# Data Structure and Algorithm (CS-102)

Dr. Sambit Bakshi
Dept. of CSE, NIT Rourkela

# Discussed So far

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

# Tree

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

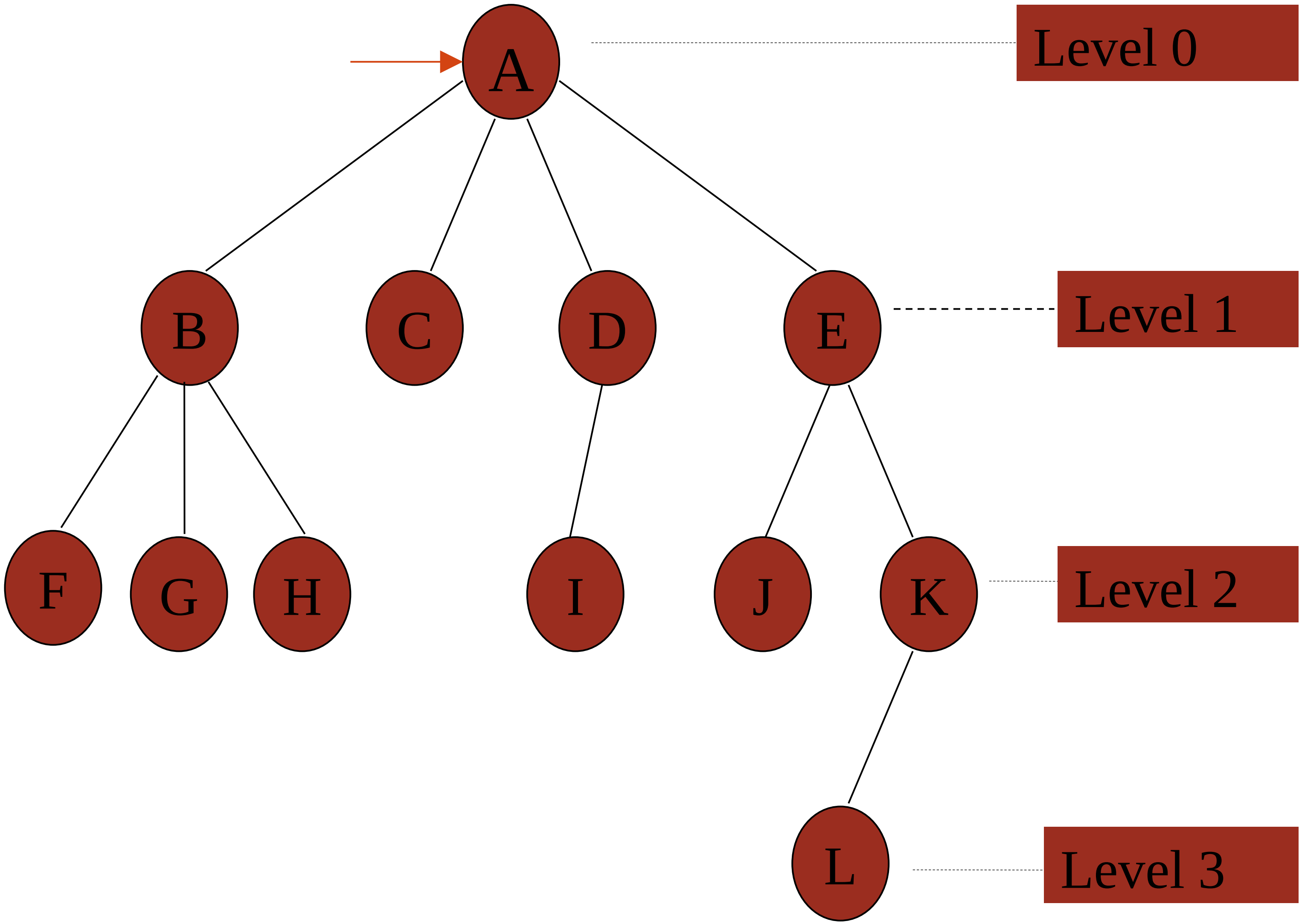
a. There is a specially designated **node** called the **root** and
b. The rest of the nodes could be partitioned into **n** disjoint sets (**n**$\geq$0) each set representing a **tree** $T_i$, i=1,2, . . . n known as **subtree** of the tree.

It is a recursive definition!

# 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 0
Level 1
Level 2
Level 3

# Tree

- Definition of Tree emphasizes on the aspect of

  [a] Connectedness, and

  [b] Absence of closed loops

**Tree** is a undirected graph G(U,V) with all connected nodes and no loops.

But in discussion of computer science, tree is directed and rooted (i.e., one special node is marked as root).

# Basic terminologies

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

# Tree Terminology (continued)

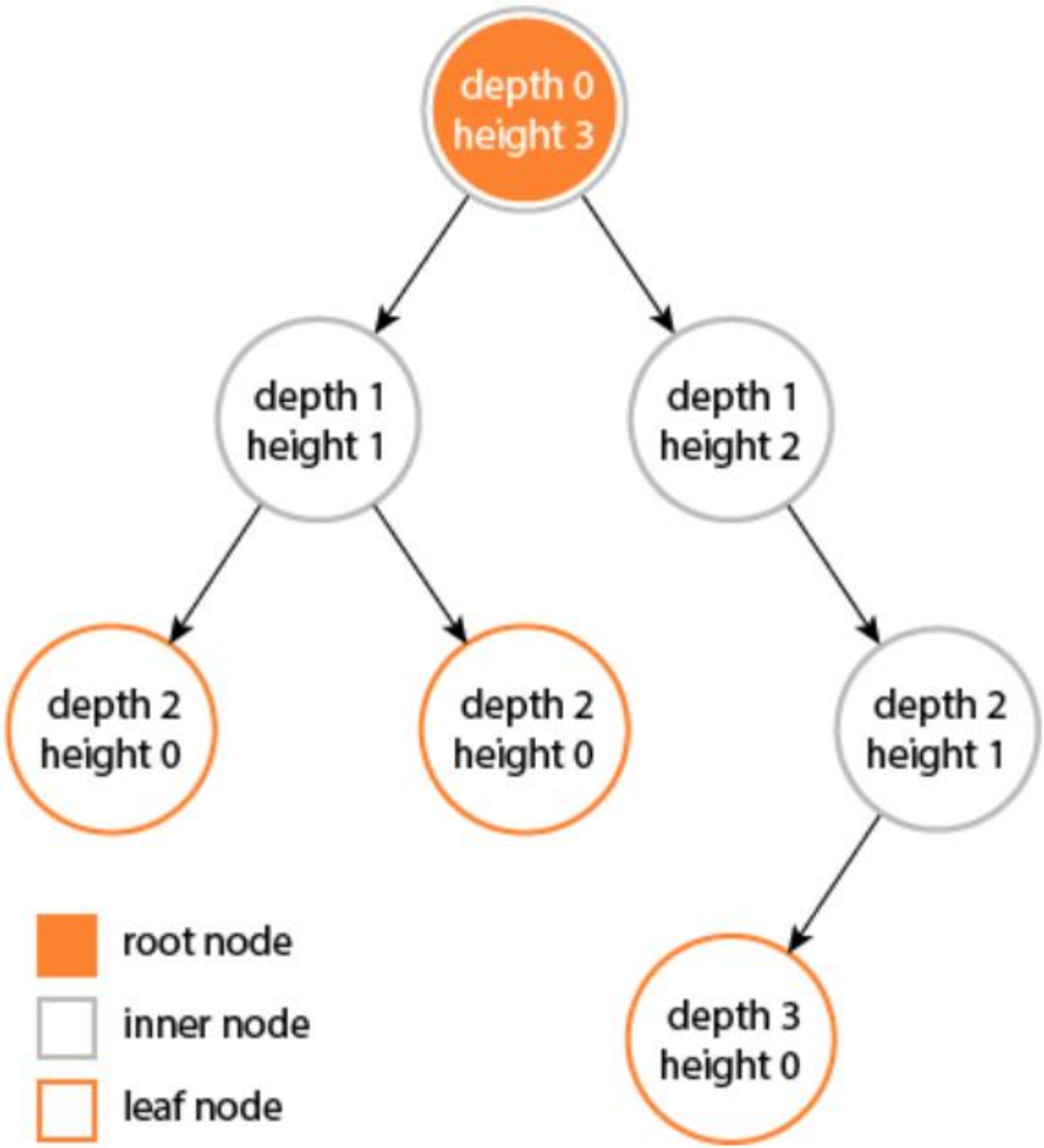
- Node: stores the actual data and links to other nodes
- Parent: immediate predecessor of a node
- Root: specially designated node which has no parent
- Child: immediate successor of a node.

# Tree Terminology(continued)

- Leaf: node without any child
- Level: represents the hierarchy.
  - 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
  - Level of root node is 0.
- Subtree: of a node is a tree whose root is a child of that node

