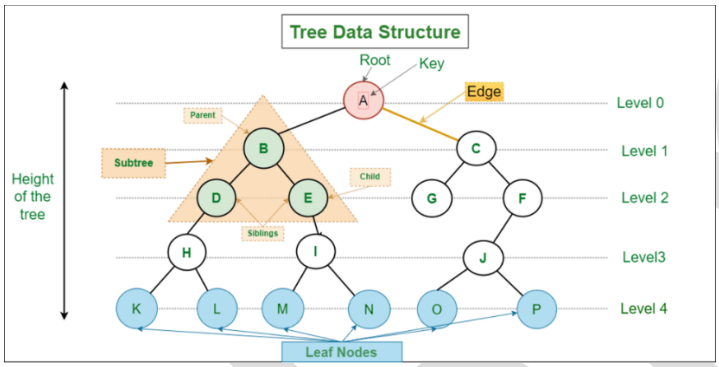
Tree

- 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.



Basic Terminology

1. **Nodes:** Each node in a tree contains some data and may also link to other nodes, known as its children. The top node is called the root node.
2. **Edges:** The connections between nodes are called edges. These edges represent the relationships between nodes.
3. **Root:** The root node is the topmost node in the tree structure. It has no parent and serves as the starting point for accessing other nodes in the tree.
4. **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.
5. **Sibling:** Children of the same parent node are called siblings
6. **Child Nodes:** Nodes directly connected to another node (the parent) are referred to as its children. A node may have zero, one, or multiple children.
7. **Internal node:** A node with at least one child is called Internal Node.
8. **Leaf Nodes:** Nodes that have no children are called leaf nodes. They are the endpoints of the tree structure.
9. **Subtrees:** A subtree is a smaller tree within a larger tree. Any node in a tree, along with its descendants (children, grandchildren, and so on), forms a subtree.

10. **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
11. **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.
12. **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.
13. **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.
14. **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.

Application of Tree Data Structure:

File System: This allows for efficient navigation and organization of files.

Data Compression: Huffman coding 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.
- To learn more about the advantages of Tree Data Structure, refer to this article.

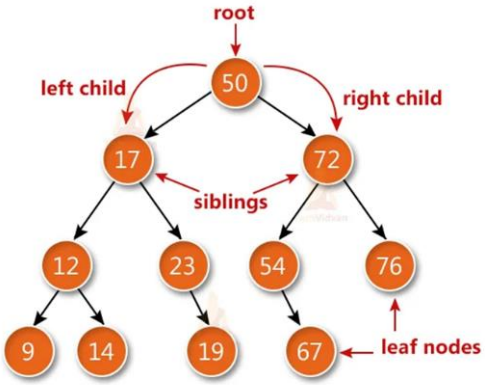
Disadvantages of Tree Data Structure:

- 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 and require a good understanding of the algorithms.

Binary Tree

- A binary tree is a special tree where each node can have not more than two children. A non-empty binary tree consists of the following:
- A node called the root node
- A left sub-tree
- A right sub-tree

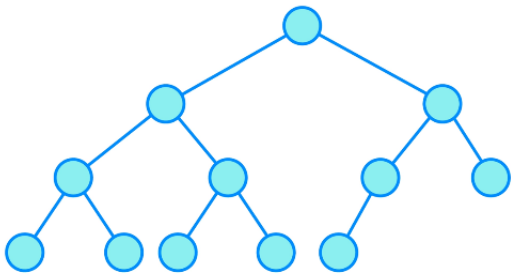


Types of Binary Tree

Complete Binary Tree

- A complete binary tree is a specific type of binary tree where every level of the tree is completely filled, except possibly for the last level, which is filled from left to right.
- Key characteristics of a complete binary tree:

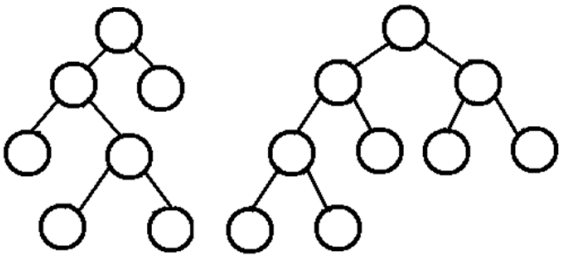
1. **Filled Levels:** All levels of the tree, except possibly the last level, are completely filled with nodes. This means that all nodes are as far left as possible.
2. **Last Level:** If the last level is not completely filled, it is filled from left to right. In other words, the nodes are added from left to right, leaving no gaps between nodes towards the left side.



