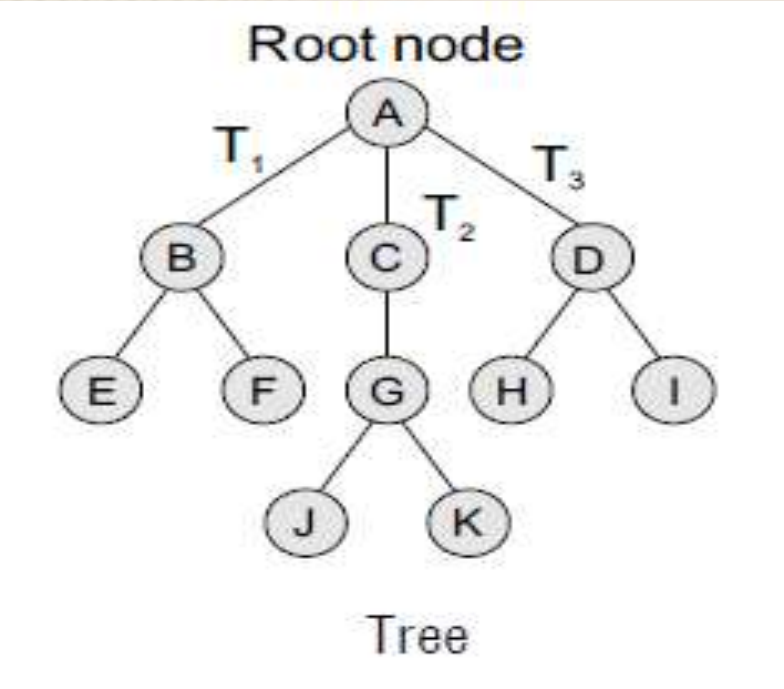
UNIT III - NON LINEAR DATA STRUCTURES – TREES

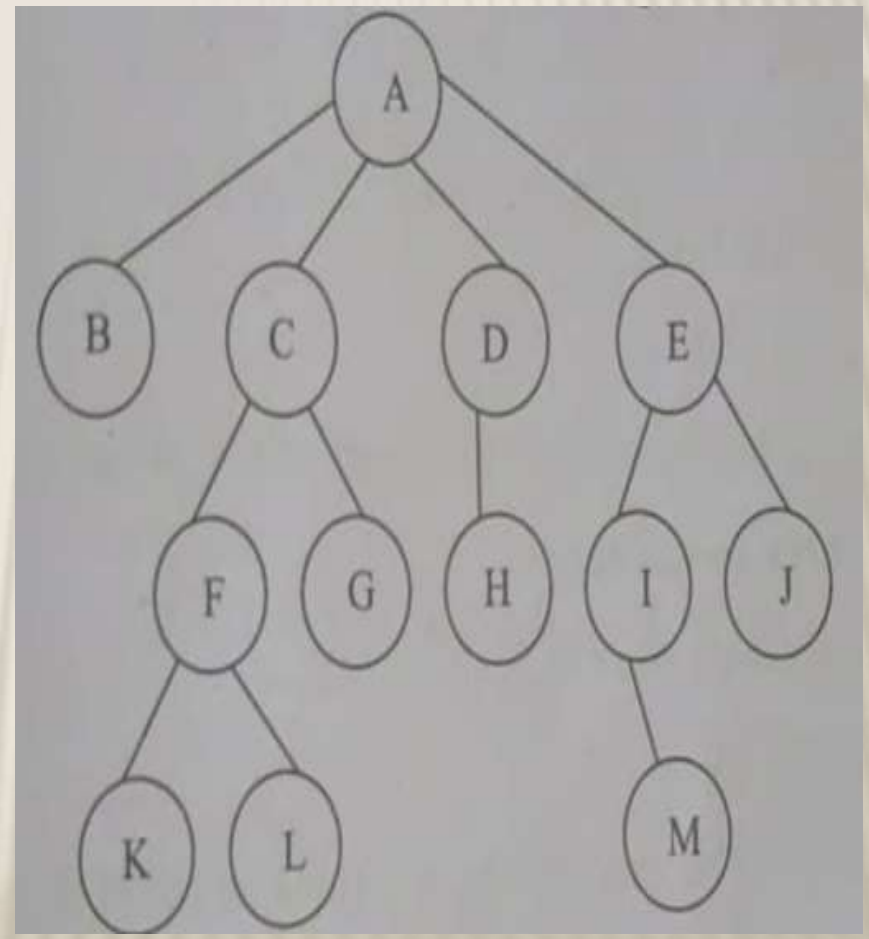
Tree ADT – Tree traversals - Binary Tree ADT – Expression trees – Applications Of Trees – Binary Search Tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap.

TREES

- A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and
- All the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.
- A **tree** structure means that the data are organized so that items of information are related by branches.
- Trees organize information hierarchically.
- A tree is a collection of elements (nodes)

TREES EXAMPLE





TREE TERMINOLOGIES

- Root – A node which does not have a parent. Root is A.
- Node – Item of Information.
- Leaf – A node which does not have a child is called Leaf or Terminal node. Here B,K,L,G,H,M,J are leafs.
- Siblings - Children of the same parents are said to be siblings.
- Path –
- Length –
- Degree -

TREE TERMINOLOGIES

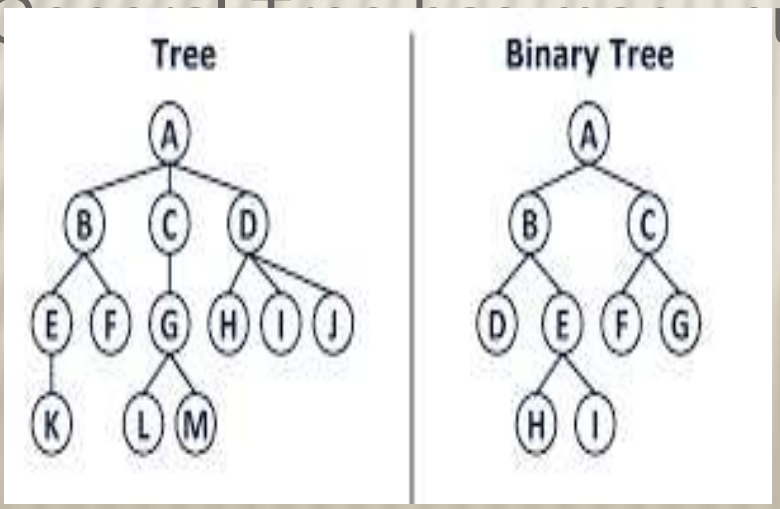
- Level
- Depth
- Height
- Ancestor node
- Descendant node

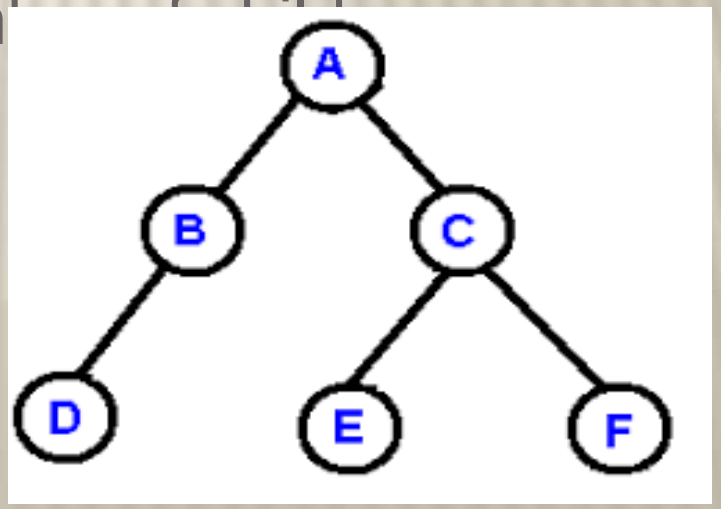
TREE TERMINOLOGIES

- Level
- Depth
- Height
- Ancestor node
- Descendant node

BINARY TREE

- Binary Tree is a tree in which no node can have more than two children.
- 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.
- Maximum number of nodes at level i of a binary tree is 2^{i+1}
- G[illegible]um[illegible]



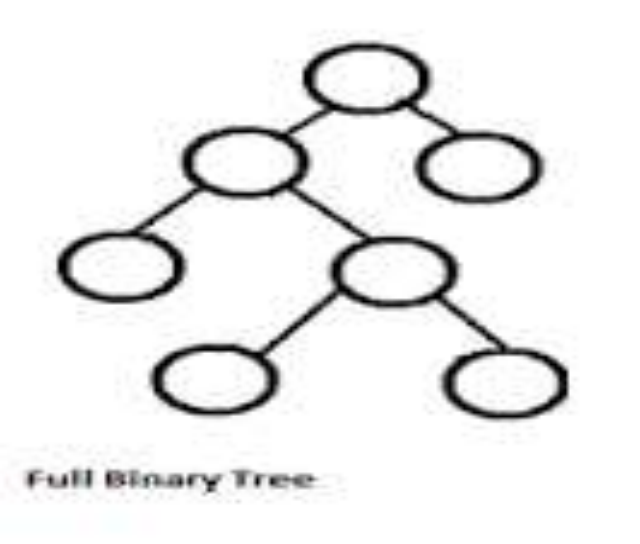


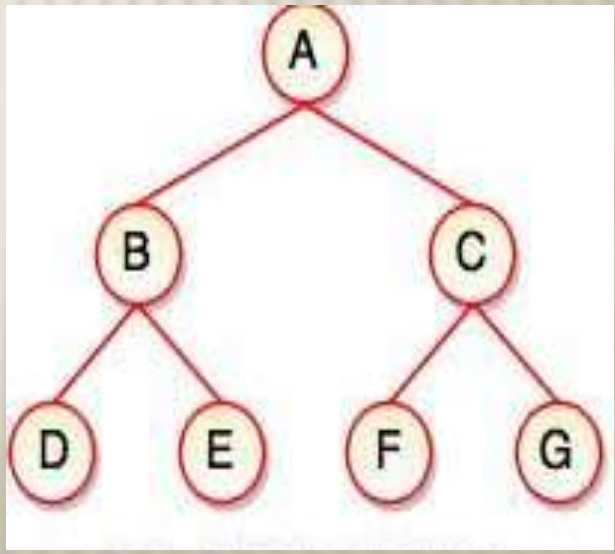
BINARY TREE NODE DECLARATION

```
Struct TreeNode
{
        int Element;
        Struct TreeNode *Left;
        Struct TreeNode *Right;
}
```

TYPES OF BINARY TREES

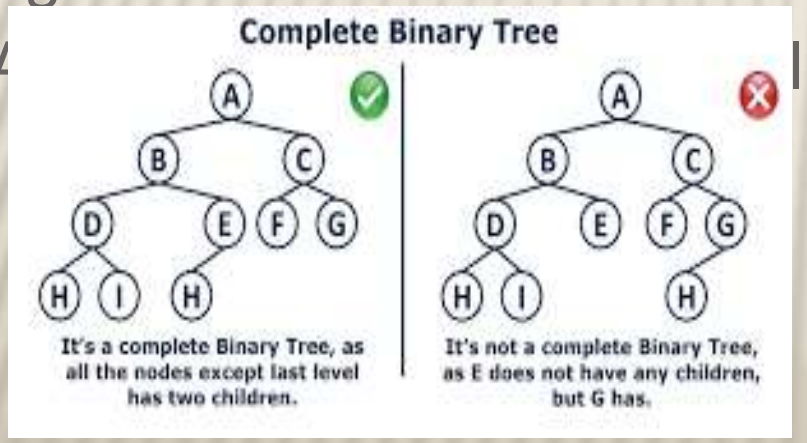
- Full Binary Tree: A full BT of height H has $2^{H+1}-1$ nodes.
- A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

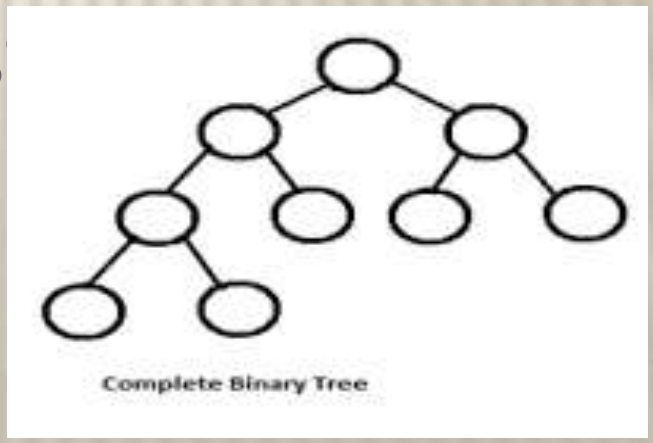




COMPLETE BINARY TREE

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A complete BT of height H has between 2^H and $2^{H+1}-1$ nodes.
- In Last level, the elements should be filled from left to right.
- A ll CB





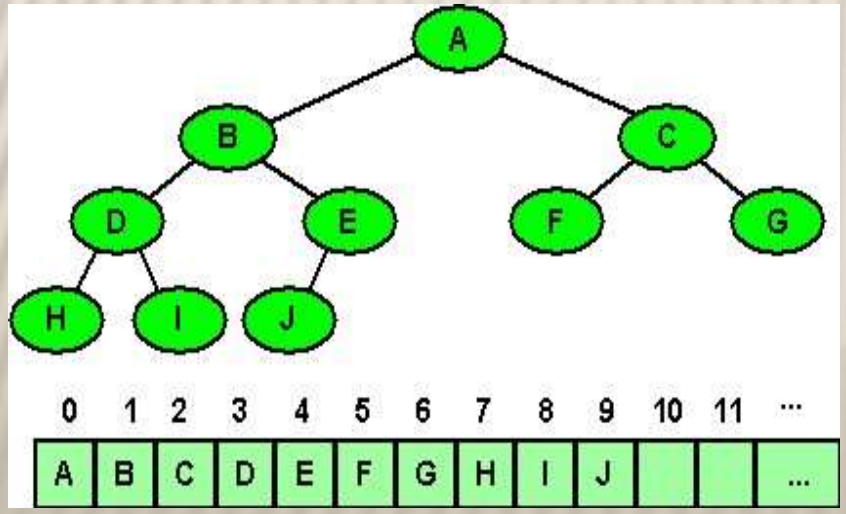
REPRESENTATION OF A BINARY TREES

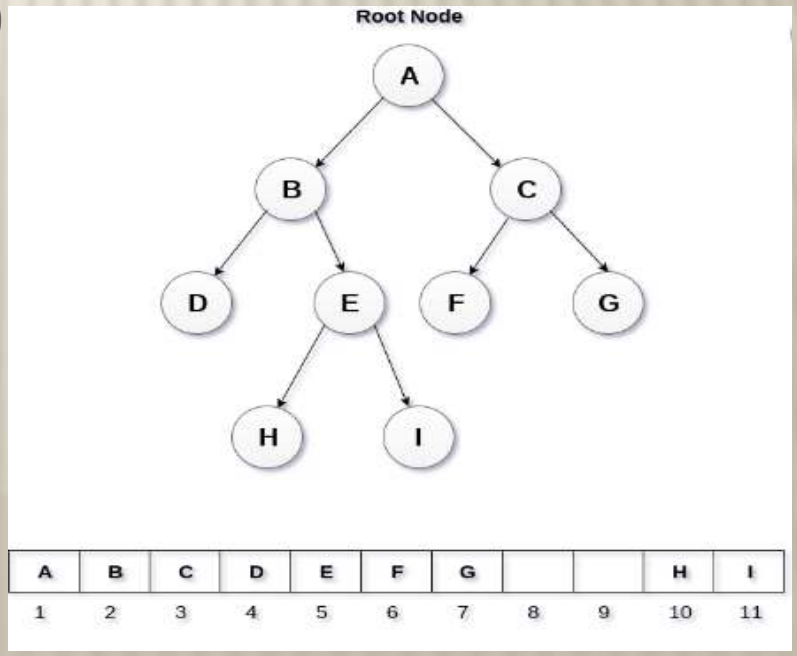
There are two ways for representing BT

- Linear Representation
- Linked Representation

LINEAR REPRESENTATION

- The elements are represented using arrays.
- For any element in position i,
- The left child is in position (2i)
- The right child is in position (2i+1)
- The parent is in position (i/2)

