



# Unit 5

## Trees

---

**Amir Hussain**  
Cyber Security Trainer  
Computer Science & Engineering





## **Topic – 1**

# Trees





## Definition

A tree is a widely used abstract data type (ADT) that simulates a hierarchical tree structure, with a root value and subtrees of children, represented as a set of linked nodes.

- A tree T is a non linear data structure and set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following
- if T is not empty, T has a special tree called the root that has no parent
- Each node v of T different than the root has a unique parent node w, each node with parent w is a child of w



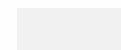
## Terminologies

**Tree:** A hierarchical data structure consisting of nodes, with a single root from which all other nodes descend. Each node contains a value or data and pointers to its child nodes.

**Node:** The basic unit of a tree, containing data and pointers to child nodes.

**Root:** The topmost node of a tree. There is exactly one root in a tree.

**Edge:** A connection between two nodes, representing the parent-child relationship.





## Terminologies

**Parent:** A node that has one or more child nodes.

**Child:** A node that descends from another node (parent).

**Leaf (External Node):** A node that does not have any children.

**Internal Node:** A node that has at least one child.

**Sibling:** Nodes that share the same parent.

**Ancestor:** A node reachable by repeated proceeding from child to parent (e.g., parent, grandparent, etc.).





## Terminologies

**Descendant:** A node reachable by repeated proceeding from parent to child (e.g., child, grandchild, etc.).

**Subtree:** A tree consisting of a node and all its descendants.

**Level (Depth):** The level of a node is the number of edges from the root to the node. The root is at level 0.

**Height:** The height of a node is the number of edges on the longest path from the node to a leaf. The height of a tree is the height of the root.

**Depth:** The depth of a node is the number of edges from the root to that node. It is the same as the level.



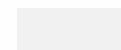


## Terminologies

**Degree:** The degree of a node is the number of children it has. The degree of a tree is the maximum degree of any node in the tree.

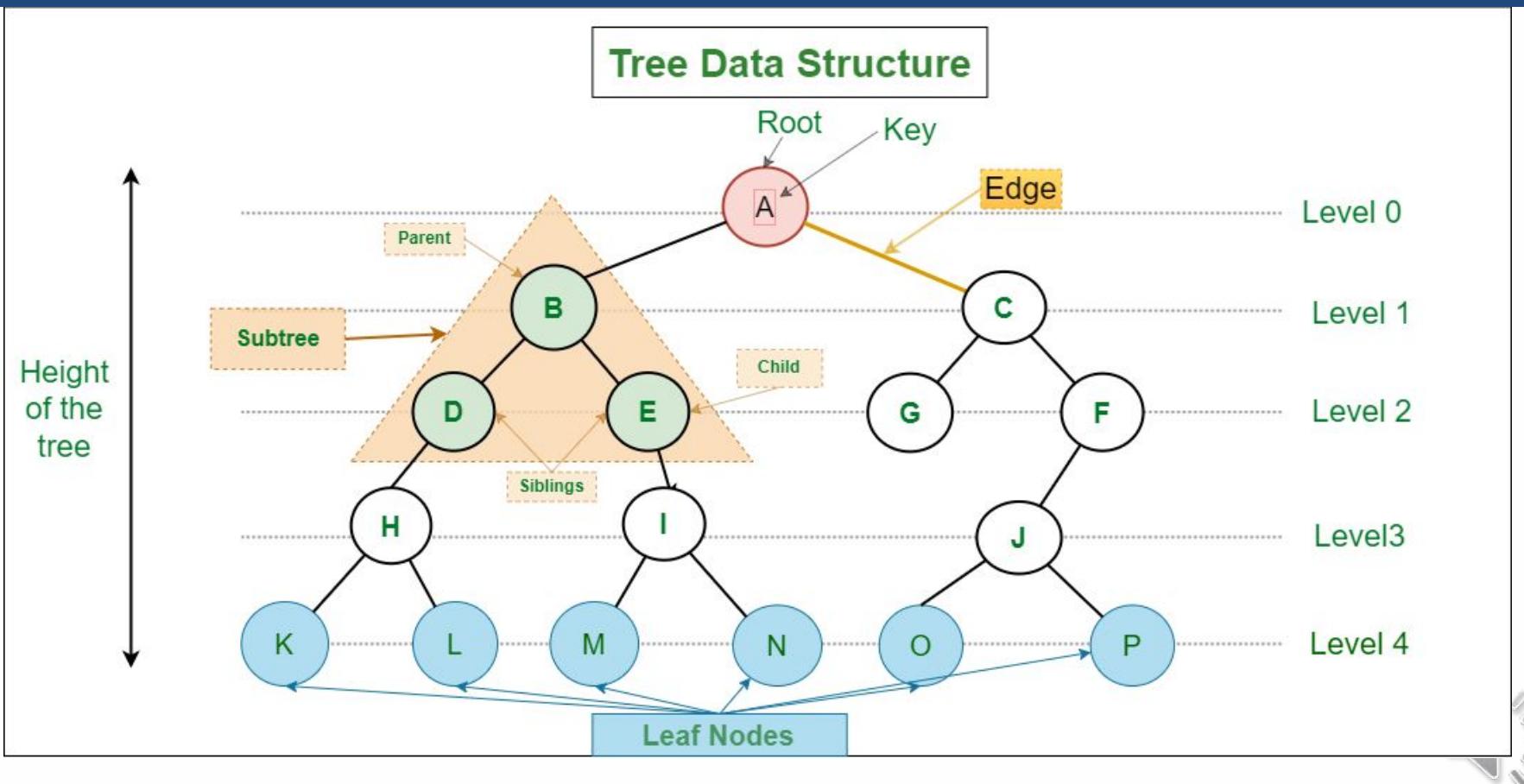
**Path:** A sequence of nodes and edges connecting a node with a descendant.

**Forest:** A collection of disjoint trees.



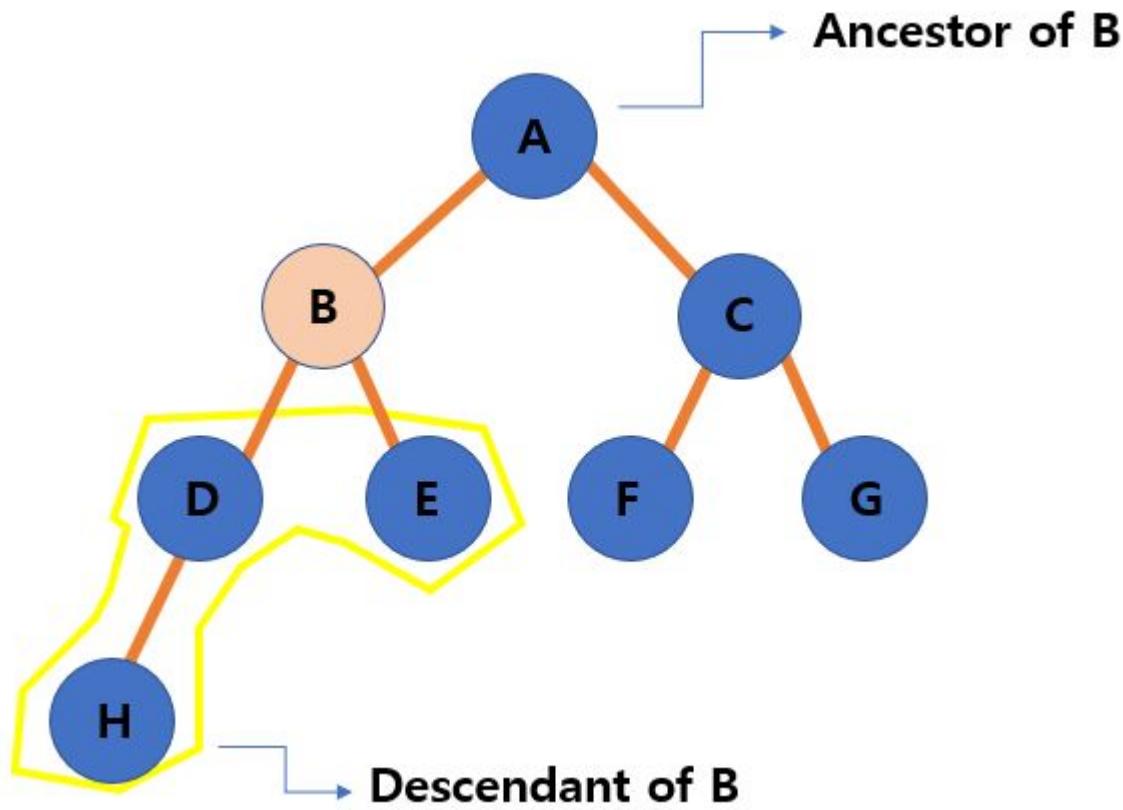


# Terminologies





# Terminologies

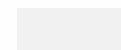




## Terminologies

Whatever the implementation of a tree is, its interface is the following

root()  
size()  
isEmpty()  
parent(v)  
children(v)  
isInternal(v)  
isExternal(v)  
isRoot()





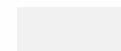
## Types of Tree

**Binary Tree:** A tree in which each node has at most two children, referred to as the left child and the right child.

**Full Binary Tree:** A binary tree where every node other than the leaves has two children or 0.

**Complete Binary Tree:** A binary tree in which all levels are fully filled except possibly the last level, which is filled from left to right. ACBT

**Perfect Binary Tree:** A binary tree where all internal nodes have two children, and all leaves are at the same level.





## Types of Tree

**Balanced Binary Tree:** A binary tree where the height of the left and right subtrees of any node differ by at most one.

