UNIT II

TREE STRUCTURES

What are trees?

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style without any closed region.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

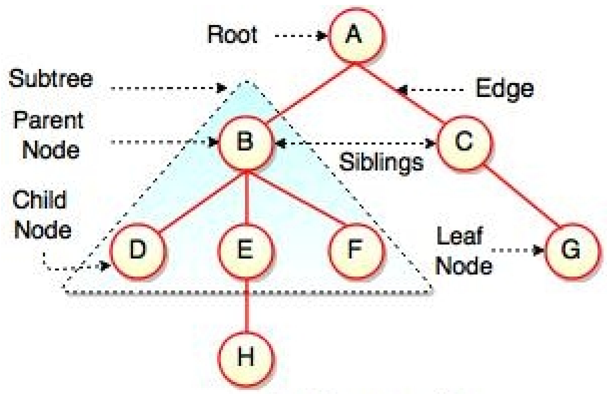
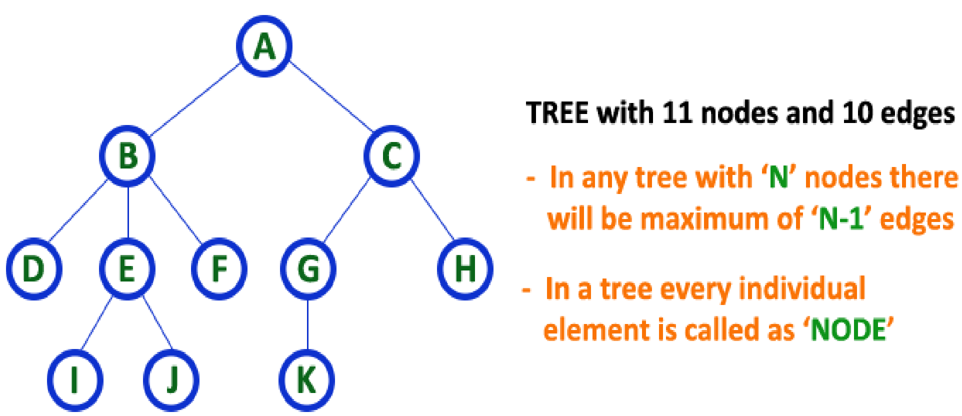


Fig. Structure of Tree

- The above figure represents structure of a tree. A is a parent of B and C. B is called a child of A and also parent of D, E, F
- Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

Example

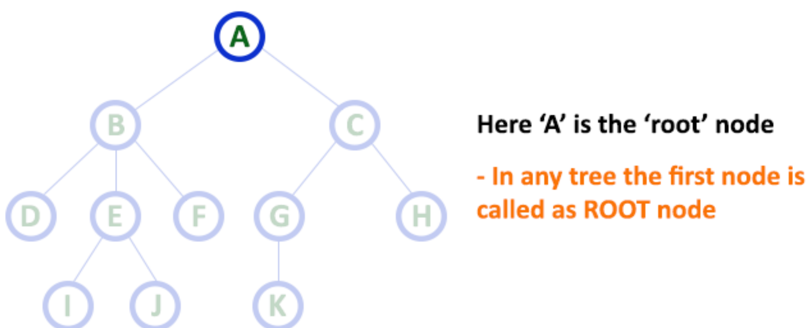


Tree Terminology

- Tree Terminologies are
  - Root
  - edge
  - parent
  - child
  - sibling
  - internal nodes
  - degree
  - height
  - depth
  - levels
  - path
  - subtree

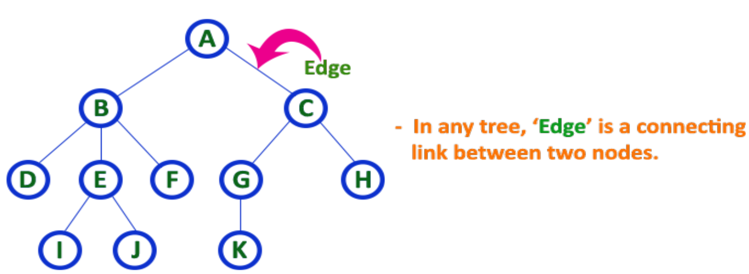
1. Root

- In a tree data structure, the first node is called as Root Node.
- Every tree must have a root node.
- We can say that the root node is the origin of the tree data structure.
- In any tree, there must be only one root node.
- We never have multiple root nodes in a tree.



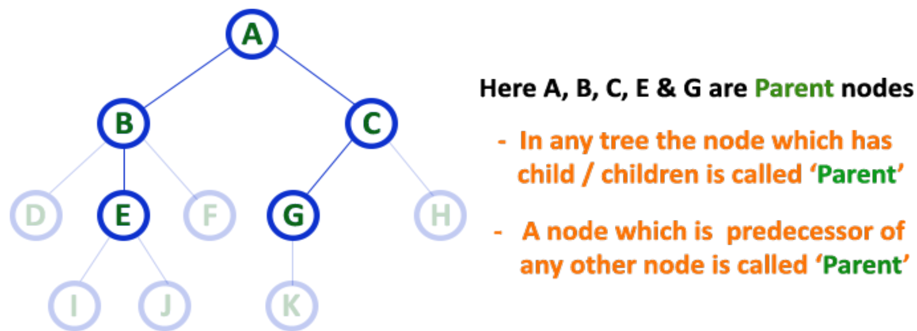
2. Edge

- In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



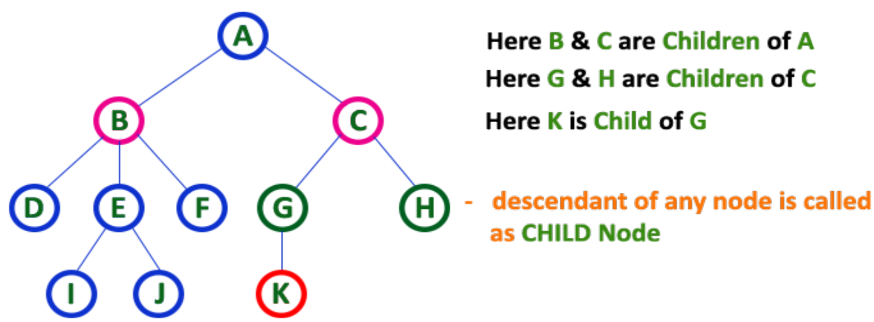
3. Parent

- The node which has a branch from it to any other node is called as a parent node.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



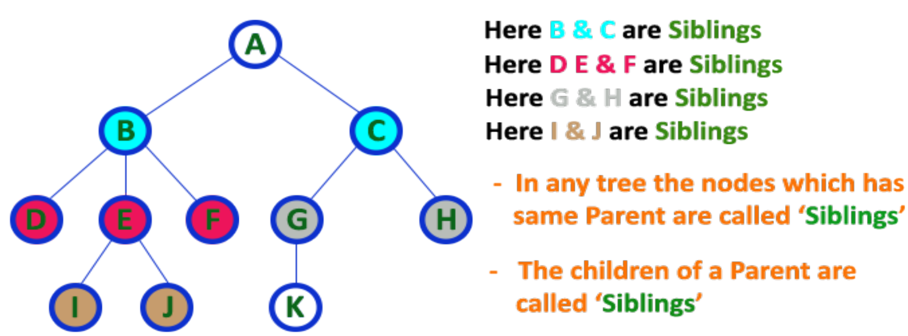
4. Child

- The node which is a descendant of some node is called as a child node.
- All the nodes except root node are child nodes.



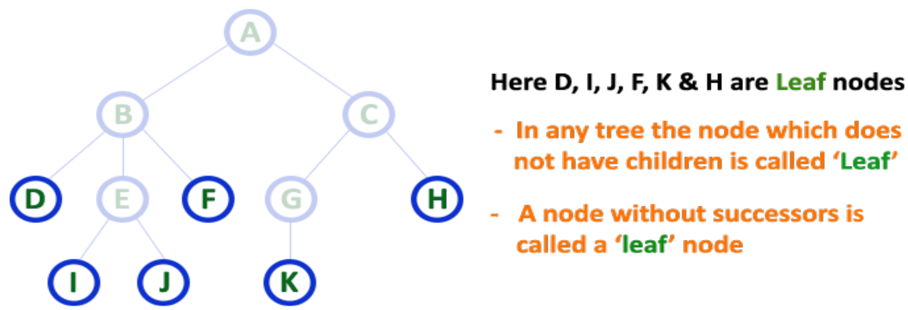
5. Siblings

- Nodes which belong to the same parent are called as siblings.
- In other words, nodes with the same parent are sibling nodes.



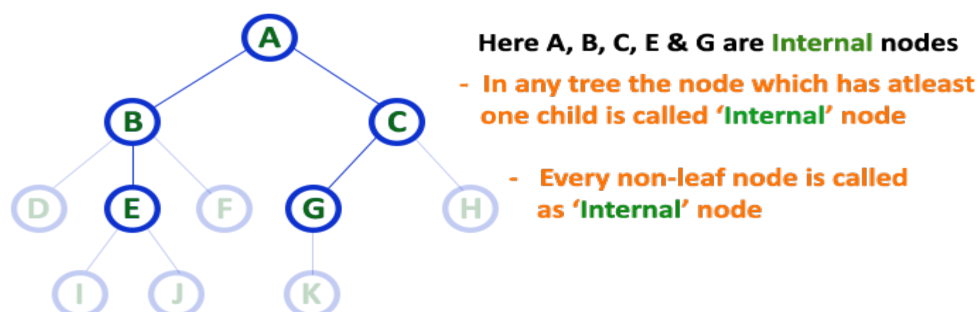
6. Leaf Node

- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



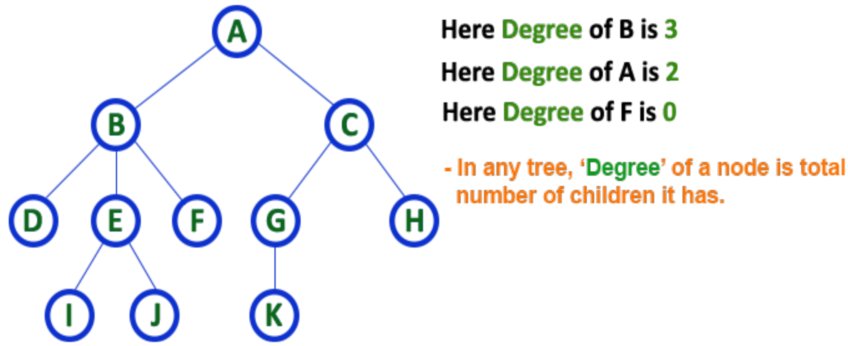
7. Internal Node

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



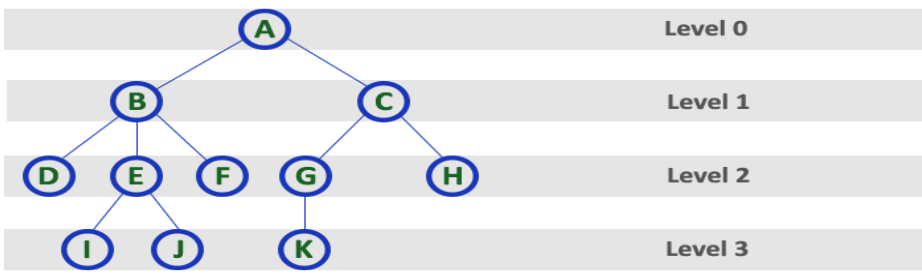
8. Degree-

- Degree of a node is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.
- in this example highest degree of a node is B is 3



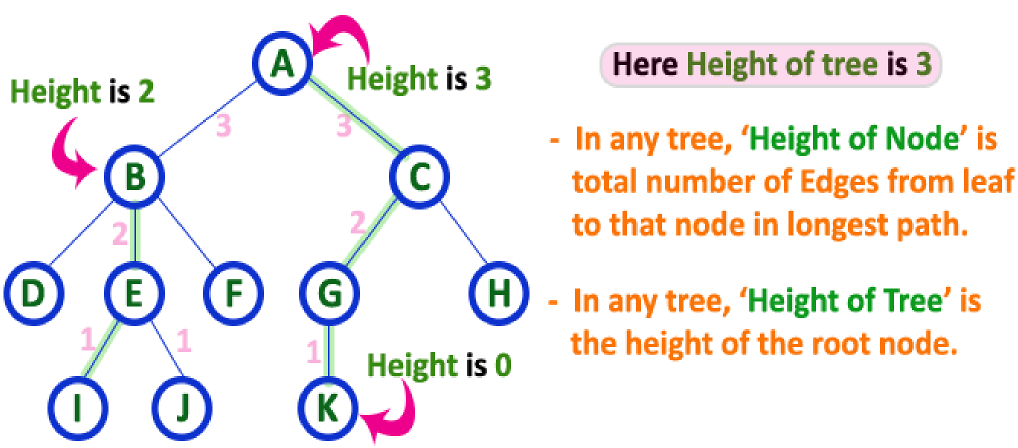
9. Level

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



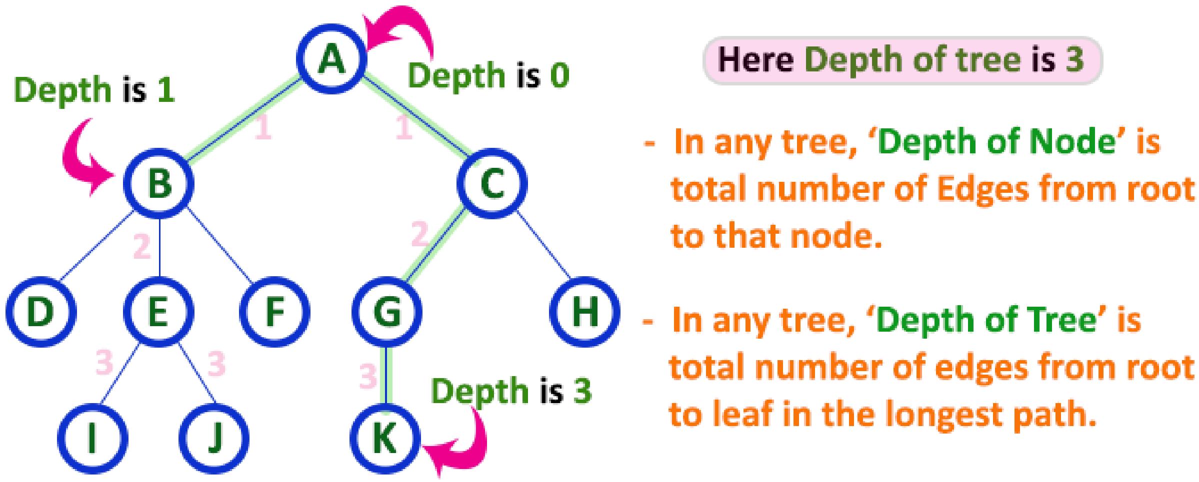
10. Height-

- Total number of edges that lies on the longest path from any **leaf node to a particular node** is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0



