

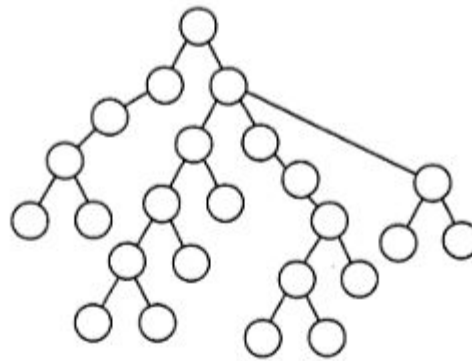
# DATA STRUCTURE

TREES



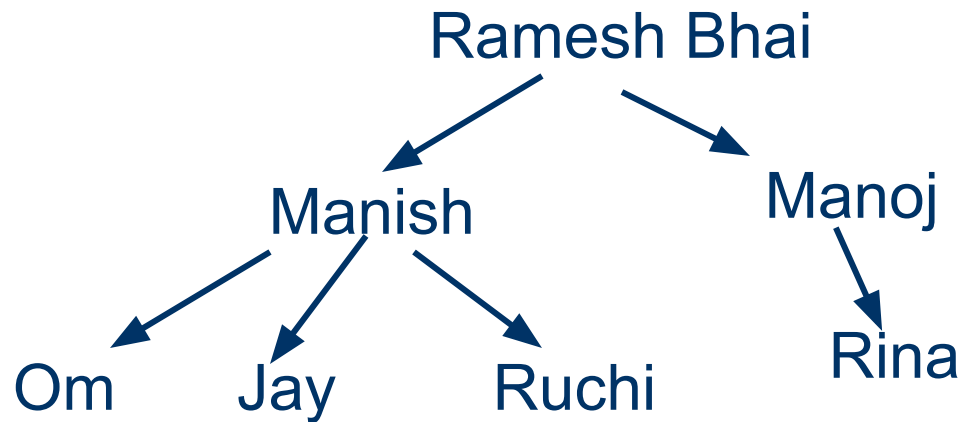
# Non Linear Data Structure

- Non linear data structure means elements appear in non linear fashion.
- Ex : Tree and Graph



# TREE

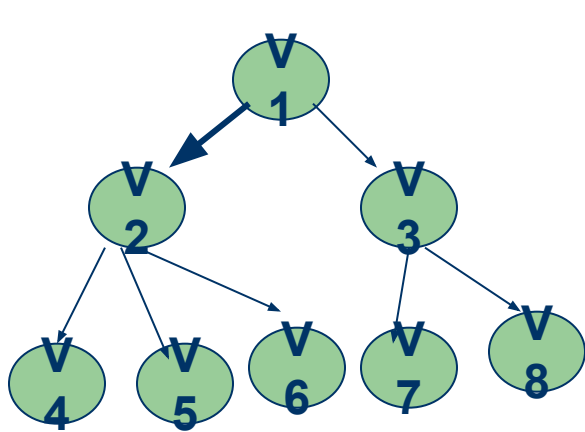
- Mainly used to represent data containing a hierarchical relationship between elements.
- Ex. – family tree



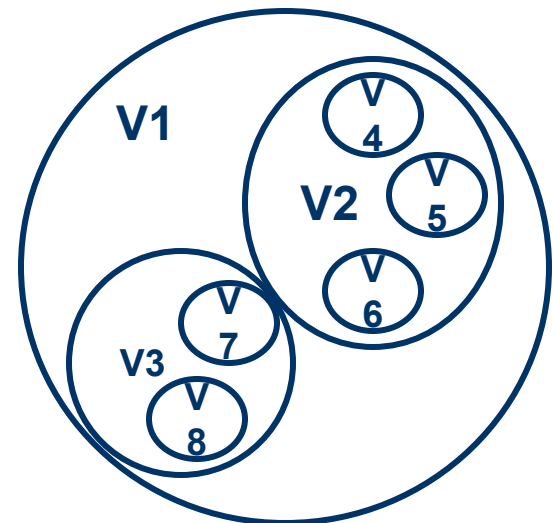
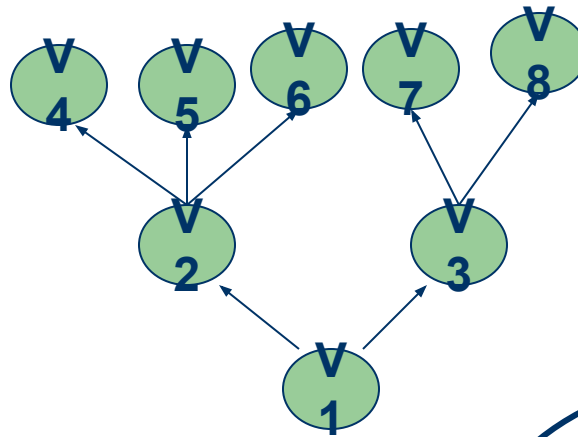
# Tree:

- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.
- One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root

