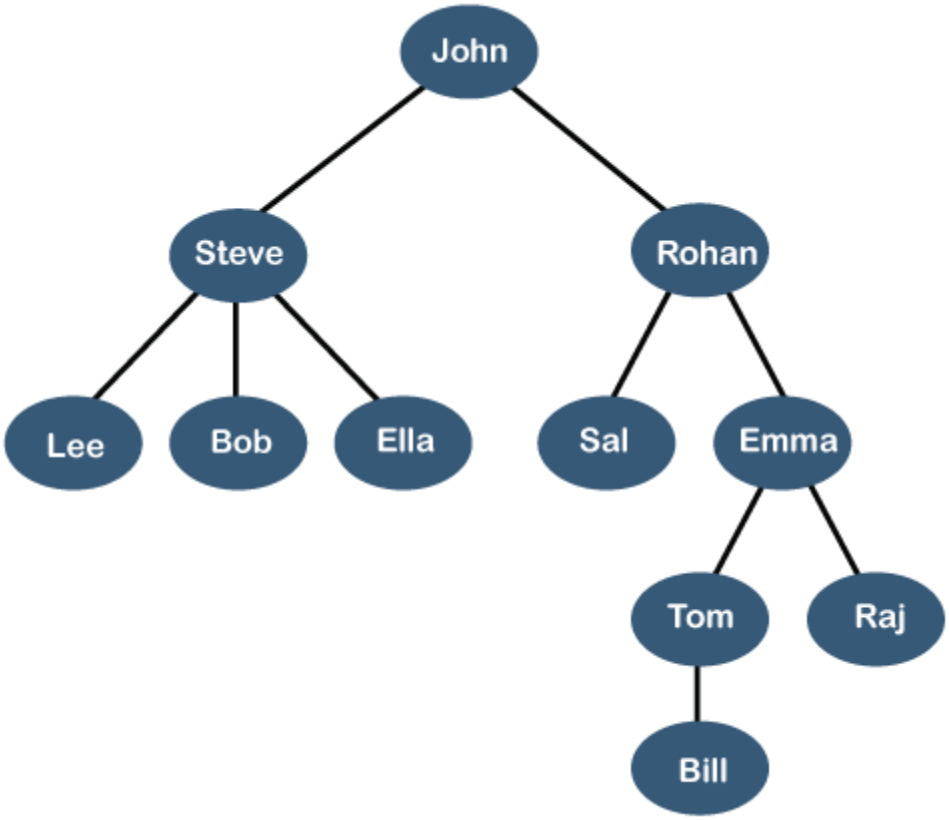
Tree

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



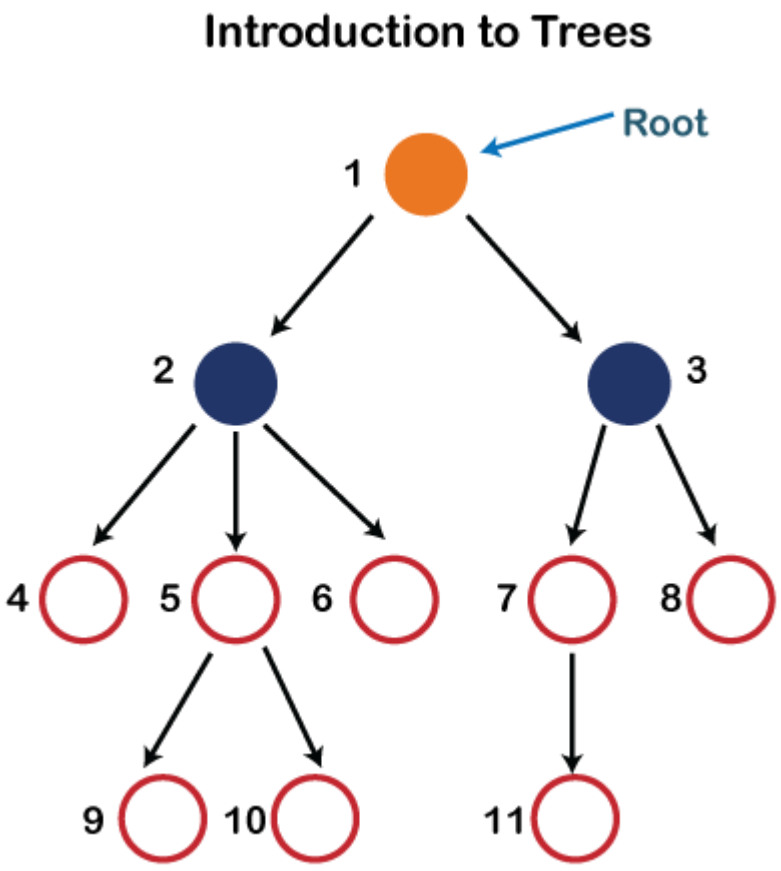
The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***. Tom has one direct report named ***Bill***. This particular logical structure is known as a ***Tree***. Its structure is similar to the real tree, so it is named a ***Tree***. In this structure, the ***root*** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

Let's understand some key points of the Tree data structure.

- 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.

Let's consider the tree structure, which is shown below:



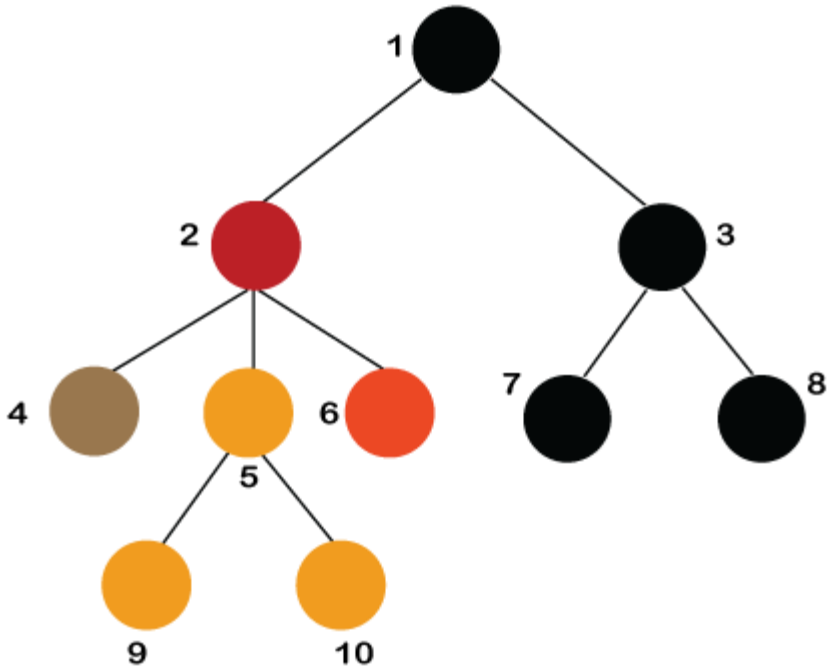
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast 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.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree

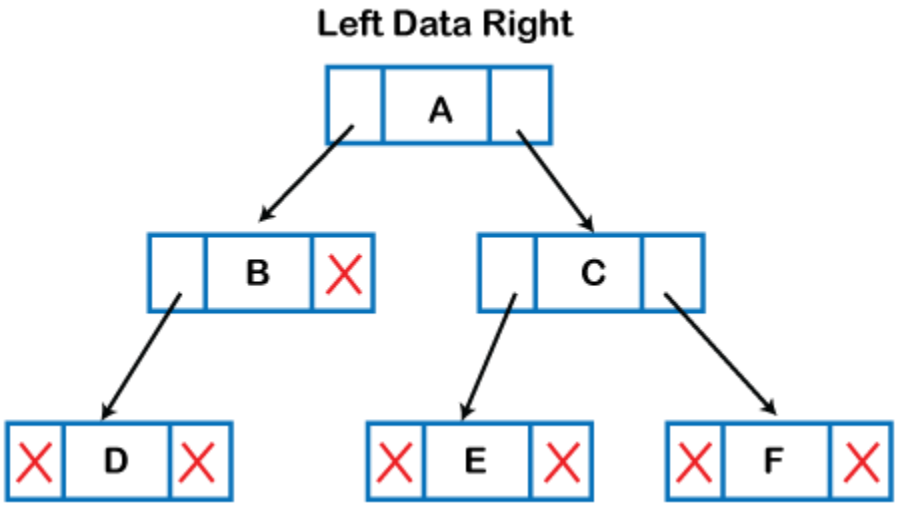
data structure is implemented in various applications.



- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

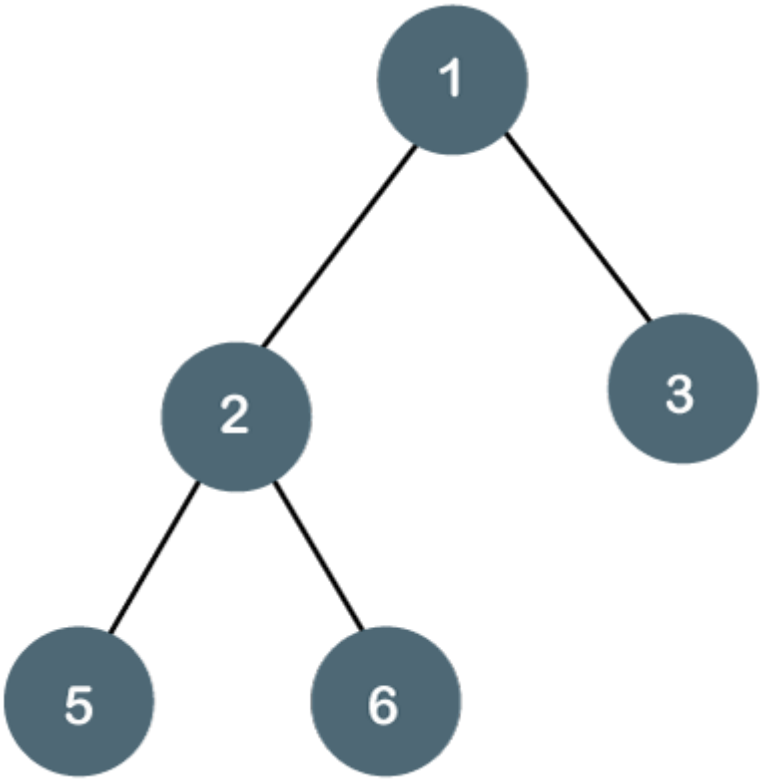
In programming, the structure of a node can be defined as:

```
1. struct node
2. {
3.   int data;
4. struct node *left;
5. struct node *right;
6. }
```

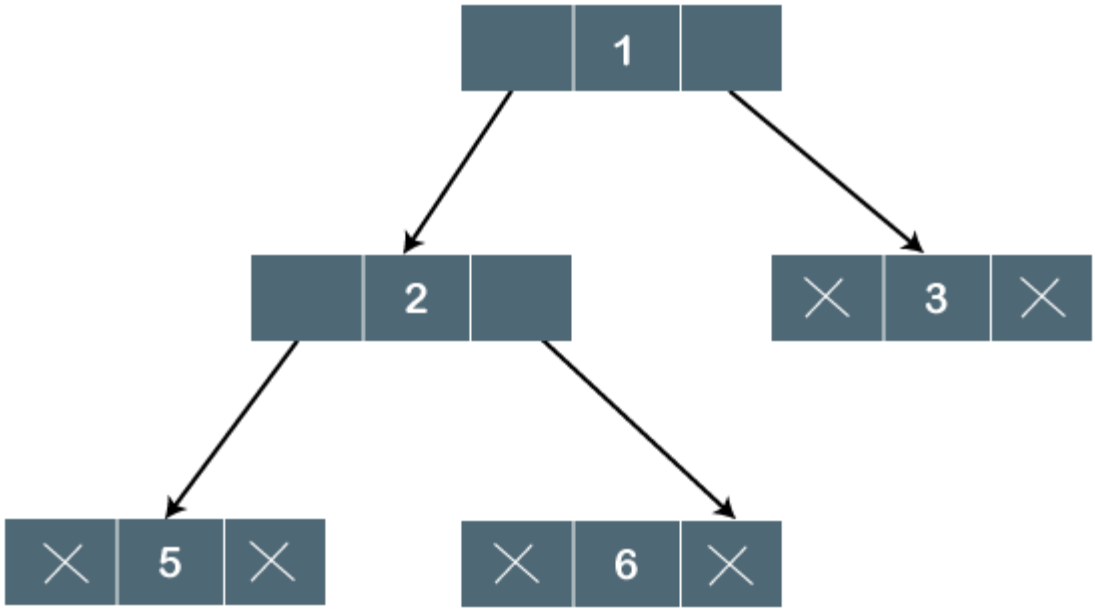
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + ...2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

$n = h+1$

h= n-1

Array and linked Representation of Binary Trees

1. Array Representation of Binary Trees in C

In the array representation, we store the elements of the tree in an array. The position of each element is based on the relationship between parent and child nodes.

Example:

Consider the binary tree:

```
    1
   / \
  2   3
 / \   \
4   5   6
```

This binary tree can be represented in an array as follows:

```
#include <stdio.h>


int main() {
    int tree[7]; // Assuming a tree with up to 7 nodes (1-based indexing)


    // Assigning values to the array (1-based index)
    tree[1] = 1; // Root
    tree[2] = 2; // Left child of root
    tree[3] = 3; // Right child of root
    tree[4] = 4; // Left child of node 2
    tree[5] = 5; // Right child of node 2
    tree[6] = 6; // Right child of node 3
```

```
    // Accessing the nodes
    printf("Root: %d\n", tree[1]);
    printf("Left child of root: %d\n", tree[2]);
    printf("Right child of root: %d\n", tree[3]);
    printf("Left child of node 2: %d\n", tree[4]);
    printf("Right child of node 2: %d\n", tree[5]);
    printf("Right child of node 3: %d\n", tree[6]);

    return 0;
}
```

In this code:

- The array `tree` holds the binary tree's nodes.
- The root is at index `1`.
- The left and right children of each node are accessed using the formulas `2i` and `2i + 1`, respectively.

This binary tree can be represented in an array as follows:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100

int tree[MAX_SIZE];

void insert(int data, int index) {
    if (index < MAX_SIZE) {
        tree[index] = data;
    } else {
```

```
        printf("Array out of bounds\n");
    }
}

int getLeftChild(int index) {
    int leftIndex = 2 * index + 1;
    if (leftIndex < MAX_SIZE && tree[leftIndex] != 0) {
        return leftIndex;
    } else {
        return -1; // No left child
    }
}

int getRightChild(int index) {
    int rightIndex = 2 * index + 2;
    if (rightIndex < MAX_SIZE && tree[rightIndex] != 0) {
        return rightIndex;
    } else {
        return -1; // No right child
    }
}

int getParent(int index) {
    if (index > 0) {
        return (index - 1) / 2;
    } else {
        return -1; // Root node has no parent
```

```
    }
}

void printTree() {
    printf("Array representation of the binary tree:\n");
    for (int i = 0; i < MAX_SIZE; i++) {
        if (tree[i] != 0) {
            printf("%d ", tree[i]);
        } else {
            printf("- ");
        }
    }
    printf("\n");
}

int main() {
    // Initialize the array with 0s
    for (int i = 0; i < MAX_SIZE; i++) {
        tree[i] = 0;
    }

    // Insert some values into the tree
    insert(1, 0); // Root
    insert(2, 1); // Left child of root
    insert(3, 2); // Right child of root
    insert(4, 3); // Left child of 2
    insert(5, 6); // Right child of 3
```

```
    printTree();


    return 0;
}
```

Explanation:

- Array: The binary tree is stored in an array.
- Indexing: The root node is stored at index 0.
- Children: The left child of a node at index i is stored at index 2*i + 1, and the right child is stored at index 2*i + 2.
- Parent: The parent of a node at index i (except for the root) is stored at index (i - 1) / 2.
- Functions: The code provides functions for inserting elements, getting left and right children, getting the parent, and printing the tree.

2. Linked Representation of Binary Trees in C

In the linked representation, each node is a structure with data and pointers to its left and right children.

Example:

```
#include <stdio.h>

#include <stdlib.h>


// Definition of a tree node

struct Node {

    int data;

    struct Node* left;

    struct Node* right;

};
```

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

// Function to traverse the tree in pre-order
void preOrderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    printf("%d ", node->data); // Visit the root
    preOrderTraversal(node->left); // Traverse the left subtree
    preOrderTraversal(node->right); // Traverse the right subtree
}

int main() {
    // Creating the binary tree manually
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->right = createNode(6);
```

```
    // Traversing the tree in pre-order
    printf("Pre-order Traversal: ");
    preOrderTraversal(root);
    printf("\n");

    return 0;
}
```

In this code:

- A `Node` structure is defined with an integer `data` and two pointers: `left` and `right`.
- The `createNode` function allocates memory for a new node and initializes it.
- The `preOrderTraversal` function demonstrates tree traversal, visiting nodes in the order: root, left subtree, and then right subtree.

Summary:

- **Array Representation** is simple and provides constant-time access to any node but can be inefficient in terms of space for sparse trees.
- **Linked Representation** is more flexible and space-efficient, allowing for easy modifications to the tree structure, but access time is proportional to the height of the tree.

Binary Tree Traversal in Data Structure

The tree can be defined as a non-linear data structure that stores data in the form of nodes, and nodes are connected to each other with the help of edges. Among all the nodes, there is one main node called the ***root node***, and all other nodes are the children of these nodes.