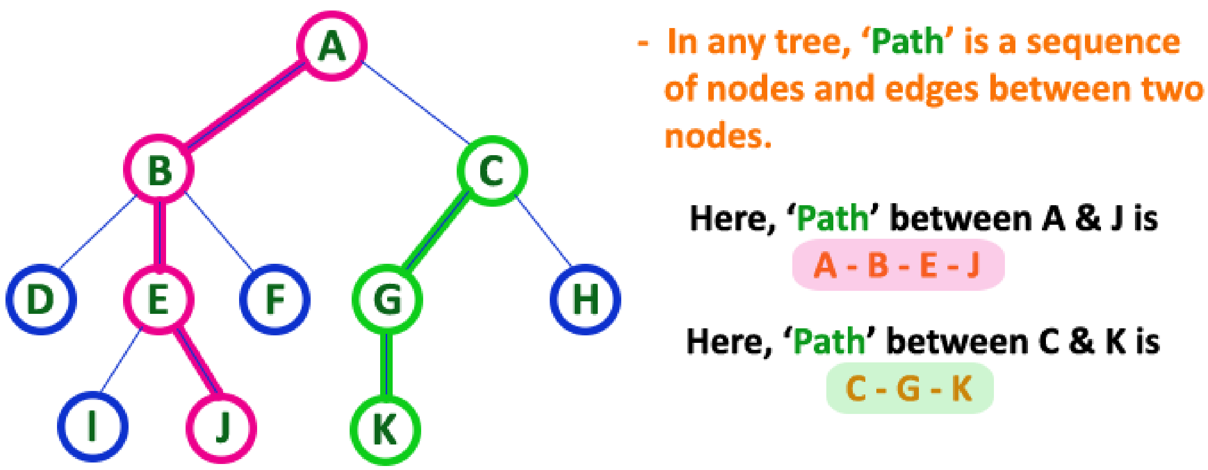
11. Depth

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



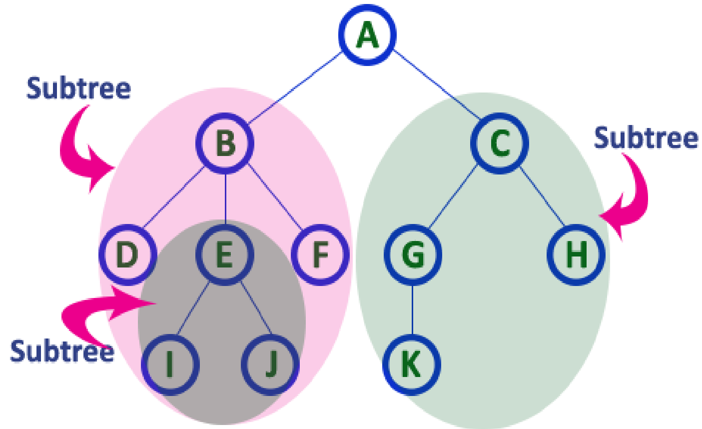
12. Path

- In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes
- **Length of a Path** is total number of nodes in that path.
- In below example **the path A - B - E - J has length 4**.



13. Subtree-

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.



Binary Tree

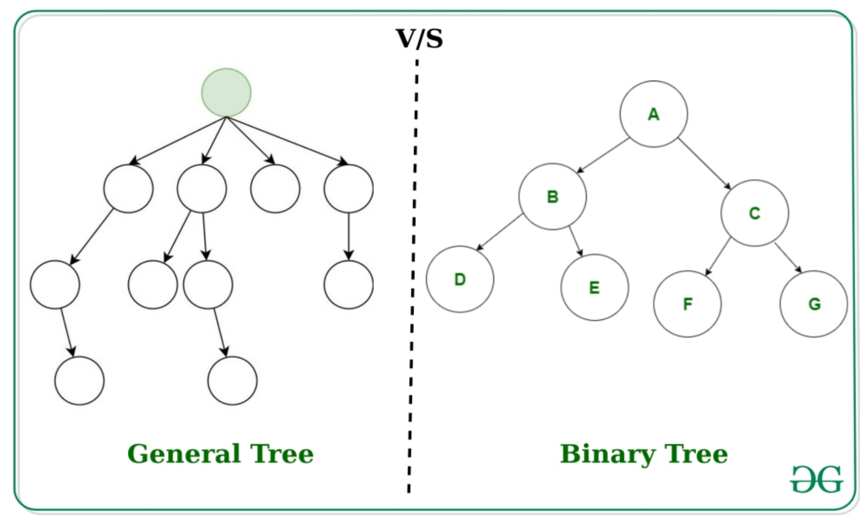
- In a **normal tree**, every node can have any number of children.
- A **binary tree** is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as a left child and the other is known as right child.

> **"A tree in which every node can have a maximum of two children is called Binary Tree".**

- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.
- Binary tree node declarations:

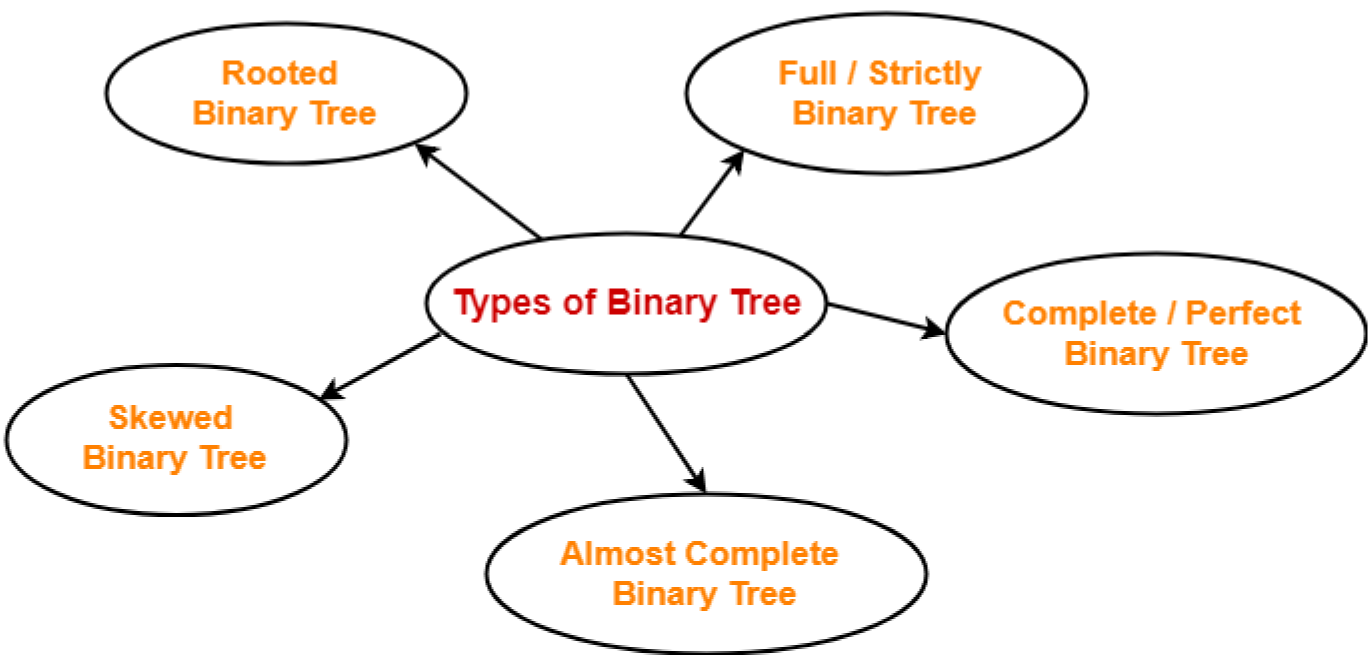
```
struct Tnode
{
int element;
struct Tnode *left;
struct Tnode *right;
};
```

Example



Types of Binary Trees

Binary trees can be of the following types-



1. Rooted Binary Tree

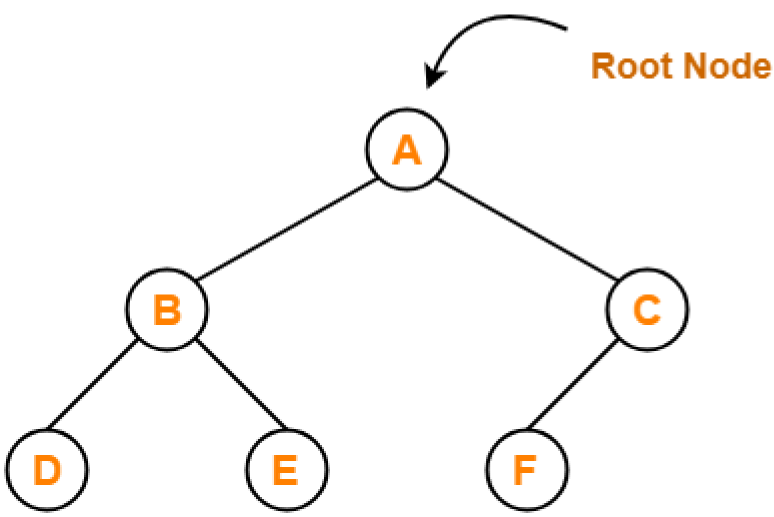
2. Full / Strictly Binary Tree
3. Complete / Perfect Binary Tree
4. Almost Complete Binary Tree
5. Skewed Binary Tree

1. Rooted Binary Tree-

A **rooted binary tree** is a binary tree that satisfies the following 2 properties-

- It has a root node.
- Each node has at most 2 children.

Example-

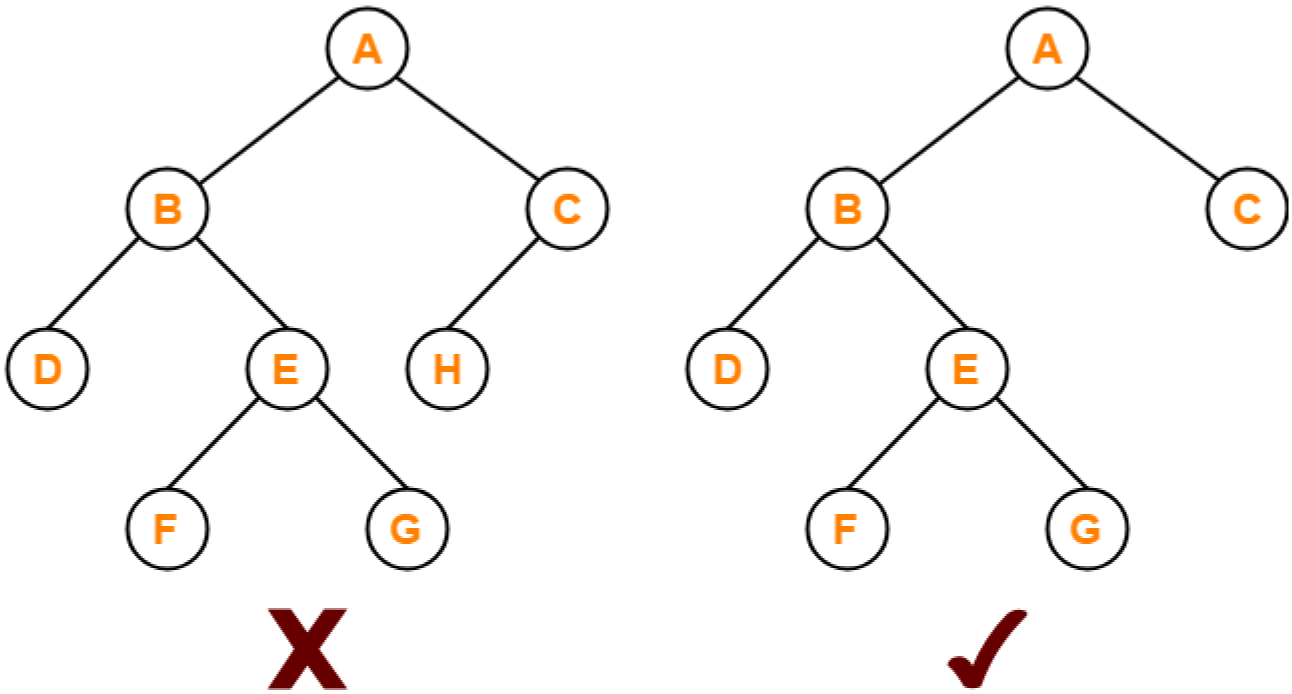


Rooted Binary Tree

2. Full / Strictly Binary Tree-

- A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.
- Full binary tree is also called as **Strictly binary tree**.

Example-



Here,

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

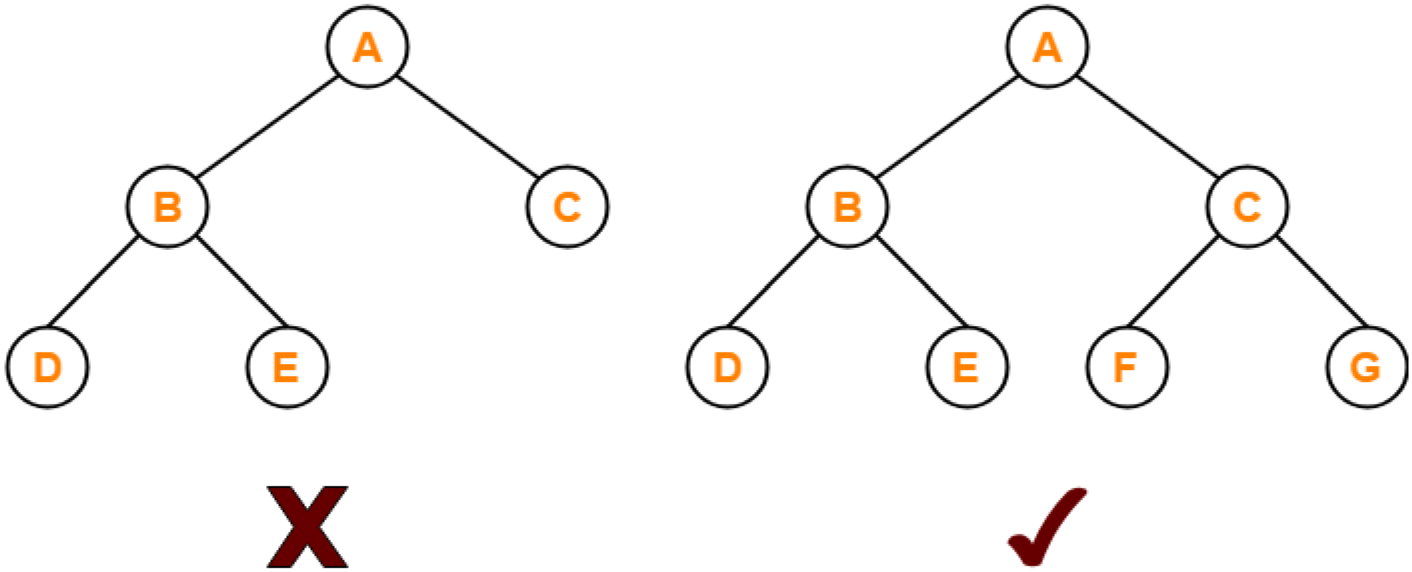
3. Complete / Perfect Binary Tree-

A **complete binary tree** is a binary tree that satisfies the following 2 properties-

- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.

