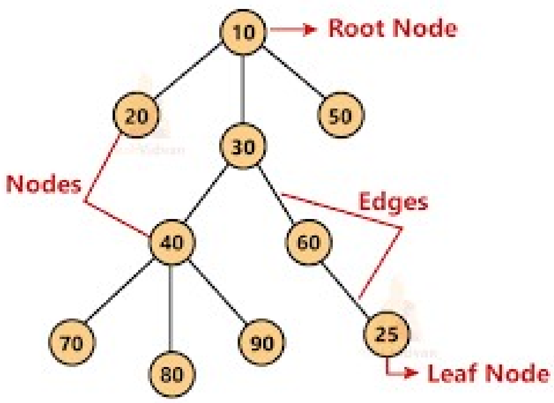
DATA STRUCTURE

TREE UNIT - 5

AKTU

Tree

- A tree is a nonlinear hierarchical data structure .
- It consists of nodes connected by edges.
- In which , easy to navigate and search.

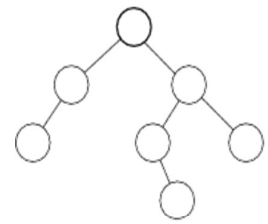


Terminologies of Tree

- **Node**
  - A node is an entity that contains a key or value and pointers to its child nodes.
- **Edge**
  - It is the link between any two nodes.
- **Root**
  - It is the topmost node of a tree. [10]
- **Child node:**
  - Any subnode of a given node is called a child node.
  - Ex: 20 & 30 & 50 are children of 10 etc.
- **Parent:**
  - If node contains any sub-node, then node is called parent of that sub-node.
  - Ex: 30 is parent of 60 and 40 is parent of 70 etc.
- **Sibling:**
  - The nodes that have the same parent are known as siblings.
- **Leaf Node:-**
  - The node of the tree, which doesn't have any child node
  - Leaf nodes can also be called external nodes.
  - Ex : 70 ,80, 90 , 25,20, 50
- **Internal nodes:**
  - A node has atleast one child node .
  - Ex: 10 , 30 , 40 , 60
- **Height of a Node**
  - The height of a node is the number of edges from the node to the deepest leaf
  - Ex:height(30) =2 &height(20) = 0 &height(10) =3
- **Depth of a Node**
  - The depth of a node is the number of edges from the root to the node.
  - Ex : depth(30) =1 & depth(80) = 3 depth(10) =0
- **Height of a Tree**
  - The height of a Tree is the height of the root node .
  - Ex: Height(root=10) = 3

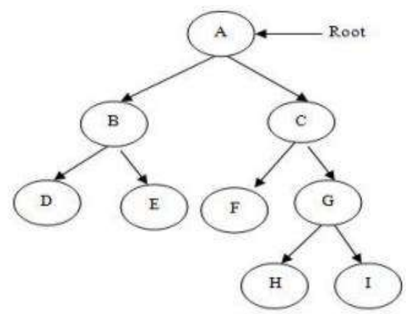
Binary tree:

- When any tree has at most two child , those tree is said to be binary tree.



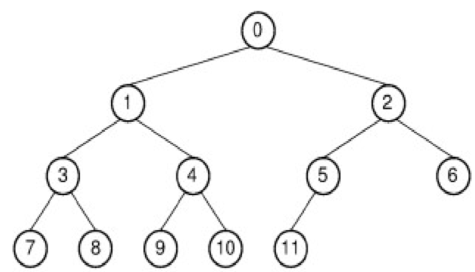
Strictly binary tree

- A strictly binary tree is a binary tree in which each node has either 0 or 2 children, i.e., no node has only one child.



Complete binary tree

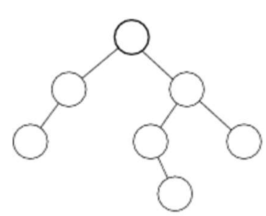
- All levels are filled, except possibly the last.
- Nodes are filled from left to right on all levels.

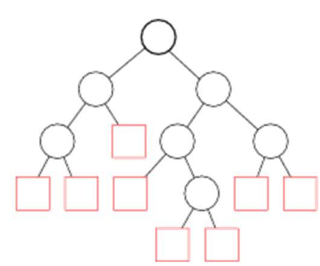


(a)

Extended binary tree

- A binary tree T is said to be 2-tree or extended binary tree if each node
- has either 0 or 2 children.
- b. Nodes with 2 children are called internal nodes and nodes with 0 children
- are called external nodes.





