**NON LINEAR DATA STRUCTURES**

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 non-linear data structure used in wide range of applications. A tree data structure can be defined as follows...

**Trees Basic Concepts:**

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a sub-tree of the root.

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

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

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

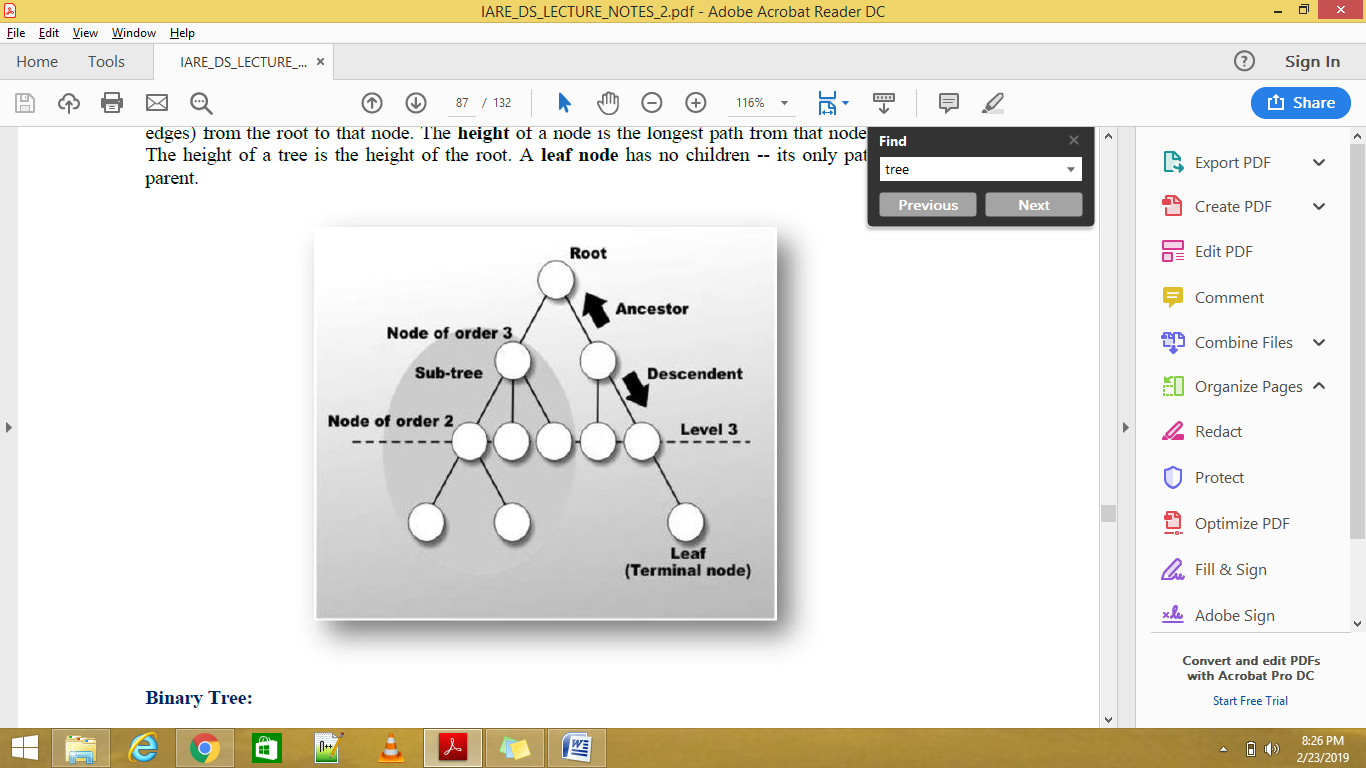
• If T is not empty, T has a special tree called the **root** that has no parent.

• Each node v of T different than the root has a unique parent node w; each node with parent w is a child of w.

Tree nodes have many useful properties.

* The **depth** of a node is the length of the path (or the number of edges) from the root to that node.
* The **height** of a node is the longest path from that node to its leafs. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.
* In a tree data structure, if we have **N** number of nodes then we can have a

maximum of **N-1** number of links.

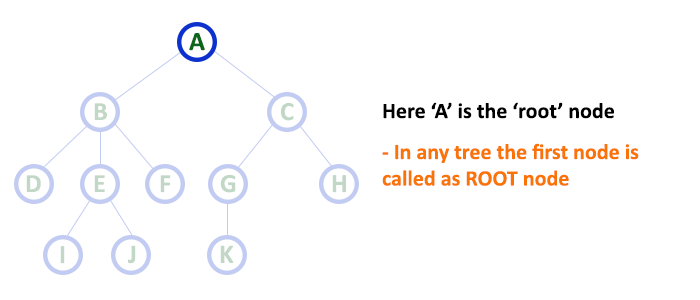


**Terminology**

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

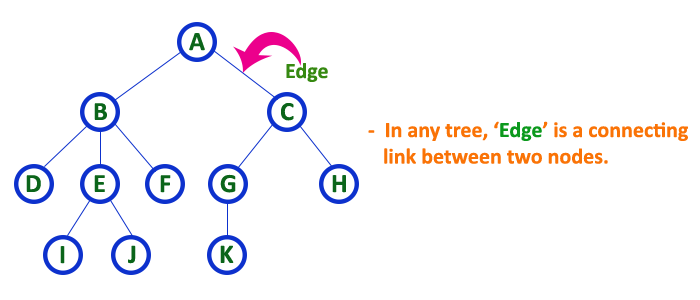
**1. Root**

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.



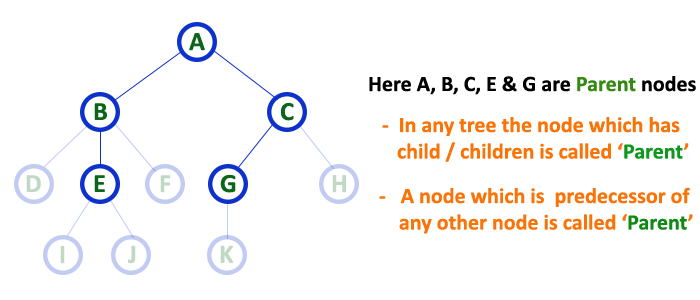
**2. Edge**

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.



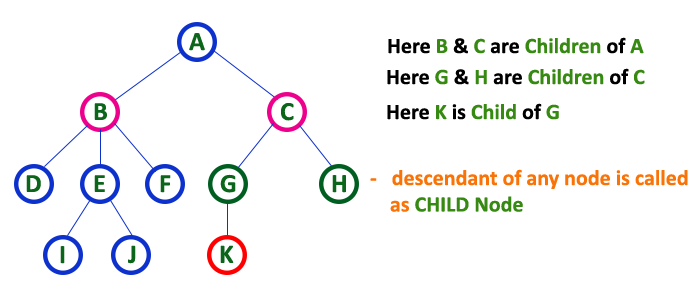
**3. Parent**

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**".



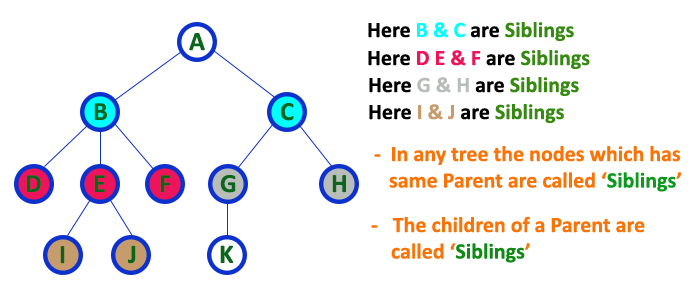
**4. Child**

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**

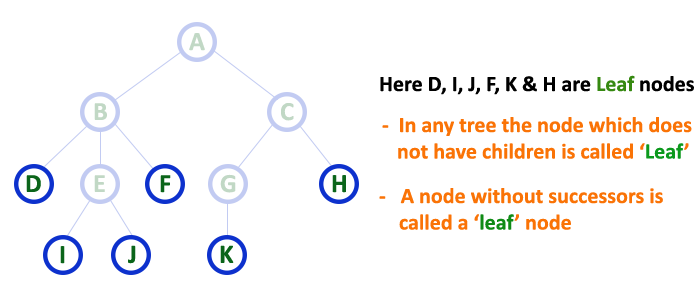
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



**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.

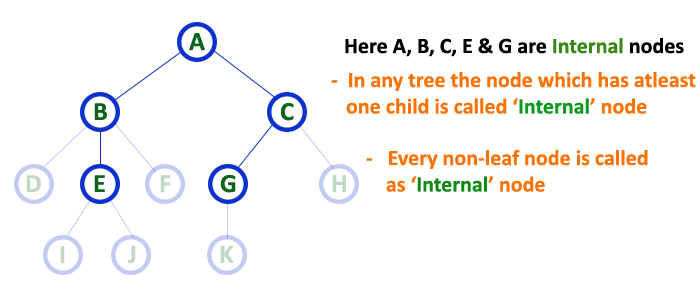
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



**7. Internal Nodes**

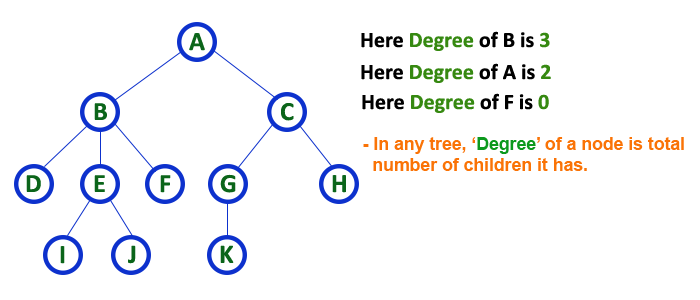
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

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.



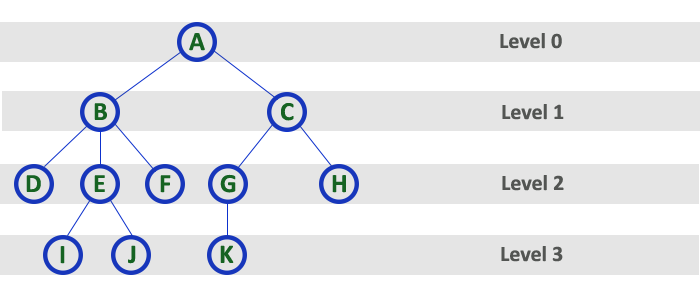
**8. Degree**

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**'



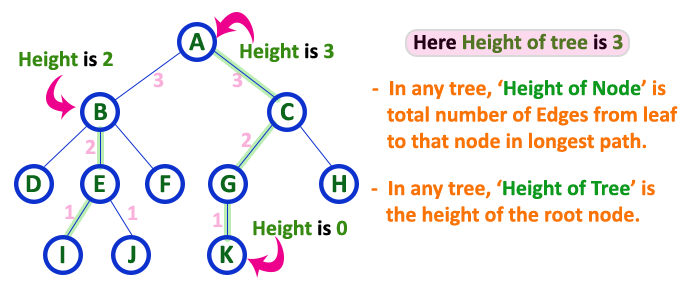
**9. Level**

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).



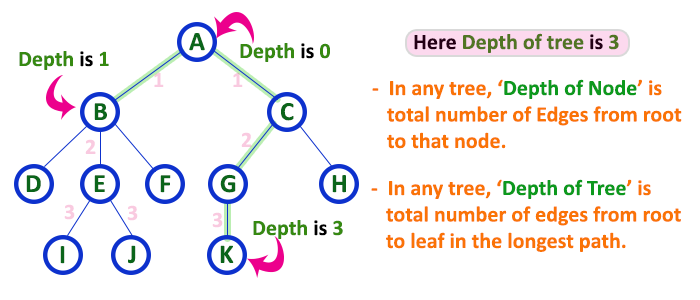
**10. Height**

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



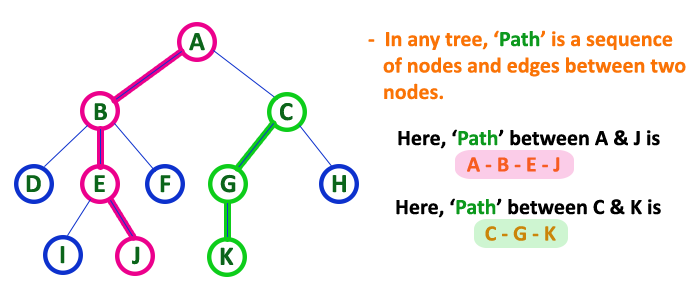
**11. Depth**

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



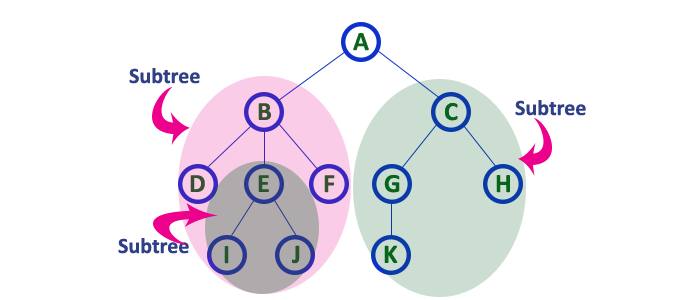
**12. Path**

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**.