Complete binary tree is also called as **Perfect binary tree**.

Example-



Here,

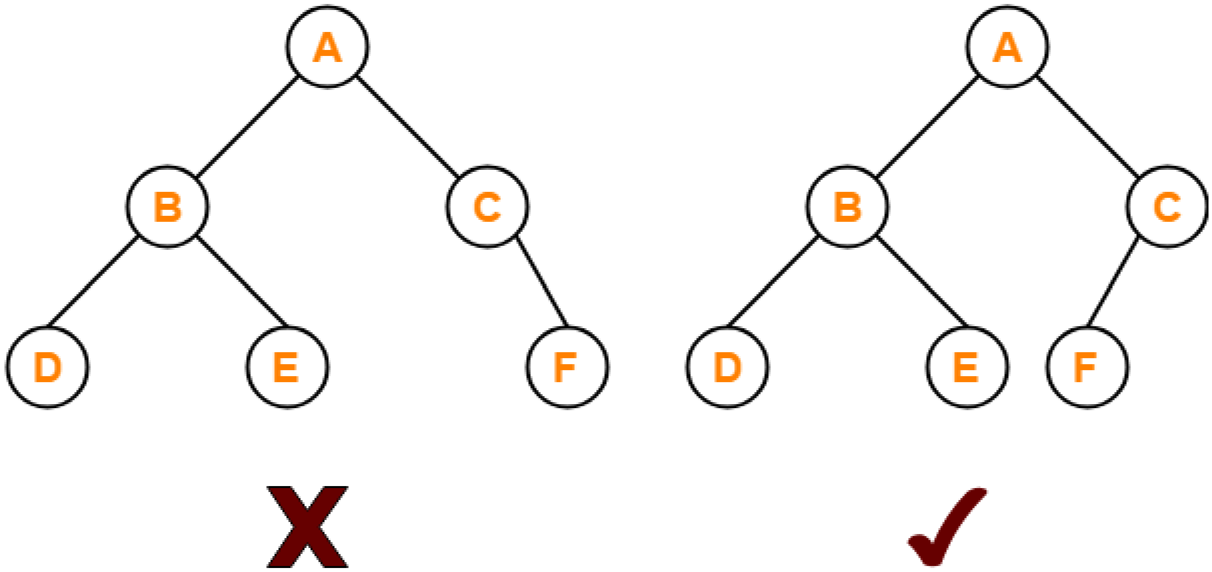
- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

4. Almost Complete Binary Tree-

An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.

Example-



Here,

- First binary tree is not an almost complete binary tree.
- This is because the last level is not filled from left to right.

5. Skewed Binary Tree-

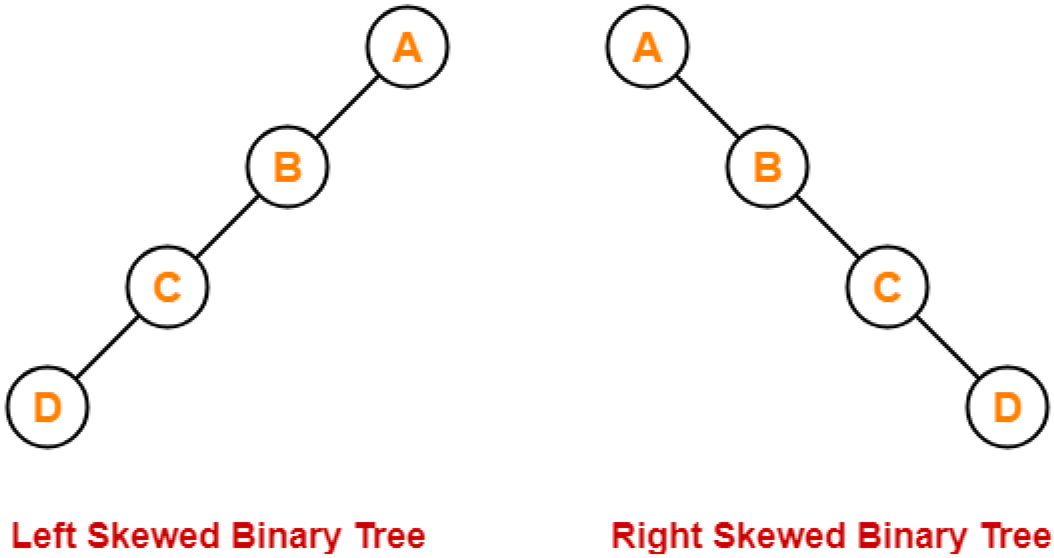
A **skewed binary tree** is a binary tree that satisfies the following 2 properties-

- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).

Example-



Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

Array Representation

- In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
- for any element in position i=1
- left child is in position 2i
- right child is in position 2i+1
- parent is in position i/2

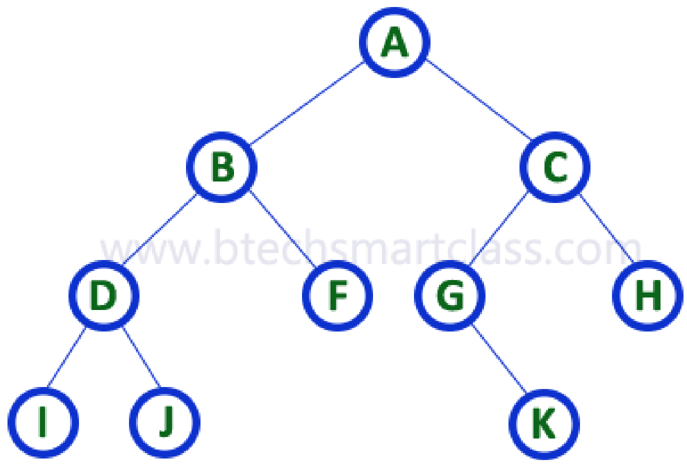
Steps:

1. Convert given binary tree into complete binary tree by adding empty node.

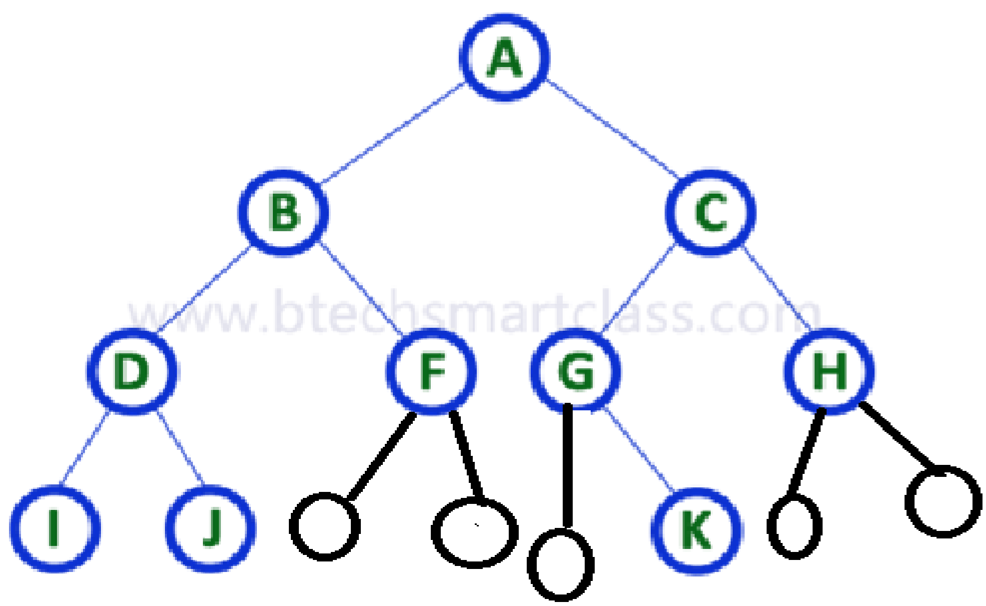
2. Give the index of the every node from root to leaf.

3. Size of the array is equal to No. of node in the CBT.

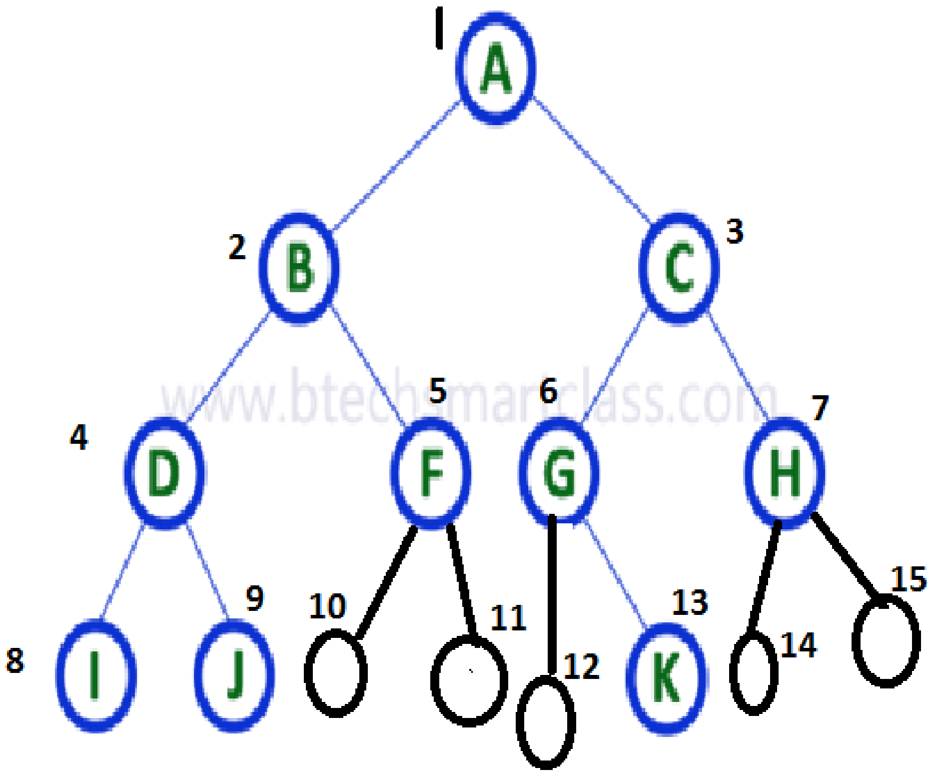
Example:



1. Convert given binary tree into complete binary tree by adding empty node.



2. Give the index of the every node from root to leaf.



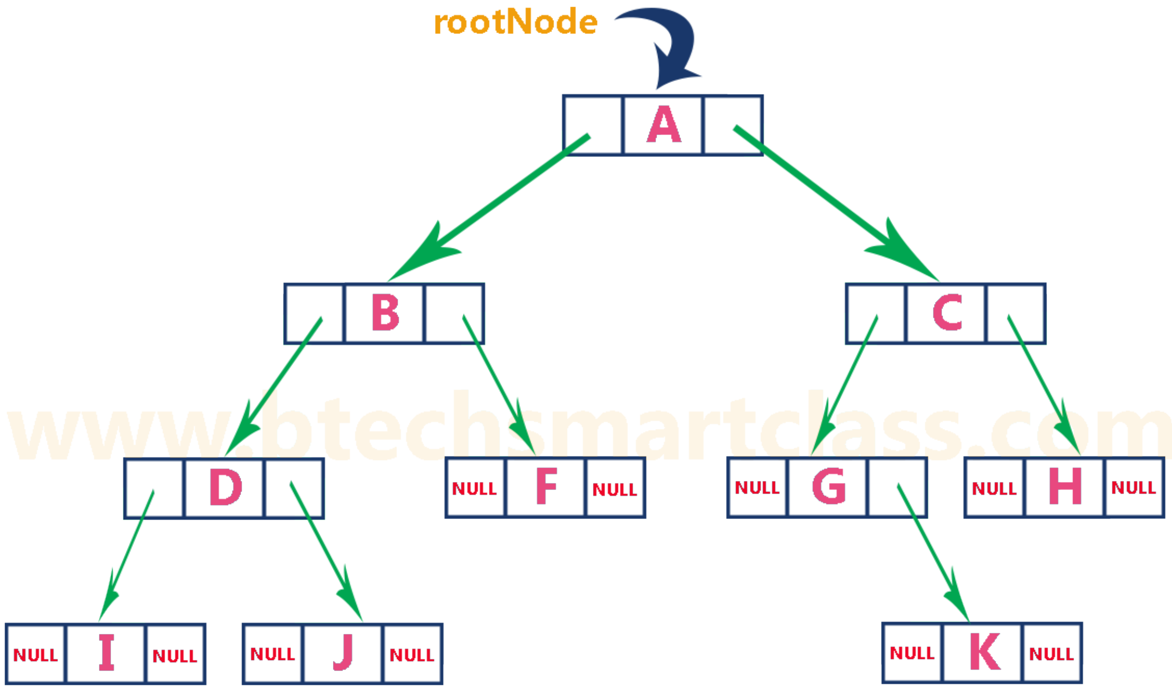
Size of the array =15

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-

<u>Linked List Representation of Binary Tree</u>

- We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields.
- First field for storing left child address, second for storing actual data and third for storing right child address.
- In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...



<u>Tree Applications</u>

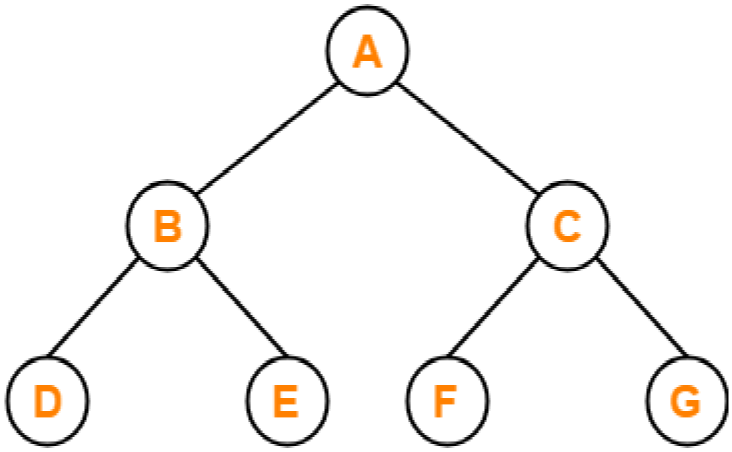
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and B+-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Example:

Calculate the no of nodes with binary tree of height 2.

