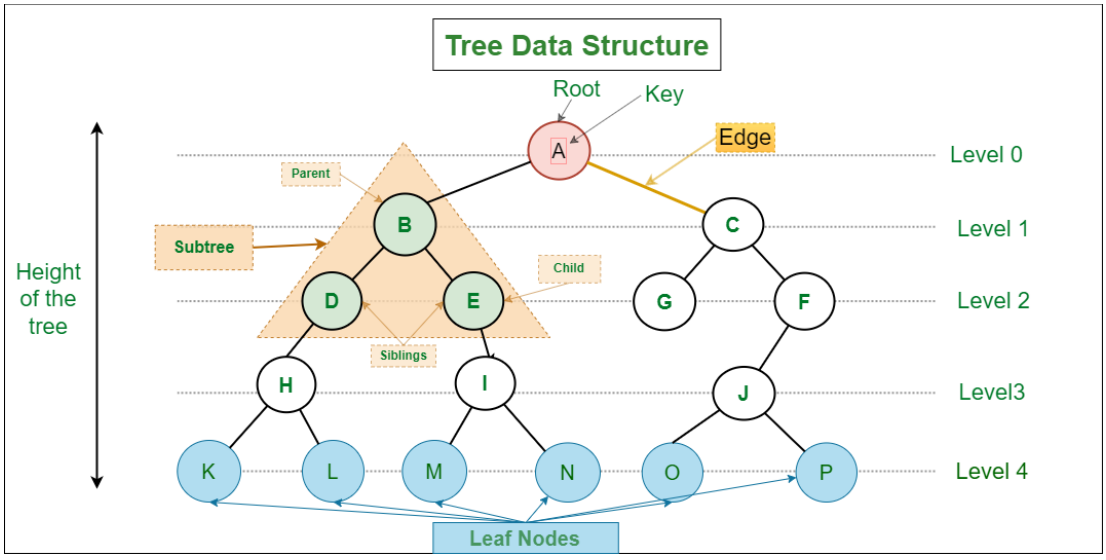
Tree – Data Structure and Algorithm

A **tree data structure** is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. 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**.

This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has **roots, branches, and leaves** connected with one another.
In a tree with N node there will be maximum N-1 edges.



Why Tree is considered a non-linear data structure?

The data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

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.

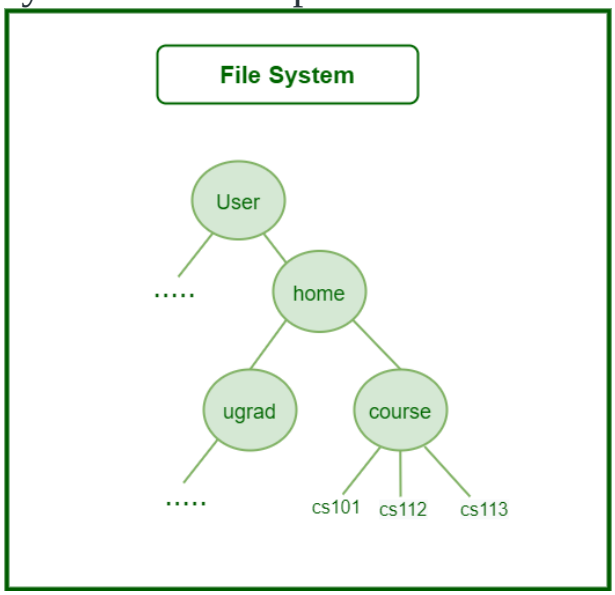
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{K, L, M, N, O, P}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** Any successor node on the path from the leaf node to that node. **{E,I}** are the descendants of the node **{B}.**
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

Properties of a Tree:

- **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.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

Why to use Tree Data Structure?