Full Binary Tree

- A full binary tree, also known as a proper binary tree or 2-tree, is a specific type of tree where every node other than the leaf nodes has exactly two children.
- Key characteristics of a full binary tree:

1. **Nodes and Children:** Each internal node (non-leaf node) in a full binary tree has exactly two children, a left child and a right child.
2. **Leaf Nodes:** Leaf nodes in a full binary tree are the endpoints and have zero children.

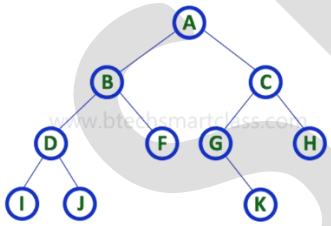


full tree complete tree

Binary Tree Representations

A binary tree data structure is represented using two methods.

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



1. Array Representation of Binary Tree

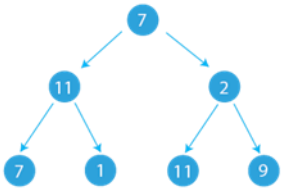
In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
Consider the above example of a binary tree and it is represented as follows...

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

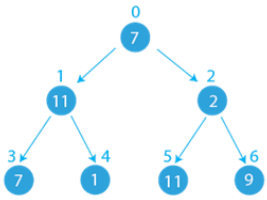
To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

For the array representation of binary tree, we will form an array of size 2*n+1 size where n is the number of nodes the binary tree. Now we will move step by step for the array representation of binary tree.

1. First, suppose we have a binary tree with seven nodes



2.Now, for the array representation of binary tree, we have to give numbering to the corresponding nodes. The standard form of numbering the nodes is to start from the root node and move from left to right at every level .After numbering the tree and nodes will look like this

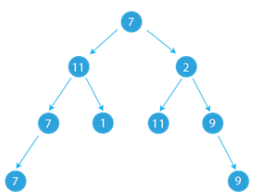


3. Now as we have numbered from zero you can simply place the corresponding number in the matching index of an array then the array will look like this:

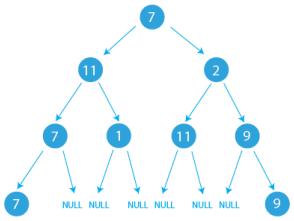
4. That is the array representation of binary tree but this is the easy case as every node has two child so what about other cases? We will discuss other cases below.

5. In these cases, we will discuss the scenarios for the array representation of binary tree where there the lead node is not always on the last level.

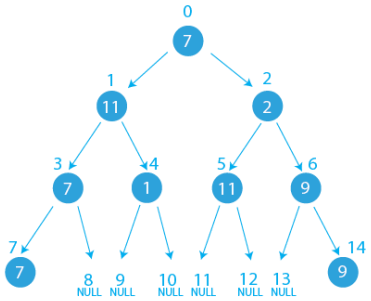
6. So consider the binary tree given below:



7. While giving a number you are stuck with the cases where you encounter a leaf node so just make the child node of the leaf nodes as NULL then the tree will look like this:



8. Now just number the nodes as done above for the array representation of binary tree after that the tree will look like this:



9. Now we have the number on each node we can easily use the tree for array representation of binary tree and the array will look like this:

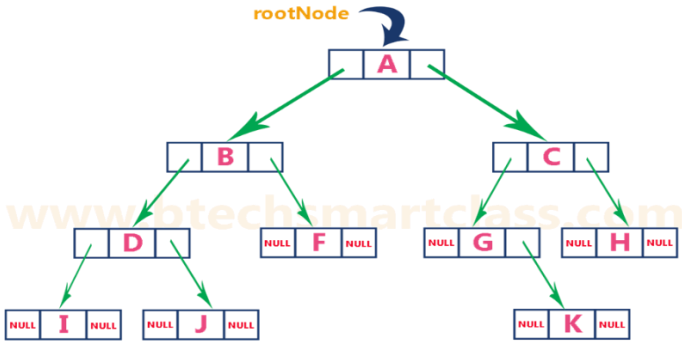
If the node is in the ith index then

- Left child would be at (2*i)+1
- Right child would be at (2*i)+2
- Parent node would be at floor[(i-1)/2]

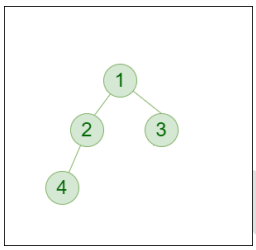
2. Linked List Representation of Binary Tree

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

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



Implement a binary tree using C




```c
#include <stdio.h>
#include <stdlib.h>
 struct node {
    int data;
    struct node* left;
    struct node* right;
};
```

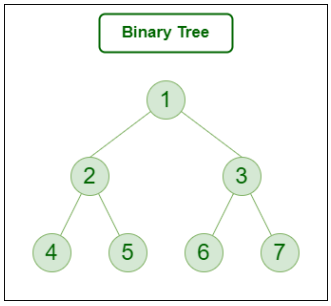
```c
// newNode() allocates a new node
// with the given data and NULL left
// and right pointers.
struct node* newNode(int data)
{
    // Allocate memory for new node
    struct node* node
        = (struct node*)malloc(sizeof(struct node));

    // Assign data to this node
    node->data = data;
```

```
    // Initialize left and
    // right children as NULL
    node->left = NULL;
    node->right = NULL;
    return (node);
}
 int main()
{
    // Create root
    struct node* root = newNode(1);

    /* following is the tree after above statement
        1
       /  \
     NULL NULL
    */
    root->left = newNode(2);
    root->right = newNode(3);

    /* 2 and 3 become left and right children of 1
            1
           / \
          2   3
         / \ / \
    NULL NULL NULL NULL
    */
    root->left->left = newNode(4);

    /* 4 becomes left child of 2
         1
        / \
       2   3
      /  \ / \
     4 NULL NULL NULL
     /   \
    NULL NULL
    */
    getchar();
    return 0;
}
```

Representation of Binary Tree:

Each node in the tree contains the following:

- Data
- Pointer to the left child
- Pointer to the right child




```
// Structure of each node of the tree
 struct node {
    int data;
    struct node* left;
    struct node* right;
};
```

Basic operations on a binary tree

Insertion: To insert a node into a binary tree, you typically start from the root and compare the value to be inserted with the current node. If it's smaller, move to the left child; if it's larger, move to the right child. Repeat this process until you find an empty spot to insert the new node.

Insertion of node into a binary tree

```
#include <stdio.h>
#include <stdlib.h>
// Structure for a node in the binary tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;  }
```

```
// Function to insert a new node in the binary tree
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return newNode(data);
    } else {
        if (data <= root->data) {
            root->left = insert(root->left, data);
        } else {
            root->right = insert(root->right, data);
        }
    }
    return root;  }
// Function to print the binary tree using inorder traversal
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
// Example usage
int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
```

```
        printf("Inorder traversal of the binary tree: ");
        inorderTraversal(root);
        return 0;   }
```

Deletion: Deleting a node involves locating the node to be deleted and then handling its removal while maintaining the tree's structure. The deletion process depends on whether the node to be deleted is a leaf node, has one child, or has two children

Deletion of node into a binary tree

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node *left;
   struct Node *right;  };
struct Node* newNode(int data) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->data = data;
   node->left = NULL;
   node->right = NULL;
   return node;  }
struct Node* insert(struct Node* root, int data) {
   if (root == NULL) {
      return newNode(data);
   } else {
      if (data <= root->data) {
         root->left = insert(root->left, data);
      } else {
         root->right = insert(root->right, data);
      }
 }
   return root;
}
```