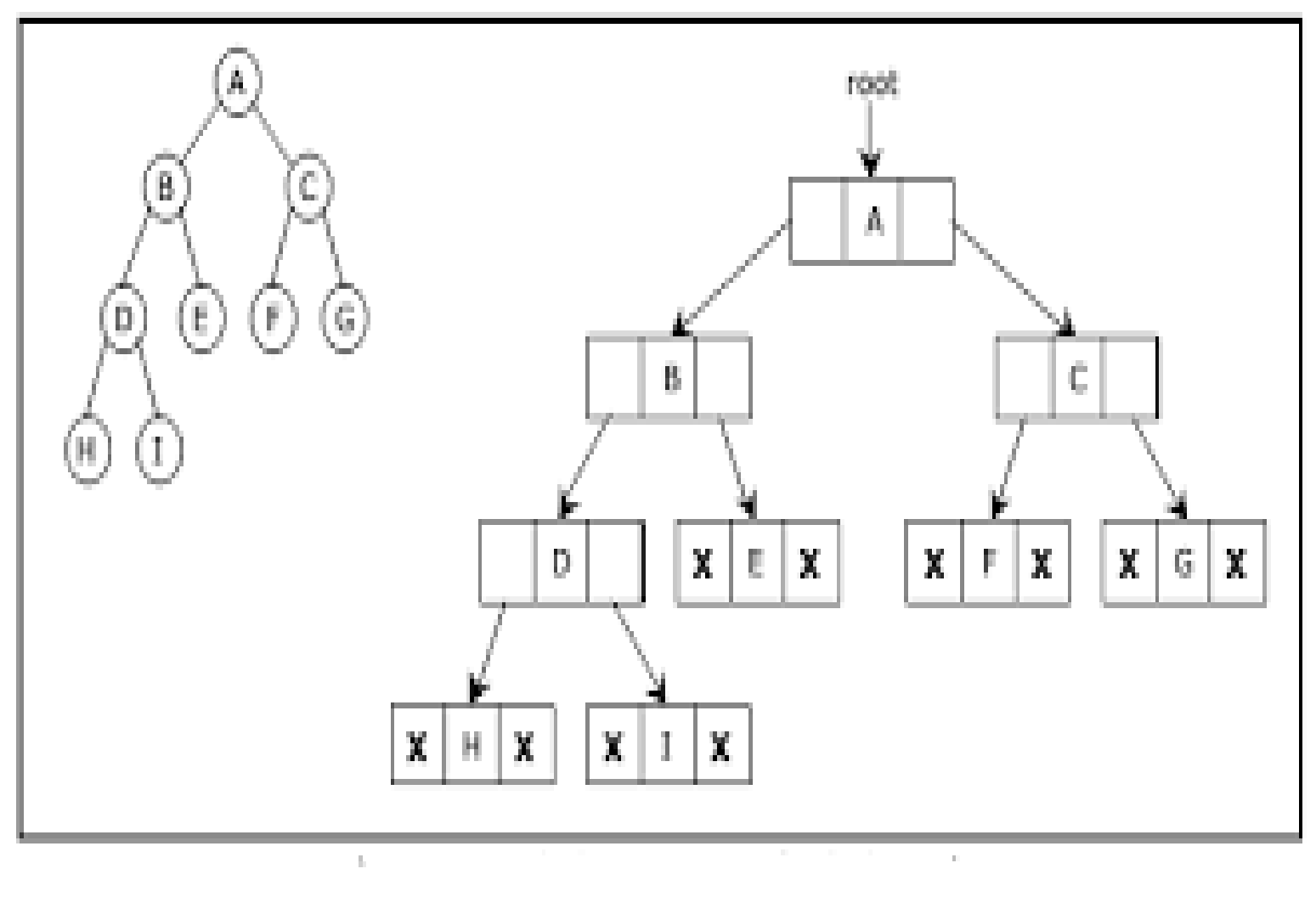




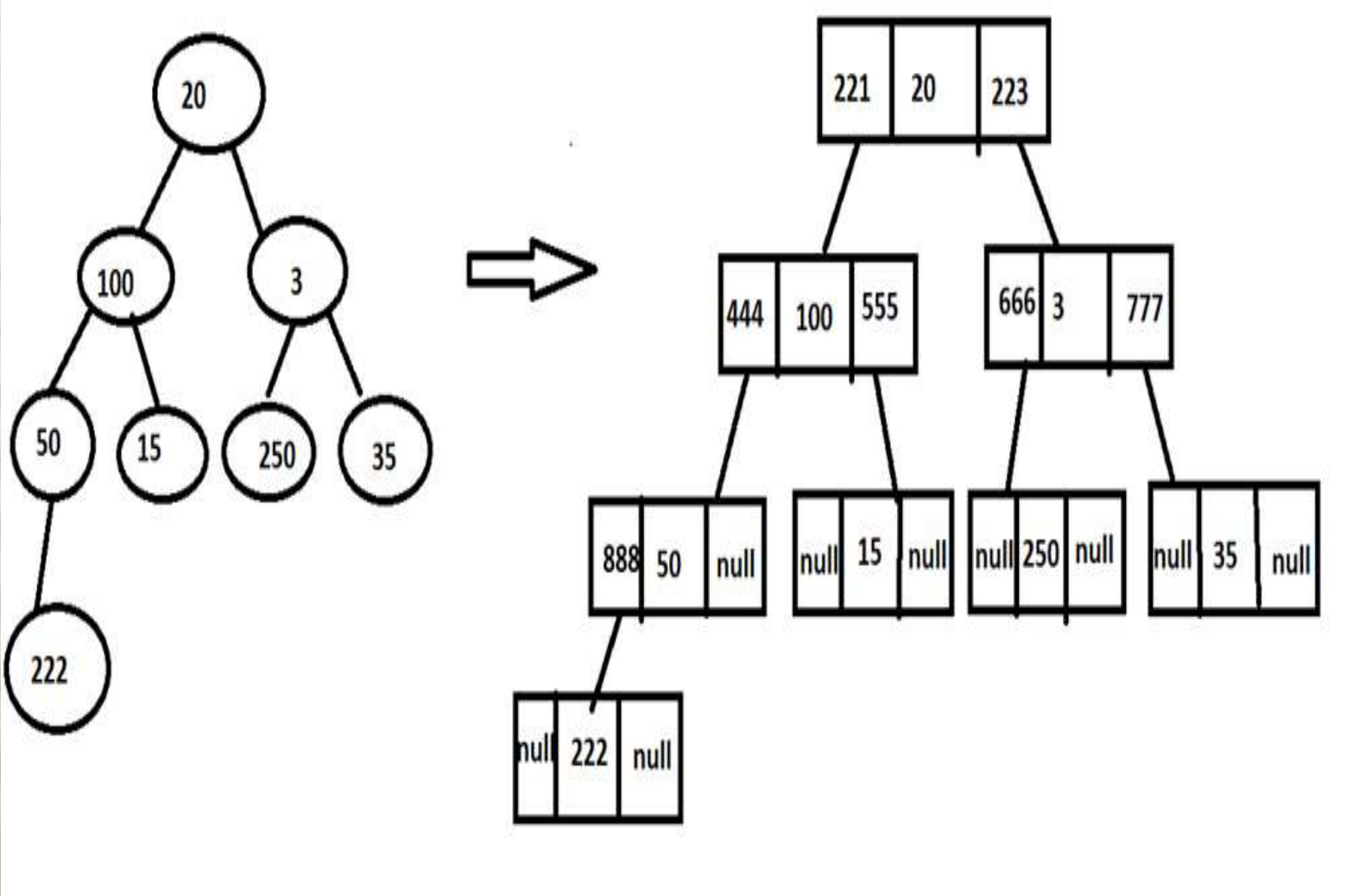
LINKED REPRESENTATION

- The elements represented using pointers.
- Each node in linked representation has three fields. Namely
  - Pointer to the Left Subtree
  - Data Field
  - Pointer to the Right Subtree
- In Leaf nodes both pointer fields are assigned as NULL.

LINKED REPRESENTATION



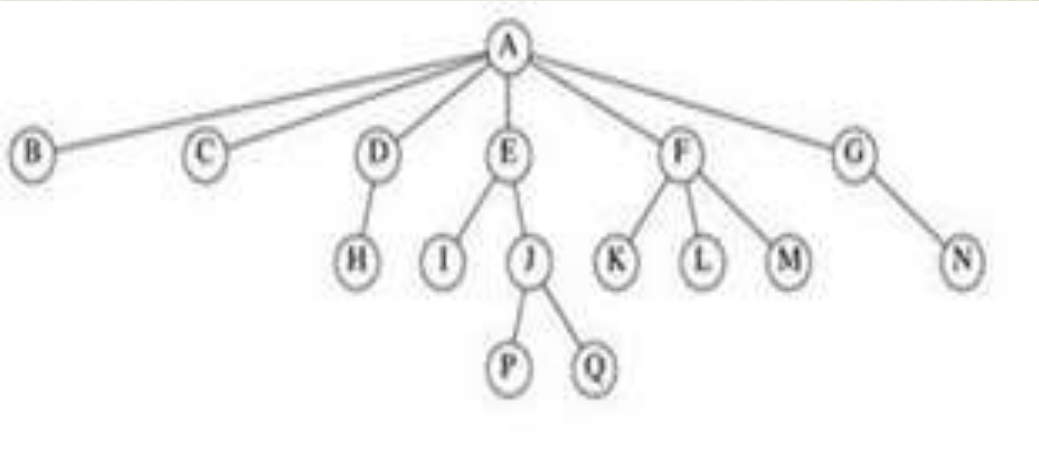
LINKED REPRESENTATION

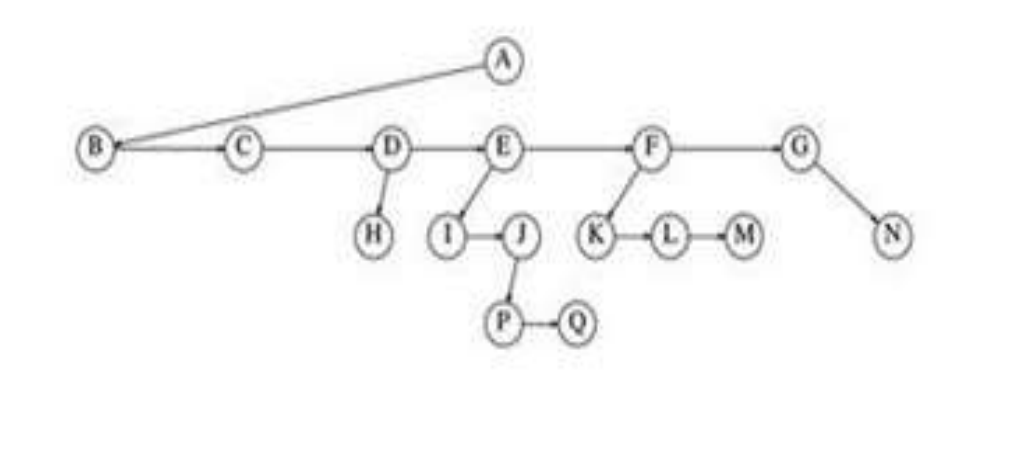


FIRST CHILD NEXT SIBLING REPRESENTATION

- In this type representation, cellspace contains three field's namely leftmost child, label and rightmost child.
- A node in identified with the index of the cell in cellspace that represents it as a child.
- Then the next pointers of cellspace point to right siblings and the information contained in the nodespace array can be held by introducing a field leftmost child in cellspace.

EXAMPLE





TREE TRAVERSAL

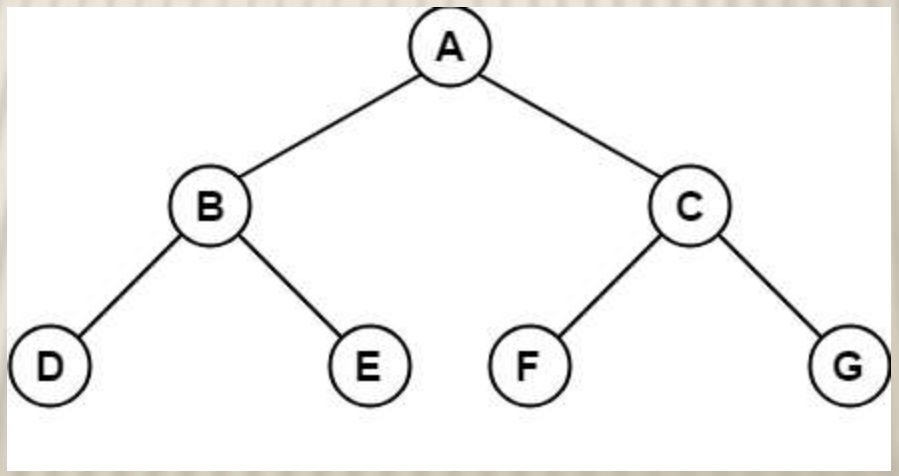
- Tree traversal is a method for visiting all the node in the tree exactly once.
- There are 3 types of tree traversal techniques
  - Inorder Traversal
  - Preorder Traversal
  - Postorder Traversal

INORDER TRAVERSAL – (LEFT, ROOT, RIGHT)

The Inorder traversal of a Binary tree is performed as

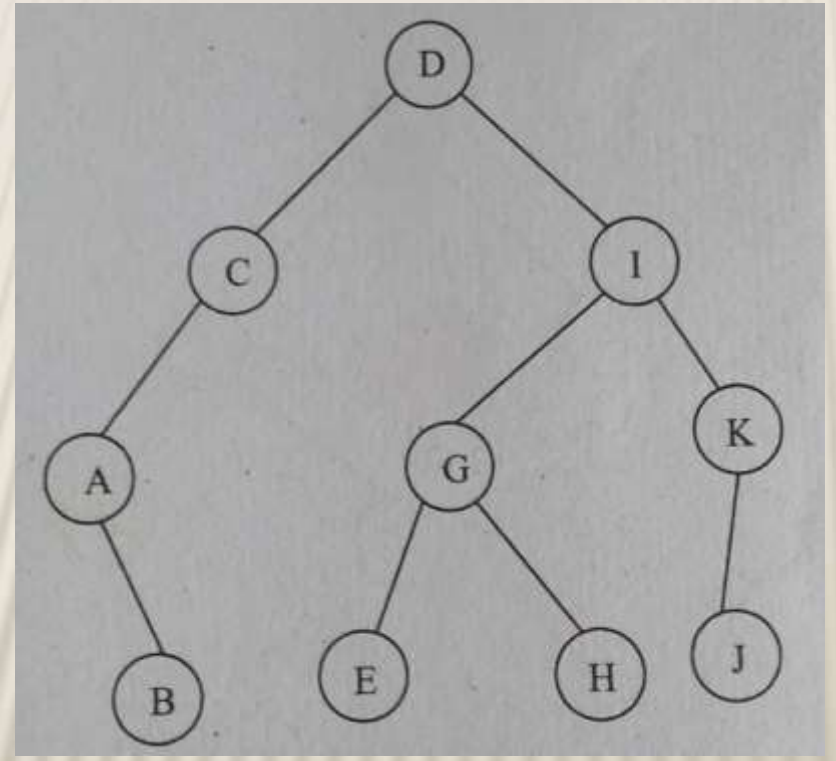
- Traverse the LEFT subtree in Inorder
- Visit the ROOT
- Traverse the RIGHT subtree in Inorder

Ex : Inorder : D B E A F C G



EXAMPLE

- Inorder : A B C D E G H I J K



RECURSIVE ROUTINE FOR INORDER

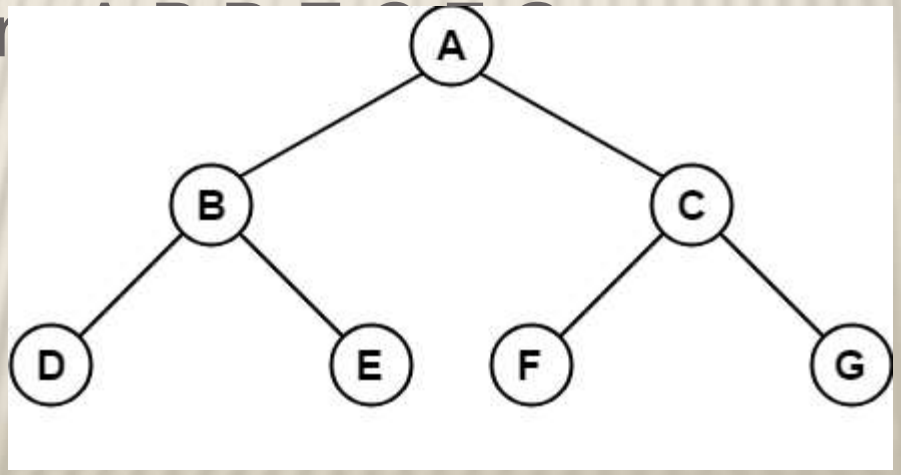
```
void inorder(Tree T)
{
        if (T!=NULL)
        {
                inorder (T□  left);
                printElement (T□  Element);
                inorder (T□  right);
        }
}
```

PREORDER TRAVERSAL - (ROOT, LEFT ,RIGHT)

The Preorder traversal of a Binary tree is performed as

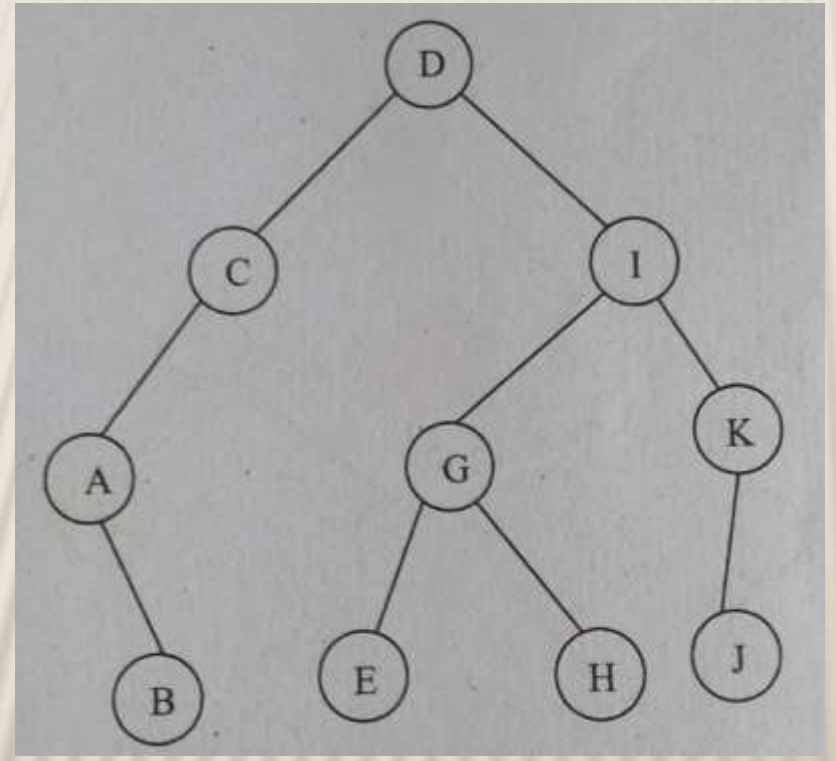
- Visit the ROOT
- Traverse the LEFT subtree in Inorder
- Traverse the RIGHT subtree in Inorder

Ex: Preorder



EXAMPLE

- Preorder : D C A B I G E H K J



RECURSIVE ROUTINE FOR PREORDER

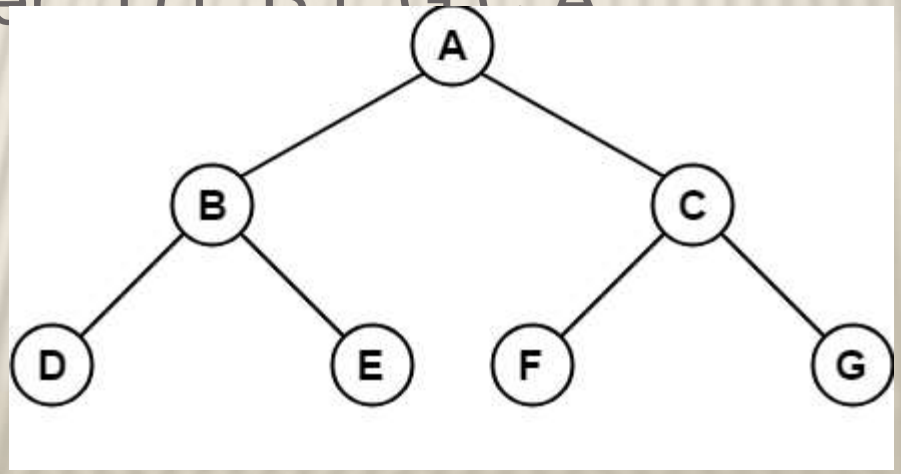
```
void preorder(Tree T)
{
        if (T!=NULL)
        {
                printElement (T□  Element);
                preorder (T□  left);
                preorder (T□  right);
        }
}
```

POSTORDER TRAVERSAL - (LEFT ,RIGHT, ROOT)

