UNIT-IV

1. TREES

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. Figure 1.1 shows a tree where node A is the root node; nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.

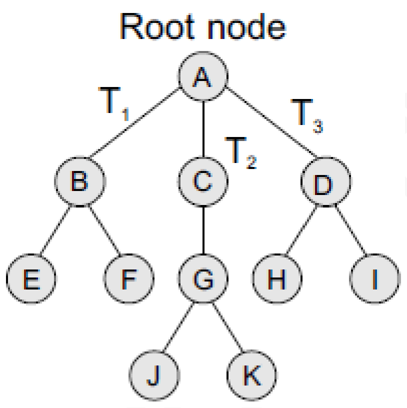


Figure 1.1 Tree

1.1 BASIC TERMINOLOGY IN TREES:

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. 1.1, the path from the root node A to node I is given as: A, D, and I.

Ancestor node: An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig. 1.1, nodes A, C, and G are the ancestors of node K.

Descendant node: A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig. 1.1, 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.

1.2 BINARY TREES:

Definition:

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children. A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.

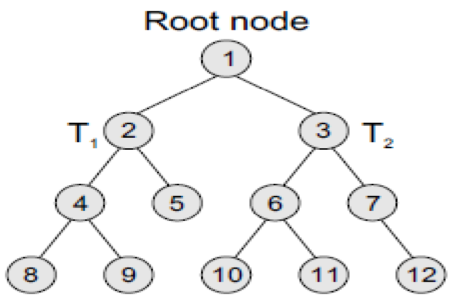


Figure 1.2 Binary tree

Figure 1.2 shows a binary tree. In the figure, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.

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

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

A binary tree is recursive by definition as every node in the tree contains a left sub-tree and a right sub-tree. Even the terminal nodes contain an empty left sub-tree and an empty right sub-tree. Look at Fig. 1.2, nodes 5, 8, 9, 10, 11, and 12 have no successors and thus said to have empty sub-trees.

Complete Binary Trees:

A *complete binary tree* is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible.

In a complete binary tree T_n, there are exactly n nodes and level r of T can have at most 2^r nodes. Figure 1.3 shows a complete binary tree.

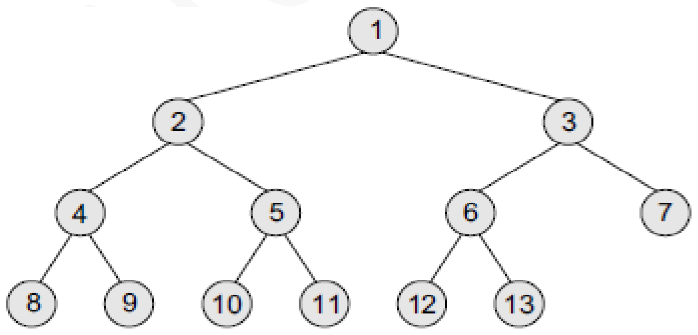


Figure 1.3 Complete binary tree

Note that in Fig. 1.3, level 0 has 2^0 = 1 node, level 1 has 2^1 = 2 nodes, level 2 has 2^2 = 4 nodes, level 3 has 6 nodes which is less than the maximum of 2^3 = 8 nodes.

In Fig. 1.3, tree T_{13} has exactly 13 nodes. They have been purposely labelled from 1 to 13, so that it is easy for the reader to find the parent node, the right child node, and the left child node of the given node. The formula can be given as—if K is a parent node, then its left child can be calculated as 2 × K and its right child can be calculated as 2 × K + 1. For example, the children of the node 4 are 8 (2 × 4) and 9 (2 × 4 + 1). Similarly, the parent of the node K can be calculated as | K/2 |. Given the node 4, its parent can be calculated as | 4/2 | = 2. The height of a tree T_n having exactly n nodes is given as: $H_n = | \log_2 (n + 1) |$. This means, if a tree T has 10,00,000 nodes, then its height is 21.

Extended Binary Trees:

A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children. Figure 1.4 shows how an ordinary binary tree is converted into an extended binary tree.

In an extended binary tree, nodes having two children are called *internal nodes* and nodes having no children are called *external nodes*. In Fig. 1.4, the internal nodes are represented using circles and the external nodes are represented using squares.

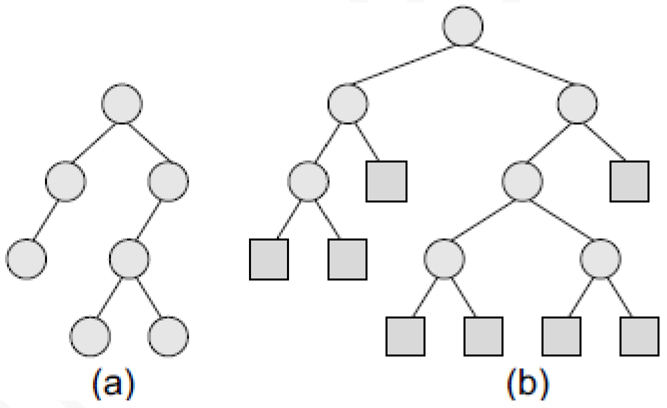


Figure 1.4 (a) Binary tree and (b) extended binary tree

To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes.

1.2.1 Properties of Binary Trees:

- A tree with ‘n’ nodes has exactly (n-1) edges or branches.
- The height or depth of a binary tree is the number of levels in it.
- In a tree every node except the root has exactly one parent (and the root node does not have a parent node).
- The maximum number of nodes in a binary tree of height ‘h’ is $(2^{h+1})-1$ where h>=0.

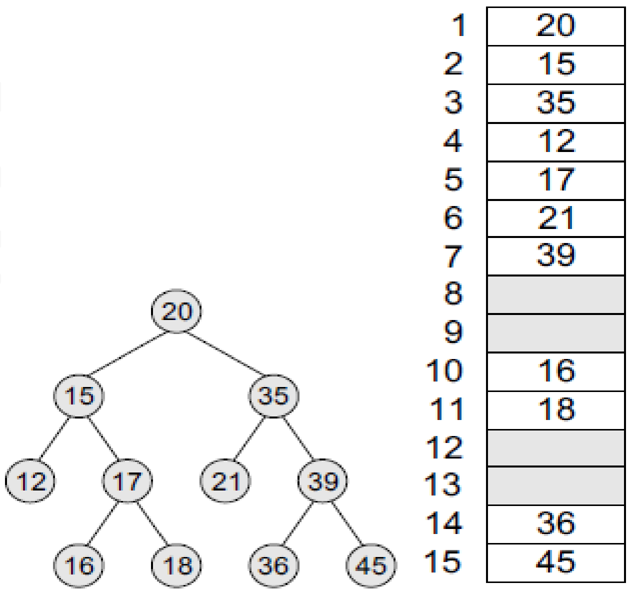
1.2.2 Representation of Binary Trees:

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

1) 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.



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 1.5 Binary tree and its sequential representation

Figure 1.5 shows a binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4.

2) 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. 1.6. The schematic diagram of the linked representation of the binary tree is shown in Fig. 1.7.

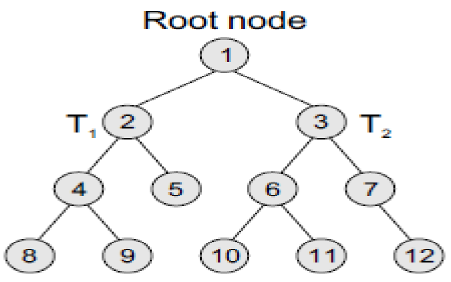


Figure 1.6 Binary tree

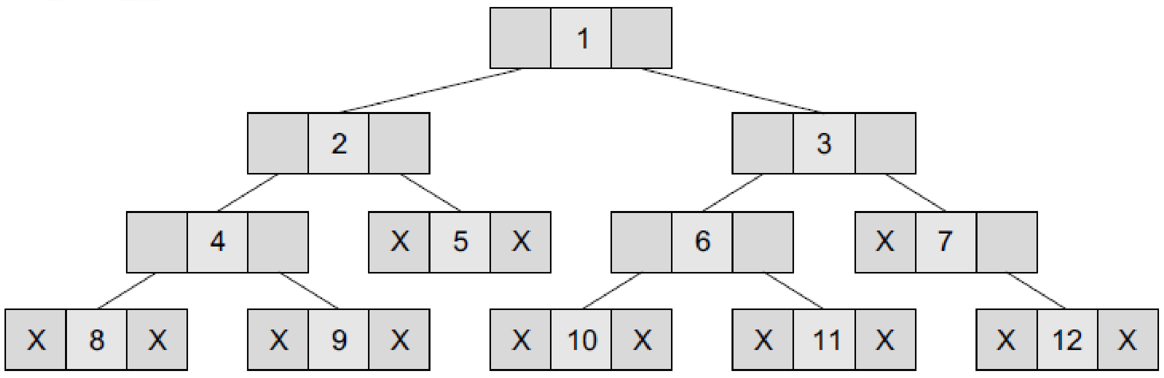


Figure 1.7 Linked representation of a binary tree

In Fig. 1.7, 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).

1.3 BINARY SEARCH TREES:

1.3.1 Basic Concepts:

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

In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)

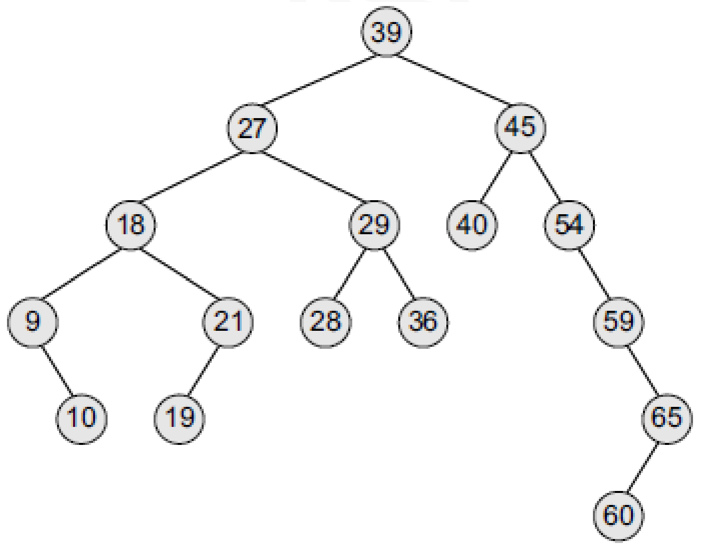


Figure 1.8 Binary search tree

Look at Fig. 1.8. The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65. Recursively, each of the sub-trees also obeys the binary search tree constraint. For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10, 18, 19, 21) are smaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node's value.

Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced. Whenever we search for an element, we do not need to traverse the entire tree. At every node, we get a hint regarding which sub-tree to search in. For example, in the given tree, if we have to search for 29, then we know that we have to scan only the left sub-tree. If the value is present in the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value). The left sub-tree has a root node with the value 27. Since 29 is greater than 27, we will move to the right sub-tree, where we will find the element. Thus, the average running time of a search operation is O(log2n), as at every step, we eliminate half of the sub-tree from the search process. Due to its efficiency in searching elements, binary search trees are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

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

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

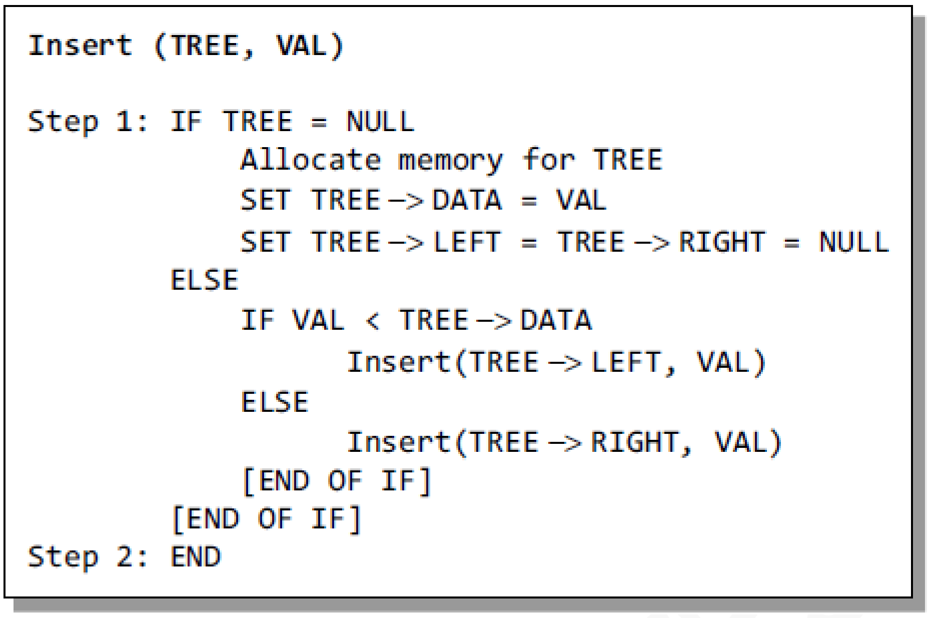
- The left sub-tree of a node N contains values that are less than N's value.
- The right sub-tree of a node N contains values that are greater than N's value.
- Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

1.3.2 BST Operations:

1) Inserting a New Node in a Binary Search Tree:

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. Figure 1.9 shows the algorithm to insert a given value in a binary search tree.

The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.



```
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 1.9 Algorithm to insert a given value in a binary search tree

In Step 1 of the algorithm, the insert function checks if the current node of TREE is NULL. If it is NULL, the algorithm simply adds the node, else it looks at the current node's value and then recurs down the left or right sub-tree. If the current node's value is less than that of the new node, then the right sub-tree is traversed, else the left sub-tree is traversed. The insert function continues moving down the levels of a binary tree until it reaches a leaf node. The new node is added by following the rules of the binary search trees. That is, if the new node's value is greater than that of the parent node, the new node is inserted in the right sub-tree, else it is inserted in the left sub-tree. The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case.

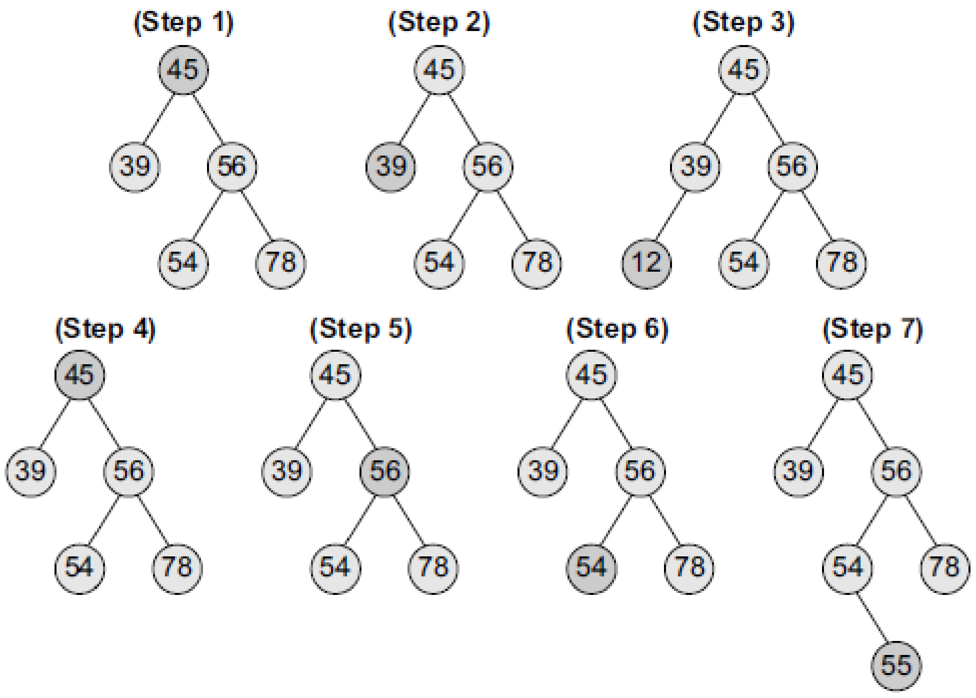


Figure 1.10 Inserting nodes with values 12 and 55 in the given binary search tree

Look at Fig. 1.10 which shows insertion of values in a given tree. We will take up the case of inserting 12 and 55.

- In the step1, the value 12 is compared with root node 45, it is less than 45. So, it goes to left sub tree of 45.
- In the step2, again 12 is compared with 39, and 12 is less than 39. So, again it goes to left sub tree of 39.
- In the step3, it checks for the values but there are no elements at the left sub tree of 39. So, the new value 12 is inserted at left side of the node 39.
- Similarly, this process repeats for the element 55.

2) Deleting a Node from a Binary Search Tree:

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process. We will take up three cases in this section and discuss how a node is deleted from a binary search tree.

Case 1: Deleting a Node that has No Children:

Look at the binary search tree given in Fig. 1.11. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.

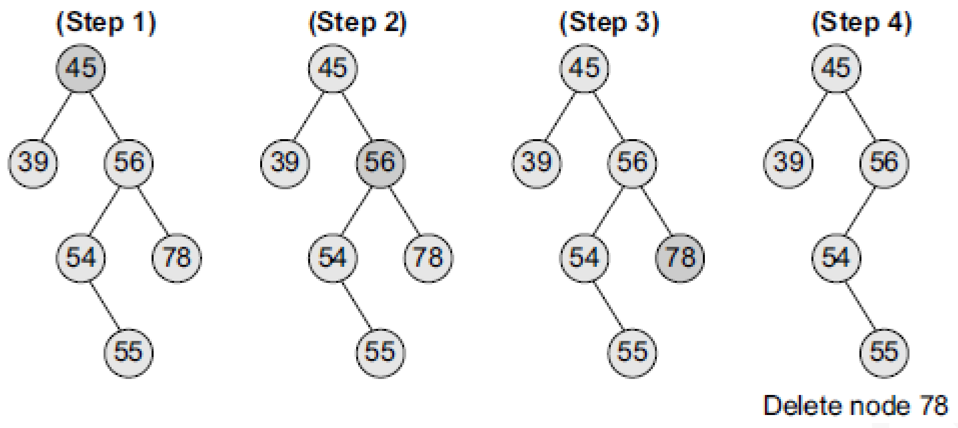


Figure 1.11 Deleting node 78 from the given binary search tree

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. Look at the binary search tree shown in Fig. 1.12 and see how deletion of node 54 is handled.

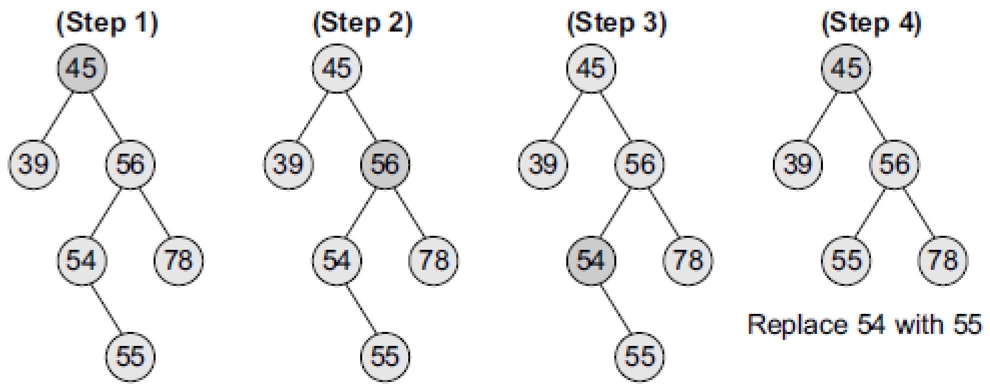


Figure 1.12 Deleting node 54 from the given binary search tree

Case 3: Deleting a Node with Two Children:

To handle this case, replace the node's value with its *in-order predecessor* (largest value in the left sub-tree) or *in-order successor* (smallest value in the right sub-tree). The in-order predecessor or the successor can then be deleted using any of the above cases. Look at the binary search tree given in Fig. 1.13 and see how deletion of node with value 56 is handled.

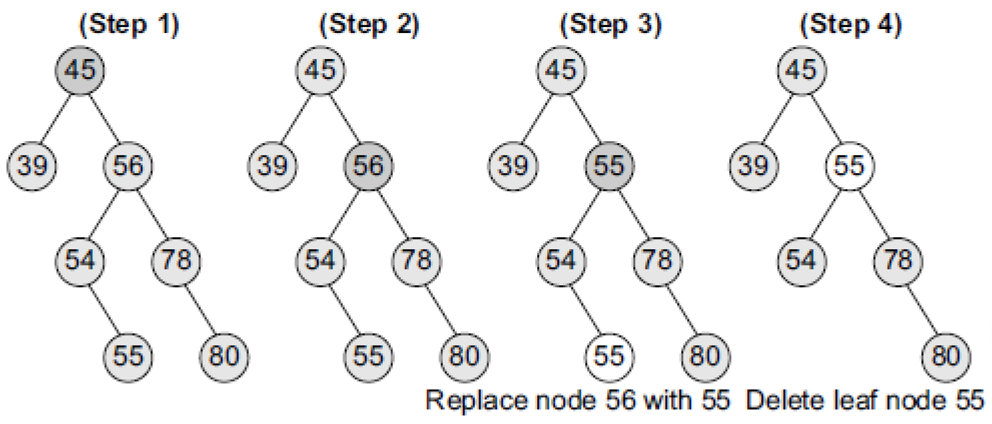


Figure 1.13 Deleting node 56 from the given binary search tree

This deletion could also be handled by replacing node 56 with its in-order successor, as shown in Fig. 1.14.

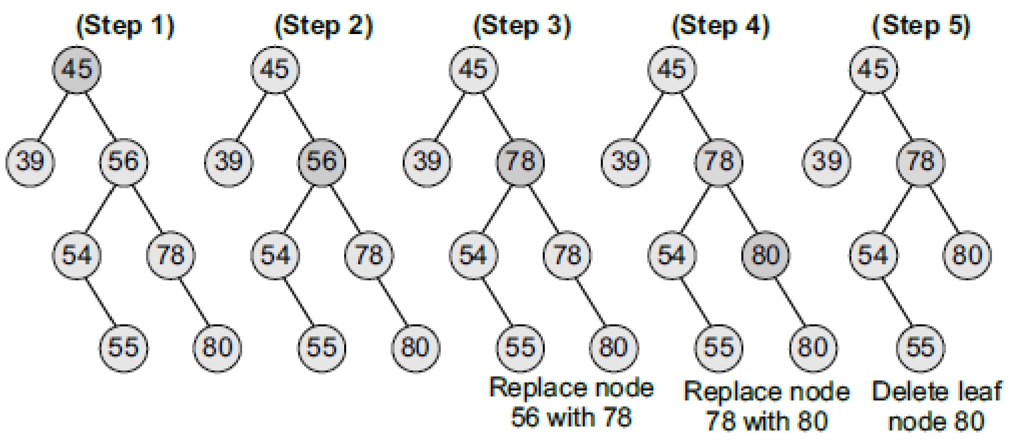


Figure 1.14 Deleting node 56 from the given binary search tree

Now, let us look at Fig. 1.15 which shows the algorithm to delete a node from a binary search tree. In Step 1 of the algorithm, we first check if TREE=NULL, because if it is true, then the node to be deleted is not present in the tree. However, if that is not the case, then we check if the value to be deleted is less than the current node's data. In case the value is less, we call the algorithm recursively on the node's left sub-tree, otherwise the algorithm is called recursively on the node's right sub-tree.

