Tree Terminology
In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in 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.

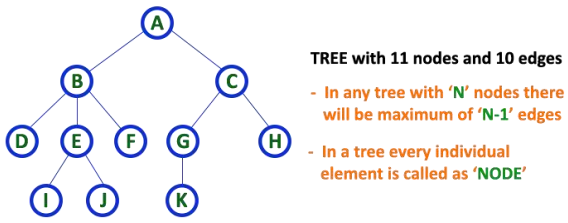
A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

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



Terminology

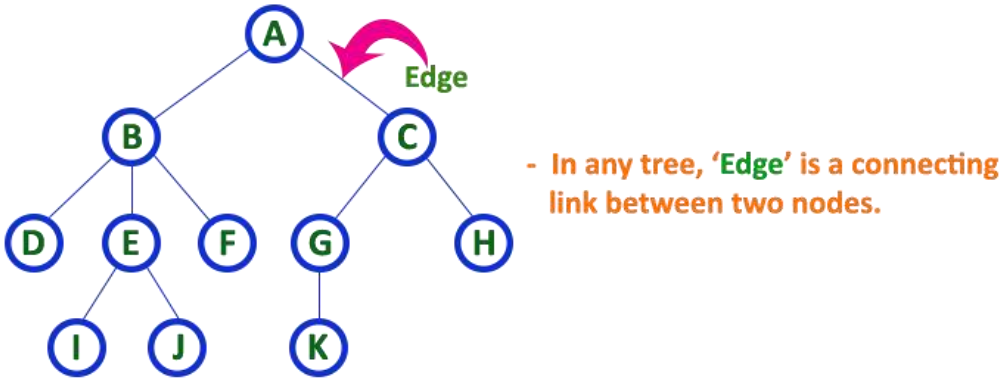
In a tree data structure, we use the following terminology...

1. Root

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

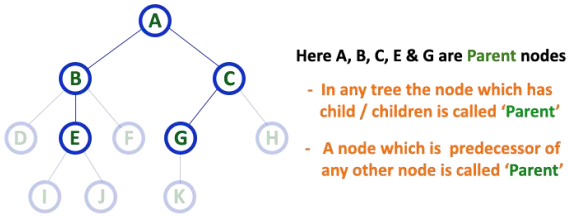
2. Edge

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.



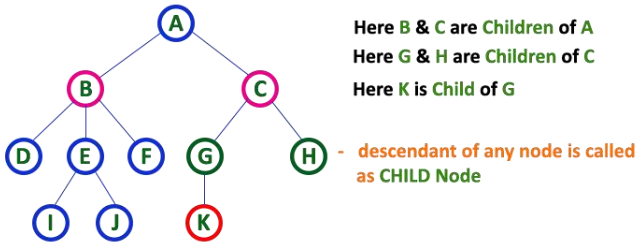
3. Parent

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



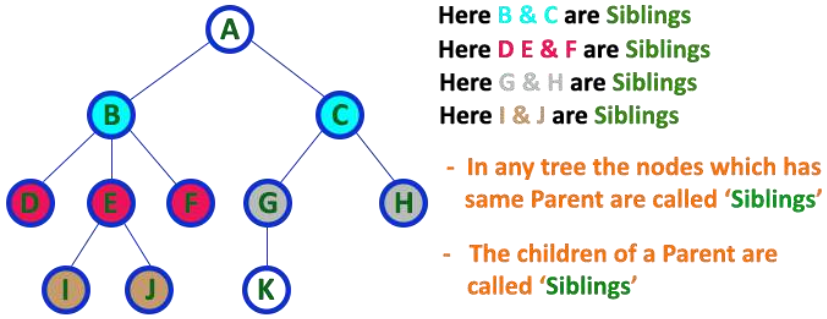
4. Child

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

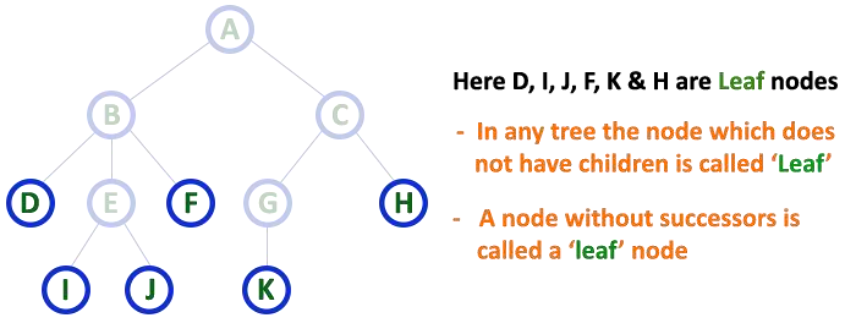
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



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.

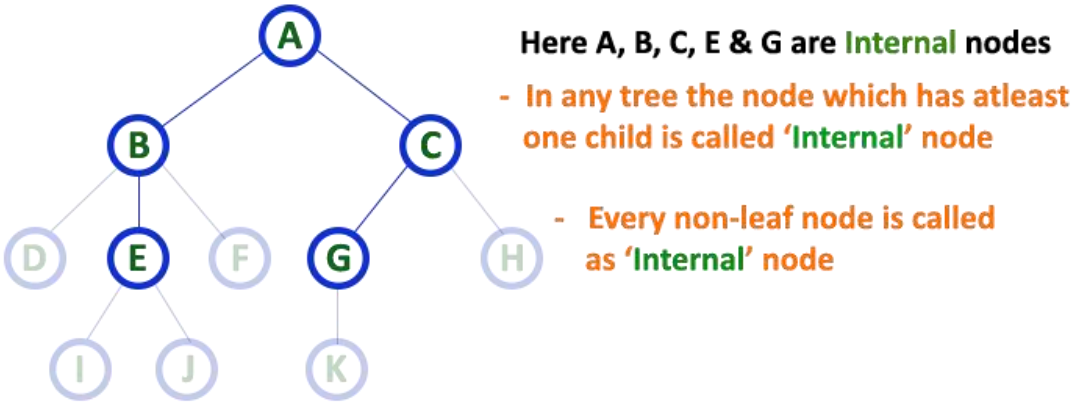
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



7. Internal Nodes

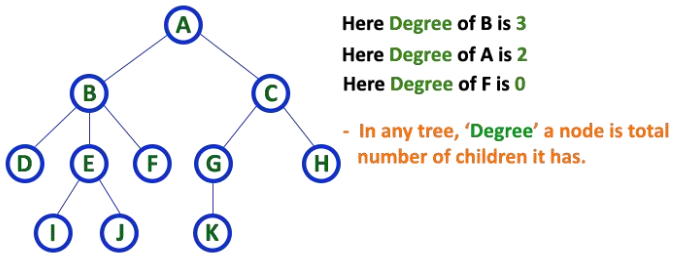
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

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.



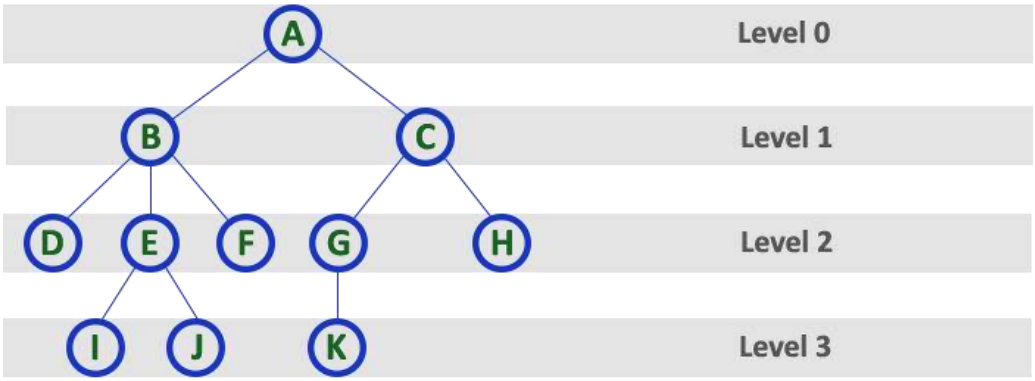
8. Degree

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**'



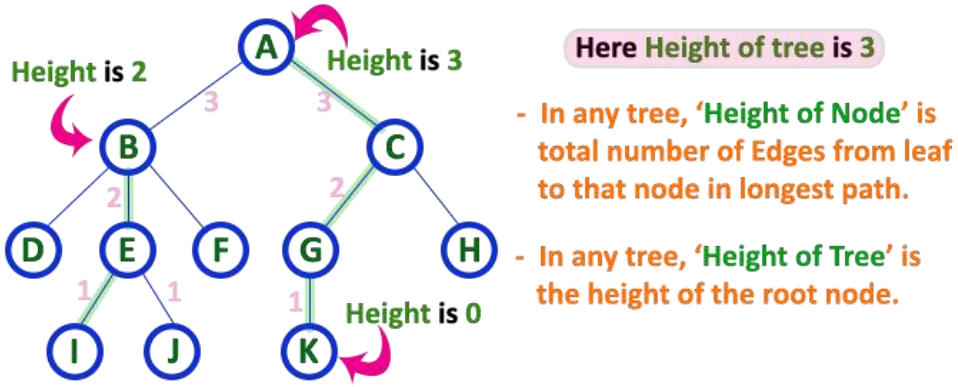
9. Level

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).



