

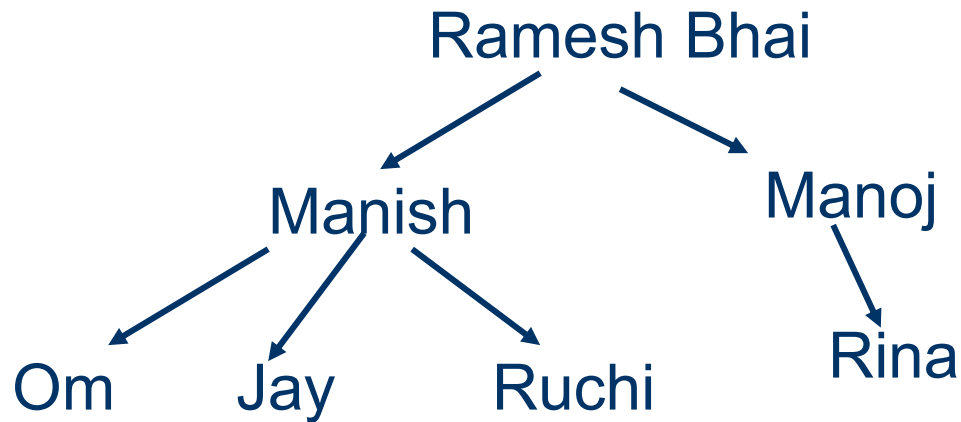
# DATA STRUCTURE

TREES

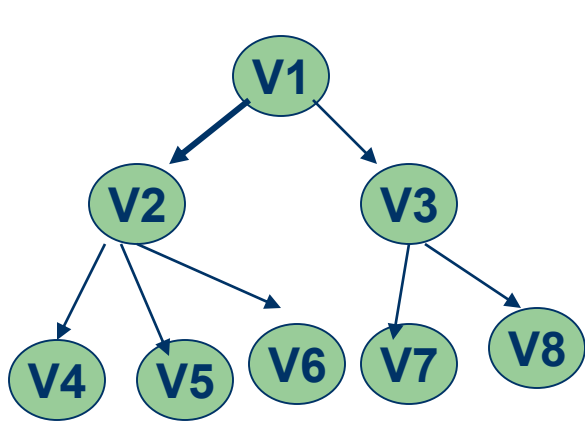


# TREE

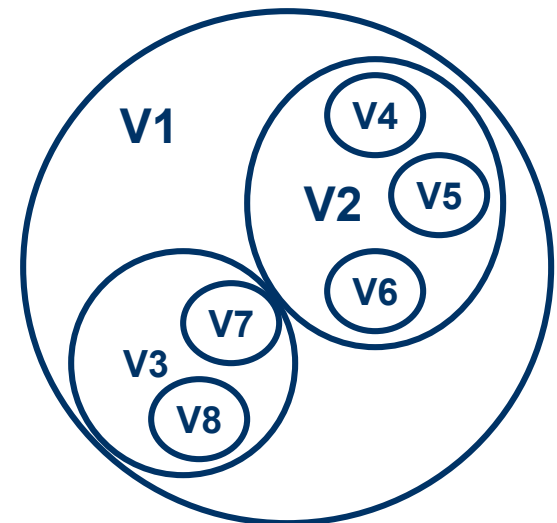
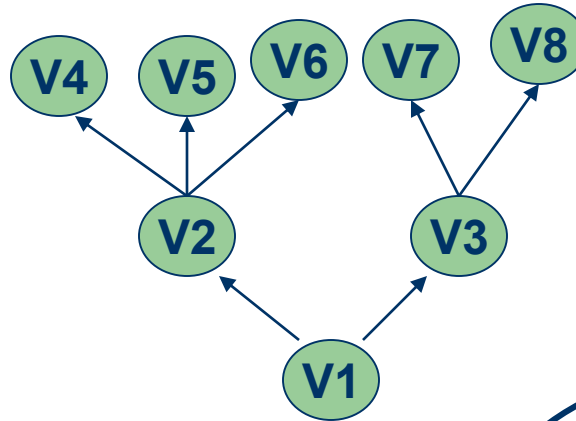
- Mainly used to represent data containing a hierarchical relationship between elements.
- Ex. – family tree



# Different Representation of Tree



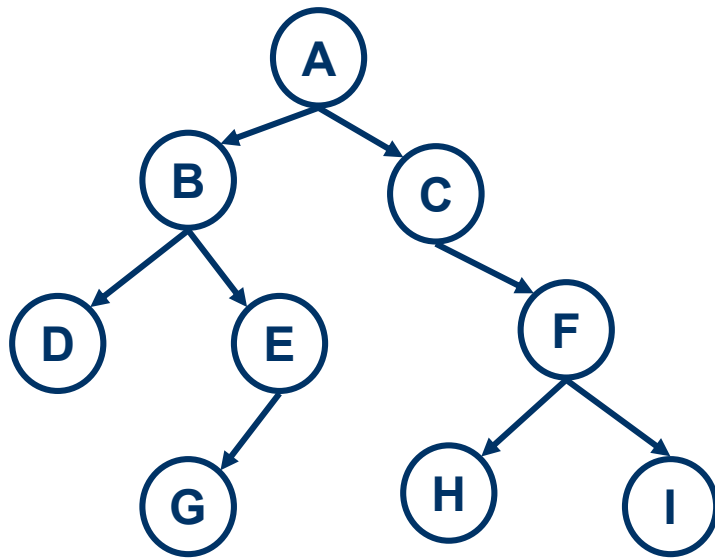
$(V1(V2(V4)(V5)(V6))(V3(V7)(V8)))$



- 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 the outdegree of every node is exactly equal to  $m$  or  $0$  and the number of nodes at **level  $i$**  is  $m^{i-1}$  then tree is **full or complete  $m$ -ary** tree.
- If  $m=2$  then its called binary tree.

# BINARY TREES

- A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets.
- First elements called the **root** of the tree.
- Other two subsets are themselves binary tree, called **left** and **right subtrees** of the original tree.
- Each element of binary tree is called a **node** of the tree.



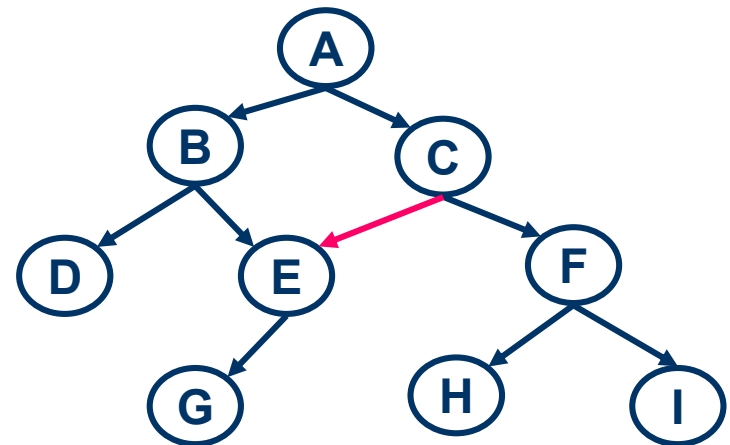
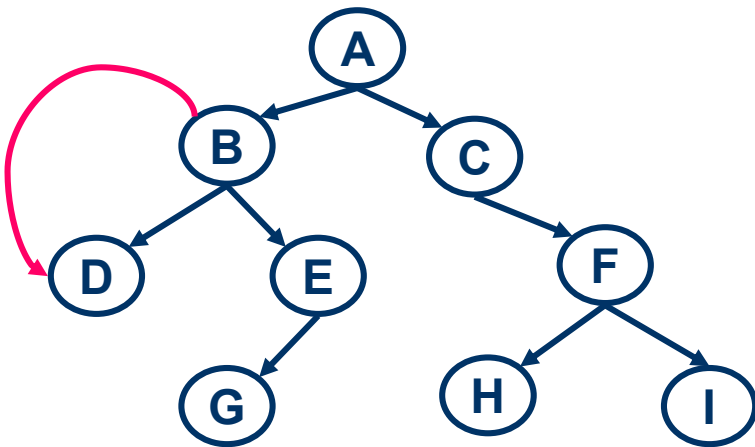
Nodes : 9