In any data structure, traversal is an important operation. In the traversal operation, we walk through the data structure visiting each element of the data structure at least once.

Types of Traversal of Binary Tree

There are three types of traversal of a binary tree.

1. Inorder tree traversal

2. Preorder tree traversal
3. Postorder tree traversal

Inorder Tree Traversal

The left subtree is visited first, followed by the root, and finally the right subtree in this traversal strategy. Always keep in mind that any node might be a subtree in and of itself. The output of a binary tree traversal in order produces sorted key values in ascending order.

C Code

Let's write a basic C program for Inorder traversal of the binary search tree.

```
1. //C Program for Inorder traversal of the binary search tree
2.
3. #include<stdio.h>
4. #include<stdlib.h>
5.
6. struct node
7. {
8.     int key;
9.     struct node *left;
10.    struct node *right;
11. };
12.
13. //return a new node with the given value
14. struct node *getNode(int val)
15. {
16.    struct node *newNode;
17.
18.    newNode = malloc(sizeof(struct node));
19.
20.    newNode->key   = val;
21.    newNode->left  = NULL;
22.    newNode->right = NULL;
```

```
23.
24.     return newNode;
25. }
26.
27. //inserts nodes in the binary search tree
28. struct node *insertNode(struct node *root, int val)
29. {
30.      if(root == NULL)
31.          return getNode(val);
32.
33.      if(root->key < val)
34.          root->right = insertNode(root->right,val);
35.
36.      if(root->key > val)
37.          root->left = insertNode(root->left,val);
38.
39.      return root;
40. }
41.
42. //inorder traversal of the binary search tree
43. void inorder(struct node *root)
44. {
45.     if(root == NULL)
46.         return;
47.
48.     //traverse the left subtree
49.     inorder(root->left);
50.
51.     //visit the root
52.     printf("%d ",root->key);
53.
54.     //traverse the right subtree
55.     inorder(root->right);
56. }
```

```
57.
58. int main()
59. {
60.    struct node *root = NULL;
61.
62.
63.     int data;
64.     char ch;
65.        /*  Do while loop to display various options to select from to decide the input  */
66.        do
67.        {
68.            printf("\nSelect one of the operations::");
69.            printf("\n1. To insert a new node in the Binary Tree");
70.            printf("\n2. To display the nodes of the Binary Tree(via Inorder Traversal).\n");
71.
72.            int choice;
73.            scanf("%d",&choice);
74.            switch (choice)
75.            {
76.            case 1 :
77.                printf("\nEnter the value to be inserted\n");
78.                scanf("%d",&data);
79.                root = insertNode(root,data);
80.                break;
81.            case 2 :
82.                printf("\nInorder Traversal of the Binary Tree::\n");
83.                inorder(root);
84.                break;
85.            default :
86.                printf("Wrong Entry\n");
87.                break;
88.            }
89.
90.            printf("\nDo you want to continue (Type y or n)\n");
```

```
91.             scanf(" %c",&ch);
92.         } while (ch == 'Y'|| ch == 'y');
93.
94.    return 0;
95. }
```

Output

The above C code hives the following output.

```
Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree (via Inorder Traversal).
1

Enter the value to be inserted
12

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Inorder Traversal).
1

Enter the value to be inserted
98

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Inorder Traversal).
1

Enter the value to be inserted
23

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
```

```
2. To display the nodes of the Binary Tree(via Inorder Traversal).
1

Enter the value to be inserted
78

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Inorder Traversal).
1

Enter the value to be inserted
45

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Inorder Traversal).
1

Enter the value to be inserted
87

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Inorder Traversal).
2

Inorder Traversal of the Binary Tree::
12 23 45 78 87 98
Do you want to continue (Type y or n)
n
```

Preorder Tree Traversal

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

Code

Let's write a C code for **the** Preorder traversal of the binary search tree.

```
/*
 * Program: Preorder traversal of the binary search tree
 * Language: C
 */

#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left;
    struct node *right;
};

//return a new node with the given value
struct node *getNode(int val)
{
    struct node *newNode;

    newNode = malloc(sizeof(struct node));

    newNode->key   = val;
    newNode->left  = NULL;
    newNode->right = NULL;

    return newNode;
}

//inserts nodes in the binary search tree
struct node *insertNode(struct node *root, int val)
{
```

```
33.     if(root == NULL)
34.         return getNode(val);
35.
36.     if(root->key < val)
37.         root->right = insertNode(root->right,val);
38.
39.     if(root->key > val)
40.         root->left = insertNode(root->left,val);
41.
42.     return root;
43. }
44.
45. //preorder traversal of the binary search tree
46. void preorder(struct node *root)
47. {
48.     if(root == NULL)
49.         return;
50.
51.     //visit the root
52.     printf("%d ",root->key);
53.
54.     //traverse the left subtree
55.     preorder(root->left);
56.
57.     //traverse the right subtree
58.     preorder(root->right);
59. }
60.
61. int main()
62. {
63.    struct node *root = NULL;
64.
65.    int data;
66.     char ch;
```

```
67.       /*  Do while loop to display various options to select from to decide the input  */
68.       do
69.       {
70.          printf("\nSelect one of the operations::");
71.          printf("\n1. To insert a new node in the Binary Tree");
72.          printf("\n2. To display the nodes of the Binary Tree(via Preorder Traversal).\n");
73.
74.          int choice;
75.          scanf("%d",&choice);
76.          switch (choice)
77.          {
78.          case 1 :
79.             printf("\nEnter the value to be inserted\n");
80.             scanf("%d",&data);
81.             root = insertNode(root,data);
82.             break;
83.          case 2 :
84.             printf("\nPreorder Traversal of the Binary Tree::\n");
85.             preorder(root);
86.             break;
87.          default :
88.             printf("Wrong Entry\n");
89.             break;
90.          }
91.
92.          printf("\nDo you want to continue (Type y or n)\n");
93.          scanf(" %c",&ch);
94.       } while (ch == 'Y'|| ch == 'y');
95.
96.    return 0;
97. }
```

Output:

```
Select one of the operations::
```

```
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
45

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
53

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
1

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
2

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1
```

```
Enter the value to be inserted
97

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
22

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
2

Preorder Traversal of the Binary Tree::
45 1 2 22 53 97
Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
76

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
30
```

```
Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
67

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
1

Enter the value to be inserted
4

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Preorder Traversal).
2

Preorder Traversal of the Binary Tree::
45 1 2 22 4 30 53 97 76 67
Do you want to continue (Type y or n)
n
```

Postorder Tree Traversal

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

Code

Let's write a program for Postorder traversal of the binary search tree.

1. /*

```
2.   * Program: Postorder traversal of the binary search tree
3.   * Language: C
4.   */
5.
6.  #include<stdio.h>
7.  #include<stdlib.h>
8.
9.  struct node
10. {
11.     int key;
12.     struct node *left;
13.     struct node *right;
14. };
15.
16. //return a new node with the given value
17. struct node *getNode(int val)
18. {
19.     struct node *newNode;
20.
21.     newNode = malloc(sizeof(struct node));
22.
23.     newNode->key   = val;
24.     newNode->left  = NULL;
25.     newNode->right = NULL;
26.
27.     return newNode;
28. }
29. //inserts nodes in the binary search tree
30. struct node *insertNode(struct node *root, int val)
31. {
32.      if(root == NULL)
33.          return getNode(val);
34.
35.      if(root->key < val)
```

```
36.          root->right = insertNode(root->right,val);
37.
38.      if(root->key > val)
39.          root->left = insertNode(root->left,val);
40.
41.      return root;
42. }
43.
44. //postorder traversal of the binary search tree
45. void postorder(struct node *root)
46. {
47.     if(root == NULL)
48.         return;
49.
50.     //traverse the left subtree
51.     postorder(root->left);
52.
53.     //traverse the right subtree
54.     postorder(root->right);
55.
56.     //visit the root
57.     printf("%d ",root->key);
58. }
59. int main()
60. {
61.    struct node *root = NULL;
62.
63.
64.    int data;
65.     char ch;
66.         /*  Do while loop to display various options to select from to decide the input  */
67.         do
68.         {
69.             printf("\nSelect one of the operations::");
```

```
70.         printf("\n1. To insert a new node in the Binary Tree");
71.         printf("\n2. To display the nodes of the Binary Tree(via Postorder Traversal).\n");
72.
73.         int choice;
74.         scanf("%d",&choice);
75.         switch (choice)
76.         {
77.         case 1 :
78.            printf("\nEnter the value to be inserted\n");
79.            scanf("%d",&data);
80.            root = insertNode(root,data);
81.            break;
82.         case 2 :
83.            printf("\nPostorder Traversal of the Binary Tree::\n");
84.            postorder(root);
85.            break;
86.         default :
87.            printf("Wrong Entry\n");
88.            break;
89.         }
90.
91.         printf("\nDo you want to continue (Type y or n)\n");
92.         scanf(" %c",&ch);
93.       } while (ch == 'Y'|| ch == 'y');
94.
95.   return 0;
96. }
```

