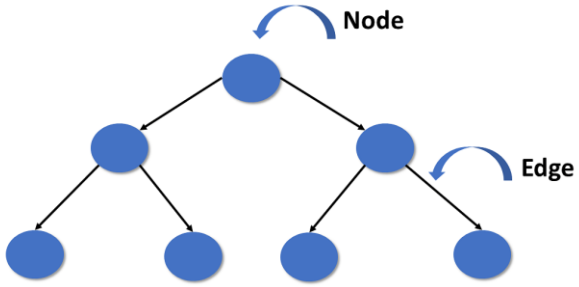
Trees and binary trees: Introduction, Trees: definition and basic terminologies, representation of trees. Binary trees: basic terminologies and types, representation of binary trees, binary tree traversals, applications. **Binary search trees and AVL trees**: Introduction, binary search trees: definition and operations, **AVL Trees**: definition and operations, applications. **Heaps**: Heaps, Priority Queues, Definition of a Max Heap, Insertion into a Max Heap, Deletion from a Max Heap, Applications: Heap Sort.

Introduction to Trees:

A tree is a hierarchical data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges. A tree has the following properties:

1. The tree has one node called root. The tree originates from this, and hence it does not have any parent.
2. Each node has one parent only but can have multiple children.
3. Each node is connected to its children via edge.



The Necessity for a Tree in Data Structures

Other data structures like arrays, linked-list, stacks, and queues are linear data structures, and all these data structures store data in sequential order. Time complexity increases with increasing data size to perform operations like insertion and deletion on these linear data structures. But it is not acceptable for today's world of computation.

The non-linear structure of trees enhances the data storing, data accessing, and manipulation processes by employing advanced control methods traversal through it. You will learn about tree traversal in the upcoming section.

Tree Node

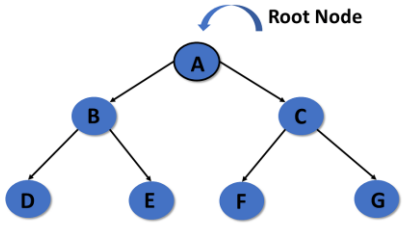
A node is a structure that contains a key or value and pointers in its child node in the tree data structure.

Structure of a node:

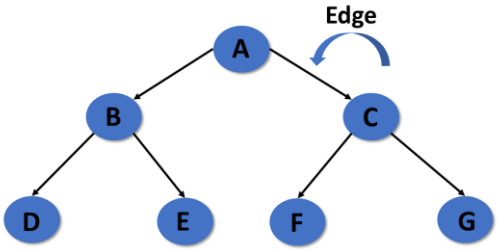
```
Structnode{
        Int data;
        Struct node *left;
        Struct node *right;
};
```

Tree terminology:

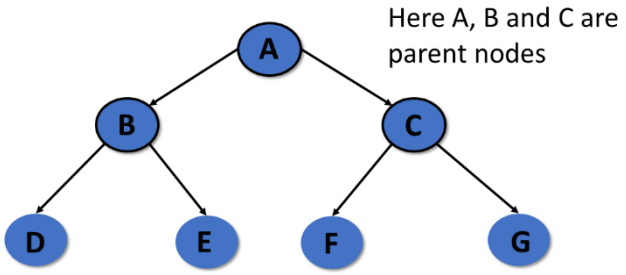
- **Root:** In a tree data structure, the root is the first node of the tree. The root node is the initial node of the tree in data structures. In the tree data structure, there must be only one root node.



- **Edge:** In a tree in data structures, the connecting link of any two nodes is called the edge of the tree data structure. In the tree data structure, N number of nodes connecting with N -1 number of edges.

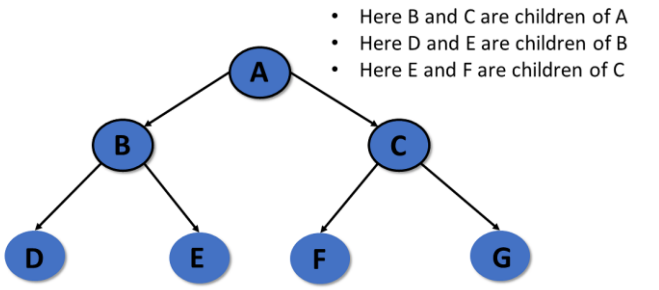


- **Parent :**In the tree in data structures, the node that is the predecessor of any node is known as a parent node, or a node with a branch from itself to any other successive node is called the parent node.

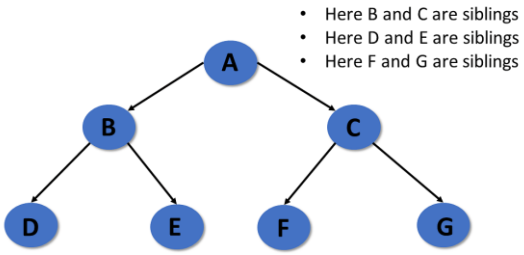


- **Child**:The node, a descendant of any node, is known as child nodes in data structures. In a tree, any number of parent nodes can have any number of child nodes.In a tree, every node except the root node is a child node.

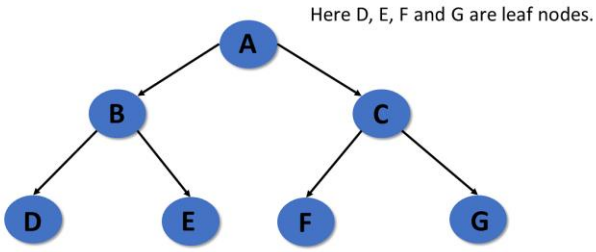
- **Sibling**: In trees nodes that parent are



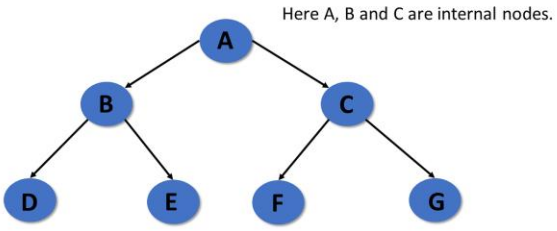
in the data structure, belong to the same called siblings.



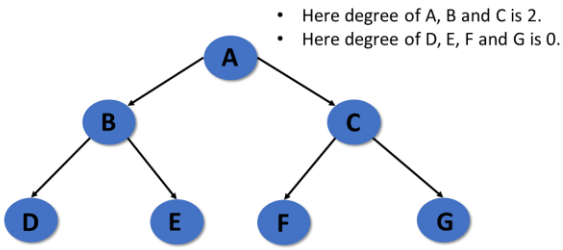
- **Leaf**: Trees in the data structure, the node with no child, is known as a leaf node. In trees, leaf nodes are also called external nodes or terminal nodes



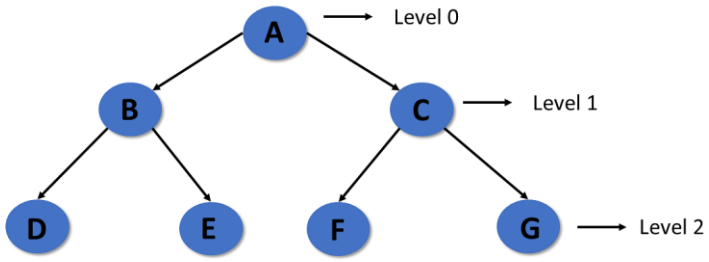
- **Internal nodes:**Trees in the data structure have at least one child node known as internal nodes.In trees, nodes other than leaf nodes are internal nodes.Sometimes root nodes are also called internal nodes if the tree has more than one node.



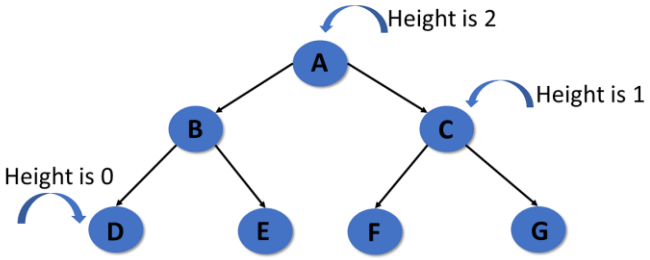
- **Degree** :In the tree data structure, the total number of children of a node is called the degree of the node. The highest degree of the node among all the nodes in a tree is called the Degree of Tree.



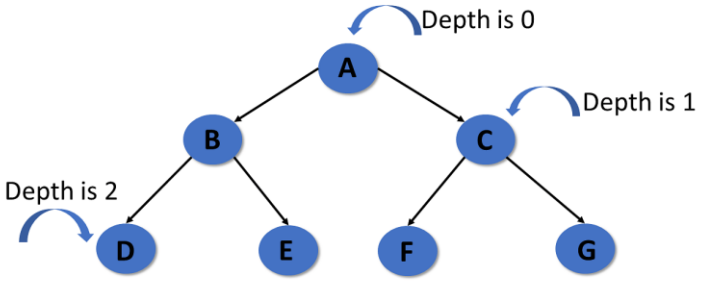
- **Level:**In tree data structures, the root node is said to be at level 0, and the root node's children are at level 1, and the children of that node at level 1 will be level 2, and so on.



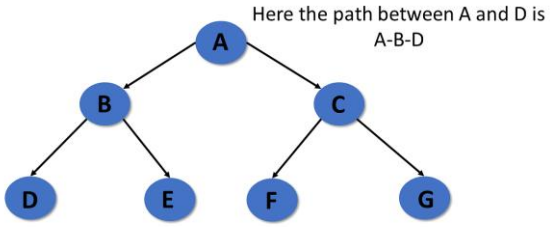
- **Height:** In a tree data structure, the number of edges from the leaf node to the particular node in the longest path is known as the height of that node. In the tree, the height of the root node is called "Height of Tree". The tree height of all leaf nodes is 0.



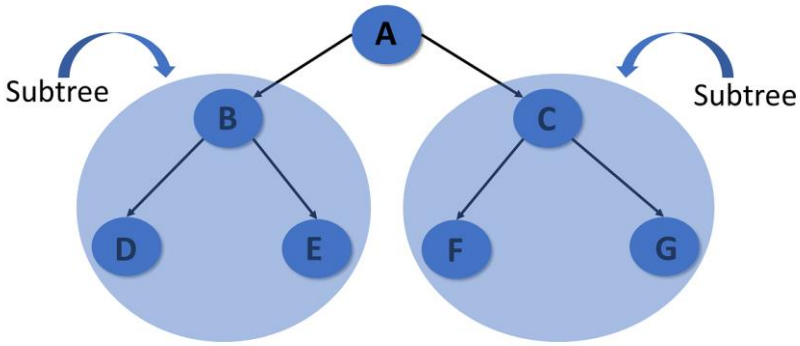
- **Depth:** In a tree, many edges from the root node to the particular node are called the depth of the tree. In the tree, the total number of edges from the root node to the leaf node in the longest path is known as "Depth of Tree". In the tree data structures, the depth of the root node is 0.



- **Path:** In the tree in data structures, the sequence of nodes and edges from one node to another node is called the path between those two nodes. The length of a path is the total number of nodes in a path.



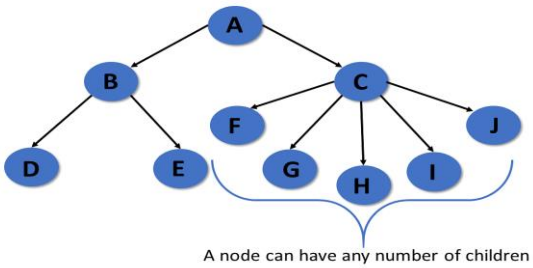
- **Subtree:**In the tree in data structures, each child from a node shapes a sub-tree recursively and every child in the tree will form a sub-tree on its parent node.



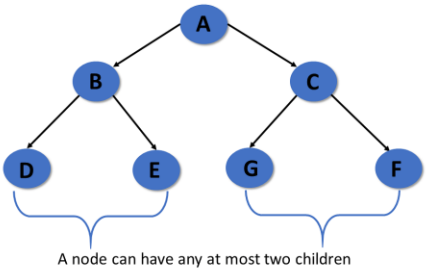
Types of Trees:

Types of trees depend on the number of children a node has. There are two major tree types:

1. **General Tree**: A tree in which there is no restriction on the number of children a node has, is called a General tree. Examples are Family tree, Folder Structure.



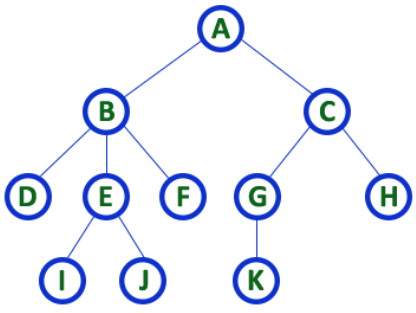
2. **Binary Tree:** In a Binary tree, every node can have at most 2 children, left and right. In diagram below, B & D are left children and C, E & F are right children.



