Module 2:

Introduction to Trees

CO2: Classify, **apply** and **analyze** the concepts of trees in real life problem solving.

Subject In-charge

Ms.Pratibha Rane

Assistant Professor

Class : III SEM SEIT- B

Module 2: Introduction to Trees

CO2: **Classify**, **apply** and **analyze** the concepts trees in real life problem solving.

Introduction to Trees: Terminology, Types of Binary trees. Non recursive Preorder, in-order and post-order traversal. Creation of binary trees from the traversal of binary trees. Binary search tree: Traversal, searching, insertion and deletion in binary search tree.

Threaded Binary Tree: Finding in-order successor and predecessor of a node in threaded tree. Insertion and deletion in threaded binary tree.

AVL Tree: Searching and traversing in AVL trees. Tree Rotations: Right Rotation, Left Rotation. Insertion and Deletion in an AVL Tree.

B-tree: Searching, Insertion, Deletion from leaf node and nonleaf node. B+ Tree, Digital Search Tree, Game Tree & Decision Tree

Self-learning Topics: Implementation of AVL and B+ Tree

Chapter 9 & 10: Reema Thareja; Data Structures using C; Oxford

A tree data structure 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**.

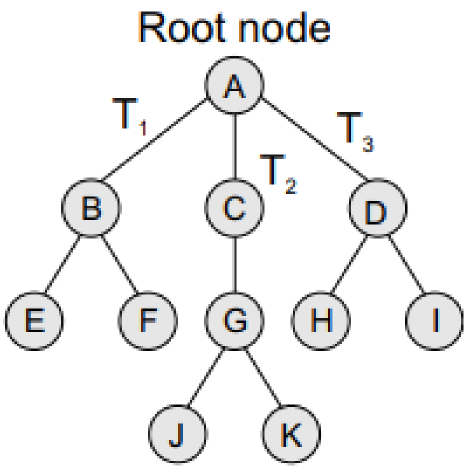


Figure 9.1 Tree

Advantage :

- To represent structural relationship, hierarchical relationships
- Flexible to add and delete data

Basic Terminology

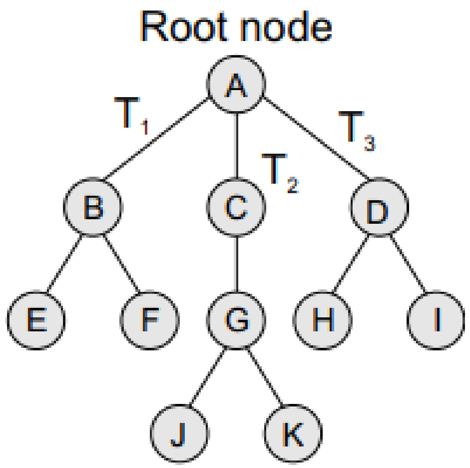


Figure 9.1 Tree

Node: every element int the tree

Root node: topmost element (only one root)

Child Node: a node which has link from parent node, node on left link is called left child and node on right link is called right child

Leaf node: lower nodes who doesn’t have child nodes

Path: A sequence of consecutive edges is called a path.

Ancestor node : predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig. 9.1, nodes A, C, and G are the ancestors of node K.

Descendant node

A descendant node: is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig. 9.1, nodes C, G, J, and K are the descendants of node A.

Subtree: every child from a node forms a subtree

Basic Terminology

Level number Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent’s level number + 1.

Degree of a node : maximum number of children that a node has. The degree of a leaf node is zero.

In-degree In-degree of a node is the number of edges arriving at that node.

Out-degree Out-degree of a node is the number of edges leaving that node.

Depth or height of a tree: maximum level of any leaf node .

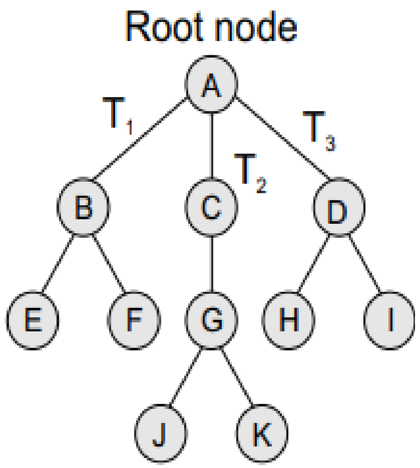


Figure 9.1 Tree

Types of tree

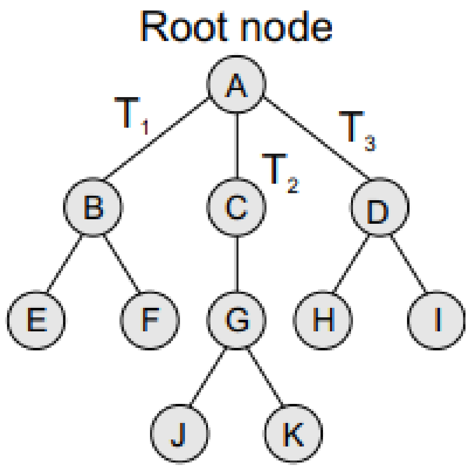


Figure 9.1 Tree

1. **General tree:** tree with multiple nodes
2. **Forests:** A forest is a disjoint union of trees
3. **Binary tree:** specialized tree data structure in which every node is allowed to have maximum of two child nodes .
4. **Binary search tree:** A binary search tree, also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in an order

Binary tree

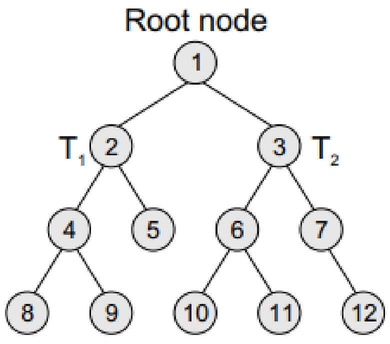


Figure 9.3 Binary tree

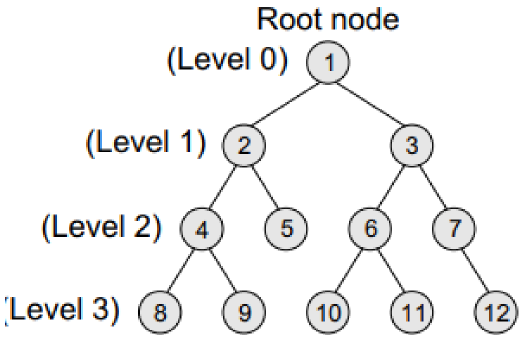


Figure 9.4 Levels in binary tree

1. A tree in which every node has atmost **two children**
2. A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes
3. So, if every level has two nodes then a tree with height h will have at the most $2^h - 1$ nodes as at level 0, there is only one element called the root
4. The height of a binary tree with n nodes is at least $\log_2 (n+1)$ and at most n.
5. Types of binary tree:

- Strictly binary tree
- Complete binary tree
- Almost complete binary tree
- Skewed Binary tree
- Extended binary tree

Binary tree

- **Extended binary tree:**

- **A full binary tree b**
- **By Adding Dummy nodes wherever required**

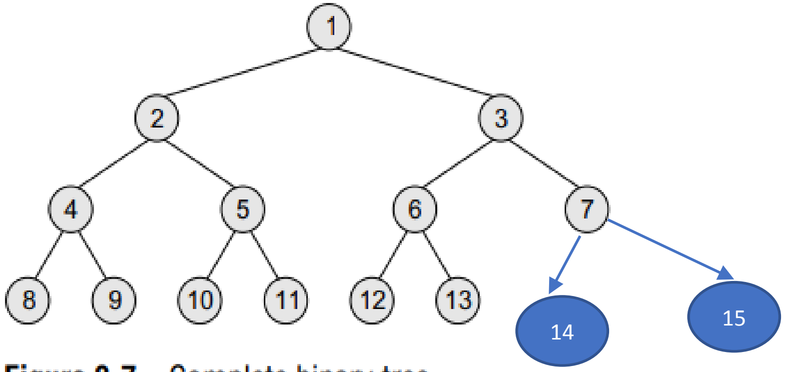


Figure 9.7 Complete binary tree

Binary tree

- **Strict binary tree:**

- **Every node is must to have exactly two child or zero nodes**
- **Also called full binary or 2-Tree**

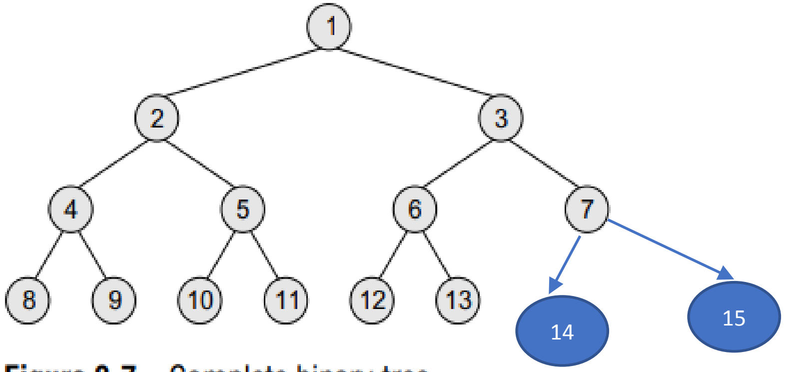
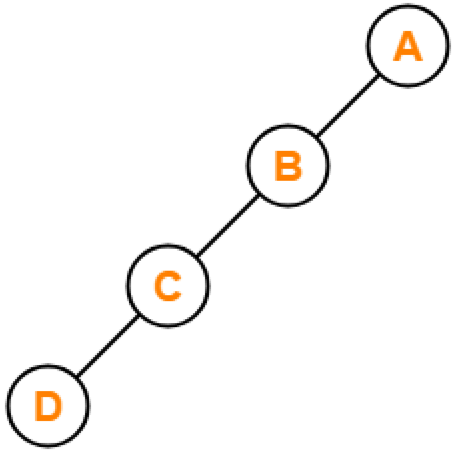


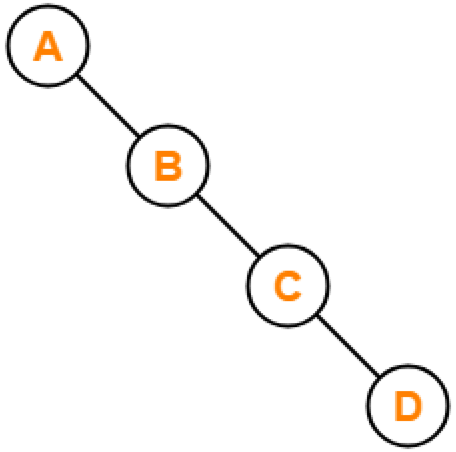
Figure 9.7 Complete binary tree

Binary tree

- **Skewed binary tree:**
- Which is either only left branches (left skewed) are present or only right branches(right skewed) are present



Left Skewed Binary Tree



Right Skewed Binary Tree

Representation of Binary tree in the memory

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

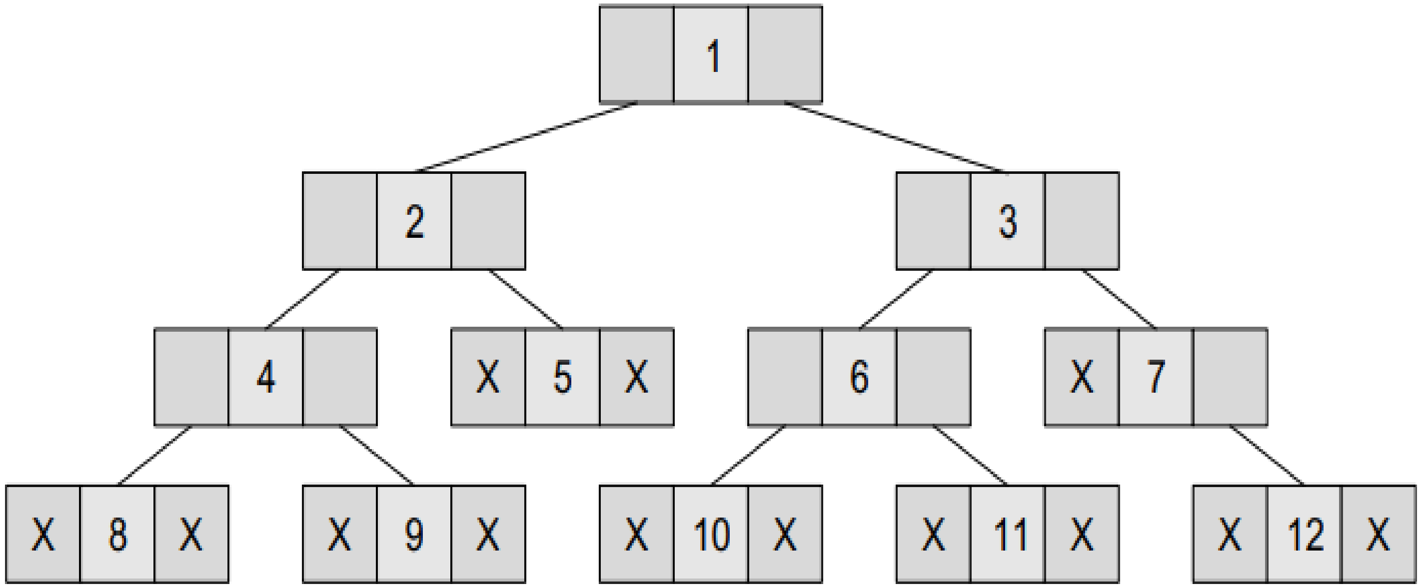


Figure 9.9 Linked representation of a binary tree

Look at the tree given in Fig. 9.10. Note how this tree is represented in the main memory using a linked list (Fig. 9.11).