The Postorder traversal of a Binary tree is performed as

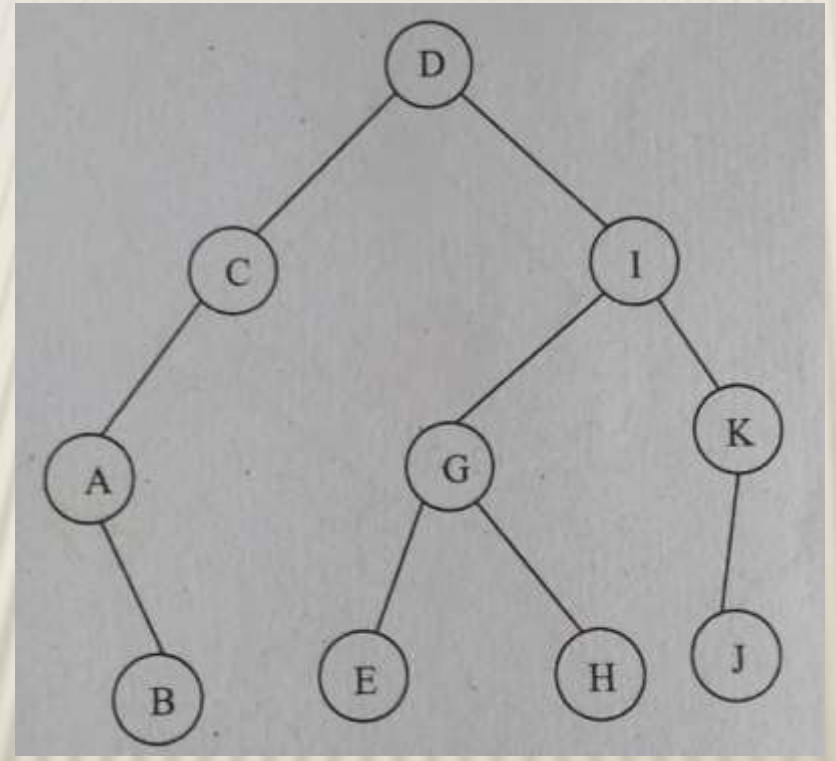
- Traverse the LEFT subtree in Inorder
- Traverse the RIGHT subtree in Inorder
- Visit the ROOT

Ex: Postorder : D E B F G C A



EXAMPLE

- Postorder : B A C E H G J K I D

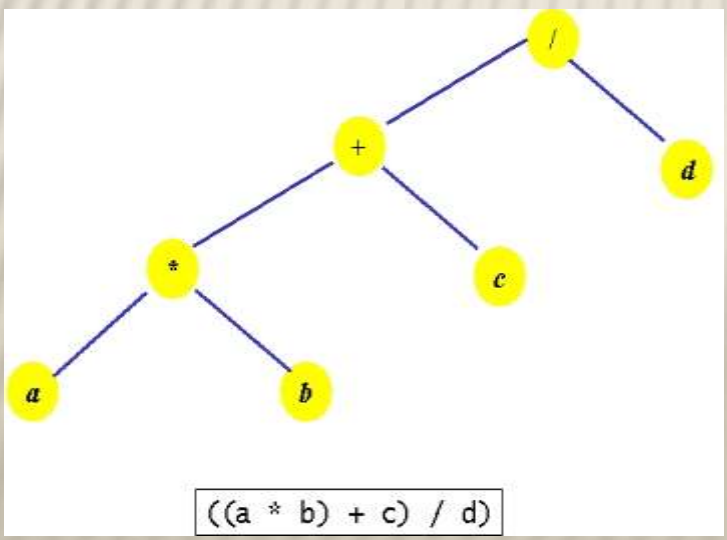


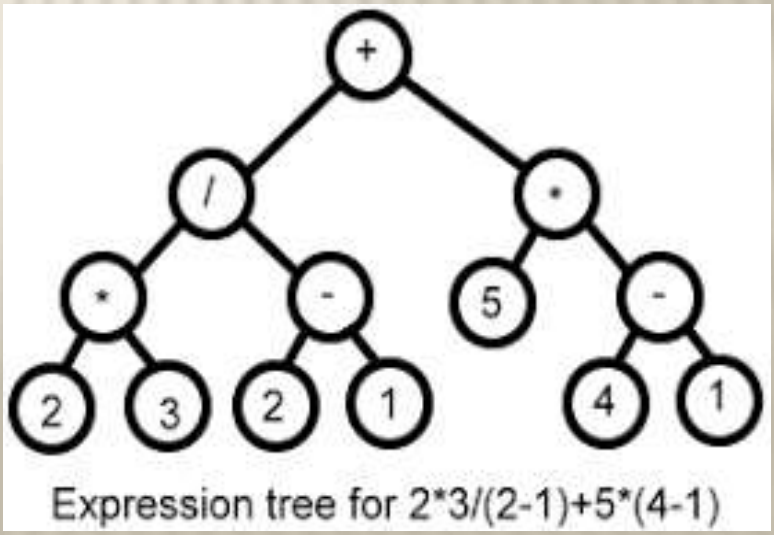
RECURSIVE ROUTINE FOR POSTORDER

```
void preorder(Tree T)
{
        if (T!=NULL)
        {
                postorder (T□   left);
                postorder (T□   right);
                printElement (T□   Element);
        }
}
```

EXPRESSION TREE

- It is a binary tree in which the leaf nodes are operands and the interior nodes are operators.
- Like binary tree, Expression tree can also traversed by Inorder, Preorder and Postorder traversal.





CONSTRUCTING AN EXPRESSION TREE

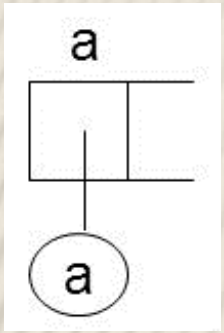
- Read one symbol at a time from the postfix expression.
- Check whether the symbol is an operand or operator

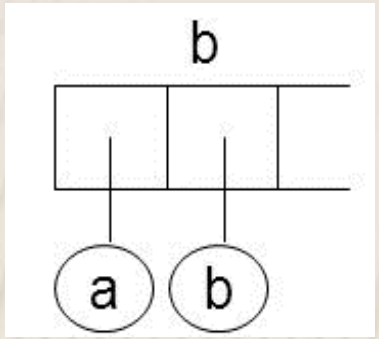
(a) If the symbol is an operand, create a one node tree and push a pointer on to the stack.

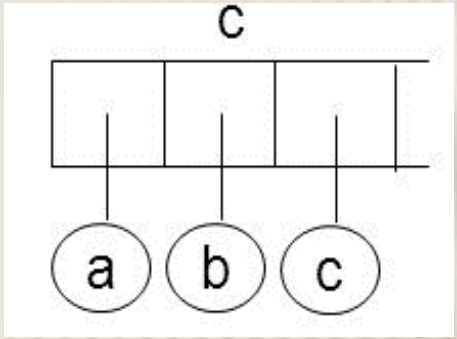
(b) If the symbol is an operator pop two pointers from the stack namely T1 and T2 and form a new tree with root as the operator and T2 as a left child and T1 as a right child. A pointer to this new tree is then pushed onto the stack.

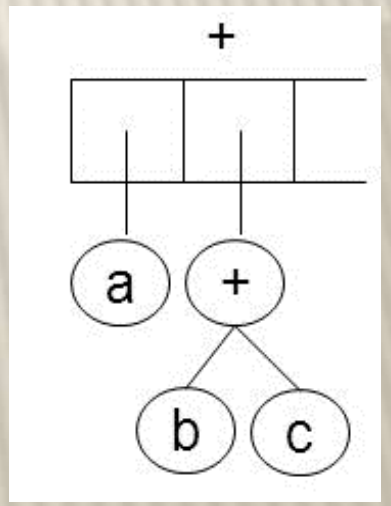
EXAMPLE

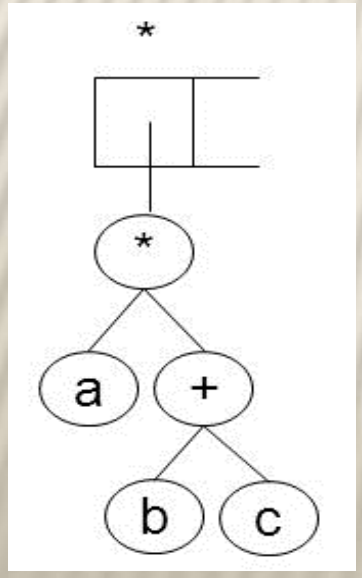
Expression : Infix : a*(b+c)/d □ Postfix: abc+*d/

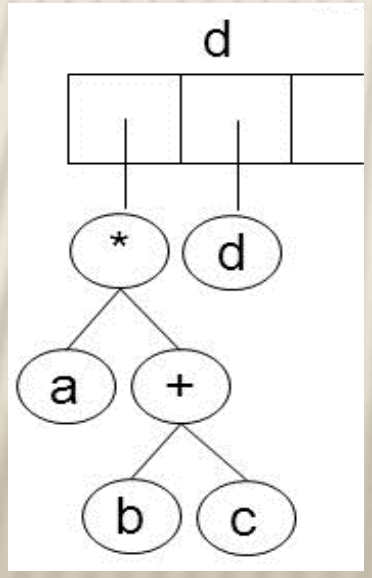


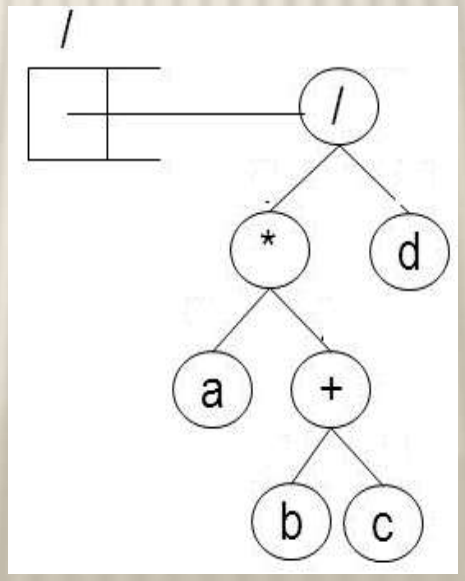










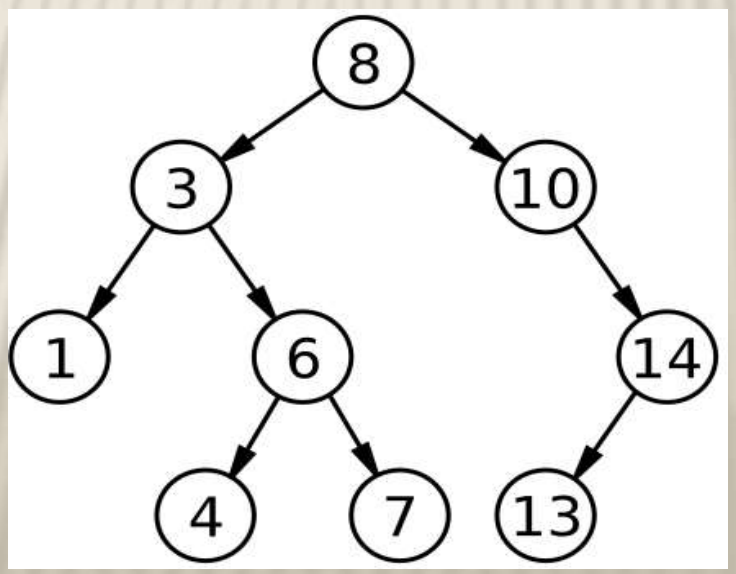


APPLICATIONS OF TREE

- Binary Search Tree.
- Expression Tree.
- Threaded Binary Tree.
- B and B+ Tree.

BINARY SEARCH TREE

- BST is a binary tree in which each node is systematically arranged that is left child has less value than its parent node and right child value has greater value than its parent node.
- The searching of any node in such a tree becomes efficient in BST.



BINARY SEARCH TREE

- Every BST is a Binary Tree.
- All Binary Trees need not be a BST.
- left_subtree (keys) < node (key) ≤ right_subtree (keys)

Node Declartion

```
Struct node
{
        int data;
        struct node *leftChild;
        struct node *rightChild;
};
```

ADVANTAGE OF BST

- The major **advantage of binary search trees** over other data structures is that the related sorting algorithms and **search** algorithms such as in-order traversal can be very efficient.

OPERATIONS ON BST

- Insertion
- Find
- Find Min
- Find Max
- Deletion

    Case 1 : Node with no Children

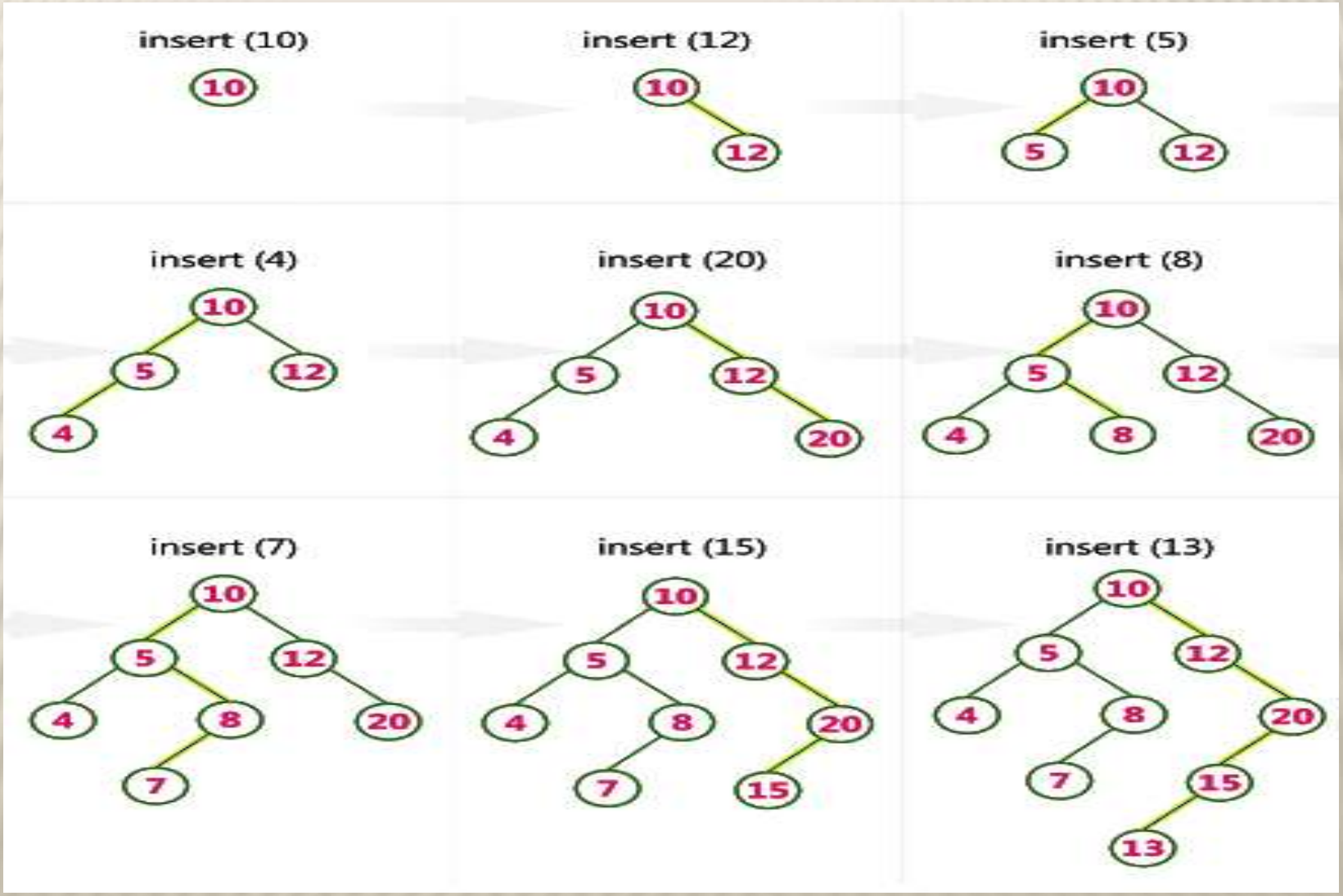
    Case 2: Node with One Child

    Case 3: Node with Two Child

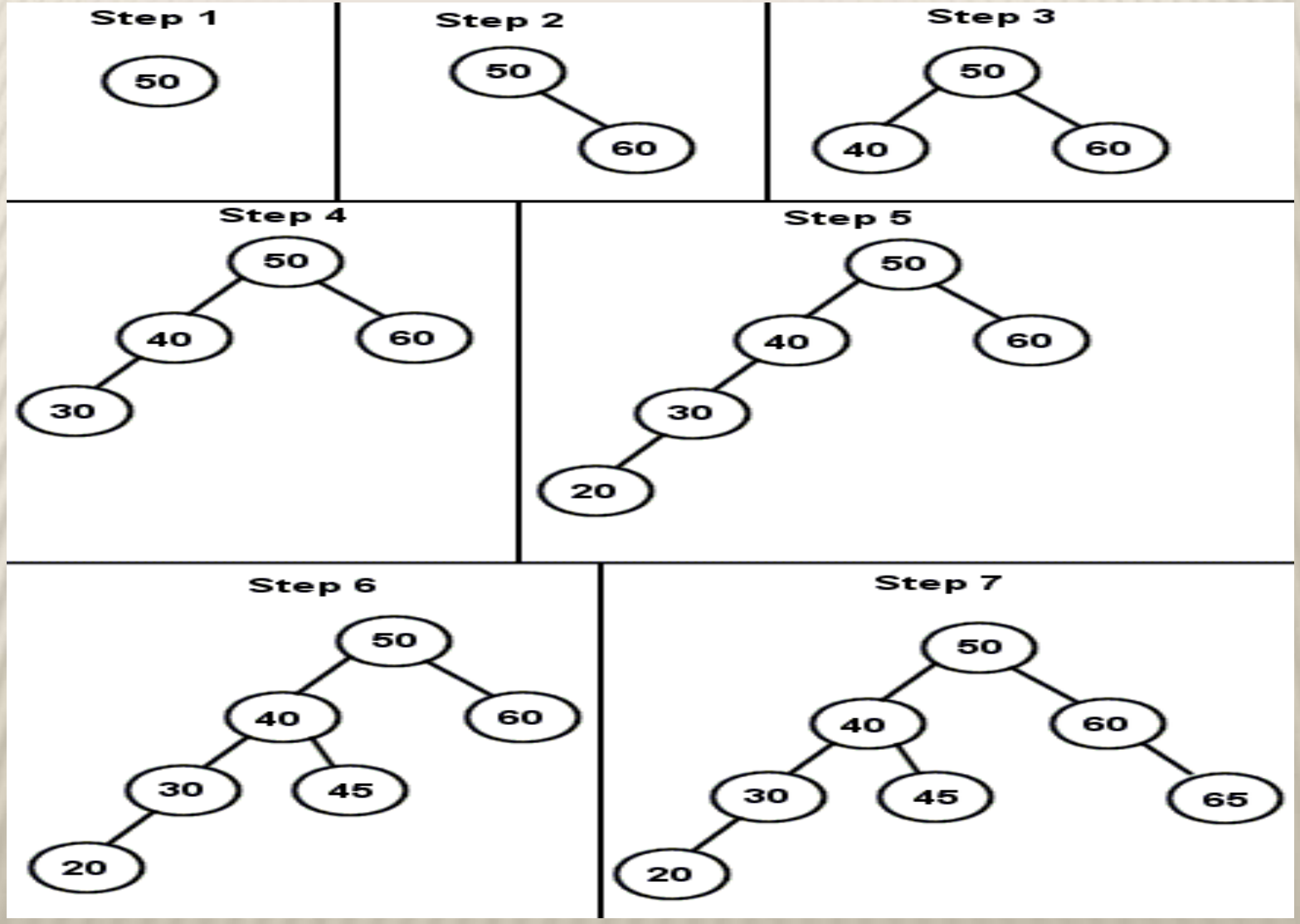
INSERTION ALGORITHM

```
SearchTree Insert(int X,SearchTree T)
{
        if(T==NULL)
        {
                T=malloc(sizeof(Struct TreeNode));
                if(T!=NULL)
                {
                        T□  Element=x;
                        T□  Left=NULL;
                        T□  Right=NULL;
                }
        }
        else
                If(X<T□  Element)
                T□  Left=Insert(X,T□  Left);
        else
                If(X>T□  Element)
                T□  Right=Insert(X,T□  Right);
        return T;
}
```