```
Delete (TREE, VAL)

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

Figure 1.15 Algorithm to delete a node from a binary search tree

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

If the node to be deleted does not have any child, then we simply set the node to NULL. Last but not the least, if the node to be deleted has either a left or a right child but not both, then the current node is replaced by its child node and the initial child node is deleted from the tree.

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

3) Tree Traversals:

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. These algorithms differ in the order in which the nodes are visited. In this section, we will discuss these algorithms.

a) 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.

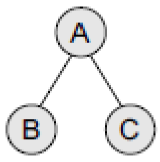


Figure 1.16 Binary tree

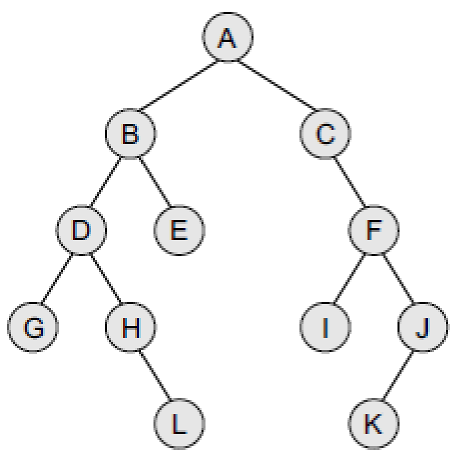
Consider the tree given in Fig. 1.16. The pre-order traversal of the tree is given as A, B, C. Root node first, the left sub-tree next, and then the right sub-tree. Pre-order traversal is also called as *depth-first traversal*. In this algorithm, the left sub-tree is always traversed before the right sub-tree. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right). The algorithm for pre-order traversal is shown in Fig. 1.17.

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

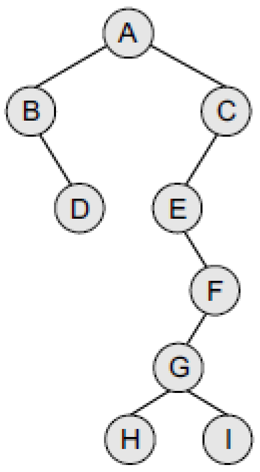
Figure 1.17 Algorithm for pre-order traversal

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

Solution:



(a)



(b)

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

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

b) 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.

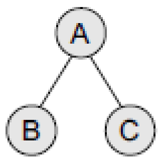


Figure 1.18 Binary tree

Consider the tree given in Fig. 1.18. The in-order traversal of the tree is given as B, A, and C. Left sub-tree first, the root node next, and then the right sub-tree. In-order traversal is also called as *symmetric traversal*. In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree. The word 'in' in the in-order specifies that the root node is accessed in between the left and the right sub-trees. In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right). The algorithm for in-order traversal is shown in Fig. 1.19.

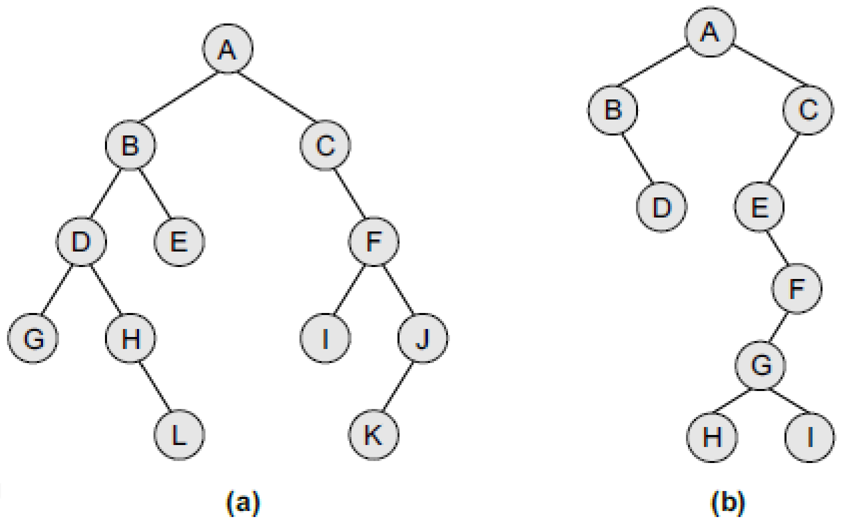
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             INORDER(TREE -> LEFT)
Step 3:             Write TREE -> DATA
Step 4:             INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 1.19 Algorithm for in-order traversal

In-order traversal algorithm is usually used to display the elements of a binary search tree. Here, all the elements with a value lower than a given value are accessed before the elements with a higher value.

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

Solution:



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

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

c) Post-order Traversal

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

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

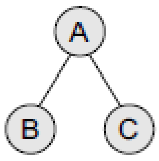


Figure 1.20 Binary tree

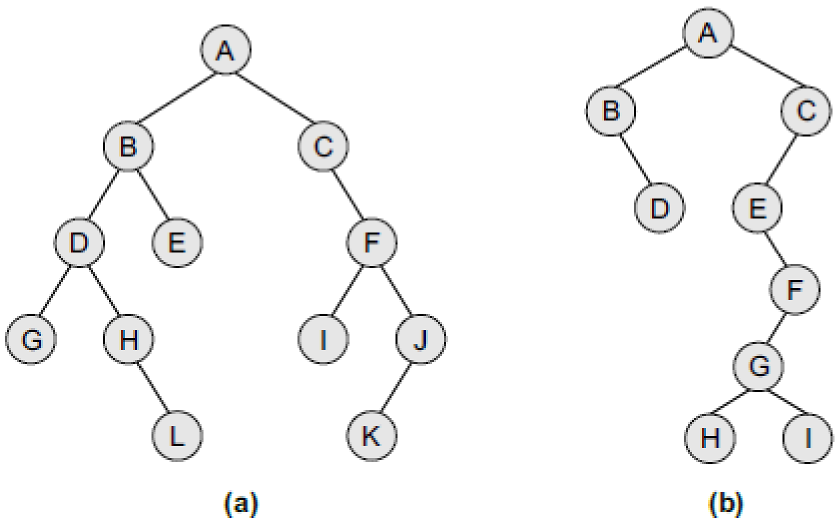
Consider the tree given in Fig. 1.20. The post-order traversal of the tree is given as B, C, and A. Left sub-tree first, the right sub-tree next, and finally the root node. In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node. The word ‘post’ in the post-order specifies that the root node is accessed after the left and the right sub-trees. Post-order algorithm is also known as the LRN traversal algorithm (Left Right-Node). The algorithm for post-order traversal is shown in Fig. 1.21. Post-order traversals are used to extract postfix notation from an expression tree.

