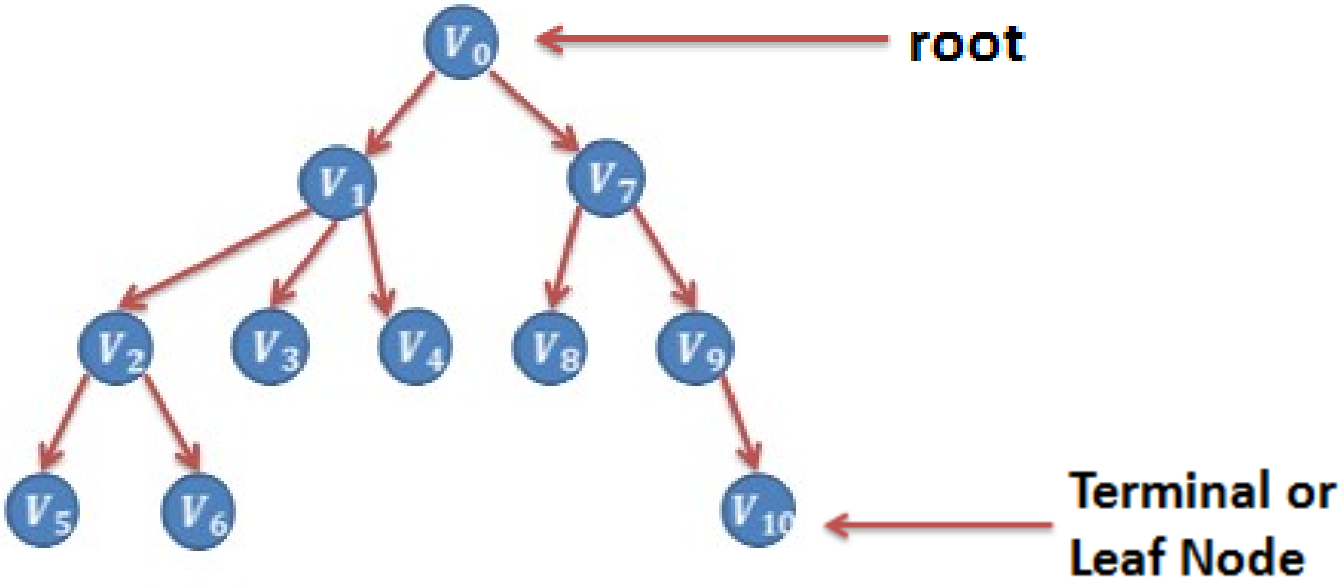
Tree

Prof. Siddharth Shah

Tree

- A tree is a **nonlinear** **hierarchical** data structure that consists of nodes connected by edges.
- A directed tree is an **acyclic digraph** which has one node called its **root** with in degree 0, while all other nodes have in degree 1.
- Every directed tree must have at least one node. An isolated node is also a directed tree.



Tree Terminologies

Node:

- A node is an entity that contains a key or value and pointers to its child nodes.
- The topmost node with indegree 0, is called a **root** node.
- The last nodes of each path are called **leaf / external / terminal** nodes that do not contain a link/pointer to child nodes.
- The node having at least a child node is called an **internal** node.

Edge:

- It is the link between any two nodes.

Tree Terminologies

- **Height of a Node:**

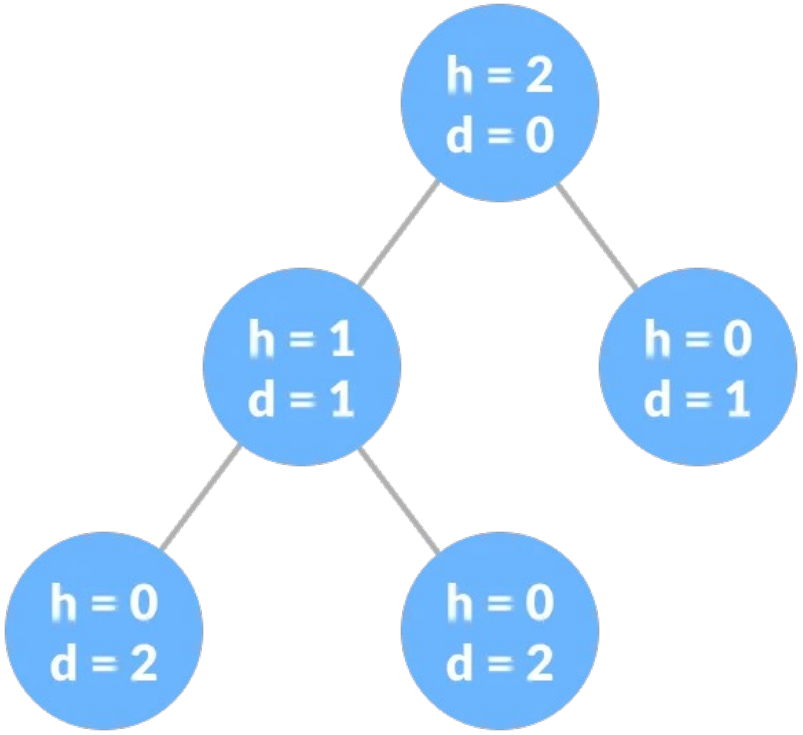
  The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