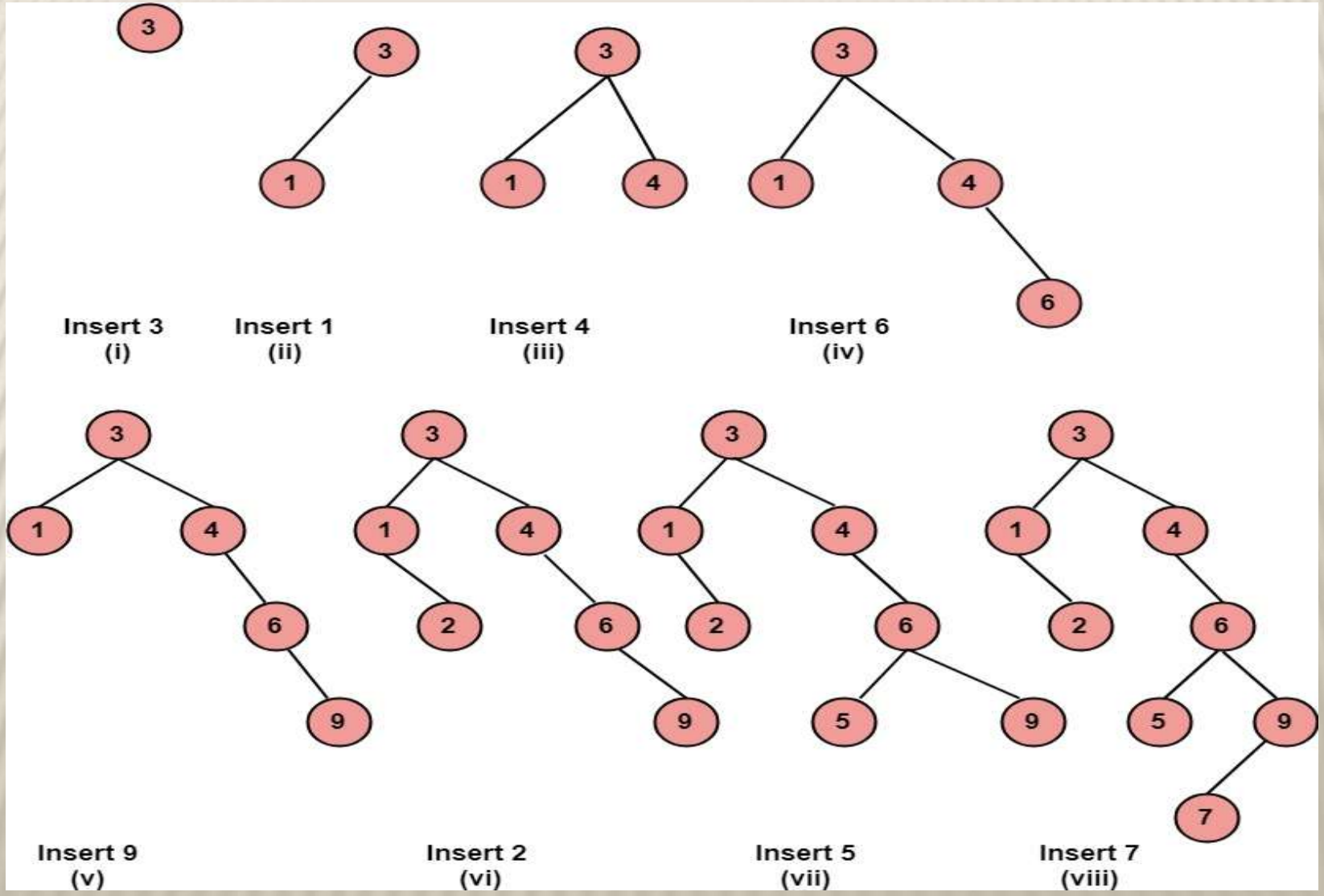
INSERTION ALGORITHM – EXAMPLE 1



INSERTION ALGORITHM – EXAMPLE 2



INSERTION ALGORITHM – EXAMPLE 3



FINDMIN

- This operation returns the position of the smallest element in the tree.
- To perform FindMin start at the root and go left as there is a left child.
- The stopping point is the smallest element.

FIND MIN

NON RECURSIVE

```
int FindMin(SearchTree T)
{
        if(T!=NULL)
        while(T□  Left!=NULL)
        T=T□  Left;
        return T;
}
```

RECURSIVE

```
int FindMin(SearchTree T)
{
        if(T==NULL)
                return NULL;
        else if(T□  Left==NULL)
                return T;
        else
        return FindMin(T□  Left);

}
```

FINDMAX

- This operation returns the position of the largest element in the tree.
- To perform FindMax start at the root and go right as there is a left child.
- The stopping point is the largest element.

FIND MAX

NON RECURSIVE

```
int FindMax(SearchTree T)
{
        if(T!=NULL)
        while(T□  Right!=NUL
L)
        T=T□  Right;
        return T;
}
```

RECURSIVE

```
int FindMax(SearchTree T)
{
        if(T==NULL)
                return NULL;
        else
if(T□  Right==NULL)
                return T;
        else
        return
FindMax(T□  Right);

}
```

DELETION

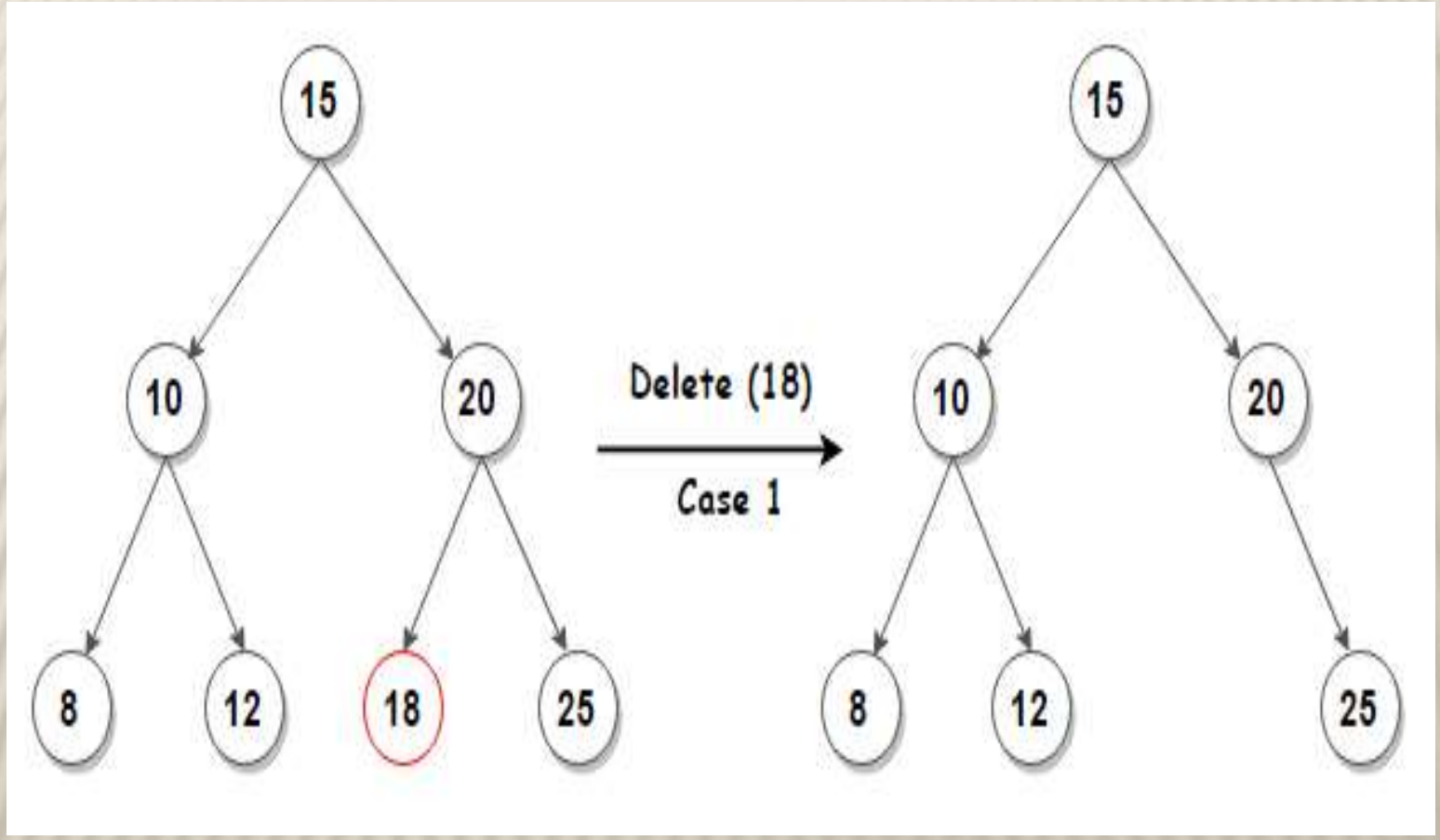
- Deletion operation is the complex in BST.

