

UNIT-3 CHAPTER: 1

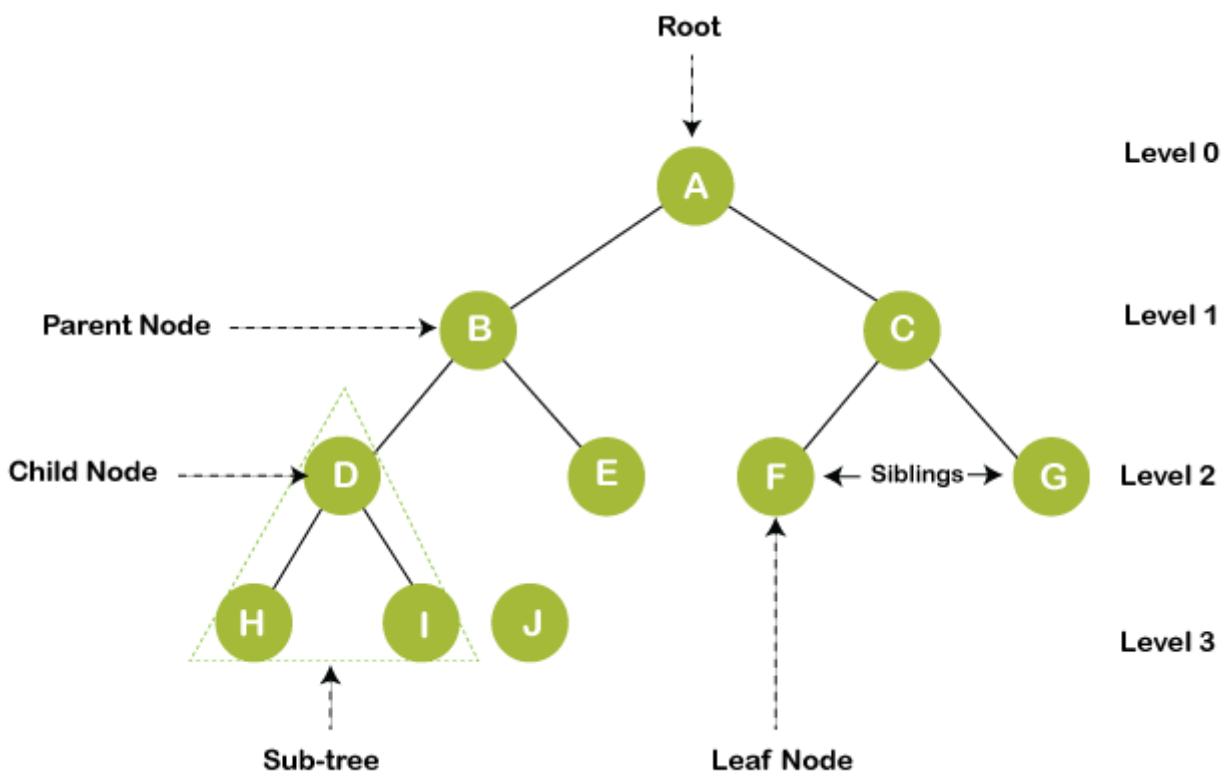
INTRODUCTION TO NON-LINEAR DATA STRUCTURES

A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and **Graphs** are the types of non-linear data structure.

- **Tree**

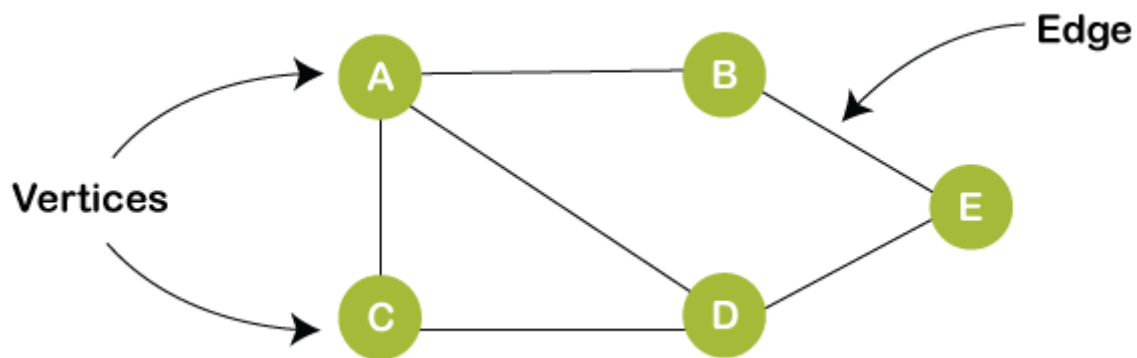
It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a **tree** data structure is shown below:



For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the above figure, **A** represents a manager, **B** and **C** represent the officers, and other nodes represent the clerks.

- **Graph**

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Face book. In the case of face book, a single user can be considered as a node, and the connection of a user with others is known as edges.



TREE – TERMINOLOGY

Tree –definition:-

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications.

A tree data structure can be defined as follows...

“Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.”

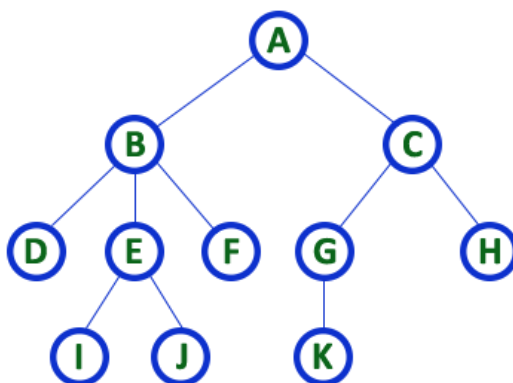
A tree data structure can also be defined as follows...

“Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively”

In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example



TREE with 11 nodes and 10 edges

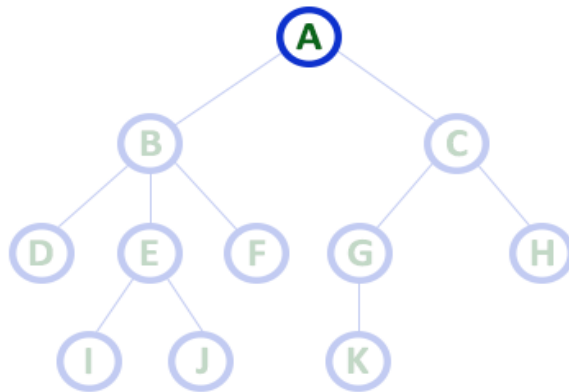
- In any tree with '**N**' nodes there will be maximum of '**N-1**' edges
- In a tree every individual element is called as '**NODE**'

Tree -Terminology

In a tree data structure, we use the following terminology...

1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

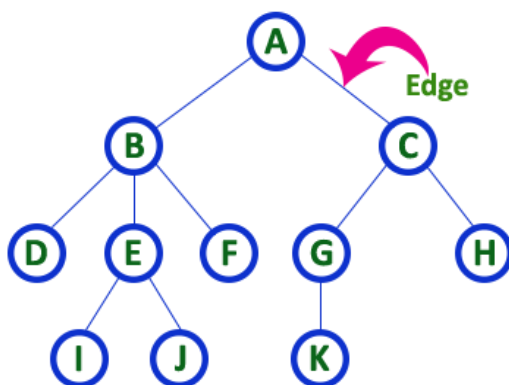


Here 'A' is the 'root' node

- In any tree the first node is called as **ROOT node**

2. Edge

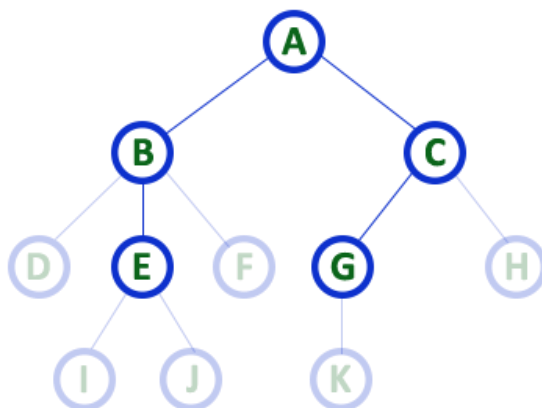
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".



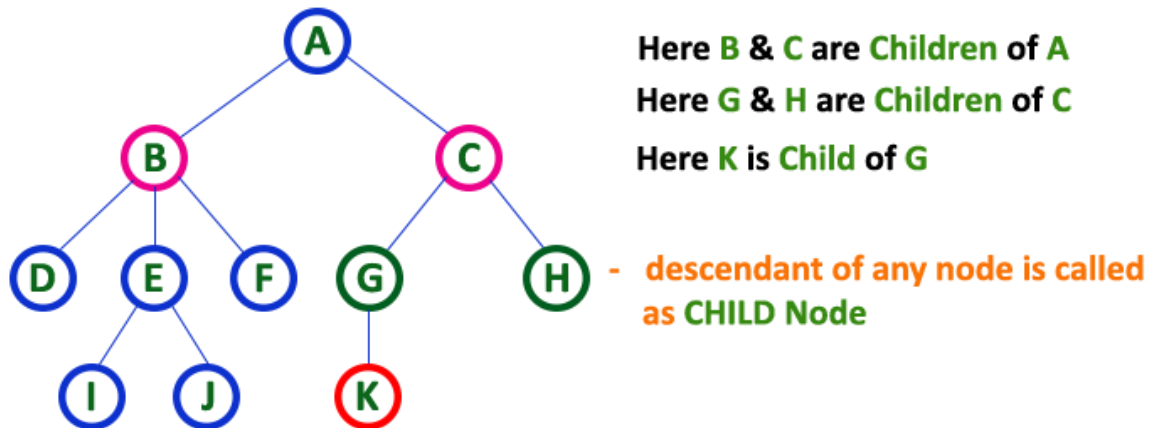
Here A, B, C, E & G are **Parent nodes**

- In any tree the node which has child / children is called '**Parent**'

- A node which is predecessor of any other node is called '**Parent**'

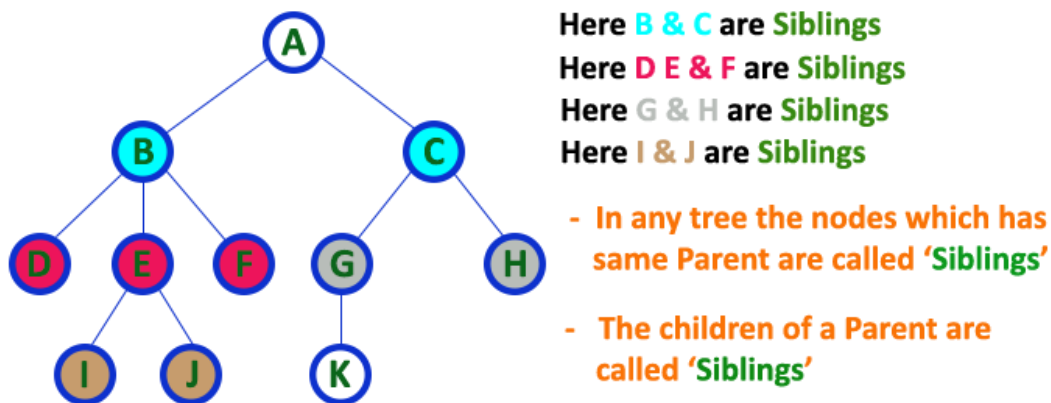
4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



