

Data Structures

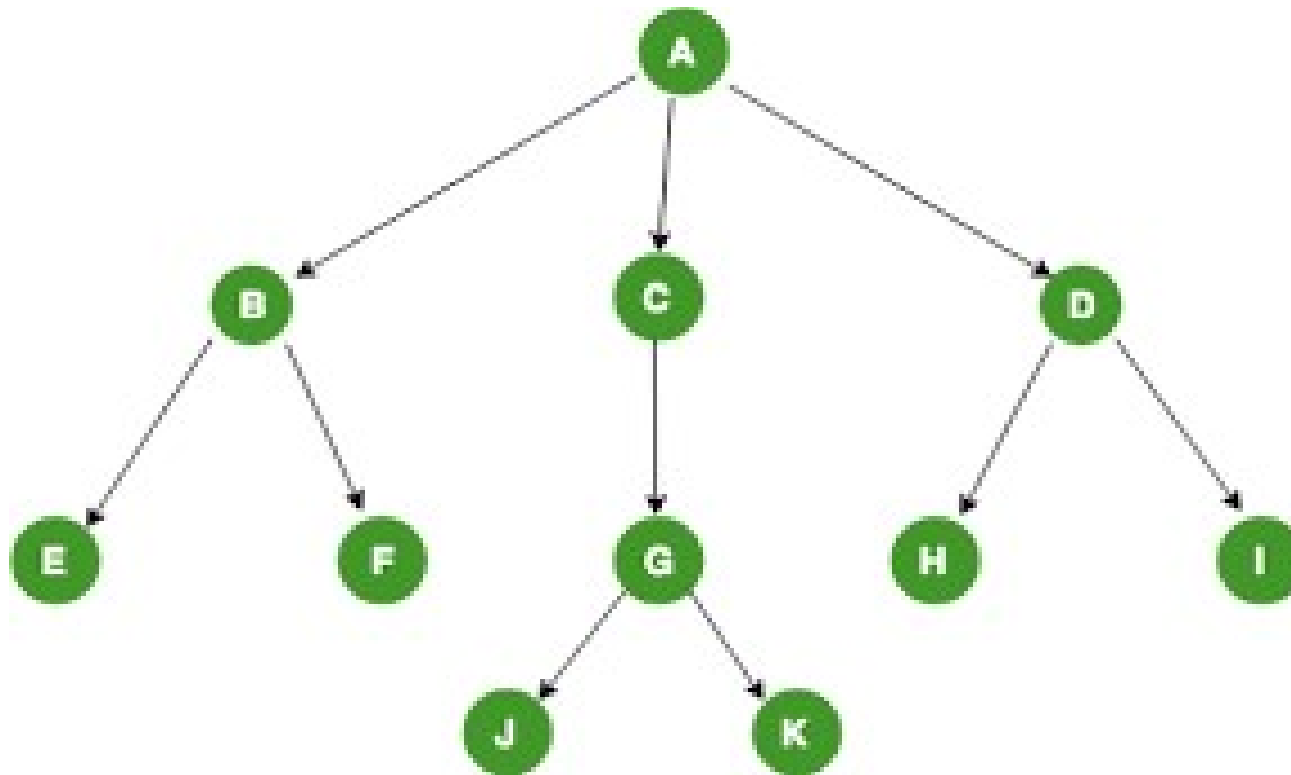
TREES

TREE

A Tree:

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

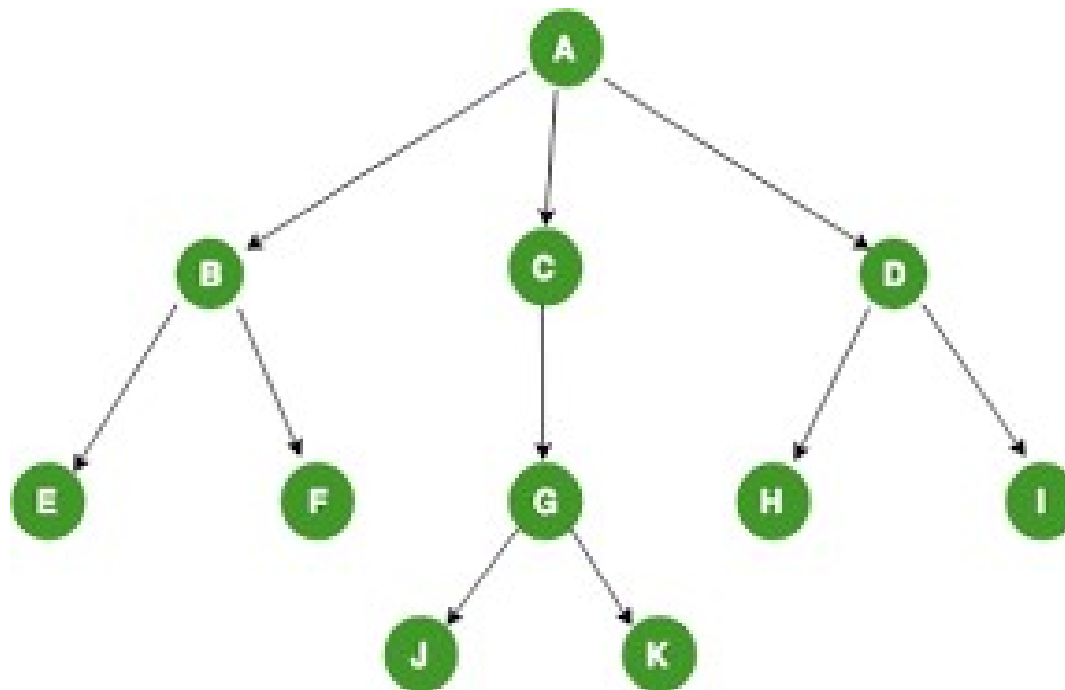
Basic Concept



Tree terminologies

Node→

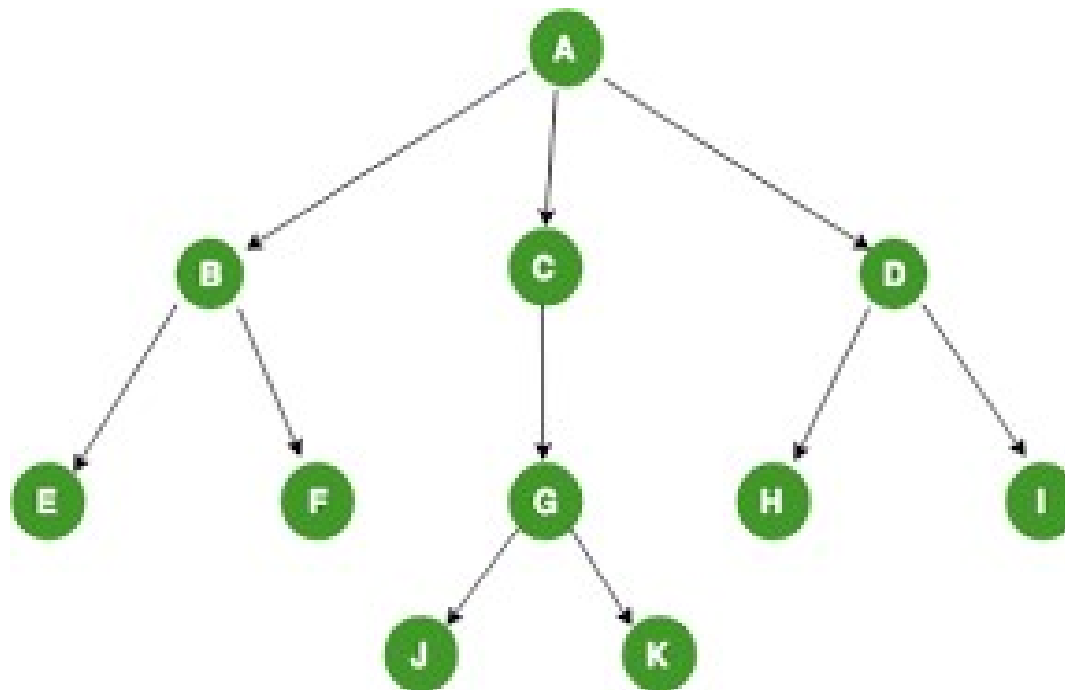
It is the data element of tree. Apart from storing a value it also specifies links to the other nodes



Tree terminologies

Root →

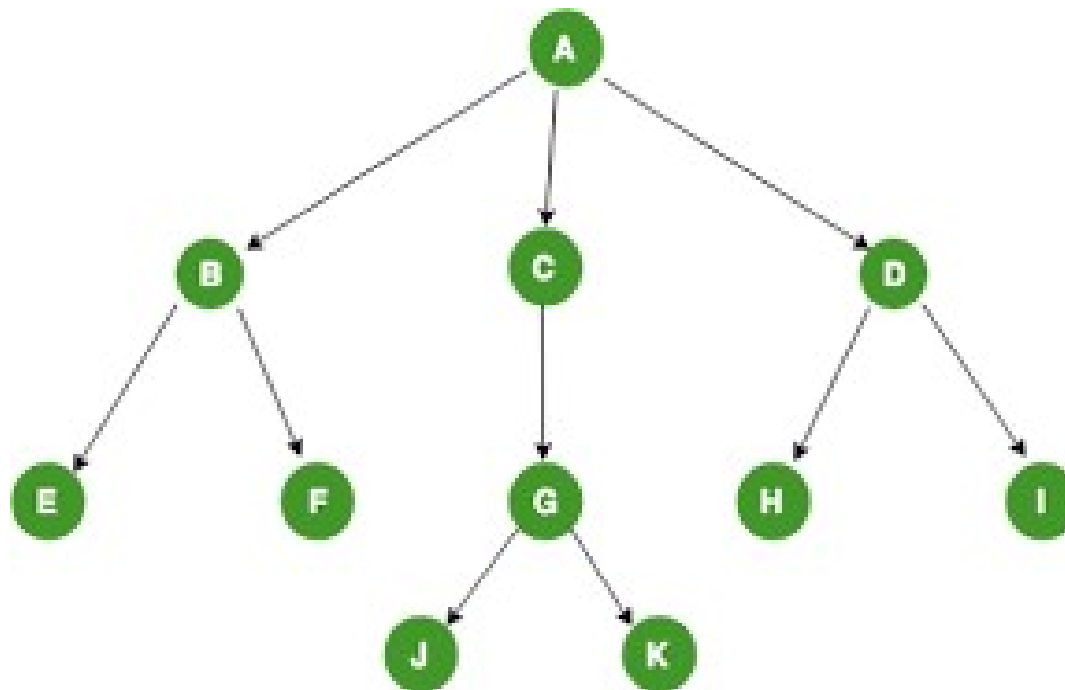
It is the top node in the tree



Tree terminologies

Parent→

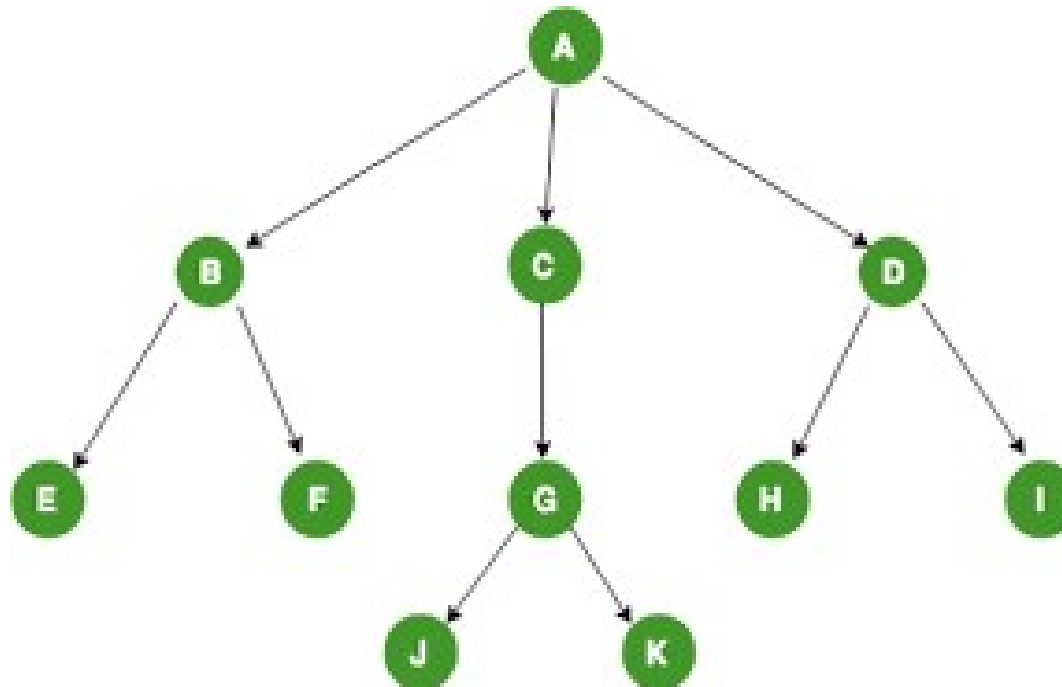
A node that has one or more child nodes



Tree terminologies

Child→

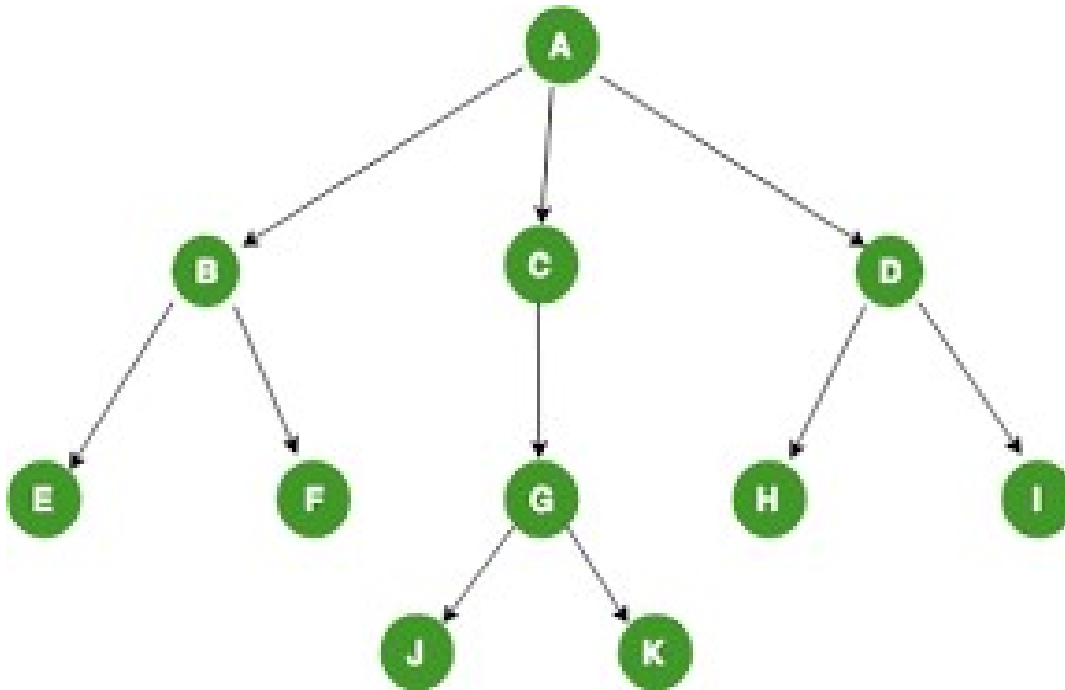
All nodes in a tree except the root node are child nodes of their immediate predecessor nodes



Tree terminologies

Leaf/ Terminal Node→

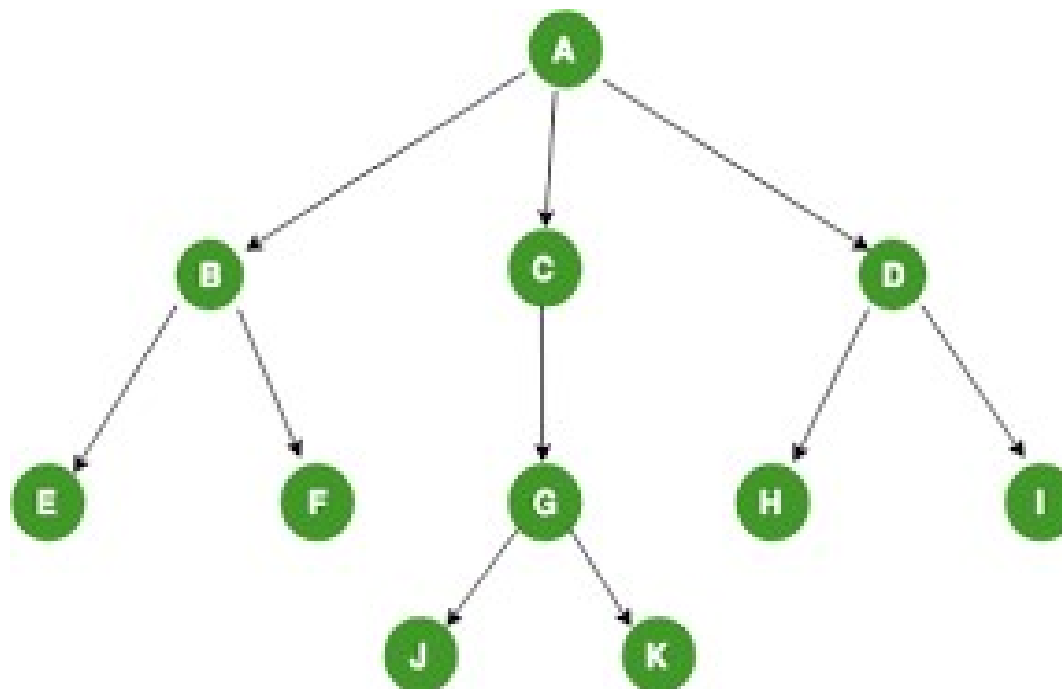
It is the terminal node that does not have any child nodes



Tree terminologies

Internal Node→

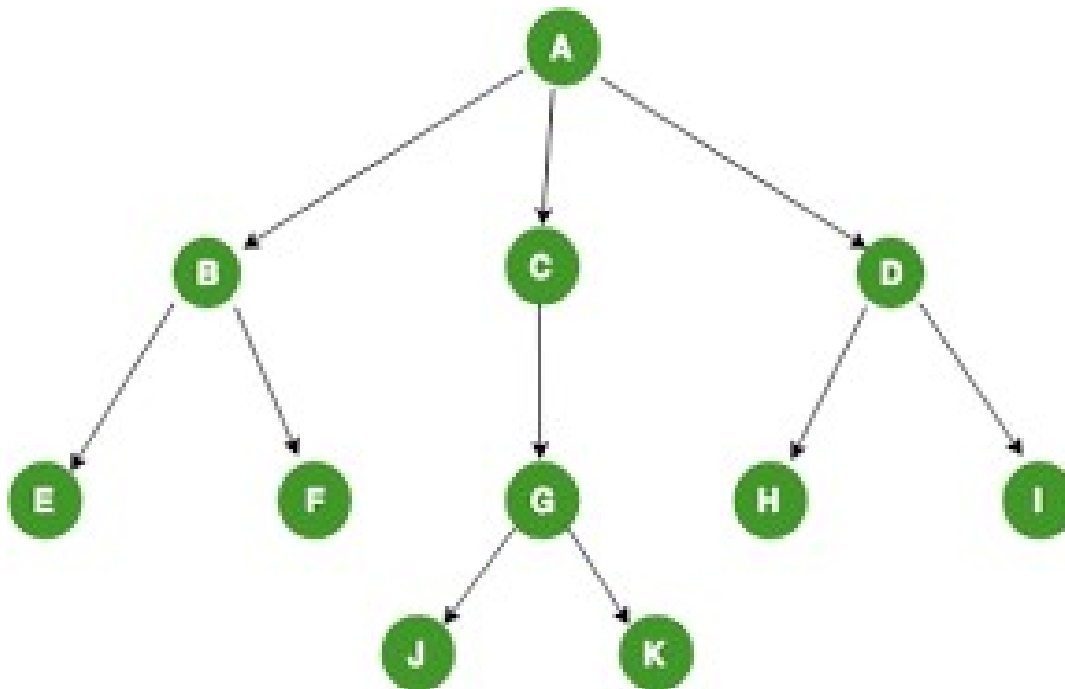
All nodes except root and leaf nodes are referred as internal nodes



Tree terminologies

Non Terminal Node→

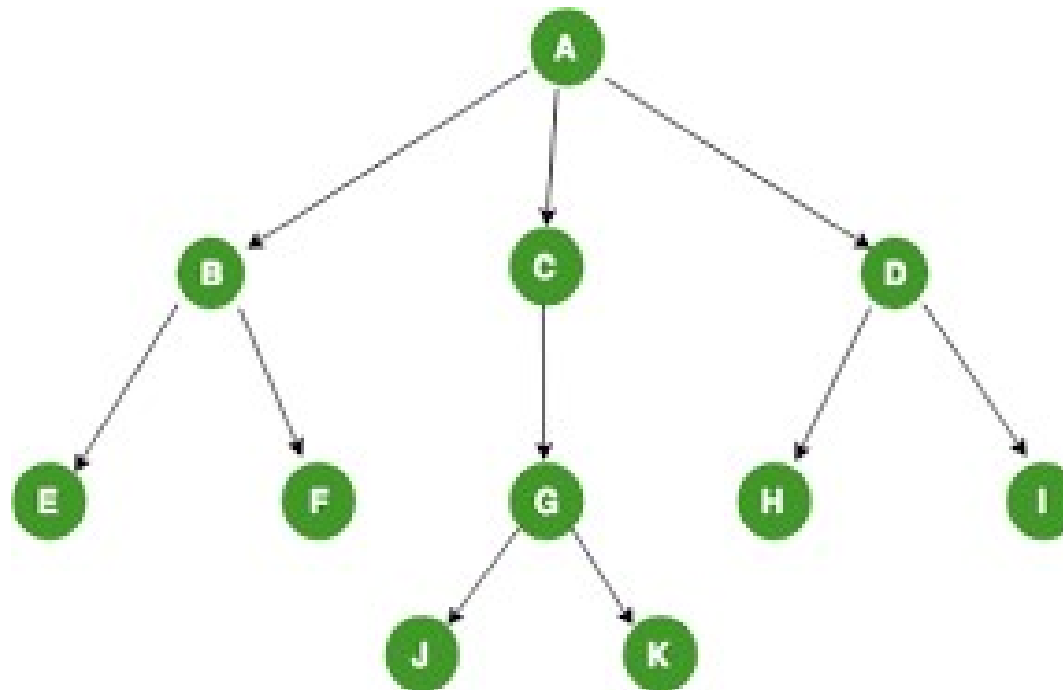
All nodes except leaf nodes are referred as Non Terminal nodes



Tree terminologies

Sibling→

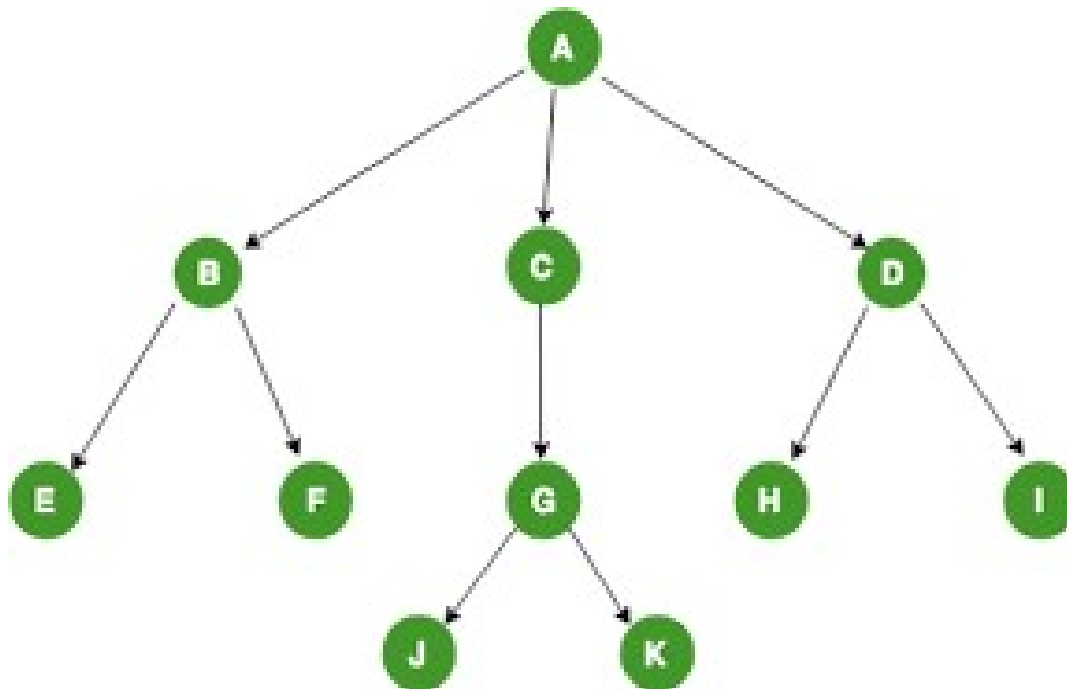
All child nodes of same parent are referred as Siblings



Tree terminologies

Degree→

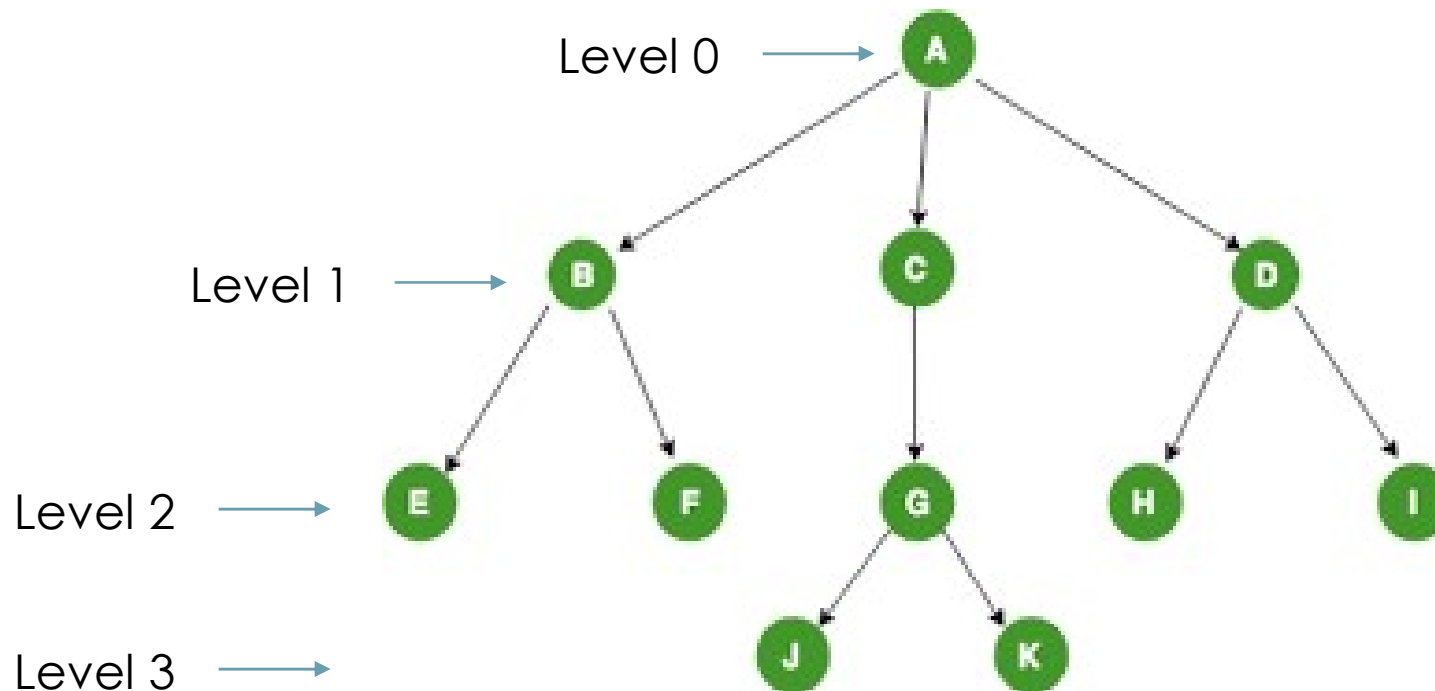
The degree of a node is the number of subtrees coming out of the node



Tree terminologies

Level→

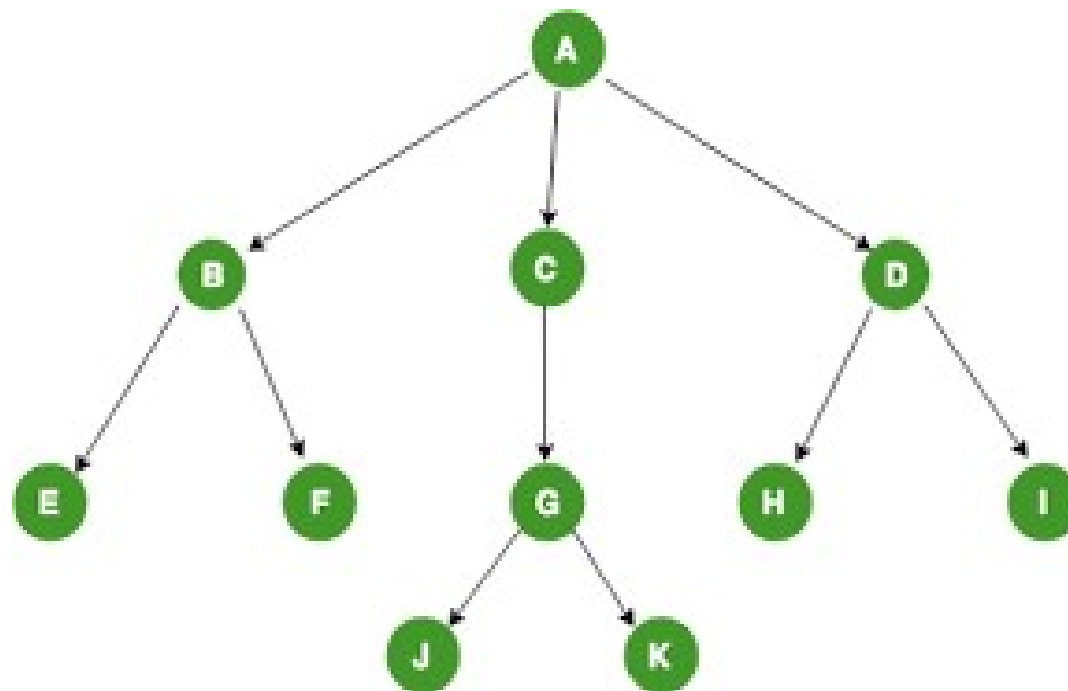
All the tree nodes are present at different levels. Root node is at level 0.