- binary tree of height h has 2^{h+1}-1 nodes
- Here height is 2
- therefore No.of nodes in complete binary tree is=2^{2+1} -1

=7 nodes.



Tree Traversals

- Traversing means visiting each nodes only ones.
- Tree Traversals is a method for visiting all the nodes in the tree exactly once.
- all nodes are connected via edges (links) we always start from the root (head) node.
- That is, we cannot randomly access a node in a tree.
- **There are three types of tree traversals**
  1. In-order Traversal
  2. Pre-order Traversal
  3. Post-order Traversal

In-order Traversal (L r R)

In this traversal method,

- First, visit all the nodes in the left subtree
- Then the root node
- Visit all the nodes in the right subtree

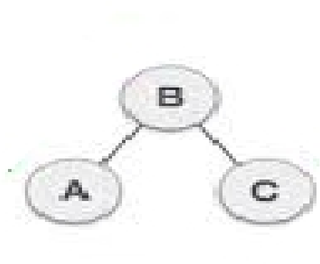
Preorder traversal(rLR)

- Visit root node
- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree

Postorder traversal(LRr)

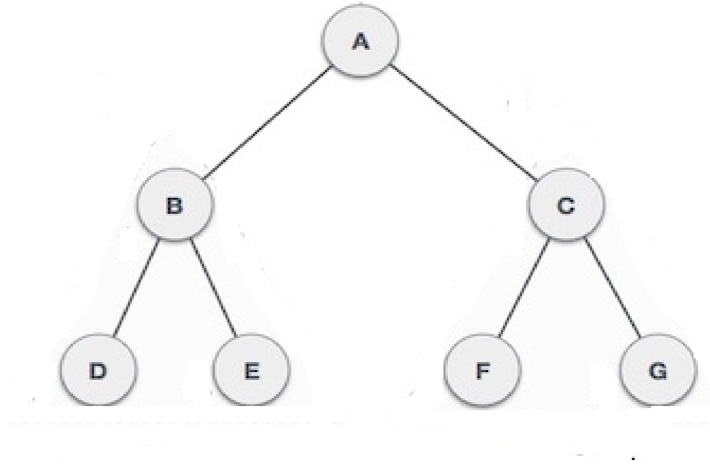
- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree
- Visit the root node

Example:1

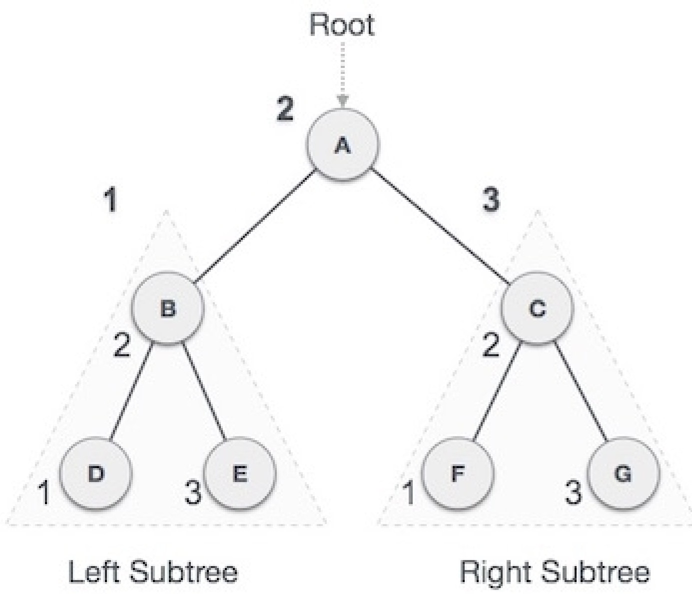


Inorder: A B C (L r R)
Preorder: BAC (r L R)
Postorder: ACB (L R r)

Example: 2

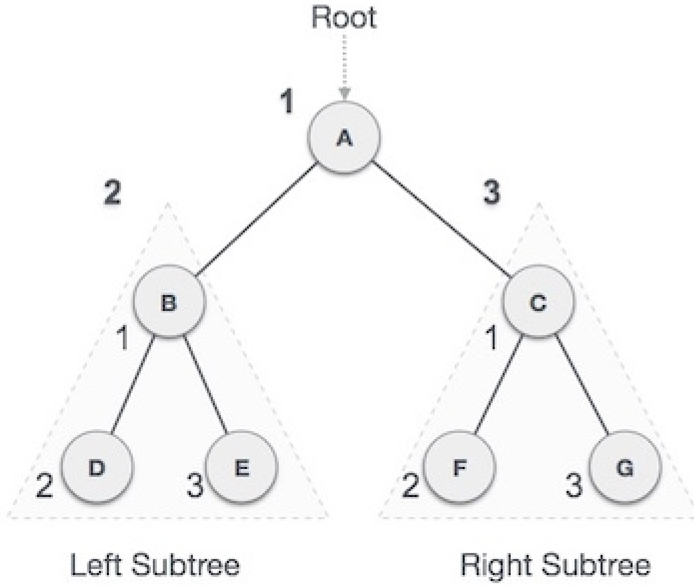


Inorder (L r R)



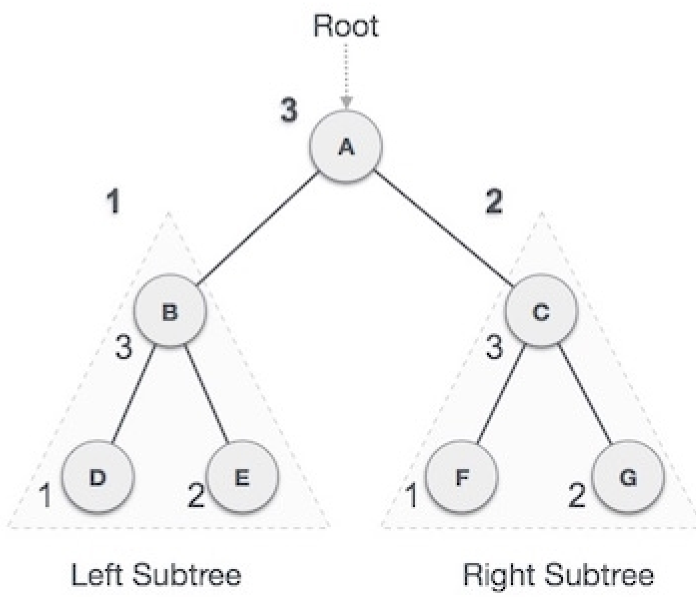
D B E A F C G

Preorder   (r L R)



A   B D E C F G

<u>Postorder   (L R   r)</u>



D E B FG C A

Recursive routine for inorder traversal:

```
void inorder(node *T)
{
if (T!=null)
{
inorder(T->left);
printf("%d", T->data);
inorder(T->right);
}
```

Recursive routine for preorder traversal:

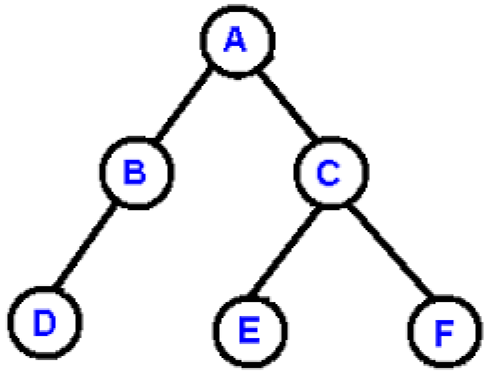
```
void preorder(node *T)
{
if (T!=null)
{
printf("%d", T->data);
preorder(T->left);
preorder(T->right);
}
```

Recursive routine for postorder traversal:

```
void postorder(node *T)
{
if (T!=null)
{
postorder(T->left);
postorder(T->right);
printf("%d", T->data);
}
```

Other example

1. Traverse the given tree using inorder, preorder and post order

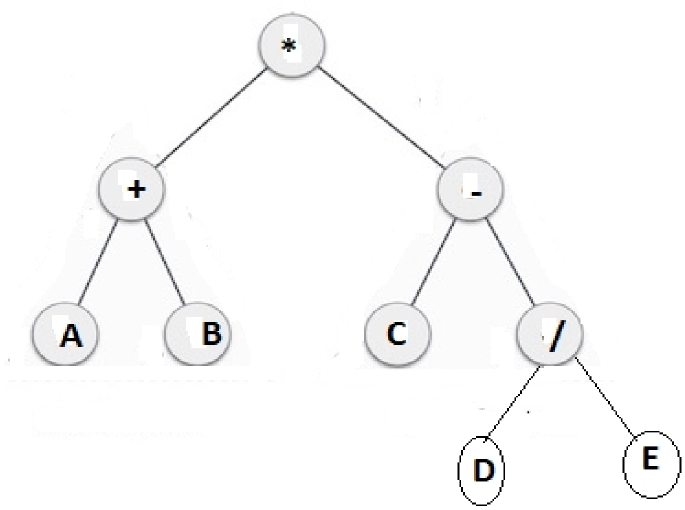


INORDER(L r R): D B A E C F

PRE ORDER (r L R): A B D C E F

POST ORDER(L R r): D B E F C A

2. Traverse the given tree using inorder, preorder and post order

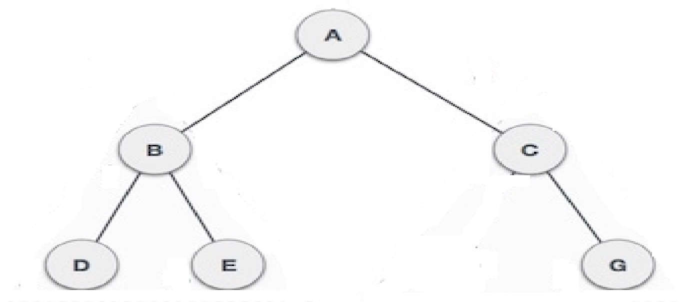


INORDER: A+B*C-D/E

PREORDER: *+AB-C/DE

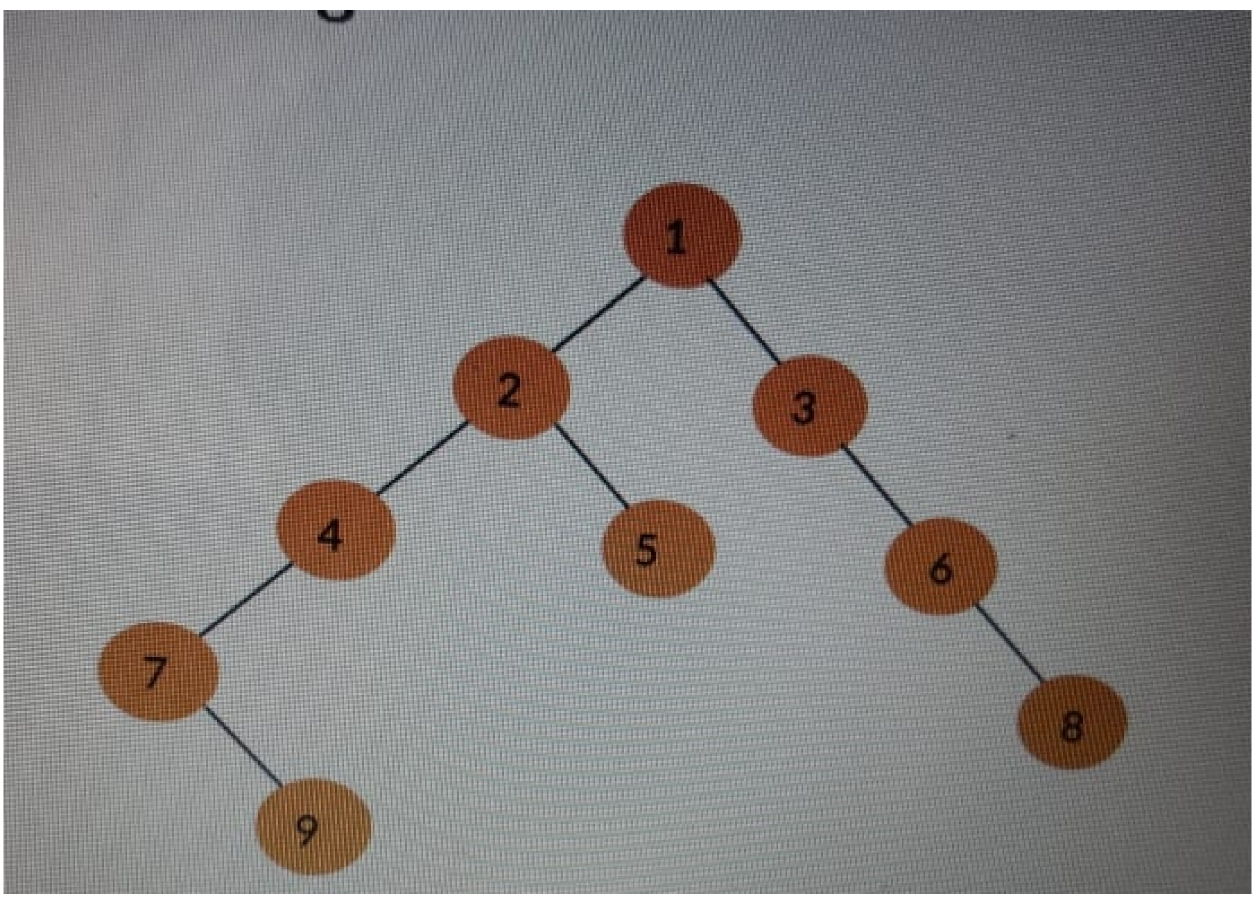
POST ORDER: AB+CDE/-*

3. Traverse the given tree using inorder, preorder and post order



DBEACG   ABDECG   DEBGCA

4. Traverse the given tree using inorder, preorder and post order



794251368

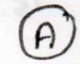
construction of Tree with inorder and Preorder

Example: 1

Preorder : A B C ⟶ ΥLR

Inorder : B A C ⟶ LΥR

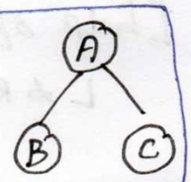
step1:



[in preorder 1st element is Root] so, construct the node.