5. Siblings

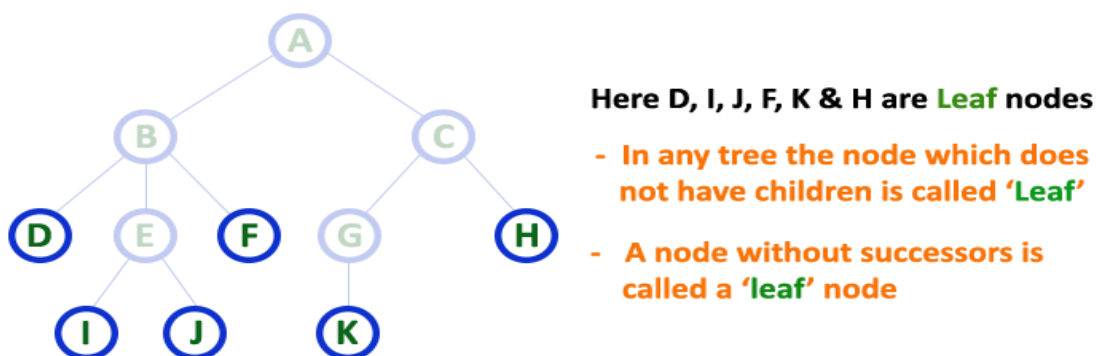
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

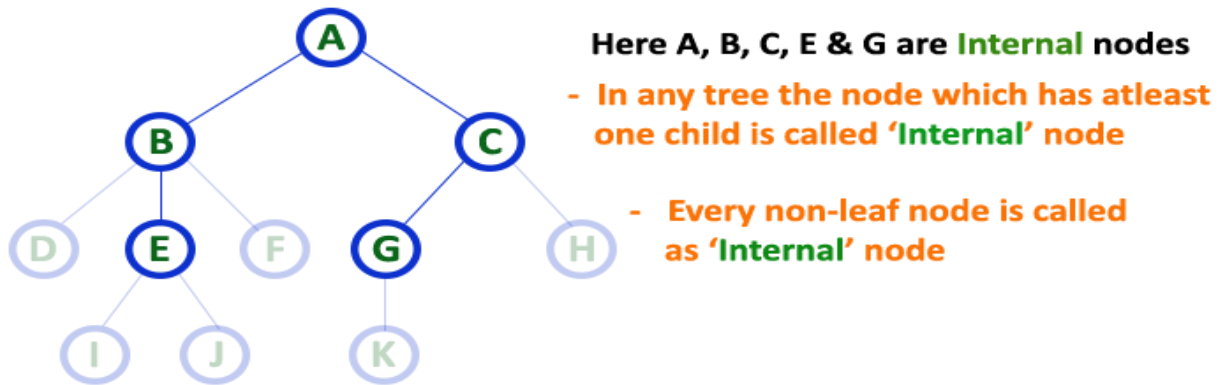
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



7. Internal Nodes

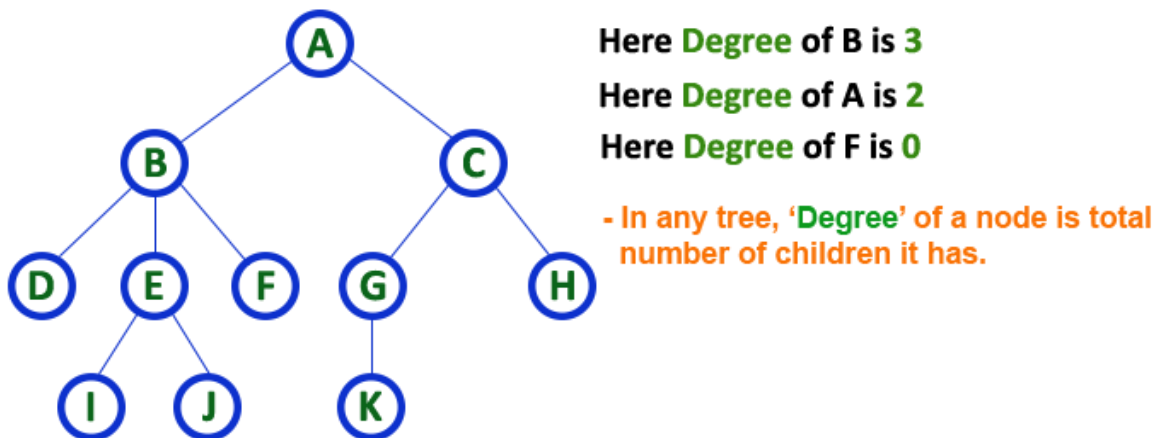
In a tree data structure, the node which has at least one child is called as **INTERNAL Node**. In simple words, an internal node is a node with at least one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. The **root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



8. Degree

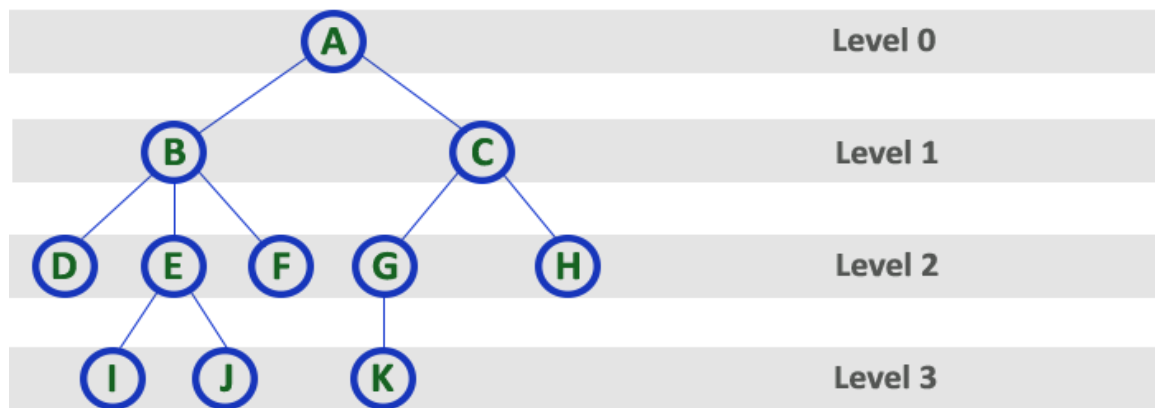
In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



9. Level

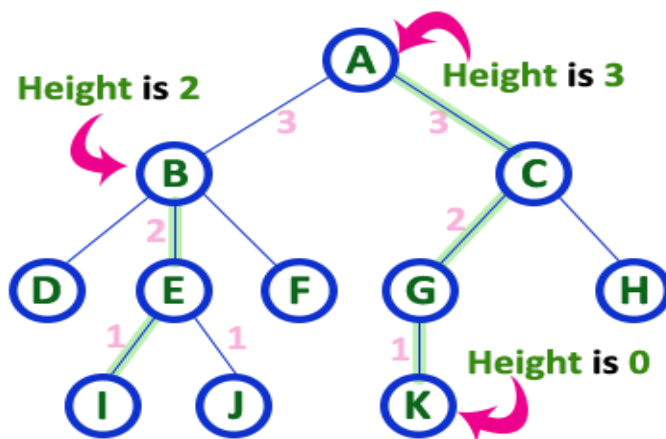
In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...

In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'**.

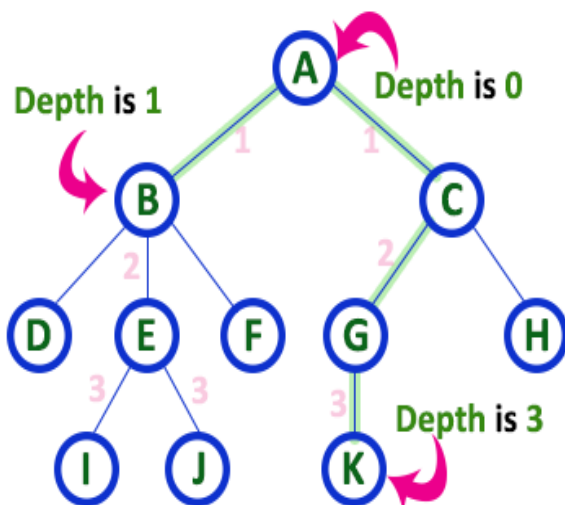


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'**.

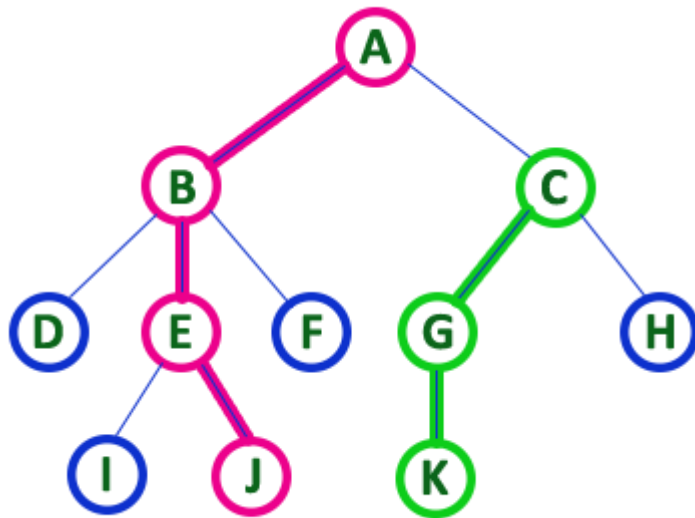


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example the path A - B - E - J has length 4.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

