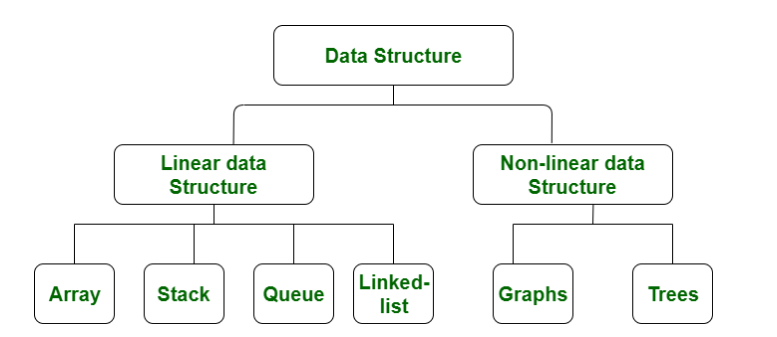
**Classification of Data Structure**



[**Non-linear Data Structure**](https://www.geeksforgeeks.org/overview-of-data-structures-set-2-binary-tree-bst-heap-and-hash/)**:**   
Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**. In a non-linear data structure, single level is not involved. Therefore, we can’t traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are [trees](https://www.geeksforgeeks.org/data-structures/) and [graphs](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/).

**Trees**

A tree data structure consists of various nodes linked together. The structure of a tree is hierarchical that forms a relationship like that of the parent and a child. The structure of the tree is formed in a way that there is one connection for every parent-child node relationship. Only one path should exist between the root to a node in the tree.

**Graph**

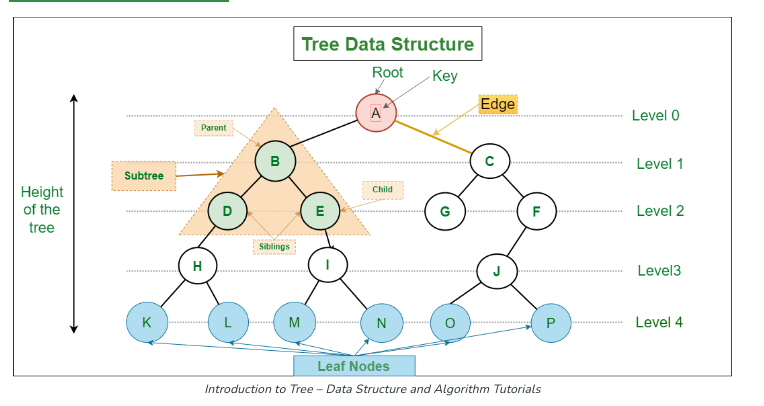
Graphs are those types of non-linear data structures which consist of a definite quantity of vertices and edges. The vertices or the nodes are involved in storing data and the edges show the vertices relationship. The difference between a graph to a tree is that in a graph there are no specific rules for the connection of nodes. Real-life problems like social networks, telephone networks, etc. can be represented through the graphs.

**General Tree**

It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

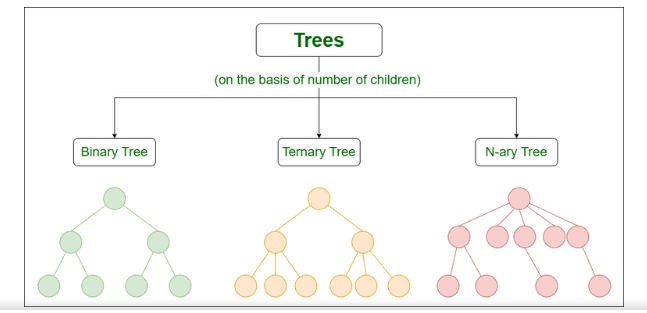
We can also say that tree data structure has roots, branches, and leaves connected with one another.



**Basic Terminologies In Tree Data Structure:**

* **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.**{B}** is the parent node of **{D, E}**.
* **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
* **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
* **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{K, L, M, N, O, P, G}** are the leaf nodes of the tree.
* **Internal node:** A node with at least one child is called Internal Node
* **The ancestors** of a node are all the nodes along the path from the root to that node. For example, ancestors of node **E** are **{A,B}**
* **The descendants** of a node are all the nodes along the path from that node to a leaf node . For example, **{I, M, N}** are descendants of **E**
* **Sibling:** Children of the same parent node are called siblings.**{D,E}** are called siblings.
* **Level of a node** is defined by letting the root to be at level **zero**, while a node at level **l** has children at level **l + 1.**
* **The depth of a node** is its level number.
* **The degree of a node** is the number of subtrees of that node. For example, node **A** is of degree 2.
* **The degree of the tree** is the **maximum** degree of all nodes.
* **The height of a tree** is the **maximum** level of any node in this tree.
* **Subtree**: any node of the tree along with its descendant.
* **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has ***N*** nodes then it will have ***(N-1)*** edges. There is only one path from each node to any other node of the tree.

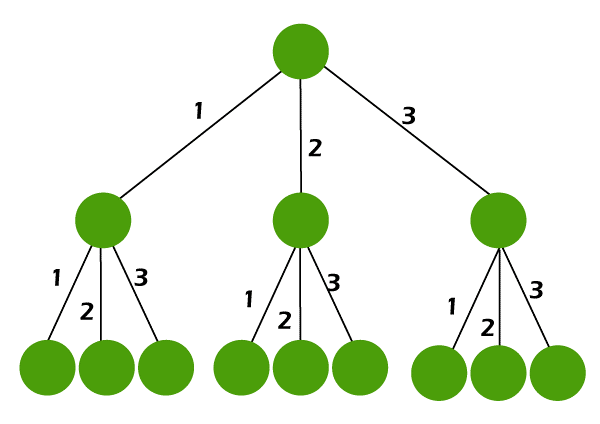
## Types of Tree data structures:



* [**Binary tree**](https://www.geeksforgeeks.org/types-of-trees-in-data-structures/)**:** In a binary tree, each node can have a maximum of two children linked to it.
* [**Ternary Tree**](https://www.geeksforgeeks.org/ternary-tree/)**:** A Ternary Tree is a tree in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
* [**N-ary Tree or Generic Tree**](https://www.geeksforgeeks.org/generic-treesn-array-trees/)**:** if every internal node has no more than N children. A tree is called a full N-ary tree if every internal node has exactly N children.

Note

If we have a full tree with degree *d* and height *h* , the number of node is **(dh+1 -1)/(d-1)**



**d=3, h=2, #nodes = (32+1 - 1)(3-1)= 26/2=13**

The minimum number of nodes in any tree with height *h* is *h+1*

**Basic Operation Of Tree Data Structure:**

* **Create** – create a tree in the data structure.
* **Insert** − Inserts data in a tree.
* **Search** − Searches specific data in a tree to check whether it is present or not.
* **Traversal**:
  + **Preorder Traversal** – perform Traveling a tree in a pre-order manner in the data structure.
  + **In order Traversal** – perform Traveling a tree in an in-order manner.
  + **Post-order Traversal** –perform Traveling a tree in a post-order manner.