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

Figure 1.21 Algorithm for post-order traversal

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

Solution:



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

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

d) Level-order Traversal

In level-order traversal, all the nodes at a level are accessed before going to the next level. This algorithm is also called as the *breadth-first traversal algorithm*. Consider the trees given in Fig. 1.22 and note the level order of these trees.

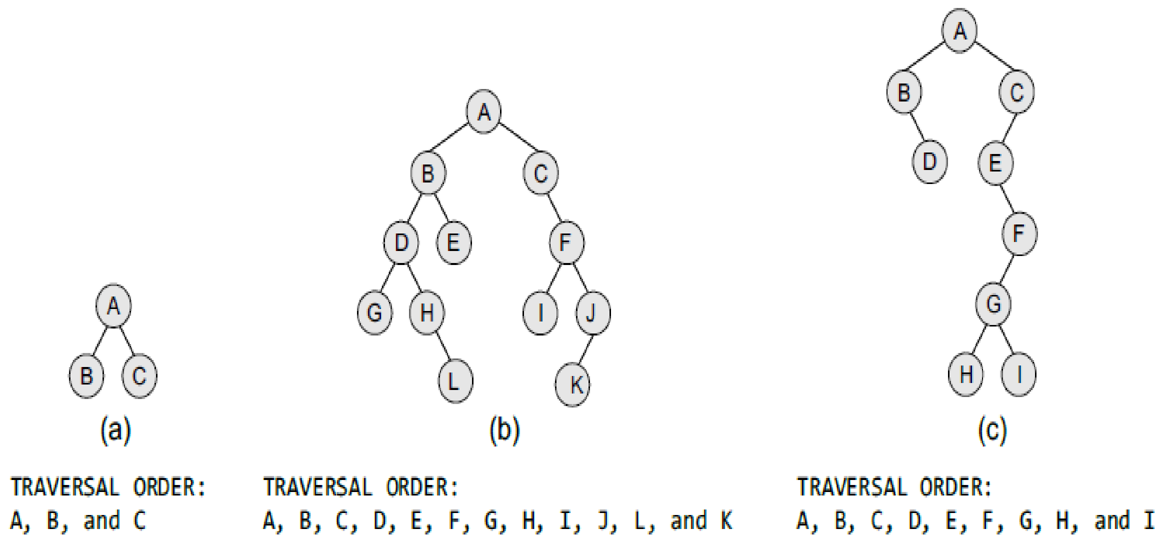


Figure 1.22 Binary trees

1.4 APPLICATIONS:

1.4.1 Expression Trees:

A binary expression tree is a specific kind of a binary tree used to represent expressions. Two common types of expressions that a binary expression tree can represent are algebraic and boolean. These trees can represent expressions that contain both unary and binary operators.

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. 1.23

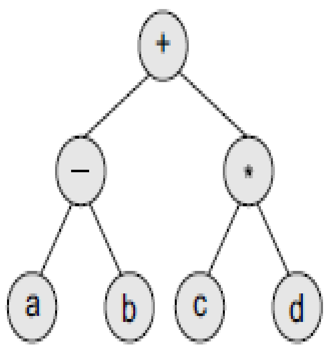
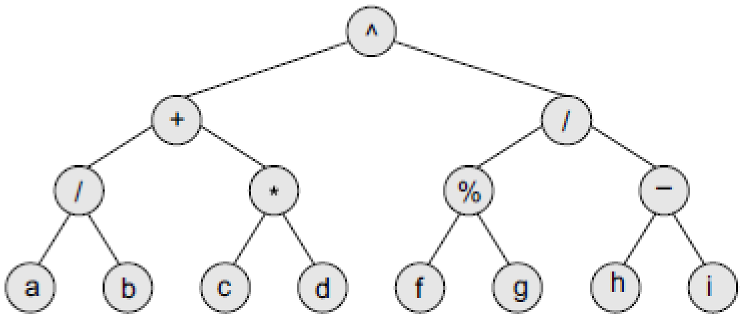


Figure 1.23 Expression tree.

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

Solution:



Expression for the above binary tree is [{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

1.4.2 Heap Sort (Binary Heap):

A binary heap is a complete binary tree in which every node satisfies the heap property which states that: If B is a child of A, then key(A) ≥ key(B)

This implies that elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a *max-heap*.

Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a *min-heap*.

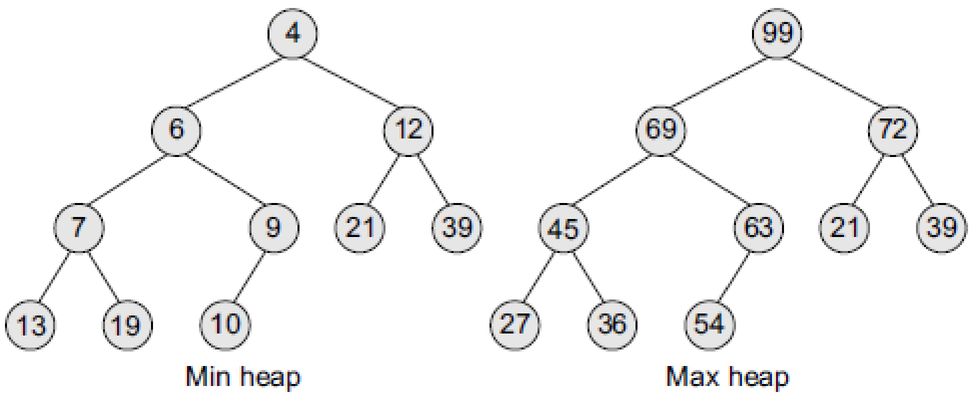


Figure 1.24 Binary heaps

Figure 1.24 shows a binary min heap and a binary max heap. The properties of binary heaps are given as follows:

- Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position

2i and its right child at position 2i+1. Conversely, an element at position i has its parent stored at position i/2.

- Being a complete binary tree, all the levels of the tree except the last level are completely filled.
- The height of a binary tree is given as log2n, where n is the number of elements.
- Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.

A binary heap is a useful data structure in which elements can be added randomly but only the element with the highest value is removed in case of max heap and lowest value in case of min heap. A binary tree is an efficient data structure, but a binary heap is more space efficient and simpler.

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: Consider the max heap given in Fig. 1.25 and insert 99 in it.

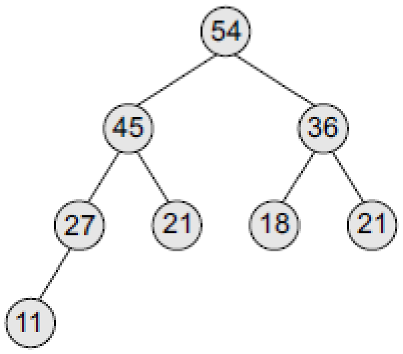


Figure 1.25 Binary heap

Solution:

The first step says that insert the element in the heap so that the heap is a complete binary tree. So, insert the new value as the right child of node 27 in the heap. This is illustrated in Fig. 1.26.

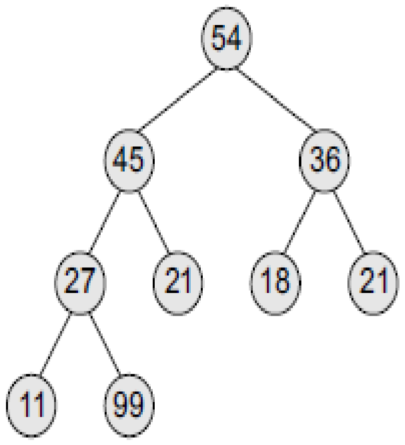


Figure 1.26 Binary heap after insertion of 99

Now, as per the second step, let the new value rise to its appropriate place in H so that H becomes a heap as well. Compare 99 with its parent node value. If it is less than its parent's value, then the new node is in its appropriate place and H is a heap. If the new value is greater than that of its parent's node, then swap the two values. Repeat the whole process until H becomes a heap. This is illustrated in Fig. 1.27.

