Chapter 05
Tree

Definition:

A non-linear data structure called tree. A tree is a structure which is mainly used to store data that is hierarchical in nature.

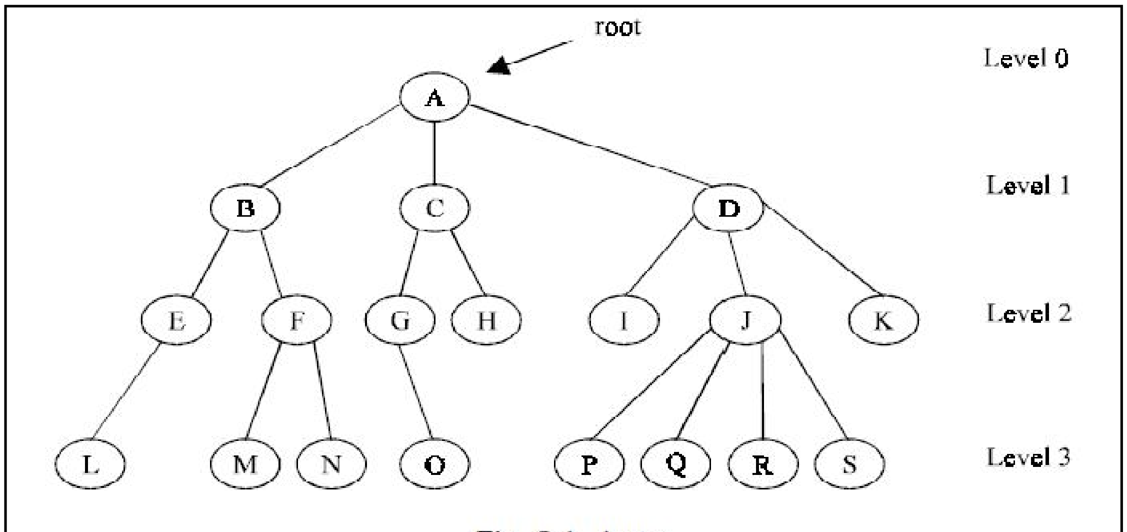


Fig. 8.1. A tree

01 Tree Terminology:

- **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(Parent 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(Child 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 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.

- **Depth:** A tree is the maximum level of any node in a given tree. This is number of level one can descend the tree from its root node to the terminal nodes (leaves).

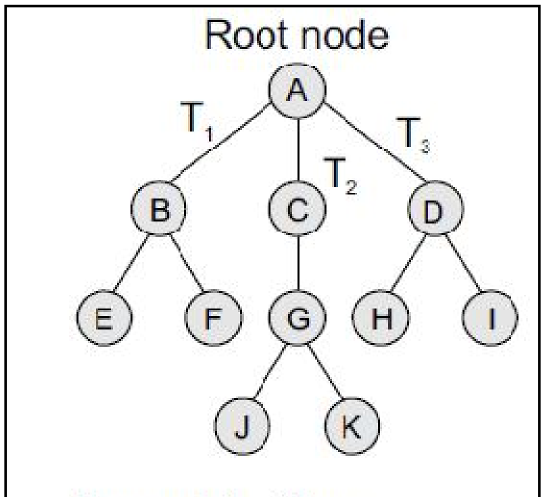


Figure 9.1 Tree

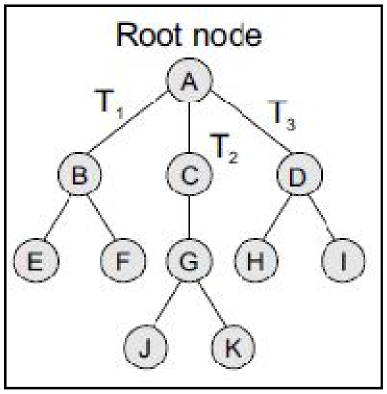
TYPES OF TREES:

Trees are of following 6 types:

1. General trees
2. Forests
3. Binary trees
4. Binary search trees
5. Expression trees

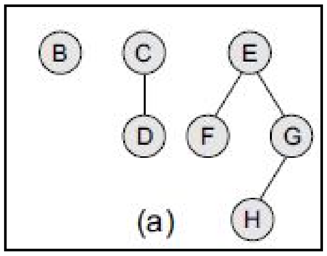
General Trees:

General trees are data structures that store elements hierarchically. The top node of a tree is the root node and each node, except the root, has a parent. A node in a general tree (except the leaf nodes) may have zero or more sub-trees.



Forests:

A forest is a disjoint union of trees. A set of disjoint trees (or forests) is obtained by deleting the root and the edges connecting the root node to nodes at level 1.



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.

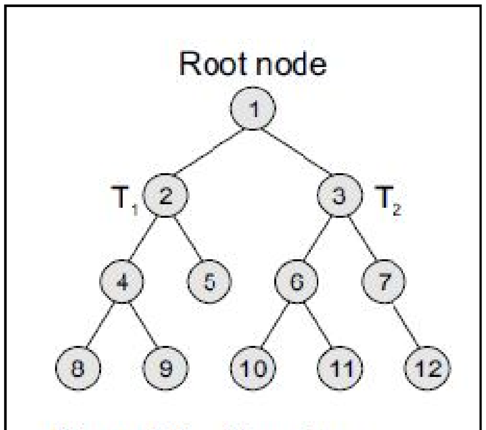


Figure 9.3 Binary tree

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.

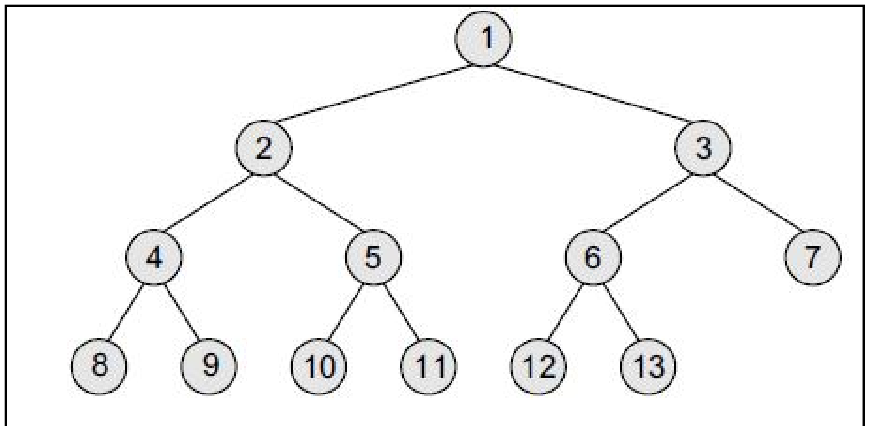


Figure 9.7 Complete binary tree

Binary Search Trees

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

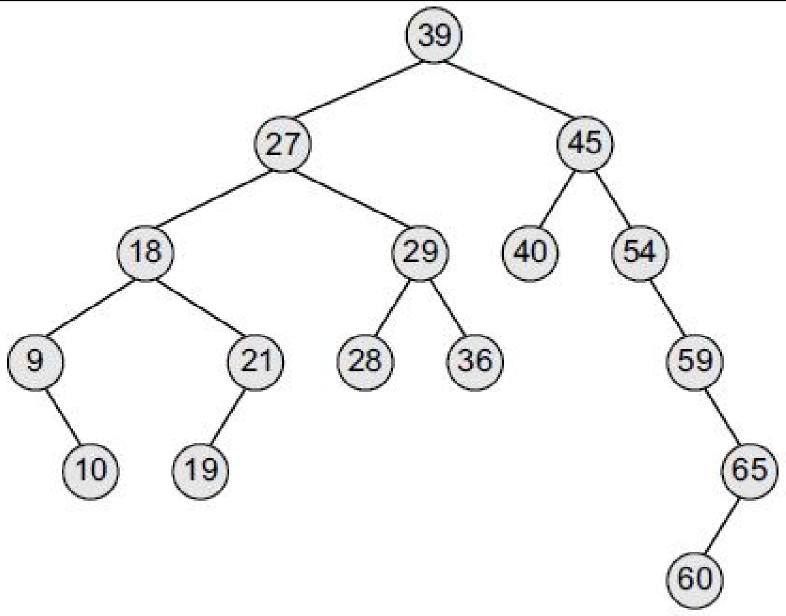


Figure 10.1 Binary search tree

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 Fig. 9.13.

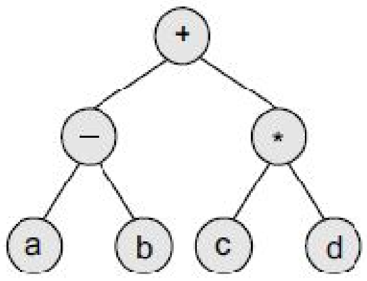


Figure 9.13 Expression tree

02 Tree Representation:

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. 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. 9.3. The schematic diagram of the linked representation of the binary tree is shown in Fig. 9.9.

In Fig. 9.9, 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).