**Application of Tree Data Structure:**

* **File System:** This allows for efficient navigation and organization of files.
* **Data Compression**:[Huffman coding](https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/) is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
* **Compiler Design:** In compiler design, a syntax tree is used to represent the structure of a program.
* **Database Indexing**: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.

**Advantages of Tree Data Structure:**

* Tree offer **Efficient Searching** Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
* Trees provide a hierarchical representation of data, making it**easy to organize and navigate**large amounts of information.
* The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

**Disadvantages of Tree Data Structure:**

* Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to **inefficient search times.**
* Trees demand**more memory space requirements** than some other data structures like arrays and linked lists, especially if the tree is very large.
* The implementation and **manipulation of trees can be complex.**

**Binary Tree**

*A binary tree is a tree data structure in which each internal node can have at most two children, which are referred to as the left child and the right child.*

The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.

# Types of Binary Tree

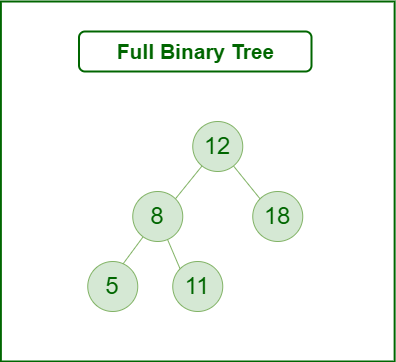
**1) Types of Binary Tree based on the number of children:**

Following are the types of Binary Tree based on the number of children:

1. Full Binary Tree
2. Degenerate Binary Tree
3. Skewed Binary Trees

### ****1. Full Binary Tree****

 A Binary Tree is a full binary tree if every node has 0 or 2 children(every parent node/internal node has either two or no children.)



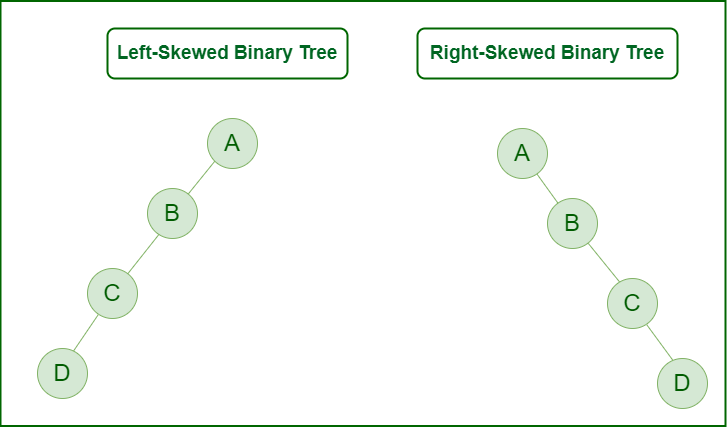
### ****2. Degenerate (or pathological) tree****

A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.



### ****3. Skewed Binary Tree****

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



**Types of Binary Tree On the basis of the completion of levels:**

1. Complete Binary Tree
2. Perfect Binary Tree
3. Balanced Binary Tree

# Properties of Binary Tree

### ****1. The maximum number of nodes at level ‘l’ of a binary tree is 2l:****

**Note:**Here level is the number of nodes on the path from the root to the node (including root and node). The level of the root is 0.

### ****2. The**** ****Maximum number of nodes in a binary tree of height ‘h’ is 2h+1 – 1:****

note that : if height is h then there is h+1 levels , so

#nodes= 20 + 21 + 22 + ……+ 2h  =

**3. The** **Minimum number of nodes in a binary tree of height ‘h’ is h+1.**

**4. The number of leave nodes in BT is the number of nodes that have 2 children +1**

**5. If BT has *N* nodes, The Max number of levels is *N( each node has only one child)***

* **In a full binary tree, every node except the leaves has exactly two children**: In a full binary tree, all non-leaf nodes have exactly two children. This means that there are no unary nodes in a full binary tree.

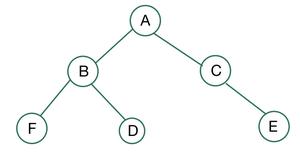
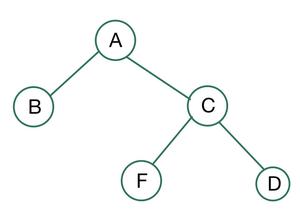
Max # nodes = 2h+1 -1, Min #nodes = 2h+1, h= log2(n+1) - 1

* **In a complete binary tree, every level of the tree is completely filled except for the last level, which can be partially filled:**  This means that there are **no gaps** in the tree and all nodes are connected to their parent nodes.

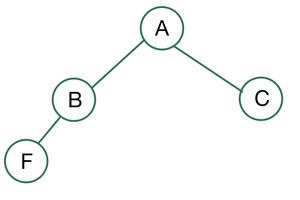
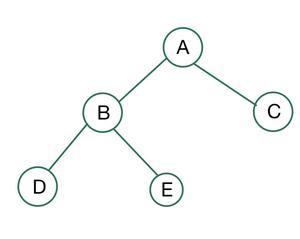
Max # nodes = 2h+1 -1, Min #nodes = 2h,

In other words: a binary tree is said to be a **complete binary tree** if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the **last level appear as far left as possible ( left to right )**.



 neither complete nor full

Full but not complete

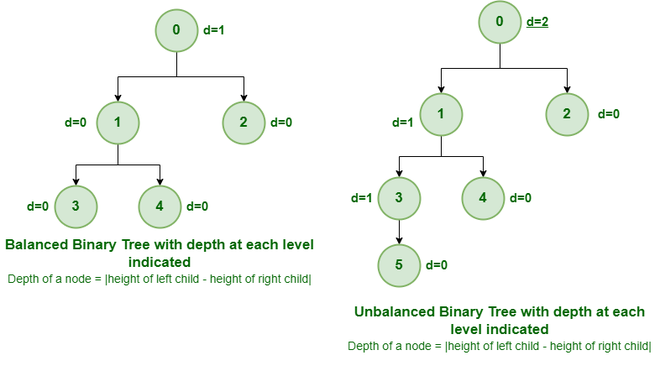
 

Complete but not Full Complete and Full

### A balanced binary tree (height-balanced binary tree, )is a Binary tree that follows the 3 conditions:

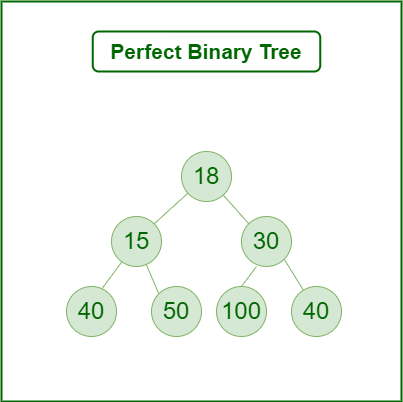
* The height of the left and right tree for any node does not differ by more than 1.
* The left subtree of that node is also balanced.
* The right subtree of that node is also balanced.

