**303105201 - Design of Data Structures**

# UNIT-5 Trees

->Trees

Terminology

Binary Trees

Properties of Binary trees

Array and linked Representation of Binary Trees

->Binary Tree Traversals - In Order, Post Order, Pre Order

Additional Binary tree operations

Threaded binary trees

->Binary Search Trees –

Definition

Insertion

Deletion

Traversal

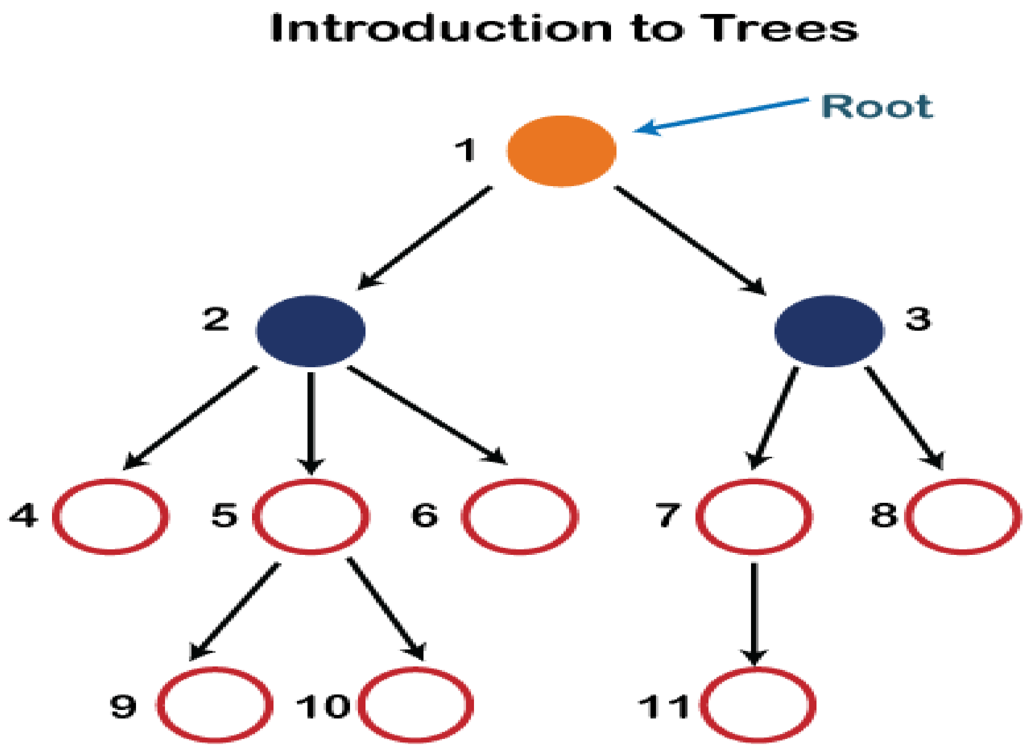
Searching

Application of Trees-Evaluation of Expression

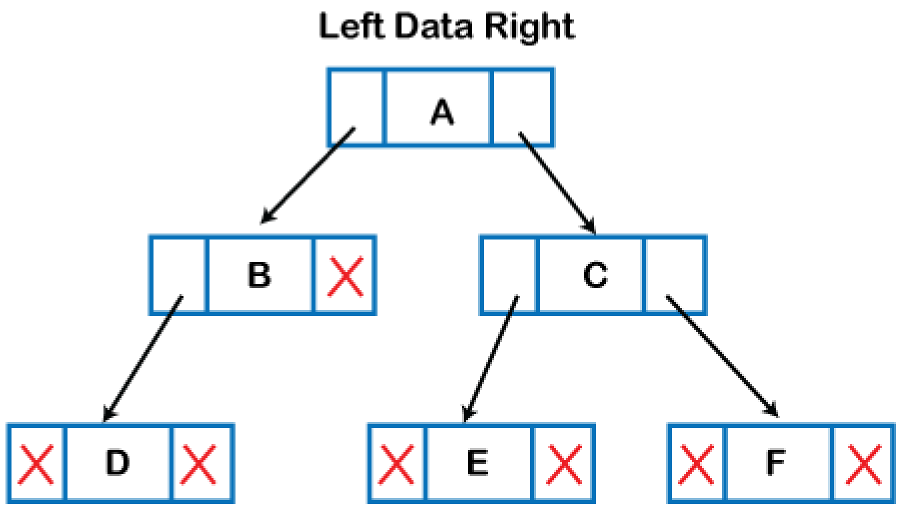
**TREE**

- A tree is also one of the data structures that represent hierarchical data.
- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children

*Some basic terms used in Tree data structure*



*Implementation of Tree*



*Applications of trees:*

The following are the applications of trees:

*Storing naturally hierarchical data:* Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.

*Organize data:* It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.

*Trie:* It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.

*Heap:* It is also a tree data structure implemented using arrays. It is used to implement priority queues.

*B-Tree and B+Tree:* B-Tree and B+Tree are the tree data structures used to implement indexing in databases.

*Routing table:* The tree data structure is also used to store the data in routing tables in the routers.

*Node:* A basic unit of a tree that contains data and references (pointers) to other nodes. In C, nodes are often represented using structures.

```
struct TreeNode {
   int data;
   struct TreeNode *left;
   struct TreeNode *right;
};
```

*Root:* The topmost node of a tree. It serves as the starting point for accessing other nodes in the tree.

```
struct TreeNode *root = NULL;
```

*Parent Node:* A node that has child nodes directly connected to it.

*Child Node:* Nodes directly connected to another node (its parent).

*Leaf Node:* A node that has no children.

*Sibling Nodes:* Nodes that share the same parent.

*Depth of a Node:* The distance from the root to the node, measured by the number of edges (or levels).

*Height of a Node:* The distance from a node to its deepest descendant (leaf node). In terms of edges, it's the number of edges on the longest path from the node to a leaf.

*Internal nodes:* A node has at least one child node known as an internal

*Ancestor node:* An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

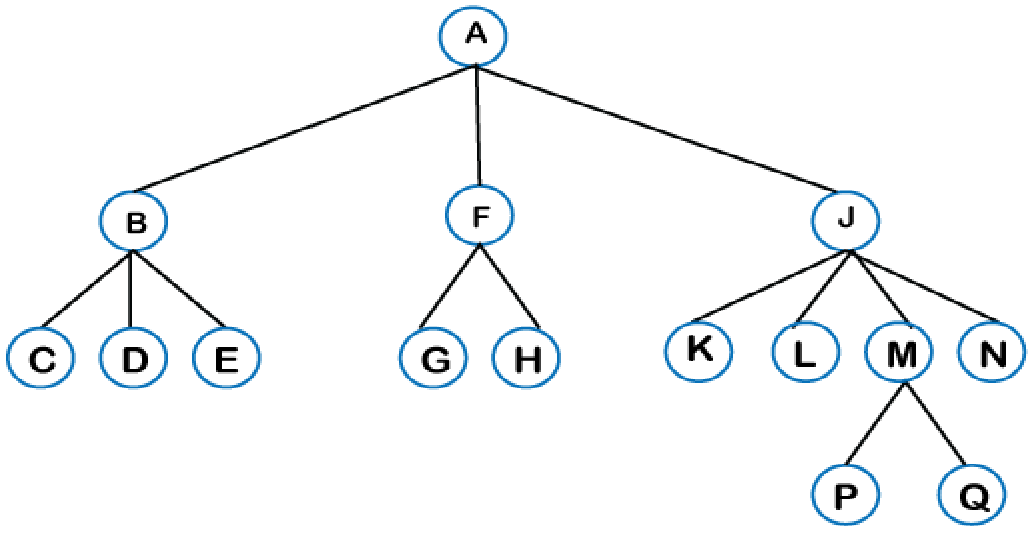
*Descendant:* The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

**Types of Tree data structure**

The following are the types of a tree data structure:

- General tree
- Binary tree
- Binary Search tree
- AVL tree
- Red-Black Tree

*General tree:* The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as subtrees.

