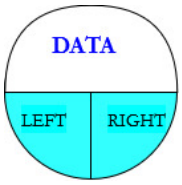
TREE

Tree

A Tree is a data structure that organizes data and represents a hierarchical relation among them. The data is stored inside a node. In other words, a tree is a collection of nodes containing data organized in a hierarchical structure. A structure is used to a node.

Creating a Node using Structure

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

Illustration of a Node


A real life tree can be divided into three areas such as root, leaves and branches. Similarly, in data structure, the nodes of a tree can be divided into three categories:

- **Root Node:** A Root Node is the first node of a tree and does not have any parent node.
- **Leaf Node:** A Leaf Node is a node that does not have any child node.
- **Intermediate Node:** An intermediate node is a node that is neither the root node nor a leaf node. It not only has a parent node but also one or more child nodes.

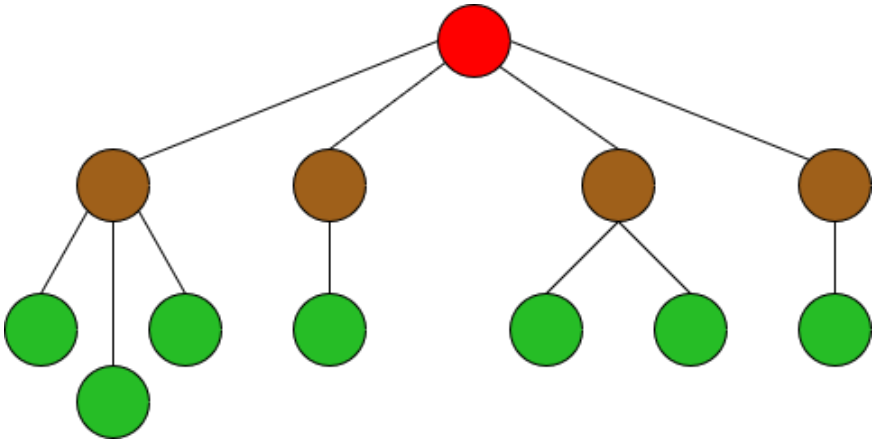


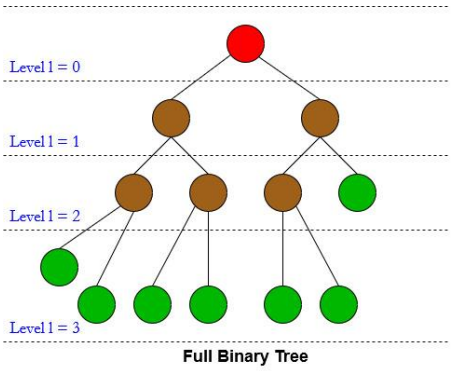
Figure: An Illustration of different type of nodes of a Tree

Based on the number of child that a node can have, we can divide a tree into two categories:

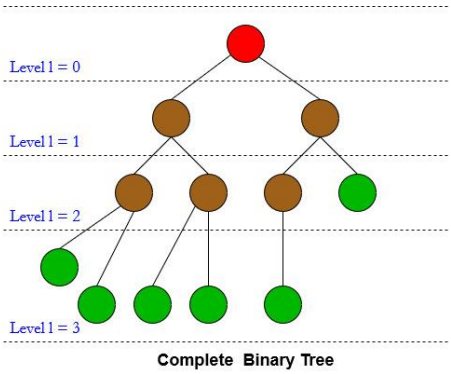
- **Binary Tree:** All the nodes of a binary tree can have a maximum of two child nodes.
- **M-ary Tree:** All the nodes of an M-ary tree can have any number of child nodes.

Binary tree can be divided into three categories:

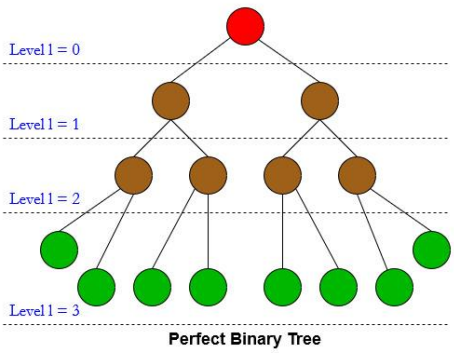
- **Full Binary Tree:** A binary tree where all the nodes apart from the leaf nodes have exactly two child node.
- **Complete Binary Tree:** A binary tree where all the nodes apart from the leaf nodes can have a maximum of two child nodes and the leaf nodes are distributed from left to right.
- **Perfect Binary Tree:** A binary tree where all the nodes apart from the leaf nodes have exactly two child nodes and the all the leaf nodes are in the same level.