1. One reason to use trees might be because we want to store information that naturally forms a hierarchy. For example, the file system on a computer:



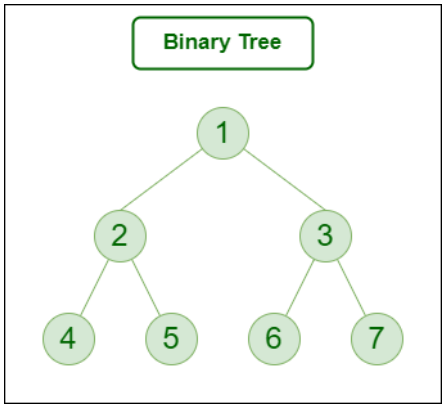
File System

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Basic Operation Of Tree:

Create – create a tree in data structure.
Insert – Inserts data in a tree.
Search – Searches specific data in a tree to check it is present or not.

Traversal (detailed description is provided in the next section)
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.



Here,

Node 1 is the root node

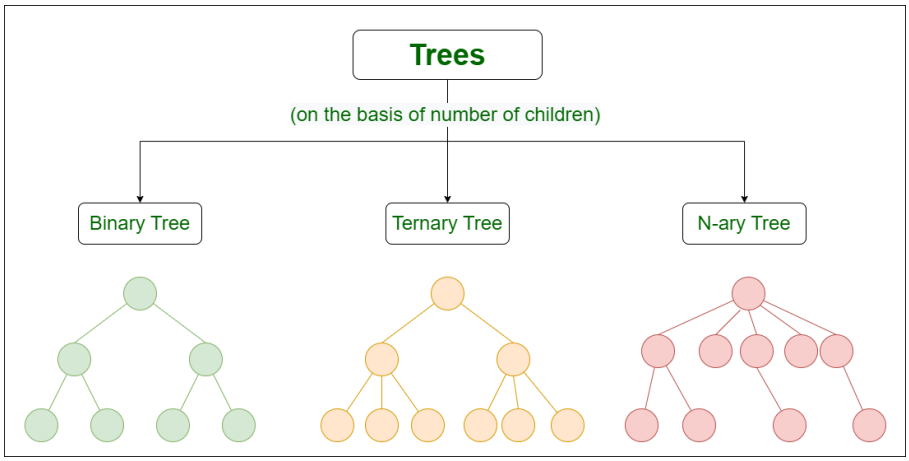
1 is the parent of 2 and 3

2 and 3 are the siblings

4, 5, 6 and 7 are the leaf nodes

1 and 2 are the ancestors of 5

Different types of tree



Binary Tree — is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

The ADT Binary Tree.

Abstract Data Type binaryTree.

{

Instances

Collection of elements; if not empty, the collection is partitioned into a root, left subtree, & right subtree; each subtree is also a binary tree;

Operations

empty (): return true if empty, otherwise false;

size(): return number of elements/nodes in the tree;

PreOrder (visit): preorder traversal of binary tree; visit is the visit function to use;

inOrder (visit): inorder traversal of binary tree;

postOrder (visit): postOrder traversal of binary tree;

level Order (visit): levelorder traversal of binary tree;

}

abstract class "binaryTree"

```
template <class T>
class binaryTree
{
  Public:
    virtual ~binaryTree() { }
    virtual bool empty() const = 0;
    virtual int size() const = 0;
    virtual void preOrder (void (*) (T*)) = 0;
    virtual void inOrder (void (*) (T*)) = 0;
    virtual void postOrder (void (*) (T*)) = 0;
    virtual void levelOrder (void (*) (T*)) = 0;
};
```

NOTE

The class T refers to the datatype of the nodes in the binary tree.

The data type `void (*)(T*)` is used to specify the data type of single parameter.

```
virtual void levelOrder(BinaryTreeNode<T> *root)=0;
};
```

program: Data members and visit method

```
template <class T>
class LinkedBinaryTree: public binaryTree <T>
{ private:
      binaryTreeNode<T> *root; // pointer to root
      int treeSize;
  public:
      LinkedBinaryTree() // constructor
       { root = NULL;
         treeSize=0;
       }
  ~LinkedBinaryTree(); //destructor
     { delete root;
     }
  bool empty() const { return treeSize==0; }
  int size() const { return treeSize; }
  void preOrder(binaryTreeNode<T> *root);
  void inOrder(binaryTreeNode<T> *root);
  void postOrder(binaryTreeNode<T> *root);
  void levelOrder(binaryTreeNode<T> *root);
  int height(binaryTreeNode<T> *root) const;
};
```