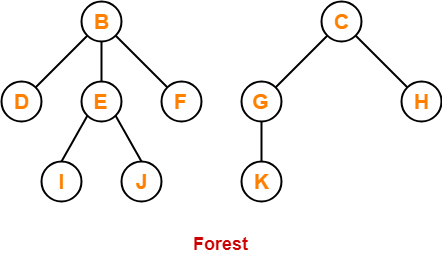
**13. Sub Tree**

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.



1. **Forest-**

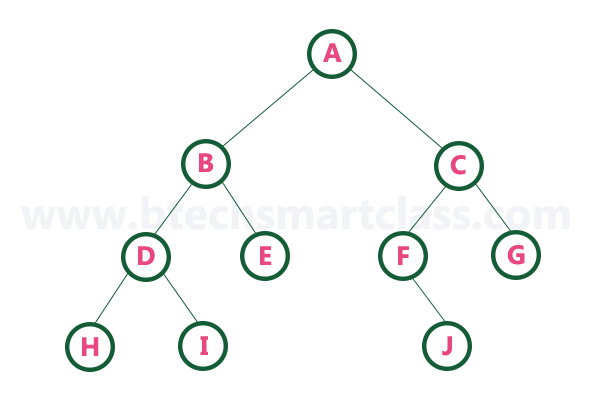
**A forest is a set of disjoint trees**

****

**Ancestor and Descendent** If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A*.*

**Binary Tree Data structure**

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



***The maximum number of nodes at any level is 2n.***

**Properties of Binary Trees:**

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then

a. Maximum number of leafs = 2h

b. Maximum number of nodes = 2h + 1 - 1

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

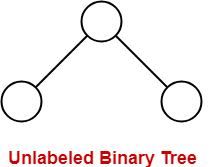
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2l node at level l.

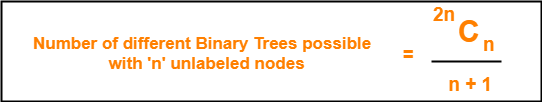
4. The **total number of edges** in a full binary tree **with n node is n - 1.**

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

## Unlabeled Binary Tree-

A binary tree is unlabeled if its nodes are not assigned any label.





## ****Example-****

Consider we want to draw all the binary trees possible with 3 unlabeled nodes.

Using the above formula, we have-

Number of binary trees possible with 3 unlabeled nodes

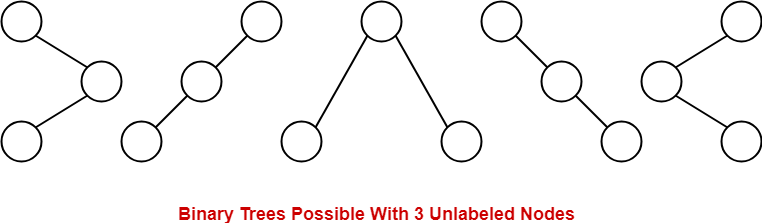
= 2 x 3C3 / (3 + 1)

= 6C3 / 4

= 5

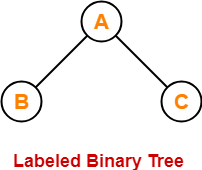
Thus,

* With 3 unlabeled nodes, 5 unlabeled binary trees are possible.
* These unlabeled binary trees are as follows-



## Labeled Binary Tree-

A binary tree is labeled if all its nodes are assigned a label.





## ****Example-****

Consider we want to draw all the binary trees possible with 3 labeled nodes.

Using the above formula, we have-

 Number of binary trees possible with 3 labeled nodes

= { 2 x 3C3 / (3 + 1) } x 3!

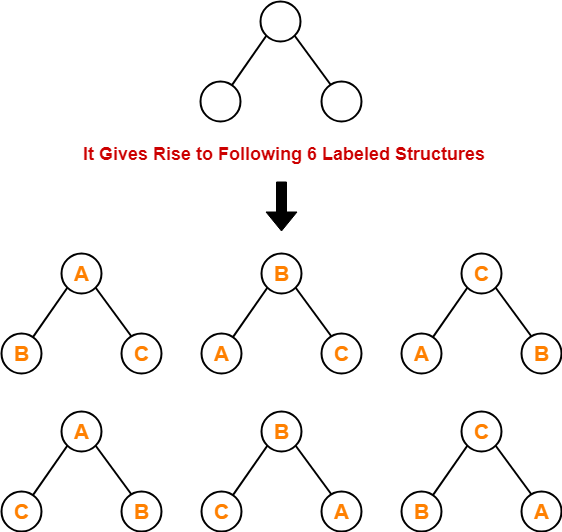
= { 6C3 / 4 } x 6

= 5 x 6

= 30

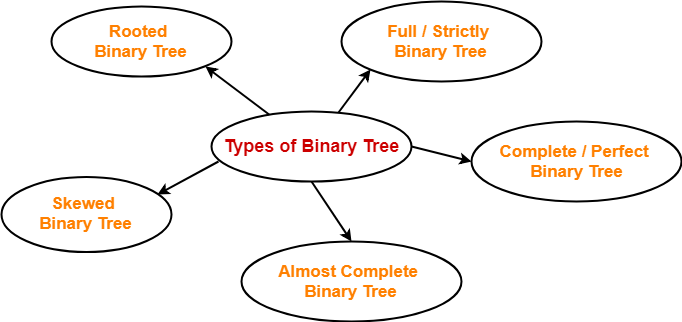
 Thus,

* With 3 labeled nodes, 30 labeled binary trees are possible.
* Each unlabeled structure gives rise to 3! = 6 different labeled structures.



## ****Types of Binary Trees-****

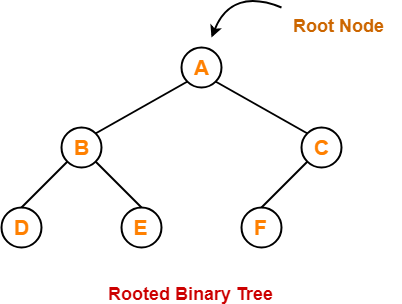
 Binary trees can be of the following types-



## ****1. Rooted Binary Tree-****

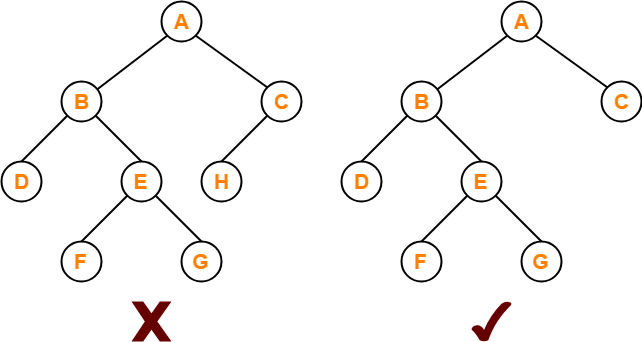
A **rooted binary tree** is a binary tree that satisfies the following 2 properties-

* It has a root node.
* Each node has at most 2 children.



## ****2. Full / Strictly Binary Tree-****

* A binary tree in which every node has ***either 0 or 2 children*** is called as a **Full binary tree**.
* Full binary tree is also called as **Strictly binary tree** or **Proper Binary Tree** or **2-Tree**



Here,

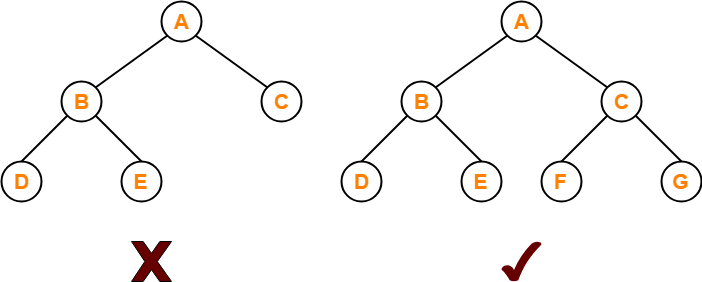
* First binary tree is not a full binary tree.
* This is because node C has only 1 child.

## ****3. Complete / Perfect Binary Tree-****

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

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

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



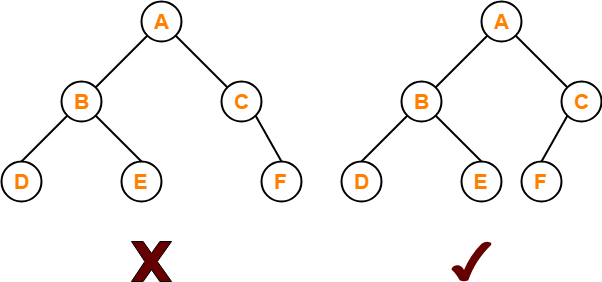
Here,

* First binary tree is not a complete binary tree.
* This is because all the leaf nodes are not at the same level.

## ****4. Almost Complete Binary Tree-****

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

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



Here,

* First binary tree is not an almost complete binary tree.
* This is because the last level is not filled from left to right.

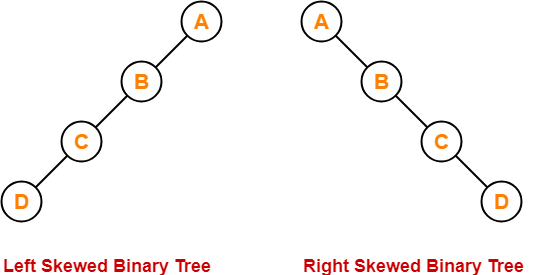
## ****5. Skewed Binary Tree-****

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

* All the nodes except one node has one and only one child.
* The remaining node has no child.

**OR**

A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).



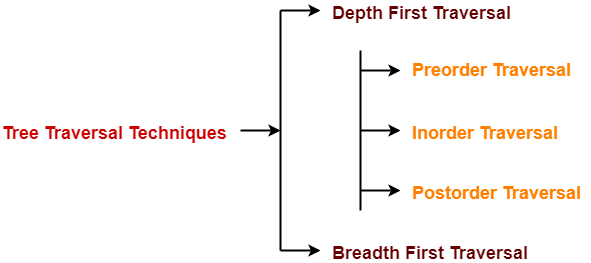
**Binary Tree Traversals**

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.

Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.

**Various tree traversal techniques are-**



## ****Depth First Traversal-****

Following three traversal techniques fall under Depth First Traversal-

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

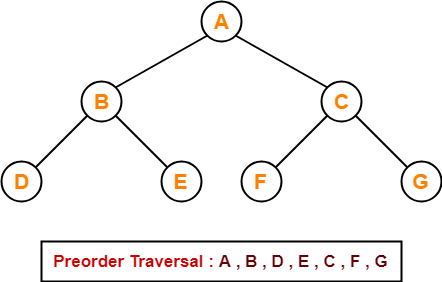
## ****1. Preorder Traversal-****

**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)

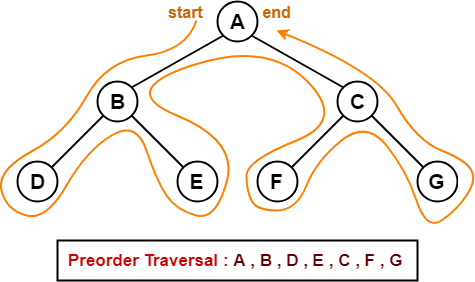
**Root**→**Left**→**Right**

Consider the following example-



## ****Preorder Traversal Shortcut****

Traverse the entire tree starting from the root node keeping yourself to the left.

.

## ****Applications-****

* Preorder traversal is used to get prefix expression of an expression tree.
* Preorder traversal is used to create a copy of the tree.

## ****2. Inorder Traversal-****

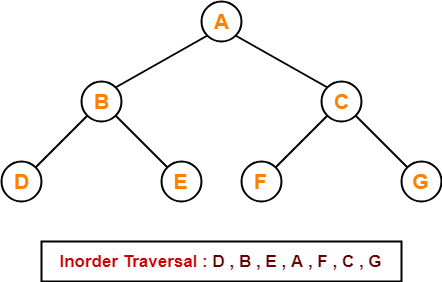
**Algorithm-**

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**

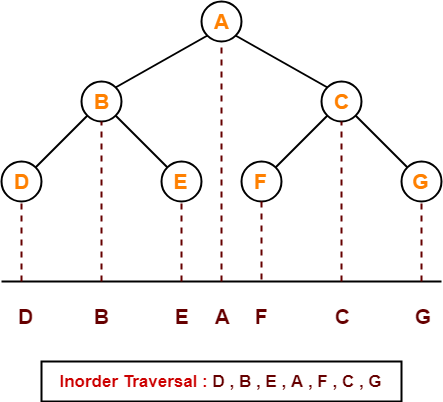
## ****Example-****

 Consider the following example-



## ****Inorder Traversal Shortcut****

Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



## ****Application-****

* Inorder traversal is used to get infix expression of an expression tree.