```
// Helper function to find the minimum value node in a tree
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}
// Function to delete a node from the binary tree
struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        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;
        }
        struct Node* temp = minValueNode(root->right);
        root->data = temp->data;
```

```
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    printf("Inorder traversal before deletion: ");
    inorderTraversal(root);
    printf("\n");
    root = deleteNode(root, 20); // Delete node with value 20
    printf("Inorder traversal after deletion: ");
    inorderTraversal(root);
    printf("\n");
    return 0;
}
```

This C code includes a deleteNode function that handles the deletion of a node from the binary tree based on the provided data value. It considers various cases such as when the node to be deleted is a leaf node, has one child, or has two children.

Compile and run this code to see how the binary tree changes after the deletion of a node with a specific value (20 in this case). The inorder traversal before and after deletion will demonstrate the effect of the deletion operation on the tree structure.

Binary Tree Traversals:
Tree Traversal algorithms can be classified broadly into two categories:

- Depth-First Search (DFS) Algorithms
- Breadth-First Search (BFS) Algorithms

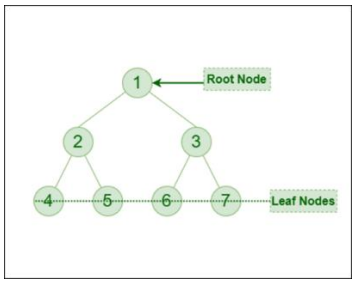
Tree Traversal using Depth-First Search (DFS) algorithm can be further classified into three categories:

- **Preorder Traversal (current-left-right):** Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.
- **Inorder Traversal (left-current-right):** Visit the current node after visiting all nodes inside the left subtree but before visiting any node within the right subtree. Here, the traversal is left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.
- **Postorder Traversal (left-right-current):** Visit the current node after visiting all the nodes of the left and right subtrees. Here, the traversal is left child – right child – root. It means that the left child has traversed first then the right child and finally its root node.

Tree Traversal using Breadth-First Search (BFS) algorithm can be further classified into one category:

- **Level Order Traversal:** Visit nodes level-by-level and left-to-right fashion at the same level. Here, the traversal is level-wise. It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

Let us traverse the following tree with all four traversal methods:



Pre-order Traversal of the above tree: 1-2-4-5-3-6-7
In-order Traversal of the above tree: 4-2-5-1-6-3-7

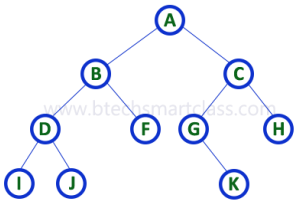
Post-order Traversal of the above tree: 4-5-2-6-7-3-1
Level-order Traversal of the above tree: 1-2-3-4-5-6-7

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



. In - Order Traversal (leftChild - root - rightChild)

- In-order traversal is also called as symmetric traversal
- In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree.
- The word 'in' in the in-order specifies that the root node is accessed in between the left and the right sub-trees.
- In-order algorithm is also known as LNR traversal algorithm(Left-Node-Right)

Algorithm(Recursive)