[help of inorder to find L & R element]

step2 :



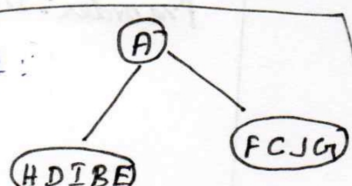
Example:2

Preorder : ABDHIECFGJ – ΥLR

inorder : $\underbrace{HDIBE}_{LST}$A$\underbrace{FCJG}_{RST}$. – LΥR.

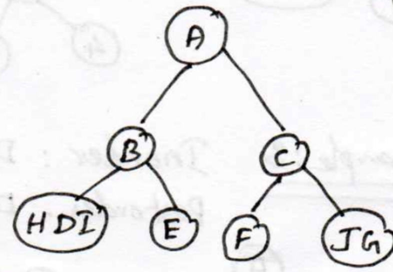
step1

Step2 :

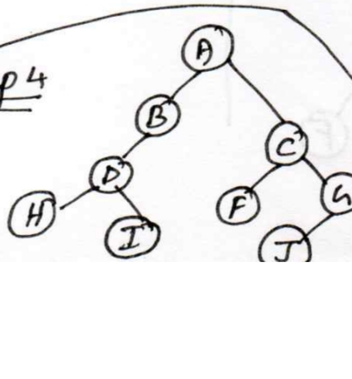


find the root

step3 ⟹



step4



⟹ Final Answer

Construction of Tree with inorder and Post order ②

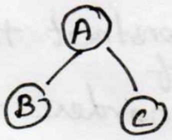
Example : 1

Postorder : B C $\underline{\underline{A}}$ → L R r

Inorder : B $\underline{\underline{A}}$ C. → L r R.

step1:

step2:



[in postorder last element is root so construct root node].

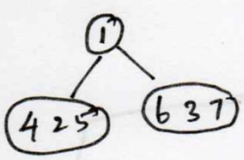
[help of inorder to find L & R element].

Example:2

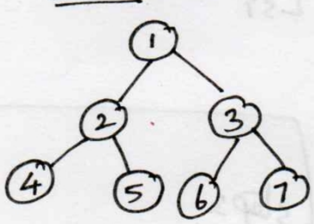
Inorder: 4 2 5 $\underline{1}$ 6 3 7

Postorder: 7 6 3 5 4 2 $\underline{\underline{1}}$ — Root

step1:



step 2



other example

① Inorder: HDIBEAFCJG
Postorder: HIDEBFJGCA

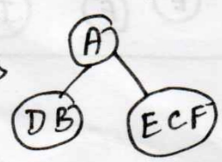
② Inorder: ABCDEGHIJK.
Preorder: DCABIGEHKJ

Example:3

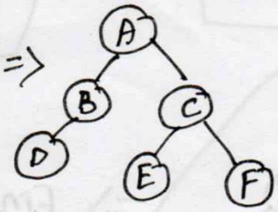
Inorder : DBAECF

Postorder. DB EFCA

① ⇒



② ⇒



Expression tree

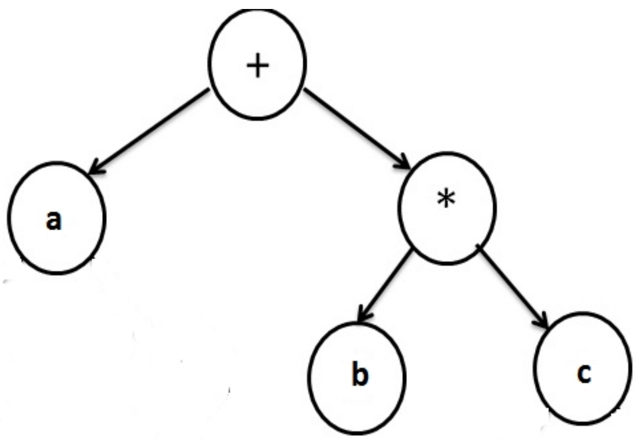
- Expression Tree is used to represent expressions and one of the applications of tree.
- **Expression Tree** is a binary tree in which the **leaf nodes are operands and the internal nodes are operators**. Like binary tree, expression tree can also be traversed by inorder, preorder and postorder

Example: a+b*c

^, ( ) R-L

***, / L-R**

+, - L-R



Infix: a+b*c

prefix : +a*bc

postfix: abc*+

- Expression Tree is a special kind of binary tree with the following properties:
  - Each leaf is an operand. Examples: a, b, c, 6, 100
  - The root and internal nodes are operators. Examples: +, -, *, /, ^
  - Subtrees are subexpressions with the root being an operator.
- An expression and expression tree shown below

```
a + (b * c) + d * (e + f)
```

Example (without stack - direct)

$a + (b * c) \pm d * (e + f)$

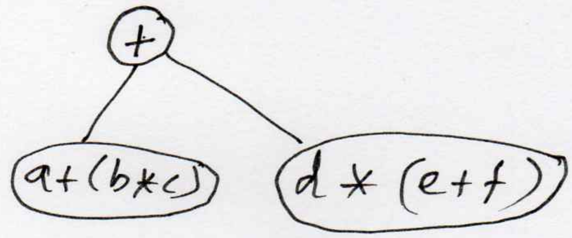
^ () = R, L

* / = L, R

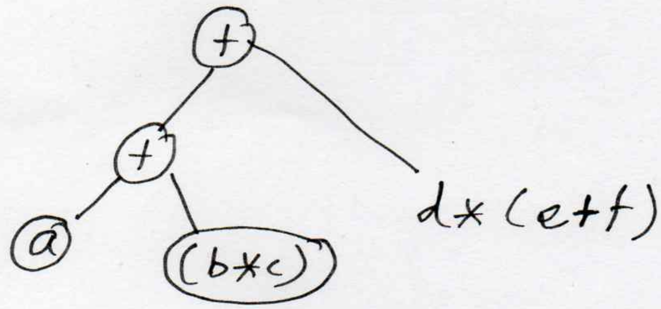
+ - = L, R.

go Low to high

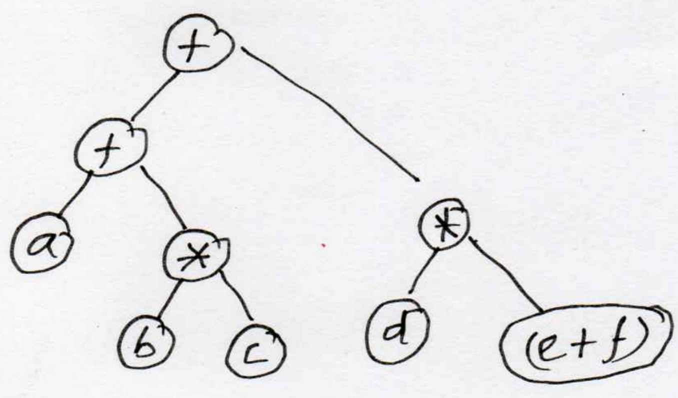
1.



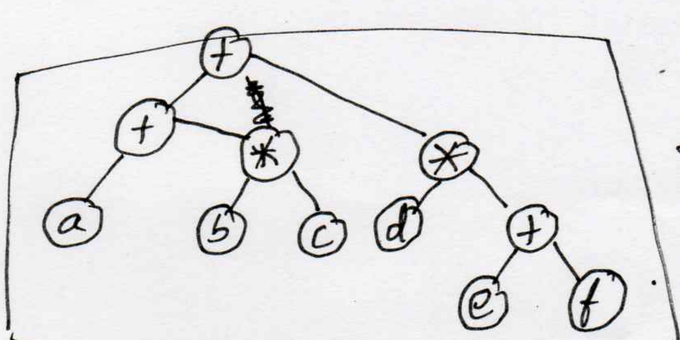
2.



3.



4.



final Result.

Construction of Expression Tree

We consider that a **postfix expression is given as an input for constructing an expression tree**. Following are the step to construct an expression tree:

1. Read one symbol at a time from the postfix expression.
2. Check if the symbol is an operand or operator.
3. If the symbol is an operand, create a one node tree and pushed a pointer onto a stack
4. If the symbol is an operator, pop two pointer from the stack namely T_1 & T_2 and form a new tree with root as the operator, T_2 as a left and T_1 as right child
5. A pointer to this new tree is pushed onto the stack

Example

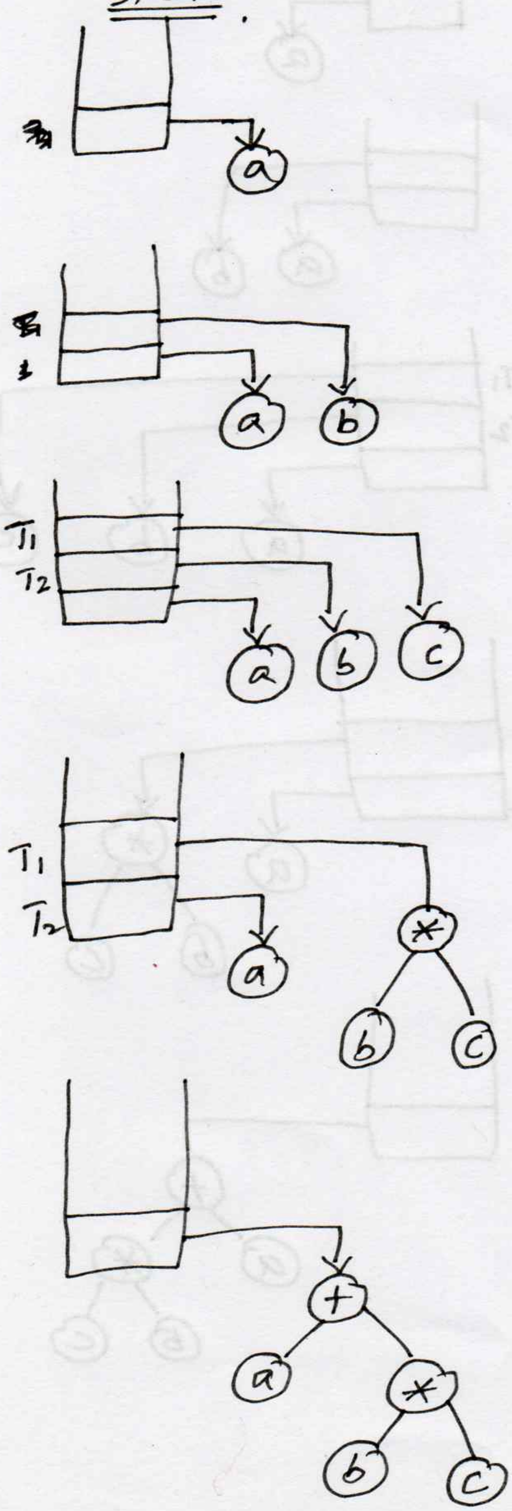
Construction of Expression Tree

Example:

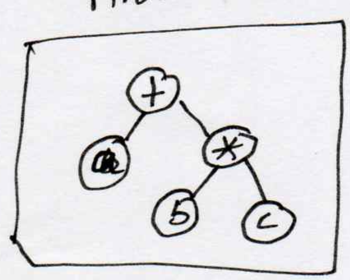
a b c * +

steps	symbol
1.	a
2.	b
3.	c
4.	*
5.	+

stack

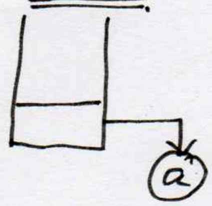
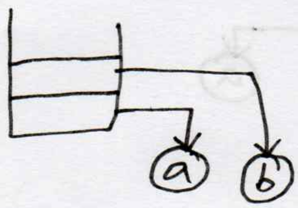
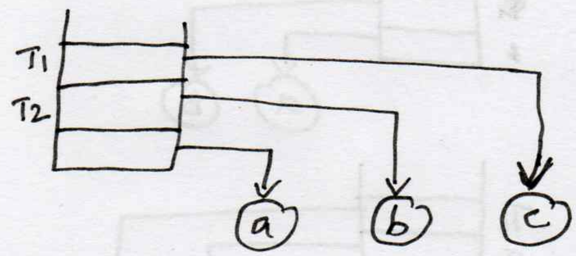
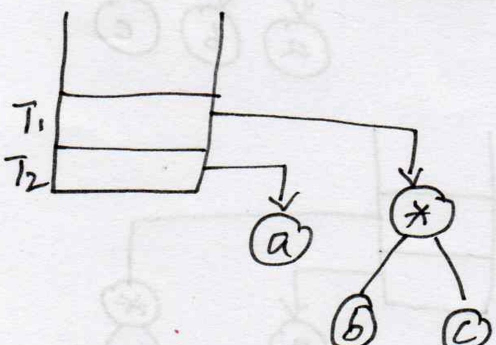
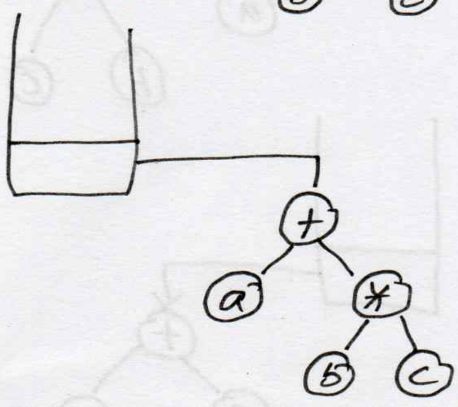


Final Result.

