

Trees

INTRODUCTION:

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.

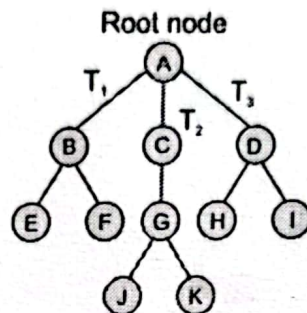


Figure 1: Tree

Figure: 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.

Basic Terminology:

Root node: The root node R is the topmost node in the tree. If $R = \text{NULL}$, 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, 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, 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, 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.

TYPES OF TREES:

Trees are of following 6 types:

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

1. 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. General trees which have 3 sub-trees per node are called ternary trees. However, the number of sub-trees for any node may be variable. For example, a node can have 1 sub-tree, whereas some other node can have 3 sub-trees.

2. 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.

A forest can also be defined as an ordered set of zero or more general trees.

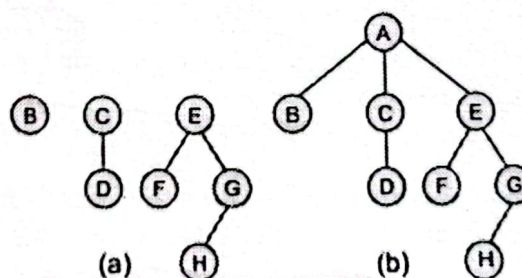


Fig.2 Forest and corresponding trees

3. 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**. 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.

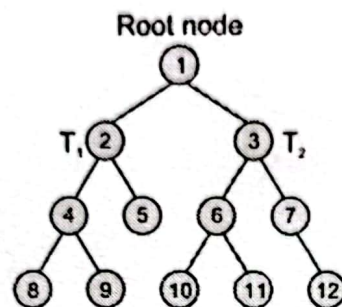
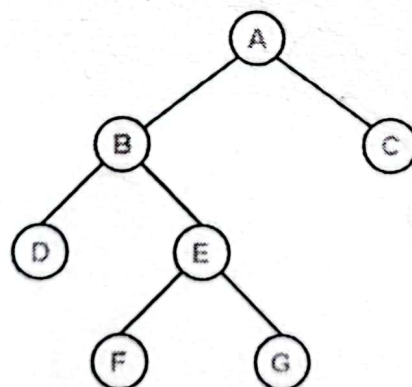


Fig.3 Binary tree

Figure 3 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.

Full / Strictly Binary Tree-

A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**. Full binary tree is also called as **strictly binary tree**.

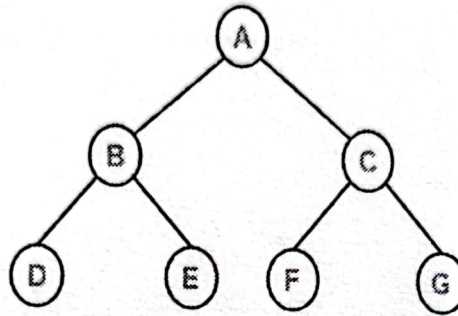


Complete / Perfect Binary Tree-

A **complete binary tree** is a binary tree that satisfies the following 2 properties-

- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.

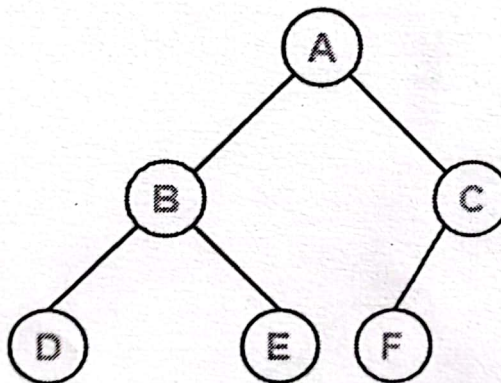
Complete binary tree is also called as **Perfect binary tree**.



Almost Complete Binary Tree-

An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-

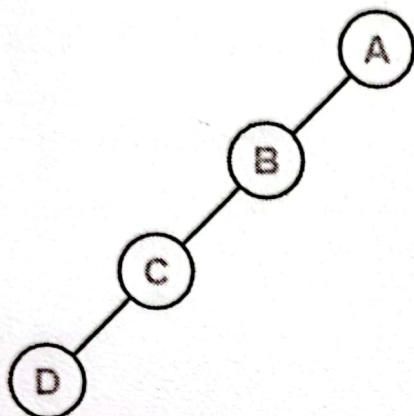
- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.



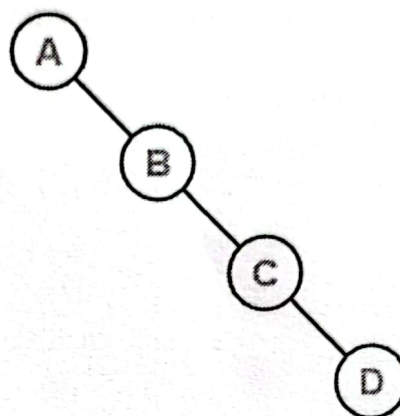
Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.

- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.



Left Skewed Binary Tree



Right Skewed Binary Tree

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

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

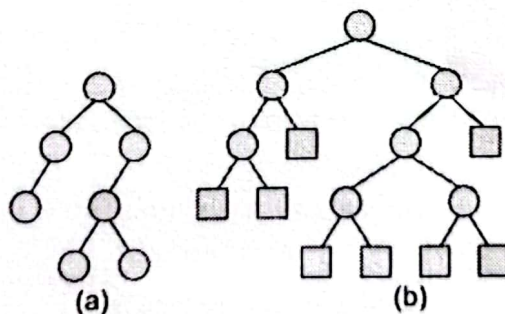


Fig. (a) Binary tree and (b) extended binary tree.

Terminology:

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

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

Similar binary trees: Two binary trees T and T' are said to be similar if both these trees have the same structure. Figure 9.5 shows two similar binary trees.

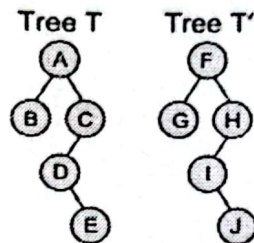


Fig. Similar binary tree

Copies Two binary trees: T and T' are said to be copies if they have similar structure and if they have same content at the corresponding nodes.

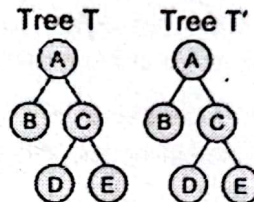


Fig. T' copy tree of T .

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

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

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

Note: A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. The height of a binary tree with n nodes is at least $\log_2(n+1)$ and at most n .

In-degree/out-degree of a node: It is the number of edges arriving at a node. The root node is the only node that has an in-degree equal to zero. Similarly, out-degree of a node is the number of edges leaving that node.

Representation of Binary Trees in the Memory:

TRAVERSING A BINARY TREE:

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a nonlinear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. They are:

1. Pre-order Traversal
2. In-order Traversal
3. Post-order Traversal
4. Level-order Traversal

1. Pre-order Traversal

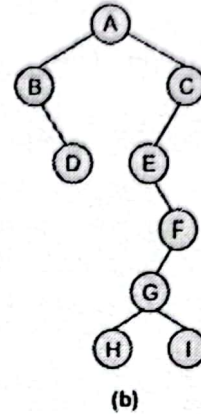
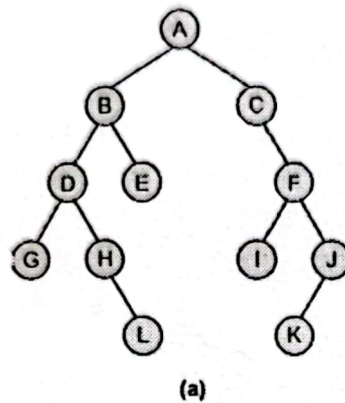
To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node,
 2. Traversing the left sub-tree, and finally
 3. Traversing the right sub-tree
- Pre-order traversal is also called as **depth-first traversal**.
 - Pre-order traversal algorithms are used to extract a prefix notation from an expression tree.

Algorithm for pre-order traversal:

```
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
```

For example: find the sequence of nodes that will be visited using pre-order traversal algorithm.



