CHAPTER 4:
TREES

Content

- Introduction, Tree Terminologies
- Binary Tree
- Binary Tree Representation
- Types of Binary Tree
- Binary Tree Traversals
- Binary Search Tree
- Operations on Binary Search Tree
- Applications of Binary Tree-Expression Tree, Huffman Encoding
- Search Trees-AVL, rotations in AVL Tree

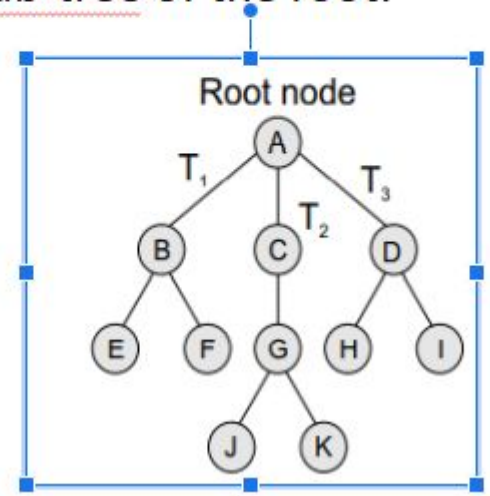
Introduction to Trees

✅ What is a Tree in Data Structures?

A **Tree** is a **hierarchical (non-linear)** data structure consisting of **nodes**.

It is used to represent **hierarchical relationships**, like a family tree, file system, or organization chart.

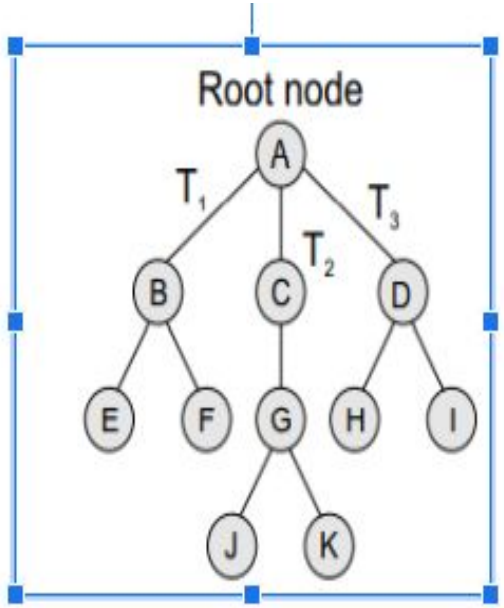
A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.



Basic Tree Terminology

📌 Key Terms:

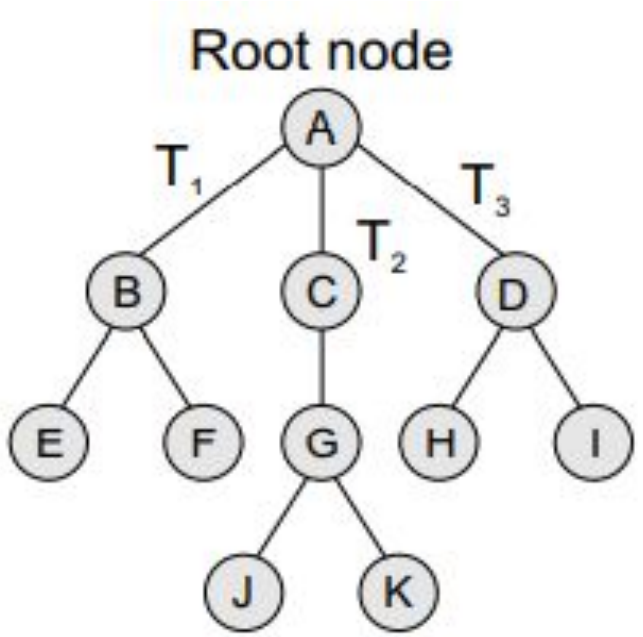
Term	Meaning
Node	Element of the tree (stores data + references to children)
Root	The topmost node of the tree (has no parent)
Parent	A node that has one or more children
Child	A node derived from another node (has a parent)
Leaf	A node with no children
Edge	The connection between two nodes
Subtree	A tree formed from a node and its descendants
Depth	Number of edges from root to that node
Height	Number of edges on the longest path from a node to a leaf
Level	Distance from the root (root is at level 0)



Basic Tree Terminology

- **Path:** A sequence of consecutive edges is called a path. For example, 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, 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.



Somaiya TRUST

Types of Trees

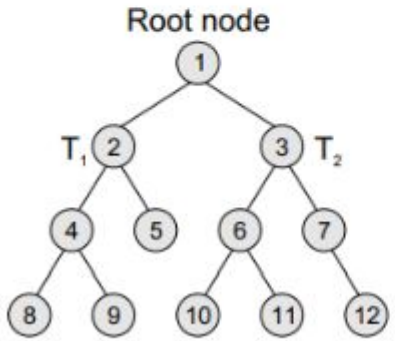
🌿 Types of Trees:

Tree Type	Description
Binary Tree	Each node has **at most 2 children** (left and right)
Binary Search Tree (BST)	Left child < root < Right child (sorted structure)
AVL Tree	Balanced BST (self-balancing)
B-Trees / B+ Trees	Used in databases and file systems (multi-child search trees)

Somaiya TRUST

Binary Tree

- Binary trees
  - 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.



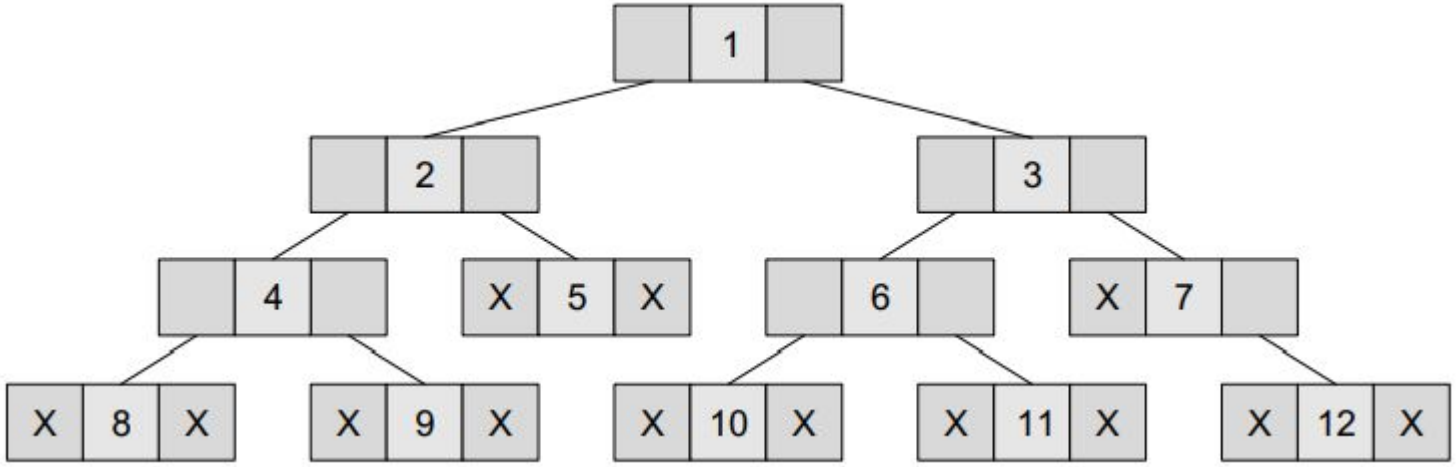
Binary tree

Somaiya TRUST

Representation of Binary Tree in memory

In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.

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.



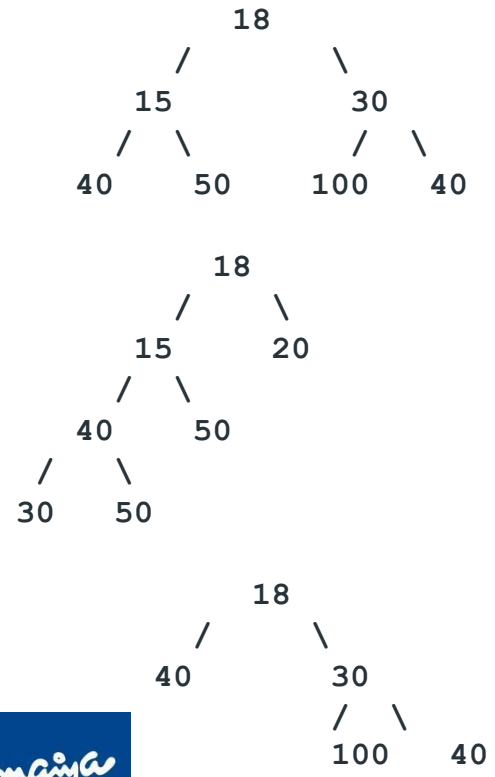
Somaiya TRUST

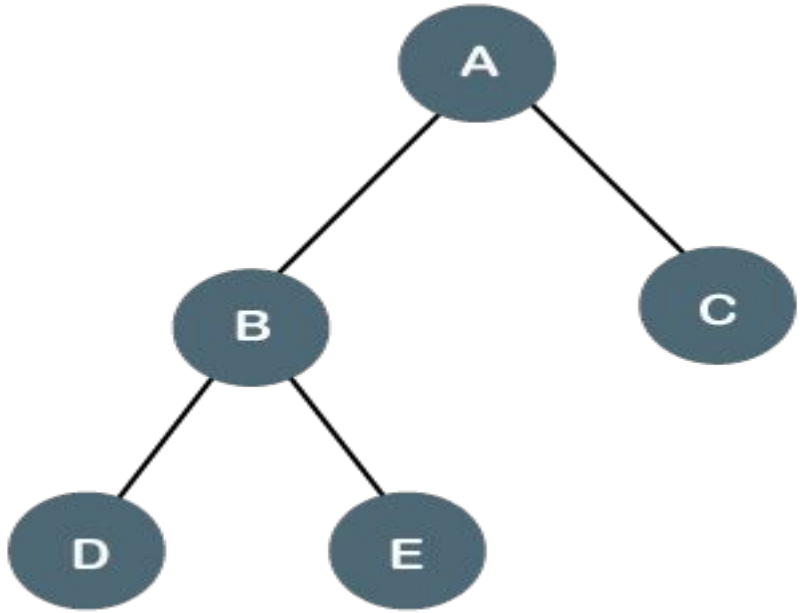
Types of Binary tree

Full Binary Tree:

A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.





Somaiya TRUST

Types of Binary tree

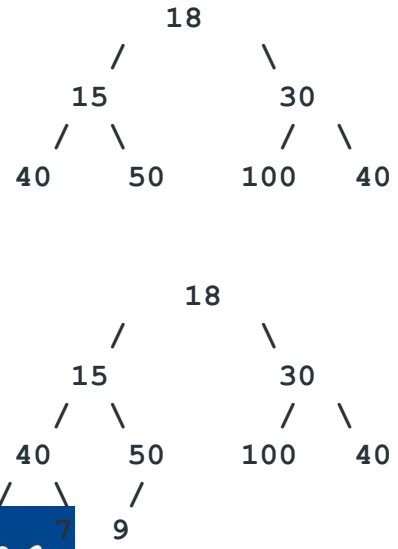
Complete Binary Tree:-

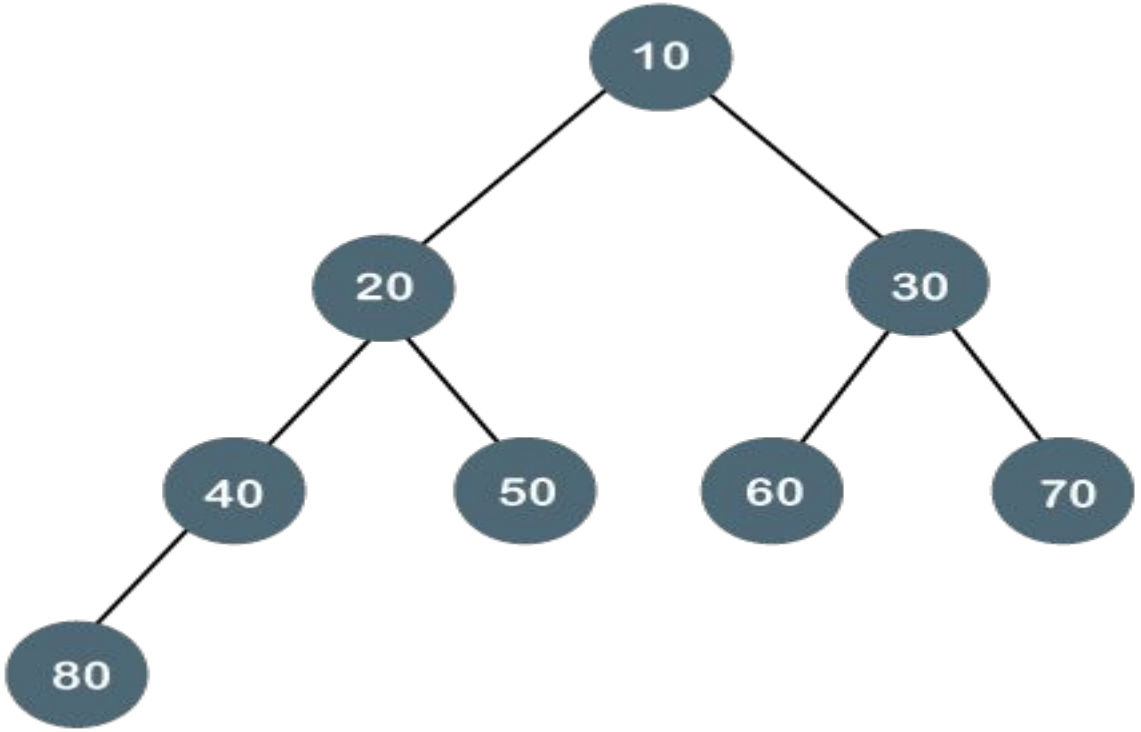
A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

A complete binary tree is just like a full binary tree, but with two major differences:

- **Every level must be completely filled**
- **All the leaf elements must lean towards the left.**
- **The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.**

The following are examples of Complete Binary Trees.