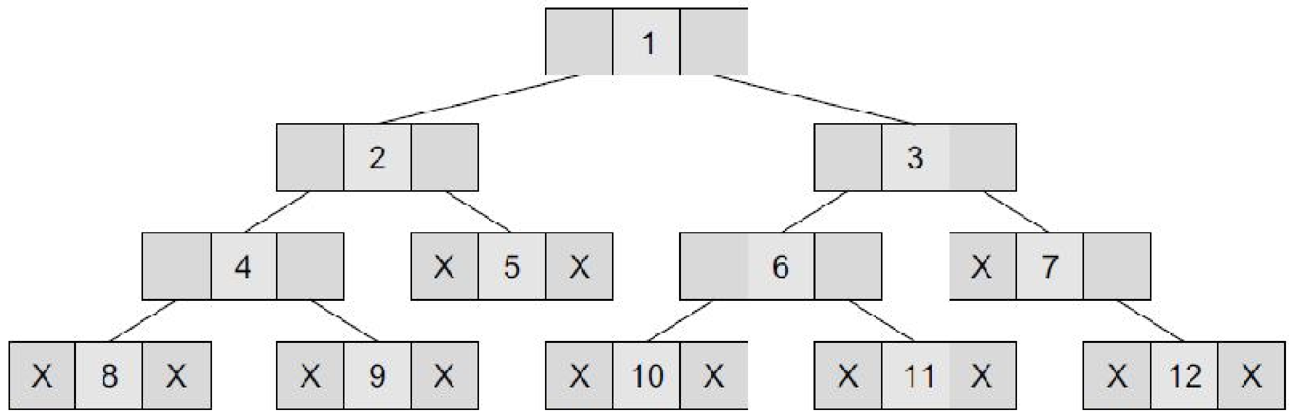


Figure 9.9 Linked representation of a binary tree

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

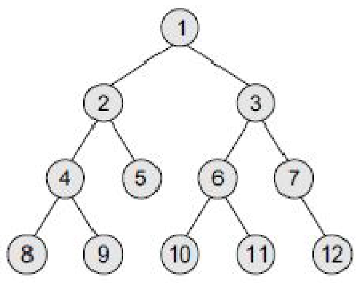
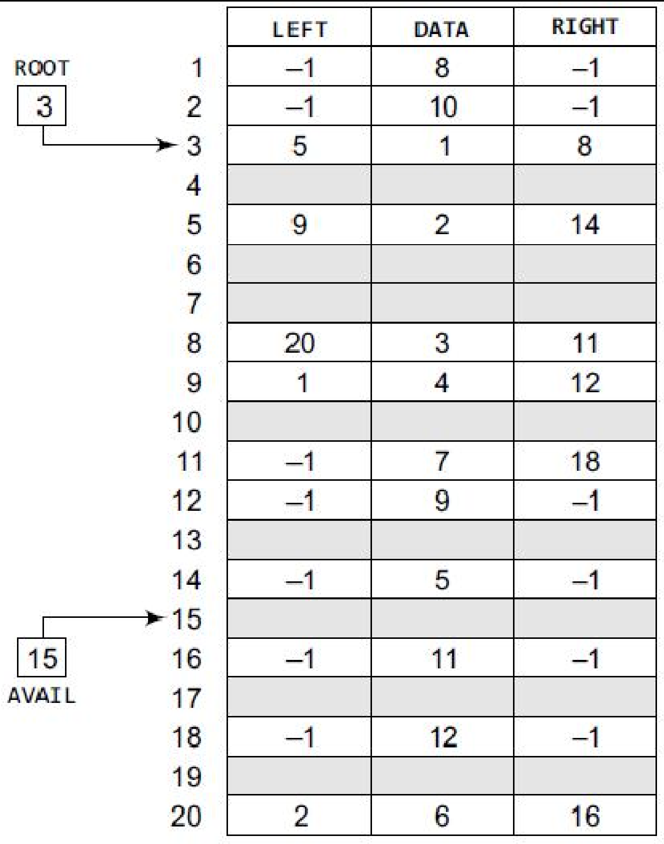


Figure 9.10 Binary tree T


ROOT
3
15
AVAIL

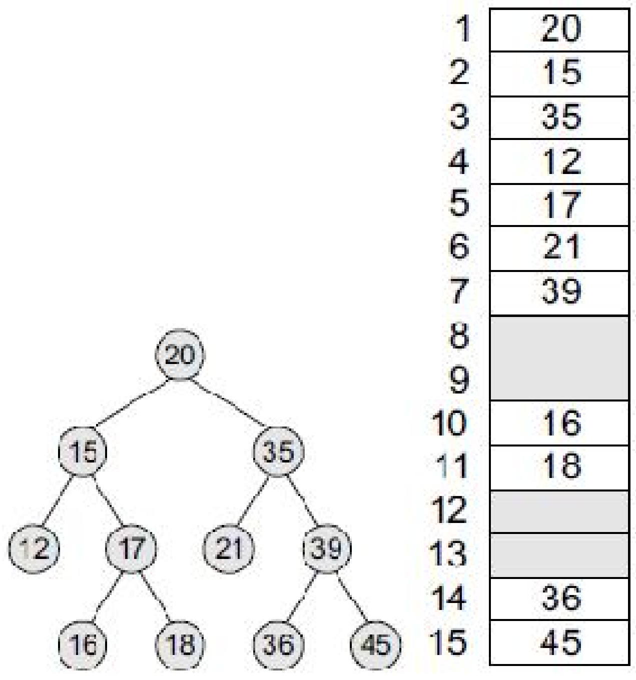
	LEFT	DATA	RIGHT
1	–1	8	–1
2	–1	10	–1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	–1	7	18
12	–1	9	–1
13			
14	–1	5	–1
15			
16	–1	11	–1
17			
18	–1	12	–1
19			
20	2	6	16

Figure 9.11 Linked representation of binary tree T

Sequential representation of binary trees Sequential representation of trees is done using single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space. 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 9.12 shows a binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4.



1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	▨
9	▨
10	16
11	18
12	▨
13	▨
14	36
15	45

Figure 9.12 Binary tree and its sequential representation

03 Tree Traversal methods:

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 nonlinear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals.

1 Pre-order Traversal
2 In-order Traversal
3 Post-order Traversal

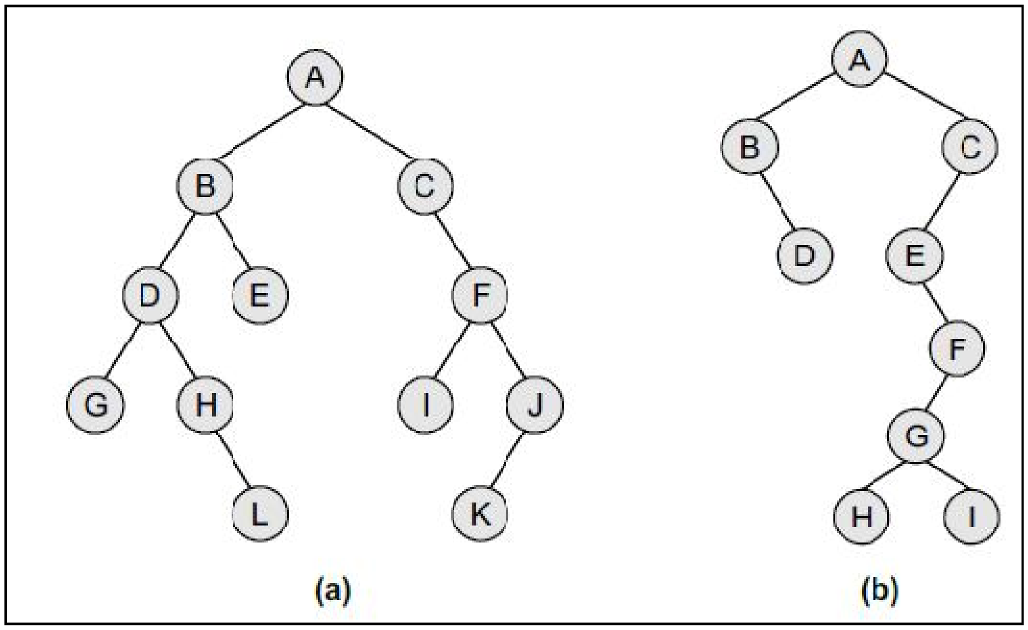
1 Pre-order Traversal:

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.

Algorithm: PREORDER()
Input: ROOT
Output: NODE_VISIT_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
```



Example In Figs (a) and (b), find the sequence of nodes that will be visited using pre-order traversal algorithm.

Solution

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

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

2 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:

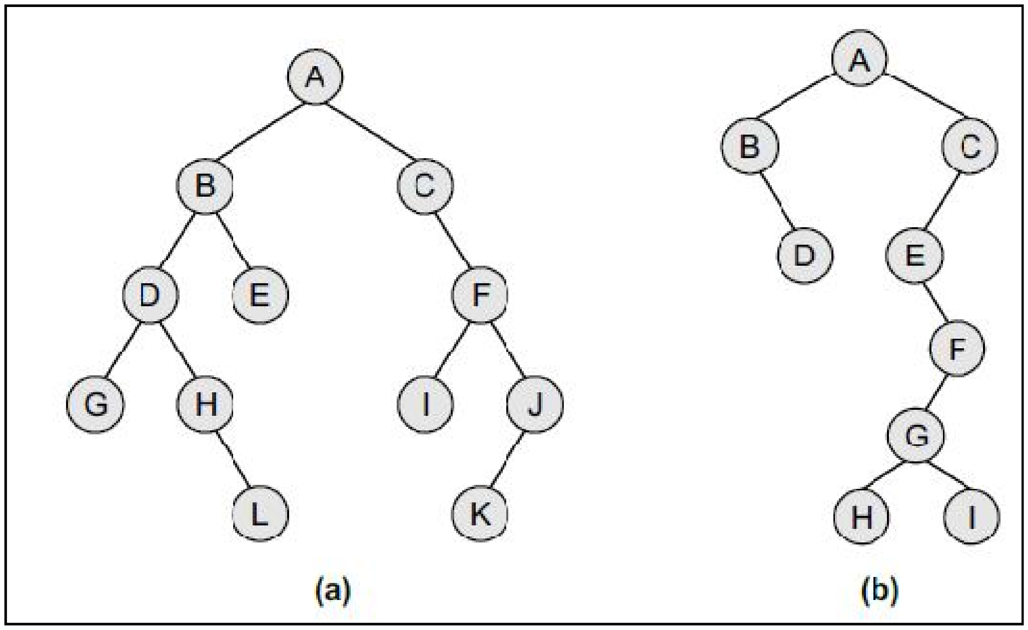
1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

Algorithm: INORDER()

Input: ROOT

Output: NODE_VISIT_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
```



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

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

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

3 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:

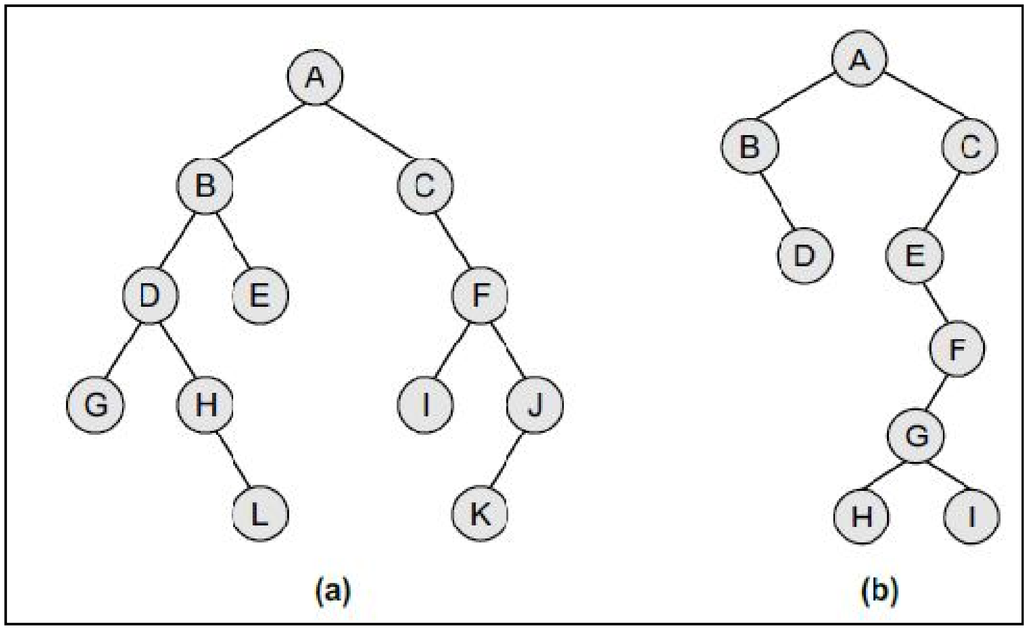
1. Traversing the left sub-tree,
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

Algorithm: POSTORDER()

