Tree

Traversing a Binary Tree

- Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.

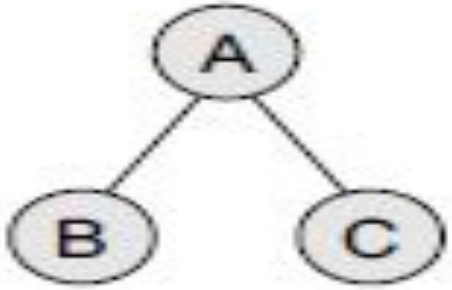
- Unlike linear data structures in which the elements are traversed sequentially, tree is a non linear data structure in which the elements can be traversed in many different ways.

- There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited.

Pre-order Traversal

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

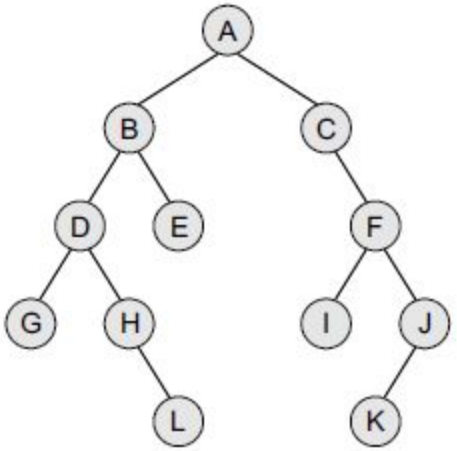
1. Visiting the root node,
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.



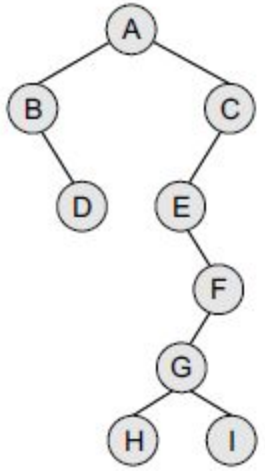
The pre-order traversal of the tree is given as A, B, C.

Algorithm for Pre-Order

```
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
```



(a)



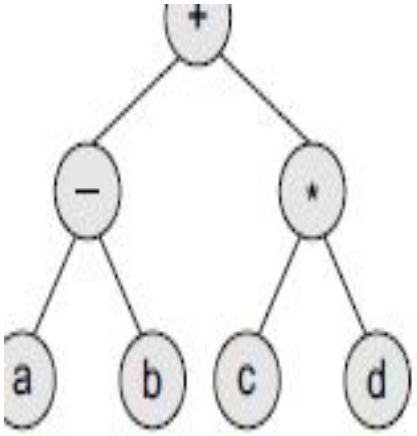
(b)

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

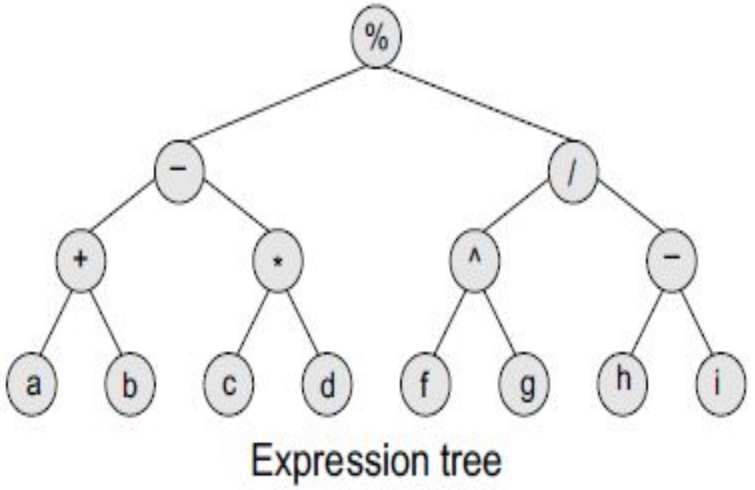
TRAVERSAL ORDER: A, B, D, C, D, E, F, G, H, and I

Pre-order traversal algorithms are **used to extract a prefix notation** from an expression tree.

For example, consider the expressions given below. When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression.



+ – a b * c d



Expression tree

% – + a b * c d / ^ f g – h i

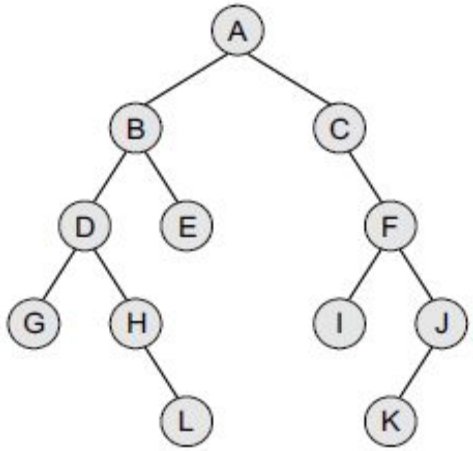
In-order Traversal

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

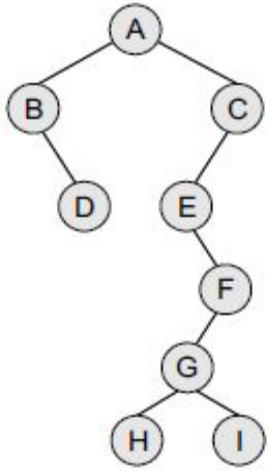
1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

Algorithm for In-Order

```
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
```



(a)



(b)