Full Binary Tree



Complete Binary Tree

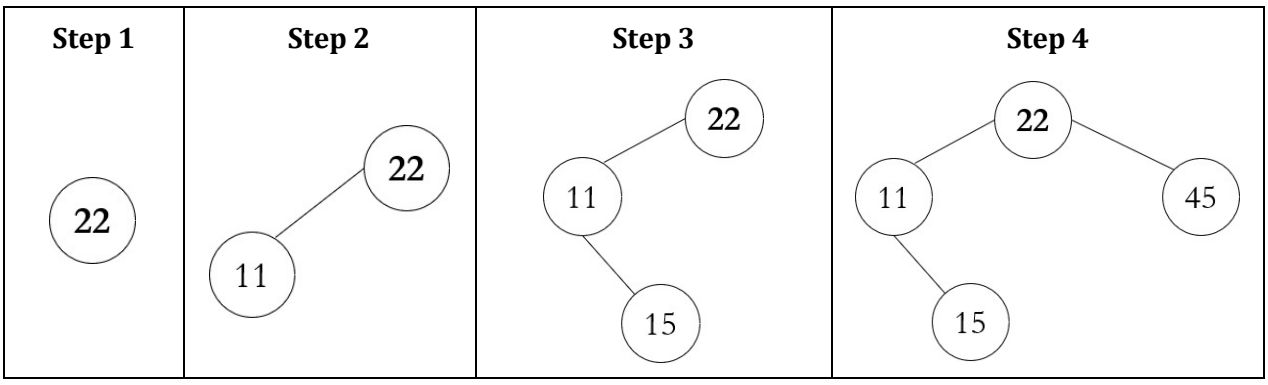


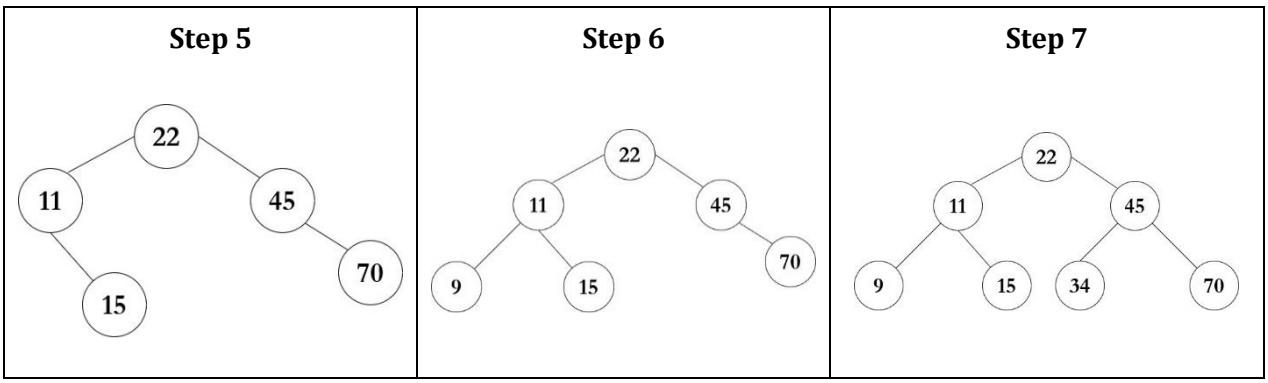
Perfect Binary Tree

Binary Search Tree: A Binary Search Tree is a tree where all the nodes in the left of the root node has smaller data than the data of root node and all the nodes in the right of the root node has larger data than the data of root node.

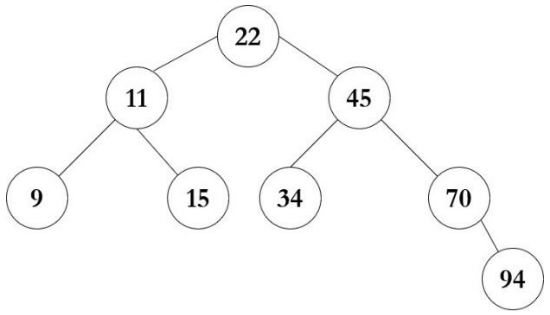
Simulation to create a BST

Data: **22, 11, 15, 45, 70, 9, 34, 94, 5, 47**

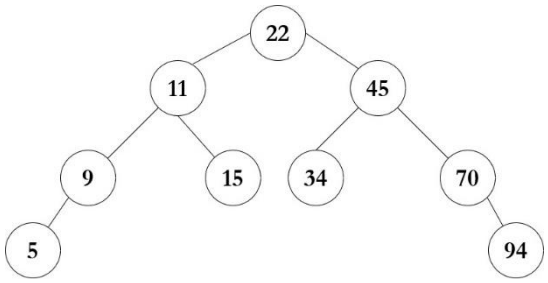




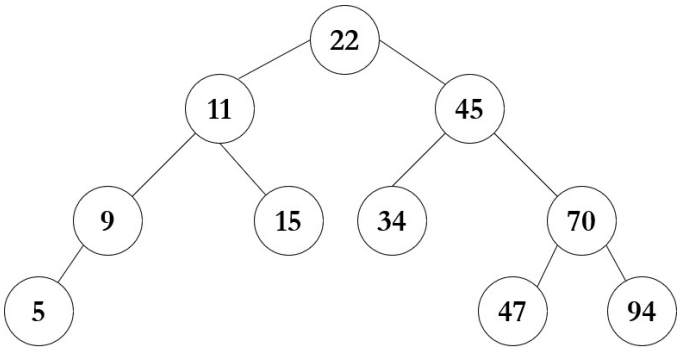
Step 8



Step 9



Step 10



newNode: A node just created
data: value of newNode.
left: a pointer to the left node of a node.
right: a pointer to the right node of a node.
root: the first node of the tree
current: a node we are working on.