A single node is always balanced. It is also referred to as a height-balanced binary tree. In other words, It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



* In a perfect binary tree , leaf nodes are at the same depth, and all non-leaf nodes have two children. In simple terms, this means that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.

Max # nodes = 2h+1 -1, # leaf nodes = 2h, h= **log(N + 1) – 1**



**Some Special Types of Trees:**

**On the basis of node values**, the Binary Tree can be classified into the following special types:

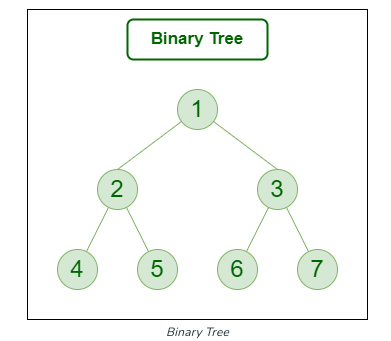
1. Binary Search Tree
2. AVL Tree
3. Red Black Tree
4. B Tree
5. B+ Tree
6. Segment Tree

# Lightbox

## ****Applications of Binary Tree:****

Binary trees have many applications in computer science, including data storage and retrieval, expression evaluation, network routing, and game AI. They can also be used to implement various algorithms such as searching, sorting, and graph algorithms.

* In compilers, **Expression Trees** are used which is an application of binary trees.
* **Huffman coding** trees are used in data compression algorithms.
* **Priority Queue** is another application of binary tree that is used for searching maximum or minimum in O(1) time complexity.
* Binary trees can be used to represent the **decision-making process of computer-controlled characters in games**, such as in **decision trees**.
* **Searching algorithms**, such as in binary search trees which can be used to quickly find an element in a sorted list.
* **Sorting algorithms**, such as in heap sort which uses a binary heap to sort elements efficiently.

**Representation of Binary Tree:**

Each node in the tree contains the following:

* Data
* Pointer to the left child
* Pointer to the right child

Below is an example of a tree node with integer data.

 struct node {

    int data;

    struct node\* left;

    struct node\* right;

};

**Basic Operations On Binary Tree:**

* Inserting an element.
* Removing an element.
* Searching for an element.
* Deletion for an element.
* Traversing an element. There are three types of traversals in a binary tree which will be discussed ahead.

**Auxiliary Operations On Binary Tree:**

* Finding the height of the tree
* Find the level of the tree
* Finding the size of the entire tree.

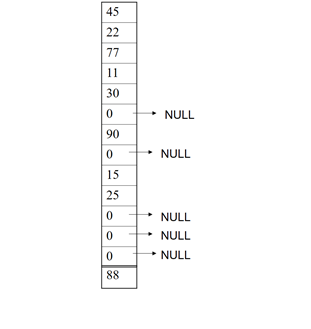
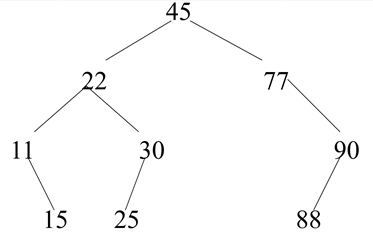
**Representing Binary Tree in memory**

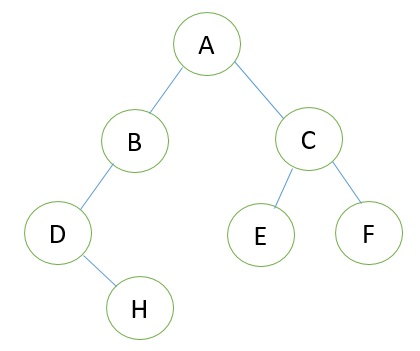
Let **T** be a Binary Tree. There are two ways of representing **T** in the memory as follow : **Sequential and Link Representation of Binary Tree.**

**Sequential representation of Binary Trees**

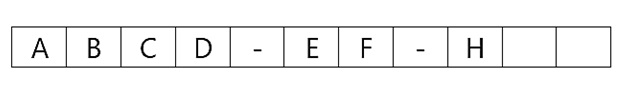
This representation uses only a single linear array Tree as follows:

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



* It can be seen that a sequential representation of a binary tree requires numbering of nodes; starting with nodes on level 1, then on level 2, and so on. The nodes are numbered from left to right.
* It is an ideal case for the representation of a **complete binary tree** and in this case, **no space is wasted**. However, for other binary trees, most of the space remains unutilized. As can be seen in the figure, we require 14 locations in the array even though the tree has only 9 nodes. If null entries for successors of the terminal nodes are included, we would actually require 29 locations instead of 14. Thus sequential representation is usually inefficient unless the binary tree is complete or nearly complete.
* **For Example:**
* Consider the following Tree:

* **Its sequential representation is as follow:**

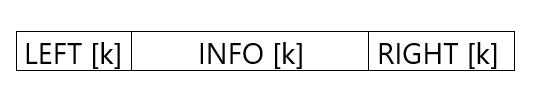


**Linked Representation of Binary Tree**

Consider a Binary Tree **T**. **T** will be maintained in memory by means of a linked list representation which uses three parallel arrays; **INFO**, **LEFT**, and **RIGHT** pointer variable **ROOT** as follows. In Binary Tree each node **N** of **T** will correspond to a location **k** such that

1. **LEFT [k]** contains the location of the left child of node **N**.
2. **INFO [k]** contains the data at the node **N**.
3. **RIGHT [k]** contains the location of right child of node **N**.

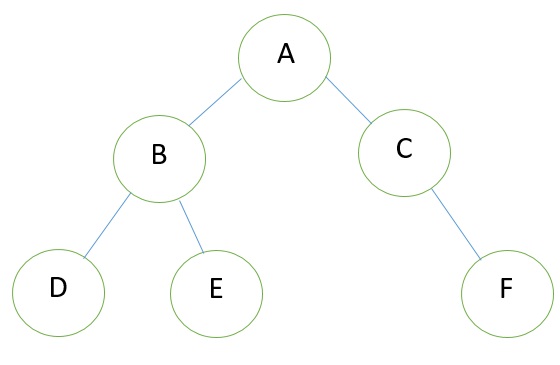
**Representation of a node:**



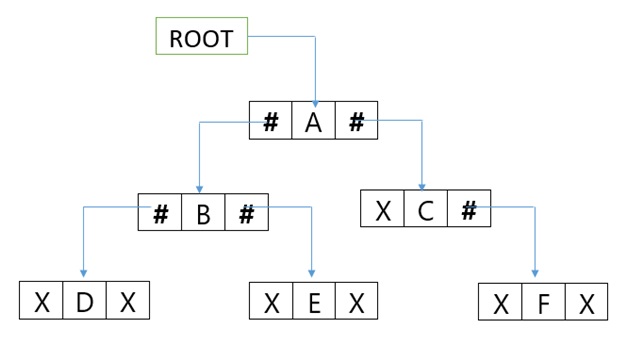
In this representation of binary tree root will contain the location of the root **R** of **T**. If any one of the subtree is empty, then the corresponding pointer will contain the null value if the tree **T** itself is empty, the **ROOT** will contain the null value.