Tree terminologies

Height OR Depth→

It is maximum level of a node in the tree

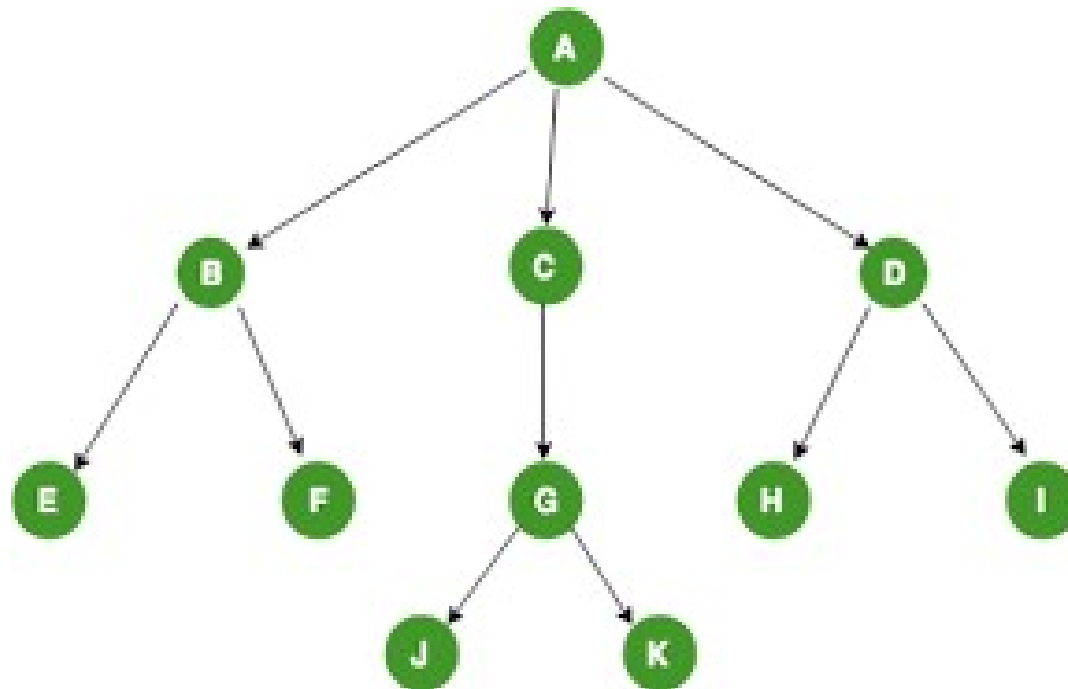


Height= 3

Tree terminologies

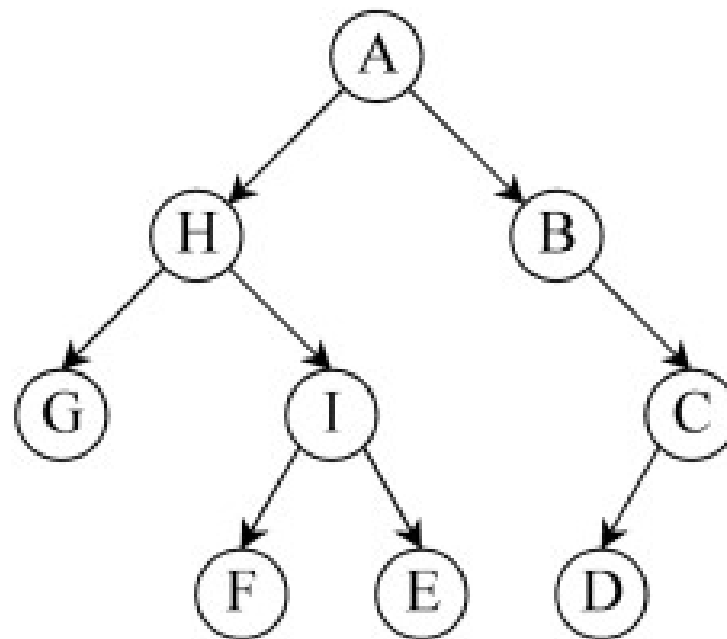
Path→

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



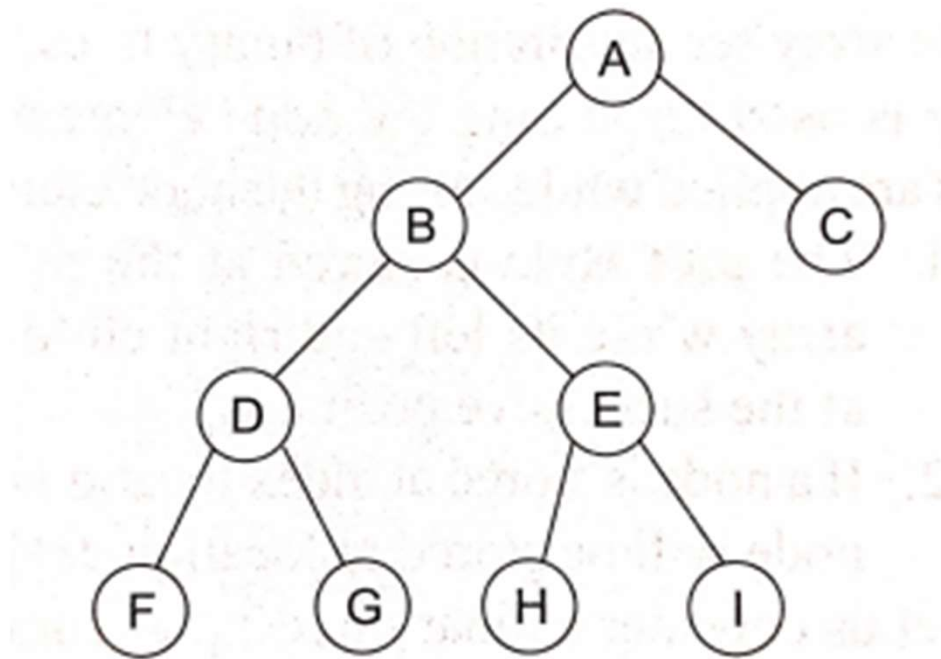
Binary Tree

- 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: Pointer to left subtree. Pointer to right subtree



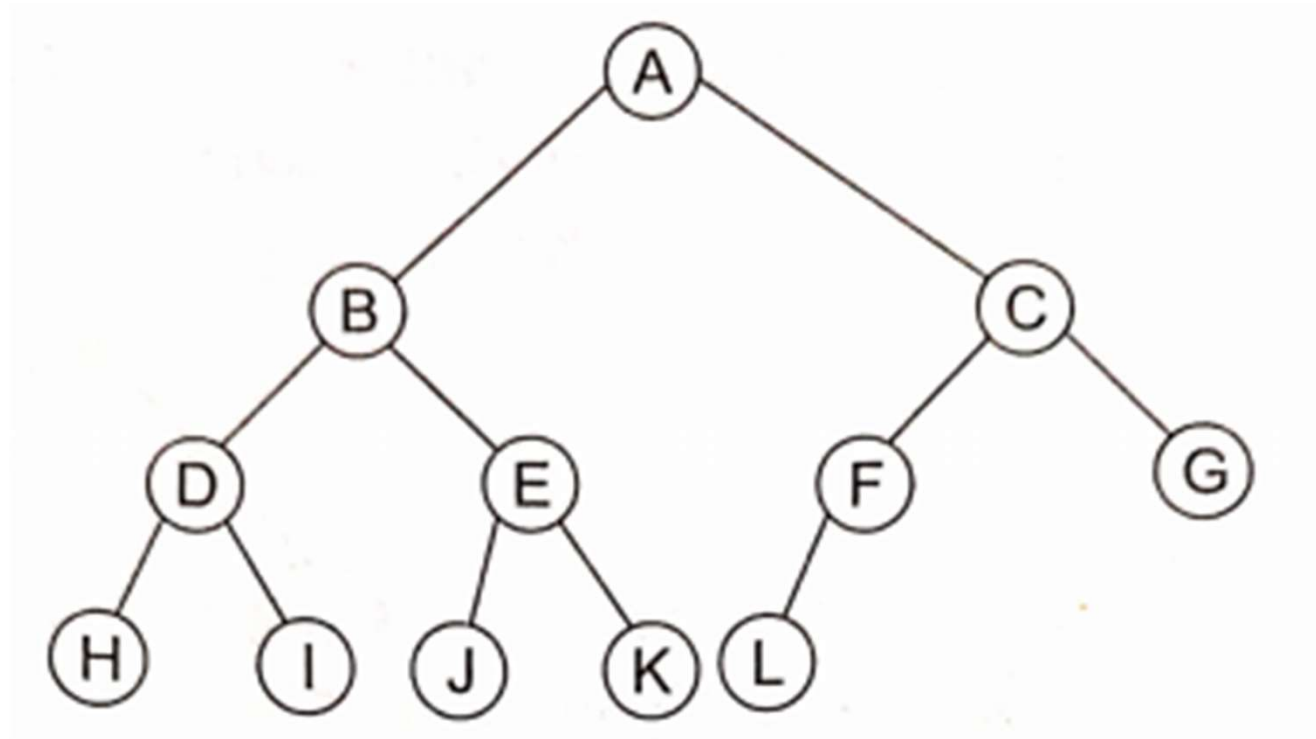
Strictly Binary Tree

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



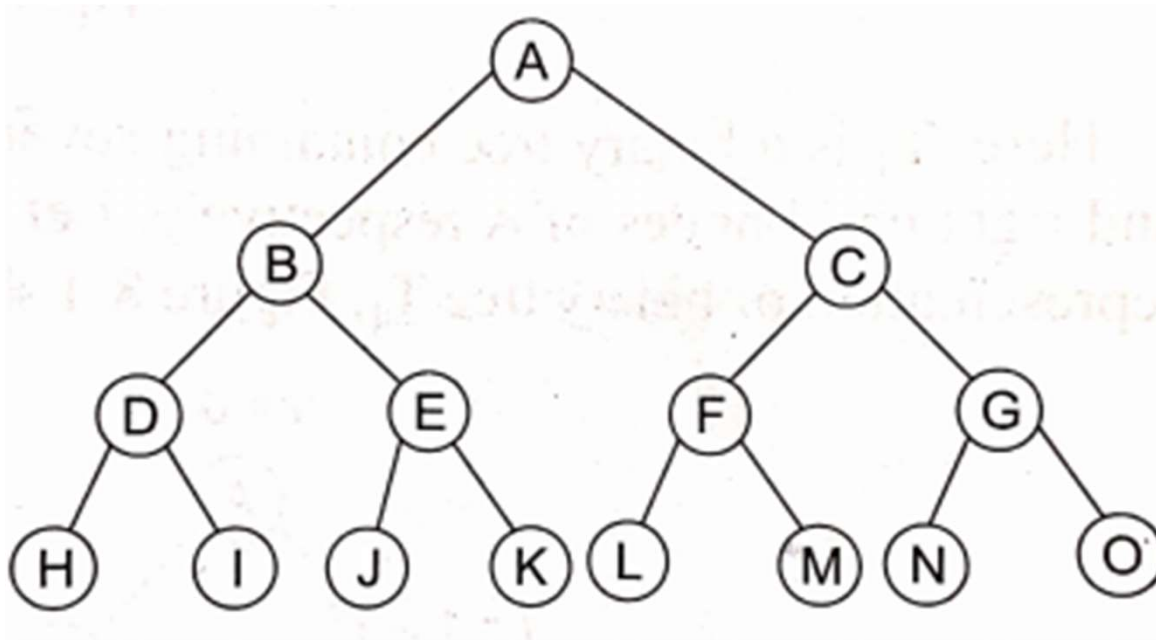
Complete Binary Tree

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



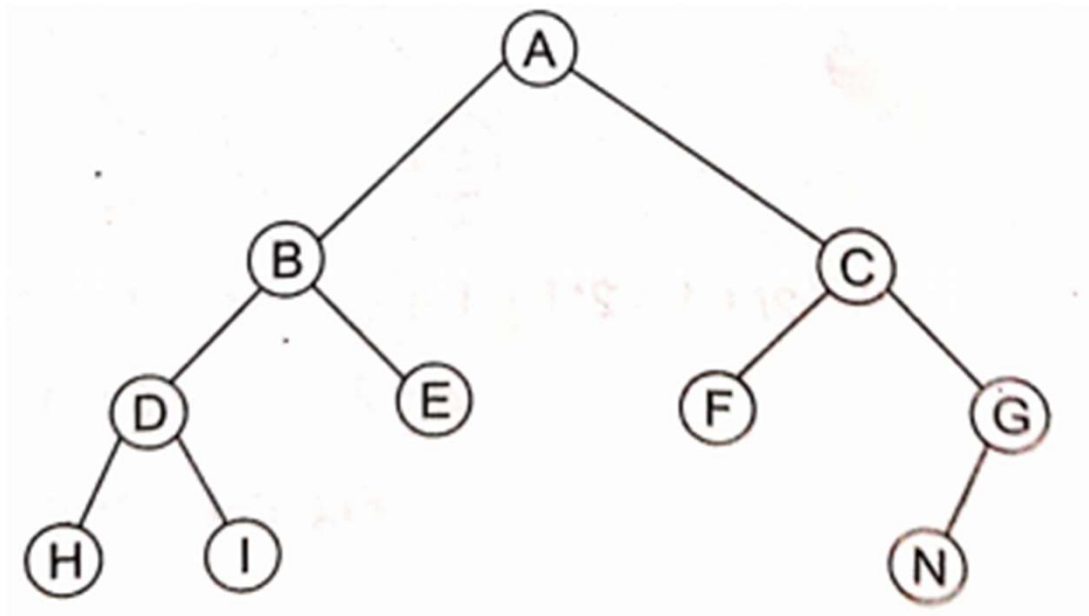
Perfect Binary Tree

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 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:

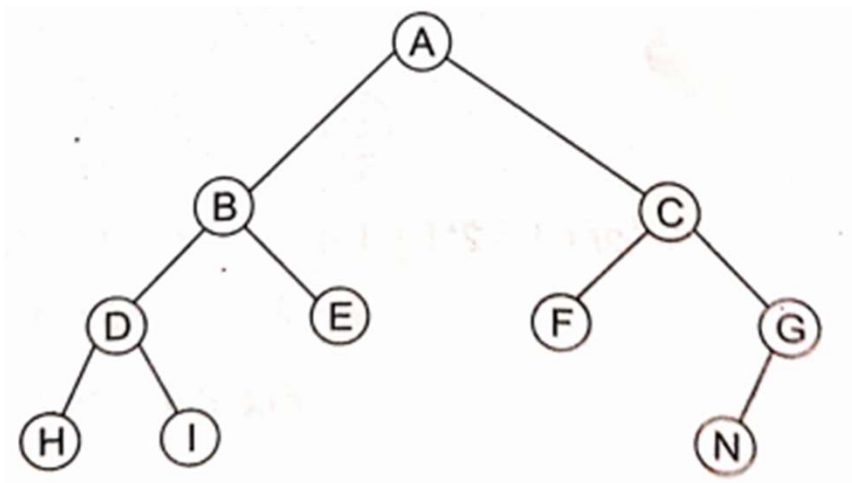
1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal

Preorder Traversal

Preorder Traversal:

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

ROOT - LEFT - RIGHT



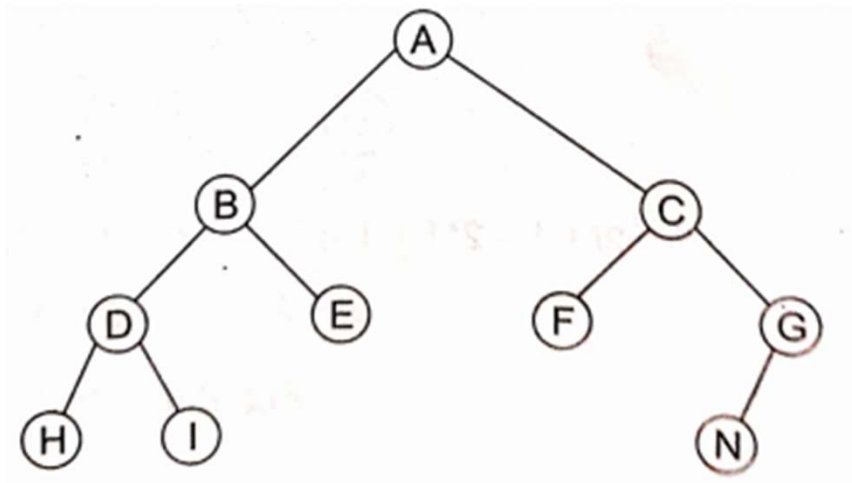
A B D H I E C F G N

Inorder Traversal

Inorder Traversal:

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

LEFT – ROOT - RIGHT



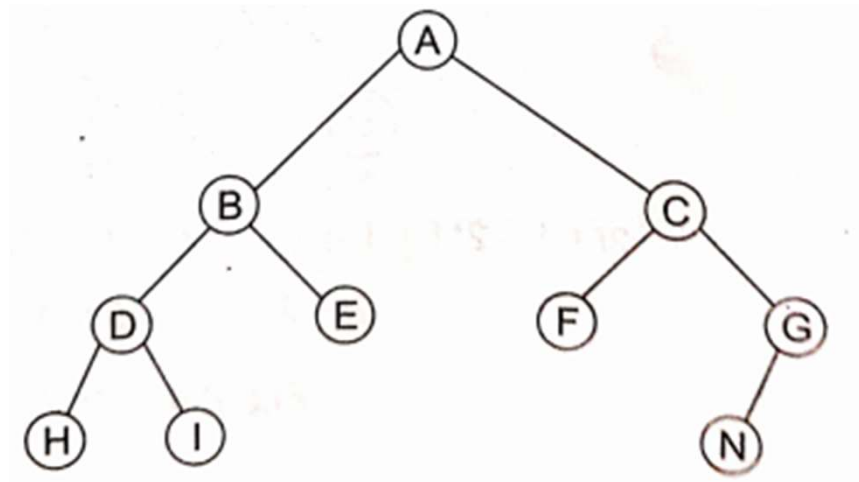
HDIB EAF CNG

Postorder Traversal

Postorder Traversal:

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

LEFT - RIGHT - ROOT



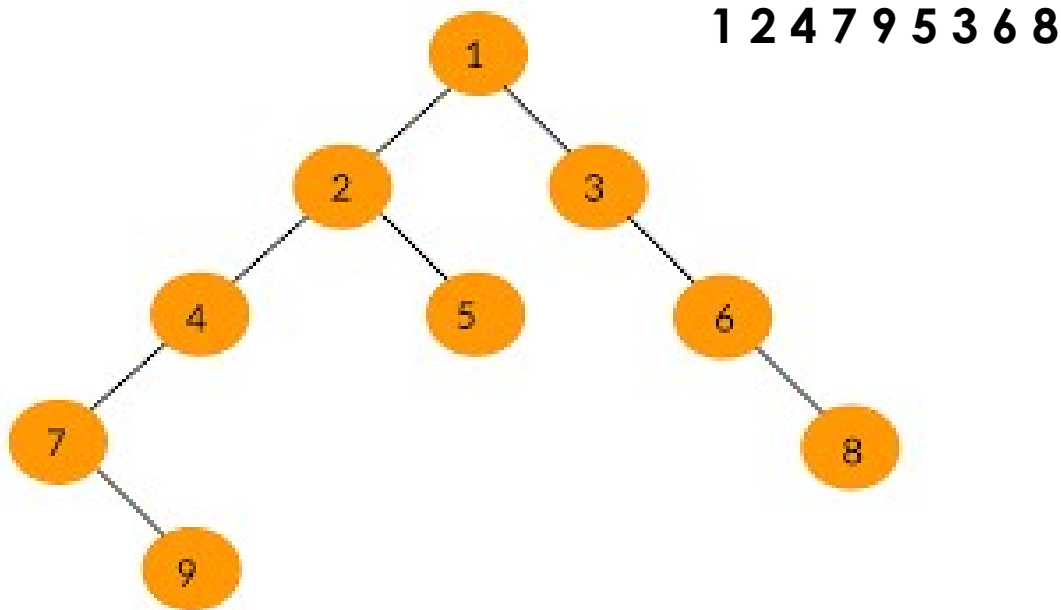
H I D E B F N G C A

Preorder Traversal

Preorder Traversal:

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

ROOT - LEFT - RIGHT

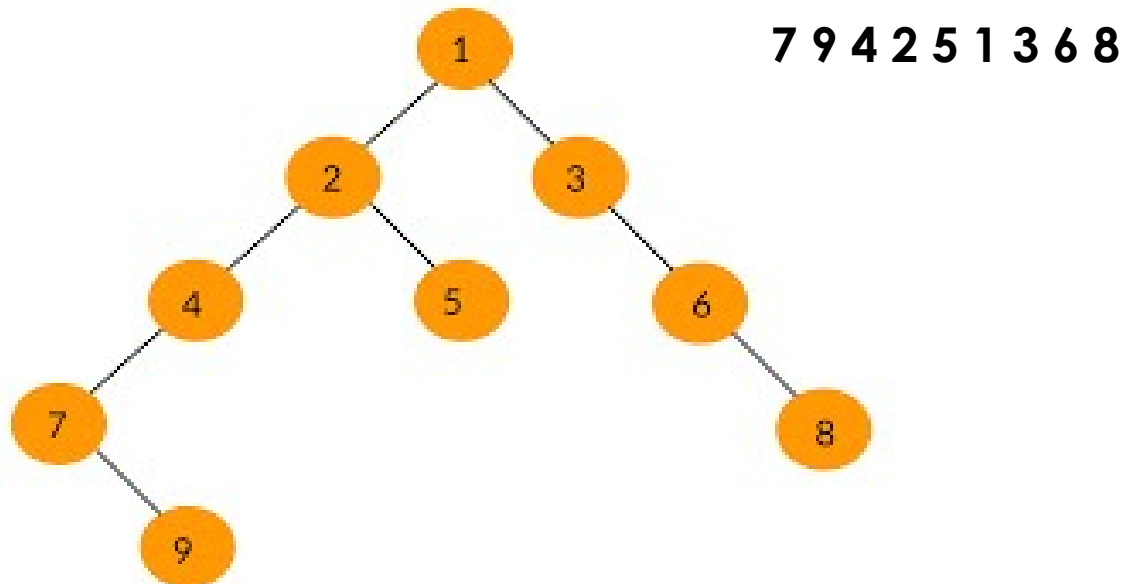


Inorder Traversal

Inorder Traversal:

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

LEFT – ROOT - RIGHT

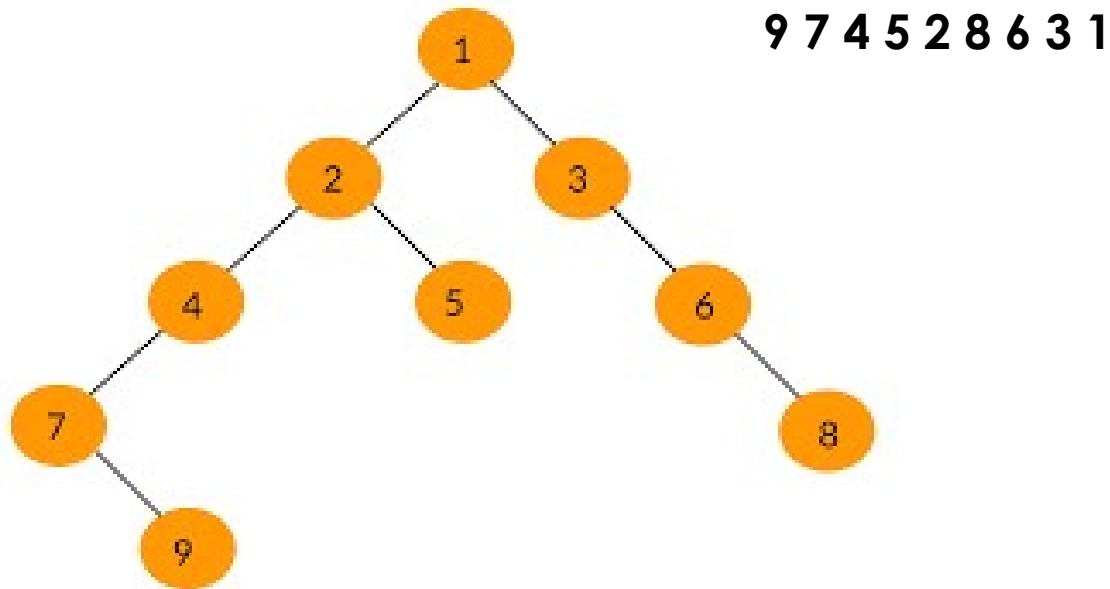


Postorder Traversal

Postorder Traversal:

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

LEFT - RIGHT - ROOT

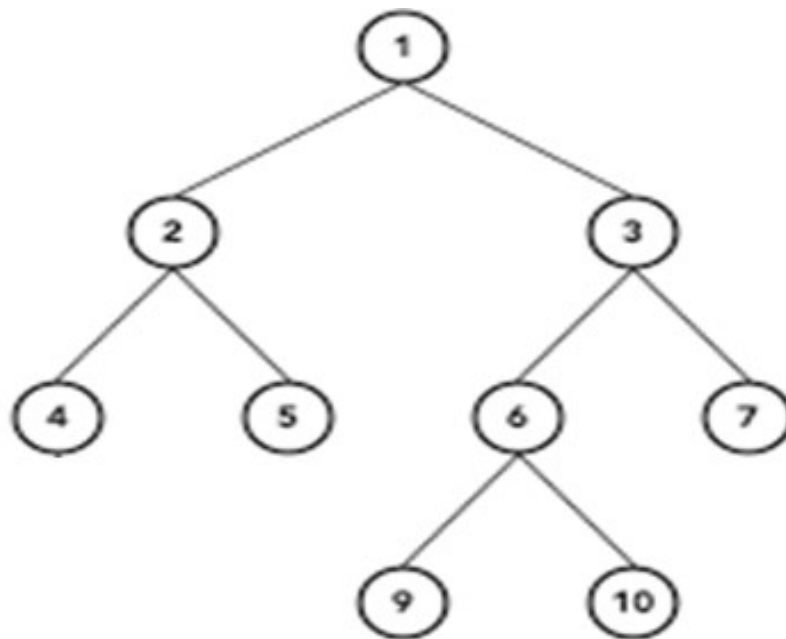


Preorder Traversal

Preorder Traversal:

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

ROOT - LEFT - RIGHT



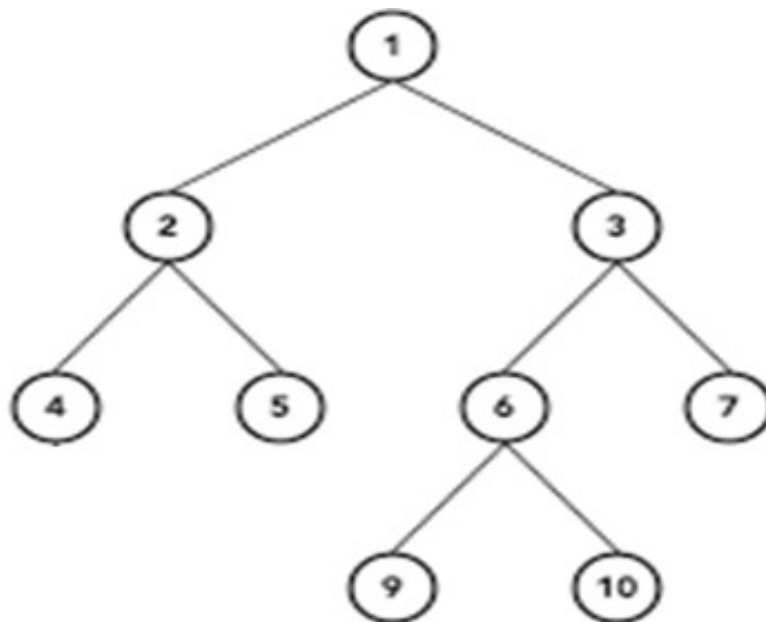
1 2 4 5 3 6 9 10 7

Inorder Traversal

Inorder Traversal:

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

LEFT – ROOT - RIGHT



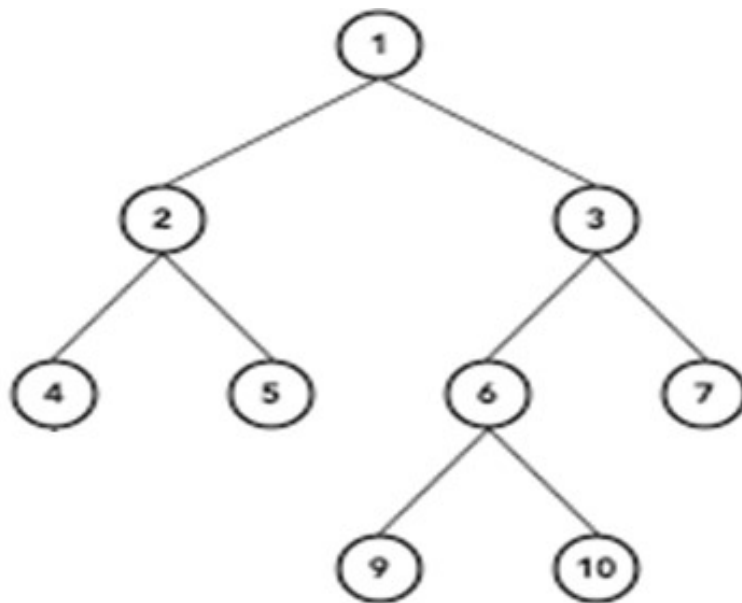
4 2 5 1 9 6 10 3 7

Postorder Traversal

Postorder Traversal:

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

LEFT - RIGHT - ROOT



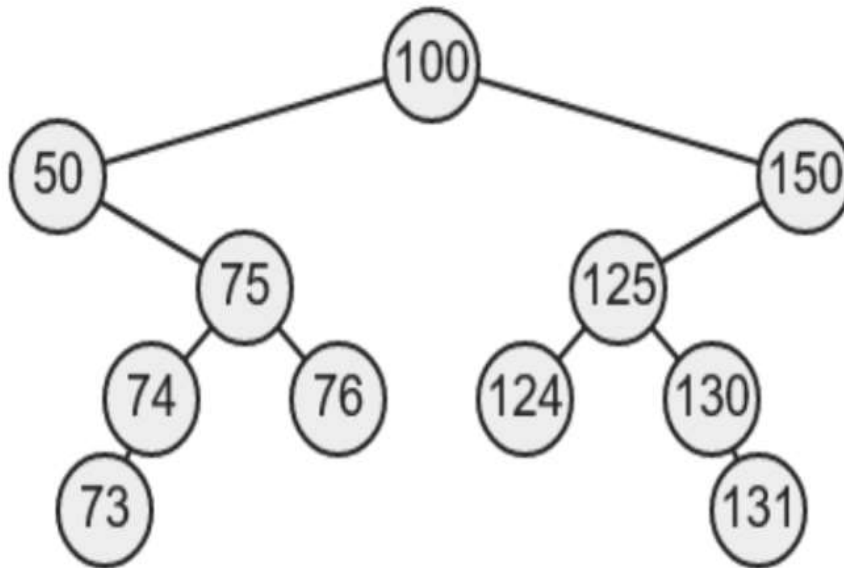
4 5 2 9 10 6 7 3 1

Preorder Traversal

Preorder Traversal:

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

ROOT - LEFT - RIGHT



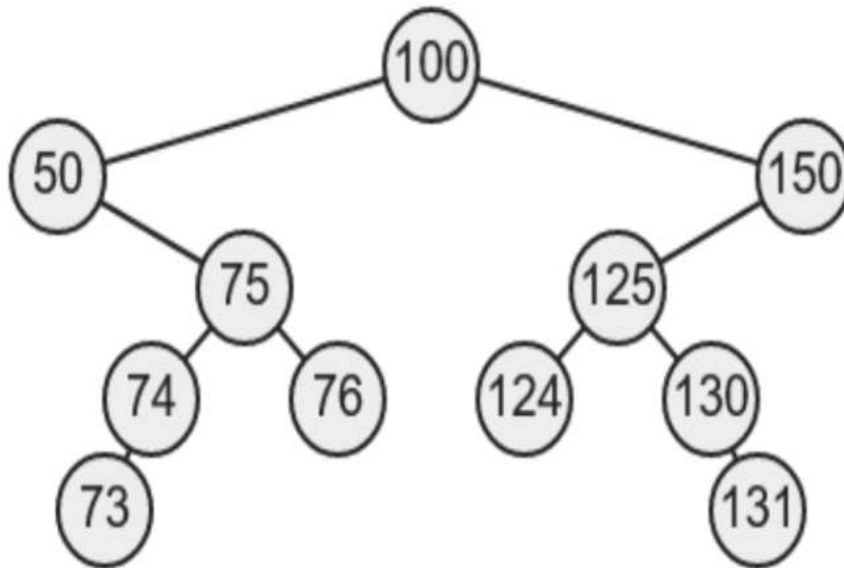
100, 50, 75, 74, 73, 76, 150, 125, 124, 130, 131

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

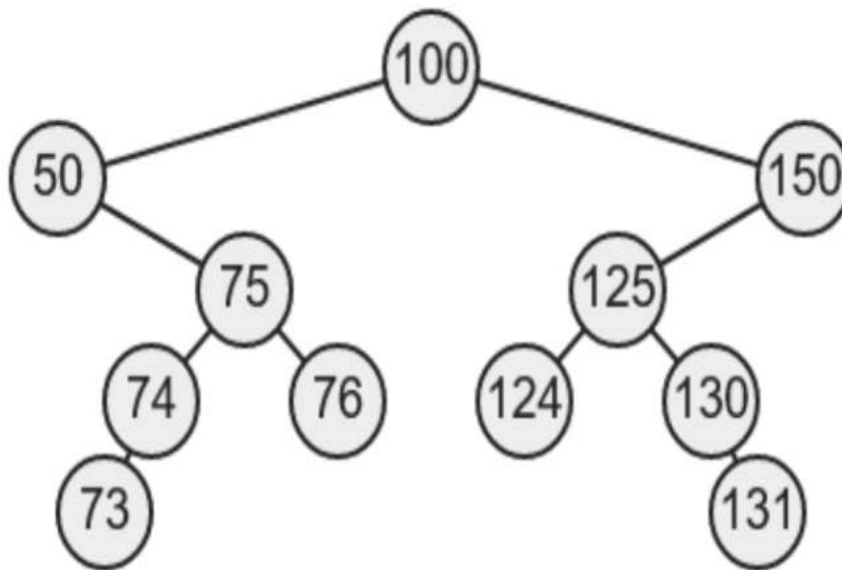
Kanak Kalyani

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

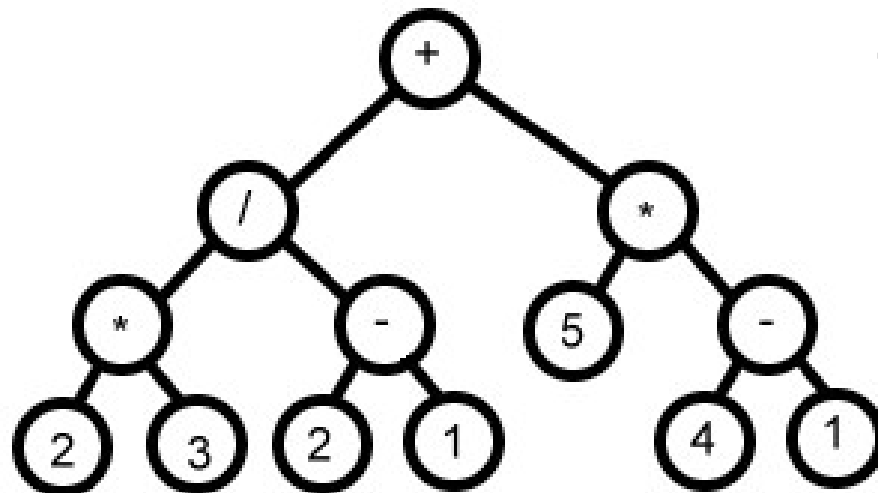
Kanak Kalyani

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 $2 * 3 / (2 - 1) + 5 * (4 - 1)$

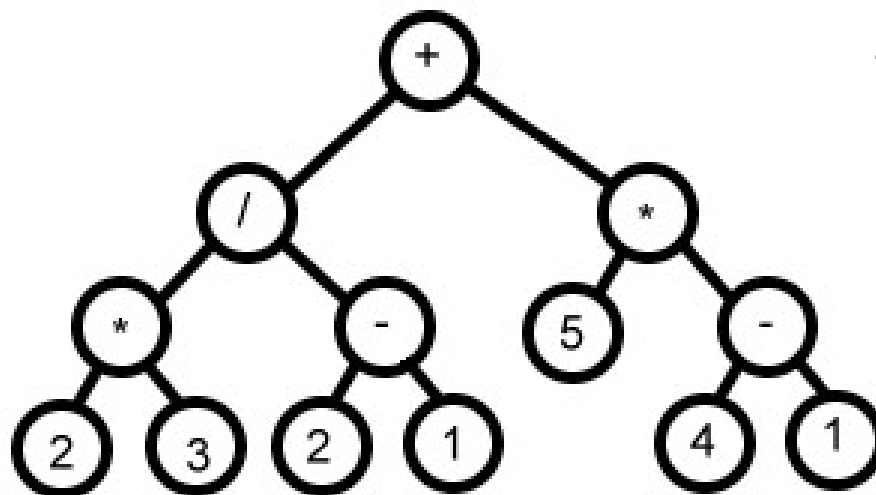
$+ / * 2 3 - 2 1 * 5 - 4 1$

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 $2 * 3 / (2 - 1) + 5 * (4 - 1)$

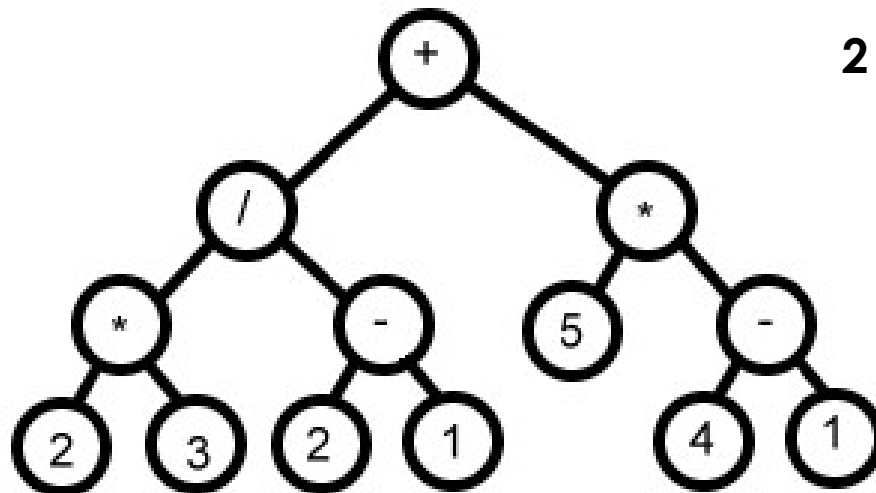
$2 * 3 / 2 - 1 + 5 * 4 - 1$

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

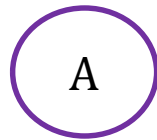
Construct a binary tree with the given traversal techniques

- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G

Constructing a Tree

Construct a binary tree with the given traversal techniques

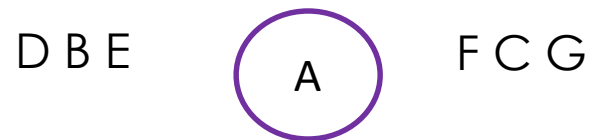
- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

Construct a binary tree with the given traversal techniques

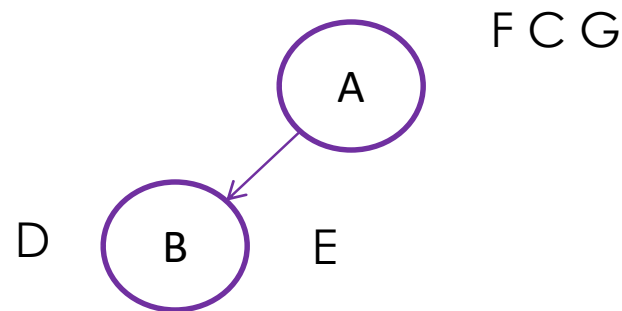
- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

Construct a binary tree with the given traversal techniques

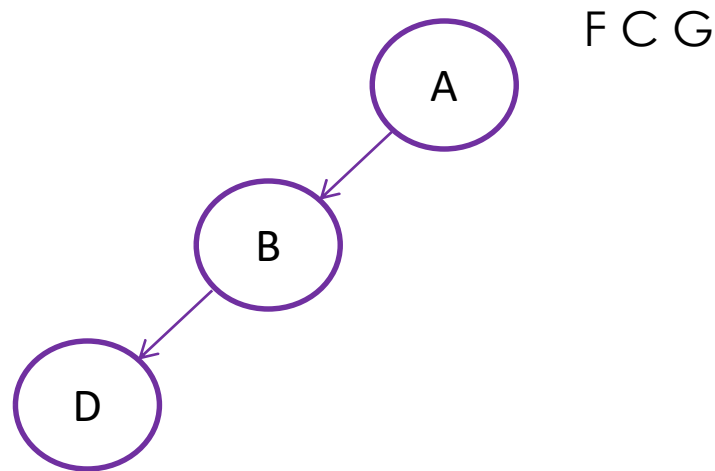
- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

Construct a binary tree with the given traversal techniques

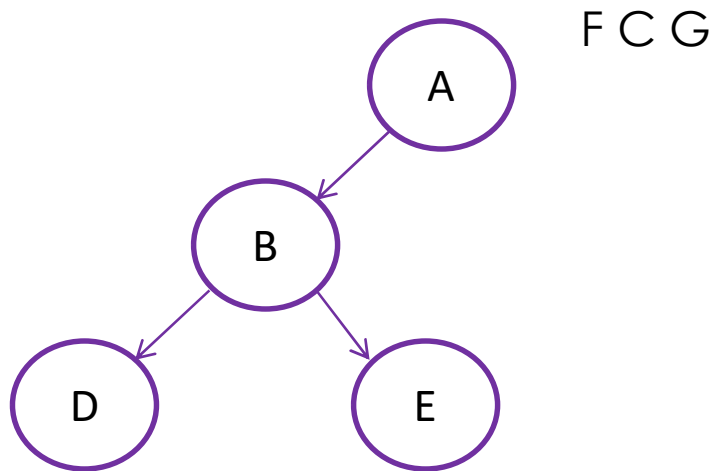
- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

Construct a binary tree with the given traversal techniques

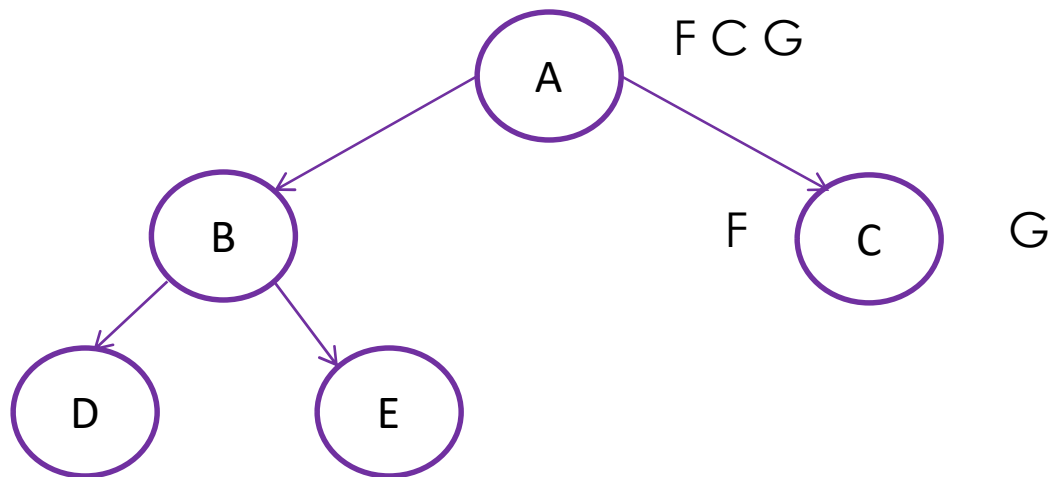
- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

Construct a binary tree with the given traversal techniques

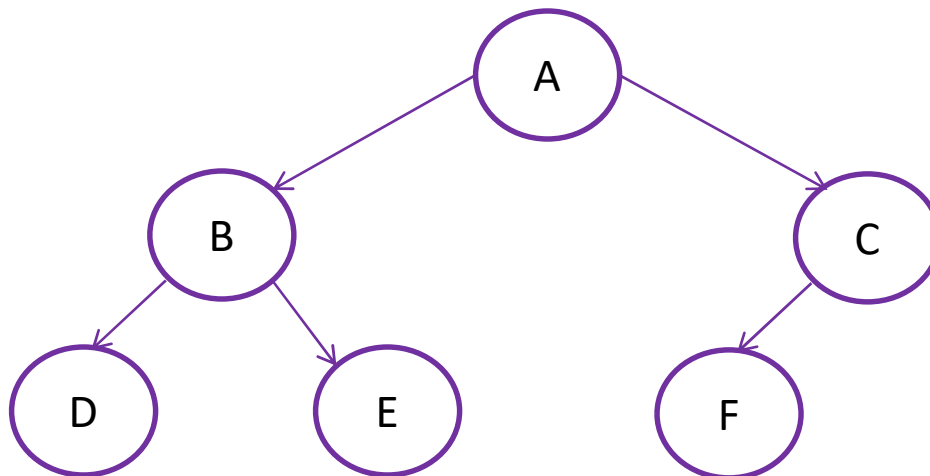
- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

Construct a binary tree with the given traversal techniques

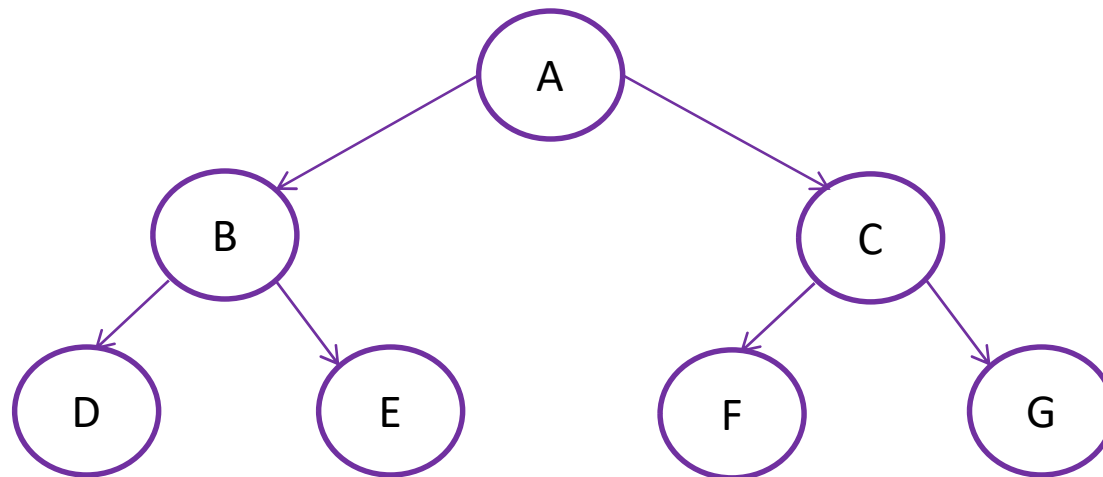
- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

Construct a binary tree with the given traversal techniques

- In-order Traversal: (L-Root-R): D B E A F C G
- Pre-order Traversal: (Root-L-R): A B D E C F G



Constructing a Tree

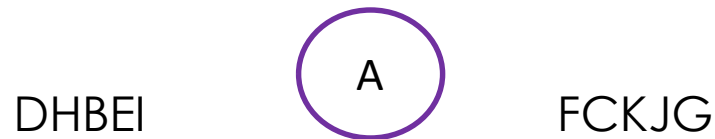
Construct a binary tree with the given traversal techniques

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

Constructing a Tree

Construct a binary tree with the given traversal techniques

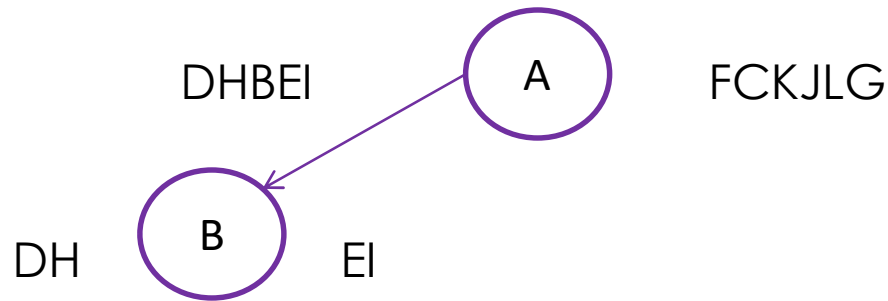
- In-order Traversal (L-Root-R): D H B E I A F C K J L G
- Post-order Traversal (L-R-Root): H D I E B F K L J G C A



Constructing a Tree

Construct a binary tree with the given traversal techniques

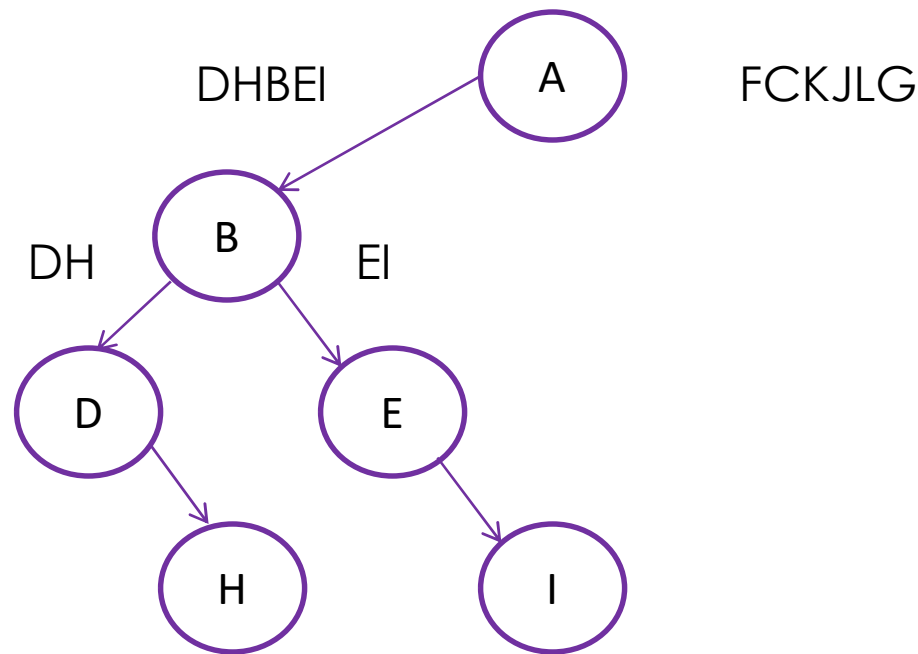
- In-order Traversal (L-Root-R): D H B E I A F C K J L G
- Post-order Traversal (L-R-Root): H D I E B F K L J G C A



Constructing a Tree

Construct a binary tree with the given traversal techniques

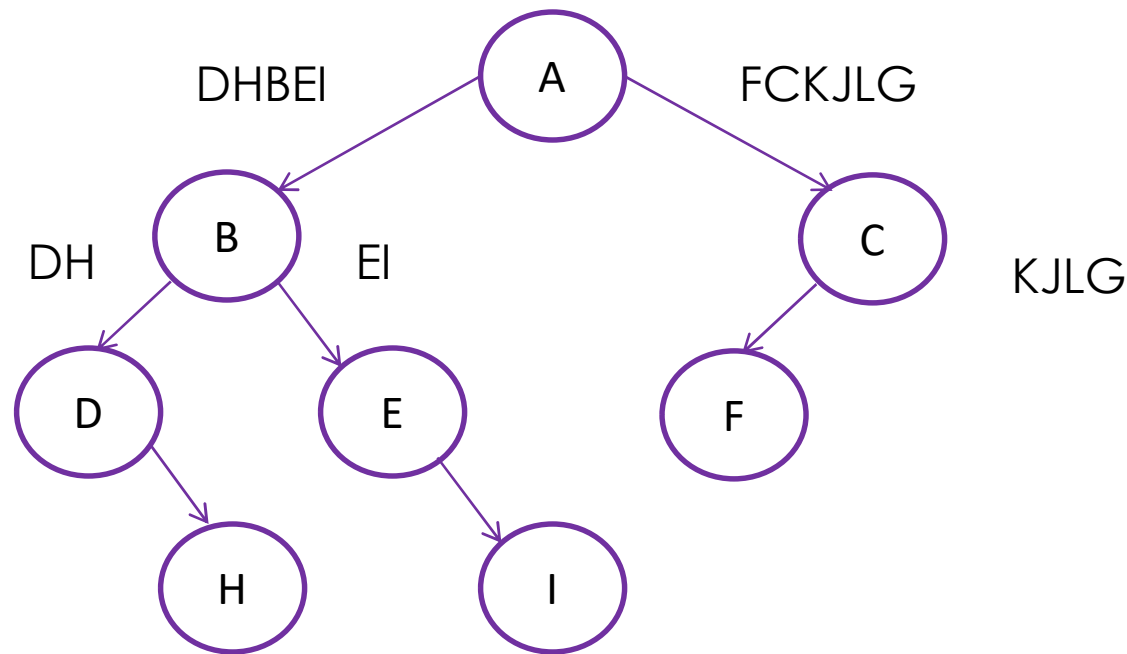
- In-order Traversal (L-Root-R): D H B E I A F C K J L G
- Post-order Traversal (L-R-Root): H D I E B F K L J G C A



Constructing a Tree

Construct a binary tree with the given traversal techniques

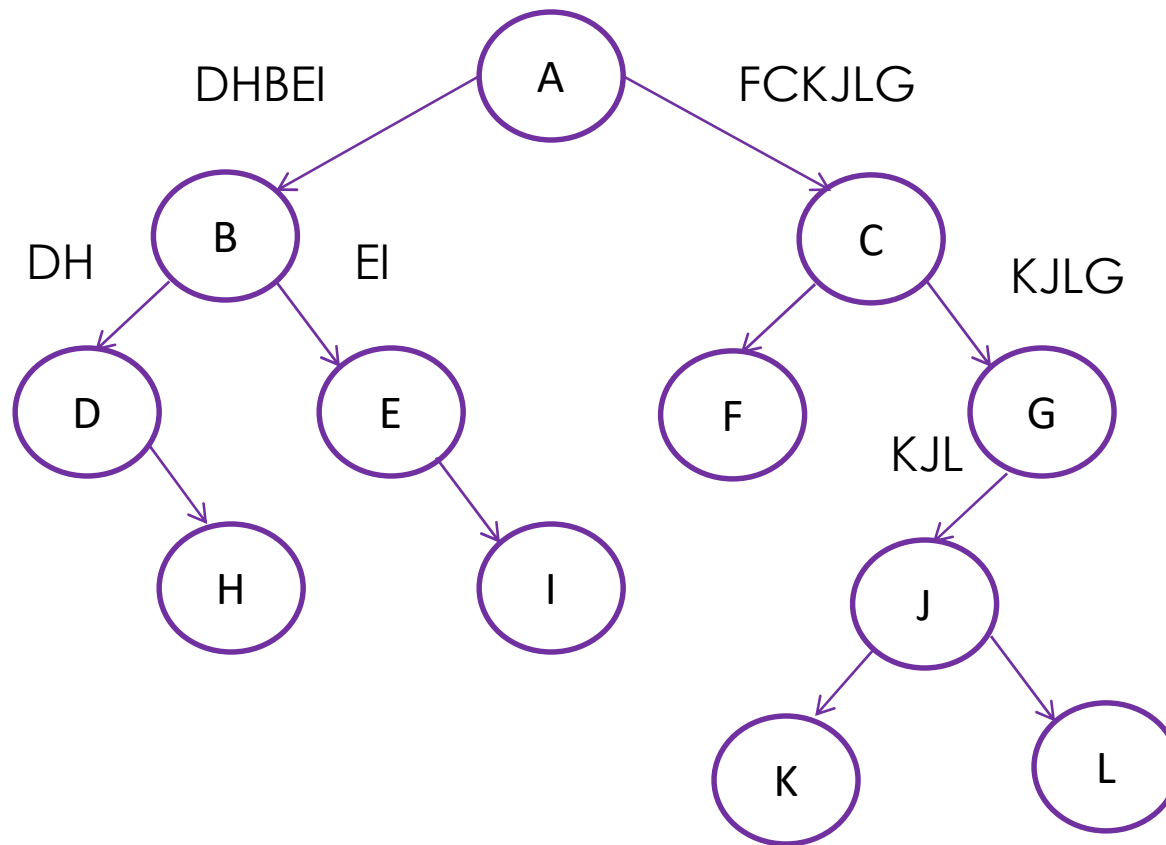
- In-order Traversal (L-Root-R): D H B E I A F C K J L G
- Post-order Traversal (L-R-Root): H D I E B F K L J G C A



Constructing a Tree

Construct a binary tree with the given traversal techniques

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

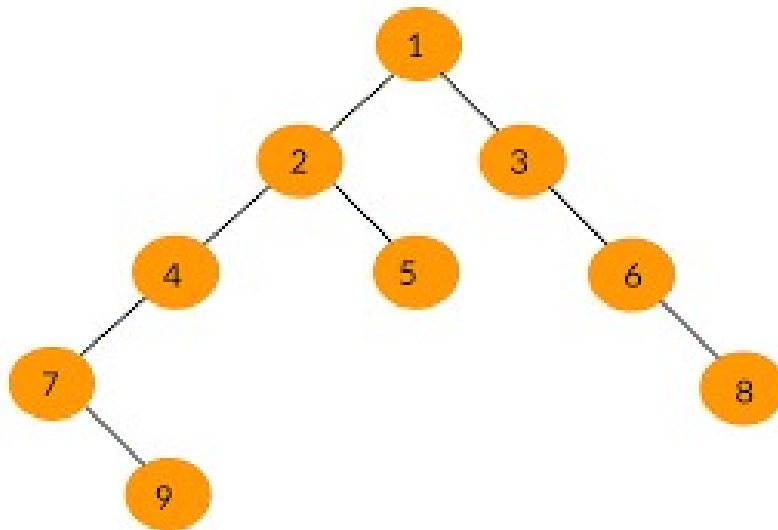


Constructing a Tree

Construct a binary tree with the given traversal techniques

► In-order Traversal: (L-Root-R): **7 9 4 2 5 1 3 6 8**

► Post-order Traversal: (L-R-Root) **9 7 4 5 2 8 6 3 1**



Constructing a Tree

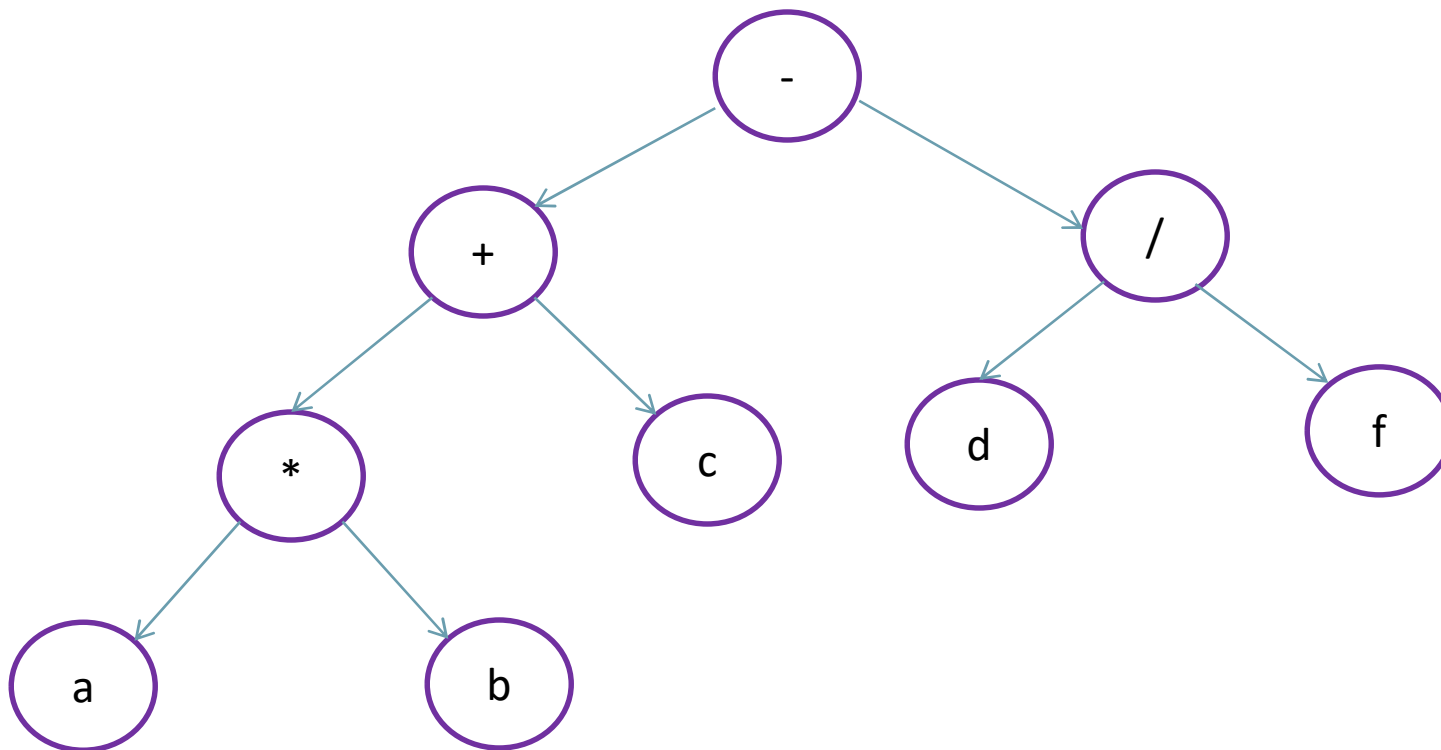
Building a Binary Expression tree from a Prefix expression

- 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



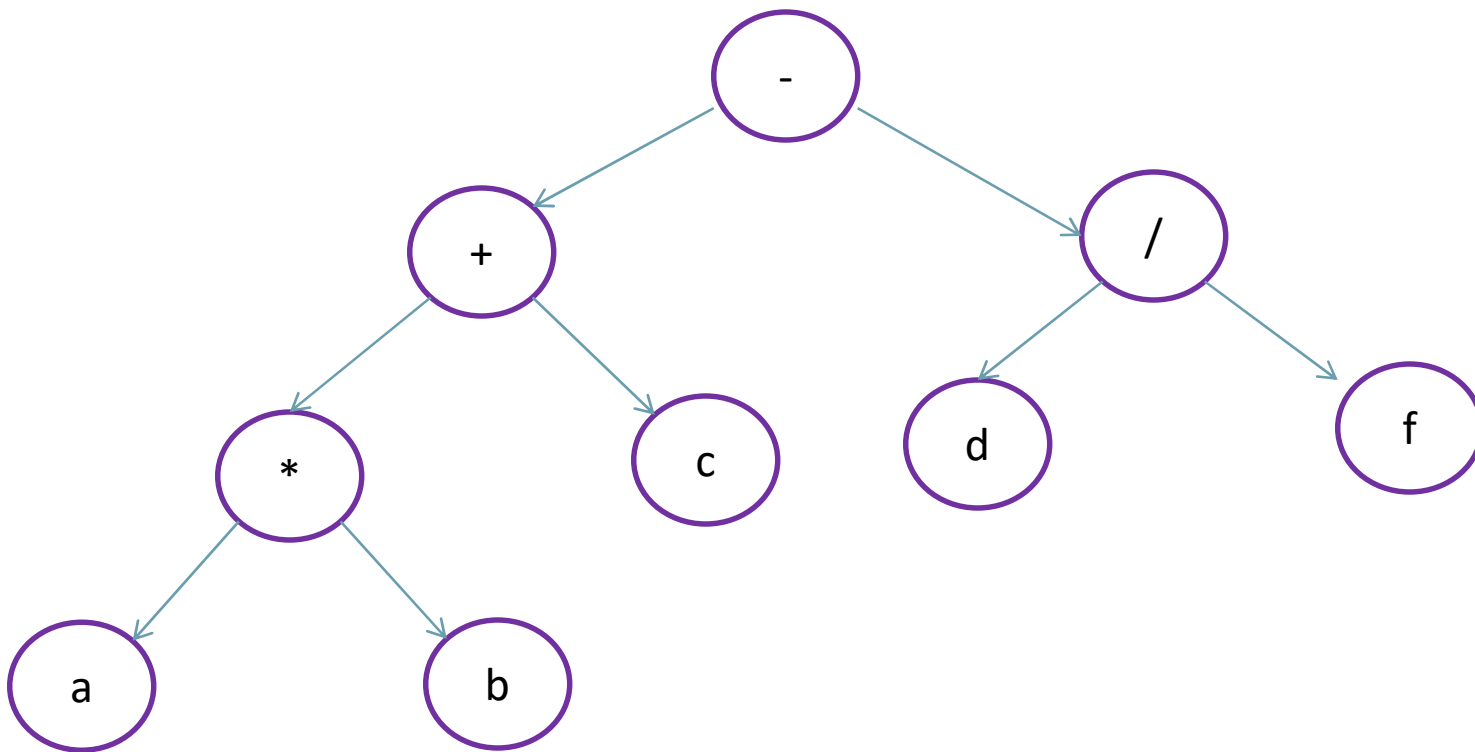
Constructing a Tree

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

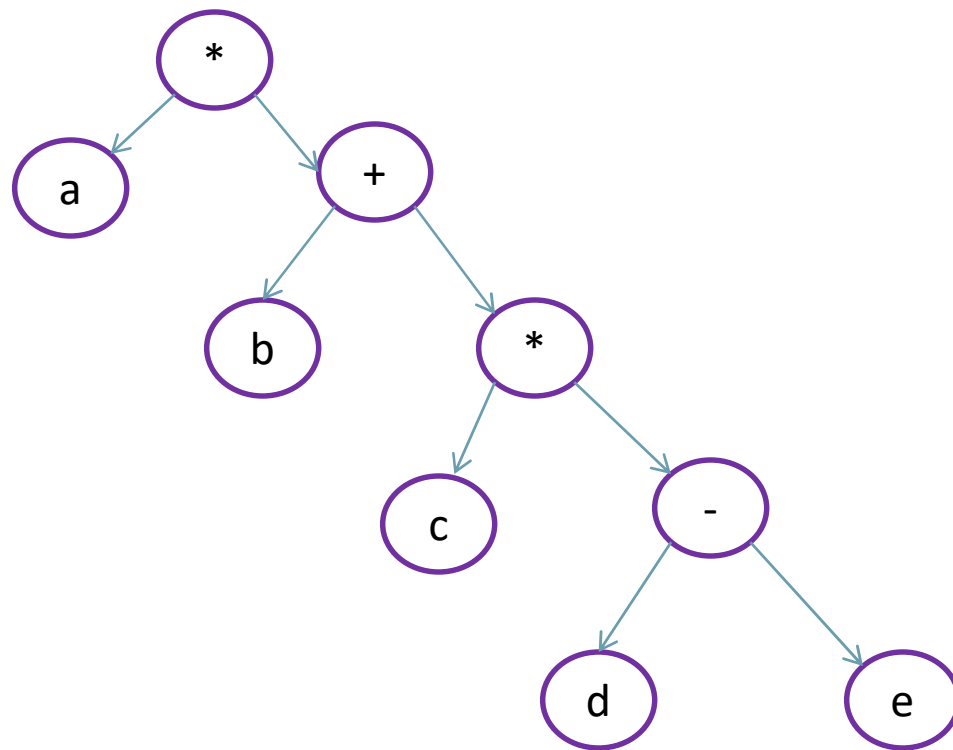
Given the Postfix Expression Construct a Expression Tree
Postfix Expression: (L-R-Root) : a b * c + d f / -



Constructing a Tree

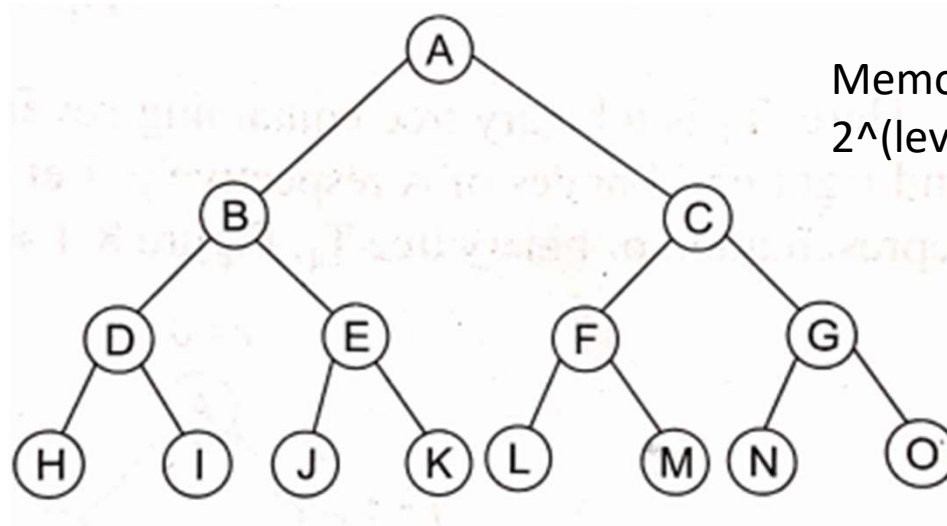
- Given the Prefix Expression Construct a Expression Tree

Prefix Expression: * a + b * c - d e



Binary Tree Representation

Array Representation of Tree

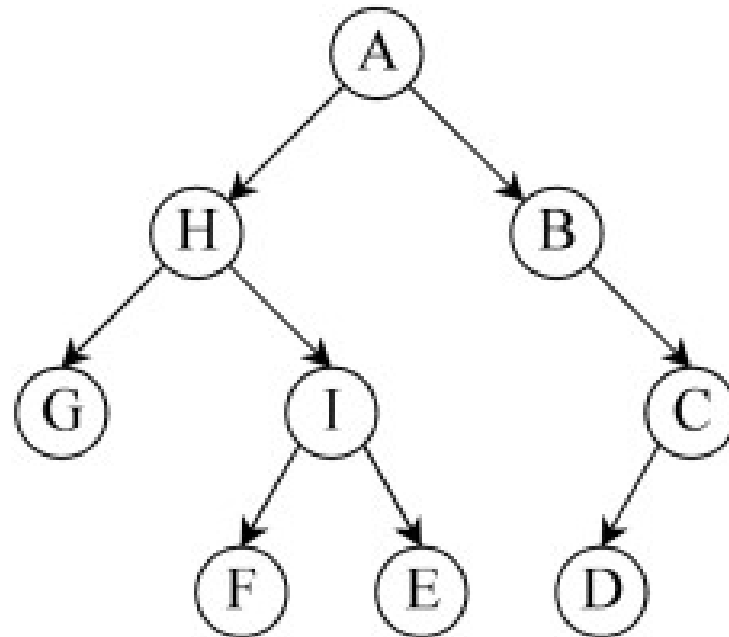


Memory allocated of array:
 $2^{(\text{levels}+1)} - 1$

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

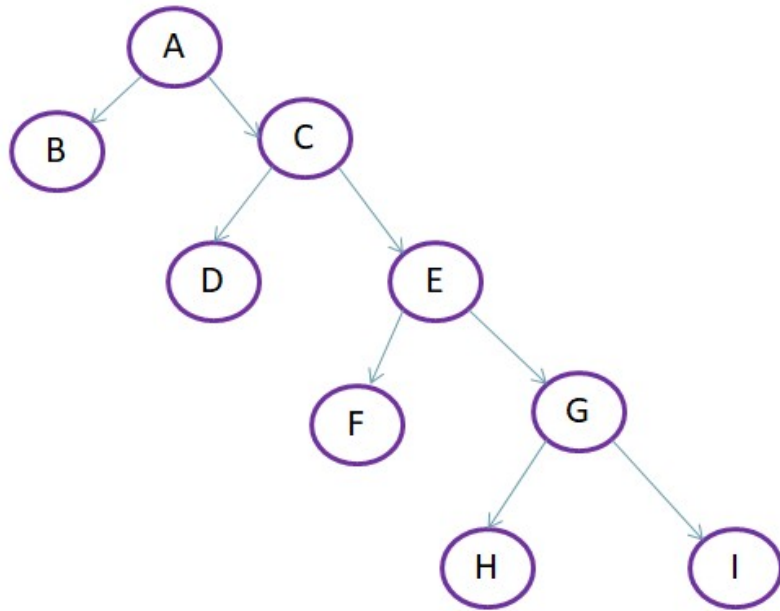
- Notice:
 - The left child of index i is at index $2*i+1$
 - The right child of index i is at index $2*i+2$

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 of array:
 $2^{(\text{level}+1)} - 1$