**Binary Search Tree (BST):** A binary tree in which for every node, the value of all the nodes in the left subtree is less than the node's value, and the value of all the nodes in the right subtree is greater.

**AVL Tree:** A self-balancing binary search tree where the difference in heights of left and right subtrees cannot be more than one for all nodes.

**Red-Black Tree:** A self-balancing binary search tree where each node has an extra bit for denoting the color of the node (red or black), ensuring the tree remains balanced during insertions and deletions.





# Binary Tree

- **Binary Tree**

If in a directed tree the **out degree of every node is less than or equal to 2** then tree is called binary tree.

- **Strictly Binary Tree**

A strictly binary tree (sometimes proper binary tree or 2-tree or full binary tree) is a tree in **which every node other than the leaves has two children**.

The **depth** of a node is the number of edges present in path from the root node of a tree to that node.

The **height** of a node is the number of edges present in the longest path connecting that node to a leaf node



# Binary Tree

**Definition:** A binary tree is a tree such that

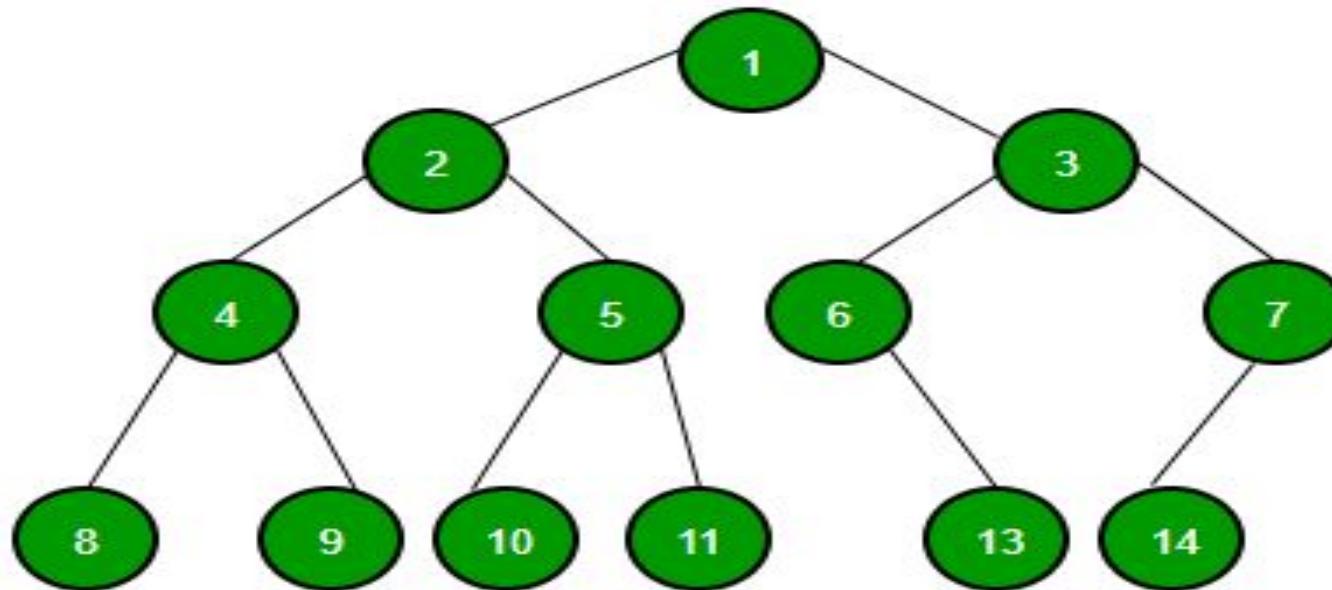
- every node has at most 2 children
- each node is labeled as being either a left child or a right child

**In a binary tree**

- level 0 has  $\leq 1$  node
- level 1 has  $\leq 2$  nodes
- level 2 has  $\leq 4$  nodes
- ...
- level  $i$  has  $\leq 2^i$  nodes



# Binary Tree





# Types of Binary Tree

**Full Binary Tree**

**Degenerate Binary Tree**

**Skewed Binary Trees**

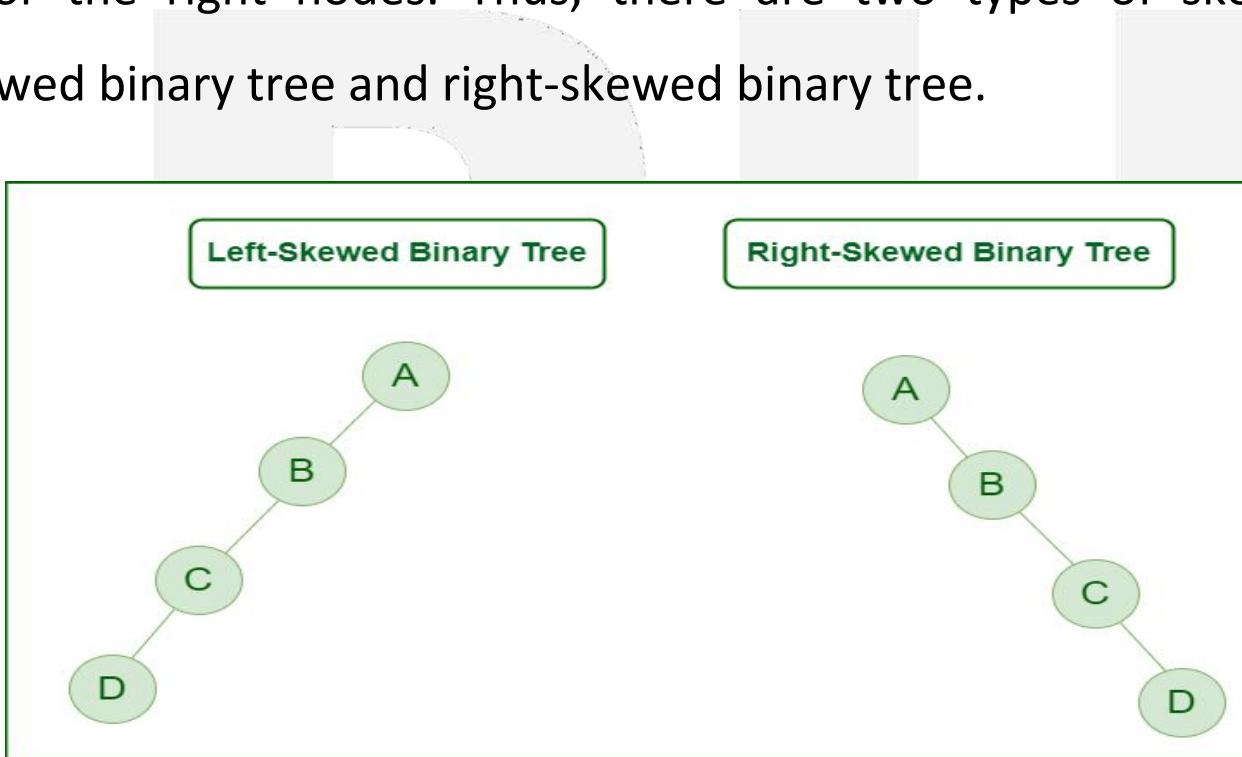
A Binary Tree is a full binary tree if every node has 0 or 2 children.

A Tree where every internal node has one child. Such trees are performance-wise same as linked list.



## Types of Binary Tree

A skewed binary tree is a tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.