**Example**

Consider the binary tree **T** in the figure. A schematic diagram of the linked list representation of **T** appears in the following figure. Observe that each node is pictured with its three fields, and that the empty subtree is pictured by using x for null entries.

**Binary Tree**

**Linked Representation of the Binary Tree**



**Traversing Binary Trees**

There are three standard ways of traversing a binary tree T with root R. These are preorder, inorder and postorder traversals

* **Preorder**

1. PROCESS the root R
2. Traverse the left subtree of R in preorder
3. Traverse the right subtree of R in preorder

**Uses of Preorder:**

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

* **Inorder**

1. Traverse the left subtree of R in inorder
2. Process the root R
3. Traverse the right subtree of R in inorder

**Uses of Inorder Traversal:**

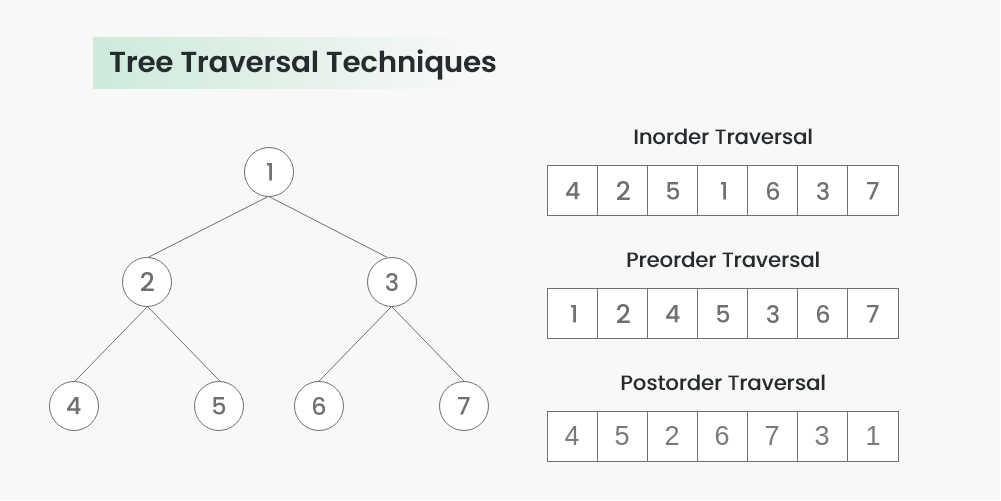
In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

* **Postorder**

1. Traverse the left subtree of R in postorder
2. Traverse the right subtree of R in postorder
3. Process the root R

**Uses of Postorder:**

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



**Construct Tree from Inorder & Preorder**

Let us consider the below traversals:

* Inorder sequence: D B E A F C
* Preorder sequence: A B D E C F

In a Preorder sequence, the leftmost element is the root of the tree. So we know ‘A’ is the root for given sequences. By searching ‘A’ in the Inorder sequence, we can find out all elements on the left side of ‘A’ is in the left subtree and elements on right in the right subtree. So we know the below structure now.

A

/ \

/ \

D B E F C

We recursively follow the above steps and get the following tree.

A

/ \

/ \

B C

/ \ /

/ \ /

D E F

**Algorithm:**buildTree()

1. Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick the next element in the next recursive call.
2. Create a new tree node tNode with the data as the picked element.
3. Find the picked element’s index in Inorder. Let the index be inIndex.
4. Call buildTree for elements before inIndex and make the built tree as a left subtree of tNode.
5. Call buildTree for elements after inIndex and make the built tree as a right subtree of tNode.
6. return tNode.

**Construct Tree from Inorder & Postorder**

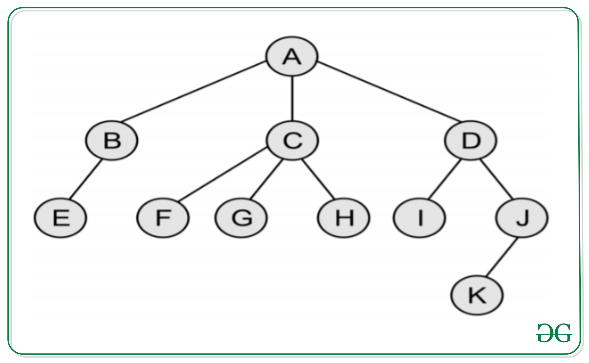
**Construct Tree from Postorder & Preorder**

# Convert a Generic Tree(N-array Tree) to Binary Tree

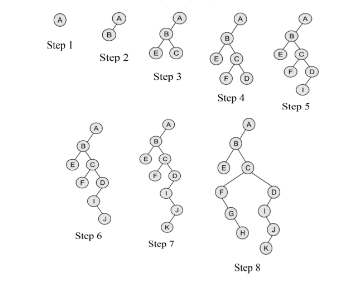
we will discuss the conversion of the Generic Tree to a Binary Tree. Following are the rules to convert a [Generic(N-array Tree)](https://www.geeksforgeeks.org/generic-treesn-array-trees/) to a [Binary Tree](https://www.geeksforgeeks.org/binary-tree-data-structure/):

* The root of the Binary Tree is the Root of the Generic Tree.
* The left child of a node in the Generic Tree is the Left child of that node in the Binary Tree.
* The right sibling of any node in the Generic Tree is the Right child of that node in the Binary Tree.

**Examples:**  
Convert the following Generic Tree to Binary Tree:

[](https://media.geeksforgeeks.org/wp-content/uploads/20200324122406/GenricTree.png)

Below is the Binary Tree of the above Generic Tree:



***Note:****If the parent node has only the right child in the general tree then it becomes the rightmost child node of the last node following the parent node in the binary tree. In the above example, if node****B****has the right child node****L****then in binary tree representation****L****would be the right child of node****D****.*

Below are the steps for the conversion of Generic Tree to Binary Tree:

1. As per the rules mentioned above, the root node of general tree **A** is the root node of the binary tree.
2. Now the leftmost child node of the root node in the general tree is **B** and it is the leftmost child node of the binary tree.
3. Now as **B** has **E** as its leftmost child node, so it is its leftmost child node in the binary tree whereas it has **C** as its rightmost sibling node so it is its right child node in the binary tree.
4. Now **C** has **F** as its leftmost child node and **D** as its rightmost sibling node, so they are its left and right child node in the binary tree respectively.
5. Now **D** has **I** as its leftmost child node which is its left child node in the binary tree but doesn’t have any rightmost sibling node, so doesn’t have any right child in the binary tree.
6. Now for **I**, **J** is its rightmost sibling node and so it is its right child node in the binary tree.
7. Similarly, for **J**, **K** is its leftmost child node and thus it is its left child node in the binary tree.
8. Now for **C**, **F** is its leftmost child node, which has **G** as its rightmost sibling node, which has **H** as its just right sibling node and thus they form their left, right, and right child node respectively.