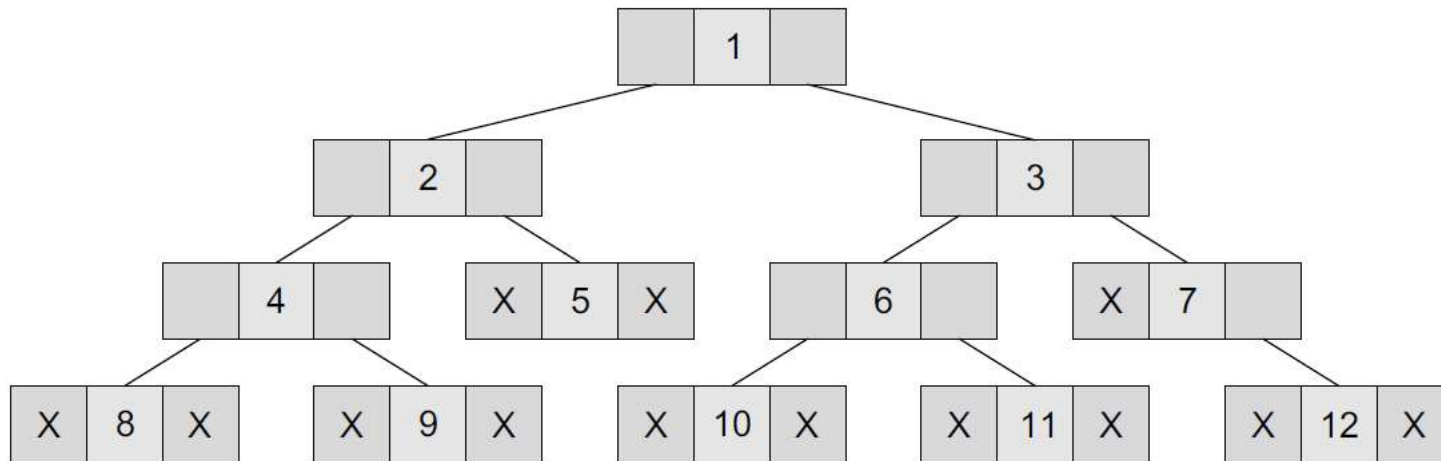
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	B	C			D	E							F	G				

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

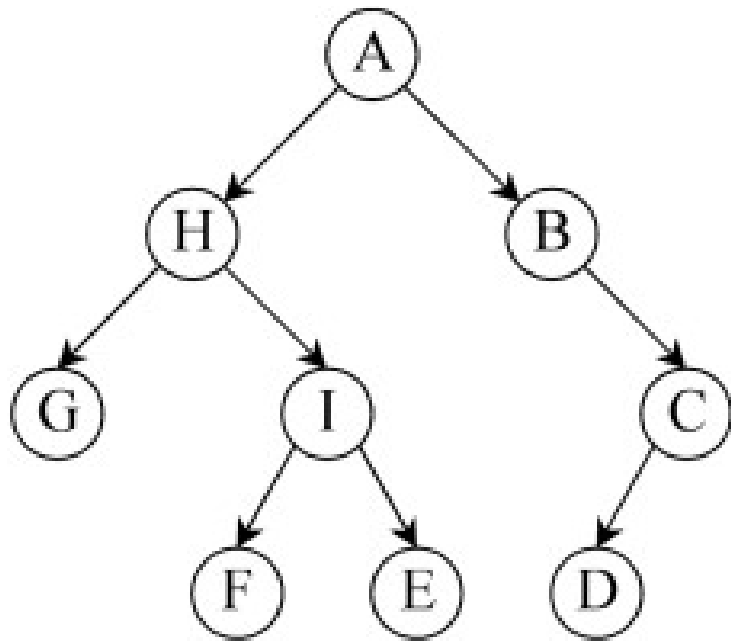
Kanak Kalyani

Linked representation of binary trees

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

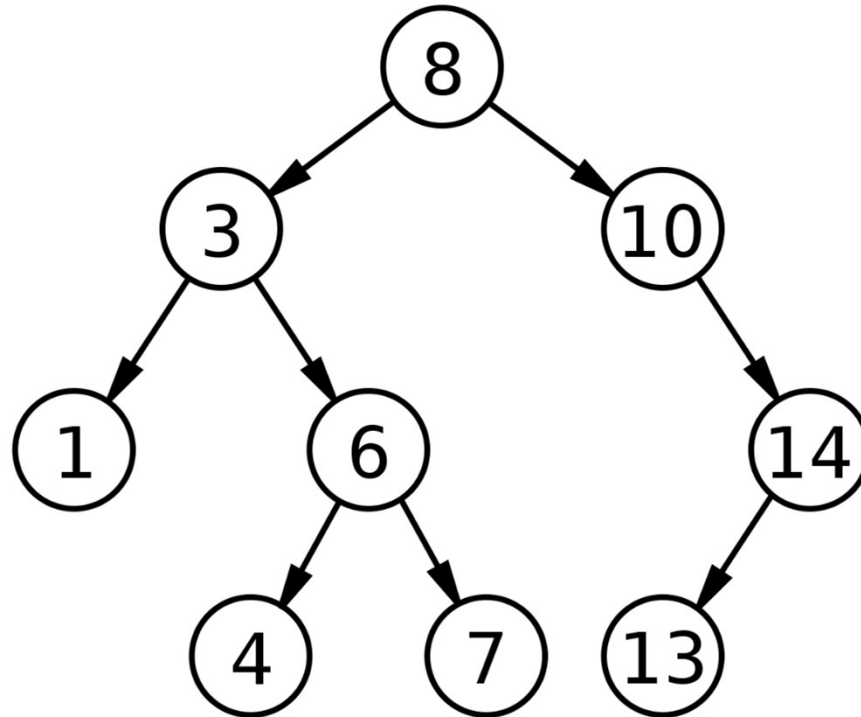


Linked Tree Representation of Binary Tree: Static Allocation



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			

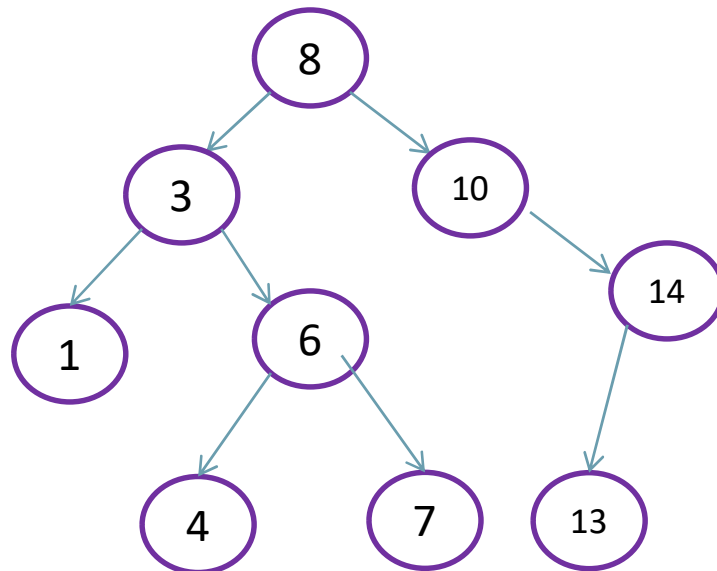
Binary Search Tree (BST)



The left subtree of a node contains nodes with keys lesser or equal 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.

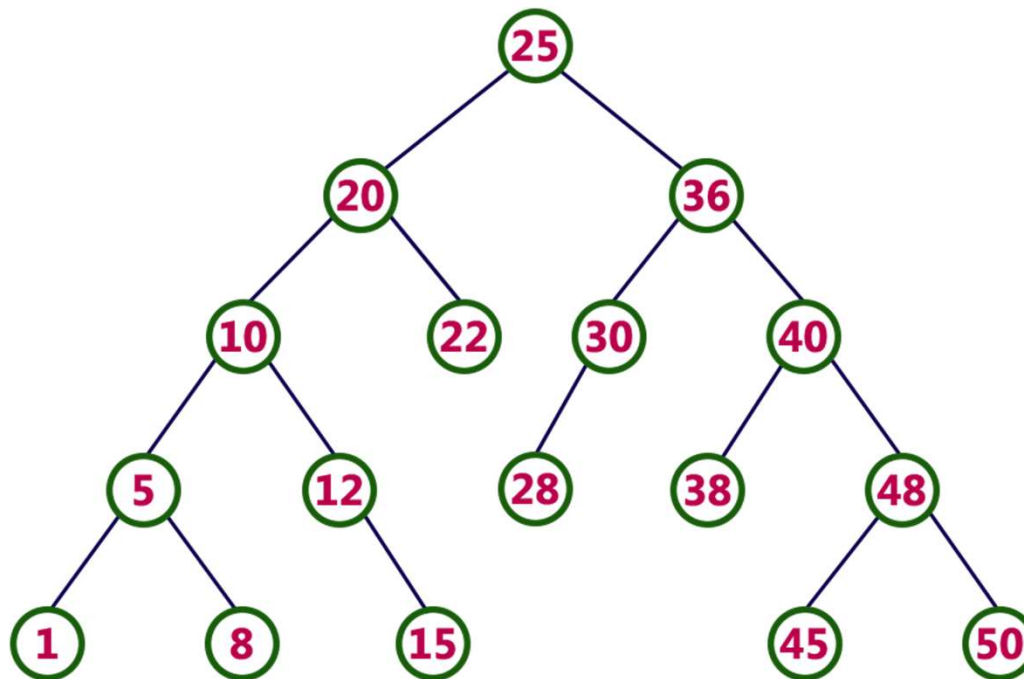
Creation of BST

- 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



Do it yourself

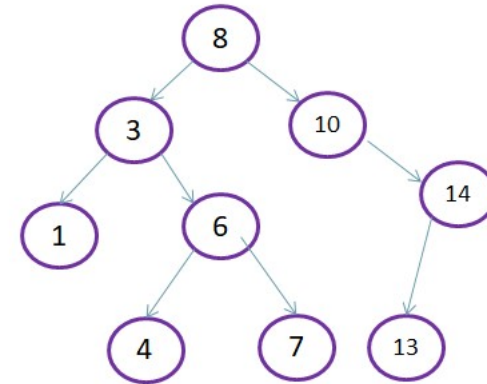
- Create a Binary Search Tree with the following node values
1) 45, 20, 10, 50, 80, 60, 22, 58, 74, 12, 56,
2) 50,45,40,30,25,20,15,10

BST: Node Declaration

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

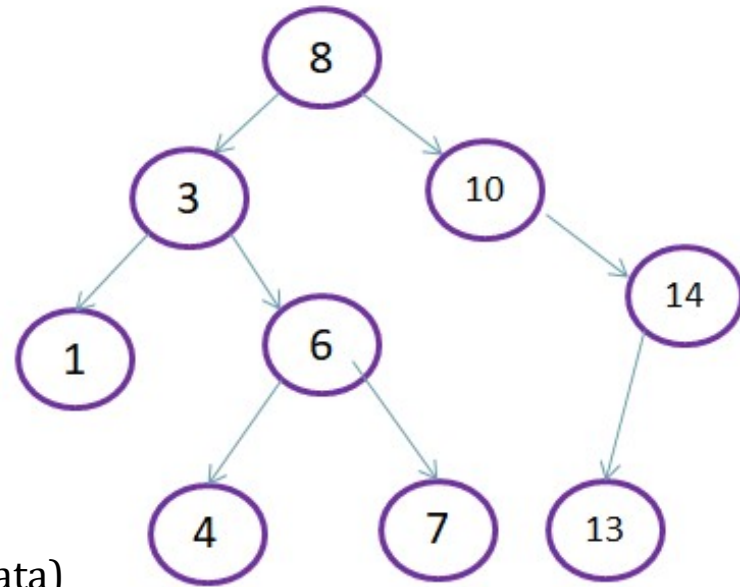
BST: Create()

```
treenode* create(){  
    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()

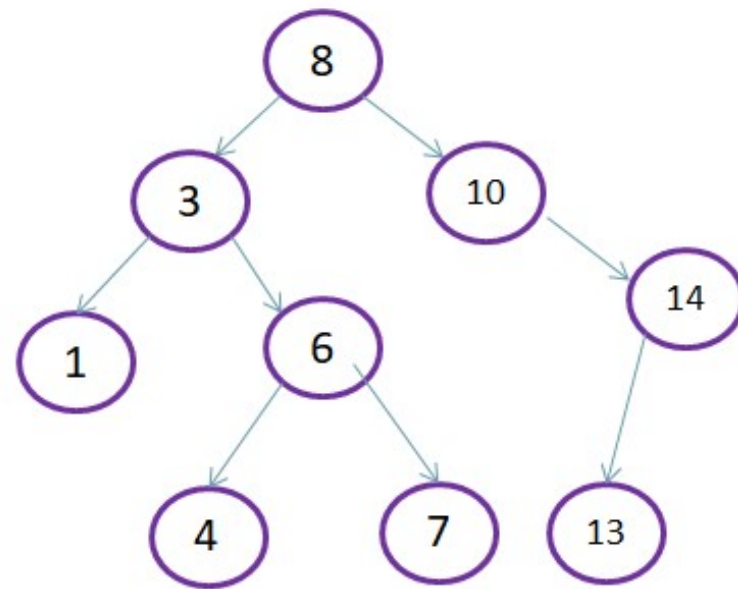
```
if(root==NULL)
    root=temp;
else{
    ptr=root;
    while(ptr!=NULL){
        prev=ptr;
        if(ptr->data < temp->data)
            ptr=ptr->right;
        else
            ptr=ptr->left;
    }
    if(prev->data < temp->data)
        prev->right=temp;
    else
        prev->left=temp;
    }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



BST: Inorder Traversal

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

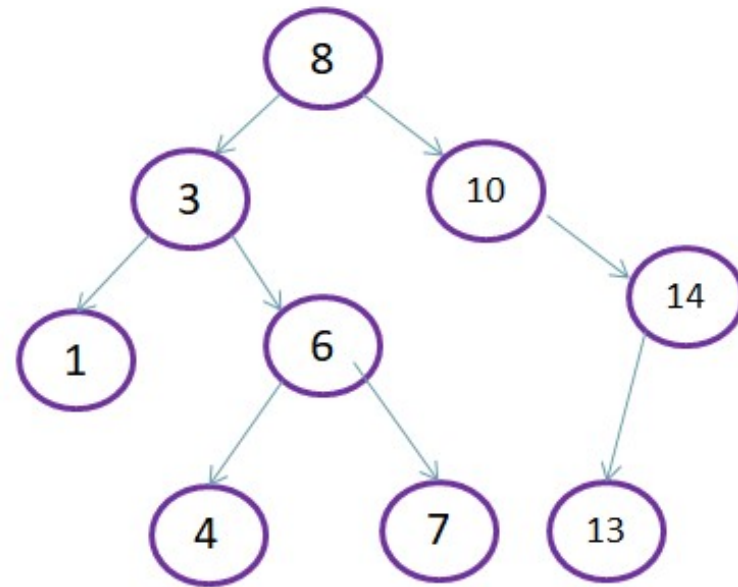
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}
```



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 printPostorder(treenode* node)
{
    if (node == NULL)
        return;

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

BINARY TREE DELETION

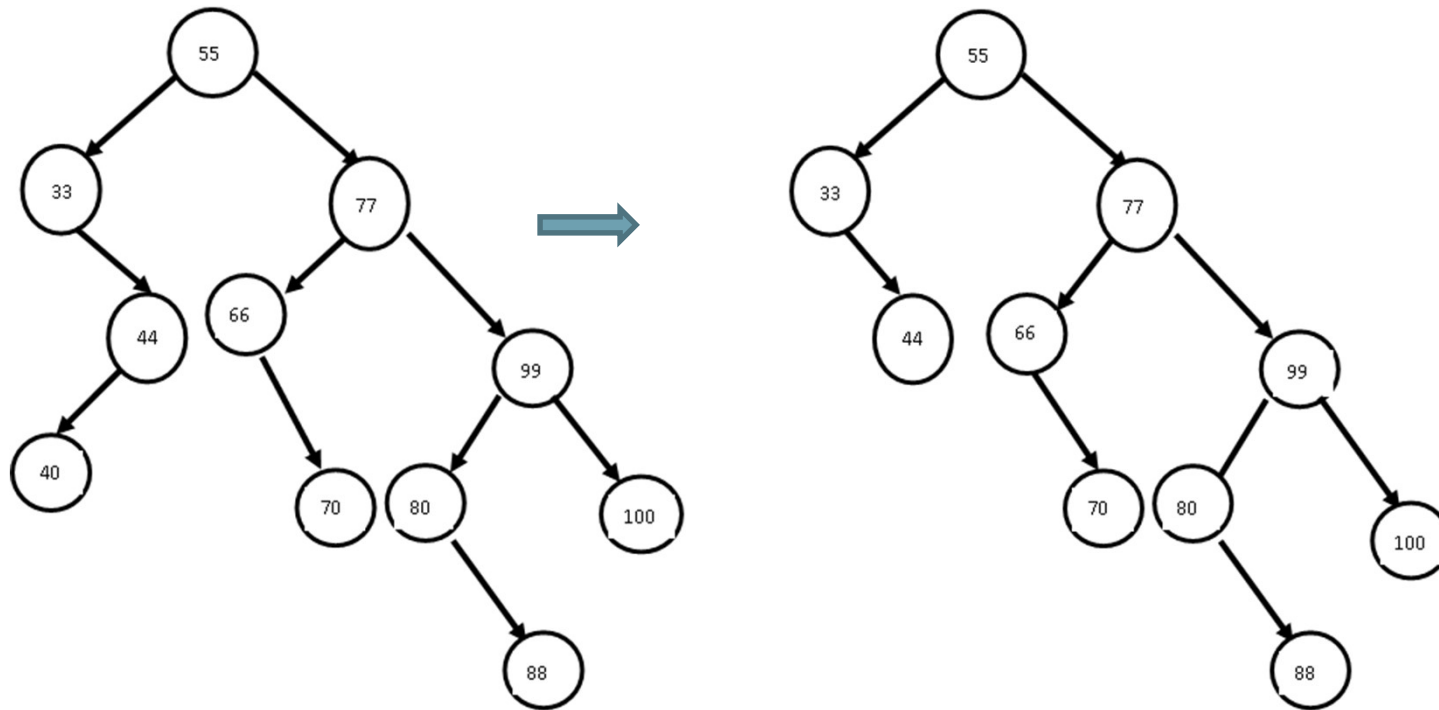
BINARY SEARCH TREE DELETION

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.

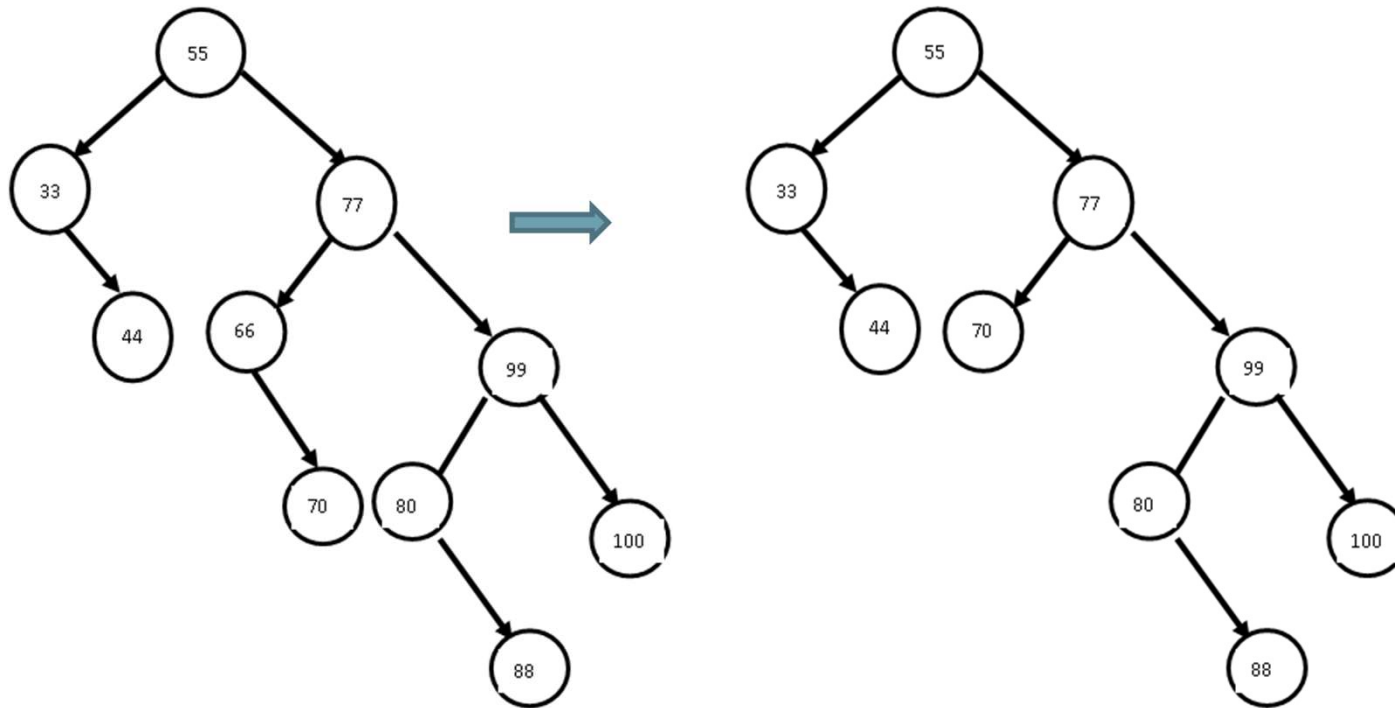
BINARY SEARCH TREE DELETION

➤ Delete 40



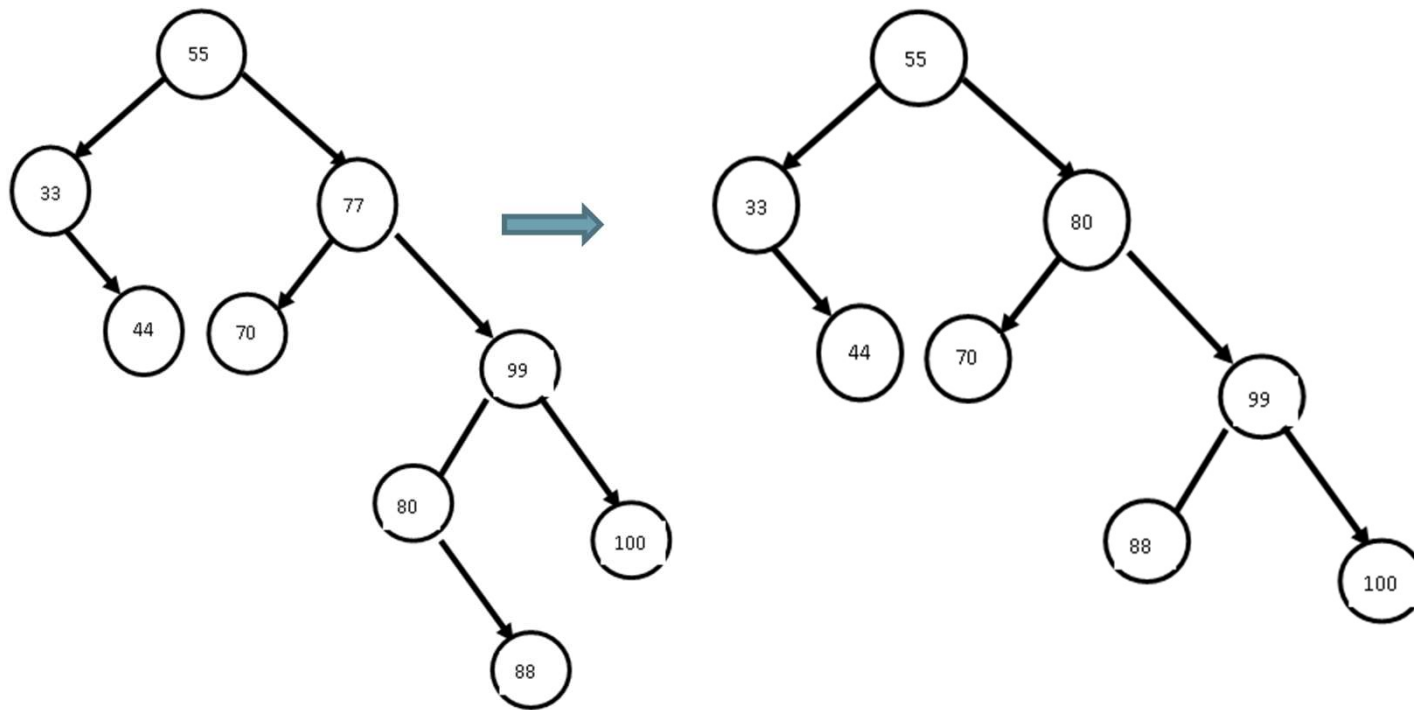
BINARY SEARCH TREE DELETION

➤ Delete 66



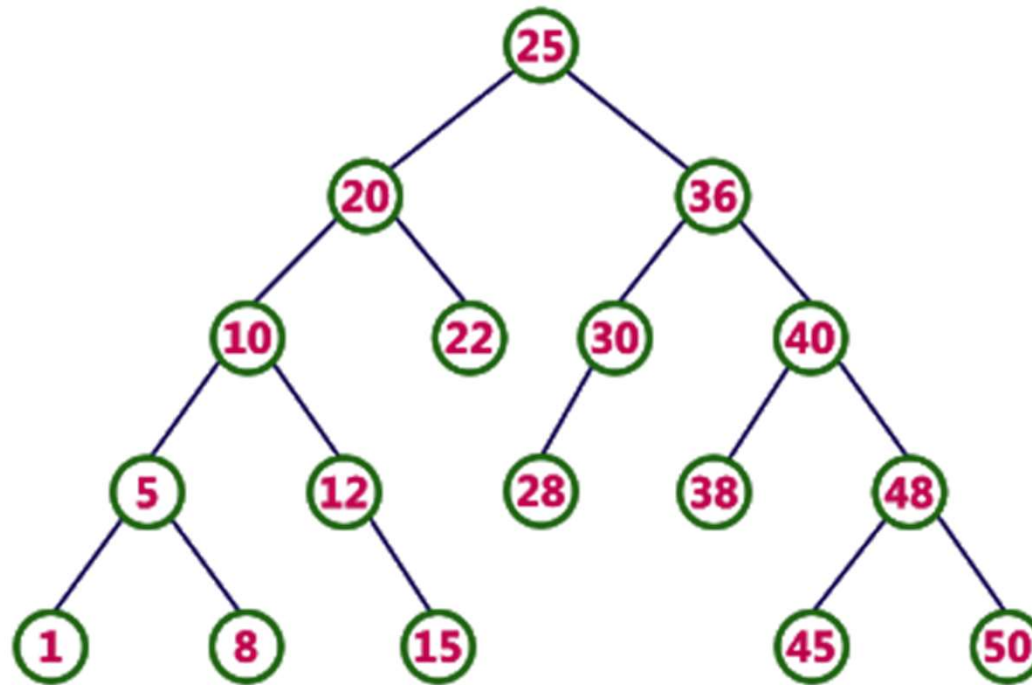
BINARY SEARCH TREE DELETION

➤ Delete 77



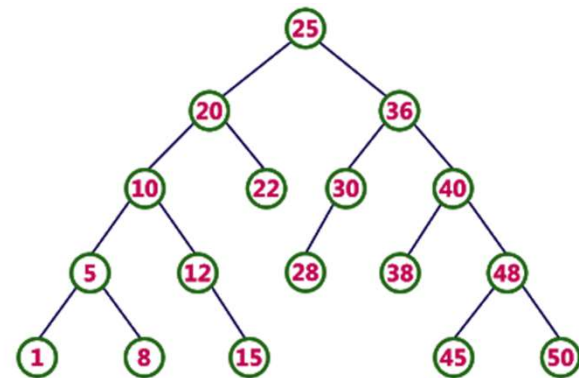
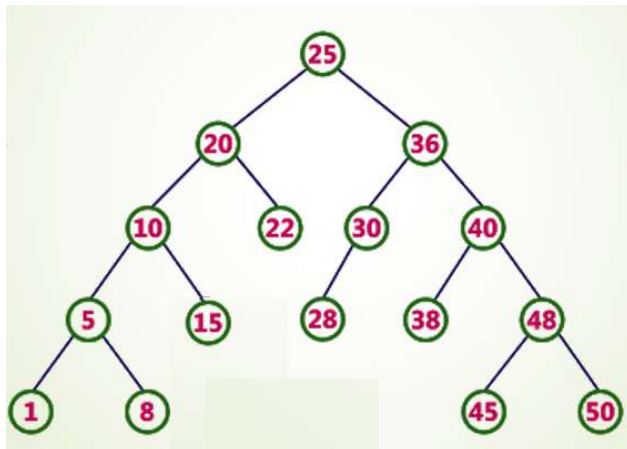
Example...

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



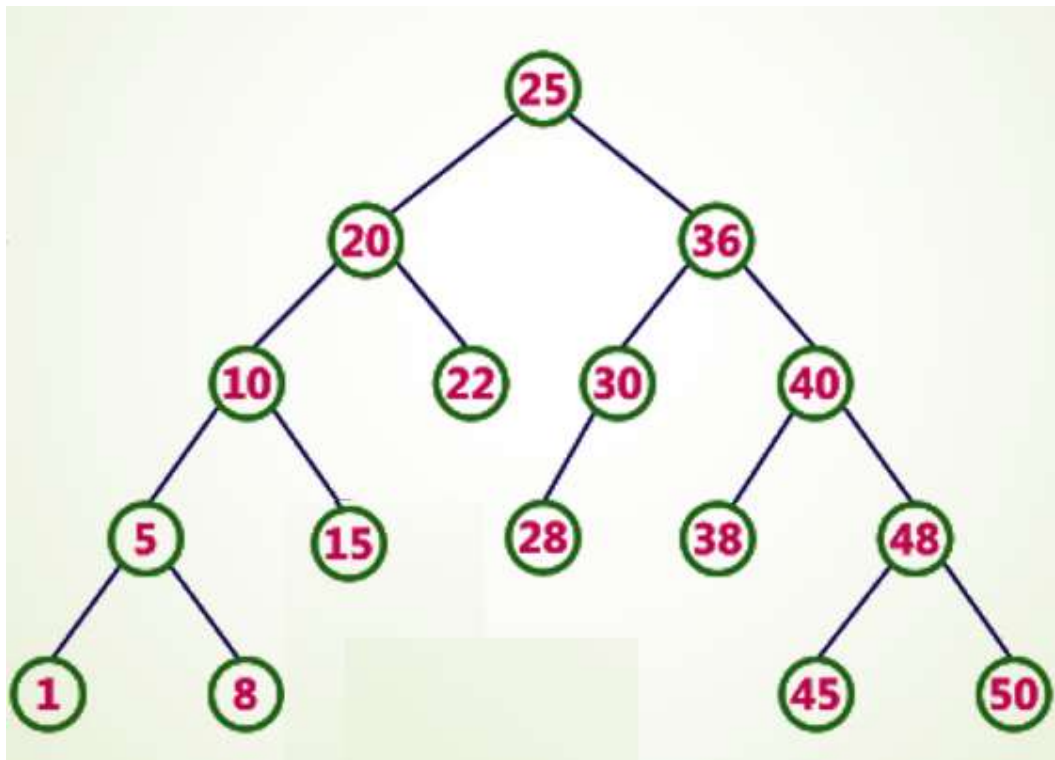
Example...

- Delete node 12



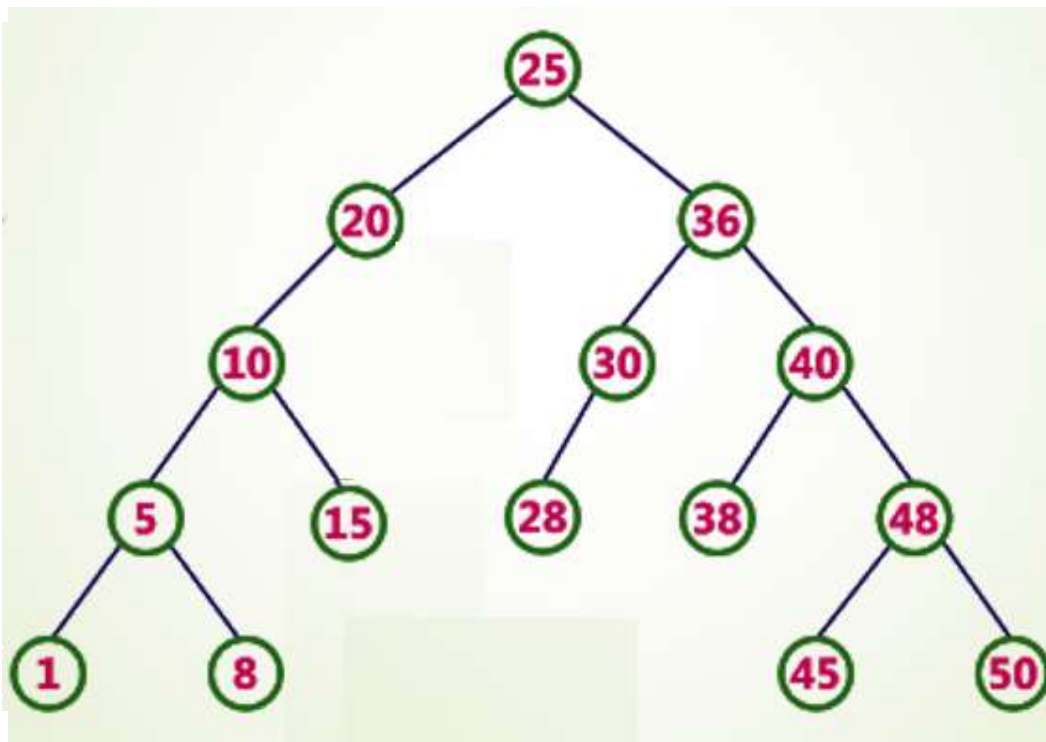
Example...

- Delete node 22



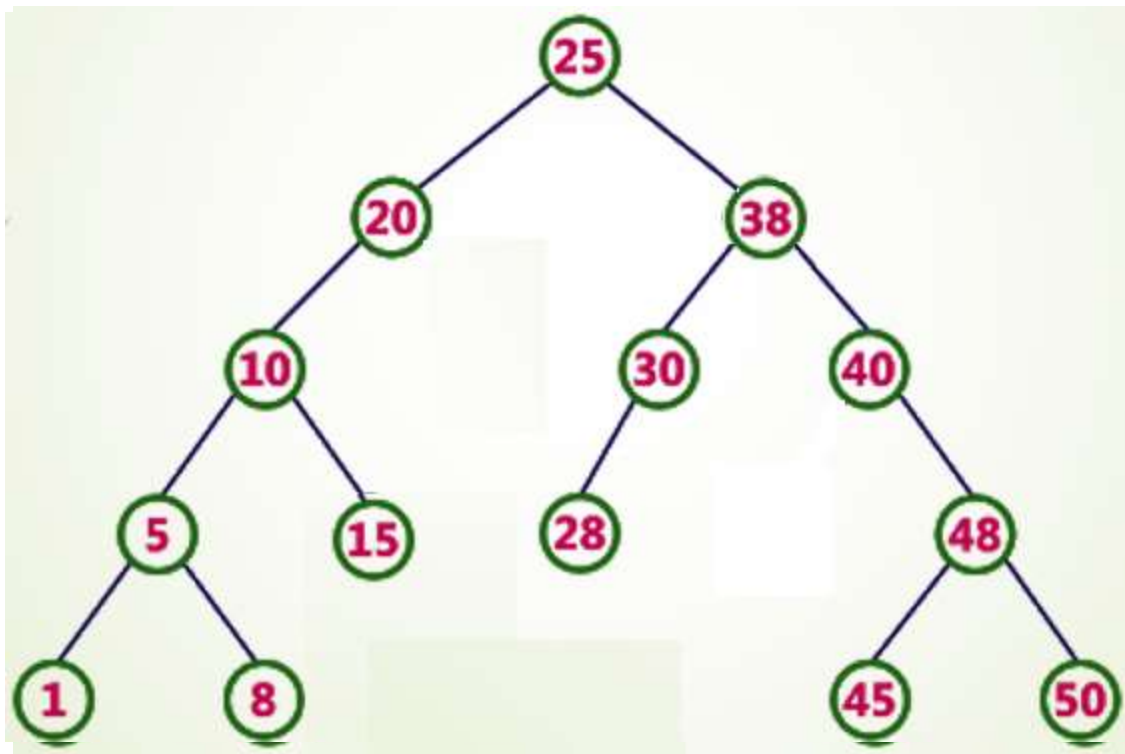
Example...

- Delete node 36

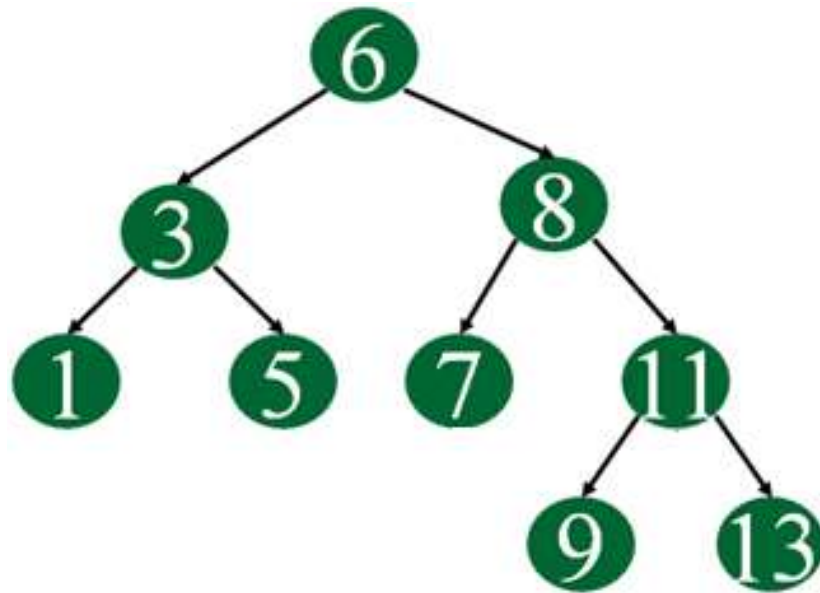


Example...

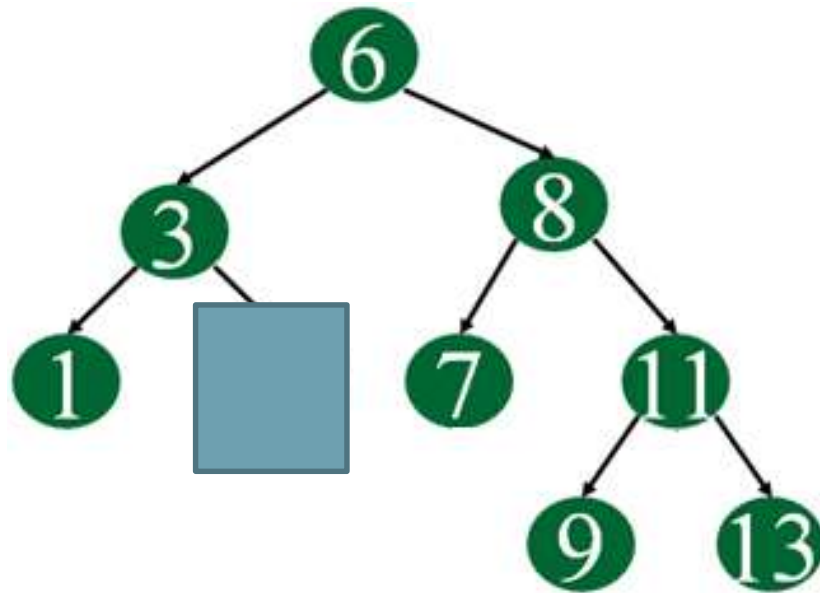
- Delete node 25



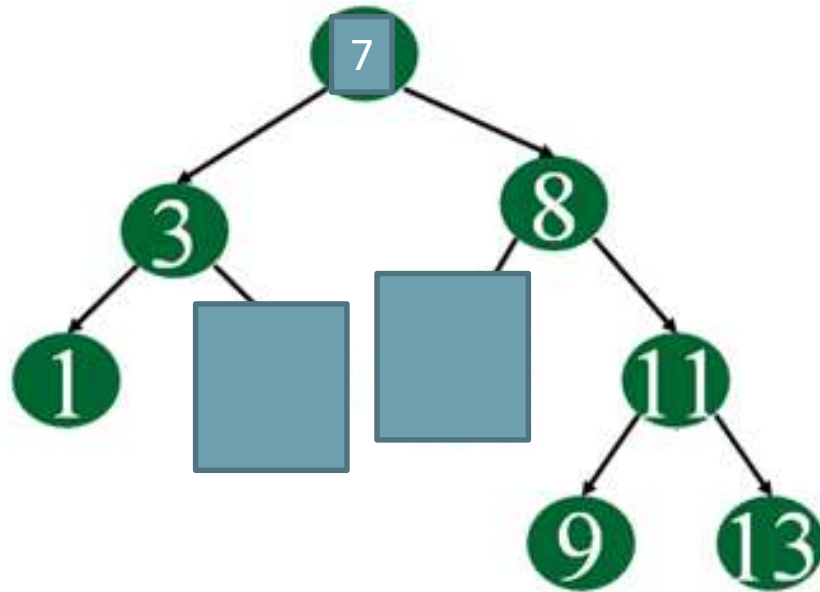
Delete 5,6,8,11



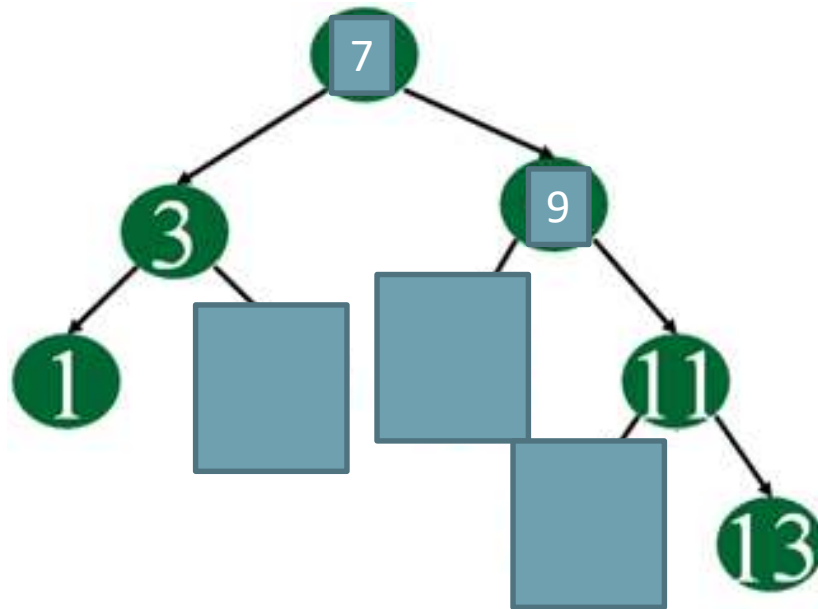
Delete 5



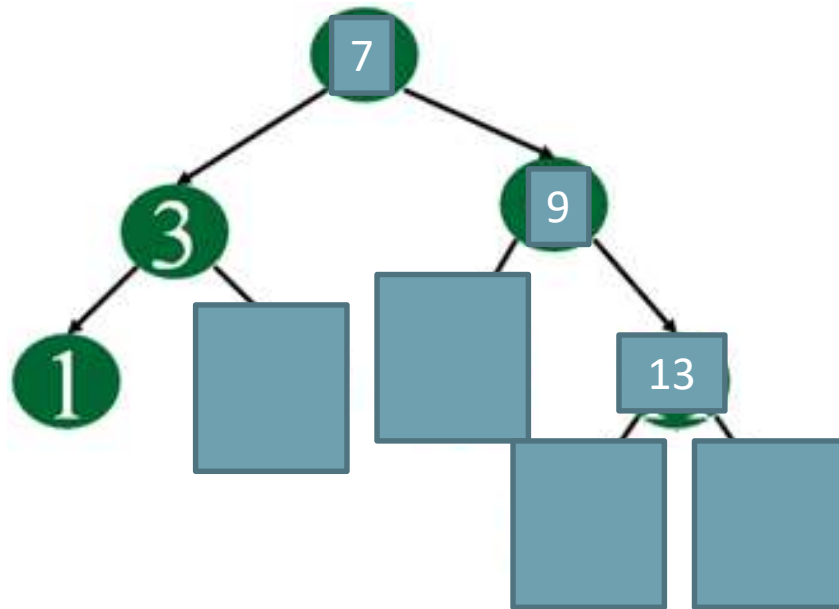
Delete 6



Delete 8



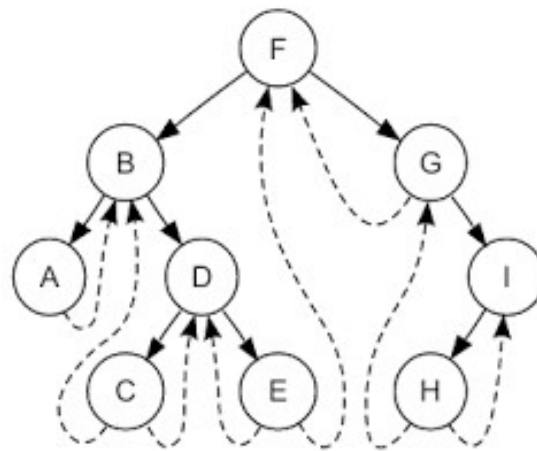
Delete 11



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

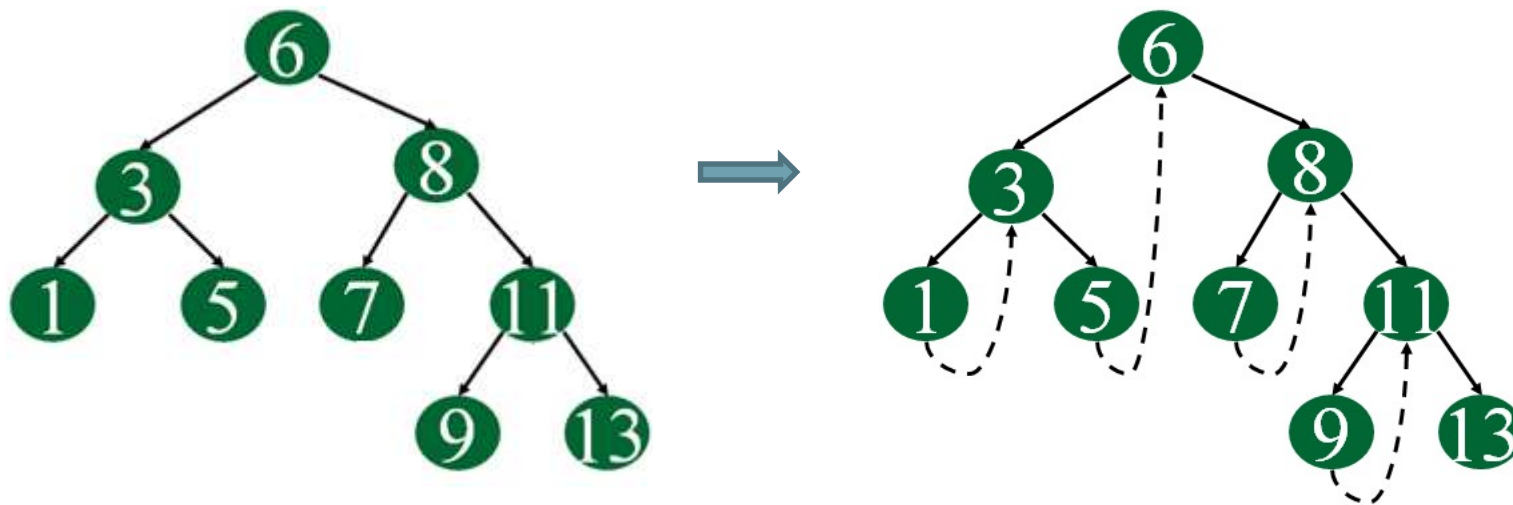


INORDER THREADED BINARY TREES

- 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;
```

