SHRI RAMDEOBABA COLLEGE OF ENGINEERING AND MANAGEMENT, NAGPUR - 440013

Data structures and algorithm (CSP252)
III semester section a

Course Coordinator: Rina Damdoo

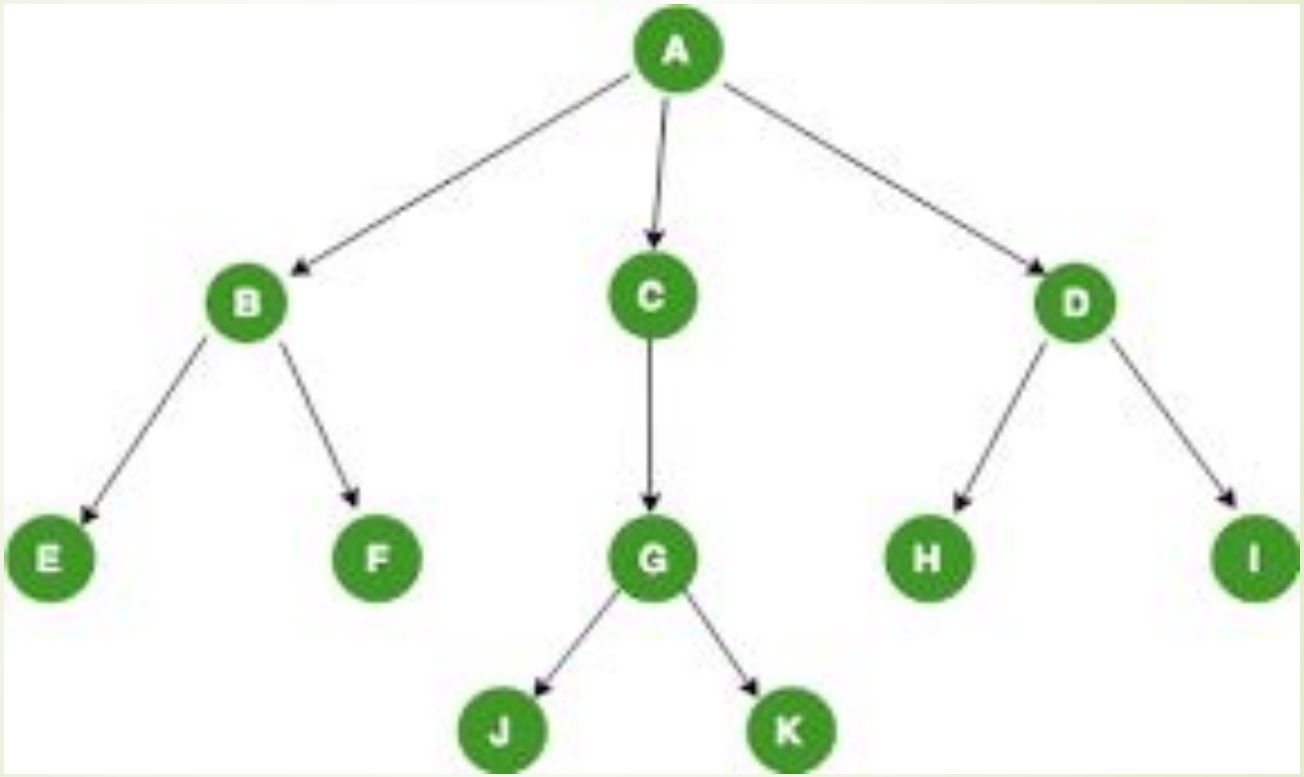
Department of Computer Science and Engineering

TREE

A **Tree:**

- is a nonlinear **data structure.**
- can be empty with no nodes or a **tree** is a **structure** consisting of one node called the root and zero or more subtrees.
- stores the data elements in a hierarchical manner.

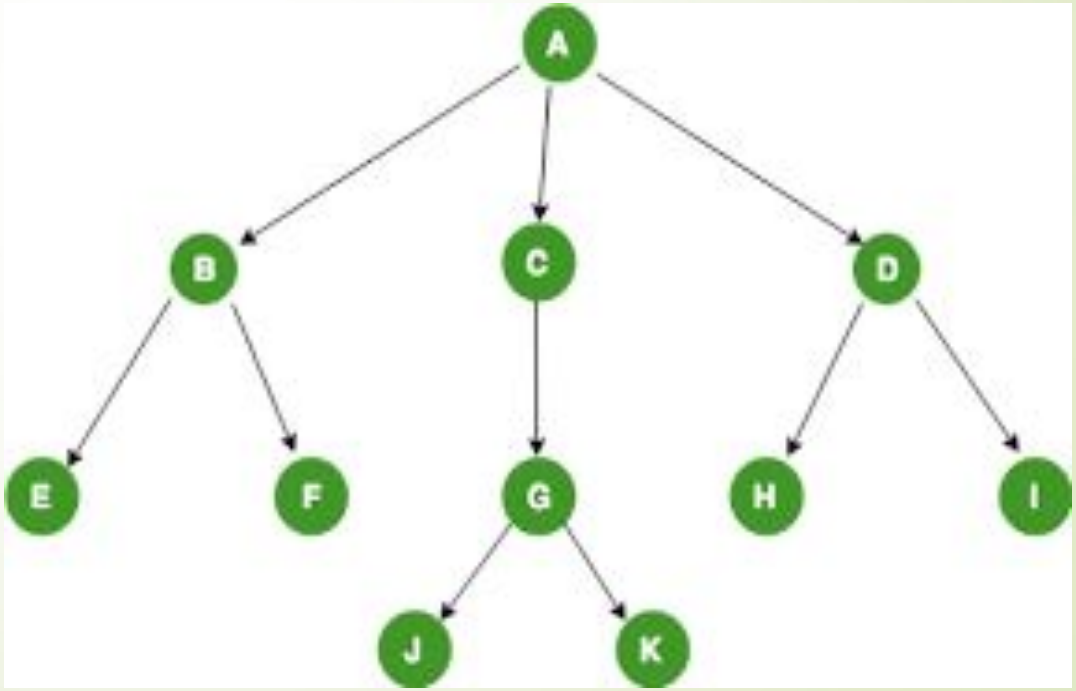
Basic Concept



Tree terminologies

Node·

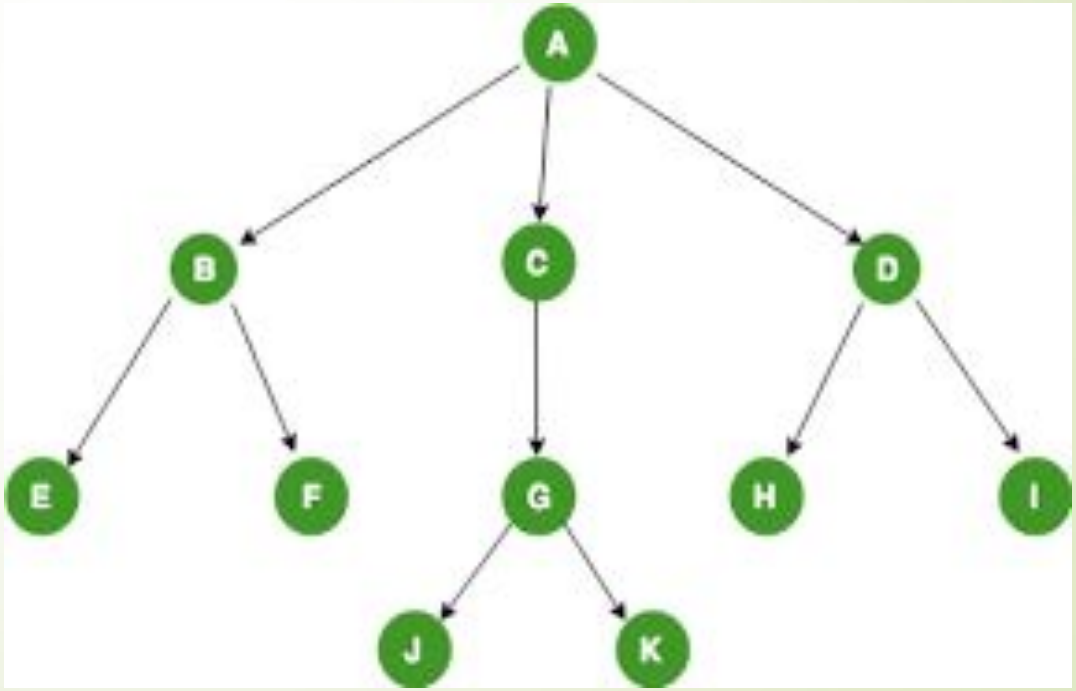
It is the data element of tree. Apart from storing a value it also specifies links to the other nodes



Tree terminologies

Root ·

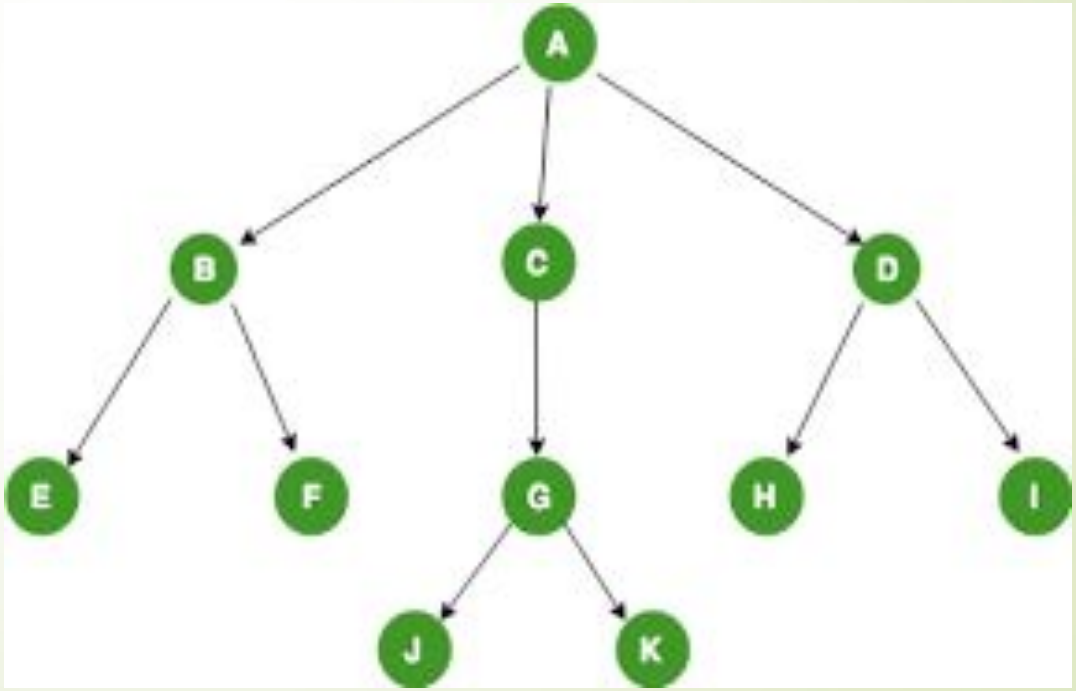
It is the top node in the tree



Tree terminologies

Parent·

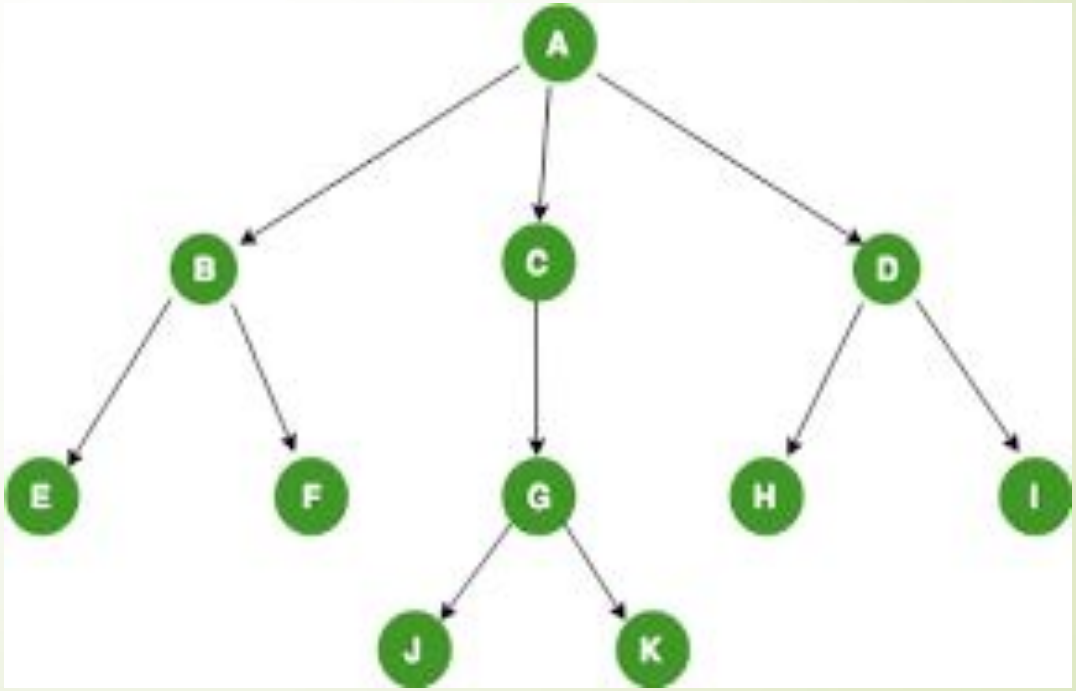
A node that has one or more child nodes



Tree terminologies

Child·

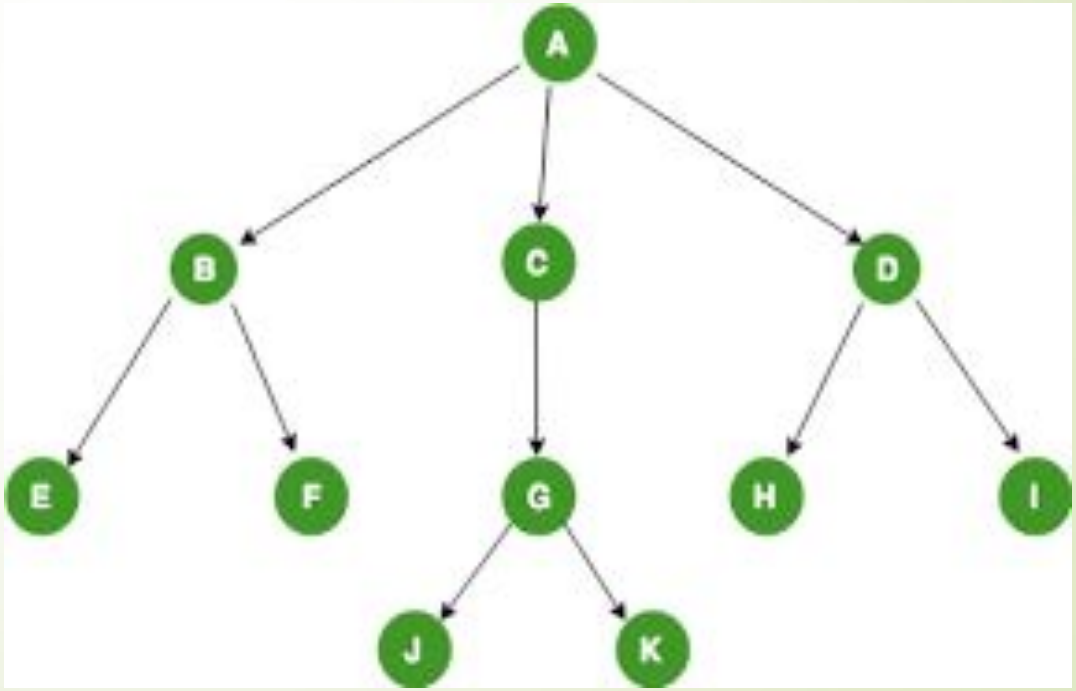
All nodes in a tree except the root node are child nodes of their immediate predecessor nodes



Tree terminologies

Leaf/ Terminal Node·

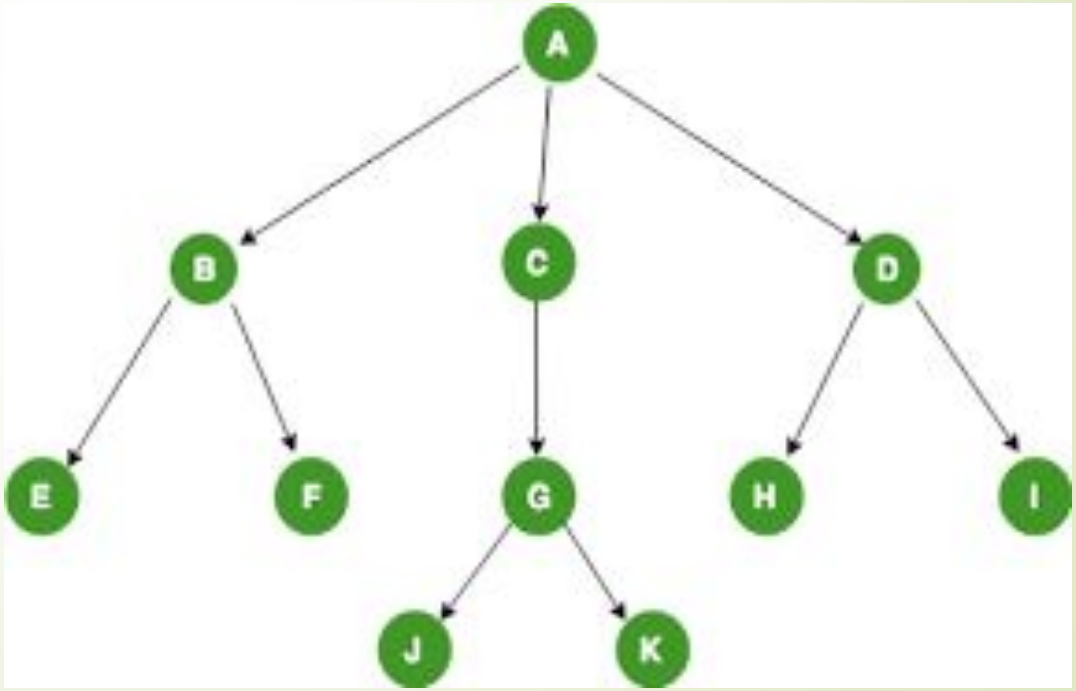
It is the terminal node that does not have any child nodes



Tree terminologies

Internal Node·

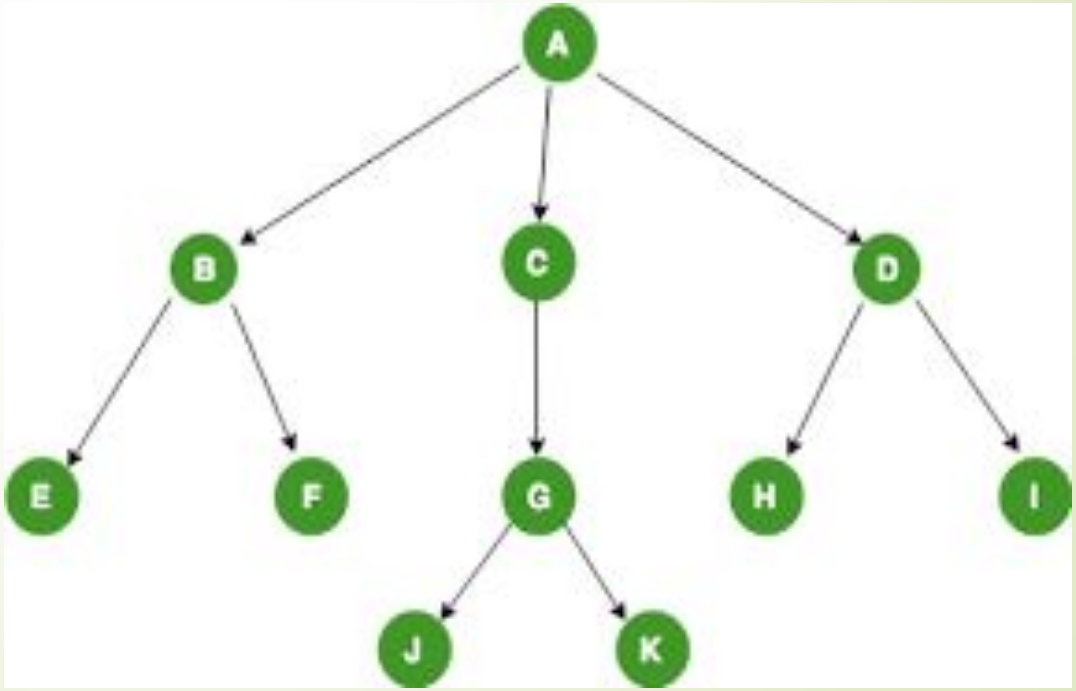
All nodes except root and leaf nodes are referred as internal nodes



