Binary Tree Algorithm

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

* data item

* address of left child

* address of right child

In computing, binary trees are used in two very different ways:

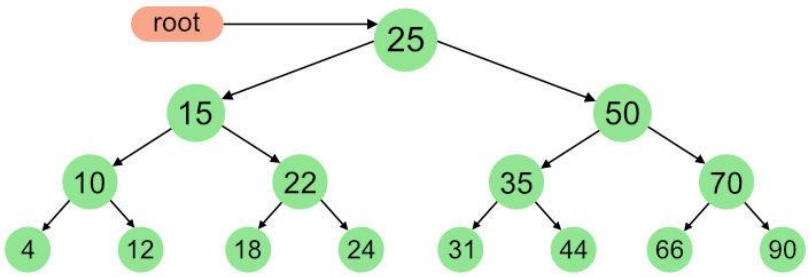
* First, as a means of accessing nodes based on some value or label associated with each node. Binary trees labelled this way are used to implement binary search trees and binary heaps, and are used for efficient searching and sorting. The designation of non-root nodes as left or right child even when there is only one child present matters in some of these applications, in particular, it is significant in binary search trees. However, the arrangement of particular nodes into the tree is not part of the conceptual information. For example, in a normal binary search tree the placement of nodes depends almost entirely on the order in which they were added, and can be re-arranged (for example by balancing) without changing the meaning.

* Second, as a representation of data with a relevant bifurcating structure. In such cases, the particular arrangement of nodes under and/or to the left or right of other nodes is part of the information (that is, changing it would change the meaning). Common examples occur with Huffman coding and cladograms. The everyday division of documents into chapters, sections, paragraphs, and so on is an analogous example with n-ary rather than binary trees.

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Node.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DataStructures.BinaryTree
{
    public class Node
    {
        public int data { get; set; }
        public Node right { get; set; }
        public Node left { get; set; }

        public Node(int nodeData)
        {
             data = nodeData;
             right = null;
             left = null;
        }
    }
}
```

BinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DataStructures.BinaryTree
{
    public class BinaryTree
    {
        public BinaryTree()
        {
            headNode = null;
            Count = 0;
        }

        public int Count { get; private set; }
        private Node headNode { get; set; }

        public Node Insert(int data)
```

```
        {
            Node newNode = new Node(data);
            if (headNode == null)
            {
                headNode = newNode;
                newNode.left = newNode.right = null;
                Count++;
                return headNode;
            }
            else
            {
                return InsertNewNode(headNode, data);
            }

        }

        private Node InsertNewNode(Node rootNode, int data)
        {
            if (rootNode == null)
            {
                rootNode = new Node(data);
                rootNode.left = rootNode.right = null;
                Count++;
            }
            else
            {
                if (data < rootNode.data )
                {
                    rootNode.left = InsertNewNode(rootNode.left,
data);
                }
                else
                {
                    rootNode.right = InsertNewNode(rootNode.right,
data);
                }
            }
            return rootNode;
        }

        public Node SearchBinaryTree(int data)
        {
            return SearchBinaryTreeRecursively(headNode, data);
        }

        private Node SearchBinaryTreeRecursively(Node parentNode, int
data)
        {
            if (parentNode == null)
            {
                return null;
            }
```

```
            if (parentNode.data.Equals(data))
            {
                return parentNode;
            }
            else
            {
                if (data < parentNode.data)
                {
                    return
SearchBinaryTreeRecursively(parentNode.left, data);
                }
                else
                {
                    return
SearchBinaryTreeRecursively(parentNode.right, data);
                }
            }
        }

        public int GetTreeDepth()
        {
            return getTreeDepth(headNode);
        }

        private int getTreeDepth(Node rootNode)
        {
            if (rootNode == null)
            {
                return 0;
            }
            else
            {
                return 1 + Math.Max(getTreeDepth(rootNode.left),
getTreeDepth(rootNode.right));
            }
        }

        /// <summary>
        /// Traverse the entire tree in InOrder routine
        /// </summary>
        /// <returns></returns>
        public List<int> GetInOrderTraversal()
        {
            return InOrderTraversal(headNode);
        }

        private List<int> InOrderTraversal(Node node)
        {
            List<int> traversedData = new List<int>();
            if (node != null)
            {
                traversedData.AddRange(InOrderTraversal(node.left));
```

```
                traversedData.Add(node.data);
                traversedData.AddRange(InOrderTraversal(node.right));
            }
            return traversedData;
        }

        public List<int> GetPreOrderTraversal()
        {
            return PreOrderTraversal(headNode);
        }

        private List<int> PreOrderTraversal(Node node)
        {
            List<int> traversedData = new List<int>();
            if (node != null)
            {
                traversedData.Add(node.data);
                traversedData.AddRange(PreOrderTraversal(node.left));
                traversedData.AddRange(PreOrderTraversal(node.right));
            }
            return traversedData;
        }

        public List<int> GetPostOrderTraversal()
        {
            return PostOrderTraversal(headNode);
        }

        private List<int> PostOrderTraversal(Node node)
        {
            List<int> traversedNode = new List<int>();
            if (node != null)
            {
                traversedNode.AddRange(PostOrderTraversal(node.left));

traversedNode.AddRange(PostOrderTraversal(node.right));
                traversedNode.Add(node.data);
            }

            return traversedNode;
        }
    }
}
```