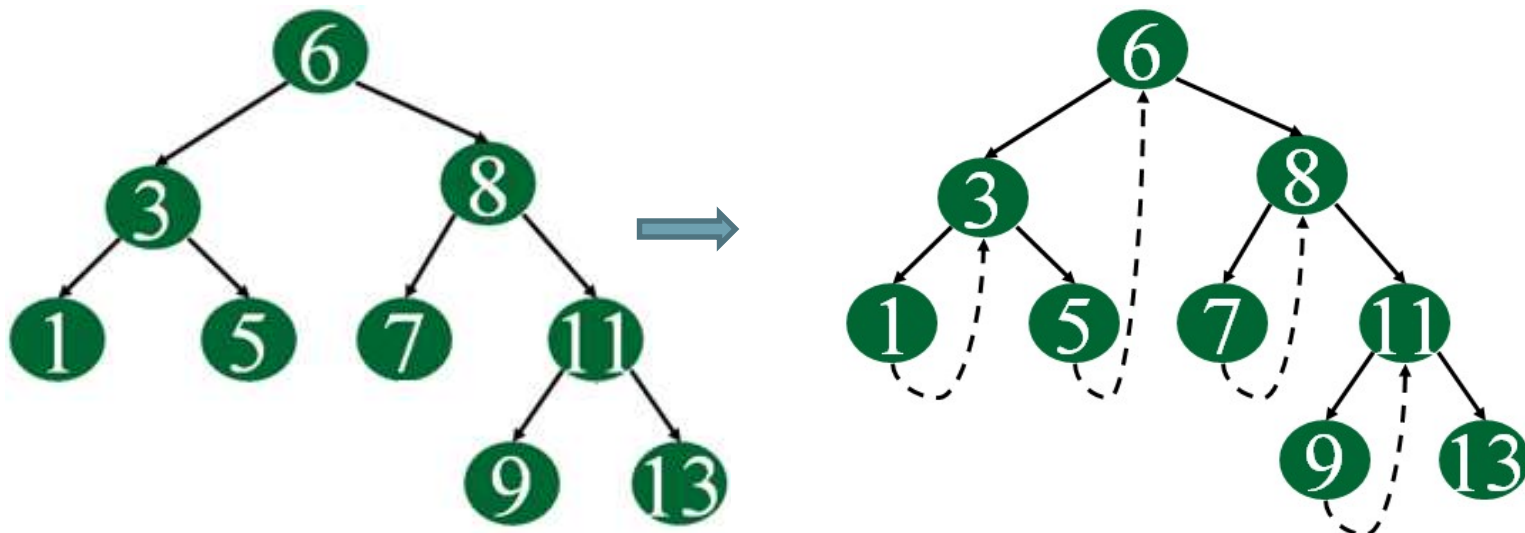
```
    Node * right;
```

```
    bool rightThread;
```

```
}
```

left	data	right	rightThread
------	------	-------	-------------

left	data	right	0/1
------	------	-------	-----



Kanak Kalyani

DOUBLE THREADED BINARY TREES

```
struct Node
```

```
{
```

```
    bool leftThread;
```

```
    Node *left;
```

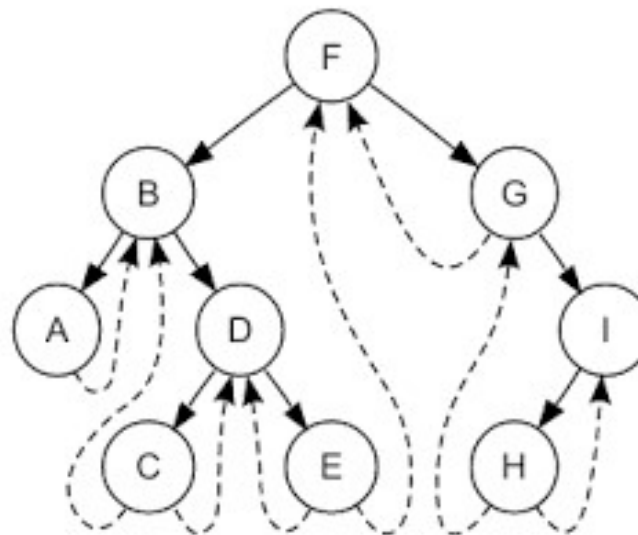
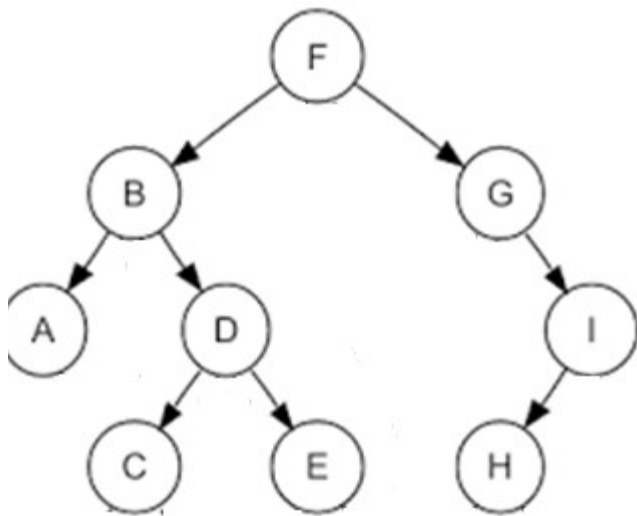
```
    int data;
```

```
    Node *right;
```

```
    bool rightThread;
```

```
}
```

Left Thread	left	data	right	Right Thread
-------------	------	------	-------	--------------

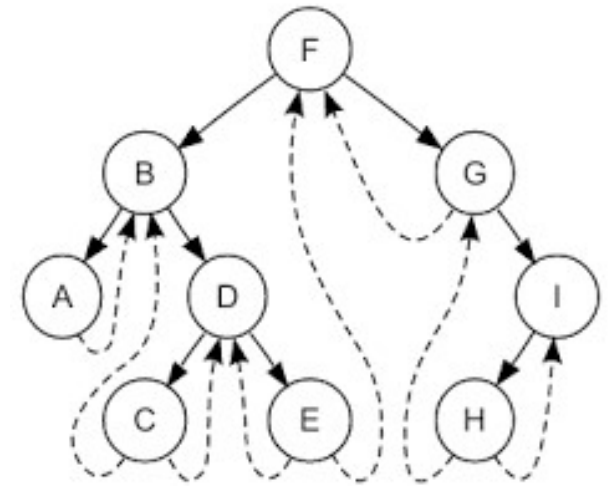


Program for inorder traversal of threaded binary tree

```
struct Node* leftMost(struct Node *node)
{
    if (node == NULL)
        return NULL;
    while (node ->left != NULL)
        node = node ->left;
    return node;
}
```

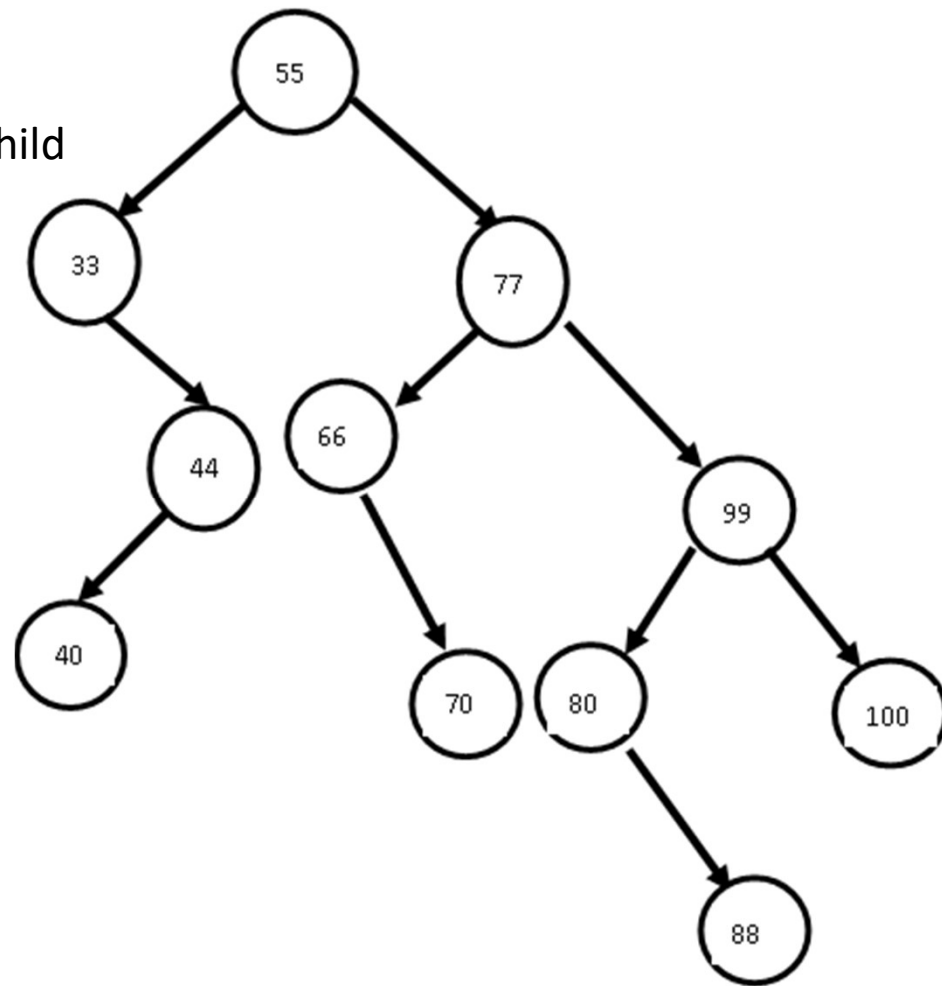

Program for inorder traversal of threaded binary tree

```
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);
    }
}
```



Trees

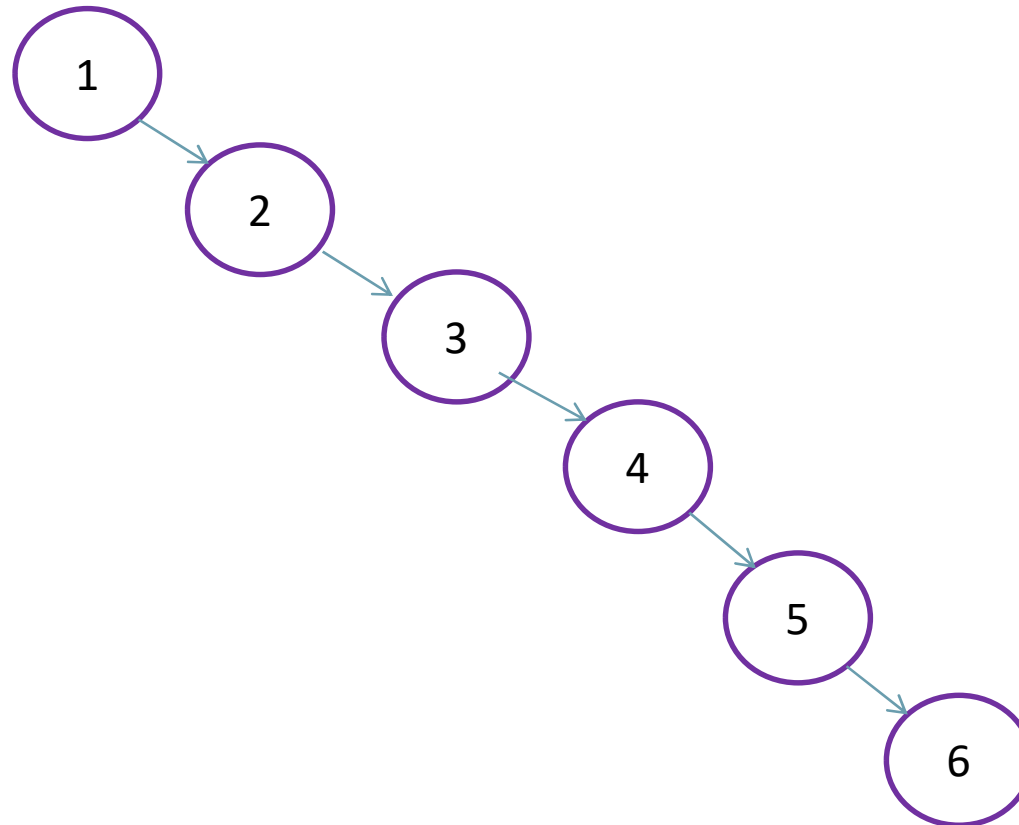
- 1) Count number of nodes in the tree
- 2) Count number of leaf nodes
- 3) Count number of nonleaf nodes
- 4) Count number of nodes with 1 child



Creation of BST

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

Skewed BST



AVL TREES



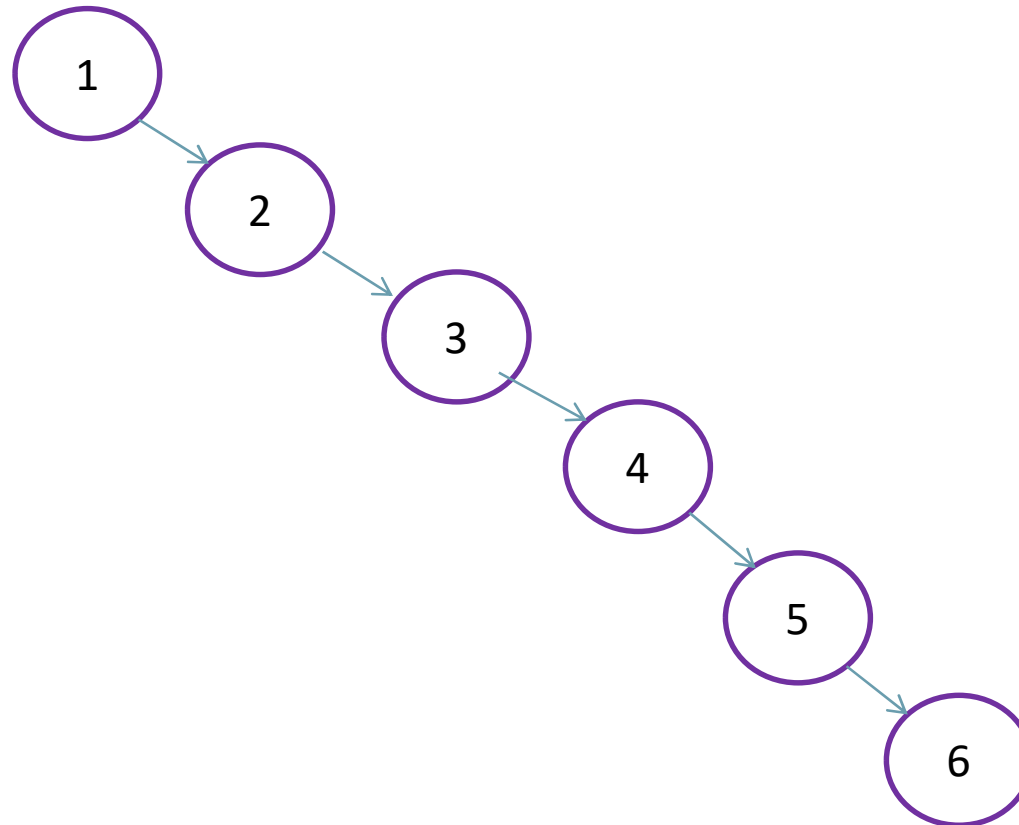
AVL TREES

- AVL tree is a self-balancing binary search tree invented by Adelson, Velsky and Landis in 1962

AVL TREES

- In an AVL tree, the heights of the two sub-trees of a node may differ by at most one.
- Due to this property, the AVL tree is also known as a height-balanced tree.
- The key advantage of using an AVL tree is that it takes **$O(\log n)$** time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to **$O(\log n)$** .

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



Skewed BST

AVL TREES

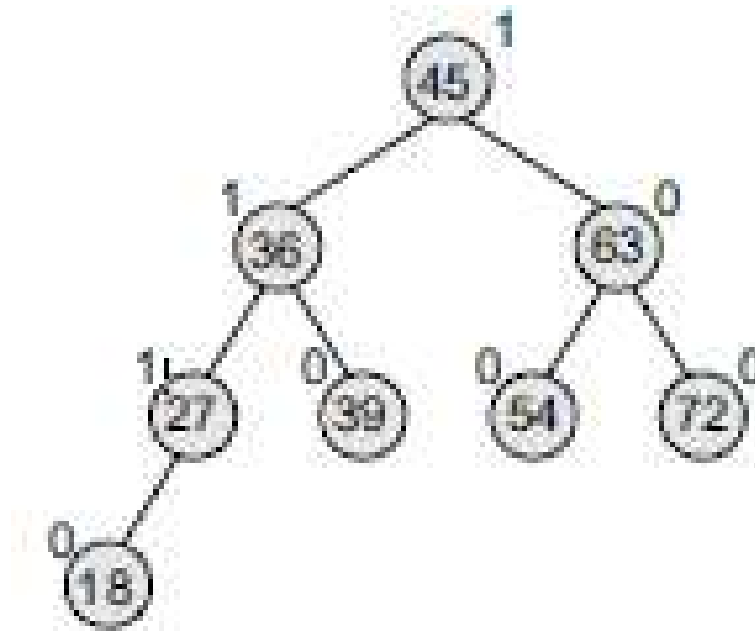
- The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the Balance Factor.
- Every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

$$\text{Balance factor} = \text{Height (left sub-tree)} - \text{Height (right sub-tree)}$$

- A binary search tree in which every node has a balance factor of -1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

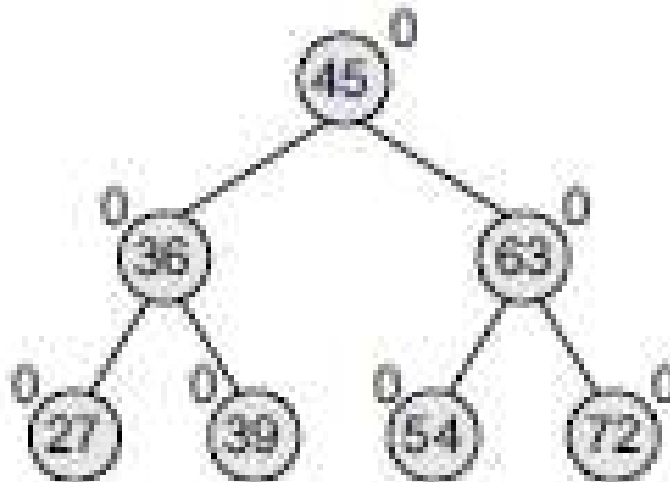
Left-Heavy Tree

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a left-heavy tree.



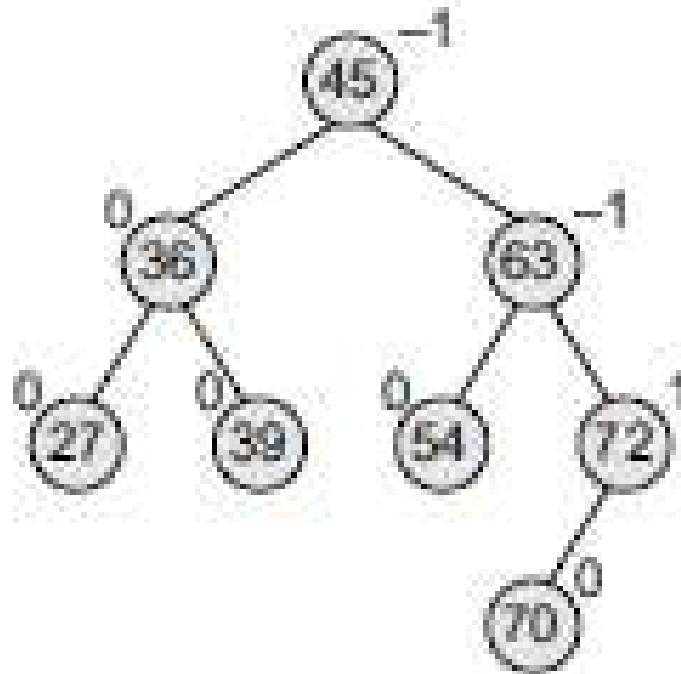
Balanced AVL TREES

- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.



Right-Heavy tree

- If the balance factor of a node is -1 , then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a right-heavy tree.



Operations on AVL Trees

Searching for a Node in an AVL Tree:

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Due to the height-balancing of the tree, the search operation takes $O(\log n)$ time to complete.
- Since the operation does not modify the structure of the tree, no special provisions are required.

Inserting a New Node

Inserting a New Node in an AVL Tree:

- Insertion in an AVL tree is also done in the same way as it is done in a binary search tree.
- But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree.
- However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still -1 , 0 , or 1 , then rotations are not required.
- During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node.

Inserting a New Node

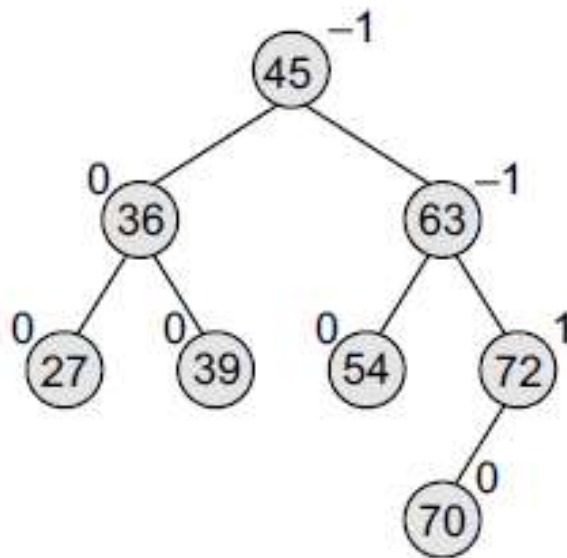
Inserting a New Node in an AVL Tree:

The possible changes which may take place in any node on the path are as follows:

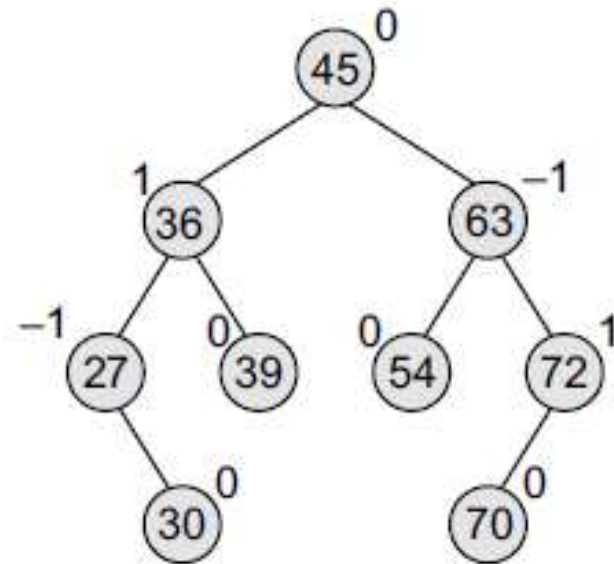
- Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
- Initially, the node was balanced and after insertion, it becomes either left-heavy or right-heavy.
- Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a ***critical node***.