## Properties of Binary Tree

**1. The maximum number of nodes at level 'l' of a binary tree is  $2^l$**

Here level is the number of nodes on the path from the root to the node. The level of the root is 0.

For root,  $l = 0$ , number of nodes =  $2^0 = 1$

Assume that the maximum number of nodes on level 'l' is  $2^l$



## Properties of Binary Tree

**2. The Maximum number of nodes in a binary tree of height ‘h’ is  $2^h - 1$ :**

Here the height of a tree is the maximum number of nodes on the root-to-leaf path. The height of a tree with a single node is considered as 1.

If height start from 0 then  $2^{h+1} - 1$ .



## Properties of Binary

**3. A Binary Tree with L leaves has at least  $\lceil \log_2 L \rceil + 1$  levels:**

A Binary tree has the maximum number of leaves (and a minimum number of levels) when all levels are fully filled. Let all leaves be at level  $l$ , then below is valid for the number of leaves  $L$ .

**4. In a Binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children:**



## Properties of Binary Tree

**5. In a non-empty binary tree, if  $n$  is the total number of nodes and  $e$  is the total number of edges, then  $e = n-1$ :**

Every node in a binary tree has exactly one parent with the exception of the root node. So if  $n$  is the total number of nodes then  $n-1$  nodes have exactly one parent. There is only one edge between any child and its parent. So the total number of edges is  $n-1$ .



# Array and linked Representation of Binary Tree

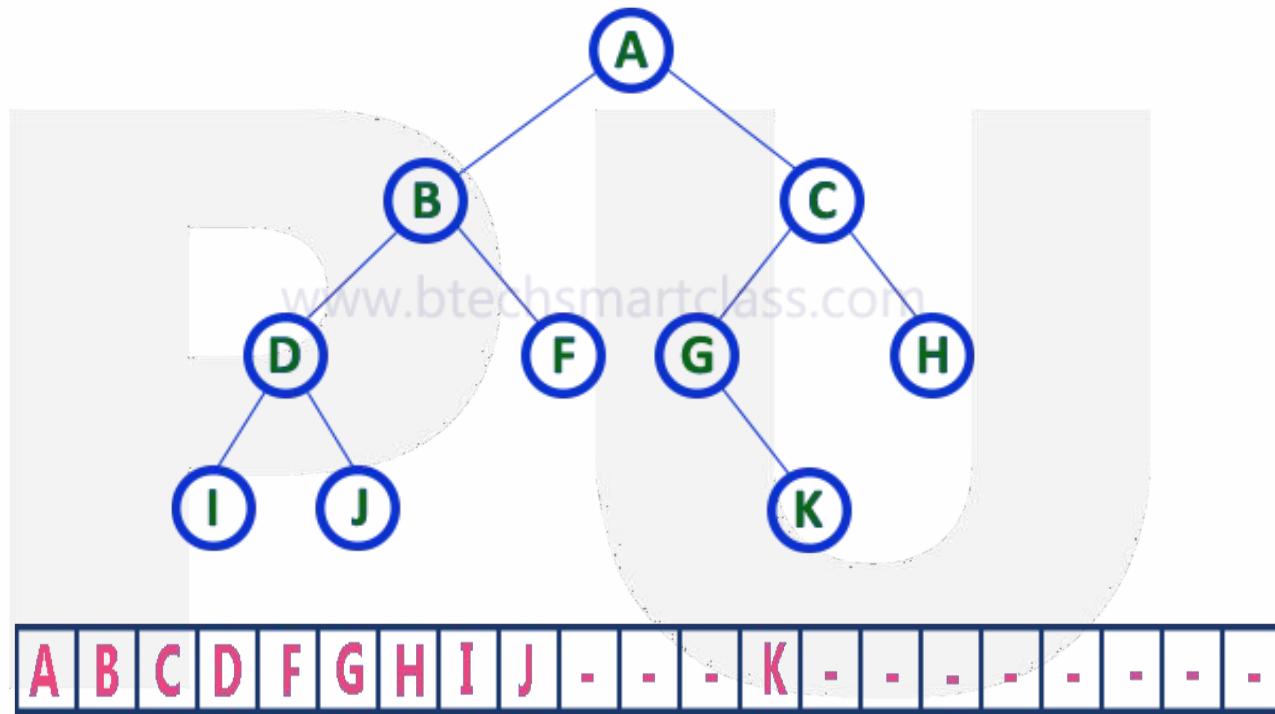
A binary tree data structure is represented using two methods. Those methods are as follows...

**Array Representation**

**Linked List Representation**



# Array and linked Representation of Binary Tree





# Array and linked Representation of Binary Tree

## 1. Array Representation of Binary Tree

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

Consider the above example of a binary tree and it is represented as follows...

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

if node occupied  $a[i]$

then left:  $2*i+1$

right:  $@*i+2$



# Array and linked Representation of Binary Tree

## 2. Linked List Representation of Binary Tree

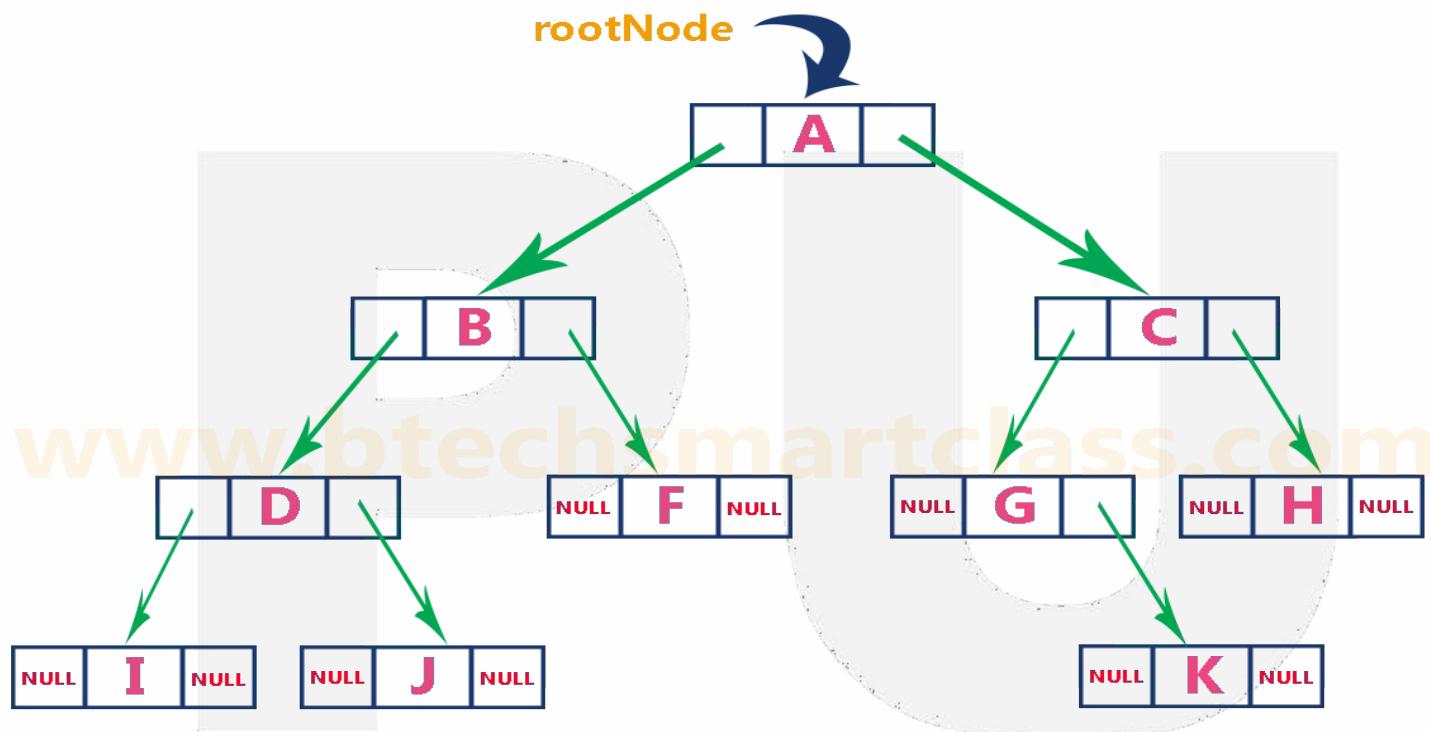
We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...