Root : A

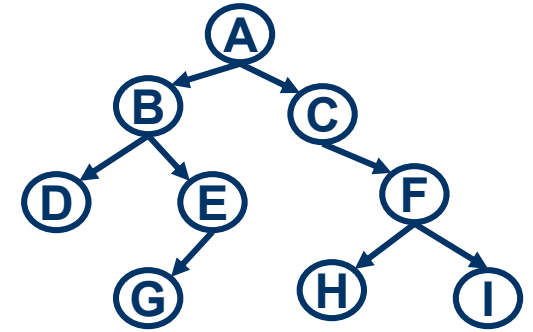
Left Subtree : Rooted at B

Right Subtree : Rooted at C

Structures that are not binary tree



# Definitions

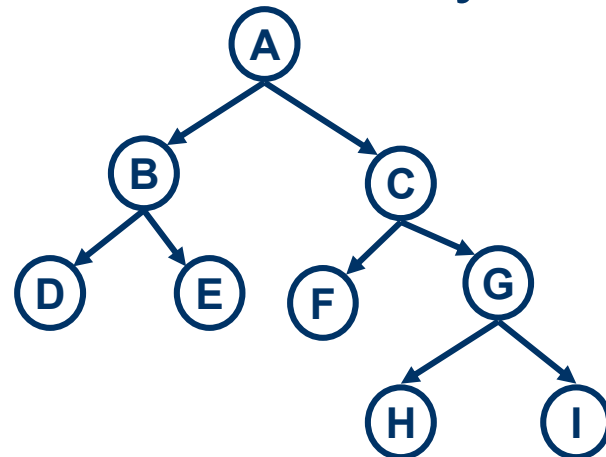


- If A is the root of a binary tree and B is the root of its left or right subtree, then A is said to be the **father** of B and B is said to be the **left** or **right son** of A.
- A node has no sons is called **leaf**. (D, G, H, I are leaf)
- Node **n1** is an **ancestor** of node **n2** if **n1** is either the father of **n2** or father of some ancestor of **n2**.
- (A is ancestor of G,E,D,B and B is ancestor of D).
- **n2** is a **descendant** of **n1**.
- (D is descendant of A and B , but E is neither ancestor or descendant of C.)
- A node **n2** is a **left/right descendant** of node **n1** if **n2** is either left/right son of **n1** or a descendant of the left/right son of **n1**.
- (E is left descendant of A. E is not left descendant of B.)
- Two nodes are **brothers** if they are left and right sons of same father.

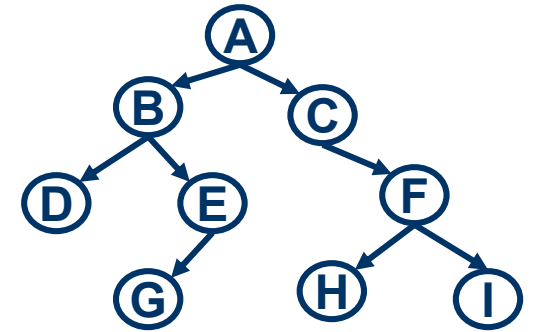
- **Strictly binary tree**

If every nonleaf node is a binary tree has nonempty left and right subtree then it is termed strictly binary tree.

- Strictly binary tree with  $n$  leaves always contains  $2n-1$  nodes.



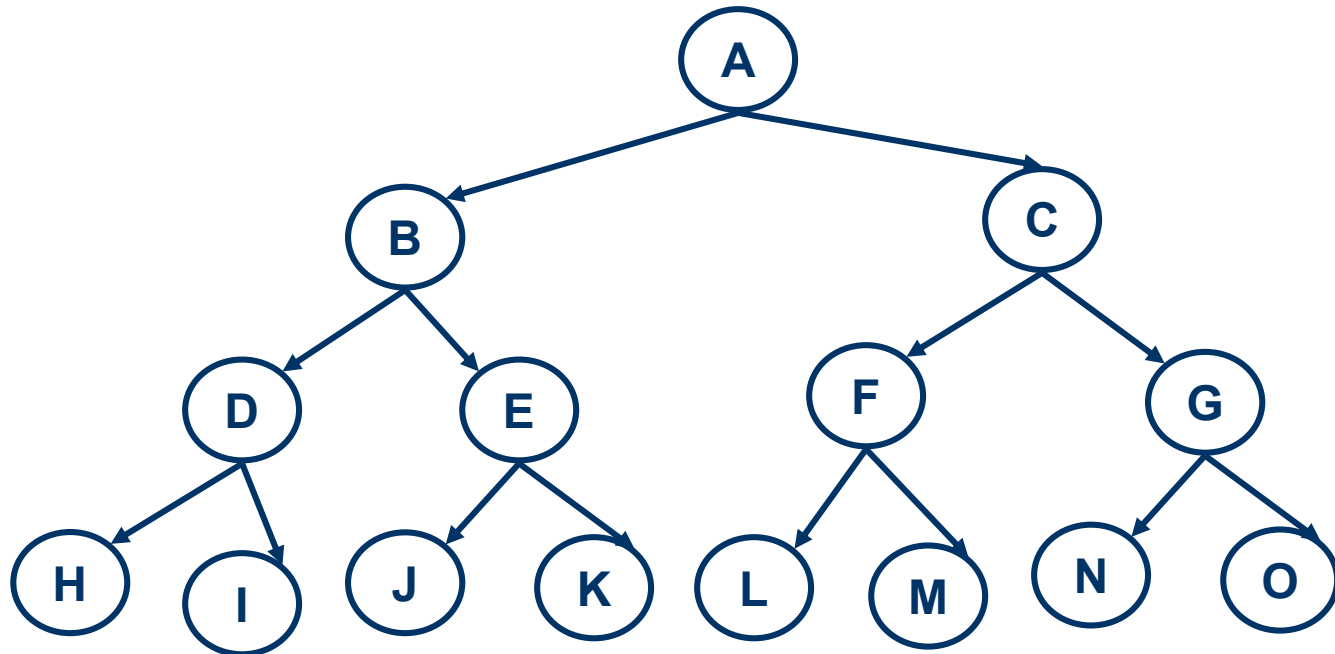




- Level :  
the root of the tree has level 0, and the level of any other node in the tree is one more than the level of its father.
- Above figure B is at level 1, G at level 3.
- The depth of a binary tree is the maximum level of any leaf in the tree.
- This is equals the length of the longest path from the root to any leaf.
- Depth of tree is 3.

# Complete Binary Tree

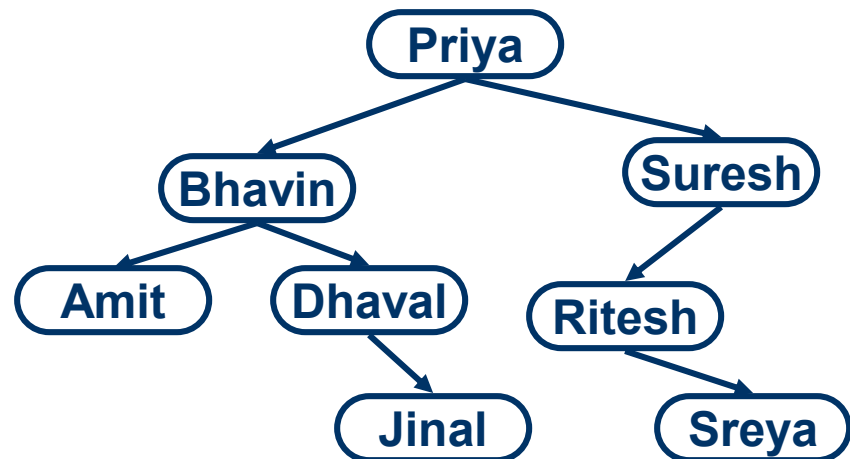
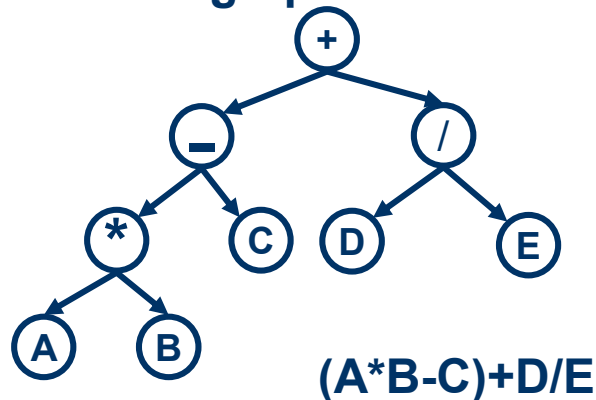
- A Complete binary tree of depth  $d$  is the strictly binary tree all of whose leaves are at level  $d$ .