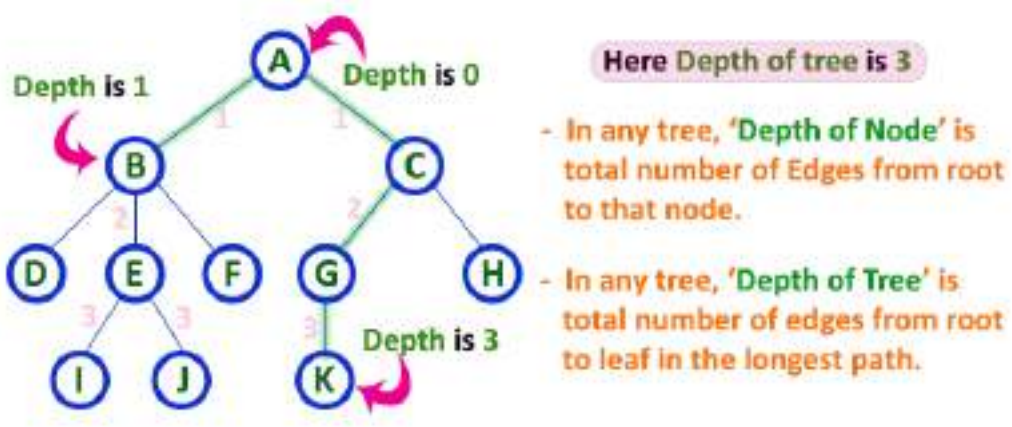
10. Height

In a tree data structure, the total number of egdes 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'.**



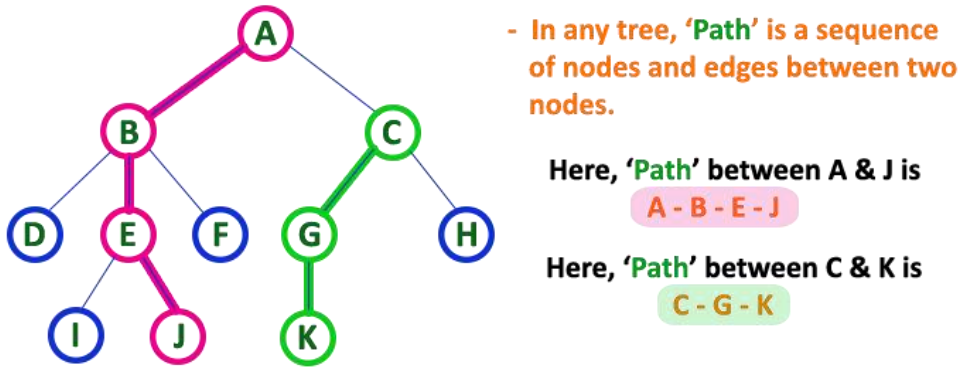
11. Depth

In a tree data structure, the total number of egdes 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'.**



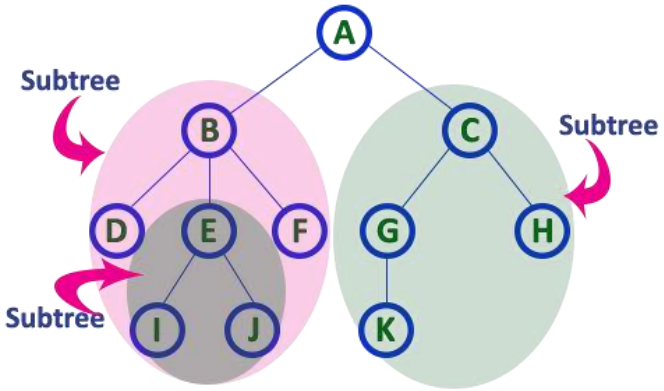
12. Path

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**.



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.

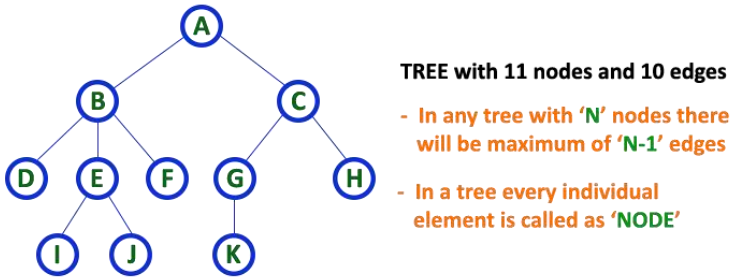


Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1. **List Representation**
2. **Left Child - Right Sibling Representation**

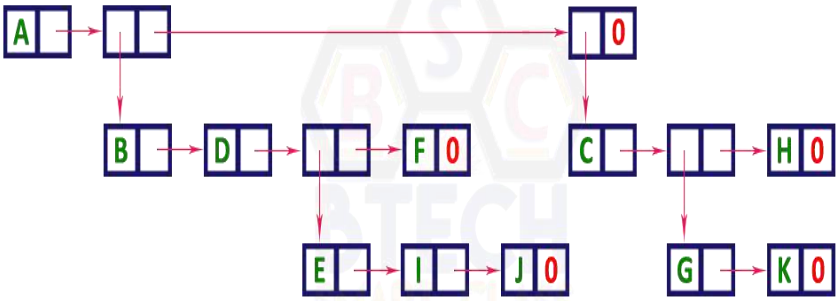
Consider the following tree...



1. List Representation

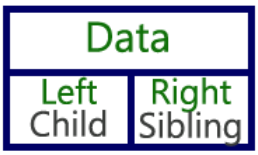
In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...



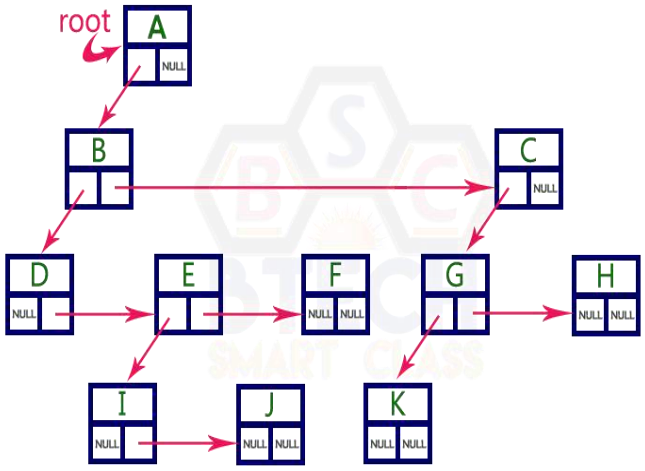
2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL.

The above tree example can be represented using Left Child - Right Sibling representation as follows...



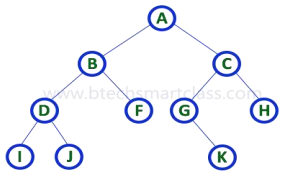
Binary Tree

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



There are different types of binary trees and they are...

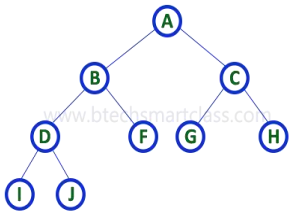
1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

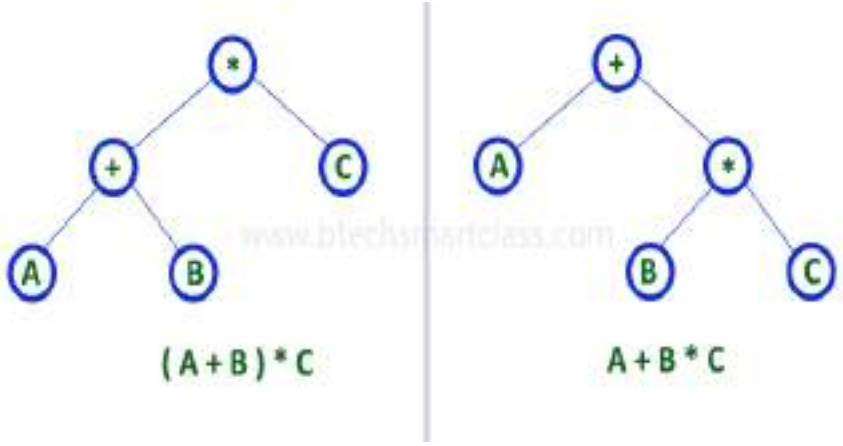
---

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



Strictly binary tree data structure is used to represent mathematical expressions.

Example

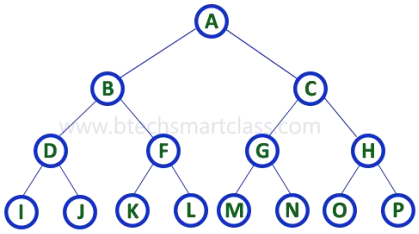


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

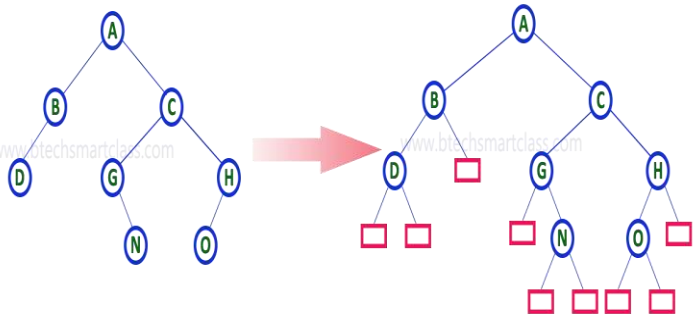
Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



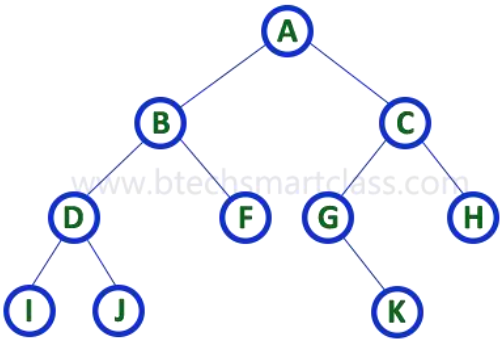
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

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

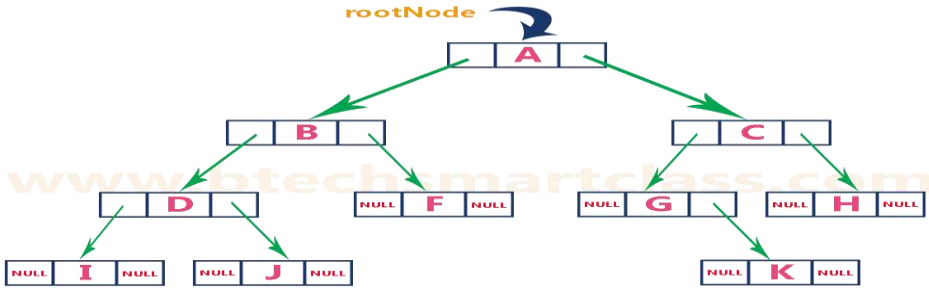
In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.
Consider the above example of binary tree and it is represented as follows...