Tree terminologies

Non Terminal Node·

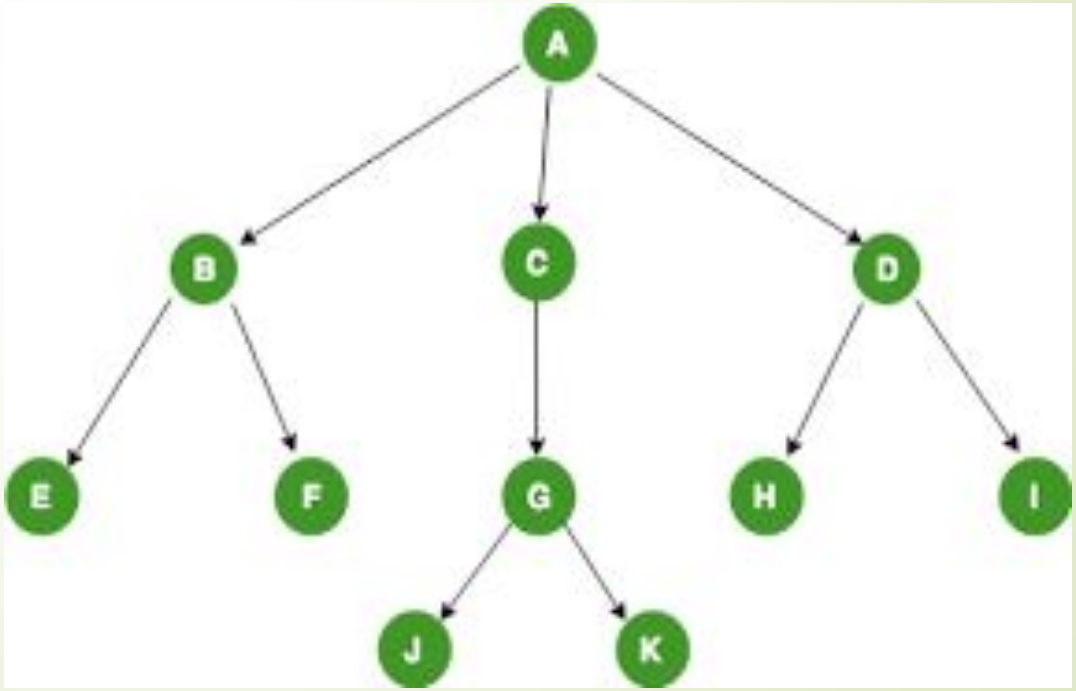
All nodes except leaf nodes are referred as Non Terminal nodes



Tree terminologies

Sibling·

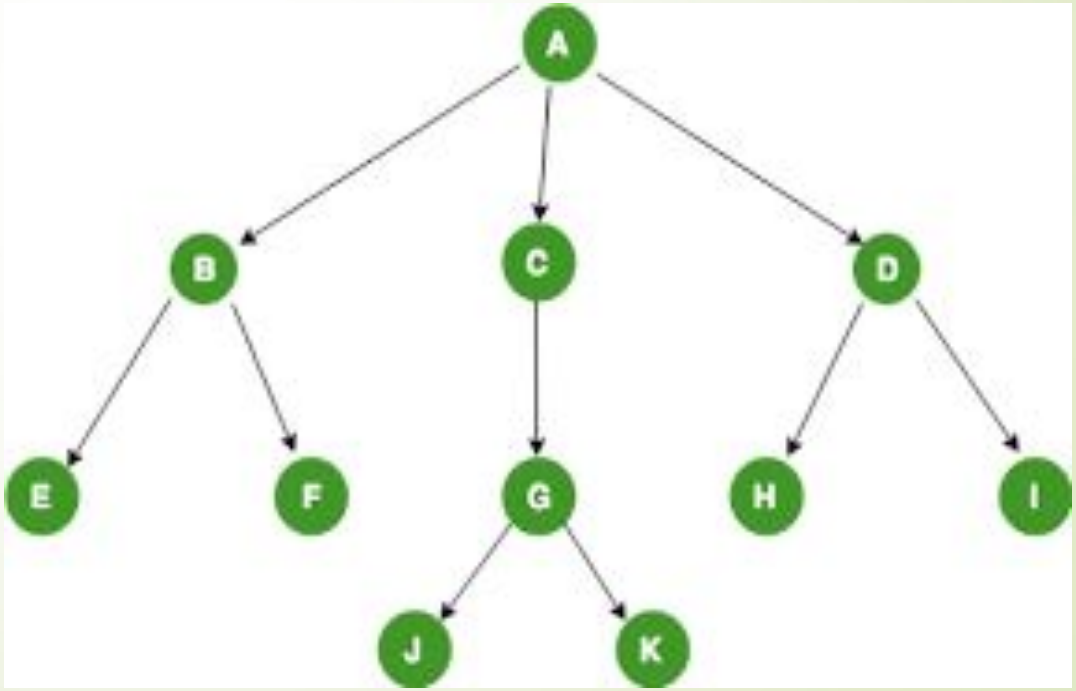
All child nodes of same parent are referred as Siblings



Tree terminologies

Degree·

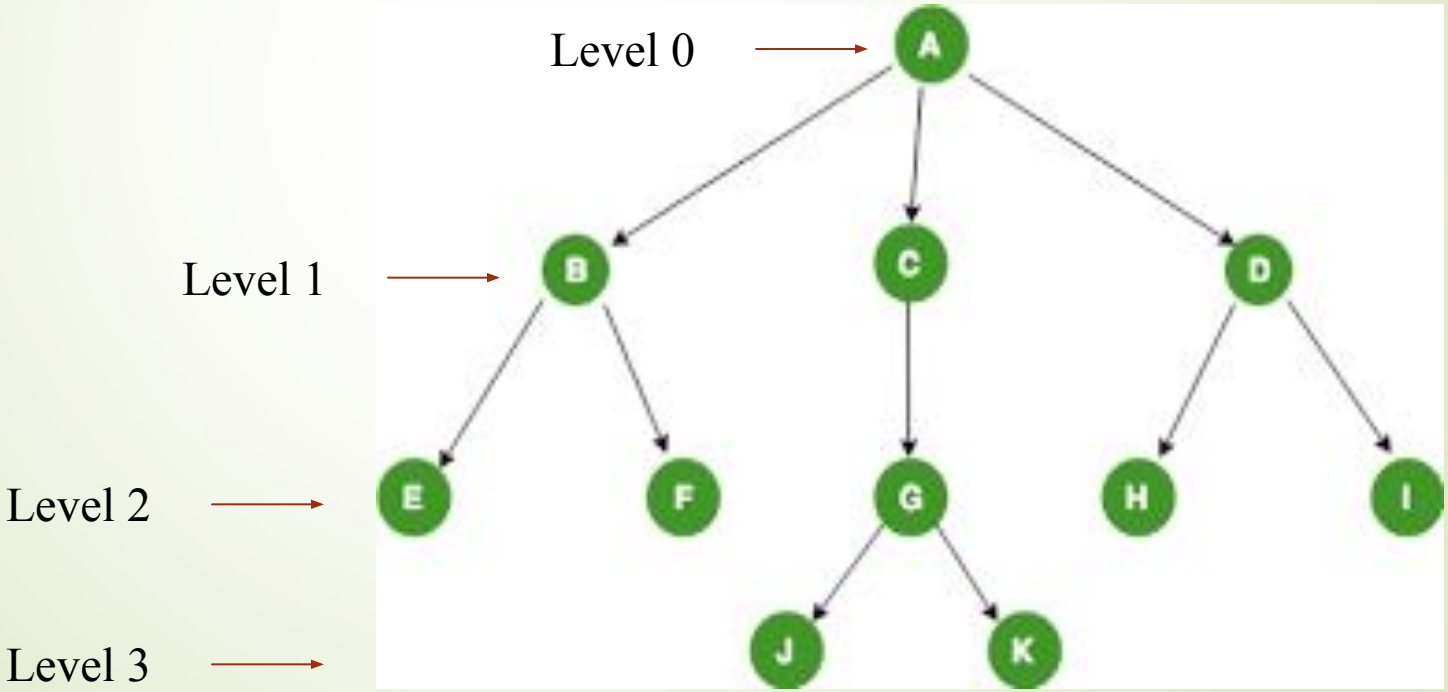
The degree of a node is the number of subtrees coming out of the node



Tree terminologies

Level·

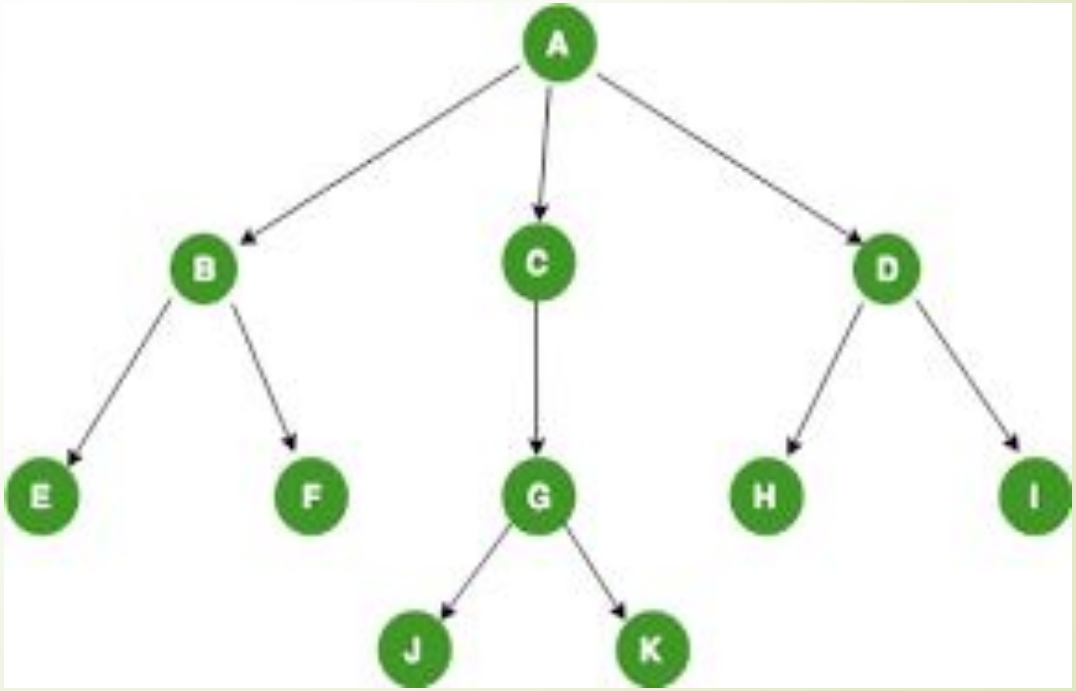
All the tree nodes are present at different levels. Root node is at level 0.



Tree terminologies

Height OR Depth·

It is maximum level of a node in the tree

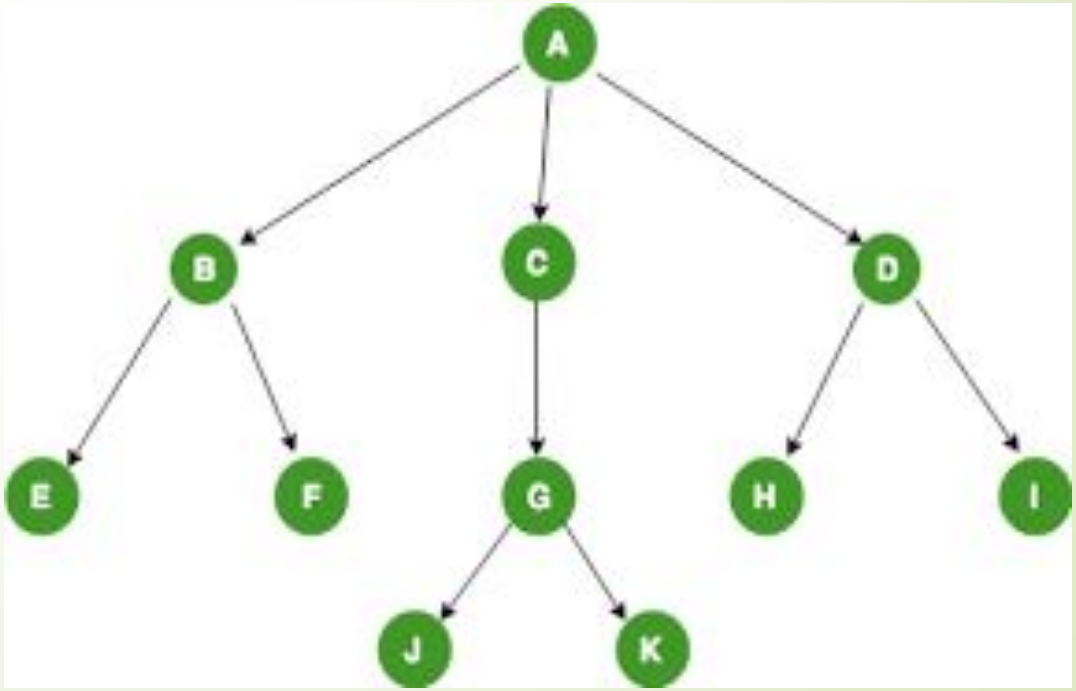


Height= 3

Tree terminologies

Path·

It is the sequence of nodes from source node to destination node

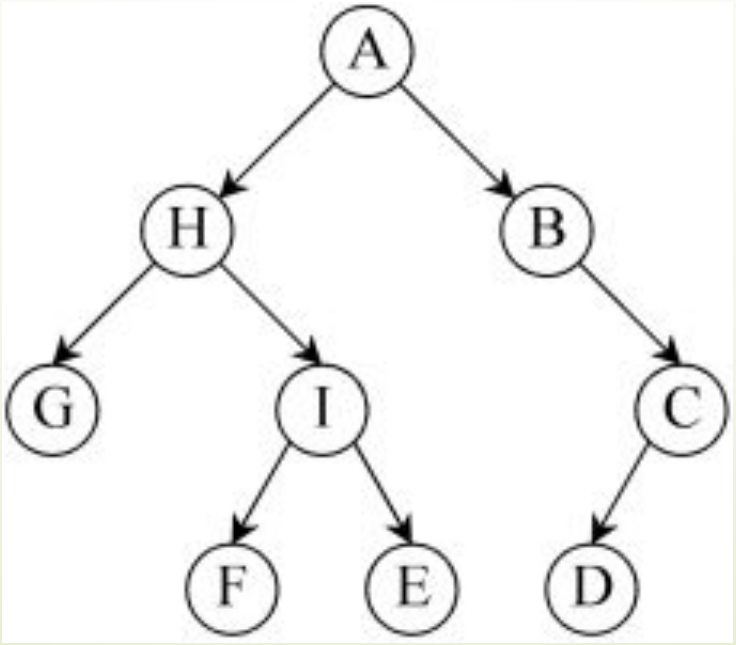


Height= 3

Binary Tree

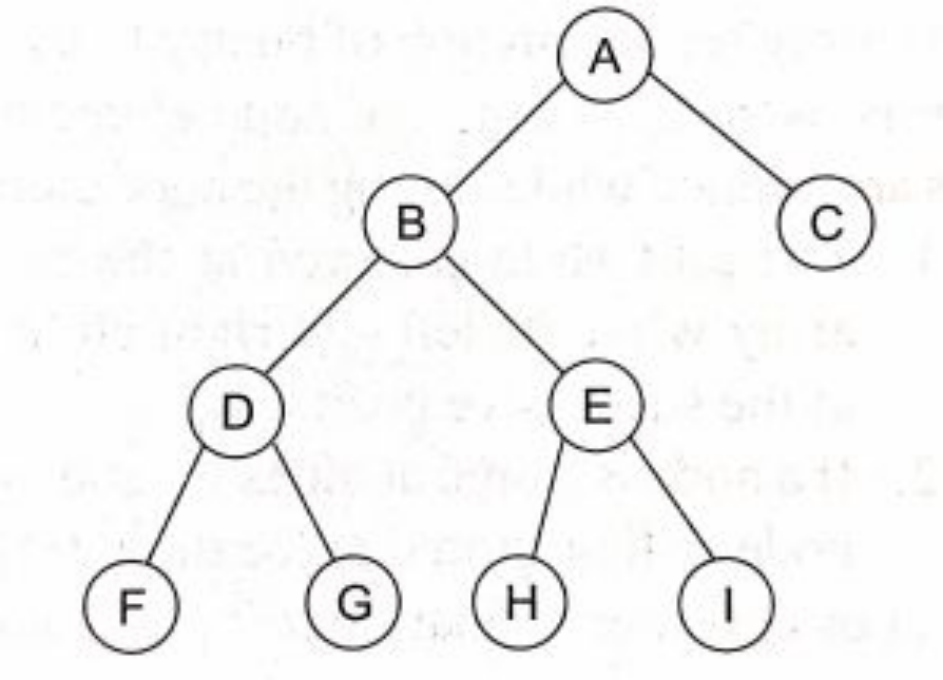
- A binary tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.