Implementation in C++

#include <iostream>

#include <vector>

class TreeNode {

public:

    int val;

    TreeNode\* left;

    TreeNode\* right;

    std::vector<TreeNode\*> children;

    TreeNode(int val)

    {

        this->val = val;

        this->left = this->right = nullptr;

    }

};

TreeNode\* convert(TreeNode\* root)

{

    if (!root) {

        return nullptr;

    }

    if (root->children.size() == 0) {

        return root;

    }

    if (root->children.size() == 1) {

        root->left = convert(root->children[0]);

        return root;

    }

    root->left = convert(root->children[0]);

    root->right = convert(root->children[1]);

    for (int i = 2; i < root->children.size(); i++) {

        TreeNode\* rightTreeRoot = root->right;

        while (rightTreeRoot->left != nullptr) {

            rightTreeRoot = rightTreeRoot->left;

        }

        rightTreeRoot->left = convert(root->children[i]);

    }

    return root;

}

void printTree(TreeNode\* root)

{

    if (!root) {

        return;

    }

    std::cout << root->val << " ";

    printTree(root->left);

    printTree(root->right);

}

int main()

{

    TreeNode\* root = new TreeNode(1);

    root->children.push\_back(new TreeNode(2));

    root->children.push\_back(new TreeNode(3));

    root->children.push\_back(new TreeNode(4));

    root->children.push\_back(new TreeNode(5));

    root->children[0]->children.push\_back(new TreeNode(6));

    root->children[0]->children.push\_back(new TreeNode(7));

    root->children[3]->children.push\_back(new TreeNode(8));

    root->children[3]->children.push\_back(new TreeNode(9));

    TreeNode\* binaryTreeRoot = convert(root);

    // Output: 1 2 3 4 5 6 7 8 9

    printTree(binaryTreeRoot);

}

**Output**

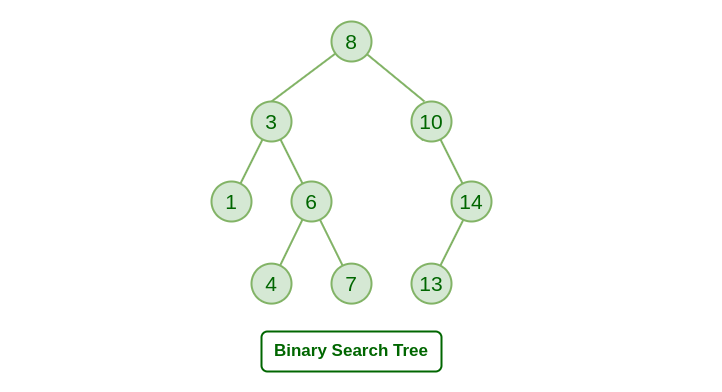
1 2 6 7 3 4 5 8 9

**Time Complexity: O(n) , Auxiliary Space: O(h)**

**Binary Search Tree (BST )**

**Binary Search Tree** is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.



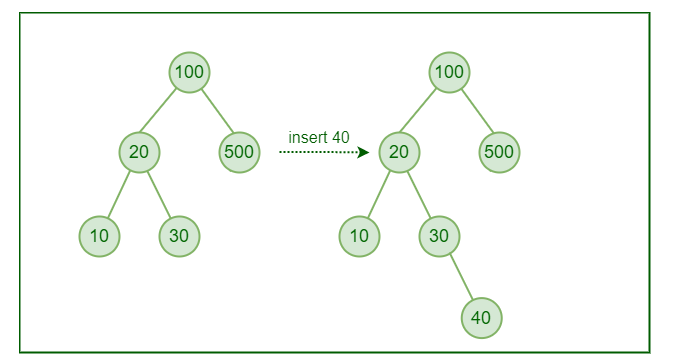
**Basic Operations:**

1. [Insertion in Binary Search Tree](https://www.geeksforgeeks.org/insertion-in-binary-search-tree/)
2. [Searching in Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)
3. [Deletion in Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/)
4. [Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order](https://www.geeksforgeeks.org/binary-search-tree-traversal-inorder-preorder-post-order/)
5. [Convert a normal BST to Balanced BST](https://www.geeksforgeeks.org/convert-normal-bst-balanced-bst/)

**Insertion in Binary Search Tree**

Given a **BST**, the task is to insert a new node in this **BST**.

Example:

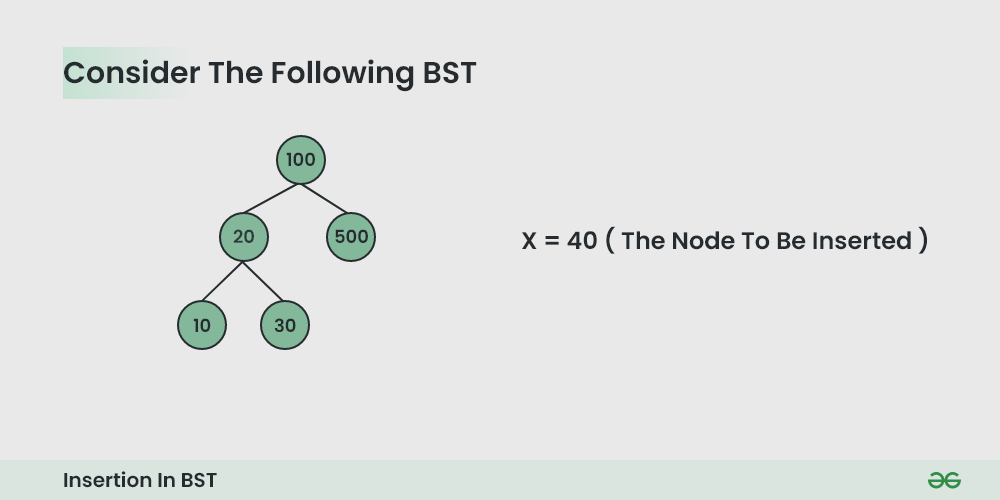
****

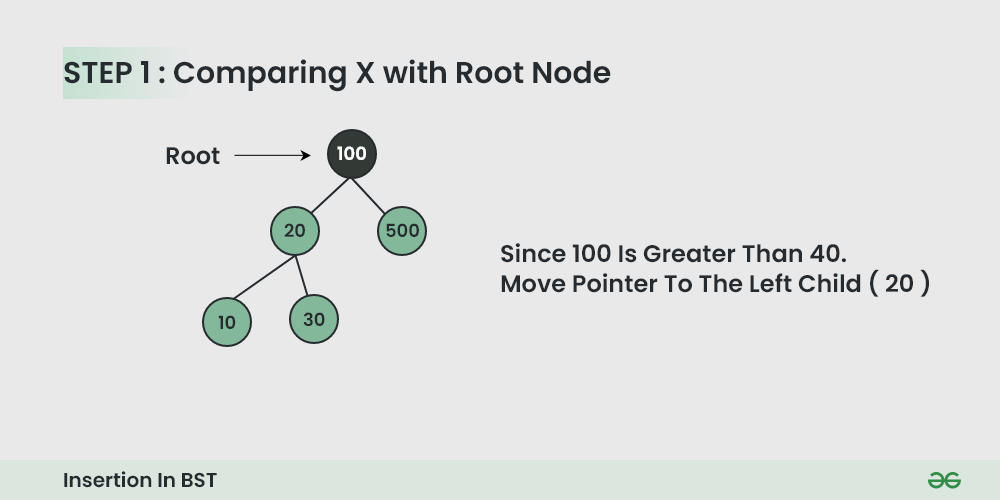
**How to Insert a value in a Binary Search Tree:**