## ****3. Postorder Traversal-****

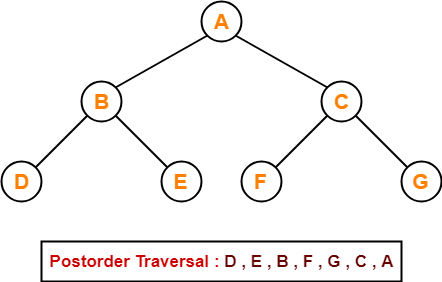
## ****Algorithm****

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**

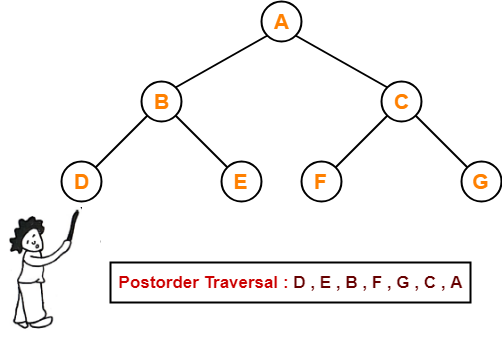
**Example-**

 Consider the following example-



## ****Postorder Traversal Shortcut****

  Pluck all the leftmost leaf nodes one by one.



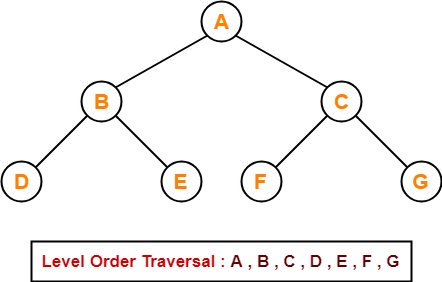
## ****Applications-****

* Postorder traversal is used to get postfix expression of an expression tree.
* Postorder traversal is used to delete the tree.
* 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**.

**Example-**



## ****Application-****

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

Consider the following binary tree...



**In-Order Traversal for above example of binary tree is** 

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

**Pre-Order Traversal for above example binary tree is** 

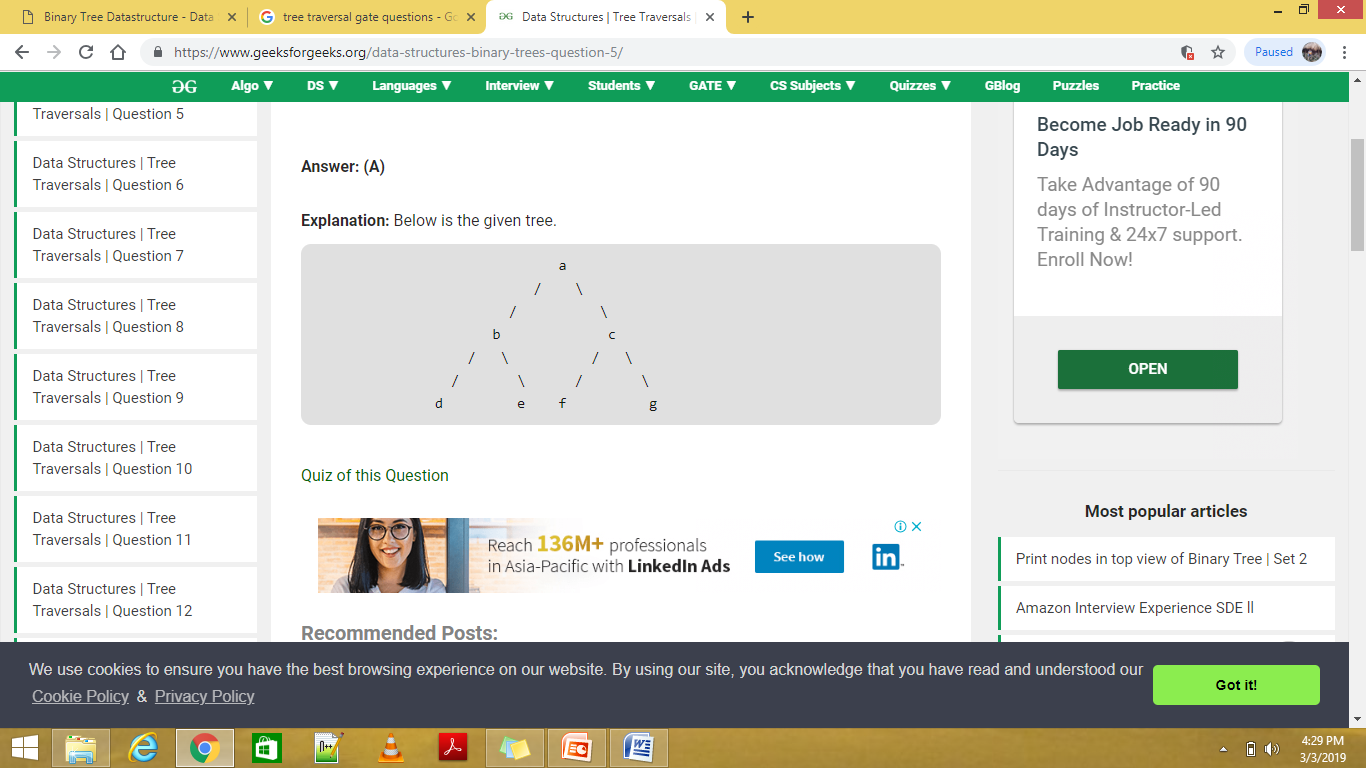
#### A - B - D - I - J - F - C - G - K - H

**Post-Order Traversal for above example binary tree is** 

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

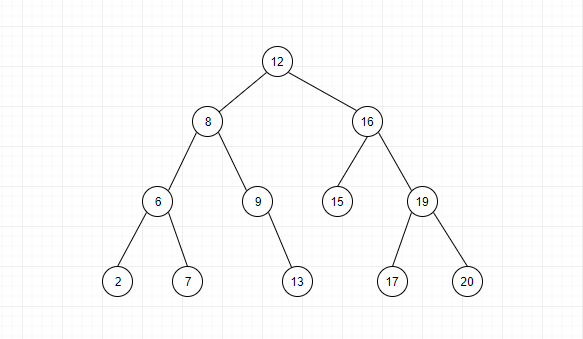
The inorder and preorder traversal of a binary tree are d b e a f c g and a b d e c f g, respectively. The postorder traversal of the binary tree is:

**(A)** d e b f g c a

****

The post-order traversal of a binary search tree is given by 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12. Then the pre-order traversal of this tree is:

**Explanation:** Since given tree is binary search tree, so inorder traversal will be always sorted order, i.e., 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20.  
Now we can draw that binary search tree using given postorder and inorder traversal. Final tree will be:



12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20

# Tree Representations

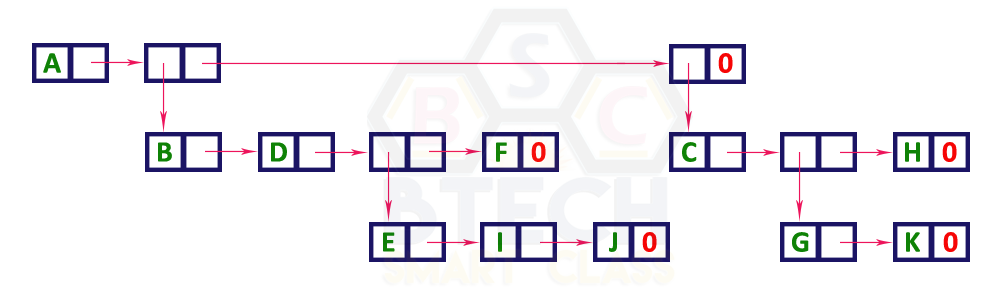
A tree data structure can be represented in two methods. Those methods are as follows...

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

**1. List Representation**

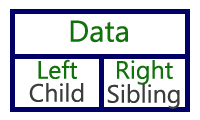
In this representation, we use two types of nodes one for representing the node with data called **'data node'** and another for representing only references called **'reference node'.** We start with a 'data node' from root node in the tree. Then it is linked to an internal node through a 'reference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

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

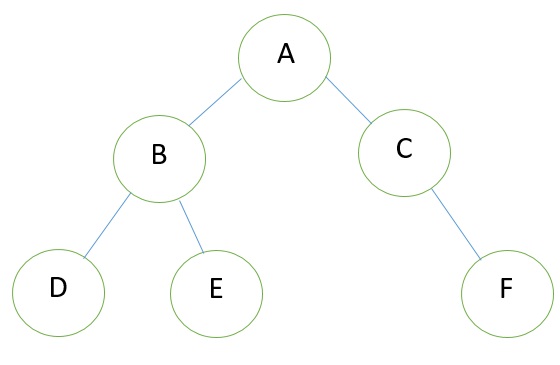


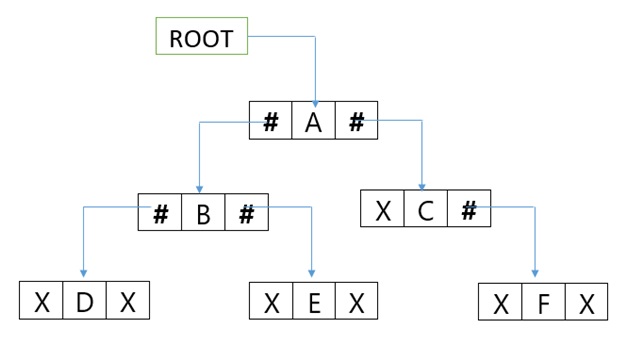
**2. Left Child - Right Sibling Representation**

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 stores NULL. If that node has right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.

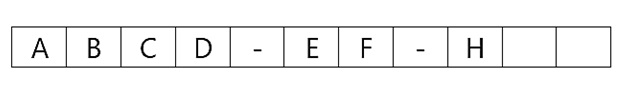




### 3. Sequential representation of Binary Tree

Let us consider that we have a tree **T**. let our tree **T** is a binary tree that us complete binary tree. Then there is an efficient way of representing **T** in the memory called the sequential representation or array representation of **T**. This representation uses only a linear array TREE as follows:

1. The root **N** of **T** is stored in **TREE [1]**.
2. If a node occupies **TREE [k]** then its left child is stored in **TREE [2 \* k]** and its right child is stored into **TREE [2 \* k + 1]**.



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 binary tree, the elements are arranged in the order they arrive to 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 special type of binary tree known as Binary Search Tree. Binary search tree mainly focuses 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.

**Operations on a Binary Search Tree**

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

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

**Search Operation in BST**

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

* **Step 1 -**Read the search element from the user.
* **Step 2 -**Compare the search element with the value of root node in the tree.
* **Step 3 -**If both are matched, then display "Given node is found!!!" and terminate the function
* **Step 4 -**If both are not matched, then check whether search element is smaller or larger than that node value.
* **Step 5 -**If search element is smaller, then continue the search process in left subtree.
* **Step 6-**If search element is larger, then continue the search process in right subtree.
* **Step 7 -**Repeat the same until we find the exact element or until the search element is compared with the leaf node
* **Step 8 -**If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
* **Step 9 -**If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

**Insertion Operation in BST**

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

* **Step 1 -**Create a newNode with given value and set its **left** and **right** to **NULL**.
* **Step 2 -**Check whether tree is Empty.
* **Step 3 -**If the tree is **Empty**, then set **root** to **newNode**.
* **Step 4 -**If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).
* **Step 5 -**If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.
* **Step 6-**Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).
* **Step 7 -**After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

**Deletion Operation in BST**

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

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

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

**Case 1: Deleting a leaf node**

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

* **Step 1 - Find** the node to be deleted using **search operation**
* **Step 2 -**Delete the node using **free** function (If it is a leaf) and terminate the function.

**Case 2: Deleting a node with one child**

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

* **Step 1 - Find** the node to be deleted using **search operation**
* **Step 2 -**If it has only one child then create a link between its parent node and child node.
* **Step 3 -**Delete the node using **free** function and terminate the function.