- **Depth / Level of a Node:**

  The depth of a node is the number of edges from the root to the node.

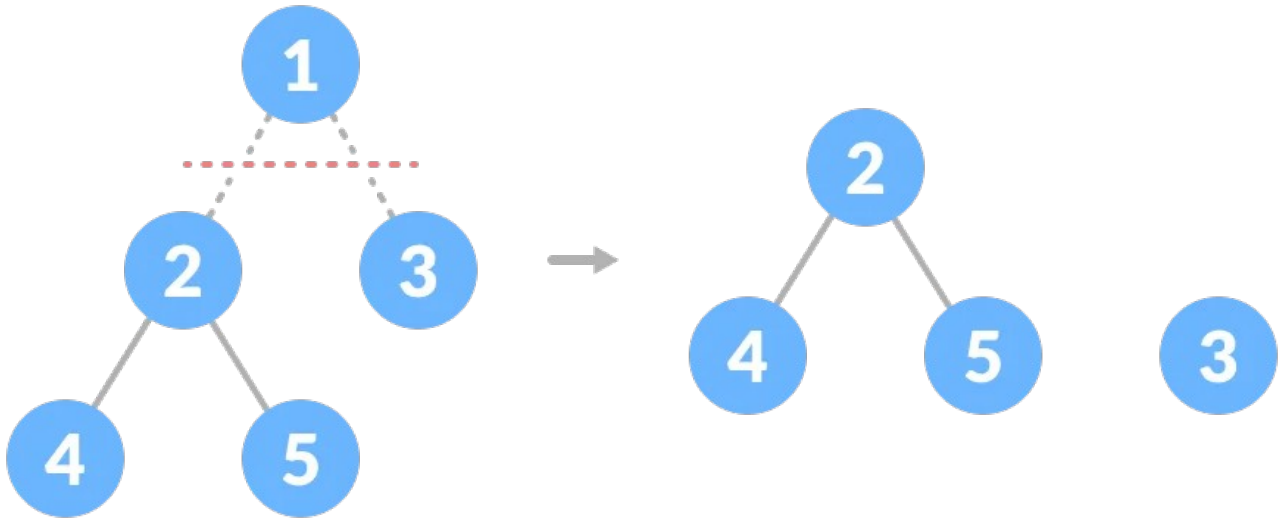
- **Height of a Tree:**

  The height of a Tree is the height of the root node or the depth of the deepest node.



Tree Terminologies

- **Degree of a Node:** The degree of a node is the total number of branches of that node.
- **Sibling:** Siblings are nodes that share the same parent node.
- **Forest**
  - A collection of disjoint trees is called a forest.
  - You can create a forest by cutting the root of a tree.

