MODULE 5: TREES

5.1 Definitions and Properties

There are many basic data structures that can be used to solve application problems. Array is a good static data structure that can be accessed randomly and is fairly easy to implement. Linked Lists on the other hand is dynamic and is ideal for application that requires frequent operations such as add, delete, and update. One drawback of linked list is that data access is sequential.

Arrays and Linked Lists are **linear structures** and therefore the time required to search a "linear" list is proportional to the size of the data set.

We can extend the concept of linked data structure (linked list, stack, queue) to a structure that may have multiple relations among its nodes. Such a structure is called a **tree**.

A tree is a collection of nodes connected by directed (or undirected) edges.

Definition of Tree:

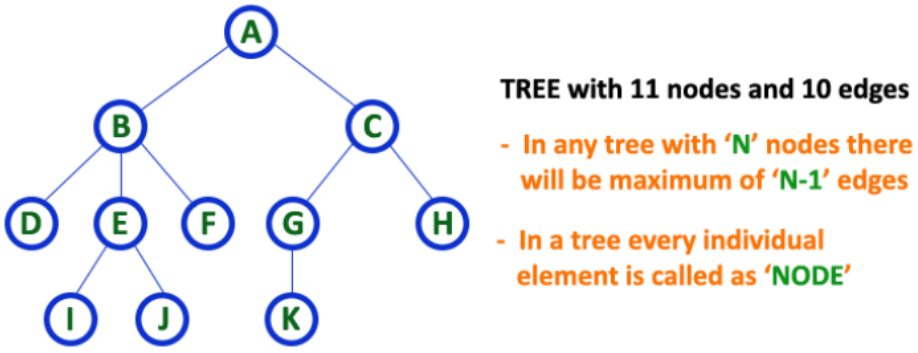
A tree is a ***nonlinear*** data structure, compared to arrays, linked lists, stacks and queues which are linear data structures.

A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more subtrees

Properties of a Tree

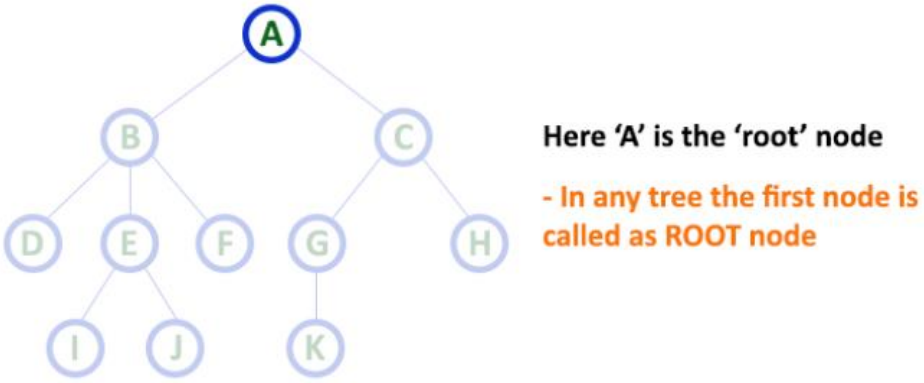
- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent -> children*

5.1.1 Terminology



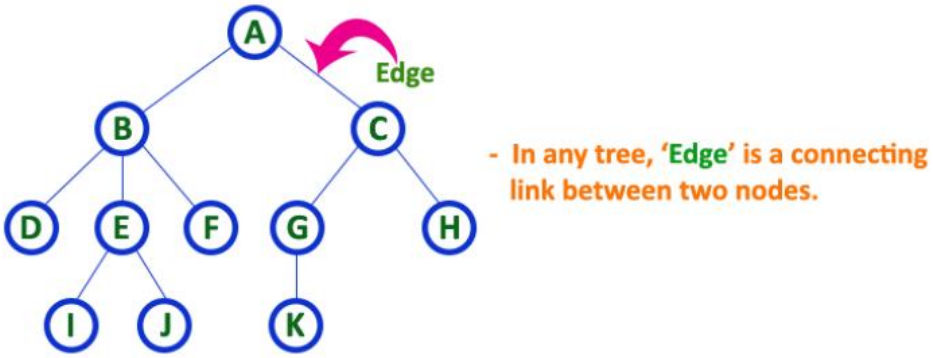
1. Root

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



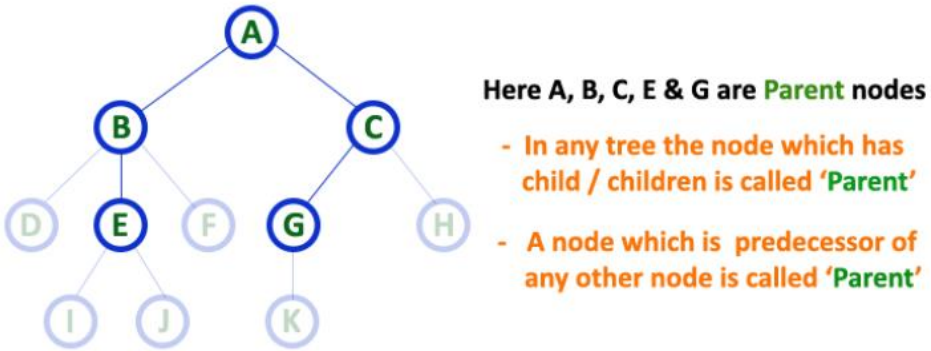
2. Edge

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



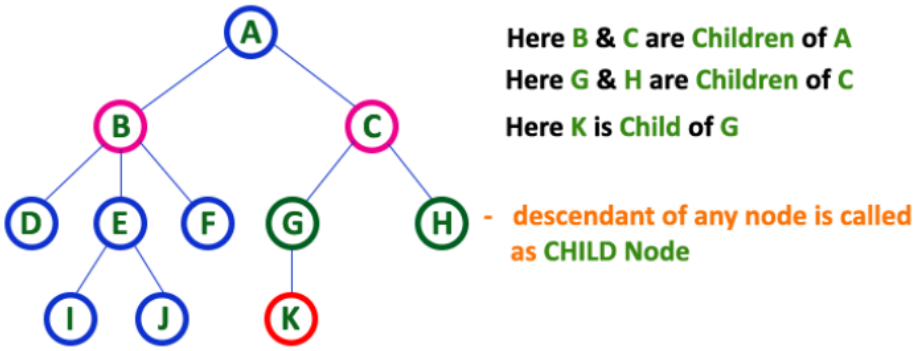
3. Parent

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



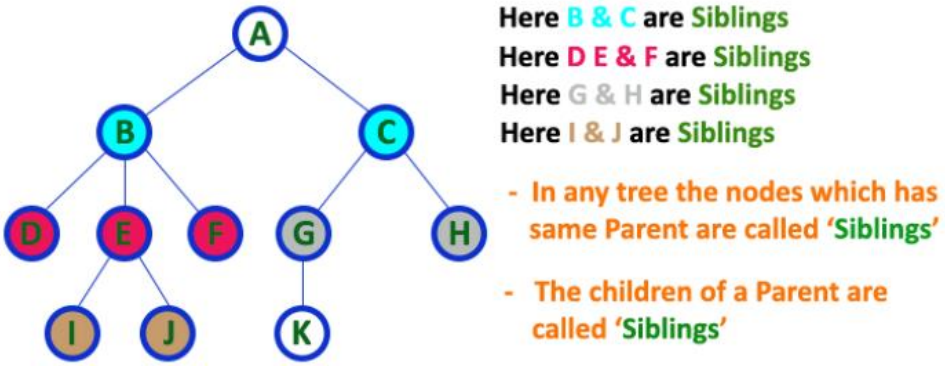
4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



5. Siblings

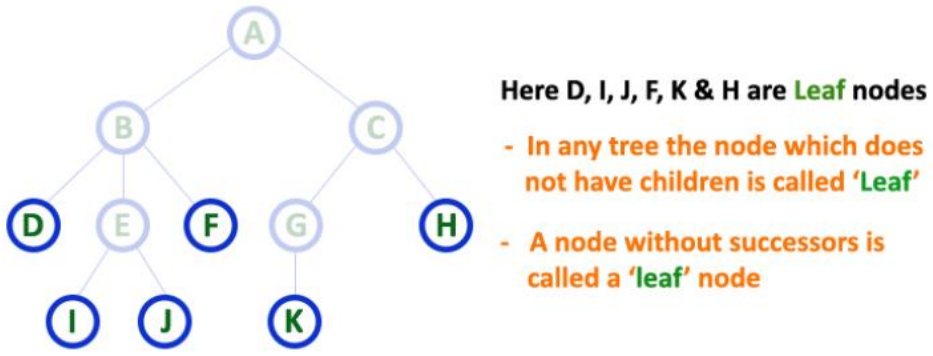
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