# Array and linked Representation of Binary Tree

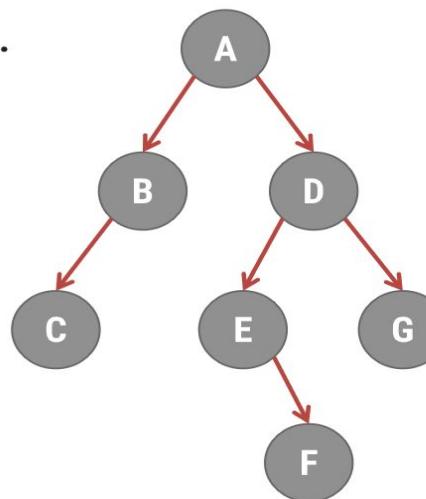




# Binary Tree Traversals

## Tree Traversal

- The most common operations performed on tree structure is that of traversal.
- This is a **procedure by which each node in the tree is processed exactly once** in a systematic manner.
- There are three ways of traversing a binary tree.
  1. Preorder Traversal
  2. Inorder Traversal
  3. Postorder Traversal





## Binary Tree Traversals

- Binary tree traversal is a form of graph traversal and refers to the process of visiting each node in a tree data structure exactly once, in a systematic way.
- Such traversals are classified by the order in which the nodes are visited.
- These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and traversing to the right child node.





# Binary Tree Traversals

- So there are three types of traversal:

**Pre-order, In-order and Post-order.**

## 1. Pre-Order Traversal

It is defined as follows:

1. Process the root node,
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

When a particular subtree is empty, it is considered to be fully traversed.





# Binary Tree Traversals

## Algorithm

Function Rpreorder(T)

Step::1 [process the root node]

    if  $T \neq \text{NULL}$

        Write(DATA (T))

    Else

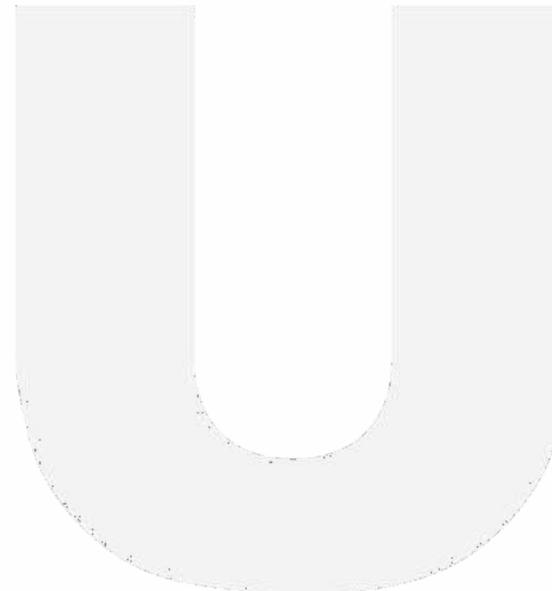
        Write('EMPTY TREE')

    Return.

Step::2 [process the left subtree]

    if  $\text{LPTR}(T) \neq \text{NULL}$

        RPREODER(LPTR(T))



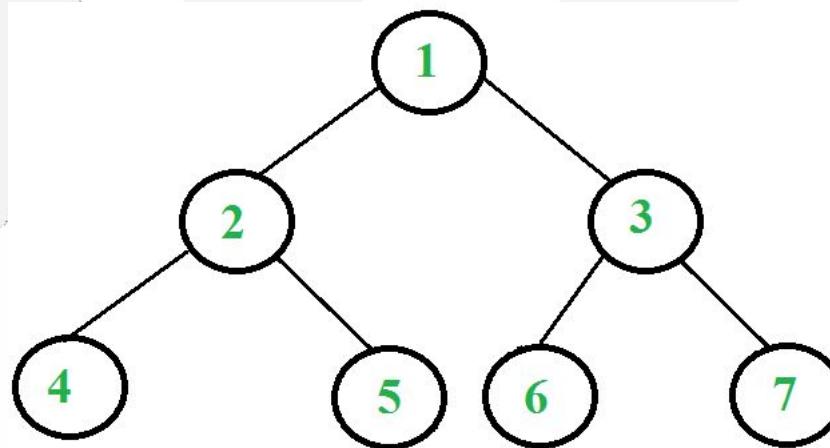


## Binary Tree Traversals

Step::3 [process the right subtree]  
if  $\text{RPTR}(T) \neq \text{NULL}$

$\text{RPREODER}(\text{RPTR}(T))$

Step::4 [FINISH]  
return.



**Pre-order : 1, 2 , 4 , 5 , 3 , 6 , 7**





## Binary Tree Traversals

### 2. Inorder Traversal is defined as:

- 1) Traverse the left subtree in inorder
- 2) Process the root node.
- 3) Traverse the right subtree in inorder

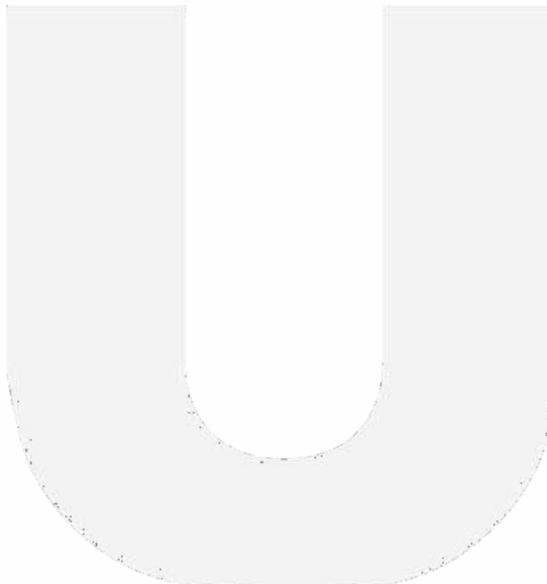
Function Rinorder(T)

Step::1 [Check for empty tree]

if T=NULL

    Write ("EMPTY TREE")

    Return.



Step::2 [Process the left subtree]

if LPTER(T)≠NULL

    RINORDER(LPTR(T))





## Binary Tree Traversals

Step::3 [Process the root node]

    Write (DATA (T))

Step::4 [Process the right subtree]

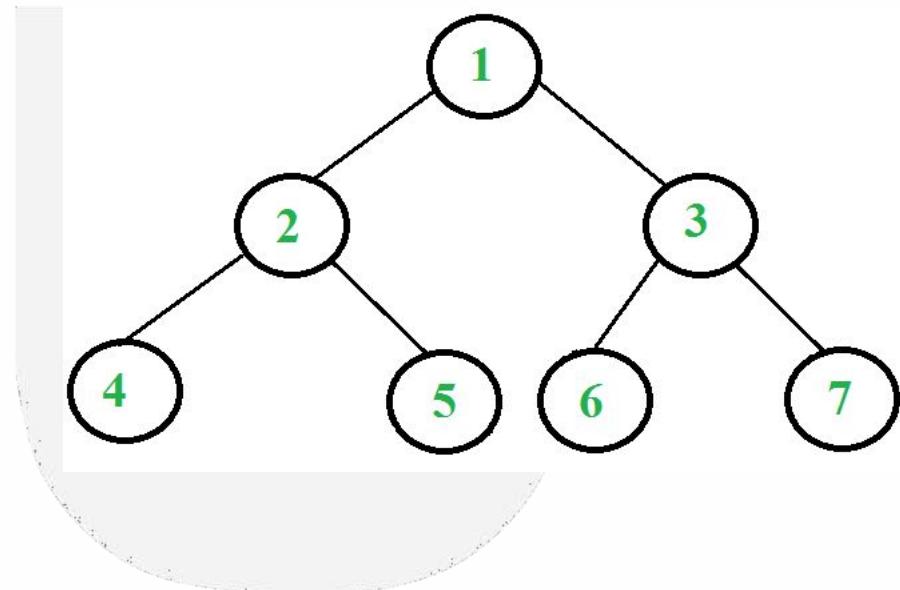
    if RPTER(T)≠NULL

        RINORDER(RPTR(T))

Step::5 [Finish]

    Return.

In-order :4 ,2 , 5 , 1 , 6 , 3 , 7





# Binary Tree Traversals

**POSTORDER DEFINE AS :**

- 1) Traverse the left subtree in postorder
- 2) Traverse the right subtree in postorder
- 3) Process the root node.

Function Rpostorder(T)

Step::1 [Check for an empty tree]

if T=NULL

    Write ("EMPTY TREE")

    Return.