pre order Method

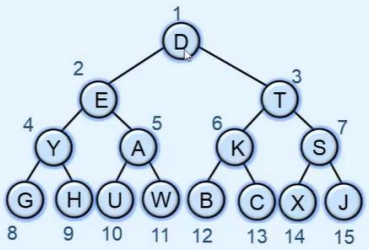
```
template <class T>
void LinkedBinaryTree<T>::preOrder(binaryTreeNode<T>
                                   *root)
{ if(root!=NULL)
  { cout << root -> element << endl;
    preOrder(root -> leftChild);
    preOrder(root -> rightChild);
  }}
```

program : Determining the height of a Binary Tree

```
template <class T>
int LinkedBinaryTree<T>::height(BinaryTreeNode<T>
                                *root)
{ if (root == NULL)
  { cout << "Tree is empty" << endl;
    return 0;
  }
  int hl = height(root->leftChild); // height of left subtree
  int hr = height(root->rightChild); // height of right subtree
    if(hl > hr)
      return ++hl;
    else
      return ++hr;
}
```

Sequential Representation of Binary Trees

One-dimensional array



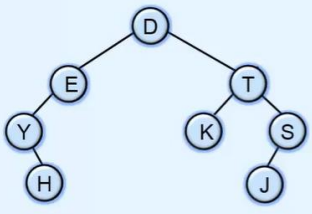
node numbered k Data stored in `tree[k]`

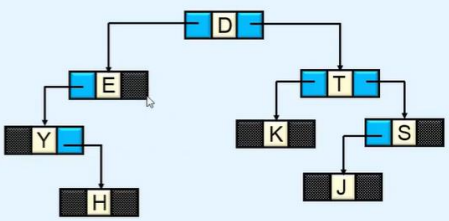
tree

	D	E	T	Y	A	K	S	G	H	U	W	B	C	X	J
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

DS

Linked Representation of Binary Trees

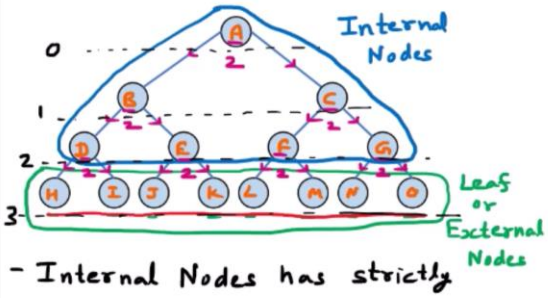




DS

Types of Binary Tree

Perfect Binary Tree



Largest Level = 3

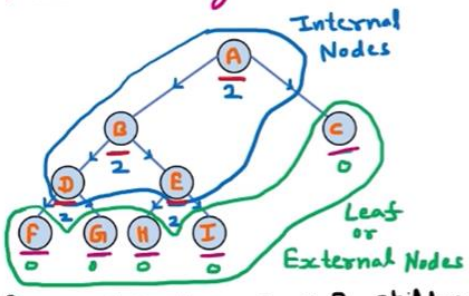
Height (Depth) = Largest Level + 1
= 3 + 1
= 4

