Unit-III

Tree

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

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

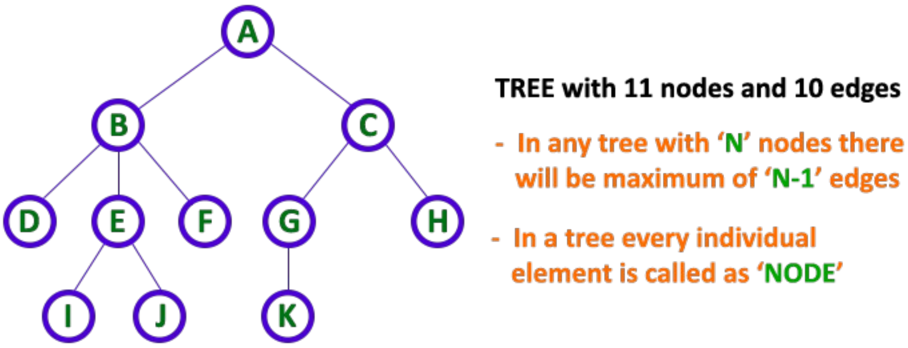
A tree data structure can also be defined as follows...

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

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

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

Ex:

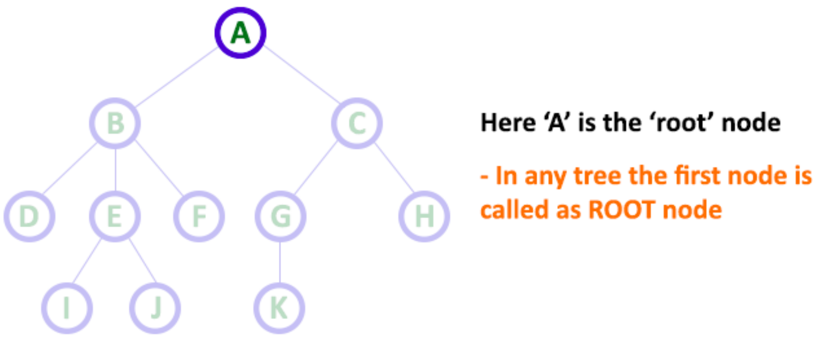


Terminology

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

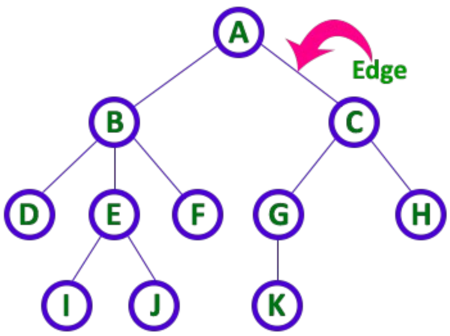
Root

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



Edge

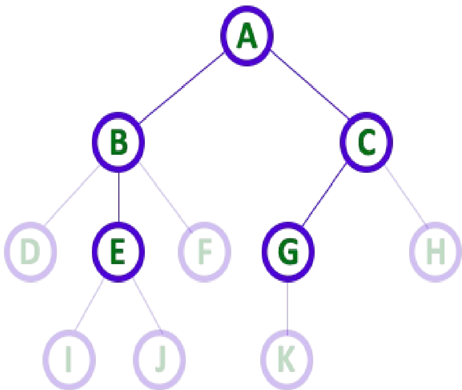
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



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

Parent

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

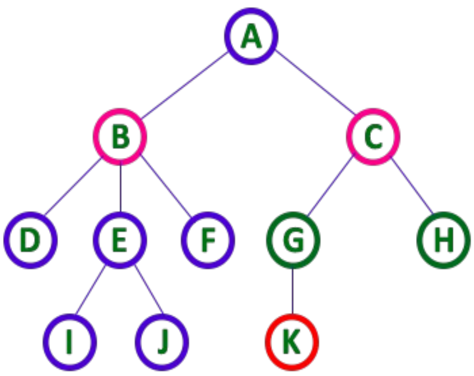


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

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.



Here B & C are Children of A

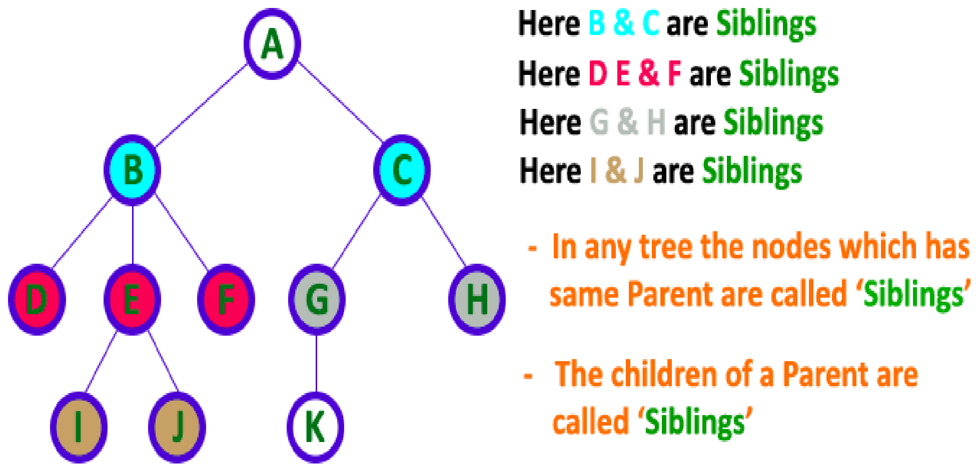
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

Siblings

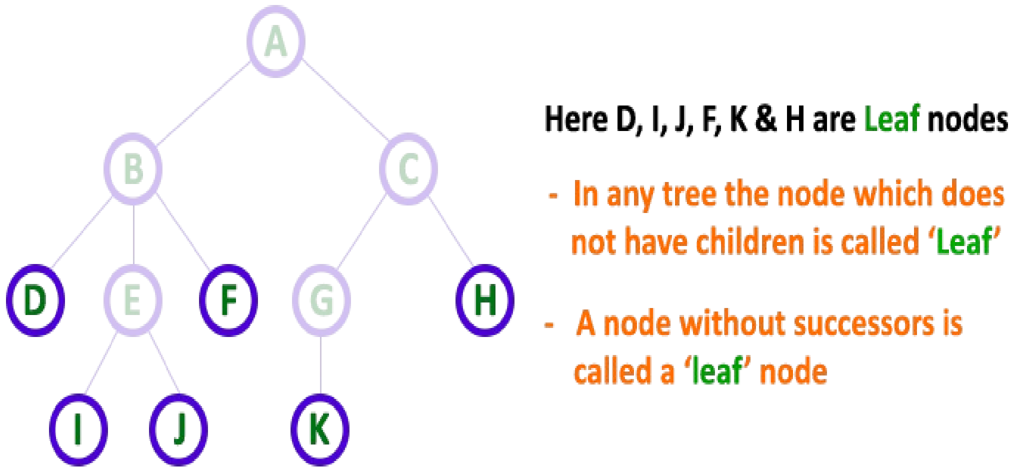
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



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.

