# Data Structure and Algorithm

# DS-Lec-12-13: Tree

# Discussed So far

- Here are some of the data structures we have studied so far:
  - Arrays
  - Stacks, Queues and deques
  - Linked list

- Inherently uni-dimensional in structure

# Tree

A **tree** is defined as a finite set of one or more nodes such that

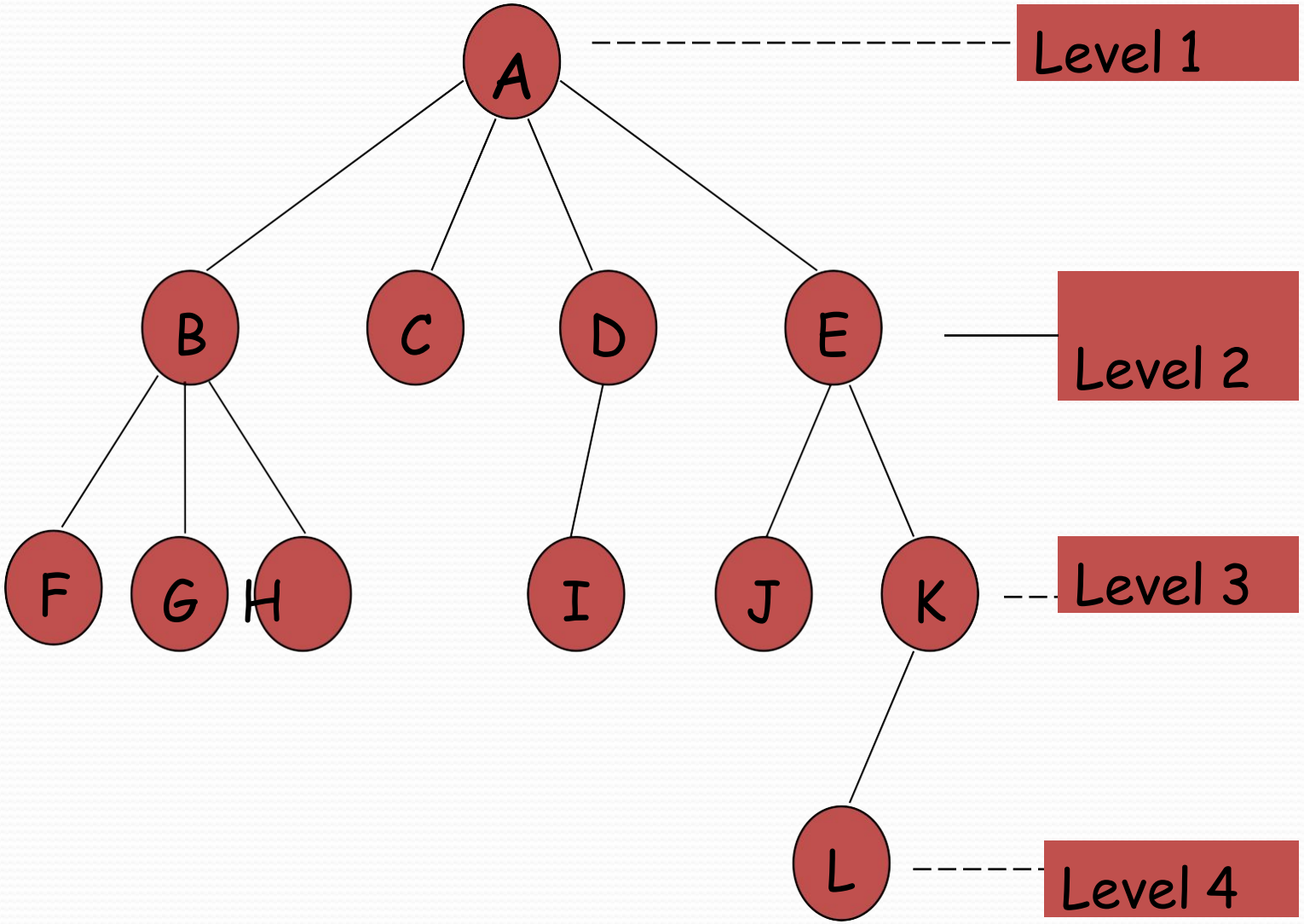
[a] There is a specially designated **node** called the **root** and

[b] The rest of the nodes could be partitioned into **t** disjoint sets (**t** $\geq$0) each set representing a tree $T_i$, i=1,2, .

t known as **subtree** of the tree.

# Tree

A **node** in the definition of the tree represents an item of information, and the links between the nodes termed as **Branches** represent an association between the items of information.


A
B
C
D
E
F
G
H
I
J
K
L
Level 1
Level 2
Level 3
Level 4

# Basic terminologies

- degree of the node
- leaf nodes or terminal nodes
- non terminal nodes
- children
- siblings
- ancestors
- degree of a tree
- hierarchical structure
- height or depth of a tree
- forest

# Tree Terminology

- A tree consists of a collection of elements or nodes, with each node linked to its successors
- The node at the top of a tree is called its **root**
- The links from a node to its successors are called **branches**
- The successors of a node are called its **children**

# Tree Terminology (continued)

- Each node in a tree has exactly **one parent** except for the root node, which has no parent
- Nodes that have the same parent are **siblings**
- A node that has no children is called a **leaf node**
- A generalization of the parent-child relationship is the ancestor-descendent relationship

# Tree Terminology (continued)

- The predecessor of a node is called its **Parent**
- A **subtree** of a node is a tree whose root is a child of that node
- The level of a node is a measure of its distance from the root

# Tree Terminology (continued)

- Node: stores the actual data and links to other nodes
- Parent: immediate predecessor of a node
- Root: specially designated node which has no parent
- Child:immediate successor of a node.

# Tree Terminology(continued)

- Leaf: node without any child

- Level: rank of the hierarchy and root node has level zero(0). Node at level **i** has the level **i+1** for its child and **i-1** for its parent. This is true for all nodes except the root

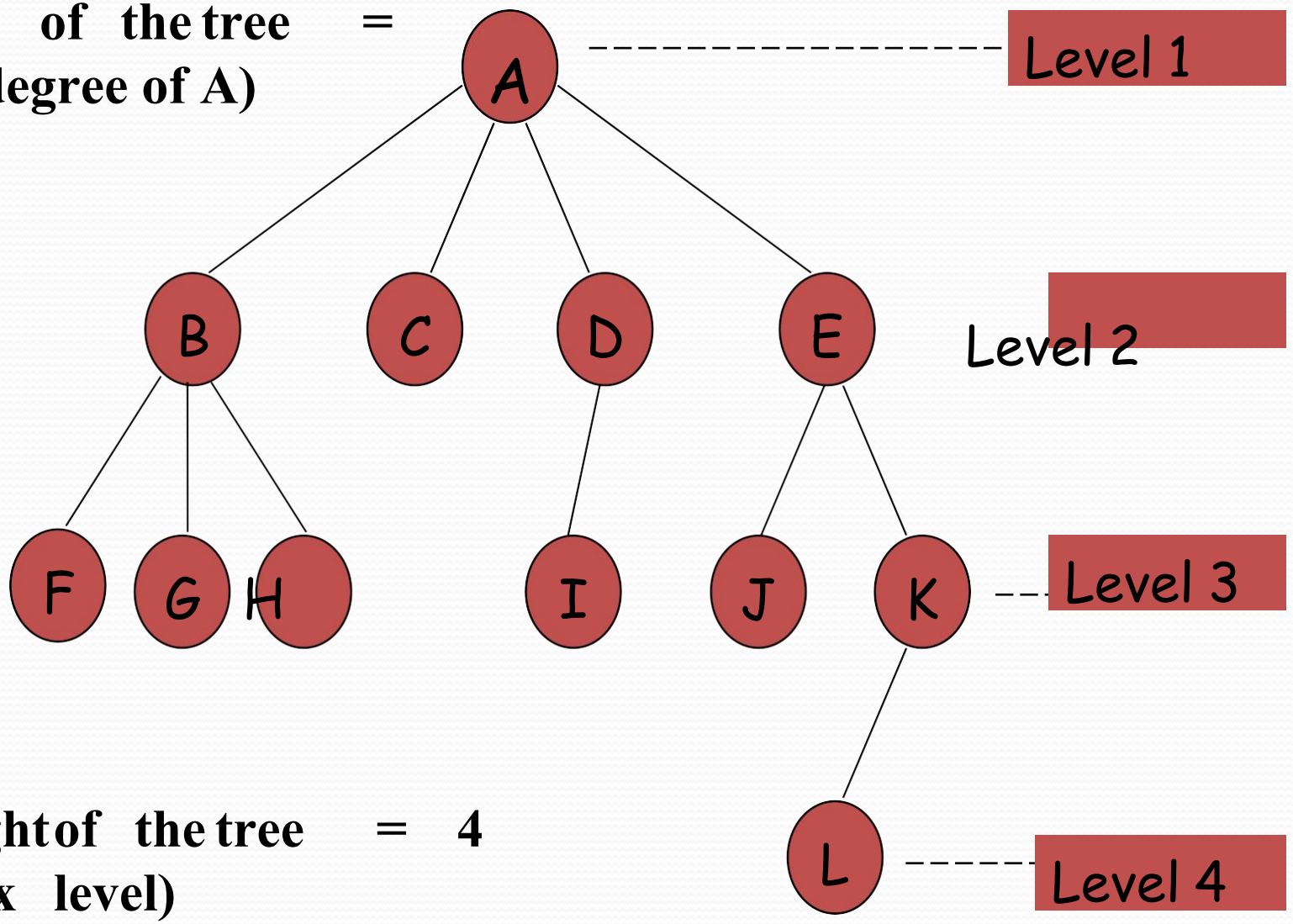
# Tree Terminology (continued)