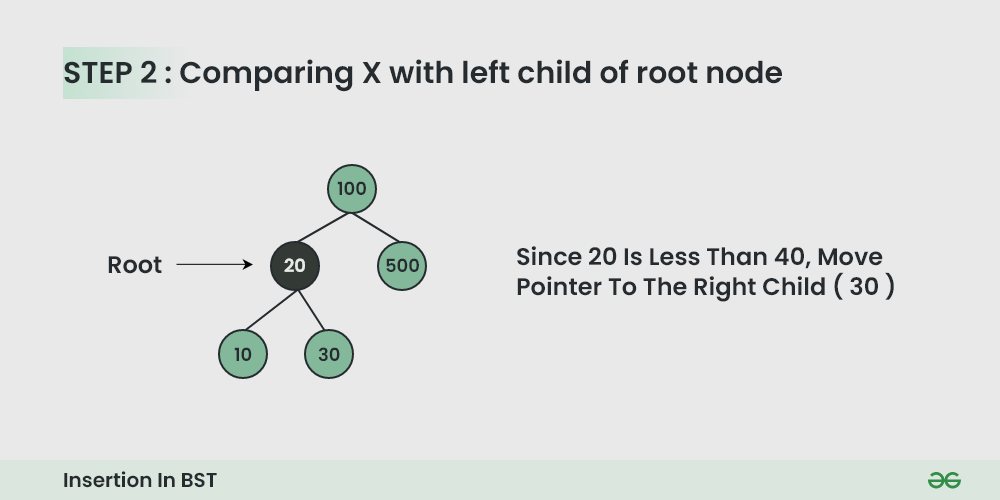
A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

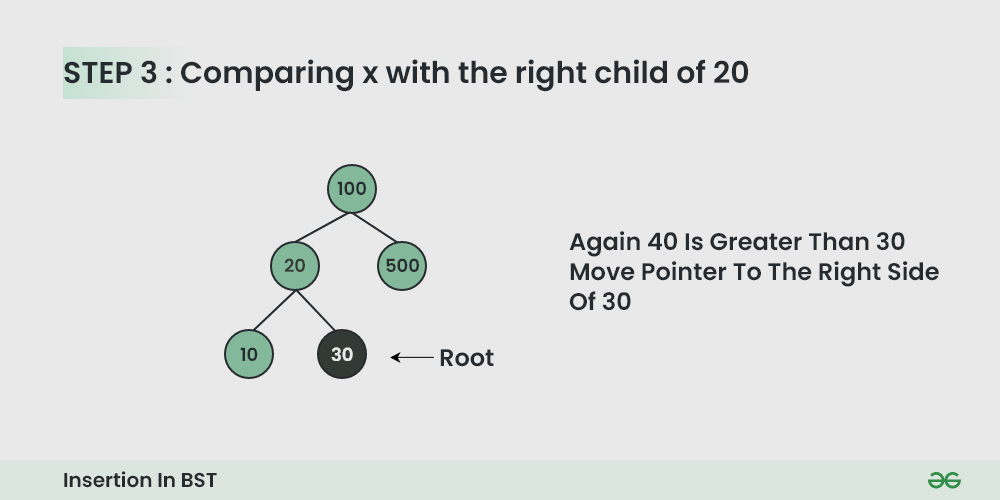
* Check the value to be inserted (say **X**) with the value of the current node (say **val**) we are in:
  + If **X** is less than **val**move to the left subtree.
  + Otherwise, move to the right subtree.
* Once the leaf node is reached, insert **X** to its right or left based on the relation between **X** and the leaf node’s value.

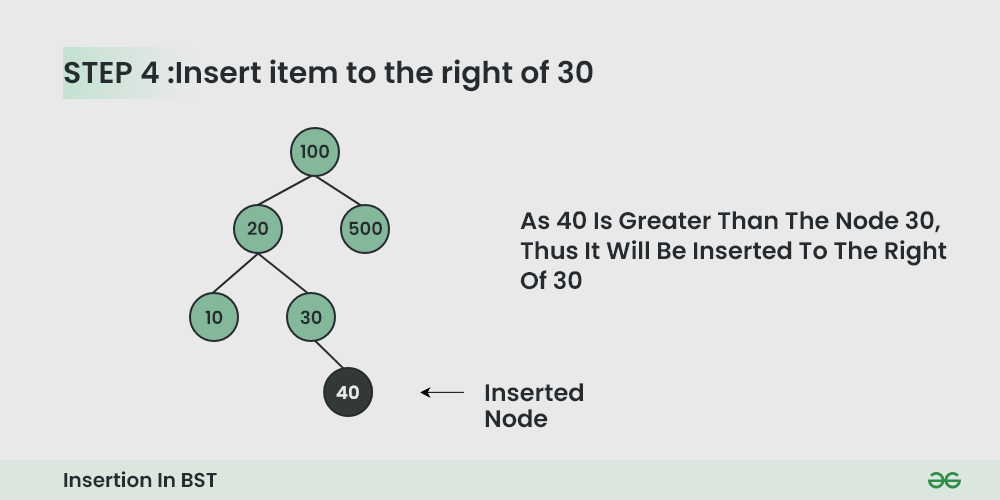
**Illustration**











**Time Complexity:**

* The worst-case time complexity of insert operations is**O(h)** where **h** is the height of the Binary Search Tree.
* In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become **n** and the time complexity of insertion operation may become**O(n).**

**Auxiliary Space:**The auxiliaryspace complexity of insertion into a binary search tree is**O(1)**

# Searching in Binary Search Tree (BST)

Given a **BST**, the task is to search a node in this **BST**.