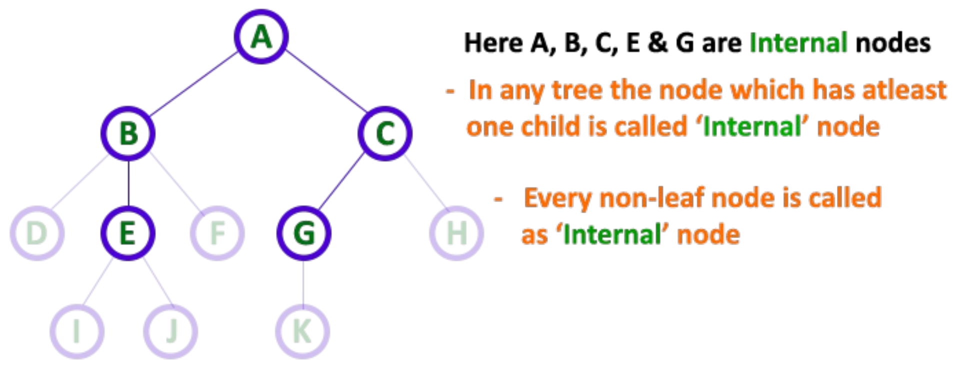
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



Internal Nodes

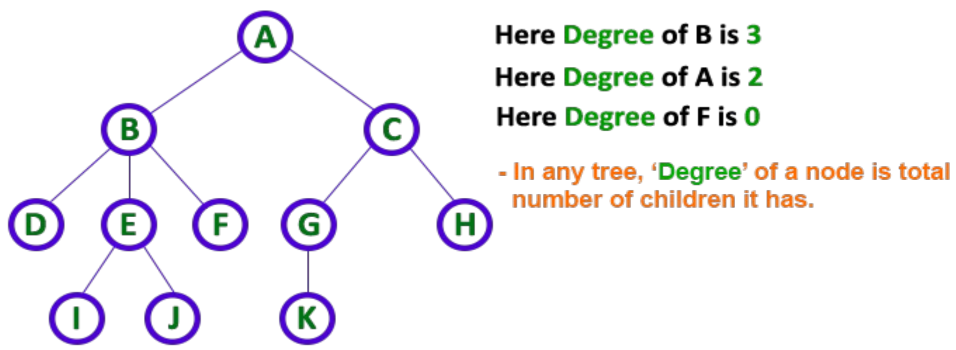
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

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.



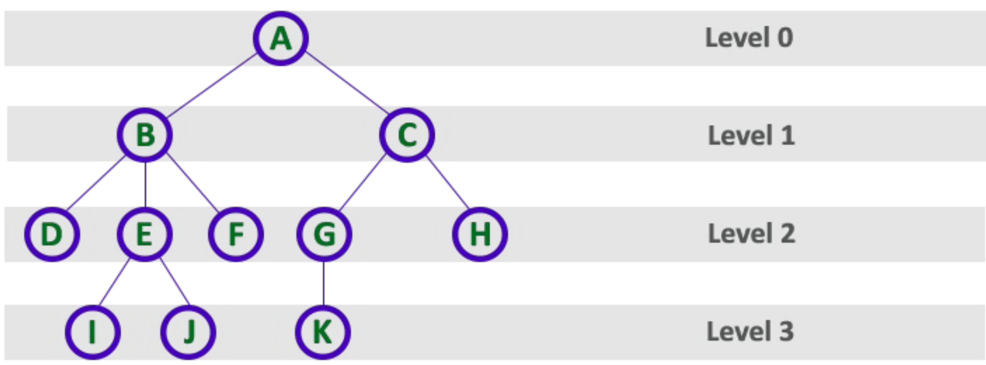
Degree

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**'



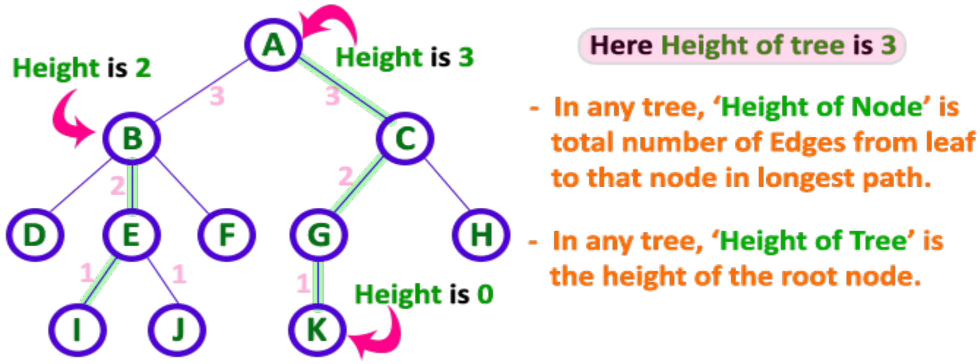
Level

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).



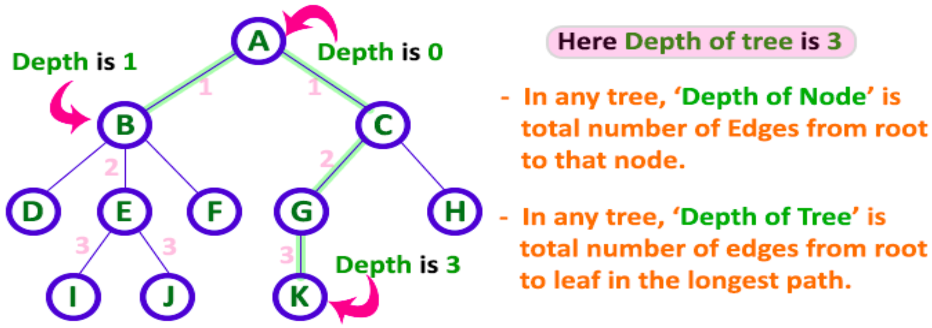
Height

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. <u>In a tree, height of the root node is said to be **height of the tree**</u>. In a tree, <u>**height of all leaf nodes is '0'.**</u>