Solution:

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

TRAVERSAL ORDER: A, B, D, C, 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:

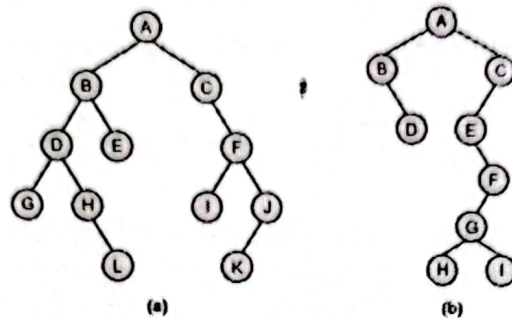
1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

➤ In-order traversal is also called as symmetric traversal.

Algorithm for in-order traversal:

```
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
```

For example: Find the sequence of nodes that will be visited using in-order traversal algorithm.



Solution:

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

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

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.

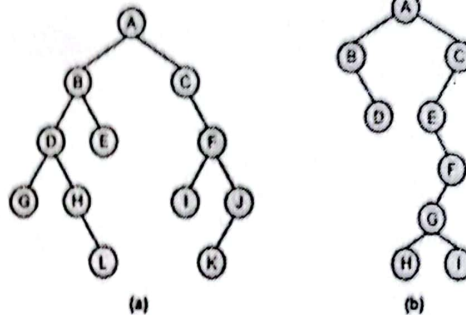
➤ Post-order traversals are used to extract postfix notation from an expression tree.

Algorithm for post-order traversal:

```

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
  
```

For example: Give the sequence of nodes that will be visited using post-order traversal algorithm.



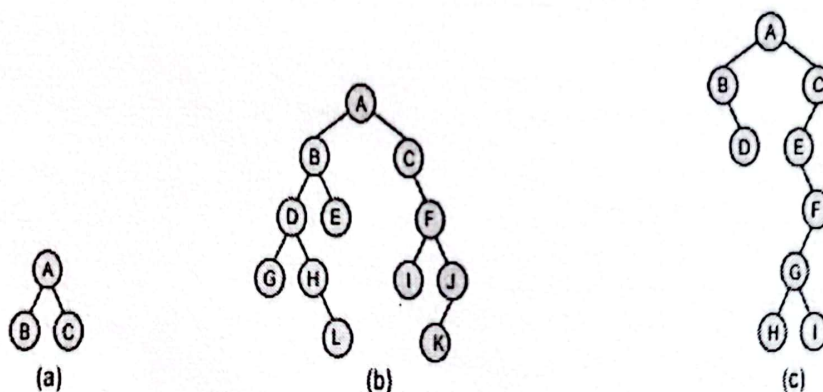
Solution:

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

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

4. 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. and note the level order of these trees.



TRAVERSAL ORDER:
A, B, and C

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

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

5. Constructing a Binary Tree from Traversal Results:

We can construct a binary tree if we are given at least two traversal results. The first traversal must be the in-order traversal and the second can be either pre-order or post-order traversal. The in-order traversal result will be used to determine the left and the right child nodes, and the pre-order/post-order can be used to determine the root node.

For example, consider the traversal results given below:

In-order Traversal: D B E A F C G

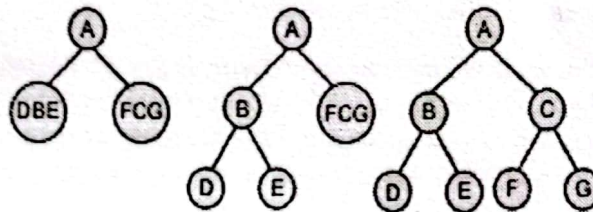
Pre-order Traversal: A B D E C F G

Here, we have the in-order traversal sequence and pre-order traversal sequence. Follow the steps given below to construct the tree:

Step 1 Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.

Step 2 Elements on the left side of the root node in the in-order traversal sequence form the left sub-tree of the root node. Similarly, elements on the right side of the root node in the in-order traversal sequence form the right sub-tree of the root node.

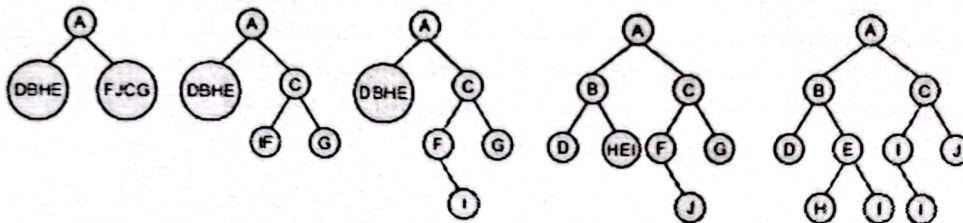
Step 3 Recursively select each element from pre-order traversal sequence and create its left and right sub-trees from the in-order traversal sequence.



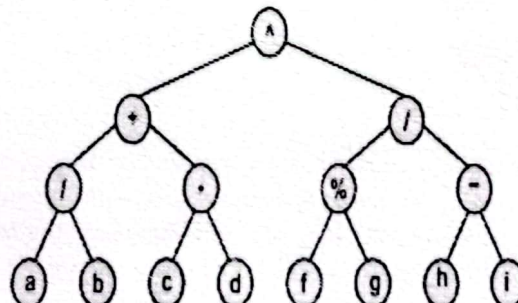
For Example:

In-order Traversal: D B H E I A F J C G

Post order Traversal: D H I E B J F G C A



Exercise: Given the binary tree, write down the pre-order, post-order and in-order traversal of expression tree that it represents.



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.

For example:

BY: Abhimanyu Yadav@DSA

(12)

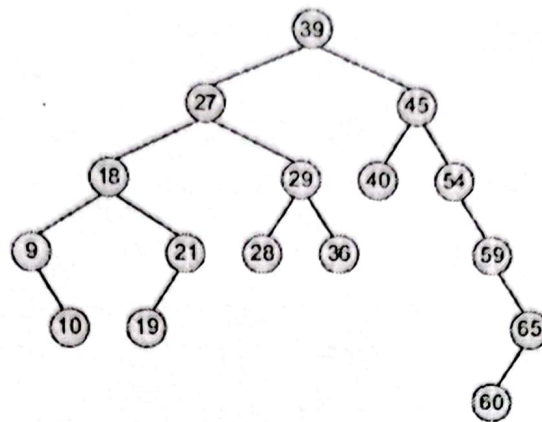


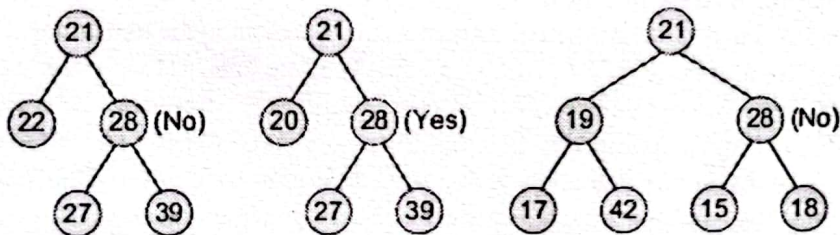
Figure: Binary search tree

Look at Fig. 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.

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

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

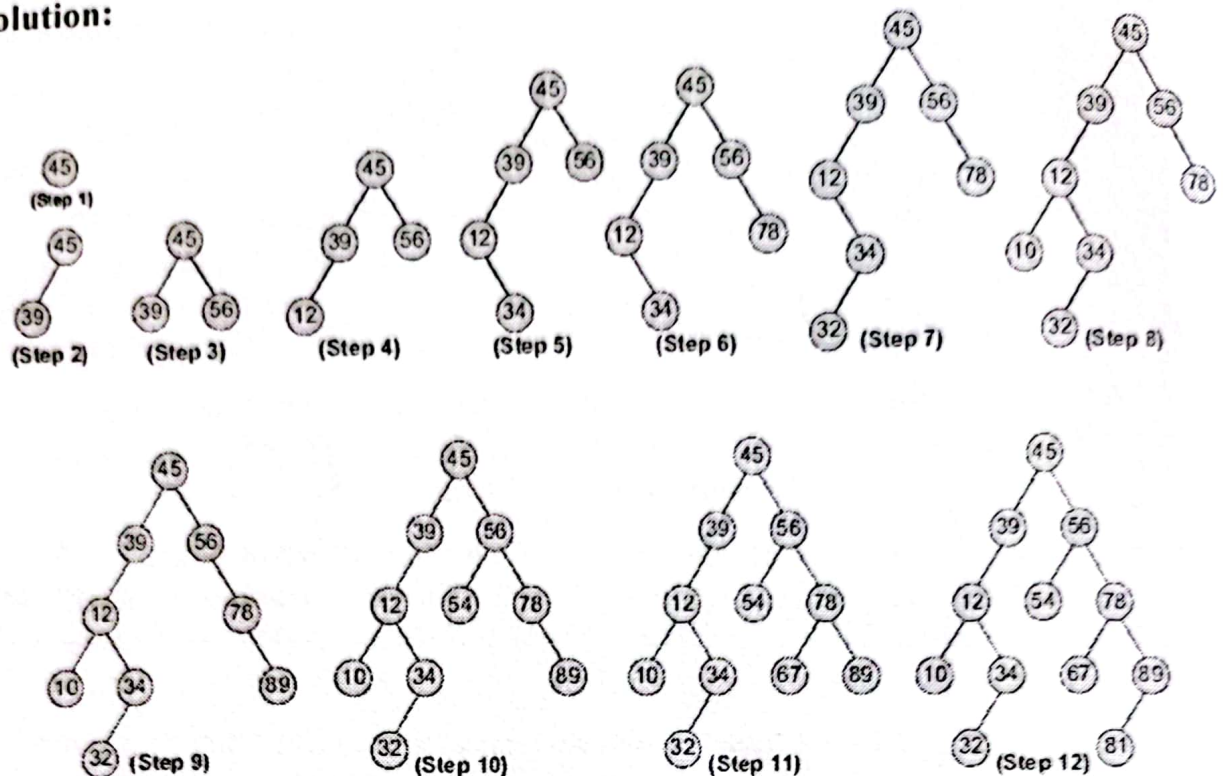
Example 1. State whether the binary trees in Fig. are binary search trees or not.



