UNIT-IV

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.

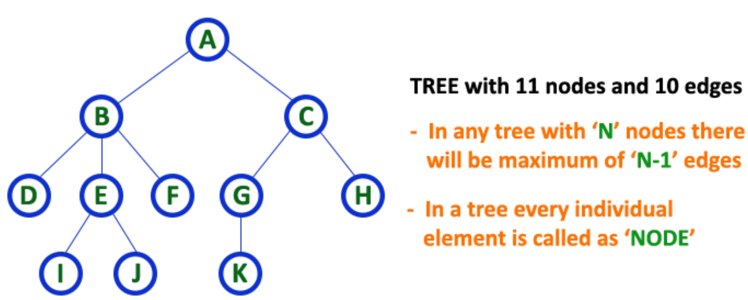
A tree data structure can also be defined as follows...

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.

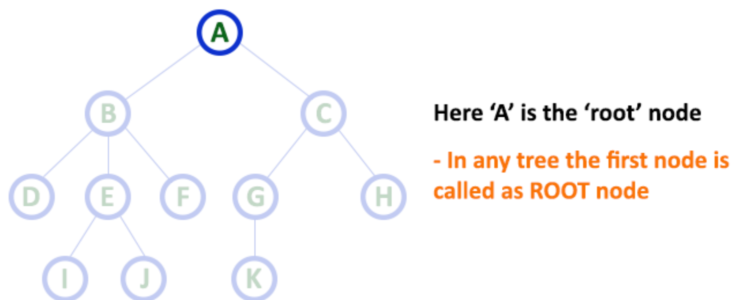
Example



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

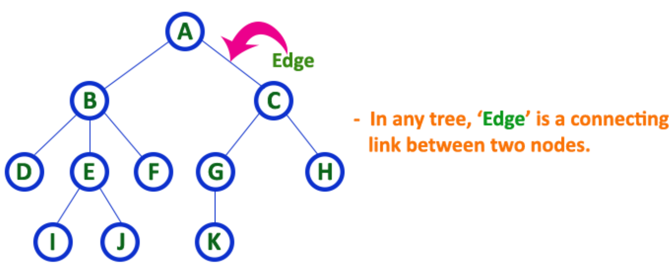
1. Root

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.



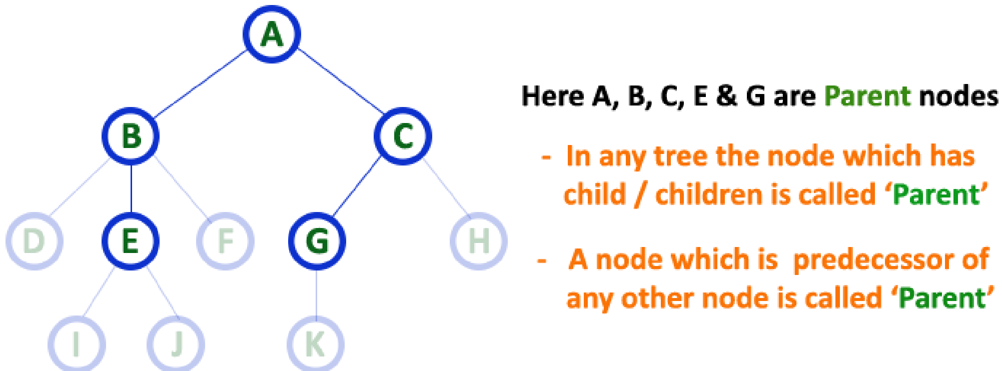
2. Edge

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.



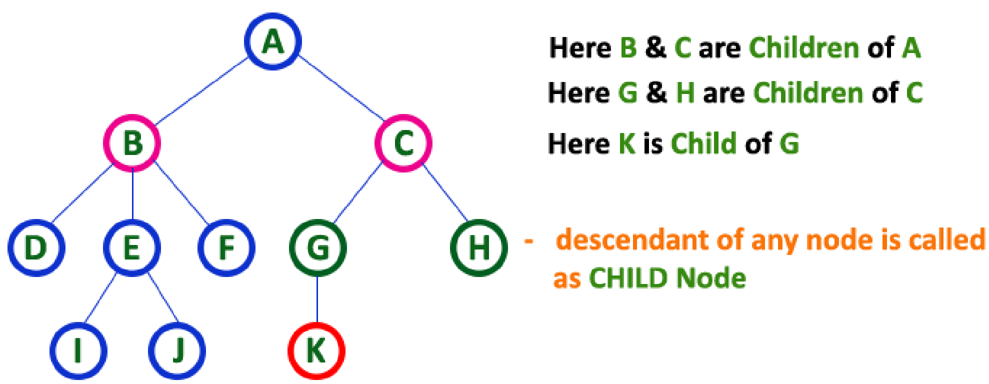
3. Parent

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**".