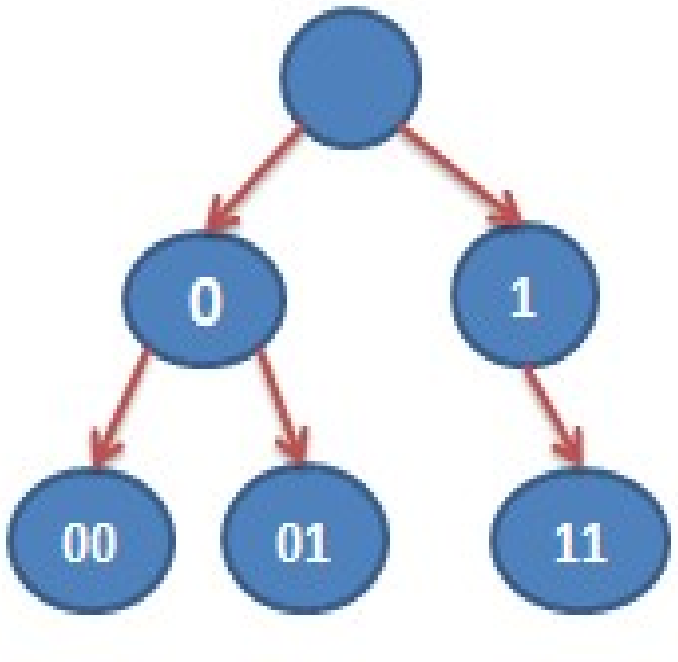


Types of Tree

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree
- Red-Black Tree

Binary Tree – (1)

- It is a directed tree in which the out degree of every node is less than or equal to 2.



Binary Tree – (2)

- **Strictly Binary Tree:** Every node other than the leaves has two children. Also known as proper binary tree or 2-tree **(Fig 1)**.
- **Complete binary tree:** The out degree of each and every node is exactly equal to 2 or 0 and the number of nodes at level i is 2^i **(Fig 2)**.
- **Almost complete binary Tree:** A Binary Tree of depth d is Almost Complete iff **(Fig 3)** :

1. The tree is Complete Binary Tree (All nodes) till level (d-1).
2. At level d, (i.e the last level), if a Node is present, then all the Nodes to the left of that node should also be present.

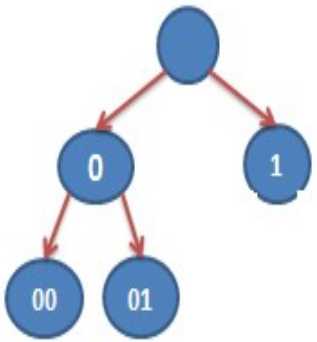


Fig 1

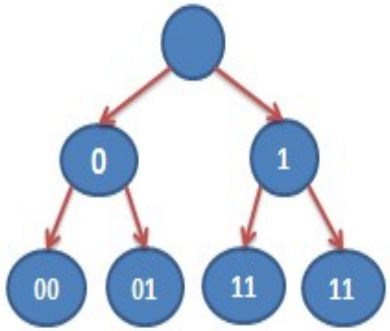


Fig 2

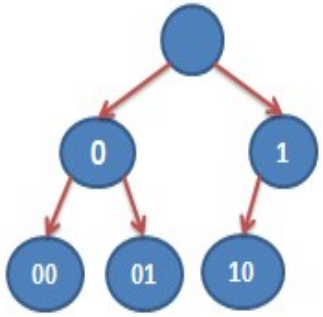
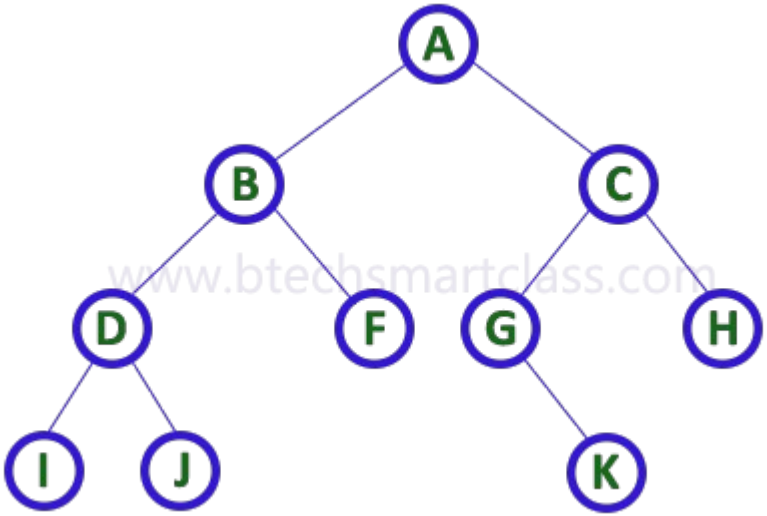