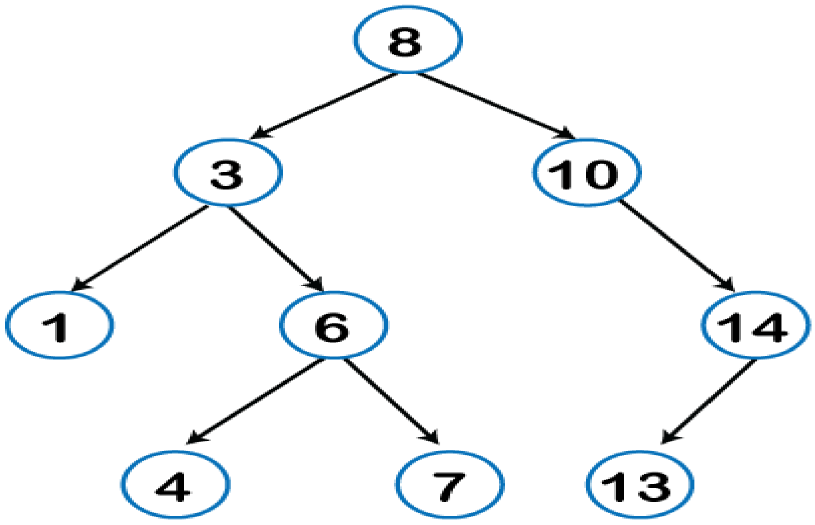


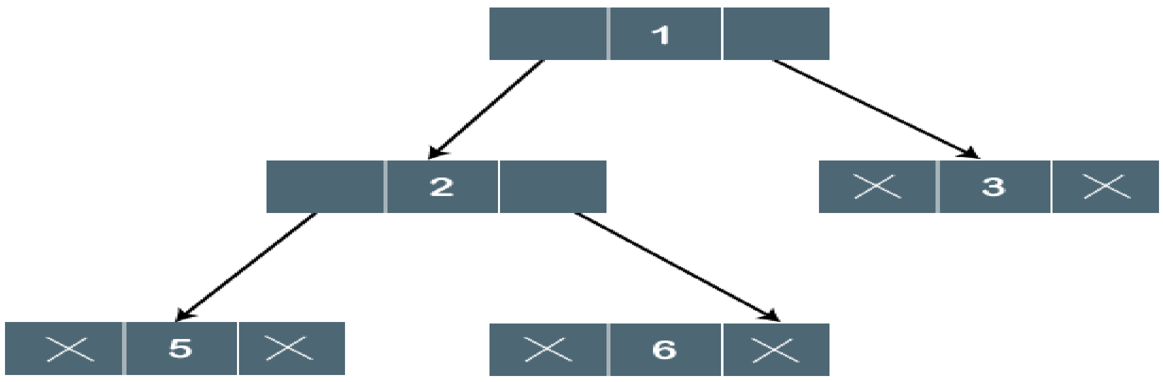
**Binary Tree**

*Definition of a Binary Tree:*

A tree in which each node has at most two children: left child and right child. A binary tree is a hierarchical data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child. The topmost node in the tree is called the root.

```
struct TreeNode {

    int data;

    struct TreeNode *left;

    struct TreeNode *right;

};
```





*Key Properties:*

*Rooted Structure:* A binary tree has a root node from which all other nodes are descendants. This root node has no parent.

*Node Structure:* Each node in a binary tree contains:

- Data (or value) associated with the node.
- Reference (or pointer) to the left child.
- Reference (or pointer) to the right child.

*Child Nodes:*

- Each node in a binary tree can have at most two children.
- These children are ordered; there is a specific left child and a specific right child associated with each node.

*Depth and Height:*

*Depth:* The depth of a node is the number of edges from the root to that node.

*Height:* The height of a node is the number of edges on the longest path from that node down to a leaf node. The height of the tree itself is the height of the root node.

The maximum height can be computed as:

As we know that,

$n = h+1$

$h= n-1$

*Insertion:* Adding a new node to a tree. This involves finding the correct position based on the node's value and linking it appropriately.

```
struct TreeNode* insertNode(struct TreeNode* root, int data) {
   if (root == NULL) {
       return createNode(data); // Assume createNode function is defined
   }
   if (data < root->data) {
       root->left = insertNode(root->left, data);
   } else if (data > root->data) {
       root->right = insertNode(root->right, data);
   }
   return root;
}
```

*Deletion:* Removing a node from a tree while maintaining the properties of the tree structure (e.g., binary search tree property in binary search trees).

*Search:* Finding a specific node (or its data) within a tree.

```
struct TreeNode* searchNode(struct TreeNode* root, int key) {
    if (root == NULL || root->data == key) {
        return root;
    }
    if (key < root->data) {
        return searchNode(root->left, key);
    }
    return searchNode(root->right, key);
}
```

*Traversal:* The process of visiting all nodes in a tree in a specific order. Common types of tree traversal include in-order, pre-order, and post-order traversals.

## Array Representation of Binary Trees

In an array representation, the binary tree is stored in a linear array, where nodes are placed in a specific order. This approach is efficient in terms of space when the tree is complete (every level, except possibly the last, is completely filled)

Mapping Nodes to Array Indices:

*For a node at index $i$ in the array:

-Its left child is at index $2*i+1$

-Its right child is at index $2*i+2$

*Advantages:*

- Simple to implement for complete binary trees.
- Requires less memory overhead compared to linked representation.

*Disadvantages:*

- Wasteful for sparse trees or trees that are not complete.
- Insertion and deletion operations can be complex and may require shifting elements.

Example Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100  // Define a maximum size for the array
// Function to get the left child index
int leftChild(int i) {
   return 2 * i + 1;
}
// Function to get the right child index
int rightChild(int i) {
   return 2 * i + 2;
}
// Function to get the parent index
int parent(int i) {
   return (i - 1) / 2;
}
// Function to print the binary tree array
void printTree(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
// Main function to demonstrate the array representation
```

```
int main() {
    int tree[MAX_SIZE] = {1, 2, 3, 4, 5}; // Example binary tree array
    int size = 5; // Number of elements in the tree
    printf("Binary Tree Representation:\n");
    printTree(tree, size);
    // Example: Accessing nodes
    printf("Root node: %d\n", tree[0]);
    printf("Left child of node at index 0: %d\n", tree[leftChild(0)]);
    printf("Right child of node at index 0: %d\n", tree[rightChild(0)]);
    printf("Parent of node at index 4: %d\n", tree[parent(4)]);
    return 0;
}
```

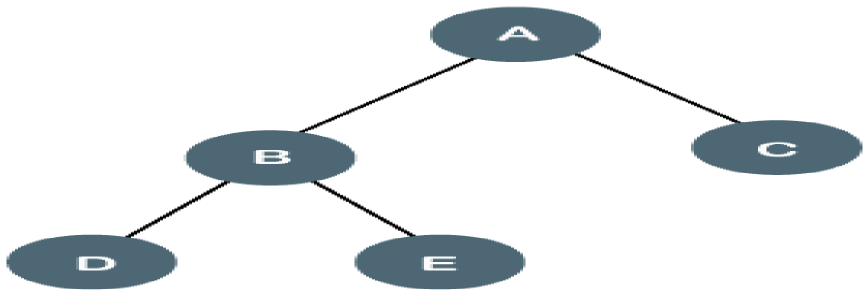
*Usage and Applications:*

Binary trees are fundamental in computer science and are used for:

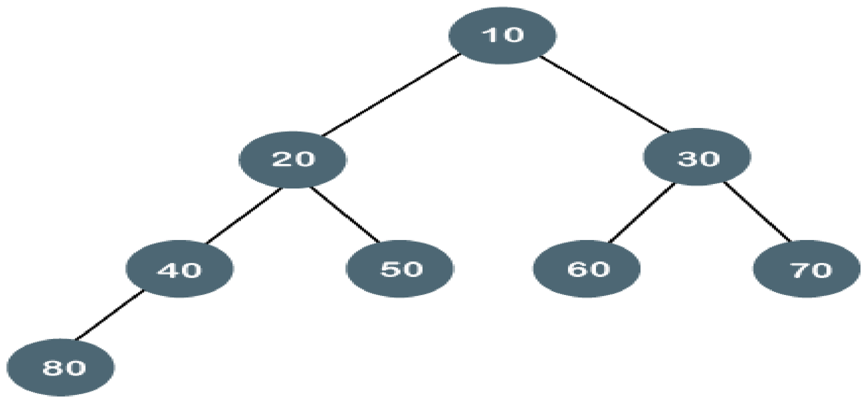
- Representing hierarchical data (e.g., file systems, organizational structures).
- Efficient searching and sorting operations (e.g., binary search trees).
- Expression parsing and evaluation (e.g., arithmetic expressions).
- Routing algorithms in networks (e.g., binary tree-based routing).

***Types of Binary Trees:***

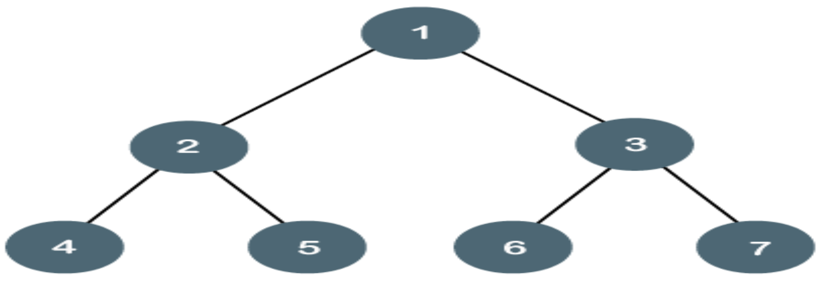
- *Full / proper/ strict Binary Tree:* Every node other than the leaves has two children.



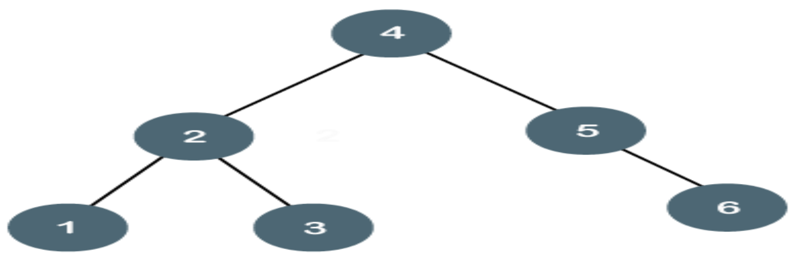
- *Complete Binary Tree:* Every level of the tree is fully filled, except possibly for the last level, which is filled from left to right.



- *Perfect Binary Tree:* A binary tree where all interior nodes have two children and all leaves have the same depth or same level.



- *Balanced Binary Tree:* A binary tree is balanced if the height of the left and right subtrees of any node differ by no more than one.



**Binary Tree Traversals - In Order, Post Order, Pre Order**

The term 'tree traversal' means traversing or visiting each node of a tree. There are multiple ways to traverse a tree that are listed as follows

- Preorder traversal
- Inorder traversal
- Postorder traversal

*Preorder traversal*

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

In preorder traversal, nodes are visited in the following order:

1. Root node
2. Left subtree
3. Right subtree

The applications of preorder traversal include -

- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

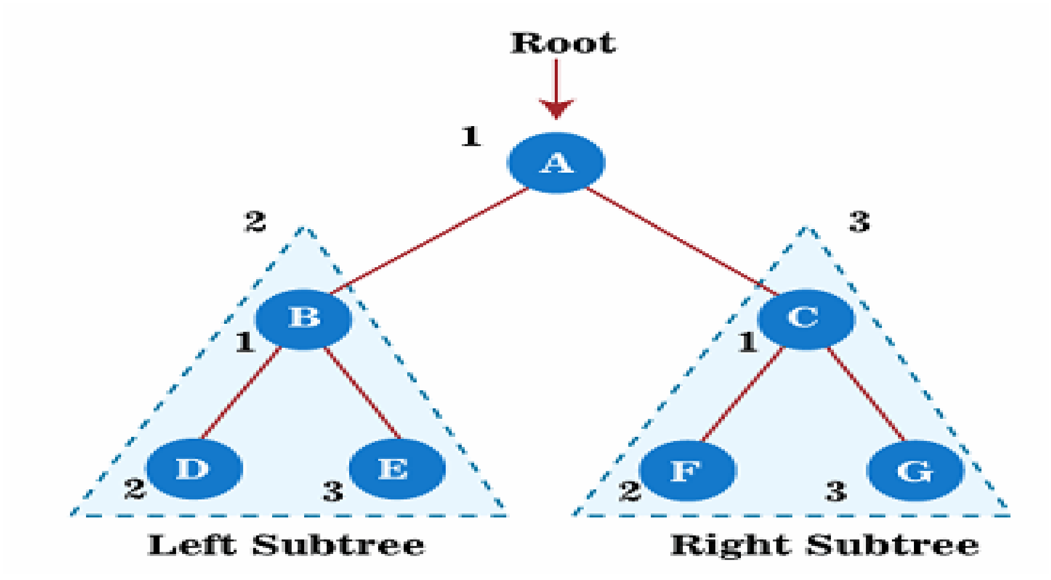
*Algorithm*

Until all nodes of the tree are not visited

Step 1 - Visit the root node

Step 2 - Traverse the left subtree recursively.

Step 3 - Traverse the right subtree recursively.



The output of the preorder traversal of the above tree is -

A → B → D → E → C → F → G

*<u>Program:</u>*

