Trees

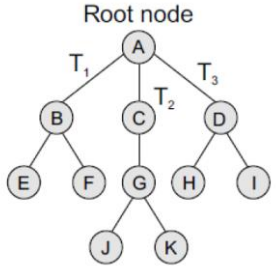


Fig 9.1 Tree

Root node The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

Sub-trees If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.

Leaf node A node that has no children is called the leaf node or the terminal node.

Path A sequence of consecutive edges is called a path. For example, in Fig. 9.1, the path from the root node A to node I is given as: A, D, and I.

Ancestor node An ancestor of a node is any 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.

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 Degree of a node is equal to the 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.

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.

A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.

Figure 4.2 shows a binary tree. In the figure, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.

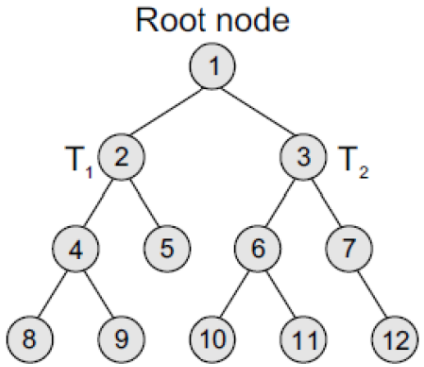


Figure 4.2 Binary tree

Note that the left sub-tree of the root node consists of the nodes: 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of nodes: 3, 6, 7, 10, 11, and 12.

In the tree, root node 1 has two successors: 2 and 3. Node 2 has two successor nodes: 4 and 5. Node 4 has two successors: 8 and 9. Node 5 has no successor. Node 3 has two successor nodes: 6 and 7. Node 6 has two successors: 10 and 11. Finally, node 7 has only one successor: 12.

A binary tree is recursive by definition as every node in the tree contains a left sub-tree and a right sub-tree. Even the terminal nodes contain an empty left sub-tree and an empty right sub-tree.

Terminology:

Parent If N is any node in T that has left successor S_1 and right successor S_2, then N is called the parent of S_1 and S_2. Correspondingly, S_1 and S_2 are called the left child and the right child of N. Every node other than the root node has a parent.

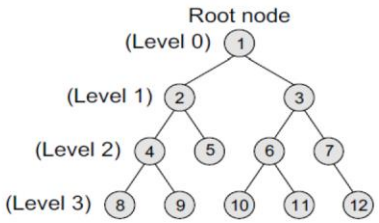


Figure 4.3 Levels in binary tree

Level number Every node in the binary tree is assigned a level number (refer Fig. 4.3). The root node is defined to be at level 0. The left and the right child of the root node have a level number 1. Similarly, every node is at one level higher than its parents. So all child nodes are defined to have level number as parent's level number + 1.

Degree of a node It is equal to the number of children that a node has. The degree of a leaf node is zero. For example, in the tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1.

Sibling All nodes that are at the same level and share the same parent are called siblings (brothers). For example, nodes 2 and 3; nodes 4 and 5; nodes 6 and 7; nodes 8 and 9; and nodes 10 and 11 are siblings.

Leaf node A node that has no children is called a leaf node or a terminal node. The leaf nodes in the tree are: 8, 9, 5, 10, 11, and 12.

Similar binary trees Two binary trees T and T' are said to be similar if both these trees have the same structure. Figure 4.4 shows two similar binary trees.

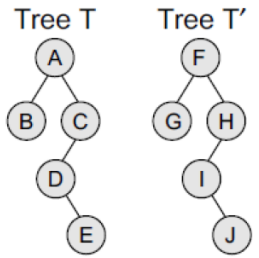


Figure 4.4 Similar binary trees

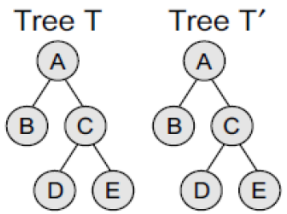


Figure 4.5 T′ is a copy of T

Copies Two binary trees T and T' are said to be copies if they have similar structure and if they have same content at the corresponding nodes. Figure 4.5 shows that T' is a copy of T.

Edge It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly n – 1 edges because every node except the root node is connected to its parent via an edge.

Path A sequence of consecutive edges. For example, in Fig. 4.3, the path from the root node to the node 8 is given as: 1, 2, 4, and 8.

Depth The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

Height of a tree It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1.

A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. This is because every level will have at least one node and can have at most 2 nodes. 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. The height of a binary tree with n nodes is at least $\log_2(n+1)$ and at most n.

Complete Binary Trees

A complete binary tree is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible.
In a complete binary tree T_n, there are exactly n nodes and level r of T can have at most 2^r nodes. Figure 9.7 shows a complete binary tree.