Traversing a Binary tree

visiting each node in the tree exactly once

Types: (name based on order of Rootnode visit)

Pre-order traversal

In-order traversal

Post-order traversal

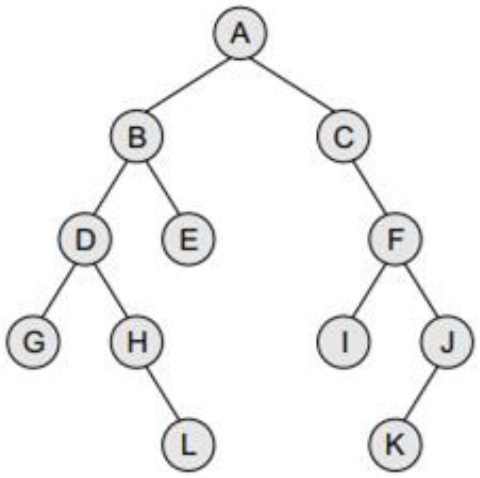
In-order Traversing a Binary tree

The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.17 Algorithm for in-order traversal



(a)

find the sequence of nodes that will be visited using in-order traversal algorithm.

TRAVERSAL ORDER: G, D, H, L, B, E, A, C, I, F, K, and J

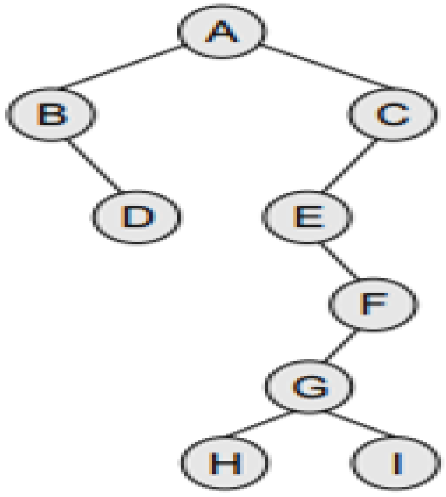
In-order Traversing a Binary tree

The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             INORDER(TREE -> LEFT)
Step 3:             Write TREE -> DATA
Step 4:             INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.17 Algorithm for in-order traversal



(b)

find the sequence of nodes that will be visited using in-order traversal algorithm.

TRAVERSAL ORDER: B, D, A, E, H, G, I, F, and C

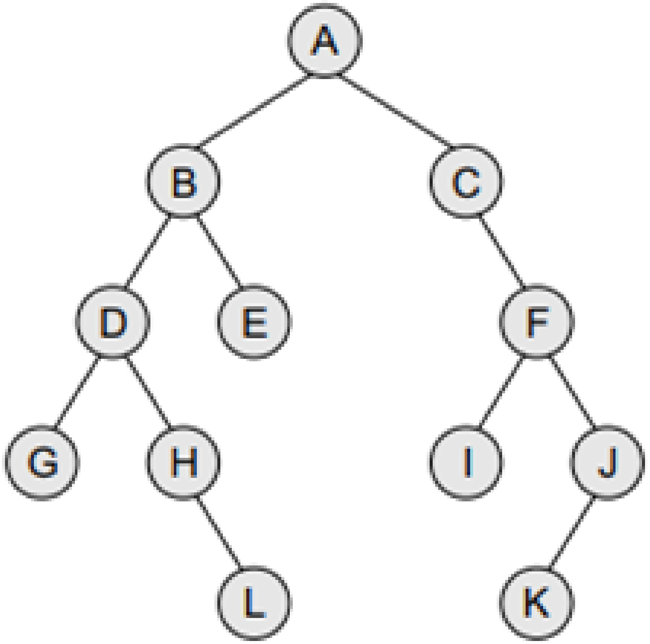
Post-order Traversal in a Binary tree

The algorithm works by:

1. Traversing the left sub-tree,
2. Traversing the right sub-tree,
3. and finally Visiting the root node.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           POSTORDER(TREE -> LEFT)
Step 3:           POSTORDER(TREE -> RIGHT)
Step 4:           Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

Figure 9.18 Algorithm for post-order traversal



For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: G, L, H, D, E, B, I, K, J, F, C, and A

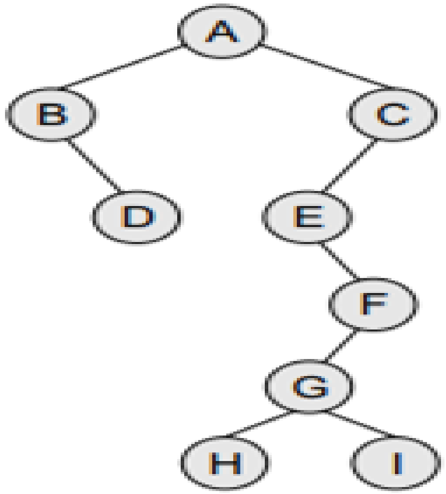
Post-order Traversal in a Binary tree

The algorithm works by:

1. Traversing the left sub-tree,
2. Traversing the right sub-tree,
3. and finally Visiting the root node.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             POSTORDER(TREE -> LEFT)
Step 3:             POSTORDER(TREE -> RIGHT)
Step 4:             Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

Figure 9.18 Algorithm for post-order traversal



(b)

For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: D, B, H, I, G, F, E, C, and A

Pre-order Traversal in a Binary tree

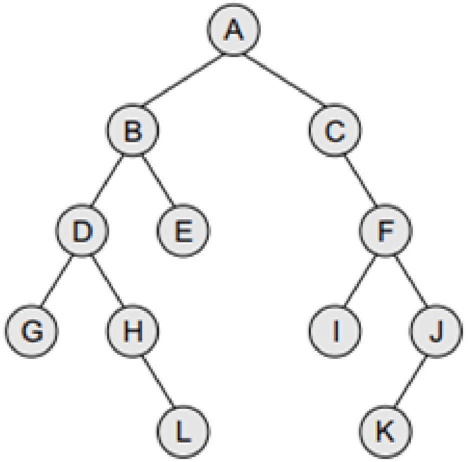
The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree,
3. and finally Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             Write TREE -> DATA
Step 3:             PREORDER(TREE -> LEFT)
Step 4:             PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.16 Algorithm for pre-order traversal

When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a **prefix expression.**



For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: A, B, D, G, H, L, E, C, F, I, J, and K

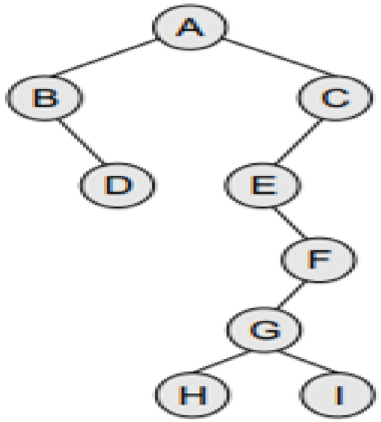
Pre-order Traversal in a Binary tree

The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree,
3. and finally Traversing the right sub-tree.
4. Also called as depth-first traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             Write TREE -> DATA
Step 3:             PREORDER(TREE -> LEFT)
Step 4:             PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.16 Algorithm for pre-order traversal



(b)

For the trees given in Example 9.6, give the sequence of nodes that will be visited using post-order traversal algorithm.

TRAVERSAL ORDER: A, B, D, C, D, E, F, G, H, and I

Traversal in a Binary tree

For the trees given in Example 9.13, give the sequence of nodes that will be visited using post-order, pre-order and in-order traversal algorithm.

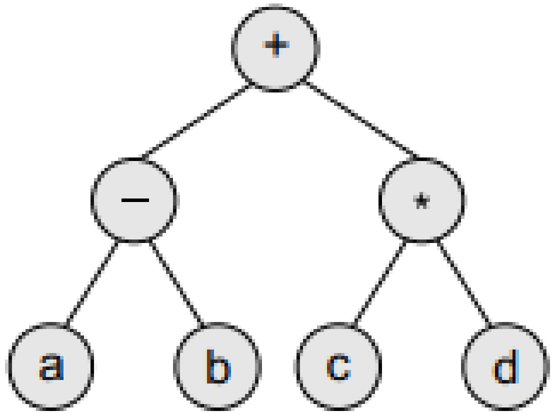


Figure 9.13 Expression tree

Pre-Order

+ – a b * c d

When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a **prefix expression**.

In-order

a – b + c * d

When we traverse the elements of a tree using the in-order traversal algorithm, the expression that we get is a **infix expression**.

Post-order

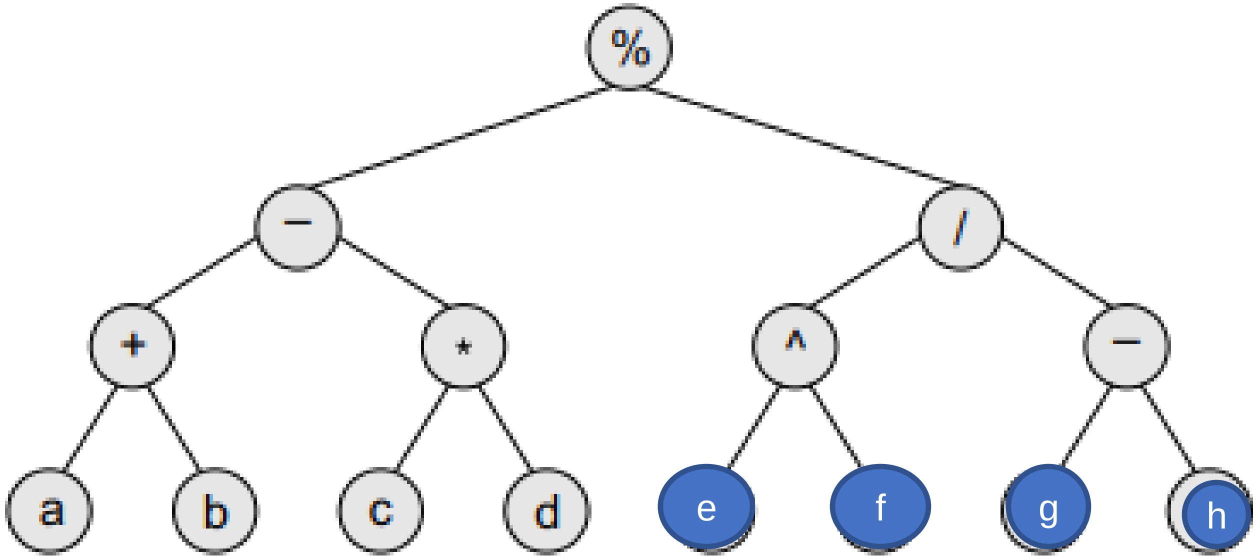
a b – c d * +

Post-order traversals are used to extract **postfix notation from an expression tree.**