# Tree Terminology (continued)

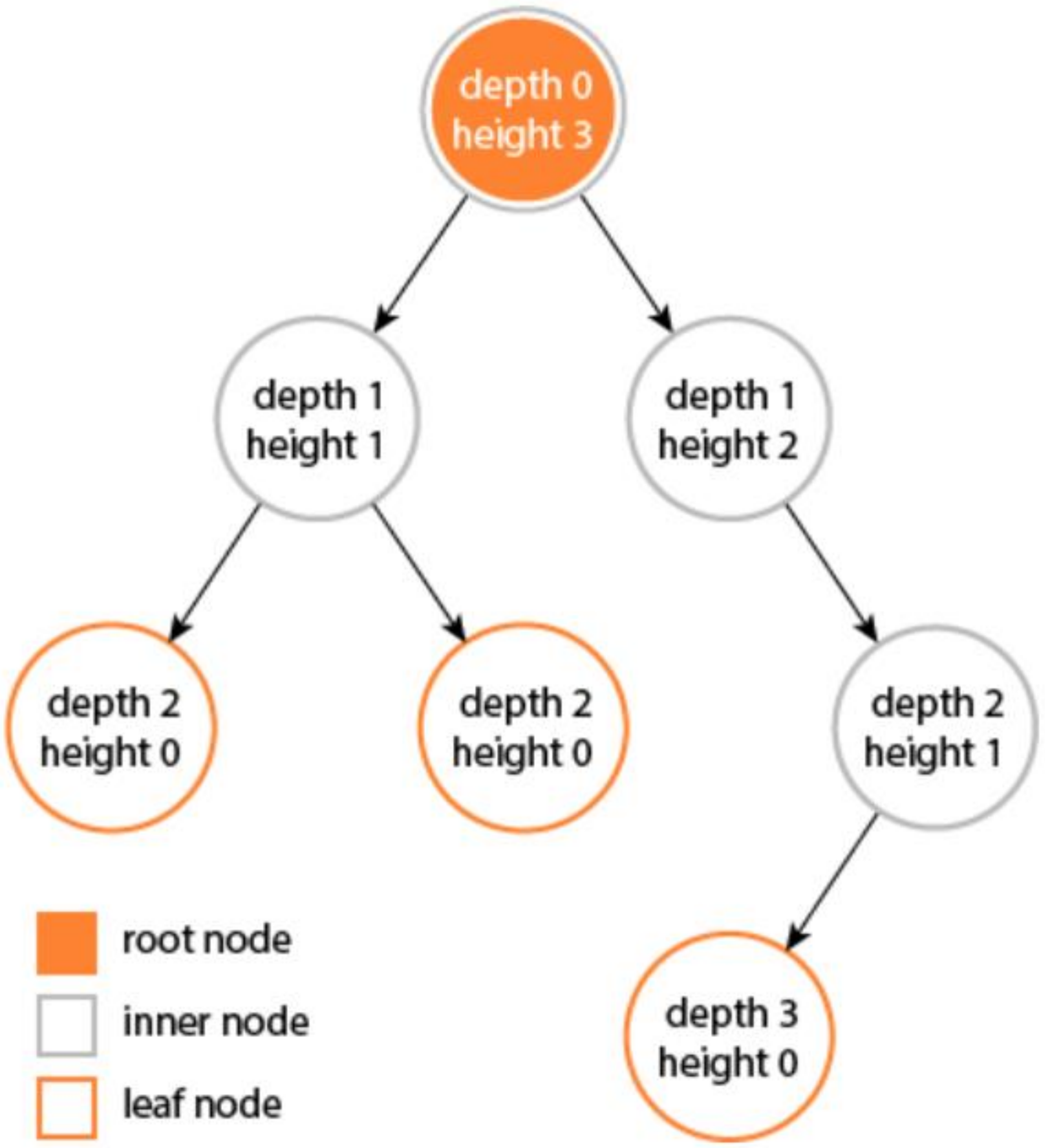
- Height of a node: no of edges in the longest path to a leaf from that node
- Depth of a node: no of edges in the path from root to that node


depth 0
height 3
depth 1
height 1
depth 1
height 2
depth 2
height 0
depth 2
height 0
depth 2
height 1
depth 3
height 0
root node
inner node
leaf node

# Tree Terminology (continued)

- Height of a node: no of ~~edges~~ nodes in the longest path to a leaf from that node
- Depth of a node: no of ~~edges~~ nodes in the path from root to that node

- [Correct definitions]




**Change the values as per the last definition….**

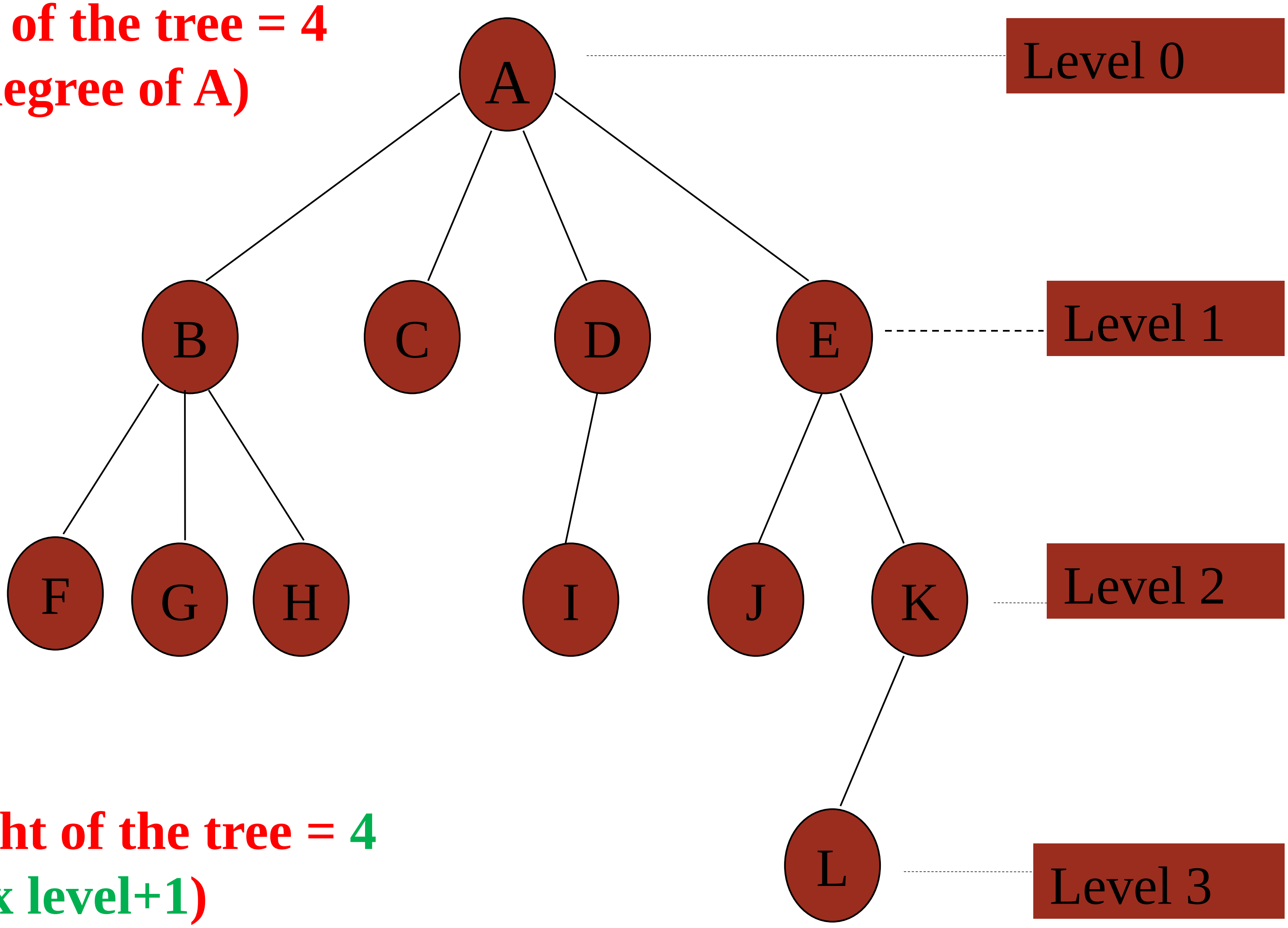
# Tree Terminology (continued)

- Height (depth) of tree: largest no of ~~edges~~ nodes in a path from the root node to a leaf node.
  - Height of a tree given by **$h = l_{max}+1$**, $l_{max}$ is the maximum level of the tree.
- Degree of node: number of children for the node