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

We use 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 binary tree represented using Linked list representation is shown as follows...



Binary Tree Traversals

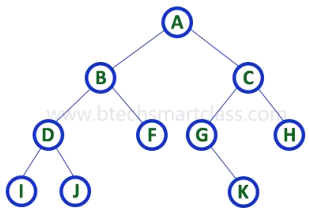
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

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 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 right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most 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 right part of the node A. In right of A again there is a subtree 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 right most 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 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 subtrees 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 left most 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 right most 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

2. 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

Properties of Binary tree

1. **The maximum number of nodes at level 'l' of a binary tree is 2^l.**
   Here level is the number of nodes on the path from the root to the node (including root and node). Level of the root is 0.
   This can be proved by induction.
   For root, l = 0, number of nodes = $2^0 = 1$
   Assume that the maximum number of nodes on level 'l' is $2^l = 2$
   Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^l = 4$ i.e. hen l = 2, nodes $2^2 = 4$
2. **The Maximum number of nodes in a binary tree of height 'h' is $2^h – 1$:**

   Note: Here the height of a tree is the maximum number of nodes on the root-to-leaf path. The height of a tree with a single node is considered as 1
   This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So the maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^{h-1}$. This is a simple geometric series with h terms and the sum of this series is $2^h – 1$.

   In some books, the height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} – 1$
3. **In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $Log_2(N+1)$:**
   Each level should have at least one element, so the height cannot be more than N.
   A binary tree of height 'h' can have a maximum of $2^h – 1$ nodes (previous property). So the number of nodes will be less than or equal to this maximum value

$N <= 2h – 1$ $2h >= N+1$

$\log_2(2^h) >= \log_2(N+1)$ (Taking log both sides) $h\log_2 2 >= \log_2(N+1)$ (h is an integer)

$h >= |\log_2(N+1)|$

So the minimum height possible is $|\log_2(N+1)|$

4. A Binary Tree with L leaves has at least $|\text{Log}_2 L|+ 1$ levels

A Binary tree has the maximum number of leaves (and a minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for the number of leaves L.

- ✓ $L <= 2^{l-1}$ [From Point 1]
- ✓ $l = |\text{Log}_2 L| + 1$
- ✓ where l is the minimum number of levels.

5. In a Binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children:

$L = T + 1$

Where L = Number of leaf nodes

T = Number of internal nodes with two children

PROOF:

No. of leaf nodes (L) i.e. total elements present at the bottom of tree = 2h-1 (h is height of tree) No. of internal nodes = {total no. of nodes} – {leaf nodes} = { 2h – 1 } – {2h-1} = 2h-1 (2-1) – 1 = 2h-1 – 1 So , L = 2h-1 T = 2h-1 – 1 Therefore L = T + 1 Hence proved.

6. In a non-empty binary tree, if n is the total number of nodes and e is the total number of edges, then e = n-1:

Every node in a binary tree has exactly one parent with the exception of the root node. So if n is the total number of nodes then n-1 nodes have exactly one parent. There is only one edge between any child and its parent. So the total number of edges is n-1.

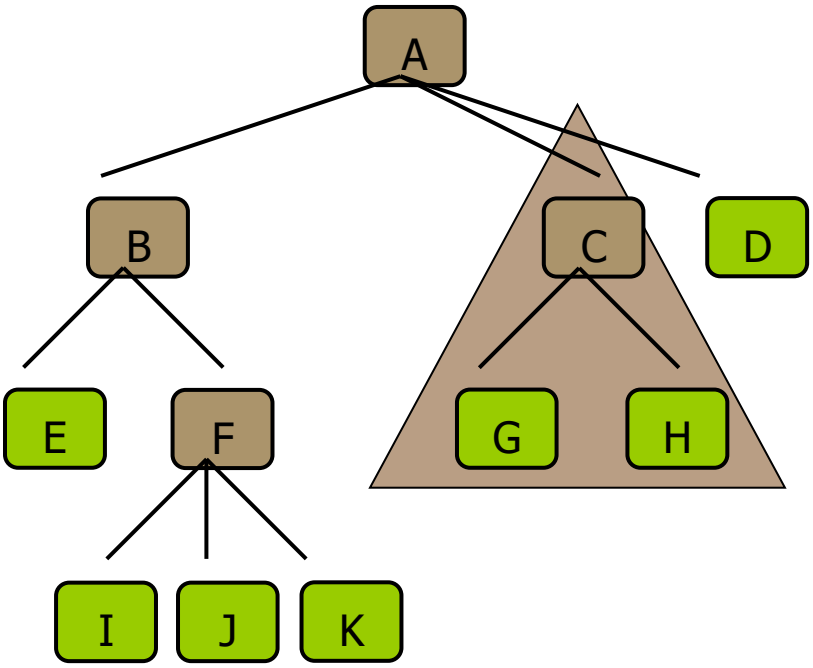
Binary Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)

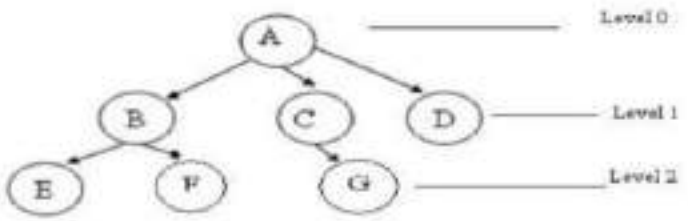
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grandgrandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node

(3)

- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum number of its node.
- The **successor nodes** of a node are called its *children*
- The **predecessor node** of a node is called its *parent*
- **Subtree**: tree consisting of a node and its descendants



CONTD:



- A is the root node
- B is the parent of E and F
- D is the sibling of B and C
- E and F are children of B
- E, F, G, D are external nodes or leaves
- A, B, C are internal nodes
- Depth of F is 2
- the height of tree is 2
- the degree of node A is 3
- The degree of tree is 3

Common types of Binary tree

- Full Binary Tree: A Binary Tree is full if every node has 0 or 2 children.
  - *In a Full Binary, number of leaf nodes is number of internal nodes plus 1*

    L = I + 1

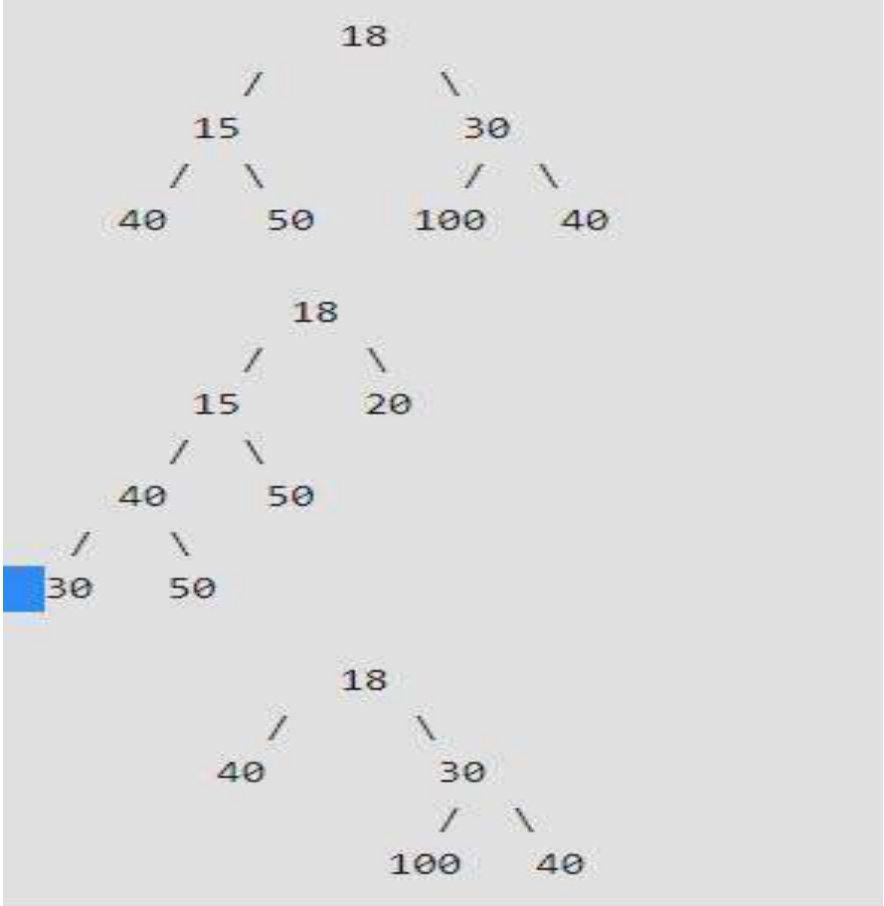
    Where

    L = Number of leaf nodes,

    I = Number of internal nodes

- A full binary tree of a given height k has $2^{k+1}-1$ nodes.