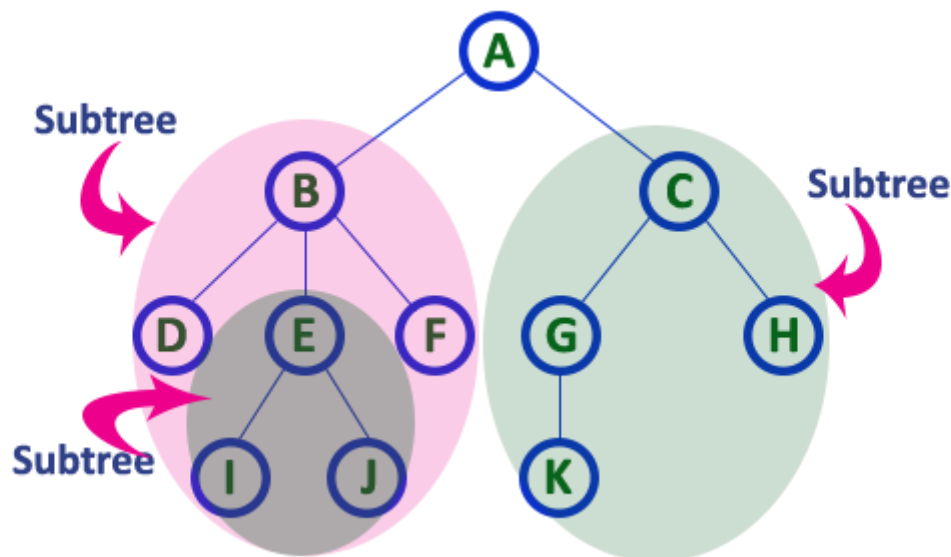
A - B - E - J

Here, 'Path' between C & K is

C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Types of Trees:-

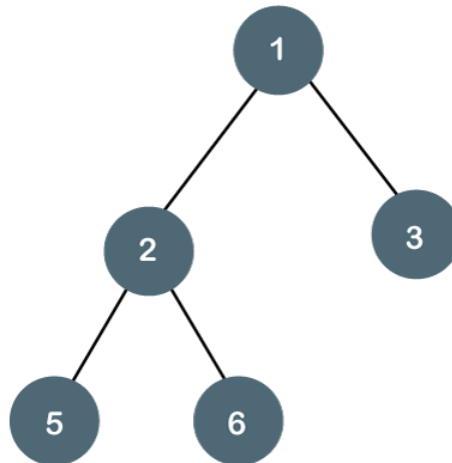
Trees are of following 6 types:

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

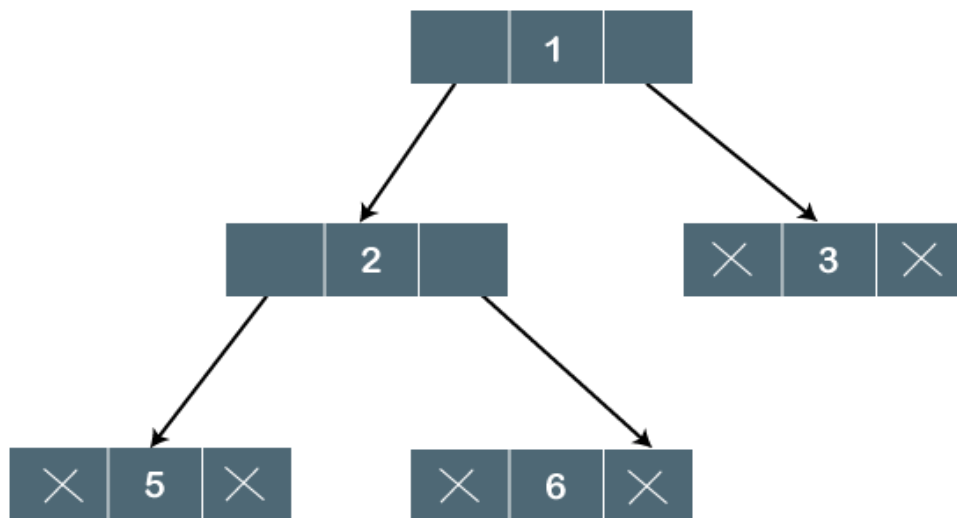
BINARY TREE DATA STRUCTURE

Definition of Binary Tree: - The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Example: -Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i , the maximum number of nodes is 2^i .
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes

at height 3 is equal to $(1+2+4+8) = 15$. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \dots + 2^h) = 2^{h+1} - 1$.

- The minimum number of nodes possible at height h is equal to **$h+1$** .
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$$n = 2^{h+1} - 1$$

$$n+1 = 2^{h+1}$$

Taking log on both the sides,

$$\log_2(n+1) = \log_2(2^{h+1})$$

$$\log_2(n+1) = h+1$$

$$h = \log_2(n+1) - 1$$

The maximum height can be computed as:

As we know that,

$$n = h+1$$

$$h = n-1$$

TYPES OF BINARY TREE

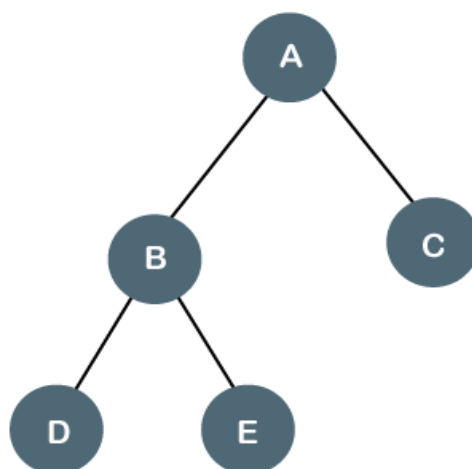
There are four types of Binary tree:

1. Full/ proper/ strict Binary tree
2. Complete Binary tree
3. Perfect Binary tree
4. Degenerate Binary tree
5. Balanced Binary tree

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., $2^{h+1} - 1$.
- The minimum number of nodes in the full binary tree is $2*h+1$.
- The minimum height of the full binary tree is $\log_2(n+1) - 1$.
- The maximum height of the full binary tree can be computed as:

$$n = 2*h + 1$$

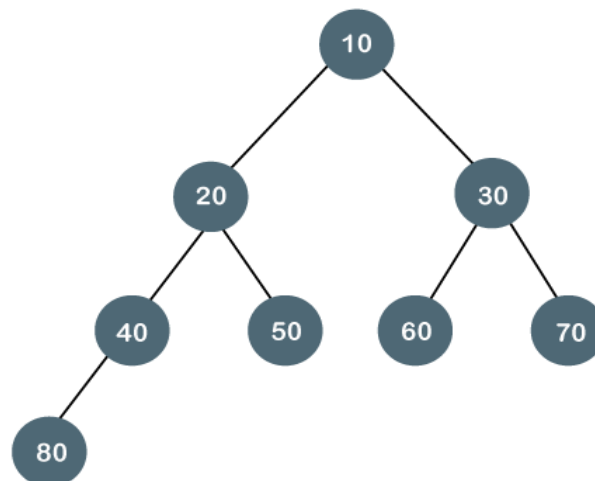
$$n-1 = 2*h$$

$$h = (n-1)/2$$

2. Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



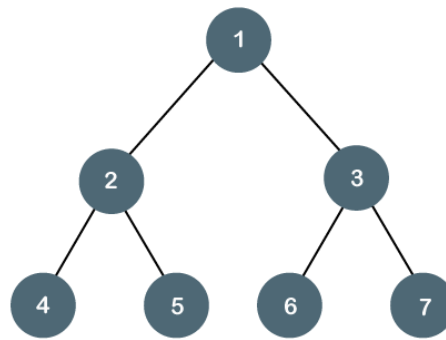
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

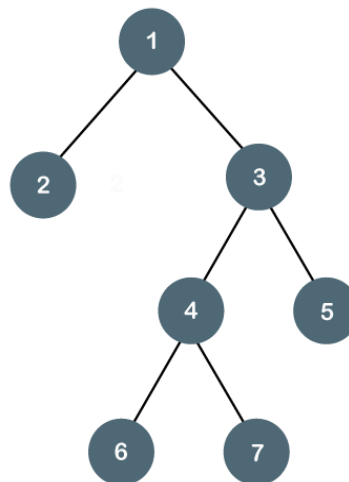
- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h .
- The minimum height of a complete binary tree is $\log_2(n+1) - 1$.
- The maximum height of a complete binary tree is $\log n$.

3. Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



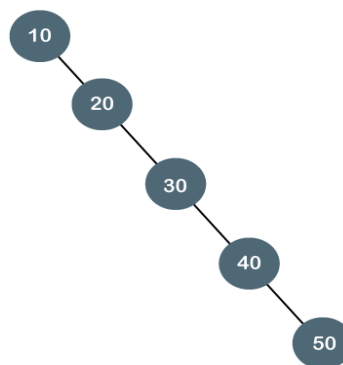
The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.



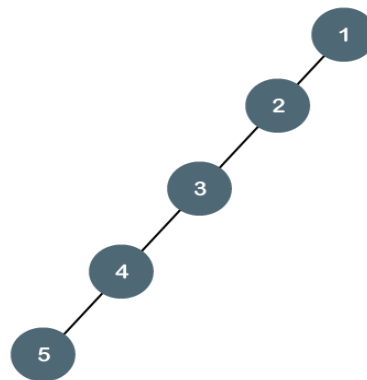
4. Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one child.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.



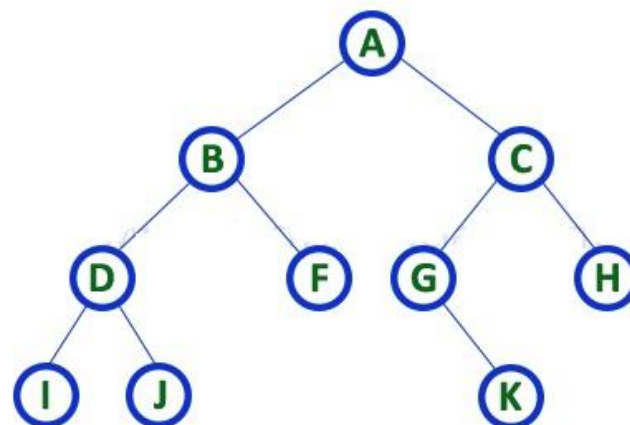
The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

BINARY TREE REPRESENTATIONS

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...



To represent a binary tree of depth ' n ' using array representation, we need one dimensional array with a maximum size of $2^{n+1}-1$

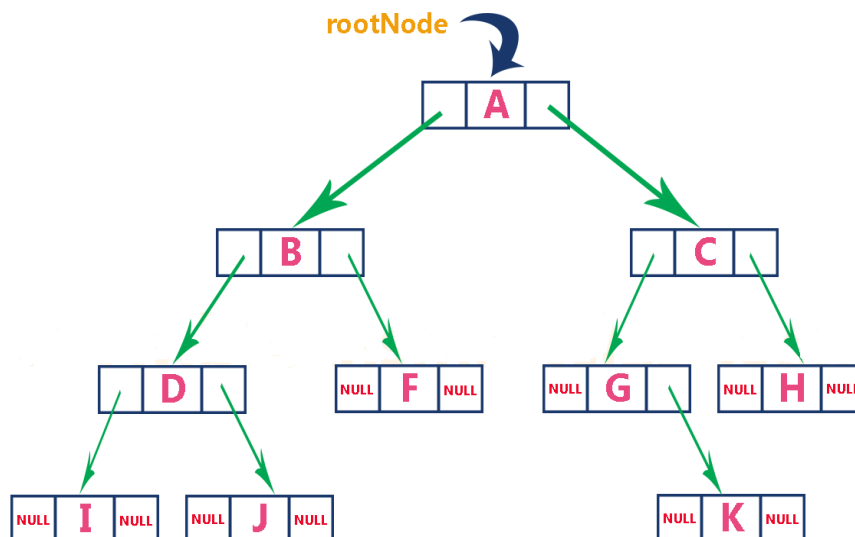
2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of the binary tree represented using Linked list representation is shown as follows...