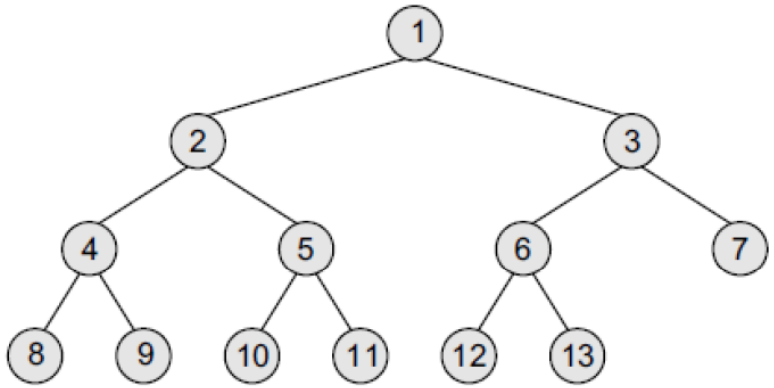


Figure 4.6 Complete binary tree

Note that in Fig. 4.6, level 0 has $2^0 = 1$ node, level 1 has $2^1 = 2$ nodes, level 2 has $2^2 = 4$ nodes, level 3 has 6 nodes which is less than the maximum of $2^3 = 8$ nodes.

In Fig. 4.6, tree T_{13} has exactly 13 nodes. They have been purposely labelled from 1 to 13, so that it is easy for the reader to find the parent node, the right child node, and the left child node of the given node.

The formula can be given as—if K is a parent node, then its left child can be calculated as $2 \times K$ and its right child can be calculated as $2 \times K + 1$.
For example, the children of the node 4 are 8 (2×4) and 9 ($2 \times 4 + 1$).
Similarly, the parent of the node K can be calculated as $| K/2 |$.
Given the node 4, its parent can be calculated as $| 4/2 | = 2$. The height of a tree T_n having exactly n nodes is given as: $H_n = | \log_2 (n + 1) |$

Extended Binary Trees

A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children. Figure 4.7 shows how an ordinary binary tree is converted into an extended binary tree.
In an extended binary tree, nodes having two children are called internal nodes and nodes having no children are called external nodes. In Fig. 4.7, the internal nodes are represented using circles and the external nodes are represented using squares.
To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes.

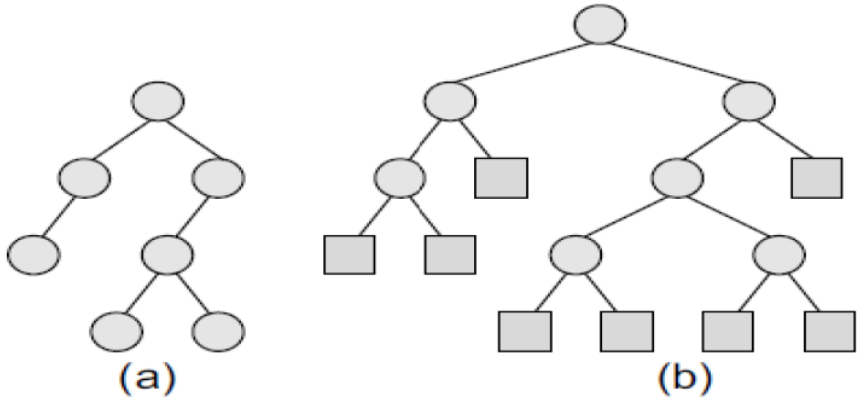


Figure 4.7 (a) Binary tree and (b) extended binary tree

Binary Tree Representation in Memory

Linked representation of binary trees In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.
So in C, the binary tree is built with a node type given below.

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

Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty. Consider the binary tree given in Fig. 4.2. The schematic diagram of the linked representation of the binary tree is shown in Fig. 4.8.

In Fig. 4.8, the left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).

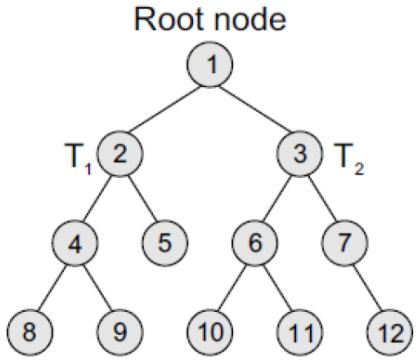


Figure 4.2 Binary tree

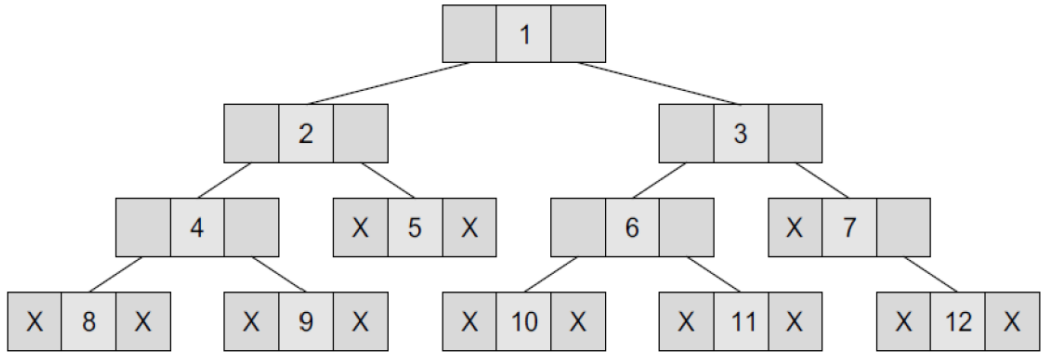


Figure 4.8 Linked representation of a binary tree

A sequential binary tree follows the following rules:

- ✓ A one-dimensional array, called TREE, is used to store the elements of tree.
- ✓ The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
- ✓ The children of a node stored in location K will be stored in locations (2 × K) and (2 × K+1).
- ✓ The maximum size of the array TREE is given as (2h–1), where h is the height of the tree.
- ✓ An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.

Figure 4.9 shows a binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4.

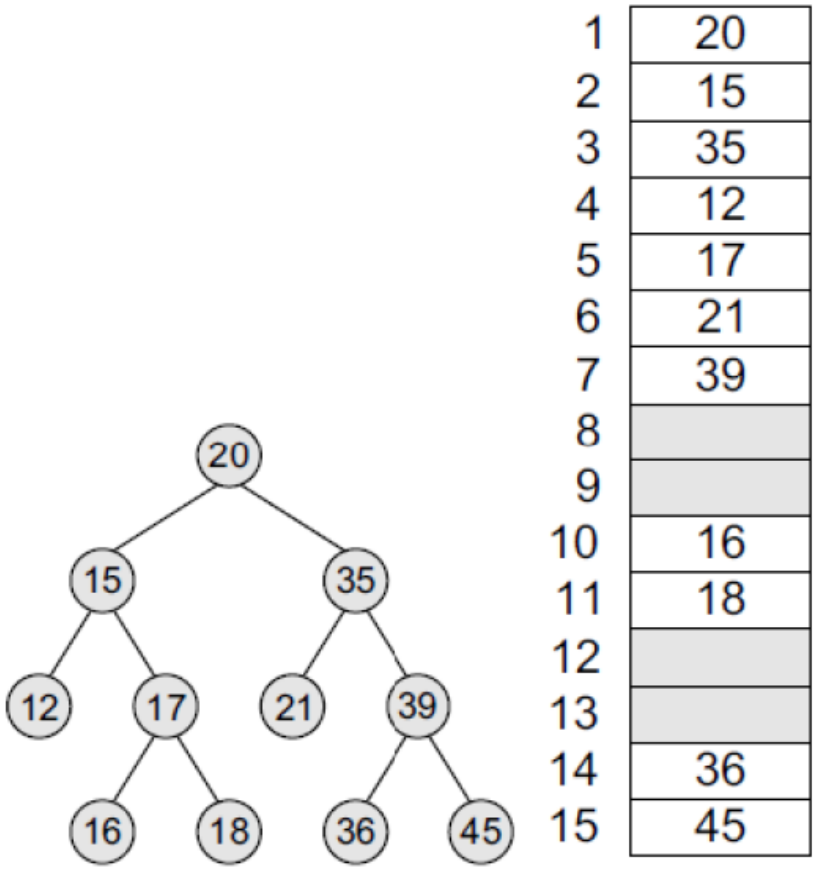


Figure 4.9 Binary tree and its sequential representation

Traversing Binary Tree

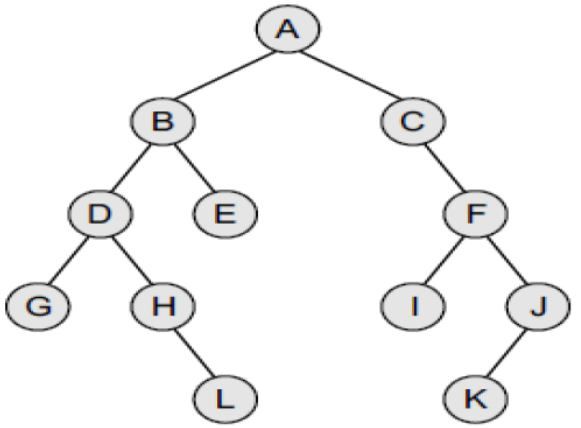
Pre-order Traversal

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.

Pre-order traversal is also called as depth-first traversal. In this algorithm, the left sub-tree is always traversed before the right sub-tree. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right).

```
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
```



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

In-order Traversal

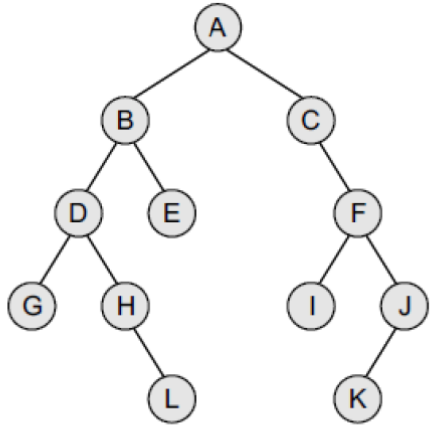
To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

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

```
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
```

In-order traversal is also called as symmetric traversal. In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree.