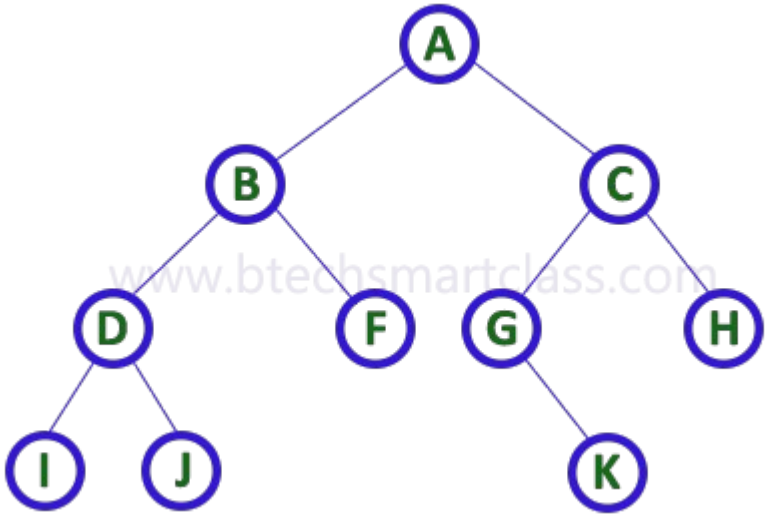
Fig 3

Binary Tree Representations

- A binary tree data structure is represented using two methods. Those methods are as follows.
  1. Array Representation
  2. Linked List Representation
- Consider the following binary tree.