- Each node contains three components: Data element, Pointer to left subtree, Pointer to right subtree



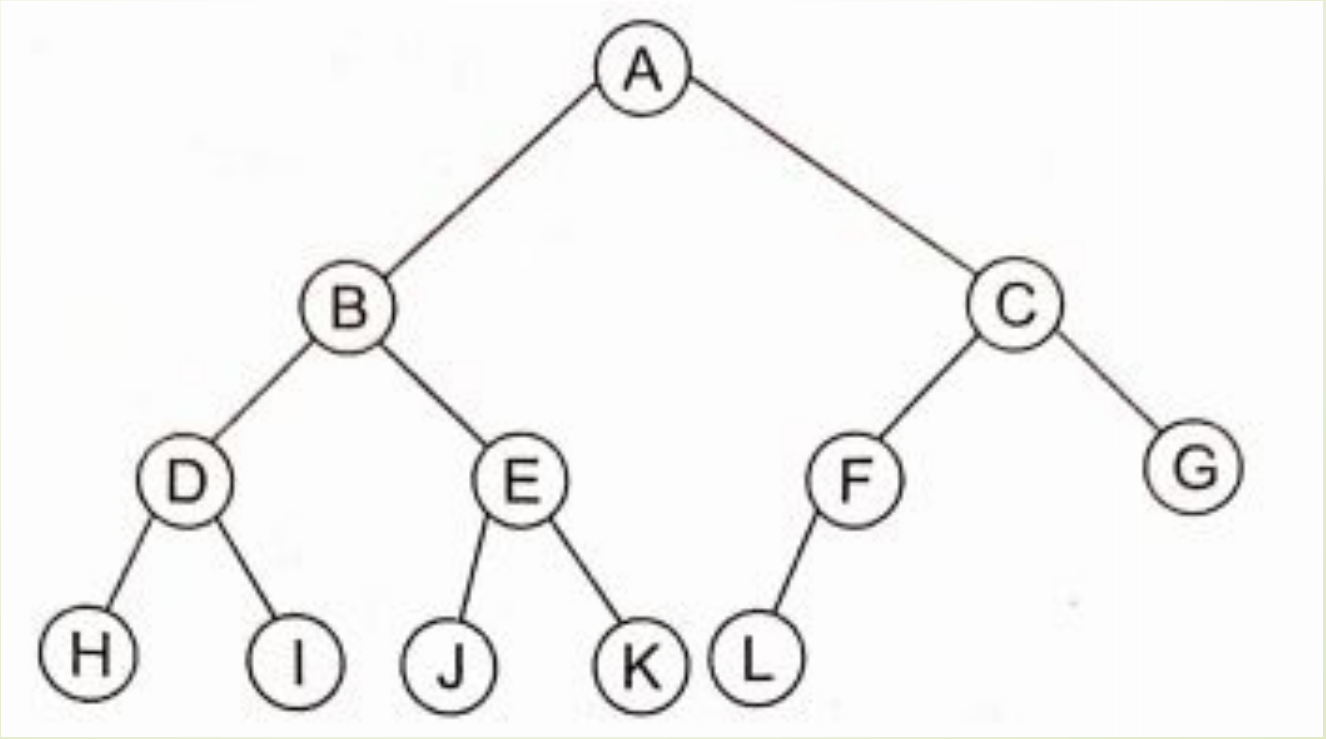
Strictly Binary Tree

A binary tree is called strictly binary if all its nodes except the leaf nodes contain two child nodes



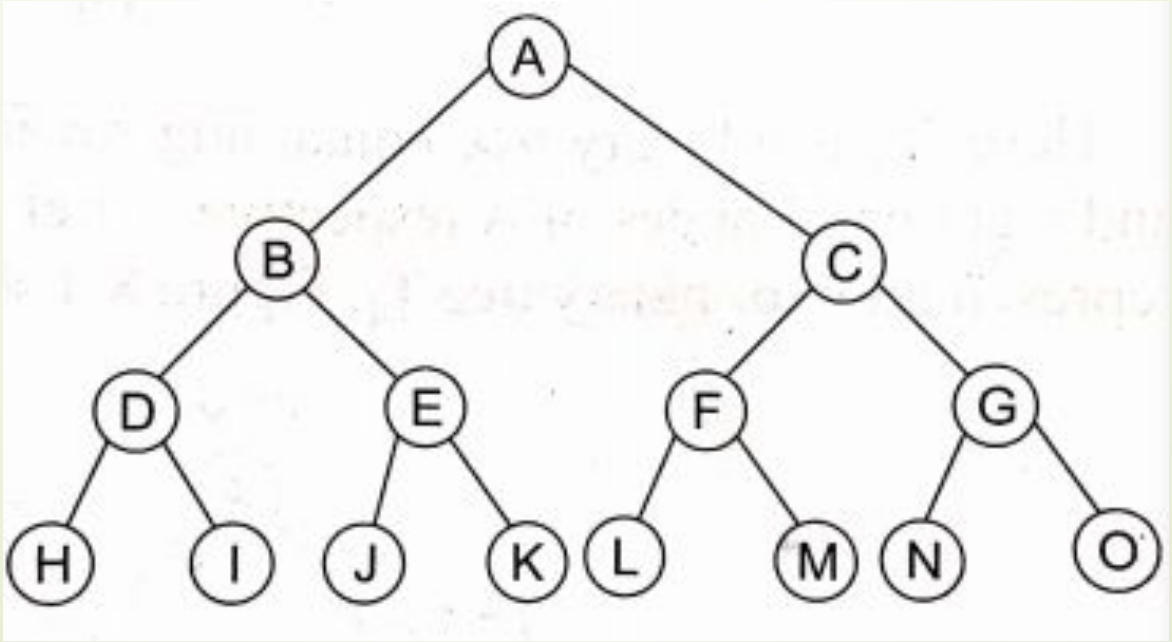
Complete Binary Tree

A binary tree of depth d is called complete binary tree if all its levels from 0 to d-1 contain maximum possible number of nodes and all the leaf nodes present at level d are placed towards the left side



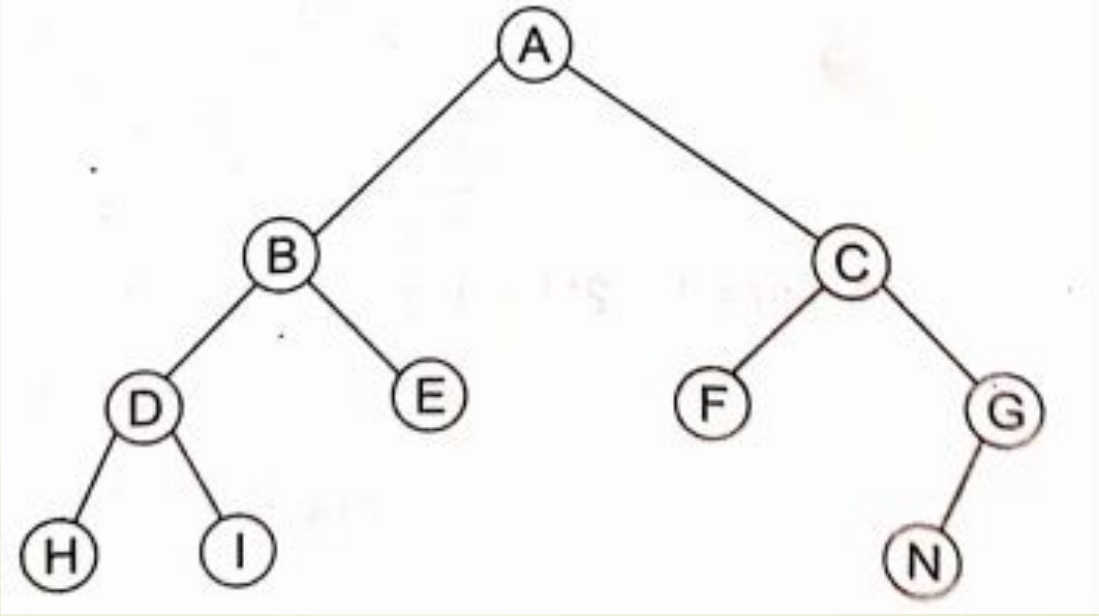
Perfect Binary Tree

A binary tree is called perfect binary tree if all its leaf nodes are at the lowest level (with Maximum Level number)and all the non-leaf nodes contain two child nodes.



Balanced Binary Tree

A binary tree is called balanced binary tree if the depths of the subtrees of all its nodes do not differ by more than 1



Binary Tree Traversal

A binary tree traversal means visiting every node of the tree only once.

Three Traversal methods are:

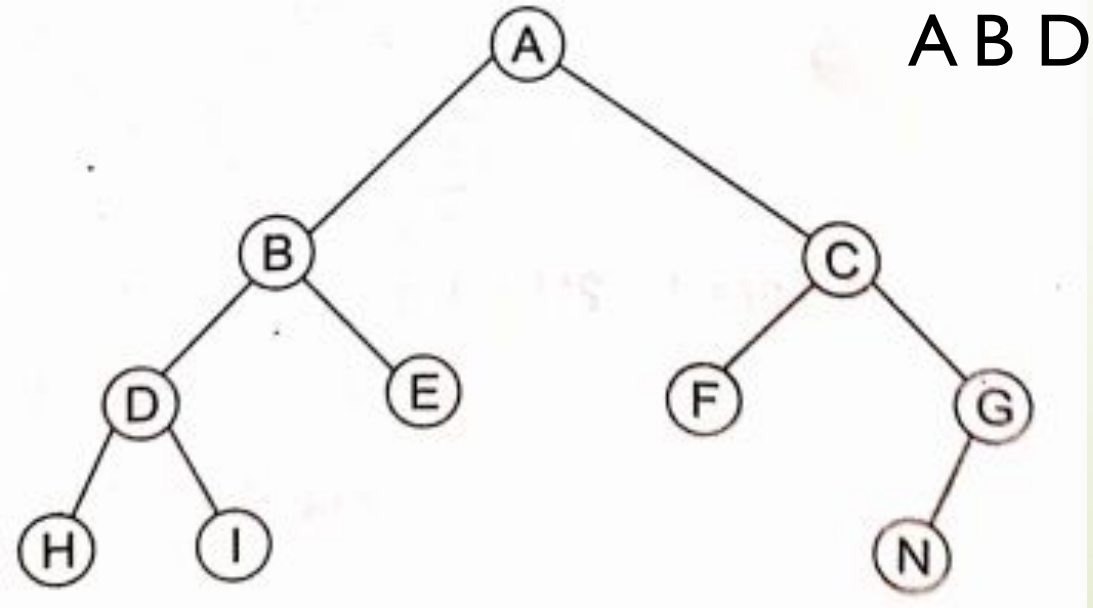
1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal

Preorder Traversal

Preorder Traversal:

- Visit and print the root node
- Traverse the left sub-tree
- Traverse the right sub-tree

ROOT - LEFT - RIGHT



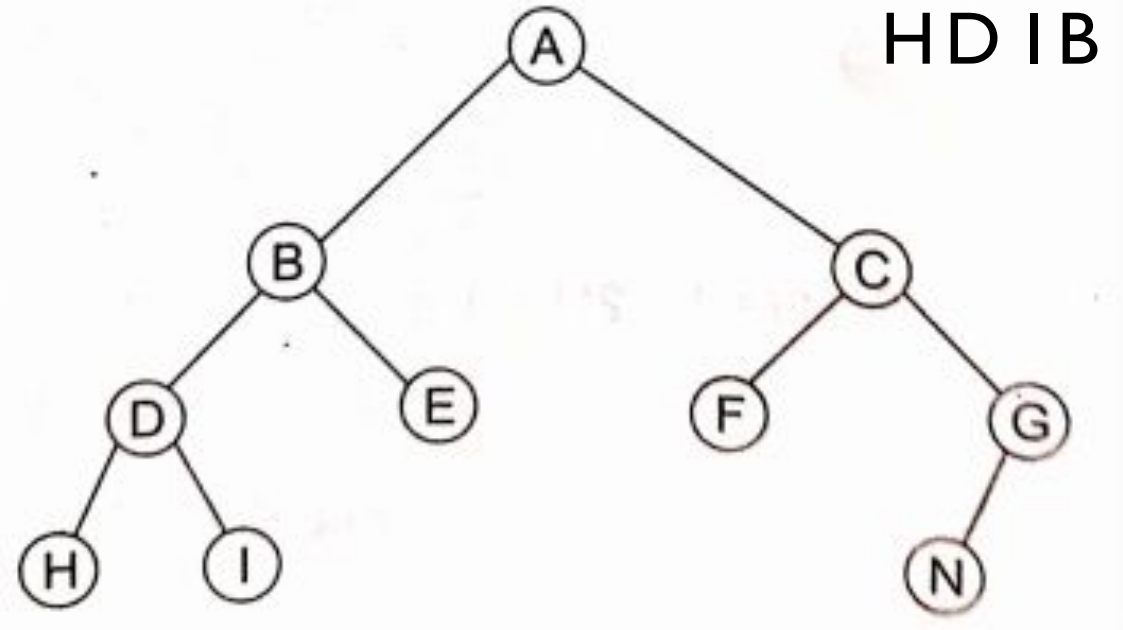
A B D H I E C F G N

Inorder Traversal

Inorder Traversal:

- Traverse the left sub-tree
- Visit and print the root node
- Traverse the right sub-tree

LEFT – ROOT - RIGHT



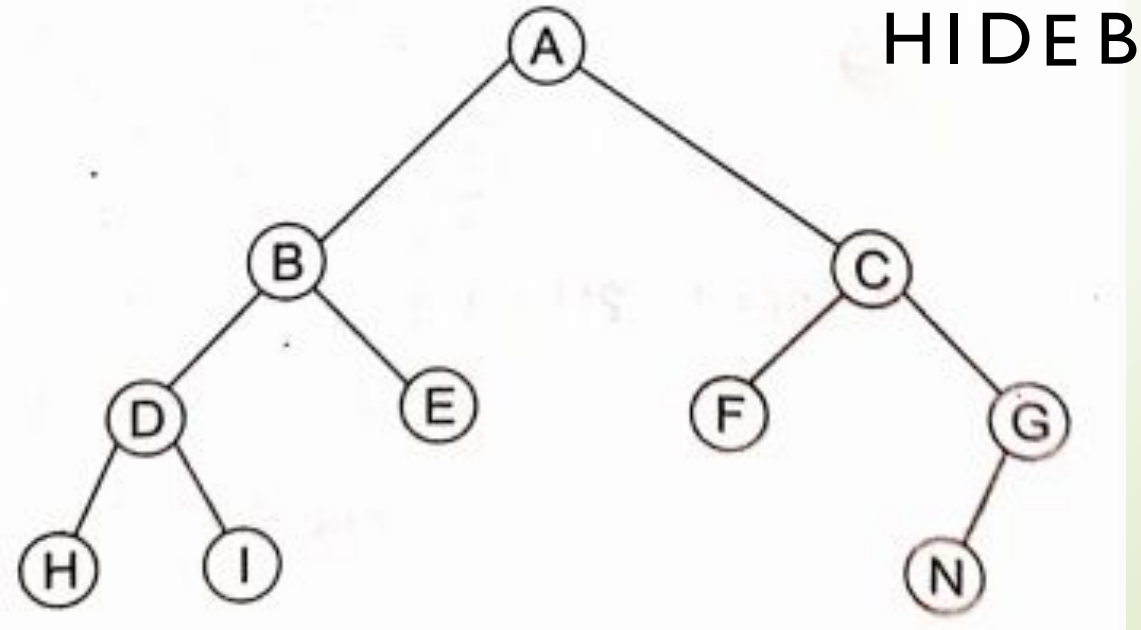
HD I B EA F CNG

Postorder Traversal

Postorder Traversal:

- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit and print the root node

LEFT - RIGHT - ROOT



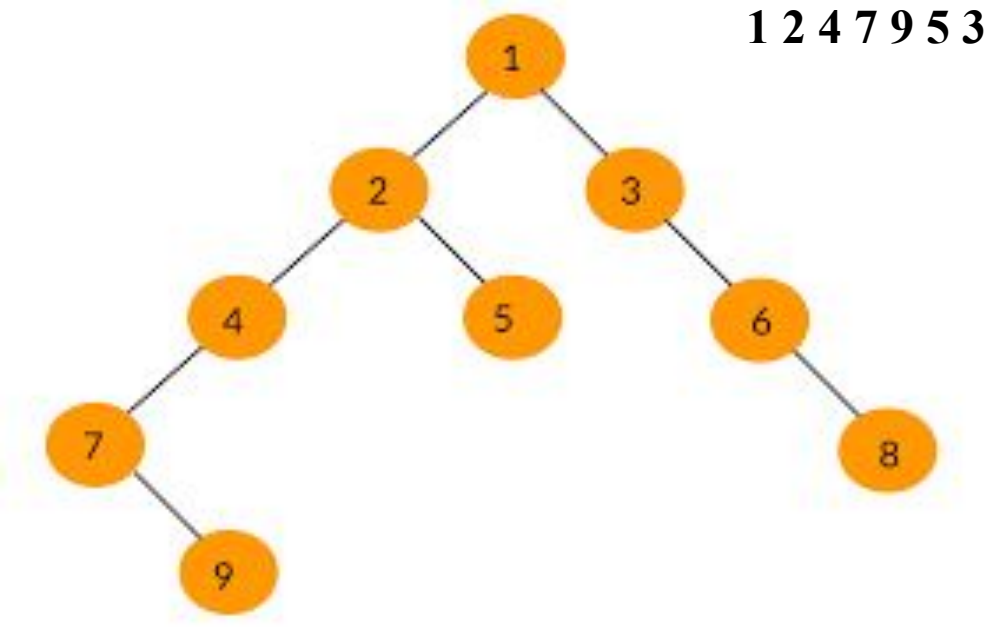
HIDEBFNGCA

Preorder Traversal

Preorder Traversal:

- Visit and print the root node
- Traverse the left sub-tree
- Traverse the right sub-tree

ROOT - LEFT - RIGHT



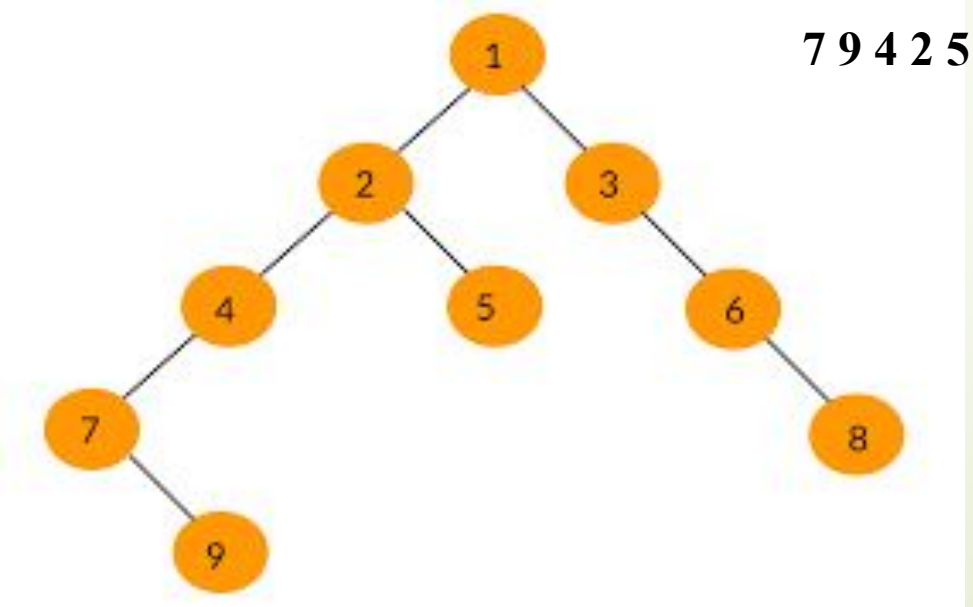
1 2 4 7 9 5 3 6 8

Inorder Traversal

Inorder Traversal:

- Traverse the left sub-tree
- Visit and print the root node
- Traverse the right sub-tree

LEFT – ROOT - RIGHT



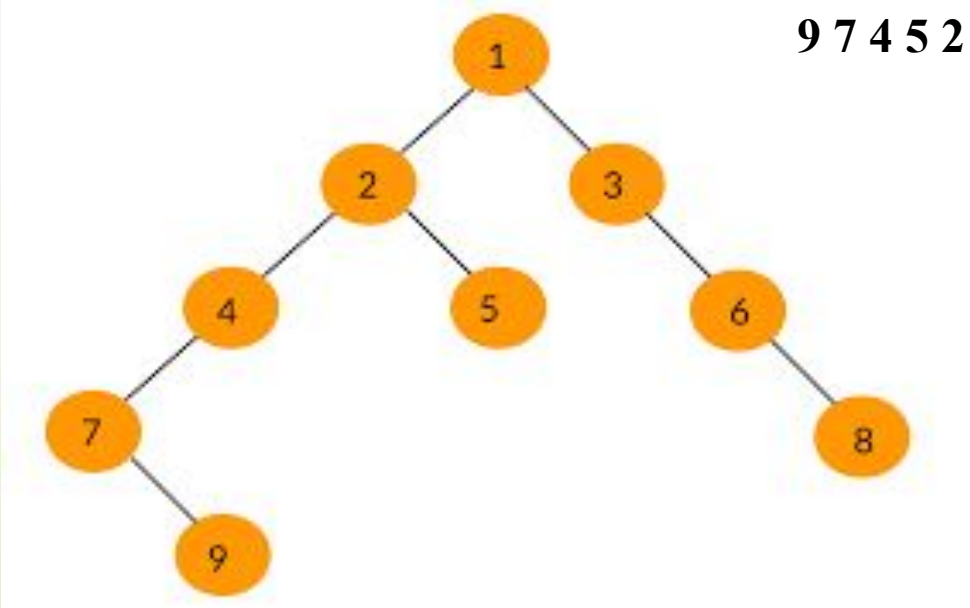
7 9 4 2 5 1 3 6 8

Postorder Traversal

Postorder Traversal:

- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit and print the root node

LEFT - RIGHT - ROOT



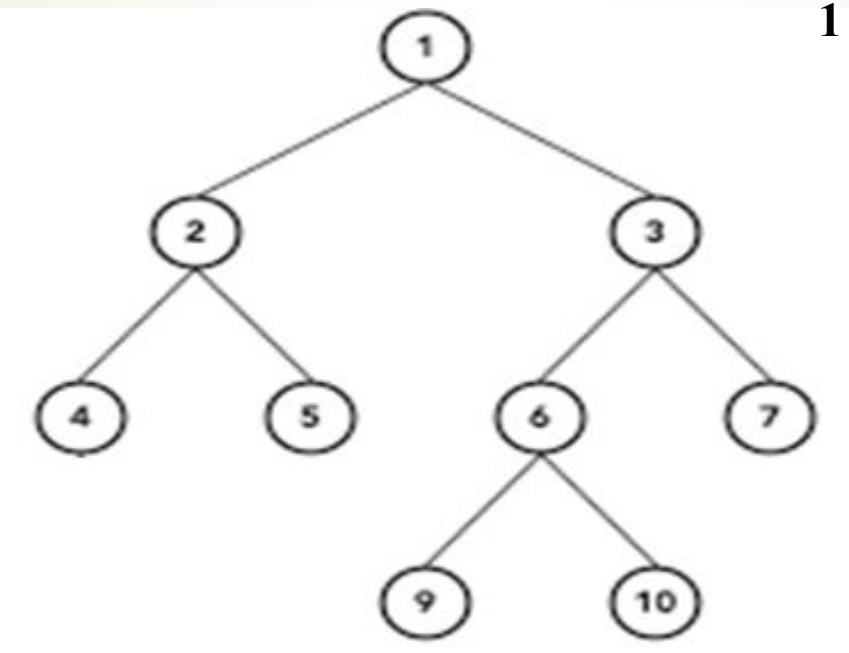
9 7 4 5 2 8 6 3 1

Preorder Traversal

Preorder Traversal:

- Visit and print the root node
- Traverse the left sub-tree
- Traverse the right sub-tree

ROOT - LEFT - RIGHT



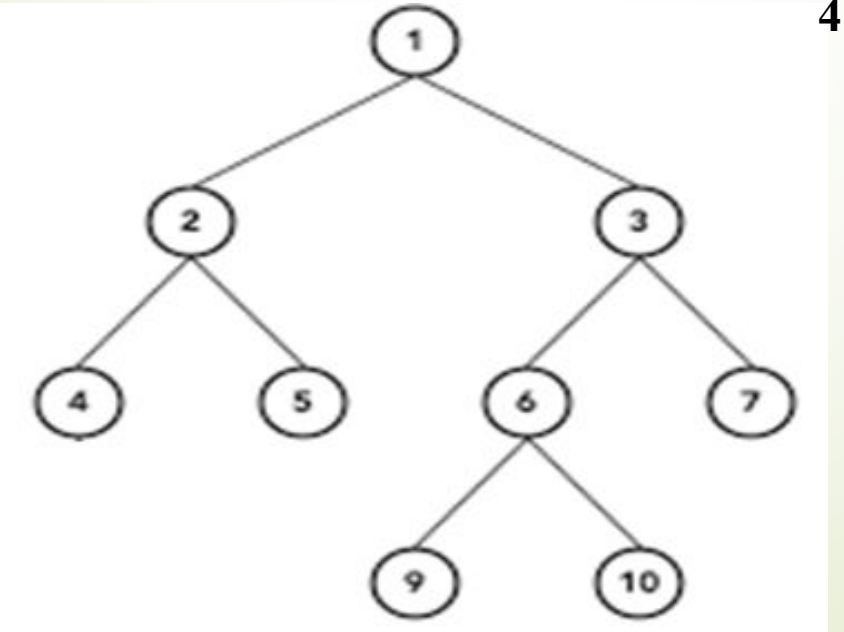
1 2 4 5 3 6 9 10 7

Inorder Traversal

Inorder Traversal:

- Traverse the left sub-tree
- Visit and print the root node
- Traverse the right sub-tree

LEFT – ROOT - RIGHT



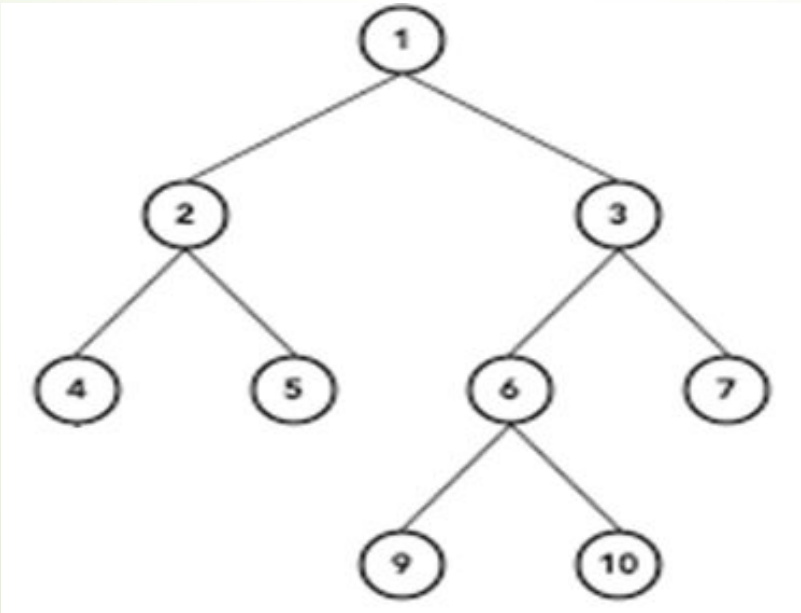
4 2 5 1 9 6 10 3 7

Postorder Traversal

Postorder Traversal:

- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit and print the root node

LEFT - RIGHT - ROOT



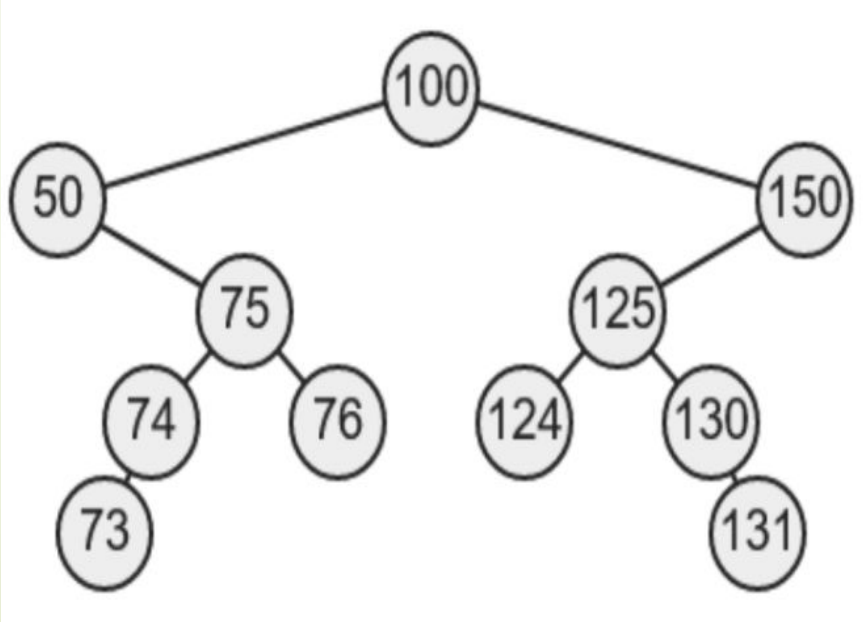
4 5 2 9 10 6 7 3 1

Preorder Traversal

Preorder Traversal:

- Visit and print the root node
- Traverse the left sub-tree
- Traverse the right sub-tree

ROOT - LEFT - RIGHT



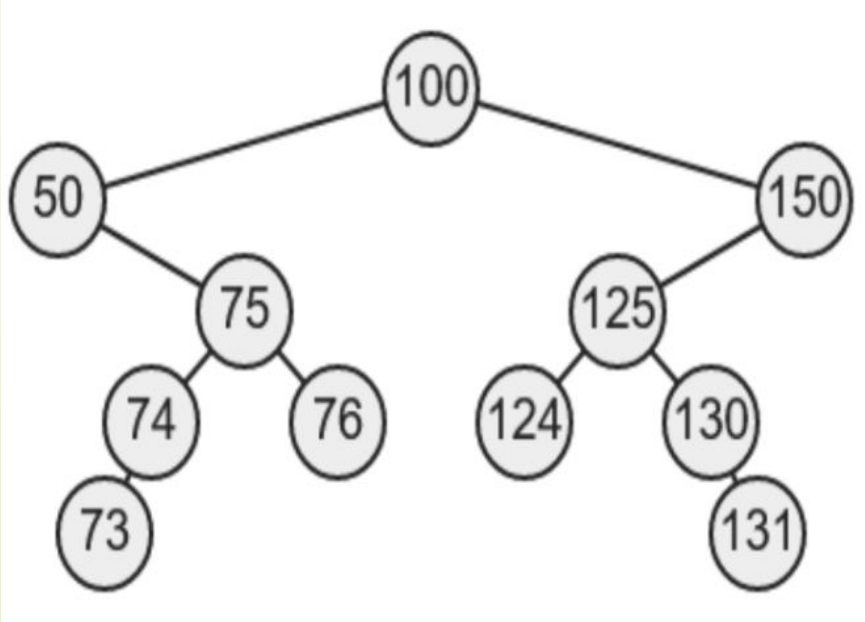
100, 50, 75, 74, 73, 76, 150, 125, 124, 130, 131

Inorder Traversal

Inorder Traversal:

- Traverse the left sub-tree
- Visit and print the root node
- Traverse the right sub-tree

LEFT – ROOT - RIGHT



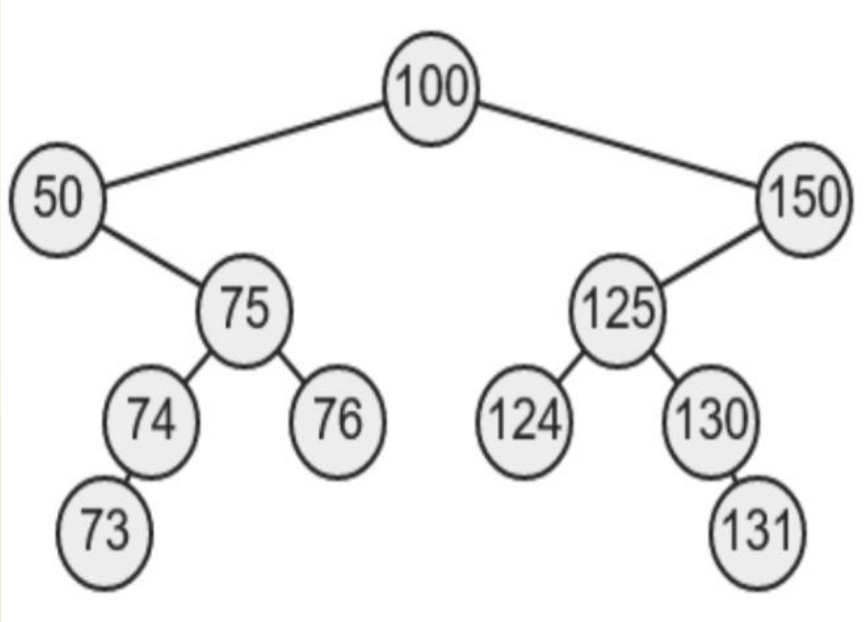
50, 73, 74, 75, 76, 100, 124, 125, 130, 131, 150

Postorder Traversal

Postorder Traversal:

- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit and print the root node

LEFT - RIGHT - ROOT



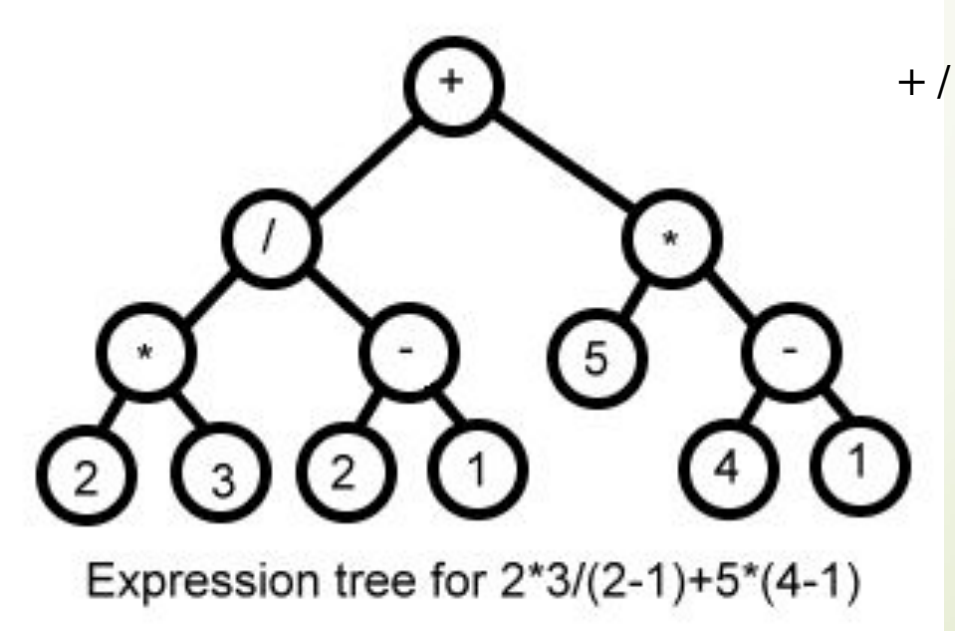
73, 74, 76, 75, 50, 124, 131,130, 125, 150, 100

Preorder Traversal

Preorder Traversal:

- Visit and print the root node
- Traverse the left sub-tree
- Traverse the right sub-tree

ROOT - LEFT - RIGHT



Expression tree for 2*3/(2-1)+5*(4-1)

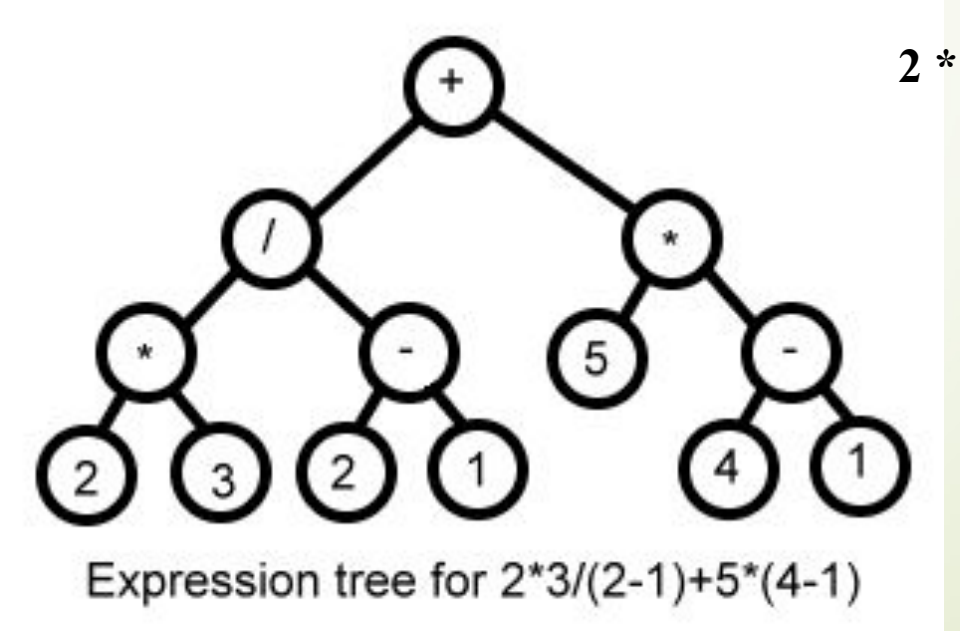
+ / * 2 3 – 2 1 * 5 – 4 1

Inorder Traversal

Inorder Traversal:

- Traverse the left sub-tree
- Visit and print the root node
- Traverse the right sub-tree

LEFT – ROOT - RIGHT



Expression tree for 2*3/(2-1)+5*(4-1)

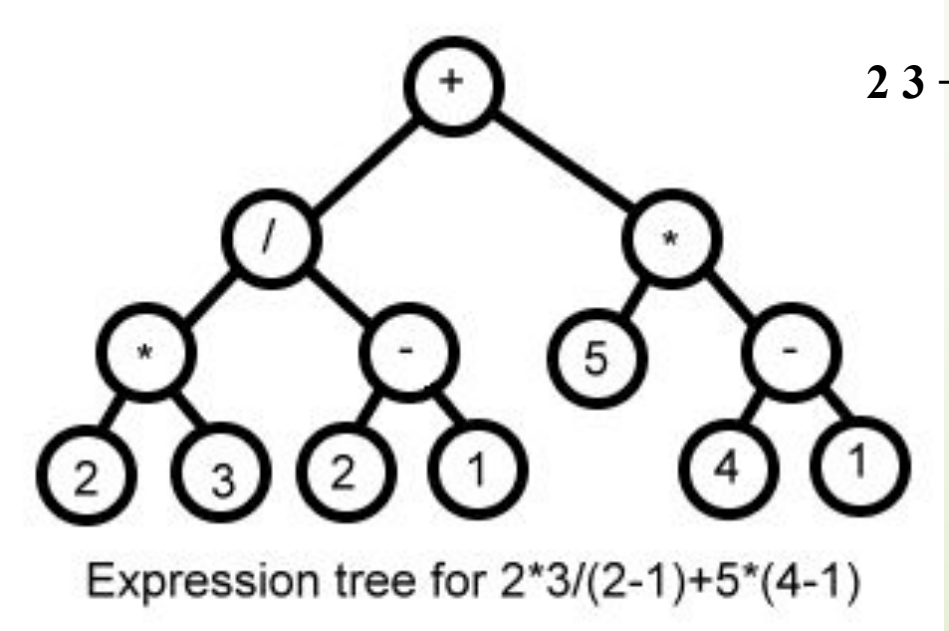
2 * 3 / 2 – 1 + 5 * 4 - 1

Postorder Traversal

Postorder Traversal:

- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit and print the root node

LEFT - RIGHT - ROOT



Expression tree for 2*3/(2-1)+5*(4-1)