Example 2. Create a binary search tree using the following data elements:

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

Solution:



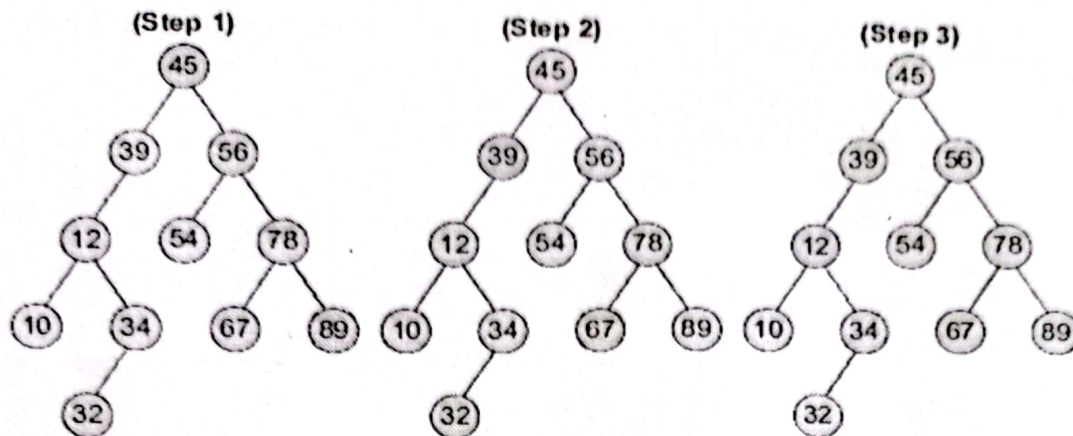
OPERATIONS ON BINARY SEARCH TREES:

In this section, we will discuss the different operations that are performed on a binary search tree.

1. Searching for a Node in a Binary Search Tree

The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node. The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message. However, if there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

Example 3. Searching a node with value 12 in the given binary search tree.



Algorithm to search for a given value in a 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

2. Inserting a New Node in a Binary Search Tree:

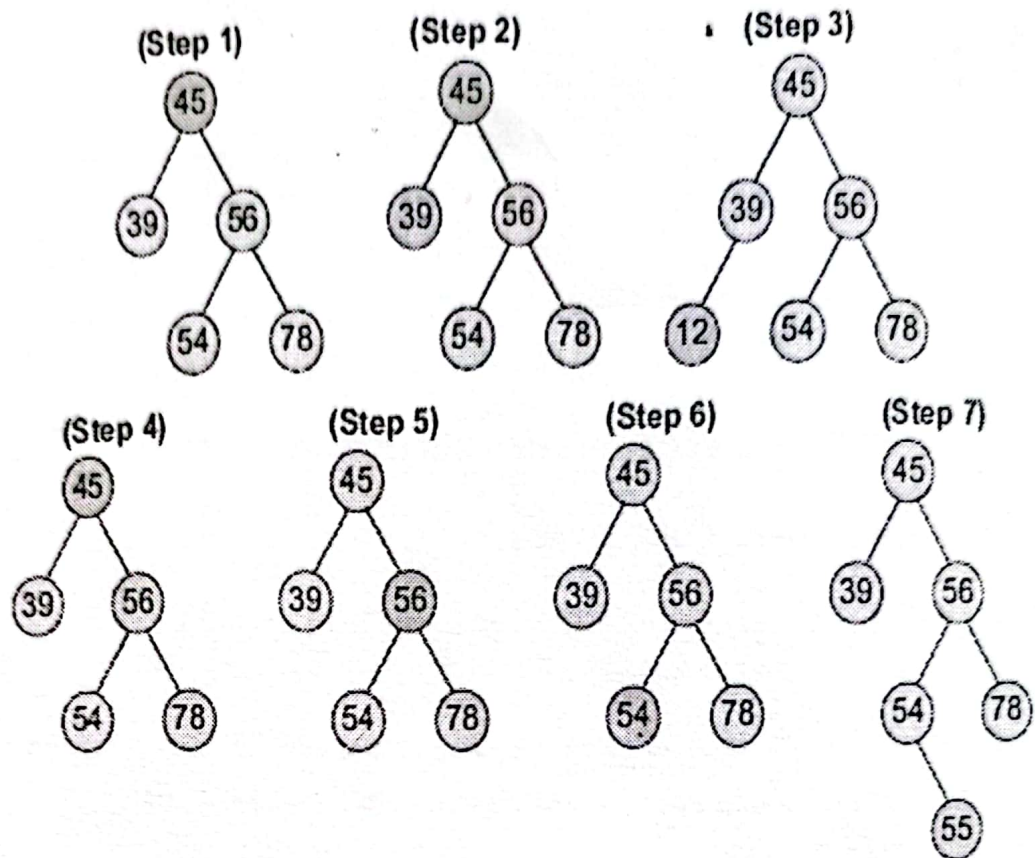
The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree.

The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position.

The 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.

Example 4. Inserting nodes with values 12 and 55 in the given binary search tree.



Algorithm to insert a given value 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]

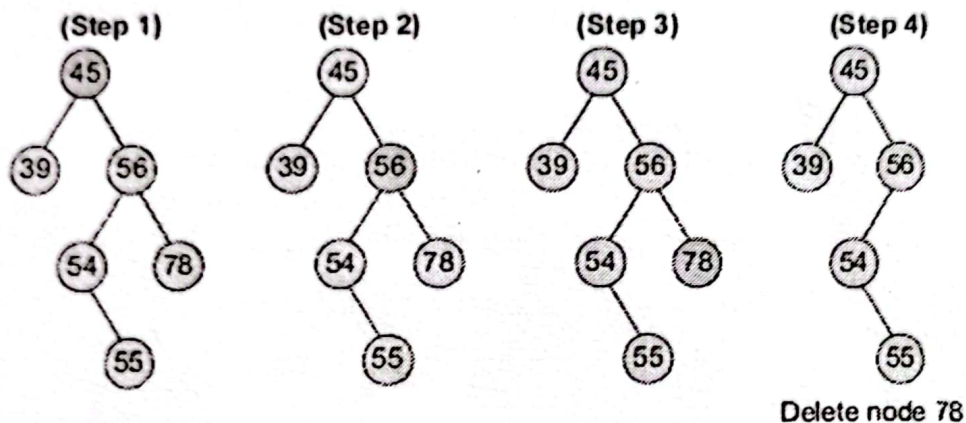
Step 2: END

3. Deleting a Node from a Binary Search Tree:

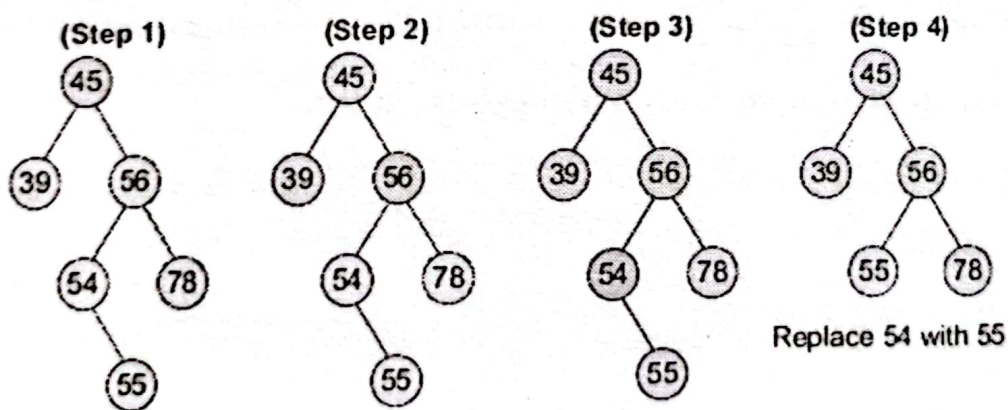
The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process. We will take up three cases:

- Case 1: Deleting a Node that has No Children.
- Case 2: Deleting a Node with One Child.
- Case 3: Deleting a Node with Two Children.