Input: ROOT

Output: NODE_VISIT_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
```



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

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

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

04 AVL search 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.

In its structure, it stores an additional variable called the BalanceFactor. 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.

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.

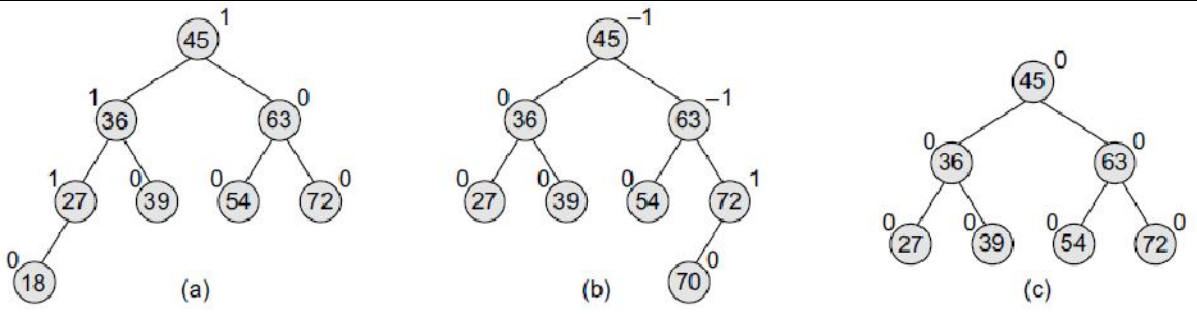


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Operations on AVL Trees:

01 Inserting a New Node in an AVL Tree:

Insertion in an AVL tree is also done in the same way as it is done in a binary search tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required.

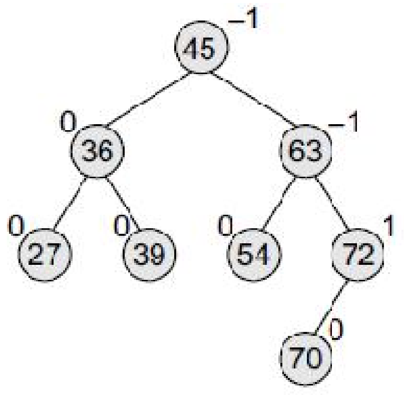


Figure 10.36 AVL tree

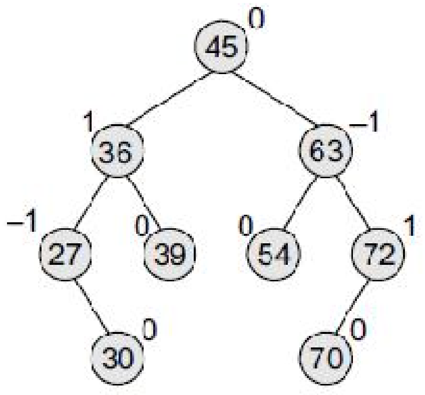


Figure 10.37 AVL tree after inserting a node with the value 30

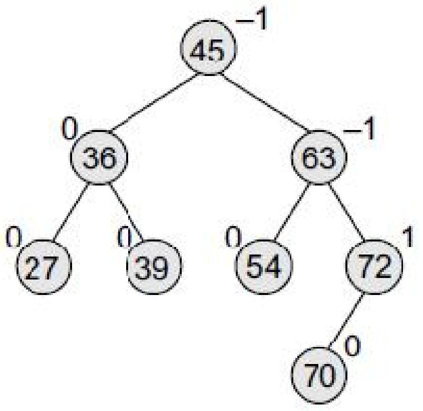


Figure 10.38 AVL tree

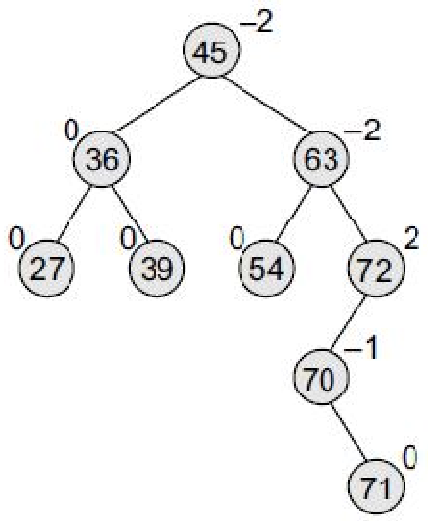


Figure 10.39 AVL tree after inserting a node with the value 71

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

Solution

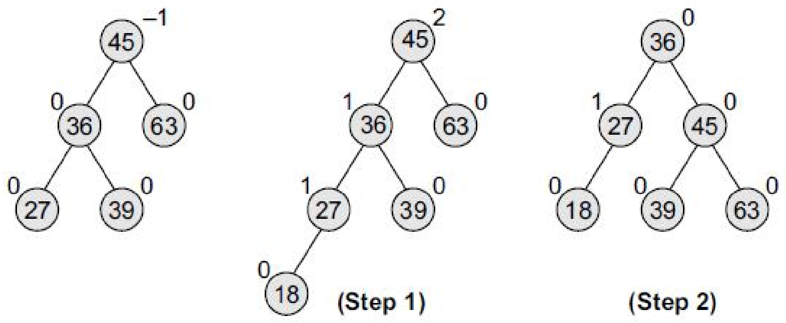


Figure 10.41 AVL tree

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

Solution

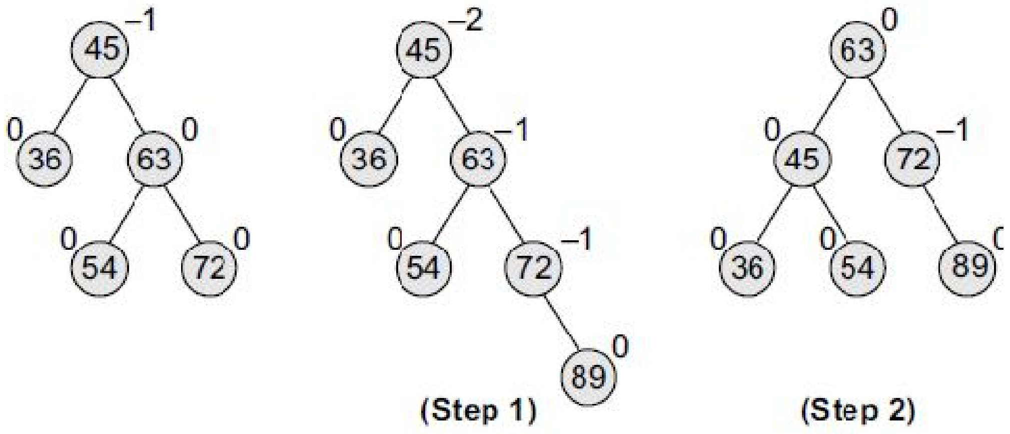


Figure 10.43 AVL tree

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

Solution

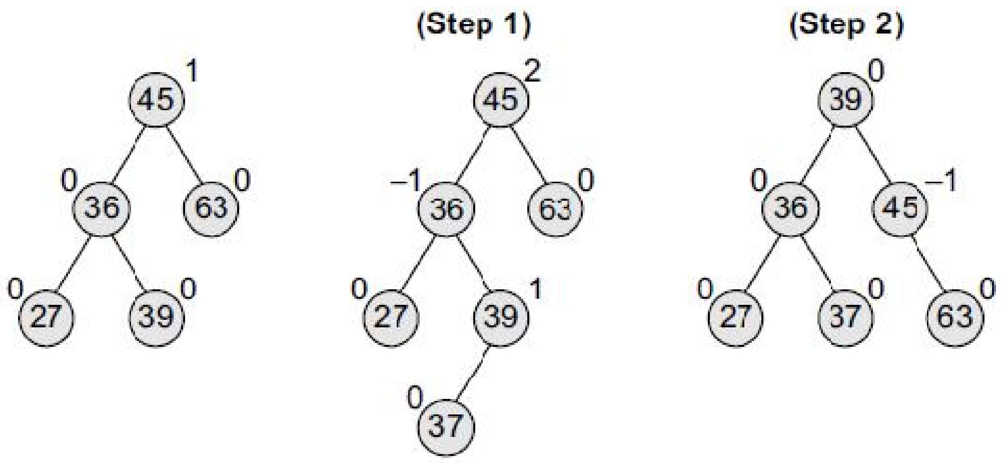
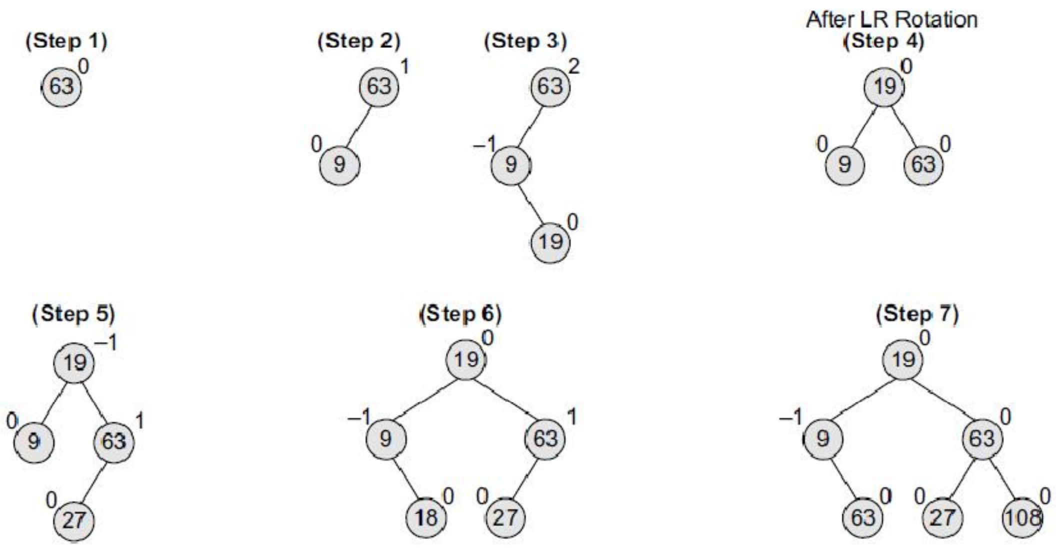


Figure 10.45 AVL tree

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



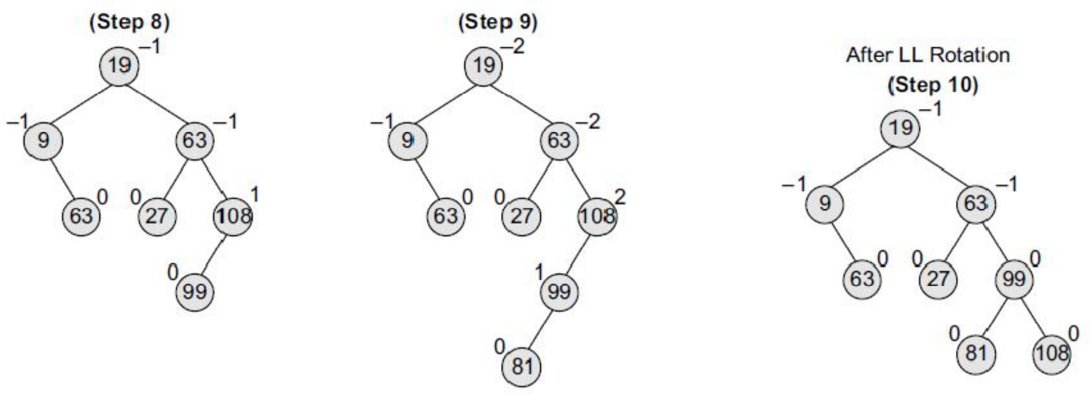


Figure 10.47 AVL tree

02 Deleting a Node from an AVL Tree:

Deletion of a node in an AVL tree is similar to that of binary search trees. But it goes one step ahead. Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree, we need to perform rotations. There are two classes of rotations that can be performed on an AVL tree after deleting a given node. These rotations are R rotation and L rotation.

Example 10.7 Consider the AVL tree given in Fig. 10.49 and delete 72 from it.

Solution

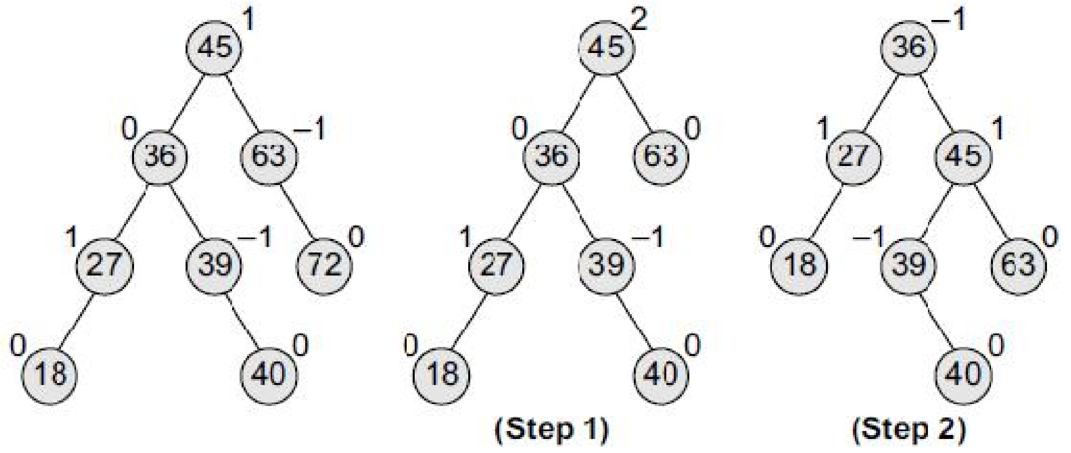


Figure 10.49 AVL tree

Example 10.8 Consider the AVL tree given in Fig. 10.51 and delete 72 from it.
Solution

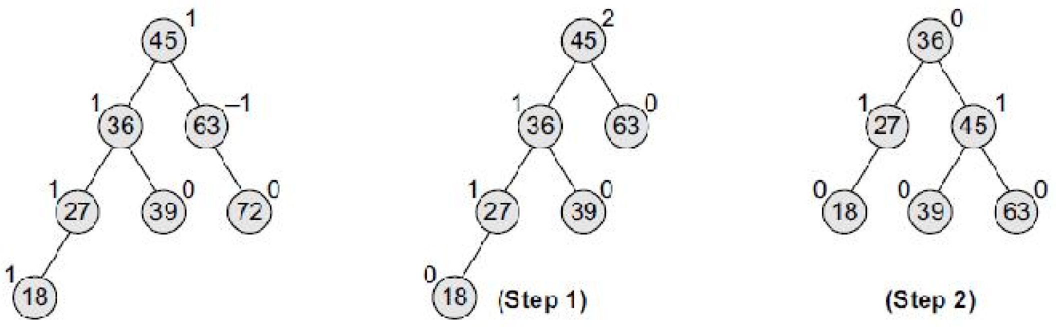


Figure 10.51 AVL tree

Example 10.9 Consider the AVL tree given in Fig. 10.53 and delete 72 from it.
Solution

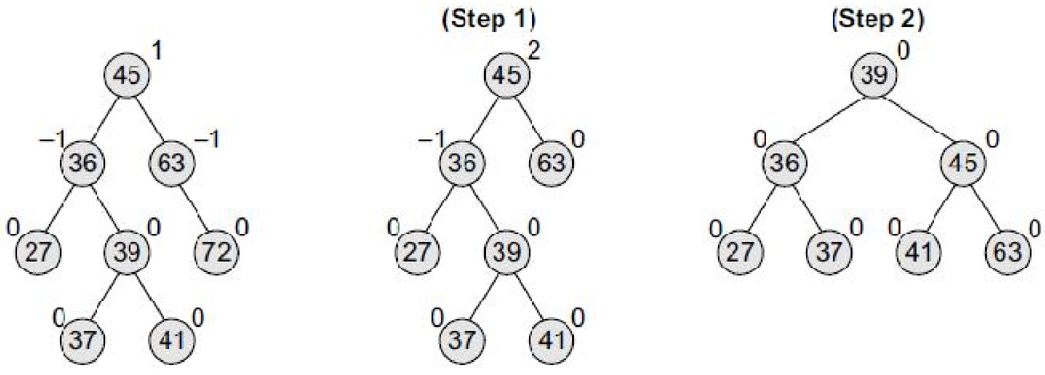


Figure 10.53 AVL tree

Example 10.10 Delete nodes 52, 36, and 61 from the AVL tree given in Fig. 10.54.
Solution

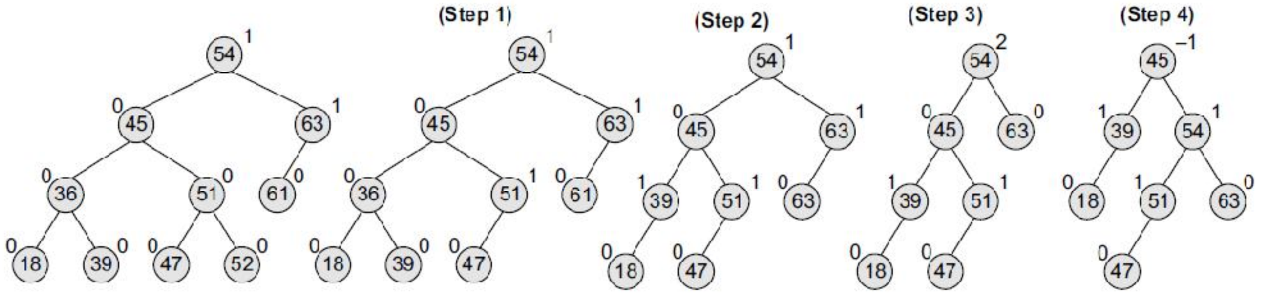


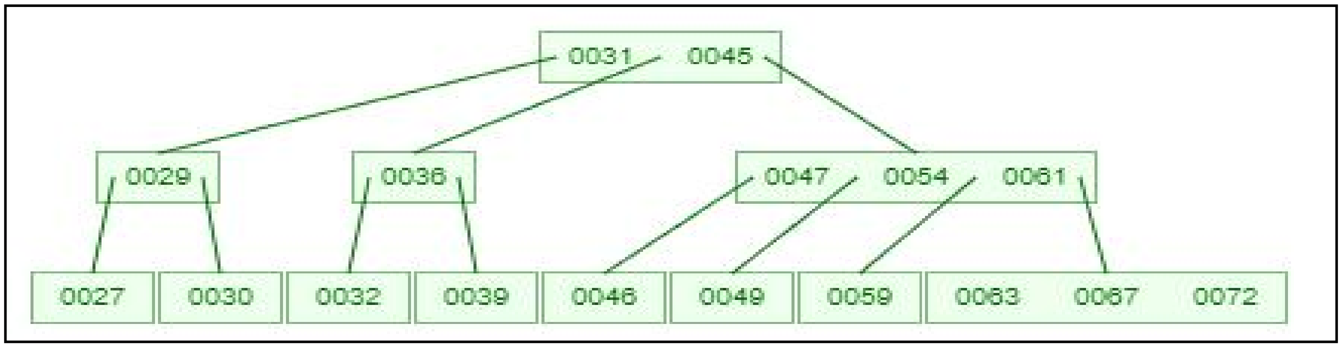
Figure 10.54 AVL tree

05 B Tree:

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees. A B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time. A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. In addition it has the following properties:

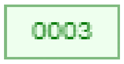
1. Every node in the B tree has at most (maximum) m children.
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2 children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.
3. The root node has at least two children if it is not a terminal (leaf) node.
4. All leaf nodes are at the same level.



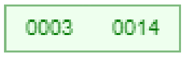
B Tree Order 4 (Max size 3)

Example Create a B tree of order 6 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.

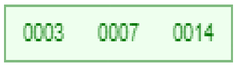
Insert 30



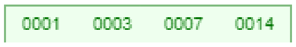
Insert 14



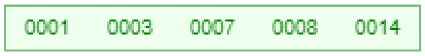
Insert 7



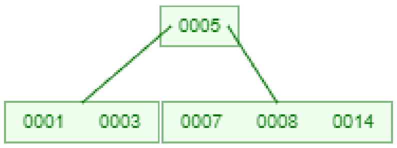
Insert 1



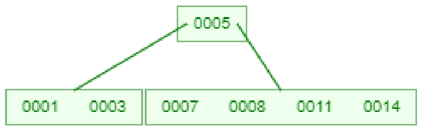
Insert 8



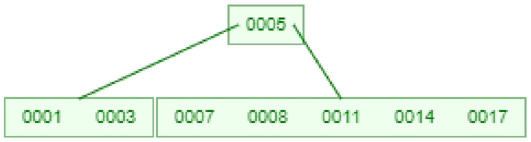
Insert 5



