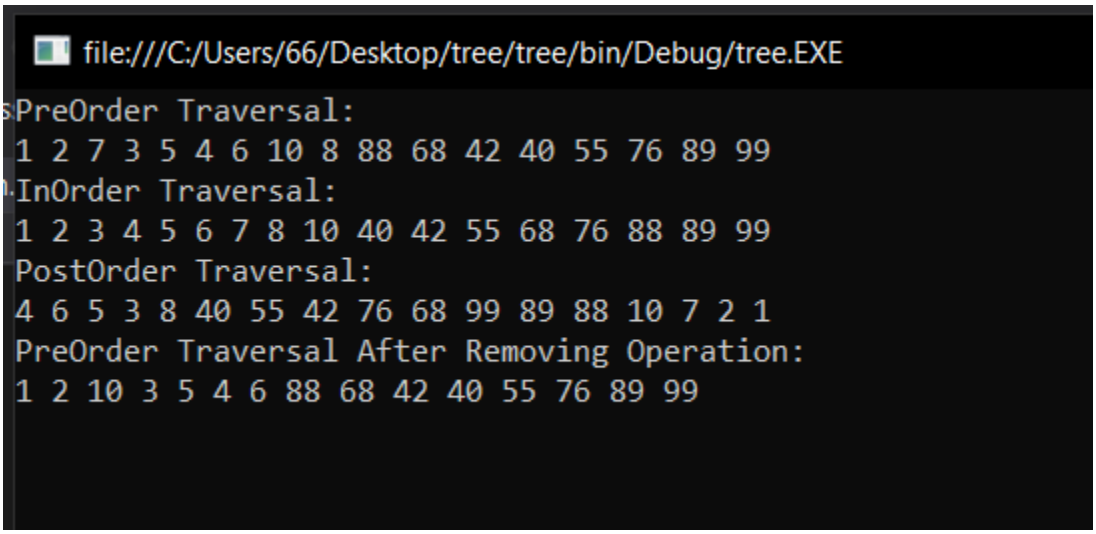
Program Binary Tree



Source Code

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

namespace tree
{
    class Program
    {
        static void Main(string[] args)
        {
            BinaryTree_of_Nickzaza Nick_BinaryTree = new BinaryTree_of_Nickzaza()
;


            Nick_BinaryTree.Add(1);
            Nick_BinaryTree.Add(2);
            Nick_BinaryTree.Add(7);
            Nick_BinaryTree.Add(3);
            Nick_BinaryTree.Add(10);
            Nick_BinaryTree.Add(5);
            Nick_BinaryTree.Add(8);
            Nick_BinaryTree.Add(8);
            Nick_BinaryTree.Add(88);
            Nick_BinaryTree.Add(89);
            Nick_BinaryTree.Add(68);
            Nick_BinaryTree.Add(76);
            Nick_BinaryTree.Add(42);
            Nick_BinaryTree.Add(55);
            Nick_BinaryTree.Add(40);
            Nick_BinaryTree.Add(99);
            Nick_BinaryTree.Add(4);
            Nick_BinaryTree.Add(6);


            Node_of_Nick node = Nick_BinaryTree.Find(5);
            int depth = Nick_BinaryTree.GetTreeDepth();

            Console.WriteLine("PreOrder Traversal:");
            Nick_BinaryTree.TraversePreOrder(Nick_BinaryTree.Root);
            Console.WriteLine();
```

```
            Console.WriteLine("InOrder Traversal:");
            Nick_BinaryTree.TraverseInOrder(Nick_BinaryTree.Root);
            Console.WriteLine();

            Console.WriteLine("PostOrder Traversal:");
            Nick_BinaryTree.TraversePostOrder(Nick_BinaryTree.Root);
            Console.WriteLine();

            Nick_BinaryTree.Remove(7);
            Nick_BinaryTree.Remove(8);

            Console.WriteLine("PreOrder Traversal After Removing Operation:");
            Nick_BinaryTree.TraversePreOrder(Nick_BinaryTree.Root);
            Console.WriteLine();

            Console.ReadLine();
        }
    }

    class Node_of_Nick
    {
        public Node_of_Nick LeftNode { get; set; }
        public Node_of_Nick RightNode { get; set; }
        public int Data { get; set; }
    }

    class BinaryTree_of_Nickzaza
    {
        public Node_of_Nick Root { get; set; }

        public bool Add(int value)
        {
            Node_of_Nick before = null, after = this.Root;

            while (after != null)
            {
                before = after;
                if (value < after.Data) //Is new node in left tree?
                    after = after.LeftNode;
                else if (value > after.Data) //Is new node in right tree?
                    after = after.RightNode;
                else
                {
                    //Exist same value
```

```
                return false;
            }
        }

        Node_of_Nick newNode = new Node_of_Nick();
        newNode.Data = value;

        if (this.Root == null)//Tree ise empty
            this.Root = newNode;
        else
        {
            if (value < before.Data)
                before.LeftNode = newNode;
            else
                before.RightNode = newNode;
        }

        return true;
    }

    public Node_of_Nick Find(int value)
    {
        return this.Find(value, this.Root);
    }

    public void Remove(int value)
    {
        Remove(this.Root, value);
    }

    private Node_of_Nick Remove(Node_of_Nick parent, int key)
    {
        if (parent == null) return parent;

        if (key < parent.Data) parent.LeftNode = Remove(parent.LeftNode, key)
;
        else if (key > parent.Data)
            parent.RightNode = Remove(parent.RightNode, key);

        // if value is same as parent's value, then this is the node to be de
leted
        else
        {
            // node with only one child or no child
            if (parent.LeftNode == null)
```

```
                return parent.RightNode;
            else if (parent.RightNode == null)
                return parent.LeftNode;

            // node with two children: Get the inorder successor (smallest in
the right subtree)
            parent.Data = MinValue(parent.RightNode);

            // Delete the inorder successor
            parent.RightNode = Remove(parent.RightNode, parent.Data);
        }

        return parent;
    }

    private int MinValue(Node_of_Nick node)
    {
        int minv = node.Data;

        while (node.LeftNode != null)
        {
            minv = node.LeftNode.Data;
            node = node.LeftNode;
        }

        return minv;
    }

    private Node_of_Nick Find(int value, Node_of_Nick parent)
    {
        if (parent != null)
        {
            if (value == parent.Data) return parent;
            if (value < parent.Data)
                return Find(value, parent.LeftNode);
            else
                return Find(value, parent.RightNode);
        }

        return null;
    }

    public int GetTreeDepth()
    {
        return this.GetTreeDepth(this.Root);
```

```
        }

        private int GetTreeDepth(Node_of_Nick parent)
        {
            return parent == null ? 0 : Math.Max(GetTreeDepth(parent.LeftNode), G
etTreeDepth(parent.RightNode)) + 1;
        }

        public void TraversePreOrder(Node_of_Nick parent)
        {
            if (parent != null)
            {
                Console.Write(parent.Data + " ");
                TraversePreOrder(parent.LeftNode);
                TraversePreOrder(parent.RightNode);
            }
        }

        public void TraverseInOrder(Node_of_Nick parent)
        {
            if (parent != null)
            {
                TraverseInOrder(parent.LeftNode);
                Console.Write(parent.Data + " ");
                TraverseInOrder(parent.RightNode);
            }
        }

        public void TraversePostOrder(Node_of_Nick parent)
        {
            if (parent != null)
            {
                TraversePostOrder(parent.LeftNode);
                TraversePostOrder(parent.RightNode);
                Console.Write(parent.Data + " ");
            }
        }
    }
}
```