

Trees

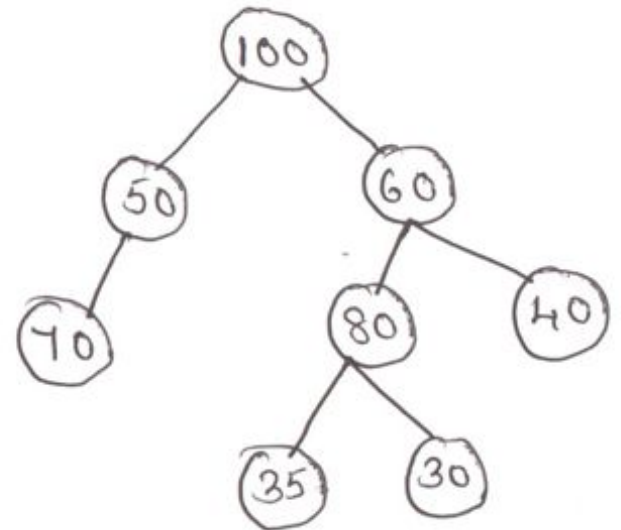
Module 4
MCA

Trees

- Tree is a finite set of nodes.
- Tree defined recursively
- A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r , called the root, and zero or more non-empty (sub) trees T_1, T_2, \dots, T_k each of whose roots are connected by a directed edge from r .
- A tree is a collection of N nodes, one of which is the root and $N-1$ edges.

Terminology of Tree

- Root node : No Parents
- Child node : Obtain from parent node
- Sibling : Child node which has common parent node
- Ancestors : node which meet till Root node
- Descendants : Nodes moving towards downwards
- Left descendants : Left side
- Right descendants : Right side
- Left subtree :
- Right Subtree:
- Degree of nodes : Number of subti



Cont.

- Leaf node : Node degree is zero
- Internal Node : Other node than leaf node
- Level of Node : Distance of node from the root.
Distance of root node to itself is zero.
- Depth(Height) : Max level of trees.

Cont.

- The root of each subtree is said to be a **child** of r and r is said to be the **parent** of each subtree root.
- **Leaves**: nodes with no children (also known as external nodes)
- **Internal Nodes**: nodes with children
- **Siblings**: nodes with the same parent

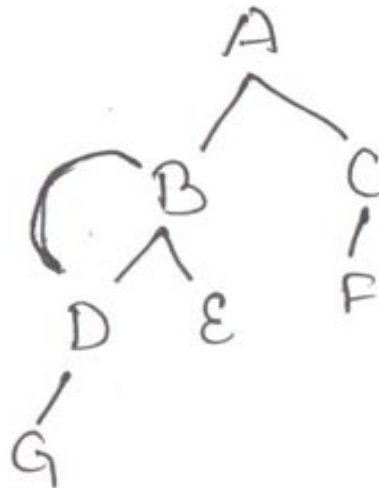
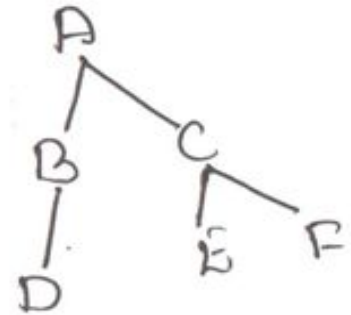
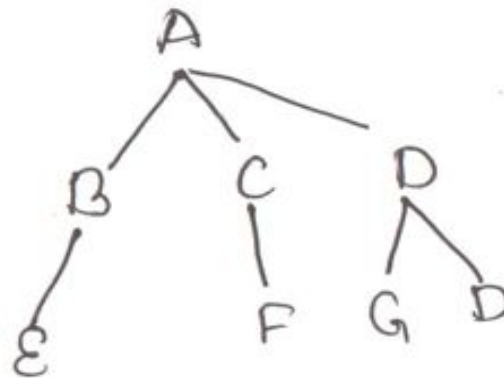
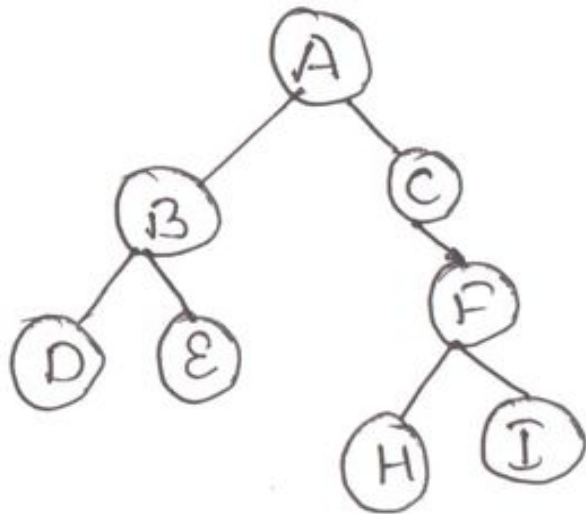
Cont.

- Depth (of node): the length of the unique path from the root to a node.
- Depth (of tree): The depth of a tree is equal to the depth of its deepest leaf.
- Height (of node): the length of the longest path from a node to a leaf.
 - All leaves have a height of 0
 - The height of the root is equal to the depth of the tree

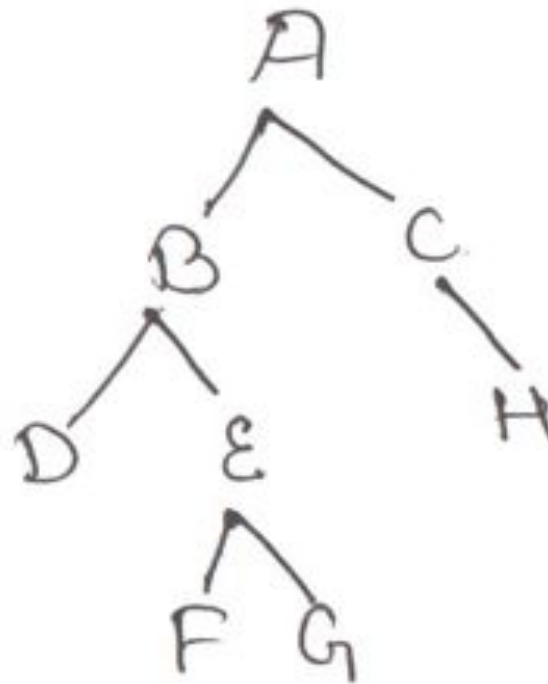
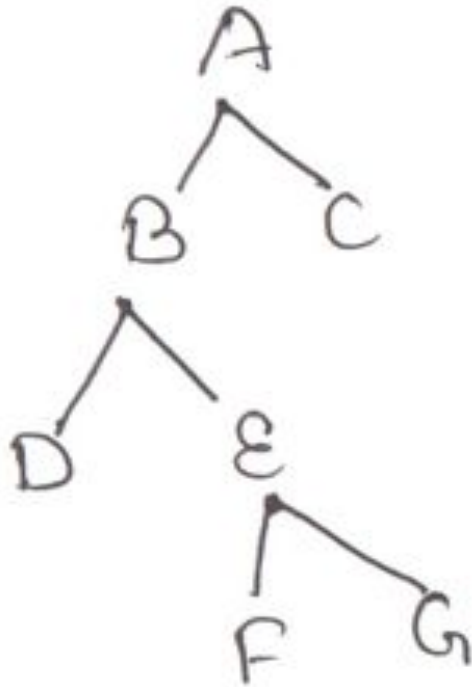
Types of Tree

- Binary Tree : Subtree can have 0 or 1 or 2 nodes
- Strictly Binary Tree : All nodes have either 0 or 2 nodes
- Complete Binary Tree: 2 properties: All nodes must be filled with 2 nodes and all leaf node are on same level.
- Almost Binary Tree: All levels are completely filled except last possible Left to Right node. Last node must be left aligned.
- Skewed Binary Tree : No child or must have one child