- If a binary tree contains  $M$  nodes at level  $L$ , it contains at  $2M$  node at level  $L+1$ .
- At level  $L$  Number of nodes =  $2^L$
- Total number of nodes of complete binary tree of depth  $d$ . =  $2^{d+1} - 1$

# BINARY TREE REPRESENTATION

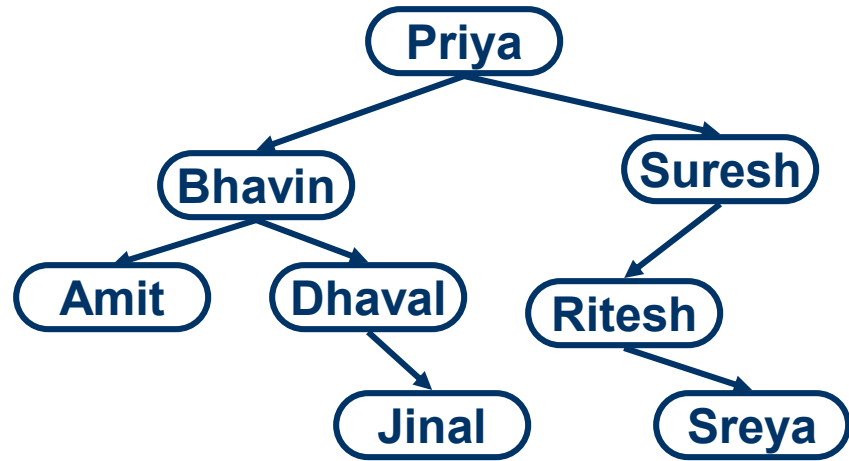
- The approach taken in constructing a tree is often application-dependent.
- Ex. – binary tree representation of mathematical expression depends on the right and left operands associated with each binary operator.
- Creation of binary could be based on the information associated with each node. The ordering list of names to be kept in lexicographical order.



- Insertion of a node into a lexically order tree must maintain that ordering.
- Such operation performed at the leaf level.
- Insertion into an empty tree results in appending the new node as the root of the tree.
- Inserting a new node into a nonempty tree. New node first compare to the root name and lexically precedes the root name, then new node is appended to the tree as a left leaf to the existing tree if the left subtree is empty.
- Otherwise process repeated with root node of left subtree. Same for other case.

Names input in sequence:

1. Priya
2. Bhavin
3. Dhaval
4. Suresh
5. Amit
6. Ritesh
7. Jinal
8. Sreya



# NODE REPRESENTATION OF TREE

- Node will contain atleast three field

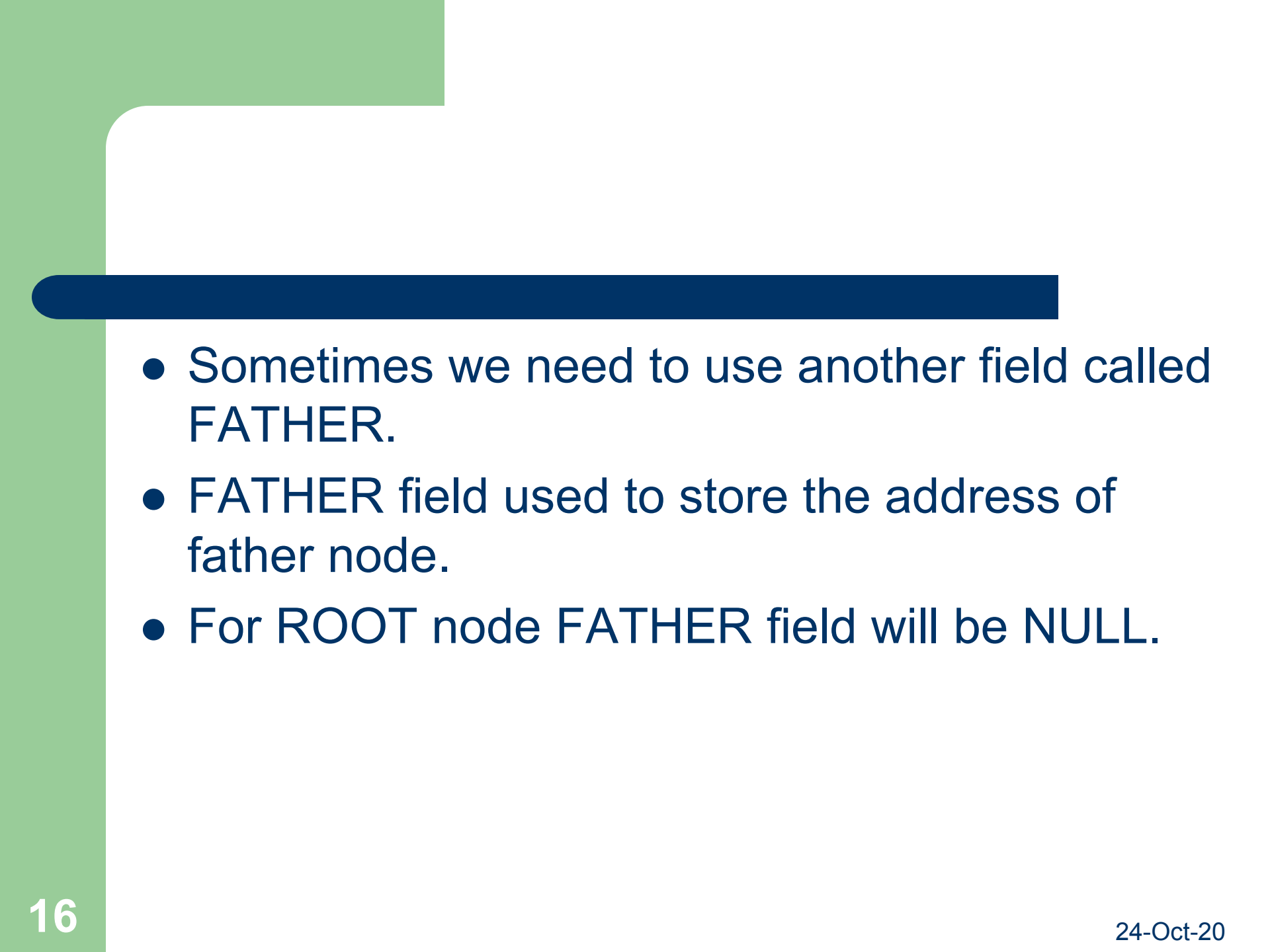
1. LPTR

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

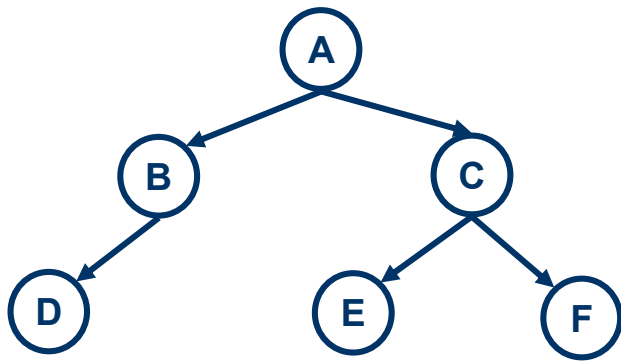
- struct node  
{  
    int info;  
    struct nodetype \*left;  
    struct nodetype \*right;  
};