# Tree Terminology

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

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



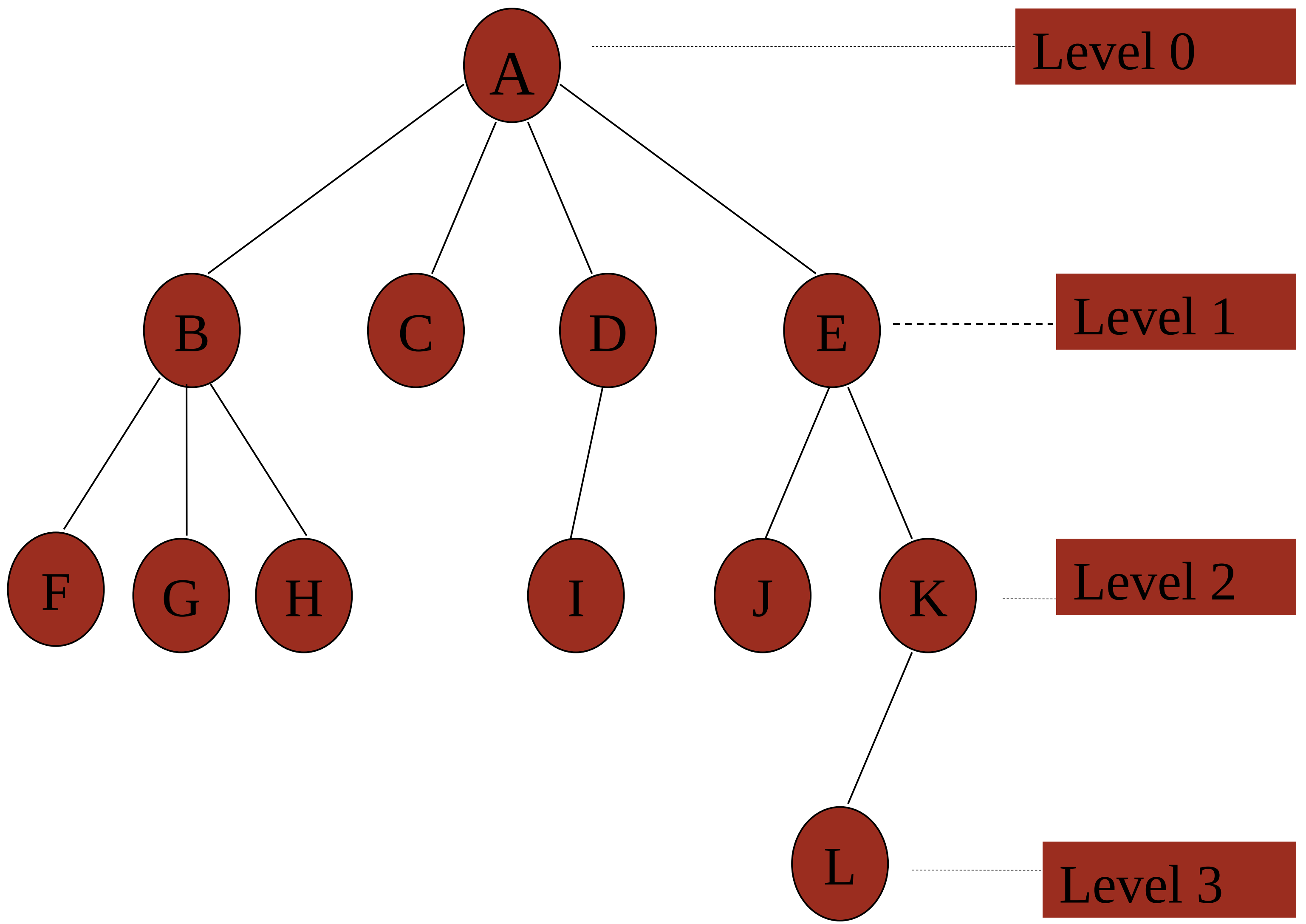

**Height of the tree = 4**
**(Max level+1)**

# Representation of a tree

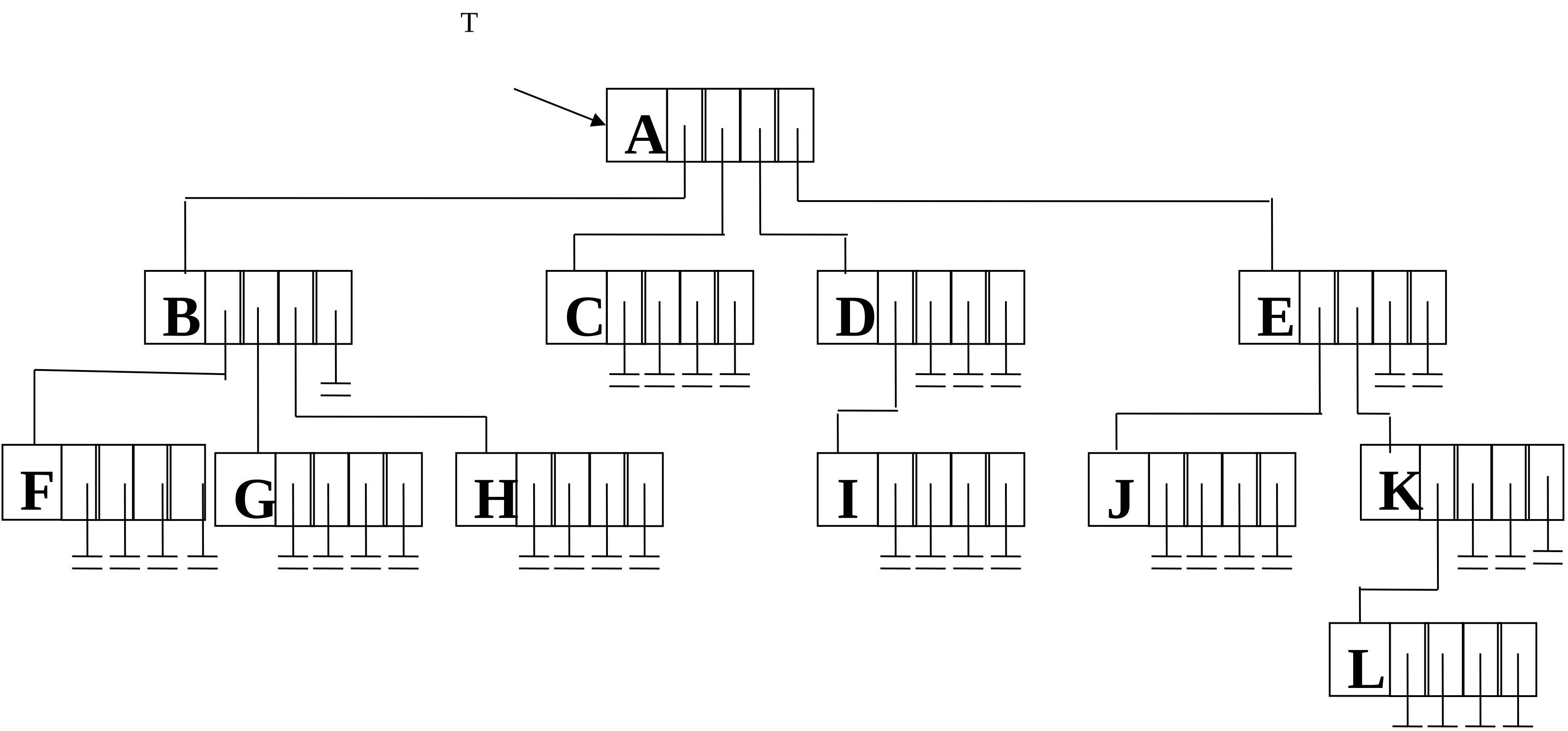
- List Representation

(A (**B**(**F,G,H**), **C**, **D**(**I**), **E**(**J,K**(**L**))) ) for the tree Considered in the Example

- Linked List Representation


A
B
C
D
E
F
G
H
I
J
K
L
Level 0
Level 1
Level 2
Level 3


(a) General node structure




(b) Linked list representation of the tree

# An alternative elegant linked representation

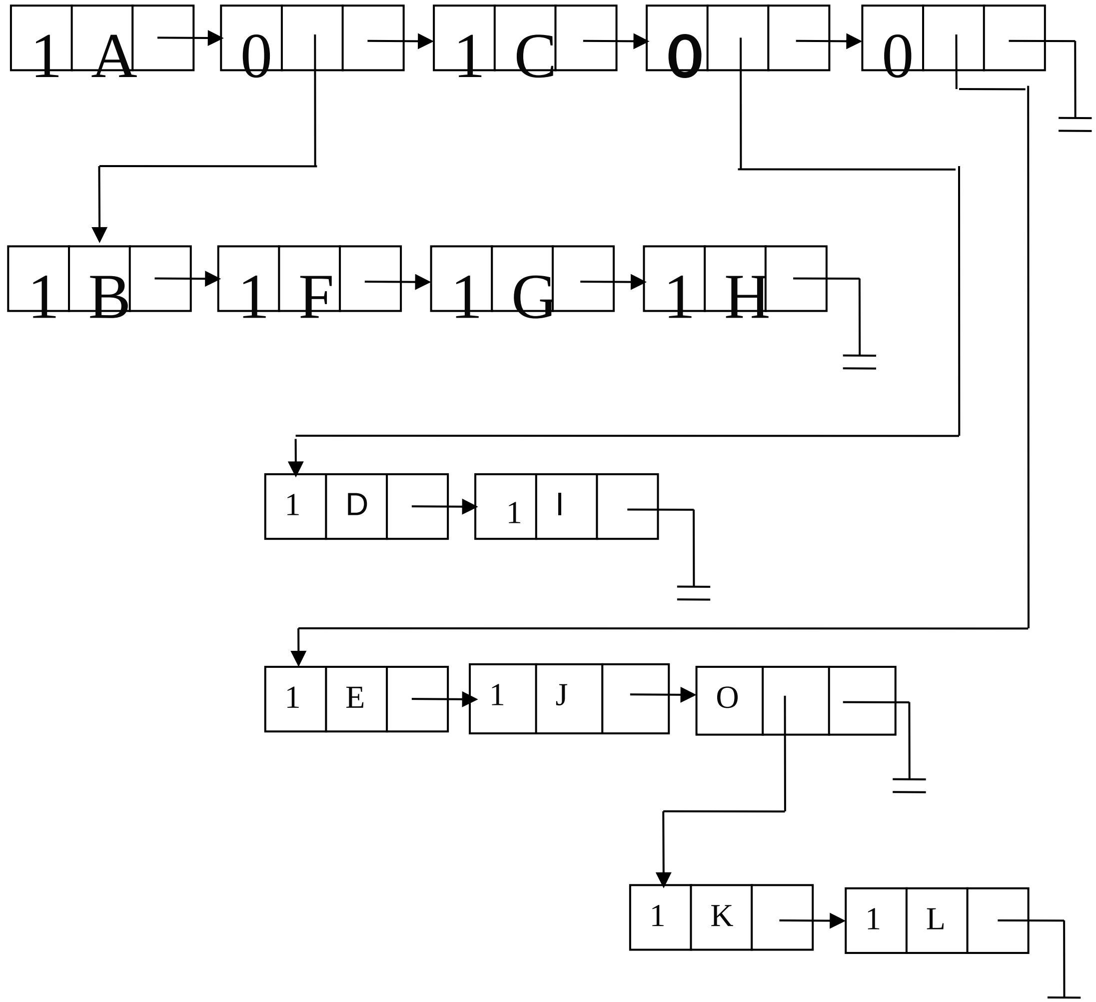
| TAG | DATA / DOWNLINK | LINK |
|---|---|---|

1/0

(a) General node structure

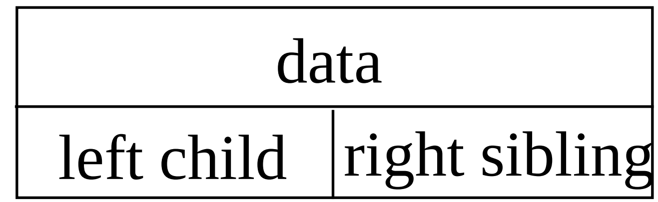
TAG = 1 when next part contains DATA (for root of the subtree and its leaf children),
TAG = 0 when next part contains DOWNLINK (for non-leaf children)

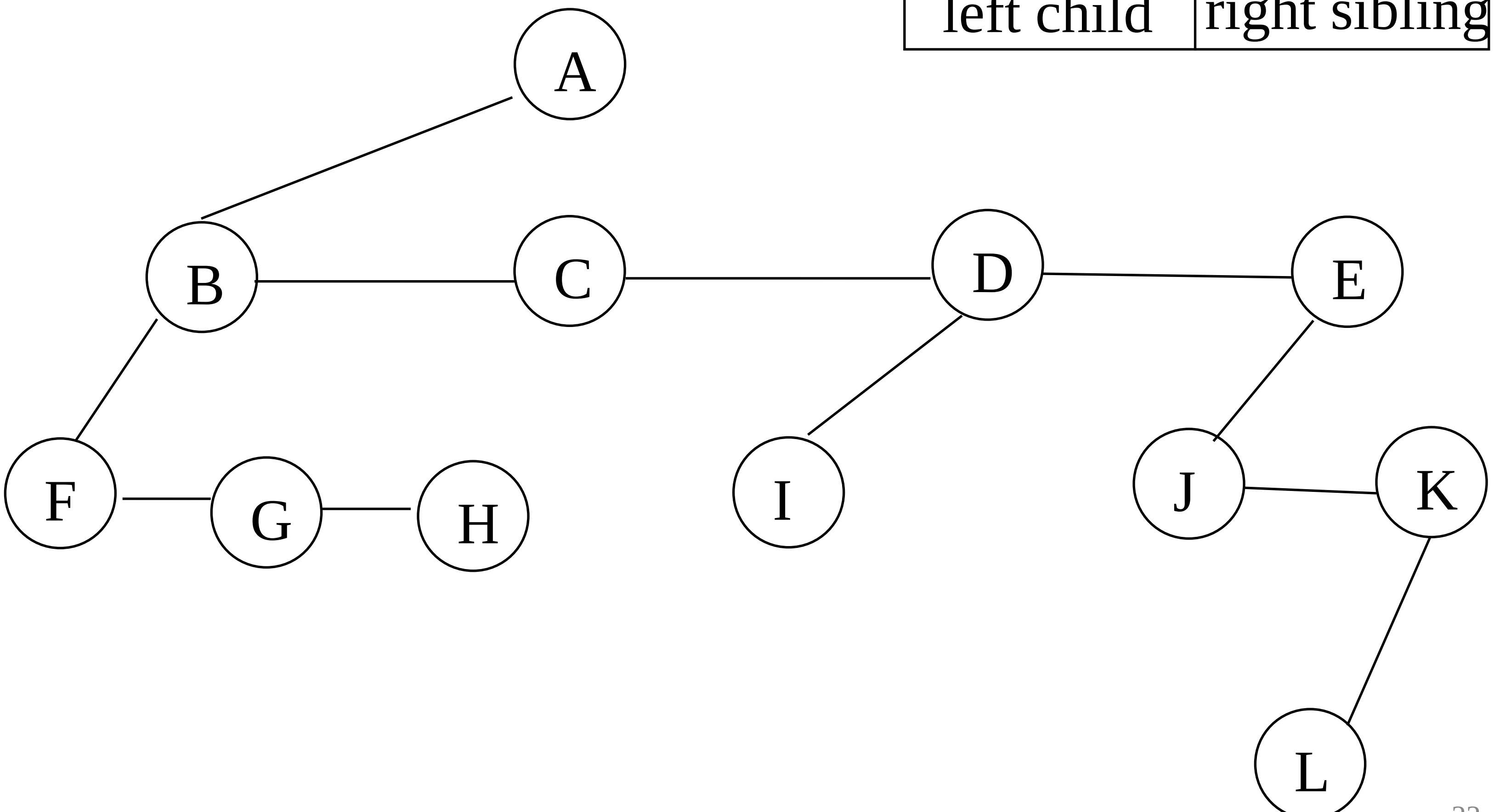
# An alternative elegant linked representation