Output:

```
Select one of the operations::
1. To insert a new node in the Binary Tree
2.  To  display  the  nodes  of  the  Binary  Tree(via  Postorder
Traversal).
1

Enter the value to be inserted
```

```
12

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2.  To  display  the  nodes  of  the  Binary  Tree(via  Postorder
Traversal).
1

Enter the value to be inserted
31

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2.  To  display  the  nodes  of  the  Binary  Tree(via  Postorder
Traversal).
24
Wrong Entry

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2.  To  display  the  nodes  of  the  Binary  Tree(via  Postorder
Traversal).
1

Enter the value to be inserted
24

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2.  To  display  the  nodes  of  the  Binary  Tree(via  Postorder
Traversal).
1

Enter the value to be inserted
88
```

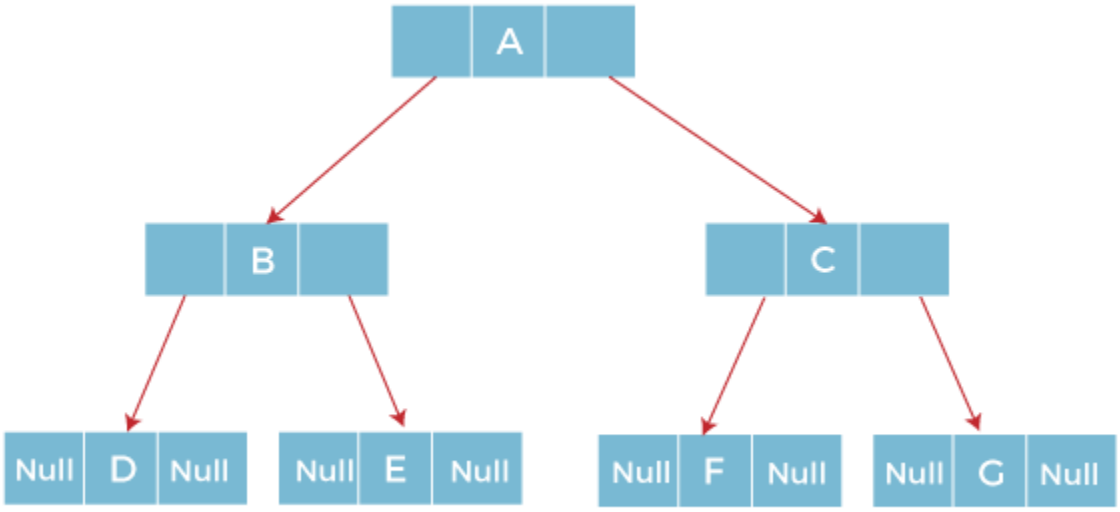
```
Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
1

Enter the value to be inserted
67

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
1

Enter the value to be inserted
56

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
1

Enter the value to be inserted
90

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
1

Enter the value to be inserted
```

```
44

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
1

Enter the value to be inserted
71

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
1

Enter the value to be inserted
38

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
1

Enter the value to be inserted
29

Do you want to continue (Type y or n)
y

Select one of the operations::
1. To insert a new node in the Binary Tree
2. To display the nodes of the Binary Tree(via Postorder
Traversal).
2
```

```
Postorder Traversal of the Binary Tree::
29 24 38 44 56 71 67 90 88 31 12
Do you want to continue (Type y or n)
n
```

What do you mean by Threaded Binary Tree?

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton 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.



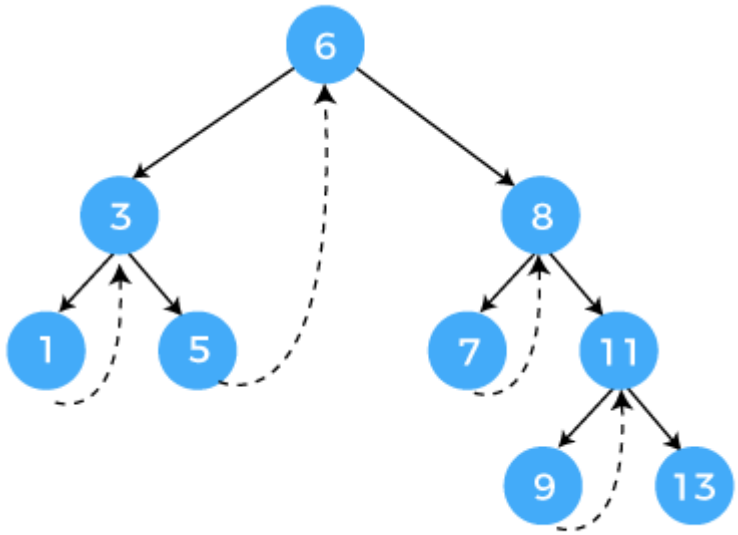
Threaded binary tree

Types of Threaded Binary Tree

There are two types of threaded Binary Tree:

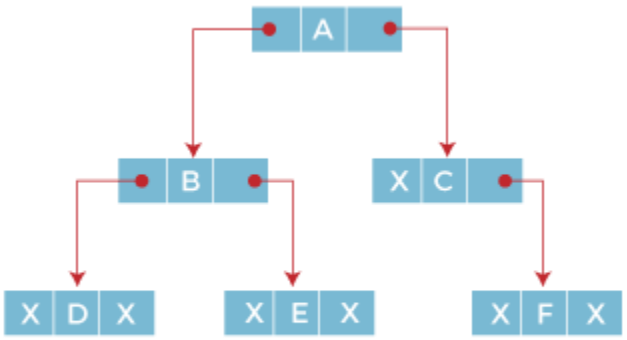
- One-way threaded Binary Tree
- Two-way threaded Binary Tree

One-way threaded Binary trees:

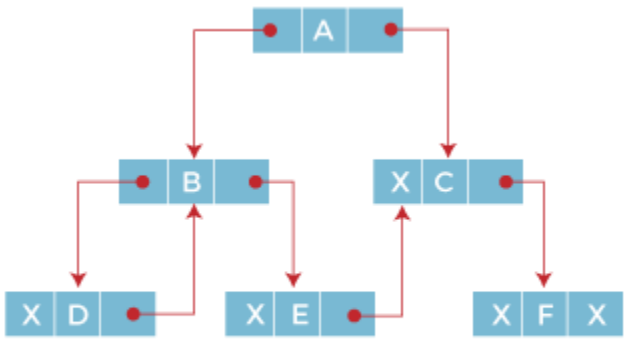


Single Threaded Binary Tree

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.** Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees. In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.



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

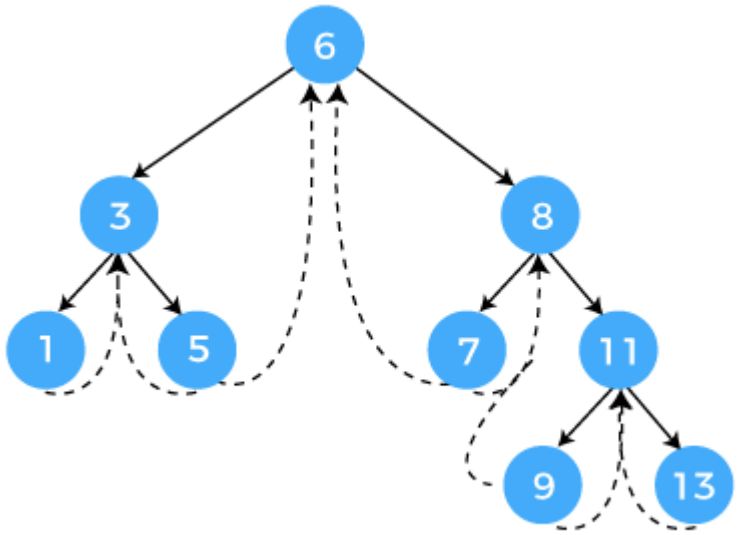


A right - threaded binary tree

The above figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the

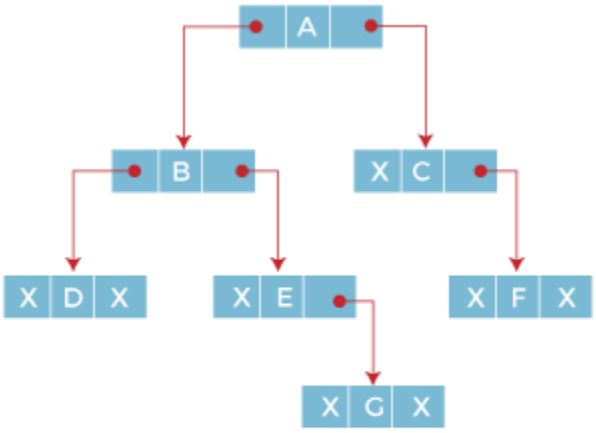
inorder successor of a node D. In the same way other nodes containing values in the right link field will contain NULL value.