BINARY TREE IMPLEMENTATION

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```

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

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Creation of Binary Tree program in C:-

```
#include<stdio.h>

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

struct node *create()
{
    struct node *newnode;
    int data;
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("Press 0 to exit");
    printf("\nPress 1 for new node");
    printf("Enter your choice : ");
    scanf("%d", &choice);
    if(choice==0)
    {
        return 0;
    }
    else
    {
        printf("Enter the data:");
        scanf("%d", &data);
        newnode->data = data;
        printf("Enter the left child of %d", data);
        newnode->left = create();
        printf("Enter the right child of %d", data);
        newnode->right = create();
        return temp;
    }
}

void main()
{
    struct node *root;
    root = create();
}
```

```
}
```

The above code is calling the create() function recursively and creating new node on each recursive call. When all the nodes are created, then it forms a binary tree structure.

BINARY TREE TRAVERSALS

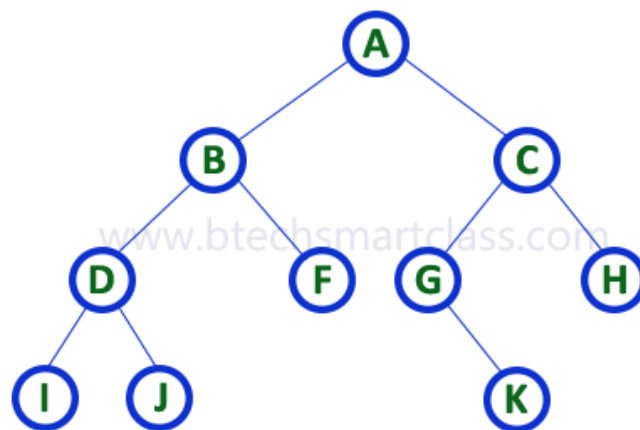
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all sub trees in the tree. This is performed recursively for all nodes in the tree. In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left sub tree. So we try to visit its (B's) left child 'D' and again D is a root for sub tree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a sub tree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit

G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all sub trees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the leftmost child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this, we have completed node C's root and left parts. Next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

BINARY SEARCH TREE

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

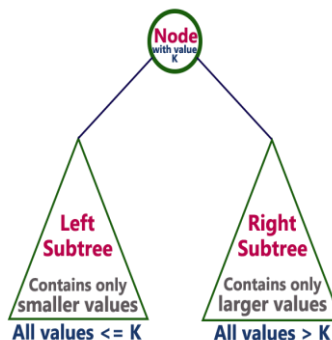
A binary tree has the following time complexities...

1. Search Operation - $O(n)$
2. Insertion Operation - $O(1)$
3. Deletion Operation - $O(n)$

To enhance the performance of binary tree, we use a special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

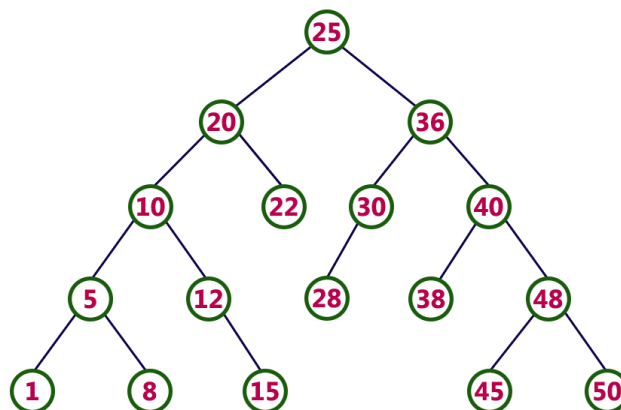
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left sub tree of any node contains smaller values and all the nodes in the right sub tree of any node contains larger values as shown in the following figure...



Example

The following tree is a Binary Search Tree. In this tree, left sub tree of every node contains nodes with smaller values and right sub tree of every node contains larger values.



Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

OPERATIONS ON A BINARY SEARCH TREE (BST)

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.

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

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

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

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

Step 6 - If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

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

Step 1 - Create a newNode with given value and set its **left** and **right** to **NULL**.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is **Empty**, then set **root** to **newNode**.

Step 4 - If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

Step 5 - If newNode is **smaller** than or **equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.

Step 6 - Repeat the above steps until we reach to the **leaf** node (i.e., reaches to **NULL**).

Step 7 - After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller** or **equal** to that leaf node or else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1 - Find the node to be deleted using **search operation**

Step 2 - Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1 - Find the node to be deleted using **search operation**

Step 2 - If it has only one child then create a link between its parent node and child node.

Step 3 - Delete the node using **free** function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1 - Find the node to be deleted using **search operation**

Step 2 - If it has two children, then find the **largest** node in its **left sub tree** (OR) the **smallest** node in its **right sub tree**.

Step 3 - Swap both **deleting node** and node which is found in the above step.

Step 4 - Then check whether deleting node came to **case 1** or **case 2** or else goto step 2

Step 5 - If it comes to **case 1**, then delete using case 1 logic.

Step 6- If it comes to **case 2**, and then delete using case 2 logic.

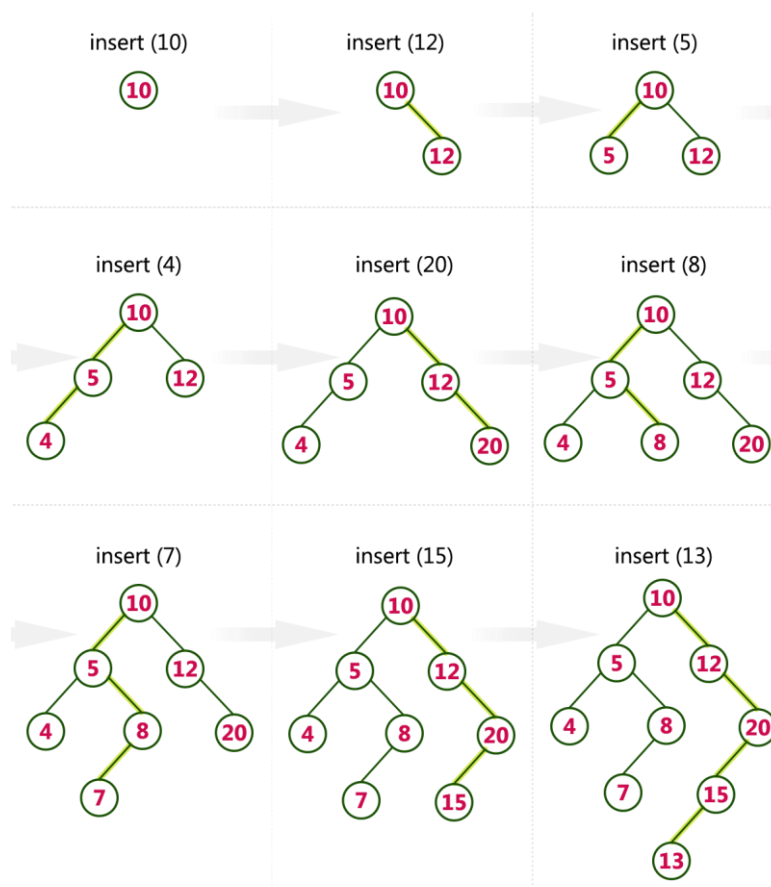
Step 7 - Repeat the same process until the node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10, 12, 5, 4, 20, 8, 7, 15 and 13

Above elements are inserted into a Binary Search Tree as follows...



RECONSTRUCTING OF A BINARY TREE

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:

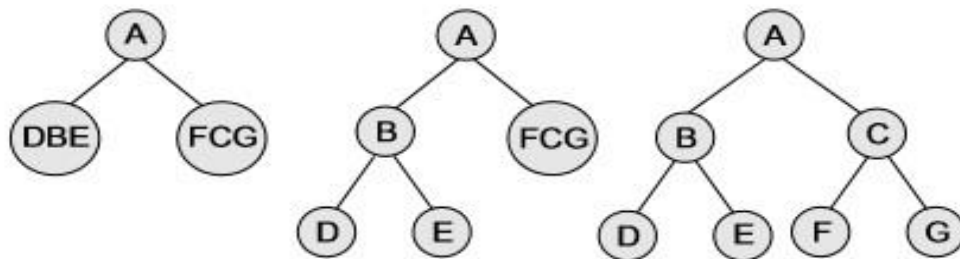
Step 1 Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.

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.

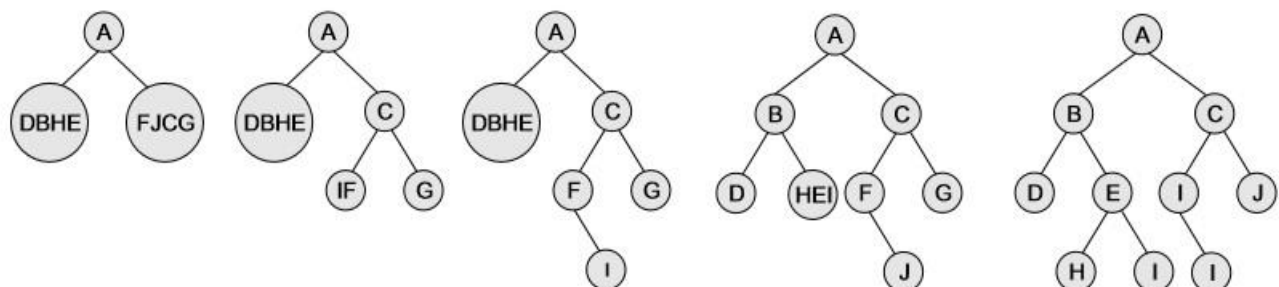
Look at the following Figure which constructs the tree from its traversal results.

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



Now consider the in-order traversal and post-order traversal sequences of a given binary tree. Before constructing the binary tree, remember that in post-order traversal the root node is the last node. Rest of the steps will be the same as mentioned in below figure.

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



Steps to show binary tree

APPLICATIONS OF BINARY TREES

The following are the applications of binary trees:

1. **Binary Search Tree** - Used in many search applications that constantly show and hide data, such as data. For example, map and set objects in many libraries.
2. **Binary Space Partition** - Used in almost any 3D video game to determine which objects need to be rendered.
3. **Binary Tries** - Used in almost every high-bandwidth router to store router tables.
4. **Syntax Tree** - Constructed by compilers and (implicit) calculators to parse expressions.
5. **Hash Trees** - Used in P2P programs and special image signatures that require a hash to be validated, but the entire file is not available.
6. **Heaps** - Used to implement efficient priority queues and also used in heap sort.
7. **Treap** - Randomized data structure for wireless networks and memory allocation.
8. **T-Tree** - Although most databases use a form of B-tree to store data on the drive, databases that store all (most) data often use T-trees.
9. **Huffman Coding Tree (Chip Uni)** - Used in compression algorithms, eg. For example, in .jpeg and .mp3. GGM Trees file formats - used in cryptographic applications to generate a tree with pseudo-random numbers.