- Internal Nodes has strictly 2 children & External Nodes are at same Level

Perfect Binary Tree of Height h has $2^h - 1$ Nodes

$2^4 - 1 = 16 - 1 = 15$

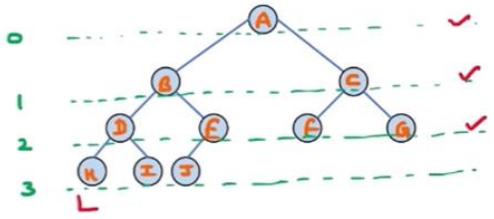
Full Binary Tree



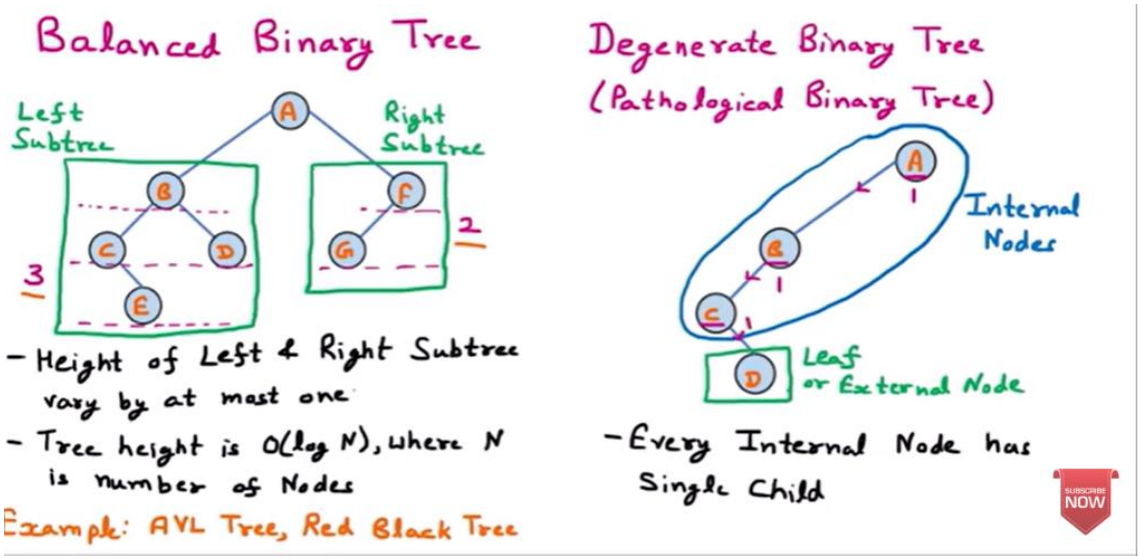
- Each Node has 0 or 2 children
- Each Node except External Nodes has 2 Children

Leaf Nodes = Internal Nodes + 1
5 4

Complete Binary Tree

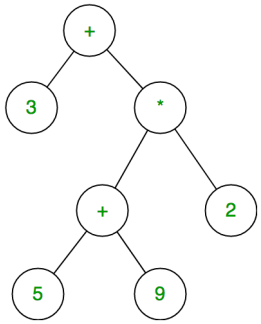


- All levels filled with Nodes except lowest level & lowest level nodes reside on left side



Expression Tree (binary)

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:



Construction of Expression Tree:

Now for constructing an expression tree, we use a stack. We loop through input expressions and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

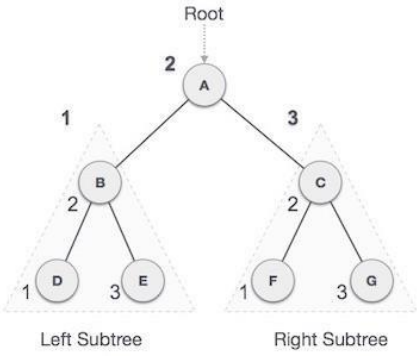
In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes based on their values. In a binary tree, the

elements are arranged in the order they arrive at the tree from top to bottom and left to right.

Different methods of tree traversal

Inorder (left subtree root right subtree)

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.



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 in-order 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.

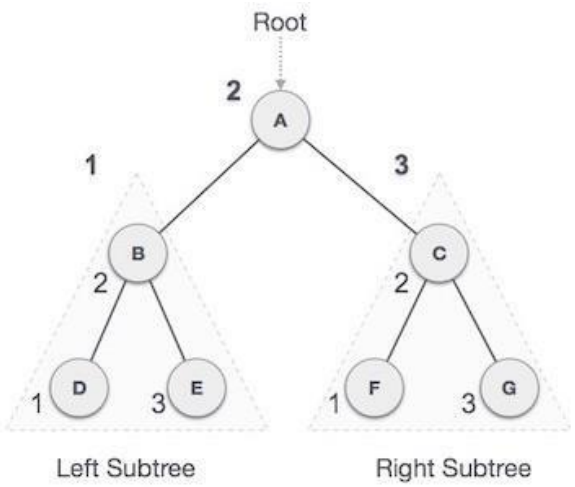
Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Pre-order Traversal (root node left subtree right subtree)

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

Preorder traversal is also used to get prefix expressions on an expression tree.