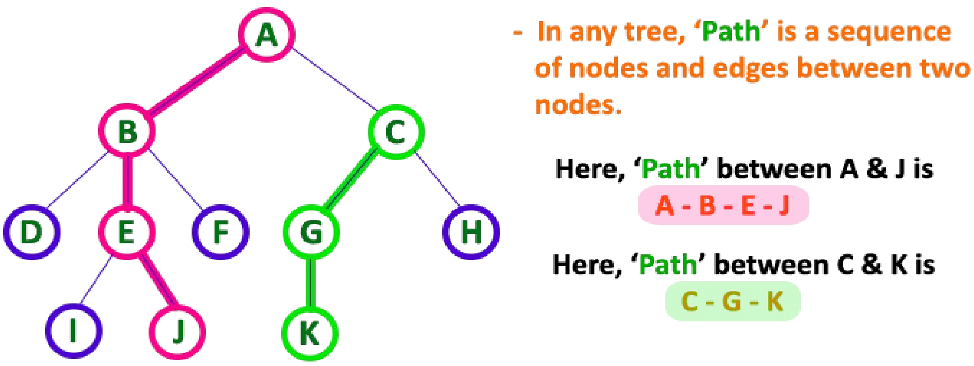
Depth

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'.**



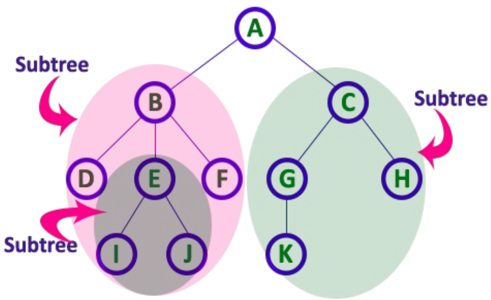
Path

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**.



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.



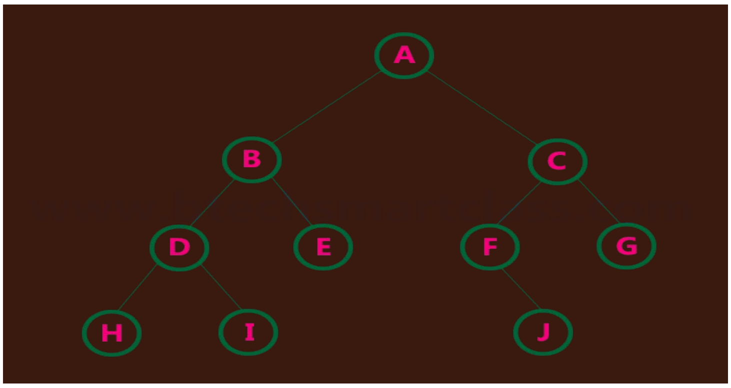
Binary Tree

In a normal tree, every node can have any number of children. A 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 a 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 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.

Ex:



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

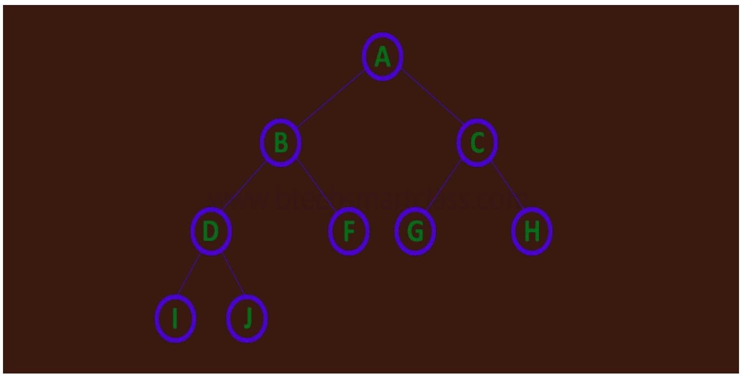
1. Strictly Binary Tree

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**

Ex

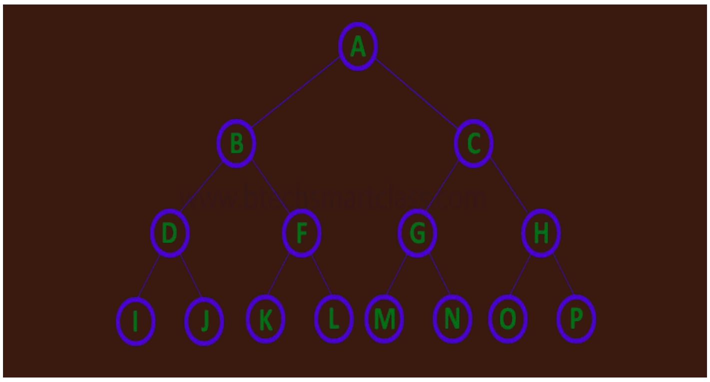


2. Complete Binary Tree

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

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

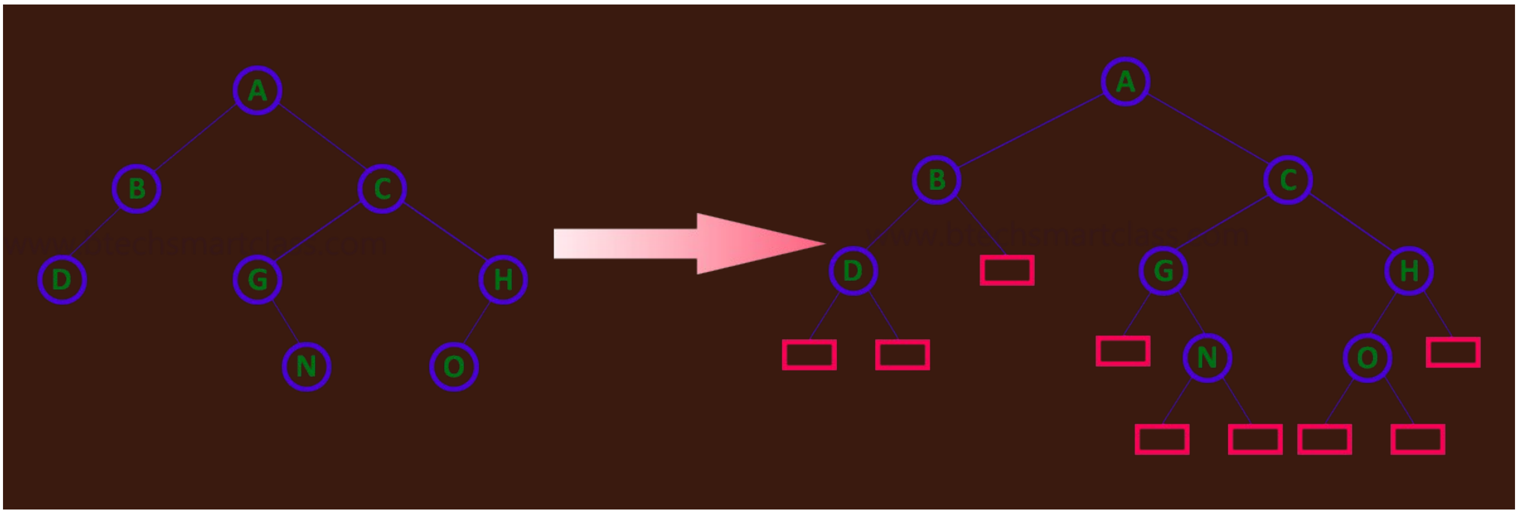
Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