Example : Given an expression,

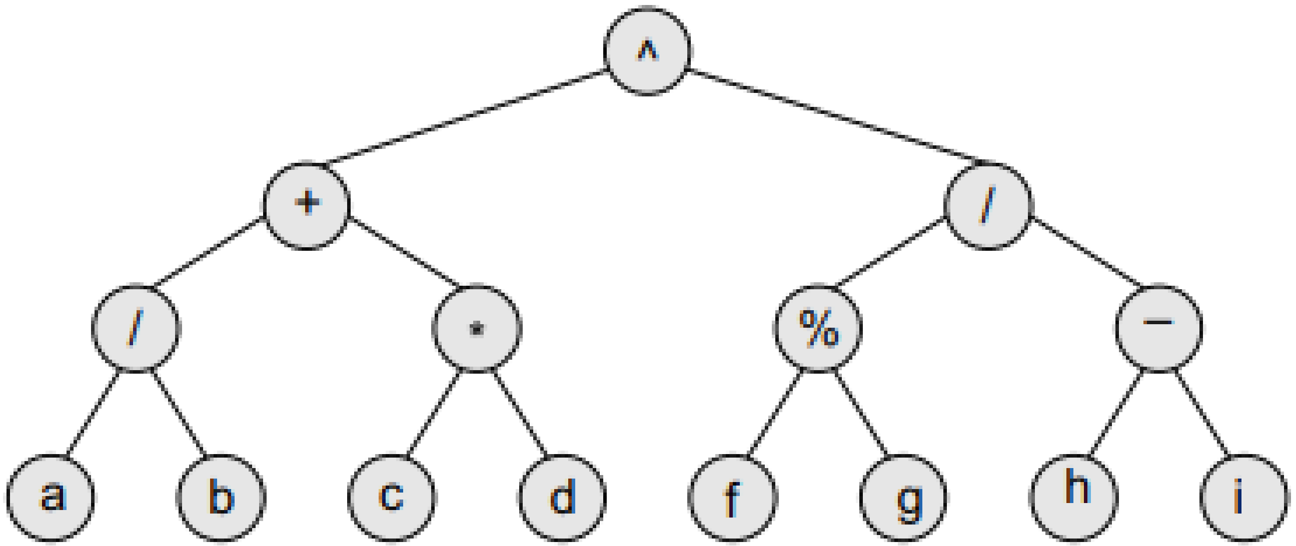
Exp = ((a + b) – (c * d)) % ((e ^f) / (g – h)),

construct the corresponding binary tree.



Expression tree

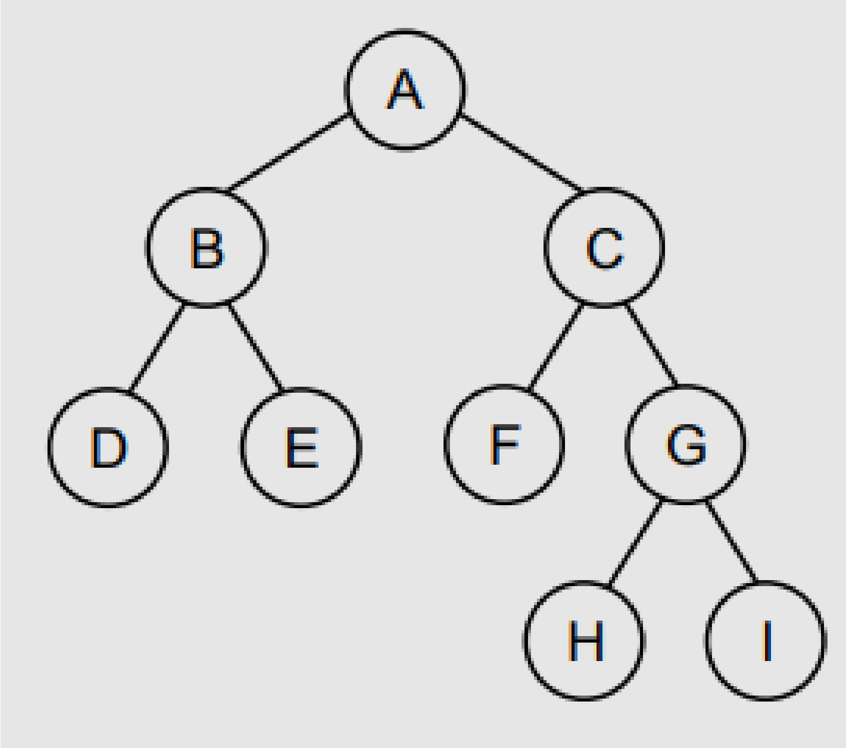
Example : Given the binary tree, write down the expression that it represents.



[{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

Example : Consider the tree given below. Now, do the following:

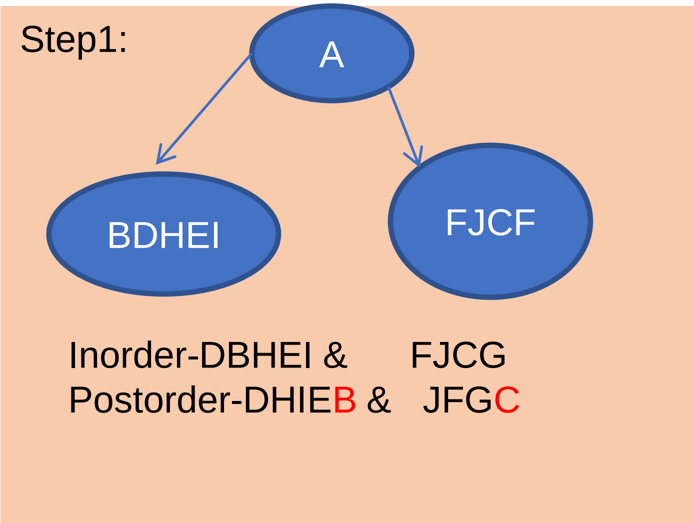
(a) Name the leaf nodes
(b) Name the non-leaf nodes
(c) Name the ancestors of E
(d) Name the descendants of A
(e) Name the siblings of C
(f) Find the height of the tree
(g) Find the height of sub-tree rooted at E
(h) Find the level of node E
(i) Find the in-order, pre-order, post-order traversal]

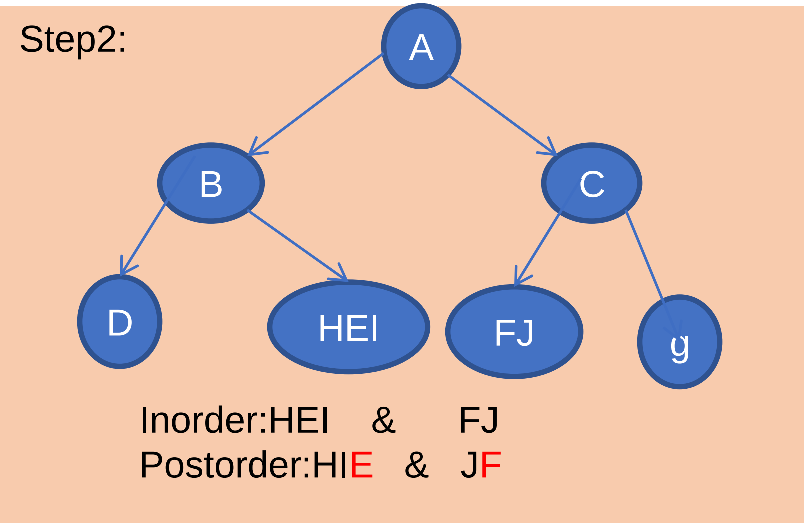


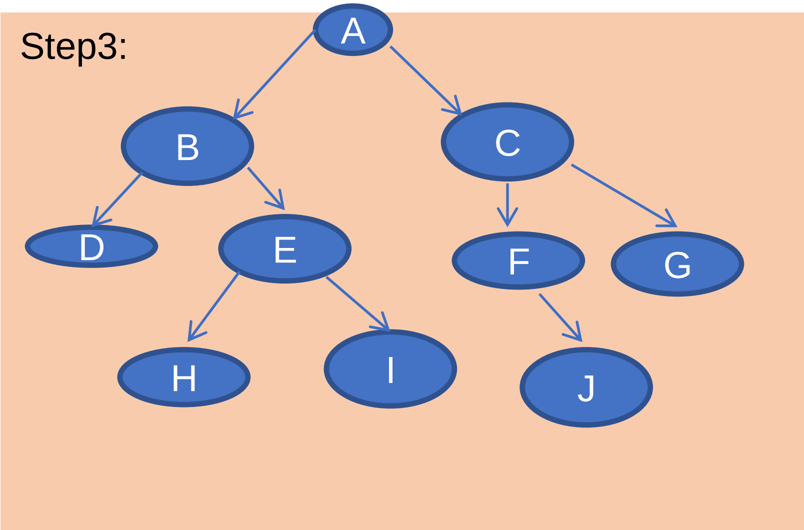
Construct a binary tree for:

In-order : DBHEIAFJCG

Post-order: DHIEBJFGCA



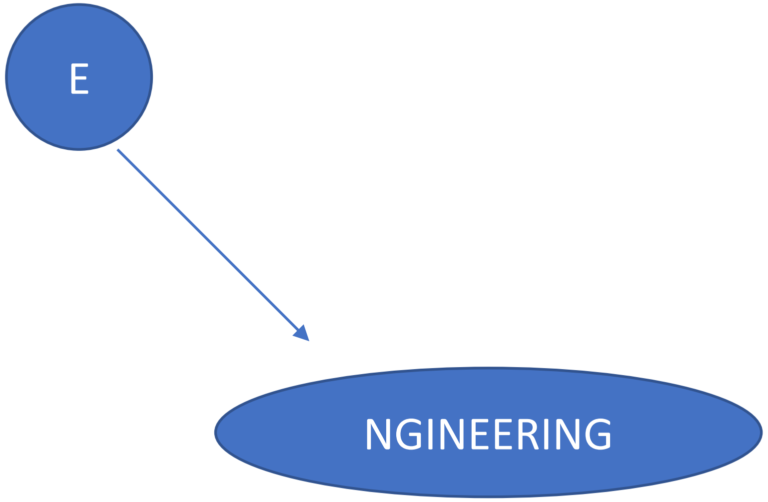




Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

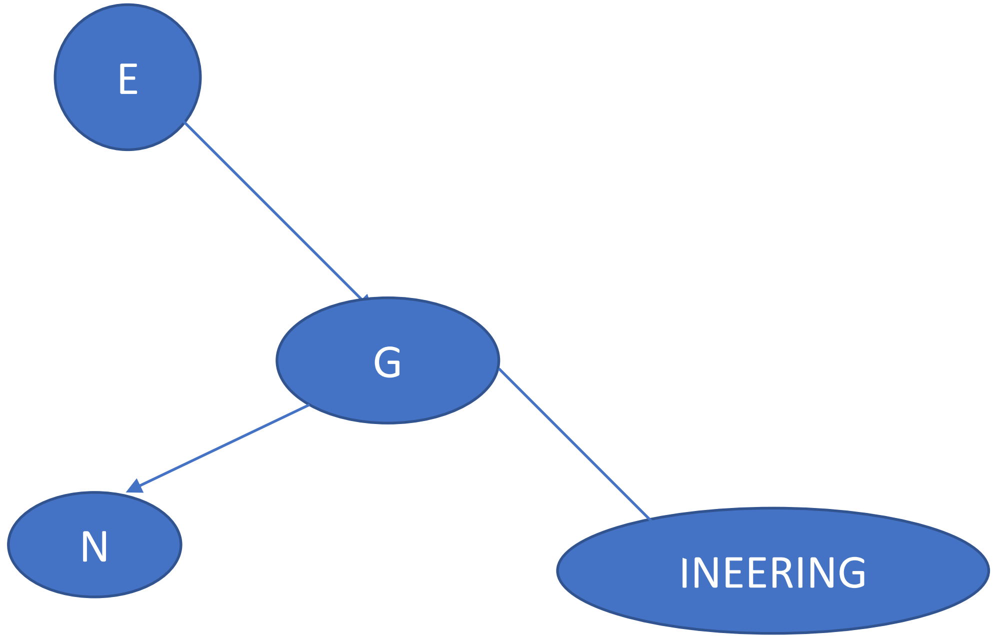
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