  Full Binary Tree

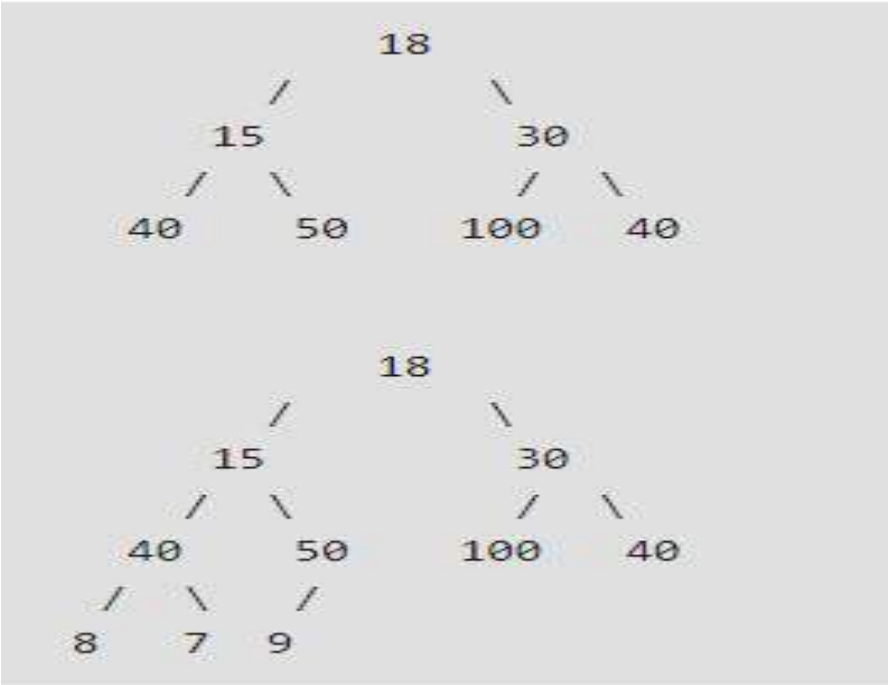


Complete Binary Tree

- A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A complete binary tree is just like a full binary tree, but with two major differences
- **Every level must be completely filled**

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

Complete Binary Tree



PERFECT BINARY TREE

Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.
Following are examples of Perfect Binary Trees.

- A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ nodes.

STRICTLY BINARY TREE

Strictly Binary tree with n leaves always contains 2n-1 nodes

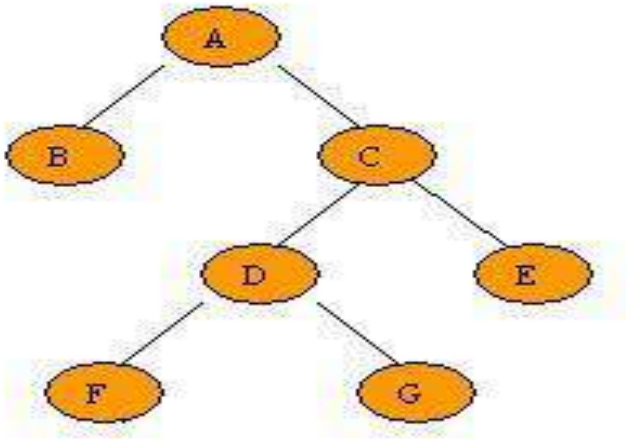
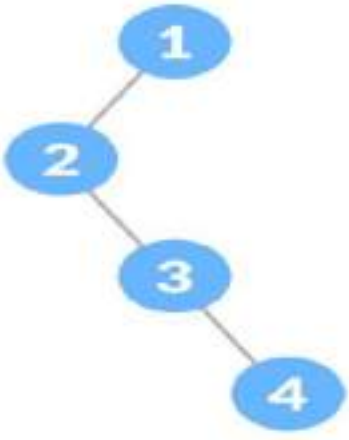


Fig .5.1.4 Strictly binary tree

Degenerate or Pathological Tree

A degenerate or pathological tree is the tree having a single child either left or right.



Degenerate Binary Tree

SKEWED BINARY TREE :

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left- skewed binary tree** and **right-skewed binary tree**.

Skewed Binary Tree

BALANCED BINARY TREE: It is a type of binary tree in which the difference between the height of the left subtree and the right subtree for each node is either 0 or 1.

Balanced Binary Tree

df = |height of left child - height of right child|

UnBalanced Binary Tree

BINARY TREE REPRESENTATION:

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

```
class node
{
 int data;
 node *left;
 node *right;
 };
```

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

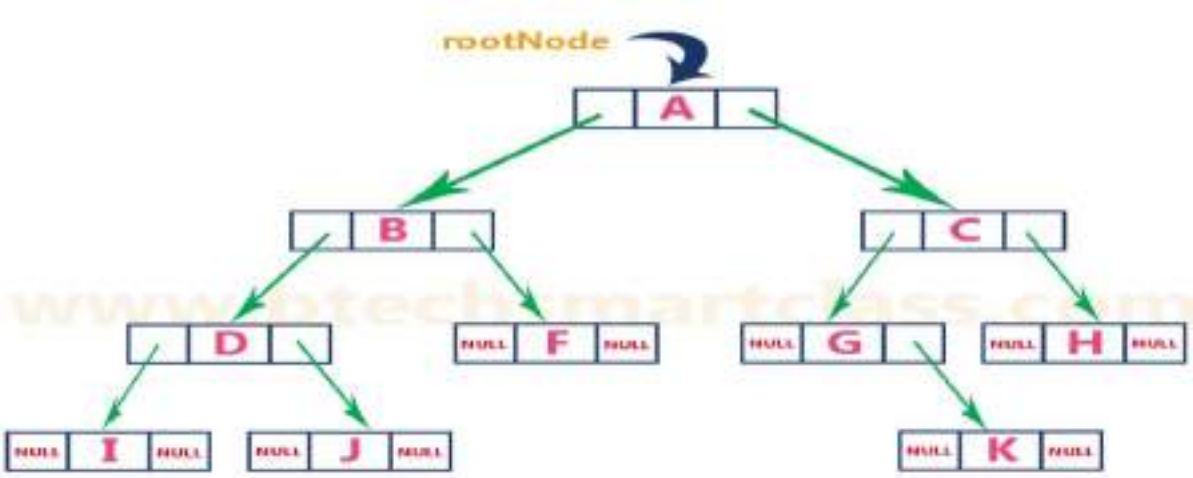
1. **Array Representation**
2. **Linked List Representation**

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

In general, if a node is stored at the ith location then it's left and right child is stored at 2i+1 and 2i+2 location respectively.

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...

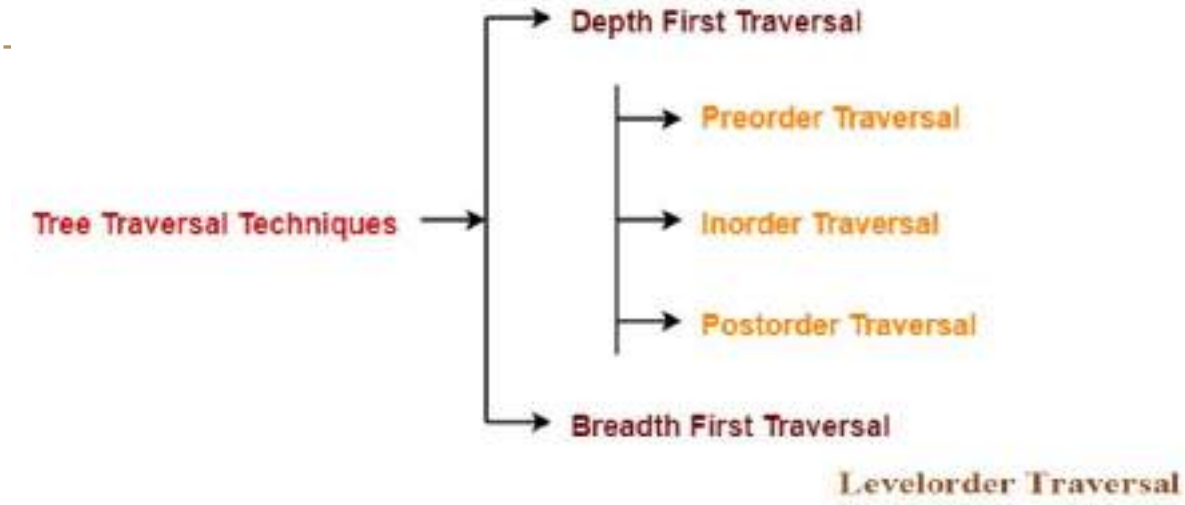


TREE TRAVERSALS:

- A ***traversal*** of a tree requires that each node of the tree be ***visited*** once

- Linear data structures like Stack, Queue, linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node.