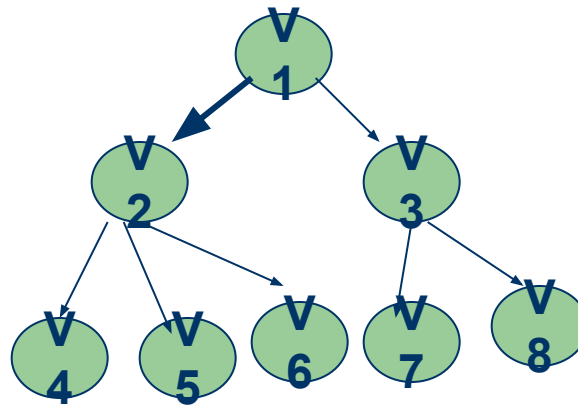
# Different Representation of Tree



$(V1(V2(V4)(V5)(V6))(V3(V7)(V8)))$



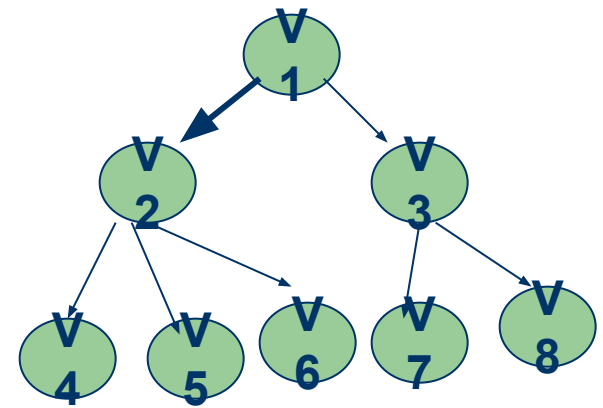
# Basic Terminologies



# Basic Terminologies

- **NODE:** A node of a tree stores the actual data and links to the other node.
  - Ex: v1,v2,v3,v4,v5,v6 etc are nodes.
- **Parent :** Parent of a node is a immediate predecessor of a node.
  - Ex: v1 is a parent of v2 and v3
- **Child:** All immediate successors of a node are known as children of that node.
  - Ex: v2 and v3 are children of node v1.

# Basic Terminologies



- Root:
  - This is a specially designated node which has no parent.
  - Ex: v1 is root node.
- Leaf:
  - The node which is at the end and does not have any child is called leaf.
  - Ex: v4,v5,v6,v7,v8



# Basic Terminologies

- Level
  - Level is the rank in the hierarchy.
  - The root node has level 0.
  - Children of the root node are at level number 1.
  - Thus, every node is at one level higher than its parent.
  - So, all child nodes have a level number given by parent's level number + 1

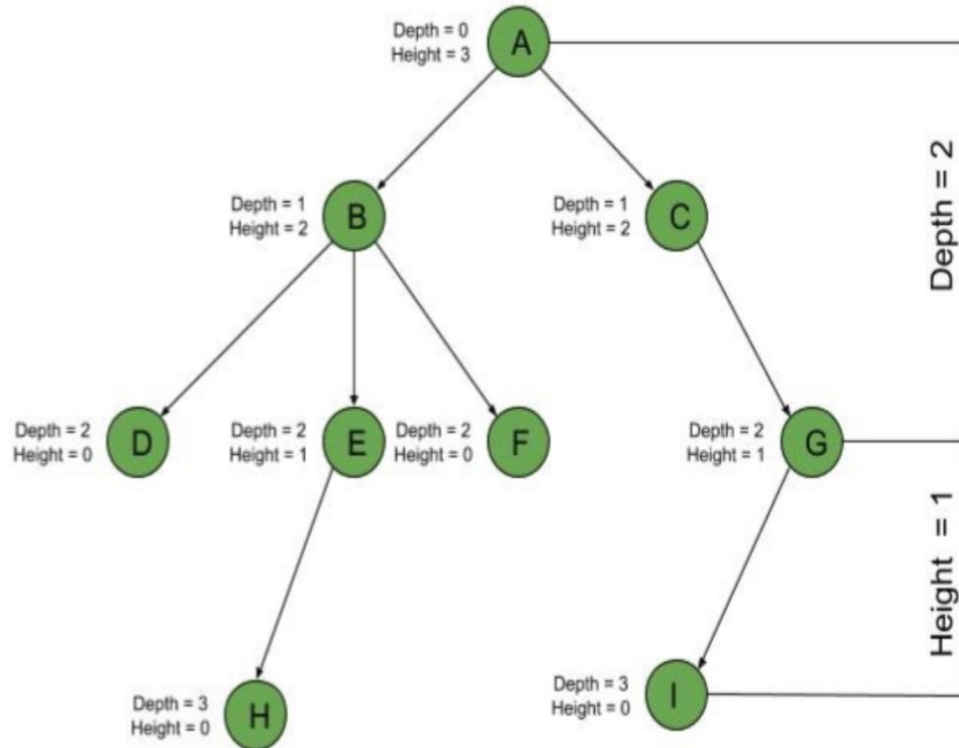
# Basic Terminologies

- Height
  - A node's **height** is the number of edges to its most distant leaf node.

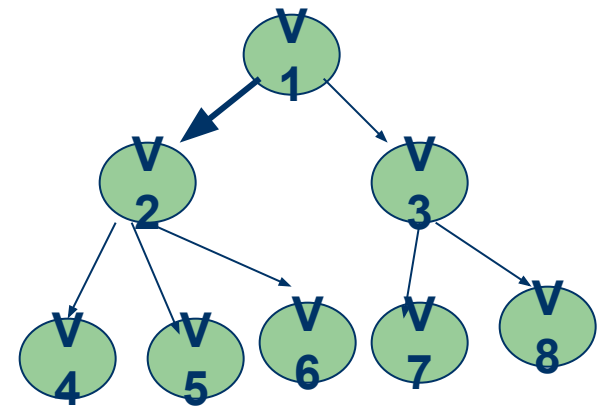
# Basic Terminologies

- Depth
- **A node's depth is the number of edges back up to the root.**
- So the root node always has a depth 0 while leaf node always has a height 0.