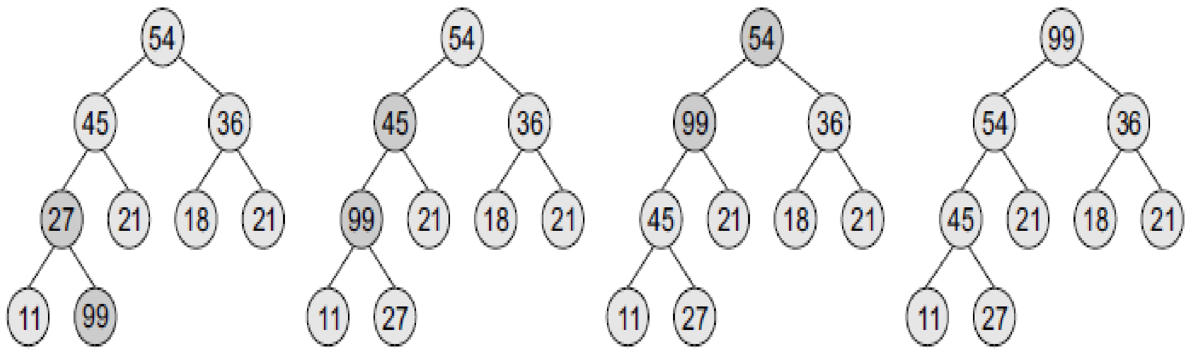
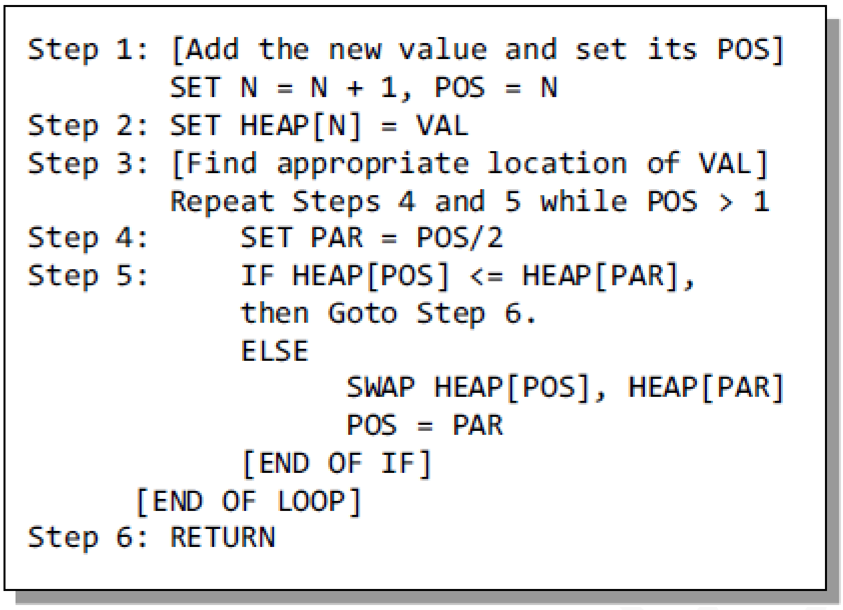


Figure 1.27 Heapify the binary heap

After discussing the concept behind inserting a new value in the heap, let us now look at the algorithm to do so as shown in Fig. 1.28.



```
Step 1: [Add the new value and set its POS]
        SET N = N + 1, POS = N
Step 2: SET HEAP[N] = VAL
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS > 1
Step 4:     SET PAR = POS/2
Step 5:     IF HEAP[POS] <= HEAP[PAR],
            then Goto Step 6.
            ELSE
                  SWAP HEAP[POS], HEAP[PAR]
                  POS = PAR
            [END OF IF]
      [END OF LOOP]
Step 6: RETURN
```

Figure 1.28 Algorithm to insert an element in a max heap

We assume that H with n elements is stored in array HEAP. VAL has to be inserted in HEAP. The location of VAL as it rises in the heap is given by POS, and PAR denotes the location of the parent of VAL.

Note that this algorithm inserts a single value in the heap. In order to build a heap, use this algorithm in a loop. For example, to build a heap with 9 elements, use a for loop that executes 9 times and in each pass, a single value is inserted.

The complexity of this algorithm in the average case is O(1). This is because a binary heap has O(log n) height. Since approximately 50% of the elements are leaves and 75% are in the bottom two levels, the new element to be inserted will only move a few levels upwards to maintain the heap.

In the worst case, insertion of a single value may take O(log n) time and, similarly, to build a heap of n elements, the algorithm will execute in O(n log n) time.

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).

Here, the value of root node = 54 and the value of the last node = 11. So, replace 54 with 11 and delete the last node.

Example: Consider the max heap H shown in Fig. 1.29 and delete the root node's value.

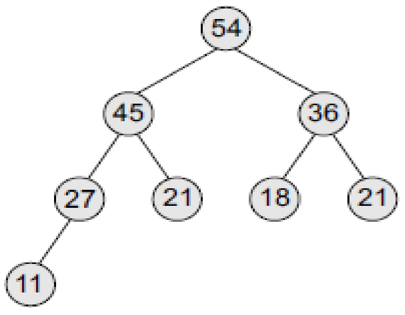
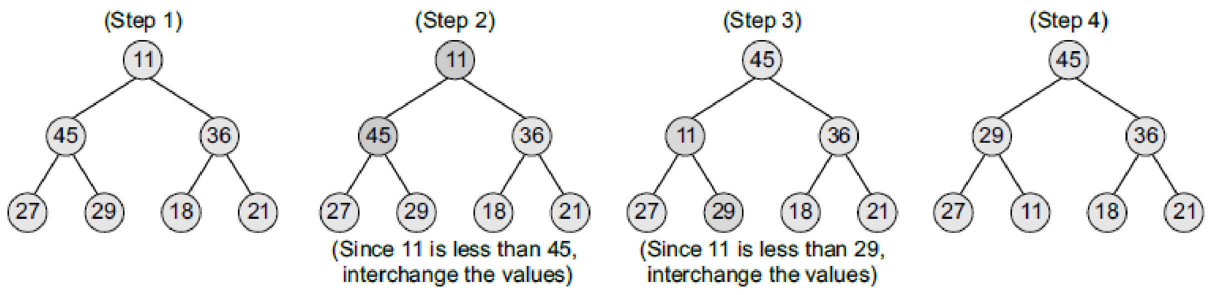


Figure 1.29 Binary heap

Solution:



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

Figure 1.30 Binary heap

After discussing the concept behind deleting the root element from the heap, let us look at the algorithm given in Fig. 1.31.

```
Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
         HEAP[PTR] >= HEAP[RIGHT]
              Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
              SWAP HEAP[PTR], HEAP[LEFT]
              SET PTR = LEFT
        ELSE
              SWAP HEAP[PTR], HEAP[RIGHT]
              SET PTR = RIGHT
        [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
```

Figure 1.31 Algorithm to delete the root element from a max heap

We assume that heap H with n elements is stored using a sequential array called HEAP. LAST is the last element in the heap and PTR, LEFT, and RIGHT denote the position of LAST and its left and right children respectively as it moves down the heap.

1.5 BALANCED BINARY TREES:

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. 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 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. A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

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

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

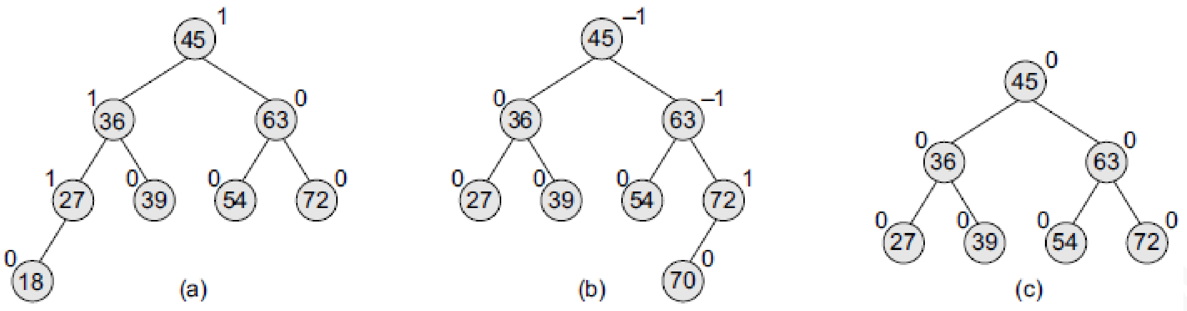


Figure 1.32 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Look at Fig. 1.32. Note that the nodes 18, 39, 54, and 72 have no children, so their balance factor = 0. Node 27 has one left child and zero right child. So, the height of left sub-tree = 1, whereas the height of right sub-tree = 0. Thus, its balance factor = 1. Look at node 36, it has a left sub-tree with height = 2, whereas the height of right sub-tree = 1. Thus, its balance factor = 2 – 1 = 1. Similarly, the balance factor of node 45 = 3 – 2 =1; and node 63 has a balance factor of 0 (1 – 1).

Now, look at Figs 1.32 (a) and (b) which show a right-heavy AVL tree and a balanced AVL tree.

The trees given in Fig. 1.32 are typical candidates of AVL trees because the balancing factor of every node is either 1, 0, or –1. However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done. The tree is rebalanced by performing rotation at the critical node. There are four types of rotations: LL rotation, RR rotation, LR rotation, and RL rotation. The type of rotation that has to be done will vary depending on the particular situation.

1) 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.

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

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

Consider the AVL tree given in Fig. 1.33.

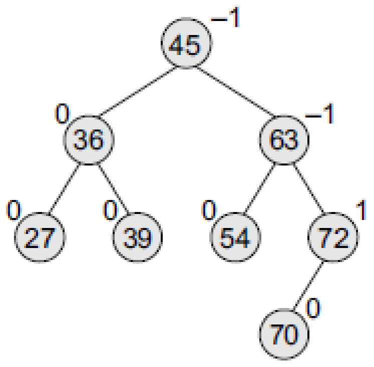


Figure 1.33 AVL tree

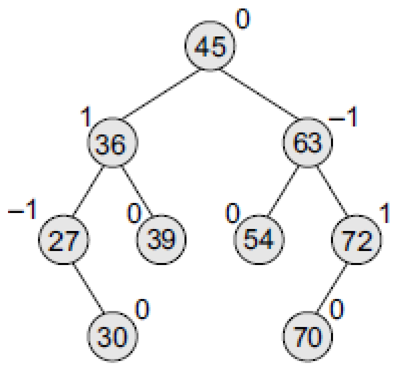


Figure 1.34 AVL tree after inserting a node with the value 30

If we insert a new node with the value 30, then the new tree will still be balanced and no rotations will be required in this case. Look at the tree given in Fig. 1.34 which shows the tree after inserting node 30.

Let us take another example to see how insertion can disturb the balance factors of the nodes and how rotations are done to restore the AVL property of a tree. Look at the tree given in Fig. 1.35.