- **Step1: Repeat steps 2 to 4 while TREE!=NULL**
- **Step2: INORDER(TREE->LEFT)**
- **Step3: Write TREE->DATA**
- **Step4: INORDER(TREE->RIGHT)**
- **[END OF LOOP]**
- **Step5: END**

In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**. With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A. Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit **'G'** and then visit G's right child K. With this we have completed the left part of node C. Then visit root node **'C'** and next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process. That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is
I - D - J - B - F - A - G - K - C – H

Pre - Order Traversal (root - leftChild - rightChild)

- Pre-order traversal is also called as depth-first traversal
- In this algorithm, the left sub-tree is always traversed before the right sub-tree
- The word ‘pre’ in the pre-order specified that the root node is accessed first.
- Pre-order algorithm is also known as the NLR traversal algorithm(Node-Left-Right)
- Pre-order traversal algorithms are used to extract a prefix expression tree.

Algorithm(Recursive)

Step1: Repeat steps 2 to 4 while TREE!=NULL

Step2: Write TREE->DATA

Step3: PREORDER(TREE->LEFT)

Step4: PREORDER(TREE->RIGHT)

[END OF LOOP]

Step5: END

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.
In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F. So we visit B's left child **'D'** and again D is a root for I and J. So we visit D's left child **'I'** which is the leftmost child. So next we go for visiting D's right child **'J'**. With this we have completed root, left and right parts of node D and root, left parts of node B. Next

visit B's right child **'F'**. With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this, we have completed node C's root and left parts. Next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K – H

Post - Order Traversal (leftChild - rightChild - root)

- In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node.
- The word ‘post’ specifies that the root node is accessed after the left and the right sub trees.
- Post-order algorithm is also known as LRN traversal algorithm(Left-Right-Node)
- Post-order traversals are used to extract postfix notation from an expression tree

Algorithm(Recursive)

- **Step1: Repeat steps 2 to 4 while TREE!=NULL**
- **Step2: POSTORDER(TREE->LEFT)**
- **Step3: POSTORDER(TREE->RIGHT)**
- **Step4: Write TREE->DATA**
- **[END OF LOOP]**
- **Step5: END**

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C – A

Binary tree traversal algorithm without recursion

- Using Stack is the obvious way to traverse tree without recursion

Pre-Order traversal Algorithm

Step1: Start with root node and Push onto stack.

Step2: Repeat steps 3 to 5 while stack is not empty

Step3: POP the top element (PTR) from the stack and process the node

Step4: Push the right child of PTR onto stack

Step5: Push the left child of PTR onto stack.

Step6: Exit

In-Order traversal Algorithm

Step1: Set PTR = ROOT

Step2: Repeat steps while PTR!=NULL

(a) Push PTR onto stack

(b) Set PTR=PTR->LEFT

Step3: If PTR is NULL and stack is not empty then

(a) POP the top element(PTR) from the stack and process it.

(b) Set PTR=PTR->RIGHT

(c) Goto Step2