# Basic Terminologies

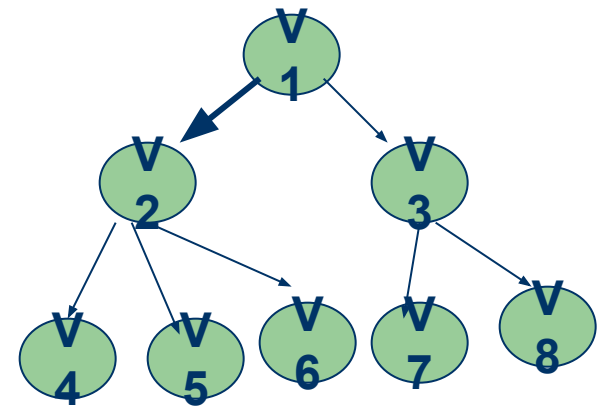


# Basic Terminologies



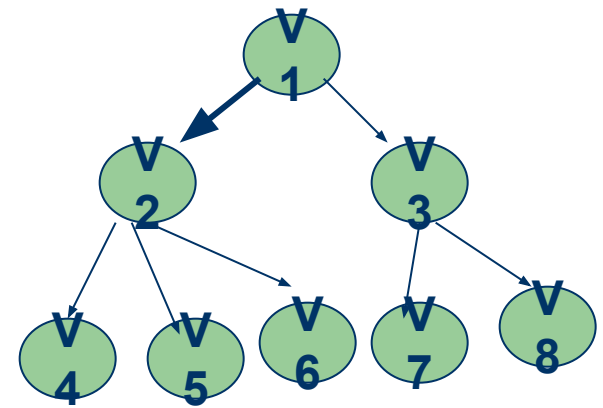
- Degree
  - The maximum number of children that is possible for any node is known as the degree of a node.
  - Ex: degree of v1 and v3 is 2 and v2 is 3.
  - degree of leaf node is 0.

# Basic Terminologies



- In- Degree: In-degree of a node is the number of edges arriving at that node.
- Out-degree Out-degree of a node is the number of edges leaving that node.
- Outdegree of V3 is 3 and In degree is 1.

# Basic Terminologies



- Siblings
  - The nodes which have the same parent is known as siblings.
  - Ex: v2 and v3 are siblings.
- Path
  - A sequence of consecutive edges is called a path.
  - Path from root node to V4 is v1-v2-v4.

# Basic Terminologies

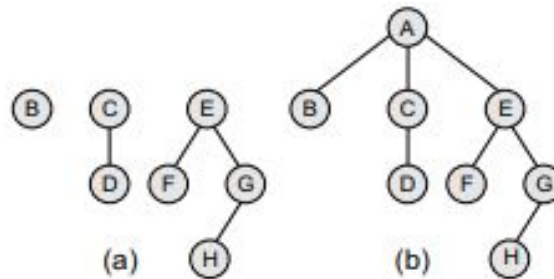
- Directed edge:
- Directed edge is an edge that represent some direction.



# General Tree

- General trees are data structures that store elements hierarchically. The top node of a tree is the root node and each node, except the root, has a parent. A node in a general tree (except the leaf nodes) may have zero or more sub-trees.
- However, the number of sub-trees for any node may be variable.

- Forest
- A forest is a disjoint union of trees. A set of disjoint trees (or forests) is obtained by deleting the root and the edges connecting the root node to nodes at level 1.

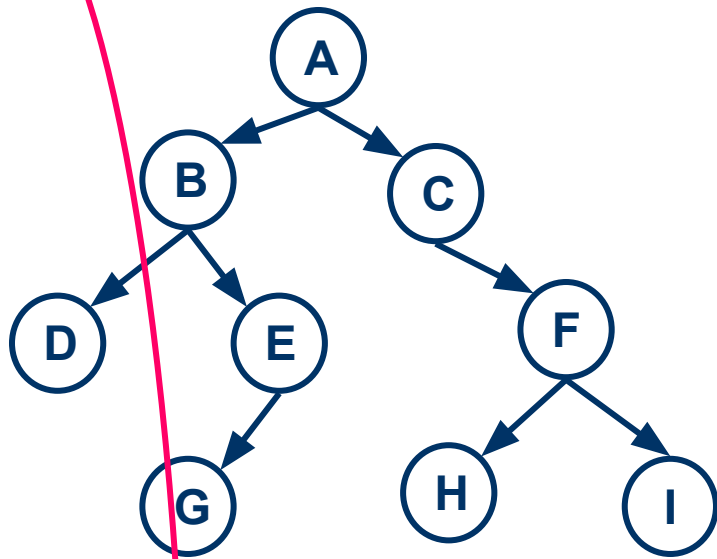


# M-ary tree:

- M-ary tree:  
if in a tree the **outdegree** of every node is less than or equal to  $m$ , then the tree is called m-ary tree.
- If  $m=2$  then its called binary tree.

# BINARY TREES

- In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.
- A node may have at most two children.