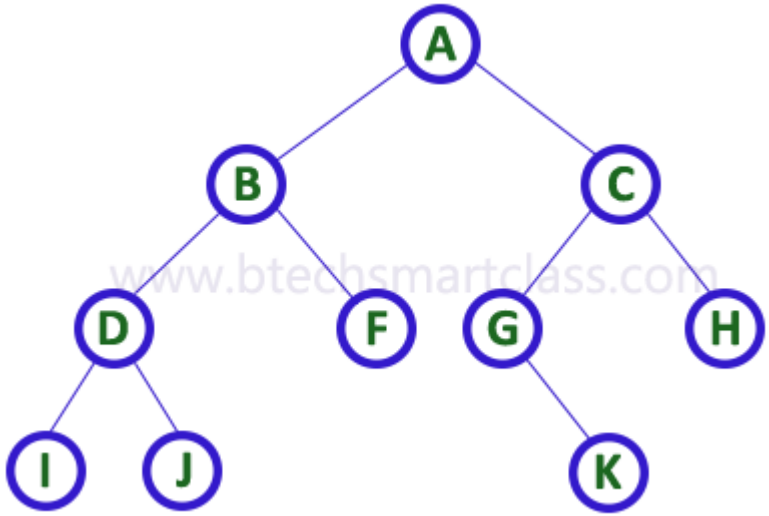
Binary Tree Representations - Array

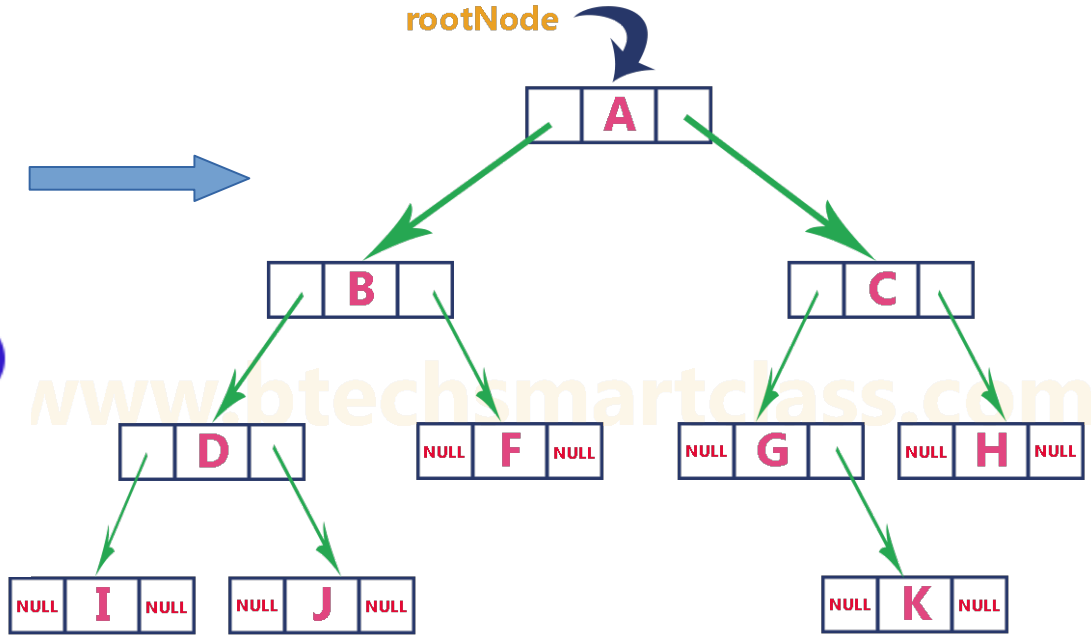


A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

Binary Tree Representations – Linked List


```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```





Tree Traversal

- In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.
- Traversal is a procedure by which each node in the tree is processed exactly once in a systematic manner.
- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.
- There are three ways of traversing a binary tree.

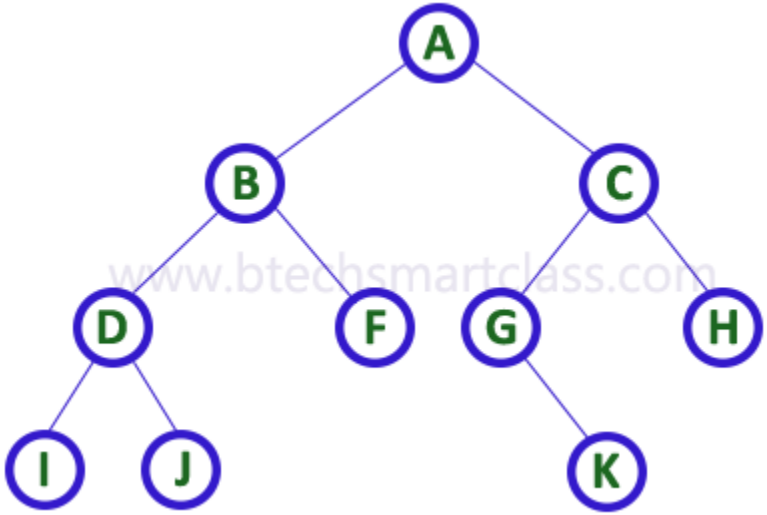
    1.Inorder Traversal

    2.Preorder Traversal

    3.Postorder Traversal

Inorder Traversal

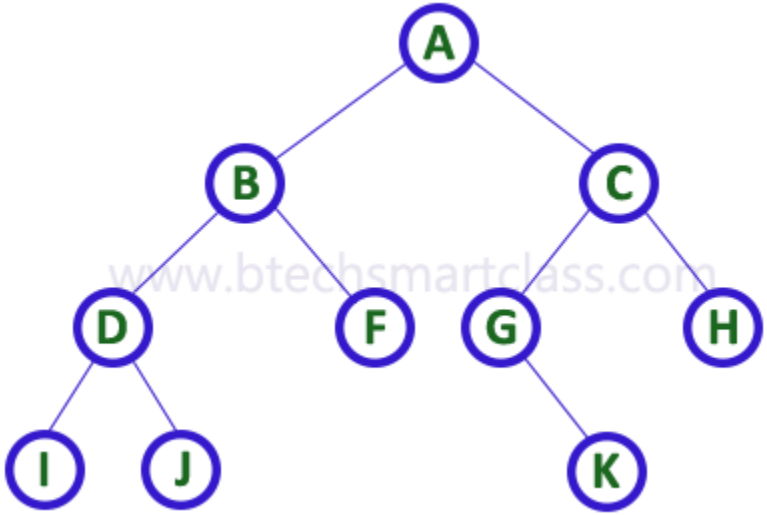
1. Traverse the left subtree in inorder **(Left)**
2. Process the root node **(Root)**
3. Traverse the right subtree in inorder **(Right)**



Inorder Traversal : I - D - J - B - F - A - G - K - C - H

Preorder Traversal

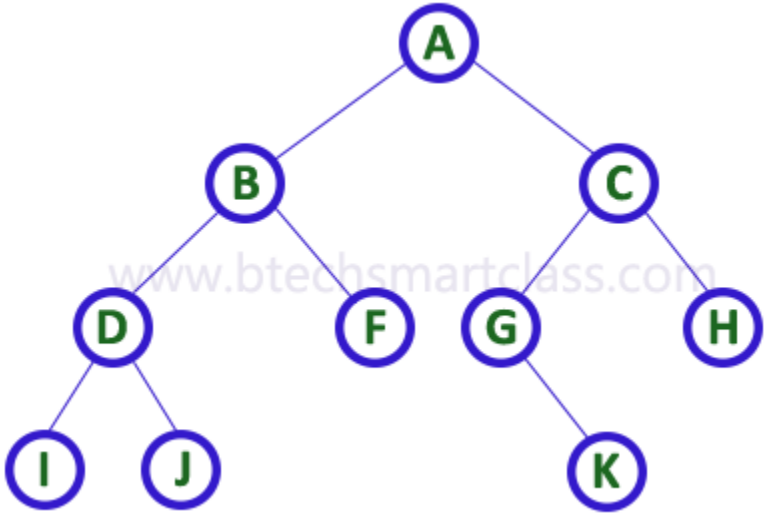
1. Process the root node **(Root)**
2. Traverse the left subtree in preorder **(Left)**
3. Traverse the right subtree in preorder **(Right)**



Preorder Traversal : A - B - D - I - J - F - C - G - K - H

Postorder Traversal

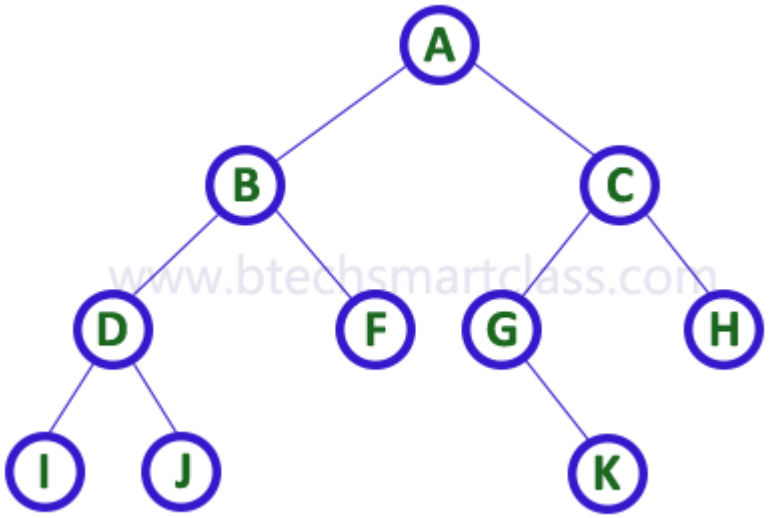
1. Traverse the left subtree in postorder **(Left)**
2. Traverse the right subtree in postorder **(Right)**
3. Process the root node **(Root)**



Postorder Traversal : I - J - D - F - B - K - G - H - C - A

Converse Tree Traversal

- If we interchange left and right words in the preceding definitions, we obtain three new traversal orders which are called **Converse Inorder**, **Converse Preorder** and **Converse Postorder**.