Somaiya TRUST

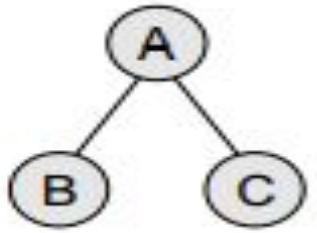
TRAVERSING A BINARY TREE

- **Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.**
- **There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited.**
  - Pre-Order
  - In-order
  - Post-Order

Somaiya TRUST

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 ->A B C**

Somaiya TRUST

Pre-order Algorithm

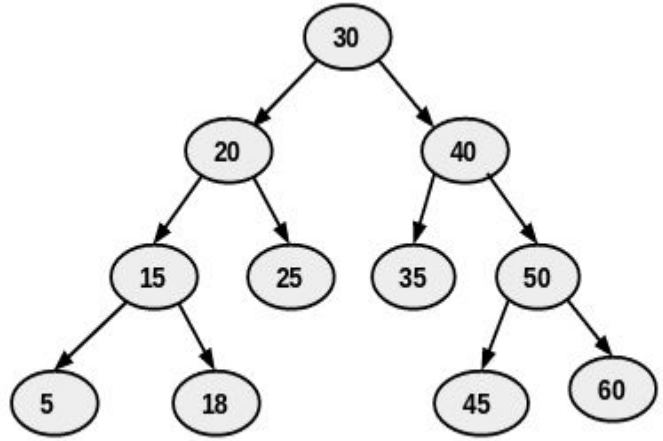
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:              Write TREE -> DATA
Step 3:              PREORDER(TREE -> LEFT)
Step 4:              PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Algorithm for pre-order traversal

Somaiya TRUST

pre order

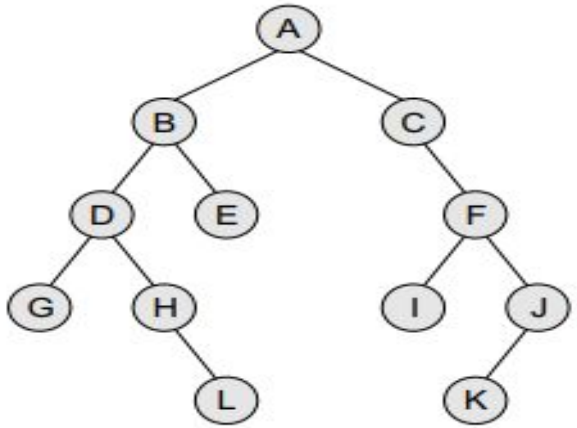
- Start with root node 30 .print 30 and recursively traverse the left subtree.
- next node is 20. now 20 have subtree so print 20 and traverse to left subtree of 20 .
- next node is 15 and 15 have subtree so print 15 and traverse to left subtree of 15.
- 5 is next node and 5 have no subtree so print 5 and traverse to right subtree of 15.
- next node is 18 and 18 have no child so print 18 and traverse to right subtree of 20.
- 25 is right subtree of 20 .25 have no child so print 25 and start traverse to right subtree of 3
- next node is 40. node 40 have subtree so print 40 and then traverse to left subtree of 40.
- next node is 35. 35 have no subtree so print 35 and then traverse to right subtree of 40.
- next node is 50. 50 have subtree so print 50 and traverse to left subtree of 50.
- next node is 45. 45 have no subtree so print 45 and then print 60(right subtree) of 50.
- our final output is {30 , 20 , 15 , 5 , 18 , 25 , 40 , 35 , 50 , 45 , 60}



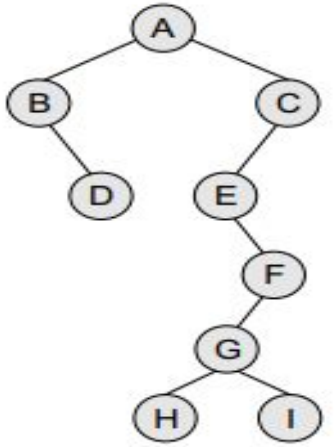
Somaiya TRUST

Example Pre-order

- **Pre-order for Tree-a**
  - A,B,D,G,H,L,E,C,F,I,J,K
- **Pre-order for Tree-b**
  - A,B,D,C,E,F,G,H,I



(a)

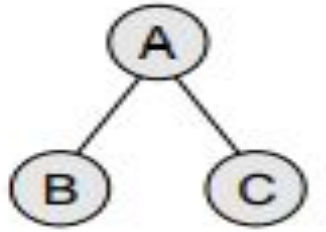


(b)

Somaiya TRUST

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.



- **In-Order: B A C**

Somaiya TRUST

In-order Algorithm

```
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
```

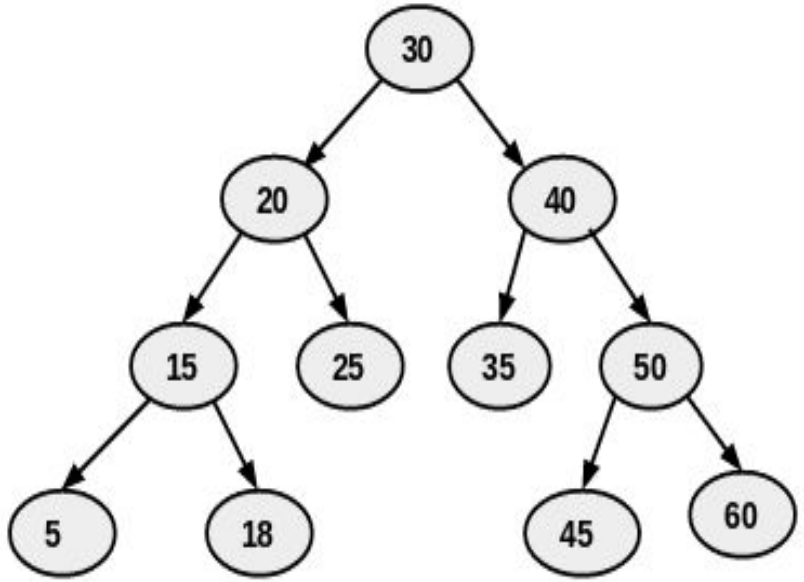
Algorithm for in-order traversal

Somaiya TRUST

Inorder Traversal

- we start recursive call from 30(root) then move to 20 (20 also have sub tree so apply in order on it),15 and 5.
- 5 have no child .so print 5 then move to it's parent node which is 15 print and then move to 15's right node which is 18.
- 18 have no child print 18 and move to 20 .print 20 then move it right node which is 25 .25 have no subtree so print 25.
- print root node 30 .
- now recursively traverse to right subtree of root node . so move to 40. 40 have subtree so traverse to left subtree of 40.
- left subtree of 40 have only one node which is 35. 35 had no further subtree so print 35. move to 40 and print 40.
- traverse to right subtree of 40. so move to 50 now have subtree so traverse to left subtree of 50 .move to 45 , 45 have no further subtree so print 45.
- move to 50 and print 50. now traverse to right subtree of 50 hence move to 60 and print 60.

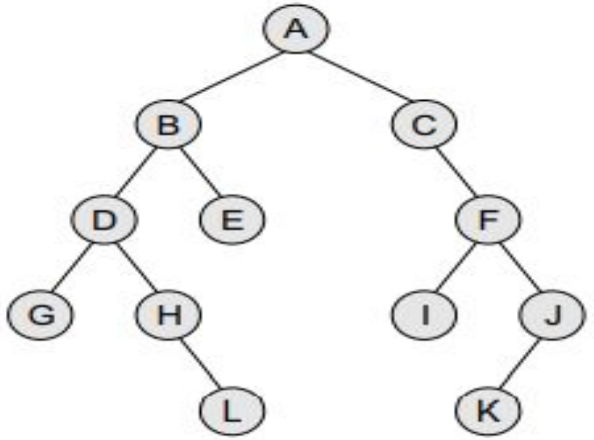
our final output is {5 , 15 , 18 , 20 , 25 , 30 , 35 , 40 , 45 , 50 , 60}



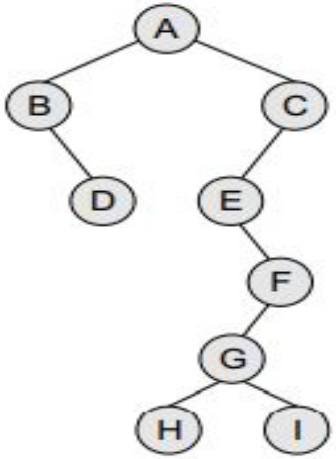
TRUST

In-order example

In-order Tree -a: G, D, H, L, B, E, A, C, I, F, K, and J
In-order Tree-b:B, D, A, E, H, G, I, F, and C



(a)

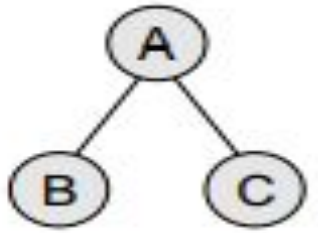


(b)

Somaiya TRUST

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.

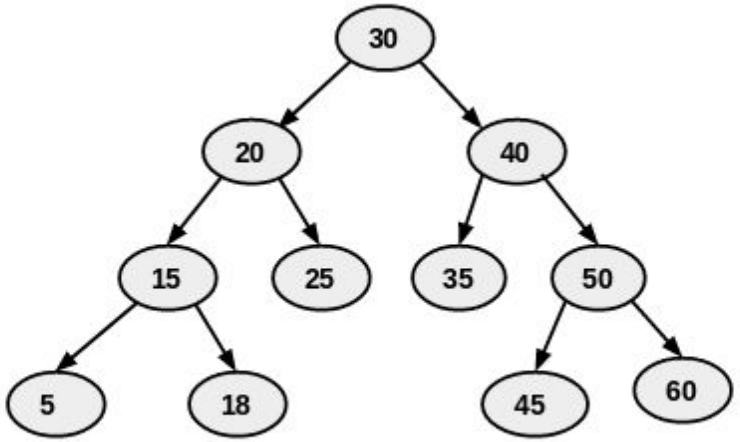


- **Post-order: B C A**

Somaiya TRUST

Post Order Traversal

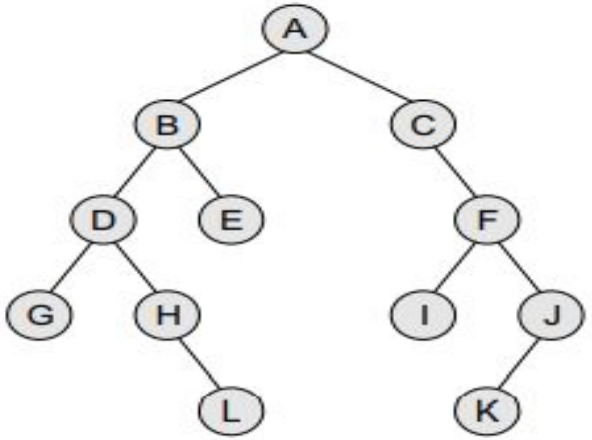
- We start from 30, and following Post-order traversal, we first visit the left subtree 20. 20 is also traversed post-order.
- 15 is left subtree of 20 .15 is also traversed post order.
- 5 is left subtree of 15. 5 have no subtree so print 5 and traverse to right subtree of 15 .
- 18 is right subtree of 15. 18 have no subtree so print 18 and then print 15. post order traversal for 15 is finished.
- next move to right subtree of 20.
- 25 is right subtree of 20. 25 have no subtree so print 25 and then print 20. post order traversal for 20 is finished.
- next visit the right subtree of 30 which is 40 .40 is also traversed post-order(40 have subtree).
- 35 is left subtree of 40. 35 have no more subtree so print 35 and traverse to right subtree of 40.
- 50 is right subtree of 40. 50 should also traversed post order.
- 45 is left subtree of 50. 45 have no more subtree so print 45 and then print 60 which is right subtree of 50.
- next print 50 . post order traversal for 50 is finished.
- now print 40 ,and post order traversal for 40 is finished.
- print 30. post order traversal for 30 is finished.
- our final output is {5 , 18 , 15 , 25 , 20 , 35 , 45 , 60 , 50 , 40 , 30}



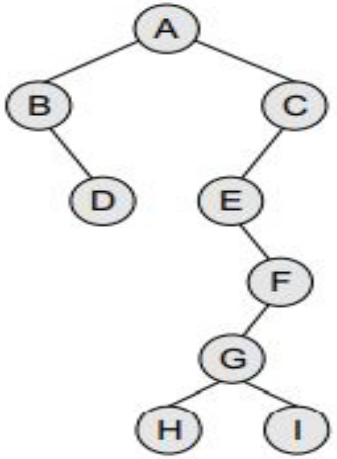
Somaiya TRUST

Post-order example

post-order Tree-a –G,L,H,D,E,B,I,K,J,F,C,A
post-order Tree-b:D,B,H,I,G,F,E,C,A



(a)



(b)

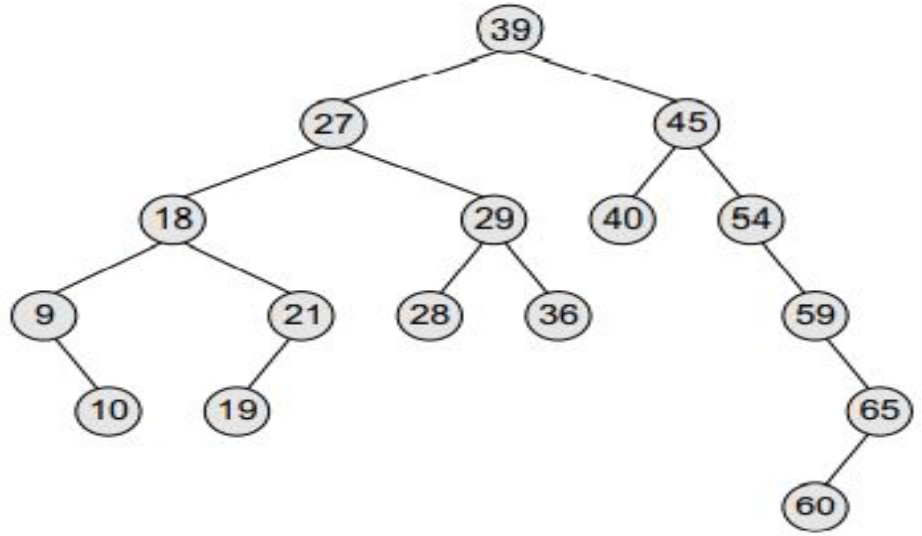
Somaiya TRUST

BINARY SEARCH TREES

- **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.**

Binary Search Tree

- 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.



Binary search tree

Somaiya TRUST

Why Binary Search Tree?

- **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.**

Example of Binary Search Tree

Example 10.1 State whether the binary trees in Fig. 10.3 are binary search trees or not.