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 −

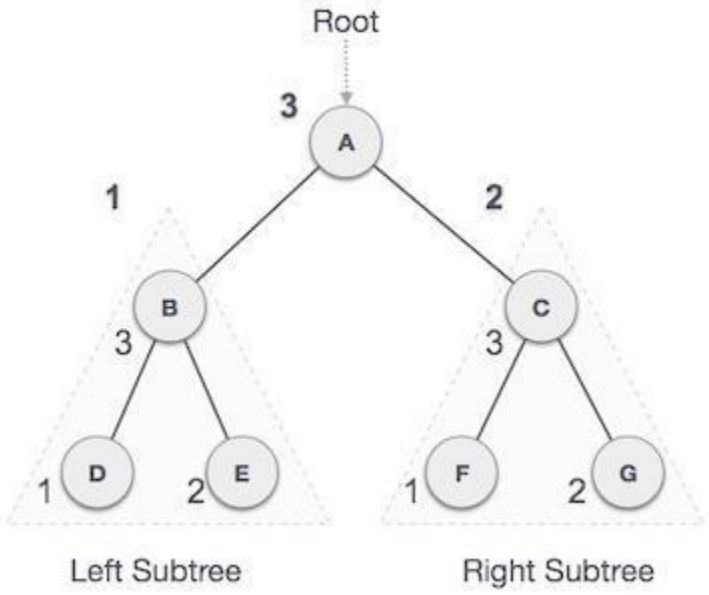
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order Traversal (left subtree right subtree root node)

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.



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.

Pre-order traversal

```
template <class T>
Void preOrder (binaryTreeNode <T> * t)
{
   if (t != NULL)
   {
      Visit (t);   // visit tree root
      preOrder (t -> leftchild); // do left subtree
      preOrder (t -> rightchild); // do right subtree
   }
}
```

In order Traversal

```
template <class T>
Void inOrder (binaryTreeNode <T> * t)
{
   if (t != NULL)
   {
      inOrder (t -> leftchild) ; // do left subtree
      Visit (t) ; // visit tree root
      inOrder (t -> rightchild) ; // do right subtree
   }
}
```

Post traversal.

```
template <class T>
Void postOrder (binaryTreeNode <T> * t)
{
   if (t != NULL)
   {
      postOrder (t -> leftchild) ;
      postOrder (t -> rightchild) ;
      Visit (t) ;
   }
}
```

```
Visit function.
template <class T>
Void Visit (binaryTreeNode<T> * x)
{
   Cout << x -> element;
}
```

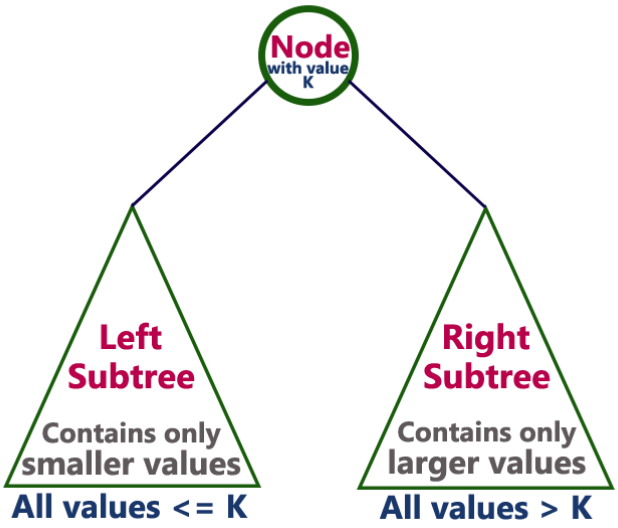
A binary tree has the following time complexities.