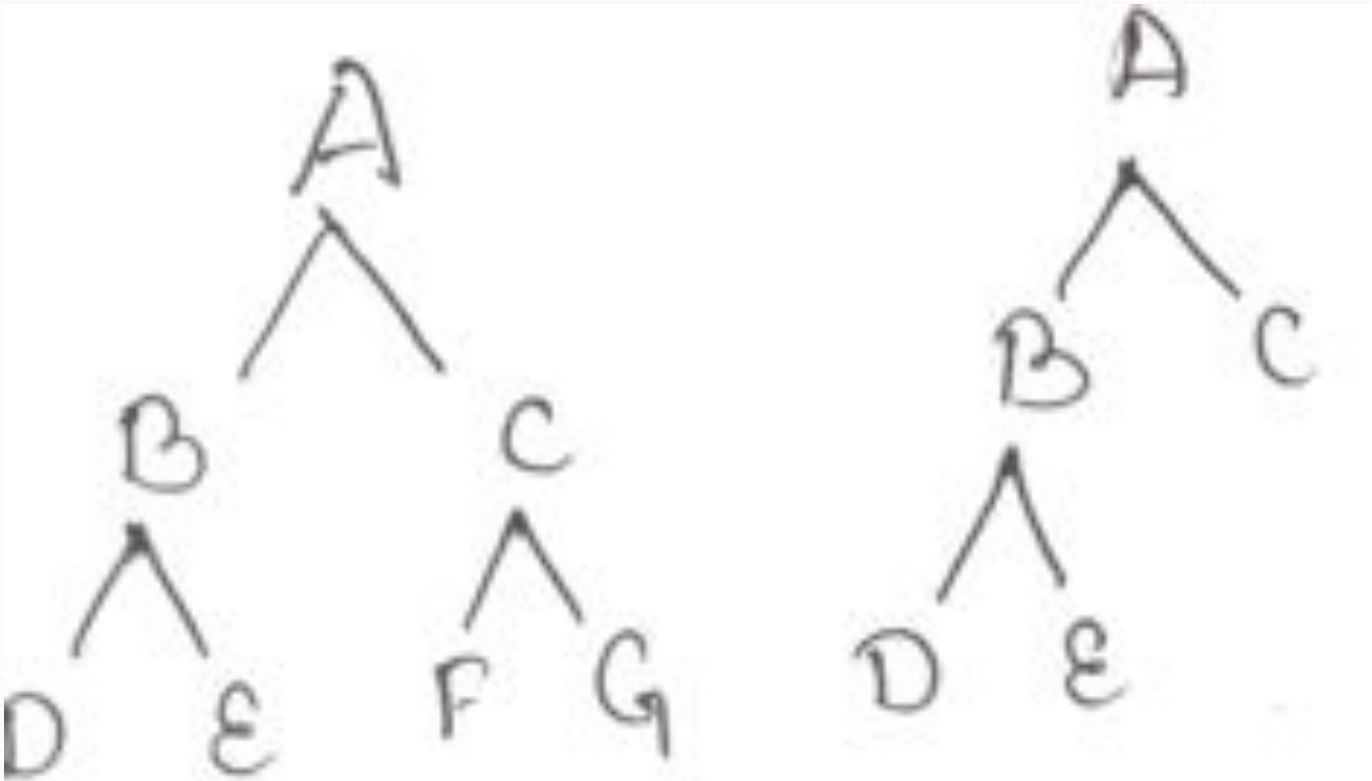
Binary Tree



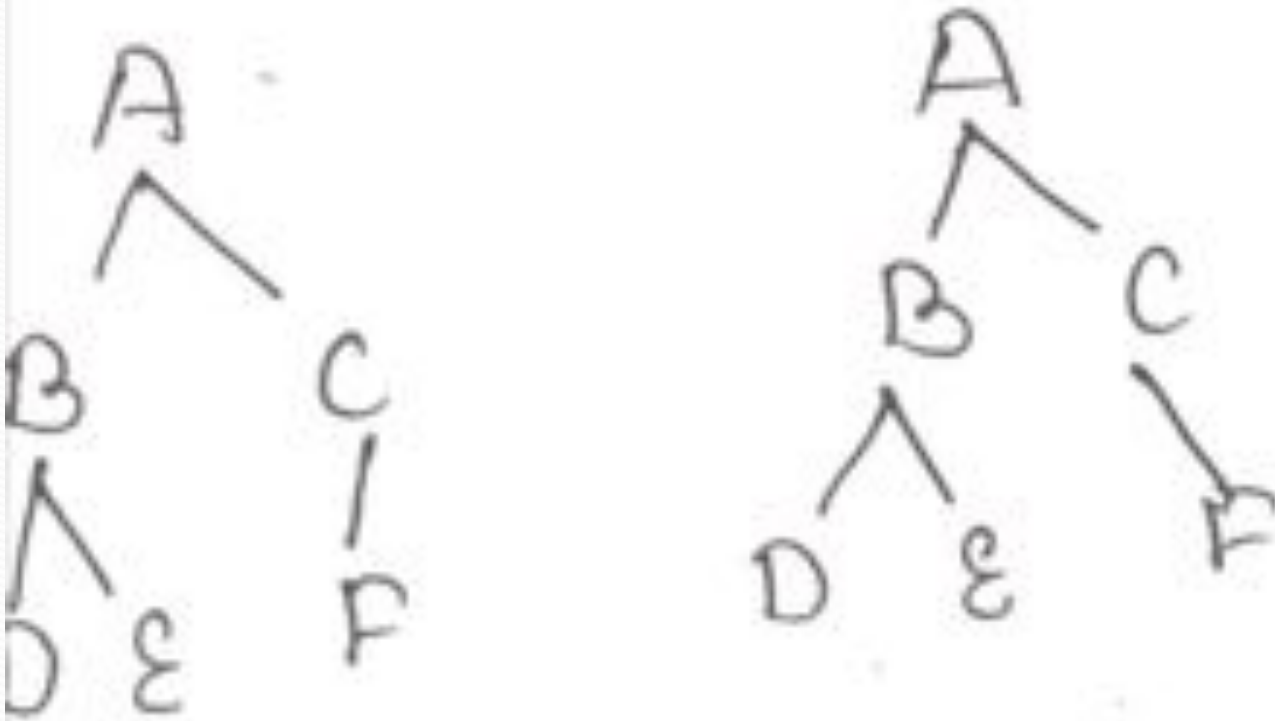
Strictly Binary Tree



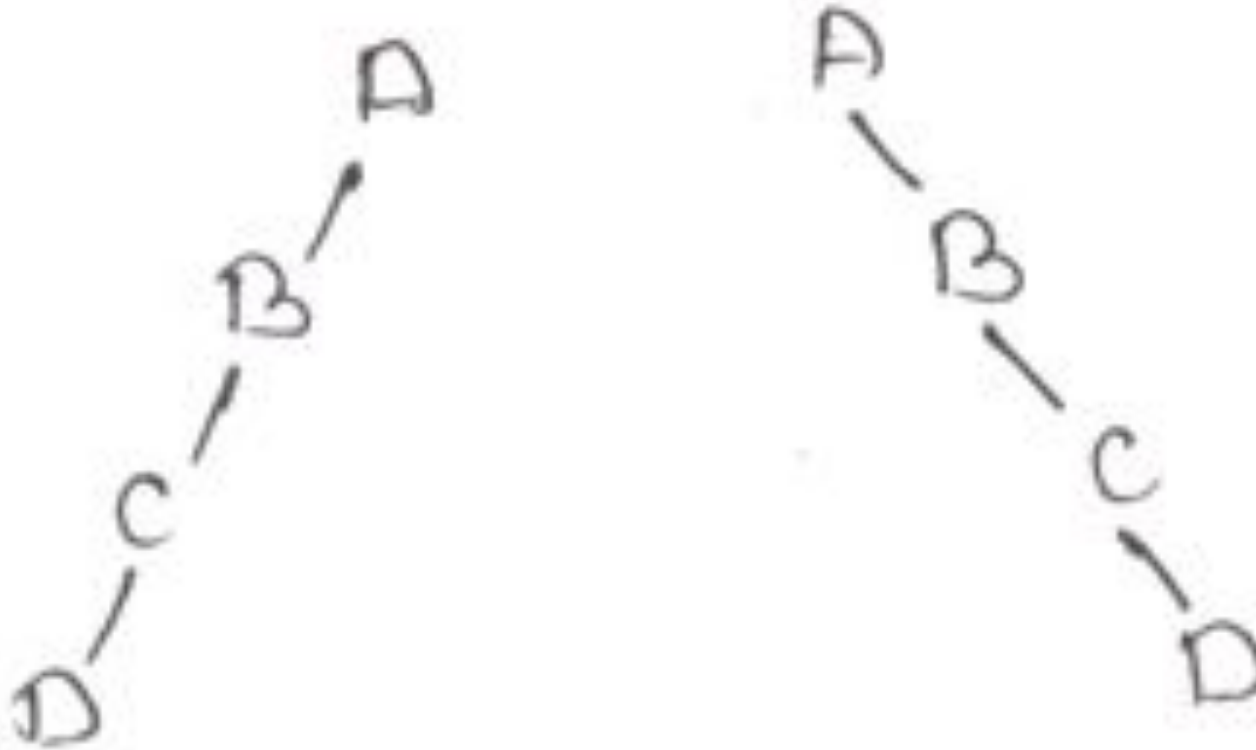
Complete/Full Binary Tree



Almost Binary Tree



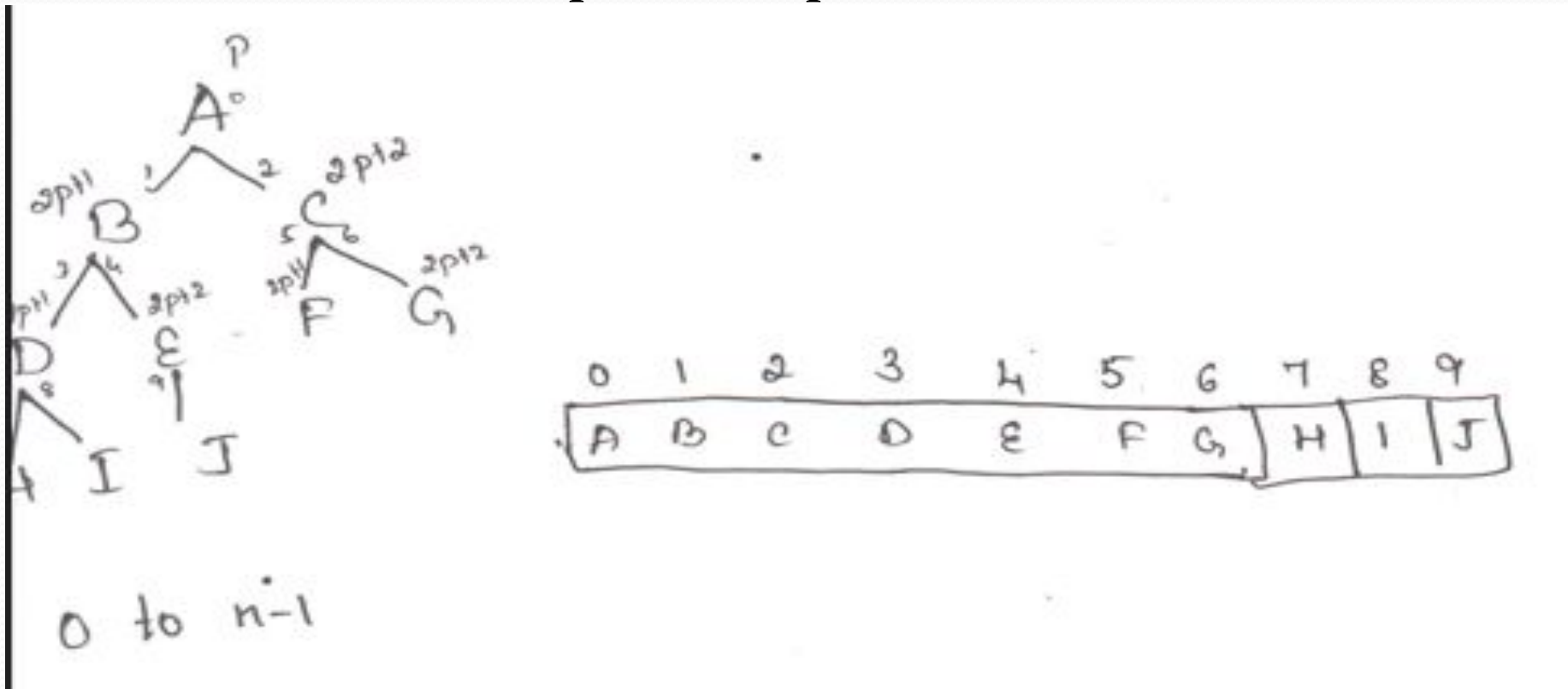
Skewed Binary Tree

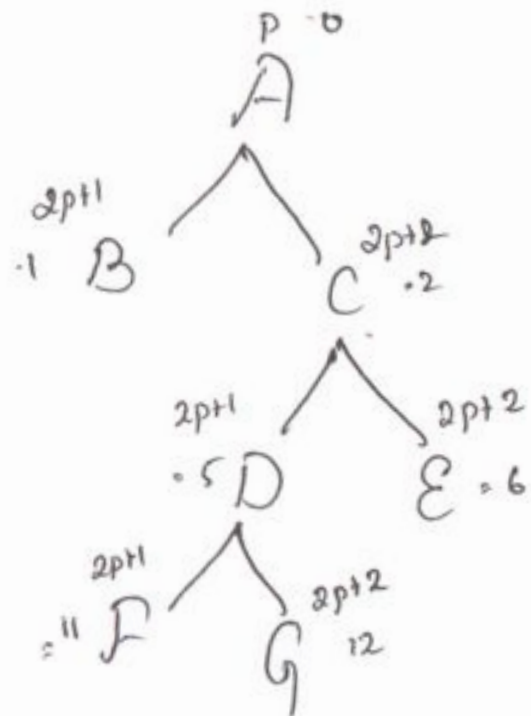


Representation on Binary Tree

- Array representation of Binary Tree

Parent: p L : $2p+1$ R: $2p+2$

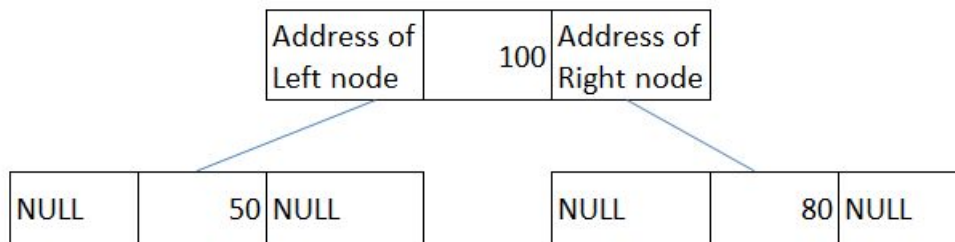


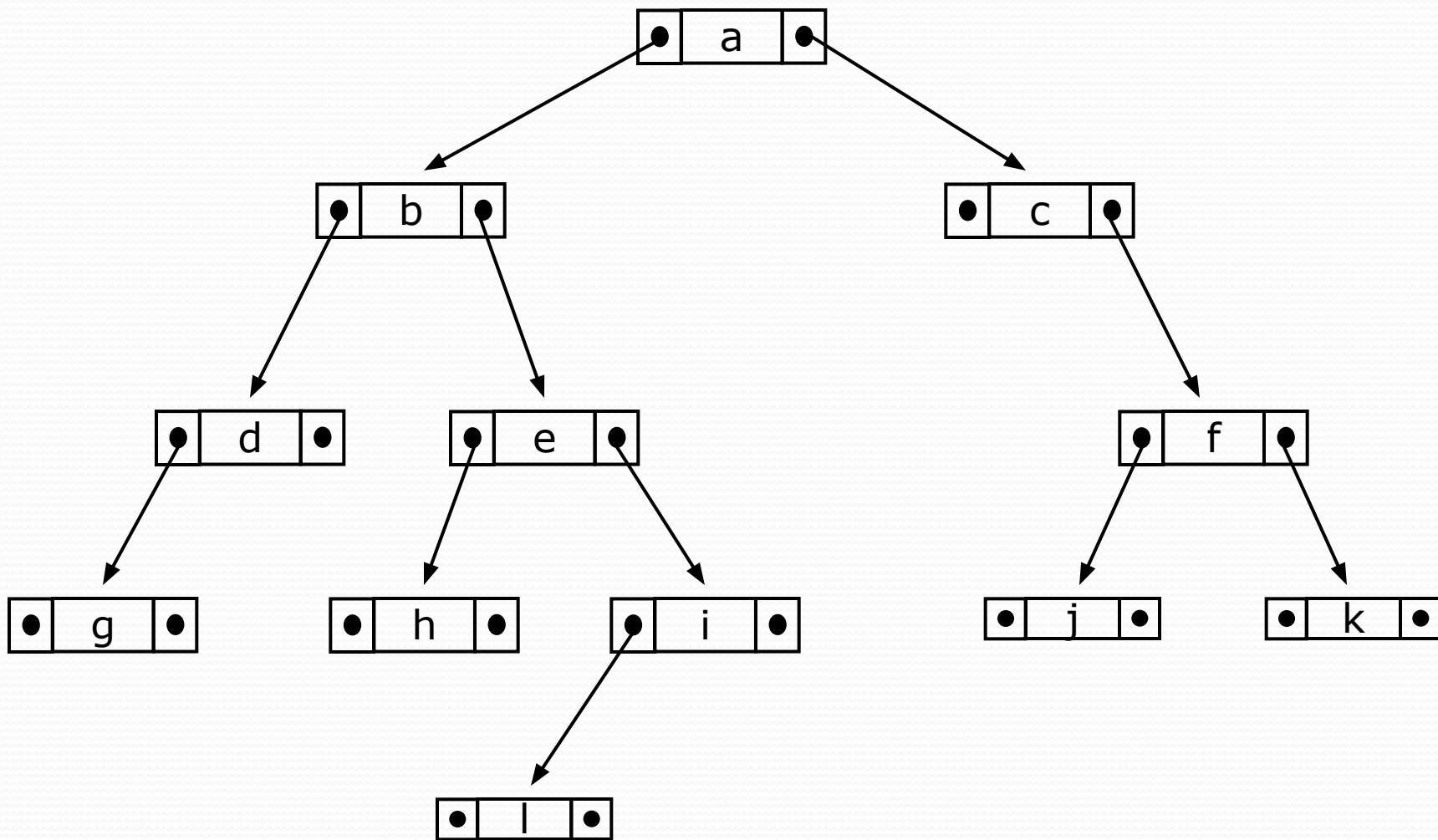


0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			D	E					F	G

Cont.

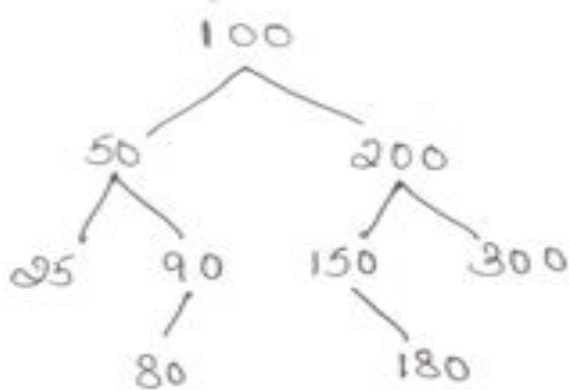
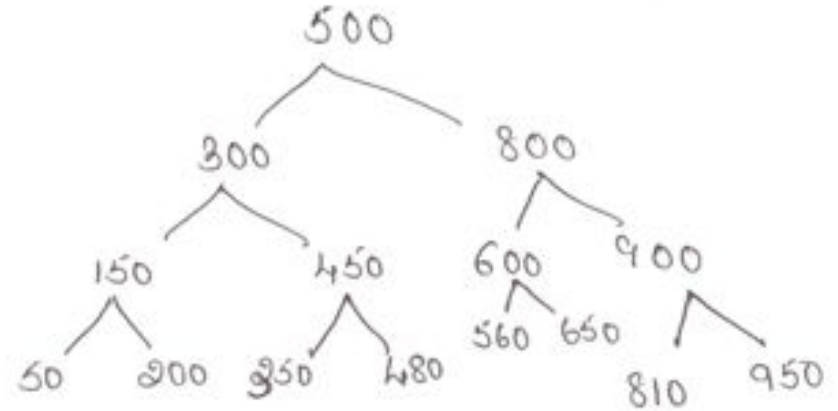