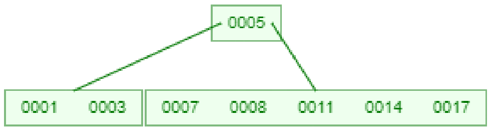
Insert 11



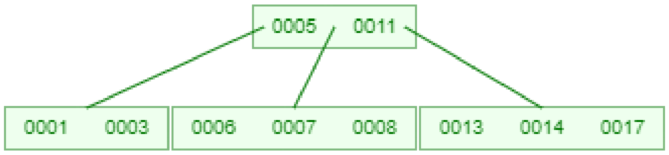
Insert 17



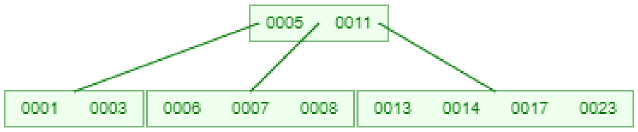
Insert 13



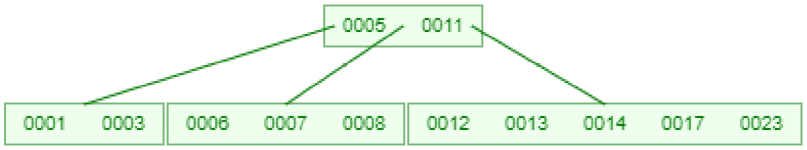
Insert 6



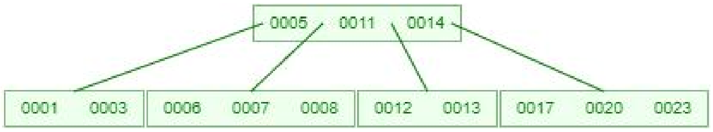
Insert 23



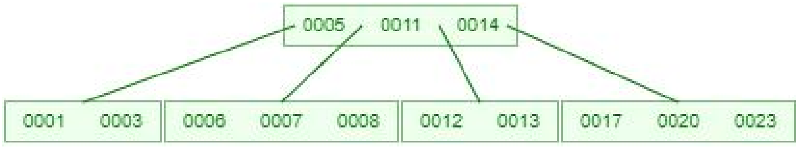
Insert 12



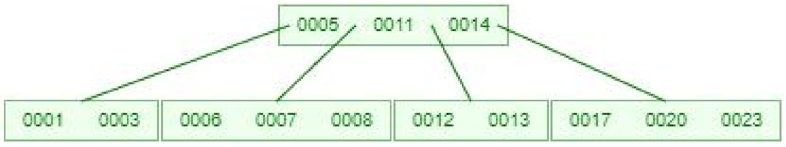
Insert 20



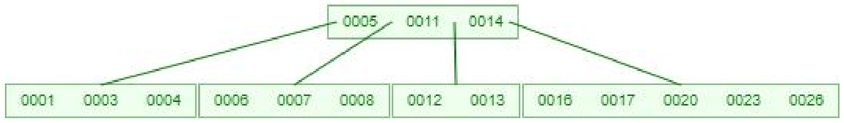
Insert 26



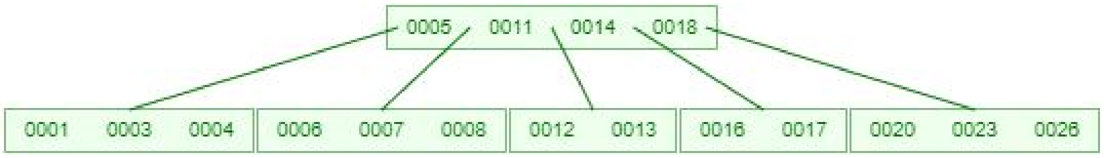
Insert 4



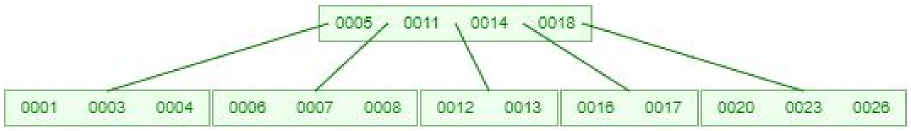
Insert 16



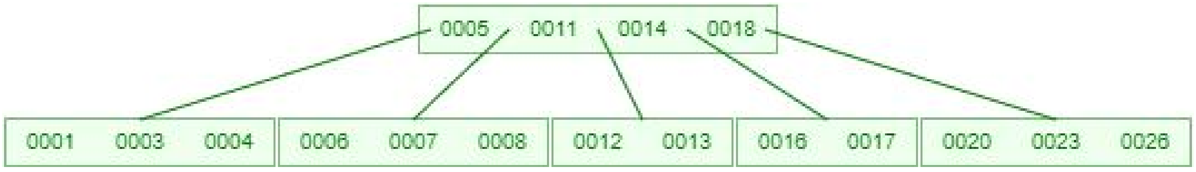
Insert 18



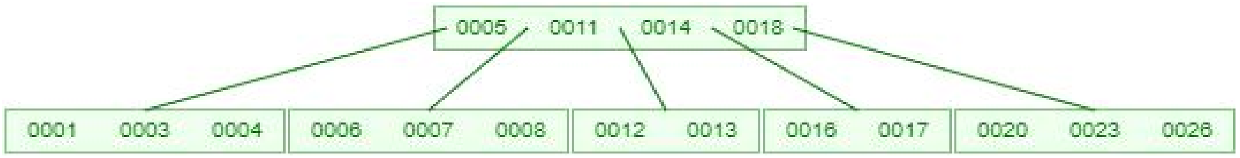
Insert 24



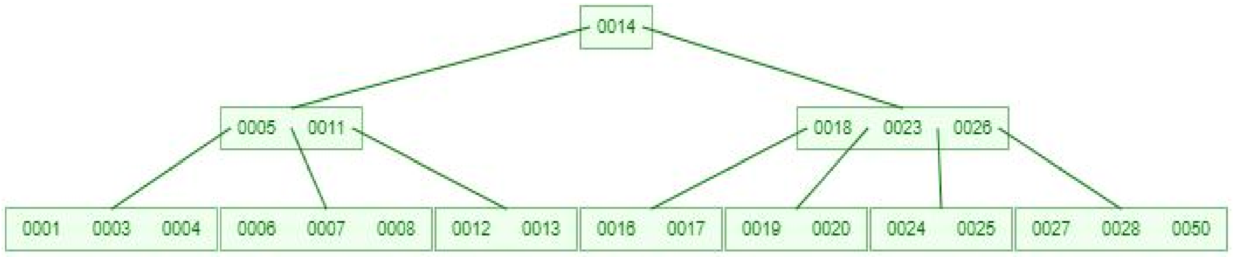
Insert 25



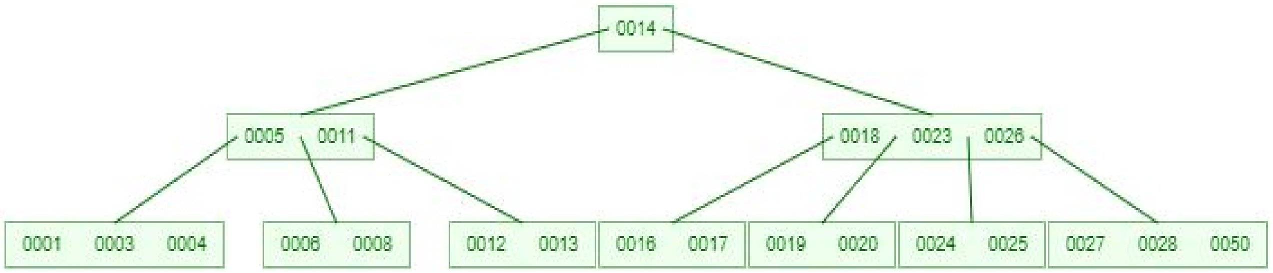
Insert 19



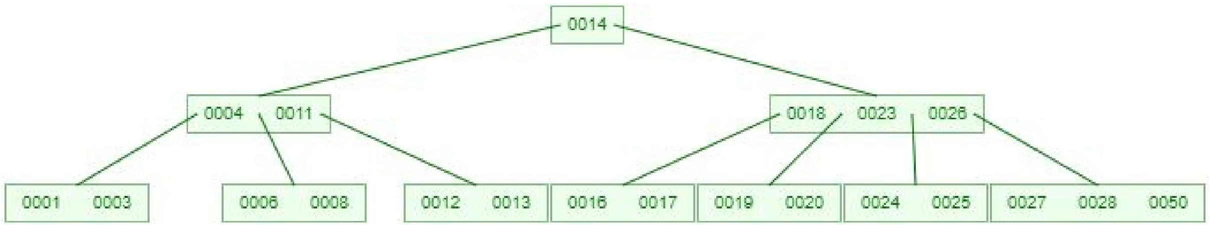
Example 11.2 Consider the following B tree of order 6 and delete values 7, 5, 23, and 14 from it.



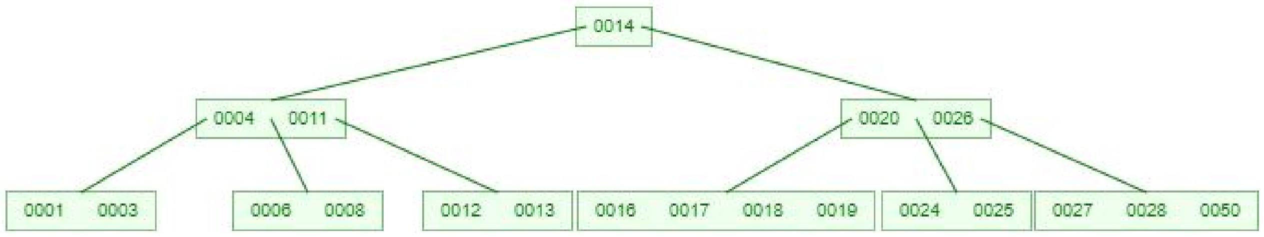
Delete 7



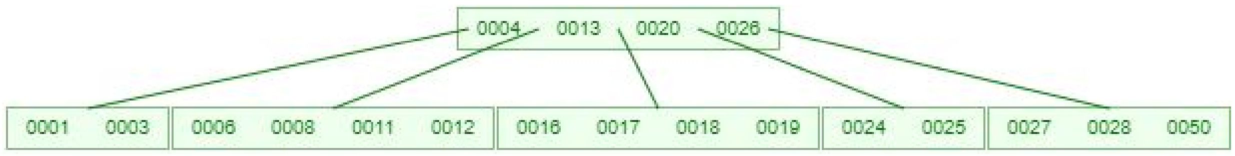
Delete 5



Delete 23

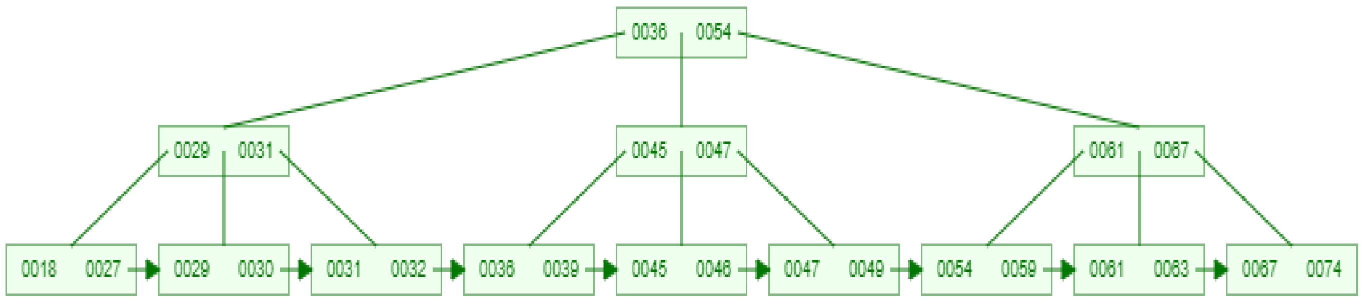


Delete 14



06 B+ Tree:

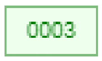
A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a *key*. While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree.



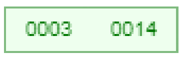
B+ Tree Order 4 (Max size 3)

Example Create a B+ tree of order 6 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.

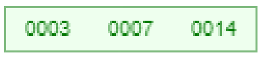
Insert 3



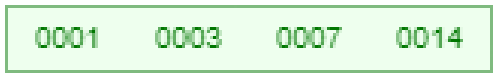
Insert 14



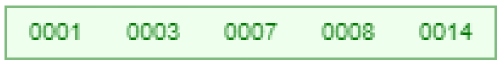
Insert 7



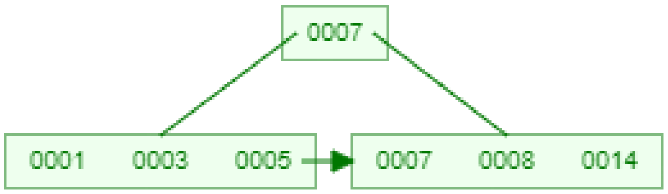
Insert 1



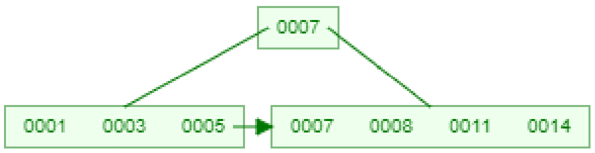
Insert 8



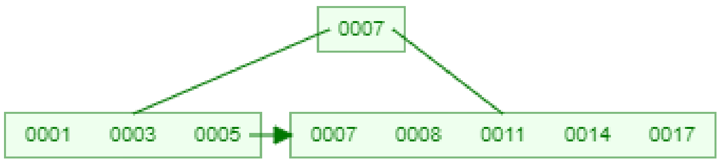
Insert 5



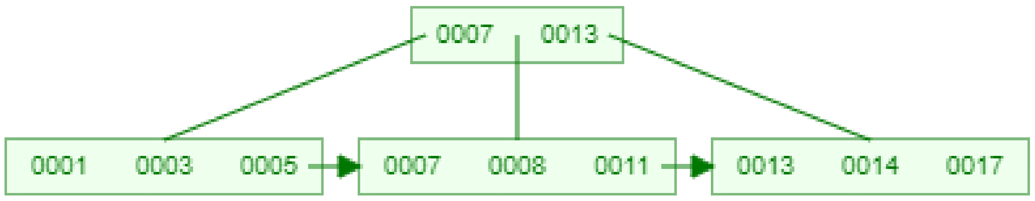
Insert 11



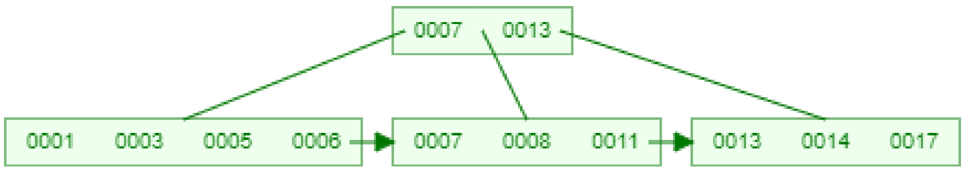
Insert 17



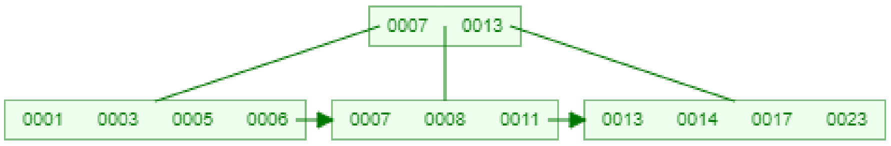
Insert 13



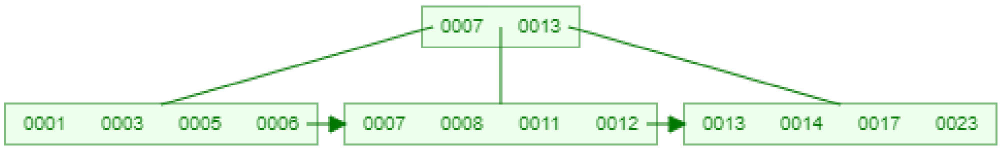
Insert 6