- Height (depth): Maximum number of nodes possible in a path starting from root node to leaf node. Height of a tree is the maximum level of the tree.
- Degree of the Node: Maximum number of the children possible for a node.
- Siblings: Nodes having the same parent

# Tree Terminology

- Ancestor of a Node: Those nodes that occur on the path from the root to the given node
- Degree of a Tree: Maximum degree of the node in the tree
- Forest : A set of Zero or more Disjoint trees.

**Degree of the tree = 4 (Max degree of A)**




**Height of the tree = 4 (Max level)**

# Representation of a tree

- List Representation

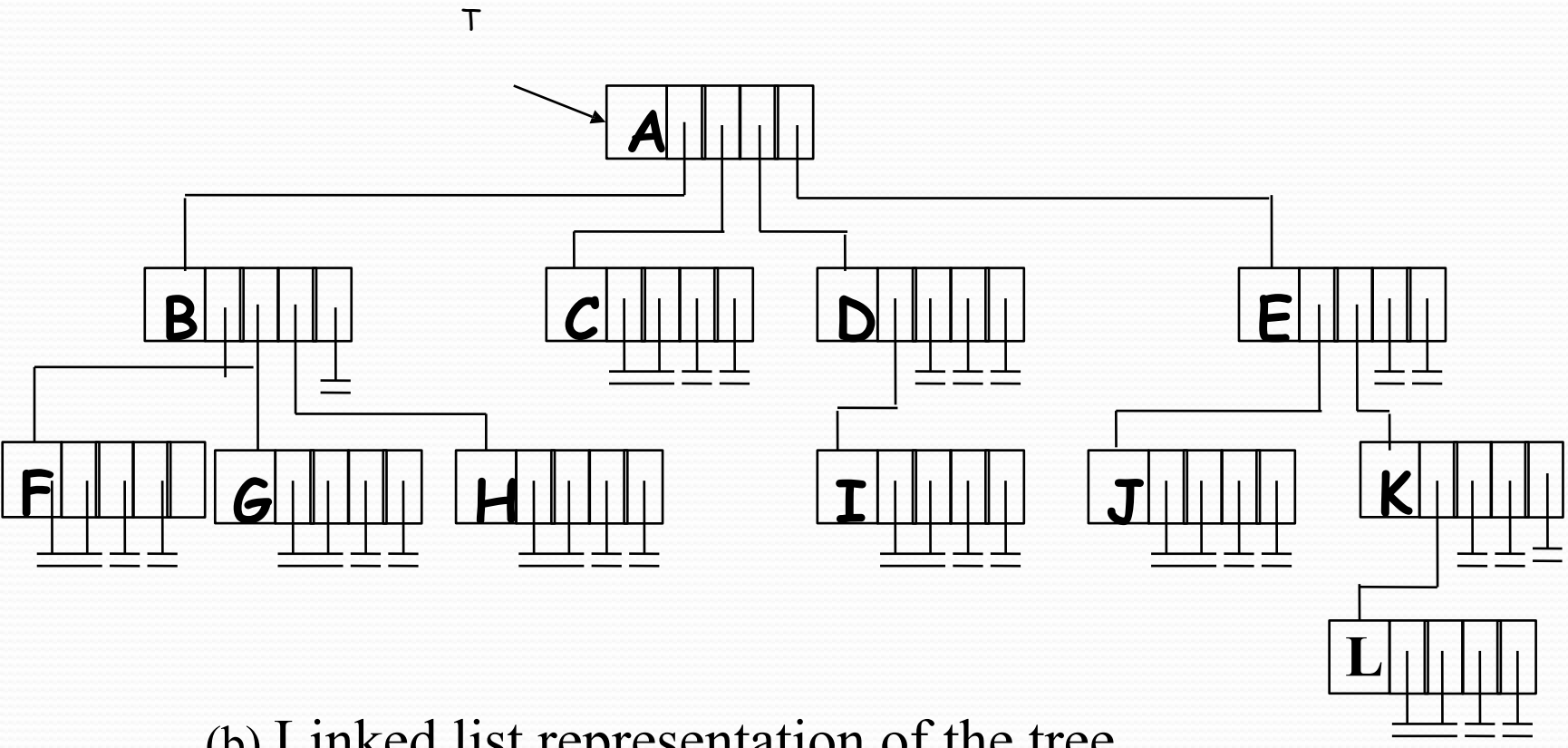
  (A (**B**(**F**,**G**,**H**), **C**, **D**(**I**), **E**(**J**,**K**(**L**))) )

  For the tree Considered in the Example

- Linked List Representation

| DATA | LINK 1 | LINK 2 | ... | LINK n |
|---|---|---|---|---|

(a) General node structure




(b) Linked list representation of the tree

# Binary Trees

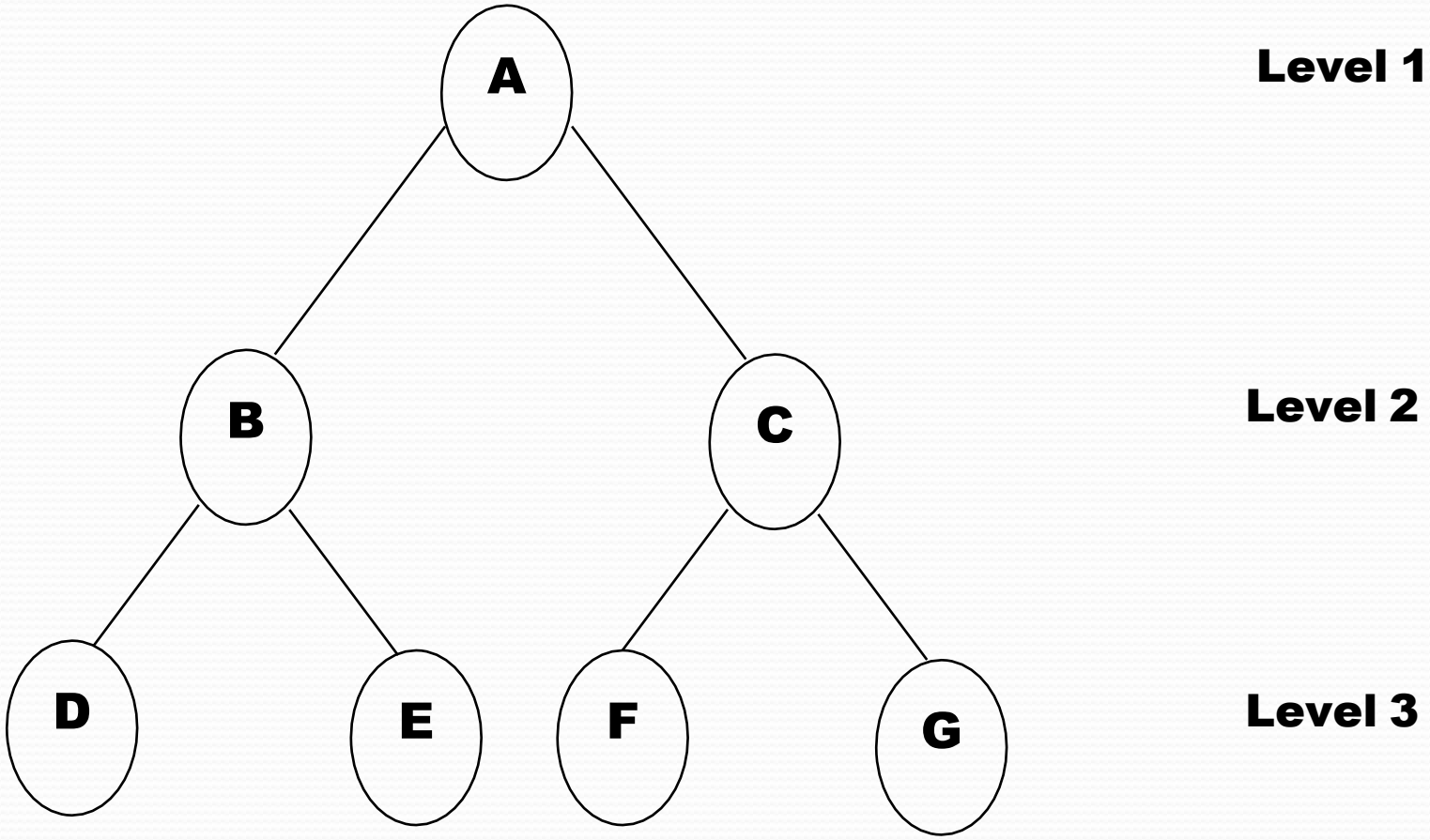
A binary tree T is defined as a finite set of elements called nodes such that

