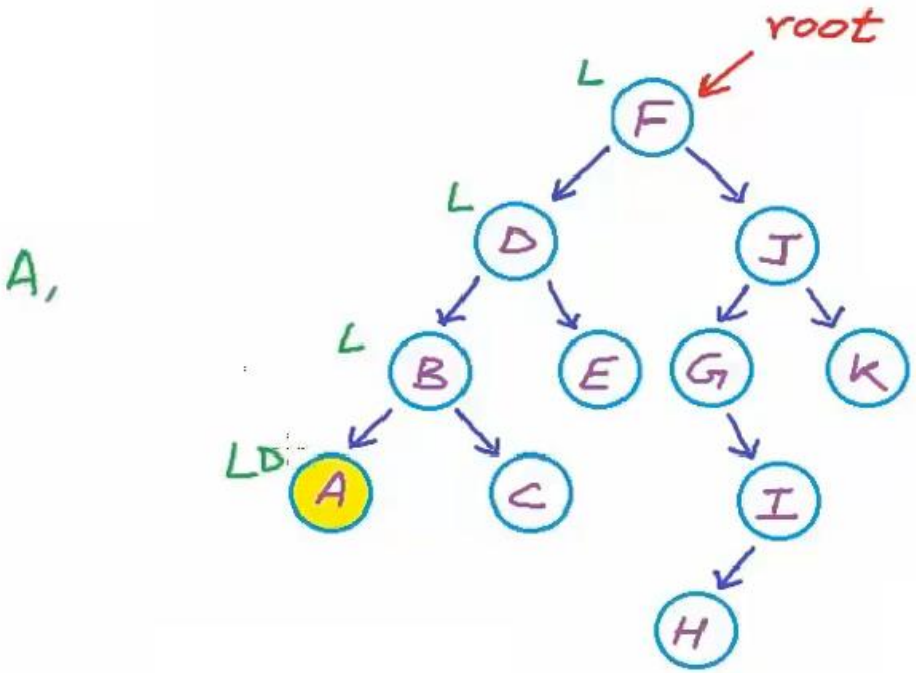
Binary Tree Traversal
In-Order Traversal

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



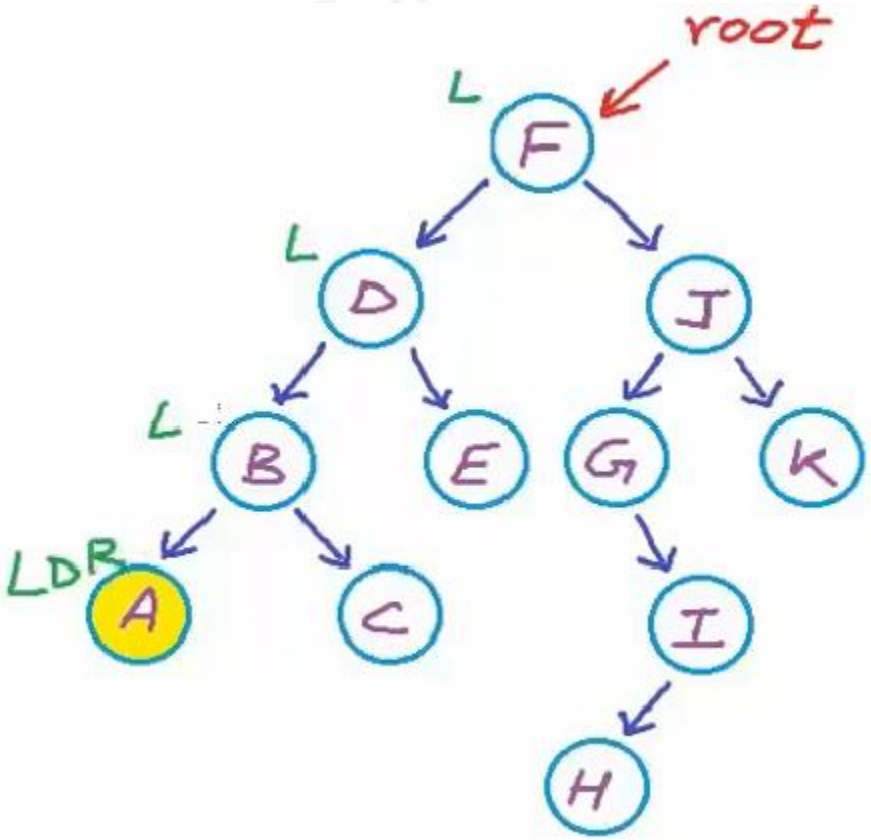
Binary Tree Traversal
In-Order Traversal