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

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



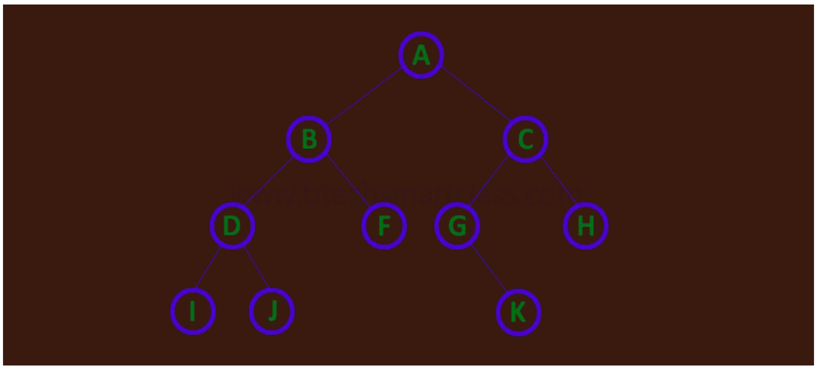
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

Binary Tree Representations

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

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

Consider the following binary tree...



1. Array Representation of Binary Tree

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

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

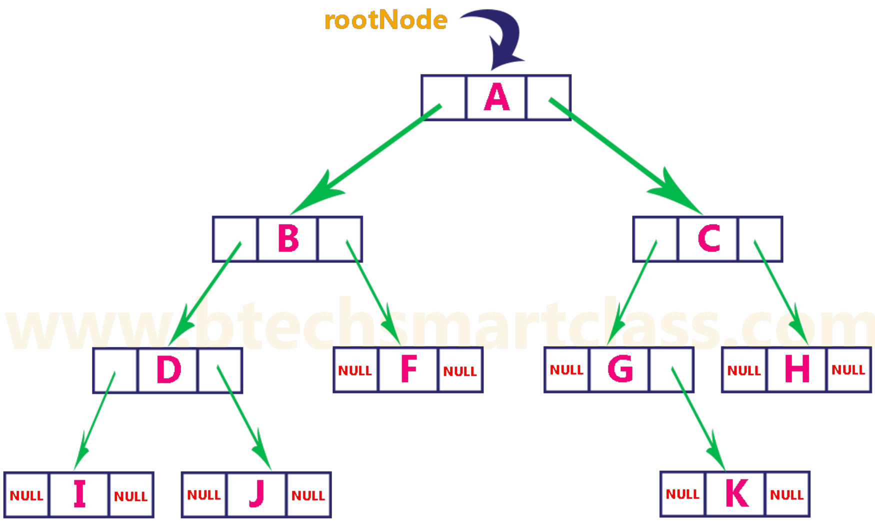
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2n + 1$.

2. Linked List Representation of Binary Tree

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

Left Child Address	Data	Right Child Address

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



Binary Tree Traversals

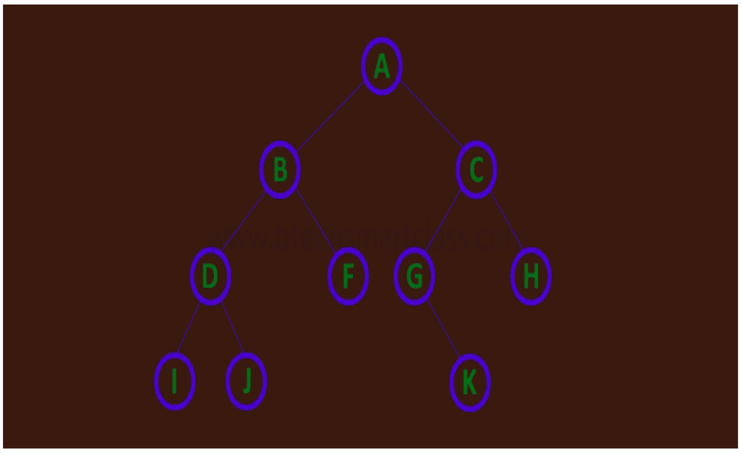
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

- 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 the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left 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 leftmost child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**. With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A. Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a 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 rightmost child in the tree. So we stop the process.

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

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

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

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

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all 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 leftmost child. So next we go for visiting D's right child **'J'**. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**. With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this, we have completed node C's root and left parts. Next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.

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

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

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

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

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

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

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

I - J - D - F - B - K - G - H - C – A

```
//Binary Tree Display using In-Order Traversals
#include<stdio.h>
#include<conio.h>

struct Node{
   int data;
   struct Node *left;
   struct Node *right;
};

struct Node *root = NULL;
int count = 0;

struct Node* insert(struct Node*, int);
void display(struct Node*);

void main(){
   int choice, value;
   clrscr();
   printf("\n----- Binary Tree -----\n");
   while(1){
      printf("\n***** MENU *****\n");
      printf("1. Insert\n2. Display\n3. Exit");
      printf("\nEnter your choice: ");
      scanf("%d",&choice);
      switch(choice){
		case 1: printf("\nEnter the value to be insert: ");
		scanf("%d", &value);
		root = insert(root,value);
		break;
		case 2: display(root); break;
		case 3: exit(0);
		default: printf("\nPlease select correct operations!!!\n");
      }
   }
}

struct Node* insert(struct Node *root,int value){
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(root == NULL){
      newNode->left = newNode->right = NULL;
      root = newNode;
      count++;
   }
   else{
      if(count%2 != 0)
		root->left = insert(root->left,value);
      else
		root->right = insert(root->right,value);
   }
   return root;
}
// display is performed by using Inorder Traversal
void display(struct Node *root)
{
   if(root != NULL){
      display(root->left);
      printf("%d\t",root->data);
      display(root->right);
   }
}
```

Search Trees

Binary Search Tree

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

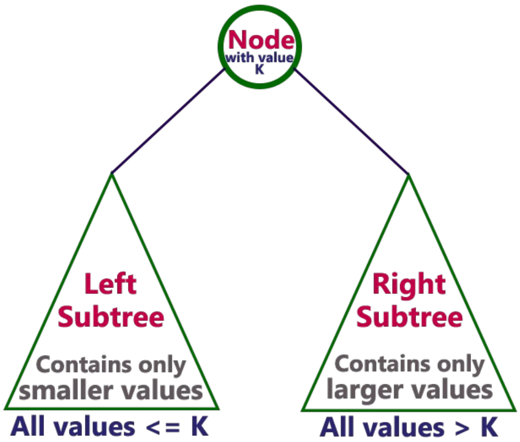
A binary tree has the following time complexities...