UNIT-3 CHAPTER: 2

AVL TREE (OR) HEIGHT BALANCED BST

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

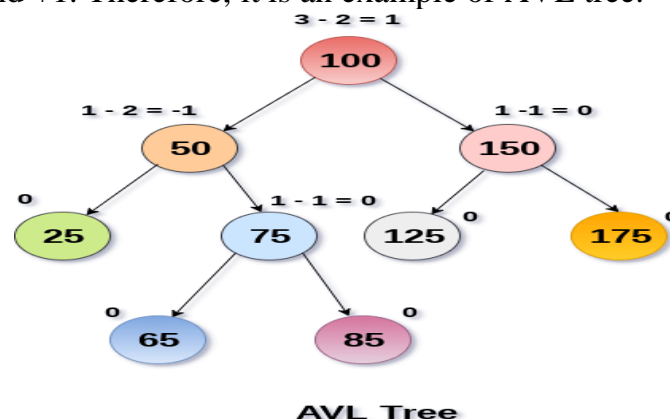
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. Therefore, it is an example of AVL tree.



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

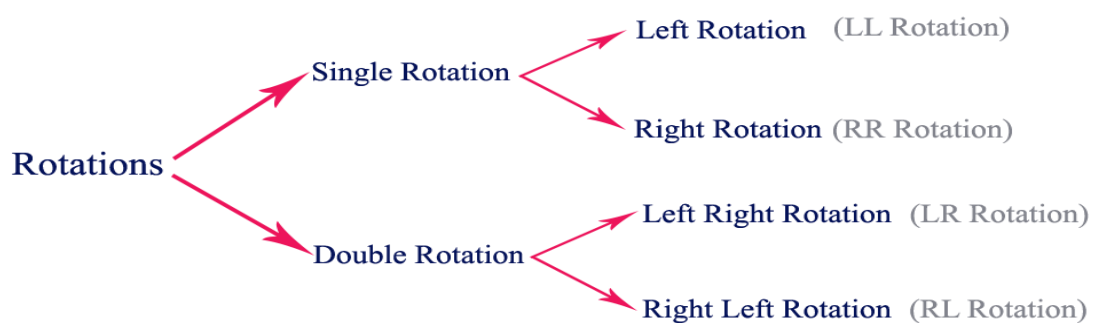
Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

AVL TREE ROTATIONS

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced. Rotation operations are used to make the tree balanced.

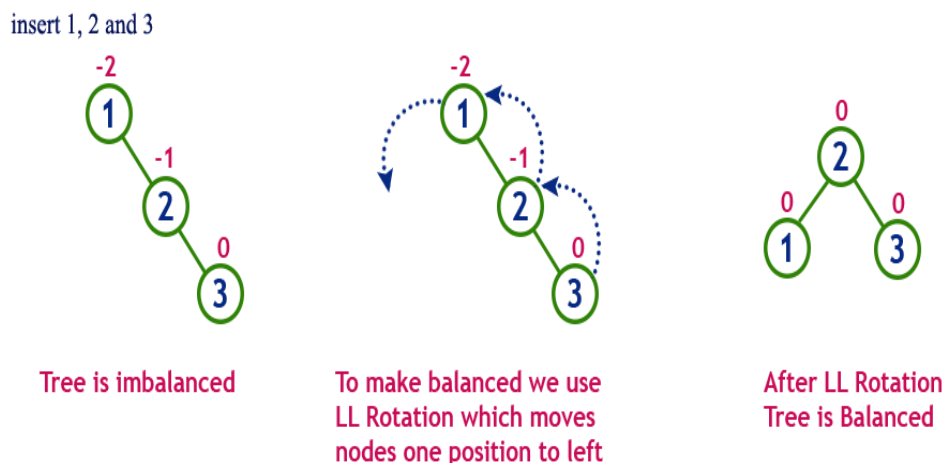
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

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



Single Left Rotation (LL Rotation)

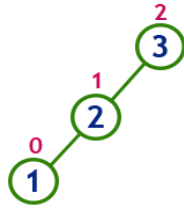
In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



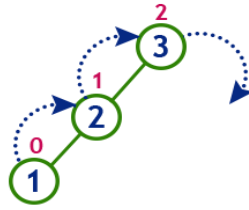
Single Right Rotation (RR Rotation)

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

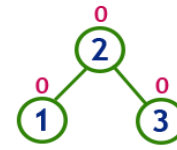
insert 3, 2 and 1



Tree is imbalanced
because node 3 has balance factor 2



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

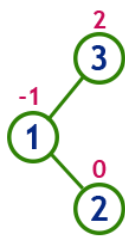


After RR Rotation
Tree is Balanced

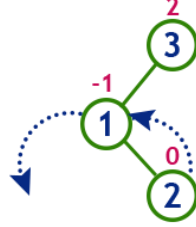
Left Right Rotation (LR Rotation)

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

insert 3, 1 and 2

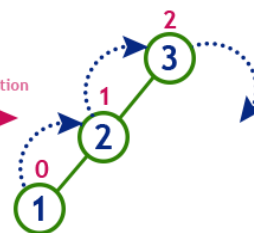


Tree is imbalanced
because node 3 has balance factor 2



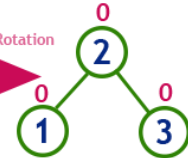
LL Rotation

After LL Rotation



RR Rotation

After RR Rotation

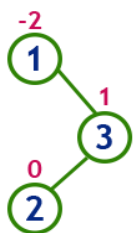


After LR Rotation
Tree is Balanced

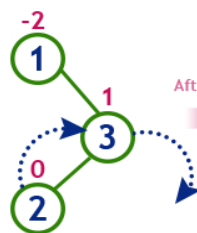
Right Left Rotation (RL Rotation)

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

insert 1, 3 and 2

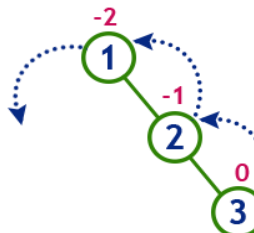


Tree is imbalanced
because node 1 has balance factor -2



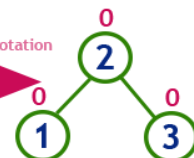
RR Rotation

After RR Rotation



LL Rotation

After LL Rotation



After RL Rotation
Tree is Balanced

OPERATIONS ON AN AVL TREE

The following operations are performed on AVL tree...

1. Search

2. Insertion

3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with $O(\log n)$ time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

Step 1 - Read the search element from the user.

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

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

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

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

Step 6 - If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.

Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in AVL Tree

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

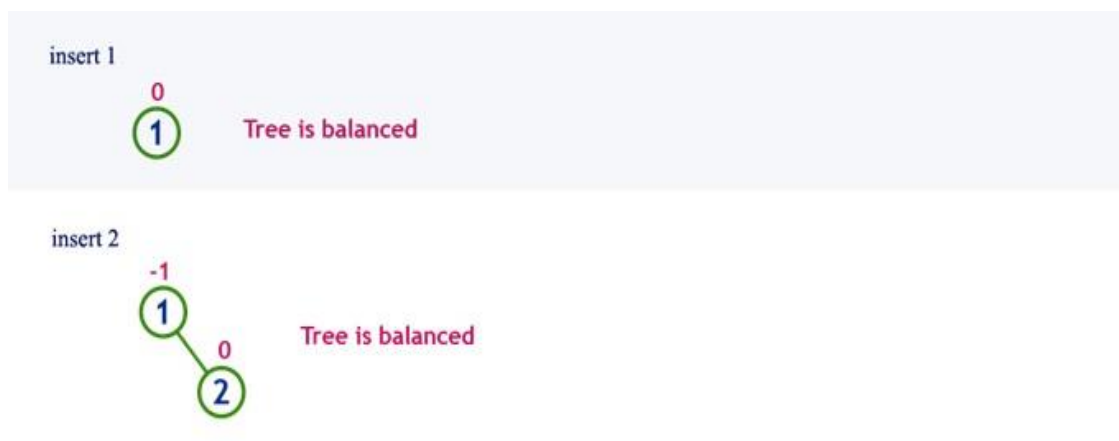
Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

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

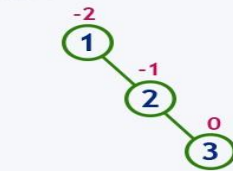
Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

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

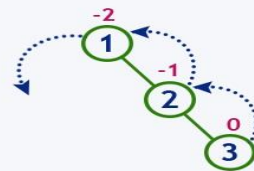
Example: Construct an AVL Tree by inserting numbers from 1 to 8.



insert 3

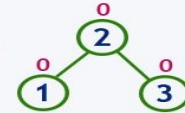


Tree is imbalanced



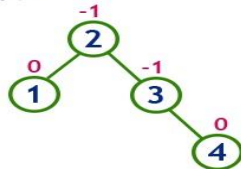
LL Rotation

After LL Rotation



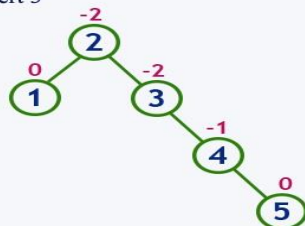
Tree is balanced

insert 4

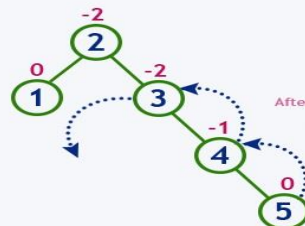


Tree is balanced

insert 5

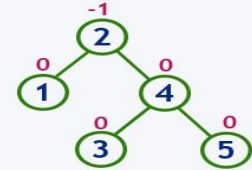


Tree is imbalanced



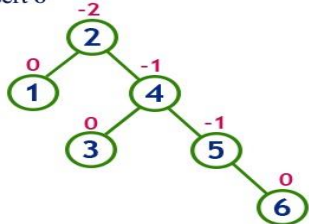
LL Rotation at 3

After LL Rotation at 3

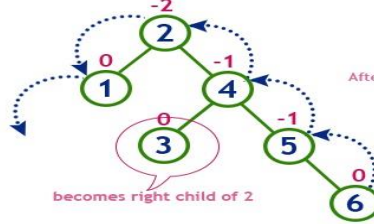


Tree is balanced

insert 6



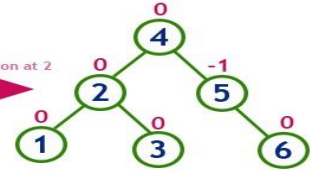
Tree is imbalanced



LL Rotation at 2

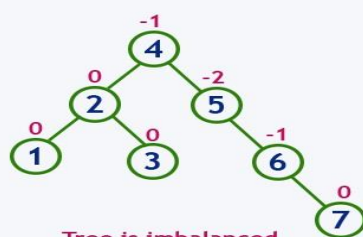
becomes right child of 2

After LL Rotation at 2

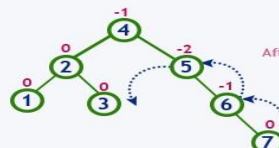


Tree is balanced

insert 7

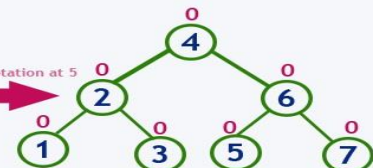


Tree is imbalanced



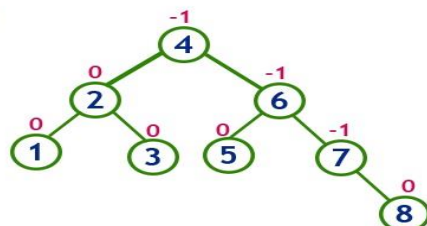
LL Rotation at 5

After LL Rotation at 5



Tree is balanced

insert 8



Tree is balanced

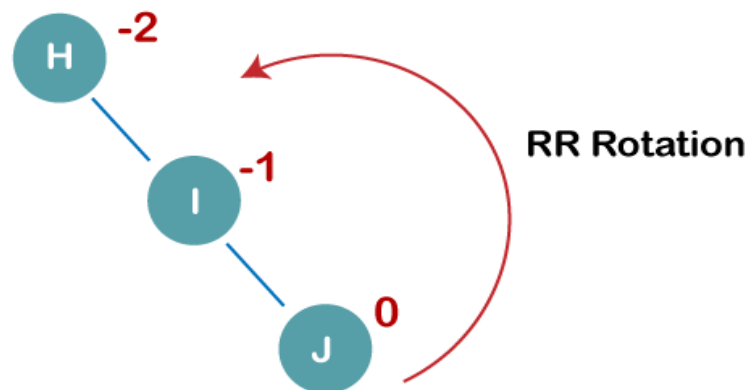
Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Construct an AVL tree having the following elements