Binary tree

extended binary tree

Representation of Binary Tree using linked list

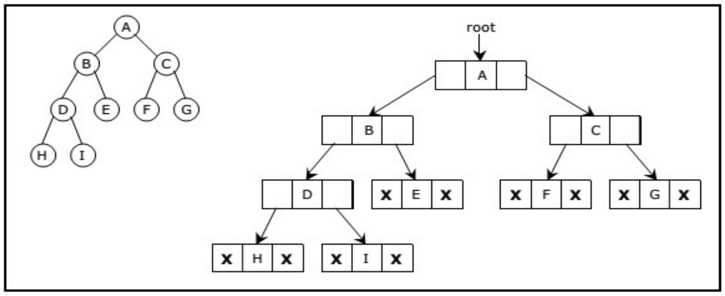
- **Node structure :**

Each node in tree is represented by an object .

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

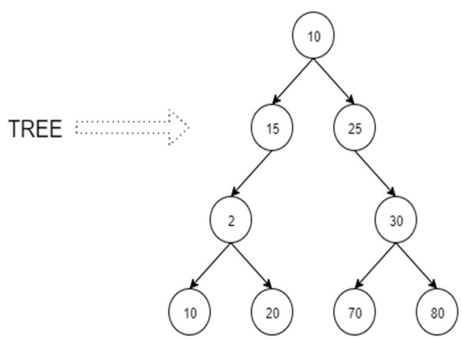
left	data	right

- **Data** : the value stored in node .
- **Left child**: pointer of left subtree of that node
- **Right child** : pointer of right subtree of that node



Representation of Binary Tree using array

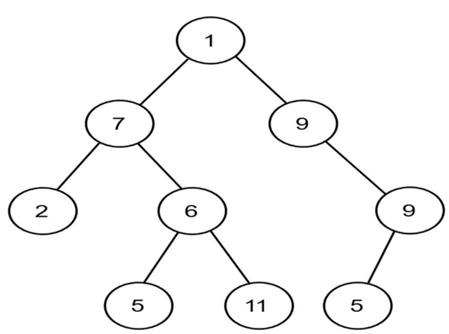
- **Root Node :** index 0 of the array
- **Parent-child relationship :** for any element at index I, its left child is at index 2* I +1 and its right child is at index 2*1 +2



SEQUENTIAL REPRESENTATION

10	15	25	2	-	-	30	10	20	-	-	-	-	70	80

Traversal of binary Tree



- Tree traversal is the process of visiting and processing each node in a tree data structure.
- The three main types of tree traversal are:
- **In-order:** 2 7 5 6 11 1 5 9 9
- **Pre-order:** 1 7 2 6 5 11 9 9 5
- **Post-order**: 2 5 11 6 7 5 9 9 1

In-order traversal: (L N R)

Algorithm :

1. Traverse the left sub-tree.
2. Visit the root node.
3. Traverse the right sub-tree.

Pseudo-code :

```
void in-order(struct node *root)
{
    if(root!= NULL)
    {
        in-order(root→ left);
        printf("%d",root→ data);
        in-order(tree→ right);
    }
}
```

Pre-order traversal: (N L R)

Algorithm :

1.Visit the root node.
2.Traverse the left subtree.
3.Traverse the right subtree.

Pseudo-code :

```
void in-order(struct node *root)
{
    if(root!= NULL)
    {
  printf("%d",root→ data);
        in-order(root→ left);

        in-order(tree→ right);
    }
}
```

Post-order traversal: (L R N)