[a] T is empty (Called the Null tree or Empty tree) or

[b] T contains a distinguished node R called the root of T and the remaining nodes of T form an ordered pair of disjoint binary trees $T_1$ and $T_2$

# Binary Trees

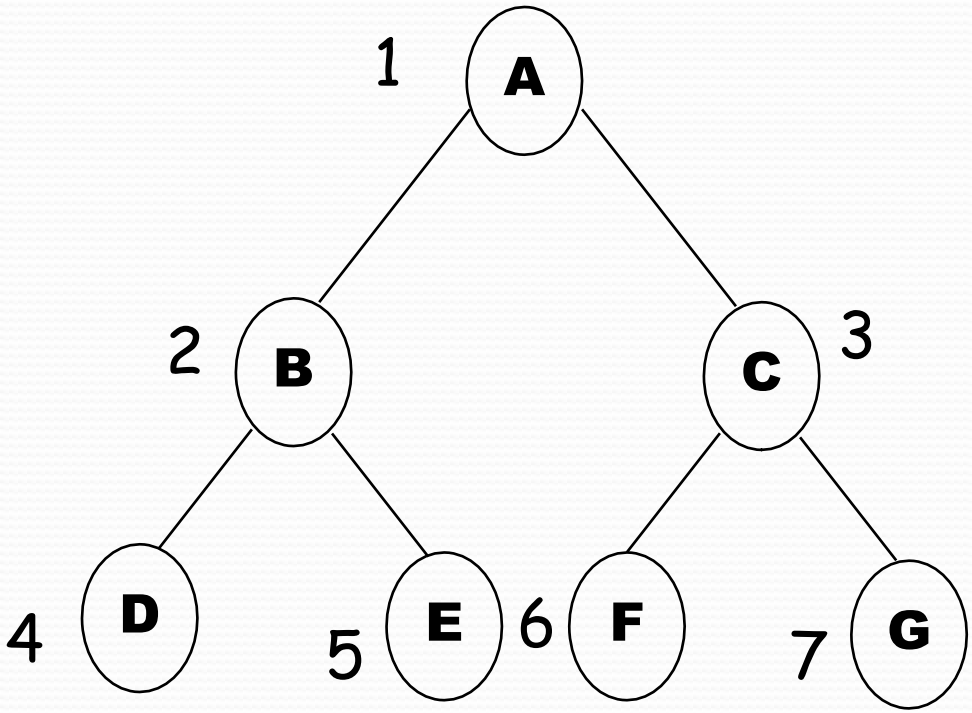
- A **binary tree** has the characteristic of all nodes having at most two branches, that is, all nodes have a **degree of at most 2**.
- A binary tree can therefore be **empty** or consist of a root node and two disjoint binary trees termed **left subtree** and **right subtree**.


A
B
C
D
E
F
G
Level 1
Level 2
Level 3

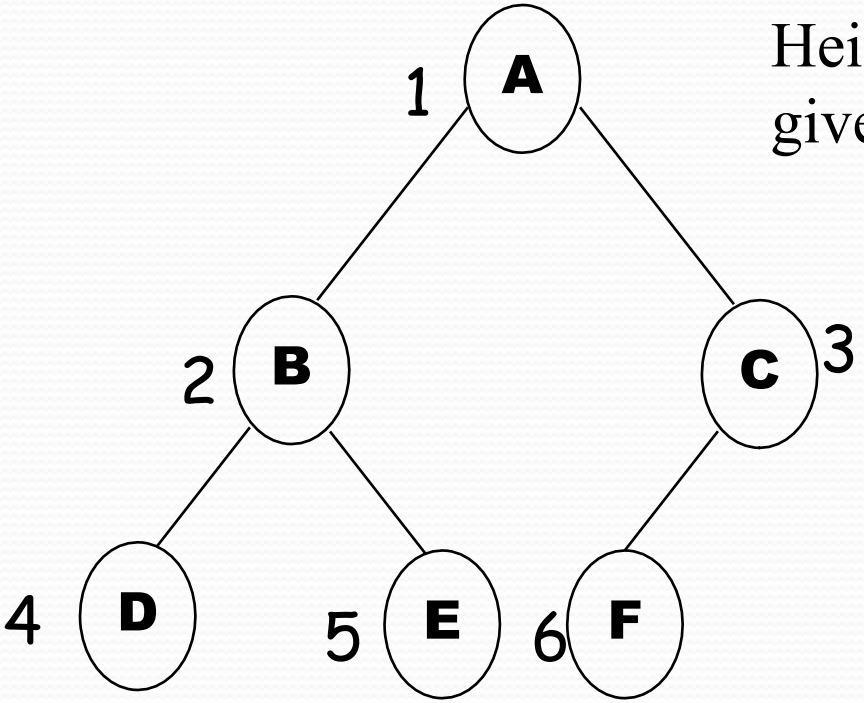
Important observations regarding binary trees:

- The maximum number of nodes on level **i** of a binary tree is $2^{i-1}$, $i \geq 1$
- The maximum number of nodes in a binary tree of height **h** is $2^{h}-1, h \geq 1$
- For any non empty binary tree, if $t_o$ is the number of terminal nodes and $t_2$ is the number of nodes of degree 2 then $t_o = t_2 + 1$

A binary tree of height **h** which has all its permissible maximum number of nodes **$2^h$-1** intact is known as a full binary tree of height h



A binary tree with **n'** nodes and height **h** is **complete** if its nodes correspond to the nodes which are numbered **1** to **n** (**n'≤n**) in a full binary tree of height **h**.