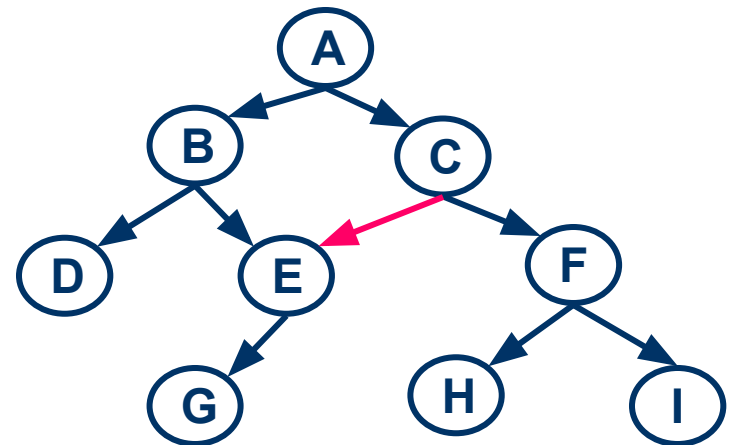
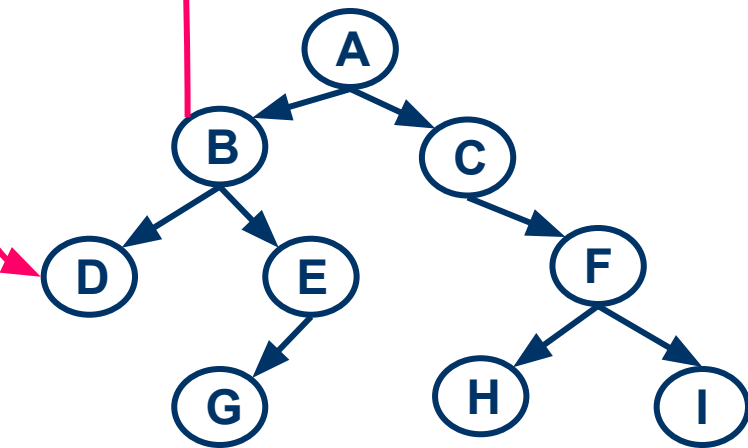
Nodes : 9

Root : A

Left Subtree : Rooted at B

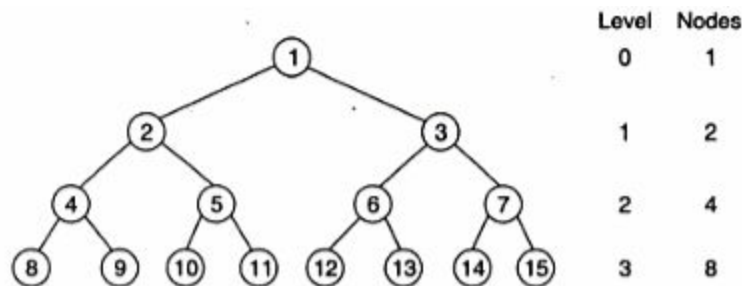
Right Subtree : Rooted at C

Structures that are not binary tree



# Full Binary Tree

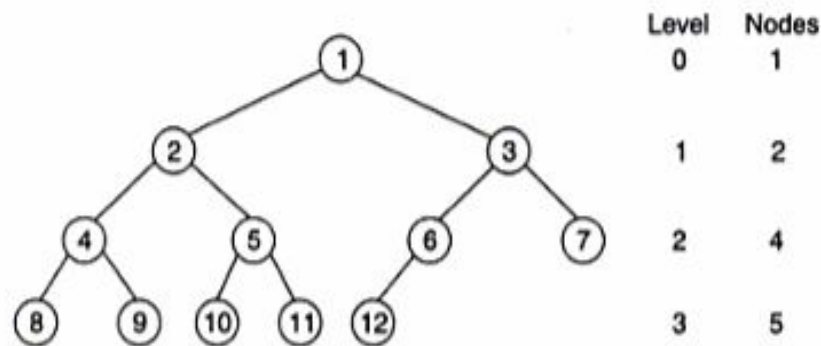
- Full binary tree: A binary tree is a full binary tree if it contains the maximum number of nodes at all levels.



(a) A full binary tree of height 4

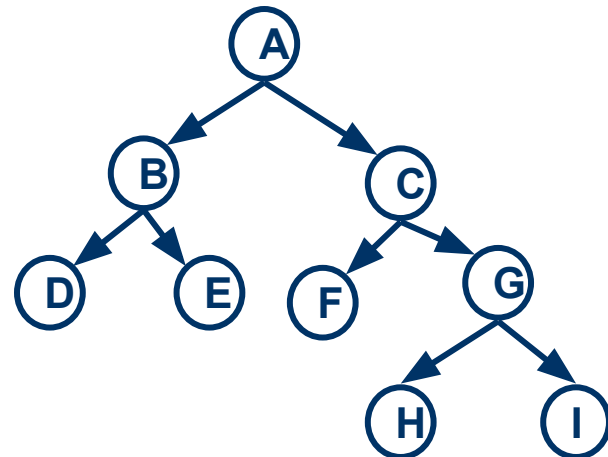
# Complete Binary Tree

- A binary tree is said to be a complete binary tree if all levels except the possibly the last level have the maximum number of possible nodes and all the nodes in the last level appear as far left as possible..



(b) A complete binary tree of height 14

- **Strictly binary tree**
- **A strict binary tree is a tree in which every node has zero or two children.**