Solution

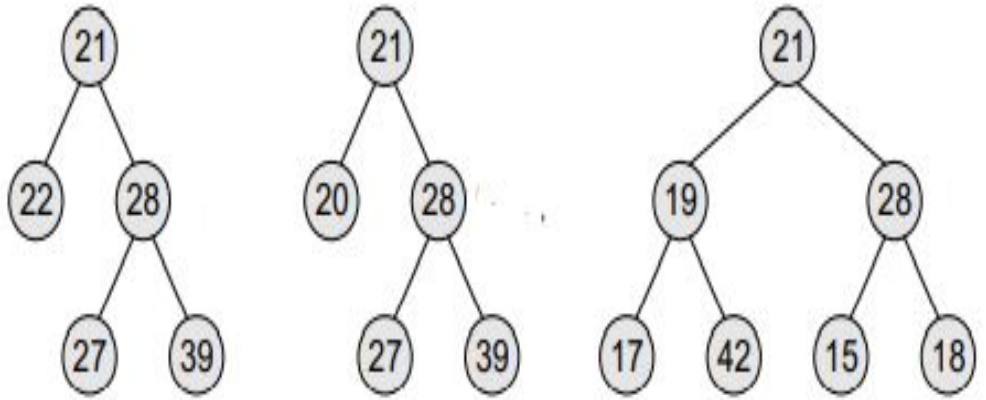


Figure 10.3 Binary trees

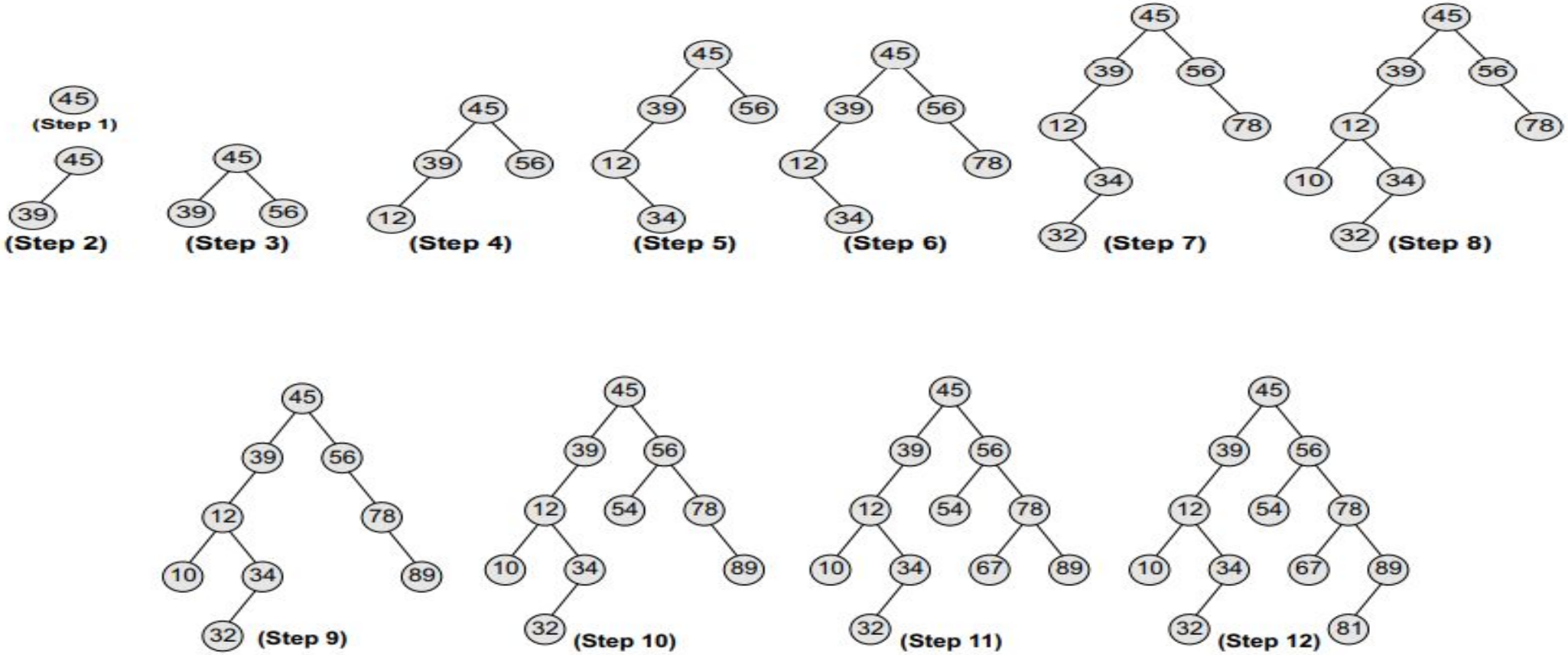
Somaiya TRUST

Example of Binary Search Tree

Example 10.2 Create a binary search tree using the following data elements:

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

Solution



Operations on Binary Search Tree

- **Search**
- **Insert**
- **Delete**

Somaiya TRUST

Search 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
```

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

Somaiya TRUST

Search in Binary Search tree(Example)

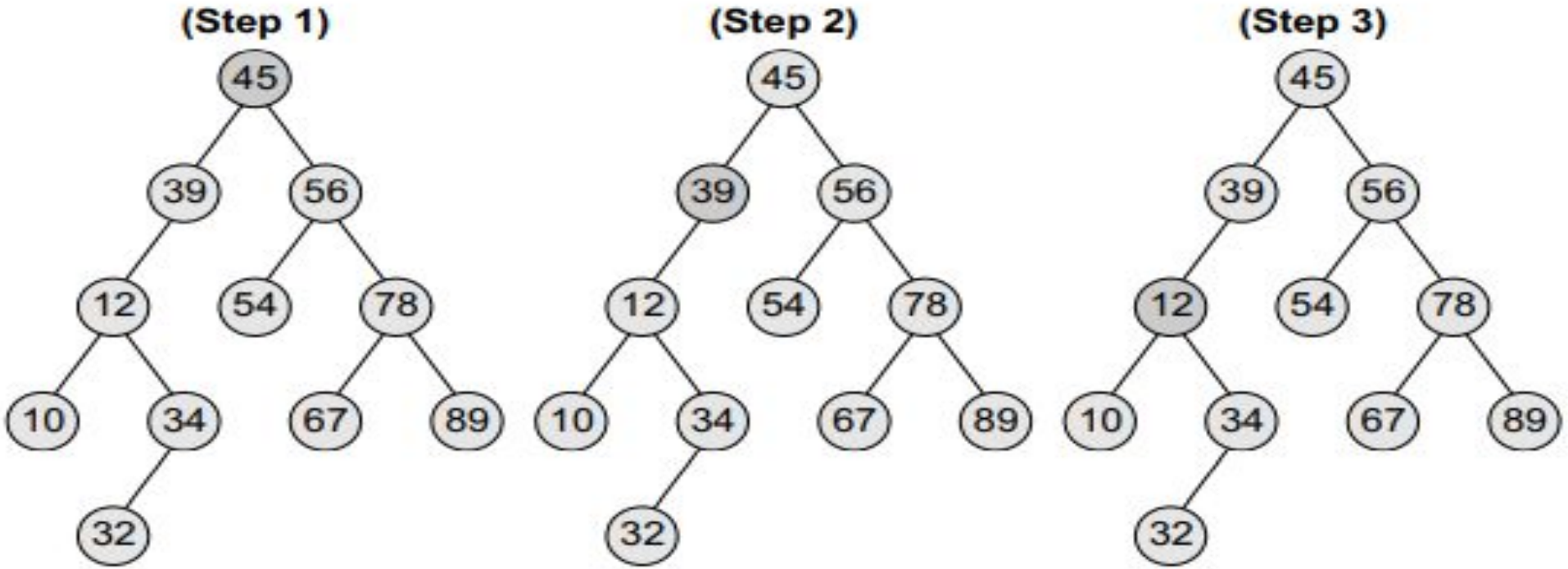
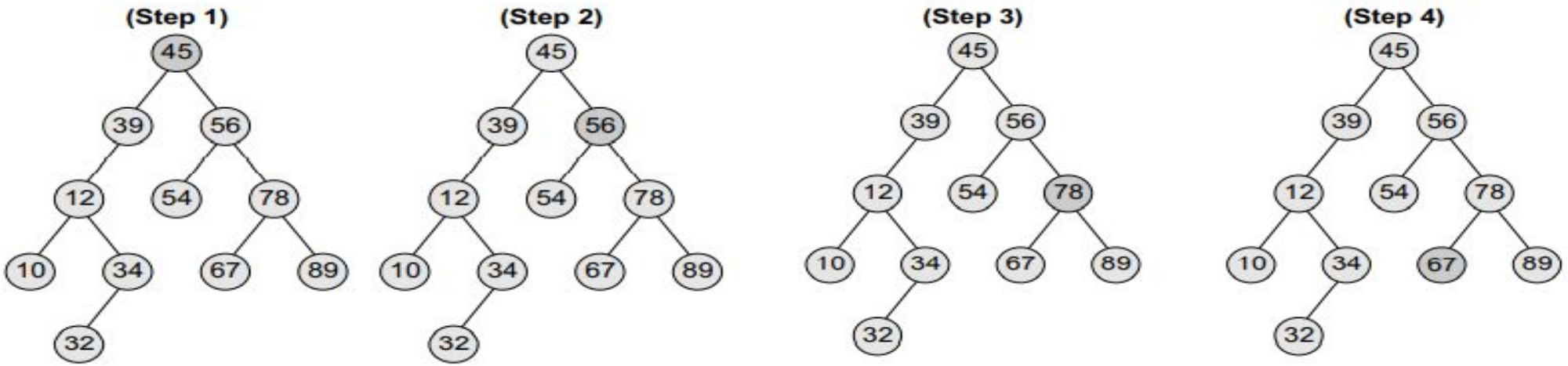


Figure 10.5 Searching a node with value 12 in the given binary search tree

Search in Binary Search tree(Example)



Searching a node with value 67 in the given binary search tree

Somaiya TRUST

Inserting a New Node in a 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

Inserting Node in Binary Search Tree

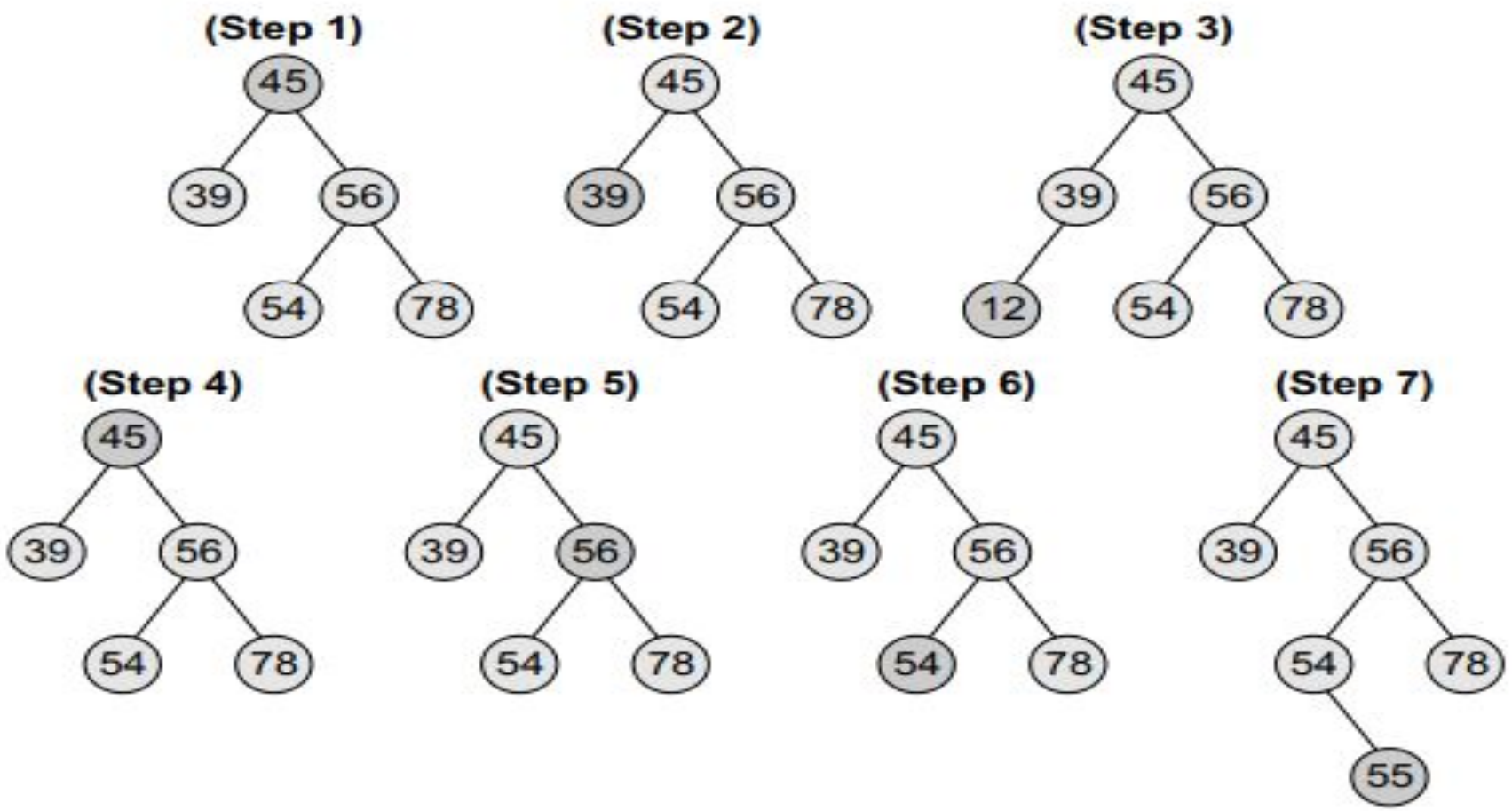


Figure 10.10 Inserting nodes with values 12 and 55 in the given binary search tree

Somaiya TRUST

Deleting Node in Binary Search Tree

- **Case 1: Deleting a Node that has No Children Look at the binary search tree given in Fig. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.**

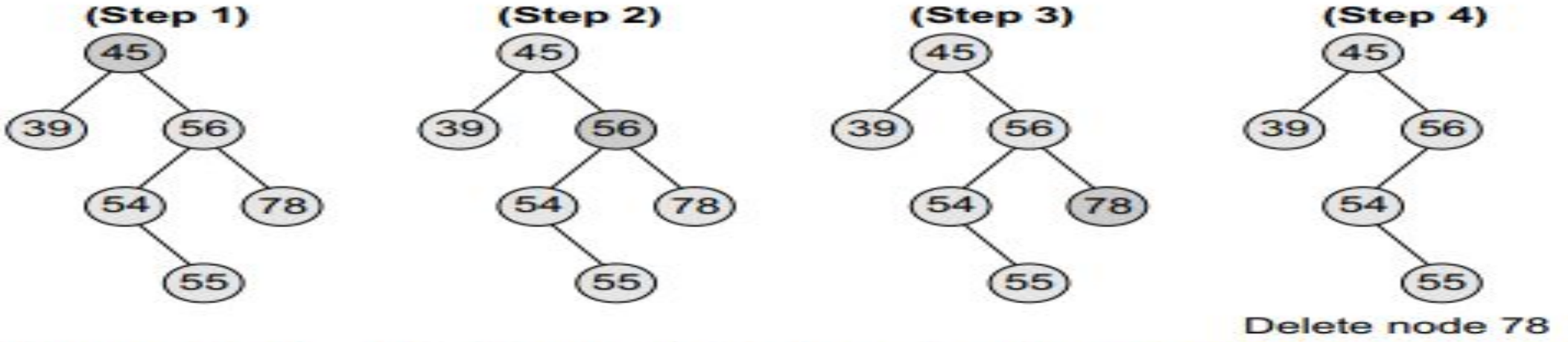


Figure 10.11 Deleting node 78 from the given binary search tree

Deleting Node in Binary Search Tree

- **Case 2: Deleting a Node with One Child 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.**

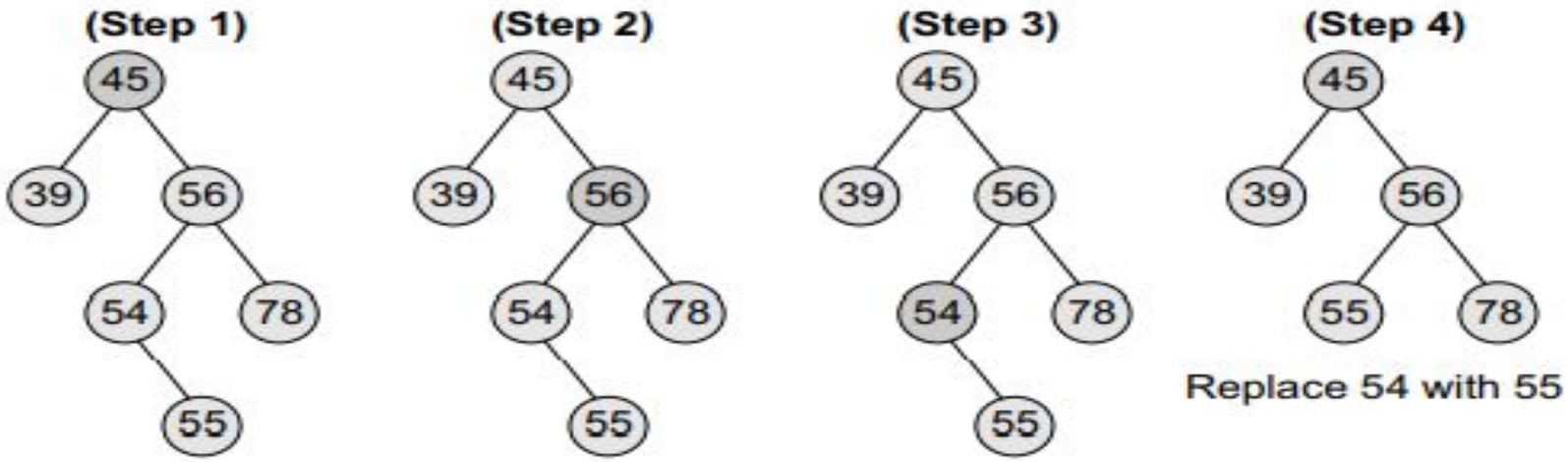


Figure 10.12 Deleting node 54 from the given binary search tree

Somaiya TRUST

The node to be deleted has two children.

Case 3: Deleting a Node with Two Children

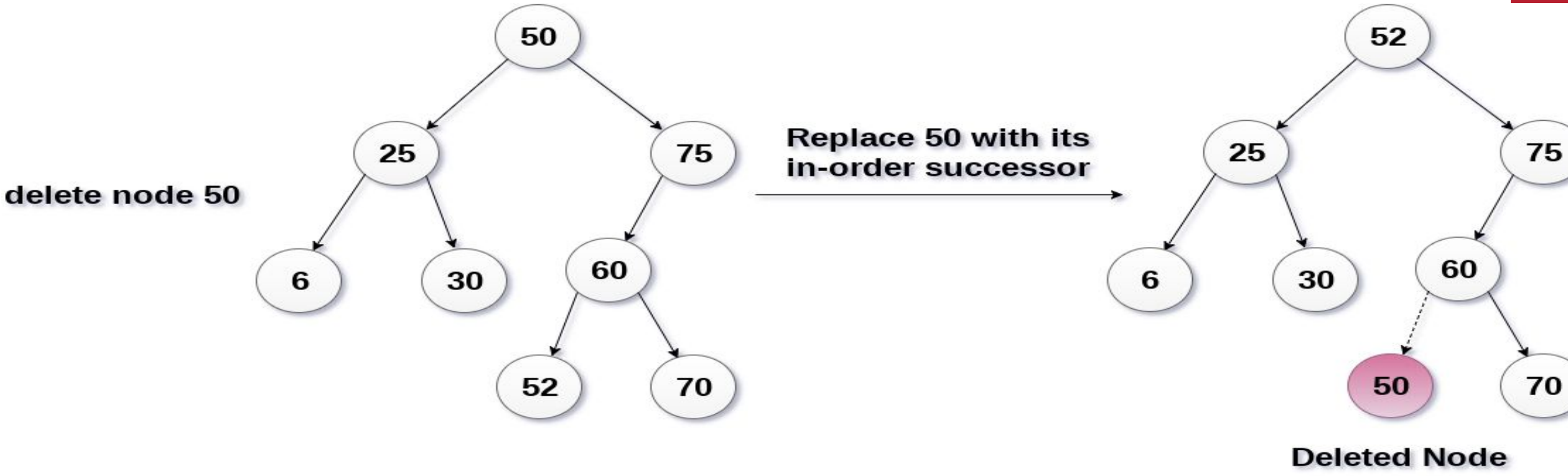
the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree.

After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Applications of Binary Tree

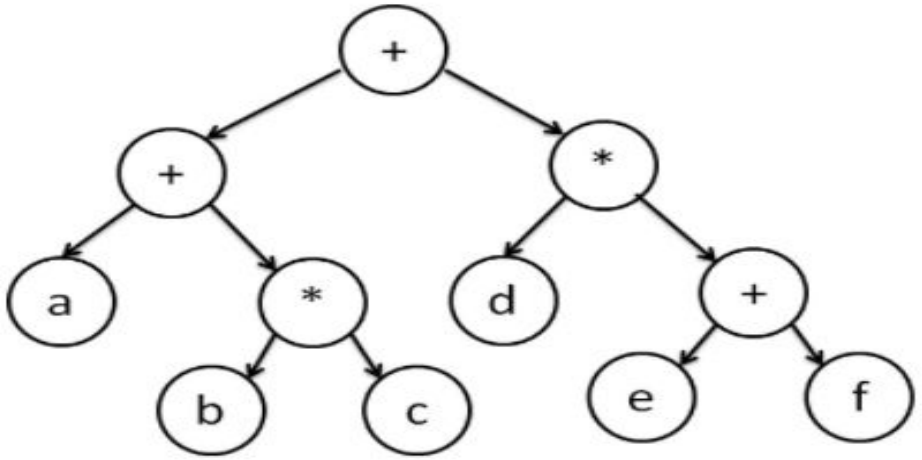
- **Applications of Binary Tree-**
  - Expression Tree
  - Huffman Encoding

Expression Tree

Expression Tree is used to represent expressions.

An expression and expression tree shown below