  Case 1 : Node with no Children

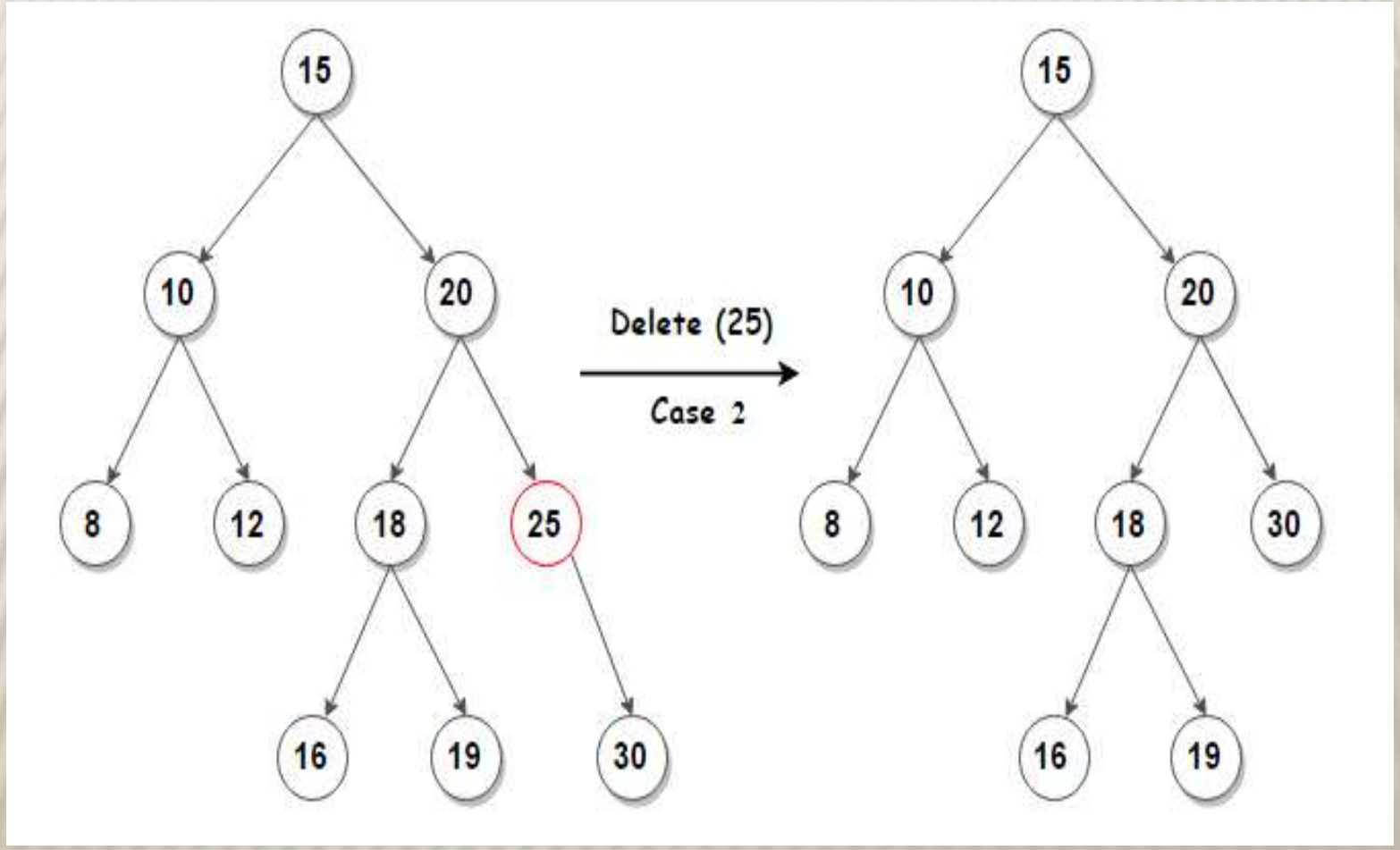
  Case 2: Node with One Child

  Case 3: Node with Two Child

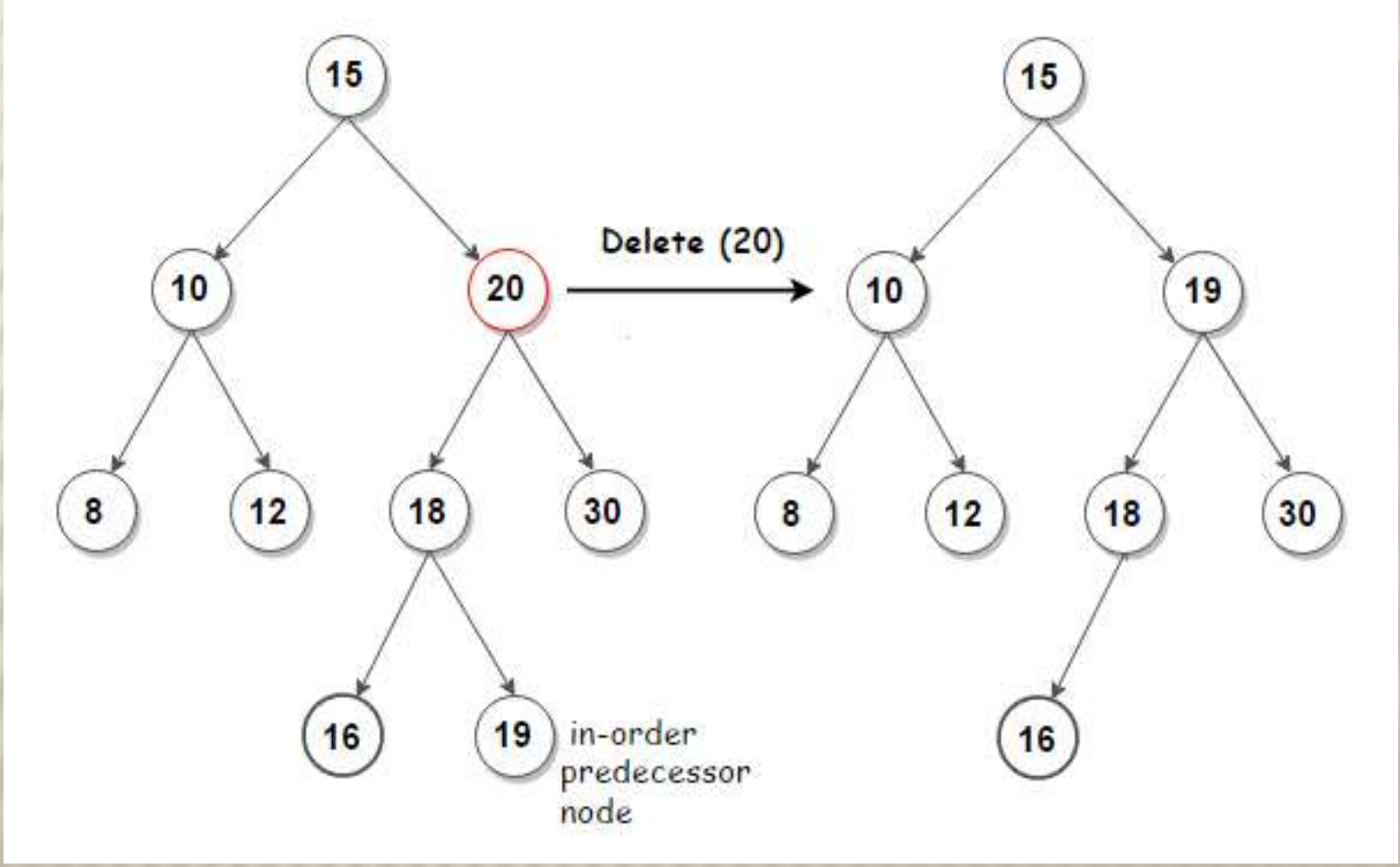
CASE 1 : NODE WITH NO CHILDREN



CASE 2: NODE WITH ONE CHILD



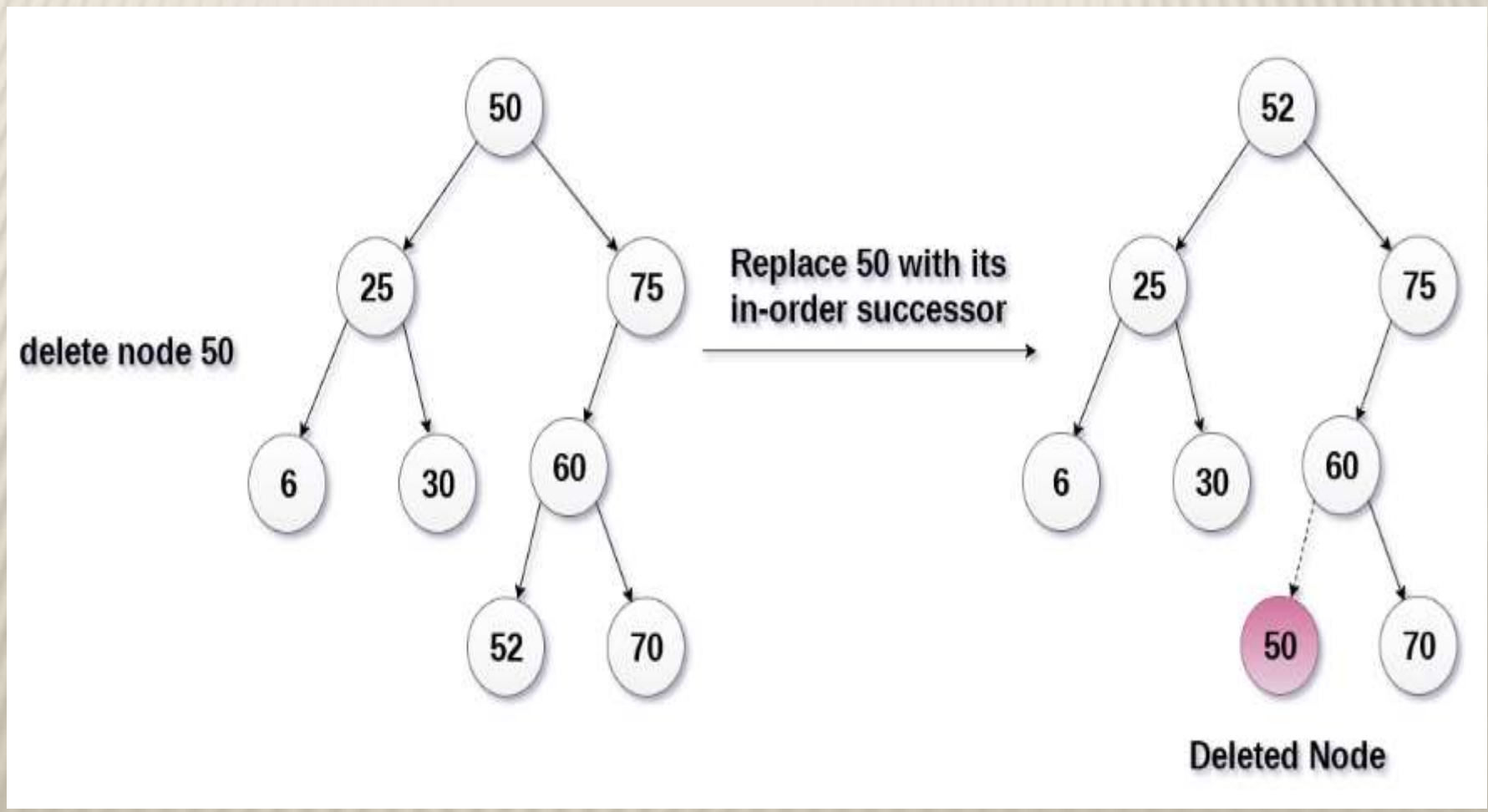
CASE 3: NODE WITH TWO CHILD



STRATEGY FOR CASE 3

- Strategy 1 : The general strategy is to replace the data if the node to be deleted with its smallest data of the right subtree and recursively delete that node.
- Strategy 2: Using Inorder
  - First find the successor (or predecessor) of the this node.
  - Delete the successor (or predecessor) from the tree.
  - Replace the node to be deleted with the successor (or predecessor)

CASE 3: NODE WITH TWO CHILD



DELETION ROUTINE

```
SearchTree Delete(int X, SearchTree T)
{
        int Tmpcell;
        if(T==NULL)
                Error("Tree is Empty");
        else
        if(X<T□  Element)
                T□  Left=Delete(X,T□  Left);
        else
        if(X>T□  Element)
                T□  Right=Delete(X,T□  Right);
```

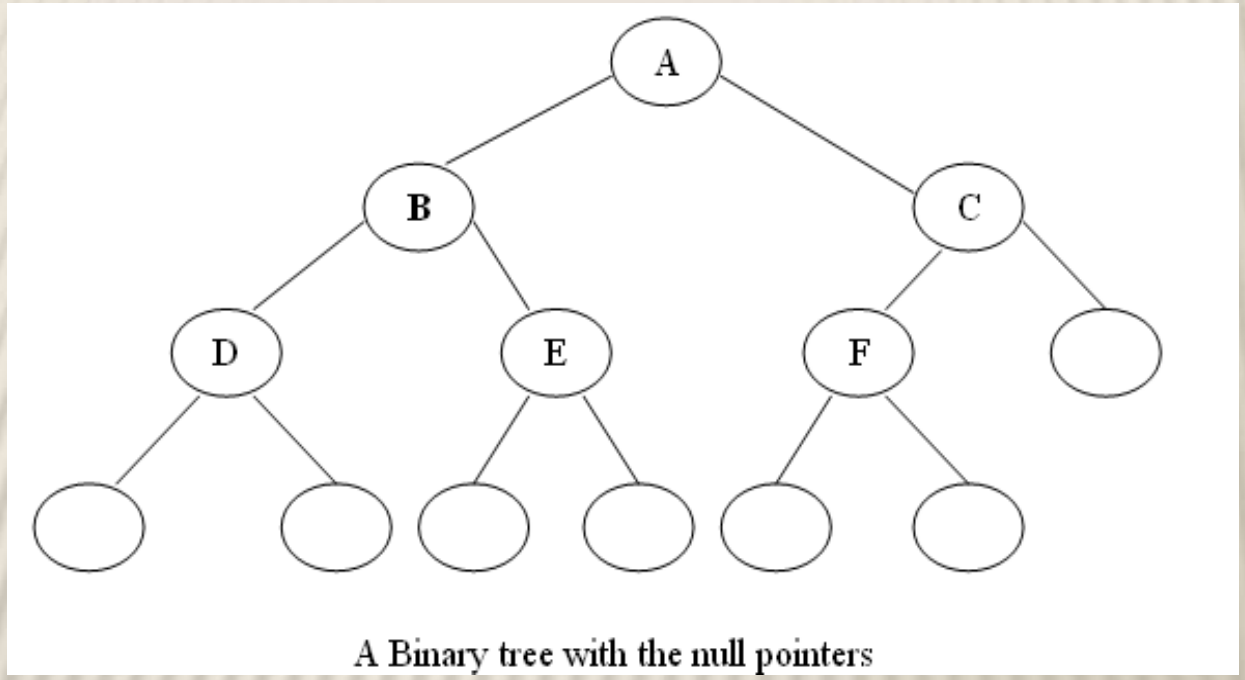
DELETION ROUTINE – CONT.,

```
else  //Two Children ( Replace with smallest data in the right subtree)
if(T□  Left && T□  Right)
{
        Tmpcell=FindMin(T□  Right);
        T□  Element=Tmpcell□  Element;
        T□  Right=Delete(T□  Element;T□  Right);
}
else  //One or Zero Children
{
        Tmpcell=T;
        if(T□  Left==NULL)
                T=T□  Right;
        if(T□  Right==NULL)
                T=T□  Left;
        Free(Tmpcell);
}
        return T;
}
```

THREADED BINARY TREE

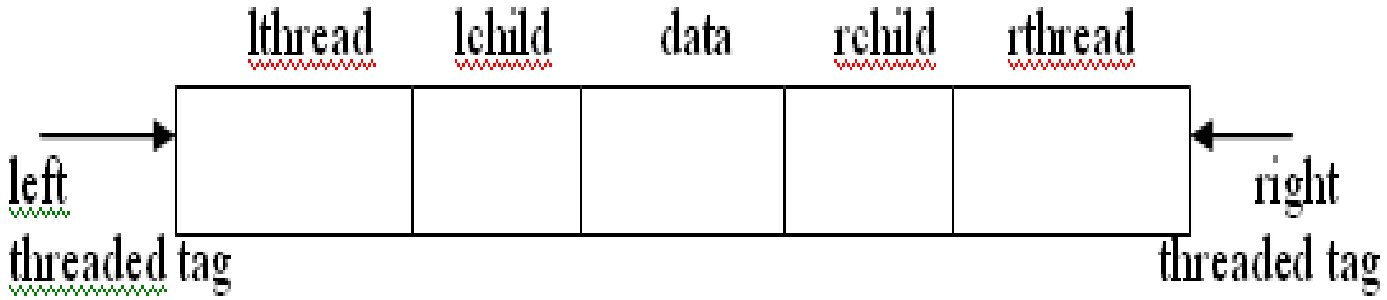
- In TBT the leaf nodes are having the NULL values in the left and right link fields.
- To avoid the NULL value the threads are used.
- The threads are nothing but the links to predecessor and successor nodes.
- Instead of left NULL pointer the link points to Inorder predecessor.
- Instead of right NULL pointer the link points to Inorder successor.

THREADED BINARY TREE



A Binary tree with the null pointers

THREADED BINARY TREE



For any node p, in a threaded binary tree.

lthred(p)=1 indicates lchild (p) is a thread pointer

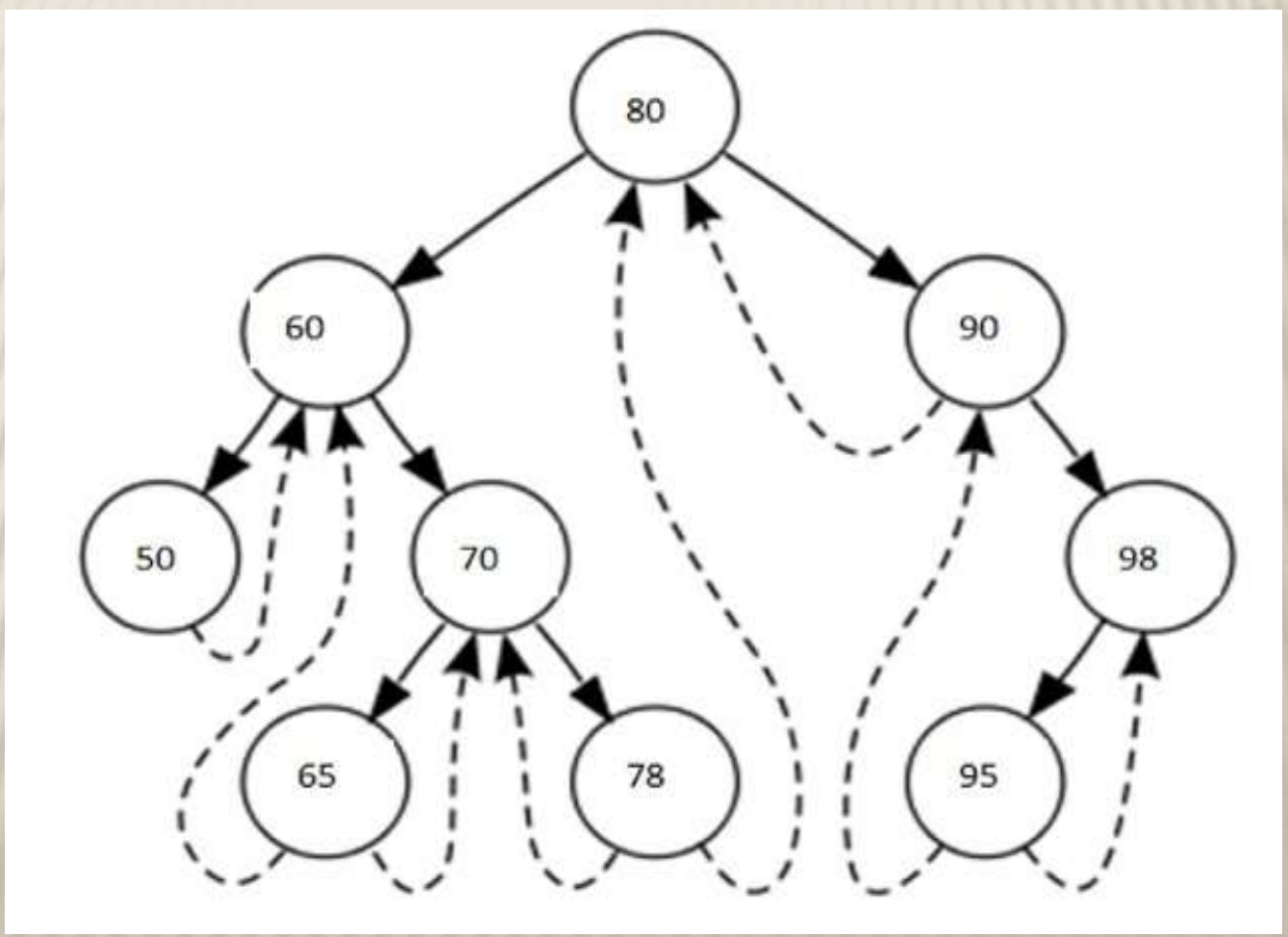
lthred(p)=0 indicates lchild (p) is a normal

rthred(p)=1 indicates rchild (p) is a thread

rthred(p)=0 indicates rchild (p) is a normal pointer

THREADED BINARY TREE

- Inorder : 50 60 65 70 78 80 90 95 98



TYPES OF TBT

There are two types

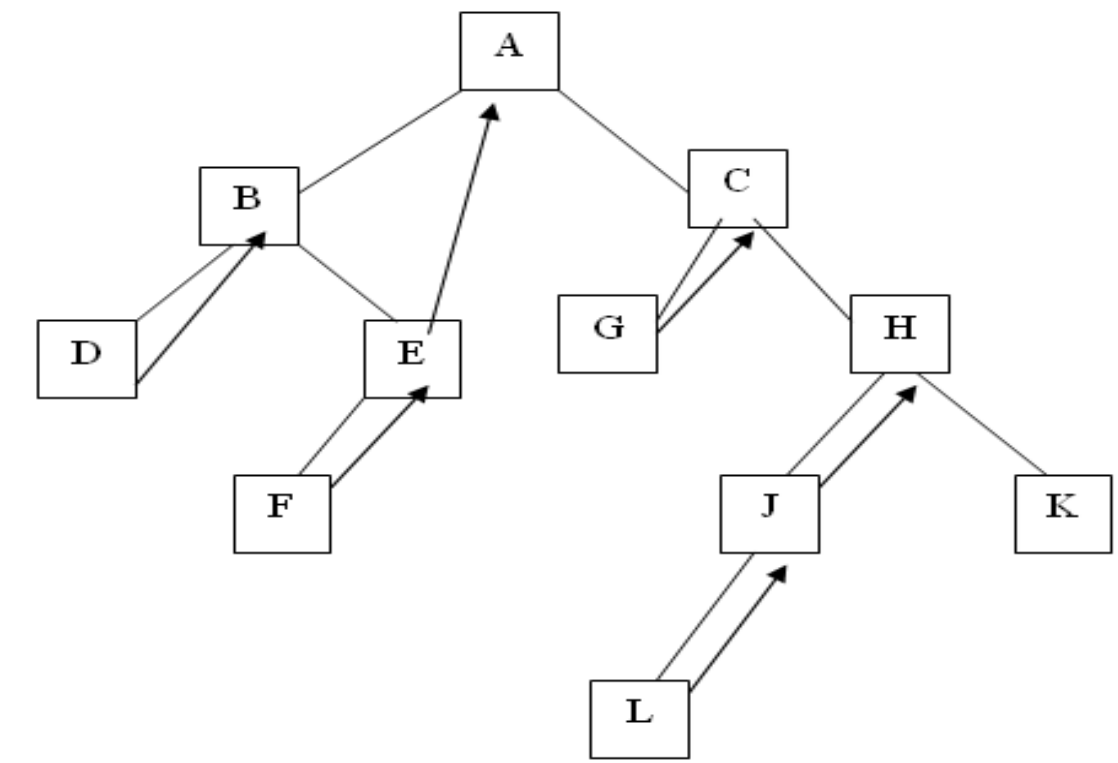
- One way Threading
- Two way Threading

ONE WAY THREADING

- Accordingly, in the one way threading of T, a thread will appear in the right field of a node and will point to the next node in the in-order traversal of T.
- See the bellow example of one-way in-order threading.

EXAMPLE

- Inorder of bellow tree is: D,B,F,E,A,G,C,L,J,H,K



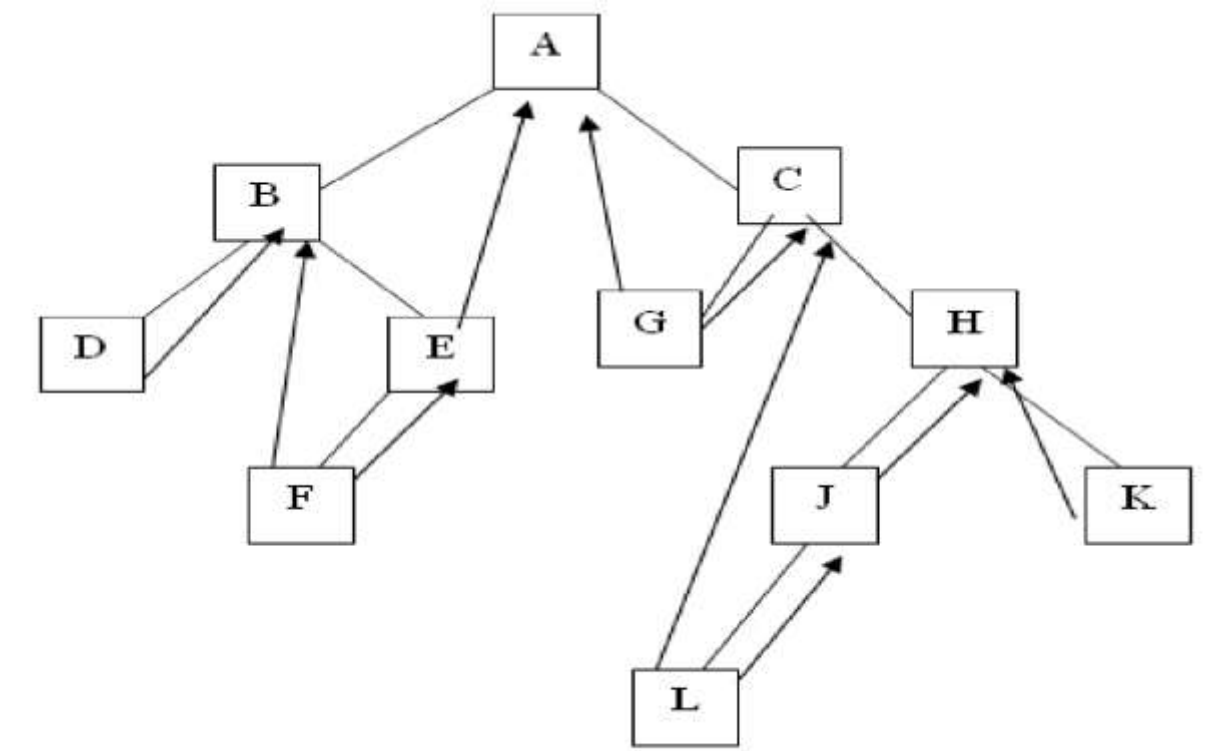
One-way inorder threading

TWO WAY THREADING

- A thread will also appear in the left field of a node and will point to the preceding node in the in-order traversal of tree T.
- Furthermore, the left pointer of the first node and the right pointer of the last node (in the in-order traversal of T) will contain the null value when T does not have a header node.