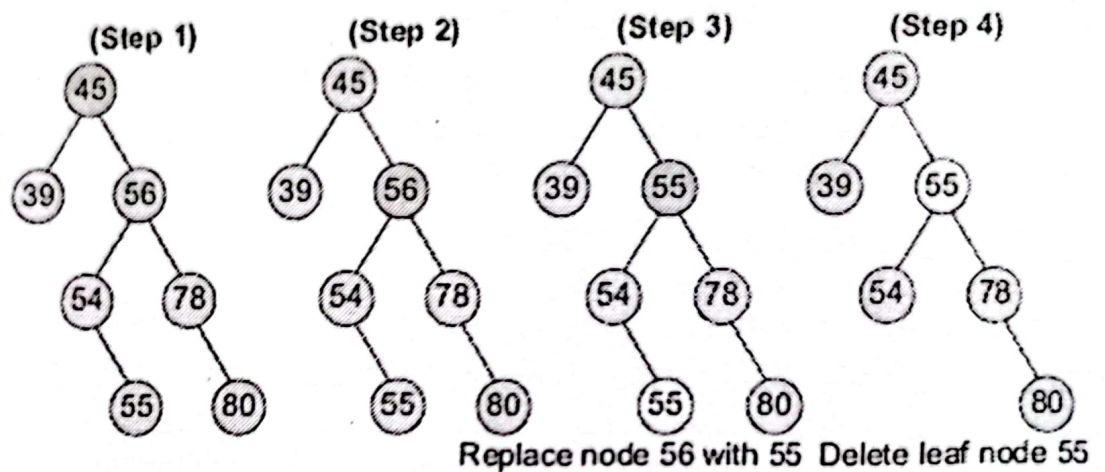
Example 5. Deleting node 78 from the given binary search tree.



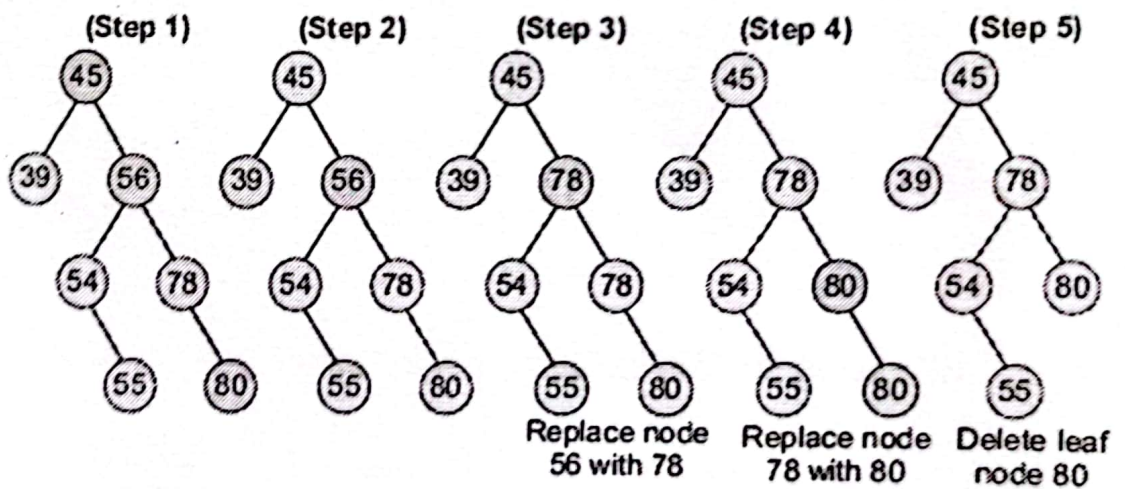
Example 6. Deleting node 54 from the given binary search tree.



Example 7. Deleting node 56 from the given binary search tree.



Example 8. Deleting node 56 from the given binary search tree.



Algorithm to delete a node from a binary search 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

AVL Tree Data structure

AVL tree is a height balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right sub trees of every node in the tree is either -1, 0 or +1.

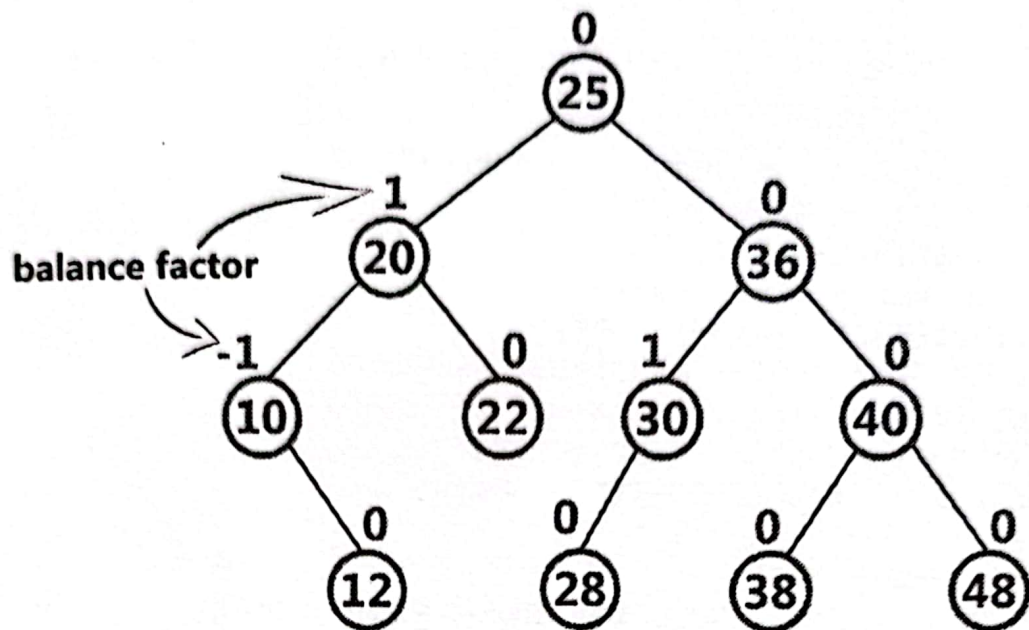
An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

We calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL Tree

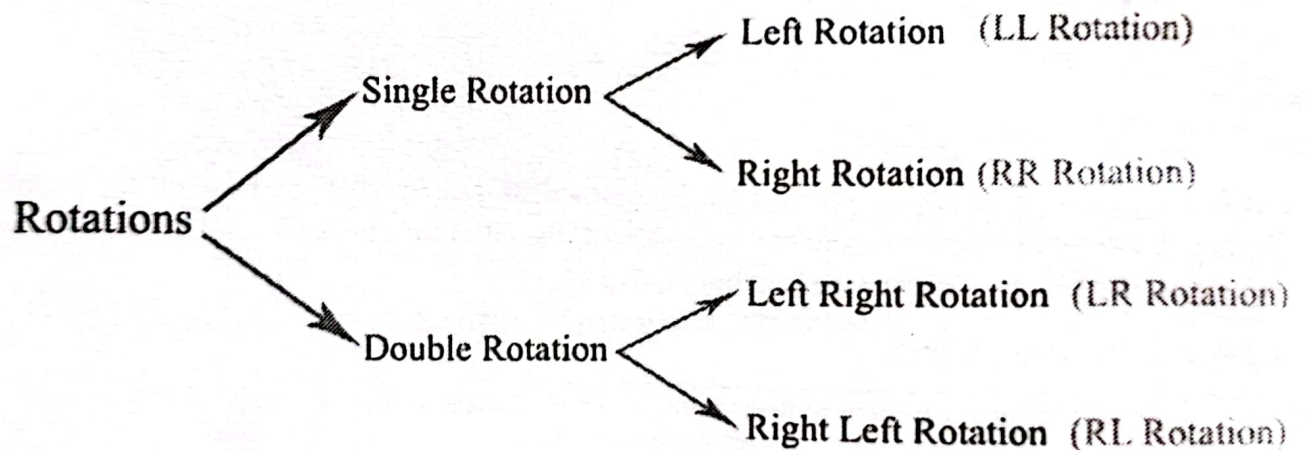


Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

Rotation operations are used to make the tree balanced.