**Case 3: Deleting a node with two children**

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

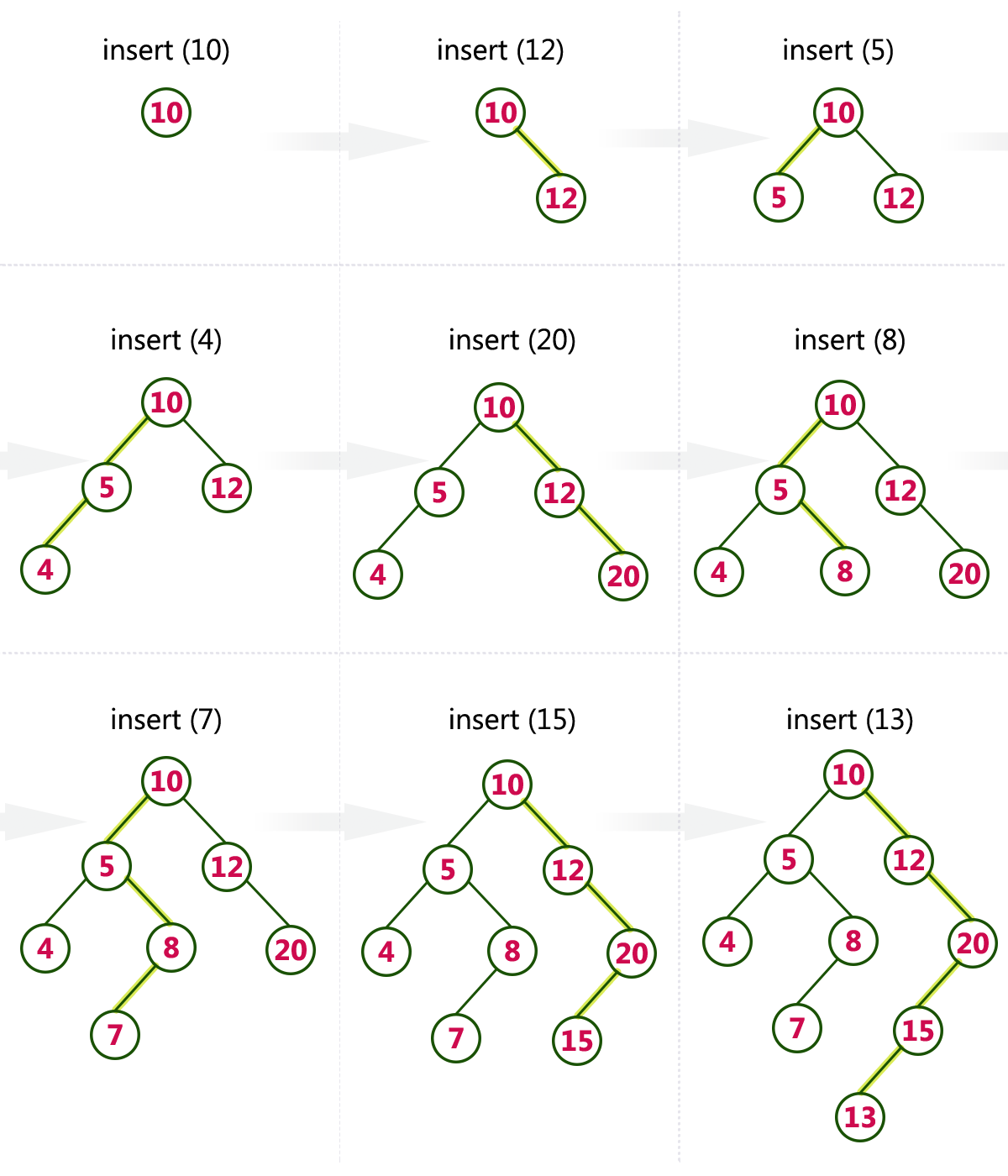
* **Step 1 - Find** the node to be deleted using **search operation**
* **Step 2 -**If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
* **Step 3 - Swap** both **deleting node** and node which is found in the above step.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

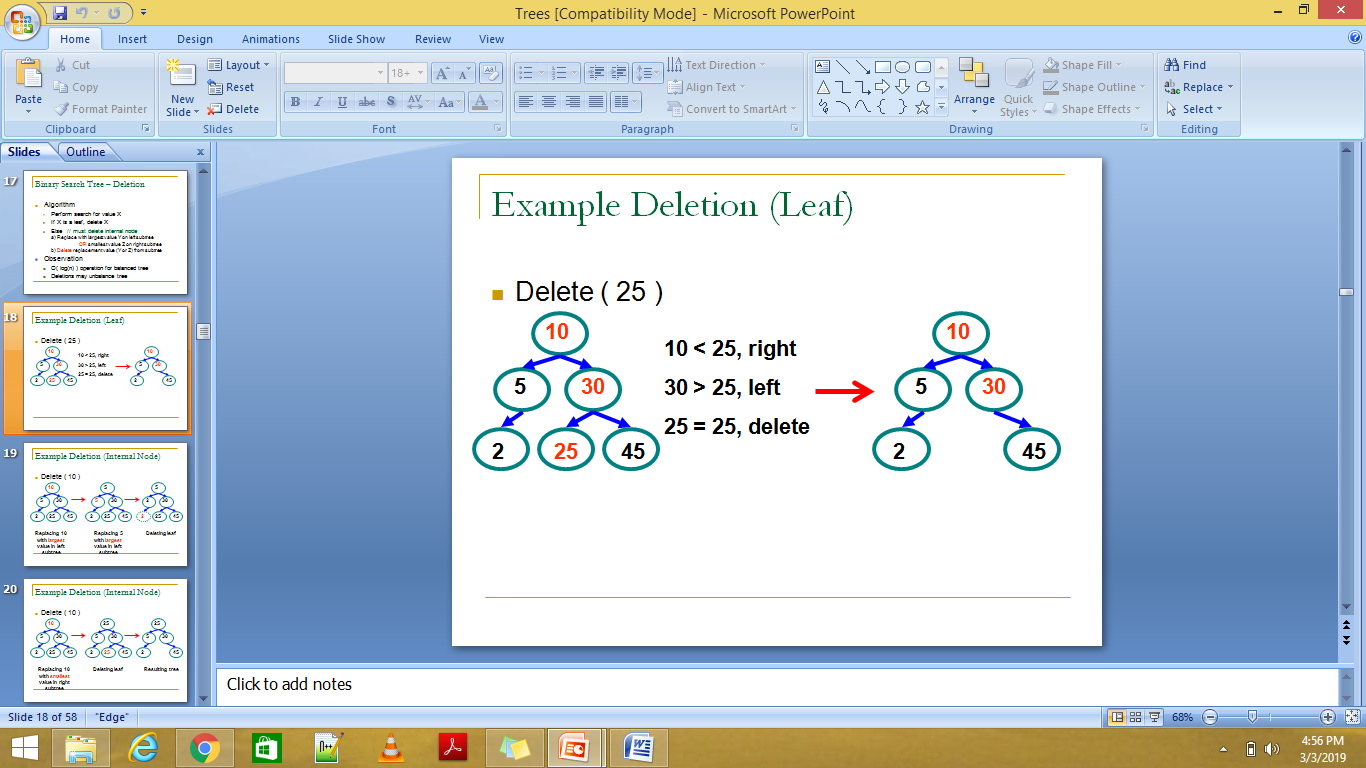
##### Example

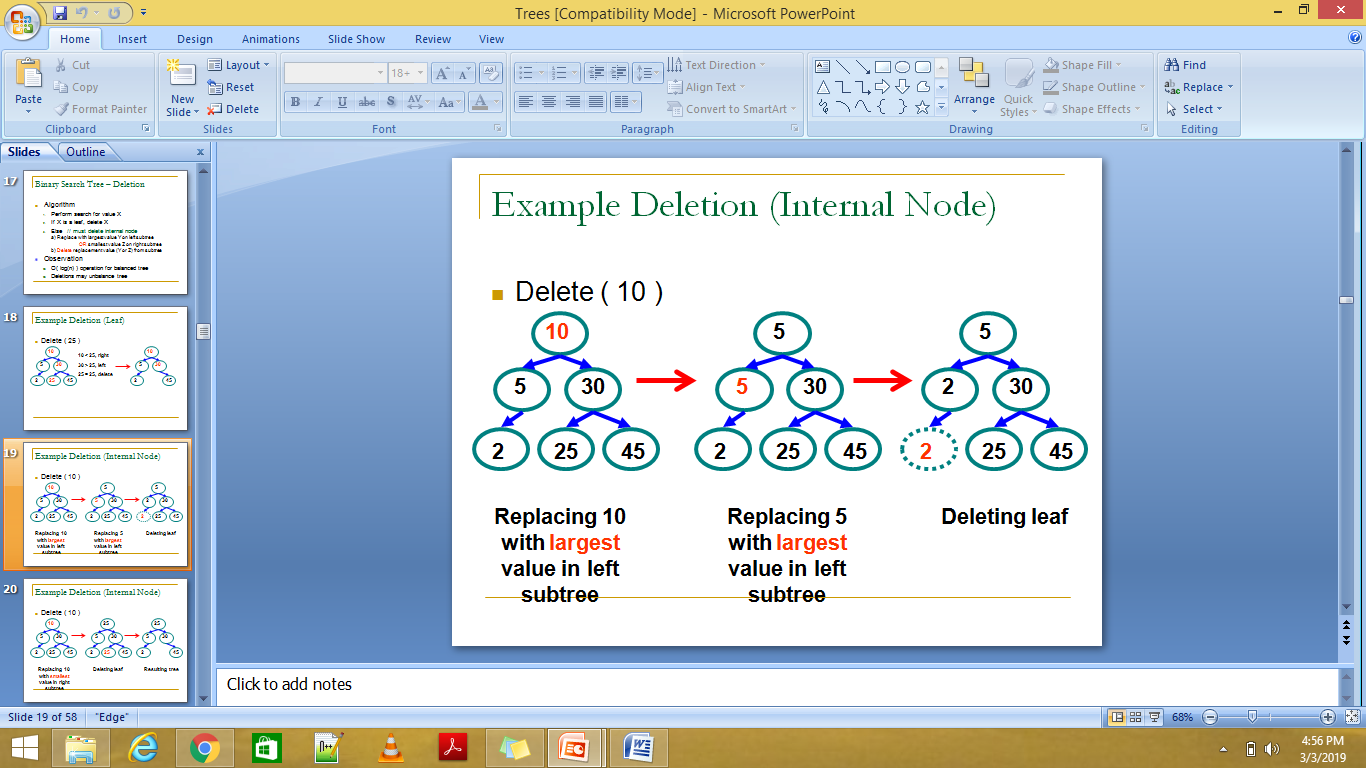
Construct a Binary Search Tree by inserting the following sequence of numbers...

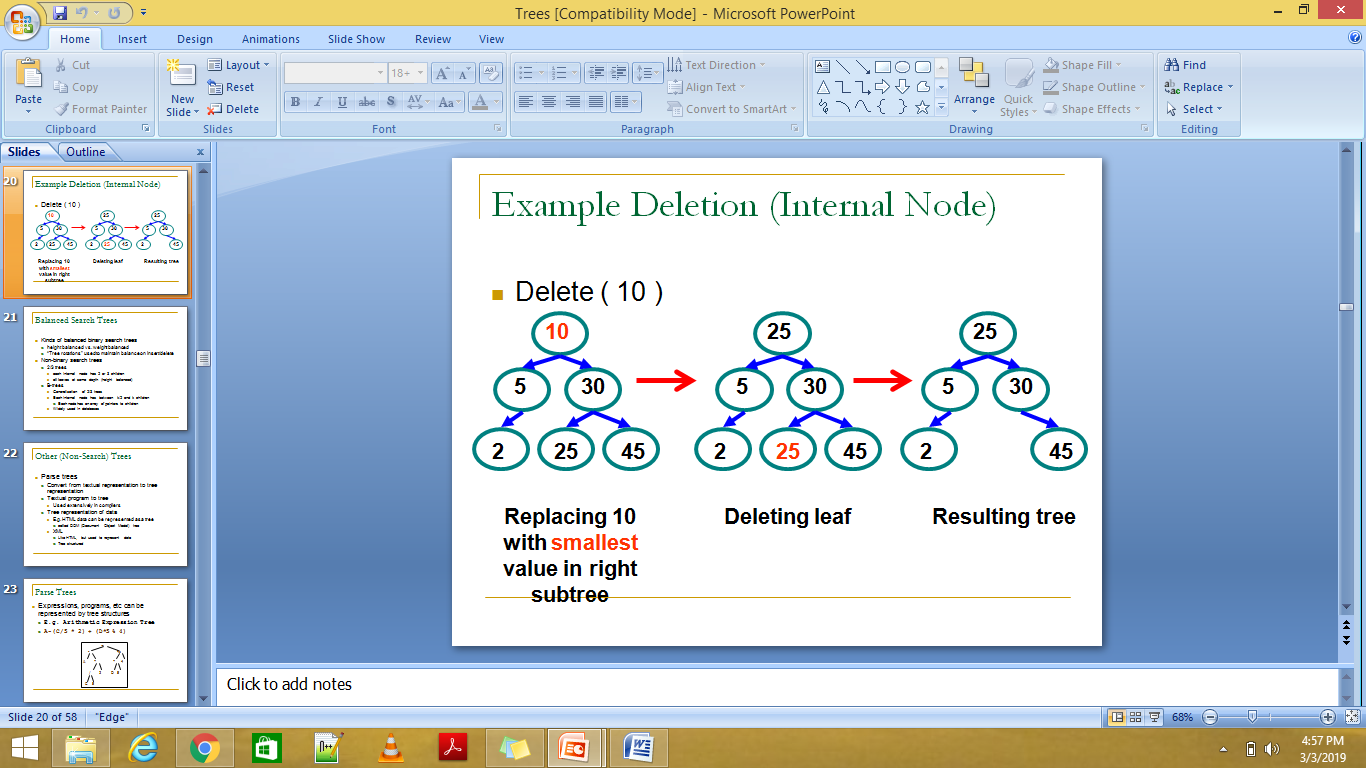
#### 10,12,5,4,20,8,7,15 and 13

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









# Implement binary search tree.

/\*

\* C Program to Construct a Binary Search Tree and perform deletion, inorder traversal on it

\*/

#include <stdio.h>

#include <stdlib.h>

struct btnode

{

int value;

struct btnode \*l;

struct btnode \*r;