EXAMPLE

- Here, right pointer=next node of in-order traversal and left pointer=previous node of in-order traversal
- Inorder of bellow tree is: D,B,F,E,A,G,C,L,J,H,K



Two-way inorder threading

AVL TREE – (ADELSON, VELSKILL & LANDIS)

- AVL tree is a binary search tree except that for every node in the tree, the height of the left and right subtree can differ by almost 1.
- The height of the empty tree is defined to be -1.
- Balance factor:

Height of the Left Subtree - Height of the Right Subtree

AVL TREE – (ADELSON, VELSKILL & LANDIS)

- For an AVL tree all balance factor should be +1,0,-1.
- If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, then the tree has to be balanced by making either single or double rotations.

ROTATIONS

An AVL tree causes imbalance, when any one of the following conditions occur.

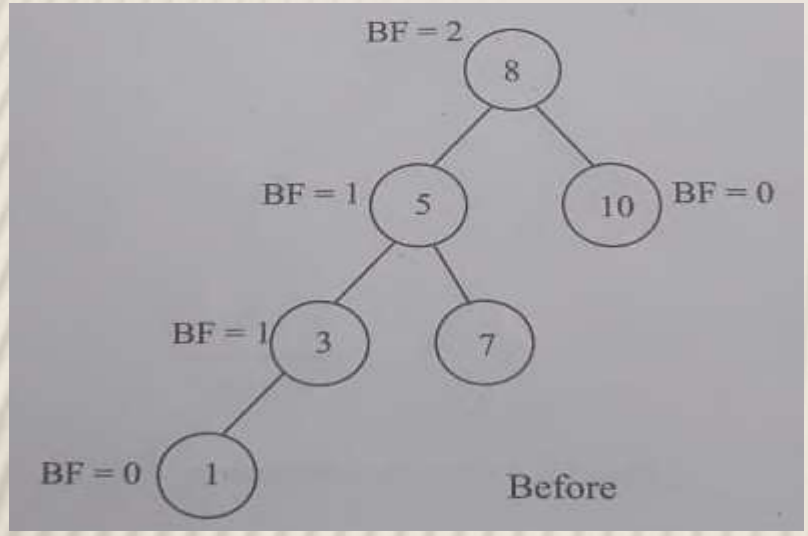
- Case 1: An insertion into the left subtree of the left child.
- Case 2: An insertion into the right subtree of the left child.
- Case 3: An insertion into the left subtree of the right child.
- Case 4: An insertion into the right subtree of the right child.

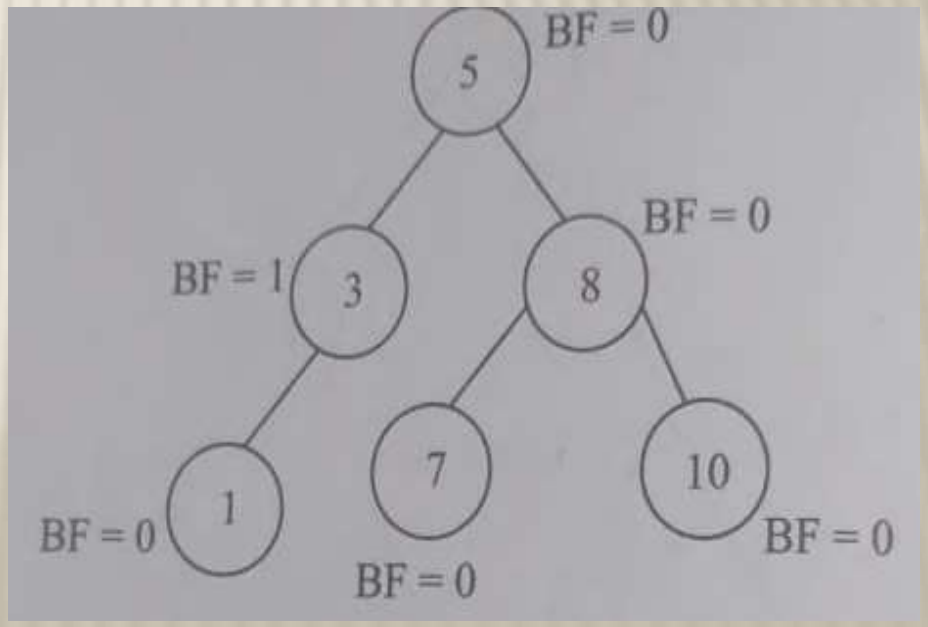
These imbalances can be overcome by

- **1.Single Rotation** – Performed to fix Case 1 and Case 4.
- **2.Double Rotation** – Performed to fix Case 2 and Case

CASE 1: AN INSERTION INTO THE LEFT SUBTREE OF THE LEFT CHILD.

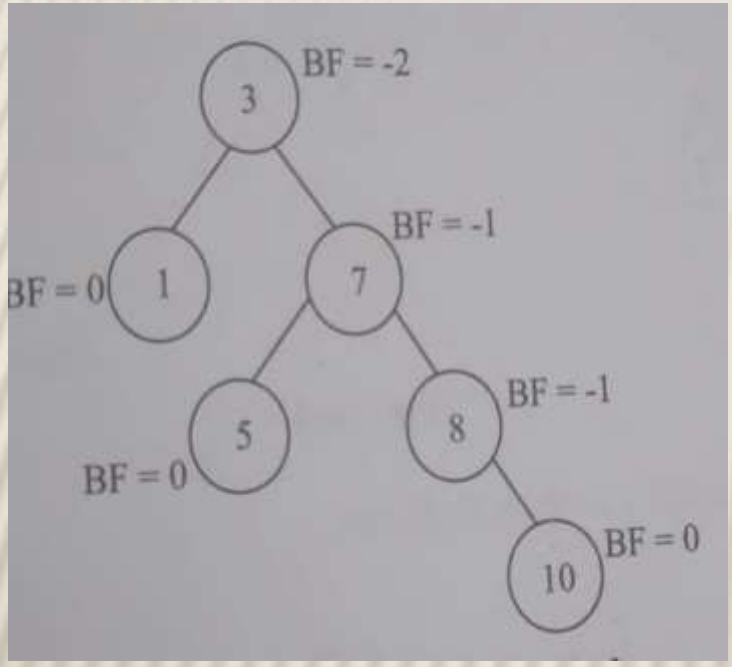
- To Insert :1

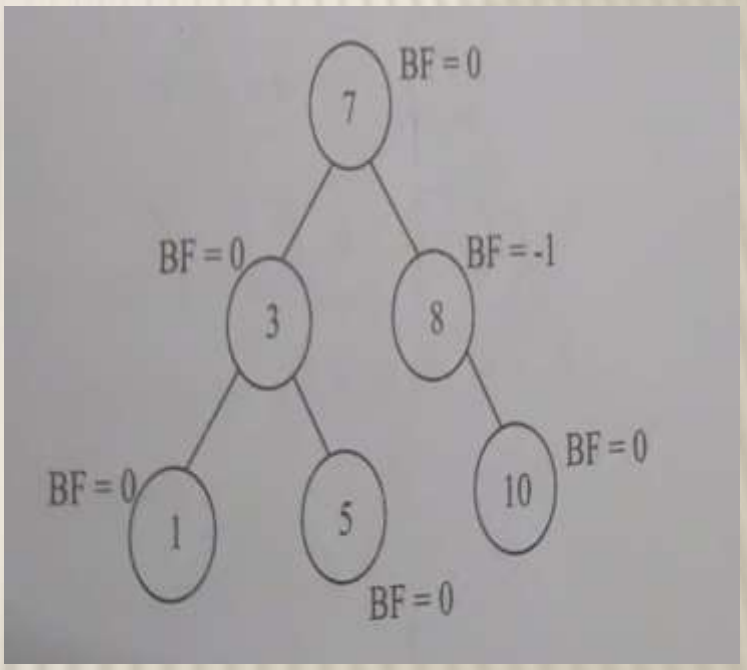




CASE 4: AN INSERTION INTO THE RIGHT SUBTREE OF THE RIGHT CHILD.

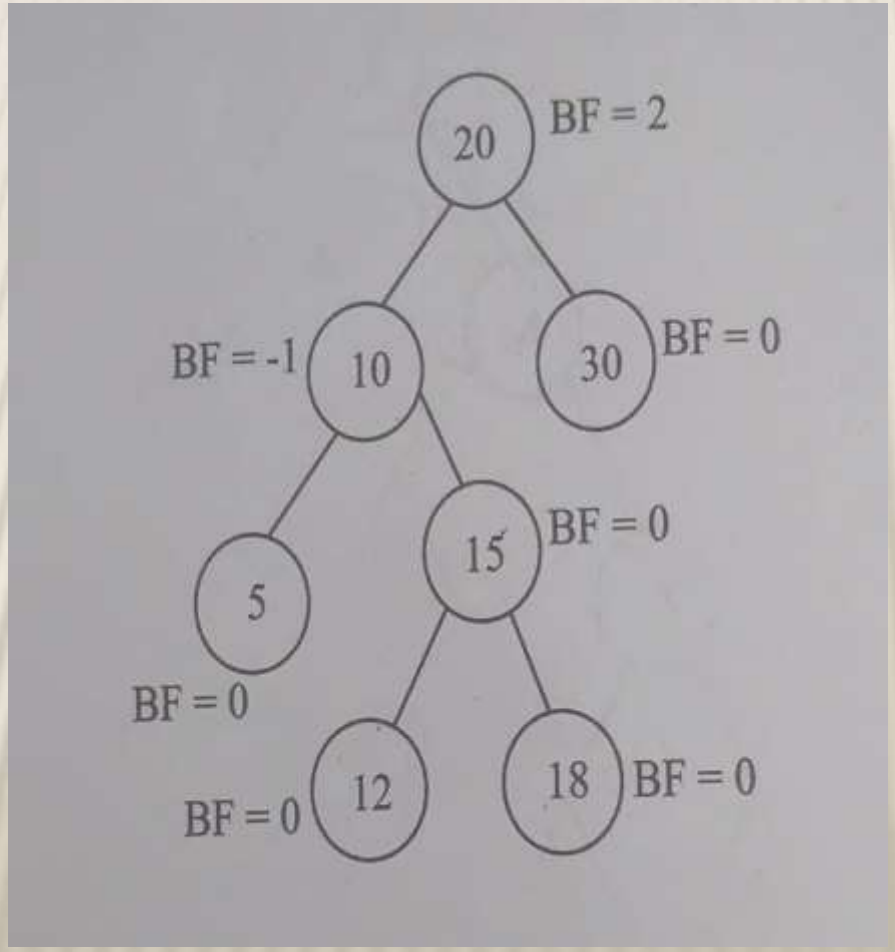
- To insert : 10



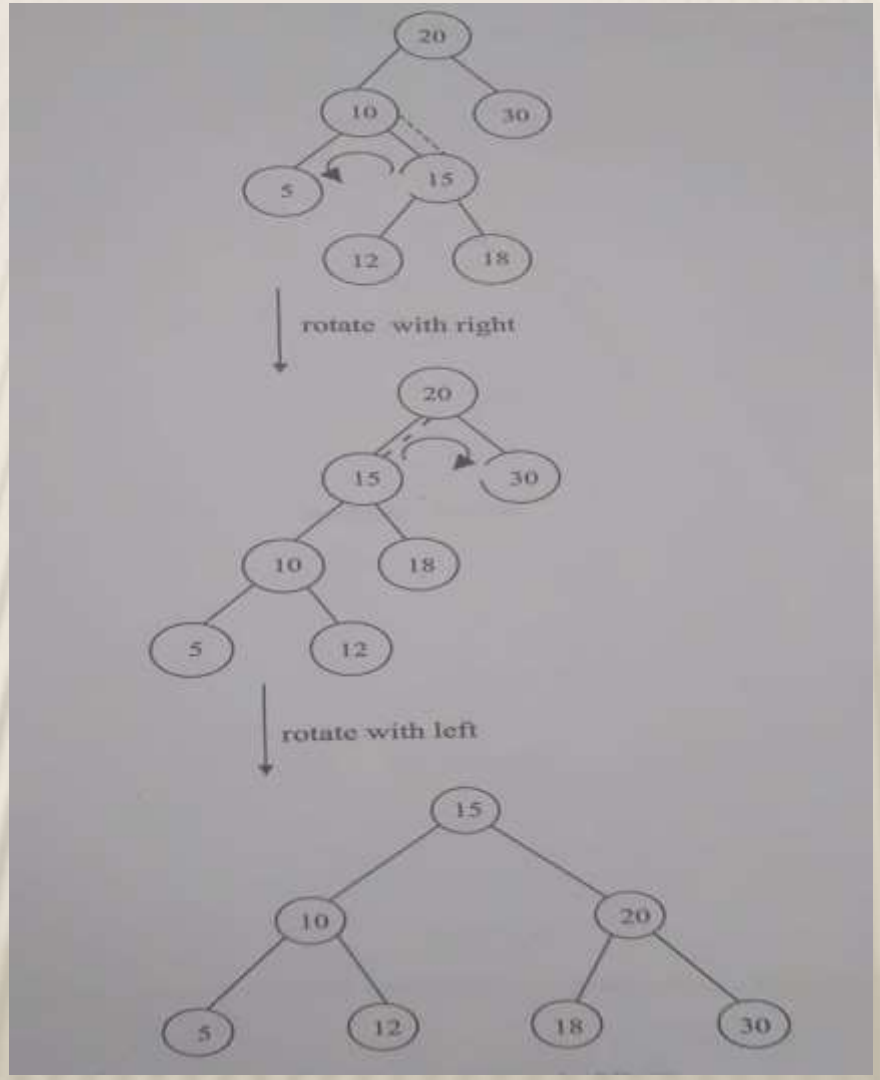


CASE 2: AN INSERTION INTO THE RIGHT SUBTREE OF THE LEFT CHILD.

- Insertion of either ’12’ and ‘18’

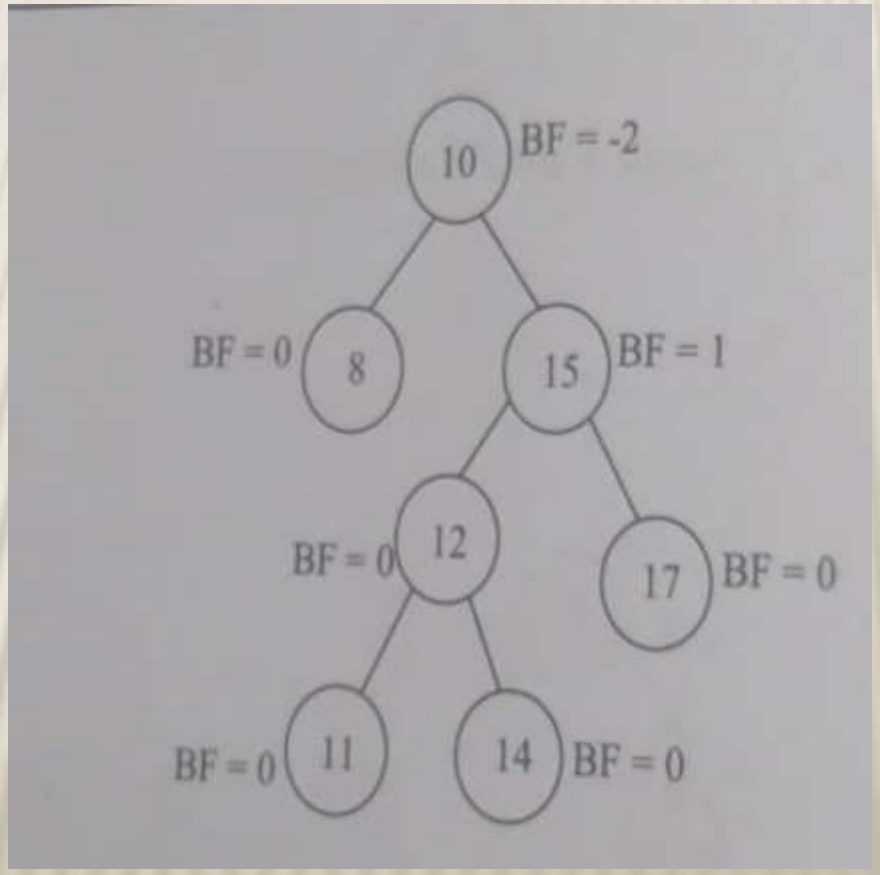


CASE 4: AN INSERTION INTO THE RIGHT SUBTREE OF THE RIGHT CHILD.

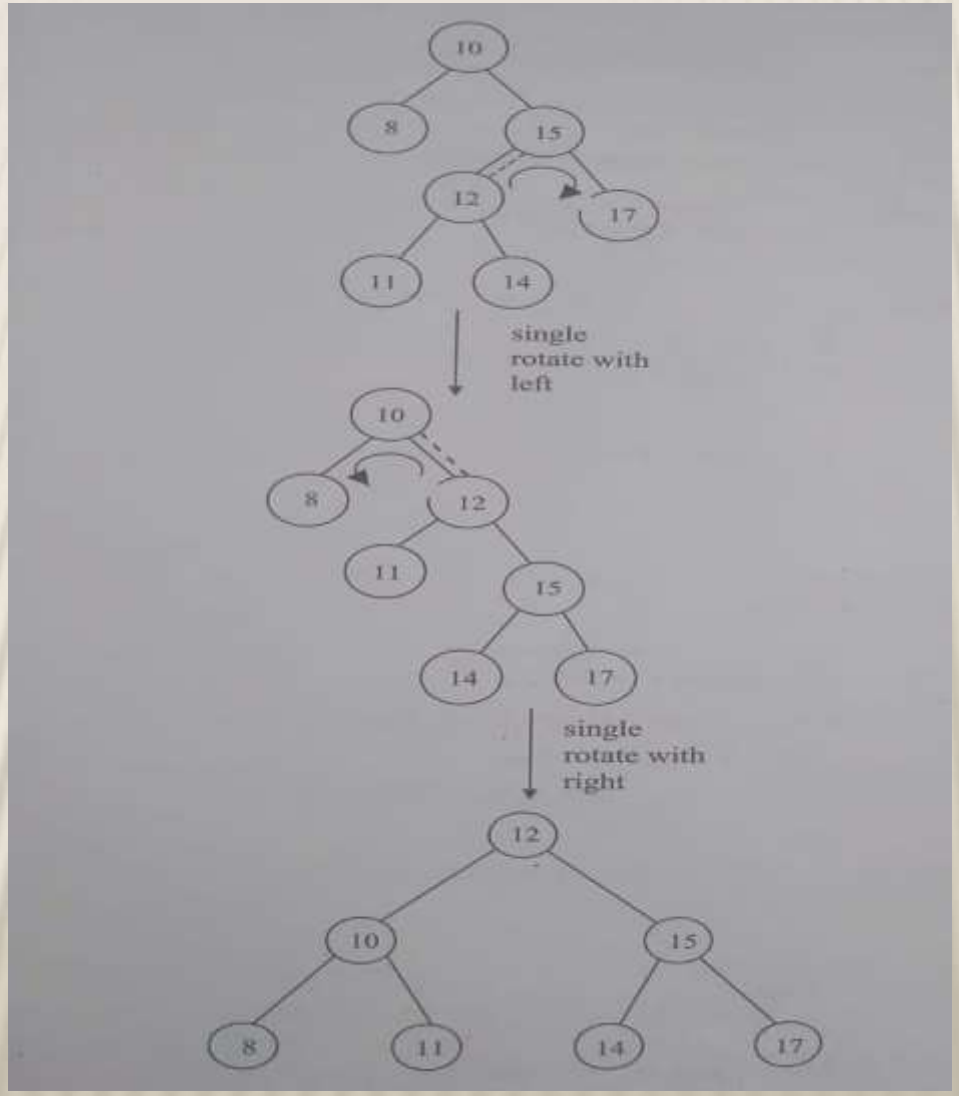


CASE 3: AN INSERTION INTO THE LEFT SUBTREE OF THE RIGHT CHILD.