```
a + (b * c) + d * (e + f)
```



Expression Tree

Examples of Expression Tree

All the below are also expressions. Expressions may includes constants value as well as variables

```
a * 6
```

```
16
```

```
(a^2)+(b^2)+(2 * a * b)
```

```
(a/b) + (c)
```

```
m * (c ^ 2)
```

Somaiya TRUST

Huffman Coding-

- Huffman Coding is a famous Greedy Algorithm.
- It is used for the lossless compression of data.
- It uses variable length encoding.
- It assigns variable length code to all the characters.
- The code length of a character depends on how frequently it occurs in the given text.
- The character which occurs most frequently gets the smallest code.
- The character which occurs least frequently gets the largest code.
- It is also known as Huffman Encoding.

Major Steps in Huffman Coding-

- **There are two major steps in Huffman Coding-**
  - Building a Huffman Tree from the input characters.
  - Assigning code to the characters by traversing the Huffman Tree.

Somaiya TRUST

Huffman Tree

The steps involved in the construction of Huffman Tree are as follows-

Step-01:

Create a leaf node for each character of the text.

Leaf node of a character contains the occurring frequency of that character.

Step-02:

Arrange all the nodes in increasing order of their frequency value.

Step-03:

Considering the first two nodes having minimum frequency,

Create a new internal node.

The frequency of this new node is the sum of frequency of those two nodes.

Make the first node as a left child and the other node as a right child of the newly created node.

Step-04:

Keep repeating Step-02 and Step-03 until all the nodes form a single tree.

The tree finally obtained is the desired Huffman Tree.

Somaiya TRUST

Huffman Tree

Problem-

A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression, determine-

1. Huffman Code for each character
2. Average code length
3. Length of Huffman encoded message (in bits)

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Somaiya TRUST

Huffman Tree

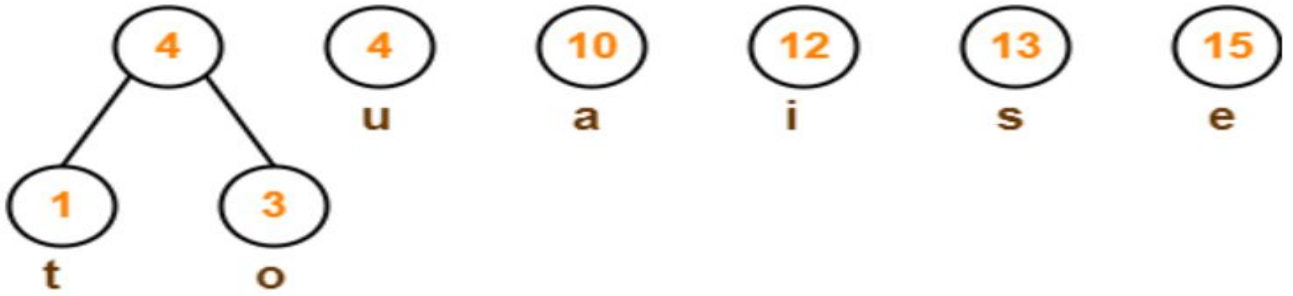
Solution-

First let us construct the Huffman Tree.

Huffman Tree is constructed in the following steps-

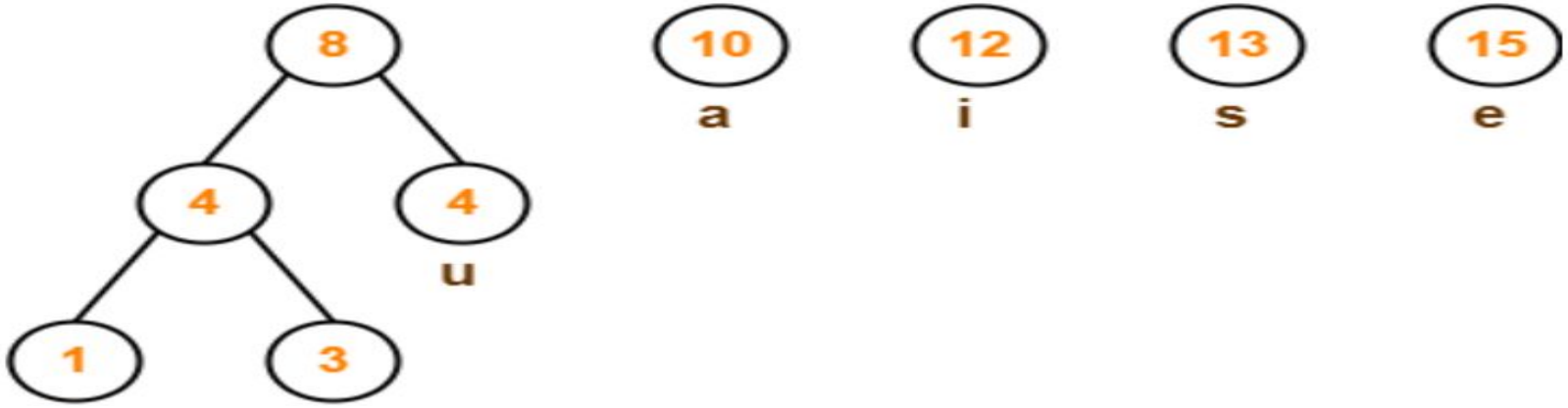
Step-01:

Step-02:

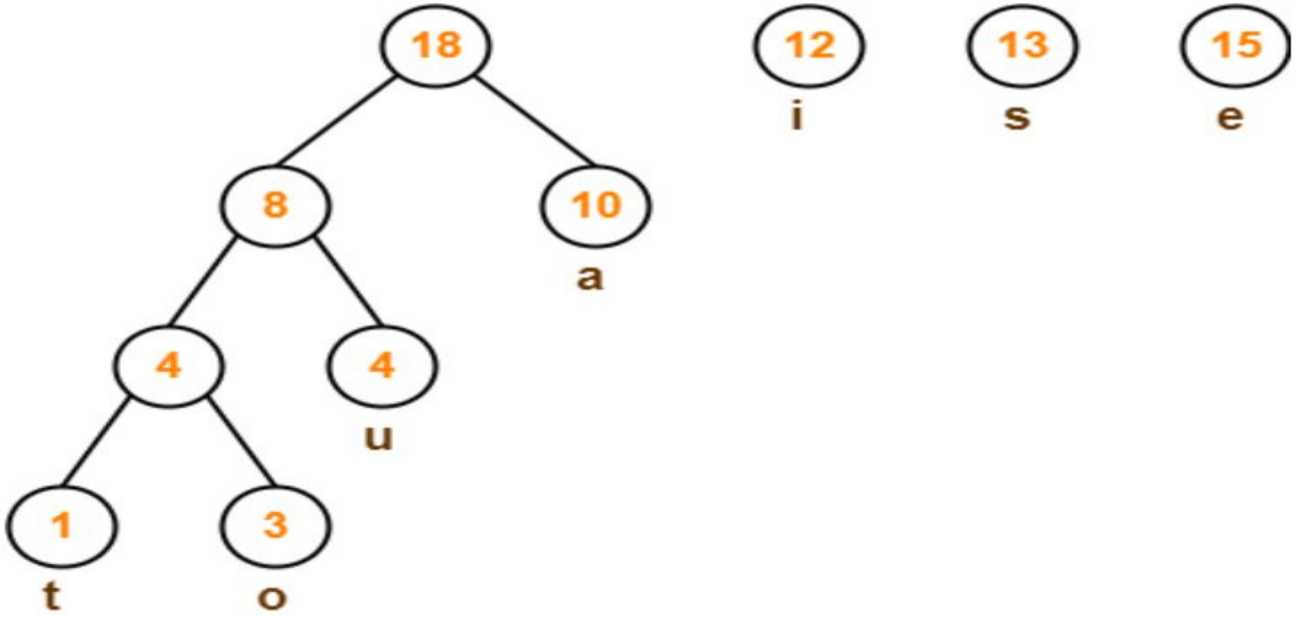


Somaiya TRUST

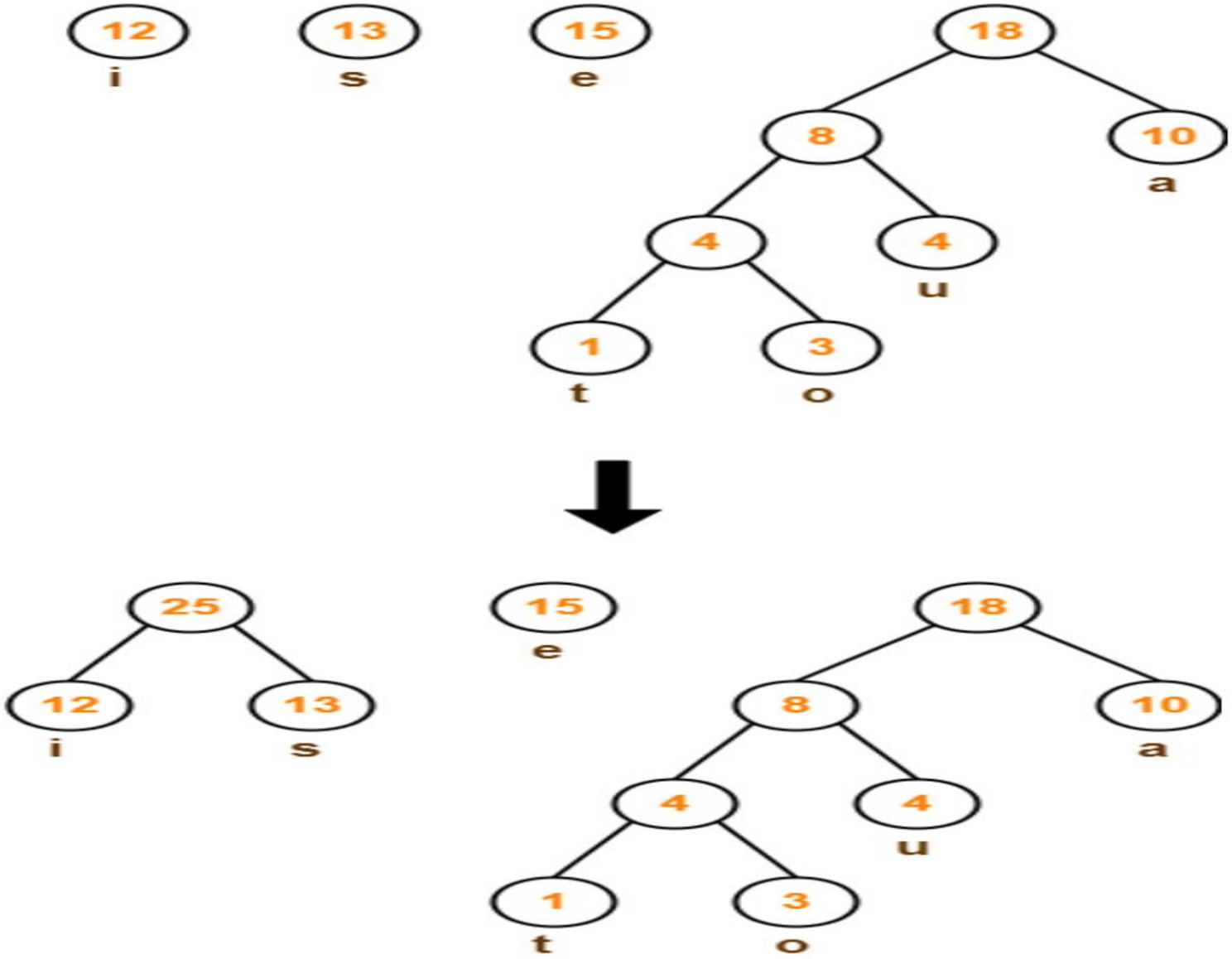
Step-03:



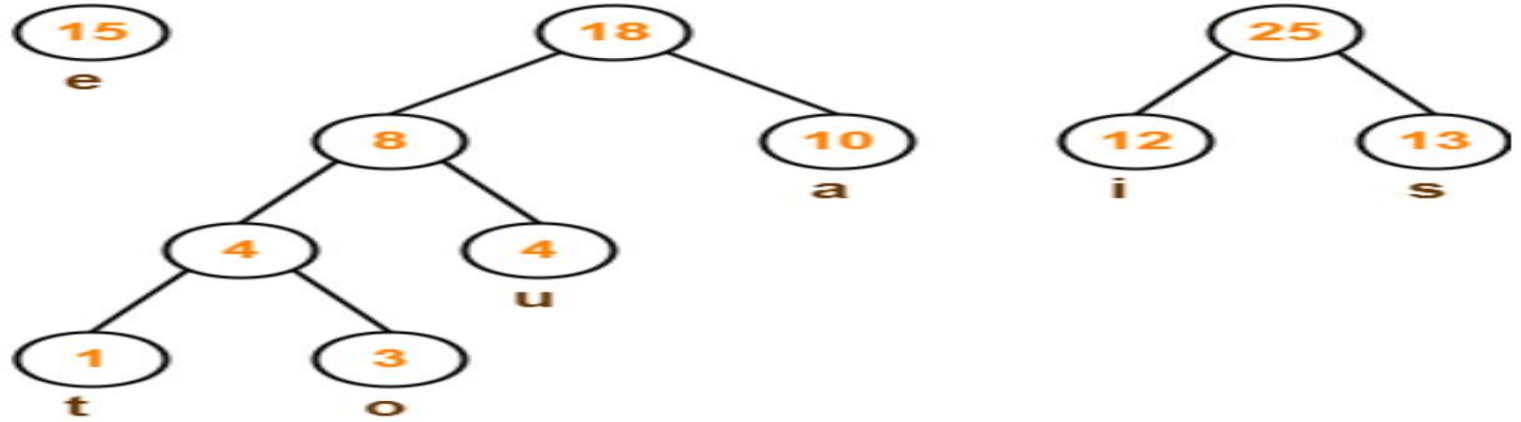
Step-04:

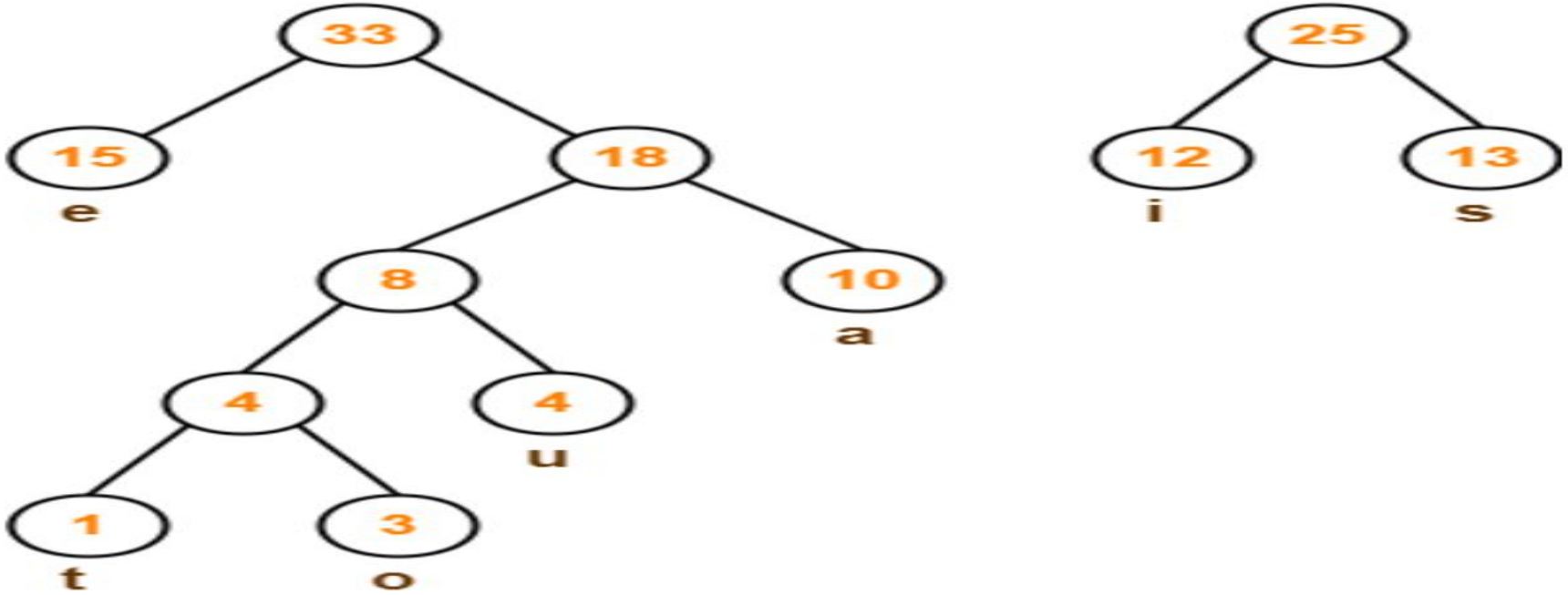


Step-05:

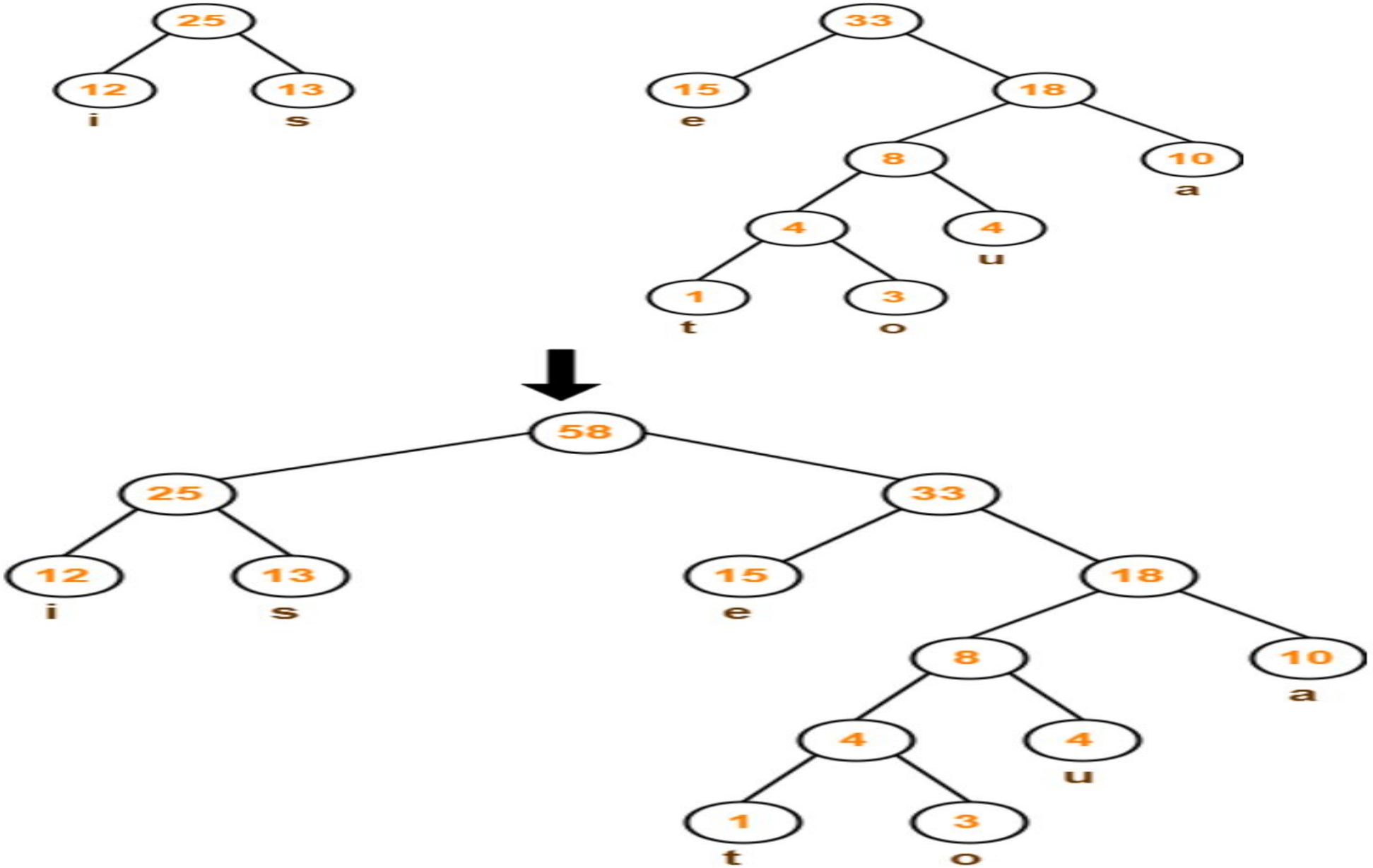


Step-06:


15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
13
i
s


33
15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