- 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

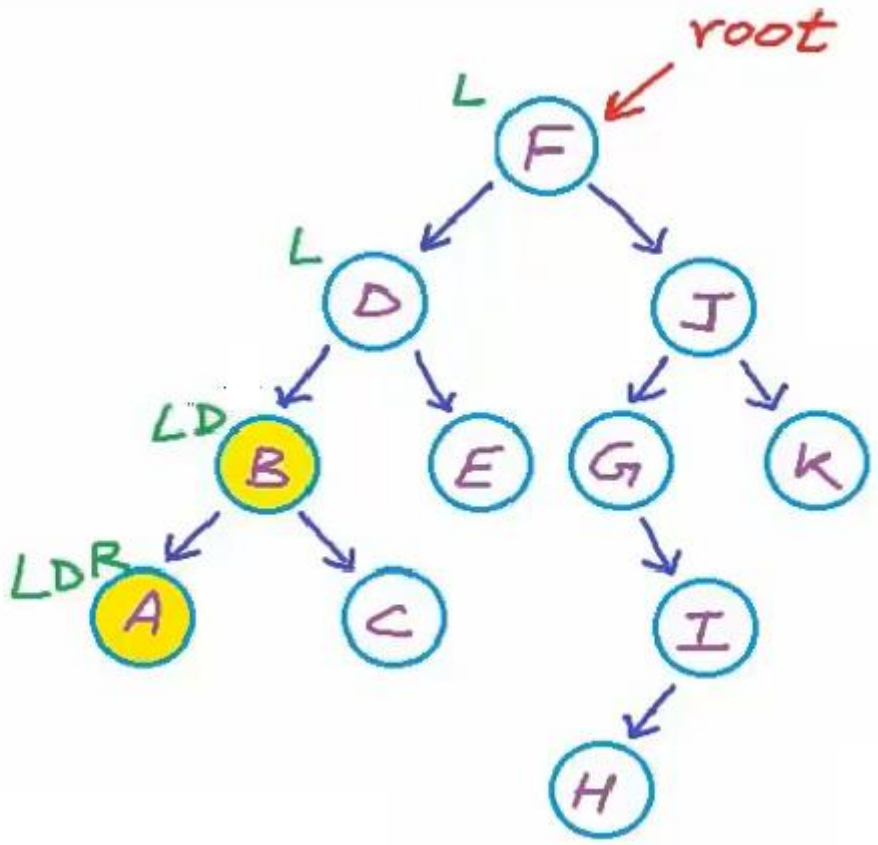
- Now we can go node's (A) right but there is nothing to the right, we can say right is done.