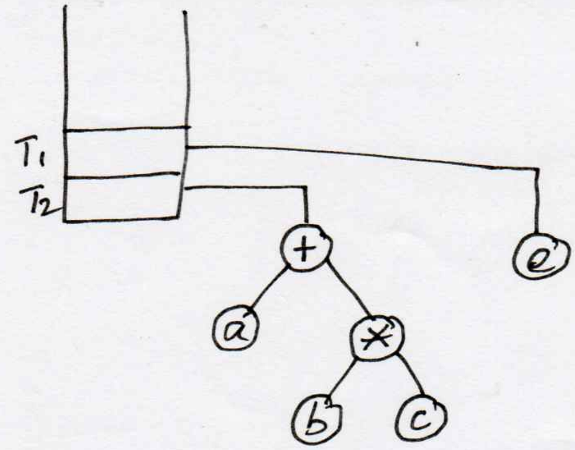


2. Example

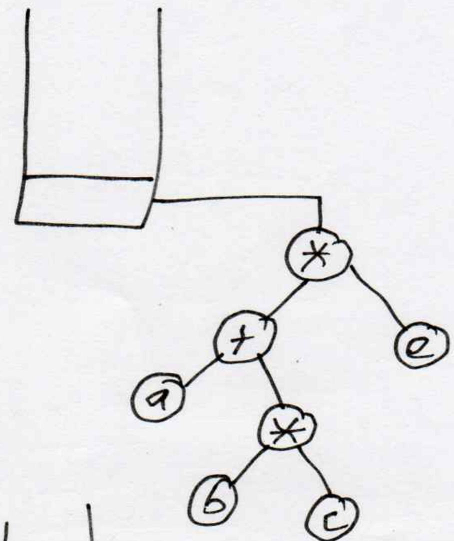
abc * + e * f + postfix expression

steps	symbol	stack
1.	a	
2.	b	
3.	c	
4	*	
5.	+	

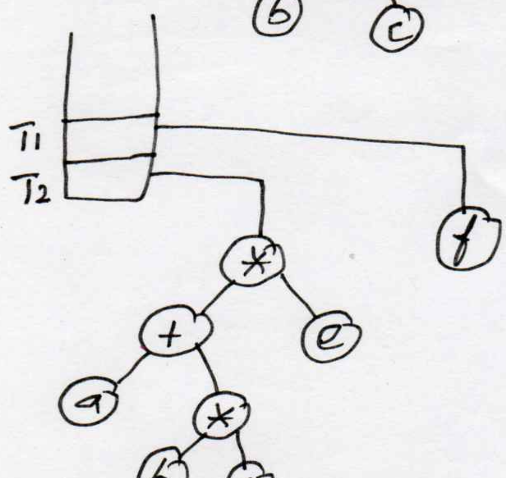
6. e



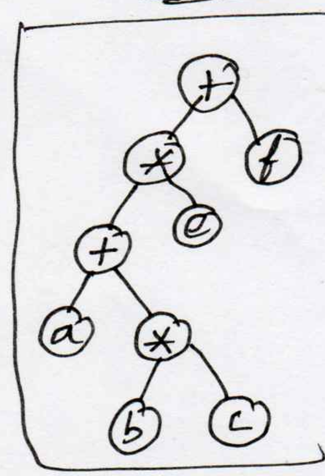
7. *



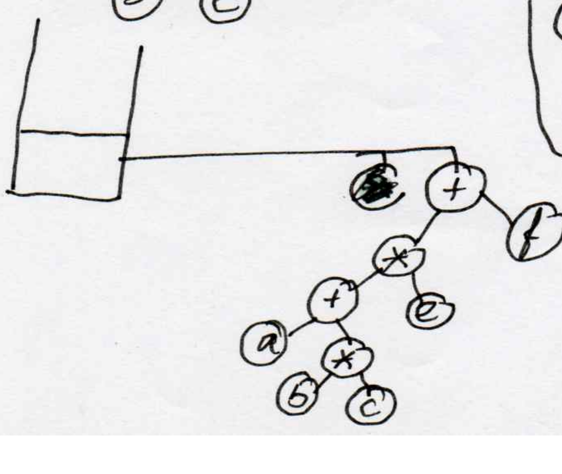
8. f



Final Result.



9. +



Other example:(H.W)

1. ab+c*

2.12*c+

3. ab+cde+**

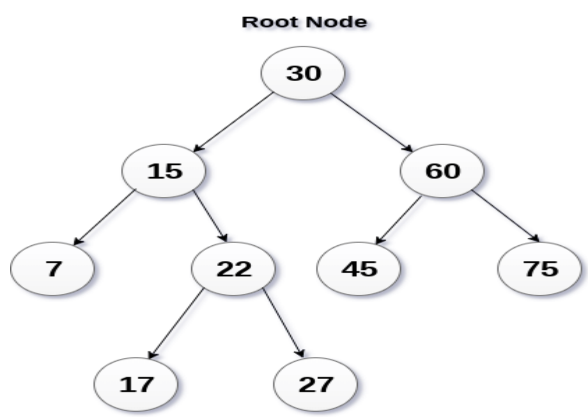
Binary Search Tree

- Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
- In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
- This rule will be recursively applied to all the left and right sub-trees of the root.

NOTE:

1. Every binary search tree is a binary tree.

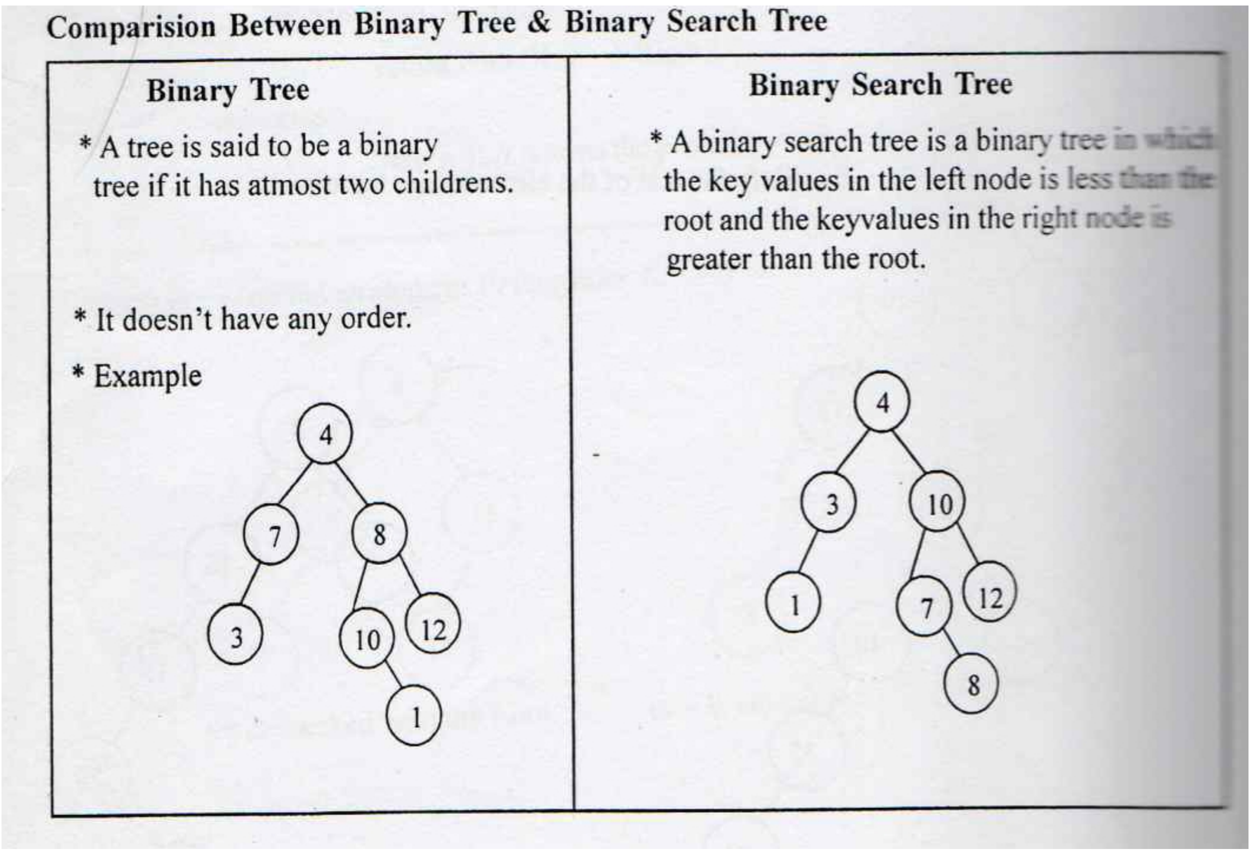
2. All binary trees need not be a binary search tree.



Binary Search Tree

Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step.
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.


Comparision Between Binary Tree & Binary Search Tree

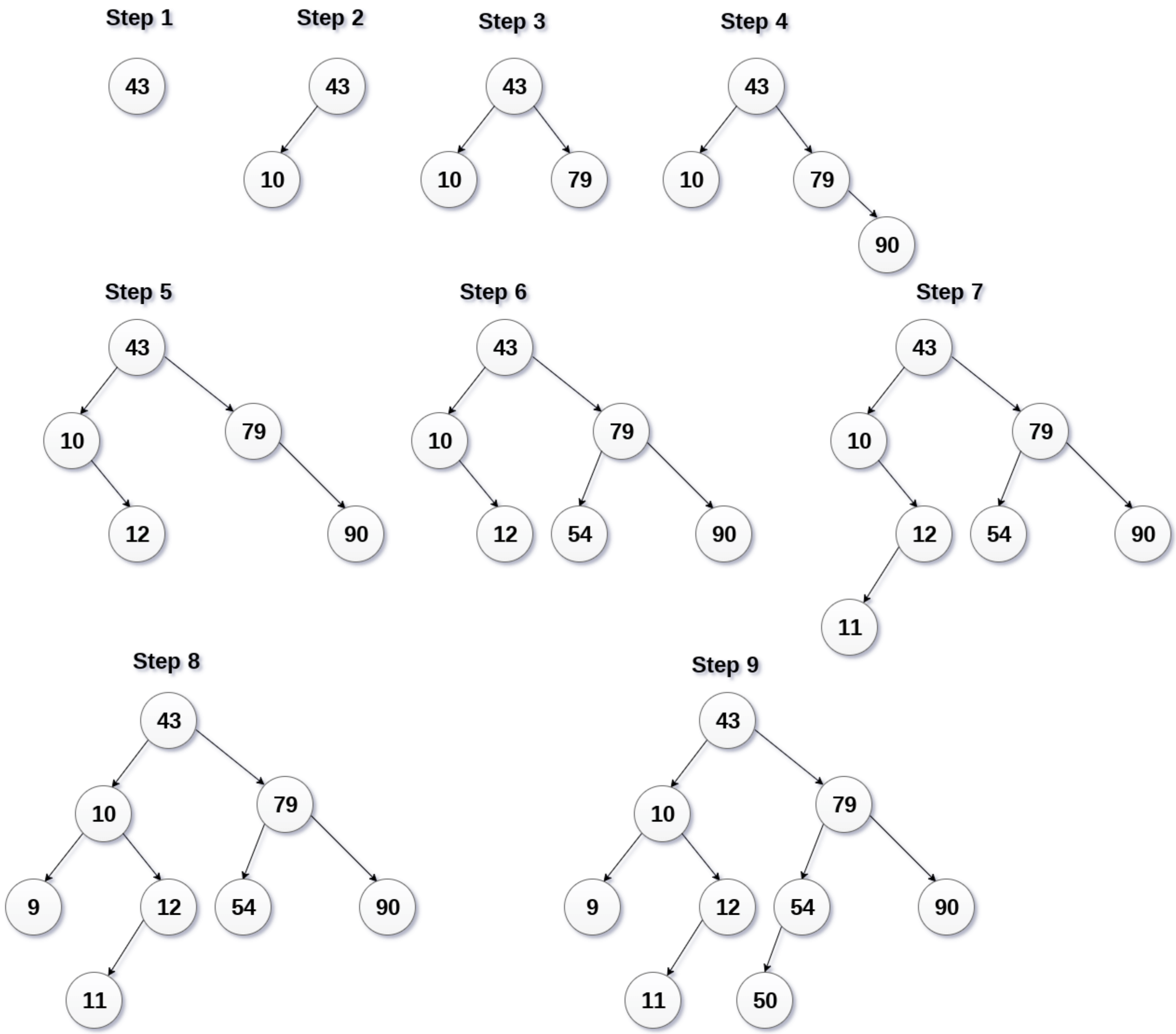
Binary Tree	Binary Search Tree
* A tree is said to be a binary tree if it has atmost two childrens. * It doesn't have any order. * Example	* A binary search tree is a binary tree in which the key values in the left node is less than the root and the keyvalues in the right node is greater than the root.

1. Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Other example: H.W

1. 7,2,9,0,5,6,8,1

2. 10,3,15,22,6,45,65,23,78,34,5

Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

The value of the key of the left sub-tree is less than the value of its parent (root) node's key.

The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as

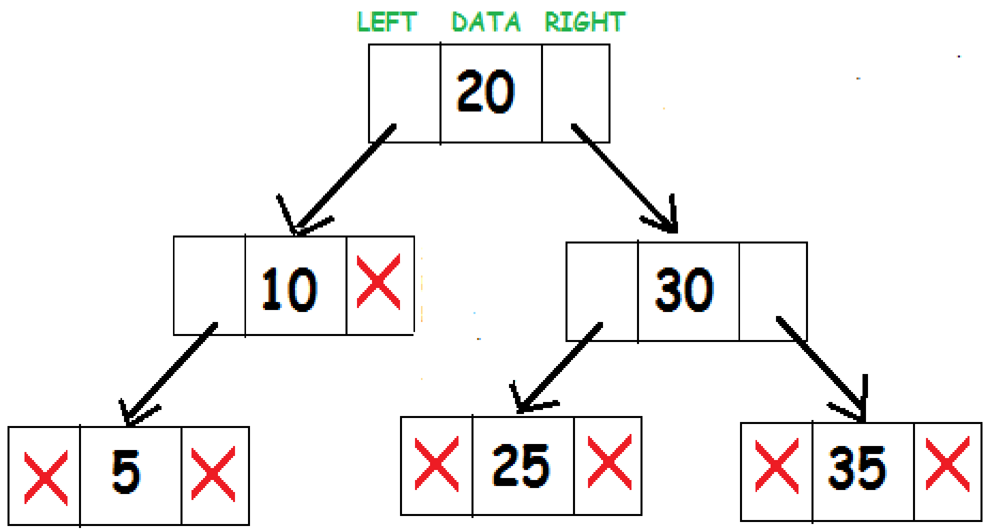
Basic Operations:

1. Insertion
2. Deletion
3. Find
4. Find max
5. Find min
6. Make empty

Declaration of binary search tree:

Define a node having some data, references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
} struct node *root=null;
```



Insert Operation

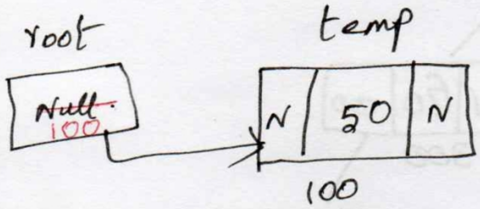
- Whenever an element is to be inserted, first locate its proper location.
- Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data.
- Otherwise, search for the empty location in the right subtree and insert the data.