# 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:**

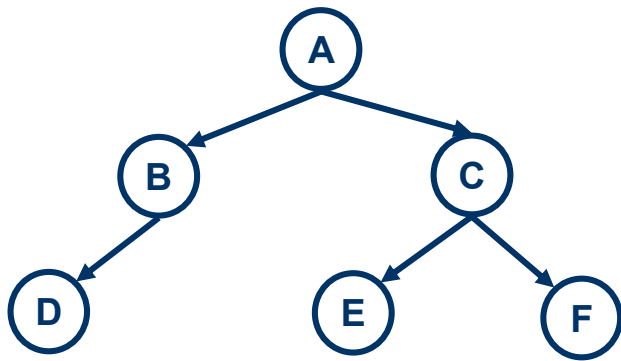
**A B D C E F**

# ALGORITHM

- PREORDER(T)
  1. [Process the root node].  
If T != NULL then  
    Write (T->DATA)  
Else  
    Write ("Empty Tree")  
    return
  2. [Process the left sub tree].  
If T->LPTR != NULL then  
    call PREORDER(T->LPTR)
  3. [Process the right sub tree].  
If T->RPTR !=NULL then  
    call PREORDER(T->RPTR)
  4. return

# INORDER TRAVERSAL

- In Inorder 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**

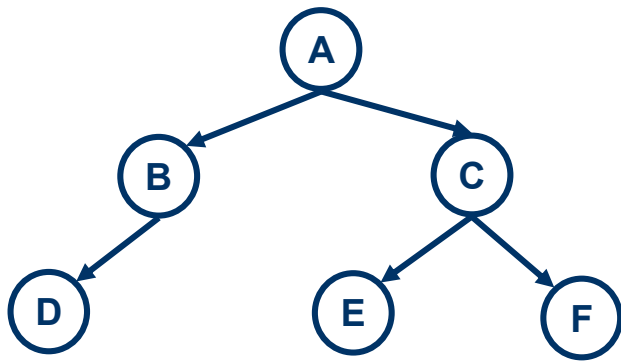
# ALGORITHM

- 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.**

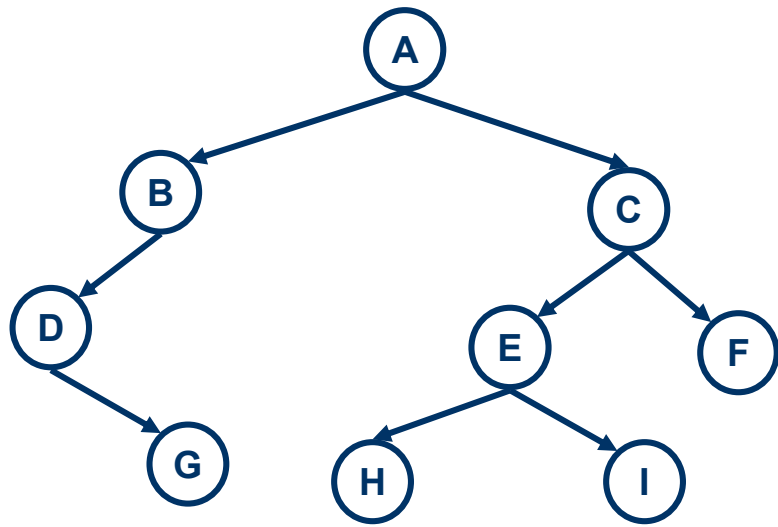
**Visit root node A.**

**Postorder sequence:**

**D B E F C A**

# ALGORITHM

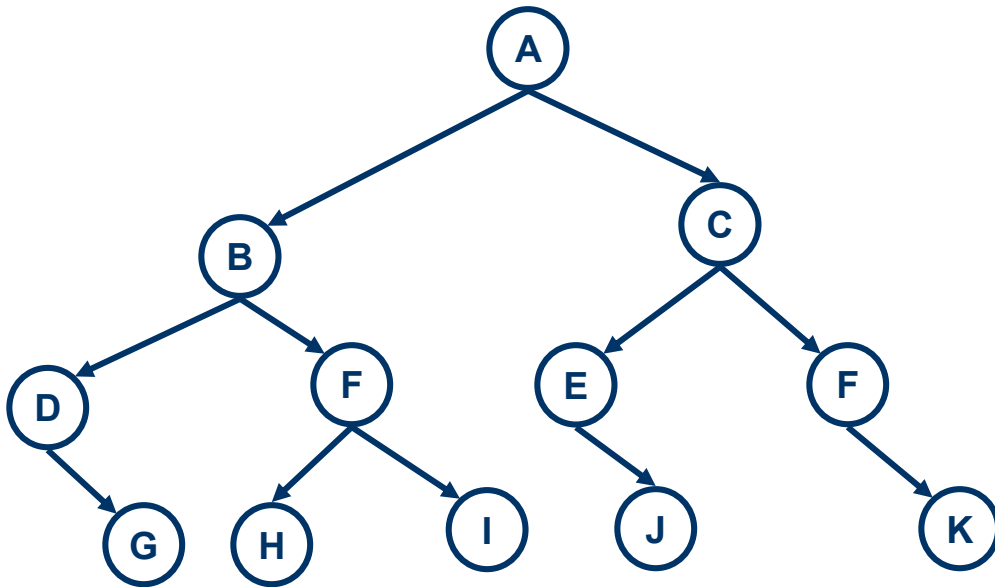
- POSTORDER ( T )
  1. [Check for empty tree]  
If T = NULL then  
    Write ("Empty tree ")  
    return
  2. [Process the left subtree]  
if T->LPTR != NULL then  
    call POSTORDER(TLP->TR)
  3. [Process the right subtree]  
If TRPTR !=NULL then  
    call POSTORDER(T->RPTR)
  4. Write T->Data
 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

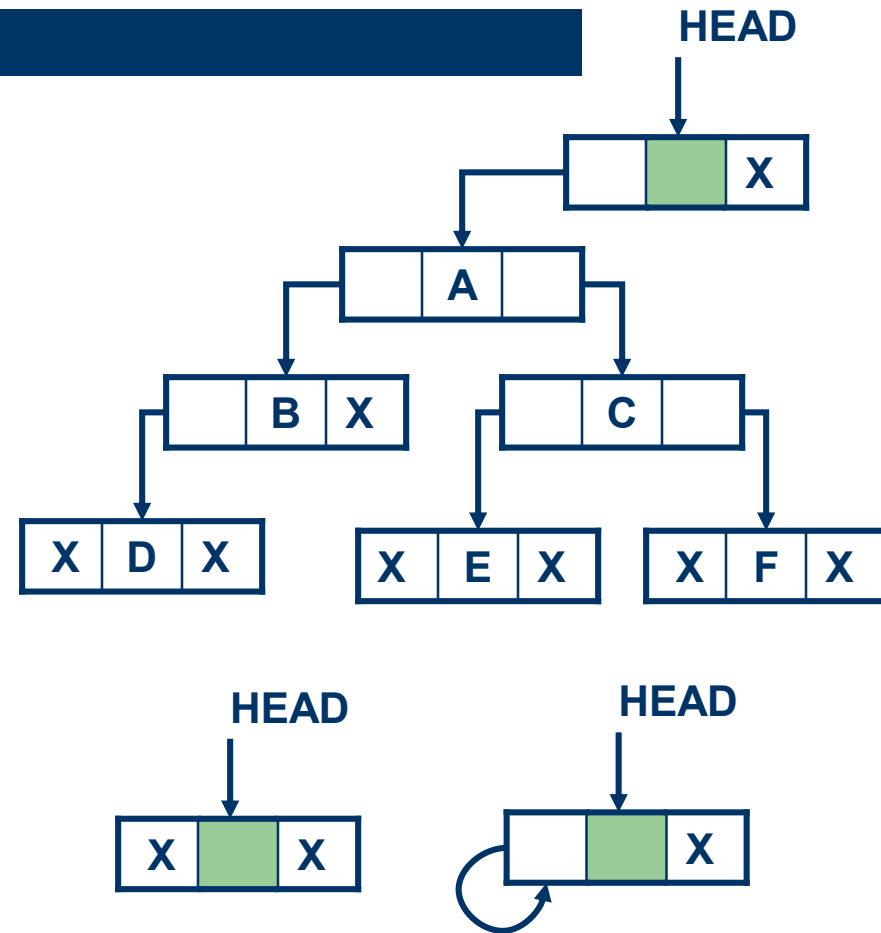
# COPY TREE

- COPY( T )
  1. [Null pointer]  
if T=NULL then  
    return NULL
  2. [create new node.]  
new=CreateNode()
  3. [copy information field]  
new->Data=T->Data
  4. [Set structural links]  
new->LPTR=COPY(T->LPTR)  
new->RPTR=COPY(T->RPTR)
  5. [return the address of new node]  
Return new