(b) Linked representation of the tree

# Left Child - Right Sibling representation



# Binary Trees

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

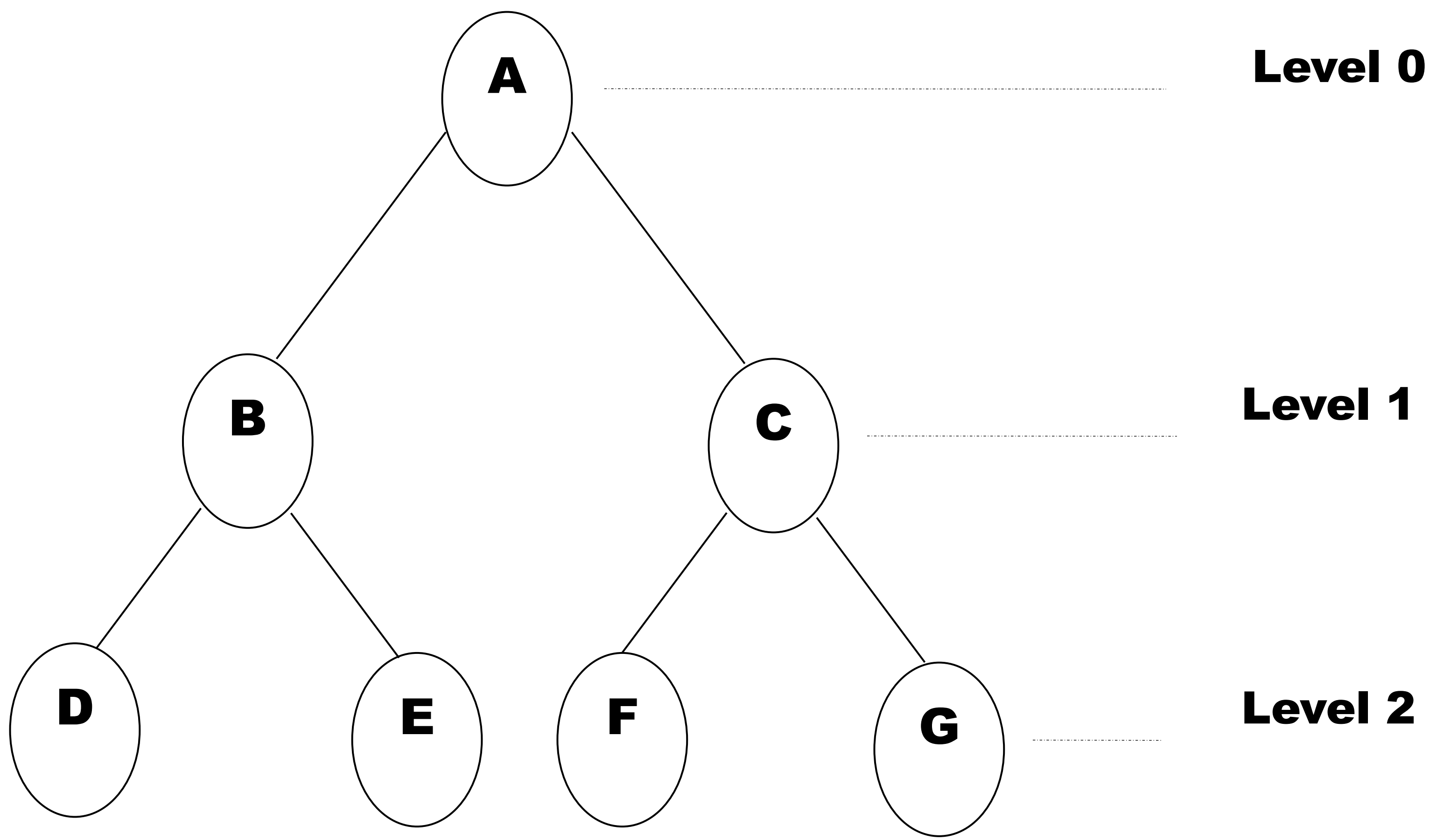
[a] T is empty or bears zero nodes (Called the Null tree or Empty tree) or

[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**.

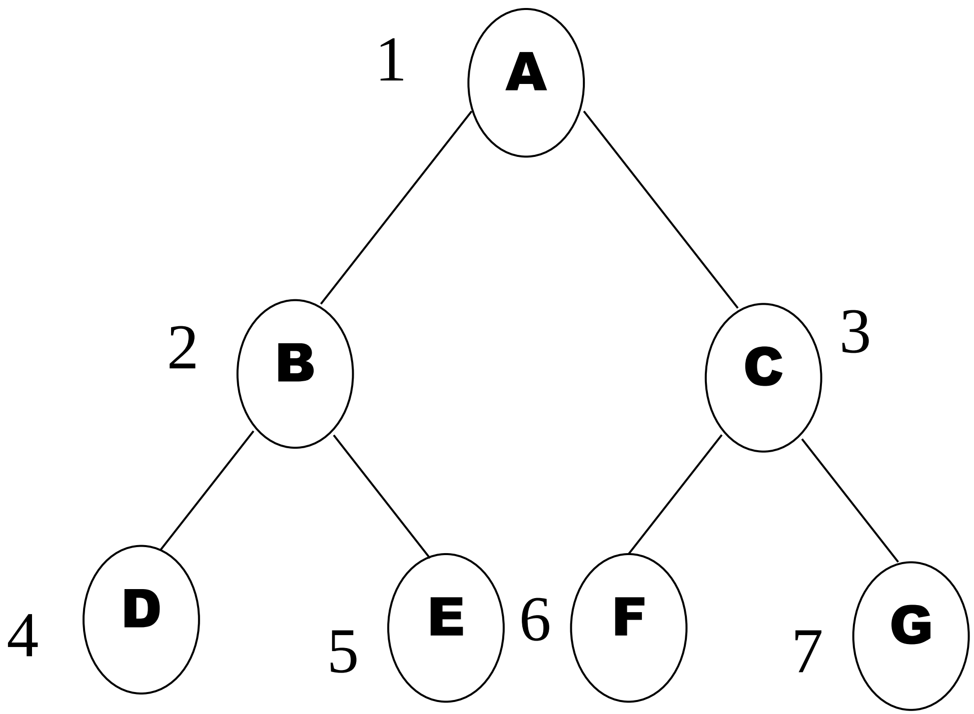
- A binary tree can therefore be **empty** or consist of a root node and two disjoint binary trees termed **left subtree** and **right subtree**.


A
B
C
D
E
F
G
Level 0
Level 1
Level 2

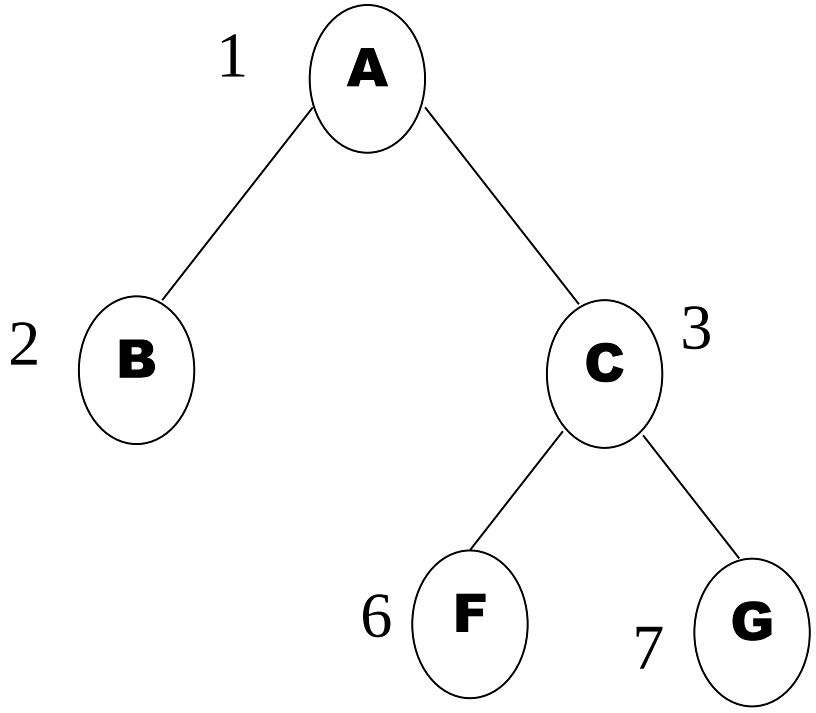
Important observations regarding binary trees:

- The maximum number of nodes on level **i** of a binary tree is $2^i$, $\mathbf{i \geq 0}$
- The maximum number of nodes in a binary tree of height **h** is $\mathbf{2^h - 1, h \geq 1}$
  - $\mathbf{2^0+2^1+2^2+\ldots+2^{h-1}=2^h-1}$
- For any non empty binary tree, if $\mathbf{t_o}$ is the number of terminal nodes and $\mathbf{t_2}$ is the number of nodes of degree 2, then $\mathbf{t_o=t_2+1}$

A binary tree of height **h** which has all its permissible maximum number of nodes viz., **$2^h$-1 i**ntact is known as a **perfect binary tree of height h.**



A binary tree of height **h** which has all nodes with either 0 or 2 children is known as a **strict / full** binary tree of height h.