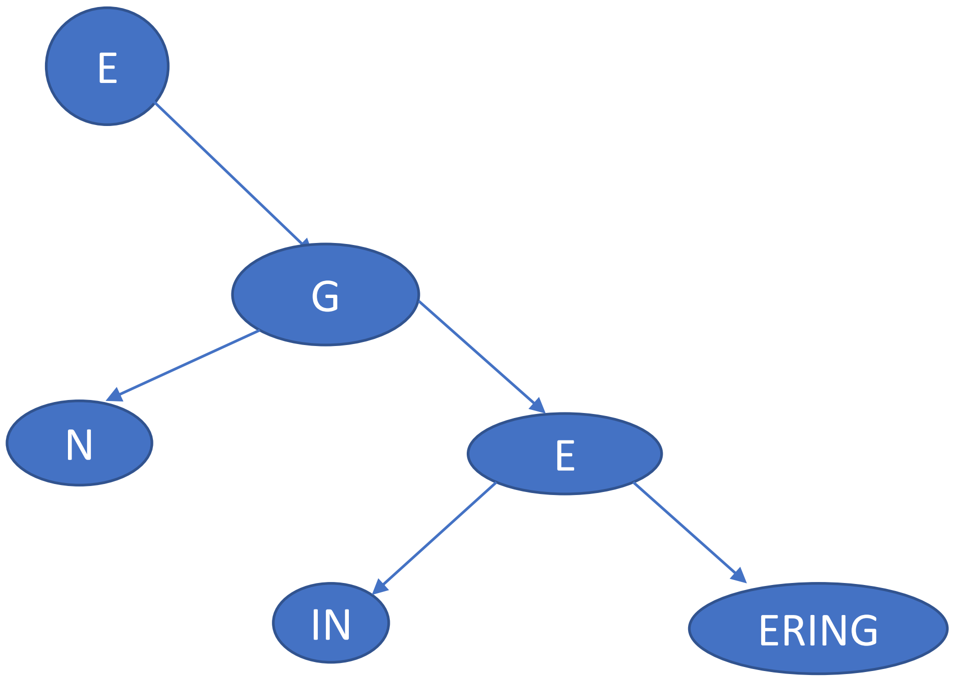
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

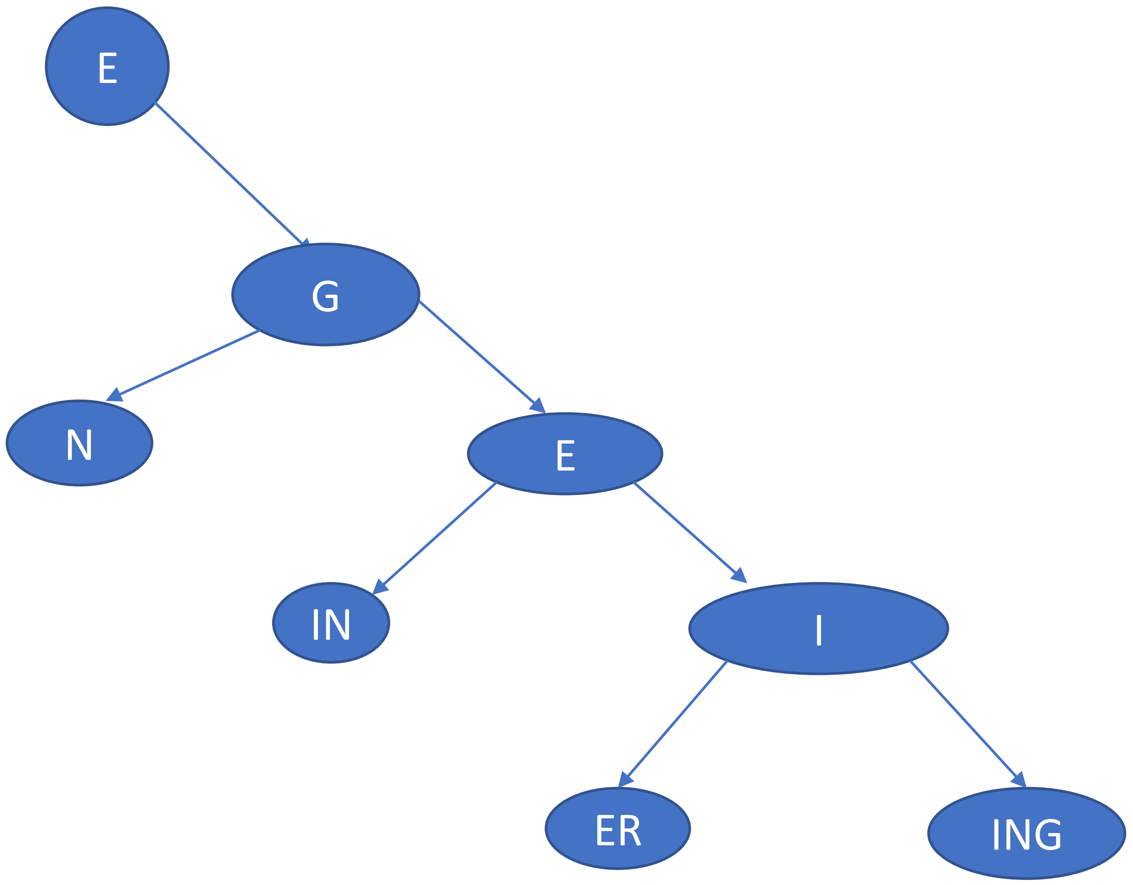
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

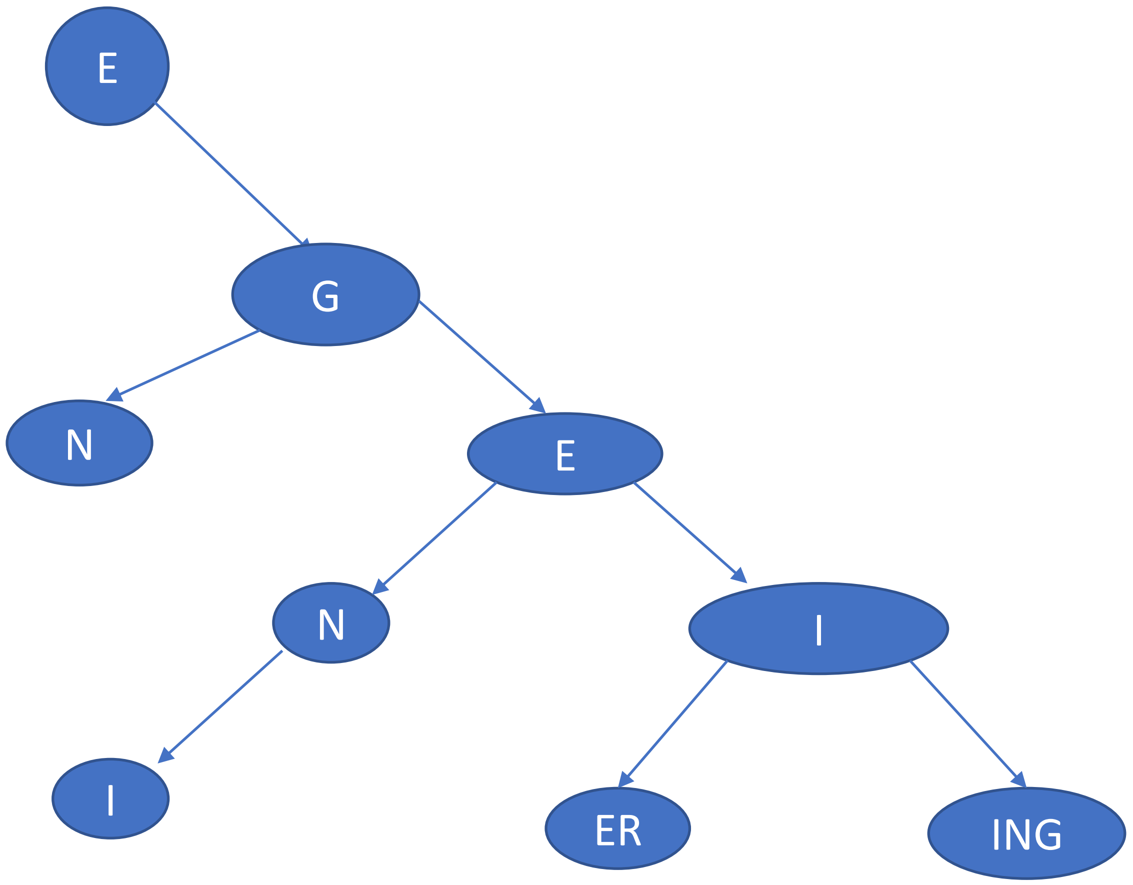
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

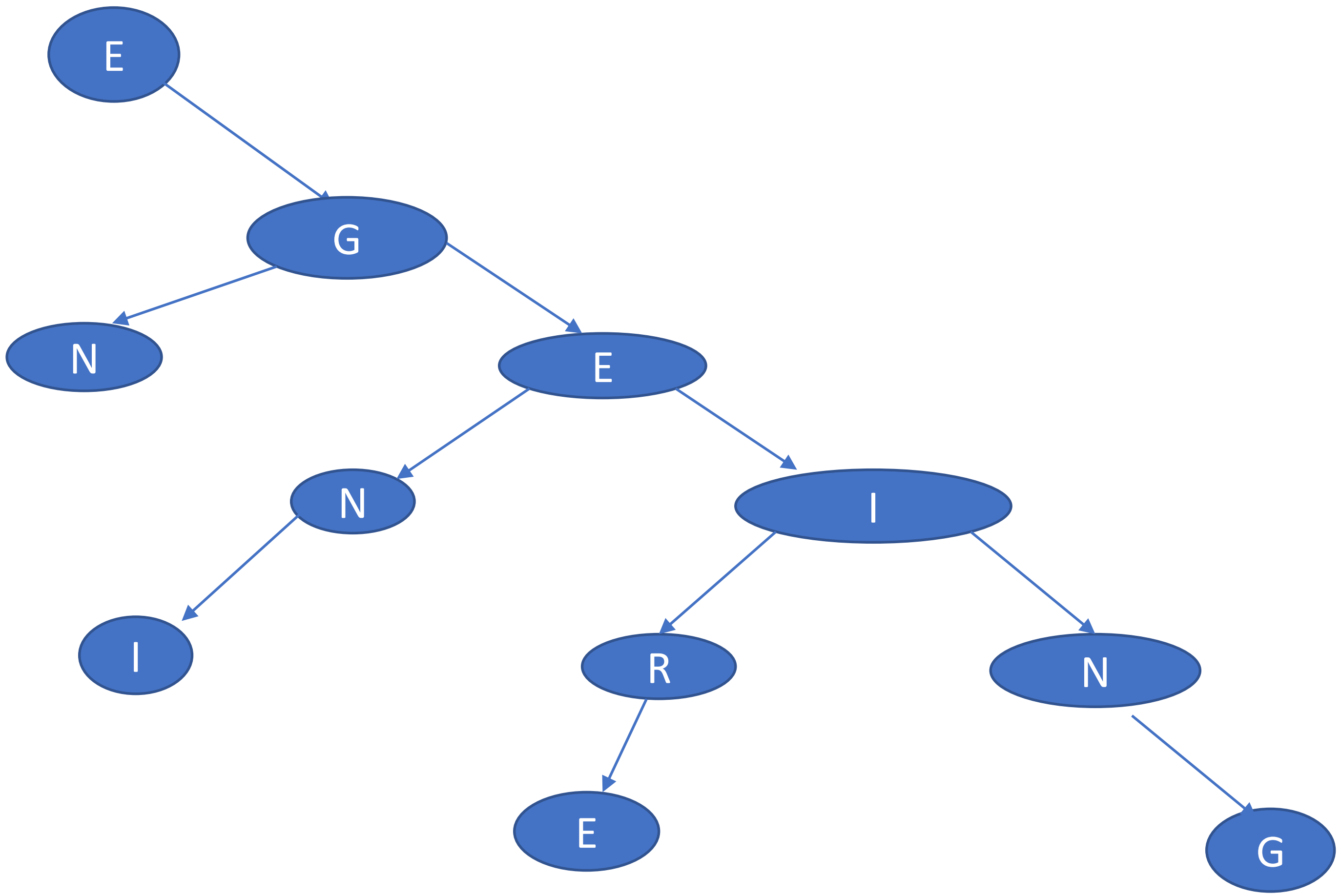
Pre-Order : EGNENIIRENG



Construct a binary tree for the in-order and pre-order traversal sequence given below