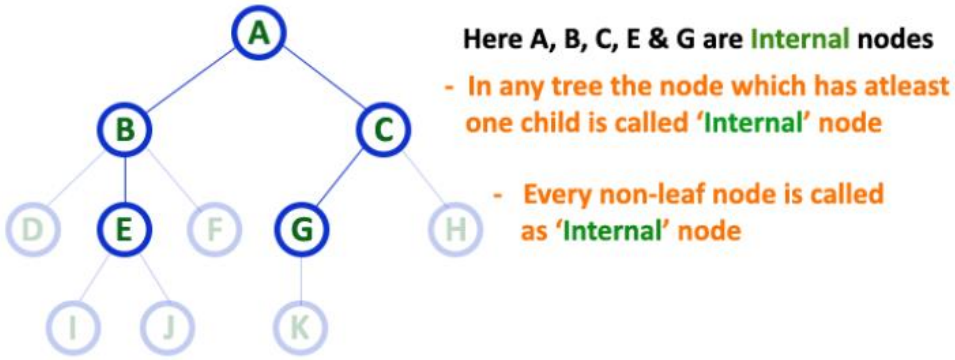
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as **'Terminal' node**.



7. Internal Nodes

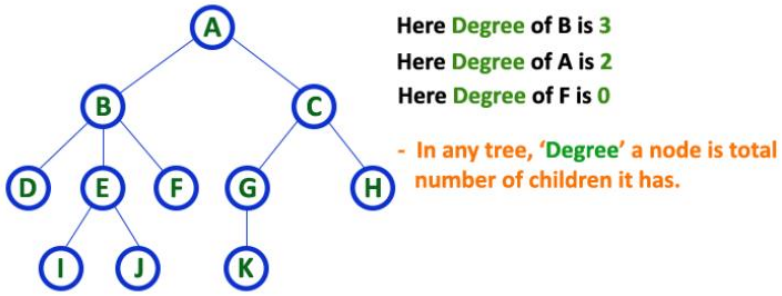
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as 'Non-**Terminal' nodes**.



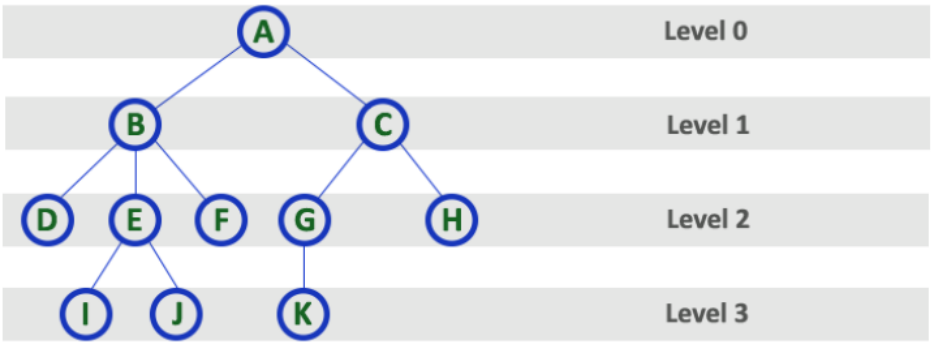
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as **'Degree of Tree'**



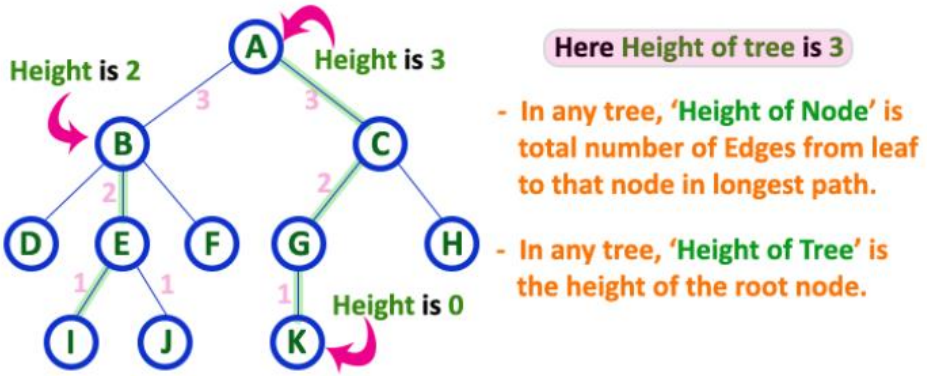
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



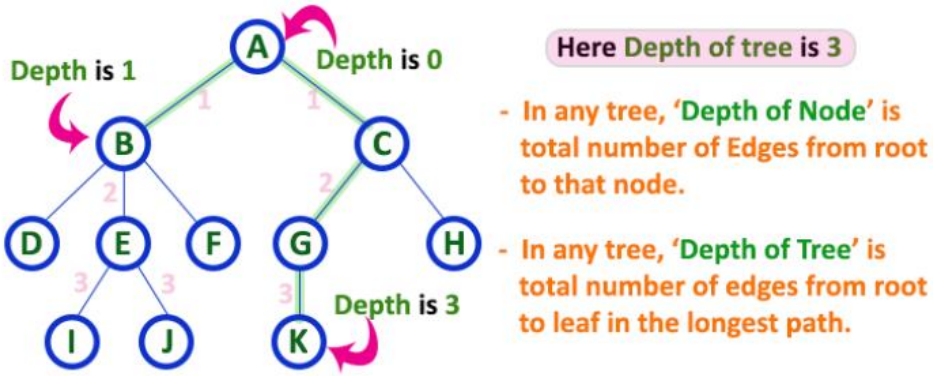
10. Height

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



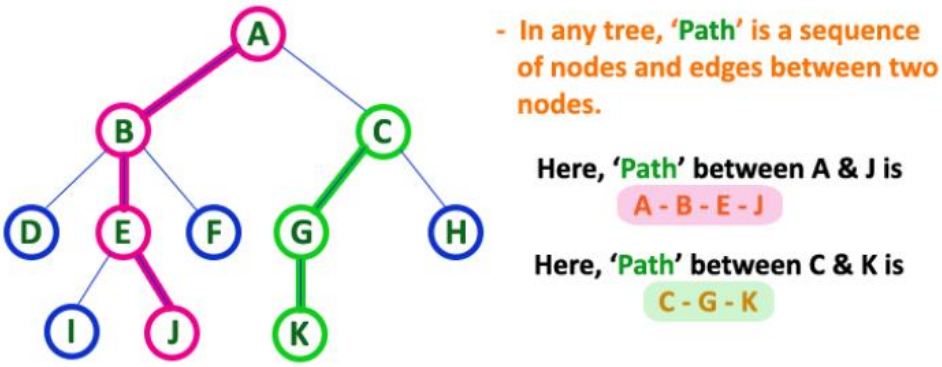
11. Depth

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



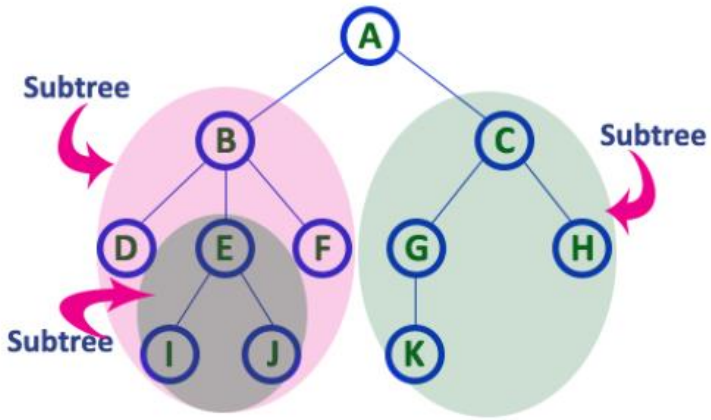
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example the path **A - B - E - J has length 4**.