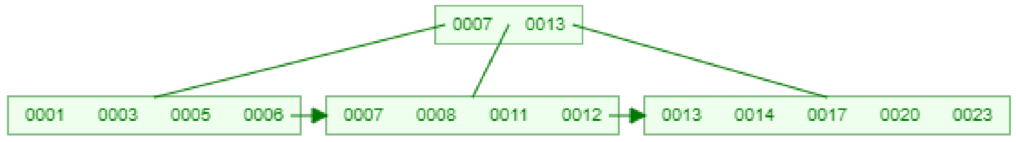
Insert 23



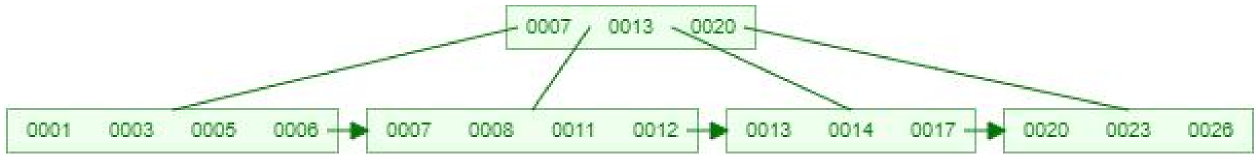
Insert 12



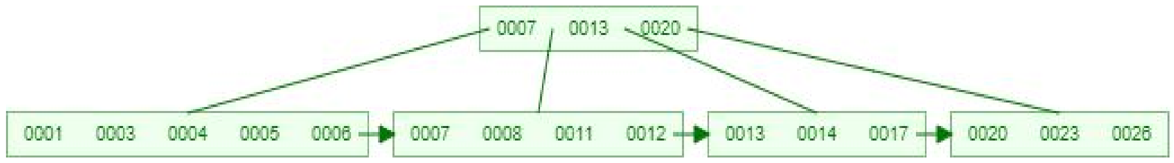
Insert 20



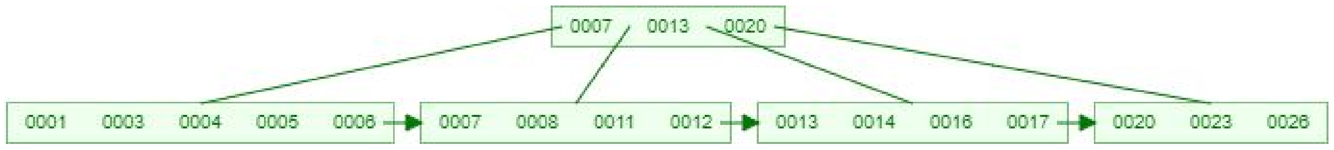
Insert 26



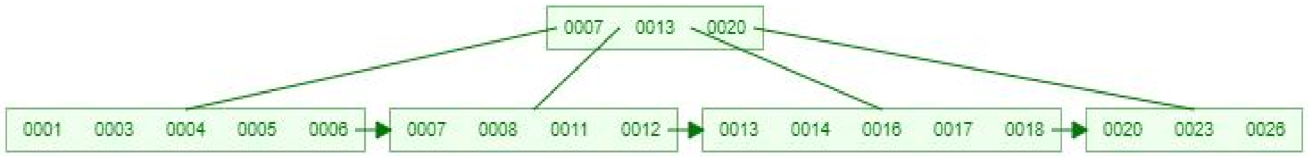
Insert 4



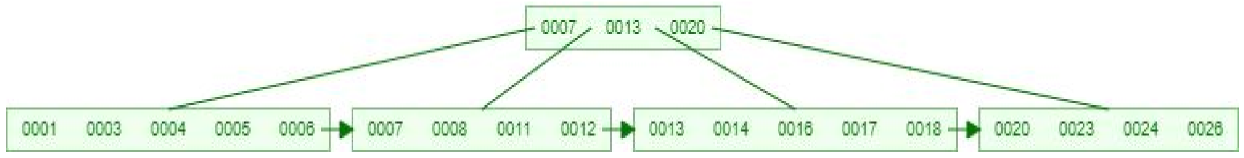
Insert 16



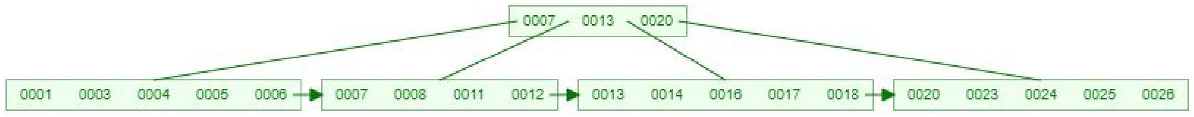
Insert 18



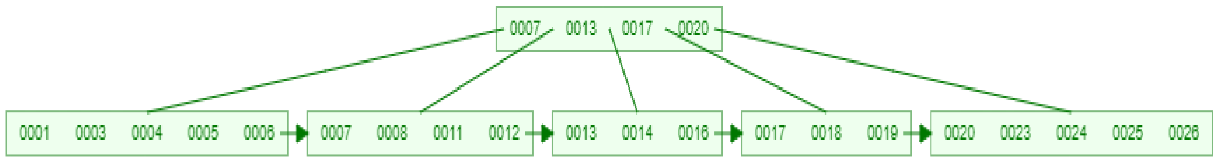
Insert 24



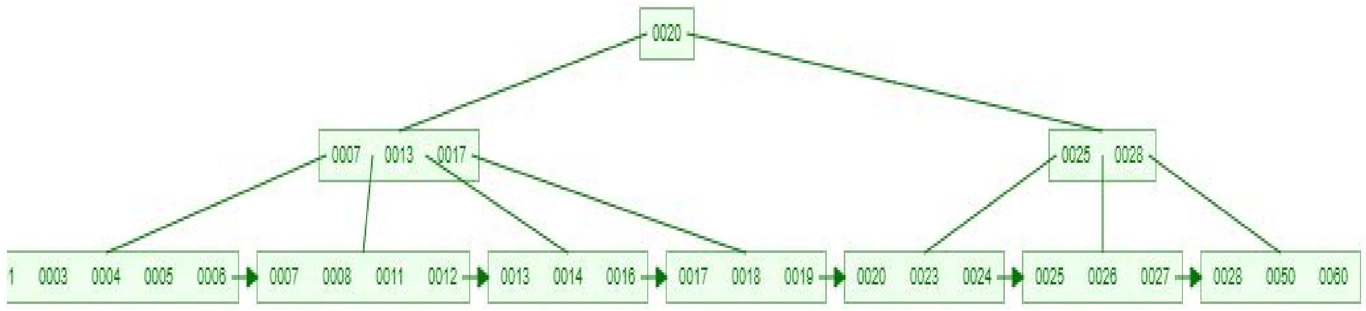
Insert 25



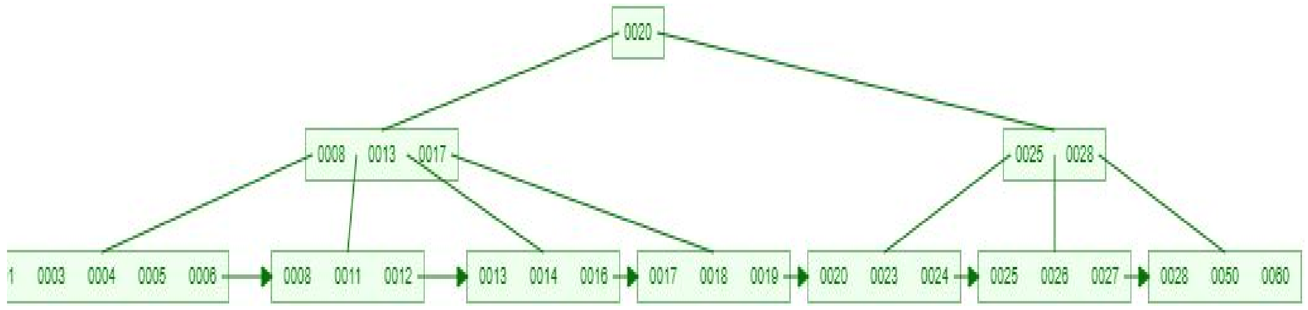
Insert 19



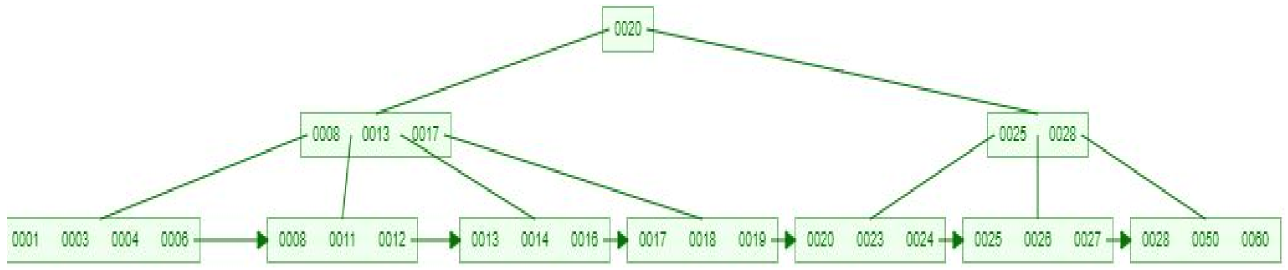
Example 11.2 Consider the following B+ tree of order 6 and delete values 7, 5, 28, and 20 from it.



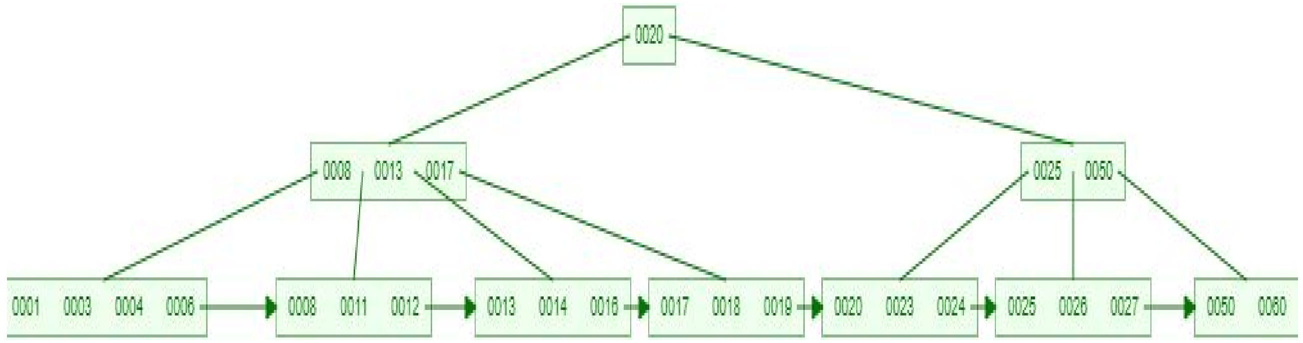
Delete 7



Delete 5



Delete 28



Delete 20

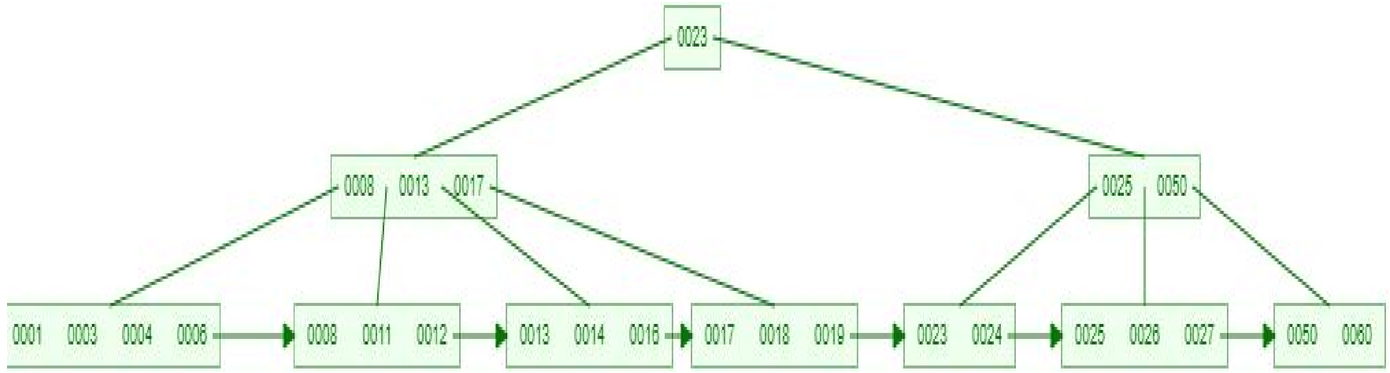


Table 11.1 Comparison between B trees and to B+ trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

07 Heaps:

A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father. The root of the binary tree (*i.e.,* the first array element) holds the largest key in the heap. This type of heap is usually called descending heap or mere heap, as the path from the root node to a terminal node forms an ordered list of elements arranged in descending order.

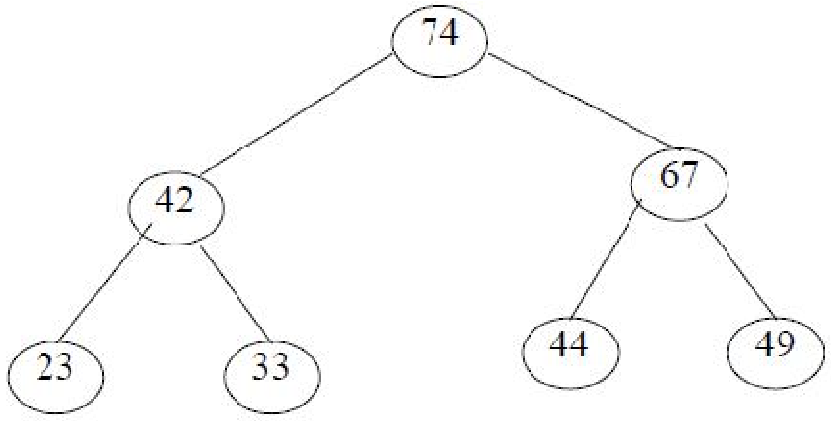


Fig. 6.1. Heap representation

74	42	67	23	33	44	49

Fig. 6.2. Sequential representation

08 Heap Operations:

1 Inserting a New Element in a Binary Heap:

Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and
Step 2 is repeated.

Example 12.1 Consider the max heap given in Fig. 12.2 and insert 99 in it.

Solution

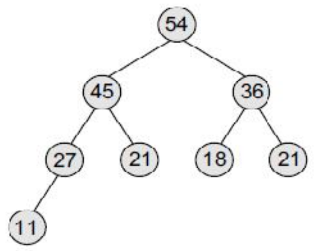


Figure 12.2 Binary heap

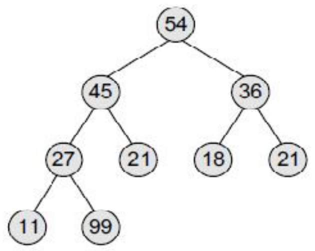


Figure 12.3 Binary heap after insertion of 99

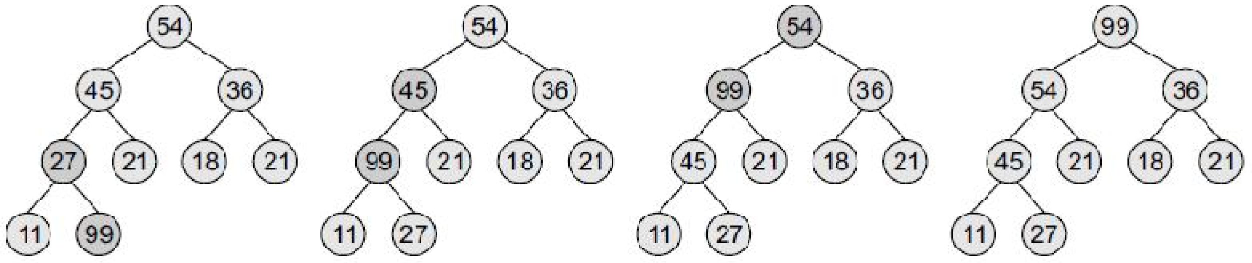
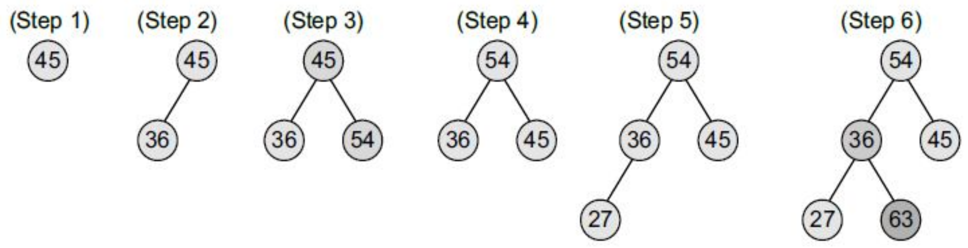


Figure 12.4 Heapify the binary heap