Height of a complete binary tree with **n** given by

$$h = \lceil \log_2 (n+1) \rceil$$

A complete binary tree obeys the following properties with regard to its node numbering:

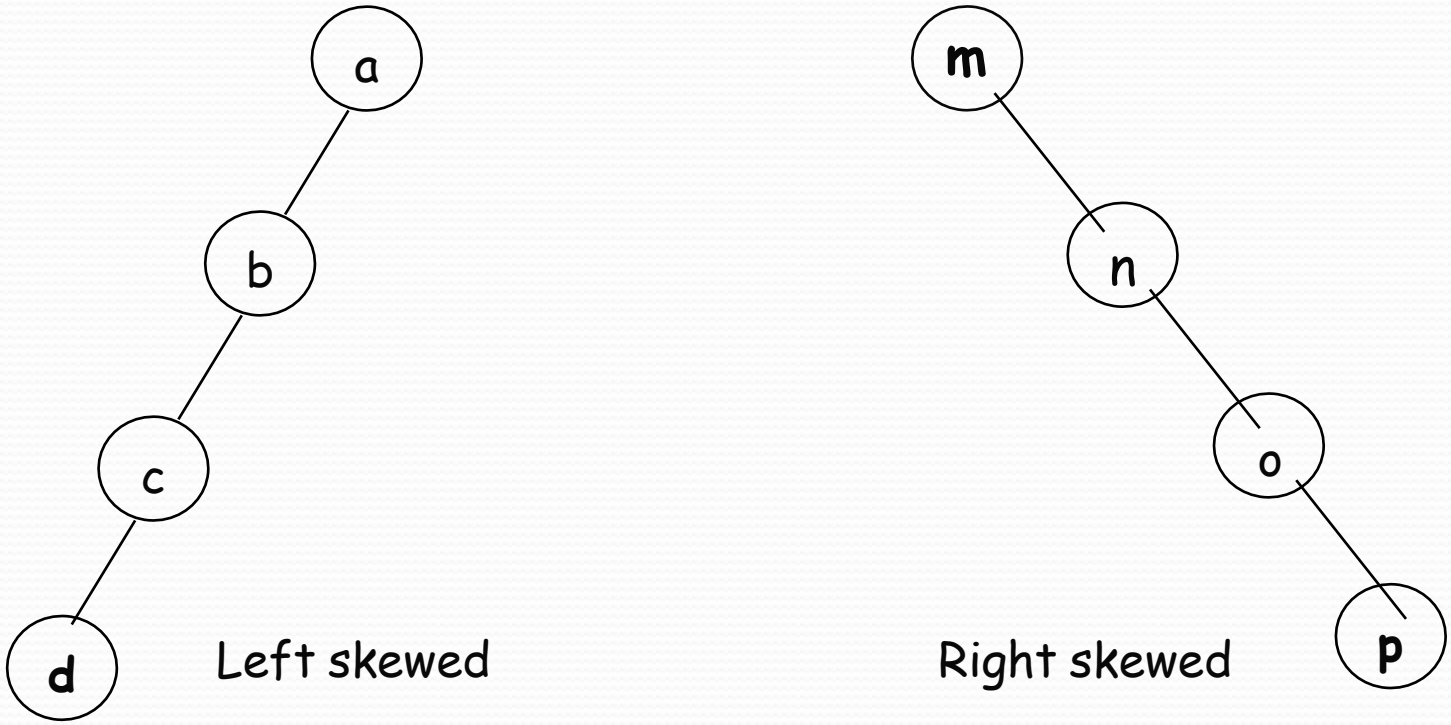
[a] If a parent node has a number i then its left child has the number 2i ($2i \leq n$). If $2i > n$ then i has no left child.

[b] If a parent node has a number i, then its right child has the number 2i+1 ($2i + 1 \leq n$). If $2i + 1 > n$ then i has no right child.

A complete binary tree obeys the following properties with regard to its node numbering:

[c] If a child node (left or right) has a number i then the parent node has the number $\lfloor i/2 \rfloor$ if $i \neq 1$. If i=1 then **i** is the root and hence has no parent.

A binary tree which is dominated solely by left child nodes or right child nodes is called a **skewed binary tree** or more specifically **left skewed binary tree** or **right skewed binary tree** respectively.



# Extended Binary Tree: 2-Tree

A binary tree **T** is said to be 2-Tree or an extended binary tree if each node **N** has either 0 or 2 children.

Nodes with 2 children are called internal nodes and the nodes with 0 children are called external nodes.

# Representation of Binary Tree

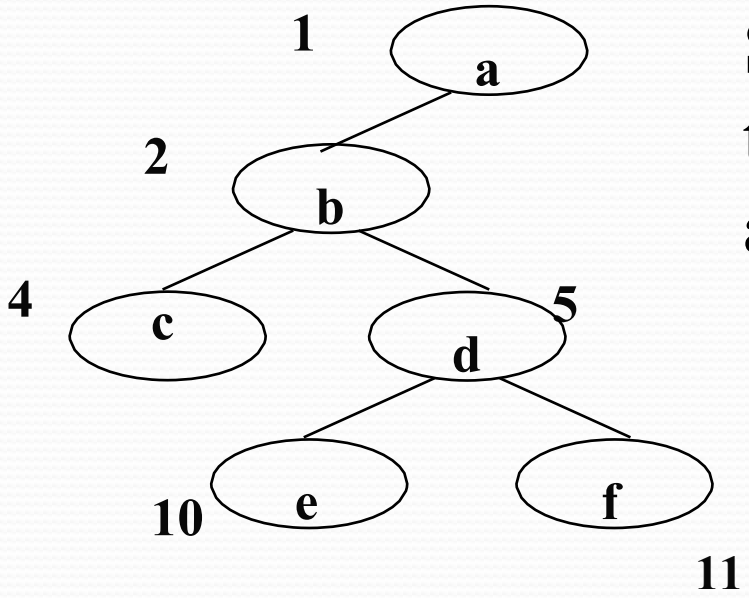
Binary tree can be represented by means of

[a] Array

[b] linked list

# Representation Of Binary Trees

## Array Representation

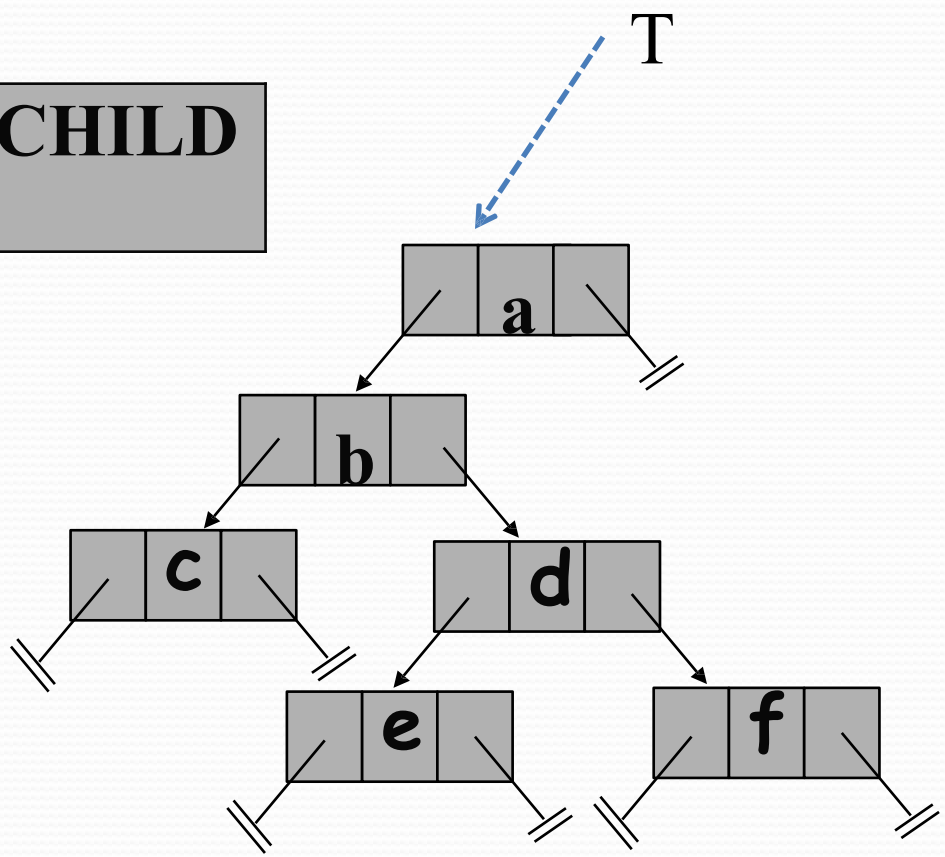



Sequential representation of a tree with depth **d** will require an array with approx $2^{d+1}$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | | c | d | | | | | e | f |

# Linked representation

| LCHILD | DATA | RCHILD |
|---|---|---|



- Observation regarding the linked representation of Binary Tree

[a] If a binary tree has **n** nodes then the number of pointers used in its linked representation is **2*n**

[b] The number of null pointers used in the linked representation of a binary tree with **n** nodes is **n+1**