13
i
s

Step-07:



Huffman Tree

Now,

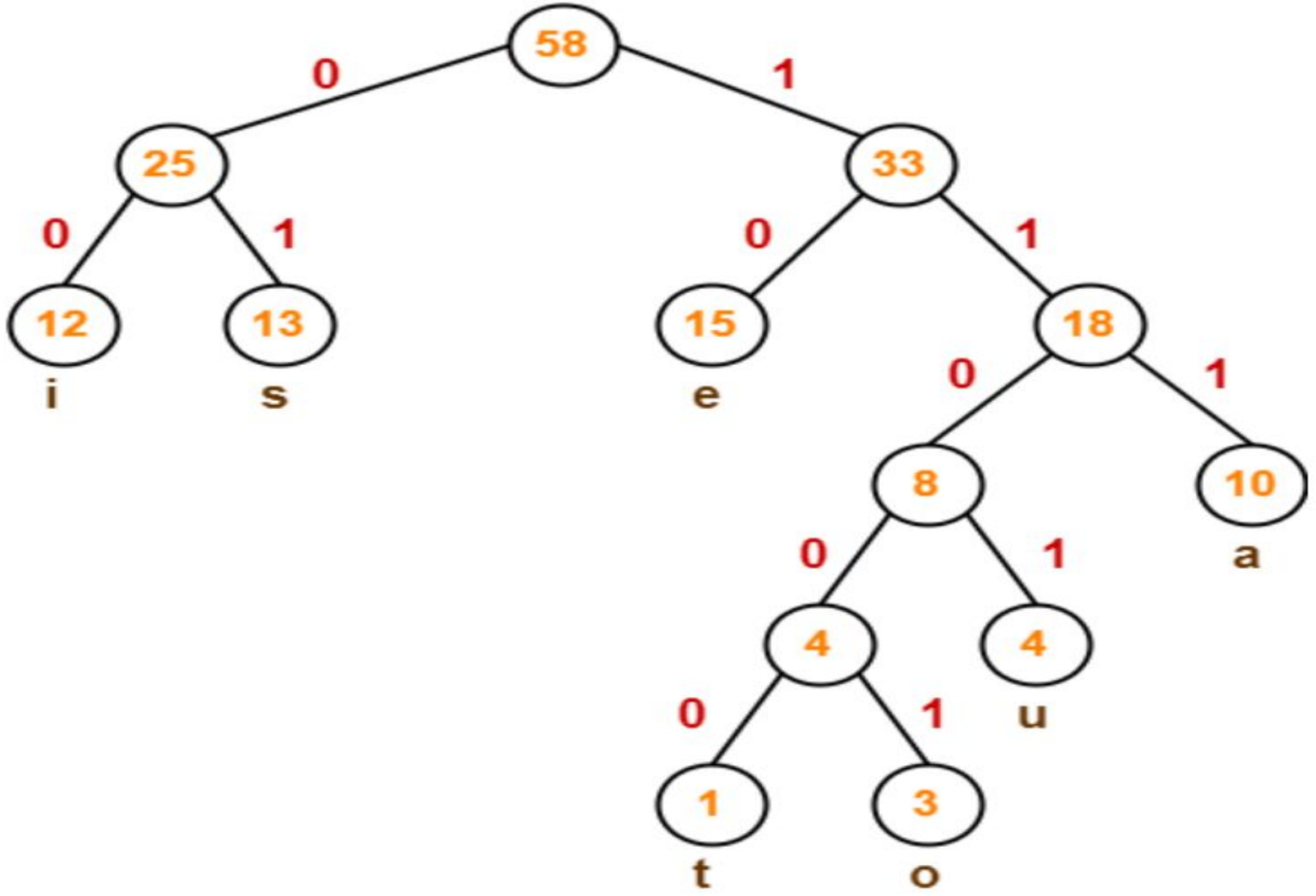
- We assign weight to all the edges of the constructed Huffman Tree.
- Let us assign weight ‘0’ to the left edges and weight ‘1’ to the right edges.

Rule

- If you assign weight ‘0’ to the left edges, then assign weight ‘1’ to the right edges.
- If you assign weight ‘1’ to the left edges, then assign weight ‘0’ to the right edges.
- Any of the above two conventions may be followed.
- But follow the same convention at the time of decoding that is adopted at the time of encoding.

Somaiya TRUST

After assigning weight to all the edges, the modified Huffman Tree is-



Huffman Tree

Now, let us answer each part of the given problem one by one-

1. Huffman Code For Characters-

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.

Following this rule, the Huffman Code for each character is-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

2. Average Code Length-

Using formula-01, we have-

Average code length

$= \sum (\text{frequency}_i \times \text{code length}_i) / \sum (\text{frequency}_i)$

$= \{ (10 \times 3) + (15 \times 2) + (12 \times 2) + (3 \times 5) + (4 \times 4) + (13 \times 2) + (1 \times 5) \} / (10 + 15 + 12 + 3 + 4 + 13 + 1)$

$= 2.52$

3. Length of Huffman Encoded Message-

Using formula-02, we have-

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

$= 58 \times 2.52$

$= 146.16$

$\cong$ 147 bits

AVL Trees

- **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.**

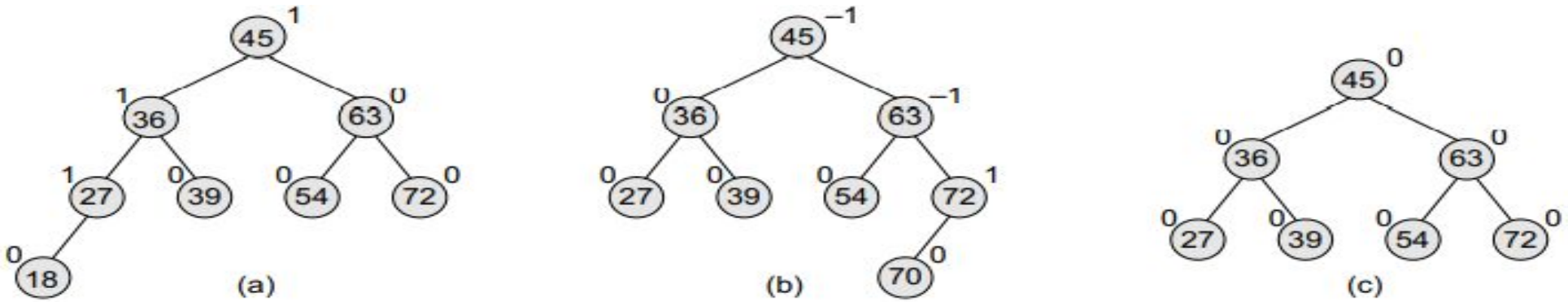
AVL Trees

- 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)

AVL 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 Tree



(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Somaiya TRUST

Operations on AVL Trees

- **Searching for a Node in an AVL Tree**
  - Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
  - Due to the height-balancing of the tree, the search operation takes O(log n) time to complete. Since the operation does not modify the structure of the tree, no special provisions are required.

Operations on AVL Trees

Inserting a New Node in an AVL Tree

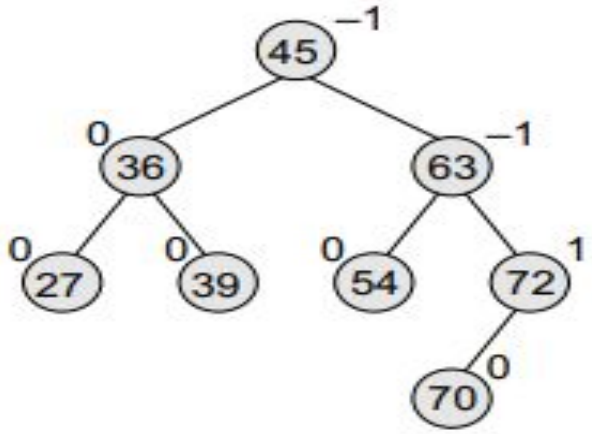


Figure 10.38 AVL tree

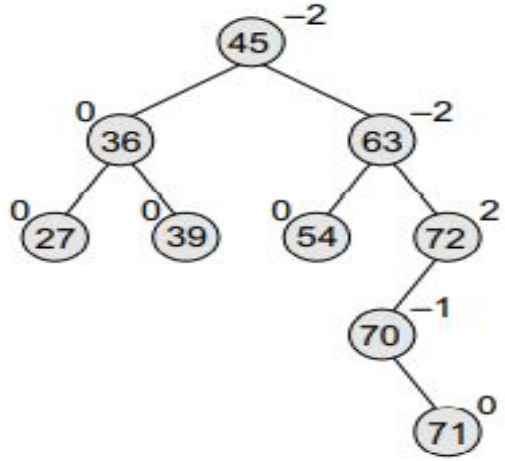


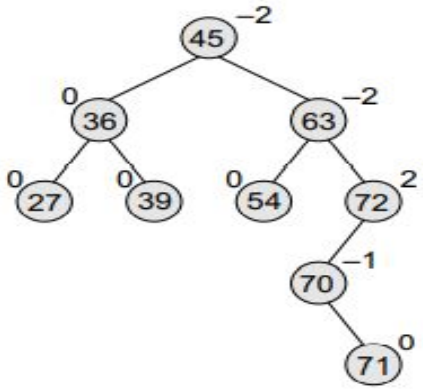
Figure 10.39 AVL tree after inserting a node with the value 71

Somaiya TRUST

Making a Tree Height Balanced

- **Note that there are three nodes in the tree that have their balance factors 2, –2, and –2, thereby disturbing the AVLness of the tree.**
- **So, here comes the need to perform rotation.**