- Various tree traversal techniques are-



- Depth FIRST TRAVERSAL- Following three traversal techniques fall under Depth First Traversal-

    1. Preorder traversal
    2. Inorder Traversal
    3. Postorder Traversal

Preorder Traversal: **Algorithm-**

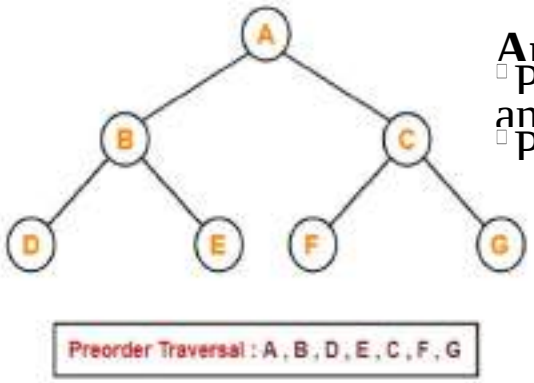
1. Visit the root

2. Traverse the left sub tree i.e. call Preorder (left sub tree)

3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

Consider the following example:



Applica

- Preorder traversal is used to get an
- Preorder traversal is used to create

INORDER TRAVERSAL :

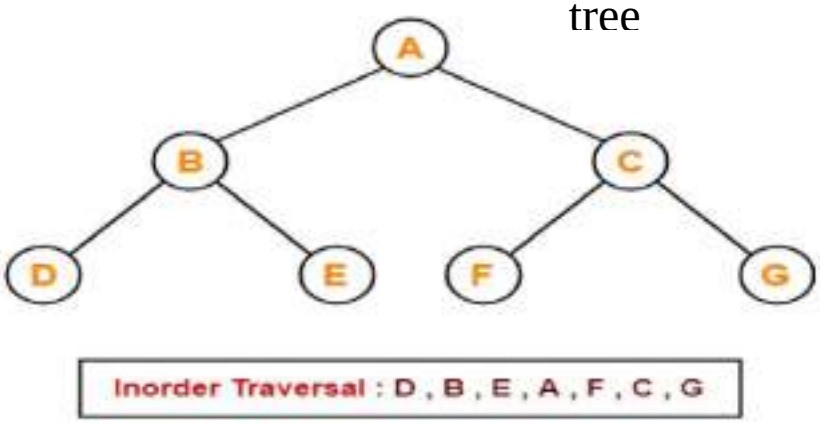
Algorithm-

1. Traverse the left sub tree i.e. call Inorder (left sub tree)
2. Visit the root
3. Traverse the right sub tree i.e. call Inorder (right sub tree)

Left → Root → Right

Example-Application-Inorder traversal is used to get infix expression of an expression

tree

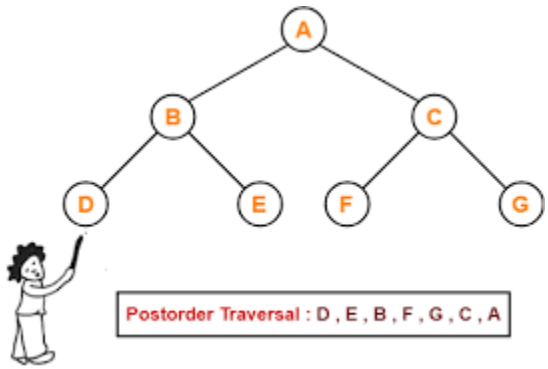


POSTORDER TRAVERSALS –Algorithm-

1. Traverse the left sub tree i.e. call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root

Appilcations:1. Postorder traversal is used to get postfix expression of an expression tree. 2. Postorder traversal is used to delete the tree. 3. This is because it deletes the children first and then it deletes the parent.

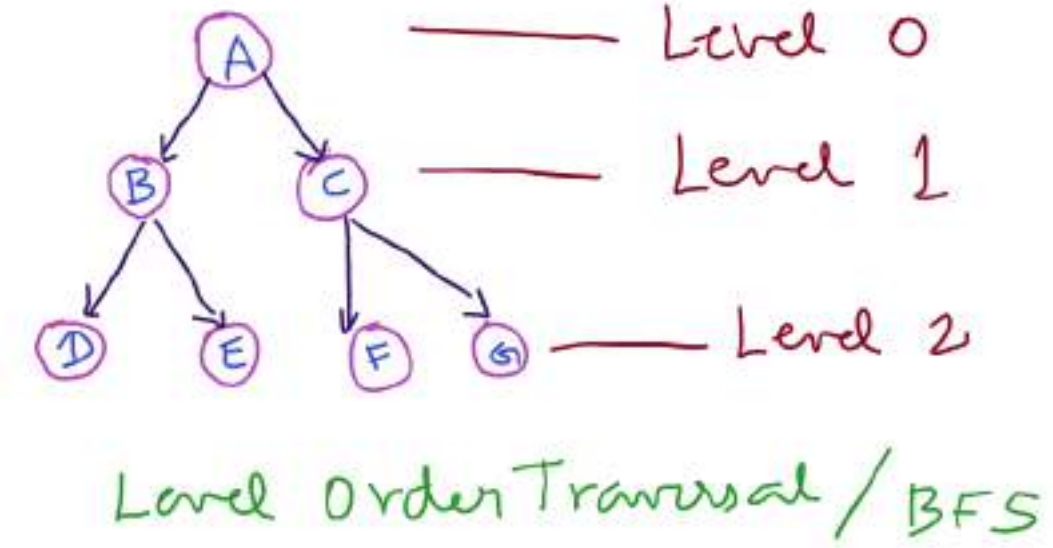


Breadth First Traversal-

- ✓ Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- ✓ Breadth First Traversal is also called as " Level order traversal".

Applications:

Level order traversal is used to print the data in the same order as stored in array representation of a complete binary tree.



LEVEL OF ORDER TRAVERSAL : A B C D E F G

THREADED BINARY TREE: 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**.
- ✓ There are two types of threaded binary trees.

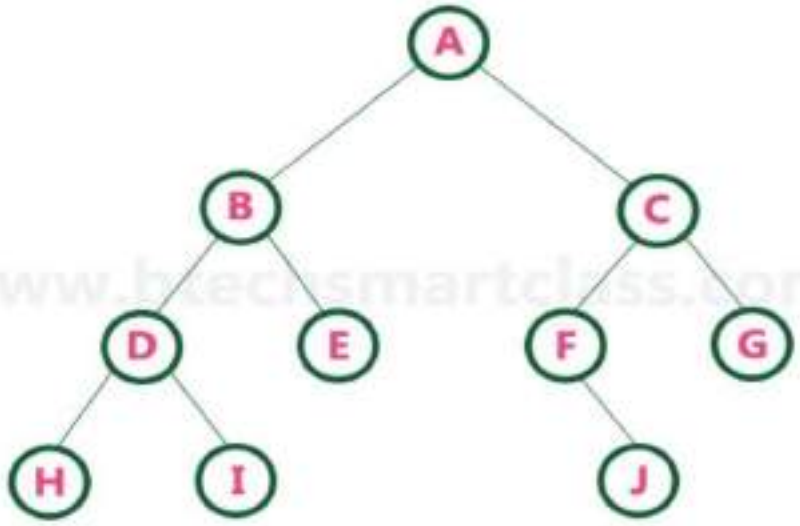
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively.

- ✓ The predecessor threads are useful for reverse inorder traversal and postorder traversal.
- ✓ The threads are also useful for fast accessing ancestors of a node.
- ✓ **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.



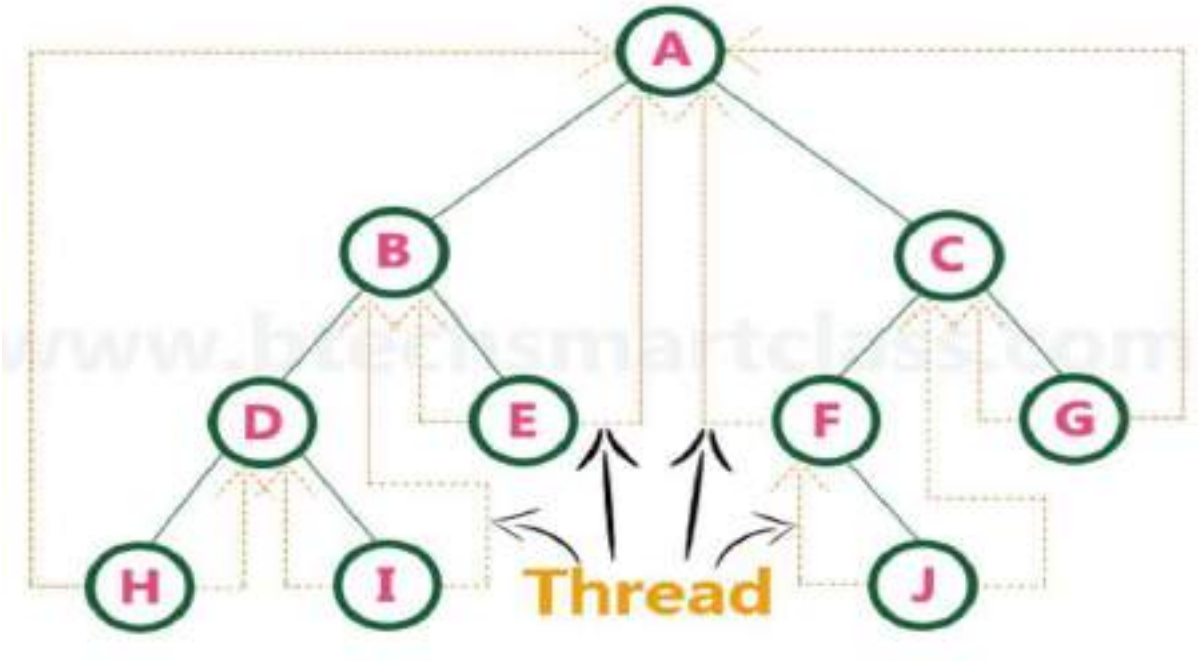
✓ Consider the following binary tree...

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 inorder 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.

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...