# Build Tree

```
void insert(int item){
        Node* newbie = new Node();
        Node* parent = NULL;
        newbie->left = NULL;
        newbie->right= NULL;
        newbie->data = item;
        if(isEmpty()){
            root = newbie;
        }
```

```
else{
Node* ptr = root;
while(ptr != NULL){
parent = ptr;
if(item > ptr->data){
                    ptr = ptr->right;
          }
      else if(item < ptr->data){
                              ptr = ptr->left;
                    }
                    else
cout << "This object already exists in the tree!"<< endl;

  if(item > parent->data ){                    //put on the right
       parent->right = newbie;
                    }
if(item < parent->data ){                    //put on the left
                              parent->left = newbie;
                    } }}
```

# Traversing Binary Tree

Three ways of traversing the binary tree **T** with root **R**

**Preorder**

[a] Process the root **R**

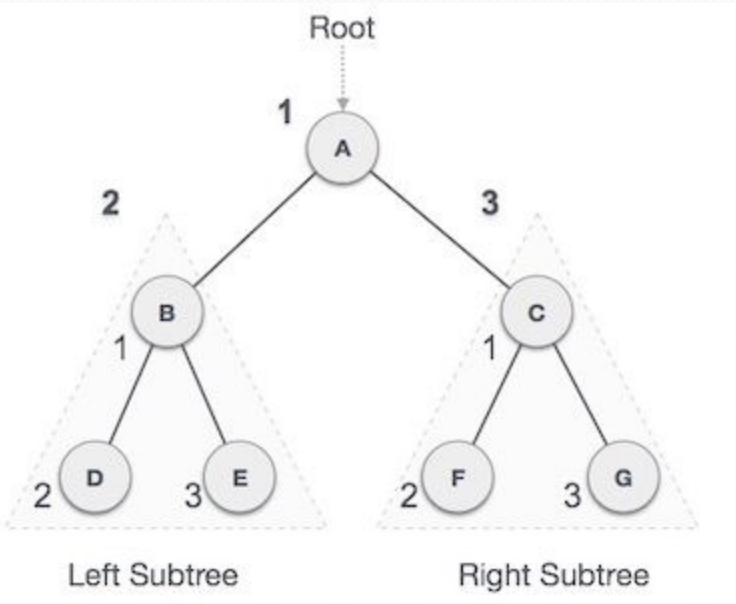
[b] Traverse the left sub-tree of **R** in preorder

[c] Traverse the right sub-tree of **R** in preorder

a. k. a node-left-right traversal (NLR)

# Pre-order Traversing

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.




Until all nodes are traversed −

**Step 1** − Visit root node.

**Step 2** − Recursively traverse left subtree.

**Step 3** − Recursively traverse right subtree.

We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow \quad B \rightarrow D \rightarrow E \rightarrow \quad C \rightarrow F \rightarrow G$$

# Programming Code for preorder

```
void BinaryTree::printPreorder(BinaryTree *node)
{
    if (node == NULL)
        return;
    cout << node->data << " ";
  printPreorder(node->left);
    printPreorder(node->right);
}
```

Until all nodes are traversed −
**Step 1** − Visit root node.
**Step 2** − Recursively traverse left sub
**Step 3** − Recursively traverse right su

# Traversing Binary Tree

**In-order**

[a] Traverse the left sub-tree of **R** in inorder

[b] Process the root **R**

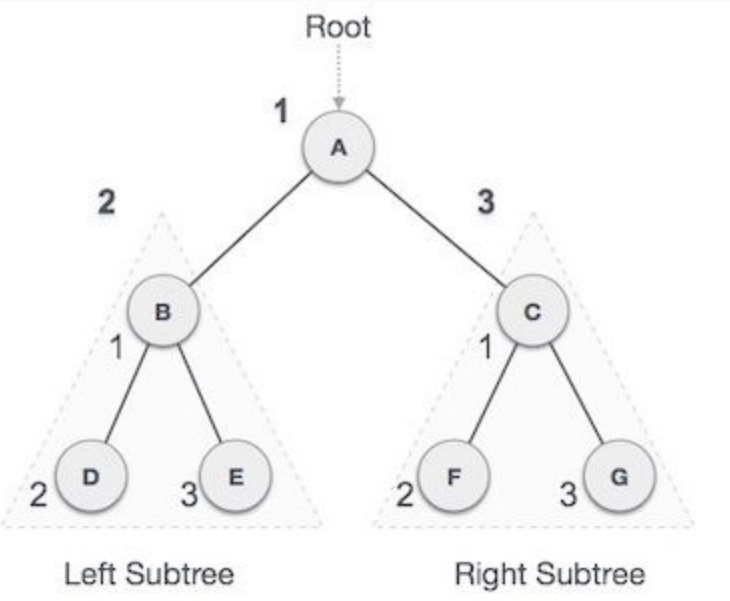
[c] Traverse the right sub-tree of **R** in in- order

a. k. a left-node-right traversal (LNR)

# In-order Traversing

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.




Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Visit root node.

**Step 3** − Recursively traverse right subtree.

We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \quad \rightarrow A \rightarrow \quad F \rightarrow C \rightarrow G$$

# Programming Code for inorder

```
void BinaryTree::printInorder(BinaryTree *node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    cout << node->data << " ";
    printInorder(node->right);
}
```

# Traversing Binary Tree

**Post-order**

[a] Traverse the left sub-tree of **R** in post-order

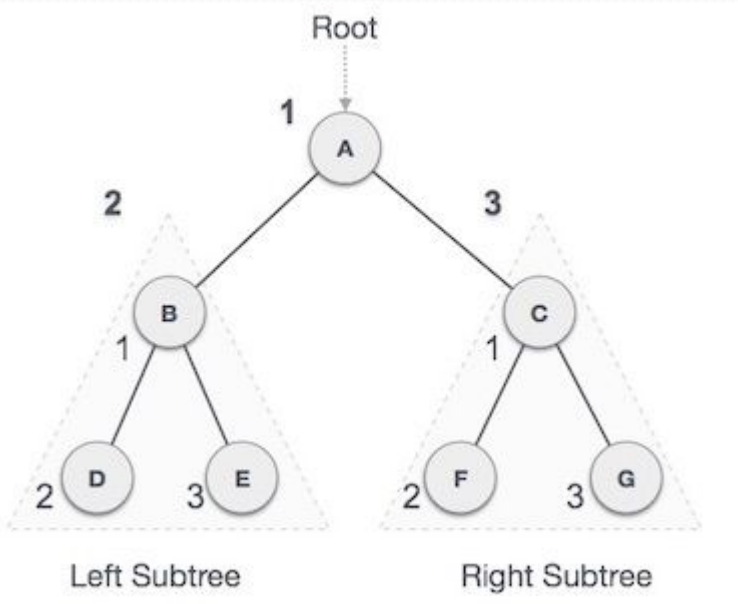
[b] Traverse the right sub-tree of **R** in postorder

[c] Process the root **R**

a. k. a left-right-node traversal (LRN)

# Post-order Traversing

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.




Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Recursively traverse right subtree.

**Step 3** −Visit root node.

We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow \quad F \rightarrow G \rightarrow C \quad \rightarrow A$$

# Programming Code for postorder

```
void BinaryTree::printPostorder(BinaryTree *node)
{
    if (node == NULL)
        return;
    printPostorder(node->left);
    printPostorder(node->right);
    cout << node->data << " ";
}
```

# Tree Traversal

There are basically three ways of binary tree traversals.