# HEADER NODES

- Suppose T is a tree. Sometimes extra, special node, called Header node, is added to beginning of T.
- When this extra node is used, the tree pointer variable, which will call HEAD instead of ROOT, will point to header node.
- Header node left pointer will point to the root of T.

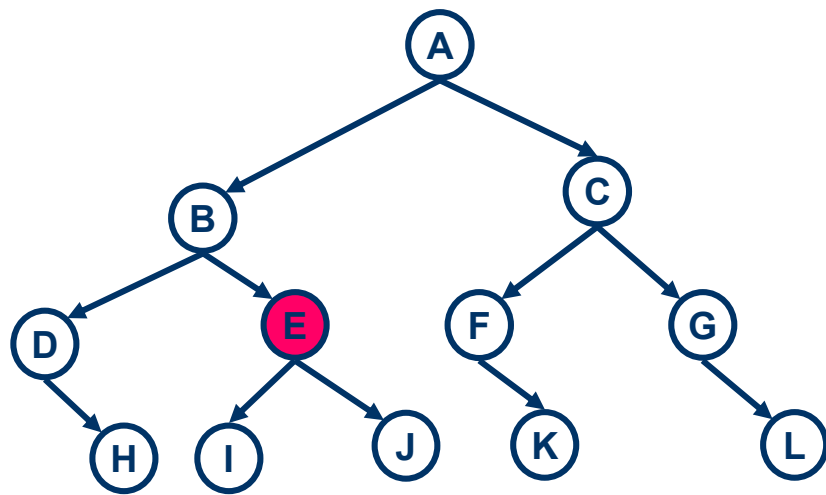
- Suppose a binary tree T is empty. Then T will still contain a header node, but the left pointer point to NULL.
- Sometimes left pointer contain the address pf header node instead of NULL value.
- $HEAD \rightarrow LEFT = HEAD$  condition is used to checked tree is empty or not.



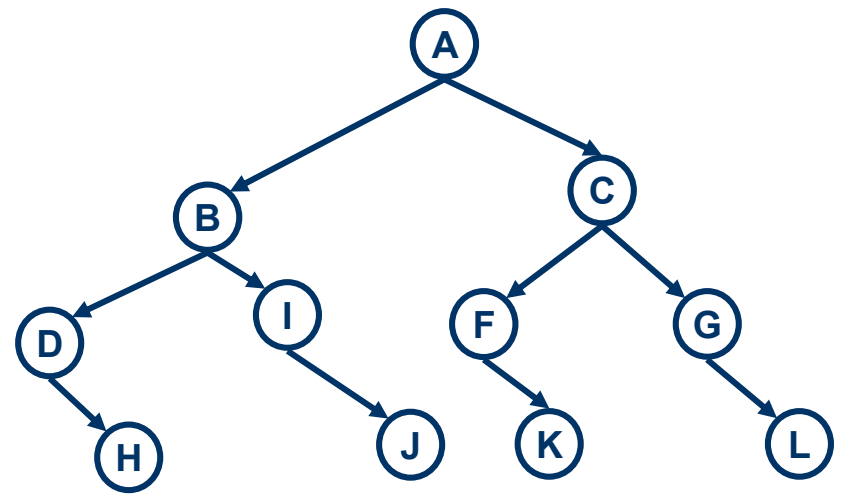
# DELETING NODES

- Deletion of tree depends on the ordering of the tree.
- If nodes are not lexically ordered then traversal is used to find out the node and it is removed.
- If nodes are ordered then we can search node with comparing values.
- For deletion two cases arise.
  - If left or right subtree is empty then connect other subtree to the parent of node.
  - Find out the inorder successor of the node replace the node with its inorder successor node.



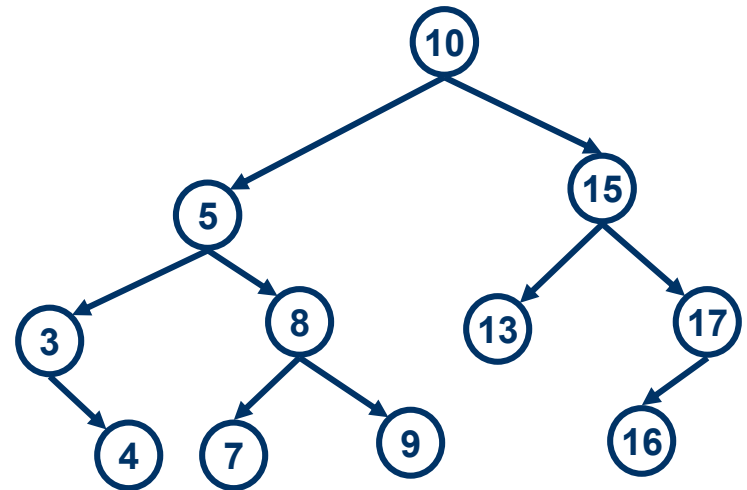
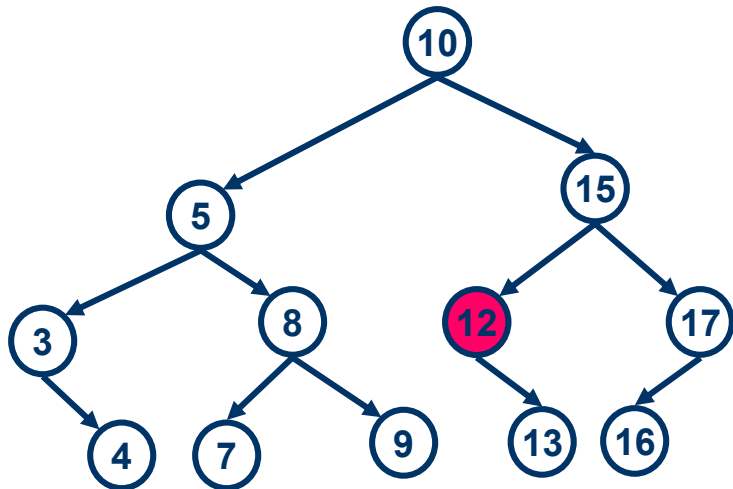