Representation of Trees:

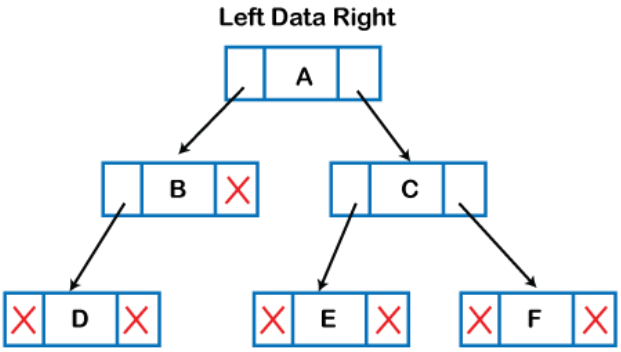
Trees can be represented in two ways:

1. With arrays:



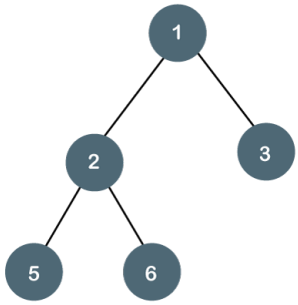
0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J	K

2. With linked list:



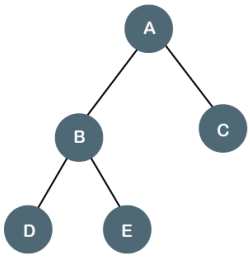
Binary tree:

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



Binary trees are further divided into many types based on its application.

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

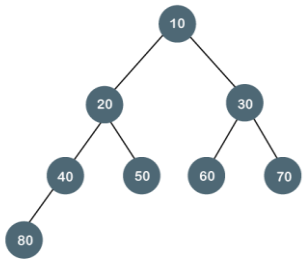


Properties of Full Binary Tree

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

n= 2*h - 1
n+1 = 2*h
h = n+1/2

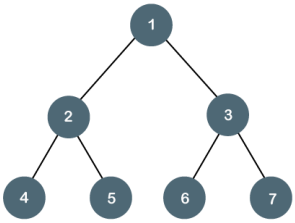
2. Complete Binary Tree: The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.



Properties of Complete Binary Tree

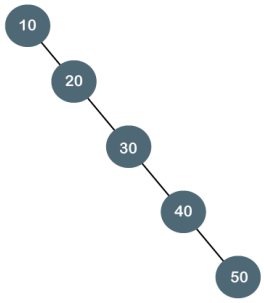
- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

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

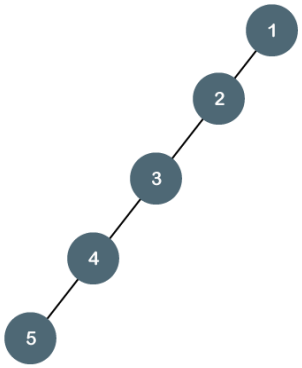


4. Degenerate Binary Tree: The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.

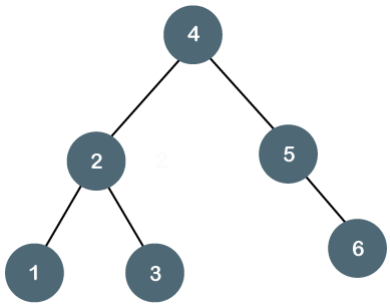


The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.



5.Balanced Binary Tree: The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.

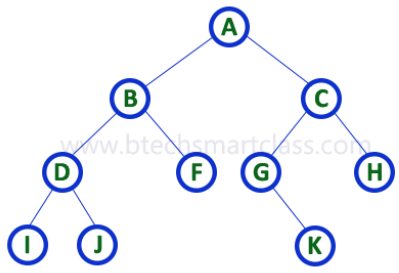


Representation of binary trees:

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

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

Consider the following binary tree...



1. Array Representation of Binary Tree

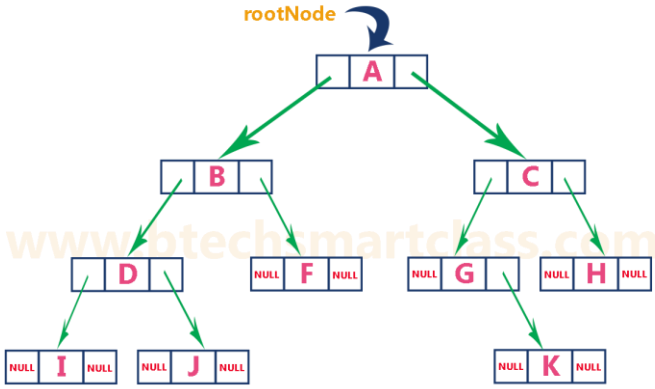
In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
Consider the above example of a binary tree and it is represented as follows...

To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **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 childaddress.
In this linked list representation, a node has the following structure...

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



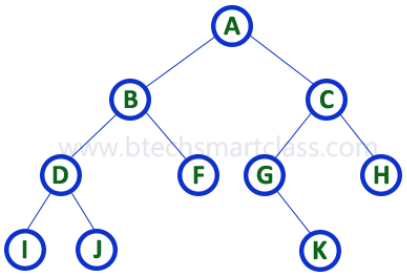
Binary tree traversals:

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.

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 :

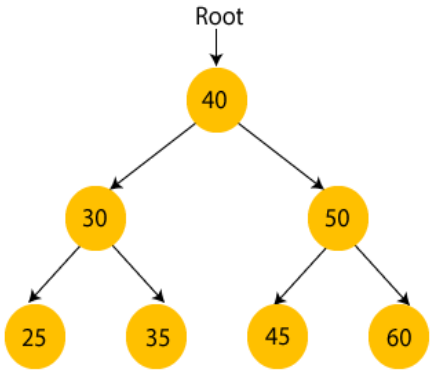
I - J - D - F - B - K - G - H - C - A

Applications:

- Binary Tree is used to as the basic data structure in Microsoft Excel and spreadsheets in usual.
- Binary Tree is used to implement indexing of Segmented Database.
- Splay Tree (Binary Tree variant) is used in implemented efficient cache is hardware and software systems.
- Binary Space Partition Trees are used in Computer Graphics, Back face Culling, Collision detection, Ray Tracing and algorithms in rendering game graphics.
- Syntax Tree (Binary Tree with nodes as operations) are used to compute arithmetic expressions in compilers like GCC, AOCL and others.
- Binary Heap (Binary Tree variant of Heap) is used to implement Priority Queue efficiently which in turn is used in Heap Sort Algorithm.
- Binary Search Tree is used to search elements efficiently and used as a collision handling technique in Hash Map implementations.
- Balanced Binary Search Tree is used to represent memory to enable fast memory allocation.
- Huffman Tree (Binary Tree variant) is used internally in a Greedy Algorithm for Data Compression known as Huffman Encoding and Decoding.
- Merkle Tree/ Hash Tree (Binary Tree variant) is used in Blockchain implementations and p2p programs requiring signatures.
- Binary Tries (Tries with 2 child) is used to represent a routing data which vacillate efficient traversal.
- Morse code is used to encode data and uses a Binary Tree in its representation.
- Goldreich, Goldwasser and Micali (GGM) Tree (Binary Tree variant) is used compute pseudorandom functions using an arbitrary pseudorandom generator.
- Scapegoat tree (a self-balancing Binary Search Tree) is used in implementing Paul-Carole games to model a faulty search process.
- Treap (radomized Binary Search Tree)

Binary search trees

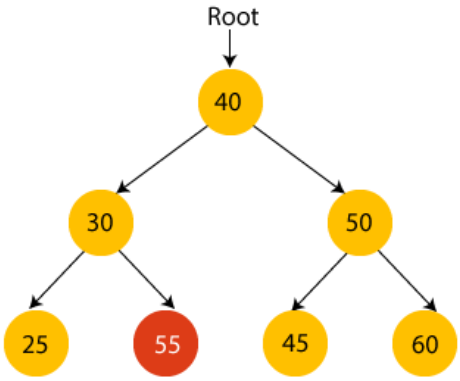
A binary search tree follows some order to arrange the elements. In a Binary search tree, **the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node**. This rule is applied recursively to the left and right subtrees of the root. Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

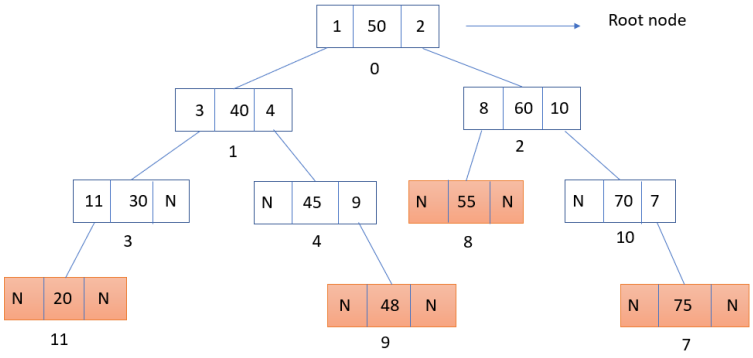
Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.
Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.

- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

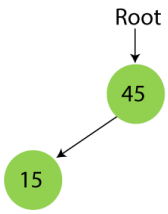
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below –



Step 1 - Insert 45.

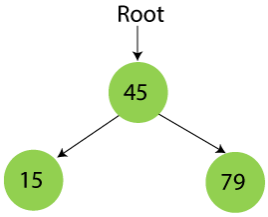
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



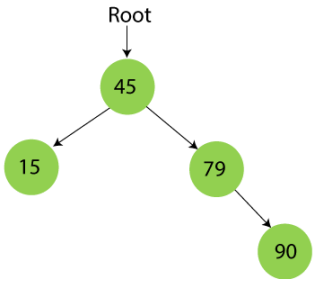
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.

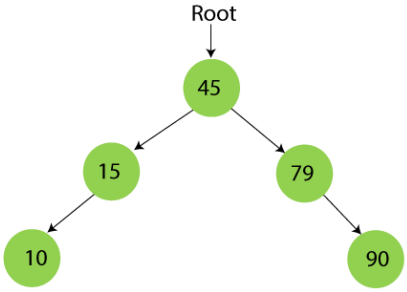


Step 4 - Insert 90.

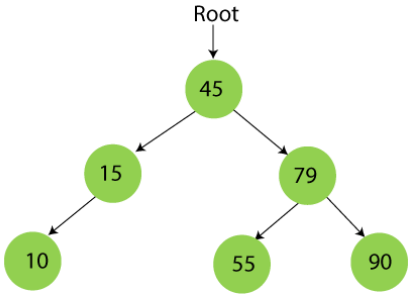
90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



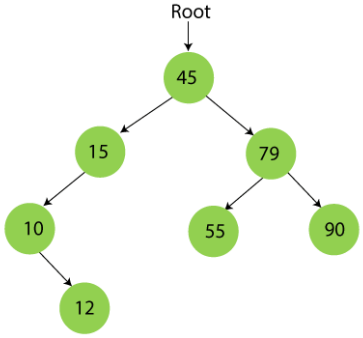
Step 5 - Insert 10.
10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



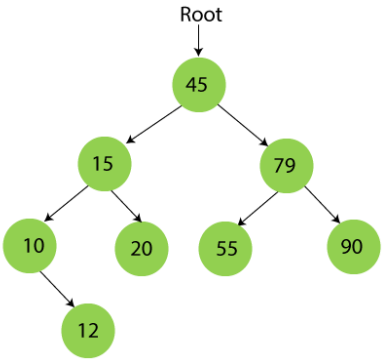
Step 6 - Insert 55.
55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



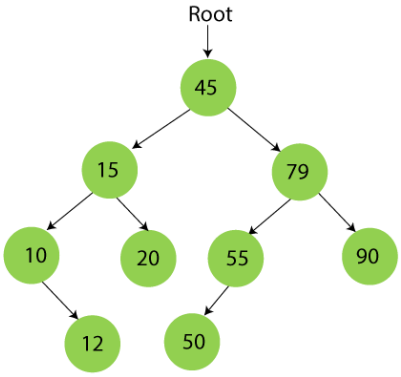
Step 7 - Insert 12.
12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



Step 8 - Insert 20.
20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.
50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

Operations on Binary Search Tree:

- ☒ Insertion
- ☒ Deletion
- ☒ Search
- ☒ Display

Insertion in Binary Search tree

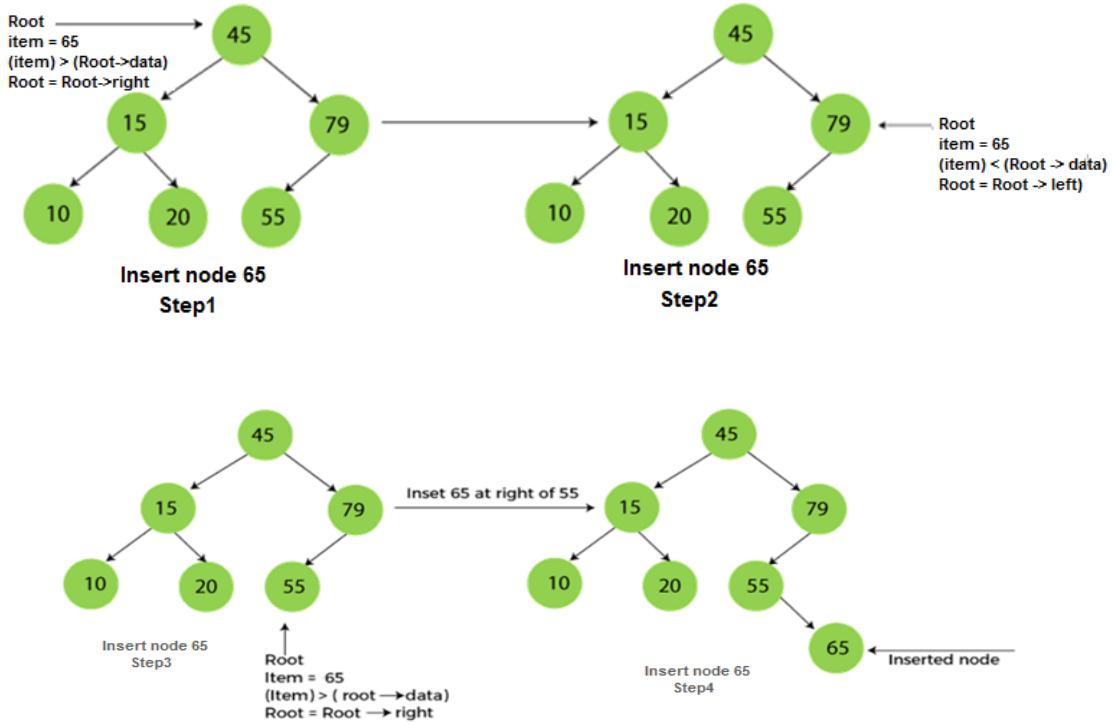
In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **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**.

Now, let's see the process of inserting a node into BST using an example.

Insert 65



Let's understand how a search is performed on a binary search tree.

Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

Step1 : First, compare the element to be searched with the root element of the tree.

Step 2: If root is matched with the target element, then return the node's location.

Step3: If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.

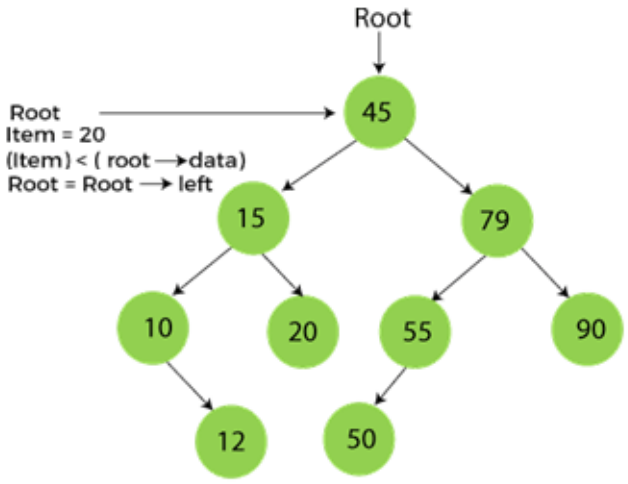
Step4: If it is larger than the root element, then move to the right subtree.

Step 5:Repeat the above procedure recursively until the match is found.

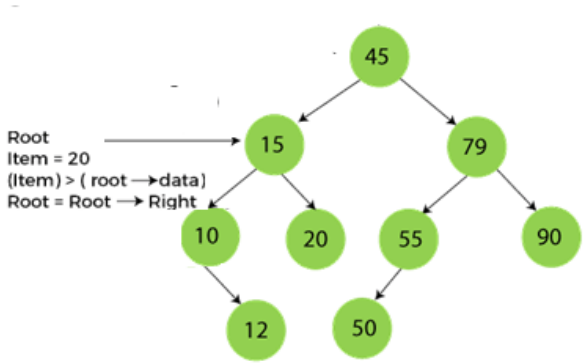
Step6: If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

Step1:



Step2:



Step3:

Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

Search (root, item)

```
Step 1 - if (item = root → data) or (root = NULL)
                return root
         else if (item < root → data)
                return Search(root → left, item)
        else
                return Search(root → right, item)
        END if
Step 2 - END
```

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Deletion in Binary Search tree

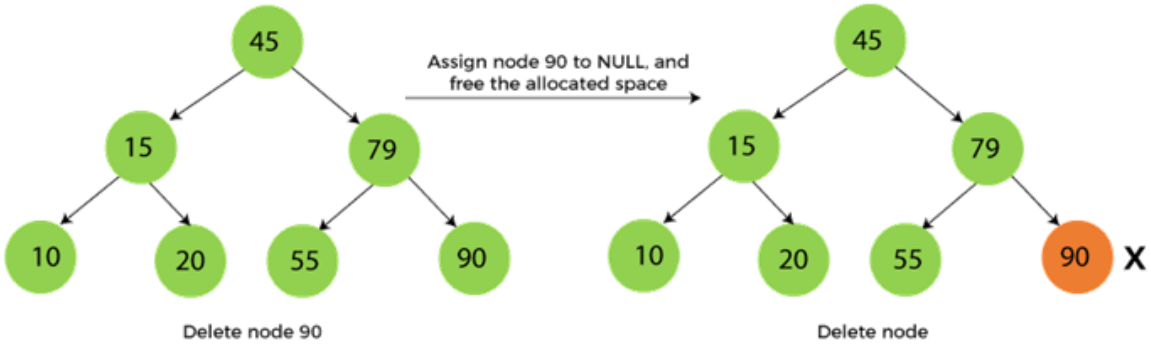
In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated.

To delete a node from BST, there are three possible situations occur -

- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

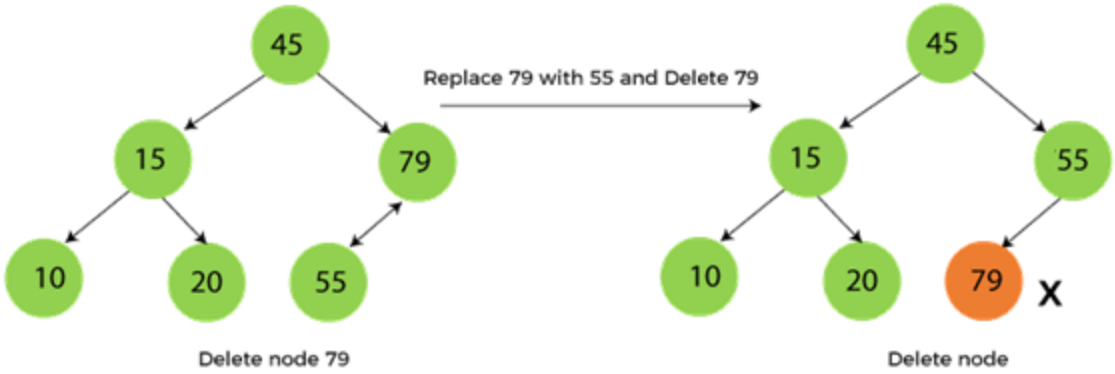
1. **When the node to be deleted is the leaf node:**

- It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.
- We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



2. When the node to be deleted has only one child

- In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted.
- So, we simply have to replace the child node with NULL and free up the allocated space.
- We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.
- So, the replaced node 79 will now be a leaf node that can be easily deleted.



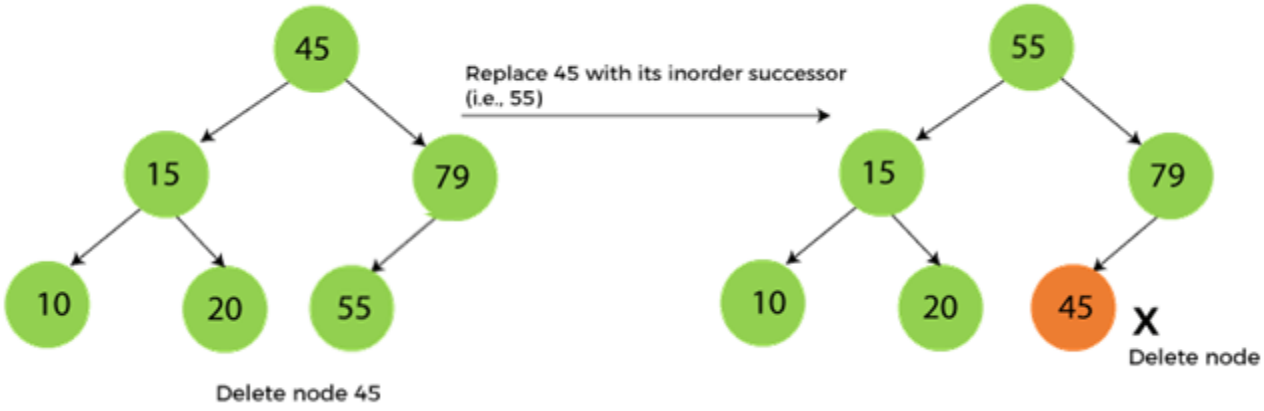
3. When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



The complexity of the Binary Search tree

Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

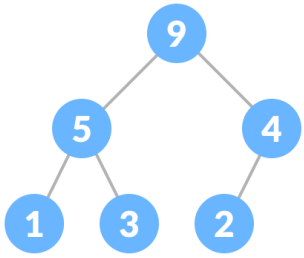
- The space complexity of all operations of Binary search tree is O(n).

Heap

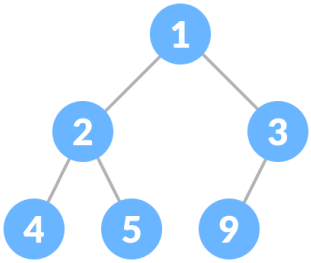
Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

Always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap property.**

Always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap property**.



Max-heap



Min-heap

Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

Heapify: Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

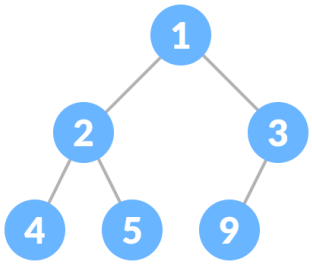
How can we arrange the nodes in the Tree?
There are two types of the heap:

- Min Heap
- Max heap

Min Heap: The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:
A[Parent(i)] <= A[i]
Let's understand the min-heap through an example.

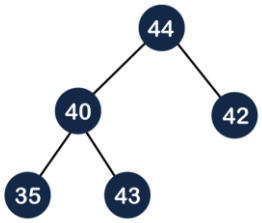


Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always log n; therefore, the time complexity would also be O(log n).

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

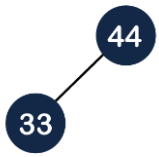
44, 33, 77, 11, 55, 88

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

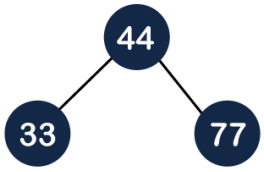
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

Step 1: First we add the 44 element in the tree as shown below:

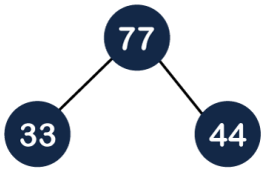
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



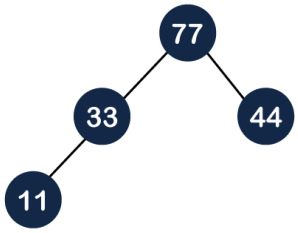
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



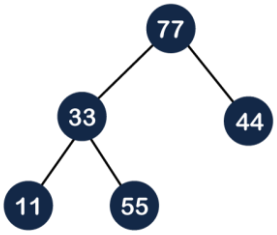
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



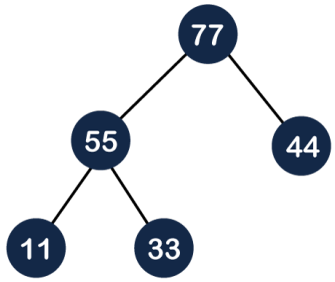
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



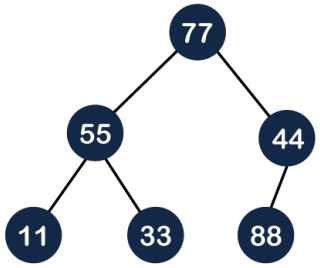
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



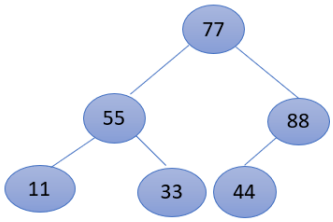
As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



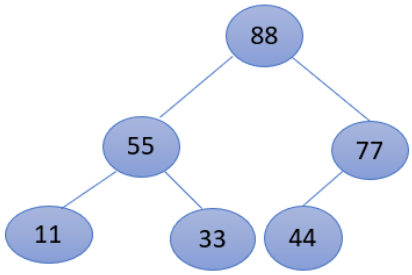
Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



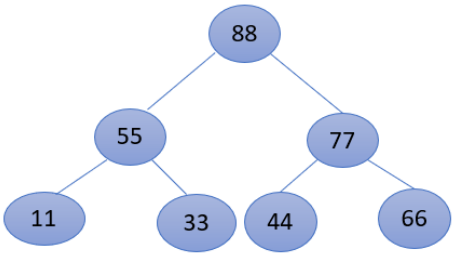
As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:



Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:



Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:



In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

Insertion Operation in Max Heap

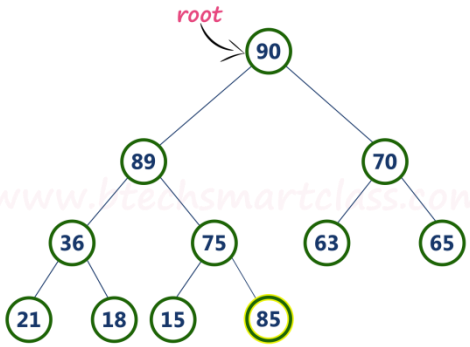
Insertion Operation in max heap is performed as follows...