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

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

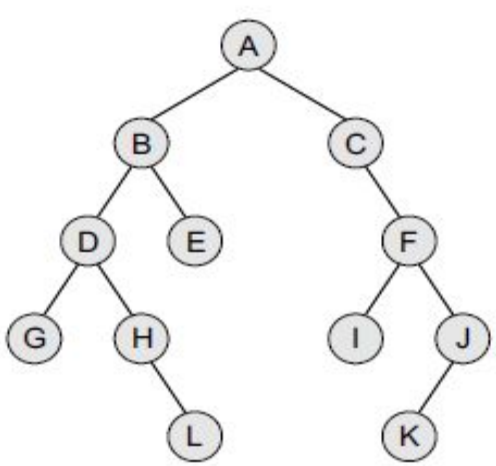
Post-Order Traversal

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

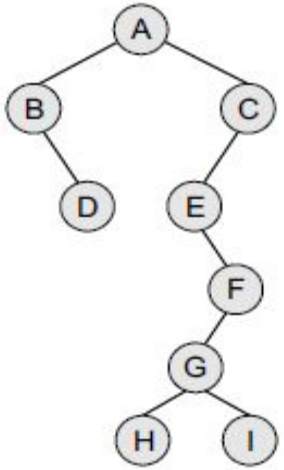
1. Traversing the left sub-tree,
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

Algorithm for Post-Order

```
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
```



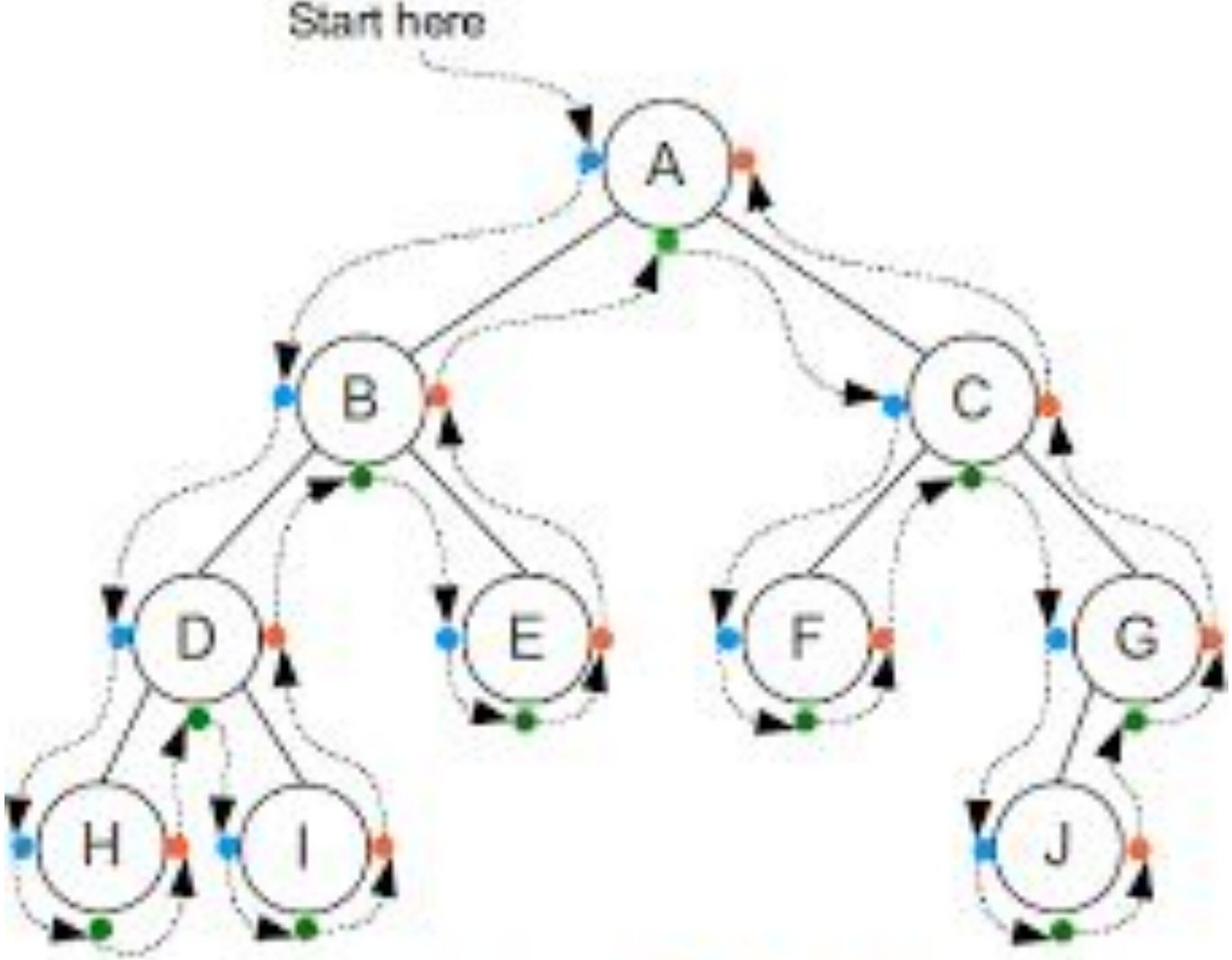
(a)