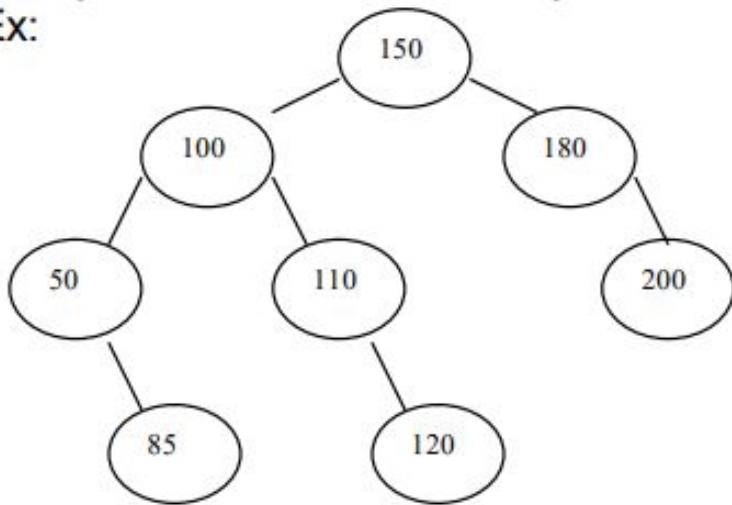
- Linked list representation of Binary Tree





Binary Search Tree

Ex:



Binary Tree Traversals

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive
- Since a binary tree has three “parts,” there are six possible ways to traverse the binary tree:
 - root, left, right
 - left, root, right
 - left, right, root

Traversal of a Tree :

The rules for these traversals are as below:

Pre-order traversal:

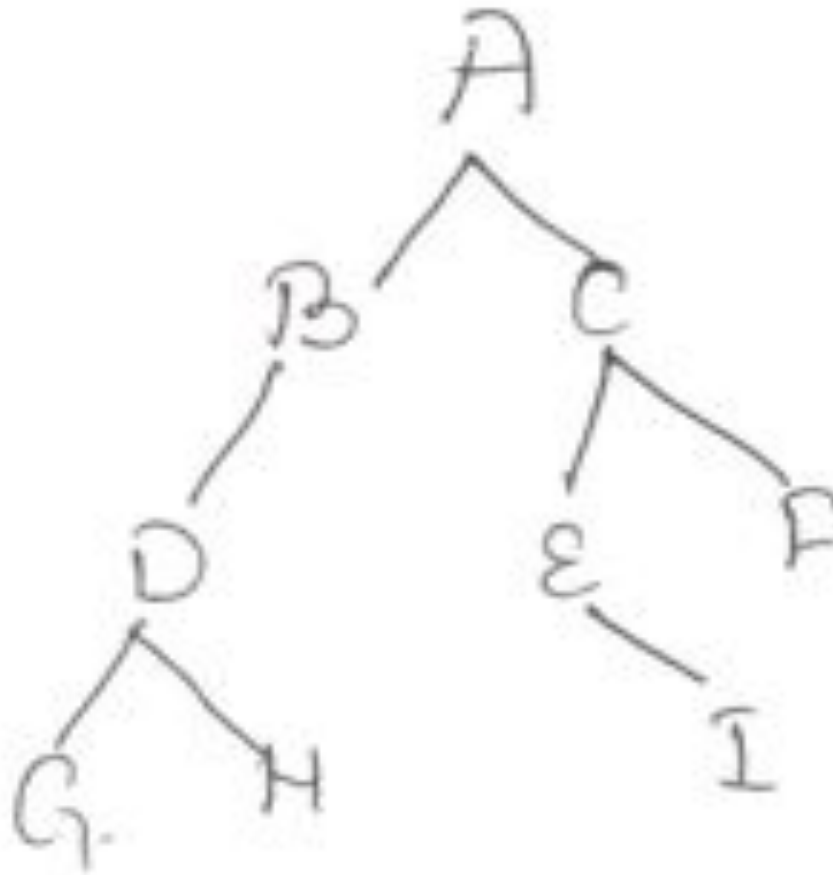
- i) Visit the root.
- ii) Traverse left subtree using pre-order traversal method.
- iii) Traverse right subtree using pre-order traversal method.