In-order : ENGINEERING

Pre-Order : EGNENIIRENG



Binary search tree

- A binary search tree, also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in an order.

a binary search tree is a binary tree with the following properties:

- The left sub-tree of a node N contains values that are less than N’s value.
- The right sub-tree of a node N contains values that are greater than N’s value.
- Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

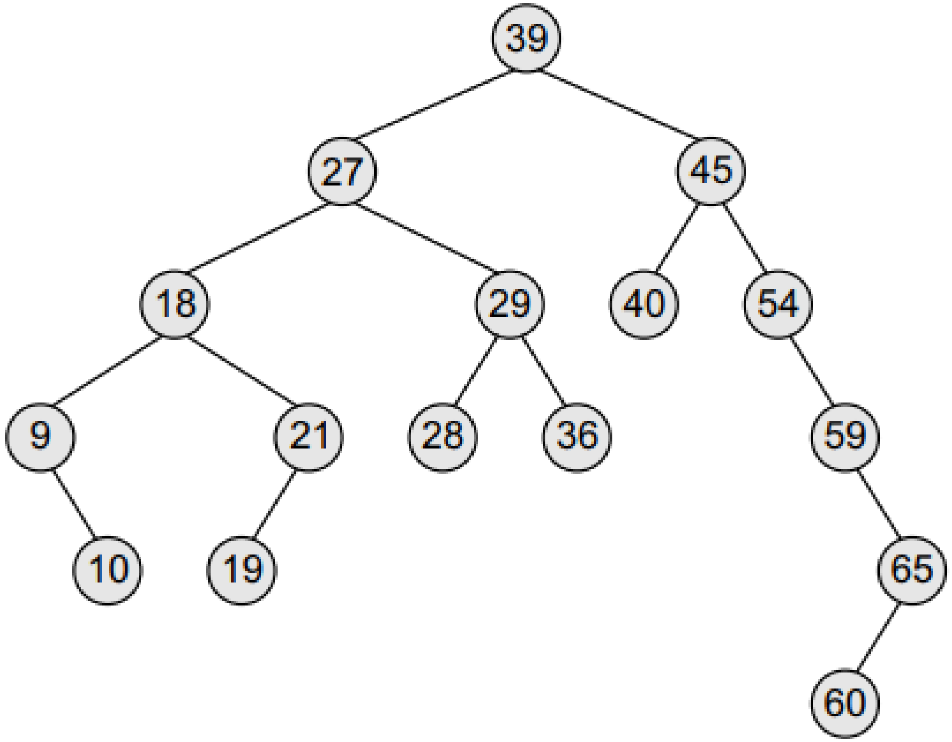


Figure 10.1 Binary search tree

Binary search tree

- A binary search tree, also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in an order.

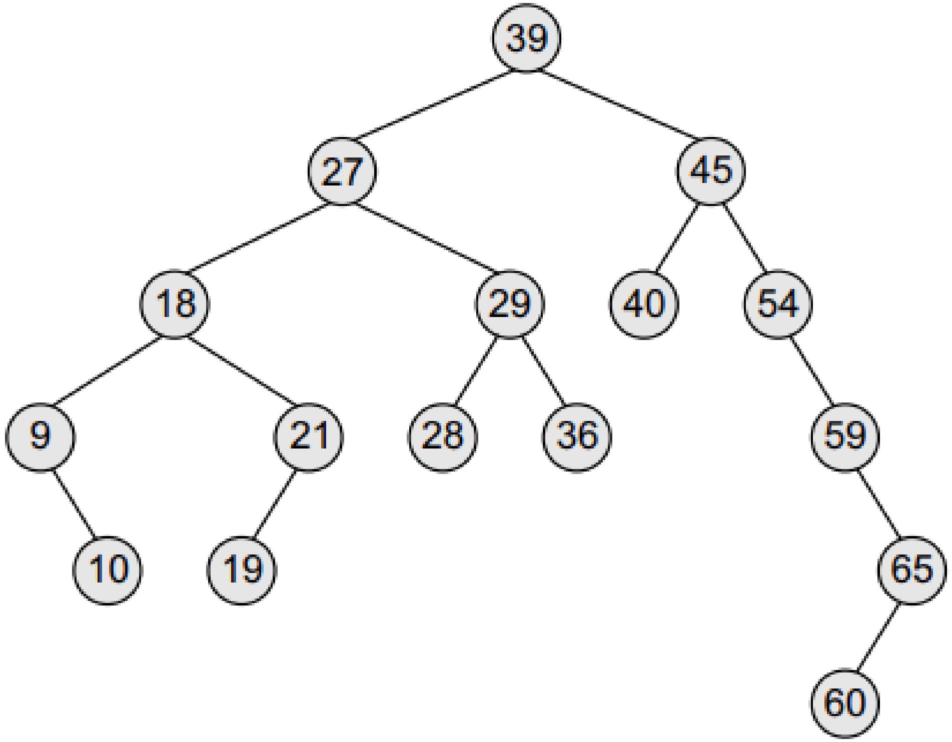


Figure 10.1 Binary search tree

Advantage:

- time needed to search an element in the tree is greatly reduced.
- speeds up the insertion and deletion operations
- Efficient data structures than arrays and linked list

Binary search tree

State whether the binary trees in Fig. 10.3 are binary search trees or not.

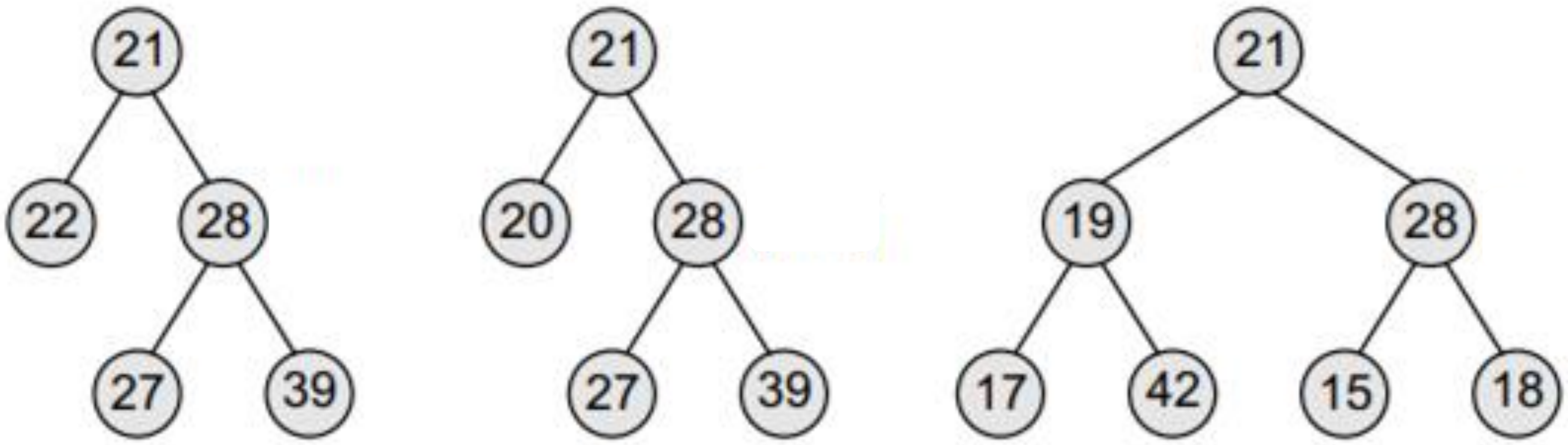


Figure 10.3 Binary trees

Binary search tree

State whether the binary trees in Fig. 10.3 are binary search trees or not.

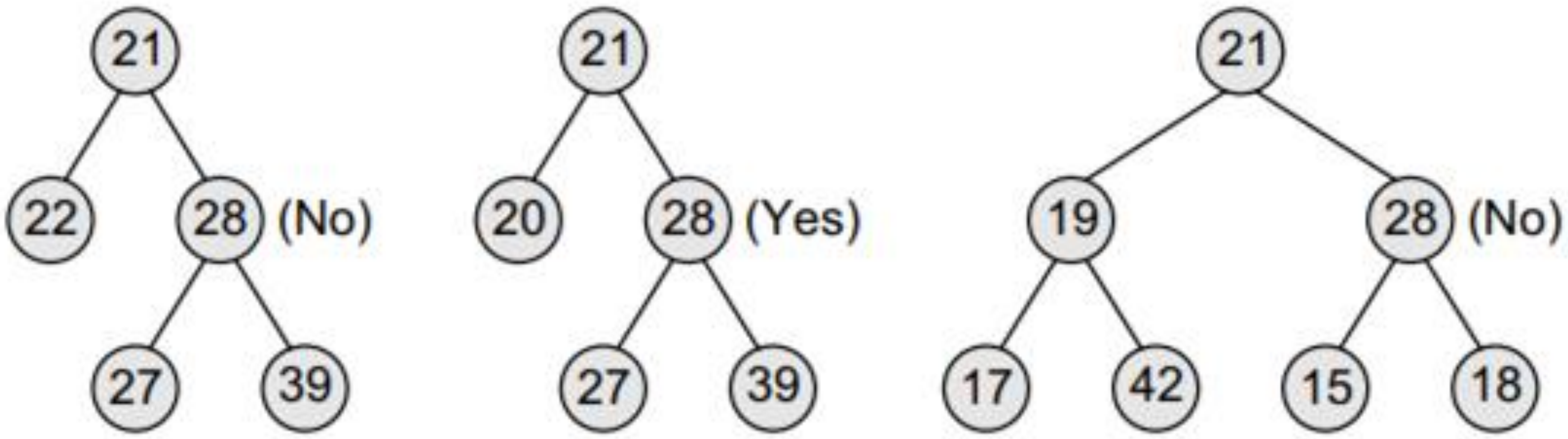
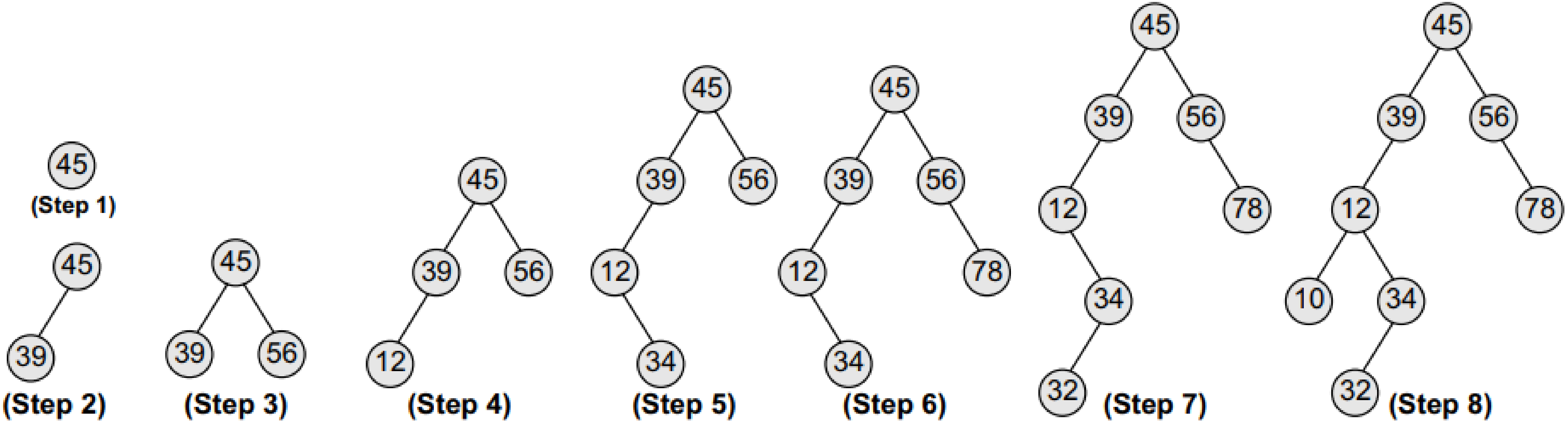


