Python - Binary Tree

Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties –

- One node is marked as Root node.
- Every node other than the root is associated with one parent node.
- Each node can have an arbiatry number of chid node.

We create a tree data structure in python by using the concept os node discussed earlier. We designate one node as root node and then add more nodes as child nodes. Below is program to create the root node.

Create Root

We just create a Node class and add assign a value to the node. This becomes tree with only a root node.

Example

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data
   def PrintTree(self):
      print(self.data)

root = Node(10)
root.PrintTree()
```

```
  10
```

Output

When the above code is executed, it produces the following result –

Inserting into a Tree

To insert into a tree we use the same node class created above and add a insert class to it. The insert class compares the value of the node to the parent node and decides to add it as a left node or a right node. Finally the PrintTree class is used to print the tree.

Example

```
   def insert(self, data):
# Compare the new value with the parent node
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
         elif data > self.data:
               if self.right is None:
                  self.right = Node(data)
               else:
                  self.right.insert(data)
      else:
         self.data = data

# Print the tree
   def PrintTree(self):
      if self.left:
         self.left.PrintTree()
      print( self.data),
      if self.right:
         self.right.PrintTree()

# Use the insert method to add nodes
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
root.PrintTree()
```

```
3
6
12
14
```

Output

When the above code is executed, it produces the following result –

```
3 6 12 14
```

Traversing a Tree

The tree can be traversed by deciding on a sequence to visit each node. As we can clearly see we can start at a node then visit the left sub-tree first and right sub-tree next. Or we can also visit the right sub-tree first and left sub-tree next. Accordingly there are different names for these tree traversal methods.

Tree Traversal Algorithms

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree.

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the In-order traversal logic is implemented by creating an empty list and adding the left node first followed by the root or parent node.

At last the left node is added to complete the In-order traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

Example

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data
```

```
# Insert Node
   def insert(self, data):
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
         else data > self.data:
            if self.right is None:
               self.right = Node(data)
            else:
               self.right.insert(data)
      else:
         self.data = data
# Print the Tree
   def PrintTree(self):
      if self.left:
         self.left.PrintTree()
      print( self.data),
      if self.right:
         self.right.PrintTree()
# Inorder traversal
# Left -> Root -> Right
   def inorderTraversal(self, root):
      res = []
      if root:
         res = self.inorderTraversal(root.left)
         res.append(root.data)
         res = res + self.inorderTraversal(root.right)
      return res
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.inorderTraversal(root))
```

Output

When the above code is executed, it produces the following result –

```
[10, 14, 19, 27, 31, 35, 42]
```

Pre-order Traversal

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

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the Pre-order traversal logic is implemented by creating an empty list and adding the root node first followed by the left node.

At last, the right node is added to complete the Pre-order traversal. Please note that, this process is repeated for each sub-tree until all the nodes are traversed.

Example

```
            elif data > self.data:
               if self.right is None:
                  self.right = Node(data)
               else:
                  self.right.insert(data)
            else:
               self.data = data
   # Print the Tree
      def PrintTree(self):
         if self.left:
            self.left.PrintTree()
         print( self.data),
         if self.right:
            self.right.PrintTree()
   # Preorder traversal
   # Root -> Left ->Right
      def PreorderTraversal(self, root):
         res = []
         if root:
            res.append(root.data)
            res = res + self.PreorderTraversal(root.left)
            res = res + self.PreorderTraversal(root.right)
         return res
   root = Node(27)
   root.insert(14)
   root.insert(35)
   root.insert(10)
   root.insert(19)
   root.insert(31)
   root.insert(42)
   print(root.PreorderTraversal(root))
```

Output

When the above code is executed, it produces the following result –

```
[27, 14, 10, 19, 35, 31, 42]
```

Post-order Traversal

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

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the Post-order traversal logic is implemented by creating an empty list and adding the left node first followed by the right node.

At last the root or parent node is added to complete the Post-order traversal. Please note that, this process is repeated for each sub-tree until all the nodes are traversed.

Example

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data
# Insert Node
   def insert(self, data):
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
         else if data > self.data:
            if self.right is None:
               self.right = Node(data)
            else:

               self.right.insert(data)
      else:
         self.data = data
# Print the Tree
   def PrintTree(self):
      if self.left:
```

```
            self.left.PrintTree()
    print( self.data),
    if self.right:
    self.right.PrintTree()
    # Postorder traversal
    # Left ->Right -> Root
    def PostorderTraversal(self, root):
    res = []
    if root:
    res = self.PostorderTraversal(root.left)
    res = res + self.PostorderTraversal(root.right)
    res.append(root.data)
    return res
    root = Node(27)
    root.insert(14)
    root.insert(35)
    root.insert(10)
    root.insert(19)
    root.insert(31)
    root.insert(42)
    print(root.PostorderTraversal(root))
```

Output

When the above code is executed, it produces the following result –

```
[10, 19, 14, 31, 42, 35, 27]
```