- **Step 1 -** Insert the **newNode** as **last leaf** from left to right.
- **Step 2 -** Compare **newNode value** with its **Parent node**.
- **Step 3 -** If **newNode value is greater** than its parent, then **swap** both of them.
- **Step 4 -** Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

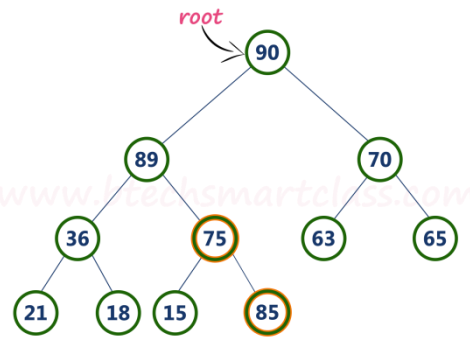
Example

Consider the above max heap. **Insert a new node with value 85.**

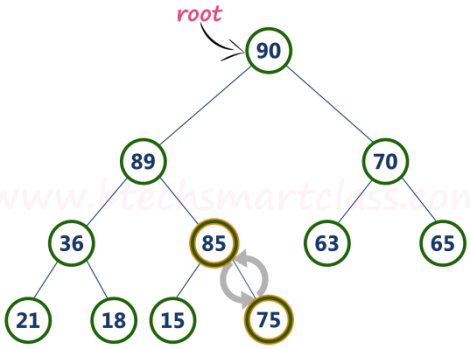
- **Step 1 -** Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



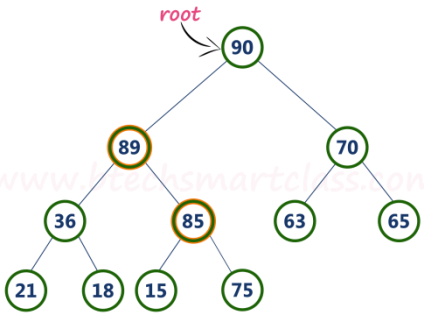
- **Step 2 -** Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



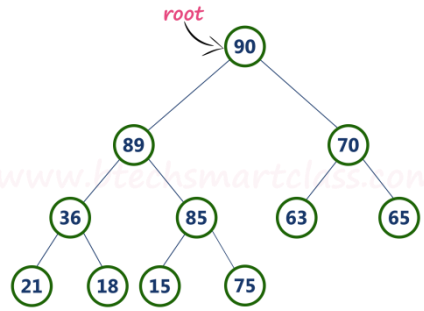
- **Step 3 -** Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After swapping, max heap is as follows...



- **Step 4 -** Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



Deletion Operation in Max Heap

In a max heap, deleting the last node is very simple as it does not disturb max heap properties.

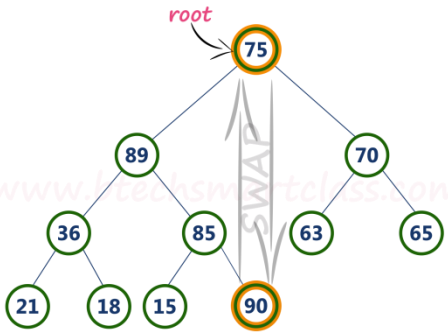
Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

- **Step 1 - Swap** the **root** node with **last** node in max heap
- **Step 2 - Delete** last node.
- **Step 3 -** Now, compare **root value** with its **left child value**.
- **Step 4 -** If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**
- **Step 5 -** If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.
- **Step 6 -** If **root value is larger** than its left child, then compare **root value** with its **right child** value.
- **Step 7 -** If **root value is smaller** than its **right child**, then **swap root** with **right child** otherwise **stop the process**.
- **Step 8 -** Repeat the same until root node fixes at its exact position.

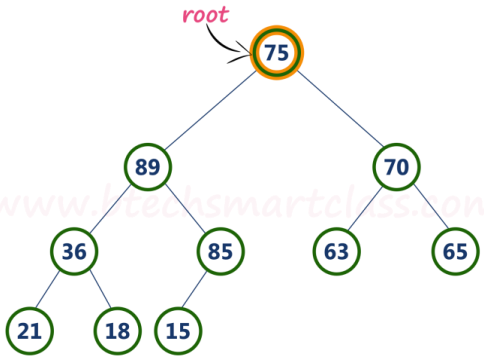
Example

Consider the above max heap. **Delete root node (90) from the max heap.**

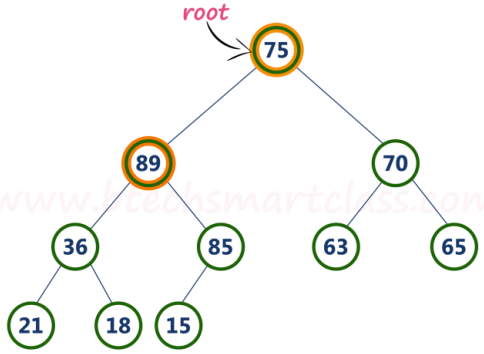
- **Step 1 - Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



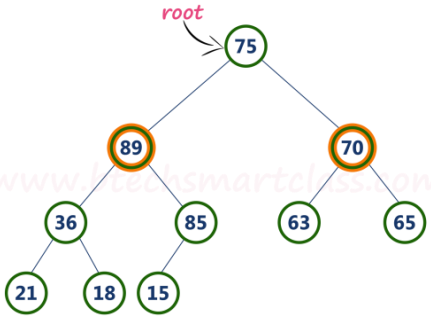
- **Step 2 - Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



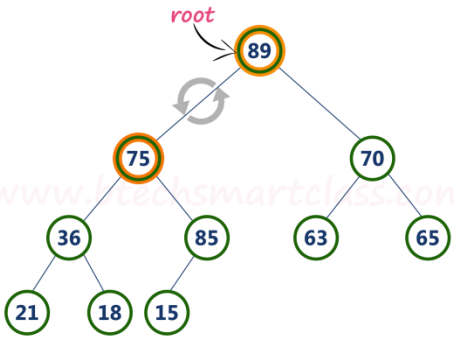
- **Step 3 -** Compare **root node (75)** with its **left child (89)**.



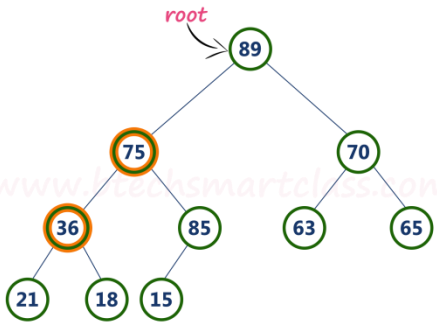
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



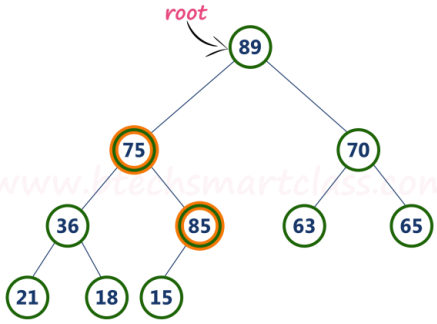
- **Step 4 -** Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



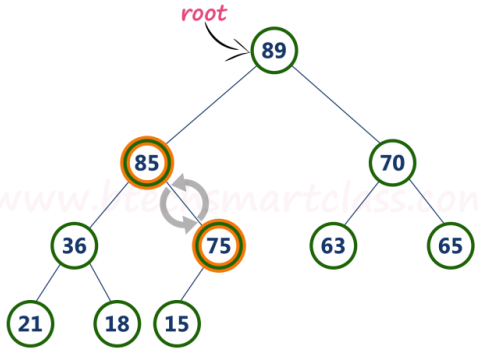
- **Step 5 -** Now, again compare **75** with its **left child (36)**.



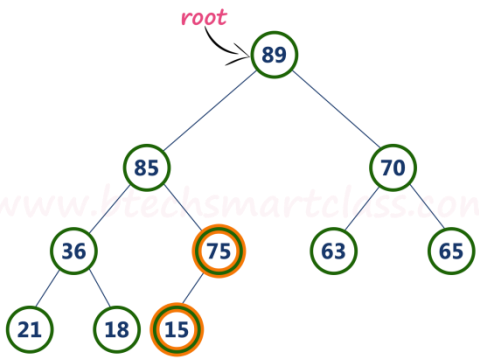
Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85**.



- **Step 6 -** Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...

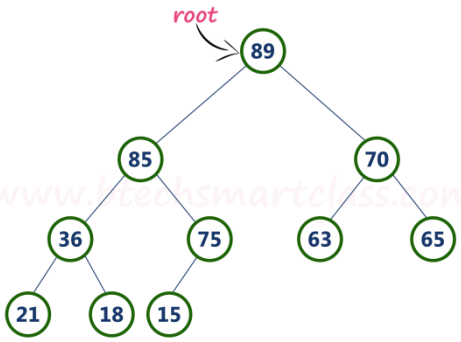


- **Step 7 -** Now, compare node with value **75** with its left child (**15**).



Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (**90**) is as follows...



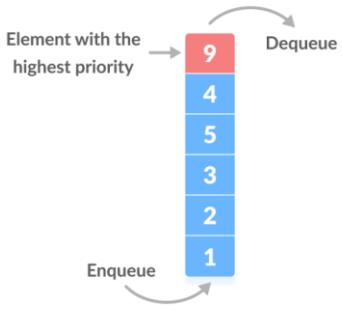
Priority queue:

A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
However, if elements with the same priority occur, they are served according to their order in the queue

Assigning Priority Value
Generally, the value of the element itself is considered for assigning the priority. For example,

The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.

We can also set priorities according to our needs.



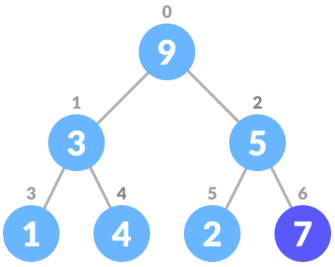
Difference between Priority Queue and Normal Queue
In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

Implementation of Priority Queue
Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues
Priority Queue Operations
Basic operations of a priority queue are inserting, removing, and peeking elements.

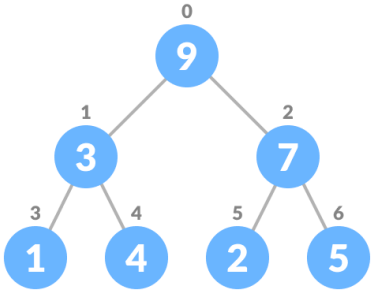
1. Inserting an Element into the Priority Queue

Inserting an element into a priority queue (max-heap) is done by the following steps.

Insert the new element at the end of the tree.



Heapify the tree



Algorithm for insertion of an element into priority queue (max-heap)

If there is no node,

create a newNode.

else (a node is already present)

insert the newNode at the end (last node from left to right.)

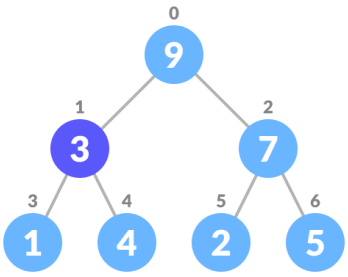
heapify the array

For Min Heap, the above algorithm is modified so that parentNode is always smaller than `newNode`.