```
// Function to perform preorder traversal of a binary tree
void preorderTraversal(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }
    printf("%d ", root->data);       // Visit root node
    preorderTraversal(root->left);   // Visit left subtree
    preorderTraversal(root->right);  // Visit right subtree
}
// Main function (continued from previous example)
```

```
int main() {
   // Constructing the binary tree (continued from previous example)


   // Perform preorder traversal
   printf("Preorder traversal: ");
   preorderTraversal(root);
   printf("\n");
   return 0;
}
```

*Inorder traversal*

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

In inorder traversal, nodes are visited in the following order:

1. Left subtree
2. Root node
3. Right subtree

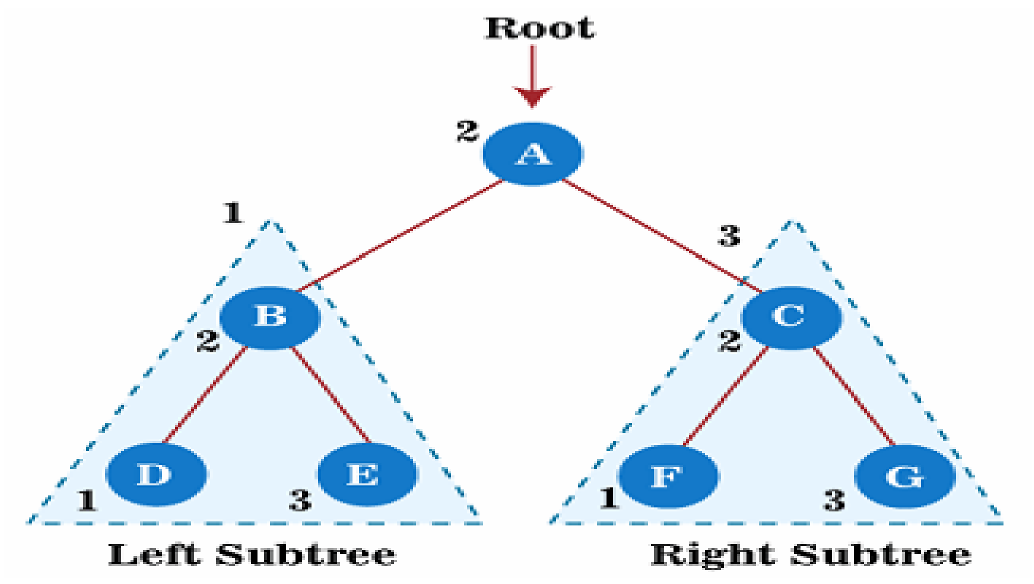
The applications of Inorder traversal includes -

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

*Algorithm*

```
Until all nodes of the tree are not visited

Step 1 - Traverse the left subtree recursively.

Step 2 - Visit the root node.
```

Step 3 - Traverse the right subtree recursively.



The output of the inorder traversal of the above tree is -

D → B → E → A → F → C → G

*Program*

```
#include <stdio.h>

#include <stdlib.h>

// Definition of a binary tree node

struct TreeNode {

    int data;

    struct TreeNode* left;

    struct TreeNode* right;

};

// Function to perform inorder traversal of a binary tree

void inorderTraversal(struct TreeNode* root) {

    if (root == NULL) {
        return;
    }
    inorderTraversal(root->left);    // Visit left subtree
    printf("%d ", root->data);       // Visit root node
    inorderTraversal(root->right);   // Visit right subtree
}
// Function to create a new node
struct TreeNode* newNode(int data) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct
TreeNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
// Main function
int main() {
    // Constructing a binary tree
    struct TreeNode* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    // Perform inorder traversal
    printf("Inorder traversal: ");
    inorderTraversal(root);
    printf("\n");
    return 0;
}
```

*Postorder traversal*

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

In postorder traversal, nodes are visited in the following order:

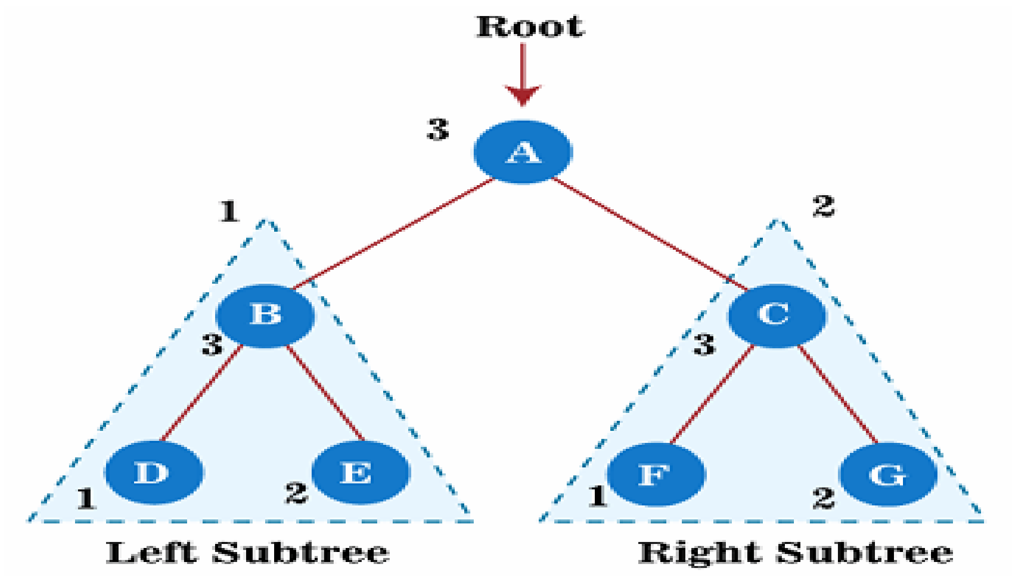
1. Left subtree
2. Right subtree
3. Root node

The applications of postorder traversal include -

- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

*Algorithm*

```
Until all nodes of the tree are not visited
Step 1 - Traverse the left subtree recursively.
Step 2 - Traverse the right subtree recursively.
Step 3 - Visit the root node.
```



The output of the postorder traversal of the above tree is -

D → E → B → F → G → C → A

*Program:*

```
// Function to perform postorder traversal of a binary tree
void postorderTraversal(struct TreeNode* root) {
    if (root == NULL) {
        return;
    }
    postorderTraversal(root->left);   // Visit left subtree
    postorderTraversal(root->right);  // Visit right subtree
    printf("%d ", root->data);       // Visit root node
}
// Main function (continued from previous example)
int main() {
    // Constructing the binary tree (continued from previous example)
    // Perform postorder traversal
    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");
    return 0;
}
```

*Complexity of Tree traversal techniques*

- The time complexity of tree traversal techniques discussed above is O(n), where 'n' is the size of binary tree.
- The space complexity of tree traversal techniques discussed above is O(1) if we do not consider the stack size for function calls. Otherwise, the space complexity of these techniques is O(h), where 'h' is the tree's height.

*Program on binary tree traversal (inorder,preorder,postorder)*