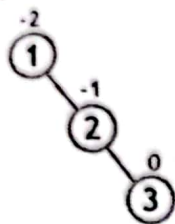
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types.

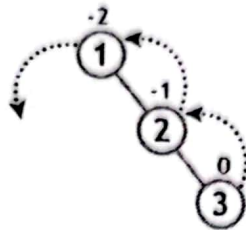


Single Left Rotation (LL Rotation)

insert 1, 2 and 3



Tree is imbalanced



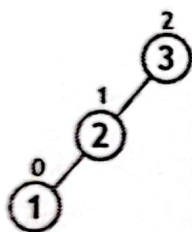
To make balanced we use LL Rotation which moves nodes one position to left



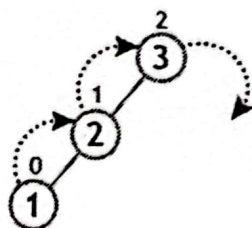
After LL Rotation
Tree Is Balanced

Single Right Rotation (RR Rotation)

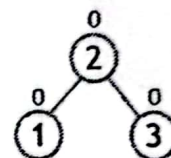
insert 3, 2 and 1



Tree is imbalanced
because node 3 has balance factor 2



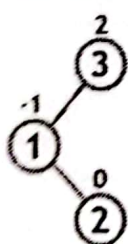
To make balanced we use RR Rotation which moves nodes one position to right



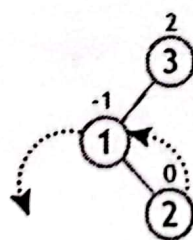
After RR Rotation
Tree Is Balanced

Left Right Rotation (LR Rotation)