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

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

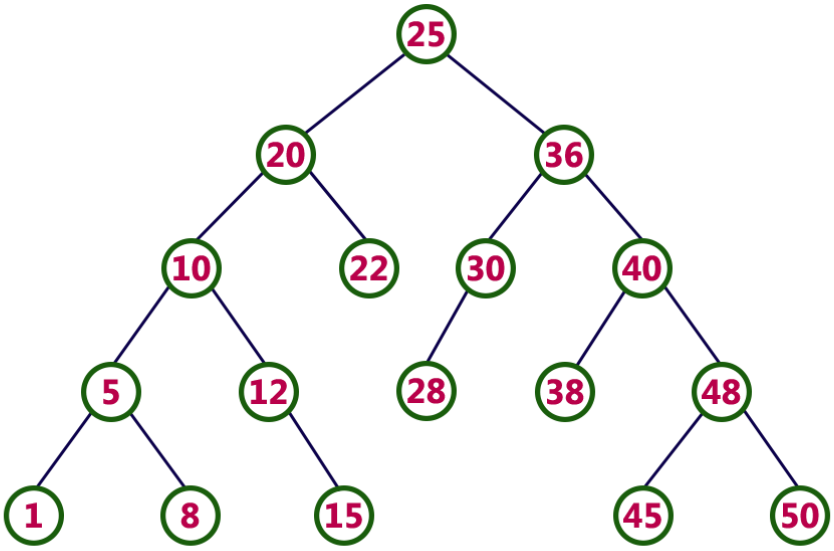
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contain smaller values and all the nodes in the right subtree of any node contain larger values as shown in the 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 every binary tree need not be binary search tree.

Operations on a Binary Search Tree

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

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

Search Operation in BST

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

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

Insertion Operation in BST

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

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

Deletion Operation in BST

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

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: Deleting a leaf node

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

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

Case 2: Deleting a node with one child

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

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

Case 3: Deleting a node with two children

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

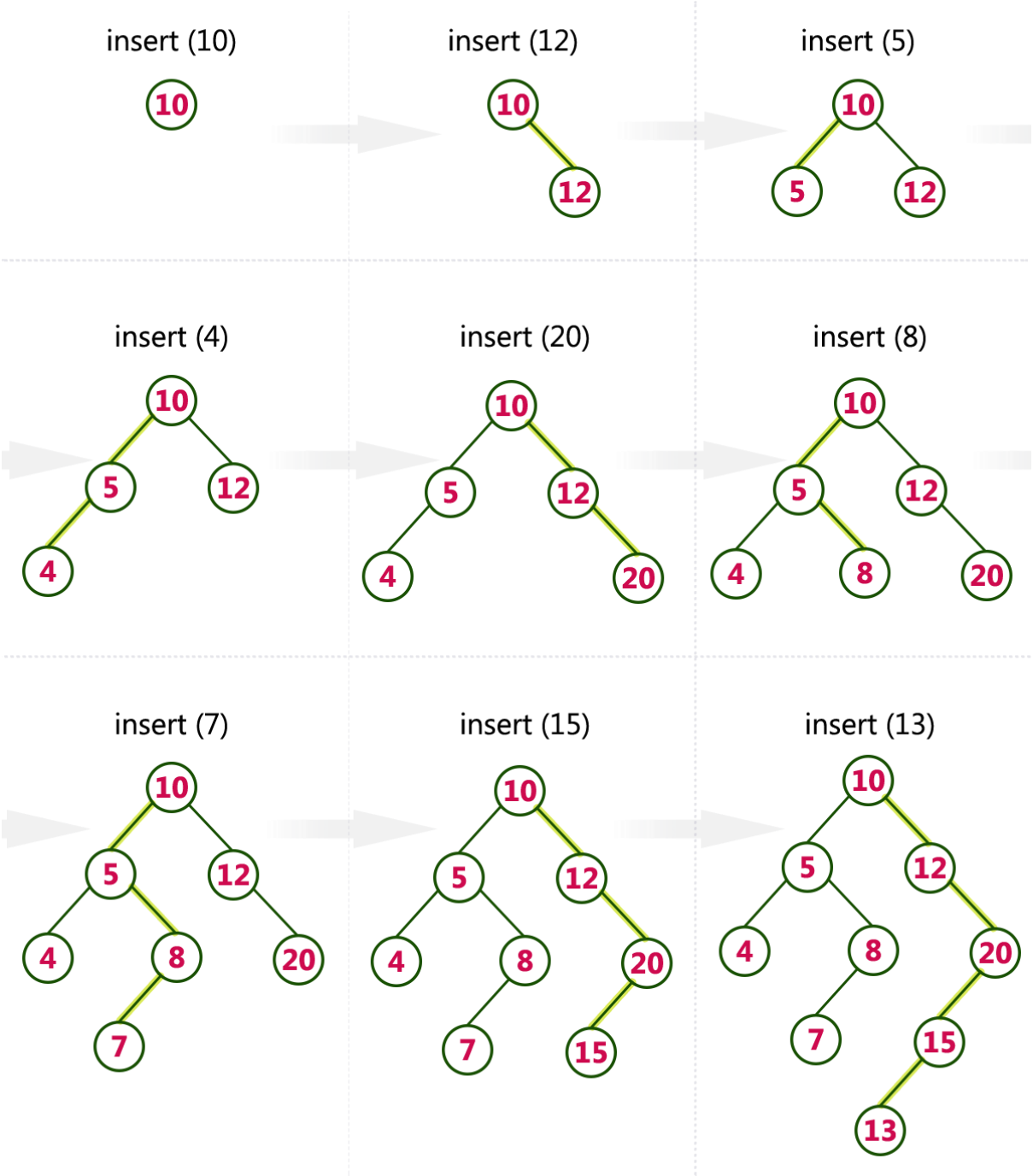
- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3 - Swap** both **deleting node** and node which is found in the above step.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