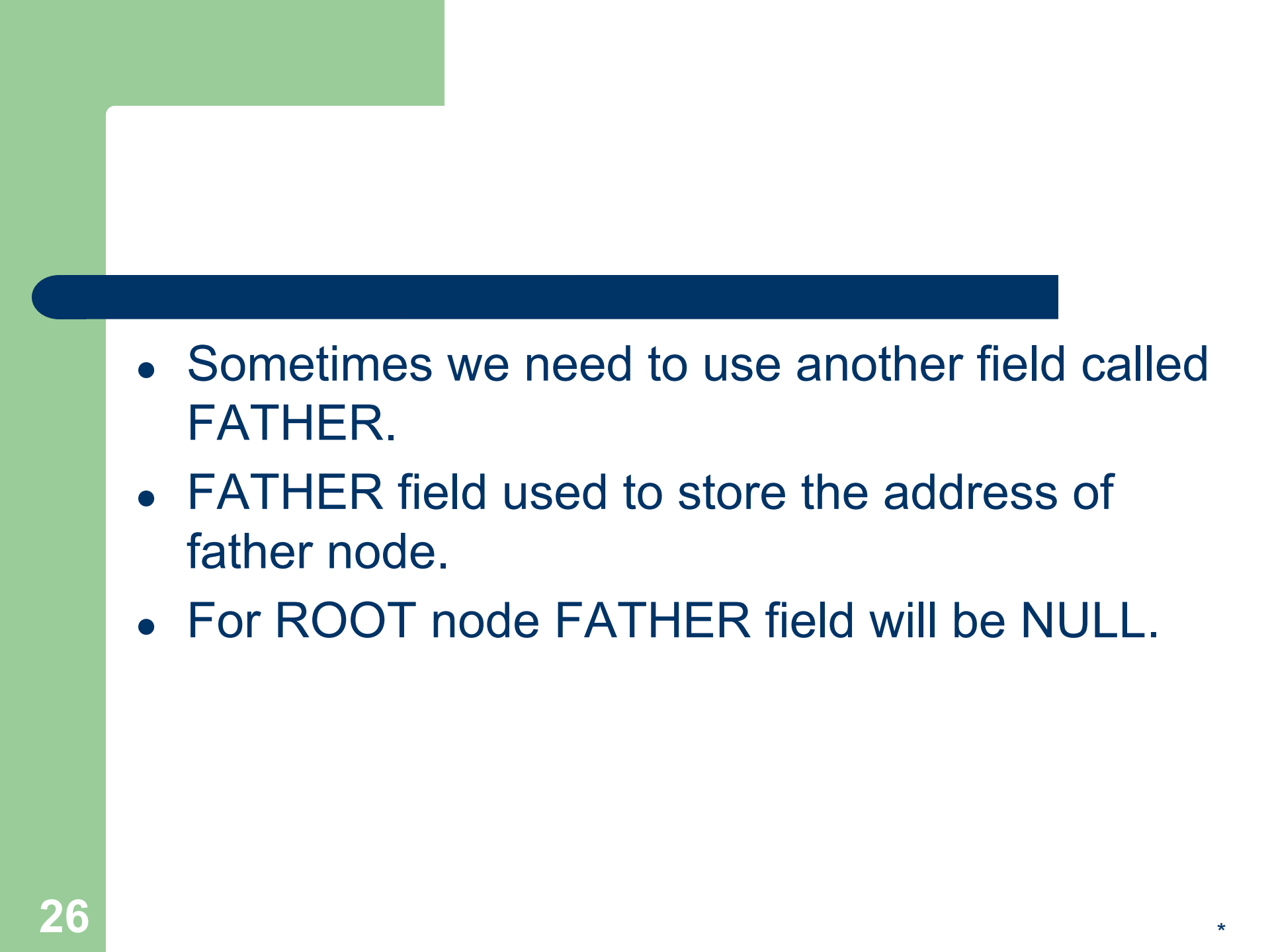




# NODE REPRESENTATION OF TREE

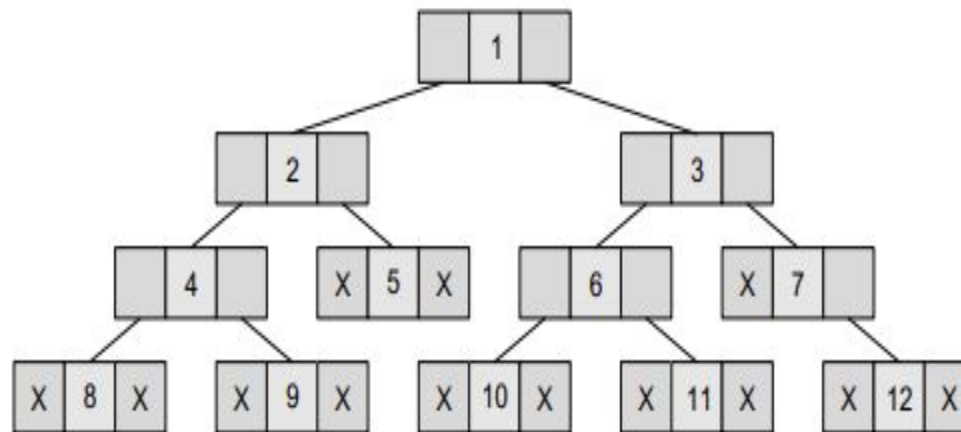
- Node will contain atleast three field
  1. LPTR
  2. DATA
  3. RPTR
- LPTR will store address of left subtree
- RPTR will store address of right subtree
- DATA will store our data.



- 
- Sometimes we need to use another field called FATHER.
  - FATHER field used to store the address of father node.
  - For ROOT node FATHER field will be NULL.

# C REPRESENTATION OF NODE

- struct node  
  {  
    int info;  
    struct nodetype \*left;  
    struct nodetype \*right;  
  };



**Figure 9.9** Linked representation of a binary tree

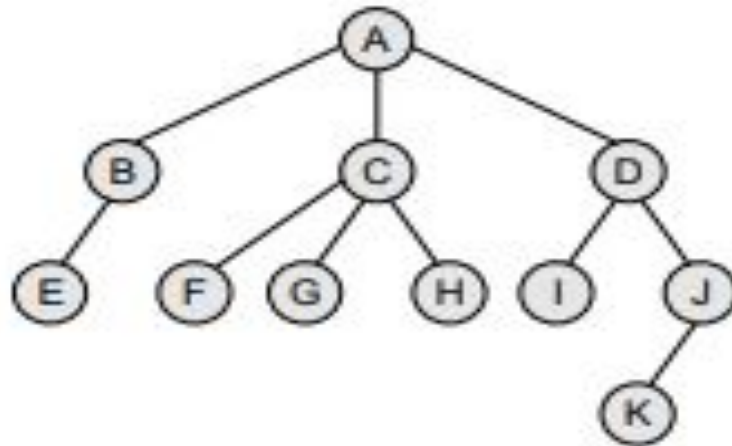
# Conversion of General Tree to Binary Tree