```
#include <stdio.h>
#include <stdlib.h>
// Definition of a binary tree node
struct TreeNode {
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
};
// Function to create a new node
struct TreeNode *newNode(int data) {
    struct TreeNode *node = (struct TreeNode
*)malloc(sizeof(struct TreeNode));
    if (node == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
// Function to perform preorder traversal of a binary tree
void preorderTraversal(struct TreeNode *root) {
    if (root == NULL) {
        return;
    }
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
// Function   to perform inorder traversal of a binary tree
void inorderTraversal(struct TreeNode *root) {
    if (root == NULL) {
        return;
    }
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}
// Function to perform postorder traversal   of a binary tree
void postorderTraversal(struct TreeNode *root) {
    if (root == NULL) {
        return;
    }
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d  ", root->data);
}
// Main function
int main(){
    // Constructing the binary tree
    struct TreeNode *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    // Perform traversals
    printf("Preorder traversal: ");
    preorderTraversal(root);
    printf("\n");
    printf("Inorder traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");
    // Free memory
    // ... (implement a function to free the entire tree)
    return 0;
}
```

**Additional Binary tree operations**

*1. Search Operation*

The search operation is used to find a specific node with a given key (or data) in the binary tree.

*Iterative Approach:*

```
struct TreeNode* search(struct TreeNode* root, int key) {
  while (root != NULL && root->data != key) {
    if (key < root->data)
      root = root->left;
    else
      root = root->right;
  }
```

```
    return root;  // Returns NULL if key not found
}
```

*Recursive Approach:*

```
struct TreeNode* search(struct TreeNode* root, int key) {
    if (root == NULL || root->data == key)
        return root;
    if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}
```

*2. Insertion Operation*

The insertion operation adds a new node with a specified key into the binary tree while maintaining the binary search tree (BST) property.

*Iterative Approach:*

```
struct TreeNode* insert(struct TreeNode* root, int key) {
    struct TreeNode* newNode = newNode(key);
    if (root == NULL)
        return newNode;
    struct TreeNode* current = root;
    struct TreeNode* parent = NULL;
    while (current != NULL) {
        parent = current;
        if (key < current->data)
            current = current->left;
        else
            current = current->right;
    }
    if (key < parent->data)
        parent->left = newNode;
    else
        parent->right = newNode;
    return root;
}
```

*Recursive Approach:*

```
struct TreeNode* insert(struct TreeNode* root, int key) {
    if (root == NULL)
        return newNode(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
        root->right = insert(root->right, key);
    return root;
}
```

## *3. Deletion Operation*

The deletion operation removes a node with a specified key from the binary tree while maintaining the binary search tree (BST) property.

Case 1: Node to be deleted has no children (leaf node):

```
struct TreeNode* deleteNode(struct TreeNode* root, int key) {
   if (root == NULL)
      return root;
   if (key < root->data)
      root->left = deleteNode(root->left, key);
   else if (key > root->data)
      root->right = deleteNode(root->right, key);
   else {
      // Node with only one child or no child
      if (root->left == NULL) {
         struct TreeNode* temp = root->right;
         free(root);
         return temp;
      } else if (root->right == NULL) {
         struct TreeNode* temp = root->left;
         free(root);
         return temp;
      }
      // Node with two children: Get the inorder successor (smallest in the right subtree)
      struct TreeNode* temp = minValueNode(root->right);
```

```
        root->data = temp->data;

        root->right = deleteNode(root->right, temp->data);

    }

    return root;

}
```

Utility function to find the inorder successor (smallest node in right subtree):

```
struct TreeNode* minValueNode(struct TreeNode* node) {

    struct TreeNode* current = node;

    while (current->left != NULL)

        current = current->left;

    return current;

}
```

*4. Height of Binary Tree*

The height of a binary tree is the number of edges on the longest path from the root node to a leaf node.

Recursive Approach:

```
int height(struct TreeNode* root) {

    if (root == NULL)

        return 0;

    int leftHeight = height(root->left);

    int rightHeight = height(root->right);

    return 1 + max(leftHeight, rightHeight);

}
```

```
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

*5. Counting Nodes*

Counting the number of nodes in a binary tree.

Recursive Approach:

```
int countNodes(struct TreeNode* root) {
    if (root == NULL)
        return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}
```

*6. Checking if a Binary Tree is a Binary Search Tree (BST)*

A binary tree is a BST if for each node, its left subtree contains nodes with keys less than the node's key, and its right subtree contains nodes with keys greater than the node's key.

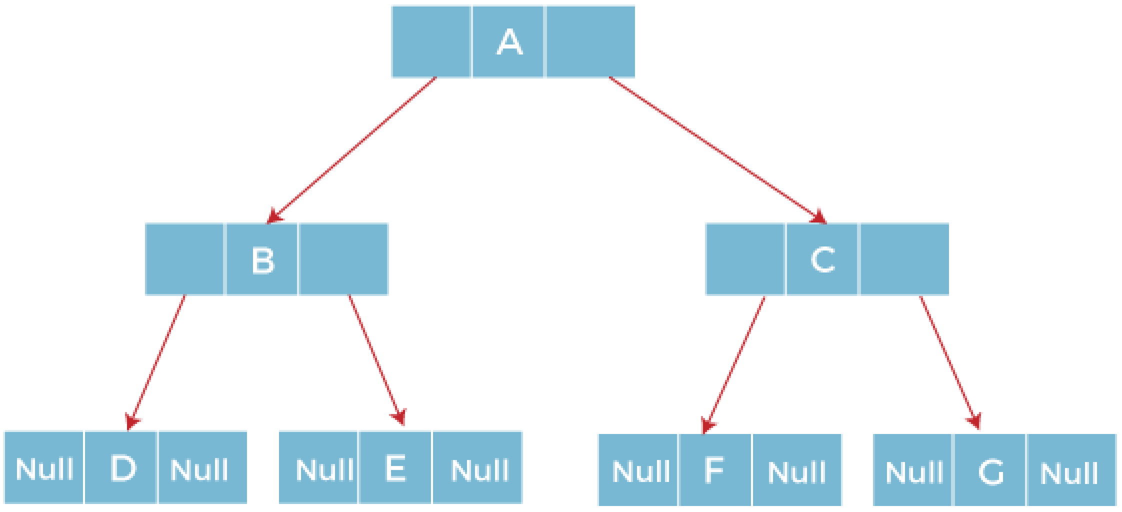
*Recursive Approach:*

```
int isBST(struct TreeNode* root) {
    return isBSTUtil(root, INT_MIN, INT_MAX);
}
int isBSTUtil(struct TreeNode* root, int min, int max) {
    if (root == NULL)
        return 1;
    if (root->data < min || root->data > max)
```

```
        return 0;
    return isBSTUtil(root->left, min, root->data - 1) &&
           isBSTUtil(root->right, root->data + 1, max);
}
```

**Threaded Binary Tree**

If a binary tree consists of n nodes then n+1 link fields contain NULL values. In which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as threaded binary trees. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.



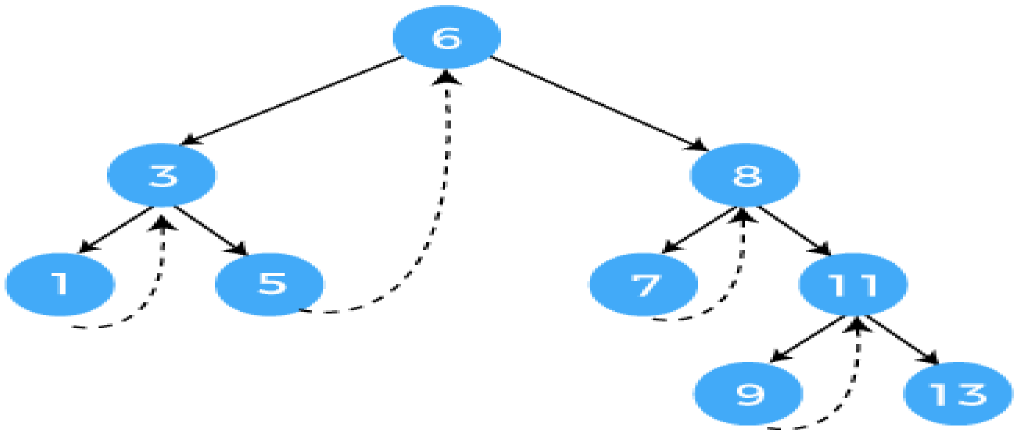
Threaded binary tree

Types of Threaded Binary Tree

There are two types of threaded Binary Tree:

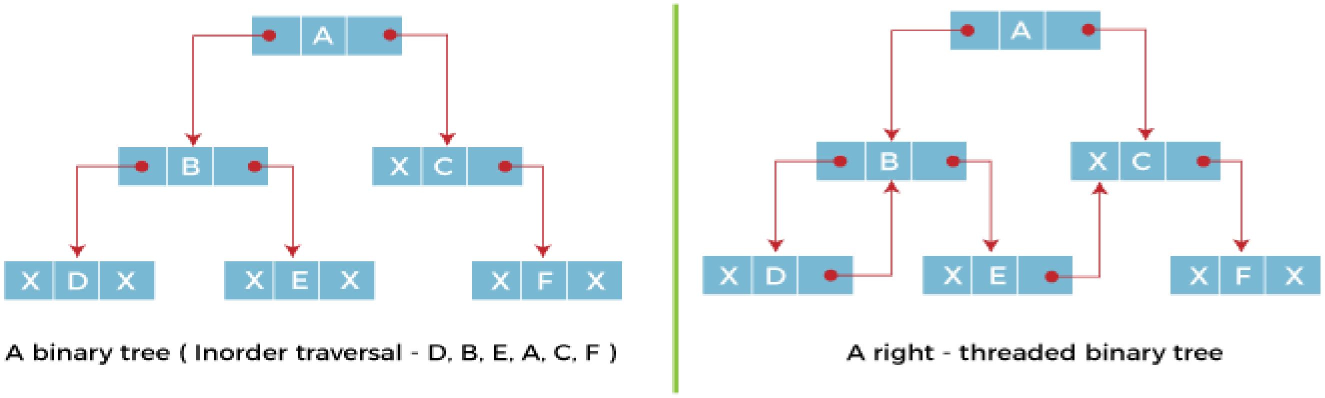
- One-way threaded Binary Tree
- Two-way threaded Binary Tree

One-way threaded Binary trees:



Single Threaded Binary Tree

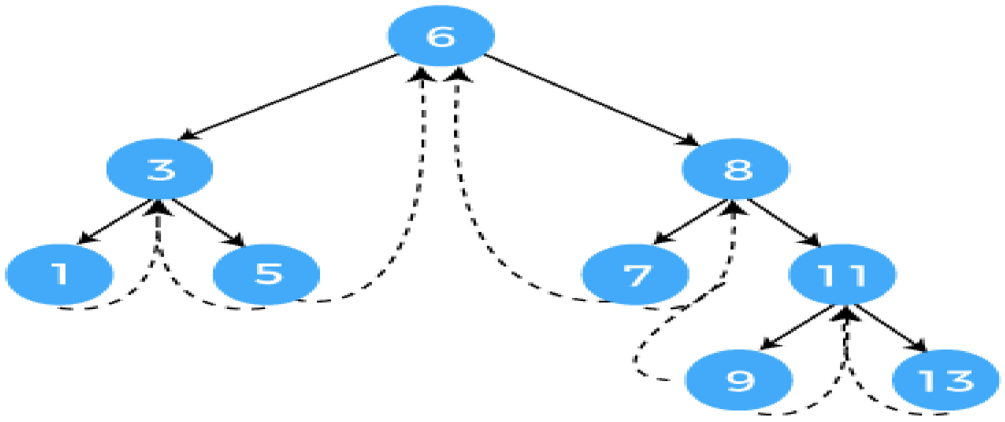
- In one-way threaded binary trees, a thread will appear either in the right or left link field of a node.
- If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called Right threaded binary trees.
- If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called Left threaded binary trees.



A binary tree ( Inorder traversal - D, B, E, A, C, F )

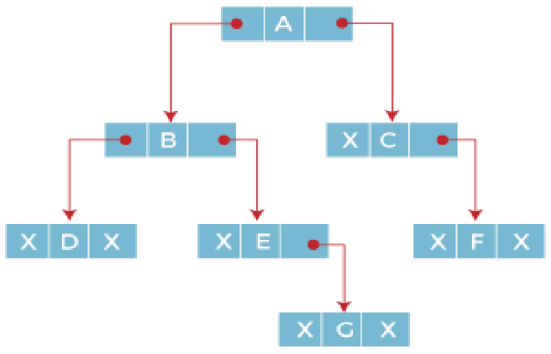
A right - threaded binary tree

*Two-way threaded Binary Trees:*

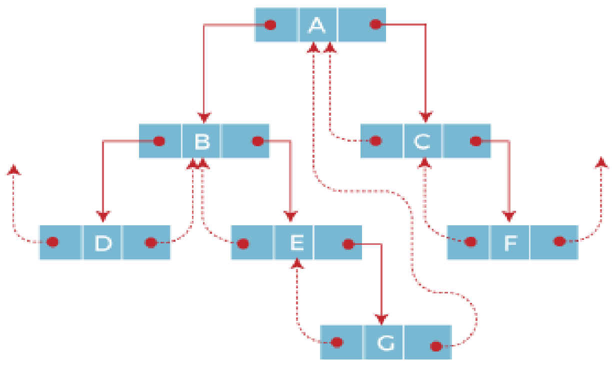


Double Threaded Binary Tree

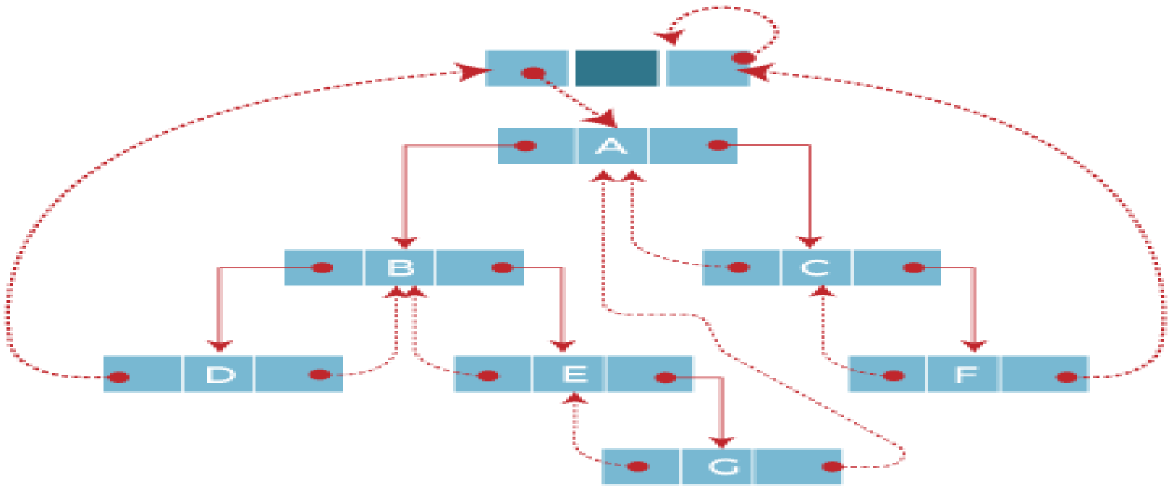
In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.



A binary tree ( Inorder traversal - D, B, E, G, A, C, F )



A two - way threaded binary tree



Two-way threaded - tree with header node

## *Advantages of Threaded Binary Trees*

*Efficient Traversal:* Inorder traversal can be performed in O(n) time without recursion or using an explicit stack.

*Saves Space:* Eliminates the need for a stack used in traditional recursive traversal methods.

*Simplifies Implementation:* Once threads are properly set, traversal becomes straightforward and efficient.

## *Disadvantages of Threaded Binary Trees*

*Complexity in Insertion and Deletion:* Inserting or deleting nodes requires careful management of threads to maintain the tree structure and thread consistency.

*Additional Space Overhead:* Requires an additional flag (isThreaded) in each node to indicate whether the right pointer is a thread.

## *Difference between Threaded BT and Normal BBT*

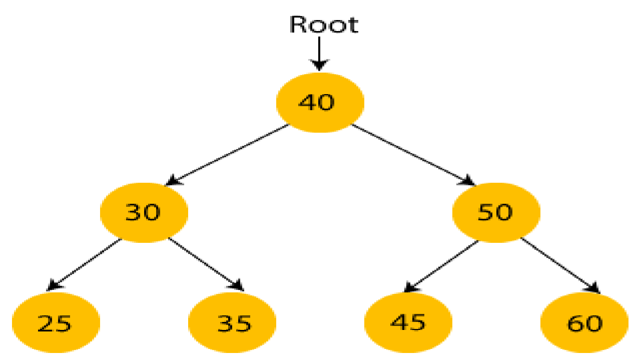
| Threaded BT | Normal BBT |
|---|---|
| • In threaded binary tree, The null pointers are used as thread. | In a normal binary trees,the null pointers remains null. |
| • We canusee the null pointers which is a efficient way to use computers memory. | We can't use null pointers so it is a wastage of memeory. |
| • Traversal is easy.Completed without using stack or recursive function. | Traverse is not easy and not memory efficient. |
| • Structure is complex. | Less complex that threaded binary tree |

| | |
|---|---|
| ● Insertion and deletion takes more time | Less Time consuming than Threaded Binary tree |

**Binary search tree**

<u>**Definition** of a Binary search tree:</u>

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.



A Binary Search Tree (BST) is a node-based data structure where each node has at most two children, referred to as the left and right child.

<u>The properties of a BST are:</u>

- 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 subtrees each must also be a binary search tree.

Advantages of Binary search tree:

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

**Insertion** in a Binary search tree:

-First, we have to insert (say X element) into the tree as the root of the tree.

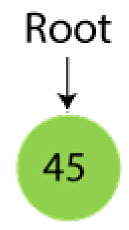
-Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.

-Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Example:

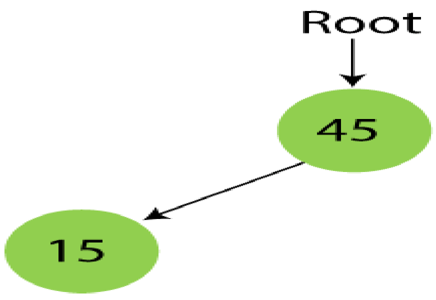
The data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50

Step 1 - Insert 45



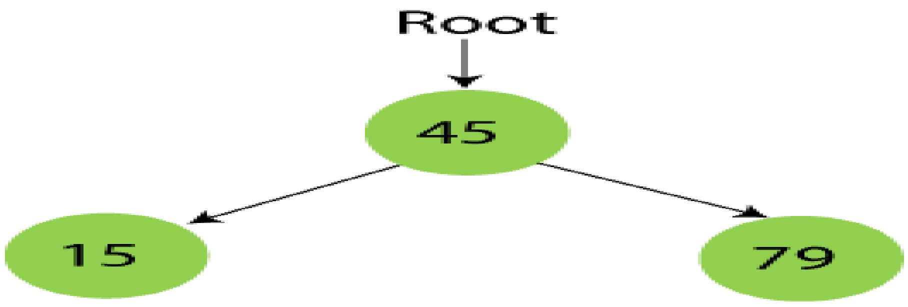
Step 2 - Insert 15

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



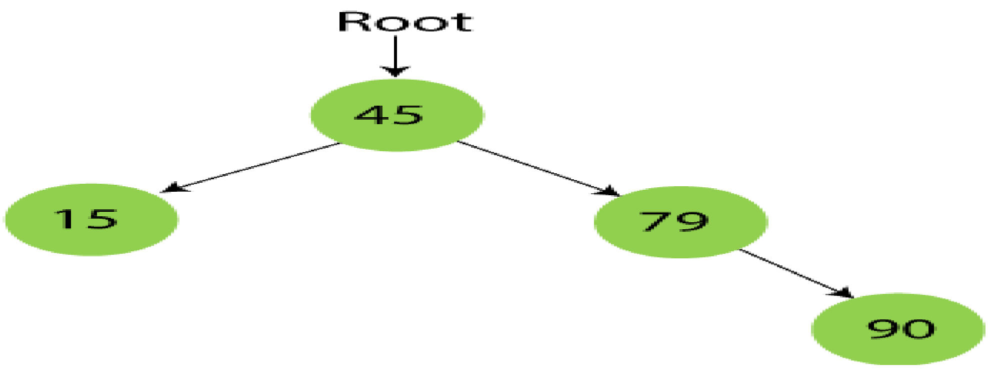
Step 3 - Insert 79

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



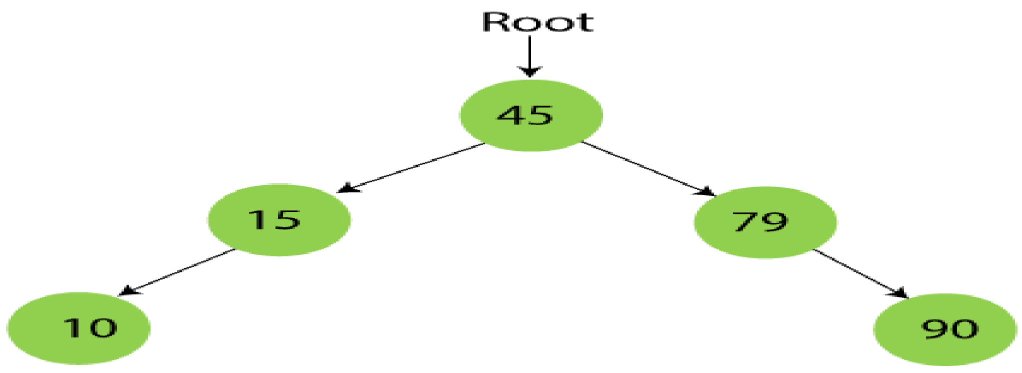
Step 4 - Insert 90

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



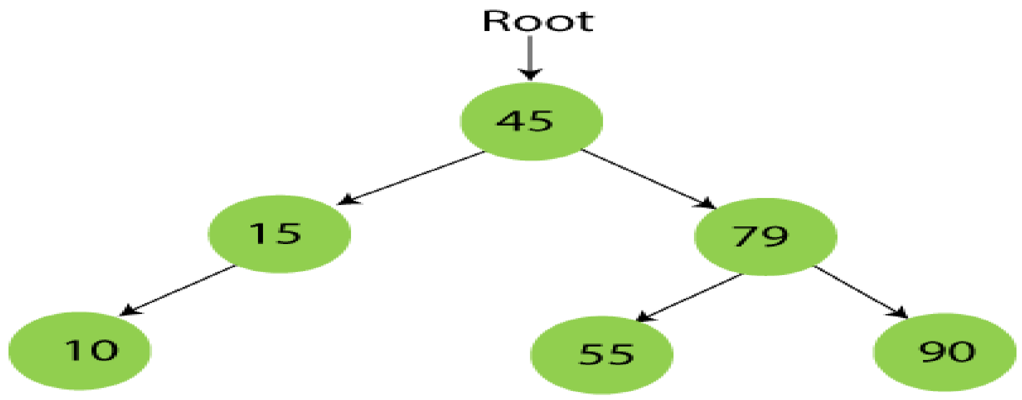
Step 5 - Insert 10

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



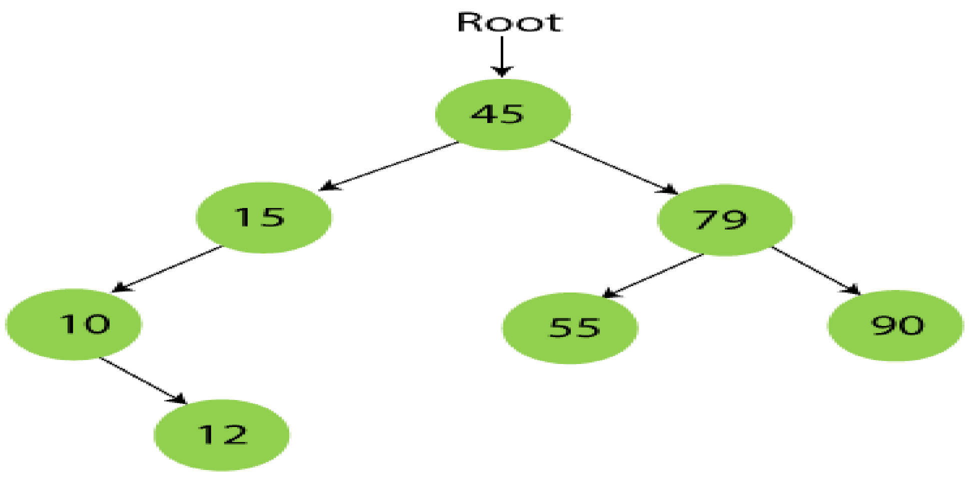
Step 6 - Insert 55

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



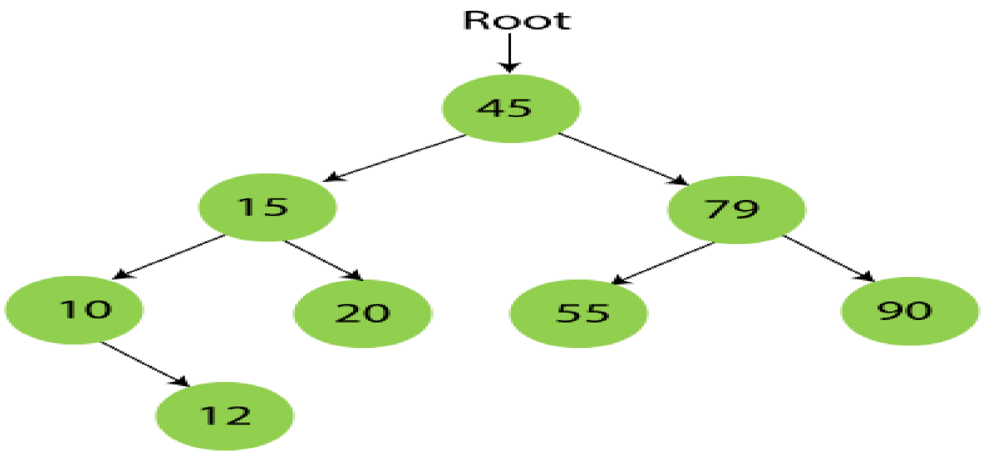
Step 7 - Insert 12

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



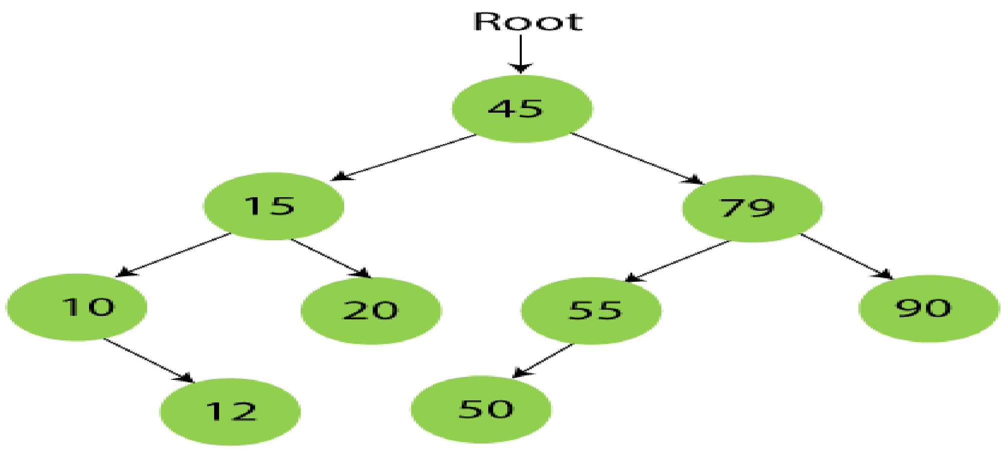
Step 8 - Insert 20

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



Step 9 - Insert 50

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



**<u>Deletion</u>** <u>in Binary Search tree</u>

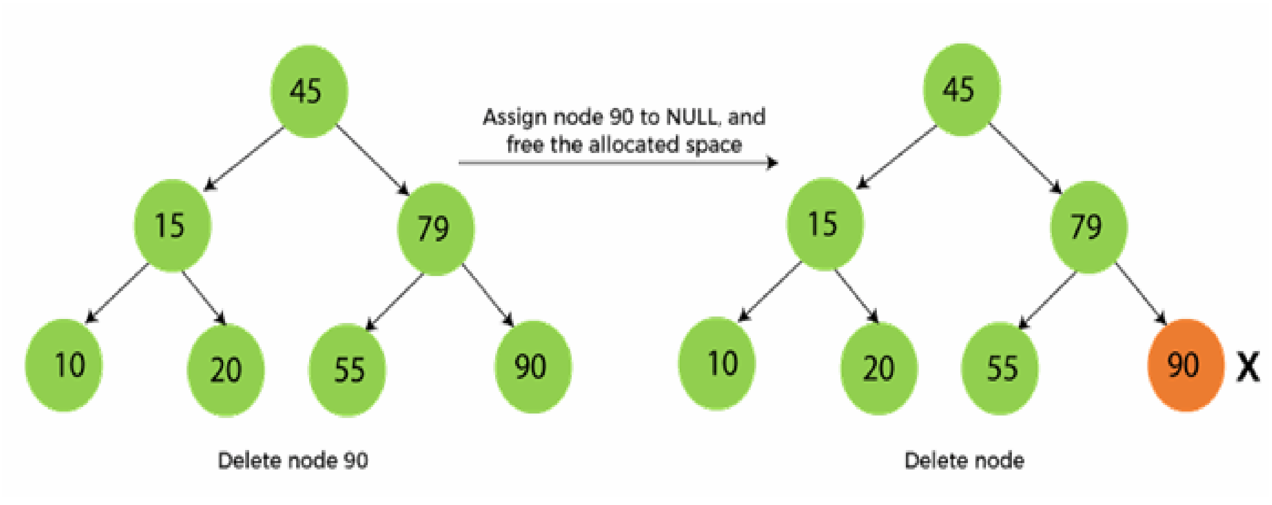
In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated.

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

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

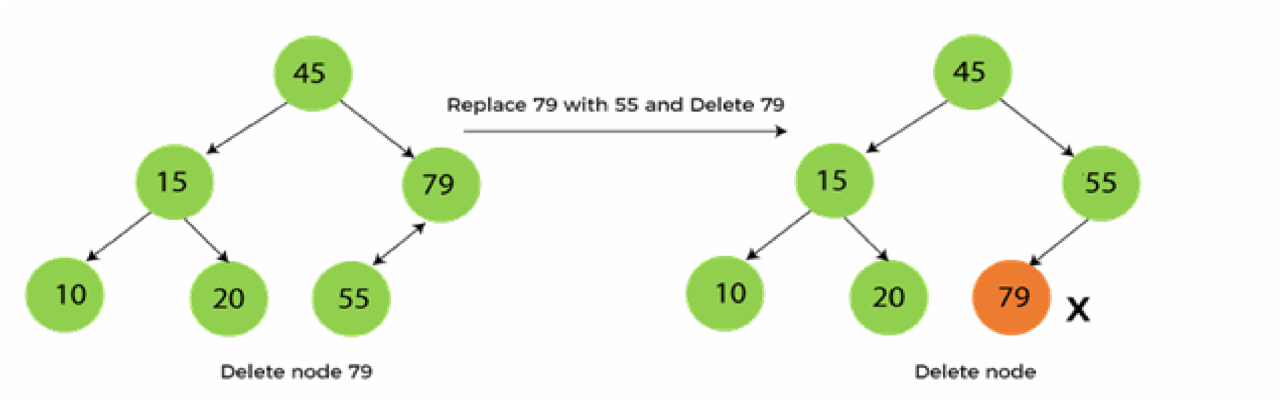