Step::2 [Process the left subtree]

if LPTER(T)≠NULL

    RPOSTORDER(LPTR(T))





## Binary Tree Traversals

Step::3 [Process the right subtree]

if RPTER(T)≠NULL

RPOSTORDER(RPTR(T))

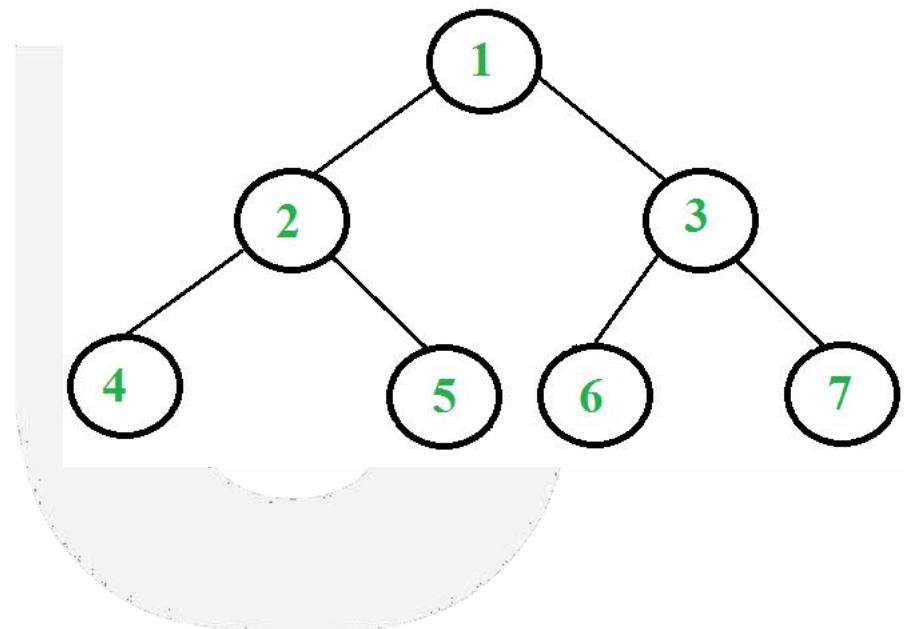
Step::4 [Process the root node]

Write DATA (T)

Step::5 [Finish]

Return

**Post-order : 4 , 5 , 2 , 6 , 7 , 3 , 1**





## Binary Tree: Insertion

To insert a new node into a binary tree using level order traversal, we follow a specific algorithm.

1. Start with the root node of the binary tree.
2. If the root is NULL, create a new node with the given data and set it as the root node.
3. Otherwise, create a queue data structure to hold the nodes during traversal.
4. Enqueue the root node into the queue.





## Binary Tree: Insertion

5. Repeat the following steps until the queue becomes empty:

a. Dequeue a node from the front of the queue.

b. Check if the left child of the dequeued node is NULL.

If so, create a new node with the given data and set it as the left child.

If not, enqueue the left child into the queue.

c. Check if the right child of the dequeued node is NULL.

If so, create a new node with the given data and set it as the right child.

If not, enqueue the right child into the queue.

6. Once the queue is empty, the insertion process is complete.





## Binary Tree: Deletion

Delete a node from Binary Tree by making sure that the tree shrinks from the bottom (i.e. the deleted node is replaced by the bottom-most and rightmost node).

### Algorithm:

1. Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.





## Threaded Binary Tree

The **wasted NULL** links in the binary tree storage representation can be **replaced by threads**

A binary **tree** is **threaded according** to particular **traversal order**. e.g.: Threads for the inorder traversals of tree are pointers to its higher nodes, for this traversal order

### In-Threaded Binary Tree

- If left link of **node P is null**, then this link is **replaced by the address of its predecessor**
- If right link of **node P is null**, then this link is **replaced by the address of its successor**





# Threaded Binary Tree

Because the left or right link of a node can denote **either structural link or a thread**, we must somehow be able to distinguish them

**Method 1:- Represent thread a Negative address**

**Method 2:- To have a separate Boolean flag** for each of left and right pointers, node structure for this is given below

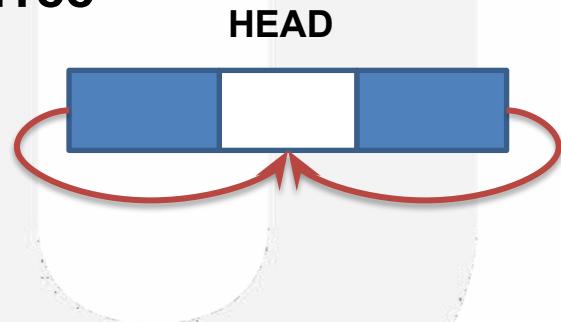


# Threaded Binary Tree



## Typical node of Threaded Binary Tree

- **LTHREAD = true** = Denotes leaf thread link
- **LTHREAD = false** = Denotes leaf structural link
- **RTHREAD = true** = Denotes right threaded link
- **RTHREAD = false** = Denotes right structural link



**Head node is simply another node which serves as the predecessor and successor of first and last tree nodes.**

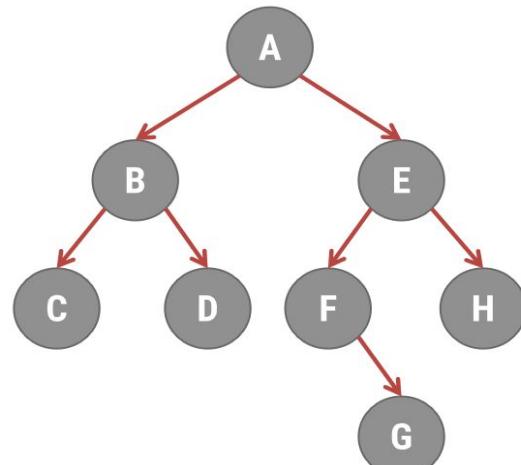
**Tree is attached to the left branch of the head node.**





# Threaded Binary Tree

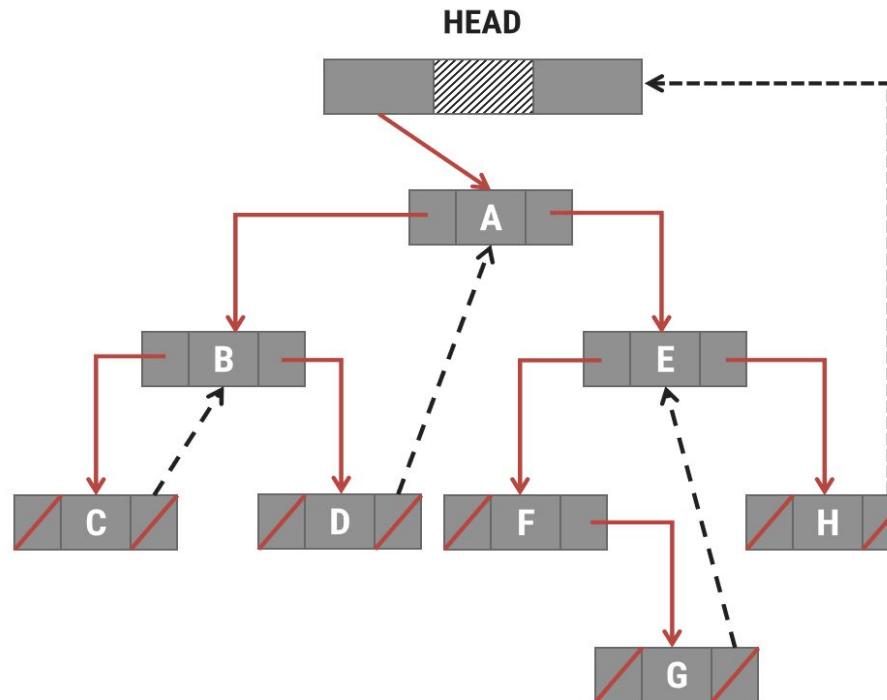
## Threaded Binary Tree



Inorder Traversal

**C B D A F G E H**

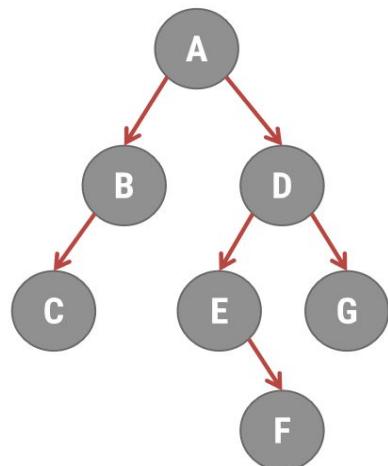
Construct Right In-Threaded Binary Tree of given Tree