In-order traversal:

- i) Traverse left subtree using in-order traversal method.
- ii) Visit the root.
- iii) Traverse right subtree using in-order traversal method.

Post-order traversal:

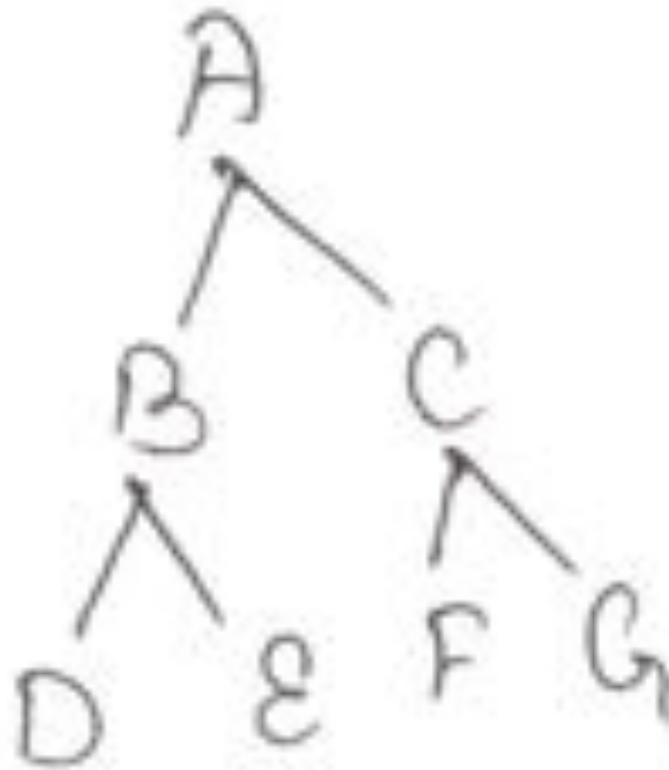
- i) Traverse left subtree using post-order traversal method.
- ii) Traverse right subtree using pos-order traversal method.
- iii) Visit the root.



Inorder : GDHBAEICF

Preorder : ABDGHCEIF

Postorder : GHDBIEFCA



Inorder : DBEAFCCG

Preorder : ABDECFCG

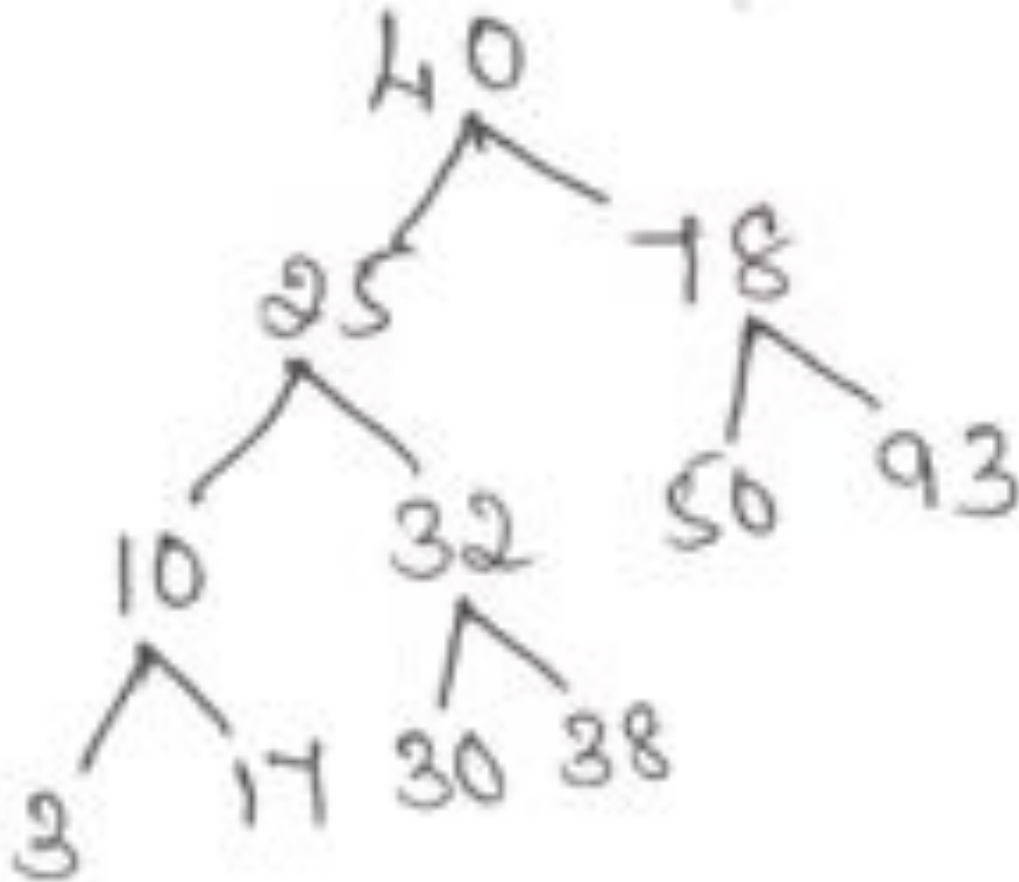
Postorder : DEBFGCA



Inorder : 304344485657668491

Preorder : 574330484456916684

Postorder : 304456484384669157



Inorder : 310172530323840507893

Preorder : 402510317323038785093

Postorder : 317103038322550937840

Preorder traversal

- In preorder, the root is visited *first*
- Here's a preorder traversal to print out all the elements in the binary tree:

```
public void preorderPrint(BinaryTree bt)
{
    if (bt == null) return;
    System.out.println(bt.value);
    preorderPrint(bt.leftChild);
    preorderPrint(bt.rightChild);
}
```


Inorder traversal

- In *inorder*, the root is visited *in the middle*
- Here's an inorder traversal to print out all the elements in the binary tree:

```
public void inorderPrint(BinaryTree bt) {  
    if (bt == null) return;  
    inorderPrint(bt.leftChild);  
    System.out.println(bt.value);  
    inorderPrint(bt.rightChild);  
}
```

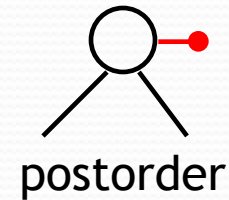
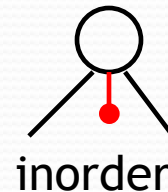
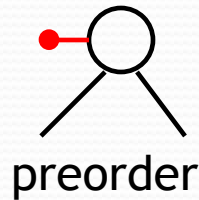
Postorder traversal

- In *postorder*, the root is visited *last*
- Here's a *postorder* traversal to print out all the elements in the binary tree:

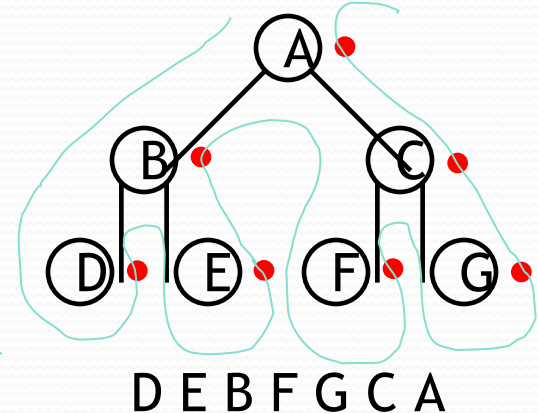
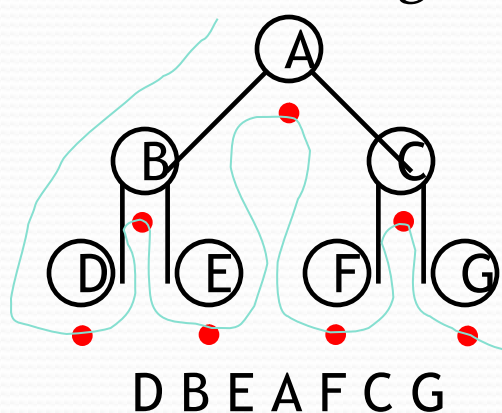
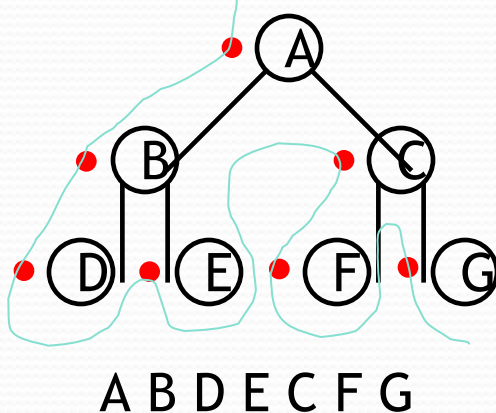
```
public void postorderPrint(BinaryTree  
bt) {  
    if (bt == null) return;  
    postorderPrint(bt.leftChild);  
    postorderPrint(bt.rightChild);  
    System.out.println(bt.value);  
}
```

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:



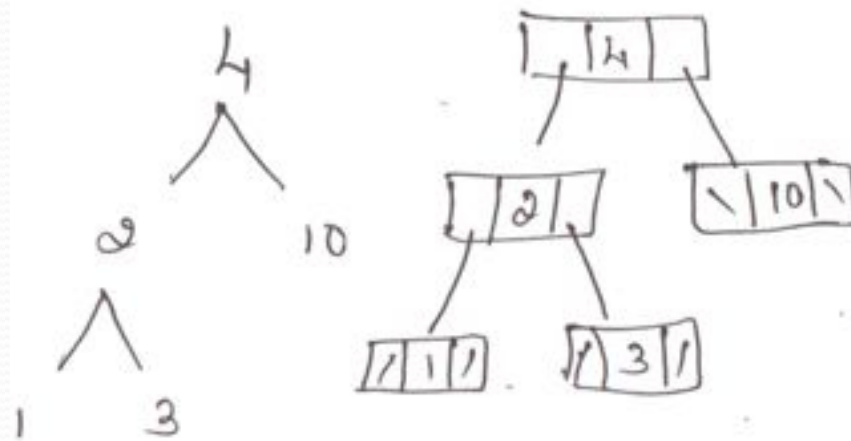
- To traverse the tree, collect the flags:



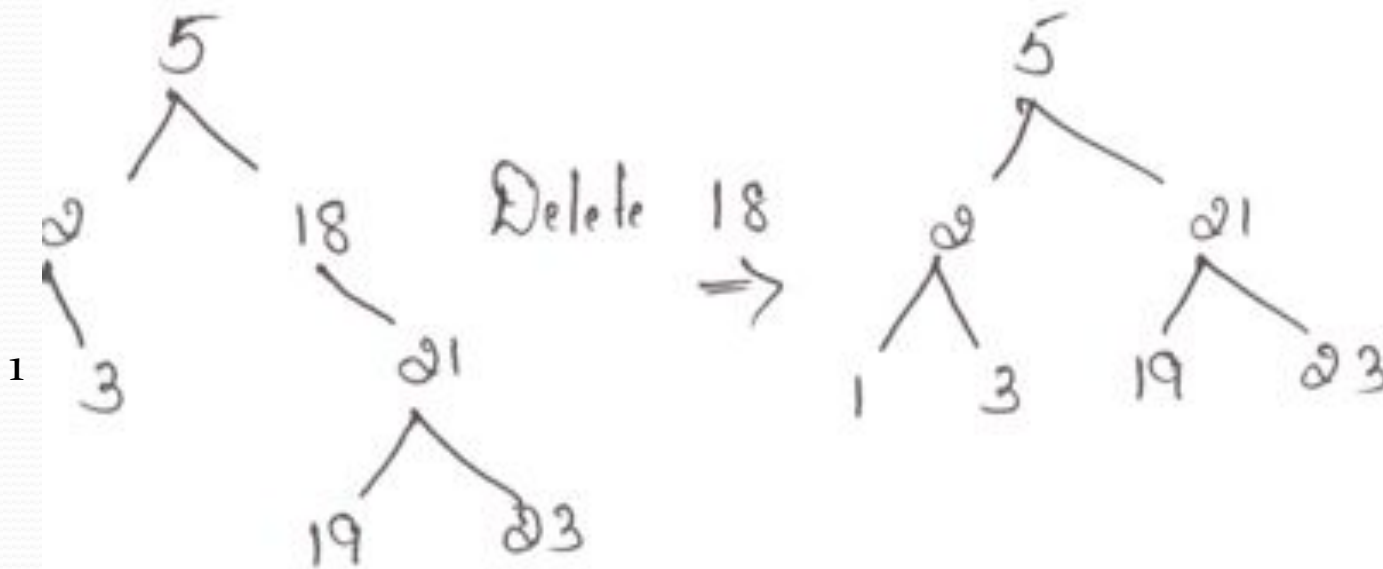
Other traversals

- The other traversals are the reverse of these three standard ones
 - That is, the right subtree is traversed before the left subtree is traversed
- Reverse preorder: root, right subtree, left subtree
- Reverse inorder: right subtree, root, left subtree
- Reverse postorder: right subtree, left subtree, root

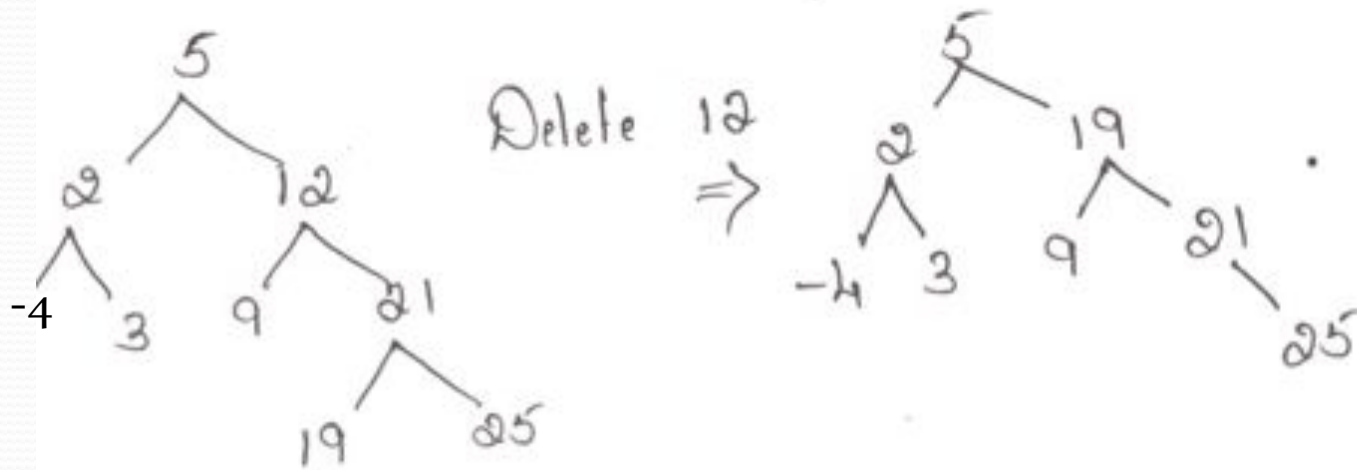
Case 1: Node to be removed has no children



Case 2: Node to be removed has one child



Case 3 : Node to be removed has 2 children (Inorder Successor)



Inorder successor : -4 2 3 5 9 12 19 21 25

Binary Tree Traversals

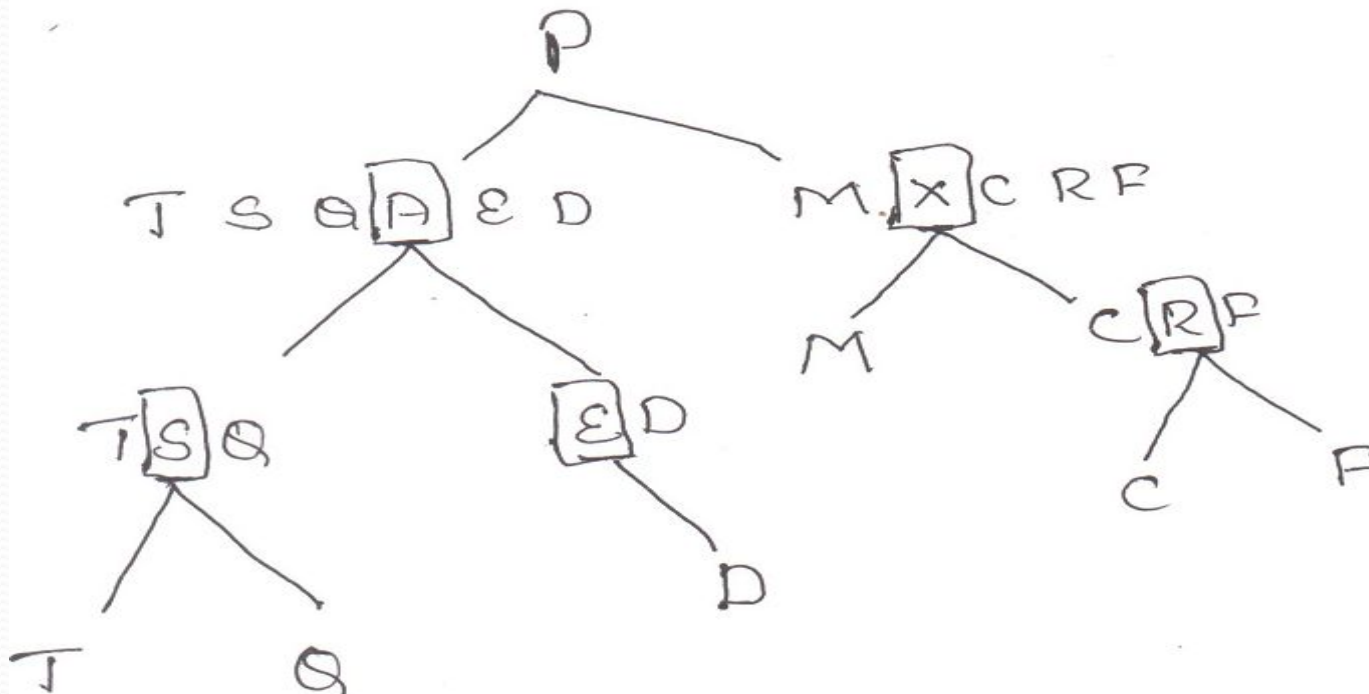
- Preorder Traversal : Root L R
- Inorder Traversal : L Root R
- Postorder Traversal : L R Root

How to construct a BST given traversal

- Inorder : T S Q A E D P M X C R F
- Preorder: P A S T Q E D X M R C F

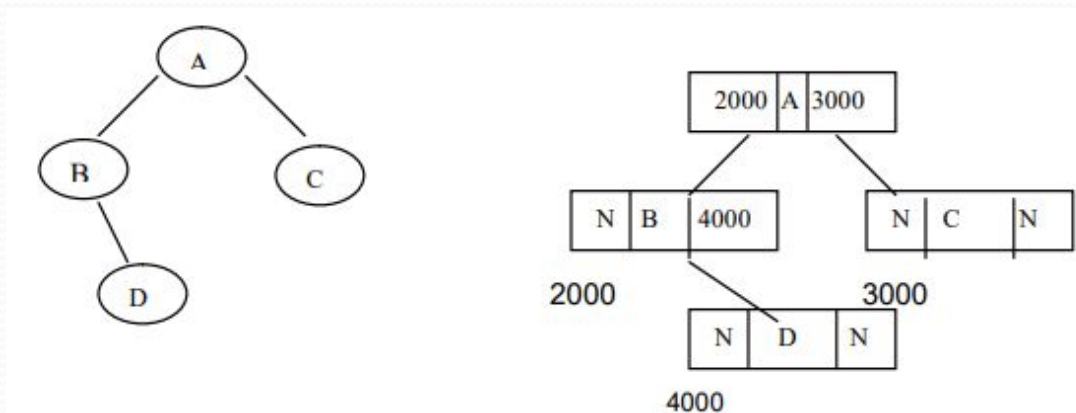
↳ Root Right Inorder : T S Q A E D P M X C R F


↳ Root Left Right Preorder : P A S T Q E D X M R C F



Threaded binary trees

- Since each of the traversal methods are recursive in nature, every time the program needs to push the items into the stack and to pop-up from the stack





```
struct node
{
int data;
int lchild;
int rchild;
struct node *llink;
struct node *rlink;
}; typedef struct node *NODE;
```

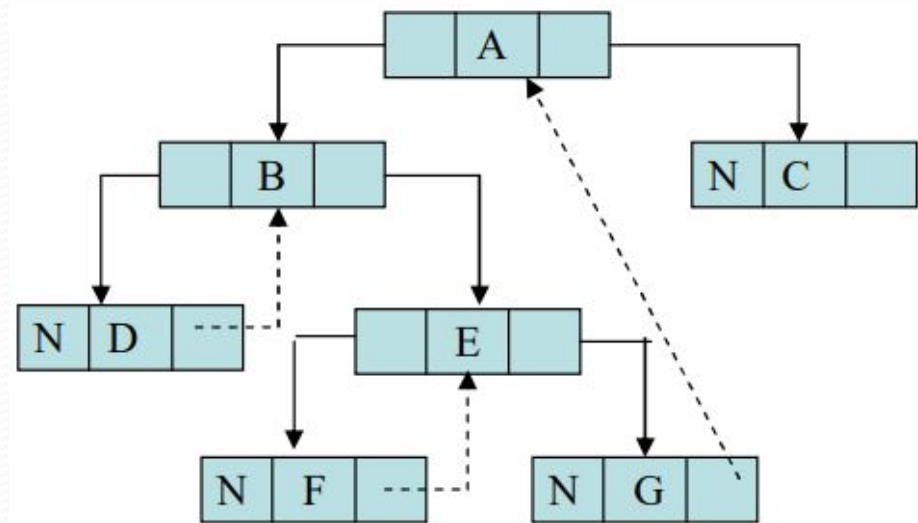
Threading may correspond to any of the three traversals.