Two-way threaded Binary Trees:

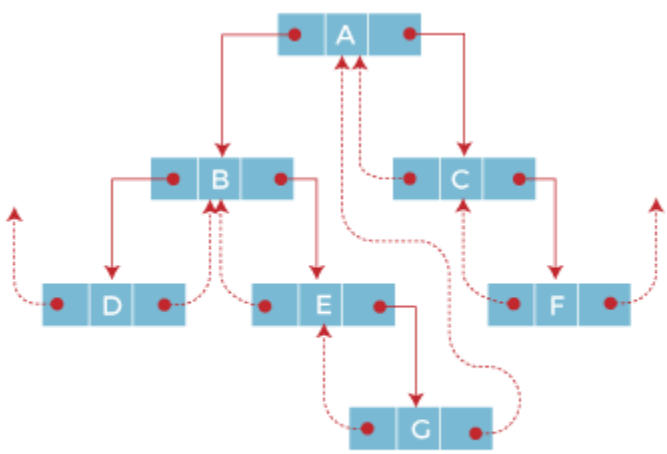


Double Threaded Binary Tree

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.

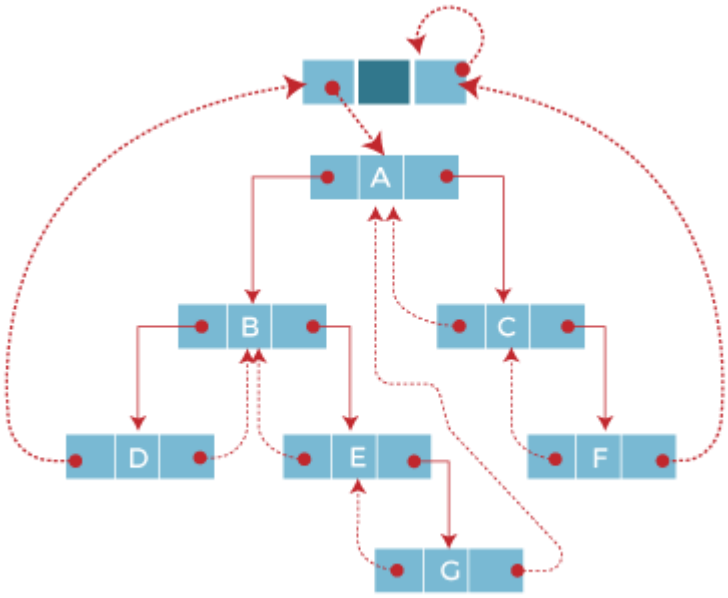


A binary tree (Inorder traversal - D, B, E, G, A, C, F)



A two - way threaded binary tree

The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F. If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B. Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor. In the same way, other nodes containing NULL values in their link fields are filled with threads.



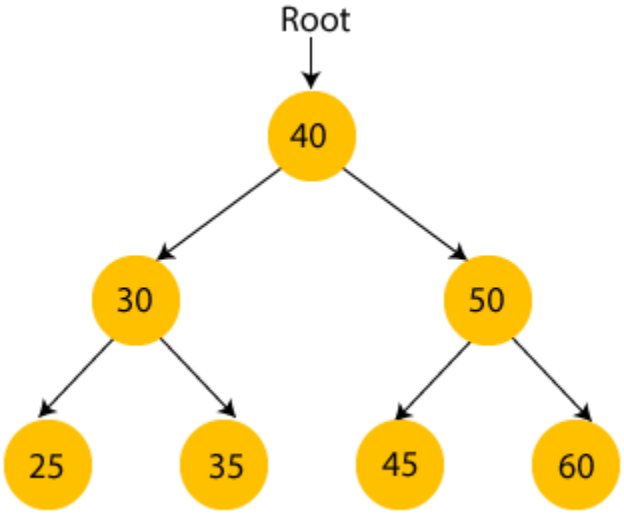
Two-way threaded - tree with header node

In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node. This is because they don't have any inorder predecessor and successor respectively. This is indicated by threads pointing nowhere. So in order to maintain the uniformity of threads, we maintain a special node called the **header node**. The header node does not contain any data part and its left link field points to the root node and its right link field points to itself. If this header node is included in the two-way threaded Binary tree then this node becomes the inorder predecessor of the first node and inorder successor of the last node. Now threads of left link fields of the first node and right link fields of the last node will point to the header node.

What is a Binary Search tree?

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.

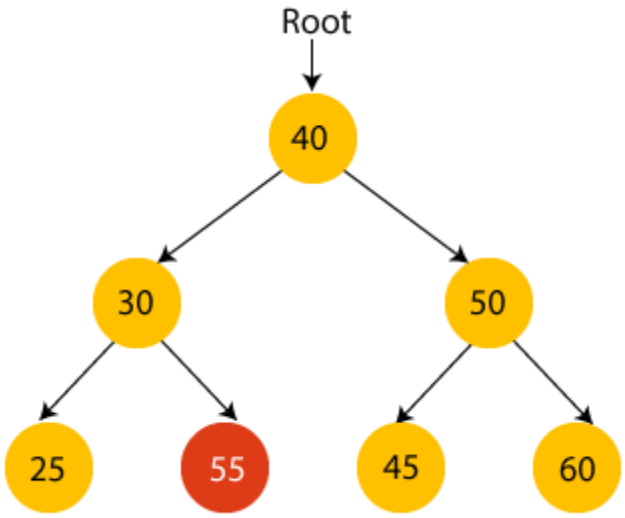
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not 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.

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**

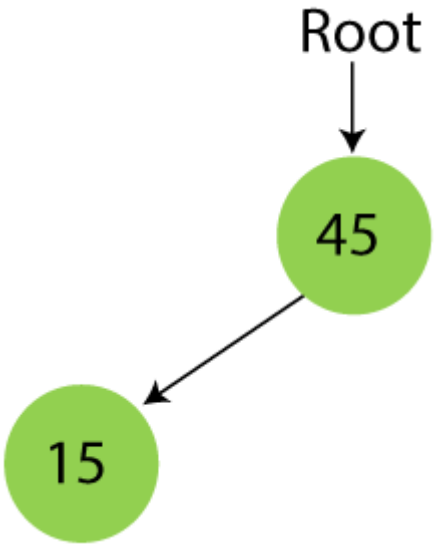
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of 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.

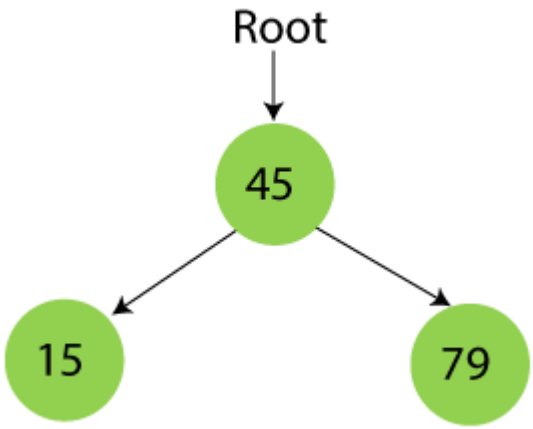
Step 2 - Insert 15.

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



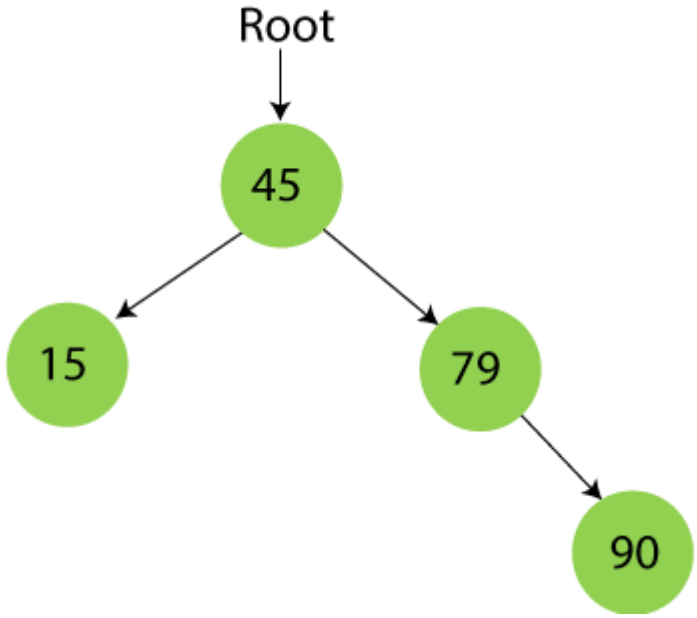
Step 3 - Insert 79.