[End of If]

Step4: If PTR is NULL and stack is empty then we are done.

Step5: Exit

Post-Order traversal Algorithm

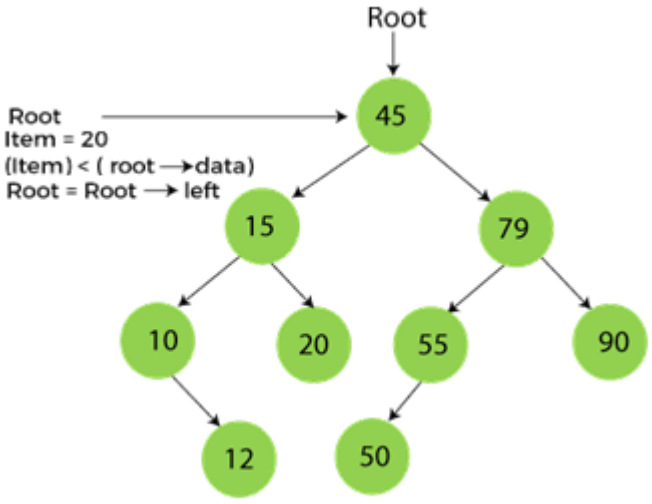
```
Step1:  PTR =ROOT, PREV=NULL
Step2: Repeat Steps while PTR!=NULL or Stack not empty
Step3:  IF(PTR!=NULL)
            Push PTR onto Stack
            PTR=PTR->LEFT
          ELSE
            PTR=TOP[Stack]
            If (PTR->RIGHT==NULL or  PTR->RIGHT==PREV)
                  POP the top element(PTR) from the stack
                   Process PTR
                   SET PREV=PTR
                   PTR=NULL
             ELSE
                    PTR=PTR->RIGHT
             [End of If]
```

[End of If]
[End of Loop]
Step4: EXIT

Binary Search Tree(BST)

- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order.
- In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
    - The left sub-tree of a node N contains values that are less than N's value.
    - The right sub-tree of a node N contains values that are greater than N's value.
    - Both left and right binary trees also satisfy these properties and thus are binary search trees

Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced.
Thus, the average running time of a search operation is $O(\log_2 n)$
BST also speed up the insertion and deletion operations.
The tree has a speed advantage when the data in the structure's changes rapidly.
Binary search trees are efficient data structures especially when compared with sorted linear arrays and linked lists.



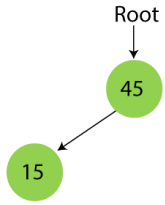
Example of creating a binary search tree
Now, let's see the creation of binary search tree using an example.
Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50
First, we have to insert 45 into the tree as the root of the tree.
Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

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

Step 1 - Insert 45.

Binary Search tree

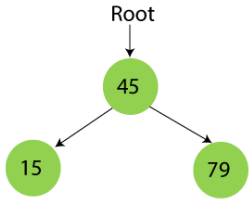
Step 2 - Insert 15.



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

Binary Search tree

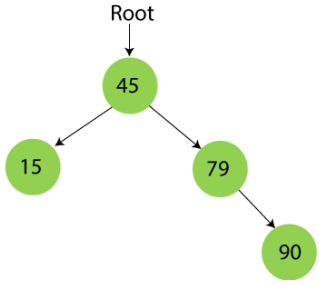
Step 3 - Insert 79.



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

Binary Search tree

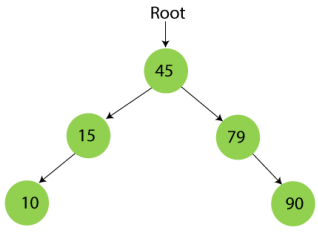
Step 4 - Insert 90.



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

Binary Search tree

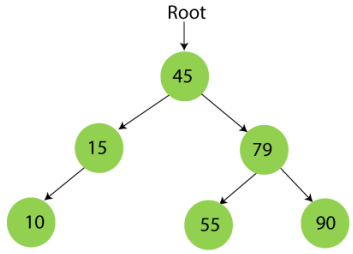
Step 5 - Insert 10.



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

Binary Search tree

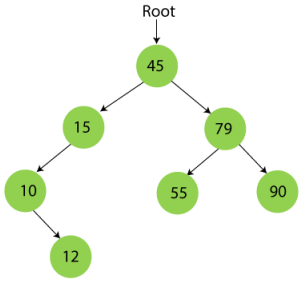
Step 6 - Insert 55.



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

Binary Search tree

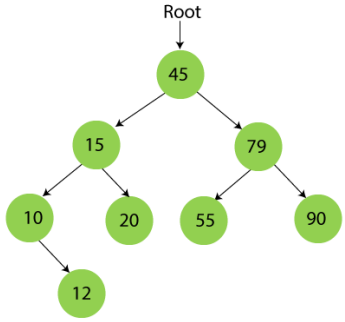
Step 7 - Insert 12.



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

Binary Search tree

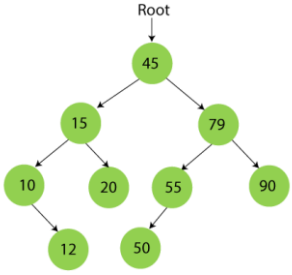
Step 8 - Insert 20.



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

Binary Search tree

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.

Binary Search tree

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

We can perform insert, delete and search operations on the binary search tree.
Let's understand how a search is performed on a binary search tree.

Operations of Binary Search Trees

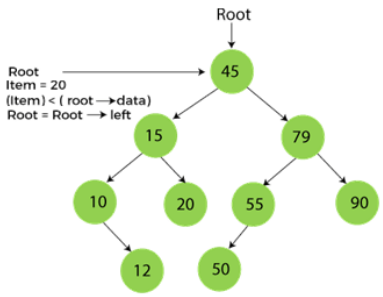
Searching Algorithm