Figure 10.3 Binary trees

Binary search tree

Create a binary search tree using the following data elements:
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



Binary search tree

Create a binary search tree using the following data elements:

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

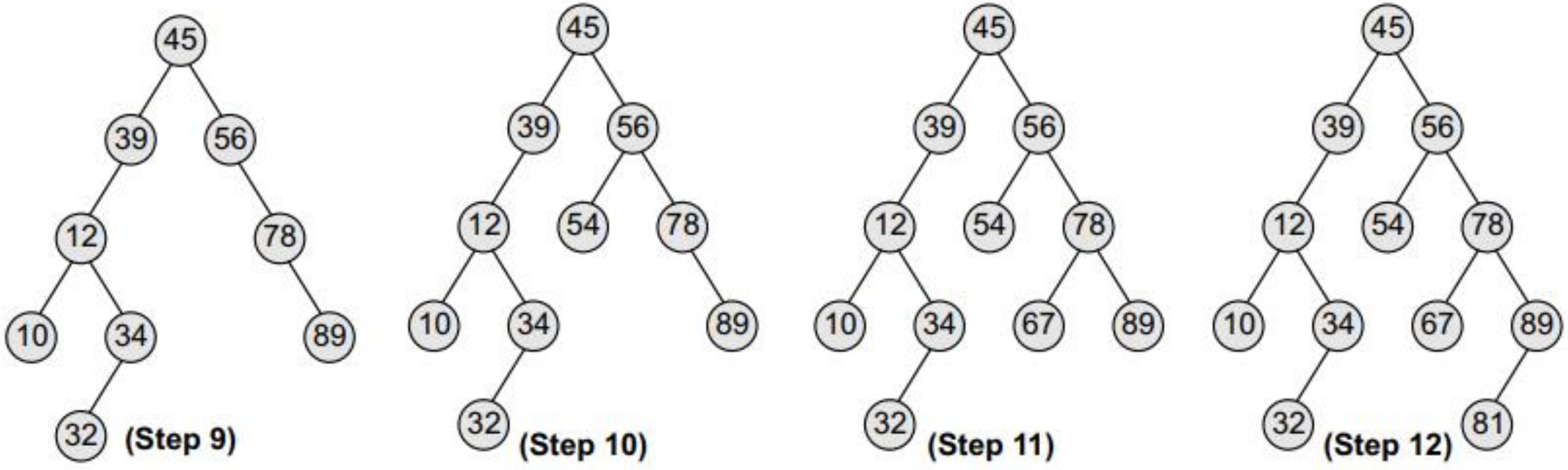


Figure 10.4 Binary search tree

Binary search tree

Create a binary search tree using the following data elements:

J,R,D,G,T,E,M,H,P,A,F,Q

Binary search tree

Create a binary search tree using the following data elements:
50, 33, 44, 22, 77, 35, 60, 40

OPERATIONS ON BINARY SEARCH TREES

- Traversal
- Searching
- Insertions
- deletions

Searching operation on binary search trees

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
           Return searchElement(TREE -> LEFT, VAL)
         ELSE
           Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

Figure 10.8 Algorithm to search for a given value in a binary search tree

insertion operation on binary search trees

add a new node with a given value at the correct position in the binary search tree

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

Figure 10.9 Algorithm to insert a given value in a binary search tree

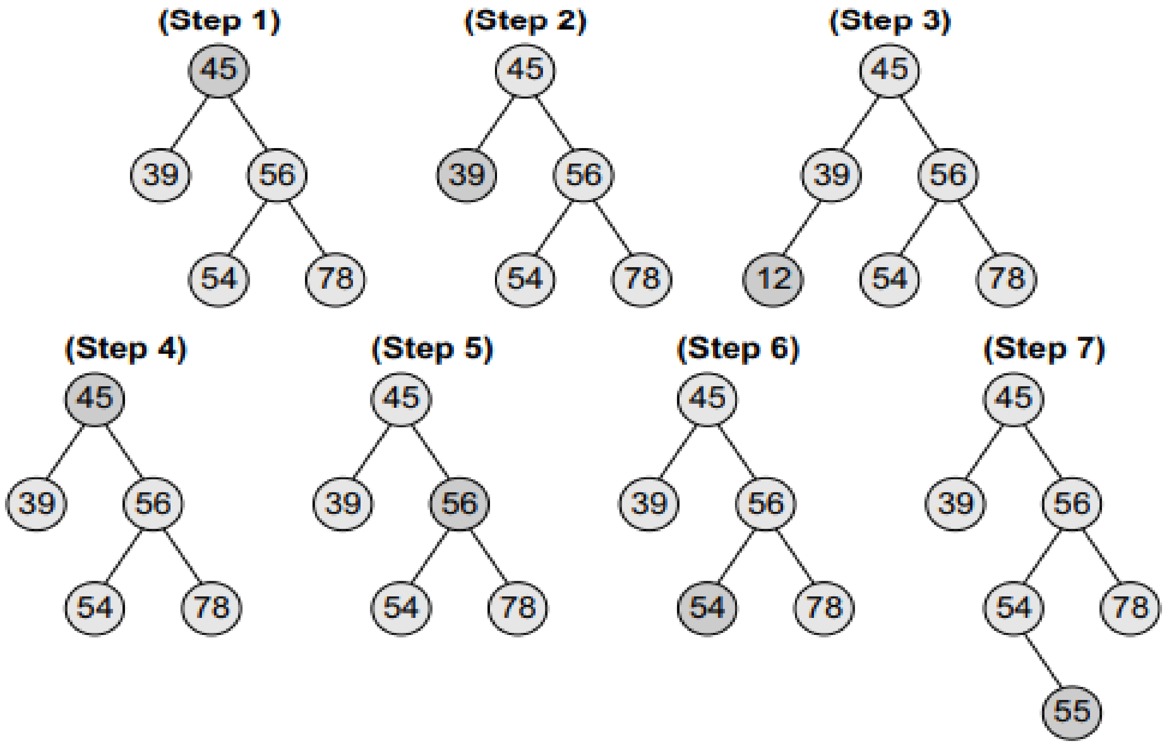


Figure 10.10 Inserting nodes with values 12 and 55 in the given binary search tree

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Three Cases of deletion

1. Deleting a leaf node
2. Delete node with one child
3. Delete node with two children

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
               SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Three Cases of deletion

1. Deleting a leaf node

- Find a node to be deleted using search operation
- Delete the node using free function

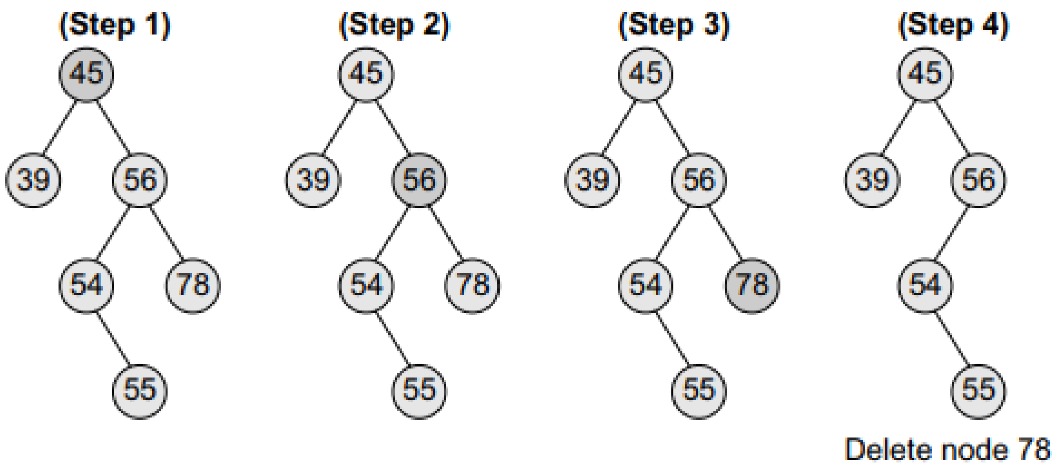


Figure 10.11 Deleting node 78 from the given binary search tree

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
              SET TREE = TREE -> LEFT
          ELSE
              SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Three Cases of deletion

2. **Delete node with one child**

- Find a node to be deleted using search operation
- If it has one child, then create link between its parent and child nodes
- Delete the node using free function

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
               SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Figure 10.15 Algorithm to delete a node from a binary search tree

Delete node with one child

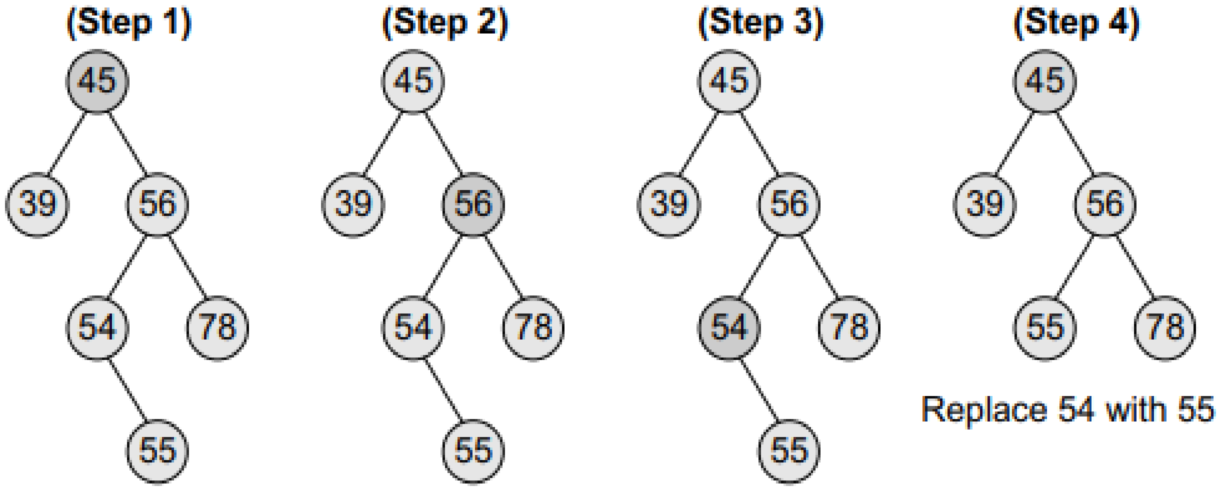


Figure 10.12 Deleting node 54 from the given binary search tree

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
              SET TREE = TREE -> LEFT
          ELSE
              SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Figure 10.15 Algorithm to delete a node from a binary search tree

i. Three Cases of deletion

3. **Delete node with two children**

i. Find a node to be deleted using search operation
ii. If it has two children, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree)
iii. Swap both deleting node and node which is found in above step
iv. Check whether deleting node came to case 1 or 2 else goto step 2
v. If it comes to case 1, delete the node using case 1
vi. If it comes to case 2, delete using case 2 logic
vii. Repeat the same process until node is deleted from the tree

Deletion operation on binary search trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
         IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Figure 10.15 Algorithm to delete a node from a binary search tree

If the node to be deleted has both left and right children, then we find the in-order predecessor of the node by calling **findLargestNode(TREE -> LEFT)** and replace the current node's value with that of its in-order predecessor. Then, we call Delete(TREE -> LEFT, TEMP -> DATA) to delete the initial node of the in-order predecessor. Thus, we reduce the case 3 of deletion into either case 1 or case 2 of deletion.