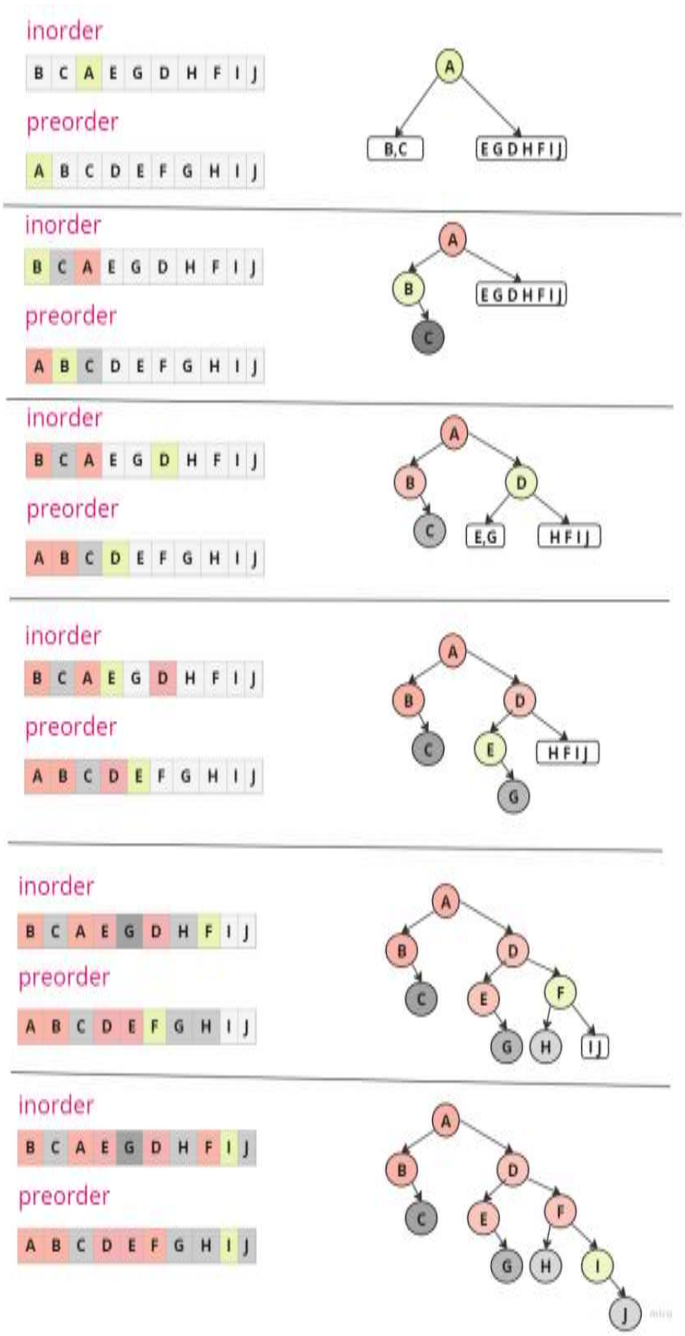
Algorithm :

1.Traverse the left subtree.
2.Traverse the right subtree.
3.Visit the root node.

Pseudo-code :

```
void in-order(struct node *root)
{
    if(root!= NULL)
    {
        in-order(root→ left);

        in-order(tree→ right);
      printf("%d",root→ data);

    }
}
```

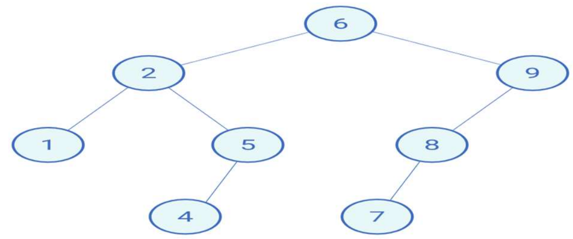
Draw a binary tree with following traversals :
In-order : B C A E G D H F I J
Pre-order : A B C D E F G H I J
Find the post-order of the tree.