1. **Search Operation – O(n)**
2. **Insertion Operation – O(1)**
3. **Deletion Operation – O(n)**

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

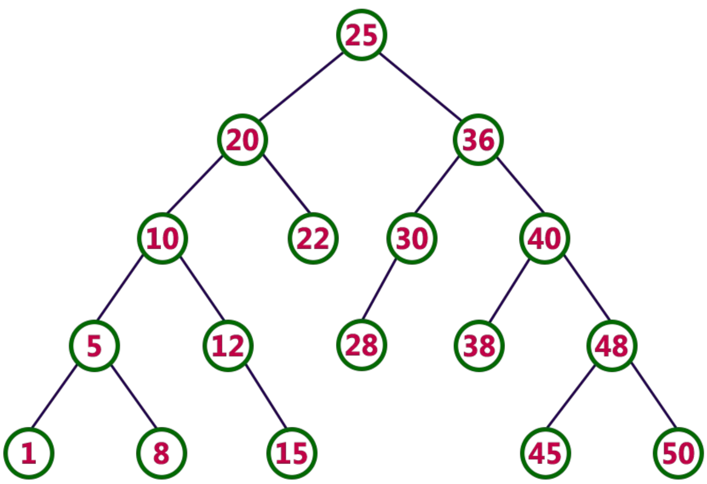
- Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

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



Ex:

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



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

Operations on a Binary Search Tree

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

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.

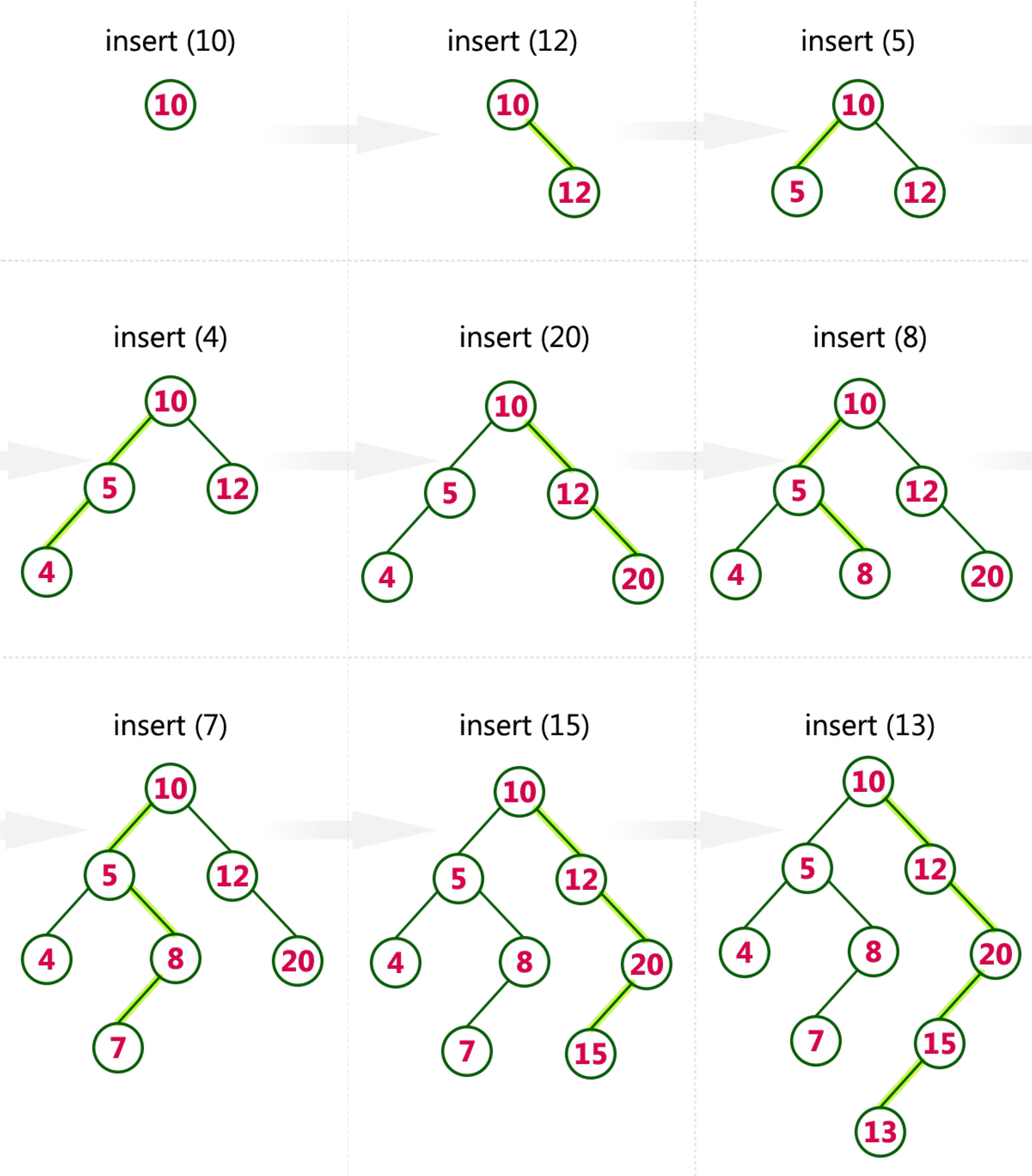
Step 7 - Repeat the same process until the node is deleted from the tree.

Ex:

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...


insert (10)
insert (12)
insert (5)
insert (4)
insert (20)
insert (8)
insert (7)
insert (15)
insert (13)

```
//Binary Search Tree Implementation
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

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


void inorder(struct node *root)
{
	if(root)
	{
	inorder(root->left);
	printf("  %d",root->data);
	inorder(root->right);
	}
}

int main()
{
	int n , i;
	struct node *p , *q , *root;
	printf("Enter the number of nodes to be insert: ");
	scanf("%d",&n);

	printf("\nPlease enter the numbers to be insert: ");

	for(i=0;i<i++)
	{
	p = (struct node*)malloc(sizeof(struct node));
	scanf("%d",&p->data);
	p->left = NULL;
	p->right = NULL;
	if(i == 0)
	{
		root = p; // root always point to the root node
	}
	else
	{
		q = root;   // q is used to traverse the tree
		while(1)
		{
			if(p->data > q->data)
			{
				if(q->right == NULL)
					{
					q->right = p;
					break;
					}
				else
					q = q->right;
			}
			else
			{
				if(q->left == NULL)
					{
					q->left = p;
					break;
					}
				else
					q = q->left;
			}
		}

	}

	}

printf("\nBinary Search Tree nodes in Inorder Traversal: ");
inorder(root);
printf("\n");

return 0;
}
```

AVL Tree

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the 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.