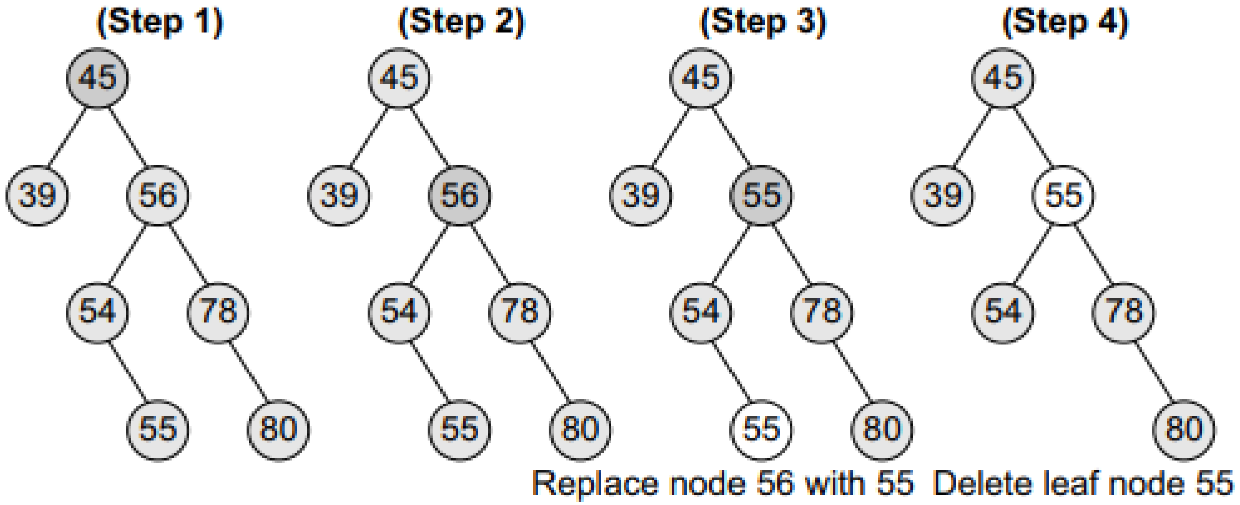


Figure 10.13 Deleting node 56 from the given binary search tree

Threaded Binary Tree:

A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.

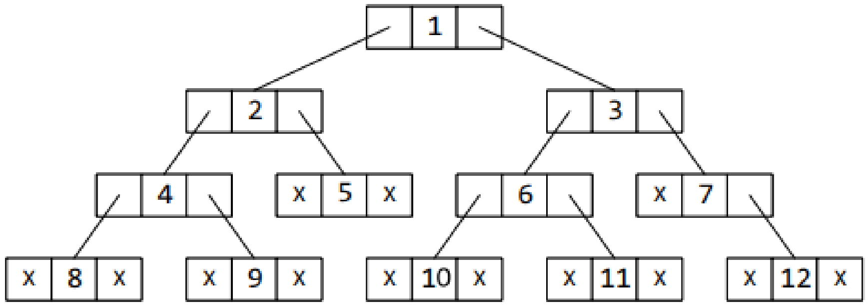


Figure 10.29 (b) Linked representation of the binary tree (without threading)

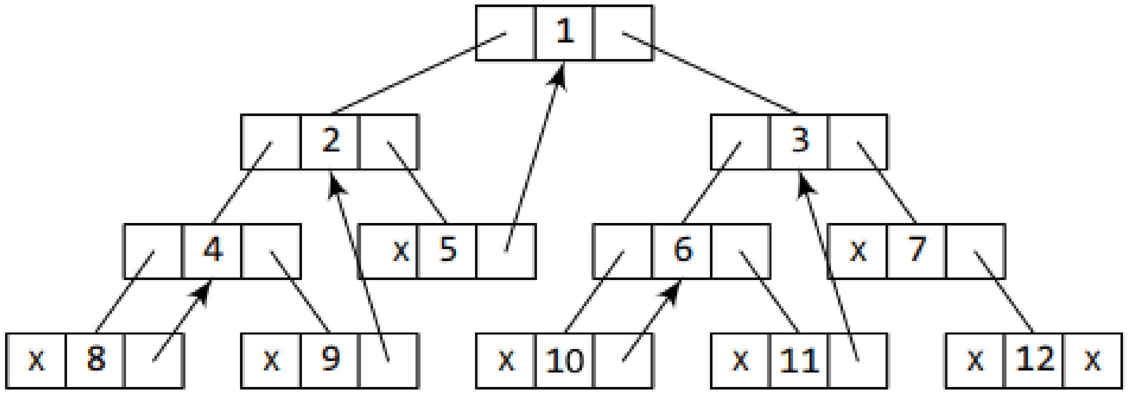


Figure 10.30 (a) Linked representation of the binary tree with one-way threading

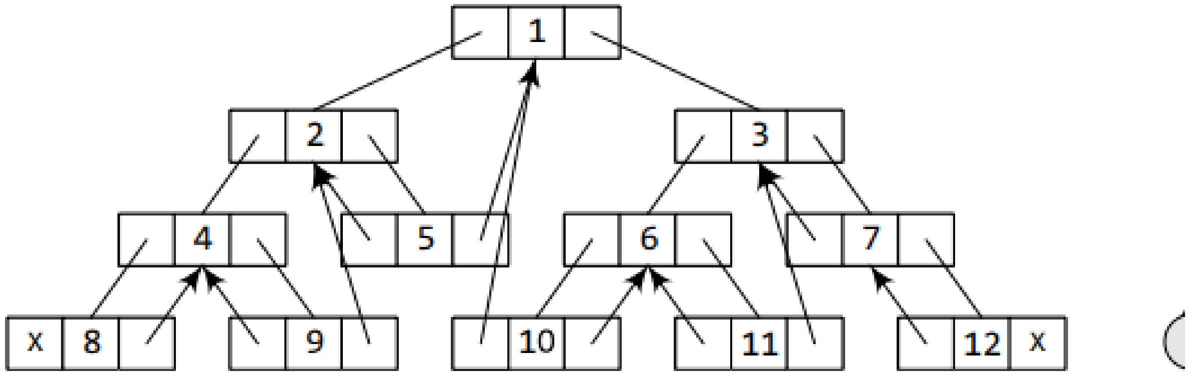


Figure 10.31 (a) Linked representation of the binary tree with threading,

Threaded Binary Tree:

- In the linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both
- space wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information
- the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node
- special pointers are called threads and binary trees containing threads are called threaded trees
- Allows fast traversal
- Types – Single threaded, Double threaded

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack.

The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion.

A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

Single Threaded Binary Tree:

- Each node is threaded towards **either** in-order predecessor or successor

Double Threaded Binary Tree:

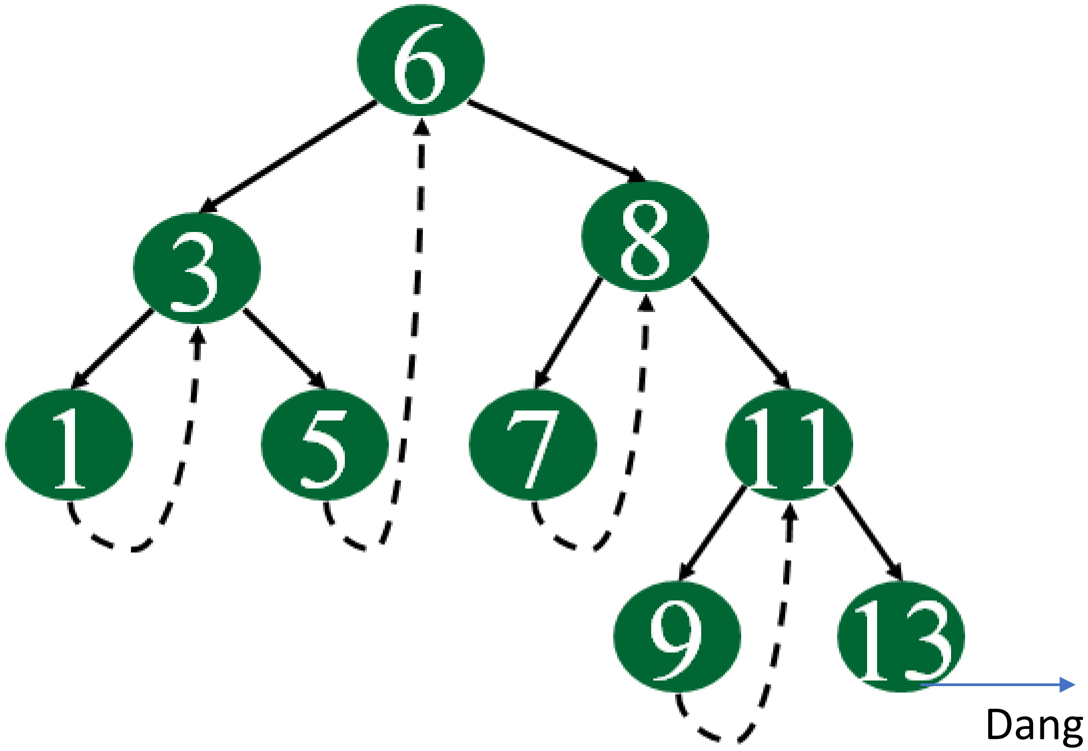
- Each node is threaded **both** in-order predecessor and successor (left-right)

Single Threaded Binary Tree:

- Each node is threaded towards **either** in-order predecessor or successor

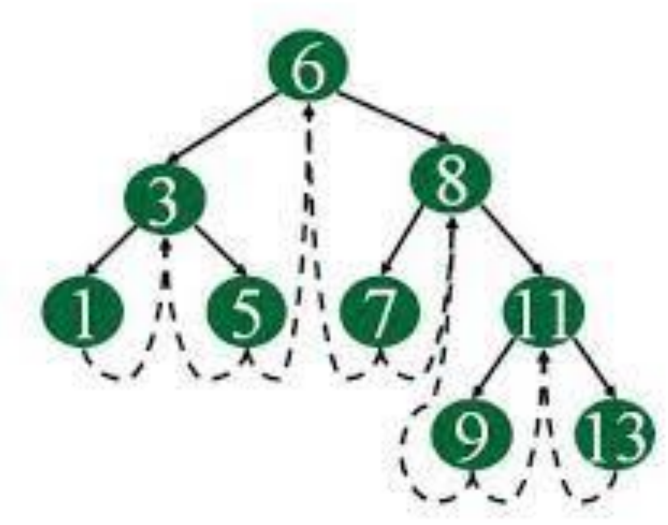
Double Threaded Binary Tree:

- Each node is threaded **both** in-order predecessor and successor (left-right)



Dangling or Null

Single threaded in-order predecessor (right in-threaded binary tree)



Double Threaded Binary Tree

AVL TREES:

- Self balanced binary search tree
- Difference between the heights of left and right subtrees of every node in the tree is in the range of -1, 0 or +1
- Means height of children differ by atmost 1
- Every node maintains extra node called balance factor

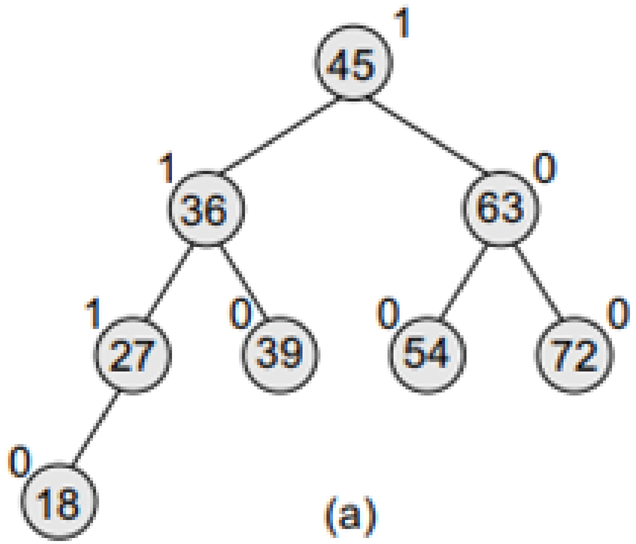
AVL TREES:

Balance factor = Height (left sub-tree) – Height (right sub-tree)

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a left-heavy tree.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a right-heavy tree.

AVL TREES:

Balance factor = Height (left sub-tree) – Height (right sub-tree)



(a)

The nodes 18, 39, 54, and 72 have no children, so their balance factor = 0.