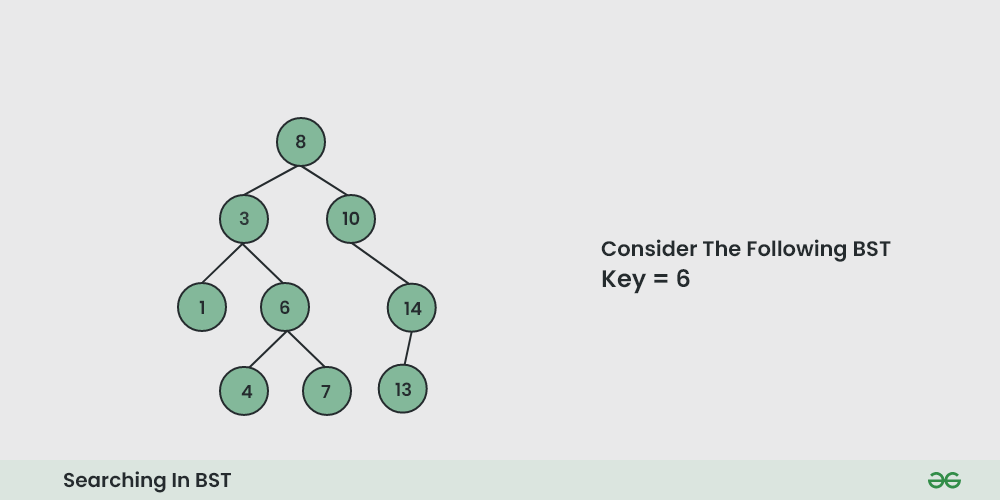
*For searching a value in BST, consider it as a sorted array. Now we can easily perform search operation in BST using*[***Binary Search Algorithm***](https://www.geeksforgeeks.org/binary-search/)*.*

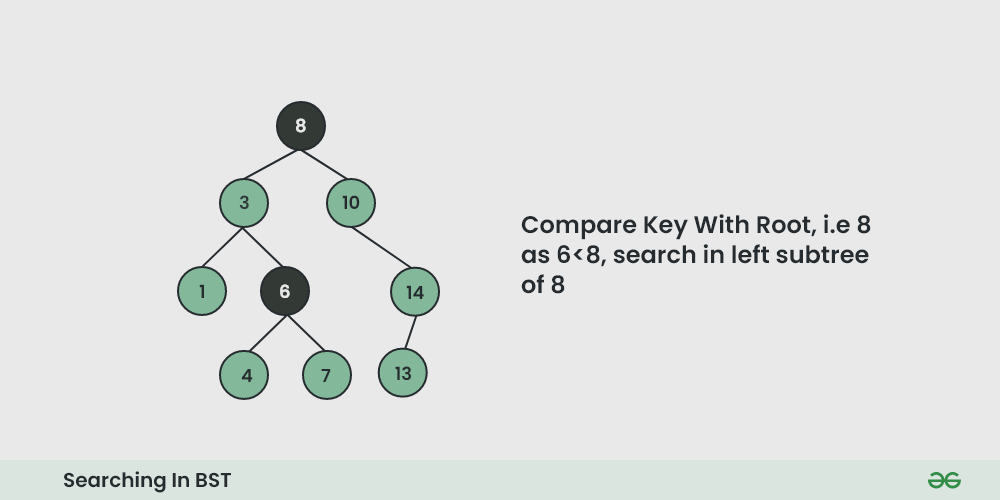
**Algorithm to search for a key in a given Binary Search Tree:**

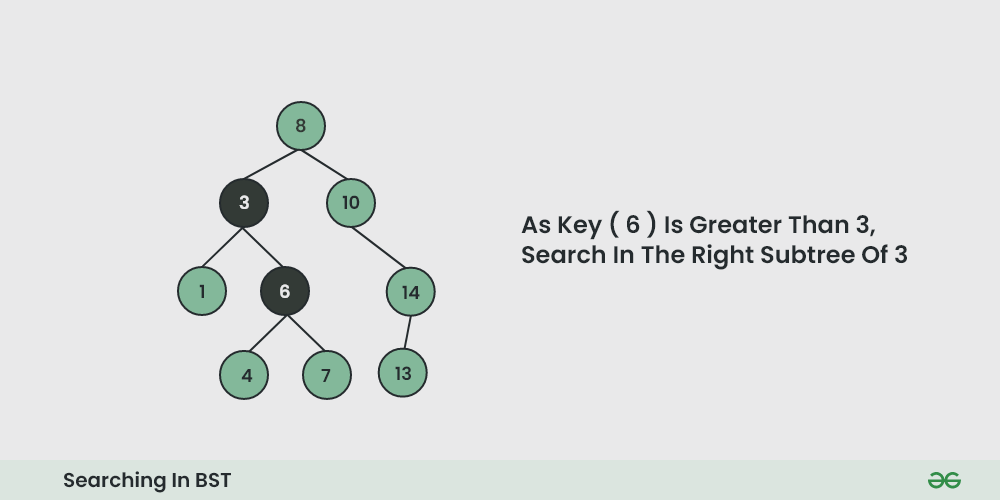
Let’s say we want to search for the number **X,**We start at the root. Then:

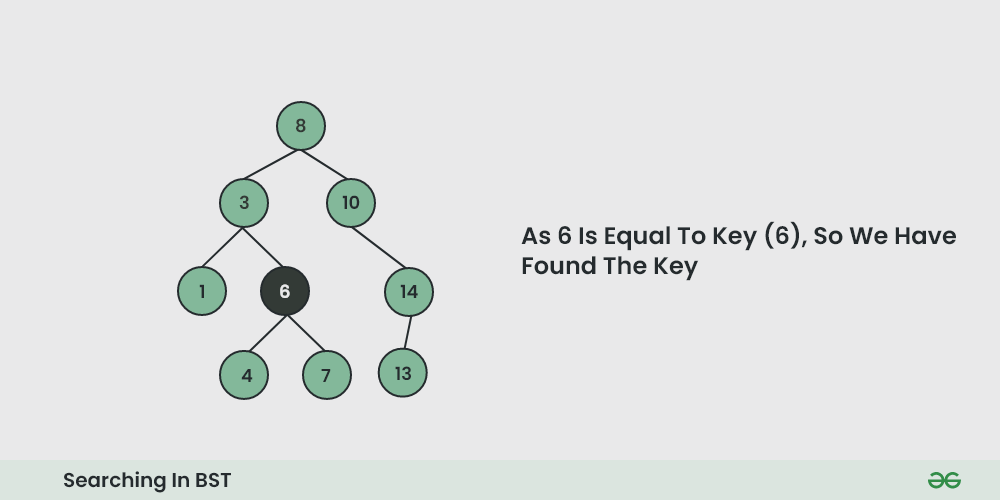
* We compare the value to be searched with the value of the root.
  + If it’s equal we are done with the search if it’s smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
* Repeat the above step till no more traversal is possible
* If at any iteration, key is found, return True. Else False.