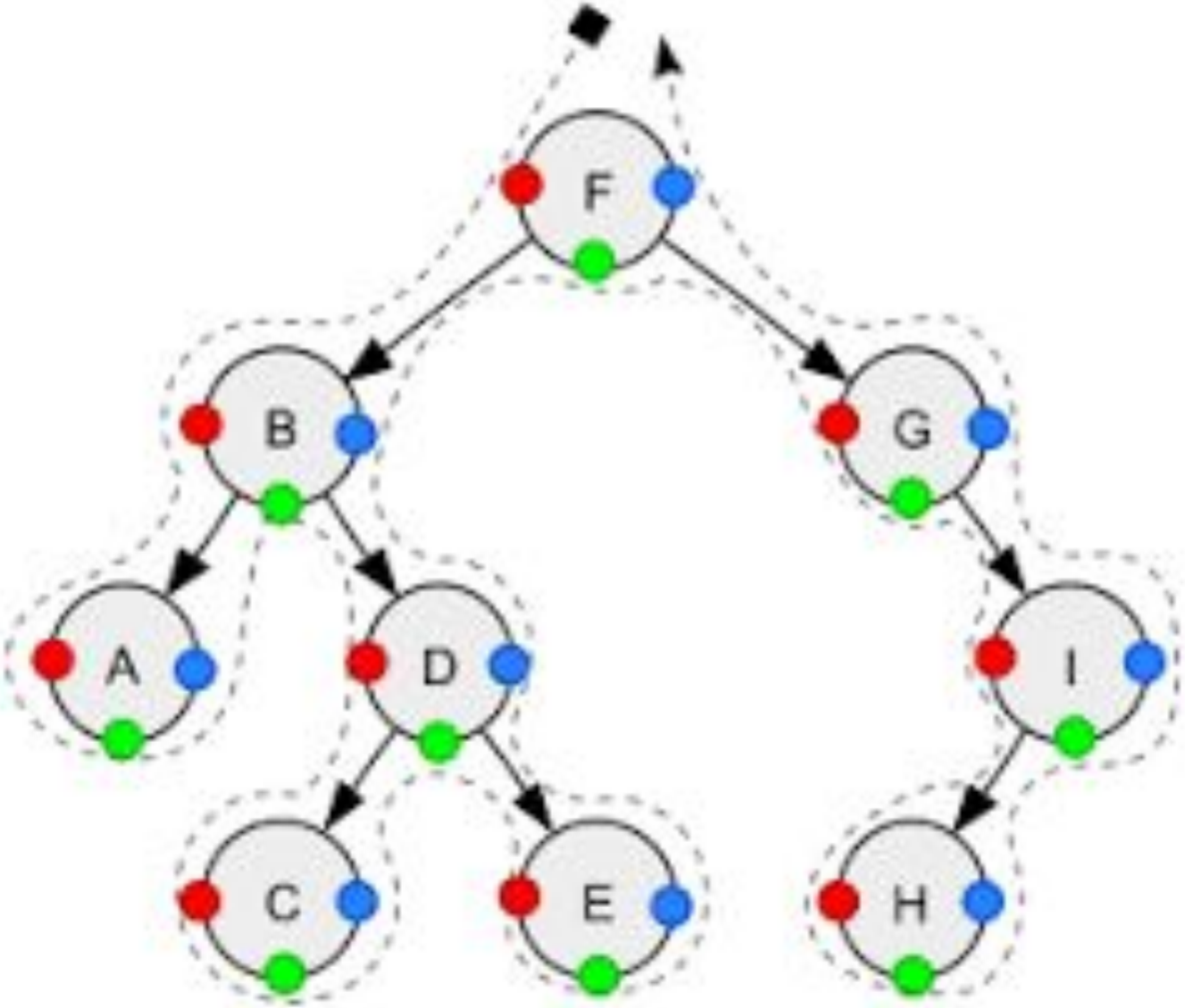
(b)

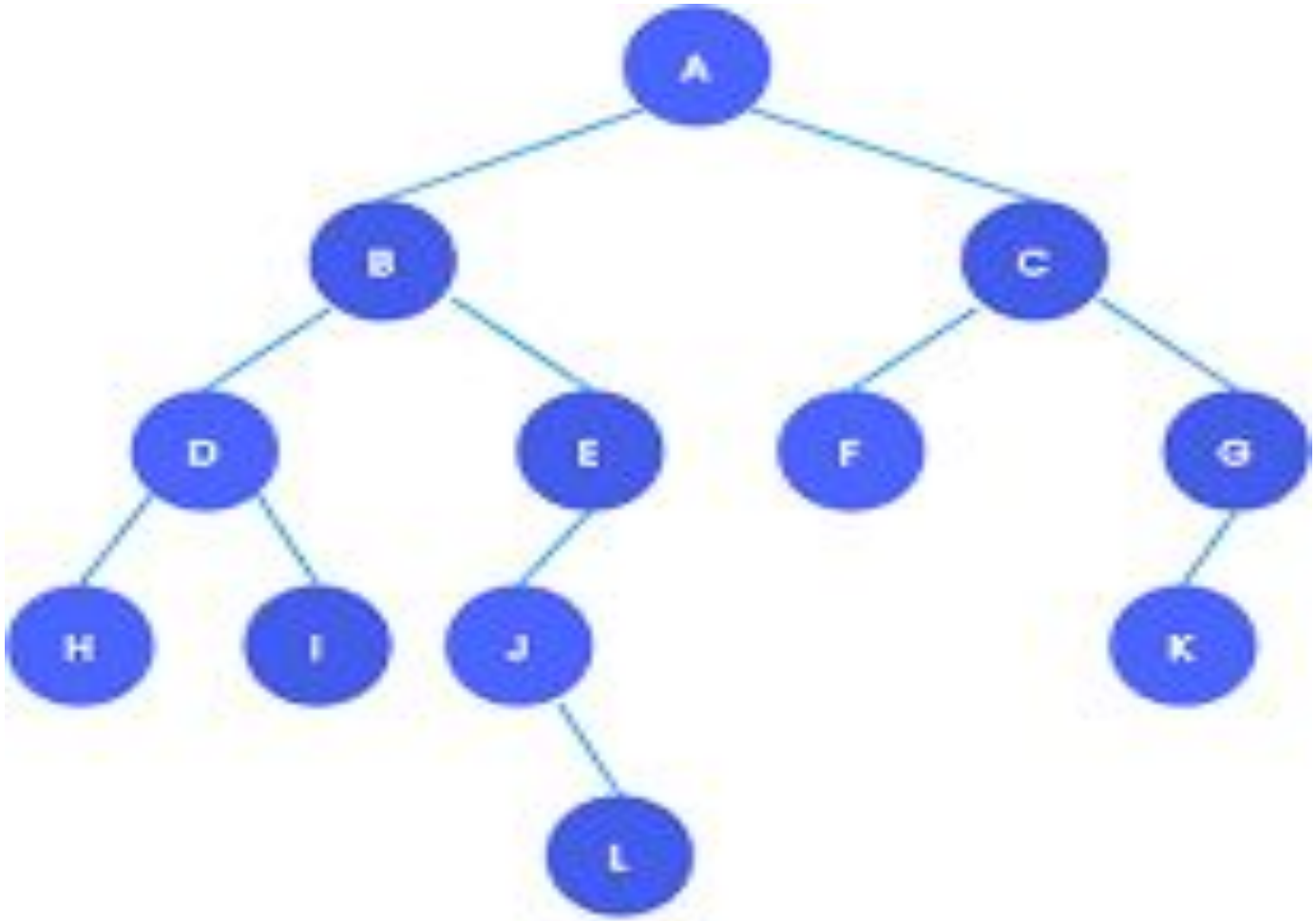
TRAVERSAL ORDER: G, L, H, D, E, B, I, K, J, F, C, and A