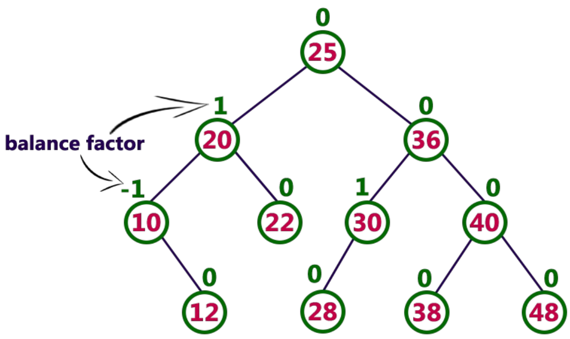
An AVL tree is defined as follows...

- 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 the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree – heightOfRightSubtree

Ex:



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

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

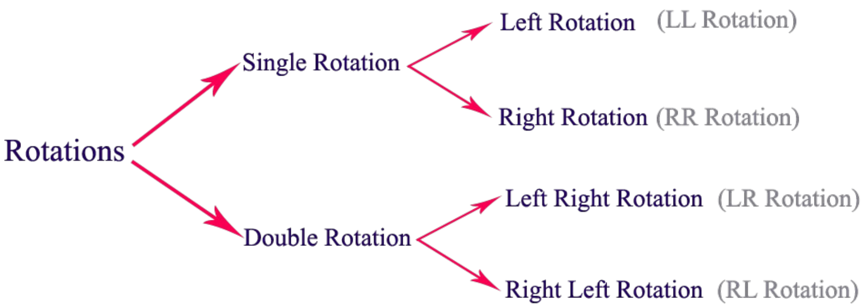
AVL Tree Rotations

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

Rotation operations are used to make the tree balanced.

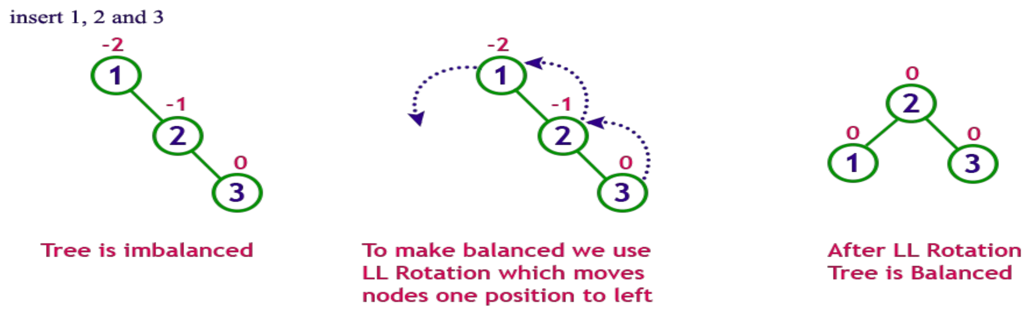
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.

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



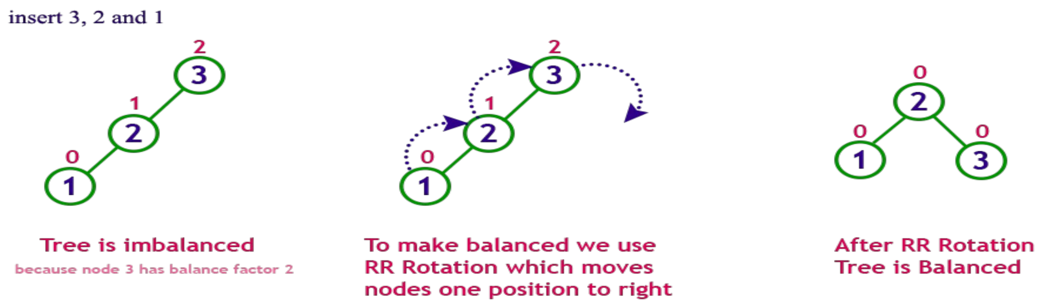
Single Left Rotation (LL Rotation)

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



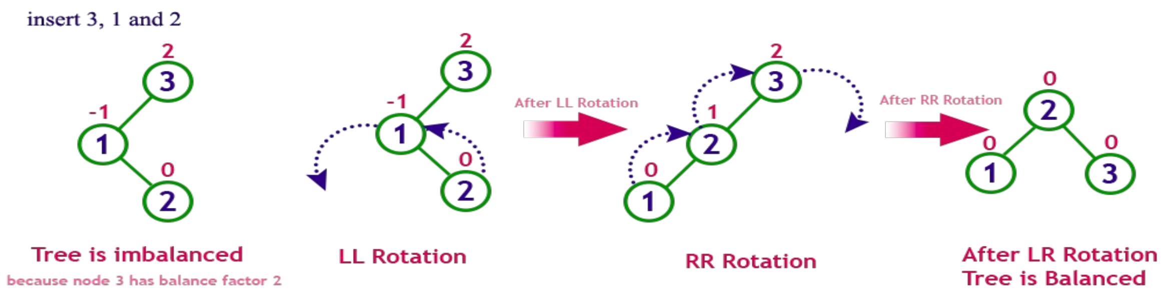
Single Right Rotation (RR Rotation)

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



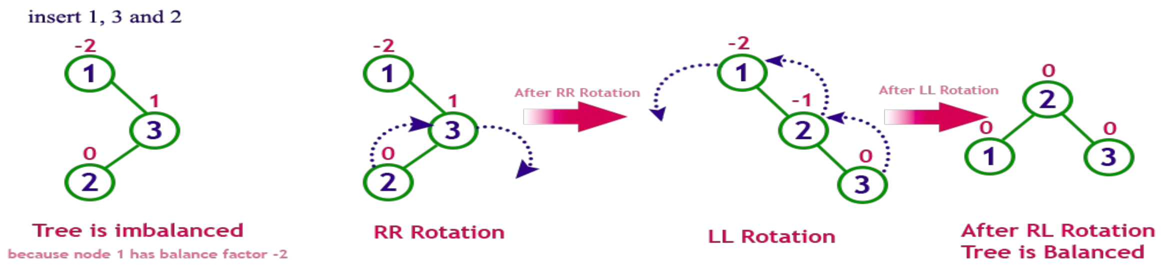
Left Right Rotation (LR Rotation)

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



Right Left Rotation (RL Rotation)

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



Operations on an AVL Tree

The following operations are performed on AVL tree...

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

Search Operation in AVL Tree

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

Step 1 - Read the search element from the user.