Theorem: Let T be a nonempty, full binary tree Then:

(a) If T has I internal nodes, the number of leaves is $L = I + 1$.

(b) If T has I internal nodes, the total number of nodes is $N = 2I + 1$.

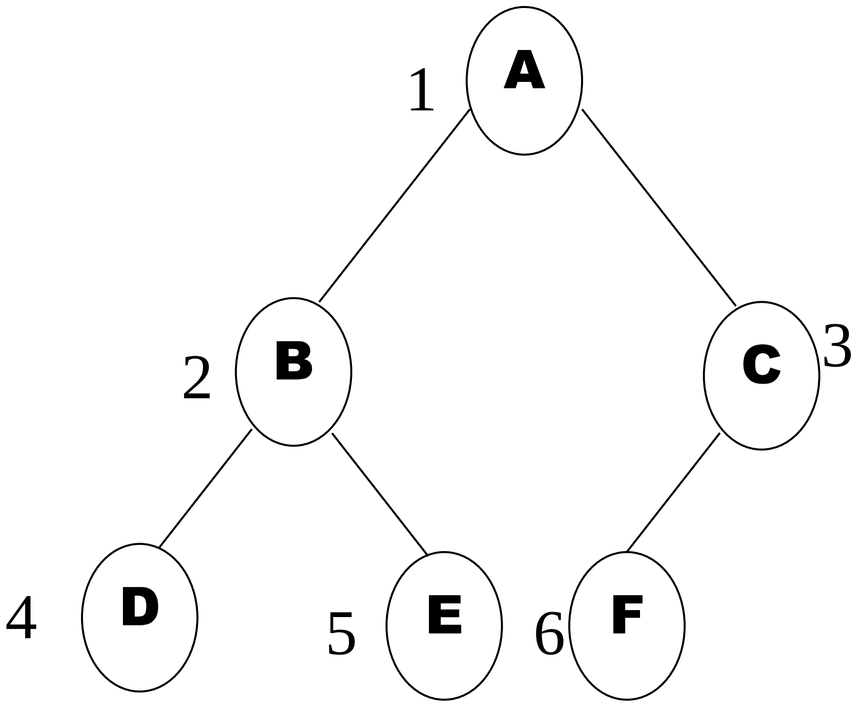
(c) If T has a total of N nodes, the number of internal nodes is $I = (N - 1)/2$.

(d) If T has a total of N nodes, the number of leaves is $L = (N + 1)/2$.

(e) If T has L leaves, the total number of nodes is $N = 2L - 1$.

(f) If T has L leaves, the number of internal nodes is $I = L - 1$.

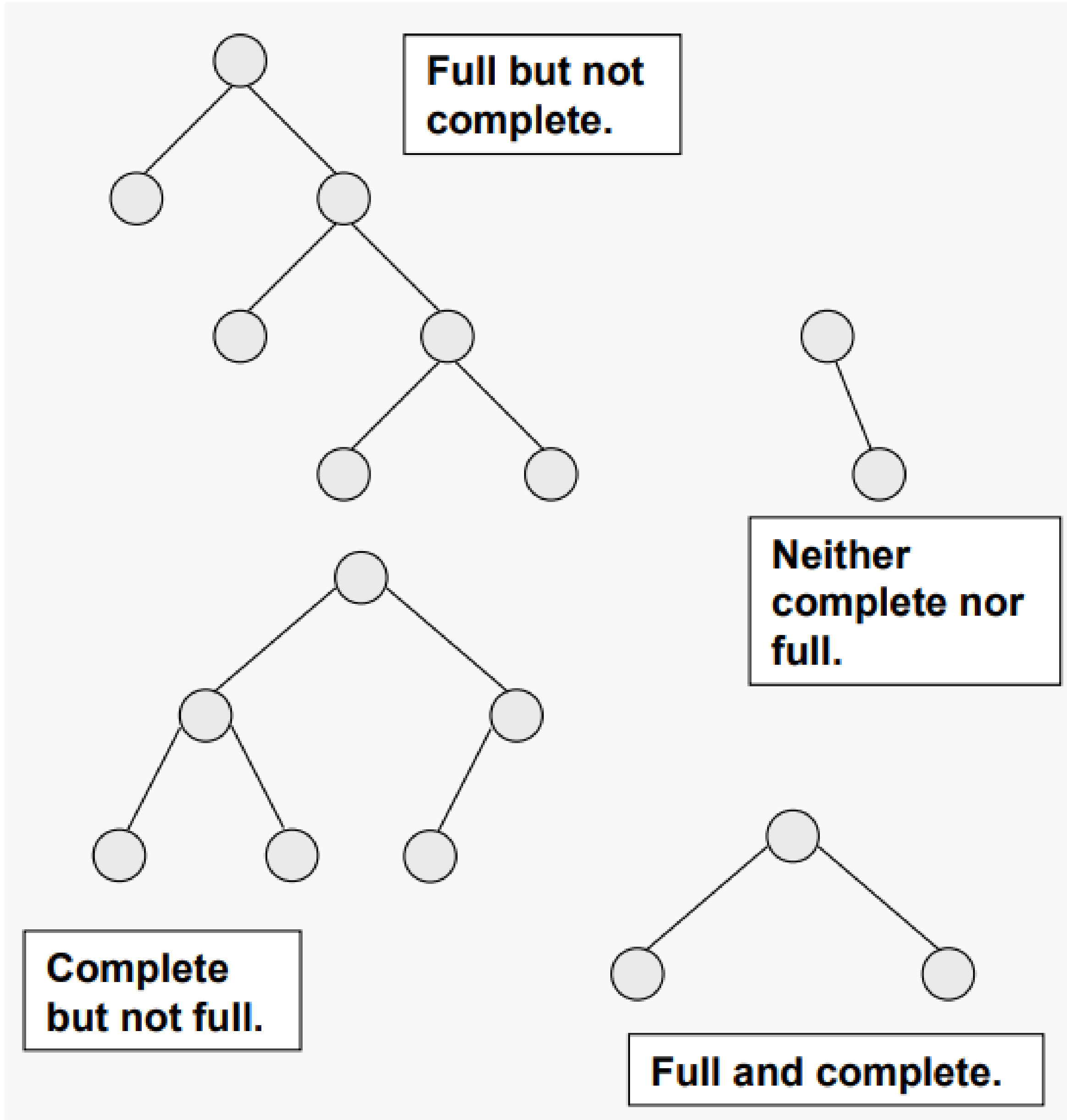
A binary tree with **n'** nodes and height **h** is **complete** if its node numbers correspond to the node numbers **1** to **n** (**n' ≤ n**) in a perfect binary tree of height **h**.




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

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

- In a **complete** binary tree, every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left as possible.


Full but not complete.
Neither complete nor full.
Complete but not full.
Full and complete.

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.

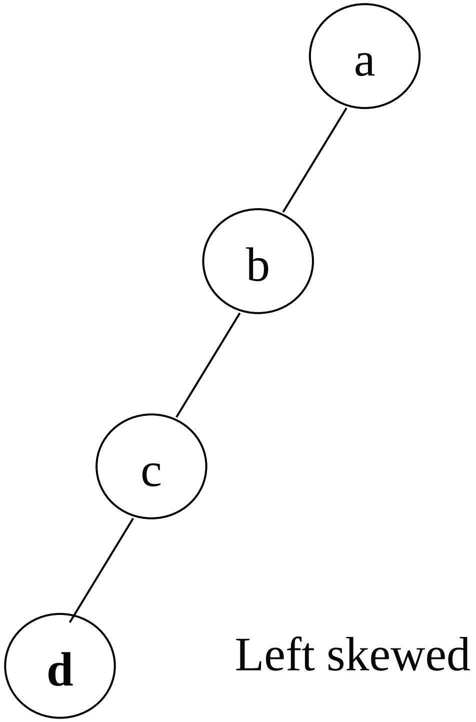
[b] If a parent node has a number i, then its right child has the number 2i+1 ($2i + 1 \leq n$).
-- If $2i + 1 > n$ then i has no right child.

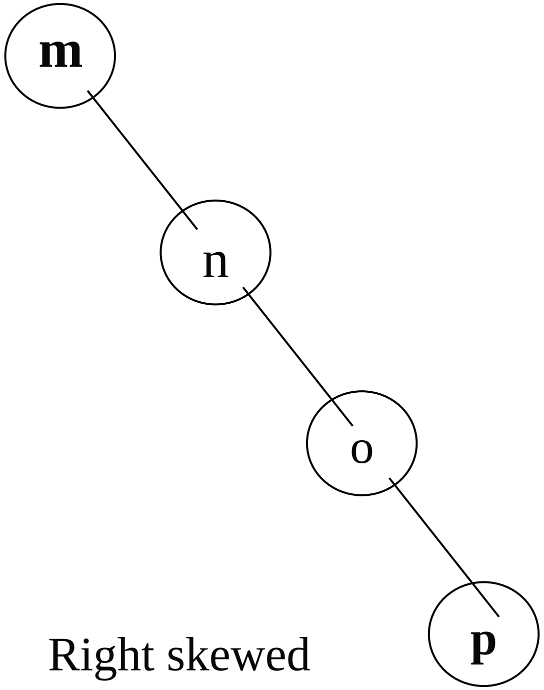
[c] If a child node (left or right) has a number i then the parent node has the number $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$ then **i** is the root and hence has no parent.

- Is every full binary tree a complete binary tree?

- Is every complete binary tree a full binary tree?