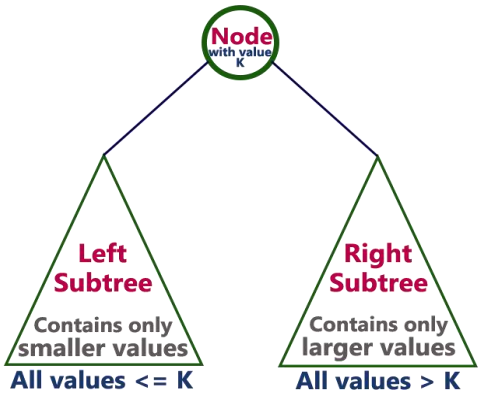
- ✓ **Search Operation - O(n)**
- ✓ **Insertion Operation - O(1)**
- ✓ **Deletion Operation - O(n)**

✓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

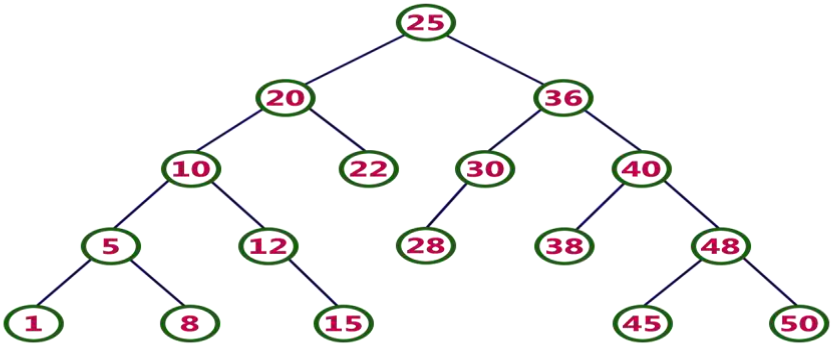
✓ **Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.**

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as



Example:

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

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

Search

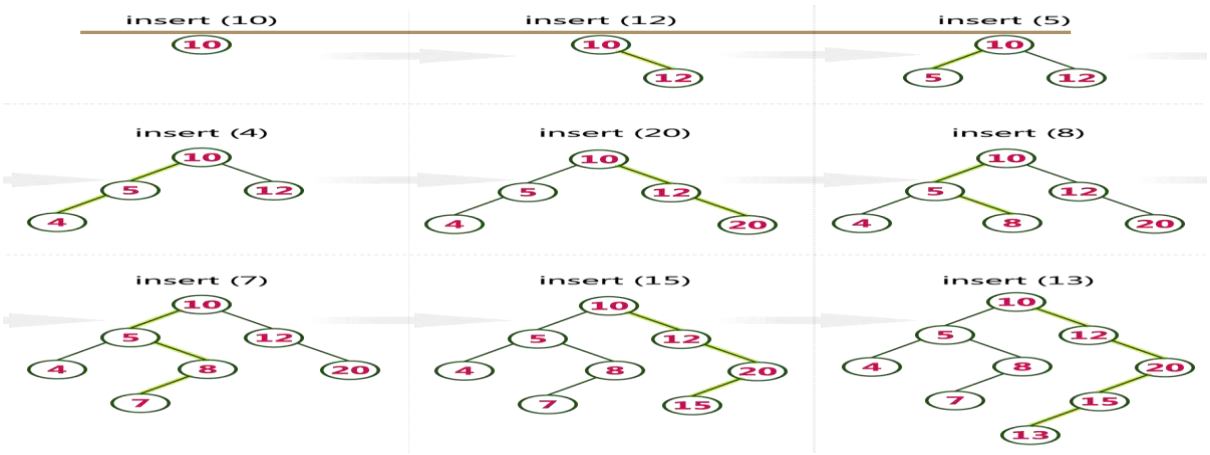
Insertion

Deletion

Example

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

- ✓ **10,12,5,4,20,8,7,15 and 13**



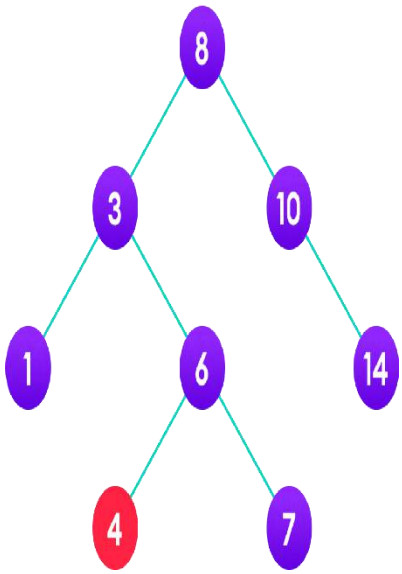
Deletion Operation :

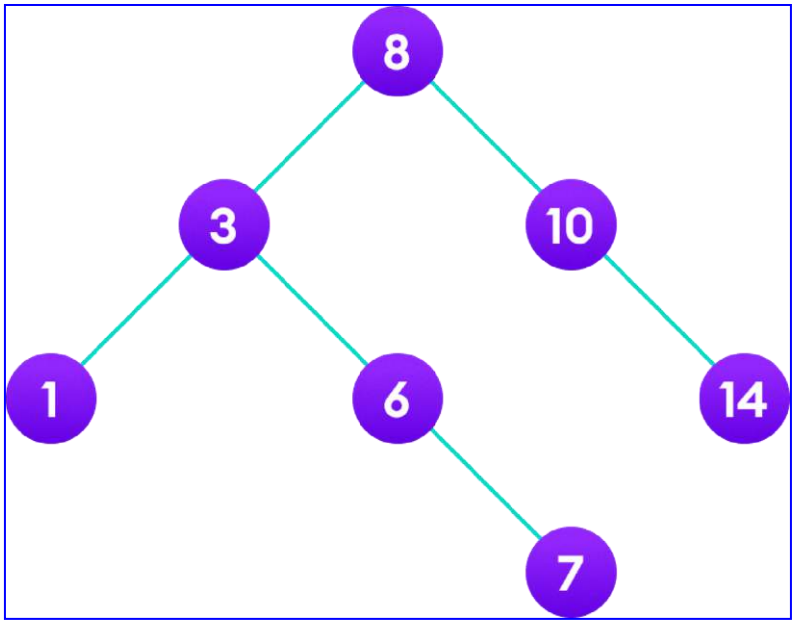
- ✓ There are three cases for deleting a node from a binary search tree.

Case I :

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

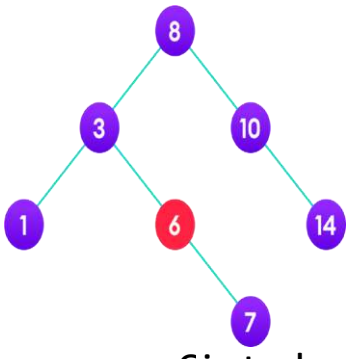
- In the first case, the node to be deleted is the leaf node. the node from the tree.

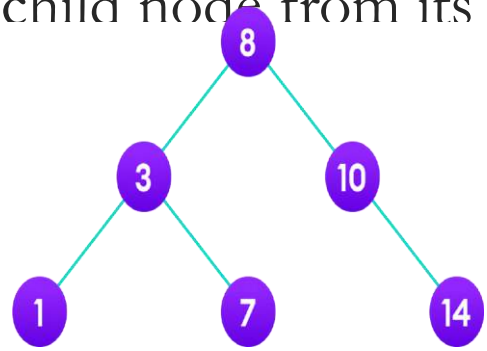




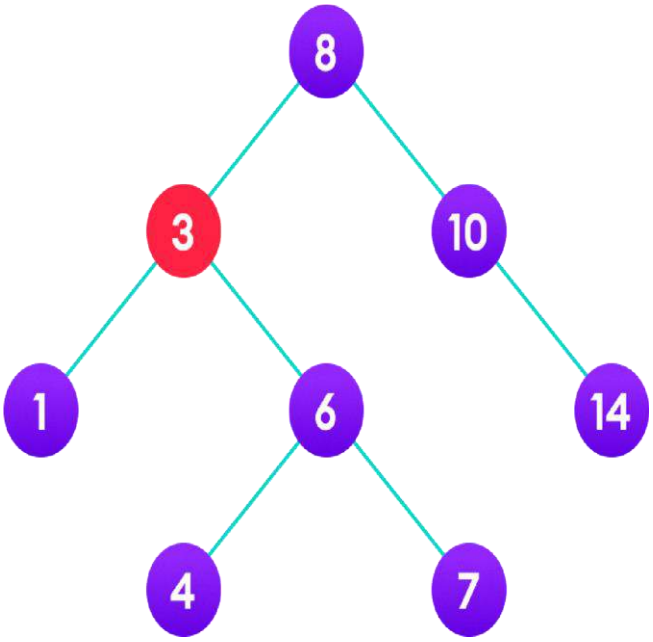
:

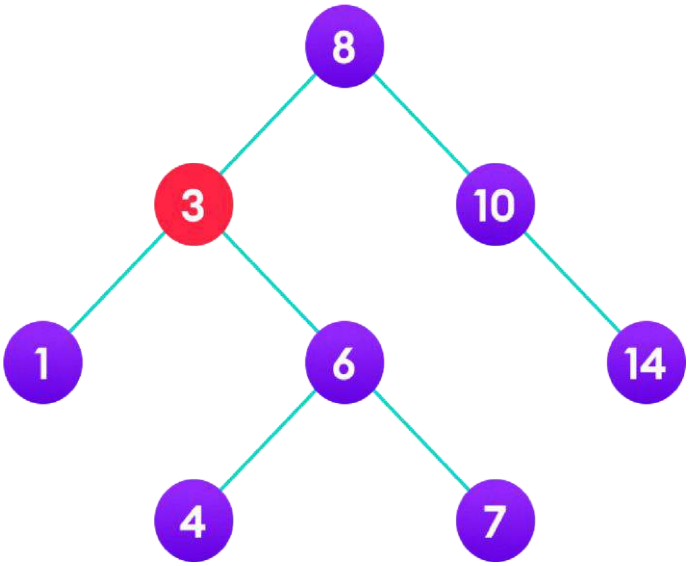
- Replace that node with its
- Remove the child node from its original





6is to be copy the value of its child to the node and





3 is to be deleted Copy the value of the

CASE II

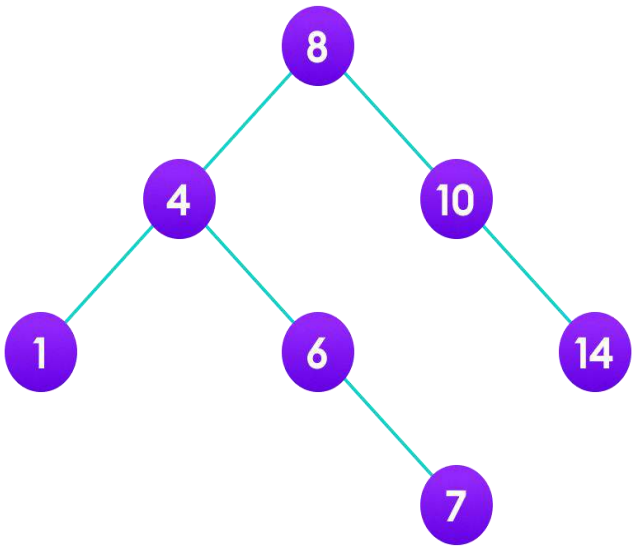
CASE III

In the third case, the node to be deleted has two children. In such a case follow the steps below

Get the inorder successor of that node.