```
SearchElement(TREE,VAL)
Step1:  IF TREE->DATA = VAL OR TREE=NULL
            Return TREE
        ELSE
        IF VAL< TREE->DATA
                Return SearchElement(TREE->LEFT,VAL)
        ELSE
                Return SearchElement(TREE->RIGHT,VAL)
        [END OF IF]
    [END OF IF]
Step2: END
```

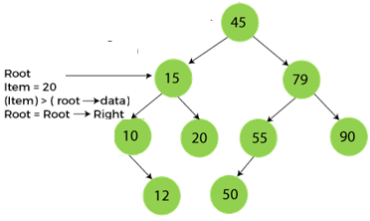
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the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

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

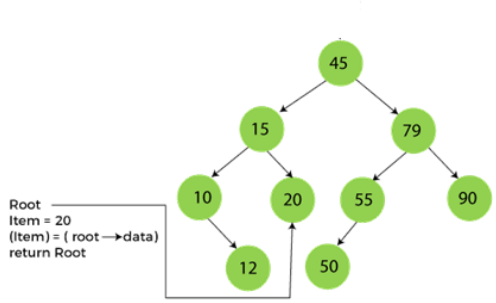
Step1:



Step2:



Step3:



C Program to Seach a node in a BST

```
#include <stdio.h>

#include <stdlib.h>

// Structure for a BST node

struct Node {

    int data;

    struct Node* left;

    struct Node* right;  };

// Function to create a new node

struct Node* createNode(int value) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;  }

// Function to insert a node into the BST

struct Node* insert(struct Node* root, int value) {

    if (root == NULL) {

        return createNode(value);    }
```

```
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);     }
        return root;
}
// Function to search for a value in the BST
struct Node* search(struct Node* root, int value) {
    if (root == NULL || root->data == value) {
        return root;          }
        if (value < root->data) {
        return search(root->left, value);
    } else {
        return search(root->right, value);
    }      }
int main() {
    // Input BST: [8,3,10,1,6,null,14,null,null,4,7,13,null]
    struct Node* root = NULL;
    root = insert(root, 8);
    root = insert(root, 3);
    root = insert(root, 10);
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 4);
    root = insert(root, 7);
```

```
    root = insert(root, 14);

    root = insert(root, 13);

      int key = 6;

    struct Node* result = search(root, key);

       if (result != NULL) {

       printf("Key %d found in the BST.\n", key);

    } else {

       printf("Key %d not found in the BST.\n", key);

    }

      return 0;

}
```

Inserting a New Node in a Binary Search Tree

- The insert function is used to add a new node with a given value at the correct position in the binary search tree.
- Adding the node at the correct position means that the node should not violate the properties of the binary search tree.

Algorithm

```
  Insert(TREE,VAL)
Step1: IF TREE=NULL
                Allocate memory for TREE
                SET TREE->DATA = VAL
                SET TREE->LEFT =TREE->RIGHT=NULL
        ELSE
                IF VAL<TREE->DATA
                        Insert(TREE->LEFT,VAL)
                ELSE
                        Insert(TREE->RIGHT,VAL)
                [END OF IF]
        [END OF IF]
Step2: END
```

```
#include <stdio.h>
#include <stdlib.h>
// Structure for a node in BST
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
```

```
// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a new node into BST
struct Node* insert(struct Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);