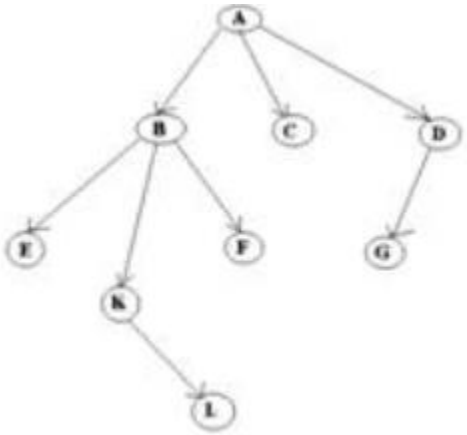


13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Example



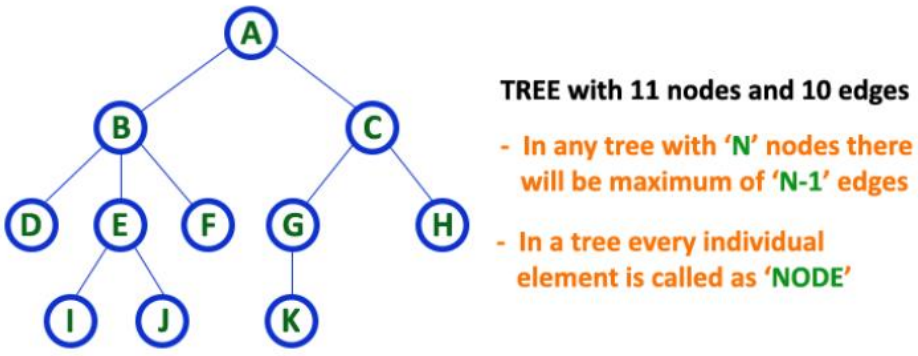
- In the above picture, the root has 3 subtrees.
- Each node can have arbitrary number of children. Nodes with no children are called **leaves, or external** nodes.
- In the above picture, C, E, F, L, G are leaves. Nodes, which are not leaves, are called **internal** nodes. Internal nodes have at least one child.
- Nodes with the same parent are called siblings. In the picture, B, C, D are called siblings.
- The **depth of a node** is the number of edges from the root to the node. The depth of K is 2.
- The **height of a node** is the number of edges from the node to the deepest leaf. The height of B is 2.
- The **height of a tree** is a height of a root.

5.1.2 Tree Representations

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

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

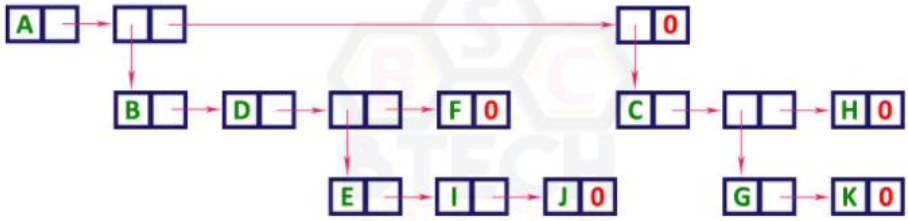
Consider the following tree.



1. List Representation

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

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

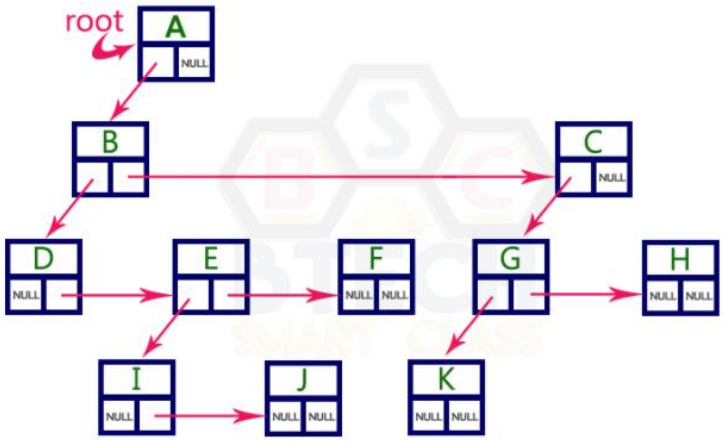


2. Left Child - Right Sibling Representation

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

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

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



5.2 Binary Trees

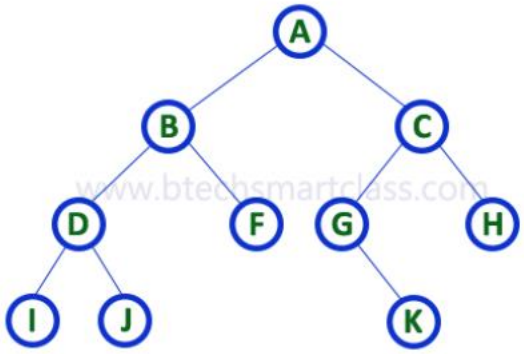
In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

Definition:

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

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

Example



There are different types of binary trees and they are.

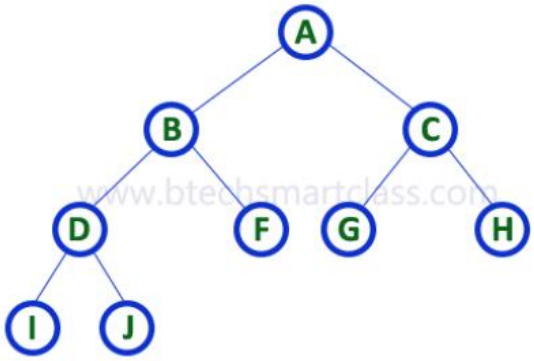
1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children.

Definition:

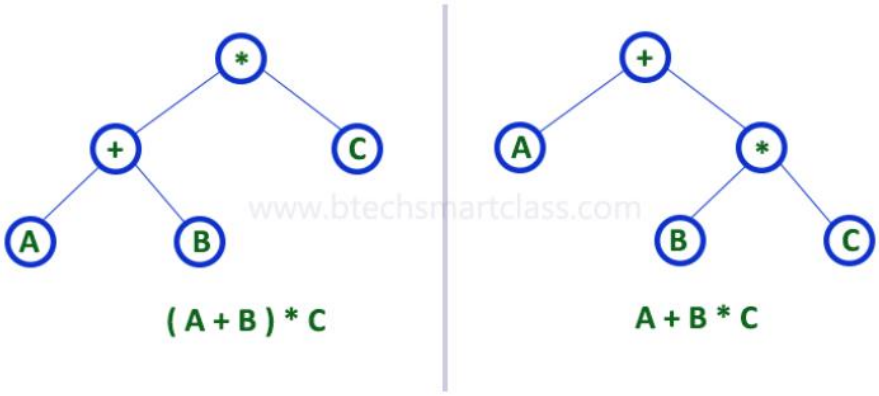
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

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



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

Example



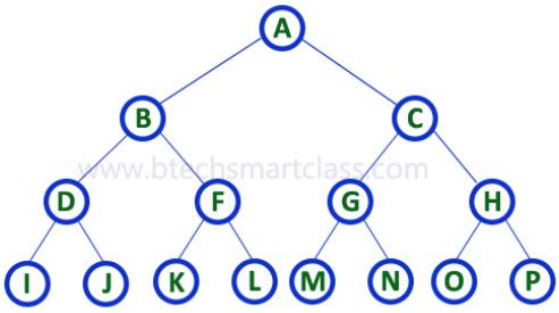
2. Complete Binary Tree

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

Definition:

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

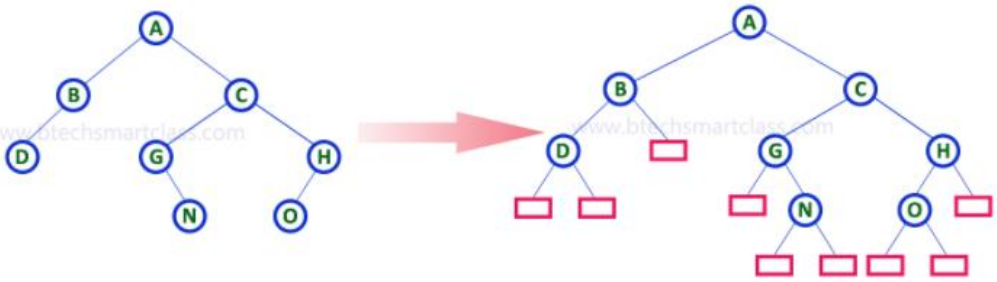


3. Extended Binary Tree

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

Definition:

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 color).