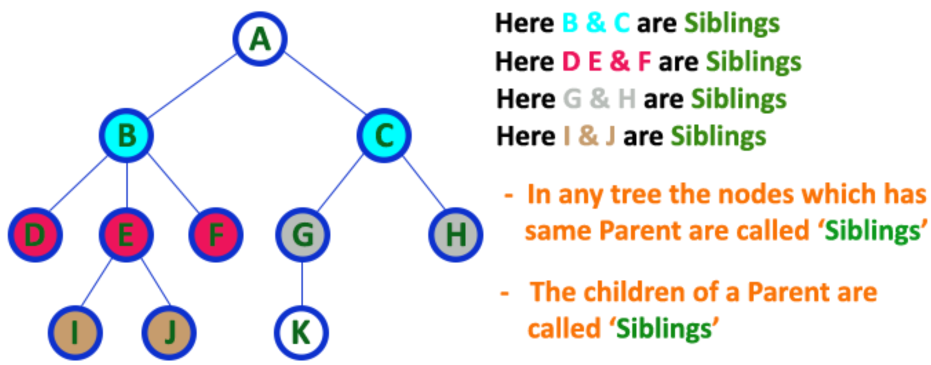
4. Child

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

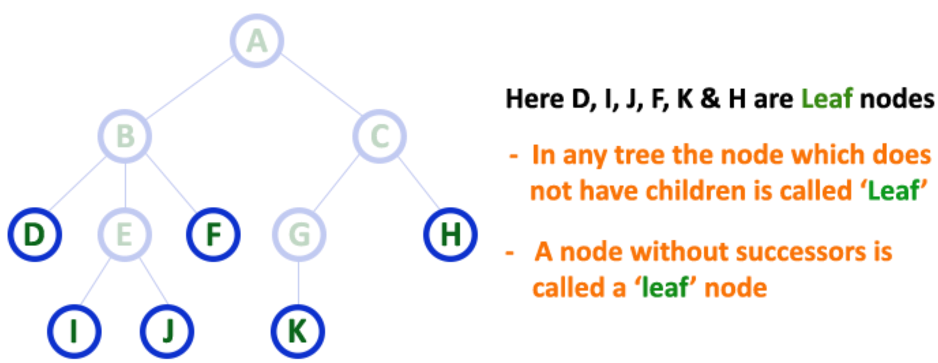
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



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.

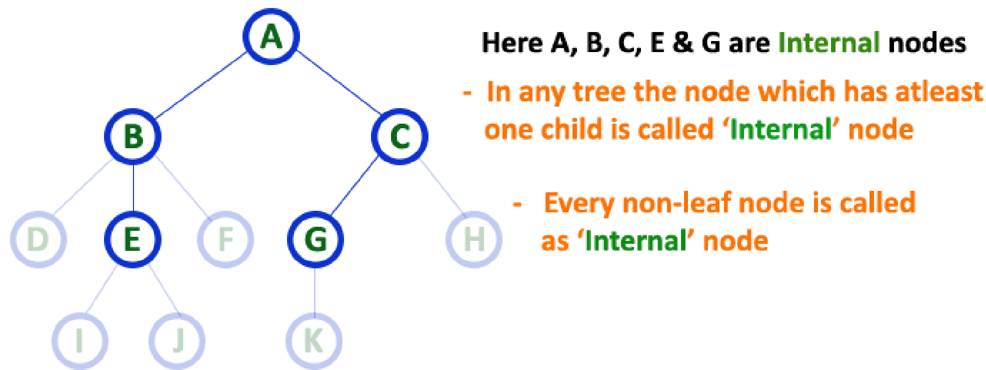
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



7. Internal Nodes

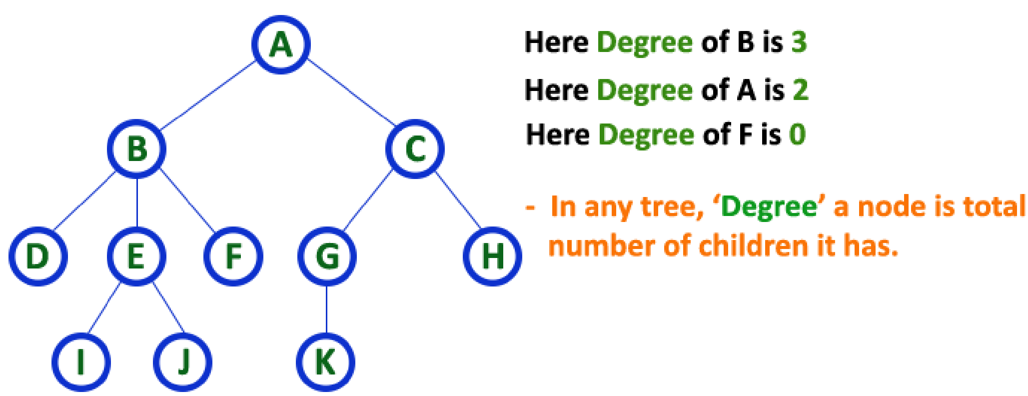
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

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.



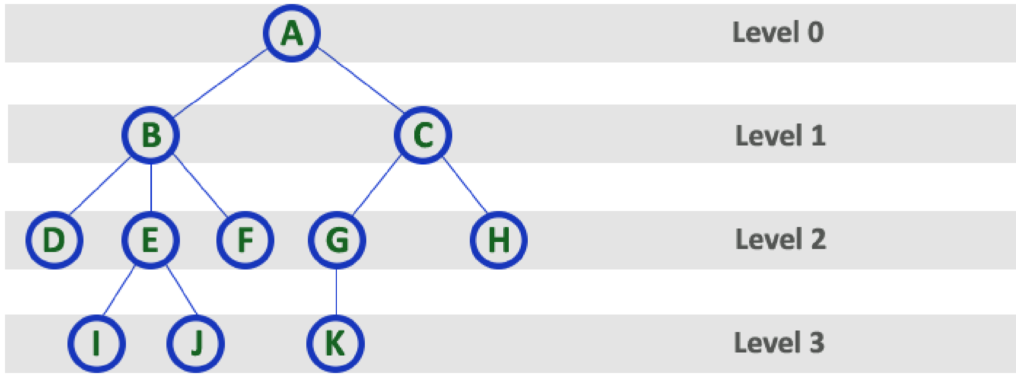
8. Degree

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**'