}\*root = NULL, \*temp = NULL, \*t2, \*t1;

void delete1();

void insert();

void delete();

void inorder(struct btnode \*t);

void create();

void search(struct btnode \*t);

void preorder(struct btnode \*t);

void postorder(struct btnode \*t);

void search1(struct btnode \*t,int data);

int smallest(struct btnode \*t);

int largest(struct btnode \*t);

int flag = 1;

void main()

{

int ch;

printf("\nOPERATIONS ---");

printf("\n1 - Insert an element into tree\n");

printf("2 - Delete an element from the tree\n");

printf("3 - Inorder Traversal\n");

printf("4 - Preorder Traversal\n");

printf("5 - Postorder Traversal\n");

printf("6 - Exit\n");

while(1)

{

printf("\nEnter your choice : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

inorder(root);

break;

case 4:

preorder(root);

break;

case 5:

postorder(root);

break;

case 6:

exit(0);

default :

printf("Wrong choice, Please enter correct choice ");

break;

}

}

}

/\* To insert a node in the tree \*/

void insert()

{

create();

if (root == NULL)

root = temp;

else

search(root);

}

/\* To create a node \*/

void create()

{

int data;

printf("Enter data of node to be inserted : ");

scanf("%d", &data);

temp = (struct btnode \*)malloc(1\*sizeof(struct btnode));

temp->value = data;

temp->l = temp->r = NULL;

}

/\* Function to search the appropriate position to insert the new node \*/

void search(struct btnode \*t)

{

if ((temp->value > t->value) && (t->r != NULL)) /\* value more than root node value insert at right \*/

search(t->r);

else if ((temp->value > t->value) && (t->r == NULL))

t->r = temp;

else if ((temp->value < t->value) && (t->l != NULL)) /\* value less than root node value insert at left \*/

search(t->l);

else if ((temp->value < t->value) && (t->l == NULL))

t->l = temp;

}

/\* recursive function to perform inorder traversal of tree \*/

void inorder(struct btnode \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display");

return;

}

if (t->l != NULL)

inorder(t->l);

printf("%d -> ", t->value);

if (t->r != NULL)

inorder(t->r);

}

/\* To check for the deleted node \*/

void delete()

{

int data;

if (root == NULL)

{

printf("No elements in a tree to delete");

return;

}

printf("Enter the data to be deleted : ");

scanf("%d", &data);

t1 = root;

t2 = root;

search1(root, data);

}

/\* To find the preorder traversal \*/

void preorder(struct btnode \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display");

return;

}

printf("%d -> ", t->value);

if (t->l != NULL)

preorder(t->l);

if (t->r != NULL)

preorder(t->r);

}

/\* To find the postorder traversal \*/

void postorder(struct btnode \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display ");

return;

}

if (t->l != NULL)

postorder(t->l);

if (t->r != NULL)

postorder(t->r);

printf("%d -> ", t->value);

}

/\* Search for the appropriate position to insert the new node \*/

void search1(struct btnode \*t, int data)

{

if ((data>t->value))

{

t1 = t;

search1(t->r, data);

}

else if ((data < t->value))

{

t1 = t;

search1(t->l, data);

}

else if ((data==t->value))

{

delete1(t);

}

}

/\* To delete a node \*/

void delete1(struct btnode \*t)

{

int k;

/\* To delete leaf node \*/

if ((t->l == NULL) && (t->r == NULL))

{

if (t1->l == t)

{

t1->l = NULL;

}

else

{

t1->r = NULL;

}

t = NULL;

free(t);

return;

}

/\* To delete node having one left hand child \*/

else if ((t->r == NULL))

{

if (t1 == t)

{

root = t->l;

t1 = root;

}

else if (t1->l == t)

{

t1->l = t->l;

}

else

{

t1->r = t->l;

}

t = NULL;

free(t);

return;

}

/\* To delete node having right hand child \*/

else if (t->l == NULL)

{

if (t1 == t)

{

root = t->r;

t1 = root;

}

else if (t1->r == t)

t1->r = t->r;

else

t1->l = t->r;

t == NULL;

free(t);

return;

}

/\* To delete node having two child \*/

else if ((t->l != NULL) && (t->r != NULL))

{

t2 = root;

if (t->r != NULL)

{

k = smallest(t->r);

flag = 1;

}

else

{

k =largest(t->l);

flag = 2;

}

search1(root, k);

t->value = k;

}

}

/\* To find the smallest element in the right sub tree \*/

int smallest(struct btnode \*t)

{

t2 = t;

if (t->l != NULL)

{

t2 = t;

return(smallest(t->l));

}

else

return (t->value);

}

/\* To find the largest element in the left sub tree \*/

int largest(struct btnode \*t)

{

if (t->r != NULL)

{

t2 = t;

return(largest(t->r));

}

else

return(t->value);

}

**AVL Tree Data structure**

***AVL tree is a height balanced binary search tree***. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right 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 the height of left and right children of every node differ by 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.

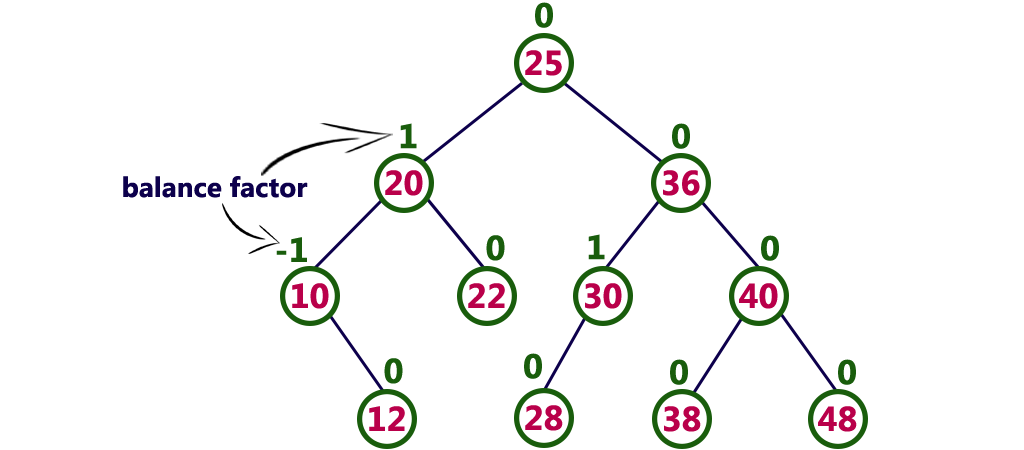
An AVL tree is defined as follows...

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

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

## Complexity

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Average case** | **Worst case** |
| Space | o(n) | o(n) |
| Search | o(log n) | o(log n) |
| Insert | o(log n) | o(log n) |
| Delete | o(log n) | o(log n) |

# 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.

Depending upon the type of insertion, the Rotations are categorized into four categories.

|  |  |  |
| --- | --- | --- |
| **SN** | **Rotation** | **Description** |
| 1 | [LL Rotation](https://www.javatpoint.com/ll-rotation-in-avl-tree) | The new node is inserted to the left sub-tree of left sub-tree of critical node. |
| 2 | [RR Rotation](https://www.javatpoint.com/rr-rotation-in-avl-tree) | The new node is inserted to the right sub-tree of the right sub-tree of the critical node. |
| 3 | [LR Rotation](https://www.javatpoint.com/lr-rotation-in-avl-tree) | The new node is inserted to the right sub-tree of the left sub-tree of the critical node. |
| 4 | [RL Rotation](https://www.javatpoint.com/rl-rotation-in-avl-tree) | The new node is inserted to the left sub-tree of the right sub-tree of the critical node. |

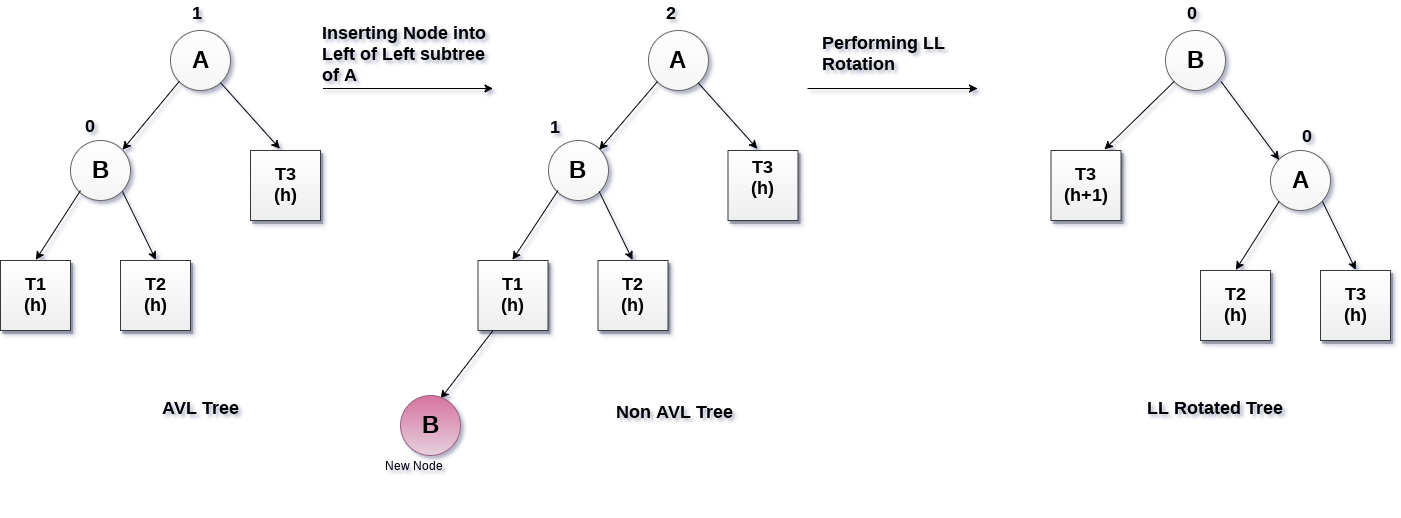
# Single Right Rotation (LL Rotation)

The tree shown in following figure is an AVL Tree, however, we, need to insert an element into the left of the left sub-tree of A. the tree can become unbalanced with the presence of the critical node A.

The node whose balance factor doesn't lie between -1 and 1, is called critical node.

In order to rebalance the tree, LL rotation is performed as shown in the following diagram.

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

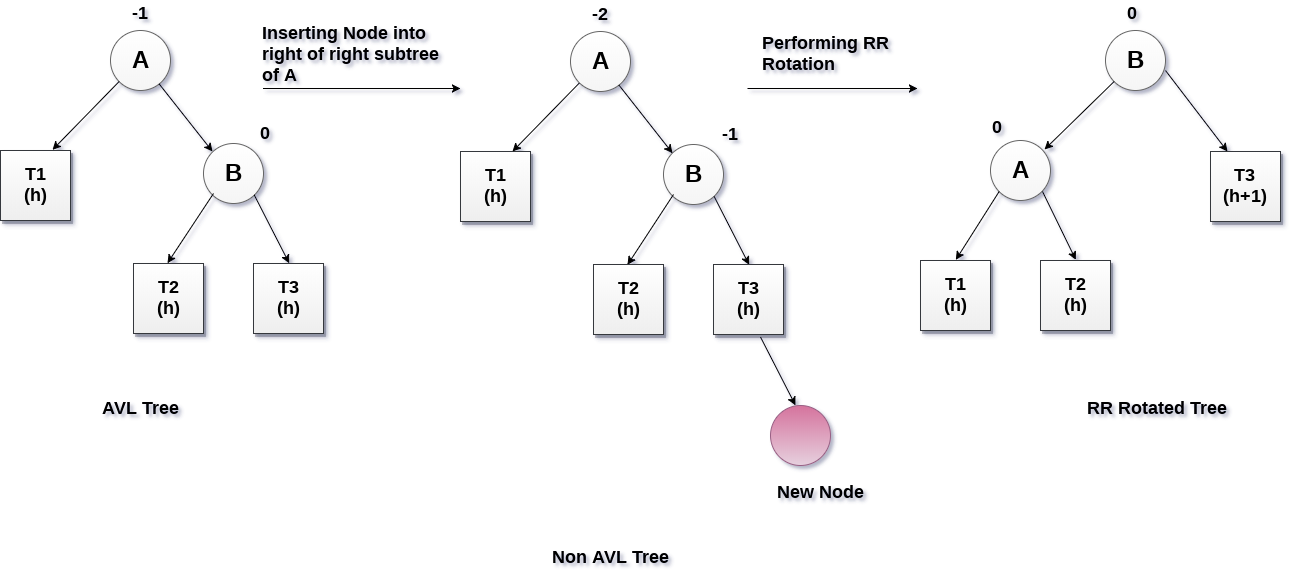


# Single Left Rotation (RR Rotation)

If the node is inserted into the right of the right sub-tree of a node A and the tree becomes unbalanced then, in that case, RR rotation will be performed as shown in the following diagram.

While the rotation, the node B becomes the root node of the tree. The critical node A will be moved to its left and becomes the left child of B.