**1. Inorder --- (left child,root,right child)**

**2. Preorder --- (root,left child,right child)**

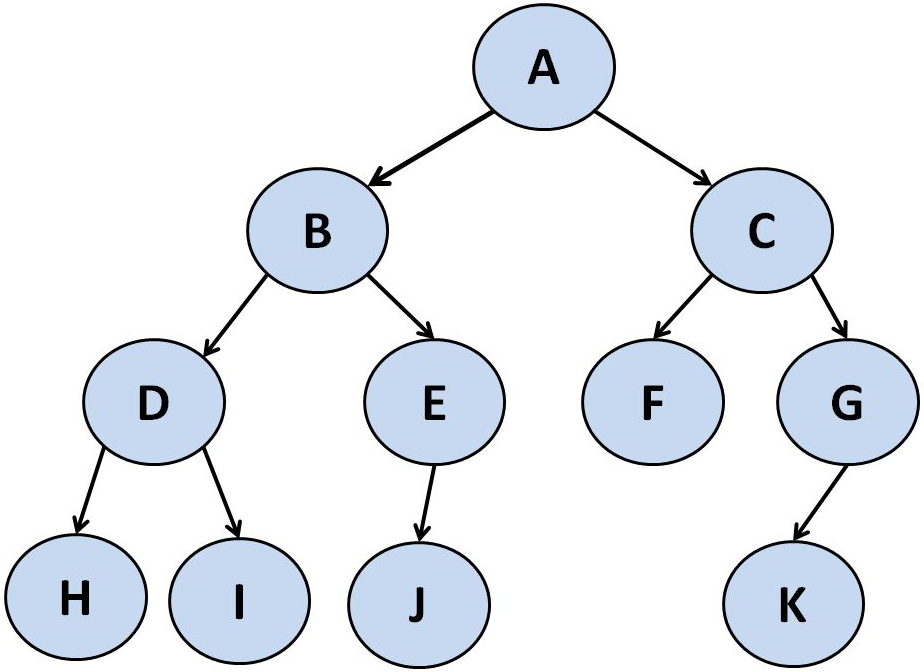
**3. Postorder --- (left child,right child,root)**

# APPLICATIONS

1. Some applications of preorder traversal are the evaluation of expressions in prefix notation and the processing of abstract syntax trees by compilers.
2. Binary search trees (a special type of BT) use inorder traversal to print all of their data in alphanumeric order.
3. A popular application for the use of postorder traversal is the evaluating of expressions in postfix notation.

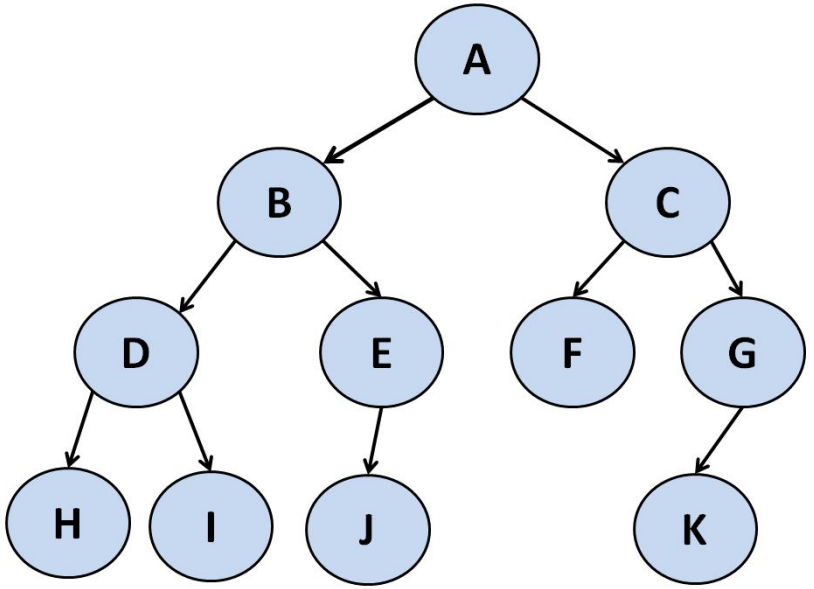
# Illustrations for Traversals

- Assume: visiting a node is printing its label
- **Preorder**:A,
- **Inorder**:
- **Postorder**:



# Illustrations for Traversals

- Assume: visiting a node is printing its info
- **Preorder**: A B D H I E J C F G K
- **Inorder**: H D I B J E A F C K G
- **Postorder**: H I D J E B F K G C A



# Formulation of Binary tree from Its traversal

1. If preorder is given=>First node is the root
   If postorder is given=>Last node is the root

2. Once the root node is identified, all nodes in the left subtrees and right subtrees of the root node can be identified.