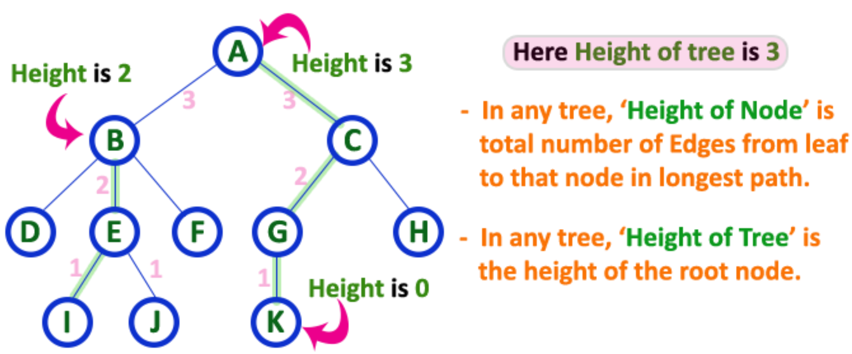
9. Level

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).



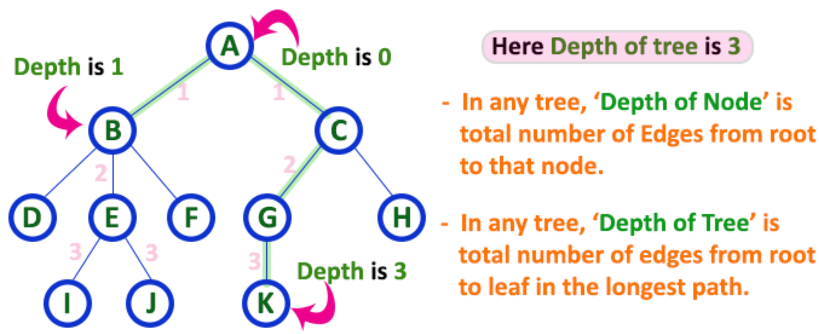
10. Height

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'.**



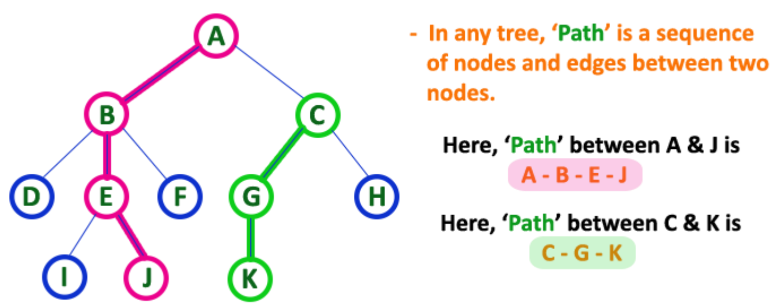
11. Depth

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'.**



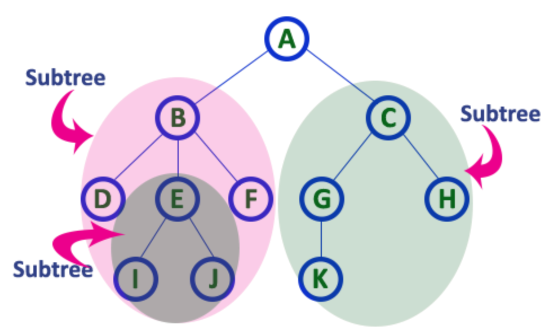
12. Path

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.



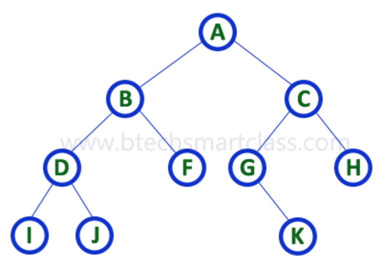
Binary Tree

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



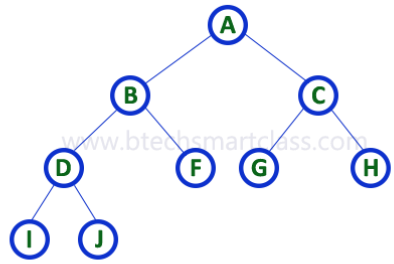
There are different types of binary trees and they are...

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...

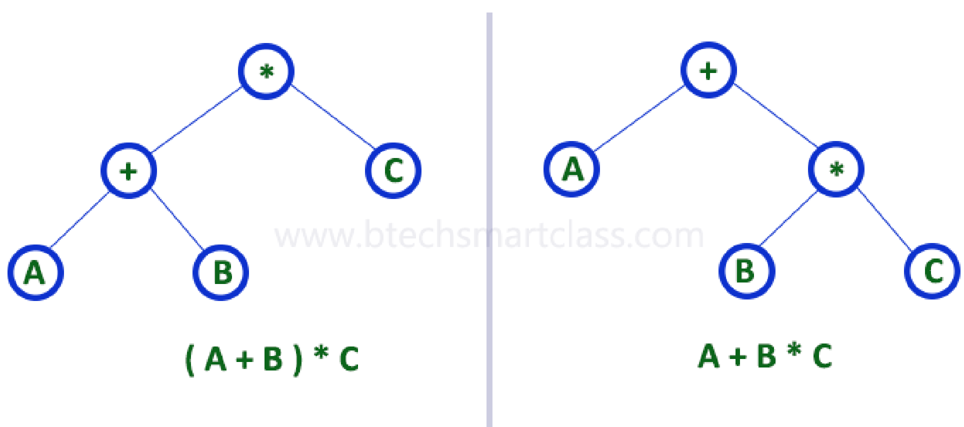
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

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

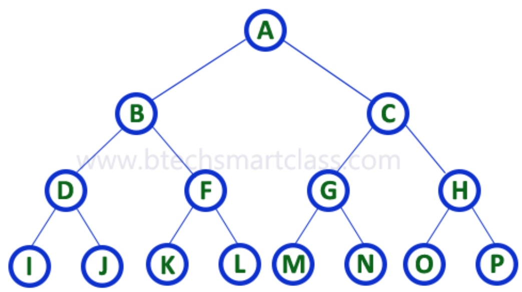


2. Complete Binary Tree

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.