Example 12.2 Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.

Solution



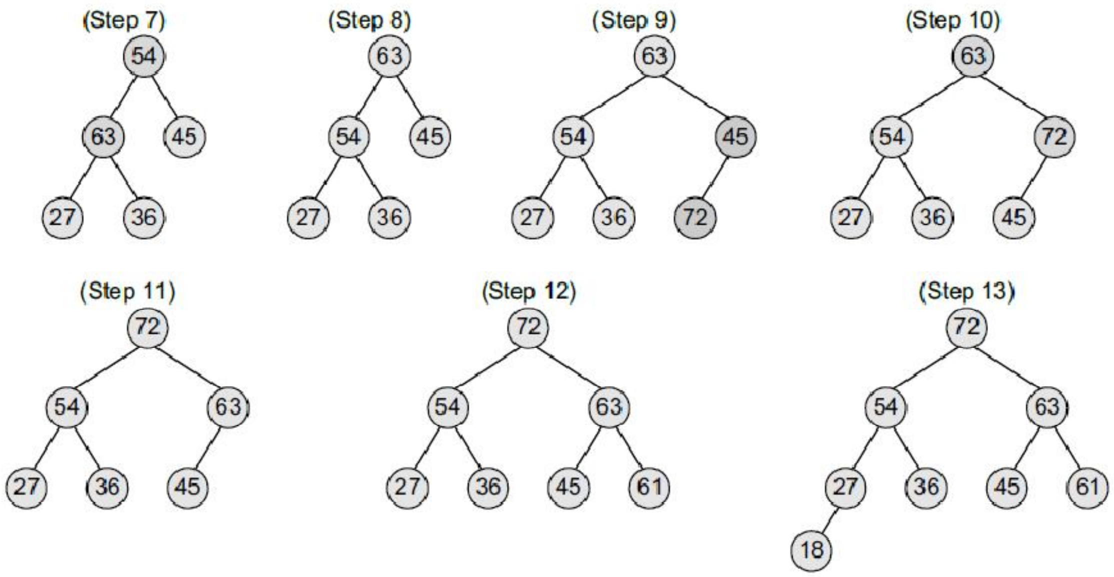


Figure 12.5

The memory representation of H can be given as shown in Fig. 12.6.

HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]	HEAP[10]
72	54	63	27	36	45	61	18		

Figure 12.6 Memory representation of binary heap H

2 Deleting an Element from a Binary Heap

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
2. Delete the last node.
3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Example 12.3 Consider the max heap H shown in Fig. 12.8 and delete the root node's value.

Solution

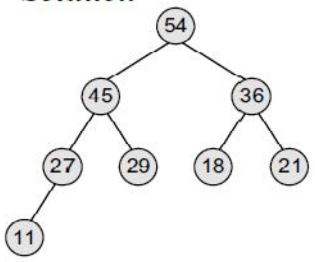


Figure 12.8 Binary heap

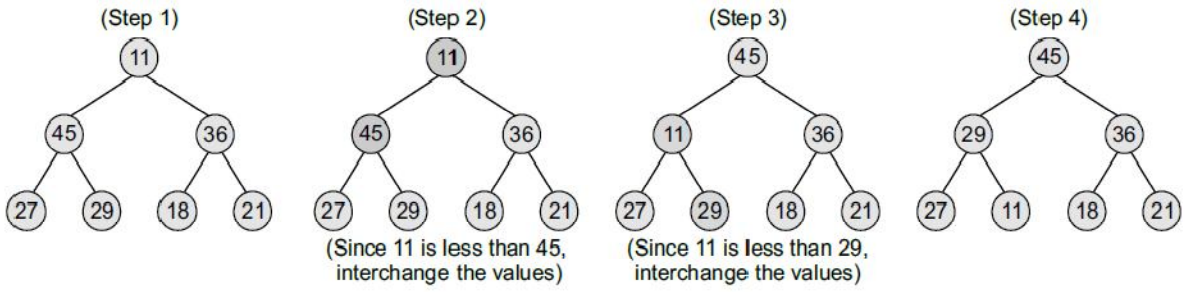

(Step 1)
(Step 2)
(Step 3)
(Step 4)
11
45
36
27
29
18
21
(Since 11 is less than 45, interchange the values)
(Since 11 is less than 29, interchange the values)

Figure 12.9 Binary heap

09 Heap applications:

Binary heaps are mainly applied for

1. Sorting an array using heapsort algorithm.
2. Implementing priority queues.

10 Heap sort:

Given an array ARR with n elements, the heap sort algorithm can be used to sort ARR in two phases:

- In phase 1, build a heap H using the elements of ARR.
- In phase 2, repeatedly delete the root element of the heap formed in phase 1.

In a max heap, we know that the largest value in H is always present at the root node. So in phase 2, when the root element is deleted, we are actually collecting the elements of ARR in decreasing order.

Algorithm: Heap_sort()

Input: ARR(List of element unsorted)

Output: List of Element in descending order

```
HEAPSORT(ARR, N)

Step 1: [Build Heap H]
      Repeat for I = 0 to N-1
            CALL Insert_Heap(ARR, N, ARR[I])
      [END OF LOOP]
Step 2: (Repeatedly delete the root element)
      Repeat while N>0
            CALL Delete_Heap(ARR, N, VAL)
            SET N = N + 1
      [END OF LOOP]
Step 3: END
```