5.2.1 Properties of Binary Tree

1) The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}.

Here level is number of nodes on path from root to the node (including root and node). Level of root is 1. This can be proved by induction.

For root, l = 1, number of nodes = $2^{1-1} = 1$

Assume that maximum number of nodes on level l is 2^{l-1}

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$

2) Maximum number of nodes in a binary tree of height 'h' is $2^h – 1$.

Here height of a tree is maximum number of nodes on root to leaf path. Height of a tree with single node is considered as 1.

This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + ... + 2^h-1$. This is a simple geometric series with h terms and sum of this series is $2^h – 1$.

In some books, height of a leaf is considered as 0. In this way, the above formula becomes $2^{h+1} – 1$

3) In a Binary Tree with N nodes, min possible height or minimum number of levels is ⌈Log2(N+1)⌉

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes ⌈ Log2(N+1) ⌉ – 1

4) A Binary Tree with L leaves has at least ⌈Log2L ⌉+ 1 levels

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

L <= 2^{l-1} [From Point 1]

l = ⌈ Log2L ⌉ + 1

where l is the minimum number of levels.

5) In Binary tree, number of leaf nodes is always one more than nodes with two children.

L = T + 1

Where L = Number of leaf nodes

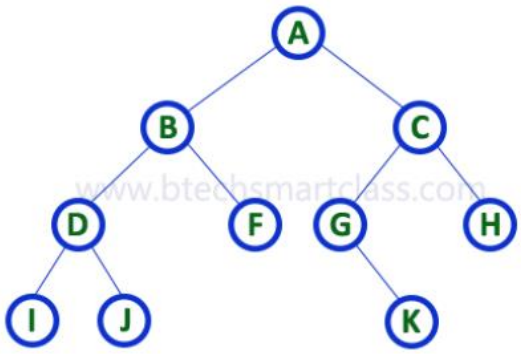
T = Number of internal nodes with two children

5.2.2 Representations 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

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

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

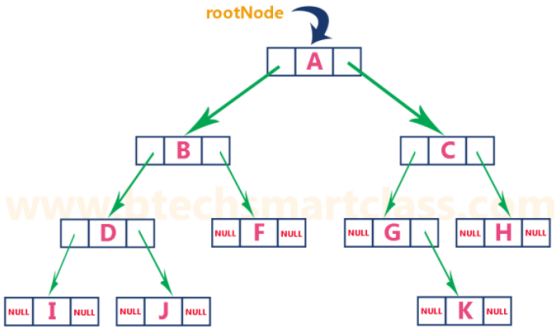
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2^{n+1} - 1.

2. Linked List Representation

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

In this linked list representation, a node has the following structure...

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



5.2.3 Operations/Traversals of Binary Trees

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

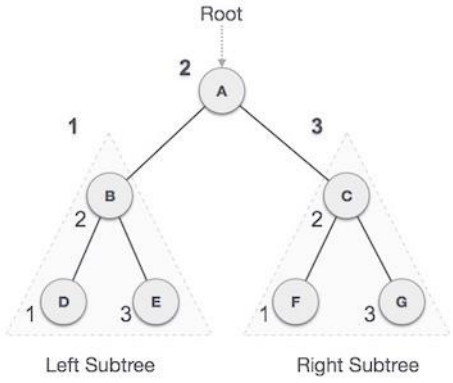
1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal
4. Iterative In-order Traversal
5. Level-Order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

1. In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −

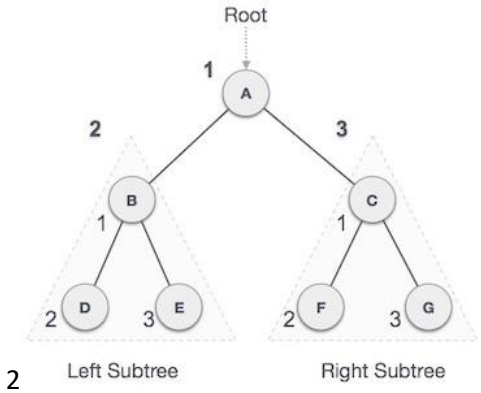
Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

2. Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



2

We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −

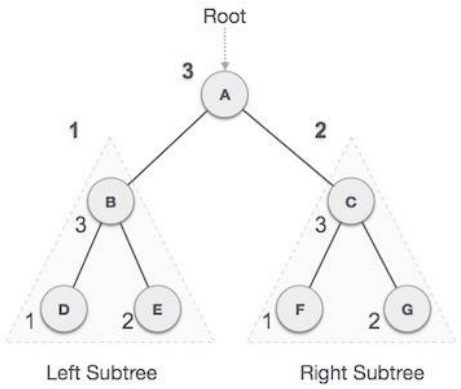
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

3. Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

4. Iterative Inorder Traversal

Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack.

Algorithm:

1) Create an empty stack S.

2) Initialize current node as root

3) Push the current node to S and set current = current->left until current is NULL

4) If current is NULL and stack is not empty then

   a) Pop the top item from stack.

   b) Print the popped item, set current = popped_item->right

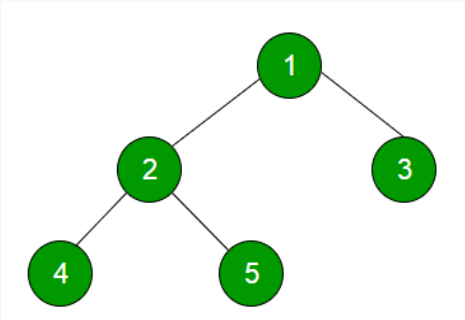
   c) Go to step 3.

5) If current is NULL and stack is empty then we are done.

5. Level-Order Traversal

Level order traversal of a tree is breadth first traversal for the tree.

In level order traversal, we visit the nodes level by level from left to right.



Level order traversal of the above tree is 1 2 3 4 5.

C program for preorder, postorder and inorder traversal

```
// C program for different tree traversals
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Given a binary tree, print its nodes according to the
  "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
    if (node == NULL)
      return;
```

```
    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    printf("%d ", node->data);

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root  = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right  = newNode(5);
```

```
    printf("\nPreorder traversal of binary tree is \n");
    printPreorder(root);
    printf("\nInorder traversal of binary tree is \n");
   printInorder(root);
   printf("\nPostorder traversal of binary tree is \n");
    printPostorder(root);
   getchar();
    return 0;
}
```

5.3 Binary Search Tree(BST)

Binary Search Tree

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

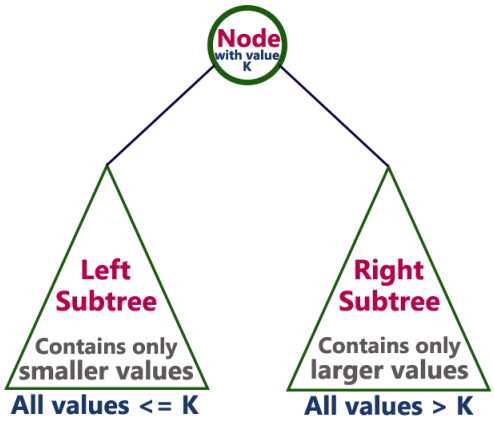
A binary tree has the following time complexities.

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

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

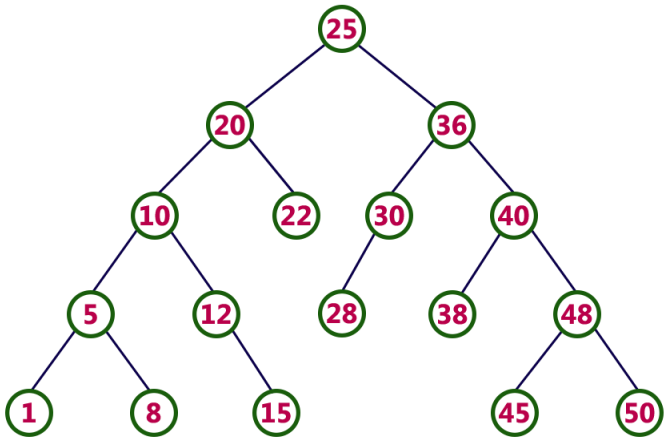
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

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



Example

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



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

5.3.1 Operations on a Binary Search Tree

The following operations are performed on a binary search tree.