Inserting a New Node

Insert 30



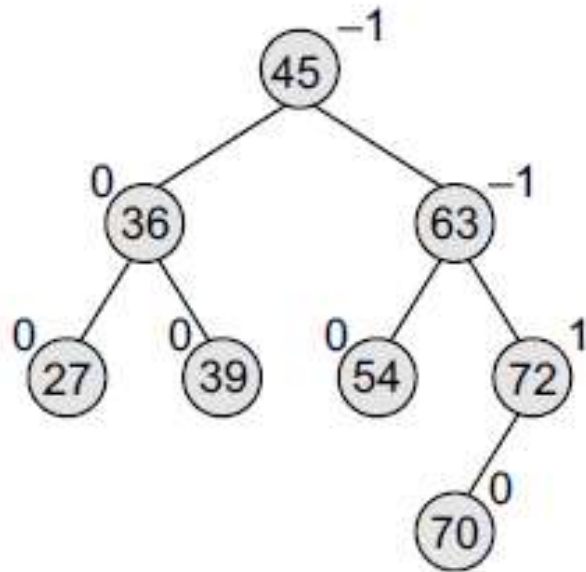
Balanced



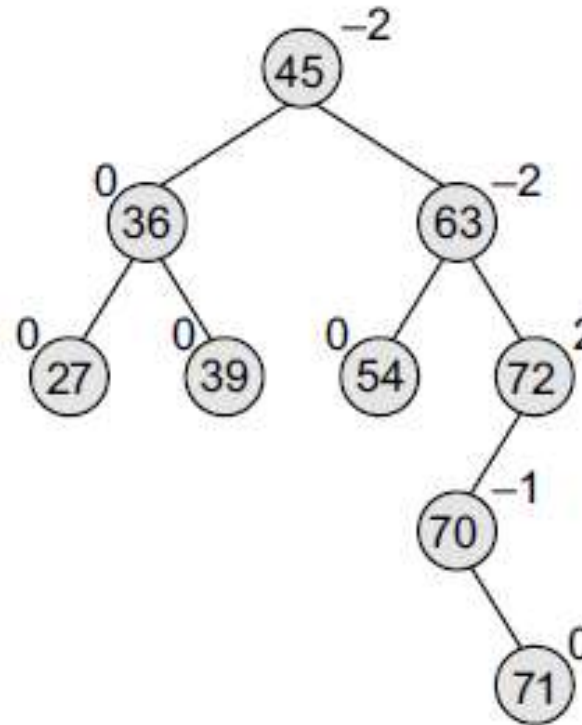
Balanced

Inserting a New Node

Insert 71



Balanced



Not Balanced

Critical Node

Rotations in AVL tree

- **LL rotation:**

The new node is inserted in the left sub-tree of the left sub-tree of the critical node.

- **RR rotation:**

The new node is inserted in the right sub-tree of the right sub-tree of the critical node.

- **LR rotation:**

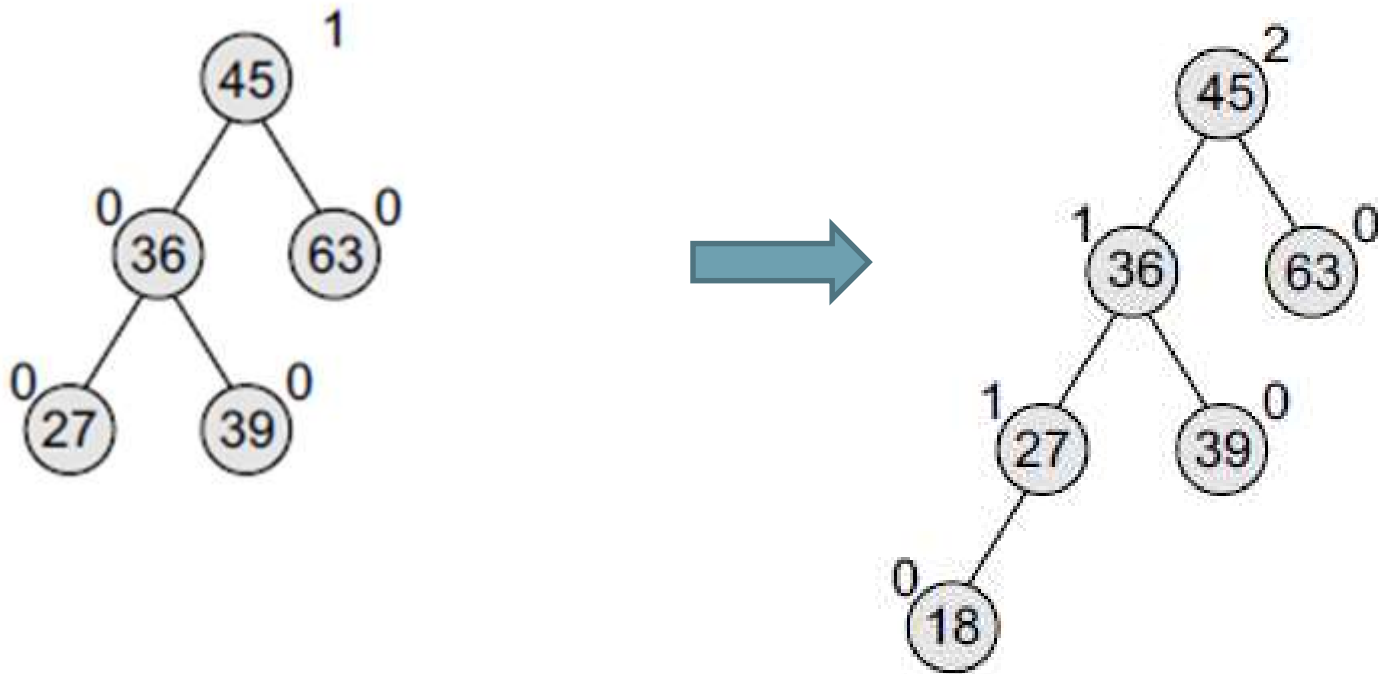
The new node is inserted in the right sub-tree of the left sub-tree of the critical node.

- **RL rotation:**

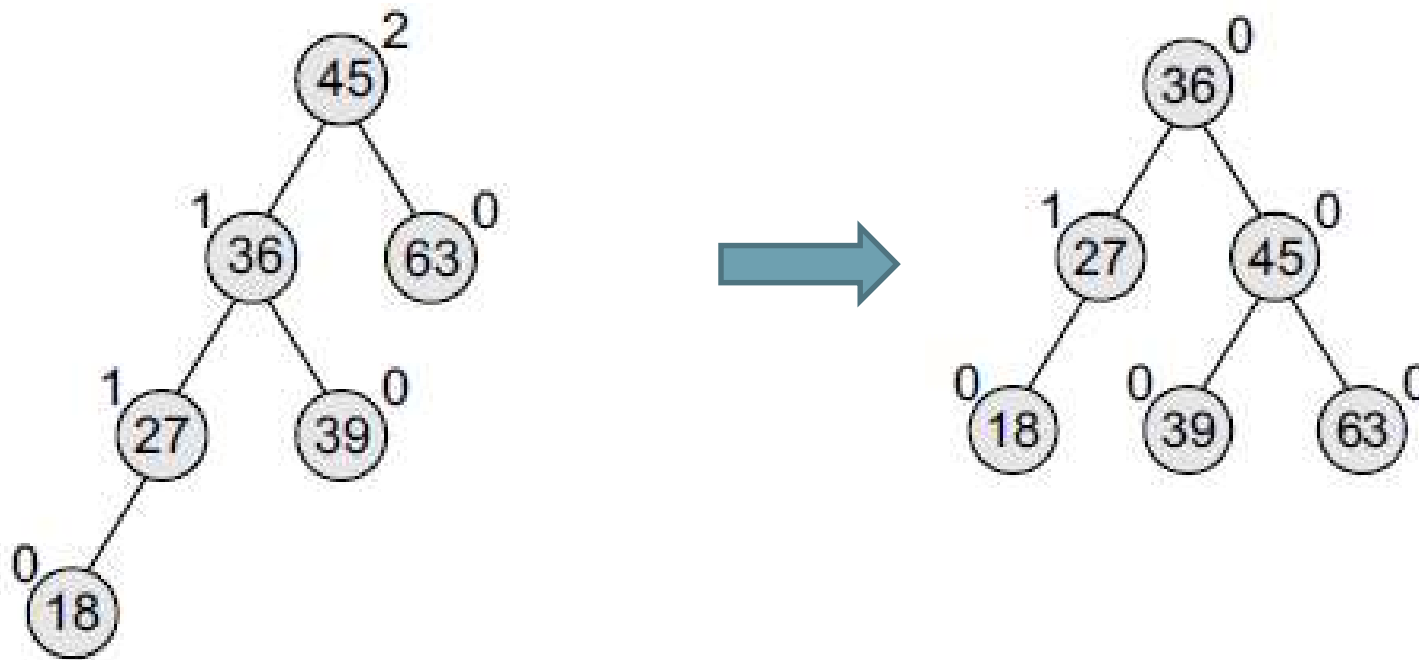
The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

LL rotation in AVL tree

Insert 18

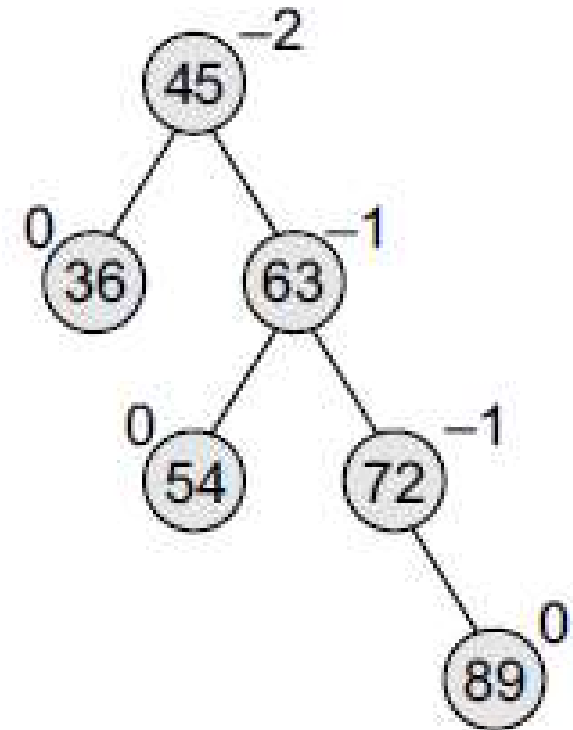
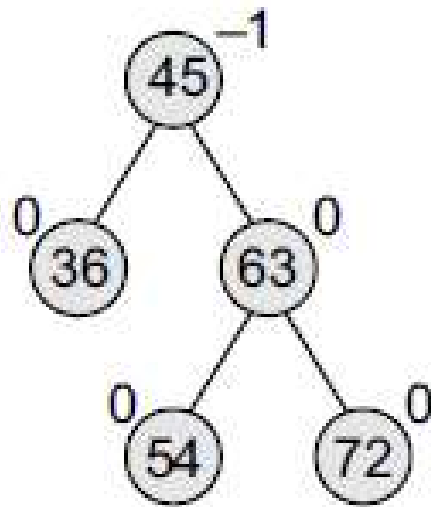


LL Rotations in AVL Tree

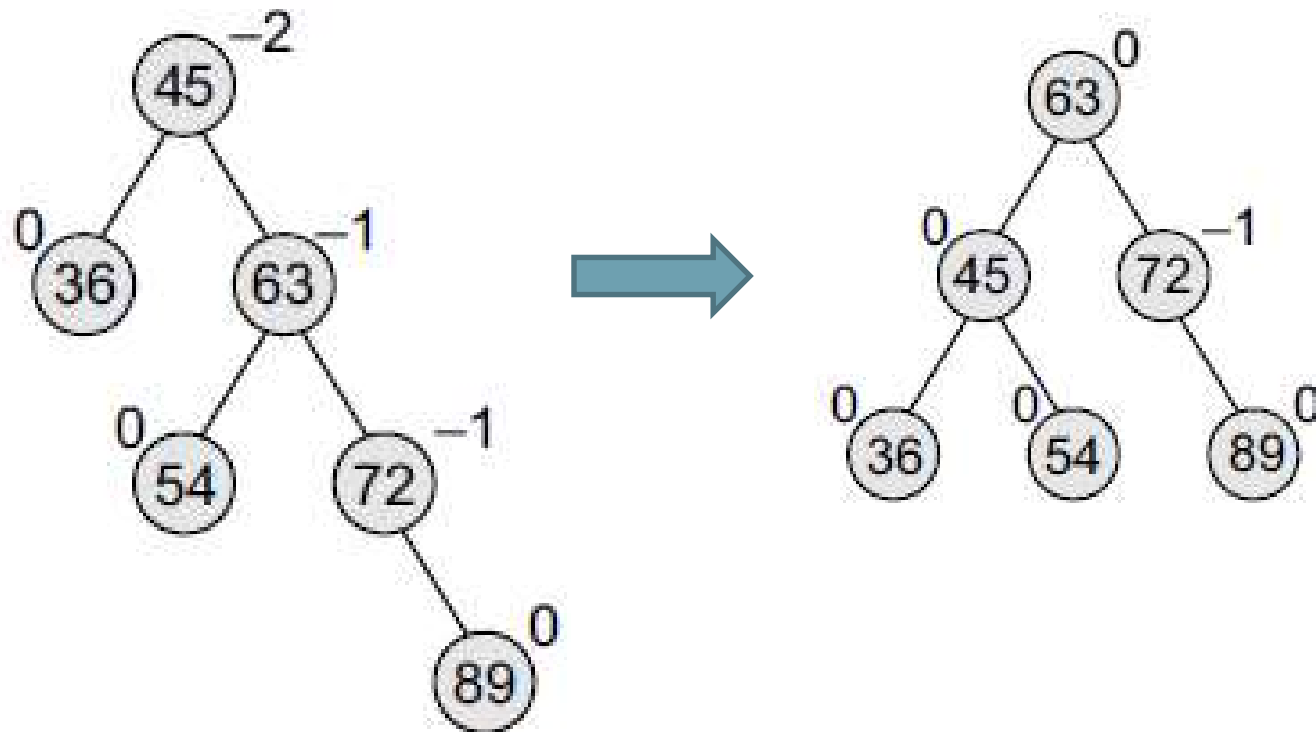


RR rotation in AVL Tree

Insert 89

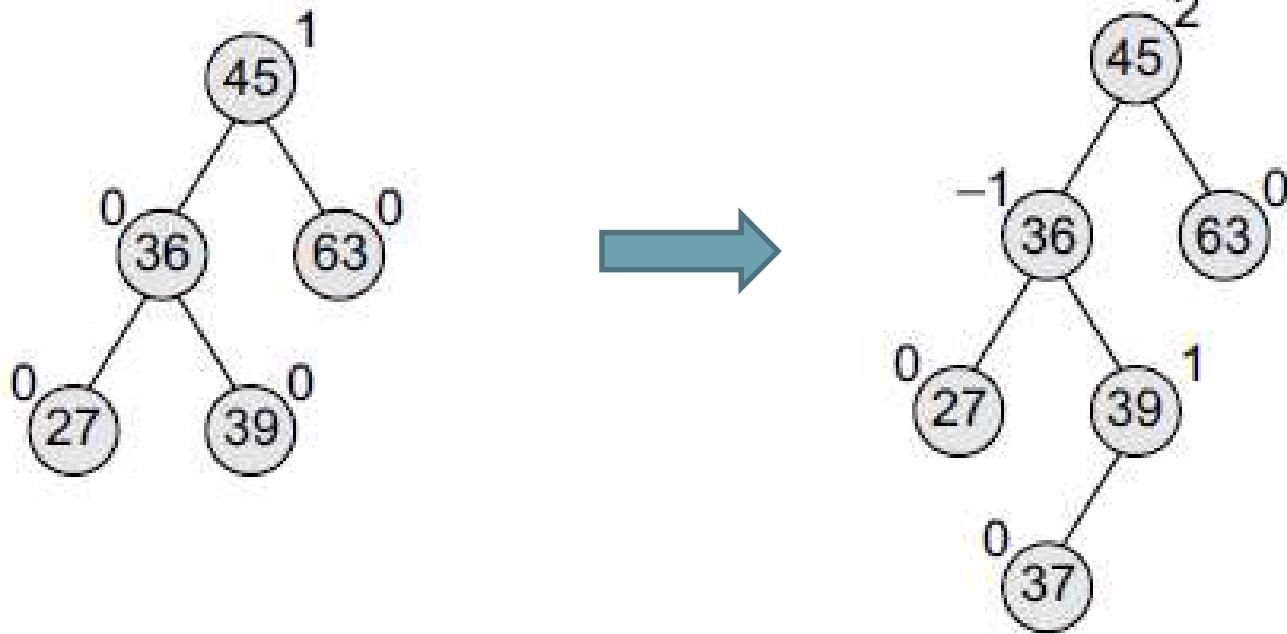


RR Rotations in AVL Tree

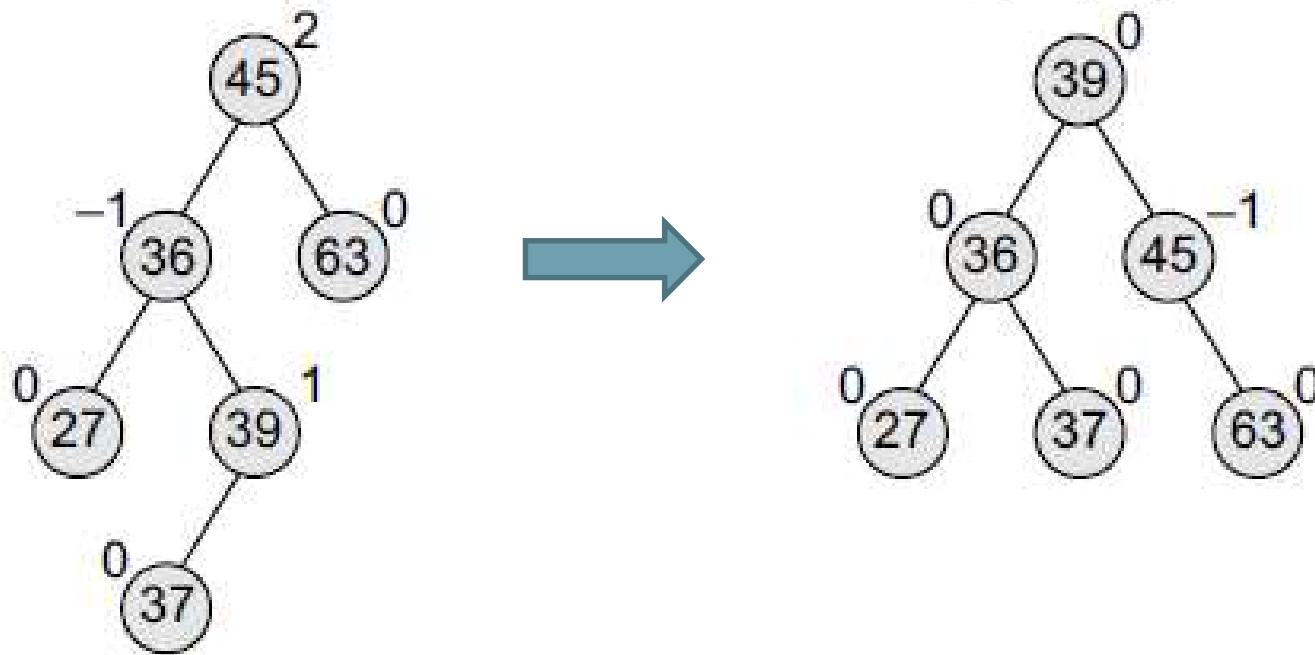


LR Rotation in AVL Tree

Insert 37

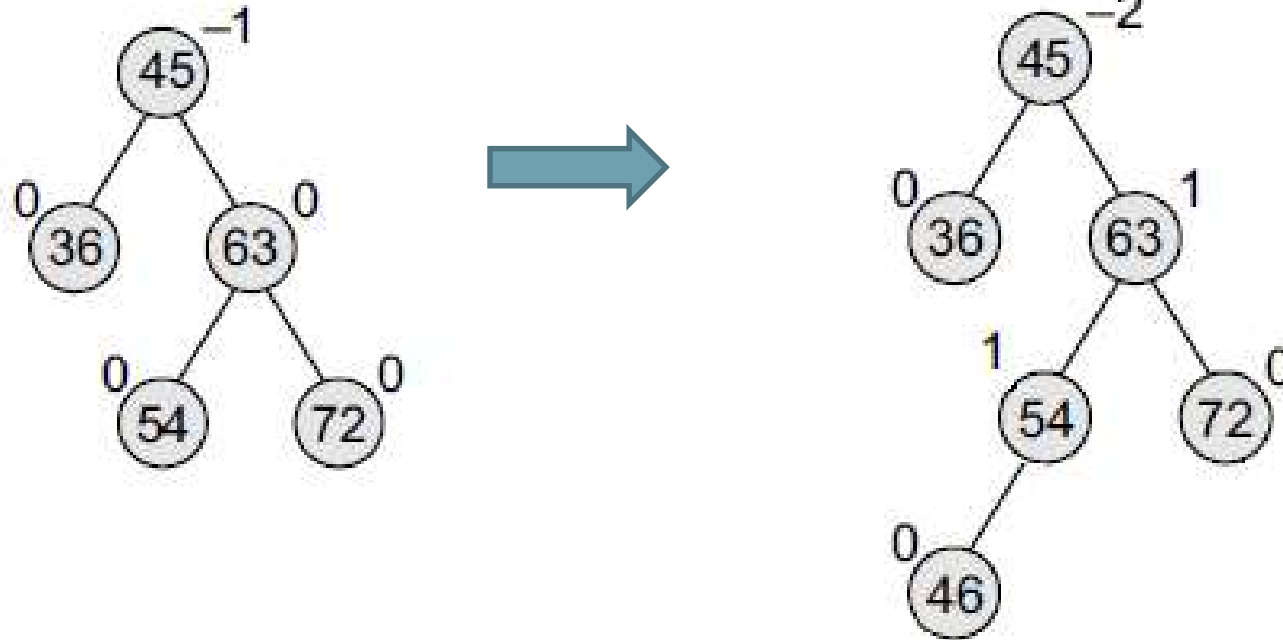


LR Rotation in AVL Tree

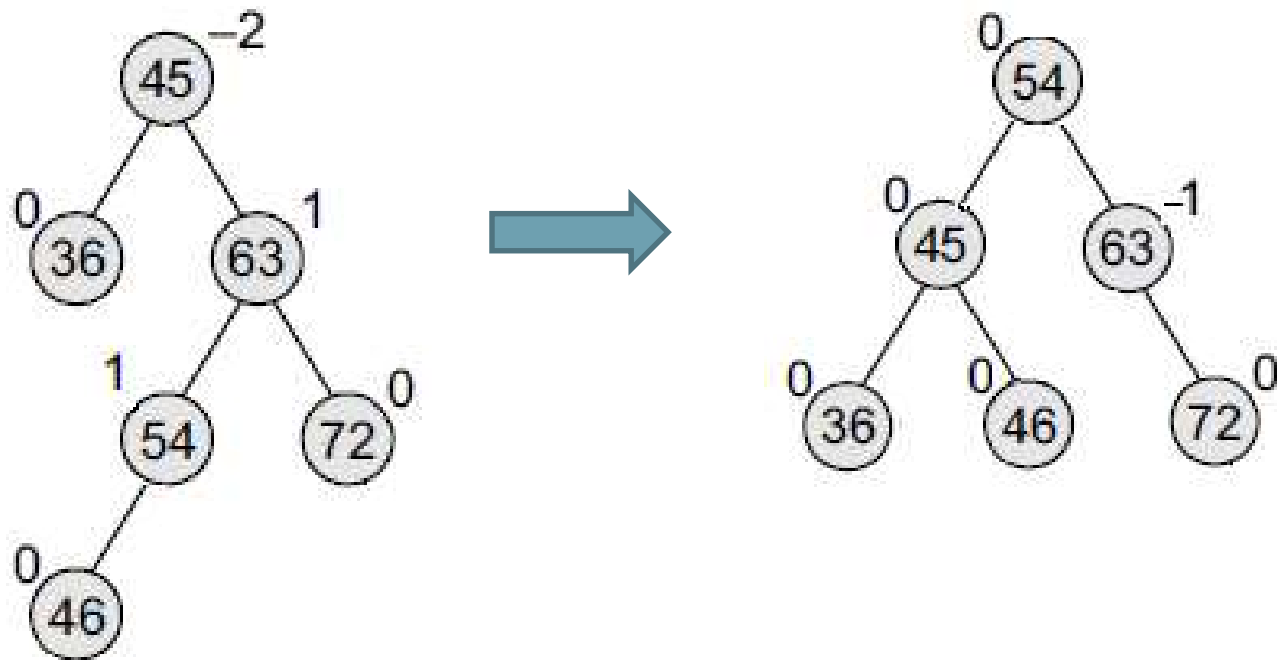


RL Rotation in AVL Tree

Insert 46



RL Rotation in AVL Tree



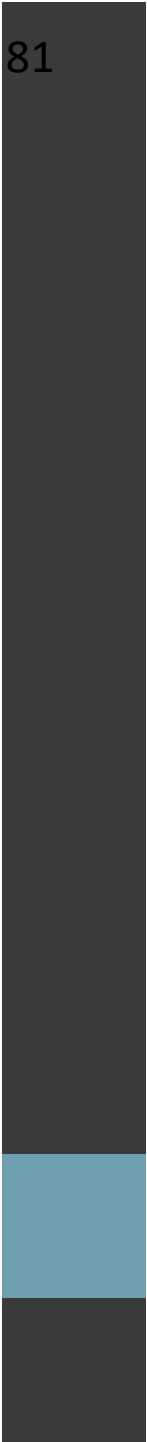
Example to Construct an AVL Tree

- Construct an AVL tree by inserting the following elements in the given order.
63, 9, 19, 27, 18, 108, 99, 81.



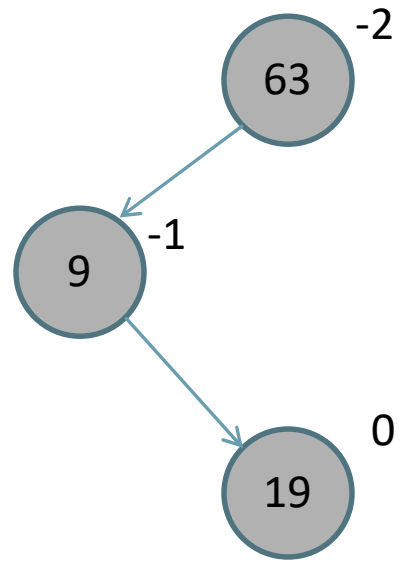
0

63, 9, 19, 27, 18, 108, 99, 81



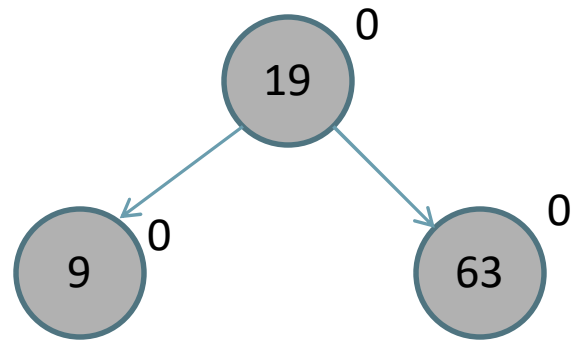
Kanak Kalyani

63, 9, 19, 27, 18, 108, 99, 81

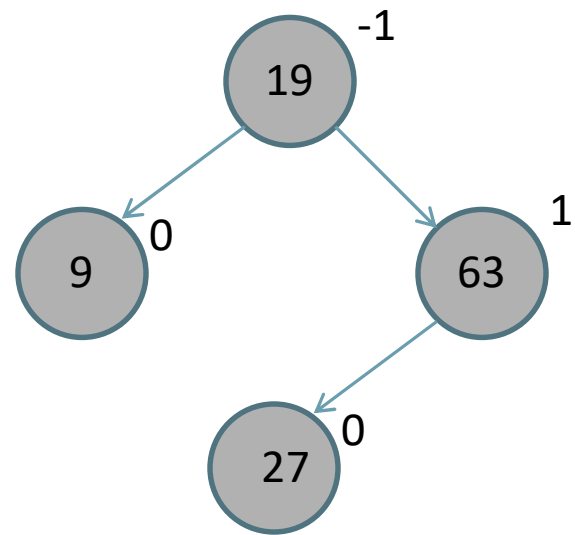


Imbalanced – LR Rotation

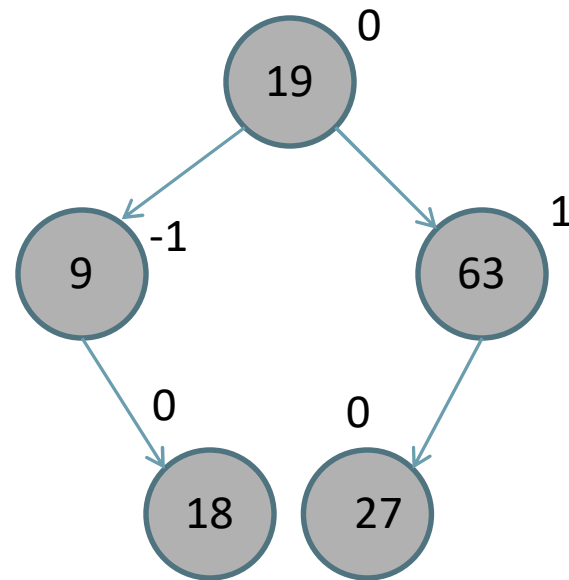
63, 9, 19, 27, 18, 108, 99, 81



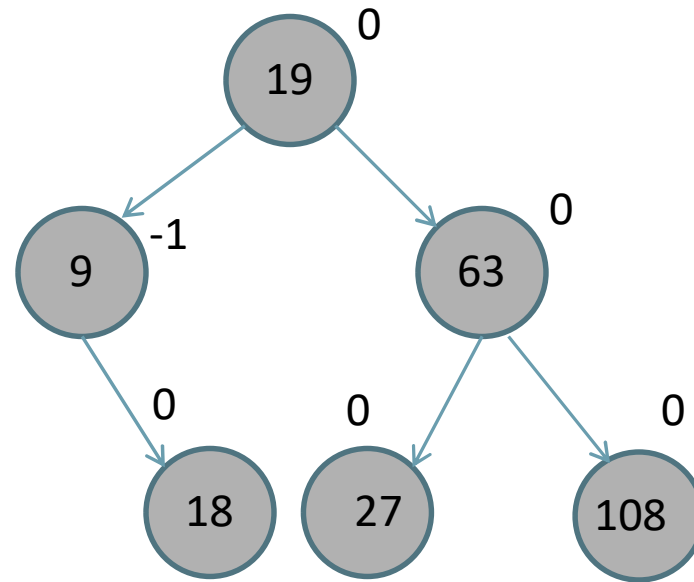
63, 9, 19, 27, 18, 108, 99, 81



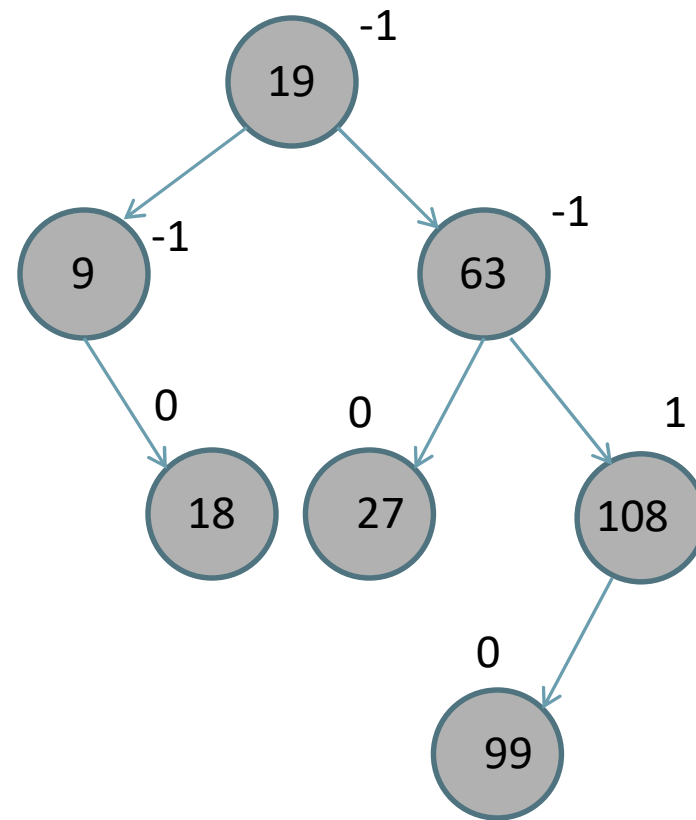
63, 9, 19, 27, 18, 108, 99, 81



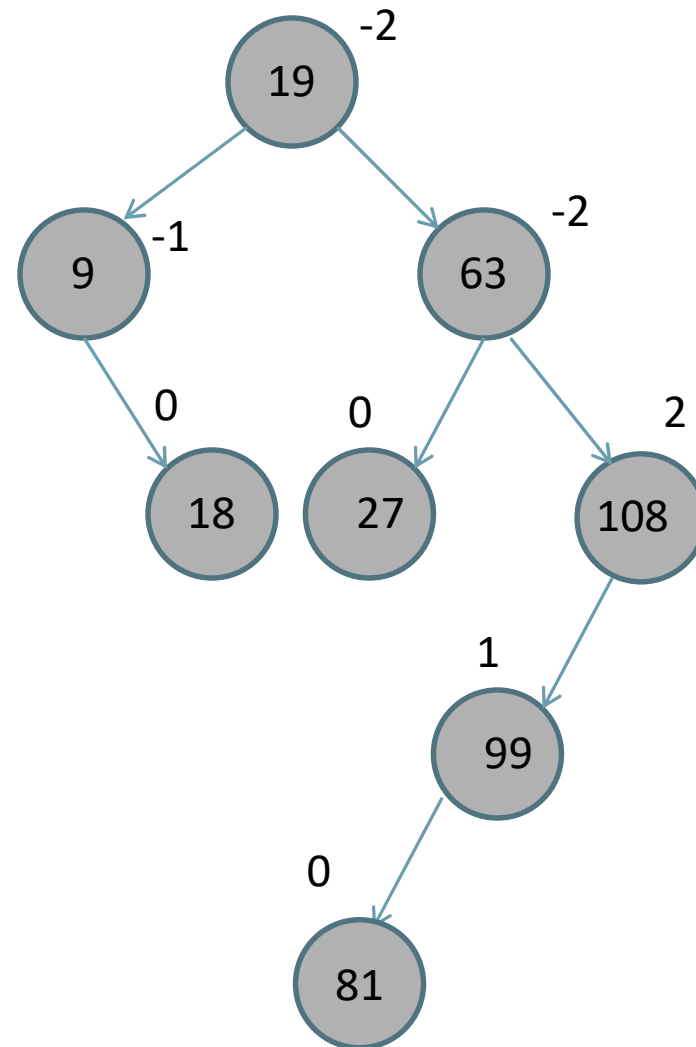
63, 9, 19, 27, 18, 108, 99, 81



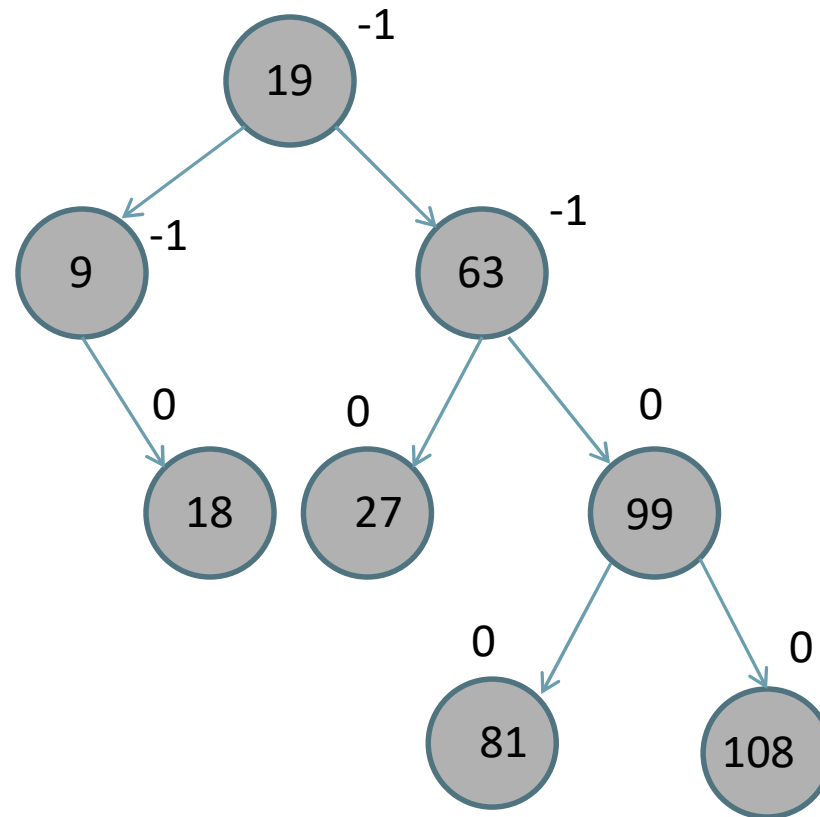
63, 9, 19, 27, 18, 108, 99, 81



63, 9, 19, 27, 18, 108, 99, 81



63, 9, 19, 27, 18, 108, 99, 81

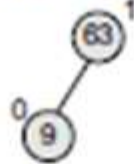


LL Rotation – 108 , 81, 99

(Step 1)