E is marked for delete

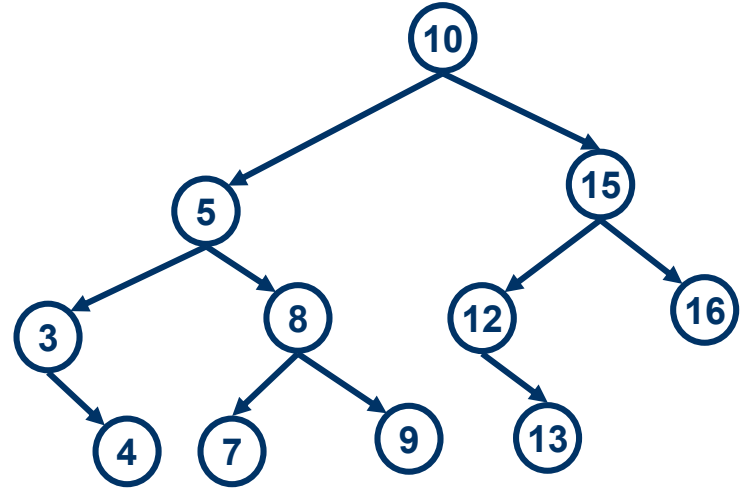
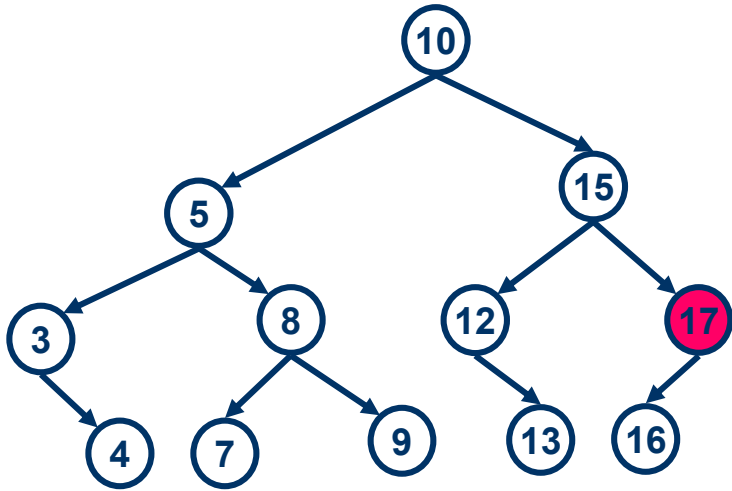


After deleting node.

If tree is lexically ordered tree.  
If left subtree is empty

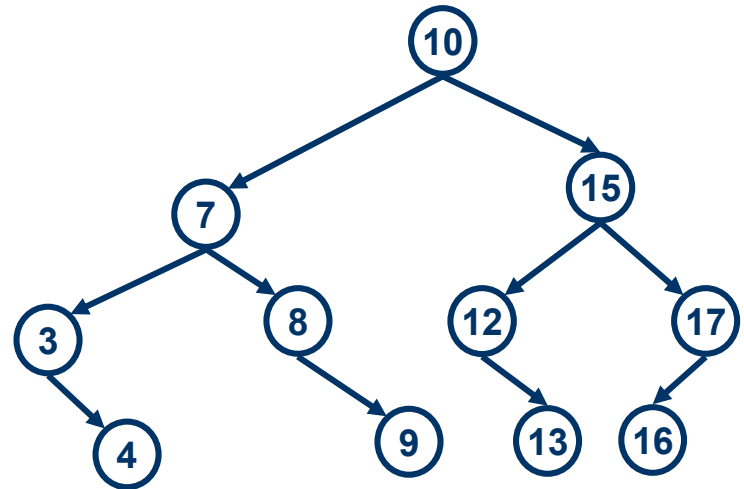
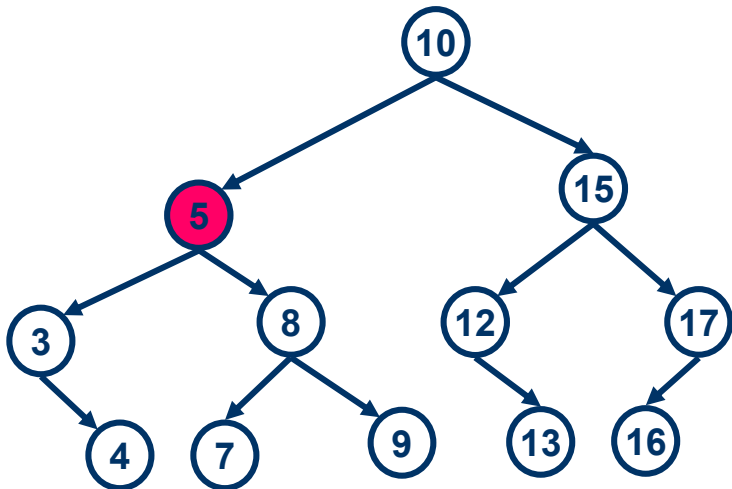


If right subtree is empty.



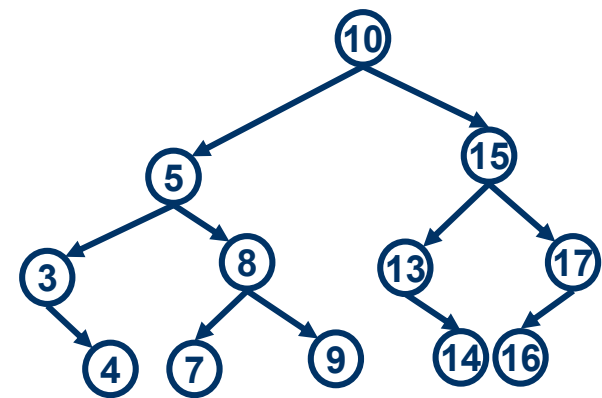
If right and left subtrees are not empty.

Inorder sequence: 3 4 5 7 8 9 10 12 13 15 16 17



# ALGORITHM

- DELETE(HEAD,X)
  1. [Initialize]  
If HEAD->LPTR != HEAD  
CUR=HEAD->LPTR  
PARENT=HEAD  
D='L'  
ELSE  
write("Node not found")  
return
  2. [search for node marked for deletion]  
FOUND=false  
Repeat while not FOUND and CUR!=NULL  
If CUR->DATA = X THEN  
FOUND=true  
ELSE IF X<CUR->DATA THEN  
PARENT=CUR  
CUR=CUR->LPTR  
D='L'  
ELSE  
PARENT=CUR  
CUR=CUR->LPTR  
D='R'  
END WHILE  
IF FOUND=FALSE THEN  
write("Node not found")  
return



3. [perform deletion]
  - if CUR->LPTR=NULL then
    - Q = CUR->RPTR
  - else if CUR->RPTR=NULL then
    - Q = CUR->LPTR
  - else
    - SUC=CUR->RPTR
    - if SUC.LPTR=NULL then
      - SUC->LPTR=CUR->LPTR
      - Q=SUC
    - else
      - PRED= CUR->RPTR
      - SUC = PRED.LPTR
      - repeat while SUC->LPTR!=NULL
        - PRED=SUC
        - SUC=PRED->LPTR