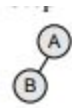
- The rules for converting a general tree to a binary tree are given below
- Rule 1: Root of the binary tree = Root of the general tree
- Rule 2: Left child of a node in the binary tree = Leftmost child of the node in the general tree
- Rule 3: Right child of a node in the binary tree = Right sibling of the node in the general tree

# Conversion of General Tree to Binary Tree

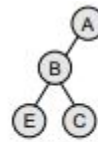
- Step 1: Node A is the root of the general tree, so it will also be the root of the binary tree.
- Step 2: Left child of node A is the leftmost child of node A in the general tree and right child of node A is the right sibling of the node A in the general tree. Since node A has no right sibling in the general tree, it has no right child in the binary tree.
- Step 3: Now process node B. Left child of B is E and its right child is C (right sibling in general tree).

# Conversion of General Tree to Binary Tree





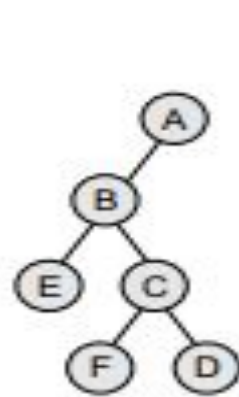
Step 2



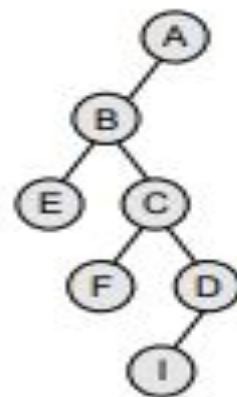
Step 3



- Step 4: Now process node C. Left child of C is F (leftmost child) and its right child is D (right sibling in general tree).
- Step 5: Now process node D. Left child of D is I (leftmost child). There will be no right child of D because it has no right sibling in the general tree.

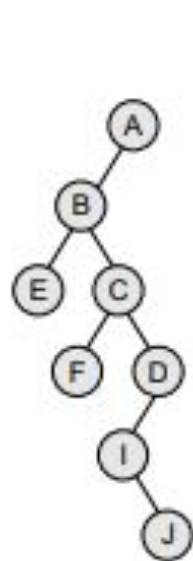


Step 4

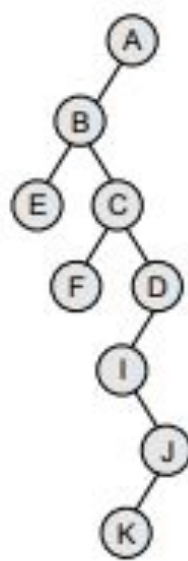


Step 5

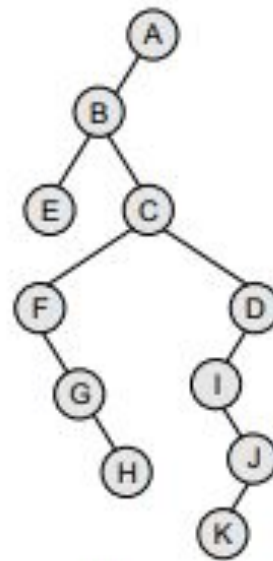
- Step 6: Now process node I. There will be no left child of I in the binary tree because I has no left child in the general tree. However, I has a right sibling J, so it will be added as the right child of I.
- Step 7: Now process node J. Left child of J is K (leftmost child). There will be no right



Step 6



Step 7



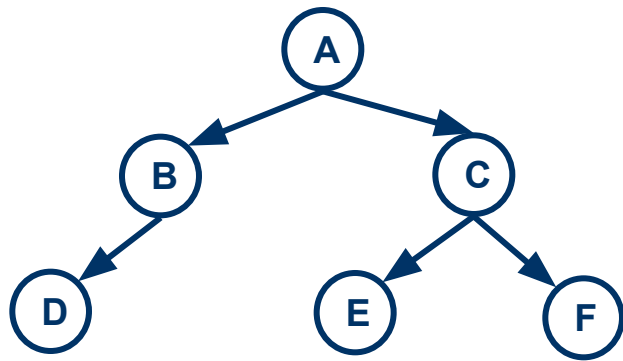
Step 8

# TREE TRAVERSAL

- Three methods for traversing a tree
  - Preorder (depth-first order) traversal
  - Inorder traversal
  - Postorder traversal

# Preorder traversal

- In preorder traversal of nonempty tree we need to perform following steps.
  1. Visit the root.
  2. Traverse the left subtree in preorder.
  3. Traverse the right subtree in preorder.