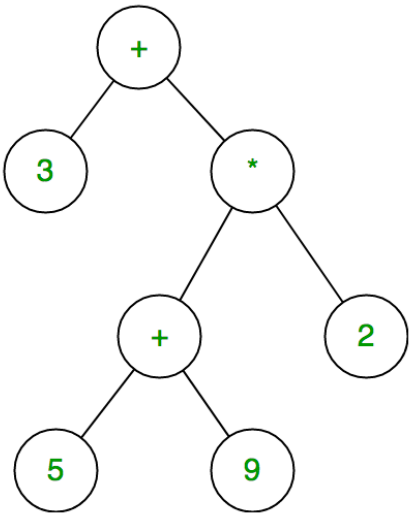
Example

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

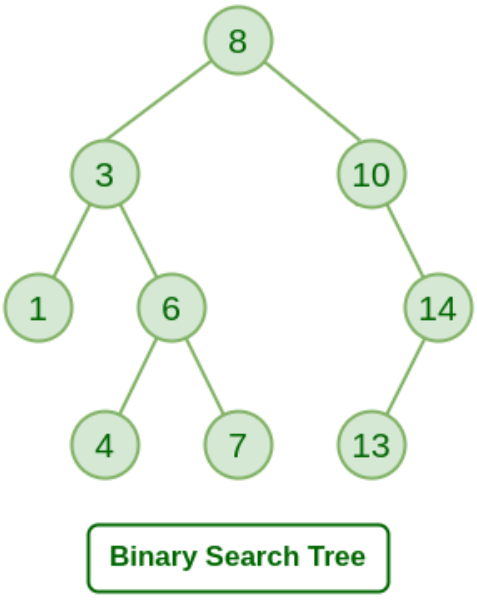
10,12,5,4,20,8,7,15 and 13

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





Inorder traversal of expression tree produces infix version of given postfix expression (same with postorder traversal it gives postfix expression)