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



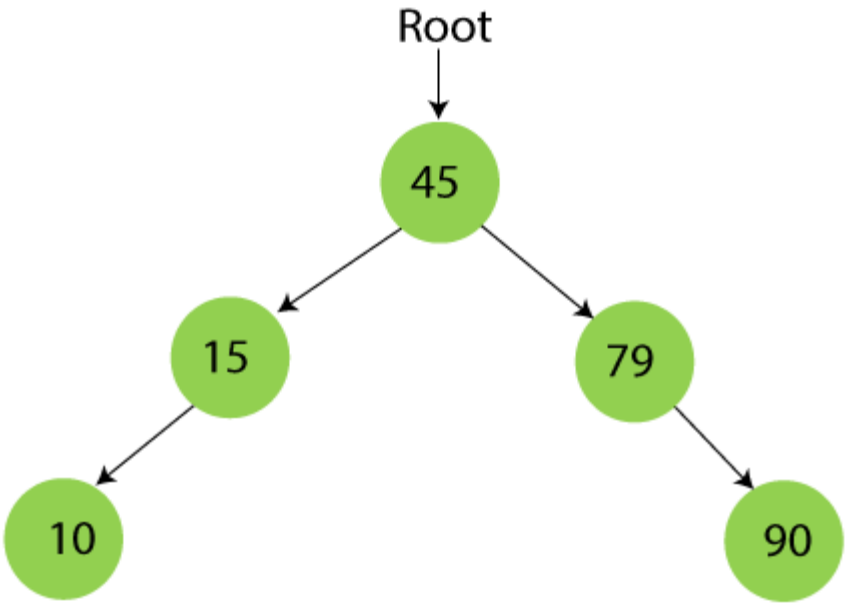
Step 4 - Insert 90.

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



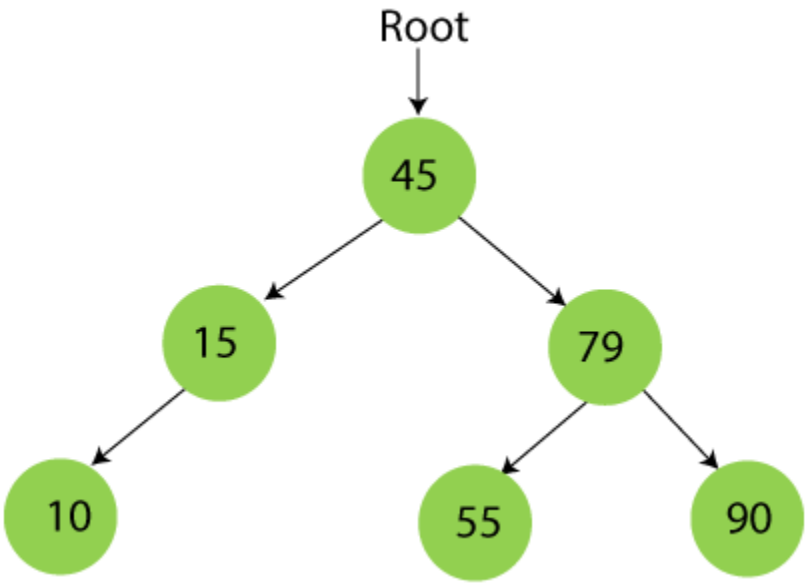
Step 5 - Insert 10.

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



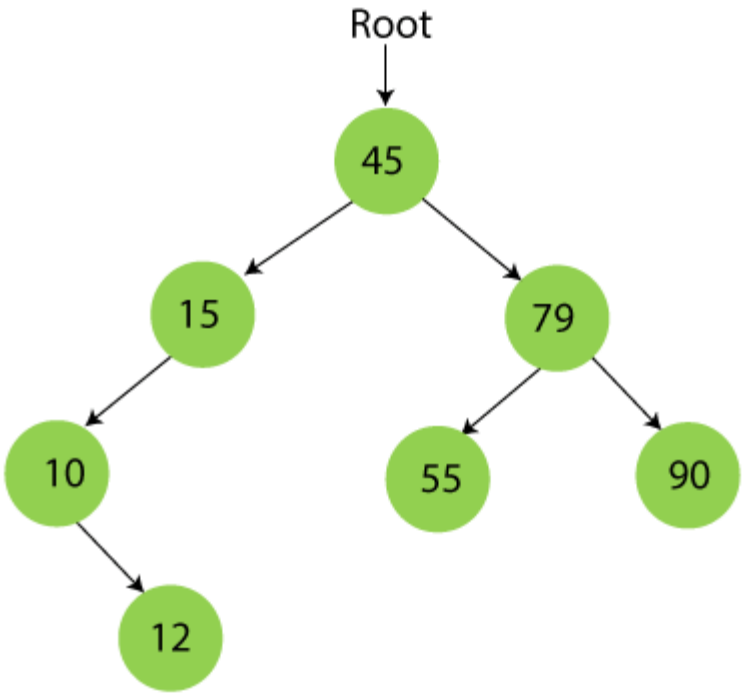
Step 6 - Insert 55.

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



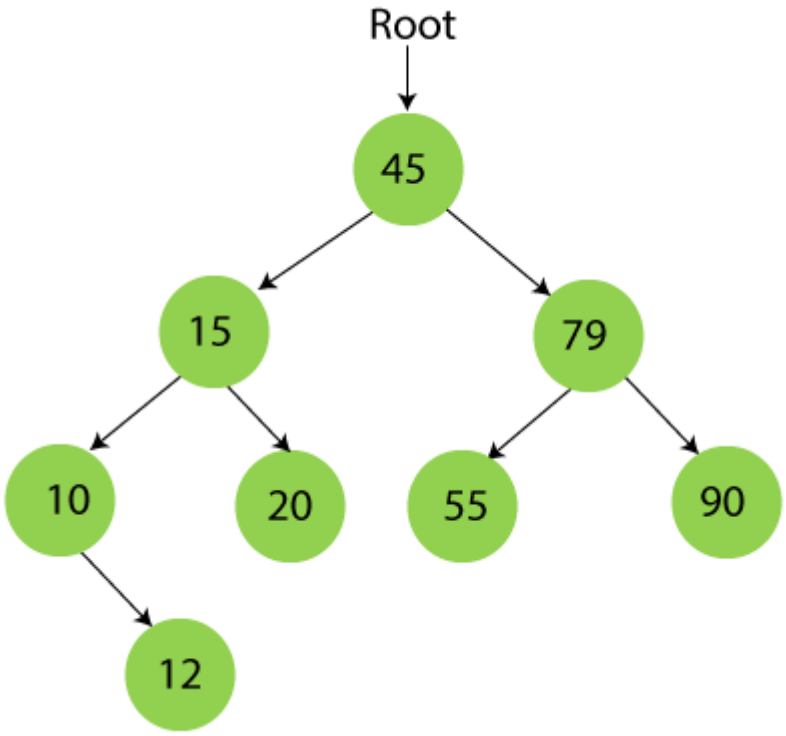
Step 7 - Insert 12.

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



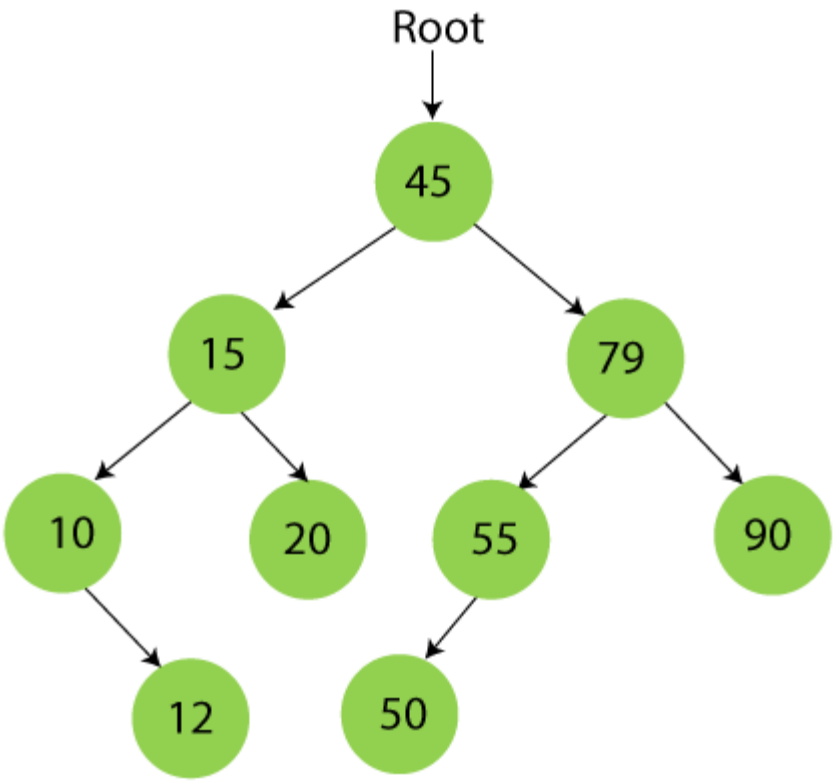
Step 8 - Insert 20.