The word 'in' in the in-order specifies that the root node is accessed in between the left and the right sub-trees. In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right).



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

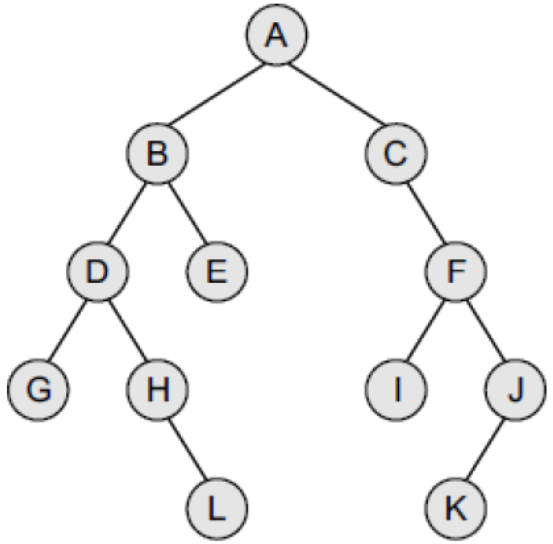
Post-order Traversal

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.

In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node. The word 'post' in the post-order specifies that the root node is accessed after the left and the right sub-trees.
Post-order algorithm is also known as the LRN traversal algorithm (Left-Right-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
```

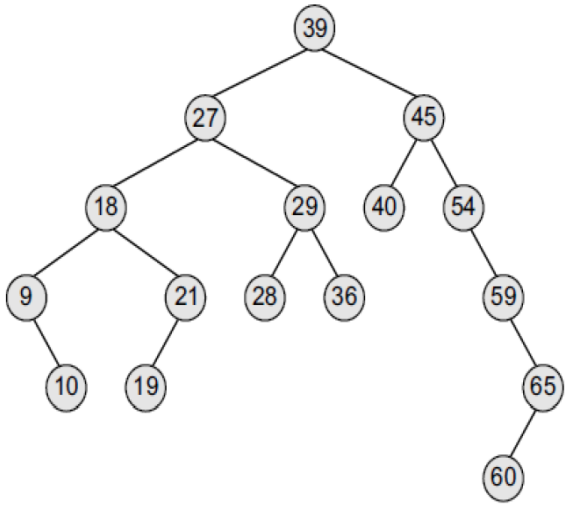
DATA STRUCTURES



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

Binary Search Tree

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. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. 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.)