Converse Inorder Traversal :

H – C – K - G – A – F - B - J - D - I

Converse Preorder Traversal :

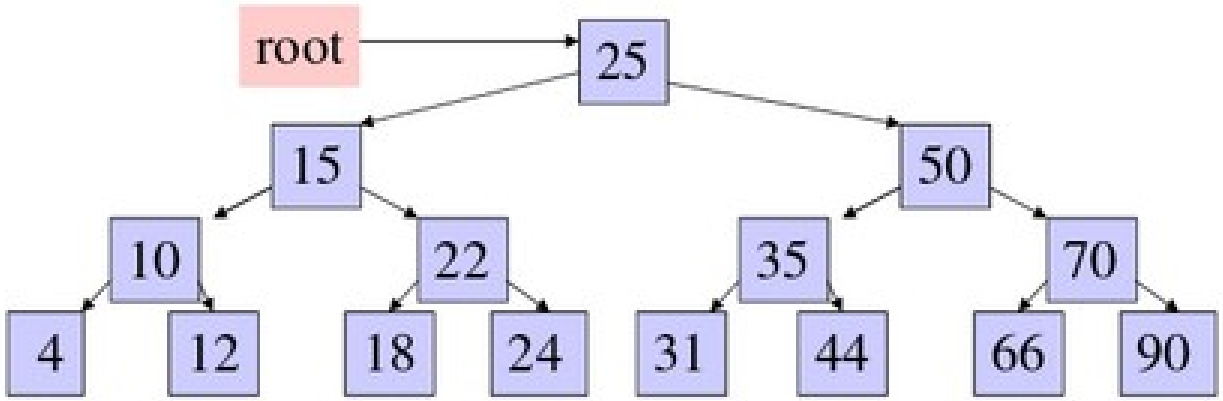
A – C – H - G – K – B - F - D - J - I

Converse Postorder Traversal :

H – K – G - C – F – J - I - D - B - A

Example

- Find Inorder, Preorder and Postorder traversal of the following tree.



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

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

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

Constructing a Binary Tree From Traversals

Inorder = {2,5,6,10,12,14,15}

Preorder = {10,5,2,6,14,12,15}

Constructing a Binary Tree From Traversals

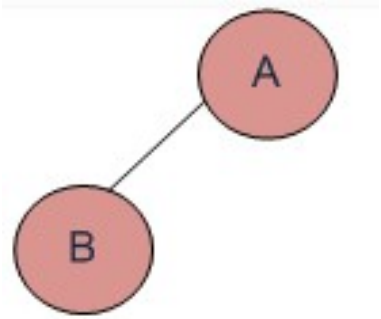
Inorder = {4, 2, 5, 1, 6, 3, 7}

Postorder = {4, 5, 2, 6, 7, 3, 1}

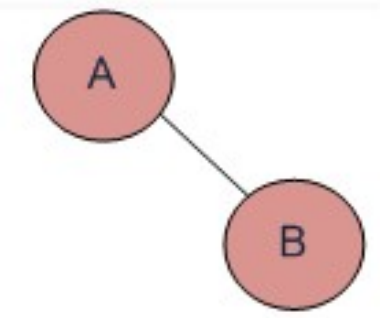
Constructing a Binary Tree From Traversals

Preorder = {A, B}

Postorder = {B, A}



OR



- It is not possible to construct a general Binary Tree from preorder and postorder traversals.
- But if it is known that the tree is strictly binary tree, it is possible to construct the tree without ambiguity.

Constructing a Binary Tree From Traversals

Preorder = {1, 2, 4, 8, 9, 5, 3, 6, 7}

Postorder = {8, 9, 4, 5, 2, 6, 7, 3, 1}

- In Preorder, the leftmost element is root of tree. Since the tree is strictly binary tree and array size is more than **1**. The value next to **1** in Preorder, must be left child of root. So we know **1** is root and **2** is left child.

- **How to find the all nodes in left subtree?**

  - We know **2** is root of all nodes in left subtree. All nodes before **2** in Postorder must be in left subtree.

  - Now we know **1** is root, elements **{8, 9, 4, 5, 2}** are in left subtree, and the elements **{6, 7, 3}** are in right subtree.