**Illustration of searching in a BST:**





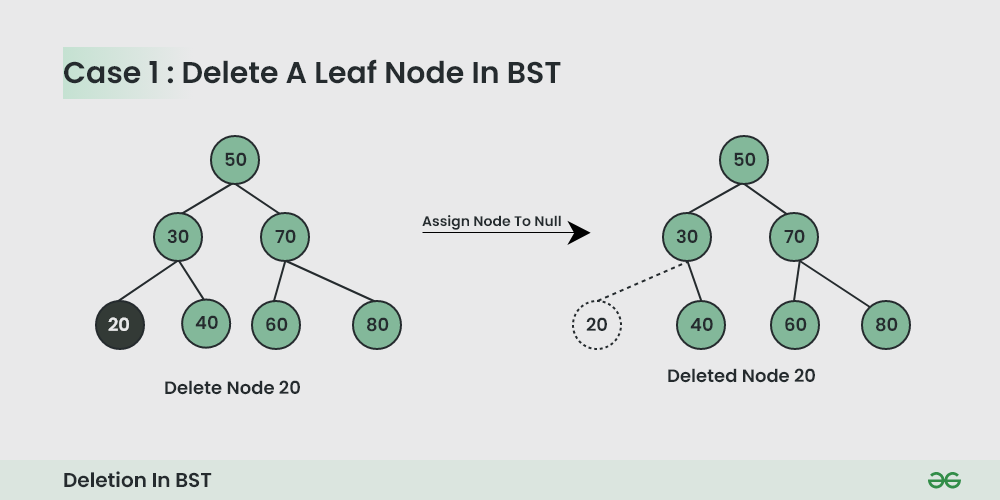




# Deletion in Binary Search Tree (BST)

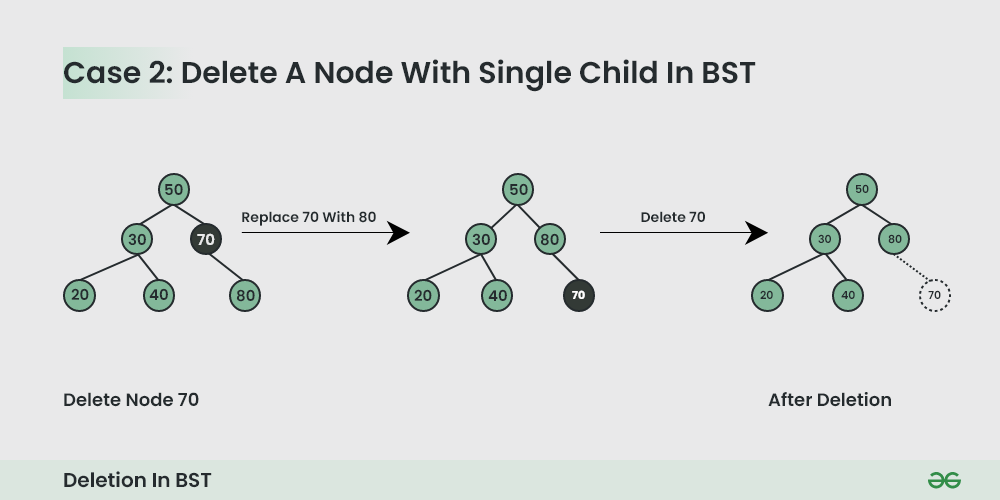
Given a **BST**, the task is to delete a node in this **BST**, which can be broken down into 3 scenarios:

### Case 1. Delete a Leaf Node in BST



### Case 2. Delete a Node with Single Child in BST

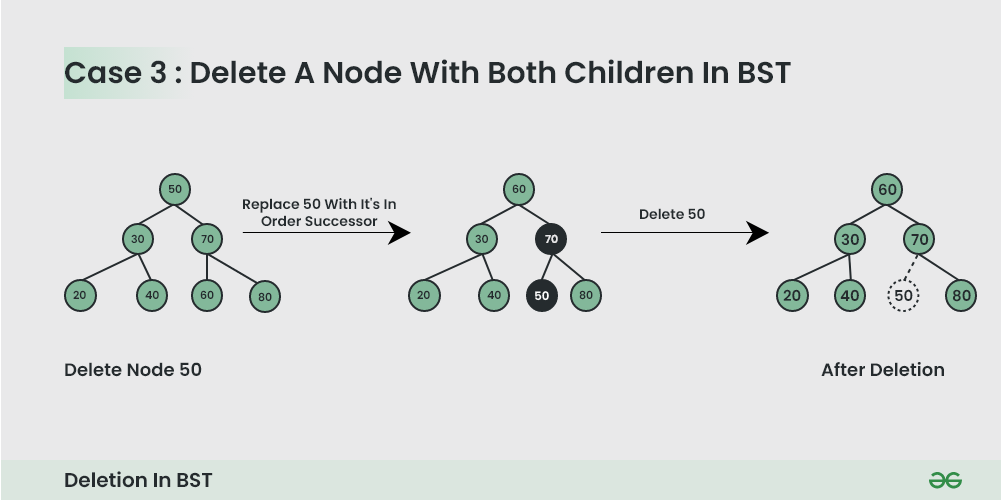
*Deleting a single child node is also simple in BST.****Copy the child to the node and delete the node****.*



### Case 3. Delete a Node with Both Children in BST

*Deleting a node with both children is not so simple. Here we have to****delete the node is such a way, that the resulting tree follows the properties of a BST.***

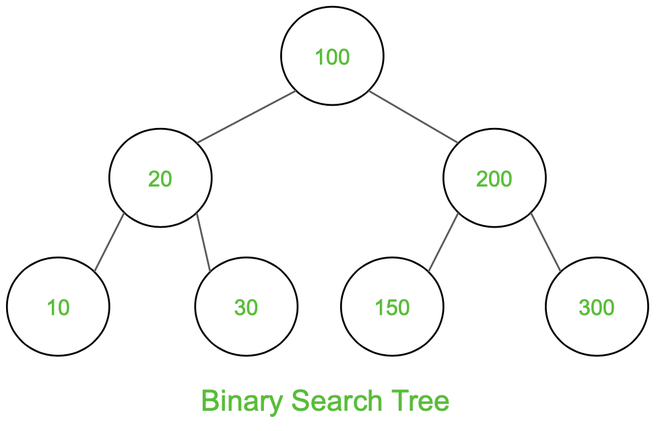
*The trick is to find the* ***inorder successor*** *of the node. Copy contents of the inorder successor to the node, and delete the inorder successor.*



**Note:** Inorder successor is needed only when the right child is not empty. In this particular case, the inorder successor can be obtained by finding the minimum value in the right child of the node.

# Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order

Given a [Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-data-structure/), The task is to print the elements in inorder, preorder, and postorder traversal of the Binary Search Tree.

***Input:***

***Output:*** *Inorder Traversal: 10 20 30 100 150 200 300  
Preorder Traversal: 100 20 10 30 200 150 300  
Postorder Traversal: 10 30 20 150 300 200 100*

***Note***

The ***[inorder traversal](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/)****of t*he BST gives the values of the nodes in sorted order. To get the decreasing order visit the right, root, and left subtree.