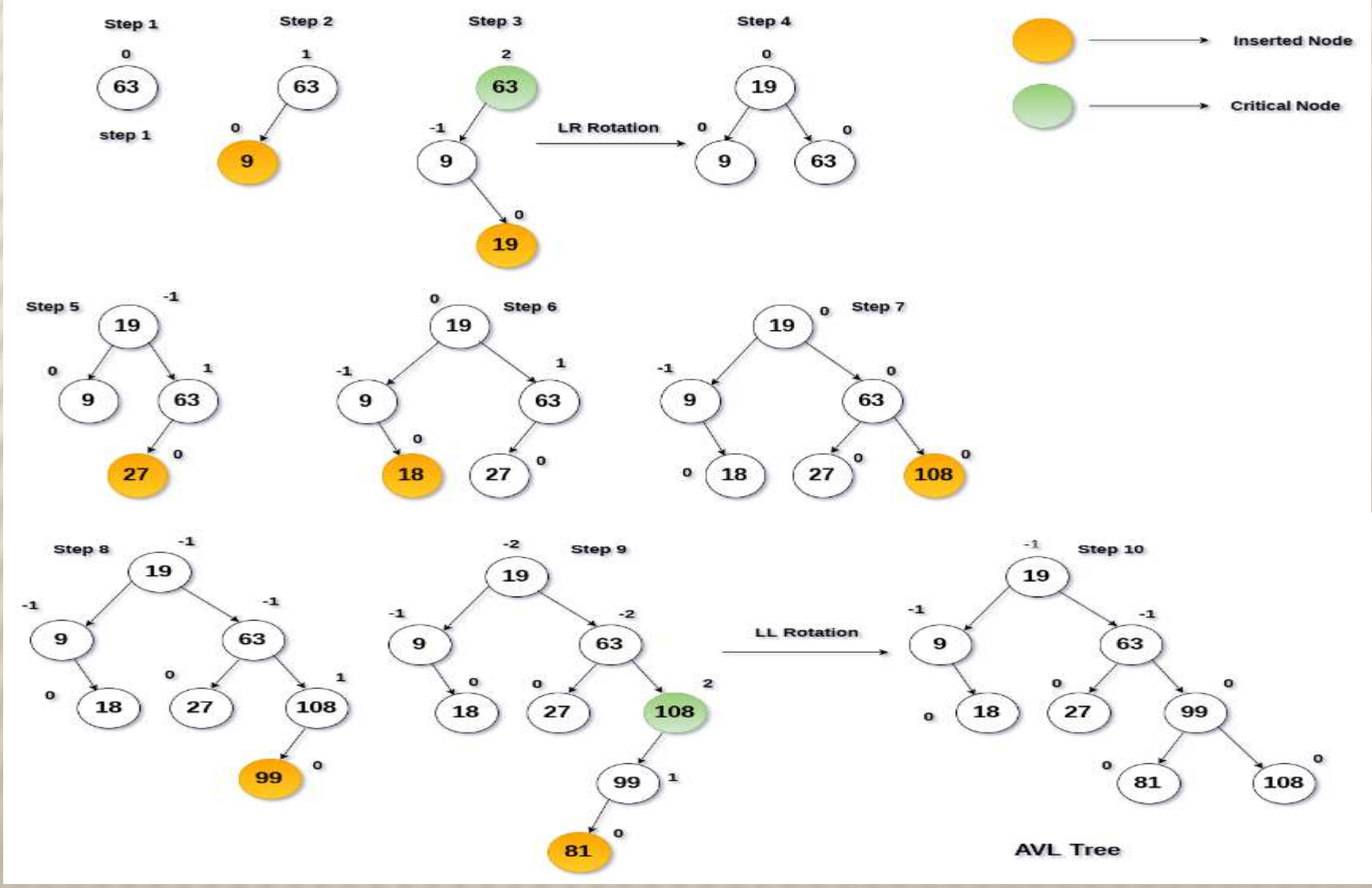
- To Insert : 11 and 14



CASE 3: AN INSERTION INTO THE LEFT SUBTREE OF THE RIGHT CHILD.



EXAMPLE


Step 1
Step 2
Step 3
Step 4
Inserted Node
Critical Node
LR Rotation
step 1
Step 5
Step 6
Step 7
Step 8
Step 9
Step 10
LL Rotation
AVL Tree

Routine for Insertion

```
AVLTree Insert(AVLTree T,int x)
{
  if(T==NULL)
  {
            T=malloc(sizeof(Struct AVLNode));
            if(T==NULL)
                  Error("Out Of Space");
            else
            {
                  T→ Data=X;
                  T→ Height=0;
                  T→ Left=NULL;
                  T→ Right = NULL;
            }
  }
```

Routine for Insertion - Contd.,

```
else
{
if(X<T→ Data)
{
   T→ Left=Insert(T→ Left,X);
   if(Height(T→ Left) - Height(T→ Right)==2)
                  if(X<T→ Left→ data)
                              T=SingleRotateWithLeft(T);
                  else
                              T=DoubleRotateWithLeft(T);
}
else
```

Routine for Insertion - Contd.,

```
if(X>T→ Data)
{
   T→ Right=Insert(T→ Right,X);
   if(Height(T→ Right) - Height(T→ Left)==2)
                  if(X>T→ Right→ data)
                                 T=SingleRotateWithRight(T);
                  else
                                 T=DoubleRotateWithRight(T);
}
T→ Height=Max(Height(T→ Left),Height(T→ Right))+1;
return T;
}
```

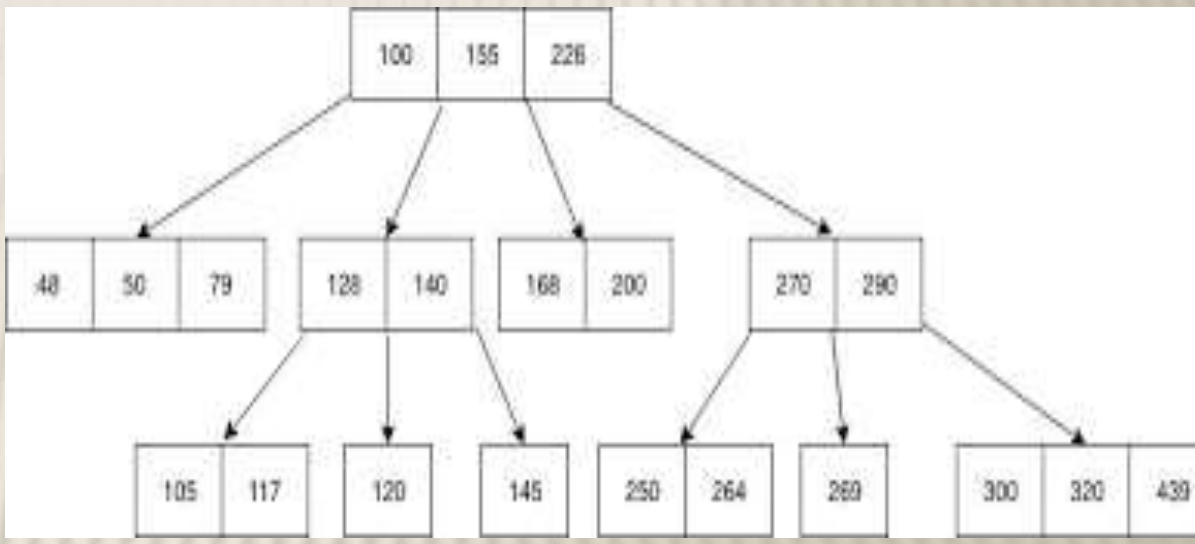
B-TREE

A B-Tree of order m is an m-way search tree with the following properties.

- The root node must have atleast 2 child nodes and atmost m child nodes.
- All internal nodes other than root node must hav e atleast m/2 to m non empty child nodes.
- The number of keys in each internal node is one less than its number of child nodes, which will partition the keys of the tree into subtree.
- All internal nodes are at the same level.

Example

- B-Tree is known as a self-balancing tree as its nodes are sorted in the inorder traversal.

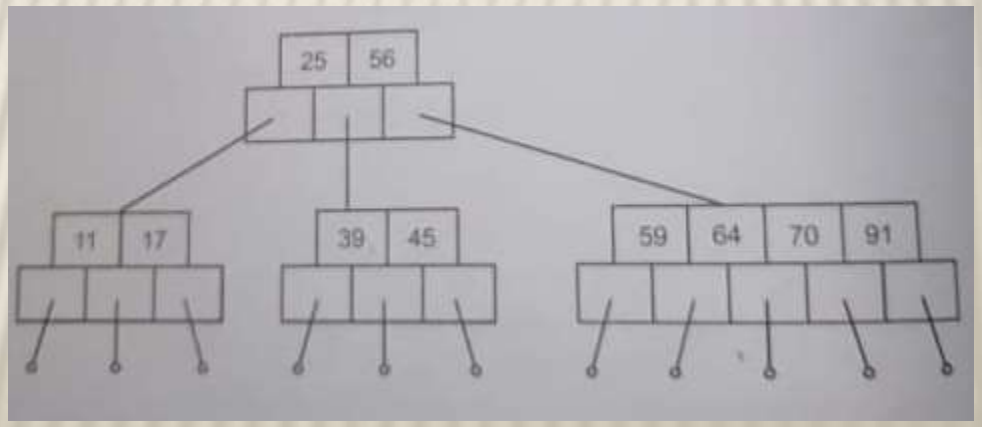


Operation on B-Tree

- Insertion
- Deletion

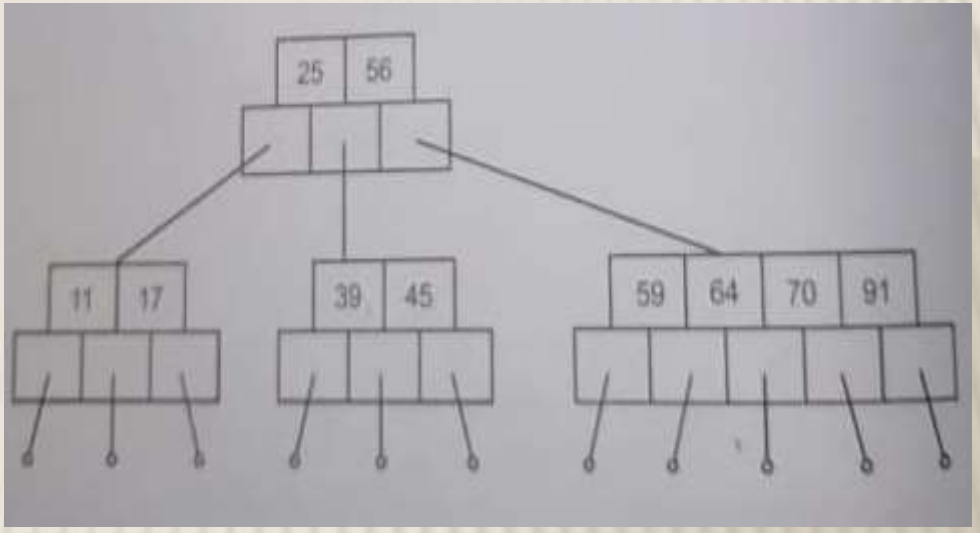
Insertion

- Case 1: When the node X of the B-Tree of order m can accommodate the key K, then it is inserted in that node and the number of the child pointer fields are appropriately upgraded.

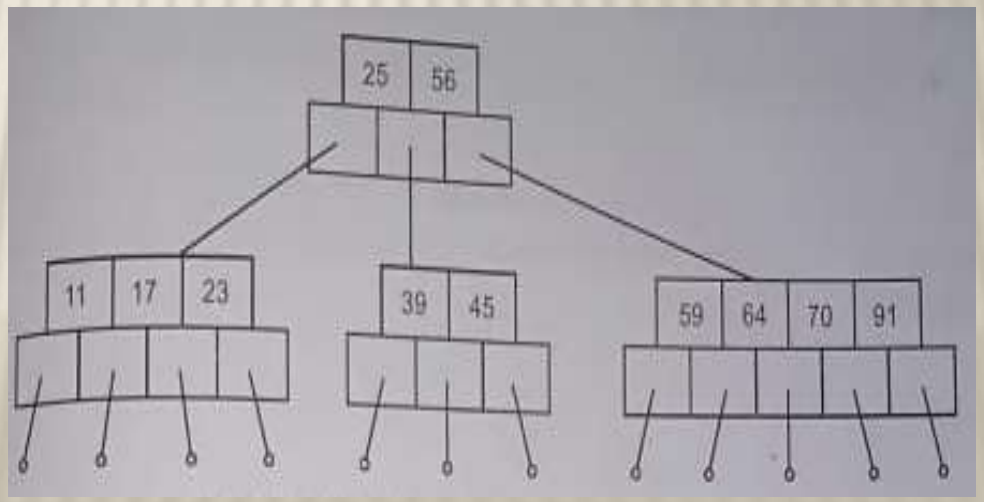


Case 1 - Example

- M=5
- Before Insertion



- To Insert : 23

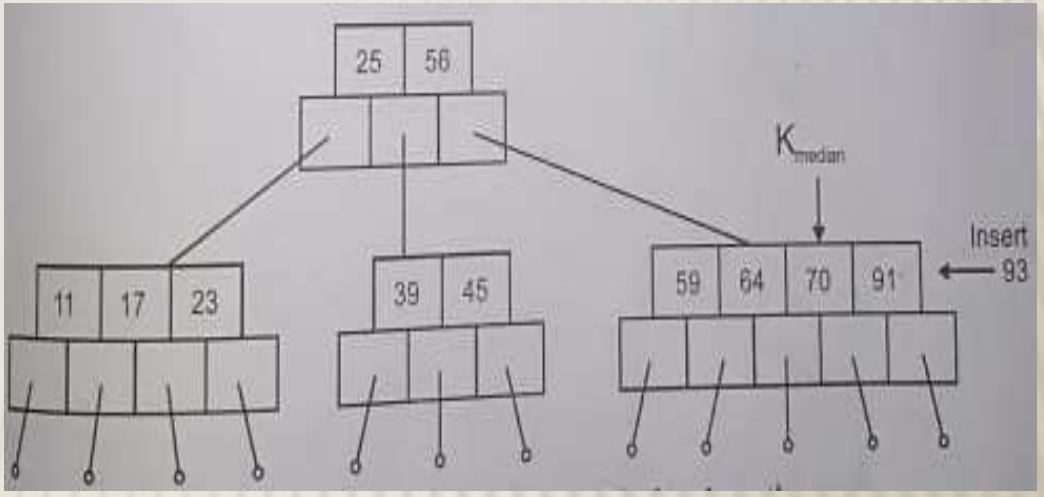


Case 2

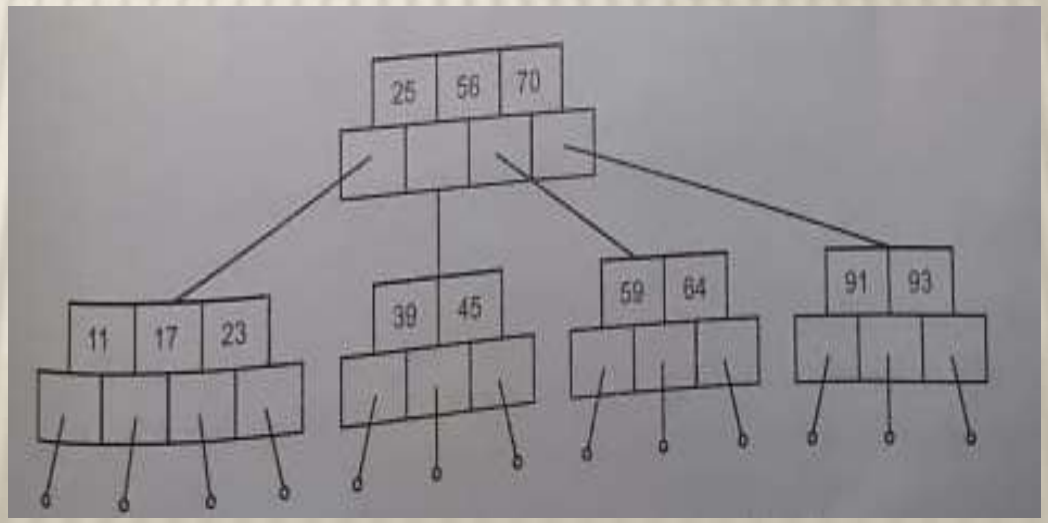
- If the node is full then the key K is apparently inserted into the list of elements and the list is splitted into two on the same level at it median(K_{median}).
- The keys which are less than K_{median} are placed in the X_{left} and those greater than Kmedian are placed at X_{right}
- The median key is not placed into either of the two new nodes, but instead moved up the tree to be inserted into the parent node of X.
- This insertion inturn will call case 1 and 2 depending upon whether the parent node can accommodate or not.

Case 2 - Example

- Before Insertion
- To insert 93



- After Insertion

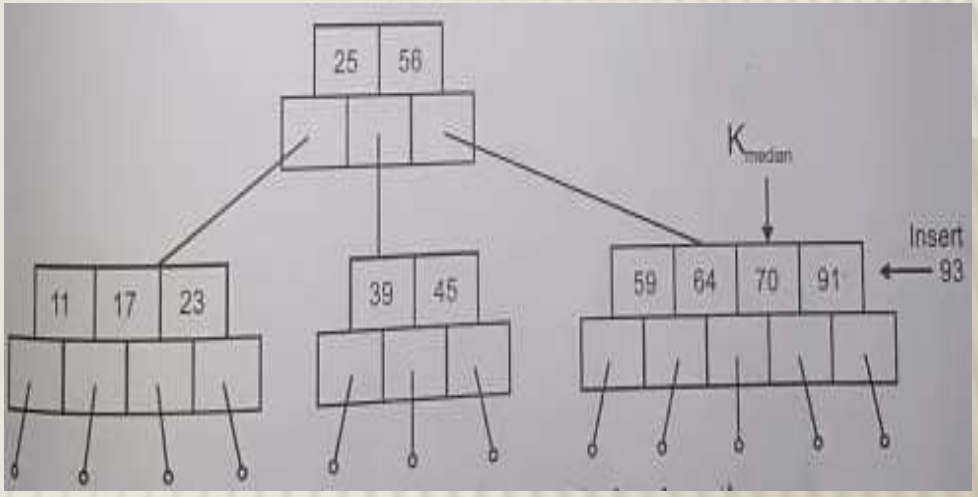


Deletion – Case 1

- If the key K to be deleted belongs to a leaf nodes and its deletion does not result in the node having less than its minimum number of elements.
- Then delete the key from the leaf and adjust child pointers.