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.



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.

Searching in Binary search tree

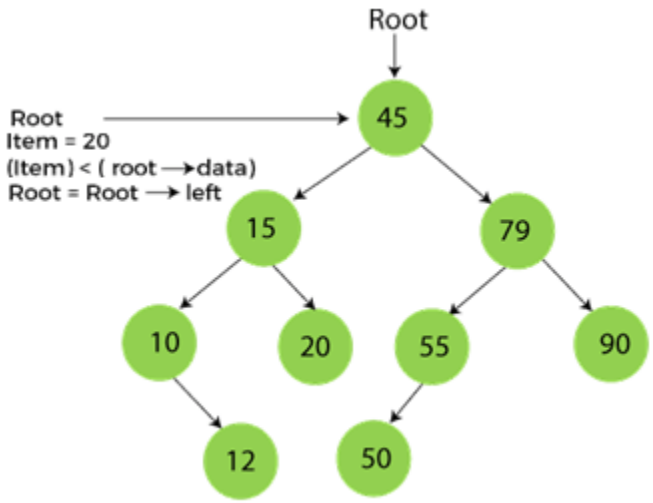
Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

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.

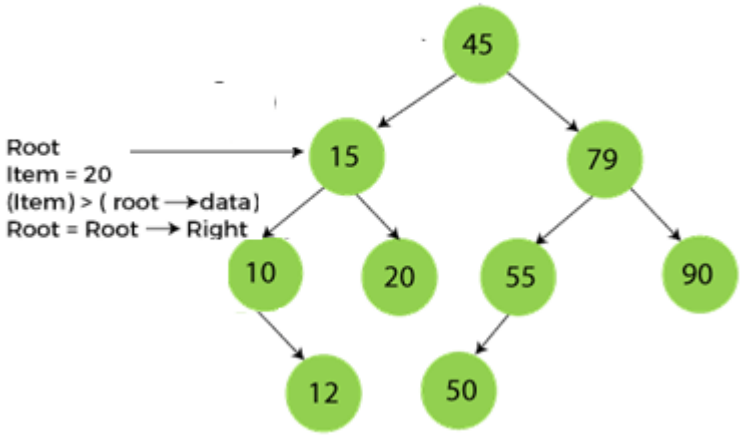
6. If the element is not found or not present in the tree, then return NULL.

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.

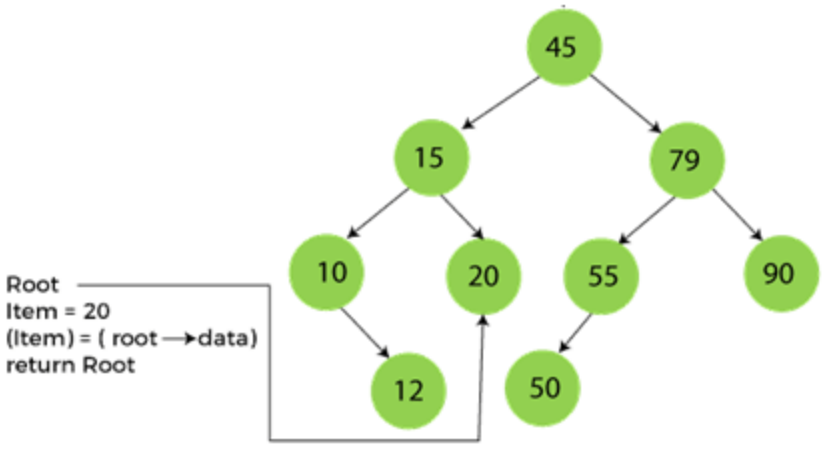
Step1:



Step2:



Step3:



Deletion in Binary Search tree

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 -

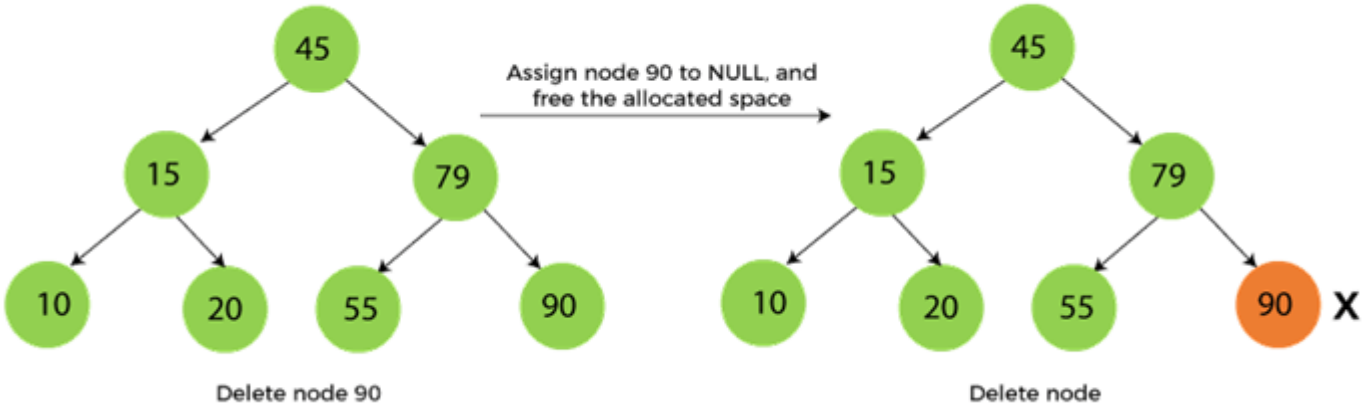
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

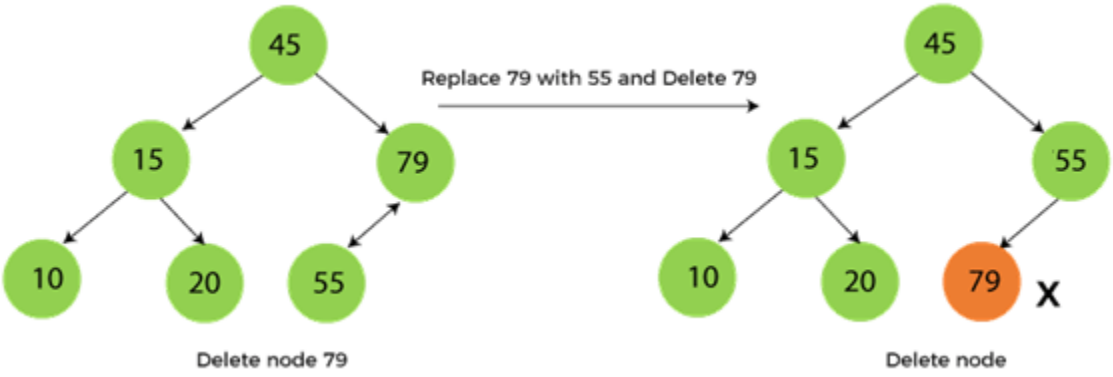


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



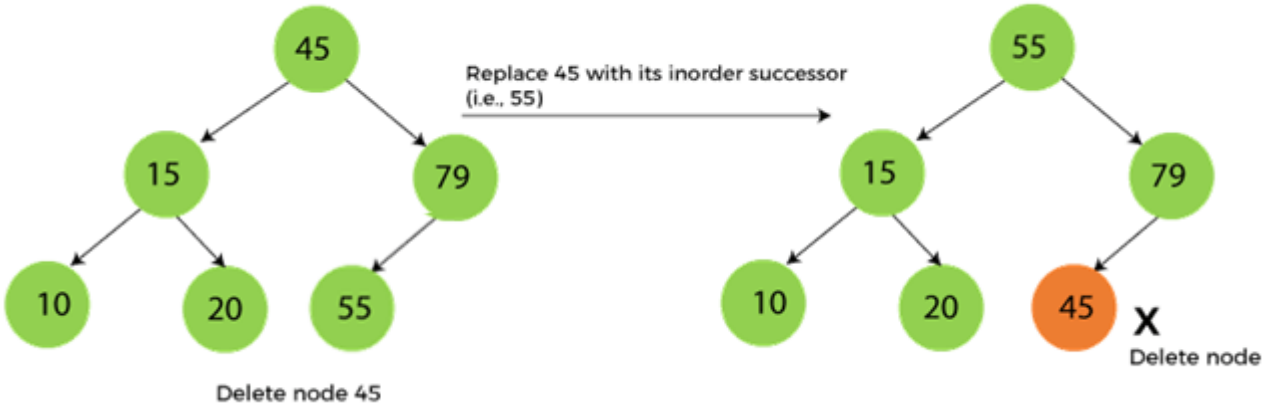
When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, 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 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.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