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.

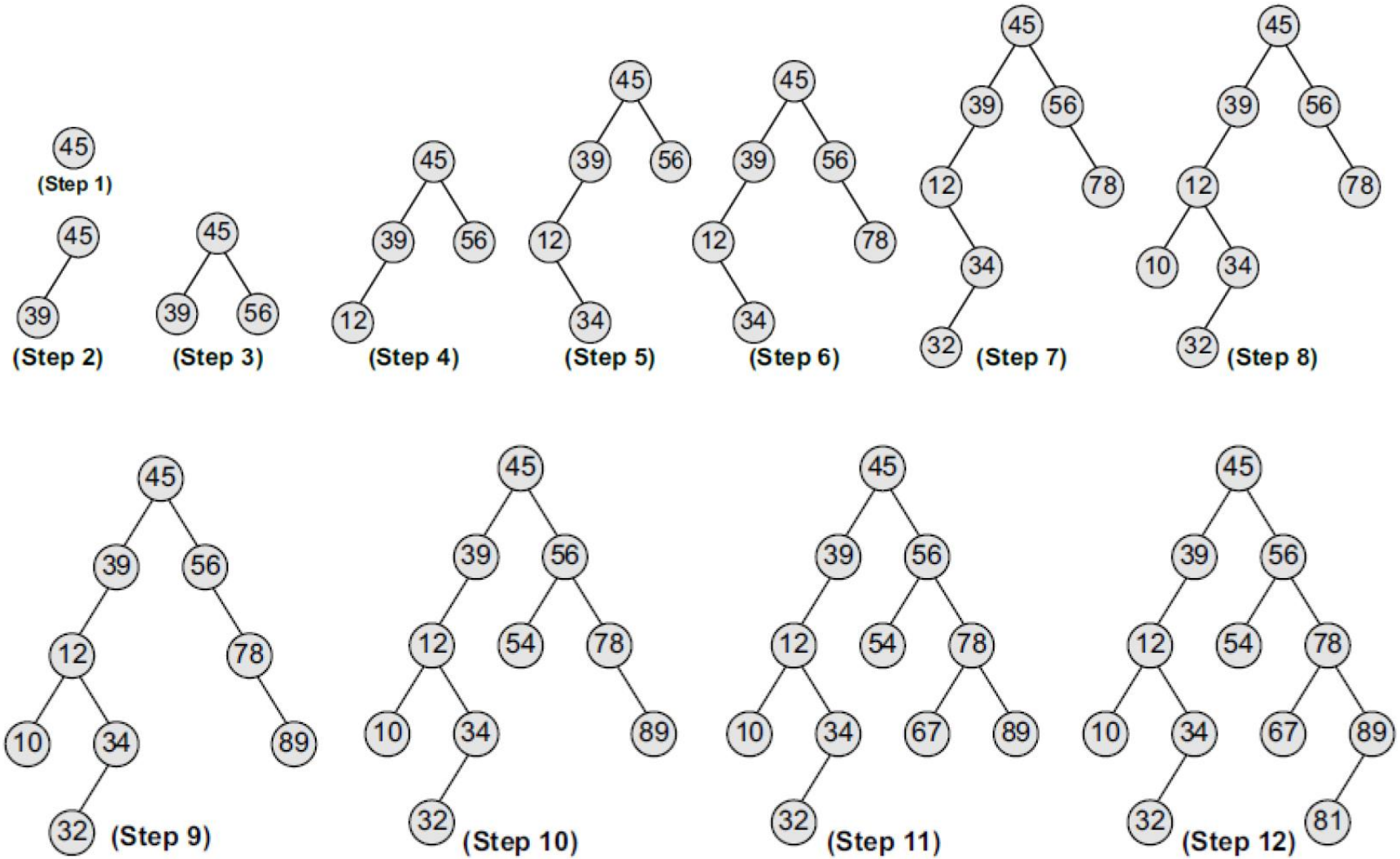
Binary search trees also speed up the insertion and deletion operations. The tree has a speed advantage when the data in the structure changes rapidly.
Binary search trees are considered to be efficient data structures especially when compared with sorted linear arrays and linked lists. In a sorted array, 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.

To summarize, 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.

Example:

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



Inserting a New Node in a Binary Search Tree

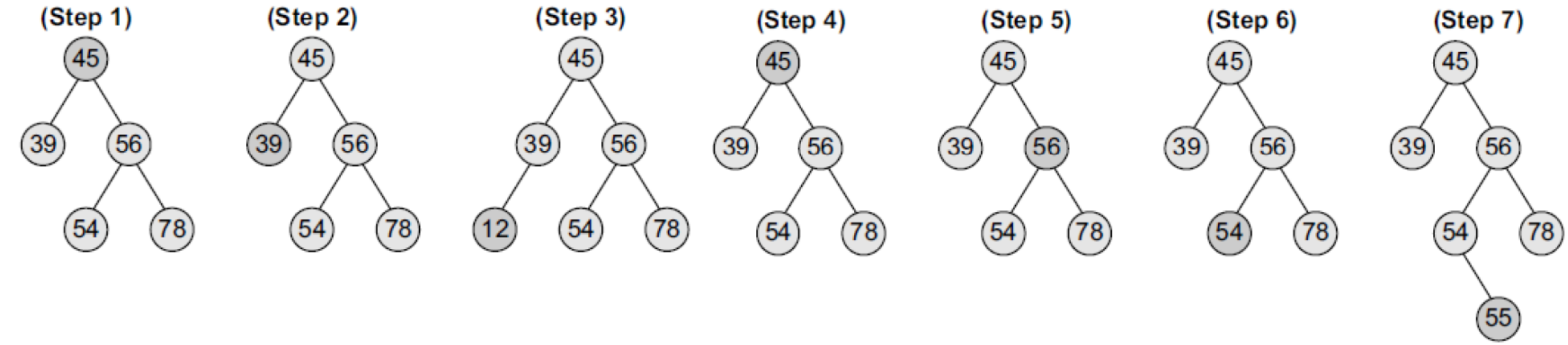
The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

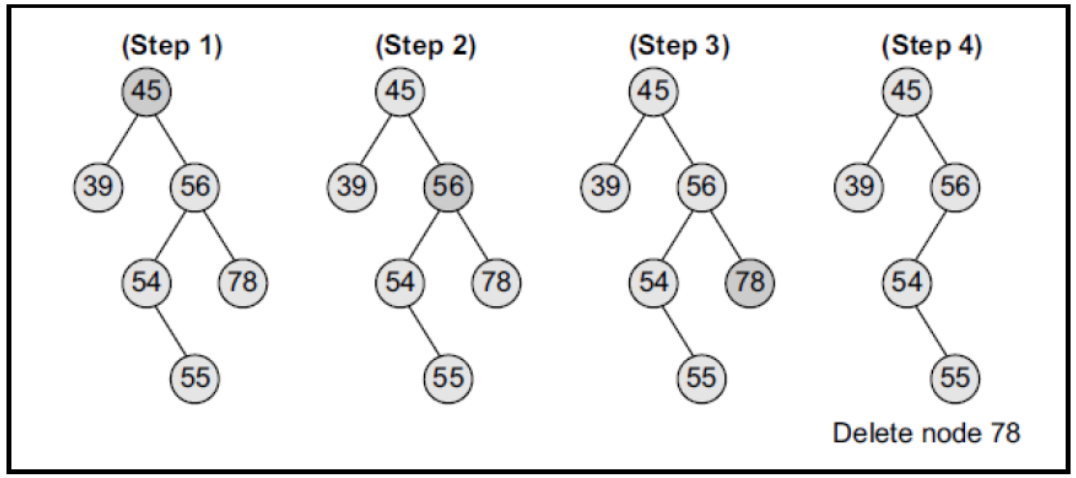
The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case.