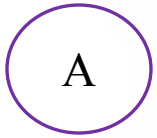
2 3 + 2 1 - / 5 4 1 - * +

Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G

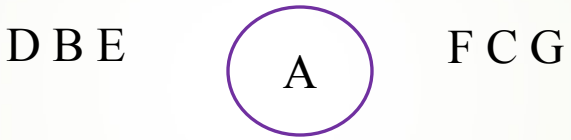


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G

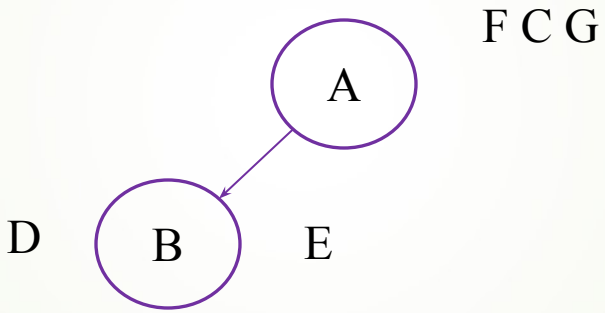


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G

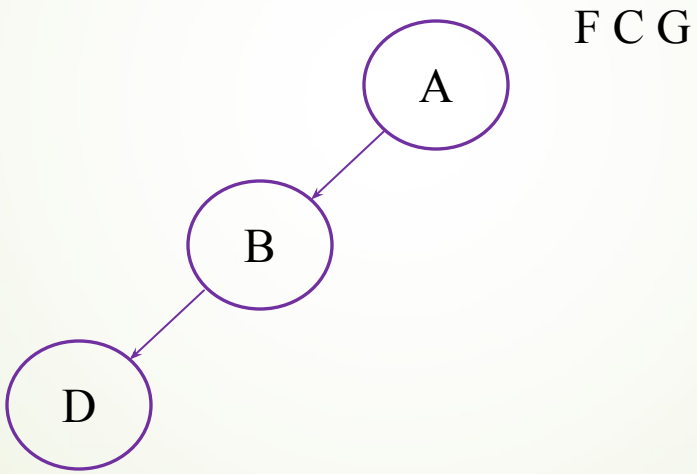


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G

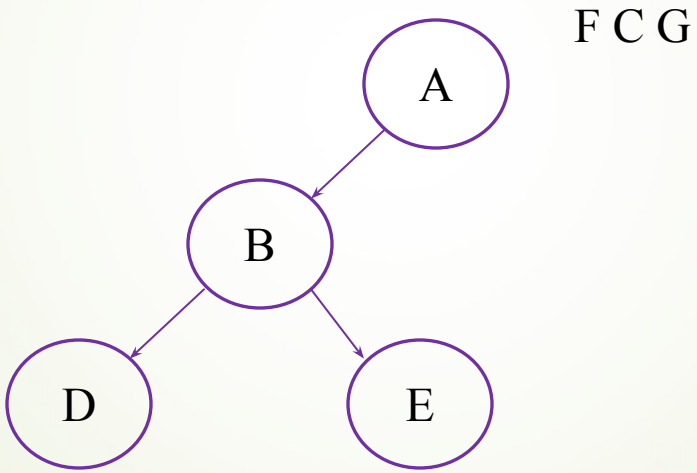


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G



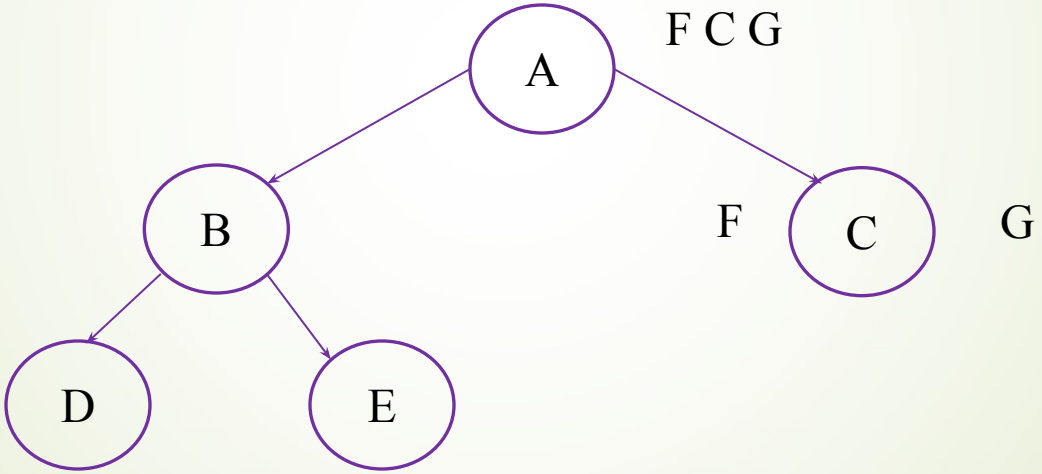
F C G

Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G

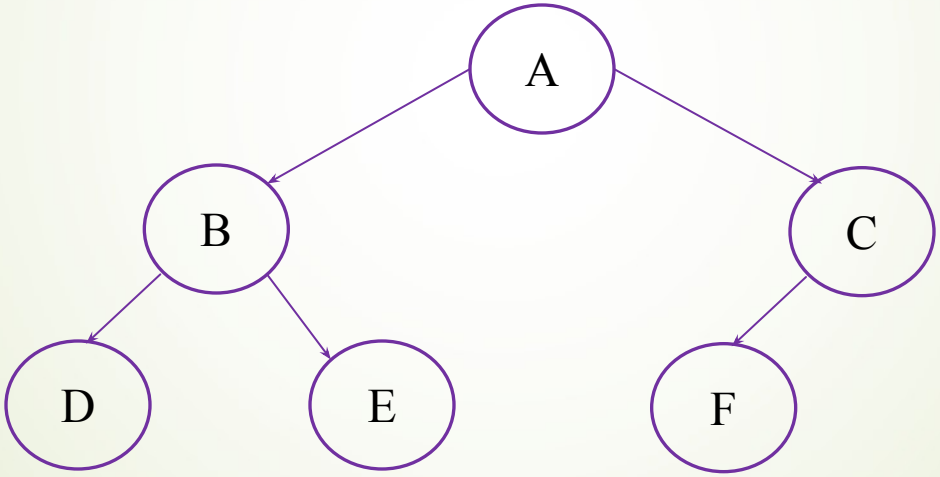


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G

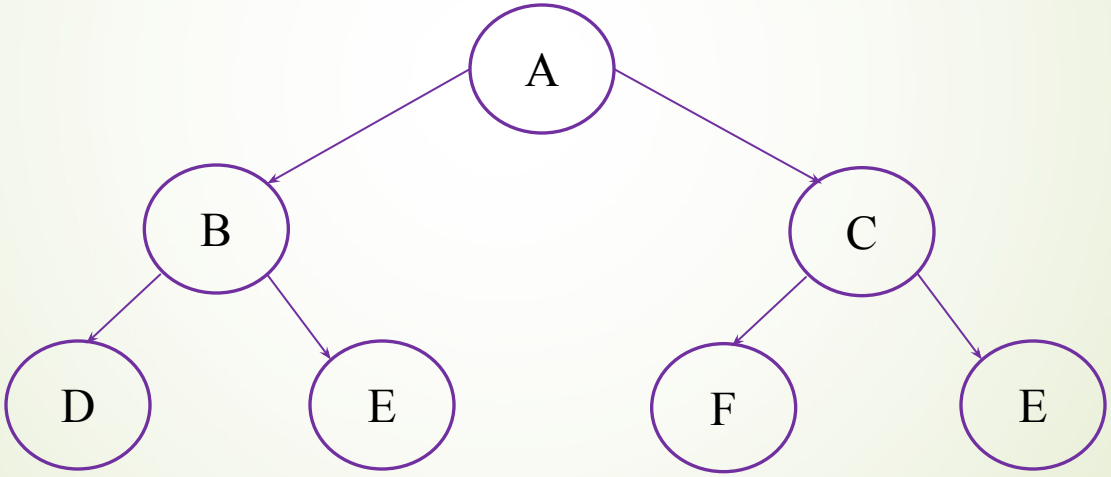


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: D B E A F C G

- Pre–order Traversal: A B D E C F G

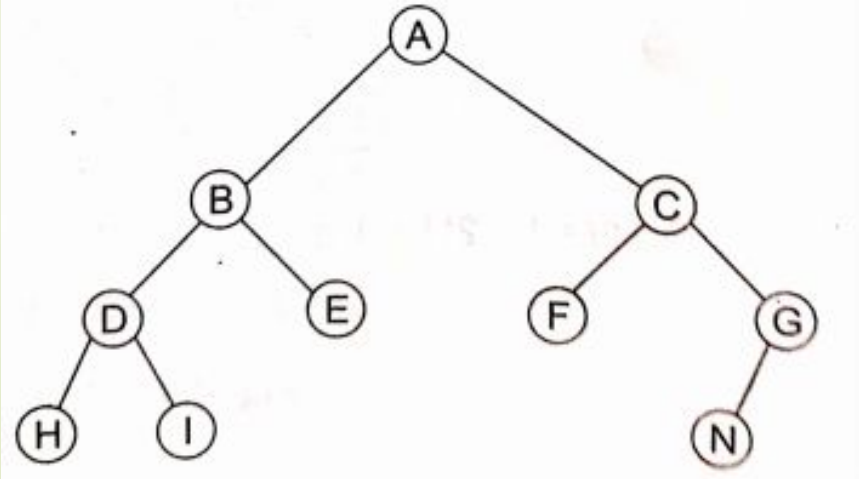


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: H D I B E A F C N G

- Pre–order Traversal: A B D H I E C F G N



Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal (L-R-RT):D H B E I A F C K J L G

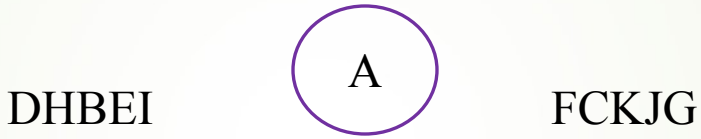
- Post–order Traversal(L-RT-R): H D I E B F K L J G C A

Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal:D H B E I A F C K J L G

- Post–order Traversal: H D I E B F K L J G C A

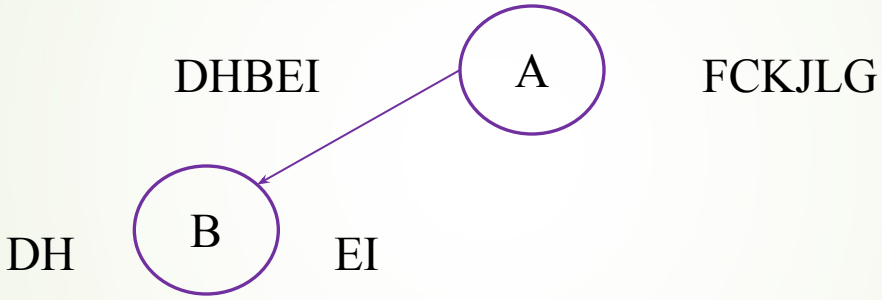


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal:D H B E I A F C K J L G

- Post–order Traversal: H D I E B F K L J G C A

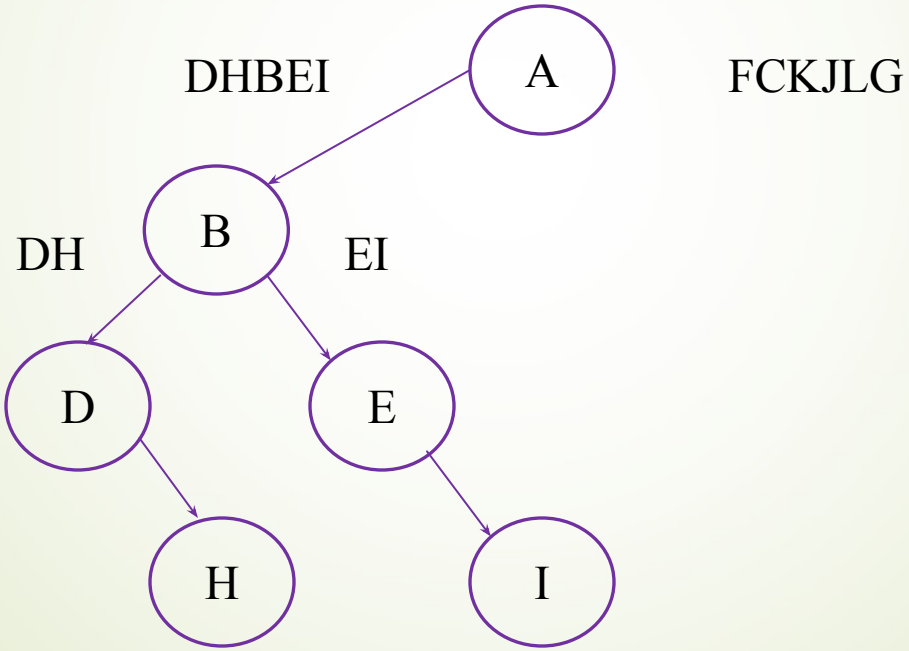


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal:D H B E I A F C K J L G

- Post–order Traversal: H D I E B F K L J G C A

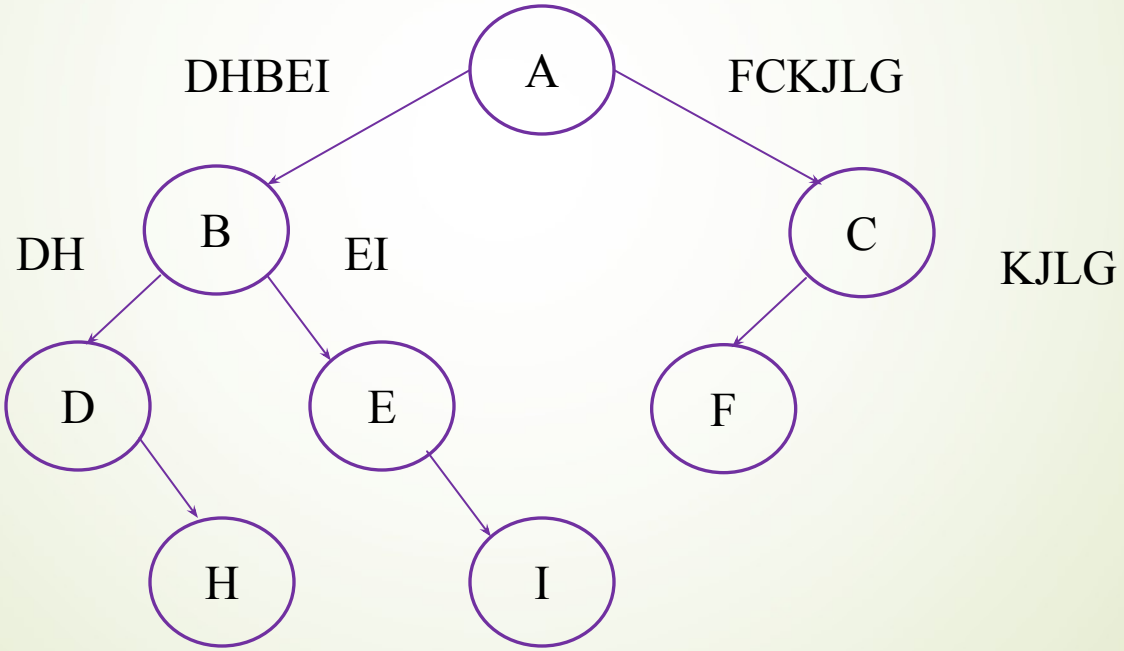


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal:D H B E I A F C K J L G

- Post–order Traversal: H D I E B F K L J G C A

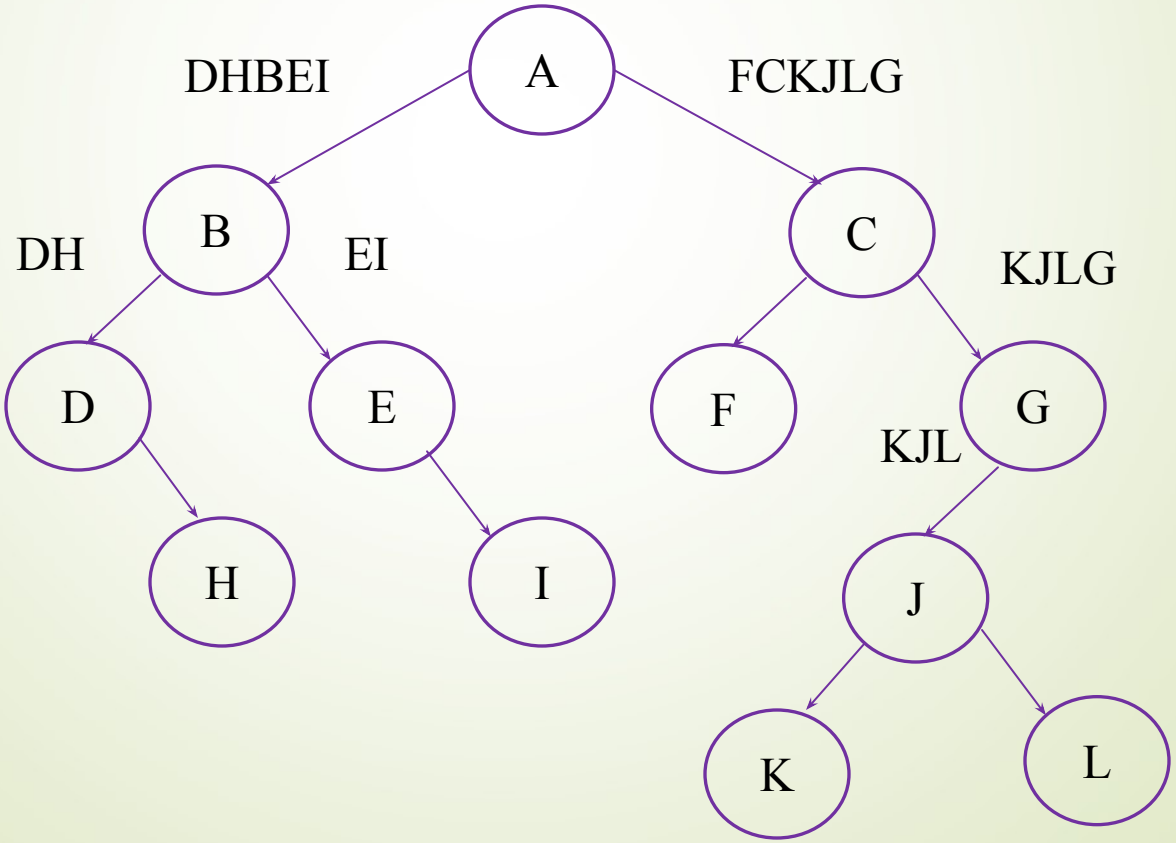


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal:D H B E I A F C K J L G

- Post–order Traversal: H D I E B F K L J G C A

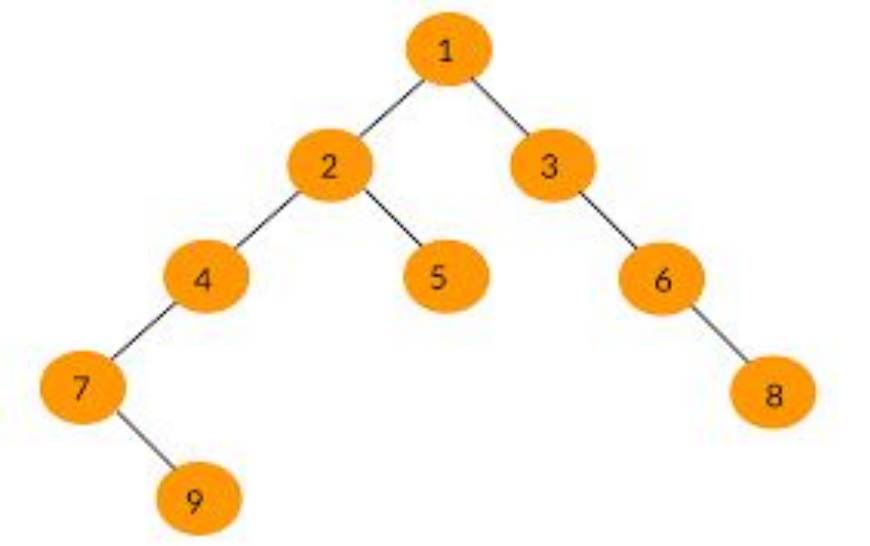


Constructing a Tree

Construct a binary tree with the given traversal techniques

- In–order Traversal: **7 9 4 2 5 1 3 6 8**

- Post–order Traversal: **9 7 4 5 2 8 6 3 1**



Constructing a Tree

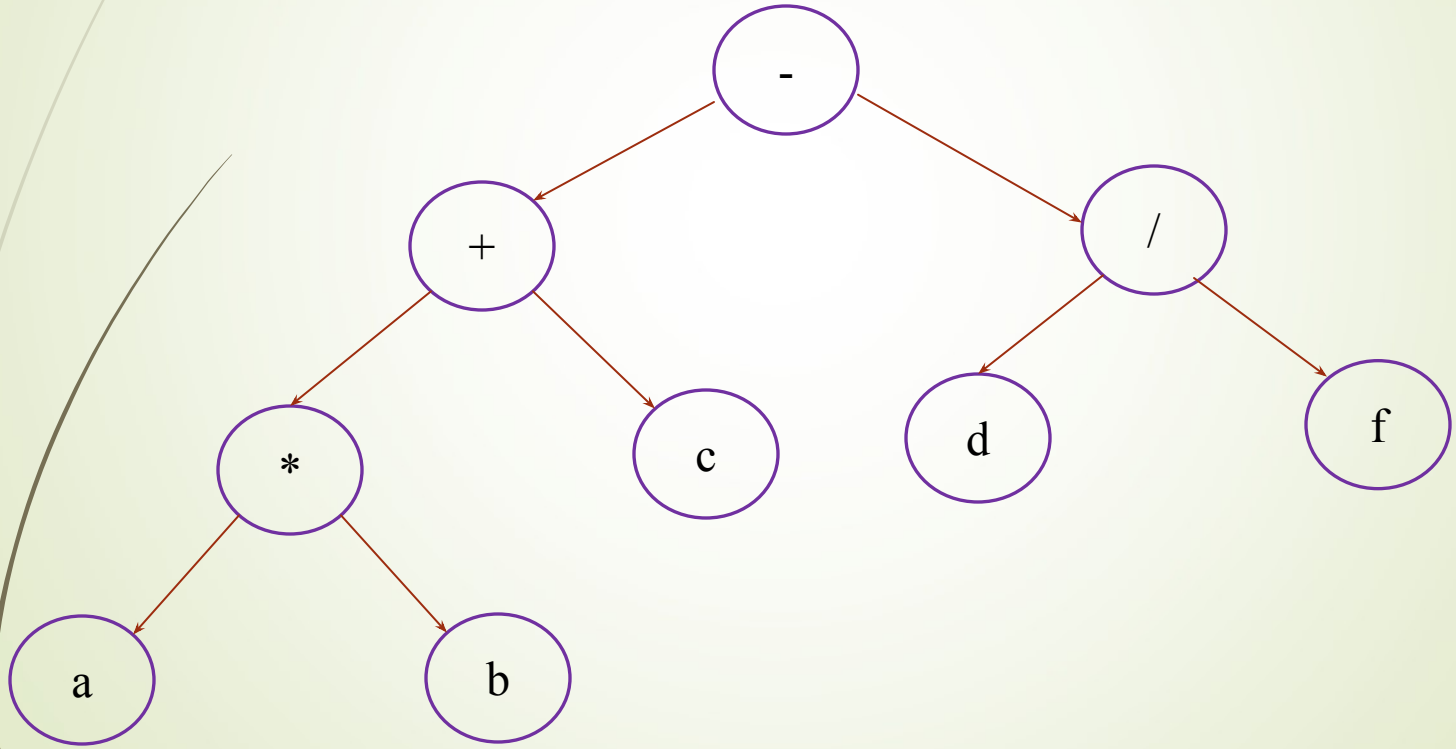
Building a Binary Expression tree from a Prefix expression