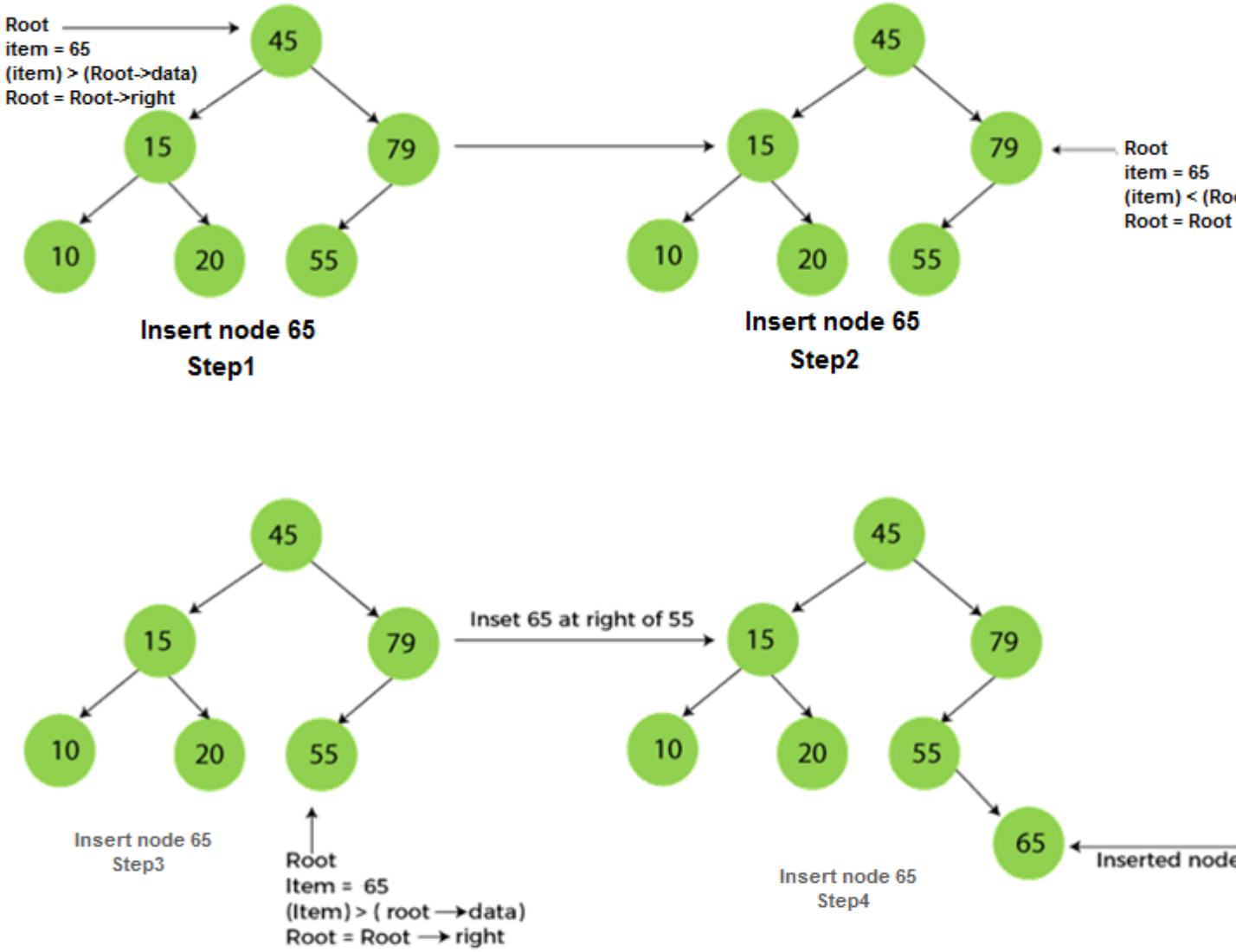


Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.



Implementation of Binary search tree

```
#include <stdio.h>

#include <stdlib.h>


// Structure for a BST node

typedef struct Node {
```

```
    int data;
    struct Node *left;
    struct Node *right;
} Node;

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

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

// Function to search for a value in the BST
```

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

// Function to find the minimum value node in the BST
Node* findMin(Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

// Function to delete a node from the BST
Node* deleteNode(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 {
        // Node with only one child or no child
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        // Node with two children: Get the inorder successor (smallest in the right subtree)
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Function to perform in-order traversal of the BST
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

```
// Main function to demonstrate BST operations
int main() {
    Node* root = NULL;

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

    printf("In-order traversal: ");
    inorder(root);
    printf("\n");

    // Search for a value
    Node* searchResult = search(root, 40);
    if (searchResult != NULL) {
        printf("Element 40 found in the BST.\n");
    } else {
        printf("Element 40 not found in the BST.\n");
    }

    // Delete a node
```

```
    root = deleteNode(root, 20);
    printf("In-order traversal after deleting 20: ");
    inorder(root);
    printf("\n");


    return 0;
}
```

Application of Trees

1. Hierarchical Data Representation

- **File Systems**: Operating systems often use trees (e.g., directory trees) to represent the structure of files and folders.
- **HTML/XML Document Object Model (DOM)**: Trees are used to represent the structure of HTML or XML documents where each node represents an element, attribute, or text.

2. Search Algorithms

- **Binary Search Trees (BST)**: Trees are used to implement binary search, which helps in searching, inserting, and deleting elements in O(logn)O(\log n)O(logn) time.
- **Balanced Trees (AVL Trees, Red-Black Trees)**: These are self-balancing trees used in maintaining sorted data and ensuring efficient search operations.

3. Routing Algorithms

- **Network Routing**: Trees are used in network routing algorithms (e.g., spanning trees) to find the shortest path or minimum spanning trees for efficient data transmission.

4. Database Indexing

- **B-Trees and B+ Trees**: Widely used in databases and file systems for indexing large amounts of data, allowing efficient search, insertion, and deletion operations.

5. Expression Parsing

- **Expression Trees**: Used in compilers and interpreters to parse expressions and perform computations. Each node in the tree represents an operation, and the children represent operands.

6. Artificial Intelligence

- **Decision Trees**: Used in machine learning for decision-making processes. They help in classification and regression tasks.

- **Minimax Algorithm**: Game trees are used in AI algorithms like Minimax, which is used for decision-making in games (e.g., chess, tic-tac-toe).

Evaluation of Expression

A. Evaluation Process

1. Order of Evaluation

Operator Precedence: Determines the order in which different types of operators are evaluated. For example, in 3 + 5 * 2, multiplication has higher precedence than addition, so 5 * 2 is evaluated first.

Operator Associativity: Defines the order in which operators of the same precedence level are evaluated. Most operators are left-associative (e.g., 3 - 2 - 1 is evaluated as (3 - 2) - 1), while some like exponentiation are right-associative.

2. Using Stacks (Expression Trees)

Infix Notation: The standard way of writing expressions (e.g., A + B). Requires consideration of operator precedence and associativity for evaluation.

Postfix Notation (Reverse Polish Notation - RPN): Operators follow their operands (e.g., A B +). Easier to evaluate using stacks since no parentheses or precedence rules are needed.

Prefix Notation (Polish Notation): Operators precede their operands (e.g., + A B). Similar to postfix, it can be easily evaluated using a stack.

B. Evaluation Algorithms

1. Using Stacks for Postfix Expressions

Traverse the postfix expression from left to right.

Push operands onto a stack.

When an operator is encountered, pop the required number of operands from the stack, apply the operator, and push the result back onto the stack.

After the entire expression has been traversed, the stack should contain one element, which is the final result.

Example:

Postfix Expression: 3 5 2 * +

Stack Operations:

Push 3

Push 5

Push 2

Apply * (Pop 5 and 2, multiply, push result 10)

Apply + (Pop 3 and 10, add, push result 13)

Final result is 13.

2. Using Expression Trees

Convert the expression into an expression tree, where each node is either an operator or an operand.

Traverse the tree (typically using post-order traversal) to evaluate the expression.

In post-order traversal, evaluate the left subtree, then the right subtree, and finally apply the operator at the current node.

C. Practical Use Cases

Calculator Programs: Simple calculators use these algorithms to evaluate arithmetic expressions entered by users.

Compilers and Interpreters: Used to evaluate expressions in programming languages during the execution of programs.

Spreadsheet Applications: Used to evaluate formulas entered by users.

Example Evaluation:

Consider the expression: 2 * (3 + 4) - 5

Convert to Postfix: 2 3 4 + * 5 -

Evaluate Postfix:

Push 2, 3, 4

Apply +: Result 7

Apply *: Result 14

Push 5

Apply -: Result 9

The final evaluated result is 9.