```
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
```

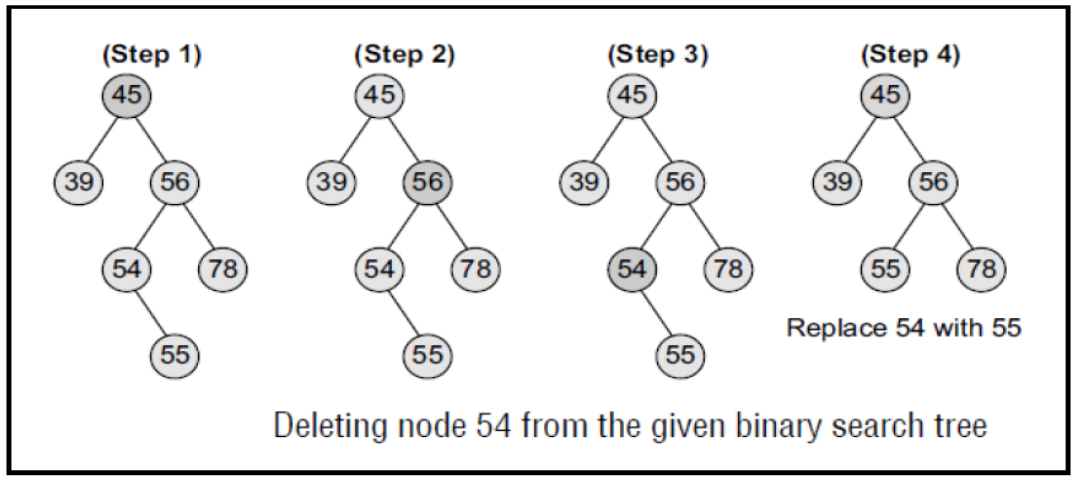
Example: Inserting nodes with values 12 and 55 in the given binary search tree





Case1: Deleting a Node that has No Children

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



To handle this case, 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.

Look at the binary search tree shown in figure and see how deletion of node 54 is handled.

Balanced AVL Tree

AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. The tree is named AVL in honour of its inventors. In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the AVL tree is also known as a height-balanced tree. The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(log n).

The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the Balance Factor. Thus, every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of −1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