- **To perform rotation, our first task is to find the critical node.**
- **Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither –1, 0, nor 1.**

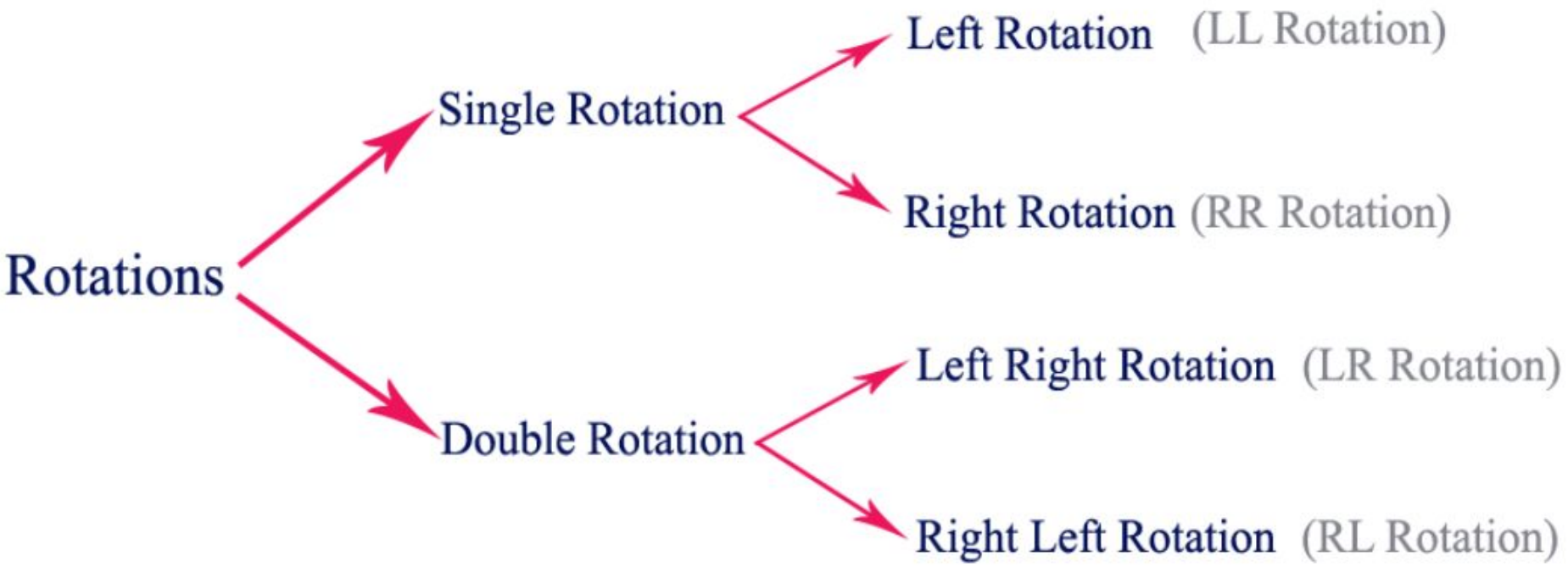
Making a Tree Height Balanced



AVL tree after inserting a node with the value 71

- In the tree given above, the critical node is 72
- The second task in rebalancing the tree is to determine which type of rotation has to be done.
- There are four types of rebalancing rotations and application of these rotations depends on the position of the inserted node with reference to the critical node.
- 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.

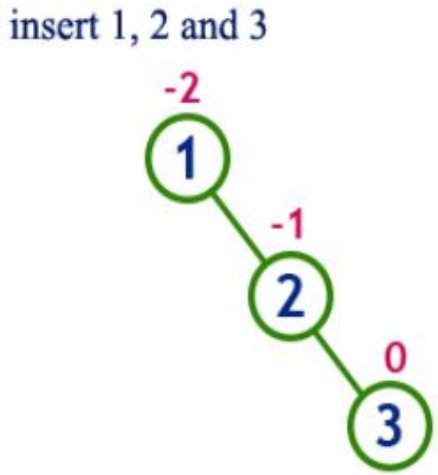
Somaiya TRUST



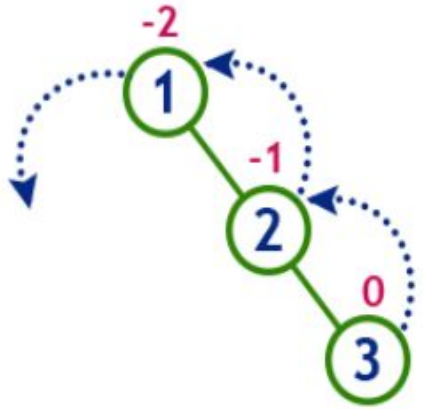
Somaiya TRUST

Single Left Rotation (LL Rotation)

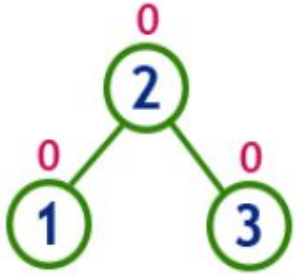
In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left



After LL Rotation Tree is Balanced

Somaiya TRUST

RR-Rotation Example

Example 10.3 Consider the AVL tree given in Fig. 10.41 and insert 18 into it.

Solution

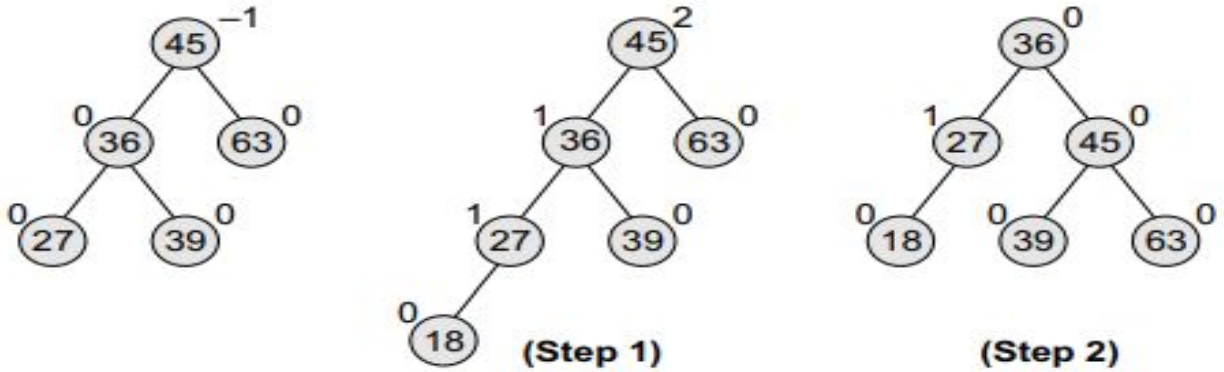


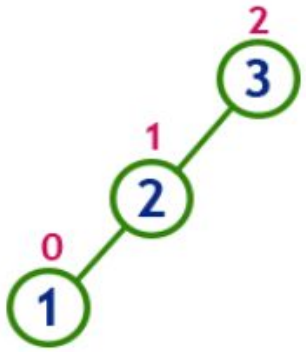
Figure 10.41 AVL tree

Somaiya TRUST

Single Right Rotation (RR Rotation)

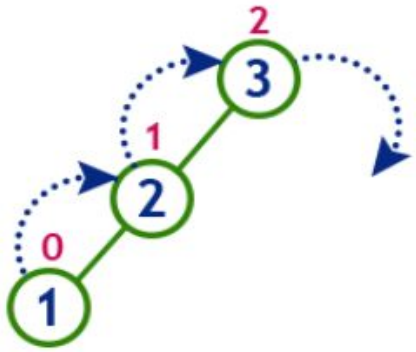
In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

insert 3, 2 and 1

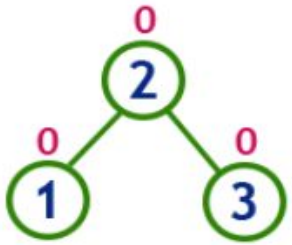


Tree is imbalanced

because node 3 has balance factor 2



To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

Somaiya TRUST

RR-Rotation example

Example 10.4 Consider the AVL tree given in Fig. 10.43 and insert 89 into it.

Solution

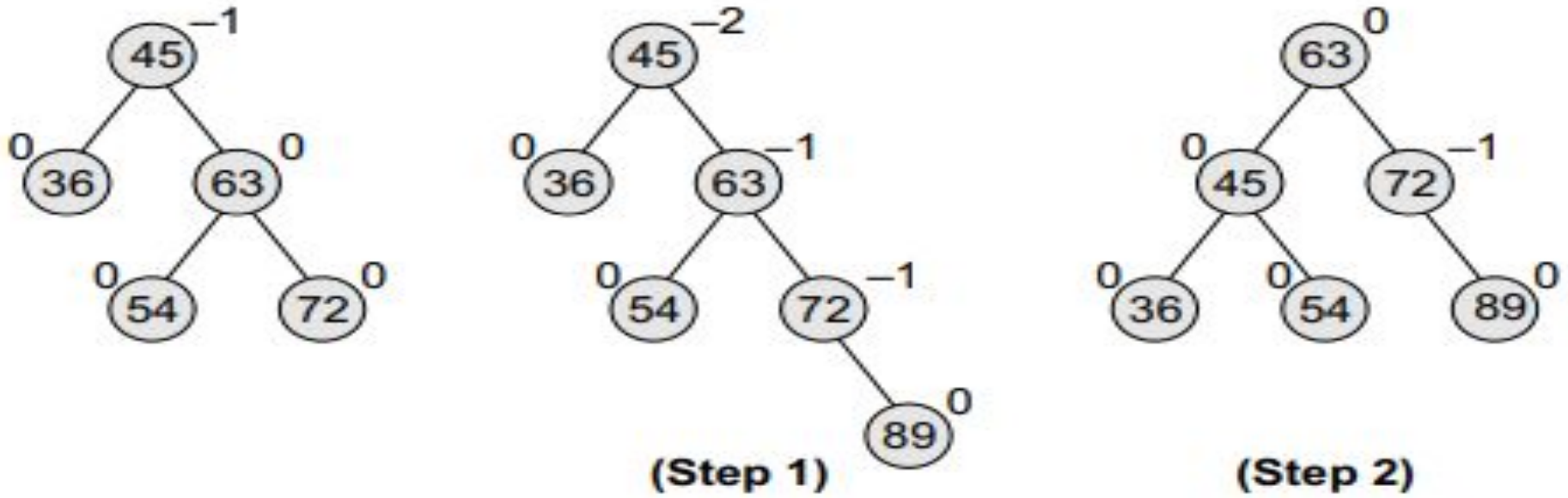
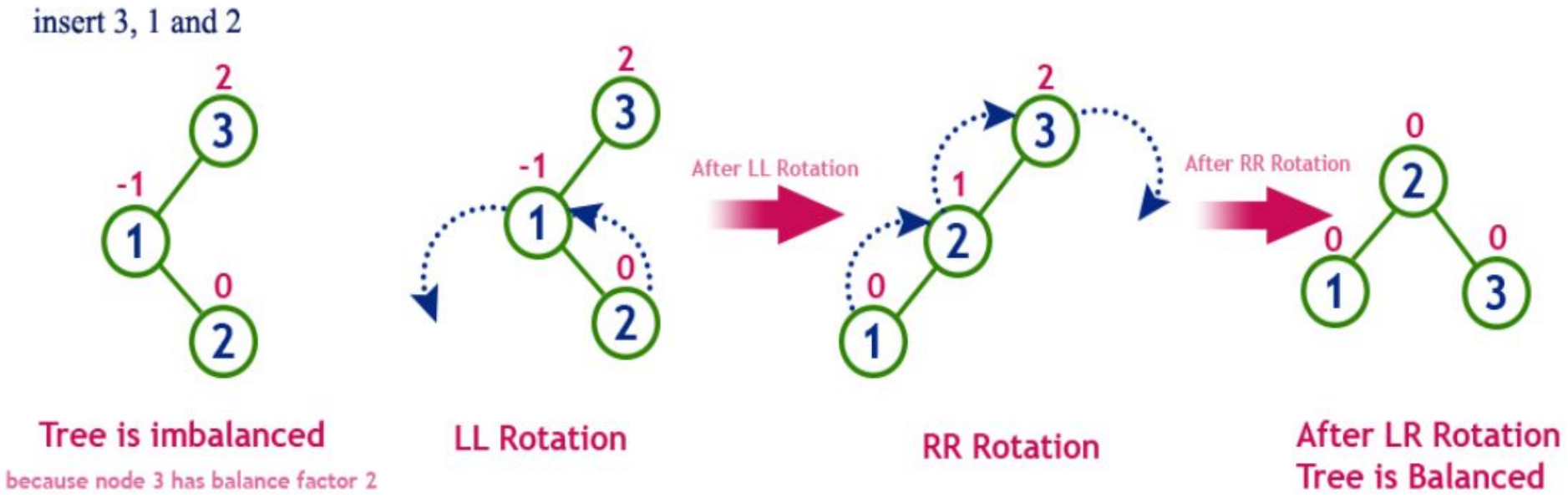


Figure 10.43 AVL tree

Somaiya TRUST

Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Example LR-Rotation

Example 10.5 Consider the AVL tree given in Fig. 10.45 and insert 37 into it.

Solution

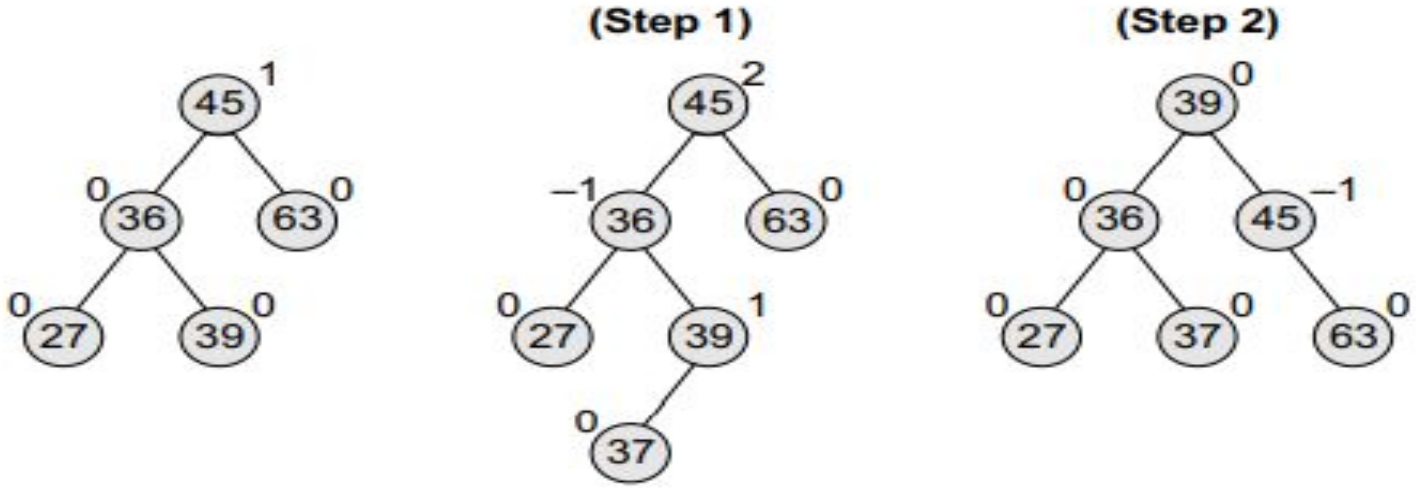
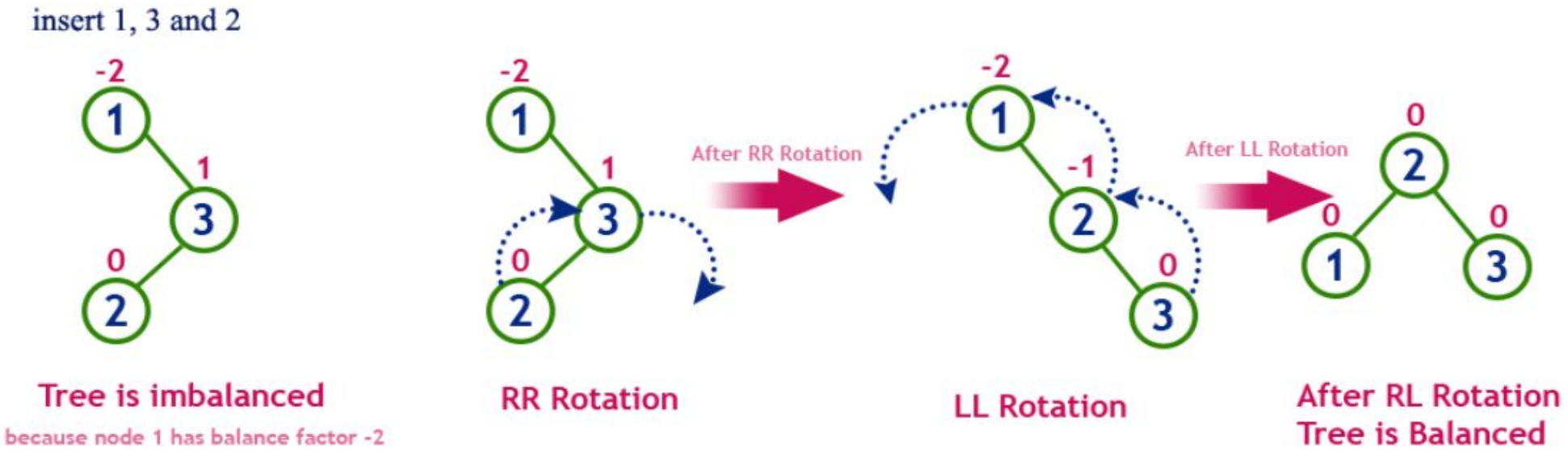


Figure 10.45 AVL tree

Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the **Balance Factor** of every node.

Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.

Somaiya TRUST

insert 1

Tree is balanced

insert 2

Tree is balanced

insert 3

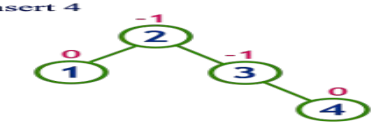
Tree is imbalanced

LL Rotation

After LL Rotation

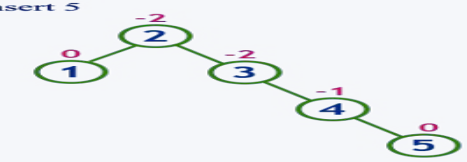
Tree is balanced

insert 4



Tree is balanced

insert 5



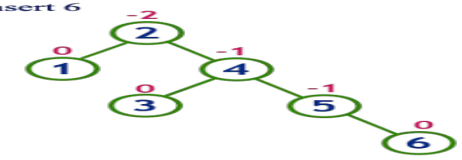
Tree is imbalanced

LL Rotation at 3

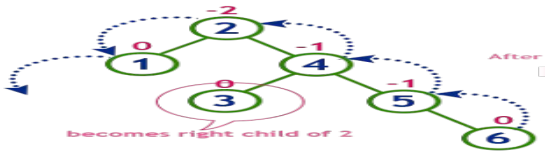
After LL Rotation at 3