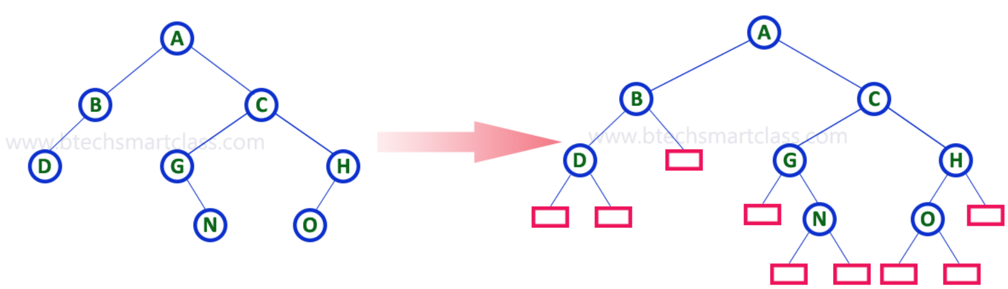
Complete binary tree is also called as **Perfect Binary Tree**



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.



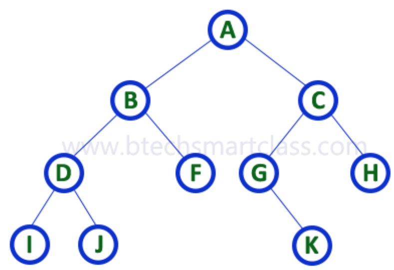
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes .

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

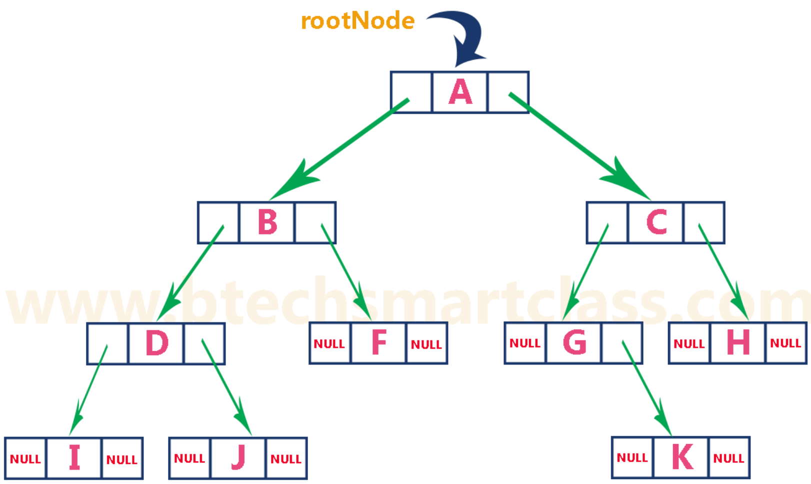
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

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

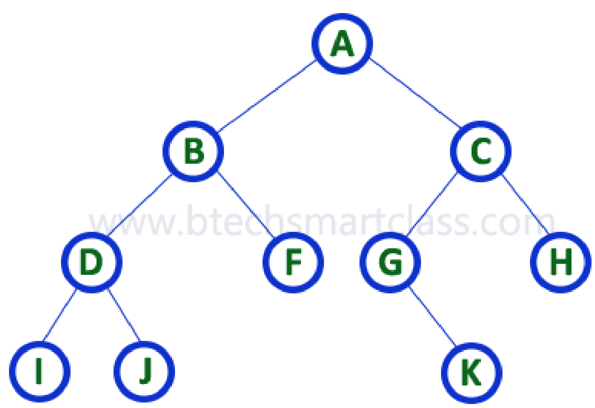
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

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

Tree variants(Types Of Trees):

1.General Tree

2.Binary Tree

3.Binary Search Tree

4.AVL Tree

5.B Tree

Applications of Trees:

- Decision Making
    - Next Move in games
    - Computer chess games build a huge tree (training) which they prune at runtime using heuristics to reach an optimal move.
- Networking
    - Router algorithms -Network Routing, where the next path/route of the packet is determined
    - Social networking is the current buzzword in CS research. It goes without saying that connections/relations are very naturally modeled using graphs. Often, trees are used to represent/identify more interesting phenomena.
- Representation
    - Chemical formulas representation
    - XML/Markup parsers use trees
    - Producers/consumers often use a balanced tree implementation to store a document in memory.
- Manipulate Hierarchical Data
    - Make information easy to search
    - Manipulate sorted lists of data.
- Workflow
    - As a workflow for compositing digital images for visual effects.
- Organizing Things
    - Folders/ files in the Operating system
    - HTML Document Object Model (DOM)
    - Company Organisation Structures
    - PDF is a tree-based format. It has a root node followed by a catalog node followed by a pages node which has several child page nodes.
- Faster Lookup
    - Auto-correct applications and spell checker
    - Syntax Tree in Compiler
- Task Tracker
    - Undo function in a text editor

Binary Search Tree

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

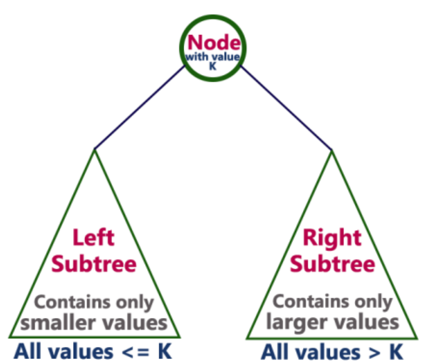
A binary tree has the following time complexities...