- In-order Threading
- Pre-order Threading
- Post-order Threading

Each of these type may be of

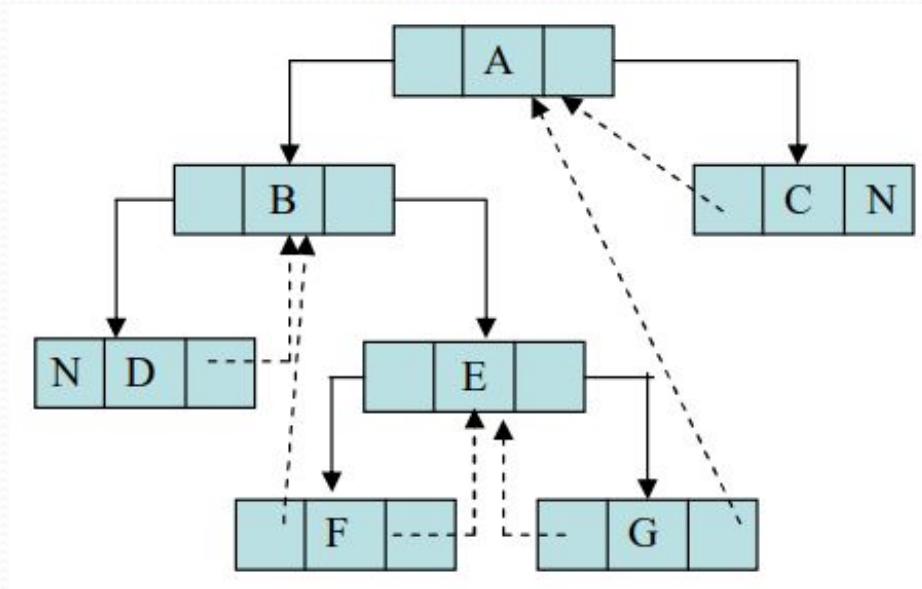
- One-way threading
- Two-way threading

- One-way In-order Threading:
 - The right link field of the node will have the address of its in-order successor.



- Two-way In-order Threading:

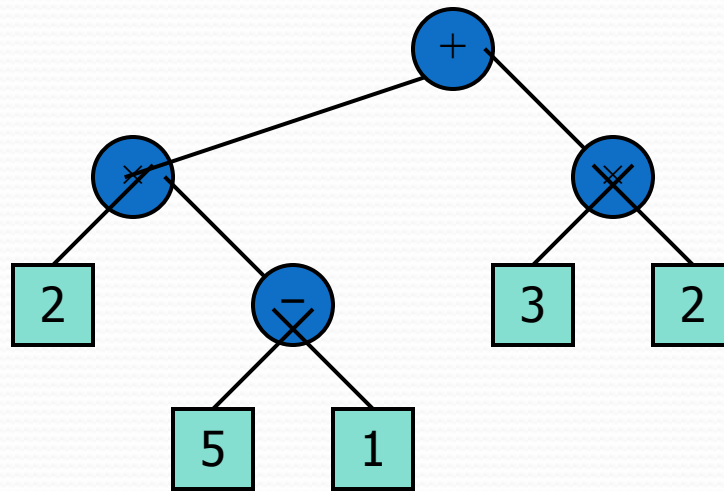
- The right link field of the node will have the address of its in-order successor.
- And the left link field of the node will have the address of its in-order predecessor.
- It is also known as Fully threaded binary tree.





Arithmetic Expression Tree

- Binary tree for an arithmetic expression
 - internal nodes: operators
 - leaves: operands
- Example: arithmetic expression tree for the expression
$$((2 \times (5 - 1)) + (3 \times 2))$$

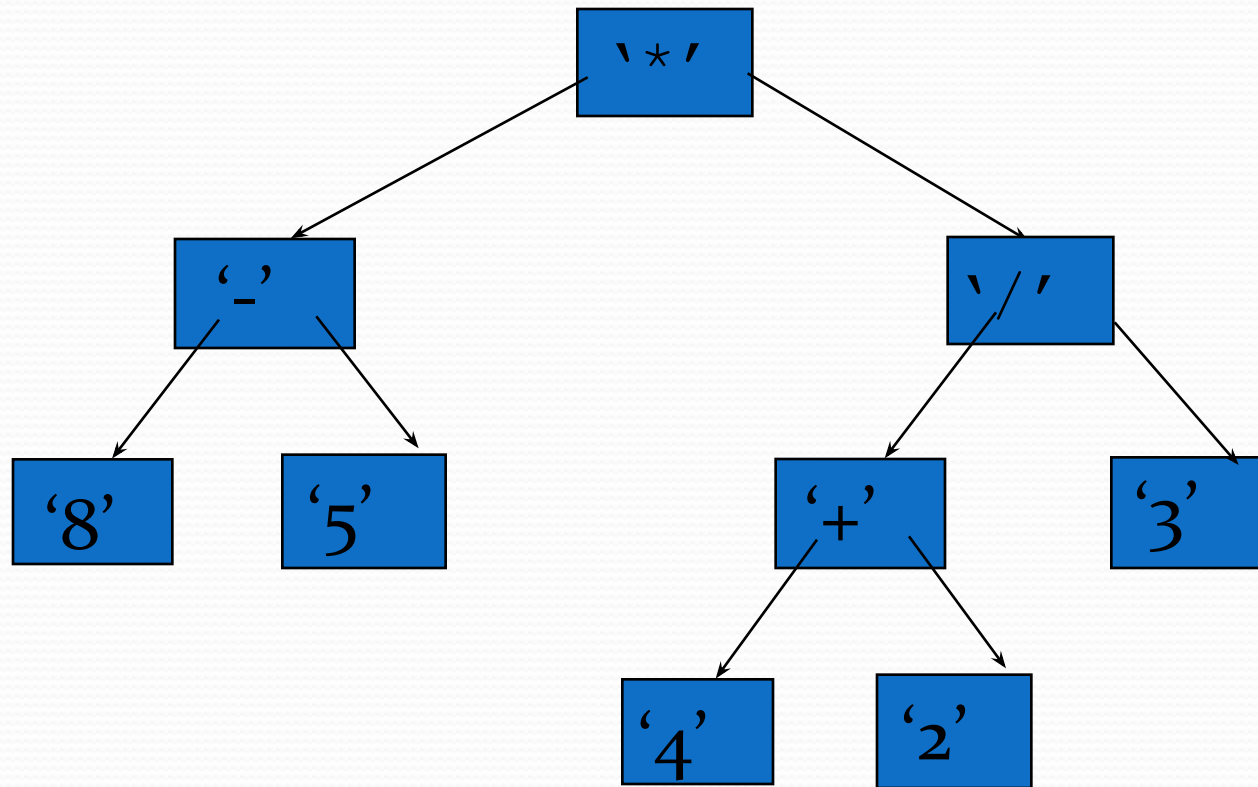


A Binary Expression Tree is . . .

A special kind of binary tree in which:

1. Each **leaf node** contains a single operand
2. Each **nonleaf node** contains a single binary operator
3. The left and right subtrees of an operator node represent **subexpressions** that must be evaluated **before** applying the operator at the root of the subtree.

A Four-Level Binary Expression



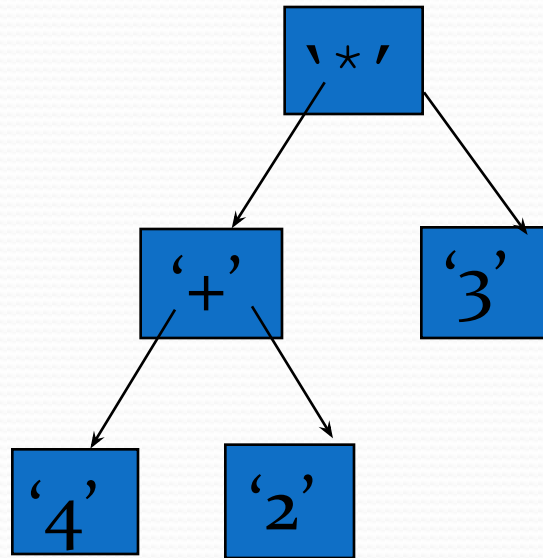
Levels Indicate Precedence

The levels of the nodes in the tree indicate their relative precedence of evaluation (we do not need parentheses to indicate precedence).

Operations at higher levels of the tree are evaluated later than those below them.

The operation at the root is always the last operation performed.