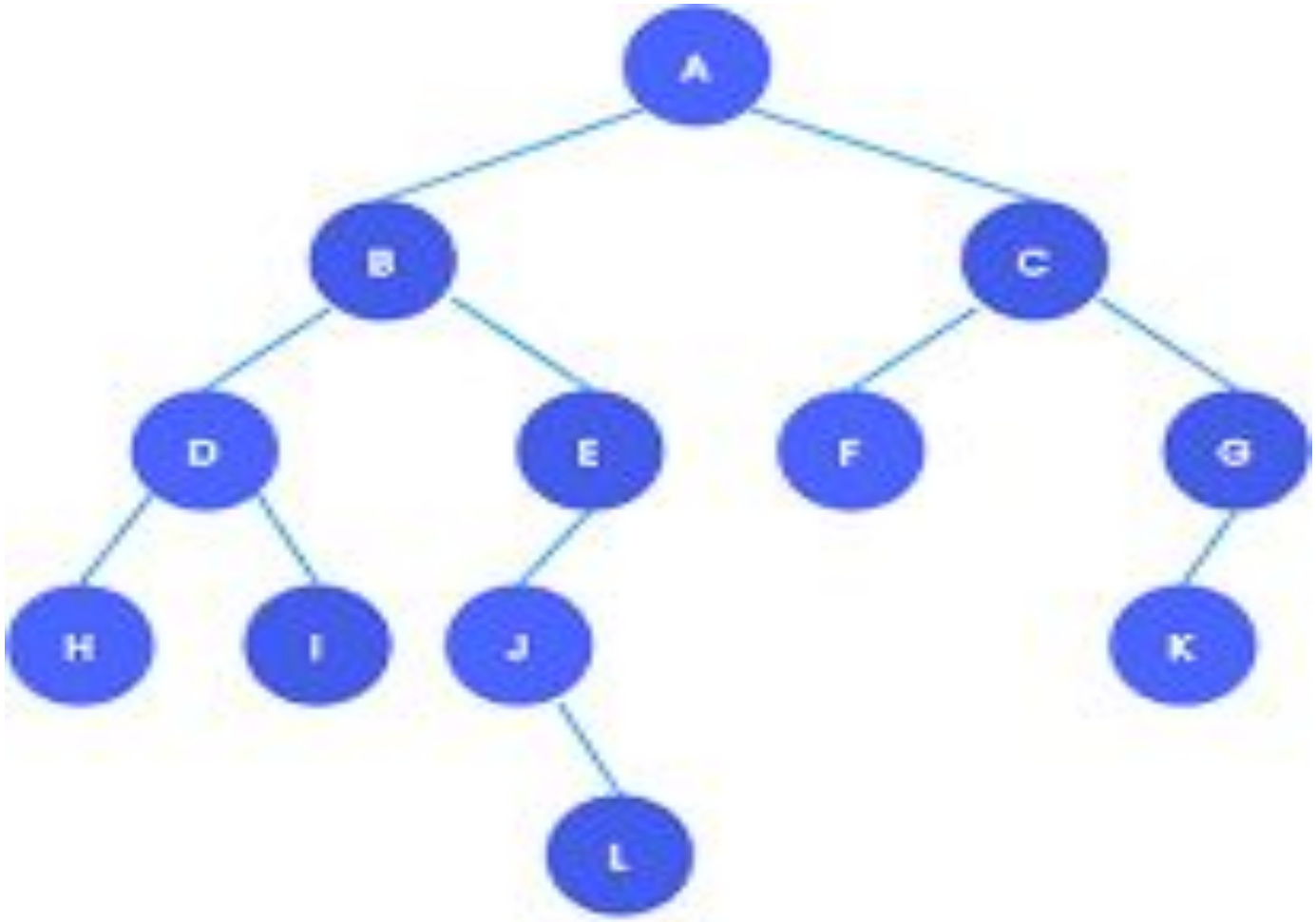
TRAVERSAL ORDER: D, B, H, I, G, F, E, C, and A