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

1. Search Operation in BST

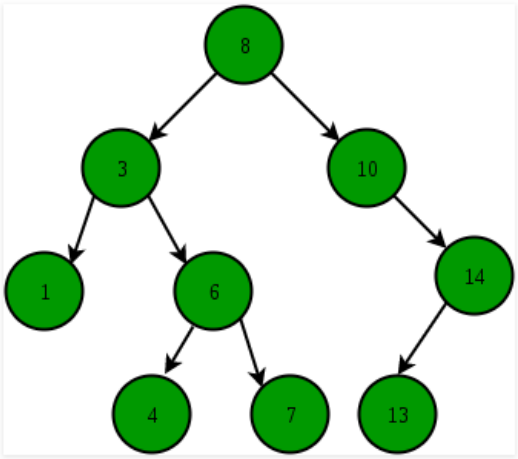
In a binary search tree, the search operation is performed with **O (log n)** time complexity. The search operation is performed as follows.

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node

- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Illustration to search 6 in below tree:

1. Start from root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If element to search is found anywhere, return true, else return false.



2. 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 value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5:** If newNode is **smaller** than **or equal** to the node, then move to its **left** child. If newNode is **larger** than the node, then move to its **right** child.
- **Step 6:** Repeat the above step until we reach to a **leaf** node (i.e., reach to NULL).
- **Step 7:** After reaching a leaf node, then insert the newNode as **left child** if newNode is **smaller or equal** to that leaf else insert it as **right child**.

Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

3. Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O (log n)** time complexity. Deleting a node from Binary search tree has 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 and child nodes.
- **Step 3:** Delete the node using **free** function and terminate the function.

Case 3: Deleting a node with two children

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

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

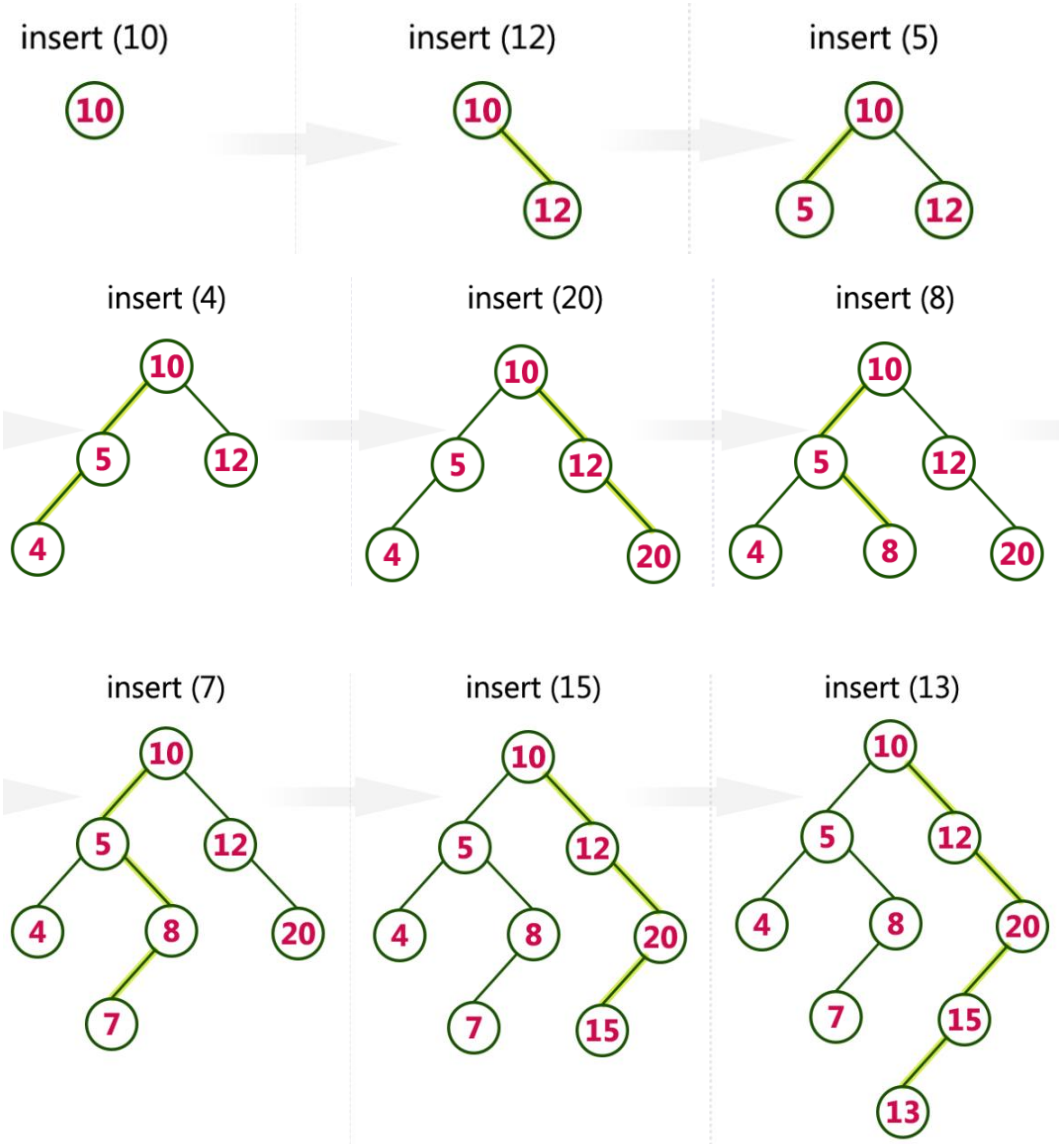
- **Step 5:** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6:** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7:** Repeat the same process until node is deleted from the tree.

Example

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

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

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



5.4 AVL Trees

AVL tree is a self-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.

A binary tree is said to be balanced, if the difference between the 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 differ by at most one.

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

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

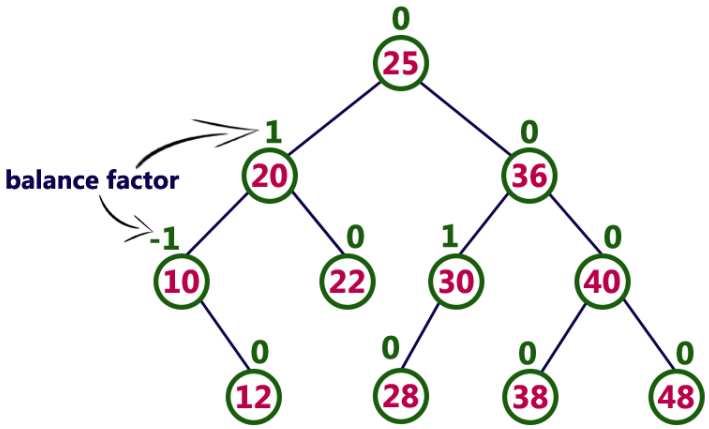
Definition:

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

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

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example



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

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

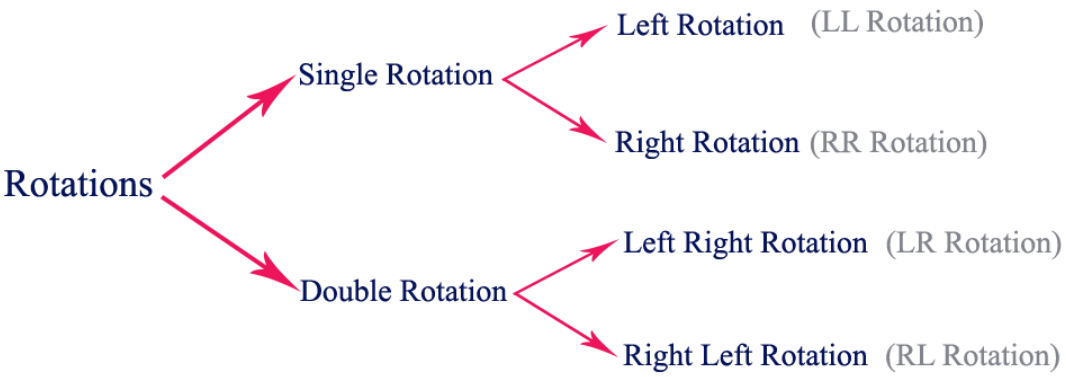
5.4.1 AVL Tree Rotations

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

Rotation operations are used to make a tree balanced.