```
void insert(int d)
{
  struct node * temp *current,* parent;
 struct node *temp = (struct node*) malloc(sizeof(struct node));
  temp->data = d;
  temp->left = NULL;
  temp->right = NULL;
 if(root == NULL)   //if tree is empty
{
     root = temp;
   }
else
 {
    current = root;
    while(current)
{
        parent = current;
        if(temp->data > current->data) //go to RIGHT of the tree
 {
          current = current->right;
}
else                //go to LEFT of the tree
{
          current = current->left;
 }
if (  temp->data > parent->data )
{
          parent ->right=temp;
}
else
{
               parent->left=temp;

}

}
```

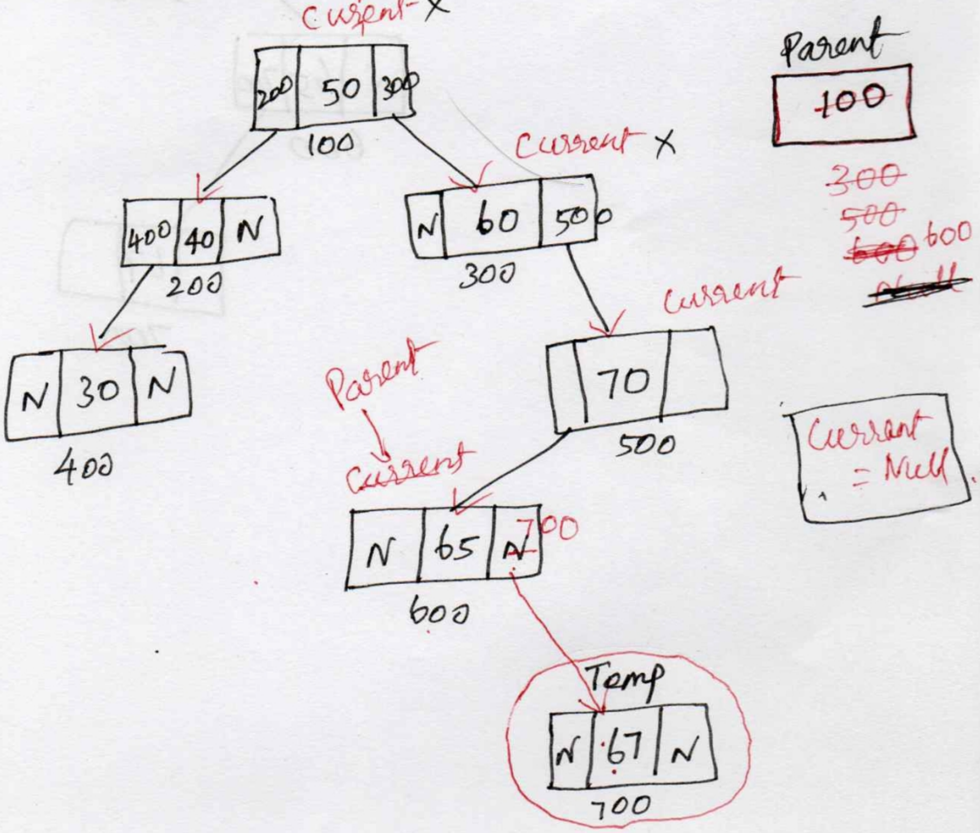
Binary Search Tree

Insertion

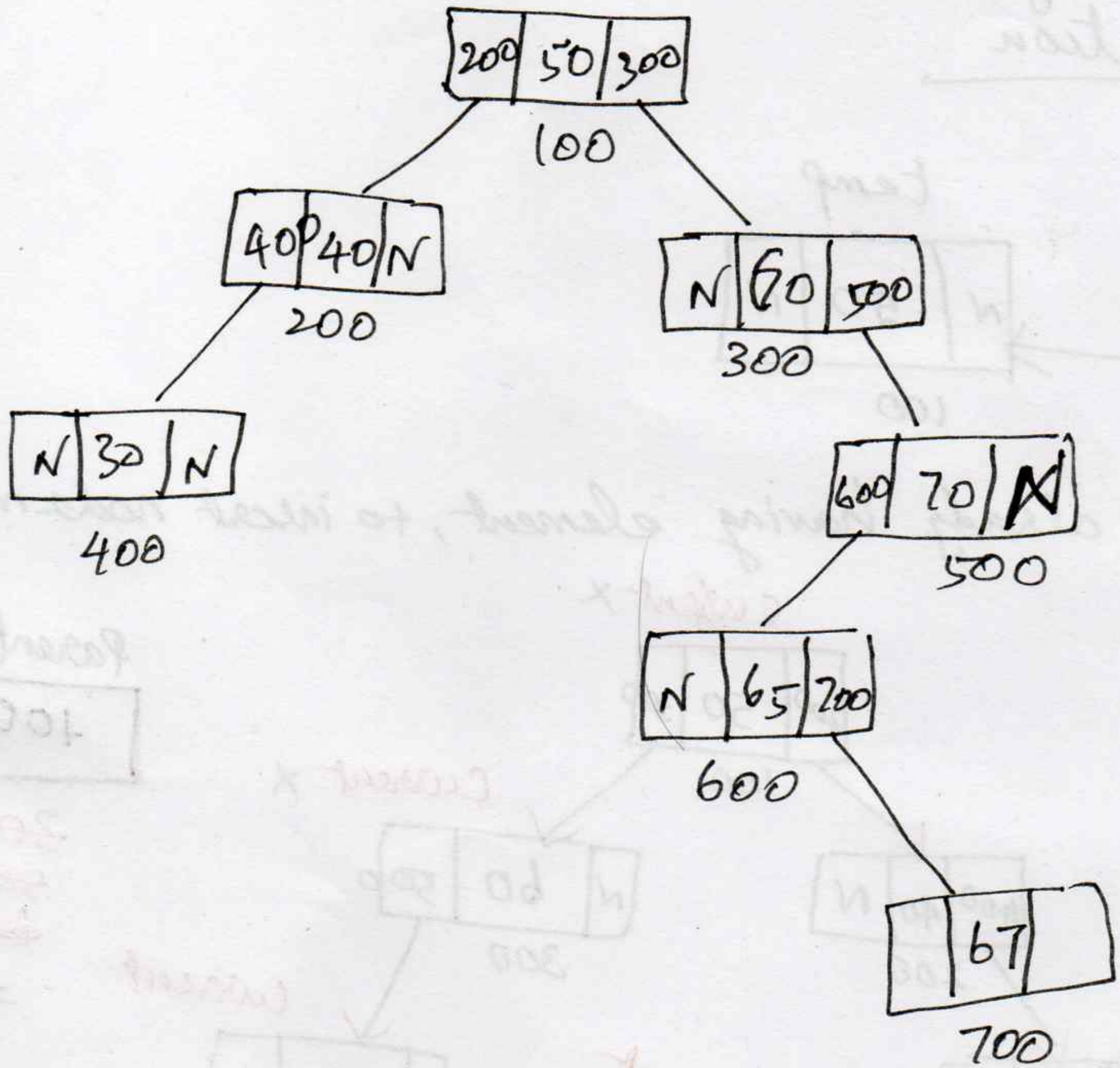
1



else (already having element, to insert node means)

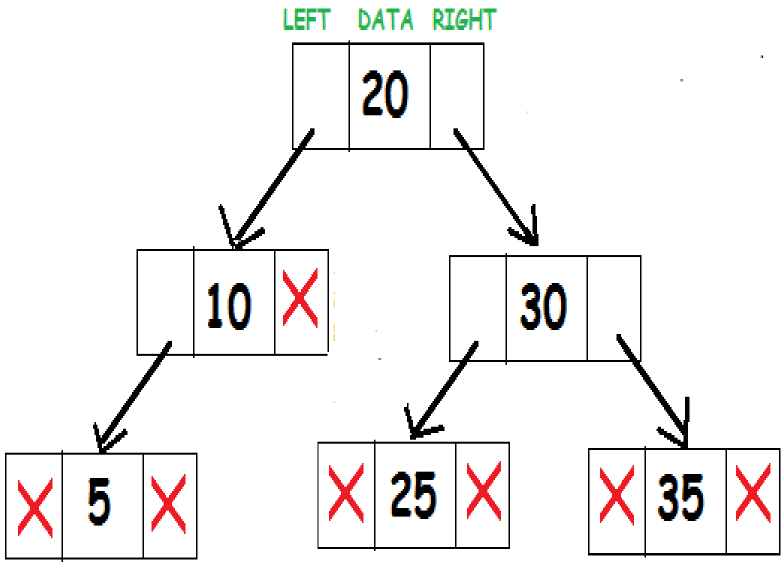


Final O/P



200
50
300
100
40
40
N
200
N
60
500
300
N
30
N
400
600
70
N
500
N
65
200
600
67
700

FIND OPERATION: (SEARCH)



- Whenever an element is to be searched, start searching from the root node.
- Then if the data is less than the key value, search for the element in the left subtree.
- Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

```
struct node* find(int data)
{
struct node *current = root;
while(current!= NULL)
{  35=35
  if(data == Current->data)
{

return (current)
}  35>35
if(data  > current->data)
{
  current = current->right;
  }
else
{
current = current->left;
}
```

```
}
  return current;
}
```

Find minimum

- Whenever an smallest element is to be searched, start searching from the root node.
- Then if the data is less than the key value, find for the element in the left subtree.
- This stopping point is smallest data.

```
struct node* find(int data)
{
struct node *current = root;
while(current!=null)
{
if(current->left!=null)
{
current=current->left;
}
return(current)
}
```

Find maximum

- Whenever an largest element is to be searched, start searching from the root node.
- Then if the data is greater than the key value, find for the element in the right subtree.
- This stopping point is smallest data.

```
struct node* find(int data)
{
struct node *current = root;
while(current!=null)
{
if(current->right!=null)
{
current=current->right;
}
return(current)
}
```

Make empty

- Delete every node of the tree.
- It also release memory occurred on the node.

```
makeempty(node *T)
if(T!=null)
{
makeempty(T->left);
makeempty(T->right);
free(T);
}
return(null)
}
}
```

Binary Search Tree

Deletion operation

* It is complex operation in the BST.
* To delete an element, consider the 3 cases.

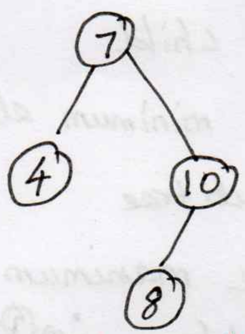
case 1: Deleting a node with no children (leaf)

case 2: Deleting a node having one child

case 3: Deleting a node having two children.

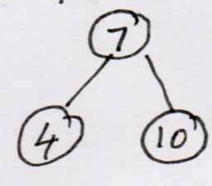
case 1: Deleting a node with no children (leaf node)

* if node is a leaf node, it can be deleted immediately.



Before deletion

Delete : 8 :



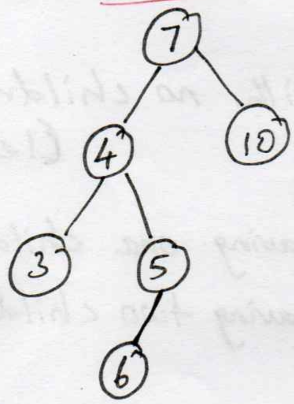
After deletion

case 2: Deleting a node having one child

* if the node has one child, it can be deleted adjusting its parent pointer that points to its child node.

> " child will Replace the position of parent "

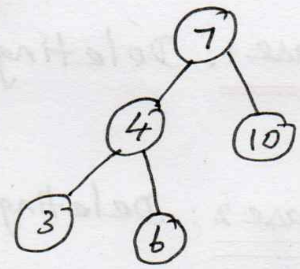
Before deletion



Delete 5.

⟹

After deletion



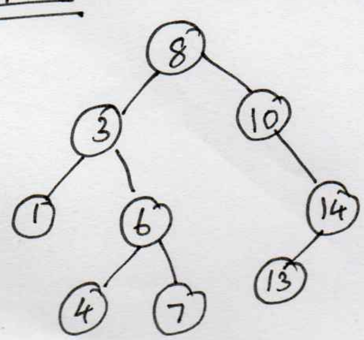
case 3: Deleting a node having two children

↳ Replace parent with child

① ↳ child can be minimum element of Right sub tree

② ↳ child can be manimum element of Left subtree

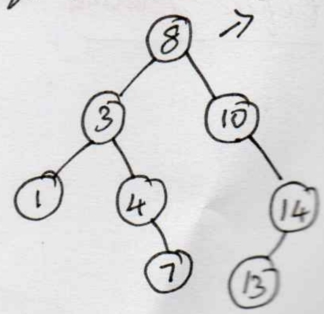
Example 1



Delete 6

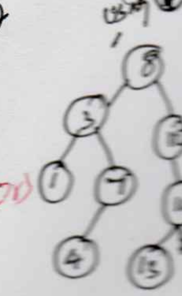
⟹

using ①

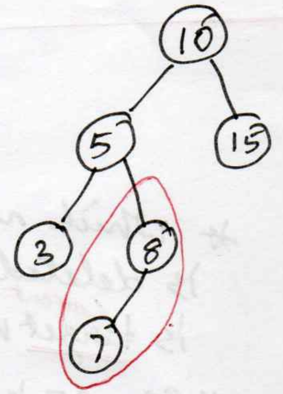


(or)

using ②

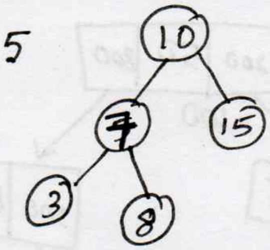


Example 2:



Delete 5 ⟹

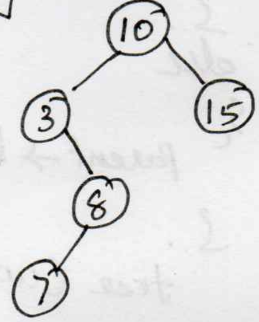
follow RST



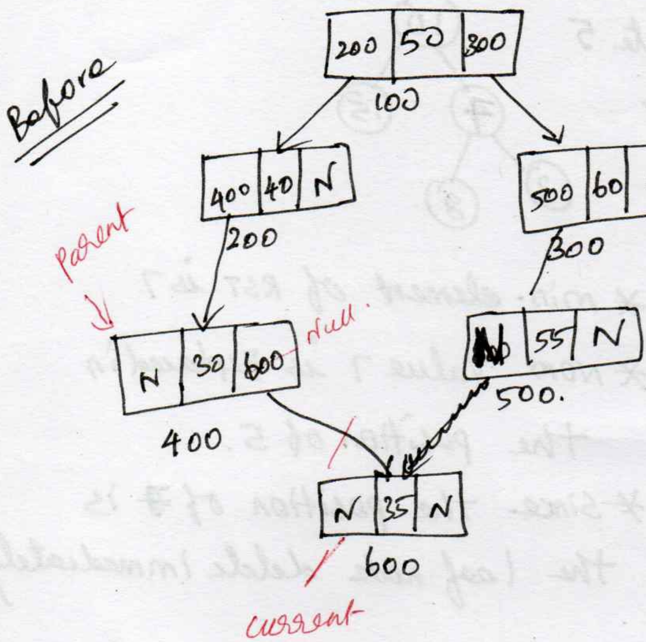
* min. element of RST is 7
* Now value 7 is Replaced in the position of 5.
* Since the position of 7 is the leaf node delete immediately

delete 5 ⇓

follow LST

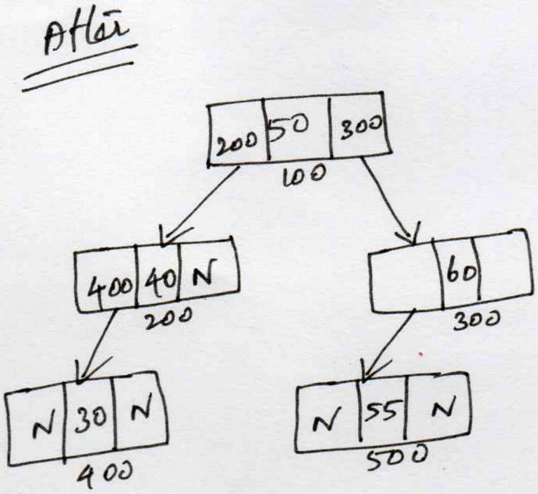


Pgm for Deleting a node with no child (leaf):



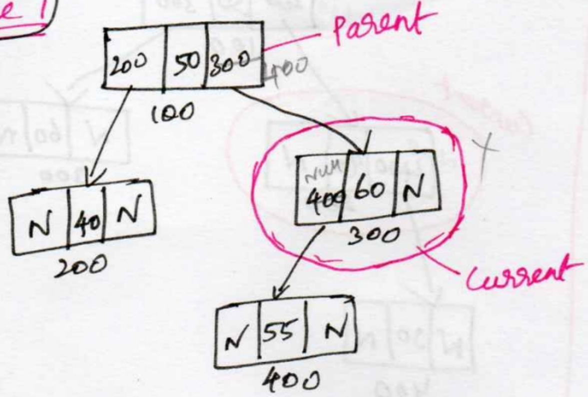
* Which node is deleted that is ~~target~~ current node
* Eg: 35 is deleted