*<u>When the node to be deleted is the leaf node?</u>*

we have to replace the leaf node with NULL and simply free the allocated space.



*<u>When the node to be deleted has only one child?</u>*

we have to replace the target node with its child, and then delete the child node.



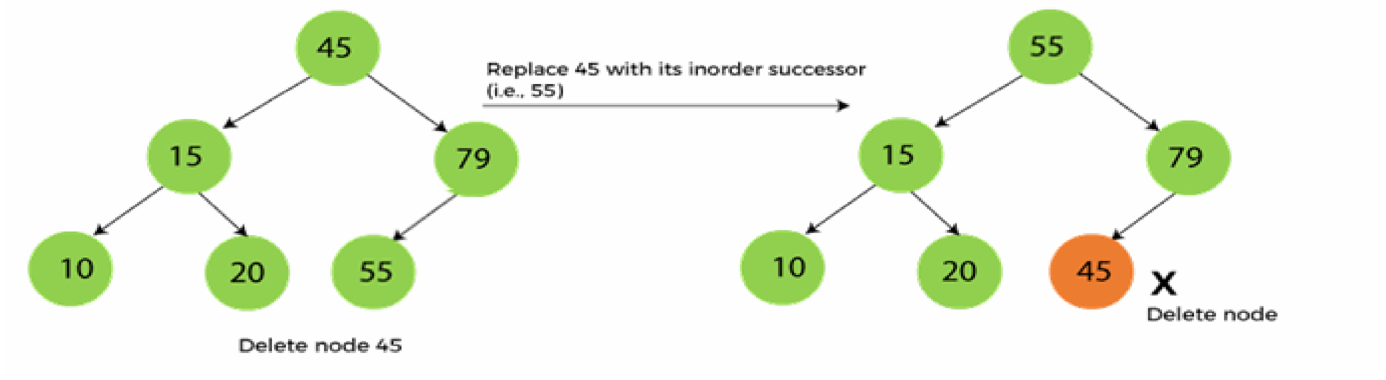
*<u>When the node to be deleted has two children?</u>*

The steps to be followed are listed as follows -

-First, find the inorder successor of the node to be deleted.

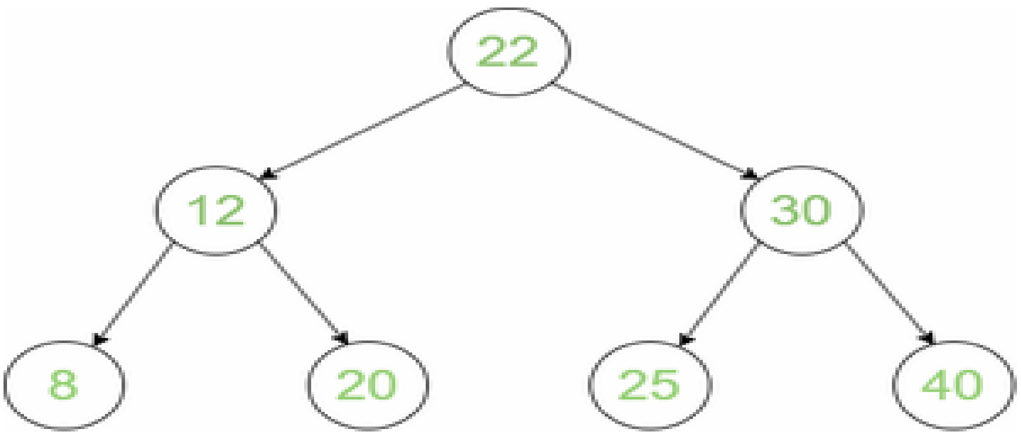
-After that, replace that node with the inorder successor(next) until the target node is placed at the leaf of tree.

-And at last, replace the node with NULL and free up the allocated space.



**Traversals** in binary search tree:

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



Inorder Traversal: 8 12 20 22 25 30 40

Preorder Traversal: 22 12 8 20 30 25 40

Postorder Traversal: 8 20 12 25 40 30 22

*Inorder Traversal:*

Follow the below steps to implement the idea:

- Traverse left subtree
- Visit the root and print the data.
- Traverse the right subtree

*Preorder Traversal:*

Follow the below steps to implement the idea:

- Visit the root and print the data.
- Traverse left subtree
- Traverse the right subtree

*Postorder Traversal:*

Follow the below steps to implement the idea:

- Traverse left subtree
- Traverse the right subtree
- Visit the root and print the data.