Binary Tree Traversal
In-Order Traversal

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

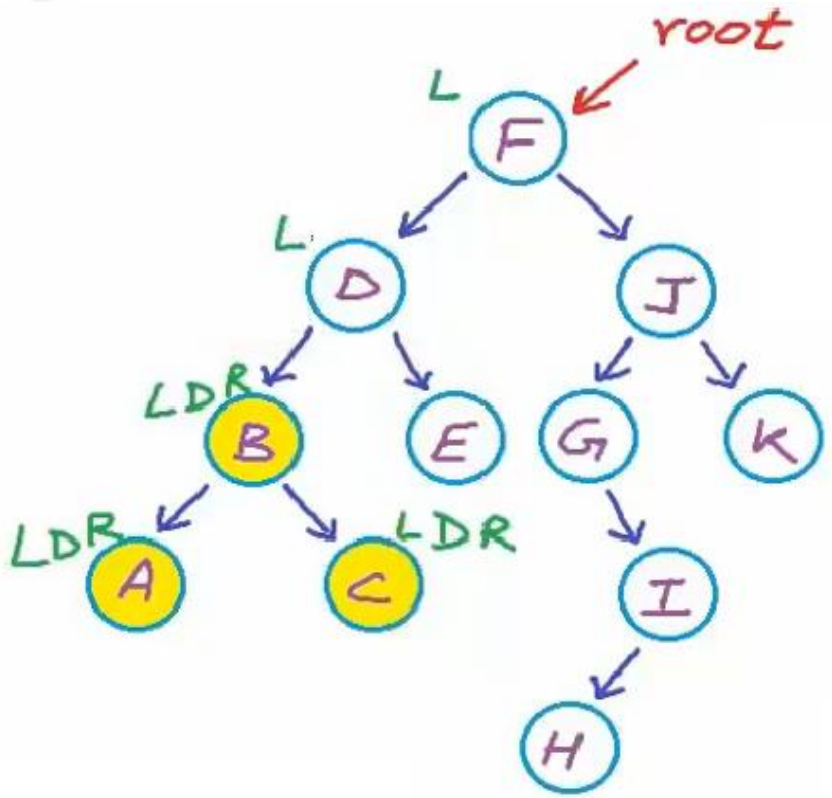
A, B



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.