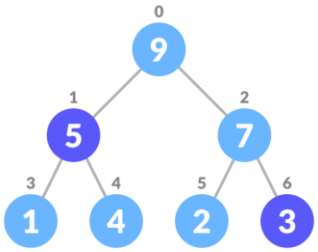
2. Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max-heap) is done as follows:

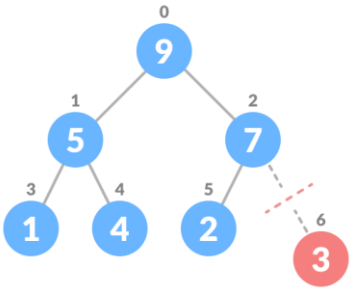
Select the element to be deleted.



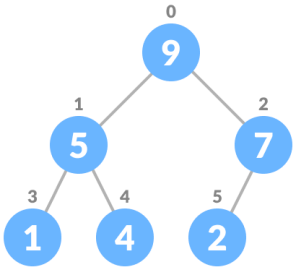
Swap it with the last element



Remove the last element



Heapify the tree



Algorithm for deletion of an element in the priority queue (max-heap)

If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode

remove noteToBeDeleted
heapify the array

3. Peeking from the Priority Queue (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

return rootNode

4. Extract-Max/Min from the Priority Queue

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum value after removing it from Min Heap.

Applications:

Heap sort:

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

Step by Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

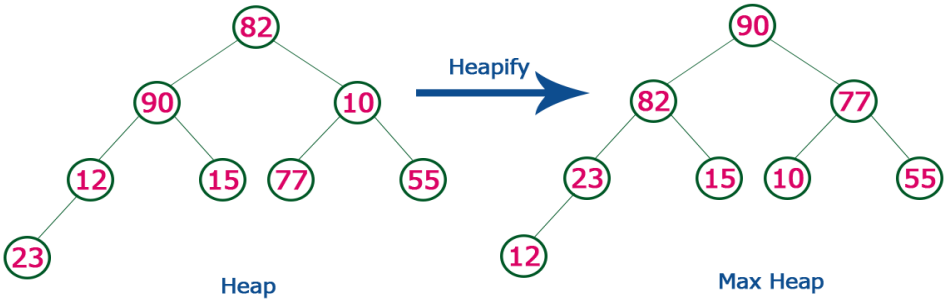
- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap.**
- **Step 3 -** Delete the root element from Min Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

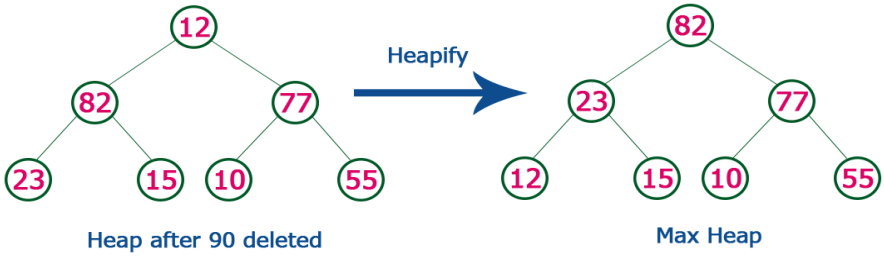
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

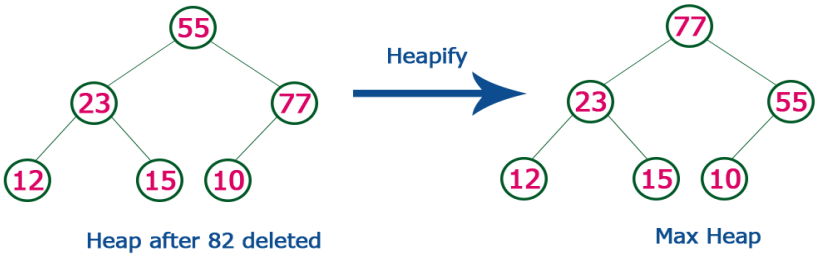
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, 90

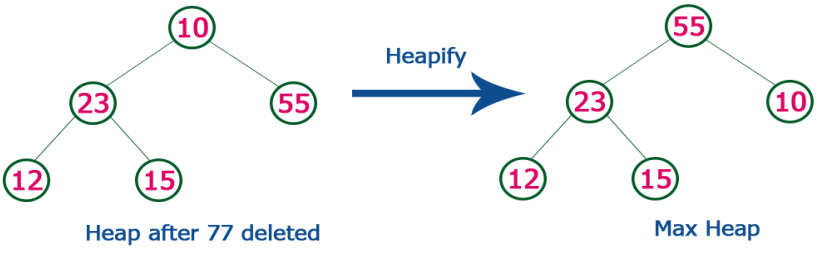
Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, 82, 90

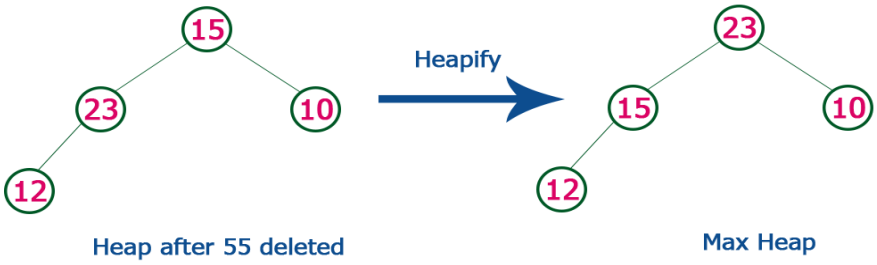
Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, 77, 82, 90

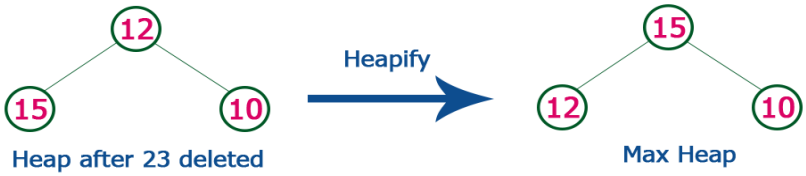
Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55, 77, 82, 90**

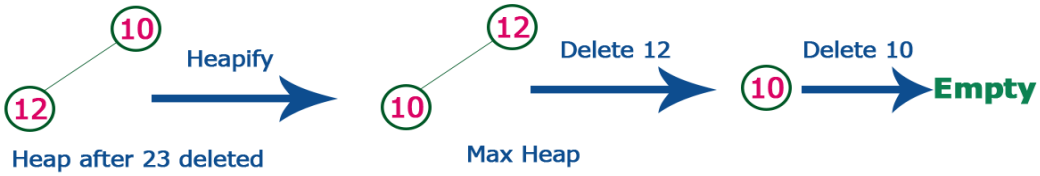
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Complexity of the Heap Sort Algorithm

To sort an unsorted list with '**n**' number of elements, following are the complexities...

Worst Case : O(n log n)
Best Case : O(n log n)
Average Case : O(n log n)

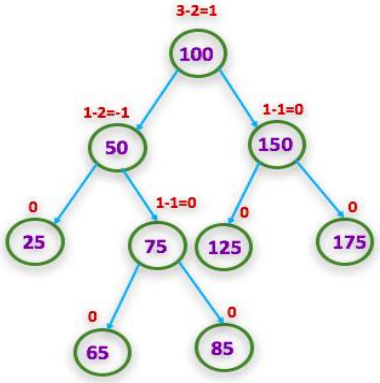
AVL TREES

- AVL stands for ”Adelson, Velski & Landis”
- AVL tree is a self-balanced binary search tree. That means, an AVL tree is a binary search tree but it is a balanced tree.
- A binary search 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 for every node, height of its children differs by at most one.
- In an AVL tree, every node maintains an extra information known as **balance factor** to take care of the self-balancing nature of the tree.

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 = height Of Left Subtree — height Of Right Subtree

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between the range of -1 to +1.

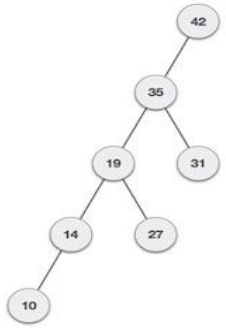


AVL Tree

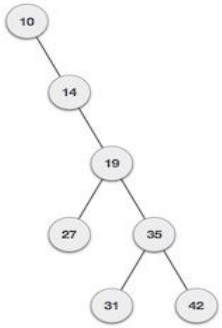
Note: Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

Why AVL Tree ?

- AVL tree controls the height of the binary search tree by not letting it to be skewed.
- The time taken for all operations in a binary search tree of height h is O(h). However, it can be extended to O(n) if the BST becomes skewed (i.e. worst case).



If input 'appears' non-increasing manner



If input 'appears' in non-decreasing manner

- By limiting this height to log n, AVL tree imposes an upper bound on each operation to be O(log n) where n is the number of nodes.

AVL Tree Rotations?

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

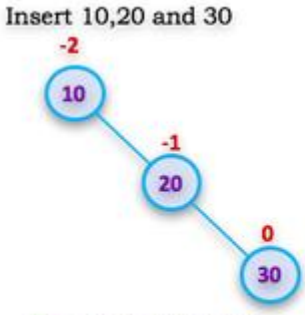
Rotation is the process of moving the nodes to either left or right to make tree balanced in terms of its height.

To balance itself, an AVL tree may perform the following four kinds of rotations −

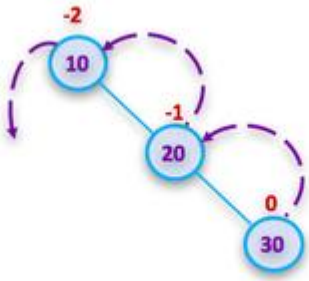
1. **Left rotation (Single)-LL**
2. **Right rotation (Single)-RR**
3. **Left-Right rotation (Double)-LR**
4. **Right-Left rotation (Double)-RL**

1. Left Rotation (Single LL):

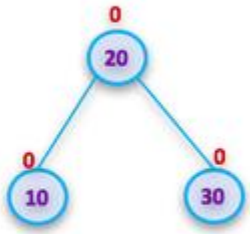
In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree…



Tree is imbalanced



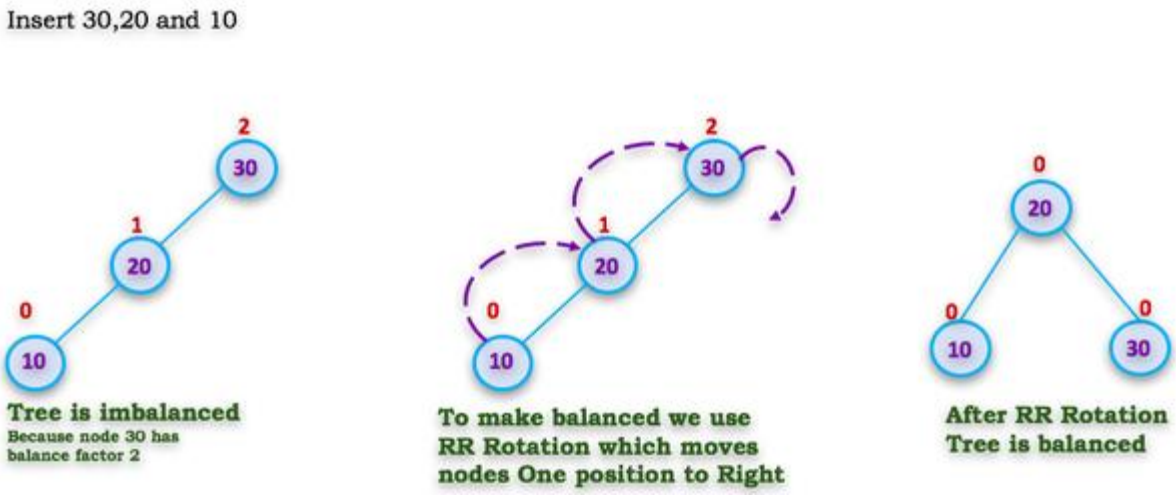
To make balanced we use LL Rotation which moves nodes One position to left



After LL Rotation Tree is balanced

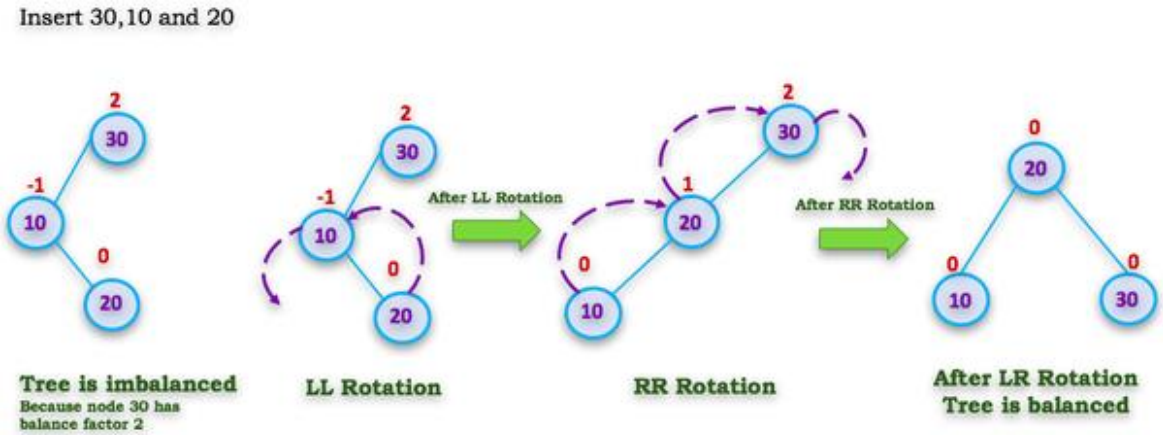
2. Right Rotation (Single RR)