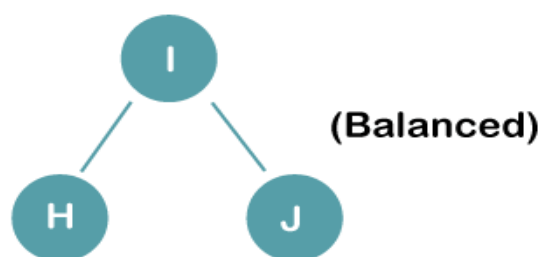
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

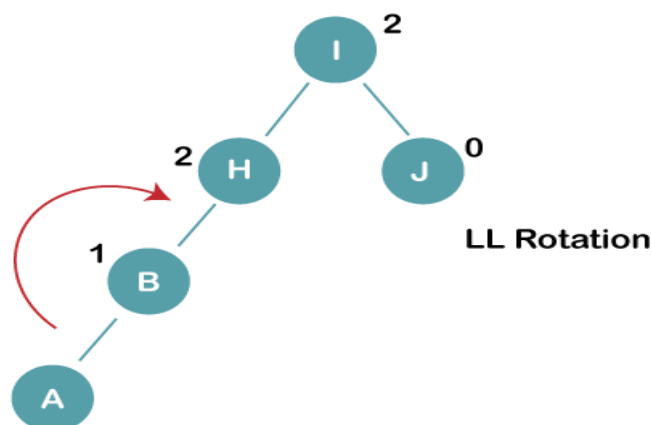


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

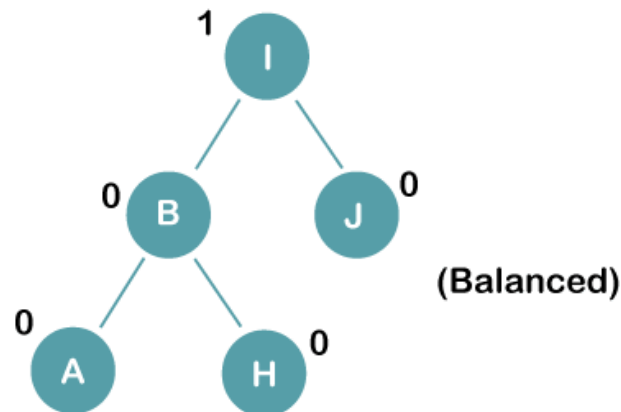


2. Insert B, A

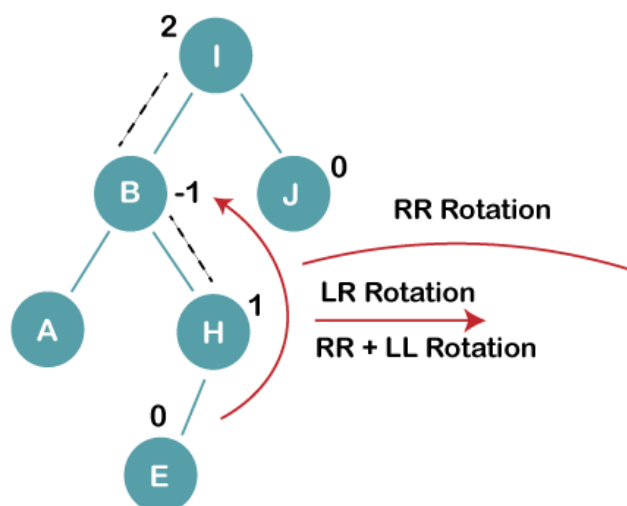


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



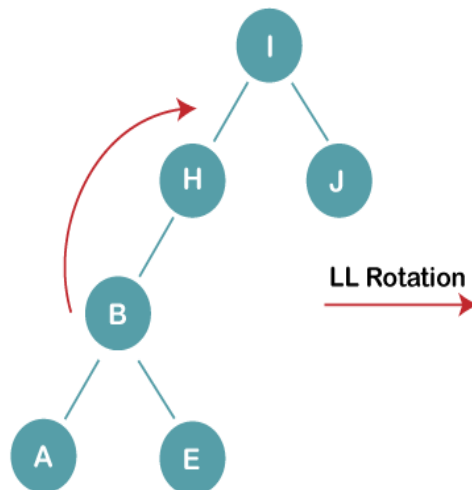
3. Insert E



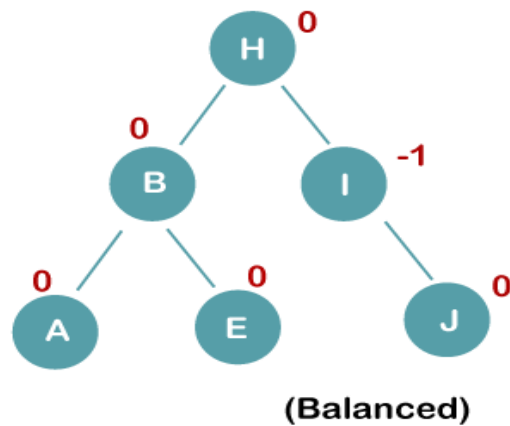
On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = LL + RR rotation

3 a) We first perform RR rotation on node B

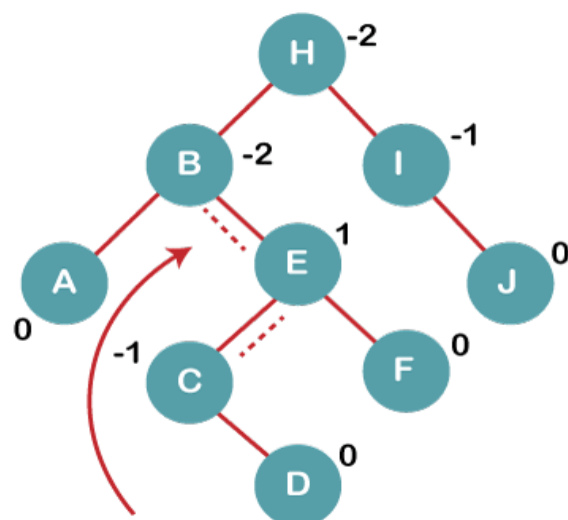
The resultant tree after RR rotation is:



3b) We first perform LL rotation on the node I
The resultant balanced tree after LL rotation is:



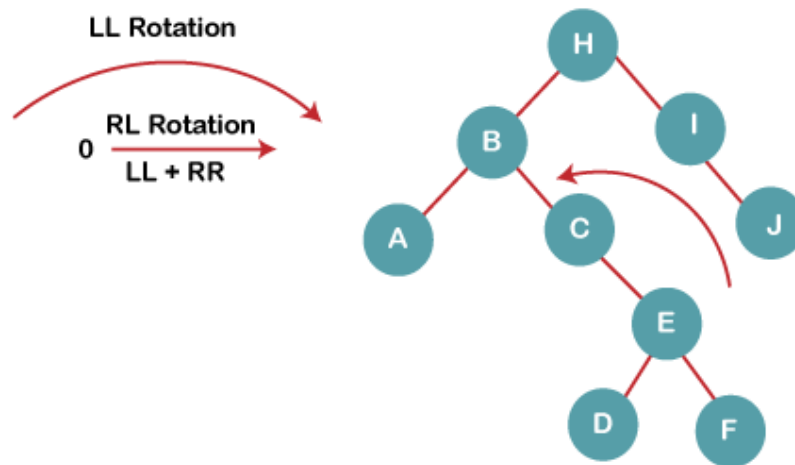
A. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

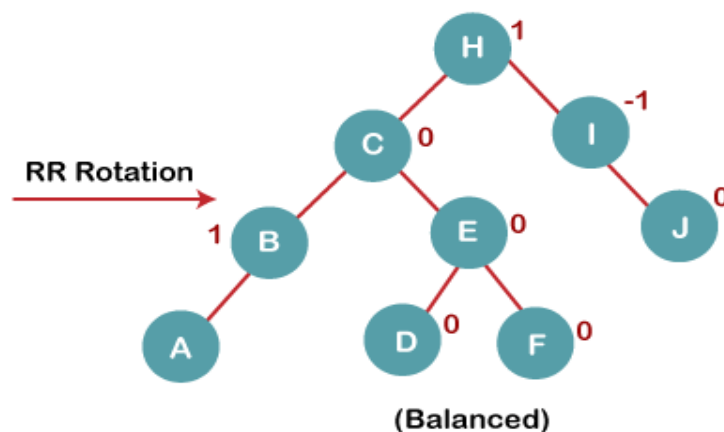
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