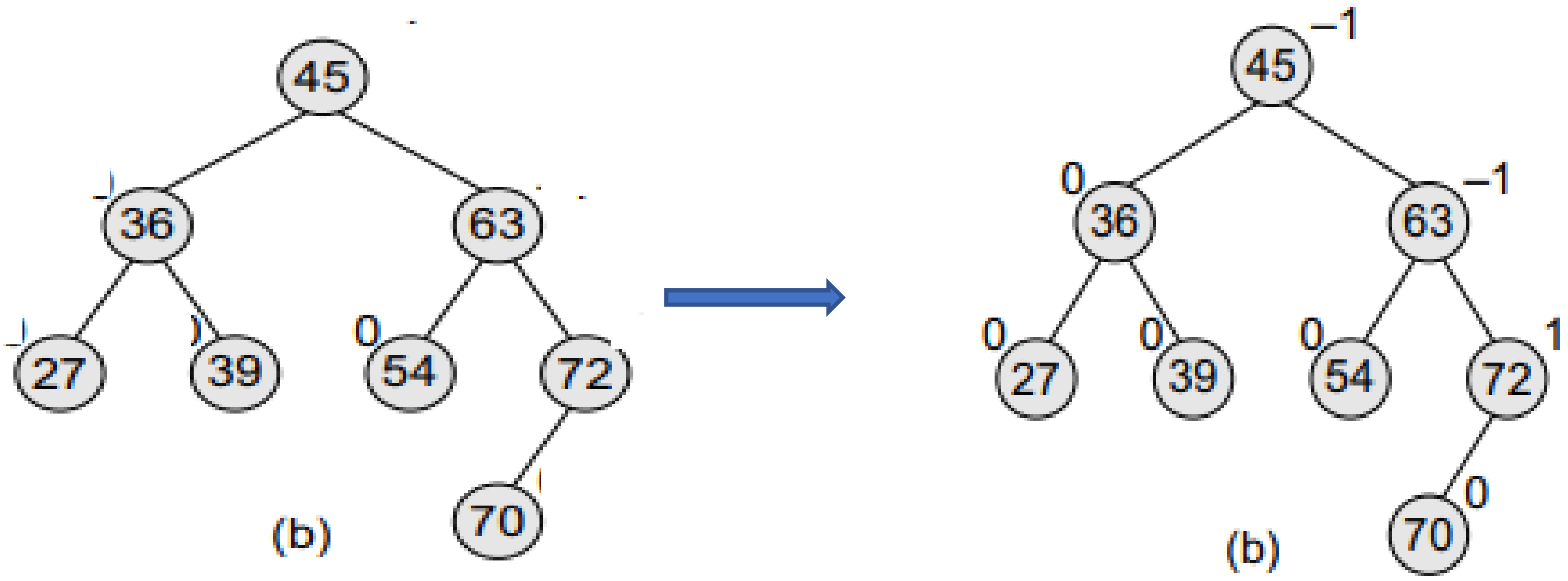
Node 27 has one left child and zero right child. So, the height of left sub-tree = 1, whereas the height of right sub-tree = 0. Thus, its balance factor = 1.

Look at node 36, it has a left sub-tree with height = 2, whereas the height of right sub-tree = 1. Thus, its balance factor = 2 – 1 = 1.

Similarly, the balance factor of node 45 = 3 – 2 =1; and node 63 has a balance factor of 0 (1 – 1)

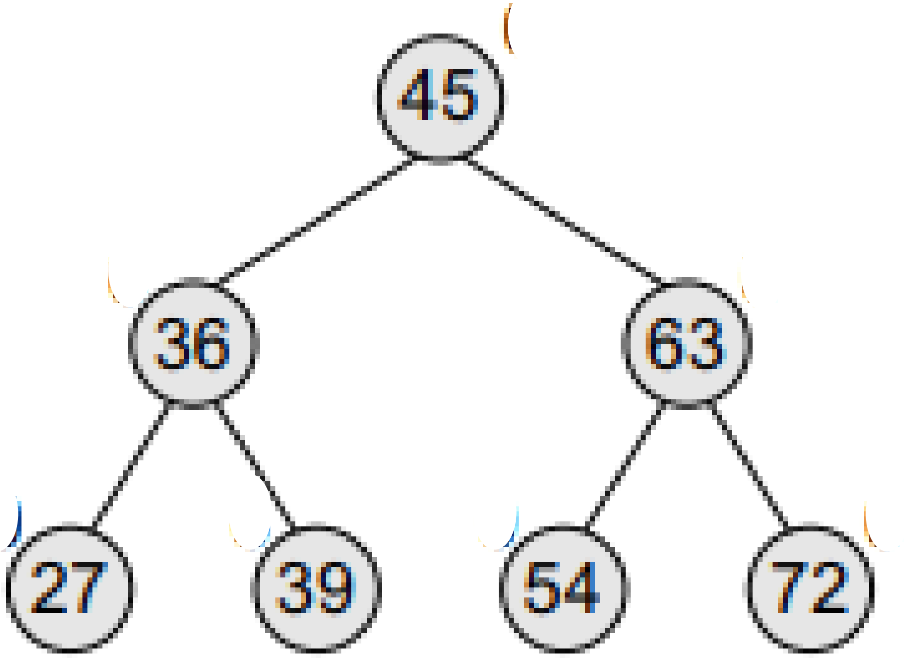
AVL TREES:

Balance factor = Height (left sub-tree) – Height (right sub-tree)



AVL TREES:

Balance factor = Height (left sub-tree) – Height (right sub-tree)



(c)

AVL TREES:

Balance factor = Height (left sub-tree) – Height (right sub-tree)

insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, **rebalancing of the tree** may have to be done.

by performing rotation at the critical node.

AVL TREES:

There are four types of rotations:
LL rotation,
RR rotation,
LR rotation, and
RL rotation.

AVL TREES:

During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node.

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

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

AVL TREES:

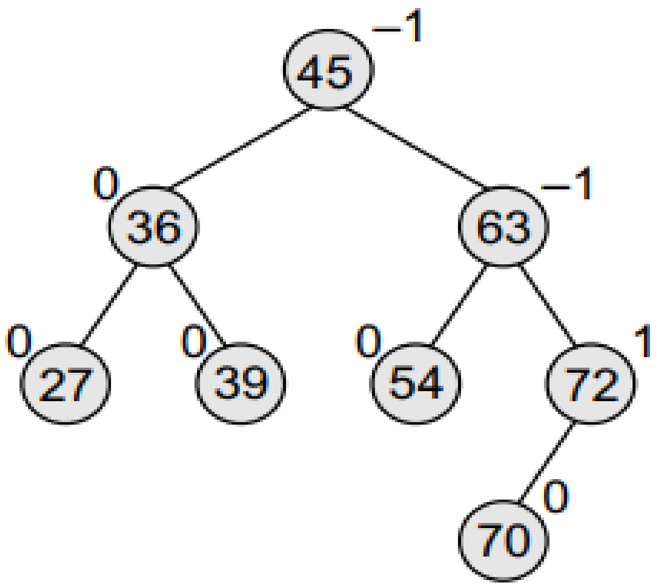


Figure 10.36 AVL tree

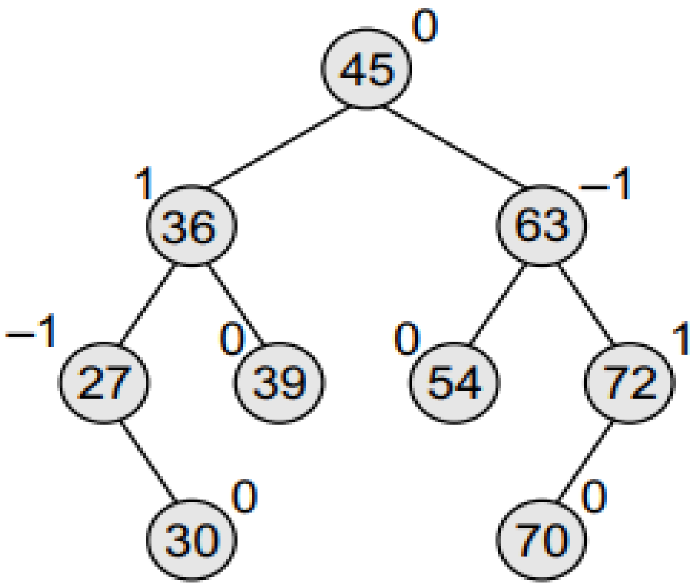


Figure 10.37 AVL tree after inserting a node with the value 30

The tree is still balanced after adding node 30, so no rotation is required

AVL TREES:

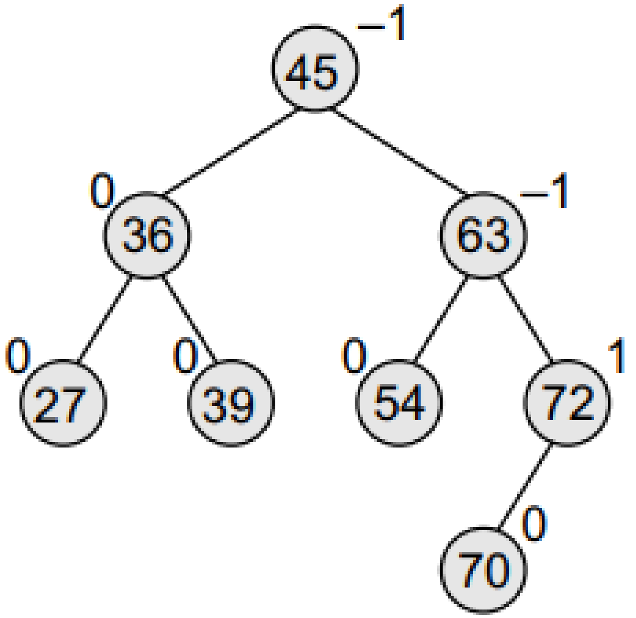
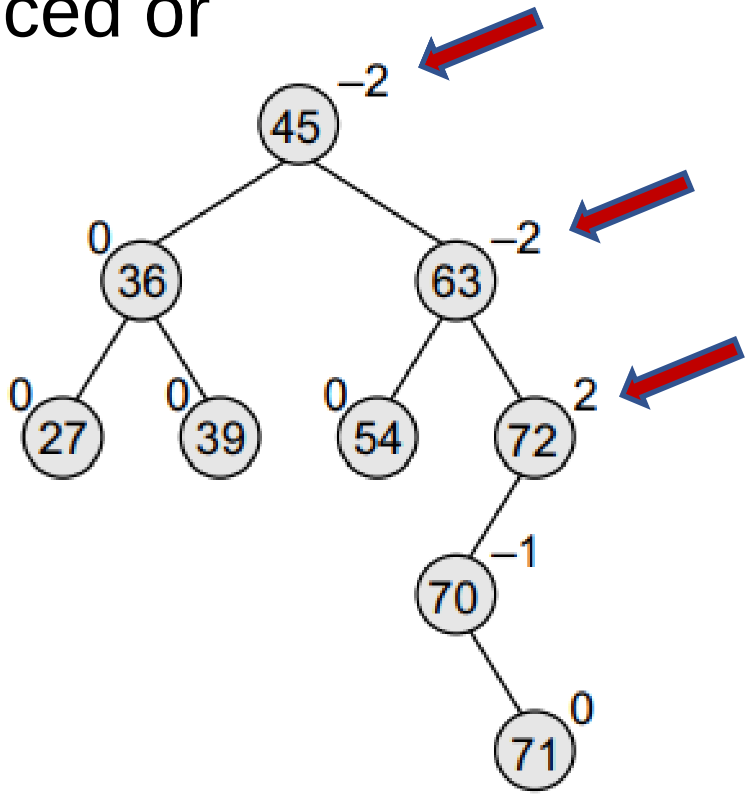


Figure 10.38 AVL tree

Add Node 71 to this node and check if the tree is balanced or not?

Ans:

Unbalanced tree



AVL TREES:

Refer PDF for problem on AVL trees

AVL TREES:

Refer PDF for problem on AVL trees

References: Data Structures Using C by Reema Thareja