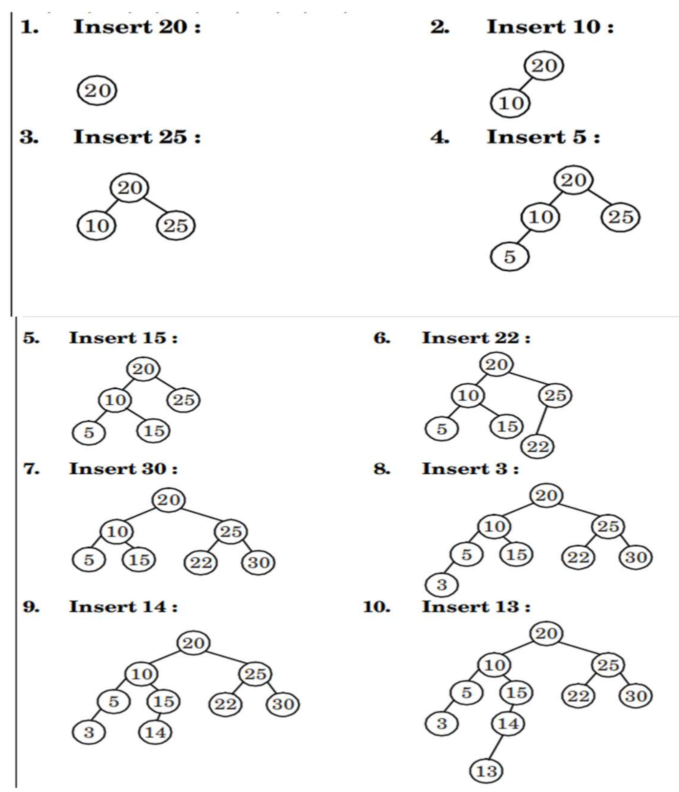
Postorder of tree : C B G E H J I F D A

Binary search tree

- A binary search tree is a binary tree.
- Each node has at most two children.
- The value of the left child is smaller than the parent node.
- The value of the right child is greater than the parent node.
- This ordering is applied recursively to left and right subtrees.
- In BST,No two elements share the same value.
- Efficient for searching, insertion, and deletion operations.
- In-order traversal results in sorted order.

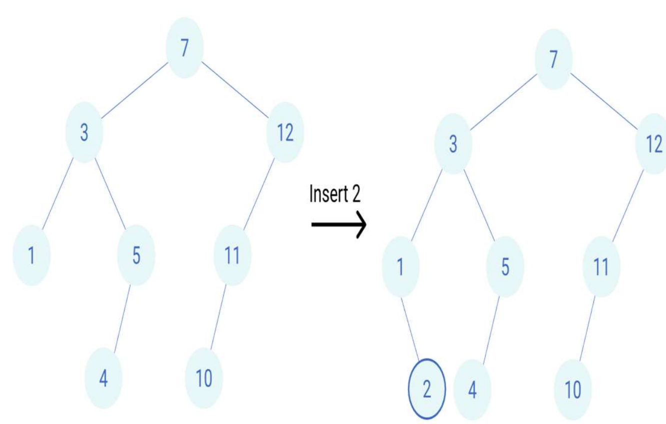


Create BST for the following data, show all steps :
20, 10, 25, 5, 15, 22, 30, 3, 14, 13



▪ Insertion in BST:

1.if root== null, create BST node with key and return the node pointer.
2.If root.key > key , recursively insert the new node to the left subtree.
3.If root.key < key , recursively insert the new node to the right subtree.



Algorithm

```
insert(root, key):
   if root is null:
      return new Node(key)

   if key < root.key:
      root.left = insert(root.left, key)
   else if key > root.key:
      root.right = insert(root.right, key)
   return root
```

- **Deletion in BST:**

Case 1: No Children (Leaf Node)

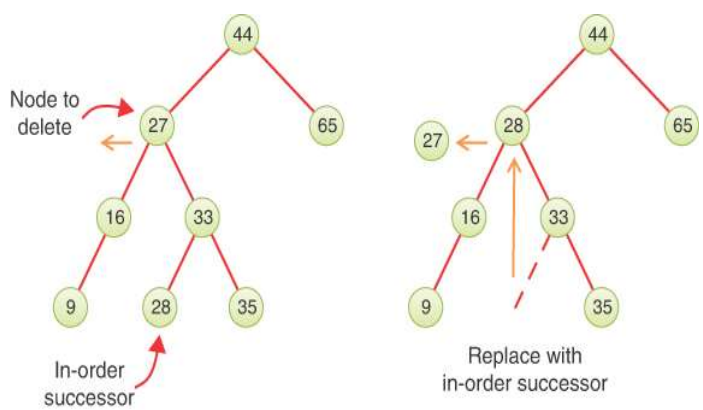
If the node to be deleted (N) has no children, delete it by replacing its parent's pointer with a null pointer

Case 2: One Child

If the node to be deleted (N) has exactly one child, delete it by replacing its parent's pointer with the pointer to its only child.

Case 3: Two Children

If the node to be deleted (N) has two children, find its in-order successor (S(N)), delete S(N) using Case 1 or Case 2, and then replace N with S(N).