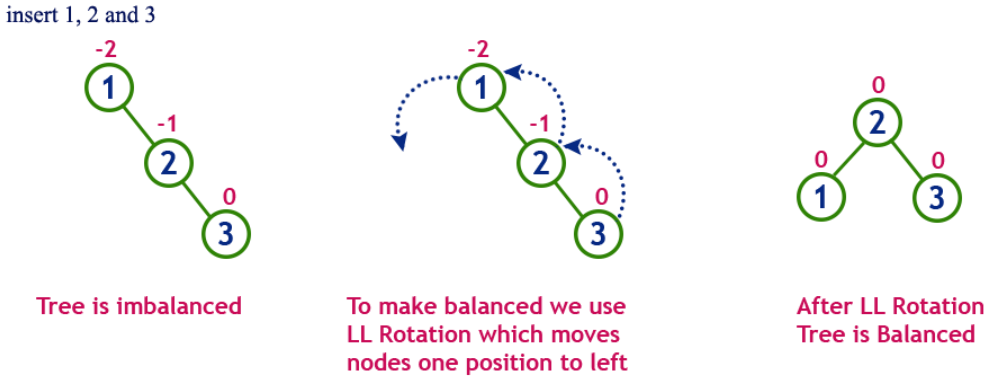
***Rotation is the process of moving the nodes to either left or right to make tree balanced*.**

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



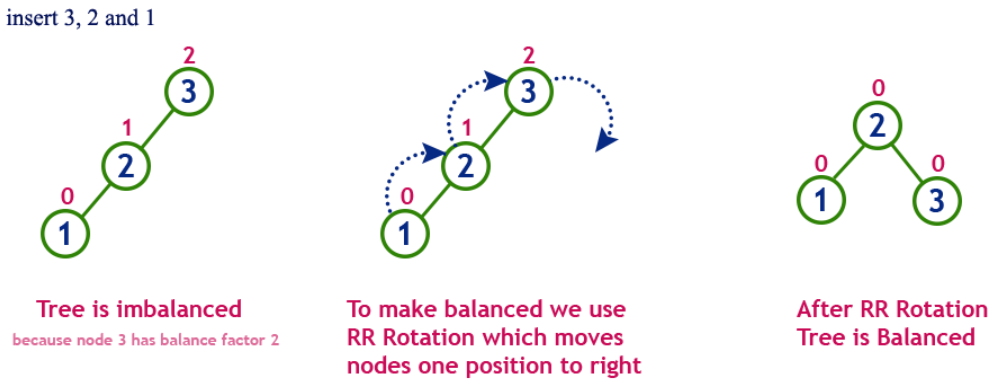
1. Single Left Rotation (LL Rotation)

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



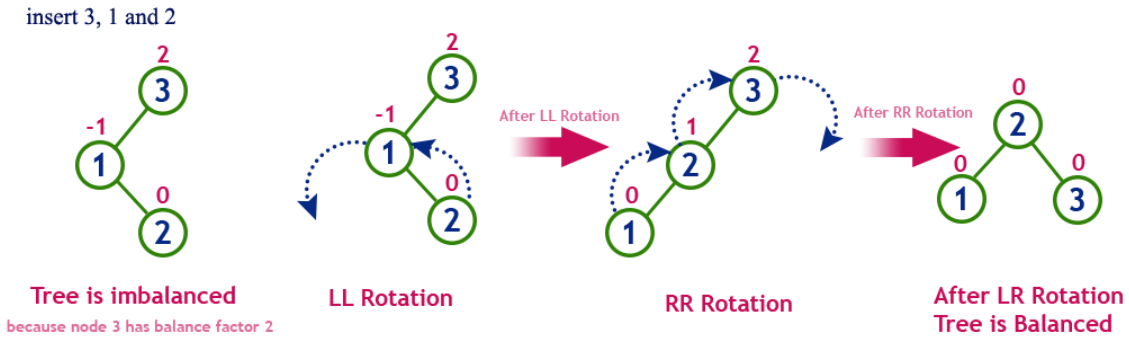
2. Single Right Rotation (RR Rotation)

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



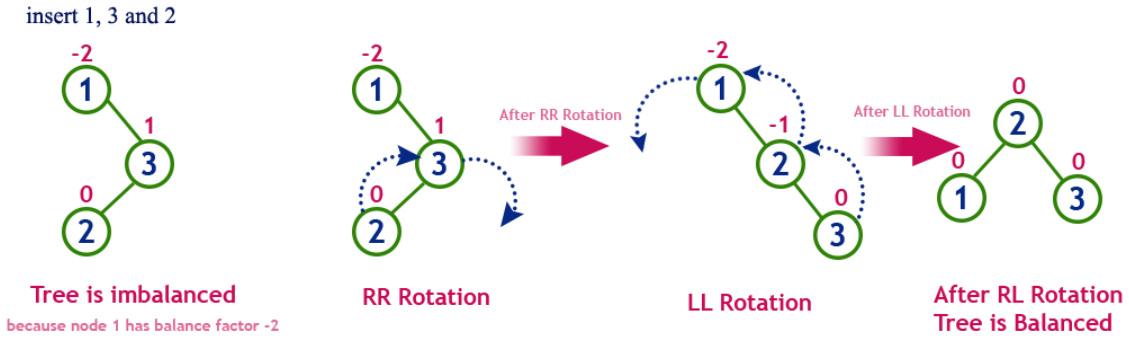
3. Left Right Rotation (LR Rotation)

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



4. Right Left Rotation (RL Rotation)

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



5.4.2 Operations on an AVL Tree

The following operations are performed on an AVL tree.

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

1. Search Operation in AVL Tree

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

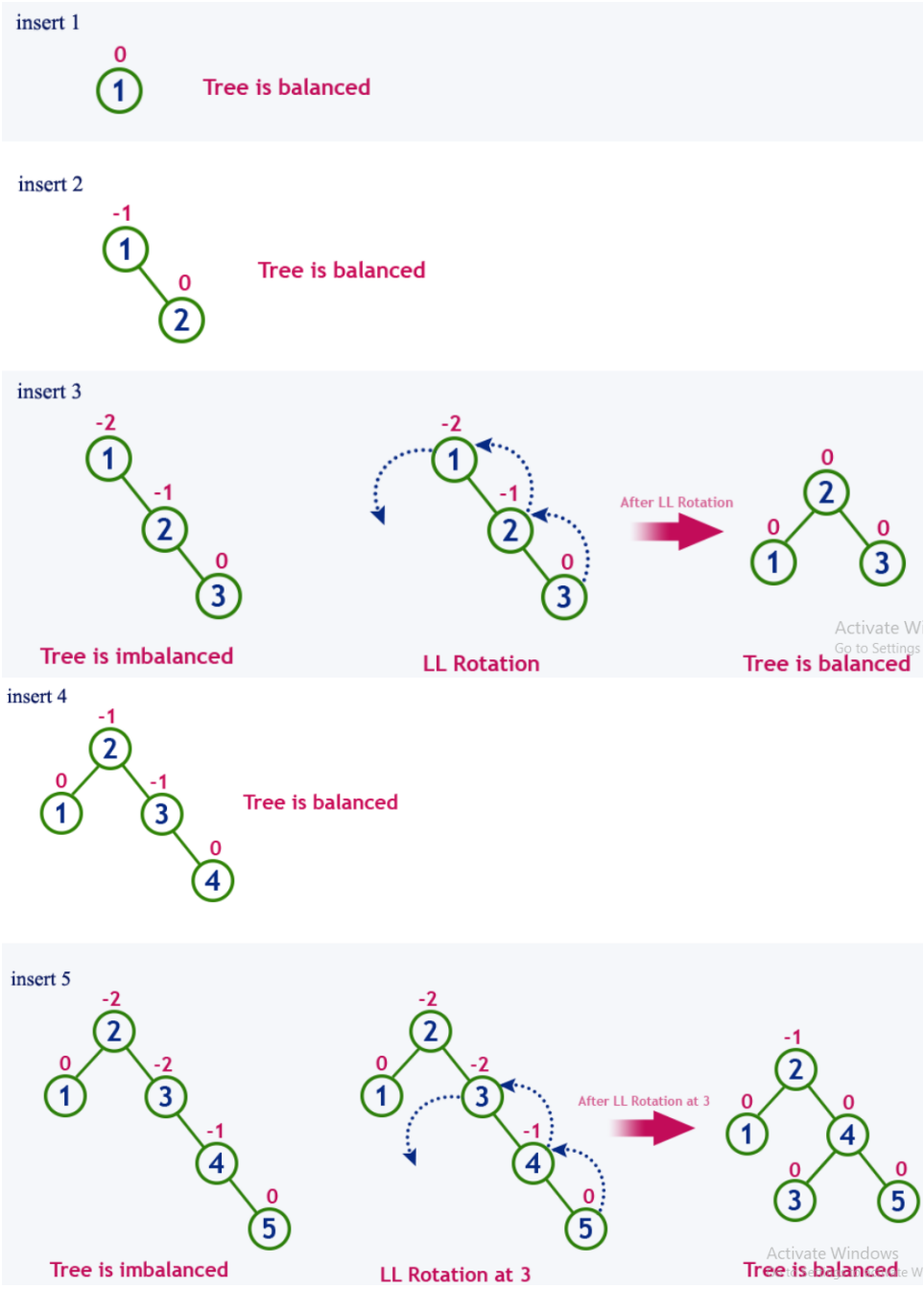
- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element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