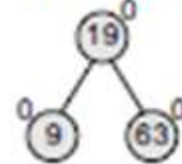
(Step 2)



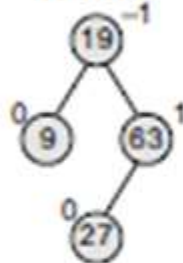
(Step 3)



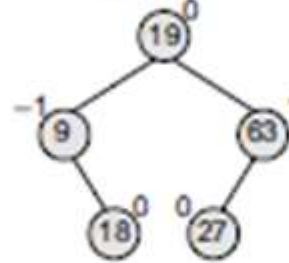
After LR Rotation
(Step 4)



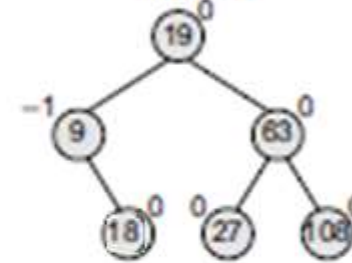
(Step 5)



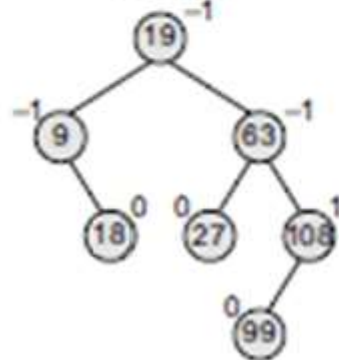
(Step 6)



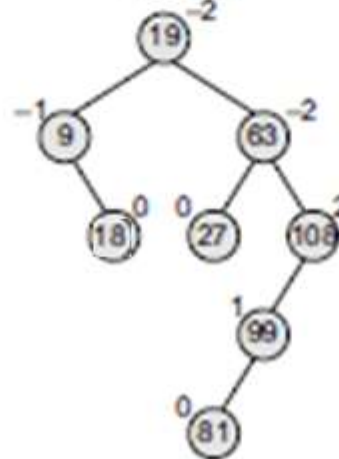
(Step 7)



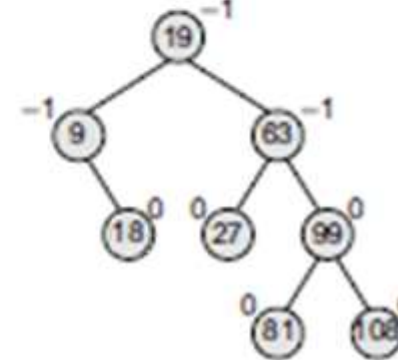
(Step 8)



(Step 9)

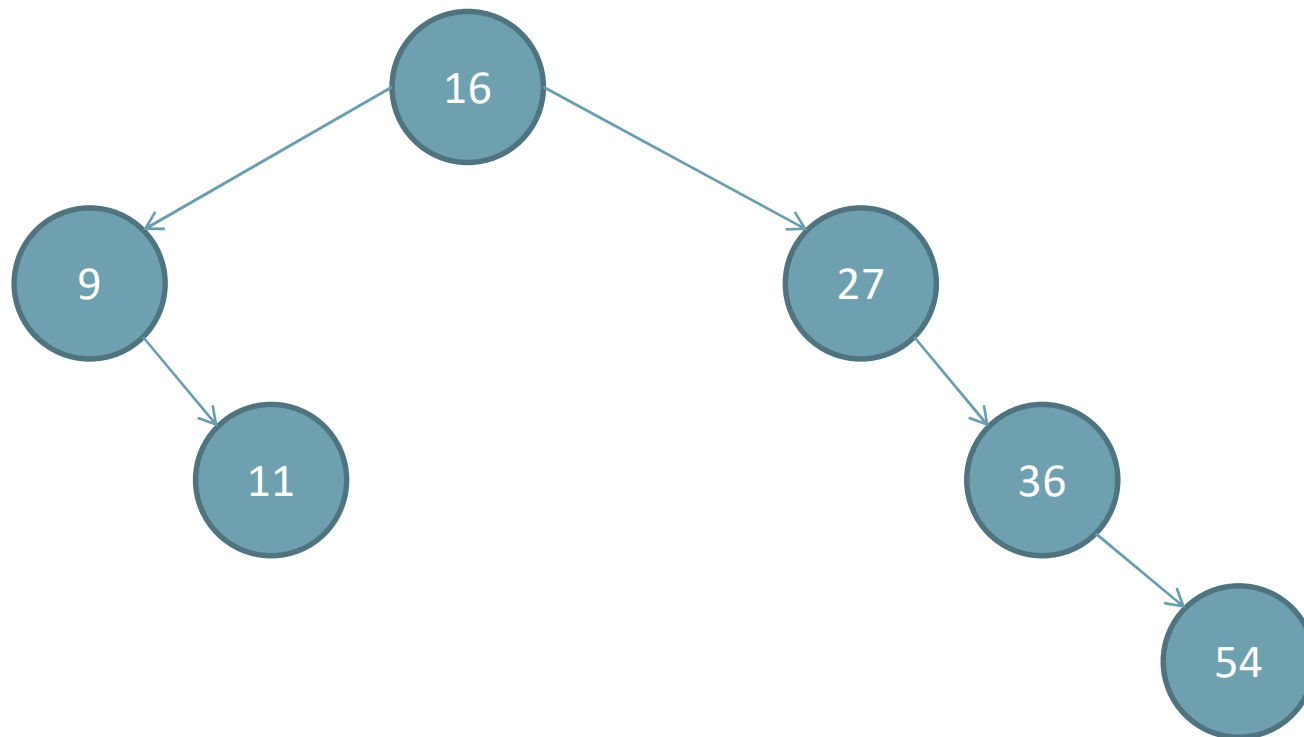


After LL Rotation
(Step 10)



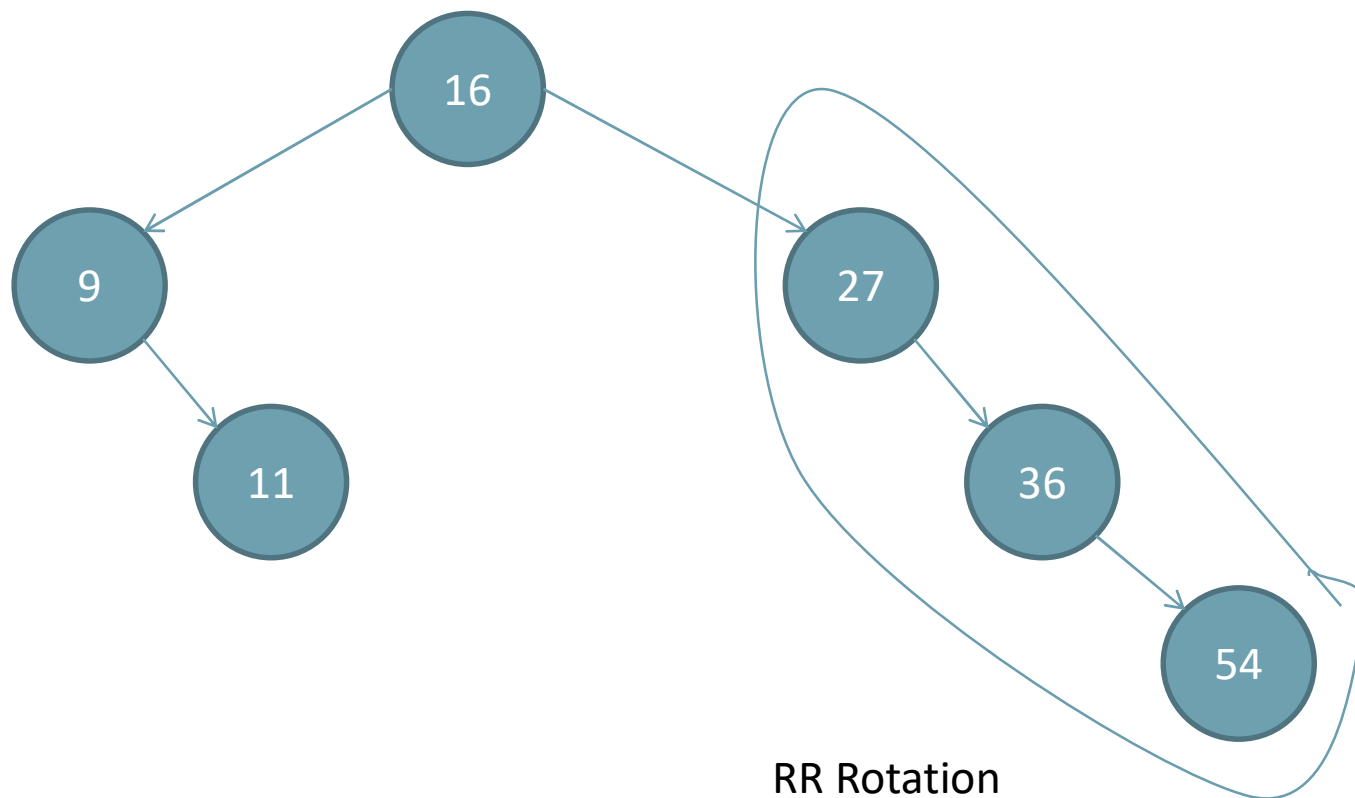
AVL TREE

- Construct an AVL Tree with following nodes:
16, 27, 9, 11, 36, 54, 81, 63, 72



AVL TREE

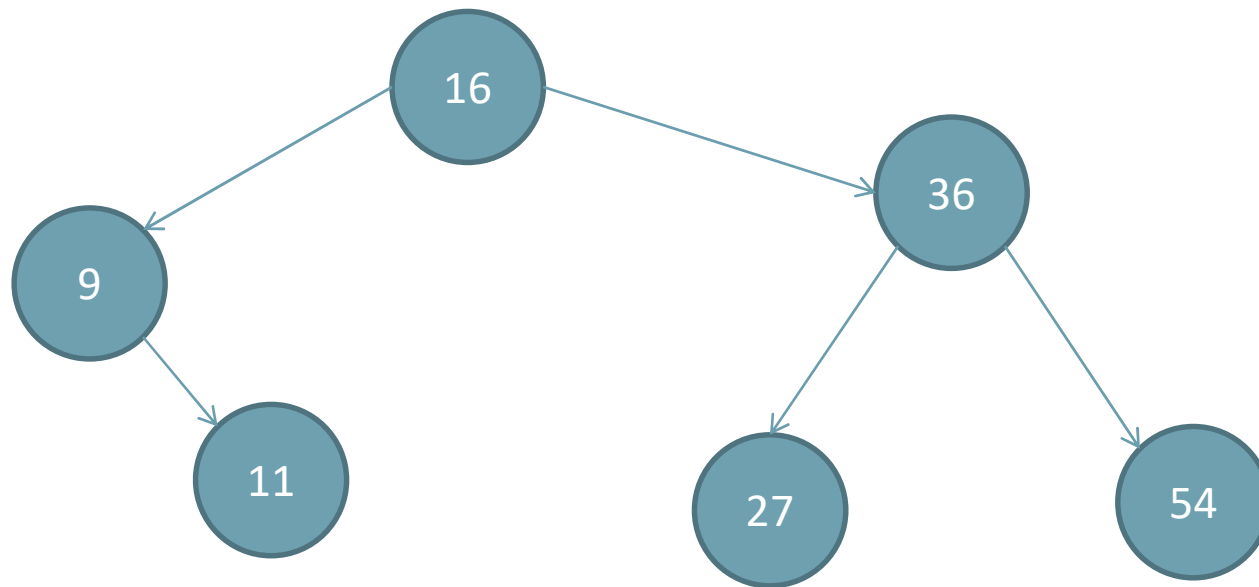
- Construct an AVL Tree with following nodes:
16, 27, 9, 11, 36, 54, 81, 63, 72



Kanak Kalyani

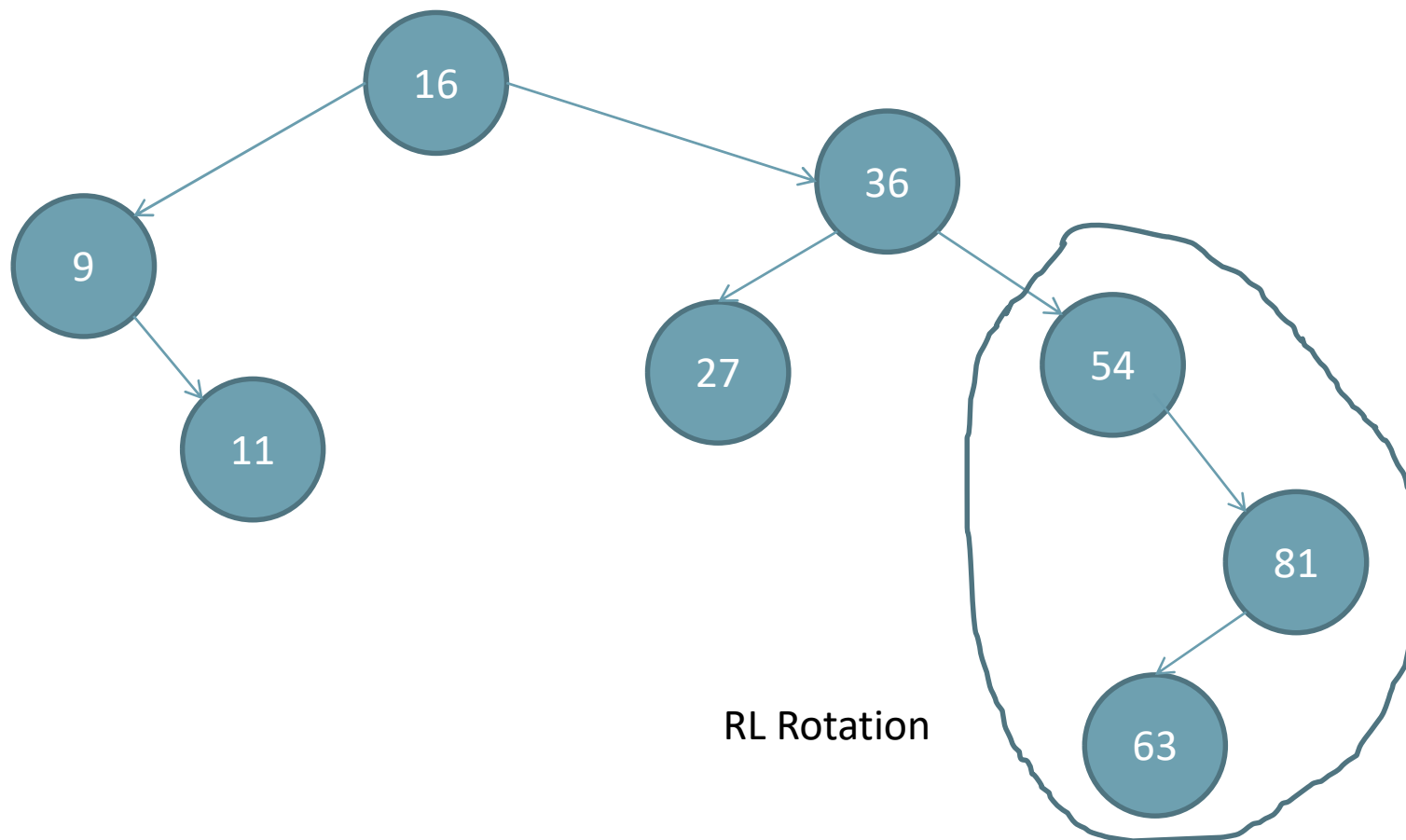
AVL TREE

- Construct an AVL Tree with following nodes:
16, 27, 9, 11, 36, 54, 81, 63, 72



AVL TREE

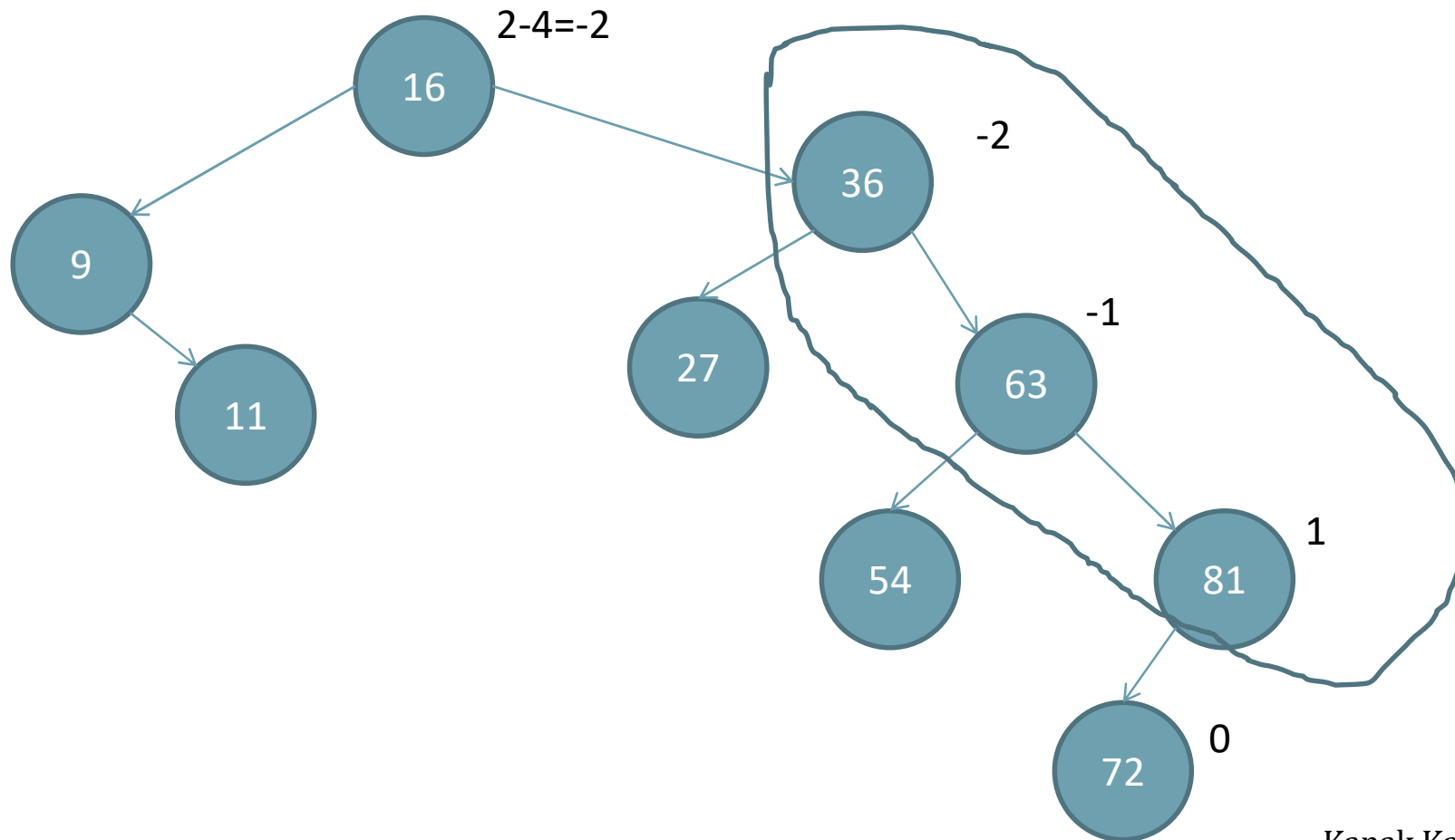
- Construct an AVL Tree with following nodes:
16, 27, 9, 11, 36, 54, 81, 63, 72



Kanak Kalyani

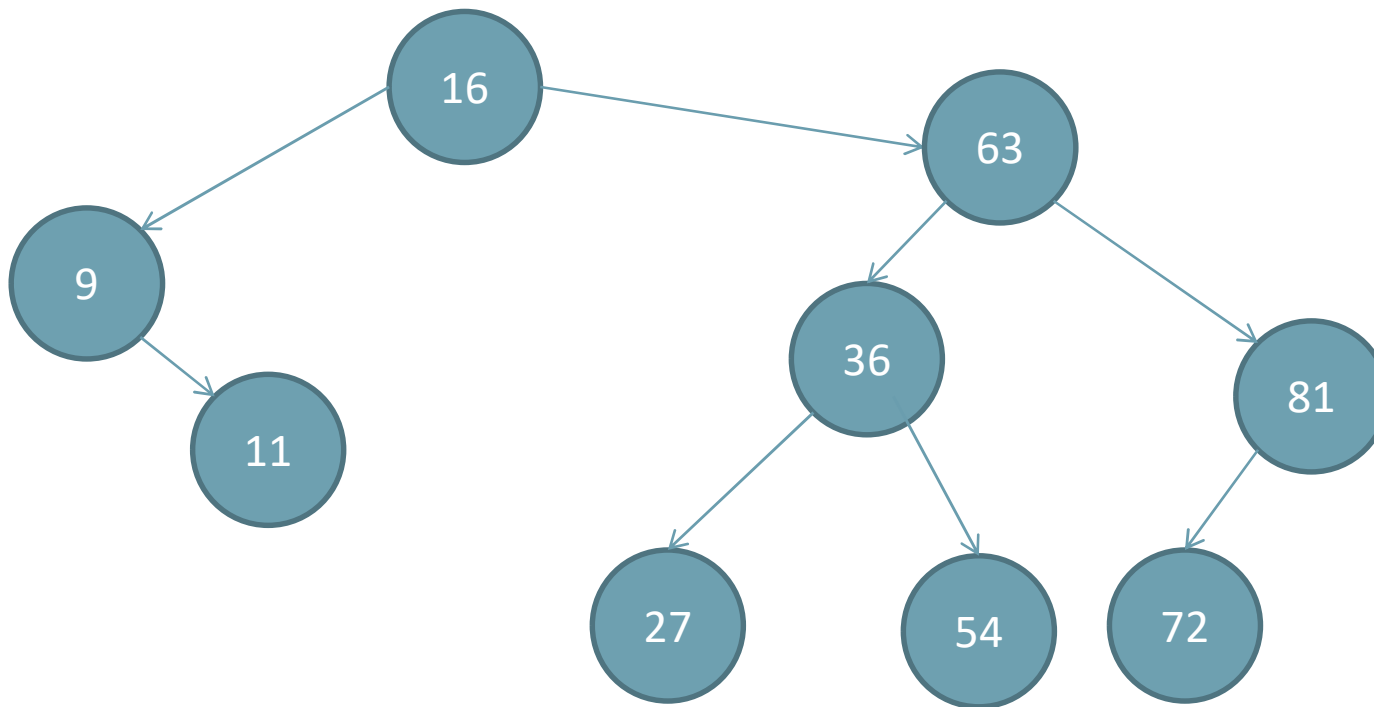
AVL TREE

- Construct an AVL Tree with following nodes:
16, 27, 9, 11, 36, 54, 81, 63, 72



AVL TREE

- Construct an AVL Tree with following nodes:
16, 27, 9, 11, 36, 54, 81, 63, 72



Multi-way Search Trees

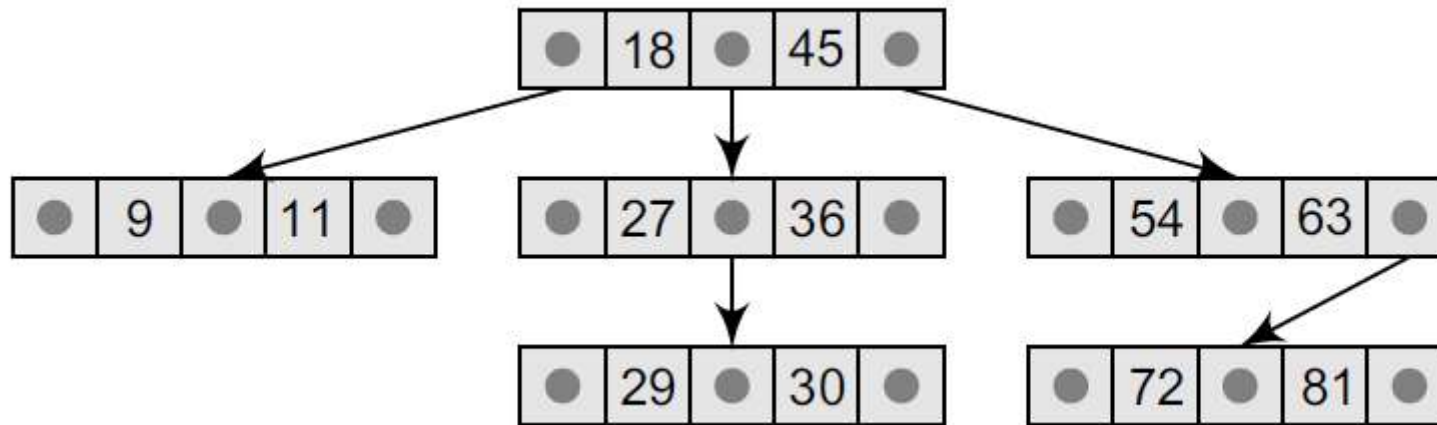
M-way search tree

- *M-way* search tree is similar to binary tree but has $M - 1$ values per node and M subtrees.
- M is called the degree of the tree
- If $M = 2$, means it has one value and two sub-trees
- every internal node of an M -way search tree consists of pointers to M sub-trees and contains $M - 1$ keys, where $M > 2$.

P_0	K_0	P_1	K_1	P_2	K_2	$\dots\dots$	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-------	-------	-------	--------------	-----------	-----------	-------

M-way search tree

- M-way search tree of order 3



M-way search tree

Properties of an M-way search tree

- **Key values in the sub-tree pointed by P_0 are less than the key value K_0 .** Similarly, all the key values in the sub-tree pointed by P_1 are less than K_1 , so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are less than K_i , where $0 \leq i \leq n-1$.
- **Key values in the sub-tree pointed by P_1 are greater than the key value K_0 .** Similarly, all the key values in the sub-tree pointed by P_2 are greater than K_1 , so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are greater than K_{i-1} , where $0 \leq i \leq n-1$.

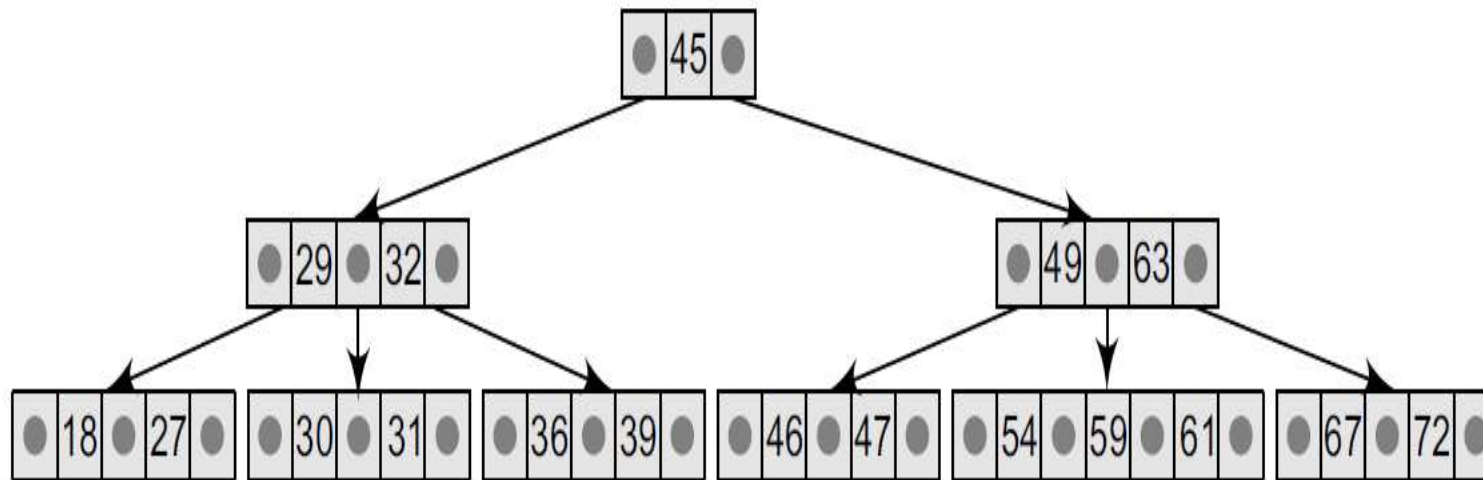
B - Tree

- B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970
- It was widely used for Disk access.
- A B tree of order m can have a maximum of $m-1$ keys and m pointers to its sub-trees.
- A B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small

B - Tree

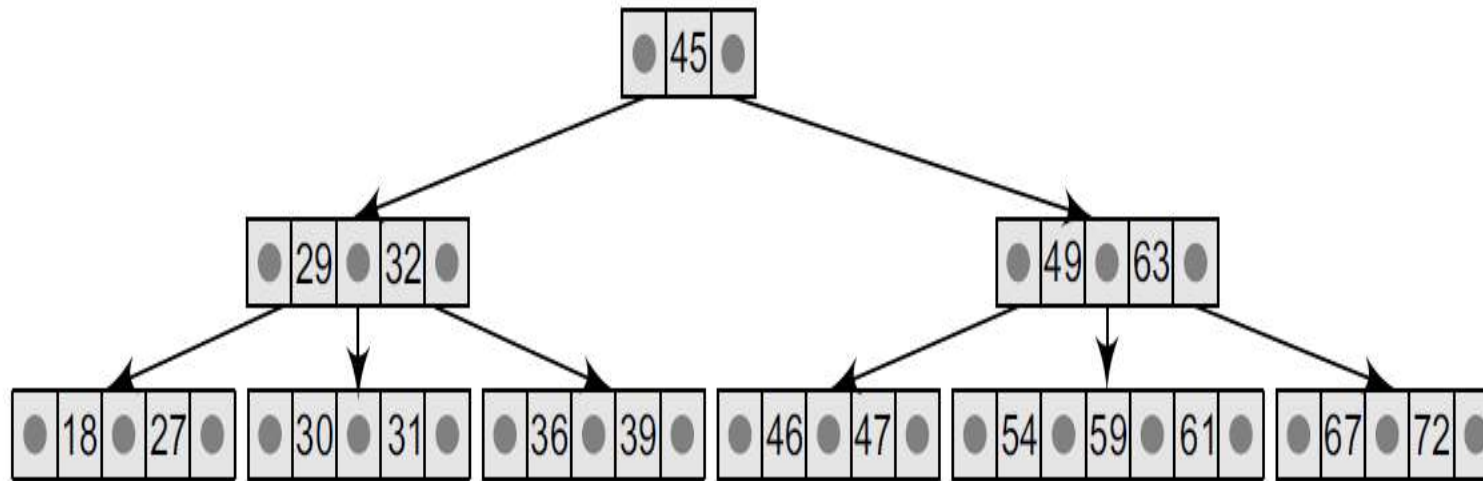
- A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree.
- In addition it has the following properties:
 1. Every node in the B tree **has at most m children**.
 2. **Every node** in the B tree except the root node and leaf nodes **has at least (minimum) $m/2$ children**.
 3. **The root node has at least two children** if it is not a terminal (leaf) node.
 4. **All leaf nodes are at the same level**.