insert 3, 1 and 2

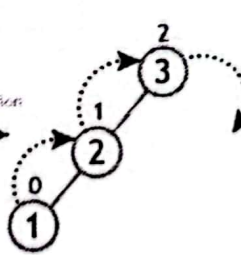


Tree is imbalanced
because node 3 has balance factor 2



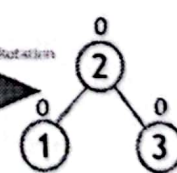
LL Rotation

After LL Rotation



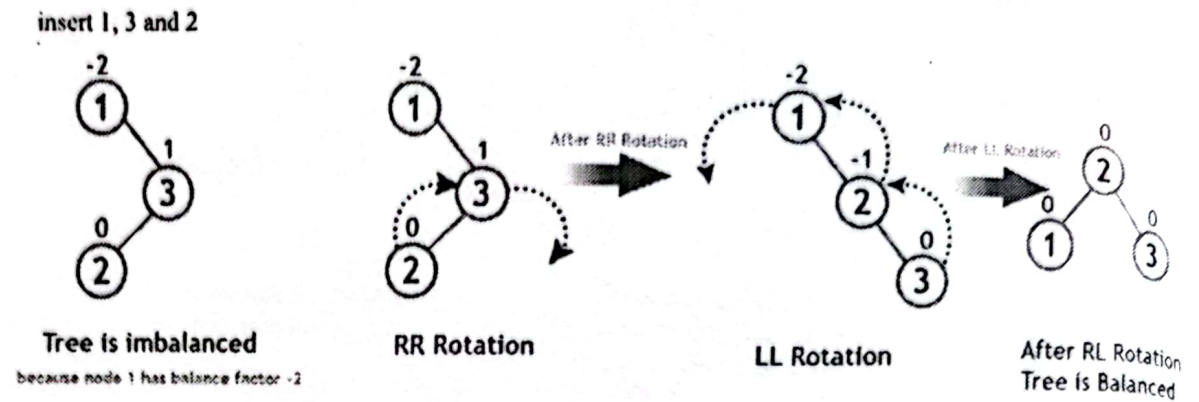
RR Rotation

After RR Rotation

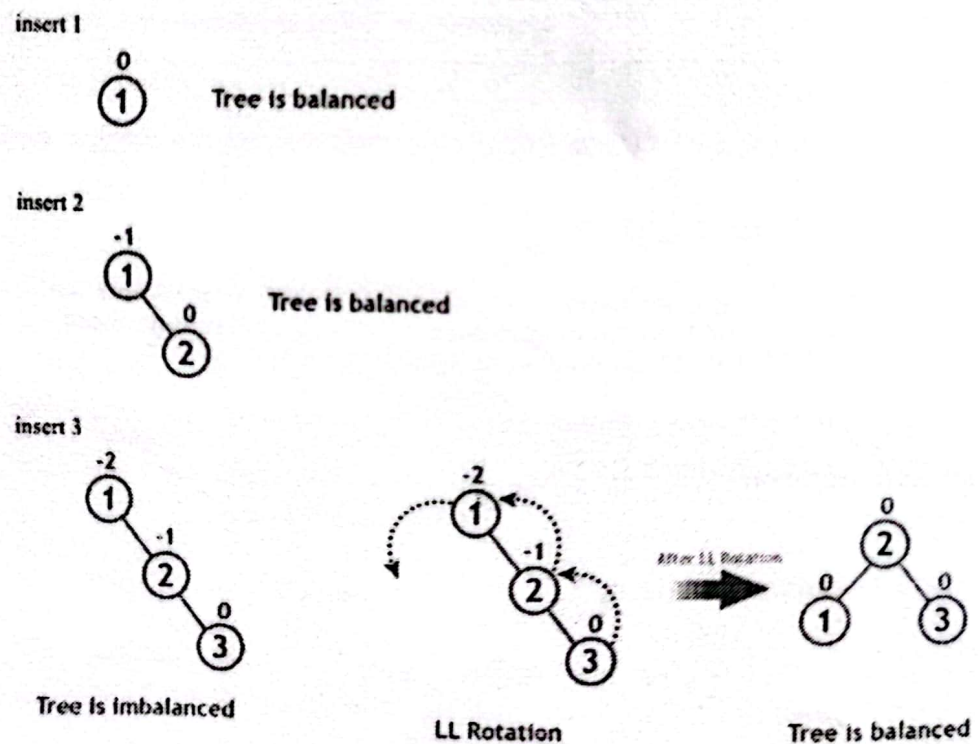


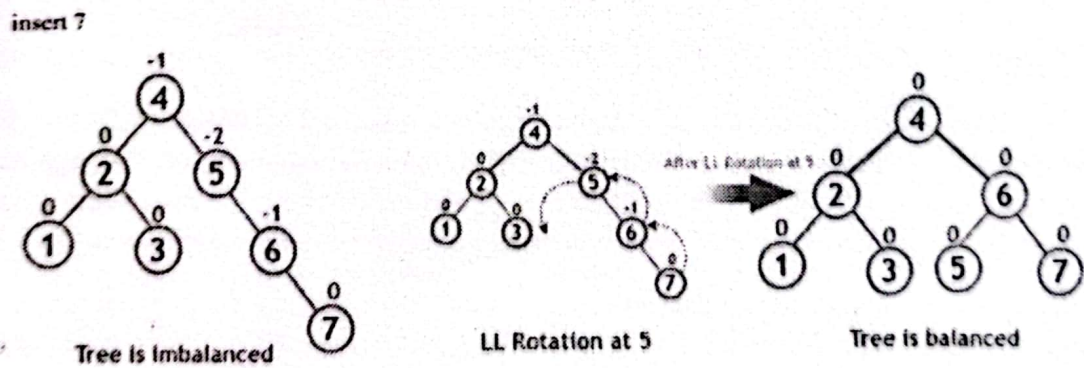
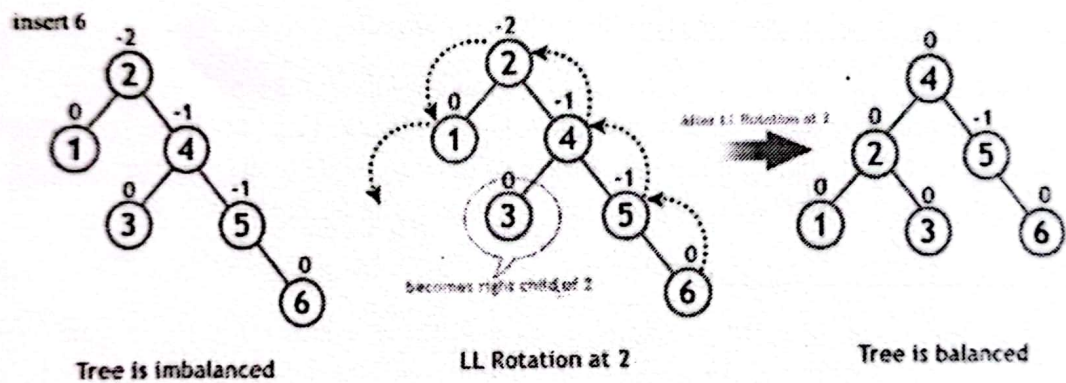
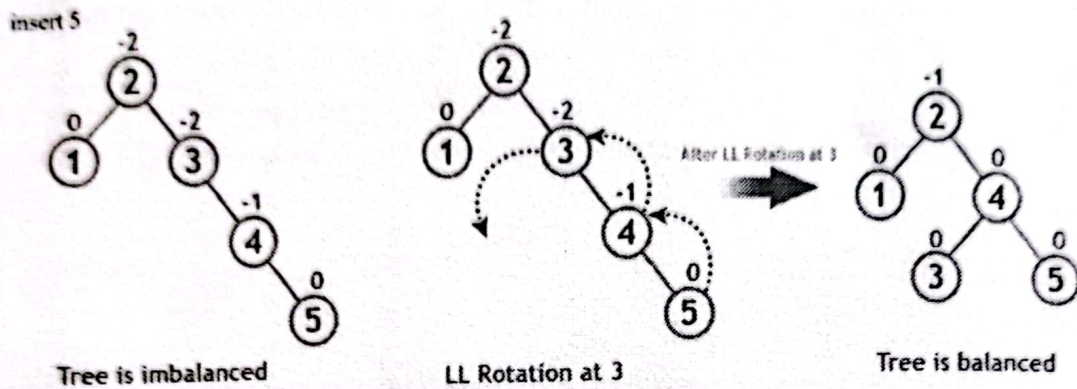
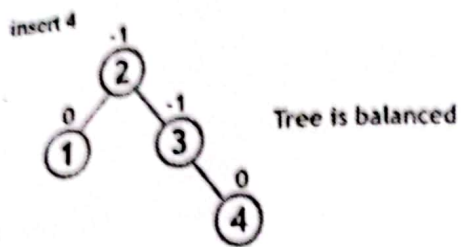
After LR Rotation
Tree Is Balanced

Right Left Rotation (RL Rotation)

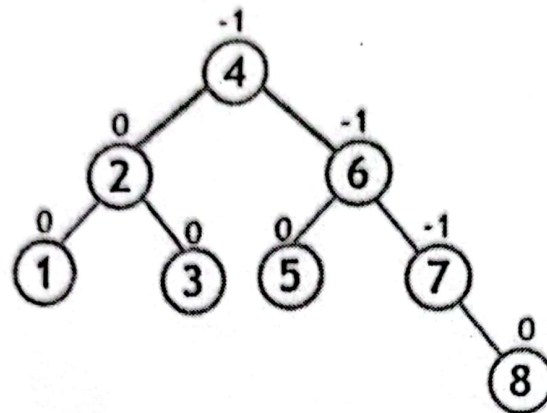


Example: Construct an AVL Tree by inserting numbers from 1 to 8.





insert 8



Tree is balanced

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.

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. 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 otherwise balance the AVL tree using different rotation.

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.

B TREES:

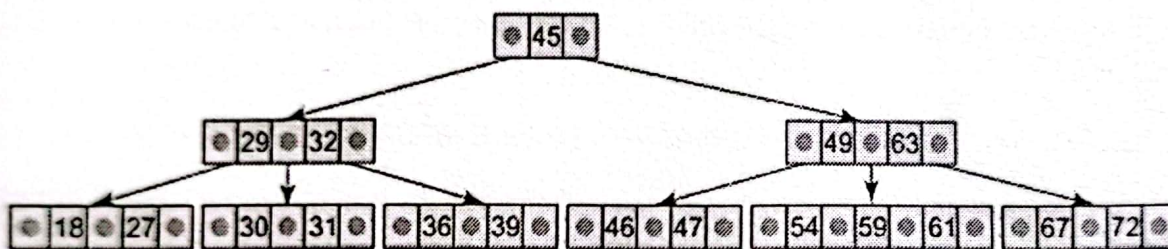
A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order m can have a maximum of $m-1$ keys and m pointers to its sub-trees. A B tree may contain a large number of key values and pointers to sub-trees. Storing a large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

An internal node in the B tree can have n number of children, where $0 \leq n \leq m$. It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least $m/2$ children. As B tree of order 4 is:



While performing insertion and deletion operations in a B tree, the number of child nodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split.

Applications of B Trees

A database is a collection of related data. The prime reason for using a database is that it stores organized data to facilitate its users to update, retrieve, and manage the data. The data stored in the database may include names, addresses, pictures, and numbers. For example, a teacher may wish to maintain a database of all the students that includes the names, roll numbers, date of birth, and marks obtained by every student.

We take a large value of m mainly because of three reasons:

1. Disk access is very slow. We should be able to fetch a large amount of data in one disk access.
2. Disk is a block-oriented device. That is, data is organized and retrieved in terms of blocks. So while using a B tree (generalized M-way search tree), a large value of m is used so that one single node of the tree can occupy the entire block. In other words, m represents the maximum number of data items that can be stored in a single block. m is maximized to speed up processing. More the data stored in a block, lesser the time needed to move it into the main memory.
3. A large value minimizes the height of the tree. So, search operation becomes really fast.

HUFFMAN'S TRE:

Huffman coding is an entropy encoding algorithm developed by David A. Huffman that is widely used as a lossless data compression technique. The Huffman coding algorithm uses a variable length code table to encode a source character where the variable-length code table is derived on the basis of the estimated probability of occurrence of the source character.

The key idea behind Huffman algorithm is that it encodes the most common characters using shorter strings of bits than those used for less common source characters.

The algorithm works by creating a binary tree of nodes that are stored in an array. A node can be either a leaf node or an internal node. Initially, all the nodes in the tree are at the leaf level and store the source character and its frequency of occurrence (also known as weight).

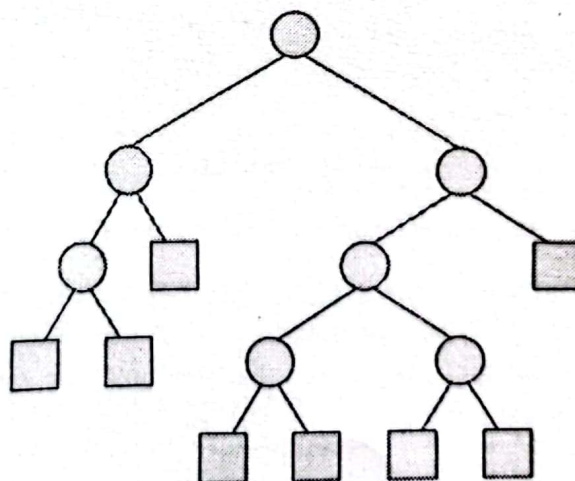
While the internal node is used to store the weight and contains links to its child nodes, the external node contains the actual character. Conventionally, a '0' represents following the left child and a '1' represents following the right child. A finished tree that has n leaf nodes will have $n - 1$ internal nodes.

Let us first learn how to calculate the length of the paths in the tree.

The external path length of a binary tree is defined as the sum of all path lengths summed over each path from the root to an external node.

The internal path length is also defined in the same manner. The internal path length of a binary tree is defined as the sum of all path lengths summed over each path from the root to an internal node.