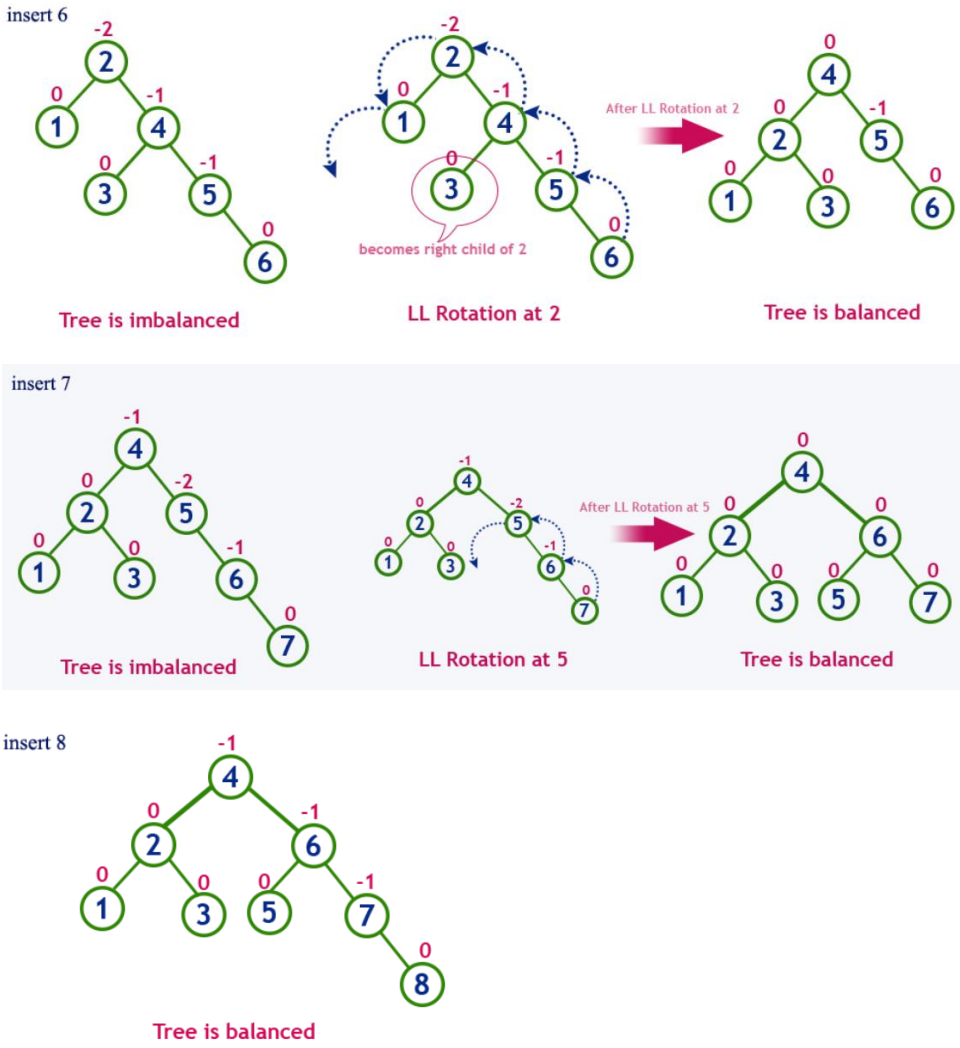
2. Insertion Operation in AVL Tree

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

- **Step 1:** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2:** After insertion, check the **Balance Factor** of every node.
- **Step 3:** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4:** If the **Balance Factor** of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

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





3. Deletion Operation in AVL Tree

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

5.5 Heap Sort

Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.

Consider an array *Arr* which is to be sorted using Heap Sort.

- Initially build a max heap of elements in Arr.
- The root element, that is *Arr* [1], will contain maximum element of *Arr*. After that, swap this element with the last element of *Arr* and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.

- Repeat the step 2, until all the elements are in their correct position.

Algorithm/Pseudo Code:

```
void heap_sort(int Arr[ ])
{
   int heap_size = N;
   build_maxheap(Arr);
   for(int i = N; i >= 2 ; i-- )
   {
      swap|(Arr[ 1 ], Arr[ i ]);
      heap_size = heap_size - 1;
      max_heapify(Arr, 1, heap_size);
   }
}
```

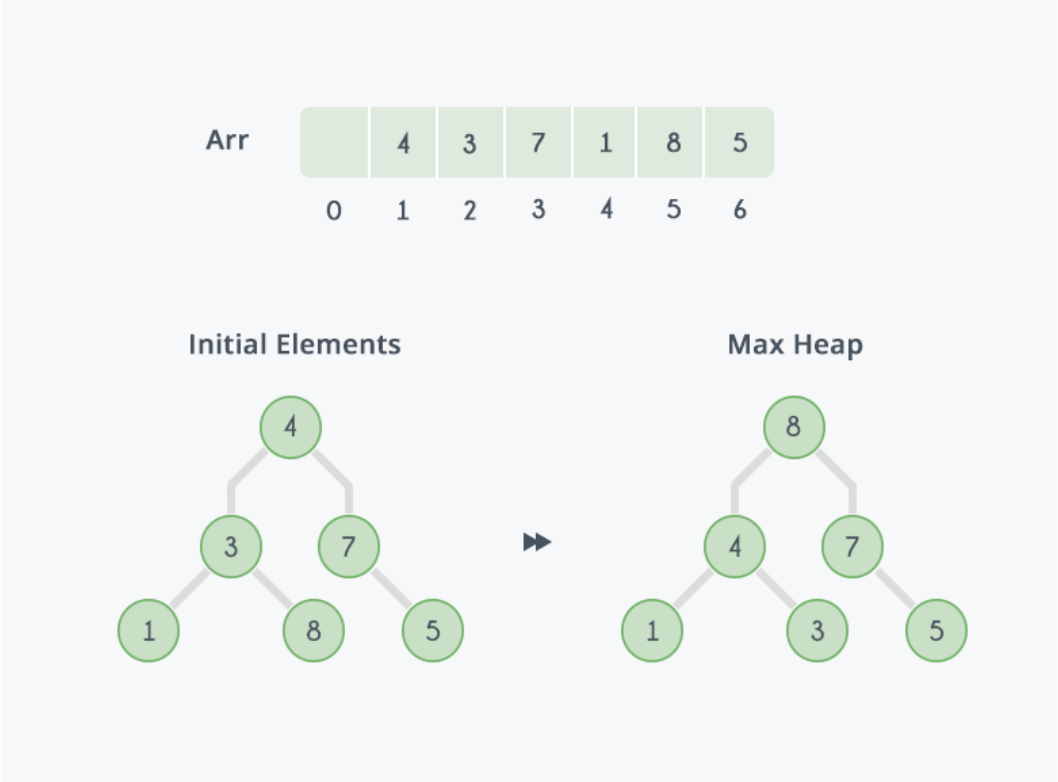
Complexity:

max_heapify has complexity O(log n), build_maxheap has complexity O(n) and we run max_heapify n−1 times in heap_sort function.