Pre-Order	ABDHIECFGJ
In-Order	HDIBEAFCJG
Post-Order	HIDEBFJGCA


F
B
G
A
D
I
C
E
H


A
B
C
D
E
F
G
H
I
J
K
L



In-order Traversal - H, D, I, B, J, L, E, A, F, C, K ,G

Pre-order Traversal - A, B, D, H, I, E, J, L, C, F, G, K

Post-order Traversal - H, I, D, L, J, E, B, F, K, G, C, A

Binary Search Tree

Binary Search Tree

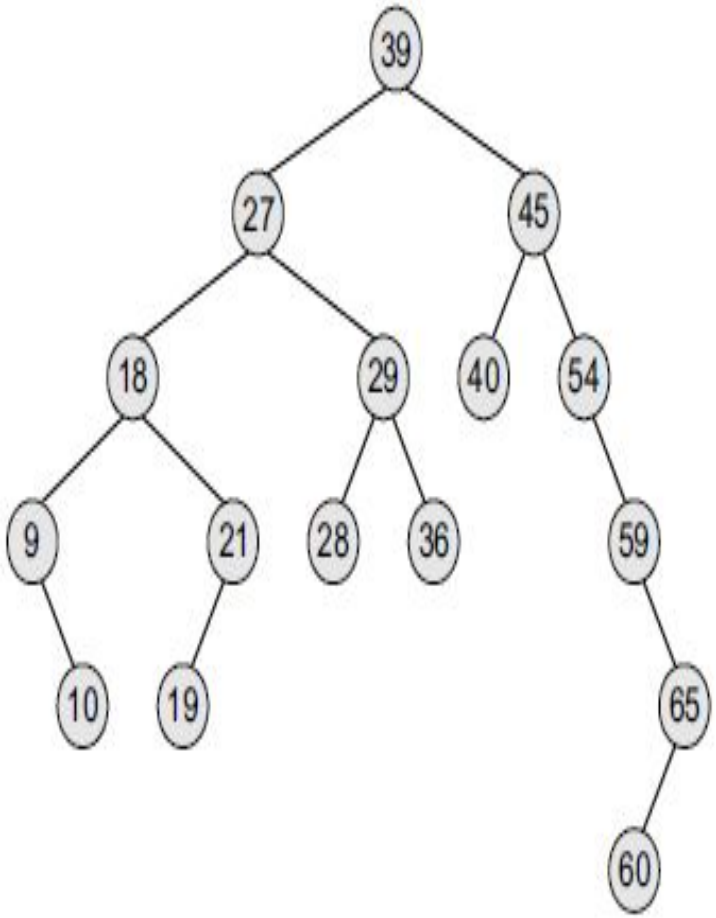
1. A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order.

2. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node.

3. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node.

4. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)

Binary Search Tree



- The root node is 39.
- The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node.
- The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65.
- Recursively, each of the sub-trees also obeys the binary search tree constraint. For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10, 18, 19, 21) are smaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node's value.

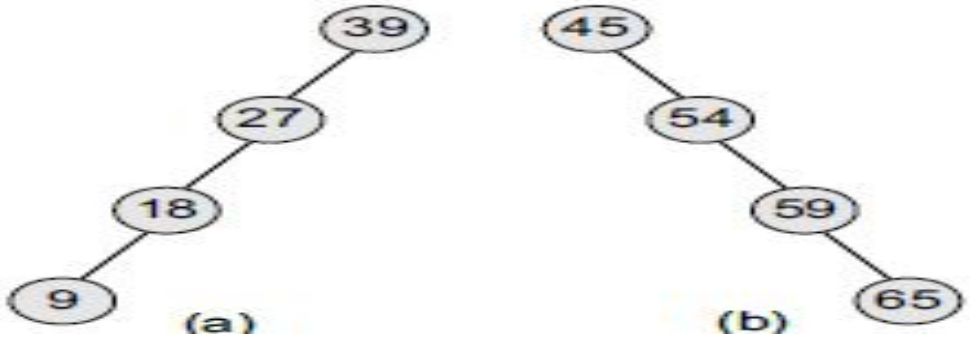
Binary Search Tree - Operations

- Binary search trees speed up the insertion and deletion operations. The tree has a speed Advantage when the data in the structure changes rapidly.

- **Binary search trees are considered to be efficient data structures** especially when compared with sorted linear arrays and linked lists.

- In a sorted array, searching can be done in $O(\log_2 n)$ time, but insertions and deletions are quite expensive.
- In contrast, inserting and deleting elements in a linked list is easier, but searching for an element is done in O(n) time.

- However, in the **worst case, a binary search tree will take O(n) time to search** for an element. The worst case would occur when the tree is a linear chain of nodes as given in Figure.



To summarize, a binary search tree is a binary tree with the following properties:

1. The left sub-tree of a node N contains values that are less than N's value.

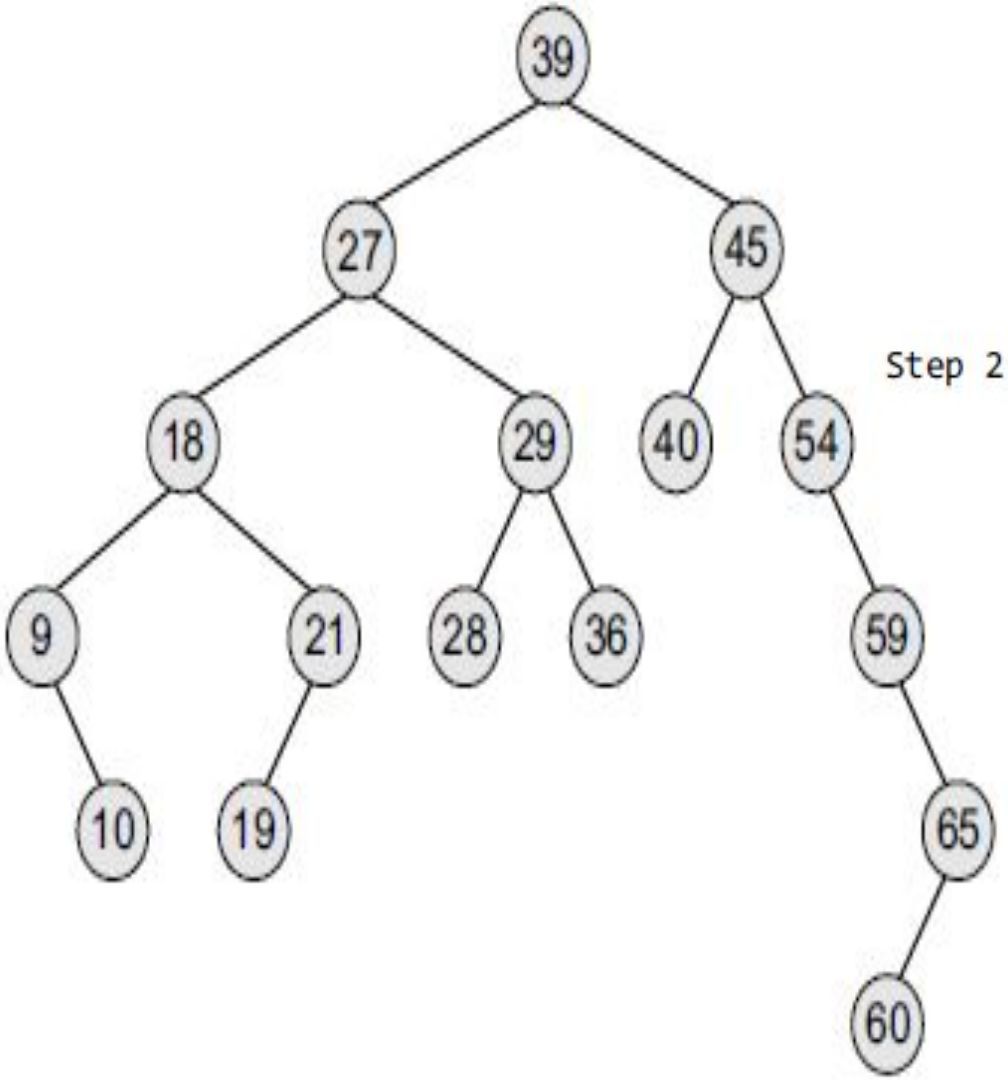
2. The right sub-tree of a node N contains values that are greater than N's value.

3. Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

Binary Search tree Operations

Searching for a node in Binary Search Tree

```
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
```

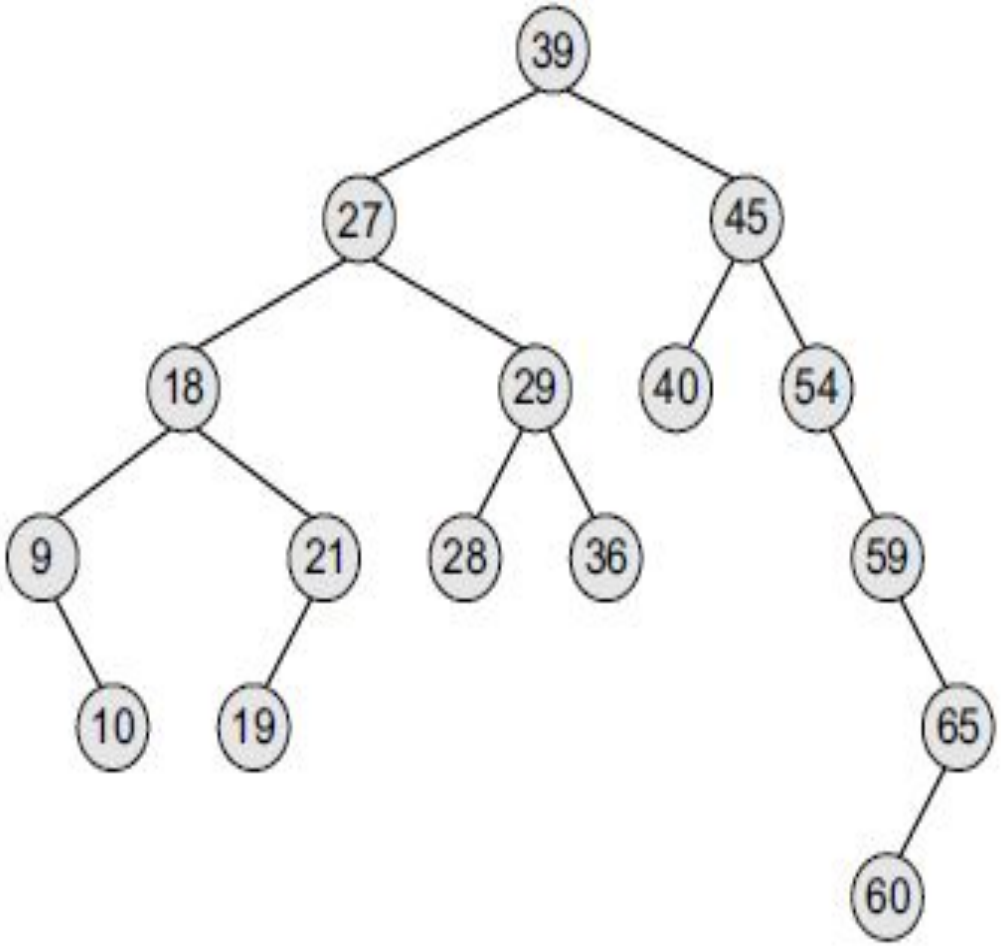



```
searchElement (TREE, VAL)

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
```

Insertion into 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
```

Insertion into 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
```

Construct Binary Search Tree
by inserting following Nodes
39,27,45,18,29,40,9,21,10,19,54,59,65,60

```
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
```