✓ Replace the node with the inorder successor.

- ✓ Remove the inorder successor from its original position.



DELETE INORDER SUCCESSOR.

BALANCED BINARY TREE :

- ✓ A **height-balanced binary tree** is defined as a binary tree in which the **height** of the left and the right subtree of any node differ by not more than 1.
- ✓ **AVL tree, red-black tree are examples of height-balanced trees.**

AVL TREE :

✓ 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 height of left and right subtrees of every node in the tree is either -1, 0 or +1.

✓ In an AVL tree, every node maintains an extra information known as **balance factor**.

✓ The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

✓ 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.**

✓ Balance factor of a node is the difference between the heights of the left and right subtrees of that node.

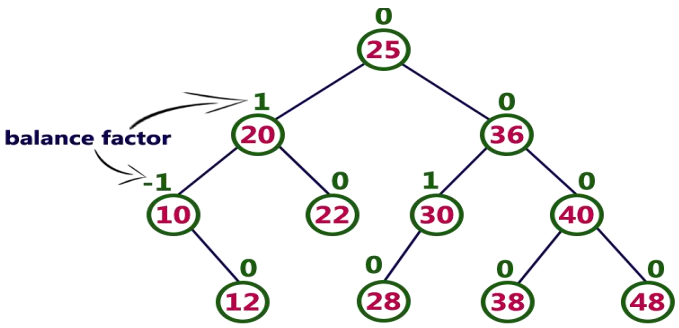
✓ The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**.

✓ In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL TREE

The below 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

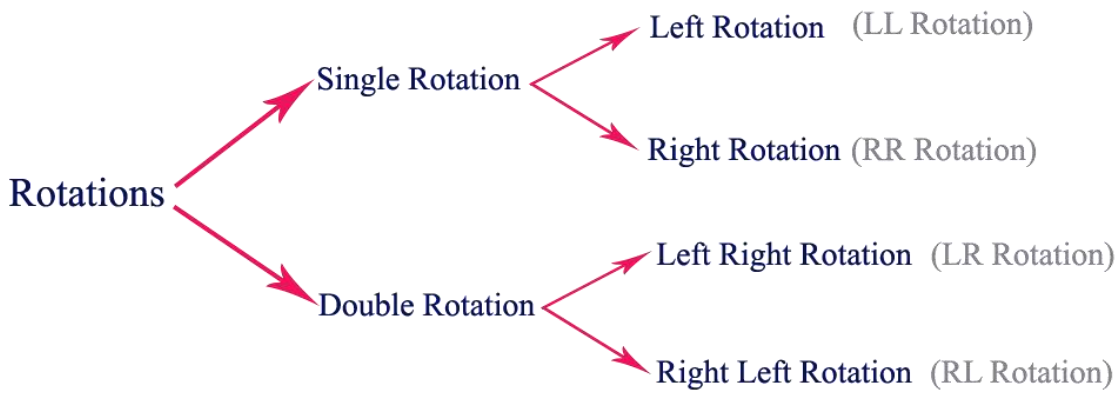
✓ conclude the operation otherwise we must make it balanced.

✓ 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.

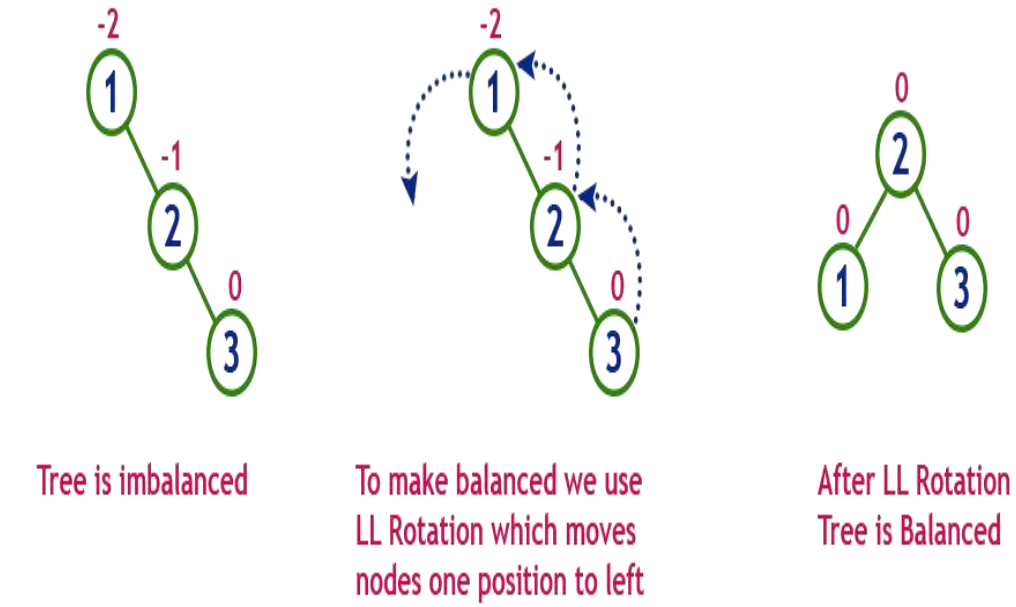


Single Left Rotation (LL Rotation)

- 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

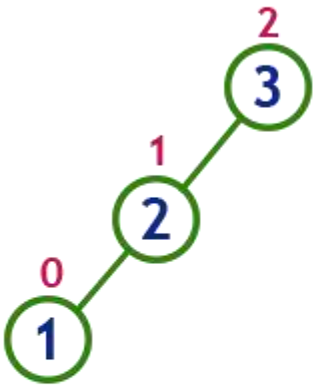
- insert 1, 2 and 3



Single Right Rotation (RR Rotation)

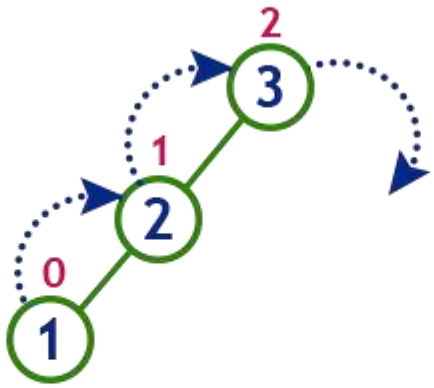
- In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL

insert 3, 2 and 1



Tree is imbalanced

because node 3 has balance factor 2



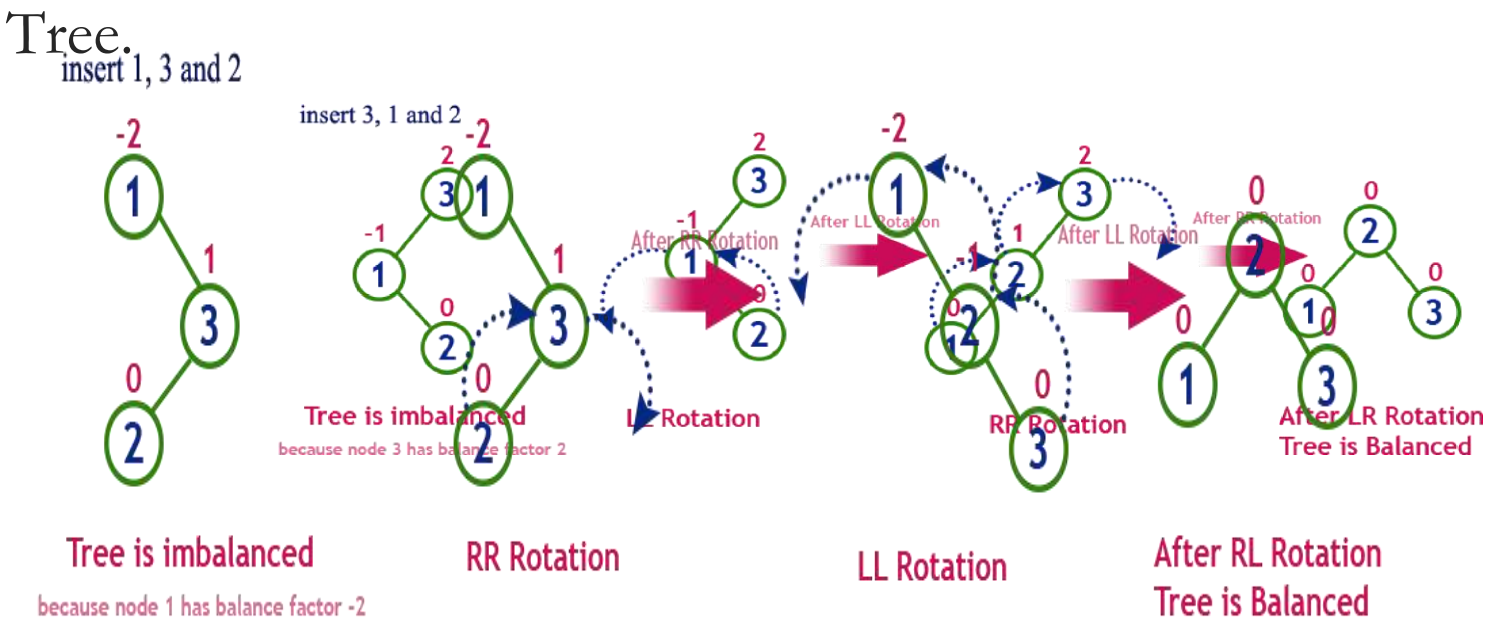
To make balanced we use RR Rotation which moves nodes one position to right

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...