The sub-tree T3 becomes the right sub-tree of A. T1 and T2 becomes the left and right sub-tree of node A.



# Left Right Rotation (LR Rotation)

LR rotation is to be performed if the new node is inserted into the right of the left sub-tree of node A.

In LR rotation, node C (as shown in the figure) becomes the root node of the tree, while the node B and A becomes its left and right child respectively.

T1 and T2 becomes the left and right sub-tree of Node B respectively whereas, T3 and T4 becomes the left and right sub-tree of Node A.

# LR Rotation in avl tree

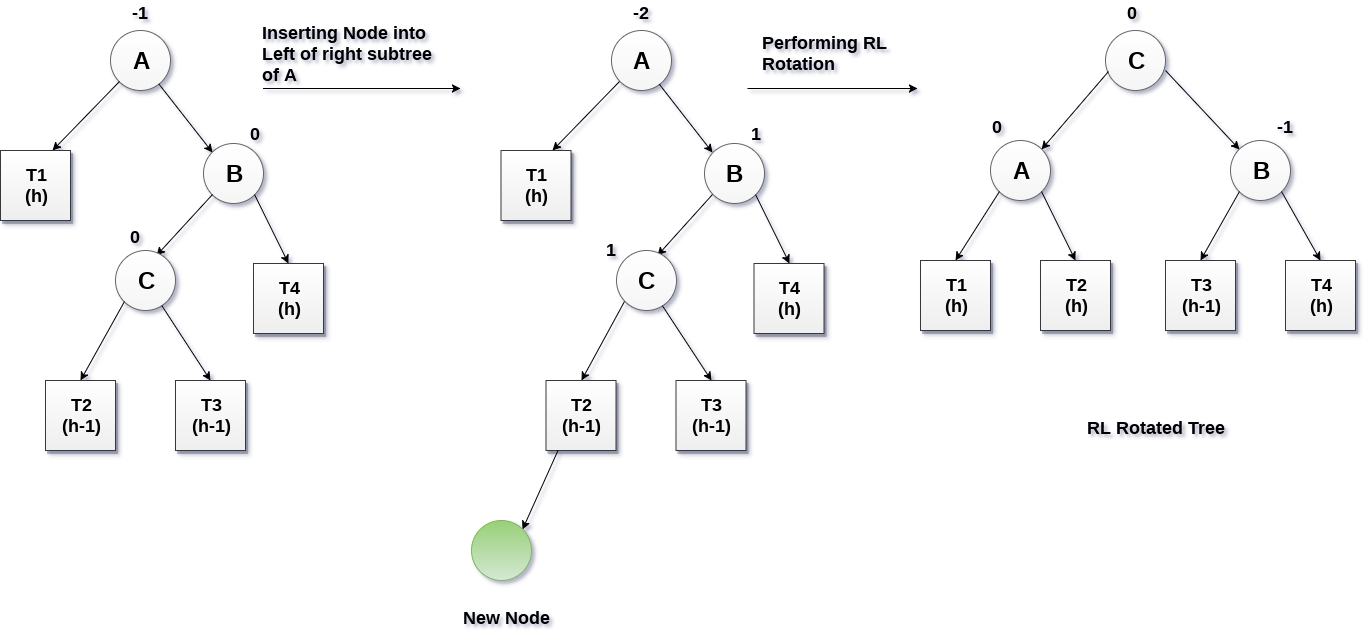
# Right Left Rotation (RL Rotation)

RL rotations is to be performed if the new node is inserted into the left of right sub-tree of the critical node A. Let us consider, Node B is the root of the right sub-tree of the critical node, Node C is the root of the sub-tree in which the new node is inserted.

Let T1 be the left sub-tree of the critical node A, T2 and T3 be the left and right sub-tree of Node C respectively, sub-tree T4 be the right sub-tree of Node B.

Since, RL rotation is the mirror image of LR rotation. In this rotation, the node C becomes the root node of the tree with A and B as its left and right children respectively. Sub-trees T1 and T2 becomes the left and right sub-trees of A whereas, T3 and T4 becomes the left and right sub-trees of B.

The process of RL rotation is shown in the following image.



# Operations on an AVL Tree

The following operations are performed on AVL tree...

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

# Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in AVL tree is similar to search operation in 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, 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 the following elements in the given order.

**63, 9, 19, 27, 18, 108, 99, 81**

All the elements are inserted in order to maintain the order of binary search tree.

# Insertion in avl tree

# 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.

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation. Here, we will discuss R rotations. L rotations are the mirror images of them.

If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

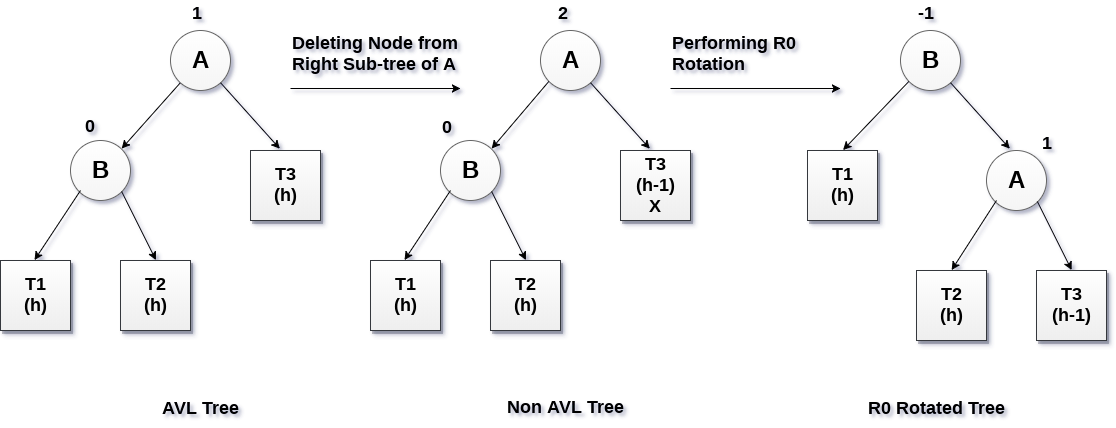
Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

## R0 rotation (Node B has balance factor 0 )

If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R0 rotation is shown in the following image.

LL rotation

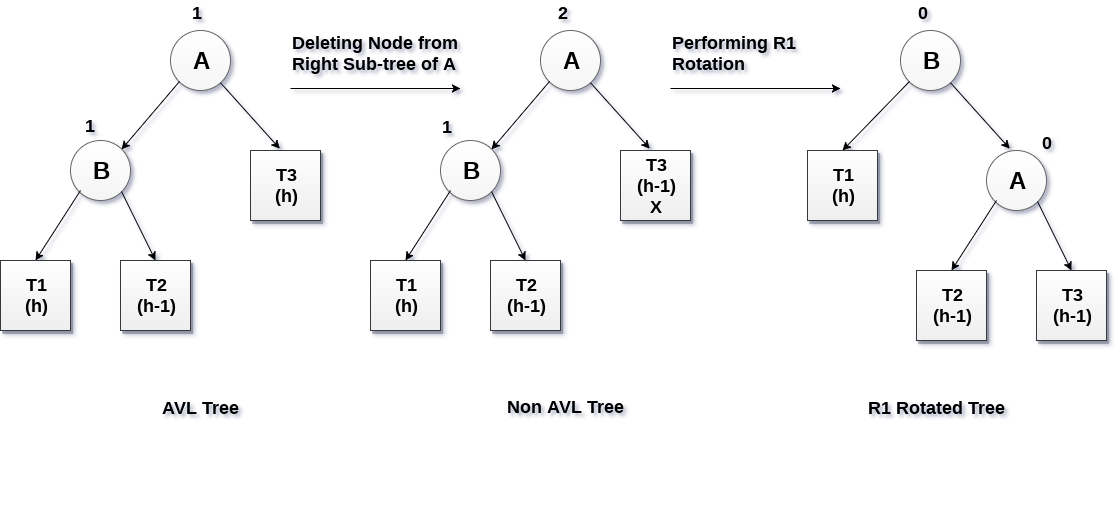


## R1 Rotation (Node B has balance factor 1)

R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

The process involved in R1 rotation is shown in the following image.

LL Rotation



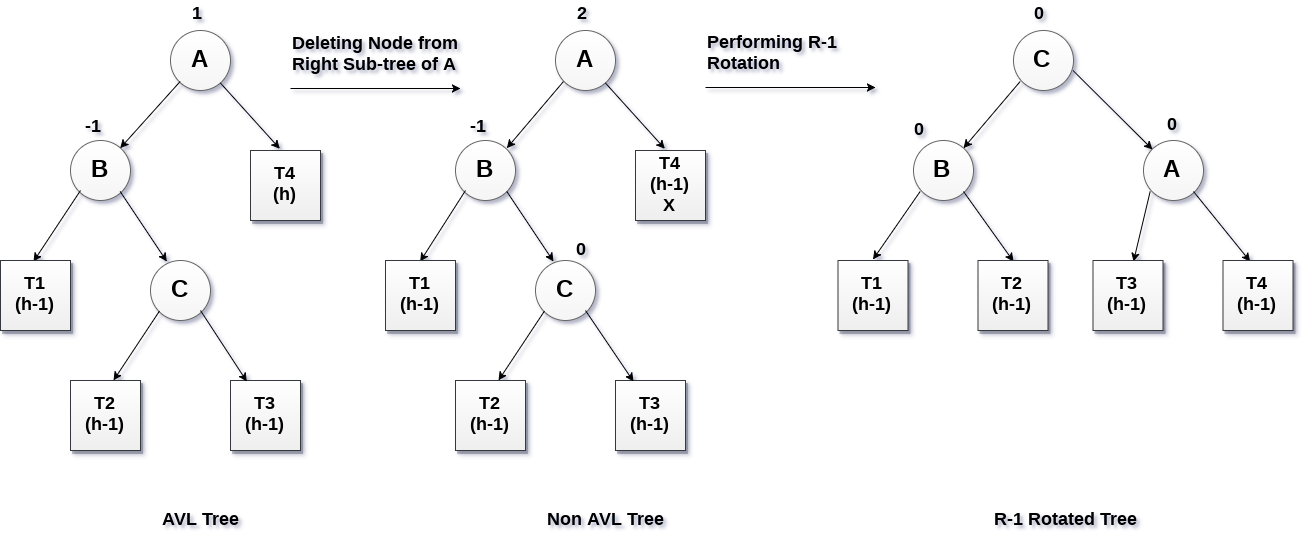
## R-1 Rotation (Node B has balance factor -1)

R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively.

The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.

The process involved in R-1 rotation is shown in the following image.

LR Rotation



**Similarly L0, L1,L-1**

# B - Tree Data structure

In search trees like binary search tree, AVL Tree, Red-Black tree etc., every node contains only one value (key) and maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 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 in which every node contains multiple keys and has more than two children.**

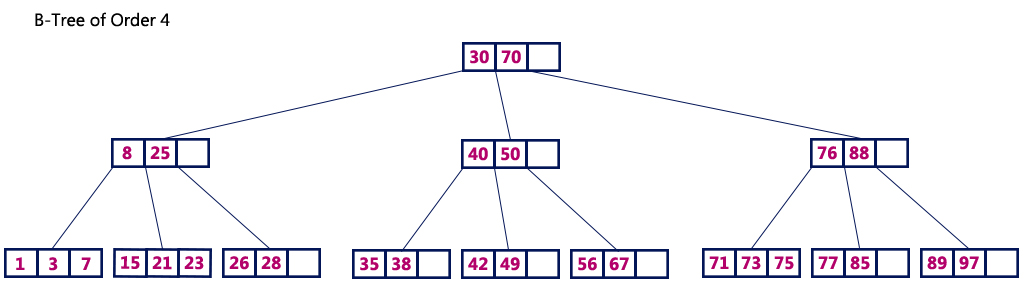
Here, number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

**B-Tree of Order m** has the following properties...

* **Property #1** - All **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 **atleast 2** children.
* **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
* **Property #6** - All the **key values in a node** must be in **Ascending Order**.