Algorithm:

```
function deleteNode(root, key):
   if root is null:
      return root  // Key not found, no deletion

   // Case 1: No Children (Leaf Node)
   if root.key equals key and root.left is null and
root.right is null:
      return null

   // Recursive cases
   if key < root.key:
      root.left = deleteNode(root.left, key)
   else if key > root.key:
      root.right = deleteNode(root.right, key)
   else:
      // Case 2: One Child
      if root.left is null:
         return root.right
      else if root.right is null:
         return root.left

      // Case 3: Two Children
      successor = findMin(root.right)
      root.key = successor.key
      root.right = deleteNode(root.right, successor.key)
   return root

function findMin(node):
   // Helper function to find the node with the minimum
key in a BST
   while node.left is not null:
      node = node.left
   return node
```

- **Searching in BST:**

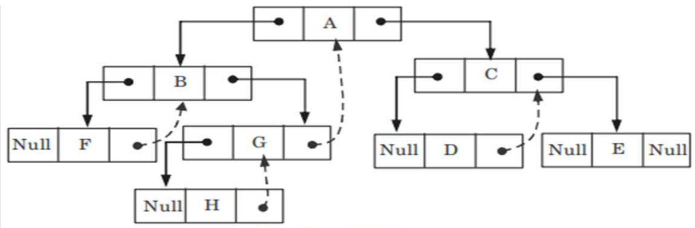
1. Searching for a key in a Binary Search Tree (BST) involves traversing the tree in a way that takes advantage of its structure.
2. The key property of a BST is that for each node:
   I. All nodes in its left subtree have keys less than the node's key.
   II. All nodes in its right subtree have keys greater than the node's key.

algorithm

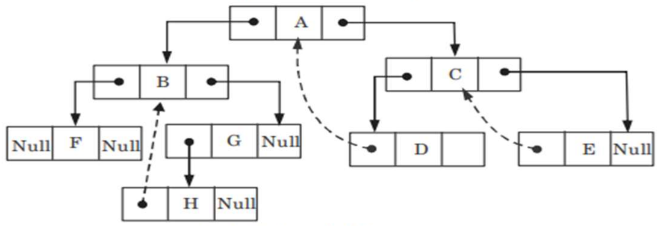
```
function searchBST(root, key):
   // Base case: If the tree is empty or the key is found
   if root is null or root.key equals key:
      return root

   // If the key is smaller, search in the left subtree
   if key < root.key:
      return searchBST(root.left, key)
   // If the key is larger, search in the right subtree
   else:
      return searchBST(root.right, key)
```

Threaded Binary tree

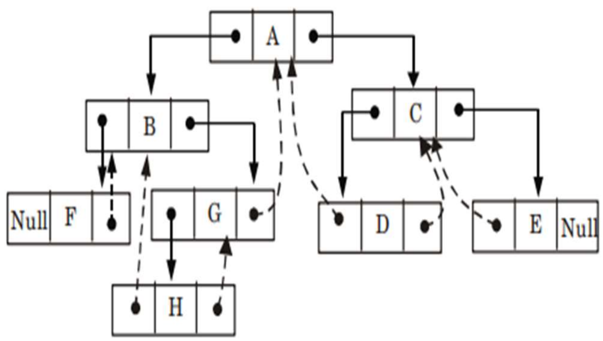
- A binary tree in which some nodes have additional pointers (threads) that allow for faster in-order traversals.
- Threads are typically used to avoid the need for recursion or a stack during in-order traversal.
- Threads can be of two types: left threads and right threads.
- A node with a left thread points to its in-order predecessor, and a node with a right thread points to its in-order successor.



(a) Right threaded binary tree.