Pseudocode to Create a BST

Input and Initializations: int n, struct Node, Node *Root, Node *newNode, Node *current.

Process:

1. Create a new node (***newNode***).
2. Enter the value of the ***data*** for ***newNode*** and initialize it. The ***left*** pointer for the ***newNode*** will be null. Also, the ***right*** pointer for the ***newNode*** will be null.
3. If, the value of ***root*** is null, go to (4), else go to (5).
4. The value of ***root*** will be ***newNode***. Also, the value of ***current*** will be ***newNode***. Go to (8).
5. The value of ***current*** will be ***root.***
6. If the value of ***data*** for ***newNode*** is less than the value of ***data*** for ***current*** go to (a). Else if, the value of ***data*** for ***newNode*** is greater than the value of ***data*** for ***current*** go to (b). Else, go to (c).
    a) If, the ***left*** pointer for ***current*** is not null, go to (i), else go to (ii).
        i. The value of ***current*** will be the ***left*** pointer of ***current***.
        ii. The value of ***left*** pointer of ***current*** will be ***newNode***. Go to (8).
    b) If, the ***right*** pointer for ***current*** is not null go to (i), else go to (ii).
        i. The value of ***current*** will be the ***right*** pointer of ***current***.
        ii. The value of ***right*** pointer of ***current*** will be ***newNode***. Go to (8).
    c) Else, Print "Invalid Entry". Go to (8).
7. Repeat (6) till the value of ***current*** is not null.
8. Repeat (1), (2), (3) and, (4) or (5), (6), (7) for ***n*** times and the BST is ready.

Output: Print the data of all the nodes in any of the traversal technique.

Tree Traversal:

Pre-Order:	Root – Left Child – Right Child :	**22-11-9-5-15-45-34-70-47-94**
In-Order:	Left Child – Root – Right Child :	**5-9-11-15-22-34-45-47-70-94**
Post-Order:	Left Child – Right Child – Root :	**5-9-15-11-34-47-94-70-45-22**

Pseudocode for Preorder Traversal:

Input and Initializations: A BST.

Process:

```
function preorder(Node *current)
{
   if(current != NULL)
   {
      print current->data;
      preorder(current->left);
      preorder(current->right);
   }
}
```

Pseudocode for Inorder Traversal:

Input and Initializations: A BST.

Process:

```
function inorder(Node *current)
{
   if(current != NULL)
   {
      inorder (current->left);
      print current->data;
      inorder (current->right);
   }
}
```

Pseudocode for Postorder Traversal:

Input and Initializations: A BST.

Process:

```
function postorder(Node *current)
{
   if(current != NULL)
   {
      postorder(current->left);
      postorder(current->right);
      print current->data;
   }
}*
```

Recursion for Inorder Traversal:

1st Method – Current -> 22 (Root)

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#2
        print current->data;
        inorder(current->right); //#11
    }
} //return to main method Method
```

2nd Method – Current -> 11

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#3
        print current->data;
        inorder(current->right); //#8
    }
}//return to 1st Method
```

3rd Method – Current -> 9

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#4
        print current->data;
        inorder(current->right); //#7
    }
}//return to 2nd Method
```

4th Method – Current -> 5

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#5
        print current->data;
        inorder(current->right); //#6
    }
}//return to 3rd Method
```

5th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
} //return to 4th Method
```

6th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
} //return to 4th Method
```

7th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
} //return to 3rd Method
```

8th Method – Current -> 15

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#9
        print current->data;
        inorder(current->right); //#10
    }
}//return to 2nd Method
```

9th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
}//return to 8th Method
```

10th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
}//return to 8th Method
```

11th Method – Current -> 45

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#12
        print current->data;
        inorder(current->right); //#15
    }
} //return to 1st Method
```

12th Method – Current -> 34

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#13
        print current->data;
        inorder(current->right); //#14
    }
} //return to 11th Method
```

13th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
}//return to 12th Method
```

14th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
}//return to 12th Method
```

15th Method – Current -> 70

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#16
        print current->data;
        inorder(current->right); //#19
    }
} //return to 11th Method
```

16th Method – Current -> 47

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#17
        print current->data;
        inorder(current->right); //#18
    }
} //return to 15th Method
```

17th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
} //return to 16th Method
```

18th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
} //return to 16th Method
```

19th Method – Current -> 94

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left); //#20
        print current->data;
        inorder(current->right); //#21
    }
} //return to 15th Method
```

20th Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
} //return to 19th Method
```

21st Method – Current -> NULL

```
function inorder(Node *current)
{
    if(current != NULL)
    {
        inorder(current->left);
        print current->data;
        inorder(current->right);
    }
} //return to 19th Method
```