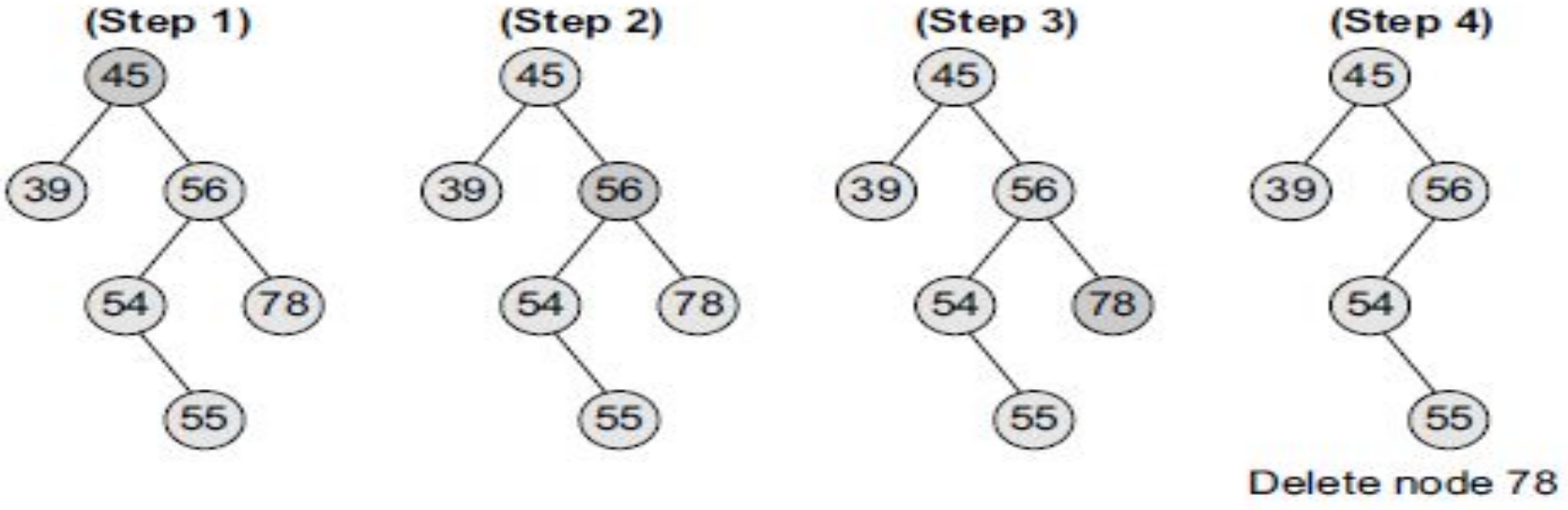
Node Deletion
Binary Search Tree

Deletion from Binary Search Tree

Case 1: Deleting a Node that has No Children

binary search tree given in Figure

If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.



Case 2: Deleting a Node with One Child

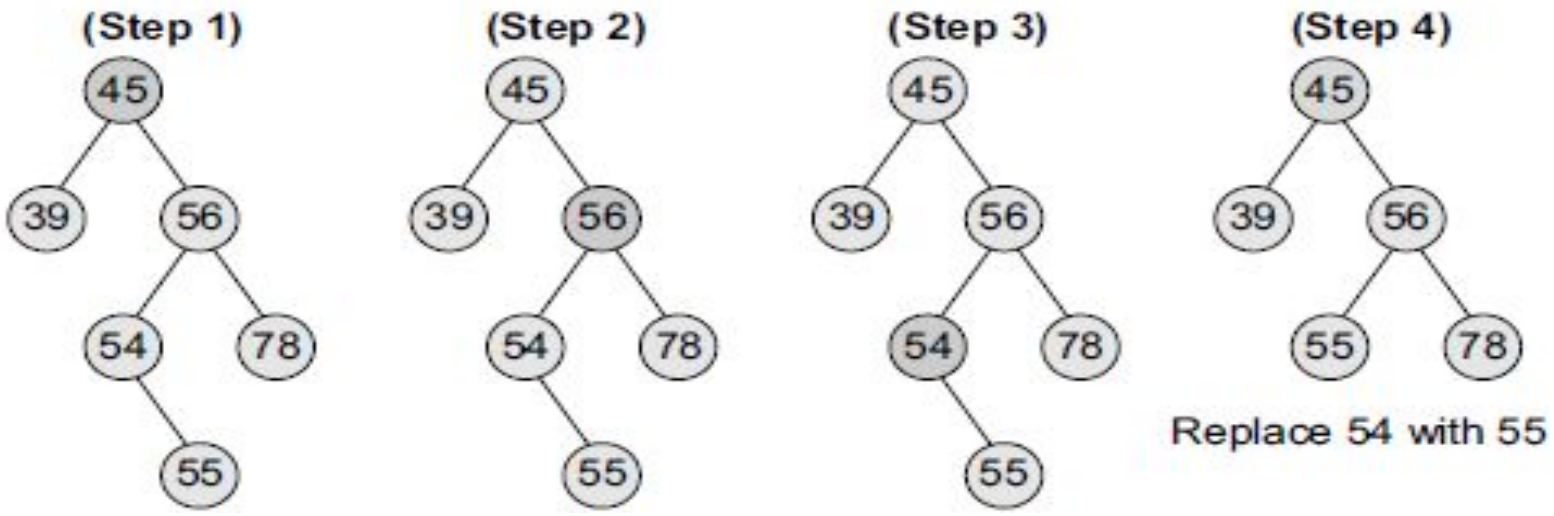
the **node's child is set as the child of the node's parent**.

In other words, replace the node with its child.

Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent.

Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.