**Searching** in Binary search tree

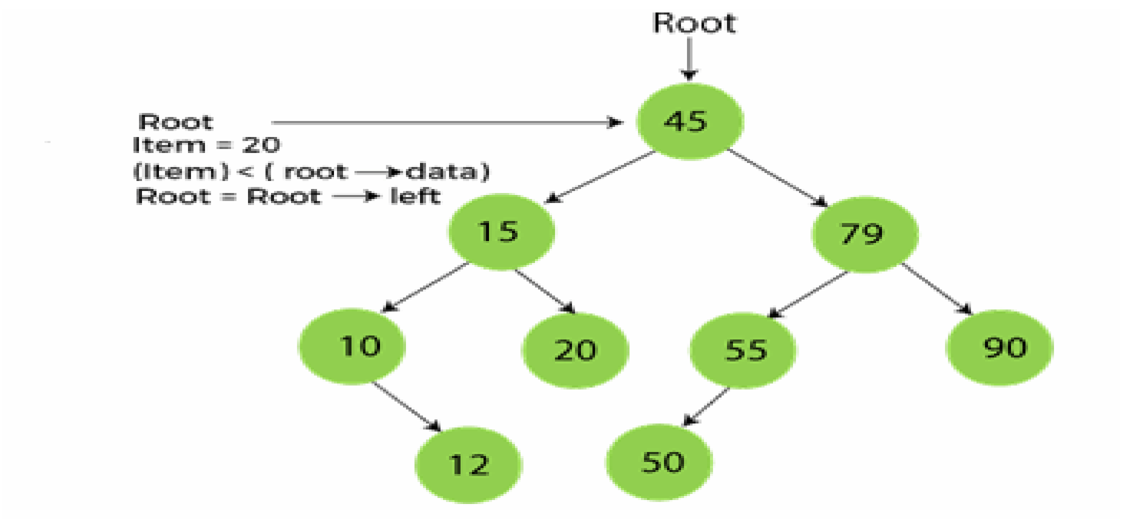
Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order.

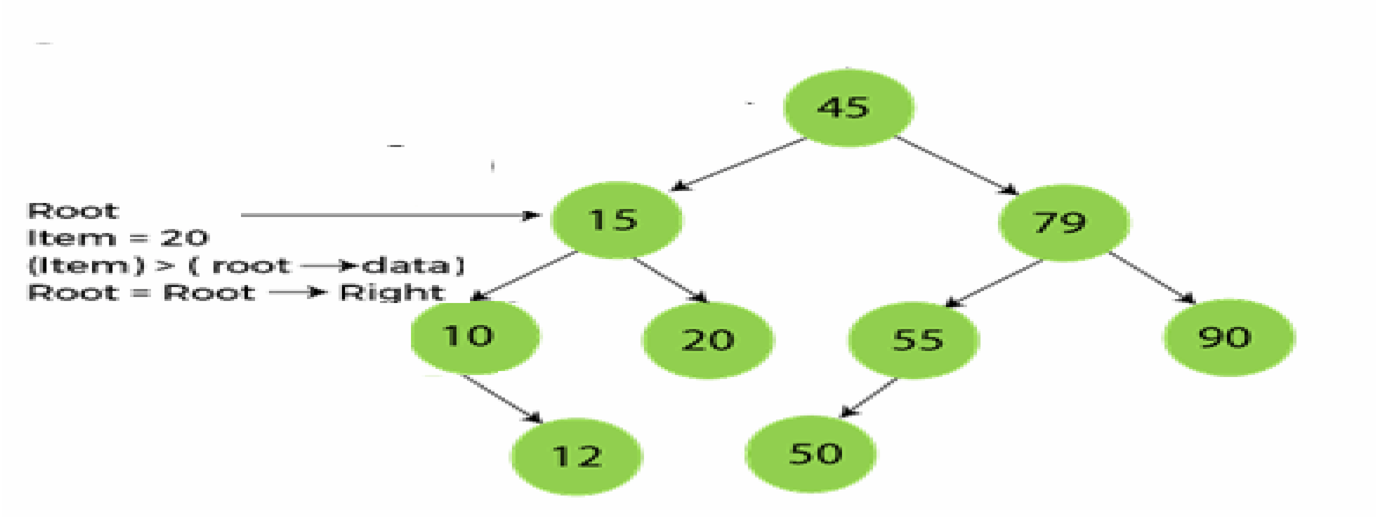
The steps of searching a node in Binary Search tree are listed as follows -

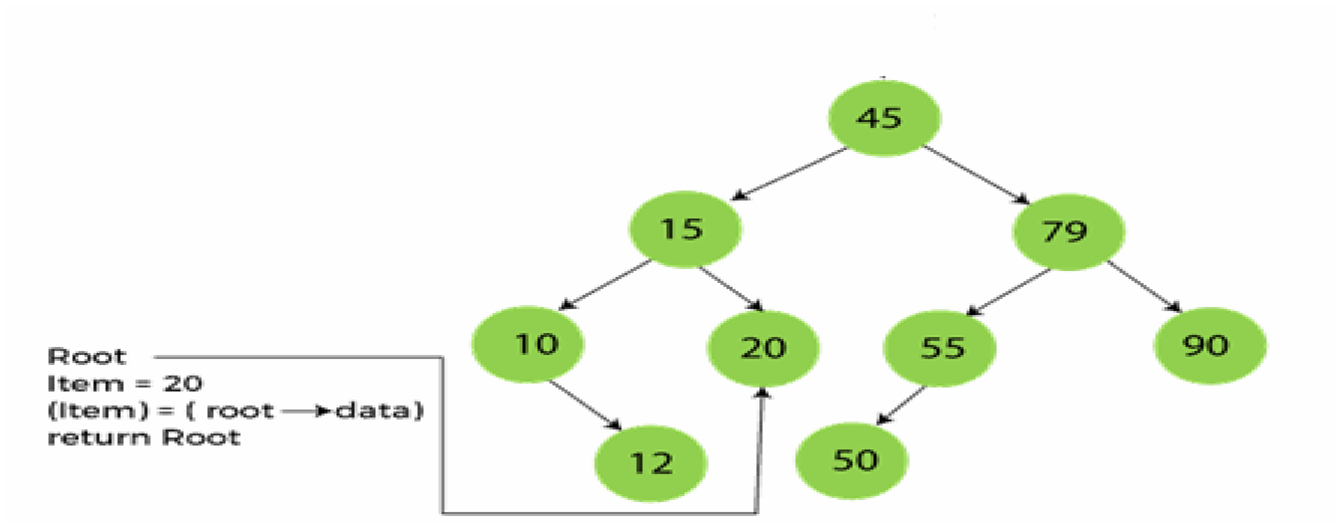
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.

4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

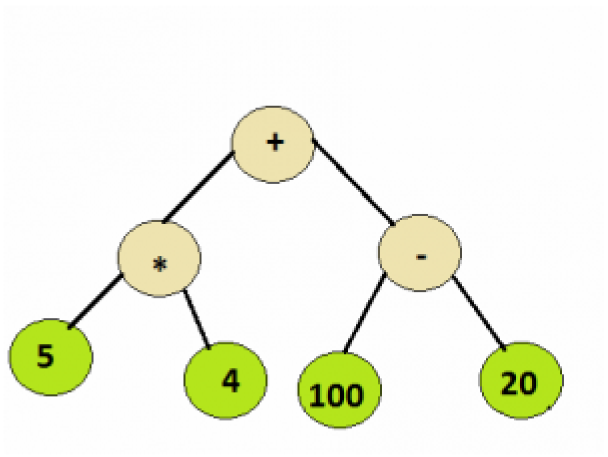
Example:







*Application of Trees-Evaluation of Expression:*



An expression tree is a binary tree used to represent arithmetic expressions. Each internal node represents an operator, and each leaf node represents an operand.

*Example:*

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

// Node definition
```

```
typedef struct Node {
    char op;  // Operator for internal nodes, or '\0' for leaf nodes
    int value; // Operand value for leaf nodes
    struct Node* left;
    struct Node* right;
} Node;
// Function to create a new node
Node* createNode(char op, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->op = op;
    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;


}
// Function to determine if a character is an operator
int isOperator(char c) {    return c == '+' || c == '-' || c == '*' || c ==
'/';
}
// Construct the expression tree from postfix expression
Node* constructTree(char* postfix) {
    Node* stack[100]; // Adjust size as needed
    int top = -1;
    }
    char* token = strtok(postfix, " ");
    while (token != NULL) {
        if (isOperator(token[0])) {
            Node* node = createNode(token[0], 0);
            node->right = stack[top--];
            node->left = stack[top--];
            stack[++top] = node;
        } else {
            Node* node = createNode('\0', atoi(token));
            stack[++top] = node;
        }
        token = strtok(NULL, " ");
    }
    return stack[top];
}
// Evaluate the expression tree
int evaluate(Node* root) {
    if (root == NULL) return 0;
    // Leaf node
    if (root->op == '\0') {
        return root->value;
    }
    // Internal node
    int leftEval = evaluate(root->left);
    int rightEval = evaluate(root->right);
    switch (root->op) {
        case '+': return leftEval + rightEval;
        case '-': return leftEval - rightEval;
        case '*': return leftEval * rightEval;
        case '/': return leftEval / rightEval;
        default: return 0;
    }
}
// Free the memory of the tree nodes
void freeTree(Node* root) {
    if (root == NULL) return;
    freeTree(root->left);
    freeTree(root->right);
    free(root);
}
int main() {
    char postfix[] = "3 4 5 * +"; // Example postfix expression
    Node* root = constructTree(postfix);
    printf("Result: %d\n", evaluate(root));
    freeTree(root); // Free allocated memory
    return 0;
}
```