**Visit root - print A**

**Visit left subtree rooted at B in preorder.**

**Visit root B.**

**Visit left subtree rooted at D.**

**visit root D.**

**No left subtree**

**No right subtree.**

**Visit right subtree of B. (Not Available)**

**Visit right subtree rooted at C in preorder.**

**Visit root C.**

**Visit left subtree root at E.**

**visit root E.**

**No left subtree.**

**No right subtree.**

**Visit right subtree root at F.**

**Visit root F.**

**No left subtree**

**No right subtree.**

**Preorder sequence:**

**A B D C E F**

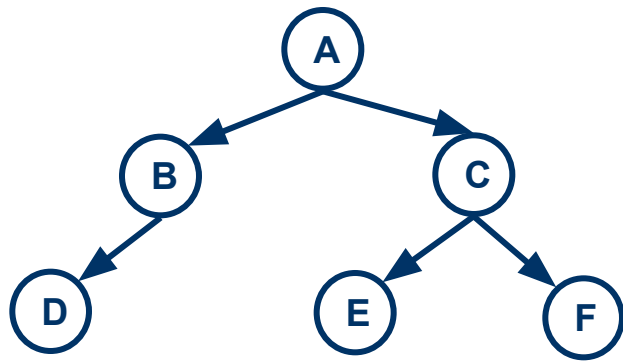
# ALGORITHM

- PREORDER(T)
  1. [Process the root node].  
If  $T \neq \text{NULL}$  then  
    Write ( $T \rightarrow \text{DATA}$ )  
Else  
    Write ("Empty Tree")  
    return
  2. [Process the left sub tree].  
If  $T \rightarrow \text{LPTR} \neq \text{NULL}$  then  
    call PREORDER( $T \rightarrow \text{LPTR}$ )
  3. [Process the right sub tree].  
If  $T \rightarrow \text{RPTR} \neq \text{NULL}$  then  
    call PREORDER( $T \rightarrow \text{RPTR}$ )
  4. return



# INORDER TRAVERSAL

- In preorder traversal of nonempty tree we need to perform following steps.
  1. Traverse the left subtree in inorder.
  2. Visit the root.
  3. Traverse the right subtree in inorder.



**Visit left subtree rooted at B in inorder.**

**Visit left subtree rooted at D.**

**No left subtree**

**visit root D.**

**No right subtree.**

**Visit node B.**

**Visit right subtree of B. (Not Available)**

**Visit root node A.**

**Visit right subtree rooted at C in inorder.**

**Visit left subtree root at E.**

**No left subtree.**

**visit root E.**

**No right subtree.**

**Visit root C.**

**Visit right subtree root at F.**

**No left subtree**

**Visit root F.**

**No right subtree.**

**Inorder sequence:**

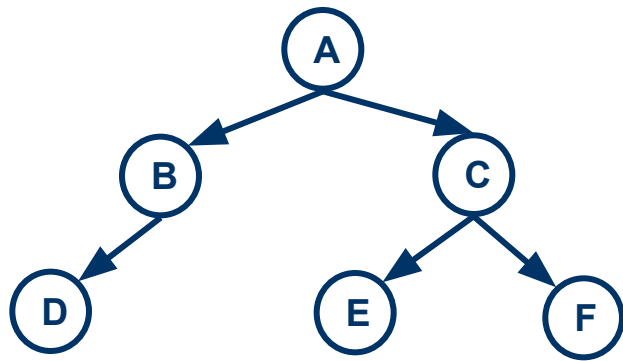
**D B A E C F**

# ALGORITHM

- INORDER ( T )
  1. [Check for empty tree]  
If T = NULL then  
    Write ("Empty tree ")  
    return
  2. [Process the left subtree]  
if T->LPTR != NULL then  
    call INORDER(T->LPTR)
  3. Write T->Data
  4. [Process the right subtree]  
If T->RPTR !=NULL then  
    call INORDER(T->RPTR)
  5. return

# POSTORDER TRAVERSAL

- In preorder traversal of nonempty tree we need to perform following steps.
  1. Traverse the left subtree in postorder.
  2. Traverse the right subtree in postorder.
  3. Visit the root.