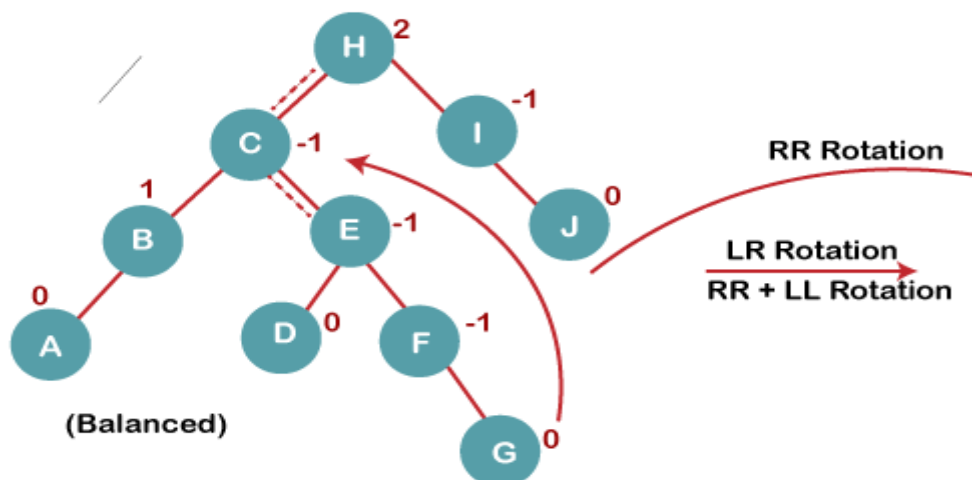


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



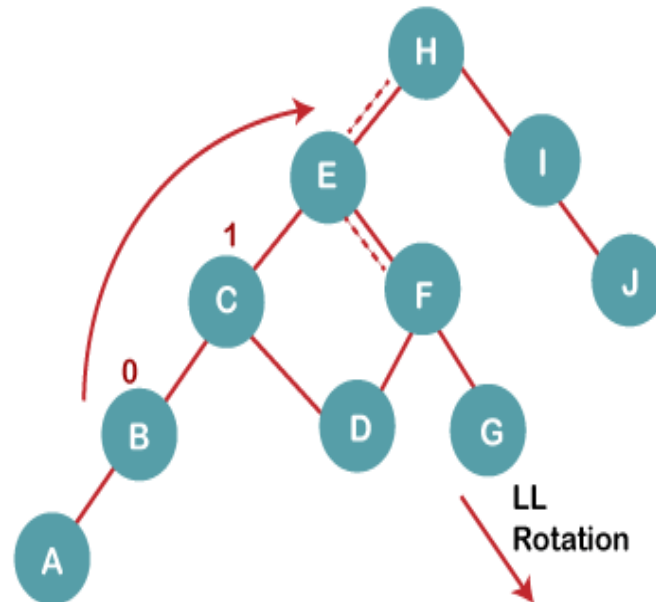
A. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

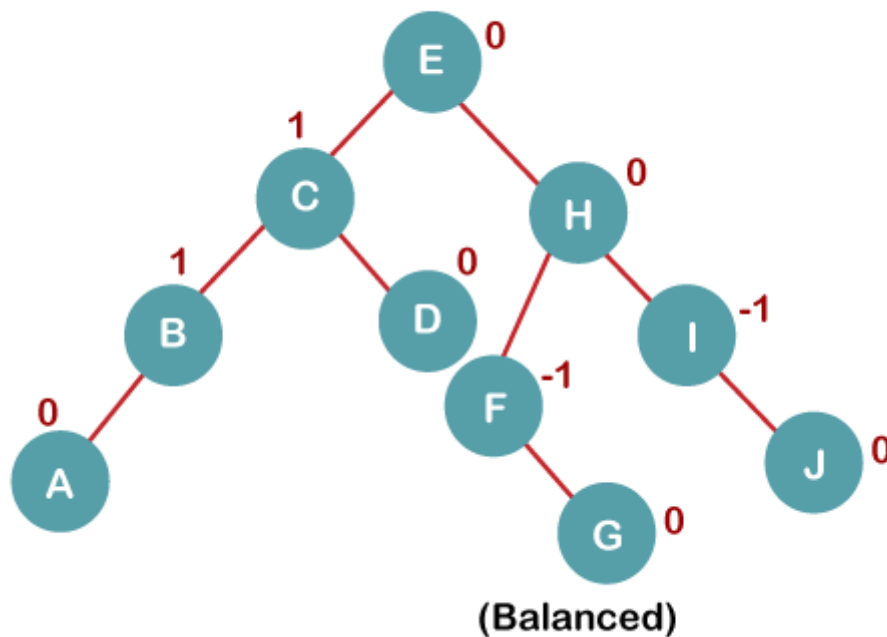
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

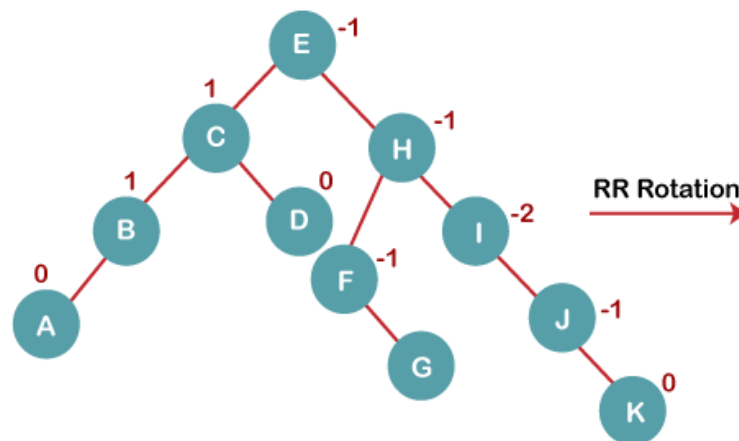


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

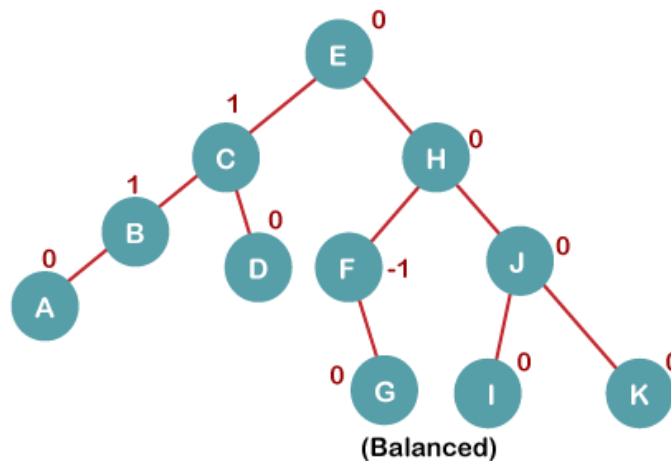


6. Insert K



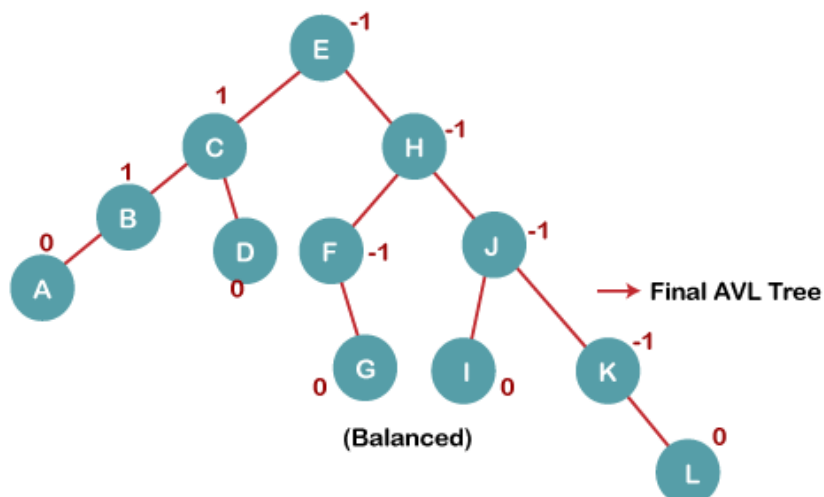
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



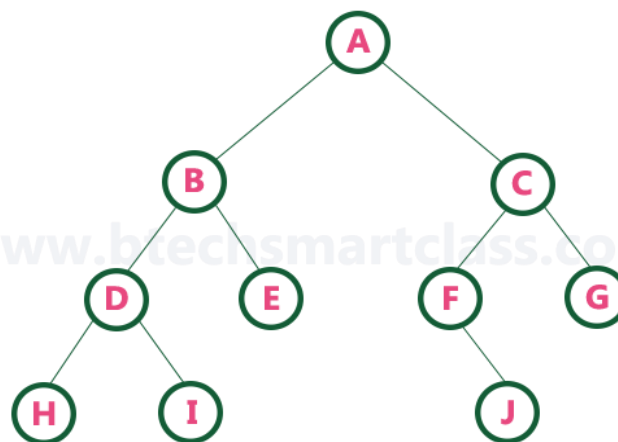
THREADED BINARY TREES

A binary tree can be represented using array representation or linked list representation. When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer. In any binary tree linked list representation, there is a number of NULL pointers than actual pointers. Generally, in any binary tree linked list representation, if there are $2N$ number of reference fields, then $N+1$ number of reference fields are filled with NULL ($N+1$ are NULL out of $2N$). This NULL pointer does not play any role except indicating that there is no link (no child).

A.J. Perlis and C. Thornton have proposed new binary tree called "**Threaded Binary Tree**", which makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as **threads**.

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

If there is no in-order predecessor or in-order successor, then it points to the root node. Consider the following binary tree...



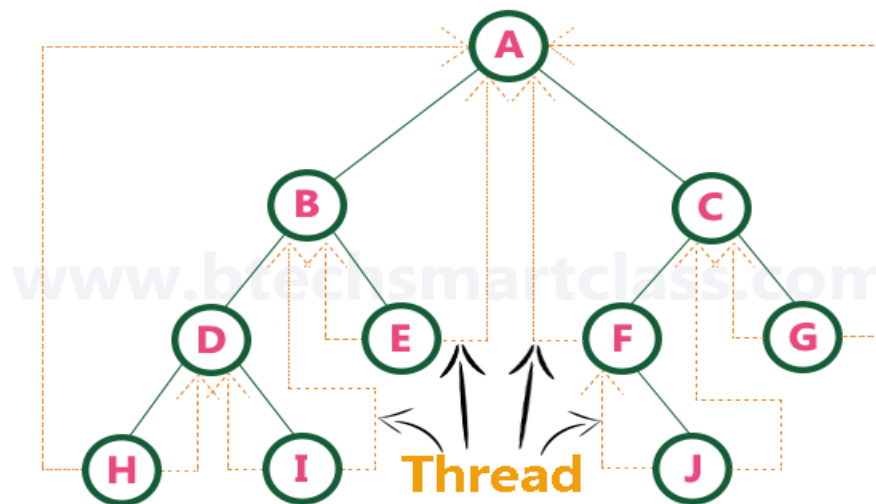
To convert the above example binary tree into a threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

$H - D - I - B - E - A - F - J - C - G$

When we represent the above binary tree using linked list representation, nodes **H, I, E, F, J** and **G** left child pointers are NULL. This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes **H, I, E, J** and **G** right child pointers are NULL. These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

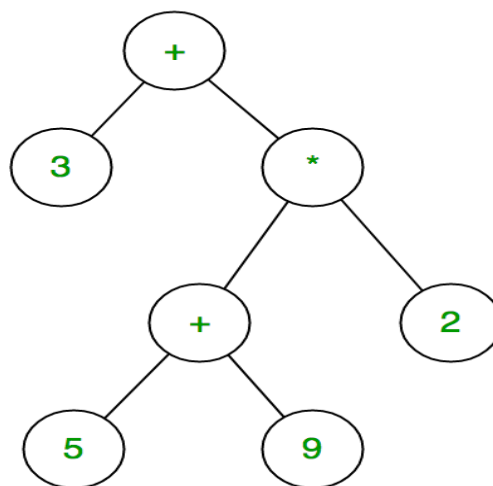
Above example binary tree is converted into threaded binary tree as follows.