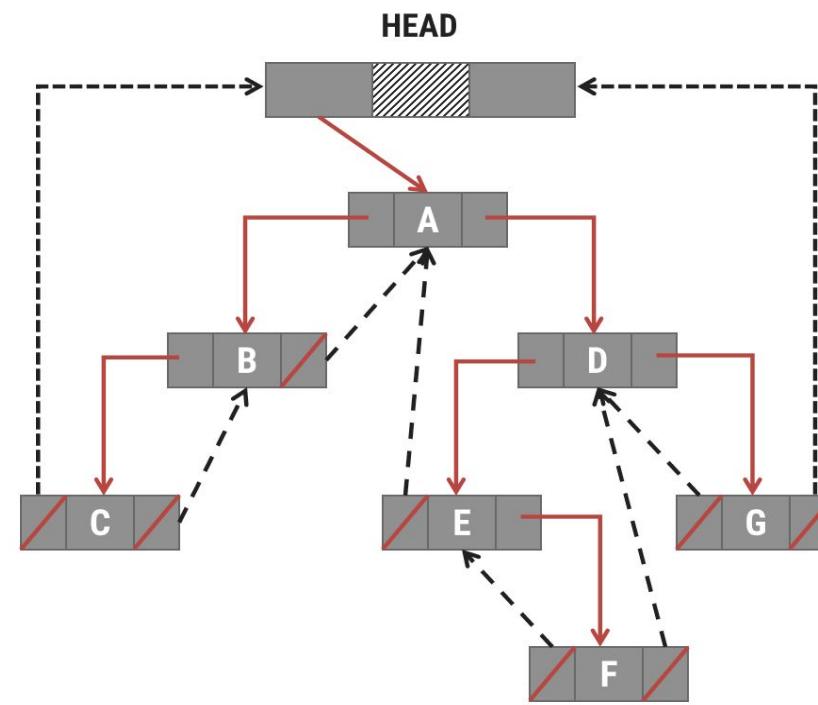
# Double Threaded Binary Tree

## Threaded Binary Tree



Inorder Traversal

C B A E F D G



Fully In-Threaded Binary Tree



## Advantage of Threaded Binary Tree

**Inorder traversal is faster** than unthreaded version as stack is not required.

**Effectively determines** the **predecessor and successor** for inorder traversal, for unthreaded tree this task is more difficult.

**A stack is required** to provide upward pointing information **in binary tree** which **threading provides without stack**.

It is possible to **generate successor or predecessor** of any node **without** having over head of **stack** with the help of threading.





## Disadvantage of Threaded Binary Tree

Threaded trees are **unable to share common sub trees**.

If **Negative addressing is not permitted** in programming language, **two additional fields are required**.

**Insertion** into and **deletion** from threaded binary tree are **more time consuming** because both thread and structural link must be maintained.





## Binary Search Tree

A **binary search tree** is a **binary tree** in which **each node** possessed a key that **satisfy** the **following conditions**

1. All **key** (if any) in **the left sub tree** of the root **precedes the key** in the **root**
2. The **key in the root precedes** all **key** (if any) in the **right sub tree**
3. The **left and right sub trees** of the root are again **search trees**

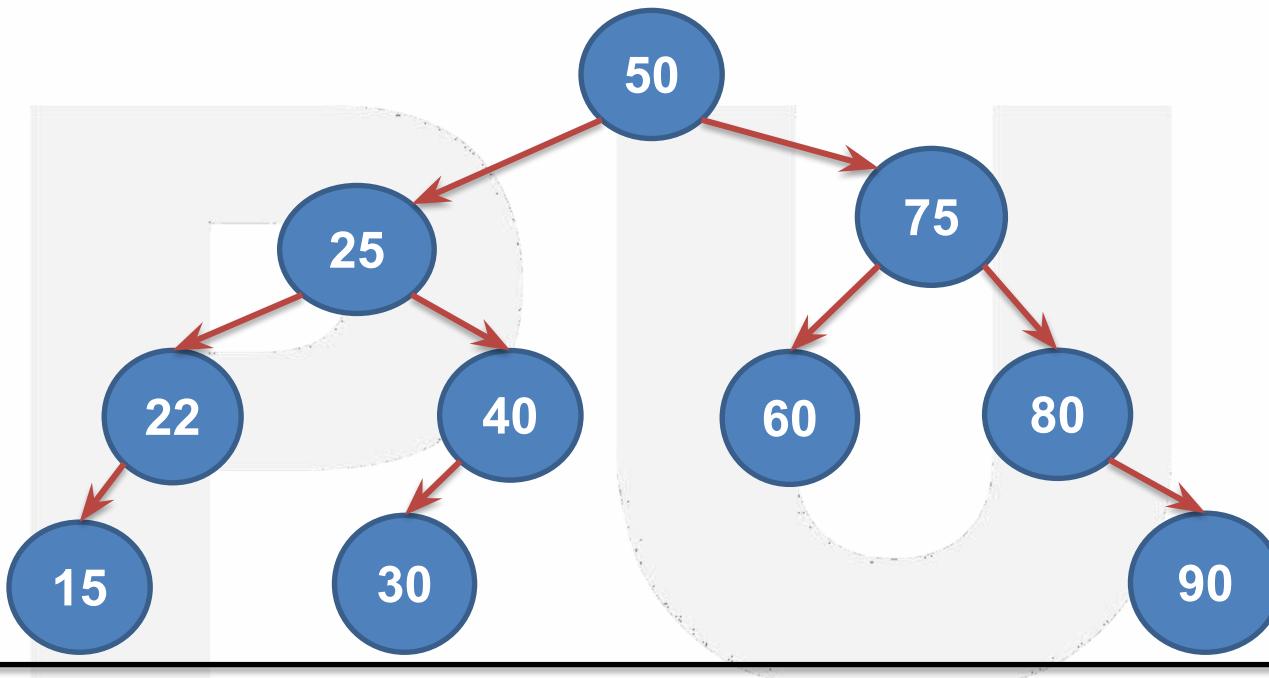
Construct binary search tree for the following data

50 , 25 , 75 , 22 , 40 , 60 , 80 , 90 , 15 , 30





# Binary Search Tree



Construct binary search tree for the following data  
10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5





## Binary Search Tree

- **Search a node in Binary Search Tree**

To search for target value.

We first compare it with the key at root of the tree.

If it is not same, we go to either Left sub tree or Right sub tree as appropriate and repeat the search in sub tree.

If we have **In-Order List** & we want to search for specific node it requires **O(n)** time.

In case of **Binary tree** it requires **O(Log<sub>2</sub>n)** time to search a node.





## Binary Search Tree: Search

- To search in a binary search tree, we need to take care that each left subtree has values below the root and each right subtree has values above the root.

```
If root == NULL  
    return NULL;  
If number == root->data  
    return root->data;  
If number < root->data  
    return search(root->left)  
If number > root->data  
    return search(root->right)
```





## Binary Search Tree: Search

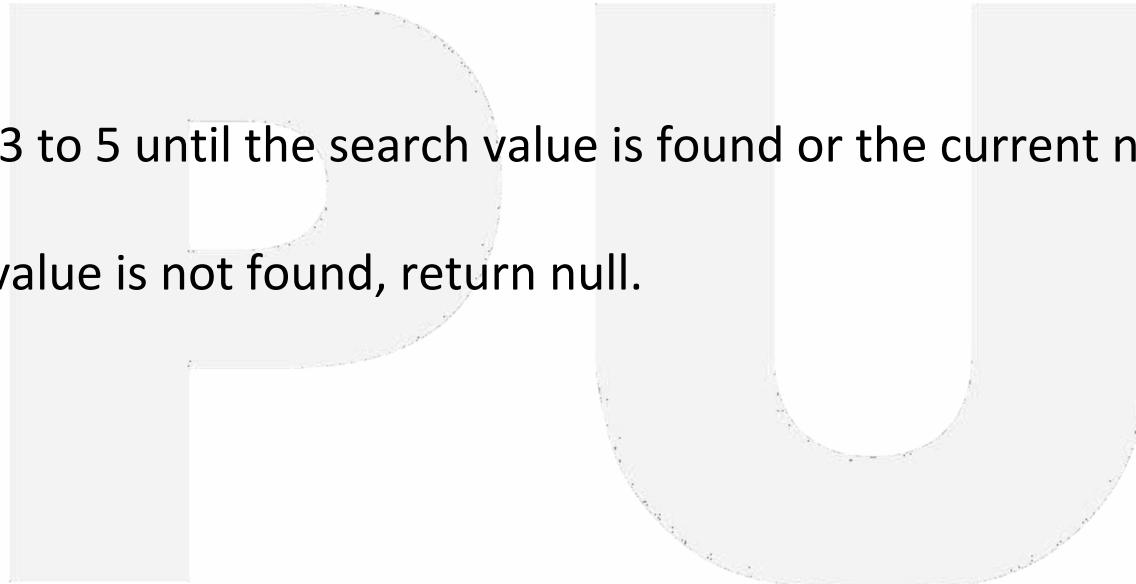
- Start from the root node.
- If the root node is null, the tree is empty, and the search is unsuccessful.
- If the search value is equal to the value of the current node, return the current node.
- If the search value is less than the value of the current node, go to the left child node.