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

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.

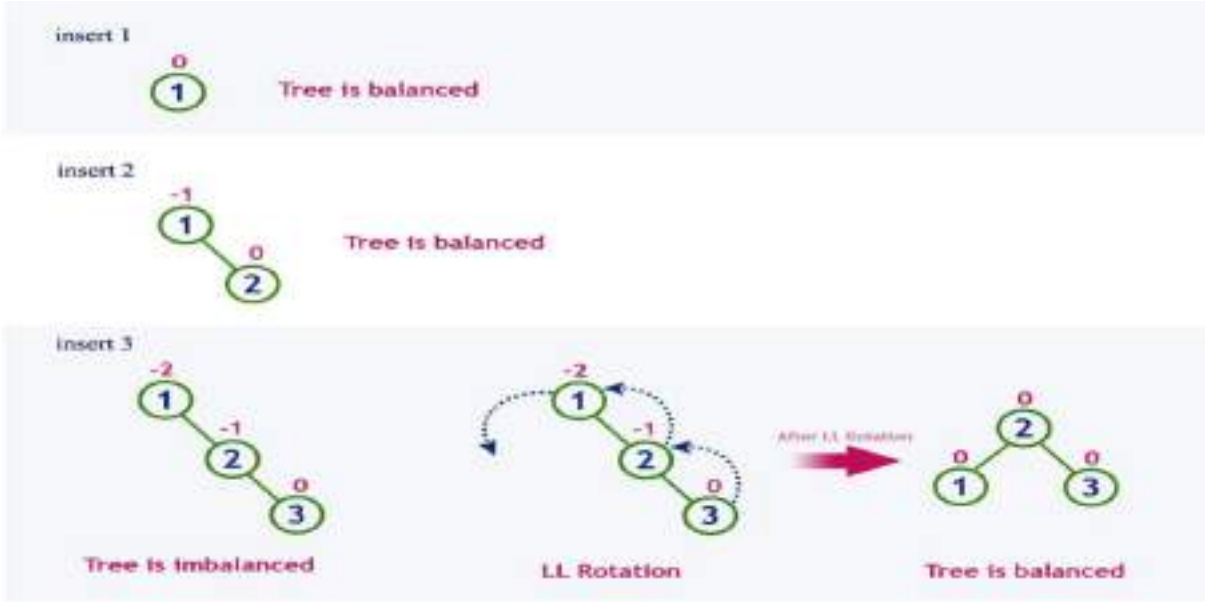
Insertion Operation in AVL Tree

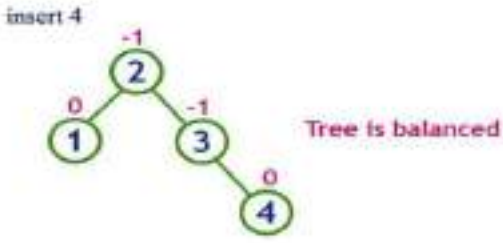
- In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

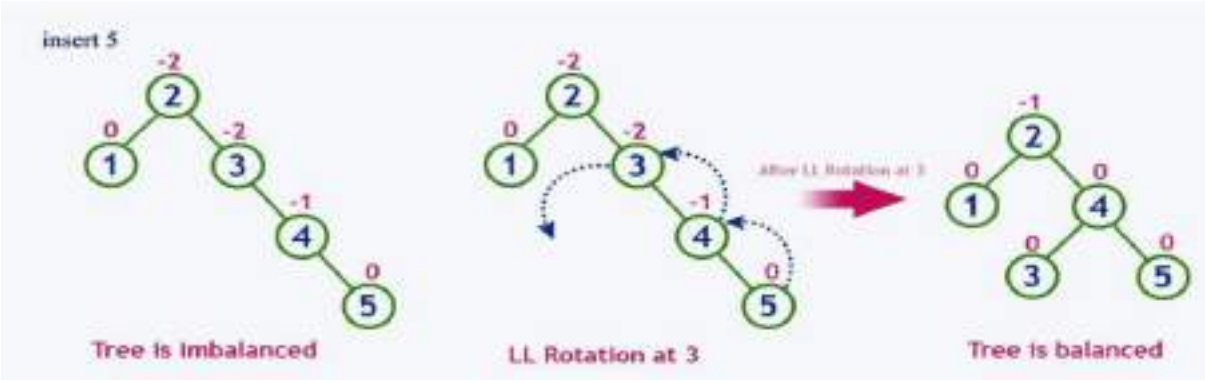
- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

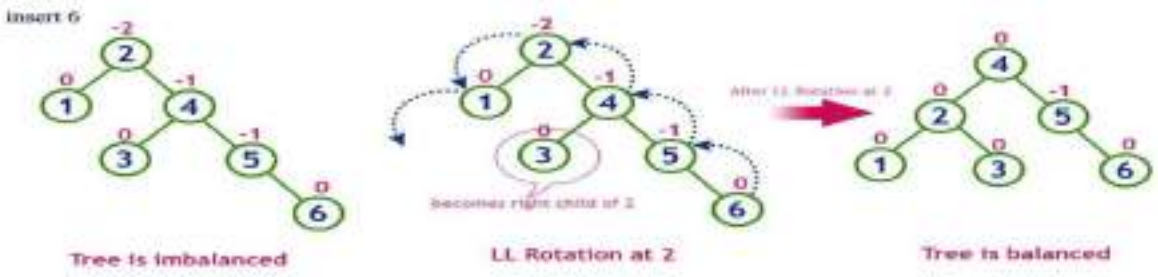
Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

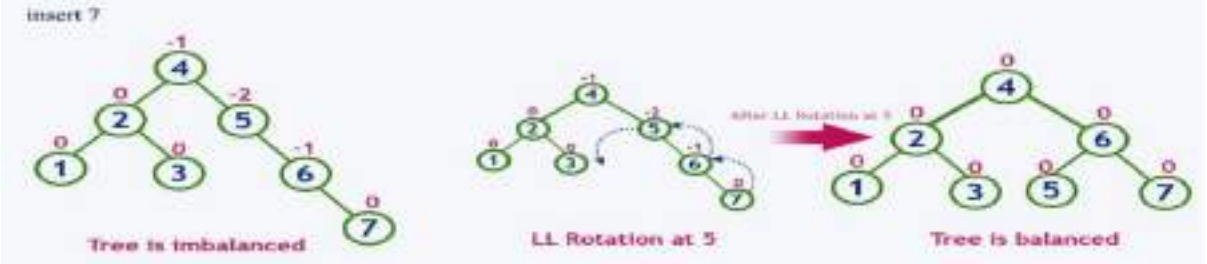
Example : Construct an AVL Tree by inserting numbers from 1 to 8.


insert 1
0
1
Tree is balanced
insert 2
-1
1
0
2
Tree is balanced
insert 3
-2
1
-1
2
0
3
Tree is imbalanced
-2
1
-1
2
0
3
LL Rotation
0
2
0
1
3
Tree is balanced

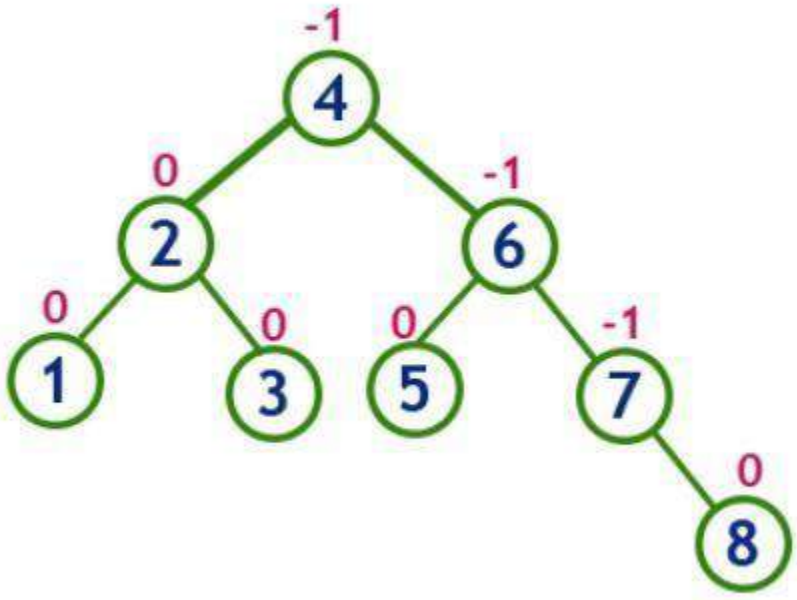

insert 4
-1
2
0
1
-1
3
0
4
Tree is balanced


insert 5
-2
2
0
1
-2
3
-1
4
0
5
Tree is imbalanced
-2
2
0
1
-2
3
-1
4
0
5
LL Rotation at 3
-1
2
0
1
0
4
0
3
0
5
Tree is balanced





insert 8



Tree is balanced

Q: Construct an AVL tree having the following elements H, I, J, B, A, E, C, F, D, G, K, L
Note: Consider alphabetical order.

Q: Construct an AVL tree having the following elements 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

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.