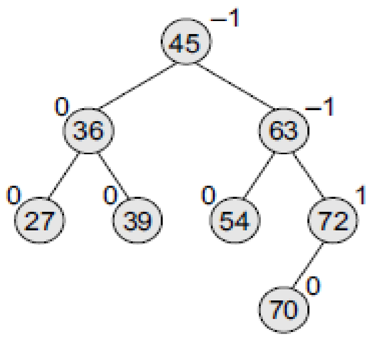


Figure 1.35 AVL tree

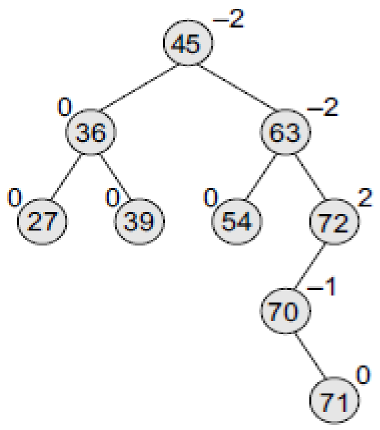


Figure 1.36 AVL tree after inserting a node with the value 71

After inserting a new node with the value 71, the new tree will be as shown in Fig. 1.36. 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. 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:

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

LL Rotation*:*

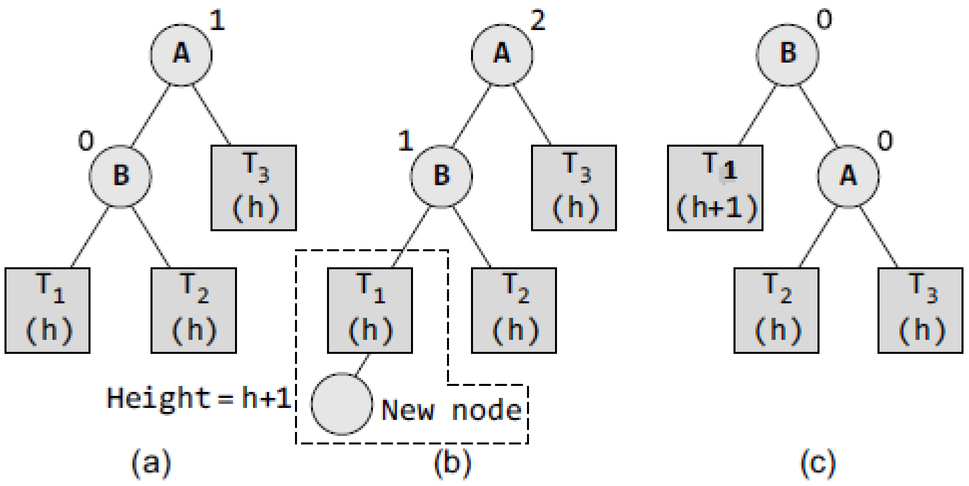


Figure 1.37 LL rotation in an AVL tree

Consider the tree given in Fig. 1.37 which shows an AVL tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply LL rotation as shown in tree (c).

While rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A. Note that the new node has now become a part of tree T1.

Example: Consider the AVL tree given in Fig. 1.38 and insert 18 into it.

Solution:

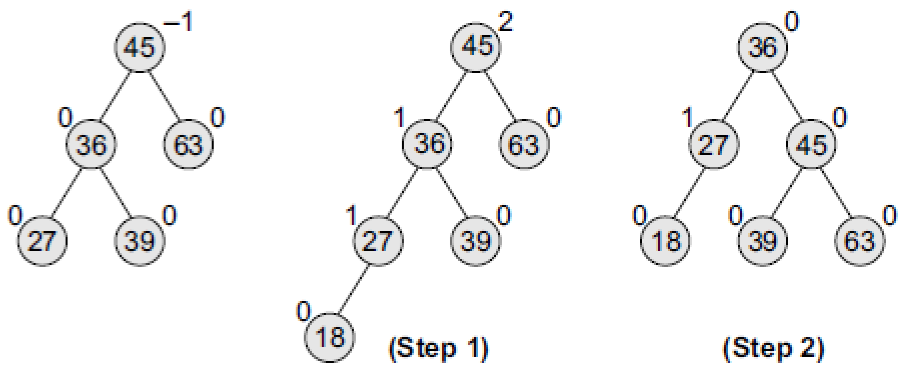


Figure 1.38 AVL tree

RR Rotation:

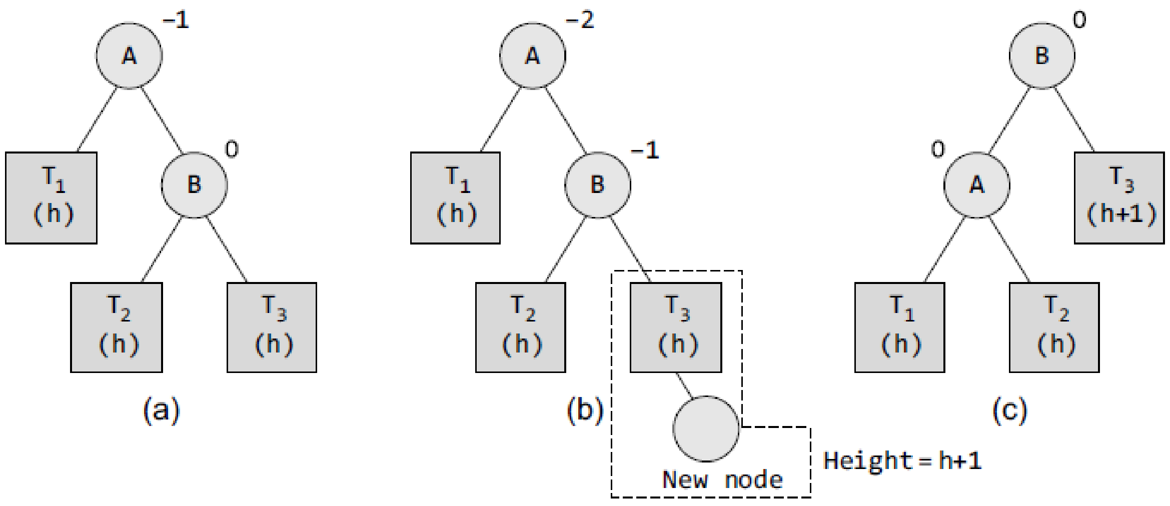


Figure 1.39 RR rotation in an AVL tree

Let us now discuss where and how RR rotation is applied. Consider the tree given in Fig. 1.39 which shows an AVL tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply RR rotation as shown in tree (c). Note that the new node has now become a part of tree T3.

While rotation, node B becomes the root, with A and T3 as its left and right child. T1 and T2 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.40 and insert 89 into it.

Solution:

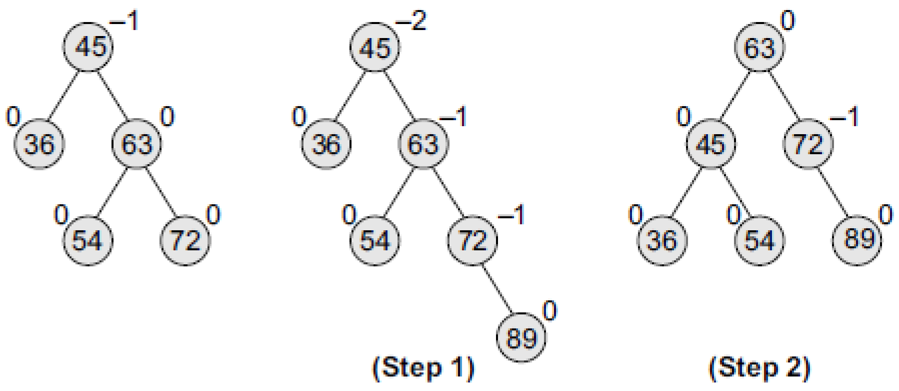


Figure 1.40 AVL tree

LR Rotation:

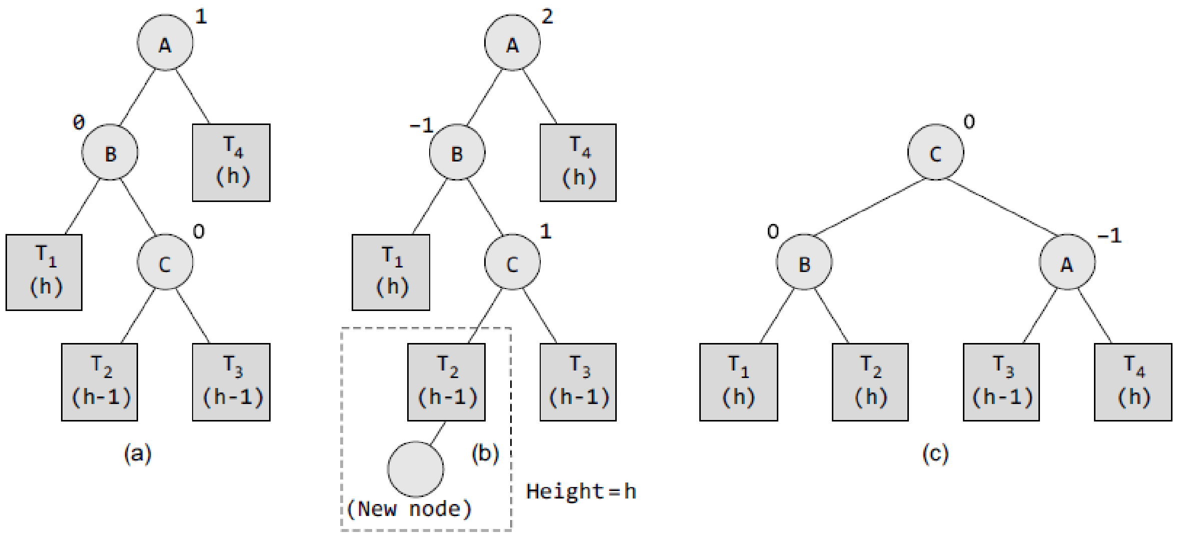


Figure 1.41 LR rotation in an AVL tree

Consider the AVL tree given in Fig. 1.41 and see how LR rotation is done to rebalance the tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0 or 1), so we apply LR rotation as shown in tree (c). Note that the new node has now become a part of tree T2. While rotation, node C becomes the root, with B and A as its left and right children. Node B has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node A.