## Binary Search Tree: Search

- If the search value is greater than the value of the current node, go to the right child node.
- Repeat steps 3 to 5 until the search value is found or the current node is null.
- If the search value is not found, return null.





## Binary Search Tree: Insertion

To insert an element in a BST, we need to maintain the rule that the left subtree is lesser than the root, and the right subtree is larger than the root.

```
If node == NULL  
    return createNode(data)  
if (data < node->data)  
    node->left = insert(node->left, data);  
else if (data > node->data)  
    node->right = insert(node->right, data);  
return node;
```





## Binary Search Tree: Insertion

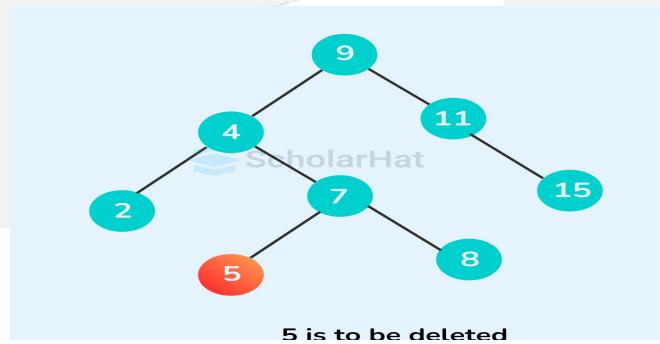
- Start at the root of the tree.
- If the tree is empty, create a new node and make it the root.
- If the value of the new node is less than the value of the current node, move to the left child of the current node.
- If the value of the new node is greater than the value of the current node, move to the right child of the current node.
- Repeat steps 3 and 4 until a leaf node is reached.





## Binary Search Tree: Deletion

- To delete an element from a BST, the user first searches for the element using the search operation.
- If the element is found, there are three cases to consider:
  - The node to be deleted has no children: In this case, simply remove the node from the tree.





## Binary Search Tree: Deletion

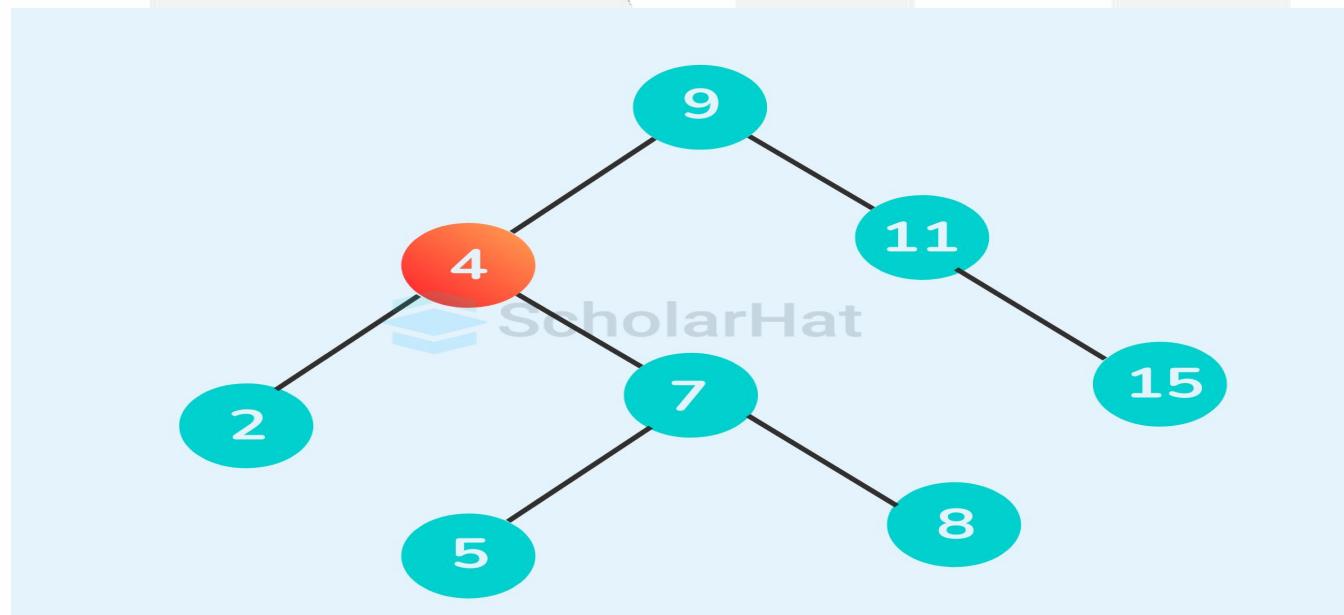
2. The node to be deleted has only one child: In this case, remove the child node from its original position and replace the node to be deleted with its child node





## Binary Search Tree: Deletion

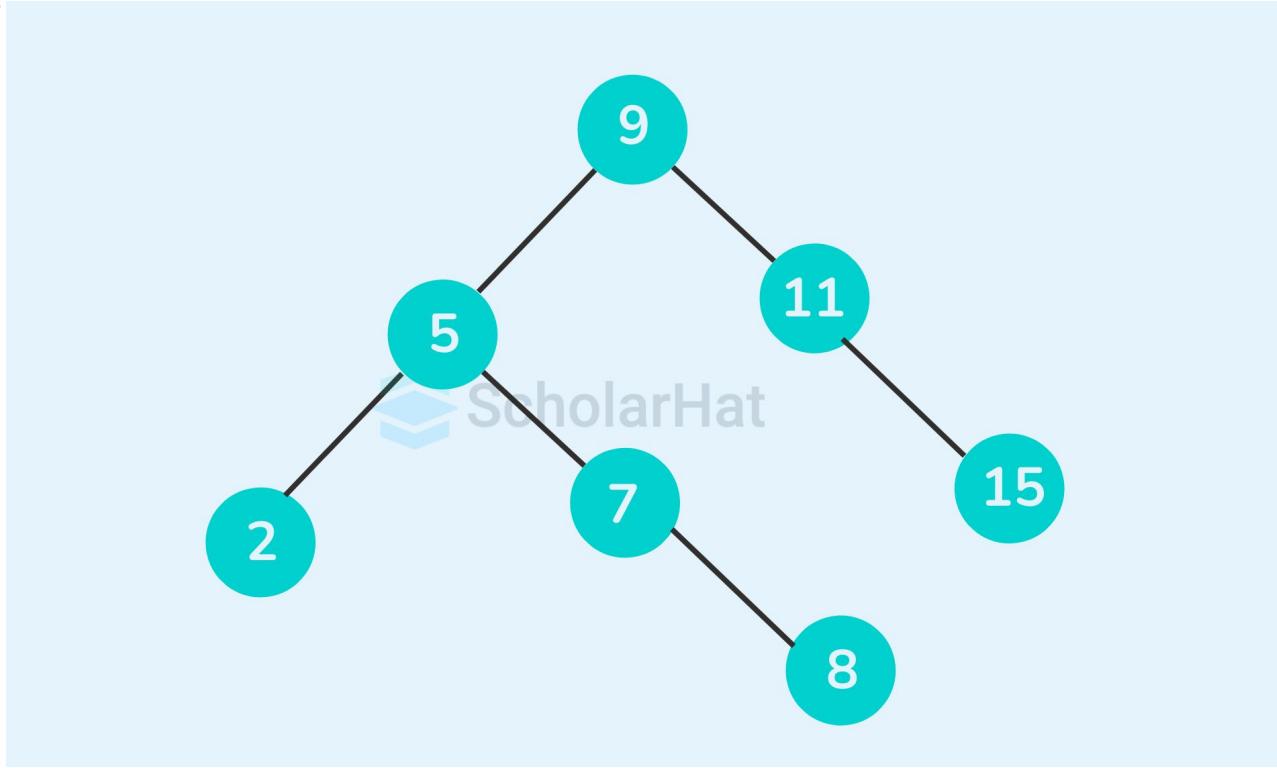
3. The node to be deleted has two children: In this case, get the in-order successor of that node, replace the node with the inorder successor, and then remove the inorder successor from its original position.