Example: Binary Tree



The internal path length, $L_i = 0 + 1 + 2 + 1 + 2 + 3 + 3 = 12$

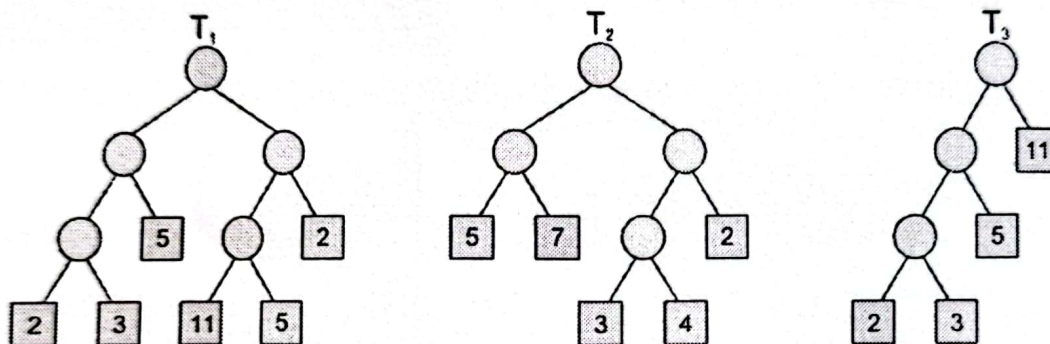
The external path length, $L_e = 2 + 3 + 3 + 2 + 4 + 4 + 4 + 4 = 26$

Note that, $L_i + 2 * n = 12 + 2 * 7 = 12 + 14 = 26 = L_e$

Thus, $L_i + 2n = L_e$, where n is the number of internal nodes. Now if the tree has n external nodes and each external node is assigned a weight, then the weighted path length P is defined as the sum of the weighted path lengths.

Therefore, $P = w_1 L_1 + w_2 L_2 + \dots + w_n L_n$
 where w_i and L_i are the weight and path length of an external node w_i .

Example: Consider the trees T_1 , T_2 , and T_3 given below, calculate their weighted external path lengths.



Binary tree

Solution:

Weighted external path length of T_1 can be given as,

$$P_1 = 2 \cdot 3 + 3 \cdot 3 + 5 \cdot 2 + 11 \cdot 3 + 5 \cdot 3 + 2 \cdot 2 = 6 + 9 + 10 + 33 + 15 + 4 = 77$$

Weighted external path length of T_2 can be given as,

$$P_2 = 5 \cdot 2 + 7 \cdot 2 + 3 \cdot 3 + 4 \cdot 3 + 2 \cdot 2 = 10 + 14 + 9 + 12 + 4 = 49$$

Weighted external path length of T_3 can be given as,

$$P_3 = 2 \cdot 3 + 3 \cdot 3 + 5 \cdot 2 + 11 \cdot 1 = 6 + 9 + 10 + 11 = 36$$

Technique:

Given n nodes and their weights, the Huffman algorithm is used to find a tree with a minimum weighted path length. The process essentially begins by creating a new node whose children are the two nodes with the smallest weight, such that the new node's weight is equal to the sum of the

children's weight. That is, the two nodes are merged into one node. This process is repeated until the tree has only one node. Such a tree with only one node is known as the Huffman tree.

The Huffman algorithm can be implemented using a priority queue in which all the nodes are placed in such a way that the node with the lowest weight is given the highest priority.

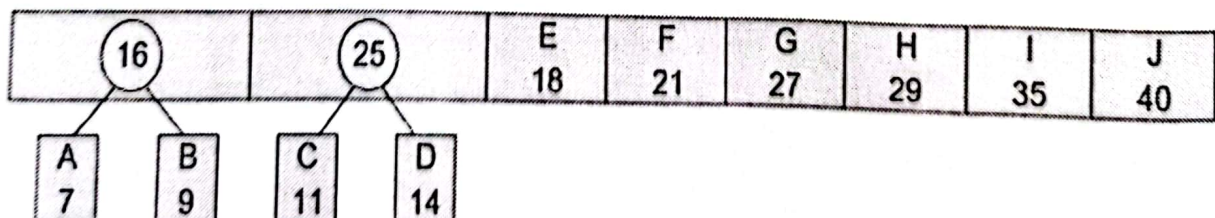
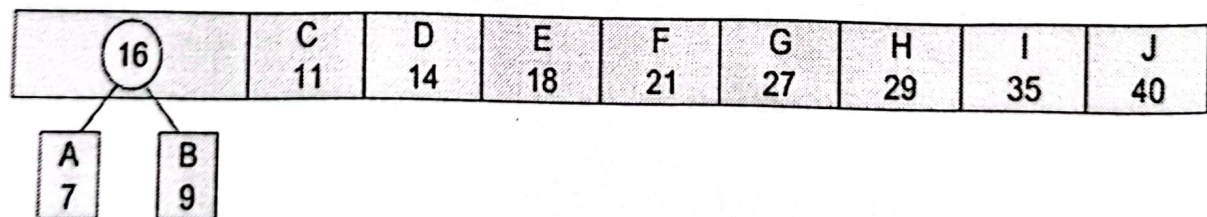
Huffman Algorithm:

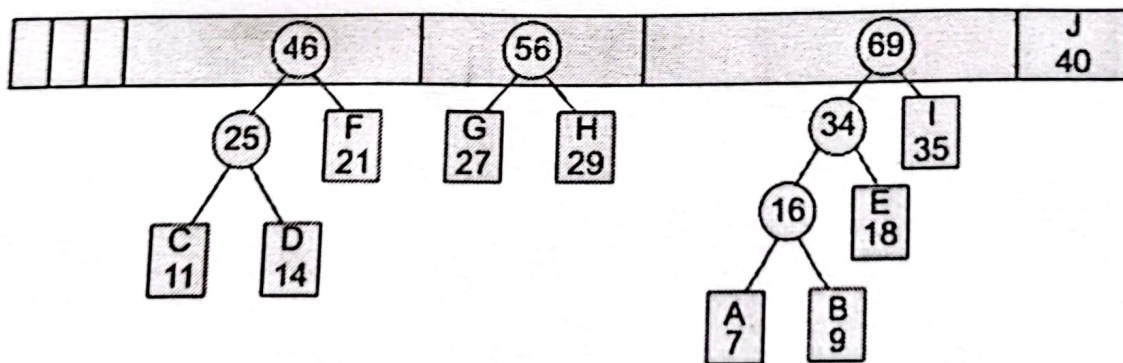
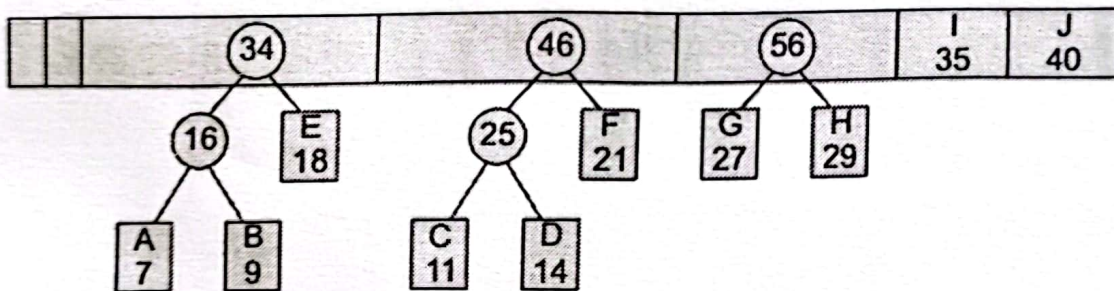
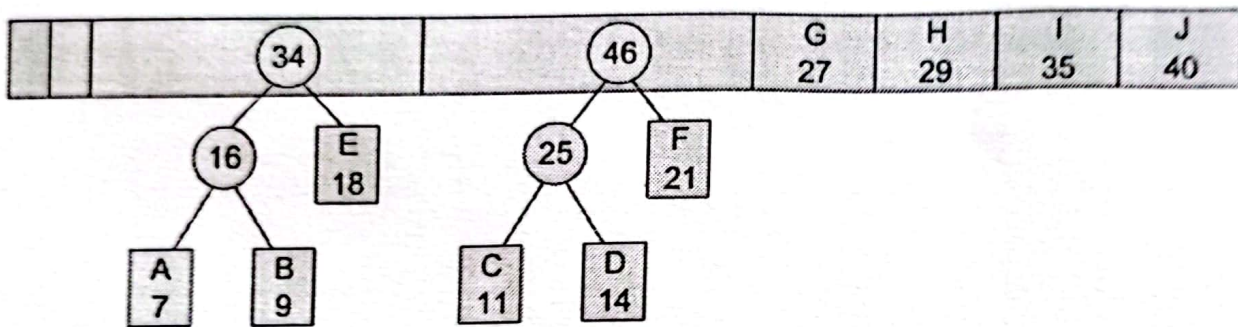
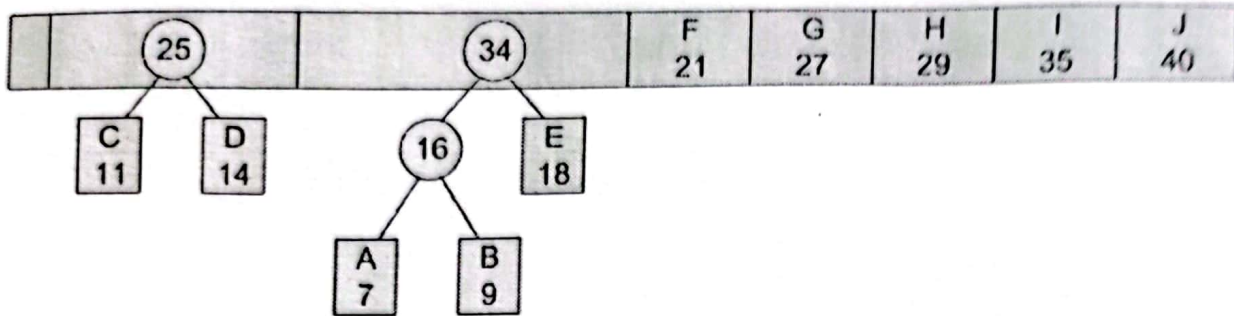
- Step 1: Create a leaf node for each character. Add the character and its weight or frequency of occurrence to the priority queue.
- Step 2: Repeat Steps 3 to 5 while the total number of nodes in the queue is greater than 1.
- Step 3: Remove two nodes that have the lowest weight (or highest priority).
- Step 4: Create a new internal node by merging these two nodes as children and with weight equal to the sum of the two nodes' weights.
- Step 5: Add the newly created node to the queue.