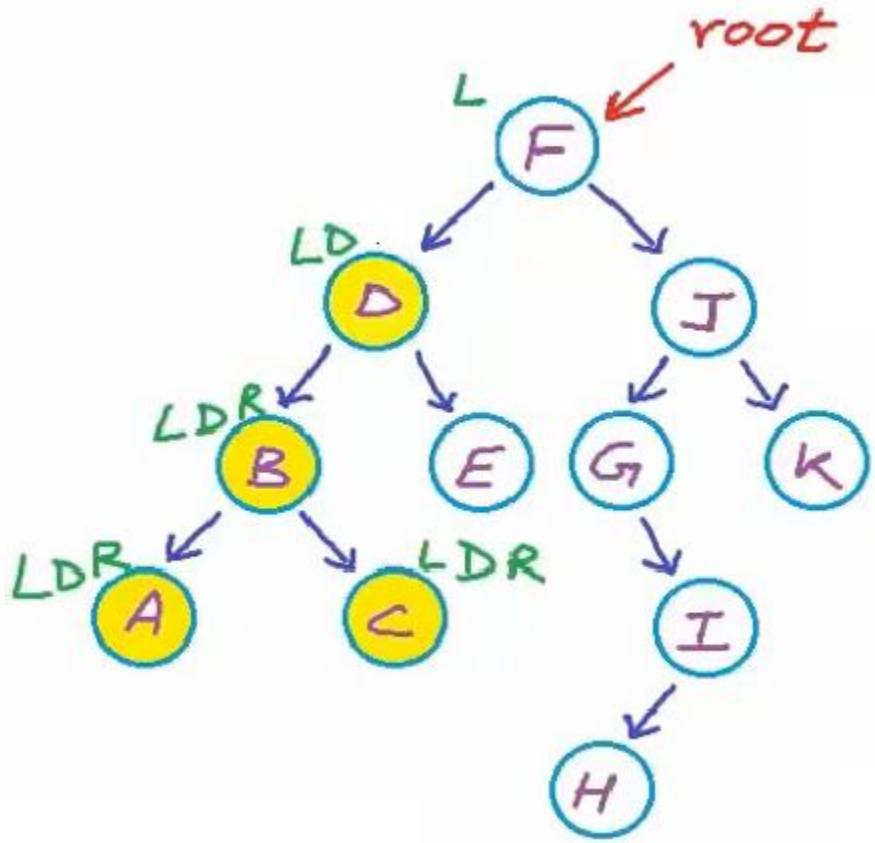
A, B, C



Binary Tree Traversal
In-Order Traversal

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

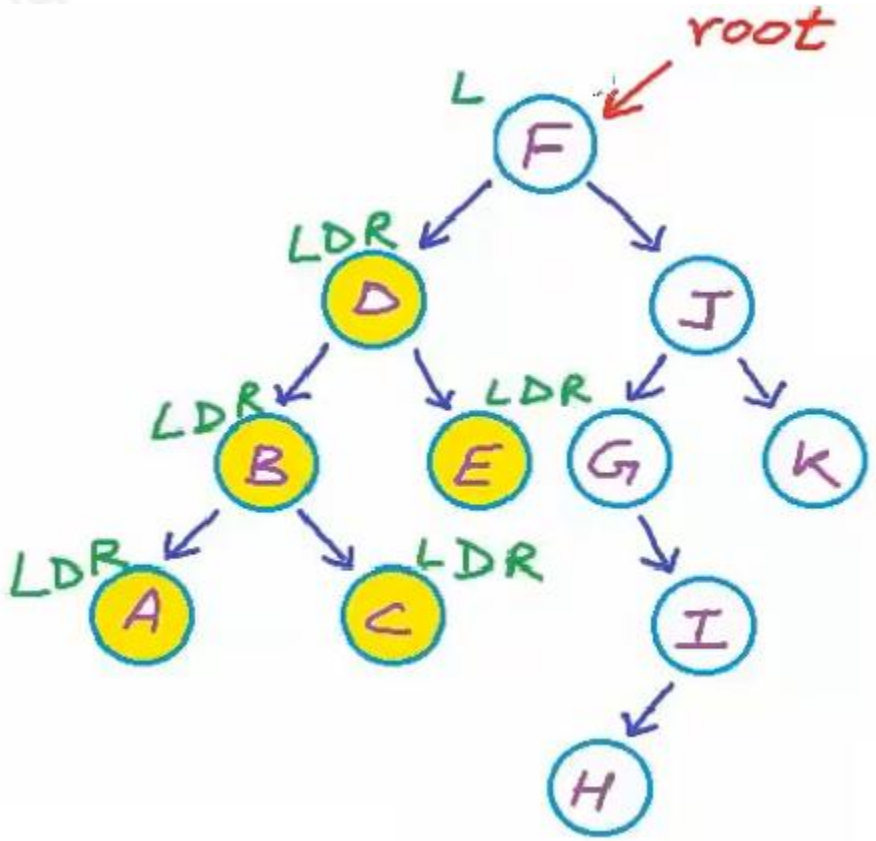
A, B, C, D,



Binary Tree Traversal
In-Order Traversal

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

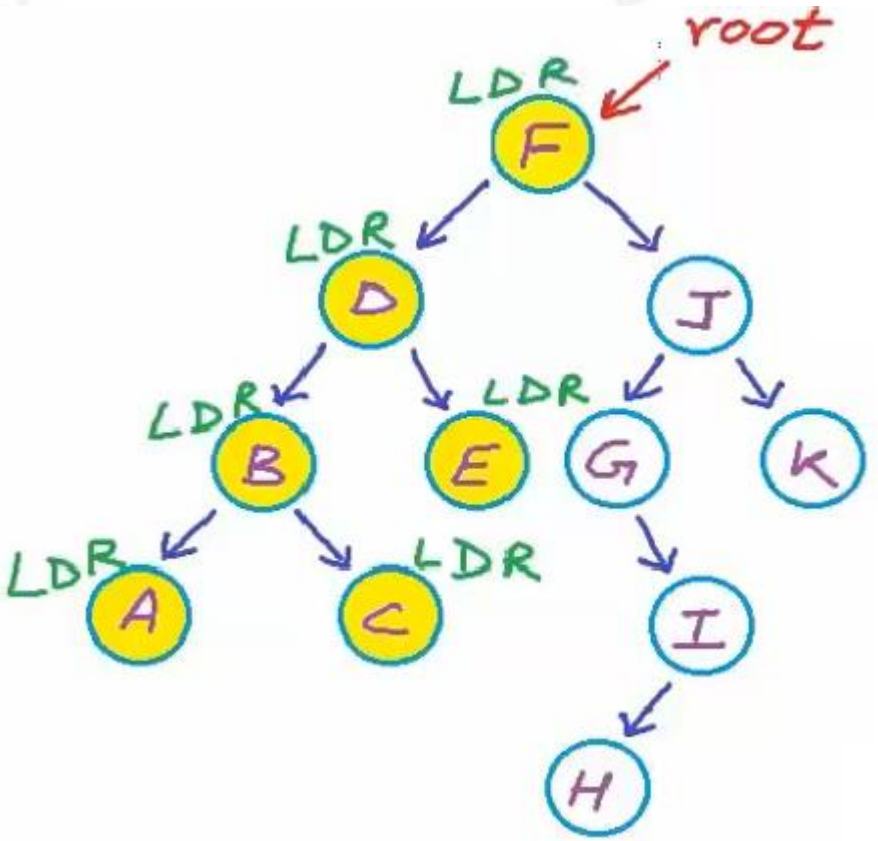
A, B, C, D, E,



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.