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.

The four categories of rotations are:

- **LL rotation:** The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
- **RR rotation:** The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
- **LR rotation:** The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
- **RL rotation:** The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

LL Rotation

Example Consider the AVL tree given in Fig. and insert 18 into it.

Solution

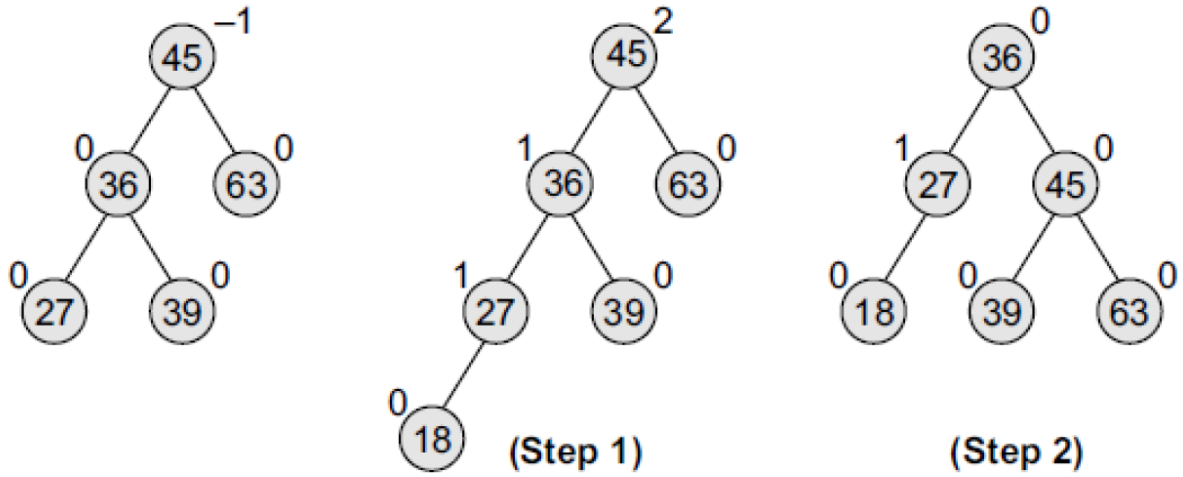
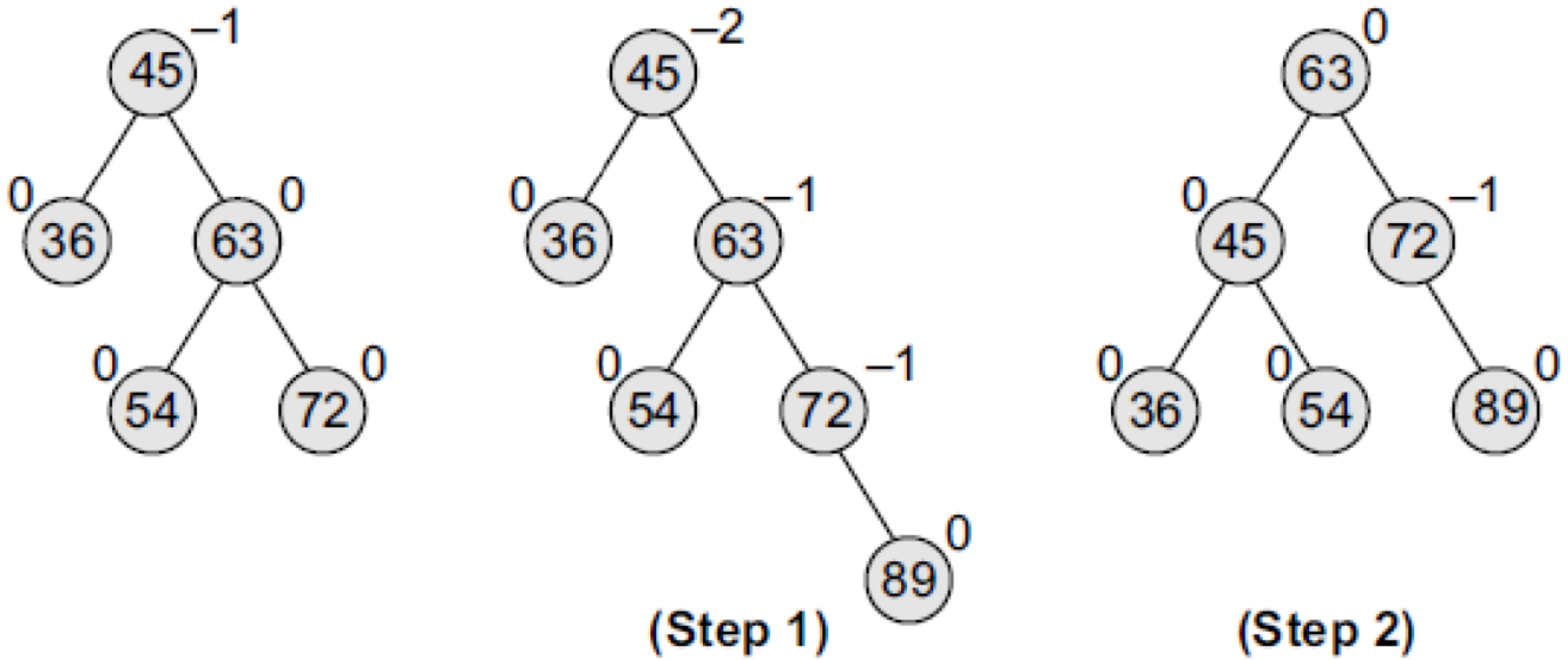


Figure AVL tree

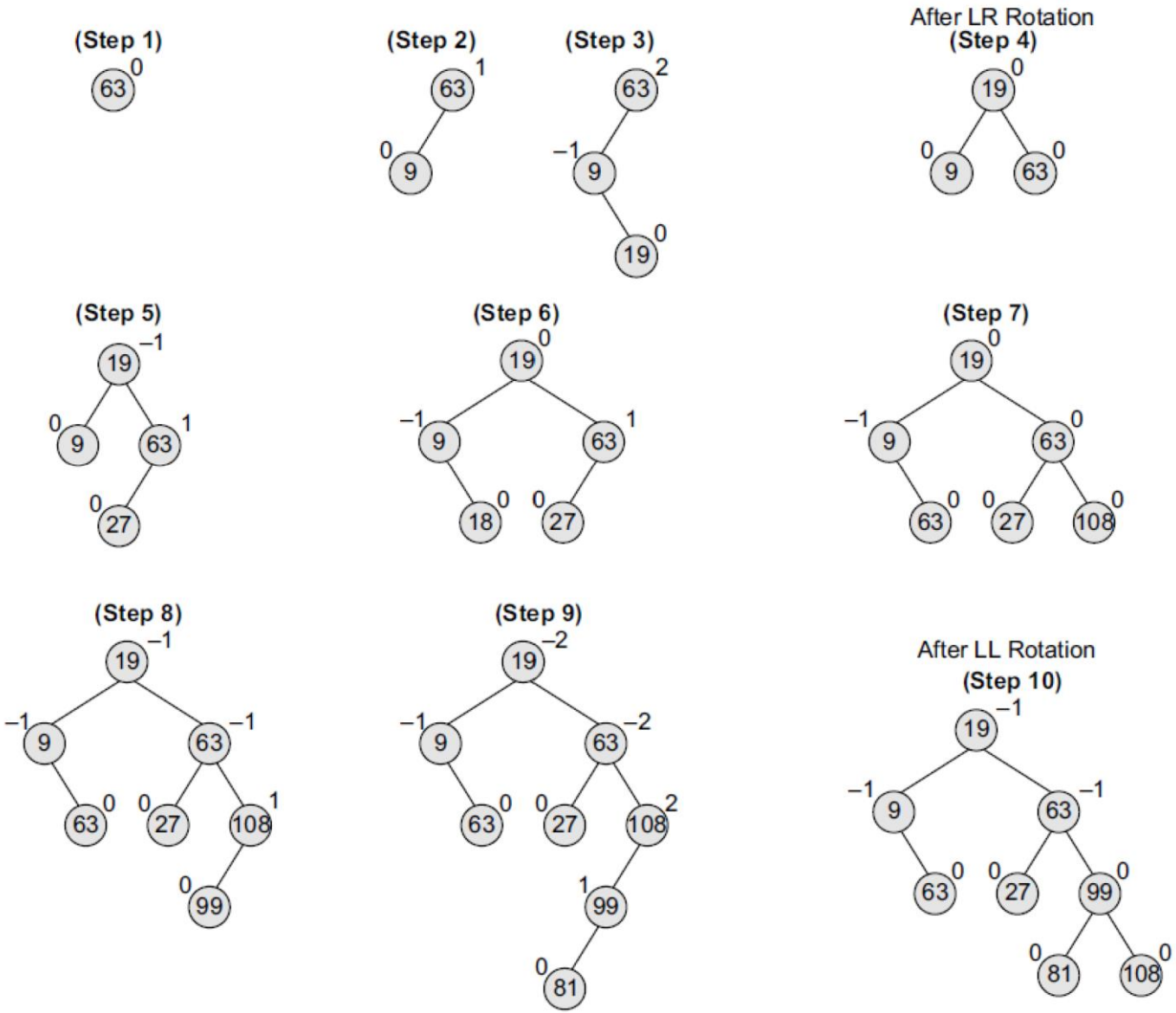
RR Rotation

Example Consider the AVL tree given in Fig. and insert 89 into it.

Solution



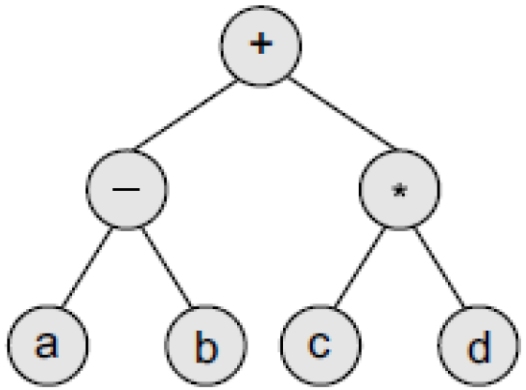
Example: Construct an AVL tree by inserting the following elements in the given order.
63, 9, 19, 27, 18, 108, 99, 81.



Use of Trees