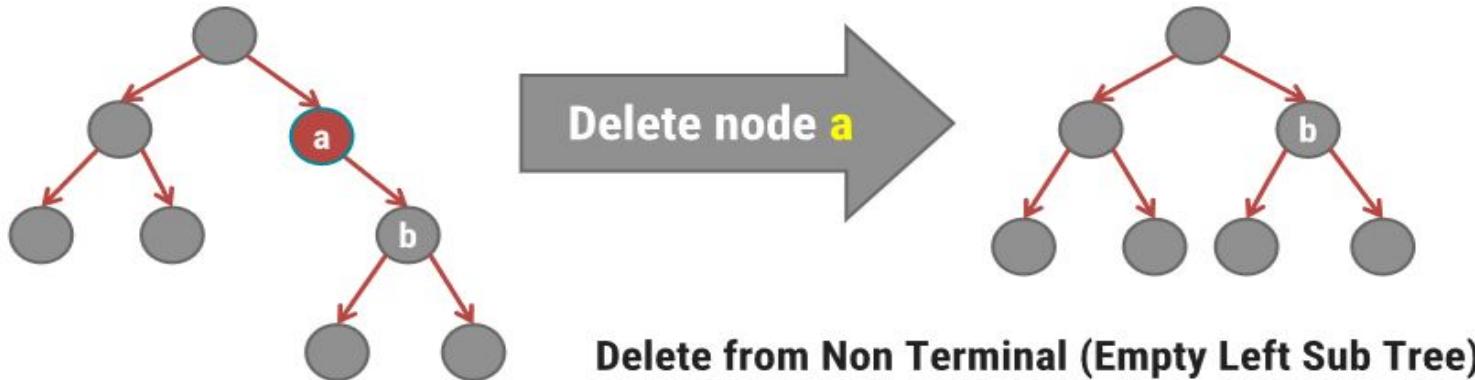
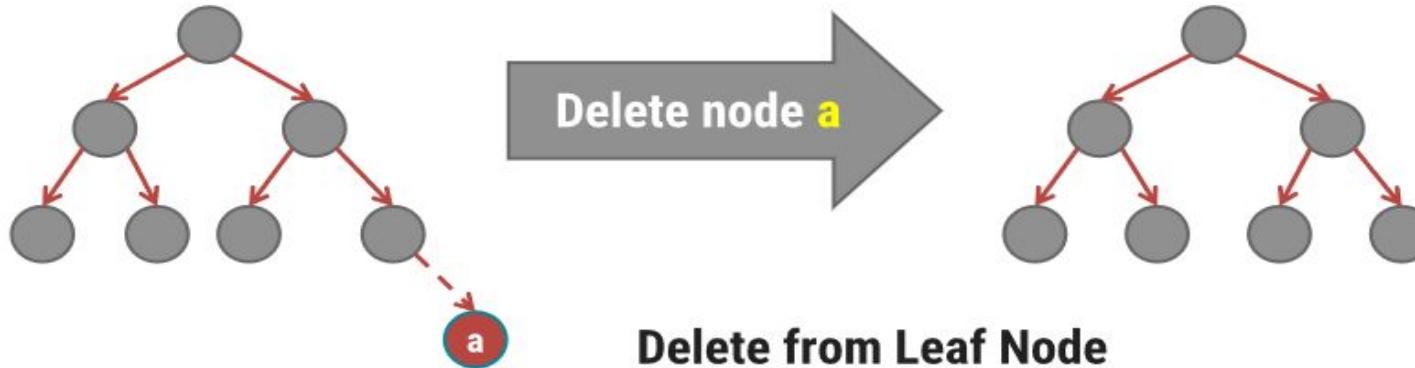
## Binary Search Tree: Deletion

Final Tree:



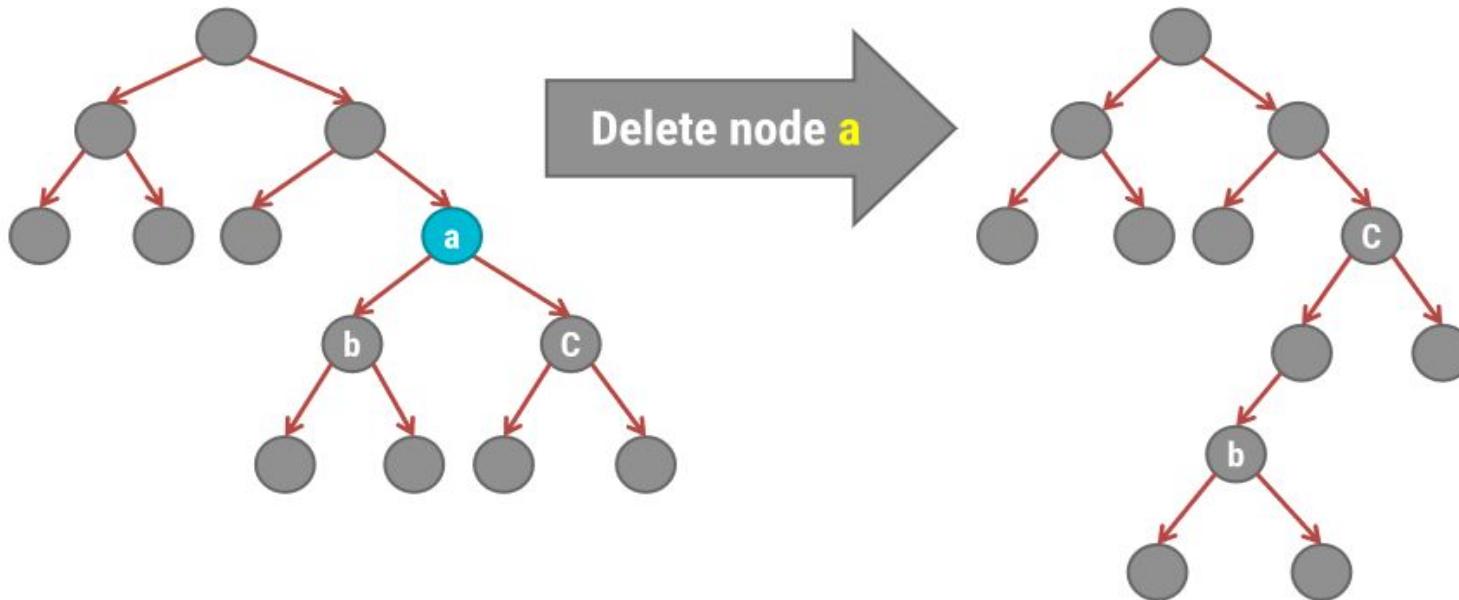


# Binary Search Tree





# Binary Search Tree



Delete from Non Terminal (Neither Sub Tree is Empty)



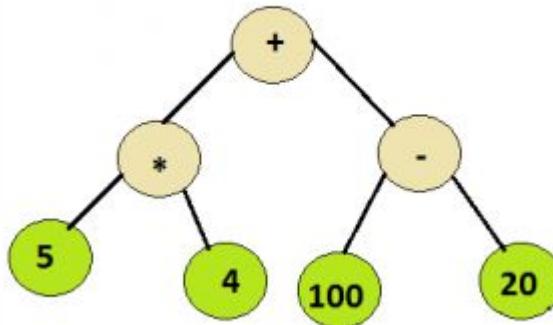


## Application of Trees-Evaluation of Expression

Using trees to evaluate expressions is a common application of data structures in computer science.

The most common type of tree used for this purpose is the expression tree.

An expression tree is a binary tree where each internal node represents an operator, and each leaf node represents an operand





## Application of Trees-Evaluation of Expression

As all the operators in the tree are binary, hence each node will have either 0 or 2 children.

As it can be inferred from the examples above, all the integer values would appear at the leaf nodes, while the interior nodes represent the operators.

Therefore we can do inorder traversal of the binary tree and evaluate the expression as we move ahead.





## Application of Trees-Evaluation of Expression

To evaluate the syntax tree, a recursive approach can be followed  
Algorithm:

Let t be the syntax tree  
If t is not null then  
    If t.info is operand then  
        Return t.info  
    Else  
        A = solve(t.left)  
        B = solve(t.right)  
        return A operator B, where operator is the info contained in t



- x **DIGITAL LEARNING CONTENT**



**Parul® University**



[www.paruluniversity.ac.in](http://www.paruluniversity.ac.in)

n