1. **Search Operation - O(logn)**
2. **Insertion Operation - O(logn)**
3. **Deletion Operation - O(logn)**

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

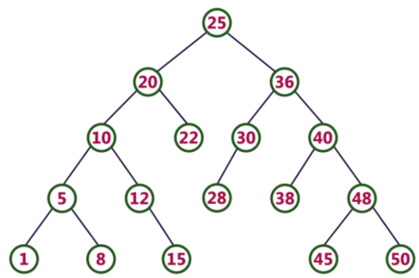
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

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



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.



NOTE: Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

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 matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, 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 found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.

- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element 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 set **root** to **newNode.**
- **Step 4:** If the tree is **Not Empty**, then check whether 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 step until we reach to a **leaf** node (e.i., reach to NULL).
- **Step 7:** After reaching a leaf node, then isert the newNode as **left child** if newNode is **smaller or equal** to that leaf 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 has follwing 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 and child nodes.
- **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...

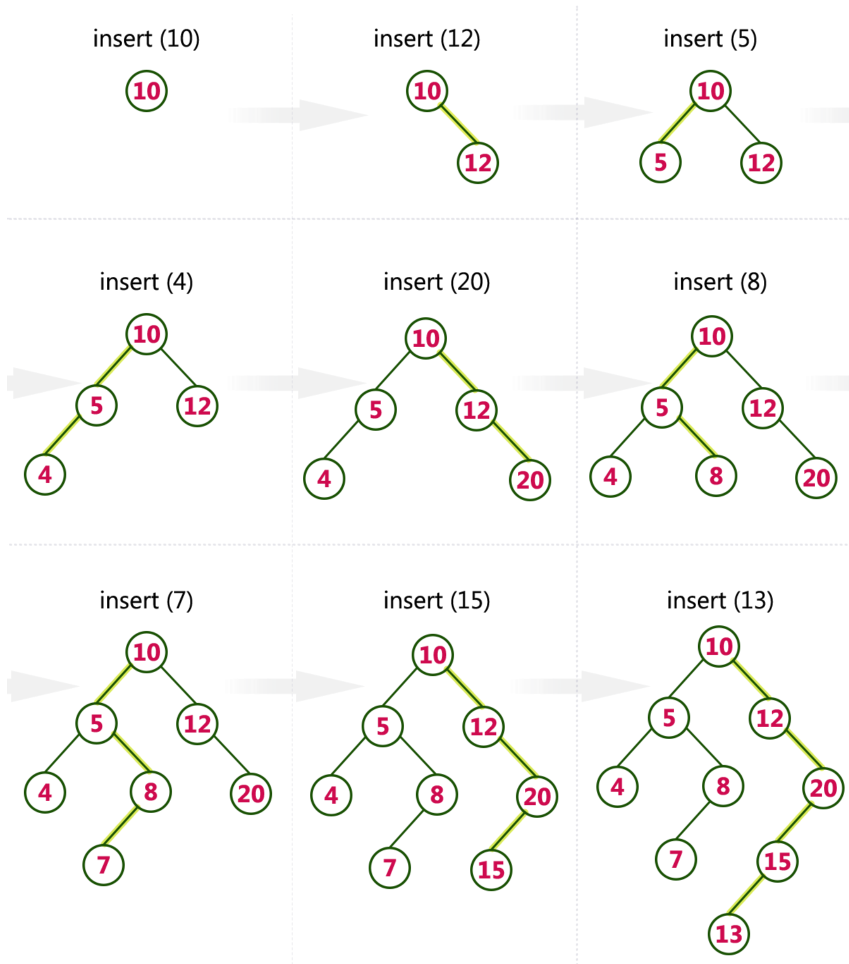
- **Step 1:** **Find** the node to be deleted using **search operation**
- **Step 2:** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3:** **Swap** both **deleting node** and node which found in above step.
- **Step 4:** Then, check whether deleting node came to **case 1** or **case 2** else goto steps 2
- **Step 5:** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6:** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7:** Repeat the same process until 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...



AVL Tree

AVL tree is a self 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 hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as **balance factor**. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

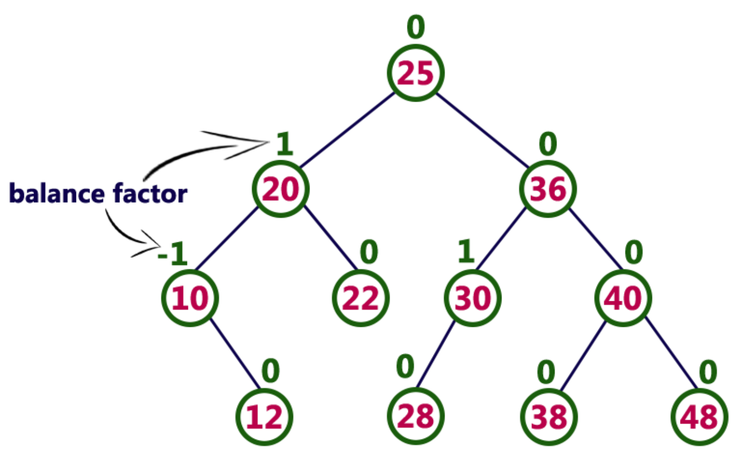
An AVL tree is defined as follows...