- Scan Prefix expression from Left to Right
  - 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 fashion

Constructing a Tree

- Given the Prefix Expression Construct a Expression Tree

  Prefix Expression: -* + a b c / d f



Constructing a Tree

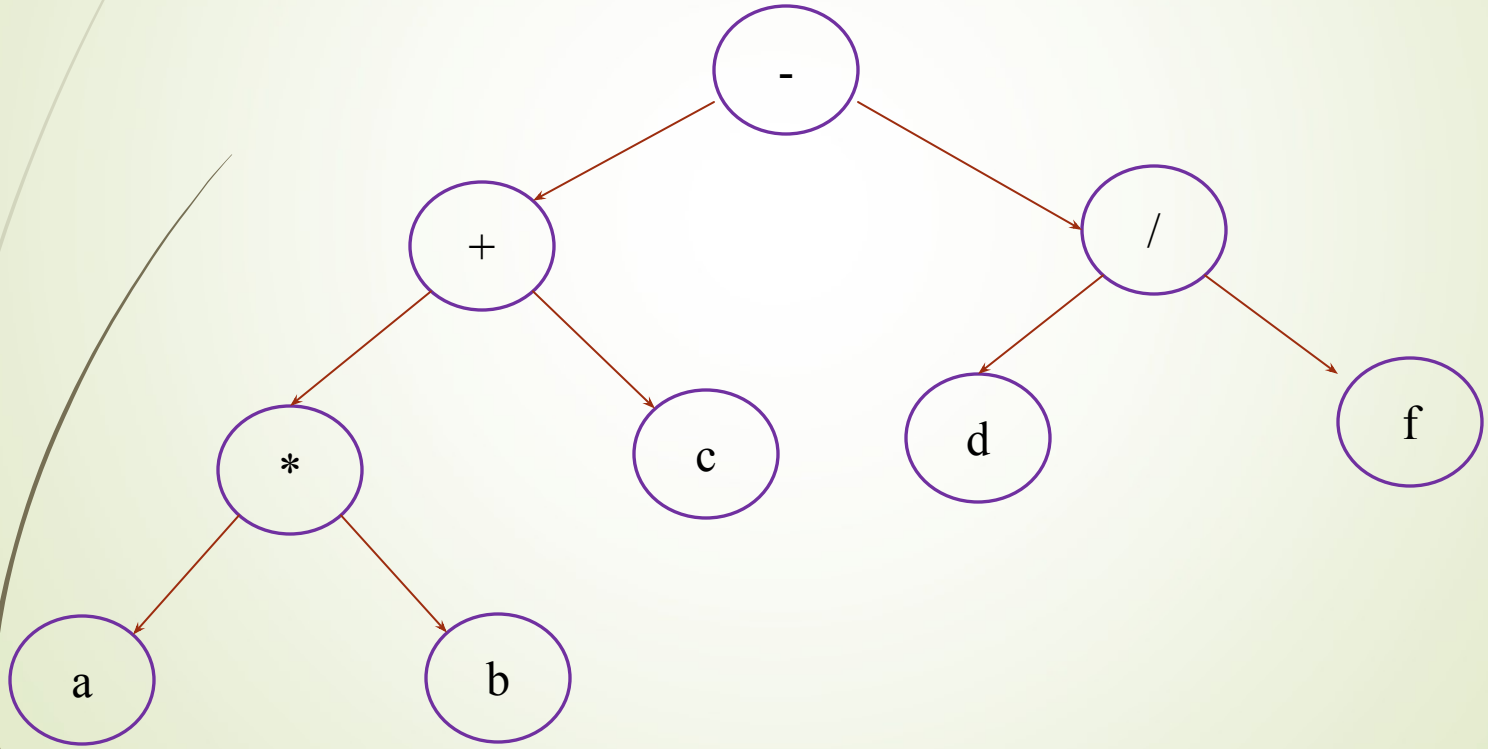
Building a Binary Expression tree from a Postfix expression

- Scan Postfix expression from Right to Left
  - Insert new nodes each time moving to the right until an operand has been inserted
  - Backtrack to the last operator, and put the next node to its left
  - Continue in the same fashion

Constructing a Tree

- Given the Postfix Expression Construct a Expression Tree

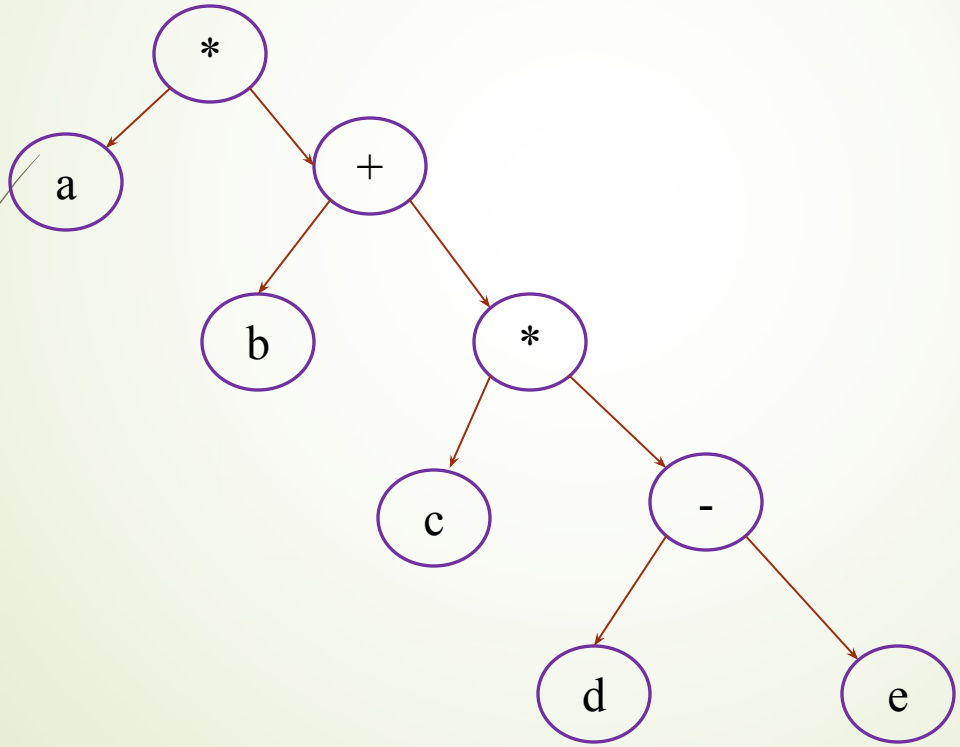
  Postfix Expression: a b * c + d f / -



Constructing a Tree

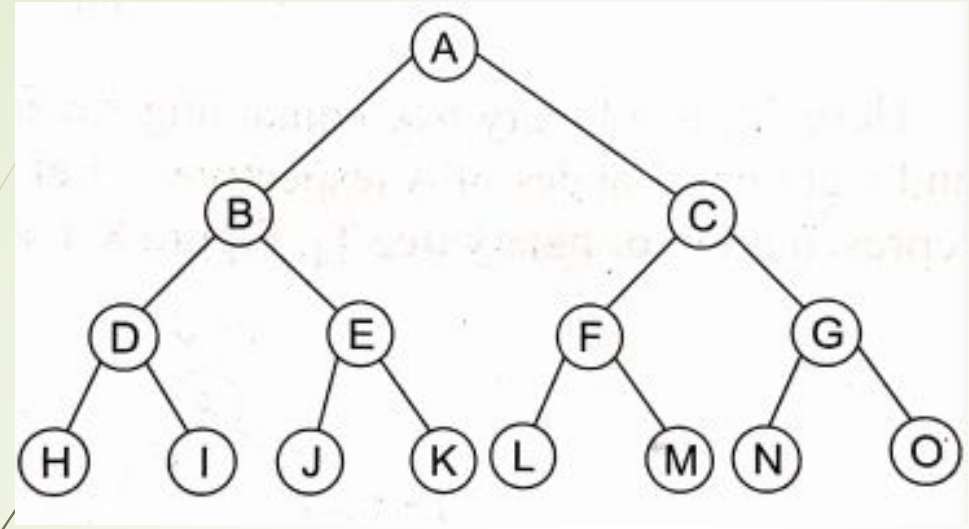
- Given the Prefix Expression Construct a Expression Tree

  Prefix Expression:* a + b * c – d e



Binary Tree Representation

Array Representation of Tree

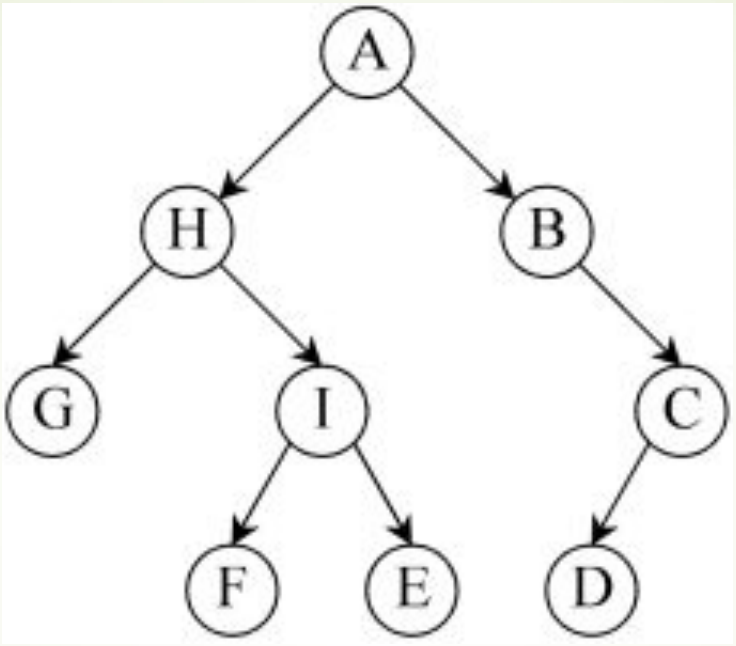


Memory allocated to array:
2^(height+1) - 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

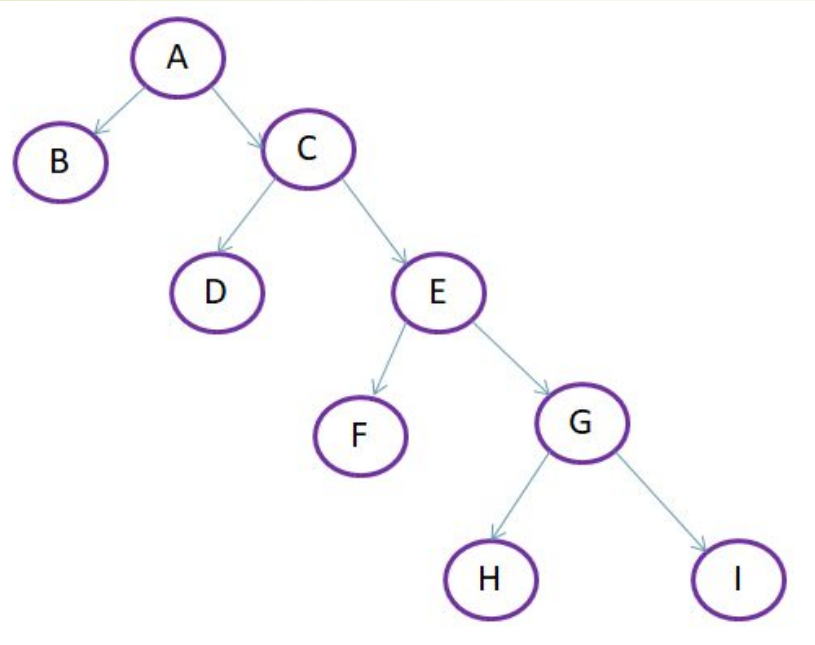
- Notice:
  - The left child of index i is at index 2*i+1
  - The right child of index i is at index 2*i+2

Array Representation of Tree



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	H	B	G	I		C			F	E			D	

Array Representation of Tree



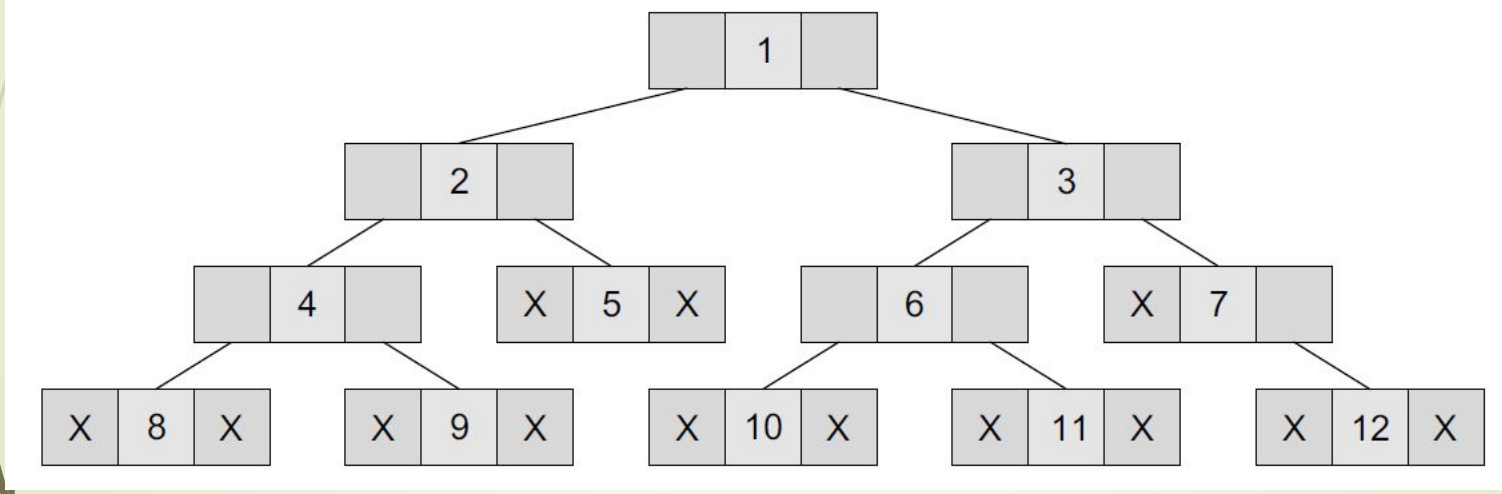
Memory allocated to array:
2^(height+1) - 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	B	C			D	E							F	G				

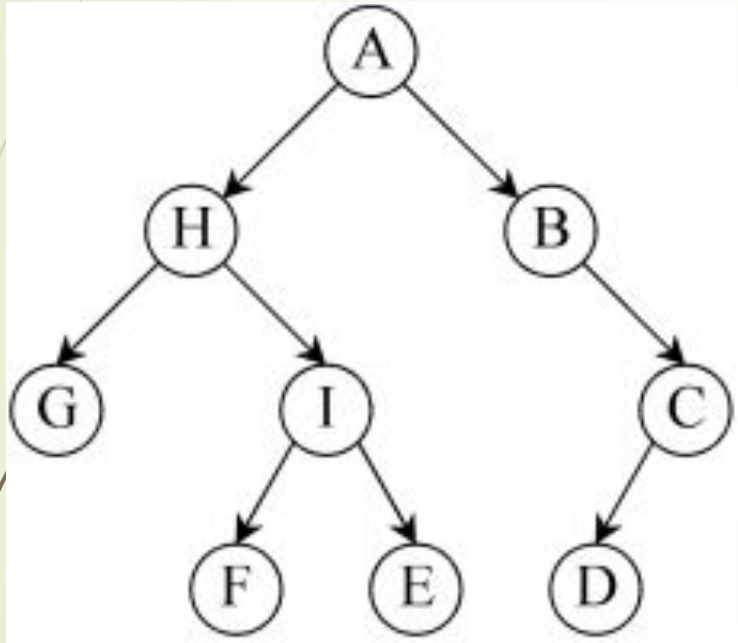
19	20	21	22	23	24	25	26	27	28	29	30							
										H	I							

linked Representation of Tree


```
struct node {
        struct node *left;
        int data;
        struct node *right;
};
```

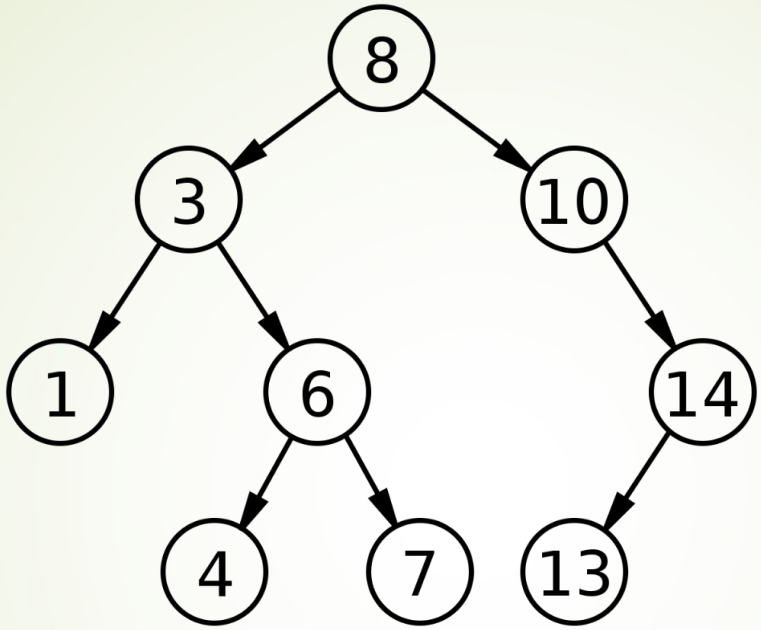


linked Representation of Tree: Static Allocation



Row#	Left	Data	Right
0	1	A	2
1	3	H	4
2	-1	B	5
3	-1	G	-1
4	6	I	7
5	8	C	-1
6	-1	F	-1
7	-1	E	-1
8	-1	D	-1
9			
10			

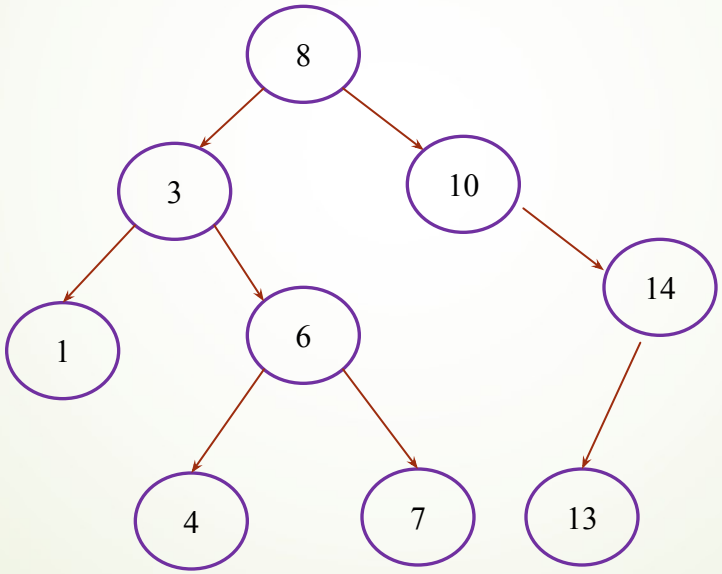
Binary Search Tree (BST)



- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Creation of BST

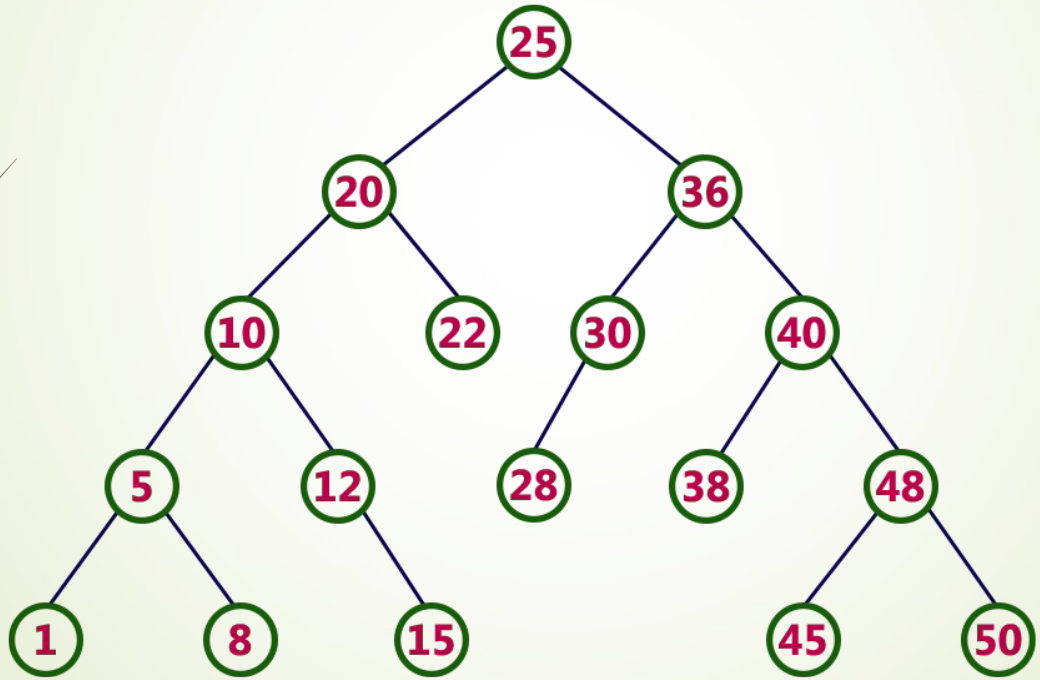
- Create a BST with following nodes:
- 8, 3, 10,14,13, 6, 7, 4, 1



Creation of BST

- Create a BST with following nodes:

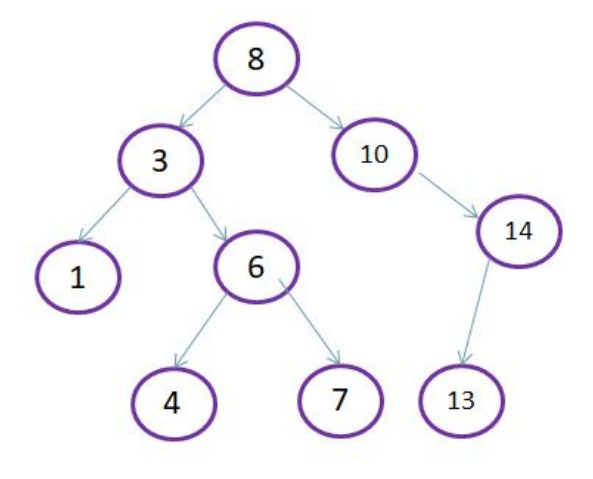
25, 36, 20,10, 5, 22, 30,12, 28, 40, 38, 48, 1, 8,15, 45, 50



BST: Node Declaration

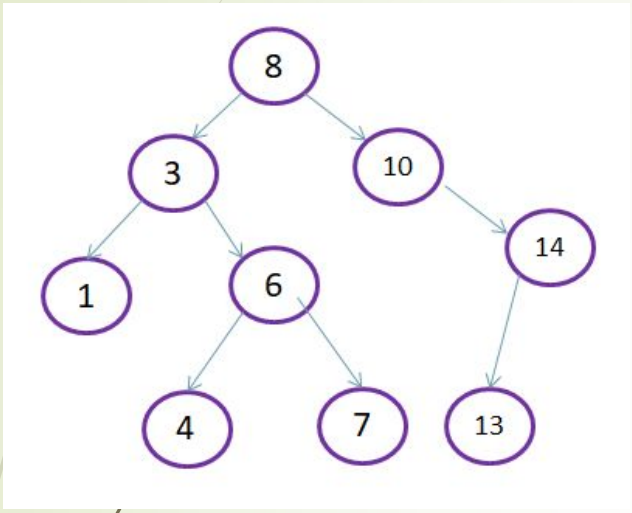
```
struct treenode
{
    struct treenode *left;
    int data;
    struct treenode *right;
};
typedef treenode treenode;
```

BST: Create()




```
void create(treenode *root){
int num, i;
int a[ ]= {8, 3, 10,14,13,6,7,4,1};
treenode *ptr, *temp, *prev;
root=NULL;
for(i=0;i<8;i++){
        temp=(treenode*)malloc(sizeof(treenode));
        temp->data=a[i];
        temp->left=NULL;
        temp->right=NULL;
```

BST: Create()




```
if(root==NULL)
        root=temp;
else{
        ptr=root;
        while(ptr != NULL){
                prev=ptr;
                if(ptr->data < temp->data)
                        ptr=ptr->right;
                else
                        ptr=ptr->left;
                }
                if(prev->data < temp->data)
                        prev->right=temp;
                else
                        prev->left=temp;
        }
    }
return root;
}
```

Thank You……

BST: Inorder Traversal

```
void printInorder(treenode* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}
```

BST: Preorder Traversal

```
void printPreorder(treenode* node)
{
    if (node == NULL)
        return;

    printf("%d ", node->data);
    printPreorder(node->left);
    printPreorder(node->right);
}
```

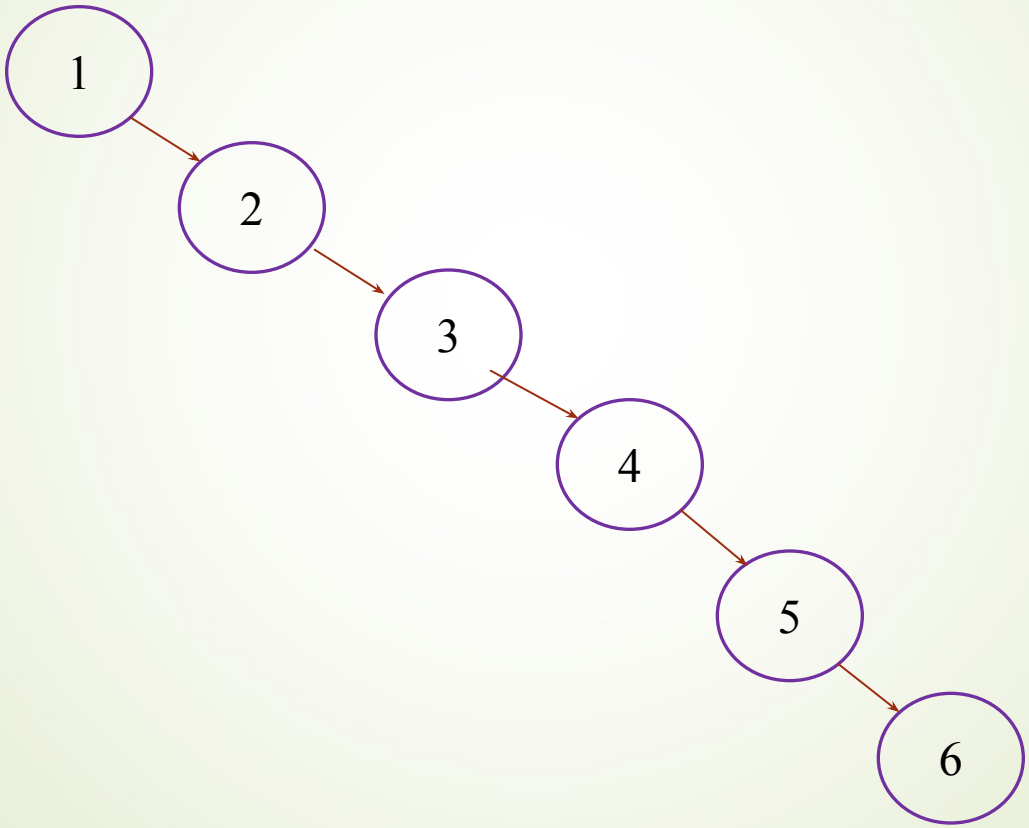
BST: Postorder Traversal

```
void printPreorder(treenode* node)
{
    if (node == NULL)
        return;

    printPreorder(node->left);
    printPreorder(node->right);
    printf("%d ", node->data);
}
```

Creation of BST

- Create a BST with following nodes: 1,2,3,4,5,6



Skewed BST

Binary Tree Deletion

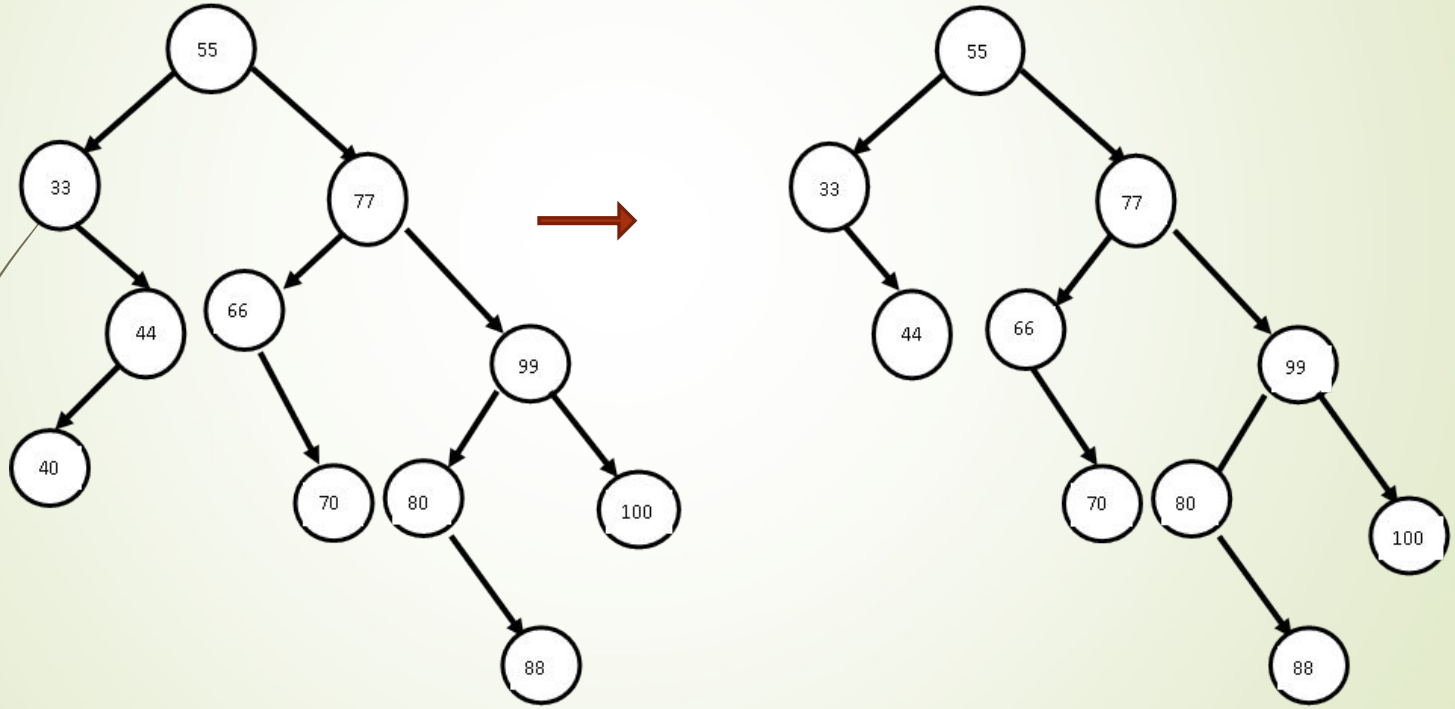
Binary Search Tree Deletion

When we delete a node, three possibilities arise.

- ***1. Node to be deleted is leaf:***
  - Simply remove the Node from the tree
- ***2. Node to be deleted has only one child:***
  - Replace the Node by its only Child Node.
- ***3. Node to be deleted has two children:***
  - Find inorder successor of the Node.
  - Replace the Node by its inorder successor.