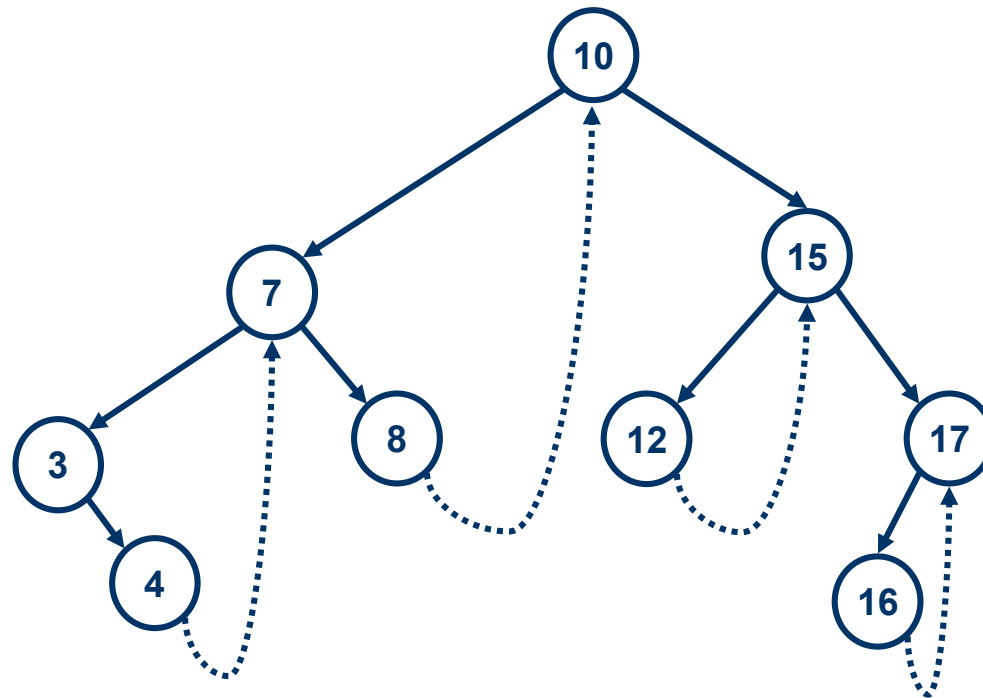
PRED->LPTR=SUC->RPTR  
 SUC->LPTR=CUR->LPTR  
 SUC->RPTR=CUR->RPTR  
 Q=SUC

4. If D='L' then
  - PARENT->LPTR = Q
- else
  - PARENT->RPTR=Q
5. return

# THREADS

- If we consider any binary tree then half of the entries in the pointer field LPTR and RPTR will contain null elements.
- This space may be more efficiently used by replacing the null entries with other information.
- We can replace certain null entries by special pointers which points to nodes higher in tree.
- This special pointers are called ***threads***.
- Binary tree with such pointer is called ***threaded trees***.

# EXAMPLE



- Threads in threaded tree must be distinguished in some way from ordinary pointers.
- The threads in threaded tree are usually indicated by dotted lines.
- In computer memory, an extra 1-bit TAG field may be used to distinguished threads from ordinary pointers.
- Alternatively, thread may be used as negative pointer variable and normal pointer can be used as positive.
- (In C language negative value of pointer not possible so we can take another variable to store type of pointer.)

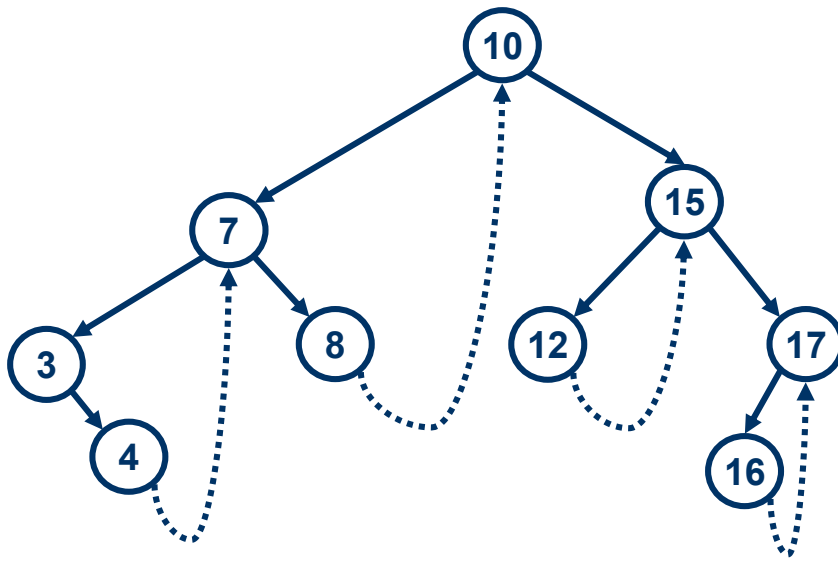
# TYPES OF THREAD

- Many ways to thread a binary tree T.
- Each threading will correspond to a particular traversal of T.
- Also we can following category
  - One way threading
  - Two way threading

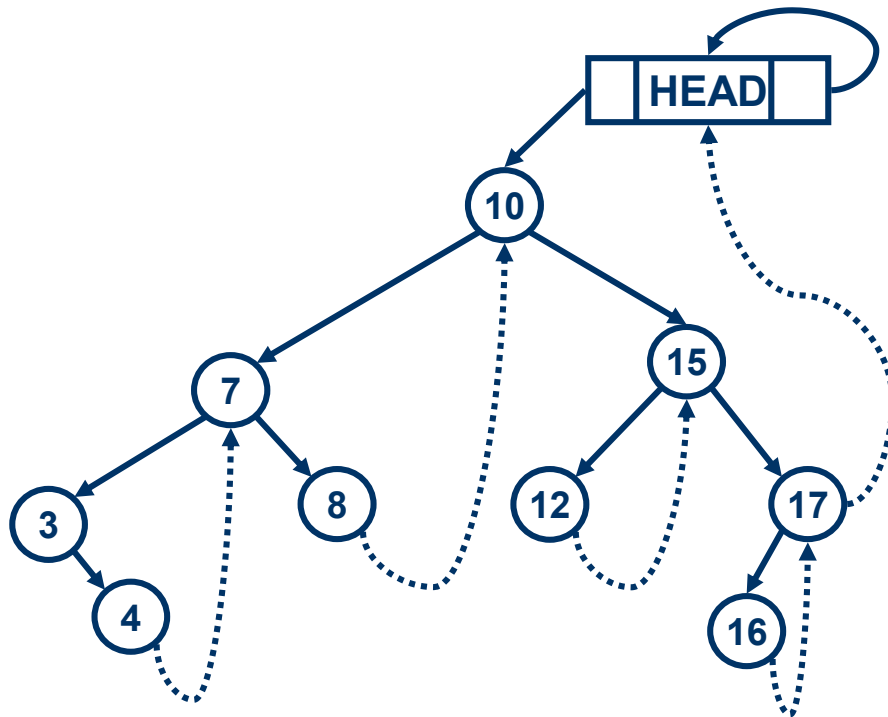


# One Way Threading

- In one way threading, Thread will appear only on right pointer variable RPTR.
- Assuming our threading corresponds to the inorder traversal.
- Node, which doesn't has right child, RPTR of that node will point to next node in inorder traversal of T.



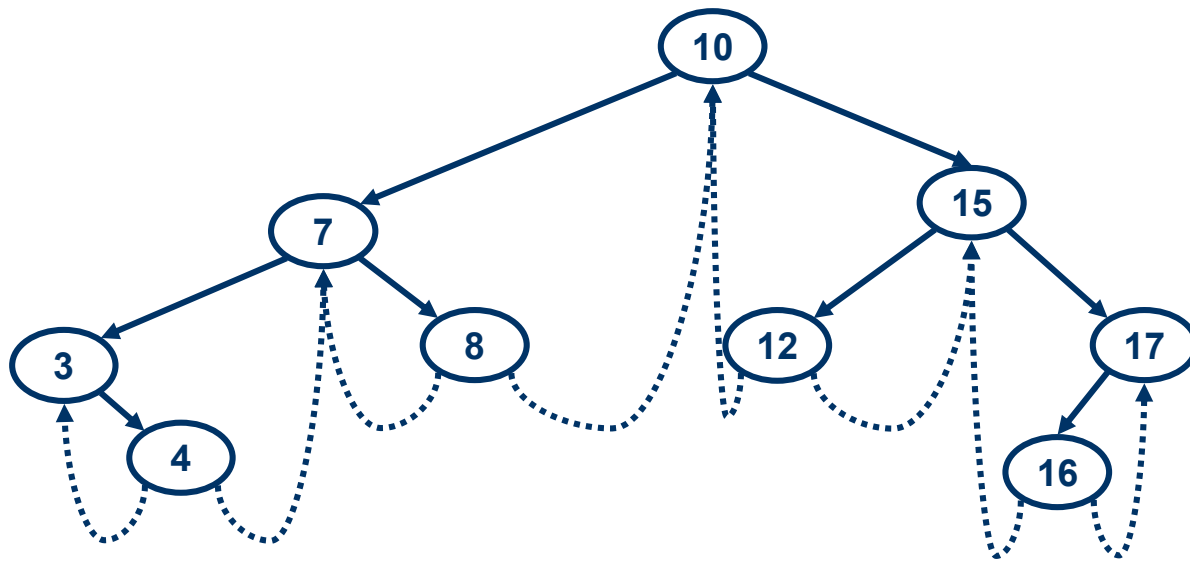
Inorder Sequence: 3 4 7 8 10 12 15 16 17