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 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 are calculating as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example



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 all the Binary Search Trees need not to be AVL trees.

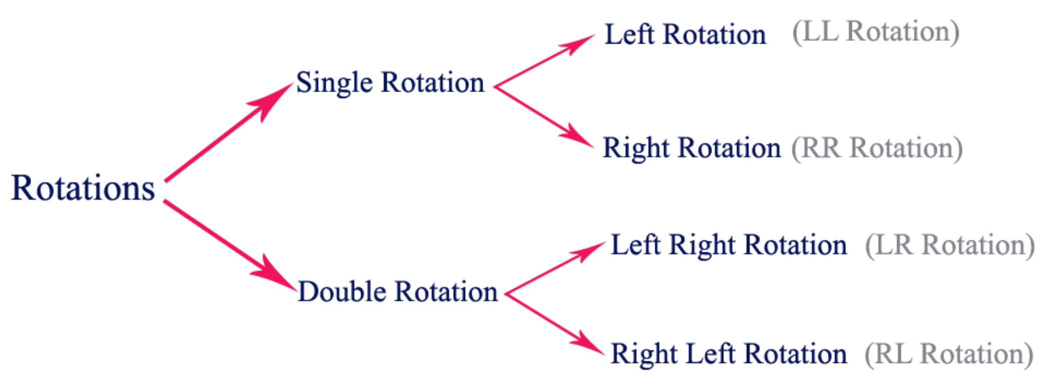
AVL Tree Rotations

In AVL tree, after performing every operation 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. We use **rotation** operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

Rotation operations are used to make a tree balanced.

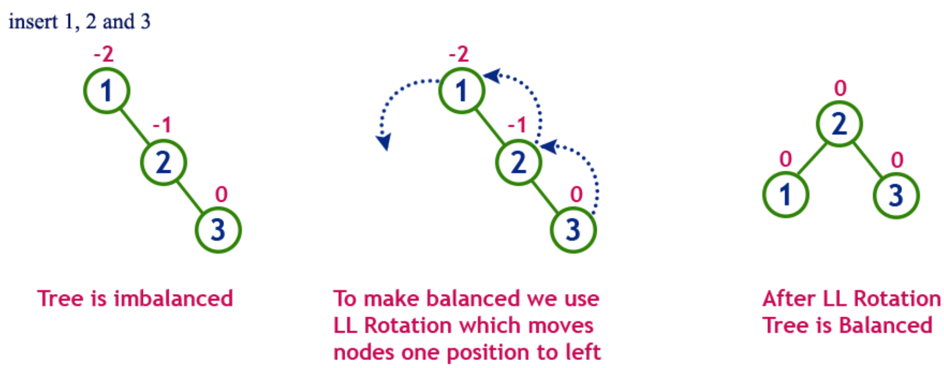
Rotation is the process of moving the nodes to either left or right to make tree balanced.

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



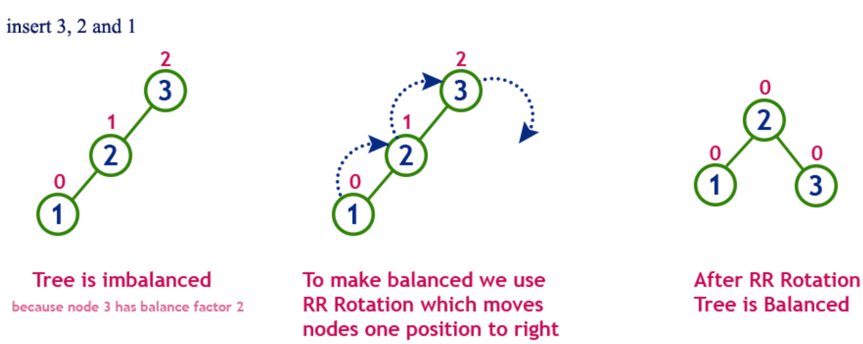
Single Left Rotation (LL Rotation)

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



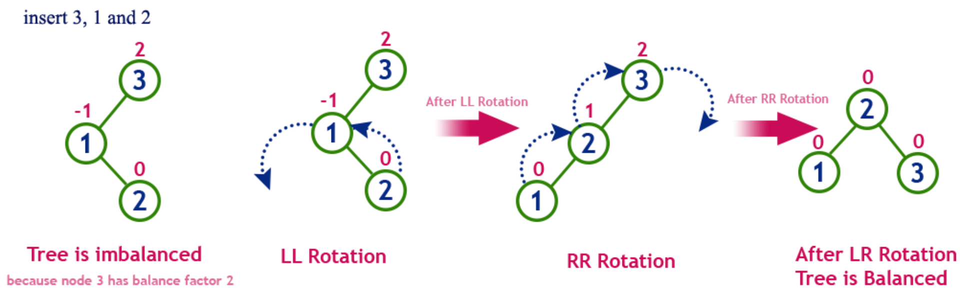
Single Right Rotation (RR Rotation)

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



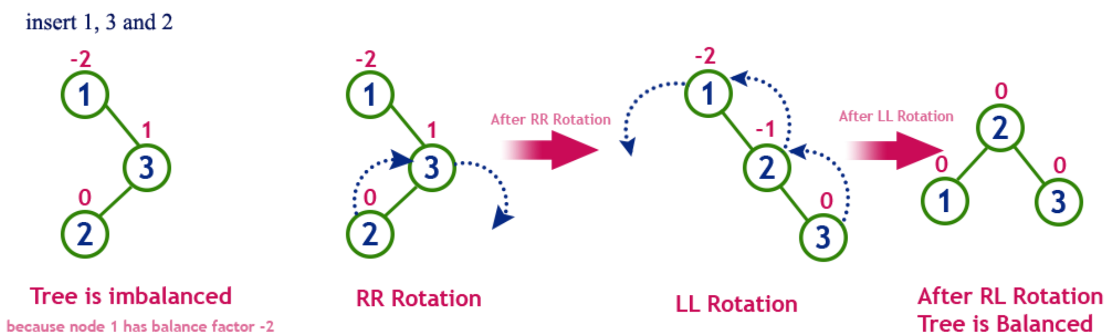
Left Right Rotation (LR Rotation)

The LR Rotation is combination of single left rotation followed by single right rotation. In LR Roration, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Roration, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree...