- Trees are used to store simple as well as complex data. Here simple means an integer value, character value and complex data means a structure or a record.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating system use.
- Another variation of tree, B-trees are prominently used to store tree structures on disc. They are used to index a large number of records.
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
- Trees are an important data structure used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables.

Expression Trees

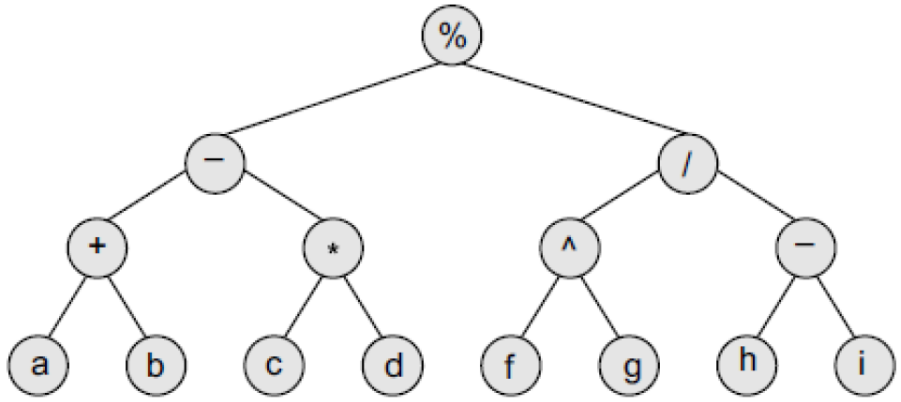


Binary trees are widely used to store algebraic expressions. For example, consider the algebraic expression given as:

Exp = (a – b) + (c * d)

This expression can be represented using a binary tree as shown in Figure.

Given an expression, Exp = ((a + b) – (c * d)) % ((e ^f) / (g – h)), construct the corresponding binary tree.



Expression tree