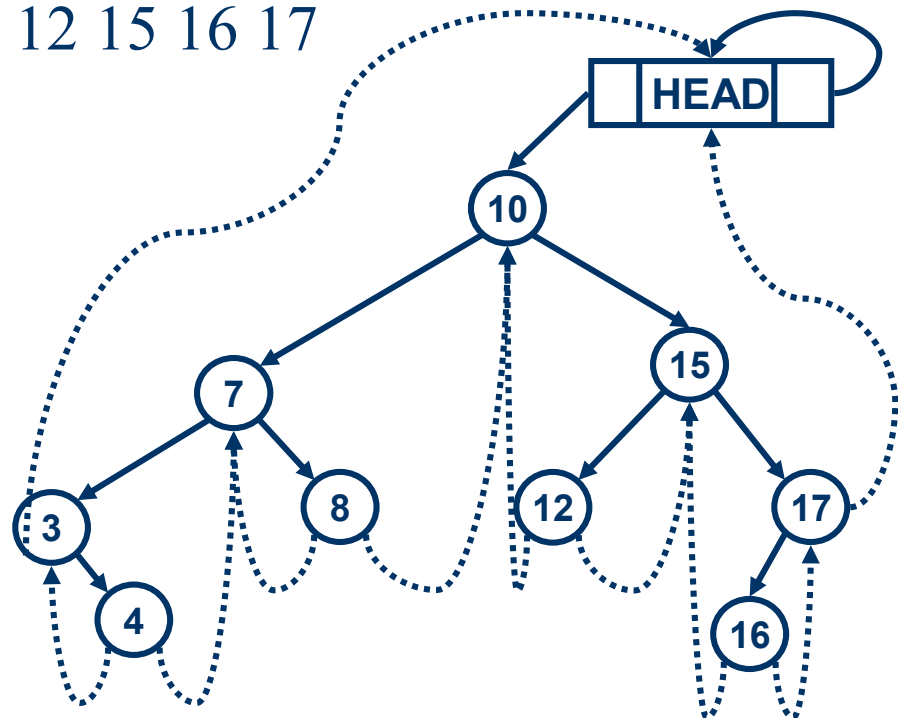
One-way threading with header node.

# Two way Threading

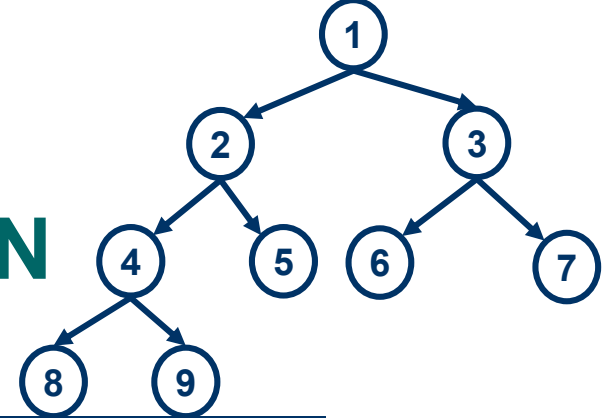
- In two way threading both the pointer LPTR and RPTR is used.
- RPTR is pointing to next node in inorder traversal of T.
- LPTR is pointing to preceding node in inorder traversal of T.
- Left pointer of first node and the right pointer of last node will contain NULL when T does not have header node.



Inorder Sequence: 3 4 7 8 10 12 15 16 17

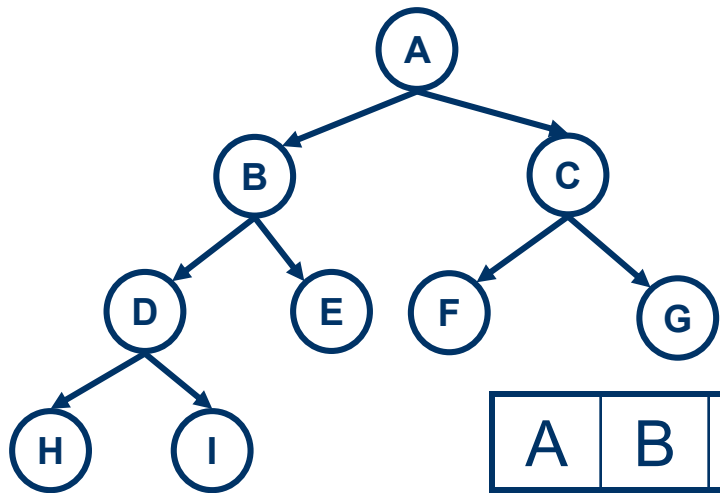


# ARRAY REPRESENTATION OF TREE

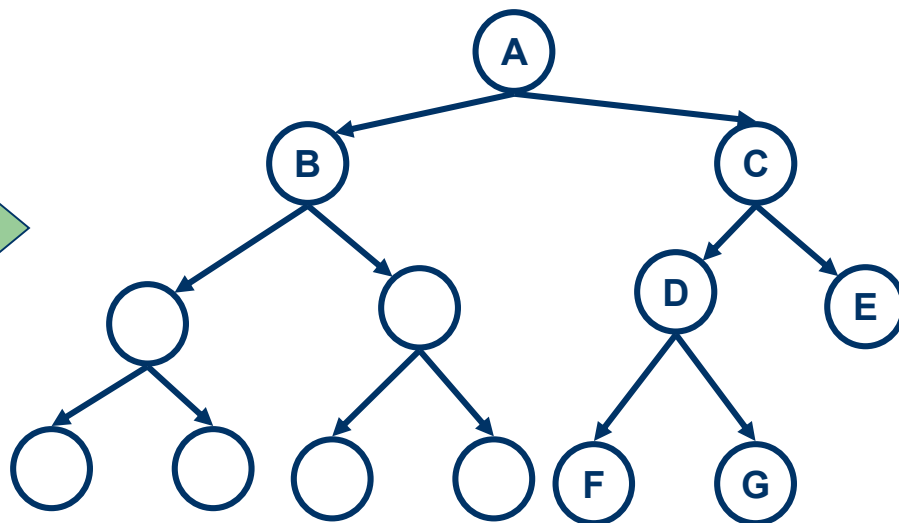
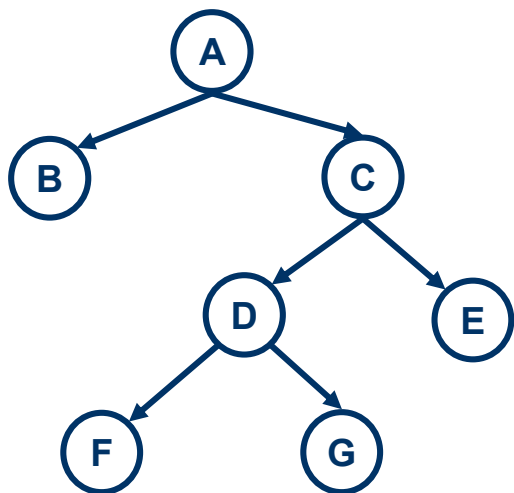


- N nodes of an almost complete binary tree can be numbered from 1 to n.
- Number assigned a left son is twice the number assigned to its father.
- Number assigned to a right son is 1 more than twice the number assigned to its father.
- So we can represent almost complete binary tree in array without using **left**, **right** and **father** pointer.
- We need to store only **info** field in appropriate index.

- If we are representing into C language then array start from 0. So we need to numbered from 0 to  $n-1$ .
- So for any father node  $P$  left son will reside on position  $2P+1$  and right son at index  $2P+2$ .



A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8



A	B	C			D	E					F	G
0	1	2	3	4	5	6	7	8	9	10	11	12

- Suppose if we want to visit the left or right son of any node  $P$  then  $P.LPTR$  can be implemented by  $2 * P + 1$ .
- $P.RPTR$  can be implemented by  $2 * P + 2$ .