Tree is balanced

insert 6



Tree is imbalanced



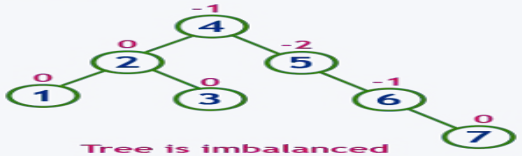
becomes right child of 2

LL Rotation at 2

After LL Rotation at 2

Tree is balanced

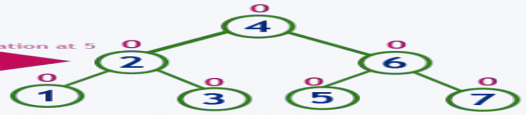
insert 7



Tree is imbalanced

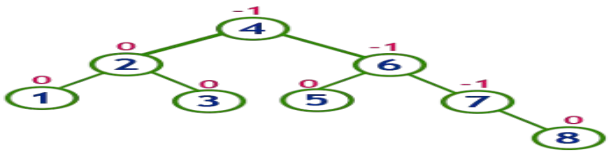
LL Rotation at 5

After LL Rotation at 5



Tree is balanced

insert 8

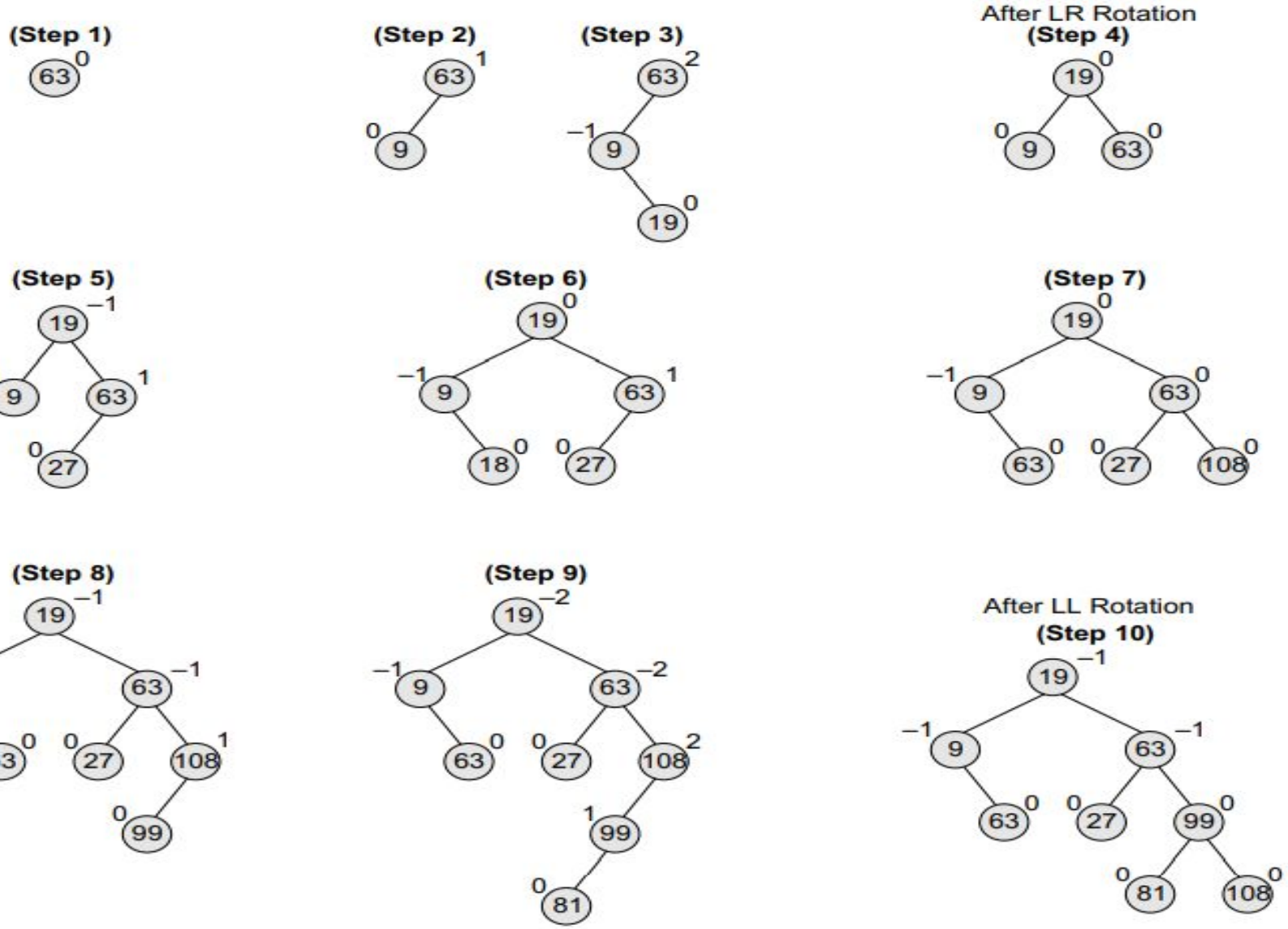


Tree is balanced

Somaiya TRUST

Example 10.6 Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

Solution



AVL tree

B - Tree Data structure

- In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children.
- But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children.
- B-Tree was developed in the year 1972 by Bayer and McCreight with the name *Height Balanced m-way Search Tree*.
- Later it was named as B-Tree.

Somaiya TRUST

B-Tree

- **B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.**
- **Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.**

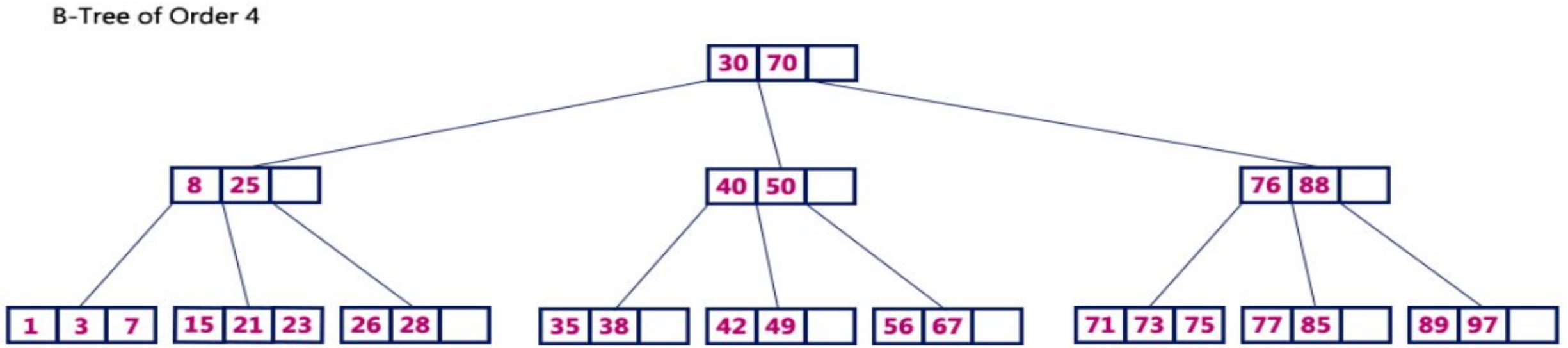
B-Tree of Order m has the following properties...

- **Property #1** - All **leaf nodes** must be **at same level**.
- **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **Property #4** - If the root node is a non leaf node, then it must have **atleast 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order**.

Somaiya TRUST

B-Tree

- **For example, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.**



Somaiya TRUST

Operations on B-Tree

The following operations are performed on a B-Tree...

- **Search**
- **Insertion**
- **Deletion**

Somaiya TRUST

Search Operation in B-Tree

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

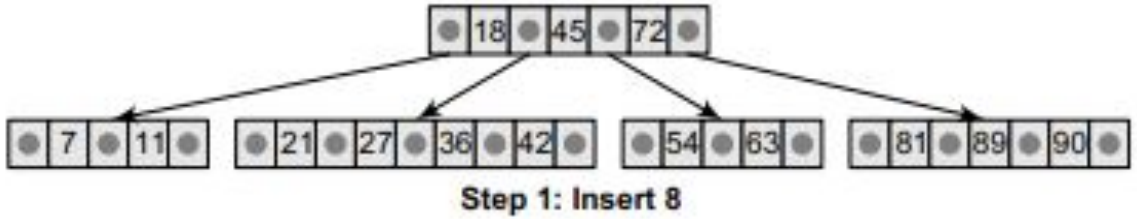
Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

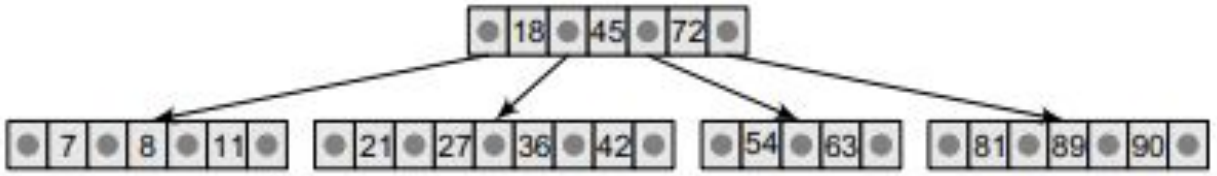
Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

B tree Insertion

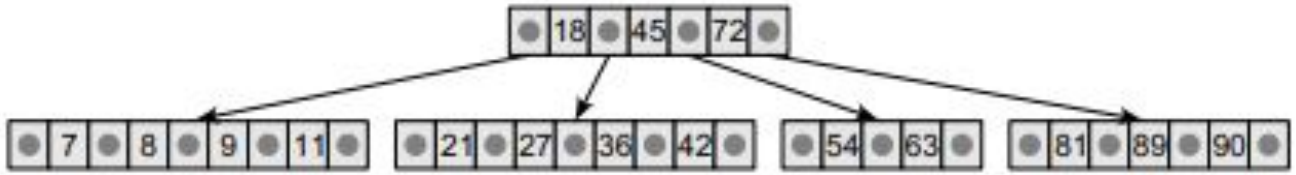
Example 11.1 Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.



Step 1: Insert 8



Step 2: Insert 9



Somaiya TRUST

B tree Insertion

Till now, we have easily inserted 8 and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42. The median value is 36, so push 36 into its parent's node and split the leaf nodes.

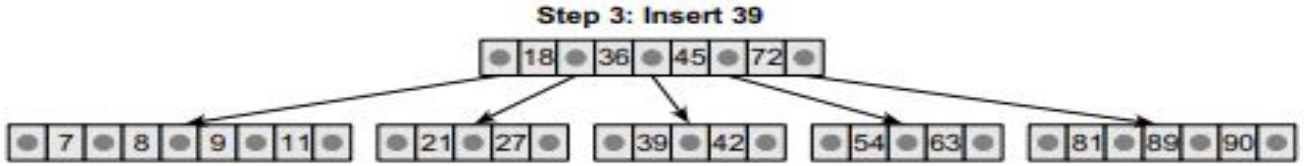


Figure 11.5(b)

Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.

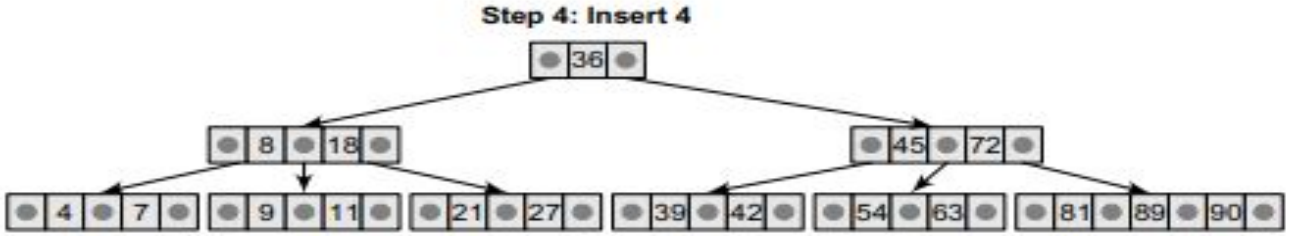


Figure 11.5(c) B tree

B tree Deletion

11.2.3 Deleting an Element from a B Tree

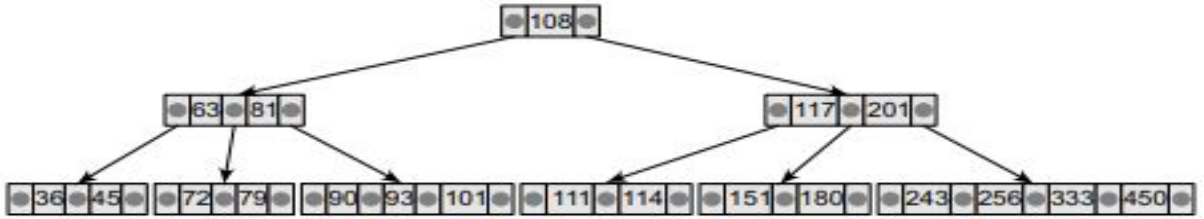
Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted.