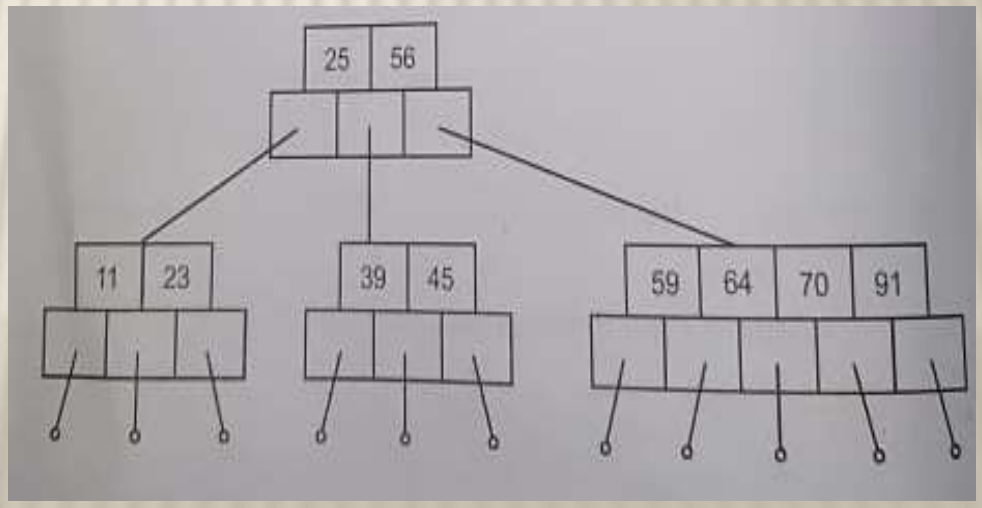
Case 1 - Example

- Before Deletion



- To delete: 17

- After Deletion

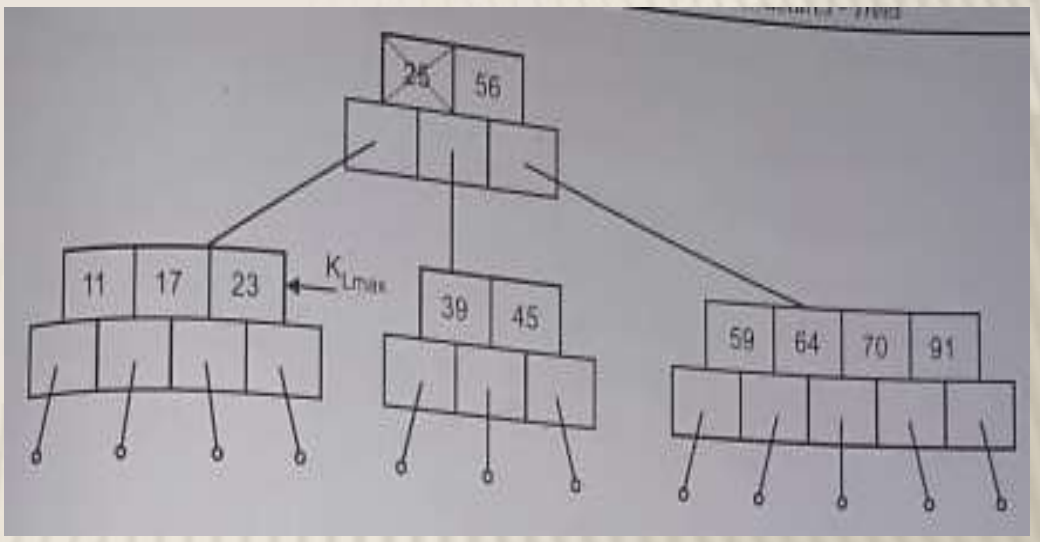


Case 2

- If the key belongs to a non leaf node.
- Then replace K with largest key K_{Lmax} in the left subtree of K or the smallest key K_{Rmin} from the right subtree of K and then delete K_{Lmax} or K_{Rmin} from the node, which in turn will trigger case 1 or 2.

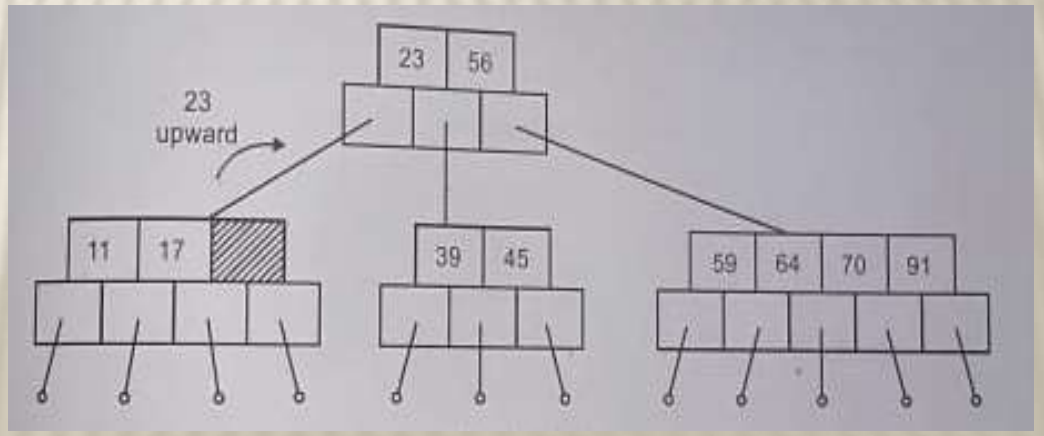
Case 2 - Example

- Before Deletion



- To Delete : 25

- After Deletion

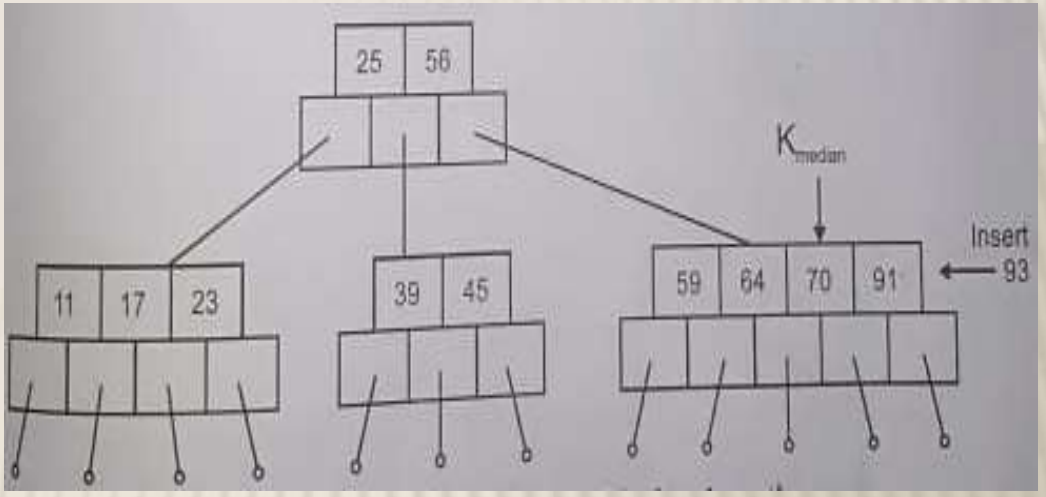


Case 3

- If the key K to be deleted from a node leaves it with less than its minimum number of elements, then the elements may be borrowed either from left or right sibling.
- If the left sibling node has an element to spare, then move the largest key K_{Lmax} in the left sibling node to the parent node and the element P in the parent is moved down to set the vacancy by the deletion of K in node X.
- If the left sibling node has no element to spare then move to case 4.

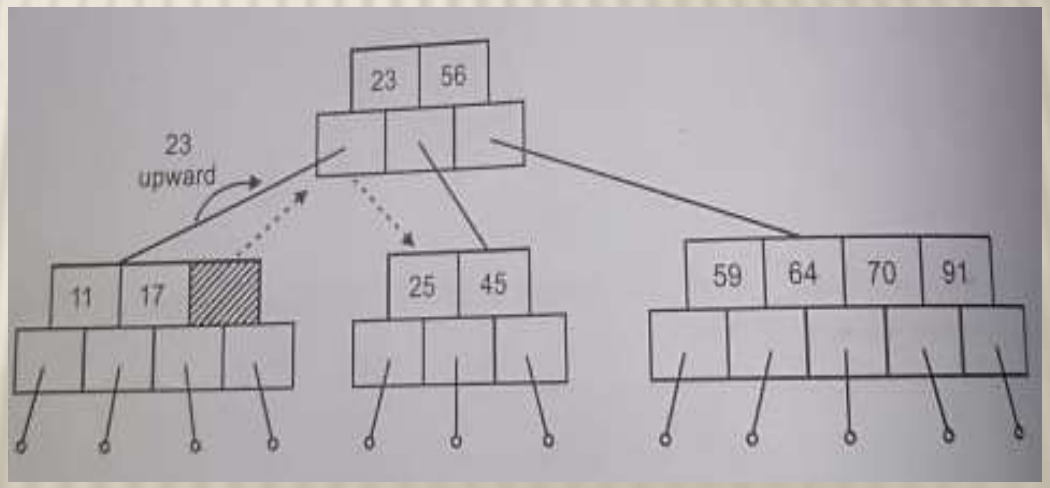
Case 3 - Example

- Before Deletion



- To Delete: 39

- After Deletion :

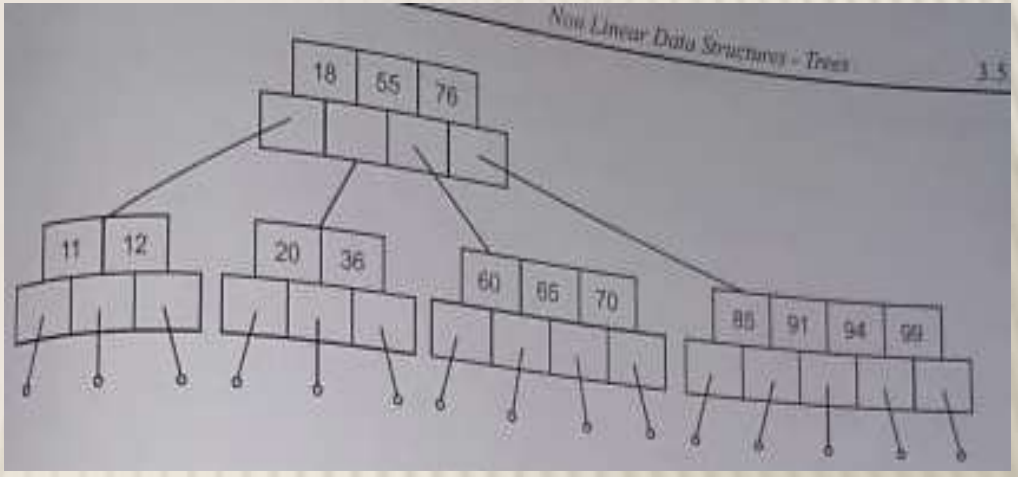


Case 4

- If the key to be deleted from a node X leaves it with less than its minimum number of elements and both the sibling nodes are unable to spare an element.
- Then the node X is merged with one of the sibling nodes along with intervening element P in the parent node.

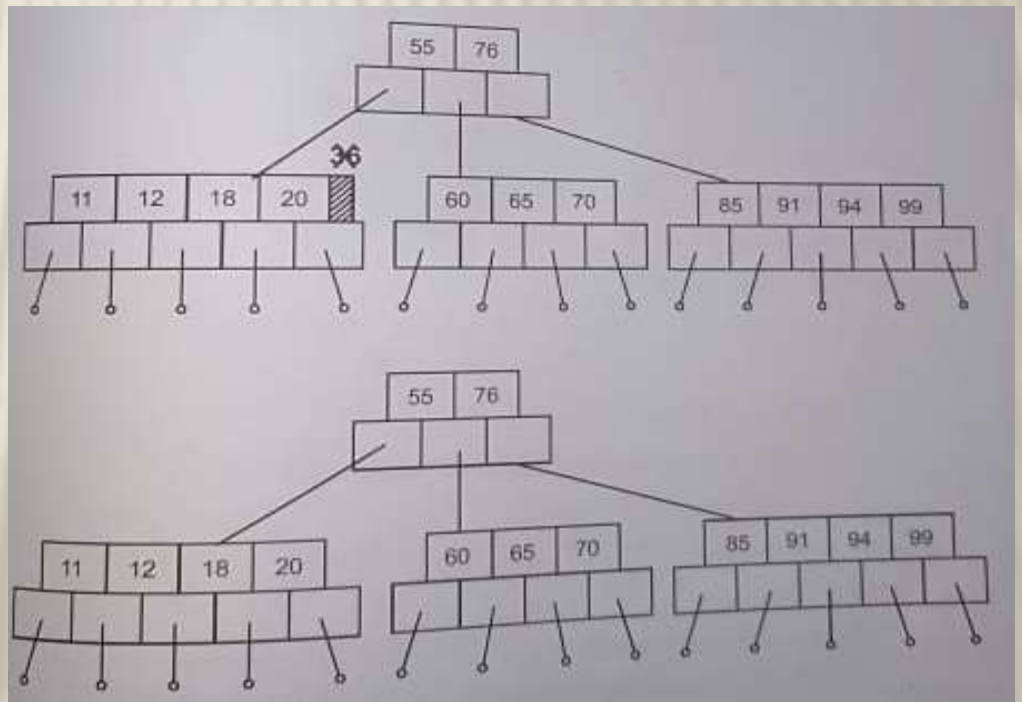
Case 4 – Example

- Before Deletion



- To Delete : 36

- After Deletion



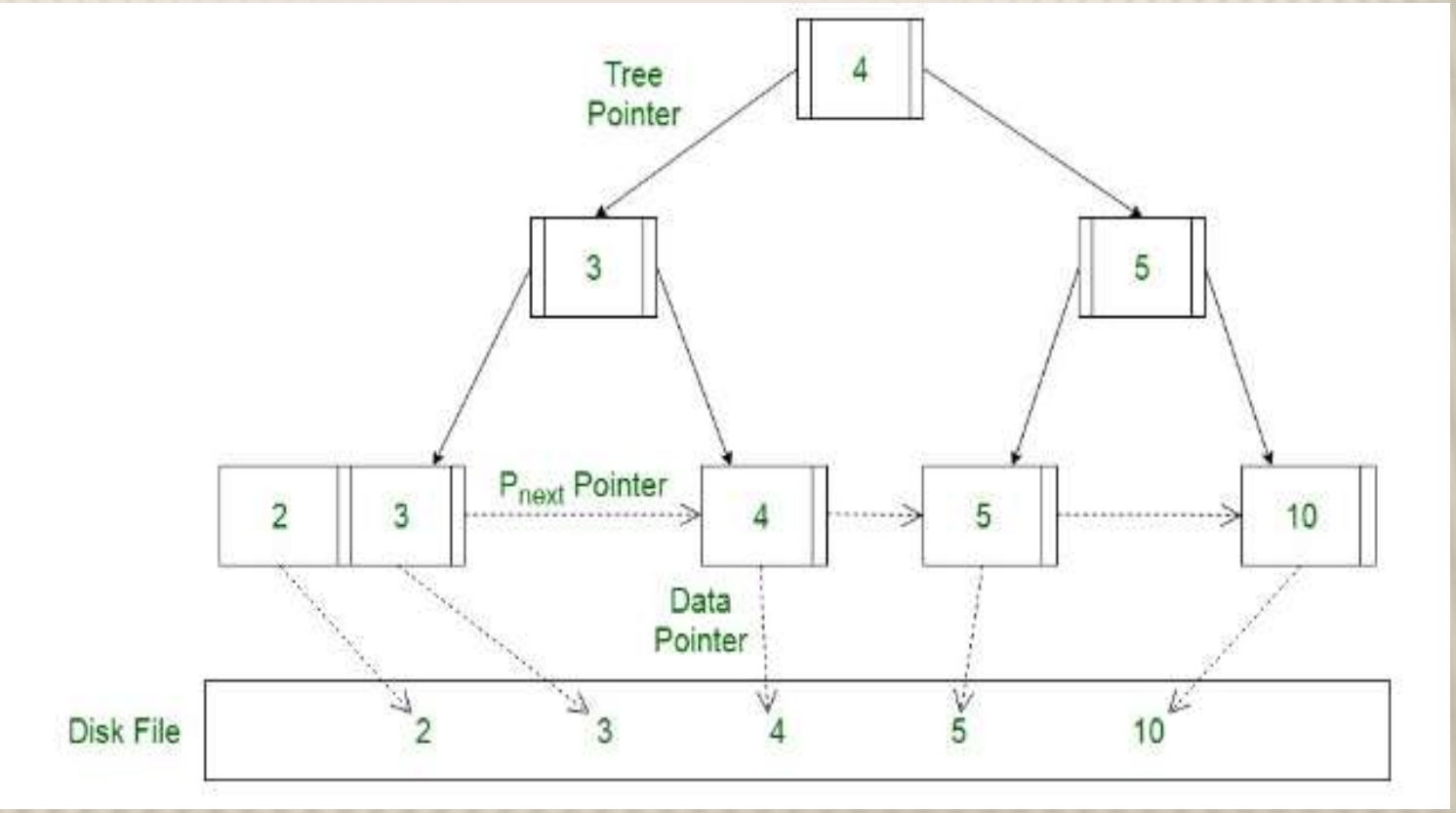
B+ Tree

- B+ tree eliminates the drawback B-tree used for indexing by storing data pointers only at the leaf nodes of the tree.
- The structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree.
- It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

B+ Tree

- Moreover, the leaf nodes are linked to providing ordered access to the records.
- The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index.
- Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

Example


Tree
Pointer
4
3
5
2
3
P_{next} Pointer
4
5
10
Data
Pointer
Disk File
2
3
4
5
10

B Tree Vs. B+ Tree

S. No	B Tree	B+ Tree
1	All internal and leaf nodes have data pointers.	Only leaf nodes have data pointers.
2	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and accurate.
3	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at leaf.
4	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
5	Deletion of internal node is very complex and tree has to undergo lot of transformations.	Deletion of any node is easy because all node are found at leaf.
6	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
7	No redundant search keys are present.	Redundant search keys may be present.