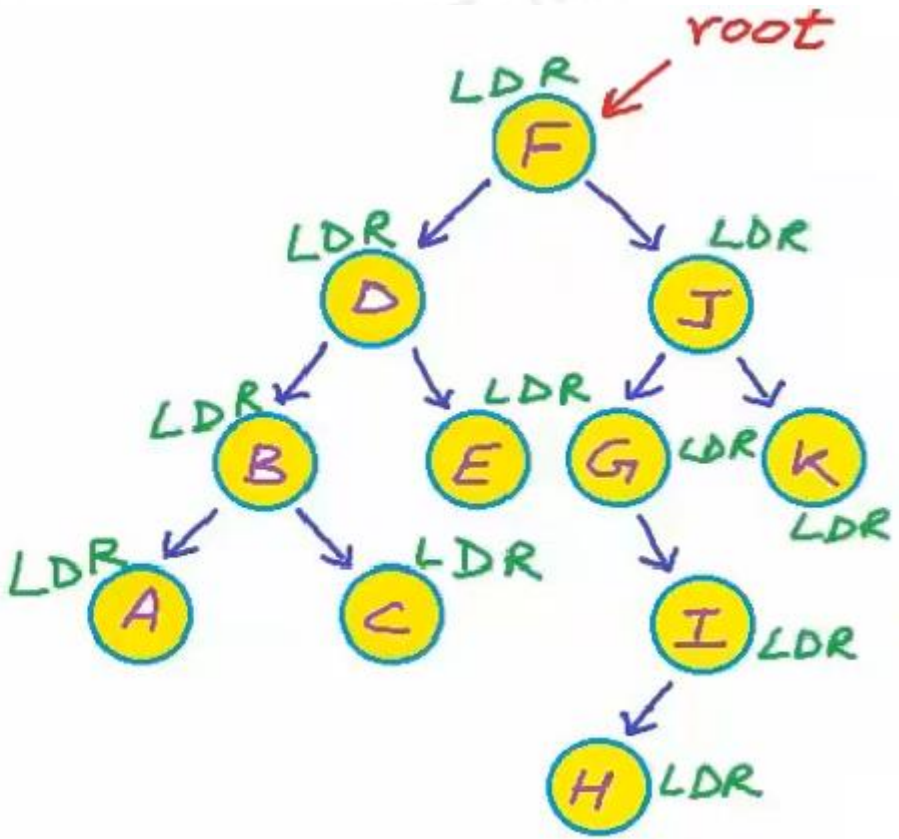
A, B, C, D, E, F,



Binary Tree Traversal
In-Order Traversal

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

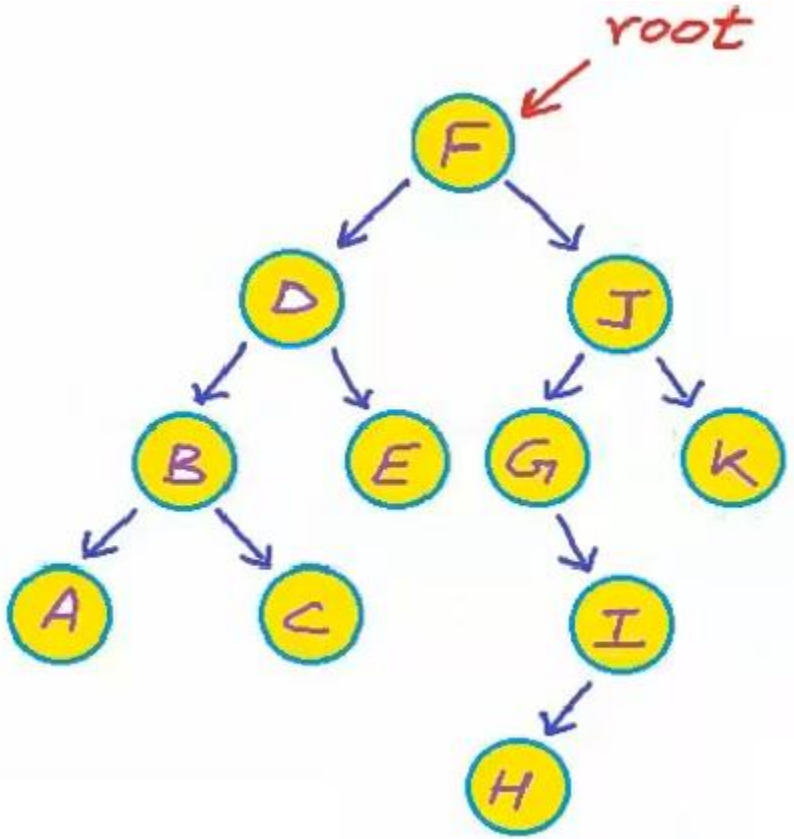
A, B, C, D, E, F, G, H, I, J, K



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???…