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

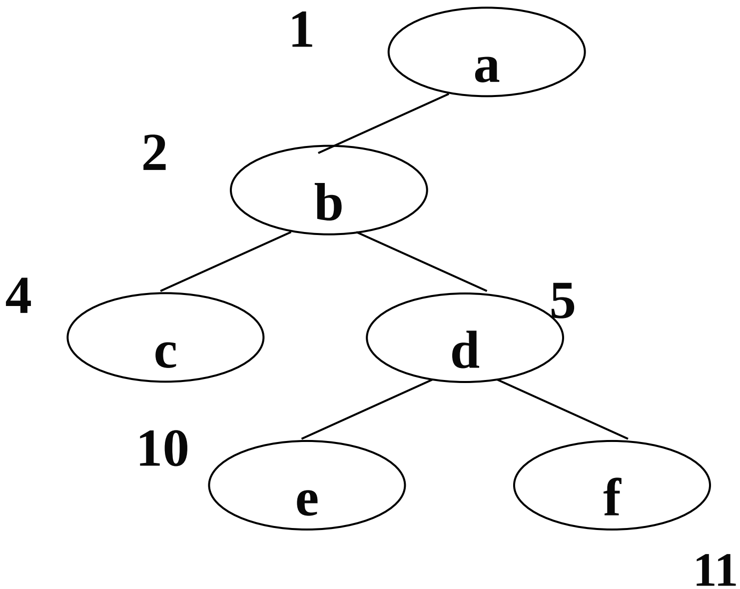
Binary tree can be represented by means of

[a] Array

[b] linked list

# Representation Of Binary Trees

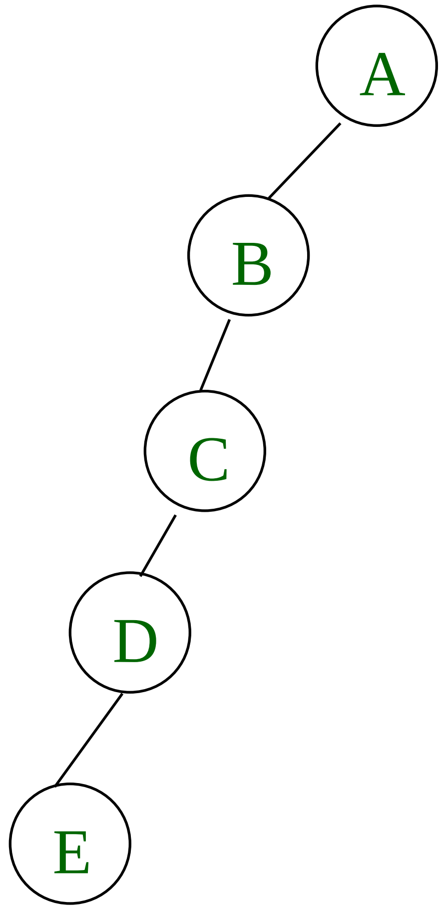
## Array Representation




Sequential representation of a tree with depth **d** will require an array with approx **$2^d$-1** elements

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

# Array Representation

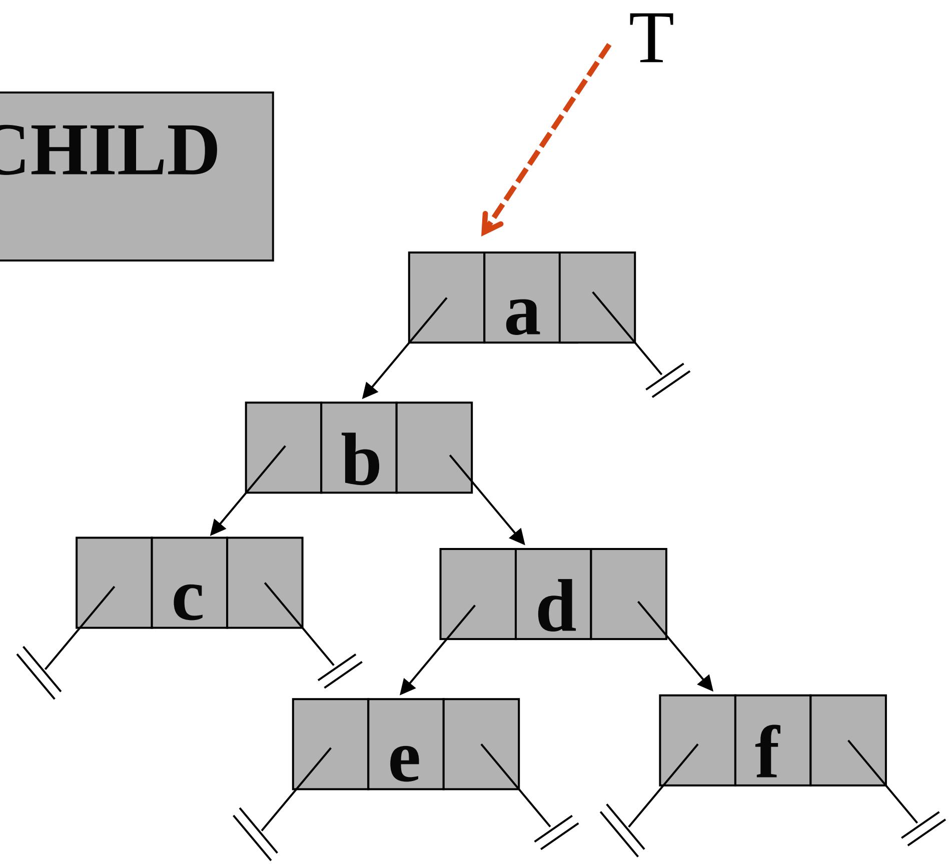



| | |
|---|---|
| [1] | A |
| [2] | B |
| [3] | -- |
| [4] | C |
| [5] | -- |
| [6] | -- |
| [7] | -- |
| [8] | D |
| [9] | -- |
| . | . |
| [16] | E |

Memory wastage !!

# 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**

# 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)

# Traversing Binary Tree

## In-order

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

[b] Process the root **R**

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

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

# Traversing Binary Tree

## Post-order

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

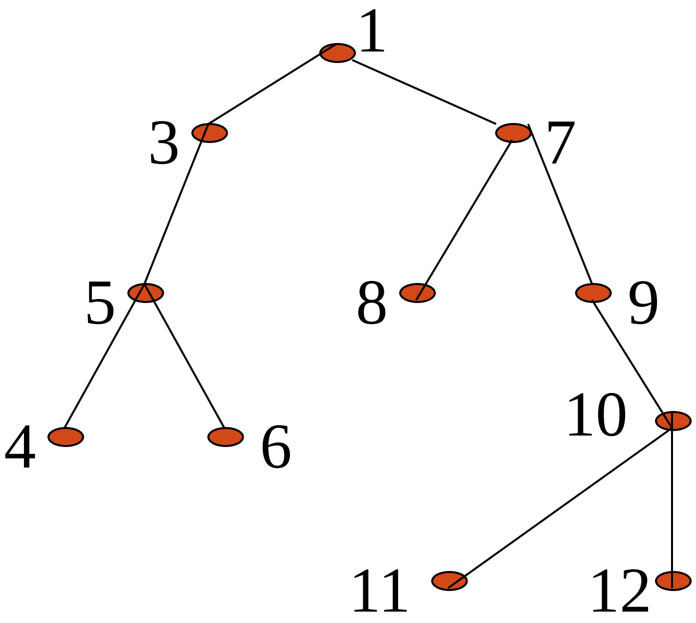
[b] Traverse the right sub-tree of **R** in post-order

[c] Process the root **R**

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

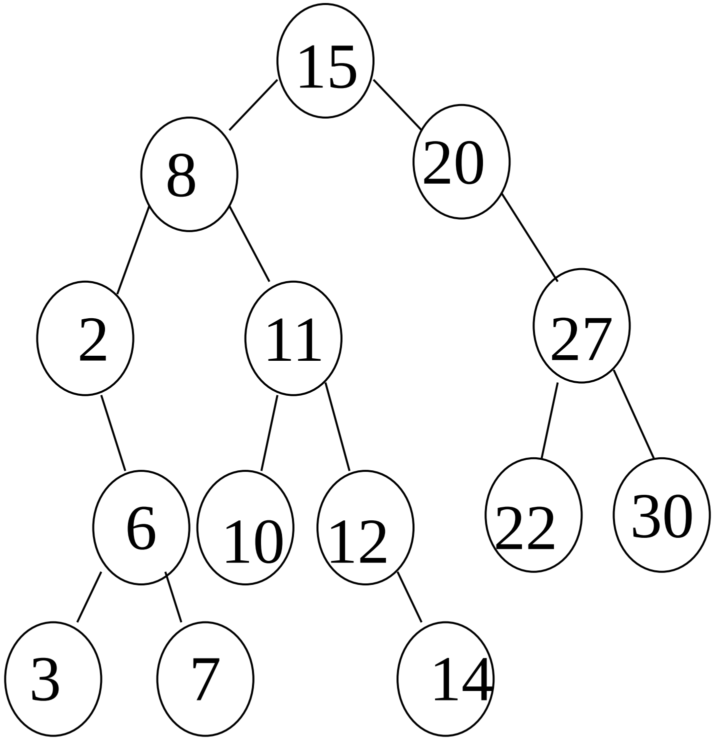
# Illustrations for Traversals

- Assume: visiting a node is printing its <u>label</u>
- **Preorder**:
  1 3 5 4 6 7 8 9 10 11 12
- **Inorder**:
  4 5 6 3 1 8 7 9 11 10 12
- **Postorder**:
  4 6 5 3 8 11 12 10 9 7 1



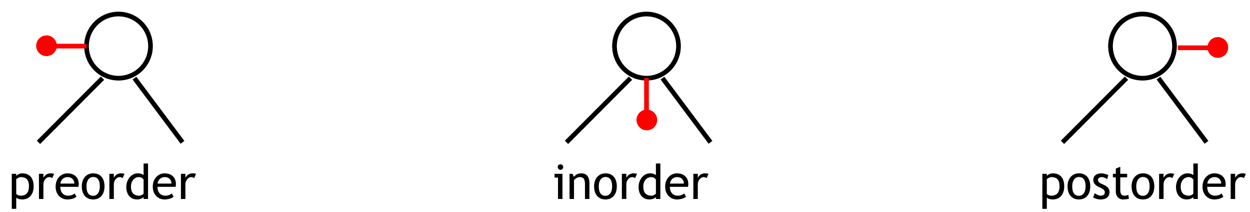
# Illustrations for Traversals (Contd.)