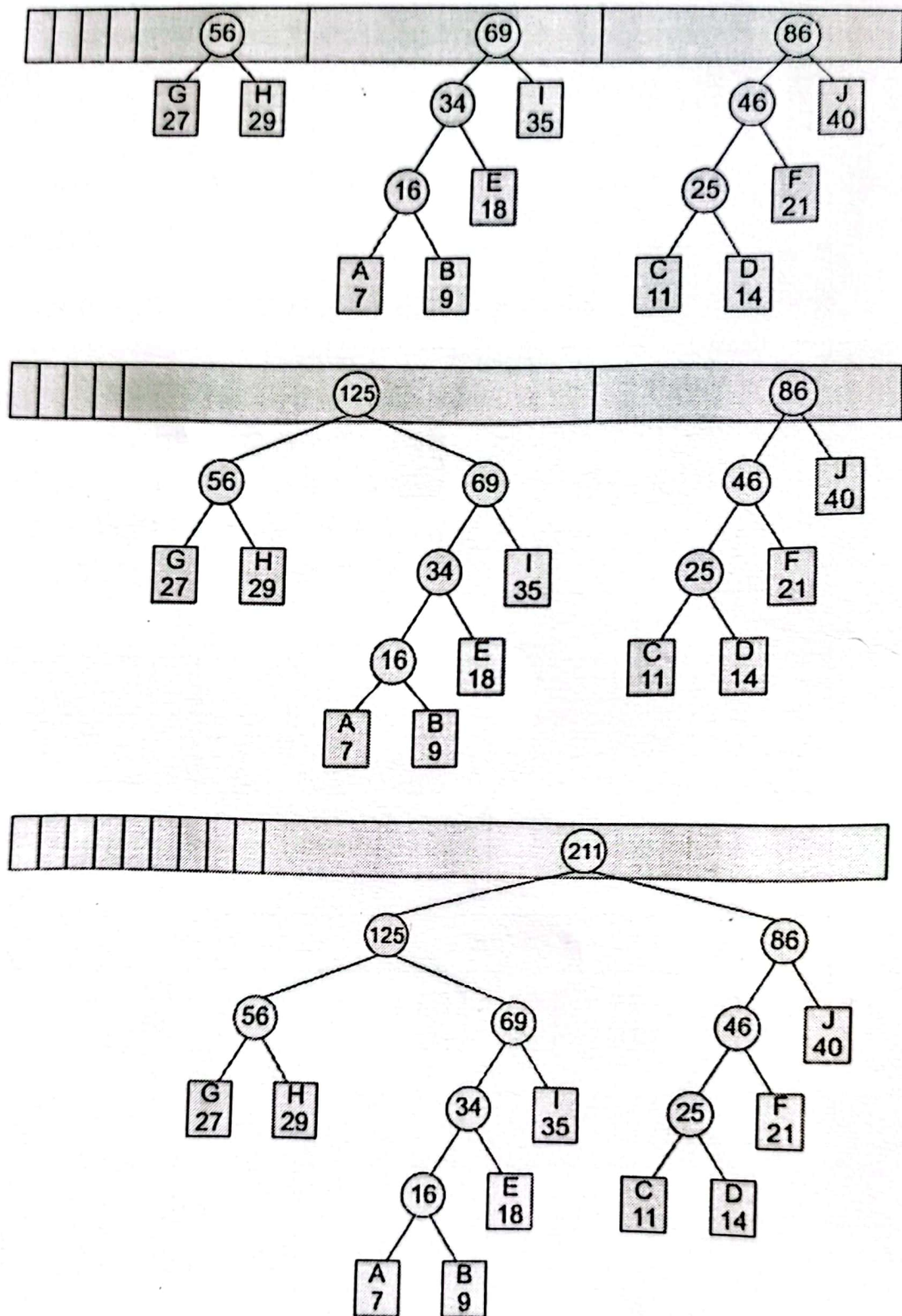
Example: Create a Huffman tree with the following nodes arranged in a priority queue.

A	B	C	D	E	F	G	H	I	J
7	9	11	14	18	21	27	29	35	40

Solution:







Data Coding:

When we want to code our data (character) using bits, then we use r bits to code 2^r characters.

For example, if $r=1$, then two characters can be coded. If these two characters are A and B, then A can be coded as 0 and B can be coded as 1 and vice versa.

The range of characters that can be coded by using $r=2$ and $r=3$.

Code	Character	Code	Character
00	A	000	A
01	B	001	B
10	C	010	C
11	D	011	D
		100	E
		101	F
		110	G
		111	H

Now, if we have to code the data string ABBBBBBAACDEFGGGGH, then the corresponding code would be:

0000010010010010010010000000000001001110010111011011011011

- This coding scheme has a fixed-length code. Variable-length coding is preferred over fixed-length coding because it requires lesser number of bits to encode the same data.

For variable-length encoding, we first build a Huffman tree. First, arrange all the characters in a priority queue in which the character with the lowest frequency of occurrence has the highest priority. Then, create a Huffman tree.

- ❖ In the Huffman tree, circles contain the cumulative weights of their child nodes. Every left branch is coded with 0 and every right branch is coded with 1. So, the characters A, E, R, W, X, Y, and Z are coded as:

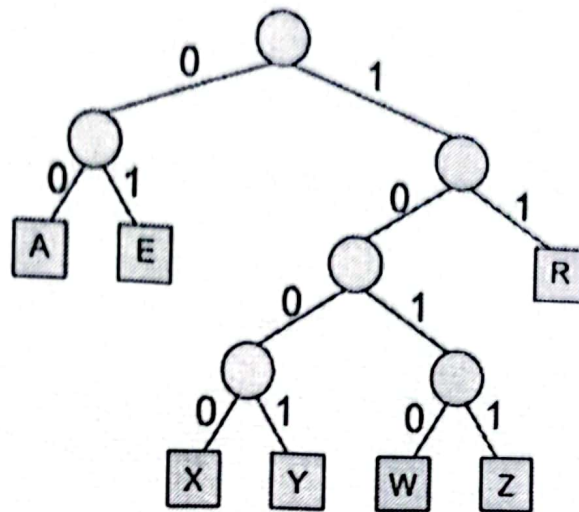


Fig: Huffman Tree

Characters with their codes:

Character	Code
A	00
E	01
R	11
W	1010
X	1000
Y	1001
Z	1011

Thus, we see that frequent characters have a shorter code and infrequent characters have a longer code.

APPLICATIONS OF TREES:

- Trees are used to store simple as well as complex data. Here simple means an integer value, character value and complex data means a structure or a record.
- Trees are an important data structure used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.