B - Tree of order 4



Searching for an Element in a B Tree

- Search 31



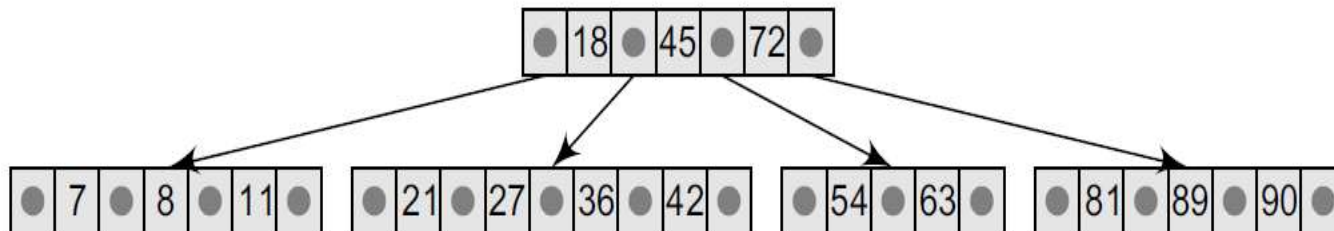
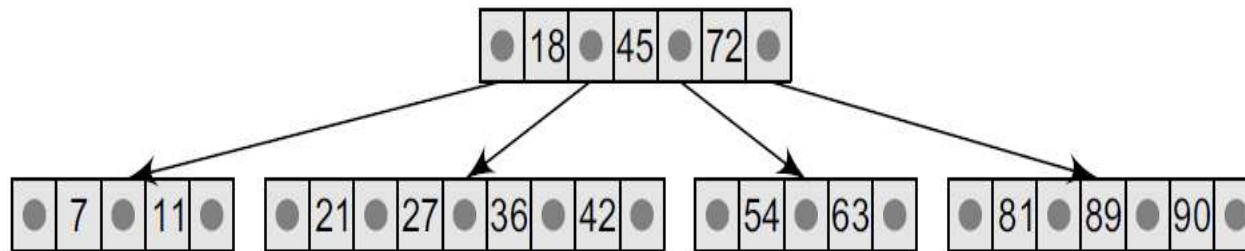
Inserting a node in B Tree

1. **Search the B tree** to find the leaf node where the new key value should be inserted.
2. If the **leaf node is not full**, that is, it contains less than $m-1$ key values, then **insert the new element** in the node keeping the node's elements ordered.
3. If the **leaf node is full**, that is, the leaf node already contains $m-1$ key values, then
 - (a) insert the new value in order into the existing set of keys,
 - (b) split the node at its median into two nodes (note that the split nodes are half full), and
 - (c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Inserting a node in B Tree

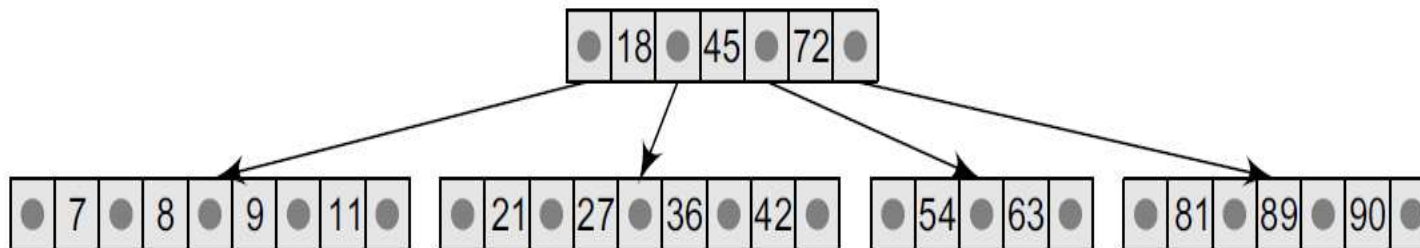
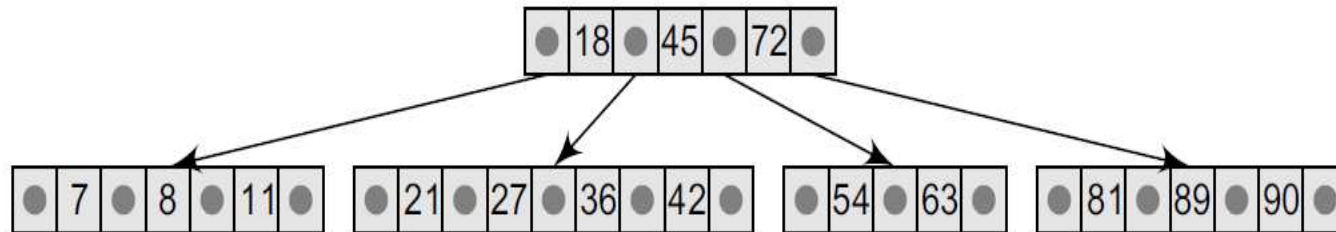
B Tree of Order 5

- Insert 8



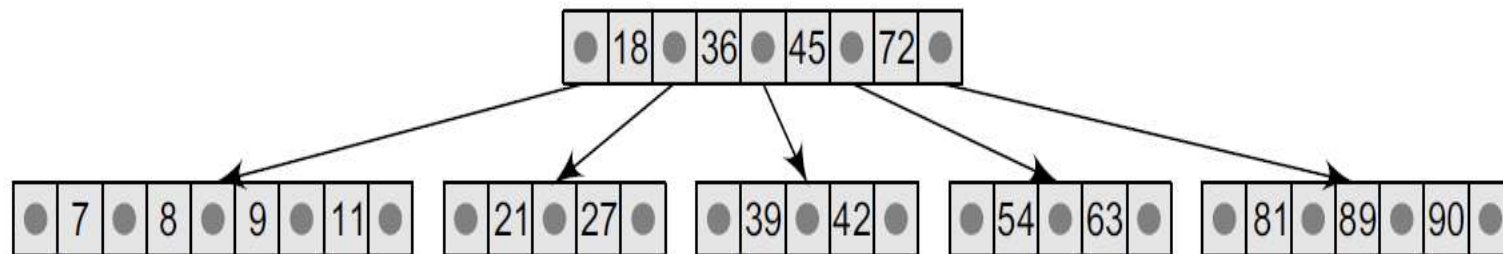
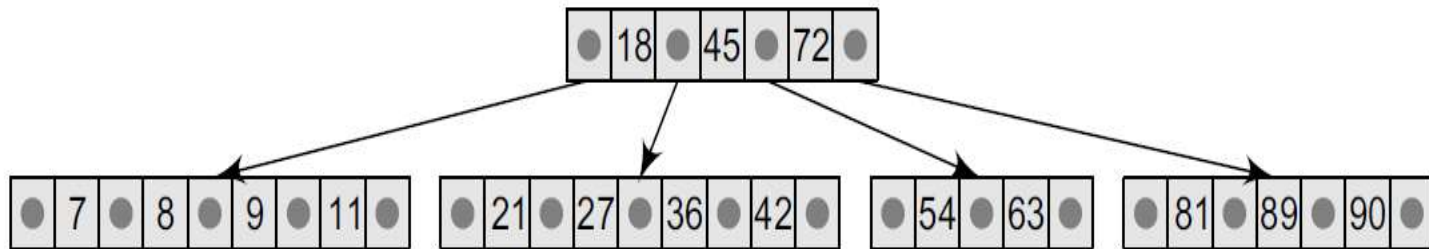
Inserting a node in B Tree

- **B Tree of Order 5**
- Insert 9



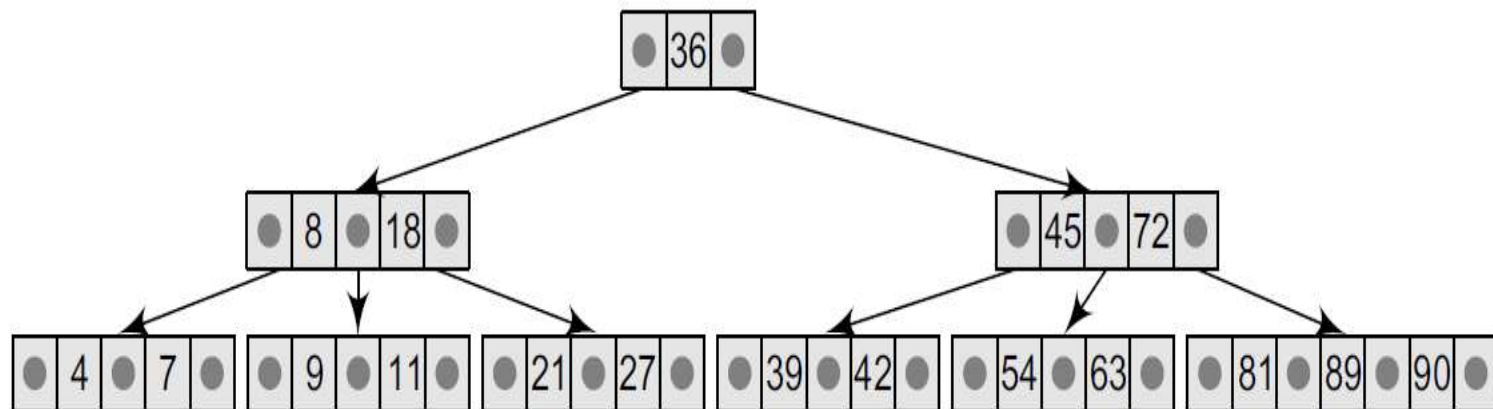
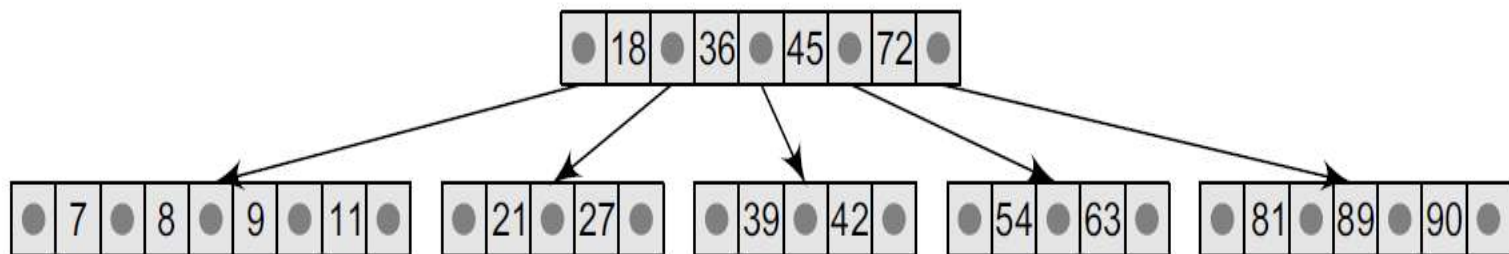
Inserting a node in B Tree

- **B Tree of Order 5**
- Insert 39



Inserting a node in B Tree

- **B Tree of Order 5**
- Insert 4



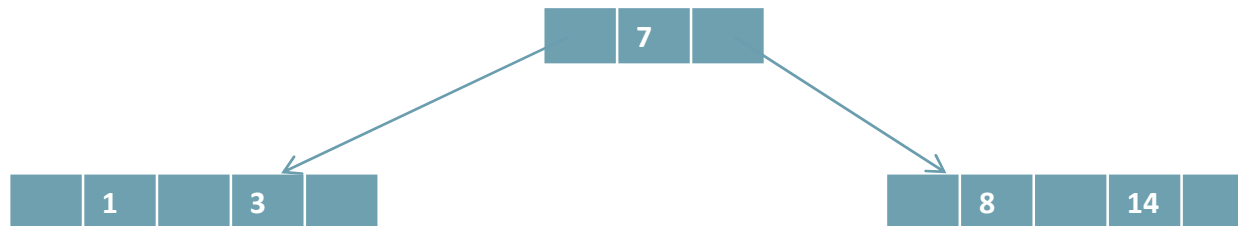
Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25



Insert 8

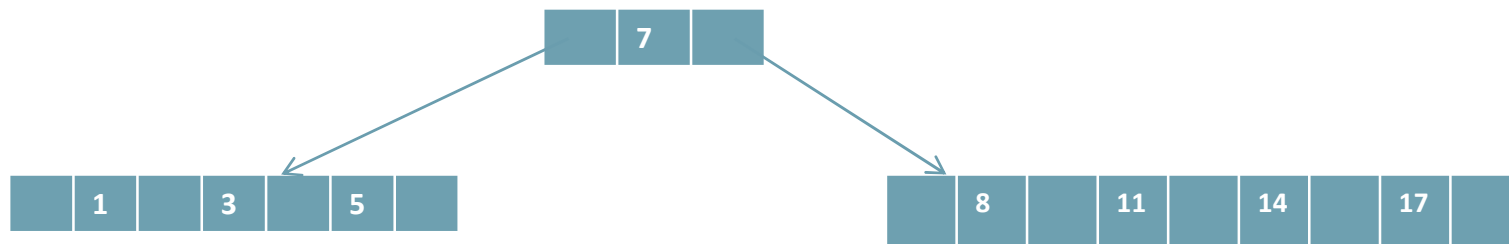


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 5, 11, 17

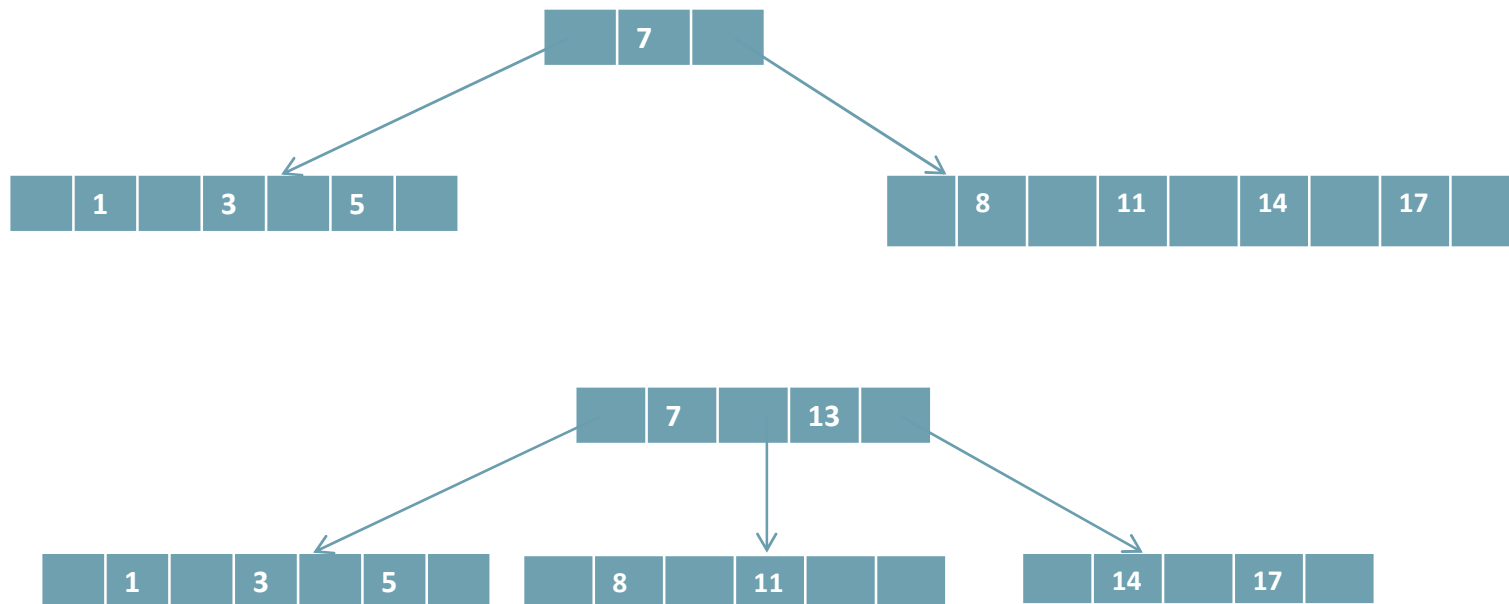


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 13

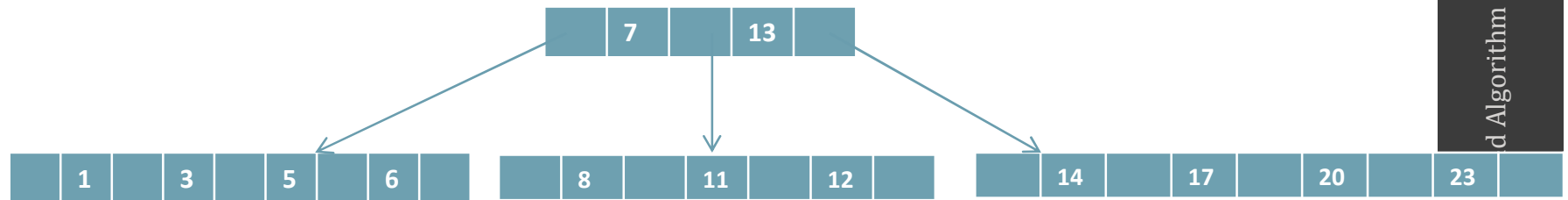


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 6,23,12,20

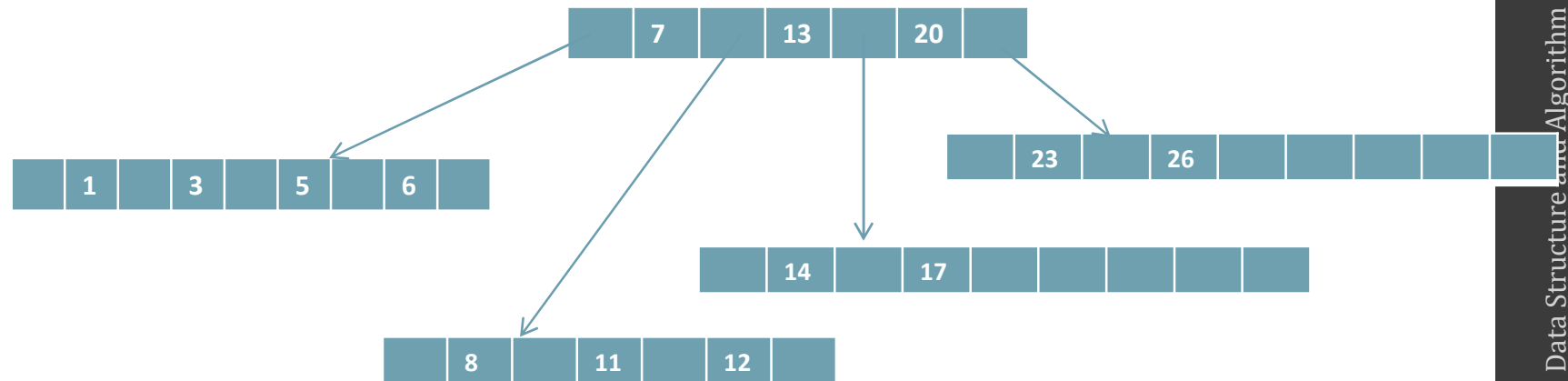


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 26

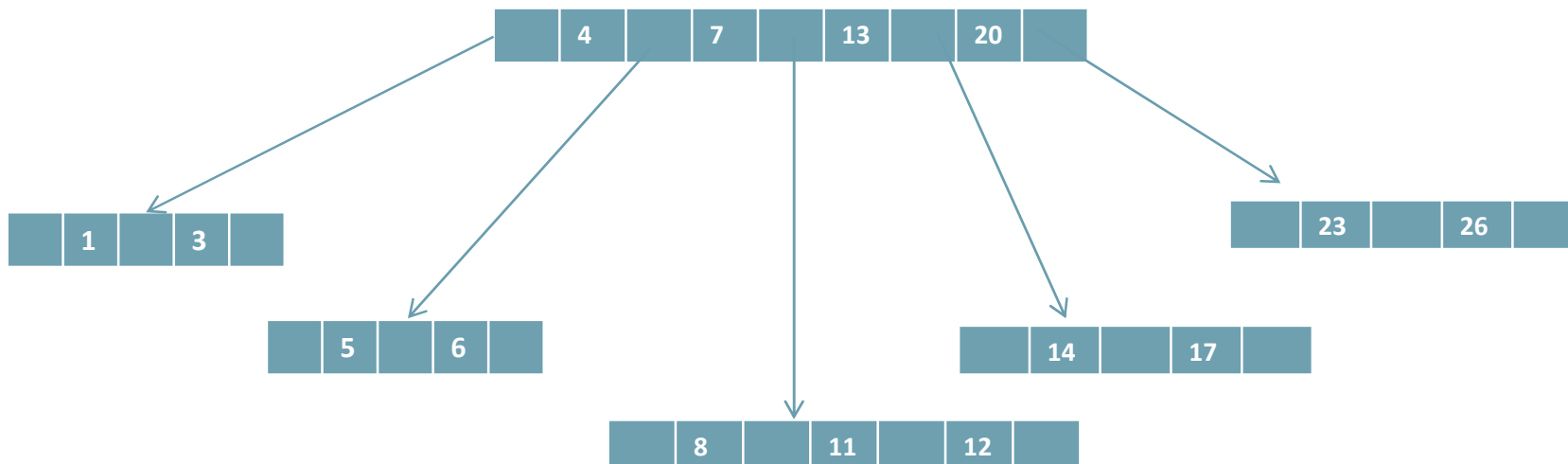


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 4

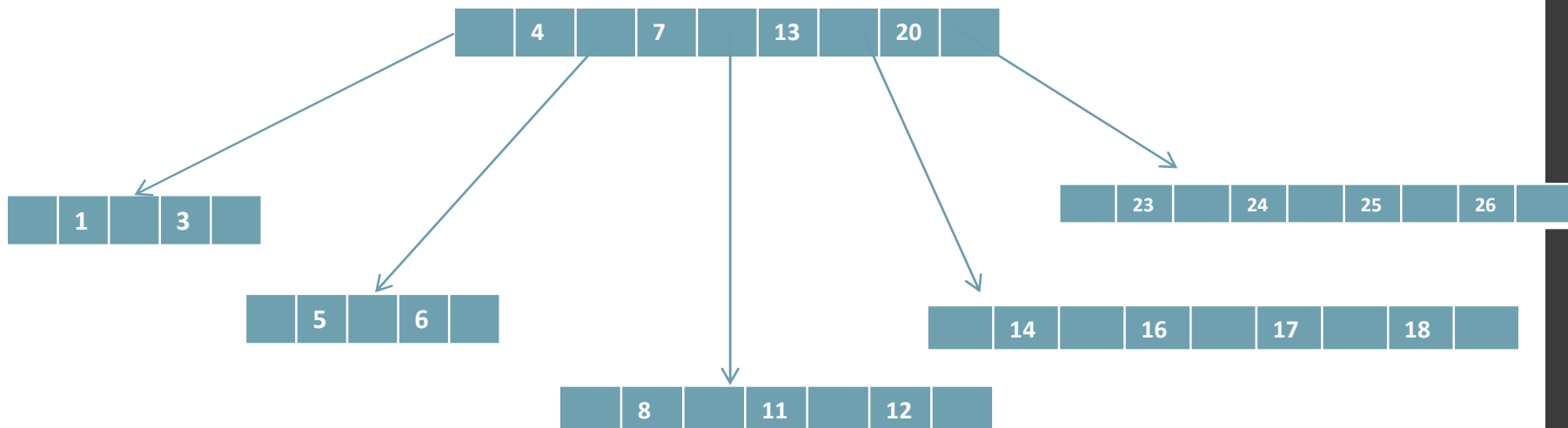


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 16,18,24,25



- Example:-

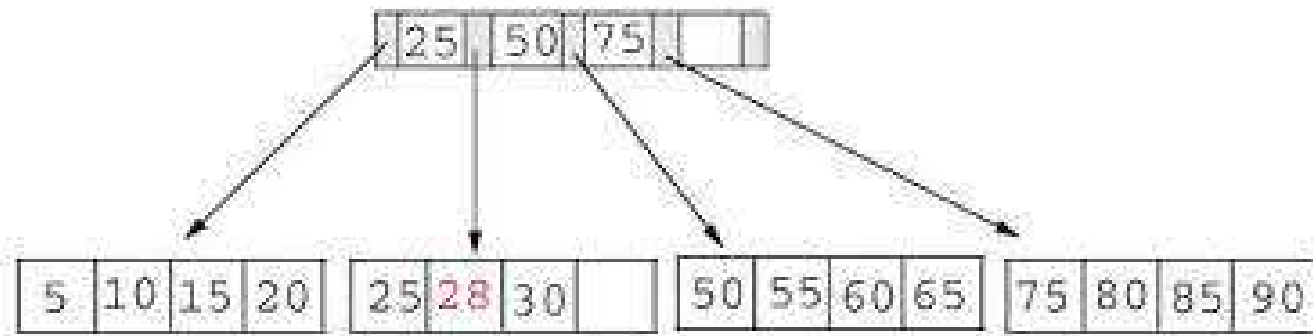
Create a B Tree of order 4:

26, 36, 48, 22, 80, 16, 14, 9, 8, 7

B+ Tree

- A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a *key*
- A B+ tree **stores all the records at the leaf level** of the tree; only keys are stored in the interior nodes
- The leaf nodes of a B+ tree are often linked to one another in a linked list
- B+ trees are used to store large amounts of data that cannot be stored in the main memory
- With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory

B+ Tree



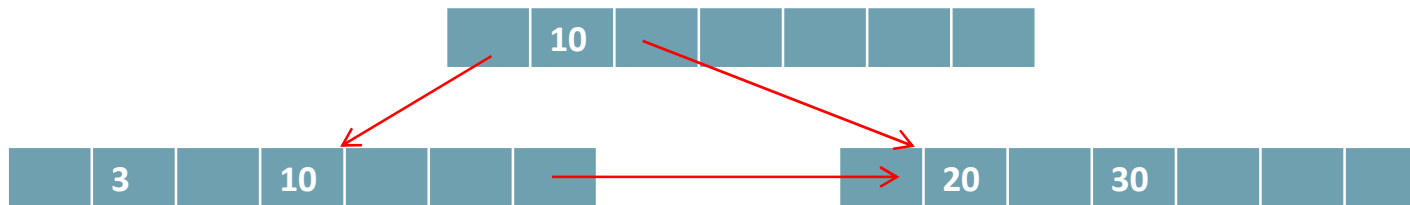
Creating B+ Tree

- Construct a B+ Tree of Order 4
10,20,30,3,6,9,15,27,32,34,48,

Insert 10, 20, 30



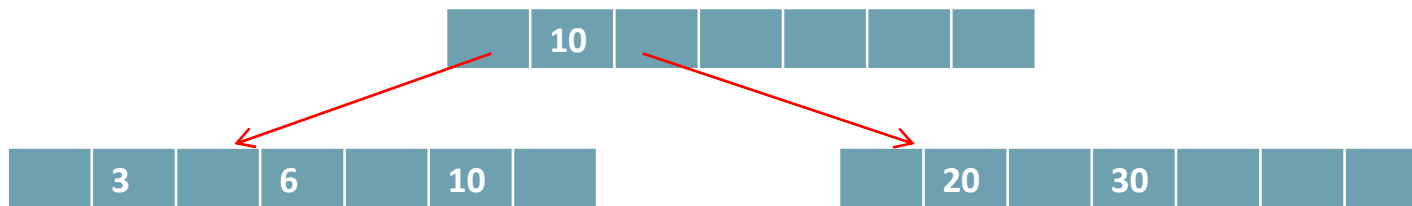
Insert 3



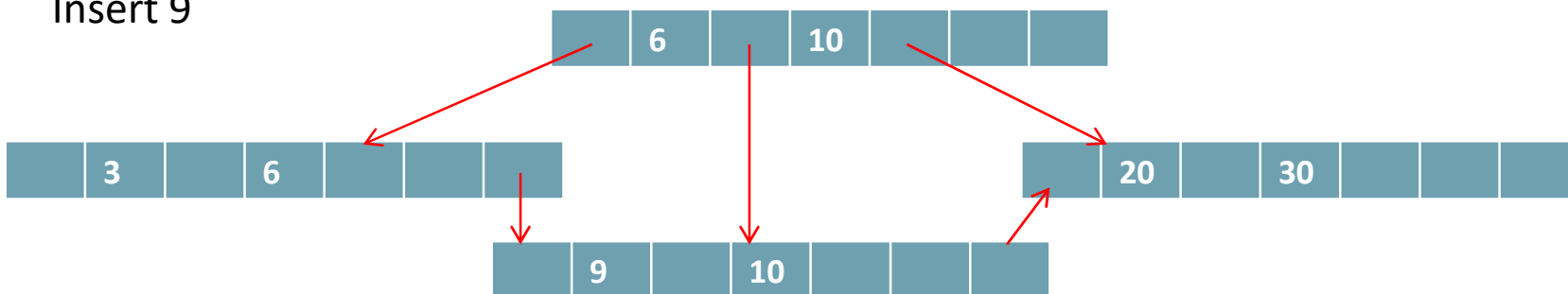
Creating B+ Tree

- Construct a B+ Tree of Order 4
10,20,30,3,6,9,15,27,32,34,48,

Insert 6



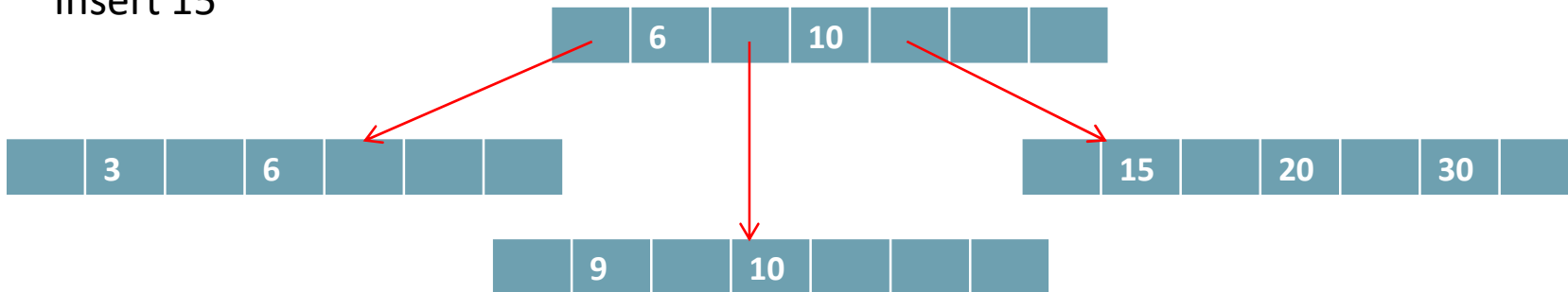
Insert 9



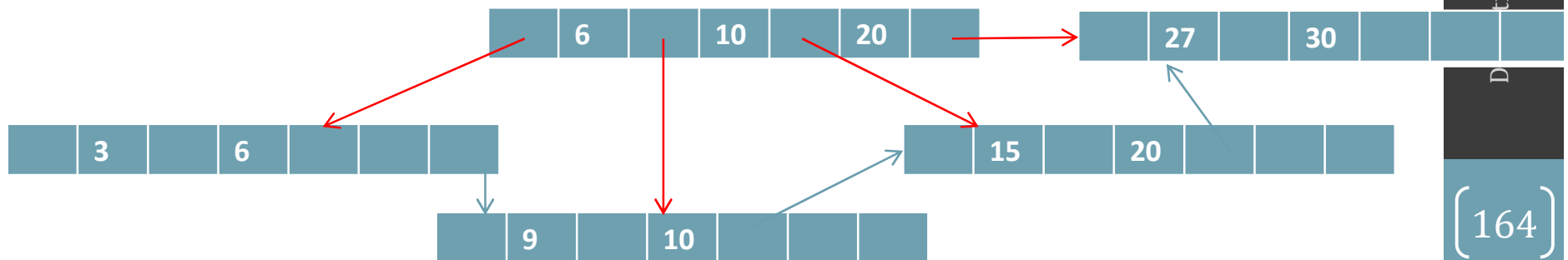
Creating B+ Tree

- Construct a B+ Tree of Order 4
10,20,30,3,6,9,15,27,32,34,48,

Insert 15



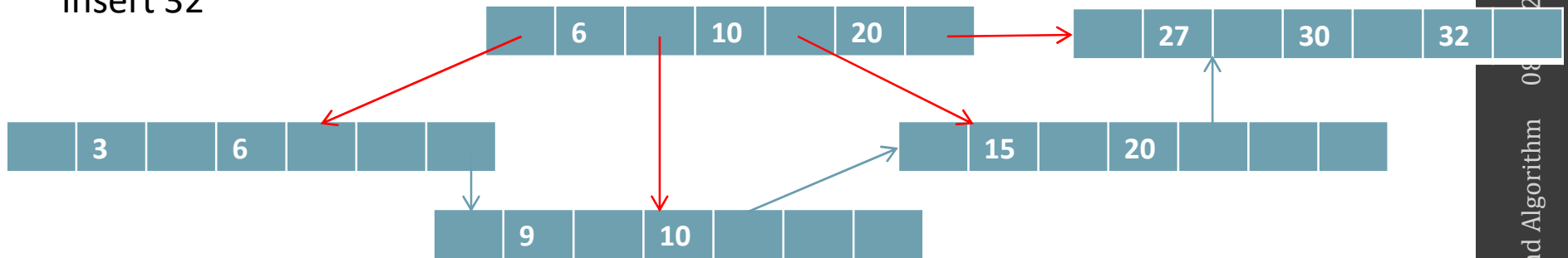
Insert 27



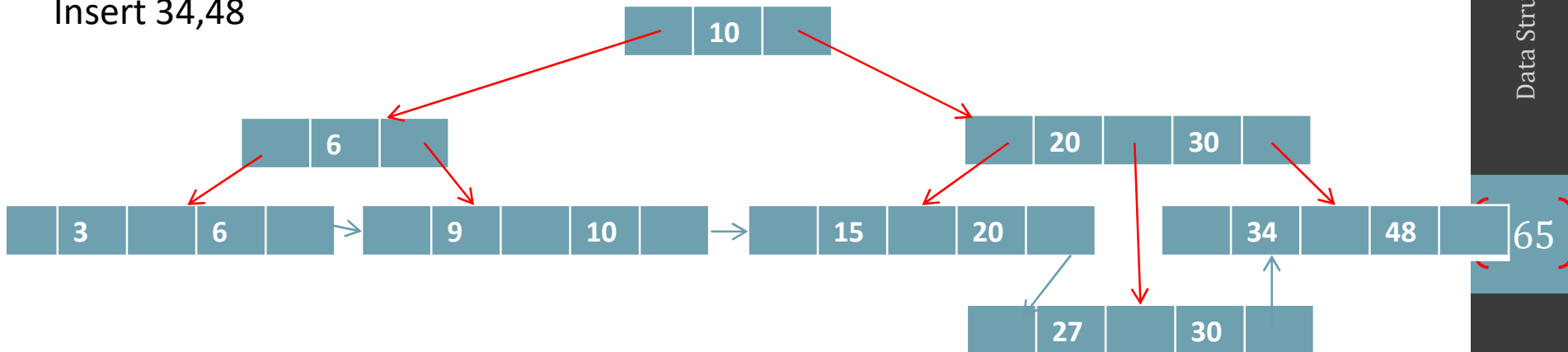
Creating B+ Tree

- Construct a B+ Tree of Order 4
10,20,30,3,6,9,15,27,32,34,48,

Insert 32



Insert 34,48



Kanak Kalyani

Advantages of B+ trees

1. Records can be fetched in equal number of disk accesses
2. It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level
3. Height of the tree is less and balanced
4. Supports both random and sequential access to records
5. Keys are used for indexing

Comparison Between B Trees and B+ Trees

B Tree	B+ Tree
<ol style="list-style-type: none">1. Search keys are not repeated2. Data is stored in internal or leaf nodes3. Searching takes more time as data may be found in a leaf or non-leaf node4. Deletion of non-leaf nodes is very complicated5. Leaf nodes cannot be stored using linked lists6. The structure and operations are complicated	<ol style="list-style-type: none">1. Stores redundant search key2. Data is stored only in leaf nodes3. Searching data is very easy as the data can be found in leaf nodes only4. Deletion is very simple because data will be in the leaf node5. Leaf node data are ordered using sequential linked lists6. The structure and operations are simple

- Create a B+ Tree of order 5
- 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42, 50