- Assume: visiting a node is printing its <u>data</u>
- **Preorder**: 15 8 2 6 3 7 11 10 12 14 20 27 22 30
- **Inorder**: 2 3 6 7 8 10 11 12 14 15 20 22 27 30
- **Postorder**: 3 7 6 2 10 14 12 11 8 22 30 27 20 15

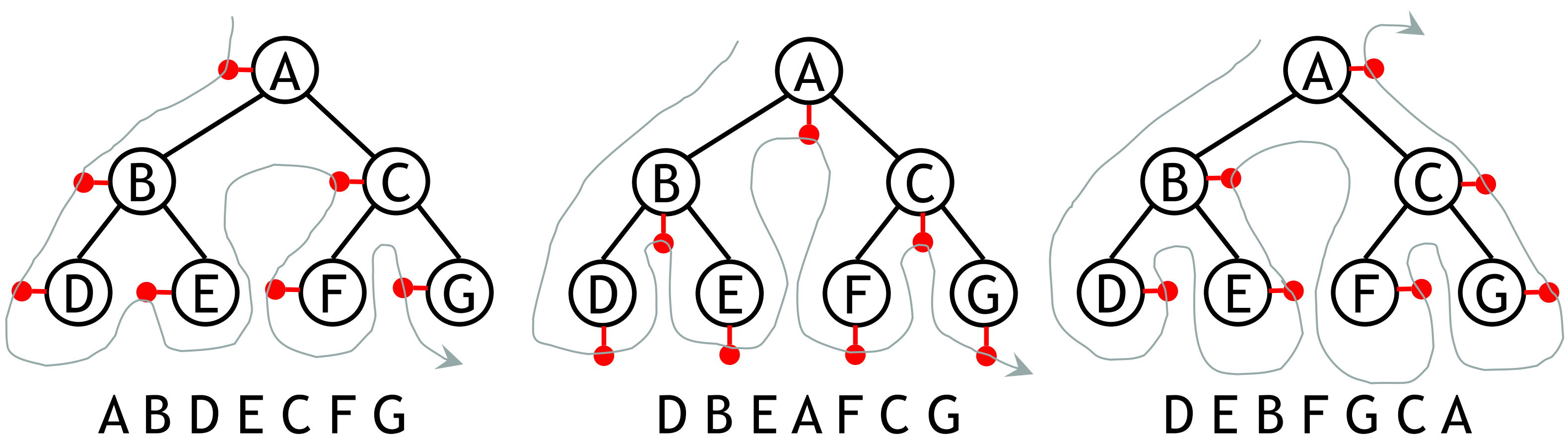


# Euler's Tree traversals using "flags"

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:




- To traverse the tree, collect the flags:



# Formulation of Binary tree from Its traversal

Easy if one given traversal sequence in inorder.

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 from inorder.

3. Same technique can be applied repeatedly to form subtrees

# 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

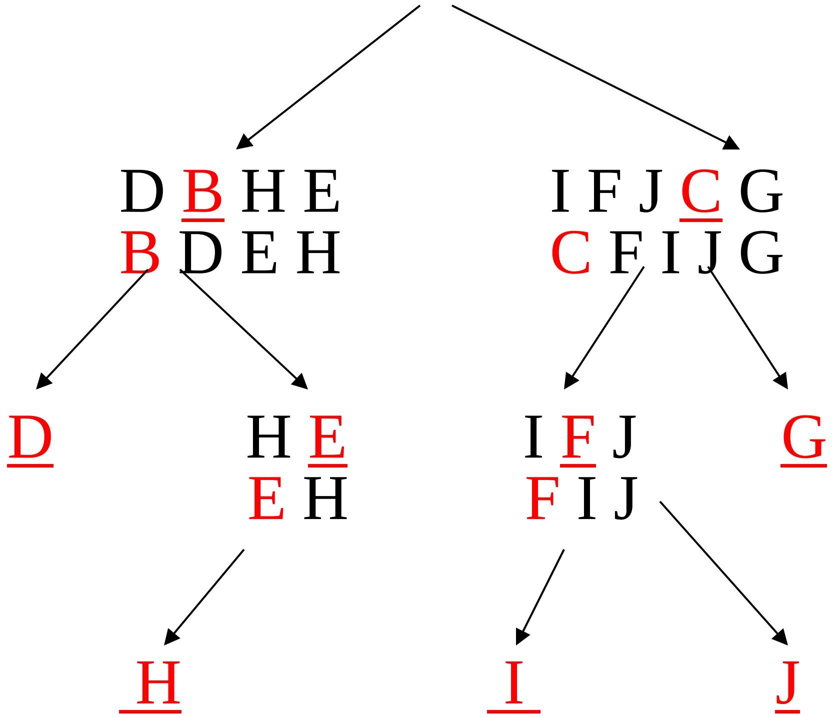
Left subtree: D B H E

Right subtree: I F J C G

# continued…

In: D B H E A I F J CG

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

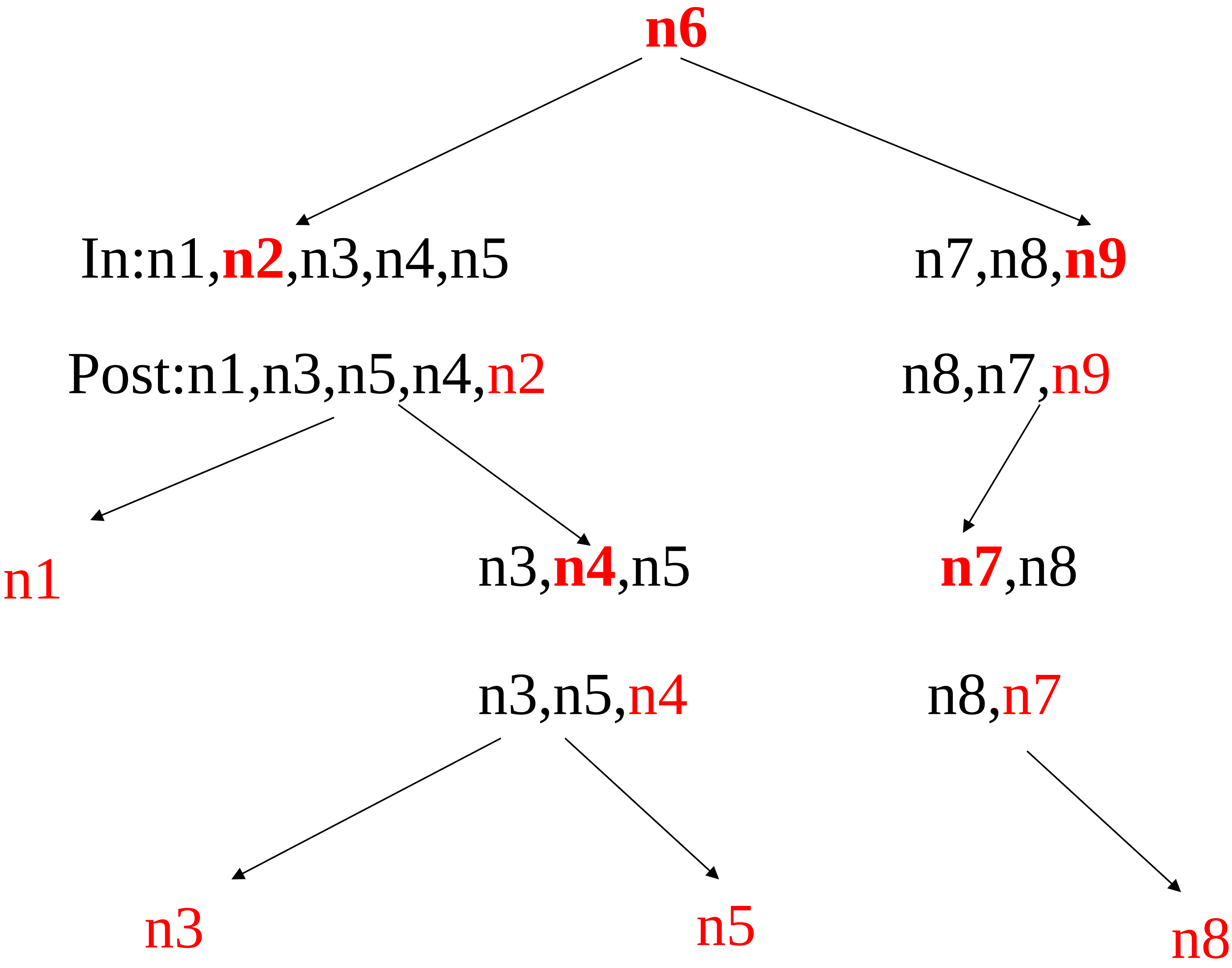


# Example: For Given Inorder and Postorder

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

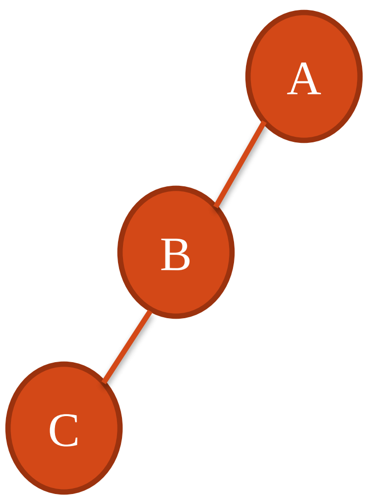

n6
In:n1,n2,n3,n4,n5
Post:n1,n3,n5,n4,n2
n7,n8,n9
n8,n7,n9
n1
n3,n4,n5
n3,n5,n4
n7,n8
n8,n7
n3
n5
n8