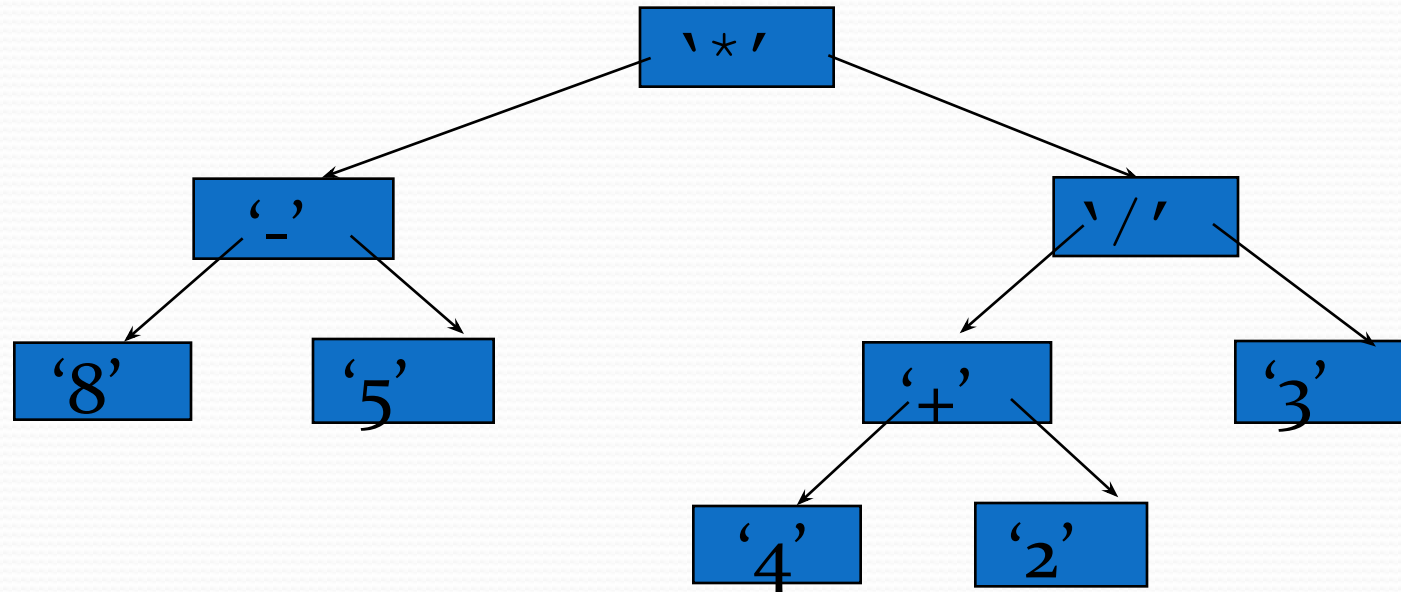
A Binary Expression Tree



What value does it have?

$$(4 + 2) * 3 = 18$$

Easy to generate the infix, prefix, postfix expressions (how?)