Example: Consider the AVL tree given in Fig. 1.42 and insert 37 into it.

Solution:

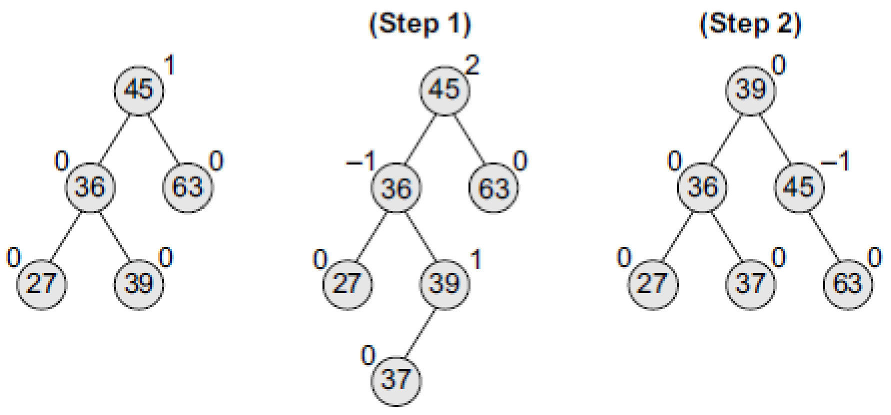


Figure 1.42 AVL tree

RL Rotation:

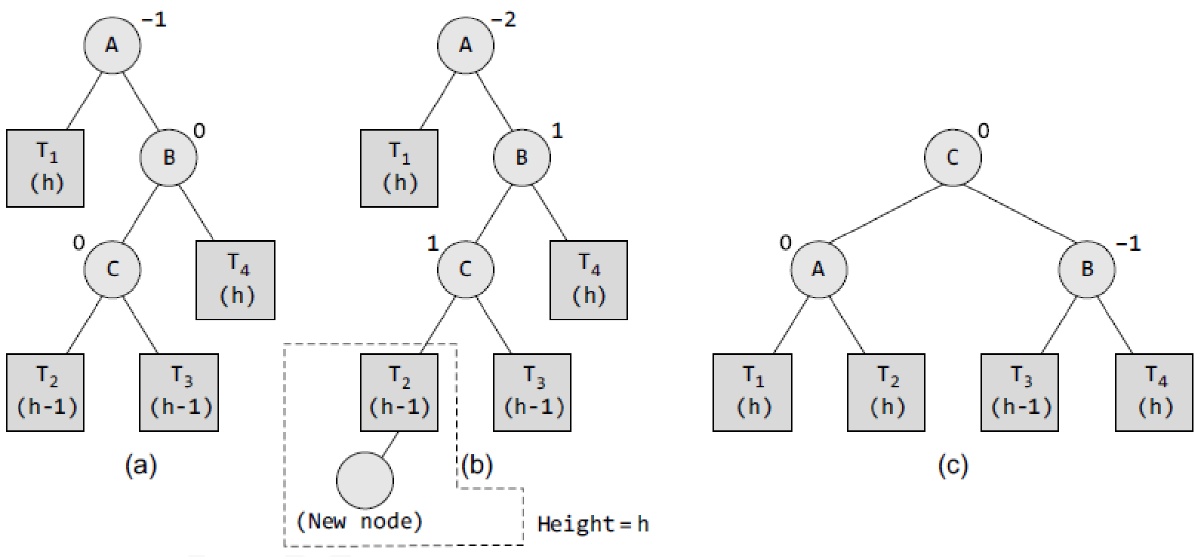
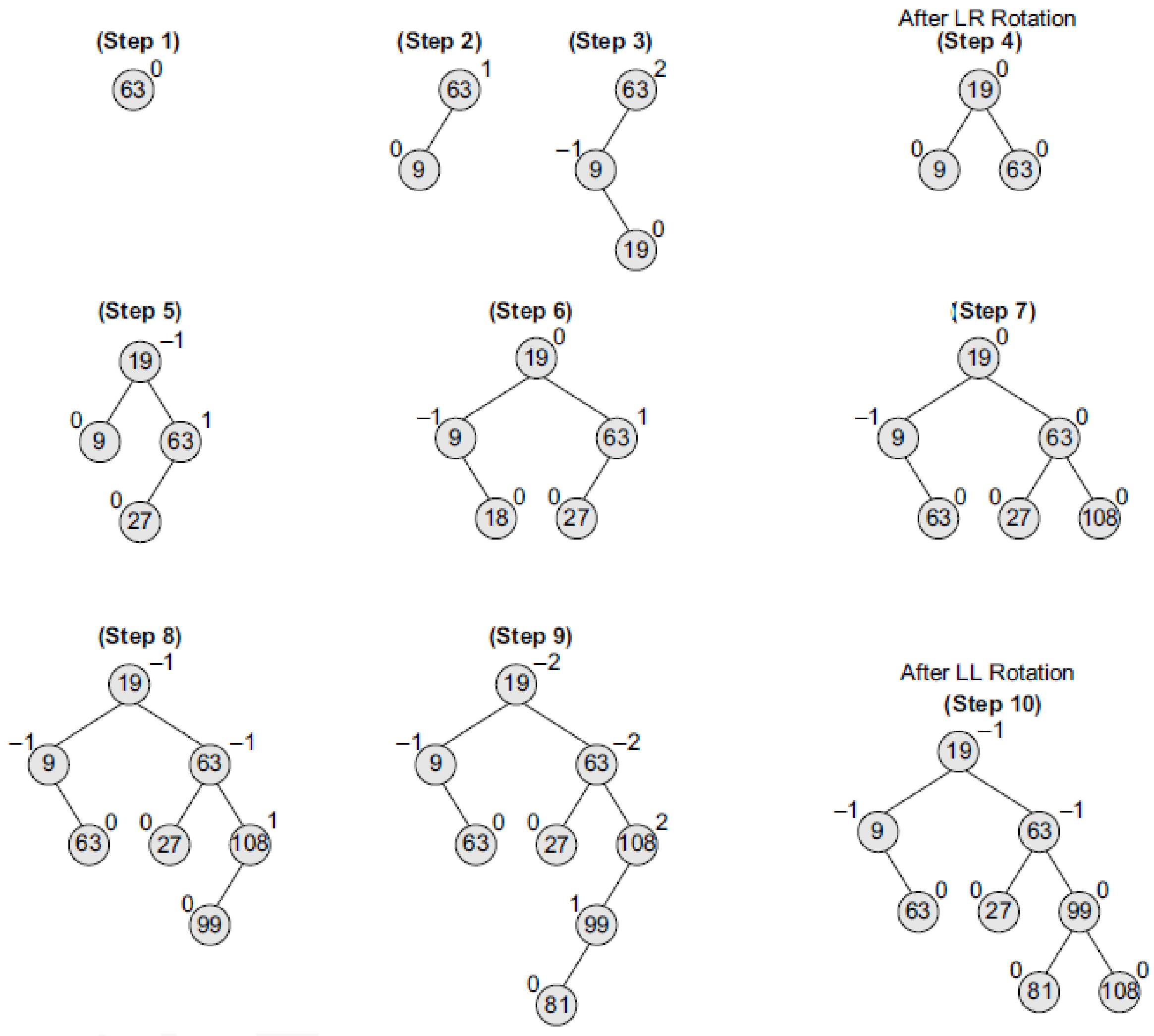


Figure 1.43 RL rotation in an AVL tree

Consider the AVL tree given in Fig. 1.43 and see how RL rotation is done to rebalance the tree. Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1), so we apply RL rotation as shown in tree (c). Note that the new node has now become a part of tree T2. While rotation, node C becomes the root, with A and B as its left and right children. Node A has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node B.

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

Solution:



2) 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. On deletion of node X from the AVL tree, if node A becomes the critical node (closest ancestor node on the path from X to the root node that does not have its balance factor as 1, 0, or –1), then the type of rotation depends on whether X is in the left sub-tree of A or in its right sub-tree. If the node to be deleted is present in the left sub-tree of A, then L rotation is applied, else if X is in the right sub-tree, R rotation is performed. Further, there are three categories of L and R rotations. The variations of L rotation are L–1, L0, and L1 rotation. Correspondingly for R rotation, there are R0, R–1, and R1 rotations. In this section, we will discuss only R rotation. L rotations are the mirror images of R rotations.

R0 Rotation:

Let B be the root of the left or right sub-tree of A (critical node). R0 rotation is applied if the balance factor of B is 0. This is illustrated in Fig. 1.44.

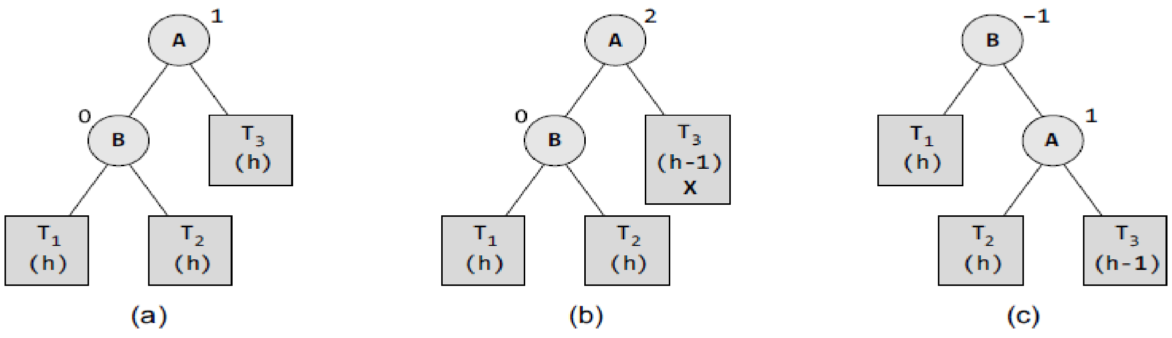


Figure 1.44 R0 rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 0, we apply R0 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.45 and delete 72 from it.