The following operations are performed on an 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 is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

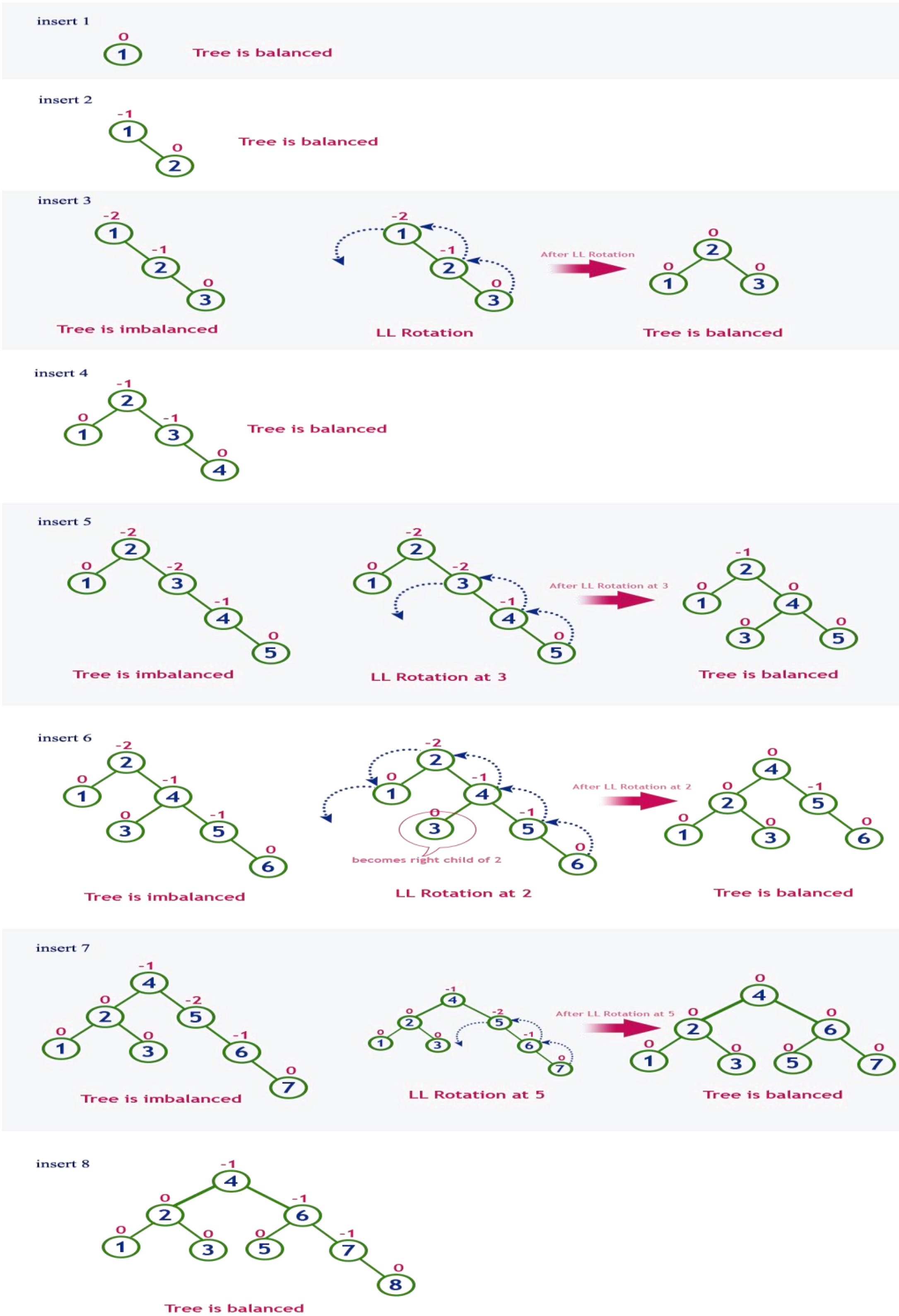
- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element 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 found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element 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, 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 tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

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



Deletion Operation in AVL Tree

In an AVL Tree, the deletion operation 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 then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

B - Trees

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree. B-Tree can be defined as follows...