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



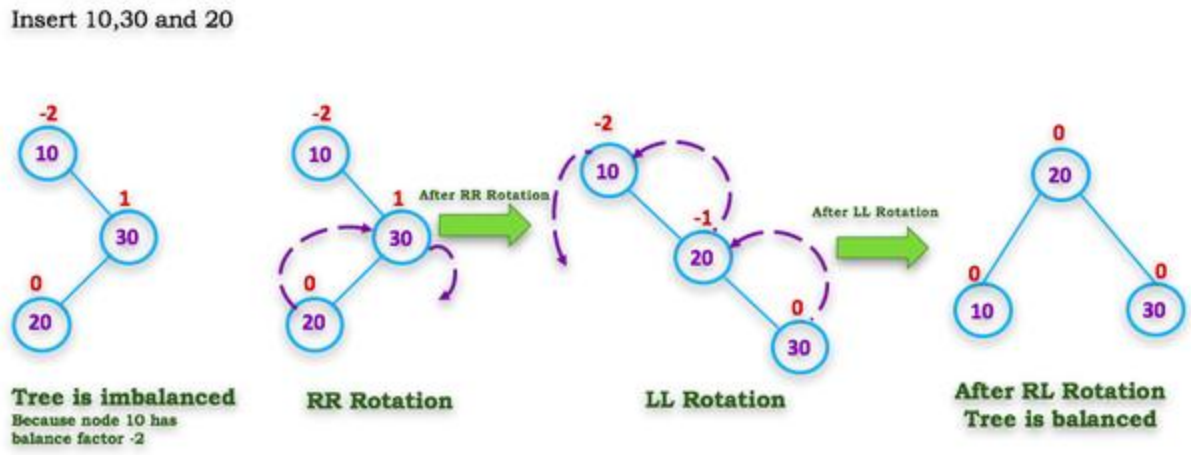
3.Left Right Rotation (Double — LR Rotation)

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



4. Right Left Rotation (Double — RL Rotation)

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



Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property.

1. Insertion
2. Deletion
3. Search
4. Display

Insertion:

Insertion Operation is performed to insert an element in the AVL Tree.

To insert an element in the AVL tree, follow the following steps-

- Insert the element in the AVL tree in the same way the insertion is performed in BST.
- After insertion, check the balance factor of each node of the resulting tree.

Now, following two cases are possible-

Case-01:

- After the insertion, the balance factor of each node is either 0 or 1 or -1.
- In this case, the tree is considered to be balanced.
- Conclude the operation.
- Insert the next element if any.

Case-02:

- After the insertion, the balance factor of at least one node is not 0 or 1 or -1.
- In this case, the tree is considered to be imbalanced.
- Perform the suitable rotation to balance the tree.
- After the tree is balanced, insert the next element if any.

Rules To Remember-

Rule-01:

After inserting an element in the existing AVL tree,

- Balance factor of only those nodes will be affected that lies on the path from the newly inserted node to the root node.

Rule-02:

To check whether the AVL tree is still balanced or not after the insertion,

- There is no need to check the balance factor of every node.
- Check the balance factor of only those nodes that lies on the path from the newly inserted node to the root node.

Rule-03:

After inserting an element in the AVL tree,

- If tree becomes imbalanced, then there exists one particular node in the tree by balancing which the entire tree becomes balanced automatically.
- To re balance the tree, balance that particular node.

To find that particular node,

- Traverse the path from the newly inserted node to the root node.
- Check the balance factor of each node that is encountered while traversing the path.
- The first encountered imbalanced node will be the node that needs to be balanced.

To balance that node,

- Count three nodes in the direction of leaf node.
- Then, use the concept of AVL tree rotations to re balance the tree.

Algorithm for insertion:

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.

Problem-

Construct AVL Tree for the following sequence of numbers-

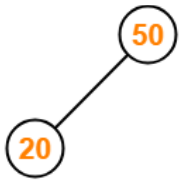
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

Solution-

Step-01: Insert 50

Step-02: Insert 20

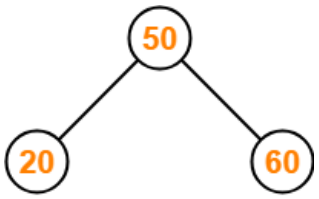
As 20 < 50, so insert 20 in 50's left sub tree.



Tree is Balanced

Step-03: Insert 60

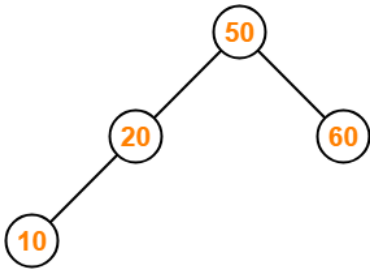
As 60 > 50, so insert 60 in 50's right sub tree.



Tree is Balanced

Step-04: Insert 10

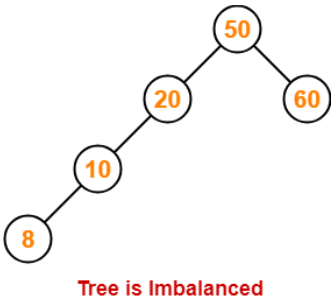
- As 10 < 50, so insert 10 in 50's left sub tree.
- As 10 < 20, so insert 10 in 20's left sub tree.



Tree is Balanced

Step-05: Insert 8

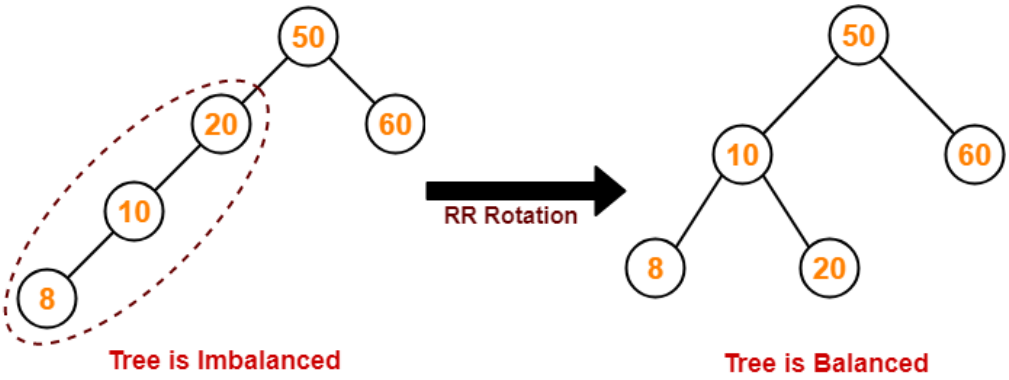
- As 8 < 50, so insert 8 in 50's left sub tree.
- As 8 < 20, so insert 8 in 20's left sub tree.
- As 8 < 10, so insert 8 in 10's left sub tree.



To balance the tree,

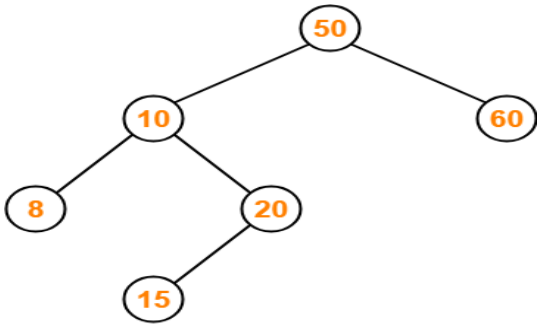
- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Step-06: Insert 15

- As 15 < 50, so insert 15 in 50's left sub tree.
- As 15 > 10, so insert 15 in 10's right sub tree.
- As 15 < 20, so insert 15 in 20's left sub tree.

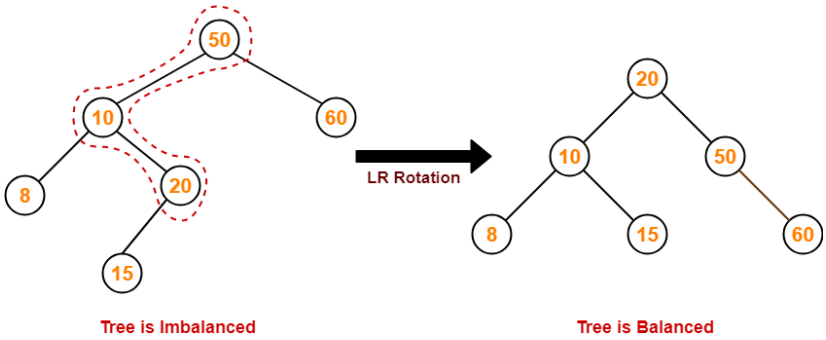


Tree is Imbalanced

To balance the tree,

- Find the first imbalanced node on the path from the newly inserted node (node 15) to the root node.
- The first imbalanced node is node 50.
- Now, count three nodes from node 50 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-

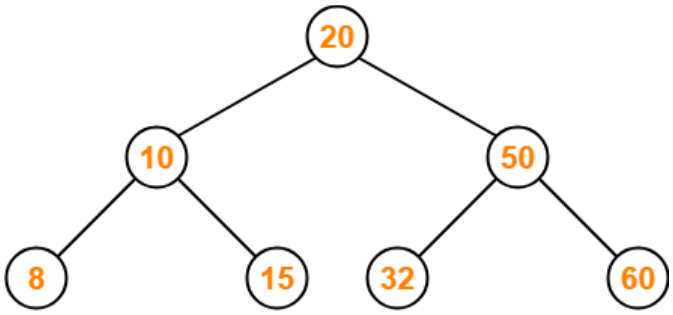


Tree is Imbalanced

Tree is Balanced

Step-07: Insert 32

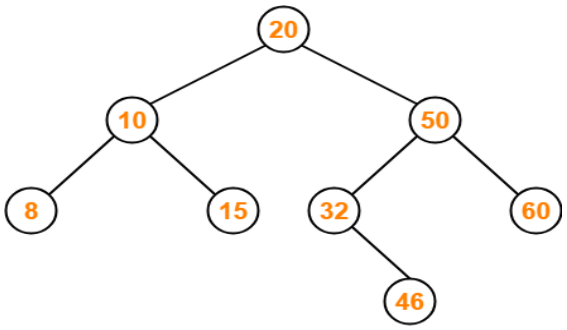
- As 32 > 20, so insert 32 in 20's right sub tree.
- As 32 < 50, so insert 32 in 50's left sub tree.



Tree is Balanced

Step-08: Insert 46

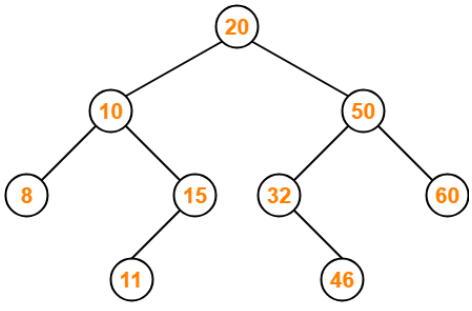
- As 46 > 20, so insert 46 in 20's right sub tree.
- As 46 < 50, so insert 46 in 50's left sub tree.
- As 46 > 32, so insert 46 in 32's right sub tree.



Tree is Balanced

Step-09: Insert 11

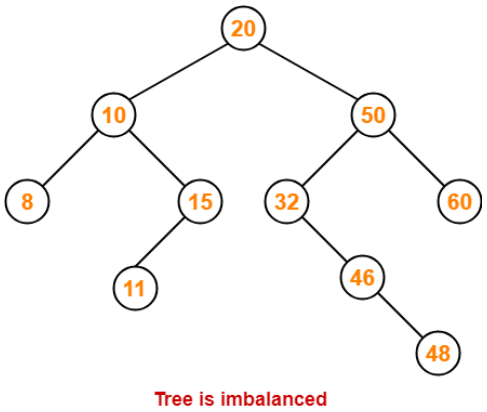
- As 11 < 20, so insert 11 in 20's left sub tree.
- As 11 > 10, so insert 11 in 10's right sub tree.
- As 11 < 15, so insert 11 in 15's left sub tree.



Tree is Balanced

Step-10: Insert 48

- As 48 > 20, so insert 48 in 20's right sub tree.
- As 48 < 50, so insert 48 in 50's left sub tree.
- As 48 > 32, so insert 48 in 32's right sub tree.
- As 48 > 46, so insert 48 in 46's right sub tree.