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

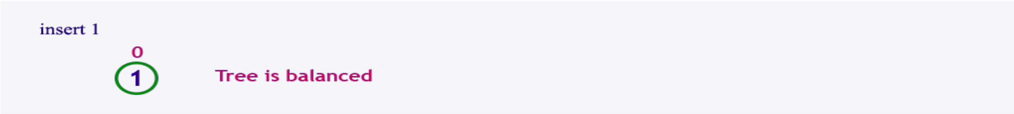
Insertion Operation in AVL Tree

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

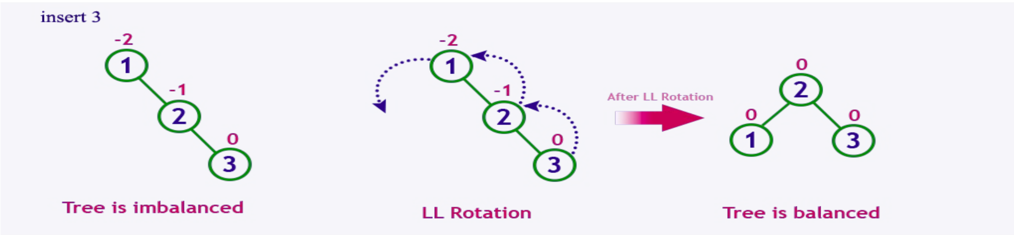
Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the **Balance Factor** of every node.
Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

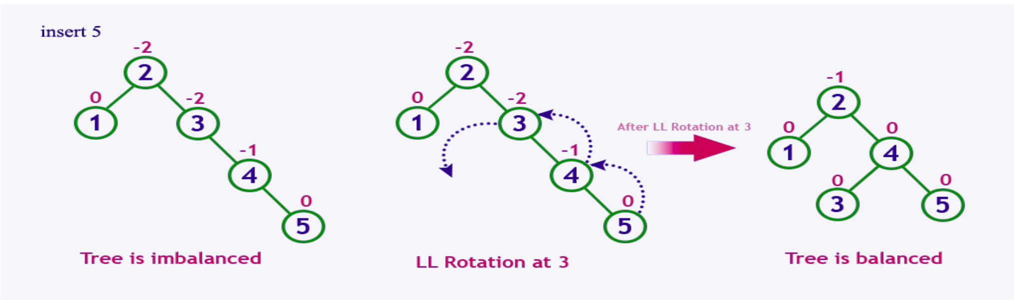
Ex: Construct an AVL Tree by inserting numbers from 1 to 8.

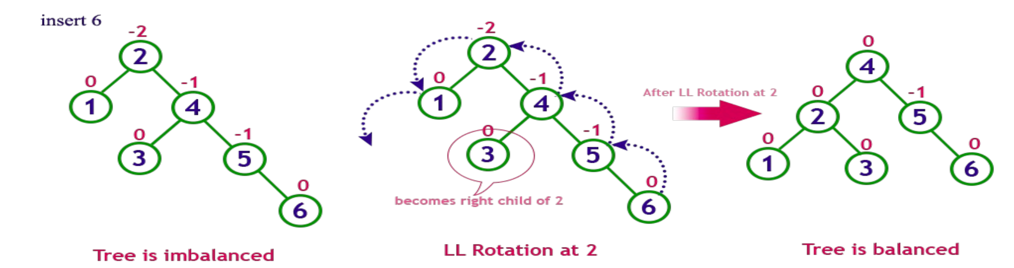

insert 1
Tree is balanced

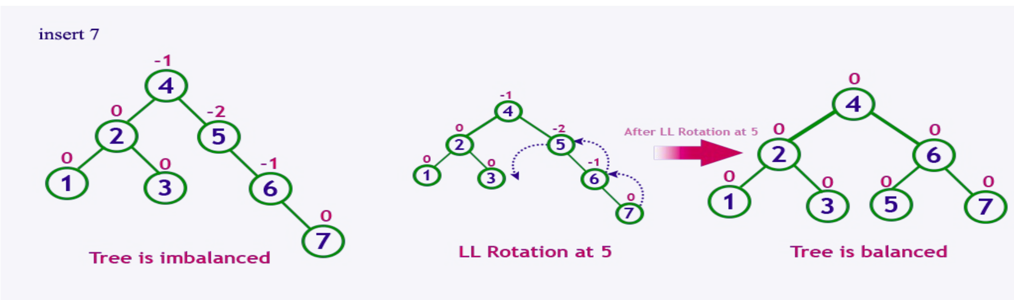

insert 2
Tree is balanced

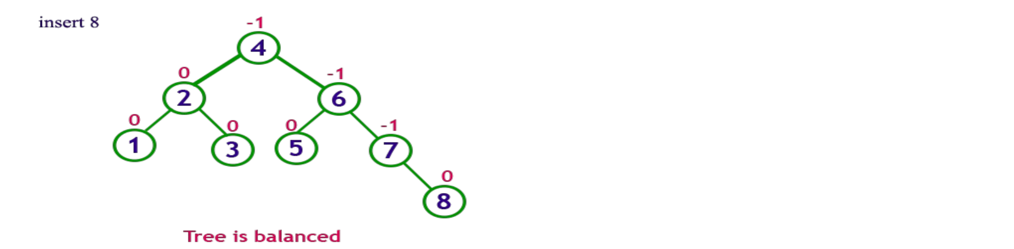

insert 3
Tree is imbalanced
LL Rotation
After LL Rotation
Tree is balanced


insert 4
Tree is balanced


insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced


insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced


insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced


insert 8
Tree is balanced

Deletion Operation in AVL Tree

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

Red - Black

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.

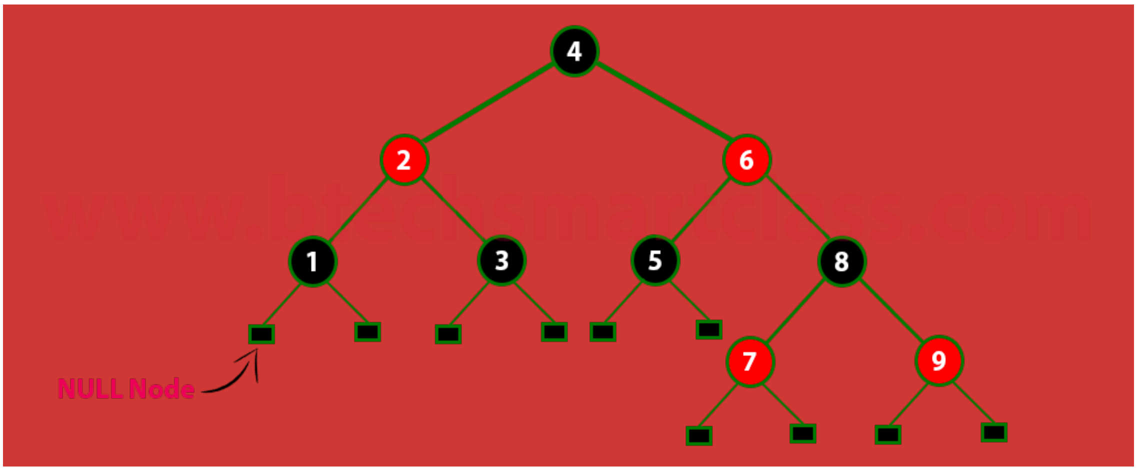
In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

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.