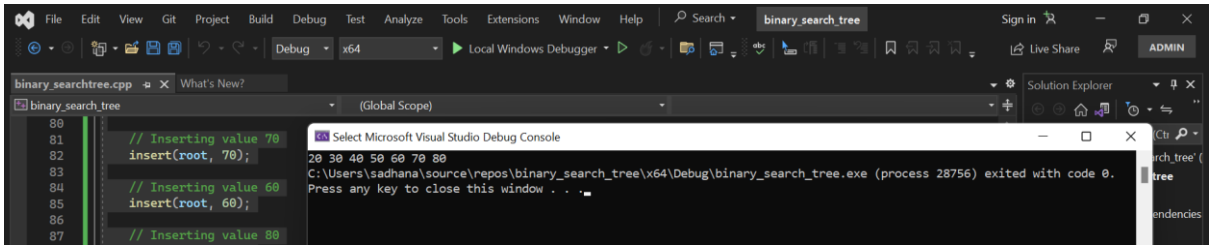



```
// C++ function to search a given key in a given BST
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
```

```
    return search(root->left, key);
}


// C++ program to insert a node
// in a BST
#include <iostream>
using namespace std;

// Given Node
struct node
{
    int key;
    struct node* left, * right;
};

// Function to create a new BST node
struct node* newNode(int item)
{
    struct node* temp = new node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to insert a new node with
// given key in BST
struct node* insert(struct node* node, int key)
{

    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node->key)
    {
        node->left = insert(node->left, key);
    }
    else if (key > node->key)
    {
        node->right = insert(node->right, key);
    }

    // Return the node pointer
    return node;
}

// Function to do inorder traversal of BST
void inorder(struct node* root)
{
    if (root != NULL)
    {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

// Driver Code
int main()
{
```

```
    /* Let us create following BST
              50
           /     \
          30      70
         / \    / \
        20  40  60  80
    */
    struct node* root = NULL;

    // Inserting value 50
    root = insert(root, 50);

    // Inserting value 30
    insert(root, 30);

    // Inserting value 20
    insert(root, 20);

    // Inserting value 40
    insert(root, 40);

    // Inserting value 70
    insert(root, 70);

    // Inserting value 60
    insert(root, 60);

    // Inserting value 80
    insert(root, 80);

    // Print the BST
    inorder(root);

    return 0;
}
```

Output



Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

```
    // C++ program to delete
    // a node of BST
    #include <iostream>
    using namespace std;
```

```
// Given Node node
struct node {
    int key;
    struct node* left, * right;
};

// Function to create a new BST node
struct node* newNode(int item)
{
    struct node* temp
        = (struct node*)malloc(
            sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to insert a new node with
// given key in BST
struct node* insert(struct node* node, int key)
{
    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node->key) {
        node->left = insert(node->left, key);
    }
    else if (key > node->key) {
        node->right = insert(node->right, key);
    }

    // Return the node pointer
    return node;
}

// Function to do inorder traversal of BST
void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        cout << " " << root->key;
        inorder(root->right);
    }
}

// Function that returns the node with minimum
// key value found in that tree
struct node* minValueNode(struct node* node)
{
    struct node* current = node;

    // Loop down to find the leftmost leaf
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

// Function that deletes the key and
// returns the new root
struct node* deleteNode(struct node* root,
```

```
    int key)
{
    // base Case
    if (root == NULL)
        return root;

    // If the key to be deleted is
    // smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key) {
        root->left
            = deleteNode(root->left, key);
    }

    // If the key to be deleted is
    // greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key) {

        root->right
            = deleteNode(root->right, key);
    }

    // If key is same as root's key,
    // then this is the node
    // to be deleted
    else {

        // Node with only one child
        // or no child
        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children:
        // Get the inorder successor(smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's
        // content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right
            = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver Code
int main()
{
    /* Let us create following BST
              50
           /     \
```

```
        30      70
       / \    / \
      20  40  60  80
    */
    struct node* root = NULL;

    // Creating the BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Function Call
    root = deleteNode(root, 60);
    inorder(root);

    return 0;
}
```

```
Enter the number of nodes to be insert: 6

Please enter the numbers to be insert: 2 6 9 1 5 8

Binary Search Tree nodes in Inorder Traversal:   1  2  5  6  8  9

Process returned 0 (0x0)   execution time : 15.837 s
Press any key to continue.
```