In the above figure, threads are indicated with dotted links.

EXPRESSION TREE

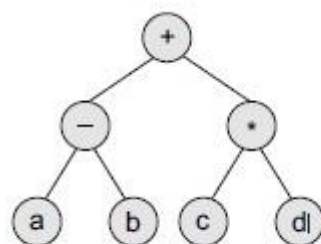
The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for $3 + ((5+9)*2)$ would be:



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

$$\text{Exp} = (a - b) + (c * d)$$

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



Expression tree

HEAP DATA STRUCTURE

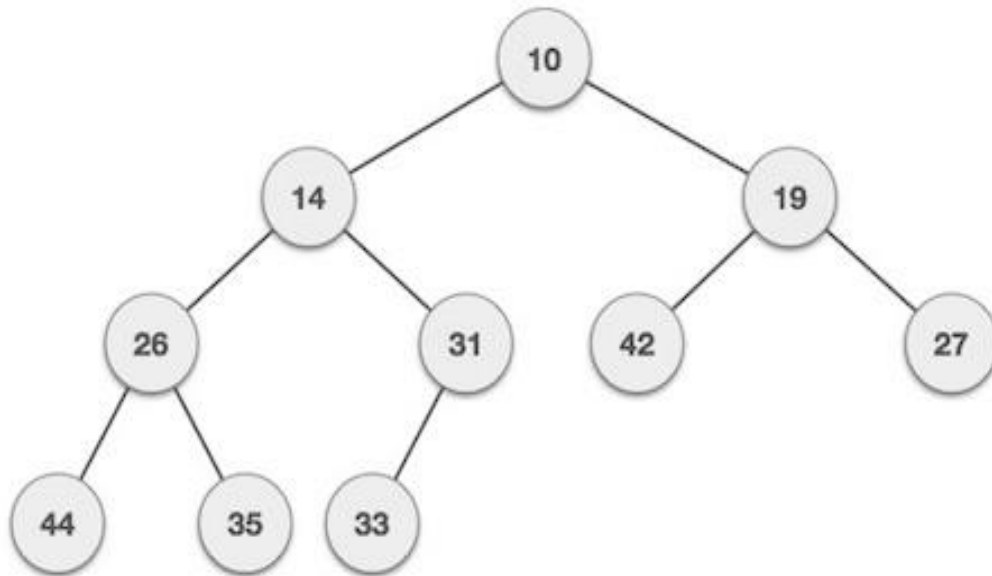
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then –

$$\text{Key } (\alpha) \geq \text{key } (\beta)$$

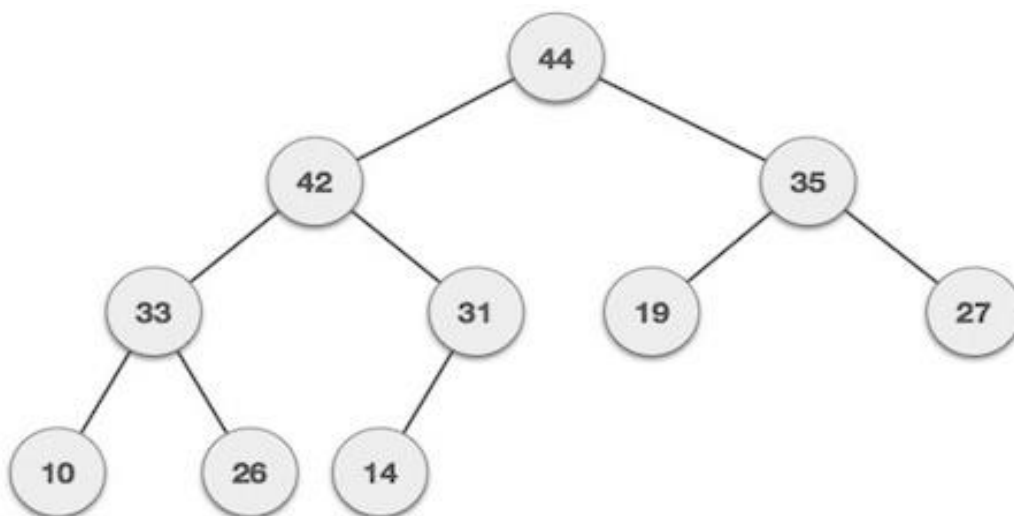
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap – Where the value of the root node is less than or equal to either of its children.



Max-Heap – Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 – Create a new node at the end of heap.

Step 2 – Assign new value to the node.

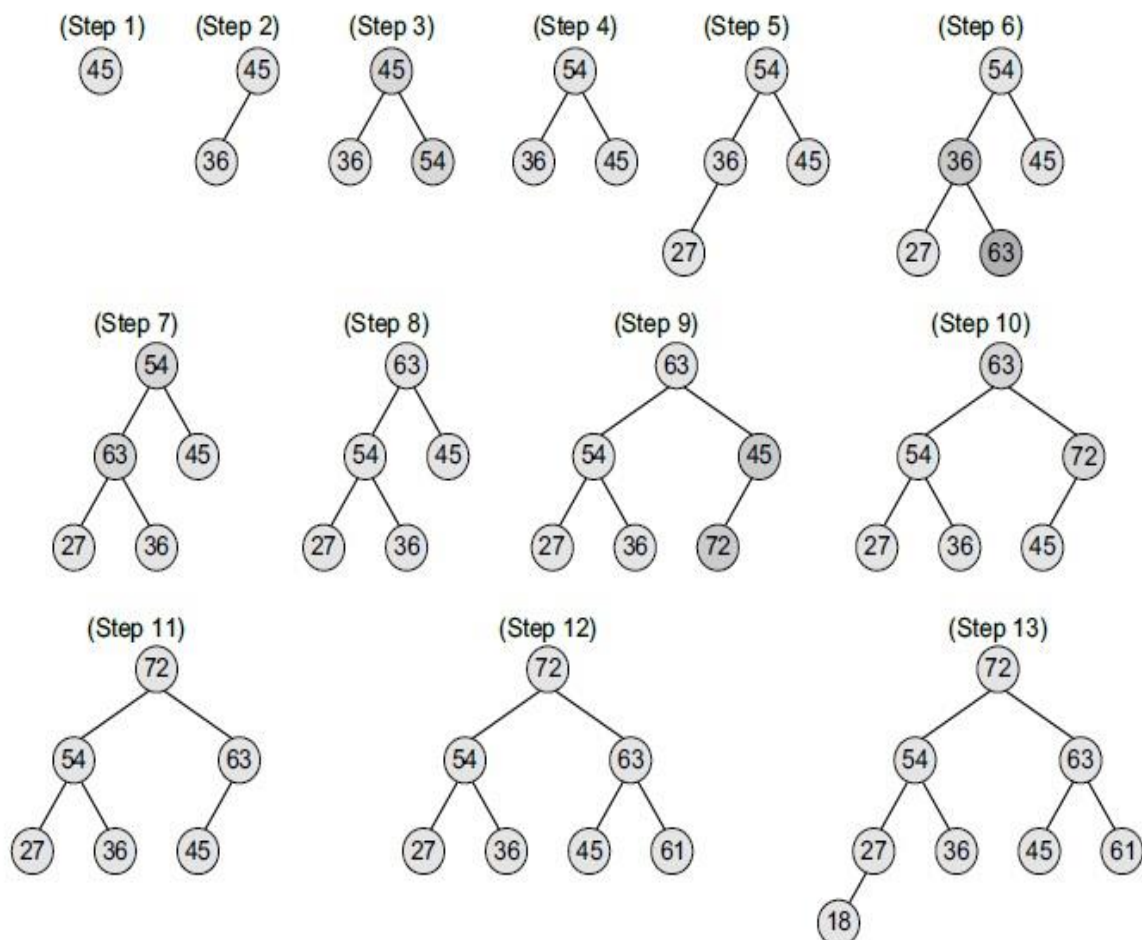
Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

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

Solution



Max Heap Deletion Algorithm

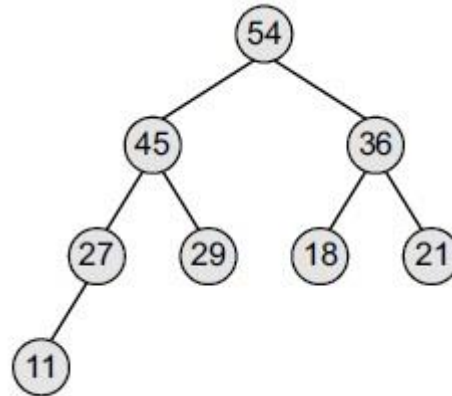
Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.

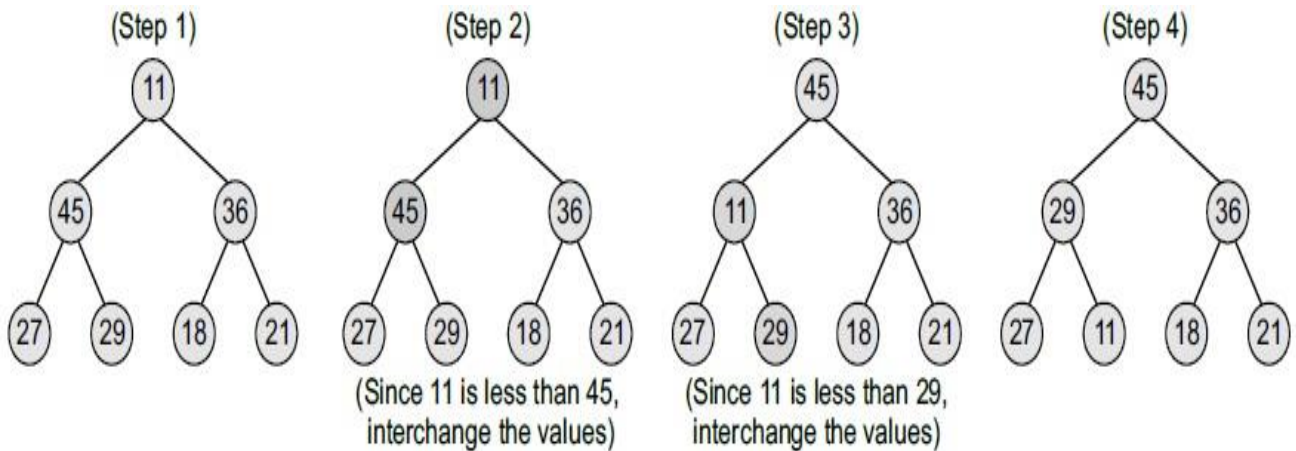
2. Delete the last node.

3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

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



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.



APPLICATIONS OF HEAPS

Heaps are preferred for applications that include:

1. **Heap sort** It is one of the best sorting methods that has no quadratic worst-case scenarios.
2. **Selection algorithms** these algorithms are used to find the minimum and maximum values in linear or sub-linear time.
3. **Graph algorithms** Heaps can be used as internal traversal data structures. This guarantees that runtime is reduced by an order of polynomial. Heaps are therefore used for implementing Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.