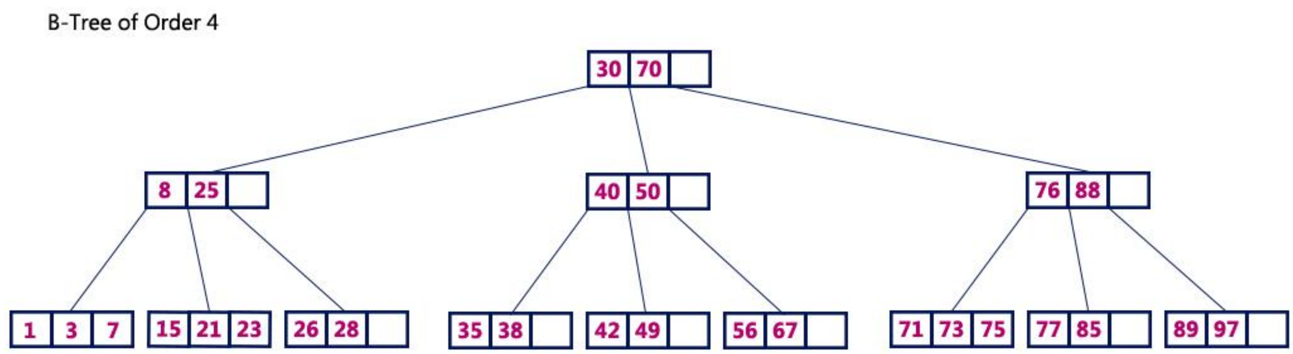
B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order. **B-Tree of Order m** has the following properties...

- Property #1 - All the **leaf nodes** must be **at same level**.
- Property #2 - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2**children.
- Property #4 - If the root node is a non leaf node, then it must have **at least 2** children.
- Property #5 - A non leaf node with **n-1** keys must have **n** number of children.
- Property #6 - All the **key values within a node** must be in **Ascending Order**.

For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.

Example



The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, 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 first key value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that key value.

- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match or comparision completed with last key value in a leaf node.
- **Step 7:** If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows...

- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is **Empty**, then create a new node with new key value and insert into the tree as a root node.
- **Step 3:** If tree is **Not Empty**, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
- **Step 4:** If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
- **Step 5:** If that leaf node is already full, then **split** that leaf node by sending middle value to its parent node. Repeat tha same until sending value is fixed into a node.
- **Step 6:** If the spilting is occuring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

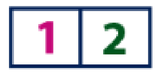
Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

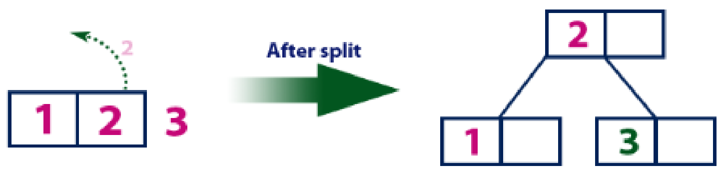
insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



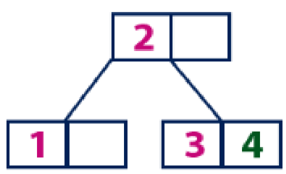
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



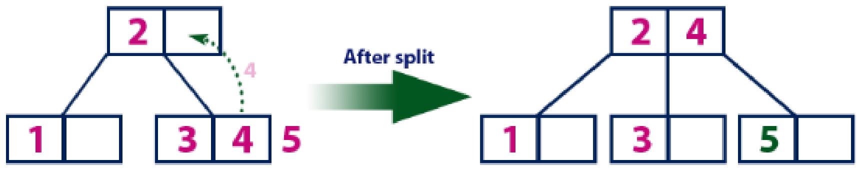
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



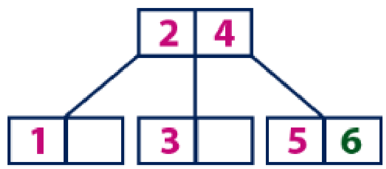
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



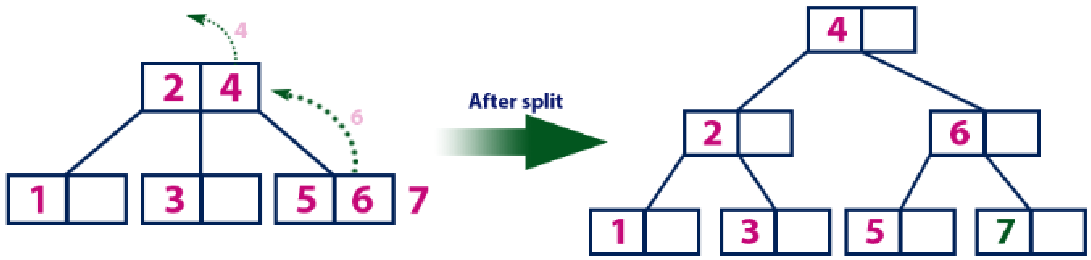
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



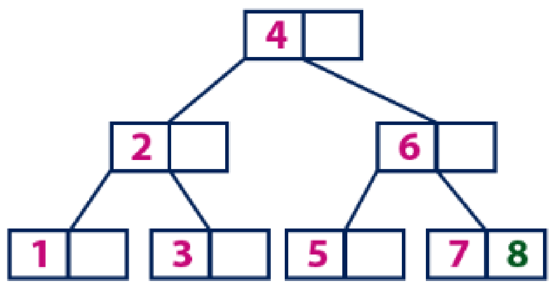
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



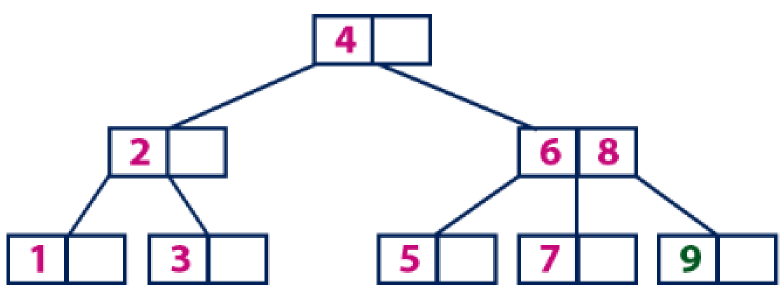
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.