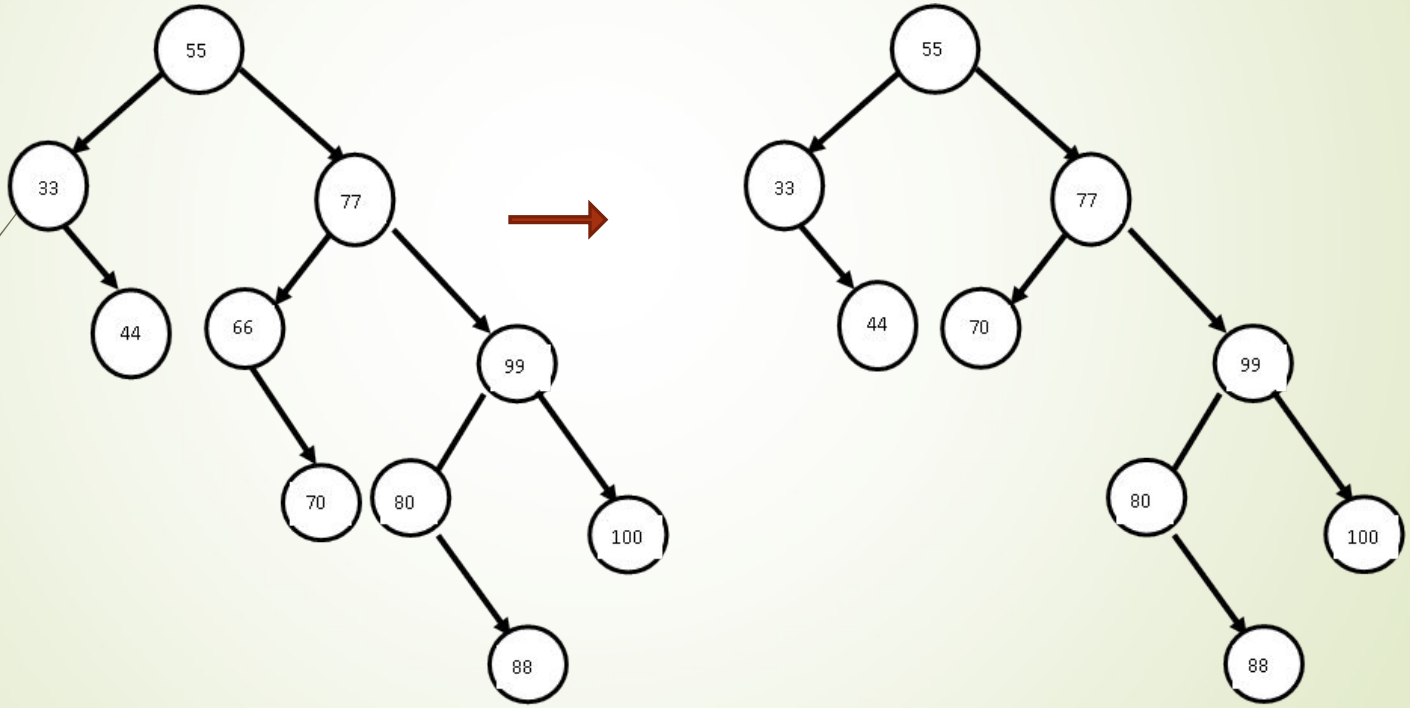
Binary Search Tree Deletion

- Delete 40



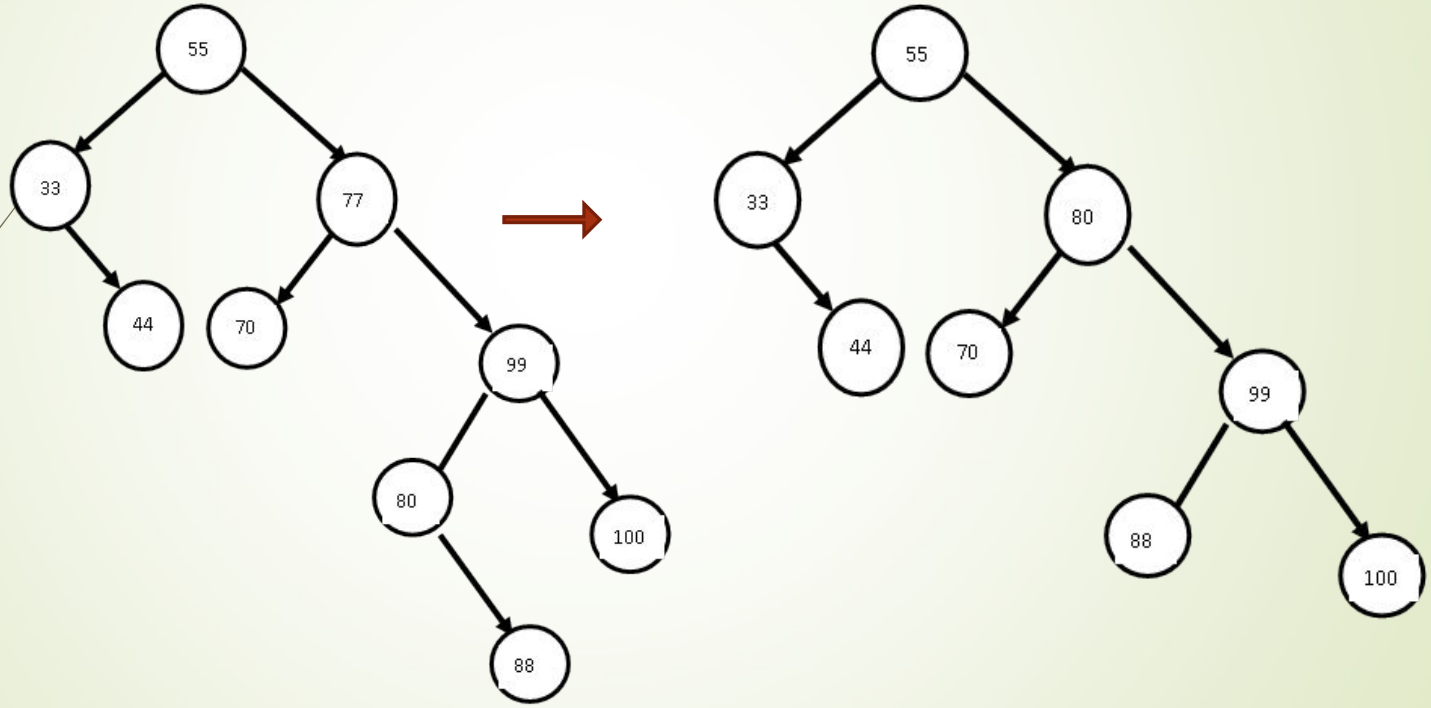
Binary Search Tree Deletion

- Delete 66



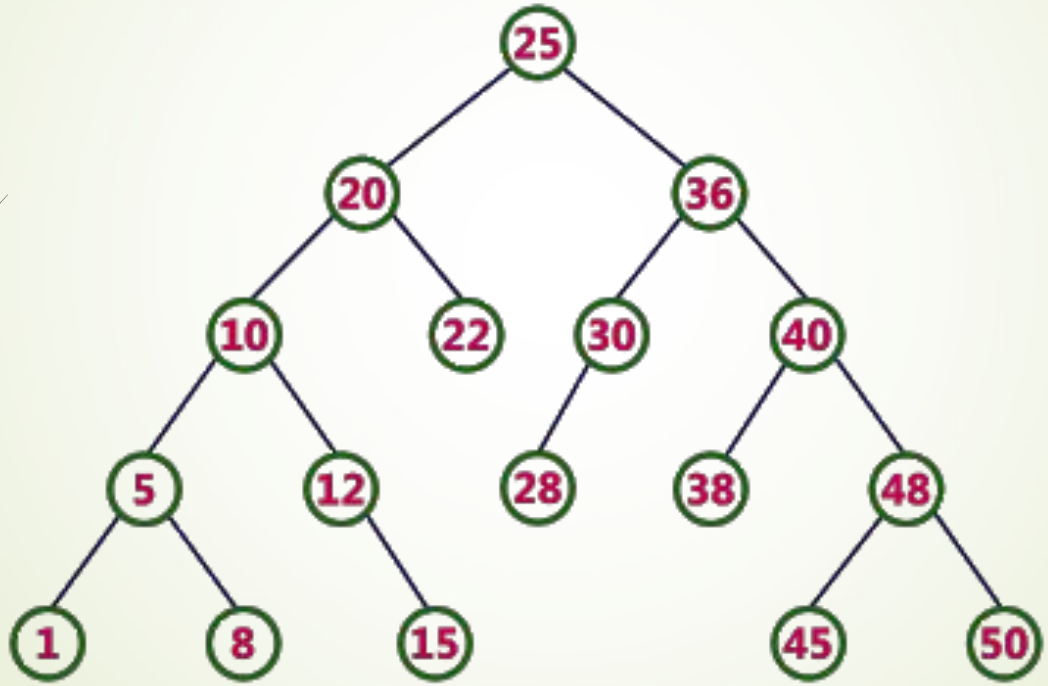
Binary Search Tree Deletion

- Delete 77



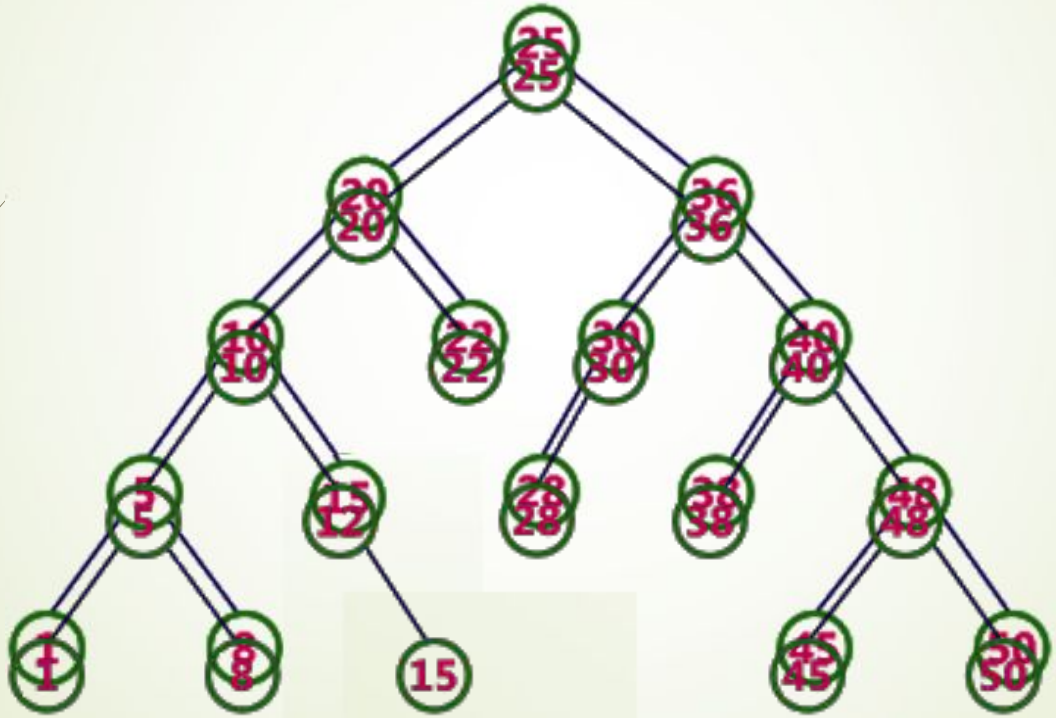
Example…

- Delete nodes in given sequence: 12, 22, 36, 25



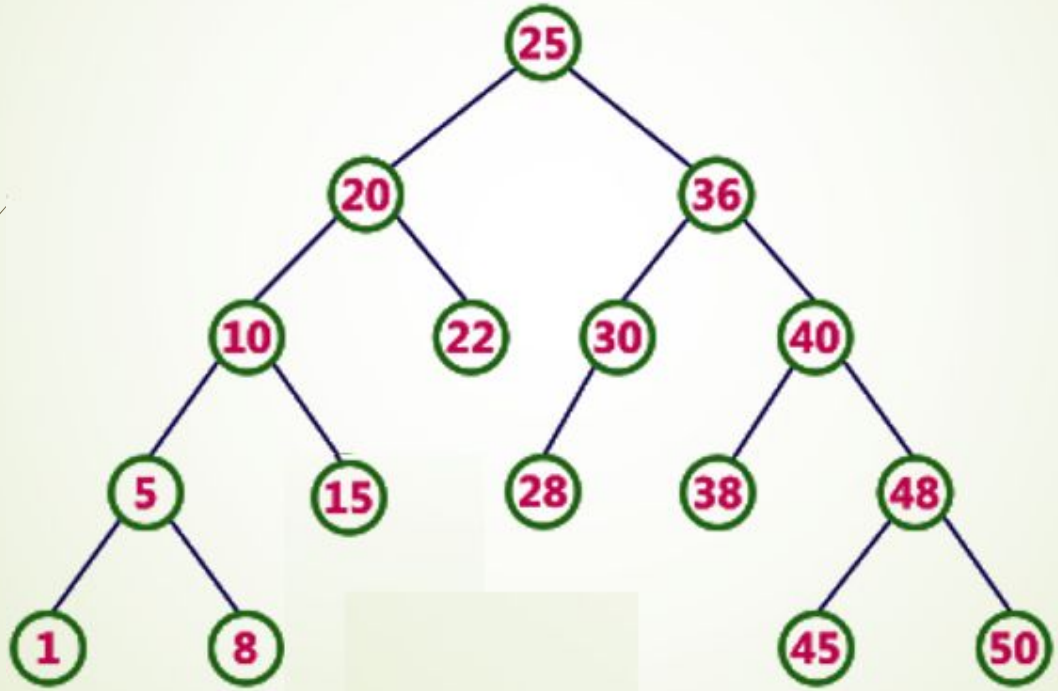
Example…

- Delete node 12



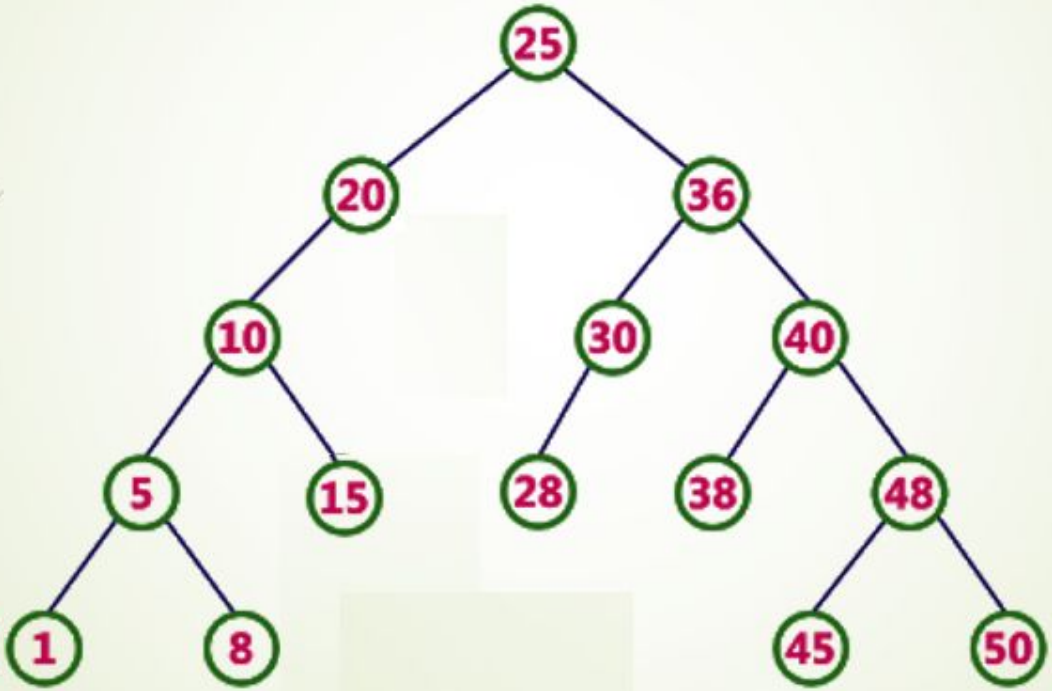
Example…

- Delete node 22



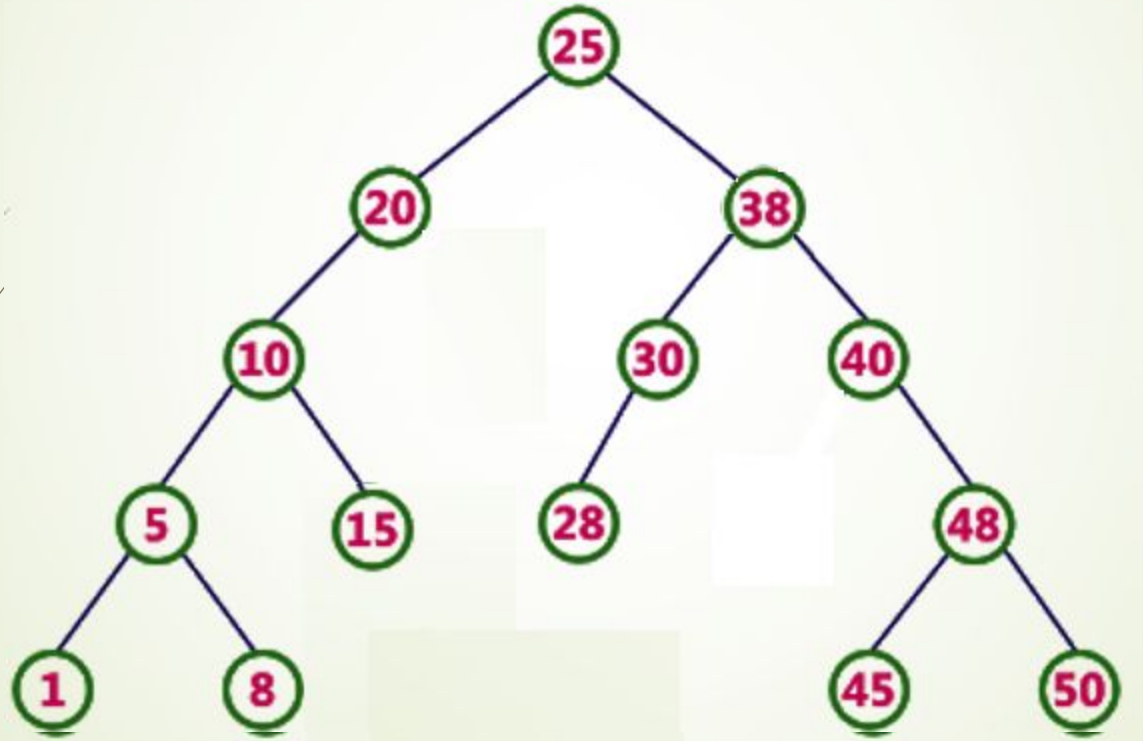
Example…

- Delete node 36



Example…

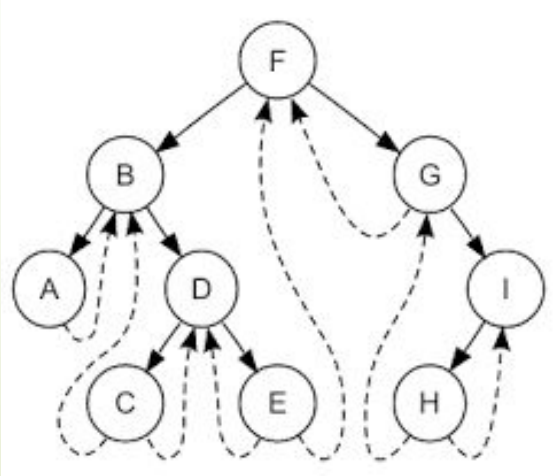
- Delete node 25



Threaded Binary Tree

threaded binary trees

- Binary Trees have a lot of wasted space: The leaf nodes each have 2 NULL pointers.
- We can use these pointers to help us in inorder traversal.
- Thread: a pointer to other node in the tree for replacing NULL links.
- But we need to know if pointer is an actual link to child node or a thread



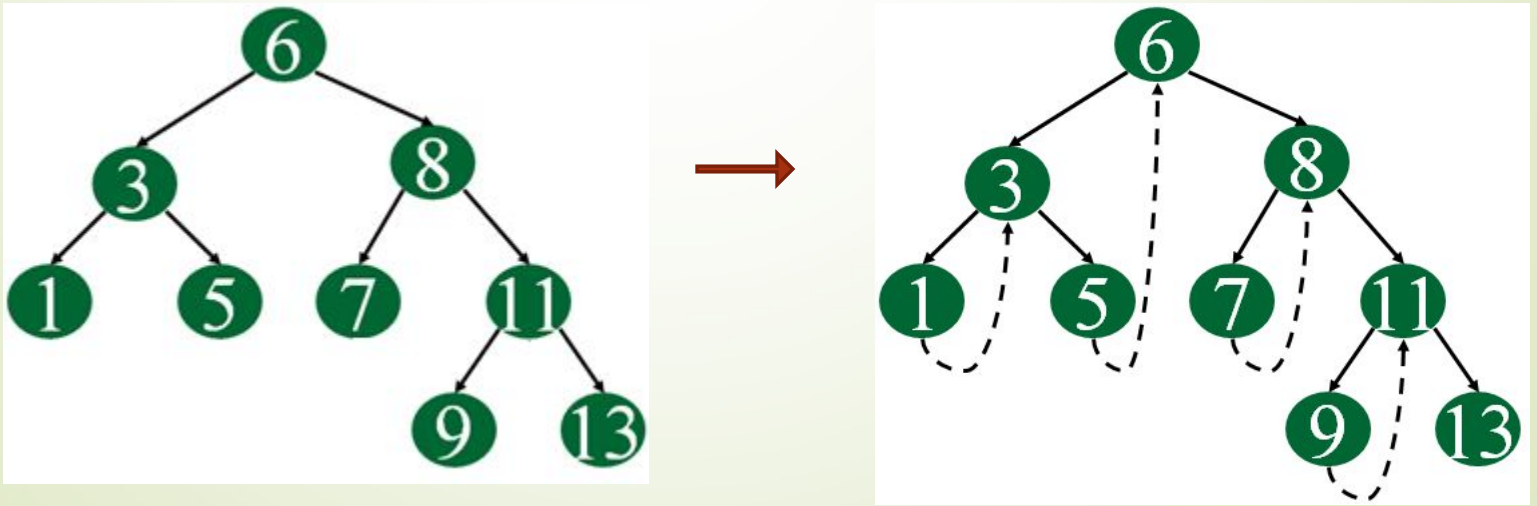
Inorder threaded binary trees

- An Inorder Threaded Binary Tree is a binary tree in which every node that does not have a rightchild has a THREAD to its INORDER successor.

- By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

Inorder threaded binary trees

- A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
- The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion.



threaded binary trees

There are two types of threaded binary trees.

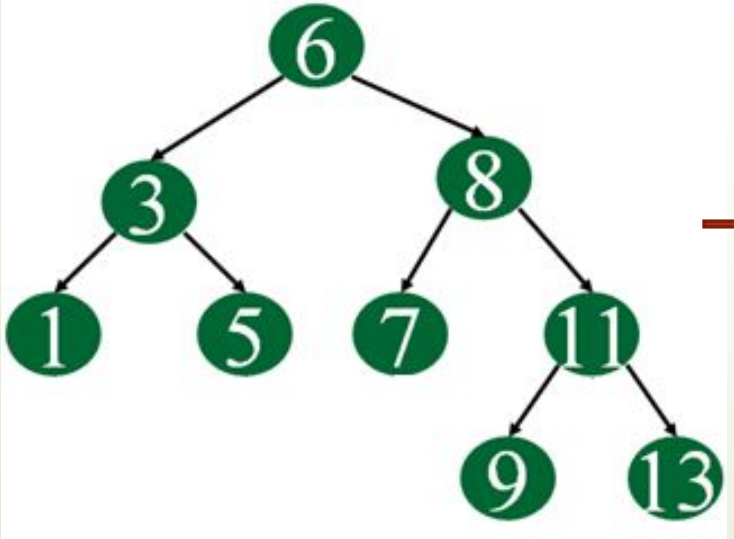
- **Single Threaded:** NULL right pointers is made to point to the inorder successor.
- **Double Threaded:** Both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

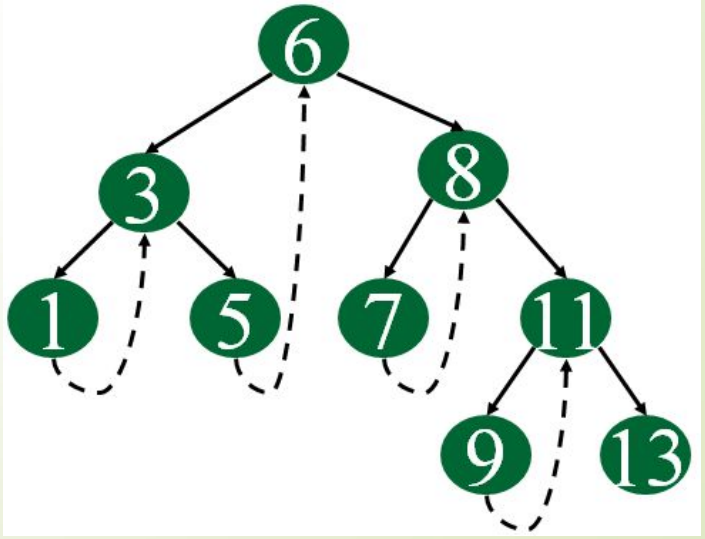
threaded binary trees

```
struct Node
{
        Node *left;
        int data;
        Node * left;
        bool rightThread;
}
```

left	data	right	rightThread

left	data	right	0/1

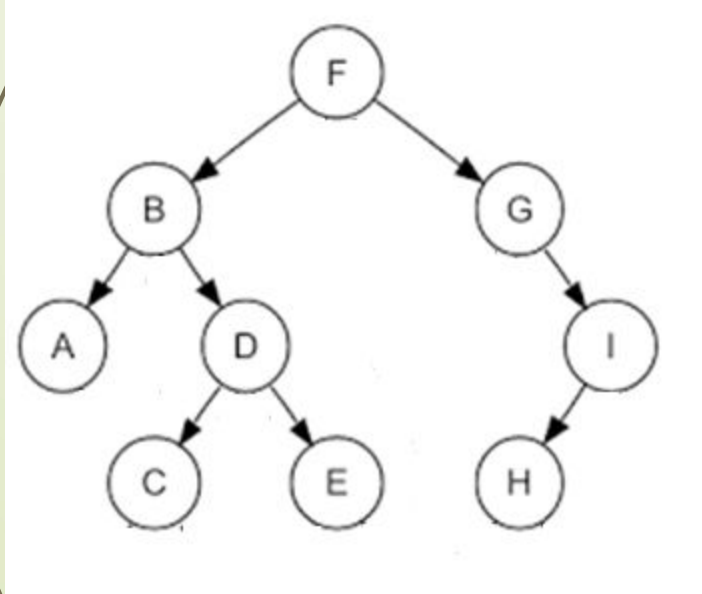


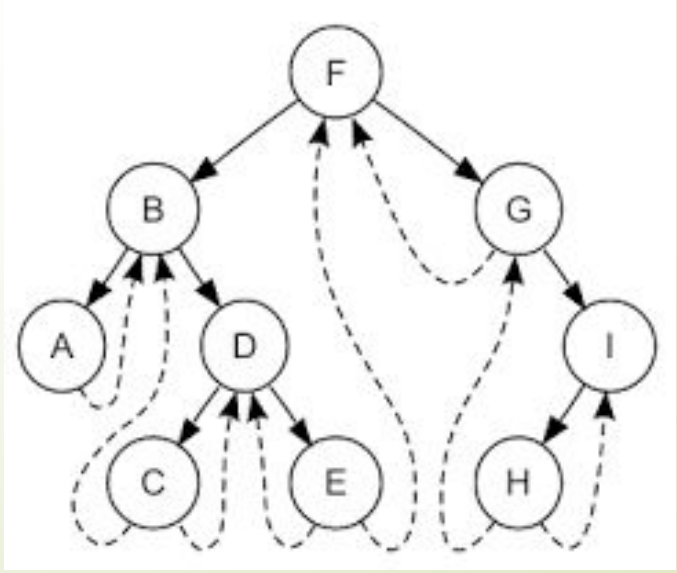


Double threaded binary trees

```
struct Node
{
        bool leftThread;
        Node *left;
        int data;
        Node *right;
        bool rightThread;
}
```

Left Thread	l e f t	d a t a	r i g h t	Right Thread





Program for inorder traversal of threaded binary tree

```
struct Node* leftMost(struct Node *node)
{

        if (node == NULL)
        return NULL;
        while (node ->left != NULL)
                node = node ->left;
        return node;
}
```

Program for inorder traversal of threaded binary tree

```
void inOrder(struct Node *root)
{
        struct Node *cur = leftmost(root);
        while (cur != NULL)
        {
                printf("%d ", cur->data);
                // Check if this node is a thread node
                if (cur->rightThread)
                        cur = cur->right;
                else
                        cur = leftmost(cur->right);
        }
}
```