Therefore, complexity of heap_sort function is **O (n log n).**

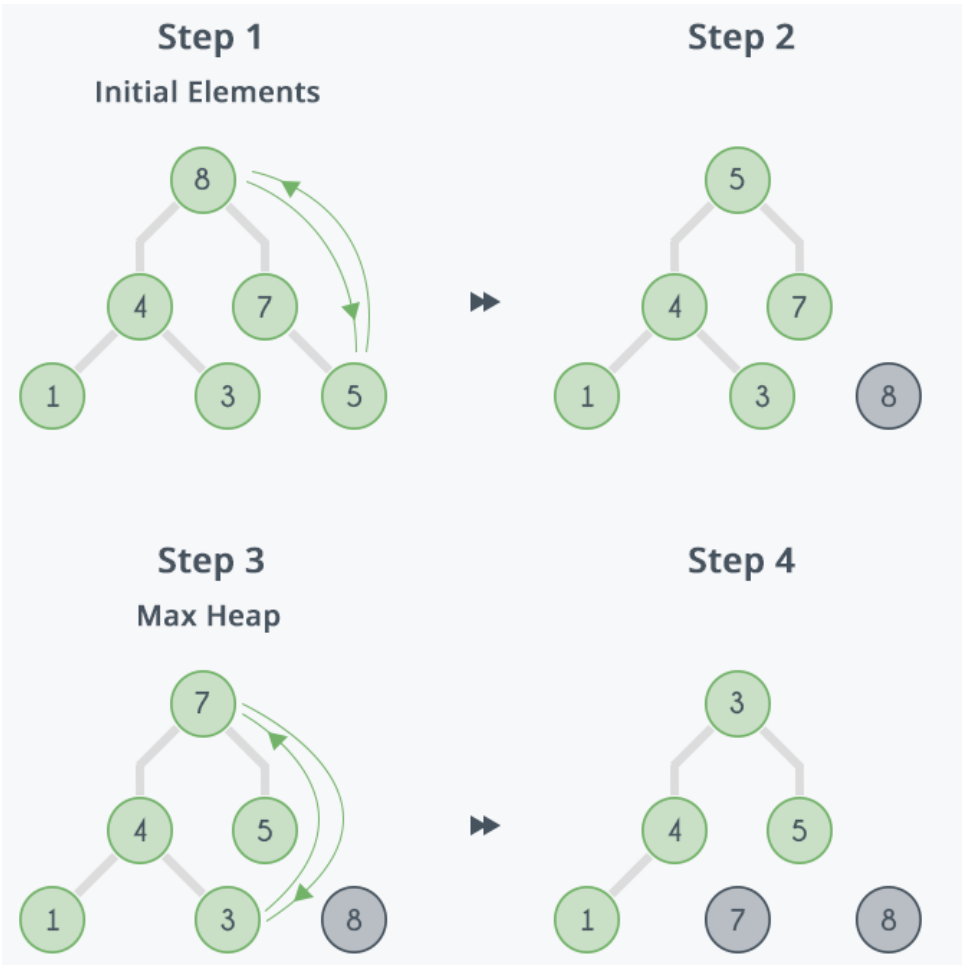
Example:

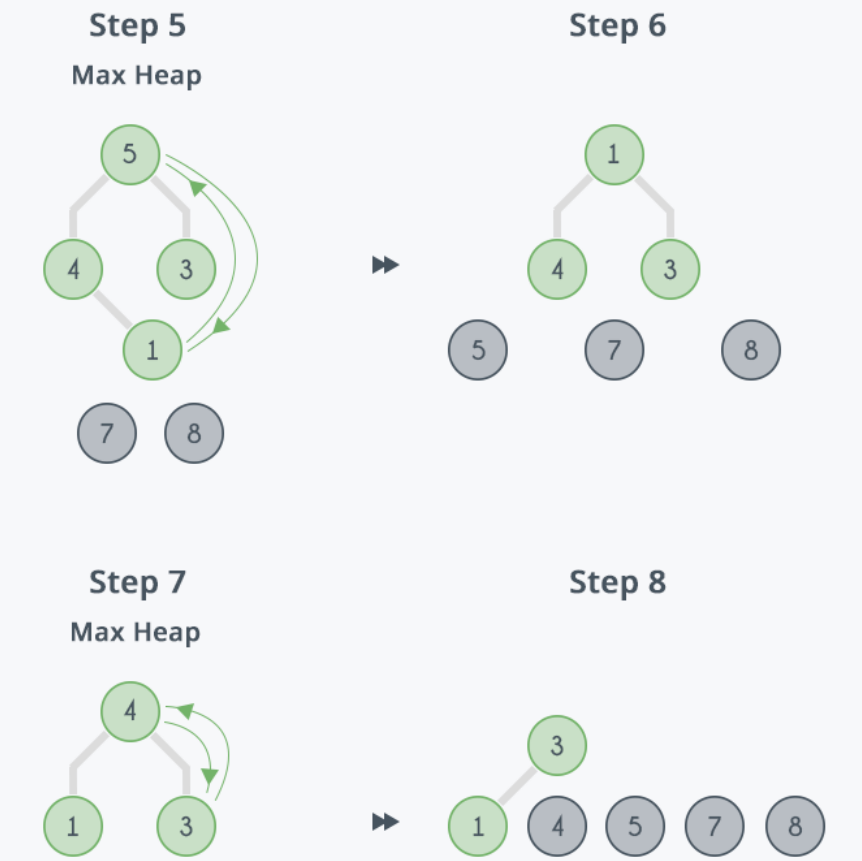
In the diagram below, initially there is an unsorted array *Arr* having 6 elements and then max-heap will be built.



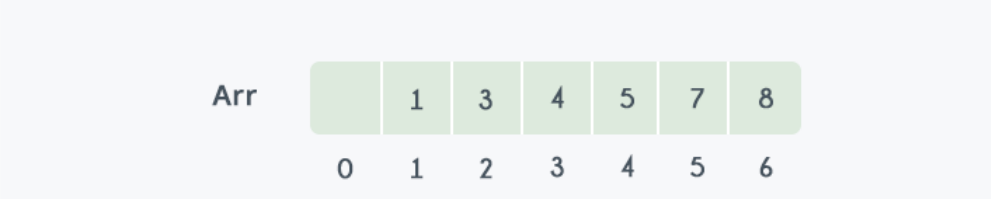
After building max-heap, the elements in the array Arr will be:

- Step 1: 8 is swapped with 5.
- Step 2: 8 is disconnected from heap as 8 is in correct position now and.
- Step 3: Max-heap is created and 7 is swapped with 3.
- Step 4: 7 is disconnected from heap.
- Step 5: Max heap is created and 5 is swapped with 1.
- Step 6: 5 is disconnected from heap.
- Step 7: Max heap is created and 4 is swapped with 3.
- Step 8: 4 is disconnected from heap.
- Step 9: Max heap is created and 3 is swapped with 1.
- Step 10: 3 is disconnected.





After all the steps, we will get a sorted array.



C program of Heap Sort

```
#include<stdio.h>

void create(int []);
void down_adjust(int [],int);

void main()
{
   int heap[30],n,i,last,temp;
   printf("Enter no. of elements:");
   scanf("%d",&n);
   printf("\nEnter elements:");
   for(i=1;i<=n;i++)
      scanf("%d",&heap[i]);

   //create a heap
   heap[0]=n;
   create(heap);

   //sorting
   while(heap[0] > 1)
   {
      //swap heap[1] and heap[last]
      last=heap[0];
      temp=heap[1];
      heap[1]=heap[last];
      heap[last]=temp;
      heap[0]--;
      down_adjust(heap,1);
   }

   //print sorted data
   printf("\nArray after sorting:\n");
   for(i=1;i<=n;i++)
      printf("%d ",heap[i]);
}

void create(int heap[])
{
   int i,n;
   n=heap[0]; //no. of elements
   for(i=n/2;i>=1;i--)
      down_adjust(heap,i);
}

void down_adjust(int heap[],int i)
{
   int j,temp,n,flag=1;
```

```
    n=heap[0];

    while(2*i<=n && flag==1)
    {
        j=2*i;    //j points to left child
        if(j+1<=n && heap[j+1] > heap[j])
            j=j+1;
        if(heap[i] > heap[j])
            flag=0;
        else
        {
            temp=heap[i];
            heap[i]=heap[j];
            heap[j]=temp;
            i=j;
        }
    }
}
```