# Given Preorder & Postorder

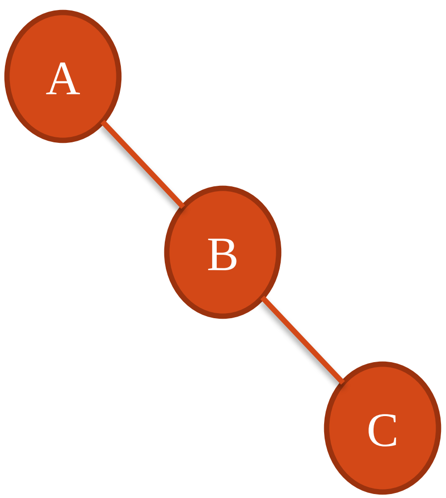
- Tree may not be unique

    preorder: A B C

    postorder: C B A




Tree-1

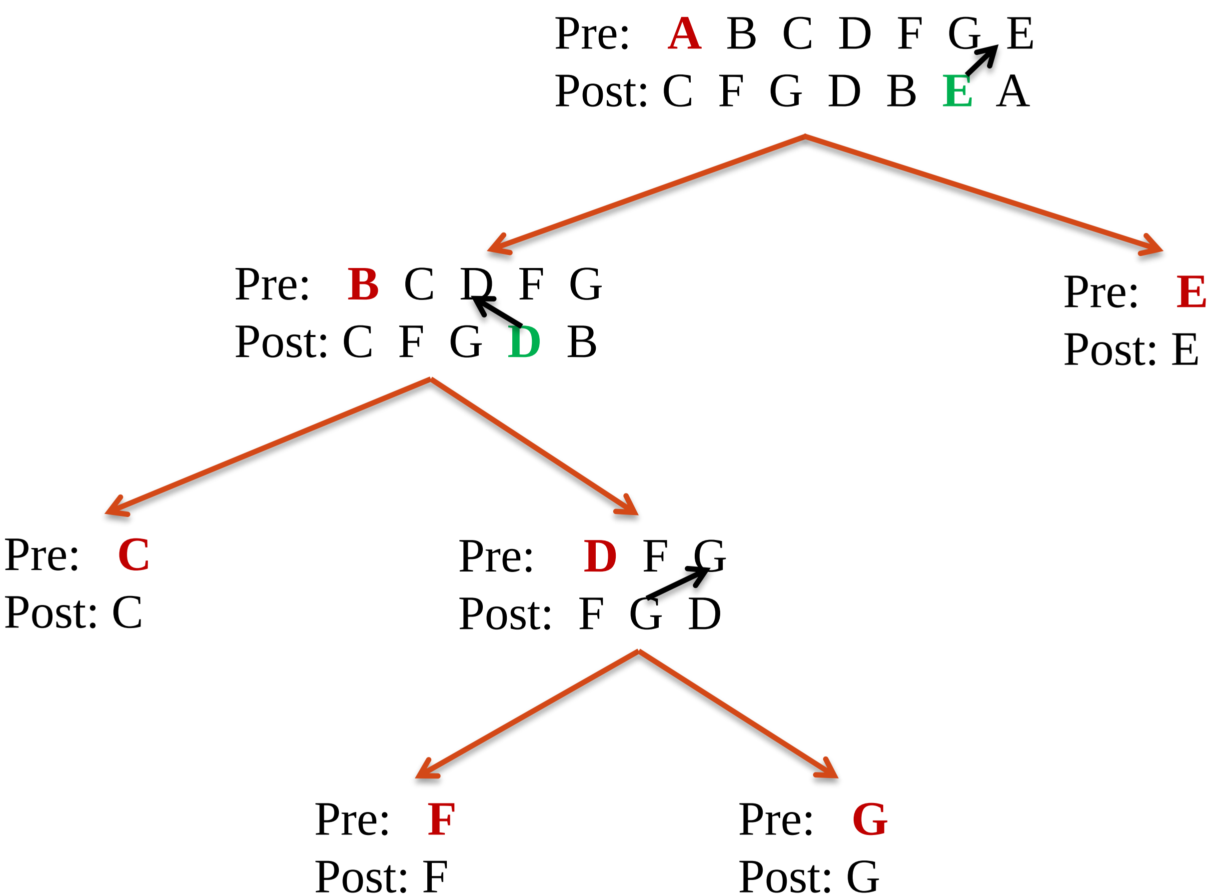



Tree-2

# Given Preorder & Postorder

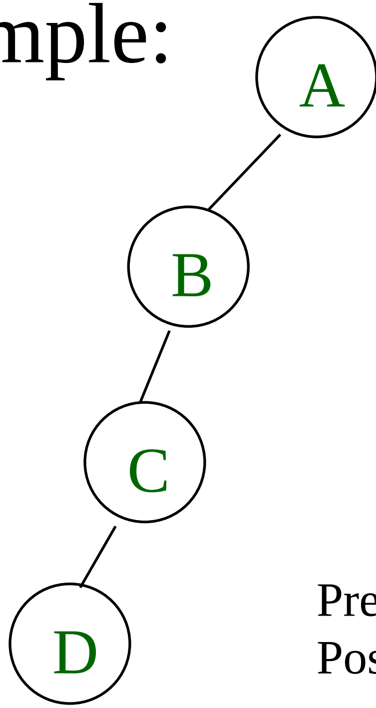
1. First node in preorder is ROOT (same as last node in postorder)
2. Find previous node of ROOT in postorder (say X) and locate it in preorder
3. Nodes before X in preorder is the left subtree of ROOT
4. X and nodes after X in preorder is the right subtree of ROOT
5. Repeat stepts 1 to 5 until each subtree contains one element

# Given Preorder & Postorder

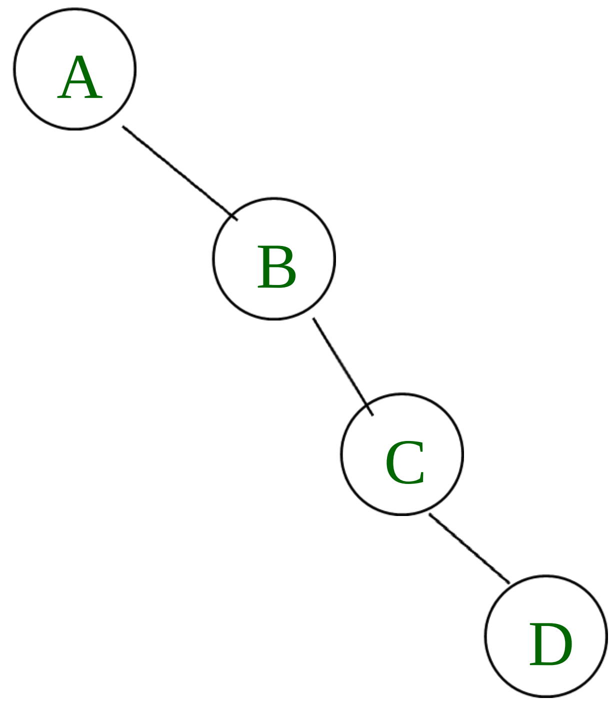


- Given pre-order and post order traversal of a tree, can you accurately generate back the tree?

- Answer: Not always
- Example:




Pre: A B C D
Post: D C B A



The pre-order and post order traversal of a Binary Tree generates the same output. The tree can have maximum

- A One node
- B Two nodes
- C Three nodes
- D Any number of nodes

Option: [A]

# Traversal Algorithm Using Stack

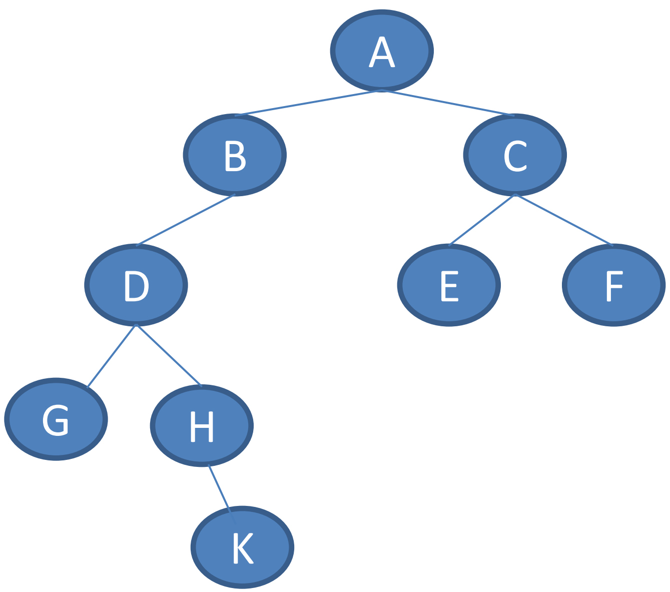
- Binary Tree is represented by

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

- A pointer **PTR** will contain the location of the node N currently being scanned.
- An array **STACK** will hold the addresses of the node for future processing

# Pre-order tree traversal with a stack

1. Push root onto the stack
2. While stack is not empty
   – Pop a vertex off stack, and write it to the output list
   – Push its children right-to-left onto stack




| ***Step*** | ***Output*** | ***Stack*** |
|---|---|---|
| **0** | | **<u>A</u>** |
| **1** | **A** | **C,<u>B</u>** |
| **2** | **B** | **C,<u>D</u>** |
| **3** | **D** | **C,H,<u>G</u>** |
| **4** | **G** | **C,<u>H</u>** |
| **5** | **H** | **C,<u>K</u>** |
| **6** | **K** | **<u>C</u>** |
| **7** | **C** | **F,<u>E</u>** |
| **8** | **E** | **<u>F</u>** |
| **9** | **F** | **--** |

# In-order Traversal with a stack

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

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

[2] Repeat while PTR ≠ NULL [Push 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]

[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 with a stack

[1] Push the Left-most path from ROOT onto STACK

[2] While STACK is not empty

(a) Pop and process node X

(b) If Right Child of X (say Y) exists then

Push left most path from Y onto STACK

# In-order Traversal with a stack

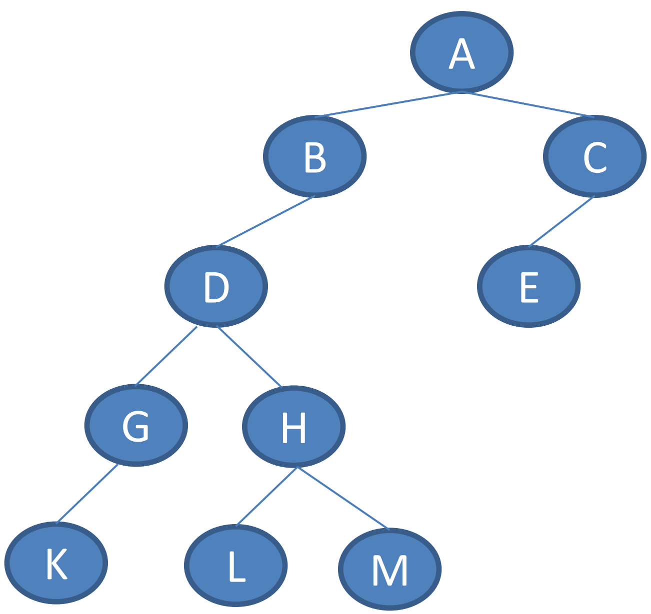
[1] Push the Left-most path from ROOT onto STACK

[2] While STACK is not empty

(a) Pop and process node X

(b) If Right Child of X (say Y) exists then

Push left most **yet unprocessed** path from Y onto STACK




| *Step* | *Output* | *Stack* |
|---|---|---|
| **1** | | **A,B,D,G,<u>K</u>** |
| **2** | **K** | **A,B,D,<u>G</u>** |
| **3** | **G** | **A,B,<u>D</u>** |
| **4** | **D** | **A,B,H,<u>L</u>** |
| **5** | **L** | **A,B,<u>H</u>** |
| **6** | **H** | **A,B,<u>M</u>** |
| **7** | **M** | **A,<u>B</u>** |
| **8** | **B** | **<u>A</u>** |
| **9** | **A** | **C,<u>E</u>** |
| **10** | **E** | **<u>C</u>** |
| **11** | **C** | **--** |

## **Self Study**

Write an algorithm to traverse a binary tree in **postorder** traversal using stack. Discuss the algorithm with an example.