**Visit left subtree rooted at B in postorder.**

**Visit left subtree rooted at D.**

**No left subtree**

**No right subtree.**

**visit root D.**

**Visit right subtree of B. (Not Available)**

**Visit node B.**

**Visit right subtree rooted at C in postorder.**

**Visit left subtree root at E.**

**No left subtree.**

**No right subtree.**

**visit root E.**

**Visit right subtree root at F.**

**No left subtree**

**No right subtree.**

**Visit root F.**

**Visit root C.**

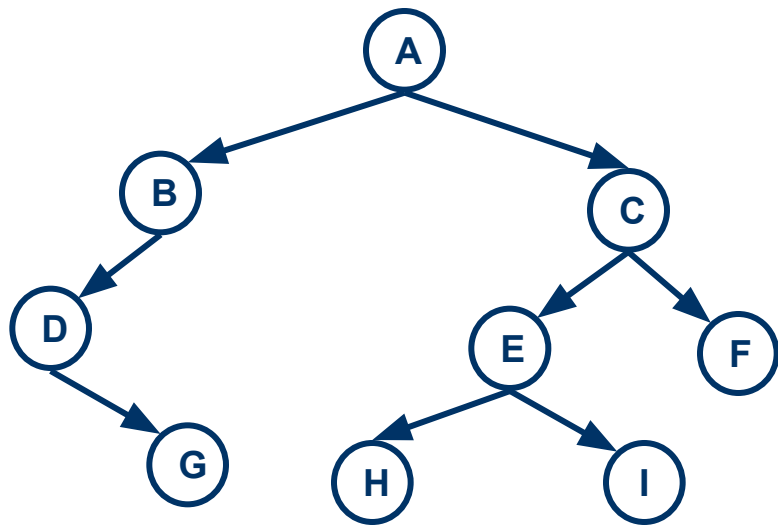
**Visit root node A.**

**Postorder sequence:**

**D B E F C A**

# ALGORITHM

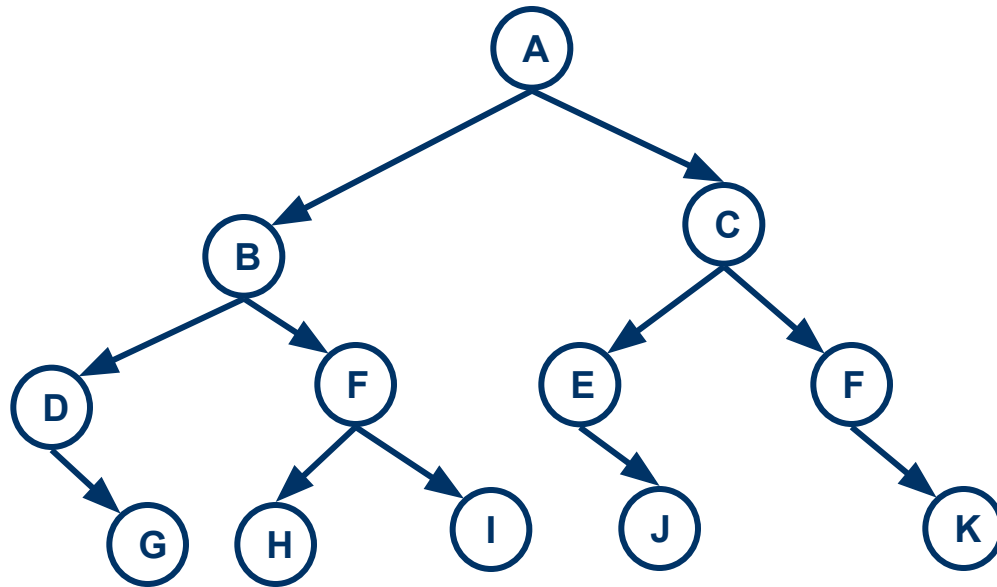
- POSTORDER ( T )
  1. [Check for empty tree]  
If T = NULL then  
    Write (“Empty tree “)  
    return
  2. [Process the left subtree]  
if T->LPTR != NULL then  
    call POSTORDER(TLP->TR)
  3. [Process the right subtree]  
If TRPTR !=NULL then  
    call POSTORDER(T->RPTR)
  4. Write T->Data
  5. return



Preorder : A B D G C E H I F

Inorder : D G B A H E I C F

Postorder : G D B H I E F C A



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

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

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

# Applications of Tree

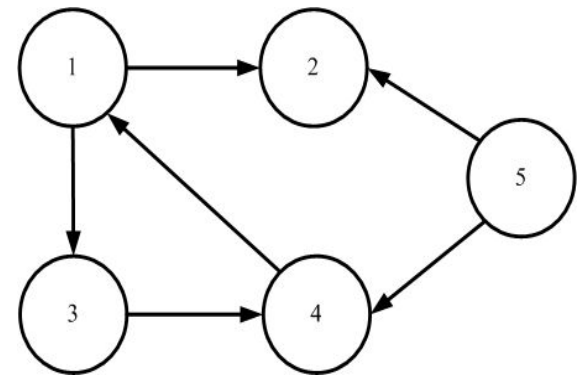
1. Trees are often used for implementing other types of data structures like hash tables, sets, and maps
2. A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating system use.
3. Another variation of tree, B-trees are prominently used to store tree structures on disc.
4. B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.



# Applications of Tree

- 5. Trees are an important data structure used for compiler construction
- 6. Trees are also used in database design.
- 7. Trees are used in file system directories.
- 8. Trees are also widely used for information storage and retrieval in symbol tables.

# Graphs: Review

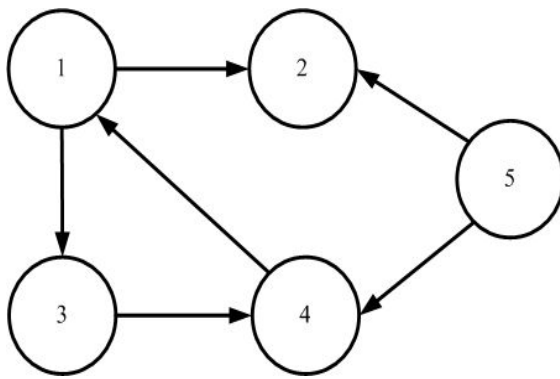


- A graph
  - A graph  $G$  consists of a nonempty set  $V$  called the set of nodes (points or vertices) of the graph and set  $E$  which is set of edges of the graph and a mapping from the set of edges  $E$  to a set of pairs of elements of  $V$ .
  - Denoted by  $G = (V, E)$ 
    - $V$  = set of vertices,  $E$  = set of edges

# Representing Graphs

- Adjacency-Matrix

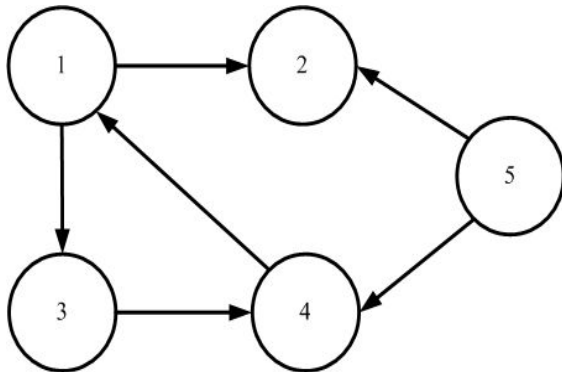
- Consider Graph  $G(V,E)$  and Assume  $V = \{1, 2, \dots, n\}$
- An adjacency matrix represents the graph as a  $n \times n$  matrix  $A$ :
- $A[i, j] = 1$  if edge  $(i, j) \in E$  (or weight of edge)  
     $= 0$  if edge  $(i, j) \notin E$



	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	0
3	0	0	0	1	0
4	1	0	0	0	0
5	0	1	0	1	0

# Representing Graphs

- Adjacency-List
  - There is a one list for each node (vertex) of graph G and each list contains adjacent nodes.



1	<input type="checkbox"/>	2	3	\
2	<input type="checkbox"/>	\		
3	<input type="checkbox"/>	4	\	
4	<input type="checkbox"/>	1	\	
5	<input type="checkbox"/>	2	4	\