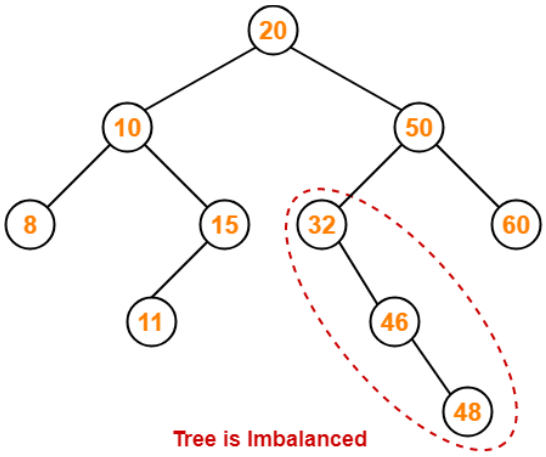


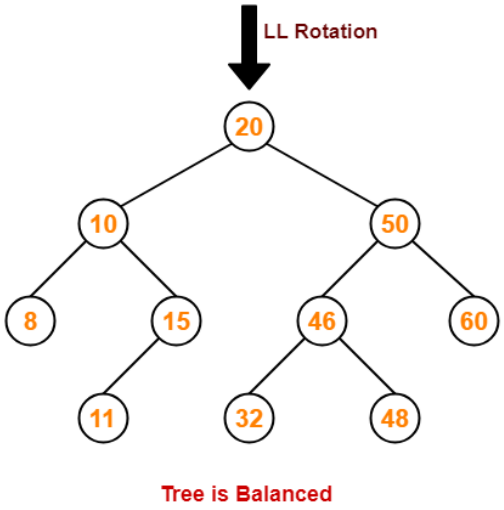
Tree is imbalanced

To balance the tree,

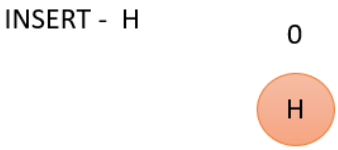
- Find the first imbalanced node on the path from the newly inserted node (node 48) to the root node.
- The first imbalanced node is node 32.
- Now, count three nodes from node 32 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



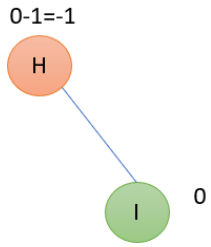


Example 2:
Construct AVL tree for the following elements : H, I, J, B, A, E, C, F, D, G, K, L
Step 1:



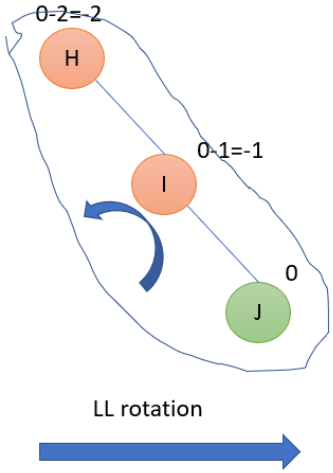
Step 2:

INSERT - I



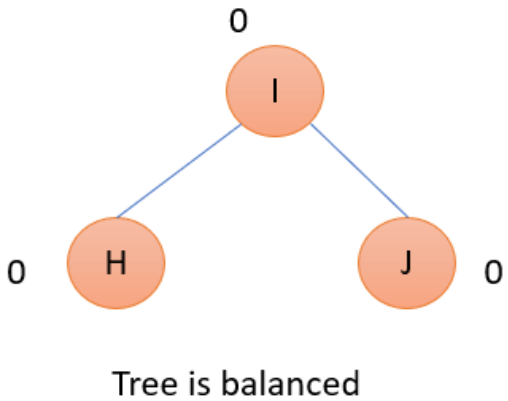
Step 3:

INSERT - J

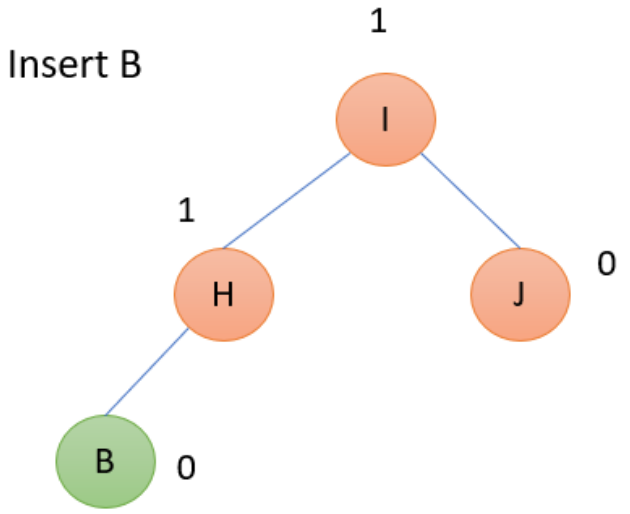


Here at node H the balance factor is 2 so the tree is imbalanced. To balance the given tree we need to use appropriate rotation. For the above tree we have to use LL rotation

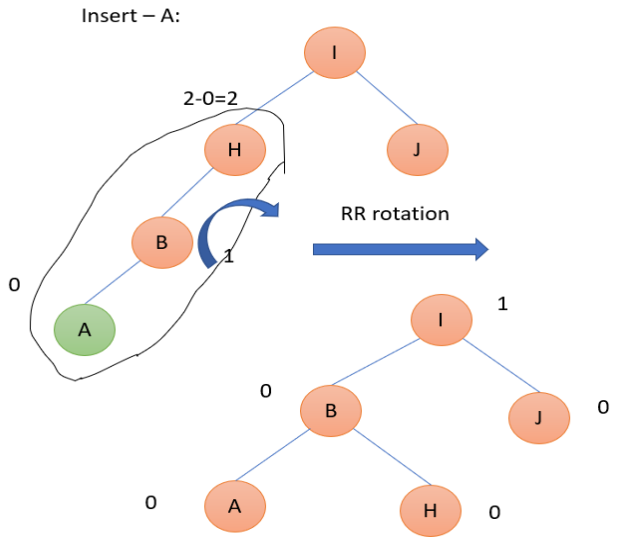
Step 4:



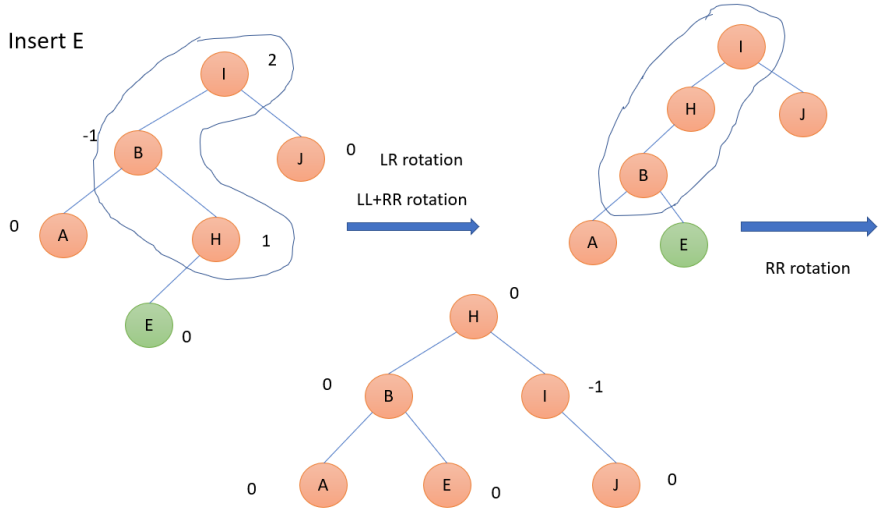
Step 5:



Step 6:

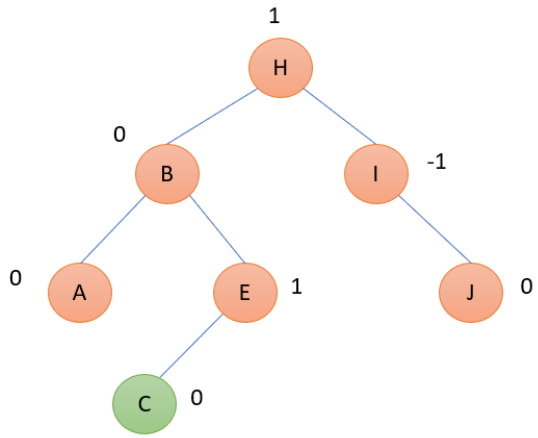


Step 7:



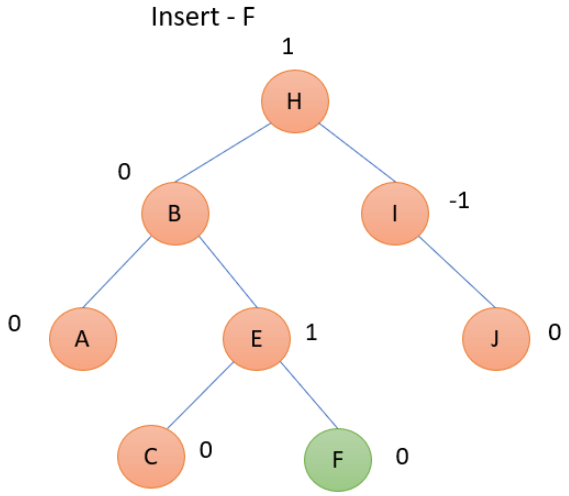
Step 8:

Insert - C



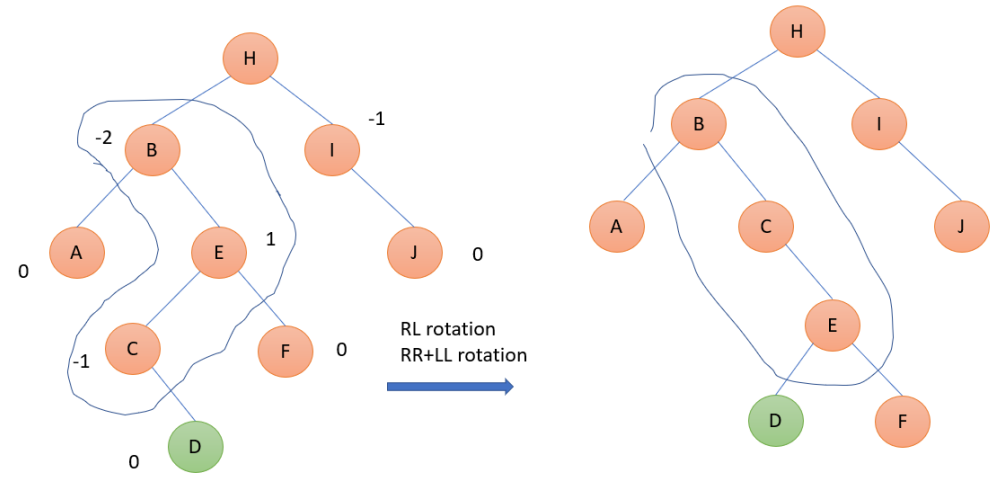
Step 9:

Insert - F

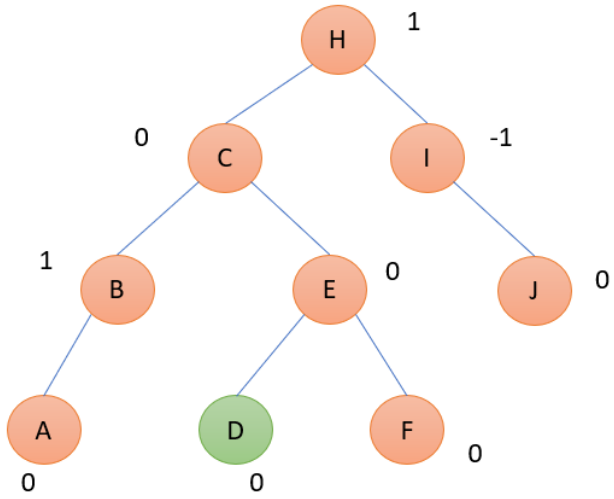


Step 10:

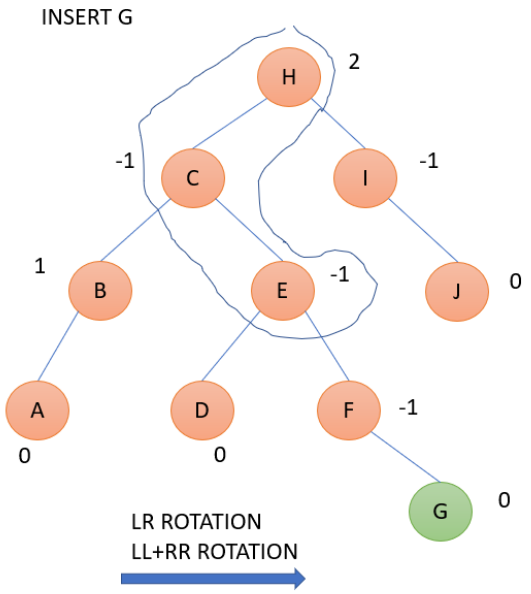
Insert D



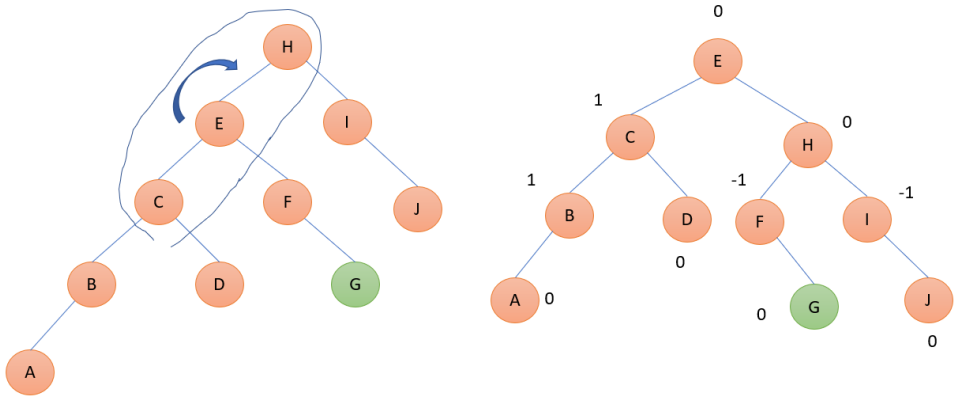
Step 11:



Step 12:

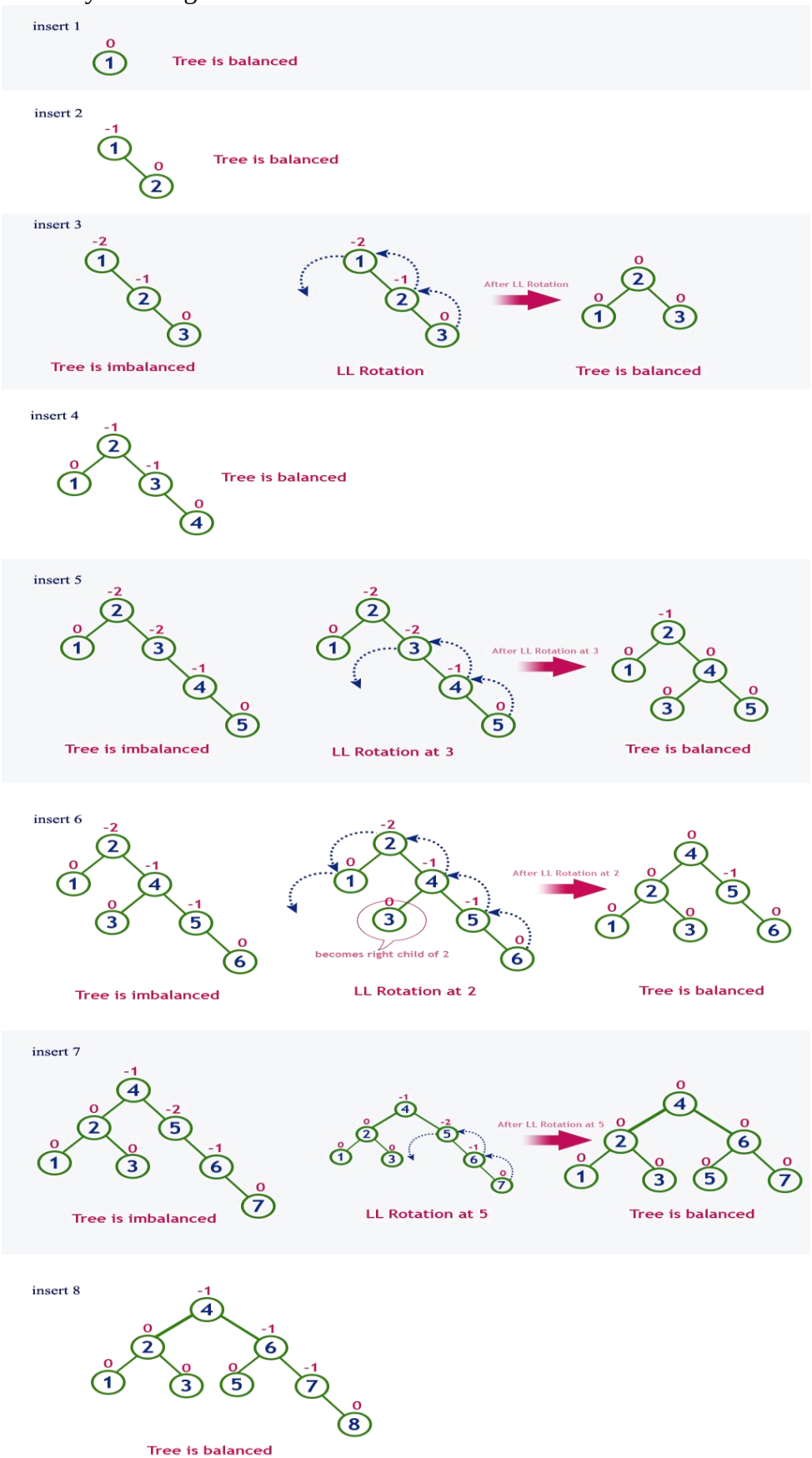


Step 13:



FINAL AVL TREE WITH BALANCING

Example 3:
Construct an AVL tree by inserting numbers from 1 to 8

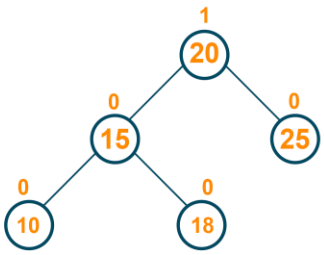


Deletion operation:
The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

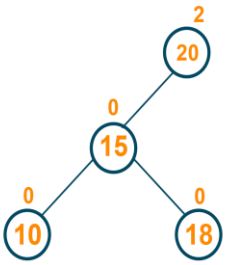
The algorithm steps of deletion operation in an AVL tree are:

1. Locate the node to be deleted
2. If the node does not have any child, then remove the node
3. If the node has one child node, replace the content of the deletion node with the child node and remove the node
4. If the node has two children nodes, find the in-order successor node ‘k' which has no child node and replace the contents of the deletion node with the ‘k’ followed by removing the node.
5. Update the balance factor of the AVL tree

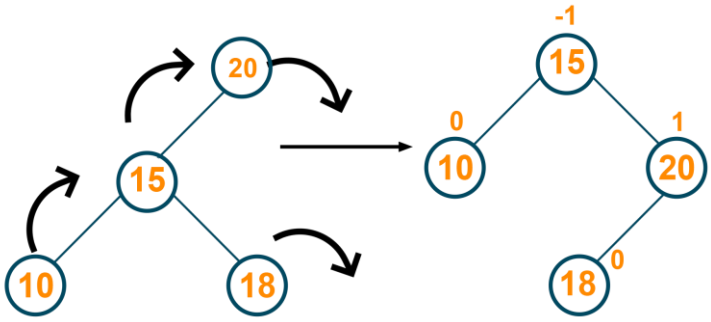
Example: **Delete node with zero child**
Let us consider the below AVL tree with the given balance factor as shown in the figure below



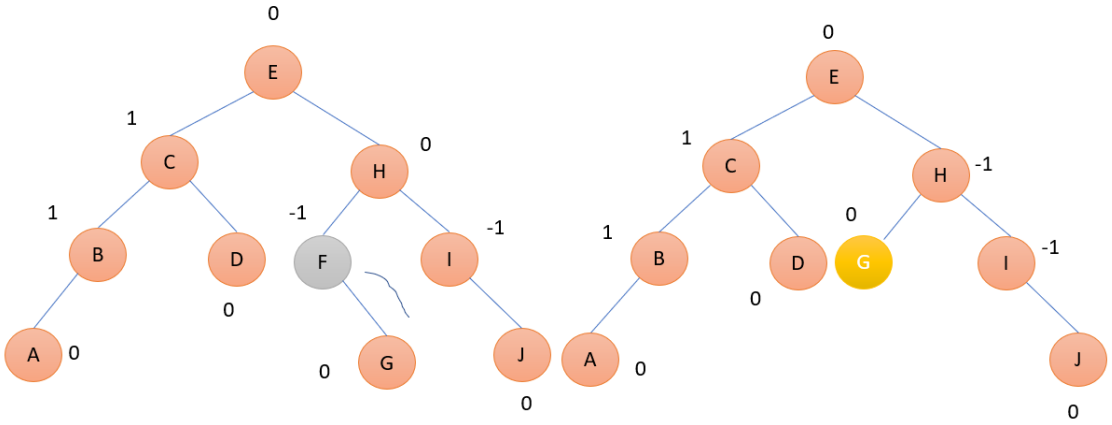
Here, we have to delete the node '25' from the tree. As the node to be deleted does not have any child node, we will simply remove the node from the tree



After removal of the tree, the balance factor of the tree is changed and therefore, the rotation is performed to restore the balance factor of the tree and create the perfectly balanced tree

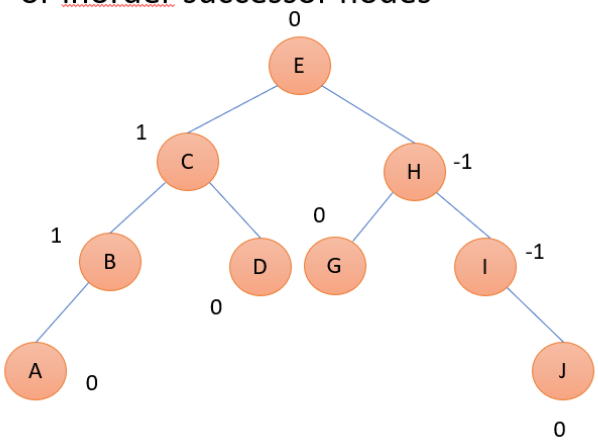


Example : **Delete node with single child**



Example: **Delete node with 2 child nodes**

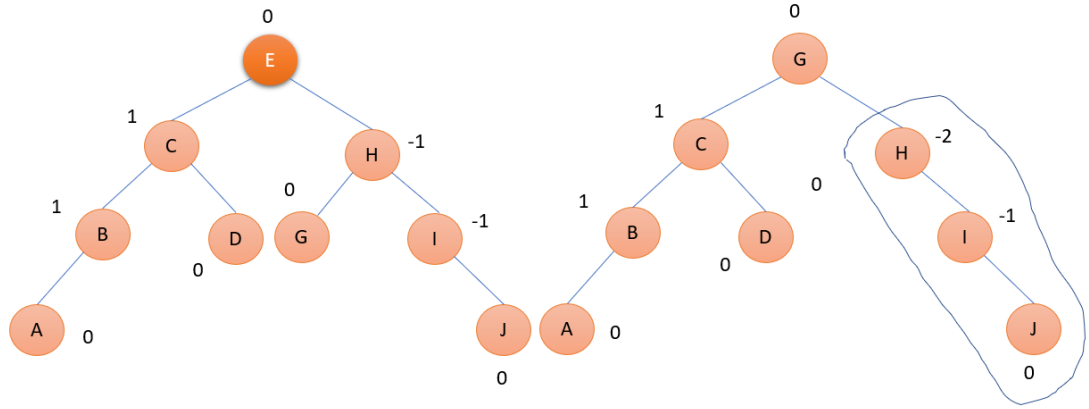
- To delete node with 2 childs, we have to consider inorder predecessor or inorder successor nodes

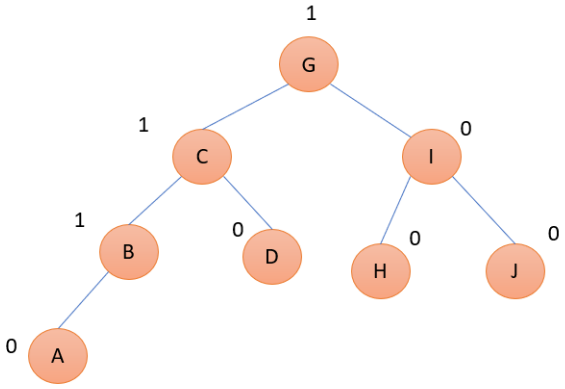


Inorder : A, B, C, D, E, G, H, I, J

Delete node : E
Replace E with inorder predecessor (D) or inorder successor (G)

Replace E with G





After deleting node E the final balanced binary search tree

Search operation:

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.

Applications of AVL trees:

- AVL trees are mostly used for in-memory sorts of sets and dictionaries.
  - AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
- It is used in applications that require improved searching apart from the database applications.