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

##### Example



# Operations on a B-Tree

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

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

# Search Operation in B-Tree

The search operation in B-Tree is similar to search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make n-way decision every time. Where 'n' is the total number of children the node has. In a B-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 first key 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 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 the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
* **Step 7 -**If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

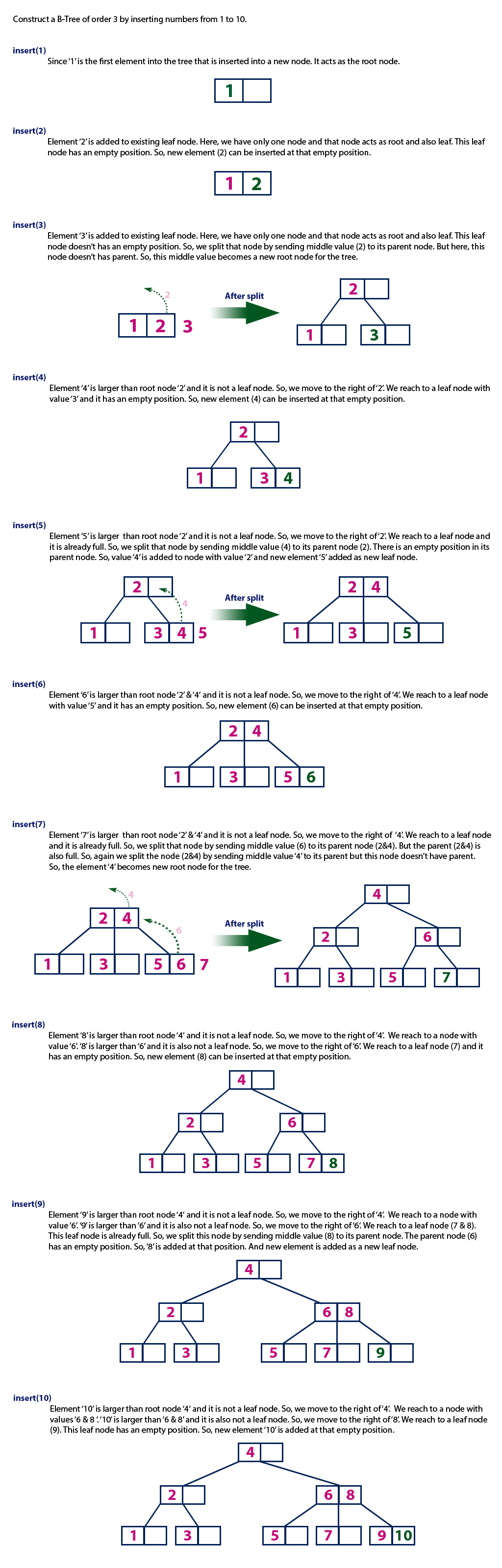
# Insertion Operation in B-Tree

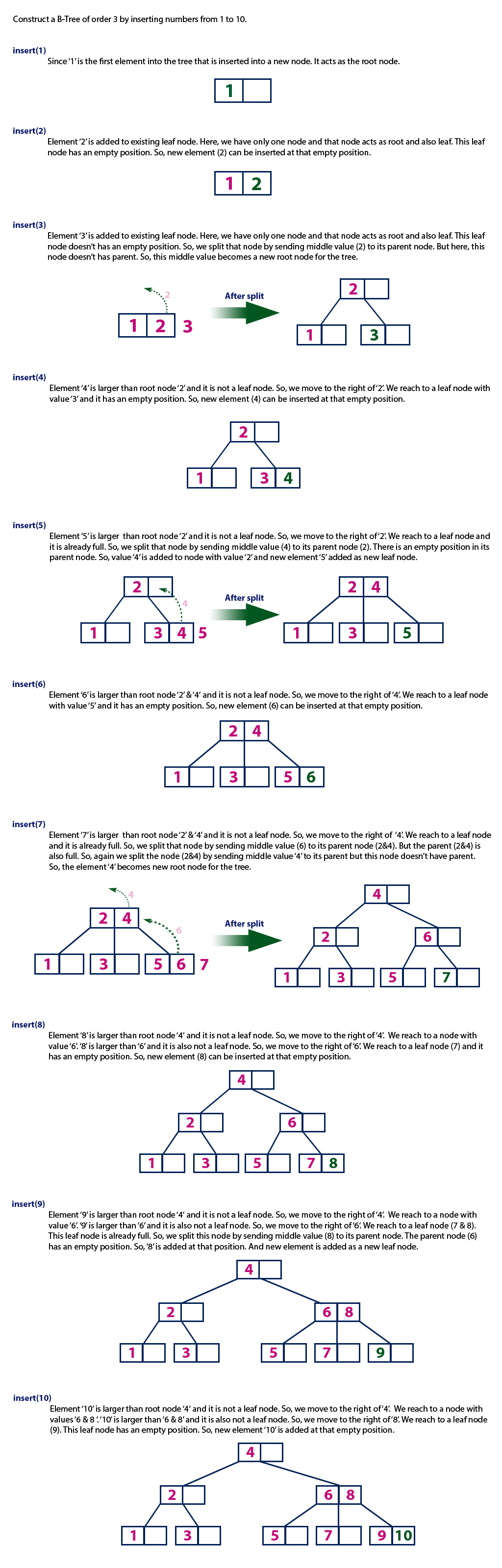
In a B-Tree, new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

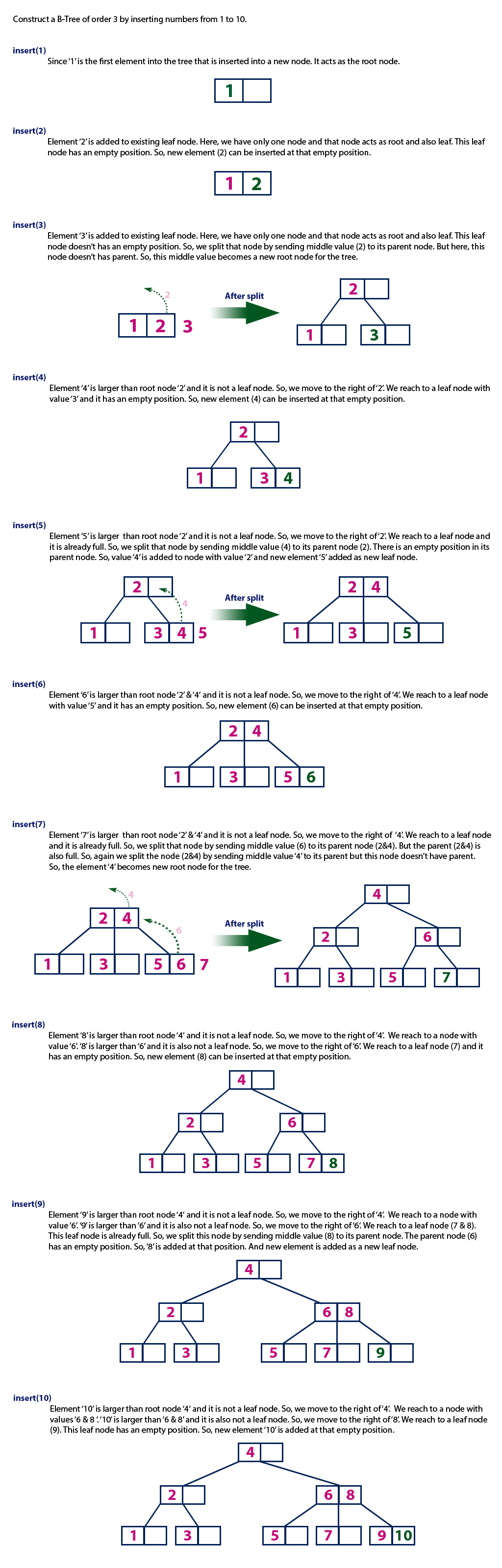
* **Step 1 -**Check whether tree is Empty.
* **Step 2 -**If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.
* **Step 3 -**If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
* **Step 4 -**If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
* **Step 5 -**If that leaf node is ***already full***, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
* **Step 6 -**If the splitting is performed at 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.







### Deletion

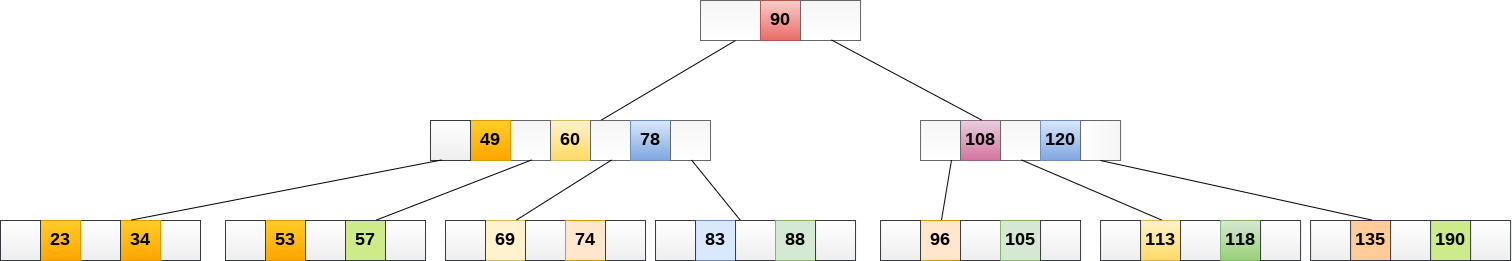
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
   * If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
   * If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

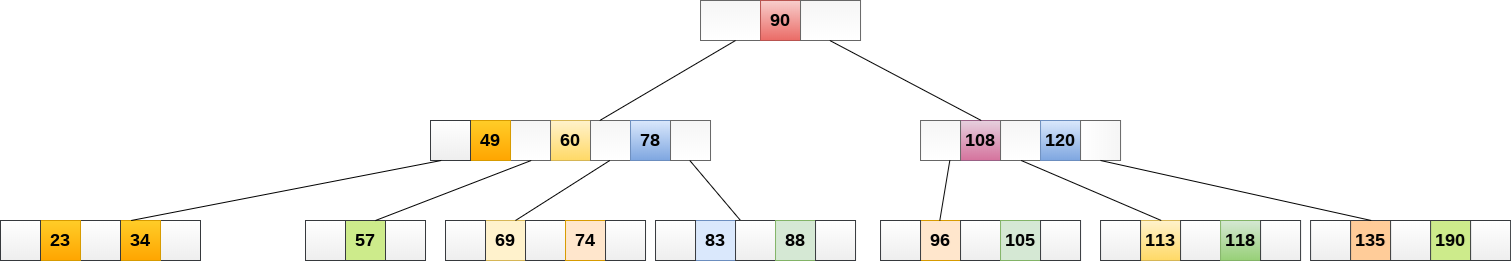
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

**Example 1**

Delete the node 53 from the B Tree of order 5 shown in the following figure.

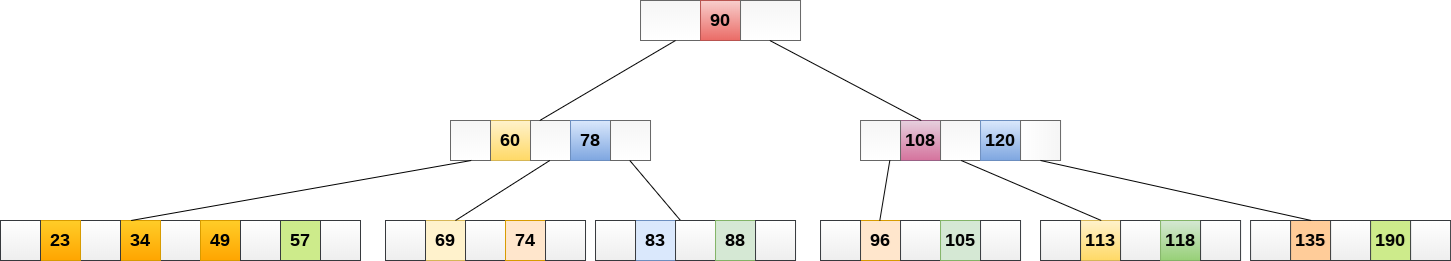


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



## Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

# Splay Tree Data structure

Splay tree is an another variant of binary search tree. In splay tree, **recently accessed element is placed at the root of the tree.** A splay tree is defined as follows...

**Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.**

In splay tree, every operation is performed at root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

**Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.**

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at root of the tree. The search operation in splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

In splay tree, to splay any element we use the following rotation operations...