    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}

// Function to print the inorder traversal of BST
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
int main()
{
    struct Node* root = NULL;
    int values[] = { 50, 30, 70, 20, 40, 60, 80 };
    // Insert values into BST
    for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++) {
        root = insert(root, values[i]);
    }
    printf("Inorder traversal of BST after insertion: ");
    inorderTraversal(root);
    printf("\n");
    int newValue = 55;
    // Insert a new value into BST
    root = insert(root, newValue);
    printf("Inorder traversal of BST after inserting %d: ", newValue);
    inorderTraversal(root);
    printf("\n");
    return 0;
}
```

Deleting a New Node in a Binary Search Tree

- The delete function deletes a node from the binary search tree.
- But properties of the binary search tree are not to be violated when deleting nodes.
- There are 3 cases
- **Case 1: Deleting a node that has no children:** We can simply remove this node without any issue. This is the simplest case of deletion.
- **Case 2: Deleting a Node with one Child:** The node's child is set as the child of the nod's parent.
- **Case 3: Deleting a node with two children:** Replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree). Then delete the in-order predecessor or the successor

Algorithm

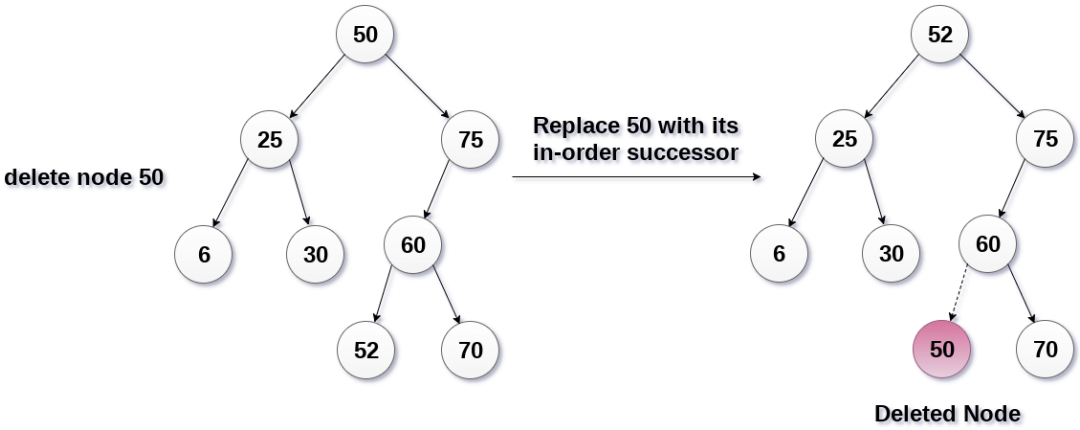
Delete (TREE, ITEM)

```
Step 1: IF TREE = NULL
   Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
   Delete(TREE -> RIGHT, ITEM)
  ELSE IF TREE -> LEFT AND TREE -> RIGHT
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA
   Delete(TREE -> LEFT, TEMP -> DATA)
  ELSE
   SET TEMP = TREE
   IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
   SET TREE = NULL
  ELSE IF TREE -> LEFT != NULL
  SET TREE = TREE -> LEFT
  ELSE
    SET TREE = TREE -> RIGHT
  [END OF IF]
  FREE TEMP
[END OF IF]
Step 2: END
```

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

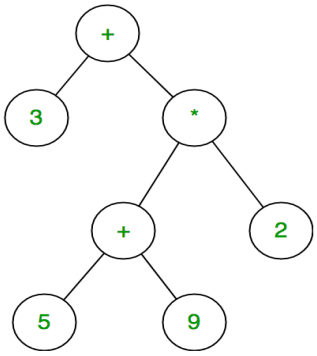
replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



EXPRESSION TREE

- An expression tree is a special type of binary tree that is used to store algebraic expressions.
- In an expression tree, each internal node corresponds to the operator and each leaf node corresponds to the operand.
- The in-order traversal of the tree returns the infix expression.
- Similarly, the pre-order and post-order traversal of the expression tree will return prefix and postfix expression respectively.

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 expression and do the following for every character.

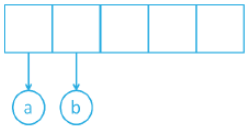
1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

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

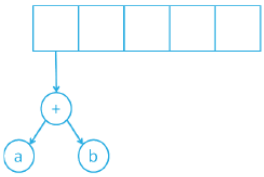
Examples:

```
a b + c *
```

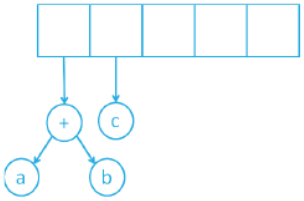
Step1 : The first two symbols are operands, for which we generate a one-node tree and push a reference to the stack.



Step2 : Next, read a'+' symbol to pop two pointers to the tree, form a new tree, and push a pointer to it onto the stack.



Step3 : In the next stage 'c' is read, we build a single node tree and store a pointer to it on the stack.



Step4 : Finally, we read the last symbol' *', pop two tree pointers, and build a new tree with a,"*as root, and a pointer to the final tree remains on the stack.