  - We recursively follow the above approach and get the following tree.

Assignment

Generate binary tree from the In order and Post order traversal of it as given below.

Inorder = {D, B, H, F, I, E, G, A, J, C, L, K, M}

Postorder = {D, H, I, F, G, E, B, J, L, M, K, C, A}

Inorder Traversal (Recursive)

Procedure : **RINORDER(T)**

Given a binary tree whose root node address is given by pointer variable T, this procedure traverses the tree in inorder, in a recursive manner.

1. [Check for empty tree]

If T= NULL

Then write ('EMPTY TREE')

Return

2. [Process the Left Subtree]

If LPTR(T) ≠ NULL

Then RINORDER (LPTR(T))

3. [Process the root node]

Write(DATA(T))

4. [Process the Right Subtree]

If RPTR(T) ≠ NULL

Then RINORDER (RPTR(T))

5. [Finished]

Return

Inorder Traversal (Iterative)

Procedure : **INORDER(T)**

- Given a binary tree whose root node address is given by pointer variable T, this procedure traverses the tree in inorder, in an iterative manner.
- S and TOP denote an auxiliary stack and its associated top index, respectively. The pointer variable P denotes the current node in the tree.

1. [Intialize]

If T= NULL
Then write ('EMPTY TREE')
Return
Else TOP ← 0
P ← T

2. [Traverse in inorder]

Repeat step 3 while TOP > 0 or P ≠ NULL

3. [Descend left]

If P ≠ NULL
Then Call PUSH(S, TOP, P)
P ← LPTR(P)
Else
P ← POP(S, TOP)
Write (DATA(P))
P ← RPTR(P)

4. [Finished]

Return

Preorder Traversal (Recursive)

Procedure : **RPREORDER(T)**

Given a binary tree whose root node address is given by pointer variable T, this procedure traverses the tree in preorder, in a recursive manner.

1. [Check for empty tree]