Ex: 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 the 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 the 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 the 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 parentnode'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.

Ex:

Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

insert (18)

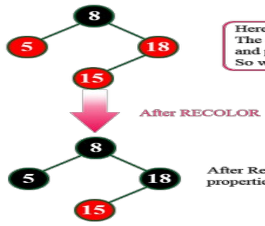
Tree is not Empty. So insert newNode with red color.

insert (5)

Tree is not Empty. So insert newNode with red color.

insert (15)

Tree is not Empty. So insert newNode with red color.

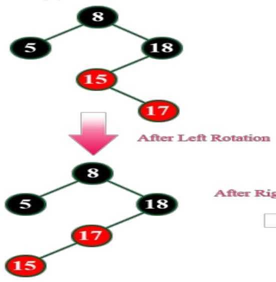


Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (17)

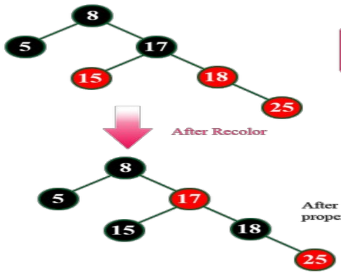
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

insert (25)

Tree is not Empty. So insert newNode with red color.

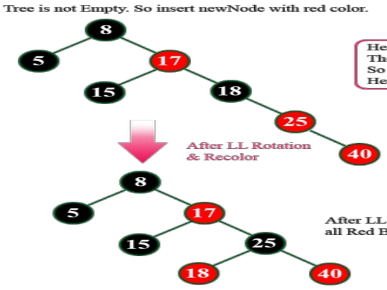


Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (40)

Tree is not Empty. So insert newNode with red color.

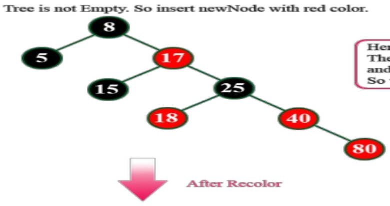


Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

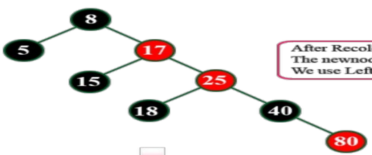
After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (80)

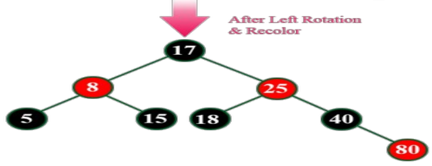
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.



After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.



Finally above tree is satisfying all the properties of Red Black Tree and
it is a perfect Red Black tree.

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 are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

Splay Tree

Splay tree is another variant of a binary search tree. In a 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 a splay tree, every operation is performed at the 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.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

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 the root of the tree. The search operation in a 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...

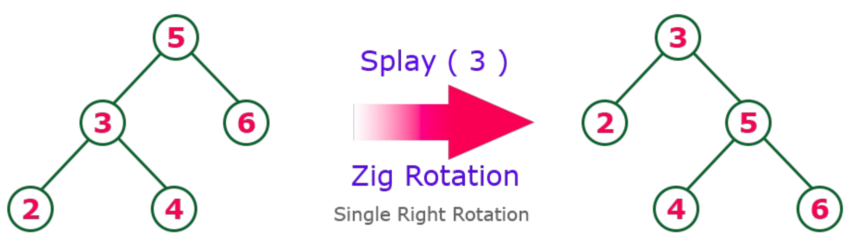
Rotations in Splay Tree

1. **Zig Rotation**
2. **Zag Rotation**
3. **Zig - Zig Rotation**
4. **Zag - Zag Rotation**
5. **Zig - Zag Rotation**
6. **Zag - Zig Rotation**

Ex:

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation 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

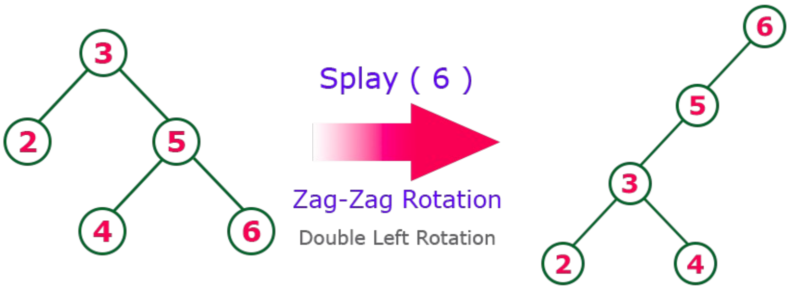
The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

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...

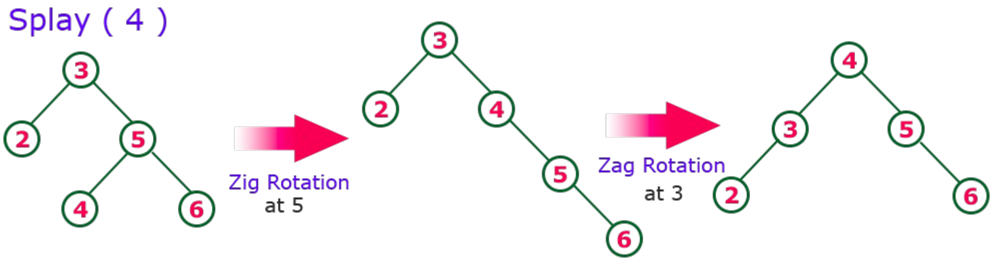
Zag-Zag Rotation

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...



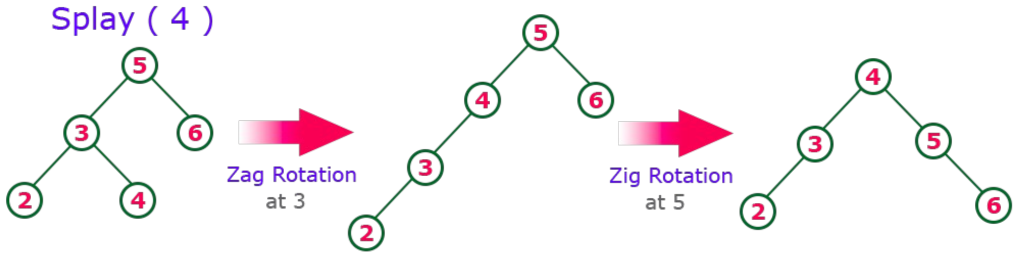
Zig-Zag Rotation

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.