3. Same technique can be applied repeatedly to form subtrees

4. Two traversals are essential out of which one should inorder, another may be preorder or postorder

5. But we can't form a binary tree if only preorder and postorder has given.

# Example: For Given Inorder and Preorder

Inorder: D B H E A I F J  CG

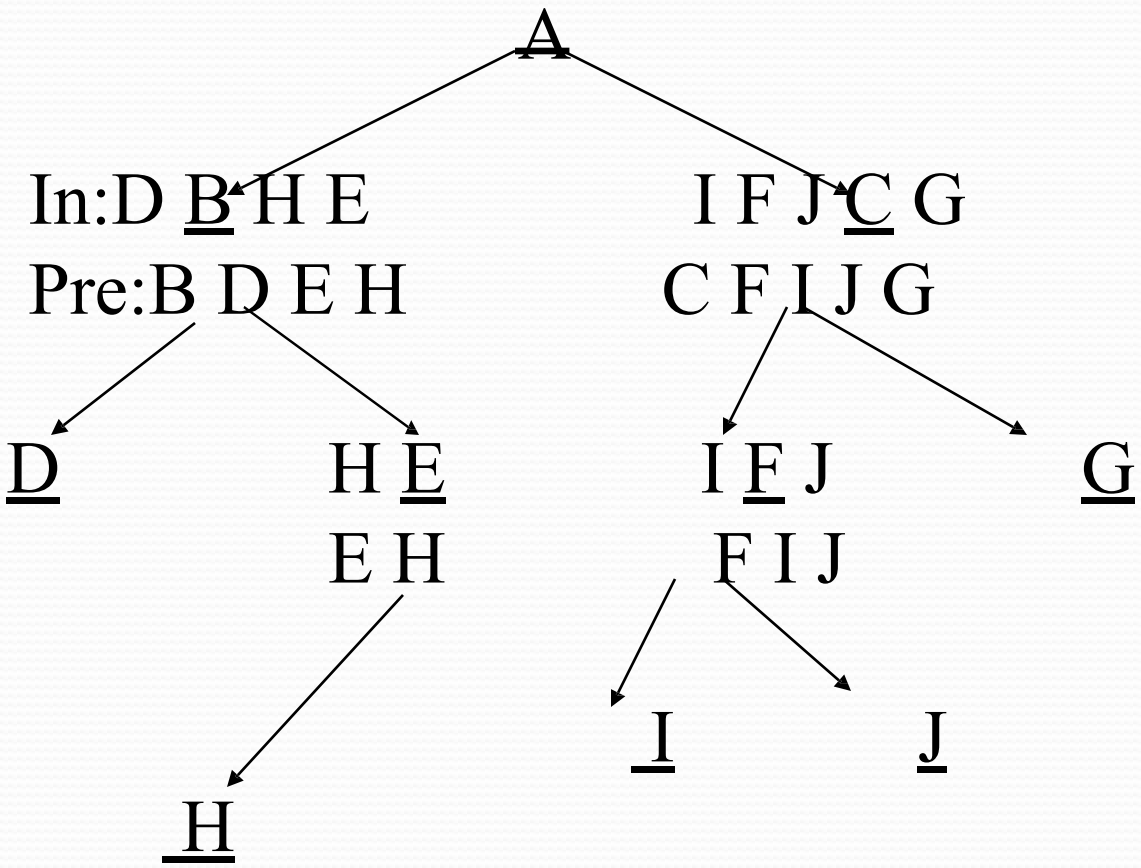
Preorder: A B D E H C F I J G

Now root is A

Left subtree: D B H E

Right subtree: I F J C G

# continues.

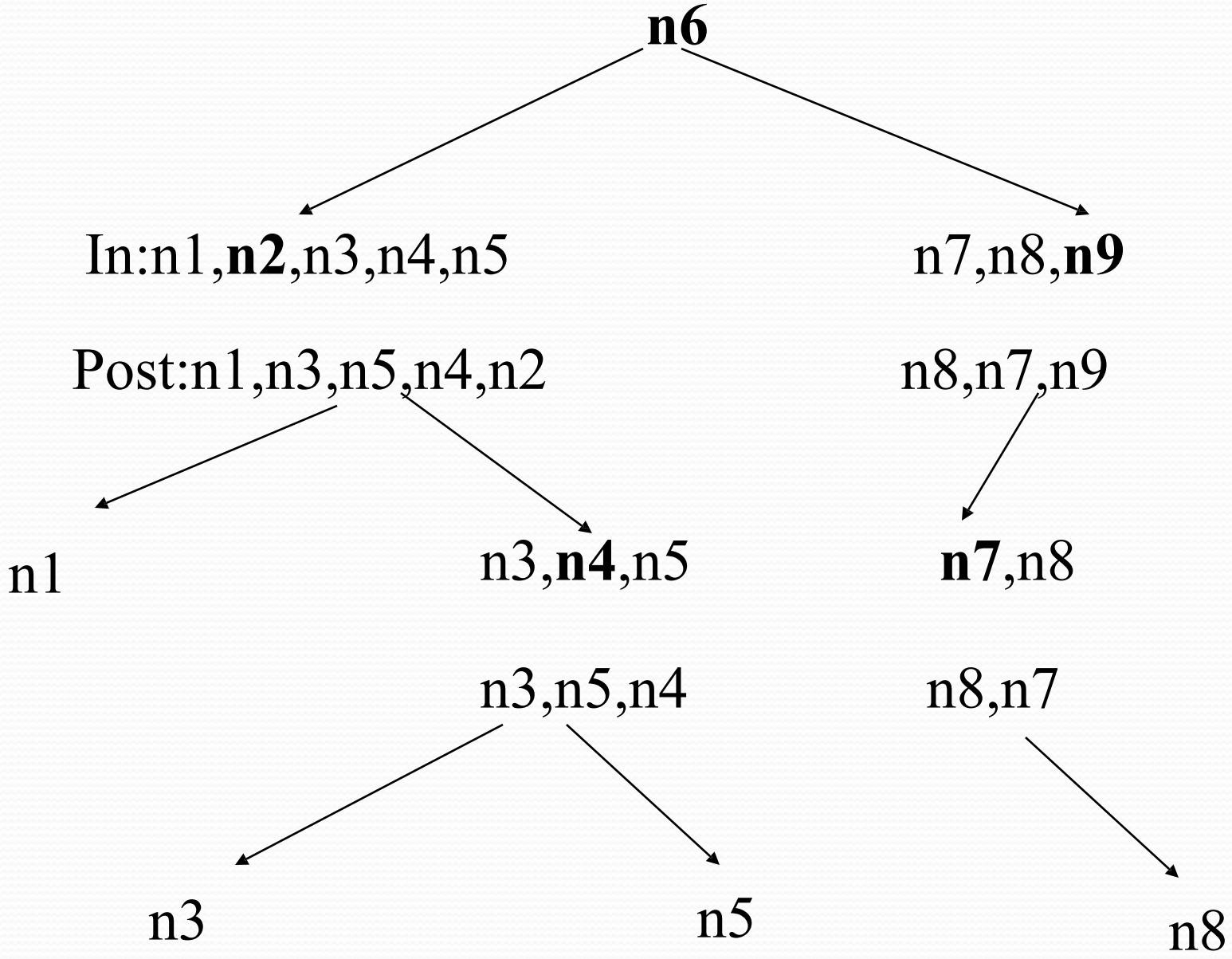


Example: For Given Inorder and Postorder

Inorder: n1,n2, n3, n4, n5, **n6**, n7, n8, n9

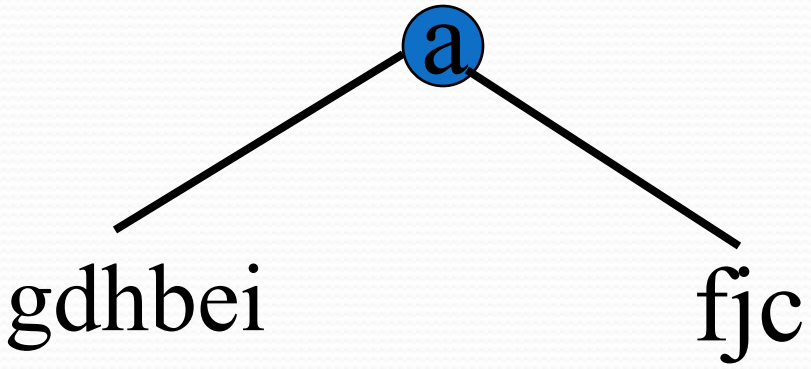
Postorder: n1,n3, n5, n4, n2, n8, n7, n9, **n6**

So here n6 is the root

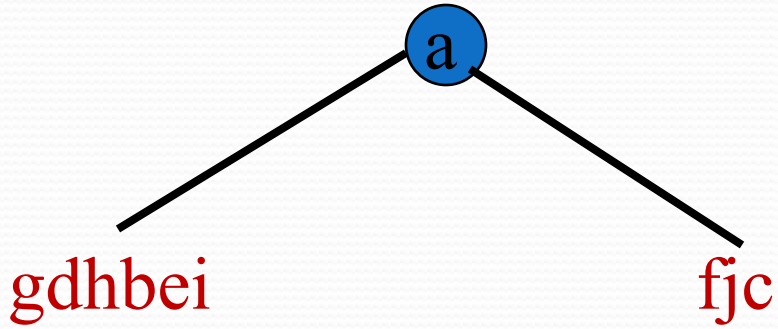

**n6**
In:n1,**n2**,n3,n4,n5
n7,n8,**n9**
Post:n1,n3,n5,n4,n2
n8,n7,n9
n1
n3,**n4**,n5
**n7**,n8
n3,n5,n4
n8,n7
n3
n5
n8

# Inorder And Preorder

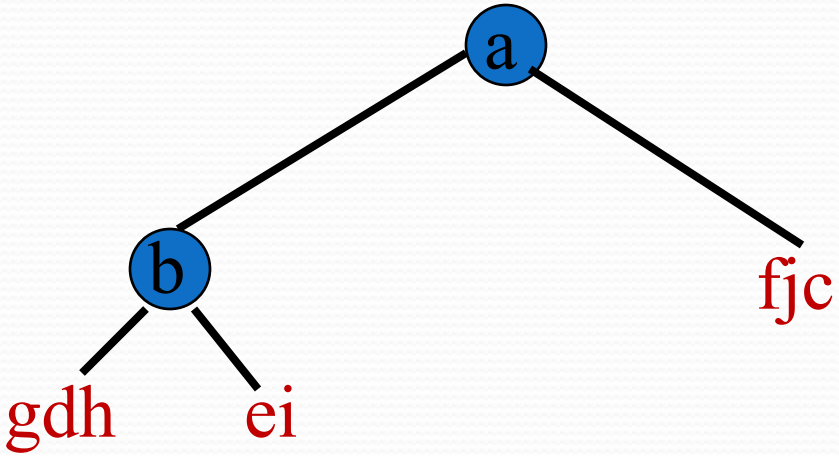
- inorder = g d h b e i a f j c
- preorder = a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.



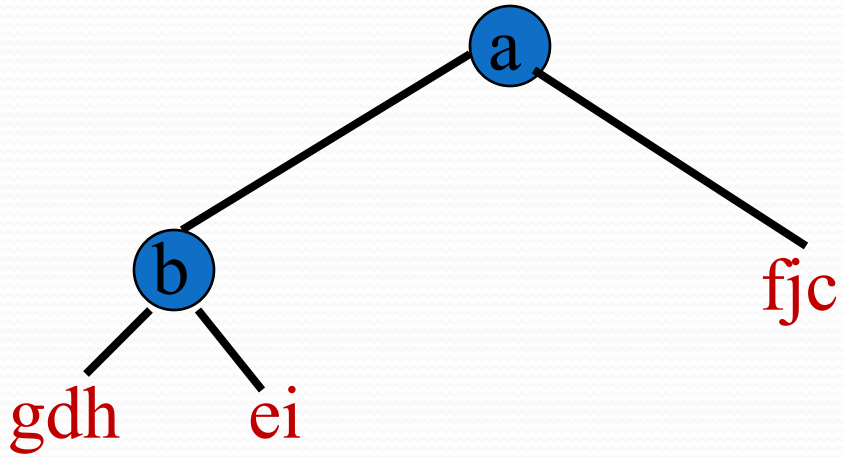
# Inorder And Preorder



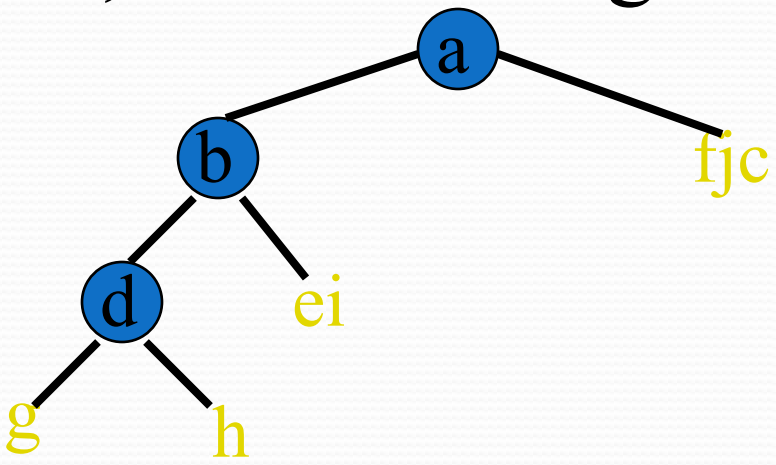

- preorder = a b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



# Inorder And Preorder




- preorder = a b d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.



# Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- postorder = g h d i e b j f c a
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

# Traversal Algorithm Using Stack

Assumption

Binary Tree is represented by

**TREE(INFO,LEFT, RIGHT,ROOT)**

A variable **PTR** (pointer) will contain the location of the node N currently being scanned. An array STACKwill hold the addresses of the node for future processing

# Pre-Order Traversal

[1] [ Initially push NULL onto STACK and initialize PTR]

Set TOP =1, STACK[1] = NULL and PTR = ROOT

[2] Repeat Steps 3 to 5 while PTR !=NULL

[3] Apply PROCESS to PTR->INFO

[4] [Right Child ?]

If PTR -> RIGHT !=NULL, then [Push on STACK]

SET TOP = TOP + 1, STACK[TOP] = PTR->RIGHT

[5] [Left Child ?]

If PTR->LEFT !=NULL, Then

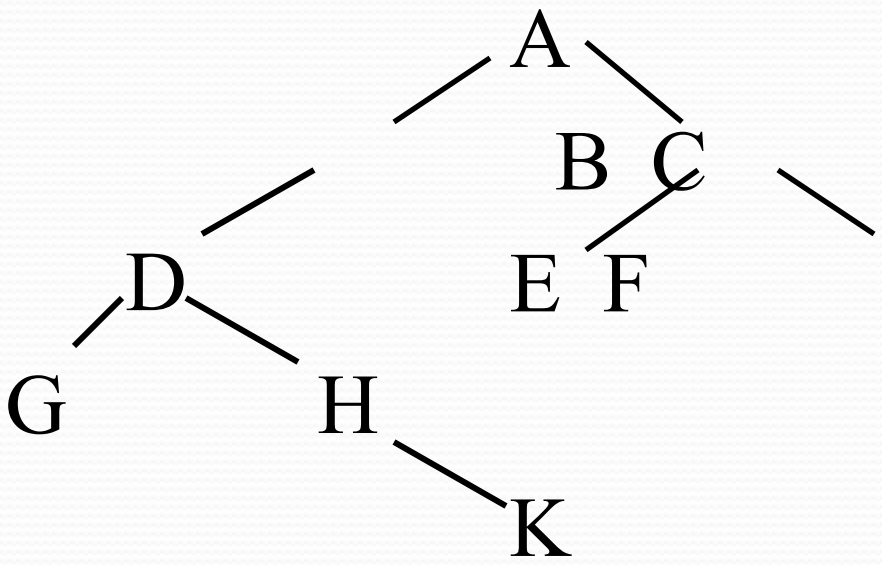
Set PTR = PTR->LEFT

Else

Set PTR = STACK[TOP], TOP = TOP-1

[6] Exit

# Pre-Order Traversal



[1] Initially push NULL onto STACK

STACK = φ

Set PTR = A , the root of T

[2] Proceed down the left-most path rooted at PTR = A as follows

(i) Process A and Push its right child C onto STACK.

STACK: φ, C

(ii) Process B. (There is no Right Child)

(iii) Process D and push its Right Child H onto STACK.

STACK: φ, C, H

(iv) Process G (There is no right child)

[3] [Backtracking] Pop the top element H from STACK, and set PTR = H

STACK: φ, C

[4] Proceed down the left-most path rooted at PTR =H as follows

(v) Process H and Push its right child K onto STACK:

STACK: φ, C, K

[No other node is processed, since H has no left child]

[5] [Backtracking] Pop the top element K from STACK, and set PTR = K

STACK: φ, C

[6] Proceed down the left-most path rooted at PTR = K as follows

(vi) Process K. (There is no right child)

[No other node is processed, since K has no left child] [7]

[Backtracking] Pop the top element C from STACK,

and set PTR = C

STACK: φ

[8] Proceed down the left-most path rooted at PTR = C as follows

(vii) Process C and push its right child F onto STACK.

STACK: φ, F

(viii) Process E (There is no right child)

[9] [Backtracking] Pop the top element F from STACK, and set PTR = F

STACK: φ

[10] Proceed down the left-most path rooted at PTR = F as follows

(ix) Process F. (There is no right child)

[No other node is processed, since F has no left child]

[11] [Backtracking] Pop the top element NULL from STACK, and set PTR = NULL

# In-order Traversal

[1] [Push NULL onto STACK and initialize PTR]

Set TOP =1, STACK[1] = NULL, PTR = ROOT

[2] Repeat while PTR != NULL

[Pushes the Left-most path onto STACK]

(a) Set TOP = TOP + 1, STACK[TOP] = PTR

(b) Set PTR = PTR -> LEFT

[3] Set PTR = STACK[TOP], TOP = TOP -1

[Pops node from STACK]

# In-order Traversal

[4] Repeat Steps 5 to 7 while PTR!=

NULL: [Backtracking]

[5] Apply PROCESS to PTR->INFO

[6] [Right Child ?] If PTR->RIGHT □ NULL then

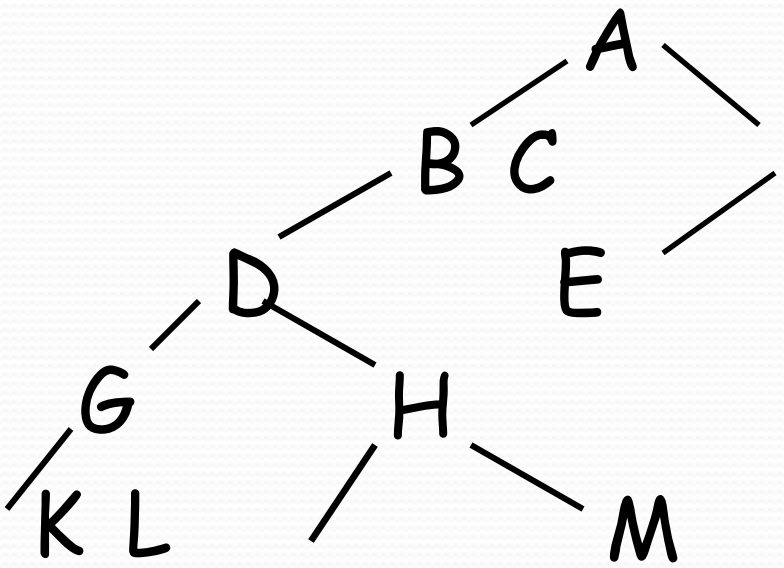
(a) Set PTR = PTR->RIGHT

(b) Go to Step 2

[7] Set PTR = STACK[TOP], TOP = TOP -1

[8] Exit

# In-Order Traversal



[1] Initially Push NULL onto STACK

STACK = φ

Set PTR = A , the root of T

[2] Proceed down the left-most path rooted at PTR = A, pushing the nodes A, B, D, G and K onto STACK:

STACK = φ, A, B, D, G, K

[3] [Backtracking] The nodes K, G and D are popped and processed

STACK = φ, A, B

Set PTR = H [Right Child of D]

[4] Proceed down the left-most path rooted at PTR = H, pushing the nodes H and L onto STACK:

STACK = φ, A, B, H, L

[5] [Backtracking] Nodes L and H are popped and processed

STACK = φ, A, B

Set PTR = M, the Right child of H

[6] Proceed down the left-most path rooted at PTR = M, pushing node M onto STACK

STACK = φ, A, B, M

[7] [Backtracking] Nodes M, B and A are popped and processed

STACK = φ

Set PTR = C, the Right child of A

[8] Proceed down the left-most path rooted at PTR = C, pushing node C and E onto STACK

STACK = φ, C, E

[9] [Backtracking] Nodes E and C are popped and processed.