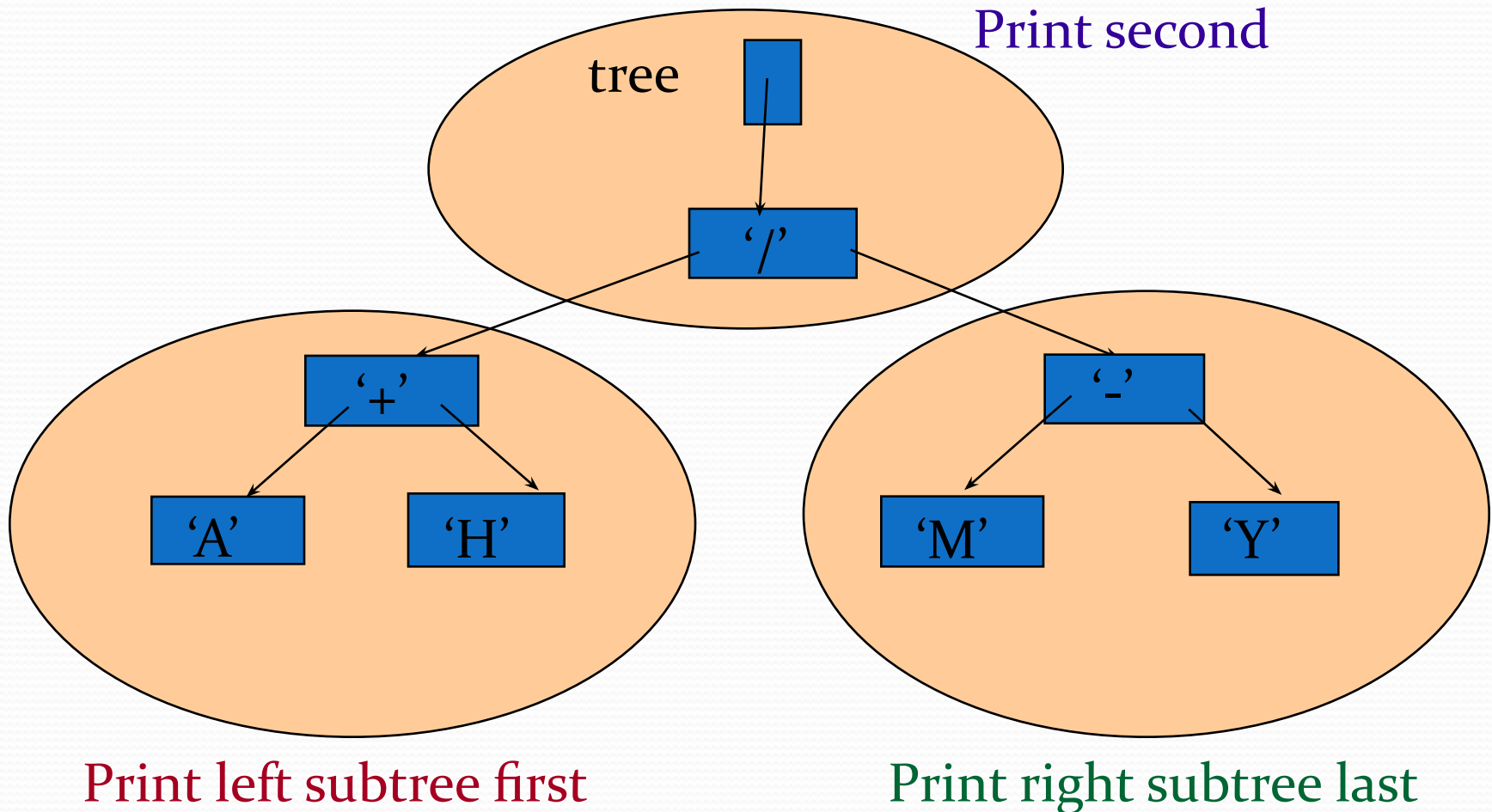


Infix: $((8 - 5) * ((4 + 2) / 3))$

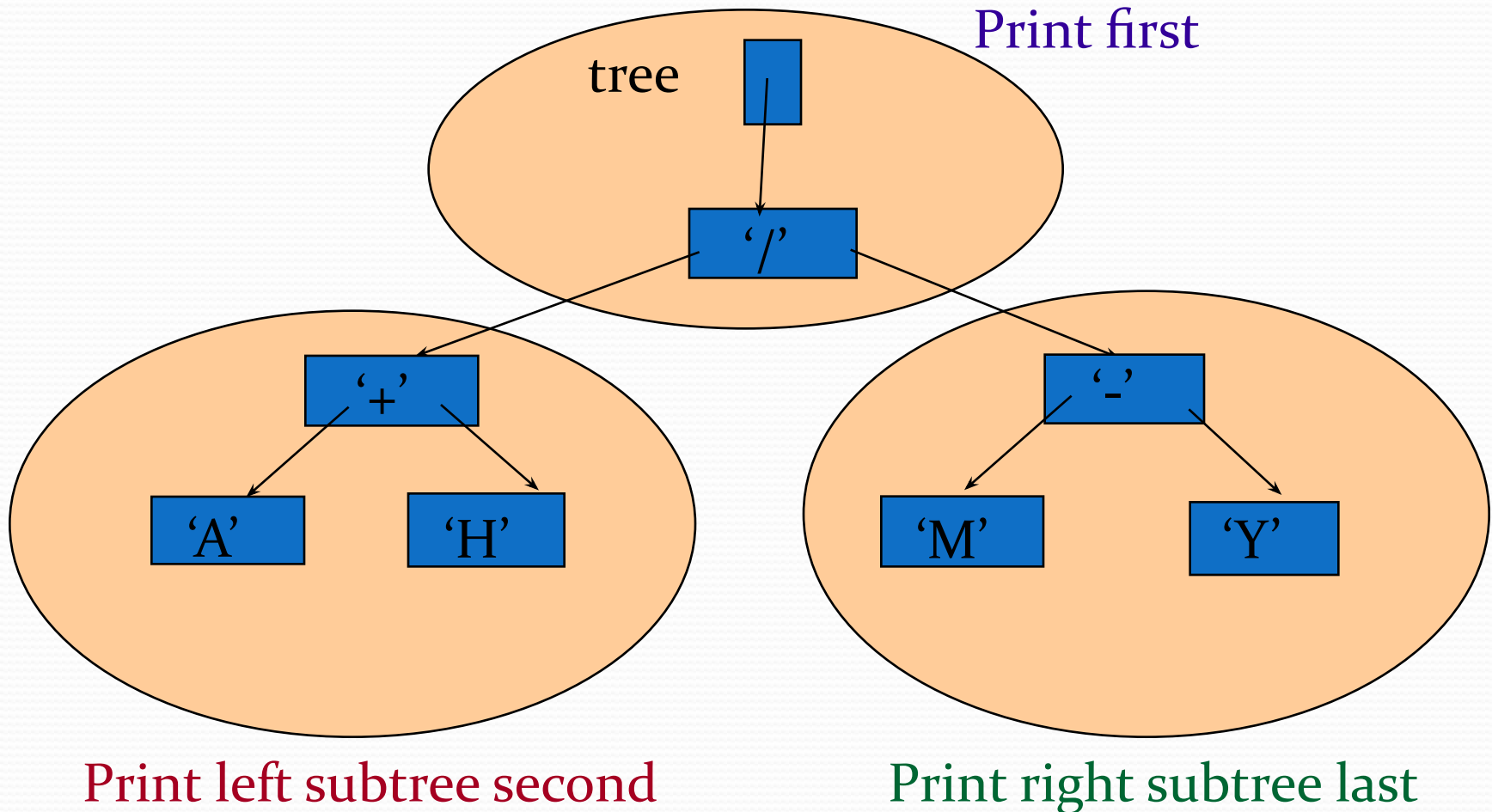
Prefix: $* - 8 5 / + 4 2 3$

Postfix: $8 5 - 4 2 + 3 / *$

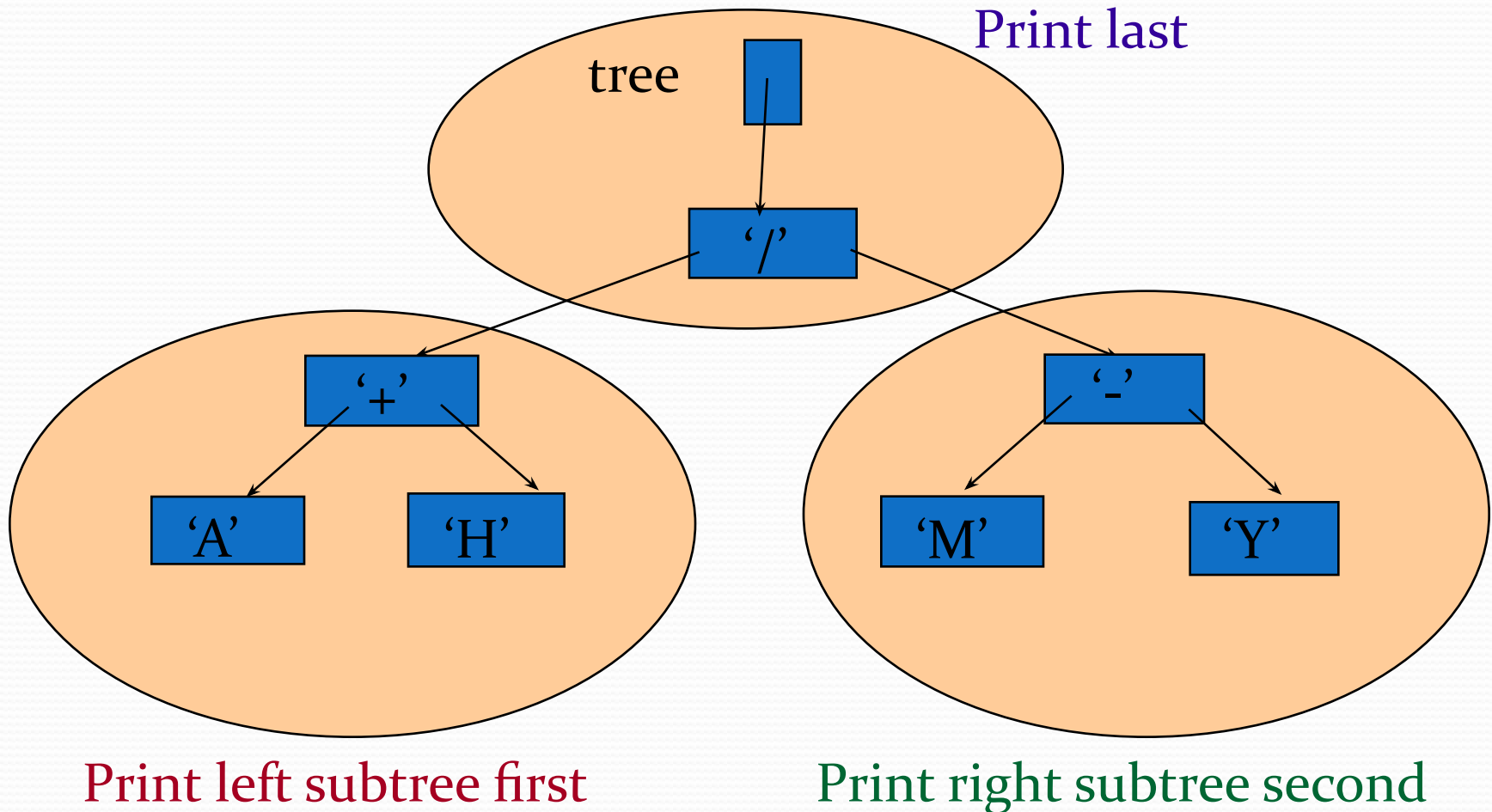
Inorder Traversal: $(A + H) / (M - Y)$



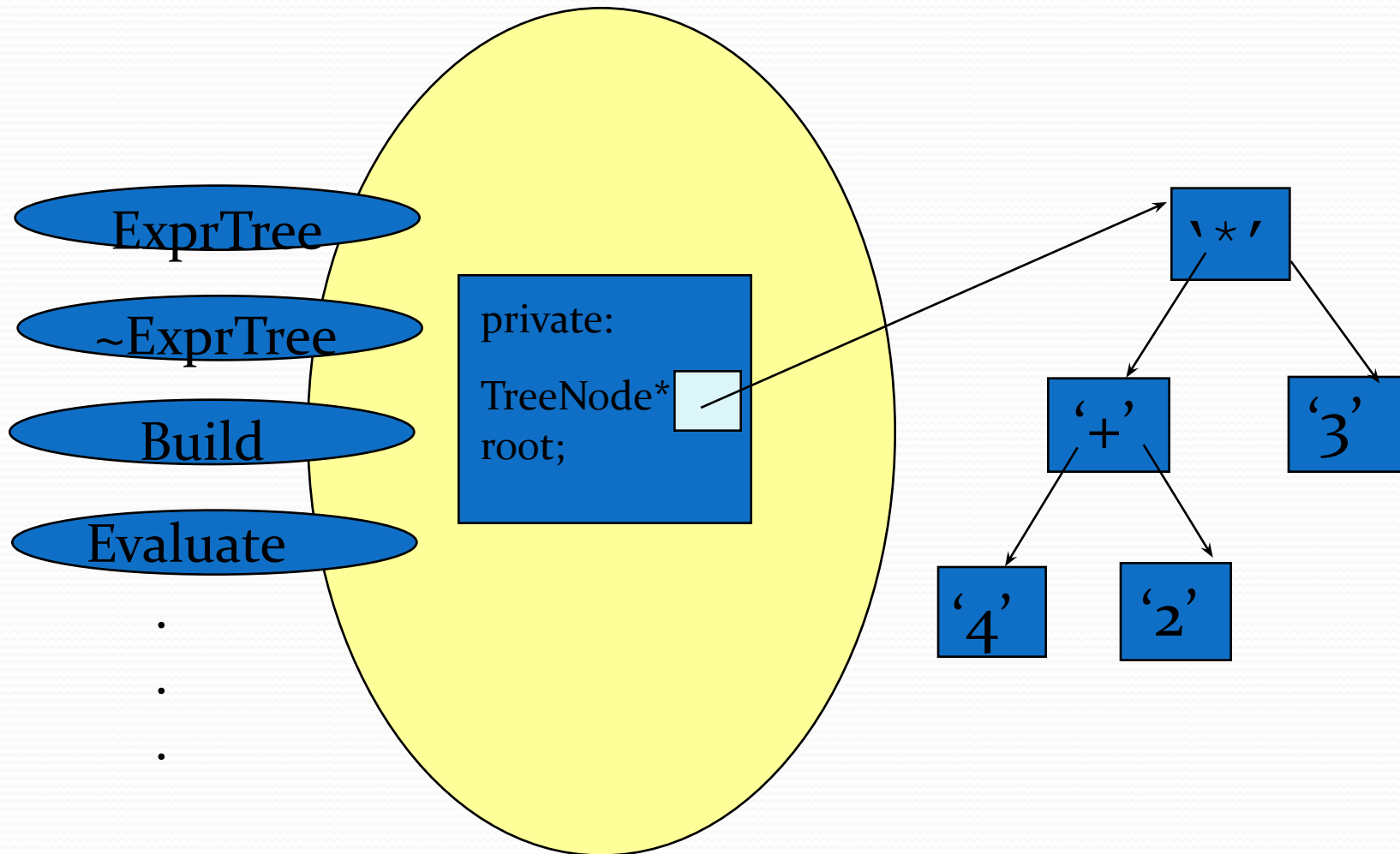
Preorder Traversal: $/ + A H - M Y$



Postorder Traversal: A H + M Y - /

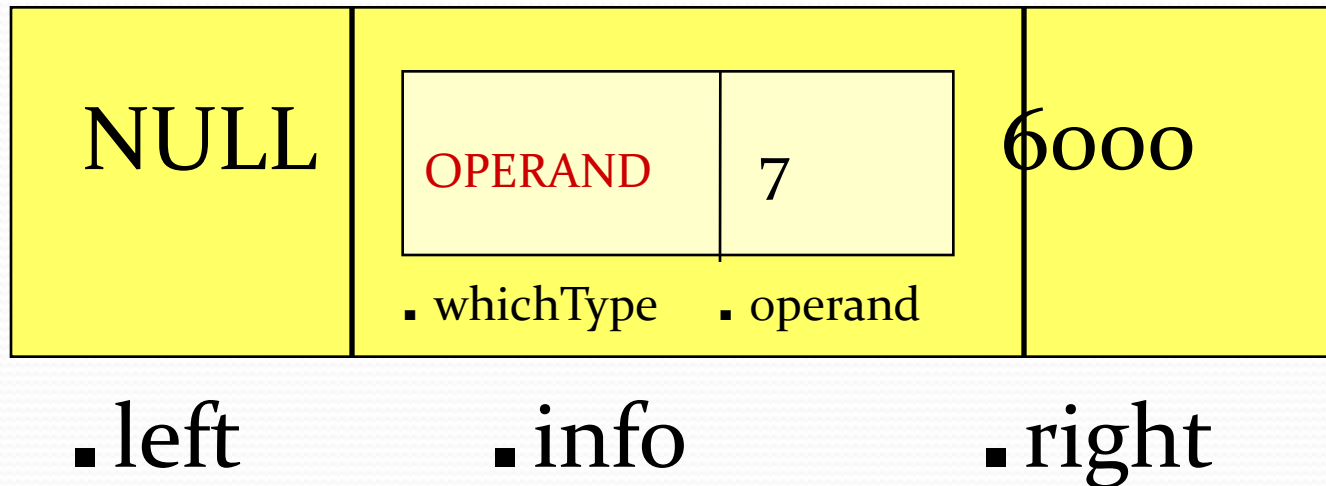


class ExprTree



Each node contains two pointers

```
struct TreeNode
{
    InfoNode info ;           // Data member
    TreeNode* left ;         // Pointer to left child
    TreeNode* right ;        // Pointer to right child
};
```



InfoNode has 2 forms

```
enum OpType { OPERATOR, OPERAND } ;
```

```
struct InfoNode
```

```
{  
    OpType    whichType;  
    union      // ANONYMOUS union  
    {  
        char  operation ;  
        int   operand ;  
    }  
};
```

OPERATOR	‘+’
----------	-----

▪ whichType ▪ operation

OPERAND	7
---------	---

▪ whichType ▪ operand

```
int Eval ( TreeNode* ptr )
{
    switch ( ptr->info.whichType )
    {
        case OPERAND : return ptr->info.operand ;
        case OPERATOR :
            switch ( tree->info.operation )
            {
                case '+' : return ( Eval ( ptr->left ) + Eval ( ptr->right ) ) ;

                case '-' : return ( Eval ( ptr->left ) - Eval ( ptr->right ) ) ;

                case '*' : return ( Eval ( ptr->left ) * Eval ( ptr->right ) ) ;
                case '/' : return ( Eval ( ptr->left ) / Eval ( ptr->right ) ) ;
            }
        }
    }
}
```

Building a Binary Expression Tree from an expression in prefix notation

- Insert new nodes, each time moving to the left until an operand has been inserted.
- Backtrack to the last operator, and put the next node to its right.
- Continue in the same pattern.