# Rotations in Splay Tree

**1. Zig Rotation (Right Rotation LL)**

**2. Zag Rotation ( Left Rotation RR)**

**3. Zig - Zig Rotation**

**4. Zag - Zag Rotation**

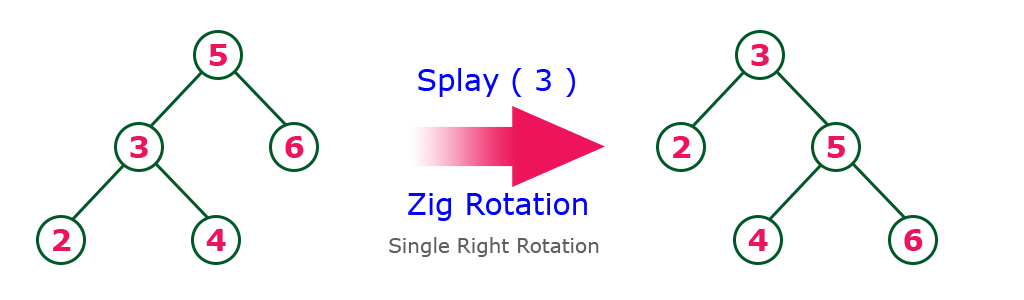
**5. Zig - Zag Rotation(Zig followed by Zag )**

**6. Zag - Zig Rotation(Zag followed by Zig )**

##### Example

# Zig Rotation

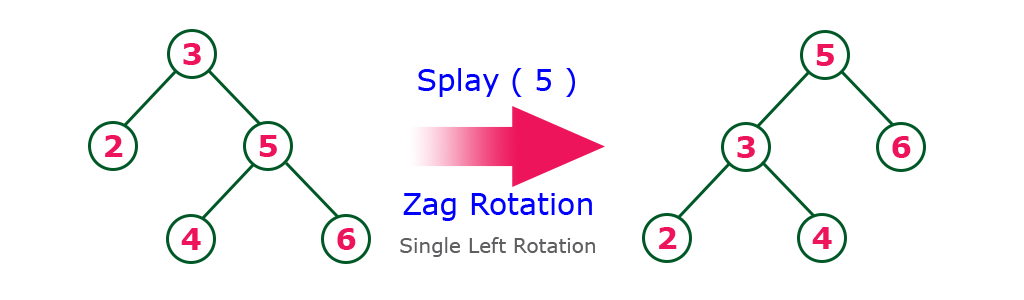
The **Zig Rotation** in splay tree is similar to the single right rotation (LL) in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



# Zag Rotation

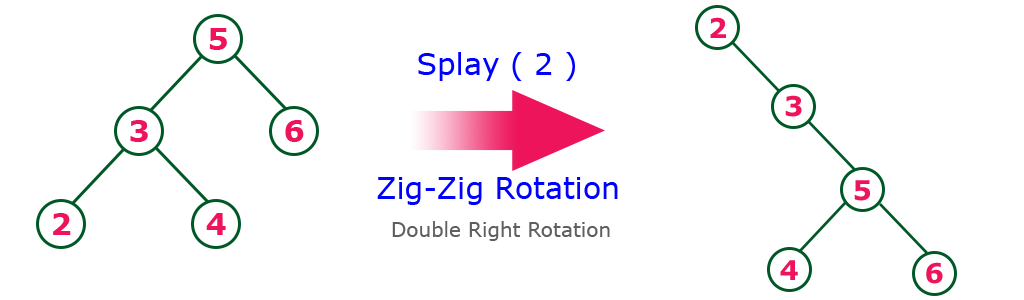
The **Zag Rotation** in splay tree is similar to the single left rotation (RR) in AVL Tree rotations.

In zag rotation, every node moves one position to the left from its current position. Consider the following example...



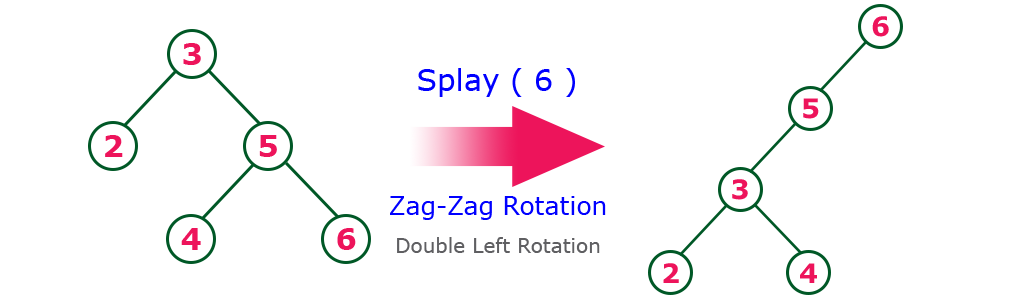
# Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



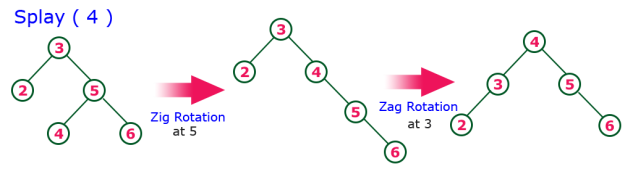
# Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



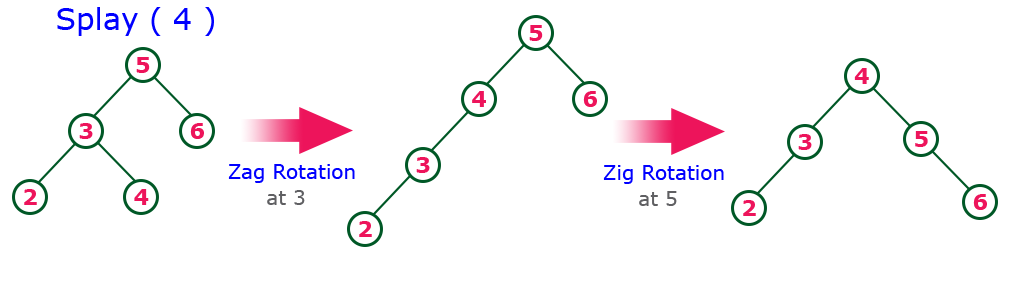
# Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



# Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



**Every Splay tree must be a binary search tree but it is need not to be balanced tree.**

# Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

* **Step 1 -**Check whether tree is Empty.
* **Step 2 -**If tree is Empty then insert the **newNode** as Root node and exit from the operation.
* **Step 3 -**If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
* **Step 4 -**After insertion, **Splay** the **newNode**

# Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

# Red - Black Tree Data structure

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...

**Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.**

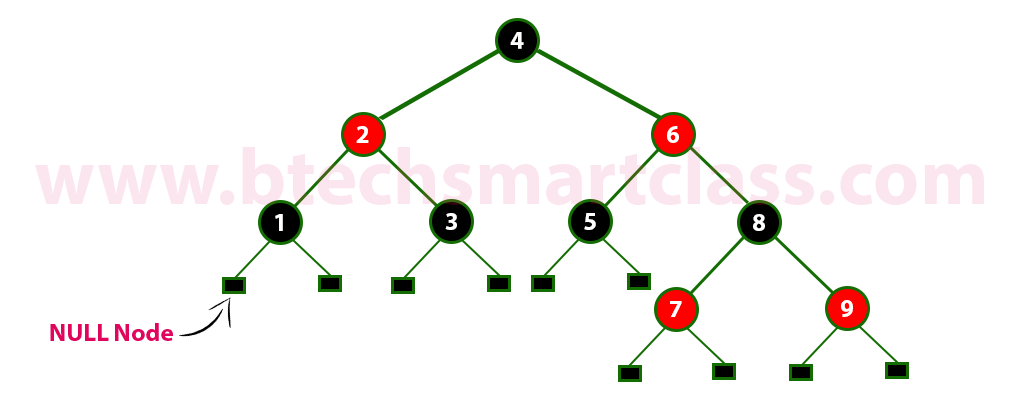
In Red Black Tree, the color of a node is decided based on the properties of Red Black Tree. Every Red Black Tree has the following properties.

# Properties of Red Black Tree

* **Property #1:** Red - Black Tree must be a Binary Search Tree.
* **Property #2:** The **ROOT** node must be colored **BLACK.**
* **Property #3:** The **children of Red** colored node must be colored **BLACK.** (There should not be two consecutive RED nodes).
* **Property #4:** In **all the paths** of the tree, there should be **same number of BLACK** colored nodes.
* **Property #5:** Every **new node** must be **inserted with RED** color.
* **Property #6:** **Every leaf** (e.i. NULL node) must be **colored BLACK.**

##### Example

Following is a Red Black Tree which is created by inserting numbers from 1 to 9.



The above tree is a Red Black tree where every node is satisfying all the properties of Red Black Tree.

**Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.**

# Insertion into RED BLACK Tree

In a Red Black Tree, every new node must be inserted with color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform following operation to make it Red Black Tree.

**1. Recolor**

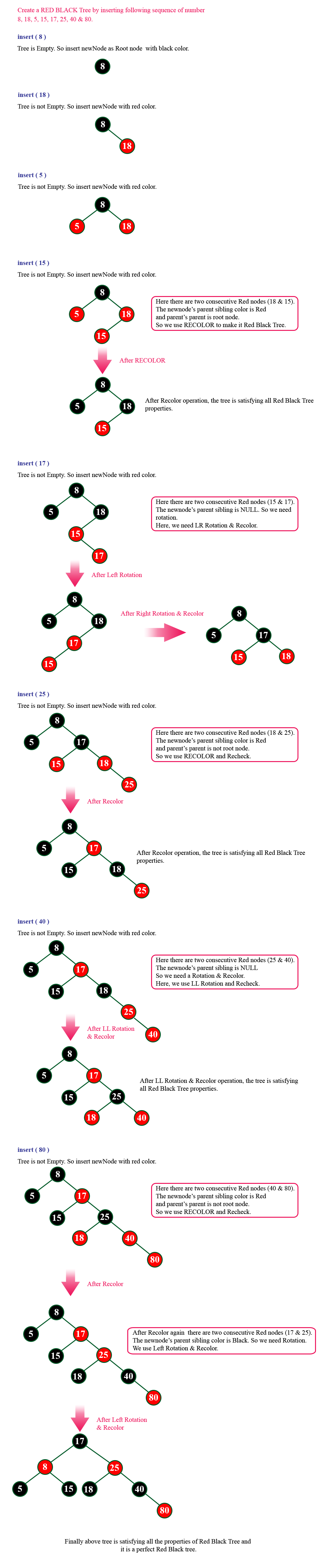
**2. Rotation**

**3. Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using following steps...

* **Step 1 -**Check whether tree is Empty.
* **Step 2 -**If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.
* **Step 3 -**If tree is not Empty then **insert the newNode** as leaf node with **color Red**.
* **Step 4 -**If the parent of newNode is Black then exit from the operation.
* **Step 5 -**If the parent of newNode is Red then check the color of parent node's sibling of newNode.
* **Step 6 -**If it is colored Black or NULL then make suitable Rotation and Recolor it.
* **Step 7 -**If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

##### Example



# Red Black Tree Construction

# Red Black Tree Construction

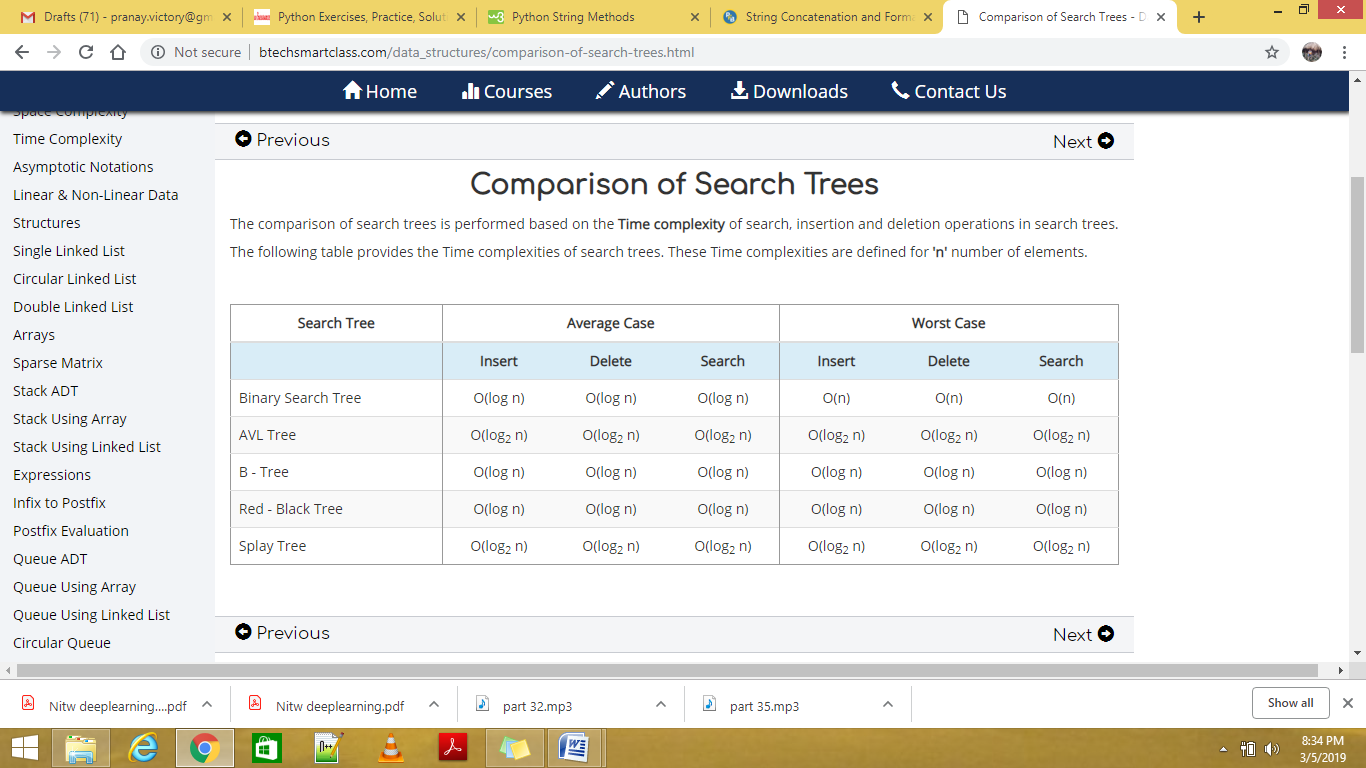
# Red Black Tree Construction

# Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation we need to check with the Red Black Tree properties. If any of the properties is violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

**Comparison of Search Trees**

The comparison of search trees is performed based on the **Time complexity** of search, insertion and deletion operations in search trees. The following table provides the Time complexities of search trees. These Time complexities are defined for **'n'** number of elements.

****