2. If the leaf node contains more than the minimum number of key values (more than `m/2` elements), then delete the value.
3. Else if the leaf node does not contain `m/2` elements, then fill the node by taking an element either from the left or from the right sibling.
   (a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
   (b) Else, if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is, `m`). If pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B tree.

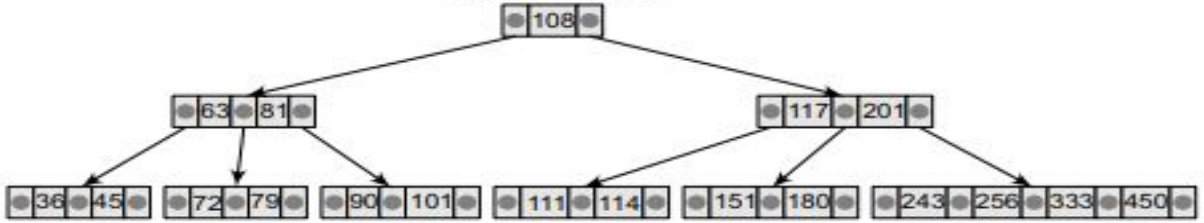
To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

B tree Deletion

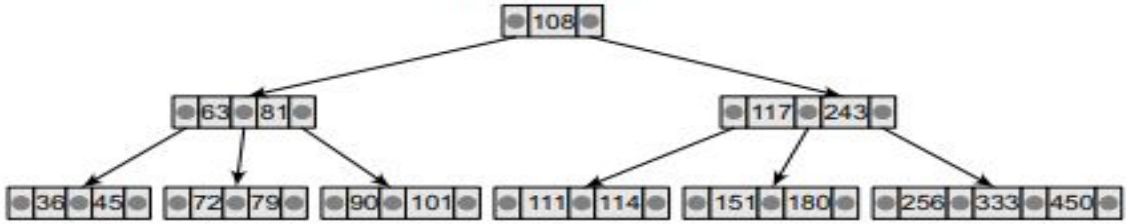
Example 11.2 Consider the following B tree of order 5 and delete values 93, 201, 180, and 72 from it (Fig. 11.6(a)).



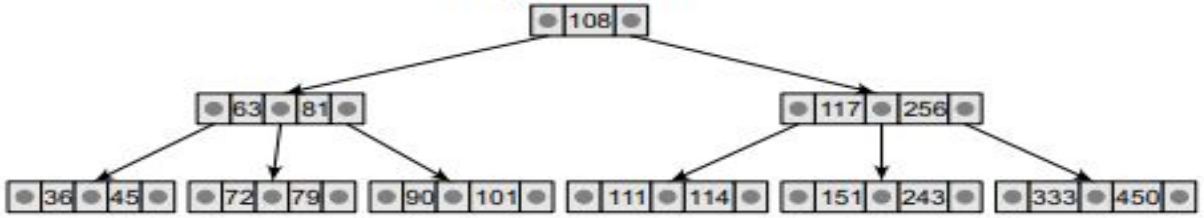
Step 1: Delete 93



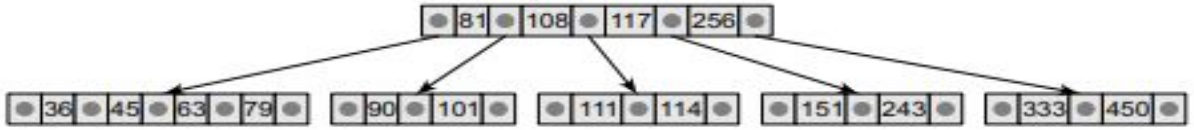
Step 2: Delete 201



Step 3: Delete 180



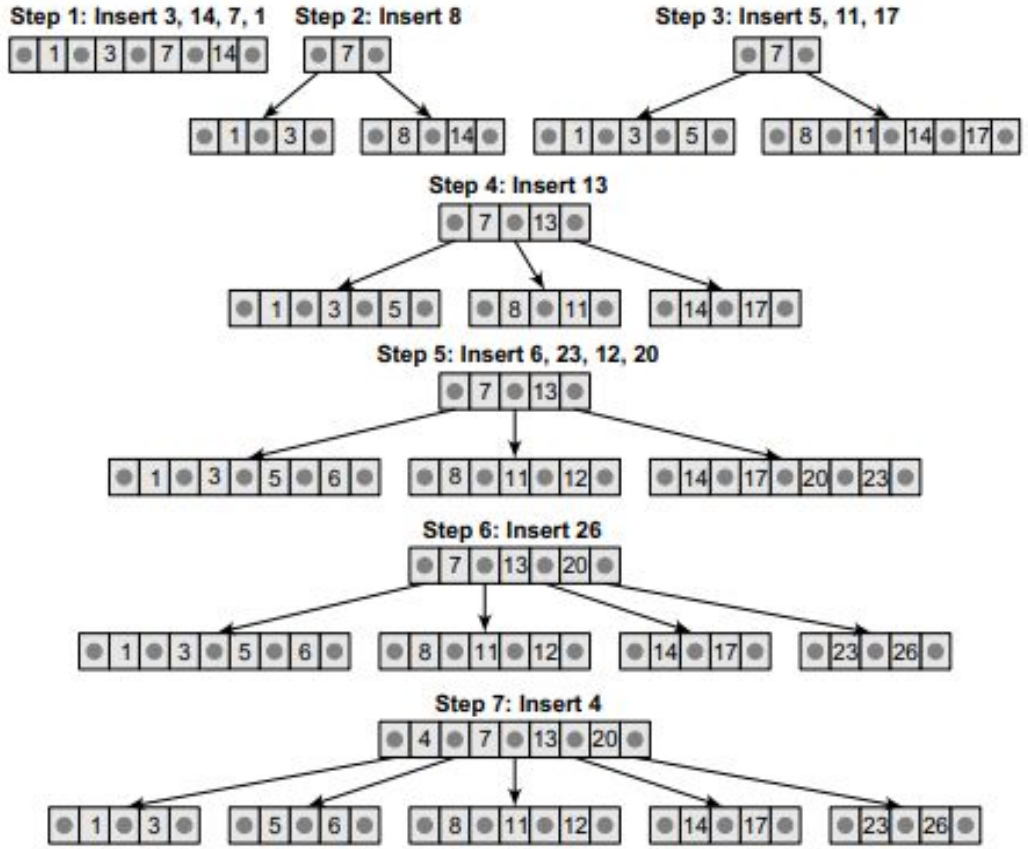
Step 4: Delete 72



Somaiya TRUST

B tree Insertion and Deletion

Example 11.4 Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.



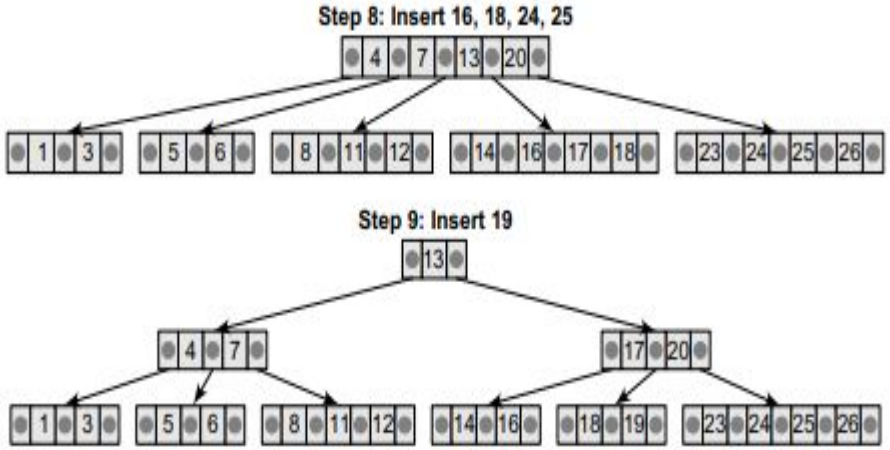


Figure 11.8 B tree

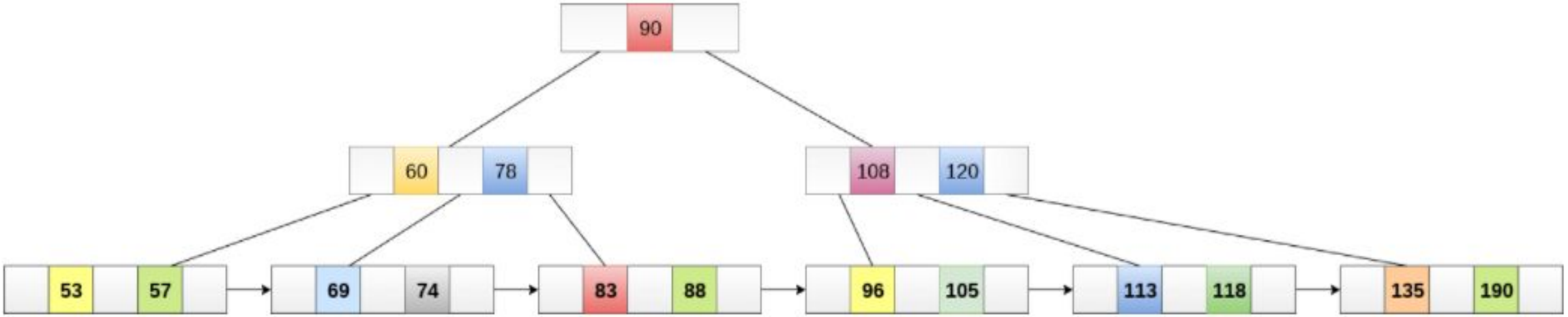
Somaiya TRUST

B+ Tree

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Tree are used to store the large amount of data which can not be stored in the main memory.
- Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

B+ Tree

- **The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.**



Somaiya TRUST

Advantages of B+ Tree

- **Records can be fetched in equal number of disk accesses.**
- **Height of the tree remains balanced and less as compare to B tree.**
- **We can access the data stored in a B+ tree sequentially as well as directly.**
- **Keys are used for indexing.**
- **Faster search queries as the data is stored only on the leaf nodes.**

Somaiya TRUST

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

SAMPLE QUESTIONS

QUESTION NO.	SAMPLE QUESTIONS MODULE 4	
1	Explain Binary search tree and its operations	
2	Construct binary search tree for following elements 23,45,78,15,12,3,6,9,49,99,56,88,11,14,41 perform deletion of 44 and 14	
3	Explain expression tree. **ALSO STUDY HOW TO CONSTRUCT EXPRESSION TREE FROM A GIVEN EXP**	
4	Construct AVL tree for the following elements. 20,24,30,15,18,32,40.	
5	STUDY PROBLEMS ON INORDER/PREORDER/POST ORDER TRAVERSAL	
6	Create a Huffman tree with the following data A 7 B 9 C 11 D 14 E 18 F 21 G 27 H 29 I 35 J 40	
7	Construct B tree for following elements 3,14,7,1,8,5,11,17,13,6,23,12,20,26,4,16,18,24,25,19 delete 24, 20, 5	

Somaiya TRUST