Solution:

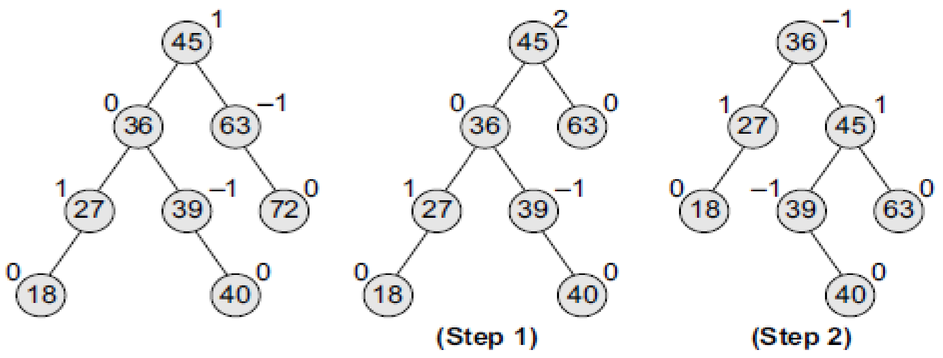


Figure 1.45 AVL tree

R1 Rotation:

Let B be the root of the left or right sub-tree of A (critical node). R1 rotation is applied if the balance factor of B is 1. Observe that R0 and R1 rotations are similar to LL rotations; the only difference is that R0 and R1 rotations yield different balance factors. This is illustrated in Fig. 1.46.

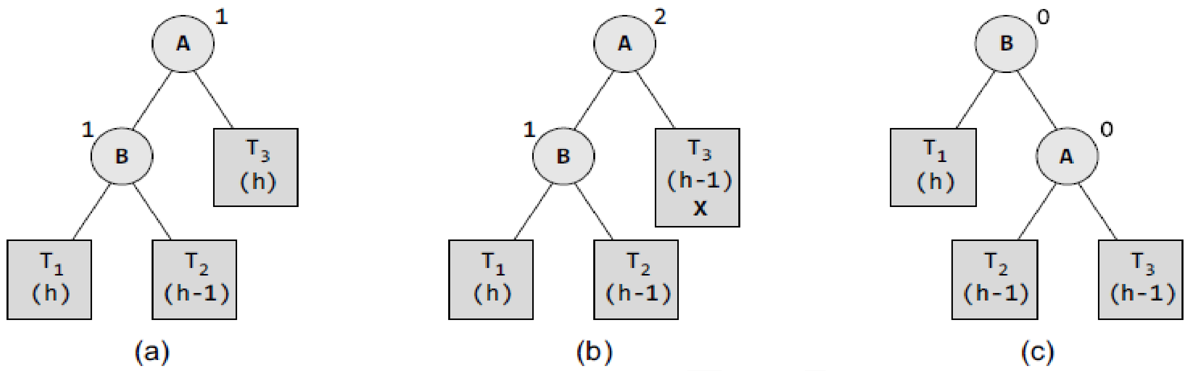


Figure 1.46 R1 rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 1, we apply R1 rotation as shown in tree (c).

During the process of rotation, node B becomes the root, with T1 and A as its left and right children. T2 and T3 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.47 and delete 72 from it.

Solution:

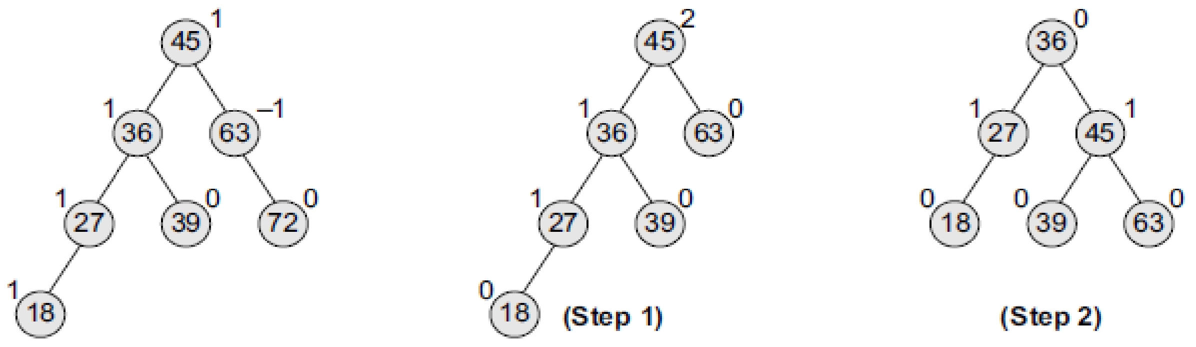


Figure 1.47 AVL tree

R–1 Rotation:

Let B be the root of the left or right sub-tree of A (critical node). R–1 rotation is applied if the balance factor of B is –1. Observe that R–1 rotation is similar to LR rotation. This is illustrated in Fig. 1.48

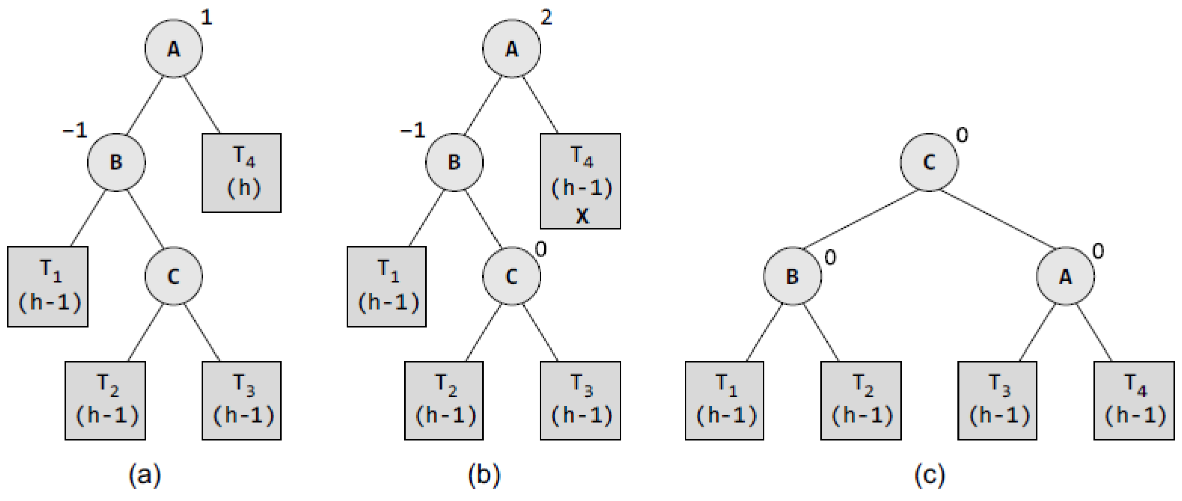


Figure 1.48 R-1 rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0 or 1). Since the balance factor of node B is –1, we apply R–1 rotation as shown in tree (c). While rotation, node C becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

Example: Consider the AVL tree given in Fig. 1.49 and delete 72 from it.

Solution:

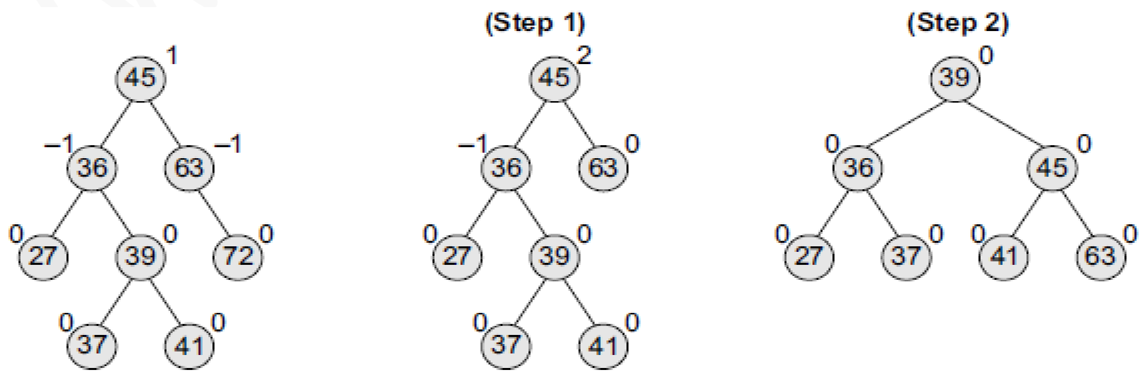
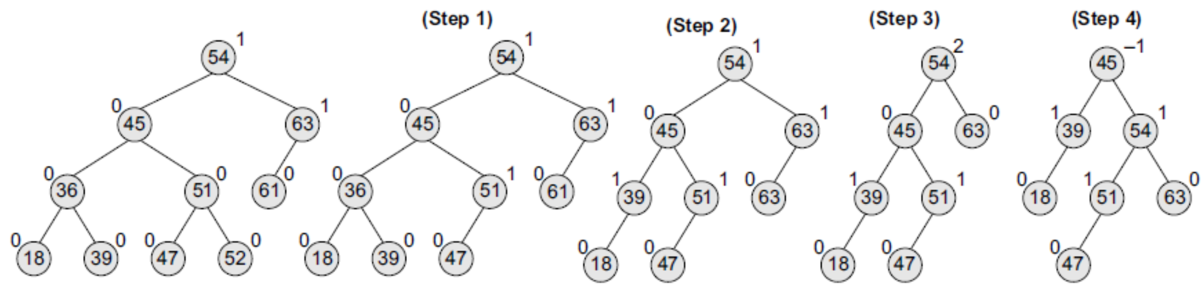


Figure 1.49 AVL tree

Example: Delete nodes 52, 36, and 61 from the given AVL tree.

Solution:


(Step 1)
(Step 2)
(Step 3)
(Step 4)