(b) Left threaded binary tree

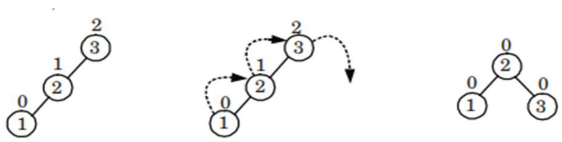


(c) Fully threaded binary tree

AVL TREE

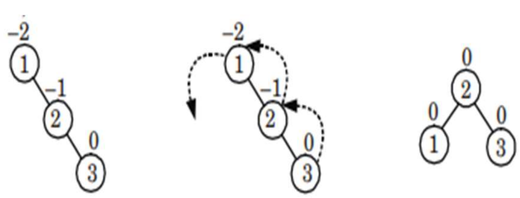
- An AVL (**Adelson-Velsky and Landis tree**) tree is a balanced binary search tree.
- It is a special type of binary search tree.
- In an AVL tree, balance factor of every node is either –1, 0 or +1.
- Balance factor of a node is the difference between the heights of left and right subtrees of that node.
- **Balance factor = height of left subtree – height of right subtree**
- In order to balance a tree, there are four cases of rotations :

1. Left Left rotation (LL rotation)



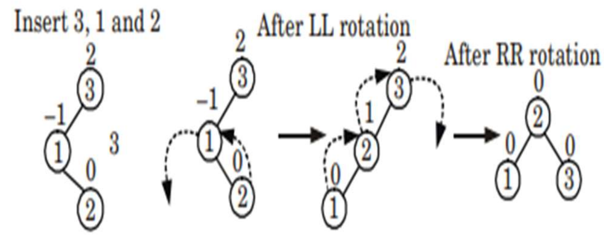
Unbalanced LL rotation balanced

2. Right Right rotation (RR rotation)

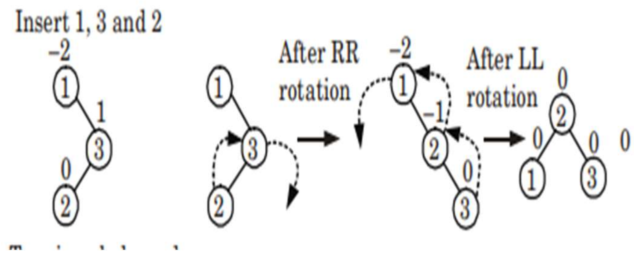


Unbalanced RR rotation balanced

3.Left Right rotation (LR rotation)

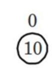


4. Right left rotation (RL rotation)

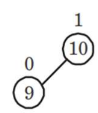


Create an AVL tree for the following elements : 10,9,8,7,6,5,4,3,2,1

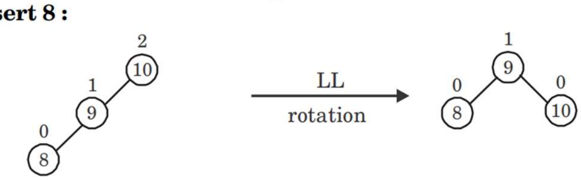
Insert 10 :



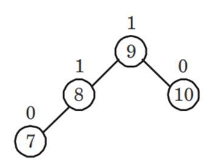
Insert 9 :



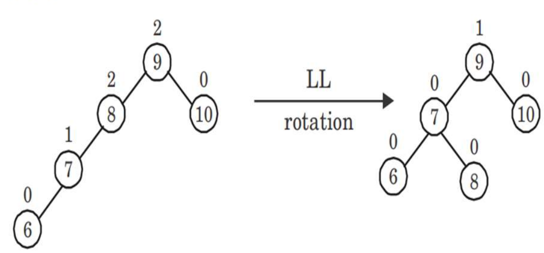
Insert 8 :



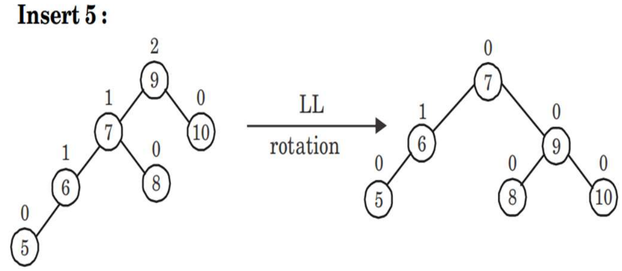
Insert 7 :



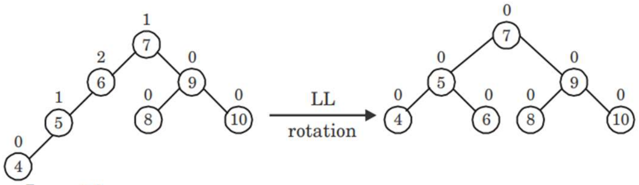
Insert 6 :