```
If      T= NULL
Then    write ('EMPTY TREE')
        Return
Else    write(DATA(T))
```

2. [Process the Left Subtree]

```
If      LPTR(T) ≠ NULL
Then    RPREORDER(LPTR(T))
```

3. [Process the Right Subtree]

```
If      RPTR(T) ≠ NULL
Then    RPREORDER(RPTR(T))
```

4. [Finished]

```
Return
```

Preorder Traversal (Iterative)

Procedure : **PREORDER(T)**

- Given a binary tree whose root node address is given by pointer variable T, this procedure traverses the tree in preorder, in an iterative manner.
- S and TOP denote an auxiliary stack and its associated top index, respectively. The pointer variable P denotes the current node in the tree.

1. [Intialize]

```
If      T= NULL
Then    write ('EMPTY TREE')
        Return
Else    TOP ← 0
        Call PUSH(S,TOP,T)
```

2. [Process each stacked branch address]

Repeat step 3 while TOP > 0

3. [Get stored address and branch left]

```
P ← POP(S, TOP)
Repeat while P ≠ NULL
    Write (DATA(P))
    If RPTR(P) ≠ NULL
    /* store address of nonempty right subtree */
    Then Call PUSH(S,TOP,RPTR(P))
    P ← LPTR(P) /* branch left */
```

4. [Finished]

Return

Postorder Traversal (Recursive)

Procedure : **RPOSTORDER(T)**

Given a binary tree whose root node address is given by pointer variable T, this procedure traverses the tree in postorder, in a recursive manner.

1. [Check for empty tree]

If T= NULL

Then write ('EMPTY TREE')

Return

2. [Process the Left Subtree]

If LPTR(T) ≠ NULL

Then RPOSTORDER (LPTR(T))

3. [Process the Right Subtree]

If RPTR(T) ≠ NULL

Then RPOSTORDER (RPTR(T))

4. [Process the root node]

Write(DATA(T))

5. [Finished]

Return

Postorder Traversal (Iterative)

Procedure : **POSTORDER(T)**

- Given a binary tree whose root node address is given by pointer variable T, this procedure traverses the tree in postorder, in an iterative manner.
- S and TOP denote an auxiliary stack and its associated top index, respectively. The pointer variable P denotes the current node in the tree.
- Here we need two types of stack entries, the first indicating that a left subtree is being traversed, and the second that a right subtree is being traversed.
- For convenience we will use negative pointer values to indicate the second type of entry. This, of course, assumes that valid pointer data are always nonzero and positive.

Postorder Traversal (Iterative)

Procedure : **POSTORDER(T)**

1. [Intialize]

If T= NULL
Then write ('EMPTY TREE')
Return
Else P ← T
TOP ← 0

2. [Traverse in postorder]

Repeat thru step 5 while true

3. [Descend left]

Repeat while P ≠ NULL
Call PUSH(S, TOP, P)
P ← LPTR(P)

4. [Process a node whose left and right subtrees have been traversed]

Repeat while S[TOP] < 0
P ← POP(S, TOP)
Write(DATA(P))
/* Have all nodes been processed? */
If TOP = 0
Then Return

5. [Branch right and then mark node from which we branched]

P ← RPTR(S[TOP])
S[TOP] ← -S[TOP]

Create copy of a binary tree

Procedure : **COPY (T)**

This procedure generates a copy of the tree with root node T and returns the address of its root node. NEW is a temporary pointer variable.

1. [Null Pointer?]

If T=NULL

Then Return (NULL)

2. [Create a new node]

NEW <= NODE

3. [Copy information field]

DATA(NEW) ← DATA(T)

4. [Set the structural links]

LPTR(NEW) ← COPY(LPTR(T))

RPTR(NEW) ← COPY(RPTR(T))

5. [Return address of new node]

Return(NEW)