```
      600            600
if (curr == parent->right)
{
   parent->right = Null;
}
else
{
   parent->left = Null;
}.
free (curr);
```



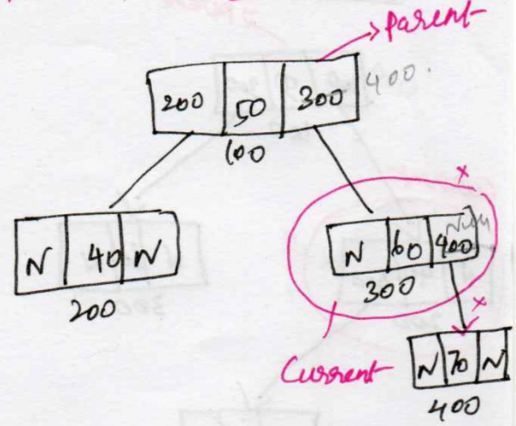
Deleting a Node with One child → 4 case

Case 1



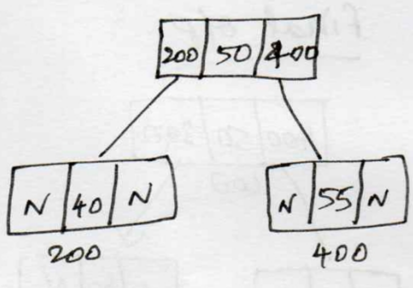

```
if (curr -> left != Null)
{
  if (curr == parent -> right)
  {
     parent -> right = curr -> left
  }}
  curr -> left == Null
  free (curr)
```

Case 2

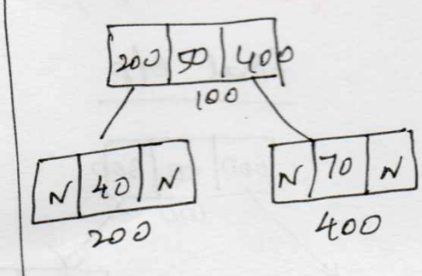



```
if (curr -> right != Null)
{
  if (curr == Parent -> right)
  {
    Parent -> right = current -> right;
  }
}
  curr -> right == Null;
  free (curr);
```

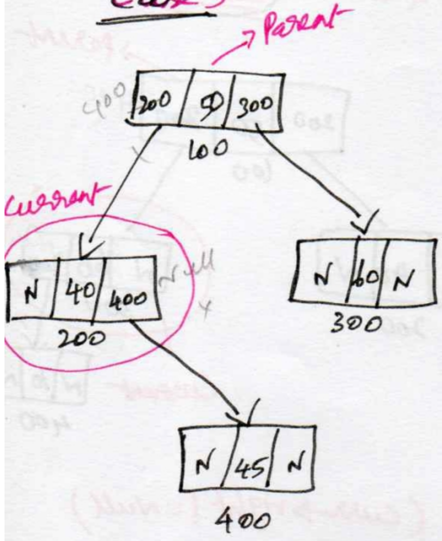
Final O/P



Final O/P

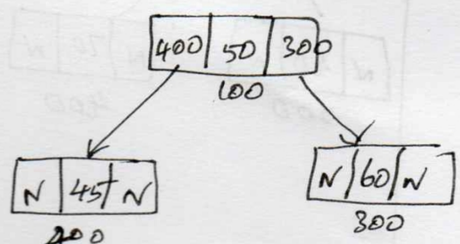


Case 3

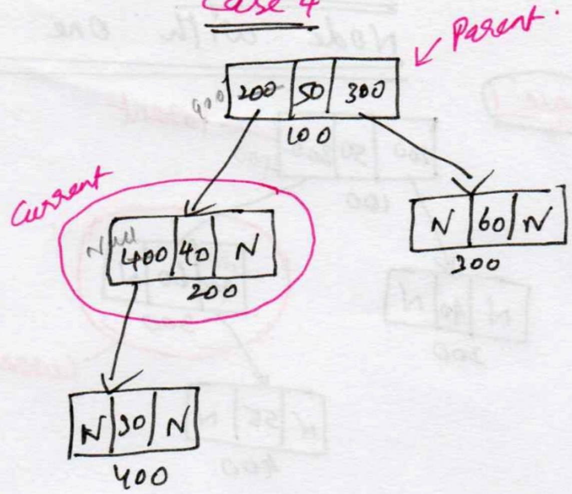



```
if (curr -> right != Null)
{
    if (curr == parent -> left)
    {
        parent -> left = curr -> right;
    }
}
curr -> right = Null ;
free (curr) ;
```

Final o/p

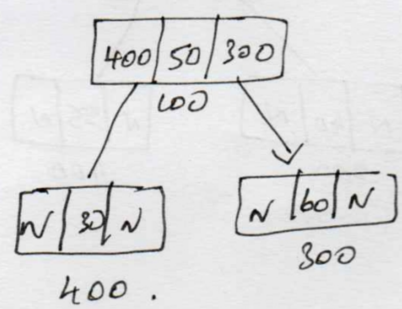


Case 4

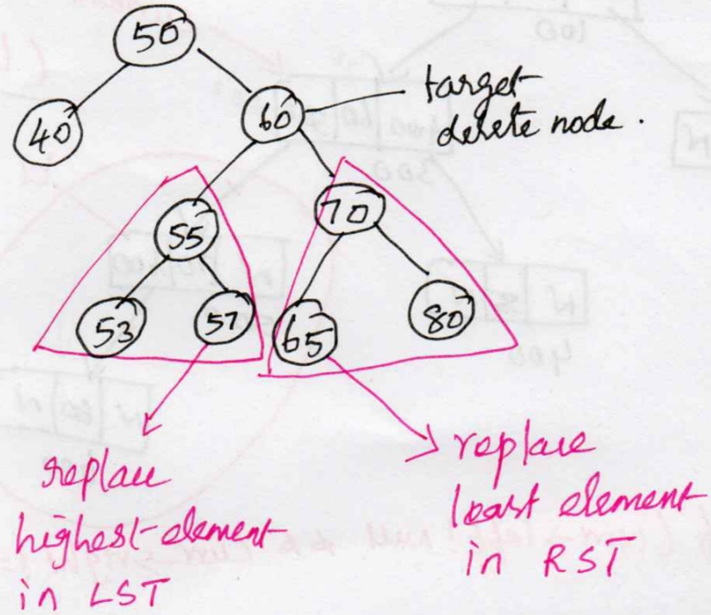



```
if (curr -> left != Null)
{
    if (curr == parent -> left)
    {
        Parent -> left = curr -> left;
    }
}
curr -> left = Null ;
free (curr);
```

Final o/p

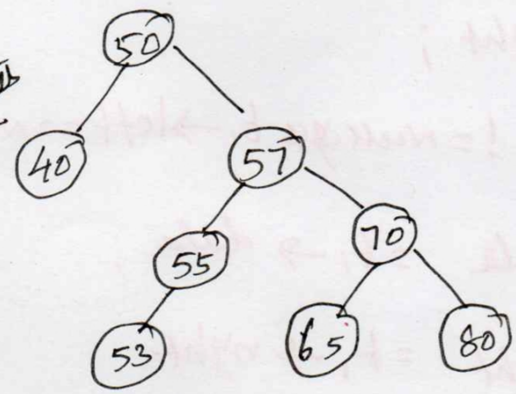


Deletion of node having two children

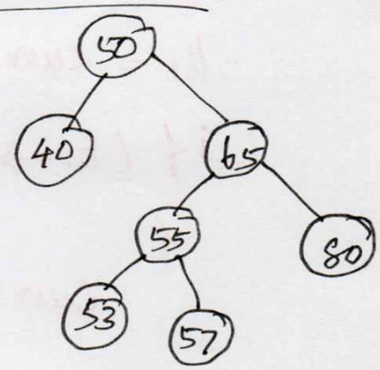


After deletion

First possibilities

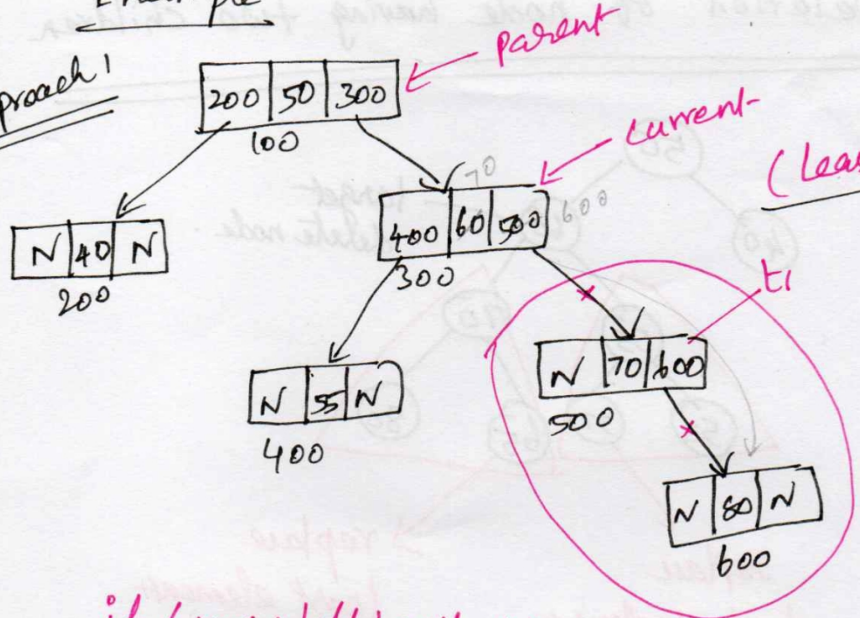


second possibilities



Example

Approach 1

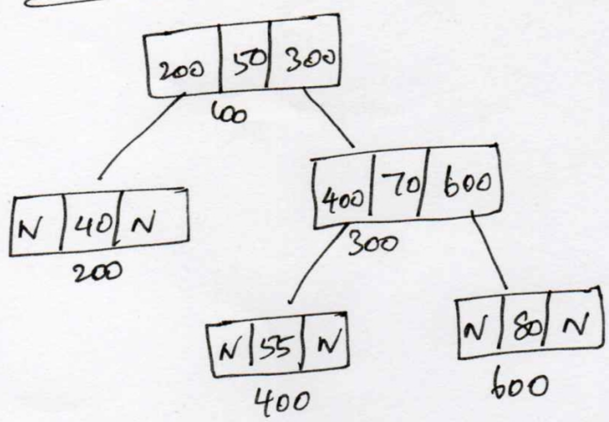


(Least element in RST).

Delete : 60.

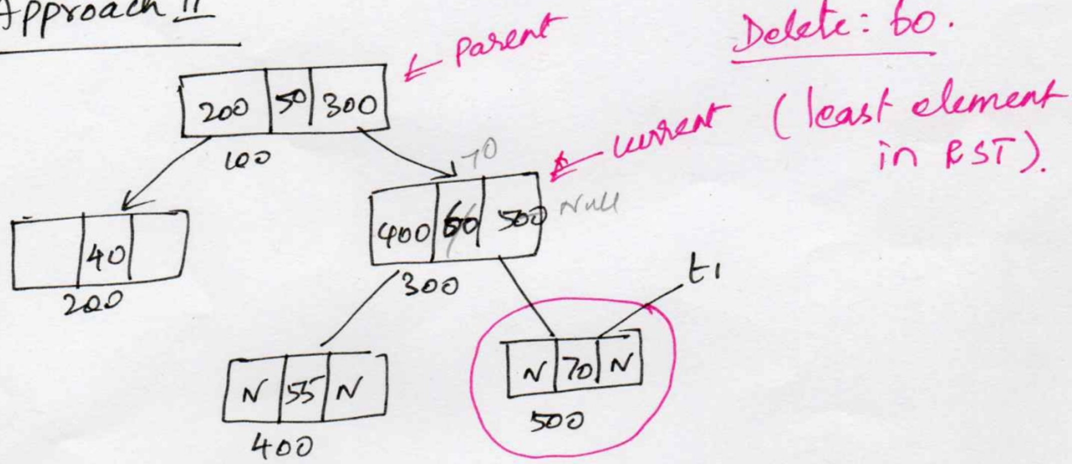
```
if (curr->left != null && curr->right != Null)
{
    struct node *t1, ;
    t1 = curr->right ;
    if (t1->right != null && t1->left == null)
    {
        curr->data = t1->data ;
        curr->right = t1->right
        t1->right = Null;
        free (t1) ;
    }
};
```

Final O/P



Approach II

Delete: 60.




```
if (curr->left != Null && curr->right != Null)
{
  struct node * t1 ;
  if (t1->left == Null && t1->right == Null)
   {
      curr->data = t1->data ;
      curr->right = Null ;
      free (t1) ;
   }
}
```

Applications of binary search trees.

- Used to efficiently store data in sorted form in order to access and search stored elements quickly.
- They can be used to represent arithmetic expressions
- BST used in Unix kernels for managing a set of virtual memory areas (VMAs).