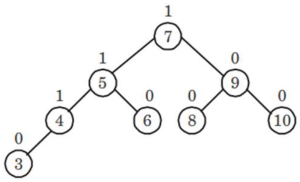
Insert 5 :



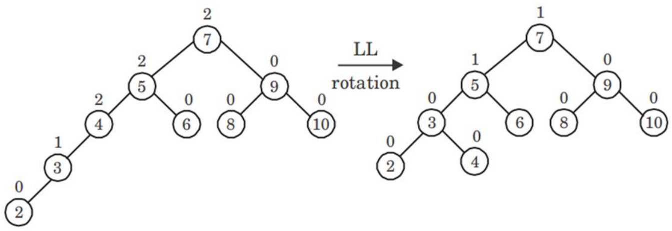
Insert 4 :



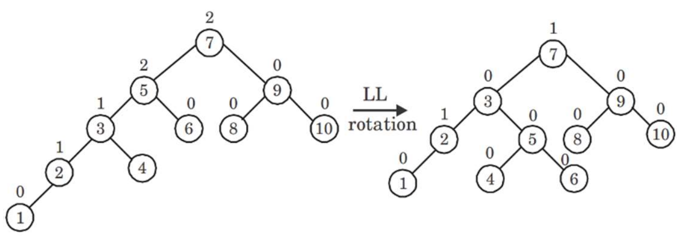
Insert 3 :



Insert 2 :



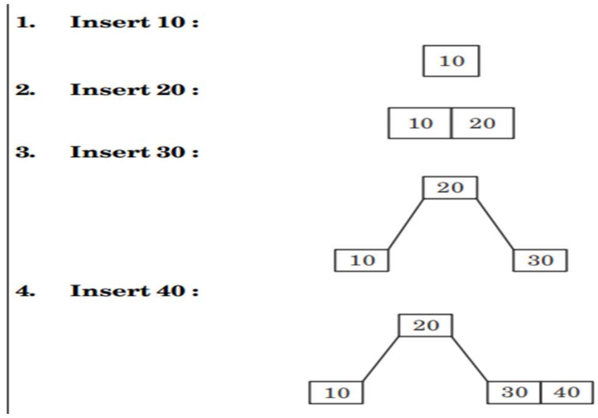
Insert 1 :

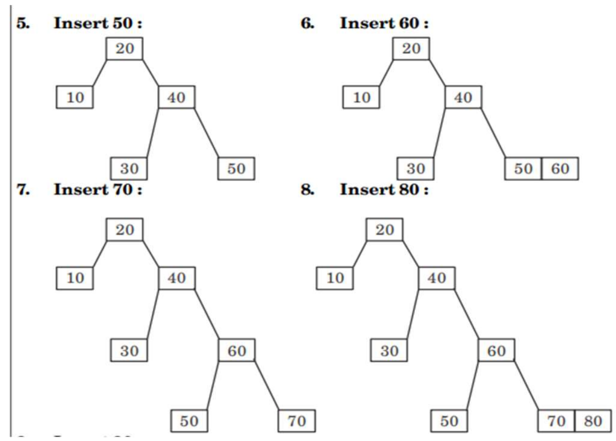


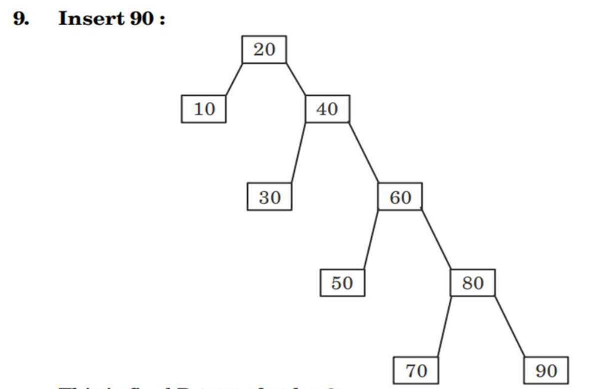
B-tree

- A B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- **A B-tree of order m is a tree which satisfies the following properties :**
  - Every node has at most m children
  - Every non-leaf node has at least m/2 children.
  - The root has at least two children if it is not a leaf node.
  - A non-leaf node with k children contains k – 1 keys.
  - All leaves appear in the same level.

Construct a B-tree of order 3p created by inserting the following elements
10,20,30,40,50,60,70,80,90







This is final B-tree of order 3.