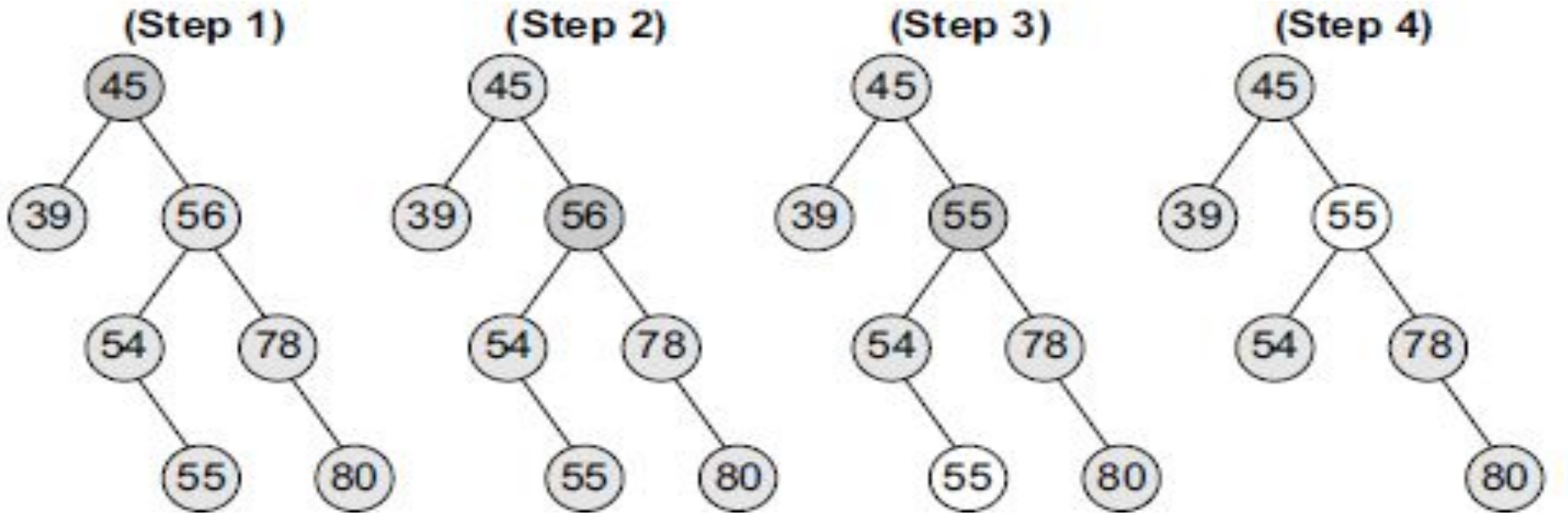
shown in Figure that how deletion of **node 54** is handled.



Case 3: Deleting a Node with 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).

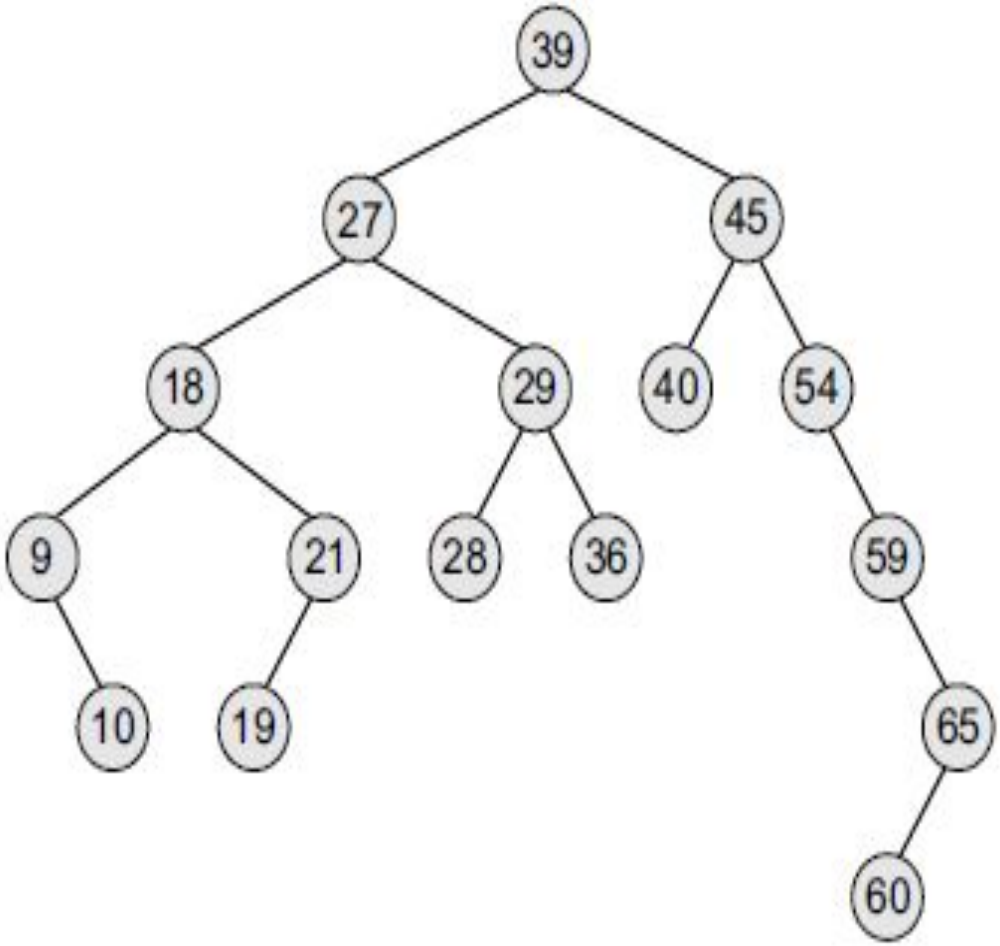
The in-order predecessor can then be deleted using any of the above cases. Look at the binary search tree given in Figure and see how deletion of node with **value 56** is handled.



Algorithm for Deletion

```
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
```

Example- Deletion




```
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]
   p 2: END
```